Informatik II

Introduction, Sorting, Recursion
— SL01 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

TABLE OF CONTENTS — SL01

1. Introduction
   Introduction

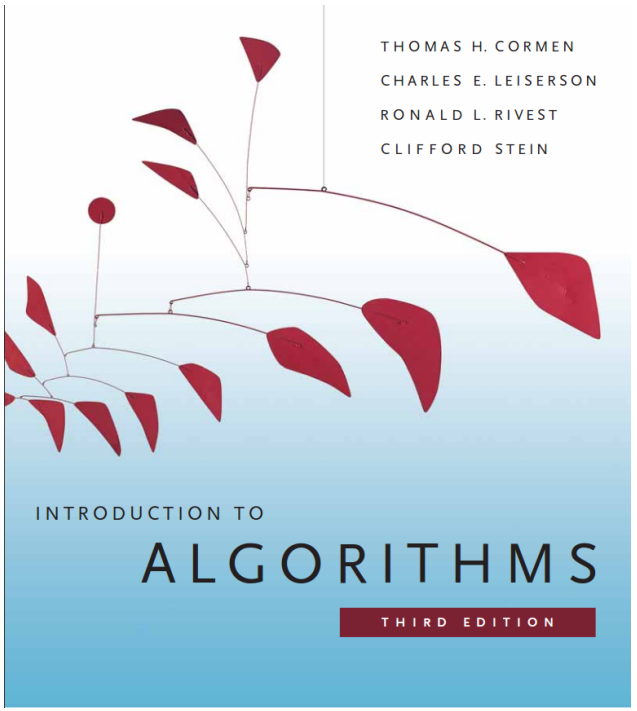
2. Algorithms

3. Sorting

4. Recursion

LITERATURE

- T. Cormen, C. Leiserson, R. Rivest and C. Stein (CLRS), Introduction to Algorithms, Third Edition, MIT Press, 2009.
- Available in IfI library (two books on display; 6 books for borrowing)
- Available as ebook from inside the UZH network:



`http://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=343613&site=ehost-live`

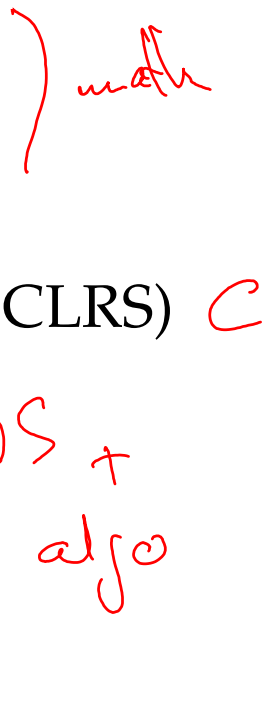
GOALS OF THIS COURSE

The main things we will learn in this course:

- To **think algorithmically** and get the spirit of how algorithms are designed.
- To get to know a **toolbox** of classical algorithms.
- To learn a number of algorithm **design techniques** (such as divide-and-conquer).
- To reason in a precise way about the **efficiency** and the **correctness** of algorithms.

SYLLABUS

1. **Introduction, basic sorting, recursion** (chap 1 in CLRS)
2. **Complexity and correctness** (chap 2, 3 in CLRS)
3. **Divide and conquer, recurrences** (chap 4 in CLRS)
4. **Heap sort, quick sort** (chap 6, 7 in CLRS)
5. **Pointers, lists, sets, abstract data types** (chap 10 in CLRS)
6. **Trees, red-black trees** (chap 12, 13 in CLRS)
7. **Hash tables** (chap 11 in CLRS)
8. **Dynamic programming** (chap 15 in CLRS)
9. **Graph algorithms** (chap 22, 23, 24 in CLRS)



ADMINISTRATION/1



Home page

- `http://www.ifi.uzh.ch/dbtg`
- `https://lms.uzh.ch/url/RepositoryEntry/16546169084`
- Check OLAT frequently

Course book

- Introduction to Algorithms, 3rd edition, Cormen et al.

Assistant

- Muhammad Saad, `saad@ifi.uzh.ch`

TA:

- Luka Popovic, `luka.popovic@uzh.ch`

ADMINISTRATION/2

Lectures

- Monday 14:00 - 15:45 in HAH-E-11 (Häldeliweg 2)
- Thursday 12:15 - 13:45 in Y15-G-40 (Irchel)
- First lecture: Monday 18.2.2019
- Last lecture: Thursday 23.5.2019
- Exceptions are the following dates:
  - TH 7.3.2019 12:15 - 13:45 no lecture
  - TH 4.4.2019 12:15 - 13:45 Lecture in Y04-G-30
  - MO 8.4.2019 14:00 - 15:45 no lecture (Sechseläuten)
  - MO 22.4.2019 14:00 - 15:45 no lecture (Spring break)
  - TH 24.4.2019 12:15 - 13:45 no lecture (Spring break)

ADMINISTRATION/3

Exercises:

- There are 12 weekly exercises
- An exercise is published one week before it is being discussed in the exercise classes.
- There is no hand in of exercises.
- Attending exercises is not mandatory.
- It is strongly recommended that you solve the exercise before the exercise class.

ADMINISTRATION/3

Exercise classes:

- Tuesday 16:15 - 17:45 and 18:15 - 19:45, BIN-0.B.06
- Tuesday 16:15 - 17:45 and 18:15 - 19:45, BIN-0.B.04
- Wednesday 16:15 - 17:45 and 18:15 - 19:45, BIN-0.B.06
- Thursday 16:15 - 17:45 and 18:15 - 19:45, BIN-0.B.06
- Exceptions:
  - The labs from 18.4 (16:15-17:45 and 18:15-19:45) take place 24 hours earlier in room 0.B.04.
  - The labs from 1.5 (16:15-17:45 and 18:15-19:45) take place 24 hours later in room 0.B.04.

Spring Break

Tutors:

- Michael Blum, `michael.blum2@uzh.ch`
- Dimitri Degkwitz, `dimitri.degkwitz@uzh.ch`
- Johann Schwabe, `johann.schwabe@uzh.ch`
- Christoph Vogel, `christoph.vogel@uzh.ch`

ADMINISTRATION/4

Exams: The course assessment consists of two midterms and a final exam.

Exam dates (check VVZ and official WWF web pages):

- Midterm 1 (90min): Friday, 22.3.2019, 12:15 - 13:45, HAH-E-03 (Häldeliweg 2).
- Midterm 2 (90min): Friday, 3.5.2019, 12:15 - 13:45, HAH-E-03 (Häldeliweg 2)
- Final exam date: Monday 27.5.2019, 14:00 - 15:30

Your final grade is calculated as follows:

```
MAX(MidTerm1, MidTerm2) * 0.25 + FinalExam * 0.75
```

General remarks/1

- Enrollment in the labs takes place in OLAT in the section "Team Enrollment". Please make sure you enroll as soon as possible. Deadline: Friday, February 23 at 23:00.
- In case of a justifiable absence in a midterm, please provide proof of your absence (e.g. document signed by the doctor, military supervisor) to Muhammad Saad as soon as possible.
- One A4 sheet with your personal notes (both sides) is allowed in all Informatics II exams.

General remarks/2

- **Hands-on exercises** are an important part of this course: an abstract understanding of the concepts is not good enough
- You must be able to **apply** your knowledge to **new** examples. Use exercises to practice this during the semester.
- Often it is most effective to first solve algorithms on paper and later key them in on the computer. no trial and error
- A very important thing is to be **simple** and **precise**.

General remarks/3

- During lectures:
  - Interaction is welcome; ask questions.
  - Speed up/slow down the progress.
  - Additional explanations if desired.
- During the lectures we solve many examples. Please participate and take notes. Trying, failing and improving is good. Not trying is bad.
- This script is designed as a **working script**. Solutions to selected exercises are included at the end.

PREREQUISITES

Introduction to programming (aka Informatik I)

- Data types, operations
- Conditional statements if
- Loops while
- Procedures and functions

Ability to edit, compile, and execute a program

- editor and terminal
- Xcode, Eclipse or equivalent

SHELL

xterm

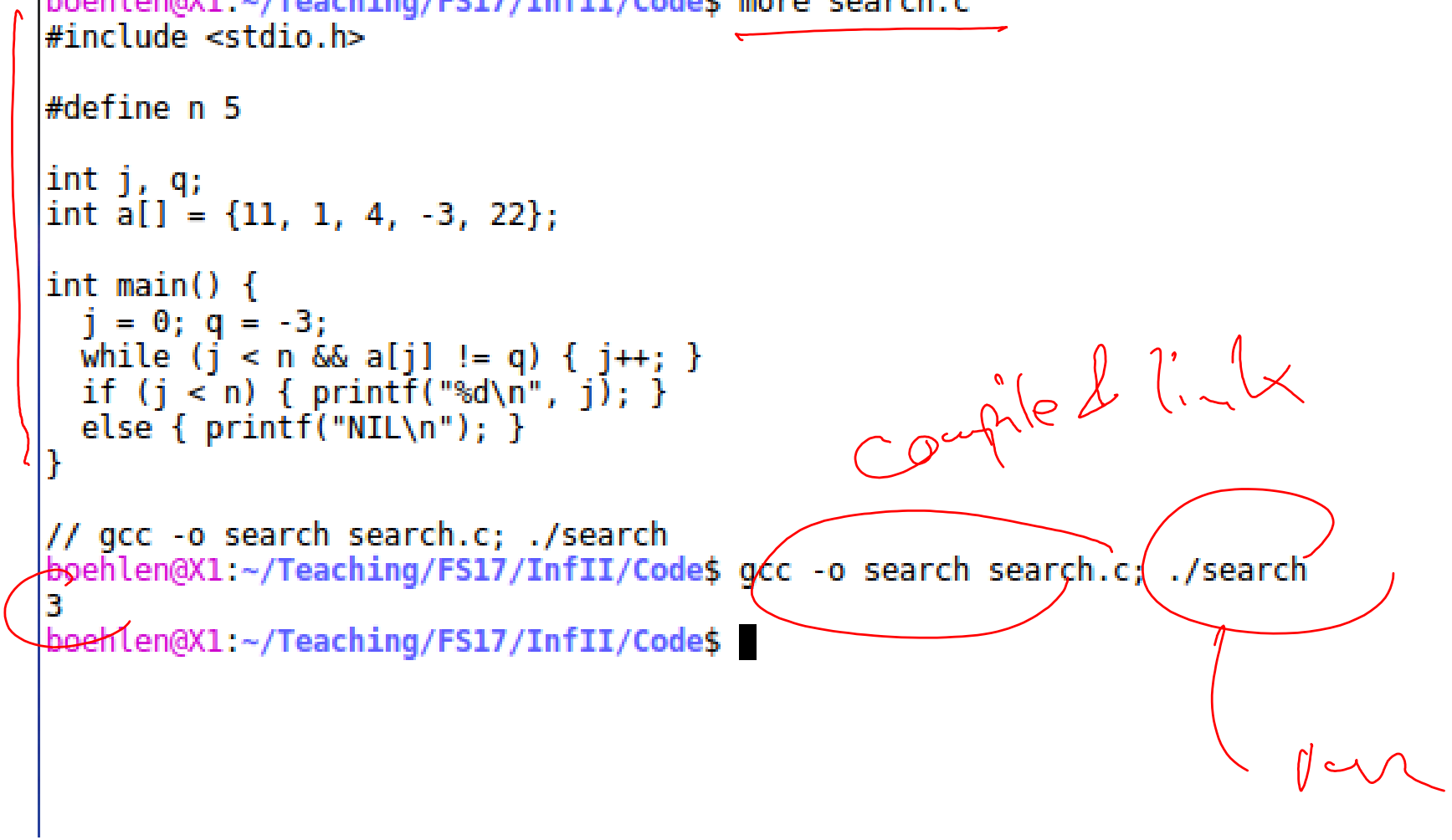
```
boehlen@X1:~/Teaching/FS17/InfII/Code$ more search.c
#include <stdio.h>

#define n 5

int j, q;
int a[] = {11, 1, 4, -3, 22};

int main() {
  j = 0; q = -3;
  while (j < n && a[j] != q) { j++; }
  if (j < n) { printf("%d\n", j); }
  else { printf("NIL\n"); }
}

// gcc -o search search.c; ./search
boehlen@X1:~/Teaching/FS17/InfII/Code$ gcc -o search search.c; ./search
3
boehlen@X1:~/Teaching/FS17/InfII/Code$
```



Acknowledgments

- The slides are based on the textbook Introduction to Algorithms from T. Cormen, C. Leiserson, R. Rivest and C. Stein.
- A very early version of these slides was developed by Simonas Saltenis from Aalborg University, Denmark.
- The course is based on a course that I taught at the Free University of Bolzano, Italy.
- Kurt Ranaltar created the initial Latex version of these slides.

TABLE OF CONTENTS — SL01

1. Introduction

2. Algorithms
   Algorithms
   Data structures

3. Sorting

4. Recursion

What is an algorithm?

- Algorithms are about solving problems
  - Enrollment at UZH
  - Booking a module
  - Graduate from IfI@UZH
  - Get me from home to work
  - Earn lots of money
  - Simulate a jet engine
  - Human genome project: algorithms to store and analyze human DNA
  - Electronic commerce: algorithms for public-key cryptography and digital signatures
  - Internet/WWW: algorithms to determine good routes for data and to quickly find data

What is an algorithm?

- Algorithms are about solving problems
  - Enrollment at UZH
  - Booking a module
  - Graduate from IfI@UZH
  - Get me from home to work
  - Earn lots of money
  - Simulate a jet engine
  - Human genome project: algorithms to store and analyze human DNA
  - Electronic commerce: algorithms for public-key cryptography and digital signatures
  - Internet/WWW: algorithms to determine good routes for data and to quickly find data
- Algorithms = procedures, recipes, process descriptions

History

- Etymology: from Persian mathematician al-Khwarizmi
- 400-300 B.C.
  - First algorithm: Euclidean algorithm, greatest common divisor
- 19th century
  - Charles Babbage (first mechanical computer)
  - Ada Lovelace (first computer program)
- 20th century
  - Alan Turing (Turing machine, cryptography)
  - Alonzo Church (decideability)
  - John von Neumann (Von Neumann architecture)

(data & program are together)

TERMINOLOGY

- **Data structure**
  - Organization of data to solve problem at hand
  - Solutions depend on the choice of data structure

TERMINOLOGY

- **Data structure**
  - Organization of data to solve problem at hand
  - Solutions depend on the choice of data structure
- **Algorithm**
  - Finite sequence of unambiguous instructions
  - Step-by-step outline of computational procedure
  - Independent from choice of programming language

TERMINOLOGY

- **Data structure**
  - Organization of data to solve problem at hand
  - Solutions depend on the choice of data structure
- **Algorithm**
  - Finite sequence of unambiguous instructions
  - Step-by-step outline of computational procedure
  - Independent from choice of programming language
- **Program**
  - Implementation of given computational procedure
  - Programming language, software engineering issues

Overall picture / 1

- Specification
  - Precisely specify problem

Overall picture/1

- Specification
  - Precisely specify problem
- Design
  - Specify structure/blocks of solution
  - Develop algorithms (often pseudocode procedures)
  - Goals: correctness & efficiency

Overall picture/1

- Specification
  - Precisely specify problem
- Design
  - Specify structure/blocks of solution
  - Develop algorithms (often pseudocode procedures)
  - Goals: correctness & efficiency
- Development
  - Implement algorithm in some language
  - Goals: robustness, reusability, adaptability

Overall picture/1

understand what must be done in all cases

- Specification
  - Precisely specify problem
- Design
  - Specify structure/blocks of solution
  - Develop algorithms (often pseudocode procedures)
  - Goals: correctness & efficiency
- Development
  - Implement algorithm in some language
  - Goals: robustness, reusability, adaptability
- Testing
  - Verify that implementation meets specification

Overall picture/2

What AlgoDat is **not** about

- Software architecture
- Computer architecture
- Programming languages
- Software engineering

Overall picture/2

What AlgoDat is **not** about

- Software architecture
- Computer architecture
- Programming languages
- Software engineering

Other related topics that we touch upon

- Computability and complexity
- Efficiency vs NP-completeness

ALGORITHMS/1

An **algorithm** is

- Any well-defined computational procedure that
  - takes some value, or set of values, as input
  - produces some value, or set of values, as output

  i.e., an unambiguous sequence of computational steps that transform the input into the output

ALGORITHMS/1

An **algorithm** is

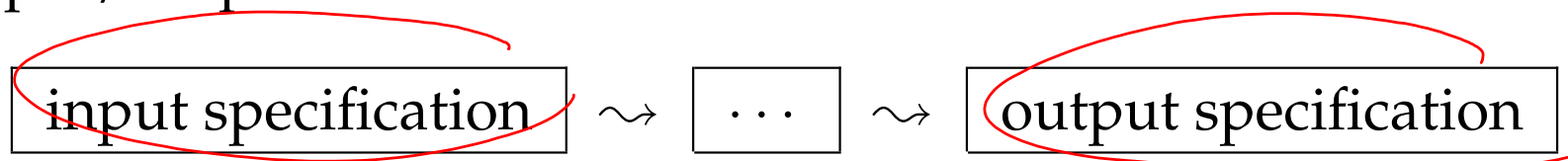
- Any well-defined computational procedure that
  - takes some value, or set of values, as input
  - produces some value, or set of values, as output

  i.e., an unambiguous sequence of computational steps that transform the input into the output
- A tool for solving a well-specified computational problem
  - Problem statement specifies the desired input/output relation
  - Algorithm describes a procedure for achieving such a relation

ALGORITHMS/2

Algorithmic problem

Input/output relation



Specification of output as function of specification of input

ALGORITHMS/2

Algorithmic problem

Input/output relation

input specification $\rightsquigarrow$ $\cdots$ $\rightsquigarrow$ output specification

Specification of output as function of specification of input

Algorithmic solution

Input/output relation

input instance $\rightsquigarrow$ algorithm $\rightsquigarrow$ output instance

Procedure transforming input instance into output instance

How to develop an algorithm

1. Understanding the problem
   - Precisely define the problem.
   - Precisely specify the input and output.
   - Construct the simplest possible representative example for the problem.
   - Consider all cases.

How to develop an algorithm

1. Understanding the problem
   - Precisely define the problem.
   - Precisely specify the input and output.
   - Construct the simplest possible representative example for the problem.
   - Consider all cases.
2. Come up with a simple plan to solve the problem at hand.
   - The plan is language independent.
   - The precise problem specification influences the plan.

How to develop an algorithm

1. Understanding the problem
   - Precisely define the problem.
   - Precisely specify the input and output.
   - Construct the simplest possible representative example for the problem.
   - Consider all cases.
2. Come up with a simple plan to solve the problem at hand.
   - The plan is language independent.
   - The precise problem specification influences the plan.
3. Turn the plan into an implementation
   - The problem representation (data structure) influences the implementation.

EXAMPLE OF AN ALGORITHM/1

- Searching problem
  - Input: a sequence (array) of n numbers $A = [a_1, a_2, \ldots, a_n]$ and a value v
  - Output: an index i such that $v = a_i (= A[i])$ or the special value `NIL` if v does not appear in A

EXAMPLE OF AN ALGORITHM/1

- Searching problem
  - Input: a sequence (array) of n numbers $A = [a_1, a_2, \ldots, a_n]$ and a value v
  - Output: an index i such that $v = a_i (= A[i])$ or the special value `NIL` if v does not appear in A

'is this precise?

- Instance of searching (example)
  - Input: $A = [31, 59, 26, 41, 58], v = 59$
  - Output: $i = 2$

$A = [1, 1], v = 1$

$i = ?$

EXAMPLE OF AN ALGORITHM/2

Example: linear search

- Input:
  sequence of numbers $A = [a_1, a_2, \ldots, a_n]$ and value v
- Output:
  index i such that $v == A[i]$ or NIL if v does not appear in A

EXAMPLE OF AN ALGORITHM/2

Example: linear search

- Input:
  sequence of numbers $A = [a_1, a_2, \ldots, a_n]$ and value v
- Output:
  index i such that $v == A[i]$ or NIL if v does not appear in A
- Algorithms

Algo: LinSearch1(A,v)

```
p = NIL;
for i = 1 to n do
    if A[i]==v then p = i;
return p;
```

gives index of last element that is = v

Algo: LinSearch2(A,v)

```
i = 1;
while i≤n ∧ A[i]≠v do i++;
if i≤n then return i;
else return NIL;
```

gives the index of first element that is = v

Example of an algorithm/3

Comparing algorithms

- Assume that $A = [31, 41, 59, 26, 41, 58]$ and $v = 41$ Then:
  - `LinSearch1`$(A, v) = 5$
  - `LinSearch2`$(A, v) = 2$
  - `LinSearch1`$(A, v) \neq$ `LinSearch2`(A, v)
- Ambiguous problem statement: first or last index?
- There is always more than one solution to a given problem.

Example of an algorithm/4

Fundamental properties

- Efficiency of an algorithm
  - `LinSearch2` is more efficient in general
  - `LinSearch1` always scans the entire array

EXAMPLE OF AN ALGORITHM/4

Fundamental properties

- Efficiency of an algorithm
  - `LinSearch2` is more efficient in general
  - `LinSearch1` always scans the entire array
- Correctness of an algorithm

Algo: LinSearch3(A,v)

```
i = 1;
while i≤n ∨ A[i]≠v do
    if i>n then break else i = i+1;
if i≤n then return i else return NIL;
```

at end $i = n+1$

$\Rightarrow A[n+1]$ does not exists

- `LinSearch3` is incorrect; `LinSearch3` crashes for all inputs.

why?

EXAMPLE OF AN ALGORITHM/5

- What about the fourth solution for searching?
- Run through the array and return the index of the value in the array.

Algo: LinSearch4(A,v)

```
for i = 1 to n do
    if A[i]==v then return i;
return NIL;
```

- OK?

yes equivalent to LinSearch2

Example of an algorithm/6

Metaphor: shopping behavior when buying a beer.

- `LinSearch1`: scan products until you get to the exit; if during the process you find a beer put it into the basket (and remove the rest).
- `LinSearch2`: scan products; stop as soon as a beer is found and go to the exit.
- `LinSearch4`: scan products; stop as soon as a beer is found and exit through next door.

Pseudo code

Use of pseudocode

- Similar in many respects to C, Pascal, Java, Python (assignments, control structures, arrays, ...)
- Provides means to convey the essence of an algorithm in a concise fashion
- No fixed format for pseudocode. OK as long as it is unambiguous and breaks down the problem to the relevant basic steps.
- Pseudocode works out the important parts and abstracts the unimportant parts.

EXERCISE SL01-1/1

A prime number is a natural number greater than 1 and divisible only by 1 and itself. Write a program that determines all prime numbers between 0 and n.

Hint: Implement the sieve of Eratosthenes. Use an array of length $n + 1$ that indicates whether the number corresponding to the index is prime. Start with an array where all numbers greater than 1 are marked as primes and then proceed as follows: first eliminate the multiples of 2, then the multiples of 3, then the multiples of 4, etc.

EXERCISE SL01-1/2

Algo: Eratosth(A[0..n])

A[0] = FALSE; A[1] = FALSE;

for i = 2 to n do A[i] = TRUE

for i = 2 to $\lfloor \frac{n}{2} \rfloor$ — $\sqrt{n}$

n = 113

for j = 2 to $\lfloor n/i \rfloor$ do

A[i*j] = FALSE

2·2 2·3 2·4 2·5 2·6

3·2 3·3 3·4 &

4·2 4·3

5·2

6·2

Squaring

SEARCHING, C SOLUTION

```c
#include <stdio.h>

#define n 5

int i, v;
int a[] = { 11, 1, 4, -3, 22 };

int main() {
  i = 0;  v = -2;
  while (i < n && a[i] != v) { i++; }
  if (i < n) { printf("%d\n", i); }
  else { printf("NIL\n"); }
}
// gcc -o search search.c; ./search
```

How to approach C, etc.

- Do not study it (it is close to Java or Python and we only use a small subset).
- Whenever you meet a new construct learn and use it.
- Here:
  - `#include <stdio.h>` includes IO library with printf function
  - `#define n 5`
    defines n as a constant with value 5
  - `printf("%d\n",i)`
    prints an integer argument (`%d` is replaced by the value of i) followed by a new line (`\n`).
  - Arrays have a fixed size and start with index 0.
  - `//` is the start of a one line comment
  - `gcc -o search search.c; ./search` command to compile and run the program from the command line

SEARCHING, JAVA SOLUTION

```
import java.io.*;

class search {
  static int n = 5;
  static int i, v;
  static int a[] = { 11, 1, 4, -3, 22 };

  public static void main(String args[]) {
    i = 0; v = 22;
    while (i < n && a[i] != v) { i++; }
    if (i < n) { System.out.println(i); }
    else { System.out.println("NIL"); }
  }
}
// javac search.java; java search
```

SEARCHING, PYTHON SOLUTION

```
a = [ 11, 1, 4, -3, 22 ]
v = -3

j = -1
for i in range (len(a)):
    if a[i] == v:
        j = i
print(j)

# python search.py
```

DATA STRUCTURES

Informal definition

- A data structure is a way to store and organize data
- As such it facilitates access and modification of data
- Elementary data structures: lists, stacks, queues, trees

DATA STRUCTURES

Informal definition

- A data structure is a way to store and organize data
- As such it facilitates access and modification of data
- Elementary data structures: lists, stacks, queues, trees

Things to consider

- No single data structure works well for all purposes
- Choice of appropriate data structure is a **crucial** issue
- Important to know strengths and limitations of a variety of data structures.

TABLE OF CONTENTS — SL01

1. Introduction

2. Algorithms

3. Sorting
   - Bubble sort
   - Selection sort
   - Insertion sort

4. Recursion

SORTING/1

Why sorting?

- Most fundamental problem in the study of algorithms
- Algorithms often employ sorting as a key subroutine
- Wide range of sorting algorithms, rich set of techniques

SORTING/1

Why sorting?

- Most fundamental problem in the study of algorithms
- Algorithms often employ sorting as a key subroutine
- Wide range of sorting algorithms, rich set of techniques

Some observations

- Sorting of arrays: does provide random access (each element can be directly accessed)
- Efficient management of space: in-place sorting
- Efficiency is generally measured in terms of
  - number of comparisons and/or
  - number of exchange operations

SORTING/2

Simple methods

- Bubble sort
- Selection sort
- Insertion sort
- Roughly n^2 comparisons

SORTING/2

Simple methods

- Bubble sort
- Selection sort
- Insertion sort
- Roughly n^2 comparisons

Fast methods

- Merge sort
- Heap sort
- Quick sort
- Roughly $n \log n$ comparisons

SORTING/3

Problem specification

- Sorting problem
  - Input: a sequence of n numbers $A = [a_1, a_2, \ldots, a_n]$
  - Output: a permutation (reordering) $[b_1, b_2, \ldots, b_n]$ of the input sequence such that $b_1 \leq b_2 \leq \cdots \leq b_n$

SORTING/3

Problem specification

- Sorting problem
  - Input: a sequence of n numbers $A = [a_1, a_2, \ldots, a_n]$
  - Output: a permutation (reordering) $[b_1, b_2, \ldots, b_n]$ of the input sequence such that $b_1 \leq b_2 \leq \cdots \leq b_n$
- Instance of sorting
  - Input sequence: $[31, 41, 59, 26, 41, 58]$
  - Output sequence: $[26, 31, 41, 41, 58, 59]$

BUBBLE SORT/1

Rough idea

- Scan sequence and swap unsorted adjacent elements
- Repeat the procedure until sequence is actually sorted

BUBBLE SORT / 1

Rough idea

- Scan sequence and swap unsorted adjacent elements
- Repeat the procedure until sequence is actually sorted

The algorithm

Algo: BubbleSort(A)

```
for i = n to 2 do
    for j = 2 to i do
        if A[j]<A[j-1] then
            t = A[j];
            A[j] = A[j-1];
            A[j-1] = t;
```

BUBBLE SORT / 1

Rough idea

- Scan sequence and swap unsorted adjacent elements
- Repeat the procedure until sequence is actually sorted

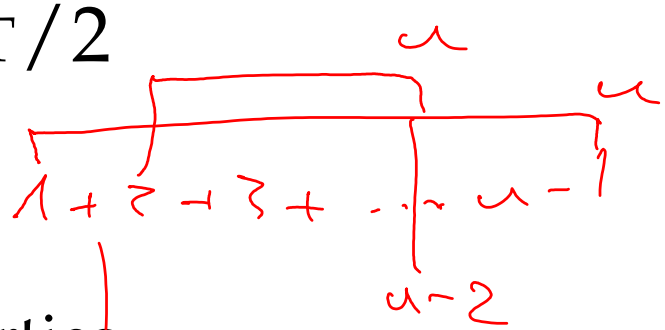
The algorithm

Algo: BubbleSort(A)

```
for i = n to 2 do
    for j = 2 to i do
        if A[j]<A[j-1] then
            t = A[j];
            A[j] = A[j-1];
            A[j-1] = t;
```

s	o	r	t	i	n	g
o	r	s	i	n	g	t
o	r	i	n	g	s	t
o	i	n	g	r	s	t
i	n	g	o	r	s	t
i	g	n	o	r	s	t
g	i	n	o	r	s	t

BUBBLE SORT/2



Basic properties

- Number of comparisons:

$$C = \sum_{i=2}^{n} (i-1) = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = \frac{n^2-n}{2}$$

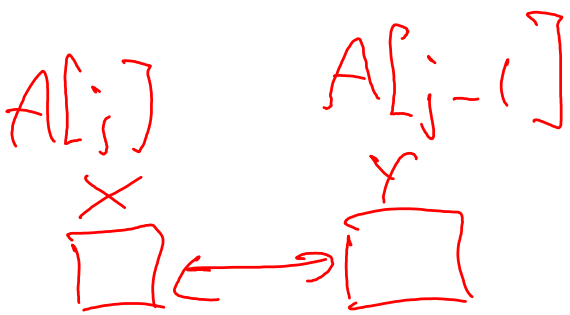
- The number of comparisons is independent of the original ordering and the values.

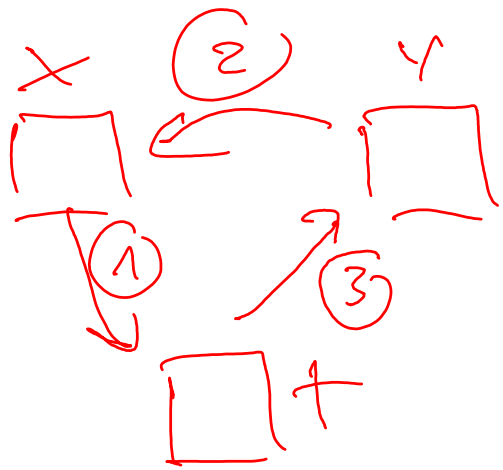
BUBBLE SORT / 3

Number of exchanges:

$$Mmin = 0$$

$$Mmax = \sum_{i=2}^{n} 3(i-1) = \frac{3n(n-1)}{2} = \frac{3n^2 - 3n}{2}$$





EXERCISE SL01-2

Modify the bubble sort algorithm such that, at each iteration of the outer loop, the smallest element of the corresponding subarray is moved to the left. Implement this variant of bubble sort and show how it works on a concrete example.

i = position where smallest elem is moved

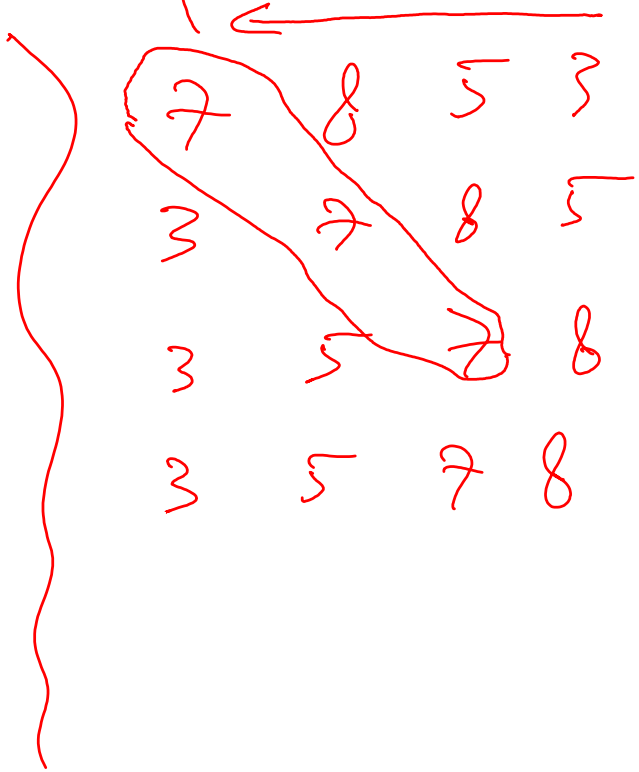
BSE(A)

for i=1 to n-1 do

for j= n-1 to i

if A[j+1] < A[j]

swap A[j+1] with A[j]



SELECTION SORT / 1

Rough idea

- Select smallest element and swap it with 1st element
- Repeat the procedure on remaining unsorted sequence

SELECTION SORT / 1

Rough idea

- Select smallest element and swap it with 1st element
- Repeat the procedure on remaining unsorted sequence

The algorithm

Algo: SelectionSort(A)

```
for i = 1 to n-1 do
    k = i;
    for j = i+1 to n do
        if A[j]<A[k] then k = j;
    exchange A[i] and A[k];
```

SELECTION SORT / 1

Rough idea

- Select smallest element and swap it with 1st element
- Repeat the procedure on remaining unsorted sequence

The algorithm

Algo: SelectionSort(A)

```
for i = 1 to n-1 do
    k = i;
    for j = i+1 to n do
        if A[j]<A[k] then k = j;
    exchange A[i] and A[k];
```

s	o	r	t	i	n	g
g	o	r	t	i	n	s
g	i	r	t	o	n	s
g	i	n	t	o	r	s
g	i	n	o	t	r	s
g	i	n	o	r	t	s
g	i	n	o	r	s	t

SELECTION SORT / 2

Basic properties

- Number of comparisons:

$$C = \sum_{i=1}^{n-1} i = \frac{n^2 - n}{2}$$

- The number of comparisons is independent of the original ordering.

SELECTION SORT / 3

Basic properties

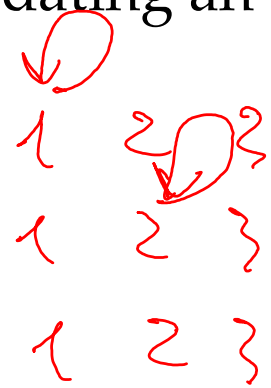
- Number of exchanges:

$$M = \sum_{i=1}^{n-1} 3 = 3(n-1) = 3n - 3$$

- Only the movement of elements is counted. Updating an index is not counted.

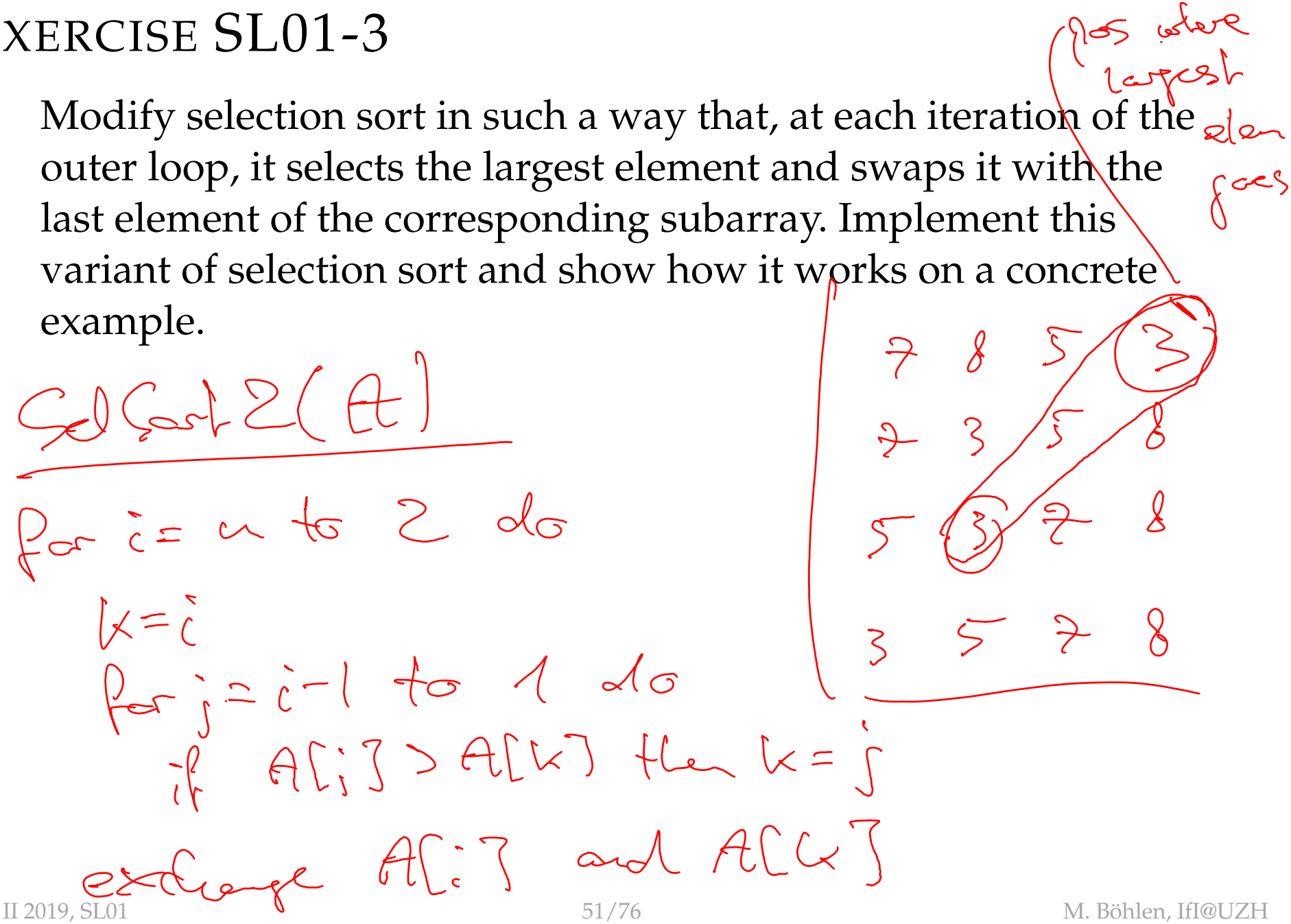
why not 0 exchanges in best case?

elem is exchanged with itself



EXERCISE SL01-3

Modify selection sort in such a way that, at each iteration of the outer loop, it selects the largest element and swaps it with the last element of the corresponding subarray. Implement this variant of selection sort and show how it works on a concrete example.



INSERTION SORT / 1

Rough idea

- Take first element and consider it as (sorted) sequence
- Continue taking elements and inserting into right place

INSERTION SORT / 1

Rough idea

- Take first element and consider it as (sorted) sequence
- Continue taking elements and inserting into right place

The algorithm

Algo: InsertionSort(A)

```
for i = 2 to n do
    j = i-1;
    t = A[i];
    while j≥1 ∧ t<A[j] do
        A[j+1] = A[j];
        j = j-1;
    A[j+1] = t;
```

INSERTION SORT / 1

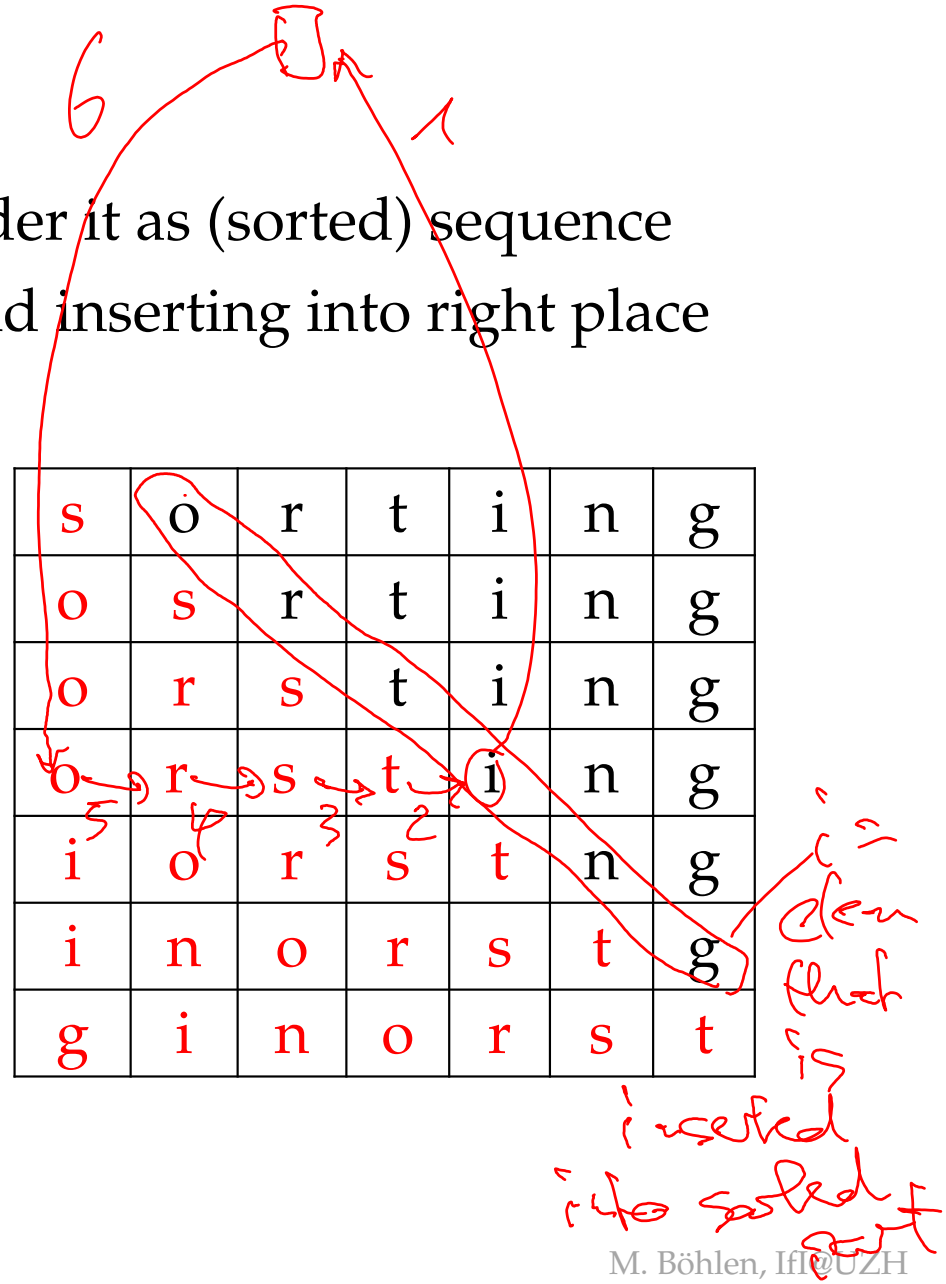
Rough idea

- Take first element and consider it as (sorted) sequence
- Continue taking elements and inserting into right place

The algorithm

Algo: InsertionSort(A)

```
for i = 2 to n do
    j = i-1;
    t = A[i];
    while j>1 ∧ t<A[j] do
        A[j+1] = A[j];
        j = j-1;
    A[j+1] = t;
```



INSERTION SORT / 2

Basic properties

- Number of comparisons:

$$Cmin = \sum_{i=2}^{n} 1 = n - 1$$

$$Cmax = \sum_{i=2}^{n} (i - 1) = \frac{n^2 - n}{2}$$

INSERTION SORT / 3

Basic properties

- Number of exchanges:

$$Mmin = \sum_{i=2}^{n} 2 = 2(n-1) = 2n-2$$

$$Mmax = \sum_{i=2}^{n} (i+1) = \frac{n^2+3n-4}{2}$$

TABLE OF CONTENTS — SL01

1. Introduction

2. Algorithms

3. Sorting

4. Recursion
   Marking a ruler

RECURSION/1

Recursive object

- It contains itself as part of it
- It is defined in terms of itself

RECURSION/1

Recursive object

- It contains itself as part of it
- It is defined in terms of itself

Recursive procedure

- Procedure that calls itself
- Multiple & mutual recursive calls
- Termination condition to stop recursion

RECURSION/1

Recursive object

- It contains itself as part of it
- It is defined in terms of itself

Recursive procedure

- Procedure that calls itself
- Multiple & mutual recursive calls
- Termination condition to stop recursion

Example: simple recursion

Factorial $$fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot fact(n-1) & \text{if } n > 0 \end{cases}$$

RECURSION/2

Evaluate the following three implementations for computing the factorial of an integer n.

Algo: fact1(n)

if n==*0* **then return** 1 **else return** n * fact1(n-1);

Algo: fact2(n)

if n==*0* **then return** 1;
return fact2(n-1) * n;

Algo: fact3(n)

return n * fact3(n-1);
if n==*0* **then return** 1;

RECURSION/3

Tracing the call `fact1(3)`:

`fact1`(3)

RECURSION/3

Tracing the call `fact1(3)`:

`fact1`(3)

3 * `fact1`(2)

RECURSION/3

Tracing the call `fact1(3)`:

`fact1`(3)

3 * `fact1`(2)

2 * `fact1`(1)

RECURSION/3

Tracing the call `fact1(3)`:

`fact1`(3)

3 * `fact1`(2)

2 * `fact1`(1)

1 * `fact1`(0)

RECURSION/3

Tracing the call `fact1(3)`:

`fact1`(3)

3 * `fact1`(2)

2 * `fact1`(1)

1 * `fact1`(0)

`fact1`(0)

RECURSION/3

Tracing the call `fact1(3)`:

`fact1`(3)	
3 * `fact1`(2)	
2 * `fact1`(1)	
1 * `fact1`(0)	
`fact1`(0)	1

RECURSION/3

Tracing the call `fact1(3)`:

`fact1`(3)	
3 * `fact1`(2)	
2 * `fact1`(1)	
1 * `fact1`(0)	1
`fact1`(0)	1

RECURSION/3

Tracing the call `fact1(3)`:

`fact1`(3)	
3 * `fact1`(2)	
2 * `fact1`(1)	2
1 * `fact1`(0)	1
`fact1`(0)	1

RECURSION/3

Tracing the call `fact1(3)`:

`fact1`(3)	
3 * `fact1`(2)	6
2 * `fact1`(1)	2
1 * `fact1`(0)	1
`fact1`(0)	1

RECURSION/3

Tracing the call `fact1(3)`:

`fact1`(3)	6
3 * `fact1`(2)	6
2 * `fact1`(1)	2
1 * `fact1`(0)	1
`fact1`(0)	1

RECURSION / 4

Example: multiple recursion

Fibonacci $$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n > 1 \end{cases}$$

RECURSION / 4

Example: multiple recursion

$$\text{Fibonacci} \quad fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n > 1 \end{cases}$$

Example: mutual recursion

$$\text{Odd number} \quad odd(n) = \begin{cases} false & \text{if } n = 0 \\ even(n-1) & \text{if } n > 0 \end{cases}$$

$$\text{Even number} \quad even(n) = \begin{cases} true & \text{if } n = 0 \\ odd(n-1) & \text{if } n > 0 \end{cases}$$

EXERCISE SL01-4

Design recursive algorithms for *odd* and *even*.

e.g. odd(2)

```
even(n)
  return n==0 || odd(n-1)

odd(n)
  return n>0 && even(n-1)
```

```
odd(n)
  return (n<0 && odd(-n)) ||
         (n>0 && even(n-1))
```

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)

RECURSION/5

Tracing the call `fib(4)`:

`fib(4)`

`fib(3)` + `fib(2)`

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)

`fib`(3) + `fib`(2)

`fib`(2) + `fib`(1)

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)

`fib`(3) + `fib`(2)

`fib`(2) + `fib`(1)

`fib`(1) + `fib`(0)

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)

`fib`(3) + `fib`(2)

`fib`(2) + `fib`(1)

`fib`(1) + `fib`(0)

`fib`(1)

RECURSION/5

Tracing the call `fib(4)`:

```
fib(4)
   fib(3) + fib(2)
      fib(2) + fib(1)
         fib(1) + fib(0)
            fib(1)                    1
```

RECURSION/5

Tracing the call `fib(4)`:

```
fib(4)
   fib(3) + fib(2)
      fib(2) + fib(1)
         fib(1) + fib(0)
            fib(1)                  1
            fib(0)
```

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)	
`fib`(3) + `fib`(2)	
`fib`(2) + `fib`(1)	
`fib`(1) + `fib`(0)	
`fib`(1)	1
`fib`(0)	0

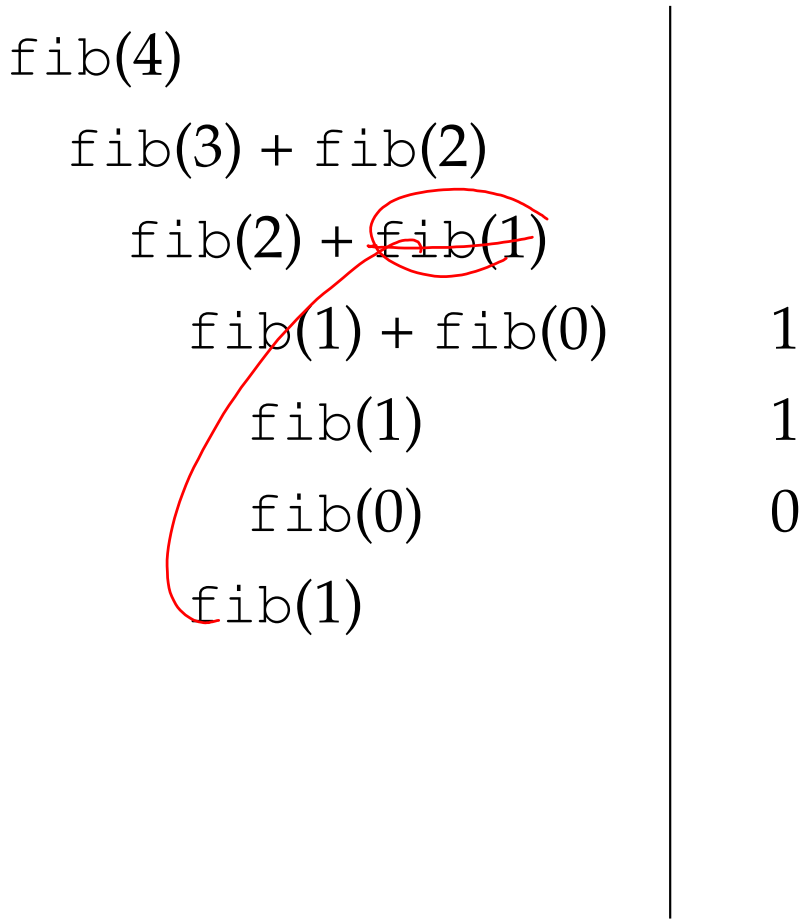
RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)	
`fib`(3) + `fib`(2)	
`fib`(2) + `fib`(1)	
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0

RECURSION/5

Tracing the call `fib(4)`:



RECURSION/5

Tracing the call `fib(4)`:

`fib(4)`	
`fib(3) + fib(2)`	
`fib(2) + fib(1)`	
`fib(1) + fib(0)`	1
`fib(1)`	1
`fib(0)`	0
`fib(1)`	1

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)	
`fib`(3) + `fib`(2)	
`fib`(2) + `fib`(1)	2
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0
`fib`(1)	1

RECURSION/5

Tracing the call `fib(4)`:

`fib(4)`	
`fib(3) + fib(2)`	
`fib(2) + fib(1)`	2
`fib(1) + fib(0)`	1
`fib(1)`	1
`fib(0)`	0
`fib(1)`	1
`fib(1) + fib(0)`	

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)	
`fib`(3) + `fib`(2)	
`fib`(2) + `fib`(1)	2
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0
`fib`(1)	1
`fib`(1) + `fib`(0)	
`fib`(1)	

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)	
`fib`(3) + `fib`(2)	
`fib`(2) + `fib`(1)	2
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0
`fib`(1)	1
`fib`(1) + `fib`(0)	
`fib`(1)	1

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)	
`fib`(3) + `fib`(2)	
`fib`(2) + `fib`(1)	2
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0
`fib`(1)	1
`fib`(1) + `fib`(0)	
`fib`(1)	1
`fib`(0)	

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)	
`fib`(3) + `fib`(2)	
`fib`(2) + `fib`(1)	2
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0
`fib`(1)	1
`fib`(1) + `fib`(0)	
`fib`(1)	1
`fib`(0)	0

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)	
`fib`(3) + `fib`(2)	
`fib`(2) + `fib`(1)	2
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0
`fib`(1)	1
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)	
`fib`(3) + `fib`(2)	3
`fib`(2) + `fib`(1)	2
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0
`fib`(1)	1
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0

RECURSION/5

Tracing the call `fib(4)`:

`fib`(4)	3
`fib`(3) + `fib`(2)	3
`fib`(2) + `fib`(1)	2
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0
`fib`(1)	1
`fib`(1) + `fib`(0)	1
`fib`(1)	1
`fib`(0)	0

RECURSION/6

Is recursion necessary?

- Practice: recursion is elegant and in some cases the best solution by far

RECURSION/6

Is recursion necessary?

- Practice: recursion is elegant and in some cases the best solution by far
- Theory: one could resort to iteration and explicitly maintain a recursion stack

RECURSION/6

Is recursion necessary?

- Practice: recursion is elegant and in some cases the best solution by far
- Theory: one could resort to iteration and explicitly maintain a recursion stack
- In the above examples recursion is not necessary: there exist simple iterative solutions

RECURSION/6

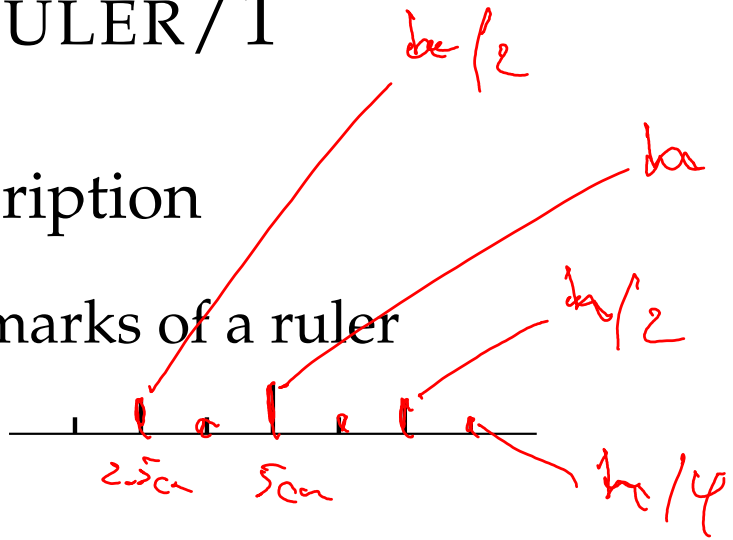
Is recursion necessary?

- Practice: recursion is elegant and in some cases the best solution by far
- Theory: one could resort to iteration and explicitly maintain a recursion stack
- In the above examples recursion is not necessary: there exist simple iterative solutions
- Recursion is more expensive than corresponding iterative solution: bookkeeping is necessary

MARKING A RULER/1

Problem description

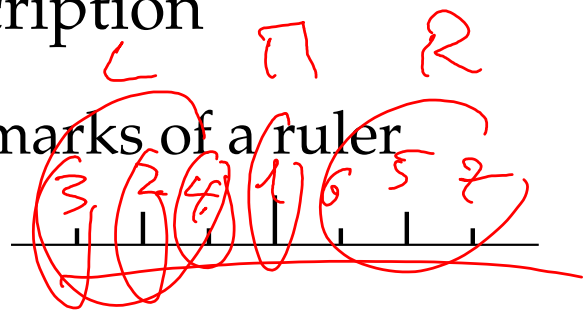
- Print the marks of a ruler
- Example:



MARKING A RULER/1

Problem description

- Print the marks of a ruler
- Example:



Recursive algorithm

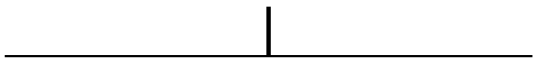
Algo: Ruler(l,r,h)

```
m = (r+l)/2;
if h>0 then
    Mark(m,h);
    Ruler(l,m,h-1);
    Ruler(m,r,h-1);
```

MARKING A RULER/2

Printing of marks:

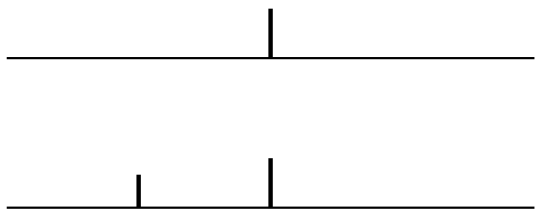
`Ruler` $(0, 8, 3)$
`Mark` $(4, 3)$



MARKING A RULER/2

Printing of marks:

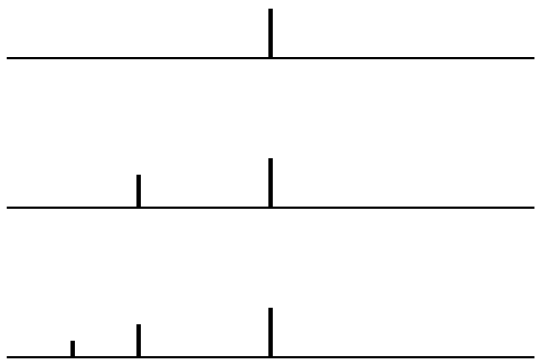
```
Ruler (0, 8, 3)
  Mark (4, 3)
  Ruler (0, 4, 2)
    Mark (2, 2)
```



MARKING A RULER/2

Printing of marks:

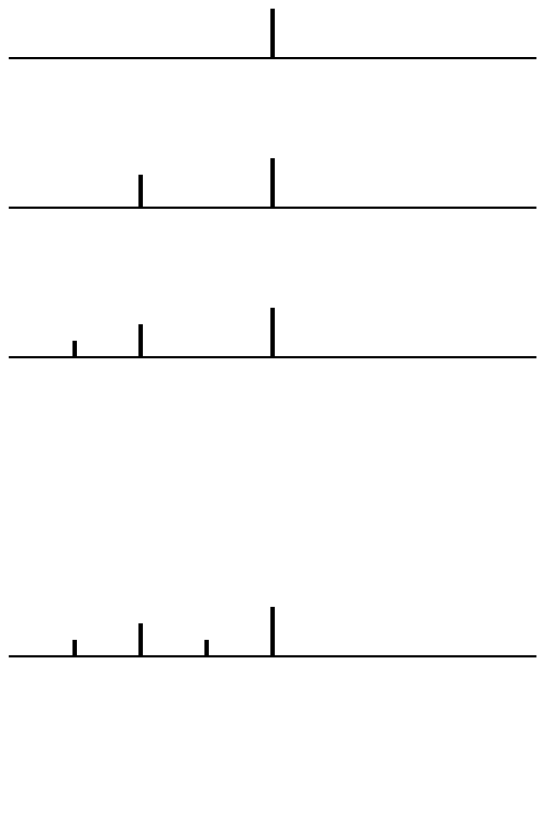
```
Ruler (0, 8, 3)
  Mark (4, 3)
  Ruler (0, 4, 2)
    Mark (2, 2)
    Ruler (0, 2, 1)
      Mark (1, 1)
      Ruler (0, 1, 0)
      Ruler (1, 2, 0)
```



MARKING A RULER/2

Printing of marks:

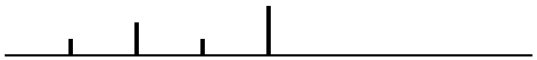
```
Ruler (0, 8, 3)
  Mark (4, 3)
  Ruler (0, 4, 2)
    Mark (2, 2)
    Ruler (0, 2, 1)
      Mark (1, 1)
      Ruler (0, 1, 0)
      Ruler (1, 2, 0)
    Ruler (2, 4, 1)
      Mark (3, 1)
      Ruler (2, 3, 0)
      Ruler (3, 4, 0)
```



MARKING A RULER/3

Printing of marks:

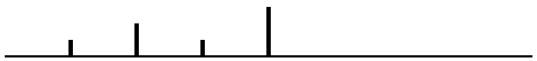
$[\cdots]$



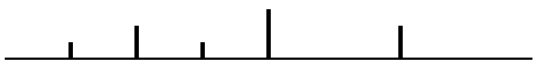
MARKING A RULER/3

Printing of marks:

$[\cdots]$



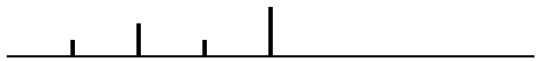
`Ruler` $(4, 8, 2)$
`Mark` $(6, 2)$



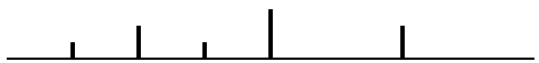
MARKING A RULER/3

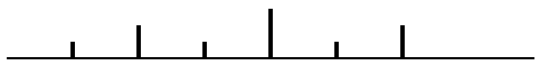
Printing of marks:

$[\cdots]$



`Ruler` $(4, 8, 2)$
  `Mark` $(6, 2)$
  `Ruler` $(4, 6, 1)$
    `Mark` $(5, 1)$
    `Ruler` $(4, 5, 0)$
    `Ruler` $(5, 6, 0)$



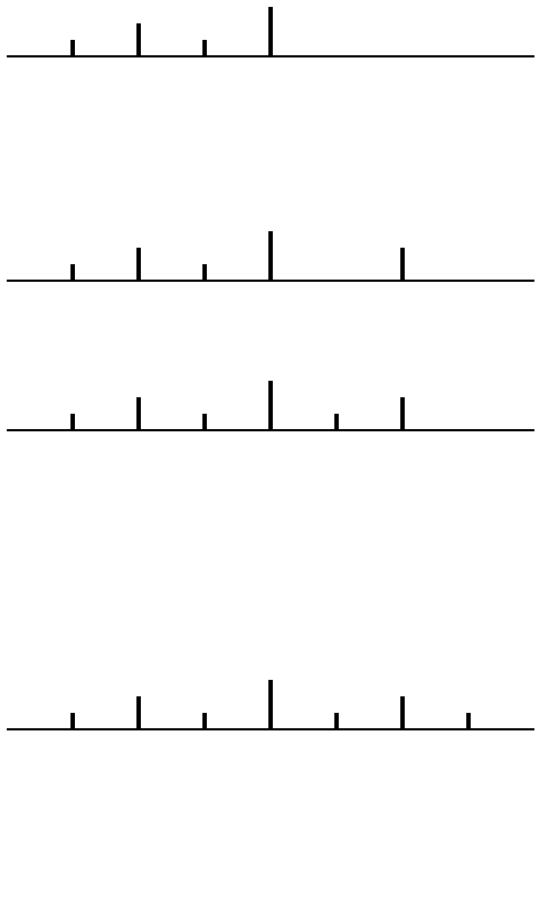


Marking a ruler/3

Printing of marks:

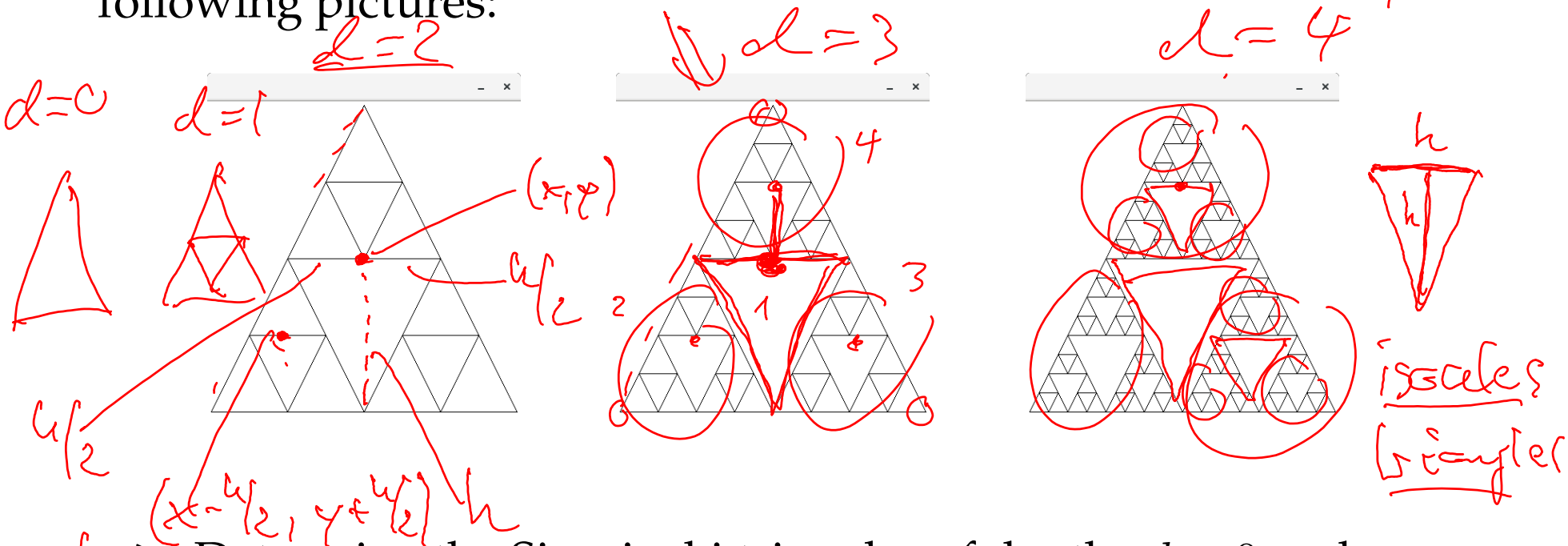
```
[···]

  Ruler (4, 8, 2)
    Mark (6, 2)
    Ruler (4, 6, 1)
      Mark (5, 1)
      Ruler (4, 5, 0)
      Ruler (5, 6, 0)
    Ruler (6, 8, 1)
      Mark (7, 1)
      Ruler (6, 7, 0)
      Ruler (7, 8, 0)
```



EXERCISE SL01-5/1

Consider the Sierpinski triangles of depths 2, 3, and 4 in the following pictures:



- Determine the Sierpinski triangles of depths $d = 0$ and $d = 1$.
- Design a recursive algorithm that draws a Sierpinski triangle of depth d.
- Show how to call your algorithm.

EXERCISE SL01-5/2

ST (4)

Sierp Triangle (x, y, h, d)

if d > 0 then

1) draw triangle: (x-h/2, y) - (x+h/2, y) - (x, y+h)

2) SierpT. (x, y-h/2, h/2, d-1)

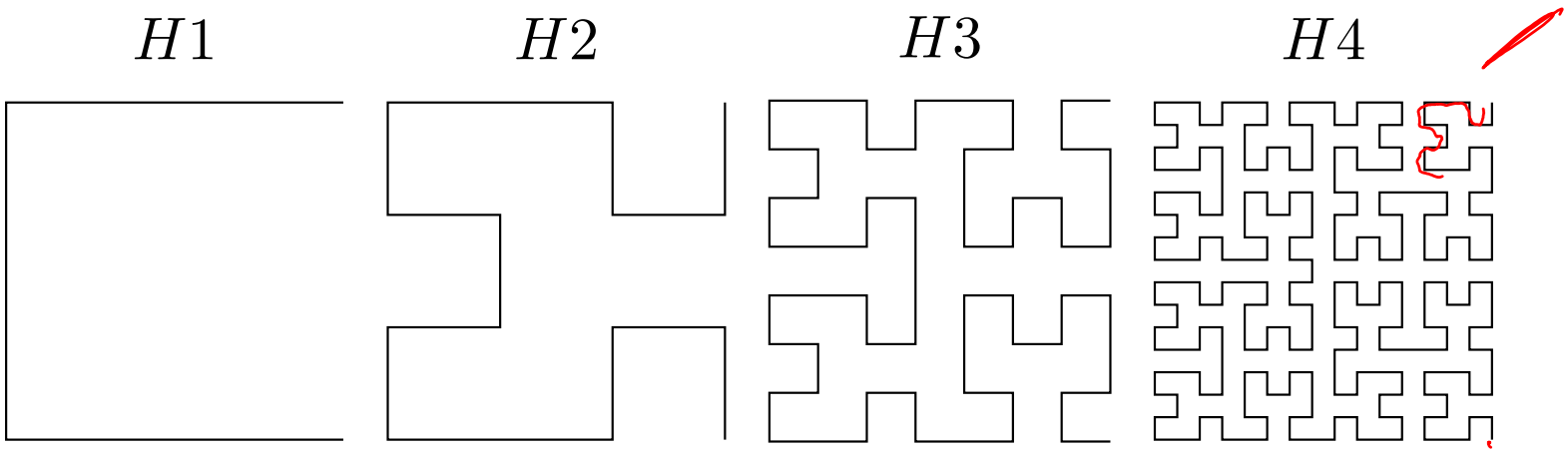
3) SierT (x-h/2, y+h/2, h/2, d-1)

4) SierT (x+h/2, y+h/2, h/2, d-1)

call: draw triangle (0, y_{max}) - (x_{max}, y_{max}) - (x_{max}/2, 0)
SierpT (h/2, h/2, h/2, d)

HILBERT CURVE/1

Consider the following patterns:

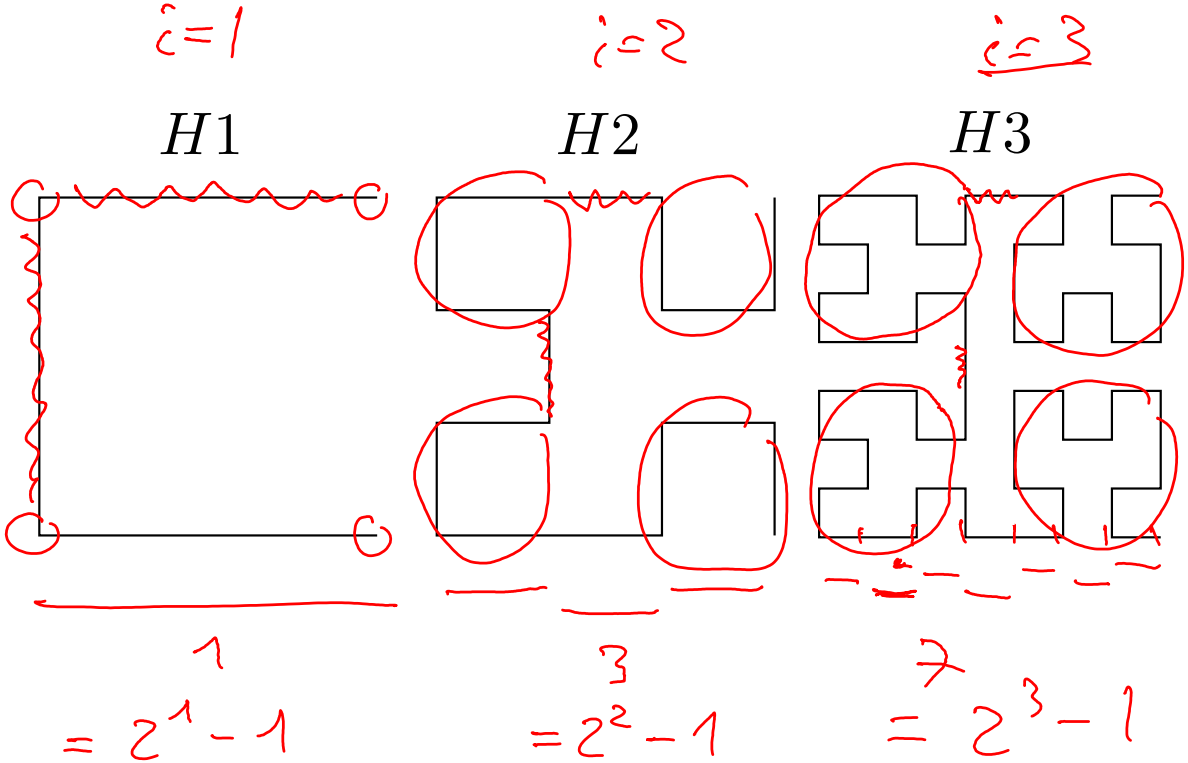


- Hi is the Hilbert curve of order i (named after the inventor David Hilbert).
- Goal: write a program to produce Hi.

HILBERT CURVE/2

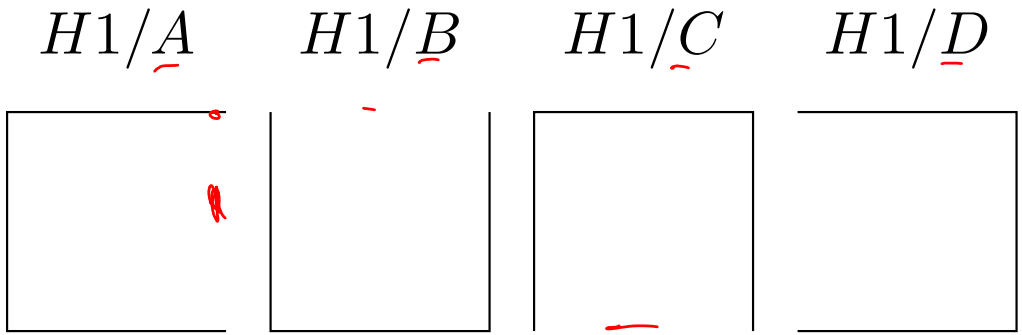
$fl_i = 2^i - 1$

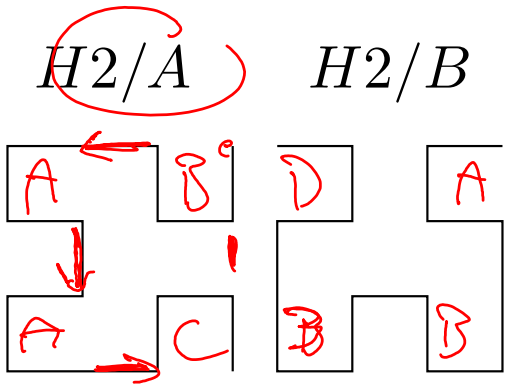
- Observation: Hi+1 is composed of four components of Hi.
- Solution: Determine the recursion scheme to generate Hi.



HILBERT CURVE/3

- Observation: Hi+1 is composed of four components of Hi.
- Solution: Determine the recursion scheme to generate Hi.





A = B← A↓ A→ C
B = A↓ B← B↑ D
C = D↑ C→ C↓ A
D = C→ D↑ D← B

HILBERT CURVE/4

The first part of the code:

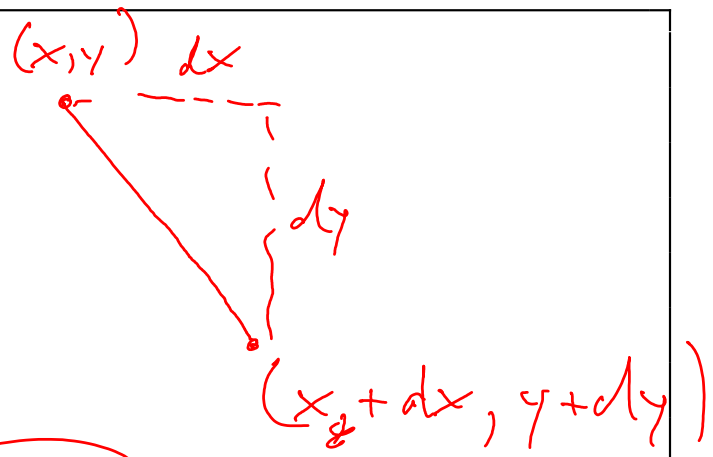
```
#include <SDL2/SDL.h>

SDL_Window *win;
SDL_Renderer *ren;

float x, y, u;

void l(float dx, float dy) {
  SDL_RenderDrawLine(ren, x, y, x+dx, y+dy);
  x=x+dx; y=y+dy;
  SDL_Delay(5);
  SDL_RenderPresent(ren);
}

...
```



HILBERT CURVE/5

The last part of the code:




```
...

int main (int argc, char** argv) {
  int d;
  sscanf(argv[1], "%d", &d);

  SDL_CreateWindowAndRenderer(400, 400, 0, &win, &ren);
  SDL_SetRenderDrawColor(ren, 255, 255, 255, 0);
  x=395; y=5; u=390/(pow(2,d)-1);
  A(d);

  SDL_Event e;
  do { SDL_PollEvent(&e); } while (e.type != SDL_QUIT);
}

// gcc hilbert.c -o hilbert -lm -lSDL2; ./hilbert 2
```

HILBERT CURVE/6

The middle part of the code:

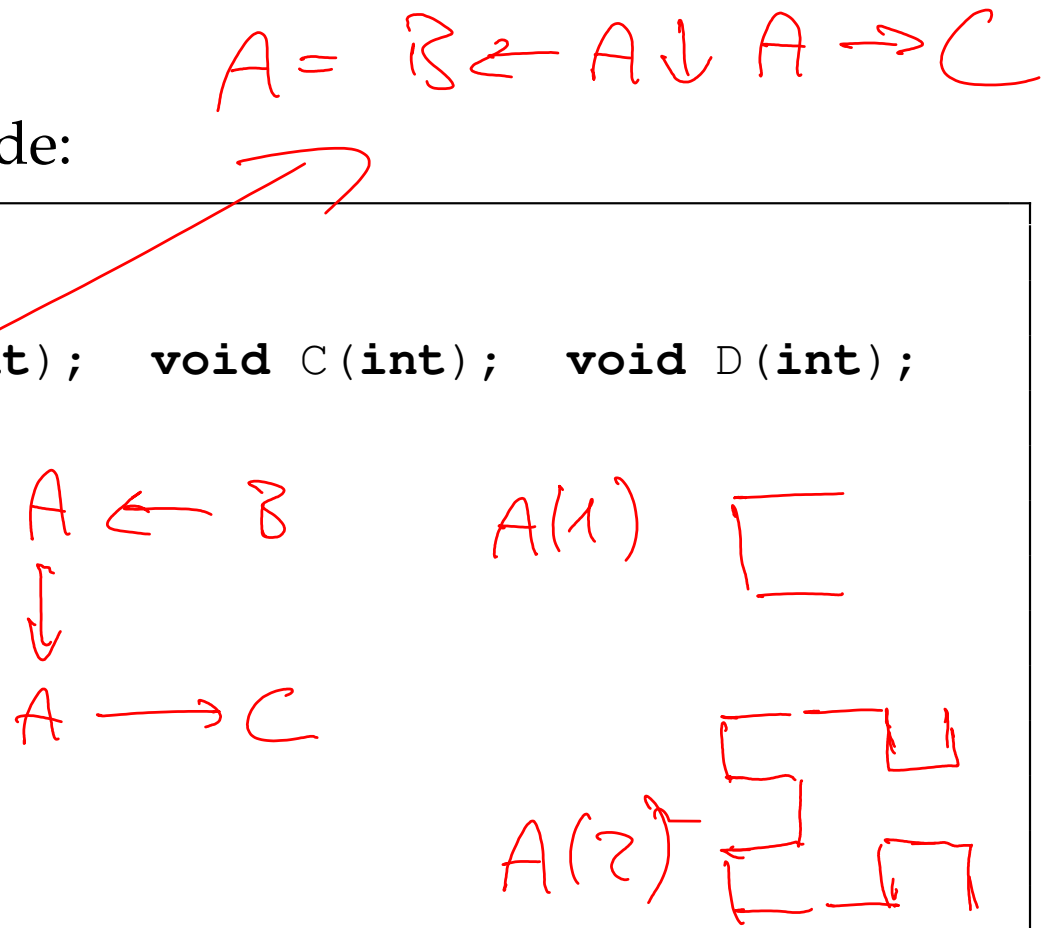
```
...

void A(int);  void B(int);  void C(int);  void D(int);

void A(int i) {
  if (i>0) {
    B(i-1); l(-u,0);
    A(i-1); l(0,u);
    A(i-1); l(u,0);
    B(i-1);
  }
}

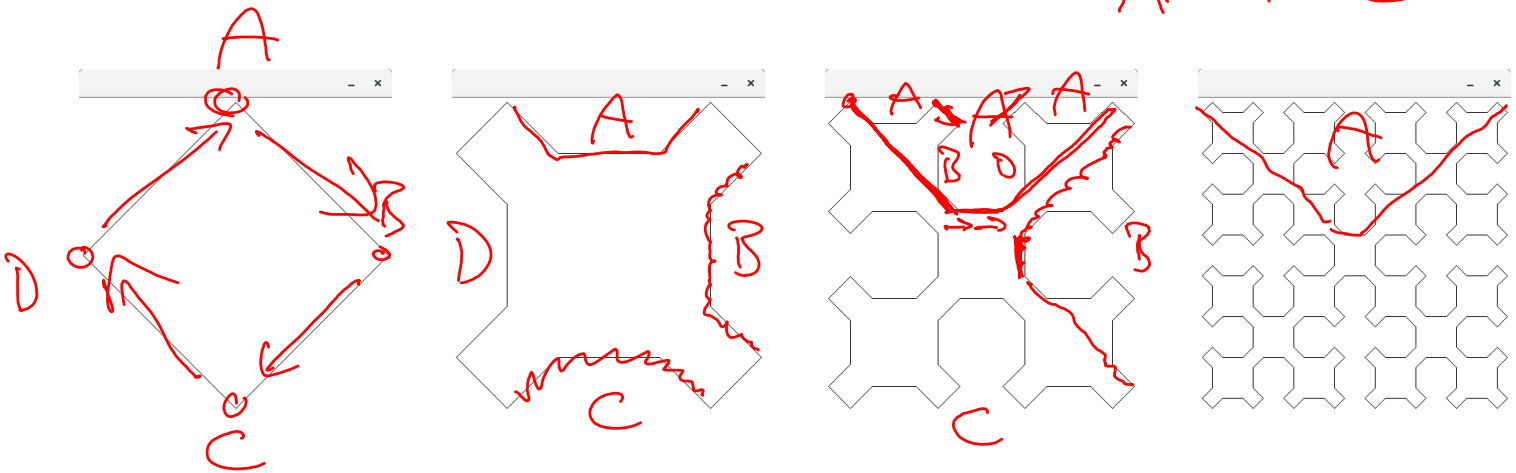

void B(int i) { ...

...
```



Exercise SL01-6/1

Consider the Sierpinski curve:



- The above pictures illustrate the Sierpinski curve of order 0, 1, 2 and 3.
- Identify the recursive pattern and sketch the code to draw the Sierpinski curve of order i.

EXERCISE SL01-6/2

Algo: S(i)

if i > 0 then

A(i); l(u, u); B(i); l(-u, u);

C(i); l(-u, -u); D(i); l(u, -u)

Algo: A(i)

if i > 0 then

A(i-1); l(u, u); ...

SUMMARY

- Algorithmic problem:
  - An algorithm transforms input data into output data
  - Precisely specify **input** and **output**; consider special cases; work out representative examples
  - A first step is to come up with a simple plan.
  - Split or revise complex plans until they get simple.
- Sorting algorithms
  - A classical and representative algorithmic problem.
  - Initialization and conditions in loops are important.
  - Make sure you get details of loops **correct by design**. Avoid trial and error.
- Recursive algorithms
  - Recursion is an important algorithmic technique.
  - Practice the design of recursive algorithms.
  - Consider **special cases**: termination, the first couple of recursive calls.

Informatik II

Complexity and Correctness
— SL02 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

TABLE OF CONTENTS — SL02

1. Algorithmic complexity
   - Efficiency
   - Runtime of insertion sort
   - Best, worst and average case
   - Runtime of binary search

2. Correctness

3. Asymptotic complexity

4. Special case analysis

EFFICIENCY / 1

Analysis of algorithms

- Predicting the resources that the algorithm requires
  - Resources in terms of time: running time time complexity
  - Resources in terms of space: memory usage space complexity

EFFICIENCY/1

Analysis of algorithms

- Predicting the resources that the algorithm requires
  - Resources in terms of time: running time
  - Resources in terms of space: memory usage
- Efficiency (use of resources) depends on input size

EFFICIENCY/1

Analysis of algorithms

- Predicting the resources that the algorithm requires
  - Resources in terms of time: running time
  - Resources in terms of space: memory usage
- Efficiency (use of resources) depends on input size
- Various ways of determining the size of input data
  - Depends on the problem being studied
  - Number of items in the input (e.g. sorting)
  - Total number of bits (e.g. integer arithmetic)

EFFICIENCY/1

Analysis of algorithms

- Predicting the resources that the algorithm requires
  - Resources in terms of time: running time
  - Resources in terms of space: memory usage
- Efficiency (use of resources) depends on input size
- Various ways of determining the size of input data
  - Depends on the problem being studied
  - Number of items in the input (e.g. sorting)
  - Total number of bits (e.g. integer arithmetic)
- Must have a model of the implementation technology

EFFICIENCY/2

- usually "nr of elements" and "nr of operations" is enough (good enough precision) cont

Random-access machine - decide what to consider / not consider

- Generic one-processor, random-access model of computation

- all details is not good/possible

EFFICIENCY / 2

Random-access machine

- Generic one-processor, random-access model of computation
- RAM contains instructions commonly found in real computers
  - Arithmetic: add, subtract, multiply, divide, etc.
  - Control: branch, subroutine call, return
  - Data movement: load, store, copy

EFFICIENCY/2

Random-access machine

- Generic one-processor, random-access model of computation
- RAM contains instructions commonly found in real computers
  - Arithmetic: add, subtract, multiply, divide, etc.
  - Control: branch, subroutine call, return
  - Data movement: load, store, copy
- Each instruction takes a constant amount of time

EFFICIENCY/2

Random-access machine

- Generic one-processor, random-access model of computation
- RAM contains instructions commonly found in real computers
  - Arithmetic: add, subtract, multiply, divide, etc.
  - Control: branch, subroutine call, return
  - Data movement: load, store, copy
- Each instruction takes a constant amount of time
- Data types: character, integer, and floating point

RUNTIME OF INSERTION SORT / 1

Analysis of the algorithm

```
for i = 2 to n do
    j = i-1;
    t = A[i];
    while j>0 ∧ t<A[j] do
        A[j+1] = A[j];
        j = j-1;
    A[j+1] = t;
```

c_1	n
c_2	$n-1$
c_3	$n-1$
c_4	$\sum_{i=2}^{n} t_i$
c_5	$\sum_{i=2}^{n} (t_i - 1)$
c_6	$\sum_{i=2}^{n} (t_i - 1)$
c_7	$n-1$

- Execution of k-th line takes time c_k (a constant)
- t_i: the number of times a nested line is executed

RUNTIME OF INSERTION SORT / 2

Exact running time

time for n elements

Sum of products of rightmost columns (cost, times)

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} t_i + c_5 \sum_{i=2}^{n} (t_i - 1) + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7(n-1)$$

RUNTIME OF INSERTION SORT / 2

Exact running time

Sum of products of rightmost columns (cost, times)

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} t_i + c_5 \sum_{i=2}^{n} (t_i - 1) + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7(n-1)$$

Some observations

- Running time seen as function of the input size n
- May depend on **which** input of size n is given
- Such behavior is captured by the parameter t_i

Runtime of insertion sort / 3

Best case running time:

Array is already sorted: $t_i = 1$

$$\begin{aligned} T(n) &= c_1 n + (c_2 + c_3 + c_4 + c_7)(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \\ &= an + b \end{aligned}$$

RUNTIME OF INSERTION SORT / 3

Best case running time:

Array is already sorted: $t_i = 1$

$$\begin{aligned} T(n) &= c_1 n + (c_2 + c_3 + c_4 + c_7)(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \\ &= an + b \end{aligned}$$

linear

Worst case running time:

Array is in reverse sorted order: $t_i = i$

$$\begin{aligned} T(n) = c_1 n + (c_2 + c_3 + c_7)(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) \\ + (c_5 + c_6)\frac{(n-1)n}{2} = an^2 + bn + c \end{aligned}$$

quadratic

BEST, WORST AND AVERAGE CASE/1

Worst vs average case

- Usually, one focuses on worst case running time

Best, worst and average case/1

Worst vs average case

- Usually, one focuses on worst case running time
- There are basically the following three reasons
  - Worst case running time provides an upper bound
    - **Guarantee** that algorithm will never take any longer
  - For some algorithms, worst case occurs fairly often
    - Example: searching for an element that is not present
  - Average case is often roughly as bad as worst case
    - Example: insertion sort is still quadratic on average

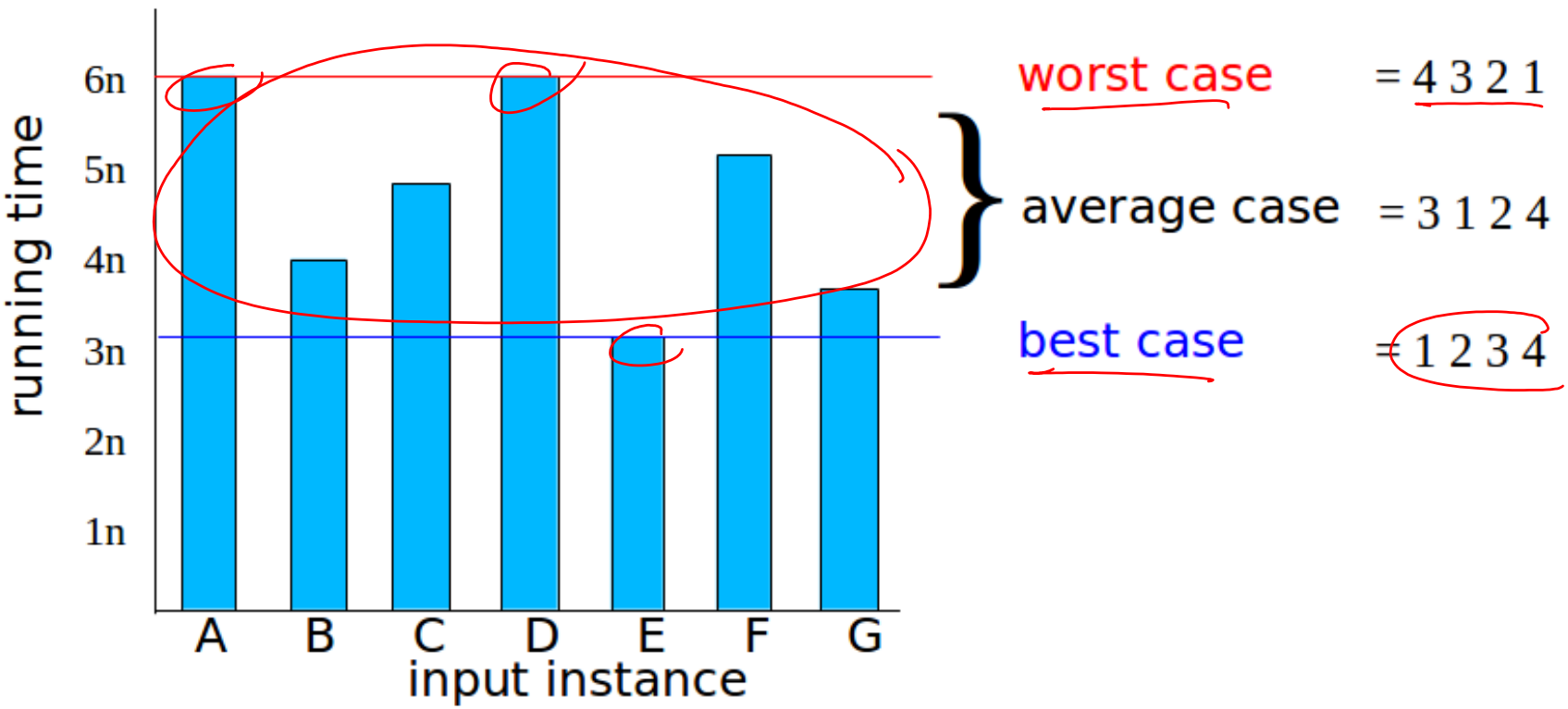
Best, worst and average case/1

Worst vs average case

- Usually, one focuses on worst case running time
- There are basically the following three reasons
  - Worst case running time provides an upper bound
    - **Guarantee** that algorithm will never take any longer
  - For some algorithms, worst case occurs fairly often
    - Example: searching for an element that is not present
  - Average case is often roughly as bad as worst case
    - Example: insertion sort is still quadratic on average
- Might be quite difficult to find typical average case

BEST, WORST AND AVERAGE CASE/2

For a specific size of input size n, investigate running times for different input instances:



EXERCISE SL02-1

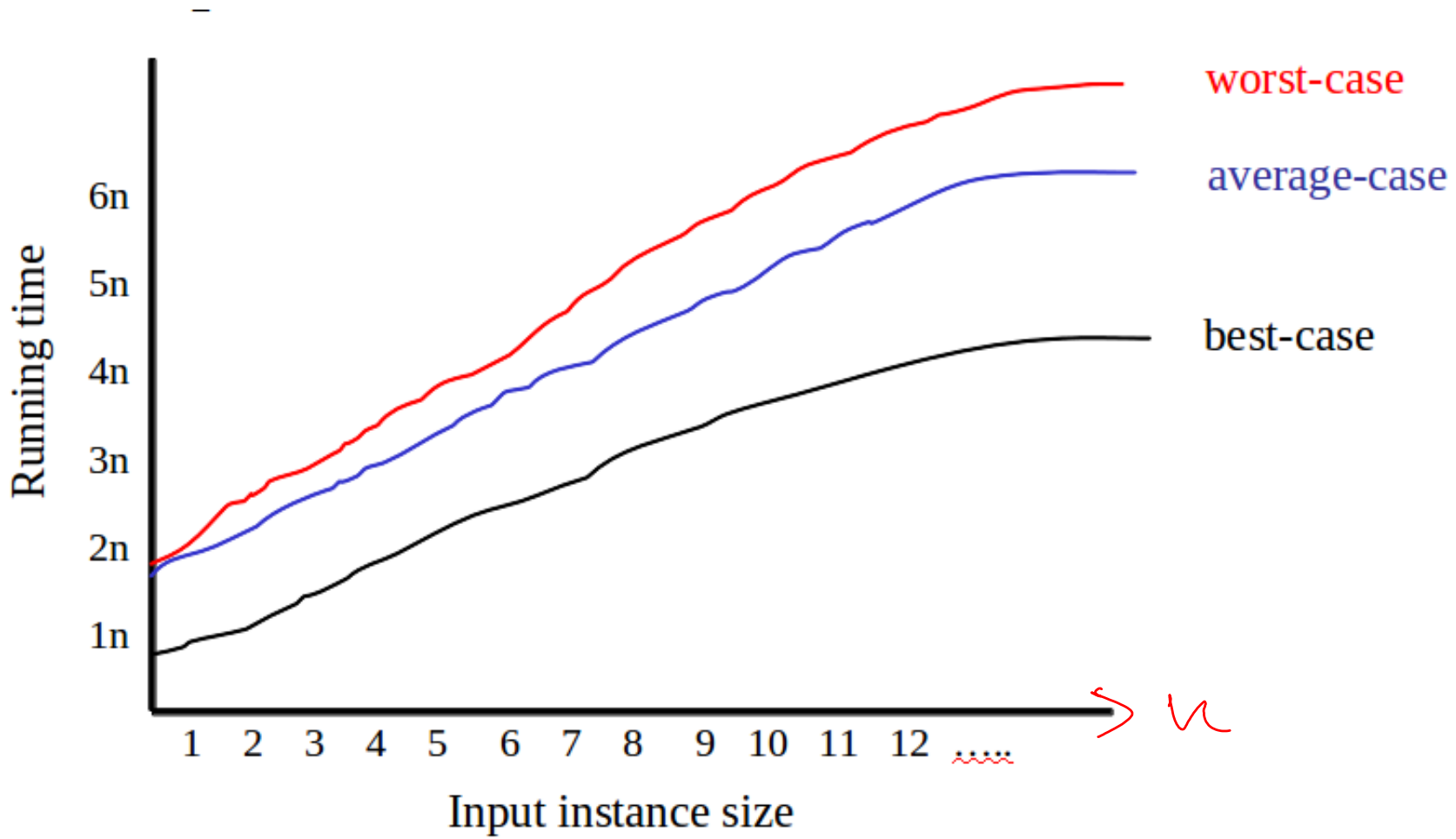
How can we modify almost any algorithm to have a good best-case running time?

- modify algo to first check if input is equal to some special case
- if so output pre-computed solution
- dess
- valid & good approach (can be use for "cheating")

→ therefor best case often not a good indicator

BEST, WORST AND AVERAGE CASE/3

For inputs of all sizes:



BEST, WORST AND AVERAGE CASE/4

Concluding remarks

- Often, counting the number of iterations of the core (innermost) part is sufficient
  - No distinction between comparisons, assignments, etc.
  - Means that each of them has roughly the same cost
  - Does provide results with good/adequate precision
  - Simplifying abstraction: asymptotic analysis

Best, worst and average case/4

Concluding remarks

- Often, counting the number of iterations of the core (innermost) part is sufficient
  - No distinction between comparisons, assignments, etc.
  - Means that each of them has roughly the same cost
  - Does provide results with good/adequate precision
  - Simplifying abstraction: asymptotic analysis
- In certain cases the cost of a specific operation may dominate all other costs
  - Example: disk I/O versus RAM operations
- Today simple cost models that consider one (the "dominating") cost only have to be refined since their accuracy is insufficient.

RUNTIME OF BINARY SEARCH/1

Searching revisited

Algo: LinSearch2(A,v)

Input: sequence $A[1..n]$ of length n, value v
Output: either index i such that $v = A[i]$ or `NIL`

```
i = 1;
while i≤n ∧ A[i]≠v do i = i+1;
if i≤n then return i else return NIL;
```

RUNTIME OF BINARY SEARCH/1

Searching revisited

Algo: LinSearch2(A,v)

Input: sequence $A[1..n]$ of length n, value v
Output: either index i such that $v = A[i]$ or `NIL`

```
i = 1;
while i≤n ∧ A[i]≠v do i = i+1;
if i≤n then return i else return NIL;
```

- Running times:
  - worst case: n
  - average case: $n/2$
  - best case: 0

RUNTIME OF BINARY SEARCH/1

Searching revisited

Algo: LinSearch2(A,v)

Input: sequence $A[1..n]$ of length n, value v
Output: either index i such that $v = A[i]$ or `NIL`

i = 1;
while $i \leq n \wedge A[i] \neq v$ **do** i = i+1;
if $i \leq n$ **then return** i **else return** NIL;

- Running times:
  - worst case: n
  - average case: $n/2$
  - best case: 0
- Do things change if the sequence is sorted?

Runtime of binary search/2

Sorted sequence

Consider middle element & narrow down the search space



RUNTIME OF BINARY SEARCH/2

Sorted sequence

Consider middle element & narrow down the search space

The algorithm

Algo: BinSearch1(A,v)

Input: sequence $A[1..n]$ of length n, value v
Output: either index i such that $v == A[i]$ or `NIL`

```
l = 1; r = n;
m = ⌊(l+r)/2⌋;
while l≤r ∧ v≠A[m] do
    if v<A[m] then r = m-1 else l = m+1;
    m = ⌊(l+r)/2⌋;
if l≤r then return m else return NIL;
```



EXERCISE SL02-2

Illustrate the binary search algorithm for the following examples:

- A = [10, 20, 30, 40, 50, 60, 70], q = 30
- A = [10, 20, 30, 40, 50], q = 22

10 20 30 40 50 60 70 $\ell=1, r=7 \quad m=4$

$\ell=1, r=3 \quad m=2$

$\ell=3, r=3 \quad m=3$

$\Rightarrow$ output is 3

10 20 30 40 50 $\ell=1, r=5 \quad m=3$

$\ell=1 \quad r=2 \quad m=1$

$\ell=2 \quad r=2 \quad m=2$

$\ell=3 \quad r=2$

$\Rightarrow$ output is NIL

RUNTIME OF BINARY SEARCH/3

Analysis of algorithm

- How many times is the loop executed?
  - Each iteration the difference between r, l is cut in half
    - Initially the difference is n
    - Stops when it becomes 0
  - How many times do we have to cut n in half to get 0?
  - $\log_2 n$ — better than brute-force linear search (n)

RUNTIME OF BINARY SEARCH/3

Analysis of algorithm

- How many times is the loop executed?
  - Each iteration the difference between r, l is cut in half
    - Initially the difference is n
    - Stops when it becomes 0
  - How many times do we have to cut n in half to get 0?
  - $\log_2 n$ — better than brute-force linear search (n)
- Cost of linear versus cost of binary search
  - For $n = 0.2M$ one gets $\log_2 200'000 \approx 18$
  - For $n = 2M$ one gets $\log_2 2'000'000 \approx 21$
  - For $n = 20M$ one gets $\log_2 20'000'000 \approx 24$

log grows very slowly

EXERCISE SL02-3



The inner loop of the insertion sort algorithm uses a linear search to search backward through the sorted array a[1..j-1] to find the place where to insert the new element. Can we use a binary search instead to improve the worst case running time of insertion sort?

- We can quickly find the insertion point but we have to move elements anyway
- this does not improve runtime
- worst case time remains

EXERCISE SL02-4

Let A_1 be an unsorted array of length n_1 and A_2 a sorted array of length n_2. Both A_1 and A_2 store integers and do not contain duplicates.

Provide an algorithm that counts the number of integers in A_1 that are also in A_2. That is, given $A_1 = [5, 3, 7, 1, 9]$ and $A_2 = [1, 2, 3, 4]$, `cntOcc` (A_1, A_2) returns 2. The asymptotic complexity of your solution must be strictly less than $O(n_1 n_2)$. Furthermore, the arrays may not be changed and your solution may not use more than a constant amount of additional memory. Determine the asymptotic running time of your solution.

for i=1 to n1 do
  for j=1 to n2 do
    - if A1[i] = A2[j] then cnt++

runtime: $n_1 \cdot n_2$

EXERCISE SL02-4

```
Algo cntOcc(A1, A2)
    cnt = 0
    for i=1 to ul do
        if BinSearch(A2, A[i]) then cnt++
    return cnt
```

runtime: $a_1 \cdot \log_2(a_2)$

if both arrays are sorted:

A1 should be the smaller array

$20M \cdot \log(50) =$

$50 \cdot \log(20M) =$

TABLE OF CONTENTS — SL02

1. Algorithmic complexity

2. Correctness
   - Correctness notions, assertions, invariants
   - Correctness of binary search
   - Correctness of insertion sort

3. Asymptotic complexity

4. Special case analysis

CORRECTNESS/1

Basic considerations

- An algorithm is correct if it terminates and produces the desired output for any legal input

CORRECTNESS/1

Basic considerations

- An algorithm is correct if it terminates and produces the desired output for any legal input
- Automatic proof of correctness is not possible

CORRECTNESS/1

Basic considerations

- An algorithm is correct if it terminates and produces the desired output for any legal input
- Automatic proof of correctness is not possible
- There are practical techniques and rigorous formalisms that help to reason about the correctness of (parts of) algorithms

CORRECTNESS/2

Partial versus total



CORRECTNESS/2

Partial versus total



- **Partial correctness** (requires no proof of termination)
  If the point indicated with ⇓ is reached, then one gets the desired output

CORRECTNESS/2

Partial versus total



- **Partial correctness**
  If the point indicated with ⇓ is reached, then one gets the desired output
- **Total correctness**
  Indeed the point indicated with ⇓ is reached, and one gets the desired output

CORRECTNESS/3

Pre/post-conditions

- **Assertion**: statement about the state of execution
  - Example: $A[1], \ldots, A[i]$ form an increasing sequence

CORRECTNESS/3

Pre/post-conditions

- **Assertion**: statement about the state of execution
  - Example: $A[1], \ldots, A[i]$ form an increasing sequence
- To prove partial correctness one associates a number of assertions with specific checkpoints in the algorithm

CORRECTNESS/3

Pre/post-conditions

- **Assertion**: statement about the state of execution
  - Example: $A[1], \ldots, A[i]$ form an increasing sequence
- To prove partial correctness one associates a number of assertions with specific checkpoints in the algorithm
- **Preconditions**: assertions that must be valid before the execution of an algorithm or a subroutine (input)

CORRECTNESS/3

Pre/post-conditions

- **Assertion**: statement about the state of execution
  - Example: $A[1], \ldots, A[i]$ form an increasing sequence
- To prove partial correctness one associates a number of assertions with specific checkpoints in the algorithm
- **Preconditions**: assertions that must be valid before the execution of an algorithm or a subroutine (input)
- **Postconditions**: assertions that must be valid after the execution of an algorithm or a subroutine (output)

CORRECTNESS / 4

Example: Specify pre- and postconditions for an algorithm that determines the two smallest numbers in a sequence of numbers (given as an array).

precise? no!

CORRECTNESS / 4

Example: Specify pre- and postconditions for an algorithm that determines the two smallest numbers in a sequence of numbers (given as an array).

Precondition (INPUT): an array of integers $A[1..n]$, $n > 0$

Postcondition (OUTPUT): ($m1 \in A$, $m2 \in A$) such that either

- $m1 < m2$ if
  $\forall i \in [1..n] : m1 \leq A[i]$
  $\forall i \in [1..n] : A[i] \neq m1 \Rightarrow m2 \leq A[i]$, or
- $m2 = m1 = A[1]$ if
  $\forall i, j \in [1..n] : A[i] = A[j]$

$A=[5]$ $m_1 = m_2 = 5$

$A=[5,5,5]$ $m_1 = m_2 = 5$

$A=[]$ illegal input

$A=[5,7,5,5,8]$
$m_1 = 5$, $m_2 = 7$

CORRECTNESS/5

Loop invariants

- To show correctness of loop statements

CORRECTNESS/5

Loop invariants

- To show correctness of loop statements
- **Invariants**: assertions that are valid any time they are reached

CORRECTNESS/5

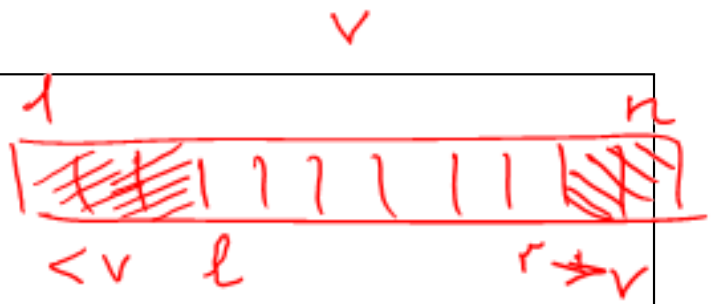
Loop invariants

- To show correctness of loop statements
- **Invariants**: assertions that are valid any time they are reached
- One must show three things about loop invariants
  - Initialization: it is true before the first iteration
  - Maintenance: if it is true before an iteration then it is true after that iteration
  - Termination: when the loop terminates, it gives a useful property that helps to show that the algorithm is correct

CORRECTNESS OF BINARY SEARCH/1

Goal: We want to show that binary search returns `NIL` if v is not in A.

```
l = 1; r = n;
m = ⌊(l+r)/2⌋;
while l≤r ∧ v≠A[m] do
    if v<A[m] then r = m-1 else l = m+1;
    m = ⌊(l+r)/2⌋;
if l≤r then return m else return NIL;
```



CORRECTNESS OF BINARY SEARCH/1

Goal: We want to show that binary search returns `NIL` if v is not in A.

```
l = 1; r = n;
m = ⌊(l+r)/2⌋;
while l≤r ∧ v≠A[m] do
    if v<A[m] then r = m-1 else l = m+1;
    m = ⌊(l+r)/2⌋;
if l≤r then return m else return NIL;
```

Invariant:

- $\forall i \in [1..l-1] : v > A[i]$, and

CORRECTNESS OF BINARY SEARCH/1

Goal: We want to show that binary search returns `NIL` if v is not in A.

```
l = 1; r = n;
m = ⌊(l+r)/2⌋;
while l≤r ∧ v≠A[m] do
    if v<A[m] then r = m-1 else l = m+1;
    m = ⌊(l+r)/2⌋;
if l≤r then return m else return NIL;
```

Invariant:

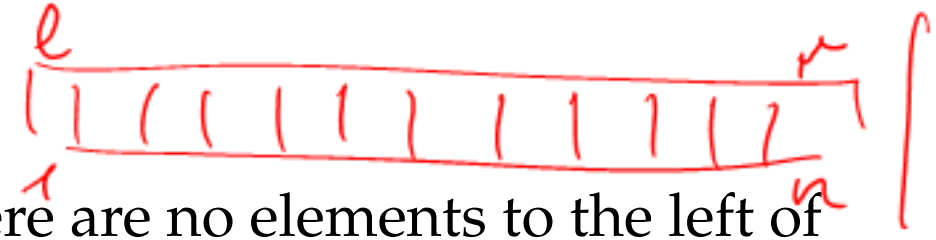
- $\forall i \in [1..l-1] : v > A[i]$, and
- $\forall i \in [r+1..n] : v < A[i]$

CORRECTNESS OF BINARY SEARCH/2

Invariant:

- $\forall i \in [1..l-1] : v > A[i]$
- $\forall i \in [r+1..n] : v < A[i]$

```
l = 1; r = n; m = ⌊(l+r)/2⌋;
while l≤r ∧ v≠A[m] do
    if v<A[m] then r = m-1 else l = m+1;
    m = ⌊(l+r)/2⌋;
if l≤r then return m else return NIL;
```



Initialization: $l = 1, r = n$

The invariant holds because there are no elements to the left of l and no elements to the right of r.

- $l = 1$ yields $\forall i \in [1..0] : A[i] < v$. This holds since the range [1..0] is empty.
- $r = n$ yields $\forall i \in [n+1..n] : A[i] > v$. This holds since the range [n+1..n] is empty.

Correctness of binary search/3

```
l = 1; r = n; m = ⌊(l+r)/2⌋;
while l≤r ∧ v≠A[m] do
    if v<A[m] then r = m-1 else l = m+1;
    m = ⌊(l+r)/2⌋;
if l≤r then return m else return NIL;
```

Invariant:

- $\forall i \in [1..l-1] : v > A[i]$
- $\forall i \in [r+1..n] : v < A[i]$

Maintenance: $l, r, m = \lfloor (l+r)/2 \rfloor$



Two cases:

- We have $A[m] \neq v$, $A[m] > v$, $r = m - 1$, A is sorted.
  This implies $\forall k \in [r+1..n] : A[k] > v$.
- We have $A[m] \neq v$, $A[m] < v$, $l = m + 1$, A is sorted.
  This implies $\forall k \in [1..l-1] : A[k] < v$.

Correctness of binary search/4

```
l = 1; r = n; m = ⌊(l+r)/2⌋;
while l≤r ∧ v≠A[m] do
    if v<A[m] then r = m-1 else l = m+1;
    m = ⌊(l+r)/2⌋;
if l≤r then return m else return NIL;
```

Invariant:

- $\forall i \in [1..l-1] : v > A[i]$
- $\forall i \in [r+1..n] : v < A[i]$

Termination: $l, r, l \leq r$

Two cases:

- $l = m + 1$ allows to conclude $\lfloor (l+r)/2 \rfloor + 1 > l$
- $r = m - 1$ allows to conclude $\lfloor (l+r)/2 \rfloor - 1 < r$

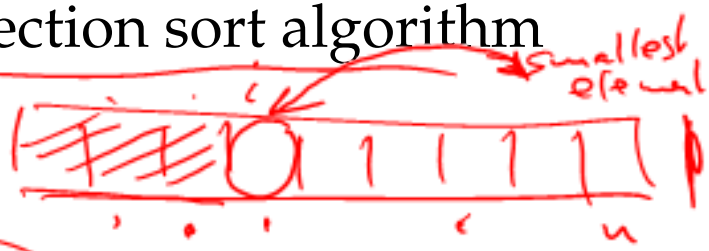
Thus, the range gets smaller during each iteration and the loop will terminate when $l \leq r$ no longer holds.

EXERCISE SL02-5

for i=1 to (n-1) do – position where smallest element shall go

n

Formulate the loop invariant for the outer loop of the selection sort algorithm. What happens if in the selection sort algorithm the outer loop runs up to n?



- element left of i are sorted
- – – – are the smallest elements of A

$\forall j \in [1..i-2]: A[j] \leq A[j+1]$

$\forall j \in [1..i-1] \forall k \in [i..n]: A[j] \leq A[k]$





FOR LOOPS

for i=1 to 5 5x
for i=1 to 2 2x
for i=1 to 1 1x

for i=1 to 0 0x
for i=1 to -2 0x

- In pseudo code the for loop can run **up** (we call this *up loop*) or **down** (we call this *down loop*).
- We have an up loop if the loop runs from low to high:
  - Example of an up loop: Array A[1..n]; for j=i to n do ...
  - If the start value of an index (start) is higher than the end value of an index (end) an up loop is executed zero times
  - Otherwise an up loop is executed (end - start + 1) times
- We have a down loop if the loop runs from high to low:
  - Example of a down loop: Array A[1..n]; for j=n to i do ...
  - If the start value of an index (start) is lower than the end value of the index (end) a down loop is executed zero times
  - Otherwise a down loop is executed (start - end + 1) times
- The value of the loop variable is undefined after the loop.

CORRECTNESS OF INSERTION SORT / 1

2 loop's

```
for i = 2 to n do
    j = i-1; t = A[i];
    while j>0 ∧ A[j]>t do
        A[j+1] = A[j];
        j = j-1;
    A[j+1] = t;
```

CORRECTNESS OF INSERTION SORT / 1

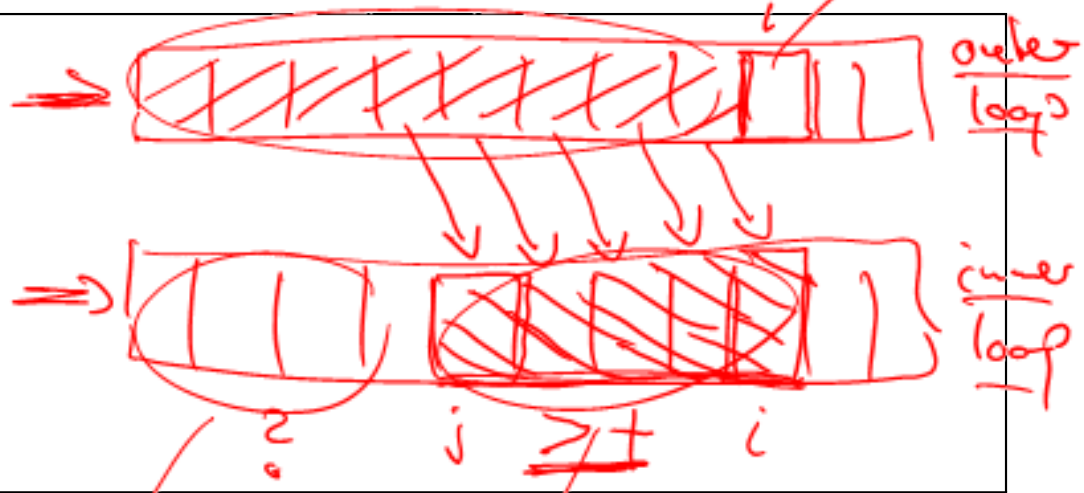
```
for i = 2 to n do
    j = i-1; t = A[i];
    while j>0 ∧ A[j]>t do
        A[j+1] = A[j];
        j = j-1;
    A[j+1] = t;
```

Loop invariant

- Outer loop: $\begin{cases} A[1 \ldots i-1] \text{ sorted} \\ A[1 \ldots i-1] \in A^{orig} \end{cases}$

CORRECTNESS OF INSERTION SORT / 1

```
for i = 2 to n do
    j = i-1; t = A[i];
    while j>0 ∧ A[j]>t do
        A[j+1] = A[j];
        j = j-1;
    A[j+1] = t;
```



Loop invariant

- Outer loop:
$$\begin{cases} A[1 \ldots i-1] \text{ sorted} \\ A[1 \ldots i-1] \in A^{orig} \end{cases}$$
- Inner loop:
$$\begin{cases} A[1 \ldots j], t, A[j+1 \ldots i-1] \\ t < A[k] \text{ for } j+1 \le k \le i-1 \\ A[1 \ldots j] \circ A[j+1 \ldots i-1] \text{ sorted} \end{cases}$$

(concatenate)

Correctness of insertion sort / 2

Outer loop:

- $A[1..i-1]$ is sorted
- $A[1..i-1] \in A^{orig}$

Inner loop:

- $A[1..j], t, A[j+1 \ldots i-1]$
- $t < A[k]$ for $j+1 \leq k \leq i-1$
- $A[1..j] \circ A[j+1..i-1]$ sorted

```
for i = 2 to n do
    j = i-1; t = A[i];
    while j>0 ∧ A[j]>t do
        A[j+1] = A[j];
        j = j-1;
    A[j+1] = t;
```

Initialization:

- $i = 2$: invariant holds; A[1..1] is trivially sorted
- $j = i-1$: $A[1..i-1], t, A[i..i-1], t = A[i]$
  $A[i..i-1]$ is empty (and thus trivially sorted)
  $A[1..i-1]$ is sorted (invariant of outer loop)

CORRECTNESS OF INSERTION SORT / 3

Outer loop:

- $A[1..i-1]$ is sorted
- $A[1..i-1] \in A^{orig}$

Inner loop:

- $A[1..j], t, A[j+1 \ldots i-1]$
- $t < A[k]$ for $j+1 \leq k \leq i-1$
- $A[1..j] \circ A[j+1..i-1]$ sorted

```
for i = 2 to n do
    j = i-1; t = A[i];
    while j>0 ∧ A[j]>t do
        A[j+1] = A[j];
        j = j-1;
    A[j+1] = t;
```

Maintenance:

- $A[1..i-1]$ is sorted + insert $A[i]$ implies $A[1..i]$ is sorted
- $A[1..j-1], t, A[j], j+1..i-1]$ satisfies condition since $A[j] > t$ and $A[1..i-1]$ is sorted

CORRECTNESS OF INSERTION SORT / 4

Outer loop:

- $A[1..i-1]$ is sorted
- $A[1..i-1] \in A^{orig}$

Inner loop:

- $A[1..j], t, A[j+1 \ldots i-1]$
- $t < A[k]$ for $j+1 \leq k \leq i-1$
- $A[1..j] \circ A[j+1..i-1]$ sorted

```
for i = 2 to n do
    j = i-1; t = A[i];
    while j>0 ∧ A[j]>t do
        A[j+1] = A[j];
        j = j-1;
    A[j+1] = t;
```

Termination:

- outer loop: $i = n+1$ and $A[1..n]$ is sorted
- $A[j] \leq t$: $A[1..j], t, A[j+1..i-1] = A[1..i]$ is sorted
- $j = 0$: $A[t, 1..i-1] = A[1..i]$ is sorted

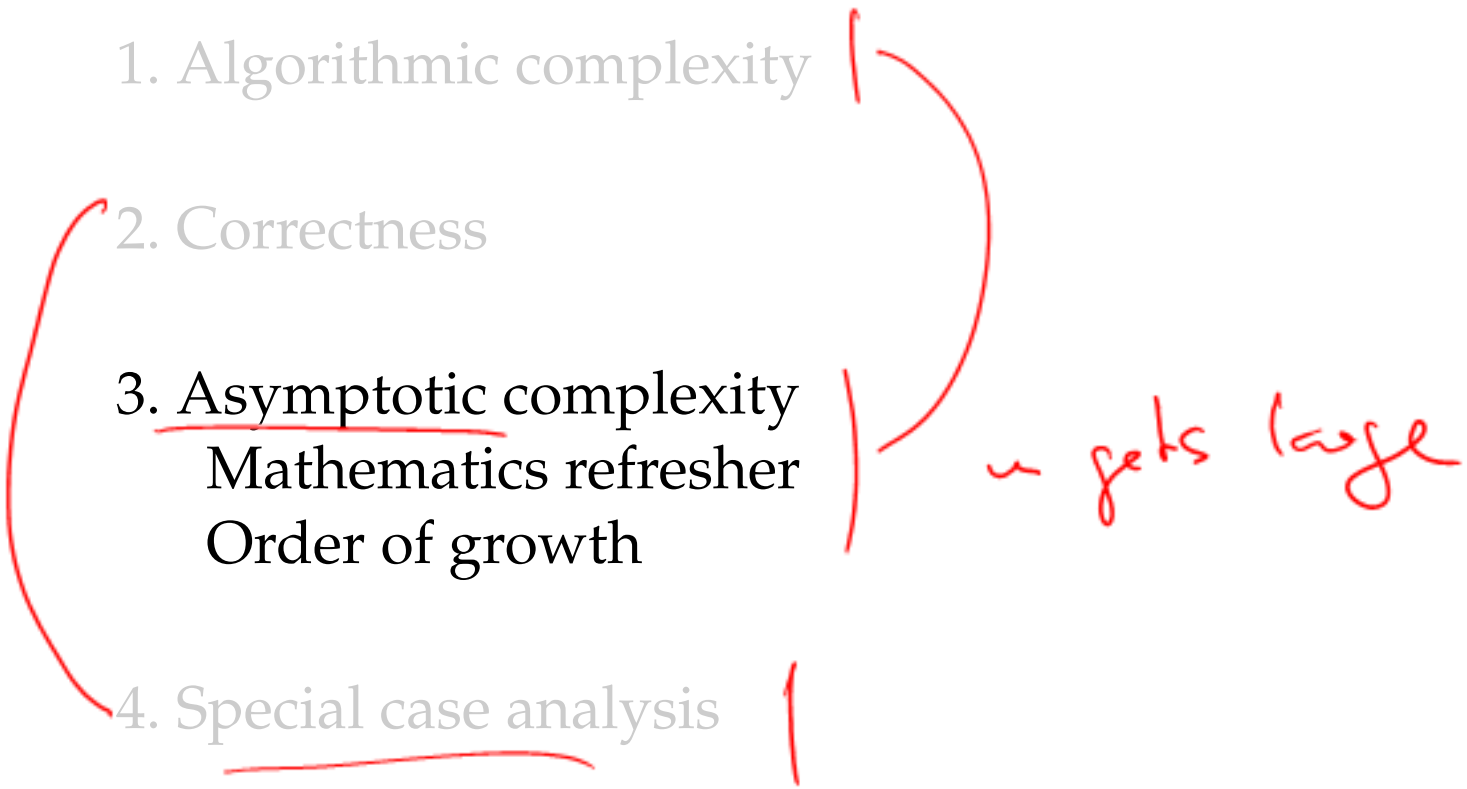
TABLE OF CONTENTS — SL02

1. Algorithmic complexity

2. Correctness

3. Asymptotic complexity
   Mathematics refresher
   Order of growth

4. Special case analysis



ASYMPTOTIC COMPLEXITY

$n \to \infty$ (n very large)

Asymptotic analysis

- Goal: simplify the analysis of running time

Asymptotic complexity

Asymptotic analysis

- Goal: simplify the analysis of running time
- Approach: getting rid of unnecessary details
  - Affected by specific implementation and hardware
  - "Rounding" of numbers: $1000213 \approx 1000000$
  - "Rounding" of functions: $3n^2 + 2n \approx n^2$

Asymptotic complexity

Asymptotic analysis

- Goal: simplify the analysis of running time
- Approach: getting rid of unnecessary details
  - Affected by specific implementation and hardware
  - "Rounding" of numbers: $1000213 \approx 1000000$
  - "Rounding" of functions: $3n^2 + 2n \approx n^2$
- Fundamental concern: how the running time increases with the size of the input in the limit

Asymptotic complexity

Asymptotic analysis

- Goal: simplify the analysis of running time
- Approach: getting rid of unnecessary details
  - Affected by specific implementation and hardware
  - "Rounding" of numbers: $1000213 \approx 1000000$
  - "Rounding" of functions: $3n^2 + 2n \approx n^2$
- Fundamental concern: how the running time increases with the size of the input in the limit
- Asymptotically more efficient algorithms are best for all but small inputs

MATHEMATICS REFRESHER/1

Summations

- Simple manipulations
  - $\sum_{i=1}^{n} ca_i = c \sum_{i=1}^{n} a_i$
  - $\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$

MATHEMATICS REFRESHER/1

Summations

- Simple manipulations
  - $\sum_{i=1}^{n} ca_i = c \sum_{i=1}^{n} a_i$
  - $\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$
- Arithmetic progression
  - $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$

MATHEMATICS REFRESHER/1

Summations

- Simple manipulations
  - $\sum_{i=1}^{n} ca_i = c \sum_{i=1}^{n} a_i$
  - $\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i$
- Arithmetic progression
  - $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$

MATHEMATICS REFRESHER/2

- Geometric progression
  - given an integer n_0 and a real number $0 < a \neq 1$:

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 \cdots + a^n = \frac{1 - a^{n+1}}{1 - a} = \frac{a^{n+1} - 1}{a - 1}$$

MATHEMATICS REFRESHER/2

- Geometric progression
  - given an integer n_0 and a real number $0 < a \neq 1$:

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 \cdots + a^n = \frac{1 - a^{n+1}}{1 - a}$$

  - given an integer n_0 and a real number $|a| < 1$:

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1 - a}$$

MATHEMATICS REFRESHER/3

Proof by induction of $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

- Base case: $\sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$ $n = 1$

MATHEMATICS REFRESHER/3

Proof by induction of $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

- Base case: $\sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$
- Inductive hypothesis: $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$ $n-1$

MATHEMATICS REFRESHER/3

Proof by induction of $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

- Base case: $\sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$
- Inductive hypothesis: $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$
- Inductive step (uses inductive hypothesis):

$$\sum_{i=1}^{n} i = n + \sum_{i=1}^{n-1} i = n + \frac{(n-1)n}{2} = \frac{n(n+1)}{2}$$

Mathematics refresher/4

Logarithms and powers

- $\log_a(bc) = \log_a b + \log_a c$
- $\log_a b = \log_c b / \log_c a$
- $\log_a b^c = c \log_a b$
- $a^m a^n = a^{m+n}$
- $a^{mn} = (a^m)^n$
- $a^{\log_a b} = b$
- $\lg b = ldb = \log_2 b$
- Notation: $\log_a b = {}^a\log b$

$\log_2(n)$

$= \frac{\log_5(n)}{\log_5(2)}$

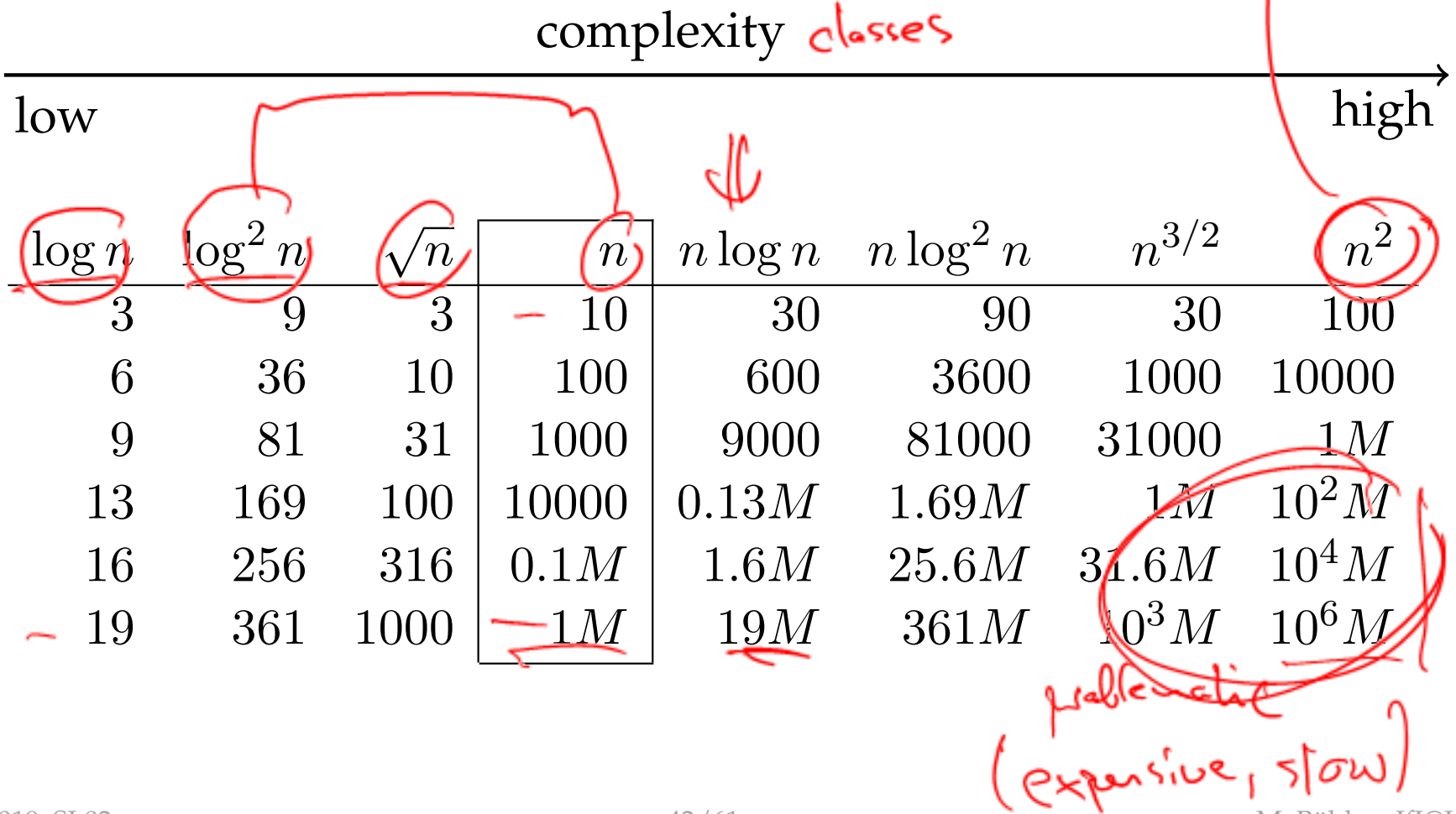
$= \frac{1}{\log_5(2)} \cdot \log_5(n)$

$= c \cdot \log_5(n)$

$\Rightarrow$ changing the base of the log = constant factor in complexity

MATHEMATICS REFRESHER/5

Some typical functions
From $n = 10$ up to $n = 1M$



$\log n$	$\log^2 n$	$\sqrt{n}$	n	$n \log n$	$n \log^2 n$	$n^{3/2}$	n^2
3	9	3	10	30	90	30	100
6	36	10	100	600	3600	1000	10000
9	81	31	1000	9000	81000	31000	$1M$
13	169	100	10000	$0.13M$	$1.69M$	$1M$	10^2M
16	256	316	$0.1M$	$1.6M$	$25.6M$	$31.6M$	10^4M
19	361	1000	$1M$	$19M$	$361M$	10^3M	10^6M

Order of growth / 1

Big-O notation

- Asymptotic upper bound

Order of growth / 1

Big-O notation

- Asymptotic upper bound
- Used for worst-case analysis

ORDER OF GROWTH / 1

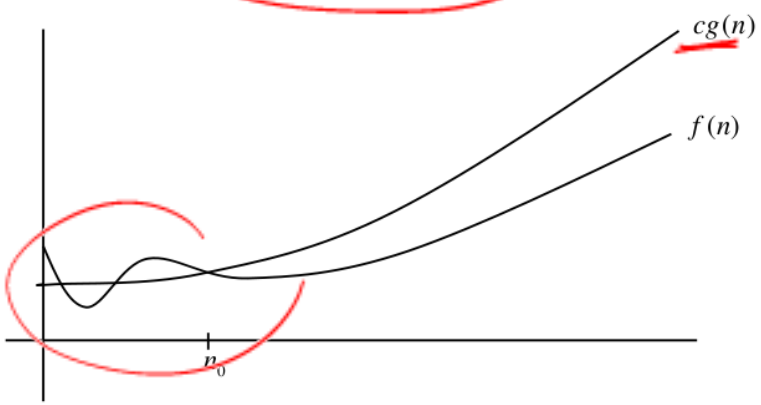
Big-O notation

- Asymptotic upper bound
- Used for worst-case analysis
- $f(n)$ and $g(n)$ are functions over non-negative integers

Order of growth / 1

Big-O notation

- Asymptotic upper bound
- Used for worst-case analysis
- $f(n)$ and $g(n)$ are functions over non-negative integers
- $f(n) = O(g(n))$ if and only if there exist constants $c > 0$ and $n_0 > 0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$



EXERCISE SL02-6

Prove or disprove the following statements:

1. $2^{n+1} = O(2^n)$
2. $2^{2^n} = O(2^n)$
3. $2n^{1.5} = O(n \lg n)$

1. $2^{n+1} \leq c \cdot 2^n \quad \forall n \geq n_0, c > 0$

$2 \cdot 2^n \leq c \cdot 2^n$

$2 \leq c$ holds for $c = 2$

2. $2^{2^n} \leq c \cdot 2^n$

$2^{2^n} \leq c \cdot 2^n$

$2^n \cdot 2^n \leq c \cdot 2^n$

$2^n \not\leq c$

no

3. $2n^{3/2} \leq cn \lg n$

$2\sqrt{n^3} \leq cn \lg n$

$n^3 \leq \frac{c^2}{4} n^2 \lg^2 n$

$n \not\leq \frac{c^2}{4} \lg^2 n$

no

Order of growth / 2

Big-Ω notation

- Asymptotic lower bound

Order of growth / 2

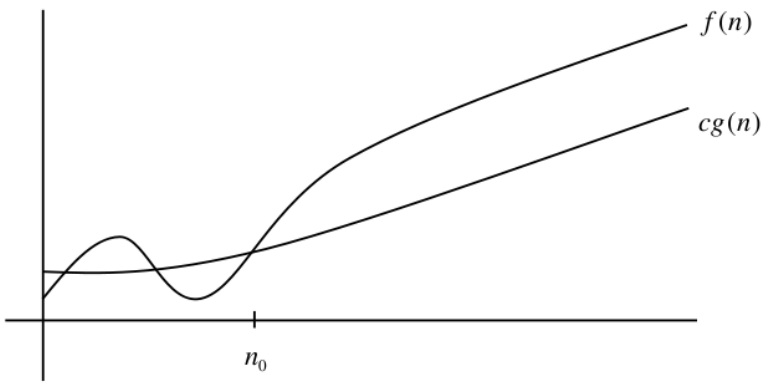
Big-Ω notation

- Asymptotic lower bound
- Used for best-case analysis

ORDER OF GROWTH / 2

Big-Ω notation

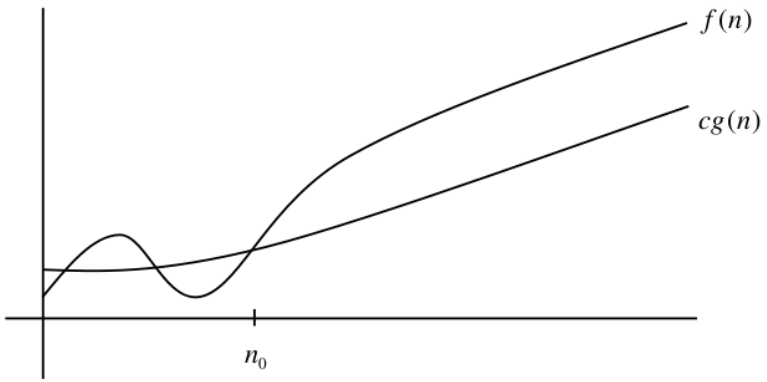
- Asymptotic lower bound
- Used for best-case analysis
- $f(n) = \Omega(g(n))$ if and only if there exist constants $c > 0$ and $n_0 > 0$ such that $cg(n) \leq f(n)$ for $n \geq n_0$



ORDER OF GROWTH/2

Big-Ω notation

- Asymptotic lower bound
- Used for best-case analysis
- $f(n) = \Omega(g(n))$ if and only if there exist constants $c > 0$ and $n_0 > 0$ such that $cg(n) \leq f(n)$ for $n \geq n_0$



- Little-o/ω: non-tight analogues of big-oh/omega

Order of growth / 3

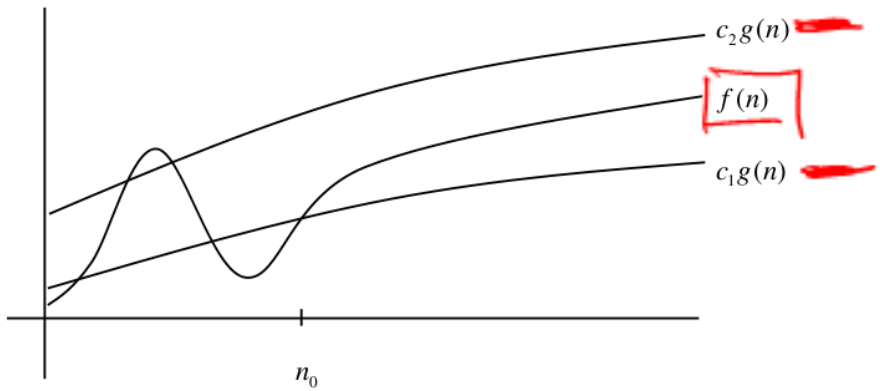
Big Θ notation

- Asymptotic tight bound

ORDER OF GROWTH / 3

Big Θ notation

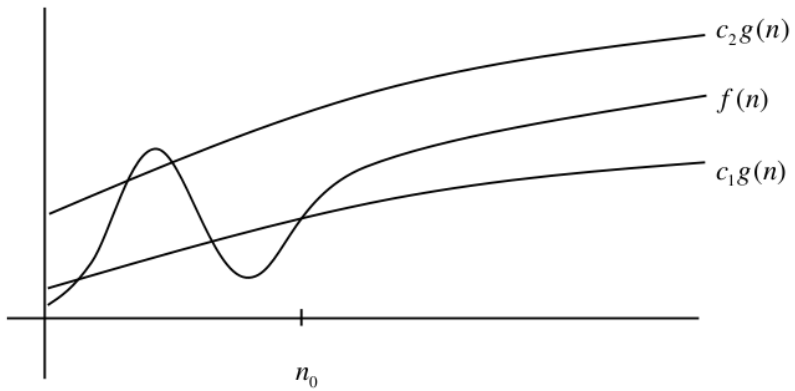
- Asymptotic tight bound
- $f(n) = \Theta(g(n))$ if and only if there exist constants $c_1 > 0$, $c_2 > 0$, and $n_0 > 0$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for $n \geq n_0$



ORDER OF GROWTH / 3

Big Θ notation

- Asymptotic tight bound
- $f(n) = \Theta(g(n))$ if and only if there exist constants $c_1 > 0$, $c_2 > 0$, and $n_0 > 0$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for $n \geq n_0$

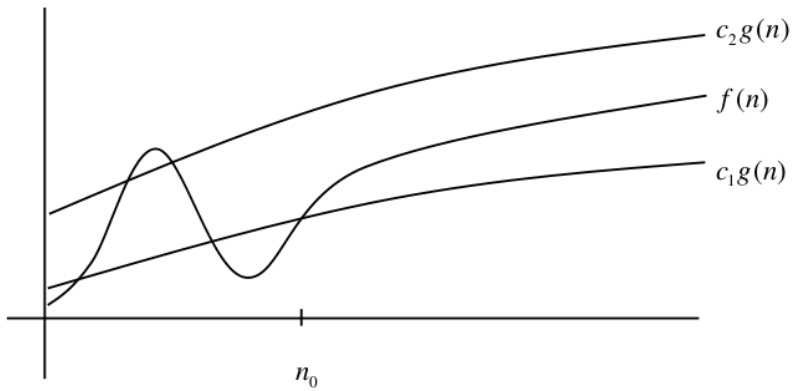


- $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

ORDER OF GROWTH / 3

Big Θ notation

- Asymptotic tight bound
- $f(n) = \Theta(g(n))$ if and only if there exist constants $c_1 > 0$, $c_2 > 0$, and $n_0 > 0$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for $n \geq n_0$



- $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- $O(f(n))$ is sometimes (ab)used as notation for $\Theta(f(n))$

EXERCISE SL02-7

Determine the asymptotic complexities of the formulas:

1. $0.001n^2 + 70'000n$ $= \Theta(n^2)$
2. $2^n + n^{10'000}$ $= \Theta(2^n)$
3. $n^k + c^n$ $= \Theta(c^n)$, $c > 1$
4. $\log_k n + n^e$ $= \Theta(n^e)$
5. $2^n + 2^{n/2}$ $= \Theta(2^n)$
6. $n^{\log c} + c^{\log n}$ ~~$= \Theta(n^{\log c}) = \Theta(c^{\lg n})$~~
7. $2n^3 + 100n \log n + 5$ $= \Theta(n^3)$
8. $30 \log n + \log 2^n$ $= \Theta(n)$
9. $4^{\log n} + n^2$

$n^{\lg c} = n^{\log_4 c} = (n^{\log_4 c})^{1/\log_4 x} = c^{1/\log_4 x} = c^{\log_4 n / \log_4 x} = c^{\lg n / \lg x}$

$\log_4 2 = \frac{\log_4 2}{\log_4 10} = \log_{10} 4 \approx 0.6$

$\Theta(n^2)$

Order of growth / 4

Useful analogy

- $f(n) = O(g(n)) \quad \leftrightarrow \quad f \leq g$
- $f(n) = o(g(n)) \quad \leftrightarrow \quad f < g$
- $f(n) = \Omega(g(n)) \quad \leftrightarrow \quad f \geq g$
- $f(n) = \omega(g(n)) \quad \leftrightarrow \quad f > g$
- $f(n) = \Theta(g(n)) \quad \leftrightarrow \quad f = g$

Order of growth / 4

$O(3n+5) = \{1, 3n, n, n-6, n/7, \sqrt{n}, \lg n, \ldots\}$

Useful analogy

- $f(n) = O(g(n)) \quad \leftrightarrow \quad f \leq g$
- $f(n) = o(g(n)) \quad \leftrightarrow \quad f < g$
- $f(n) = \Omega(g(n)) \quad \leftrightarrow \quad f \geq g$
- $f(n) = \omega(g(n)) \quad \leftrightarrow \quad f > g$
- $f(n) = \Theta(g(n)) \quad \leftrightarrow \quad f = g$

$3n+5 = O(n)$

- Abuse of notation: $f(n) = O(g(n))$ actually means $f(n) \in O(g(n))$ since $O(g(n))$ is the set of all functions for which there exists constants $c > 0$ and $n_0 > 0$ such that ...

$f(n) \leq c \cdot g(n)$

Order of growth / 5

Practical issues

- Drop multiplication constants
- Consider only the leading term of a formula
- Examples:
  - $7n - 3$ is $O(n)$
  - $50n \log n$ is $O(n \log n)$
  - $80n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$

ORDER OF GROWTH / 5

Practical issues

- Drop multiplication constants
- Consider only the leading term of a formula
- Examples:
  - $7n - 3$ is $O(n)$
  - $50n \log n$ is $O(n \log n)$
  - $80n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$

- Lower-order terms are relatively insignificant

Order of growth / 5

Practical issues

- Drop multiplication constants
- Consider only the leading term of a formula
- Examples:
  - $7n - 3$ is $O(n)$
  - $50n \log n$ is $O(n \log n)$
  - $80n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$

- Lower-order terms are relatively insignificant
- Although $50n \log n$ is $O(n^5)$, it is expected that an approximation is of smallest possible order

Comparison of Running Times

Determining the maximal problem size:

Running Time T(n) in μs	1 second	1 minute	1 hour
$400n$	2500	150'000	9'000'000
$20nlogn$	4096	166'666	7'826'087
$2n^2$	707	5477	42'426
n^4	31	88	244
2^n	19	25	31

encryption

big problems cannot be solved anymore

EXERCISE SL02-8

Provide an algorithm that selects the k-th smallest integer from an unsorted array A of length n that does not contain duplicates. For instance, given $k = 3$ and $A = [12, 4, 10, 8, 9, 33, 2, 18]$, `select_kth` (A, k) returns 8. Determine the asymptotic complexity of your algorithm.

```
selectKth(A,k)
for i = 1 to k
    m = i
    for j = i+1 to n do
        if A[j] < A[m] then m = j
    exchange A[i] and A[m]
return A[k]
```

$$\sum_{i=1}^{k} \sum_{j=i+1}^{n} 1 \quad (= \text{number of lines if shown is executed})$$

$$= \sum_{i=1}^{k} (n - i)$$

$$= kn - \sum_{i=1}^{k} i$$

$$= kn - \frac{k(k+1)}{2} = \Theta(kn)$$

$k = 1 : n-1$

$k = n : \frac{n^2}{2} - \frac{n}{2}$

(= selection sort)

n or 1 though

TABLE OF CONTENTS — SL02

1. Algorithmic complexity

2. Correctness

3. Asymptotic complexity

4. Special case analysis
   Special cases
   Correctness of binary search variant

Special case analysis/1

Basic approach

- Checking correctness of a program

Special case analysis/1

Basic approach

- Checking correctness of a program
- Consider all possible extreme cases

Special case analysis/1

Basic approach

- Checking correctness of a program
- Consider all possible extreme cases
- Make sure the solution works in these cases

Special case analysis/1

Basic approach

- Checking correctness of a program
- Consider all possible extreme cases
- Make sure the solution works in these cases
- Problem is reduced to identifying all the special cases

Special case analysis/1

Basic approach

- Checking correctness of a program
- Consider all possible extreme cases
- Make sure the solution works in these cases
- Problem is reduced to identifying all the special cases
- Related to problem specification (input/output relation)

Special case analysis/2

Special cases

- Data structure (array, list, file, etc.)
  - Empty, single element, completely filled

Special case analysis/2

Special cases

- Data structure (array, list, file, etc.)
  - Empty, single element, completely filled
- Particular values, border of domain
  - Zero, empty string, negative numbers, etc.

Special case analysis/2

Special cases

- Data structure (array, list, file, etc.)
  - Empty, single element, completely filled
- Particular values, border of domain
  - Zero, empty string, negative numbers, etc.
- Calling of functions (procedures)
  - Entering function, termination of function

Special case analysis/2

Special cases

- Data structure (array, list, file, etc.)
  - Empty, single element, completely filled
- Particular values, border of domain
  - Zero, empty string, negative numbers, etc.
- Calling of functions (procedures)
  - Entering function, termination of function
- Control and loop statements
  - Start of loop, end of loop, 1st iteration

SPECIAL CASE ANALYSIS/3

Simple example

check if array A is sorted

Algo: Sorted1(A)

Input: array $A[1..n]$ of integers
Output: TRUE if A is sorted, FALSE otherwise

for $i = 1$ **to** n **do**
  if $A[i] \geq A[i+1]$ **then return** FALSE;
return TRUE

SPECIAL CASE ANALYSIS/3

Simple example

Algo: Sorted1(A)

Input: array $A[1..n]$ of integers
Output: TRUE if A is sorted, FALSE otherwise

for $i = 1$ **to** n **do**
  if $A[i] \geq A[i+1]$ **then return** FALSE;
return TRUE

- Loop special cases:
  - $i = 1$ (start)
  - $i = n$

SPECIAL CASE ANALYSIS/3

Simple example

Algo: Sorted1(A)

Input: array $A[1..n]$ of integers
Output: TRUE if A is sorted, FALSE otherwise

```
for i = 1 to n do
    if A[i]≥A[i+1] then return FALSE;
return TRUE
```

- Loop special cases:
  - $i = 1$ (start)
  - $i = n$
- Data structure special cases:
  - empty array
  - single element array

SPECIAL CASE ANALYSIS/4

Algo: Sorted2(A)

Input: array $A[1..n]$ of integers
Output: TRUE if A is sorted, FALSE otherwise

for $i = 1$ **to** $n-1$ **do**
  if $A[i] > A[i+1]$ **then return** FALSE;
return TRUE

SPECIAL CASE ANALYSIS / 4

Algo: Sorted2(A)

Input: array $A[1..n]$ of integers
Output: TRUE if A is sorted, FALSE otherwise

for *i = 1* **to** *n-1* **do**
  if *A[i]>A[i+1]* **then return** FALSE;
return TRUE

- start of loop, i=1: A[1] > A[2] ✓
- end of loop, i=n-1: A[n-1] > A[n] ✓
- first iteration from i=1 to i=2: A[2] > A[3]
- empty array, n=0, A=[]: for i=1 to -1 ✗ ✓
- one element array, n=1, A=[1]: for i=1 to 0 ✗
- two element array, n=2, A=[1,0]: for i=1 to 1 A[1] > A[2] ✓
- A=[1,1,1], A=[1,2,3], A=[-1,0,1,-3]: ✓
- A=[3,2,1]: ??

CORRECTNESS OF BINARY SEARCH VARIANT/1

- other special cases:
  - o odd number of elements
  - o 1 element array

Use special cases analysis to analyze the following algorithm.

Algo: BinSearch2(A,v)

```
l = 1; r = n;
repeat
    m = ⌊(l+r)/2⌋;
    if A[m]==v then return m;
    else if A[m]>v then r = m-1;
    else l = m+1;
until l≥r;
return NIL;
```

a. A=[5,7], v=3
b. A=[5,7], v=5
c. A=[5,7], v=6
d. A=[5,7], v=7
e. A=[5,7], v=8

CORRECTNESS OF BINARY SEARCH VARIANT/2

b., c., e. OK

Use special cases analysis to analyze the following algorithm.

Algo: BinSearch3(A,v)

```
l = 1; r = n;
while l<r do
    m = ⌊(l+r)/2⌋;
    if A[m]≤v then l = m+1 else r = m;
if A[l-1]==v then return l-1 else return NIL;
```

a. A=[5,7], v=3
l=1, r=2
l=1, r=1
A[0] does not exist (index range exceed)

d. A=[5,7], v=7
l=1, r=2
l=2, r=2 NIL

CORRECTNESS OF BINARY SEARCH VARIANT/3

Use special cases analysis to analyze the following algorithm.

Algo: BinSearch3(A,v)

```
l = 1; r = n;
while l≤r do
    m = ⌊(l+r)/2⌋;
    if A[m]≤v then l = m+1 else r = m;
if A[l-1]==v then return l-1 else return NIL;
```

e. A=[5,7], v=3

l=1, r=2

l=1, r=1

l=1, r=1

⇒ ∞ loop

EXERCISE SL02-9

Use special case analysis to determine the correctness of BinarySearch4 and BinarySearch5.

Algo: BinSearch4(A,q)

```
l = 1; r = n;
while l≤r do
    m = ⌊(l+r)/2⌋;
    if A[m]==q then return m;
    else if A[m]>q then r = m-1;
    else l = m+1;
return NIL;
```



Algo: BinSearch5(A,q)

```
l = 1; r = n;
while l<r do
    m = ⌊(l+r)/2⌋;
    if A[m]==q then return m;
    else if A[m]>q then r = m-1;
    else l = m+1;
return NIL;
```

d. $A=[5,7]$, $v=7$

$l=1$, $r=2$

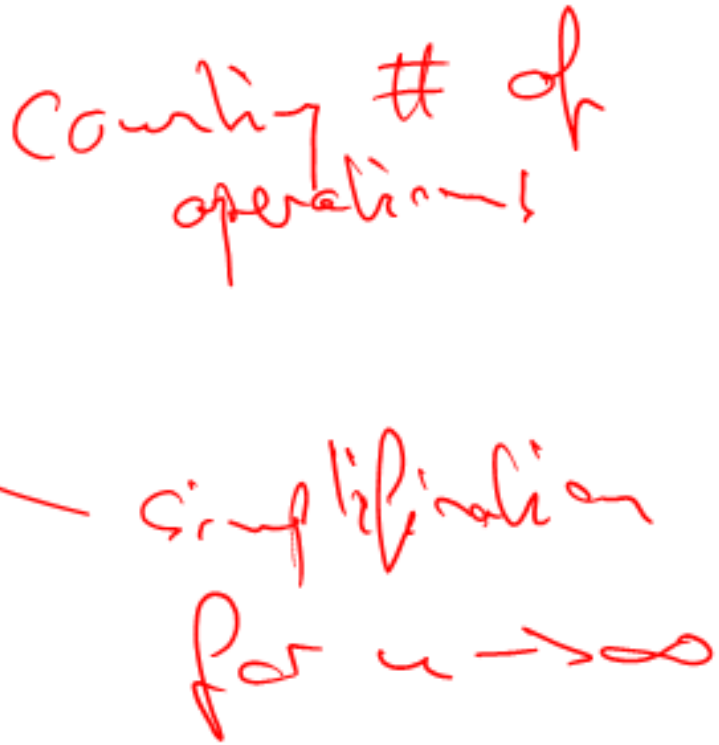
$l=2$, $r=2$

NIL ☹

Outlook

This week

- Algorithmic complexity
- Correctness of algorithms
- Asymptotic analysis
- Special case analysis



Outlook

This week

- Algorithmic complexity
- Correctness of algorithms
- Asymptotic analysis
- Special case analysis

Next week

- Divide and conquer
- Merge sort, tiling
- Recurrences

Informatik II

Divide and Conquer, Recurrences
— SL03 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

TABLE OF CONTENTS — SL03

1. Divide and conquer
   Merge sort
   Tromino tiling
2. Recurrences

recursive design of algorithms

analysis of runtime of recursive algs

DIVIDE AND CONQUER/1

Designing algorithms

- Recursive algorithms: divide-and-conquer approach

DIVIDE AND CONQUER/1

Designing algorithms

- Recursive algorithms: divide-and-conquer approach
- Principle: if problem size is small enough to be solved trivially then solve it; else

1. **Divide** problem into a number of subproblems
2. **Conquer** subproblems by solving them recursively
3. **Combine** solutions of the subproblem into solution for the original problem

DIVIDE AND CONQUER/2

Finding decomposition

- Requires some practice and is the key part

DIVIDE AND CONQUER/2

Finding decomposition

- Requires some practice and is the key part
- Fundamental properties of a decomposition
  - It reduces the problem to smaller problems
  - Often, these smaller problems are the same as the original problem
  - A sequence of decompositions eventually yields the base case (trivial solution)
  - It must contribute to solving the original problem

MERGE SORT/1

Rough idea

- Closely follows the divide-and-conquer paradigm

MERGE SORT/1

Rough idea

- Closely follows the divide-and-conquer paradigm
- To sort an n-element sequence proceed as follows
  - **Divide**: divide the sequence into two $n/2$-element sequences
  - **Conquer**: sort the two sequences recursively using merge sort
  - **Combine**: merge the two sorted sequences to produce the solution

MERGE SORT/1

Rough idea

- Closely follows the divide-and-conquer paradigm
- To sort an n-element sequence proceed as follows
  - **Divide**: divide the sequence into two $n/2$-element sequences
  - **Conquer**: sort the two sequences recursively using merge sort
  - **Combine**: merge the two sorted sequences to produce the solution
- Recursion stops when the sequence that must be sorted has length 1.

MERGE SORT / 2

Base case: array of size 1, i.e., $l = r$

The algorithm

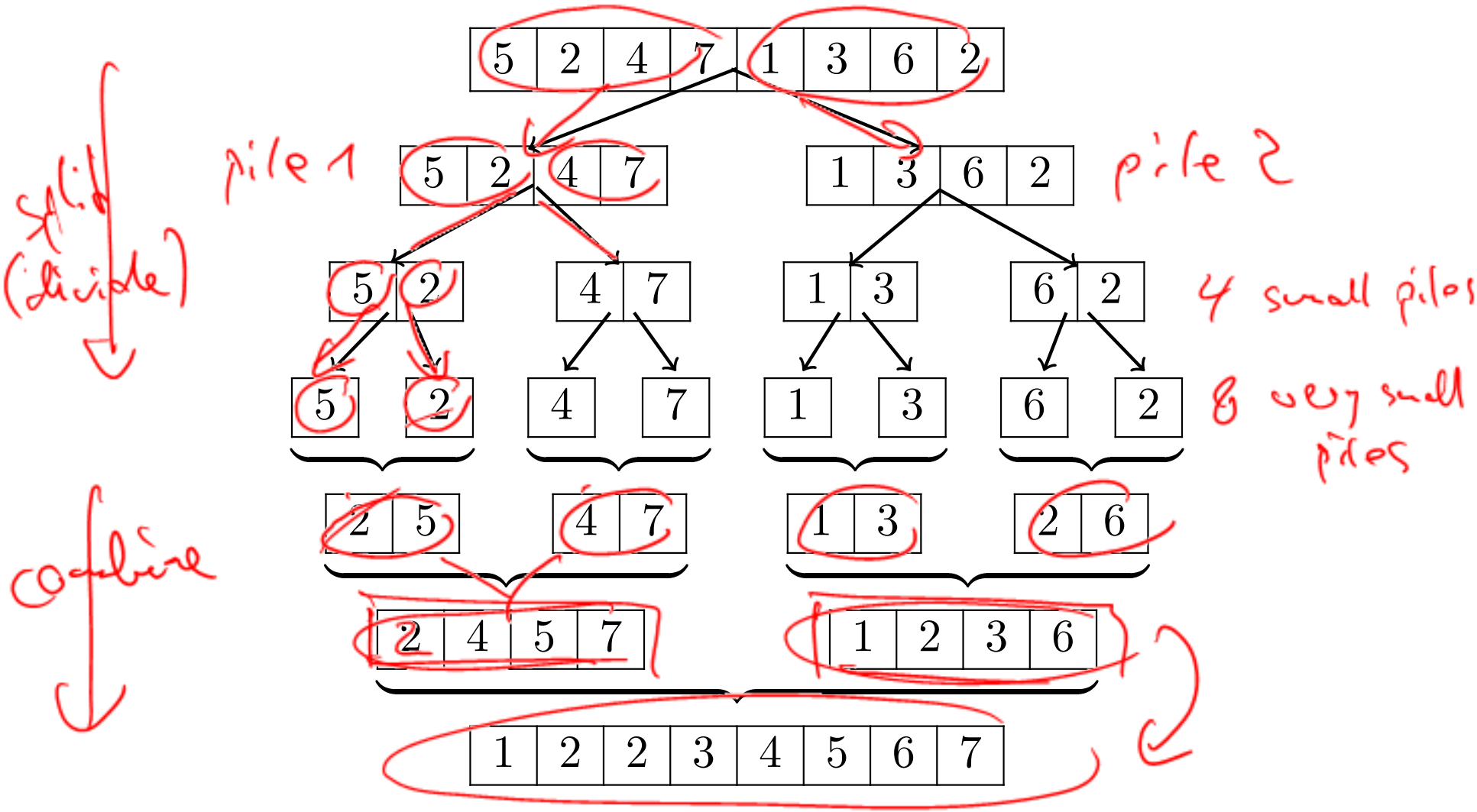
Algo: MergeSort(A,l,r)

Input: array A[l..r]
Output: permuted array A[l..r] that is sorted in increasing order

```
if l<r then
    m = ⌊(l+r)/2⌋;
    MergeSort(A,l,m);
    MergeSort(A,m+1,r);
    Merge(A,l,r,m);
```

MERGE SORT / 3

Sorting $\langle 5, 2, 4, 7, 1, 3, 6, 2 \rangle$: overview



MERGE SORT / 4

Goal: $[7, 14, 15, 20], [3, 6, 16, 22] \Rightarrow [3, 6, 7, 14, 15, 16, 20, 22]$

Algo: Merge(A,l,r,m)

Input: array A; indexes l, r and m from MergeSort
Output: A[l..r] is sorted in increasing order

```
for i = l to m do B[i] = A[i];
for i = m+1 to r do B[r+m-i+1] = A[i];
i = l; j= r;
for k = l to r do
    if B[i]<B[j] then A[k] = B[i]; i = i+1;
    else A[k] = B[j]; j = j-1;
```

The following slides show the state at the beginning of the third for loop.

MERGE SORT / 5

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- Before the (last) call of Merge:

$A =$

2	4	5	7	1	2	3	6

MERGE SORT / 5

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- Before the (last) call of Merge:

$A =$

2	4	5	7	1	2	3	6

- $i = 1, j = 8, k = 1$

$B =$

2	4	5	7	6	3	2	1

MERGE SORT / 5

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- Before the (last) call of Merge:

$A =$

2	4	5	7	1	2	3	6

- $i = 1, j = 8, k = 1$

$B =$

2	4	5	7	6	3	2	1

$A =$

.	.	.	.	.	.	.	.

MERGE SORT / 5

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- Before the (last) call of Merge:

$A =$

2	4	5	7	1	2	3	6

- $i = 1, j = 8, k = 1$

$B =$

2	4	5	7	6	3	2	1

$A =$

.	.	.	.	.	.	.	.

- $i = 1, j = 7, k = 2$

$B =$

2	4	5	7	6	3	2	1

MERGE SORT / 5

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- Before the (last) call of Merge:

$A =$

2	4	5	7	1	2	3	6

- $i = 1, j = 8, k = 1$

$B =$

2	4	5	7	6	3	2	1

$A =$

.	.	.	.	.	.	.	.

- $i = 1, j = 7, k = 2$

$B =$

2	4	5	7	6	3	2	1

$A =$

1	.	.	.	.	.	.	.

Merge sort / 6

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 1, j = 6, k = 3$

$B =$

2	4	5	7	6	3	2	1

Merge sort / 6

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 1, j = 6, k = 3$

$B =$

2	4	5	7	6	3	2	1

$A =$

1	2	.	.	.	.	.	.

Merge sort / 6

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 1, j = 6, k = 3$

$B =$

2	4	5	7	6	3	2	1

$A =$

1	2	.	.	.	.	.	.

- $i = 2, j = 6, k = 4$

$B =$

2	4	5	7	6	3	2	1

Merge sort / 6

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 1, j = 6, k = 3$

$B =$

2	4	5	7	6	3	2	1

$A =$

1	2	.	.	.	.	.	.

- $i = 2, j = 6, k = 4$

$B =$

2	4	5	7	6	3	2	1

$A =$

1	2	2	.	.	.	.	.

Merge sort/7

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 2, j = 5, k = 5$

$B =$

2	4	5	7	6	3	2	1

Merge sort / 7

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 2, j = 5, k = 5$

$B =$

2	4	5	7	6	3	2	1

$A =$

1	2	2	3	.	.	.	.

Merge sort/7

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 2, j = 5, k = 5$

$B =$

2	4	5	7	6	3	2	1

$A =$

1	2	2	3	.	.	.	.

- $i = 3, j = 5, k = 6$

$B =$

2	4	5	7	6	3	2	1

MERGE SORT/7

1 2 4 7 | 21 18 17 9

if all elem from left part have been used then i points to lowest elem of right part and we take elem from right part

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 2, j = 5, k = 5$

$B =$	2	4	5	7	6	3	2	1
$A =$	1	2	2	3	.	.	.	.

i, j, k

- $i = 3, j = 5, k = 6$

$B =$	2	4	5	7	6	3	2	1
$A =$	1	2	2	3	4	.	.	.

i, j, k

Merge sort / 8

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 4, j = 5, k = 7$

$B =$

2	4	5	7	6	3	2	1

Merge sort / 8

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 4, j = 5, k = 7$

$B =$

2	4	5	7	6	3	2	1

$A =$

1	2	2	3	4	5	.	.

Merge sort / 8

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 4, j = 5, k = 7$

$B =$

2	4	5	7	6	3	2	1

$A =$

1	2	2	3	4	5	.	.

- $i = 4, j = 4, k = 8$

$B =$

2	4	5	7	6	3	2	1

MERGE SORT / 8

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$

- $i = 4, j = 5, k = 7$

$B =$

2	4	5	7	6	3	2	1

$A =$

1	2	2	3	4	5	.	.

- $i = 4, j = 4, k = 8$

$B =$

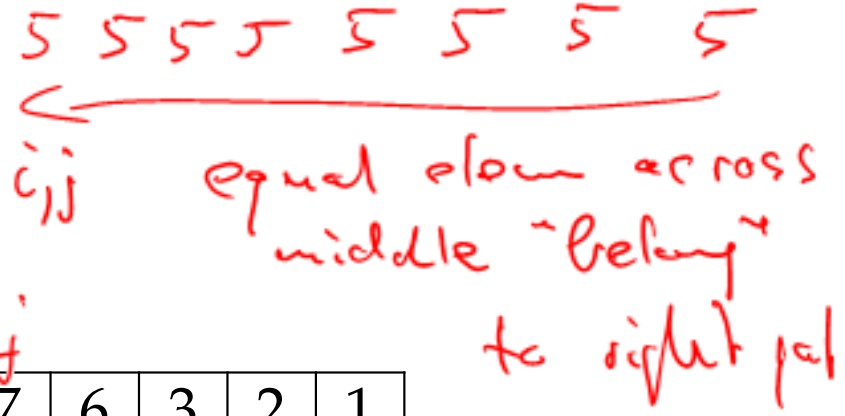
2	4	5	7	6	3	2	1

$A =$

1	2	2	3	4	5	6	.

MERGE SORT/8

Merge of $[2, 4, 5, 7, 1, 2, 3, 6]$



- $i = 4, j = 5, k = 7$

$B =$	2	4	5	7	6	3	2	1
$A =$	1	2	2	3	4	5	.	.

i j k

- $i = 4, j = 4, k = 8$

$B =$	2	4	5	7	6	3	2	1
$A =$	1	2	2	3	4	5	6	.

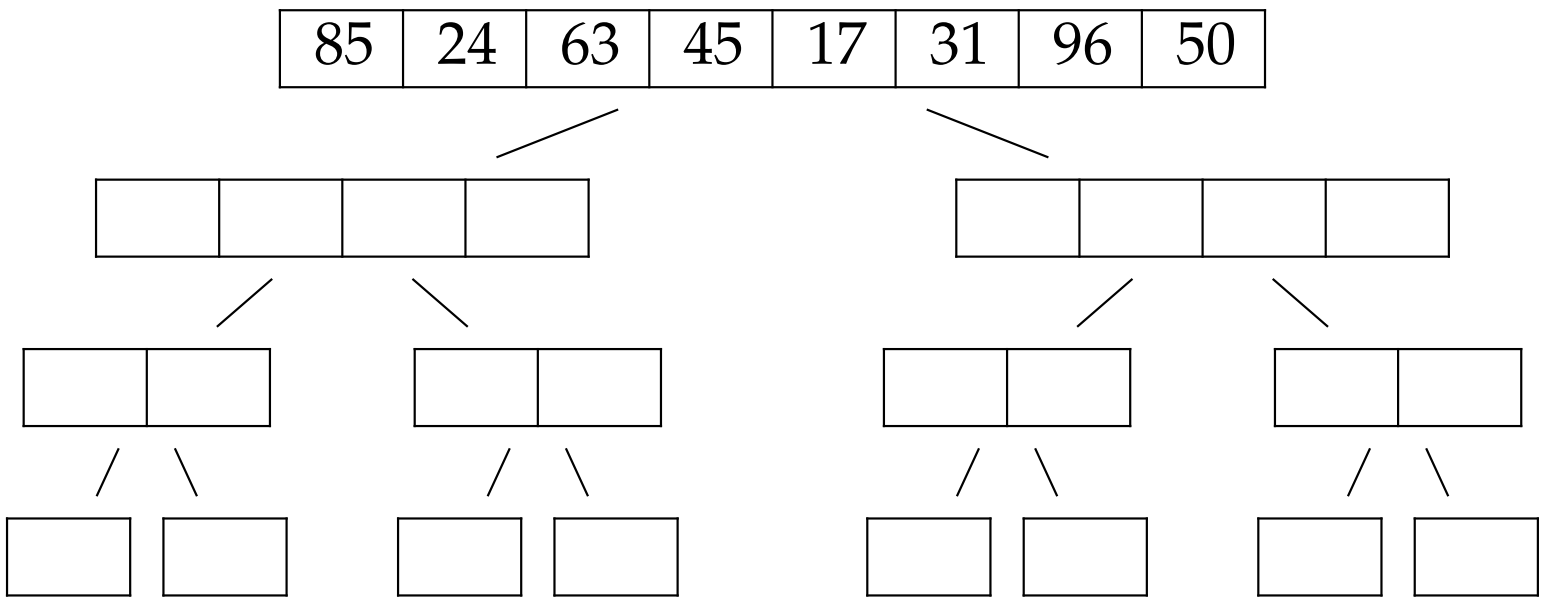
i,j k

- After last iteration of for loop:

$A =$	1	2	2	3	4	5	6	7

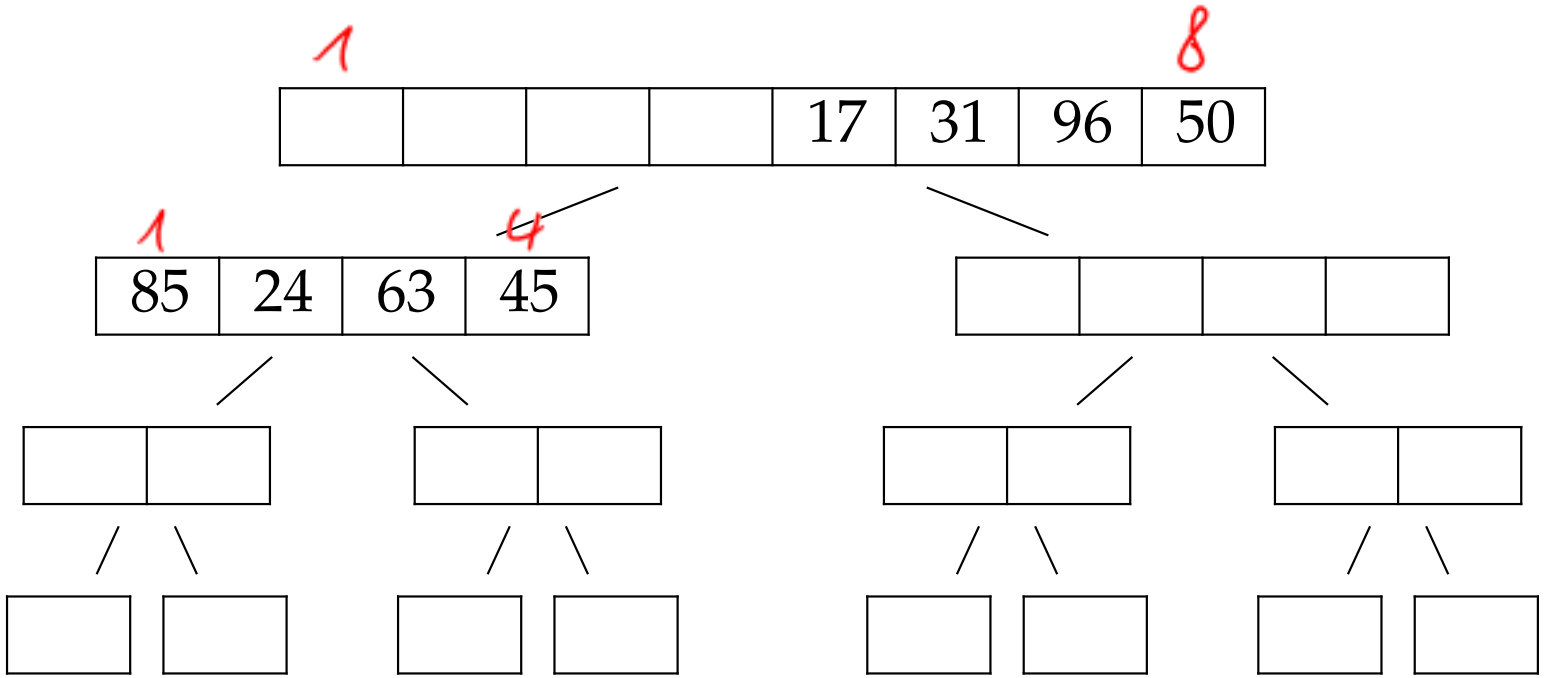
MERGE SORT/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



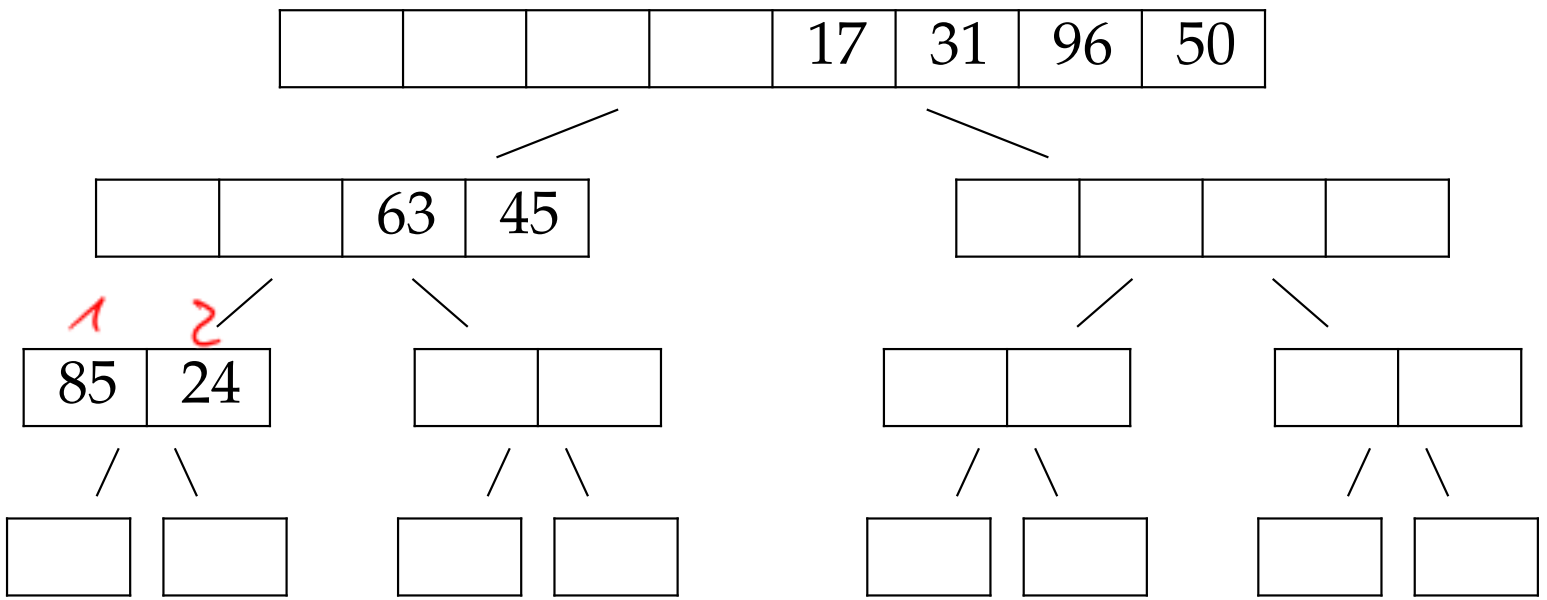
MERGE SORT / 9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



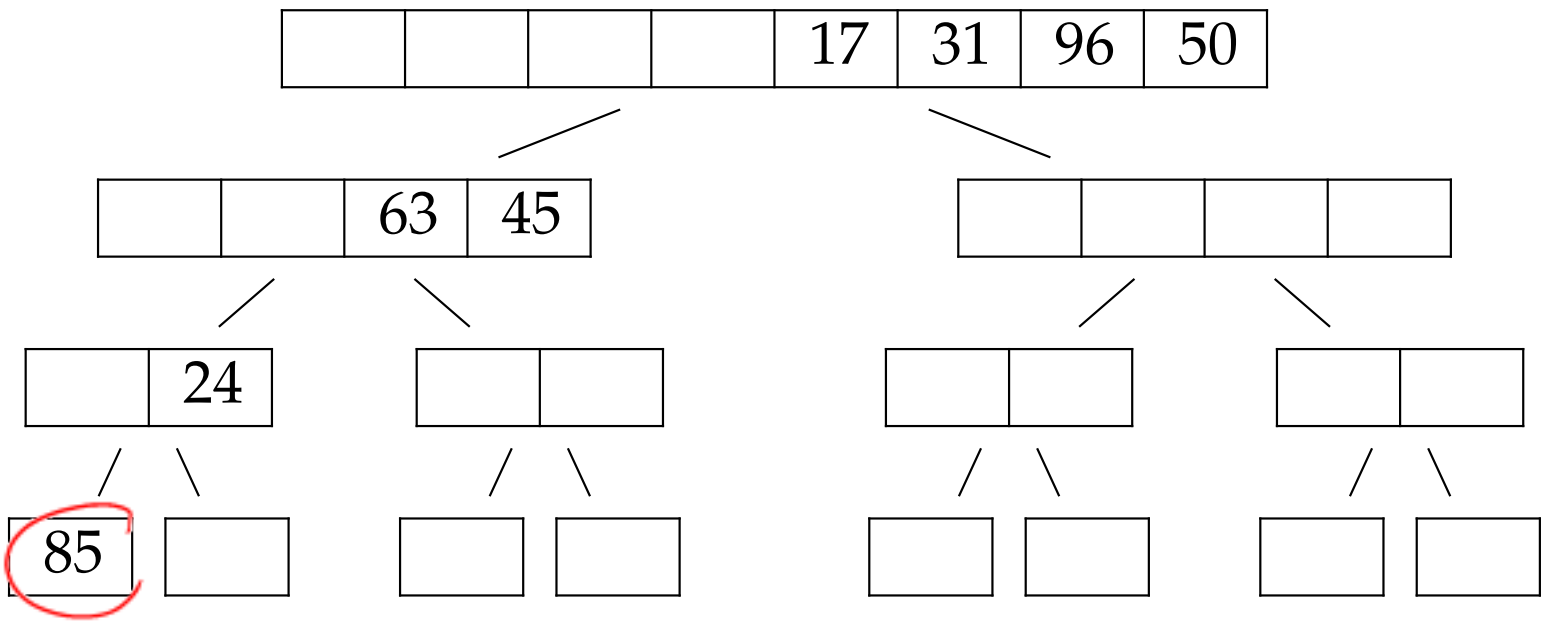
MERGE SORT/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



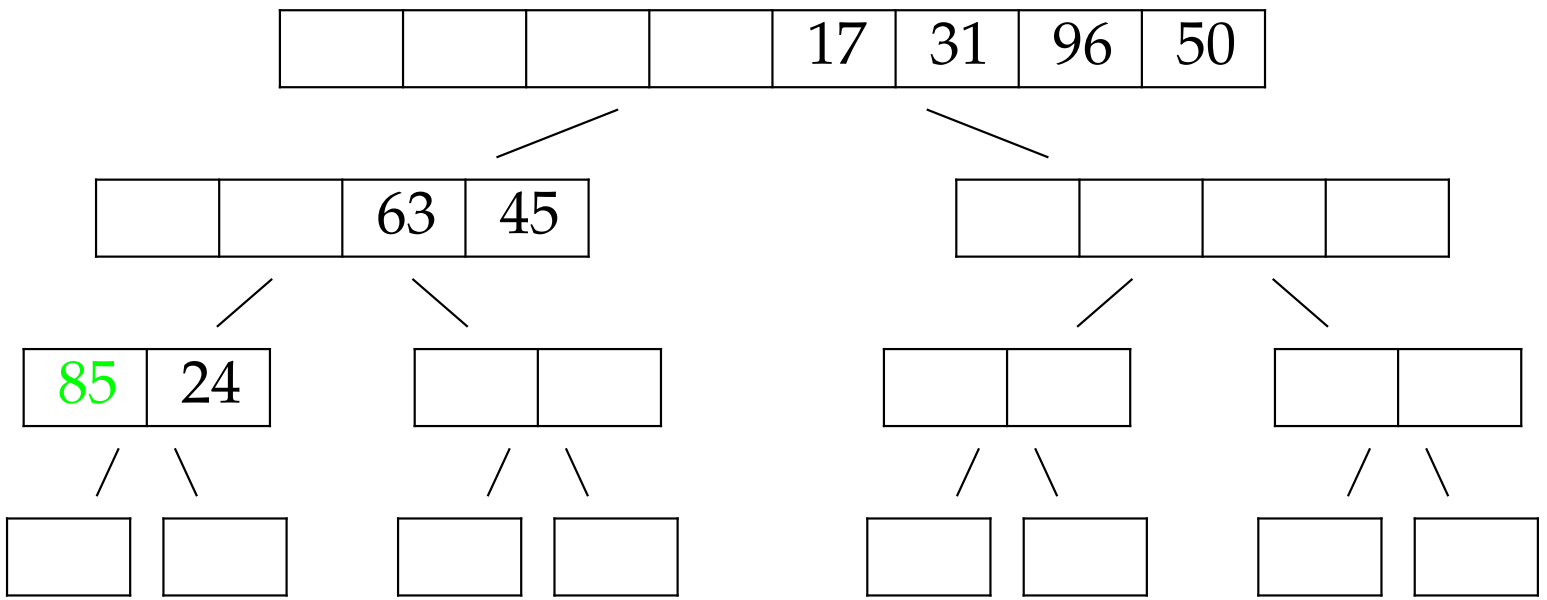
MERGE SORT / 9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



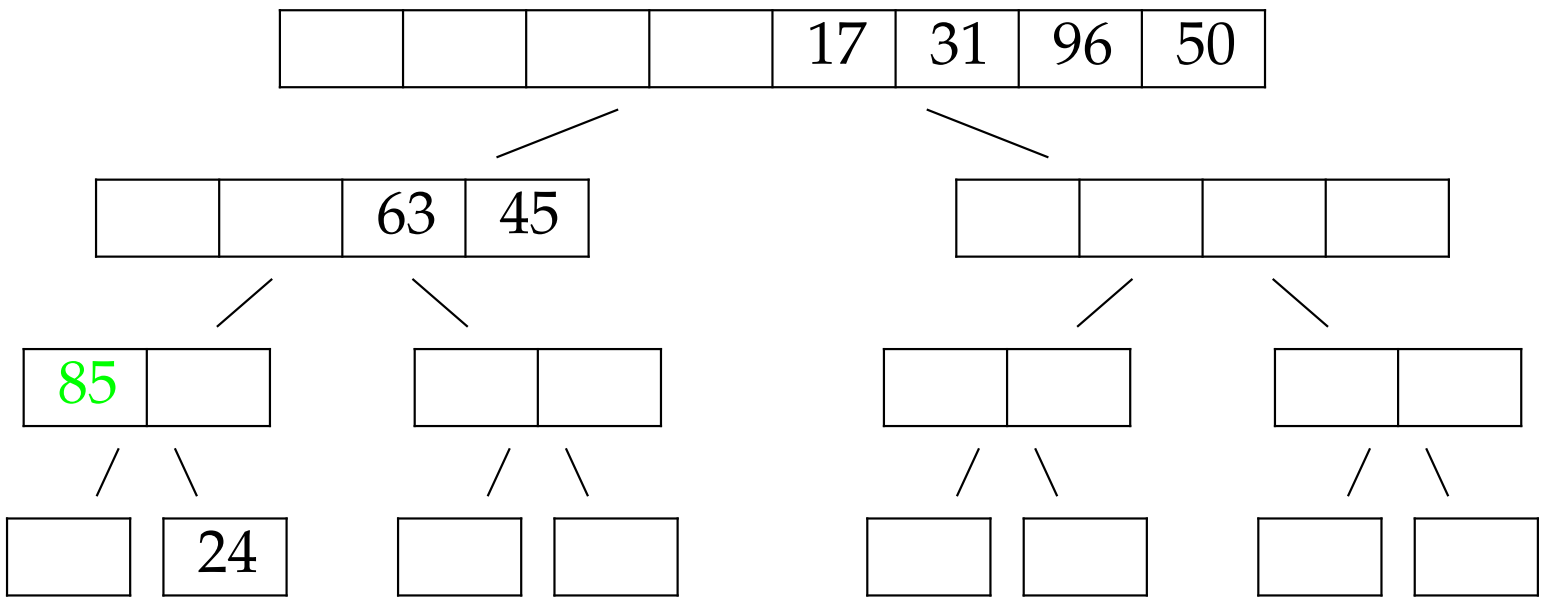
MERGE SORT/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



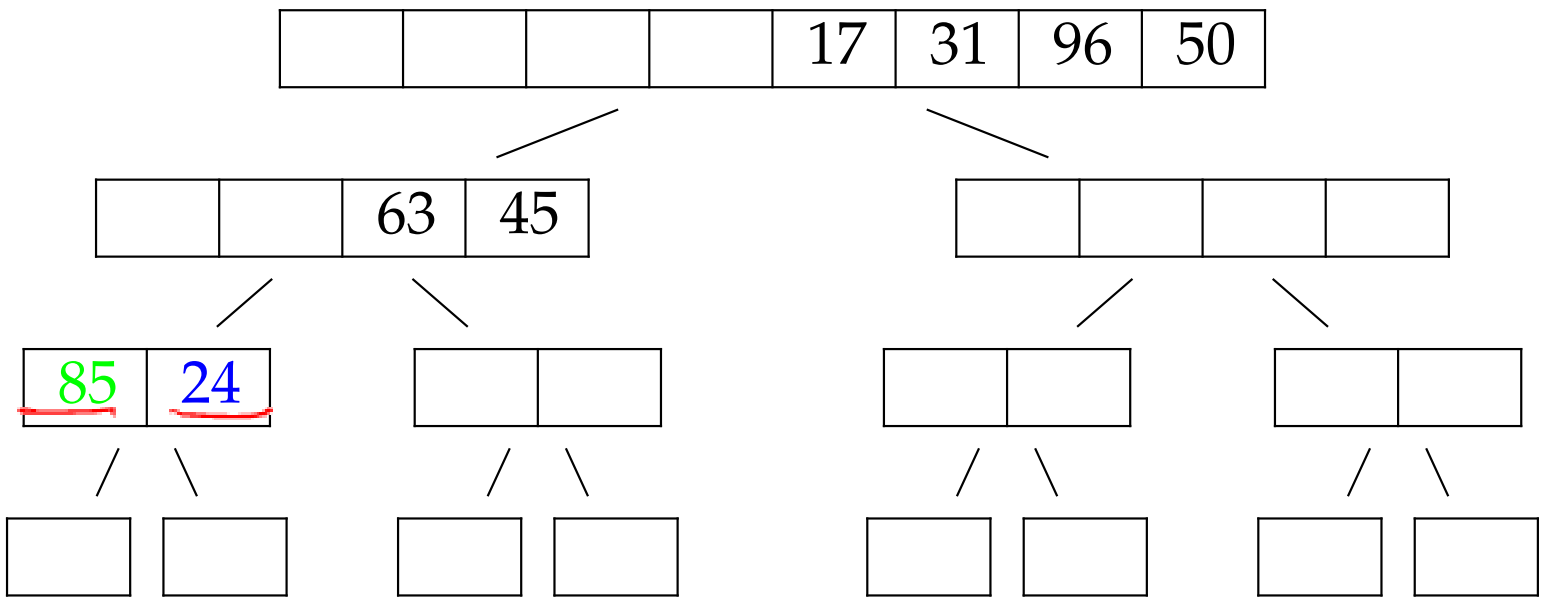
MERGE SORT/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



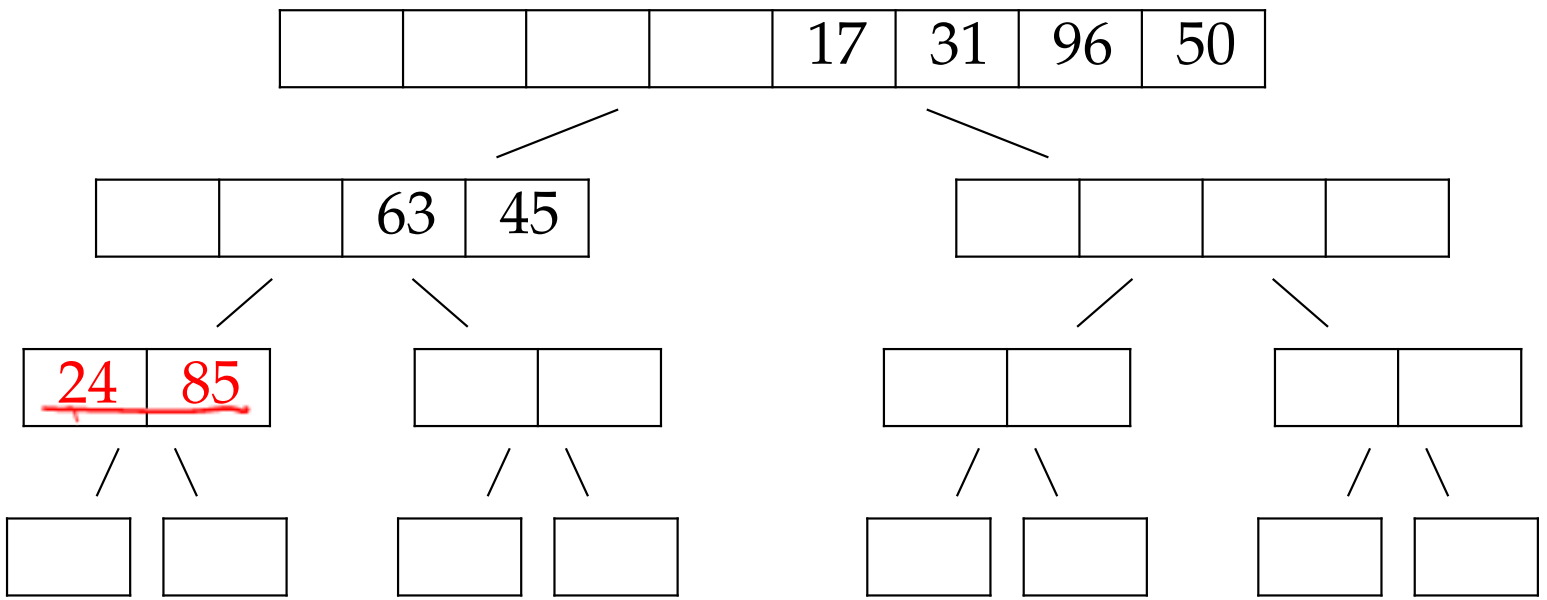
MERGE SORT/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



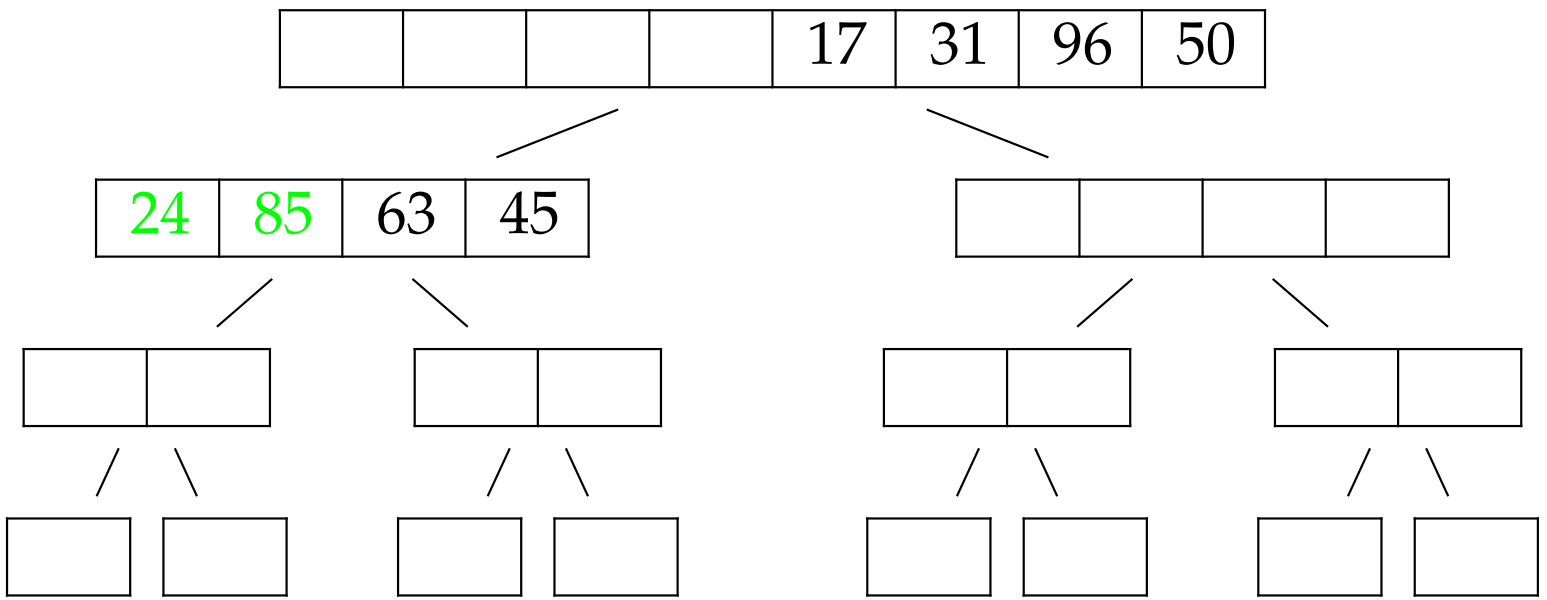
MERGE SORT/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



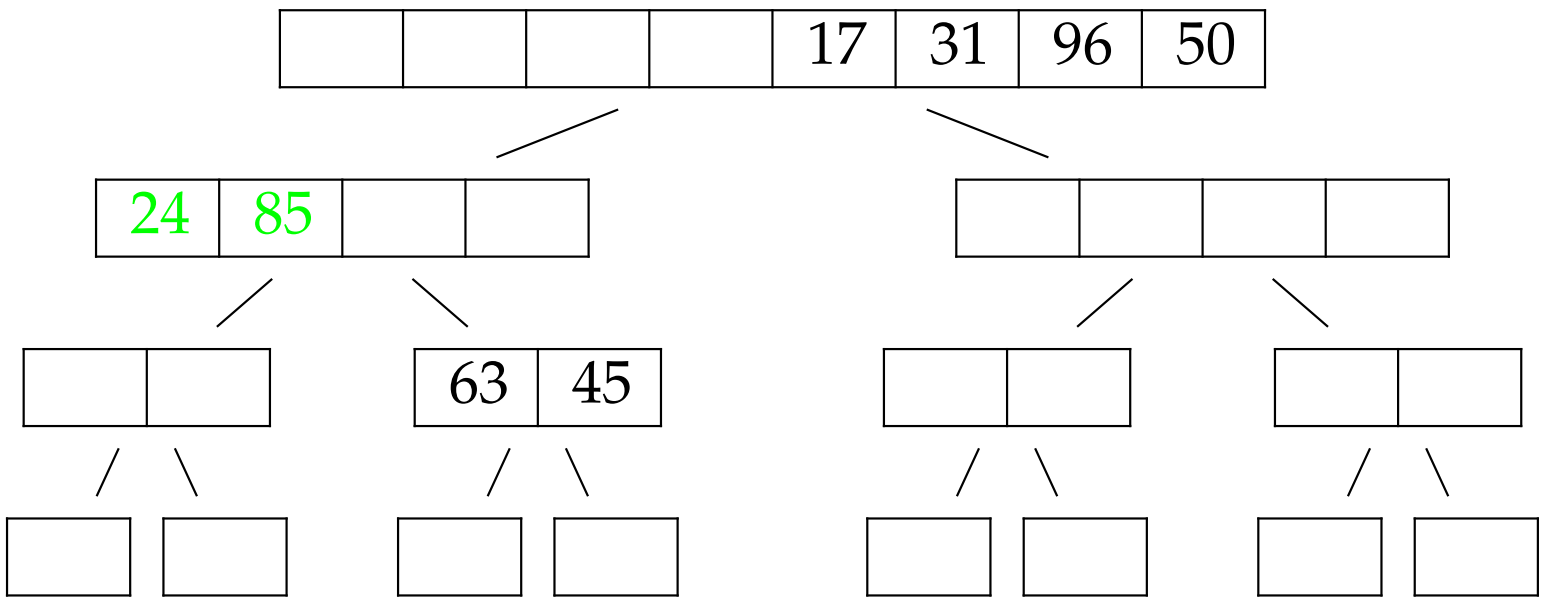
MERGE SORT/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



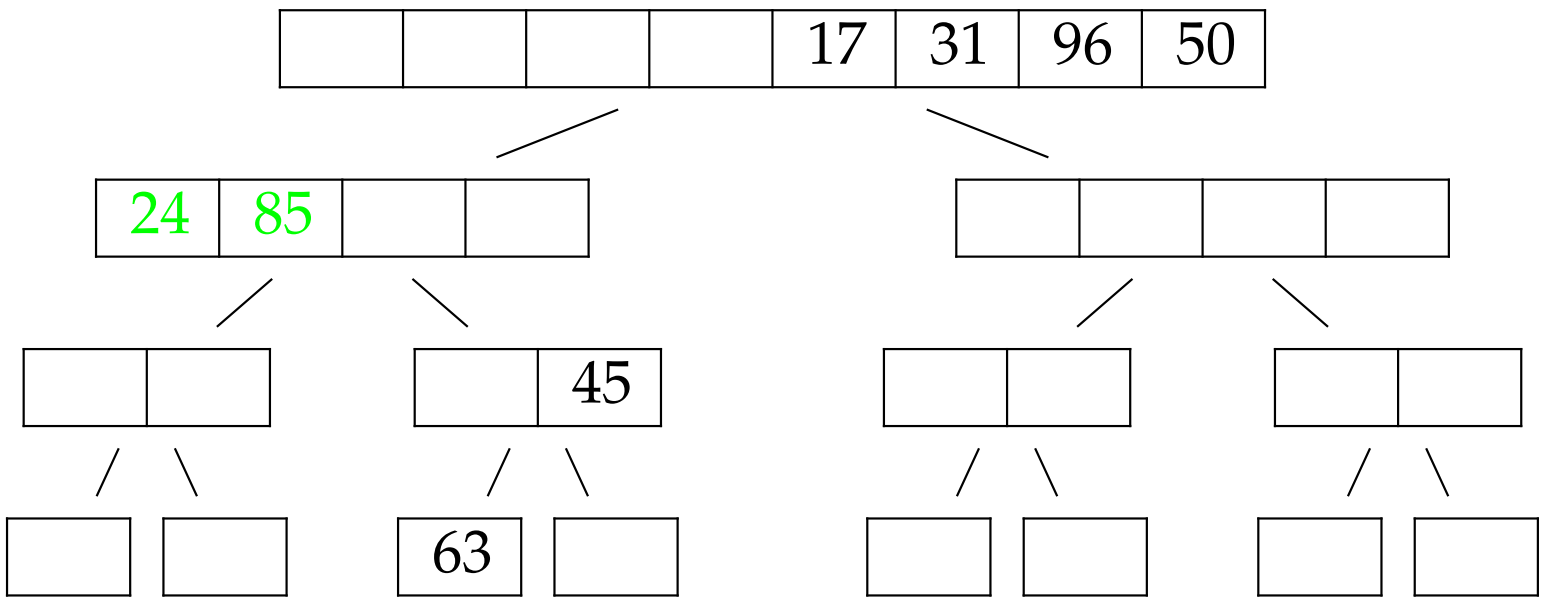
MERGE SORT/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



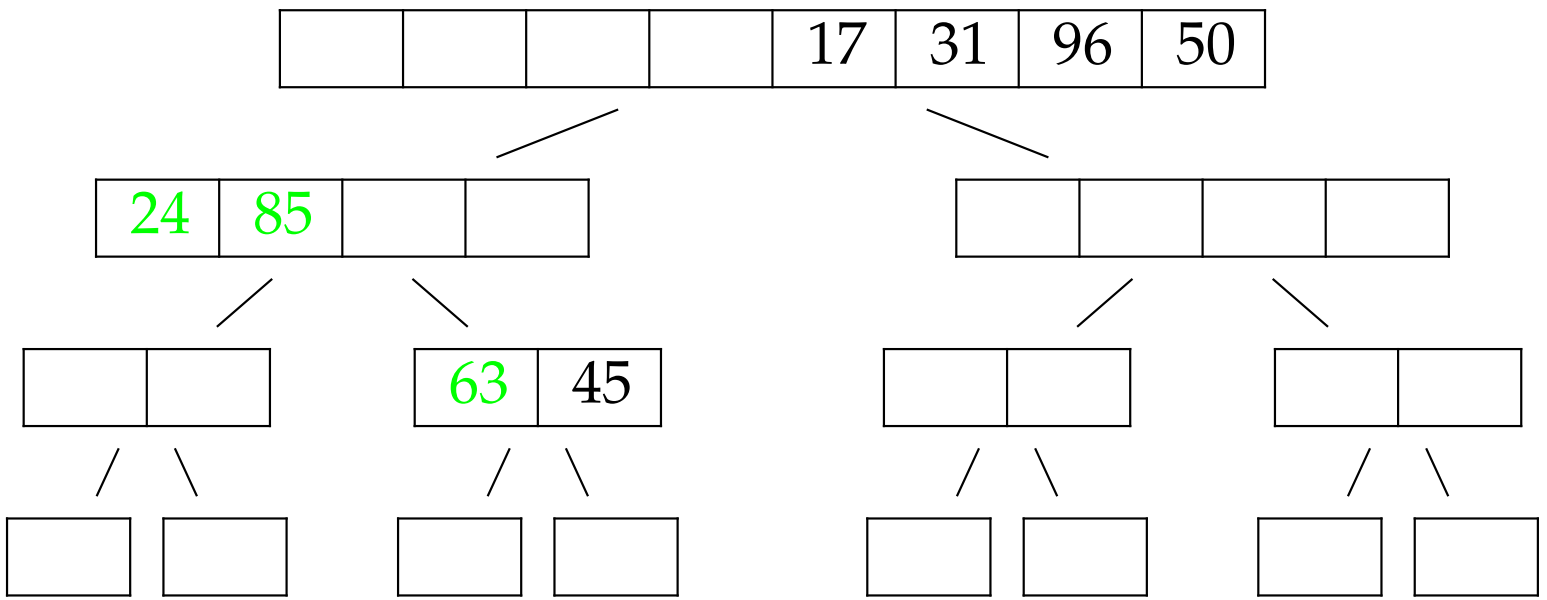
MERGE SORT/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



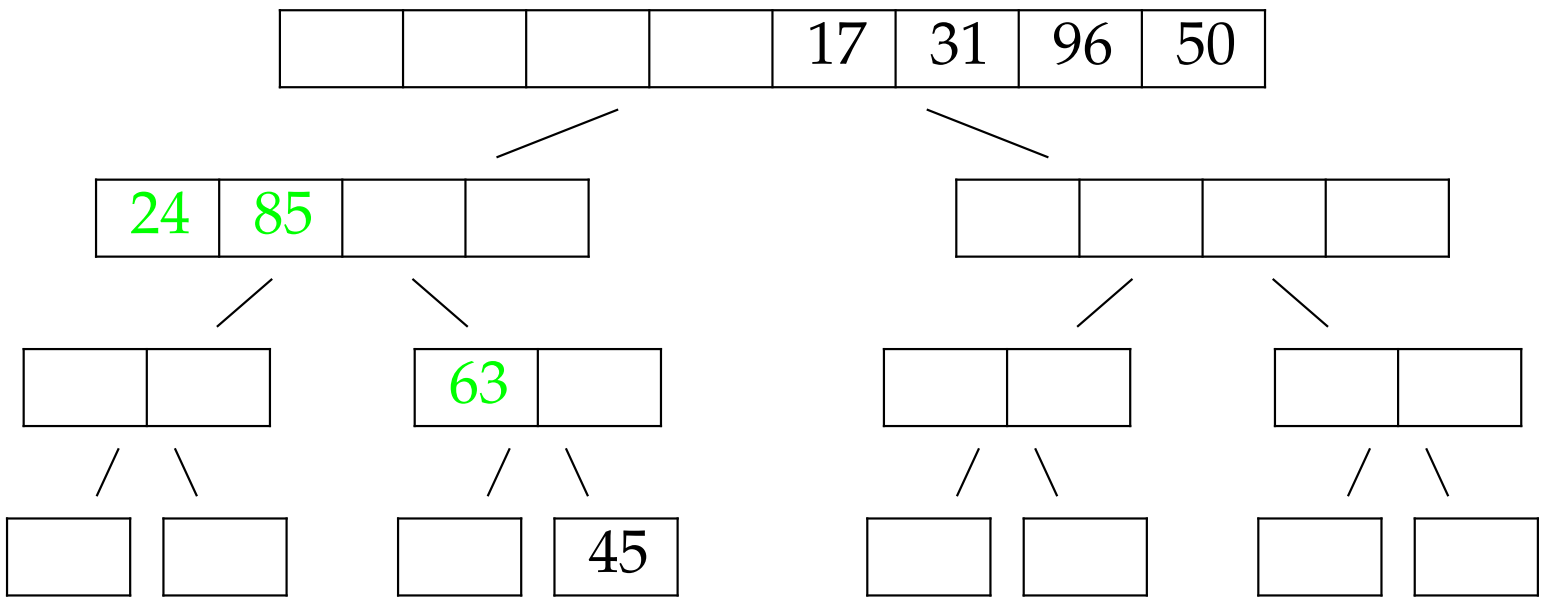
MERGE SORT/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



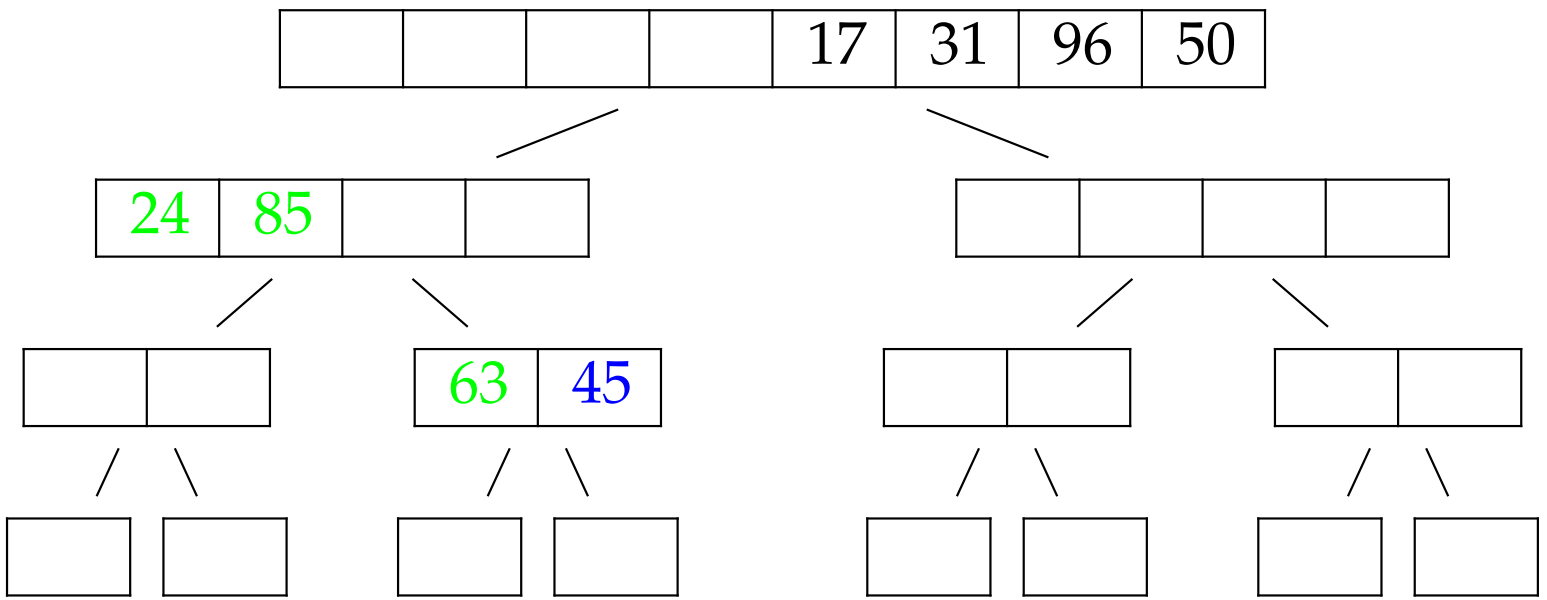
MERGE SORT/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



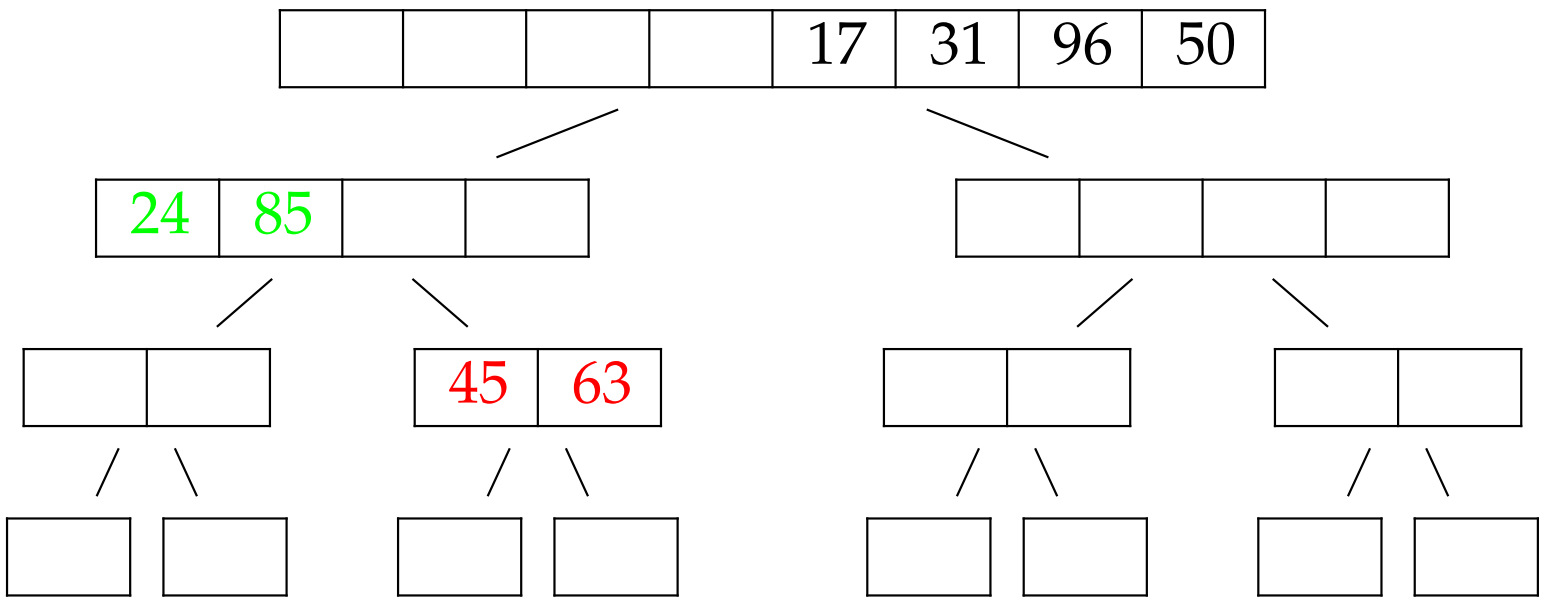
MERGE SORT/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



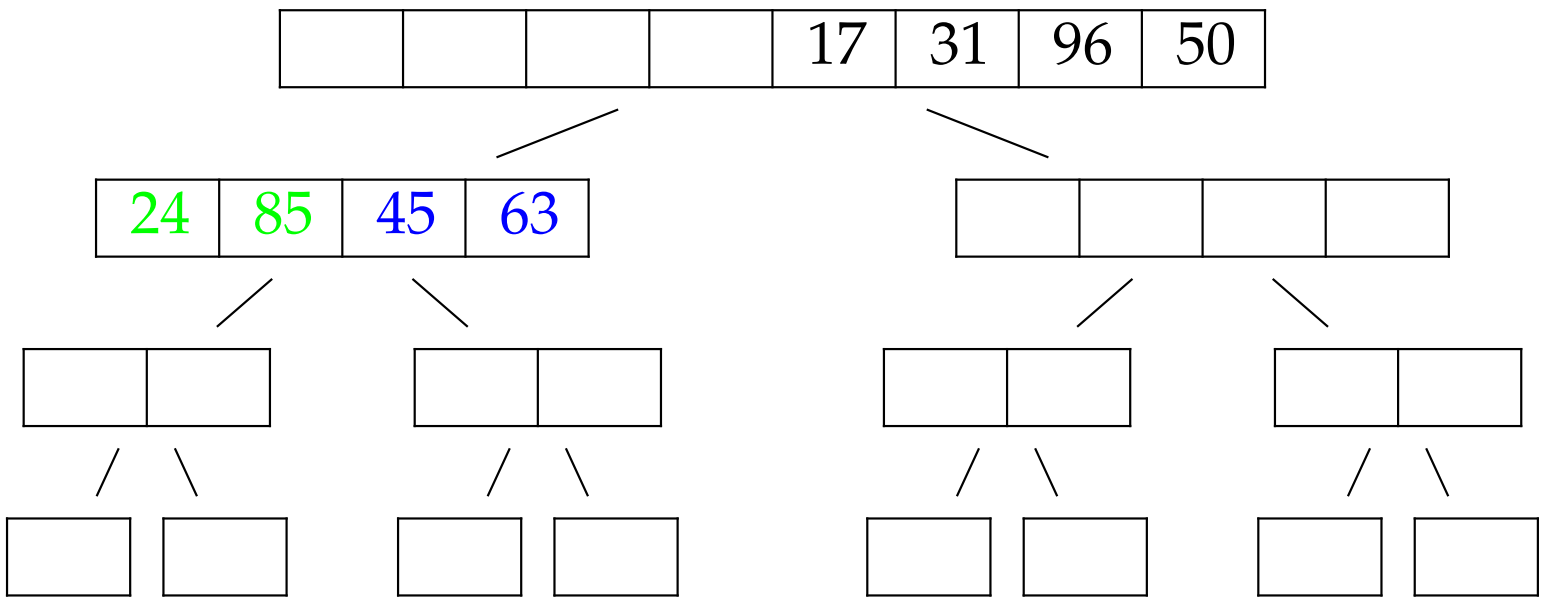
MERGE SORT/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



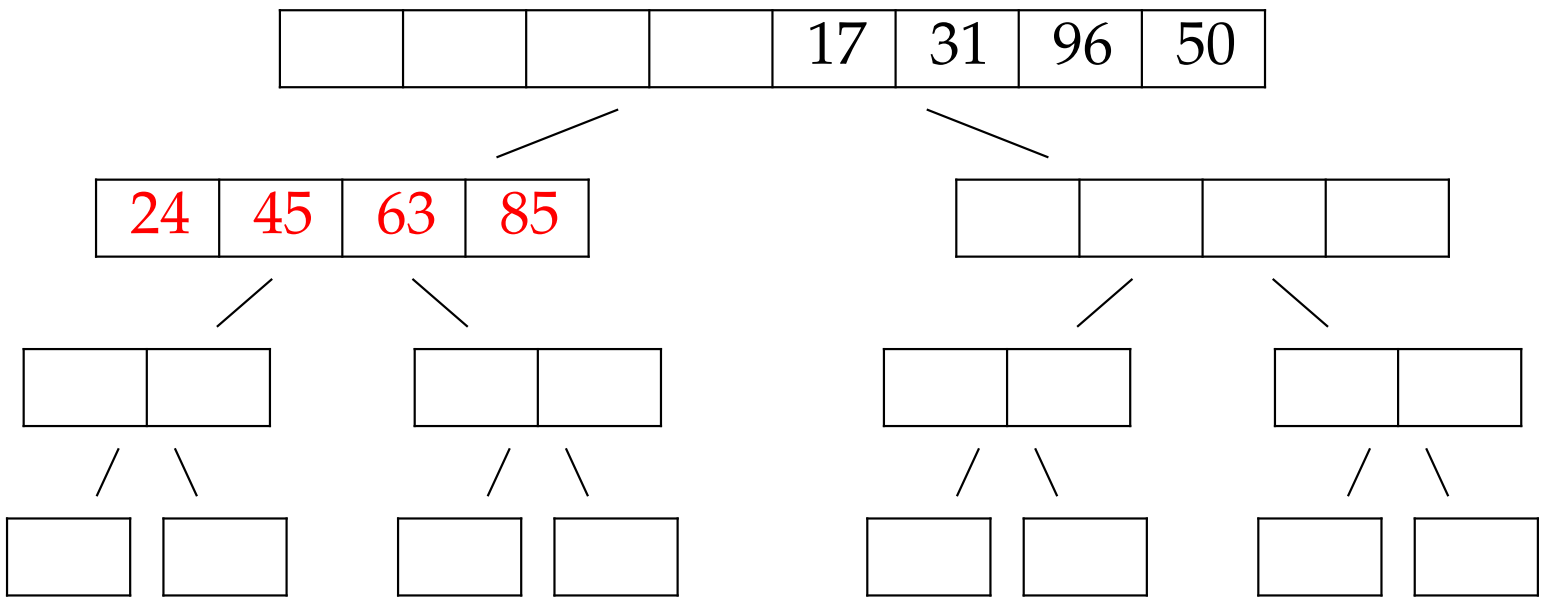
MERGE SORT/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



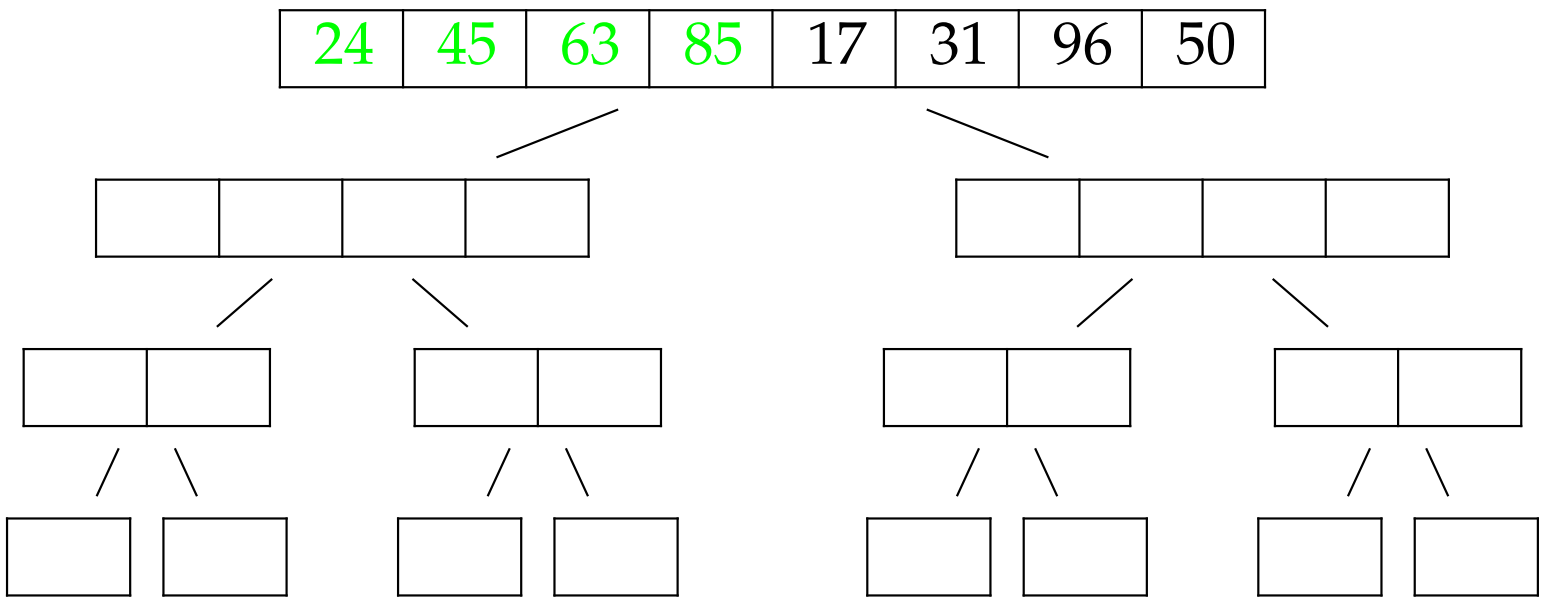
MERGE SORT/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



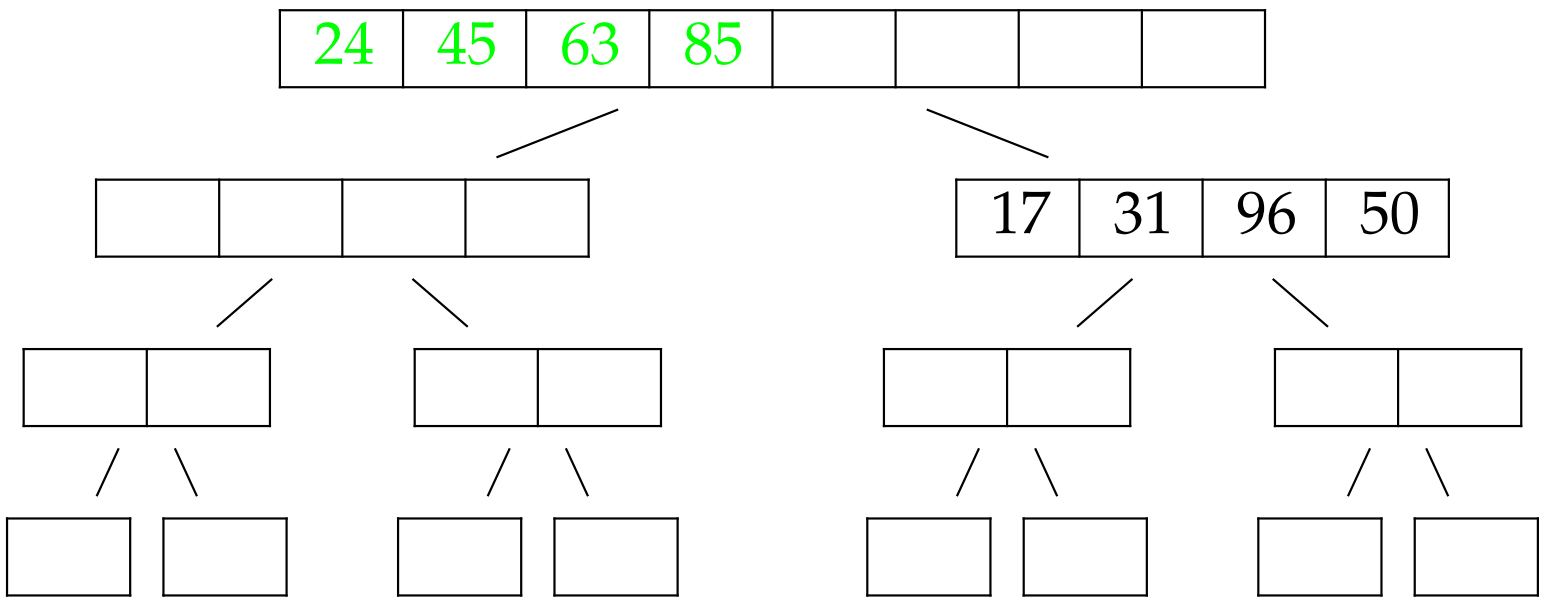
MERGE SORT/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



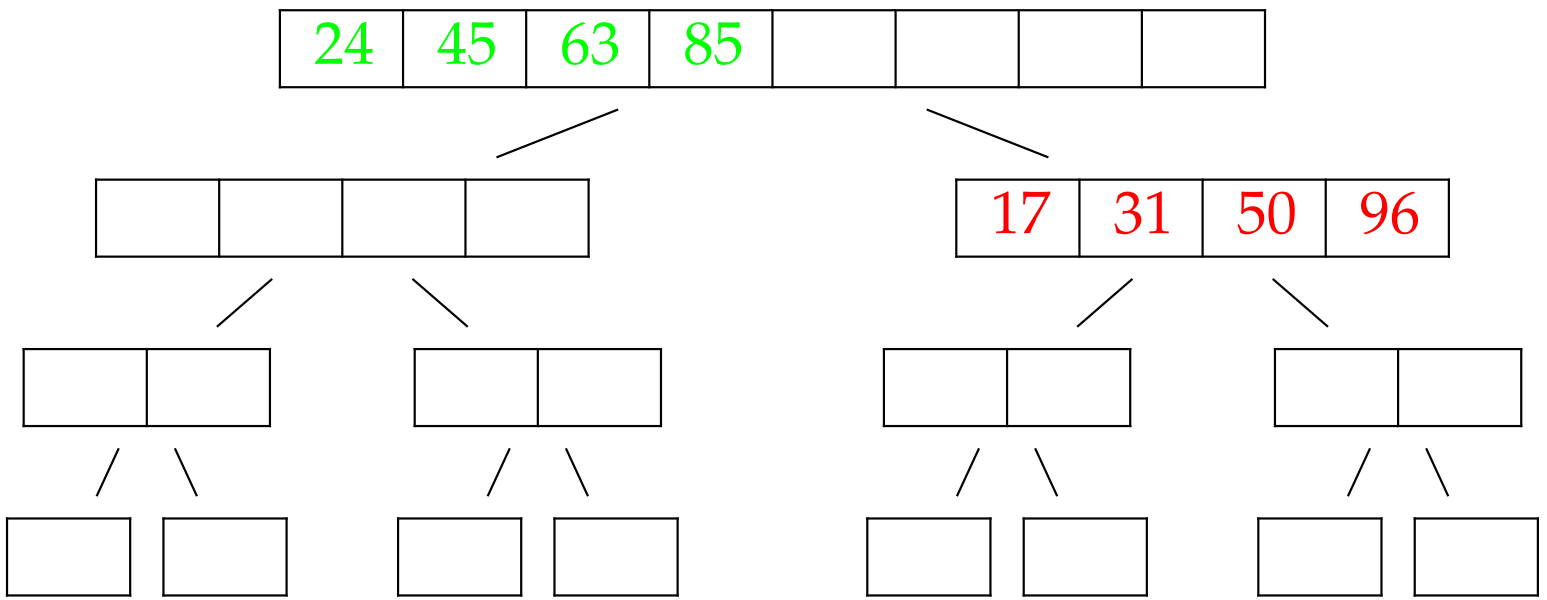
MERGE SORT/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



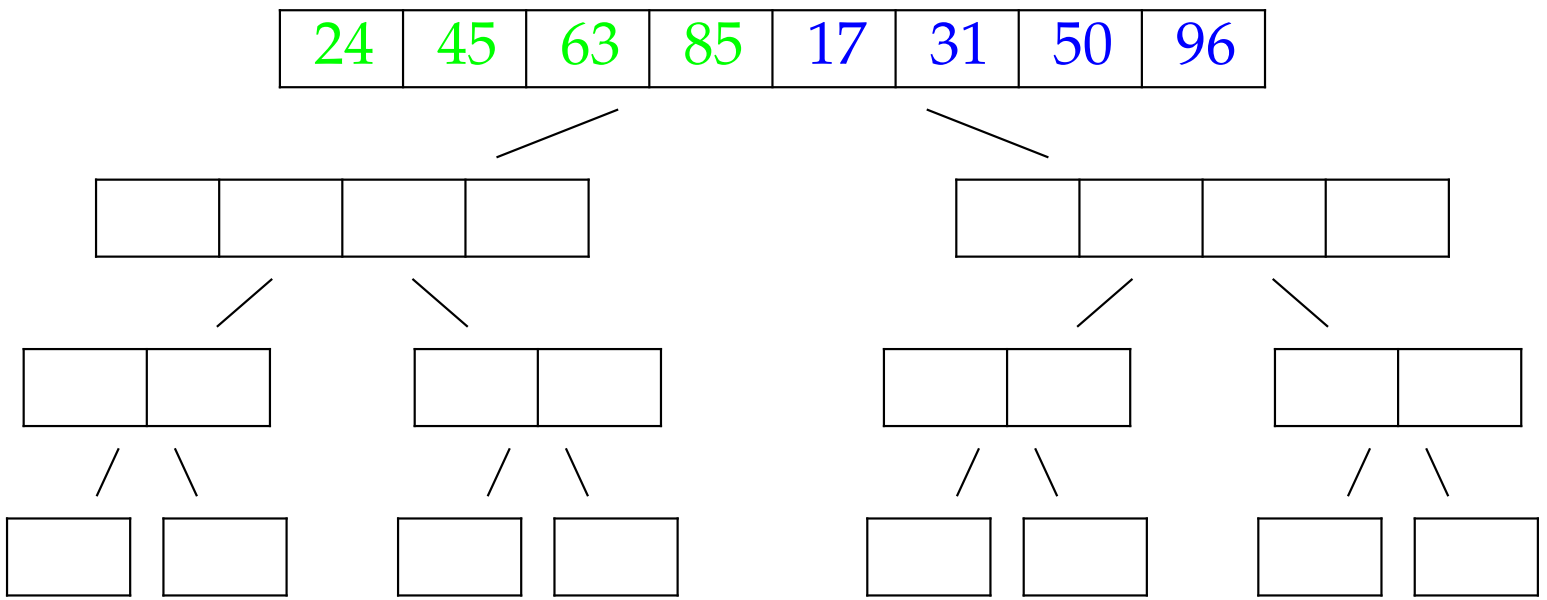
MERGE SORT/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



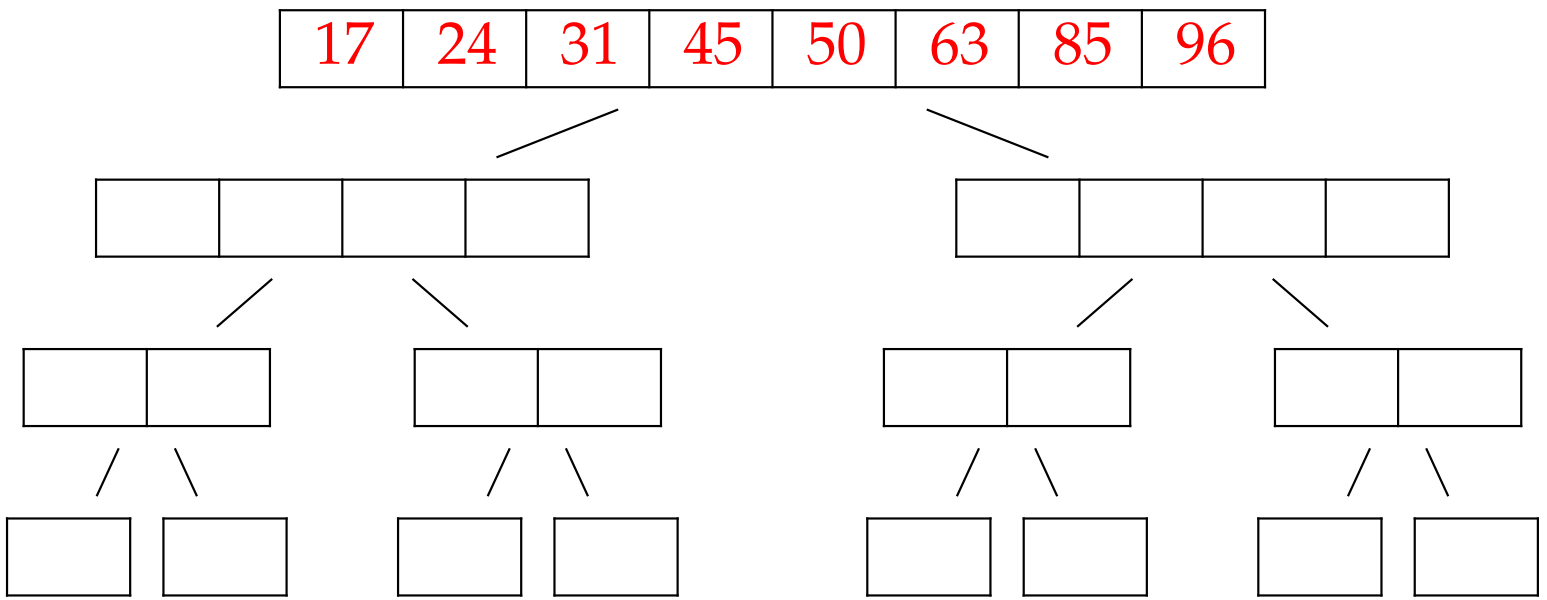
MERGE SORT/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



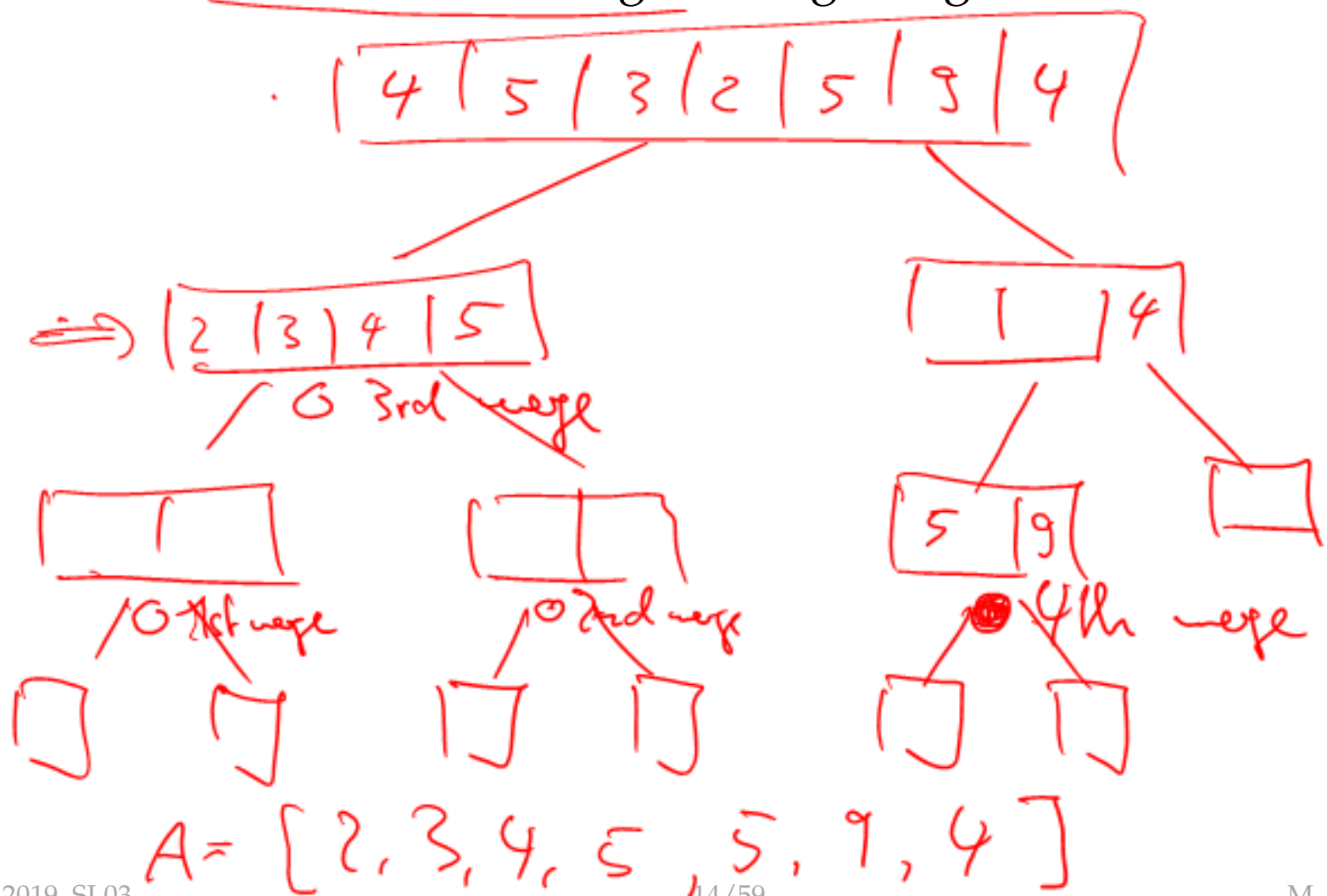
MERGE SORT/9

MergeSort of [85, 24, 63, 45, 17, 31, 31, 96, 50]



EXERCISE SL03-1

Assume $A = [4, 5, 3, 2, 5, 9, 4]$. Show the content of the array after the fourth call of Merge during MergeSort.



MERGE SORT / 10

Running time

- Recursive algorithms: $T(n)$ described by a recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

- $n \le c$: base case
- a subproblems of size $1/b$ each
- $D(n)$: to divide problem
- $C(n)$: to combine solutions

MERGE SORT / 10

Running time

- Recursive algorithms: $T(n)$ described by a recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

  - $n \leq c$: base case
  - a subproblems of size $1/b$ each
  - $D(n)$: to divide problem
  - $C(n)$: to combine solutions

- Merge sort: $a = b = 2$, $D(n) = \Theta(1)$, $C(n) = \Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

MERGE SORT/11

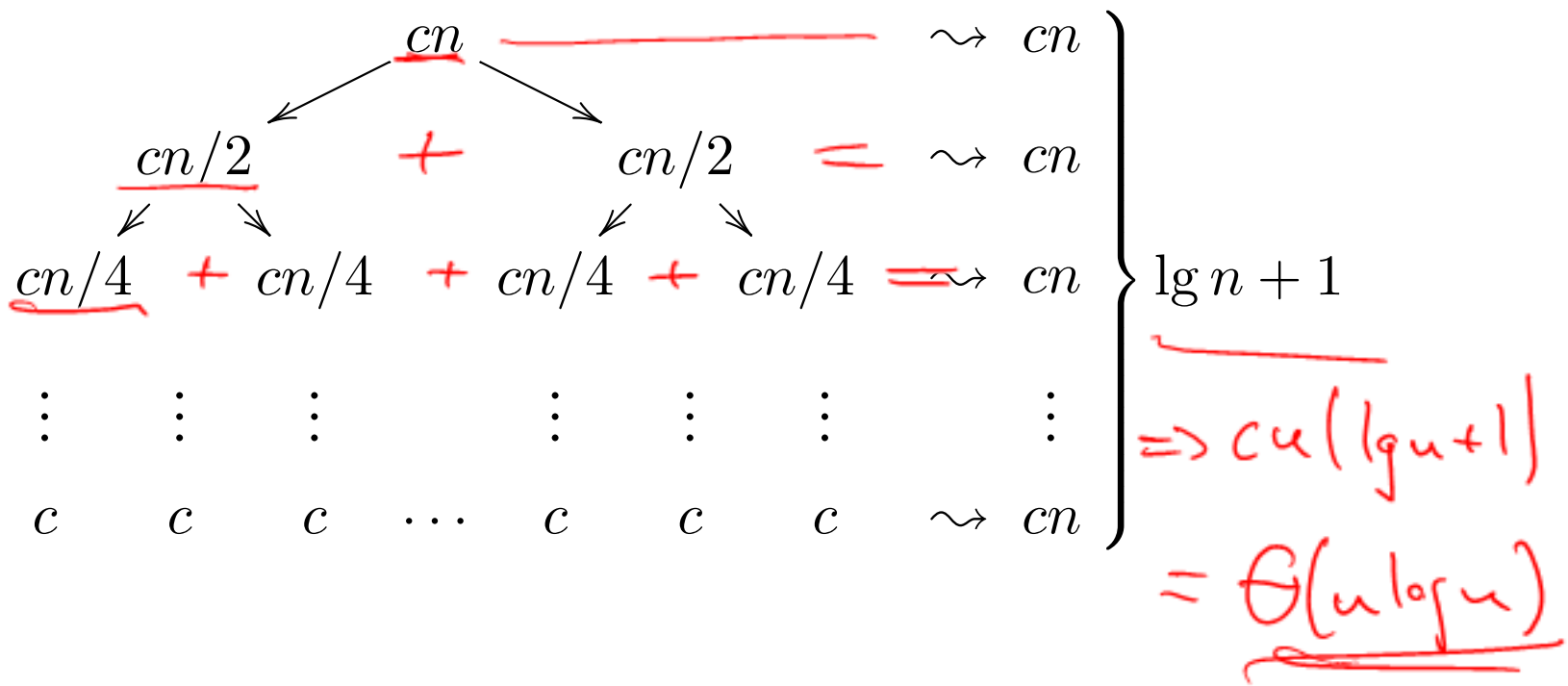
The recurrence/1

- Assume that $n > 1$ is an exact power of 2

MERGE SORT / 11

The recurrence / 1

- Assume that $n > 1$ is an exact power of 2
- $T(n) = 2T(n/2) + cn$ yields $T(n) = \Theta(n \lg n)$



MERGE SORT/12

The recurrence/2

- Alternatively, one may proceed as follows
  - For $n > 1$, let $T(n) = 2T(n/2) + cn$
  - Apply repeated backward substitution

$$T(n) = 2T(n/2) + cn \qquad \text{(substitute)}$$

MERGE SORT/12

The recurrence/2

- Alternatively, one may proceed as follows
  - For $n > 1$, let $T(n) = 2T(n/2) + cn$
  - Apply repeated backward substitution

$$\begin{aligned} T(n) &= 2T(n/2) + cn && \text{(substitute)} \\ &= 2(2T(n/4) + cn/2) + cn && \text{(expand)} \end{aligned}$$

MERGE SORT/12

The recurrence/2

- Alternatively, one may proceed as follows
  - For $n > 1$, let $T(n) = 2T(n/2) + cn$
  - Apply repeated backward substitution

$$\begin{aligned} T(n) &= 2T(n/2) + cn && \text{(substitute)} \\ &= 2(2T(n/4) + cn/2) + cn && \text{(expand)} \\ &= 4T(n/4) + 2cn && \text{(substitute)} \end{aligned}$$

MERGE SORT/12

The recurrence/2

- Alternatively, one may proceed as follows
  - For $n > 1$, let $T(n) = 2T(n/2) + cn$
  - Apply repeated backward substitution

$$\begin{aligned} T(n) &= 2T(n/2) + cn && \text{(substitute)} \\ &= 2(2T(n/4) + cn/2) + cn && \text{(expand)} \\ &= 4T(n/4) + 2cn && \text{(substitute)} \\ &= 4(2T(n/8) + cn/4) + 2cn && \text{(expand)} \end{aligned}$$

MERGE SORT/12

The recurrence/2

- Alternatively, one may proceed as follows
  - For $n > 1$, let $T(n) = 2T(n/2) + cn$
  - Apply repeated backward substitution

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn && \text{(substitute)} \\
&= 2(2T(n/4) + cn/2) + cn && \text{(expand)} \\
&= 4T(n/4) + 2cn && \text{(substitute)} \\
&= 4(2T(n/8) + cn/4) + 2cn && \text{(expand)} \\
&= 8T(n/8) + 3cn && \text{(find pattern)}
\end{aligned}
$$

MERGE SORT/12

The recurrence/2

- Alternatively, one may proceed as follows
  - For $n > 1$, let $T(n) = 2T(n/2) + cn$
  - Apply repeated backward substitution

$$\begin{aligned} T(n) &= 2T(n/2) + cn && \text{(substitute)} \\ &= 2(2T(n/4) + cn/2) + cn && \text{(expand)} \\ &= 4T(n/4) + 2cn && \text{(substitute)} \\ &= 4(2T(n/8) + cn/4) + 2cn && \text{(expand)} \\ &= 8T(n/8) + 3cn && \text{(find pattern)} \end{aligned}$$

- Note that $T(n) = 2^i T(n/2^i) + icn$ where $i_{max} = \lg n$

$$2^{\lg n} T(n/2^{\lg n}) + cn \lg n = nT(1) + cn \lg n = \Theta(n \lg n)$$

EXERCISE SL03-2

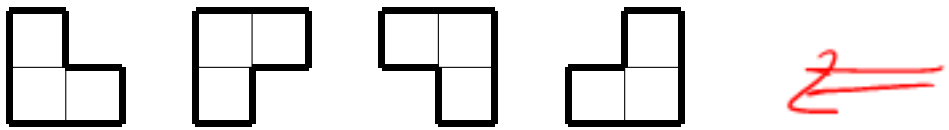
$2^0 + 2^1 + \dots + 2^{i-1} = 2^i - 1$

Solve the recurrence $T(n) = 2T(n/2) + \Theta(1)$.

$$T(n) = 2T(n/2) + \Theta(1)$$
$$= 2(2T(n/4) + \Theta(1)) + \Theta(1)$$
$$= 2^2 T(n/4) + (2+1)\Theta(1)$$
$$= 2^3 T(n/8) + (2^2 + 2^1 + 2^0)\Theta(1)$$
$$= 2^i T(n/2^i) + (2^i - 1)\Theta(1) \qquad i_{max} = \log n$$
$$= 2^{\log n} T(n/n) + 2^{\log n}\Theta(1) - \Theta(1)$$
$$= n + n\Theta(1) - \Theta(1) = \Theta(n)$$

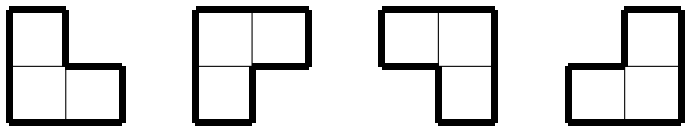
TROMINO TILING/1

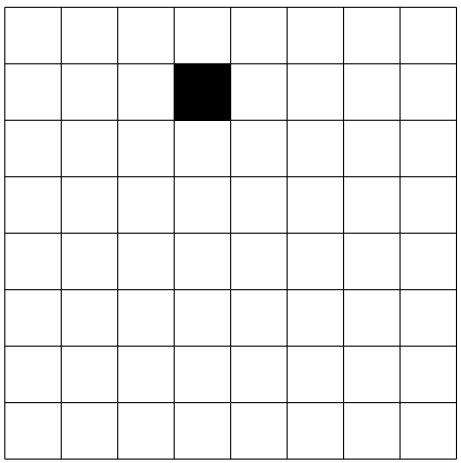
Problem description

- Tromino tiles 

TROMINO TILING/1

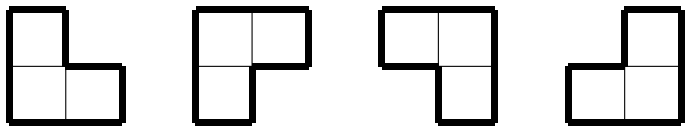
Problem description

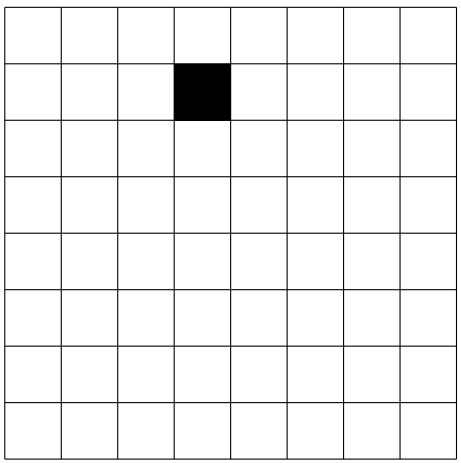
- Tromino tiles 
- Provide a tromino tiling for a $2^n \times 2^n$ board with a hole

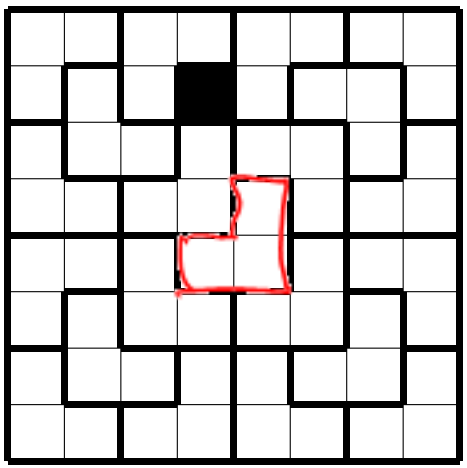


TROMINO TILING/1

Problem description

- Tromino tiles 
- Provide a tromino tiling for a $2^n \times 2^n$ board with a hole





TROMINO TILING/2

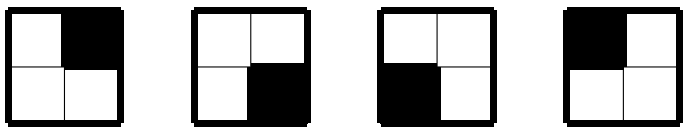
Trivial case: $n = 1$

- Tromino tiling for a $2^1 \times 2^1$ board with a hole

TROMINO TILING/2

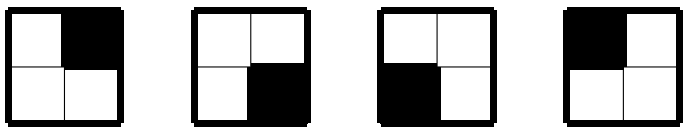
Trivial case: $n = 1$

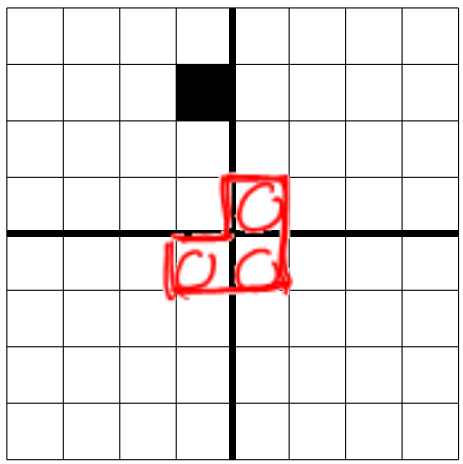
- Tromino tiling for a $2^1 \times 2^1$ board with a hole
- Solution is easy



TROMINO TILING/2

Trivial case: $n = 1$

- Tromino tiling for a $2^1 \times 2^1$ board with a hole
- Solution is easy 
- Idea for $2^n \times 2^n$ board: reduce size of the board



TROMINO TILING/3

General case: $n > 1$

- Problem: get only one problem of size $2^{n-1} \times 2^{n-1}$

TROMINO TILING/3

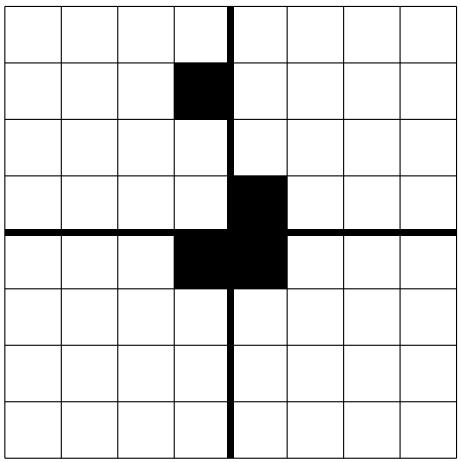
General case: $n > 1$

- Problem: get only one problem of size $2^{n-1} \times 2^{n-1}$
- But what about the other squares? There are no holes

TROMINO TILING/3

General case: $n > 1$

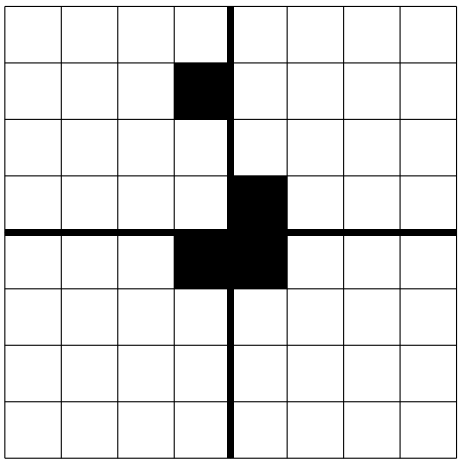
- Problem: get only one problem of size $2^{n-1} \times 2^{n-1}$
- But what about the other squares? There are no holes
- Solution: use a tromino to simulate missing holes



TROMINO TILING/3

General case: $n > 1$

- Problem: get only one problem of size $2^{n-1} \times 2^{n-1}$
- But what about the other squares? There are no holes
- Solution: use a tromino to simulate missing holes



- One can repeat this until the $2^1 \times 2^1$ case is reached

TROMINO TILING / 4

Sketch of algorithm

- Input: board size n ($2^n \times 2^n$), location l of the hole
- Output: tiling of $2^n \times 2^n$ board with hole in location l

TROMINO TILING / 4

Sketch of algorithm

- Input: board size n ($2^n \times 2^n$), location l of the hole
- Output: tiling of $2^n \times 2^n$ board with hole in location l
- Rough outline of `tile` (n, l)
  - If $n = 1$ then tile with one tromino
  - Otherwise, proceed as follows
    - Divide the board into four equal-sized boards
    - Put a tromino to simulate the three missing holes
    - Call `tile`$(n-1, l_1)$, `tile`$(n-1, l_2)$, `tile`$(n-1, l_3)$, and `tile`$(n-1, l_4)$ where l_1, l_2, l_3, and l_4 are the locations of the four holes

TROMINO TILING/5

Algo: Tile(n,L)

Input: $2^n x 2^n$ board, location L of hole
Output: tiling of the board

if $n==1$ **then** tile with one tromino and **return**;
Divide board into four equal-sized boards;
Place one tromino at center to cover 3 holes;
Let L1, L2, L3, L4 be positions of the 4 holes;
Tile(n-1,L1);
Tile(n-1,L2);
Tile(n-1,L3);
Tile(n-1,L4);

TROMINO TILING/6

Divide-and-conquer

- The algorithm is an instance of divide-and-conquer

TROMINO TILING/6

Divide-and-conquer

- The algorithm is an instance of divide-and-conquer
- The problem is trivial if the board is of size 2×2

TROMINO TILING/6

Divide-and-conquer

- The algorithm is an instance of divide-and-conquer
- The problem is trivial if the board is of size 2×2
- Divide: divide the board into four smaller boards
  - Uses extra tile to get proper problem instances

TROMINO TILING/6

Divide-and-conquer

- The algorithm is an instance of divide-and-conquer
- The problem is trivial if the board is of size 2×2
- Divide: divide the board into four smaller boards
  - Uses extra tile to get proper problem instances
- Conquer: tile the four smaller boards recursively

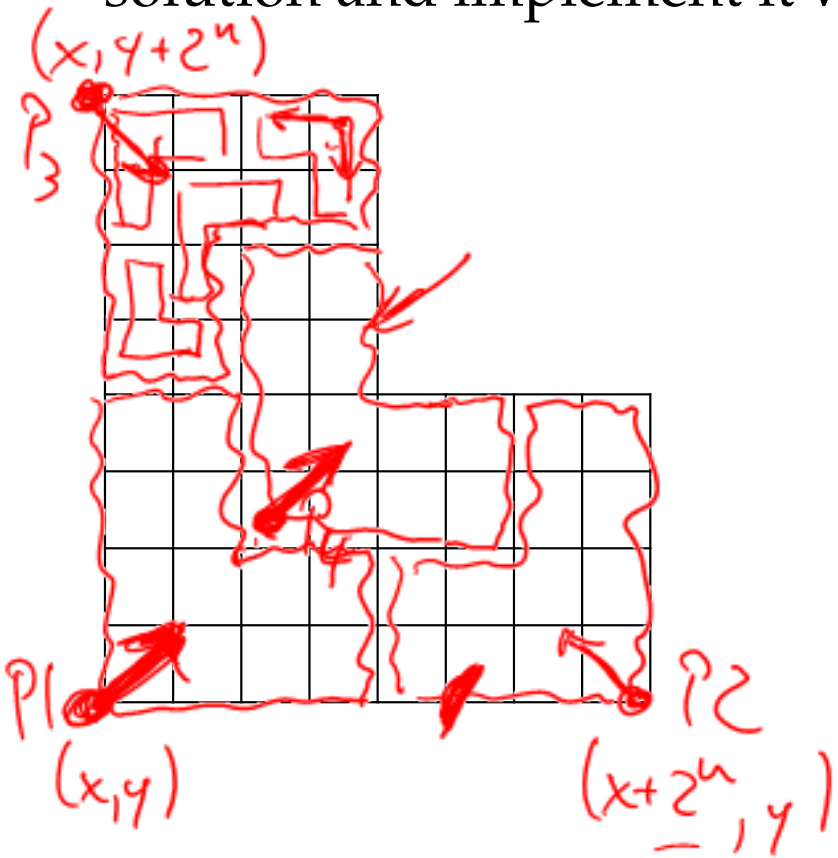
TROMINO TILING/6

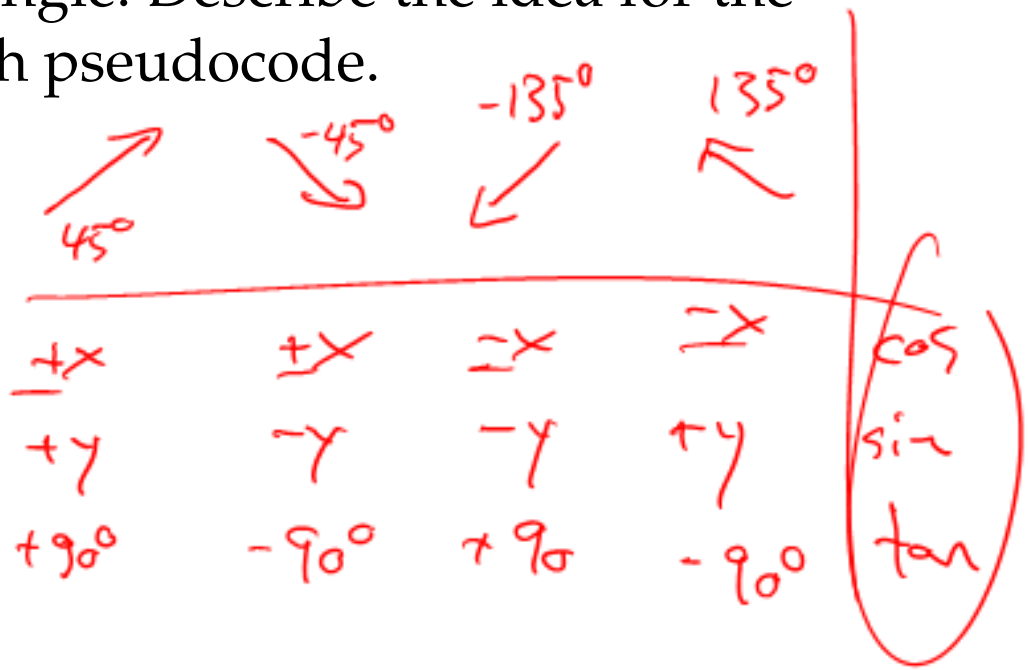
Divide-and-conquer

- The algorithm is an instance of divide-and-conquer
- The problem is trivial if the board is of size 2×2
- Divide: divide the board into four smaller boards
  - Uses extra tile to get proper problem instances
- Conquer: tile the four smaller boards recursively
- Combine: put a tromino around the center position
  - Covers missing holes to solve the original problem

EXERCISE SL03-3/1

Use a trominos to tile a right angle. Describe the idea for the solution and implement it with pseudocode.









EXERCISE SL03-3/2

TileL(n, P) → x, y, a

if (n==1) then tile with one tromino and return

Divide L-shape into 4 smaller L-shapes

TileL(n-1, P1) → x, y, a

TileL(n-1, P2) → x + sgn(cos(a)) · 2^n, y, a + sgn(tan(a)) · 90°

TileL(n-1, P3) → x, y + sgn(sin(a)) · 2^n, a − sgn(tan(a)) · 90°

TileL(n-1, P4)

x + sgn(cos(a)) · 2^n/4, y + sgn(sin(a)) · 2^n/4, a

TABLE OF CONTENTS — SL03

1. Divide and conquer

2. Recurrences
   - Substitution
   - Recursion tree
   - Master method recipe

RECURRENCES / 1

Informal definition

- Running times of algorithms with recursive calls can be described using recurrences

RECURRENCES/1

Informal definition

- Running times of algorithms with recursive calls can be described using recurrences
- A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs

RECURRENCES / 1

Informal definition

- Running times of algorithms with recursive calls can be described using recurrences
- A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs
- Example: the running time of merge sort is described by the following recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

RECURRENCES / 2

Solving recurrences

1. Repeated (backward) substitution
   - Expand the recurrence by substitution and then notice the pattern

RECURRENCES/2

Solving recurrences

1. Repeated (backward) substitution
   - Expand the recurrence by substitution and then notice the pattern
2. Substitution method
   - Guess a bound and then use induction to prove that the guess is correct

RECURRENCES/2

Solving recurrences

1. Repeated (backward) substitution
   - Expand the recurrence by substitution and then notice the pattern
2. Substitution method
   - Guess a bound and then use induction to prove that the guess is correct
3. Recursion trees
   - Convert a recurrence in a tree whose nodes represent the costs

RECURRENCES/2

Solving recurrences

1. Repeated (backward) substitution
   - Expand the recurrence by substitution and then notice the pattern
2. Substitution method
   - Guess a bound and then use induction to prove that the guess is correct
3. Recursion trees
   - Convert a recurrence in a tree whose nodes represent the costs
4. Master method
   - Templates for different classes of recurrences

RECURRENCES / 3

Merge sort revisited

- For $n > 1$, let $T(n) = 2T(n/2) + cn$

$= \Theta(n)$

RECURRENCES / 3

Merge sort revisited

- For $n > 1$, let $T(n) = 2T(n/2) + cn$
- Apply repeated substitution

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn && \text{(substitute)}\\
&= 2(2T(n/4) + cn/2) + cn && \text{(expand)}\\
&= 4T(n/4) + 2cn && \text{(substitute)}\\
&= 4(2T(n/8) + cn/4) + 2cn && \text{(expand)}\\
&= 8T(n/8) + 3cn && \text{(find pattern)}
\end{aligned}
$$

RECURRENCES / 3

Merge sort revisited

- For $n > 1$, let $T(n) = 2T(n/2) + cn$
- Apply repeated substitution

$$\begin{aligned} T(n) &= 2T(n/2) + cn && \text{(substitute)} \\ &= 2(2T(n/4) + cn/2) + cn && \text{(expand)} \\ &= 4T(n/4) + 2cn && \text{(substitute)} \\ &= 4(2T(n/8) + cn/4) + 2cn && \text{(expand)} \\ &= 8T(n/8) + 3cn && \text{(find pattern)} \end{aligned}$$

- From $8T(n/8) + 3cn$ one gets $2^iT(n/2^i) + icn$

Recurrences / 3

Merge sort revisited

- For $n > 1$, let $T(n) = 2T(n/2) + cn$
- Apply repeated substitution

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn && \text{(substitute)}\\
&= 2(2T(n/4) + cn/2) + cn && \text{(expand)}\\
&= 4T(n/4) + 2cn && \text{(substitute)}\\
&= 4(2T(n/8) + cn/4) + 2cn && \text{(expand)}\\
&= 8T(n/8) + 3cn && \text{(find pattern)}
\end{aligned}
$$

- From $8T(n/8) + 3cn$ one gets $2^iT(n/2^i) + icn$
- $i = \lg n$ yields $T(n) = nT(1) + cn \lg n = \Theta(n \lg n)$

RECURRENCES / 4

Repeated substitution

- Is not a strictly formal proof

RECURRENCES / 4

Repeated substitution

- Is not a strictly formal proof
- The procedure is straightforward
  - Substitute, expand, substitute, expand, etc.
  - Observe a pattern and determine the expression after the i-th substitution
  - Find out what the highest value of i should be to get the base case of the recurrence
  - Insert the cost of the base case and the expression for i into the observed expression

RECURRENCES / 5

A more precise running time for sort merge

- Recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + 2n + 3 & \text{if } n > 1 \end{cases}$$

copy to auxiliary array and back

} procedure calls

RECURRENCES / 5

A more precise running time for sort merge

- Recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + 2n + 3 & \text{if } n > 1 \end{cases}$$

- Observe pattern

$$T(n) = 2T(n/2) + 2n + 3 \qquad \text{(substitute)}$$

RECURRENCES / 5

A more precise running time for sort merge

- Recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + 2n + 3 & \text{if } n > 1 \end{cases}$$

- Observe pattern

$$\begin{aligned} T(n) &= 2T(n/2) + 2n + 3 && \text{(substitute)} \\ &= 2(2T(n/4) + n + 3) + 2n + 3 && \text{(expand)} \end{aligned}$$

RECURRENCES / 5

A more precise running time for sort merge

- Recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + 2n + 3 & \text{if } n > 1 \end{cases}$$

- Observe pattern

$$\begin{aligned} T(n) &= 2T(n/2) + 2n + 3 && \text{(substitute)} \\ &= 2(2T(n/4) + n + 3) + 2n + 3 && \text{(expand)} \\ &= 4T(n/4) + 4n + 9 && \text{(substitute)} \end{aligned}$$

RECURRENCES / 5

A more precise running time for sort merge

- Recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + 2n + 3 & \text{if } n > 1 \end{cases}$$

- Observe pattern

$$\begin{aligned} T(n) &= 2T(n/2) + 2n + 3 && \text{(substitute)} \\ &= 2(2T(n/4) + n + 3) + 2n + 3 && \text{(expand)} \\ &= 4T(n/4) + 4n + 9 && \text{(substitute)} \\ &= 4(2T(n/8) + n/2 + 3) + 4n + 9 && \text{(expand)} \end{aligned}$$

RECURRENCES / 5

A more precise running time for sort merge

- Recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + 2n + 3 & \text{if } n > 1 \end{cases}$$

- Observe pattern

$$\begin{aligned} T(n) &= 2T(n/2) + 2n + 3 && \text{(substitute)} \\ &= 2(2T(n/4) + n + 3) + 2n + 3 && \text{(expand)} \\ &= 4T(n/4) + 4n + 9 && \text{(substitute)} \\ &= 4(2T(n/8) + n/2 + 3) + 4n + 9 && \text{(expand)} \\ &= 8T(n/8) + 3 \cdot 2n + (4 + 2 + 1)3 && \text{(find pattern)} \end{aligned}$$

Recurrences / 6

- Find pattern:

$$\begin{aligned} T(n) &= 8T(n/8) + 3 \cdot 2n + (4+2+1)3 \\ &= 2^i T(n/2^i) + 2in + 3\sum_{j=0}^{i-1} 2^j \end{aligned}$$

RECURRENCES / 6

- Find pattern:

$$\begin{aligned} T(n) &= 8T(n/8) + 3 \cdot 2n + (4+2+1)3 \\ &= 2^i T(n/2^i) + 2in + 3\sum_{j=0}^{i-1} 2^j \end{aligned}$$

- Upper bound for i is $\lg n$
- Manipulate expression (remember: $\sum_{k=0}^{n} x^k = \frac{1-x^{n+1}}{1-x}$)

$$T(n) = 2^{\lg n} T(n/2^{\lg n}) + 2n \lg n + 3(2^{\lg n} - 1)$$

RECURRENCES / 6

- Find pattern:

$$\begin{aligned} T(n) &= 8T(n/8) + 3 \cdot 2n + (4+2+1)3 \\ &= 2^i T(n/2^i) + 2in + 3\textstyle\sum_{j=0}^{i-1} 2^j \end{aligned}$$

- Upper bound for i is $\lg n$
- Manipulate expression (remember: $\sum_{k=0}^{n} x^k = \frac{1-x^{n+1}}{1-x}$)

$$\begin{aligned} T(n) &= 2^{\lg n} T(n/2^{\lg n}) + 2n \lg n + 3(2^{\lg n} - 1) \\ &= 4n + 2n \lg n - 3 = \Theta(n \lg n) \end{aligned}$$

RECURRENCES / 6

- Find pattern:

$$\begin{aligned} T(n) &= 8T(n/8) + 3 \cdot 2n + (4+2+1)3 \\ &= 2^i T(n/2^i) + 2in + 3\textstyle\sum_{j=0}^{i-1} 2^j \end{aligned}$$

- Upper bound for i is $\lg n$
- Manipulate expression (remember: $\sum_{k=0}^{n} x^k = \frac{1-x^{n+1}}{1-x}$)

$$\begin{aligned} T(n) &= 2^{\lg n} T(n/2^{\lg n}) + 2n \lg n + 3(2^{\lg n} - 1) \\ &= 4n + 2n \lg n - 3 = \Theta(n \lg n) \end{aligned}$$

- Same asymptotic bound as for

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

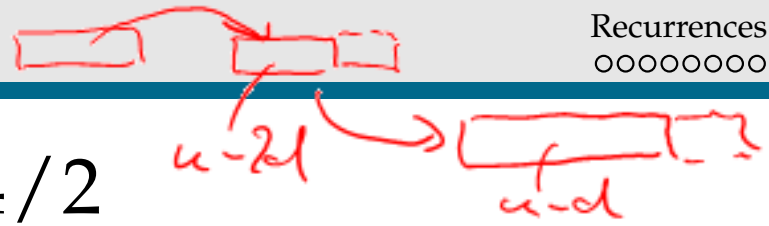
EXERCISE SL03-4/1

A program reads numbers into an array until a zero arrives. No assumption about the input can be made. The program starts with an array of size $n = 1$. If the array is full a larger array of size n' is created, the data is copied, and the old array is destroyed. Consider the following strategies:

- $n' = n + d$
- $n' = k * n$



Determine the cost of each strategy.



EXERCISE SL03-4/2

n-d array is increased to n array; n-d elements must be copied

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \quad \text{(we start with array of size 1)} \\ T(n-d) + n - d & \text{if } n > 1 \end{cases}$$

$$T(n) = T(n-d) + n - d$$

$$= T(n-2d) + n - 2d + n - d$$

$$= T(n-2d) + 2n - d(1+2)$$

$$= T(n-id) + in - d\sum_{j=1}^{i} j \qquad \left| \begin{array}{l} n - id = 1 \\ i = \frac{n-1}{d} \end{array} \right.$$

$$i_{max} = \frac{n-1}{d}$$

$$= 0 + \frac{(n-1)n}{d} - \frac{d}{2}\,\frac{n-1}{d}\left(\frac{n-1}{d} + 1\right)$$

$$= \ldots \frac{n^2 - 1 - dn + d}{2d} = \Theta(n^2)$$

EXERCISE SL03-4/3

$n \rightsquigarrow k \cdot n$

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T(n/k) + n/k \end{cases}$$

$$T(n) = T(n/k) + n/k$$

$$= T(n/k^2) + n/k^2 + n/k$$

$$= T(n/k^i) + \sum_{j=1}^{i} n/k^j \qquad i_{max} = \log_k n$$

$$= T(1) + \sum_{j=0}^{k \log_k n} (n/k^j) - n$$

$$= \frac{n-1}{k-1} = \Theta(n)$$

SUBSTITUTION / 1

Basic methodology

- The substitution method for solving recurrences entails two steps
  1. Guess the form of the solution
  2. Use induction to prove the solution

SUBSTITUTION / 1

Basic methodology

- The substitution method for solving recurrences entails two steps
  1. Guess the form of the solution
  2. Use induction to prove the solution

- Example: consider the recurrence $T(n) = 4T(n/2) + n$

SUBSTITUTION/1

Basic methodology

- The substitution method for solving recurrences entails two steps
  1. Guess the form of the solution
  2. Use induction to prove the solution

- Example: consider the recurrence $T(n) = 4T(n/2) + n$
- Guessed solution: $T(n) = n^3$, i.e. $T(n)$ is of the form n^3

SUBSTITUTION / 1

Basic methodology

- The substitution method for solving recurrences entails two steps
  1. Guess the form of the solution
  2. Use induction to prove the solution

- Example: consider the recurrence $T(n) = 4T(n/2) + n$
- Guessed solution: $T(n) = n^3$, i.e. $T(n)$ is of the form n^3
- Mathematical induction: need to show that $T(n) \leq cn^3$

SUBSTITUTION / 2

Proof by induction

- Need to show that $T(n) \leq cn^3$ for $T(n) = 4T(n/2) + n$

$T(n) = 4T(n/2) + n$ (recurrence)

SUBSTITUTION / 2

Proof by induction

- Need to show that $T(n) \leq cn^3$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n && \text{(recurrence)} \\
&\leq 4c(n/2)^3 + n && \text{(inductive hyp.)}
\end{aligned}
$$

SUBSTITUTION/2

Proof by induction

- Need to show that $T(n) \leq cn^3$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n && \text{(recurrence)} \\
&\leq 4c(n/2)^3 + n && \text{(inductive hyp.)} \\
&= cn^3/2 + n && \text{(simplification)}
\end{aligned}
$$

SUBSTITUTION/2

Proof by induction

- Need to show that $T(n) \leq cn^3$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n && \text{(recurrence)} \\
&\leq 4c(n/2)^3 + n && \text{(inductive hyp.)} \\
&= cn^3/2 + n && \text{(simplification)} \\
&= cn^3 - (cn^3/2 - n) && \text{(rearrangement)}
\end{aligned}
$$

> 0

SUBSTITUTION/2

Proof by induction

- Need to show that $T(n) \leq cn^3$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n && \text{(recurrence)} \\
&\leq 4c(n/2)^3 + n && \text{(inductive hyp.)} \\
&= cn^3/2 + n && \text{(simplification)} \\
&= cn^3 - (cn^3/2 - n) && \text{(rearrangement)} \\
&\leq cn^3 && \text{(for } c \geq 2, n \geq 1\text{)}
\end{aligned}
$$

SUBSTITUTION/2

step: $n/2 \to n$

Proof by induction

- Need to show that $T(n) \le cn^3$ for $T(n) = 4T(n/2) + n$, $T(1) = 0$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n && \text{(recurrence)}\\
&\le 4c(n/2)^3 + n && \text{(inductive hyp.)}\\
&= cn^3/2 + n && \text{(simplification)}\\
&= cn^3 - (cn^3/2 - n) && \text{(rearrangement)}\\
&\le cn^3 && \text{(for } c \ge 2, n \ge 1\text{)}
\end{aligned}
$$

$\ge c$

OK for $c = 2$

- It follows that $T(n) = O(n^3)$

$T(1) = 0 \le c \cdot 1^3$

$T(2) = 2 \le c \cdot 8$

$T(4) = 12 \le c \cdot 64$

EXERCISE SL03-5

step: $n \to n/k$

Use the substitution method to prove that by increasing an array by a factor k each time it is filled up yields a linear asymptotic runtime complexity.

$$T(n) = T(n/k) + n/k$$

hypothesis: $T(n) \leq cn$

$$T(n) = T(n/k) + n/k$$
$$\leq cn/k + n/k$$
$$= \frac{n(c+1)}{k} \stackrel{?}{\leq} cn$$

yes for $c = k$, $k \geq 2$

SUBSTITUTION / 3

Getting tighter bound / 1

- Try to show that $T(n) \leq cn^2$ for $T(n) = 4T(n/2) + n$

$$T(n) = 4T(n/2) + n \qquad \text{(recurrence)}$$

SUBSTITUTION / 3

Getting tighter bound / 1

- Try to show that $T(n) \leq cn^2$ for $T(n) = 4T(n/2) + n$

$$\begin{aligned} T(n) &= 4T(n/2) + n && \text{(recurrence)} \\ &\leq 4c(n/2)^2 + n && \text{(inductive hyp.)} \end{aligned}$$

SUBSTITUTION / 3

Getting tighter bound / 1

- Try to show that $T(n) \leq cn^2$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n && \text{(recurrence)} \\
&\leq 4c(n/2)^2 + n && \text{(inductive hyp.)} \\
&= cn^2 + n && \text{(simplification)}
\end{aligned}
$$

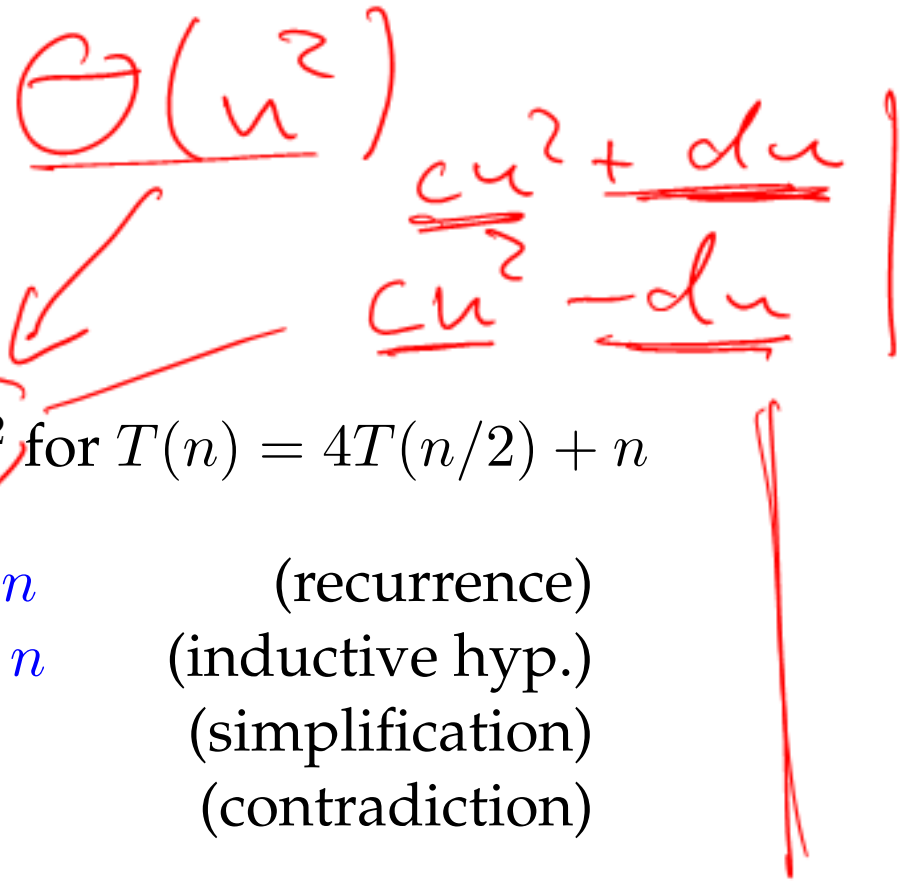
SUBSTITUTION / 3

Getting tighter bound / 1

- Try to show that $T(n) \leq cn^2$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n && \text{(recurrence)} \\
&\leq 4c(n/2)^2 + n && \text{(inductive hyp.)} \\
&= cn^2 + n && \text{(simplification)} \\
&\not\leq cn^2 && \text{(contradiction)}
\end{aligned}
$$

SUBSTITUTION/3



Getting tighter bound/1

- Try to show that $T(n) \leq cn^2$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n && \text{(recurrence)} \\
&\leq 4c(n/2)^2 + n && \text{(inductive hyp.)} \\
&= cn^2 + n && \text{(simplification)} \\
&\not\leq cn^2 && \text{(contradiction)}
\end{aligned}
$$

- It seems that induction does not work!?

SUBSTITUTION/4

Getting tighter bound/2

- Underlying problem: trying to rewrite

$$T(n) = O(n^2) = cn^2 + (\text{something positive})$$

as $T(n) \leq cn^2$ does not work in the inductive proof

SUBSTITUTION/4

Getting tighter bound/2

- Underlying problem: trying to rewrite

$$T(n) = O(n^2) = cn^2 + (\text{something positive})$$

as $T(n) \leq cn^2$ does not work in the inductive proof

- Solution: strengthen hypothesis for inductive proof

$$T(n) \leq (\text{answer you want}) - (\text{something positive})$$

SUBSTITUTION/5

Getting tighter bound/3

- Show that $T(n) \leq cn^2 - dn$ for $T(n) = 4T(n/2) + n$

$T(n) = 4T(n/2)+n$ (recurrence)

SUBSTITUTION / 5

Getting tighter bound / 3

- Show that $T(n) \leq cn^2 - dn$ for $T(n) = 4T(n/2) + n$

$$\begin{aligned} T(n) &= 4T(n/2)+n && \text{(recurrence)} \\ &\leq 4(c(n/2)^2 - dn/2) + n && \text{(inductive hyp.)} \end{aligned}$$

SUBSTITUTION / 5

Getting tighter bound / 3

- Show that $T(n) \leq cn^2 - dn$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2)+n && \text{(recurrence)} \\
&\leq 4(c(n/2)^2 - dn/2) + n && \text{(inductive hyp.)} \\
&= cn^2 - 2dn + n && \text{(simplification)}
\end{aligned}
$$

SUBSTITUTION / 5

Getting tighter bound / 3

- Show that $T(n) \leq cn^2 - dn$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2)+n && \text{(recurrence)}\\
&\leq 4(c(n/2)^2 - dn/2) + n && \text{(inductive hyp.)}\\
&= cn^2 - 2dn + n && \text{(simplification)}\\
&= cn^2 - dn - (dn - n) && \text{(rearrangement)}
\end{aligned}
$$

≥ 0 for $d \geq 1$

SUBSTITUTION / 5

Getting tighter bound / 3

- Show that $T(n) \leq cn^2 - dn$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2)+n && \text{(recurrence)} \\
&\leq 4(c(n/2)^2 - dn/2) + n && \text{(inductive hyp.)} \\
&= cn^2 - 2dn + n && \text{(simplification)} \\
&= cn^2 - dn - (dn - n) && \text{(rearrangement)} \\
&\leq cn^2 - dn && (\text{for } d \geq 1)
\end{aligned}
$$

SUBSTITUTION/5

$T(1) = 0 \leq c - d$
$T(2) = 2 \leq c \cdot 4 - d \cdot 2$
$T(4) = 12 \leq c \cdot 16 - d \cdot 4$

ok for $c = 1$, $d = 1$

Getting tighter bound/3

- Show that $T(n) \leq cn^2 - dn$ for $T(n) = 4T(n/2) + n$

$$
\begin{aligned}
T(n) &= 4T(n/2)+n && \text{(recurrence)}\\
&\leq 4(c(n/2)^2 - dn/2) + n && \text{(inductive hyp.)}\\
&= cn^2 - 2dn + n && \text{(simplification)}\\
&= cn^2 - dn - (dn - n) && \text{(rearrangement)}\\
&\leq cn^2 - dn && \text{(for } d \geq 1\text{)}
\end{aligned}
$$

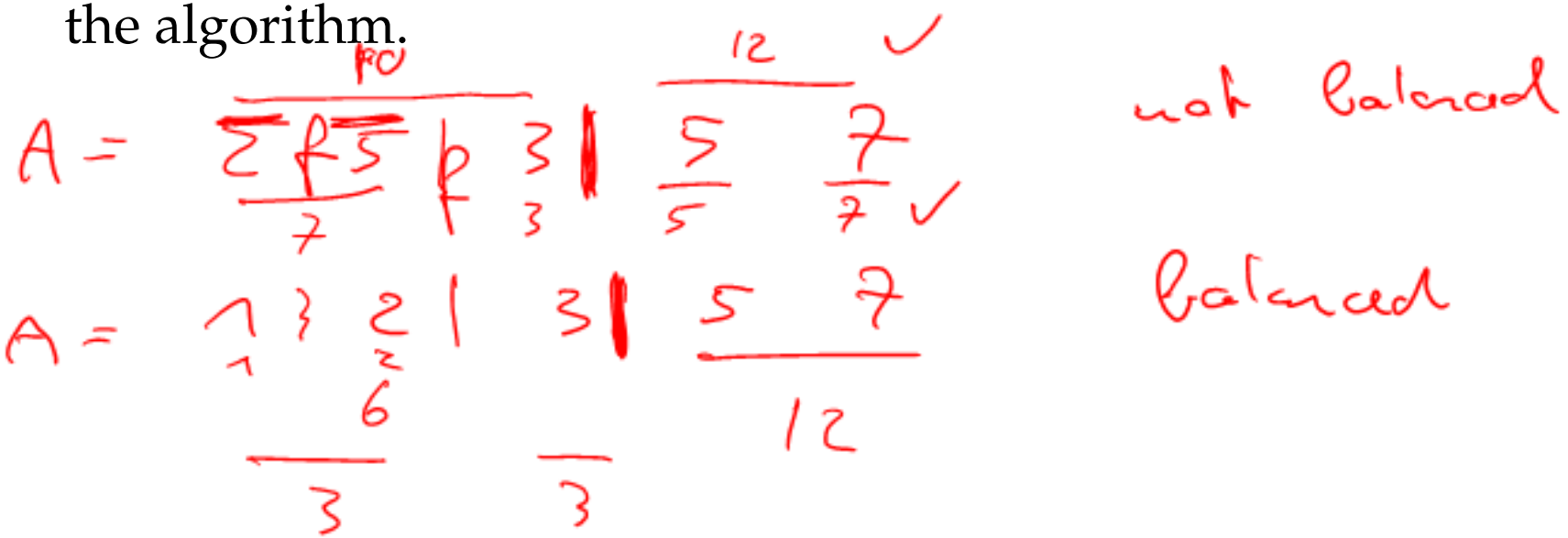
- It thus follows that $T(n) = O(n^2)$

EXERCISE SL03-6/1

An array $A[1..n]$ is balanced if

- the sum of the elements in the first half is no more than double and no less than half the sum of the elements in the second half, and
- the first and second half are balanced.

Give an algorithm that determines if an array is balanced. Use the substitution method to prove the asymptotic complexity of the algorithm.



EXERCISE SL03-6/2

$\ell = 1, \; r = 3, \; m = 2$

Algo Balanced(A, ℓ, r)
  if (ℓ == r) then return (TRUE, A[ℓ])
  m = ⌊(r + ℓ)/2⌋
  (BalLft, SumLft) = Balanced(A, ℓ, m)
  (BalRght, SumRght) = Balanced(A, m+1, r)
  return (BalLft && BalRght &&
    SumLft ≥ SumRght/2 &&
    SumLft ≤ SumRght + 2,
    SumLft + SumRght)

$1..2 \; 1$
$3..3 \; 2$
$2..2 \; 1$

EXERCISE SL03-6/3

$T(n) = 2T(n/2) + 1, \quad T(1) = 0$

Guess: $T(n) = \Theta(n)$, $T(n) \leq cn$!

$T(n) = 2T(n/2) + 1$
$\leq 2c\,n/2 + 1$
$= cn + 1$
$\not\leq cn$

$T(n) \leq c_1 n - c_2$

$T(n) = 2T(n/2) + 1$
$\leq 2(c_1 n/2 - c_2) + 1$
$= c_1 n - 2c_2 + 1$
$= c_1 n - c_2 - (c_2 - 1)$
$\leq c_1 n - c_2$ ok for $c_2 \geq 1$, $c_1 \geq 2$

RECURSION TREE/1

Basic methodology

- A recursion tree is a convenient way to visualize what happens when a recurrence is iterated

RECURSION TREE/1

Basic methodology

- A recursion tree is a convenient way to visualize what happens when a recurrence is iterated
- Good for guessing asymptotic solutions to recurrences

RECURSION TREE/1

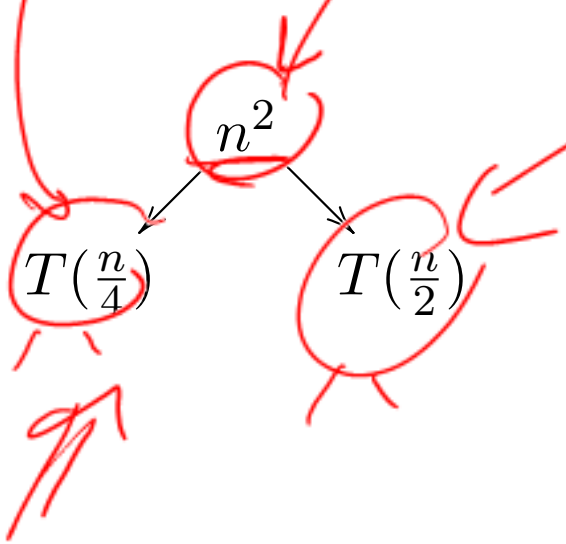
Basic methodology

- A recursion tree is a convenient way to visualize what happens when a recurrence is iterated
- Good for guessing asymptotic solutions to recurrences
- Example: consider $T(n) = T(n/4) + T(n/2) + n^2$

Recursion tree/1

Basic methodology

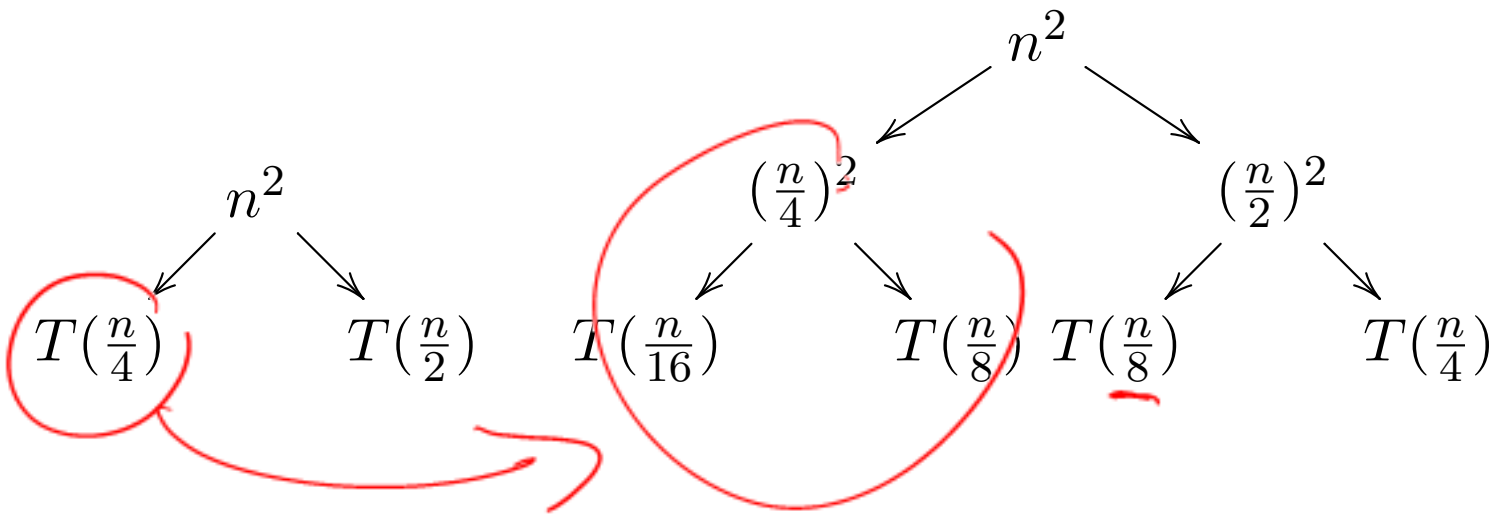
- A recursion tree is a convenient way to visualize what happens when a recurrence is iterated
- Good for guessing asymptotic solutions to recurrences
- Example: consider $T(n) = T(n/4) + T(n/2) + n^2$



Recursion tree/1

Basic methodology

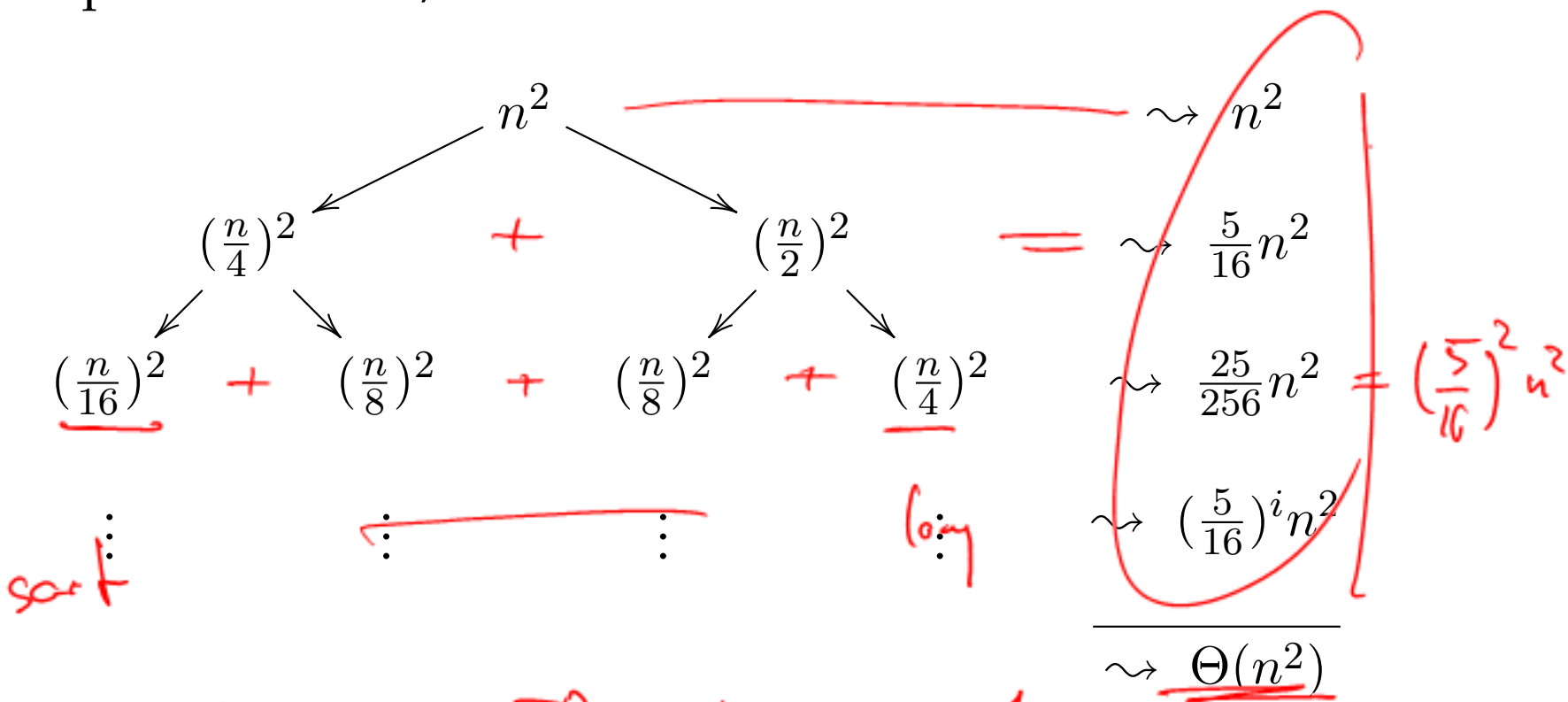
- A recursion tree is a convenient way to visualize what happens when a recurrence is iterated
- Good for guessing asymptotic solutions to recurrences
- Example: consider $T(n) = T(n/4) + T(n/2) + n^2$



Recursion tree/2

$$\sum_{i=0}^{\infty} a^i = \frac{1}{1-a} \quad \text{if } |a| < 1$$

Example recurrence/1



$$n^2 \sum_{i=0}^{\log n} \left(\frac{5}{16}\right)^i < n^2 \sum_{i=0}^{\infty} \left(\frac{5}{16}\right)^i = n^2 \frac{1}{1-5/16} = n^2 \frac{16}{11} \approx 1.5\, n^2$$

Recursion tree/3

Example recurrence/2

- Show that $T(n) \leq cn^2$ for $T(n) = T(n/2) + T(n/4) + n^2$

$$T(n) = T(n/2) + T(n/4) + n^2 \qquad \text{(recurrence)}$$

Recursion tree/3

Example recurrence/2

- Show that $T(n) \leq cn^2$ for $T(n) = T(n/2) + T(n/4) + n^2$

$$\begin{aligned} T(n) &= T(n/2) + T(n/4) + n^2 && \text{(recurrence)} \\ &\leq c(n/2)^2 + c(n/4)^2 + n^2 && \text{(inductive hyp.)} \end{aligned}$$

RECURSION TREE/3

Example recurrence/2

- Show that $T(n) \leq cn^2$ for $T(n) = T(n/2) + T(n/4) + n^2$

$$\begin{aligned} T(n) &= T(n/2) + T(n/4) + n^2 && \text{(recurrence)} \\ &\leq c(n/2)^2 + c(n/4)^2 + n^2 && \text{(inductive hyp.)} \\ &= 5cn^2/16 + n^2 && \text{(simplification)} \end{aligned}$$

Recursion tree/3

$\frac{5}{16} = 1 - \frac{11}{16}$

Example recurrence/2

- Show that $T(n) \leq cn^2$ for $T(n) = T(n/2) + T(n/4) + n^2$

$$\begin{aligned} T(n) &= T(n/2) + T(n/4) + n^2 && \text{(recurrence)} \\ &\leq c(n/2)^2 + c(n/4)^2 + n^2 && \text{(inductive hyp.)} \\ &= 5cn^2/16 + n^2 && \text{(simplification)} \\ &= cn^2 - 11cn^2/16 + n^2 && \text{(rearrangement)} \end{aligned}$$

Recursion tree/3

Example recurrence/2

- Show that $T(n) \leq cn^2$ for $T(n) = T(n/2) + T(n/4) + n^2$

$$
\begin{aligned}
T(n) &= T(n/2) + T(n/4) + n^2 && \text{(recurrence)} \\
&\leq c(n/2)^2 + c(n/4)^2 + n^2 && \text{(inductive hyp.)} \\
&= 5cn^2/16 + n^2 && \text{(simplification)} \\
&= cn^2 - 11cn^2/16 + n^2 && \text{(rearrangement)} \\
&\leq cn^2 && \text{(for } c \geq 16/11\text{)}
\end{aligned}
$$

$-11cn^2/16 + n^2 \leq 0$

RECURSION TREE/3

Example recurrence/2

- Show that $T(n) \leq cn^2$ for $T(n) = T(n/2) + T(n/4) + n^2$

$$\begin{aligned} T(n) &= T(n/2) + T(n/4) + n^2 && \text{(recurrence)} \\ &\leq c(n/2)^2 + c(n/4)^2 + n^2 && \text{(inductive hyp.)} \\ &= 5cn^2/16 + n^2 && \text{(simplification)} \\ &= cn^2 - 11cn^2/16 + n^2 && \text{(rearrangement)} \\ &\leq cn^2 && (\text{for } c \geq 16/11) \end{aligned}$$

- It thus follows that $T(n) = O(n^2)$

MASTER METHOD/1

- To solve a class of recurrences that have the form
$$T(n) = aT(n/b) + f(n)$$

MASTER METHOD/1

- To solve a class of recurrences that have the form $T(n) = aT(n/b) + f(n)$
- Assumptions:
  - $a \geq 1$,
  - $b > 1$, and
  - $f(n)$ asymptotically positive

MASTER METHOD/1

$T(n) = 2T(n/2) + n$

- To solve a class of recurrences that have the form $T(n) = aT(n/b) + f(n)$
- Assumptions:
  - $a \geq 1$,
  - $b > 1$, and
  - $f(n)$ asymptotically positive
- $T(n)$ describes running time of a divide-and-conquer algorithm
  - a subproblems of size n/b are solved recursively, each in time $T(n/b)$
  - $f(n)$ is the cost of dividing the problem and combining the solutions

MASTER METHOD/2

- Iterating the recurrence (expanding the tree) yields

$$T(n) = f(n) + aT(n/b)$$

MASTER METHOD/2

- Iterating the recurrence (expanding the tree) yields

$$\begin{aligned} T(n) &= f(n) + aT(n/b) \\ &= f(n) + af(n/b) + aT(n/b^2) \end{aligned}$$

MASTER METHOD/2

- Iterating the recurrence (expanding the tree) yields

$$
\begin{aligned}
T(n) &= f(n) + aT(n/b) \\
&= f(n) + af(n/b) + aT(n/b^2) \\
&= f(n) + af(n/b) + a^2 f(n/b^2) + \ldots \\
&\quad + a^{\log_b n - 1} f(n/b^{\log_b n - 1}) + a^{\log_b n} T(1)
\end{aligned}
$$

non-leaves

leaves

MASTER METHOD/2

- Iterating the recurrence (expanding the tree) yields

$$\begin{aligned} T(n) &= f(n) + aT(n/b) \\ &= f(n) + af(n/b) + aT(n/b^2) \\ &= f(n) + af(n/b) + a^2 f(n/b^2) + \ldots \\ &\quad + a^{\log_b n - 1} f(n/b^{\log_b n - 1}) + a^{\log_b n} T(1) \\ &= \underbrace{\textstyle\sum_{i=0}^{\log_b n - 1} a^i f(n/b^i)}_{\text{non-leaves}} + \underbrace{\Theta(n^{\log_b a})}_{\text{leaves}} \end{aligned}$$

MASTER METHOD/2

- Iterating the recurrence (expanding the tree) yields

$$\begin{aligned} T(n) &= f(n) + aT(n/b) \\ &= f(n) + af(n/b) + aT(n/b^2) \\ &= f(n) + af(n/b) + a^2 f(n/b^2) + \ldots \\ &\quad + a^{\log_b n - 1} f(n/b^{\log_b n - 1}) + a^{\log_b n} T(1) \\ &= \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i) + \Theta(n^{\log_b a}) \end{aligned}$$

- The first term is a division/combination cost (totaled across all levels of the tree)

MASTER METHOD/2

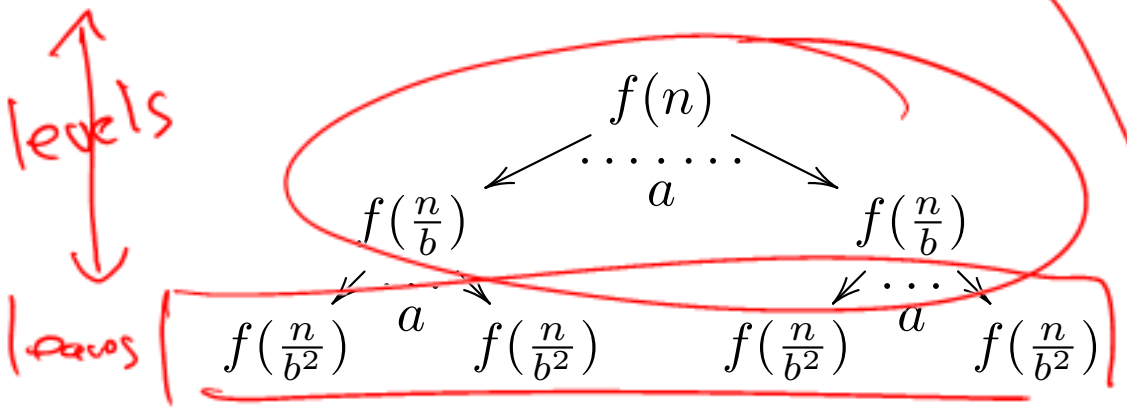
- Iterating the recurrence (expanding the tree) yields

$$\begin{aligned} T(n) &= f(n) + aT(n/b) \\ &= f(n) + af(n/b) + aT(n/b^2) \\ &= f(n) + af(n/b) + a^2 f(n/b^2) + \ldots \\ &\quad + a^{\log_b n - 1} f(n/b^{\log_b n - 1}) + a^{\log_b n} T(1) \\ &= \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i) + \Theta(n^{\log_b a}) \end{aligned}$$

- The first term is a division/combination cost (totaled across all levels of the tree)
- The second term is the cost of solving all the trivial subproblems (total of all work pushed to the leaves)

MASTER METHOD / 3

Expanding the tree



$\rightsquigarrow f(n)$

$\rightsquigarrow af(\frac{n}{b})$

$\rightsquigarrow a^2 f(\frac{n}{b^2})$

$\rightsquigarrow a^i f(\frac{n}{b^i})$

- Note:
  - $\log_b n$ levels,
  - $a^{\log_b n} = n^{\log_b a}$ leaves
- Running time: $T(n) = \Theta(n^{\log_b a}) + \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i)$

MASTER METHOD/4

Underlying intuitions

- Three possible cases
  - Running time dominated by the cost at the root (case 3)
  - Running time dominated by the cost at the leaves (case 1)
  - Running time evenly distributed throughout tree (case 2)

MASTER METHOD/4

Underlying intuitions

- Three possible cases
  - Running time dominated by the cost at the root (case 3)
  - Running time dominated by the cost at the leaves (case 1)
  - Running time evenly distributed throughout tree (case 2)
- Solve recurrence by identifying dominant term

MASTER METHOD/4

Underlying intuitions

- Three possible cases
  - Running time dominated by the cost at the root (case 3)
  - Running time dominated by the cost at the leaves (case 1)
  - Running time evenly distributed throughout tree (case 2)
- Solve recurrence by identifying dominant term
- In each case compare $f(n)$ with $n^{\log_b a}$

MASTER METHOD/4

Underlying intuitions

- Three possible cases
  - Running time dominated by the cost at the root (case 3)
  - Running time dominated by the cost at the leaves (case 1)
  - Running time evenly distributed throughout tree (case 2)
- Solve recurrence by identifying dominant term
- In each case compare $f(n)$ with $n^{\log_b a}$

MASTER METHOD/5

Master theorem, case 2

- $f(n) = \Theta(n^{\log_b a})$ (i.e., $f(n)$ and $n^{\log_b a}$ are asymptotically the same)

MASTER METHOD/5

Master theorem, case 2

- $f(n) = \Theta(n^{\log_b a})$ (i.e., $f(n)$ and $n^{\log_b a}$ are asymptotically the same)
- The work is evenly distributed: $T(n) = \Theta(n^{\log_b a} \log_b n)$

resulting asymp complexity

MASTER METHOD/5

Master theorem, case 2

- $f(n) = \Theta(n^{\log_b a})$ (i.e., $f(n)$ and $n^{\log_b a}$ are asymptotically the same)
- The work is evenly distributed: $T(n) = \Theta(n^{\log_b a} \log_b n)$

Example: $T(n) = 2T(n/2) + \Theta(n)$ (merge sort)

- $a = 2$, $b = 2$, $f(n) = \Theta(n)$
- $n^{\log_2 2} = n$ — determine $n^{\log_b a}$
- $\Theta(n) = \Theta(n)$
- case 2: $T(n) = \Theta(n \lg n)$

MASTER METHOD / 6

Master theorem, case 1

- $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$
  - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by factor n^ϵ)

MASTER METHOD/6

Master theorem, case 1

- $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$
  - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by factor n^ϵ)
- The work at the leaves dominates: $T(n) = \Theta(n^{\log_b a})$

MASTER METHOD/6

Master theorem, case 1

- $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$
  - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by factor n^ϵ)
- The work at the leaves dominates: $T(n) = \Theta(n^{\log_b a})$

Example: $T(n) = 9T(n/3) + n$

- $a = 9, b = 3, f(n) = n$
- $n^{\log_3 9} = n^2$
- $n = O(n^{\log_3 9 - 1})$
- case 1: $T(n) = \Theta(n^2)$

MASTER METHOD/7

Master theorem, case 3

- $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$
  - $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by factor n^ϵ)
  - Regularity: $\exists c < 1 . \exists n_0 > 0 . \forall n > n_0 . af(n/b) \leq cf(n)$

MASTER METHOD/7

Master theorem, case 3

- $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$
  - $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by factor n^ϵ)
  - Regularity: $\exists c < 1 . \exists n_0 > 0 . \forall n > n_0 . af(n/b) \leq cf(n)$
- The work at the root dominates: $T(n) = \Theta(f(n))$

MASTER METHOD/7

Master theorem, case 3

- $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$
  - $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by factor n^ϵ)
  - Regularity: $\exists c < 1.\exists n_0 > 0.\forall n > n_0.af(n/b) \leq cf(n)$
- The work at the root dominates: $T(n) = \Theta(f(n))$

Example: $T(n) = 3T(n/4) + n \lg n$

- $a = 3, b = 4, f(n) = n \lg n$
- $n^{\log_4 3} = n^{0.792}$
- $n \lg n = \Omega(n^{\log_4 3+0.208})$
- case 3: $T(n) = \Theta(n \lg n)$
- Need also to verify regularity condition: $af(n/b) \leq cf(n)$

$$c = 3/4 : af(n/b) = 3(n/4)\lg(n/4) \leq (3/4)n \lg n$$

MASTER METHOD / 8

Master theorem, overview

- Given a recurrence of the form $T(n) = aT(n/b) + f(n)$

MASTER METHOD / 8

Master theorem, overview

- Given a recurrence of the form $T(n) = aT(n/b) + f(n)$
- Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ yields $T(n) = \Theta(n^{\log_b a})$

MASTER METHOD / 8

Master theorem, overview

- Given a recurrence of the form $T(n) = aT(n/b) + f(n)$
- Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ yields $T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a})$ yields $T(n) = \Theta(n^{\log_b a} \log_b n)$

MASTER METHOD / 8

Master theorem, overview

- Given a recurrence of the form $T(n) = aT(n/b) + f(n)$
- Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ yields $T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a})$ yields $T(n) = \Theta(n^{\log_b a} \log_b n)$
- Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ and $af(n/b) \leq cf(n)$ ($c < 1$, $n > n_0$) yield $T(n) = \Theta(f(n))$

MASTER METHOD / 8

Master theorem, overview

- Given a recurrence of the form $T(n) = aT(n/b) + f(n)$
- Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ yields $T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a})$ yields $T(n) = \Theta(n^{\log_b a} \log_b n)$
- Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ and $af(n/b) \leq cf(n)$ ($c < 1$, $n > n_0$) yield $T(n) = \Theta(f(n))$

General strategy

- Extract a, b, and $f(n)$ from given recurrence

MASTER METHOD / 8

Master theorem, overview

- Given a recurrence of the form $T(n) = aT(n/b) + f(n)$
- Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ yields $T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a})$ yields $T(n) = \Theta(n^{\log_b a} \log_b n)$
- Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ and $af(n/b) \leq cf(n)$ ($c < 1$, $n > n_0$) yield $T(n) = \Theta(f(n))$

General strategy

- Extract a, b, and $f(n)$ from given recurrence
- Determine $n^{\log_b a}$; asymptotically compare it with $f(n)$

MASTER METHOD/8

Master theorem, overview

- Given a recurrence of the form $T(n) = aT(n/b) + f(n)$
- Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ yields $T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a})$ yields $T(n) = \Theta(n^{\log_b a} \log_b n)$
- Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ and $af(n/b) \leq cf(n)$ ($c < 1$, $n > n_0$) yield $T(n) = \Theta(f(n))$

General strategy

- Extract a, b, and $f(n)$ from given recurrence
- Determine $n^{\log_b a}$; asymptotically compare it with $f(n)$
- Determine and apply appropriate master theorem case

EXERCISE SL03-7/1

Use the Master Method to solve the recurrence

$$T(n) = 2T(n/2) + n^3$$

- $a = 2 \quad b = 2 \quad f(n) = n^3$
- $n^{\log_b a} = n^{2 \log_2 2} = n = n^1$
- $n^3 = \Omega(n^{1+\varepsilon})$ for $\varepsilon = 2$
- $2\left(\frac{n}{2}\right)^3 \leq cn^3$

  $\frac{1}{4}n^3 \leq cn^3$ holds for $c = 1/2$
- case 3: $T(n) = \Theta(n^3)$

SUMMARY / 1

- Divide and conquer is an algorithm design paradigm that is based on (multiple) recursion.
- A divide and conquer algorithm recursively breaks down a problem into two or more sub-problems of the same type, until these become simple enough to be solved directly.
- The solutions to the sub-problems are combined to give a solution to the original problem.
- Divide and conquer is the basis for many important problems (e.g., quicksort, merge sort, number multiplication, parsing, discrete Fourier transform).
- Understand and designing divide and conquer algorithms is a skill that takes time to learn (practice!).

SUMMARY/2

- The correctness of a divide and conquer algorithm is proved by induction.
- The computational cost of a divide and conquer algorithm is determined by solving recurrence relations.
- A recurrence relation is an equation that recursively defines a sequence of values.
- Thus, from one or more initial terms, each further term of the sequence is defined as a function of the preceding terms.
- We have seen four techniques to solve recurrences:
  - Repeated substitution (not a formal proof)
  - Substitution method (formal proof)
  - Recursion trees
  - Master method (recipe for most common cases)

Informatik II

Heap Sort, Quick Sort
— SL04 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

SORTING/1

Why sorting?

- Most fundamental problem in the study of algorithms

SORTING/1

Why sorting?

- Most fundamental problem in the study of algorithms
- Algorithms often employ sorting as a key subroutine

SORTING/1

Why sorting?

- Most fundamental problem in the study of algorithms
- Algorithms often employ sorting as a key subroutine
- Wide range of sorting algorithms, rich set of techniques
  - Incremental: selection sort, insertion sort
  - Divide-and-conquer: merge sort, quick sort
  - Use of particular data structure: heap sort

SORTING/1

Why sorting?

- Most fundamental problem in the study of algorithms
- Algorithms often employ sorting as a key subroutine
- Wide range of sorting algorithms, rich set of techniques
  - Incremental: selection sort, insertion sort
  - Divide-and-conquer: merge sort, quick sort
  - Use of particular data structure: heap sort
- Lower bound for sorting $\Omega(n \lg n)$ is used to prove lower bounds for other algorithmic problems

SORTING/2

Selection and insertion sort

- Worst case running time $\Theta(n^2)$

SORTING/2

Selection and insertion sort

- Worst case running time $\Theta(n^2)$
- Both are in place sorting algorithms

SORTING/2

Selection and insertion sort

- Worst case running time $\Theta(n^2)$
- Both are in place sorting algorithms

Other sorting algorithms

- Merge sort
  - Worst case $\Theta(n \lg n)$, requires extra memory

SORTING/2

Selection and insertion sort

- Worst case running time $\Theta(n^2)$
- Both are in place sorting algorithms

Other sorting algorithms

- Merge sort
  - Worst case $\Theta(n \lg n)$, requires extra memory
- Heap sort
  - Worst case $\Theta(n \lg n)$, in place sorting algorithm

SORTING/2

Selection and insertion sort

- Worst case running time $\Theta(n^2)$
- Both are in place sorting algorithms

Other sorting algorithms

- Merge sort
  - Worst case $\Theta(n \lg n)$, requires extra memory
- Heap sort
  - Worst case $\Theta(n \lg n)$, in place sorting algorithm
- Quick sort
  - Average case $\Theta(n \lg n)$, worst case $\Theta(n^2)$, in place

state-of-the-art

TABLE OF CONTENTS — SL04

1. Heap sort
   - Trees and heaps
   - Heapify
   - Building a heap
   - Heap sort algorithm

2. Quick sort

HEAP SORT

Basic intuition

- Sketch of the selection sort algorithm

```
for i = 1 to n do
    find smallest element in A[i ... n];
    swap smallest element with A[i];
```

HEAP SORT

Basic intuition

- Sketch of the selection sort algorithm

```
for i = 1 to n do
    find smallest element in A[i ... n];
    swap smallest element with A[i];
```

- Line 2 takes $\Theta(n)$, line 3 takes $\Theta(1)$: $\Theta(n^2)$ in total

HEAP SORT

Basic intuition

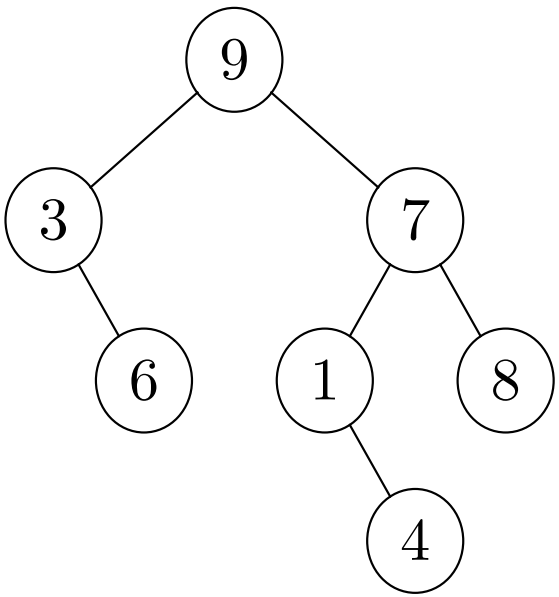
- Sketch of the selection sort algorithm

```
for i = 1 to n do
    find smallest element in A[i ... n];
    swap smallest element with A[i];
```

- Line 2 takes $\Theta(n)$, line 3 takes $\Theta(1)$: $\Theta(n^2)$ in total
- Idea for improvement: smart data structure to
  - accomplish the task of lines 2 and 3 in $\Theta(1)$
  - maintain the data structure in time $O(\lg n)$
  - obtain a total running time of $O(n \lg n)$

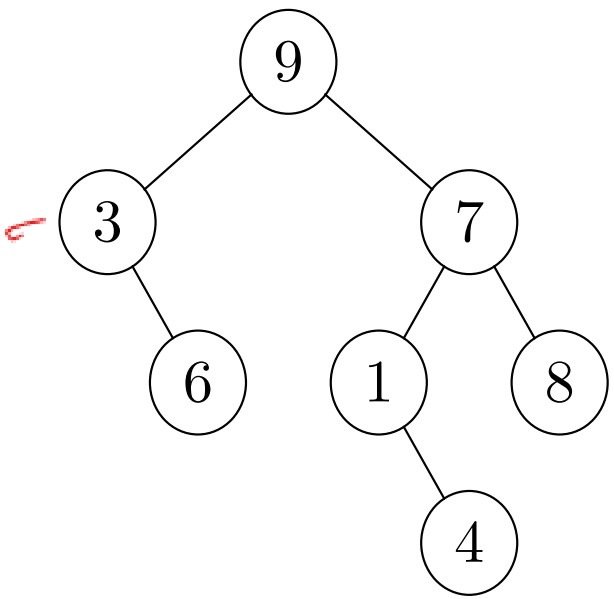
Trees and heaps/1

Binary trees/1



TREES AND HEAPS/1

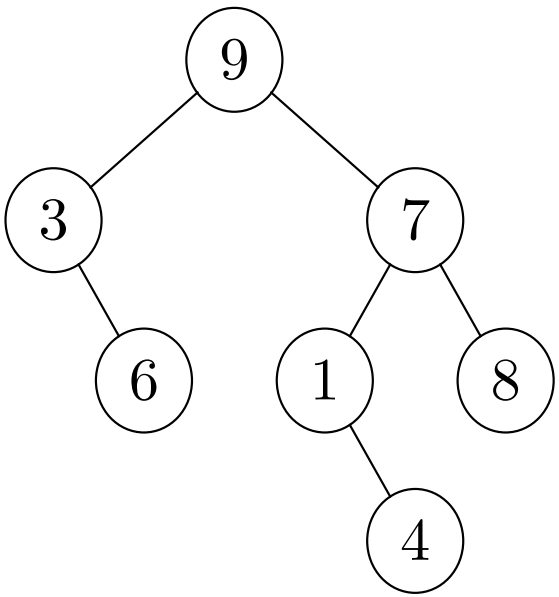
Binary trees/1



- Each node may have a left or a right child
  - 3 and 1 have no **left child**
  - 6 is the **right child** of 3

TREES AND HEAPS/1

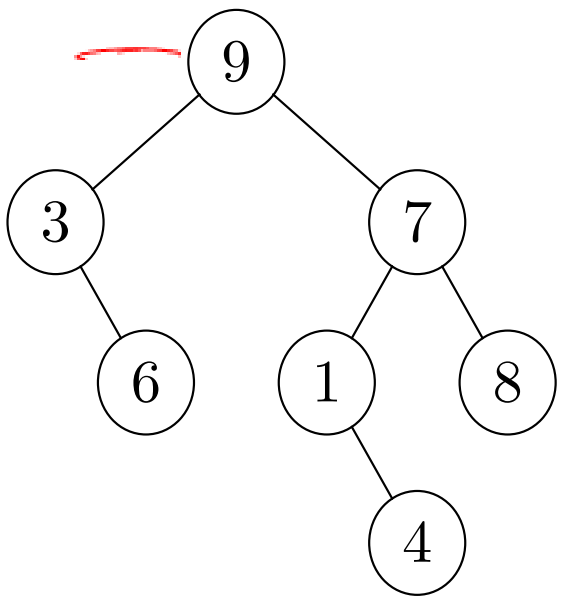
Binary trees/1



- Each node may have a left or a right child
  - 3 and 1 have no **left child**
  - 6 is the **right child** of 3
- Each node has at most one parent
  - The **parent** of 7 is 9

Trees and heaps/1

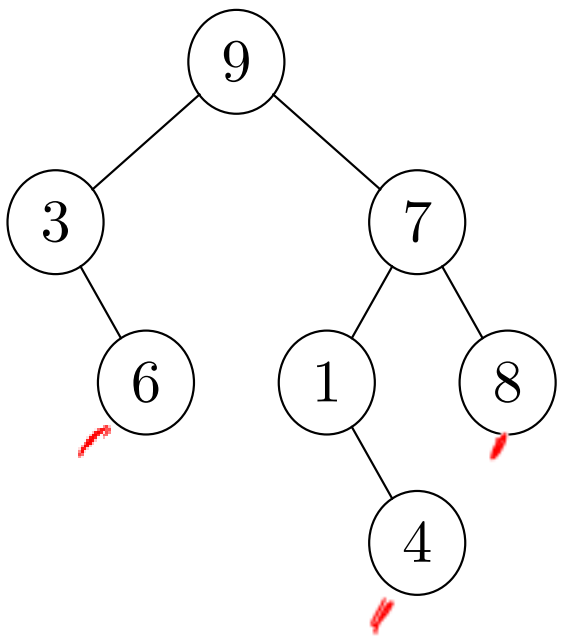
Binary trees/1



- Each node may have a left or a right child
  - 3 and 1 have no **left child**
  - 6 is the **right child** of 3
- Each node has at most one parent
  - The **parent** of 7 is 9
- The root has no parent
  - 9 is the **root** of the tree

Trees and heaps/1

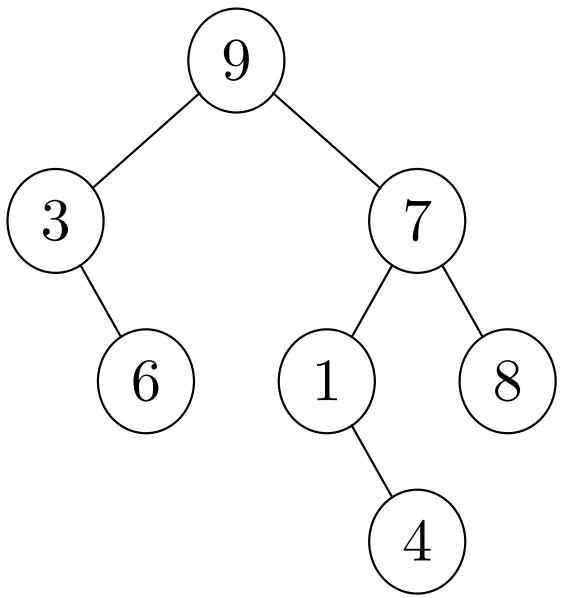
Binary trees/1



- Each node may have a left or a right child
  - 3 and 1 have no **left child**
  - 6 is the **right child** of 3
- Each node has at most one parent
  - The **parent** of 7 is 9
- The root has no parent
  - 9 is the **root** of the tree
- A leaf has no children
  - 6, 4, and 8 are **leaves**

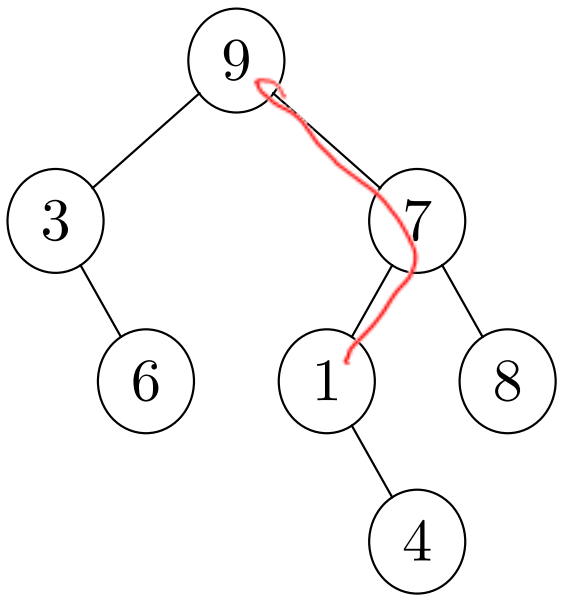
TREES AND HEAPS/2

Binary trees/2



TREES AND HEAPS/2

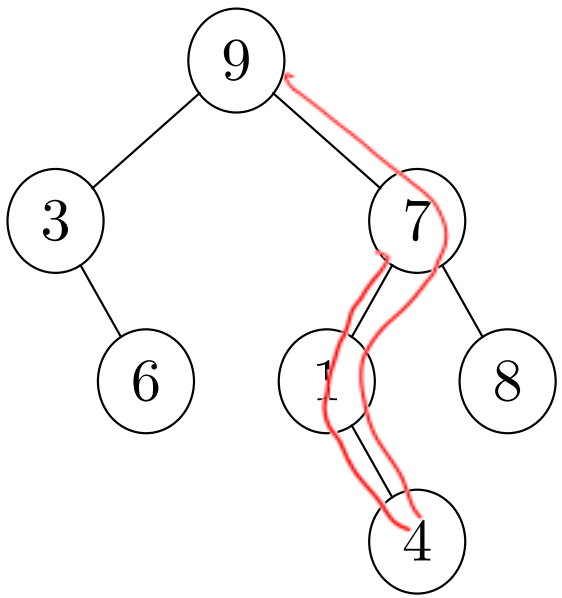
Binary trees/2



- The depth (level) of a node x is the length of the path from the root to x
  - The **depth** (**level**) of 1 is 2
  - The depth of 9 is 0

TREES AND HEAPS/2

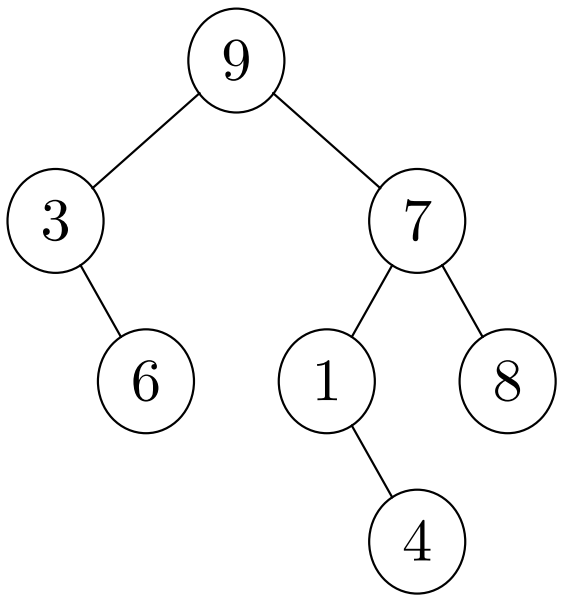
Binary trees/2



- The depth (level) of a node x is the length of the path from the root to x
  - The **depth** (**level**) of 1 is 2
  - The depth of 9 is 0
- The height of a node x is the length of the longest path from x to a leaf
  - The **height** of 7 is 2
  - The height of 9 is 3

TREES AND HEAPS/2

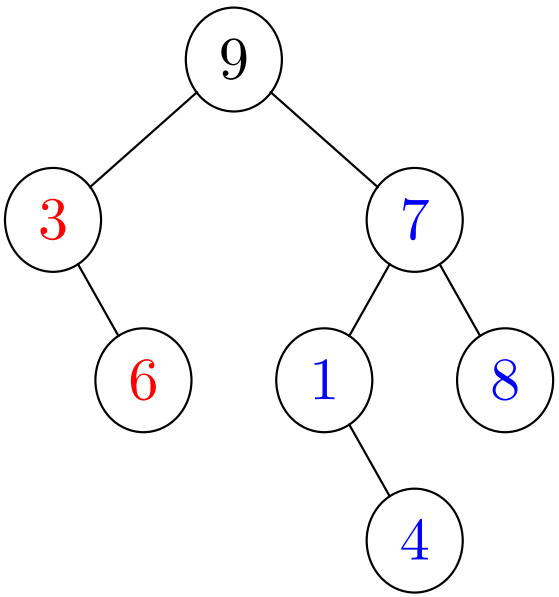
Binary trees/2



- The depth (level) of a node x is the length of the path from the root to x
  - The **depth** (**level**) of 1 is 2
  - The depth of 9 is 0
- The height of a node x is the length of the longest path from x to a leaf
  - The **height** of 7 is 2
  - The height of 9 is 3
- The height of a tree is the height of its root

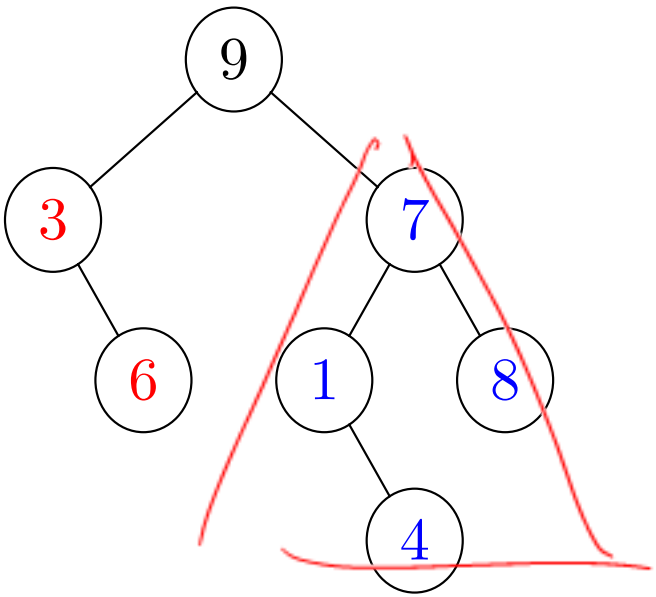
Trees and heaps/3

Binary trees/3



TREES AND HEAPS/3

Binary trees/3

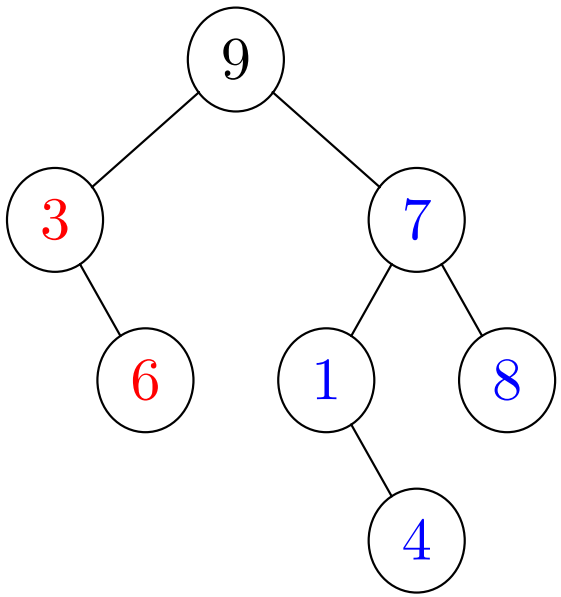


- The right subtree of a node x is the tree rooted at the right child of x
  - The **right subtree** of 9 includes all blue nodes and connecting edges
  - The root of the right subtree of 9 is 7

Trees and heaps/3

no ordering of nodes

Binary trees/3



- The right subtree of a node x is the tree rooted at the right child of x
  - The **right subtree** of 9 includes all blue nodes and connecting edges
  - The root of the right subtree of 9 is 7
- The left subtree of a node x is the tree rooted at the left child of x
  - The **left subtree** of 9 includes all red nodes and connecting edges
  - The root of the left subtree of 9 is 3
  - There is no left subtree of 3

TREES AND HEAPS/4

Complete trees

- A **complete binary tree** is a binary tree where
  - all the leaves have the same depth
  - all internal nodes have two children

TREES AND HEAPS/4

Complete trees

- A **complete binary tree** is a binary tree where
  - all the leaves have the same depth
  - all internal nodes have two children
- A **nearly complete binary tree** is a binary tree where
  - all levels of non-maximal depth d are full (have 2^d nodes)
  - all the leaves with maximal depth are as far left as possible

TREES AND HEAPS/4

Complete trees

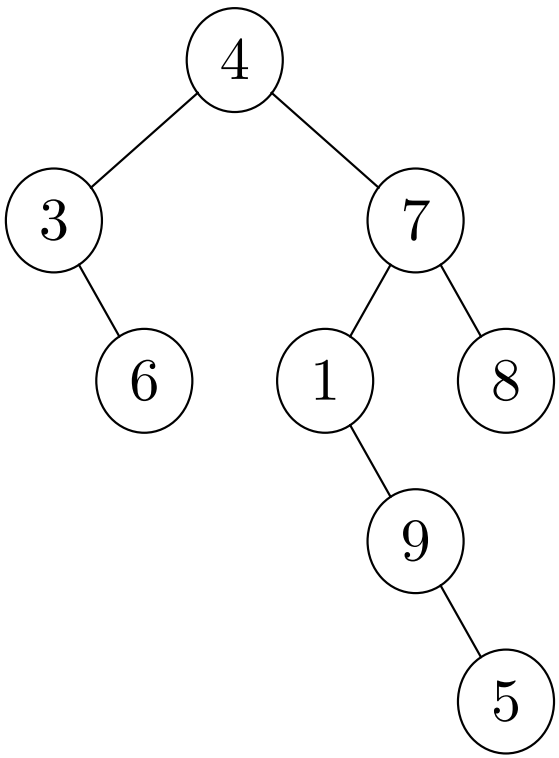
- A **complete binary tree** is a binary tree where
  - all the leaves have the same depth
  - all internal nodes have two children
- A **nearly complete binary tree** is a binary tree where
  - all levels of non-maximal depth d are full (have 2^d nodes)
  - all the leaves with maximal depth are as far left as possible

Definition of heap

- A binary tree is a (binary) **heap** if and only if
  - it is a nearly complete binary tree and, furthermore,
  - each node is greater than or equal to all its children

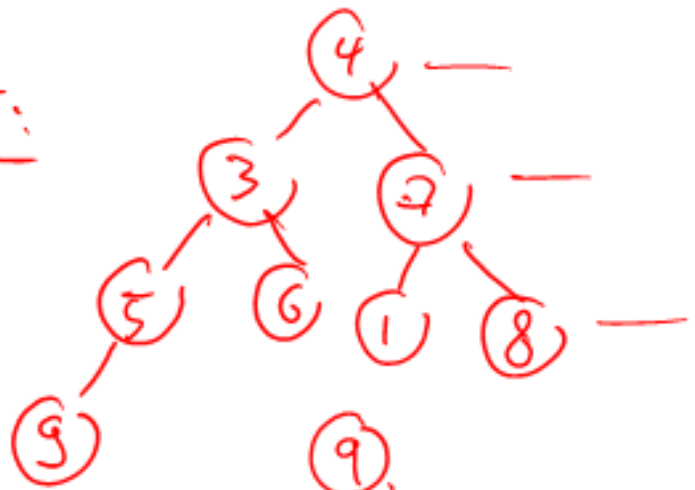
EXERCISE SL04-1

Construct a complete binary tree, a nearly complete binary tree, and a heap with the same nodes as T.

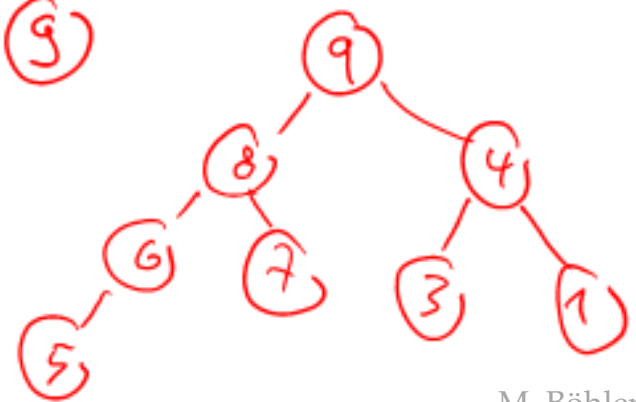


CBT: not possible
requires $2^x - 1$ nodes

NCBT:



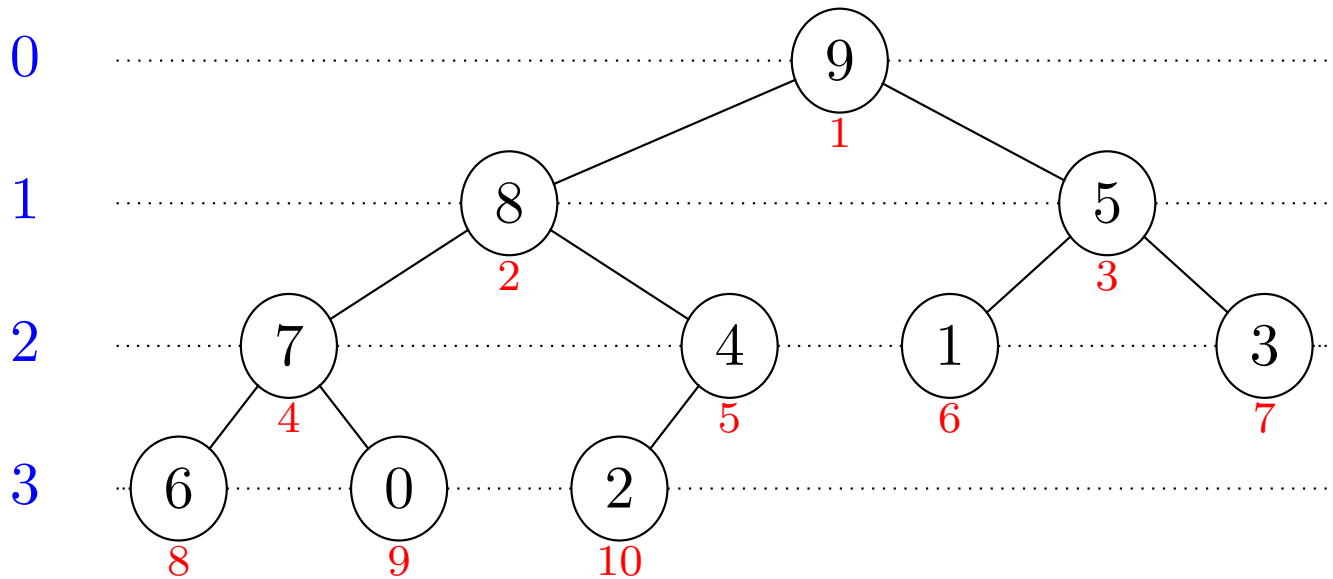
Heap:



TREES AND HEAPS/5

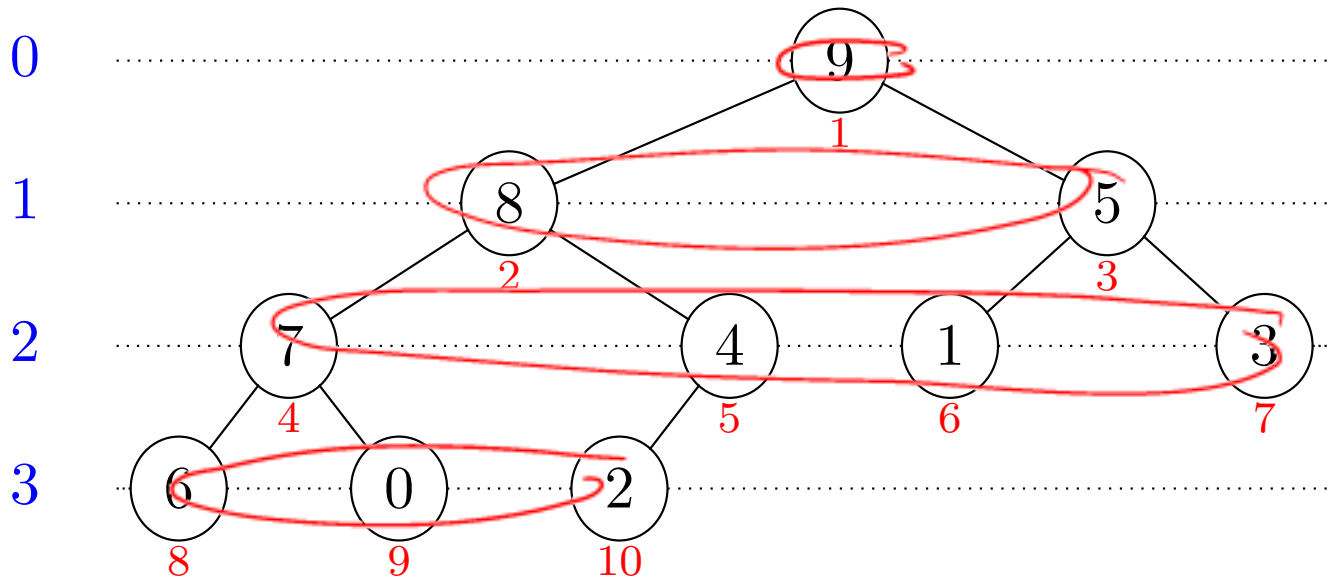
no holes / missing nodes ⇒ heap can be represented as an array

Heaps and arrays are closely related:



TREES AND HEAPS/5

Heaps and arrays are closely related:



node	9	8	5	7	4	1	3	6	0	2
ind/lev	1/0	2/1	3/1	4/2	5/2	6/2	7/2	8/3	9/3	10/3

TREES AND HEAPS/6

Fundamental properties

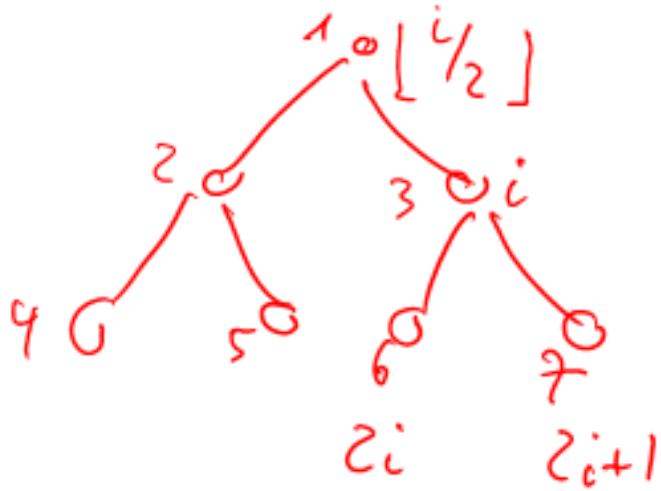
- Let i be a node index.
- Heap property: $A[parent(i)] \geq A[i]$

TREES AND HEAPS/6

Fundamental properties

- Let i be a node index.
- Heap property: $A[parent(i)] \geq A[i]$
- A binary heap can be efficiently stored as an array
  - Because it is a nearly complete binary tree

Trees and heaps/6



Fundamental properties

- Let i be a node index.
- Heap property: $A[parent(i)] \geq A[i]$
- A binary heap can be efficiently stored as an array
  - Because it is a nearly complete binary tree
- Finding parent, left child, and right child:

Algo: Parent(i)

return ⌊i/2⌋;

Algo: Left(i)

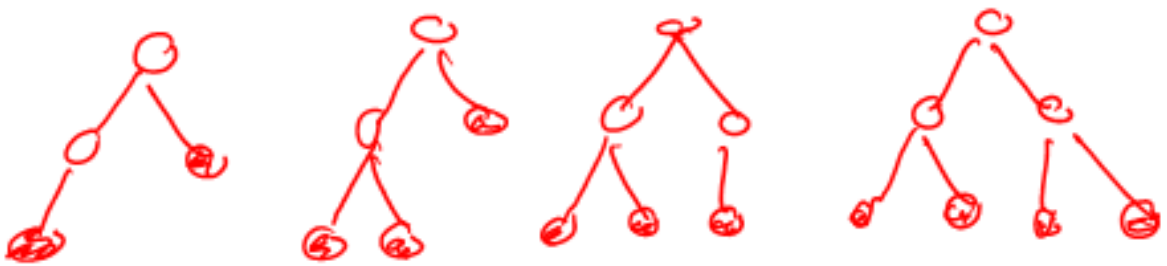
return 2*i;

Algo: Right(i)

return 2*i+1;

EXERCISE SL04-2

Assume a heap where all elements are distinct. What is the minimum and maximum number of elements in a heap of height h? Where in a MaxHeap might the smallest element be?



- min # of elements: 2^h $(2^2 = 4)$ $\sum_{i=0}^{h-1} 2^i + 1$
- max # of elements: $2^{h+1} - 1$
- smallest elem must be a leaf

TREES AND HEAPS/7

- A heap where the largest element is at the root is a max-heap.
  - Used for sorting arrays
  - Max-heap property: $A[parent(i)] \geq A[i]$
- A heap where the smallest element is at the root is a min-heap.
  - Used for priority queues
  - Min-heap property: $A[parent(i)] \leq A[i]$
- A binary heap can be exploited for sorting arrays
  - Because of the organization of the data
  - Basic operations: `heapify`, `build_heap`

HEAPIFY/1

- Input: index i in array A, number s of heap elements

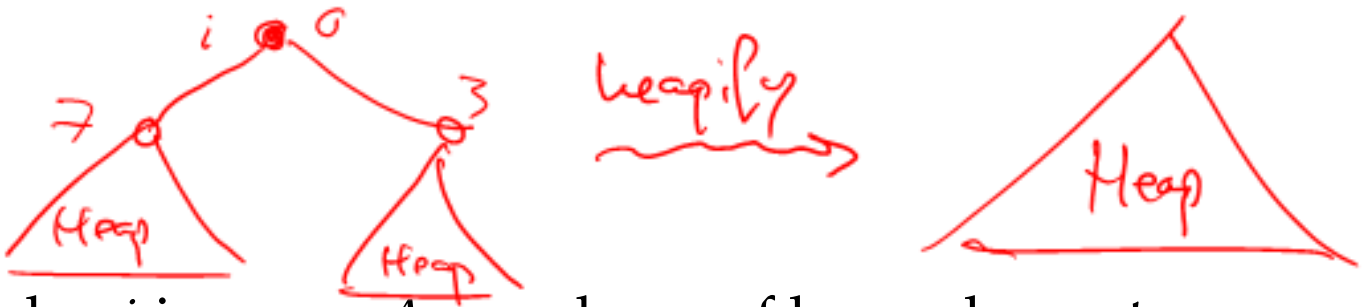
HEAPIFY/1

- Input: index i in array A, number s of heap elements
- Binary trees rooted at `left`(i) and `right`(i) are binary heaps

HEAPIFY/1

- Input: index i in array A, number s of heap elements
- Binary trees rooted at `left`(i) and `right`(i) are binary heaps
- $A[i]$ may be smaller than its children, thus violating the heap property

HEAPIFY/1



- Input: index i in array A, number s of heap elements
- Binary trees rooted at `left`(i) and `right`(i) are binary heaps
- $A[i]$ may be smaller than its children, thus violating the heap property

- `heapify`(A, i, s) transforms the binary tree rooted at i into a binary heap

HEAPIFY/1

- Input: index i in array A, number s of heap elements
- Binary trees rooted at `left`(i) and `right`(i) are binary heaps
- $A[i]$ may be smaller than its children, thus violating the heap property

- `heapify`(A, i, s) transforms the binary tree rooted at i into a binary heap
- Strategy: move $A[i]$ down the heap until the heap property is satisfied again

HEAPIFY/2

The algorithm

size of the heap

Algo: Heapify(A,i,s)

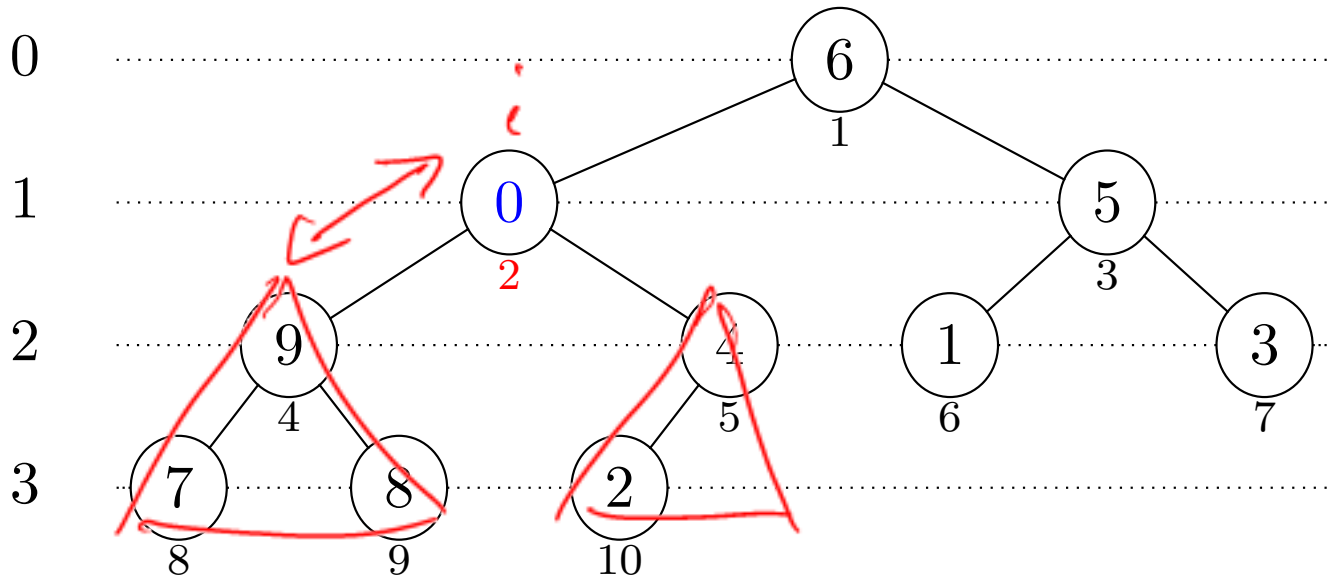
```
m = i;
l = Left(i);
r = Right(i);
if l≤s ∧ A[l]>A[m] then m = l;
if r≤s ∧ A[r]>A[m] then m = r;
if i≠m then
    exchange A[i] and A[m];
    Heapify(A,m,s);
```

left child exists

left elem is large

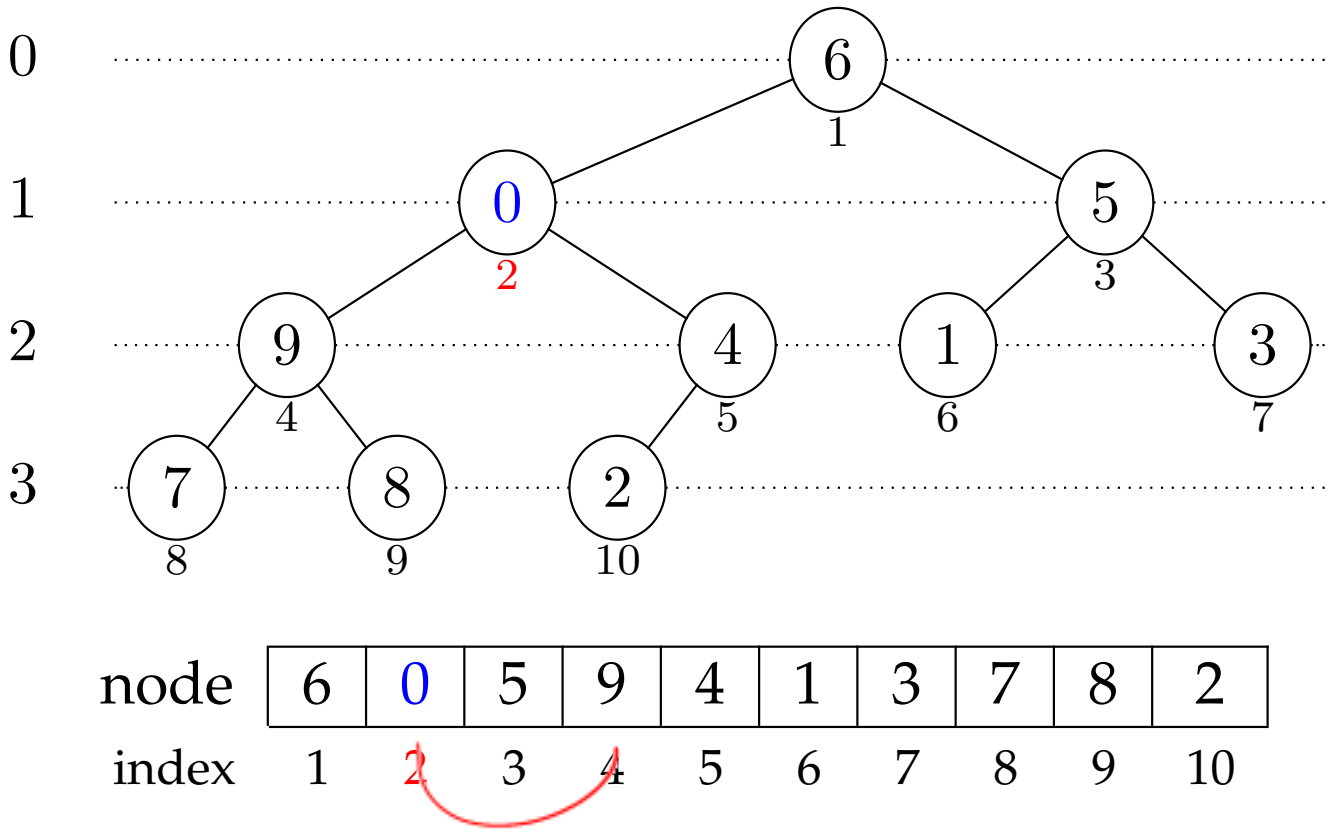
HEAPIFY/3

An example/1



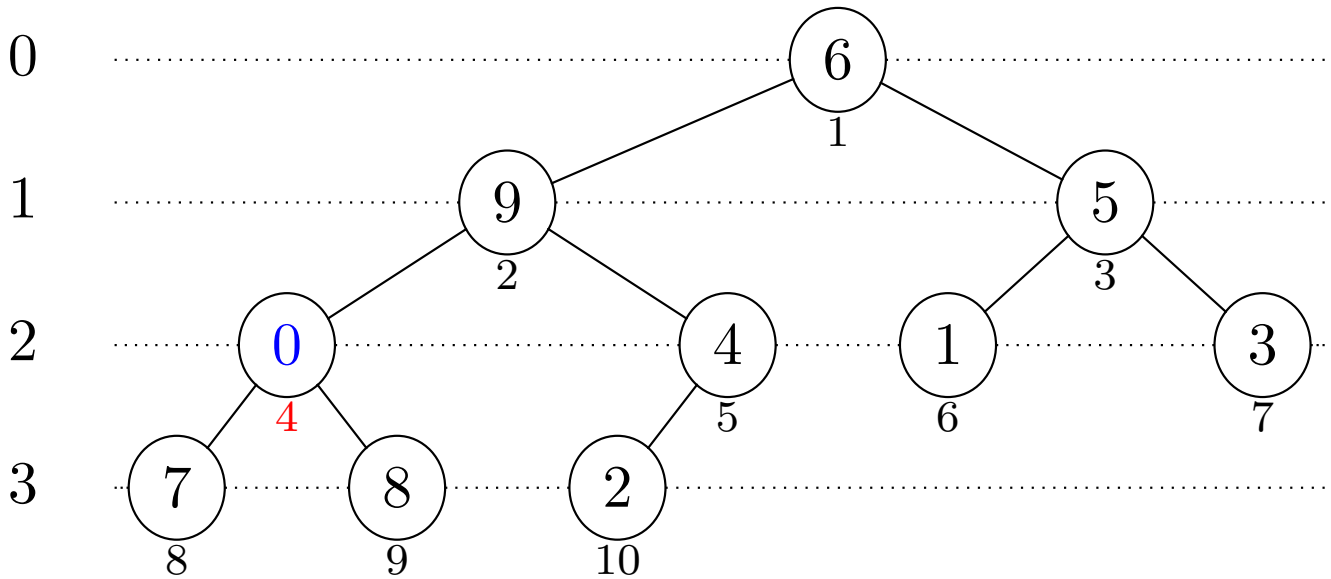
HEAPIFY/3

An example/1



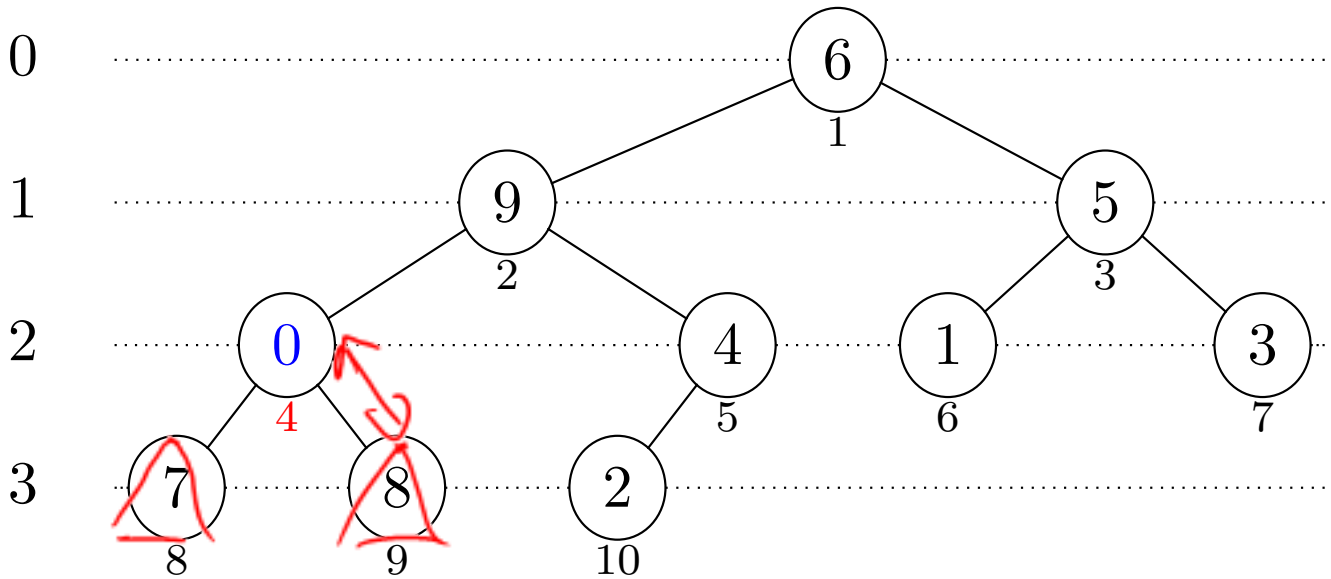
HEAPIFY/4

An example/2



HEAPIFY/4

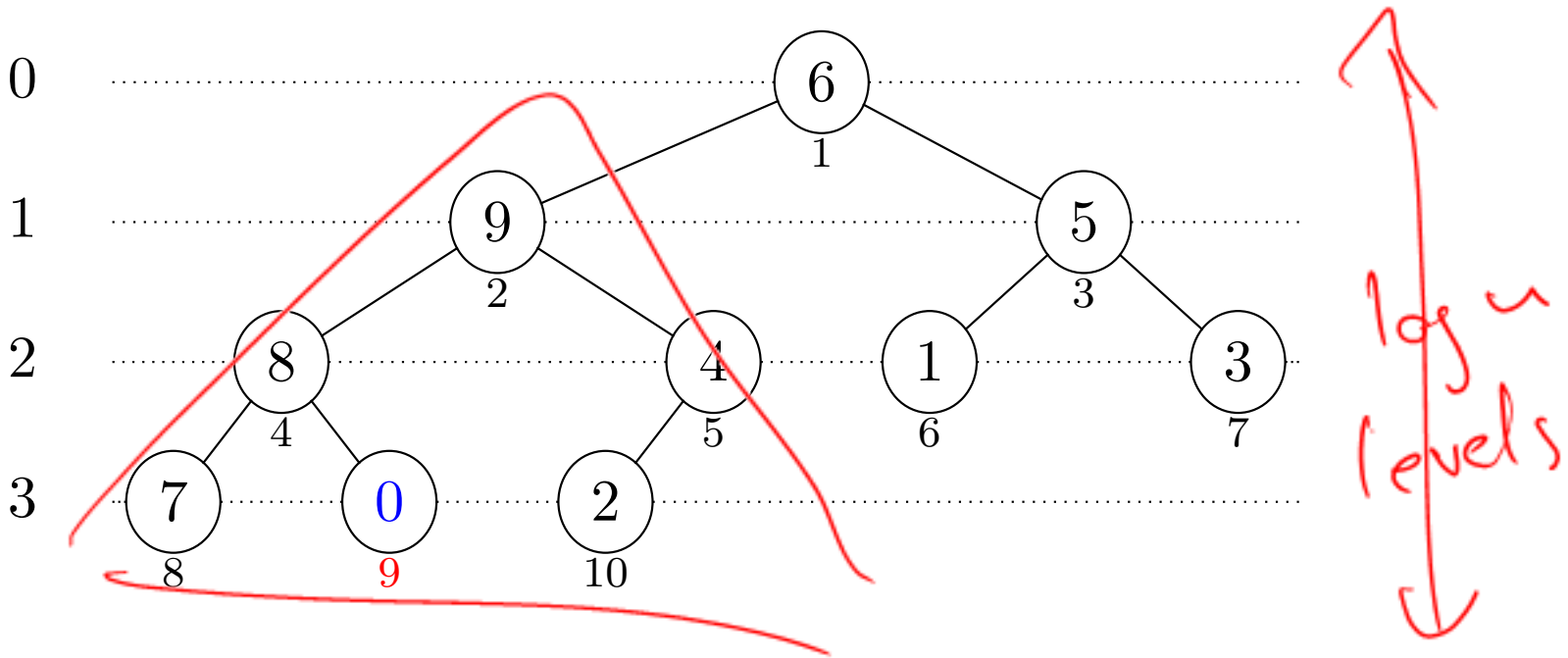
An example/2



node	6	9	5	0	4	1	3	7	8	2
index	1	2	3	4	5	6	7	8	9	10

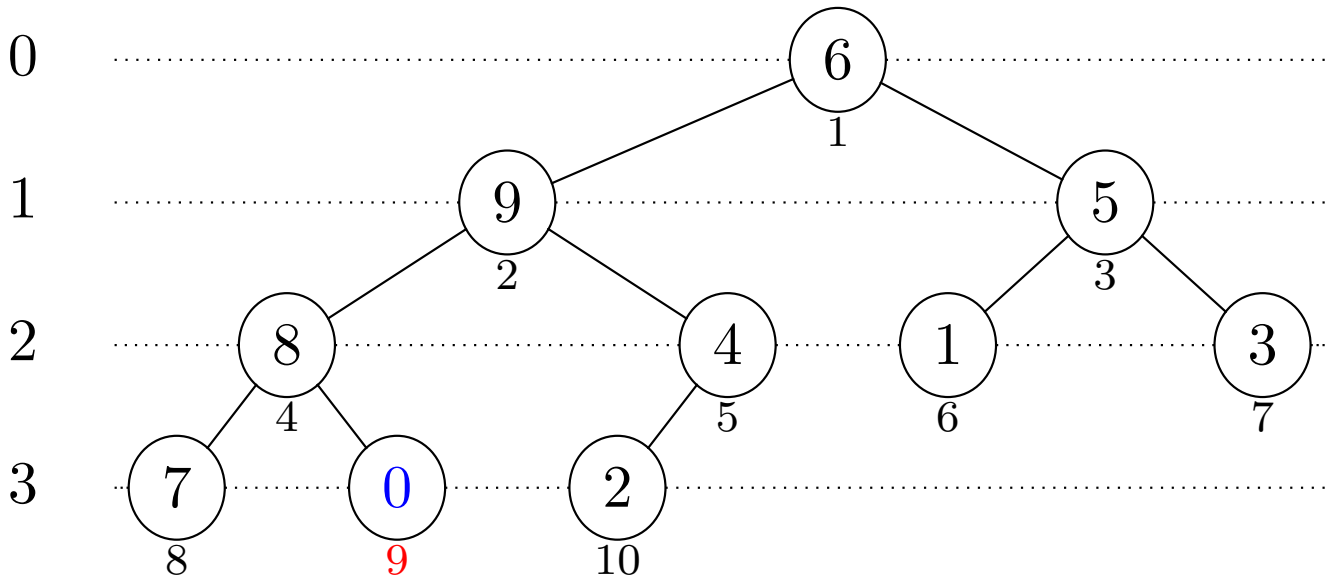
HEAPIFY/5

An example/3



HEAPIFY/5

An example/3



node	6	9	5	8	4	1	3	7	0	2
index	1	2	3	4	5	6	7	8	9	10

HEAPIFY/6

Running time

- The running time of heapify on a subtree of size n rooted at i includes time to
  - determine relationship between elements: $\Theta(1)$
  - run heapify on a subtree rooted at one of i's children

HEAPIFY/6

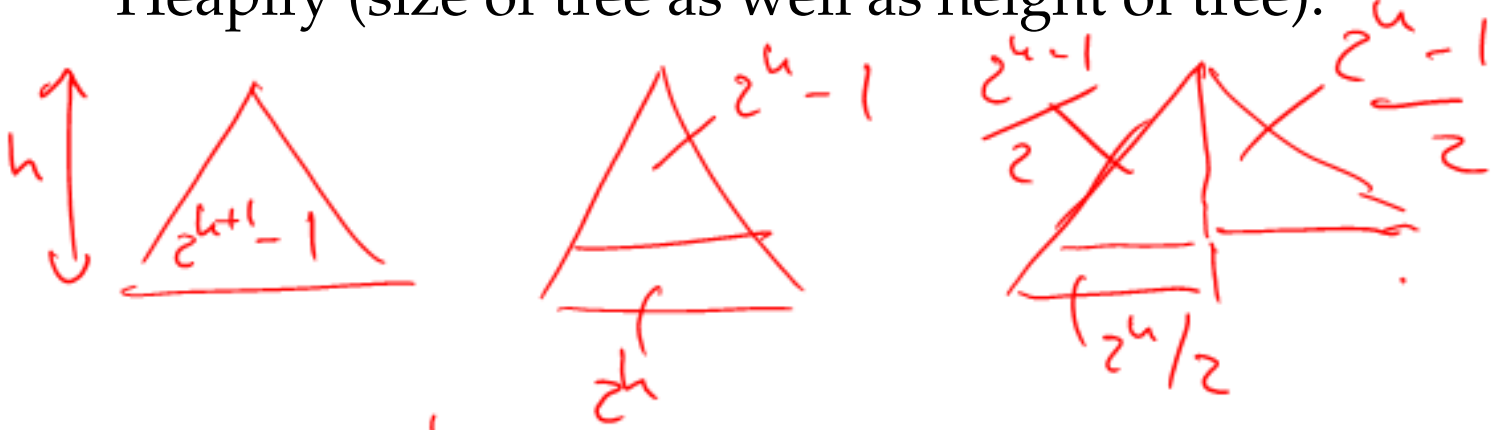
Running time

- The running time of heapify on a subtree of size n rooted at i includes time to
  - determine relationship between elements: $\Theta(1)$
  - run heapify on a subtree rooted at one of i's children
- $2n/3$ is the worst case size of the subtree (half filled bottom level) and thus

$$T(n) \leq T(2n/3) + \Theta(1), \text{ i.e. } T(n) = O(\lg n)$$

EXERCISE SL04-3

Explain the derivation of the worst case running time of Heapify (size of tree as well as height of tree).



$$2^h - 1 + 2^h/2 = n$$

$$2^h = 2/3\, n + 2/3$$

$$h = \log(2/3\, n + 2/3) \approx \log(2/3\, n) = \log(2/3) + \log(n)$$

$$\approx \log(n)$$

BUILDING A HEAP/1

Build a heap

- Convert an array A with n elements into a heap
- Note that the elements in $A[\lfloor n/2 \rfloor + 1 \ldots n]$ are 1-element heaps

BUILDING A HEAP/1

Build a heap

- Convert an array A with n elements into a heap
- Note that the elements in $A[\lfloor n/2 \rfloor + 1 \ldots n]$ are 1-element heaps

middle

root

The algorithm

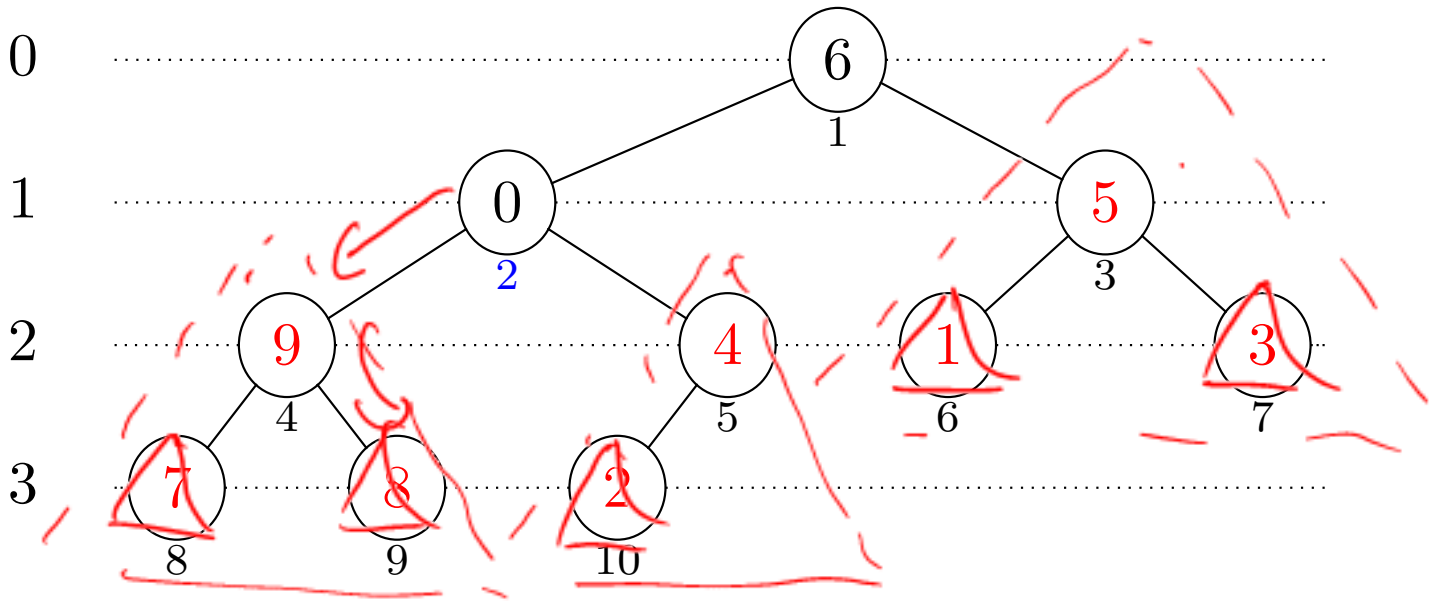
Algo: BuildHeap(A,n)

---

for $i = \lfloor n/2 \rfloor$ **to** *1* **do** Heapify(A,i,n);

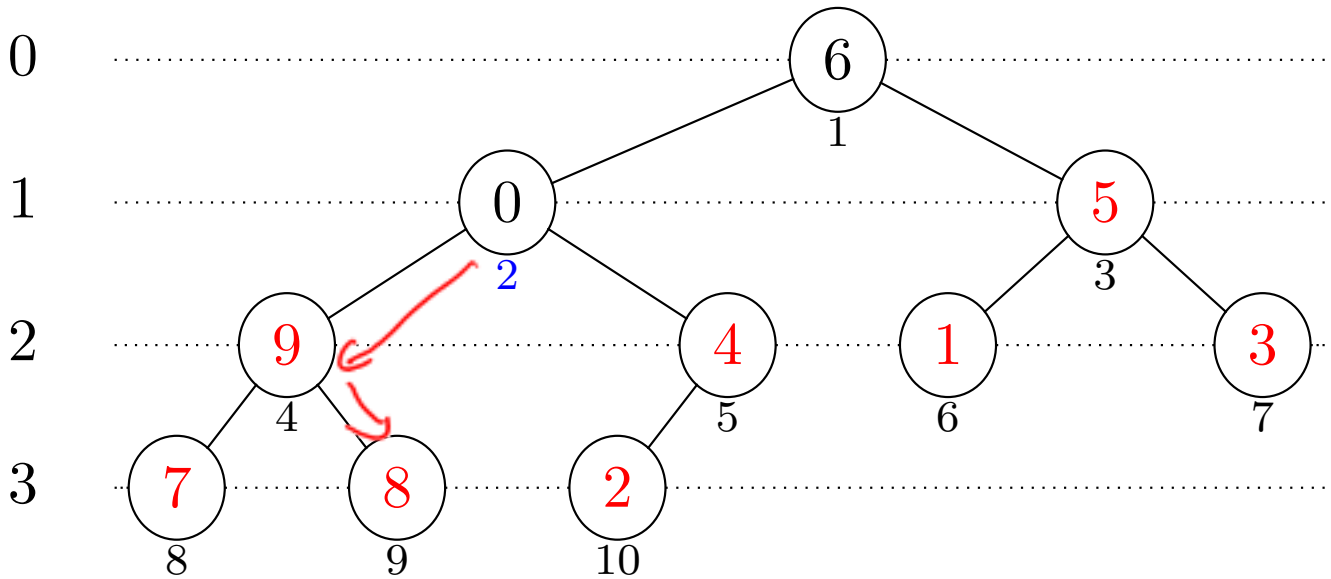
Building a heap/8

An example/1



BUILDING A HEAP/8

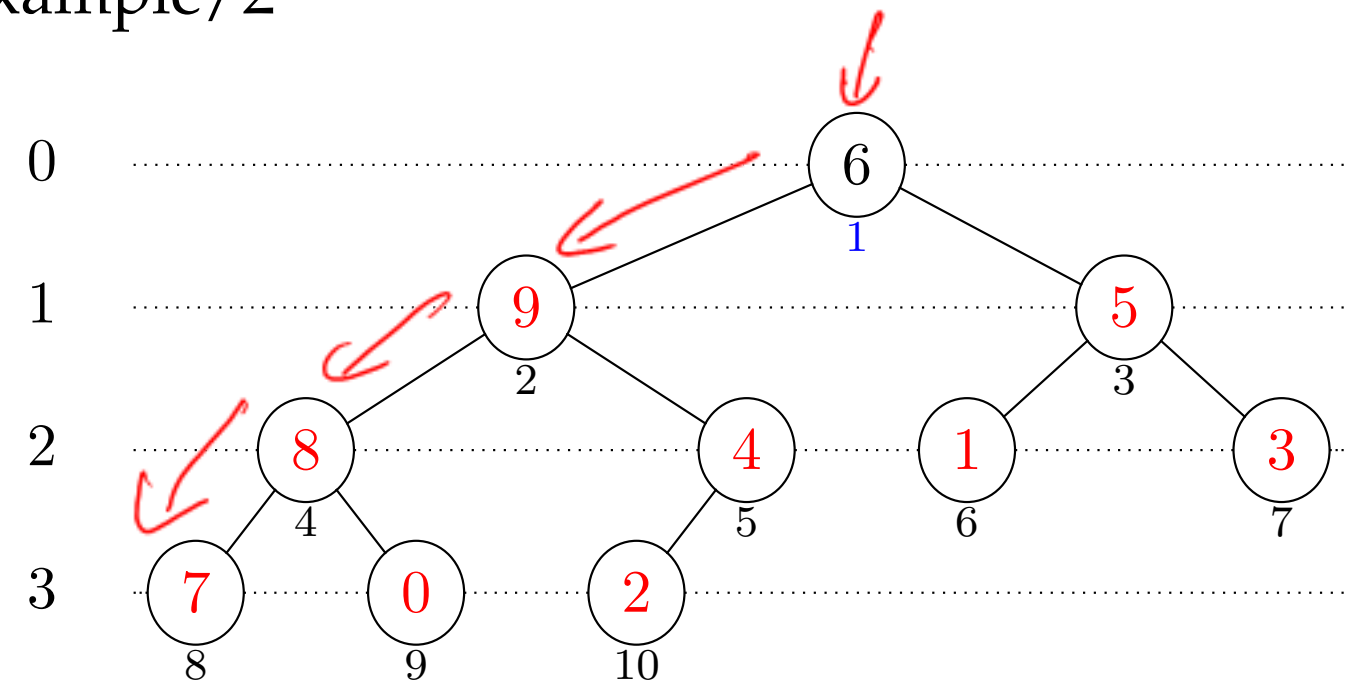
An example/1



node	6	0	5	9	4	1	3	7	8	2
index	1	2	3	4	5	6	7	8	9	10

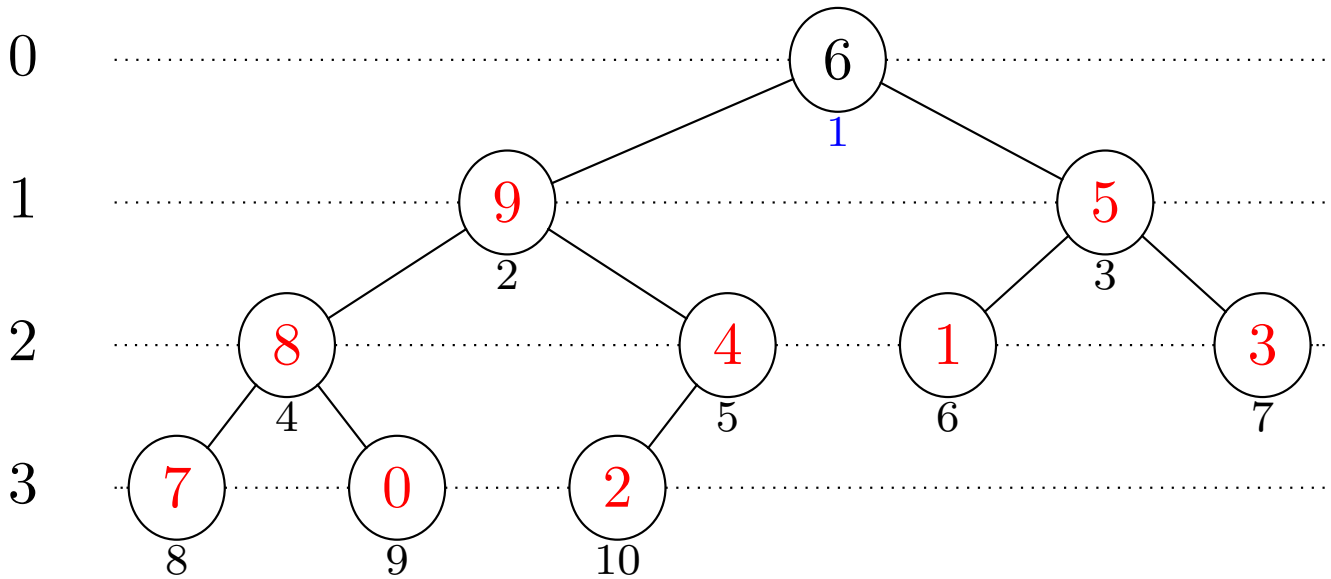
BUILDING A HEAP/3

An example/2



BUILDING A HEAP/3

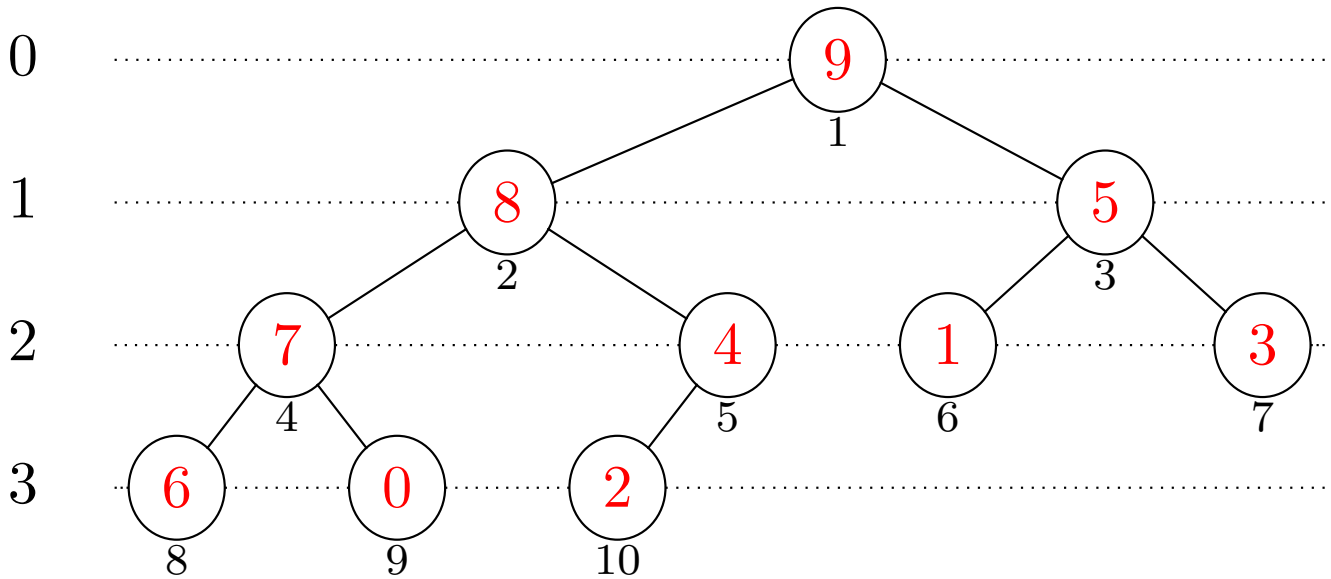
An example/2



node	6	9	5	8	4	1	3	7	0	2
index	1	2	3	4	5	6	7	8	9	10

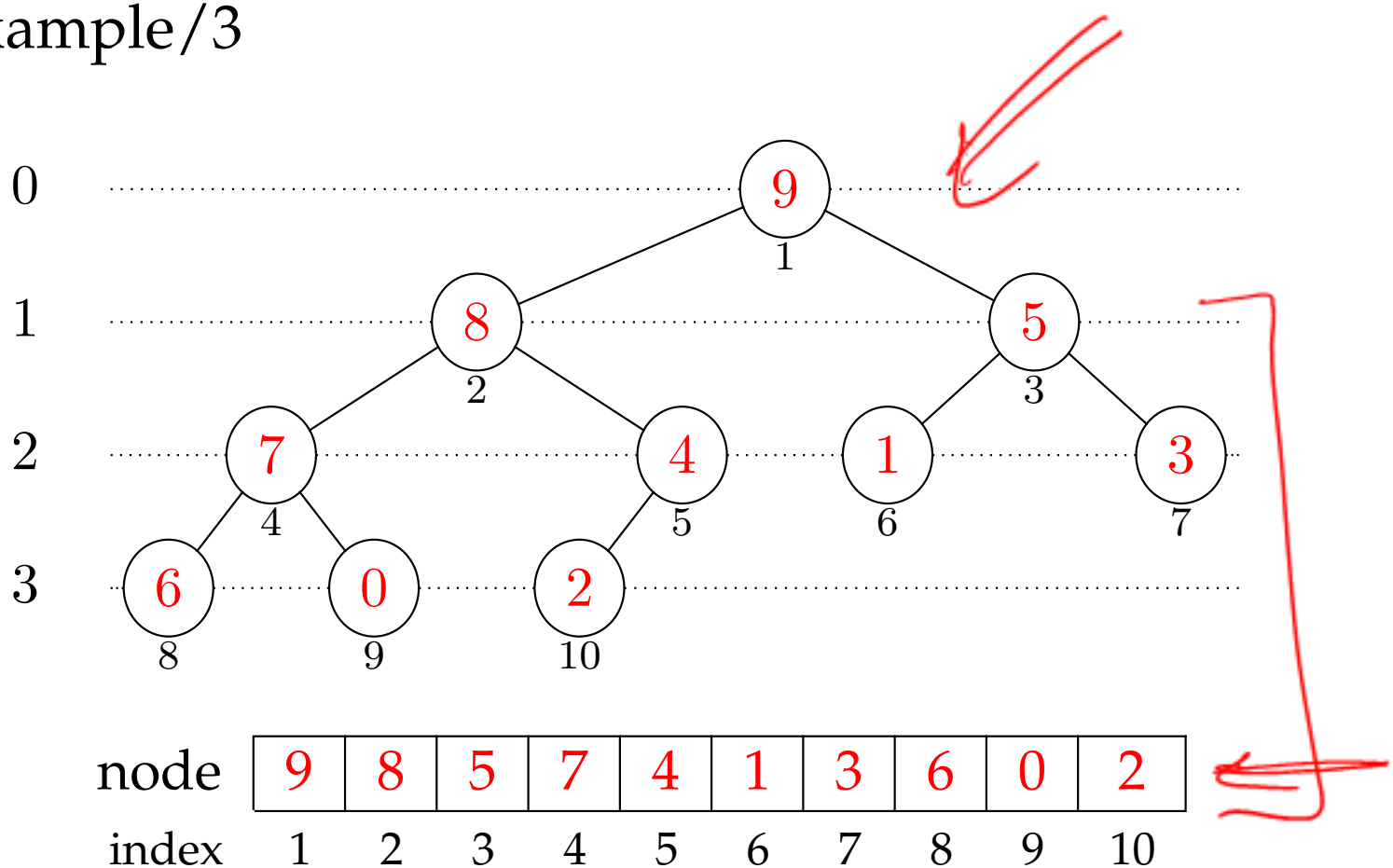
Building a heap/4

An example/3



BUILDING A HEAP/4

An example/3



BUILDING A HEAP/5

Correctness

- Loop invariant: all trees rooted at $m > i$ are heaps

BUILDING A HEAP/5

Correctness

- Loop invariant: all trees rooted at $m > i$ are heaps

Running time/1

- There are $O(n)$ calls to heapify and so $T(n) = O(n \lg n)$

BUILDING A HEAP/5

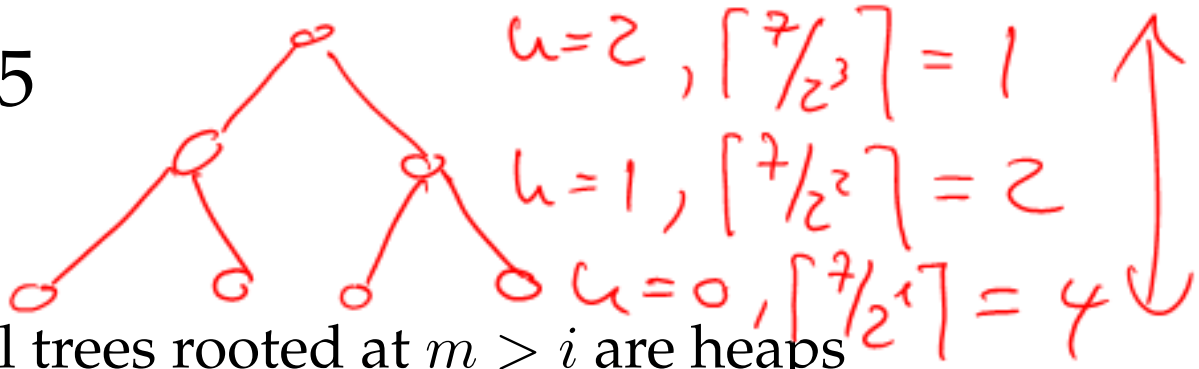
Correctness

- Loop invariant: all trees rooted at $m > i$ are heaps

Running time/1

- There are $O(n)$ calls to heapify and so $T(n) = O(n \lg n)$
- Not an asymptotically tight bound — but good enough for an overall $O(n \lg n)$ bound for heap sort

BUILDING A HEAP/5



Correctness

- Loop invariant: all trees rooted at $m > i$ are heaps

Running time/1

- There are $O(n)$ calls to heapify and so $T(n) = O(n \lg n)$
- Not an asymptotically tight bound — but good enough for an overall $O(n \lg n)$ bound for heap sort
- Intuition for tight bound: time for heapify to run at a node i varies with the height of i
  - An n-element binary heap has height $\lg n$
  - The heap has at most $\lceil n/2^{h+1} \rceil$ nodes of height h
  - The cost for one call of heapify is $O(h)$

BUILDING A HEAP/6

Running time/2

- Note: $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$
- Thus: $\sum_{k=0}^{\infty} k(1/x)^k = \frac{1/x}{(1-1/x)^2}$
- The asymptotically tight bound:

$$T(n) = \sum_{h=0}^{\lg n} \frac{n}{2^{h+1}} O(h)$$

BUILDING A HEAP/6

Running time/2

- Note: $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$
- Thus: $\sum_{k=0}^{\infty} k(1/x)^k = \frac{1/x}{(1-1/x)^2}$
- The asymptotically tight bound:

$$\begin{aligned} T(n) &= \sum_{h=0}^{\lg n} \frac{n}{2^{h+1}} O(h) \\ &= O\left(n \sum_{h=0}^{\lg n} \frac{h}{2^{h+1}}\right) \end{aligned}$$

BUILDING A HEAP/6

Running time/2

- Note: $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$
- Thus: $\sum_{k=0}^{\infty} k(1/x)^k = \frac{1/x}{(1-1/x)^2}$
- The asymptotically tight bound:

$$\begin{aligned} T(n) &= \sum_{h=0}^{\lg n} \frac{n}{2^{h+1}} O(h) \\ &= O\left(n \sum_{h=0}^{\lg n} \frac{h}{2^{h+1}}\right) \\ &\le O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O\left(n \sum_{h=0}^{\infty} h\left(\frac{1}{2}\right)^h\right) \end{aligned}$$

BUILDING A HEAP/6

Running time/2

- Note: $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$
- Thus: $\sum_{k=0}^{\infty} k(1/x)^k = \frac{1/x}{(1-1/x)^2}$
- The asymptotically tight bound:

$$\begin{aligned} T(n) &= \textstyle\sum_{h=0}^{\lg n} \frac{n}{2^{h+1}} O(h) \\ &= O\Big(n \textstyle\sum_{h=0}^{\lg n} \frac{h}{2^{h+1}}\Big) \\ &\leq O\Big(n \textstyle\sum_{h=0}^{\infty} \frac{h}{2^h}\Big) \\ &= O\Big(n \textstyle\sum_{h=0}^{\infty} h(\frac{1}{2})^h\Big) \\ &= O(n * 2) = O(n) \end{aligned}$$

HEAP SORT ALGORITHM/1

The algorithm

Algo: HeapSort(A,n)

```
s = n;
BuildHeap(A);
for i = n to 2 do
    exchange A[i] and A[1];
    s = s-1;
    Heapify(A,1,s);
```

position where largest remaining element is moved

last element

largest element

shrinks the heap / removes last element

HEAP SORT ALGORITHM/1

The algorithm

Algo: HeapSort(A,n)

```
s = n;
BuildHeap(A);
for i = n to 2 do
    exchange A[i] and A[1];
    s = s-1;
    Heapify(A,1,s);
```

Running time

- Heap sort runs in time $O(n) + nO(\lg n) = O(n \lg n)$

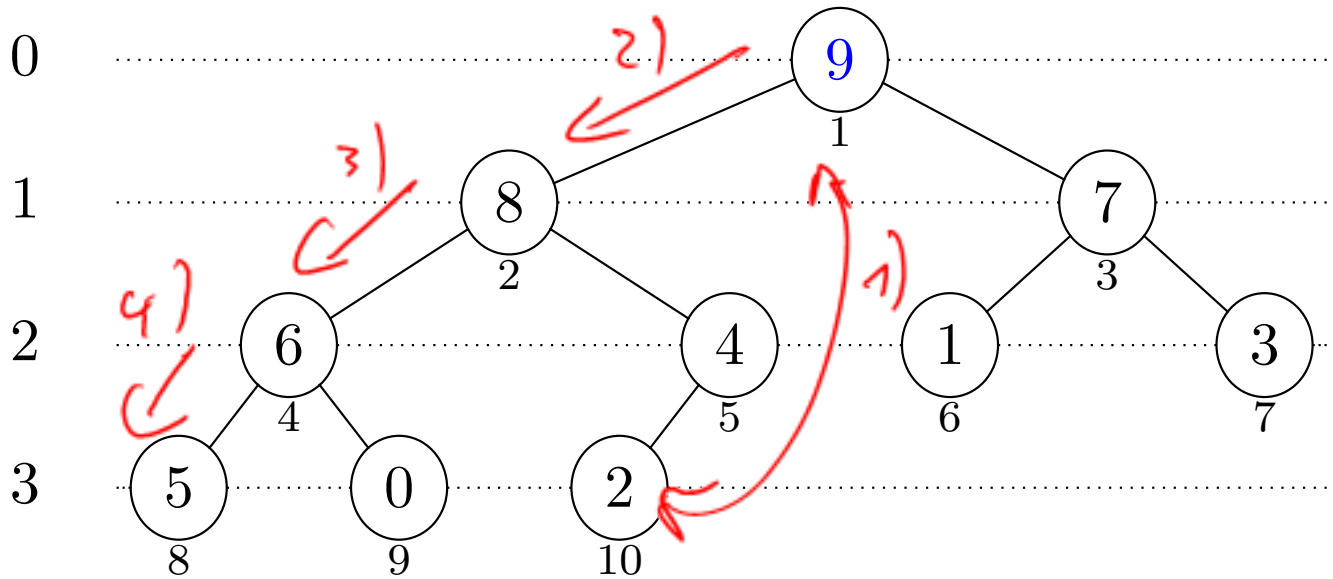
building the heap

n times

heapify

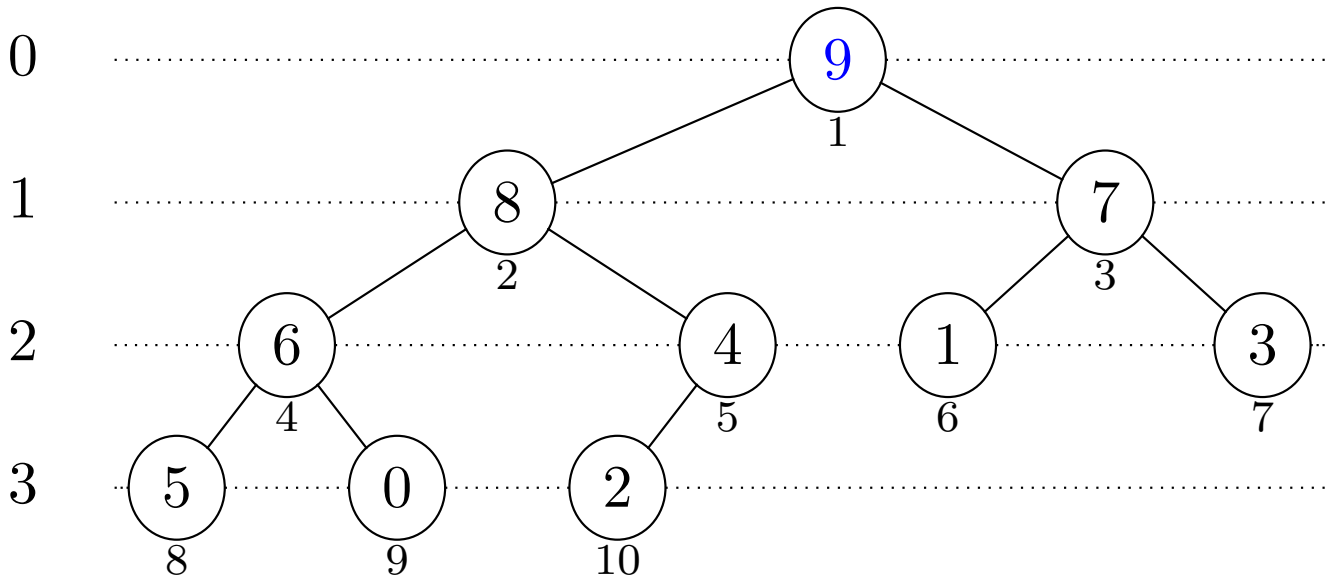
Heap sort algorithm/2

An example/1



Heap sort algorithm/2

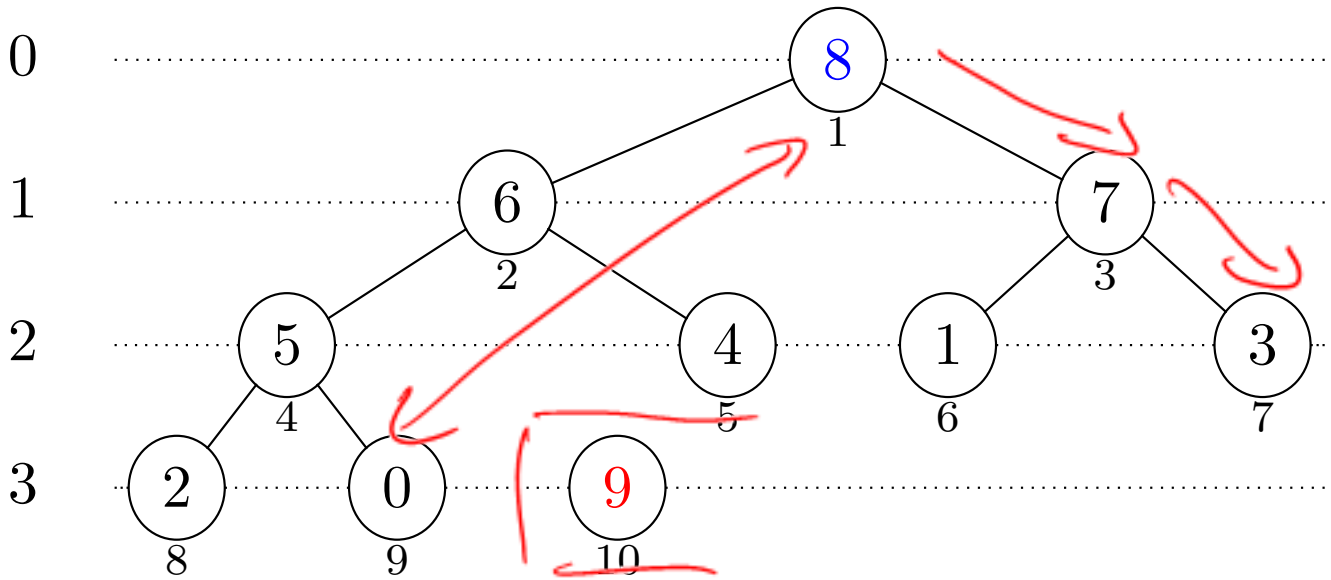
An example/1



node	9	8	7	6	4	1	3	5	0	2
index	1	2	3	4	5	6	7	8	9	10

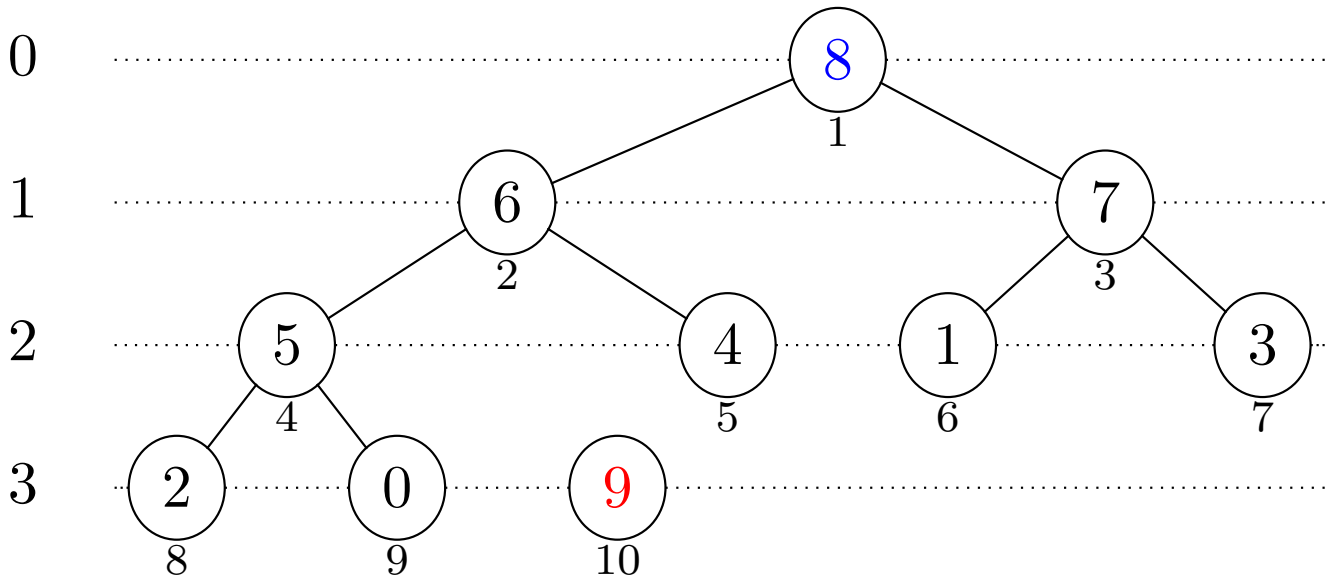
Heap sort algorithm/3

An example/2



Heap sort algorithm/3

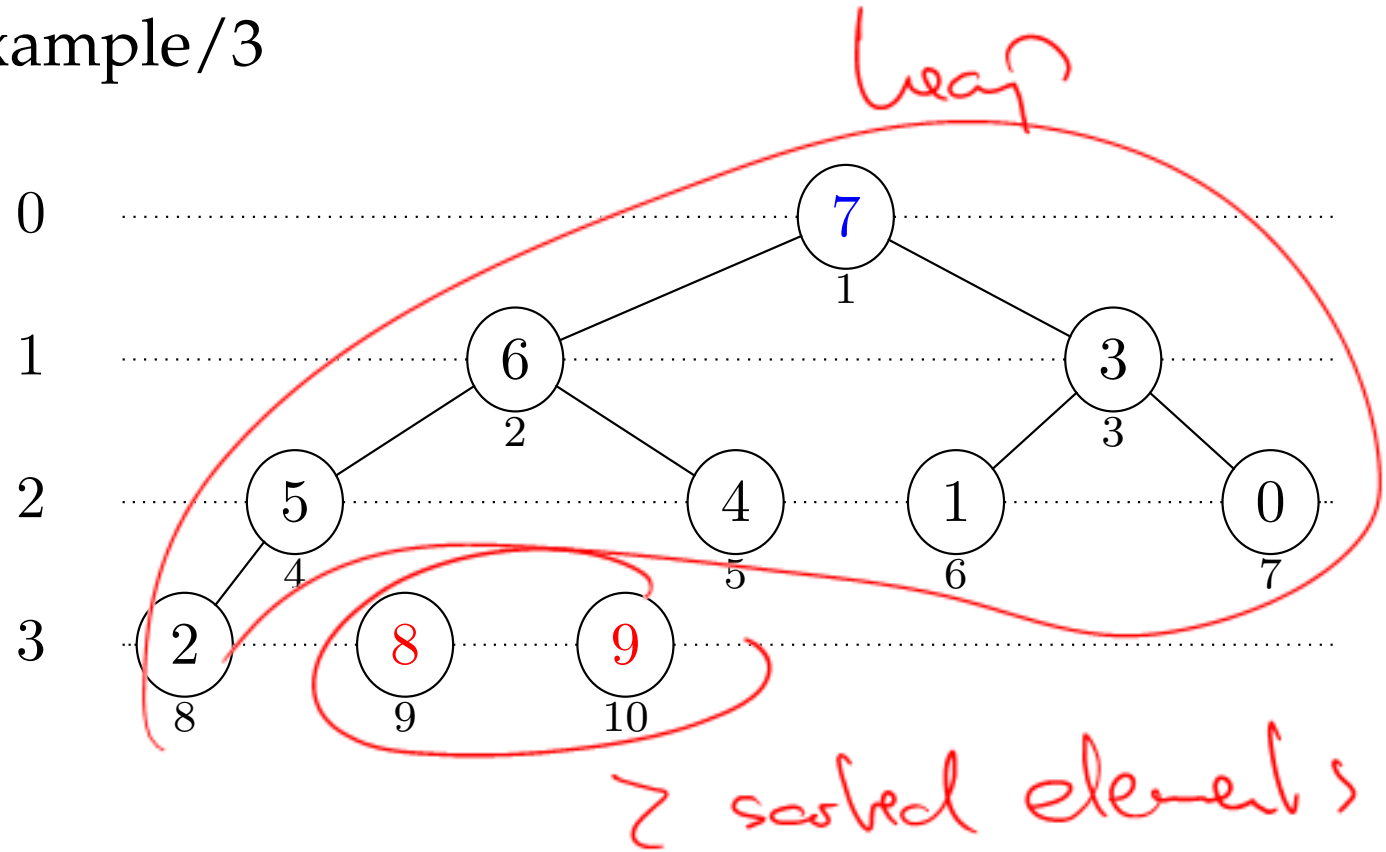
An example/2



node	8	6	7	5	4	1	3	2	0	9
index	1	2	3	4	5	6	7	8	9	10

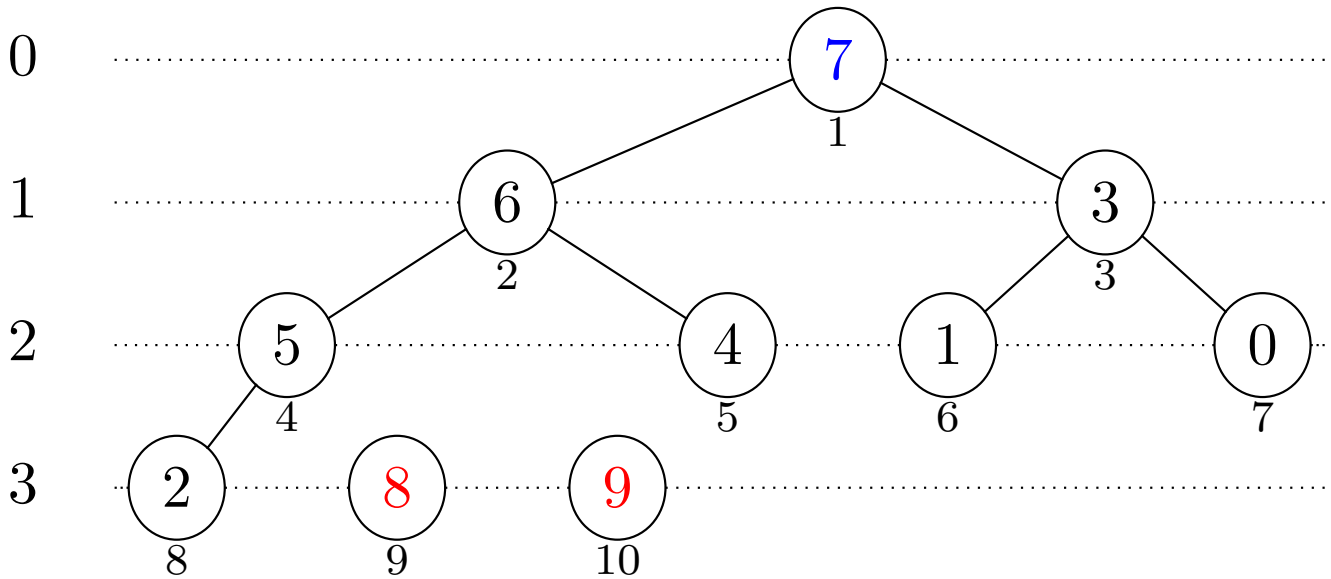
Heap sort algorithm/4

An example/3



Heap sort algorithm/4

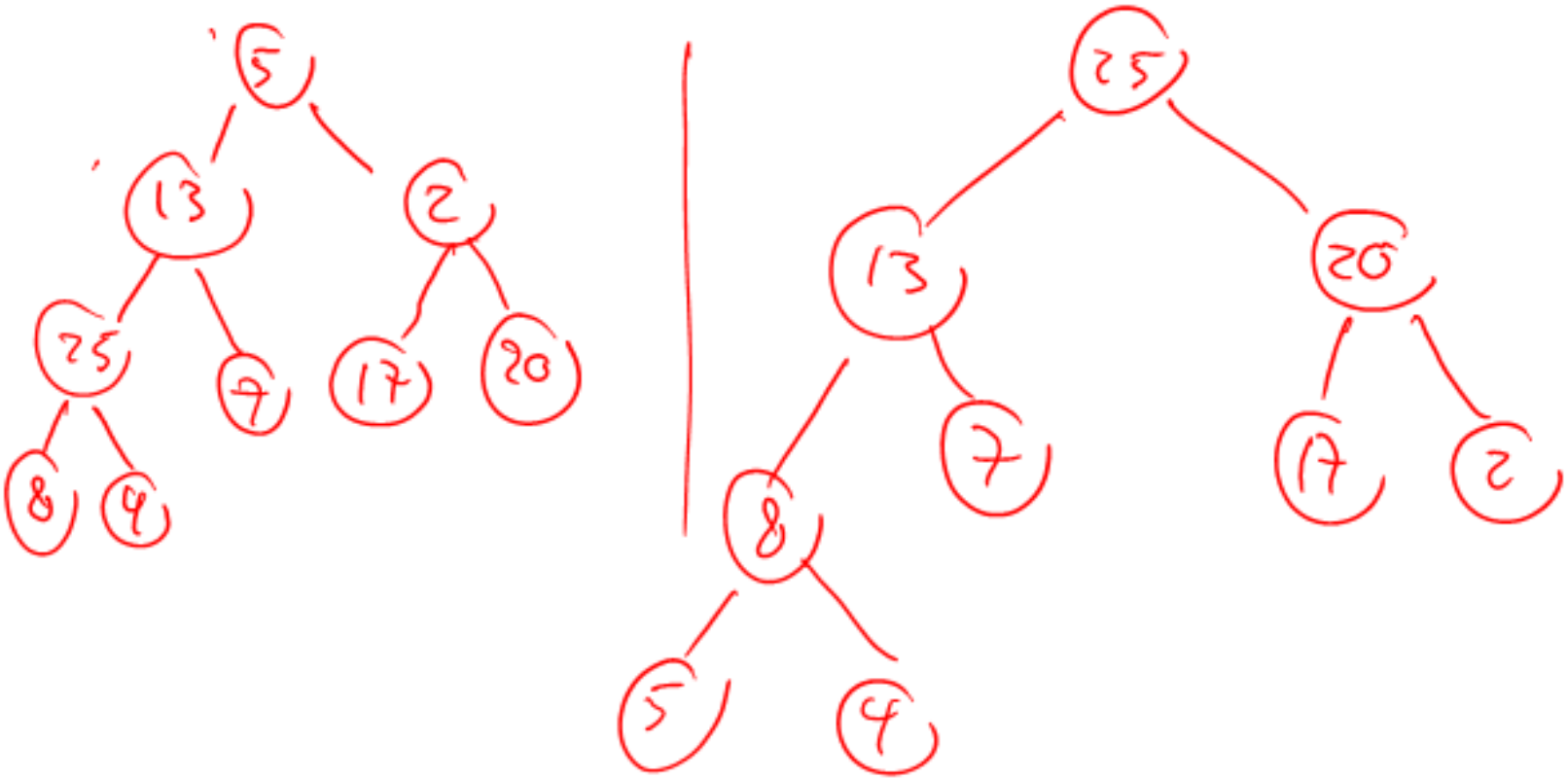
An example/3



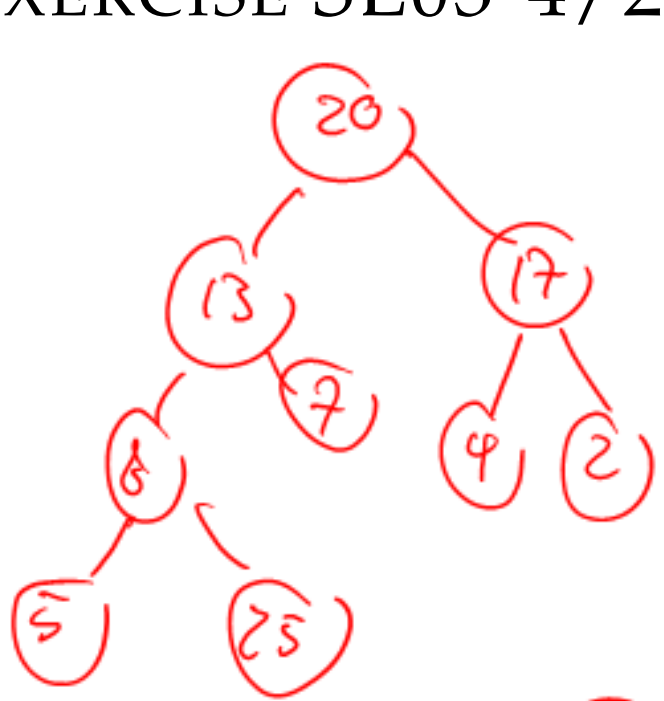
node	7	6	3	5	4	1	0	2	8	9
index	1	2	3	4	5	6	7	8	9	10

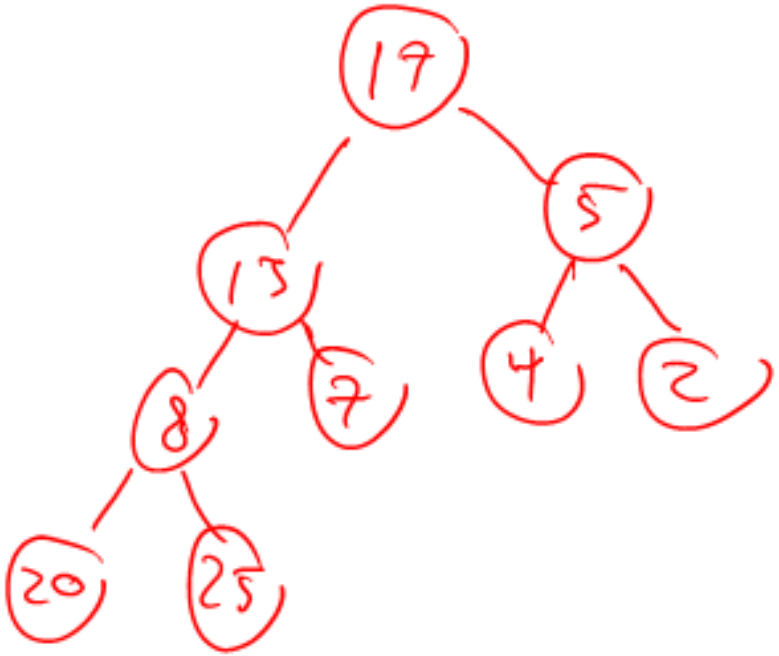
EXERCISE SL04-4/1

Run the heap sort algorithm on the array
$A = [5, 13, 2, 25, 7, 17, 20, 8, 4]$.
Show the state of the array after BuildHeap and after three iterations of the loop in the HeapSport algorithm.



EXERCISE SL03-4/2





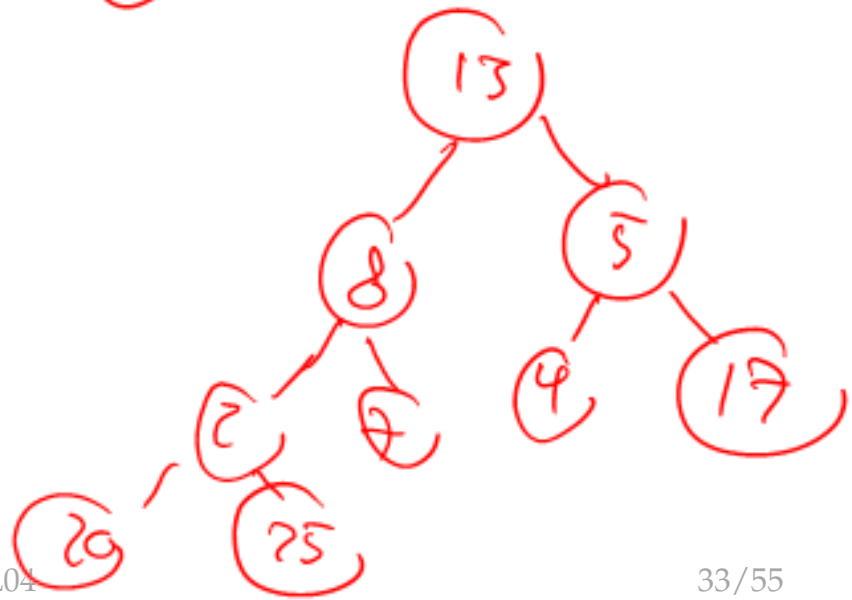


TABLE OF CONTENTS — SL04

1. Heap sort

2. Quick sort
   - Partitioning
   - Performance

Quick sort

Rough idea

- A divide-and-conquer algorithm
  - Divide: partition array into two subarrays such that the items in the lower part $\leq$ the items in the upper part
  - Conquer: recursively sort the two subarrays
  - Combine: trivial since sorting is in place

QUICK SORT

Rough idea

- A divide-and-conquer algorithm
  - Divide: partition array into two subarrays such that the items in the lower part $\leq$ the items in the upper part
  - Conquer: recursively sort the two subarrays
  - Combine: trivial since sorting is in place

The algorithm

Algo: QuickSort(A,l,r)

if $l<r$ **then**
  m = LomutoPartition(A,l,r);
  QuickSort(A,l,m-1);
  QuickSort(A,m+1,r);

LOMUTO PARTITIONING / 1

element to the right is our middle element

Lomuto partitioning

Algo: LomutoPartition(A,l,r)

```
x = A[r];
i = l-1;
for j = l to r-1 do
    if A[j]≤x then
        i = i+1;
        exchange A[i] and A[j];
exchange A[i+1] and A[r];
return i+1;
```

Lomuto Partitioning/2

Lomuto partitioning of $[2, 7, 6, 1, 3, 5, 4]$

- $i = 0, j = 1$

2	7	6	1	3	5	4

Lomuto Partitioning / 2

Lomuto partitioning of $[2, 7, 6, 1, 3, 5, 4]$

- $i = 0, j = 1$

2	7	6	1	3	5	4

- $i = 1, j = 2$

2	7	6	1	3	5	4

Lomuto Partitioning / 2

Lomuto partitioning of $[2, 7, 6, 1, 3, 5, 4]$

- $i = 0, j = 1$

2	7	6	1	3	5	4

- $i = 1, j = 2$

2	7	6	1	3	5	4

- $i = 1, j = 3$

2	7	6	1	3	5	4

Lomuto Partitioning / 2

Lomuto partitioning of $[2, 7, 6, 1, 3, 5, 4]$

- $i = 0, j = 1$

2	7	6	1	3	5	4

- $i = 1, j = 2$

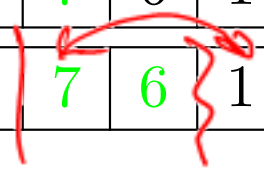
2	7	6	1	3	5	4

- $i = 1, j = 3$

2	7	6	1	3	5	4

- $i = 1, j = 4$

2	7	6	1	3	5	4



Lomuto Partitioning / 2

Lomuto partitioning of $[2, 7, 6, 1, 3, 5, 4]$

- $i = 0, j = 1$

2	7	6	1	3	5	4

- $i = 1, j = 2$

2	7	6	1	3	5	4

- $i = 1, j = 3$

2	7	6	1	3	5	4

- $i = 1, j = 4$

2	7	6	1	3	5	4

- $i = 2, j = 5$

2	1	6	7	3	5	4

Lomuto Partitioning/2

Lomuto partitioning of $[2, 7, 6, 1, 3, 5, 4]$

- $i = 0, j = 1$

2	7	6	1	3	5	4

- $i = 1, j = 2$

2	7	6	1	3	5	4

- $i = 1, j = 3$

2	7	6	1	3	5	4

- $i = 1, j = 4$

2	7	6	1	3	5	4

- $i = 2, j = 5$

2	1	6	7	3	5	4

- $i = 3, j = 6$

2	1	3	7	6	5	4

Lomuto Partitioning / 2

Lomuto partitioning of $[2, 7, 6, 1, 3, 5, 4]$

- $i = 0, j = 1$

2	7	6	1	3	5	4

- $i = 1, j = 2$

2	7	6	1	3	5	4

- $i = 1, j = 3$

2	7	6	1	3	5	4

- $i = 1, j = 4$

2	7	6	1	3	5	4

- $i = 2, j = 5$

2	1	6	7	3	5	4

- $i = 3, j = 6$

2	1	3	7	6	5	4

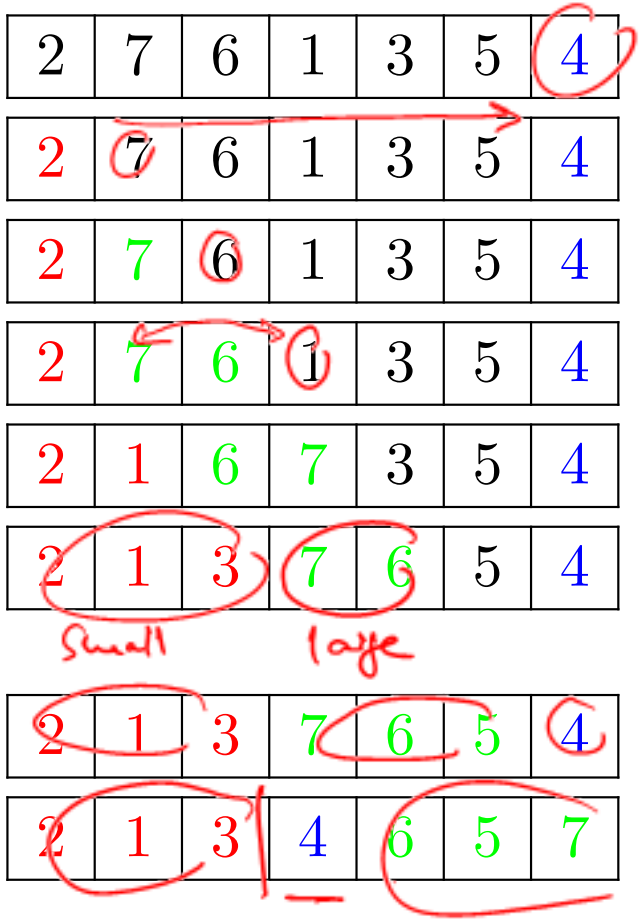
- $i = 3, j = 6$

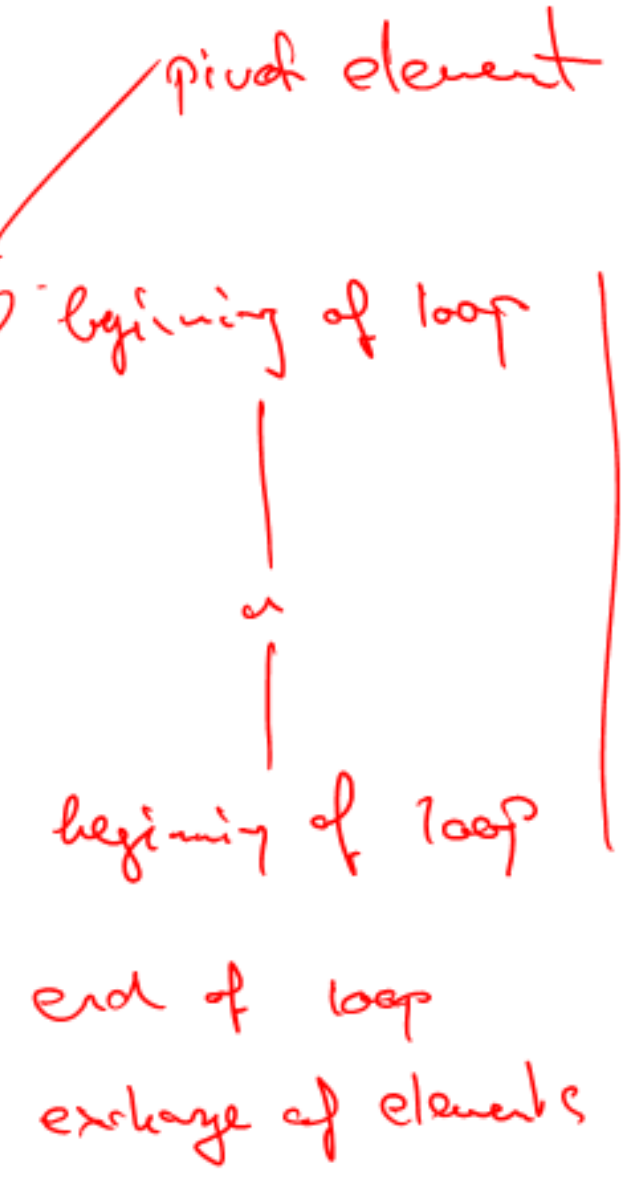
2	1	3	7	6	5	4

Lomuto Partitioning / 2

Lomuto partitioning of $[2, 7, 6, 1, 3, 5, 4]$

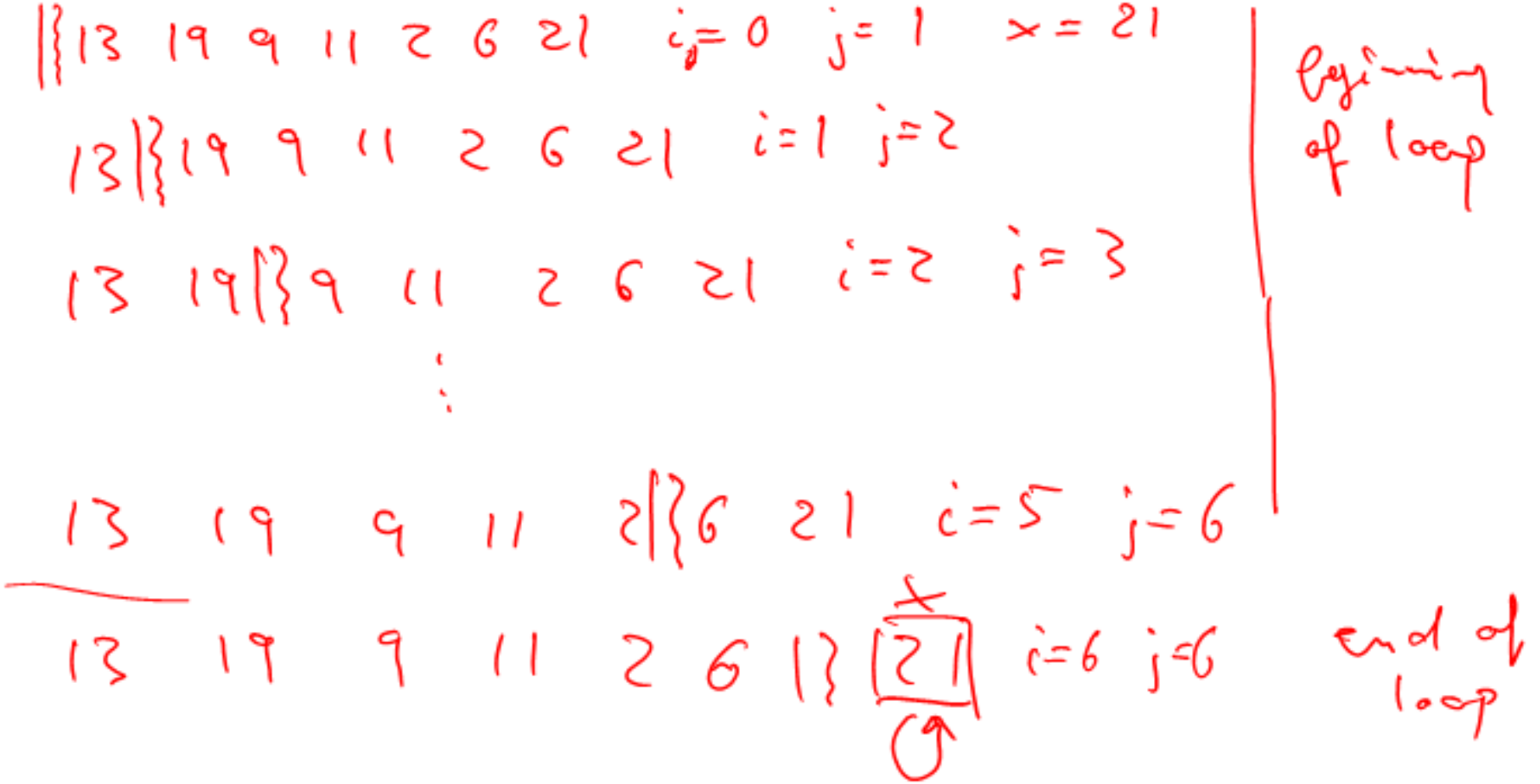
- $i = 0, j = 1$
- $i = 1, j = 2$
- $i = 1, j = 3$
- $i = 1, j = 4$
- $i = 2, j = 5$
- $i = 3, j = 6$
- $i = 3, j = 6$
- $i = 3, j = 6$



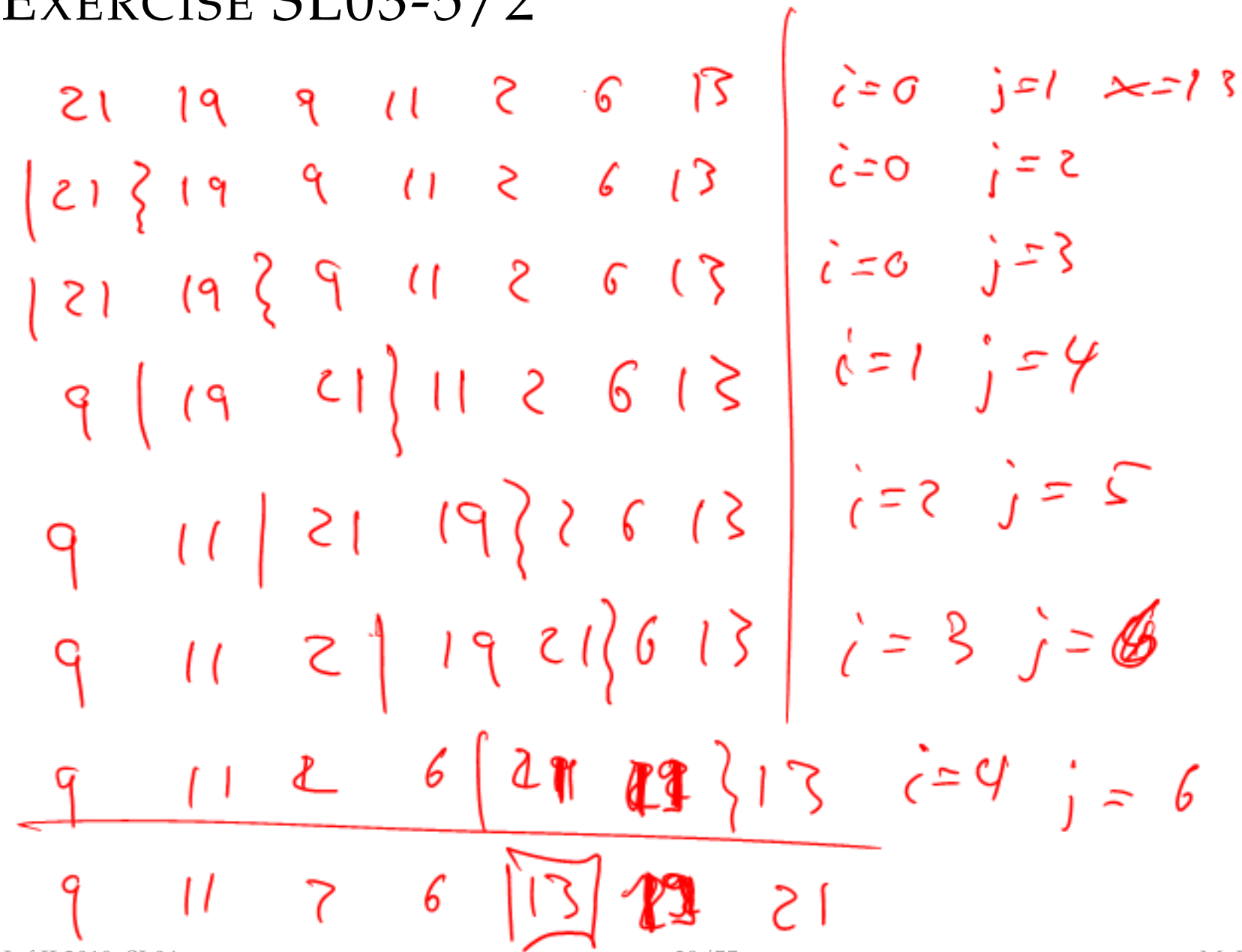


EXERCISE SL04-5/1

Demonstrate Lomuto partitioning on the arrays
$A1 = [13, 19, 9, 11, 2, 6, 21]$ and $A2 = [21, 19, 9, 11, 2, 6, 13]$.



EXERCISE SL03-5/2



EXERCISE SL04-6

Formulate the invariant for the Lomuto partitioning.



invariant:
(start of the loop)

- $A[l..i] \leq x$
- $A[i+1..j-1] > x$
- $A[r] = x$

left or right can be empty

termination $j = r-1$

$l-1 \leq i < r$

- $i = l-1 \Rightarrow A[l..l-1]$ is empty
- $i = r-1 \Rightarrow A[r..r-1]$ is empty

Hoare Partitioning/1

searching in right part for a wrong (too small) element

searching in left part for a wrong (too large) element

swap wrong elements

Hoare partitioning

Algo: HoarePartition(A,l,r)

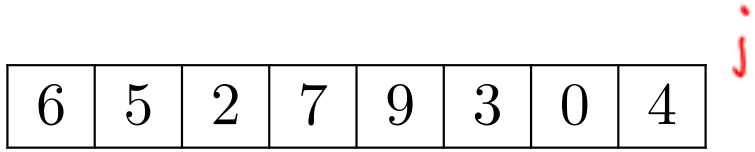
```
x = A[r]; i = l-1; j = r+1;
while true do
    repeat j = j-1 until A[j]≤x;
    repeat i = i+1 until A[i]≥x;
    if i<j then exchange A[i] and A[j] else return i;
```

Hoare Partitioning/2



Hoare partitioning of $[6, 5, 2, 7, 9, 3, 0, 4]$

- $i = 0, j = 9$



Hoare Partitioning/2

Hoare partitioning of $[6, 5, 2, 7, 9, 3, 0, 4]$

- $i = 0, j = 9$

6	5	2	7	9	3	0	4

- $i = 1, j = 8$

4	5	2	7	9	3	0	6

Hoare Partitioning/2

Hoare partitioning of $[6, 5, 2, 7, 9, 3, 0, 4]$

- $i = 0, j = 9$

6	5	2	7	9	3	0	4

- $i = 1, j = 8$

4	5	2	7	9	3	0	6

- $i = 2, j = 7$

4	0	2	7	9	3	5	6

Hoare Partitioning/2

Hoare partitioning of $[6, 5, 2, 7, 9, 3, 0, 4]$

- $i = 0, j = 9$

6	5	2	7	9	3	0	4

- $i = 1, j = 8$

4	5	2	7	9	3	0	6

- $i = 2, j = 7$

4	0	2	7	9	3	5	6

- $i = 4, j = 6$

4	0	2	3	9	7	5	6

Hoare Partitioning/2

Hoare partitioning of $[6, 5, 2, 7, 9, 3, 0, 4]$

- $i = 0, j = 9$

6	5	2	7	9	3	0	4

- $i = 1, j = 8$

4	5	2	7	9	3	0	6

- $i = 2, j = 7$

4	0	2	7	9	3	5	6

- $i = 4, j = 6$

4	0	2	3	9	7	5	6

- $i = 5, j = 4$

4	0	2	3	9	7	5	6

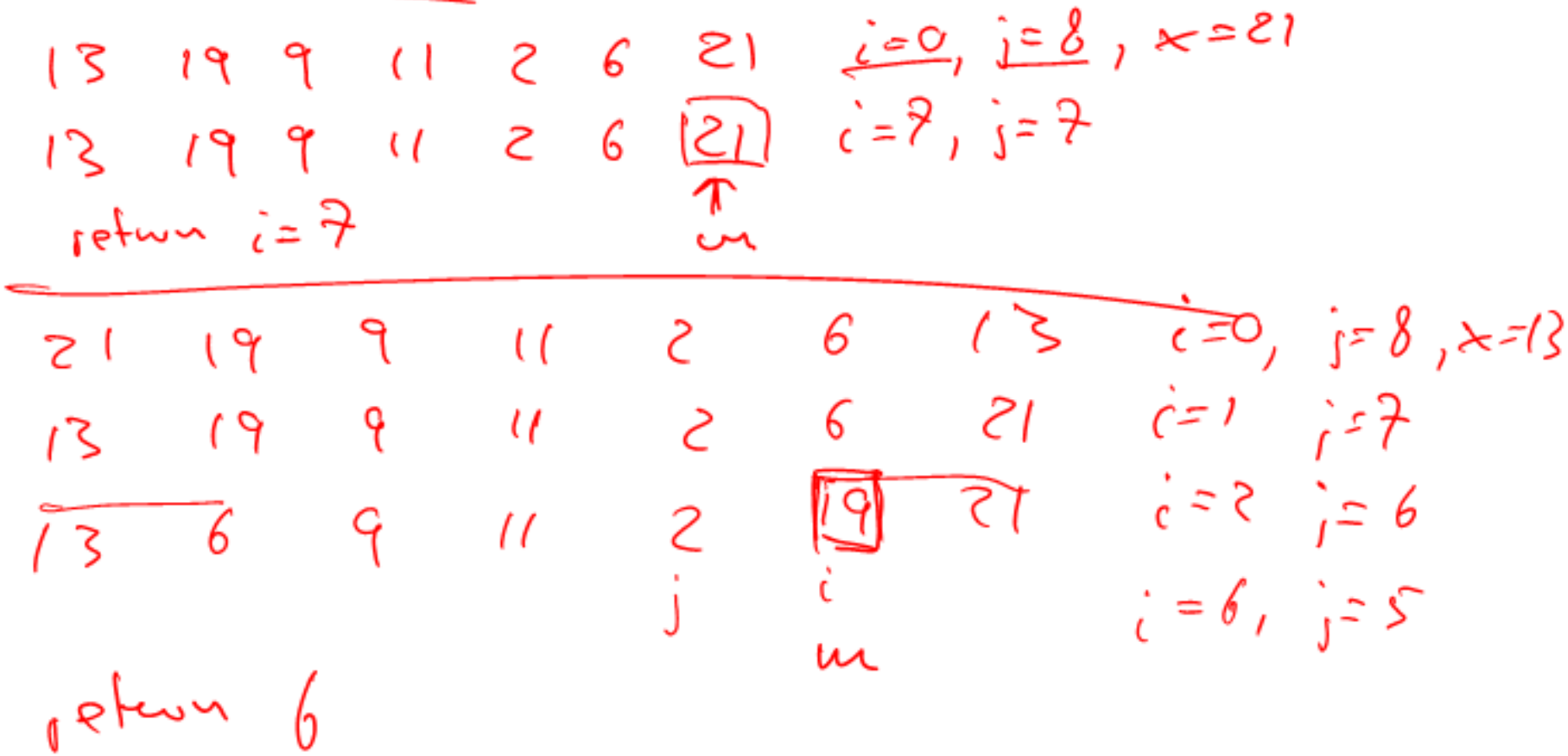
j i

ℓ i-1 i r

at end of loop

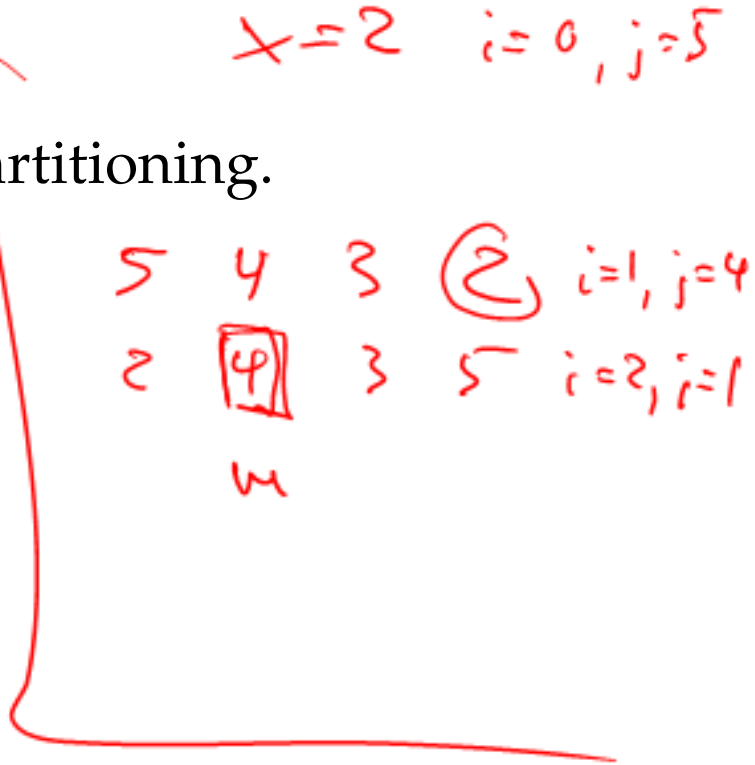
EXERCISE SL04-7

Demonstrate Hoare partitioning on the arrays
$A1 = [13, 19, 9, 11, 2, 6, 21]$ and $A2 = [21, 19, 9, 11, 2, 6, 13]$.



EXERCISE SL04-8

Formulate the invariant for the Hoare partitioning.



invariant

- $A[l \,..\, i-1] \leq x$
- $A[j+1 \,..\, r] \geq x$

termination:

$l < i \leq r \;\}\Rightarrow$ $A[l \,..\, i-1]$ and $A[i \,..\, r]$ are both non-empty !

$l \leq j \leq r$

EXERCISE SL04-9

Specify the quick sort algorithm for Hoare partitioning.

```
QS(A, l, r)
    if l < r then
    m = HoarePart(A, l, r)
    QS(A, l, m-1)
    QS(A, m, r)
```

EXERCISE SL04-10/1

What happens if

1. line 01 HoarePartition: $A[r]$ is changed to $A[l]$
2. line 05 HoarePartition: $i < j$ is changed to $i \leq j$
3. QS line 01: $l < r$ is changed to $l \leq r$
4. QS line 03: m-1 is changed to m
   QS line 04: m is changed to $m + 1$

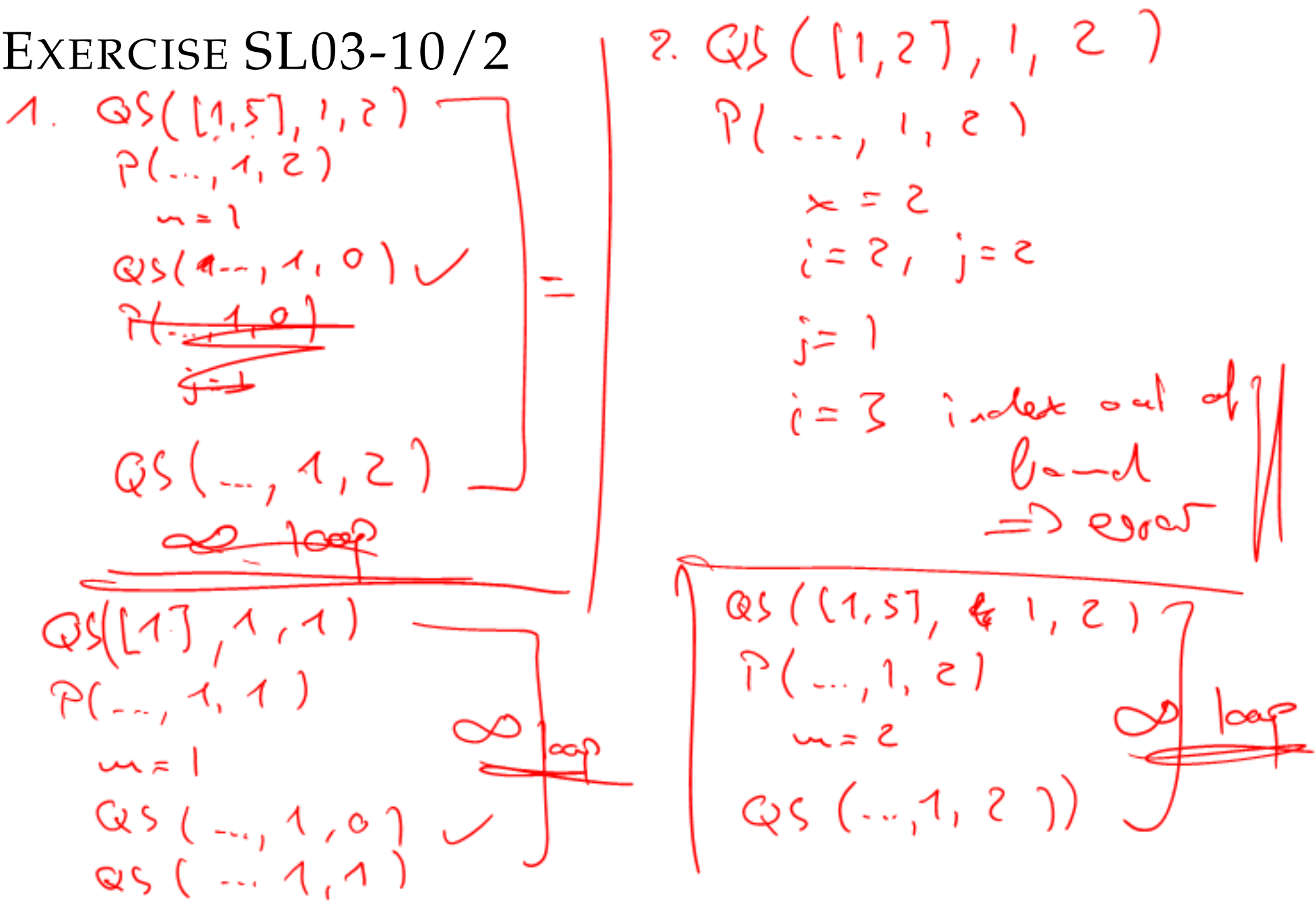
Algo: HoarePartition(A,l,r)

```
x = A[r]; i = l-1; j = r+1;
while true do
    repeat j = j-1 until A[j]≤x;
    repeat i = i+1 until A[i]≥x;
    if i<j then
        exchange A[i] and A[j]
    else
        return i
```

Algo: QuickSort(A,l,r)

```
if l<r then
    m = HoarePartition(A,l,r);
    QuickSort(A,l,m-1);
    QuickSort(A,m,r);
```

EXERCISE SL03-10/2



PERFORMANCE/1

Worst case

- Occurs when the array is already completely sorted

PERFORMANCE/1

Worst case

- Occurs when the array is already completely sorted
- Partitioning produces one subproblem with $n - 1$ items and one with 0 elements

PERFORMANCE/1

Worst case

- Occurs when the array is already completely sorted
- Partitioning produces one subproblem with $n-1$ items and one with 0 elements
- If such a partitioning arises at each recursive call then

$$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$$

where $\Theta(n)$ is the cost of partitioning the array

Performance/1

Worst case

- Occurs when the array is already completely sorted
- Partitioning produces one subproblem with $n-1$ items and one with 0 elements
- If such a partitioning arises at each recursive call then

$$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$$

  where $\Theta(n)$ is the cost of partitioning the array
- Sum of the costs incurred at each level of the recursion yields an arithmetic series (i.e., $\sum_{i=0}^{n} i$)

PERFORMANCE/1

Worst case

- Occurs when the array is already completely sorted
- Partitioning produces one subproblem with $n-1$ items and one with 0 elements
- If such a partitioning arises at each recursive call then

$$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$$

  where $\Theta(n)$ is the cost of partitioning the array
- Sum of the costs incurred at each level of the recursion yields an arithmetic series (i.e., $\sum_{i=0}^{n} i$)
- It thus follows that, in the worst case, $T(n) = \Theta(n^2)$

PERFORMANCE/2

Best case

- Partitioning produces two subproblems of size $n/2$



PERFORMANCE/2

Best case

- Partitioning produces two subproblems of size $n/2$
- If such a partitioning arises at each recursive call then

$$T(n) = 2T(n/2) + \Theta(n)$$

PERFORMANCE/2

Best case

- Partitioning produces two subproblems of size $n/2$
- If such a partitioning arises at each recursive call then

$$T(n) = 2T(n/2) + \Theta(n)$$

- It thus follows that, in the best case, $T(n) = \Theta(n \lg n)$

Performance/2

Best case

- Partitioning produces two subproblems of size $n/2$
- If such a partitioning arises at each recursive call then

$$T(n) = 2T(n/2) + \Theta(n)$$

- It thus follows that, in the best case, $T(n) = \Theta(n \lg n)$

Average case/1

- Actually much closer to best case than to worst case

PERFORMANCE/2

Best case

- Partitioning produces two subproblems of size $n/2$
- If such a partitioning arises at each recursive call then

$$T(n) = 2T(n/2) + \Theta(n)$$

- It thus follows that, in the best case, $T(n) = \Theta(n \lg n)$

Average case/1

- Actually much closer to best case than to worst case
- Quick sort's behavior is determined by the ordering of the array elements

PERFORMANCE/2

Best case

- Partitioning produces two subproblems of size $n/2$
- If such a partitioning arises at each recursive call then

$$T(n) = 2T(n/2) + \Theta(n)$$

- It thus follows that, in the best case, $T(n) = \Theta(n \lg n)$

Average case/1

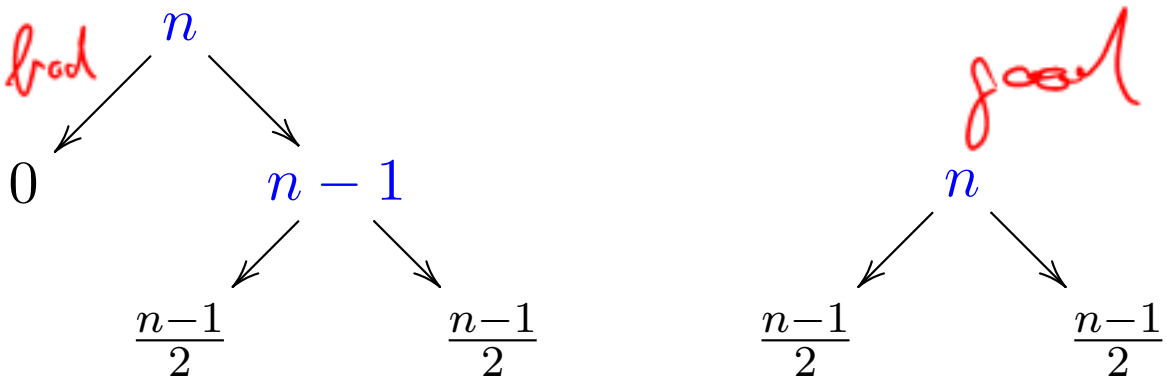
- Actually much closer to best case than to worst case
- Quick sort's behavior is determined by the ordering of the array elements
- In average, there is a mix of "good" and "bad" splits



Performance/3

Average case/2

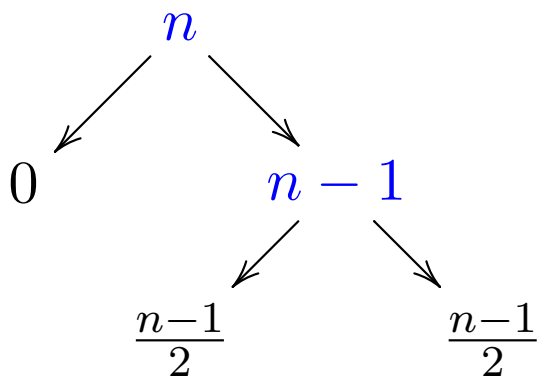
- Assume that "bad" (B) and "good" (G) splits alternate

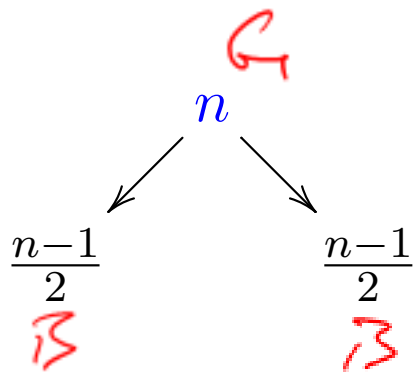


Performance/3

Average case/2

- Assume that "bad" (B) and "good" (G) splits alternate



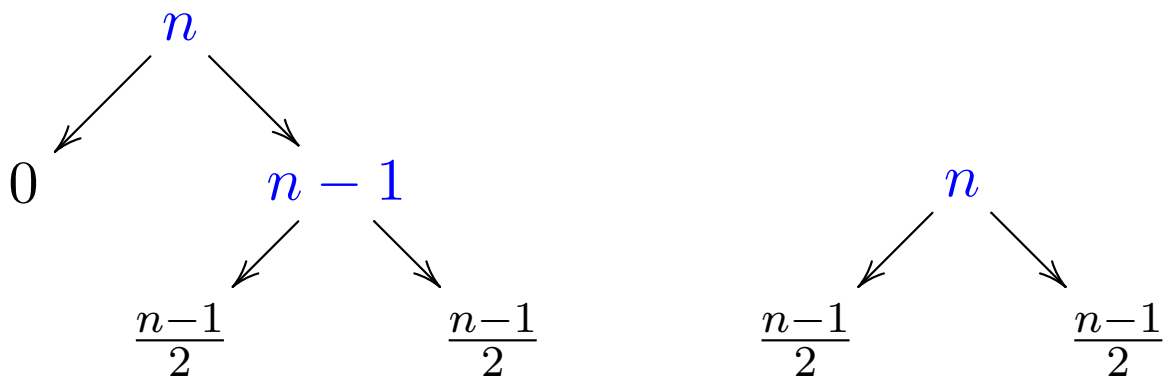


- $G(n) = 2B(n/2) + \Theta(n)$, and
  $B(n) = G(n-1) + \Theta(n)$

PERFORMANCE/3

Average case/2

- Assume that "bad" (B) and "good" (G) splits alternate

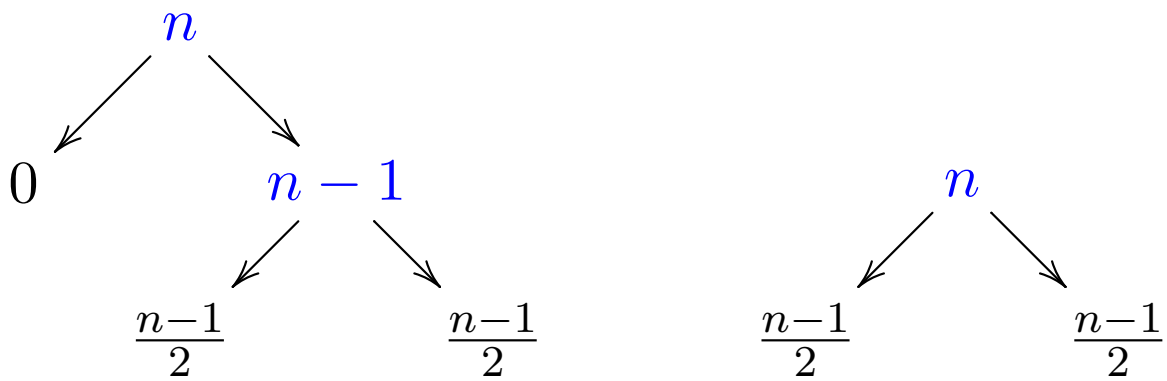


- $G(n) = 2B(n/2) + \Theta(n)$, and
  $B(n) = G(n-1) + \Theta(n)$
- $G(n) = 2B(n/2) + \Theta(n)$
  $= 2G(n/2 - 1) + \Theta(n/2) + \Theta(n)$
  $= 2G(n/2 - 1) + \Theta(n)$ $< 2G(n/2) + \Theta(n)$

PERFORMANCE/3

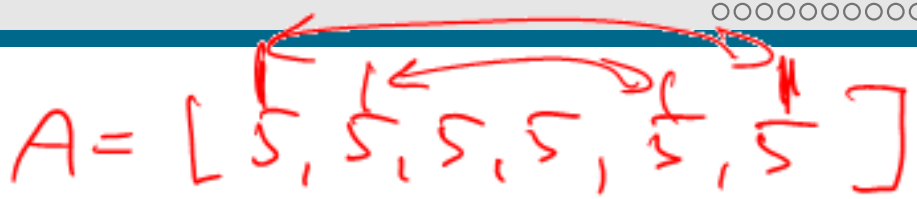
Average case/2

- Assume that "bad" (B) and "good" (G) splits alternate



- $G(n) = 2B(n/2) + \Theta(n)$, and
  $B(n) = G(n-1) + \Theta(n)$
- $$\begin{aligned} G(n) &= 2B(n/2) + \Theta(n) \\ &= 2G(n/2-1) + \Theta(n/2) + \Theta(n) \\ &= 2G(n/2-1) + \Theta(n) \end{aligned}$$
- It follows that, on average, $T(n) = \Theta(n \lg n)$

EXERCISE SL04-11



Consider an array where all elements are equal. Is this a worst case, best case or neither for quick sort?

Algo: LomutoPartition(A,l,r)

```
x = A[r]; i = l-1;
for j = l to r-1 do
    if A[j]≤x then
        i = i+1;
        exchange A[i] and A[j];
exchange A[i+1] and A[r];
return i+1;
```

Algo: HoarePartition(A,l,r)

```
x = A[r]; i = l-1; j = r+1;
while true do
    repeat j = j-1 until A[j]≤x;
    repeat i = i+1 until A[i]≥x;
    if i<j then
        exchange A[i] and A[j]
    else
        return i
```

- Lomuto: elements go all to left or right ⇒ worst case
- Hoare: splits in middle ⇒ best case

PERFORMANCE/4

Average case/3

- Underlying assumption: all permutations of the input numbers are equally likely

Performance/4

Average case/3

- Underlying assumption: all permutations of the input numbers are equally likely
- Randomized algorithm: partition around random item (instead of ~~last~~ item)
  - Running time is independent of input ordering
  - No specific input triggers worst case behavior
  - Worst case determined only by output of the random-number generator

PERFORMANCE/4

Average case/3

- Underlying assumption: all permutations of the input numbers are equally likely
- Randomized algorithm: partition around random item (instead of last item)
  - Running time is independent of input ordering
  - No specific input triggers worst case behavior
  - Worst case determined only by output of the random-number generator
- Randomization is a general tool to improve algorithms with bad worst-case but good average-case complexity

PERFORMANCE/5

improve stability further:
- take middle element of three random picks

Randomized algorithm

Algo: RandomizedLomutoPartition(A,l,r)

```
i = Random(l,r);
exchange A[i] and A[r];
return LomutoPartition(A,l,r);
```

Algo: RandomizedQuickSort(A,l,r)

```
if l<r then
    m = RandomizedLomutoPartition(A,l,r);
    RandomizedQuickSort(A,l,m-1);
    RandomizedQuickSort(A,m+1,r);
```

SUMMARY / 1

- Nearly complete binary trees
- Heap data structure
- Heapsort
  - based on heaps
  - worst case is n log n
- Quicksort:
  - partition based sort algorithm
  - popular algorithm
  - very fast on average
  - worst case performance is quadratic

SUMMARY / 2

- Comparison of sorting methods.
- Absolute runtime values are not important; always relate values to each other.
- Relate runtime values to the complexity ($n \log n$, n^2).
- Running time in seconds for $n = 2048$ and different sort orders:

	ordered	random	inverse
insertion sort	0.22	50.74	103.80
selection sort	58.18	58.34	73.46
bubble sort	80.18	128.84	178.66
heap sort	2.32	2.22	2.12
quick sort (random pivot)	0.72	1.22	0.76



Informatik II

Pointers, Abstract Data Types
— SL05 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

TABLE OF CONTENTS — SL05

1. Dynamic data structures
   Pointers
   Linked lists

2. ADT: abstract data types

DYNAMIC DATA STRUCTURES/1

Arrays as data structures

- Used to store and manipulate collections of items
  - Have the advantage that one can directly access items

DYNAMIC DATA STRUCTURES/1

Arrays as data structures

- Used to store and manipulate collections of items
  - Have the advantage that one can directly access items
- But there are certain limitations or disadvantages
  - Dimension of an array is fixed and cannot be changed
  - Occupies an amount of memory that must be determined before program is run (i.e., can be too large or too small)
  - Inserting/removing items may be expensive operations

insertion sort!

DYNAMIC DATA STRUCTURES/1

Arrays as data structures

- Used to store and manipulate collections of items
  - Have the advantage that one can directly access items
- But there are certain limitations or disadvantages
  - Dimension of an array is fixed and cannot be changed
  - Occupies an amount of memory that must be determined before program is run (i.e., can be too large or too small)
  - Inserting/removing items may be expensive operations
- A possible way out: use of dynamic data structures
  - Resources are allocated only when they are needed
  - Items are represented by records/objects whose fields can be accessed if one has a pointer to the object
  - Change the way in which items are inserted/removed

DYNAMIC DATA STRUCTURES/2

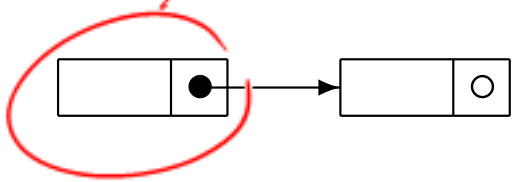
Linked data structures

- Used to deal with collections of items dynamically

DYNAMIC DATA STRUCTURES/2

Linked data structures

- Used to deal with collections of items dynamically
- Record contains fields that point to other records

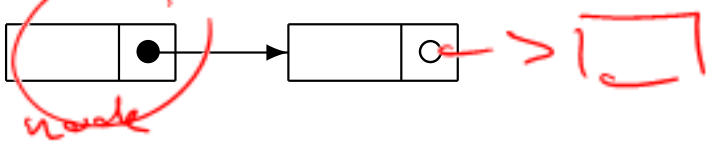


- Linked list: each item points to next item in the list
- Binary tree: each node points to left & right subtree

DYNAMIC DATA STRUCTURES/2

Linked data structures

- Used to deal with collections of items dynamically
- Record contains fields that point to other records



- Linked list: each item points to next item in the list
- Binary tree: each node points to left & right subtree

- Storage space of linked data structures (LDS) is not known in advance
  - Depends on the number of items stored in the LDS
  - Only known during runtime, i.e. program execution
  - Linked data structures can grow and shrink in size

RECORDS IN C

In C a `struct` is used to group fields:

```
struct rec { int a; int b; };
struct rec r;

int main() {
  r.a = 5; r.b = 8;
  printf("Sum a and b is %d\n", r.a + r.b);
}
// gcc -o dummy dummy.c ; ./dummy
```

accesses a component of a struct

RECURSIVE DATA STRUCTURES

The counterpart of recursive functions are recursively defined data structures.

Example: list of integers = { integer | integer, list of integer }

base case
rec case

```
struct list {
  int value;
  struct list tail;
};
```



There must be a mechanism to constrain the initial storage space of recursive data structures (it is potentially infinite).
There must be a mechanism to grow and shrink the storage space of a recursive data structures during program execution.

pointers

POINTERS / 1

Informal definition of pointers

- A common technique is to allocate storage space dynamically

POINTERS / 1

Informal definition

- A common technique is to allocate storage space dynamically
- That means the storage space (memory) is allocated during runtime

POINTERS / 1

Informal definition

- A common technique is to allocate storage space dynamically
- That means the storage space (memory) is allocated during runtime
- Compiler reserves space only for addresses to these dynamic parts

POINTERS / 1

Informal definition

- A common technique is to allocate storage space dynamically
- That means the storage space (memory) is allocated during runtime
- Compiler reserves space only for addresses to these dynamic parts
- These addresses are usually called pointers

POINTERS/2

= (huge) array

Memory management

	address	content
i	$1af783$	22
p	$1af784$	$1af78a$
r	$1af785$	33
	$1af786$	NIL
s	$1af787$	$1af785$
	$1af788$	—
	$1af789$	—
	$1af78a$	44

hex numbers

POINTERS/2

Memory management

	address	content
i	$1af783$	22
p	$1af784$	$1af78a$
r	$1af785$	33
	$1af786$	`NIL`
s	$1af787$	$1af785$
	$1af788$	—
	$1af789$	—
	$1af78a$	44

- variable i: an integer (22)

POINTERS/2

Memory management

	address	content
i	$1af783$	22
p	$1af784$	$1af78a$
r	$1af785$	33
	$1af786$	NIL
s	$1af787$	$1af785$
	$1af788$	—
	$1af789$	—
	$1af78a$	44

- variable i: an integer (22)
- variable p: pointer to an integer (44)

POINTERS/2

Memory management

	address	content
i	$1af783$	22
p	$1af784$	$1af78a$
r	$1af785$	33
	$1af786$	NIL
s	$1af787$	$1af785$
	$1af788$	—
	$1af789$	—
	$1af78a$	44

- variable i: an integer (22)
- variable p: pointer to an integer (44)
- variable r: record with 2 components
  - an integer (33)
  - a pointer to NIL

POINTERS/2

Memory management

	address	content
i	$1af783$	22
p	$1af784$	$1af78a$
r	$1af785$	33
	$1af786$	NIL
s	$1af787$	$1af785$
	$1af788$	—
	$1af789$	—
	$1af78a$	44

- variable i: an integer (22)
- variable p: pointer to an integer (44)
- variable r: record with 2 components
  - an integer (33)
  - a pointer to NIL
- variable s: pointer to the record r

EXERCISE SL05-1

What is the difference between an address and a pointer?

pointer = address + type

POINTERS / 3

Alternative presentation

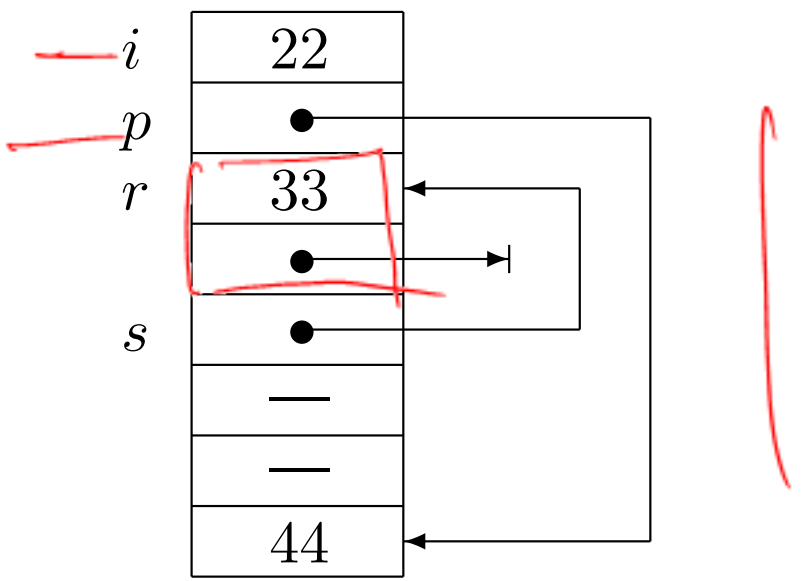
	address	content
i	$1af783$	22
p	$1af784$	$1af78a$
r	$1af785$	33
	$1af786$	`NIL`
s	$1af787$	$1af785$
	$1af788$	—
	$1af789$	—
	$1af78a$	44

POINTERS / 3

always work with graph illustrations when using pointers

Alternative presentation

	address	content
i	$1af783$	22
p	$1af784$	$1af78a$
r	$1af785$	33
	$1af786$	NIL
s	$1af787$	$1af785$
	$1af788$	—
	$1af789$	—
	$1af78a$	44



POINTERS / 4

Concluding remarks

- Pointers are not the only way to implement dynamic data structures

POINTERS / 4

Concluding remarks

- Pointers are not the only way to implement dynamic data structures
- The programmer does not have to be aware of their existence
  - Indeed, in Java objects are implemented with pointers
  - Creation of objects (i.e. new) allocates storage space
  - Accessing an object is done by following the pointer
  - Deallocation is done by so called garbage collector

POINTERS / 4

Concluding remarks

- Pointers are not the only way to implement dynamic data structures
- The programmer does not have to be aware of their existence
  - Indeed, in Java objects are implemented with pointers
  - Creation of objects (i.e. new) allocates storage space
  - Accessing an object is done by following the pointer
  - Deallocation is done by so called garbage collector
- Certain programming languages (such as C) require that the storage space is managed explicitly

POINTERS IN C/1

operations on pointers:

- To follow (chase, dereference) a pointer we write **p*

```
*p = 12
```

- To get the address of a variable *i* we write *&i*

```
p = &i
```

- To allocate memory we use *malloc(sizeof(Type))*

```
p = malloc(sizeof(int))
```

- To free storage space pointed to by a pointer *p* we use *free*

```
free(p)
```

POINTERS IN C/2

declaration of pointers:

- To declare a pointer to type T we write T*

```
int* p
```

- Note that * is used for two purposes:
  - Declaring a pointer variable

```
int* p
```

  - Following a pointer

```
*p = 15
```

In other languages these are syntactically different.

POINTERS IN C/3

```
int i
i = 23

int* p
p = malloc(sizeof(int))
*p = 55

struct rec r
rec.a = 17
rec.b = 24

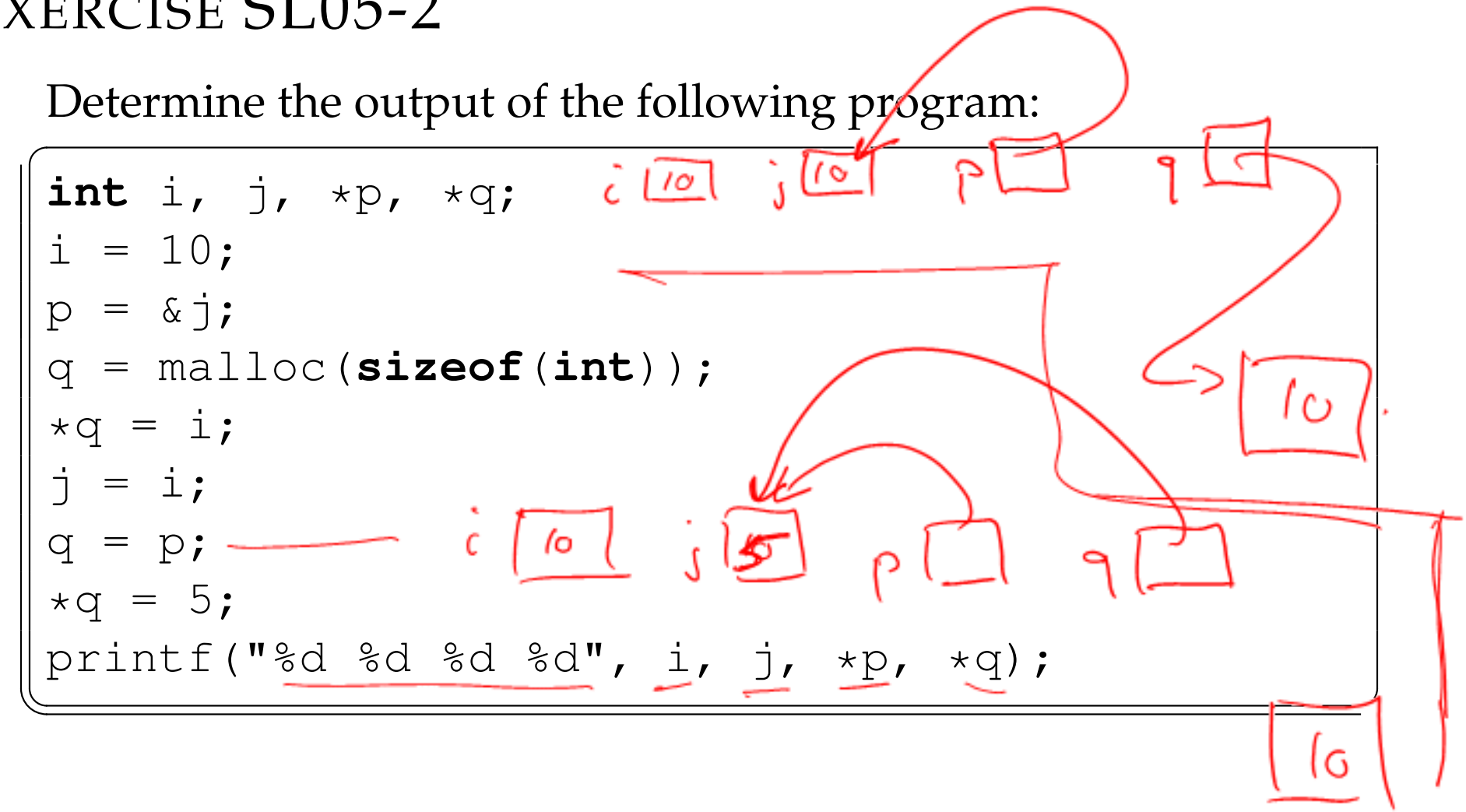
struct rec* s;
s = &r
```

	address	content
i	$1af783$	23
p	$1af784$	$1af78a$
r	$1af785$	17
	$1af786$	24
s	$1af787$	$1af785$
	$1af788$	
	$1af789$	
	$1af78a$	55

EXERCISE SL05-2

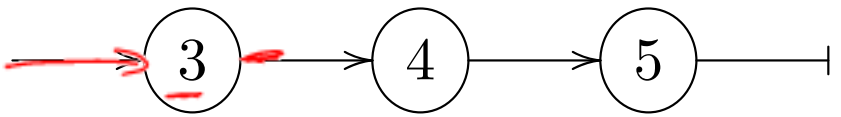
Determine the output of the following program:

```
int i, j, *p, *q;
i = 10;
p = &j;
q = malloc(sizeof(int));
*q = i;
j = i;
q = p;
*q = 5;
printf("%d %d %d %d", i, j, *p, *q);
```



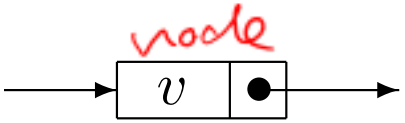
10 5 5 5

LINKED LISTS/1

- A list of integer: *root* 

- Notational convention:

  Use 

  instead of 

- Corresponding declaration in C:

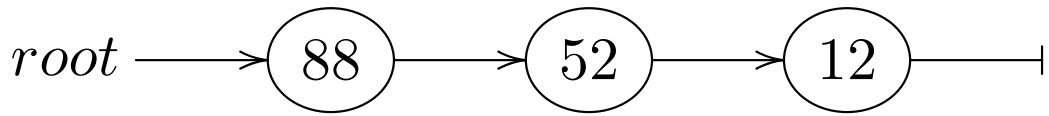
```
struct node {
  int val;
  struct node* next;
}

struct node* root;
```

- Accessing a field: `(*p).a` or more convenient `p->a`

LINKED LISTS/2

Populating a list with integers:

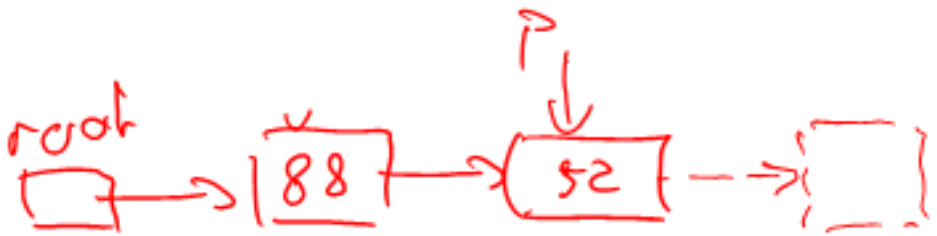



```
root = malloc(sizeof(struct node));
root->val = 88;
root->next = malloc(sizeof(struct node));

p = root->next;
p->val = 52;
p->next = malloc(sizeof(struct node));

p = p->next;
p->val = 12;
p->next = NULL;
```

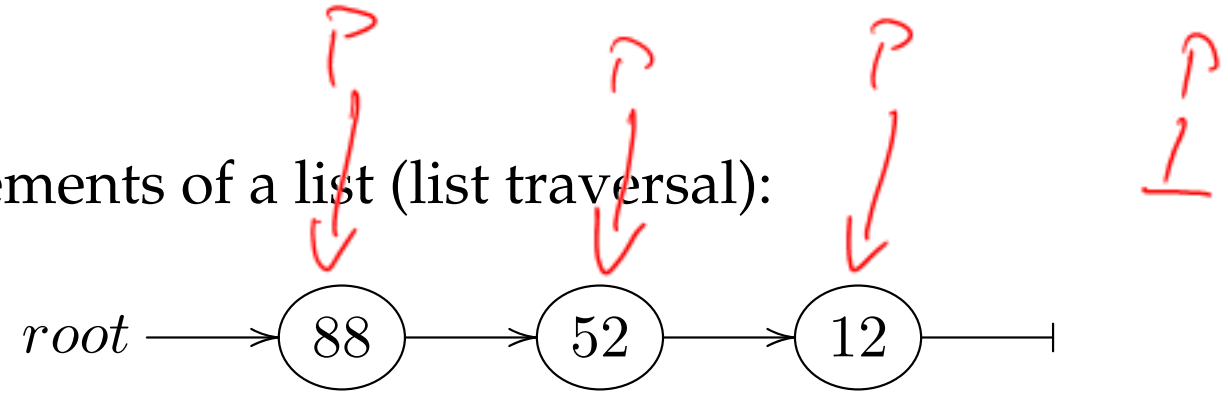






LINKED LISTS/3

Print all elements of a list (list traversal):




```
p = root;
while (p != NULL) {
  printf("%d,", p->val);
  p = p->next
}
printf("\n");
```

EXERCISE SL05-3

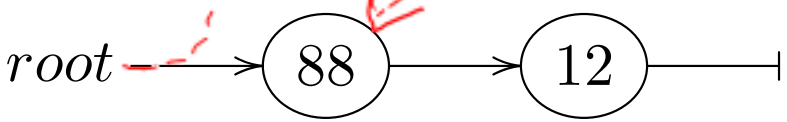
Count the occurrences of an integer x in a linked list.



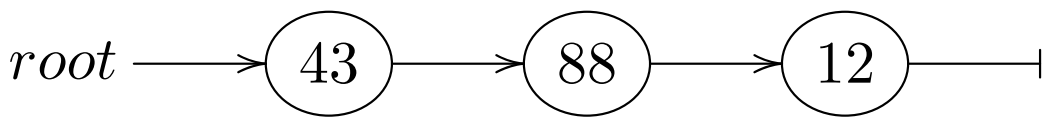

```
cnt = 0;
p = root;
while (p ≠ Nuce) {
    if (p->val == x) cnt++
    p = p->next
```

LINKED LISTS/4

Inserting 43 at the beginning of a list:

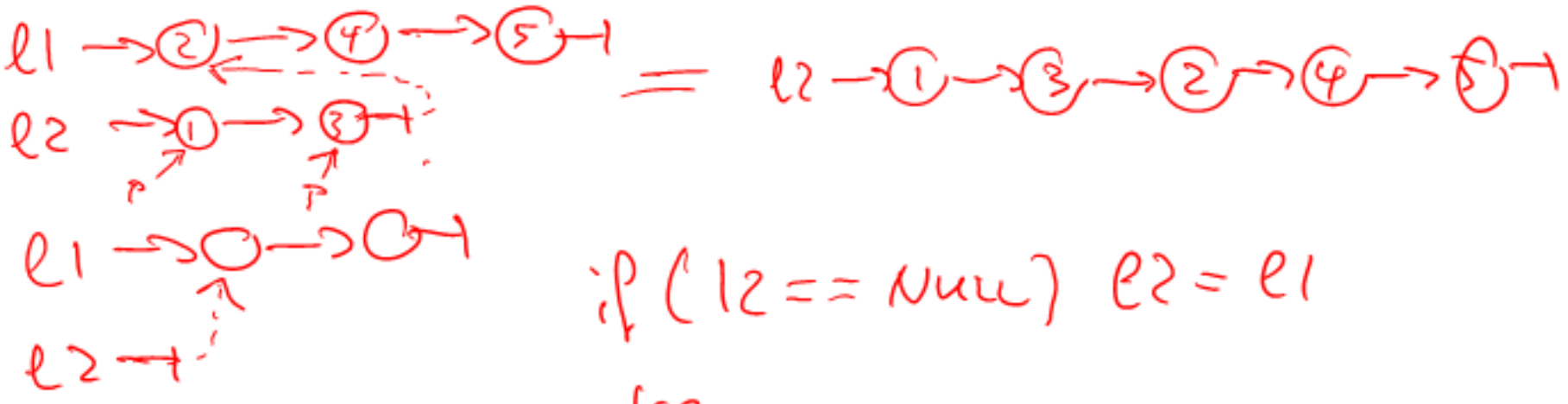



```
p = malloc(sizeof(struct node));
p->val = 43
p->next = root;
root = p;
```



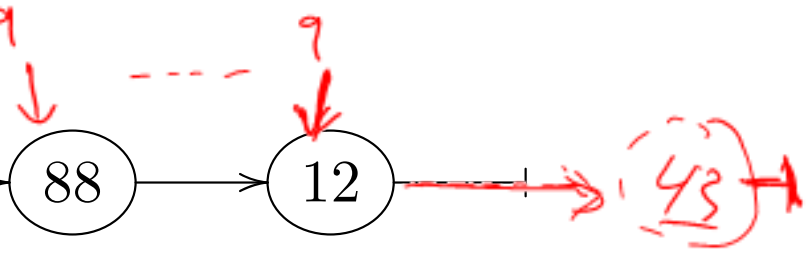
EXERCISE SL05-4

Give C code that appends linked list l1 to linked list l2.



LINKED LISTS/5

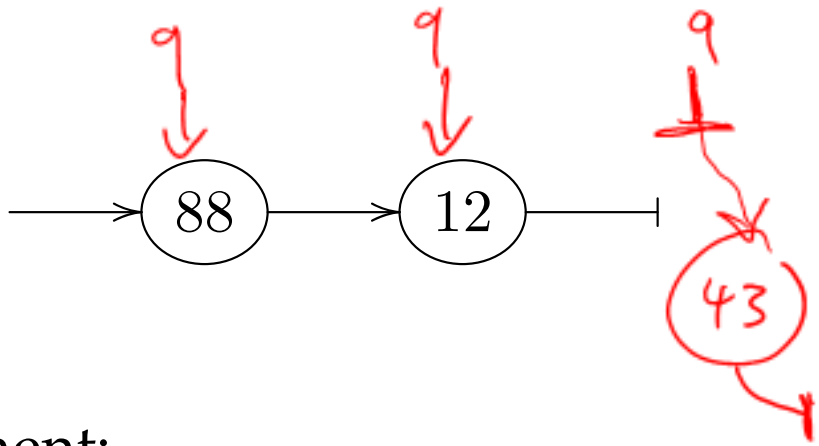
Insert 43 at the end of a list: *root*




```
if (root == NULL) {
  root = malloc(sizeof(struct node));
  root->val = 43;
  root->next = NULL;
} else {
  q = root;
  while (q->next != NULL) { q = q->next; }
  q->next = malloc(sizeof(struct node));
  q->next->val = 43;
  q->next->next = NULL;
}
```

LINKED LISTS/6

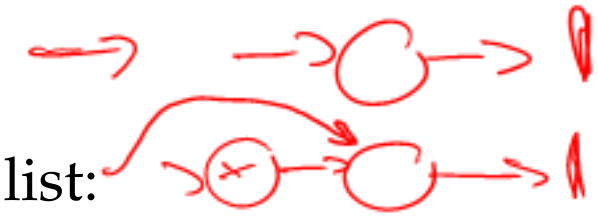
Insert 43 at the end of a list: *root*



Explain the following code fragment:

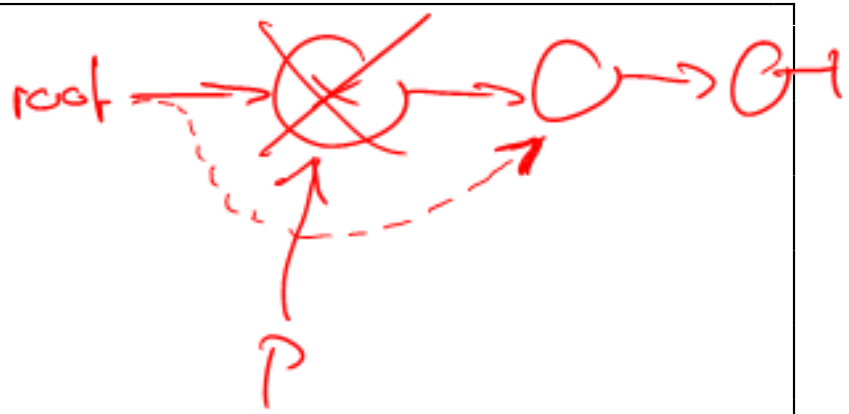
```
q = root;
while (q != NULL) { q = q->next; }
q = malloc(sizeof(struct node));
q->val = 43;
q->next = NULL;
```

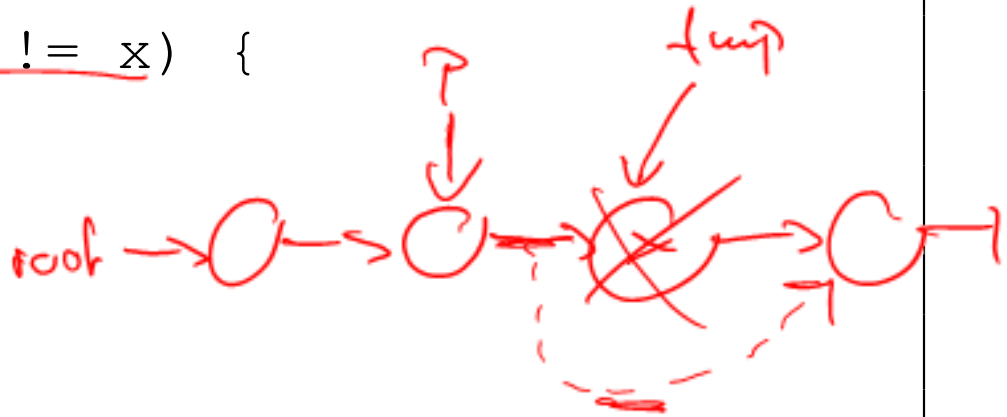
LINKED LISTS/7



Delete element x from a non-empty list:

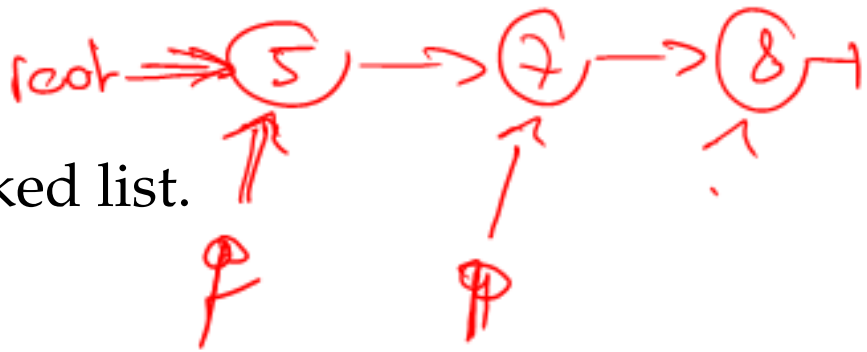
```
p = root;
if (p->val == x) {
  root = p->next;
  free(p);
} else {
  while (p->next != NULL &&
         p->next->val != x) {
    p = p->next;
  }
  tmp = p->next;
  p->next = tmp->next
  free(tmp);
}
```





EXERCISE SL05-5

Write C code to delete a linked list.




```
p = root;
while (p ≠ NULL)
    q = p
    p = p->next
    free(q)
```

```
void del(struct node* p)
    if (p ≠ NULL)
        del(p->next);
        free(p);
```

LINKED LISTS/8

Cost of linked list operations:

- Insertion at beginning: $O(1)$
- Insert at end: $O(n)$
- isEmpty: $O(1)$
- Delete: $O(n)$
- Print: $O(n)$

The code fragments for working with the linked list access the list through global variable `root`.

- Thus, there can be at most one linked list.
- The code does not allow us to define and use multiple linked lists (see ADTs for a solution)

DOUBLY LINKED LISTS

- more than 1 pointer
- use pointers differently

Variants of linked lists

- Lists with explicit tail
  - Do have an extra pointer to the last item of the list
  - No need to scan the entire list if an operation applies only to the last item

Doubly linked lists

Variants of linked lists

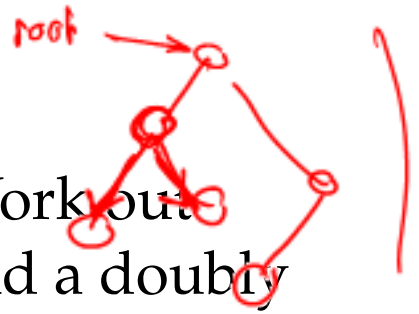
- Lists with explicit tail
  - Do have an extra pointer to the last item of the list
  - No need to scan the entire list if an operation applies only to the last item
- Doubly linked list



  - Each node has a field with a pointer to the previous node of the linked list
  - Provides means to quickly navigate forth and back in the linked list

EXERCISE SL05-6





Define a doubly linked list and a binary tree in C. Work out differences and similarities between a binary tree and a doubly linked list?

doubly linked list

used rarely

```
struct node {
    int val;
    struct node* next;
    struct node* prev;
};

struct node* head;
struct node* tail;
```

binary tree

used heavily

```
struct node {
    int val;
    struct node* lft;
    struct node* rgt;
};

struct node* root;
```

use of pointers that is different; structure is very similar

TABLE OF CONTENTS — SL05

1. Dynamic data structures

2. ADT: abstract data types
   - Stacks
   - Queues
   - Ordered lists

ABSTRACT DATA TYPES/1

Informal definition

- An abstract data type is a mathematical model that defines a data type by its behavior from the point of view of a user (in terms of possible values, possible operations, and the behavior of operations).

ABSTRACT DATA TYPES/1

Informal definition

- An abstract data type is a mathematical model that defines a data type by its behavior from the point of view of a user (in terms of possible values, possible operations, and the behavior of operations).
- It is equipped with a specific interface, i.e., a collection of signatures for the operations that can be invoked on an instance

ABSTRACT DATA TYPES/1

Informal definition

- An abstract data type is a mathematical model that defines a data type by its behavior from the point of view of a user (in terms of possible values, possible operations, and the behavior of operations).
- It is equipped with a specific interface, i.e., a collection of signatures for the operations that can be invoked on an instance
- Furthermore, it is equipped with a set of axioms (pre- and postconditions) that define the semantics of each operation (i.e. what the operations do to instances of the ADT, but not how)

Abstract data types/2

Why abstract data types?

- Give a language to talk on a higher level of abstraction

Abstract data types/2

Why abstract data types?

- Give a language to talk on a higher level of abstraction
- Encapsulate both data structures and algorithms that implement them

ABSTRACT DATA TYPES/2

Why abstract data types?

- Give a language to talk on a higher level of abstraction
- Encapsulate both data structures and algorithms that implement them
- Allow one to break work into pieces that can be seen as independent – without compromising correctness
  - Serve as specification of requirements for the building blocks of solutions to algorithmic problems

ABSTRACT DATA TYPES/2

Why abstract data types?

- Give a language to talk on a higher level of abstraction
- Encapsulate both data structures and algorithms that implement them
- Allow one to break work into pieces that can be seen as independent – without compromising correctness
  - Serve as specification of requirements for the building blocks of solutions to algorithmic problems
- Provide means to separate the concerns of correctness and performance analysis
  - Design the algorithm using an ADT
  - Count the number of ADT operations
  - Choose implementation of operations

ABSTRACT DATA TYPES/3

Similarity with OO-paradigm

- ADT = instance variables + procedures

ABSTRACT DATA TYPES/3

Similarity with OO-paradigm

- ADT = instance variables + procedures
- Class = instance variables + methods

ABSTRACT DATA TYPES/3

Similarity with OO-paradigm

- ADT = instance variables + procedures
- Class = instance variables + methods

Some popular examples

- Stacks & queues
  - LIFO vs. FIFO principle

ABSTRACT DATA TYPES/3

Similarity with OO-paradigm

- ADT = instance variables + procedures
- Class = instance variables + methods

Some popular examples

- Stacks & queues
  - LIFO vs. FIFO principle
- Priority queues
  - Another application of heaps

ABSTRACT DATA TYPES/3

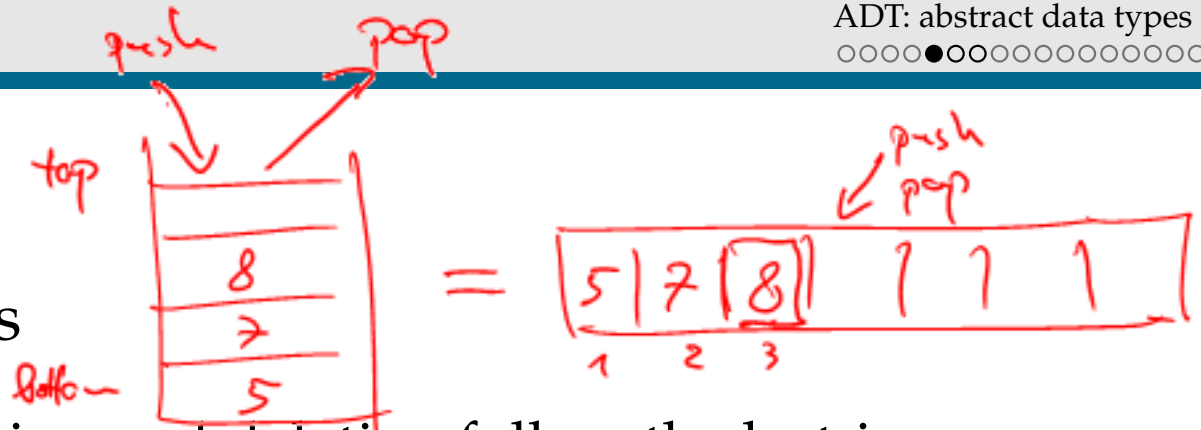
Similarity with OO-paradigm

- ADT = instance variables + procedures
- Class = instance variables + methods

Some popular examples

- Stacks & queues
  - LIFO vs. FIFO principle
- Priority queues
  - Another application of heaps
- Ordered lists
  - Linked lists where items are ordered according to a key

STACKS/1



Properties of stacks

- In a stack, insertion and deletion follow the last-in, first-out (LIFO) principle

STACKS/1

Properties of stacks

- In a stack, insertion and deletion follow the last-in, first-out (LIFO) principle
- Hence, the item that has been in the stack for the shortest time is deleted first
  - Example: elimination of recursion

STACKS / 1

Properties of stacks

- In a stack, insertion and deletion follow the last-in, first-out (LIFO) principle
- Hence, the item that has been in the stack for the shortest time is deleted first
  - Example: elimination of recursion
- Implemented by a data structure where items are
  - inserted at the end/~~beginning~~ (`push`)
  - removed from the end/~~beginning~~ (`pop`)

insert & delete at same end

STACKS/1

Properties of stacks

- In a stack, insertion and deletion follow the last-in, first-out (LIFO) principle
- Hence, the item that has been in the stack for the shortest time is deleted first
  - Example: elimination of recursion
- Implemented by a data structure where items are
  - inserted at the end/beginning (`push`)
  - removed from the end/beginning (`pop`)
- Appropriate data structures for their implementation
  - arrays (end) or singly linked lists (beginning)

We design a solution for a single stack (i.e., stack is implicit; not a parameter in operations)

STACKS/2

Array implementation

- Inserting and removing items from a stack

Algo: push(x)

```
S[t] = x;
t = t+1;
```

STACKS/2

Array implementation

- Inserting and removing items from a stack

Algo: push(x)

```
S[t] = x;
t = t+1;
```

Algo: pop()

```
t = t-1;
return S[t];
```

STACKS/2

Array implementation

- Inserting and removing items from a stack

Algo: push(x)
S[t] = x; t = t+1;

Algo: pop()
t = t-1; **return** S[t];

- The stack is implemented by an array $S[1..n]$ of length n

STACKS/2

Array implementation

- Inserting and removing items from a stack

Algo: push(x)
S[t] = x; t = t+1;

Algo: pop()
t = t-1; **return** S[t];

- The stack is implemented by an array $S[1..n]$ of length n
- The variable t functions as pointer to top of the stack
- S and t are global variables.

STACKS/2

Array implementation

- Inserting and removing items from a stack

Algo: push(x)
S[t] = x; t = t+1;

Algo: pop()
t = t-1; **return** S[t];

- The stack is implemented by an array $S[1..n]$ of length n
- The variable t functions as pointer to top of the stack
- S and t are global variables.
- Notice that if $t = 1$ then the stack is actually empty

STACKS/2

t points to first free element

Array implementation

- Inserting and removing items from a stack

Algo: push(x)
S[t] = x; t = t+1;

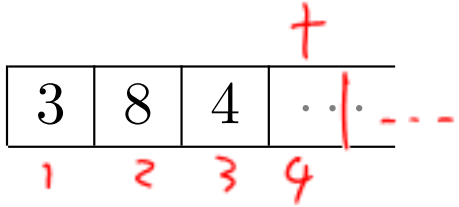
Algo: pop()
t = t-1; **return** S[t];

- The stack is implemented by an array $S[1..n]$ of length n
- The variable t functions as pointer to top of the stack
- S and t are global variables.
- Notice that if $t = 1$ then the stack is actually empty
- Error checking for under/overflow has been omitted

STACKS / 3

A simple example

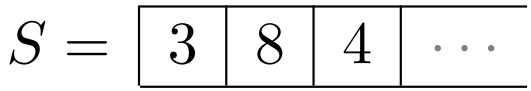
- Stack S containing 3 items ($t = 4$)

$S =$ 

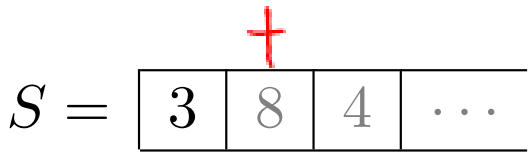
STACKS / 3

A simple example

- Stack S containing 3 items ($t = 4$)

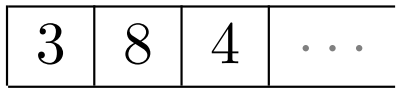


- Stack S after 2 × `pop` () ($t = 2$)



STACKS / 3

A simple example

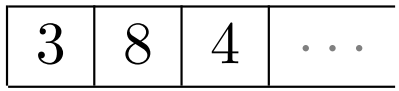
- Stack S containing 3 items ($t = 4$) $S =$ 

- Stack S after $2 \times$ `pop` () ($t = 2$) $S =$ 

- Stack S after `push` (5) ($t = 3$) $S =$ 

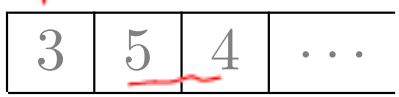
STACKS / 3

A simple example

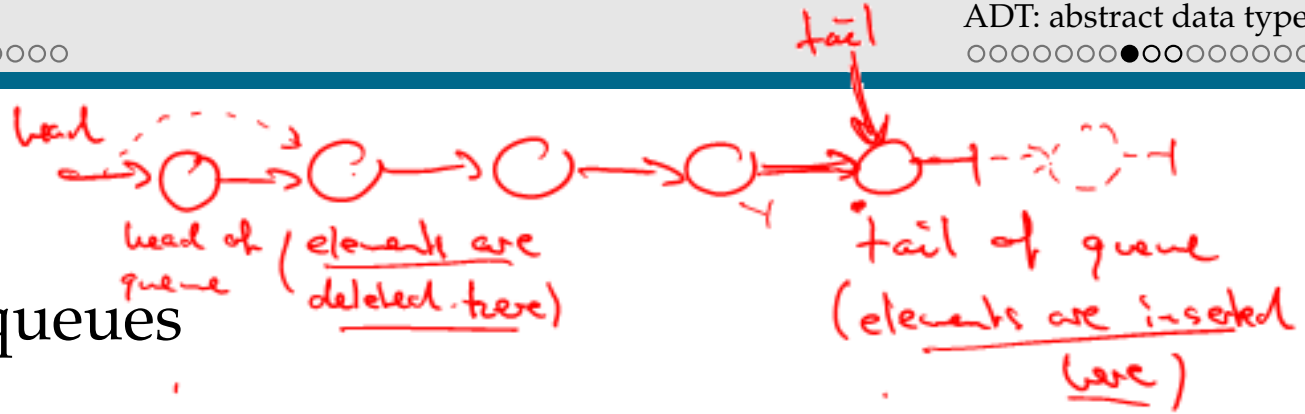
- Stack S containing 3 items ($t = 4$) $S =$ 

- Stack S after 2 × `pop` () ($t = 2$) $S =$ 

- Stack S after `push` (5) ($t = 3$) $S =$ 

- Stack S after 2 × `pop` () ($t = 1$) $S =$ 

QUEUES/1



Properties of queues

- In a queue, insertion and deletion follow the first-in, first-out (FIFO) principle

QUEUES/1

Properties of queues

- In a queue, insertion and deletion follow the first-in, first-out (FIFO) principle
- Hence, the item that has been in the queue for the longest time is deleted first
  - Example: managing jobs of a printer

QUEUES/1

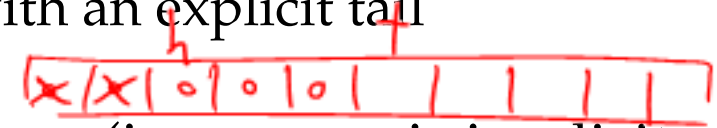
Properties of queues

- In a queue, insertion and deletion follow the first-in, first-out (FIFO) principle
- Hence, the item that has been in the queue for the longest time is deleted first
  - Example: managing jobs of a printer
- Implemented by a data structure where items are
  - inserted at the end/~~beginning~~ (`enqueue`)
  - removed from the beginning/~~end~~ (`dequeue`)

QUEUES/1

Properties of queues

- In a queue, insertion and deletion follow the first-in, first-out (FIFO) principle
- Hence, the item that has been in the queue for the longest time is deleted first
  - Example: managing jobs of a printer
- Implemented by a data structure where items are
  - inserted at the end/beginning (`enqueue`)
  - removed from the beginning/end (`dequeue`)
- Appropriate data structures for their implementation
  - arrays or singly linked lists with an explicit tail



We design a solution for a single queue (i.e., queue is implicit; not a parameter in operations)

QUEUES/2

Array implementation

- Inserting and removing items from a queue

Algo: enqueue(x)

```
Q[t] = x;
t = t+1 mod n;
```

QUEUES/2

Array implementation

- Inserting and removing items from a queue

Algo: enqueue(x)

```
Q[t] = x;
t = t+1 mod n;
```

Algo: dequeue()

```
i = h;
h = h+1 mod n;
return Q[i];
```

QUEUES/2

Array implementation

- Inserting and removing items from a queue

Algo: enqueue(x)

```
Q[t] = x;
t = t+1 mod n;
```

Algo: dequeue()

```
i = h;
h = h+1 mod n;
return Q[i];
```

- The queue is implemented by an array $Q[0..n-1]$ of length n used in circular fashion

QUEUES/2

Array implementation

- Inserting and removing items from a queue

Algo: enqueue(x)

```
Q[t] = x;
t = t+1 mod n;
```

Algo: dequeue()

```
i = h;
h = h+1 mod n;
return Q[i];
```

- The queue is implemented by an array $Q[0..n-1]$ of length n used in circular fashion
- Variables h and t function as pointers to head and tail of the queue

QUEUES/2

Array implementation

- Inserting and removing items from a queue

Algo: enqueue(x)

```
Q[t] = x;
t = t+1 mod n;
```

Algo: dequeue()

```
i = h;
h = h+1 mod n;
return Q[i];
```

- The queue is implemented by an array $Q[0..n-1]$ of length n used in circular fashion
- Variables h and t function as pointers to head and tail of the queue
- Notice that if $h = t$ then the queue is actually empty

QUEUES/3



A simple example

- Queue Q containing 3 items ($h = n - 2, t = 1$)

$$Q = \boxed{4}\ \boxed{0}\ \boxed{\cdots}\ \boxed{3}\ \boxed{8}$$

t h

QUEUES/3

A simple example

- Queue Q containing 3 items ($h = n - 2, t = 1$)

$$Q = \boxed{4}\boxed{0}\boxed{\cdots}\boxed{3}\boxed{8}$$

- Queue Q after $2 \times$ `dequeue` () ($h = 0, t = 1$)

$$Q = \boxed{4}\boxed{0}\boxed{\cdots}\boxed{3}\boxed{8}$$

h t

QUEUES/3

A simple example

- Queue Q containing 3 items ($h = n - 2, t = 1$)

$$Q = \boxed{4}\boxed{0}\boxed{\cdots}\boxed{3}\boxed{8}$$

- Queue Q after $2 \times$ `dequeue` () ($h = 0, t = 1$)

$$Q = \boxed{4}\boxed{0}\boxed{\cdots}\boxed{3}\boxed{8}$$

- Queue Q after `enqueue` (5) ($h = 0, t = 2$)

$$Q = \boxed{4}\boxed{5}\boxed{\cdots}\boxed{3}\boxed{8}$$

h t

QUEUES/3

A simple example

- Queue Q containing 3 items ($h = n - 2, t = 1$)

$$Q = \boxed{4}\boxed{0}\boxed{\cdots}\boxed{3}\boxed{8}$$

- Queue Q after $2 \times$ `dequeue` () ($h = 0, t = 1$)

$$Q = \boxed{4}\boxed{0}\boxed{\cdots}\boxed{3}\boxed{8}$$

- Queue Q after `enqueue` (5) ($h = 0, t = 2$)

$$Q = \boxed{4}\boxed{5}\boxed{\cdots}\boxed{3}\boxed{8}$$

- Queue Q after $2 \times$ `dequeue` () ($h = t = 2$)

$$Q = \boxed{4}\boxed{5}\boxed{\cdots}\boxed{3}\boxed{8}$$

ORDERED LISTS/1

- In an ordered list elemenst are ordered according to a key.
- We design a solution for multiple ordered queues (i.e., ordered queue is an explicit parameter in operations).

- Example functions on ordered list:
  - `struct node** init()`
  - `bool isEmpty(struct node** l)`
  - `int first(struct node** l)`
  - `int last(struct node** l)`
  - `void print(struct node** l)`
  - `void insert(struct node** l, int x)`

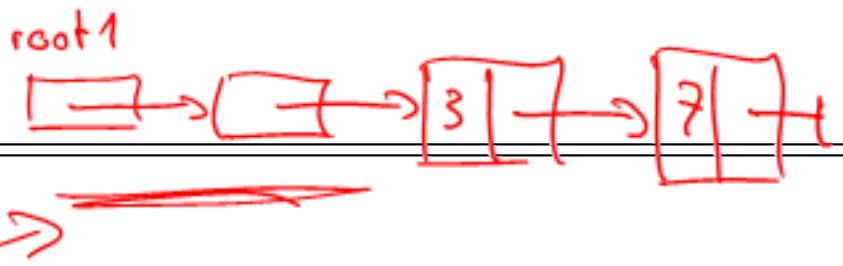
ORDERED LISTS/2

Definition of data structure in library (we do not define a global instance variable root):

```
struct node {
  int val;
  struct node* next;
};
```

Definition of instances of ordered lists in application programs:

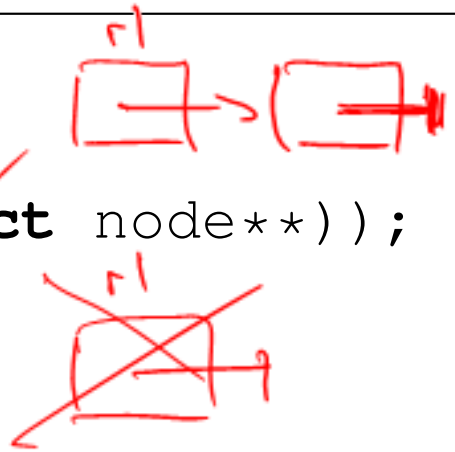
```
struct node* root1;
struct node* root2;
```



ORDERED LISTS/3

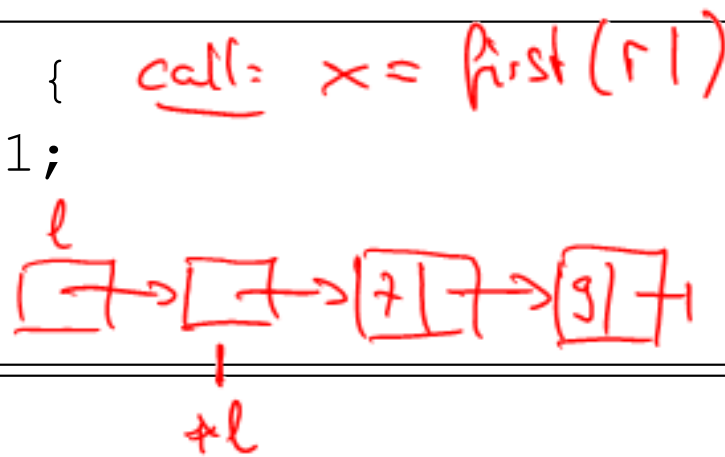
Initialization of an ordered list:

```
struct node** init() {
  struct node **l;
  l = malloc(sizeof(struct node**));
  *l = NULL;
  return l;
}
```



Retrieving the 1st element of an ordered list (-1 if list is empty):

```
int first(struct node** l) {
  if (*l == NULL) return -1;
  else return (*l)->val;
}
```



ORDERED LISTS/4

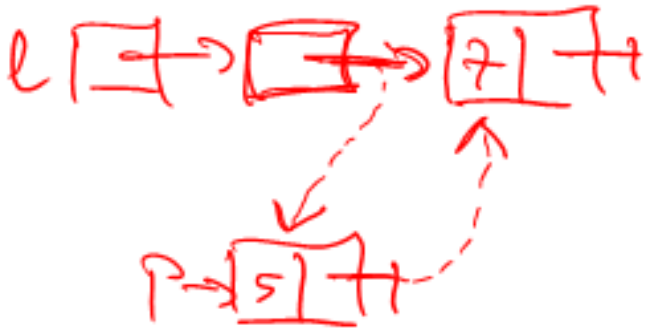
call: insert(&l, 5)

Insertion into an ordered list:

```
void insert(struct node** l, int x) {
  struct node* p;
  struct node* q;

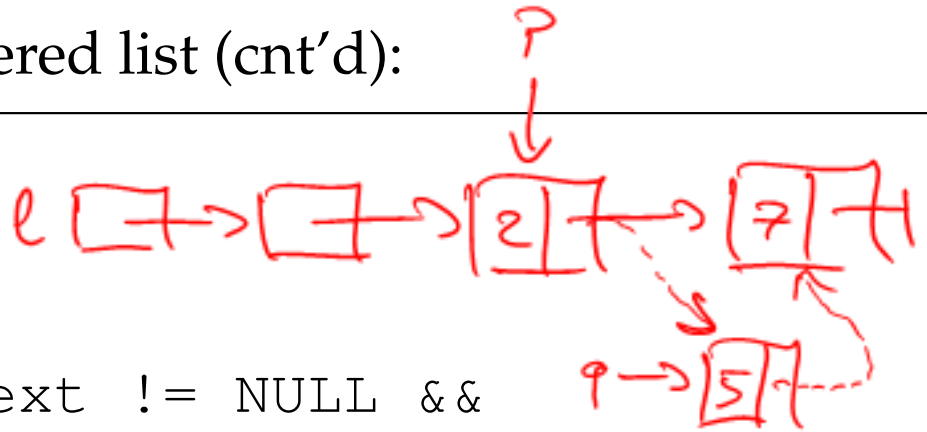
  if (*l == NULL || (*l)->val > x) {
    p = malloc(sizeof(struct node));
    p->val = x;
    p->next = *l;
    *l = p;
  } else {

...
```



ORDERED LISTS/5

Insertion into an ordered list (cnt'd):



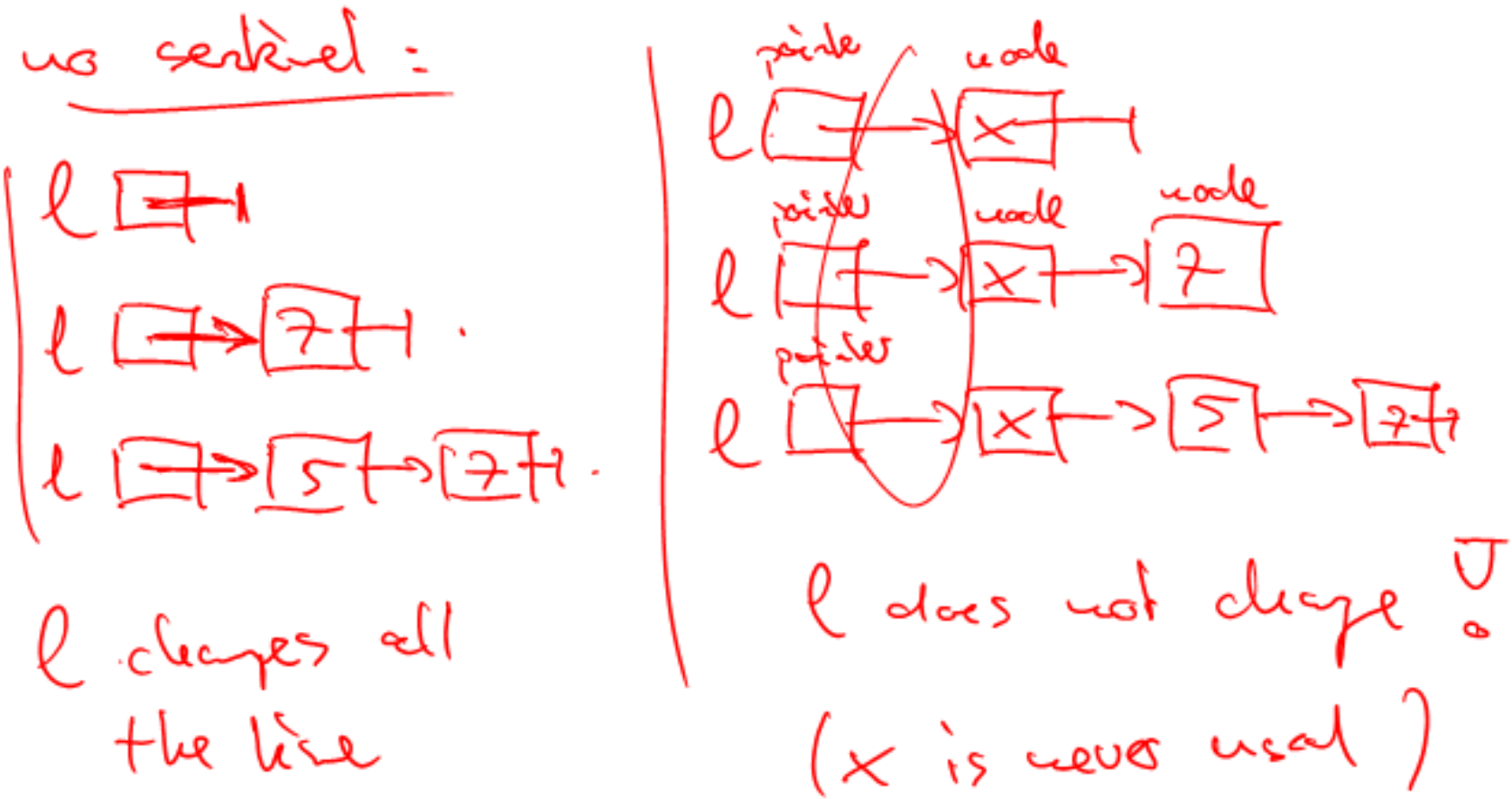

```
...
    p = *l;
    while (p->next != NULL &&
           p->next->val < x) p = p->next;
    q = malloc(sizeof(struct node));
    q->val = x;
    q->next = p->next;
    p->next = q;
  }
}
```

```
void f(int i) {
  i++
}
```

EXERCISE SL05-7/1

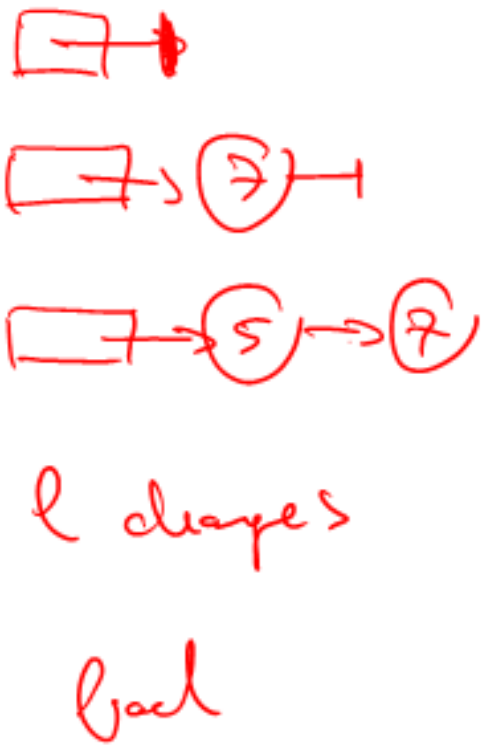
insert(l, 7)

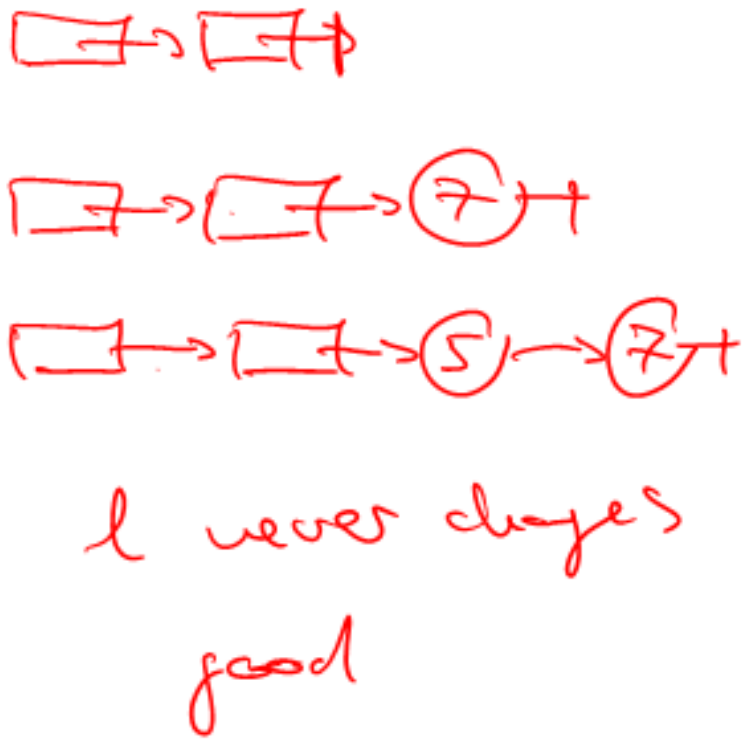
A first possibility to systematically deal with a changing root of a dynamic data structure is to use a **sentinel** (dummy element). Illustrate a sentinel for a list with zero, one and two elements.



EXERCISE SL05-7/2

A second possibility to systematically deal with a changing root of a dynamic data structure is to use a **a pointer to a pointer** at the beginning of the list. Illustrate a list with a pointer to a pointer for zero, one and two elements.





EXERCISE SL05-7/3

A third possibility to systematically deal with a changing root of a dynamic data structure is to convert all procedures that possibly modify the root into **functions that return the modified object**. Illustrate the changes (signature and call of insert operation).

a) modified object is not returned (bad because changes are lost)

```
void insert(struct node* p; int x) { .... p = ....; }
call: insert(p, 7)
```

b) modified object is returned (good because changes are not lost)

```
struct node* insert(struct node* p; int x) { ... p = ....;
                                              return p; }
call: l = insert(l, 7)
```

EXERCISE SL05-8/1

Write a C procedure that can be used to increment an integer value that is passed as a parameter.

```
int x;
void inc(int* i) { *i = *i + 1; }
call: x=7; inc(&x);
```

- normal fct call: pass a copy of object (⇒ modification to objects are lost)
- pass a reference (URL) to an object (⇒ modification remain; reference (URL) cannot be changed)
- pass a reference to a reference to an object (object and the reference can be changed)

SUMMARY

- Dynamic data structures are essential since they allow application to increase and decrease the memory space they use.
- The basic mechanism to implement dynamic data structures are pointers and the allocation and deallocation of memory.
- Memory allocation and deallocation is expensive.
- The most important dynamic data structures are lists, trees, stacks, and queues.
- Abstract data types encapsulate data structures and algorithms that operate on these data structures.
- They are the foundation of classes with methods.

Informatik II

Trees, red-black trees

— SL06 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

DICTIONARIES / 1

Dynamic data structures

- A dictionary D is a dynamic data structure with the following operations (items have key and data part)
  - `search` (D, v): searches for v in D and, eventually, returns a pointer x such that $x.key = v$ equality search
  - `delete` (D, x): deletes item pointed to by x from D
  - `insert` (D, x): inserts item pointed to by x into D

DICTIONARIES/1

hash tables support this

Dynamic data structures

- A dictionary D is a dynamic data structure with the following operations (items have key and data part)
  - `search` (D, v): searches for v in D and, eventually, returns a pointer x such that $x.key = v$
  - `delete` (D, x): deletes item pointed to by x from D
  - `insert` (D, x): inserts item pointed to by x into D
- Furthermore, one may want to support the following operations (which require that keys are comparable)
  - `minimum` (D) and `maximum` (D)
  - `successor` (D, v) and `predecessor` (D, v)

search trees support this

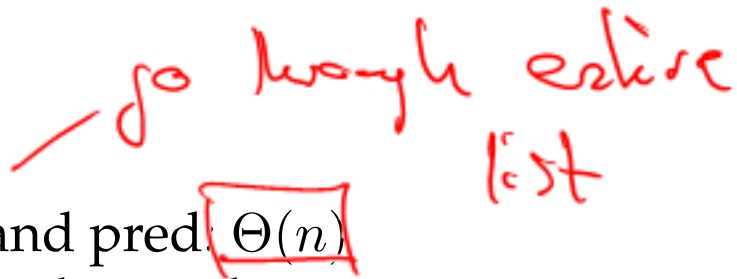
requires ordering

not usually used for phone #, student IDs

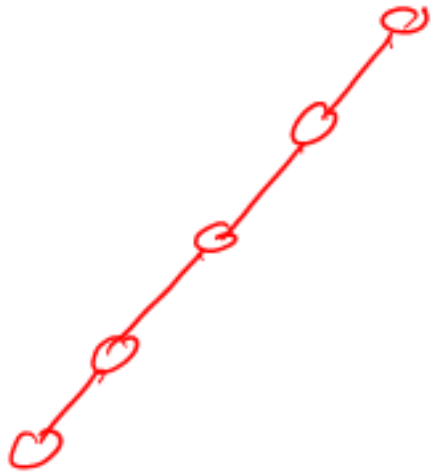
DICTIONARIES / 2

Possible implementations

- Singly linked lists
  - Insert and delete: $\Theta(1)$
  - Search, min, max, succ, and pred: $\Theta(n)$
  - Ordered lists do not provide a real improvement



DICTIONARIES / 2



Possible implementations

- Singly linked lists
  - Insert and delete: $\Theta(1)$
  - Search, min, max, succ, and pred: $\Theta(n)$
  - Ordered lists do not provide a real improvement
- Binary search trees
  - Binary search: $\Theta(\lg n)$
  - Represent search space as a binary tree
  - Insert & delete may yield unbalanced search trees

no guarantee
worst case $\Theta(n)$

DICTIONARIES / 2

Possible implementations

- Singly linked lists
  - Insert and delete: $\Theta(1)$
  - Search, min, max, succ, and pred: $\Theta(n)$
  - Ordered lists do not provide a real improvement
- Binary search trees
  - Binary search: $\Theta(\lg n)$
  - Represent search space as a binary tree
  - Insert & delete may yield unbalanced search trees
- Balanced binary trees
  - Cost of operations: $\Theta(\lg n)$ guarantee, in worst case
  - Red-black trees as device to keep trees balanced

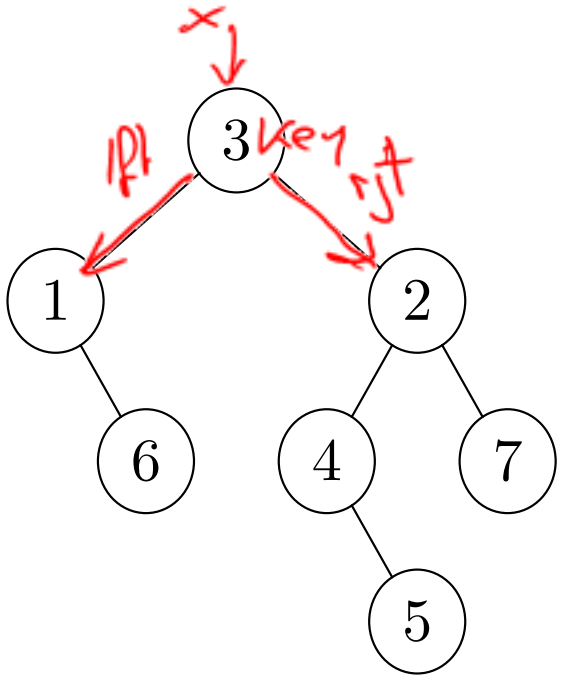
TABLE OF CONTENTS — SL06

1. Binary trees
   - Binary trees
   - Trees
   - Binary search trees
   - Inserting items
   - Deleting items

2. Red-black trees

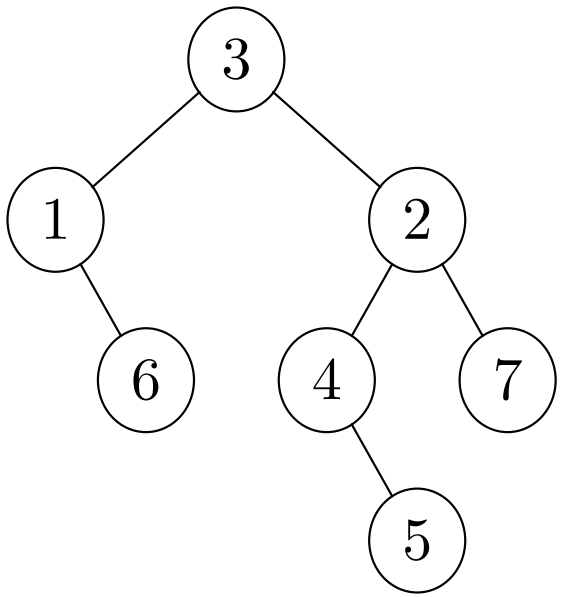
BINARY TREES/1

Linked data structure



BINARY TREES/1

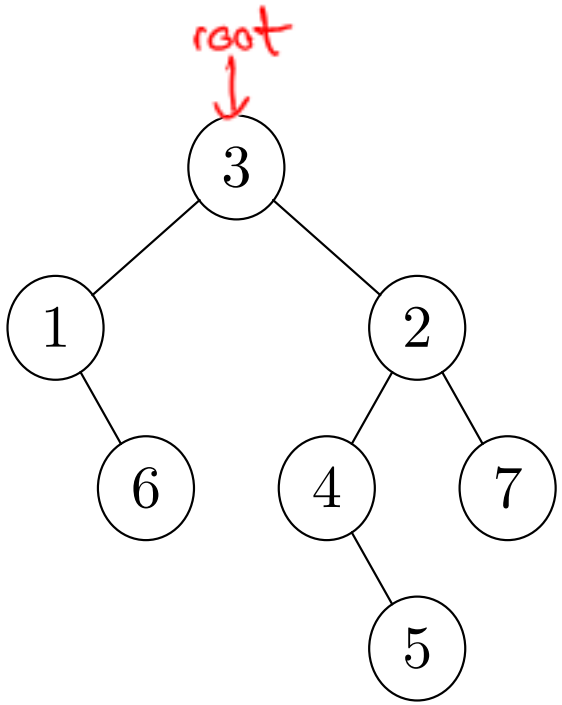
Linked data structure



- For each node x there is (we assume that a node is accessed through a pointer to the node)
  - $x \rightarrow key$: value of the key (usually an integer value; not necessarily unique)
  - $x \rightarrow left$: pointer to left child
  - $x \rightarrow rgt$: pointer to right child

BINARY TREES/1

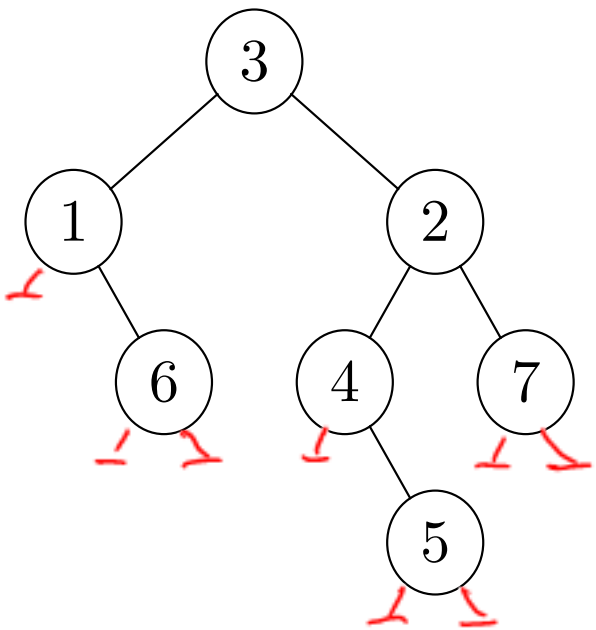
Linked data structure



- For each node x there is (we assume that a node is accessed through a pointer to the node)
  - $x{\rightarrow}key$: value of the key (usually an integer value; not necessarily unique)
  - $x{\rightarrow}lft$: pointer to left child
  - $x{\rightarrow}rgt$: pointer to right child
- $root$ points to the root of the tree

BINARY TREES/1

Linked data structure



- For each node x there is (we assume that a node is accessed through a pointer to the node)
  - $x{\rightarrow}key$: value of the key (usually an integer value; not necessarily unique)
  - $x{\rightarrow}lft$: pointer to left child
  - $x{\rightarrow}rgt$: pointer to right child
- $root$ points to the root of the tree
- For a leaf node p we have $p{\rightarrow}lft = p{\rightarrow}rgt = \texttt{NIL}$

BINARY TREES/2

Inorder tree walk

- Divide-and-conquer algorithm that visits all the nodes of a binary tree

BINARY TREES/2

Inorder tree walk

- Divide-and-conquer algorithm that visits all the nodes of a binary tree
- Running time of `InorderTreeWalk` is linear ($\Theta(n)$)

BINARY TREES/2

Inorder tree walk

- Divide-and-conquer algorithm that visits all the nodes of a binary tree
- Running time of `InorderTreeWalk` is linear ($\Theta(n)$)

The algorithm

Algo: InorderTreeWalk(p)

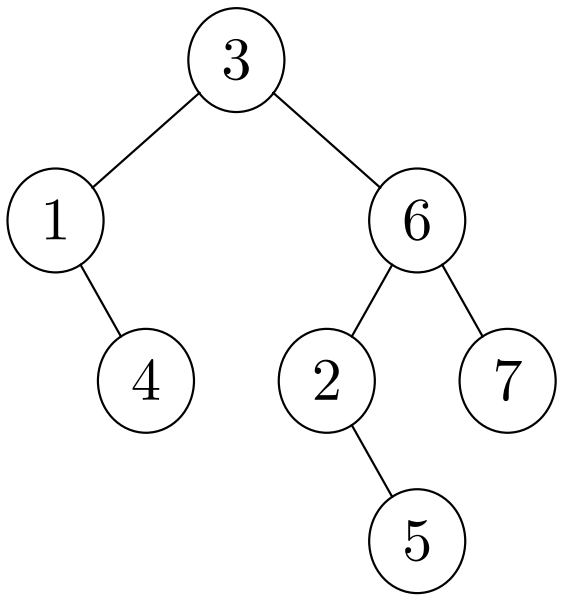
---

if $p \neq NIL$ **then**
    InorderTreeWalk(p->lft);
    VisitNode(p);
    InorderTreeWalk(p->rgt);

first go left
then go middle
then go right

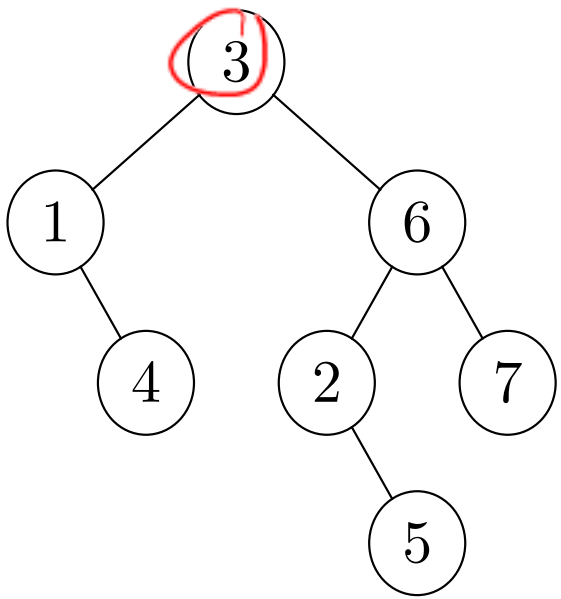
BINARY TREES/3

Example tree walk/1



BINARY TREES/3

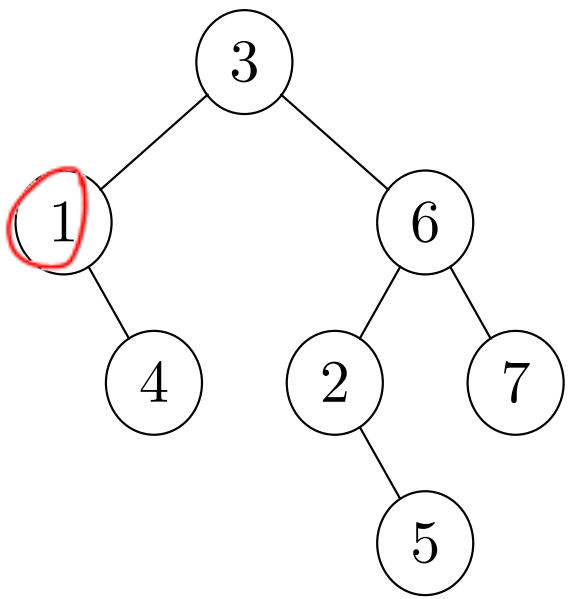
Example tree walk/1



`InorderTreeWalk` (3)

BINARY TREES/3

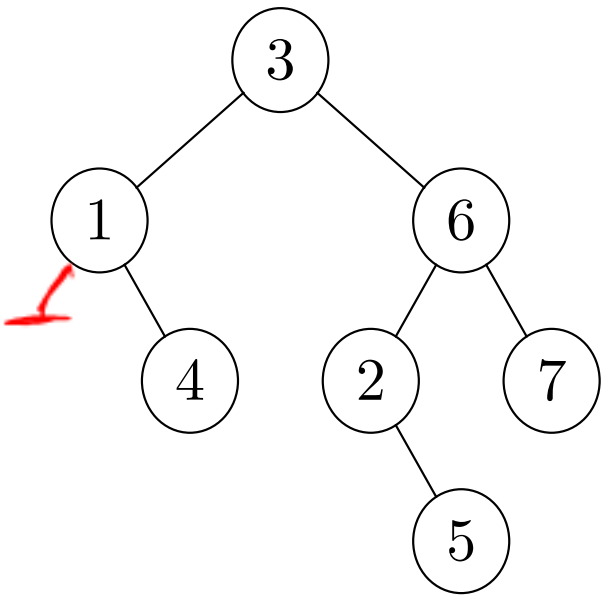
Example tree walk/1




```
InorderTreeWalk (3)
  InorderTreeWalk (1)
```

BINARY TREES/3

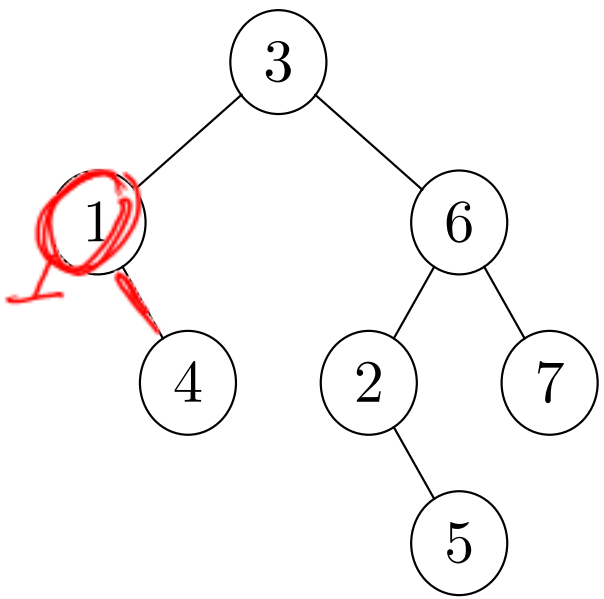
Example tree walk/1




```
InorderTreeWalk(3)
  InorderTreeWalk(1)
    InorderTreeWalk(NIL)
```

BINARY TREES/3

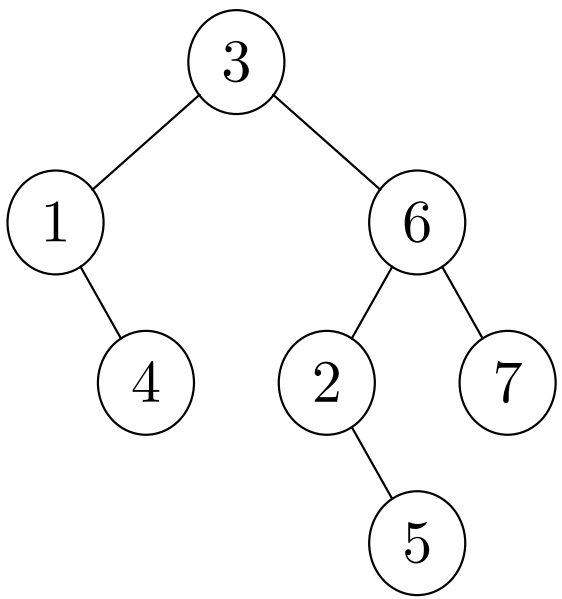
Example tree walk/1




```
InorderTreeWalk (3)
  InorderTreeWalk (1)
    InorderTreeWalk (NIL)
    VisitNode (1) — print 1
```

BINARY TREES/3

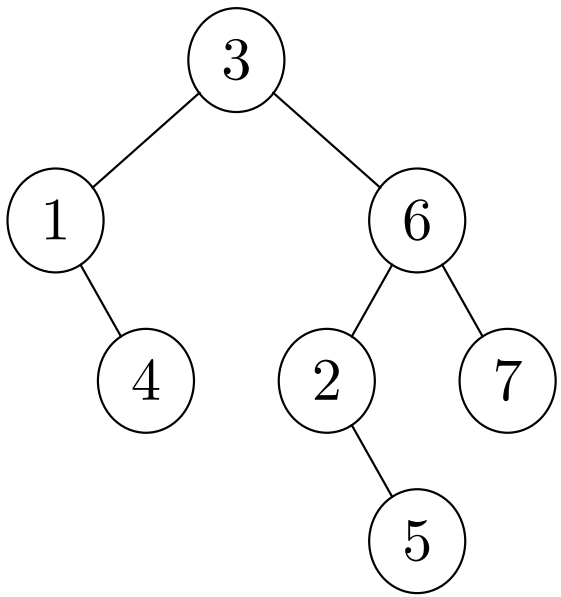
Example tree walk/1



`InorderTreeWalk` (3)
  `InorderTreeWalk` (1)
    `InorderTreeWalk` (`NIL`)
    `VisitNode` (1) — print 1
    `InorderTreeWalk` (4)

BINARY TREES/3

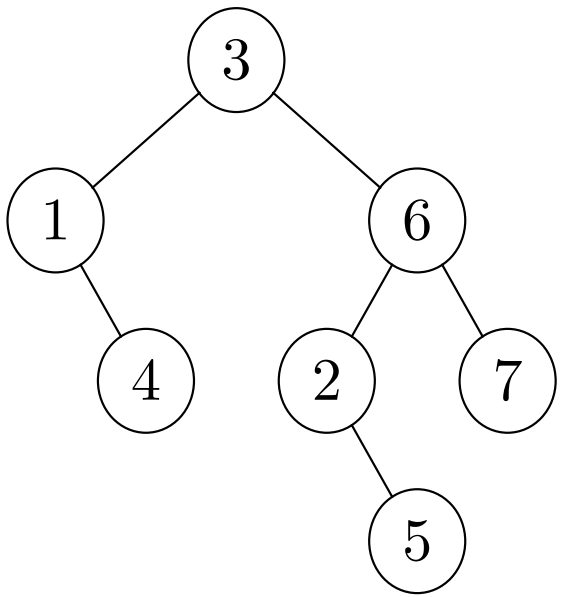
Example tree walk/1




```
InorderTreeWalk (3)
  InorderTreeWalk (1)
    InorderTreeWalk (NIL)
    VisitNode (1) — print 1
    InorderTreeWalk (4)
      InorderTreeWalk (NIL)
```

BINARY TREES/3

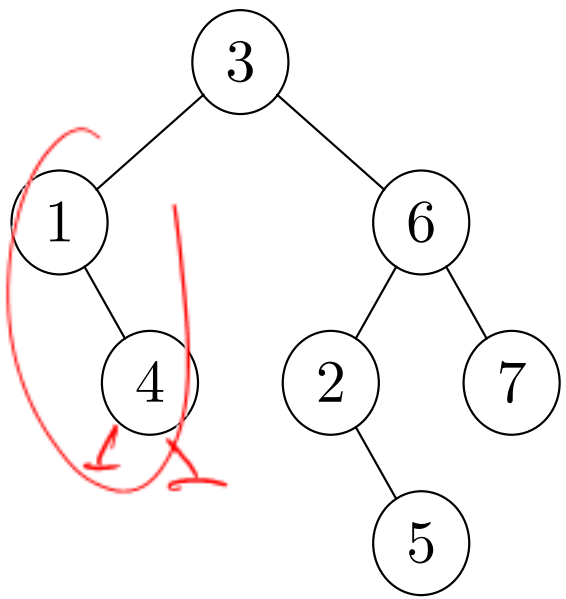
Example tree walk/1




```
InorderTreeWalk (3)
  InorderTreeWalk (1)
    InorderTreeWalk (NIL)
    VisitNode (1) — print 1
    InorderTreeWalk (4)
      InorderTreeWalk (NIL)
      VisitNode (4) — print 4
```

BINARY TREES/3

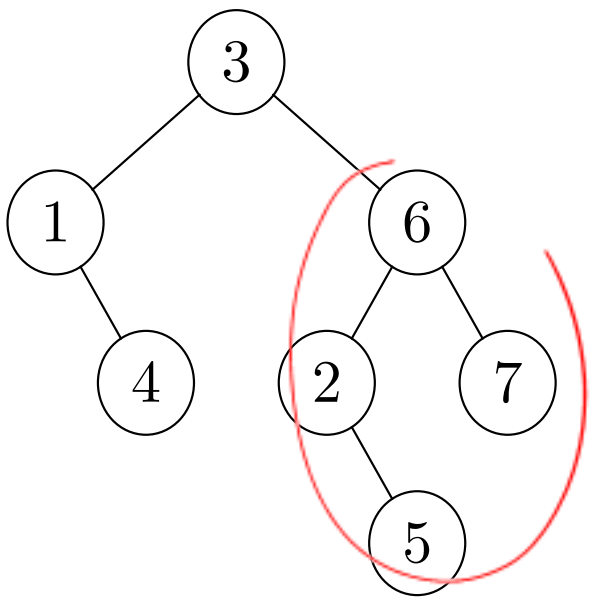
Example tree walk/1




```
InorderTreeWalk (3)
  InorderTreeWalk (1)
    InorderTreeWalk (NIL)
    VisitNode (1) — print 1
    InorderTreeWalk (4)
      InorderTreeWalk (NIL)
      VisitNode (4) — print 4
      InorderTreeWalk (NIL)
```

BINARY TREES/3

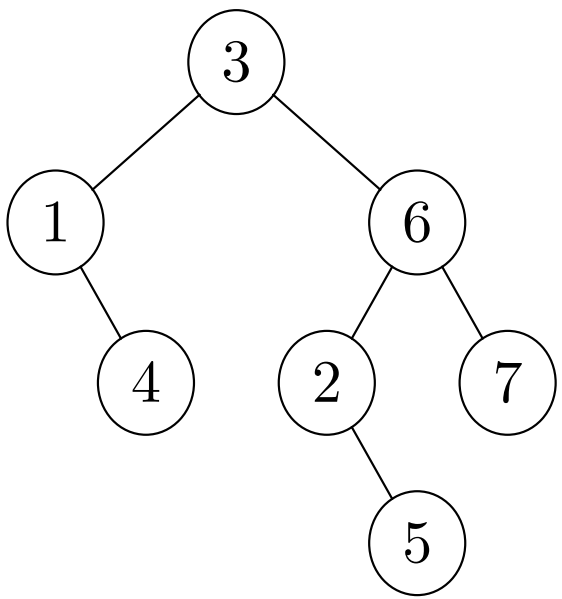
Example tree walk/1




```
InorderTreeWalk (3)
  InorderTreeWalk (1)
    InorderTreeWalk (NIL)
    VisitNode (1) — print 1
    InorderTreeWalk (4)
      InorderTreeWalk (NIL)
      VisitNode (4) — print 4
      InorderTreeWalk (NIL)
  VisitNode (3) — print 3
```

BINARY TREES/3

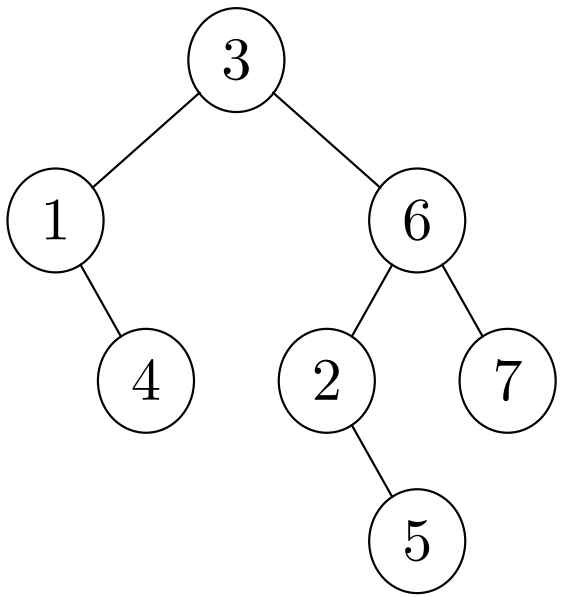
Example tree walk/1




```
InorderTreeWalk (3)
  InorderTreeWalk (1)
    InorderTreeWalk (NIL)
    VisitNode (1) — print 1
    InorderTreeWalk (4)
      InorderTreeWalk (NIL)
      VisitNode (4) — print 4
      InorderTreeWalk (NIL)
  VisitNode (3) — print 3
  InorderTreeWalk (6)
```

BINARY TREES/3

Example tree walk/1

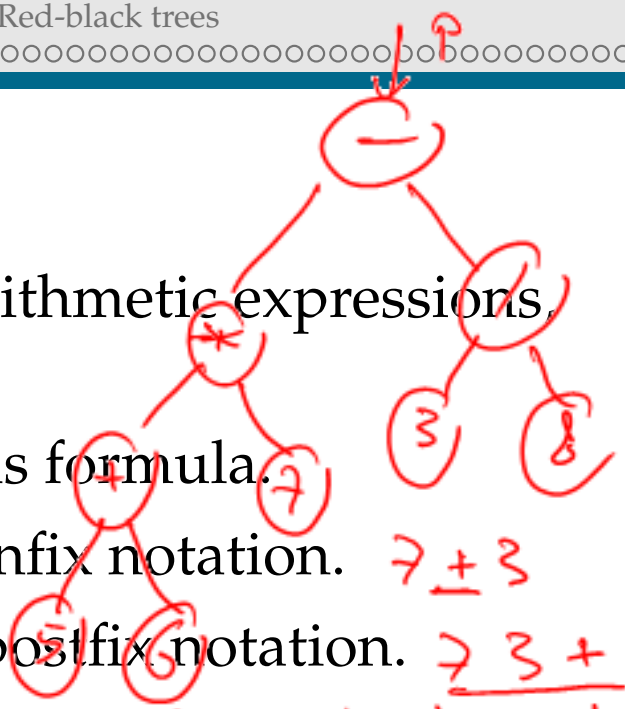



```
InorderTreeWalk (3)
  InorderTreeWalk (1)
    InorderTreeWalk (NIL)
    VisitNode (1) — print 1
    InorderTreeWalk (4)
      InorderTreeWalk (NIL)
      VisitNode (4) — print 4
      InorderTreeWalk (NIL)
  VisitNode (3) — print 3
  InorderTreeWalk (6)
    [walk through right subtree]
```

EXERCISE SL06-1

Assume a tree that is used to represent arithmetic expressions, such as $(5 + 6) * 7 - 3/8$

- Show the tree that corresponds to this formula.
- Given a tree print the expression in infix notation.
- Given a tree print the expression in postfix notation.



```
infix(struct node* p)
  if (p ≠ NULL) {
    printf("(")
    infix(p->lft)
    printf("%c", p->sym)
    infix(p->rgt)
    printf(")")
```

```
postfix(struct node* p)
  if (p ≠ NULL)
    postfix(p->lft)
    postfix(p->right)
    printf("%c", p->sym)
```

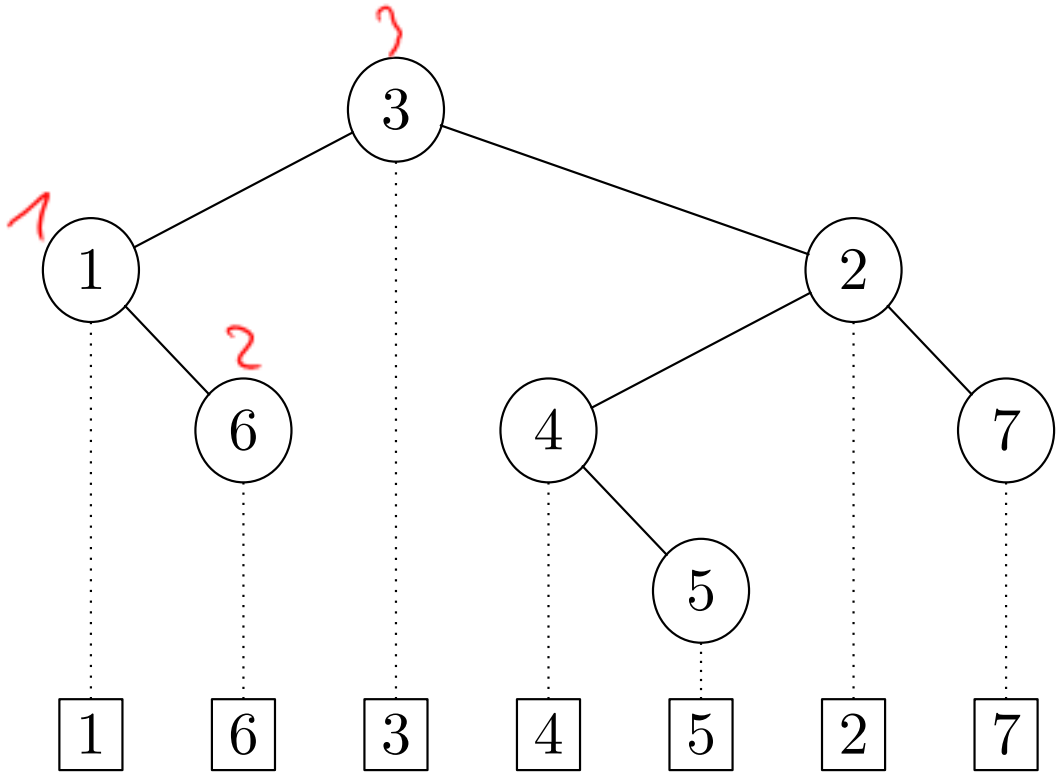
```
struct node {
  char sym;
  struct node* lft;
  struct node* rgt;}
```

(((5) + (6)) * (7)) - ((3) / (8))

5 6 + 7 * 3 8 / -

BINARY TREES/4

Example tree walk/2



BINARY TREES/5

- There are alternative ways to walk through a tree
  - Preorder: visit node before moving to the children
  - Postorder: move to the children before visiting node
  - Simply change position of `VisitNode` (p) in the code
- Inorder tree walk can be seen as a projection of the tree nodes onto a one-dimensional interval

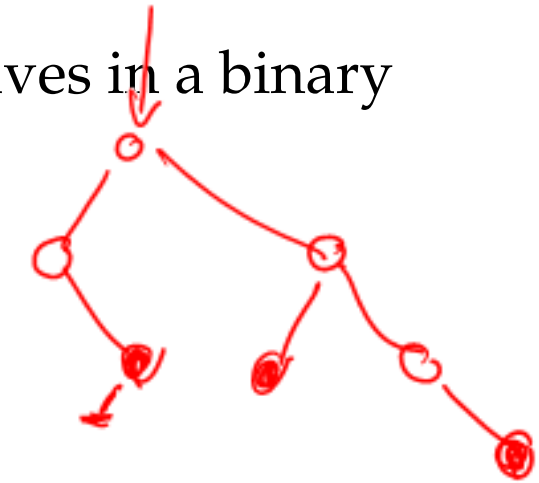
BINARY TREES/5

- There are alternative ways to walk through a tree
  - Preorder: visit node before moving to the children
  - Postorder: move to the children before visiting node
  - Simply change position of `VisitNode` (p) in the code
- Inorder tree walk can be seen as a projection of the tree nodes onto a one-dimensional interval

- A **binary search tree** is a binary tree where the nodes are ordered in a specific way.
- Binary search tree (and its variants) are used heavily since they allow to quickly find values in a large set of values.

EXERCISE SL06-2

Write a function that counts the number of leaves in a binary tree.



```
cntLeaves(struct node * p)
  if (p == NULL) return 0
  else if (p->lft == p->rgh) return 1
  else return cntLeaves(p->lft) + cntLeaves(p->rgt)
```

condition for a leaf: p->lft = NULL, p->rgt = NULL
(or p->lft = p->rgh)

BINARY SEARCH TREES/1

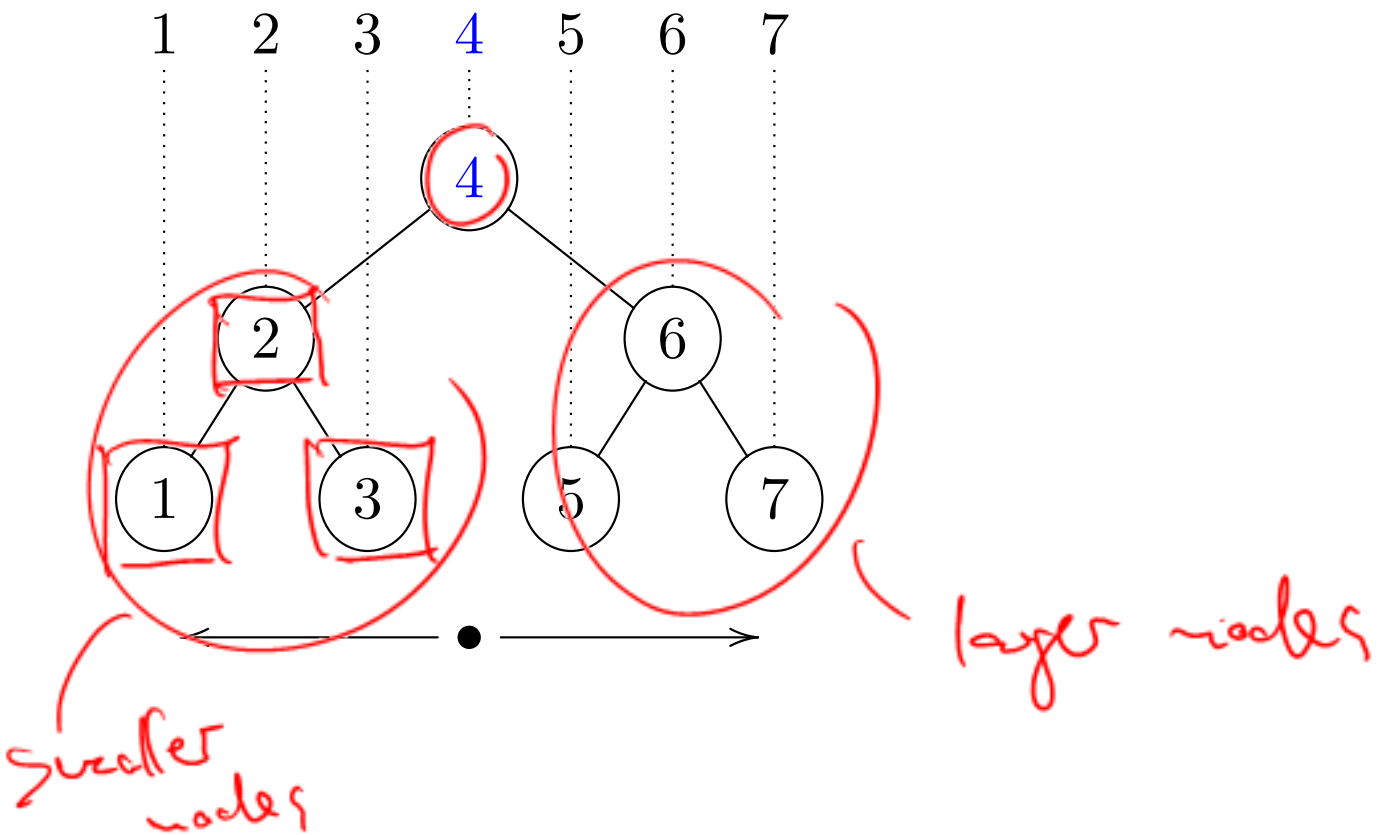
Binary search revisited

- Searching for 2 in $[1, 2, 3, 4, 5, 6, 7]$

BINARY SEARCH TREES/1

Binary search revisited

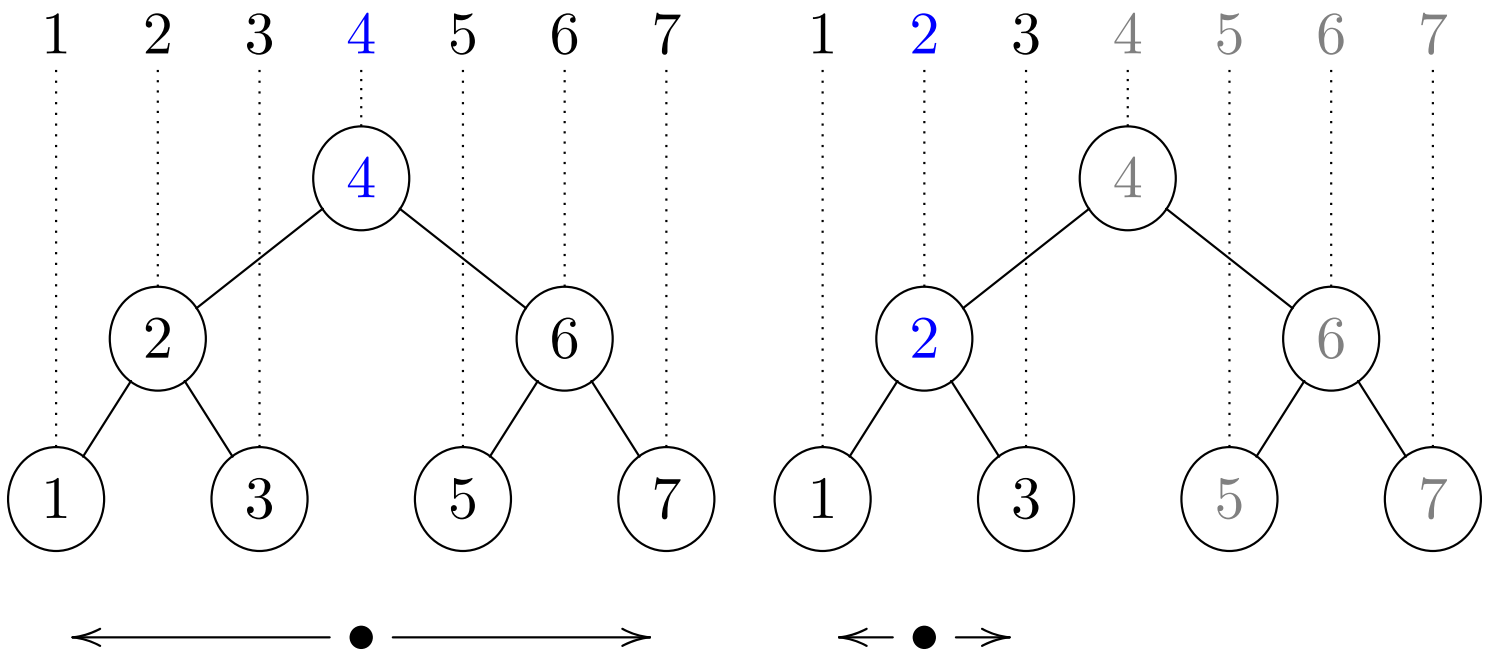
- Searching for 2 in $[1, 2, 3, 4, 5, 6, 7]$



BINARY SEARCH TREES/1

Binary search revisited

- Searching for 2 in $[1, 2, 3, 4, 5, 6, 7]$



BINARY SEARCH TREES/2

Formalizing the intuition

- Binary search tree property
  - Let x be a node in a binary tree
  - If y is a node in the left subtree of x then $y.key \leq x \rightarrow key$
  - If y is a node in the right subtree of x then $x \rightarrow key \leq y.key$

BINARY SEARCH TREES/2

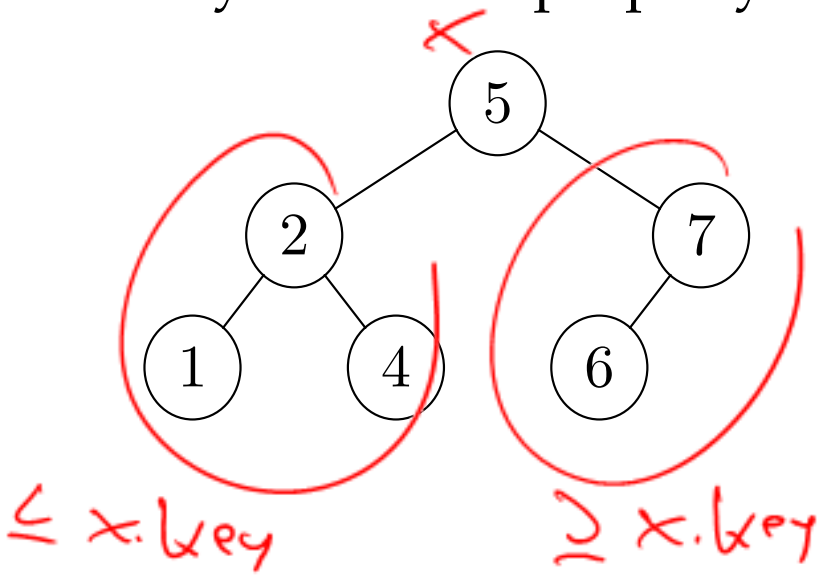
Formalizing the intuition

- Binary search tree property
  - Let x be a node in a binary tree
  - If y is a node in the left subtree of x then $y.key \leq x{\rightarrow}key$
  - If y is a node in the right subtree of x then $x{\rightarrow}key \leq y.key$
- The keys of a binary search tree are stored such that they satisfy the above property

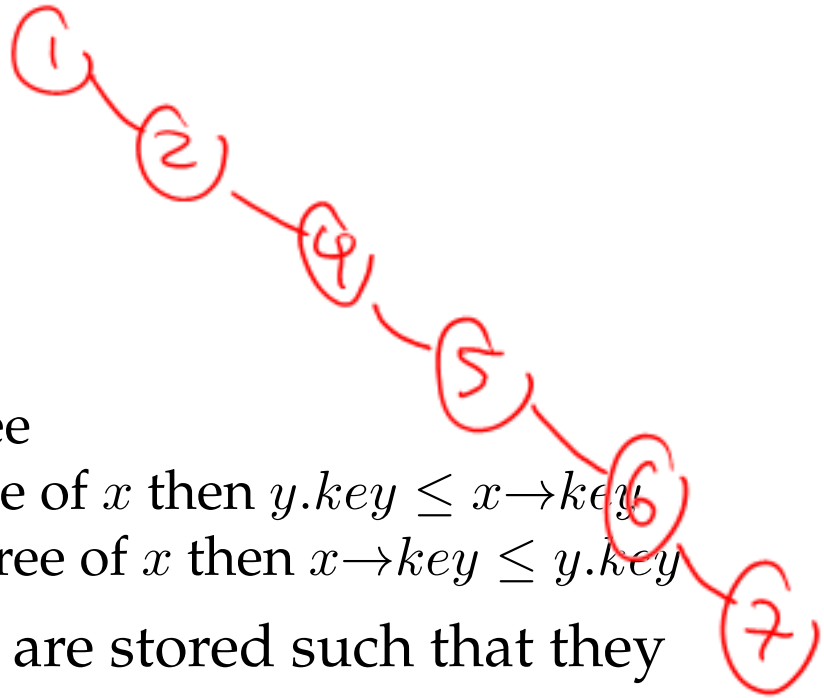
BINARY SEARCH TREES/2

Formalizing the intuition

- Binary search tree property
  - Let x be a node in a binary tree
  - If y is a node in the left subtree of x then $y.key \leq x{\rightarrow}key$
  - If y is a node in the right subtree of x then $x{\rightarrow}key \leq y.key$
- The keys of a binary search tree are stored such that they satisfy the above property

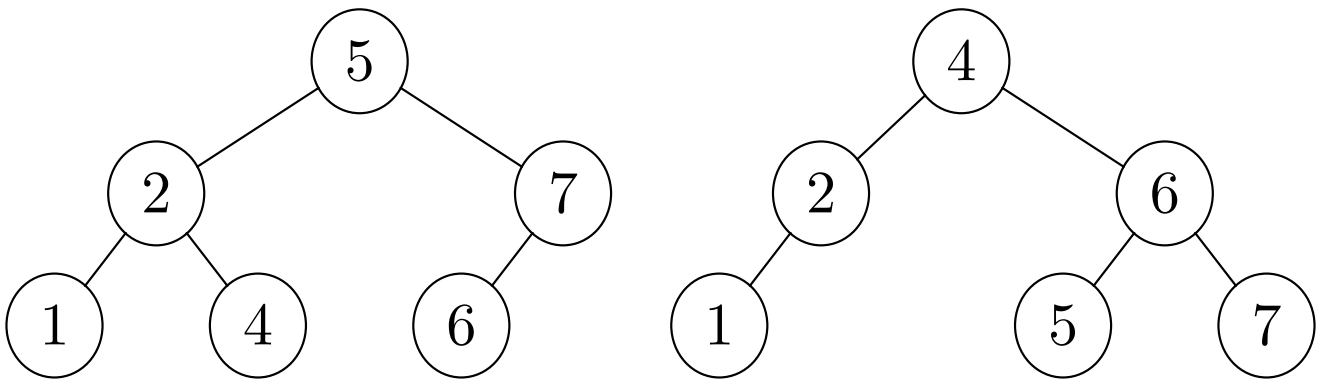


BINARY SEARCH TREES/2



Formalizing the intuition

- Binary search tree property
  - Let x be a node in a binary tree
  - If y is a node in the left subtree of x then $y.key \leq x \rightarrow key$
  - If y is a node in the right subtree of x then $x \rightarrow key \leq y.key$
- The keys of a binary search tree are stored such that they satisfy the above property



EXERCISE SL06-3

What is the difference between a binary search tree and a min heap? Which data structure can be used to print the nodes of a tree with n nodes in sorted order in $O(n)$ time?

BST: ∀ nodes x in BST:
- keys in lft subtree of x are ≤ than x->key
- —"— rgt —"— —"— ≥ —"— —"—

MinHeap: ∀ nodes x in MinHeap
- keys in lft subtree of x are ≥ than x->key
- —"— rgt —"— —"— ≥ —"— —"—

print in sorted order: BST, inorder traversal $\Theta(n)$
Heap, n times heapify $\Theta(n \log n)$

BINARY SEARCH TREES / 3

Searching for an item

- To find an item with key $= v$ in a tree rooted at p
  - check whether $p ==$ `NIL` or $p{\rightarrow}key = v$
  - if this is not the case then
    - if $v < p{\rightarrow}key$, search for v in $p{\rightarrow}lft$
    - otherwise, search for v in $p{\rightarrow}rgt$

BINARY SEARCH TREES / 3

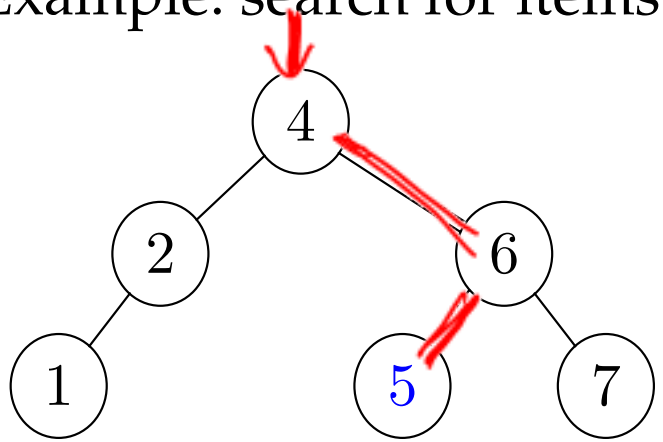
Searching for an item

- To find an item with key $= v$ in a tree rooted at p
  - check whether $p ==$ `NIL` or $p{\rightarrow}key = v$
  - if this is not the case then
    - if $v < p{\rightarrow}key$, search for v in $p{\rightarrow}lft$
    - otherwise, search for v in $p{\rightarrow}rgt$
- Example: search for items with key $= 5$ and key $= 3$

BINARY SEARCH TREES / 3

Searching for an item

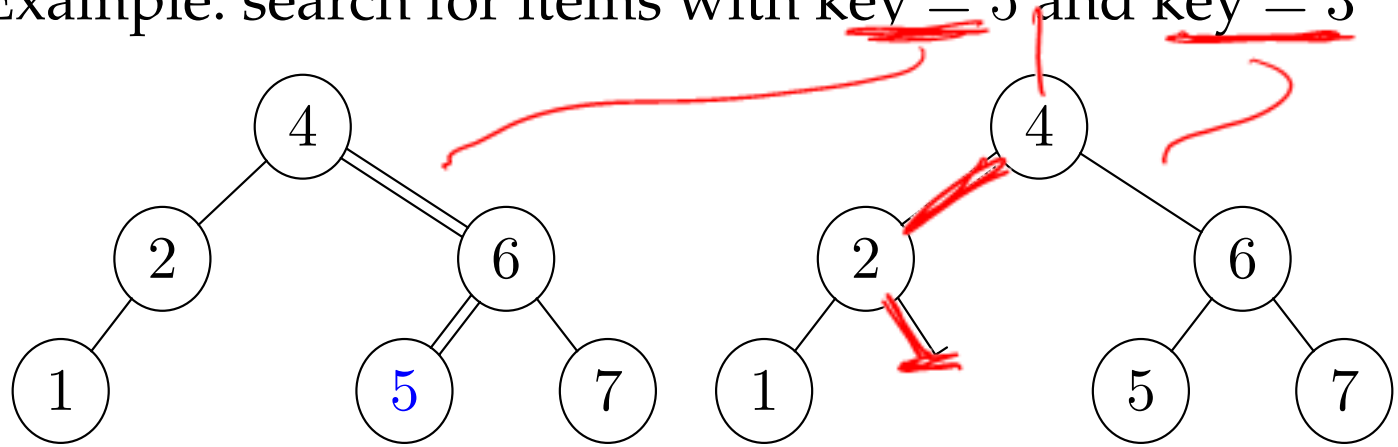
- To find an item with key $= v$ in a tree rooted at p
  - check whether $p ==$ `NIL` or $p{\rightarrow}key = v$
  - if this is not the case then
    - if $v < p{\rightarrow}key$, search for v in $p{\rightarrow}lft$
    - otherwise, search for v in $p{\rightarrow}rgt$
- Example: search for items with key $= 5$ and key $= 3$



BINARY SEARCH TREES/3

Searching for an item

- To find an item with key $= v$ in a tree rooted at p
  - check whether $p ==$ `NIL` or $p{\rightarrow}key = v$
  - if this is not the case then
    - if $v < p{\rightarrow}key$, search for v in $p{\rightarrow}lft$
    - otherwise, search for v in $p{\rightarrow}rgt$
- Example: search for items with key $= 5$ and key $= 3$



BINARY SEARCH TREES / 4

Iterative algorithm

Algo: TreeSearch(p,v)

---

while $t \neq NIL \wedge p\text{->}key \neq v$ **do**
  if $v < p\text{->}key$ **then** p = p->lft **else** p = p->rgt;

return p;

BINARY SEARCH TREES / 4

Iterative algorithm

Algo: TreeSearch(p,v)

---

while $t \neq NIL \wedge p\text{->}key \neq v$ **do**
  if $v < p\text{->}key$ **then** p = p->lft **else** p = p->rgt;

return p;

Recursive algorithm

Algo: TreeSearch(p,v)

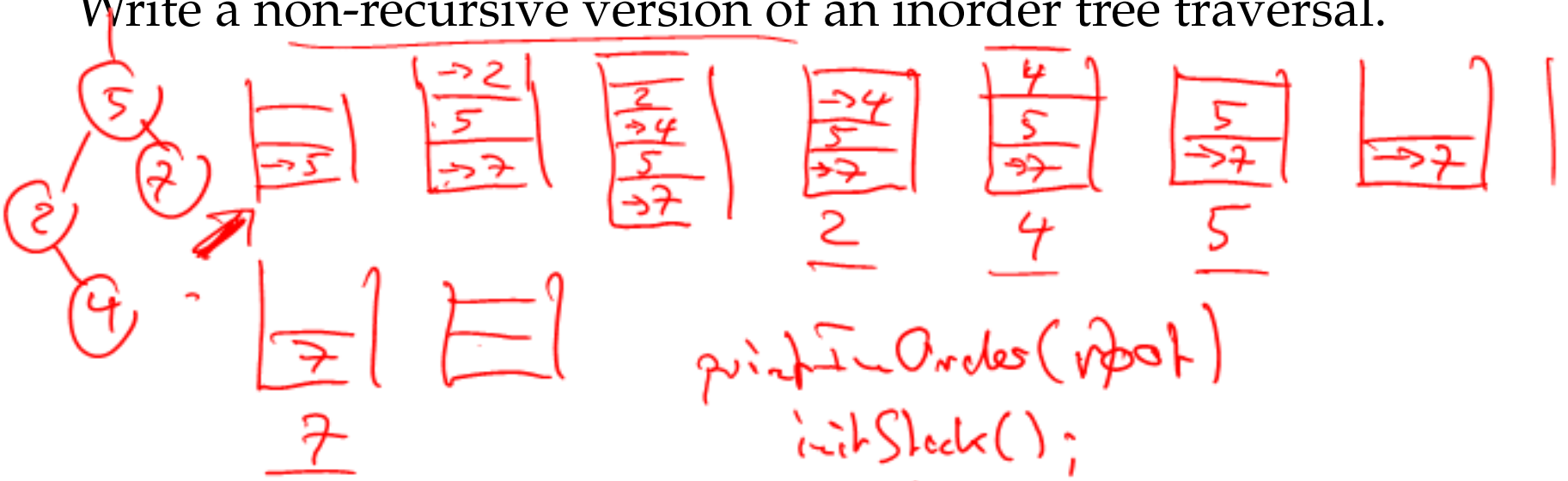
---

if $p == NIL \vee p\text{->}key == v$ **then return** p;
if $v < p\text{->}key$ **then return** TreeSearch(p->lft,v)
else return TreeSearch(p->rgt,v);

EXERCISE SL06-4

2 4 5 7

Write a non-recursive version of an inorder tree traversal.




```
printInOrder(root)
  initStack();
  push(root)
  while (!isEmptyStack()) do
    x = pop()
    if (isNumber(x)) print x
    else if (x->rgt ≠ NIL) push(x->rgt)
      push(x->key
      if (x->lft ≠ NIL) push(x->lft)
```

BINARY SEARCH TREES / 5

Analysis of tree search

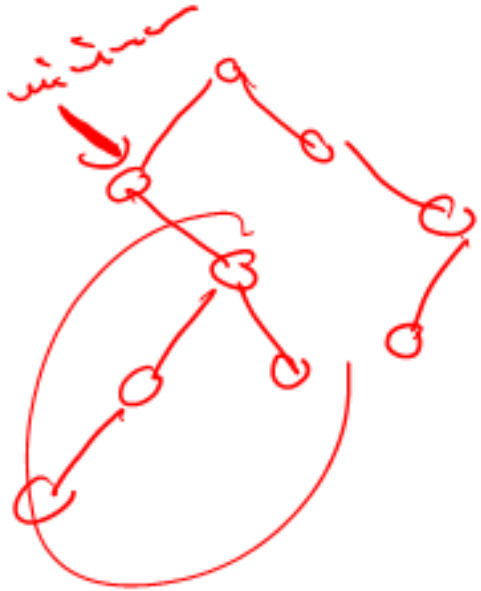
- Running time on tree of height h is $O(h)$ worst case $h = n$

BINARY SEARCH TREES / 5

Analysis of tree search

- Running time on tree of height h is $O(h)$
- In the worst case h is the number of nodes

BINARY SEARCH TREES/5



Analysis of tree search

- Running time on tree of height h is $O(h)$
- In the worst case h is the number of nodes

Finding the minimum

- Finds the minimum key in a tree rooted at p

BINARY SEARCH TREES / 5

Analysis of tree search

- Running time on tree of height h is $O(h)$
- In the worst case h is the number of nodes

Finding the minimum

- Finds the minimum key in a tree rooted at p

Algo: TreeMin(p)

---

while *p->lft≠NIL* **do** p = p->lft;
return p;

BINARY SEARCH TREES / 5

Analysis of tree search

- Running time on tree of height h is $O(h)$
- In the worst case h is the number of nodes

Finding the minimum

- Finds the minimum key in a tree rooted at p

Algo: TreeMin(p)

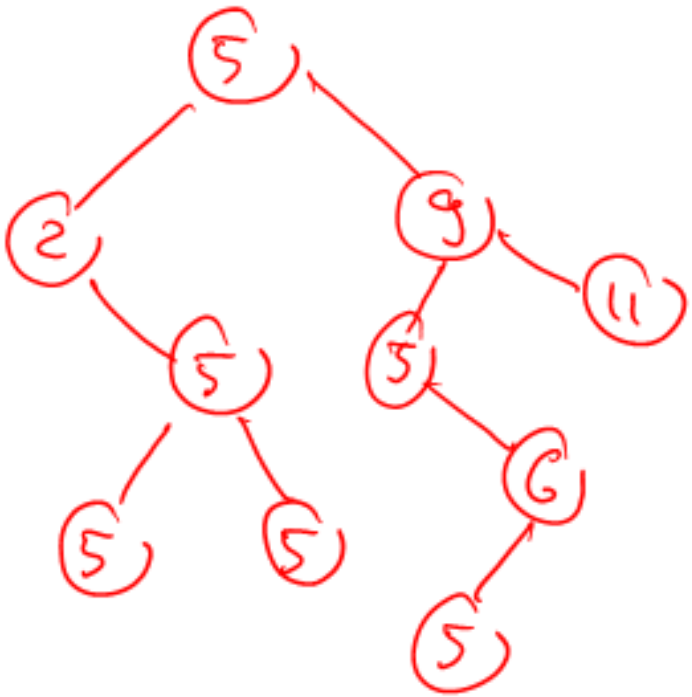
---

while *p->lft≠NIL* **do** p = p->lft;
return p;

- Running time is proportional to height of tree
- We do not have to compare keys.

EXERCISE SL06-5

A function that counts the number of occurrences of an integer x in a binary search tree. (duplicates are possible)




```
cntX(p, v)
  if p ≠ NIL then
    if v < p->val then
      return cntX(p->lft, v)
    else if v > p->val then
      return cntX(p->rgt, v)
    else return cntX(p->lft, v)
               +
           cntX(p->rgt, v)
         + 1
```

BINARY SEARCH TREES/6

Finding the successor/1 (assumption: no duplicates)

- Given node p, find the node with the smallest key $\geq p{\rightarrow}key$

BINARY SEARCH TREES/6

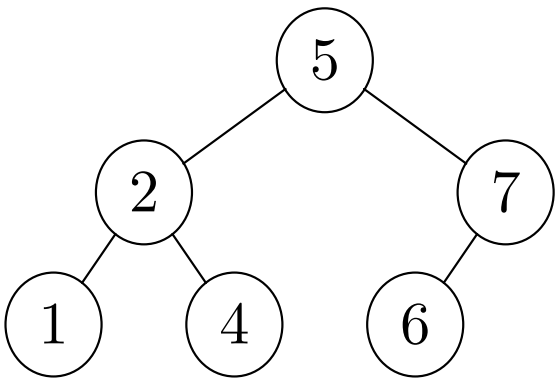
Finding the successor/1 (assumption: no duplicates)

- Given node p, find the node with the smallest key $\geq p{\rightarrow}key$
- Distinguish two cases, depending on right subtree of p

BINARY SEARCH TREES/6

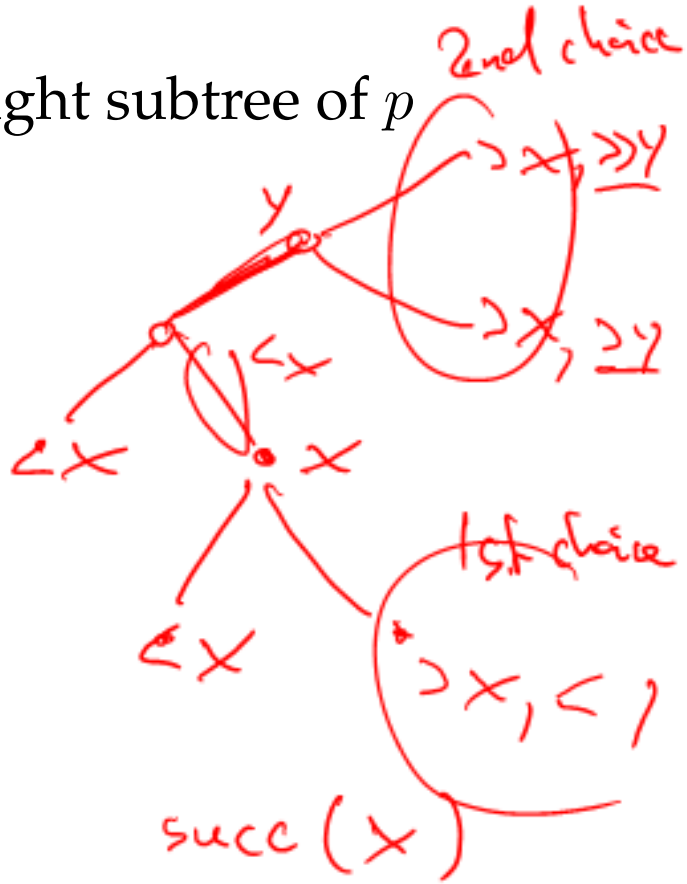
Finding the successor/1 (assumption: no duplicates)

- Given node p, find the node with the smallest key $\geq p{\rightarrow}key$
- Distinguish two cases, depending on right subtree of p



`TreeSucc` of 5 = 6

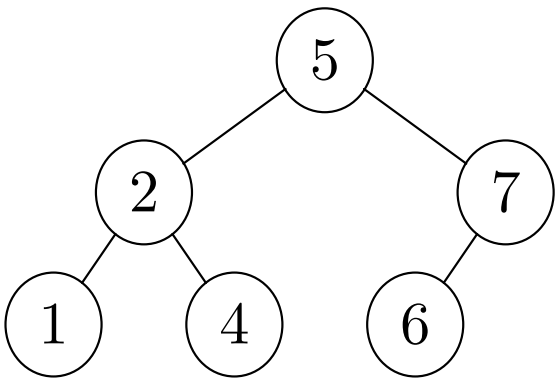
`TreeSucc` of 4 = 5



BINARY SEARCH TREES/6

Finding the successor/1 (assumption: no duplicates)

- Given node p, find the node with the smallest key $\geq p{\rightarrow}key$
- Distinguish two cases, depending on right subtree of p



`TreeSucc` of 5 = 6

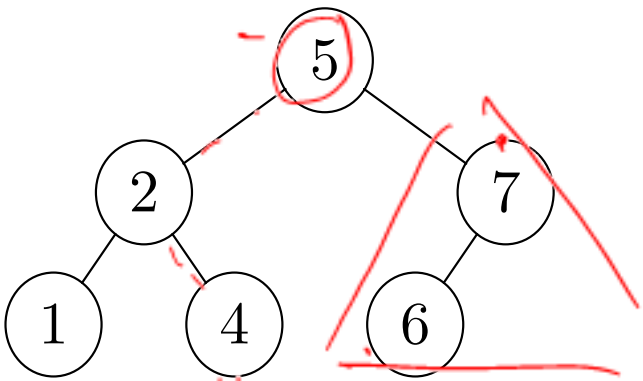
`TreeSucc` of 4 = 5

- Case 1 ($p{\rightarrow}rgt \neq$ `NIL`)
  - successor of p is left-most node in right subtree

BINARY SEARCH TREES/6

Finding the successor/1 (assumption: no duplicates)

- Given node p, find the node with the smallest key $\geq p{\rightarrow}key$
- Distinguish two cases, depending on right subtree of p



`TreeSucc` of 5 = 6

`TreeSucc` of 4 = 5

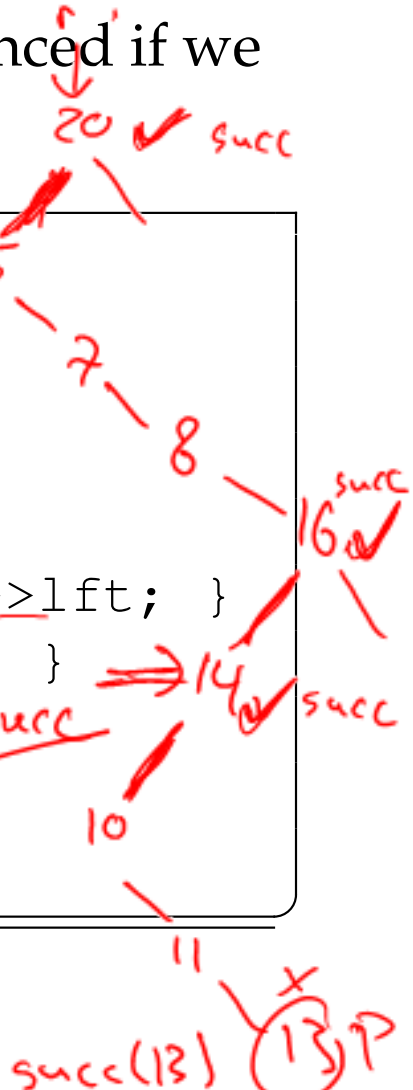
- Case 1 ($p{\rightarrow}rgt \neq$ `NIL`)
  - successor of p is left-most node in right subtree
- Case 2 ($p{\rightarrow}rgt =$ `NIL`)
  - successor of p is lowest ancestor whose left subtree contains p

BINARY SEARCH TREES/7

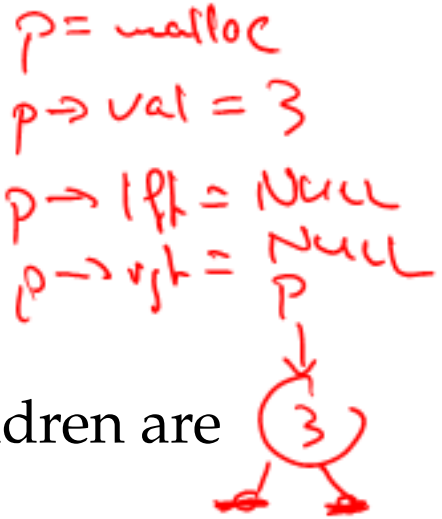
- For (standard) binary trees without a parent pointer we need a pointer that lags behind and is only advanced if we move left.

```
struct node* SuccAbove(struct node* p,
                       struct node* x) {
  struct node* succ;
  succ = NULL;
  while (p != x) {
    if (x->key < p->key) { succ = p; p = p->lft; }
    else if (x->key > p->key) { p = p->rgt; }
  }
  return succ;
}
```

Usage: `SuccAbove(root, Search(root,v))`



INSERTING ITEMS / 1



Underlying idea

- Assume that a node p whose left and right children are NIL is given

INSERTING ITEMS / 1

Underlying idea

- Assume that a node p whose left and right children are NIL is given
- Insertion is similar to searching

INSERTING ITEMS / 1

Underlying idea

- Assume that a node p whose left and right children are NIL is given
- Insertion is similar to searching
- To insert the node p into a tree rooted at $root$
  - find location in tree where p belongs to
  - then add the new node p there

INSERTING ITEMS / 1

Underlying idea

- Assume that a node p whose left and right children are NIL is given
- Insertion is similar to searching
- To insert the node p into a tree rooted at $root$
  - find location in tree where p belongs to
  - then add the new node p there
- Makes use of a "one step delayed" pointer

INSERTING ITEMS / 1

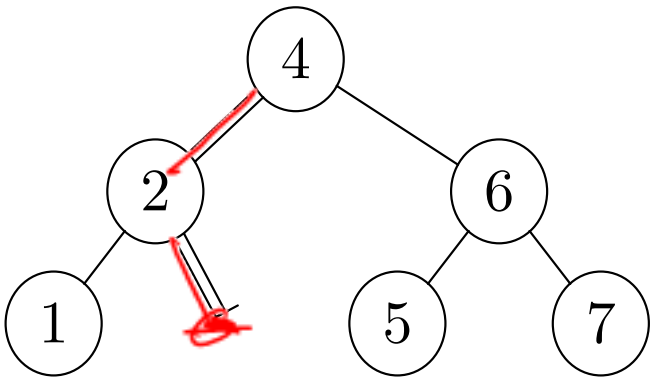
Underlying idea

- Assume that a node p whose left and right children are NIL is given
- Insertion is similar to searching
- To insert the node p into a tree rooted at $root$
  - find location in tree where p belongs to
  - then add the new node p there
- Makes use of a "one step delayed" pointer
- Running time on tree of height h is $O(h)$

INSERTING ITEMS/2

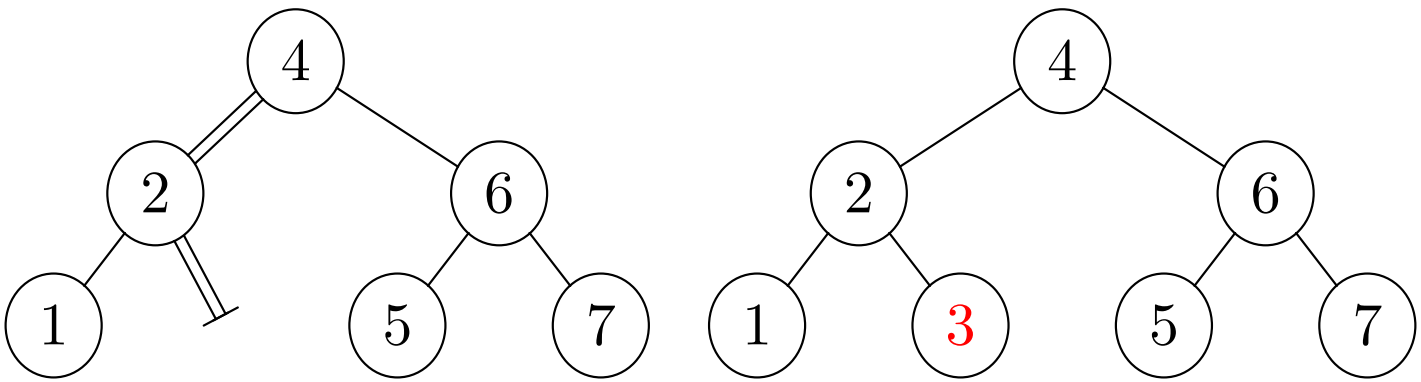
Insertion example

+3



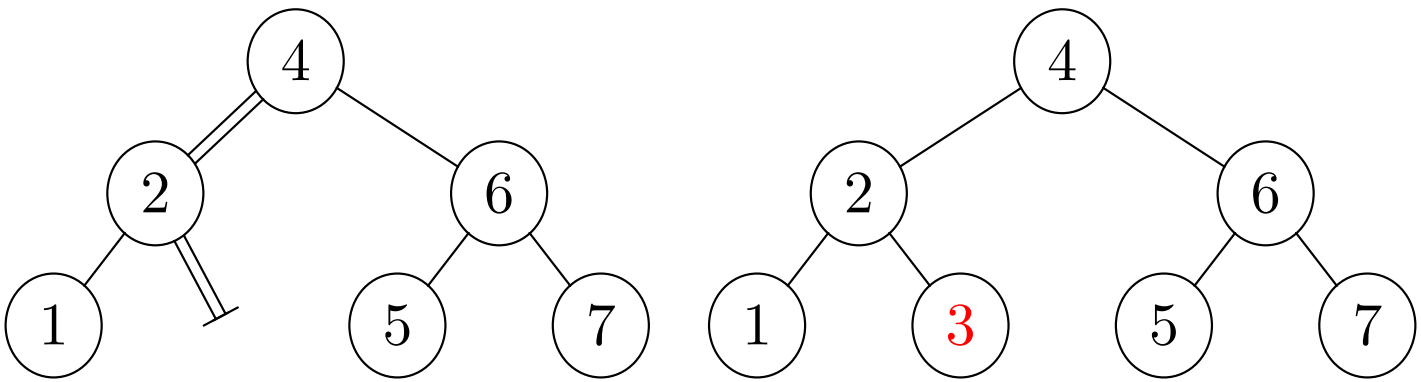
INSERTING ITEMS/2

Insertion example



INSERTING ITEMS/2

Insertion example

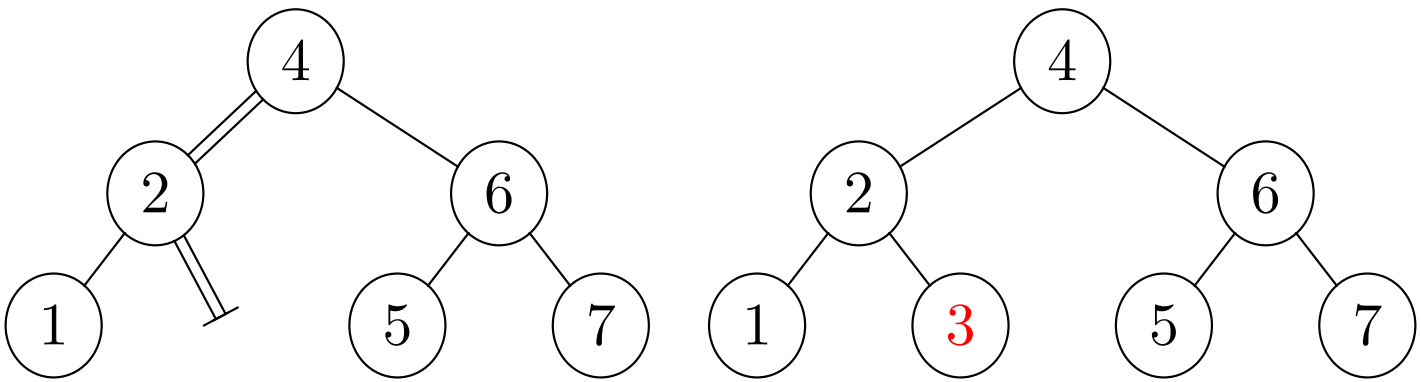


Problematic instances

- If items are inserted in order, the tree looks like a list

INSERTING ITEMS/2

Insertion example

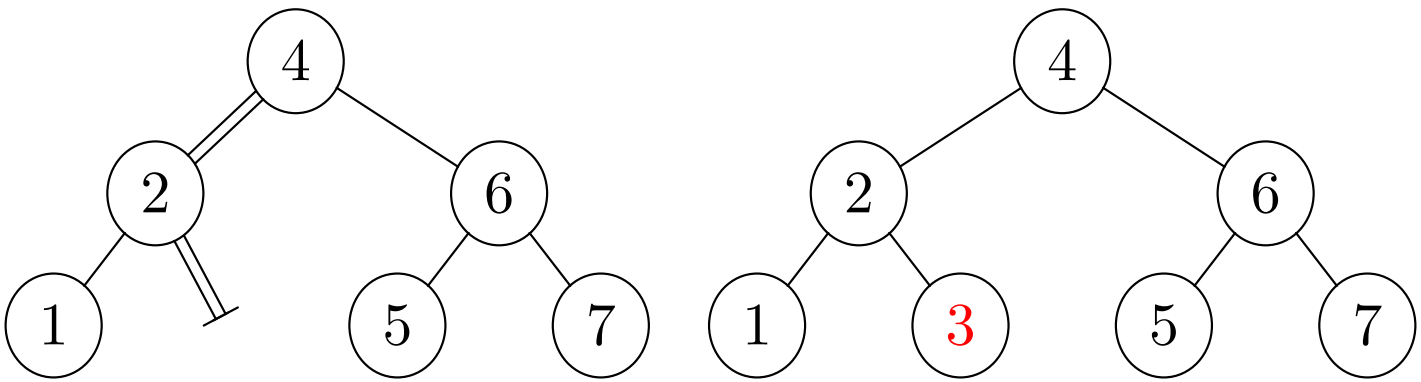


Problematic instances

- If items are inserted in order, the tree looks like a list
- That is: height $h = n$, the number of nodes in the tree

INSERTING ITEMS/2

Insertion example

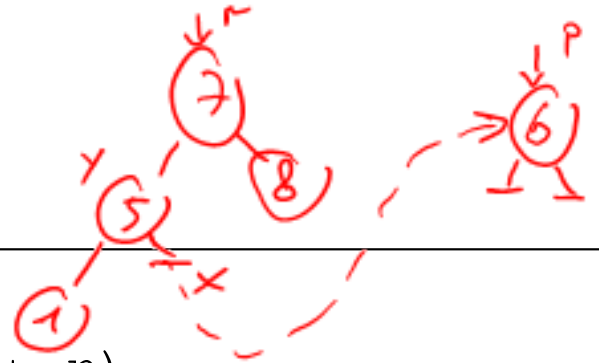


Problematic instances

- If items are inserted in order, the tree looks like a list
- That is: height $h = n$, the number of nodes in the tree
- Worst case: the running time of the operations is $O(n)$

INSERTING ITEMS / 3

Have a "one step delayed" pointer:




```
struct node* TreeInsert(
  struct node* p, struct node* r)
{
  struct node* y = NULL;
  struct node* x = r;
  while (x != NULL) {
    y := x;
    if (x->key < p->key) x = x->rgt
    else x = x->lft;
  }
  if (y == NULL) r = p;
  else if (y->key < p->key) y->rgt = p;
  else y->lft = p;
  return r;
}
```

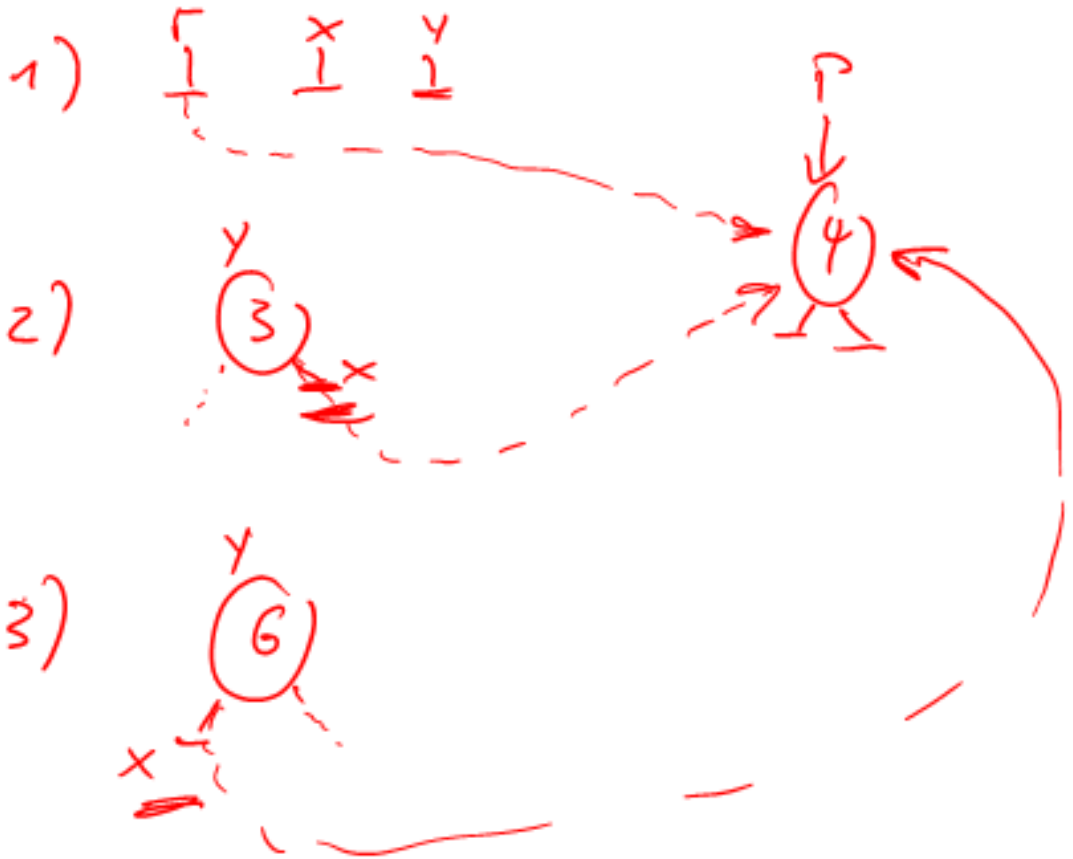
search for place x where p must be inserted. y is one step behind x

1)
2)
3)

call: root = TreeInsert(p, root)

EXERCISE SL06-6

Use representative trees to illustrate the insertion with a one step delayed pointer.



DELETING ITEMS/1

General remark

- Deletion depends on how many children a node has

DELETING ITEMS/1

General remark

- Deletion depends on how many children a node has

Deletion: case 1

- Node t has no children: remove t

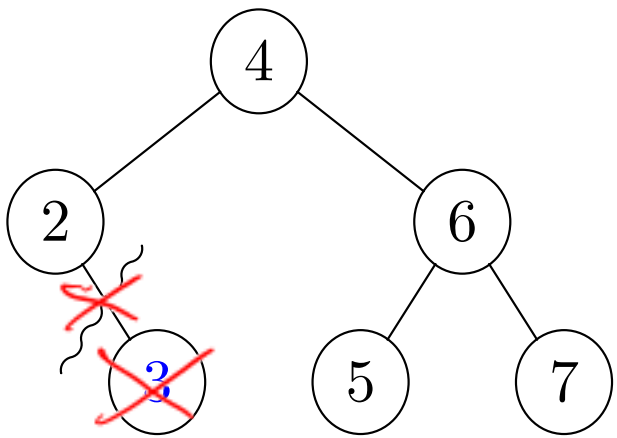
DELETING ITEMS/1

General remark

- Deletion depends on how many children a node has

Deletion: case 1

- Node t has no children: remove t



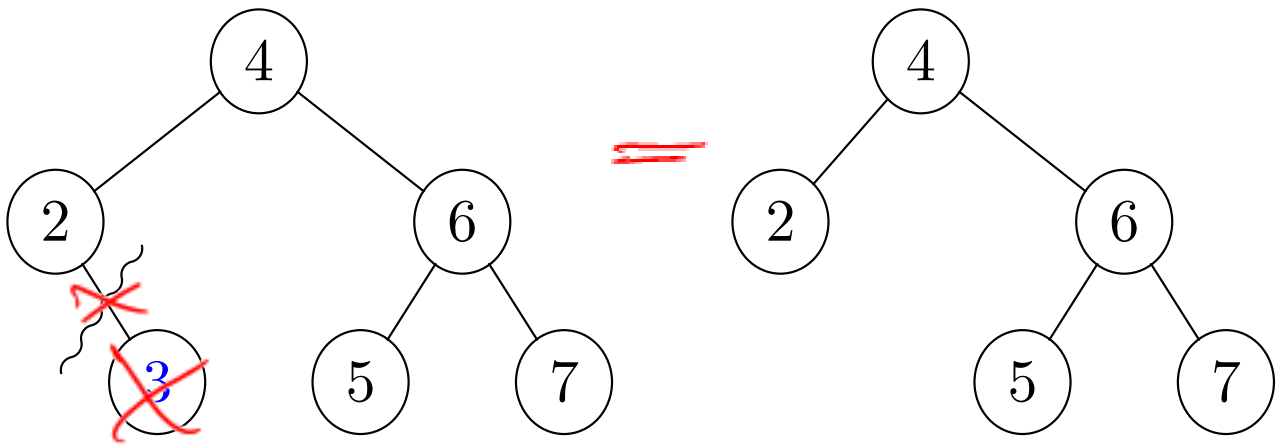
DELETING ITEMS/1

General remark

- Deletion depends on how many children a node has

Deletion: case 1

- Node t has no children: remove t



DELETING ITEMS/2

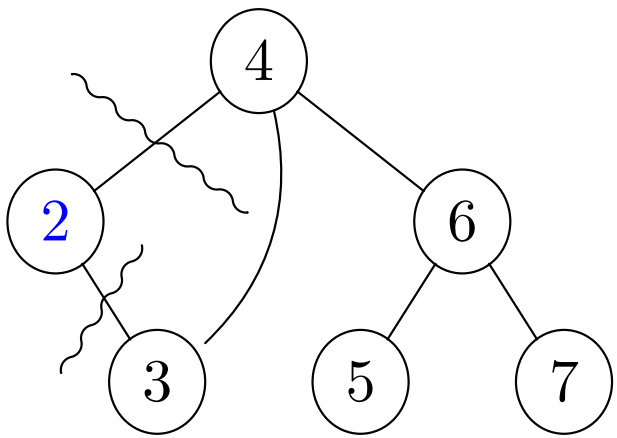
Deletion: case 2

- Node t has one child x: let parent of t point to x and then remove t

DELETING ITEMS/2

Deletion: case 2

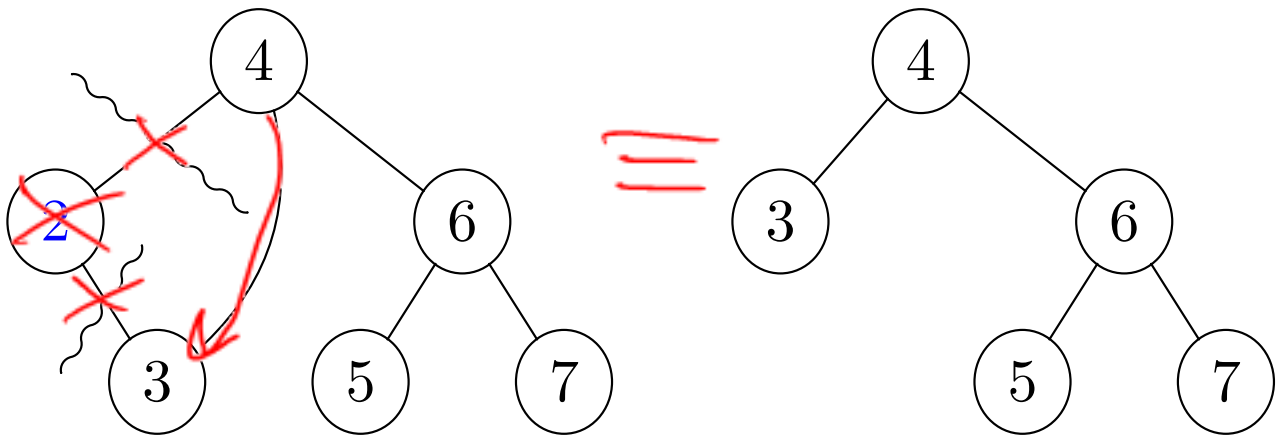
- Node t has one child x: let parent of t point to x and then remove t



DELETING ITEMS/2

Deletion: case 2

- Node t has one child x: let parent of t point to x and then remove t



DELETING ITEMS/3

Deletion: case 3

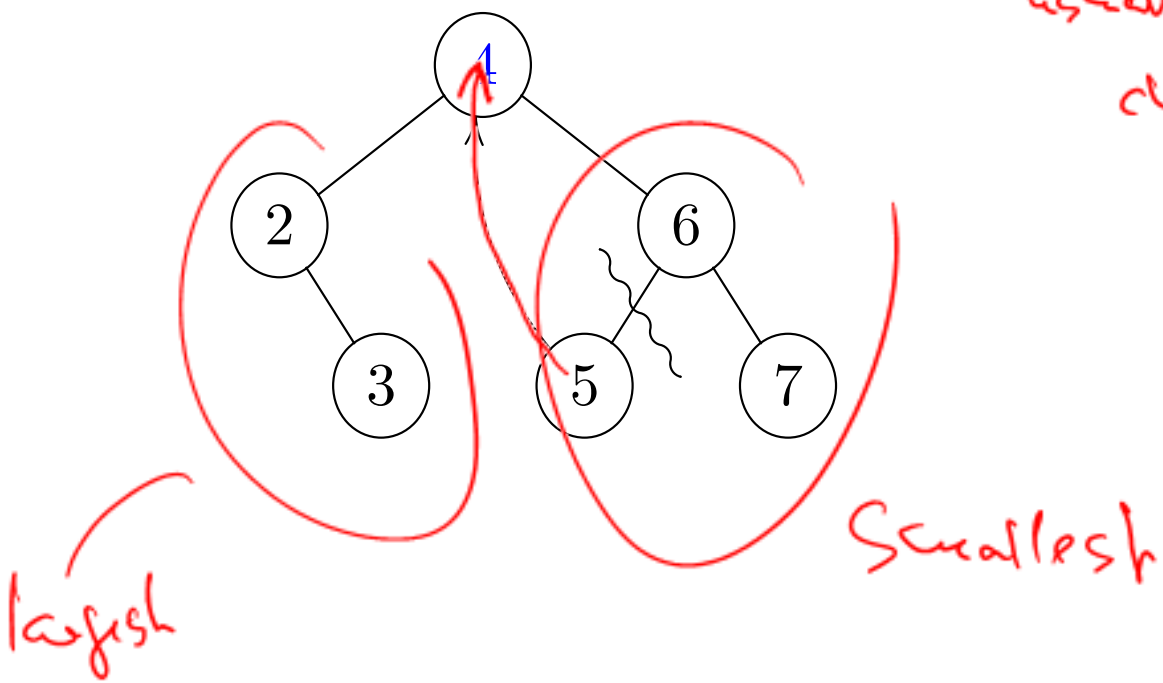
- Node t has two children: find largest (smallest) child s in left (right) subtree of t, then replace $p{\rightarrow}key$ with $s.key$ and remove s

DELETING ITEMS/3

Deletion: case 3

- Node t has two children: find largest (smallest) child s in left (right) subtree of t, then replace $p{\rightarrow}key$ with $s.key$ and remove s

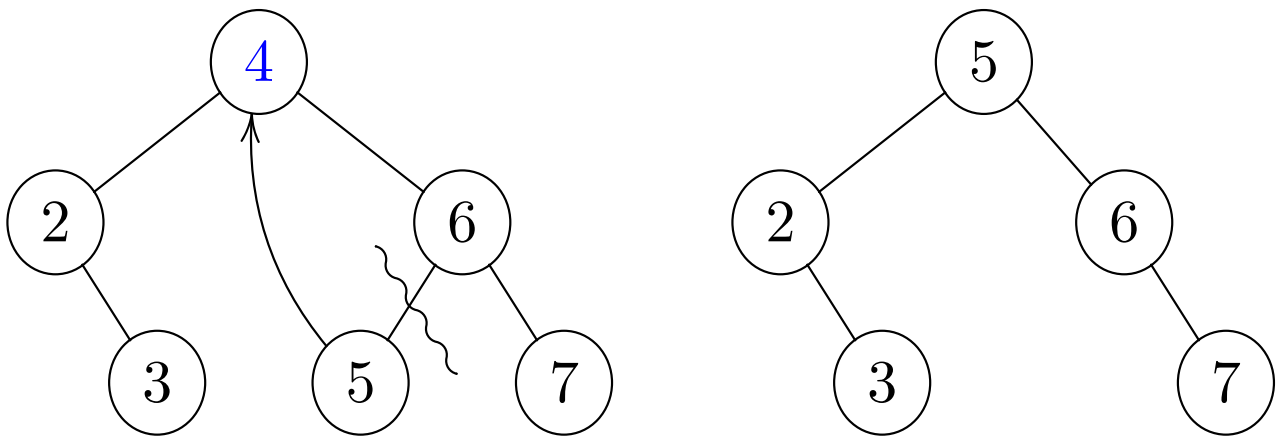
usually done by changing pointers



DELETING ITEMS/3

Deletion: case 3

- Node t has two children: find largest (smallest) child s in left (right) subtree of t, then replace $p{\rightarrow}key$ with $s.key$ and remove s



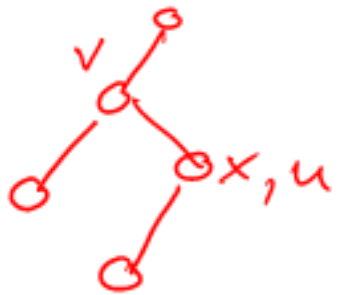
DELETING ITEMS/4

The algorithm

```
struct node* delete(struct node* root,
                    struct node* x) {

  u = root; v = NULL;
  while (u != x) {
    v = u;
    if (x->key < u->key) u = u->lft;
    else u = u->rgt;
  }

...
```



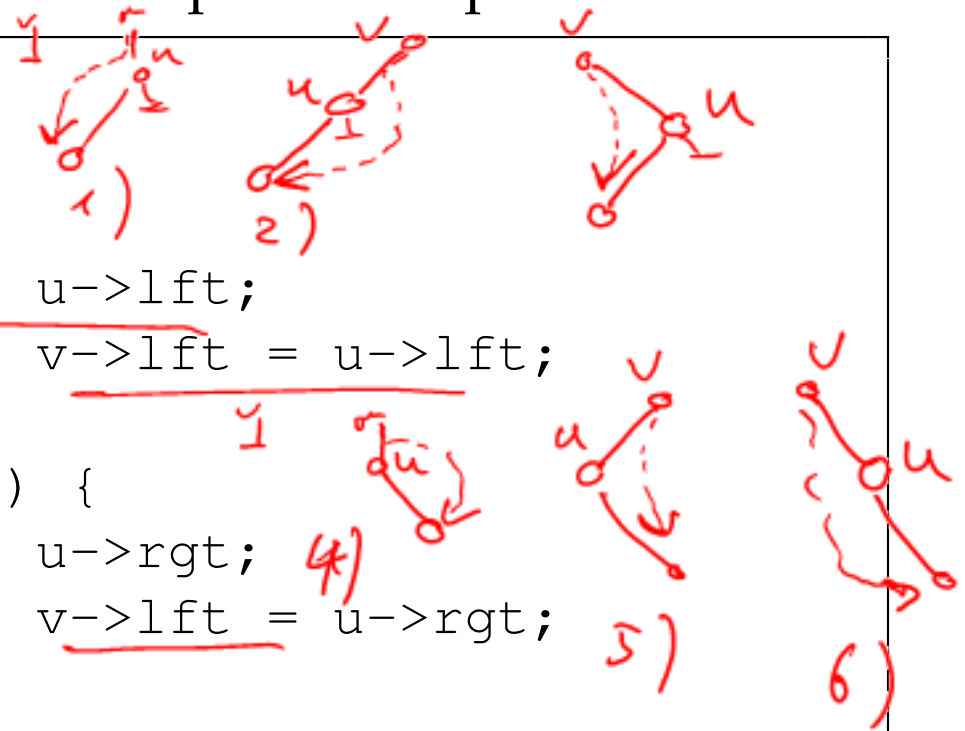
DELETING ITEMS/5

If x has less than 2 children we fix the pointer of parent of x:

```
...

  if (u->rgt == NULL) {
1) if (v == NULL) root = u->lft;
2) else if (v->lft == u) v->lft = u->lft;
3) else v->rgt = u->lft;
  else if (u->lft == NULL) {
4) if (v == NULL) root = u->rgt;
5) else if (v->lft == u) v->lft = u->rgt;
6) else v->rgt = u->rgt;
  else {

...
```



DELETING ITEMS/6

If x has 2 children:

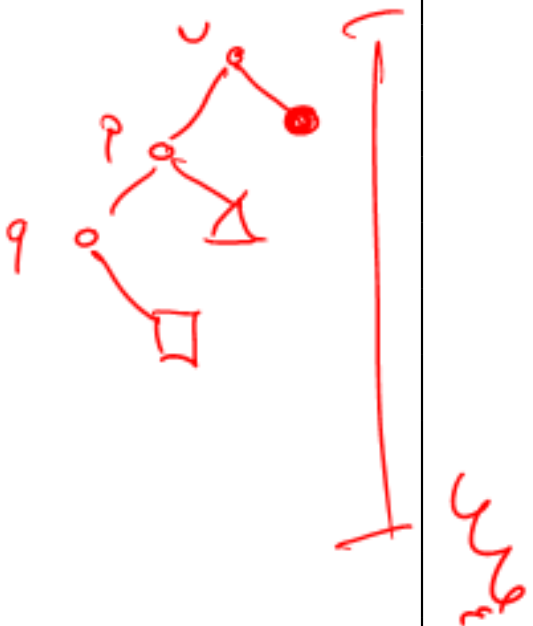
call: p = TreeSearch(r, 5)
r = delete(r, p)
free(p)

```
...
  p = x->lft; q = p;
  while (p->rgt != NULL) { q = p; p = p->rgt; }

1) if (v == NULL) root = p;
   else if (v->lft == u) v->lft = p;
   else v->rgt = p;

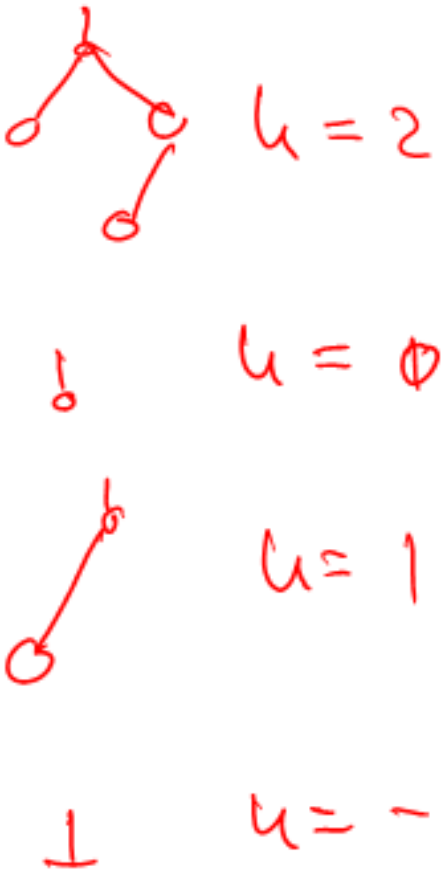
2) p->rgt = u->rgt;
   if (q != p) {
3)   q->rgt = p->lft;
4)   p->lft = u->lft;
   }
   return root
}
```





EXERCISE SL06-7

Given a binary tree write a function that determines its height.




```
int heightBT(struct node* p)
  if (p==NULL) return -1
  return max(heightBT(p->left),
             heightBT(p->right))
+
1 ;
```

TABLE OF CONTENTS — SL06

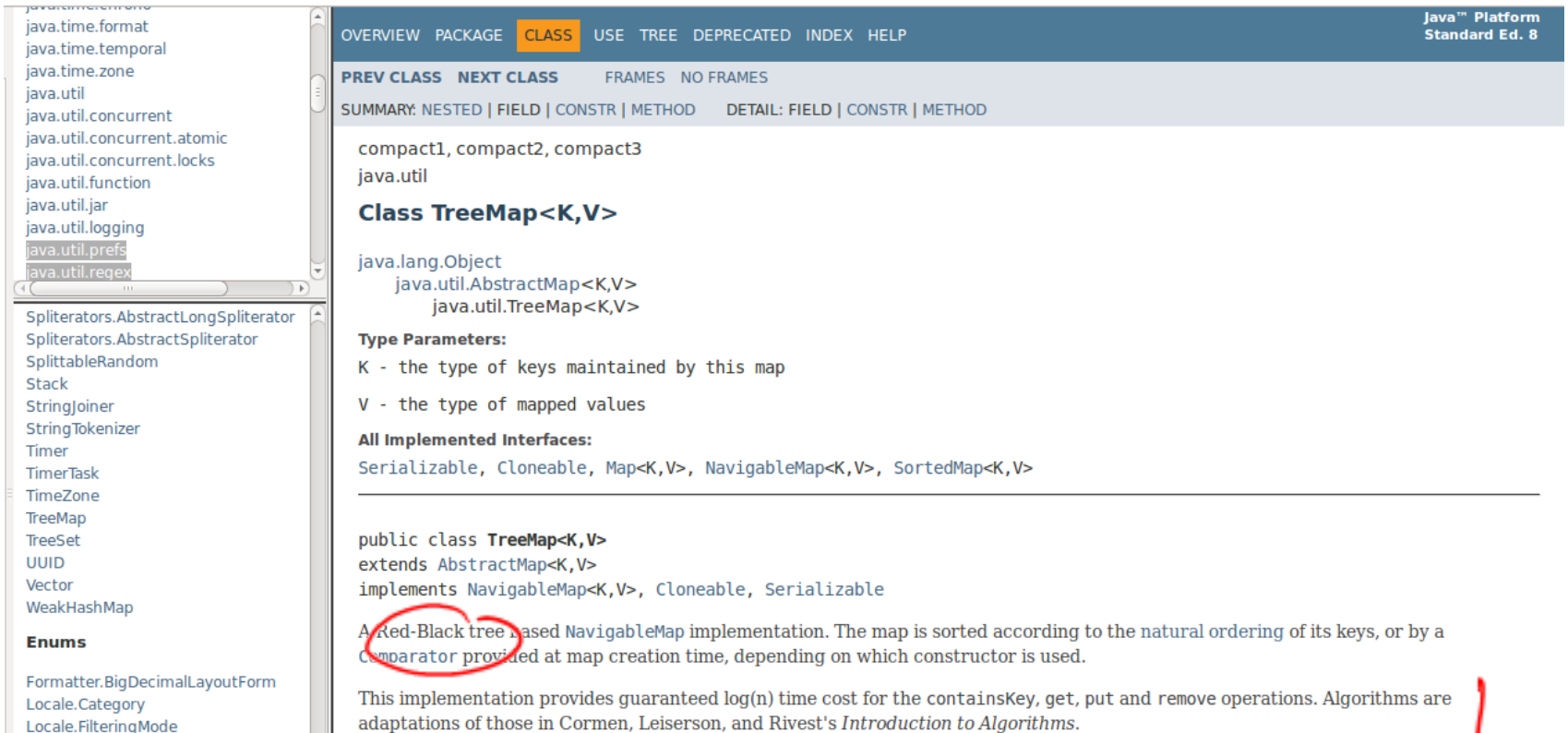
1. Binary trees

2. Red-black trees
   Inserting items
   Deleting items

guarantee for worst case

RED-BLACK TREES / 1

Example: Java's TreeMap class uses red-black trees



RED-BLACK TREES/2

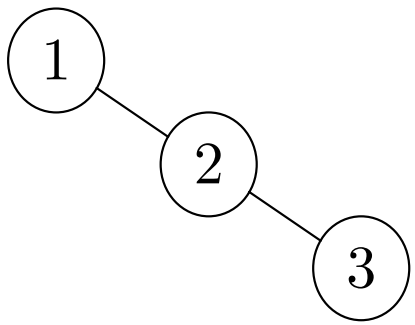
Balanced binary trees

- Problem: running time of tree operations is $O(h)$, worst case is $O(n)$

RED-BLACK TREES/2

Balanced binary trees

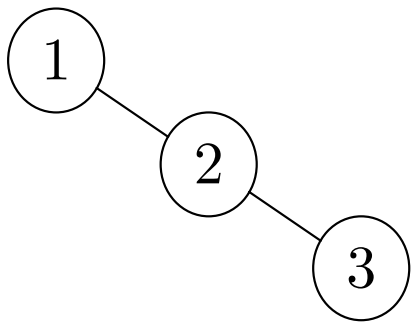
- Problem: running time of tree operations is $O(h)$, worst case is $O(n)$



RED-BLACK TREES/2

Balanced binary trees

- Problem: running time of tree operations is $O(h)$, worst case is $O(n)$

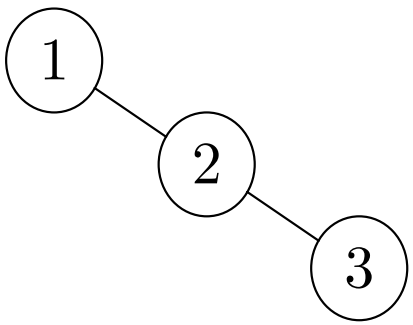


- Solution: balanced search trees guarantee small height $h = O(\lg n)$

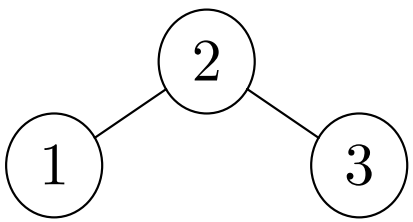
RED-BLACK TREES/2

Balanced binary trees

- Problem: running time of tree operations is $O(h)$, worst case is $O(n)$



- Solution: balanced search trees guarantee small height $h = O(\lg n)$



RED-BLACK TREES / 3

Red-black property

- A red-black tree is a binary search tree with the following (additional) properties
  1. Every node is either red or black
  2. The root is black
  3. Every leaf (`NIL`) is black
  4. If a node is red then both its children are black
  5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

RED-BLACK TREES / 3

Red-black property

- A red-black tree is a binary search tree with the following (additional) properties
  1. Every node is either red or black
  2. The root is black
  3. Every leaf (`NIL`) is black
  4. If a node is red then both its children are black
  5. For each node, all paths from the node to descendant leaves contain the same number of black nodes
- Black-height $bh(x)$ of node x: number of black nodes on any path from, but not including, x down to a leaf

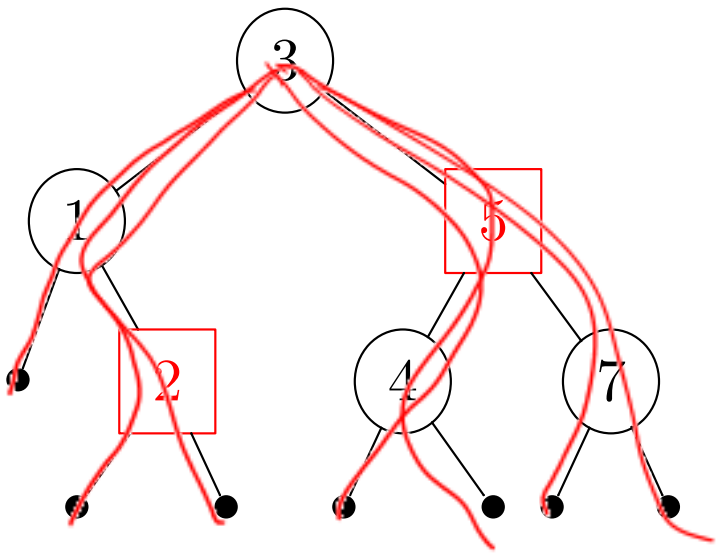
RED-BLACK TREES / 3

Red-black property

- A red-black tree is a binary search tree with the following (additional) properties
  1. Every node is either red or black
  2. The root is black
  3. Every leaf (`NIL`) is black
  4. If a node is red then both its children are black
  5. For each node, all paths from the node to descendant leaves contain the same number of black nodes
- Black-height $bh(x)$ of node x: number of black nodes on any path from, but not including, x down to a leaf
- Black-height bh of tree: $bh(root)$, black-height of root

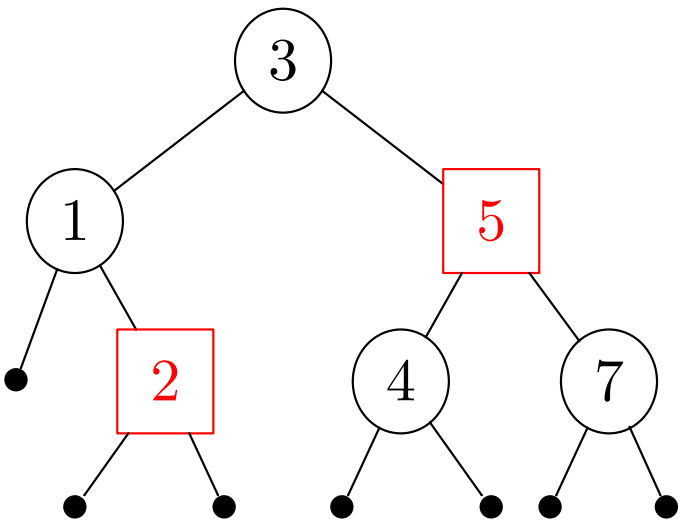
RED-BLACK TREES/4

Basic consequences/1



RED-BLACK TREES/4

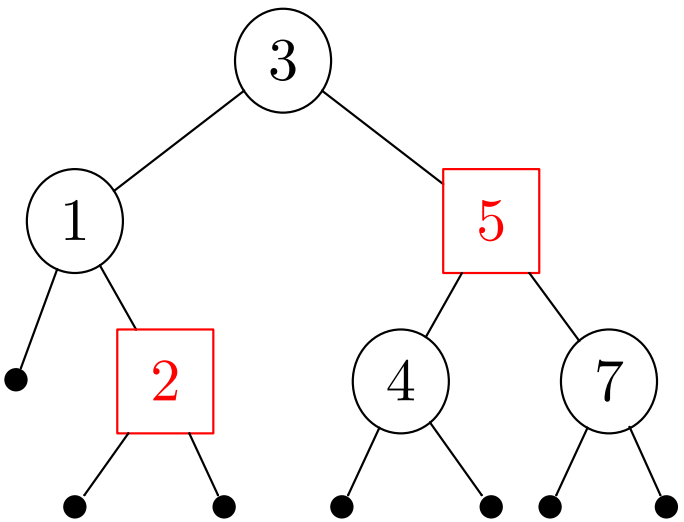
Basic consequences/1



- Some measures
  - n internal nodes
  - h - height of tree
  - bh - black height

RED-BLACK TREES/4

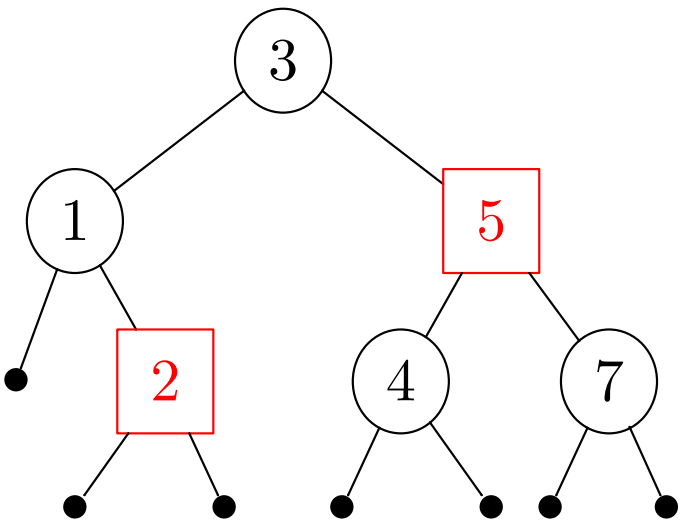
Basic consequences/1



- Some measures
  - n internal nodes
  - h - height of tree
  - bh - black height
- $h/2 \leq bh$

RED-BLACK TREES/4

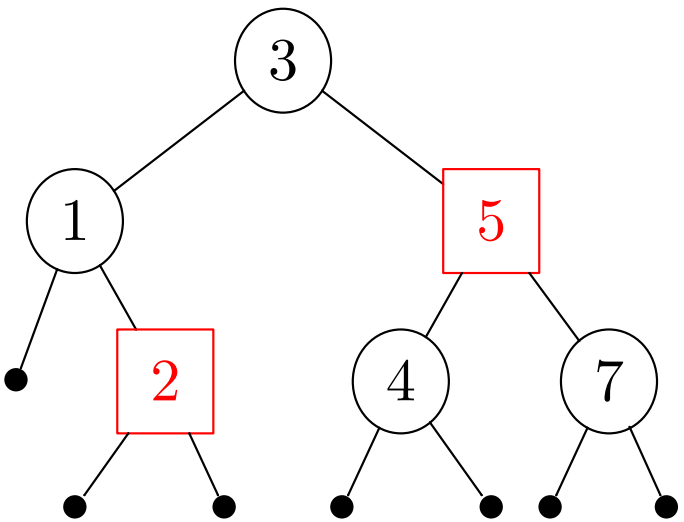
Basic consequences/1



- Some measures
  - n internal nodes
  - h - height of tree
  - bh - black height
- $h/2 \leq bh$
- $2^{bh} - 1 \leq n$

RED-BLACK TREES/4

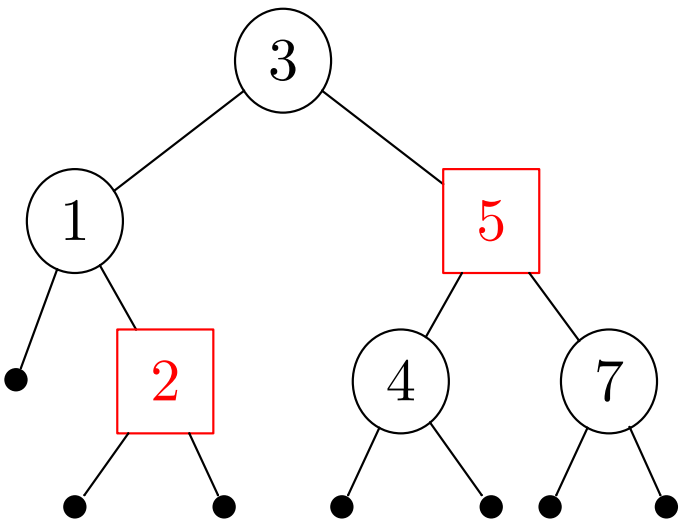
Basic consequences/1



- Some measures
  - n internal nodes
  - h - height of tree
  - bh - black height
- $h/2 \leq bh$
- $2^{bh} - 1 \leq n$
- $2^{h/2} \leq n + 1$

RED-BLACK TREES/4

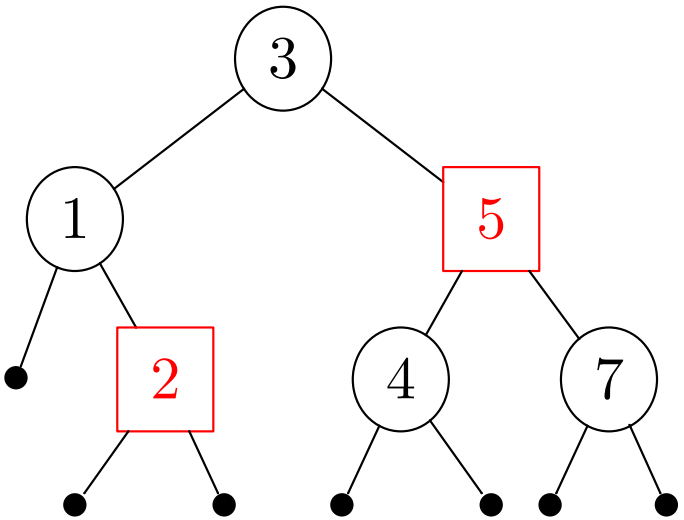
Basic consequences/1



- Some measures
  - n internal nodes
  - h - height of tree
  - bh - black height
- $h/2 \leq bh$
- $2^{bh} - 1 \leq n$
- $2^{h/2} \leq n + 1$
- $h/2 \leq \lg(n + 1)$

RED-BLACK TREES / 4

Basic consequences / 1



- Some measures
  - n internal nodes
  - h - height of tree
  - bh - black height
- $h/2 \leq bh$
- $2^{bh} - 1 \leq n$
- $2^{h/2} \leq n + 1$
- $h/2 \leq \lg(n+1)$
- $h \leq 2\lg(n+1)$ - very good



RED-BLACK TREES/5

Basic consequences/2

- Operations on a binary search tree (search, insert, delete, etc.) are accomplished in $O(h)$ time

RED-BLACK TREES/5

Basic consequences/2

- Operations on a binary search tree (search, insert, delete, etc.) are accomplished in $O(h)$ time
- A RB-tree is a binary search tree whose height is bound by $2\lg(n+1)$

RED-BLACK TREES/5

Basic consequences/2

- Operations on a binary search tree (search, insert, delete, etc.) are accomplished in $O(h)$ time
- A RB-tree is a binary search tree whose height is bound by $2\lg(n+1)$
- Thus, operations are accomplished in $O(\lg n)$ time

RED-BLACK TREES/5

Basic consequences/2

- Operations on a binary search tree (search, insert, delete, etc.) are accomplished in $O(h)$ time
- A RB-tree is a binary search tree whose height is bound by $2\lg(n+1)$
- Thus, operations are accomplished in $O(\lg n)$ time
- Note that there is the following constraint
  - Insertions and deletions may destroy RB-property
  - So, cost for restoring it cannot be more than $O(h)$

RED-BLACK TREES / 6

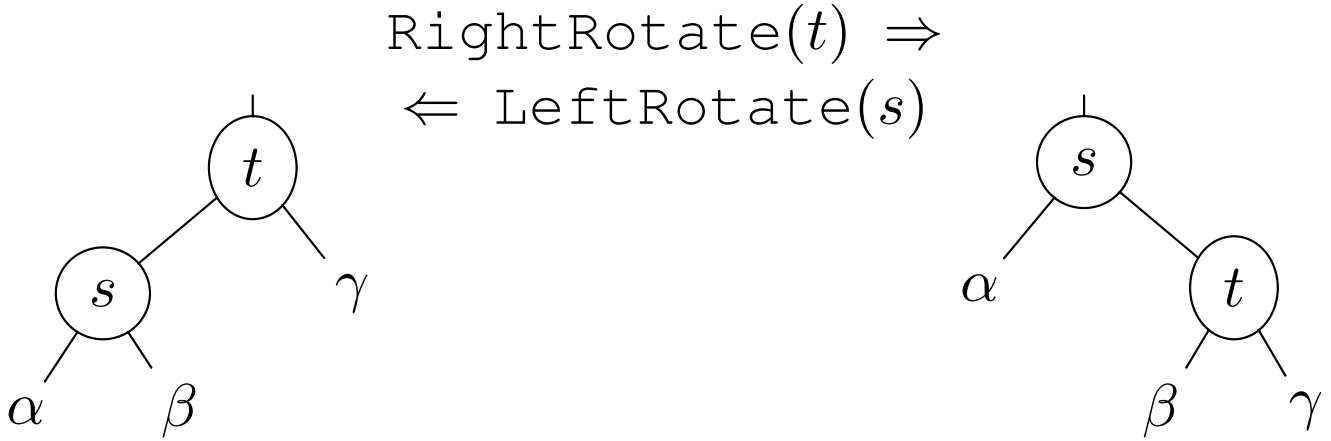
Left & right rotations / 1

- To change pointer structure of tree

RED-BLACK TREES / 6

Left & right rotations / 1

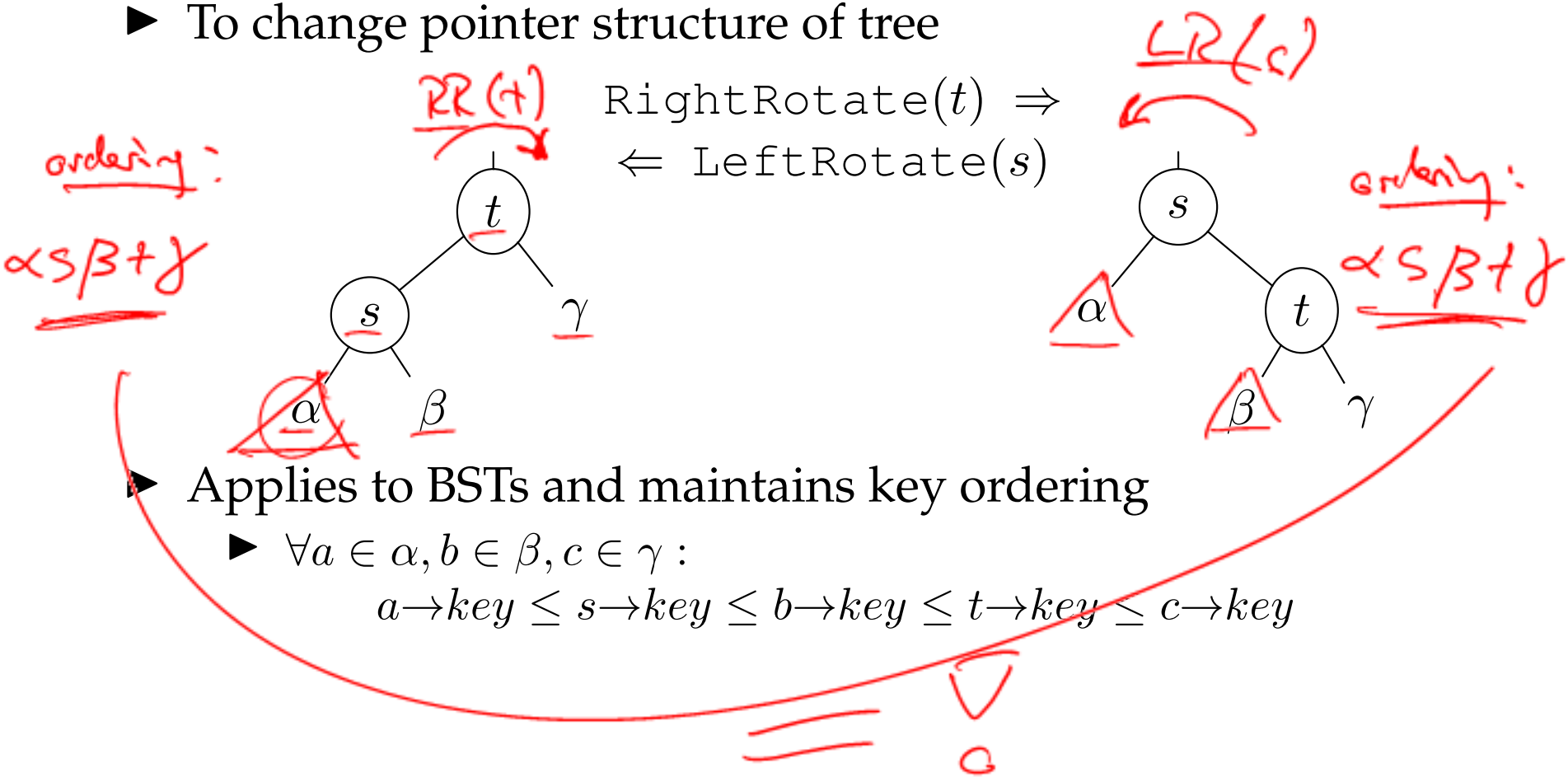
- To change pointer structure of tree



RED-BLACK TREES / 6

Left & right rotations / 1

- To change pointer structure of tree

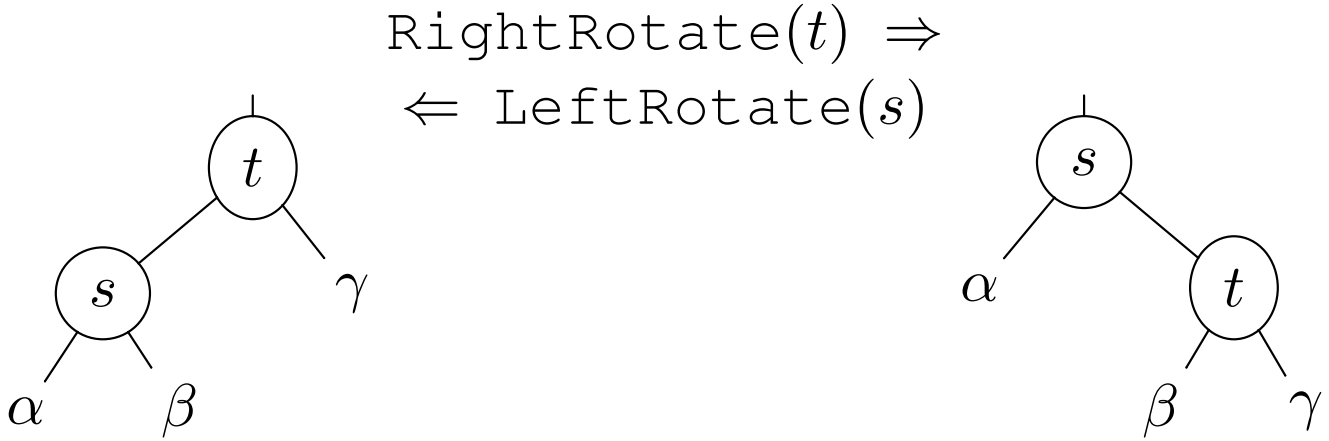


- Applies to BSTs and maintains key ordering
  - $\forall a \in \alpha, b \in \beta, c \in \gamma$:
    $a{\rightarrow}key \leq s{\rightarrow}key \leq b{\rightarrow}key \leq t{\rightarrow}key \leq c{\rightarrow}key$

RED-BLACK TREES / 6

Left & right rotations / 1

- To change pointer structure of tree



- Applies to BSTs and maintains key ordering
  - $\forall a \in \alpha, b \in \beta, c \in \gamma$:
    $$a{\rightarrow}key \leq s{\rightarrow}key \leq b{\rightarrow}key \leq t{\rightarrow}key \leq c{\rightarrow}key$$
- Rotations decrease/increase depth of α, γ (but not β)

RED-BLACK TREES/7

Left & right rotations/2

- Algorithm for right rotation (left rotation is equivalent)

RED-BLACK TREES/7

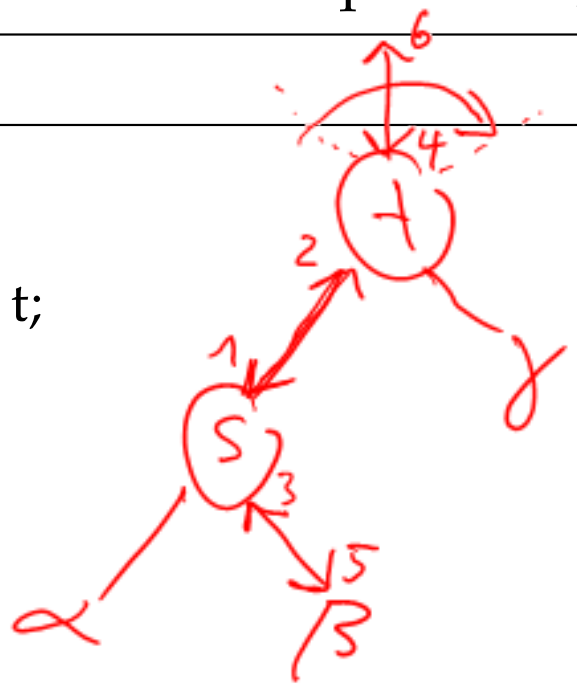
Left & right rotations/2

impl uses pointers to children and parent: simplifies the code

- Algorithm for right rotation (left rotation is equivalent)

Algo: RightRotate(t)

```
s = t->lft;
t->lft = s->rgt; s->pnt = t->pnt;
if s->rgt≠NIL then (s->rgt)->pnt = t;
if t->pnt==NIL then
  | root = s
else
  | if t==(t->pnt)->rgt then
  |   | (t->pnt)->rgt = s
  | else
  |   | (t->pnt)->lft = s
s->rgt = t; t->pnt = s;
```



change 6 pointers ≡ rotation

RED-BLACK TREES/7

Left & right rotations/2

- Algorithm for right rotation (left rotation is equivalent)

Algo: RightRotate(t)

```
s = t->lft;
t->lft = s->rgt; s->pnt = t->pnt;
if s->rgt≠NIL then (s->rgt)->pnt = t;
if t->pnt==NIL then
  | root = s
else
  | if t==(t->pnt)->rgt then
  |   | (t->pnt)->rgt = s
  | else
  |   | (t->pnt)->lft = s
s->rgt = t; t->pnt = s;
```

- Running time is $O(1)$

INSERTING ITEMS / 1

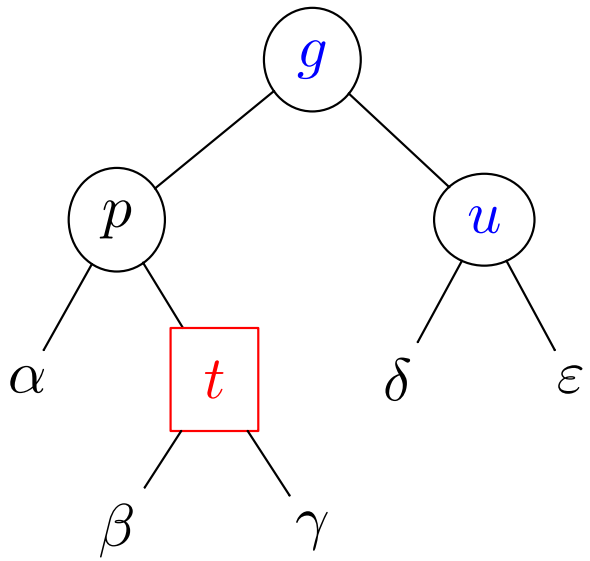
The algorithm

Algo: RBTInsert(t)

```
s = NIL;
x = root;
while x≠NIL do
    s = x;
    if t->key<x->key then x = x->lft else x = x->rgt;
t->pnt = s; t->col = red;
if s==NIL then
    root = t
else
    if t->key<s->key then s->lft = t else s->rgt = t;
InsertFixup(t)
```

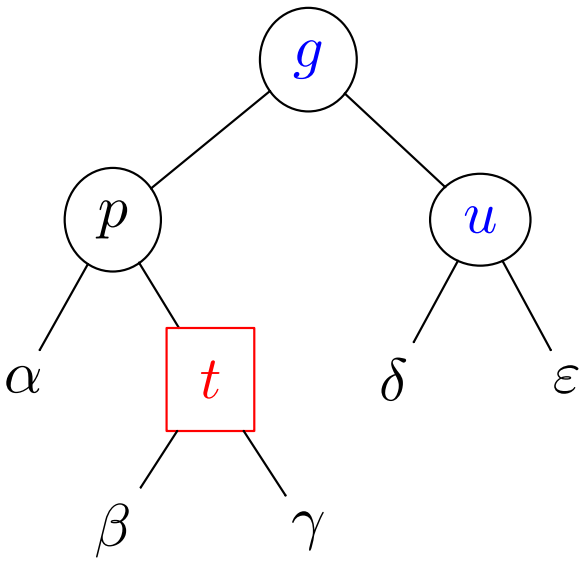
INSERTING ITEMS/2

Insertion: case 0



INSERTING ITEMS/2

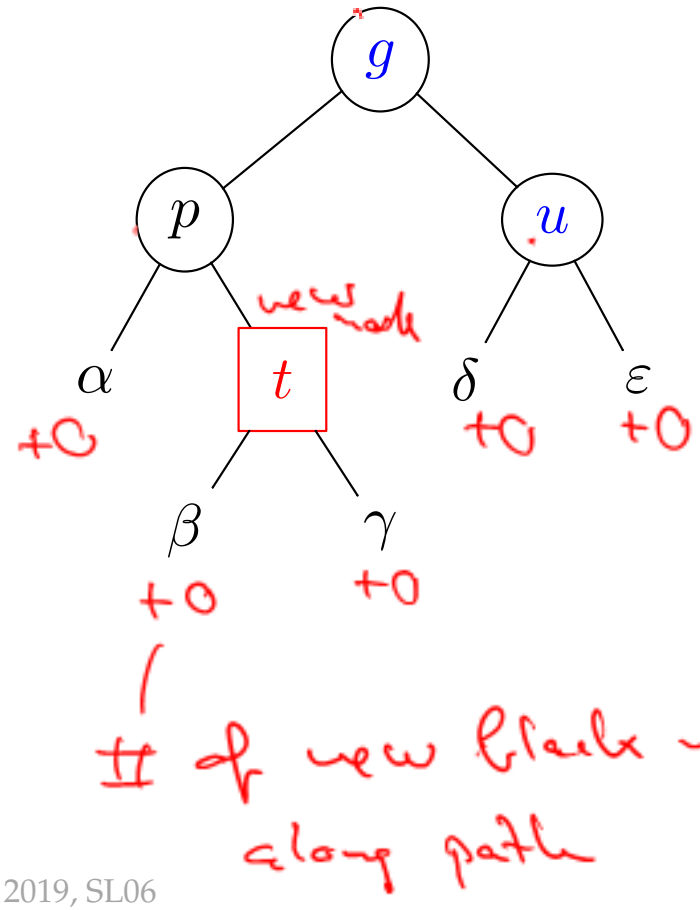
Insertion: case 0



- Assumption
  - p is left child of g

INSERTING ITEMS/2

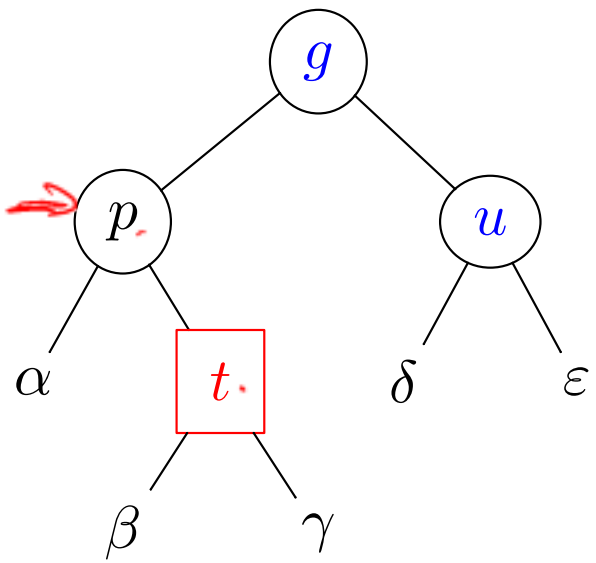
Insertion: case 0



- Assumption
  - p is left child of g
- Conventions
  - t — new node
  - p — parent of t
  - q — grandfather of t
  - u — uncle of t

INSERTING ITEMS/2

Insertion: case 0



- Assumption
  - p is left child of g
- Conventions
  - t — new node
  - p — parent of t
  - g — grandfather of t
  - u — uncle of t
- Color of p is black
  - No property violated
  - Hence nothing to do

INSERTING ITEMS / 3

Insertion: case 1

- t's uncle u is red

INSERTING ITEMS / 3

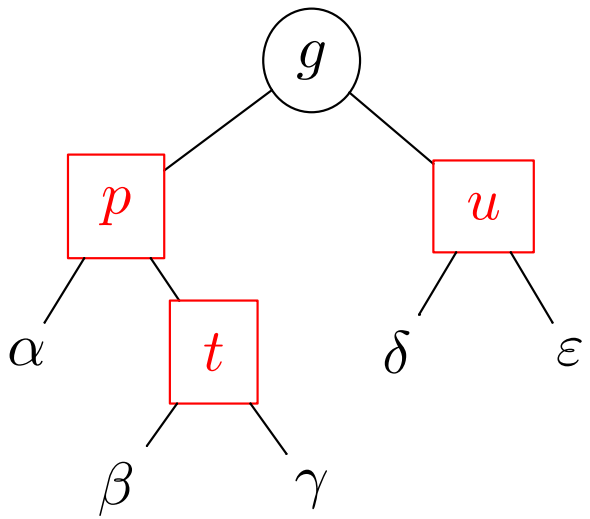
Insertion: case 1

- t's uncle u is red
- Action: $p{\rightarrow}col = u{\rightarrow}col = black$, $g{\rightarrow}col = red$, set $t = g$

INSERTING ITEMS / 3

Insertion: case 1

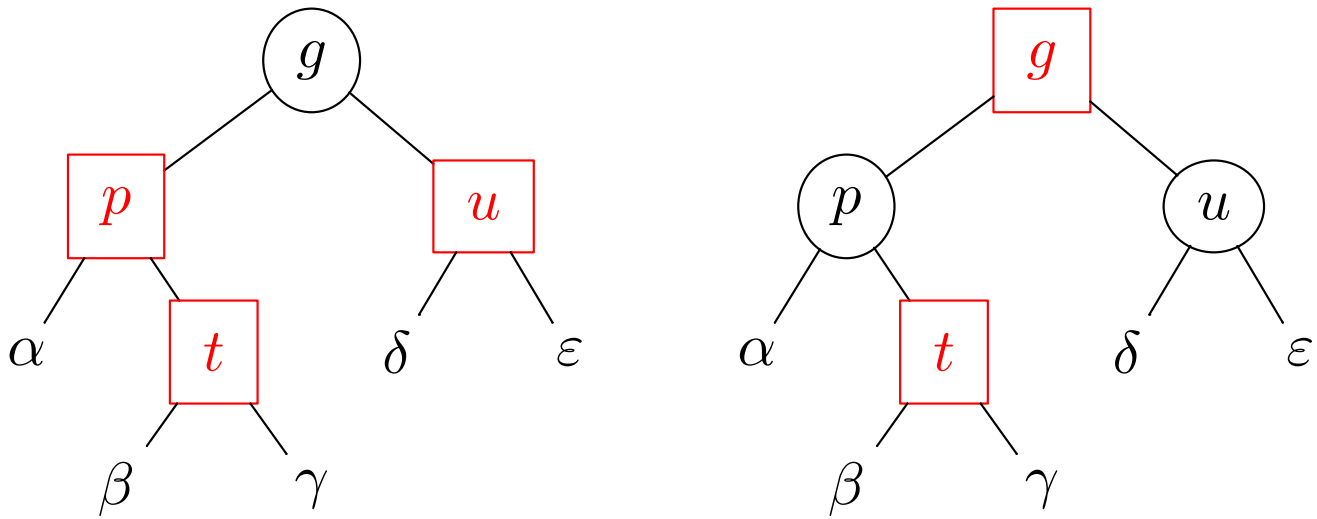
- t's uncle u is red
- Action: $p{\rightarrow}col = u{\rightarrow}col = black$, $g{\rightarrow}col = red$, set $t = g$



INSERTING ITEMS / 3

Insertion: case 1

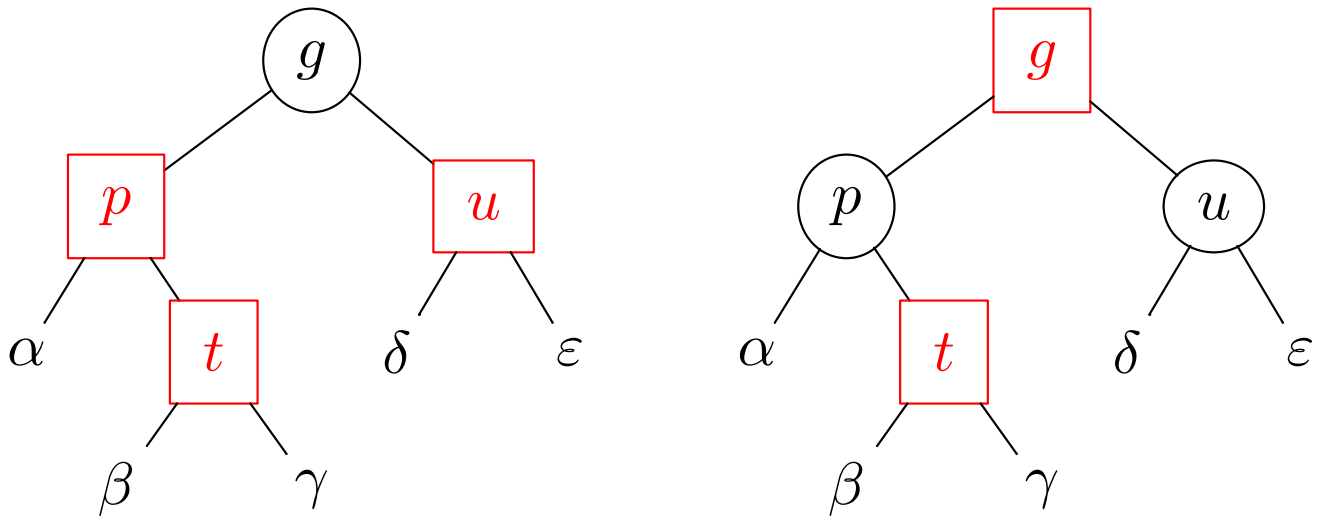
- t's uncle u is red
- Action: $p{\rightarrow}col = u{\rightarrow}col = black$, $g{\rightarrow}col = red$, set $t = g$



INSERTING ITEMS / 3

Insertion: case 1

- t's uncle u is red
- Action: $p{\rightarrow}col = u{\rightarrow}col = black$, $g{\rightarrow}col = red$, set $t = g$

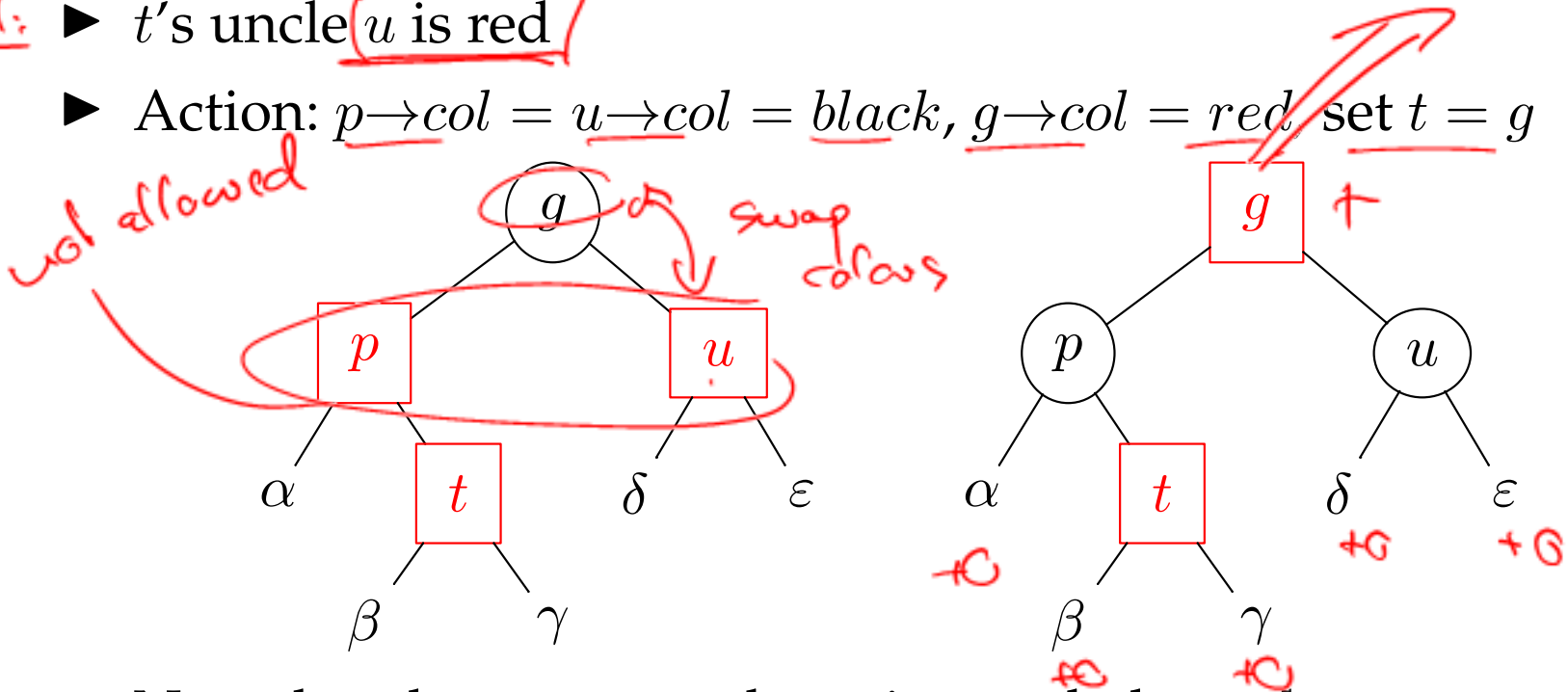


- Note that the tree rooted at g is now balanced

INSERTING ITEMS / 3

Insertion: case 1

- t's uncle u is red
- Action: $p{\rightarrow}col = u{\rightarrow}col = black$, $g{\rightarrow}col = red$, set $t = g$



- Note that the tree rooted at g is now balanced
- g is now red, so the RB-property may be violated (propagate upwards with t = g)

EXERCISE SL06-8

Analyze case 1 wrt the properties of red black trees.

1) red and black nodes: ✓

2) root is black: might be violated; just color it black

3) leaves are black: ✓ (NULL nodes are black)

4) no adjacent red nodes: fixed recursively (propagate upward)

5) same # of black nodes on all paths from root to leaf: ✓

INSERTING ITEMS / 4

Insertion: case 2

- t's uncle u is black and t is a right child

INSERTING ITEMS / 4

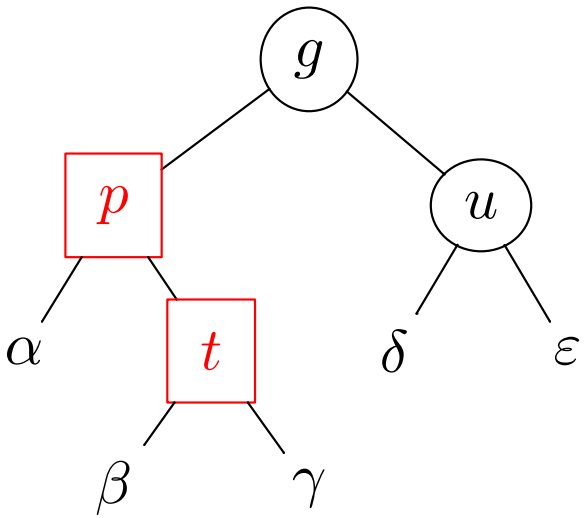
Insertion: case 2

- t's uncle u is black and t is a right child
- Action: `LeftRotate` (p), set $t = p$

INSERTING ITEMS / 4

Insertion: case 2

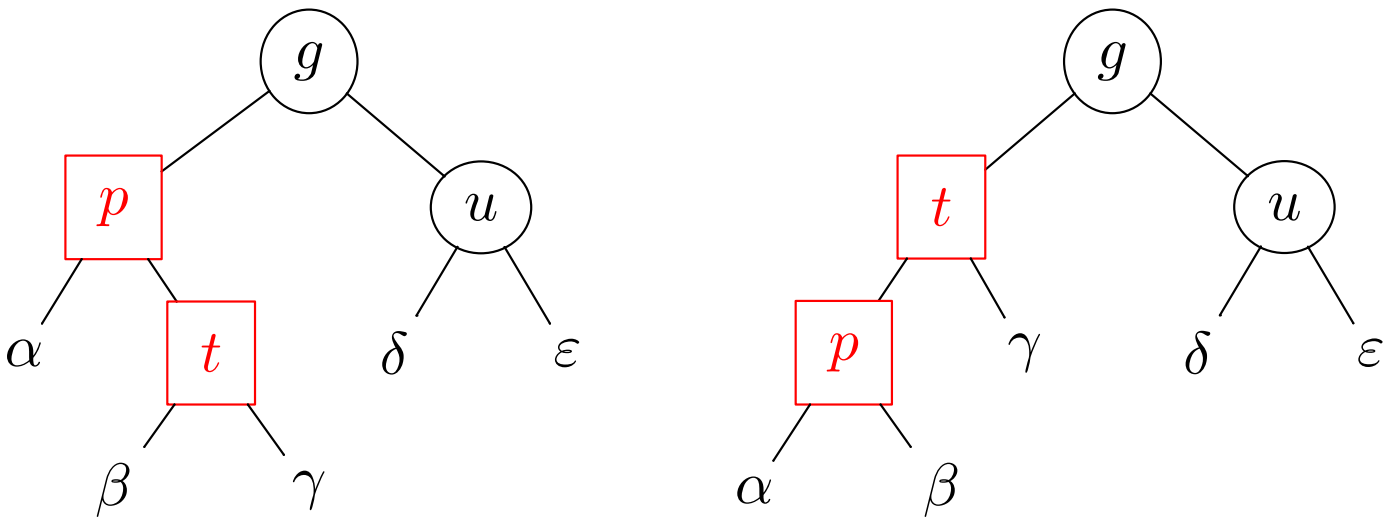
- t's uncle u is black and t is a right child
- Action: `LeftRotate` (p), set $t = p$



INSERTING ITEMS / 4

Insertion: case 2

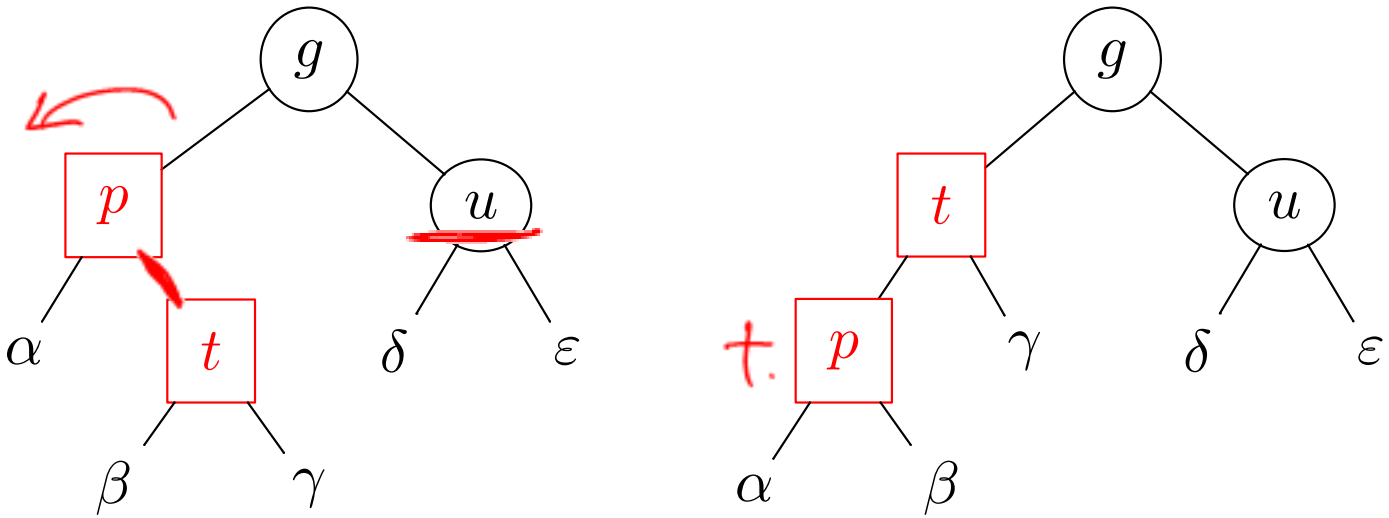
- t's uncle u is black and t is a right child
- Action: `LeftRotate` (p), set $t = p$



INSERTING ITEMS / 4

Insertion: case 2

- t's uncle u is black and t is a right child
- Action: `LeftRotate` (p), set $t = p$



- Note that in this way one gets case 3 (see below)

INSERTING ITEMS / 5

Insertion: case 3

- t's uncle u is black and t is a left child

INSERTING ITEMS / 5

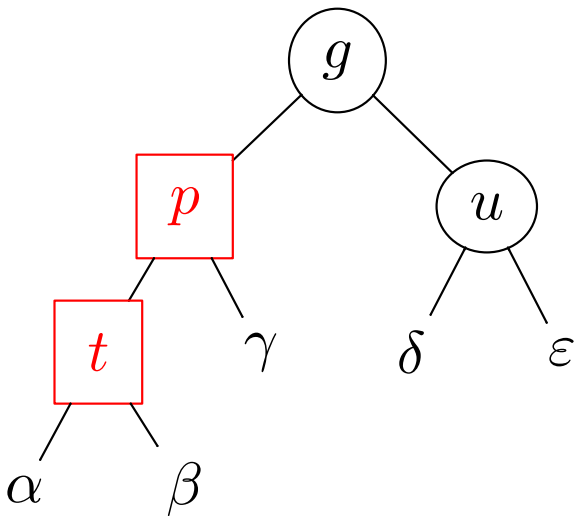
Insertion: case 3

- t's uncle u is black and t is a left child
- Action: $p{\rightarrow}col = black$, $g{\rightarrow}col = red$, `RightRotate` (g)

INSERTING ITEMS / 5

Insertion: case 3

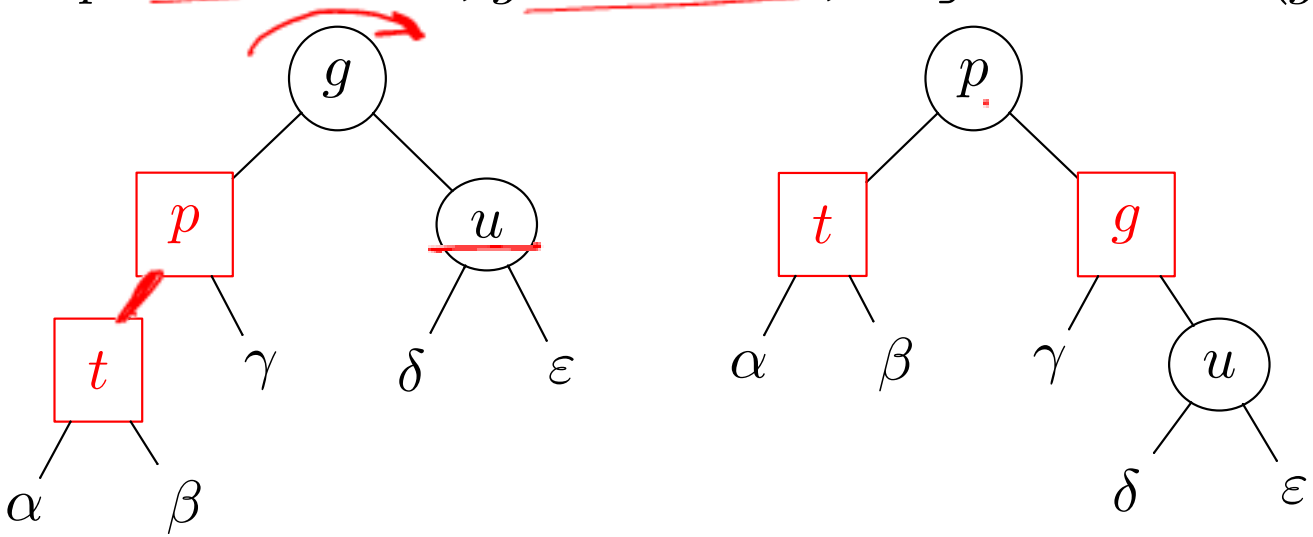
- t's uncle u is black and t is a left child
- Action: $p{\rightarrow}col = black$, $g{\rightarrow}col = red$, `RightRotate` (g)



INSERTING ITEMS / 5

Insertion: case 3

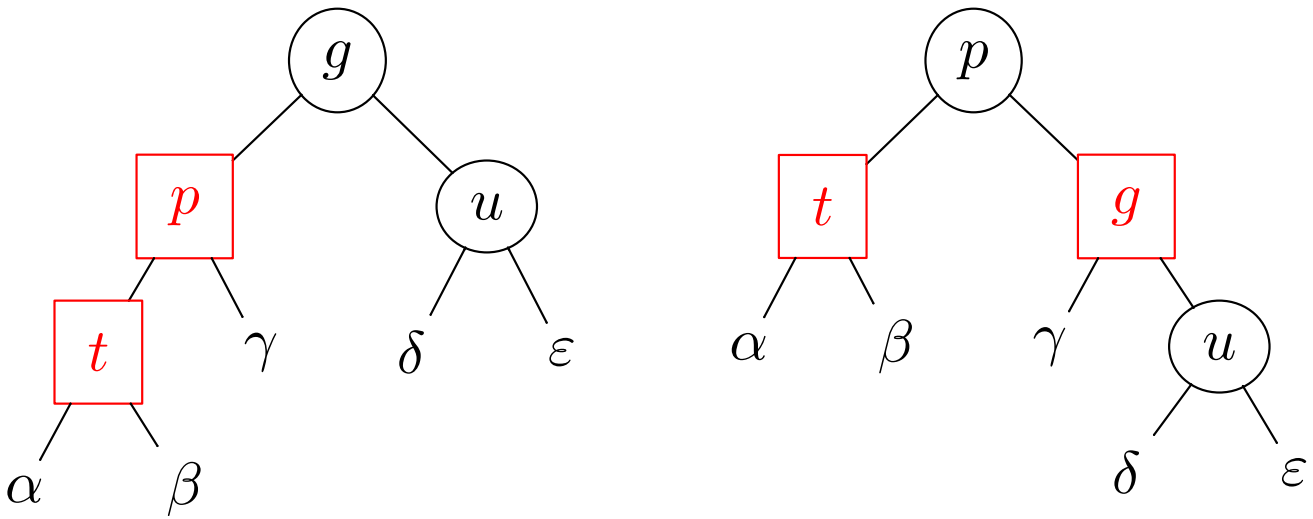
- t's uncle u is black and t is a left child
- Action: $p \rightarrow col = black$, $g \rightarrow col = red$, `RightRotate` (g)



INSERTING ITEMS / 5

Insertion: case 3

- t's uncle u is black and t is a left child
- Action: $p{\rightarrow}col = black$, $g{\rightarrow}col = red$, `RightRotate` (g)



- Note that the tree rooted at p is now balanced

INSERTING ITEMS / 6

The mirror cases

- In case that p is a right child, all the above cases are handled similarly (simply exchange "left" and "right")

INSERTING ITEMS / 6

The mirror cases

- In case that p is a right child, all the above cases are handled similarly (simply exchange "left" and "right")

Final observations

- If two red nodes are adjacent, one does either
  - restructure (with one or two rotations) and stop, or
  - propagate adjacent red nodes upwards in the tree

INSERTING ITEMS / 6

The mirror cases

- In case that p is a right child, all the above cases are handled similarly (simply exchange "left" and "right")

Final observations

- If two red nodes are adjacent, one does either
  - restructure (with one or two rotations) and stop, or
  - propagate adjacent red nodes upwards in the tree
- A restructuring takes constant time and is done at most once (it reorganizes an off-balanced part of the tree)

INSERTING ITEMS / 6

The mirror cases

- In case that p is a right child, all the above cases are handled similarly (simply exchange "left" and "right")

Final observations

- If two red nodes are adjacent, one does either
  - restructure (with one or two rotations) and stop, or
  - propagate adjacent red nodes upwards in the tree
- A restructuring takes constant time and is done at most once (it reorganizes an off-balanced part of the tree)
- The number of propagations is bound by $O(\lg n)$

INSERTING ITEMS / 6

The mirror cases

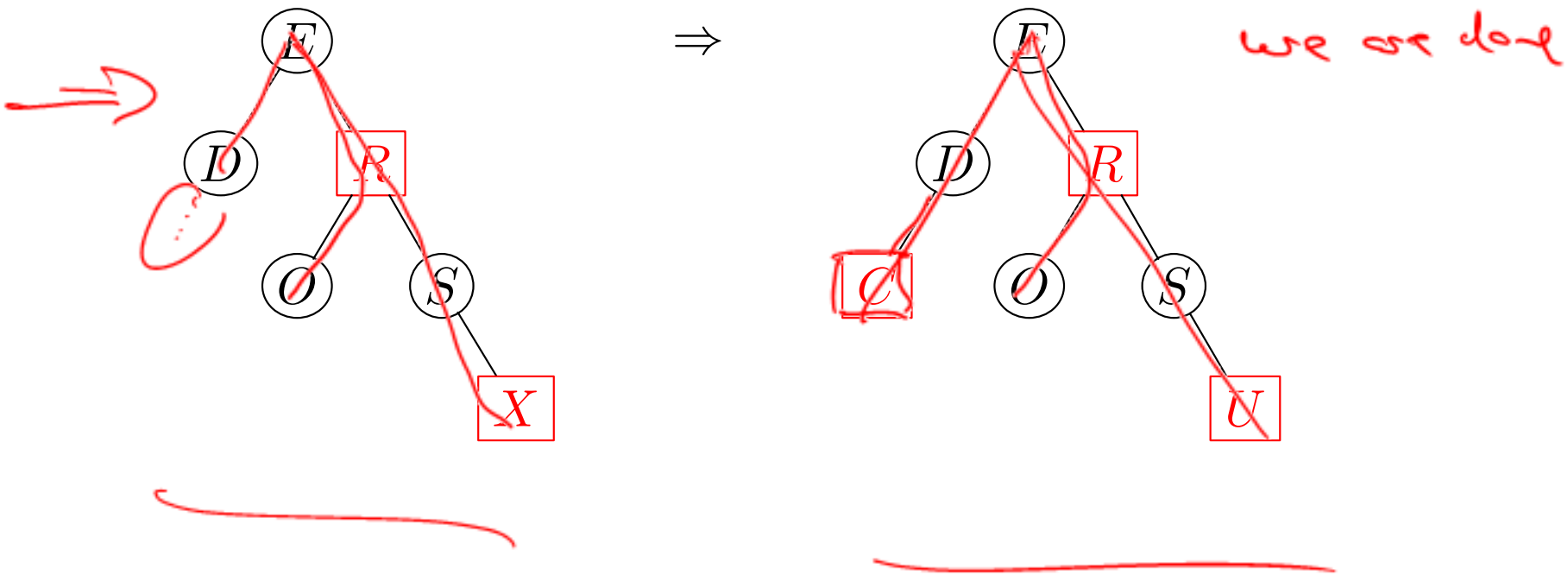
- In case that p is a right child, all the above cases are handled similarly (simply exchange "left" and "right")

Final observations

- If two red nodes are adjacent, one does either
  - restructure (with one or two rotations) and stop, or
  - propagate adjacent red nodes upwards in the tree
- A restructuring takes constant time and is done at most once (it reorganizes an off-balanced part of the tree)
- The number of propagations is bound by $O(\lg n)$
- The running time of insertion is thus $O(\lg n)$

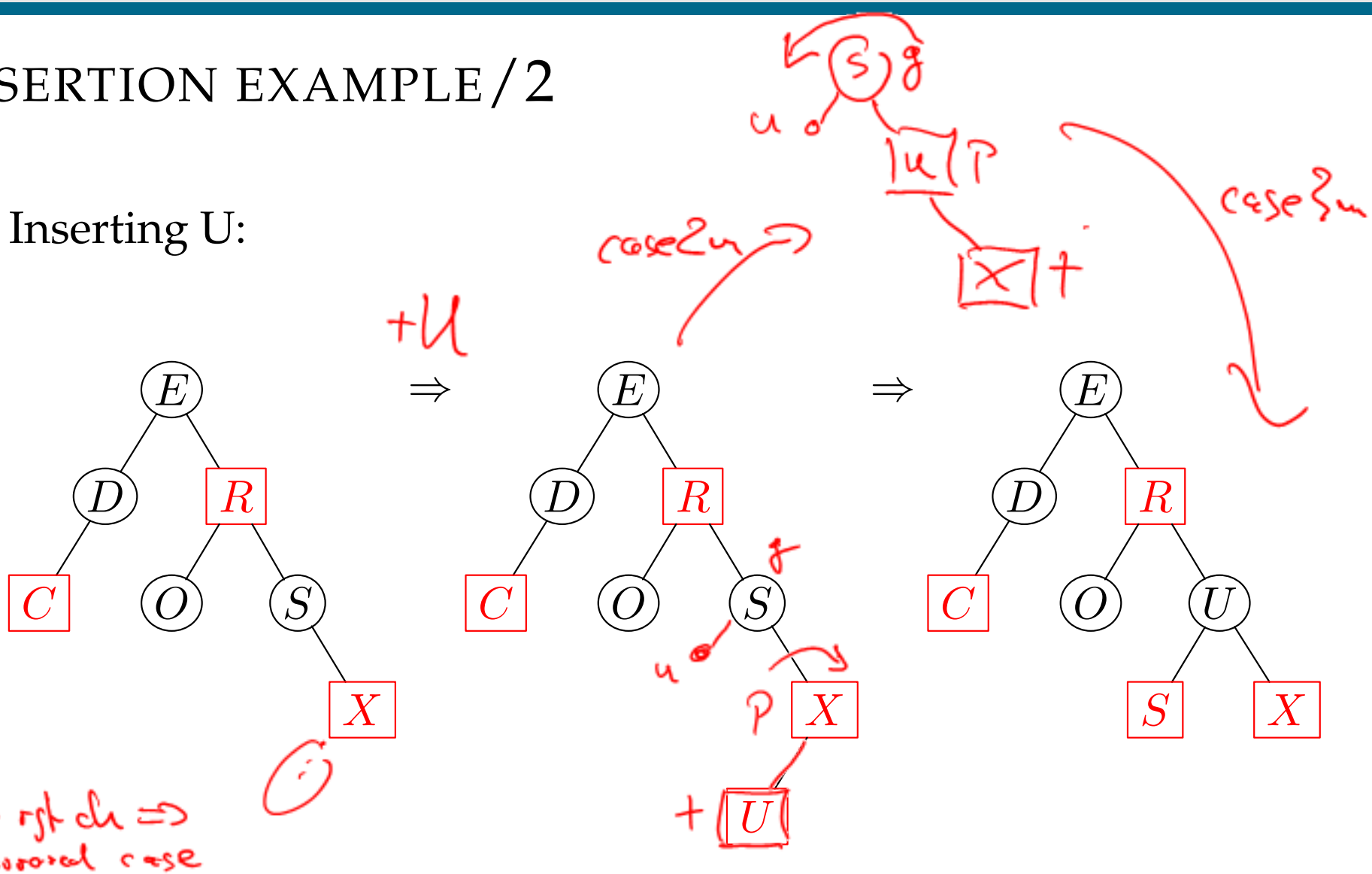
Insertion example/1

Inserting C:



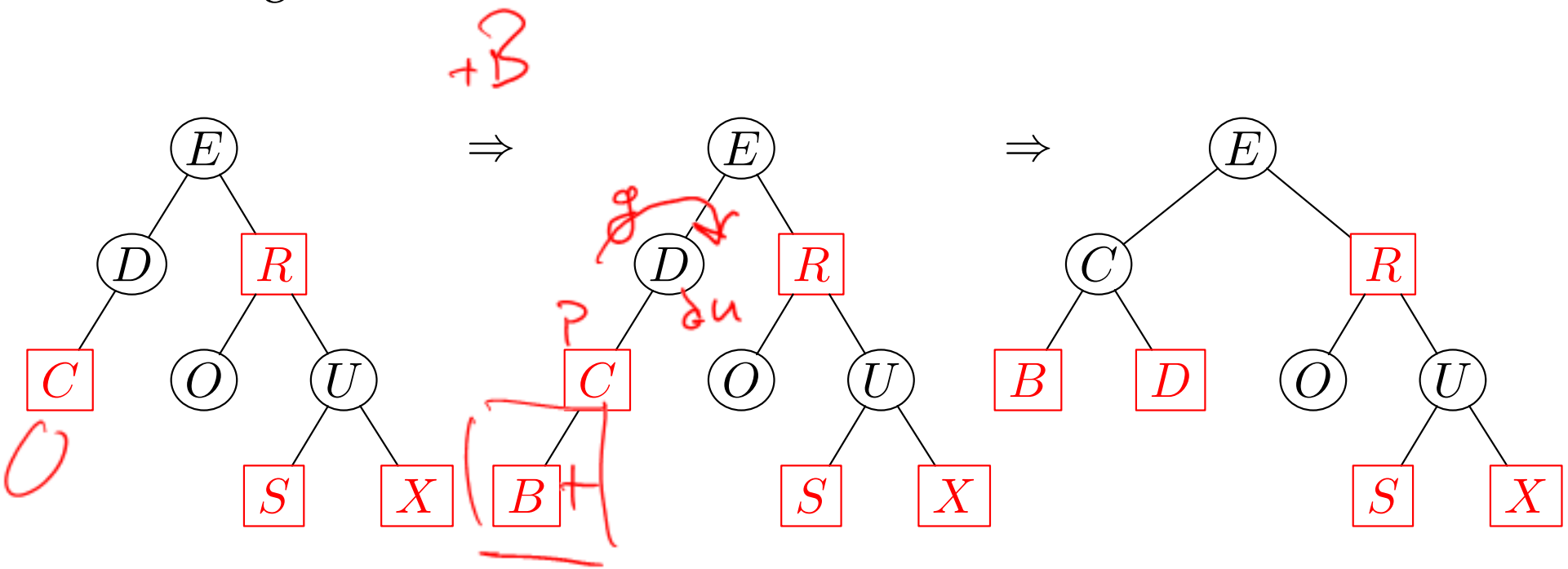
INSERTION EXAMPLE/2

Inserting U:



INSERTION EXAMPLE/3

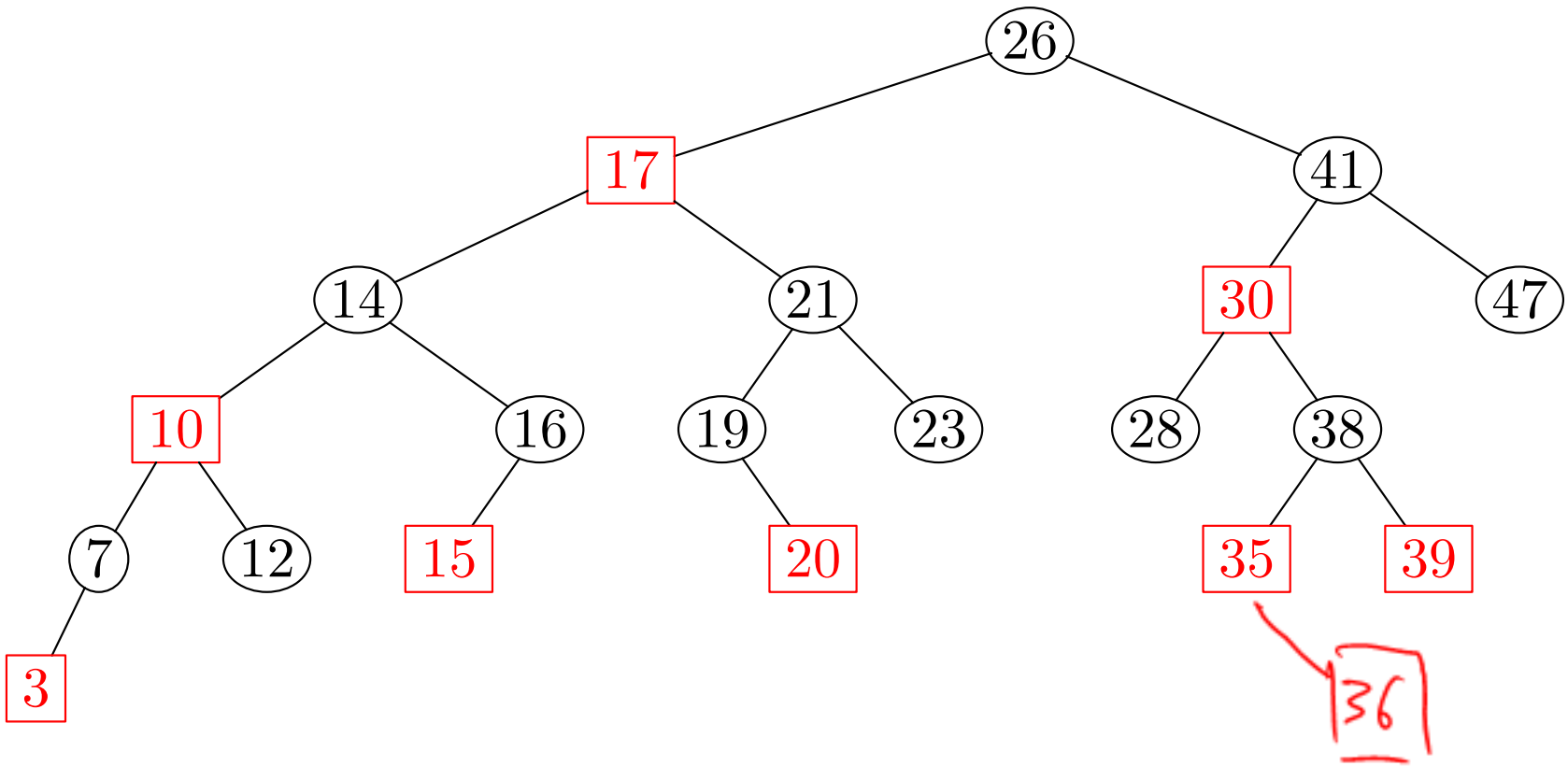
Inserting B:



case 3: u is b, & left ch ⇒ p.C = bl ; g.C = r
RR(g)

EXERCISE SL06-9/1

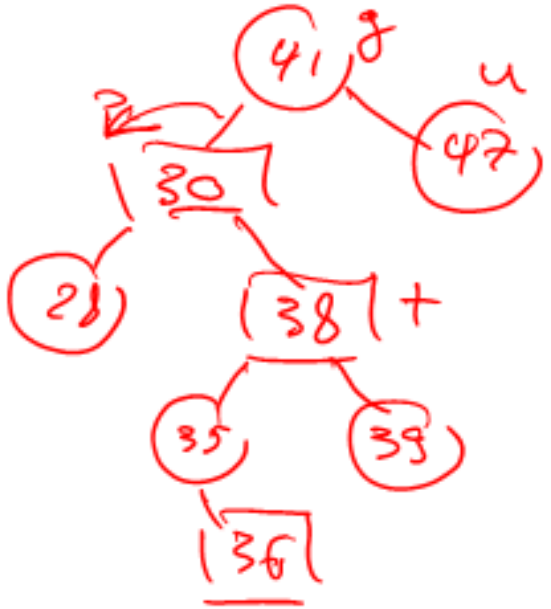
Insert 36 into the following red black tree:



EXERCISE SL06-9/2

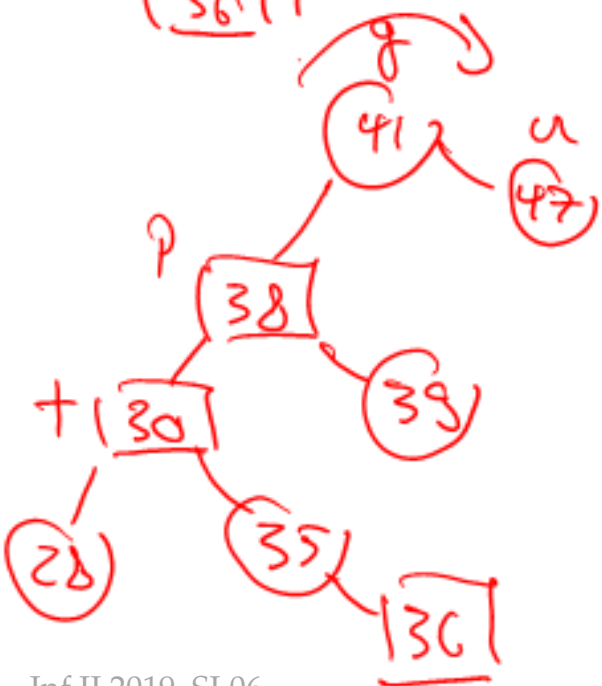


case 1: u is r
p.c = b
u.c = b
g.c = r
t = g



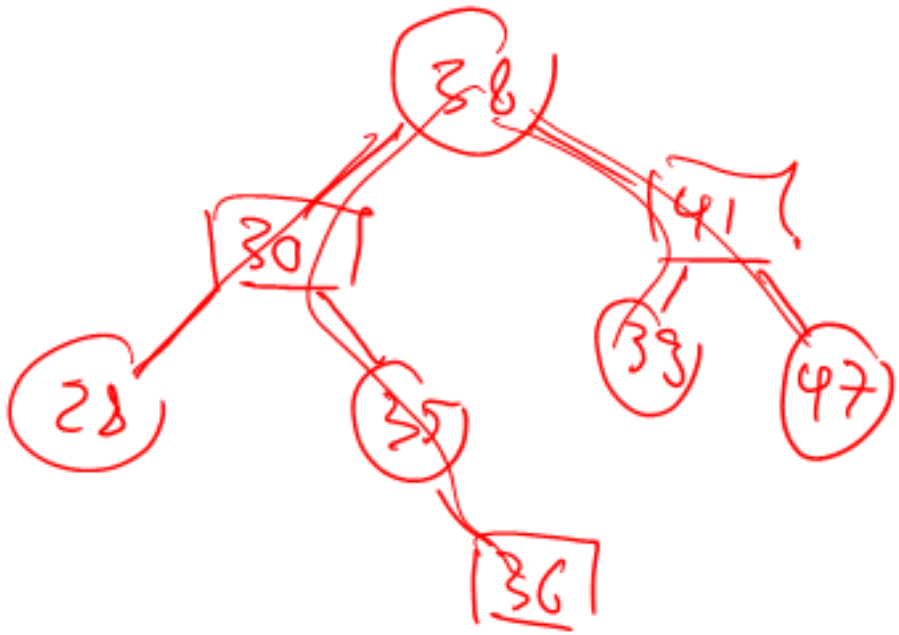
case 2:
u is bl,
t is rgt child

LR(p);
t = p



case 3:
u is bl
t is lft ch

p.c = bl
g.c = r
RR(g)

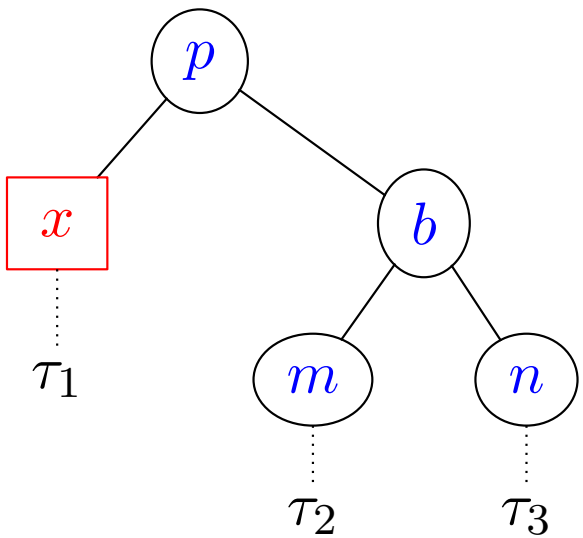


DELETING ITEMS/1

- We first apply binary search tree deletion:
  - We can easily delete a node u that has at least one NIL child.
  - If the key to be deleted is stored at a node u with two children we replace the key of u with the key of the largest node v of the left subtree of u and delete v instead.
- Deletion of node s (s is either u or v)
  - Delete s and replace it by it's child x
  - If color of s was red we are done.
  - If color of s was black we proceed according to the cases described below

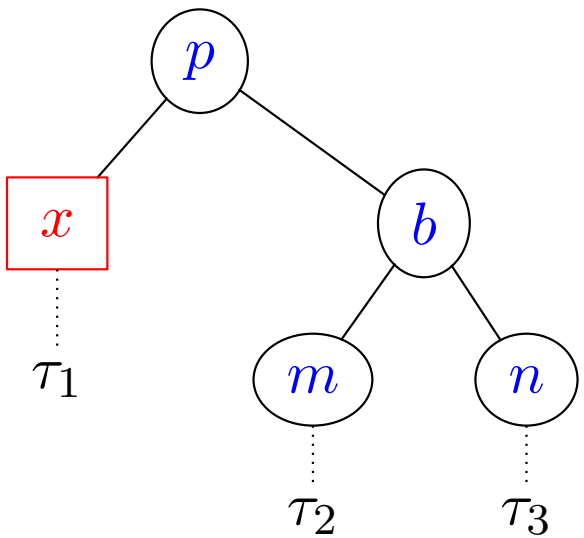
DELETING ITEMS/2

Deletion: case 0



DELETING ITEMS/2

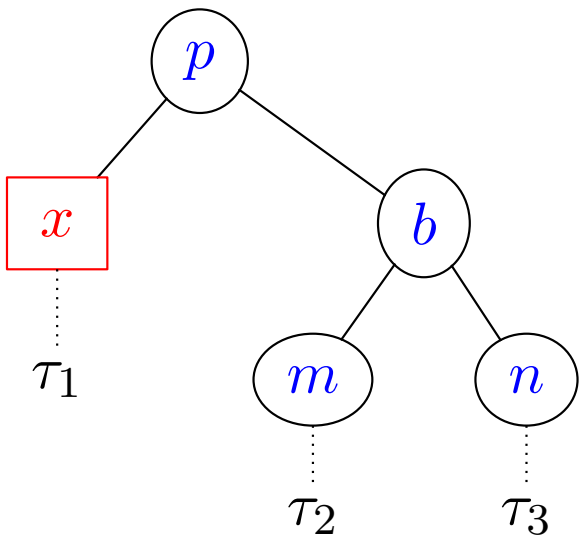
Deletion: case 0



- Assumptions
  - x is a left child
  - x replaces the deleted node s

DELETING ITEMS/2

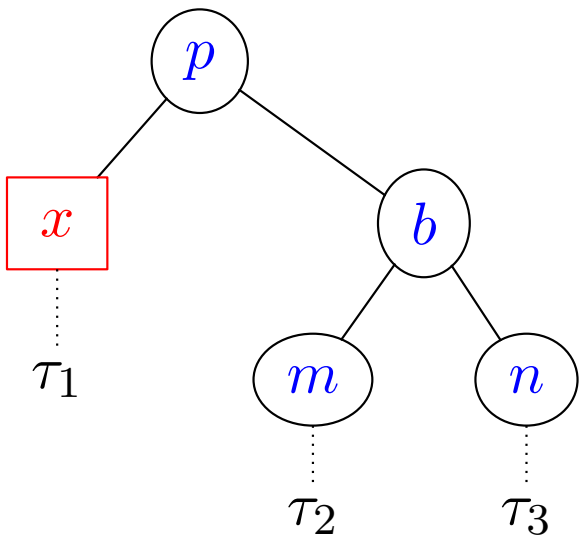
Deletion: case 0



- Assumptions
  - x is a left child
  - x replaces the deleted node s
- Conventions
  - p — parent of x
  - b — brother of x
  - m,n — children of b

DELETING ITEMS/2

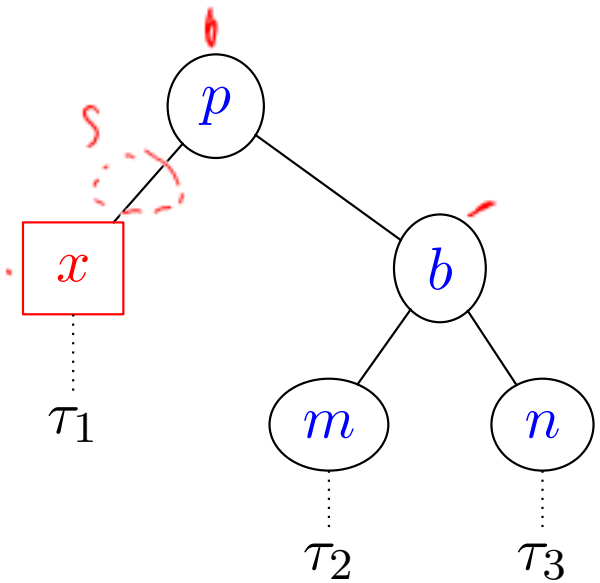
Deletion: case 0



- Assumptions
  - x is a left child
  - x replaces the deleted node s
- Conventions
  - p — parent of x
  - b — brother of x
  - m,n — children of b
- Color of deleted node was red
  - There is nothing to do

Deleting items/2

Deletion: case 0



- Assumptions
  - x is a left child
  - x replaces the deleted node s
- Conventions
  - p — parent of x
  - b — brother of x
  - m,n — children of b
- Color of deleted node was red
  - There is nothing to do
- Color of deleted node was black
  - If color of x is red then simply set it to black

DELETING ITEMS/3

Deletion: case 1

- x's brother b is red

DELETING ITEMS / 3

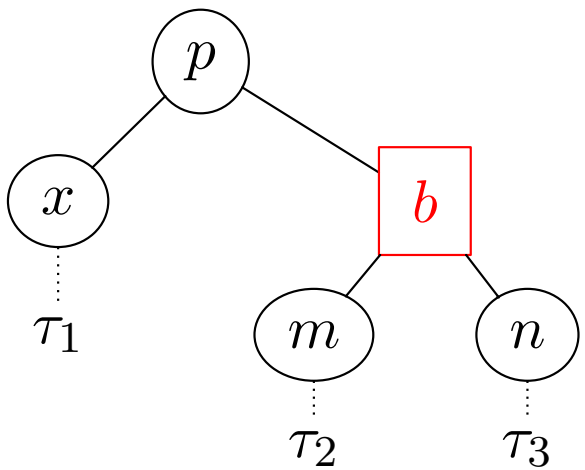
Deletion: case 1

- x's brother b is red
- Action: $b{\rightarrow}col = black$, $p{\rightarrow}col = red$, `LeftRotate` (p), set $b = m$

DELETING ITEMS/3

Deletion: case 1

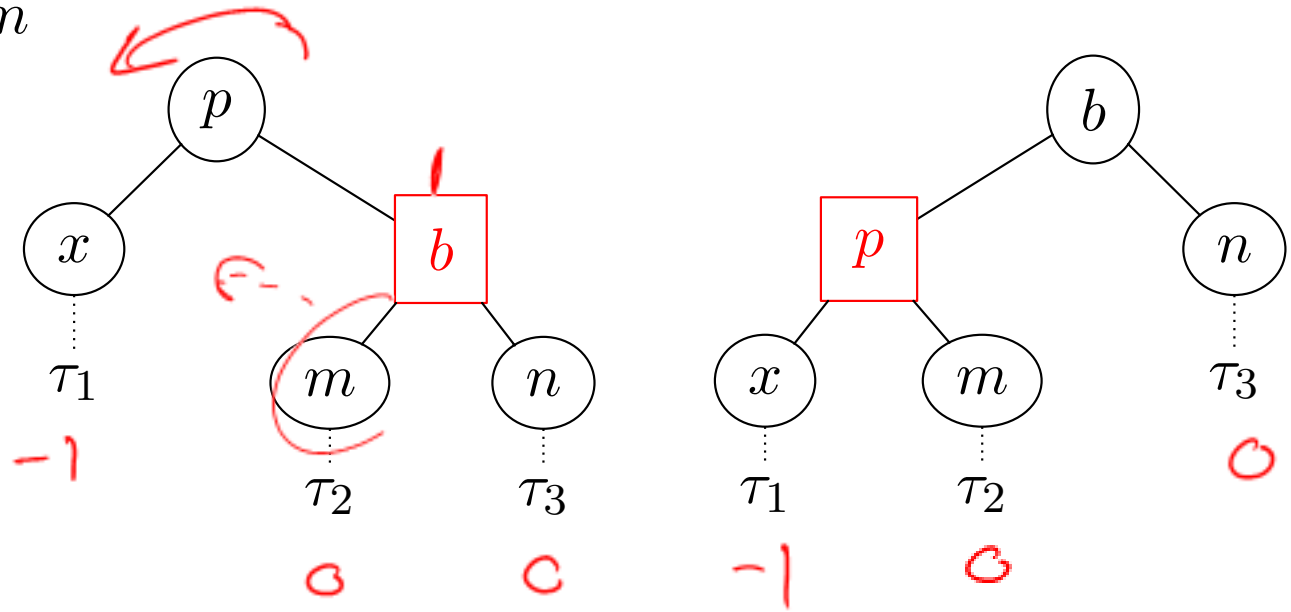
- x's brother b is red
- Action: $b{\rightarrow}col = black$, $p{\rightarrow}col = red$, `LeftRotate` (p), set $b = m$



DELETING ITEMS / 3

Deletion: case 1

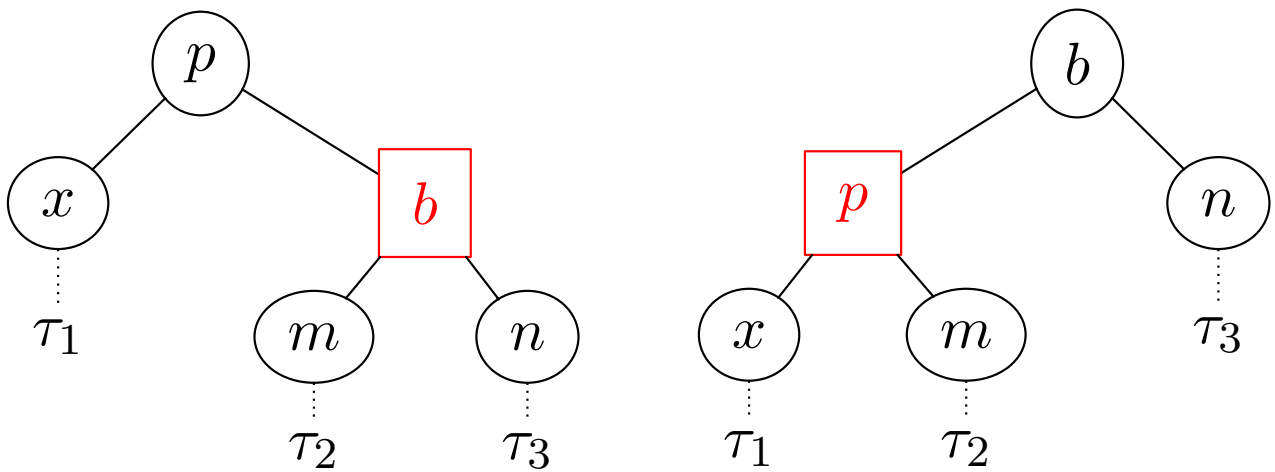
- x's brother b is red
  - Action: $b{\rightarrow}col = black$, $p{\rightarrow}col = red$, `LeftRotate` (p), set $b = m$



DELETING ITEMS / 3

Deletion: case 1

- x's brother b is red
- Action: $b{\rightarrow}col = black$, $p{\rightarrow}col = red$, `LeftRotate` (p), set $b = m$



- Note that one gets either case 2, 3, or 4 (see below)

DELETING ITEMS/4

Deletion: case 2

- x's brother b is black and both m and n are black

DELETING ITEMS / 4

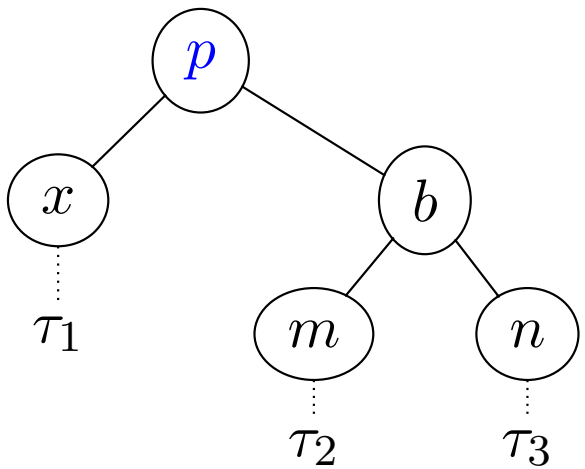
Deletion: case 2

- x's brother b is black and both m and n are black
- Action: $b{\rightarrow}col = red$, set $x = p$

DELETING ITEMS / 4

Deletion: case 2

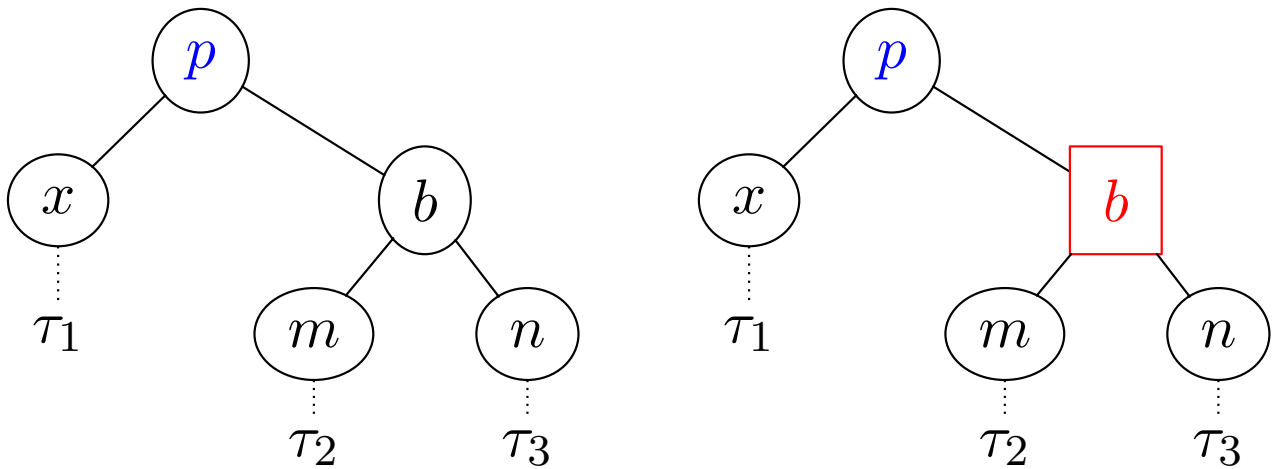
- x's brother b is black and both m and n are black
- Action: $b{\rightarrow}col = red$, set $x = p$



DELETING ITEMS / 4

Deletion: case 2

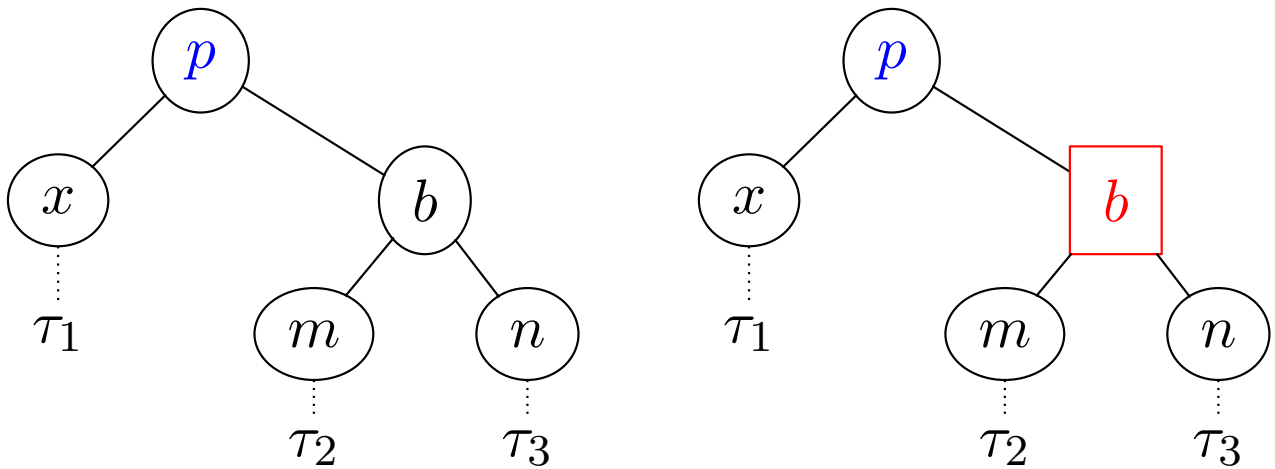
- x's brother b is black and both m and n are black
- Action: $b{\rightarrow}col = red$, set $x = p$



DELETING ITEMS / 4

Deletion: case 2

- x's brother b is black and both m and n are black
- Action: $b{\rightarrow}col = red$, set $x = p$

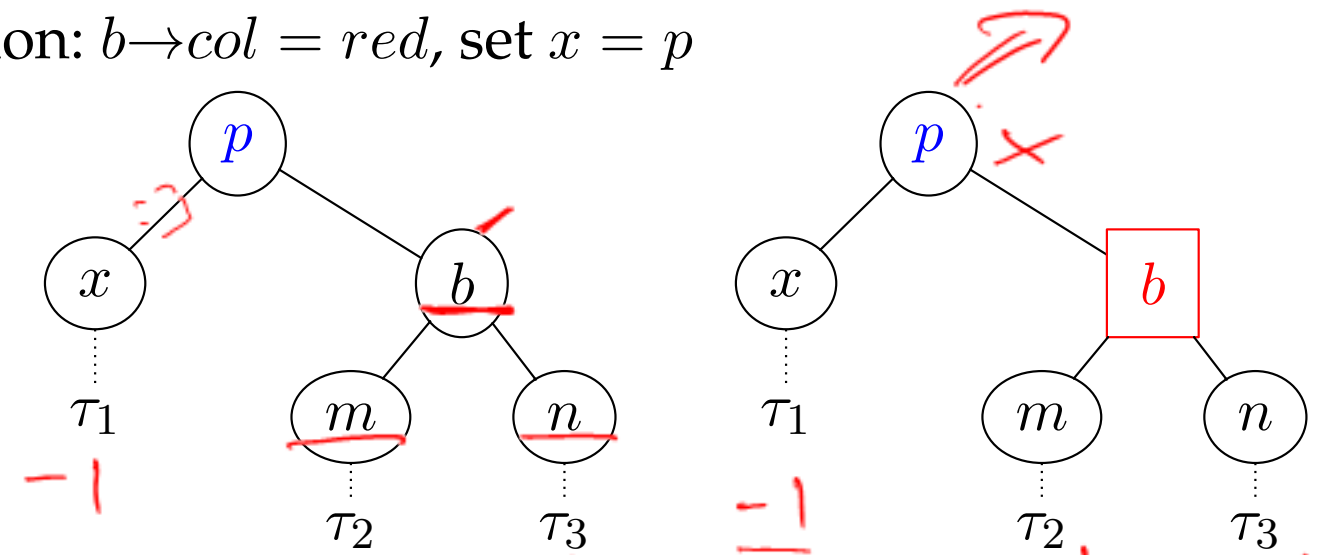


- Note that bh of both subtrees of p is reduced by one

DELETING ITEMS / 4

Deletion: case 2

- x's brother b is black and both m and n are black
- Action: $b{\rightarrow}col = red$, set $x = p$



- Note that bh of both subtrees of p is reduced by one
- Stop deletion if color of p is red and set $p{\rightarrow}col = black$

DELETING ITEMS/5

Deletion: case 3

- x's brother b is black, m is red, and n is black

DELETING ITEMS / 5

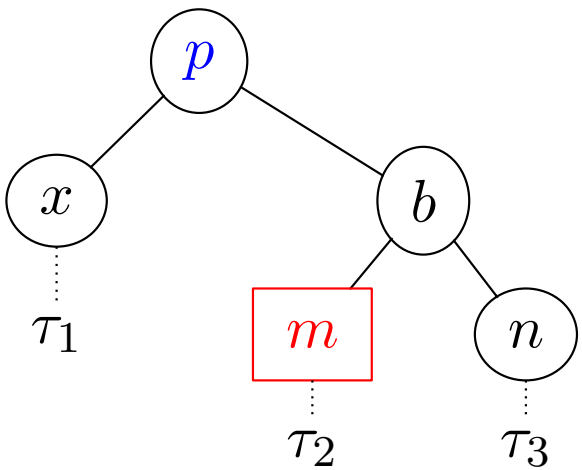
Deletion: case 3

- x's brother b is black, m is red, and n is black
- Action: $m{\rightarrow}col = black$, $b{\rightarrow}col = red$, `RightRotate` (b), set $b = m$

DELETING ITEMS / 5

Deletion: case 3

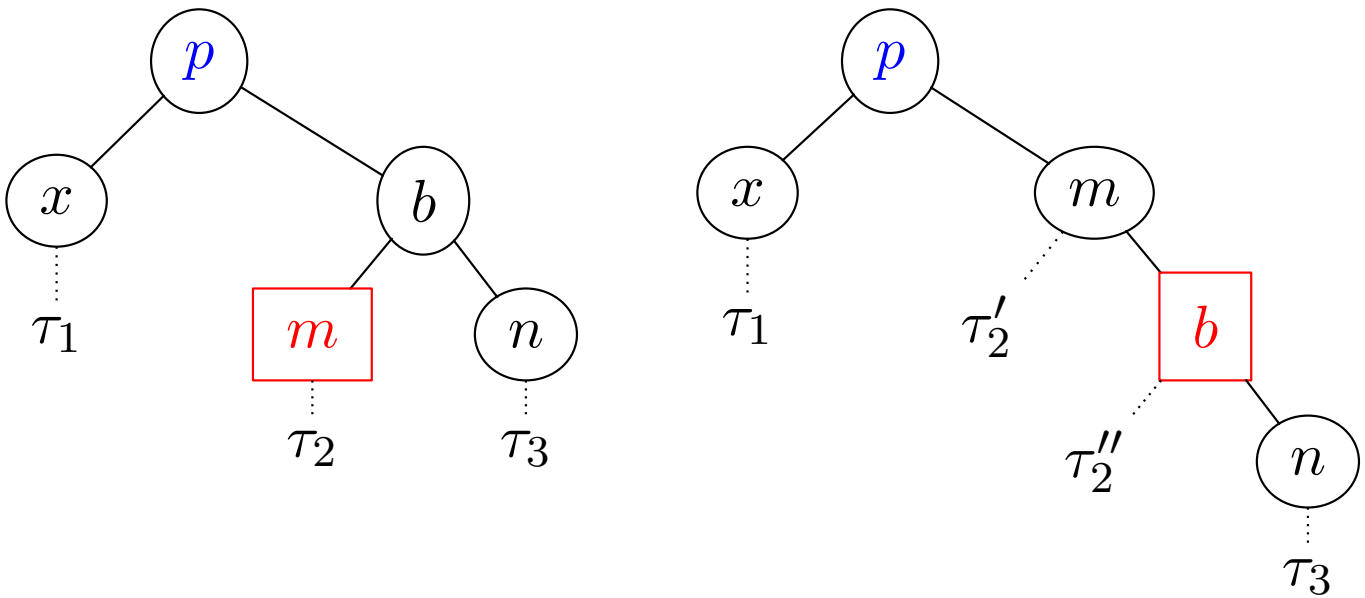
- x's brother b is black, m is red, and n is black
- Action: $m{\rightarrow}col = black$, $b{\rightarrow}col = red$, `RightRotate` (b), set $b = m$



DELETING ITEMS / 5

Deletion: case 3

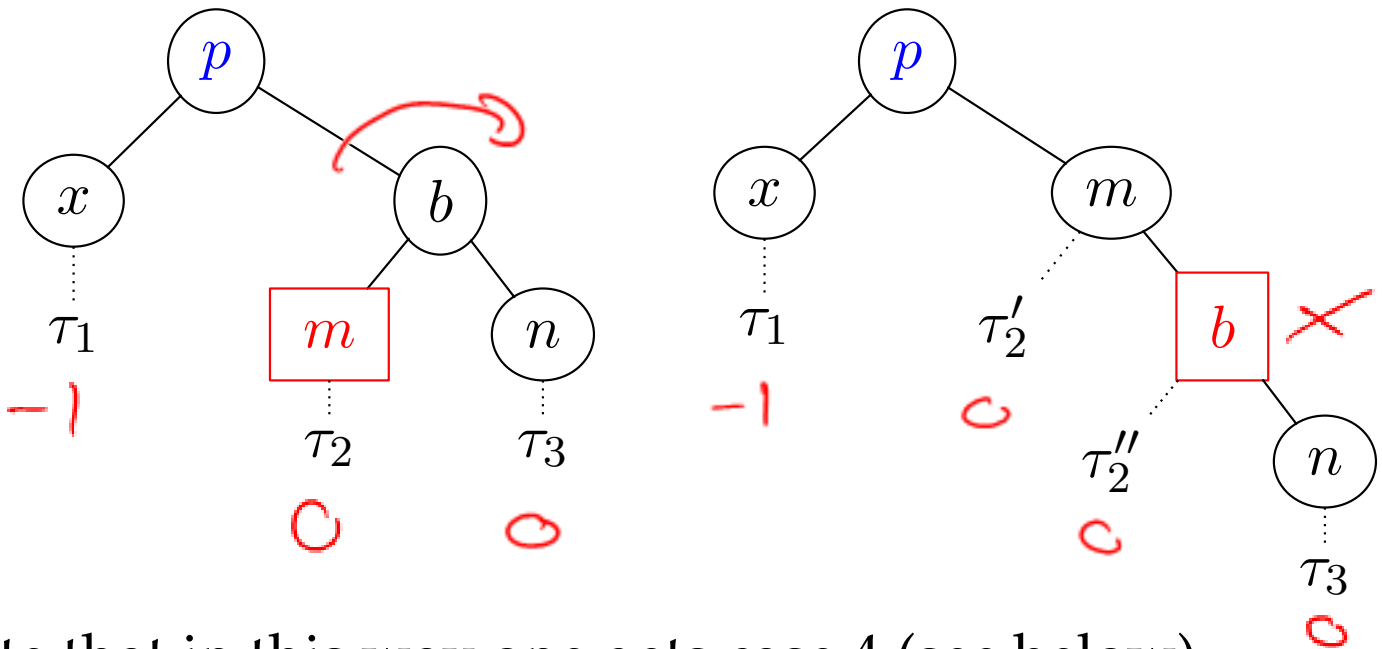
- x's brother b is black, m is red, and n is black
- Action: $m{\rightarrow}col = black$, $b{\rightarrow}col = red$, `RightRotate` (b), set $b = m$



DELETING ITEMS / 5

Deletion: case 3

- x's brother b is black, m is red, and n is black
- Action: $m{\rightarrow}col = black$, $b{\rightarrow}col = red$, `RightRotate` (b), set $b = m$



- Note that in this way one gets case 4 (see below)

DELETING ITEMS/6

Deletion: case 4

- x's brother b is black and b's right child n is red

DELETING ITEMS / 6

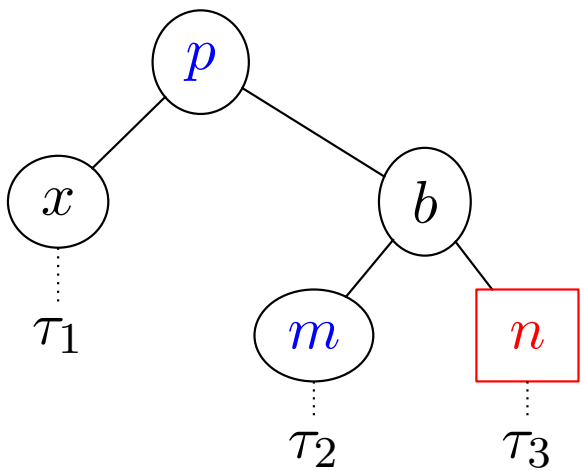
Deletion: case 4

- x's brother b is black and b's right child n is red
- Action: $b{\rightarrow}col = p{\rightarrow}col$, $p{\rightarrow}col = n{\rightarrow}col = black$, `LeftRotate` (p)

DELETING ITEMS / 6

Deletion: case 4

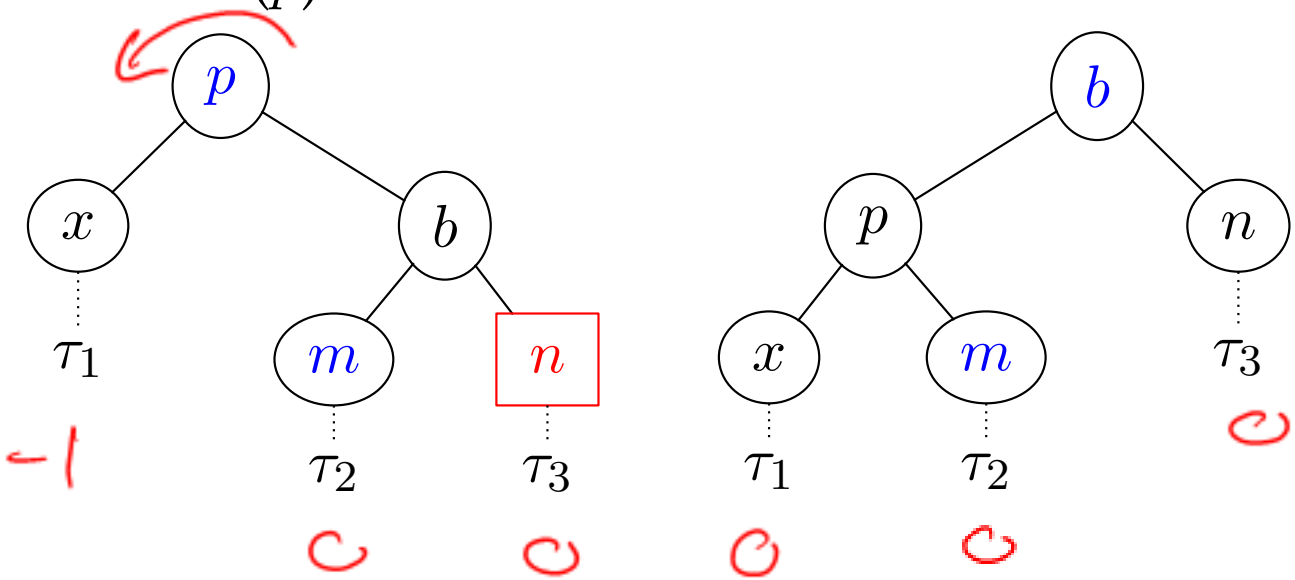
- x's brother b is black and b's right child n is red
- Action: $b{\rightarrow}col = p{\rightarrow}col$, $p{\rightarrow}col = n{\rightarrow}col = black$, `LeftRotate` (p)



DELETING ITEMS / 6

Deletion: case 4

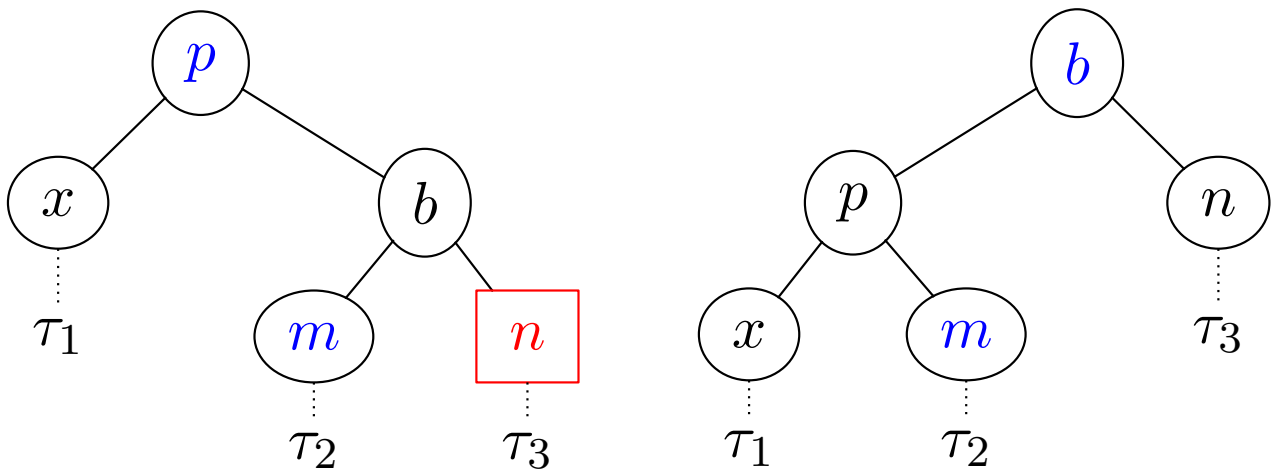
- x's brother b is black and b's right child n is red
- Action: $b{\rightarrow}col = p{\rightarrow}col$, $p{\rightarrow}col = n{\rightarrow}col = black$, `LeftRotate` (p)



DELETING ITEMS / 6

Deletion: case 4

- x's brother b is black and b's right child n is red
- Action: $b{\rightarrow}col = p{\rightarrow}col$, $p{\rightarrow}col = n{\rightarrow}col = black$, `LeftRotate` (p)



- Note that the tree rooted at b is now balanced

DELETING ITEMS/7

The mirror cases

- In case that x is a right child, all the above cases are handled similarly (simply exchange "left" and "right")

DELETING ITEMS/7

The mirror cases

- In case that x is a right child, all the above cases are handled similarly (simply exchange "left" and "right")

Final observations

- Intuitive idea is to perform a color compensation
  - Find a red node nearby and change its color into black

DELETING ITEMS/7

The mirror cases

- In case that x is a right child, all the above cases are handled similarly (simply exchange "left" and "right")

Final observations

- Intuitive idea is to perform a color compensation
  - Find a red node nearby and change its color into black
- There are two cases: (a) restructuring, (b) recoloring
  - (a) resolves problem locally, (b) propagates it upwards

DELETING ITEMS/7

The mirror cases

- In case that x is a right child, all the above cases are handled similarly (simply exchange "left" and "right")

Final observations

- Intuitive idea is to perform a color compensation
  - Find a red node nearby and change its color into black
- There are two cases: (a) restructuring, (b) recoloring
  - (a) resolves problem locally, (b) propagates it upwards
- The number of recolorings is bound by $O(\lg n)$

Deleting items/7

The mirror cases

- In case that x is a right child, all the above cases are handled similarly (simply exchange "left" and "right")

Final observations

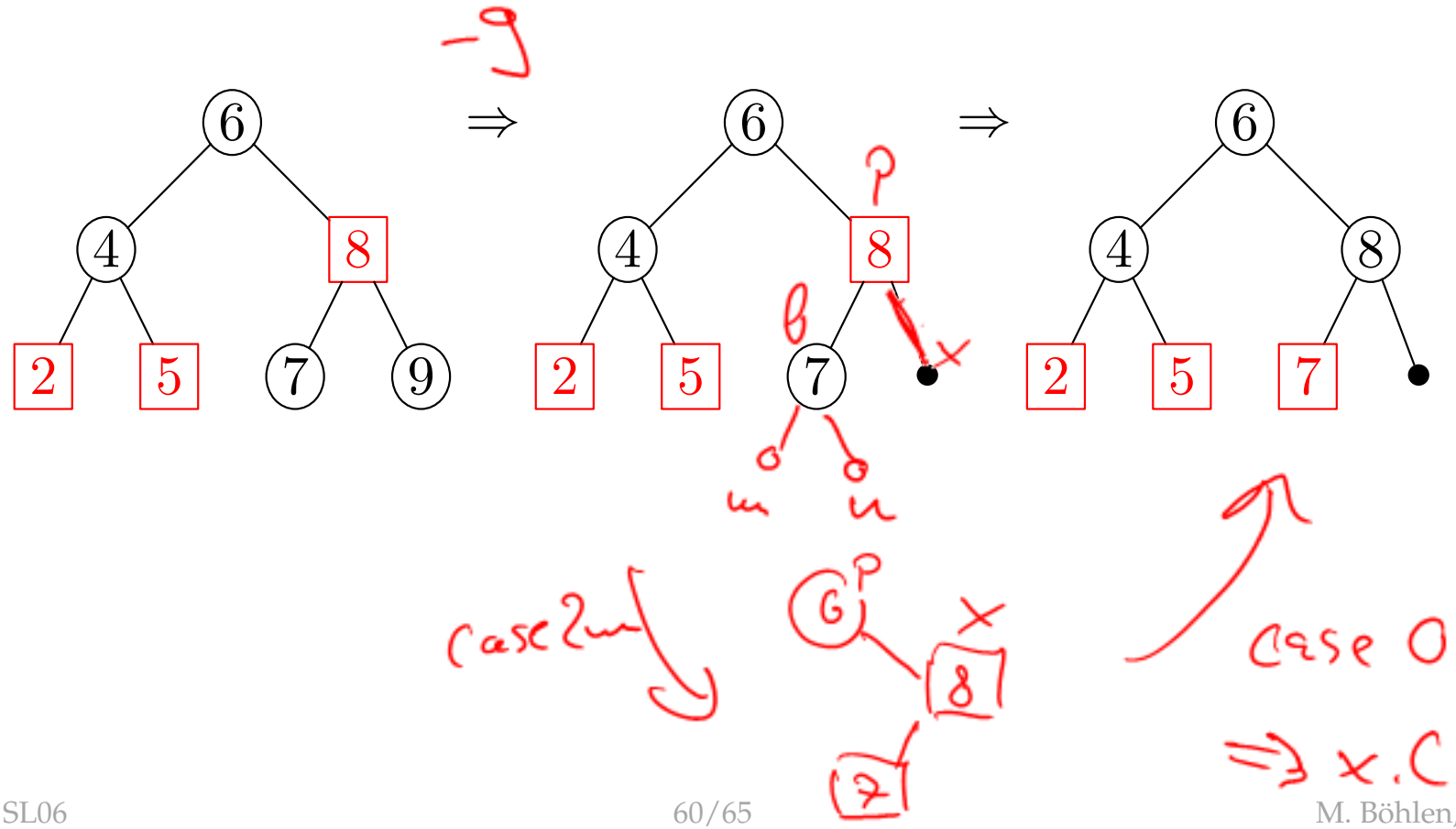
- Intuitive idea is to perform a color compensation
  - Find a red node nearby and change its color into black
- There are two cases: (a) restructuring, (b) recoloring
  - (a) resolves problem locally, (b) propagates it upwards
- The number of recolorings is bound by $O(\lg n)$
- The running time of deletion is thus $O(\lg n)$

Deletion example/1

x rgt ch ⇒ minimal case1

case 2m: p.C = r ; x = p

Deleting 9:



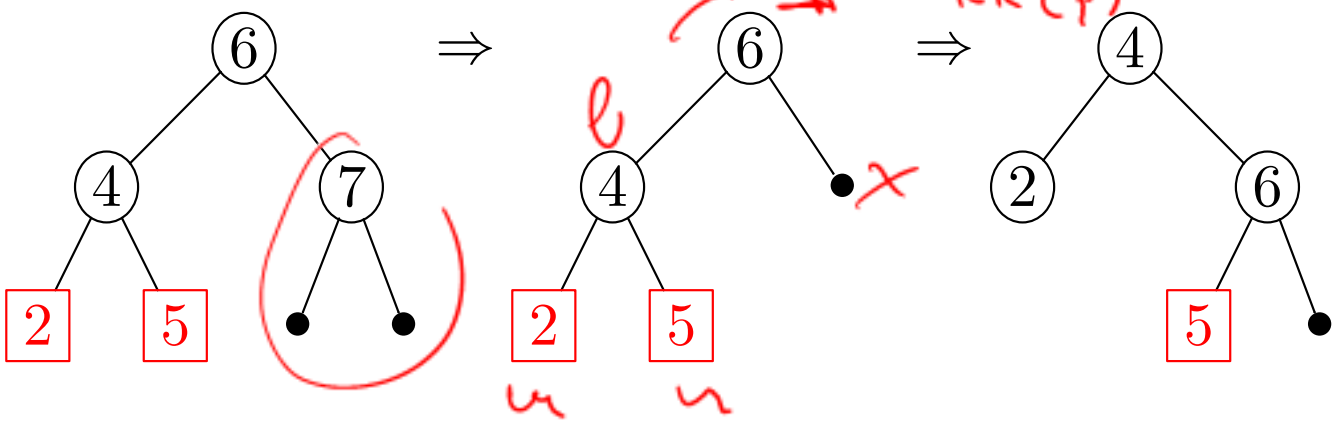
Deletion example/2

x is rgt ch ⇒ mirror case

case 4m: p is p1, u is r

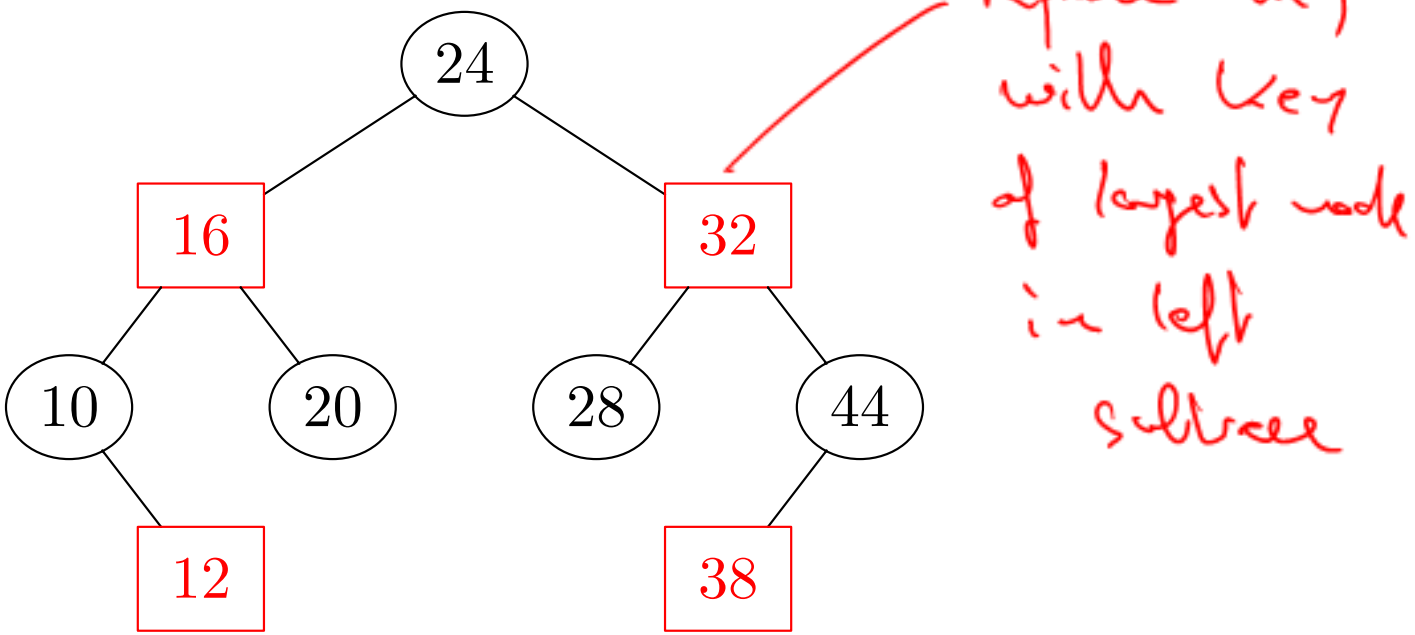
⇒ p.c = u.c
p.c = p1
u.c = p1
RR(p)

Deleting 7:



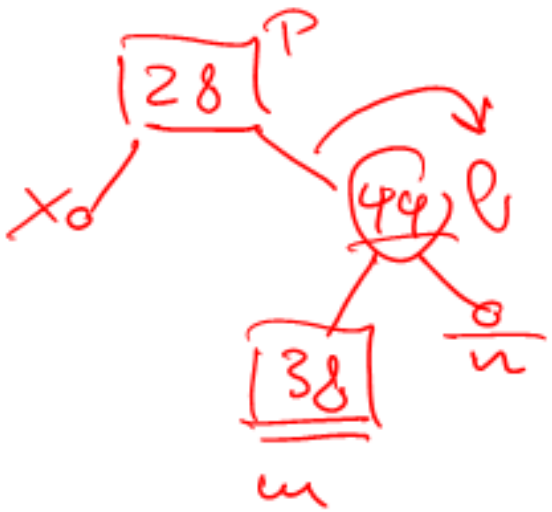
EXERCISE SL06-10/1

Consider the following red-black tree.



Delete the node with key 32 and draw the resulting tree.

EXERCISE SL06-10/2



case 3:

b is Bl, u is r, n is Bl

u.C = Bl

b.C = r

RR(B)

B = u



case 4

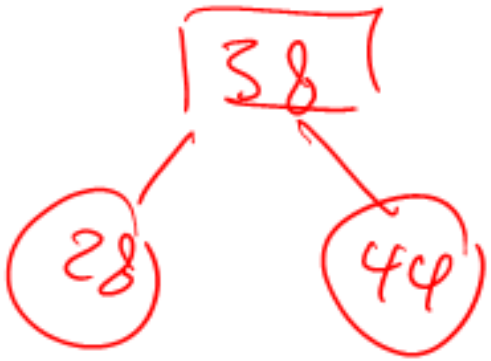
b is l, u is r

⟹ b.C = p.C

p.C = b

u.C = b

LR(p)



SUMMARY / 1

Binary trees

- In a binary tree each node has up to two children.
- A binary search tree is a binary tree where
  - in the left subtree of a node x all nodes have keys less or equal to the key of node x
  - in the right subtree of a node x all nodes have keys larger or equal to the key of node x
- Binary search trees offer $log(n)$ performance for search, insertion and deletion if the tree is balanced.
- In the worst case binary trees degenerate to lists with linear performance.

SUMMARY / 2

Red-black trees

- Red-black trees are binary search trees that are balanced.
- Red-black trees provide $log(n)$ worst case guarantees for search, insertion and deletion.
- Red-black trees are very popular in practice because of the worst-case guarantee.
- There are alternative balanced trees (sometimes very close to red-black trees):
  - AA trees
  - 2-3 trees
  - 2–3–4 tree (also called a 2–4 tree)
  - AVL trees

Informatik II

Hash tables

— SL07 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

TABLE OF CONTENTS — SL07

1. Hash tables
   Dictionaries
   Hash tables

2. Hash functions

3. Collision resolution

DICTIONARIES/1



- Dictionaries store elements so that they can be located quickly using keys.
- A dictionary may organize and manage bank accounts.
  - Each account is an object that is identified by an account number.
  - Each account stores a lot of additional information.
  - An application wishing to operate on an account would have to provide the account number as a search key.
  - Accesses without the search key (e.g., account number, personnel number) are poorly supported.

DICTIONARIES/2

no order — equality

- If order (methods such as min, max, successor, predecessor) is not required it is enough to check for equality.
- Operations that require ordering are still possible (e.g., scanning the entire database) but cannot use the dictionary access structure.
  - Usually all elements must be compared, which is slow.
  - Can be OK if it is rare enough.

DICTIONARIES / 3

- AT&T is a large phone company, and they want to provide caller ID capability:
  - given a phone number, return the caller's name
  - phone numbers range from 0 to r = $10^8 - 1$
  - want to do this as efficiently as possible
- Examples where usually no ordering is required:
  - AppleID
  - Versicherungsnummer
  - Residential addresses
- Examples where ordering could be relevant:
  - IP addresses
  - ISBN number
  - S-Number
  - Names

DICTIONARIES / 4

Dictionaries revisited

- BST: insert, search, and delete in $O(h)$ time

DICTIONARIES / 4

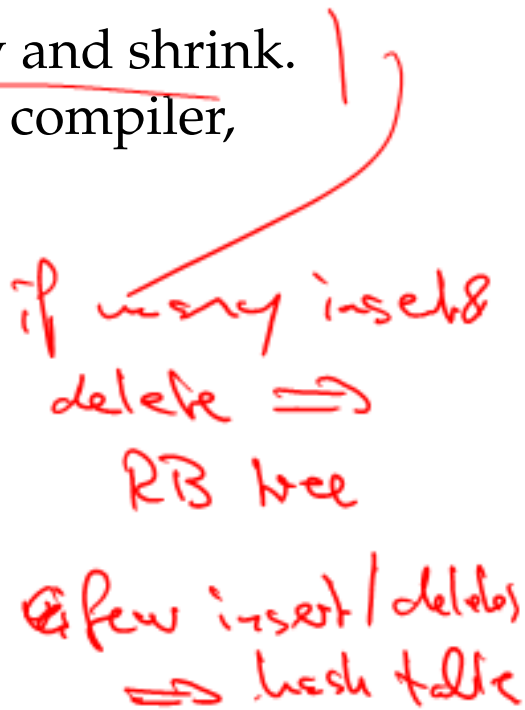
Dictionaries revisited

- BST: insert, search, and delete in $O(h)$ time
- RBT: insert, search, and delete in $O(\lg n)$ time

DICTIONARIES / 4

Dictionaries revisited

- BST: insert, search, and delete in $O(h)$ time
- RBT: insert, search, and delete in $O(\lg n)$ time
- Hashing: insert, search, and delete in $O(1)$ time
  - Other dictionary operations (min, succ, ...) are not supported
  - Hash table is expensive and difficult to grow and shrink.
  - Application: symbol table for identifiers in a compiler, customer IDs, ...



DICTIONARIES/4

Dictionaries revisited

- BST: insert, search, and delete in $O(h)$ time
- RBT: insert, search, and delete in $O(\lg n)$ time
- Hashing: insert, search, and delete in $O(1)$ time
  - Other dictionary operations (min, succ, ...) are not supported
  - Hash table is expensive and difficult to grow and shrink.
  - Application: symbol table for identifiers in a compiler, customer IDs, ...

Hash tables vs arrays

- Hash tables are a generalization of arrays

DICTIONARIES/4

Dictionaries revisited

- BST: insert, search, and delete in $O(h)$ time
- RBT: insert, search, and delete in $O(\lg n)$ time
- Hashing: insert, search, and delete in $O(1)$ time
  - Other dictionary operations (min, succ, ...) are not supported
  - Hash table is expensive and difficult to grow and shrink.
  - Application: symbol table for identifiers in a compiler, customer IDs, ...

Hash tables vs arrays

- Hash tables are a generalization of arrays
- Big advantage of arrays consists in direct addressing
  - Ability to access arbitrary array position in $O(1)$ time

DICTIONARIES / 5

- In Java: java.util Map interface defines dictionary ADT
- Different data structures to realize dictionaries:
  - arrays
  - linked lists
  - Hash tables
  - Binary search trees
  - Red/Black trees
  - B-trees

DICTIONARIES / 6

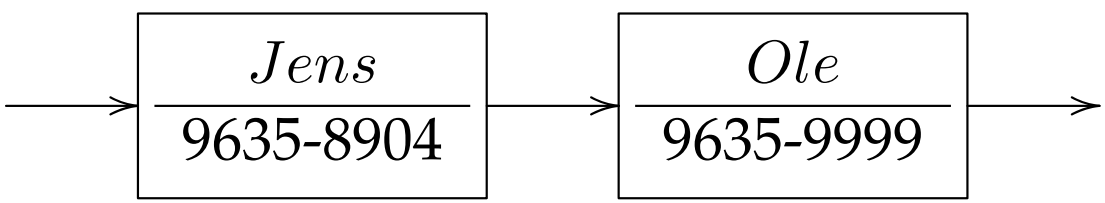
Two suboptimal ways to design this dictionary

- direct addressing: an array indexed by key

(null)	(null)	...	Jens	...	(null)
0000-0000	0000-0001	...	9635-8904	...	9999-9999

  - Requires O(1) time,
  - Requires O(r) space - huge amount of wasted space

- a linked list:



  - Requires O(n) time and O(n) space

HASH TABLES/1

- We use a **hash table** of size m to avoid wasting space.
- A hash table is like an array, but with a function to map the large range into one that we can manage efficiently.
- For instance take the original key, modulo the size of the relatively small hash table, and use that as an index.
- Example: insert *(9635-8904, Jens)* into a hash table with five slots ($m = 5$):
  - $96358904 \mod 5 = 4$

(null)	(null)	(null)	(null)	Jens
0	1	2	3	4

HASH TABLES/2

Use of hash function

- Put a record with key k in slot $h(k)$ instead of slot k

HASH TABLES/2

Use of hash function

- Put a record with key k in slot $h(k)$ instead of slot k
- Thus, a hash function h is used to compute the slot

HASH TABLES/2

Use of hash function

- Put a record with key k in slot $h(k)$ instead of slot k
- Thus, a hash function h is used to compute the slot
- Maps universe U into the slots of hash table $HT[1 \ldots m]$

$$h : U \rightarrow \{1, 2, \ldots, m\}$$

often 0 .. m-1 is used

item with key k hashes to $h(k)$, $h(k)$ is hash value of k

HASH TABLES/2

Use of hash function

- Put a record with key k in slot $h(k)$ instead of slot k
- Thus, a hash function h is used to compute the slot
- Maps universe U into the slots of hash table $HT[1 \ldots m]$

$$h\colon U \to \{1, 2, \ldots, m\}$$

item with key k hashes to $h(k)$, $h(k)$ is hash value of k

- Hash function reduces range of array indices that need to be handled, i.e., m values instead of $|U|$ values

HASH TABLES/2

Use of hash function

- Put a record with key k in slot $h(k)$ instead of slot k
- Thus, a hash function h is used to compute the slot
- Maps universe U into the slots of hash table $HT[1 \ldots m]$

$$h\colon U \to \{1, 2, \ldots, m\}$$

item with key k hashes to $h(k)$, $h(k)$ is hash value of k

- Hash function reduces range of array indices that need to be handled, i.e., m values instead of $|U|$ values
- There is a hitch: two keys may hash to the same slot

HASH TABLES/2

Use of hash function

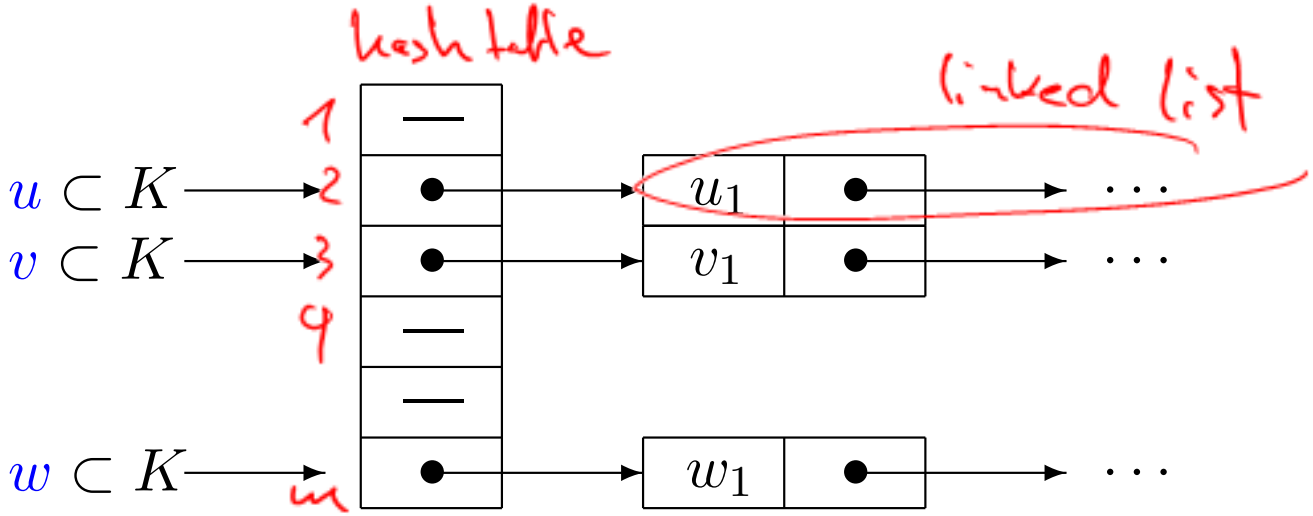
- Put a record with key k in slot $h(k)$ instead of slot k
- Thus, a hash function h is used to compute the slot
- Maps universe U into the slots of hash table $HT[1 \ldots m]$

$$h\colon U \to \{1, 2, \ldots, m\}$$

  item with key k hashes to $h(k)$, $h(k)$ is hash value of k
- Hash function reduces range of array indices that need to be handled, i.e., m values instead of $|U|$ values
- There is a hitch: two keys may hash to the same slot
- Need a technique to resolve such collision conflicts
  - Various options: chaining, probing, double hashing

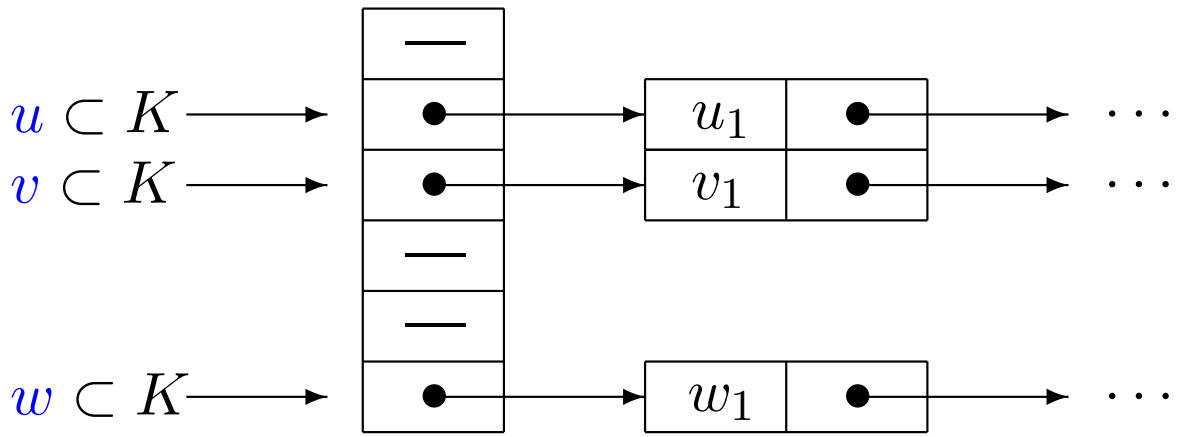
HASH TABLES/3

Collision resolution with chaining:



HASH TABLES / 3

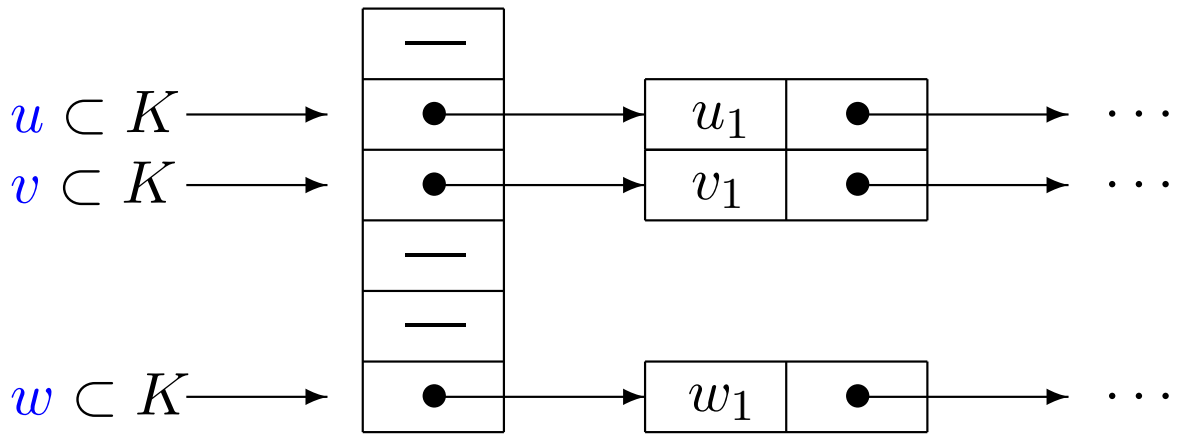
Collision resolution with chaining:



- Chaining: put items that hash to same slot in linked list

HASH TABLES / 3

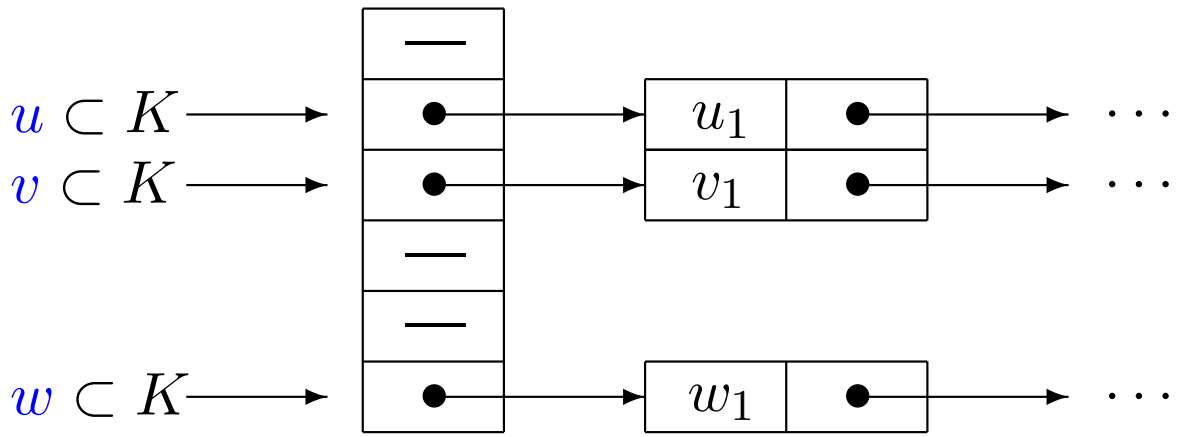
Collision resolution with chaining:



- Chaining: put items that hash to same slot in linked list
- Each slot contains either `NIL` or a pointer to head of a list

HASH TABLES / 3

Collision resolution with chaining:



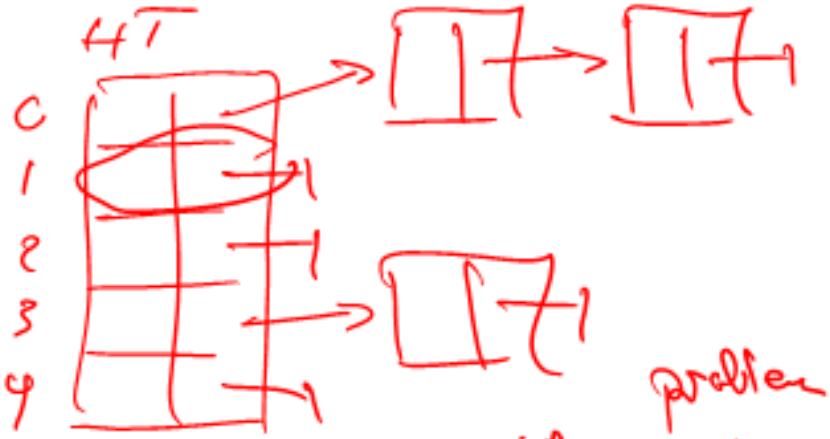
- Chaining: put items that hash to same slot in linked list
- Each slot contains either `NIL` or a pointer to head of a list
- Search, insert, and delete: relatively easy to implement

EXERCISE SL07-1

Define a hash table with chaining for collision resolution in C.

```
struct node {
    char name[20];
    struct node * next;
};

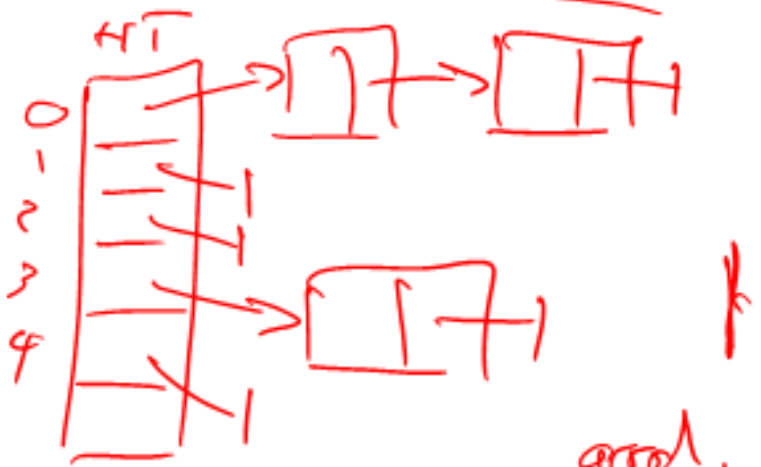
struct node HT[7000]
```



problem with empty slots

```
struct node {
    char name[20];
    struct node * next;
};

struct node * HT[7000];
```



good solution

HASH TABLES/4

Analysis of hashing

- Given a hash table HT with m slots that stores n items, define the load factor α of HT as n/m
  - α corresponds to average number of items in a list

HASH TABLES/4

Analysis of hashing

- Given a hash table HT with m slots that stores n items, define the load factor α of HT as n/m
  - α corresponds to average number of items in a list
- Assumption 1: any given item is equally likely to hash into any of the m slots of the hash table
  - This assumption is called uniform simple hashing

HASH TABLES/4

Analysis of hashing

- Given a hash table HT with m slots that stores n items, define the load factor α of HT as n/m
  - α corresponds to average number of items in a list
- Assumption 1: any given item is equally likely to hash into any of the m slots of the hash table
  - This assumption is called uniform simple hashing
- Assumption 2: $h(k)$ can be computed in constant time

HASH TABLES/4

=> leave enough slots

Analysis of hashing

- Given a hash table HT with m slots that stores n items, define the load factor α of HT as n/m
  - α corresponds to average number of items in a list
- Assumption 1: any given item is equally likely to hash into any of the m slots of the hash table
  - This assumption is called uniform simple hashing
- Assumption 2: $h(k)$ can be computed in constant time
- Search (unsuccessful/successful) takes $\Theta(1+\alpha)$ time
  - Need probability theory to prove these two results

Best case: $n=1$, $m=1M$ => $\alpha = 1/1M \approx 0$, $\Theta(1)$!

worst case: $n=1M$, $m=1$ => $\alpha = 1M/1 = n$, $\Theta(n)$!

EXERCISE SL07-2

What benefits do you expect by using a binary search tree instead of a linked list to handle hash table overflows?

- BST pays off if it gets big
  that means many collisions
- HT are only good if there are
  few collisions

=> re-design size of the HT,
hash function
h(k) = 3

TABLE OF CONTENTS — SL07

1. Hash tables

2. Hash functions
   - Goals and design
   - Division method
   - Multiplication method

3. Collision resolution

HASH FUNCTIONS/1

Design of hash functions

- Hash function h must be computed in constant time

HASH FUNCTIONS/1

Design of hash functions

- Hash function h must be computed in constant time
- Must satisfy the assumption of simple uniform hashing
  - Each key is equally likely to hash to any of the m slots

HASH FUNCTIONS/1

Design of hash functions

- Hash function h must be computed in constant time
- Must satisfy the assumption of simple uniform hashing
  - Each key is equally likely to hash to any of the m slots

Hashing non-integer keys

- Consider set $\mathbb{N}$ of natural numbers as universe of keys

HASH FUNCTIONS/1

Design of hash functions

- Hash function h must be computed in constant time
- Must satisfy the assumption of simple uniform hashing
  - Each key is equally likely to hash to any of the m slots

Hashing non-integer keys

- Consider set $\mathbb{N}$ of natural numbers as universe of keys
- If the keys are not natural number then a way is found to interpret them as such
  - Example: encoding of strings via ASCII character set

EXERCISE SL07-3

Give different options for mapping a string to an integer for the purpose of hashing. Discuss advantages and disadvantages.

ascii(red) = 82 69 68

1) add ascii codes: 82 + 69 + 68 mod 17 = ...
+ modulo ptr
⇒ red = der = esd = ...

2) ascii code of 1st character

3) concatenation of ascii codes: 826968

radix notation $(82)_{10\to 2} \cdot 128^2 + (69)_{10-2} \cdot 128^1 +$
$(68)_{10\to 2} \cdot 128^0$

HASH FUNCTIONS/2

Division method (mod)

- Hash function takes the remainder of k divided by m
  - $h(k) = (k \bmod m) + 1$
  - $m = 12, k = 100 \Rightarrow h(k) = 5$

HASH FUNCTIONS/2

Division method

- Hash function takes the remainder of k divided by m
  - $h(k) = (k \bmod m) + 1$
  - $m = 12, k = 100 \Rightarrow h(k) = 5$
- Generally, one tries to avoid certain choices for m
  - For $m = 2^p$, $h(k)$ gives the p least significant bits of k and all keys with the same ending go to the same slot

HASH FUNCTIONS/2

Division method

- Hash function takes the remainder of k divided by m
  - $h(k) = (k \bmod m) + 1$
  - $m = 12, k = 100 \Rightarrow h(k) = 5$
- Generally, one tries to avoid certain choices for m
  - For $m = 2^p$, $h(k)$ gives the p least significant bits of k and all keys with the same ending go to the same slot
- Good choice for m: prime number not close to 2^p
  - Hash table HT for roughly $n = 2000$ character strings
  - One may choose $m = 701$ because it is a prime near $\frac{2000}{3}$ (≈ 3 items per list) but not near any power of 2

EXERCISE SL07-4

this is a bad hash fct

Let $h(k) = k \mod 2^p - 1$ where k is a string interpreted in radix 2^p. Prove that strings a and b hash to the same value if a is a permutation of b.

- $a = \ldots x \ldots y \ldots$ (pos i, pos j) $\quad b = \ldots y \ldots x \ldots$
- $a_R = \ldots + x(R)^i + \ldots + y(R)^j + \ldots$
- $a_{2^p} = \ldots + x \cdot 2^{pi} + \ldots + y 2^{pj} + \ldots$
- $h(a) = a_{2^p} \mod (2^p - 1)$
- we must prove:

$$a_R \mod (2^p - 1) - b_R \mod (2^p - 1) = 0$$

EXERCISE SL07-4

$$(a_R - b_R) \bmod (2^p - 1) = 0$$

$$(x 2^{pi} + y 2^{pj} - y 2^{pi} - x 2^{pj}) \bmod (2^p - 1) = 0$$

$$(x - y)(2^{pi} - 2^{pj}) \bmod (2^p - 1) = 0$$

$$(x - y) 2^{pj} (2^{p(i-j)} - 1) \bmod (2^p - 1) = 0$$

$$\sum_{i=0}^{k} a^i = \frac{a^{k+1} - 1}{a - 1} \Rightarrow a^{k+1} - 1 = (a - 1) \sum_{i=0}^{k} a^i \qquad a = 2^p \quad k = i - j - 1$$

$$2^{p(i-j)} = (2^p - 1) \Sigma$$

$$(x - y) 2^{pj} (2^p - 1) \Sigma \ldots \bmod (2^p - 1) = 0$$

HASH FUNCTIONS/3

Multiplication method

- Multiply key k by A and extract fractional part of kA
  - $h(k) = \lfloor m(kA \bmod 1) \rfloor + 1$
  - A constant such that $0 < A < k$
  - $kA \bmod 1$: fractional part of kA, i.e. $kA - \lfloor kA \rfloor$

HASH FUNCTIONS/3

21.73 mod 1 = 0.73

Multiplication method

- Multiply key k by A and extract fractional part of kA
  - $h(k) = \lfloor m(kA \bmod 1) \rfloor + 1$
  - A constant such that $0 < A < k$
  - $kA \bmod 1$: fractional part of kA, i.e. $kA - \lfloor kA \rfloor$
- Advantage: value of m is not critical, typically use 2^p — bit shifting (fast)

HASH FUNCTIONS/3

Multiplication method

- Multiply key k by A and extract fractional part of kA
  - $h(k) = \lfloor m(kA \bmod 1) \rfloor + 1$
  - A constant such that $0 < A < k$
  - $kA \bmod 1$: fractional part of kA, i.e. $kA - \lfloor kA \rfloor$
- Advantage: value of m is not critical, typically use 2^p
- Optimal choice for A depends on data characteristics
  - Knuth suggests that $A \approx \frac{\sqrt{5}-1}{2} \approx 0.618$ works well

HASH FUNCTIONS/3

Multiplication method

- Multiply key k by A and extract fractional part of kA
  - $h(k) = \lfloor m(kA \bmod 1) \rfloor + 1$
  - A constant such that $0 < A < k$
  - $kA \bmod 1$: fractional part of kA, i.e. $kA - \lfloor kA \rfloor$
- Advantage: value of m is not critical, typically use 2^p
- Optimal choice for A depends on data characteristics
  - Knuth suggests that $A \approx \frac{\sqrt{5}-1}{2} \approx 0.618$ works well
- Example: assume that $A = 0.618$ and $m = 12$
  - $k = 102$: $h(k) = \lfloor 12 \cdot 0.036 \rfloor + 1 = \lfloor 0.432 \rfloor + 1 = 1$
  - $k = 103$: $h(k) = \lfloor 12 \cdot 0.654 \rfloor + 1 = \lfloor 7.848 \rfloor + 1 = 8$
  - $k = 104$: $h(k) = \lfloor 12 \cdot 0.272 \rfloor + 1 = \lfloor 3.264 \rfloor + 1 = 4$

TABLE OF CONTENTS — SL07

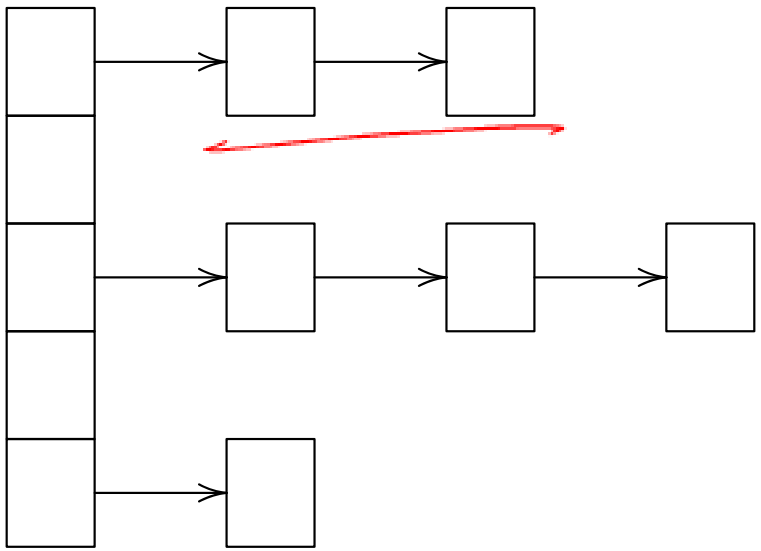
1. Hash tables

2. Hash functions

3. Collision resolution
   Chaining
   Open addressing
   - Linear probing
   - Double hashing

Chaining

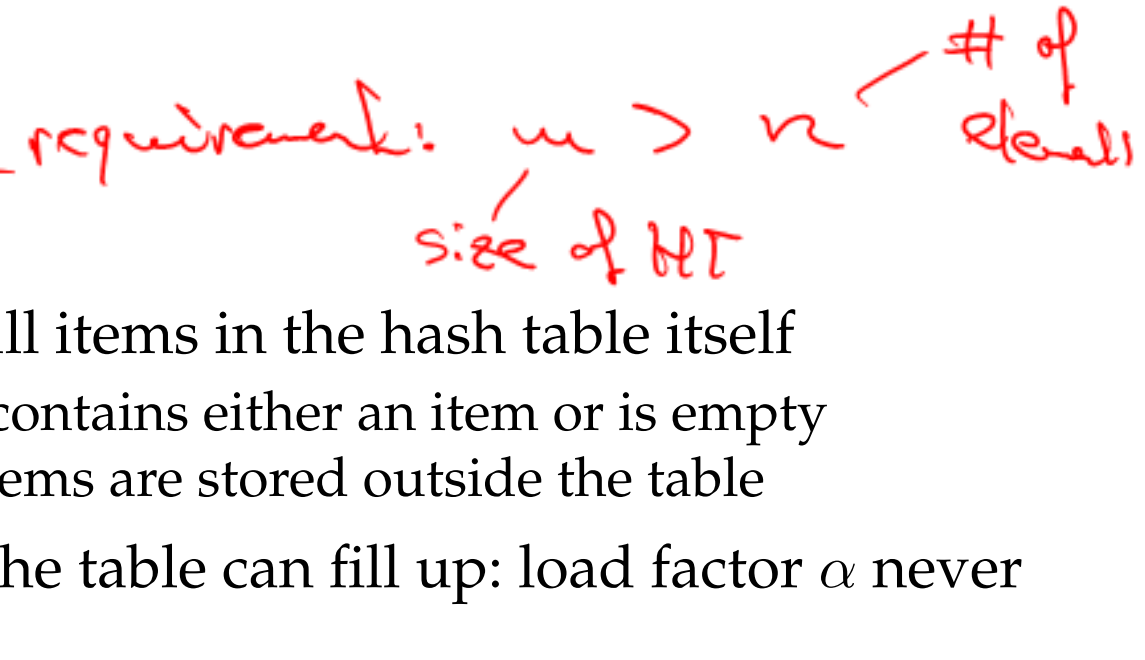
- The simplest approach is chaining.
- Chaining maintains a table of links, indexed by the keys, to lists of items with the same key.
- Pros: easy, flexible, hash table never gets full
- Cons: stores elements outside hash table, additional space for pointers



Open addressing

- The idea is to store all items in the hash table itself
  - Each table entry contains either an item or is empty
  - No lists and no items are stored outside the table

Open addressing



- The idea is to store all items in the hash table itself
  - Each table entry contains either an item or is empty
  - No lists and no items are stored outside the table
- A drawback is that the table can fill up: load factor α never exceeds 1

Open addressing

- The idea is to store all items in the hash table itself
  - Each table entry contains either an item or is empty
  - No lists and no items are stored outside the table
- A drawback is that the table can fill up: load factor α never exceeds 1
- Advantage of open addressing: avoid pointers
  - No need to follow pointers when accessing items
  - Simply compute sequence of slots to be examined
  - Larger number of slots for same amount of memory

Open addressing

- Open addressing uses an extended hash function that includes the probe number:

$$h : U \times \{1, 2, \ldots, m\} \to \{1, 2, \ldots, m\}$$

$\forall k$: $\langle h(k, 1), \ldots, h(k, m) \rangle$ is permutation of $\langle 1, \ldots, m \rangle$

probe 1 probe m

we try all slots

Open addressing

- Open addressing uses an extended hash function that includes the probe number:

$$h\colon U \times \{1, 2, \ldots, m\} \to \{1, 2, \ldots, m\}$$

  $\forall k$: $\langle h(k, 1), \ldots, h(k, m)\rangle$ is permutation of $\langle 1, \ldots, m\rangle$
- All dictionary operations systematically probe table slots
- Sequence of positions probed depends on given key

Linear Probing/1

Linear probing

- Uses a hash function of the form
  - $h(k, i) = (h'(k) + ci) \bmod m$
  - where h' is a hash function

linear fct
step size
probe nr

Linear Probing/1

Linear probing

- Uses a hash function of the form
  - $h(k, i) = (h'(k) + ci) \bmod m$
  - where h' is a hash function
- We try out (probe) a number of positions:
  NextLoc = (PreviousLoc + StepSize) mod m
- The step size should be chosen such that all locations are probed.
- Often a step size of one (c=1) is chosen. This guarantees that all locations are probed.

does not distribute data very well
⇒ many collisions ⇒ bad performance

Linear Probing/2

- If the current location is used, try the next table location:

```
Algo: LinearProbingInsert(k)
-----------------------------------------
if table is full then return error;
probe = h(k);
while HT[probe] is used do
  probe = (probe+1) mod m
table[probe] = k;
```

- Lookups walk along the table until the key or an empty slot is found
- Uses less memory than chaining: one does not have to store the links
- Slower than chaining: one might have to probe the table for a long time

EXERCISE SL07-5

Insert the keys 18, 41, 22, 44, 59, 32, 31, 73 into a hash table with 13 slots where linear probing with a step size of 1 is used as a collision resolution strategy. Show the final hash table.

$h(k,i) = (k+i) \bmod 13$

$h(18) = 5$
$h(41) = 2$
$h(22) = 9$
$h(44) = 5; 6$
$h(59) = 2$
$h(32) = 6; 7; 8$
$h(31) = 5; 6; 7; 8; 9; 10$
$h(73) = 8; 9; 10; 11$

HT = | | | 41 | | 18 | 44 | 59 | 32 | 22 | 31 | 73 |

EXERCISE SL07-6

Consider the following open addressing scheme:

```
1 i = h(k); j = 0;
2 if HT[i]==empty ∨ HT[i]==k then stop search;
3 j = (j+1) mod m; i = (i+j) mod m; goto 2;
```

probe sequence can be defined / implemented in many ways

try other next pos

Show that this schema is a quadratic probing scheme of the form $h(k,i) = (h(k) + c_1 i + c_2 i^2) \mod m$.

probe sequence: $x, x+1, x+1+2, x+1+2+3, \ldots$

$= \left(x + \sum_{j=0}^{i} j\right) \bmod m$ $\qquad \frac{i(i+1)}{2} = \frac{i^2}{2} + \frac{i}{2}$

$\Rightarrow h(k,i) = \left(h(k) + 0.5\,i + 0.5\,i^2\right) \bmod m$

DOUBLE HASHING/1

- Two hash functions:
  1. h1 determines the initial location
  2. h2 determines the step size
  3. $h(k,i) = (h_1(k) + i * h_2(k) \bmod m$
- Initial probe is to position $HT[h_1(k)]$, successive probe positions are offset by $h_2(k)$.

should not be 0

DOUBLE HASHING/2

Search:

Algo: DoubleHashingInsert(HT,k)

```
i = 0;
repeat
    probe = (h1(k) + i*h2(k)) mod m;
    i = i+1;
until (HT[probe] is free ∨ i>m);
if i>m then Error: hash table overflow;
else HT[probe] = k;
```

DOUBLE HASHING/3

no common factors in factorization

$15 = 3 \cdot 5$; $8 = 2 \cdot 2 \cdot 2$ $\Rightarrow$ 15 and 8 are relatively prime to each other

- $h_2(k)$ must be relatively prime to hash table size m
  - m a power of 2, h_2 always produces an odd number
  - m prime, h_2 always produces a positive integer $< m$

Double hashing/3

- $h_2(k)$ must be relatively prime to hash table size m
  - m a power of 2, h_2 always produces an odd number
  - m prime, h_2 always produces a positive integer $< m$
- Example: m prime and $h_2(k)$ positive integer $< m$
  - Let $h_1(k) = (k \bmod m) + 1$, $h_2(k) = (k \bmod m') + 1$
  - Choose m' to be slightly less than m (say $m' = m - 1$)
  - Given $k = 123456$, $m = 701$, $m' = 700$: $h_1(k) = 81$, $h_2(k) = 257$ (first probe to 81, then every 257th slot)

Double hashing/3

- $h_2(k)$ must be relatively prime to hash table size m
  - m a power of 2, h_2 always produces an odd number
  - m prime, h_2 always produces a positive integer $< m$
- Example: m prime and $h_2(k)$ positive integer $< m$
  - Let $h_1(k) = (k \bmod m) + 1$, $h_2(k) = (k \bmod m') + 1$
  - Choose m' to be slightly less than m (say $m' = m - 1$)
  - Given $k = 123456$, $m = 701$, $m' = 700$: $h_1(k) = 81$, $h_2(k) = 257$ (first probe to 81, then every 257th slot)
- Cost of searching in open-address hash table with $\alpha = n/m < 1$
  - unsuccessful: $1/(1-\alpha)$
  - successful: $\ln(1/(1-\alpha))/\alpha$

always goes to end of chain

stops early

$\alpha = 50\% \Rightarrow 2$
$\alpha = 90\% \Rightarrow 10$

$\alpha = 50\% \Rightarrow 1.4$
$\alpha = 90\% \Rightarrow 2.6$

EXERCISE SL07-7

Insert the keys 18, 41, 22, 44, 59, 32, 31, 73 into a hash table with double hashing where

- $h1(k) = k \mod 13$
- $h2(k) = 8 - (k \mod 8)$
- $H(k,i) = (h1(k) + i * h2(k)) \mod 13$

1 .. 8 (never 0)

h(18) = 5

h(41) = 2

h(22) = 9

h(44) = 5; 9; 0 (offset = 4)

h(59) = 7

h(32) = 6

h(31) = 5; 6; 7; 8 (offset = 1)

h(73) = 8; 2; 9; 3 (offset 7)

DOUBLE HASHING/4

Dictionary operations

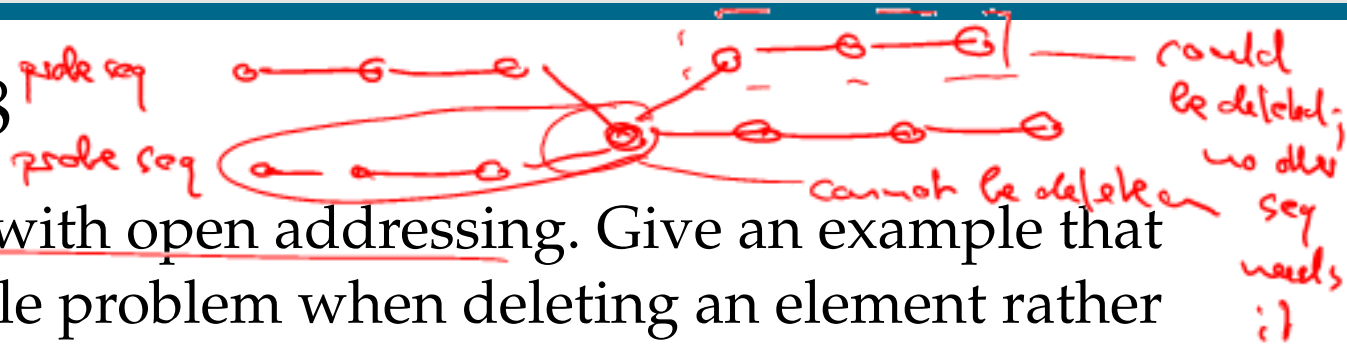
- Inserting and searching are relatively straightforward
  - At least if one assumes that there are no deletions

Double hashing/4

Dictionary operations

- Inserting and searching are relatively straightforward
  - At least if one assumes that there are no deletions
- Deletion is difficult: cannot mark deleted slot as empty
  - Solution: use special value to mark a deleted slot
  - This affects all hash table functions (e.g., the simple procedures for insertion must be extended)

EXERCISE SL07-8



Consider hashing with open addressing. Give an example that illustrates a possible problem when deleting an element rather than marking it as deleted. Can it make sense to attempt to delete elements instead of marking them as deleted? When is it possible to remove DEL?

$h(k) = k \bmod 5$ (linear probing; step size = 1)

$h(18) = 5$

$h(44) = 5; 6$

it is important to have a low load factor
⇒ deletion is important

18 is deleted

if HT[5] is set to EMPTY then search for 44 will go to slot 5 and does not 44.

Hashing with Open Addressing/1

Assume hashing with open addressing as a collision resolution strategy.

1. Define the hash table and give a procedure to initialize it.
2. Give a procedure to insert, search and delete an element

- in general domain uses all values (i.e., there are no special values for deleted and empty slots)
  ⇒ you need an extra status field
- always check status field before accessing values
- make it work for bad hash fcts
  (not all slots are visited)

HASHING WITH OPEN ADDRESSING/2

definition of hash table:

```
struct elem {
   int key;
   int status;
      /* 0: OCCupied, -1: EMPty, -2: DELeted */
};

struct elem HT [7000];
```

Algo: HTinit(HT)

---

for *i* = *0* **to** *m-1* **do** HT[i].status = EMP;

HASHING WITH OPEN ADDRESSING/3

Algo: HTinsert(HT,k)

```
i = -1;
repeat
  | i++; probe = h(k,i);
until i≥m ∨ HT[probe].status≠OCC;
if i≥m then return -1;
HT[probe].status = 0;
HT[probe].key = k;
return probe
```

probe number: 0 .. m-1

DEL or EMP slot

OCC

no free slot found (HT is full; bad hash fct)

where it was inserted

HASHING WITH OPEN ADDRESSING/4

Algo: HTsearch(HT,k)

```
i = -1;
repeat
  | i++; probe = h(k,i);
until i≥m ∨
      HT[probe].status==OCC ∧ HT[probe].key==k ∨
      HT[probe].status==EMP;
if i≥m ∨ HT[probe].status==EMP then return -1;
return probe
```

HASHING WITH OPEN ADDRESSING/5

Algo: HTdelete(HT,k)

```
i = -1;
repeat
  | i++; probe = h(k,i);
until i≥m ∨
      HT[probe].status==EMP ∨
      HT[probe].status==OCC ∧ HT[probe].key==k;
if i≥m ∨ HT[probe].status==EMP then return -1;
HT[probe].status = DEL;
return probe
```

searched all values without finding k

end of chain

found k

Comparison of hashing techniques

Expected number of probes for uniform hashing

- Chaining ($\alpha : 1 + \alpha$)
  - $0 : 1$, $0.5 : 1.5$, $1 : 2$, $n : n + 1$

Comparison of hashing techniques

Expected number of probes for uniform hashing

- Chaining ($\alpha : 1 + \alpha$)
  - $0 : 1$, $0.5 : 1.5$, $1 : 2$, $n : n + 1$
- Probing, unsuccessful ($\alpha : 1/(1 - \alpha)$)
  - $0.2 : 1.25$, $0.5 : 2$, $0.8 : 5$, $0.9 : 10$

Comparison of hashing techniques

Expected number of probes for uniform hashing

- Chaining ($\alpha : 1 + \alpha$)
  - $0 : 1$, $0.5 : 1.5$, $1 : 2$, $n : n + 1$
- Probing, unsuccessful ($\alpha : 1/(1 - \alpha)$)
  - $0.2 : 1.25$, $0.5 : 2$, $0.8 : 5$, $0.9 : 10$
- Probing, successful ($\alpha : (\ln(1/(1 - \alpha))/\alpha$)
  - $0.2 : 0.28$, $0.5 : 1.39$, $0.8 : 2.01$, $0.9 : 2.56$

Comparison of hashing techniques

Expected number of probes for uniform hashing

- Chaining ($\alpha : 1 + \alpha$)
  - $0 : 1$, $0.5 : 1.5$, $1 : 2$, $n : n + 1$
- Probing, unsuccessful ($\alpha : 1/(1 - \alpha)$)
  - $0.2 : 1.25$, $0.5 : 2$, $0.8 : 5$, $0.9 : 10$
- Probing, successful ($\alpha : (\ln(1/(1 - \alpha))/\alpha)$
  - $0.2 : 0.28$, $0.5 : 1.39$, $0.8 : 2.01$, $0.9 : 2.56$

Final remarks

- Hashing is very efficient (not obvious, probability)

Comparison of hashing techniques

Expected number of probes for uniform hashing

- Chaining ($\alpha : 1 + \alpha$)
  - $0 : 1$, $0.5 : 1.5$, $1 : 2$, $n : n + 1$
- Probing, unsuccessful ($\alpha : 1/(1 - \alpha)$)
  - $0.2 : 1.25$, $0.5 : 2$, $0.8 : 5$, $0.9 : 10$
- Probing, successful ($\alpha : (\ln(1/(1 - \alpha))/\alpha$)
  - $0.2 : 0.28$, $0.5 : 1.39$, $0.8 : 2.01$, $0.9 : 2.56$

Final remarks

- Hashing is very efficient (not obvious, probability)
- May not be easy to determine parameters (e.g. m).
- Worst case exists and cannot be avoided in general.

EXERCISE SL07-9

proof by contradiction

Consider quadratic probing of the form

$$h(k,i) = (h(k) + i/2 + i^2/2) \mod 2^p$$

Prove that with this scheme every table position is examined in the worst case.

Example: $2^3 = 8$ slots; $k = 52$

4; 5; 7; 2; 6; 3; 1; 0

We must prove: $\forall i, j, k$ $h(k,i) \neq h(k,j)$, $0 \leq i < j < 2^p$

$(h(k) + i/2 + i^2/2) \mod 2^p = (h(k,j) + j/2 + j^2/2) \mod 2^p$

$(j/2 + j^2/2 - i/2 - i^2/2) \mod 2^p = 0$

$(j-i)(j+i+1) = x\,2^{p+1}$

$\frac{(j-i)(j+i+1)}{2^{p+1}} = x \in \mathbb{N}$

EXERCISE SL07-9

look at parities:

i	j	j−i	j+i+1
odd	odd	even	odd
odd	even	odd	even
even	odd	odd	even
even	even	even	odd

⇒ different parities

from factorizing $2^{p+1} = 2 \cdot 2 \cdot 2 \cdots \cdot 2$ it follows that 2^{p+1} must divide one of the terms

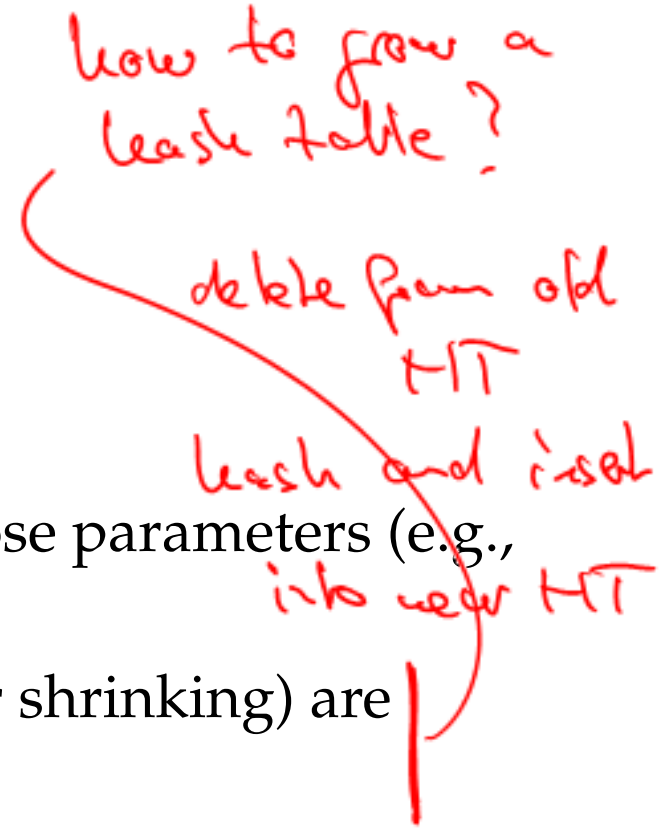
- 2^{p+1} cannot divide $j-i$ since $j-i < 2^p$

- 2^{p+1} cannot divide $(j+i+1)$ since

$j+i+1 \leq \underbrace{2^p-1}_{\text{largest possible } j} + \underbrace{2^p-2}_{\text{largest possible } i} + 1 \Rightarrow j+i+1 \leq 2^{p+1}-2$

we have a contradic

SUMMARY



- Hashing is simple to implement.
- Hash table give constant access time.
- Hashing requires some difficult to choose parameters (e.g., size of hash table).
- Hash table reorganization (enlarging or shrinking) are expensive (not constant anymore).
- Analysis of hashing requires probability theory.
- In general hashing must deal with collisions.

Informatics II

Dynamic Programming

— SL08 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

TABLE OF CONTENTS — SL08

1. Design techniques
   Design techniques
   Fibonacci numbers

2. Matrix multiplication

3. Dynamic programming

4. Longest common subsequence, coin changing

DESIGN TECHNIQUES/1

Design of algorithms

- Design techniques used so far
  - Iterative (brute-force) algorithms
    - Linear search, selection sort, insertion sort, etc
  - Data structure based algorithms
    - Heap sort, PQ sort, BST sort, RBT sort, etc
  - Divide-and-conquer algorithms
    - Binary search, merge sort, quick sort, etc

Design techniques/1

Design of algorithms

- Design techniques used so far
  - Iterative (brute-force) algorithms
    - Linear search, selection sort, insertion sort, etc
  - Data structure based algorithms
    - Heap sort, PQ sort, BST sort, RBT sort, etc
  - Divide-and-conquer algorithms
    - Binary search, merge sort, quick sort, etc
- But there exist other techniques
  - Dynamic programming (remember solutions)
    - Matrix multiplication, longest common subsequence
  - Greedy algorithms (choose local optimum in each step)
    - Graph based: minimum spanning tree, shortest path

Design techniques/2

Divide-and-conquer

- Principle: if problem size is small enough to be solved trivially then solve it; else
  1. Divide: problem into a number of subproblems
  2. Conquer: subproblems by solving them recursively
  3. Combine: solutions of the subproblem into solution for the original problem

DESIGN TECHNIQUES/2

Divide-and-conquer

- Principle: if problem size is small enough to be solved trivially then solve it; else
  1. Divide: problem into a number of subproblems
  2. Conquer: subproblems by solving them recursively
  3. Combine: solutions of the subproblem into solution for the original problem
- Subproblems are independent and non-overlapping

Ex: Quicksort: left part / right part
inorder tree traversal: left subtree / right subtree

Design techniques/2

Divide-and-conquer

- Principle: if problem size is small enough to be solved trivially then solve it; else
  1. Divide: problem into a number of subproblems
  2. Conquer: subproblems by solving them recursively
  3. Combine: solutions of the subproblem into solution for the original problem
- Subproblems are independent and non-overlapping

Dynamic programming

- Solve problem by combining solutions of subproblems

Design techniques/2

Divide-and-conquer

- Principle: if problem size is small enough to be solved trivially then solve it; else
  1. Divide: problem into a number of subproblems
  2. Conquer: subproblems by solving them recursively
  3. Combine: solutions of the subproblem into solution for the original problem
- Subproblems are independent and non-overlapping

Dynamic programming

- Solve problem by combining solutions of subproblems
- Subproblems are not independent and overlapping

FIBONACCI NUMBERS/1

Recursive definition

$$\text{Fibonacci} \quad fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-2) & \\ + fib(n-1) & \text{if } n > 1 \end{cases}$$

FIBONACCI NUMBERS/1

Recursive definition

$$\text{Fibonacci} \quad fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-2) & \\ + fib(n-1) & \text{if } n > 1 \end{cases}$$

Recursive algorithm

Algo: Fib1(n)

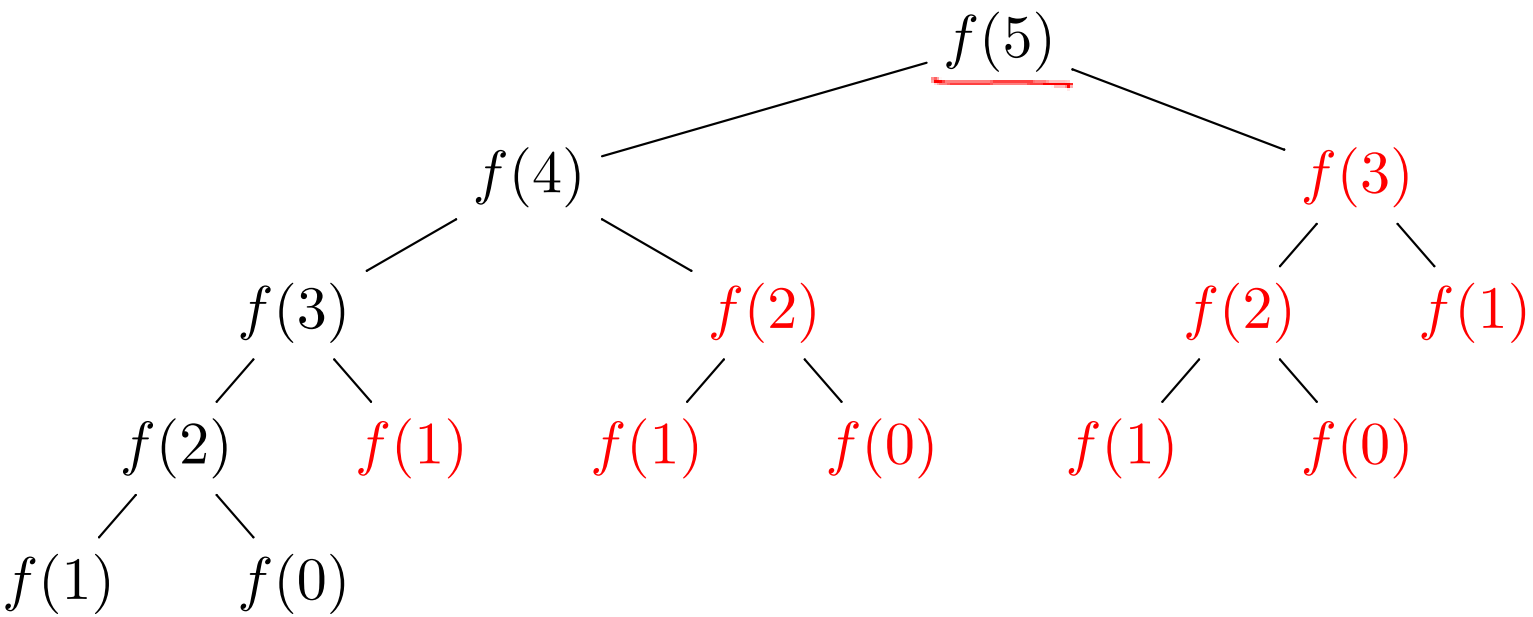
---

if $n<2$ **then return** n;
return Fib1(n-2) + Fib1(n-1);

FIBONACCI NUMBERS/2

Analysis of algorithm

- Keep calculating the same values over and over again



EXERCISE SL08-1

Determine the asymptotic runtime complexity of the recursive Fibonacci numbers algorithm.

FIBONACCI NUMBERS/3

Bottom-up approach

- Keep **remembering solutions** of solved subproblems

Algo: Fib2(n)

```
f[1] = 0;
f[2] = 1;
for i = 3 to n+1 do
  f[i] = f[i-2] + f[i-1]
return f(n+1);
```

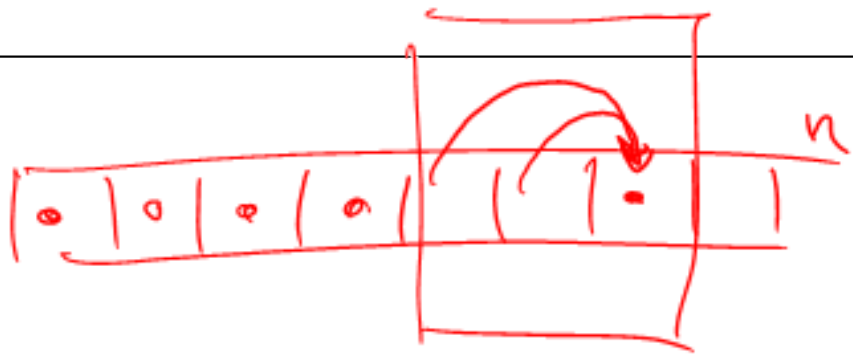
FIBONACCI NUMBERS/3

Bottom-up approach

- Keep **remembering solutions** of solved subproblems

Algo: Fib2(n)

```
f[1] = 0;
f[2] = 1;
for i = 3 to n+1 do
  f[i] = f[i-2] + f[i-1]
return f(n+1);
```



- Can compute value of $fib(n)$ in linear time (i.e., $\Theta(n)$)
- Algorithm computes the solution in bottom-up fashion.
- The bottom-up approach trades space for time.

FIBONACCI NUMBERS / 4

Optimized bottom-up algorithm

- Can do without remembering all the previous results

Algo: Fib3(n)

```
s = 0;
t = 1;
for i = 1 to n do
    r = t;
    t = t+s;
    s = r
return t;
```



FIBONACCI NUMBERS / 4

Optimized bottom-up algorithm

- Can do without remembering all the previous results

Algo: Fib3(n)

```
s = 0;
t = 1;
for i = 1 to n do
    r = t;
    t = t+s;
    s = r
return t;
```

- Thus, space complexity is constant rather than linear

Observations

- An inefficient algorithm does not imply that no efficient solution for the problem at hand exists.

OBSERVATIONS

- An inefficient algorithm does not imply that no efficient solution for the problem at hand exists.
- If the same problem is computed over and over again we can improve the performance by remembering previously computed solutions.

OBSERVATIONS

- An inefficient algorithm does not imply that no efficient solution for the problem at hand exists.
- If the same problem is computed over and over again we can improve the performance by remembering previously computed solutions.
- Such an approach trades space for time.

Observations

- An inefficient algorithm does not imply that no efficient solution for the problem at hand exists.
- If the same problem is computed over and over again we can improve the performance by remembering previously computed solutions.
- Such an approach trades space for time.
- The sequence of computation must be organized in such a way that later computed solutions can use earlier computed solutions.

TABLE OF CONTENTS — SL08

1. Design techniques

2. Matrix multiplication
   Basics of dynamic programming
   Matrix-chain multiplication

3. Dynamic programming

4. Longest common subsequence, coin changing

Overview of dynamic programming

- Invented in the 1950s by Richard Bellman as a general method for optimizing multistage decision procedures

Overview of dynamic programming

- Invented in the 1950s by Richard Bellman as a general method for optimizing multistage decision procedures
- The term *dynamic programming* (DP) refers to a tabular method (remember solutions in a table)

Overview of dynamic programming

- Invented in the 1950s by Richard Bellman as a general method for optimizing multistage decision procedures
- The term *dynamic programming* (DP) refers to a tabular method
- DP is typically applied to optimization problems
  - Have to choose one solution out of many
  - Want the solution with the optimal value
  - Characterize the structure of a solution
    - Consists of a sequence of choices that can be made
    - What choices have to be made for an optimal solution?
  - Algorithm computes an optimal value
    (plus, if needed, an optimal solution)

Matrix-chain multiplication/1

$C = A \times B$

Problem definition

- Matrices $A(n \times m)$ and $B(m \times k)$ can be multiplied to get matrix $C(n \times k)$ using nmk scalar multiplications

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

$c_{ij} = \sum_{l=1}^{m} a_{il} \cdot b_{lj}$ (e.g. $c_{22} = a_{21} \cdot b_{12} + a_{22} \cdot b_{22}$)

m multiplications for c_{ij}

we have $n * k$ c_{ij}

MATRIX-CHAIN MULTIPLICATION/1

Problem definition

- Matrices $A(n \times m)$ and $B(m \times k)$ can be multiplied to get matrix $C(n \times k)$ using nmk scalar multiplications

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

$$c_{ij} = \sum_{l=1}^{m} a_{il} \cdot b_{lj} \quad \text{(e.g. } c_{22} = a_{21} \cdot b_{12} + a_{22} \cdot b_{22}\text{)}$$

- Matrix multiplication is associative, i.e., $A(BC) = (AB)C$

we solve this problem

MATRIX-CHAIN MULTIPLICATION/1

Problem definition

- Matrices $A(n \times m)$ and $B(m \times k)$ can be multiplied to get matrix $C(n \times k)$ using nmk scalar multiplications

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

$$c_{ij} = \sum_{l=1}^{m} a_{il} \cdot b_{lj} \quad \text{(e.g. } c_{22} = a_{21} \cdot b_{12} + a_{22} \cdot b_{22}\text{)}$$

- Matrix multiplication is associative, i.e., $A(BC) = (AB)C$
- Problem: compute product of many matrices efficiently (i.e., determine the best sequence of multiplications)

MATRIX-CHAIN MULTIPLICATION/2

Problem definition

- It matters how the product $A_1 \cdots A_n$ is parenthesized

MATRIX-CHAIN MULTIPLICATION/2

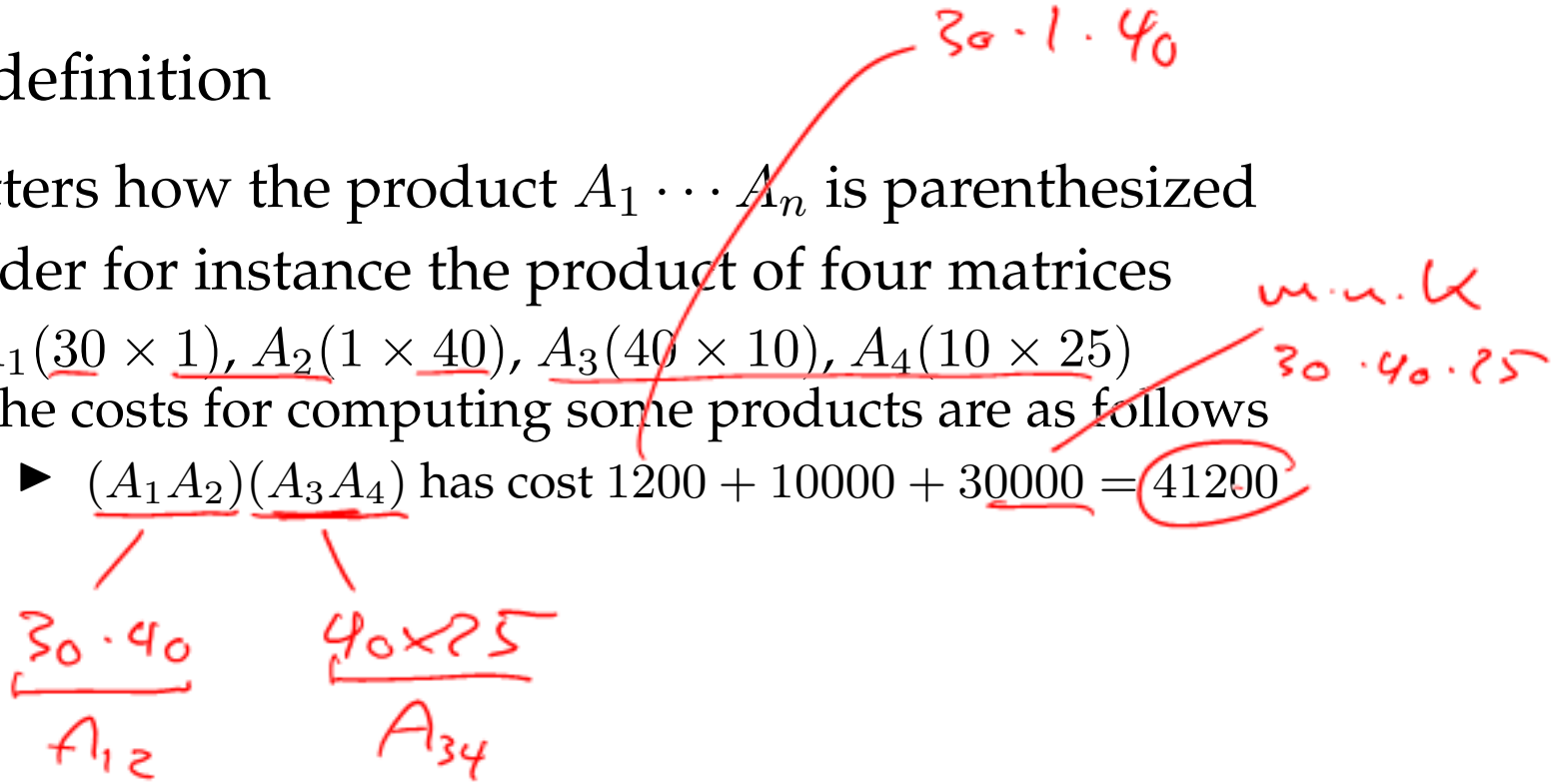
Problem definition

- It matters how the product $A_1 \cdots A_n$ is parenthesized
- Consider for instance the product of four matrices
  - $A_1(30 \times 1)$, $A_2(1 \times 40)$, $A_3(40 \times 10)$, $A_4(10 \times 25)$
  - The costs for computing some products are as follows

MATRIX-CHAIN MULTIPLICATION/2

Problem definition

- It matters how the product $A_1 \cdots A_n$ is parenthesized
- Consider for instance the product of four matrices
  - $A_1(30 \times 1), A_2(1 \times 40), A_3(40 \times 10), A_4(10 \times 25)$
  - The costs for computing some products are as follows
    - $(A_1 A_2)(A_3 A_4)$ has cost $1200 + 10000 + 30000 = 41200$



MATRIX-CHAIN MULTIPLICATION/2

Problem definition

- It matters how the product $A_1 \cdots A_n$ is parenthesized
- Consider for instance the product of four matrices
  - $A_1(30 \times 1)$, $A_2(1 \times 40)$, $A_3(40 \times 10)$, $A_4(10 \times 25)$
  - The costs for computing some products are as follows
    - $(A_1 A_2)(A_3 A_4)$ has cost $1200 + 10000 + 30000 = 41200$

      - $A_{12} = A_1 A_2$ is a (30×40) matrix

MATRIX-CHAIN MULTIPLICATION/2

Problem definition

- It matters how the product $A_1 \cdots A_n$ is parenthesized
- Consider for instance the product of four matrices
  - $A_1(30 \times 1)$, $A_2(1 \times 40)$, $A_3(40 \times 10)$, $A_4(10 \times 25)$
  - The costs for computing some products are as follows
    - $(A_1 A_2)(A_3 A_4)$ has cost $1200 + 10000 + 30000 = 41200$

      - $A_{12} = A_1 A_2$ is a (30×40) matrix
      - computation cost of A_{12} is $30 * 1 * 40 = 1200$

MATRIX-CHAIN MULTIPLICATION/2

Problem definition

- It matters how the product $A_1 \cdots A_n$ is parenthesized
- Consider for instance the product of four matrices
  - $A_1(30 \times 1)$, $A_2(1 \times 40)$, $A_3(40 \times 10)$, $A_4(10 \times 25)$
  - The costs for computing some products are as follows
    - $(A_1 A_2)(A_3 A_4)$ has cost $1200 + 10000 + 30000 = 41200$

      - $A_{12} = A_1 A_2$ is a (30×40) matrix
      - computation cost of A_{12} is $30 * 1 * 40 = 1200$
      - $A_{1234} = A_{12} A_{34}$ is a (30×25) matrix

MATRIX-CHAIN MULTIPLICATION/2

Problem definition

- It matters how the product $A_1 \cdots A_n$ is parenthesized
- Consider for instance the product of four matrices
  - $A_1(30 \times 1)$, $A_2(1 \times 40)$, $A_3(40 \times 10)$, $A_4(10 \times 25)$
  - The costs for computing some products are as follows
    - $(A_1 A_2)(A_3 A_4)$ has cost $1200 + 10000 + 30000 = 41200$

      - $A_{12} = A_1 A_2$ is a (30×40) matrix
      - computation cost of A_{12} is $30 * 1 * 40 = 1200$

      - $A_{1234} = A_{12} A_{34}$ is a (30×25) matrix
      - computation cost of A_{1234} is $30 * 40 * 25 = 30'000$

MATRIX-CHAIN MULTIPLICATION/2

Problem definition

- It matters how the product $A_1 \cdots A_n$ is parenthesized
- Consider for instance the product of four matrices
  - $A_1(30 \times 1)$, $A_2(1 \times 40)$, $A_3(40 \times 10)$, $A_4(10 \times 25)$
  - The costs for computing some products are as follows
    - $(A_1A_2)(A_3A_4)$ has cost $1200 + 10000 + 30000 = 41200$

      - $A_{12} = A_1A_2$ is a (30×40) matrix
      - computation cost of A_{12} is $30 * 1 * 40 = 1200$

      - $A_{1234} = A_{12}A_{34}$ is a (30×25) matrix
      - computation cost of A_{1234} is $30 * 40 * 25 = 30'000$
    - $((A_1A_2)A_3)A_4$ has cost $1200 + 12000 + 7500 = 20700$

MATRIX-CHAIN MULTIPLICATION/2

Problem definition

- It matters how the product $A_1 \cdots A_n$ is parenthesized
- Consider for instance the product of four matrices
  - $A_1(30 \times 1)$, $A_2(1 \times 40)$, $A_3(40 \times 10)$, $A_4(10 \times 25)$
  - The costs for computing some products are as follows
    - $(A_1A_2)(A_3A_4)$ has cost $1200 + 10000 + 30000 = 41200$

      - $A_{12} = A_1A_2$ is a (30×40) matrix
      - computation cost of A_{12} is $30 * 1 * 40 = 1200$

      - $A_{1234} = A_{12}A_{34}$ is a (30×25) matrix
      - computation cost of A_{1234} is $30 * 40 * 25 = 30'000$
    - $((A_1A_2)A_3)A_4$ has cost $1200 + 12000 + 7500 = 20700$
    - $A_1((A_2A_3)A_4)$ has cost $400 + 250 + 750 = 1400$

MATRIX-CHAIN MULTIPLICATION/2

Problem definition

- It matters how the product $A_1 \cdots A_n$ is parenthesized
- Consider for instance the product of four matrices
  - $A_1(30 \times 1)$, $A_2(1 \times 40)$, $A_3(40 \times 10)$, $A_4(10 \times 25)$
  - The costs for computing some products are as follows
    - $(A_1A_2)(A_3A_4)$ has cost $1200 + 10000 + 30000 = 41200$

      - $A_{12} = A_1A_2$ is a (30×40) matrix
      - computation cost of A_{12} is $30 * 1 * 40 = 1200$

      - $A_{1234} = A_{12}A_{34}$ is a (30×25) matrix
      - computation cost of A_{1234} is $30 * 40 * 25 = 30'000$
    - $((A_1A_2)A_3)A_4$ has cost $1200 + 12000 + 7500 = 20700$
    - $A_1((A_2A_3)A_4)$ has cost $400 + 250 + 750 = 1400$
- In other words, it is necessary to optimally parenthesize $A_1 \cdots A_n$.

EXERCISE SL08-2

Propose an appropriate problem representation for solving the matrix multiplication problem.

- array d with dimensionalities
- we need $n+1$ dimensions for a product of n matrices

- 2 matrices $A_1 \times A_2$

$A_1 = (n \times m)$ $A_2 = (m \times k)$

$d = \boxed{n \mid m \mid k}$ ⟸

0 1 2

$d = \boxed{d_0 \mid d_1 \mid d_2 \mid d_3 \mid \cdots}$

0 1 2 3

- n matrices: $A_1 = (d_0 \times d_1)$ $A_2 = (d_1 \times d_2)$, $A_3 = (d_2 \times d_3) \ldots$

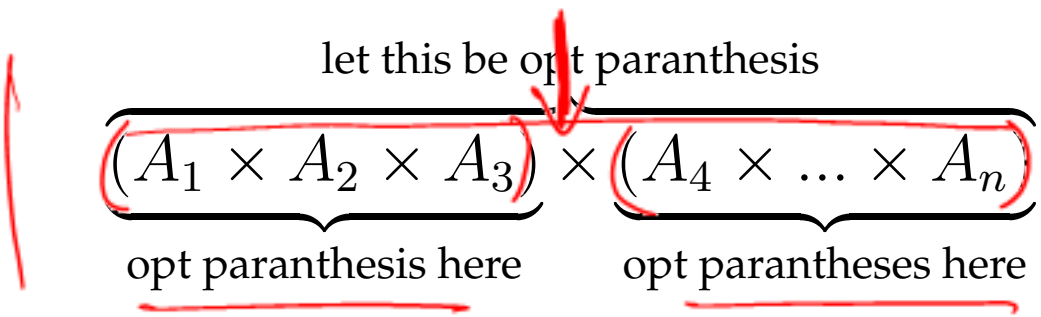
MATRIX-CHAIN MULTIPLICATION/3

Optimal substructure

precondition for DP

- Let $M(i,j)$ $(i \leq j)$ be the minimum number of scalar multiplications necessary to compute $A_{i\ldots j} = A_i \cdots A_j$
- Key observations
  - Outermost parenthesis partitions the chain of matrices (i,j) at some k $(i \leq k < j)$: $(A_i \cdots A_k)(A_{k+1} \cdots A_j)$
  - Optimal parenthesization of matrices (i,j) has optimal parenthesization on either side of k, i.e. for matrices (i,k) and $(k+1,j)$
- Example $(i = 1, j = n, k = 3)$:





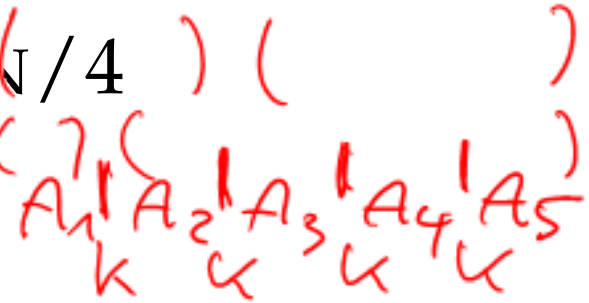
MATRIX-CHAIN MULTIPLICATION/4

Recursive problem formulation

- One can try out all possible k and take the one with the smallest cost:

$$M(i,j) = \min_{i \le k < j} \{M(i,k) + M(k+1,j) + d_{i-1} d_k d_j\}$$

MATRIX-CHAIN MULTIPLICATION/4



Recursive problem formulation

- One can try out all possible k and take the one with the smallest cost:

$$M(i,j) = \min_{i \le k < j}\{M(i,k) + M(k+1,j) + d_{i-1}d_k d_j\}$$

- The base case is $M(i,i) = 0$
- A direct recursive implementation is exponential (there is a lot of duplicated work)

MATRIX-CHAIN MULTIPLICATION/4

Recursive problem formulation

- One can try out all possible k and take the one with the smallest cost:

  $$M(i,j) = \min_{i \le k < j}\{M(i,k) + M(k+1,j) + d_{i-1}d_k d_j\}$$

- The base case is $M(i,i) = 0$
- A direct recursive implementation is exponential (there is a lot of duplicated work)
- But there are only few different subproblems (i,j) (one solution for each i and j, $i \le j$)

$$\Pi(1,4) = \min \begin{cases} \Pi(1,1) + \Pi(2,4) \equiv A_1(A_2A_3A_4) \\ \Pi(1,2) + \Pi(3,4) \equiv (A_1A_2)(A_3A_4) \\ \Pi(1,3) + \Pi(4,4) \equiv (A_1A_2A_3)A_4 \end{cases}$$

MATRIX-CHAIN MULTIPLICATION/5

Recursive solution

$$m(i,j) = \min_{i \le k < j} \{ m(i,k) + m(k+1,j) + d_{i-1} \cdot d_k \cdot d_j \}$$

- Direct implementation of the recursive problem formulation

Algo: MatrixRec(i,j)

```
if i==j then return 0;
min = MatrixRec(i,i) + MatrixRec(i+1,j) + d_{i-1} d_i d_j;
for k = i+1 to j-1 do
    q = MatrixRec(i,k) + MatrixRec(k+1,j) + d_{i-1} d_k d_j;
    if q<min then min = q;
return min;
```

k=i (initialization)

MATRIX-CHAIN MULTIPLICATION/6

M(i,j) - remember

DP solution

- Idea for dynamic programming solution: store the optimal cost $M(i, j)$ for each subproblem in a two-dimensional array $M[1 \ldots n, 1 \ldots n]$
  - Trivially, $M(i, i) = 0$ for $1 \leq i \leq n$
  - To compute $M(i, j)$ one needs only values of M for subproblems of length less than $l = j - i$
  - So, one has to solve subproblems according to their length: first subproblems of length 1, then the ones of length 2, and so on

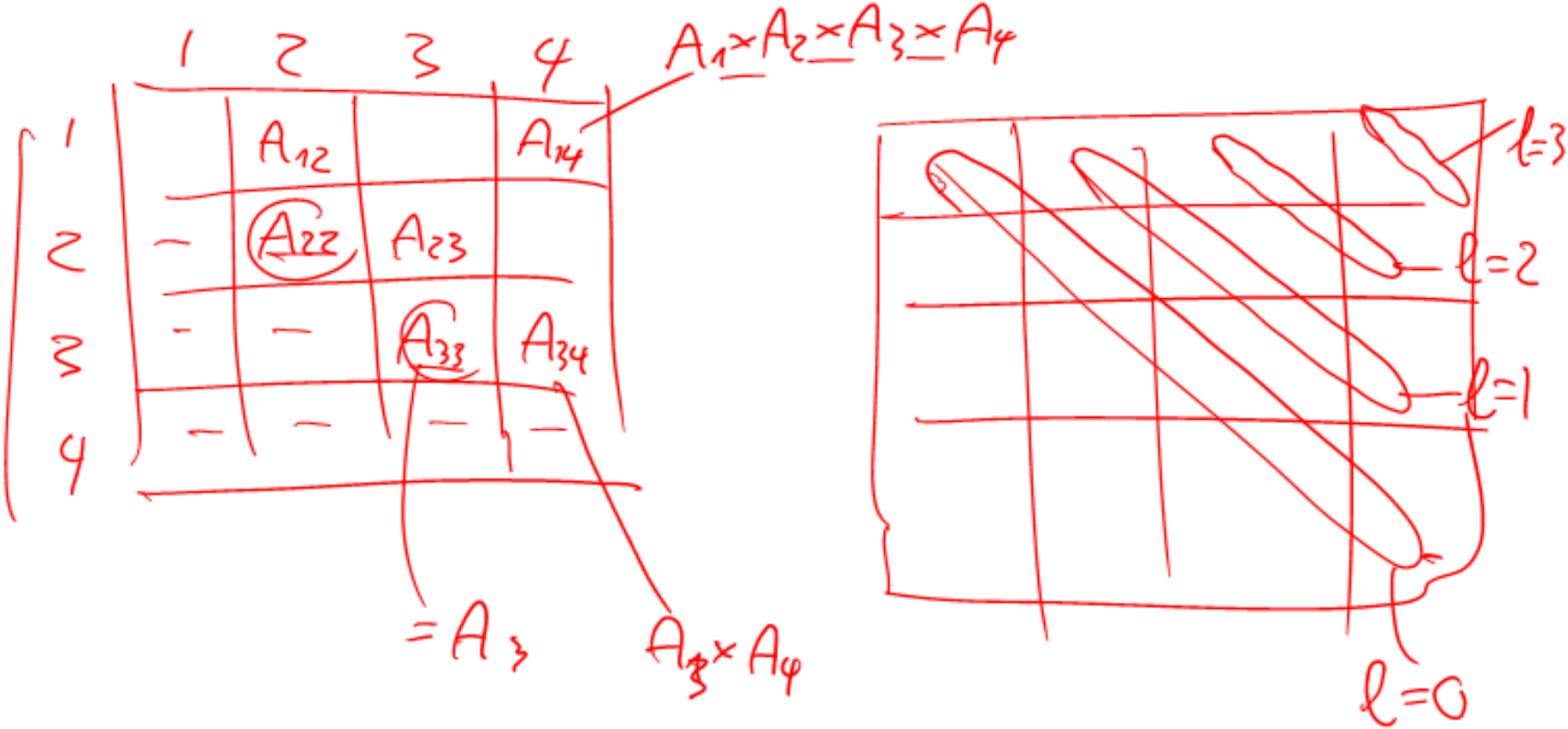
MATRIX-CHAIN MULTIPLICATION/6

DP solution

- Idea for dynamic programming solution: store the optimal cost $M(i, j)$ for each subproblem in a two-dimensional array $M[1 \ldots n, 1 \ldots n]$
  - Trivially, $M(i, i) = 0$ for $1 \leq i \leq n$
  - To compute $M(i, j)$ one needs only values of M for subproblems of length less than $l = j - i$
  - So, one has to solve subproblems according to their length: first subproblems of length 1, then the ones of length 2, and so on
- To reconstruct an optimal solution: for every (i, j), one records in $c[i, j]$ the optimal split into (i, k) and $(k + 1, j)$

EXERCISE SL08-3

Illustrate the data structure that is used to compute the optimal parentheses for a series of matrix multiplications. Illustrate the sequence in which the computation steps are performed.



MATRIX-CHAIN MULTIPLICATION/7

A simple example

- Run the algorithm on the chain $\langle A_1, A_2, A_3, A_4 \rangle$
  - $A_1(3 \times 2)$, $A_2(2 \times 4)$, $A_3(4 \times 5)$, $A_4(5 \times 1)$
  - $M[1,2] = 3 \cdot 2 \cdot 4 = 24$
  - $M[2,3] = 2 \cdot 4 \cdot 5 = 40$
  - $M[3,4] = 4 \cdot 5 \cdot 1 = 20$

Matrix-chain multiplication/7

A simple example

- Run the algorithm on the chain $\langle A_1, A_2, A_3, A_4 \rangle$
  - $A_1(3 \times 2)$, $A_2(2 \times 4)$, $A_3(4 \times 5)$, $A_4(5 \times 1)$

- $M[1,2] = 3 \cdot 2 \cdot 4 = 24$
- $M[2,3] = 2 \cdot 4 \cdot 5 = 40$
- $M[3,4] = 4 \cdot 5 \cdot 1 = 20$

- $M[1,3] = M[1,2] + M[3,3] + 3 \cdot 4 \cdot 5 = 84$
- $M[1,3] = M[1,1] + M[2,3] + 3 \cdot 2 \cdot 5 = 70$

MATRIX-CHAIN MULTIPLICATION/7

A simple example

- Run the algorithm on the chain $\langle A_1, A_2, A_3, A_4 \rangle$
  - $A_1(3 \times 2)$, $A_2(2 \times 4)$, $A_3(4 \times 5)$, $A_4(5 \times 1)$

-
  - $M[1,2] = 3 \cdot 2 \cdot 4 = 24$
  - $M[2,3] = 2 \cdot 4 \cdot 5 = 40$
  - $M[3,4] = 4 \cdot 5 \cdot 1 = 20$
-
  - $M[1,3] = M[1,2] + M[3,3] + 3 \cdot 4 \cdot 5 = 84$
  - $M[1,3] = M[1,1] + M[2,3] + 3 \cdot 2 \cdot 5 = 70$
-
  - $M[2,4] = M[2,3] + M[4,4] + 2 \cdot 5 \cdot 1 = 50$
  - $M[2,4] = M[2,2] + M[3,4] + 2 \cdot 4 \cdot 1 = 28$

$\ell = 2$

Matrix-chain multiplication/7

A simple example

- Run the algorithm on the chain $\langle A_1, A_2, A_3, A_4 \rangle$
  - $A_1(3 \times 2)$, $A_2(2 \times 4)$, $A_3(4 \times 5)$, $A_4(5 \times 1)$

- $M[1,2] = 3 \cdot 2 \cdot 4 = 24$
- $M[2,3] = 2 \cdot 4 \cdot 5 = 40$
- $M[3,4] = 4 \cdot 5 \cdot 1 = 20$
- $M[1,3] = M[1,2] + M[3,3] + 3 \cdot 4 \cdot 5 = 84$
- $M[1,3] = M[1,1] + M[2,3] + 3 \cdot 2 \cdot 5 = 70$
- $M[2,4] = M[2,3] + M[4,4] + 2 \cdot 5 \cdot 1 = 50$
- $M[2,4] = M[2,2] + M[3,4] + 2 \cdot 4 \cdot 1 = 28$
- $M[1,4] = M[1,3] + M[4,4] + 3 \cdot 5 \cdot 1 = 85$
- $M[1,4] = M[1,2] + M[3,4] + 3 \cdot 4 \cdot 1 = 56$
- $M[1,4] = M[1,1] + M[2,4] + 3 \cdot 2 \cdot 1 = 34$

EXERCISE SL08-4

Determine the optimal parentheses for computing the matrix product $A_1(3 \times 2) \times A_2(2 \times 4) \times A_3(4 \times 5) \times A_4(5 \times 1)$.

cost matrix C

	1	2	3	4
1	0			
2		0		
3			0	
4				0

	1	2	3	4
1	0	24		
2		0	40	
3			0	20
4				0

$A_1A_2A_3$

A_2A_3

	1	2	3	4
1	0	24	76	
2		0	40	28
3			0	20
4				0

	1	2	3	4
1	0	24	76	34
2		0	40	28
3			0	20
4				0

optimal cost for A_{14} is 34

EXERCISE SL08-5

$M[i, j]$ is used to determine the **cost** of the optimal parentheses. Where to actually put the parentheses is not stored. We need an additional data structure for this that is similar to M. Extract this information from the previously computed M and fill the following matrix c.

For solution (where to place parenthesis)

	1	2	3	4
1		1	1	1
2			2	2
3				3
4				

A_{14}

A_{24}

$A_1(A_2(A_3 A_4))$

MATRIX-CHAIN MULTIPLICATION/8

DP solution

Algo: MatrixDP(d)

```
for l = 1 to n do M[i,i] = 0;
for l = 1 to n-1 do
    for i = 1 to n-l do
        j = i+l;
        M[i,j] = ∞;
        for k = i to j-1 do
            q = M[i,k] + M[k+1,j] + d_{i-1} d_k d_j;
            if q<M[i,j] then M[i,j] = q; c[i,j] = k;
return (M,c);
```

MATRIX-CHAIN MULTIPLICATION/9

Analysis of the DP algorithm

- After the execution of the algorithm it holds that
  - $M[1, n]$ contains the value of the optimal solution
  - c stores the optimal subdivisions (choices of k)

MATRIX-CHAIN MULTIPLICATION/9

Analysis of the DP algorithm

- After the execution of the algorithm it holds that
  - $M[1, n]$ contains the value of the optimal solution
  - c stores the optimal subdivisions (choices of k)
- It is easy to see that the running time is $O(n^3)$
  - Note that the algorithm has three nested loops
  - Reduction from exponential to polynomial time



MATRIX-CHAIN MULTIPLICATION/10

Memoization/1

rec / 1 / exp comp, easy | mem / fast, easy organiz | dp / fast, diff to organize seq of comp

- Memoization is a technique for DP problems that preserves the recursive structure but does not do redundant computation.
- During the recursive calls newly computed solutions are stored in the DP data structure.
- The recursive procedure first checks if the solution has already been computed. If so it return the stored result.
- An advantage is that the exact computation sequence must not be programmed explicitly.

MATRIX-CHAIN MULTIPLICATION/11

Memoization/2

- Can structure solution also as a recursive algorithm

Algo: MatrixMemo(d,i,j)

if $i==j$ **then return** 0;
if $M[i,j] < \infty$ **then return** M[i,j];
for $k = i$ **to** j-1 **do**
  q = $d_{i-1} d_k d_j$ + MatrixMemo(d_0,...,d_n,i,k) +
      MatrixMemo(d_0,...,d_n,k+1,j);
  if $q<M[i,j]$ **then** M[i,j] = q;
return M;

MATRIX-CHAIN MULTIPLICATION/11

Memoization/2

- Can structure solution also as a recursive algorithm

Algo: MatrixMemo(d,i,j)

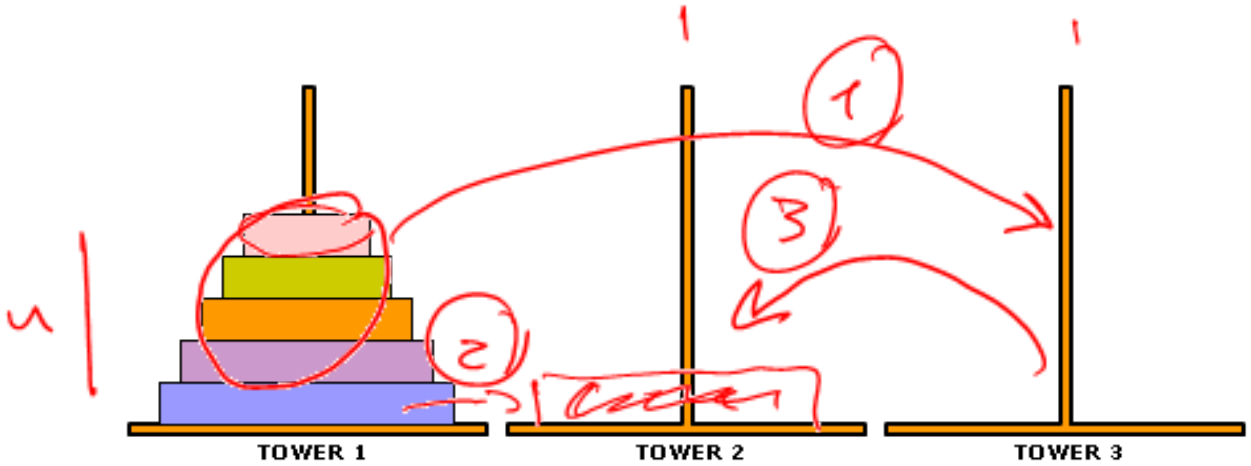
---

if i==j **then return** 0;
if $M[i,j] < \infty$ **then return** M[i,j];
for k = i **to** j-1 **do**
  q = $d_{i-1} d_k d_j$ + MatrixMemo(d_0,...,d_n,i,k) +
    MatrixMemo(d_0,...,d_n,k+1,j);
  if $q < M[i,j]$ **then** M[i,j] = q;
return M;

- Initialize all items to ∞ and run program ($i = 1$, $j = n$)

EXERCISE SL08-6/1

Use recursion and memoization to determine the number of moves that is required to solve the tower of Hanoi: moving all disks from peg 1 to peg 2 one disk at a time and never placing a larger disk on a smaller disk.



EXERCISE SL08-6/2

Algo:
```
HanoiRec(n, A, B, C)
  if (n==1) then return 1
  return HanoiRec(n-1, A, C, B) + 1 + HanoiRec(n-1, C, B, A)
```

exp runtime

$T(n) = 2 \cdot T(n-1) + 1$

Algo:
```
HanoiMemo(n, A, B, C)
  ⇒ if m[n] > 0 then return m[n];
    if (n==1) then return 1
  ⇒ m[n] = HanoiMemo(n-1, A, C, B) + 1 + HanoiMemo(n-1, C, B, A)
  return m[n]
```

linear $O(n)$

$m =$ | 1 | 3 | 7 | 22 | 176 | (--) | 1 | 1 |

1 2 3 4 5 6 7 n

TABLE OF CONTENTS — SL08

1. Design techniques

2. Matrix multiplication

3. Dynamic programming
   Dynamic programming in detail

4. Longest common subsequence, coin changing

DYNAMIC PROGRAMMING/1

- We typically apply dynamic programming to optimization problems.
- The goal is to find a solution with the optimal (minimum or maximum) value.
- Dynamic programming is used frequently in bioinformatics (sequence alignment) and economics (macroeconomic models).
- Dynamic programming solves problems by combining the solutions to subproblems.
- Dynamic programming applies when subproblems overlap.
- A dynamic-programming algorithm solves each subproblem just once.

DYNAMIC PROGRAMMING/2

- **Step 1**: Show optimal substructure – an optimal solution to a problem contains optimal solutions to subproblems
  - Solution to a problem
    - Making a choice out of a number of possibilities (look what possible choices there can be)
    - Solving one or more subproblems that are the result of such a choice (characterize the space of subproblems)
  - Show that solutions to subproblems must themselves be optimal for the whole solution to be optimal

DYNAMIC PROGRAMMING/2

- **Step 1**: Show optimal substructure – an optimal solution to a problem contains optimal solutions to subproblems
  - Solution to a problem
    - Making a choice out of a number of possibilities (look what possible choices there can be)
    - Solving one or more subproblems that are the result of such a choice (characterize the space of subproblems)
  - Show that solutions to subproblems must themselves be optimal for the whole solution to be optimal

- **Step 2**: Give a recursive problem formulation for getting the optimal value
  - $M_{opt} = \min_k\{(\text{combination of } M_{opt} \text{ of all subproblems resulting from the choice } k) + (\text{cost associated with making the choice } k)\}$

DYNAMIC PROGRAMMING/3

- **Step 3**: Compute the value of an optimal solution in a bottom-up fashion (or use memoization)

DYNAMIC PROGRAMMING/3

- **Step 3**: Compute the value of an optimal solution in a bottom-up fashion (or use memoization)
- Show that the number of different instances of subproblems is bounded by a polynomial
- Check whether it is possible to reduce the space requirements (forget no longer needed solutions)

(a matrix can be too expensive (n^2, 10^7))

Dynamic programming/3

- **Step 3**: Compute the value of an optimal solution in a bottom-up fashion (or use memoization)
- Show that the number of different instances of subproblems is bounded by a polynomial
- Check whether it is possible to reduce the space requirements (forget no longer needed solutions)

- **Step 4**: If required construct an optimal solution from the computed information (sequence of choices made)

Dynamic programming/3

- **Step 3**: Compute the value of an optimal solution in a bottom-up fashion (or use memoization)
- Show that the number of different instances of subproblems is bounded by a polynomial
- Check whether it is possible to reduce the space requirements (forget no longer needed solutions)

- **Step 4**: If required construct an optimal solution from the computed information (sequence of choices made)

TABLE OF CONTENTS — SL08

1. Design techniques

2. Matrix multiplication

3. Dynamic programming

4. Longest common subsequence, coin changing
   - Longest common subsequence (LCS)
   - Coin changes

LONGEST COMMON SUBSEQUENCE/1

Problem definition

- Two text strings are given: X and Y

LONGEST COMMON SUBSEQUENCE/1

Problem definition

- Two text strings are given: X and Y
- Want to quantify how similar they are
  - Spell checkers, DNA sequences

LONGEST COMMON SUBSEQUENCE/1

Problem definition

- Two text strings are given: X and Y
- Want to quantify how similar they are
  - Spell checkers, DNA sequences
- One of the measures of similarity is the length of a longest common subsequence (LCS)

LONGEST COMMON SUBSEQUENCE/1

Problem definition

- Two text strings are given: X and Y
- Want to quantify how similar they are
  - Spell checkers, DNA sequences
- One of the measures of similarity is the length of a longest common subsequence (LCS)
- Z is a subsequence of X if it is possible to generate Z by skipping some (possibly none) characters from X
  - LCS ("ACGGTTA", "CGTAT") = "CGTA" or "CGTT"

LONGEST COMMON SUBSEQUENCE/1

Problem definition

- Two text strings are given: X and Y
- Want to quantify how similar they are
  - Spell checkers, DNA sequences
- One of the measures of similarity is the length of a longest common subsequence (LCS)
- Z is a subsequence of X if it is possible to generate Z by skipping some (possibly none) characters from X
  - LCS ("ACGGTTA", "CGTAT") = "CGTA" or "CGTT"
- To solve the LCS problem one has to find
  - "skips" that generate LCS (X, Y) from X
  - "skips" that generate LCS (X, Y) from Y

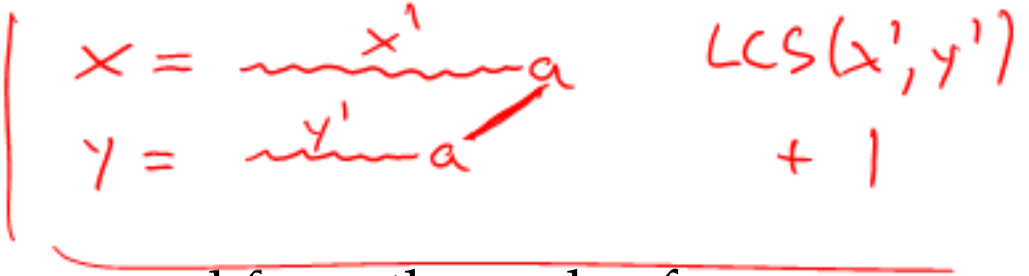
LONGEST COMMON SUBSEQUENCE/2

Optimal substructure

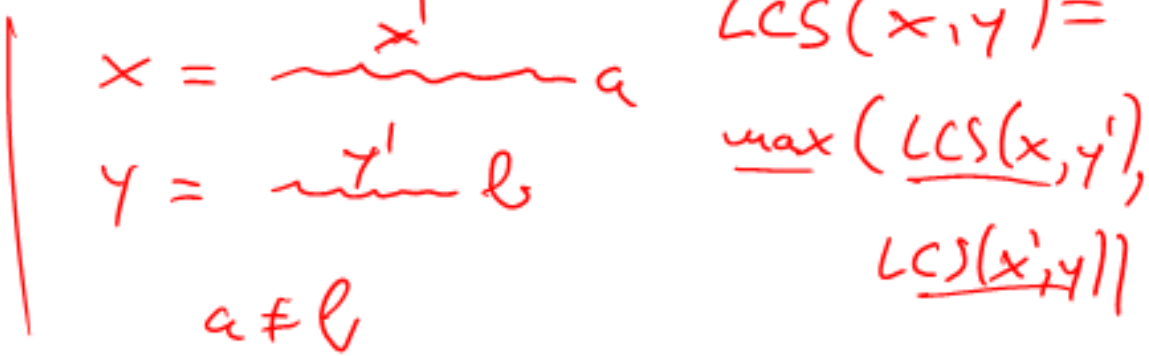
- Start with empty Z and proceed from the ends of $X_n = x_1 x_2 \dots x_n$ and $Y_m = y_1 y_2 \dots y_m$

Longest common subsequence/2

Optimal substructure



- Start with empty Z and proceed from the ends of $X_n = x_1 x_2 \ldots x_n$ and $Y_m = y_1 y_2 \ldots y_m$
- One needs to consider two cases:
  - If $x_n = y_m$, append this symbol to the beginning of Z and find an optimal solution for LCS (X_{n-1}, Y_{m-1})
  - If $x_n \neq y_m$, either skip a letter from X_n or skip a letter from Y_m and decide about the choice by comparing LCS (X_{n-1}, Y_m) and LCS (X_n, Y_{m-1})

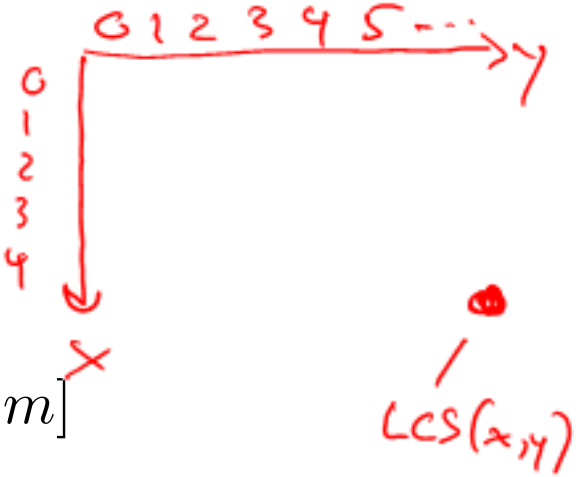


LONGEST COMMON SUBSEQUENCE/2

Optimal substructure

- Start with empty Z and proceed from the ends of $X_n = x_1 x_2 \ldots x_n$ and $Y_m = y_1 y_2 \ldots y_m$
- One needs to consider two cases:
  - If $x_n = y_m$, append this symbol to the beginning of Z and find an optimal solution for LCS (X_{n-1}, Y_{m-1})
  - If $x_n \neq y_m$, either skip a letter from X_n or skip a letter from Y_m and decide about the choice by comparing LCS (X_{n-1}, Y_m) and LCS (X_n, Y_{m-1})
- The algorithm can be extended to allow more editing operations (e.g. changing a letter)

LONGEST COMMON SUBSEQUENCE/3

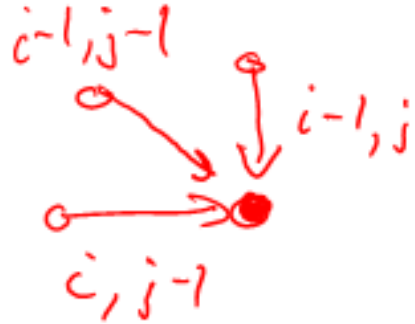


Recursive problem formulation

- Let $c[i,j] = \text{LCS}\,(X_i, Y_j)$ with $c[0\ldots n, 0\ldots m]$

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \vee j = 0 \\ c[i-1, j-1] + 1 & \text{if } i,j > 0 \wedge x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\} & \text{if } i,j > 0 \wedge x_i \neq y_j \end{cases}$$

LONGEST COMMON SUBSEQUENCE/3



Recursive problem formulation

- Let $c[i,j] = \text{LCS}\,(X_i, Y_j)$ with $c[0\ldots n, 0\ldots m]$

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \vee j = 0 \\ c[i-1, j-1] + 1 & \text{if } i,j > 0 \wedge x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\} & \text{if } i,j > 0 \wedge x_i \neq y_j \end{cases}$$

- The conditions in the problem restrict the subproblems
  - If $x_i = y_j$, one considers the subproblem of finding the LCS of X_{i-1} and Y_{j-1}
  - If $x_i \neq y_j$, one considers the subproblems of finding the LCS of X_{i-1} and Y_j and of X_i and Y_{j-1}

EXERCISE SL08-7

Give a recursive solution for solving the LCS problem.

```
LCSrec(i, j)
    if (i==0 || j==0) return 0
    else if (x[i] = y[j]) return LCSrec(i-1, j-1)+1
    else return max(LCSrec(i-1, j),
                    LCSrec(i, j-1))
```

EXERCISE SL08-8

Give a memoization solution for solving the LCS problem.

C: c[i][j]

pseudocode:

c[i,j]

```
LCSmemo(i, j)
  if (i==0 || j==0) return 0
  else if (c[i,j] ≠ -1) return c[i,j]
  else if (x[i] == y[j]) { c[i,j] = LCSmemo(i-1, j-1) + 1; return c[i,j] }
  else { c[i,j] = max(LCSmemo(i-1, j), LCSmemo(i, j-1)); return c[i,j] }
```

EXERCISE SL08-9

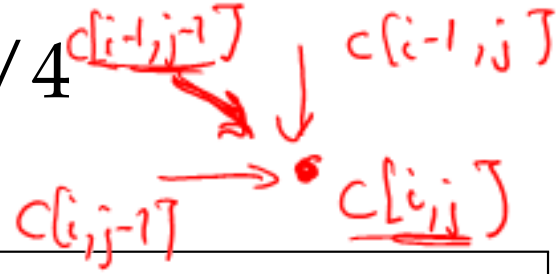
$$c[i,j] = c[i-1,j-1] + 1$$

$$c[i,j] = \max(c[i-1,j], c[i,j-1])$$

Determine the length of LCS ("GGTTCAT", "GTATCT").

		G	T	A	T	C	T
	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1
G	0	1	1	1	1	1	1
T	0	1	2	2	2	2	2
T	0	1	2	2	3	3	3
C	0	1	2	2	3	4	4
A	0	1	2	3	3	4	4
T	0	1	2	3	4	4	5

LONGEST COMMON SUBSEQUENCE / 4



DP algorithm:

Algo: LCSdyn(X_n,Y_m)

---

for $i = 1$ *to* n **do** c[i,0] = 0;
for $j = 0$ *to* m **do** c[0,j] = 0;
for $i = 1$ *to* n **do** — all rows
  for $j = 1$ *to* m **do** — all cols
    if $x_i == y_j$ **then** — match
      c[i,j] = c[i-1,j-1] + 1
    else
      if *c[i-1,j]≥c[i,j-1]* **then** — skip X
        c[i,j] = c[i-1,j]
      else — skip Y
        c[i,j] = c[i,j-1]

— take max

return c;

EXERCISE SL08-10

Print the longest common subsequence of two given text strings.



```
printLCS(x, y, i, j)
  if (i==0 || j==0) return
  if (c[i-1,j] == c[i,j] then printLCS(x, y, i-1, j)
  else if (c[i,j-1] == c[i,j]) then printLCS(x, y, i, j-1)
  else printLCS(x, y, i-1, j-1)
       printf("%c", x[i])
```



Coin Changes/1

Problem illustration

- Assume you have coins of different denominations, e.g., 5, 15, 20, 50.
- Assume that of each denomination you have an unlimited number of coins.
- Given an amount A and m denominations the goal is to change A with as few coins as possible.
- Example:
  - Denominations: 1, 2, 5, 10
  - A = 27, change = 2*10 + 1*5 + 1*2

Coin Changes/2

Problem definition

Given are coins of different denominations (e.g., 1, 5, 15, 20, 50, etc). The goal is to pay out amount A with the smallest possible numbers of coins. Develop a solution.

Three cases:

1. No solution: e.g., A=3 and the only available coin is 2.
2. Greedy solution (i.e., in each step use the highest possible coin) gives the optimal solution. Typically, denominations are designed in such a way.
3. DP solution if the optimal solution cannot be found by choosing the largest possible coin in each step (e.g., A=12, coins = (10, 6, 1))

greedy: 10 + 1 + 1 optimal: 6 + 6

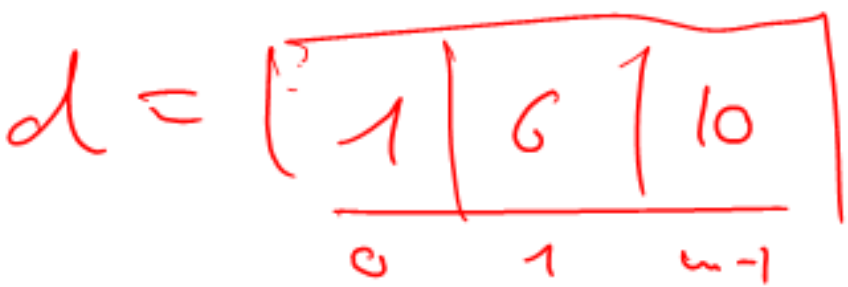
EXERCISE SL08-11/1

Optimal substructure:

if the optimal solution to change amount A includes coin X then the coin changes for A-X must also be optimal

Problem representation:

We assume an array with denominations



EXERCISE SL08-11/2

Recursive problem formulation:

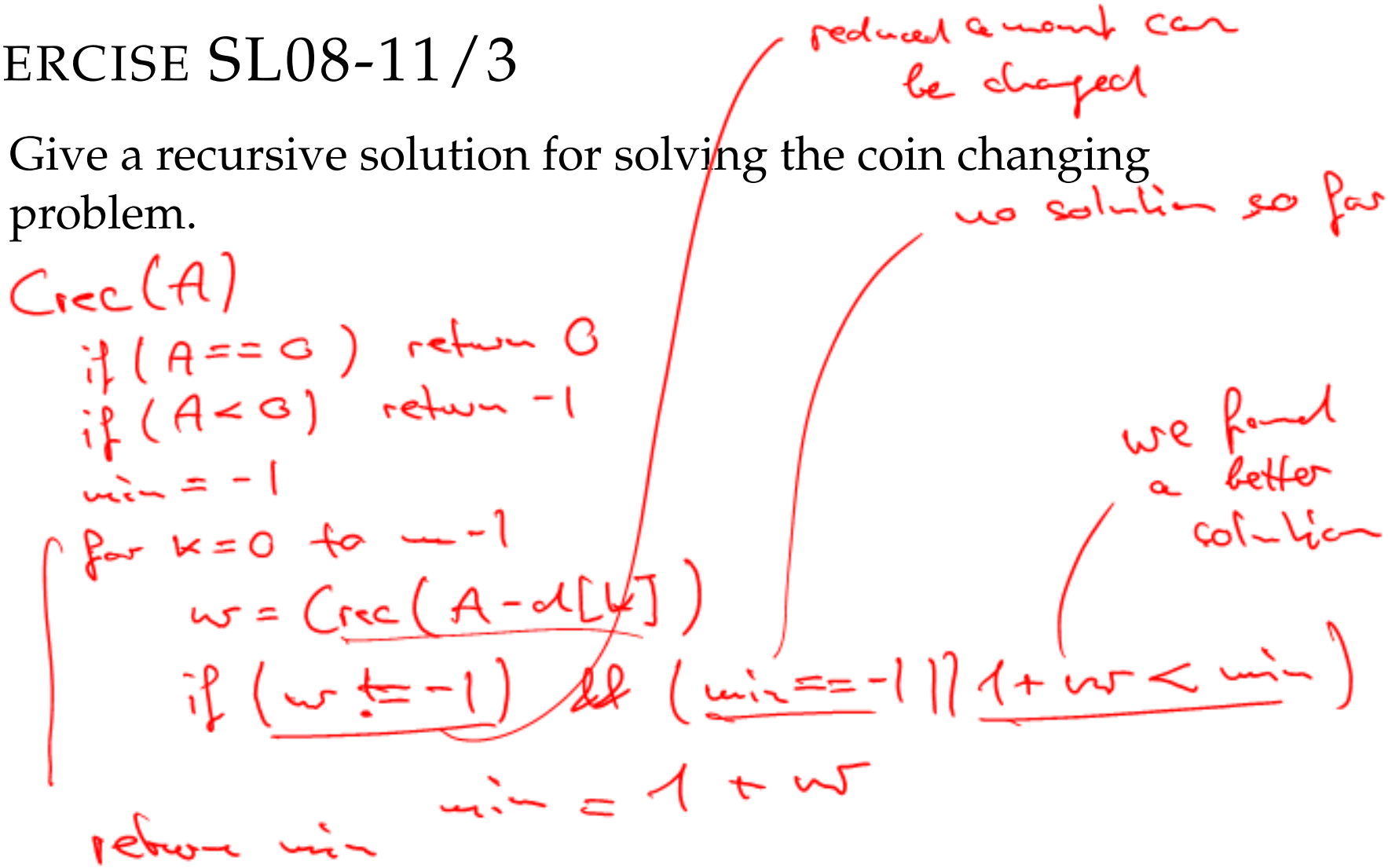
$$c(i) = \begin{cases} 0 & \text{if } i = 0 \\ \min_{0 \le k < m} \{ c(i - d[k]) + 1 \} & \text{if } i > 0 \end{cases}$$

Example: change of amount $A = 20$ with coins 10, 6, 1

$$c(20) = \min \begin{cases} c(20-10)+1 \\ c(20-6)+1 \\ c(20-1)+1 \end{cases}$$

EXERCISE SL08-11/3

Give a recursive solution for solving the coin changing problem.



EXERCISE SL08-11/4

Give a recursive solution with memoizationfor solving the coin changing problem.

```
Cmem(A)                                          20
  if (A==0) return 0                             19
  if(A<0) return -1
| if(c[A] != -1) return c[A]
  for k=0 to m-1
      w = Cmem(A - d[k])
      if (w != -1) && (c[A] == -1 || 1+w < c[A])
          c[A] = 1 + w
  return c[A]
```

EXERCISE SL08-11/5

Give a dynamic programming solution for solving the coin changing problem.

```
Cdyn
    c[0] = 0
    for i=1 to A do
        c[i] = -1
        for k=0 to m-1
            if (d[k] ≤ i && c[i-d[k]] != -1) &&
               (c[i] == -1 || 1 + c[i-d[k]] < c[i])
                c[i] = 1 + c[i-d[k]]
```

SUMMARY

- Dynamic programming (DP) is an important algorithmic design technique.
- DP is used to solve optimization problems.
- DP applies to problems that can be subdivided into optimal subproblems.

- First, come up with a recursive problem formulation.
- Organize computation in a bottom-up fashion.
- Precisely specify meanings of cells in the DP table.

Informatik II

Graph Algorithms
— SL09 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

TABLE OF CONTENTS — SL09

1. Graphs
   Definition and terminology
   Graph representation
   Graph search
2. Breadth-first search (BFS)
3. Depth-first search (DFS)
4. Topological sorting

Definition and terminology/1

- A **graph** $G = (V, E)$ consists of
  - a finite set of **vertices** V (vertex = node)
  - a set of edges $E \subseteq V \times V$

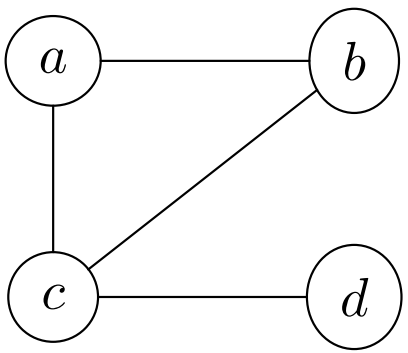
Definition and terminology/1

- A **graph** $G = (V, E)$ consists of
  - a finite set of **vertices** V
  - a set of edges $E \subseteq V \times V$
- An **edge** $e = (u, v)$ is a pair of vertices

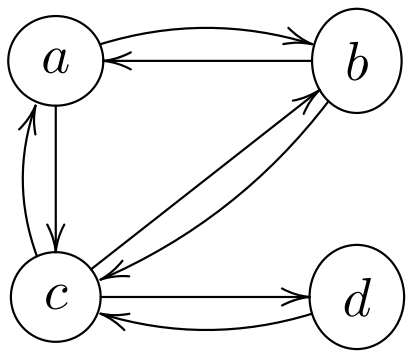
ordered:
edge from u to v

Definition and terminology/1

- A **graph** $G = (V, E)$ consists of
  - a finite set of **vertices** V
  - a set of edges $E \subseteq V \times V$
- An **edge** $e = (u, v)$ is a pair of vertices
- Generally assume **directed graphs** where edges are directed (go from one node to another node)
- **Undirected graphs**:
  - edges are unordered pairs $\{u, v\}$
  - $V = \{a, b, c, d\}$ & $E = \{\{a, b\}, \{a, c\}, \{b, c\}, \{c, d\}\}$



versus

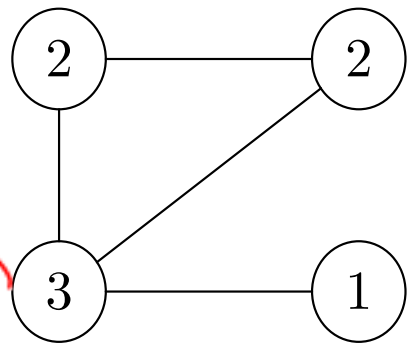


Definition and terminology/2

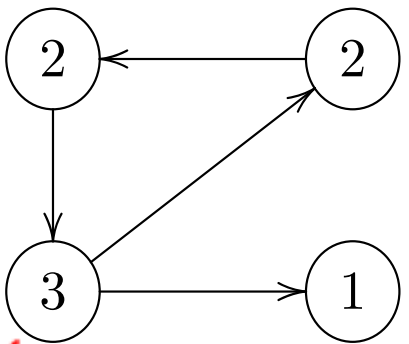
- Vertex v is **adjacent** to vertex u iff $(u, v) \in E$

Definition and terminology/2

- Vertex v is **adjacent** to vertex u iff $(u, v) \in E$
- The **degree** of a vertex is defined as follows
  - number of adjacent vertices (undirected graph)
  - number of incoming and outgoing edges (directed graph)
    - **in-degree**: number of incoming edges
    - **out-degree**: number of outgoing edges



versus

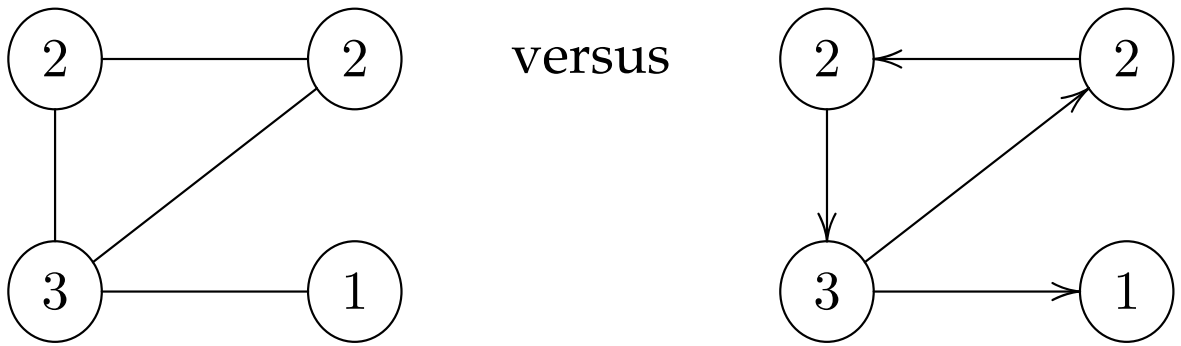


in-degree = out-degree = 3

in-degree = 1
out-degree = 2

Definition and terminology/2

- Vertex v is **adjacent** to vertex u iff $(u, v) \in E$
- The **degree** of a vertex is defined as follows
  - number of adjacent vertices (undirected graph)
  - number of incoming and outgoing edges (directed graph)
    - **in-degree**: number of incoming edges
    - **out-degree**: number of outgoing edges



- A **path** of length k is a sequence $\langle v_0, \ldots, v_k \rangle$ of vertices such that, for all $i \in \{1, \ldots, k\}$, v_i is adjacent to v_{i-1}

Definition and terminology / 3

- **Simple path**: path $\langle v_0, \ldots, v_k \rangle$ with $v_i \neq v_j$ for $i \neq j$

Definition and terminology / 3

- **Simple path**: path $\langle v_0, \ldots, v_k \rangle$ with $v_i \neq v_j$ for $i \neq j$
- **Cycle**: a simple path $\langle v_0, \ldots, v_k \rangle$, except that $v_k = v_0$

Definition and terminology / 3

- **Simple path**: path $\langle v_0, \ldots, v_k \rangle$ with $v_i \neq v_j$ for $i \neq j$
- **Cycle**: a simple path $\langle v_0, \ldots, v_k \rangle$, except that $v_k = v_0$
- **Connected graph**: every pair of vertices is connected by a path

Definition and terminology / 3

- **Simple path**: path $\langle v_0, \ldots, v_k \rangle$ with $v_i \neq v_j$ for $i \neq j$
- **Cycle**: a simple path $\langle v_0, \ldots, v_k \rangle$, except that $v_k = v_0$
- **Connected graph**: every pair of vertices is connected by a path
- **Subgraph**: subset of vertices and edges that forms a graph

Definition and terminology / 3

- **Simple path**: path $\langle v_0, \ldots, v_k \rangle$ with $v_i \neq v_j$ for $i \neq j$
- **Cycle**: a simple path $\langle v_0, \ldots, v_k \rangle$, except that $v_k = v_0$
- **Connected graph**: every pair of vertices is connected by a path
- **Subgraph**: subset of vertices and edges that forms a graph
- **Connected component**: maximal connected subgraph

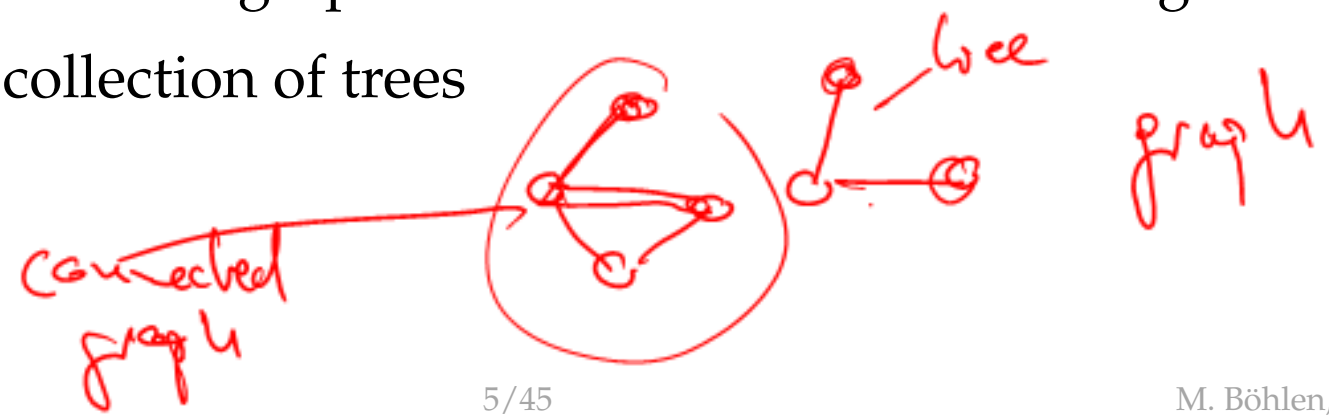
Definition and terminology / 3

- **Simple path**: path $\langle v_0, \ldots, v_k \rangle$ with $v_i \neq v_j$ for $i \neq j$
- **Cycle**: a simple path $\langle v_0, \ldots, v_k \rangle$, except that $v_k = v_0$
- **Connected graph**: every pair of vertices is connected by a path
- **Subgraph**: subset of vertices and edges that forms a graph
- **Connected component**: maximal connected subgraph
- **Tree**: connected graph with n vertices and $n - 1$ edges

Definition and terminology / 3

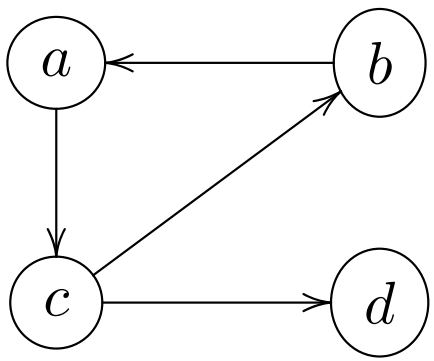
all edges have the same weight / cost / length
= (length = 1) = unweighted graph

- **Simple path**: path $\langle v_0, \ldots, v_k \rangle$ with $v_i \neq v_j$ for $i \neq j$
- **Cycle**: a simple path $\langle v_0, \ldots, v_k \rangle$, except that $v_k = v_0$
- **Connected graph**: every pair of vertices is connected by a path
- **Subgraph**: subset of vertices and edges that forms a graph
- **Connected component**: maximal connected subgraph
- **Tree**: connected graph with n vertices and $n - 1$ edges
- **Forest**: collection of trees



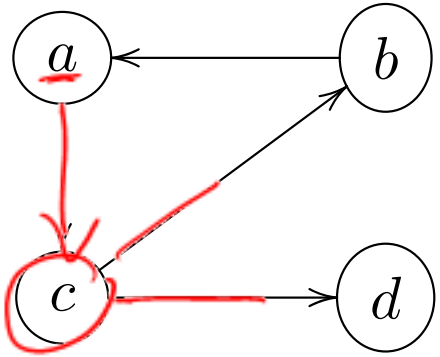
Graph representation/1

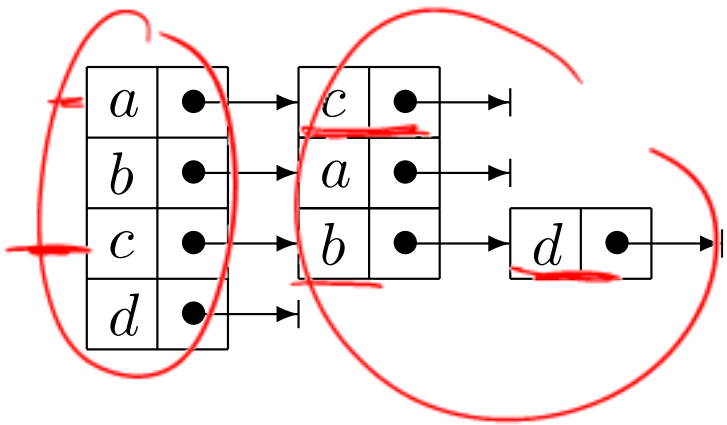
- Adjacency lists



Graph representation/1

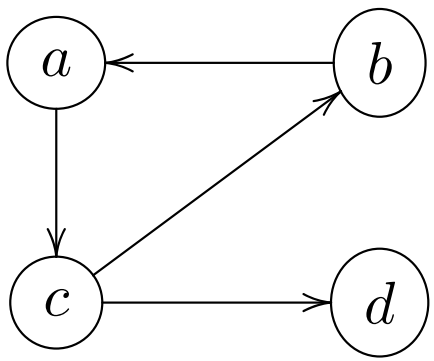
- Adjacency lists

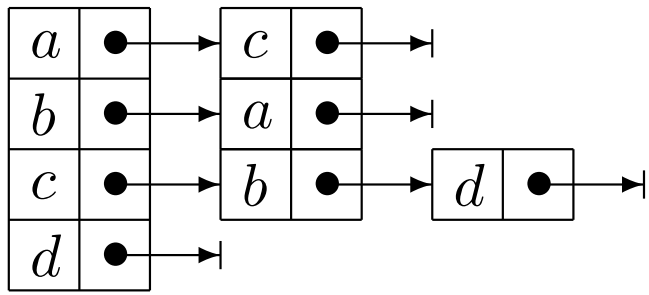




Graph representation/1

- Adjacency lists

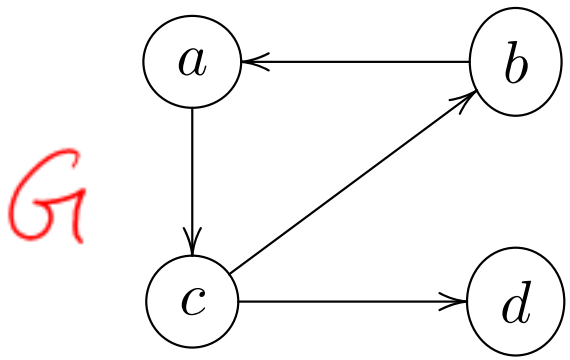


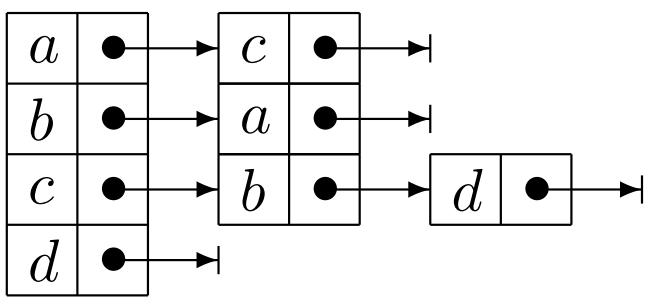


- Adjacency list of v: sequence of vertices adjacent to it

Graph representation/1

- Adjacency lists

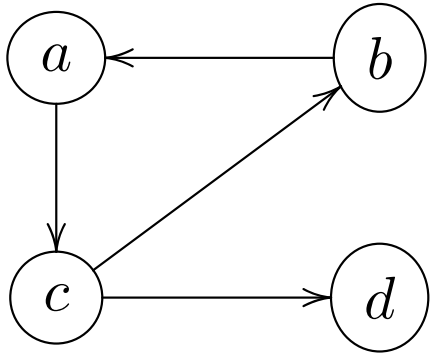


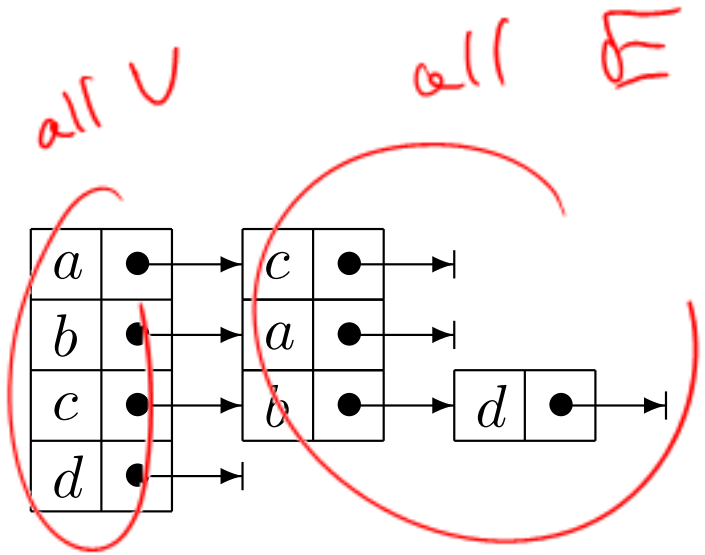


- Adjacency list of v: sequence of vertices adjacent to it
- Represent graph by the adjacency lists of its vertices

Graph representation/1

- Adjacency lists

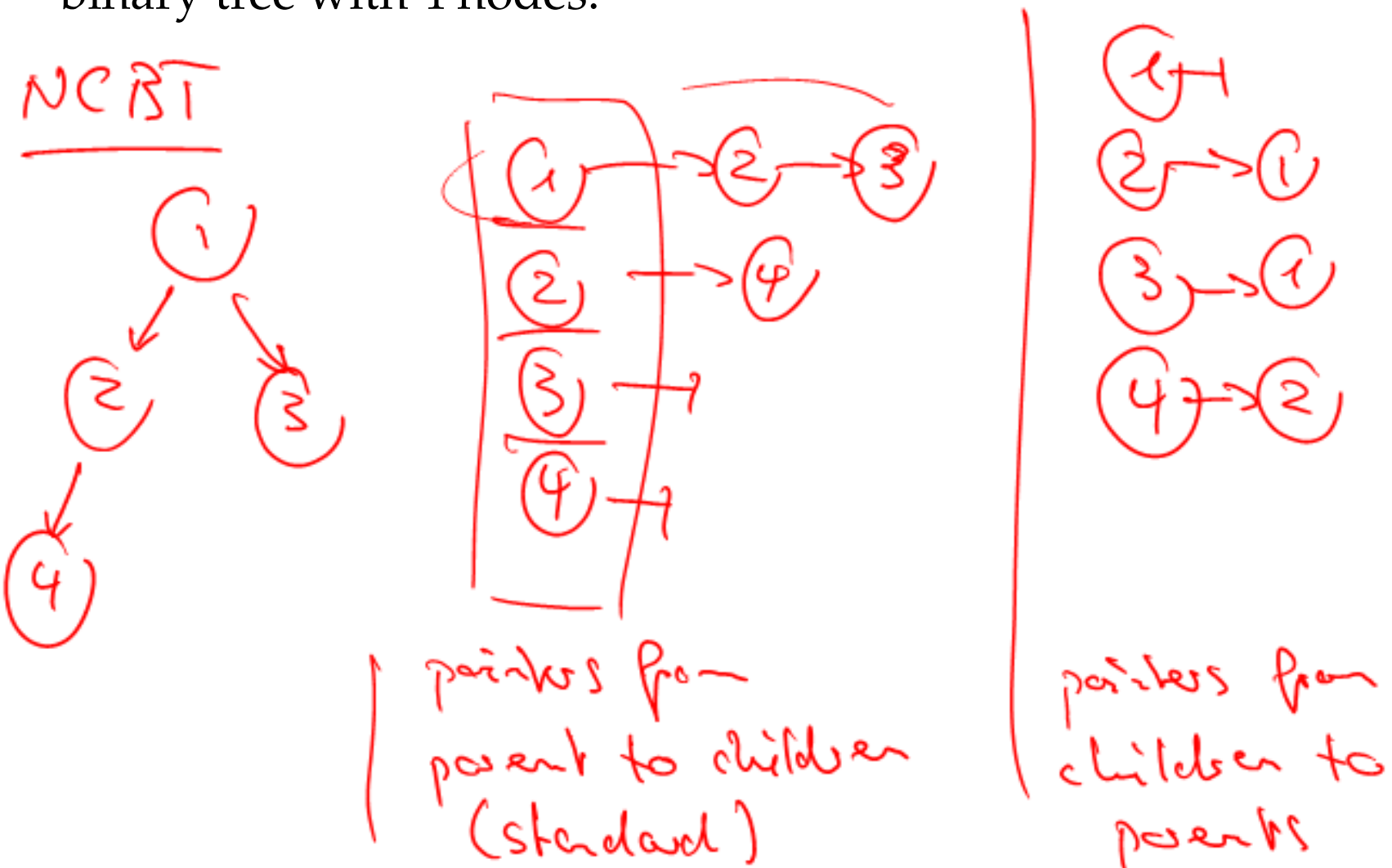




- Adjacency list of v: sequence of vertices adjacent to it
- Represent graph by the adjacency lists of its vertices
- Space requirements: $\Theta(|V| + |E|)$

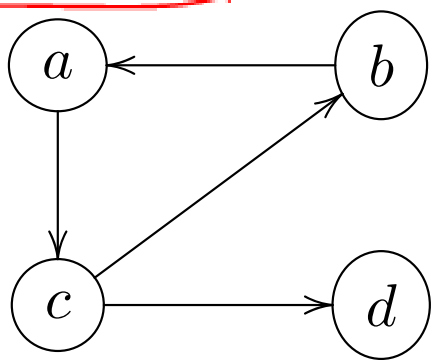
EXERCISE SL09-1

Give an adjacency list representation of a nearly complete binary tree with 4 nodes.



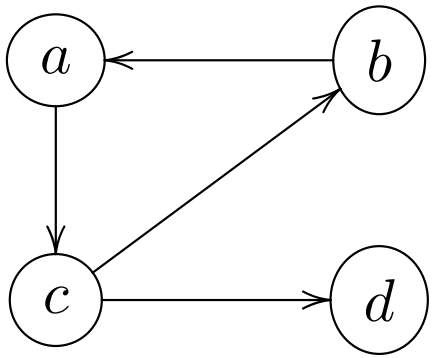
Graph representation/2

- Adjacency matrix



Graph representation/2

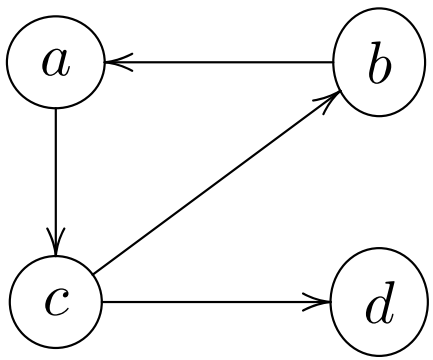
- Adjacency matrix



	a	b	c	d
a	0	0	1	0
b	1	0	0	0
c	0	1	0	1
d	0	0	0	0

Graph representation/2

- Adjacency matrix

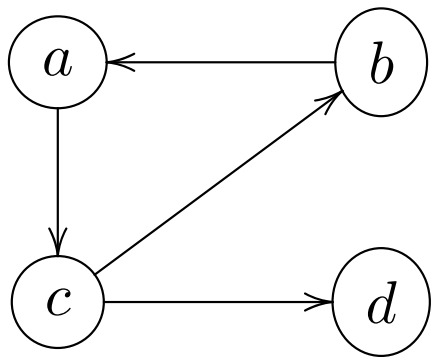


	a	b	c	d
a	0	0	1	0
b	1	0	0	0
c	0	1	0	1
d	0	0	0	0

- Represent graph by an adjacency matrix $M(|V| \times |V|)$
  - $M[x, y] = 1$ if there is an edge $e = (x, y)$ in the graph
  - $M[x, y] = 0$ if there is no edge $e = (x, y)$ in the graph

Graph representation/2

- Adjacency matrix



	a	b	c	d
a	0	0	1	0
b	1	0	0	0
c	0	1	0	1
d	0	0	0	0

- Represent graph by an adjacency matrix $M(|V| \times |V|)$
  - $M[x, y] = 1$ if there is an edge $e = (x, y)$ in the graph
  - $M[x, y] = 0$ if there is no edge $e = (x, y)$ in the graph
- Space requirements: $\Theta(|V|^2)$

good if not fully connected

G1:

good if fully connected

connected

fully connected

compare to

$\Theta(|V| + |E|)$

$|V| - 1 \leq |E| \leq |V|^2$

EXERCISE SL09-2

Consider an adjacency list representation of a directed graph. How long does it take to compute the in- and out-degree of a vertex?

directed graph (edges from parent to children):

- out-degree: scan 1 adj list

on avg E/V with $E \leq V^2$

we get $O(V)$

- in-degree: scan all adj lists

$E \leq |V|^2$ we get $O(V^2)$

undirected graph (edges in two directions): out-deg = in-deg $O(V)$

GRAPH SEARCH/1

Graph search algorithms

- Systematic search of every edge and vertex of a graph

GRAPH SEARCH/1

Graph search algorithms

- Systematic search of every edge and vertex of a graph
- Applications: compilers, maze solving, network routing

GRAPH SEARCH/1

Graph search algorithms

- Systematic search of every edge and vertex of a graph
- Applications: compilers, maze solving, network routing

Pseudocode conventions

- Each node has certain properties (fields of a record)
  - adj: list of adjacent nodes
  - col: color of the node (B, G, W)
  - $pred$: predecessor node in traversal
  - $dist$: distance from the start node (BFS)
  - $stime$: time when a node is first visited (DFS)
  - $etime$: time when a node is last visited (DFS)
  - ... (more fields as required by application)

GRAPH SEARCH/2

Coloring is used to color vertices during a traversal. The colors allow to infer properties about the graph.

- Color changes during traversal

GRAPH SEARCH/2

Coloring is used to color vertices during a traversal. The colors allow to infer properties about the graph.

- Color changes during traversal
- Vertex v is white (i.e., $v.col = W$)
  - Means that v is undiscovered

GRAPH SEARCH/2

Coloring is used to color vertices during a traversal. The colors allow to infer properties about the graph.

- Color changes during traversal
- Vertex v is white (i.e., $v.col = W$)
  - Means that v is undiscovered
- Vertex v is gray (i.e., $v.col = G$)
  - Means that v has been discovered but not all adjacent vertices have been explored

GRAPH SEARCH/2

Coloring is used to color vertices during a traversal. The colors allow to infer properties about the graph.

- Color changes during traversal
- Vertex v is white (i.e., $v.col = W$)
  - Means that v is undiscovered
- Vertex v is gray (i.e., $v.col = G$)
  - Means that v has been discovered but not all adjacent vertices have been explored
- Vertex v is black (i.e., $v.col = B$)
  - Means that v has been discovered and all the adjacent vertices have been explored
  - In other words, the adjacency list of v was examined completely

TABLE OF CONTENTS — SL09

1. Graphs

2. Breadth-first search (BFS)
   - BFS algorithm
   - BFS example
   - BFS properties

3. Depth-first search (DFS)

4. Topological sorting

BFS ALGORITHM/1

- A **breadth-first search** (BFS) starts at vertex s and visits all vertices reachable from s
  - In doing so BFS defines a **spanning tree** (a tree that includes all vertices).

BFS ALGORITHM/1

- A **breadth-first search** (BFS) starts at vertex s and visits all vertices reachable from s
  - In doing so BFS defines a **spanning tree** (a tree that includes all vertices).
- BFS is like walking in a labyrinth with a string and exploring the **entire** neighborhood first

go in all directions

BFS ALGORITHM/1

- A **breadth-first search** (BFS) starts at vertex s and visits all vertices reachable from s
  - In doing so BFS defines a **spanning tree** (a tree that includes all vertices).
- BFS is like walking in a labyrinth with a string and exploring the **entire** neighborhood first
- The starting vertex s is assigned distance 0

BFS ALGORITHM/1

- A **breadth-first search** (BFS) starts at vertex s and visits all vertices reachable from s
  - In doing so BFS defines a **spanning tree** (a tree that includes all vertices).
- BFS is like walking in a labyrinth with a string and exploring the **entire** neighborhood first
- The starting vertex s is assigned distance 0
- In the first round the string is unrolled 1 unit
  - All vertices that are one edge away from the starting vertex s are visited and assigned distance 1

BFS ALGORITHM/2

- In the second round the string is unrolled 2 units
  - All vertices that are two edges away from the starting vertex s are visited and assigned distance 2
  - Thus, all the vertices that can be reached by unrolling the string 2 units are assigned distance 2

BFS ALGORITHM/2

- In the second round the string is unrolled 2 units
  - All vertices that are two edges away from the starting vertex s are visited and assigned distance 2
  - Thus, all the vertices that can be reached by unrolling the string 2 units are assigned distance 2
- This continues until every reachable vertex has been assigned a distance

possibly not all nodes are visited

BFS ALGORITHM/2

- In the second round the string is unrolled 2 units
  - All vertices that are two edges away from the starting vertex s are visited and assigned distance 2
  - Thus, all the vertices that can be reached by unrolling the string 2 units are assigned distance 2
- This continues until every reachable vertex has been assigned a distance
- The label of any vertex v corresponds to the length of the shortest path from s to v

BFS ALGORITHM/3

Algo: BFS(G,s)

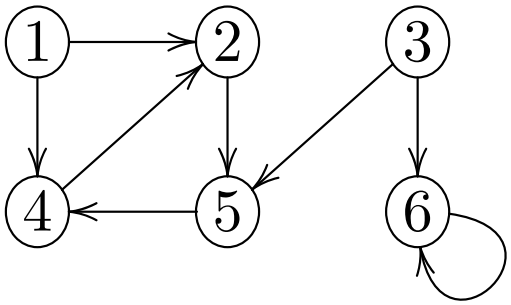
foreach $v \in G.V$ **do**
  v.col = W; v.dist = ∞; v.pred = NIL
s.col = G; s.dist = 0;
InitQueue(Q);
Enqueue(Q,s);
while $Q \neq \emptyset$ **do**
  v = Dequeue(Q);
  foreach $u \in v.adj$ **do**
    if *u.col*==*W* **then**
      u.col = G; u.dist = v.dist+1; u.pred = v; Enqueue(Q,u)
  v.col = B;

initialization

Q: FIFO queue
- shorter paths are before longer paths

EXERCISE SL09-3

Show the distances and predecessors that result from a BFS starting at node 3.



Illustrate Q during the BFS. Determine the size of Q in the general case.

node id

distance from start

3|0

5|1 6|1

4|2

2|3

Q
3
6 5
5
4
2
–

Q
3
5 6
6 4
4
2
–

f = fanout (= nr of adj nodes)

d = depth of tree

of nodes at level d = f^d

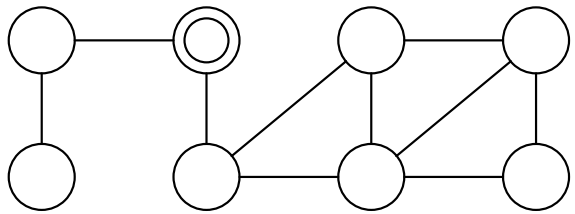
(expensive, exp-, bounded by |V|)

Q at beginning of the while

BFS EXAMPLE/1

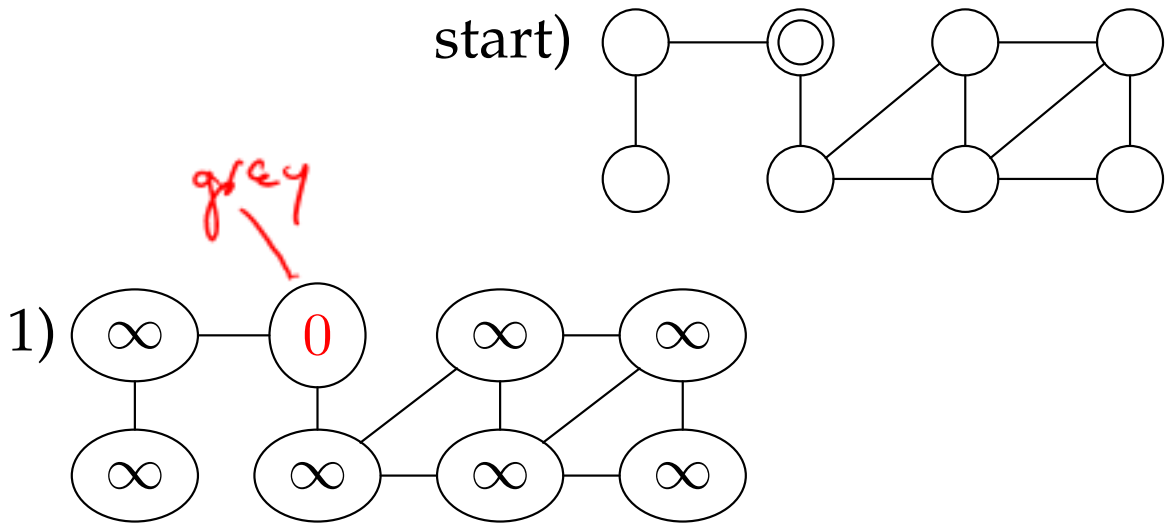
BFS example/1

start)



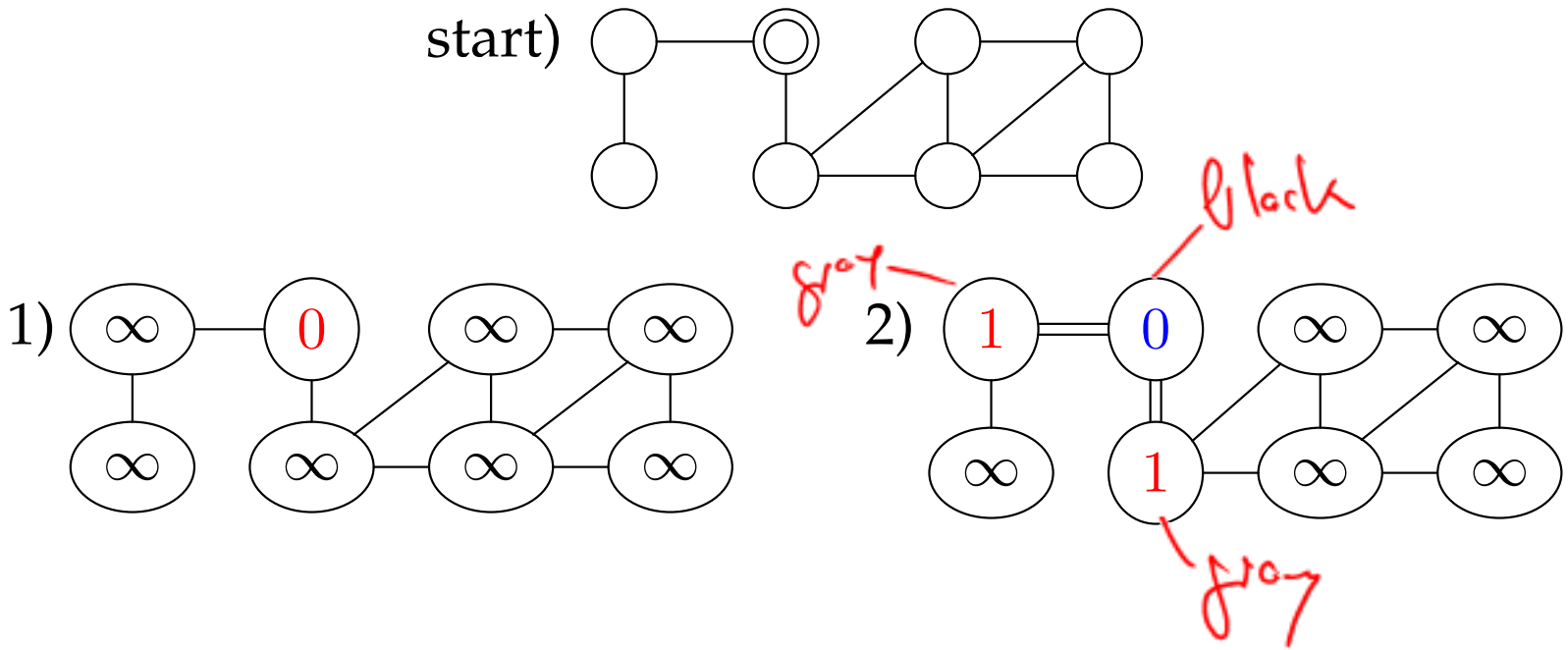
BFS EXAMPLE/1

BFS example/1



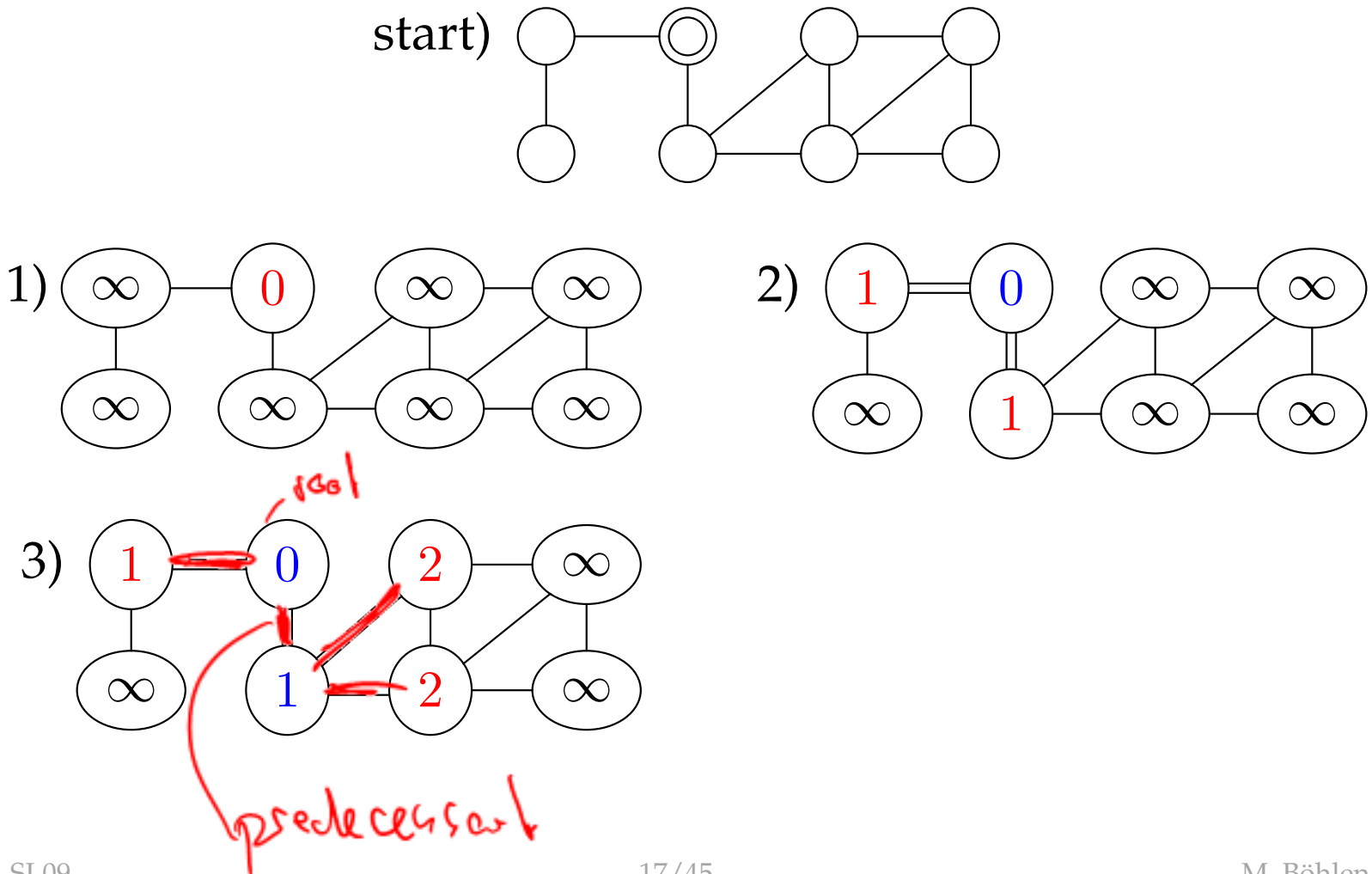
BFS EXAMPLE/1

BFS example/1



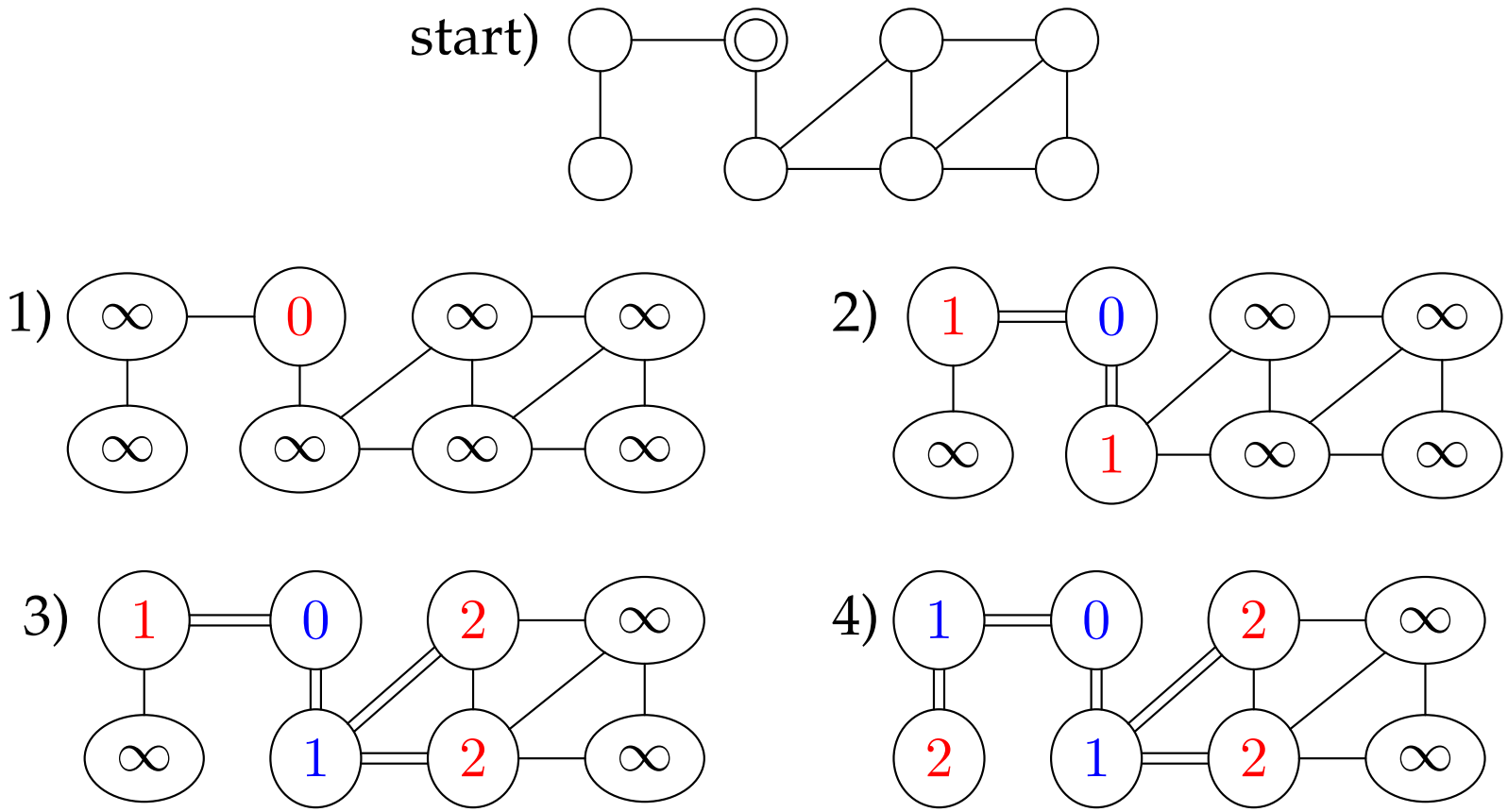
BFS EXAMPLE/1

BFS example/1



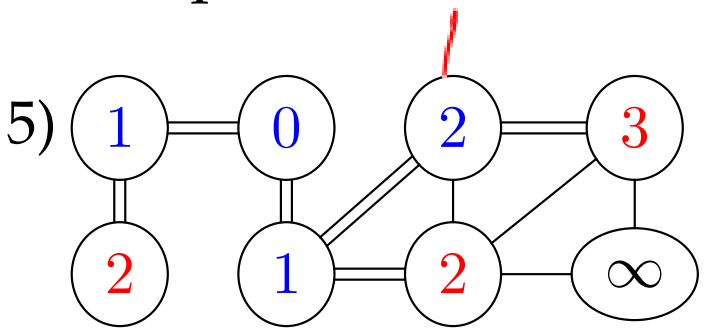
BFS EXAMPLE/1

BFS example/1



BFS EXAMPLE/2

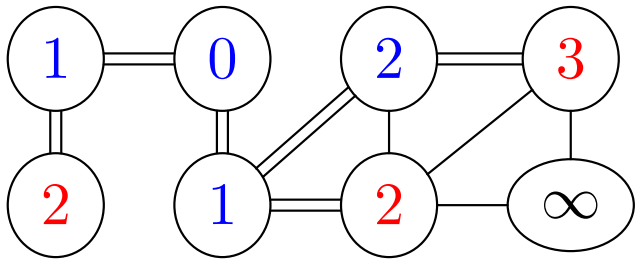
BFS example/2



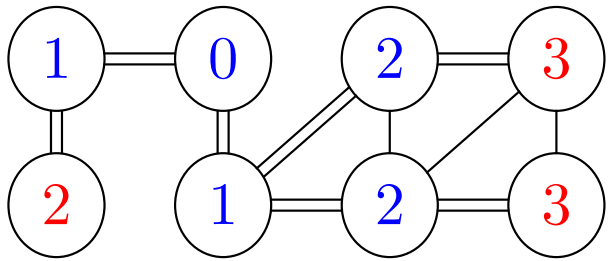
BFS EXAMPLE/2

BFS example/2

5)

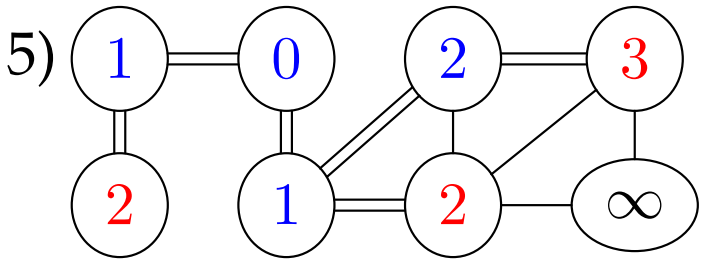


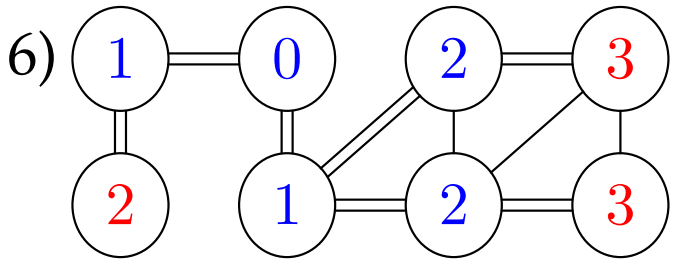
6)

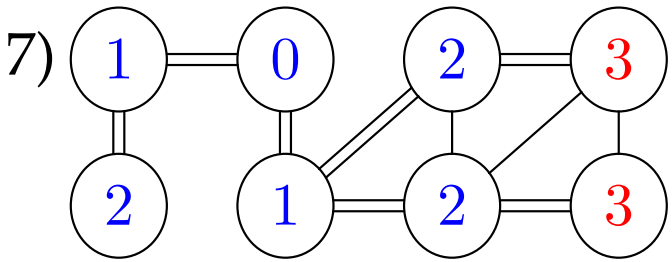


BFS EXAMPLE/2

BFS example/2

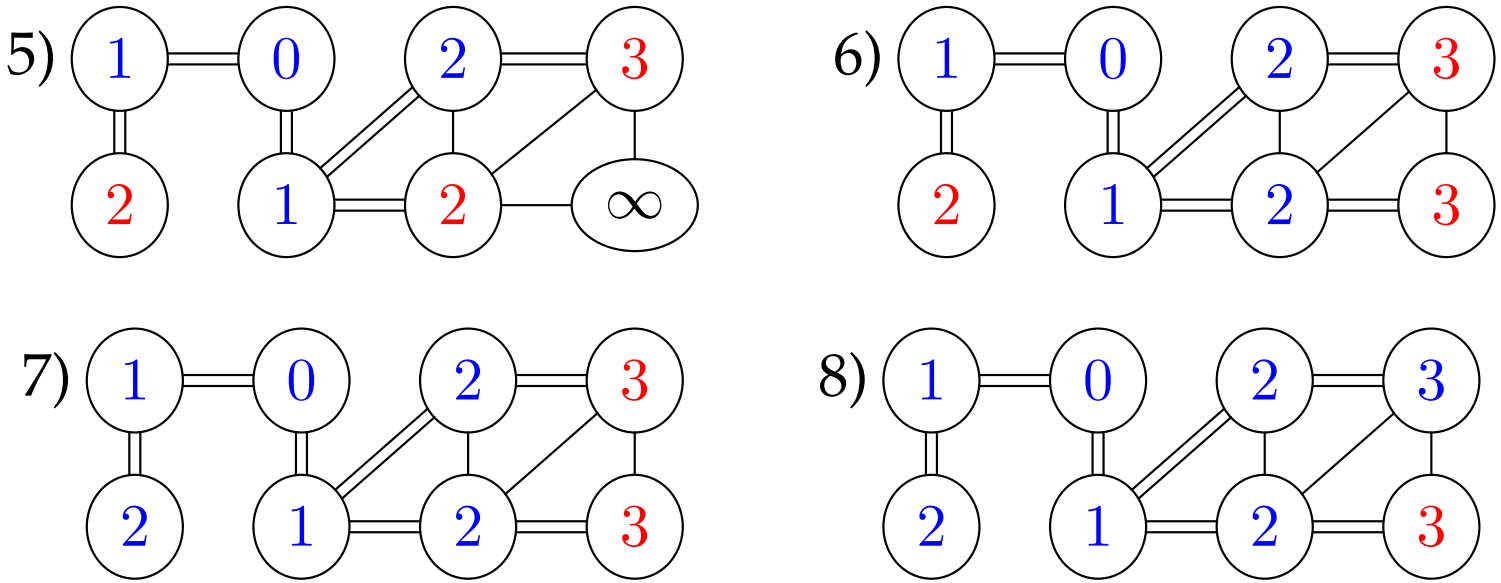






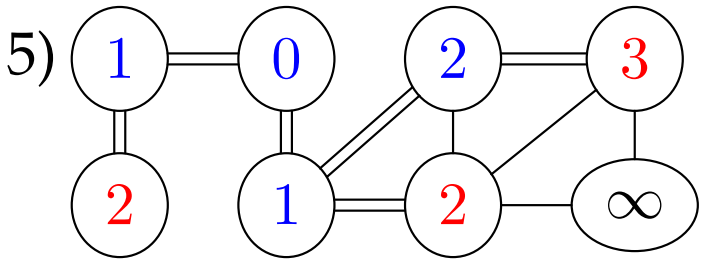
BFS EXAMPLE/2

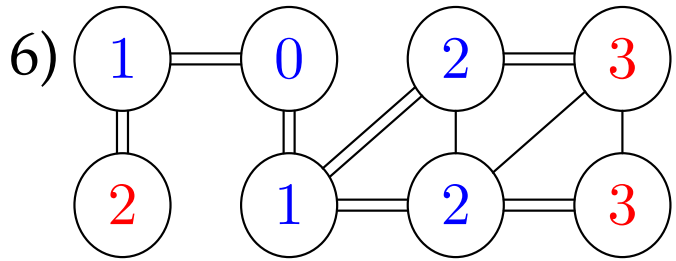
BFS example/2

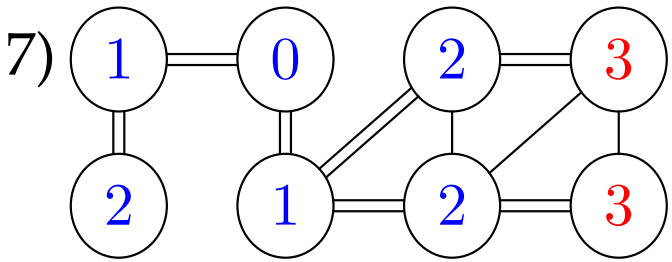


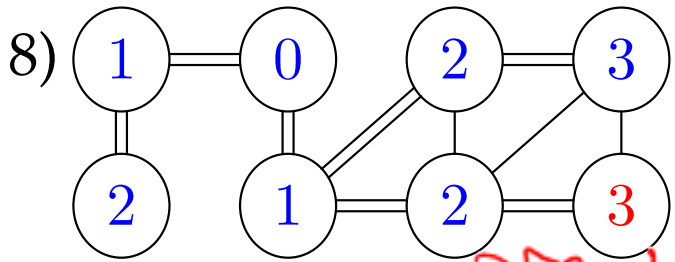
BFS EXAMPLE/2

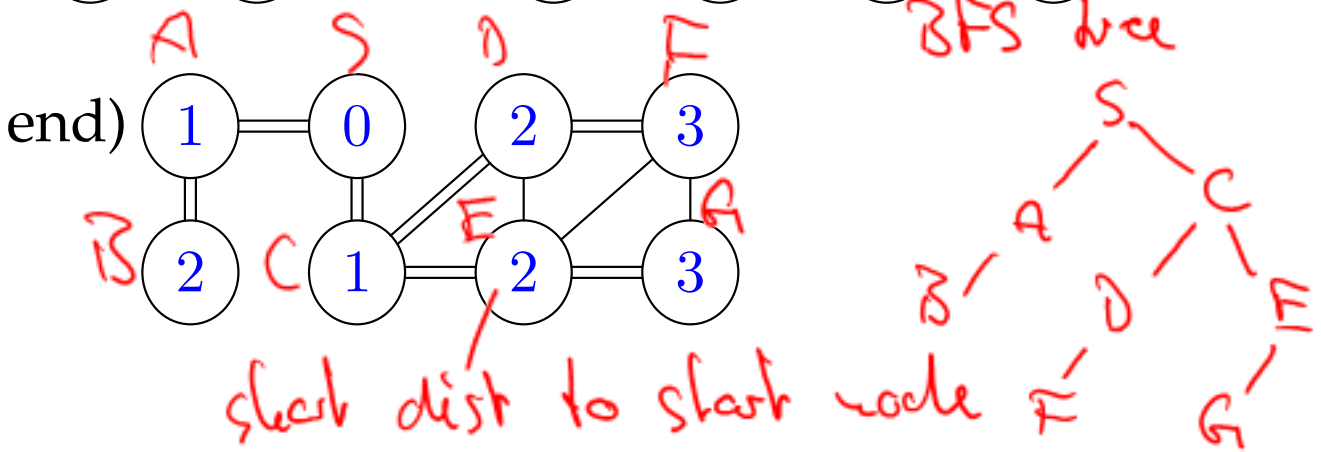
BFS example/2











EXERCISE SL09-4

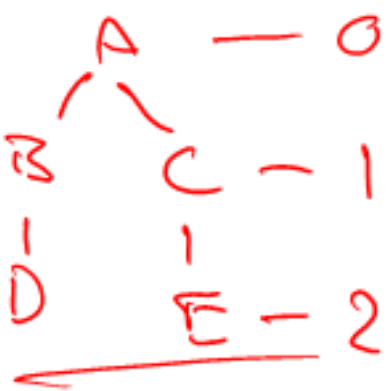
Does the ordering of adjacency lists affect

1. the distances from the starting node computed by BFS?
2. the BFS tree?

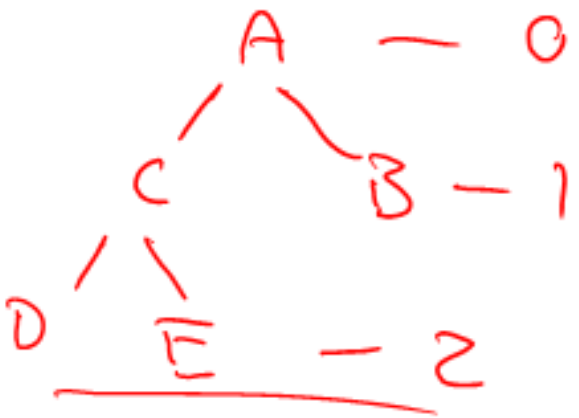
- dist: not affected (we always get shortest dist)
- BFS tree are different



BFS tree 1



BFS tree 2



BFS PROPERTIES/1

BFS running time

- Given a graph $G = (V, E)$
  - Initializing the vertices takes time $O(|V|)$
  - Vertices are enqueued if their color is white
    - Assuming that en- and dequeueing takes time $O(1)$, the total cost of this operation is $O(|V|)$
  - The adjacency list of a vertex is scanned when the vertex is dequeued
    - Since the sum of the lengths of all the adjacency lists is $O(|E|)$, the cost of scanning is $O(|E|)$

BFS PROPERTIES/1

BFS running time

- Given a graph $G = (V, E)$
  - Initializing the vertices takes time $O(|V|)$
  - Vertices are enqueued if their color is white
    - Assuming that en- and dequeueing takes time $O(1)$, the total cost of this operation is $O(|V|)$
  - The adjacency list of a vertex is scanned when the vertex is dequeued
    - Since the sum of the lengths of all the adjacency lists is $O(|E|)$, the cost of scanning is $O(|E|)$
- Total running time: $O(|V| + |E|)$ (linear in the size of the adjacency list representation of G)

BFS PROPERTIES/2

Assume a graph $G = (V, E)$:

BFS PROPERTIES/2

Assume a graph $G = (V, E)$:

- BFS discovers **all vertices reachable** from a source vertex s.

BFS PROPERTIES/2

Assume a graph $G = (V, E)$:

- BFS discovers **all vertices reachable** from a source vertex s.
- BFS computes the **shortest distance** to all reachable vertices.

BFS PROPERTIES/2

Assume a graph $G = (V, E)$:

- BFS discovers **all vertices reachable** from a source vertex s.
- BFS computes the **shortest distance** to all reachable vertices.
- BFS computes a **breadth-first tree** that contains every reachable vertex.

BFS PROPERTIES/2

Assume a graph $G = (V, E)$:

- BFS discovers **all vertices reachable** from a source vertex s.
- BFS computes the **shortest distance** to all reachable vertices.
- BFS computes a **breadth-first tree** that contains every reachable vertex. (possibly not all nodes are visited)
- For any vertex v reachable from s, the path from s to v in the breadth-first tree corresponds to a **shortest path** in G.



TABLE OF CONTENTS — SL09

1. Graphs

2. Breadth-first search (BFS)

3. Depth-first search (DFS)
   - DFS algorithm
   - DFS example
   - DFS properties

4. Topological sorting

DFS ALGORITHM/1

- A **depth-first search** (DFS) is like walking in a labyrinth with a string and following one path to the end:
  - Start at vertex v, tie the end of the string to v and paint v gray.
  - Set v as the current vertex and consider an arbitrary edge (v, u).
  - If the chosen edge leads to a non-white vertex u then return to v.
  - Otherwise, unroll the string, move to u, paint u gray, set u as current vertex, and repeat the previous steps.

DFS ALGORITHM/2

- Eventually, one will get to a point where all edges from u lead to non-white vertices.

DFS ALGORITHM/2

- Eventually, one will get to a point where all edges from u lead to non-white vertices.
- Then, one backtracks by rolling up the string until one gets back to a previously visited vertex v.

DFS ALGORITHM/2

- Eventually, one will get to a point where all edges from u lead to non-white vertices.
- Then, one backtracks by rolling up the string until one gets back to a previously visited vertex v.
- v thus becomes the new current vertex and one can repeat the previous steps.

DFS ALGORITHM/2

DFS annotations

- A DFS can be used to annotate vertices and edges with additional information:
  - start time $stime$ (when the vertex was visited first)
  - end time $etime$ (when the vertex was visited last)
  - edge classification (tree, forward, back, or cross edge)

DFS ALGORITHM/2

DFS annotations

- A DFS can be used to annotate vertices and edges with additional information:
  - start time $stime$ (when the vertex was visited first)
  - end time $etime$ (when the vertex was visited last)
  - edge classification (tree, forward, back, or cross edge)
- Such annotations reveal useful information about the graph that can be used by advanced algorithms.

DFS ALGORITHM/2

DFS annotations

- A DFS can be used to annotate vertices and edges with additional information:
  - start time $stime$ (when the vertex was visited first)
  - end time $etime$ (when the vertex was visited last)
  - edge classification (tree, forward, back, or cross edge)
- Such annotations reveal useful information about the graph that can be used by advanced algorithms.

DFS time stamping

- Vertex v is white before $v.stime$, gray between $v.stime$ and $v.etime$, and black after $v.etime$.

DFS ALGORITHM/2

DFS annotations

- A DFS can be used to annotate vertices and edges with additional information:
  - start time $stime$ (when the vertex was visited first)
  - end time $etime$ (when the vertex was visited last)
  - edge classification (tree, forward, back, or cross edge)
- Such annotations reveal useful information about the graph that can be used by advanced algorithms.

DFS time stamping

- Vertex v is white before $v.stime$, gray between $v.stime$ and $v.etime$, and black after $v.etime$.
- Gray vertices form a linear chain and the traversal of the nodes works like a stack (LIFO).

DFS ALGORITHM/3

corresponds to BFS (searches all reachable nodes; search differs)

Algo: DFS-Tree(G,s)

```
s.col = G; time = time+1; s.time = time;
foreach u ∈ s.adj do
    if u.col==W then
        u.pred = s;
        DFS-Tree(G,u)
s.col = B; time = time+1; s.etime = time;
```

stack (LIFO queue) (use recursion to maintain the stack)

process the new node first (longest path)

DFS ALGORITHM/4

process the entire graph

Algo: DFS(G)

foreach $v \in G.V$ **do**
  v.col = W; v.pred = NIL

initialization (all nodes are white)

time = 0;

foreach $v \in G.V$ **do**
  if *v.col*==*W* **then** DFS-Tree(G,v);

look at all nodes

DFS ALGORITHM/4

Algo: DFS(G)

---

foreach $v \in G.V$ **do**
  v.col = W; v.pred = NIL

time = 0;
foreach $v \in G.V$ **do**
  if *v.col*==*W* **then** DFS-Tree(G,v);

Some observations

- DFS guarantees that every vertex is visited

DFS ALGORITHM/4

Algo: DFS(G)

---

foreach $v \in G.V$ **do**
  v.col = W; v.pred = NIL

time = 0;
foreach $v \in G.V$ **do**
  if *v.col*==*W* **then** DFS-Tree(G,v);

Some observations

- DFS guarantees that every vertex is visited
- Each call of `DFS-Tree` adds a new tree to depth-first forest

DFS ALGORITHM/4

Algo: DFS(G)

---

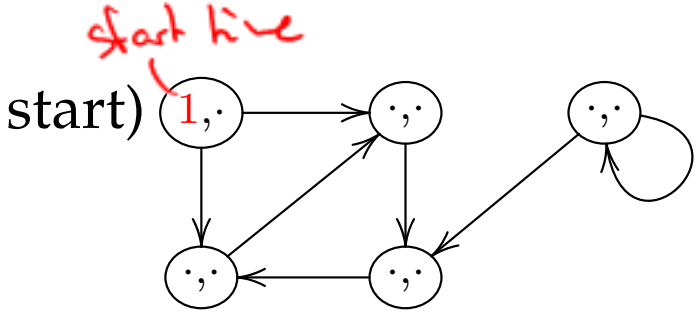
foreach $v \in G.V$ **do**
  v.col = W; v.pred = NIL
time = 0;
foreach $v \in G.V$ **do**
  if *v.col*==*W* **then** DFS-Tree(G,v);

Some observations

- DFS guarantees that every vertex is visited
- Each call of `DFS-Tree` adds a new tree to depth-first forest
- After termination, every vertex has a start and end time

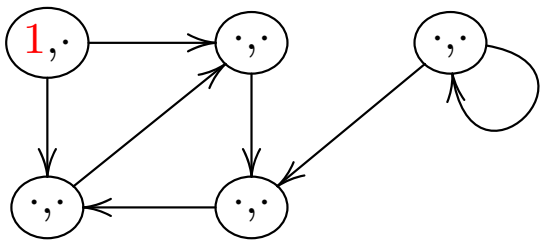
DFS EXAMPLE/1

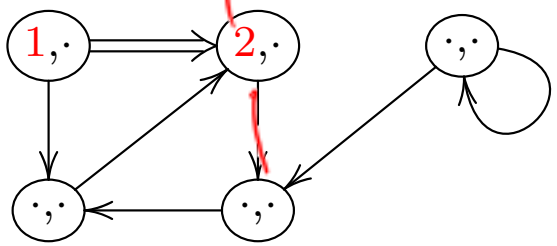
DFS search with start and end times:



DFS EXAMPLE/1

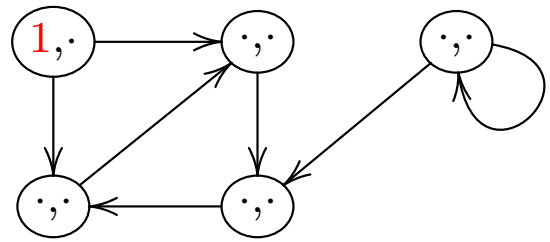
DFS search with start and end times:

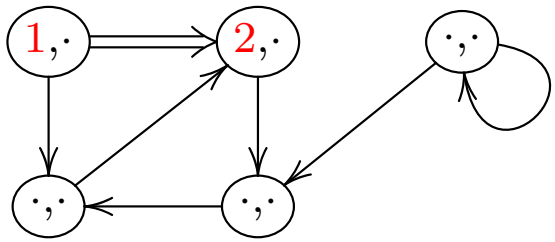
start) 

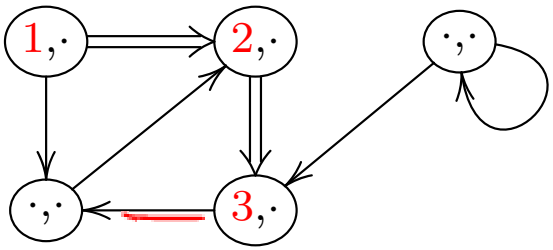
1) 

DFS EXAMPLE/1

DFS search with start and end times:

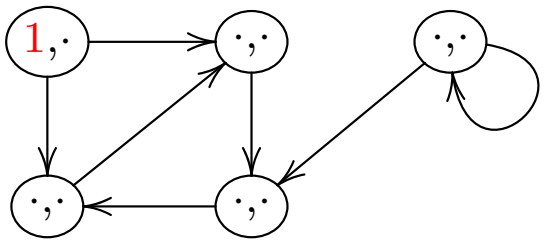
start)


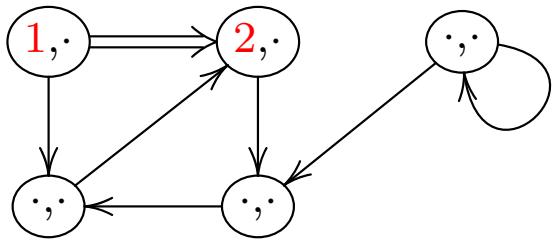
1)


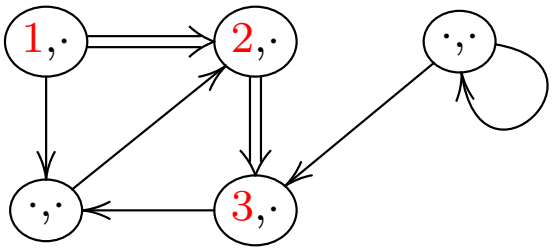
2)


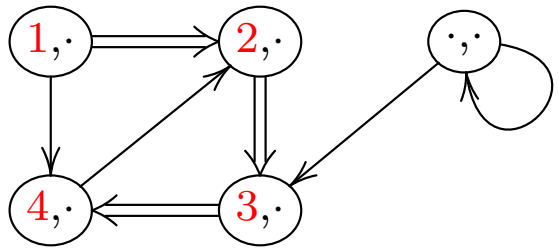
DFS EXAMPLE/1

DFS search with start and end times:

start)


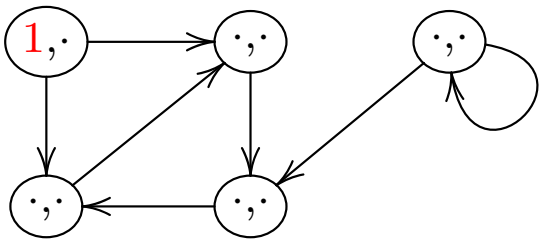
1)


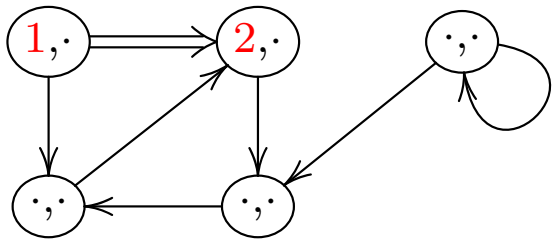
2)


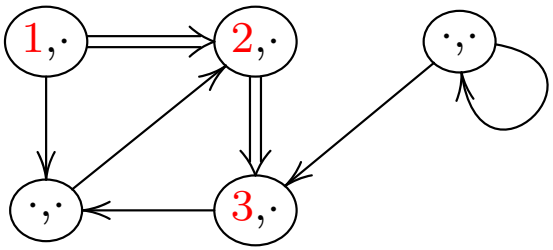
3)


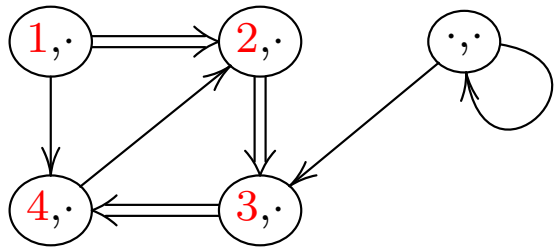
DFS EXAMPLE/1

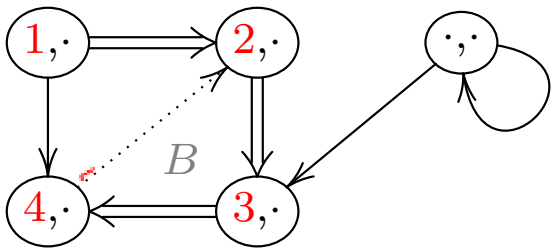
DFS search with start and end times:

start)


1)


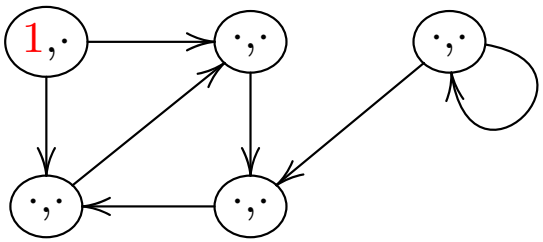
2)


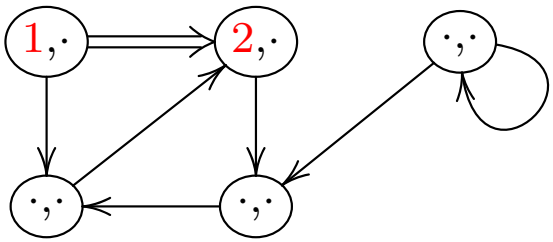
3)


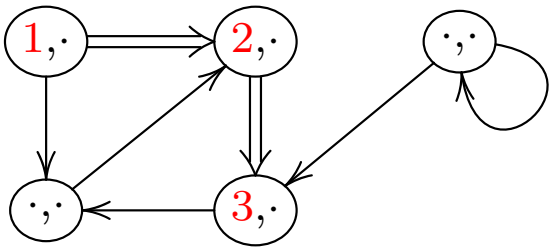
4)


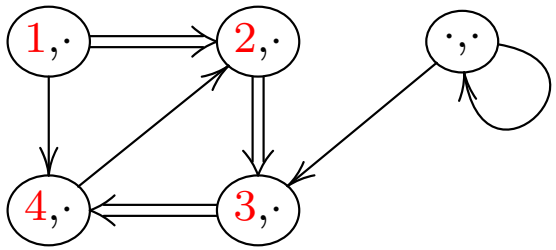
DFS EXAMPLE/1

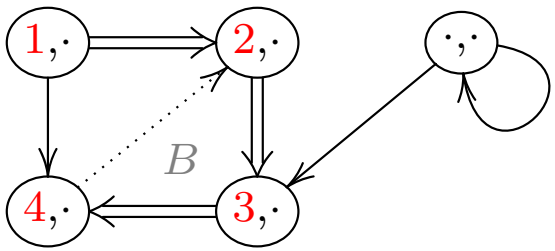
DFS search with start and end times:

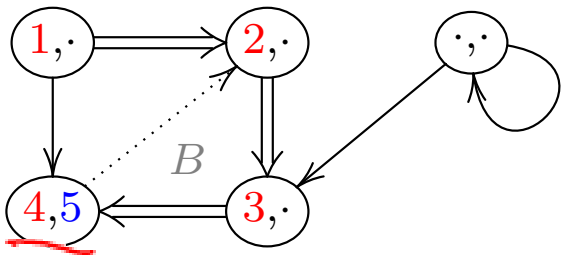
start)


1)


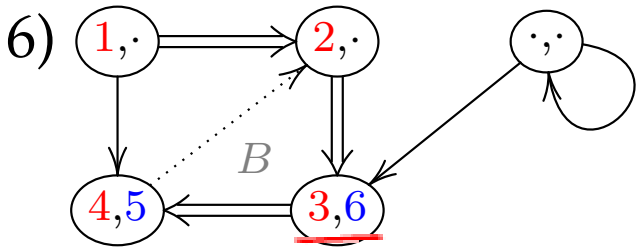
2)


3)


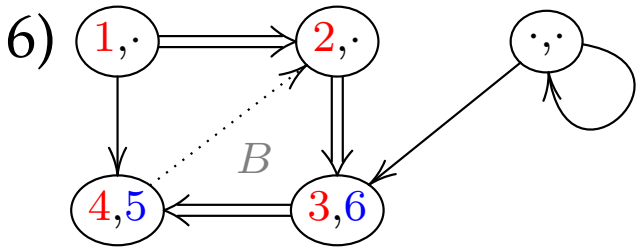
4)


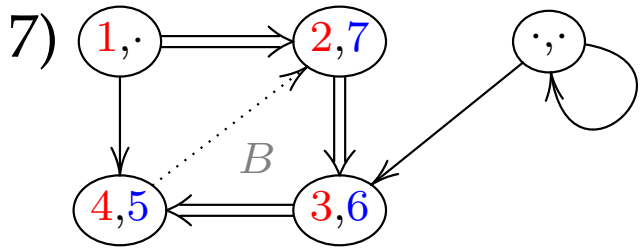
5)


DFS EXAMPLE/2



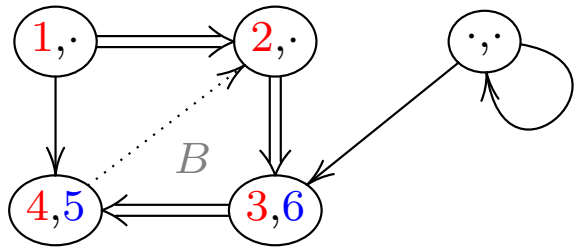
DFS EXAMPLE/2



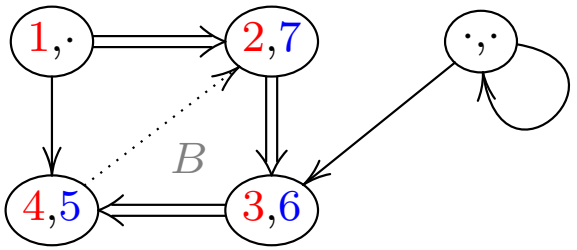


DFS EXAMPLE/2

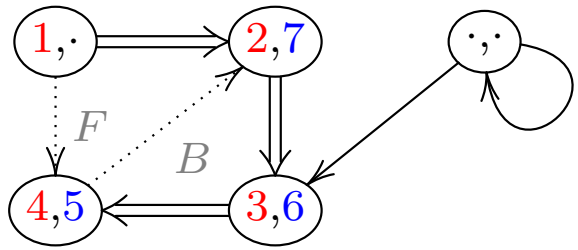
6)



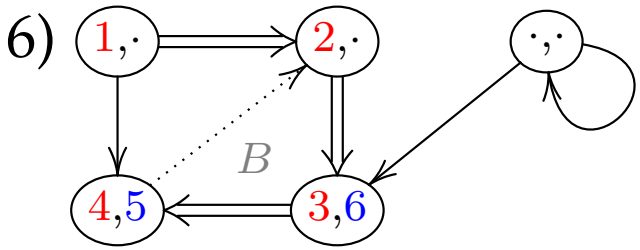
7)

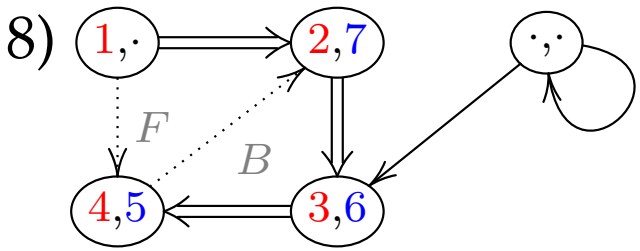


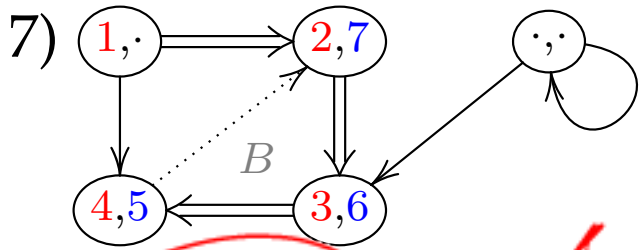
8)

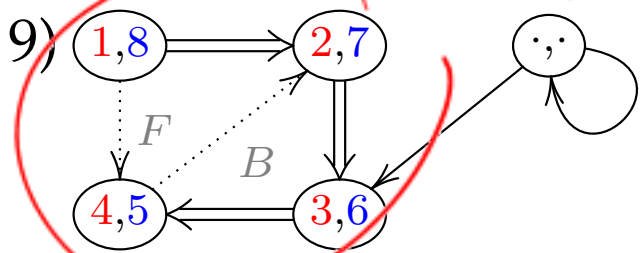


DFS EXAMPLE/2



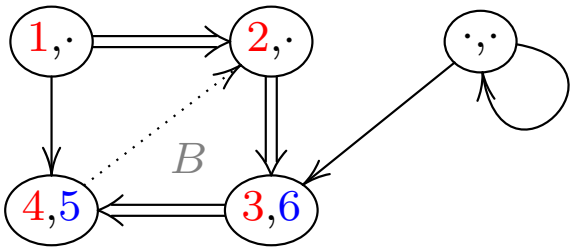


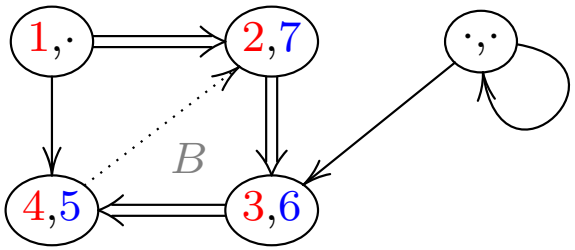


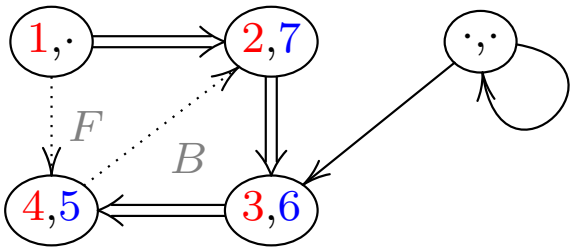


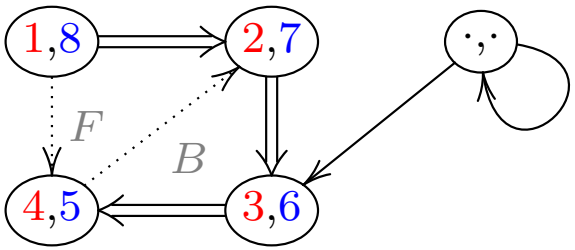
all reachable nodes

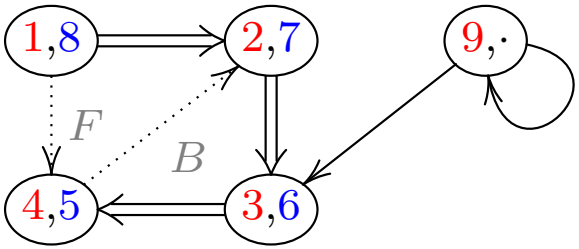
DFS EXAMPLE/2

6)


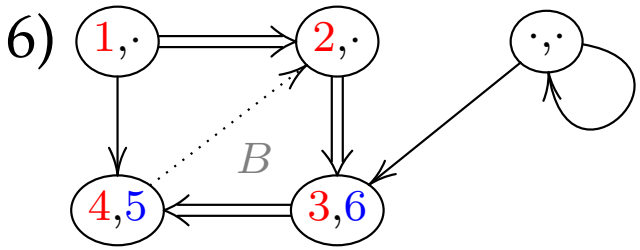
7)


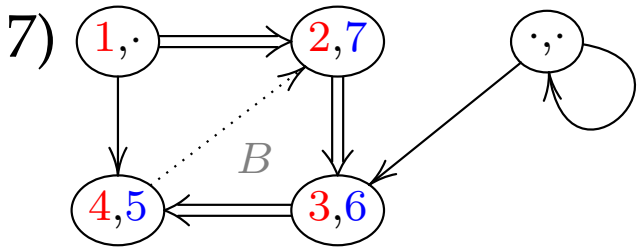
8)


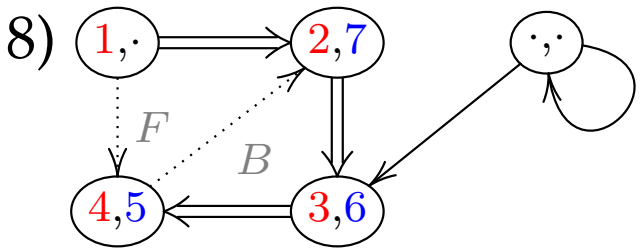
9)


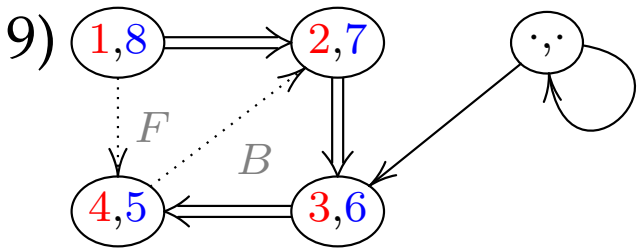
10)


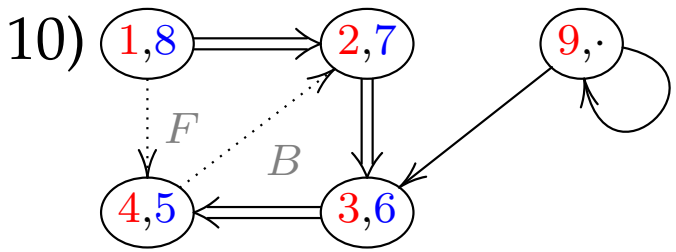
DFS EXAMPLE/2

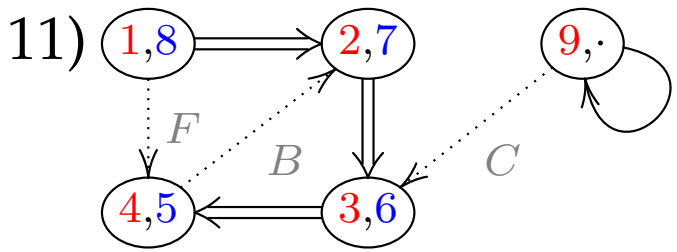




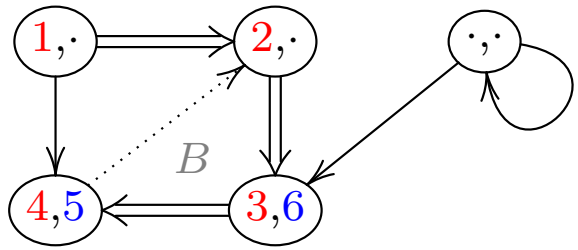


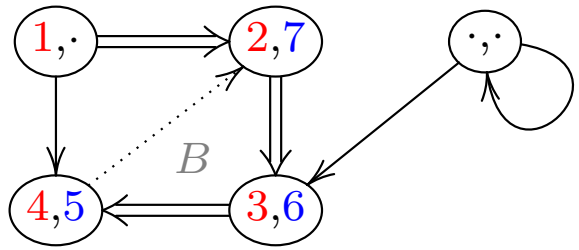


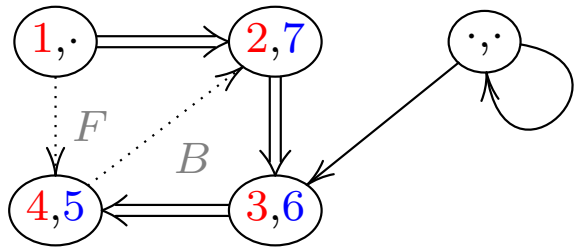


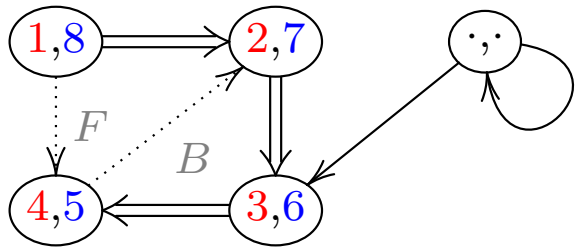


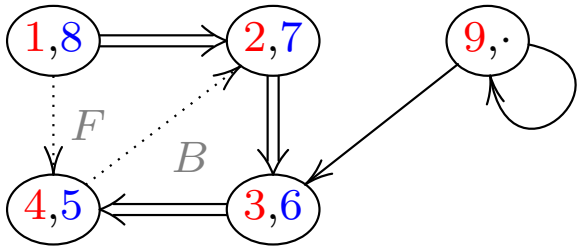
DFS EXAMPLE/2

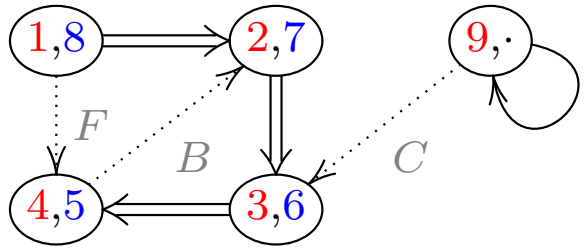
6)


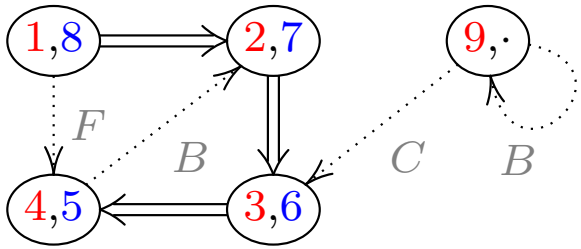
7)


8)


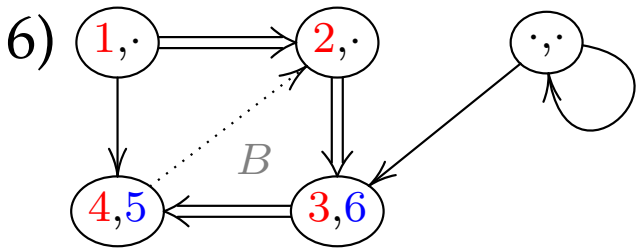
9)


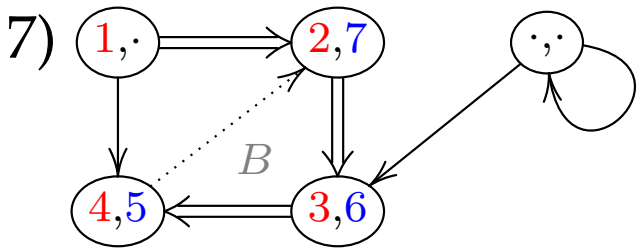
10)


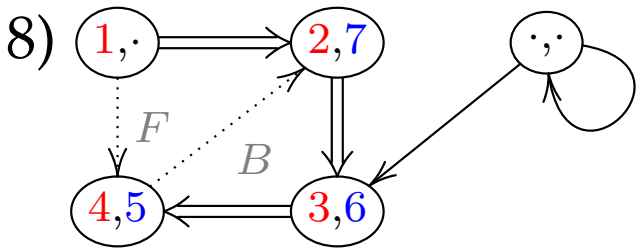
11)


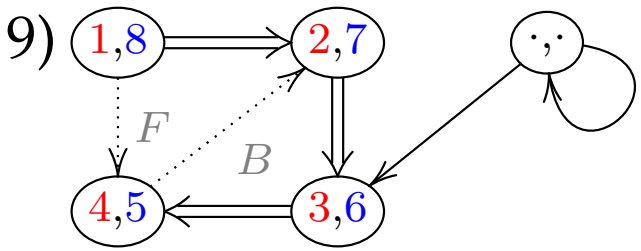
12)


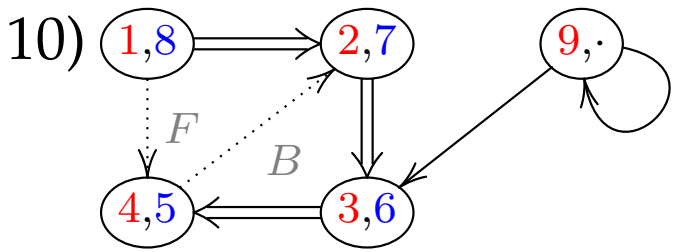
DFS EXAMPLE/2

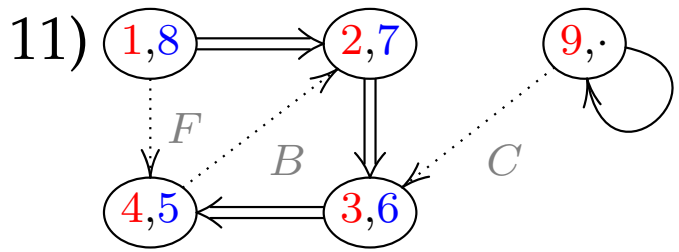


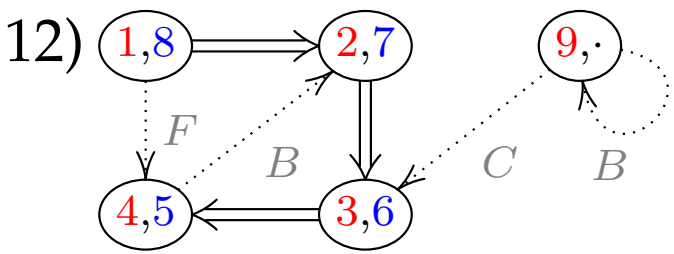


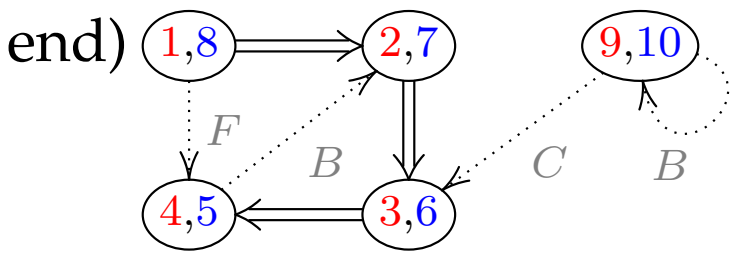












DFS PROPERTIES/1

DFS running time

- Given a graph $G = (V, E)$
  - Each loop in `DFS` takes time $O(|V|)$, excluding the time to execute `DFS-Tree`
  - `DFS-Tree` is called once for every vertex
    - It is only invoked on white vertices
    - It paints the vertex gray immediately
  - Each time, `DFS-Tree` explores an adjacency list
  - Hence, the overall cost of `DFS-Tree` is $O(|E|)$

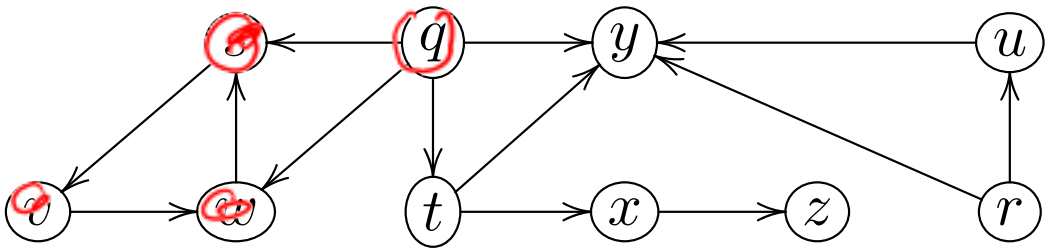
DFS PROPERTIES/1

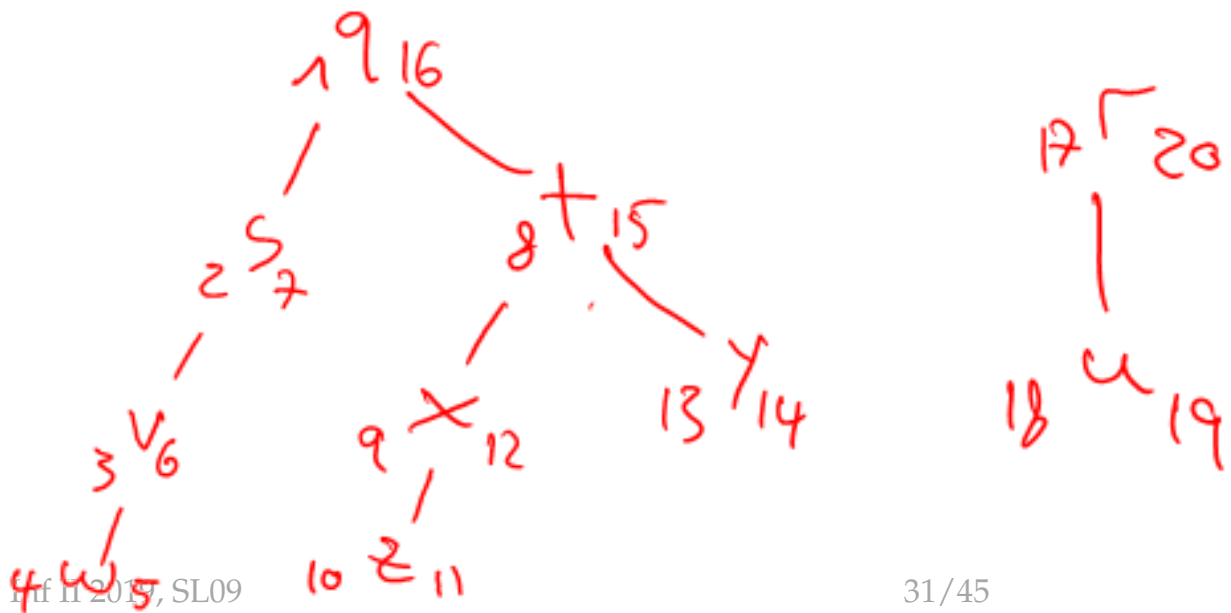
DFS running time

- Given a graph $G = (V, E)$
  - Each loop in `DFS` takes time $O(|V|)$, excluding the time to execute `DFS-Tree`
  - `DFS-Tree` is called once for every vertex
    - It is only invoked on white vertices
    - It paints the vertex gray immediately
  - Each time, `DFS-Tree` explores an adjacency list
  - Hence, the overall cost of `DFS-Tree` is $O(|E|)$
- Total running time: $O(|V| + |E|)$ (linear in the size of the adjacency list representation of G)

EXERCISE SL09-5

Illustrate DFS on the following graph (start at q). Vertices are considered alphabetically. Adjacency lists are ordered alphabetically. Show the times computed by DFS. In which sequence are the nodes visited?





DFS PROPERTIES/2

DFS versus BFS/1

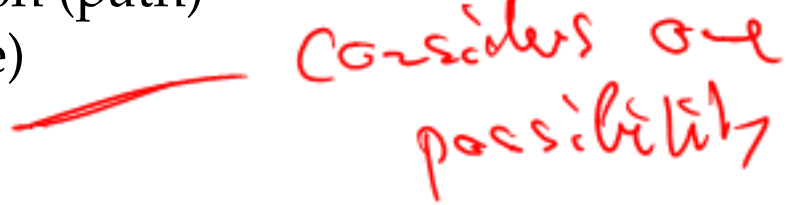
- BFS visits **all vertices that are reachable** from s
  - returns a single search tree
  - might not visit all nodes
  - considers all directions (paths)
  - finds optimal (shortest) path
  - expensive (chess)
  - is used selectively (on small subproblems)
  - $O(V + E)$

consider all possibilities (less)

DFS PROPERTIES/2

DFS versus BFS/1

- BFS visits **all vertices that are reachable** from s
  - returns a single search tree
  - might not visit all nodes
  - considers all directions (paths)
  - finds optimal (shortest) path
  - expensive (chess)
  - is used selectively (on small subproblems)
  - $O(V + E)$
- DFS visits **all vertices** in the given graph G
  - may return several search trees
  - visits all nodes
  - pursues one direction (path)
  - can get "lost" (maze)
  - cheap and practical
  - $O(V + E)$



DFS PROPERTIES/3

DFS versus BFS/2

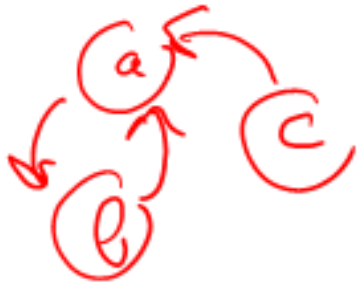
- The difference (visiting all nodes versus visiting reachable nodes) comes from the applications that make use of BFS and DFS

DFS PROPERTIES/3

DFS versus BFS/2

- The difference (visiting all nodes versus visiting reachable nodes) comes from the applications that make use of BFS and DFS
- Note: the behavior of the graph search algorithms (i.e., to search the entire graph or only part of it) could easily be changed

DFS PROPERTIES/4



start of a: (a(bb)a)(cc)

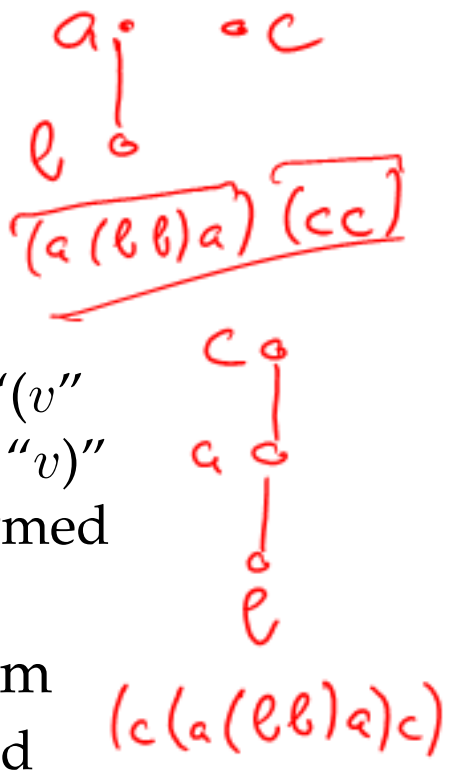
start at c: (c(a(bb)a)c)

Parenthesis theorem

- Start and end times have parenthesis structure
  - Represent start time of v with left parenthesis "(v"
  - Represent end time of v with right parenthesis "v)"
  - History of start and end times yields a well-formed expression (i.e. parethesis are properly nested)



DFS PROPERTIES/4



Parenthesis theorem

- Start and end times have parenthesis structure
  - Represent start time of v with left parenthesis "(v"
  - Represent end time of v with right parenthesis "v)"
  - History of start and end times yields a well-formed expression (i.e. parethesis are properly nested)
- Intuition for the proof of the parenthesis theorem
  - Any two intervals are either disjoint or enclosed
  - Overlapping intervals would correspond to one of the following situations
    - Finishing the ancestor before finishing the descendant
    - Starting the descendant before starting the ancestor

DFS PROPERTIES/5

how graph is traversed

Edge classification/1

- Tree edge (gray node to white node)
  - Edges in depth-first (DF) forest

DFS PROPERTIES/5

Edge classification/1

- Tree edge (gray node to white node)
  - Edges in depth-first (DF) forest
- Back edge (gray node to gray node)
  - Edges from descendant to ancestor in DF tree
  - Self-loops are a special case of back edges

cycles

important

DFS PROPERTIES/5

Edge classification/1

- Tree edge (gray node to white node)
  - Edges in depth-first (DF) forest
- Back edge (gray node to gray node)
  - Edges from descendant to ancestor in DF tree
  - Self-loops are a special case of back edges
- Forward edge (gray node to black node)
  - Non-tree edges from ancestor to descendant in DF tree

DFS PROPERTIES/5

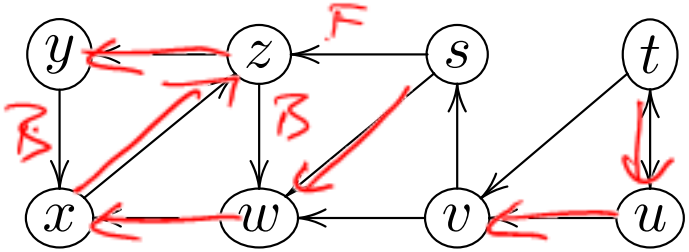
Edge classification/1

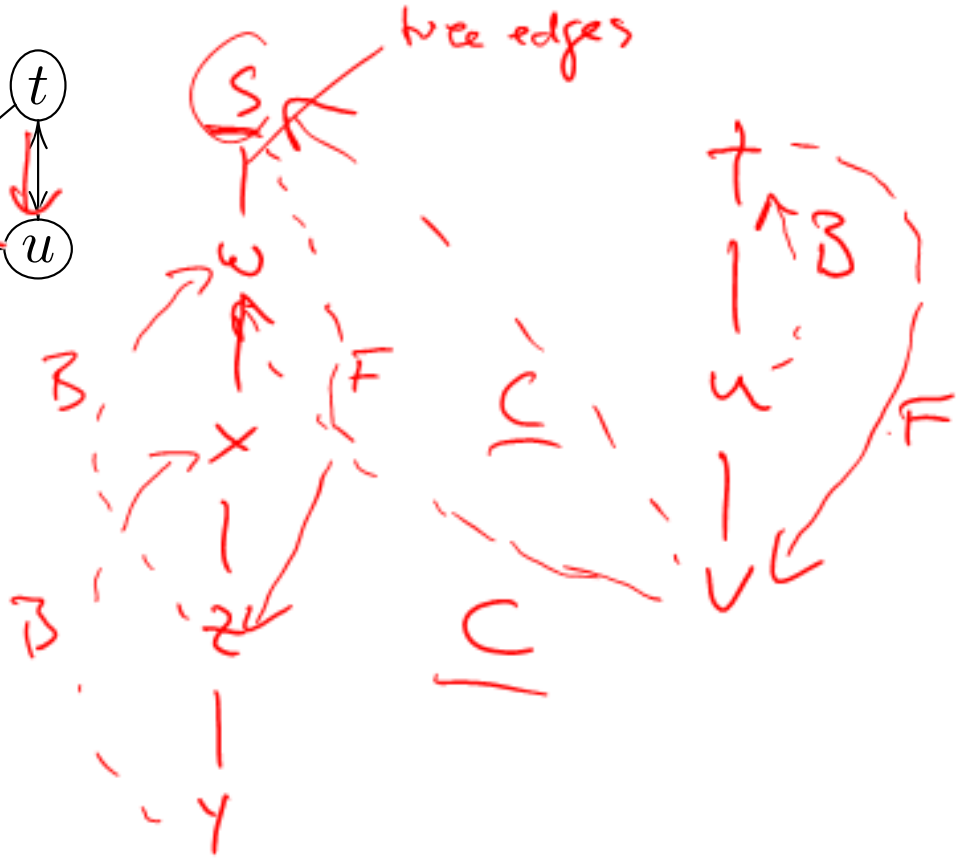
- Tree edge (gray node to white node)
  - Edges in depth-first (DF) forest
- Back edge (gray node to gray node)
  - Edges from descendant to ancestor in DF tree
  - Self-loops are a special case of back edges
- Forward edge (gray node to black node)
  - Non-tree edges from ancestor to descendant in DF tree
- Cross edge (gray node to black node)
  - Remaining edges of graph, i.e., edges between trees or subtrees

less important

EXERCISE SL09-6

Draw the DFS forest with edge types. Choose an alphabetical ordering when you can choose between nodes.





DFS PROPERTIES/6

Edge classification/2

- In DFS, the color of the vertex decides the type of the edge

DFS PROPERTIES/6

Edge classification/2

- In DFS, the color of the vertex decides the type of the edge
- It is thus impossible to distinguish forward and cross edges

DFS PROPERTIES/6

Edge classification/2

- In DFS, the color of the vertex decides the type of the edge
- It is thus impossible to distinguish forward and cross edges
- Most algorithms do not distinguish between these two types

DFS PROPERTIES/6

Edge classification/2

- In DFS, the color of the vertex decides the type of the edge
- It is thus impossible to distinguish forward and cross edges
- Most algorithms do not distinguish between these two types
- Tree and back edges are important (e.g., to detect loops)

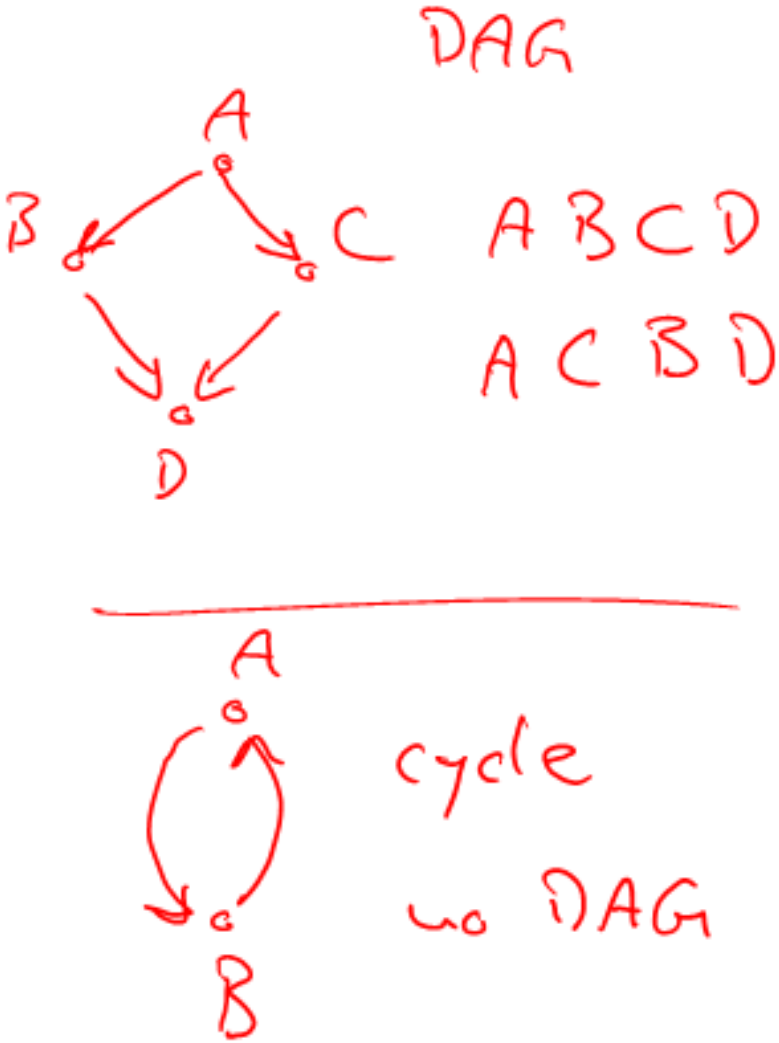
TABLE OF CONTENTS — SL09

1. Graphs

2. Breadth-first search (BFS)

3. Depth-first search (DFS)

4. Topological sorting
   Directed acyclic graph (DAG)
   Topological sorting



DAG/1

Definition of DAG

- A **directed acyclic graph** (DAG) is a directed graph without cycles

DAG/1

Definition of DAG

- A **directed acyclic graph** (DAG) is a directed graph without cycles
- DAGs are used to indicate precedence among events (x must happen before y)
  - An example would be a parallel execution of code

DAG/1

Definition of DAG

- A **directed acyclic graph** (DAG) is a directed graph without cycles
- DAGs are used to indicate precedence among events (x must happen before y)
  - An example would be a parallel execution of code
- One gets a total order using topological sorting

DAG/1

Definition of DAG

- A **directed acyclic graph** (DAG) is a directed graph without cycles
- DAGs are used to indicate precedence among events (x must happen before y)
  - An example would be a parallel execution of code
- One gets a total order using topological sorting

DAG theorem

- A directed graph G is acyclic iff DFS of G has no back edges

DAG/1

Definition of DAG

- A **directed acyclic graph** (DAG) is a directed graph without cycles
- DAGs are used to indicate precedence among events (x must happen before y)
  - An example would be a parallel execution of code
- One gets a total order using topological sorting

DAG theorem

- A directed graph G is acyclic iff DFS of G has no back edges
- Thus, DFS can be used to check if a graph is a DAG

DAG/2



Proof of theorem

- Suppose there is a back edge (u, v)
  - Then v is an ancestor of u in depth-first forest
  - Hence, there is a path from v to u in the graph
  - Such a path together with (u, v) yields a cycle

DAG/2

Proof of theorem

- Suppose there is a back edge (u, v)
  - Then v is an ancestor of u in depth-first forest
  - Hence, there is a path from v to u in the graph
  - Such a path together with (u, v) yields a cycle
- Suppose there is a cycle c in G
  - Let v be the first vertex in c to be discovered
  - Let u be a predecessor of vertex v in cycle c
  - Upon discovering v the path from v to u is white
  - Visiting this path makes u become a descendant of v
  - It follows that edge (u, v) is a back edge



EXERCISE SL09-7

use DFS to search the entire graph; stop and return FALSE if we find a back edge

Give an algorithm that checks if a graph G is a DAG.

```
DAGcheck(G,v)
  v.col = gray
  for u ∈ v.adj do
    if u.col = white then
      if !DAGcheck(G,u) then
        return FALSE
    else if u.col = gray then
      return FALSE
  v.col = black
  return TRUE
```

```
DAGgraph(G)
  for v∈G.V do v.col=white
  for v∈G.V do
    if v.Col == white then
      if !DAGcheck(G,v)
        then return
          FALSE
  return TRUE
```

call: isDAG = DAGgraph(G)

Topological sorting/1

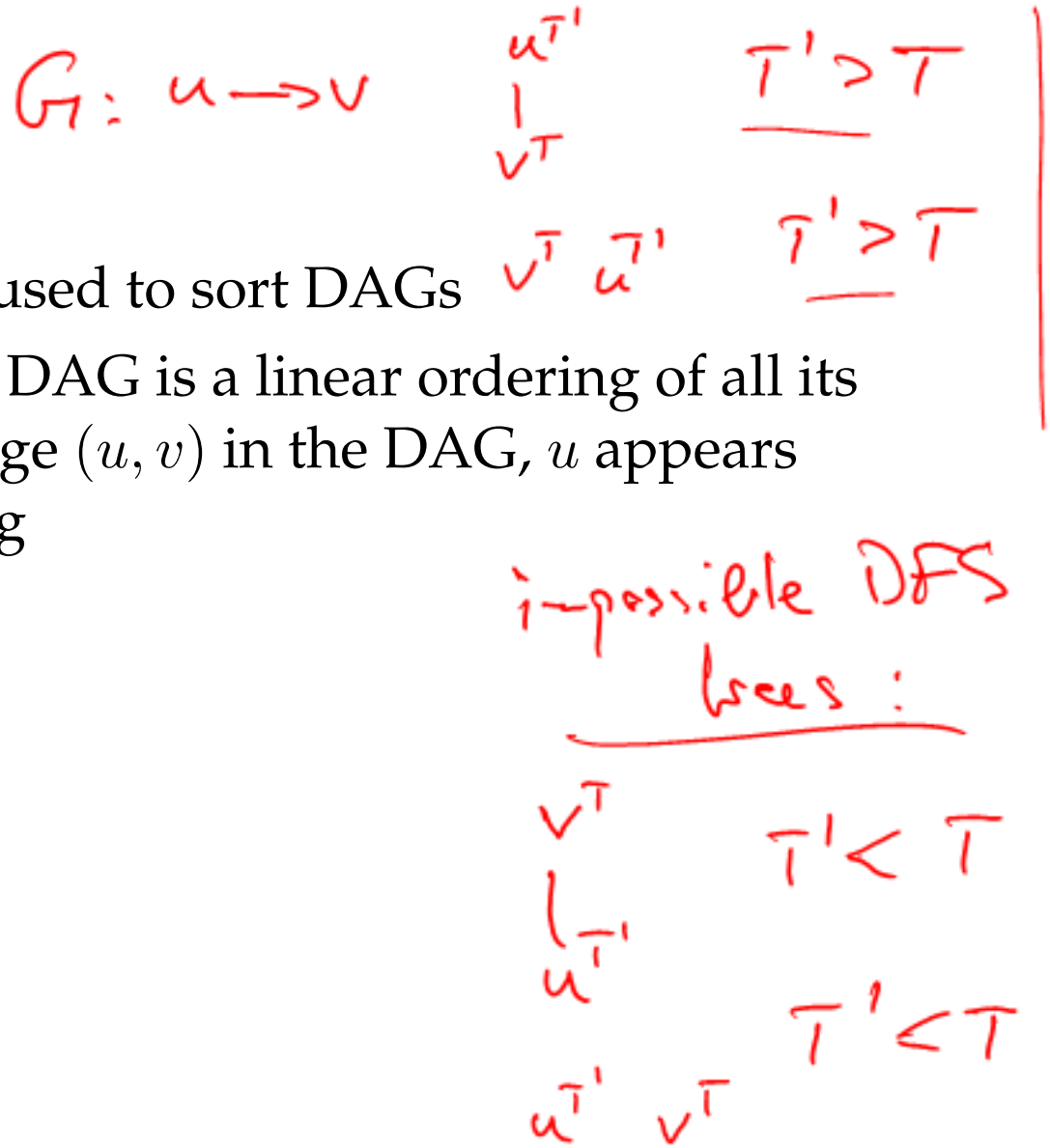
Sorting of DAGs

- Topological sorting is used to sort DAGs

TOPOLOGICAL SORTING/1

Sorting of DAGs

- Topological sorting is used to sort DAGs
- A topological sort of a DAG is a linear ordering of all its vertices s.t., for any edge (u, v) in the DAG, u appears before v in the ordering



TOPOLOGICAL SORTING/1

Sorting of DAGs

- Topological sorting is used to sort DAGs
- A topological sort of a DAG is a linear ordering of all its vertices s.t., for any edge (u, v) in the DAG, u appears before v in the ordering

Algo: TopologicalSort(G)

Call DFS(G) to compute v.etime for each v;
As each vertex v is finished:
    insert v at the beginning of a linked list;
Return the linked list of vertices

TOPOLOGICAL SORTING/2

Correctness

- DAG and $(u, v) \in E$ imply $u.etime > v.etime$

Topological sorting/2

Correctness

- DAG and $(u, v) \in E$ imply $u.etime > v.etime$
- When the edge (u, v) is explored, u is gray

TOPOLOGICAL SORTING/2

Correctness

- DAG and $(u, v) \in E$ imply $u.etime > v.etime$
- When the edge (u, v) is explored, u is gray
- So, one can distinguish the following cases
  - $v.col = G$: (u, v) is a back edge (cycle, contradiction)
  - $v.col = B$: v is already finished, so $v.etime < u.etime$
  - $v.col = W$: v becomes a descendant of u, so v will be finished before u and $v.etime < u.etime$

TOPOLOGICAL SORTING/2

Correctness

- DAG and $(u, v) \in E$ imply $u.etime > v.etime$
- When the edge (u, v) is explored, u is gray
- So, one can distinguish the following cases
  - $v.col = G$: (u, v) is a back edge (cycle, contradiction)
  - $v.col = B$: v is already finished, so $v.etime < u.etime$
  - $v.col = W$: v becomes a descendant of u, so v will be finished before u and $v.etime < u.etime$
- The definition of topological sort is satisfied

TOPOLOGICAL SORTING/2

Correctness

- DAG and $(u, v) \in E$ imply $u.etime > v.etime$
- When the edge (u, v) is explored, u is gray
- So, one can distinguish the following cases
  - $v.col = G$: (u, v) is a back edge (cycle, contradiction)
  - $v.col = B$: v is already finished, so $v.etime < u.etime$
  - $v.col = W$: v becomes a descendant of u, so v will be finished before u and $v.etime < u.etime$
- The definition of topological sort is satisfied

Running time

- Total running time of topological sort is $O(|V| + |E|)$

TOPOLOGICAL SORTING/2

Correctness

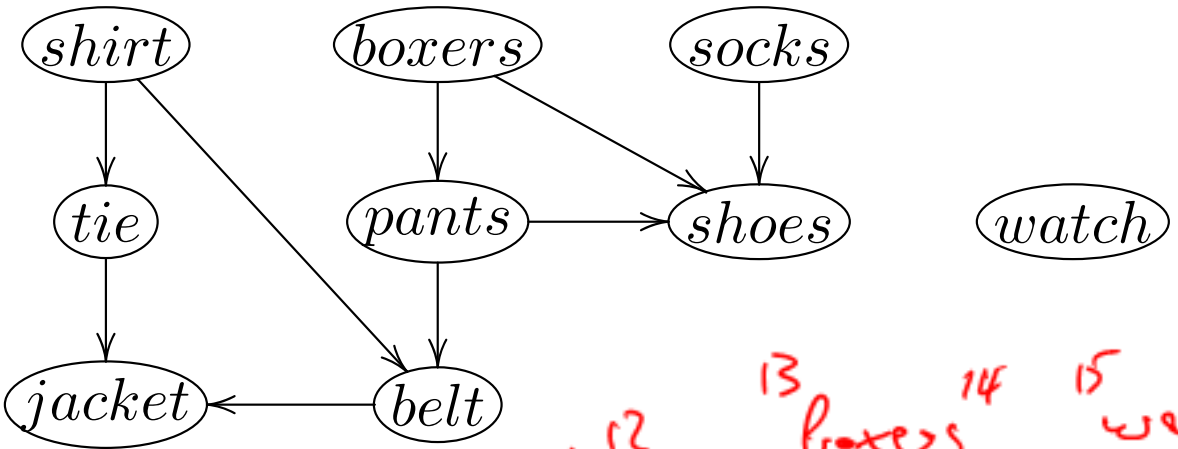
- DAG and $(u, v) \in E$ imply $u.etime > v.etime$
- When the edge (u, v) is explored, u is gray
- So, one can distinguish the following cases
  - $v.col = G$: (u, v) is a back edge (cycle, contradiction)
  - $v.col = B$: v is already finished, so $v.etime < u.etime$
  - $v.col = W$: v becomes a descendant of u, so v will be finished before u and $v.etime < u.etime$
- The definition of topological sort is satisfied

Running time

- Total running time of topological sort is $O(|V| + |E|)$
- Running time of DFS plus $|V|$ insertions into linked list

EXERCISE SL09-8

Run topological sort on the following directed acyclic graph.



1 shirt 8
2 tie 5
3 jacket 4
6 belt 7
9 pants 12
10 shoes 11
13 boxers 14
15 watch 16
17 socks 18

socks watch boxers→pants→shoes shirt belt tie→jacket

SUMMARY

- Graph algorithms
  - Definition (simple path, cycle, subgraph, connected, directed, etc)
  - Status fields of nodes (with possible additional fields)
- Breadth-first search
  - BFS algorithms and properties
  - Application of BFS (e.g., Erdös number, shortest path)
- Depth-first search
  - DFS algorithm and properties
  - Application of DFS (e.g., cycle detection)
- Topological sort: DAG, sorting

Informatik II

Minimum Spanning Trees
Single Source Shorted Paths
— SL10 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

TABLE OF CONTENTS — SL10

1. Weighted graphs

2. Minimum spanning tree (MST)

3. Single-source shortest path (SSSP)

WEIGHTED GRAPHS

- Weighted graph
  - Graph $G = (V, E)$ where each edge $(u, v) \in E$ has a weight $w(u, v)$ specifying its cost (or length)
  - In other words, G is endowed with a weight function $w\colon E \to \mathbb{R}$ that assigns real numbers to its edges
  - Note: weighted graphs can be directed or undirected

WEIGHTED GRAPHS

- Weighted graph
  - Graph $G = (V, E)$ where each edge $(u, v) \in E$ has a weight $w(u, v)$ specifying its cost (or length)
  - In other words, G is endowed with a weight function $w\colon E \to \mathbb{R}$ that assigns real numbers to its edges
  - Note: weighted graphs can be directed or undirected
- Weighted algorithms
  - Minimum spanning tree
    - Apply to undirected graphs
    - Prim-Jarnik
  - Single-source shortest path
    - Apply to directed graphs
    - Bellman-Ford, Dijkstra

TABLE OF CONTENTS — SL10

1. Weighted graphs

2. Minimum spanning tree (MST)
   Prim-Jarnik algorithm
   Prim-Jarnik example

3. Single-source shortest path (SSSP)

MINIMUM SPANNING TREE/1

Spanning trees

- A spanning tree T of G is a subgraph that contains all the vertices of G and that is a tree (thus, acyclic)

MINIMUM SPANNING TREE/1

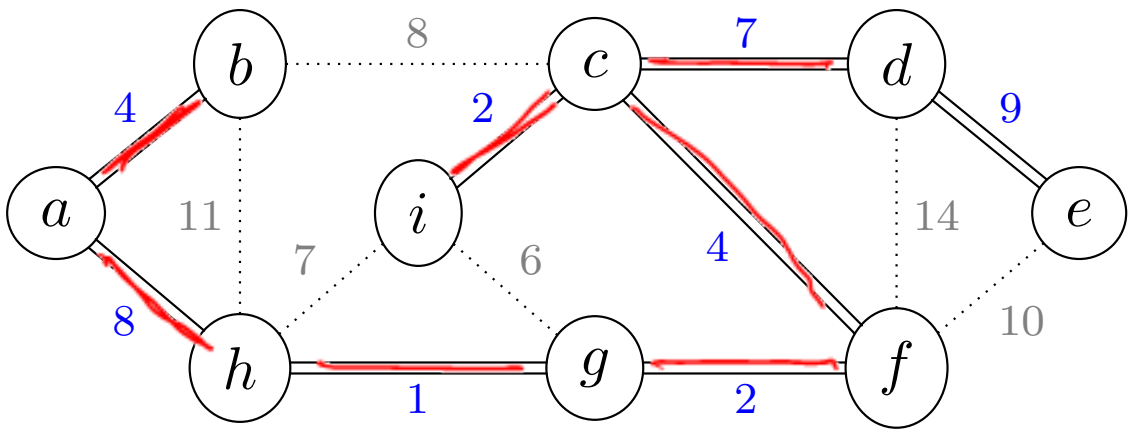
Spanning trees

- A spanning tree T of G is a subgraph that contains all the vertices of G and that is a tree (thus, acyclic)
- A minimum spanning tree of (G, w) is a spanning tree T of G that minimizes $w(T) = \sum_{(u,v)\in T} w(u, v)$

MINIMUM SPANNING TREE/1

Spanning trees

- A spanning tree T of G is a subgraph that contains all the vertices of G and that is a tree (thus, acyclic)
- A minimum spanning tree of (G, w) is a spanning tree T of G that minimizes $w(T) = \sum_{(u,v)\in T} w(u, v)$



of edges in a MST = |V| − 1

MINIMUM SPANNING TREE/2

Generic algorithm

- Have to
  make $|V|-1$ choices (edges of the MST) to arrive at solution

MINIMUM SPANNING TREE/2

Generic algorithm

- Have to
  make $|V|-1$ choices (edges of the MST) to arrive at solution

```
A = ∅;
while A does not form a spanning tree do
    Find edge e that is safe for A;
    A = A ∪ {e}
return A;
```

MINIMUM SPANNING TREE/2

Generic algorithm

- Have to
  make $|V|-1$ choices (edges of the MST) to arrive at solution

```
A = ∅;
while A does not form a spanning tree do
    Find edge e that is safe for A;
    A = A ∪ {e}
return A;
```

- After each choice, get a subproblem that is one vertex smaller than the original problem
  - At each step, want to determine cheaply a **save edge** (i.e., that definitely belongs to an MST)
  - Greedy choice property: locally optimal (i.e., greedy) choice yields a globally optimal solution

MINIMUM SPANNING TREE/3

Greedy choice/1

- Let $G = (V, E)$ and $S \subseteq V$ be a **cut** of G

MINIMUM SPANNING TREE/3

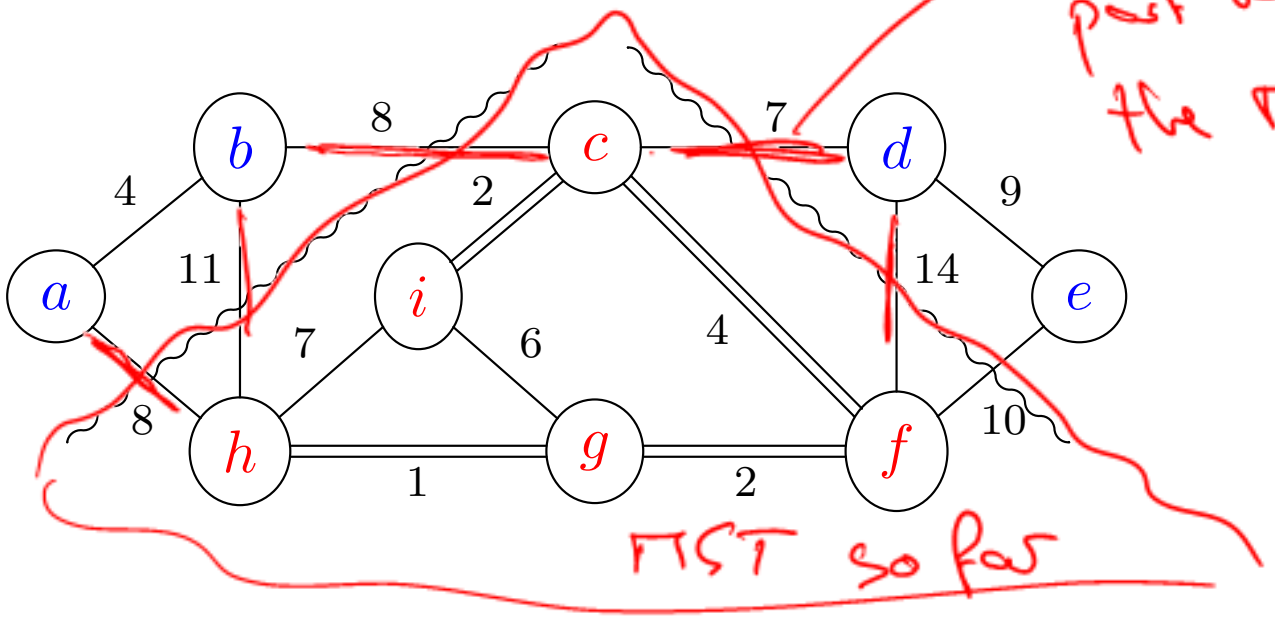
Greedy choice/1

- Let $G = (V, E)$ and $S \subseteq V$ be a **cut** of G
- That is, G is split into parts S and $V - S$

MINIMUM SPANNING TREE/3

Greedy choice/1

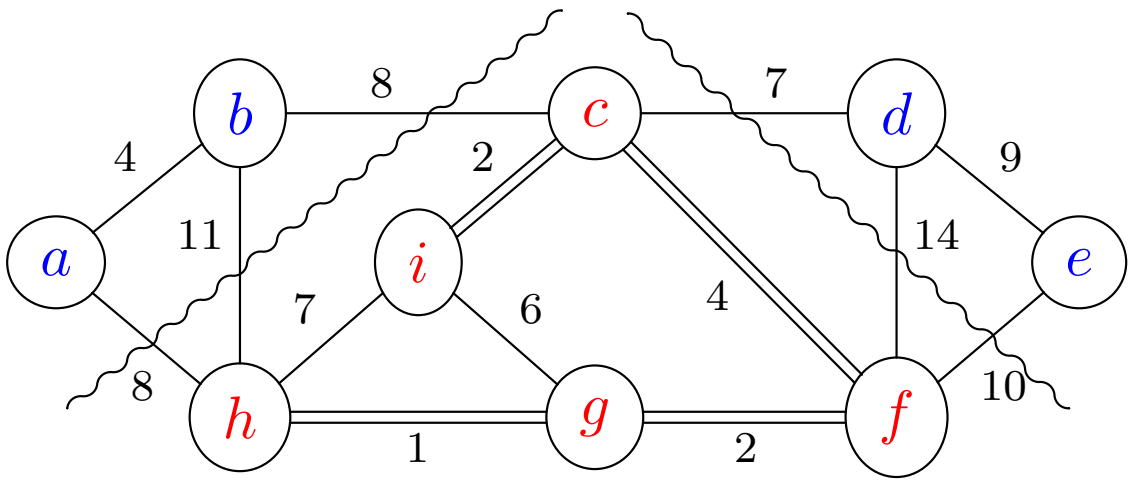
- Let $G = (V, E)$ and $S \subseteq V$ be a **cut** of G
- That is, G is split into parts S and $V - S$



MINIMUM SPANNING TREE/3

Greedy choice/1

- Let $G = (V, E)$ and $S \subseteq V$ be a **cut** of G
- That is, G is split into parts S and $V - S$



- (u, v) is a **light edge** if it is a min-weight edge of G that connects S and $V - S$ (e.g. (c, d) is such a light edge)

MINIMUM SPANNING TREE/4

Greedy choice/2

- In the above situation, if (u, v) is a light edge then it belongs to an MST of G

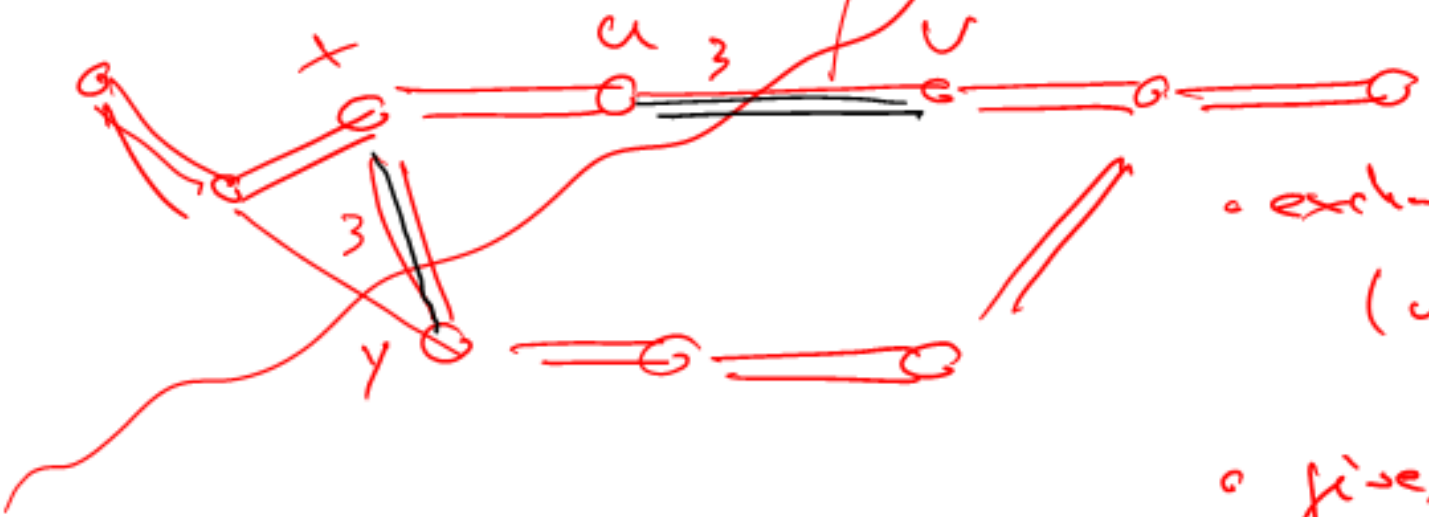
MINIMUM SPANNING TREE/4

Greedy choice/2

- In the above situation, if (u, v) is a light edge then it belongs to an MST of G
- Proof sketch
  - Assume that (u, v) is light but does not belong to any MST of G
  - Look at a path from u to v in some minimum spanning tree T of G
  - Let (x, y) be the first edge on this path that crosses from S to $V - S$
  - If the edge (x, y) of T is replaced by the edge (u, v), $w(T)$ decreases
  - This contradicts the assumption that T is a minimum spanning tree

EXERCISE SL10-1

Let (u, v) be a minimum weight edge in graph G. Show that (u, v) belongs to some MST of G. Hint: Show that from any MST without (u, v) you can construct a MST with (u, v).



minimum weight

- exchange (x, y) with (u, v)
- gives a spanning tree
- must be minimal because (u, v) has minimum weight
- (x, y) must also have min weight

Prim-Jarnik algorithm/1

- Vertex-based algorithm

PRIM-JARNIK ALGORITHM/1

- Vertex-based algorithm
- Grows a single minimum spanning tree T

PRIM-JARNIK ALGORITHM/1

- Vertex-based algorithm
- Grows a single minimum spanning tree T
- Set A covers part of T that is already computed
  - Uses a $pred$ field to indicate the predecessor

PRIM-JARNIK ALGORITHM/1

- Vertex-based algorithm
- Grows a single minimum spanning tree T
- Set A covers part of T that is already computed
  - Uses a $pred$ field to indicate the predecessor
- Annotates all the vertices v outside of the set A
  - $v.k$: minimum weight of an edge that connects vertex v with a vertex in A ($v.k = \infty$ if no such edge exists)

PRIM-JARNIK ALGORITHM/1

- Vertex-based algorithm
- Grows a single minimum spanning tree T
- Set A covers part of T that is already computed
  - Uses a *pred* field to indicate the predecessor
- Annotates all the vertices v outside of the set A
  - $v.k$: minimum weight of an edge that connects vertex v with a vertex in A ($v.k = \infty$ if no such edge exists)
- The algorithm relies on a min-priority queue
  - `init_min_pq`, `min_extract`, and `decrease_key`

PRIM-JARNIK ALGORITHM/1

- Vertex-based algorithm
- Grows a single minimum spanning tree T
- Set A covers part of T that is already computed
  - Uses a *pred* field to indicate the predecessor
- Annotates all the vertices v outside of the set A
  - $v.k$: minimum weight of an edge that connects vertex v with a vertex in A ($v.k = \infty$ if no such edge exists)
- The algorithm relies on a min-priority queue
  - `init_min_pq`, `min_extract`, and `decrease_key`
- The running time of the algorithm is $\Theta(E \log V)$

PRIM-JARNIK ALGORITHM/2

The algorithm



Algo: PrimJarnik(G,w,s)

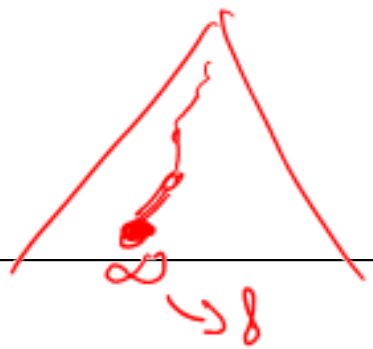
```
foreach v ∈ G.V do
    v.key = ∞;
    v.pred = NIL;
s.key = 0;
InitMinPriorityQueue(PQ,G.V);
while PQ ≠ ∅ do
    v = ExtractMin(PQ);
    foreach u ∈ v.adj do
        if u ∈ PQ ∧ w(v,u) < u.key then
            u.key = w(v,u);
            u.pred = v;
            DecreaseKey(PQ,u,w(u,v))
```

initialisation: ∞ costs to connect; no predecessors

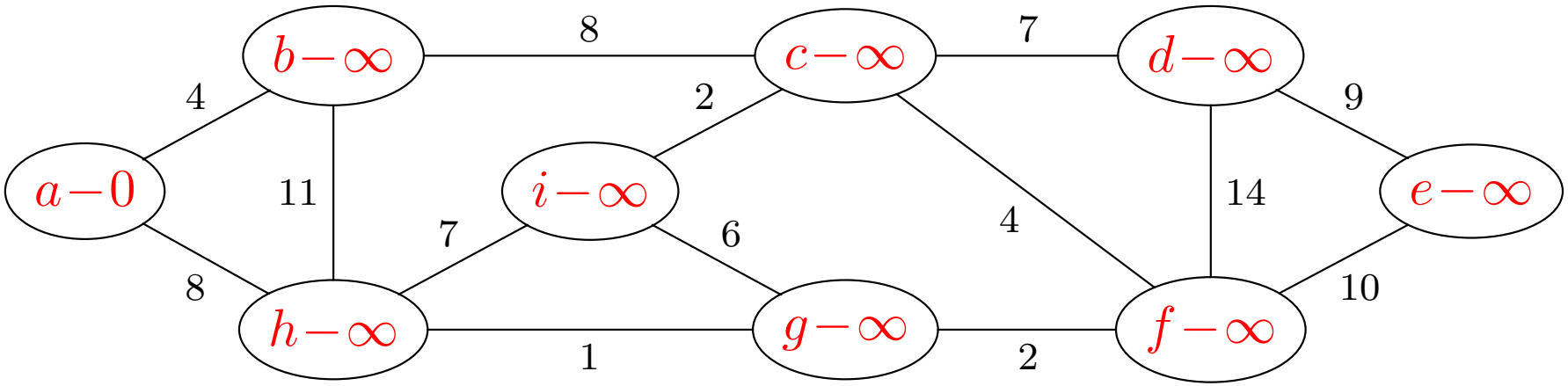
all nodes are put into PQ (s is first)

V lg V

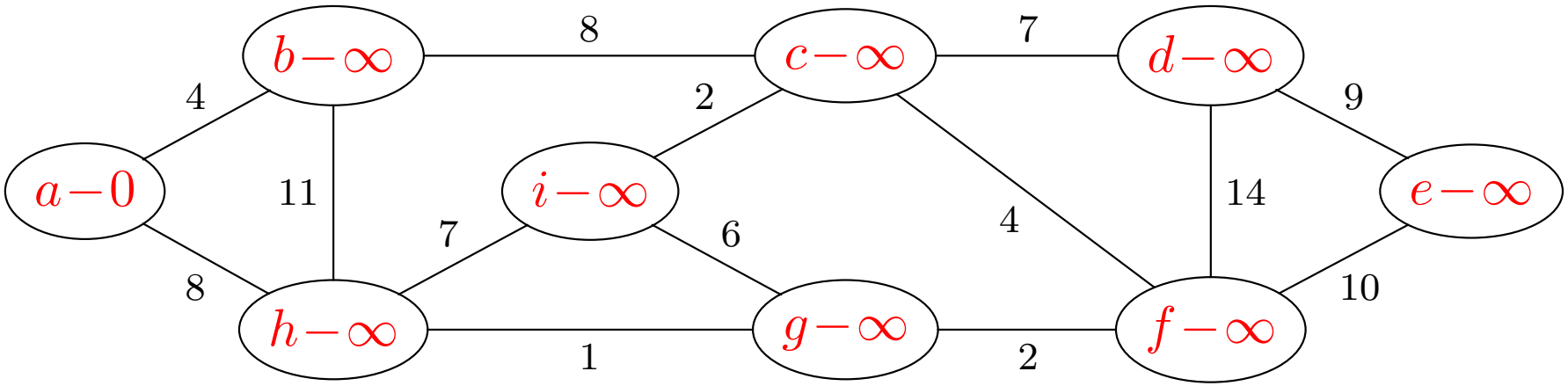
E lg V

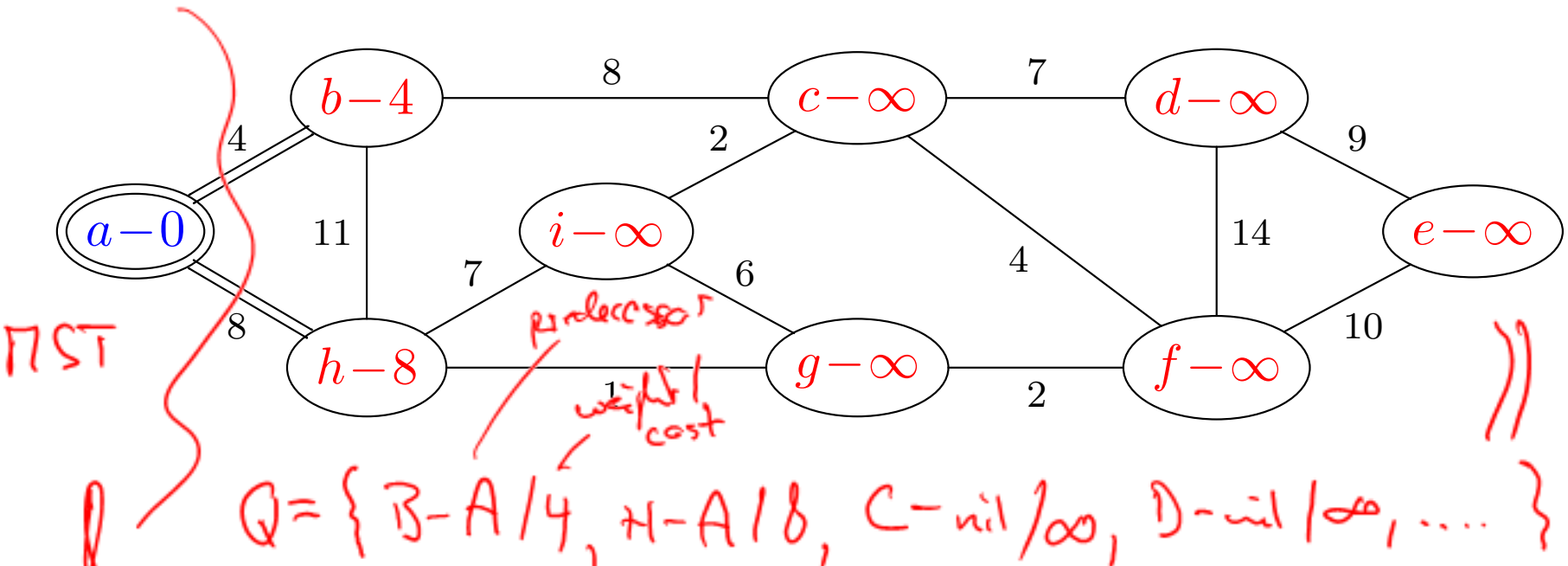


Prim-Jarnik example/1

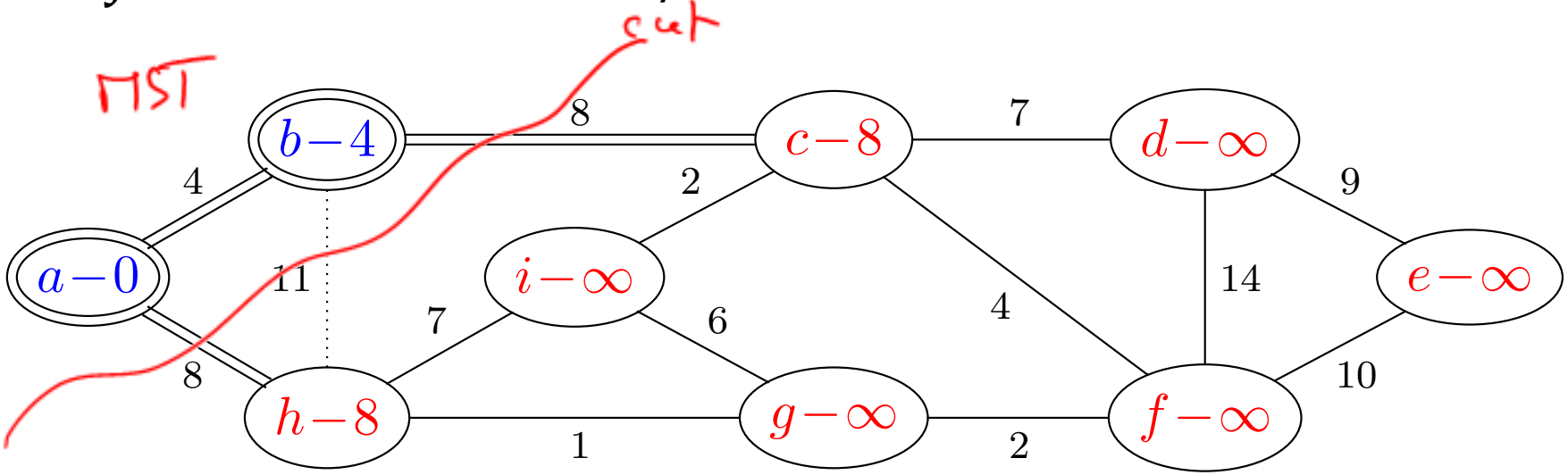


PRIM-JARNIK EXAMPLE/1

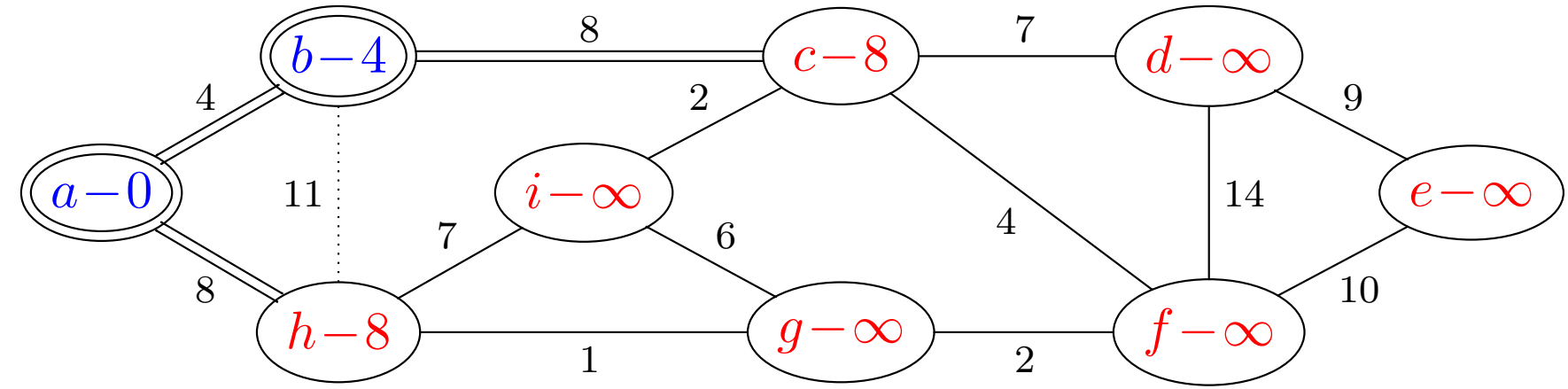


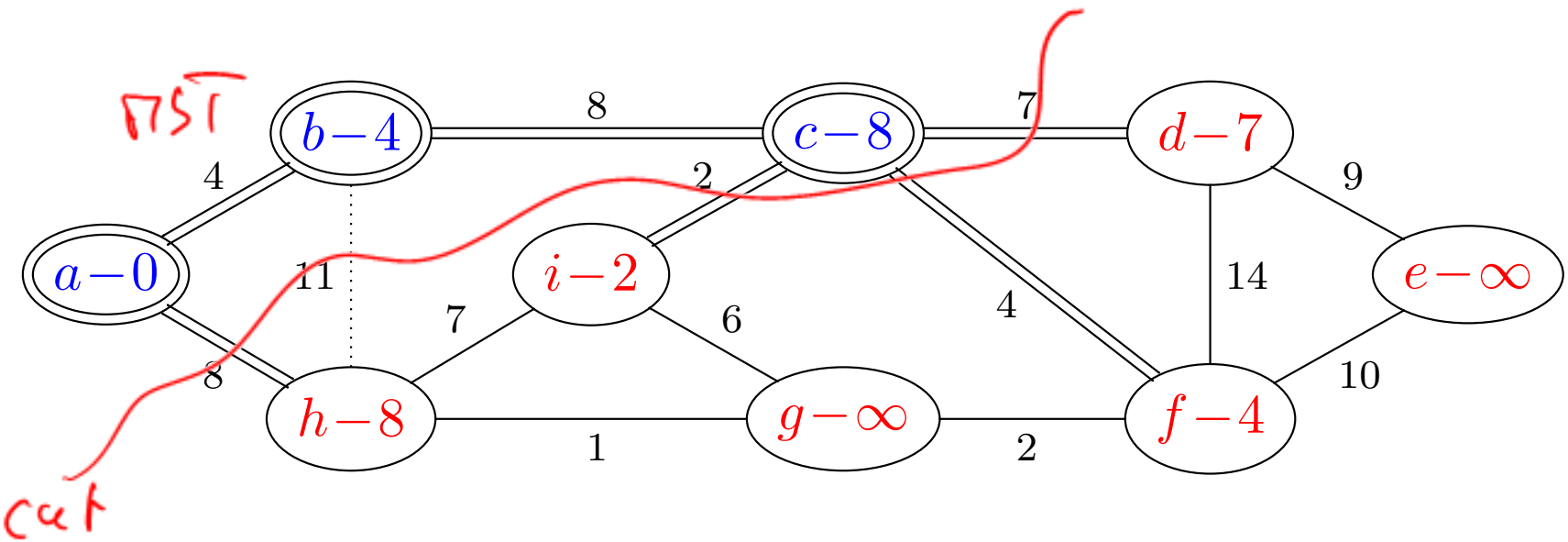


PRIM-JARNIK EXAMPLE/2



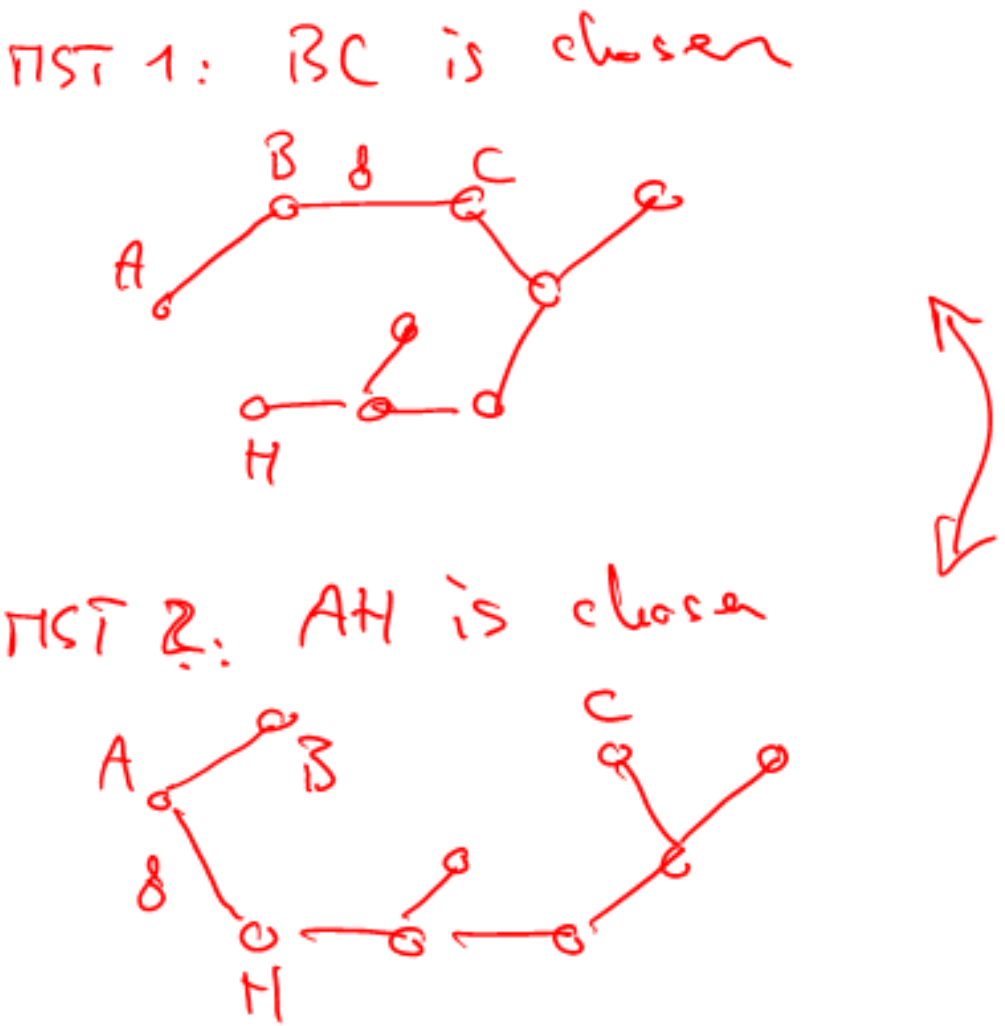
PRIM-JARNIK EXAMPLE/2



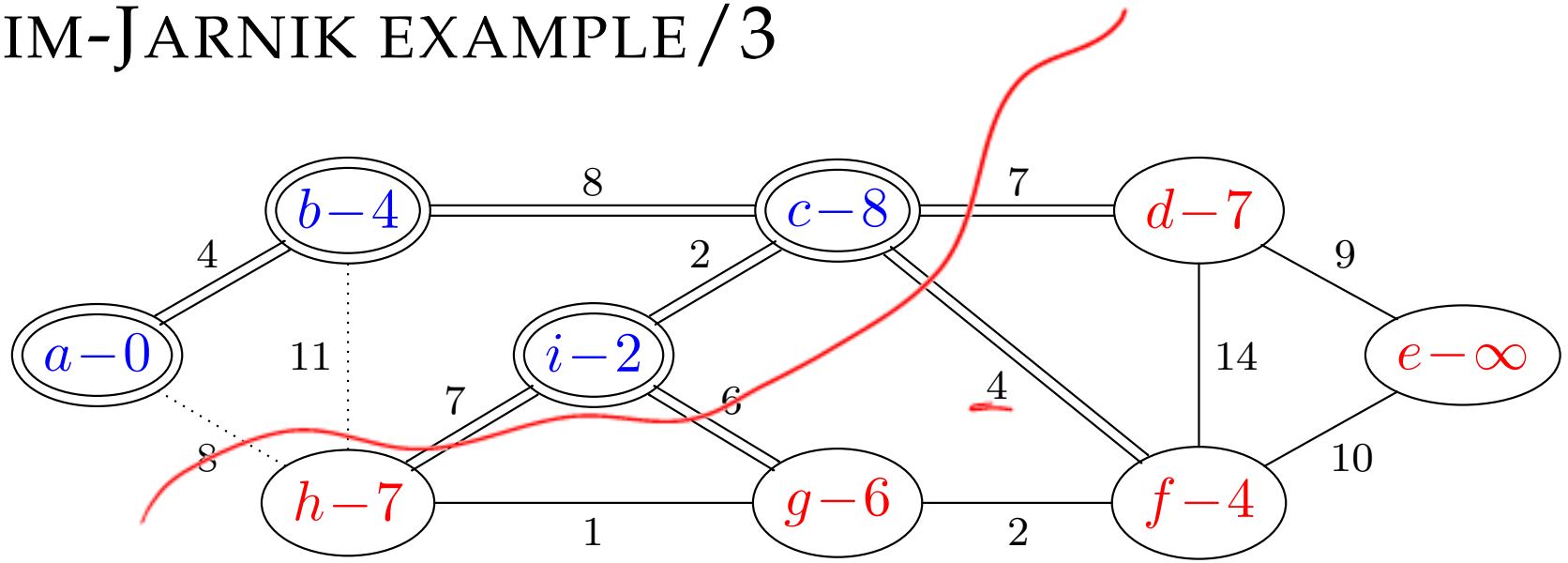


EXERCISE SL10-2

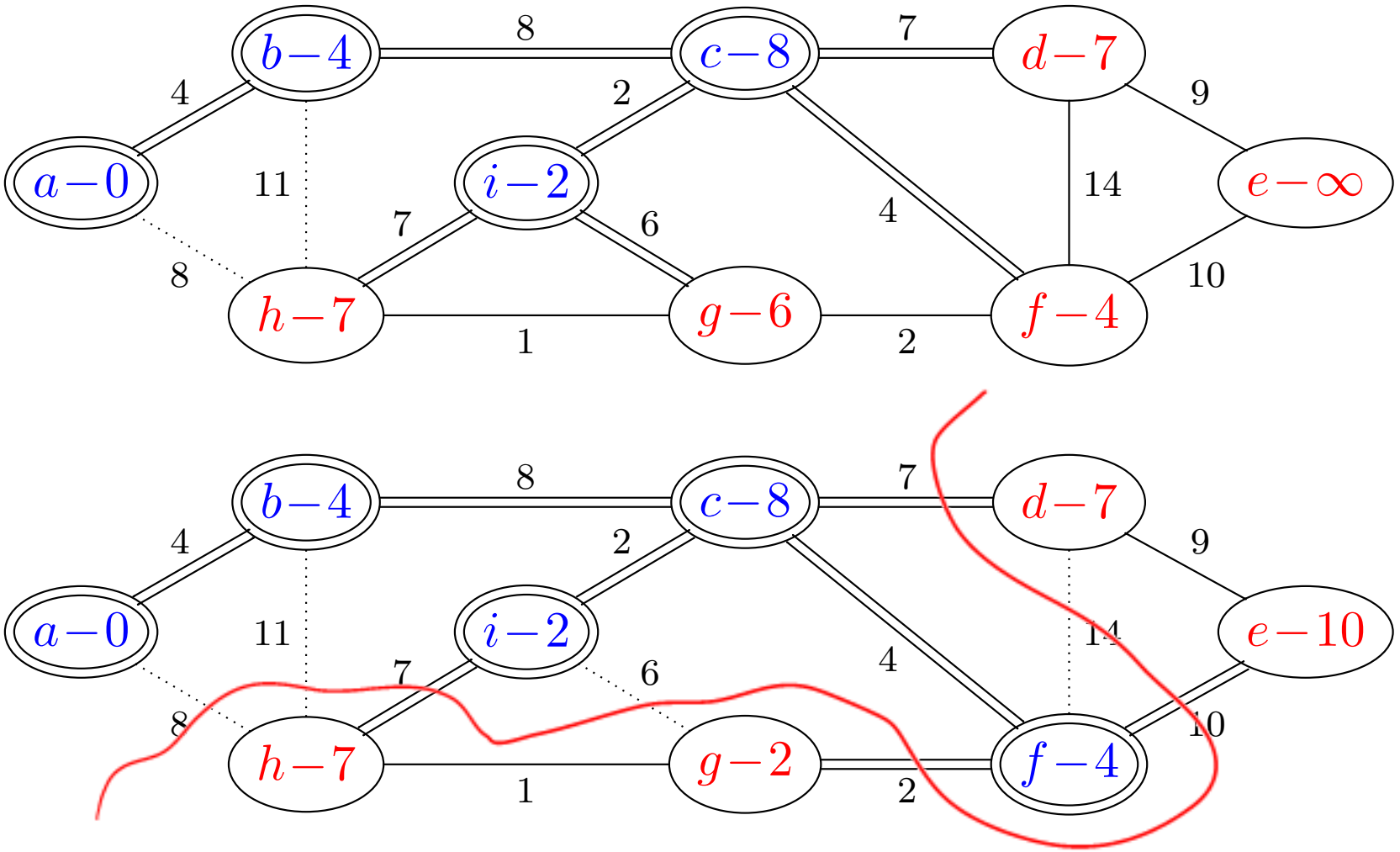
What happens if ~~BC~~ AH instead of ~~AH~~ BC is chosen?



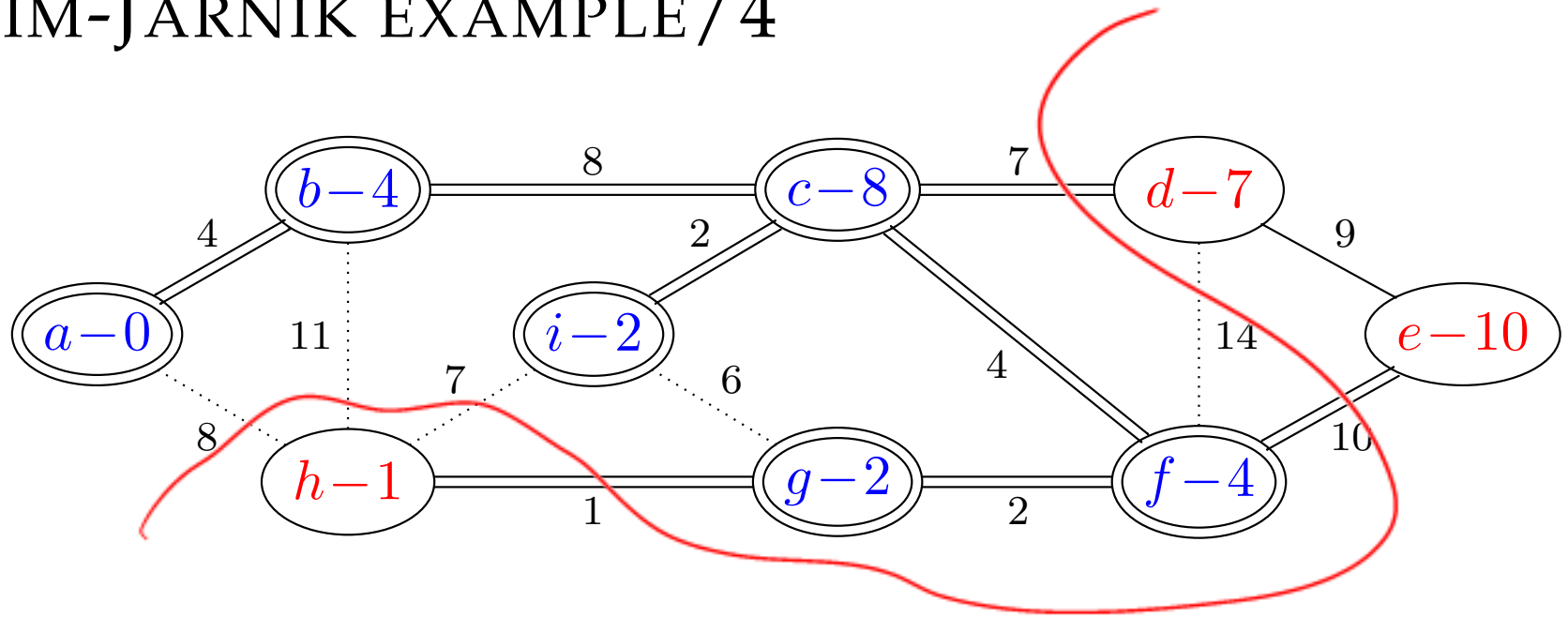
PRIM-JARNIK EXAMPLE/3



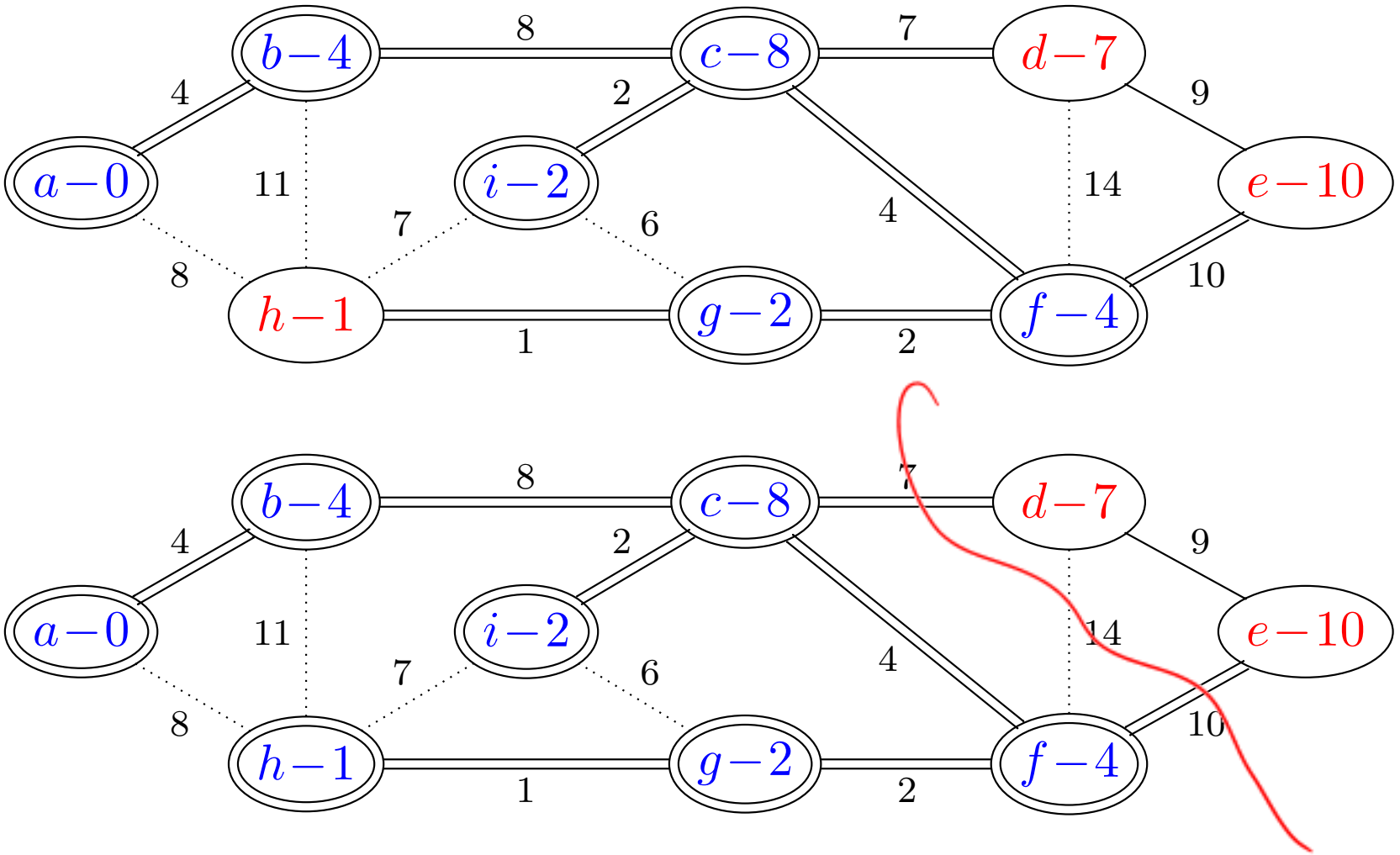
PRIM-JARNIK EXAMPLE/3



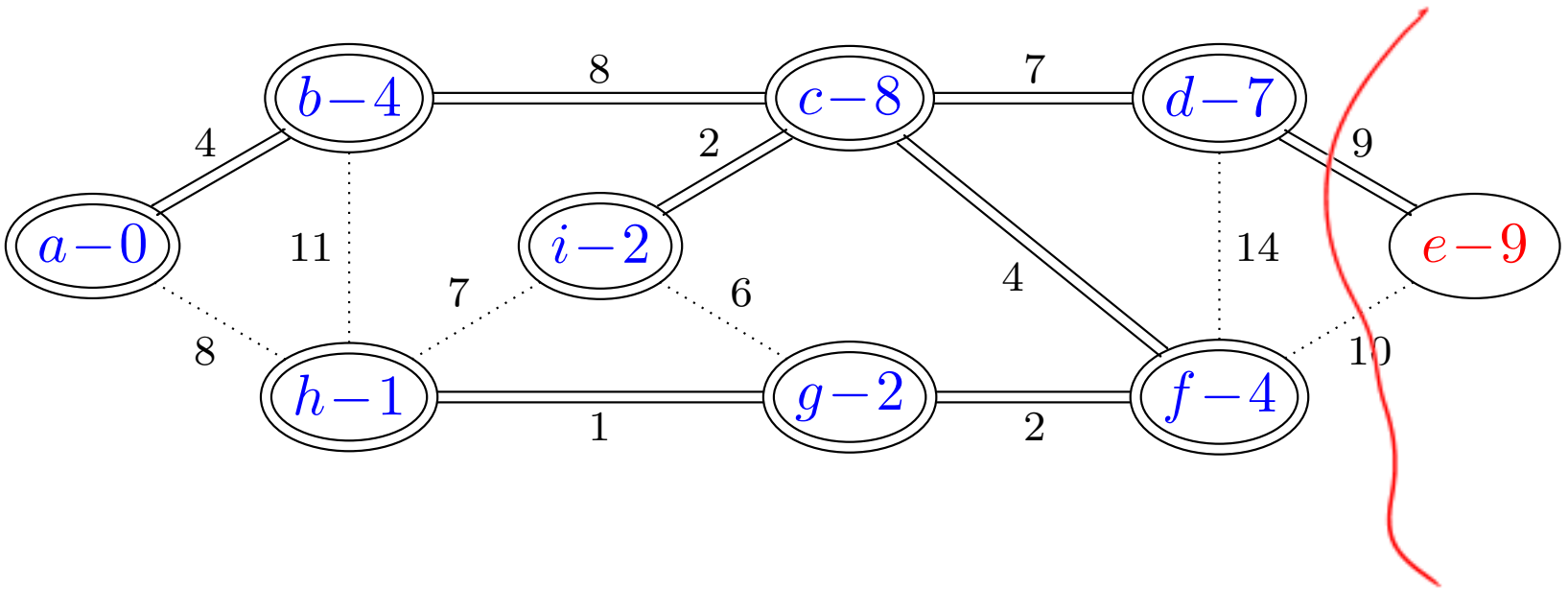
Prim-Jarnik example/4



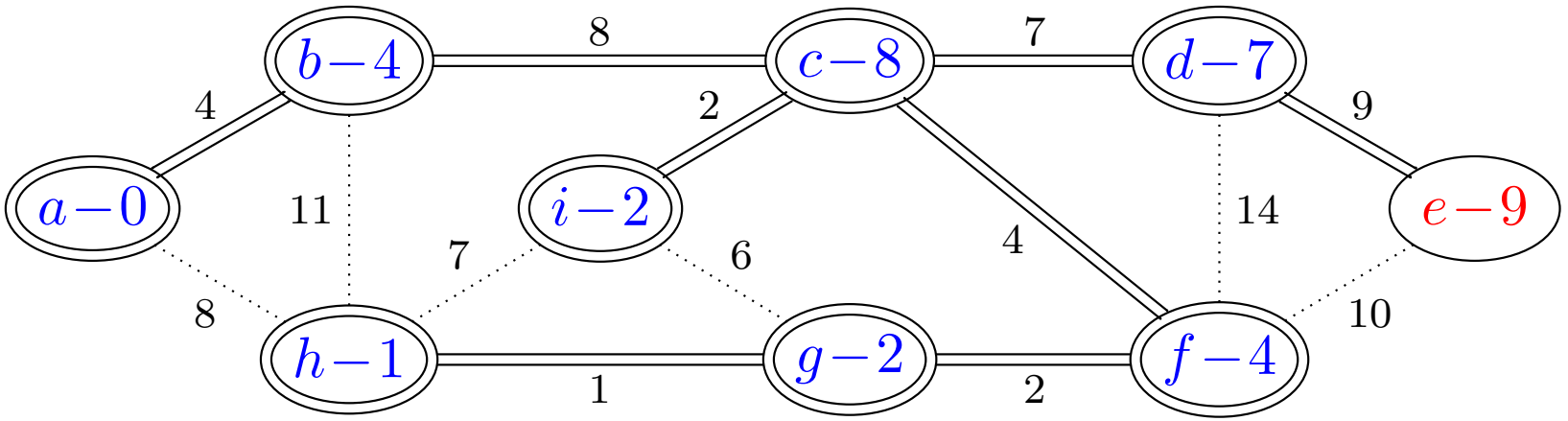
Prim-Jarnik example/4

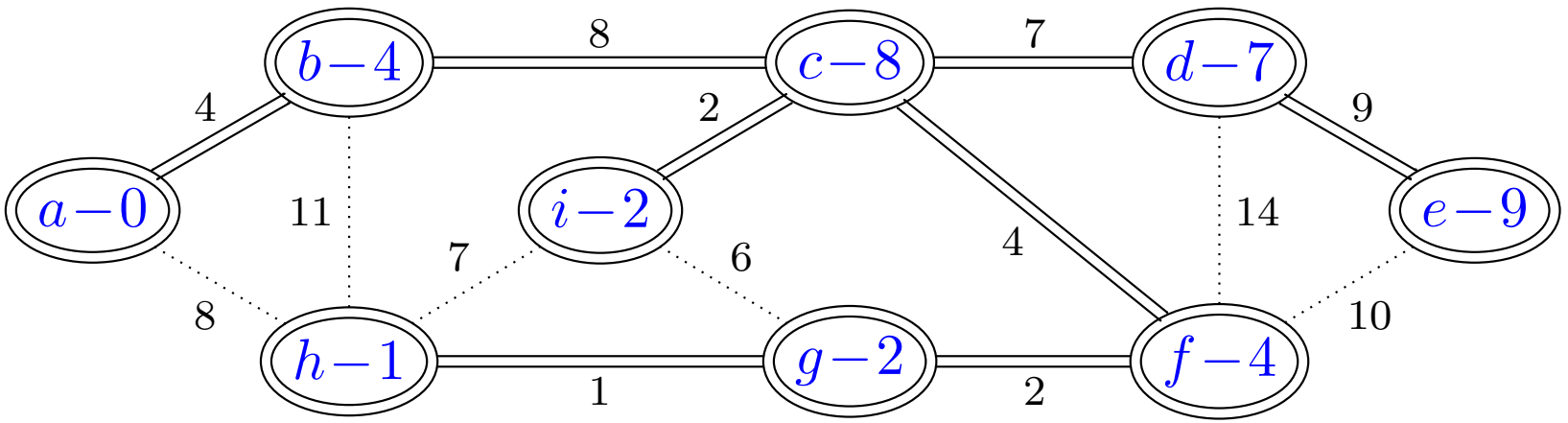


Prim-Jarnik algorithm/5



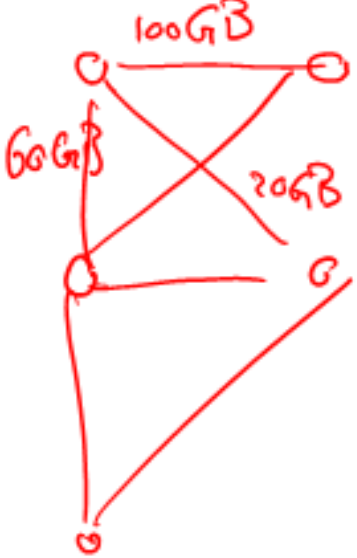
Prim-Jarnik algorithm/5





EXERCISE SL10-3

Assume a computer network. each computer is directly connected to a small number of other computers. Each direct connection has a bandwidth. Determine an efficient network that ensures any two computers are connected, minimizes the number of direct connections, and maximizes the bandwidth.



solution: maximum spanning tree with Prim algo

```
MaxST(G, s)
  for v ∈ G.V { v.key = -1; v.pred = NIL }
  s = 0
  initMaxPQ(Q, G.V)
  while PQ ≠ ∅ do
      u = extractMax(Q)
      for v ∈ u.adj do
          if v ∈ Q ∧ BW(u,v) > v.key
              modifyKey(Q,v)
              v.pred = u
              v.key = BW(u,v)
```

PRIM-JARNIK RUNNING TIME

- $Time = V * T(ExtractMin) + O(E) * T(DecreaseKey)$
- Depends on the implementation of the priority queue:

Q	T(ExtractMin)	t(DecreaseKey)	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\log V)$	$O(\log V)$	$O(E \log V)$

- $E \geq V - 1, E \leq V^2, E = O(V^2)$
- Binary heap implementation:
  $Time = O(V \log V + E \log V) = O(V^2 \log V) = O(E \log V)$

TABLE OF CONTENTS — SL10





1. Weighted graphs

2. Minimum spanning tree (MST)

3. Single-source shortest path (SSSP)
   - Single-source shortest path
   - Bellman-Ford algorithm
   - Dijkstra's algorithm

general case with negative weights

(assumes: no negative weights)

SINGLE-SOURCE SHORTEST PATH/1

Shortest paths

- Generalize directed graphs to weighted setting

SINGLE-SOURCE SHORTEST PATH/1

Shortest paths

- Generalize directed graphs to weighted setting
- Directed graph (V, E), weight function $w\colon E \to \mathbb{R}$

SINGLE-SOURCE SHORTEST PATH/1

Shortest paths

- Generalize directed graphs to weighted setting
- Directed graph (V, E), weight function $w\colon E \to \mathbb{R}$
- Definition of weight of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

SINGLE-SOURCE SHORTEST PATH/1

Shortest paths

- Generalize directed graphs to weighted setting
- Directed graph (V, E), weight function $w\colon E \to \mathbb{R}$
- Definition of weight of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

- Shortest path = a path of minimum weight (cost)

SINGLE-SOURCE SHORTEST PATH/1

Shortest paths

- Generalize directed graphs to weighted setting
- Directed graph (V, E), weight function $w\colon E \to \mathbb{R}$
- Definition of weight of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

- Shortest path = a path of minimum weight (cost)
- Single-source shortest path (SSSP) problem
  - Shortest path from source vertex s to any other vertex
  - Other problems: single pair, all pairs, unweighted (BFS)

more complex

same cost as SSSP

SINGLE-SOURCE SHORTEST PATH/1

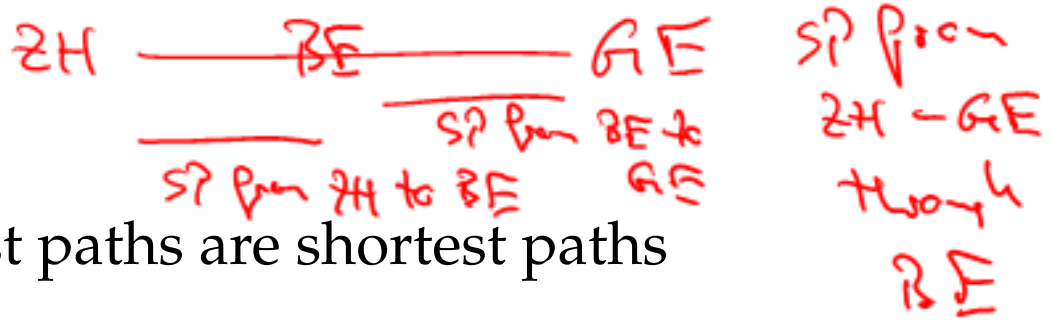
Shortest paths

- Generalize directed graphs to weighted setting
- Directed graph (V, E), weight function $w\colon E \to \mathbb{R}$
- Definition of weight of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

- Shortest path = a path of minimum weight (cost)
- Single-source shortest path (SSSP) problem
  - Shortest path from source vertex s to any other vertex
  - Other problems: single pair, all pairs, unweighted (BFS)

Optimal substructure



- The subpaths of shortest paths are shortest paths

SINGLE-SOURCE SHORTEST PATH/2

Negative weights

- Negative weights are OK, as long as there are no negative-weight cycles
  - Otherwise, could get paths of arbitrary small "length"

SINGLE-SOURCE SHORTEST PATH/2

Negative weights

- Negative weights are OK, as long as there are no negative-weight cycles
  - Otherwise, could get paths of arbitrary small "length"

Paths and cycles

- Shortest paths cannot have cycles
  - Otherwise, improve shortest path by removing cycles
  - No shortest path can have length greater than $|V| - 1$

SINGLE-SOURCE SHORTEST PATH/2

Negative weights

- Negative weights are OK, as long as there are no negative-weight cycles
  - Otherwise, could get paths of arbitrary small "length"

Paths and cycles

- Shortest paths cannot have cycles
  - Otherwise, improve shortest path by removing cycles
  - No shortest path can have length greater than $|V| - 1$

Relaxation/1

- For each vertex v in the graph, maintain $v.dist$, the estimate of the shortest path from s
  - At the start, $v.dist$ is initialized to ∞ and $s.dist$ to 0

SINGLE-SOURCE SHORTEST PATH/3

Relaxation/2

- Relaxing an edge (u, v) means testing whether one can improve the shortest path to v found so far by going through u

SINGLE-SOURCE SHORTEST PATH/3

Relaxation/2

- Relaxing an edge (u, v) means testing whether one can improve the shortest path to v found so far by going through u

Algo: Relax(u,v,w)

if $v.dist > u.dist + w(u, v)$ **then**
    v.dist = u.dist + w(u,v);
    v.pred = u

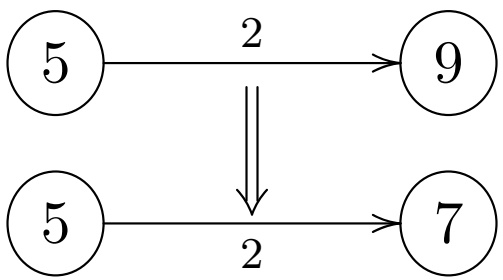
SINGLE-SOURCE SHORTEST PATH/3

Relaxation/2

- Relaxing an edge (u, v) means testing whether one can improve the shortest path to v found so far by going through u

Algo: Relax(u,v,w)

if $v.dist > u.dist + w(u, v)$ **then**
  v.dist = u.dist + w(u,v);
  v.pred = u



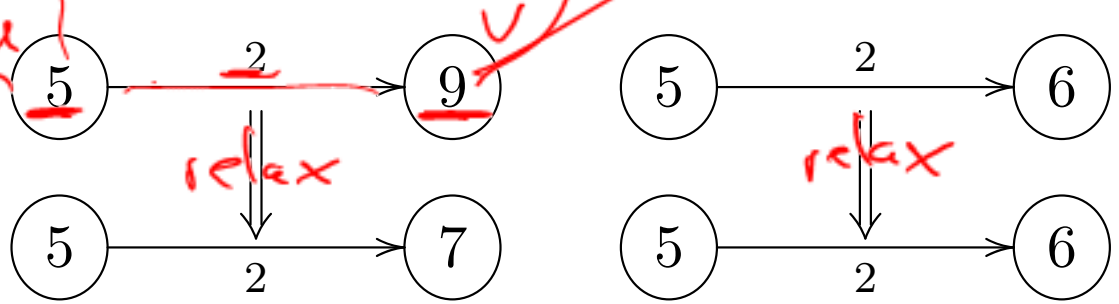
SINGLE-SOURCE SHORTEST PATH/3

Relaxation/2

- Relaxing an edge (u, v) means testing whether one can improve the shortest path to v found so far by going through u

Algo: Relax(u,v,w)

if $v.dist > u.dist + w(u, v)$ **then**
  v.dist = u.dist + w(u,v);
  v.pred = u



BELLMAN-FORD ALGORITHM/1

- Solves single-source shortest path problem for the case where edge weights may be negative

BELLMAN-FORD ALGORITHM/1

- Solves single-source shortest path problem for the case where edge weights may be negative
- Returns boolean value indicating whether there is a negative-weight cycle reachable from source vertex s.
  - If there is such a cycle then no solution exists

BELLMAN-FORD ALGORITHM/1

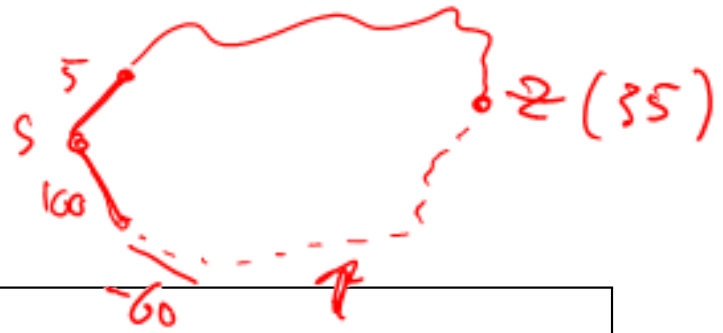
- Solves single-source shortest path problem for the case where edge weights may be negative
- Returns boolean value indicating whether there is a negative-weight cycle reachable from source vertex s.
  - If there is such a cycle then no solution exists
- Employs relaxation, progressively decreasing $v.dist$

BELLMAN-FORD ALGORITHM/1

- Solves single-source shortest path problem for the case where edge weights may be negative
- Returns boolean value indicating whether there is a negative-weight cycle reachable from source vertex s.
  - If there is such a cycle then no solution exists
- Employs relaxation, progressively decreasing $v.dist$
- The running time of the algorithm is $\Theta(|E| \cdot |V|)$

$\approx |V|^3$

BELLMAN-FORD ALGORITHM/2



The algorithm

Algo: BellmanFord(G,w,s)

```
foreach v ∈ G.V do
    v.dist = ∞; v.pred = NIL;
for i = 1 to |G.V|-1 do
    foreach (u, v) ∈ G.E do
        Relax(u,v,w);
foreach (u, v) ∈ G.E do
    if v.dist > u.dist + w(u, v) then
        return FALSE;
return TRUE;
```

initialisation

s.dist = 0

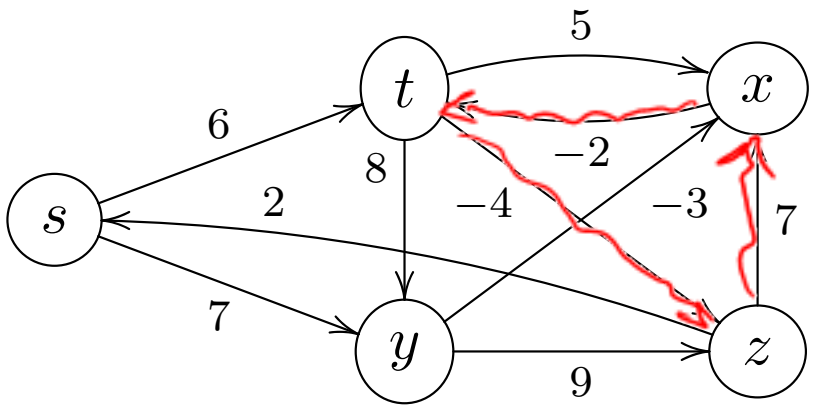
no good order ⇒ we relax all edges |V|-1 times

if we can relax more ⇒ we have a negative weight cycle

BELLMAN-FORD ALGORITHM/3

An example/1

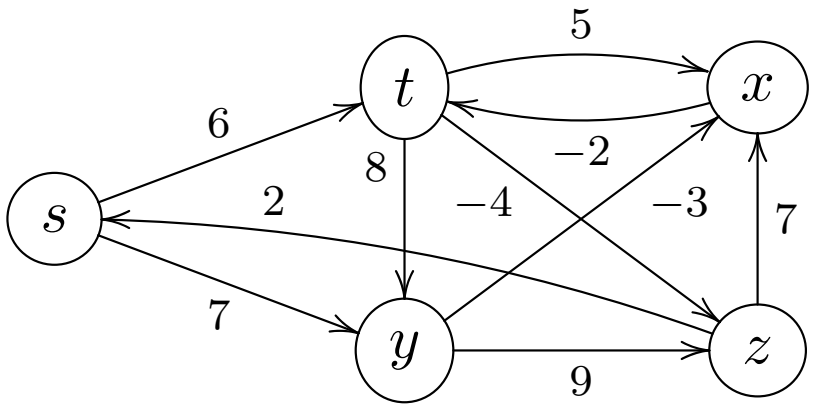
- Consider the following weighted graph



BELLMAN-FORD ALGORITHM/3

An example/1

- Consider the following weighted graph



- Relax the edges in the following order (for this example)

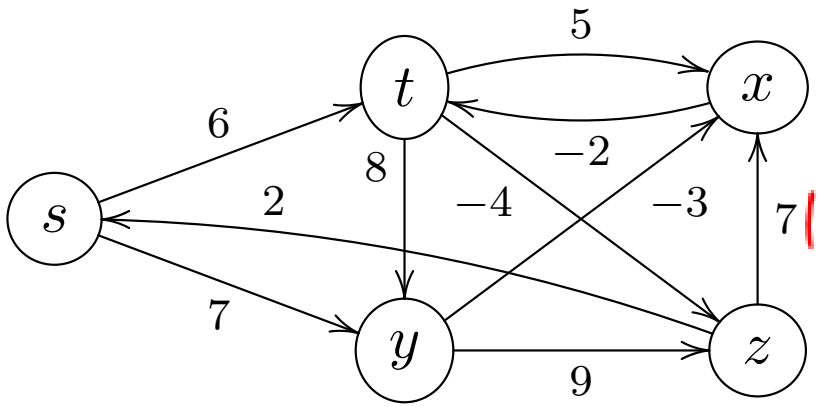
1. (t, x)	2. (t, y)	3. (t, z)	4. (x, t)	5. (y, x)
6. (y, z)	7. (z, x)	8. (z, s)	9. (s, t)	10. (s, y)

seq does not matter for final solution

BELLMAN-FORD ALGORITHM/3

An example/1

- Consider the following weighted graph



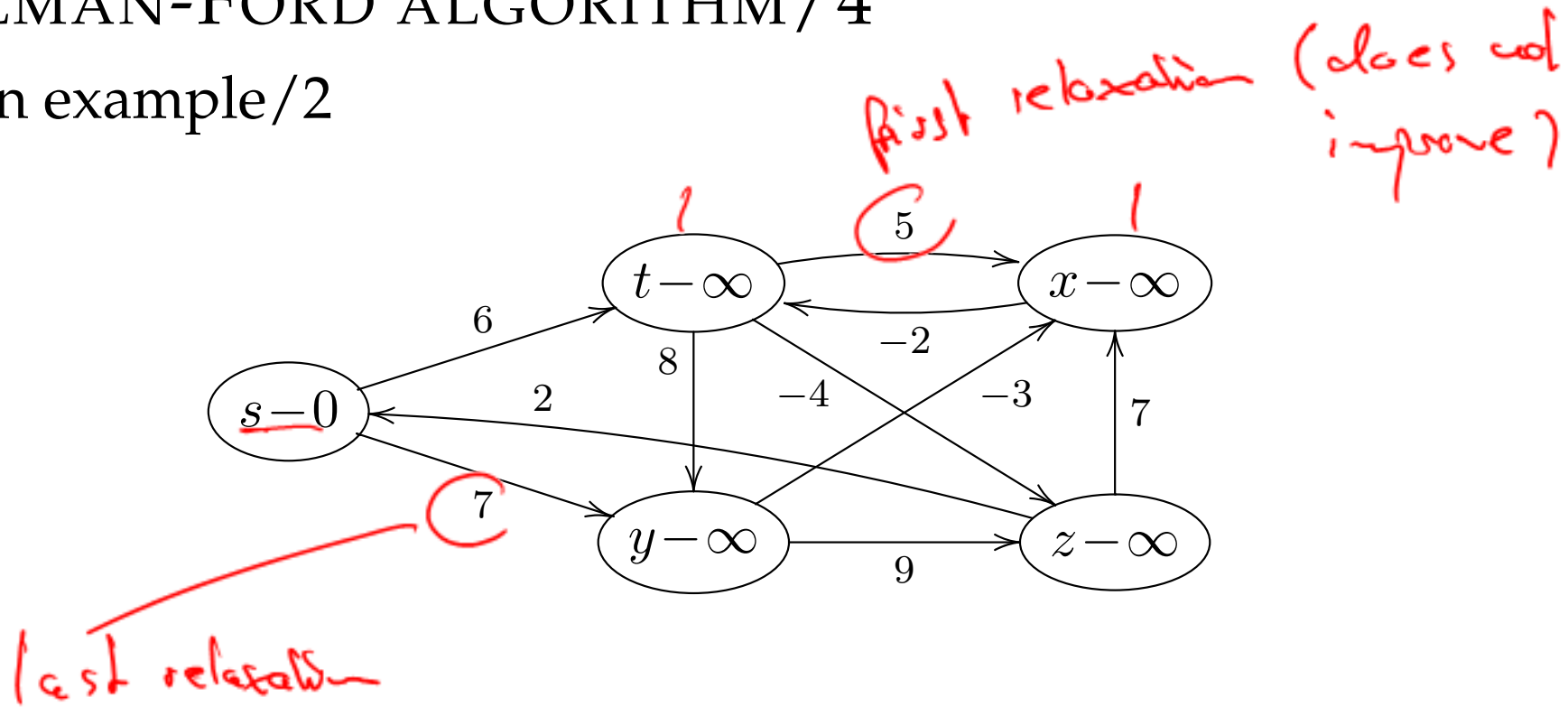
- Relax the edges in the following order

$$1.\ (t,x) \quad 2.\ (t,y) \quad 3.\ (t,z) \quad 4.\ (x,t) \quad 5.\ (y,x)$$
$$6.\ (y,z) \quad 7.\ (z,x) \quad 8.\ (z,s) \quad 9.\ (s,t) \quad 10.\ (s,y)$$

- Negative-weight cycle: let $w(z,x) = 4$

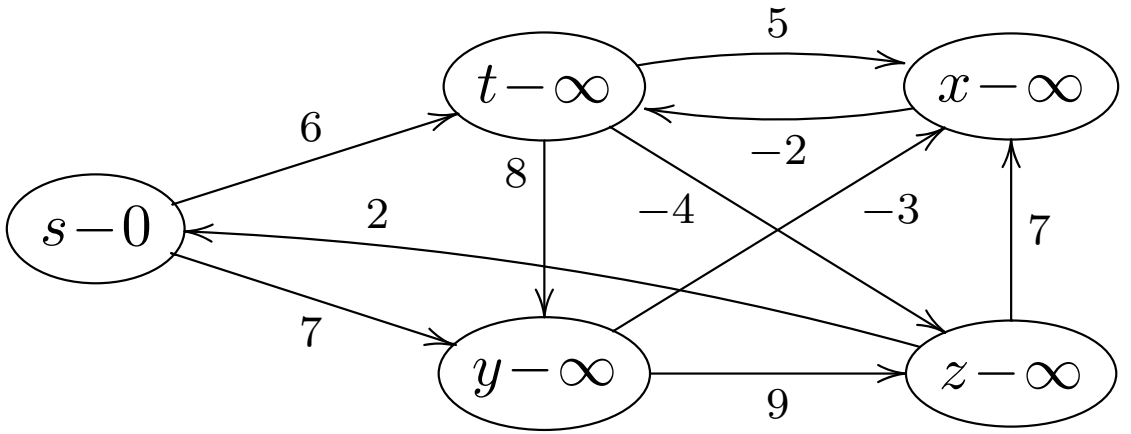
BELLMAN-FORD ALGORITHM/4

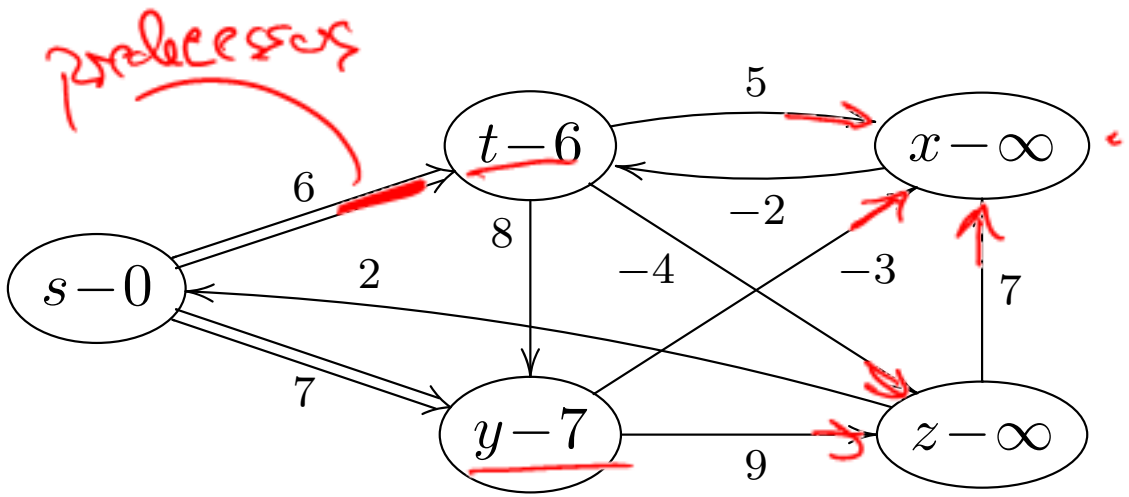
An example/2



Bellman-Ford algorithm/4

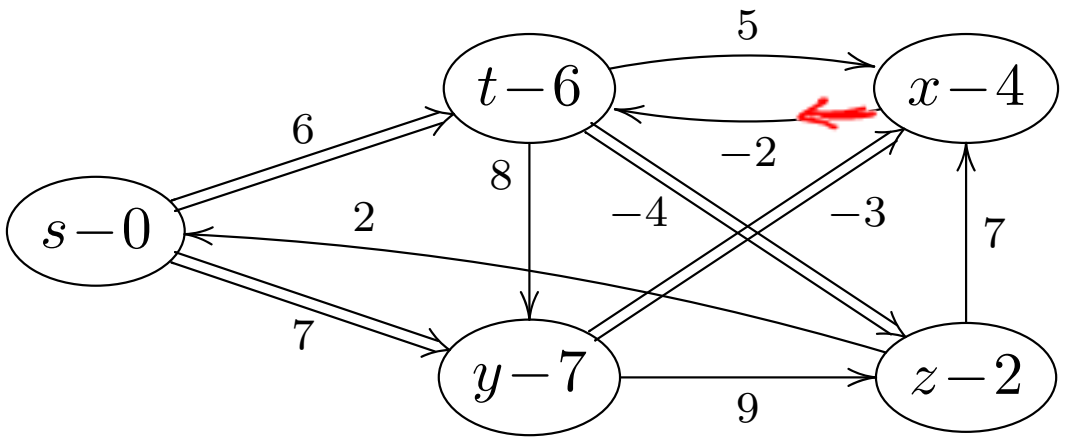
An example/2





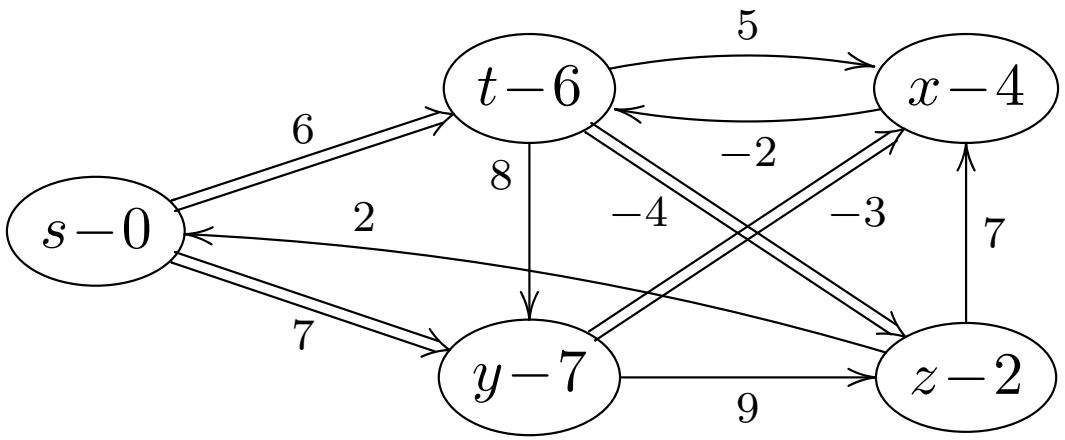
Bellman-Ford algorithm/5

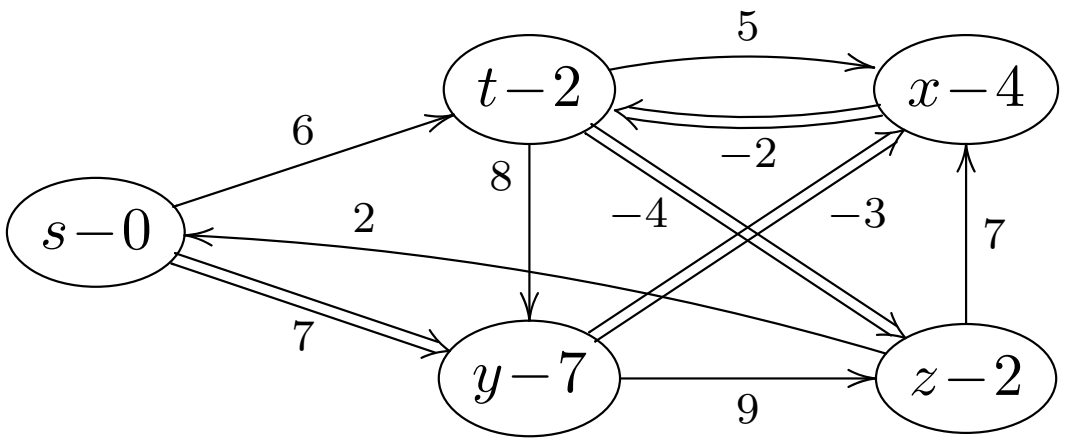
An example/3



Bellman-Ford algorithm/5

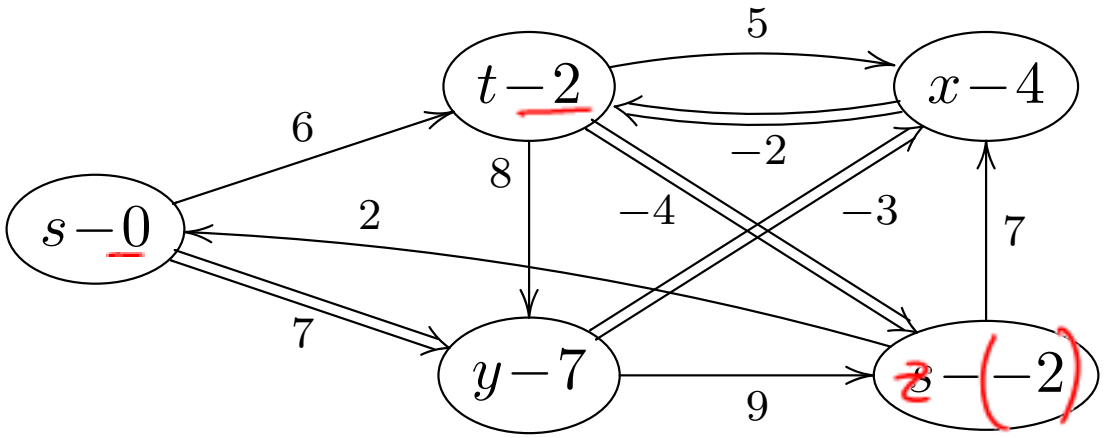
An example/3





BELLMAN-FORD ALGORITHM/6

An example/4



sp tree:

s −
7
y
−3
x
−2
t
−4
z

SSSP IN DAGS

- Finding shortest paths in a DAG is much easier

SSSP IN DAGS

- Finding shortest paths in a DAG is much easier
- Topological sort determines order of relaxations
- Algorithm:
  1. sort G topologically
  2. consider each vertex in topological sort order
  3. relax each edge leaving the vertex
- We have one relaxation per edge, which gives a runtime of $O(V + E)$ (V times faster than Bellman-Ford)

$O(V^2)$ vs $O(V^3)$

SSSP IN DAGS

- Finding shortest paths in a DAG is much easier
- Topological sort determines order of relaxations
- Algorithm:
  1. sort G topologically
  2. consider each vertex in topological sort order
  3. relax each edge leaving the vertex
- We have one relaxation per edge, which gives a runtime of $O(V + E)$ (V times faster than Bellman-Ford)
- Note: in a DAG there is no negative-weight cycle

Dijkstra's algorithm/1



- Solves single-source shortest path problem for the case where edge weights are nonnegative

Dijkstra's algorithm/1

- Solves single-source shortest path problem for the case where edge weights are nonnegative
- Greedy, similar to Prim-Jarnik algorithm for MST
  - Uses a priority queue, changes $dist$ instead of k

Dijkstra's algorithm/1

- Solves single-source shortest path problem for the case where edge weights are nonnegative
- Greedy, similar to Prim-Jarnik algorithm for MST
  - Uses a priority queue, changes $dist$ instead of k
- In some respect also similar to breadth-first search
  - $v.dist$ does not change further once the vertex v is removed from the (priority) queue

Dijkstra's algorithm/1

- Solves single-source shortest path problem for the case where edge weights are nonnegative
- Greedy, similar to Prim-Jarnik algorithm for MST
  - Uses a priority queue, changes $dist$ instead of k
- In some respect also similar to breadth-first search
  - $v.dist$ does not change further once the vertex v is removed from the (priority) queue
- The running time of the algorithm is $\Theta(|E| \lg |V|)$
  - It is thus lower than the running time of Bellman-Ford



DIJKSTRA'S ALGORITHM/2

The algorithm

Algo: Dijkstra(G,w,s)

```
foreach v ∈ G.V do
    v.dist = ∞; v.pred = NIL;
s.dist = 0;
InitMinPQ(PQ,G.V);
while PQ ≠ ∅ do
    v = ExtractMin(PQ);
    foreach u ∈ v.adj do
        Relax(v,u,w);
        DecreaseKey(PQ,u,u.dist)
```

initialization

all nodes in PQ

|E| times in total

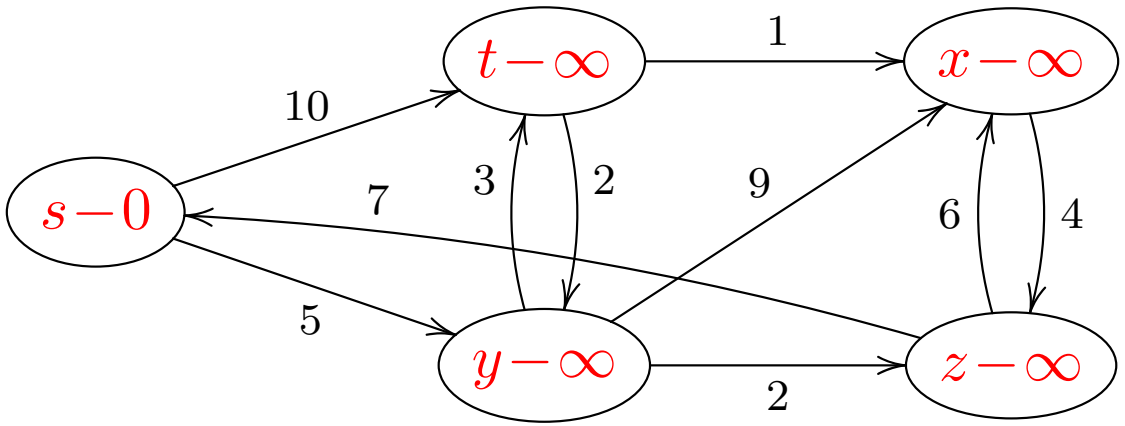
(= for each node all its edges)

= all edges

Dijkstra's algorithm/3

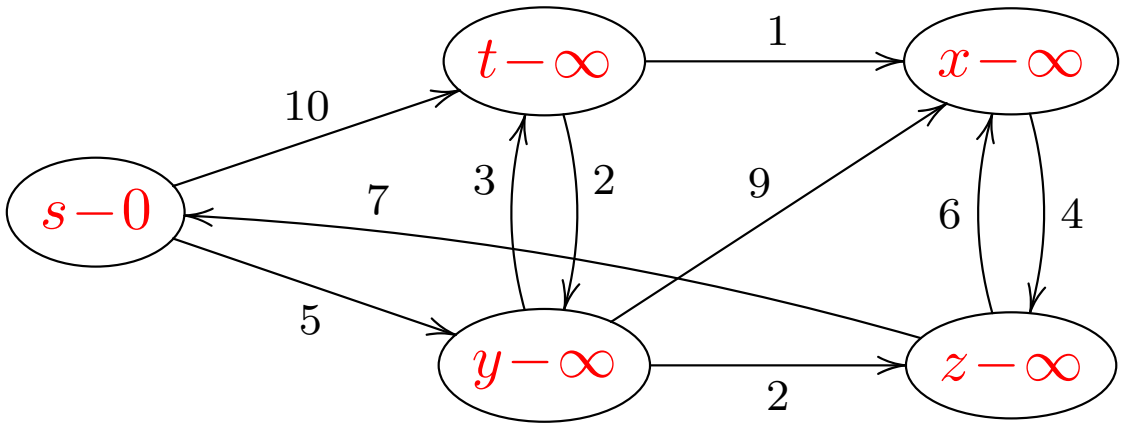
An example/1

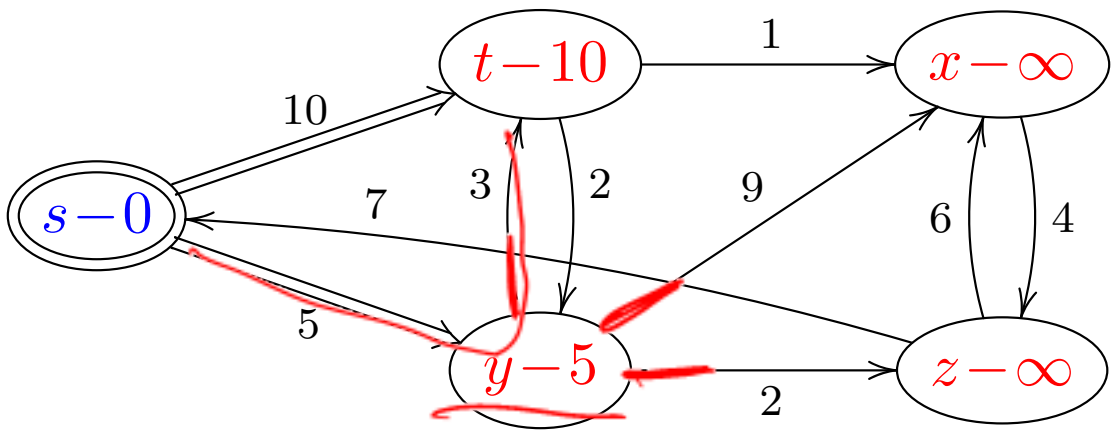
$Q = [\, s-0,\ t-\infty,\ x-\infty,\ y-\infty,\ z-\infty \,]$



Dijkstra's algorithm/3

An example/1

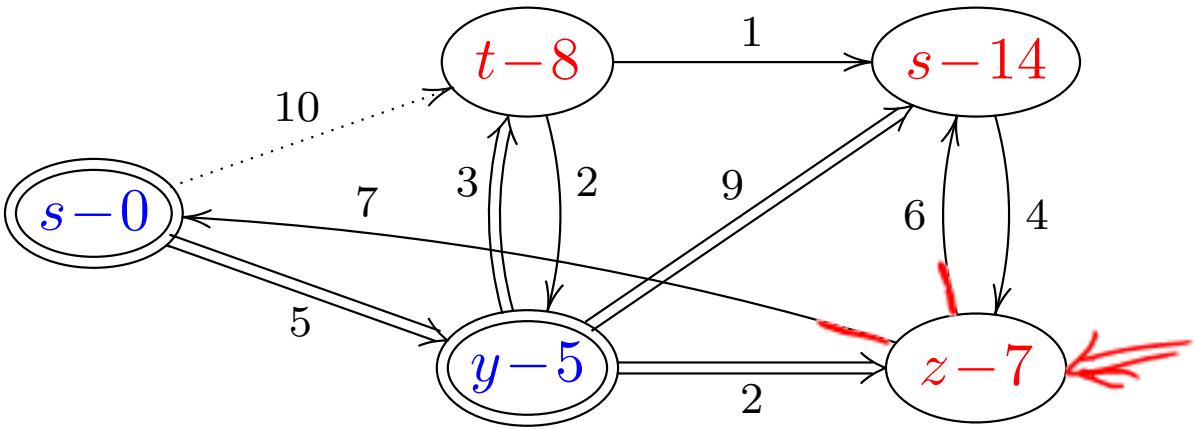




$Q = [y-5,\ t-16,\ x-\infty,\ z-\infty]$

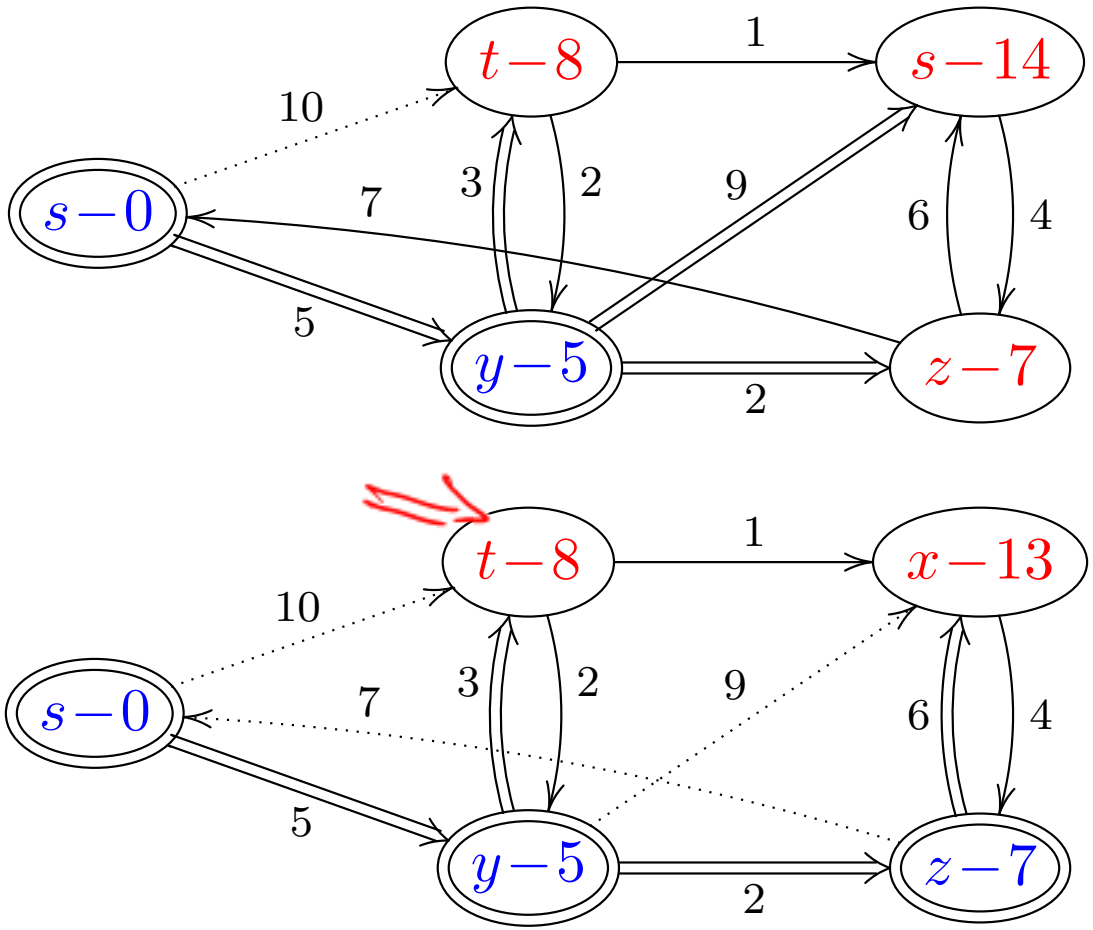
Dijkstra's algorithm/4

An example/2



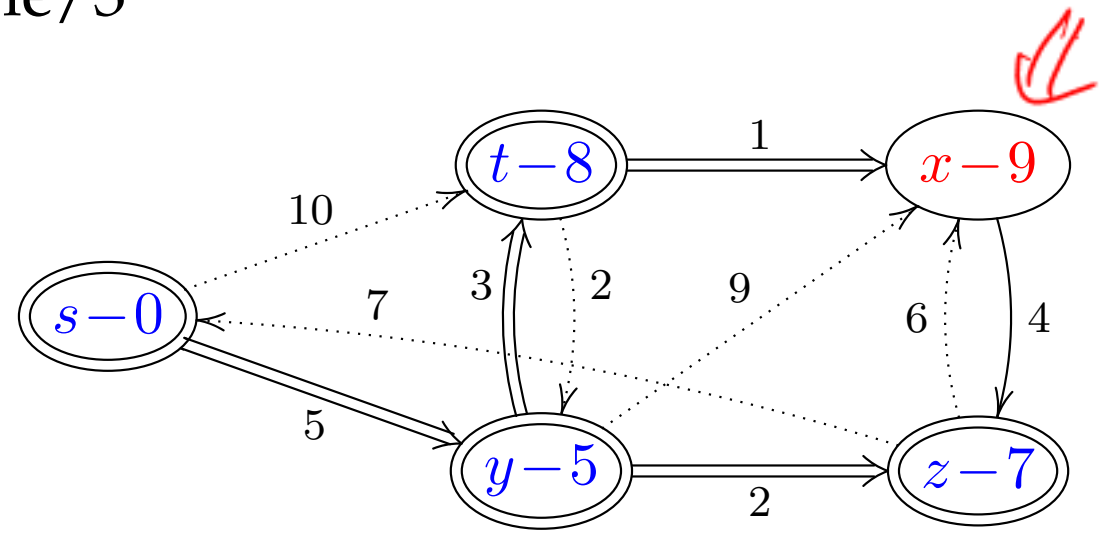
Dijkstra's algorithm/4

An example/2



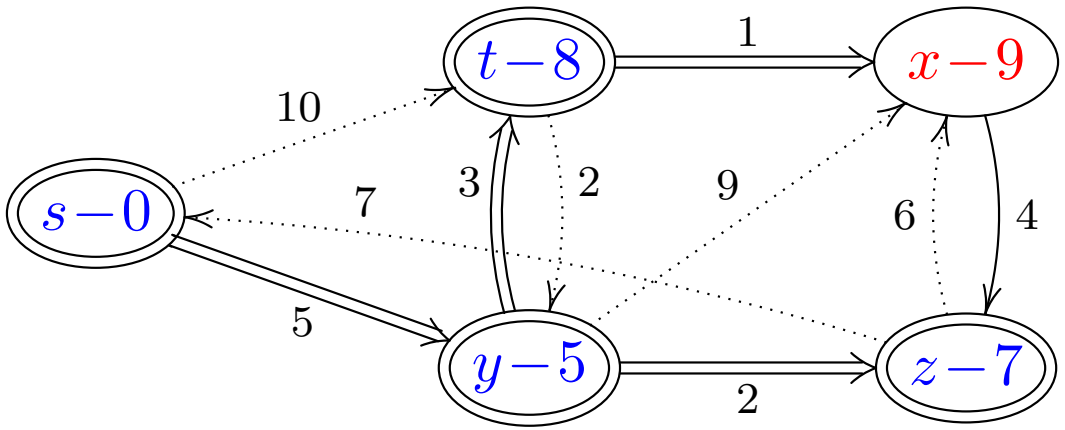
Dijkstra's algorithm/5

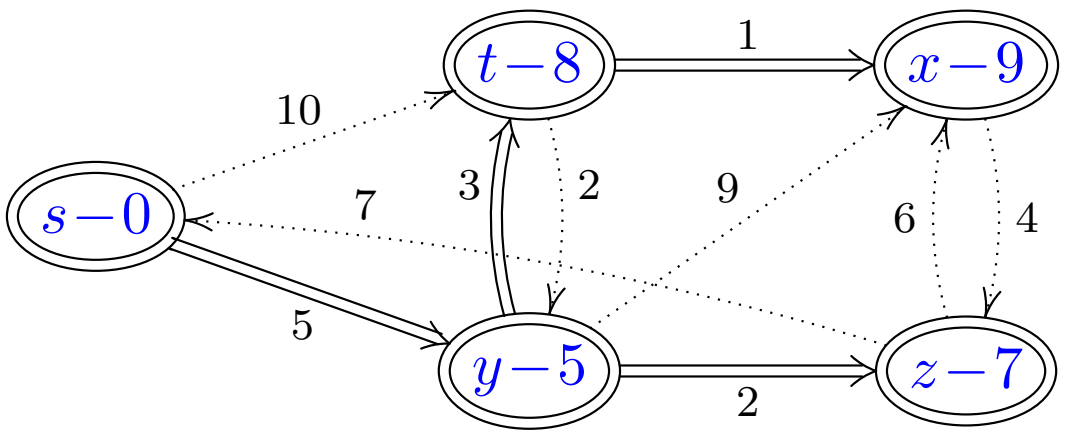
An example/3



Dijkstra's algorithm/5

An example/3







Exercise SL10-4/1

Arbitrage is the practice of taking advantage of a price differential between two or more markets. An example of arbitrage is to use exchange rates to transform one unit of a currency into more than one unit of the same currency. Consider the exchange rates in the table below. For example, with the given exchange rates one can exchange USDs into Euros, Euros into GBPs and GBPs back into USDs. Indeed, for 100 USD one would get

$$100 \cdot (0.64 \cdot 0.79 \cdot 1.98) = 100 \cdot 1.001088 \approx 100.11 \text{ USD}.$$

	USD	GBP	CAD	EUR	AUD
USD	1.00	0.50	1.02	0.64	1.04
GBP	1.98	1.00	2.02	1.26	2.06
CAD	0.98	0.49	1.00	0.62	1.02
EUR	1.57	0.79	1.60	1.00	1.64
AUD	0.96	0.48	0.98	0.61	1.00

EXERCISE SL10-4/2

Model the above arbitrage problem as a general graph problem.

Notice that $a \cdot b > 1$ can be rewritten as $(1/a) \cdot (1/b) < 1$, which in turn can be rewritten as $\log(1/a) + \log(1/b) < 0$.

Give an efficient algorithm to determine whether or not there is a sequence of currencies $c_1, \ldots, c_k$ such that $xch[c_1, c_2] \cdot \ldots \cdot xch[c_{k-1}, c_k] \cdot xch[c_k, c_1] > 1$.

```
CanEarnMoney(G,s)
  for u ∈ G.V do u.dist = ∞
  s.dist = 0
  for i = 1 to |G.V|-1 do
      for (u,v) ∈ G.E do
          Relax(u,v, log(1/xch[u,v]))
  for (u,v) ∈ G.E do
      if (v.dist > u.dist + log(1/xch[u,v])) return true
  return FALSE
```

call: CanEarnMoney(G,s) for all needed start nodes s

SUMMARY

- **Minimum Spanning Trees** (MST):
  - Prim
    - vertex-based
    - priority queue
- **Single Source Shortest Path** (SSSP):
  - Bellman-Ford
    - negative weights
    - detection of negative weight cycles
  - Dijkstra
    - very popular
    - greedy
    - priority queue

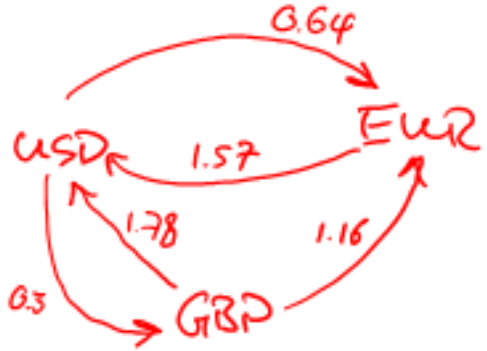
Informatics II

Syllabus and Exam
— SL11 —

Prof. Dr. Michael Böhlen

boehlen@ifi.uzh.ch

TABLE OF CONTENTS — SL11



1. Syllabus

2. The Informatics II exam

$a \cdot b \cdot c > 1$

$\frac{1}{a} \cdot \frac{1}{b} \cdot \frac{1}{c} < 1$

$\log\left(\frac{1}{a} \cdot \frac{1}{b} \cdot \frac{1}{c}\right) < \log 1$

$\log\frac{1}{a} + \log\frac{1}{b} + \log\frac{1}{c} < 0$

$\Rightarrow$ negative weight cycle

$\Rightarrow$ Bellman-Ford algo

SYLLABUS

1. Correctness and complexity of algorithms
2. Divide and conquer, recurrences
3. Sorting
4. Pointers, lists, sets, abstract data types
5. Trees, red-black trees
6. Hash tables
7. Dynamic programming
8. Graph algorithms

SYLLABUS

Data structures

- Simple data structures
  - array
  - linked lists
  - stacks
  - queues
  - trees
  - heaps
- Dictionaries
  - hash tables
  - binary search trees
  - red-black trees
- Graphs
  - memory representation
  - DAG

SYLLABUS

Algorithmic techniques

- Divide and Conquer
  - merge sort
  - quicksort
  - binary search
- Dynamic programming
  - matrix chain multiplication
  - longest common subsequence
  - coin changing
- Search (graph) algorithms
  - DFS, BFS
  - toplogical sorting
  - Prim, Dijkstra, Bellman
- Greedy algorithms
  - Prim, Dijkstra

SYLLABUS

Analyzing algorithms

- Recurrences
- Correctness of algorithms
  - pre/post-conditions
  - assertions
  - invariants
  - special case analysis
- Exact runtime
- Asymptotic complexity

SYLLABUS

Constructing algorithms

- representative example with input and output
- specification of input and output
- break up into small pieces
- ADT concept (e.g., modification of a parameter of a function)
- precise definitions of data structures and operations (signatures of procedures)

TABLE OF CONTENTS — SL11

1. Syllabus

2. The Informatics II exam

Exam

- The exam will be written, you will have 90 minutes.
- There is about half an hour per exercise.
- Auxiliary material: 1 A4 sheet (front and back side) with handwritten notes.
- In the exam you have to apply things you learned during the course.
- Try to solve all problems. Do not get lost/stuck in details.

Exam

Template for solution:

- understand and explain problem (often an example is best)
- write down plan for solution; reduce to known techniques
- come up with algorithms
- check special cases (ideally should not be exceptions)
- analyze algorithms if required
- make your solutions understandable

Exam

- Good solution = simple solution
- Being precise is important
- Solve step by step

To prepare for the exam:

- Focus on solving the exercises.
- Use textbook as reference with additional explanations and examples.
- Go to assistant, TA, tutors if you have questions.

Viel Erfolg!

Danke.