

# Algorithms and Data Structures

Ulrike von Luxburg

Winter 2013/14

Department of Computer Science, University of Hamburg

(Version as of November 8, 2013)

Contents will be updated continuously, see the webpage for the latest version.

# Table of contents

## Design and analysis of Algorithms: Introduction and Methods

Introduction .....	12
What is it about? .....	13
Why is it important for you? .....	20
Analysis of algorithms: The formal setup .....	25
What is an algorithm? .....	26
Appetizer: computing Fibonacci numbers .....	30
Pseudo-code .....	42
O-Notation .....	46
Running time analysis .....	55
Space complexity analysis .....	77
Appetizer: Multiplying two integers .....	79

# Table of contents (2)

Tool: Master theorem for solving simple recurrence relations ..... 88

## Basic Data Structures

Arrays and lists .....	107
Trees .....	115
Stack and queue .....	126
Heaps and priority queues .....	129
Heaps .....	130
Priority queue .....	161
Hashing .....	164
Basics .....	165
Some simple hash functions .....	175

## Sorting

Elementary Sorting Algorithms .....	184
Selection sort .....	187

# Table of contents (3)

Insertion sort .....	206
Bubble sort .....	213
Mergesort .....	218
Heap sort .....	232
Lower bound .....	247
Appetizer / Excursion: finding the median .....	267
Quicksort .....	296
Stable sorting algorithms .....	318
Sorting in linear time .....	331
Counting sort / Ksort .....	333
Radix sort .....	341
Bucket sort / uniform sort .....	346
Sorting: summary and state of the art .....	355
High-level summary: lessons about algorithm development .....	360

# Table of contents (4)

## Searching

Binary Search .....	366
Binary search trees: basics .....	385
AVL Trees .....	408
Outlook: more search trees .....	436
Excursion: Inverted index for document retrieval .....	441

## Graph algorithms

Recap: basic graph definitions .....	457
Data structures for representing graphs .....	471

Walking through graphs .....	479
Depth first search .....	481
Application: Connected components .....	487
Application: Strongly connected components .....	489
Application: Topological sort .....	502

# Table of contents (5)

Breadth first search .....	515
Application: shortest paths in unweighted graphs .....	522
Shortest path problems .....	534
Single Source Shortest Paths .....	541
Bellman-Ford algorithm .....	545
Dijkstra's algorithm .....	557
Generic labeling method .....	576
All pairs shortest paths on general graphs .....	581
Floyd-Warshall algorithm .....	583
Johnson's algorithm .....	594
Point to Point Shortest Paths .....	611
Bi-directional Dijkstra .....	613
Minimal spanning trees .....	628

# Table of contents (6)

## Hard problems

### Generic approaches to optimization

Greedy algorithms .....	638
Local search .....	639
Dynamic programming .....	640
Branch and bound .....	641

## Outlook

# Standard Literature

- ▶ The lecture closely follows the following book:  
K. Mehlhorn, P. Sanders: Algorithms and data structures: the basic toolbox. Springer, 2008.  
(25 Euros via Springer mycopy)  
German version of the book was supposed to be available this autumn, but got delayed until next year ☹:  
K. Mehlhorn, P. Sanders, M. Dietzfelbinger: Algorithmen und Datenstrukturen. Springer, 2014. 25 Euros.
- ▶ A book I like a lot because it conveys intuitions (not available in German):  
Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani.  
Algorithms. McGraw-Hill, 2008.  
pdf preprint available online.

## Standard Literature (2)

- ▶ The international standard text book on algorithms and data structures is the following:  
**Cormen, Leiserson, Rivest, Stein: Introduction to algorithms and data structures. 3rd edition. MIT Press, 2009.**  
The book also exists in German translation. Many copies available in the library, both in german and english.
- ▶ Another by-now-standard book on algorithms (english only):  
Jon Kleinberg, Eva Tardos: Algorithm Design. 2005

## Standard Literature (3)

For fun reading on algorithms (popular science style) I highly recommend:

- ▶ Vöcking et al., Taschenbuch der Algorithmen. Springer, 2008  
(English version: Algorithms unplugged, Springer 2011)
- ▶ John MacCormick: 9 Algorithms that changed the future.  
Princeton University Press, 2012

For preparing a job interview at one of the top companies:

- ▶ G. Laakmann: Cracking the coding interview. 2010.

# Design and analysis of Algorithms: Introduction and Methods

# Introduction

# What is it about?

# Algorithms occur in every days life, all the time!

- ▶ Railway:
  - ▶ Build a time table that satisfies the requirements.
  - ▶ Assign which physical train is supposed to serve which line of the schedule.
  - ▶ If one line is blocked because of an accident, quickly find an alternative schedule that minimizes the delays
- ▶ Stock market: auctions are performed by computers
  - ▶ Have to solve difficult combinatorial optimization problems.
  - ▶ Time is crucial (trading happens within milliseconds), and of course the quality of the solution is crucial.
- ▶ Biology:
  - ▶ Sequencing a gene!
  - ▶ Once you have the genetic code, analyze it! Compare it to others.

# Algorithms occur in every days life, all the time!

## (2)

- ▶ Internet:
  - ▶ Network protocols that are robust to failures.
  - ▶ Routing standards  $\leadsto$  questions about advantages/disadvantages of various shortest path algorithms
- ▶ Companies like google:
  - ▶ How can we index all internet pages?
  - ▶ How can we save storage capacity without giving up on redundancy?
  - ▶ How can we efficiently answer queries, say in less than a second?

# Computer science is not so much about programming ...

- ▶ Many people can program (or think that they can program). You don't need to be a computer scientist to be able to write Java Code.
- ▶ But to solve real-world problems, you need to know more than the syntax of Java. You cannot just sit down and implement solutions!
- ▶ What a computer **scientist** should be able to do:
  - ▶ Given a real world problem, "**abstract away**" all the messy, application-dependent details
  - ▶ Have a **toolbox** how to approach various problems
  - ▶ **Analyze**: How difficult is the problem? How good is my solution? Can I hope to get a better one?

# Computer science is not so much about programming ... (2)

- Once you come up with a **solution, describe it** precisely enough such that a programmer can sit down and implement it

# Algorithms are the heart of computer science!

Wikipedia: “*Computer science is the study of the theoretical foundations of information and computation and of practical techniques for their implementation and application in computer systems. Computer scientists invent algorithmic processes that create, describe, and transform information and formulate suitable abstractions to model complex systems.*”

- ▶ Want to solve particular problems with the help of a computer.
- ▶ Model the problem in a way that is abstract enough to be able to study it (that is, more abstract than actual Java code)
- ▶ Derive good solutions for it.
- ▶ Always keep in the back of our mind, that at the end of the day our solution is supposed to run on a computer.

# Algorithms are the heart of computer science! (2)

It is designing algorithms that makes computer science different from other disciplines like mathematics or electrical engineering.

Why is it important for *you*?

# Your later job!

- ▶ Most computer scientists are bound to run into algorithmic problems in their jobs
- ▶ You should at least be able to tell when this happens!
- ▶ You should have at least a basic understanding of how such problems might be handled (e.g., use an efficient data structure).
- ▶ This basic understanding is necessary to find reasonable solutions (even if you just search for literature)

# Example questions from a google job interview

- ▶ Given two binary trees:  $T_1$  (millions of nodes),  $T_2$  (hundreds of nodes). Design an algorithm to decide whether  $T_2$  is a subtree of  $T_1$ .
- ▶ You are given two sorted arrays,  $A$  and  $B$ .  $A$  has enough buffer at the end to hold  $B$ . Write a method to merge  $B$  directly into  $A$  in sorted order (without using any extra buffer).
- ▶ Design an algorithm to remove the duplicate characters in a string, without using any additional buffer (one or two additional variables are allowed, an extra copy of the array is not allowed).
- ▶ Given an image represented by an  $n \times n$  matrix, where each pixel in the image is 4 bytes, write a method to rotate the image by 90 degrees.

## Example questions from a google job interview (2)

- ▶ Assume you build a browser-game for tic-tac-toe that is going to be played online (on your server). How do you design a function to decide if someone has won? What if the board has size  $N \times N$  instead of  $3 \times 3$ ?
- ▶ Describe an algorithm to find the largest 1 million numbers in 1 billion numbers. Assume that the computer memory can hold all the 1 billion numbers.

# Formalities for this lecture

- ▶ Literatur
- ▶ Übungsgruppen
- ▶ Olat-Quizze
- ▶ Klausurzulassung
- ▶ Vorlesungszeiten

Was noch?

# Analysis of algorithms: The formal setup

# What is an algorithm?

# What is an algorithm?

Informal definition from wikipedia:

*An algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function. Starting from an initial state and initial input (perhaps empty), the instructions describe a computation that, when executed, proceeds through a finite number of well-defined successive states, eventually producing output and terminating at a final ending state.*

## What is an algorithm? (2)

It is pretty much impossible to formally define what an algorithm is.  
See the following references:

- ▶ Y. Moschovakis. What is an algorithm? In B. Engquist, W. Schmid (editors): Mathematics Unlimited, 2001 and beyond. Springer, 2001.
- ▶ A. Blass and N. Dershowitz and Y. Gurevich. When are two algorithms the same? Bulletin of Symbolic Logic, 2009.

We are going to come back to this issue at the end of the semester.

# Where does the name come from?

Book written by **Al Khwarizmi** in the 9th century: explains how to use the decimal number system.

This is the book that made the decimal system known to the western world.

Gave exact, precise, non-ambiguous descriptions about how to add, subtract, multiply, divide, take square roots, solve linear equations, etc.

Was translated into Latin in the 12th century under the title  
**“Algoritmi de numero Indorum”**

## Appetizer: computing Fibonacci numbers

Literature: Dasgupta Chapter 0

# Fibonacci numbers

Can you recall the definition of Fibonacci numbers?

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2 \end{cases}$$

Gives the sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Fibonacci numbers grow very fast. It is easy to prove that for  $n \geq 6$  we have  $2^{n/2} \leq F_n \leq 2^n$ .

## Fibonacci numbers (2)

We are now going to look at various algorithms to compute  $F_n$  and argue about how good they are.

We keep it informal in this section, take it as an appetizer.

# Fibonacci numbers (3)

EVERYBODY:

- ▶ CAN YOU TRY TO WRITE DOWN AN ALGORITHM TO COMPUTE THE FIBONACCI NUMBERS?
- ▶ CAN YOU ALSO COME UP WITH A DIFFERENT ALGORITHM FOR COMPUTING FIBONACCI NUMBERS?

# Algorithm 1: naive loop with lookup

FibLoop( $n$ )

- 1 Create Array  $A(0, \dots, n)$
- 2  $A(0) = 0; A(1) = 1$
- 3 **for**  $i=2, \dots, n$
- 4      $A(i) = A(i - 1) + A(i - 2)$
- 5 Return  $A(n)$

## Algorithm 2: recursive

FibRecursive( $n$ )

```
1 if  $n=0$ 
2   Return 0
3 if  $n=1$ 
4   Return 1
5 Return FibRecursive( $n-1$ ) + FibRecursive( $n-2$ )
```

# Running times of the Fibonacci algorithms

WHAT DO YOU THINK, WHICH ALGORITHM IS GOING TO TAKE LONGER? WHY?

HOW WOULD YOU ARGUE FORMALLY?

## Running times of the Fibonacci algorithms (2)

How many “elementary operations”  $T(n)$  do we have to perform in `FibLoop(n)`?

- ▶ Line 1: Allocate an array of length  $n$ : depending on how exactly this allocation procedure works, it might take either 1 elementary operation (just give a pointer to the beginning of the array) or up to  $n$  elementary operations
- ▶ Line 2: 2 elementary operations
- ▶ In the loop, we have to add two numbers.
  - ▶ These two numbers can be as large as  $F_{n-1}$ .
  - ▶ We have already seen that  $2^{n/2} \leq F_n \leq 2^n$ . So in binary representation,  $F_n$  is going to use at most  $n$  bits.
  - ▶ To add two numbers with  $n$  bits needs on the order of  $n$  elementary operations (just take this as a fact for now).
- ▶ The loop is going to be executed  $n - 1$  times.

## Running times of the Fibonacci algorithms (3)

So all in all we end up with  $T(n) := 1 + 2 + (n - 1) \cdot n \approx n^2$  elementary operations.

# Running times of the Fibonacci algorithms (4)

How many “elementary operations” do we have to perform in **FibRecursive**?

- ▶ Lines 1, 2, 3, 4 each need 1 elementary operation.
- ▶ Line 5 is one elementary operation (the addition), plus the time needed in the recursive calls. We ignore the elementary operation and just consider a lower bound:

We end up with the formula

$$T(n) \geq T(n-1) + T(n-2) + 4$$

Comparing with the definition of  $F_n$  we see immediately that  
 $T(n) \geq F_n$ .

As we have already seen that for  $n \geq 6$  we have  $F_n \geq 2^{n/2}$ , we finally obtain:

$$T(n) \geq 2^{n/2}$$

# Running times of the Fibonacci algorithms (5)

Comparing the running times:

- ▶ The running time of FibRecursive is exponential in  $n$ . This means that even for moderate  $n$  the running time is too large for any computer you might think of.
- ▶ The running time of FibLoop is quadratic in  $n$ . This is ok for reasonable large  $n$  (even though not great). At least it is “polynomial”.
- ▶ Do you think that there exists a faster algorithm than FibLoop? The answer is yes, and you will see this in your exercises. ☺

# Take home message

- ▶ Even for very simple problems, slightly different implementations can lead to huge performance differences!!!
- ▶ The obvious implementations are often quite slow.
- ▶ To obtain fast implementations is often not so obvious.

The main purpose of this lecture is

- ▶ to show you some of the fast, elegant algorithms for standard problems
- ▶ to give you a toolbox to be able to analyze algorithms and figure out whether they are “good”
- ▶ to teach you “algorithmic thinking”: this is what you need to develop new algorithms yourself

# Pseudo-code

# Pseudo-Code

In order to argue about different algorithms, we want to have a language to “write down” an algorithm:

- ▶ compact and informal high-level description of the operating principle of an algorithm
- ▶ It uses the structural conventions of a programming language, but is **intended for human reading** rather than machine reading.

We use what is called “pseudo-code”:

- ▶ use the standard constructions of programming languages (if, for, while, ... )
- ▶ But we do not care about particular syntax details such as semicolons, differences between = and ==, and so on.
- ▶ Often, we are more high-level than a programming language. For example, we might state:

# Pseudo-Code (2)

- ▶ “allocate an array of length  $n$ ”
- ▶ “Sort the entries of the array in increasing order”
- ▶ Many authors use different conventions and syntax for pseudo-code, it is not really important which one we use.

Conventions for the assignments:

- ▶ We are not going to insist on a particular syntax. Use java-like or C-like syntax or whatever you prefer.
- ▶ If you write pseudo-code, use common sense — your code has to be understandable by any “educated programmer” (and, as a matter of fact, by your Übungsleiter ... )
- ▶ In particular, do not use syntax-specific tricks or shortcuts such as:     $a=(b==4)$
- ▶ Always use indentations to show the block structure

# Pseudo-Code (3)

Example: search through an array  $A$  to check whether it contains a particular element  $a_0$

**NaiveSearch( $A, a_0$ )**

```
1  it = 0
2  foundit = false
3  while ( (NOT foundit) AND (it < length( $A$ ))
4      it = it + 1
5      if  $A(it) = a_0$ 
6          foundit = true
7          position = it
8      if foundit
9          Return position
10     else
11         Return NaN
```

# O-Notation

## Literature:

- ▶ Mehlhorn/Sanders Section 2.1
- ▶ Cormen Section I.3

# The growth of functions

Running times of algorithms will depend on the “size of the problem”:

- ▶ It makes a difference whether we have to sort 10 items or  $10^3$  items:

Ultimately, we are not so much interested in the running time on small instances:

- ▶ Even if we are very naive, sorting 10 numbers will not take long.

We want to know how the running times behave if the “size”  $n$  of the input “gets large”.

To this end, we want to have an easy but precise tool to discuss the growth of functions in general.

# The growth of functions (2)

Example:

Assume we have three algorithms to solve a particular problem.  
They perform the following number of elementary operations:

- ▶ Algorithm 1 needs  $17n^3 + 3n^2 - 7$  operations
- ▶ Algorithm 2 needs  $4n^3 + 100n^2$  operations
- ▶ Algorithm 3 needs  $n + 10000$  operations
- ▶ Algorithm 4 needs  $2^n$  operations

We want to solve an instance of the problem of size  $n = 10^5$ .

EVERYBODY: WHICH ONE IS BEST? HOW LARGE ARE THE DIFFERENCES IN RUNNING TIME?

# The growth of functions (3)

Punchline:

We only care for asymptotic growth rates, the “order of magnitude”. In particular,

- ▶ we only want to look at “the largest term” in the expressions above.
- ▶ we want to ignore all constants

Now we want a mathematically precise formalism for doing this:  
~ O-notation.

# O-Notation: Definitions

To analyze the running times, we are going to use the  $O$ -Notation (sometimes called Landau notation). This is a tool to describe the asymptotic growth of functions.

In this section we always assume that all functions are non-negative.

$f \in O(g)$   $f$  is of order at most  $g$ :

$$\exists c > 0 \exists n_0 > 0 \forall n > n_0 : 0 < f(n) \leq cg(n)$$
$$\iff 0 \leq \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

$f \in o(g)$   $f$  is of order strictly smaller than  $g$ :

$$\forall c > 0 \exists n_0 > 0 \forall n > n_0 : 0 < f(n) < cg(n)$$
$$\iff 0 \leq \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

# O-Notation: Definitions (2)

$f \in \Omega(g)$   $f$  is of order at least  $g$ :

$$\exists c > 0 \exists n_0 > 0 \forall n > n_0 : f(n) \geq cg(n) > 0$$

$$\iff 0 < \liminf_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq \infty$$

$$\iff g \in O(f)$$

$f \in \omega(g)$   $f$  is of order strictly larger than  $g$ :

$$\forall c > 0 \exists n_0 > 0 \forall n > n_0 : f(n) > cg(n) > 0$$

$$\iff \liminf_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

$$\iff g \in o(f)$$

$f \in \Theta(g)$   $f$  has exactly the same order as  $g$ :

$$\exists c_1, c_2 > 0 \exists n_0 > 0 \forall n > n_0 : c_1 g(n) \leq f(n) \leq c_2 g(n)$$

$$\iff 0 < \liminf_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq \limsup_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

$$\iff f \in O(g) \text{ and } f \in \Omega(g)$$

# O-Notation: Examples

- ▶ Consider  $f(n) = n^2 + 7$ . Then:  $f \in O(n^{10})$ ;  $f \in o(n^{10})$ ;  $f \in \omega(\log(n))$ ;  $f \in \Theta(n^2)$ ;
- ▶ Consider  $f(n) = n^2 + 5n$ . Then  $f \in O(n^2)$ , but  $f \notin o(n^2)$ . Instead  $f \in \Omega(n^2)$ , but  $f \notin \omega(n^2)$ . In particular,  $f \in \Theta(n^2)$ .

# O-Notation: Examples (2)

Important special cases:

- ▶ If  $f$  is any polynomial of degree  $d$ , then  $f \in \Theta(n^d)$ .
- ▶ Any polynomial function  $f$  is in  $o(\exp(n))$  and  $\omega(\log(n))$ .
- ▶ Observe: if we use logarithms, it does not matter what basis of the logarithm we choose:  $\log_{b_1}(n) \in \Theta(\log_{b_2}(n))$ .

Reason:

$$\log_b x = \frac{\log_a x}{\log_a b}$$

- ▶ Any function that is bounded between constants  $a > 0$  and  $b > 0$ , where  $a$  and  $b$  are independent of  $n$ , is said to be  $O(1)$ .

# O-Notation: Examples (3)

Some rules:

- ▶  $f \in O(g_1 + g_2)$  and  $g_1 \in O(g_2) \implies f \in O(g_2)$ .
- ▶  $f_1 \in O(g_1)$  and  $f_2 \in O(g_2) \implies f_1 + f_2 \in O(g_1 + g_2)$
- ▶  $f \in g_1 \cdot O(g_2) \implies f \in O(g_1 g_2)$
- ▶ Note: for any constant  $c > 0$ ,  $O(cg)$  is identical to  $O(g)$ .
- ▶  $f \in O(g_1)$ ,  $g_1 \in O(g_2) \implies f \in O(g_2)$
- ▶ Abuse of notation: we often write  $f = O(g)$ , but we mean  $f \in O(g)$ .

# Running time analysis

Literature:

- ▶ Mehlhorn/Sanders Sec. 2

# Physical running time

We could simply go ahead, run an algorithm on our computer, and measure its running time.

IS THIS A GOOD IDEA?

# Physical running time

We could simply go ahead, run an algorithm on our computer, and measure its running time.

IS THIS A GOOD IDEA?

NO, physical running time is not a good model:

- ▶ depends on the computer we are using
- ▶ depends on the particular input we are using

Instead we are going to do the following:

- ▶ Introduce a simplified “model” of a computer
- ▶ Count the number of “elementary operations” of this computer

# Simplified machine model: RAM machine

Random access machine (RAM) model:

- ▶ Sequential processing
- ▶ Uniform memory (all access to memory takes the same amount of time)

The machine can perform the following actions as “elementary operations” in one unit of time:

- ▶ Load one bit from the memory
- ▶ Write one bit of memory
- ▶ Elementary arithmetics: for numbers of bounded length, we assume that we can add, subtract, multiply, divide them in constant time
- ▶ Similarly, each logical operation (and, or, not) is an elementary operation.

## Simplified machine model: RAM machine (2)

Some care is needed to deal with arithmetic operations of “long numbers”:

- ▶ As long as numbers are long enough to fit into, say, 64 bits, we treat arithmetic operations as constant time operations.
- ▶ However, in some algorithms the numbers grow as the instances get larger (this was the case in FibLoop). Then we can no longer treat the arithmetic operations as elementary operation.
- ▶ In this case, we need to consider the actual costs of doing such an operation.

WHAT DO YOU THINK IS THE COST OF ADDING TWO NUMBERS OF  $n$  BITS?

WHAT ABOUT MULTIPLYING THEM?

# Simplified machine model: RAM machine (3)

Common sense tells us:

- ▶ The cost to add two numbers depends on the numbers of bits we have to touch. To add two numbers of  $u$  and  $v$  bits is going to be in  $O(\max(u, v))$ , that is it is linear in the number of bits.
- ▶ For (naive) multiplication, the cost is quadratic in the number of bits:

# Simplified machine model: RAM machine (4)

Basic Integer Multiplication:

$$\begin{array}{r} x_1 \ x_2 \ x_3 \\ 3 \ 7 \ 5 \\ \cdot \ 2 \ 4 \ 6 \\ \hline \end{array}$$

$$\begin{array}{rl} y_1 \cdot x_3 & 10 \\ y_1 \cdot x_2 & 14 \\ y_1 \cdot x_1 & 6 \\ y_2 \cdot x_3 & 20 \\ y_2 \cdot x_2 & 28 \\ y_2 \cdot x_1 & 12 \\ y_3 \cdot x_3 & 30 \\ y_3 \cdot x_2 & 42 \\ y_3 \cdot x_1 & 18 \\ \hline & 92250 \end{array}$$

See later for a formal derivation of these results.

# Counting RAM elementary operations, examples

How many elementary operations does it take to do the following operations:

- ▶ add two standard integers:

# Counting RAM elementary operations, examples

How many elementary operations does it take to do the following operations:

- ▶ add two standard integers:  $O(1)$
- ▶ Write an  $m \times n$  matrix to memory:

# Counting RAM elementary operations, examples

How many elementary operations does it take to do the following operations:

- ▶ add two standard integers:  $O(1)$
- ▶ Write an  $m \times n$  matrix to memory:  $O(n \times m)$
- ▶ Read  $n$  strings of length 10:

# Counting RAM elementary operations, examples

How many elementary operations does it take to do the following operations:

- ▶ add two standard integers:  $O(1)$
- ▶ Write an  $m \times n$  matrix to memory:  $O(n \times m)$
- ▶ Read  $n$  strings of length 10:  $O(n)$
- ▶ Perform the operation  $a_1 + a_2 \cdot a_3$  if we know that  $a_1$  has  $n^2$  bits,  $a_2$  has  $\log n$  bits,  $a_3$  has  $n$  bits:

# Counting RAM elementary operations, examples

How many elementary operations does it take to do the following operations:

- ▶ add two standard integers:  $O(1)$
- ▶ Write an  $m \times n$  matrix to memory:  $O(n \times m)$
- ▶ Read  $n$  strings of length 10:  $O(n)$
- ▶ Perform the operation  $a_1 + a_2 \cdot a_3$  if we know that  $a_1$  has  $n^2$  bits,  $a_2$  has  $\log n$  bits,  $a_3$  has  $n$  bits:
  - ▶ Computing  $a_2 \cdot a_3$  takes  $O(n \cdot \log n)$  time.
  - ▶ The resulting number is going to have of the order  $O(n \cdot \log n)$  bits.
  - ▶ To add a number with  $n^2$  bits to a number of  $n \log n$  bits is going to cost  $O(n^2)$  time.

So overall, this operation takes  $O(n^2)$ .

# How to measure the “length of an input”

We want to compute the **running time as a function of the size  $n$  of the input**.

- ▶ Sorting  $n$  standard integers  $\leadsto$  input size is  $n$
- ▶ Comparing two strings of variable lengths  $n_1$  and  $n_2$   $\leadsto$  input size is the sum of the lengths of the two strings. Because we don't care about constants, this is of the same order as the length of the longer of the two strings.
- ▶ Compare  $m$  strings with length at most  $n$   $\leadsto$  input size  $m \cdot n$
- ▶ Finding a shortest path in a graph  $\leadsto$  input size is the number of vertices and edges in the graph

If this sounds a bit confusing, don't worry, this is going to become clear during the lecture.

# What instance to look at?

Depending on the instance we look at, an algorithm might take long or not so long:

- ▶ If we want to test whether two sequences are identical, and they already disagree at the first element we are looking at, then we are done. But if they disagree only in the last element, this will take us much longer to figure out.
- ▶ If our task is to sort a sequence, and the sequence is already sorted, we are done already.
- ▶ To compute the shortest path in a tree is much simpler than computing the shortest path in a general graph.

But we want a statement about the running time that is reasonably general.

ANY IDEA WHAT WE COULD DO?

# Worst case running time: definition

Among all possible instances of size  $n$ , what is the longest time it might take the algorithm to finish?

Formally:

- ▶ Let  $\mathcal{I}_n$  denote the space of all instances of length  $n$
- ▶ Denote by  $T(I_n)$  the running time of the algorithm on instance  $I_n$
- ▶ Then the worst case running time for input of size  $n$  is defined as

$$T_{wc}(n) := \max\{ T(I_n) \mid I_n \in \mathcal{I}_n \}$$

# Worst case running time: definition (2)

Example  $\text{NaiveSearch}(A, a_0)$ :

- ▶  $n :=$  number of elements in the array
- ▶ Initialization = 2 units of time
- ▶ In the worst case we have to execute  $n$  while loops
- ▶ Inside the while loop, we have a constant number of operations [if we count exactly, line 3 is 3 operations, line 4 is 2 operations (one addition, one access to storage), lines 6 and 7 one operation each].
- ▶ after we finish the while loop, we have 2 more operations
- ▶ So the overall worst case running time is:  
$$2 + n \cdot 7 + 2 \in O(n).$$

## Worst case running time: definition (3)

ASSUME YOU WANT TO GET A LOWER / UPPER BOUND ON THE WORST CASE RUNNING TIME. WHAT DO YOU HAVE TO DO?

## Worst case running time: definition (4)

- ▶ Lower bounds are easy: If you find **one particular instance**  $I_n$  on which the running time is at least  $T_{lower}$ , then you can conclude that  $T_{wc}(n) \geq T_{lower}$ .
- ▶ Upper bounds: If you know that for **all instances**  $I_n \in \mathcal{I}_n$ , the running time is at most  $T_{upper}$ , then you can conclude that  $T_{wc}(n) \leq T_{upper}$ .

# Average case running time

Observation:

- ▶ While worst case running times always give an upper bound, they are often much too conservative:
- ▶ Some algorithms behave very well in practice even though they have a bad worst case running time.
- ▶ The reason is that the “normal” instances we have to solve in every day life are often “easy”, whereas the worst case running time only occurs for very un-natural, hand-constructed instances.

To circumvent the problem we introduce the average case running time:

## Average case running time (2)

On average over all instances of size  $n$ , what is the time it takes algorithm  $A$  to finish?

Formally:

- ▶ Let  $\mathcal{I}_n$  denote the space of all instances of length  $n$
- ▶ Denote by  $T(I_n)$  the running time of the algorithm on instance  $I_n$
- ▶ Then the average case running time for input of size  $n$  is defined as

$$T_{av}(n) := \frac{1}{|\mathcal{I}_n|} \sum_{I_n \in \mathcal{I}_n} T(I_n)$$

## Average case running time (3)

There are some subtleties about the fact that the average time analysis heavily depends on “how often” each instance is supposed to occur! We skip details, because this lecture is mainly concerned with worst case analysis.

# Special cases

We say that the running time of an algorithm is ...

- ▶ linear  $\sim \Theta(n)$
- ▶ sublinear  $\sim o(n)$
- ▶ superlinear  $\sim \omega(n)$
- ▶ polynomial  $\sim \Theta(n^a)$  for some constant  $a$  independent of  $n$
- ▶ exponential  $\Theta(2^n)$

# Space complexity analysis

# Space complexity

- ▶ We also need to know how much storage capacity an algorithm uses.
- ▶ One model “unit” of storage for:
  - ▶ A letter
  - ▶ A logical bit
  - ▶ A number of fixed, bounded length
- ▶ We usually look at worst case behavior.

For many problems, there is a space-time tradeoff:

- ▶ Some algorithms are very fast, but need a lot of space.
- ▶ Other algorithms are very slow, but need nearly no storage.

Depending on the application, one or the other case might be better.

DOES ANYBODY KNOW AN EXAMPLE OF SUCH A BEHAVIOR?

# Appetizer: Multiplying two integers

Literature:

- ▶ Dasgupta 2.1
- ▶ Mehlhorn 1.3-1.5

# Problem statement

We consider the following, very simple problem:

Given two integers  $x$  and  $y$ , compute their product  $x \cdot y$ .

For simplicity, let us assume that both  $x$  and  $y$  can be represented as binary numbers of length  $n$ , and that  $n$  is a power of 2.

# Problem statement (2)

Naive method (as in school) takes  $O(n^2)$ :

Basic integer Multiplication:

$$\begin{array}{r} x_1 \ x_2 \ x_3 \\ 3 \ 7 \ 5 \\ \cdot \ 2 \ 4 \ 6 \\ \hline \end{array}$$

$$\begin{array}{rl} y_1 \cdot x_3 & 1 \ 0 \\ y_1 \cdot x_2 & 1 \ 4 \\ y_1 \cdot x_1 & 6 \\ y_2 \cdot x_3 & 2 \ 0 \\ y_2 \cdot x_2 & 2 \ 8 \\ y_2 \cdot x_1 & 1 \ 2 \\ y_3 \cdot x_3 & 3 \ 0 \\ y_3 \cdot x_2 & 4 \ 2 \\ y_3 \cdot x_1 & 1 \ 8 \\ \hline & 9 \ 2 \ 2 \ 5 \ 0 \end{array}$$

# First divide and conquer solution

Idea: try a recursive algorithm ( $\leadsto$  divide and conquer):

- ▶ Split  $x$  and  $y$  in their left and right parts. We have:

$$x = 2^{n/2}x_l + x_r$$

$$y = 2^{n/2}y_l + y_r$$

$$\begin{aligned} x &= \boxed{\textcolor{red}{10110011}} \textcolor{green}{\boxed{00101111}} \\ y &= \boxed{\textcolor{red}{11101011}} \textcolor{green}{\boxed{00011110}} \\ &\quad \textcolor{red}{x_l} \qquad \qquad \textcolor{green}{x_r} \\ &\quad \textcolor{red}{y_l} \qquad \qquad \textcolor{green}{y_r} \end{aligned}$$

- ▶ So we can write:

$$x \cdot y = 2^n x_l y_l + 2^{n/2}(x_l y_r + x_r y_l) + x_r y_r$$

We now have **4 subproblems**: multiply integers of length  $n/2$

# First divide and conquer solution (2)

## Running time for applying this scheme once::

- ▶ Have to solve 4 subproblems, each of size  $n/2$ .
- ▶ Assume we solve each of the subproblems by the standard algorithm. Then each subproblem needs time  $(n/2)^2 = n^2/4$
- ▶ So all in all we obtain running time  $4n^2/4 = n^2$ .

## Running time for applying the scheme recursively:

- ▶ Write  $T(n)$  for the running time on integers of length  $n$
- ▶ The recursive call leads to the following recurrence relation:

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

- ▶ By the master theorem (see later), the solution to this equation is  $T(n) = O(n^2)$ .

So the recursive approach did not improve the running time either.



# Better divide and conquer solution

Simple, but powerful observation:

- We have:

$$(x_l y_r + x_r y_l) = (x_l + x_r)(y_l + y_r) - x_l y_l - x_r y_r$$

- This gives:

$$\begin{aligned}x \cdot y &= 2^n x_l y_l + 2^{n/2} (x_l y_r + x_r y_l) + x_r y_r \\&= 2^n x_l y_l + 2^{n/2} ((x_l + x_r)(y_l + y_r) - x_l y_l - x_r y_r) + x_r y_r \\&= 2^{n/2} \textcolor{red}{x_l y_l} + (1 - 2^{n/2}) \textcolor{green}{x_r y_r} + 2^{n/2} (\textcolor{blue}{x_l + x_r})(\textcolor{blue}{y_l + y_r})\end{aligned}$$

Thus, instead of computing 4 multiplications of  $n/2$  bits each, we only need 3 of them!!!

## Better divide and conquer solution (2)

**Running time if we apply this principle once:**

- ▶ Have to solve 3 subproblems, each of size  $n/2$ .
- ▶ Assume we solve each of the subproblems by the standard algorithm. Then each subproblem needs time  $(n/2)^2 = n^2/4$
- ▶ So all in all we obtain running time  $3n^2/4 = (3/4)n^2$ .

Looks like we gain a constant factor (which is already good), but it would vanish in the  $O$ -Notation, which remains  $O(n^2)$ .

## Better divide and conquer solution (3)

**Running time if we apply this principle recursively:**

The resulting algorithm satisfies the recurrence relation

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

which has the solution  $T(n) = O(n^{\log_2 3}) \approx O(n^{1.59})$  due to the master theorem.

Thus we found an algorithm that is substantially faster than the naive approach ☺

# High-level comments

- ▶ Divide and conquer sometimes helps, sometimes not. But it is always worth a trial.
- ▶ An improvement by a constant factor, gained in each level of a recursion tree, can “sum up” to something quite relevant such as decreasing the exponent of  $n$ .

## Tool: Master theorem for solving simple recurrence relations

Literature:

Dasgupta 2.2; Cormen 4.5; Mehlhorn 2.7

# Divide and conquer and recurrence relations

We are going to use the **divide and conquer principle**:

- ▶ Given a problem of size  $n$
- ▶ Decompose it into several problems of smaller size and solve these problems
- ▶ From the solutions, reconstruct the overall solution

To **analyze the running time**, we will always follow the same principle.

- ▶ Denote by  $T(n)$  the running time of the algorithm on an instance of size  $n$ .
- ▶ Let  $a$  be the number of subproblems,  $m = n/b$  the size of the subproblems, and assume we need  $O(n^d)$  time to combine the smaller solutions to the final one.

## Divide and conquer and recurrence relations (2)

- ▶ Then the running time satisfies the recurrence relation

$$T(n) = aT(\lceil n/b \rceil) + O(n^d)$$

# The master theorem for solving recurrences

## Theorem 1 (Master theorem)

If  $T(n) = aT(\lceil n/b \rceil) + O(n^d)$  for some constants  $a > 0$ ,  $b > 1$ , and  $d \geq 0$ , then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

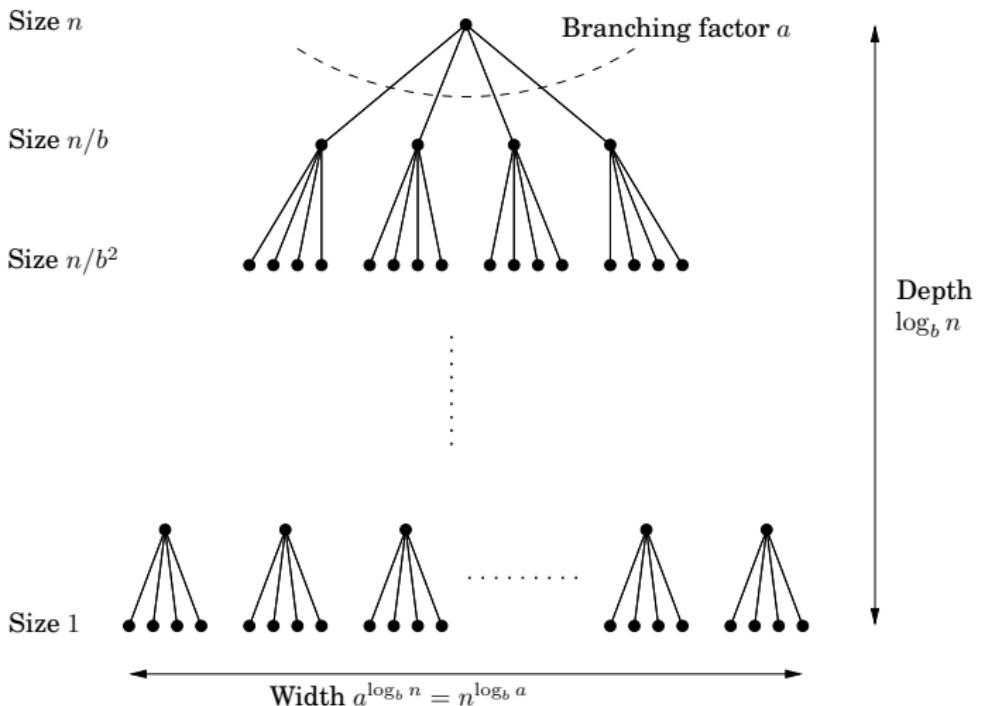
In this context, recall how to change the base of a logarithm:

$$\begin{aligned} \log_b x &= \log_b (a^{\log_a x}) = \log_a x \log_b(a) \\ \implies \log_a x &= \frac{\log_b x}{\log_b a} \end{aligned}$$

# The master theorem for solving recurrences (2)

**Proof.** The proof of the Master theorem is surprisingly simple. It boils down to analyze the following recursion tree:

# The master theorem for solving recurrences (3)



# The master theorem for solving recurrences (4)

(Remark: In the figure, “depth” is what we usually call “height” of the tree)

# The master theorem for solving recurrences (5)

- ▶ **Height of the tree:** at each level, the problem gets smaller by a factor of  $1/b$ . Thus we reach the bottom of the tree when  $n/b^{\text{height}} = 1$ , that is the height =  $\log_b n$ .

# The master theorem for solving recurrences (6)

- ▶ **Level  $k$  of the tree, for all intermediate values of  $k$ :**
  - ▶ Level  $k$  of the tree is made up of  $a^k$  subproblems
  - ▶ Each of these subproblems has size  $n_k := n/b^k$
  - ▶ Each of these problems is solved by calling further subproblems (no work required in level  $k$ ) and by then recombining these solutions (this is the work done in level  $k$ ).  
For each of these problems, this recombination takes time  $O((n_k)^d) = O((n/b^k)^d)$ .
  - ▶ So the total work to be done at level  $k$  is
- ▶ Note that  $k$  is not a constant, it is a number that depends on  $n$ . For this reason, we cannot simply ignore the term  $(a/b^d)^k$  (we are going to sum over  $k$  in the next step).

# The master theorem for solving recurrences (7)

- **Bottom level of the tree:** The bottom level consists of  $a^{\text{depth}} = a^{\log_b n} = n^{\log_b a}$  problems of constant size (say, size 1). Solving each of these problems requires  $O(1)$  time, so the time required by the algorithm in the bottom level of the tree is  $O(n^{\log_b a})$ .

Altogether, the time to go work through the whole tree is

$$\begin{aligned}& \left( \sum_{k=0}^{\text{depth}-1} O\left(n^d \cdot \left(\frac{a}{b^d}\right)^k\right) \right) + O(n^{\log_b a}) \\&= O(n^d) \cdot O\left(\sum_{k=0}^{\log_b n - 1} \left(\frac{a}{b^d}\right)^k\right) + O(n^{\log_b a})\end{aligned}$$

The rest of the proof just evaluates this expression:

# The master theorem for solving recurrences (8)

First consider the red sum.

- ▶ It is a geometric series of the form  $\sum_{k=0}^m c^k$ .
- ▶ It is straight forward to see that

$$O\left(\sum_{k=0}^m c^k\right) = \begin{cases} O(1) & \text{if } c < 1 \text{ (because sum} < 1/(1 - c)\text{)} \\ O(m) & \text{if } c = 1 \text{ (because then sum} = m\text{)} \\ O(c^m) & \text{if } c > 1 \text{ (proof by simple induction)} \end{cases}$$

In our setting, this now translates into the following three cases:

# The master theorem for solving recurrences (9)

- ▶ **Case 1.**  $c < 1 \iff a/b^d < 1 \iff d > \log_b a$ .

Then we are left with

$$O(n^d) \cdot O(1) + O(\underbrace{n^{\log_b a}}_{< n^d}) = O(n^d)$$

- ▶ **Case 2.**  $c = 1 \iff a/b^d = 1 \iff d = \log_b a$ .

We are left with

$$O(n^d) \cdot O(\log_b n) + O(\underbrace{n^{\log_b a}}_{= n^d}) = O(n^d \log n)$$

# The master theorem for solving recurrences (10)

- **Case 3.**  $c > 1 \iff a/b^d > 1 \iff d < \log_b a$ .

We are left with

$$O(n^d) \cdot O\left(\underbrace{\left(\frac{a}{b^d}\right)^{\log_b n}}_{(*)}\right) + O(n^{\log_b a})$$

$$(*) = \frac{a^{\log_b n}}{(b^{\log_b n})^d} = \frac{a^{\log_b n}}{n^d} = \frac{a^{(\log_a n)(\log_b a)}}{n^d} = \frac{n^{\log_b a}}{n^d}$$

Combining the terms leads to

$$n^d \cdot \frac{n^{\log_b a}}{n^d} + n^{\log_b a} = O(n^{\log_b a})$$

# Complex operations with O-Notation

Issues that came up in the proof:

- ▶ When we sum over something that depends on  $n$ , never throw away constants right away.
- ▶ Make sure it is clear what is a constant and what not (in the master theorem,  $a$ ,  $b$ ,  $d$  were constants, but  $k$  is not a constant).
- ▶ When you look at a sum and the index runs over an index set that depends on  $n$ , be careful as well.

If in doubt, then do it along the definition. Replace the individual  $O$ -s by the expressions from the definition:

- ▶  $O(f)$ : there exists a constant  $C$  such that ...
- ▶ carry along all constants, and just at the very end evaluate the  $O$ -notation.

# Examples

EXAMPLE ( $\leadsto$  used for the analysis of integer multiplication):

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

## Examples (2)

SOLUTION:

- ▶ Have  $a = 4$ ,  $b = 2$ ,  $d = 1$ ,
- ▶ In particular,  $\log_b a = \log_2 4 = 2 > 1 = d$ .
- ▶ Thus, by the third case of the master theorem we get

$$T_n = O(n^{\log_b a}) = O(n^2)$$

# Examples (3)

EXAMPLE:

$$T(n) = T(n/2) + O(1)$$

GIVE IT A TRY, EVERYBODY

## Examples (4)

SOLUTION:

- ▶ Recurrence equation is  $T(n) = T(n/2) + O(1)$ .
- ▶ In the notation of the Master theorem we have  $a = 1$ ,  $b = 2$ ,  $d = 0$ . In particular  $\log_b a = \log_2 1 = 0 = d$ .
- ▶ So we are in the second case of the theorem. The running time is  $O(n^d \log n) = O(n^0 \log n) = O(\log n)$ .

This is in fact the running time of binary search, see later ...

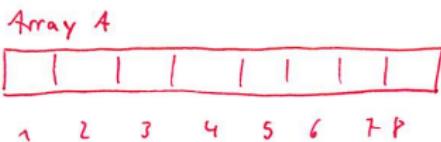
# Basic Data Structures

# Arrays and lists

# Array

Everybody knows arrays already:

- ▶ An array of length  $n$  is an arrangement of  $n$  objects of the same type in equally spaced addresses in memory.
- ▶ This is the most basic data structure one can think of.



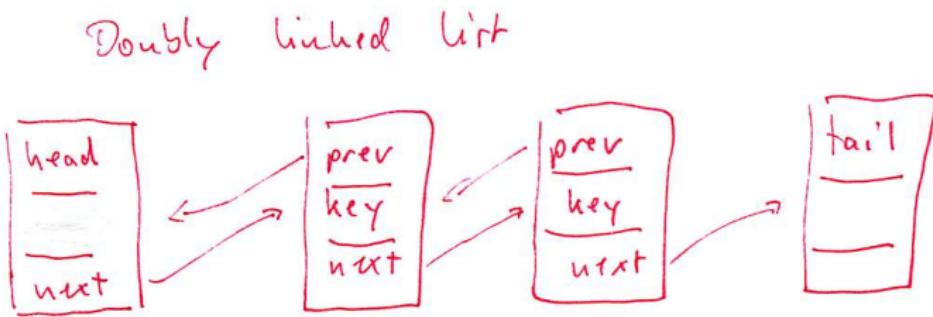
Standard operations on an array:

- ▶ Read / write an element of the array. Happens in constant time, no matter which element we want to access (reason: we know their address in memory).

Disadvantage of arrays: we need to allocate them in advance.

# Doubly linked list

- Each list element consists of a key value, a “previous” pointer and a “next” pointer



XXX IN THE FIGURE, A PREV LINK IS MISSING AT TAIL XXX

# Doubly linked list (2)

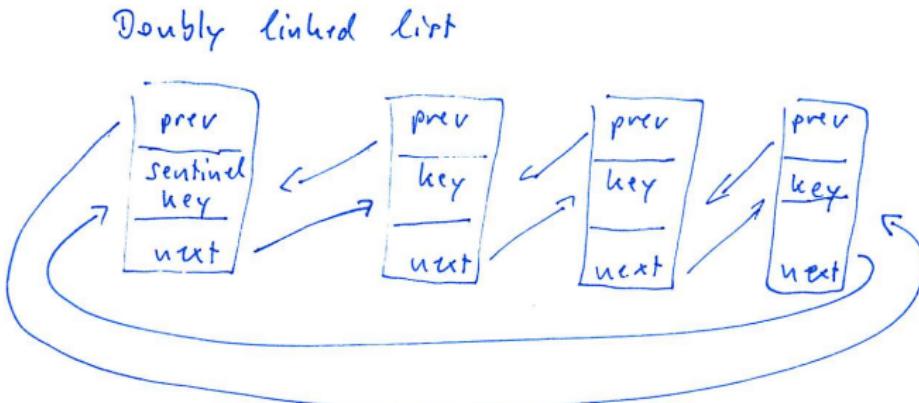
Basic operations:

- ▶ Insert element  $x$  at after element  $e$  in the list. When you are currently sitting on  $e$ , this is possible in  $O(1)$  (just change two pointers).
- ▶ Delete an element from the list. Similarly to inserting, possible in  $O(1)$
- ▶ Search element with a particular key value  $k$ , needs  $O(n)$  (need to walk through the whole list)
- ▶ Delete a whole sublist,  $O(1)$  no matter how large the sublist is (just change two pointers)
- ▶ Insert an existing second list after a particular element in the first list, also possible in  $O(1)$  (independent of the lengths of the lists!)

# Doubly linked list (3)

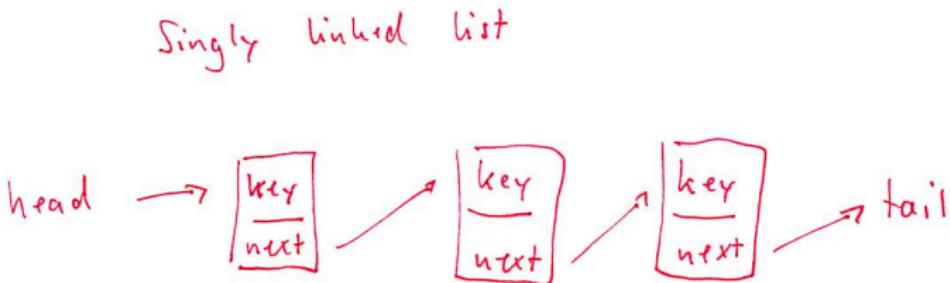
Two different ways to implement them:

- ▶ Using a “head” and “tail” element, as in the figure above
- ▶ Link tail to head, but insert a “sentinel” element (German: Wächter)



# Singly linked list

As before, but each element only contains a pointer to the next element, not to the previous one.



WHAT ARE THE MAIN DIFFERENCES BETWEEN DOUBLY AND SINGLY LINKED LISTS?

## Singly linked list (2)

- ▶ Singly linked lists need less storage (but just by a factor of 2)
- ▶ But we cannot delete arbitrary elements in constant time because we first have to find the previous element in the list

# Linked lists vs. array

Linked lists:

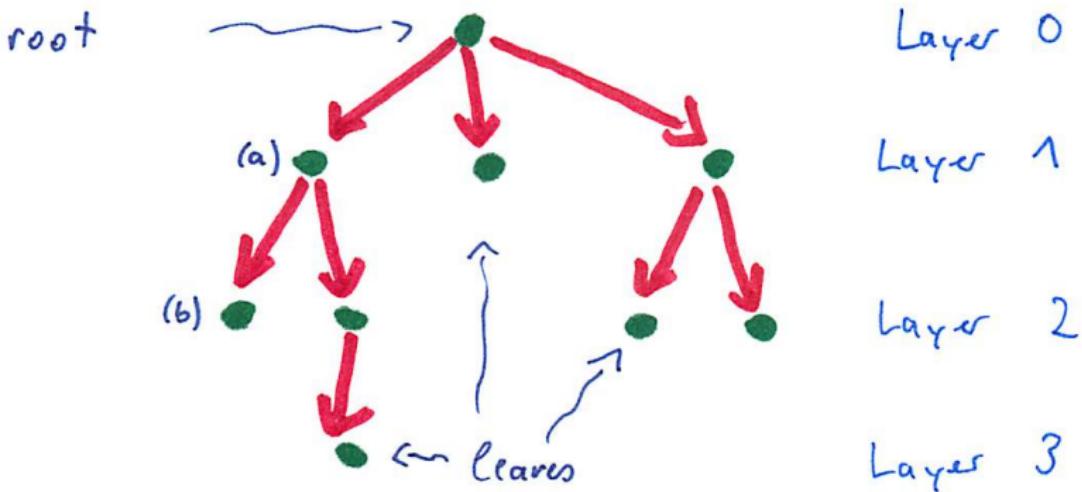
- ☺ we don't need to worry about allocation (as it would be the case in an array)
- ☹ But to access a particular element we need to walk through the list, this is costly if it has to happen very often.

Array:

- ☺ Fast access to all its elements
- ☹ Allocation is problematic if we don't know in advance how large the array is going to be.

# Trees

# General definitions



- (a) is the parent/predecessor of (b)
- (b) is a child of (a)

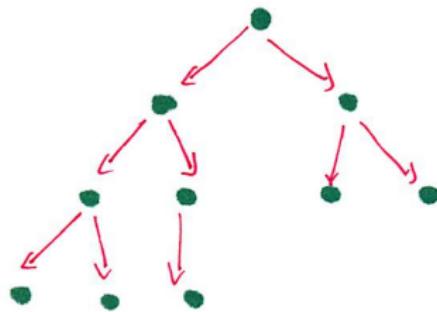
## General definitions (2)

Height of a vertex: length of the shortest path from the vertex to the root

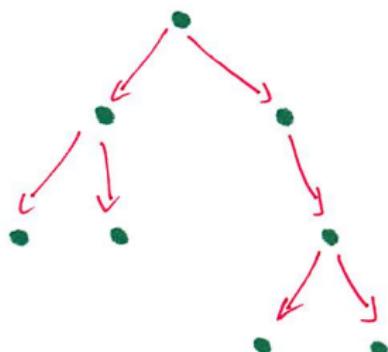
Height of the tree: maximum vertex height in the tree

# Binary tree

- ▶ In a **binary tree**, each vertex has at most 2 children.
- ▶ In a **complete binary tree**, all layers except the last one are filled. In the last layer, vertices are “as left as possible”.
- ▶ A **full binary tree** is a complete binary tree in which the last level is filled completely.



Complete binary tree



A binary tree

## Binary tree (2)

What is the height of a complete binary tree of  $n$  elements?

### Proposition 2 (Height of binary tree)

A full binary tree with  $n$  vertices has height

$\log_2(n + 1) - 1 \in \Theta(\log n)$ . A complete binary tree height  
 $\lceil \log_2(n + 1) - 1 \rceil \in \Theta(\log n)$ .

Proof:

## Binary tree (3)

- ▶ Number  $N$  of vertices in a full binary tree of height  $h$ :

$$N = 1 + 2 + 4 + 8 + 16 + \dots + 2^h$$

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

Here we used the formula for the geometric series: For  $a \neq 1$ ,

$$\sum_{i=0}^h a^h = \frac{1}{a-1} \sum_{i=0}^h (a-1)a^h = \frac{1}{a-1} (a^{h+1} - 1)$$

- ▶ Now solve the formula for  $N$ .

$$n = 2^{h+1} - 1 \iff h = \log_2(n+1) - 1$$

This gives the statement for a full binary tree.

## Binary tree (4)

- ▶ In case of a complete binary tree, the last layer might not be filled. Then we get:

$$h = \lceil \log_2(n + 1) - 1 \rceil$$

# Representation of trees

ASSUME YOU WANT TO STORE A TREE. HOW WOULD YOU DO THIS?

## Representation of trees (2)

For a complete binary tree, you can just use an array, where you fill the entries layer by layer (starting with the root).

## Representation of trees (3)

To implement an arbitrary binary tree:

- ▶ Each vertex contains the key value and pointers to left and right child vertices, and a pointer to its parent vertex

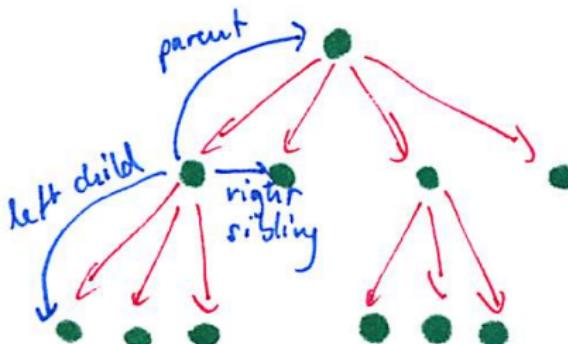
A similar scheme works for trees for which we know an upper bound for the number of children of each vertex.

## Representation of trees (4)

If the number of children per vertex is not known in advance, we use linked lists instead of arrays to store the siblings of a vertex.

A particular clever way to do this is the following:

- ▶ Each vertex has three pointers:
  - ▶ A pointer to its parent
  - ▶ A pointer to its leftmost child
  - ▶ A pointer to its right sibling



# Stack and queue

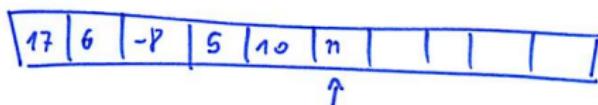
# Stack (German: Stapel)

- ▶ Dynamic structure to store a set of elements
- ▶ Analogy: Stack of books to read
- ▶ **Push(x)** (also called **Insert**) inserts the new element  $x$  to the stack
- ▶ **Pop** (also called **Delete**) removes the next element from the stack (LIFO: Last in first out)

Implementation: many possible ways.

- ▶ For example with an array and a pointer that points to the current element.
- ▶ Then Push and Pop take time  $O(1)$ .

Array:



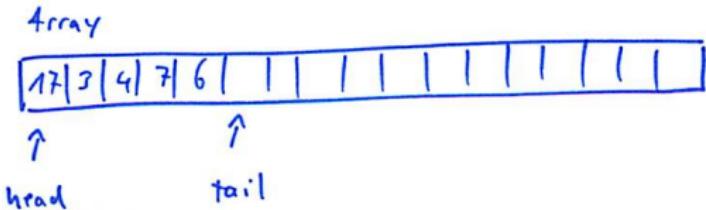
pointer to the current position

# Queue (German: Warteschlange)

- ▶ Dynamic structure to store a set of elements
- ▶ Analogy: line of customers
- ▶ **Insert(x)** (also called **Enqueue**) inserts the new element  $x$  to the end of the queue
- ▶ **Delete** (also called **Dequeue**, **Pop**) removes the next element from the queue (FIFO: First in first out)

Implementation: many possible ways.

- ▶ For example, array with two pointers, one to the head and one to the tail.
- ▶ Then both Insert and Delete take time  $O(1)$ .



# Heaps and priority queues

Literature:

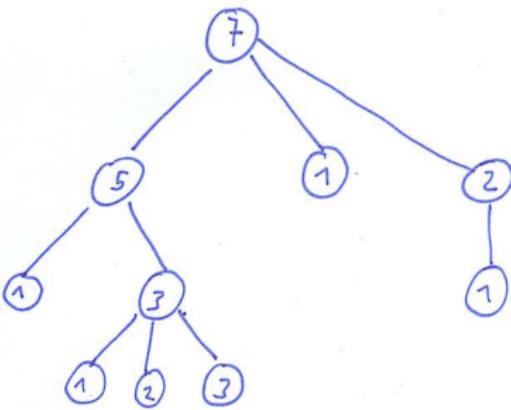
- ▶ Cormen Sec. 6
- ▶ Mehlhorn Sec. 6

# Heaps

# Heaps

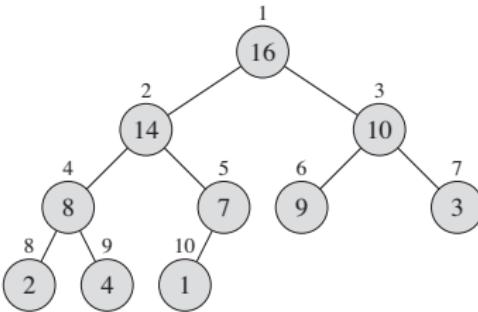
Heap in general:

- ▶ Data structure that stores elements as vertices in a tree
- ▶ Each element has a key value assigned to it
- ▶ Max-heap property: all vertices in the tree satisfy  
 $\text{key}(\text{parent}(v)) \geq \text{key}(v)$



# Heaps (2)

- ▶ Binary heap:
  - ▶ each vertex has at most two children.
  - ▶ Vertices are filled up such that we first complete a layer before we start a new one; when we do so, we fill vertices “from left to right”



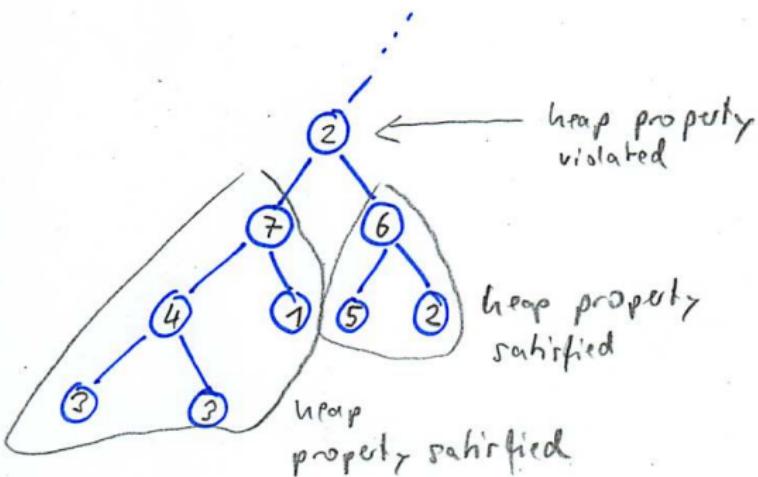
# Heaps (3)

Advantage of “binary”:

- ▶ control over height and width of the tree
- ▶ we can easily store the heap in an array without any additional pointers

# The most important operation: heapify

- ▶ Consider vertex  $i$  in the binary tree
- ▶ Assume that both of its subtrees satisfy the heap property, but the heap property is violated at  $i$  itself:  $\text{key}(i) < \text{key}(\text{child}(i))$  for at least one child of  $i$
- ▶ We now want to “repair” the heap.



# The most important operation: heapify (2)

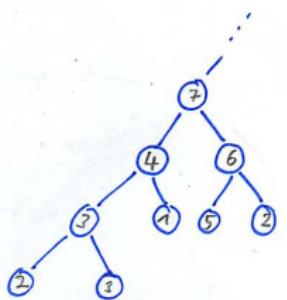
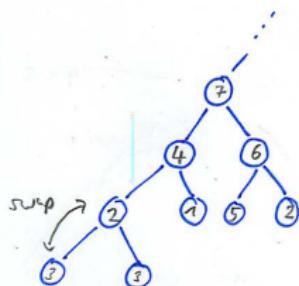
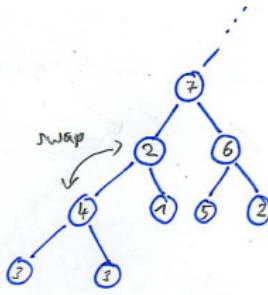
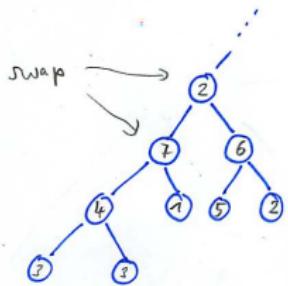
ANY IDEA?

# The most important operation: heapify (3)

Heapify procedure: “Let  $\text{key}(i)$  float down”

- ▶ Swap  $i$  with the larger of its children.
- ▶ Then recursively call heapify on this child.
- ▶ Stop when the heap condition is no longer violated.

# The most important operation: heapify (4)



# The most important operation: heapify (5)

Formally:

```
MAX-HEAPIFY( $A, i$ )
1    $l = \text{LEFT}(i)$ 
2    $r = \text{RIGHT}(i)$ 
3   if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4        $largest = l$ 
5   else  $largest = i$ 
6   if  $r \leq A.\text{heap-size}$  and  $A[r] > A[largest]$ 
7        $largest = r$ 
8   if  $largest \neq i$ 
9       exchange  $A[i]$  with  $A[largest]$ 
10      MAX-HEAPIFY( $A, largest$ )
```

# The most important operation: heapify (6)

Worst case running time of heapify:

- ▶ Assume the binary heap contains  $n$  vertices.
- ▶ The number of swapping operations is at most the height of the tree.
- ▶ Each swapping operation can be done in constant time.
- ▶ Then we know that its height is at most  $h = \lceil \log(n) \rceil = O(\log n)$ .
- ▶ So overall, the worst case running time is  $O(\log n)$ .

# Operation BuildMaxHeap

**Given** an unsorted array  $A$  of  $n$  elements, make a heap out of them.

ANY IDEAS?

## Operation BuildMaxHeap (2)

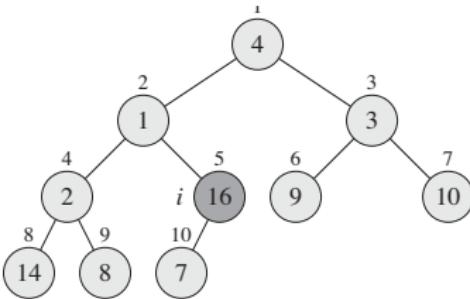
**Solution:** Starting at the bottom level, go through the vertices in each level and call heapify on each vertex. Do this “all the way up” to the root.

- 1 `# A is an unsorted array of  $n$  elements, filled in a binary tree`
- 2 `for  $i = n$  to 1 (in inverse order!)`
- 3   `MaxHeapify(A,i)`

(Note: we could be a bit cleverer by observing that elements  $\lfloor n/2 \rfloor + 1$  to  $n$  are definitely leafs, so we could start the for-loop at  $\lfloor n/2 \rfloor$  instead of  $n$ )

# Operation BuildMaxHeap (3)

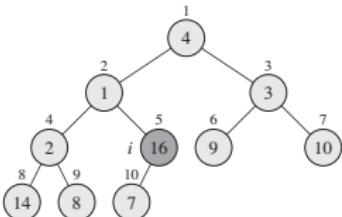
EVERYBODY: TRANSFORM THE FOLLOWING TREE IN A HEAP USING BUILDHEAP:



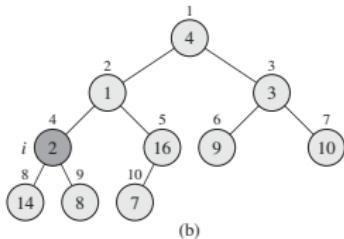
# Operation BuildMaxHeap (4)

Solution:

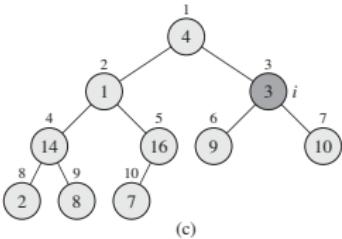
# Operation BuildMaxHeap (5)



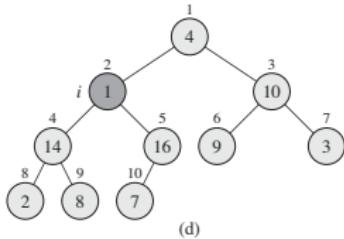
(a)



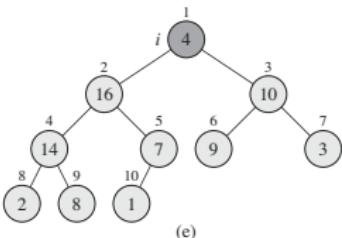
(b)



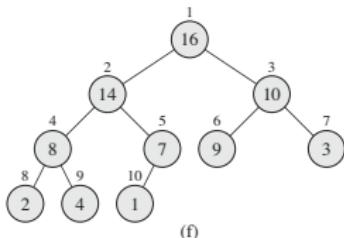
(c)



(d)



(e)



(f)

# Operation BuildMaxHeap (6)

**Running time:**

WHAT IS YOUR GUESS?

## Operation BuildMaxHeap (7)

- ▶ Simple upper bound: we call heapify on all  $n$  vertices. Each heapify operation takes  $O(\log n)$ , so overall we get  $O(n \log n)$ .
- ▶ More clever analysis: Observe that a heapify operation on a vertex of height  $h(v)$  has running time  $O(h(v))$ , and the height is much smaller than  $\log n$  for many vertices.
  - ▶ Denote by  $H$  the height of the tree.
  - ▶ To heapify a single vertex in level  $h(v)$  we have to make at most  $H - h(v)$  swap operations (bubble all the way down)
  - ▶ There are  $2^h$  vertices in level  $h$  of the tree
  - ▶ So the running time of BuildMaxHeap on  $n$  vertices is given as  $T(n) = \sum_{h=0}^H 2^h (H - h)$ .  
We can compute this sum as follows:

# Operation BuildMaxHeap (8)

$$\textcircled{1} \quad T(n) = \sum_{h=0}^H 2^h (H-h) \quad | \cdot 2$$

$$\textcircled{2} \quad 2 \cdot T(n) = \sum_{h=0}^H 2^{h+1} (H-h)$$

$$\textcircled{2} - \textcircled{1}: \quad T(n) = 2 \cdot T(n) - T(n) =$$

$$\begin{aligned}
 &= (2^1(H-0) + 2^2(H-1) + \dots + 2^H(H-(H-1))) \\
 &\quad - (2^0(H-0) + 2^1(H-1) + 2^2(H-2) + \dots + 2^{H-1}(H-1) + 2^H(H-H)) \\
 &\quad = 2^1 = 2^2 \quad = 2^H \\
 &= \underbrace{2^1 + 2^2 + \dots + 2^H}_{2^{H+1}-1} - H
 \end{aligned}$$

Substituting  $H = \log(n)$  gives  $O(2^{\log n + 1} - 1 - \log n) = O(n)$ .

# Operation BuildMaxHeap (9)



# Comments

WHY CAN'T WE START THE HEAPIFY PROCEDURE AT THE TOP OF THE TREE RATHER THAN AT THE BOTTOM?

## Comments (2)

ANSWER: The heapify procedure as we defined it only works if the subtrees are correct heaps.

EXERCISES:

- ▶ CAN YOU DEFINE A CORRECT HEAPIFY OPERATIONS THAT “BUBBLES TO THE TOP” RATHER TO THE BOTTOM?
- ▶ WHAT WOULD BE THE RUNNING TIME OF ALL THE OPERATIONS BUILDMAXHEAP, EXTRACTMAX, DECREASEKEY, ETC?

# Operation ExtractMax

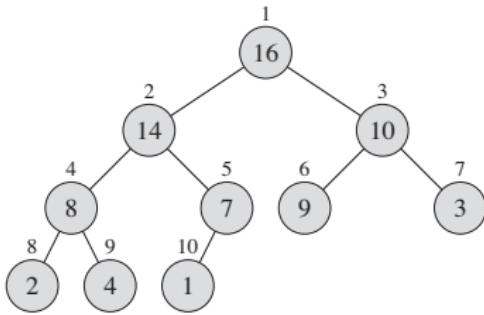
Goal: remove the largest element from the heap.

Solution:

- ▶ By construction, the largest element sits in the root.
- ▶ So we first extract the root element.
- ▶ Next, we replace the root element by the last leaf in the tree and remove that leaf
- ▶ Call  $\text{heapify}(\text{root})$

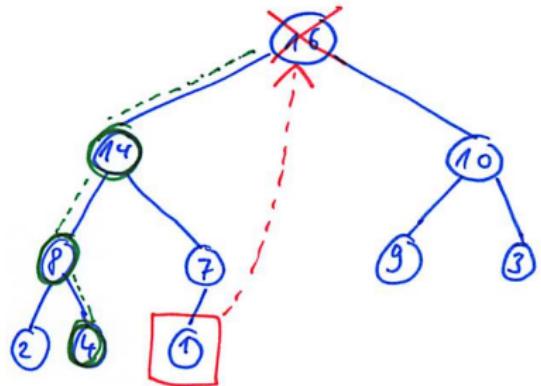
# Operation ExtractMax (2)

EVERYBODY: TRY TO REMOVE THE ROOT OF THE FOLLOWING HEAP:

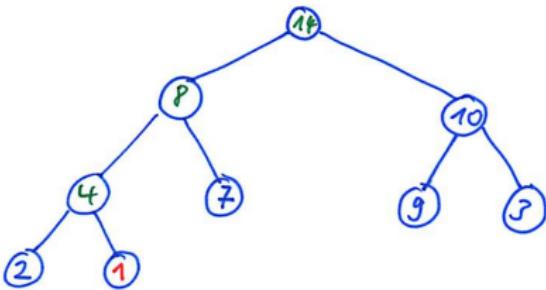


# Operation ExtractMax (3)

Solution: “let it bubble down”



Replace the root by the last element  
Heapify the root again



# Operation ExtractMax (4)

Running time:  $O(\log n)$

# Operation: DecreaseKey

Goal: *decrease* the key value of a particular element.

Solution:

- ▶ Decrease the value of the key
- ▶ Call heapify at this vertex to let it bubble down.

Running time:  $O(\log n)$

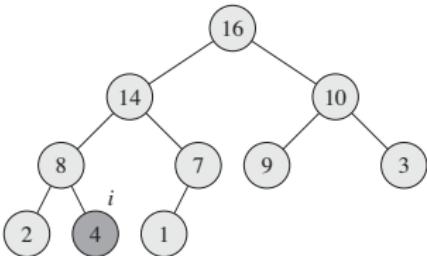
# Operation: IncreaseKey

Goal: *increase* the key value of a particular element.

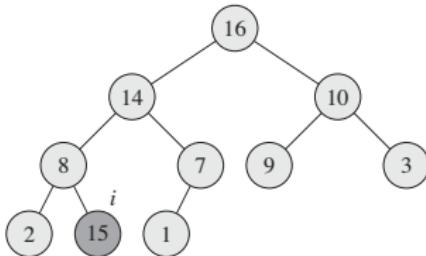
Solution:

- ▶ Increase the value of the key
- ▶ Do “heapify-up”: Walk upwards to the root, in each step exchanging the key values of a vertex and its parent if the heap property is violated.

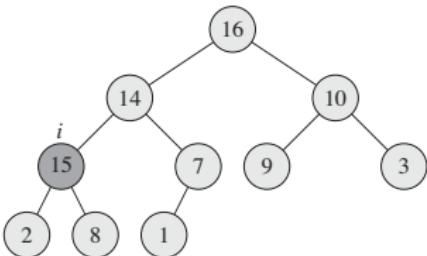
# Operation: IncreaseKey (2)



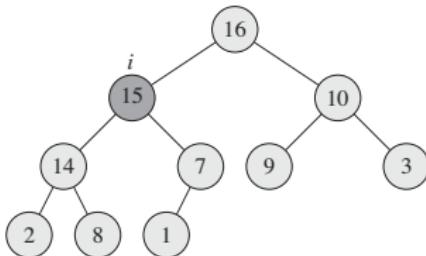
(a)



(b)



(c)



(d)

Running time:  $O(\log n)$

# Operation: InsertElement

Goal: insert a new element to the heap

Solution:

- ▶ insert it at the next free position, first assign it the key  $-\infty$ .
- ▶ Then call IncreaseKey to set the key to the given value.

Running time:  $O(\log n)$

# Comments / Outlook

Why are heaps useful?

- ▶ We can easily extract the maximum element.
- ▶ But a heap is much less restrictive than an ordered set (less costly to maintain).

## Comments / Outlook (2)

- ▶ Heaps are used all over the place, we are going to see them often in this course.
- ▶ There exist much more elaborate heap implementations that try to decrease the running times of some of the operations.

If you are interested, visit the “Algorithmik”-lectures in the master program ...

# Priority queue

# Priority queue

- ▶ Data structure to maintain a set  $S$  of elements
- ▶ Each element has a key value
- ▶ When we dequeue the next element, we want to get the one with the largest key value.

Standard operations are:

- ▶ Enqueue, Dequeue, IncreaseKey of a particular element

## Priority queue (2)

Standard implementation uses heaps:

- ▶ Store the elements in a heap
- ▶ Enqueue: heap InsertElement,  $O(\log n)$
- ▶ Dequeue: heap ExtractMax  $O(\log n)$
- ▶ IncreaseKey: heap Increase key  $O(\log n)$

Note: there exist more elaborate implementations with better amortized running times, see the “Algorithmik” master class.

# Hashing

Literature: Cormen 11, Dasgupta 1.5

# Basics

# General problem

- ▶ Assume we are an online shop and want to maintain a list of data related to the customers that are currently online (say, the pages they clicked on during the last 20 min).
- ▶ We can identify customers by IP address
- ▶ But who is online changes constantly
- ▶ We want to have fast access time to the data.

First attempt:

- ▶ create an array of the size of all possible IP address:  $2^{32}$  (IP addresses have 32 bits)
- ▶ Fill the entries corresponding to the customers that are currently active.
- ▶ Fast access time

## General problem (2)

- ▶ But obviously not a good idea as the number of active customers is much smaller than the number of possible IP addresses.

# General problem (3)

Second attempt:

- ▶ Simply generate a linked list of some kind.
- ▶ Efficient for space, but might have long access times for a particular customer.

Third attempt:

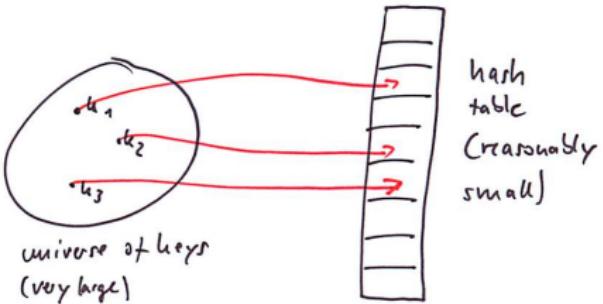
- ▶ Write the IP addresses consecutively in an array.

Problem: we don't know which IP address is where, so access time is as slow as if we used a list (we need to walk through the array to find the position of an element).

# General problem (4)

The solution is hashing.

# Hashing, general principle



- ▶ Want to store data that is assigned to particular key values (the IP addresses)
- ▶ Give a “nickname” to each of the key values (for all  $2^{32}$  IP addresses).
- ▶ Choose the space of nicknames reasonably small (a bit larger than the number of customers that are usually online)
- ▶ Have a way to compute “nicknames” from the keys themselves

# Hashing, general principle (2)

- ▶ Then store the information in an array (size = number of nicknames)

# Hashing, general principle (3)

In the example above:

- ▶ Want to write the IP addresses in a small array.
- ▶ But instead of writing it consecutively (and not knowing where each element is) we generate a “lookup-table” that tells us in which cell of the array each element is going to end up.
- ▶ This lookup-table is the hash function.

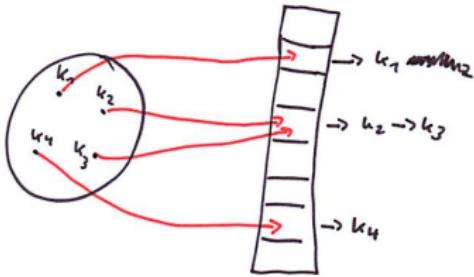
# Hashing, formally

- ▶ The set of all possible keys we want to access is called the **universe  $U$  of keys**.
- ▶ Assume that the set of actual key values is much smaller, say it has size  $m \ll |U|$ .
- ▶ A **hash function** is a function  $h : U \rightarrow \{1, \dots, m\}$ .
- ▶ We say: **the element with key  $k$  hashes to slot  $h(k)$** .
- ▶ The values  $h(k)$  are called **hash values**.

# Hashing, formally (2)

## Collisions:

- ▶ From the definition of  $h$  it is obvious that many keys get the same hash values:  $h(k_1) = h(k_2)$  even though  $k_1 \neq k_2$ .
- ▶ This is called a **collision**.
- ▶ To cope with them:  
Let each entry of the hash table point to a linked list that contains all elements with this particular hash key.



# Some simple hash functions

# What is a good hash function?

- ▶ We want that all our elements are distributed to the hash keys as uniformly as possible.
- ▶ We want to avoid collisions (they decrease our access speed as we have to travel through the list of collided keys to find the one we are interested in)
- ▶ We don't want the set of hash values to be too large (otherwise we waste space)

# Examples for simple hash functions

Division method for hashing integers:

- ▶ Assume that the universe of keys is  $\mathbb{N}$ .
- ▶ Assume you want to hash to  $m$  elements.
- ▶ Define the hash function

$$h(k) = k \bmod m.$$

# Examples for simple hash functions (2)

Choice of  $m$ :

- ▶ If we don't know anything about our data set, then any  $m$  is as good as any other.
- ▶ But in practice, some values of  $m$  should be avoided:
  - ▶ For example: If we choose  $m = 2^a$ , then  $h(k)$  is just the  $a$  lowest-order bits of  $k$ . If the lower-order bits are not distributed uniformly, then this is a bad choice.
  - ▶ A general rule of thumb: if you want to hash to approximately  $m$  slots, then choose a prime number  $m_p$  that is a bit larger than  $m$ .

# Examples for simple hash functions (3)

Multiplication method for hashing integers:

- ▶ Multiply the key value by a constant  $a$  with  $0 < a < 1$
- ▶ Then take the fractional part  $p$  of the result (the part behind the comma)
- ▶ Then multiply  $p$  by  $m$  and take the ceil

In formulas:

$$h(k) = \lceil m (k \cdot a \bmod 1) \rceil$$

Example: Say,  $m = 50$ . Choose  $a = 0.17$ . Then:

$$k = 1 \implies k \cdot a = 0.17 \implies 0.17 \cdot m = 8.5 \implies \lceil 0.17 \cdot m \rceil = 9$$

$$k = 2 \implies k \cdot a = 0.34 \implies 0.34 \cdot m = 17$$

# Examples for simple hash functions (4)

Rationale:

- ▶ if  $p$  were a number uniformly drawn from  $[0, 1]$ , then this would be a good choice.
- ▶ We hope that the actual values of  $p$  behave more or less similarly

# What is the best hash function?

Depending on the actual key values:

- ▶ If we know the distribution of the key values, we can try to exploit this when selecting a hash function.
- ▶ If we don't know it, we might make a bad choice.

CAN WE DESIGN A HASH FUNCTION THAT WORKS GOOD FOR ALL SETS OF KEYS?

# What is the best hash function? (2)

No!!!

- ▶ If you give me a hash function ...
- ▶ and I am your adversary, then I can design a data set that performs worst for your hash function (HOW?)

A “best hash function” that works for all kinds of data sets does not exist!!!

# Sorting

# Elementary Sorting Algorithms

# Problem of Sorting

Given: an array of  $n$  numbers

76, 354, 2, 653, 24, 3, 4, 6

Goal: Sort the numbers in increasing order.

2, 3, 4, 6, 24, 76, 354, 653

- ▶ This is one of the most basic operations on an array, and it is needed all over the place.
- ▶ We are now going to spend quite some time to investigate various sorting algorithms.

# Warmup: Elementary Sorting algorithms

EVERYBODY:

- ▶ Take a couple of minutes to come up with your own sorting algorithm.
- ▶ Try to write it up in pseudo-code.
- ▶ Try to estimate its running time.

# Selection sort

# Selection sort

Idea:

- ▶ Find the smallest element in the input array  $A$
- ▶ Remove it from the input array and write it in the output array  $B$
- ▶ Repeat this process.

**SelectionSort( $A$ )** (Naive implementation with arrays)

```
1 n = length(A)
2 B = new array of length n
3 for it=1, ..., n
4   Find the smallest element a and its index p in A
5   B(it) = a # Insert the element in B
6   A(p) = NaN # Replace the element in A by NaN
```

# Selection sort (2)

7 Return B

## Selection sort: example

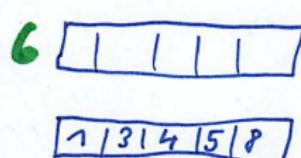
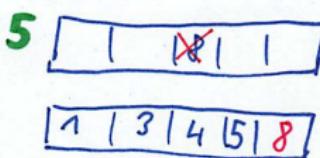
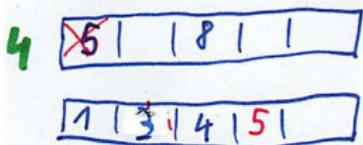
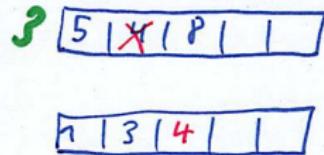
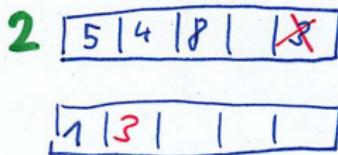
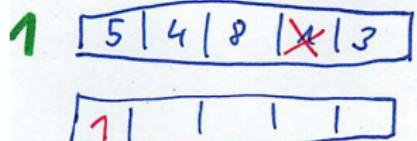
EVERYBODY: RUN SELECTION SORT ON THE EXAMPLE SEQUENCE

5,4,8,1,3

KEEP TRACK OF THE INPUT AND OUTPUT ARRAY IN EACH STEP.

# Selection sort: example (2)

Selection sort:



# Analysis of selection sort: arrays or lists?

**Is it better to implement selection sort with arrays or lists?**

# Analysis of selection sort: arrays or lists?

**Is it better to implement selection sort with arrays or lists?**

- ▶  $A$  as a list (WHY)
- ▶ For  $B$  it does not really matter. (How would we need to change the code if we used lists)?

# Analysis of selection sort: termination

**Does the algorithm always terminate?**

# Analysis of selection sort: termination

**Does the algorithm always terminate?**

Yes. The for loop has a fixed number of iterations, so nothing can go wrong.

# Analysis of selection sort: correctness

**Is the array returned by the algorithm always sorted in increasing order?**

# Analysis of selection sort: correctness (2)

**Is the array returned by the algorithm always sorted in increasing order?**

Yes. To prove it formally, we use the concept of an **invariant**:

- ▶ An invariant is a logical assertion for which we prove that it always holds true during the execution of an algorithm.

# Analysis of selection sort: correctness (3)

## Proposition 3 (Invariant of selection sort)

- (1) After the  $i$ -th execution of the for-loop of selection sort, the first  $i$  elements in Array  $B$  are sorted in increasing order.
- (2) No element in  $B$  is strictly larger than an element in  $A$ .

Proof by induction.

- ▶ Base case (“Induktionsanfang”)  $i = 1$ : clear:
  - (1)  $B$  just contains one element so far, so it is sorted.
  - (2) The element in  $B$  was the smallest in  $A$ , so (2) correct.
- ▶ Induction hypothesis (“Induktionsannahme”): Assume the statement holds for  $i = 1, \dots, k$ .

## Analysis of selection sort: correctness (4)

- ▶ Induction step (“Induktionsschritt”): We now have to prove that it also holds for  $k + 1$ .

In the for-loop, we pick the smallest remaining element of  $A$ . By the induction hypothesis (2), this is never smaller than any element in  $B$ . So when we add it to the end of  $B$ , the sequence in  $B$  is sorted increasingly, hence (1) is true for  $k + 1$ .

(2) remains true as well, obviously by the way we choose the new element.



The correctness of the algorithm now follows from the invariant.

# Analysis of selection sort: time complexity

**What is the worst case running time of the algorithm?**

# Analysis of selection sort: time complexity (2)

## What is the worst case running time of the algorithm?

The running time is  $O(n^2)$ :

- ▶ line 1:  $O(1)$
- ▶ line 2:  $O(n)$
- ▶  $n$  runs of the for-loop. Inside the for-loop:
  - ▶ line 4:  $O(n)$
  - ▶ lines 5 and 6:  $O(1)$
- ▶ line 7:  $O(1)$

So overall, the dominating part is that we run the for-loop  $n$  times and each run needs time  $O(n)$ , so we end with worst case running time  $O(n^2)$ .

## Analysis of selection sort: time complexity (3)

Note that the running time would also  $O(n^2)$  if we stored the input in a linked list.

- ▶ Advantage would be: we could just remove the elements in line 6.
- ▶ Then looking for the smallest element (Line 4) would take less time.
- ▶ Overall, line 4 would need
$$n + (n - 1) + \dots + 2 + 1 = n \cdot (n + 1)/2$$
 time steps, but this is still  $O(n^2)$ .

# Analysis of selection sort: time complexity (4)

**What is the best case running time?**

... is  $O(n^2)$  as well. WHY?

# Analysis of selection sort: space complexity

**What is the space complexity of the algorithm?**

# Analysis of selection sort: space complexity (2)

**What is the space complexity of the algorithm?**

It is  $O(n)$ :

- ▶ Arrays  $A$  and  $B$  both need space  $O(n)$ , everything else is constant.

Exercise: The algorithm can also be implemented “in-place”, that is without an additional array  $B$ .

# Insertion sort

# Insertion sort

This is the principle that most people use when playing card games and they want to sort the cards in their hand:

- ▶ One after the other, pick up the next element from the unsorted array.
- ▶ Maintain a sorted output array  $B$  and always insert the current element to its correct position.

# Insertion sort (2)

INSERTION-SORT( $A$ )

```
1  for  $j = 2$  to  $A.length$ 
2       $key = A[j]$ 
3      // Insert  $A[j]$  into the sorted sequence  $A[1..j - 1]$ .
4       $i = j - 1$ 
5      while  $i > 0$  and  $A[i] > key$ 
6           $A[i + 1] = A[i]$ 
7           $i = i - 1$ 
8       $A[i + 1] = key$ 
```

## Insertion sort: example

EVERYBODY: RUN SELECTION SORT ON THE EXAMPLE SEQUENCE

5,4,8,1,3

KEEP TRACK OF THE INPUT AND OUTPUT ARRAY IN EACH STEP.

# Insertion sort: example (2)

Insertion sort

1

<del>5</del>	4	8	11	3
5				

2

<del>5</del>	<del>4</del>	8	11	3
5				

3

1	<del>18</del>	11	3
4	5		

4

1	1	<del>18</del>	3
4	5	8	1

5

1	1	1	<del>18</del>	3
1	4	5	8	1

6

1	1	1	1
1	3	4	5

# Insertion sort: properties

Properties:

- ▶ Termination: yes
- ▶ Correctness: yes
- ▶ Worst case running time:  $O(n^2)$
- ▶ Best case running time:  $O(n)$
- ▶ Space complexity:  $O(n)$

Proof: exercise!

# Insertion sort: ideas for improvement

Here are some ideas to improve the algorithm:

- ▶ Is it better to use arrays or lists?
- ▶ Could we use binary search to speed-up the step of inserting the element to its correct position?

Answers: exercise!

# Bubble sort

# Bubble sort

You have seen this algorithm in the exercises already:

BUBBLESORT( $A$ )

```
1  for  $i = 1$  to  $A.length - 1$ 
2      for  $j = A.length$  downto  $i + 1$ 
3          if  $A[j] < A[j - 1]$ 
4              exchange  $A[j]$  with  $A[j - 1]$ 
```

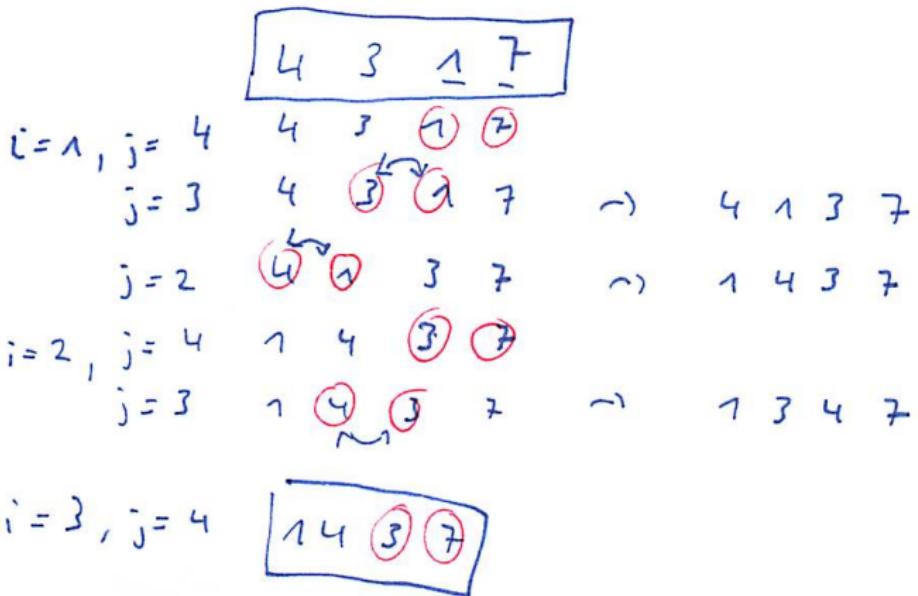
## Bubble sort (2)

EXAMPLE: RUN BUBBLE SORT WITH THE FOLLOWING  
EXAMPLE:

4 3 7 1

# Bubble sort (3)

Solution:



# Bubble sort: properties

Properties:

- ▶ Always terminates
- ▶ It is correct.
  - ▶ First iteration of the outer loop ensures that the smallest element is at the first position.
  - ▶ Second iteration moves the second smalles element to its correct place.
  - ▶ and so on ...
- ▶ Worst case running time  $O(n^2)$
- ▶ Average case running time  $O(n^2)$
- ▶ Can tweak the code to achieve best case running time  $O(n)$ .  
HOW? ON WHICH INSTANCE?

Proof: Exercise

# Mergesort

Literature: all text books

History: apparently, invented by John von Neumann in the 1940ies

# Motivation

So far we have seen a number of naive sorting algorithms, but all of them had worst case running time  $O(n^2)$ .

We now want to show an algorithm that is provably faster.

We want to use the **divide-and-conquer principle**:

- ▶ We iteratively divide the given problems into smaller subproblems
- ▶ Then we construct the solution of the whole problem by combining the solutions of the smaller problems.

# Idea of the algorithm

Idea of merge sort:

- ▶ Assume we are given a list of numbers
- ▶ Split the list in two halves
- ▶ Recursively sort each of the lists
- ▶ Then merge the two sorted lists again.

# Mergesort: Pseudo code

```
function mergesort( $a[1\dots n]$ )
```

Input: An array of numbers  $a[1\dots n]$

Output: A sorted version of this array

```
if  $n > 1$ :
```

```
    return merge(mergesort( $a[1\dots \lfloor n/2 \rfloor]$ ), mergesort( $a[\lfloor n/2 \rfloor + 1\dots n]$ ))
```

```
else:
```

```
    return  $a$ 
```

```
function merge( $x[1\dots k], y[1\dots l]$ )
```

```
if  $k = 0$ : return  $y[1\dots l]$ 
```

```
if  $l = 0$ : return  $x[1\dots k]$ 
```

```
if  $x[1] \leq y[1]$ :
```

```
    return  $x[1] \circ \text{merge}(x[2\dots k], y[1\dots l])$ 
```

```
else:
```

```
    return  $y[1] \circ \text{merge}(x[1\dots k], y[2\dots l])$ 
```

Here  $\circ$  denotes the concatenation of the lists.

# Merge sort: Example

EVERYBODY: APPLY THE MERGE OPERATIONS TO THE TWO SEQUENCES

3 4 5 9

and

1 2 3 4

## Merge sort: Example (2)

merge (3 4 5 9, 1 2 3 4)

1 o merge (3 4 5 9, 2 3 4)

1 2 o merge (3 4 5 9, 3 4)

1 2 3 o merge (4 5 9, 3 4)

1 2 3 3 o merge (4 5 9, 4)

1 2 3 3 4 o merge (5 9, 4)

1 2 3 3 4 4 o merge (5 9, [] )

1 2 3 3 4 4 5 9

## Merge sort: Example (3)

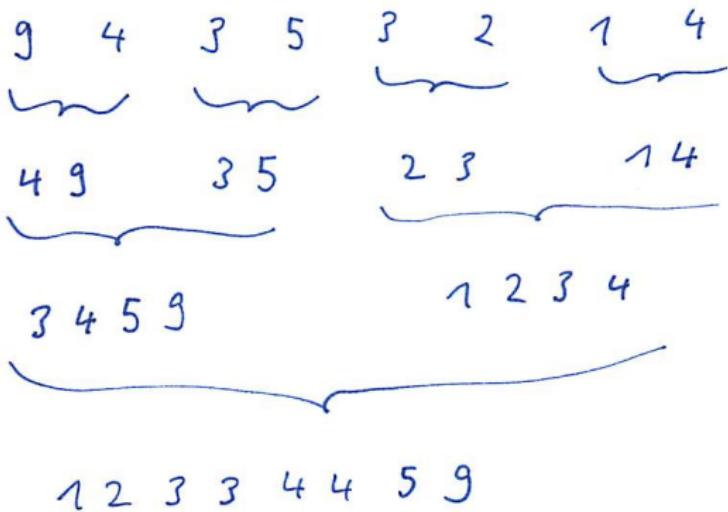
EVERYBODY: APPLY THE MERGE SORT ALGORITHM TO THE SEQUENCE

9, 4, 3, 5, 3, 2, 1, 4

## Merge sort: Example (4)

Input 9 4 3 5 3 2 14

Recursively split in singletons, then merge:



# Merge sort: Running time

Proposition 4 (Worst case running time of merge sort)

The worst case running time of merge sort is  $O(n \log n)$ .

Proof:

- ▶ `merge(x[1..k], y[1..l])` has running time  $O(k + l)$ :
  - ▶ Constant amount of work per recursive call
  - ▶  $k + l$  recursive calls
- ▶ Because we only ever apply merge to arrays of length at most  $n$ , we have  $k, l \leq n$ , hence  $O(k + l) \subset O(n)$ .
- ▶ So the running time of merge sort satisfies

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

# Merge sort: Running time (2)

- ▶ By the master theorem, this leads to  $O(n \log n)$ .



# Merge sort: Running time (3)

Be careful when you actually implement the algorithm:

- ▶ Depending on your programming language, calling `merge(A, B)` first copies the arrays  $A$  and  $B$ !
- ▶ Then the time is much worse (and space as well)! Exercise: how much?
- ▶ So make sure you just hand over the pointers to  $A$  and  $B$  ...

# Merge sort: Running time (4)

**What is the best case running time for merge sort?**

We can't really save any time. Even if we start with a sorted sequence, we still go through all the steps.

Hence the best case running time is still  $O(n \log n)$ !

Consequently, also the average case running time of merge sort is  $O(n \log n)$  as well.

## Merge sort: space complexity

- ▶ A naive implementation needs extra space of the size  $n$  of the original array in order to store the intermediate results.
- ▶ It is not straight forward but possible to achieve in-place sorting with mergesort (if this is the goal, other algorithms are better, see later for discussion). Then the running time increases slightly, to  $n \log^2 n$ .

# Termination and correctness

Is pretty straight forward to prove the correctness of the algorithm using invariants.

Exercise ...

# Heap sort

Literature: Cormen Sec. 6

Original paper for heap sort:

Williams, J. W. J. (1964). Algorithm 232 - Heapsort.  
Communications of the ACM 7 (6): 347-348

# Heapsort: idea

Idea is to exploit the heap data structure.

- ▶ We simply build a heap out of our input.
- ▶ Then we remove the largest element.
- ▶ Then we heapify again.
- ▶ And so on ...

# Heap sort: pseudo-code

Naive implementation:

**HeapSort(A)**

- 1  $n := \text{lengthA}$
- 2  $B = \text{empty array of length } n$
- 3  $H = \text{BuildMaxHeap}(A)$
- 4 **for**  $i=1 \dots \text{length}(A)$
- 5      $B(n-i) = \text{ExtractMax}(H)$

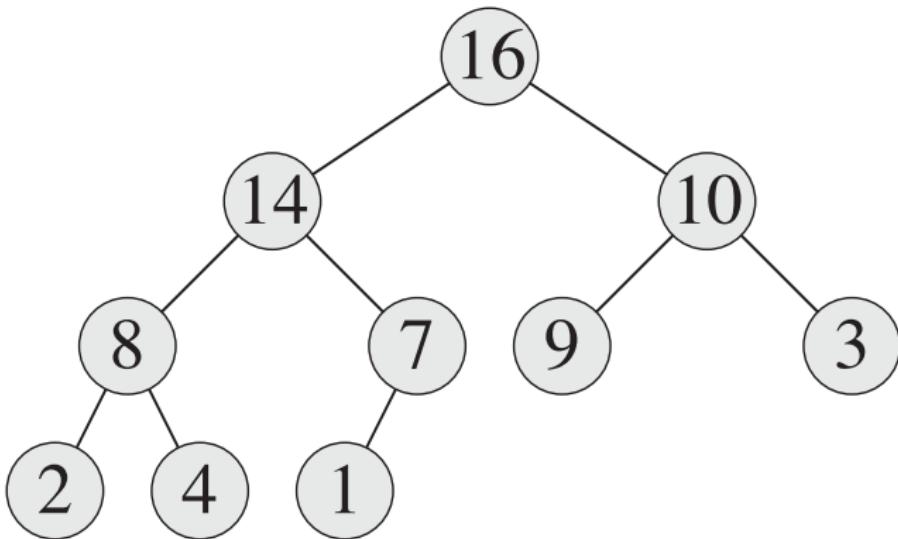
## Heap sort: pseudo-code (2)

One can also make heap-sort in-place. IDEA?

## Heap sort: example

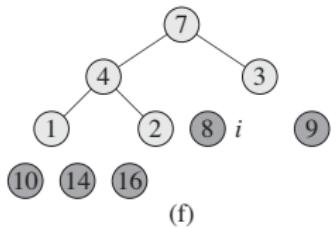
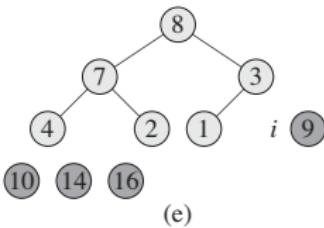
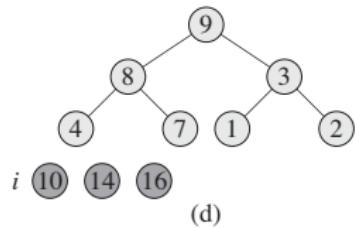
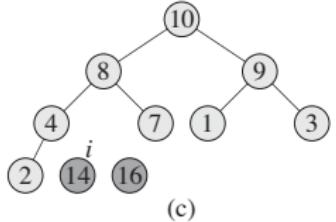
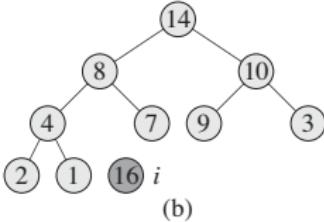
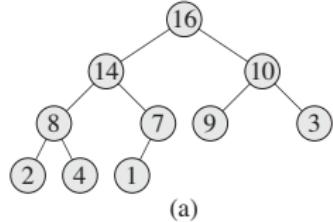
EVERYBODY, TRY HEAP SORT ON THE FOLLOWING INSTANCE:

## Heap sort: example (2)

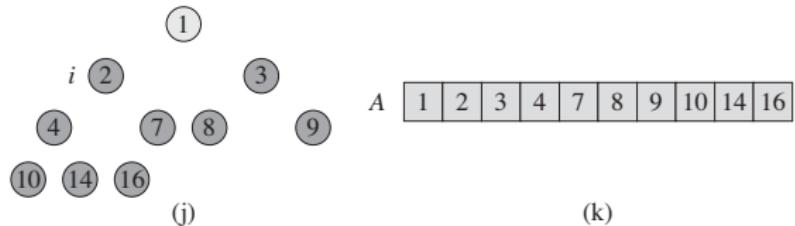
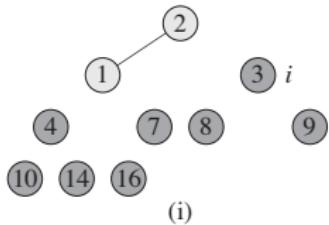
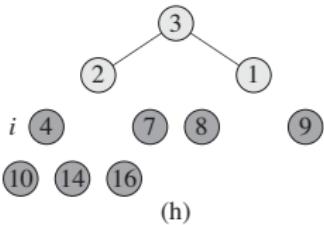
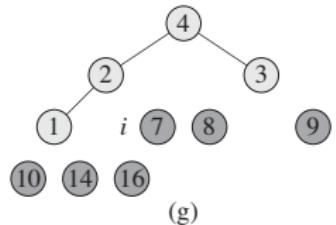


(a)

# Heap sort: example (3)



# Heap sort: example (4)



(k)

A	[ 1   2   3   4   7   8   9   10   14   16 ]
---	--

## Heap sort: example (5)

Note that this example already shows how sorting can be done in-place. DO YOU SEE HOW?

# Heap sort: termination and correctness

Termination and correctness are both clear from the heap properties.

Exercise: prove it formally.

# Heap sort: running time

**What is the running time of heap sort?**

# Heap sort: running time

**What is the running time of heap sort?**

- ▶ Build-max-heap takes  $O(n)$
- ▶ Each of the heapify calls takes  $O(\log n)$ , and we have  $n$  such calls.

So altogether:  $O(n \log n)$ .

# Heap sort: running time

**What is the running time of heap sort?**

- ▶ Build-max-heap takes  $O(n)$
- ▶ Each of the heapify calls takes  $O(\log n)$ , and we have  $n$  such calls.

So altogether:  $O(n \log n)$ .

**What about the best case running time?**

# Heap sort: running time

**What is the running time of heap sort?**

- ▶ Build-max-heap takes  $O(n)$
- ▶ Each of the heapify calls takes  $O(\log n)$ , and we have  $n$  such calls.

So altogether:  $O(n \log n)$ .

**What about the best case running time?**

Is  $O(n \log n)$  as well (WHY?)

# Heap sort: space complexity

Heap sort works in-place! So we don't need any extra space.

... make sure you really understand why ...

# Lower bound

Literature:

Dasgupta Sec 2.3

Cormen Sec. 8.1

Mehlhorn Sec. 5.3

# Motivation

We have seen several algorithms that solve the sorting problem in  $\Theta(n \log n)$ . Is this the best we can do?

To answer this question:

- ▶ We need a **lower bound** (German: untere Schranke) on the worst case running time
- ▶ The statement is going to look as follows:

Under [some assumptions], no algorithm that solves the sorting problem can have a worst case running time smaller than [lower bound].

- ▶ Lower bounds are notoriously difficult (WHY?), and only exist for a few problems (sorting is one of them).

# Comparison-based sorting algorithms

In this section we only consider sorting algorithms that are based on pairwise comparison of elements:

- ▶ Input to the algorithm is the sequence  $A = [a_1, \dots, a_n]$ . Assume that all elements in  $A$  are distinct, that is  $a_i \neq a_j$  for all  $i \neq j$ .
- ▶ The algorithm is allowed to make arbitrarily many comparisons of the form: Is  $a_i \leq a_j$  or is  $a_i > a_j$ ?
- ▶ Based on the results of these comparisons, it finally produces the correctly sorted output.

Remark:

- ▶ All sorting algorithms we have seen so far follow this principle.

# The lower bound

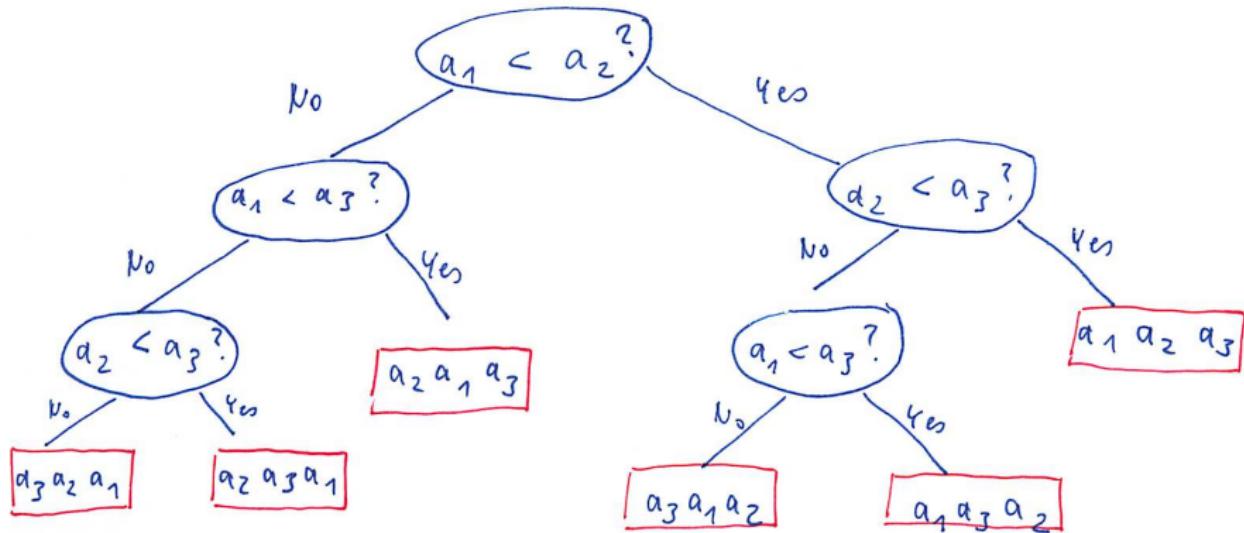
## Theorem 5 (Lower bound for sorting)

Any comparison-based sorting algorithm requires  $\Omega(n \log n)$  comparisons in the worst case.

Note: A priori, this theorem does not apply to randomized sorting algorithms (see later for a discussion).

# Proof ingredient: computation tree

Consider the computation tree of a comparison-based algorithm applied to an array with three elements  $a_1, a_2, a_3$ :



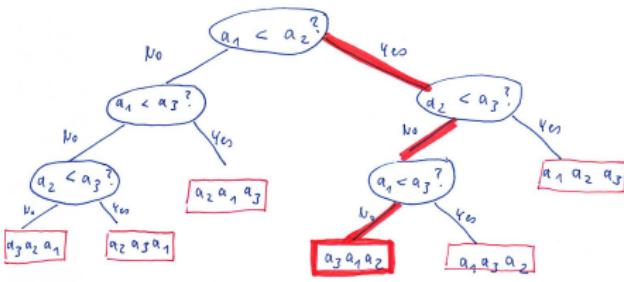
## Proof ingredient: computation tree (2)

**Any comparison-based algorithm can be described by such a computation tree (one tree for each input length  $n$ ):**

- ▶ At some point, potentially after some preprocessing, the algorithm starts with a first comparison.
- ▶ Then, depending on the outcome, the algorithm proceeds and asks a new comparison question.

# Proof ingredient: computation tree (3)

- If we fix an ordering of the input sequence, then the execution of the algorithm follows a deterministic set of questions. This corresponds to one path in the tree, from the root to the corresponding leaf.



## Proof ingredient: computation tree (4)

- ▶ It is really possible to form a tree out of all these single paths:
  - ▶ Because the algorithm is deterministic, it always starts with the same question. This is the root.
  - ▶ Whenever the answer to the first  $n_0$  questions was the same, the algorithm “does the same thing”.  
On the tree, this means that we follow the same path.
  - ▶ Just when the answer was different, the algorithm proceeds differently.  
On the tree, we now follow different paths from now on.
  - ▶ Even if an algorithm is stupid and asks the same question for a second time, we simply add the question again on the path from the root to the answer.
- ▶ So each algorithm can be described as one particular such computation tree.

# Proof ingredient: computation tree (5)

Exercise: build the tree for the mergesort algorithm, applied to a sequence of 4 elements.

## Proof ingredient: computation tree (6)

**The computation tree is always binary, but not necessarily complete.**

Exercise: Why?

## Proof of the lower bound (2)

Given the tree, how can you see the worst case running time of the algorithm?

## Proof of the lower bound (2)

Given the tree, how can you see the worst case running time of the algorithm?

The height  $h$  of the tree.

## Proof of the lower bound (2)

Given the tree, how can you see the worst case running time of the algorithm?

The height  $h$  of the tree.

To get a lower bound we need to find out what is the minimal height of the tree. Ideas?

## Proof of the lower bound (2)

Given the tree, how can you see the worst case running time of the algorithm?

The height  $h$  of the tree.

To get a lower bound we need to find out what is the minimal height of the tree. Ideas?

What do we know about the number of leaves of the tree?

## Proof of the lower bound (2)

Given the tree, how can you see the worst case running time of the algorithm?

The height  $h$  of the tree.

To get a lower bound we need to find out what is the minimal height of the tree. Ideas?

What do we know about the number of leaves of the tree?

At least  $n!$  leaves, one for each possible permutation of the input sequence.

## Proof of the lower bound (2)

Given the tree, how can you see the worst case running time of the algorithm?

The height  $h$  of the tree.

To get a lower bound we need to find out what is the minimal height of the tree. Ideas?

What do we know about the number of leaves of the tree?

At least  $n!$  leaves, one for each possible permutation of the input sequence.

What is the minimal height of a binary tree with at least  $n!$  leaves?

Height is minimal if the binary tree is complete. By Stirling's approximation,  $\log(n!) \approx n \log n - n + O(\log n)$ , that is  $h = \Omega(n \log n)$ .



## Remarks

- ▶ The lower bound shows that there cannot exist any comparison-based sorting algorithm that has worst case running time  $o(n \log n)$ .

Another way to state this:

For any comparison-based sorting algorithm there always exists an instance such that the algorithm on that instance needs  $\Omega(n \log n)$ .

## Remarks (2)

- ▶ In this sense, heap sort and merge sort are asymptotically worst-case optimal.
- ▶ Note again that all constants in the running time got swallowed by the Landau notation. It might very well be that one sorting algorithm takes  $2n \log n$  and the other one takes  $10^5 n \log n$ , which would make a considerable difference in practice.

# Outlook

- ▶ One can prove that the  $n \log n$  lower bound also holds for the average running time.
- ▶ One can also show that the  $n \log n$  lower bound also holds for the seemingly simpler “element uniqueness problem”: the problem of deciding whether a sequence contains two elements that are equal (using comparisons only).

## Outlook (2)

- ▶ History: I did not manage to find out where and when the lower bound was proved first. A much more general statement appears in the work on algebraic decision trees, with the following two seminal papers:
  - ▶ David P. Dobkin and Richard J. Lipton. On the complexity of computations under varying sets of primitives. *Journal of Computer and System Sciences*, 18(1):86-91, Feb. 1979.
  - ▶ Michael Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (STOC 1983)*, pp. 80-86, April 1983.

# Appetizer / Excursion: finding the median

Literature:

- ▶ Dasgupta Sec. 2.4
- ▶ Kleinberg Sec. 13.5
- ▶ Mehlhorn Sec. 5.5

# Median

- ▶ The median element of a sorted sequence of integers  $a_1, \dots, a_n$  is the element with index  $\lceil n/2 \rceil$ : the element in the middle.
- ▶ The median is important in many applications: it is interpreted as the “typical” element of the list.
- ▶ Note the difference between median and mean:

Example:

$$\underbrace{1, 1, 1, 1, 1, \dots, 1}_{\text{100 times}}, 1000$$

- ▶ Median is 1
- ▶ Mean is  $\sum_{i=1}^n a_i / n = 11$

# Median and selection problem

Median finding problem:

Given an unordered sequence of  $n$  elements, we want to find the median element.

Generalization: Selection problem:

Given an unordered sequence of  $n$  elements, we want to find the  $k$ -th smallest element.

# Naive solution

Any ideas how to solve the median problem?

## Naive solution

Any ideas how to solve the median problem?

Sort the sequence, and then pick the middle element. Takes  $O(n \log n)$ .

Intuitively, this seems like an overkill! (WHY?)

Can we do better?

TAKE A COUPLE OF MINUTES TO THINK ABOUT A POSSIBLE STRATEGY.

# Generic algorithm, intuition

Idea: divide and conquer

- ▶ At each step in time, select a “splitter element”  $s$  and split the given sequence in two pieces  $S_-$  and  $S_+:$ 
  - ▶  $S_-$  contains all elements that are smaller than  $s$
  - ▶  $S_=$  contains all elements that are equal to  $s$
  - ▶  $S_+$  contains all elements that are larger than  $s$
- ▶ Then, depending on the sizes of  $S_-$ ,  $S_=$  and  $S_+$ , we know in which of the two parts  $S_-$  and  $S_+$  we have to continue our search:

## Generic algorithm, intuition (2)

Sequence 10, 25, 1, 30, 17, 83, 70, 40, 21

Split at 30 :

10, 25, 1, 17, 21 || 30 || 83, 70, 40

Then we know:

- Smallest element must be in  $S_-$
- Second-smallest element must be in  $S_-$
- Median must be in  $S_-$

---

# Generic algorithm, pseudo-code

Select ( $S, k$ )

Choose splitter element  $a_i \in S$

For each  $a \in S$

If  $a < a_i$ , put  $a$  in  $S_-$

If  $a = a_i$ , put  $a$  in  $S_=$

If  $a > a_i$ , put  $a$  in  $S_+$

If  $|S_-| \leq k \leq |S_-| + |S_=|$

Return  $a_i$       The splitter was the desired answer.

If  $|S_-| \geq k$       Solution must be in  $S_-$

Return Select ( $S_-, k$ )

If  $|S_-| + |S_=| < k$       Solution must be in  $S_+$

Return Select ( $S_+, k - |S_-| - |S_=|$ )

## Example

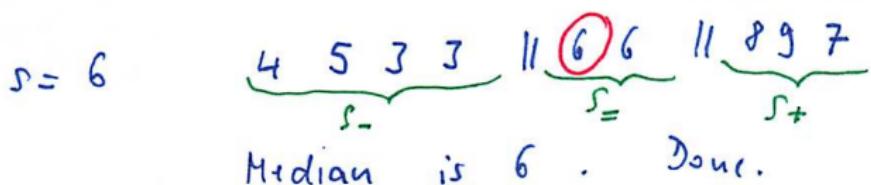
APPLY THE ALGORITHM TO THE FOLLOWING SEQUENCE  
(SELECT YOUR SPLITTER ELEMENTS RANDOMLY):

4 5 6 3 6 3 8 9

## Example (2)

One possible solution (depending on the random splitter elements):

select( 4 5 6 3 6 3 8 9 7 ,  $k=5$  )



## Example (3)

Another possible solution (depending on the random splitter elements):

Select( 4 5 6 3 6 3 8 9 7, k = 5 )

s = 4      3 3 // 4 // 5 6 6 8 9 7

Median has to be in  $S_+$ .

~> Select( 5 6 6 8 9 7, k = 2 )

s = 8      5 6 6 7 // 8 // 9

~> Select( 5 6 6 7, k = 2 )

s = 6 :    5 // 6 6 // 7

~> Solution is 6.

# How to choose a splitter?

WHAT WOULD BE THE BEST SPLITTER ELEMENT?

## How to choose a splitter? (2)

The best situation that could possibly happen is that we are very lucky, and in the first round choose the splitter as the element we are looking for.

Then we are done after this round, and just needed  $n - 1$  comparisons.

But it is unrealistic to hope that this always happens ...

## How to choose a splitter? (3)

WHAT CAN HAPPEN IF WE ALWAYS CHOOSE A BAD SPLITTER ELEMENT?

# How to choose a splitter? (4)

In the worst case:

- ▶ Assume we want to find the median of the sequence.
- ▶ But we have a stupid rule for selecting the splitter, it always chooses the maximum element in the sequence.
- ▶ Then, in each step the size of  $S_-$  decreases by 1. In each round we need to compare against all elements in  $S_i$ . We need to do so until  $S_i$  has size  $n/2$ , then we have found the median.

$$(n - 1) + (n - 2) + \dots + n/2 = \Theta(n^2)$$

number of comparisons.

## How to choose a splitter? (5)

SO WHAT WOULD BE A GOOD STRATEGY TO AVOID THE WORST CASE?

# How to choose a splitter? (6)

- ▶ Try to split the sequence in two equal-sized parts.
- ▶ The corresponding splitter element would be the median ...  
hmmm

# Randomized selection algorithm

We are now going to use the following rule for selecting the splitter:

We pick the splitter element  $s$  randomly from the elements in our list!

We call the resulting algorithm the Randomized selection algorithm.

Intuition why this makes sense:

- ▶ As long as most of the splitter elements are “somewhat from the middle”, we significantly reduce the length of the two sublists.
- ▶ And it is quite likely to choose an element “somewhat from the middle”

# Randomized algorithms

This is our first encounter with a powerful tool, a **randomized algorithm**:

- ▶ A randomized algorithm has access to a random bit generator (a black box mechanism that can produce either 0 and 1 with probability 1/2 each).
- ▶ Generating one random bit costs one unit of time.
- ▶ In this case, the algorithm is a so-called Las-Vegas-algorithm: It is always going to produce the correct result, but the running time is going to depend on “how lucky” we are when drawing the random elements.

# Randomized algorithms (2)

Remarks about the randomees:

- ▶ In practice, the “random operation” is often not a single coin flip. Such operations might be more costly than one time unit:

Example: Draw a number in  $\{1, 2, \dots, n\}$  uniformly at random.

Exercise: How many random coin flips do we need to approximate “uniformly at random” reasonably well?

- ▶ For simple random distributions we can usually ignore the costs of generating such a random number, because we can approximate them by a “small number” of coin flips. If in doubt, you need to double check as in the exercise above.

# Randomized algorithms (3)

- ▶ But if we design algorithms where these probabilities have to be very exact and do not follow a simple distribution, we need to take into account the costs of generating the desired distribution.
- ▶ Another philosophical remark: in practice, we use pseudo-random numbers ...

# Randomized algorithms (4)

For randomized algorithms, we are interested in the **expected running times**.

Be careful, this expectation is with respect to the randomness in the algorithm (not with respect to the randomness in the instance).

- ▶ Expected worst case behavior:

$$\max_{I \in \mathcal{I}_n} E(T(I))$$

“Averages” over the random coin flips, but worst case over the instances  $I$

# Randomized algorithms (5)

- ▶ Expected average case behavior:

$$\frac{1}{n} \sum_{I \in \mathcal{I}_n} E(T(I))$$

“Averages” both over the random coin flips and the instances  $I$ .

- ▶ Worst case behavior:

$$\max_{I \in \mathcal{I}_n} T(I)$$

There exists a sequence of random coin flips and an input sequence  $I$  such that the algorithm needs time  $T(n)$ . No average at all.

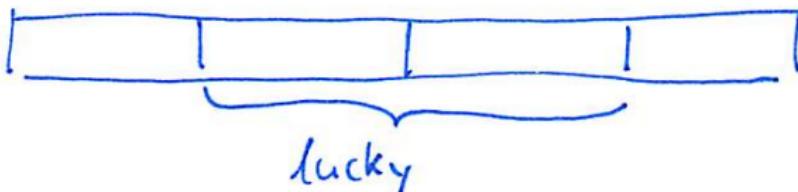
# Expected running time

## Theorem 6 (Expected running time)

On any input sequence of length  $n$ , the expected running time of the randomized median algorithm is  $O(n)$ .

Proof:

We call the splitter element  $s$  “lucky” if it is in the middle 50 % of the (sorted) elements in  $S$ :



# Expected running time (2)

If we would always get a lucky element:

- ▶ If  $s$  is lucky, then both  $S_-$  and  $S_+$  have size at most  $3/4|S|$ .
- ▶ If we were always lucky, then we would have the following recursion for the running time:

$$T(n) \leq T(3/4n) + O(n)$$

- ▶ The first term is the recursive call (after a lucky splitter element)
- ▶ The second term is the amount of work we have to do to split the array in  $S_-, S_+, S_=$ .
- ▶ The master theorem now gives us that  $ET(n) = O(n)$ .

# Expected running time (3)

Now we need to discuss how often we are going to be lucky:

- ▶ If we randomly pick an element of  $S$ , then with probability at least  $1/2$  it is going to be lucky.
- ▶ Assume we repeatedly draw “splitter candidates” from  $S$ . On average, we have to draw twice before we get a lucky one.
  - ▶ With probability  $1/2$ , we draw a lucky element at trial 1.
  - ▶ With probability  $1/2$ , we are not lucky. Then we have already “wasted” one time step, and try again.
  - ▶ Hence, the expected number  $E$  of trials before we get a lucky element satisfies

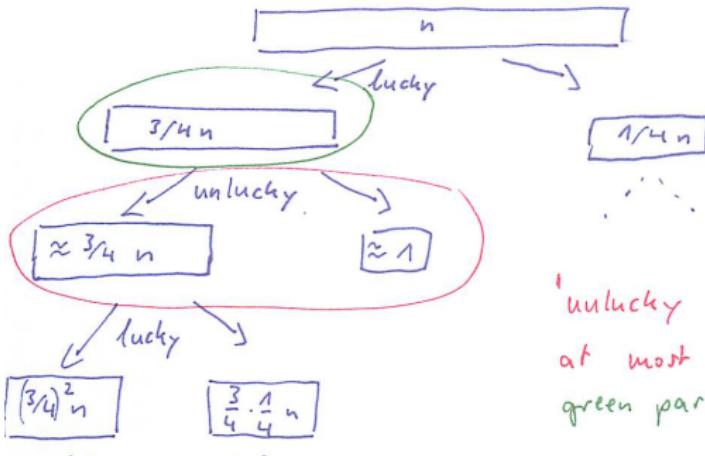
$$E = \frac{1}{2}1 + \frac{1}{2}(E + 1).$$

Solving for  $E$  leads to  $E = 2$ .

# Expected running time (4)

- ▶ So on average, after two split operations the arrays will have size at most  $3/4$  times the original size.
- ▶ In the expected running time, this simply gives a factor of 2:
  - ▶ Each time we are not lucky, we simply have one additional “layer” in the computation tree.
  - ▶ Each of these layers takes as most as much time as the layer above. See Figure next page.

# Expected running time (5)



"unlucky layers", amount of work  
at most as large as in the  
green part.



# Comments

- ▶ Obviously, the algorithm cannot be better than  $O(n)$  (WHY?).
- ▶ It is not surprising that one can solve the problem  $\text{Select}(S,1)$  or  $\text{Select}(S,n)$  in time  $O(n)$  (WHY?)
- ▶ But it is somewhat surprising that we can solve the median problem in  $O(n)$  (IS IT?)

Take this algorithm as an appetizer for the power and beauty of randomized algorithms ☺

# Quicksort

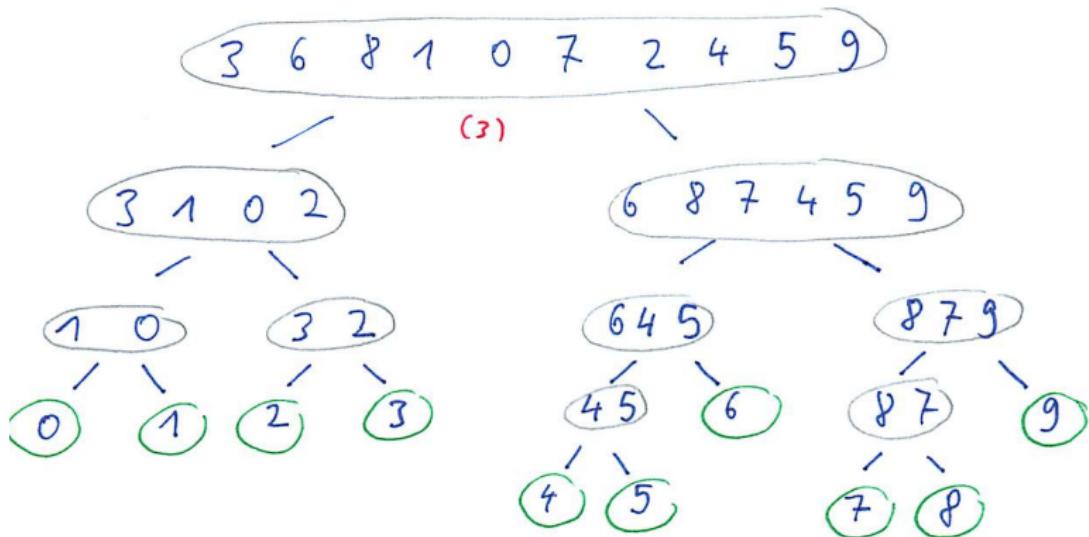
Literature: Mehlhorn Sec. 5.4, Cormen Sec. 7

# Idea

Yet again a divide-and-conquer algorithm, very similar to the median-finding algorithm.

- ▶ Given a sequence, we split it two pieces. We do this in such a way that the elements in the first piece are always smaller than the elements in the second piece. This is the “divide step”.
- ▶ Conquer step: Then we recurse. In the end, we just have to concatenate the resulting sequences. This is the “conquer” step.

## Idea (2)



## Idea (3)

What is called the “splitter element” in the selection-literature is called the “pivot element” in the quicksort literature.

Selecting the pivot element:

- ▶ As in the median finding problem, it is crucial to achieve “lucky splits”: the two resulting sequences should have (approximately) the same length.
- ▶ As in the median finding problem, a very good strategy is to select the pivot element uniformly at random from the elements of the input sequence.

The resulting algorithm is called randomized quicksort.

# Randomized quicksort pseudo-code

**Function**  $\text{quickSort}(s : \text{Sequence of Element}) : \text{Sequence of Element}$

**if**  $|s| \leq 1$  **then return**  $s$

pick  $p \in s$  uniformly at random

$a := \langle e \in s : e < p \rangle$

$b := \langle e \in s : e = p \rangle$

$c := \langle e \in s : e > p \rangle$

**return** concatenation of  $\text{quickSort}(a)$ ,  $b$ , and  $\text{quickSort}(c)$

# Correctness and termination

Clear (REALLY? WHY?)

# Worst case running time

## Theorem 7 ((Randomized) Quicksort worst case)

The worst case running time of (randomized) quicksort is  $\Theta(n^2)$ .

### Proof:

- ▶ To determine the elements of the arrays  $a$ ,  $b$ , and  $c$  needs time  $n$  (we need  $n - 1$  comparisons).
- ▶ Always select the pivot element as the largest element in the sequence.
- ▶ Then the size of the recursive problem has just been reduced by 1, and the height of the recursion tree is  $n - 1$ .
- ▶ So overall, we have  $(n - 1) + (n - 2) + \dots + 2 + 1 = \Theta(n^2)$  comparisons.

## Worst case running time (2)

So the worst-case running time is much slower than for heapsort or merge sort. 😞

# Expected running time

## Theorem 8 (Quicksort expected average time)

For each input sequence that contains  $n$  elements, the expected running time of randomized quicksort is  $O(n \log n)$ .

# Expected running time (2)

Intuition why this is true:

- ▶ Assume that each split is “somewhat balanced”: none of the arrays contains more than, say,  $3/4$  of the elements.
- ▶ Then the computation tree of the algorithm would have height  $\Theta(\log n)$ , and in each level the amount of computation is  $O(n)$  (we need to make at most  $n$  comparisons to the pivots).
- ▶ This would lead to a running time of  $O(n \log n)$ .
- ▶ The trick of the prove is to argue that most of the splits will be balanced indeed, and that the few unbalanced splits don't seriously hurt our running time.

## Expected running time (3)

Formal proof is very much along the line of the median proof:

It is easy to see that the running time of the algorithm just depends on the number of comparisons we make. So we are going to count how many comparisons we make during the algorithm.

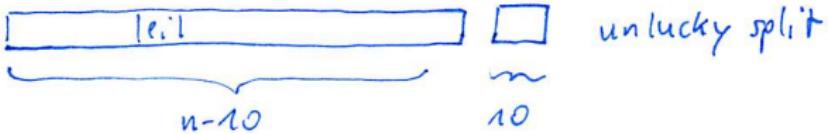
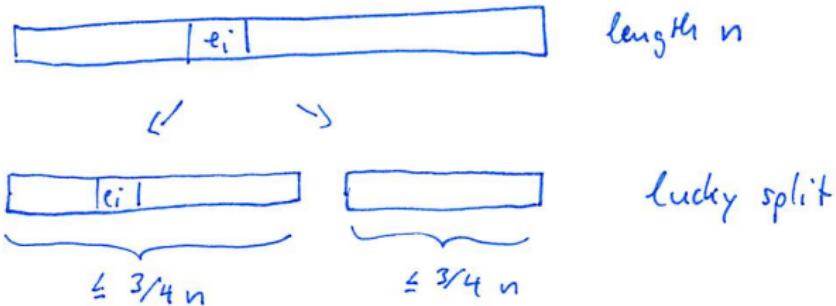
Setup:

- ▶ Fix one element  $e_i$  in the sequence.
- ▶ Denote by  $X_i$  the number of times this element is compared against a pivot element, throughout the whole algorithm.
- ▶ Then  $\sum_{i=1}^n X_i$  is the total number of comparisons.
- ▶ Obviously,  $X_i \leq n - 1$  (after each comparison, the element  $e_i$  ends in a strictly smaller subproblem, so there can only be  $n - 1$  subproblems involving  $e_i$ ).

# Expected running time (4)

Lucky comparisons:

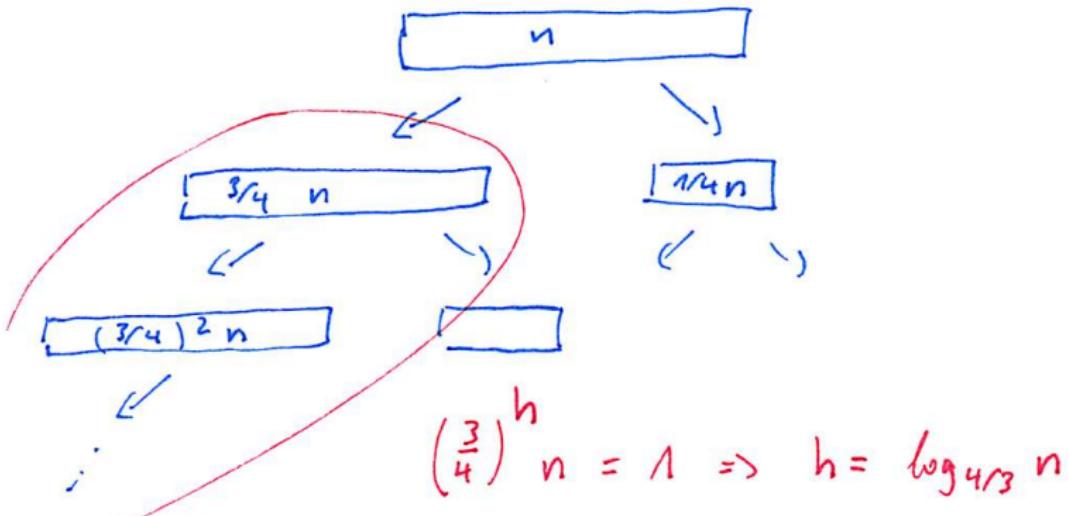
- Call a comparison between  $e_i$  and a pivot element “lucky” if  $e_i$  moves to a subproblem of size at most  $3/4$  the size of the current problem.



# Expected running time (5)

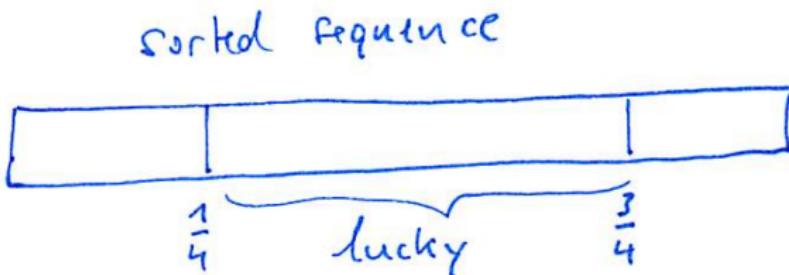
- If  $e_i$  is only involved in lucky comparisons, it needs to go through at most  $\log_{4/3} n$  comparisons.

Worst case if all splits are lucky:



## Expected running time (6)

- When we randomly pick a pivot element from the current elements, the likelihood to be “lucky” is  $1/2$  (just don’t pick one from the first and last quarter).



Half of its elements are lucky.

## Expected running time (7)

- ▶ By the same argument as in the median proof, it takes 2 splits on expectation to reduce the size of an array to  $3/4$  of its size.
- ▶ In each level, we need to perform at most  $n$  comparisons (each element against some pivot).
- ▶ So overall we end up with running time upper bound  $2n \log_{4/3} n = O(n \log n)$ .



## Comments about running time

- ▶ The theorem says “for each input sequence”. This implies that the expected best, average, and worst case running times are all  $O(n \log n)$ .
- ▶ Note that if the sequence contains repeated elements, the running time can be faster.

Extreme example:  $n$  times the same element, then we are done in time  $O(n)$ .

This does not contradict the theorem on the expected running time, which just gives an upper bound.

## Comments about running time (2)

Lower bound:

- ▶ Note that a priori, the  $\Omega(n \log n)$  lower bound for the worst case running time of comparison-based sorting algorithm does not apply to randomized algorithms. At least, our proof did not take randomness into account.
- ▶ However, the theorem and its proof can be generalized to a worst case statement about Las-Vegas randomized algorithms. The argument is as follows:
  - ▶ You can mimic any individual run of a Las-Vegas algorithm by a deterministic algorithm with a given, fixed sequence of coin flips.
  - ▶ For such a deterministic algorithm, the lower bound applies.
  - ▶ Hence, the lower bound applies to all realizations of the Las-Vegas algorithm.

## Comments about running time (3)

Again, note that the lower bound is for the worst case running time (worst instance and worst sequence of coin flips), not for the worst expected running time. **MAKE SURE YOU UNDERSTAND ALL THE DIFFERENCES.**

# Space complexity

- ▶ Quicksort must store a constant amount of information for each nested recursive call (notably, it needs to keep track of the return addresses of the recursive calls on the call stack).
- ▶ In the best case, we have to make  $O(\log n)$  recursive calls. In this case, we need  $O(\log n)$  additional space.
- ▶ In the worst case, we make  $O(n)$  recursive calls. Then we need additional  $O(n)$  space.

## Space complexity (2)

- ▶ There is a trick that can be used to limit the recursion stack of the algorithm to  $O(\log n)$ :
  - ▶ When we process the two recursive calls of quicksort, we always start with the recursively sort call of the smaller partition. The recursion depth here is  $O(\log n)$  (WHY?)
  - ▶ The other call is resolved using tail recursion:  
Procedure calls in the tail position of a procedure are treated as a direct transfer of control to the called procedure, hereby avoiding the need to add a new element to the call stack (intuitively, we replace the recursive call by a “GOTO”).

# Refinements

- ▶ Many more refinements of the quicksort algorithm are used in practice. In particular, it can be done in-place, and we can control the size of the recursion stack.
- ▶ Today, quicksort is the most widely used sorting algorithm in practice.

# History

According to wikipedia, the quicksort algorithm was developed in 1960 in the Soviet Union by Tony Hoare (who was a visitor at that time, and later became a very famous computer scientist who won a large number of prizes, among them the ACM Turing Award).

# Stable sorting algorithms

# Stability

We say that a sorting algorithm is stable if numbers with the same value occur in the output value in the same order as in the input value.

This property is important if we order lexicographically with several keys. Example:

- ▶ We have a data base of customers.
- ▶ We first sort them by their last name.
- ▶ Then we additionally sort by their first name.
- ▶ If the sorting algorithm is stable, the second sorting procedure does not destroy the first sorting.

# Stability (2)

Example:

Input		Sorted by last		Sorted by first	
Last name	First name	Last	First	Last	First
Newman	Alice	Blue	John	Blue	John
Smith	John	Newman	Alice	Newman	Alice
Blue	John	Smith	John	Smith	Bob
Smith	Bob	Smith	Bob	Smith	John

- ▶ In the second column, John Smith is before Bob Smith because this also was the case in the first column.
- ▶ In the third column, the sorting did not destroy the ordering by last name, it just additionally sorted by the first name. This only works if the sorting algorithm is stable (MAKE SURE YOU UNDERSTAND WHY).

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Insertion sort:

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Insertion sort: not stable

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Insertion sort: not stable
- ▶ Selection sort:

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Insertion sort: not stable
- ▶ Selection sort: are not

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Insertion sort: not stable
- ▶ Selection sort: are not
- ▶ Heap sort:

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Insertion sort: not stable
- ▶ Selection sort: are not
- ▶ Heap sort: not stable
- ▶ Merge sort:

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Insertion sort: not stable
- ▶ Selection sort: are not
- ▶ Heap sort: not stable
- ▶ Merge sort: stable

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Insertion sort: not stable
- ▶ Selection sort: are not
- ▶ Heap sort: not stable
- ▶ Merge sort: stable
- ▶ Quicksort:

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Insertion sort: not stable
- ▶ Selection sort: are not
- ▶ Heap sort: not stable
- ▶ Merge sort: stable
- ▶ Quicksort: not stable

# Stability (3)

Which algorithms are stable (in their “natural” implementation)?

- ▶ Insertion sort: not stable
- ▶ Selection sort: are not
- ▶ Heap sort: not stable
- ▶ Merge sort: stable
- ▶ Quicksort: not stable

# Sorting in linear time

Literature:

Mehlhorn Sec. 5.6, Cormen Sec. 8.2-8.4

# Motivation

Have seen:

- ▶ Comparison-based sorting has lower bound  $\Omega(n \log n)$ .
- ▶ If we don't have any further information about our input, it seems hardly possible to come up with a sorting algorithm that is not comparison-based.
- ▶ However, in the following sections we will see how to break the lower bound if we make further assumptions on the input data.

# Counting sort / Ksort

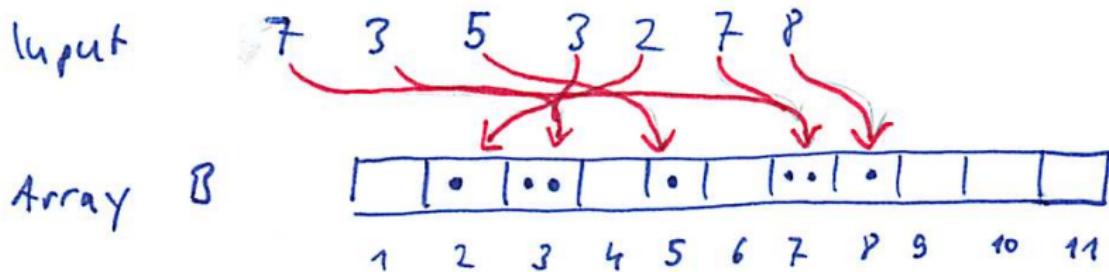
# Idea

Assume that the numbers to be sorted are integers in a fixed range, say between 0 and  $K$ .

- ▶ Provide an array  $B$  of  $K$  “buckets”
- ▶ Walk along the input sequence. Upon reading element  $A(i) =: k$ , increase a counter for bucket  $B(k)$ .
- ▶ In the end, just walk along  $B$  and collect the elements.

## Idea (2)

Counting sort:



Output      2 3 3 5 7 7 8

## Idea (3)

- ▶ More generally, assume you want to sort larger structs according to some key value.
- ▶ Assume that the key value is an integer between 0 and  $K$ .
- ▶ Provide an array of buckets again. The contents in each bucket are a linked list of elements.
- ▶ Whenever you encounter a struct with key value  $k$ , concatenate it to the list in bucket  $B(k)$ .
- ▶ In the end, concatenate the non-empty buckets.

# Pseudo-code

**Procedure**  $K\text{Sort}(s : \text{Sequence of Element})$

$b = \langle \langle \rangle, \dots, \langle \rangle \rangle : \text{Array } [0..K - 1] \text{ of Sequence of Element}$

**foreach**  $e \in s$  **do**  $b[\text{key}(e)].\text{pushBack}(e)$

$s := \text{concatenation of } b[0], \dots, b[K - 1]$

# Properties

Is counting sort stable?

# Properties

Is counting sort stable?

Yes

# Running time

Running time (worst, best, average) is  $O(n + K)$

- ▶ Allocate array  $C$  with  $K$  slots:  $O(K)$
- ▶ Read the sequence and each time update one of the counters / lists:  $O(n)$

It can beat the  $n \log n$  bound because the algorithm is not comparison-based. Such an algorithm is possible because we make additional assumptions on the input.

# Radix sort

# Motivation

Assume that we still have to sort integers, but the upper bound  $K$  can be huge (in particular,  $K = \omega(n)$ ).

Idea is now:

- ▶ Treat each integer as a string
- ▶ First sort the strings according to the last digit (least significant digit) using ksort, then sort them according to the second last digit using counting sort, and so on.
- ▶ Because ksort is stable, this is going to lead to a sorted sequence of integers.

# Radix sort, pseudo-code

Let  $d$  be the maximal length of a digit in the input sequence.

RADIX-SORT( $A, d$ )

- 1 **for**  $i = 1$  **to**  $d$
- 2     use a stable sort to sort array  $A$  on digit  $i$

The name:

- ▶ Radix notation = “Stellwert-Schreibweise”
- ▶ Intuitively, radix sort means “Fächer-Sortieren”

# Running time

When sorting integers:

- ▶ Digits can be in the range  $0, 1, \dots, 9$ .
- ▶ On each digit, we spend time  $\Theta(n + 10) = \Theta(n)$
- ▶ So overall, we spend time  $\Theta(dn)$ .

More generally:

- ▶ each key can have up to  $K$  different values
- ▶ We have  $d$  different keys to use
- ▶ Then we obtain a running time of  $O(d(n + K))$ .
  - ▶ Example:  $d = O(1)$ ,  $K = O(n) \implies$  running time  $O(n)$
  - ▶ Example:  $d = O(\log n)$ ,  $K = O(n)$ , then running time  $O(n \log n)$ .

This case applies if we sort entries in a data base and the data base has no duplicates. Reason: to express  $n$  distinct elements with keys in the range  $0, \dots, K$ , we need to use  $d \geq \log_K n$  keys.

## A variant

We could also start radix sort with the largest (most significant) digit.

Even though it sounds like a minor change in the algorithm, it can lead to a disaster in performance (unless one is particularly careful). See exercises ...

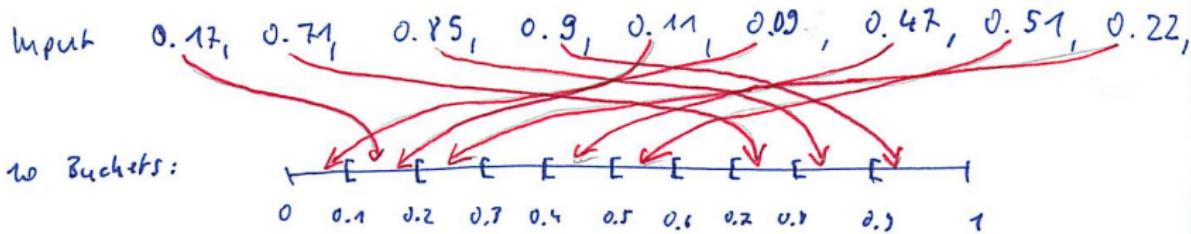
# Bucket sort / uniform sort

# Idea

- ▶ Assume that we want to sort reals that are uniformly distributed in  $[0,1]$ .
- ▶ Assume that we are given an input array of  $n$  elements.
- ▶ We split the interval  $[0,1]$  in  $n$  buckets, where bucket  $i$  is going to contain all keys from  $[i/n, (i+1)/n[$ .
- ▶ By assumption, we expect that most of the buckets only contain few elements. We sort the elements in each bucket by a naive sorting procedure, and then concatenate the buckets.

# Idea (2)

Bucket sort:



sort by  
in which  
sort:

# Pseudo-code

BUCKET-SORT( $A$ )

- 1 let  $B[0..n - 1]$  be a new array
- 2  $n = A.length$
- 3 **for**  $i = 0$  **to**  $n - 1$ 
  - 4 make  $B[i]$  an empty list
- 5 **for**  $i = 1$  **to**  $n$ 
  - 6 insert  $A[i]$  into list  $B[\lfloor nA[i] \rfloor]$
- 7 **for**  $i = 0$  **to**  $n - 1$ 
  - 8 sort list  $B[i]$  with insertion sort
- 9 concatenate the lists  $B[0], B[1], \dots, B[n - 1]$  together in order

# Running time

## Theorem 9 (Bucket sort running time)

If the keys are uniformly distributed in  $[0, 1[$ , then bucket sort has average running time  $O(n)$  and worst case running time  $O(n \log n)$ .

**Proof worst case:** In the worst case, all elements end up in the same bucket. Then we need to sort that bucket by a standard sorting algorithm (without particular assumptions), which needs  $O(n \log n)$ .

# Running time (2)

## Proof average case.

- ▶ Time for setting up the buckets and concatenating the sorted buckets is always  $O(n)$ .
- ▶ Additionally, we have to spend time  $T_i$  for sorting the elements in bucket  $i$ .
- ▶ So the expected running time on instances with  $n$  elements satisfy:

$$ET = O(n) + E \sum_{i=0}^{n-1} T_i$$

(the expectation is over the random drawing of an instance).

## Running time (3)

- Because all buckets have the same size and the input sequence is uniformly distributed, we have  $ET_i = ET_0$  for all  $i$ :

$$ET = O(n) + nET_0$$

- Now we show that even if we use insertion sort to sort the elements in bucket  $i$  we still get time  $O(n)$  in the end. With insertion sort,

$$ET = O(n) + n(EB_0)^2$$

where  $B_0$  is the number of elements in bucket 1.

# Running time (4)

- ▶ Note that  $B_0$  is binomially distributed with  $p = 1/n$ . Standard probability theory arguments then give:

$$\text{prob}(B_0 = i) = \binom{n}{i} \frac{1}{n^i} \left(1 - \frac{1}{n}\right)^{n-i} \leq \frac{n^i}{i!} \frac{1}{n^i} \leq \frac{1}{i!} \leq \left(\frac{e}{i}\right)^i$$

$$\begin{aligned} \mathbb{E}[B_0^2] &= \sum_{i \leq n} i^2 \text{prob}(B_0 = i) \leq \sum_{i \leq n} i^2 \left(\frac{e}{i}\right)^i \\ &\leq \sum_{i \leq 5} i^2 \left(\frac{e}{i}\right)^i + e^2 \sum_{i \geq 6} \left(\frac{e}{i}\right)^{i-2} \\ &\leq \mathcal{O}(1) + e^2 \sum_{i \geq 6} \left(\frac{1}{2}\right)^{i-2} = \mathcal{O}(1) \end{aligned}$$

- ▶ So we end with  $ET = O(n)$ .



# Generalization of bucket sort

Note that we might be able to use bucket sort more generally:

- ▶ Assume we know the distribution of keys (but it is not necessarily uniform).
- ▶ All we need to do is to come up with a rule to determine the boundaries of the  $n$  buckets so that the probability mass of each bucket is  $1/n$ .

# Sorting: summary and state of the art

# Overview table

	Time			Space	
	Best	Avg	Worst	Av	Worst
Quicksort	$n \log n$	$n \log n$	$n^2$	$\log n$	$n$
Merge sort	$n \log n$	$n \log n$	$n \log n$		$(n)$
lheap sort	$n \log n$	$n \log n$	$n \log n$		1
Inversion sort	$n$	$n^2$	$n^2$		1
Selection sort	$n^2$	$n^2$	$n^2$		1
Bubble sort	$n$	$n^2$	$n^2$		1

## Overview table (2)

- ▶ Looking at the asymptotic running times does not reveal many differences.
- ▶ In practice, the size of the constants are very important.

## Overview table (3)

In practice:

- ▶ For small instances, insertion sort is used.
- ▶ For larger instances, quicksort (with many refinements) is popular.
- ▶ Often, hybrid algorithms are used that try to combine the advantages of different approaches.

Example: Tim sort, published 2002:

- ▶ Hybrid algorithm with elements from merge sort and insertion sort
- ▶ Exploits carefully the situation that pieces of the input are already sorted
- ▶ Now the standard algorithm in Python, Java SE 7, on the Android platform, and in GNU Octave.

# Other properties to look at

We have seen the following properties:

- ▶ Running times (worst, best, average)
- ▶ Space complexity

There might also be other properties that are important (but we won't discuss them):

- ▶ Online application (data arrives in a stream, and we need to sort each new data point immediately to its correct position)
- ▶ Extremely limited capacity (e.g., on limited mobile devices, on particular sensors, ... )

# High-level summary: lessons about algorithm development

## Very high level lesson:

- ▶ The same problem can have a wide variety of algorithms, which all differ in running times, space complexity, ...

## Divide and conquer:

- ▶ Often it is a good idea to use divide and conquer strategies and divide your problem in smaller subproblems.  
Example: merge sort, quick sort.
- ▶ Ideally, the subproblems should become considerably smaller in each step. If you manage to reduce the problem size by at least a constant factor, the height of the recursion tree is only logarithmic.
- ▶ Divide and conquer does not always automatically reduce the running time (Example: naive recursive digit multiplication). Sometimes you have to be clever...
- ▶ To analyze the recursion formulas for the running times, the master theorem is often a good tool.

## Use of randomized algorithms:

- ▶ You need to select a “typical” element, but you don't know how to achieve it in a deterministic way. In this situation consider drawing it randomly.
- ▶ If the “good event” that you pick a typical element happens with high enough probability, then your randomized algorithm is going to work fine.
- ▶ Randomized algorithms are often very simple, but their theoretical analysis can be very difficult.

# Searching

# Introduction: The searching problem

Example:

- ▶ All books in a library
- ▶ Bookings for an airline
- ▶ keywords for google queries



Image: Wikipedia

## Introduction: The searching problem (2)

Goal is now:

- ▶ Want to construct efficient data structures to maintain a sorted sequence of elements.
- ▶ Each element has a key value, and elements should be sorted by the key value.
- ▶ We want to have the following elementary operations:
  - ▶ Find an element with a particular key value
  - ▶ Insert a new element to the correct place
  - ▶ Remove an element

# Binary Search

Mehlhorn Sec. 2.5

## Binary search, intuition

- ▶ Let  $A$  be an array with  $n$  elements that are sorted in increasing order.
- ▶ Given a query  $q$ , our goal is to find the (smallest) index  $i$  such that  $A[i] \leq q \leq a[i + 1]$ .
- ▶ Depending on the application, if the query is not in  $A$  yet, find the (smallest) index  $i$  such that if we insert the query element after  $i$ , the array remains sorted.

## Binary search, intuition (2)

Intuitively, what we do is the following:

- ▶ Consider the item in the middle, element  $A[n/2]$ .
  - ▶ If it is smaller than  $q$ , we know that we need to continue search in the right half.
  - ▶ If it is larger than  $q$ , we continue to the left half.
- ▶ and so on ..

## Binary search, pseudo-code

Binary Search, Scenario 1:

- ▶ If query element  $q$  in the array, return any index with  $A[i] = q$
- ▶ If query element not in the array, return NotFound.

## Binary search, pseudo-code (2)

```
1 BinarySearch(A[0..N-1], value) {
2     low = 0
3     high = N - 1
4     while (low <= high) {
5         // invariants: value > A[i] for all i < low
5         // invariants: value < A[i] for all i > high
6         mid = (low + high) / 2    // round to lower integer
7         if (A[mid] > value)
8             high = mid - 1
9         else if (A[mid] < value)
10            low = mid + 1
11     else
12         return mid
13 }
14 return not_found // value would be inserted at index "low"
```

strict inequalities

## Binary search, pseudo-code (3)

Let's run this algorithm on a couple of examples:

- ▶ Search for 17 in the following array: 1 3 5 17 18 20 26 40
- ▶ Search for 16 in the same array
- ▶ Search for 50 in the same array
- ▶ Search for 17 in the following array: 1 17 17 17 17 17 17 20

## Analysis: Termination

DOES THE ALGORITHM ALWAYS TERMINATE? WHY?

## Analysis: Termination (2)

Formal proof:

- ▶ We need to argue that the while loop ends.
- ▶ Assume we are in iteration  $i$  of the while loop. At the beginning of the while loop we have  $low \leq high$  (otherwise we would not have entered the while loop).
- ▶ For this reason, we always have  $low \leq mid \leq high$  (line 5).
- ▶ If the algorithm does not return a number (line 11), then the while loop either decreases high (line 7) or increases low (line 9). Hence, in each iteration of the while loop the difference  $high - low$  decreases by at least 1.
- ▶ So after at most  $n$  iterations of the while loop,  $high - low < 0$  and the loop terminates.

## Analysis: Correctness

DOES THE ALGORITHM ALWAYS RETURN A CORRECT  
( $=$ scenario 1) RESULT?

## Analysis: Correctness (2)

Yes, here comes the **Proof**:

Obviously, if the algorithm returns “mid” (line 11), then  $A[mid] = value$ , which is a correct result.

Need to prove: if  $value$  exists in the array, then the algorithm does not return with line 13, but with line 11.

First step: It is straight forward to see that the following invariances are always maintained:

- ▶  $value > A[i]$  for all  $i < low$  (strict inequality!)
- ▶  $value < A[i]$  for all  $i > high$  (strict inequality!)

## Analysis: Correctness (3)

Now we argue as follows: Assume *value* is contained in the array.

- ▶ Invariances are true. This means that at no point of the algorithm, the element we are looking for could be left of *low* or right of *high*.
- ▶ The only way we could potentially “miss” the array element is if either line 7 or line 9 lead to the situation  $low > high$  (because then we leave the while loop).
  - ▶ By construction we always have  $low \leq mid \leq high$  after line 7 has been executed.
  - ▶ As long as  $high \geq left + 2$ , we always have  $low < mid < high$ . In this situation,  $mid - 1 \geq low$  and  $mid + 1 \leq high$ , so we still have  $low \leq high$  after lines 7 or 9 have been executed.

## Analysis: Correctness (4)

The two critical cases are:

- ▶  $low = high$ . Then also  $mid = low$ . By assumption,  $value$  is in the array, so it has to be  $A[low]$ . And this is also what is going to be returned by line 11.
- ▶  $high = low + 1$ . Then  $mid = low$  (because we take the floor when computing  $mid$ ).

Now either  $A[low] = A[mid] = value$ , in which case we return the correct result in line 11.

Or  $A[low] = A[mid] < value$ , in which case we increase  $low$  by 1, then ending up in case  $low = high$  in the next iteration of the loop. This one is going to give the correct answer, as we have already seen.



## Analysis: Running time

WHAT IS YOUR GUESS?

## Analysis: Running time (2)

### Proposition 10

Binary search has running time  $O(\log n)$ .

Proof:

- ▶ 2 Operations before the while loop
- ▶ Number of iterations of the while loop:
  - ▶ If  $n$  is odd, then the subarrays  $A[low, \dots, mid]$  and  $A[mid, \dots, high]$  contain  $(n - 1)/2 \leq n/2$  elements each.
  - ▶ If  $n$  is even, then the subarrays contain  $n/2 - 1$  and  $n/2$  elements each.
  - ▶ The search ends at latest if the array only contains one element. This is the case after  $k = \log n$  iterations.
- ▶ Within each while loop, we need 4 basic operations.  
So all in all we end up with  $4 \log n + 2 = O(\log n)$  basic operations.

## Variant: Leftmost index

If *value* does not exist in the list, we want to know the leftmost index *i* such that if we insert the value after position *i* to the array, the array is still sorted.

```
1 BinarySearch_Left(A[0..N-1], value) {
2     low = 0
3     high = N - 1
4     while (low <= high) {
5         // invariants: value > A[i] for all i < low
5         //           value <= A[i] for all i > high
6         mid = (low + high) / 2
7         if (A[mid] >= value)
8             high = mid - 1
9         else
10            low = mid + 1
11     }
12     return low
13 }
```

## Variant: Leftmost index, recursively

Obviously, one can also implement binary search recursively, for example:

```
// initially called with low = 0, high = N - 1
BinarySearch_Left(A[0..N-1], value, low, high) {
    // invariants: value > A[i] for all i < low
                  value <= A[i] for all i > high
    if (high < low)
        return low
    mid = (low + high) / 2
    if (A[mid] >= value)
        return BinarySearch_Left(A, value, low, mid-1)
    else
        return BinarySearch_Left(A, value, mid+1, high)
}
```

## Variant: Exponential search

Assume you don't know how large the array is (but it is still sorted).  
Then:

- ▶ Compare the input query against the values with index  $1 = 2^0, 2 = 2^1, 2^2, 2^3, 2^4, \dots$  until you hit an index  $i$  with  $query < A[2^i]$  for the first time.
- ▶ Then you start a binary search with  $start = 2^{i-1}$  and  $2^i$ .

Easy to prove:

- ▶ Algorithm always terminates and gives the correct result.
- ▶ If  $A[m] \leq query \leq A[m + 1]$ , then the running time of this algorithm is still  $O(\log m)$ .

## Remarks

- ▶ Conceptually, binary search is very easy.
- ▶ But it is very easy to make mistakes in the pseudo-code ( $<$  instead of  $\leq$  or stuff like this).
- ▶ The only way to find out whether your code is correct is to formally argue using invariances.

## Remarks (2)

Assumptions:

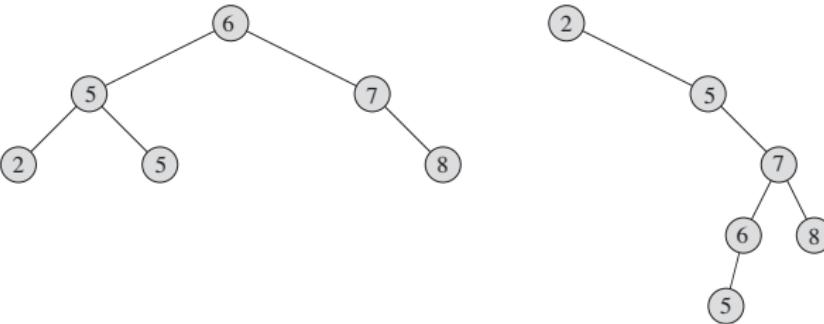
- ▶ Binary search only works if the input array is indeed sorted.
- ▶ If this condition is violated, no guarantees whatsoever can be given.
- ▶ In particular, it is also impossible to check the condition that the array is sorted in logarithmic time.

# Binary search trees: basics

Literature: Cormen 12, (Mehlhorn 7)

# Search tree: definition

- ▶ Each element consists of four things: key value, left pointer, right pointer, parent pointer
- ▶ Vertices are arranged as a binary tree.
- ▶ The following property is satisfied for all vertices:  
If  $y$  is a vertex in the tree, then
  - ▶ all vertices  $z$  in the left subtree satisfy  $\text{key}(z) \leq \text{key}(y)$
  - ▶ all vertices  $z$  in the right subtree satisfy  $\text{key}(z) \geq \text{key}(y)$



# Searching an element

Want to search whether a particular key is somewhere in the tree.  
ANY IDEAS???

# Searching an element (2)

Informal solution:

- ▶ Start at the top
- ▶ At every node  $y$ :
  - ▶ if  $\text{key}(y)$  is smaller than what we are looking for, go left
  - ▶ If it is larger, go right
  - ▶ if it is equal, the stop

# Searching an element (3)

TREE-SEARCH( $x, k$ )

- 1   **if**  $x == \text{NIL}$  or  $k == x.key$
- 2       **return**  $x$
- 3   **if**  $k < x.key$
- 4       **return** TREE-SEARCH( $x.left, k$ )
- 5   **else return** TREE-SEARCH( $x.right, k$ )

## Searching an element (4)

This operation takes  $O(h)$  steps where  $h$  is the height of the tree.

# Extracting minimum / maximum element from the tree

WHAT DO WE NEED TO DO TO FIND THE MINIMUM (MAXIMUM) ELEMENT IN THE TREE?

RUNNING TIME?

# Extracting minimum / maximum element from the tree (2)

Minimum:

- ▶ Start in the root, always walk to the left child, until you reach a leaf.
- ▶ This leaf is the minimal element in the search tree.
- ▶ Running time is  $O(h)$  where  $h$  is the height of the tree

## Find the successor

Given an element  $y$  with key value  $k(y)$  in the tree, want to find the next larger element in the tree.

Formally: want to find  $\min\{k(z) \mid k(z) \geq k(y), z \neq y\}$

ANY IDEA?

## Find the successor (2)

Just find the minimal element in the right subtree of  $y$

$\leadsto$  running time  $O(h)$

Similarly for the predecessor (max. element in left subtree).

# Output all vertices as ordered list

Given a search tree, want to write all its elements into an array such that the values are ordered.

ANY IDEAS?

## Output all vertices as ordered list (2)

INORDER-TREE-WALK( $x$ )

- 1   **if**  $x \neq \text{NIL}$
- 2       INORDER-TREE-WALK( $x.\text{left}$ )
- 3       print  $x.\text{key}$
- 4       INORDER-TREE-WALK( $x.\text{right}$ )

# Output all vertices as ordered list (3)

**Correctness:** by induction over the number  $n$  of vertices in the tree

- ▶ Base case:  $n = 1$  (just the root): clear
- ▶ Induction step from  $n$  to  $n + 1$ :
  - ▶ Call the procedure at the root
  - ▶ By induction hypothesis, the left and right subtree will be printed in the correct order
  - ▶ By definition of the search tree, if we first print the left subtree, then the root, and then the right subtree, then this is going to be ordered correctly.



# Output all vertices as ordered list (4)

RUNNING TIME?

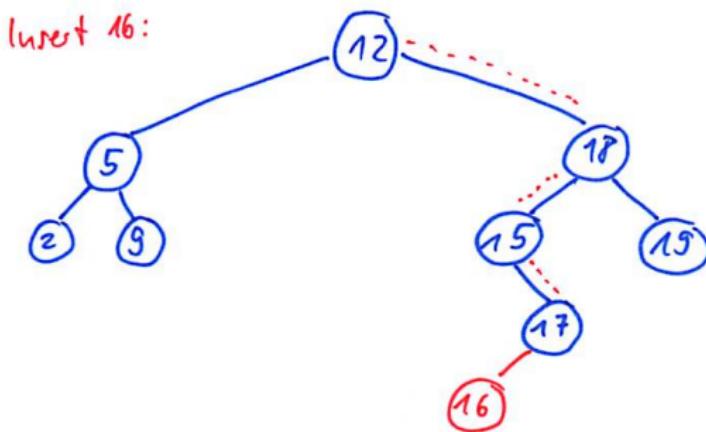
# Output all vertices as ordered list (5)

Running time:

- ▶ We visit each vertex exactly once.
- ▶ In each vertex, the actual work we have to do is:
  - ▶ to print the key of this vertex
  - ▶ call two subroutines
- ▶ So the running time is  $\Theta(n)$ .

# Inserting elements

Simply walk down the tree in the obvious way and insert a new leaf:



Obviously takes  $O(h)$  time as well.

# Deleting an element: three cases

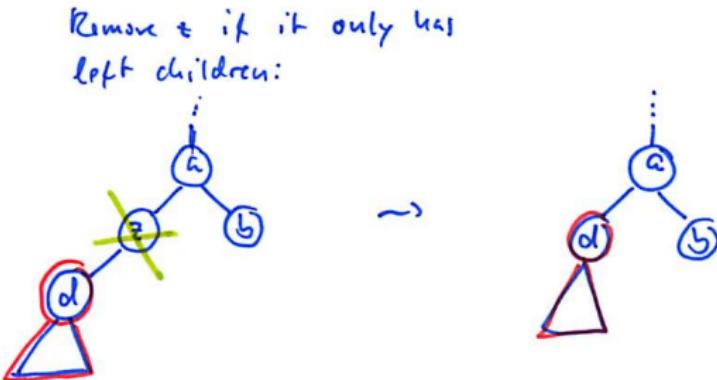
Case 1: Element has no children. Simply remove vertex.



if  $z$  has no children

## Deleting an element: three cases (2)

Case 2: Element has only left (or only right) children. Simply move them one level up.

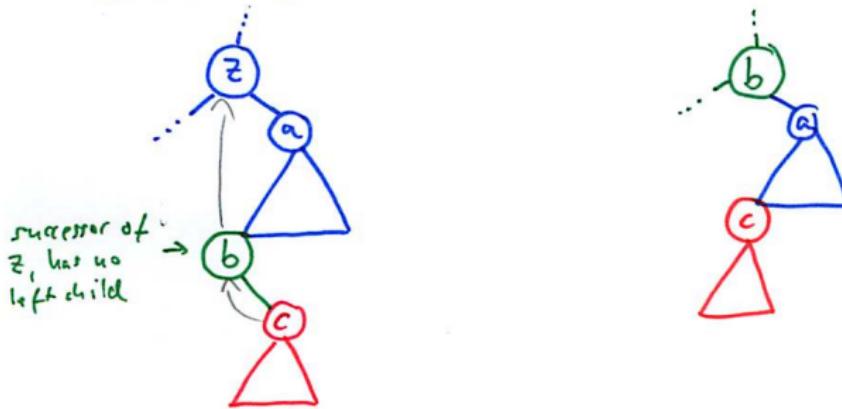


# Deleting an element: three cases (3)

General case: want to remove an interior vertex  $z$ .

- ▶ Find the successor  $b$  of  $z$  (the left most vertex in the right subtree).  
Note that the successor itself has no left child.
- ▶ Replace  $b$  by its right child  $c$  and  $z$  by  $b$ .

Remove  $z$  (general case):



# Deleting an element: three cases (4)

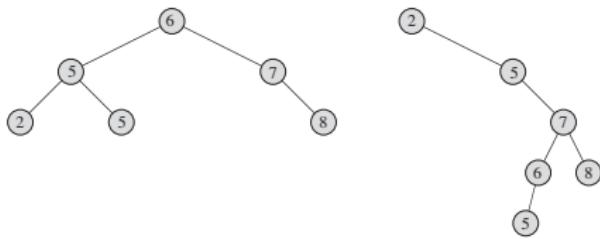
Running time:

- ▶ Finding the successor needs  $O(h)$
- ▶ the rest happens in constant times (we just have to reset a couple of pointers).
- ▶ So overall:  $O(h)$ .

# Balanced trees?

Have seen: all basic tree operations can be done in  $O(h)$  time where  $h$  is the height of the tree.

Obviously it would be nice to have trees with height as small as possible  $\rightsquigarrow h \approx \log n$ .



Here, both trees contain the same values, but the left one has a smaller height

# Balanced trees? (2)

## EXERCISES:

- ▶ Describe a sequence of  $n$  input points for which the search tree has height  $n - 1$ .

# Balanced trees? (3)

Balanced trees, two options:

- ▶ Don't do anything, but hope that the search tree won't be too imbalanced.
  - ▶ For example, one can prove that if we start with an unsorted array, then the average height of the corresponding search tree is  $O(\log n)$  (where the average is over all possible permutations of the array). ☺(Proof: see Cormen Sec. 12.4)
  - ▶ However, little is known if we mix delete / insert operations, no average case analysis ... ☹
- ▶ Apply rebalancing operations to maintain a balanced tree.
  - ▶ See next section.

## AVL Trees

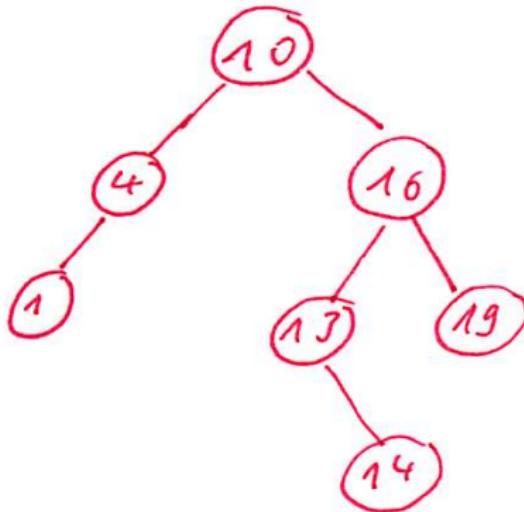
Not in any of the standard text books we use 😐

Section 5.2.1. of Ottmann / Widmeyer: Algorithmen und Datenstrukturen.

Original paper: **A**delson, **V**elskii, **L**andis: An algorithm for the organization of information. Soviet Mathematics Doklady, 1962.

## Balanced search tree

We say that a **binary search tree is balanced** if at every node, the height of the left and right subtree differs by at most 1.



# Height of a balanced search tree

## Proposition 11

The height of a balanced search tree with  $n$  vertices is  $O(\log n)$ .

### Proof.

- ▶ Assume we are given a balanced tree of height  $h$ .
- ▶ Denote by  $N(h)$  the minimal number of vertices in a balanced binary tree of height  $h$ .
- ▶ The root has two subtrees.
  - ▶ One of the subtrees has to have height  $h - 1$  (otherwise the whole tree would not have height  $h$ )
  - ▶ Then by the balancing condition, the other subtree has height at least  $h - 2$ .

## Height of a balanced search tree (2)

- So we get

$$N(h) = N(h - 1) + N(h - 2) + 1$$

DOES IT RING A BELL?

## Height of a balanced search tree (3)

Looks pretty much like the Fibonacci sequence ...

We immediately see:

$$N(h) > F_h$$

where  $F_h$  is the  $h$ -th Fibonacci number. Thus by our knowledge on  $F_h$ :

$$N(h) > F_h \geq 2^{h/2} \implies h \leq 2 \log(N)$$

## Height of a balanced search tree (4)

Alternative analysis, by foot:

$$\begin{aligned}N(h) &= \underbrace{N(h-1)}_{=N(h-2)+N(h-3)+1} + N(h-2) + 1 \\&> 2N(h-2)\end{aligned}$$

Applying the same trick recursively leads to

$$N(h) > 2^2 N(h-4) > 2^3 N(h-6) \dots > 2^{h/2}$$

which implies

$$h < 2 \log N(h)$$



## AVL tree Intuition

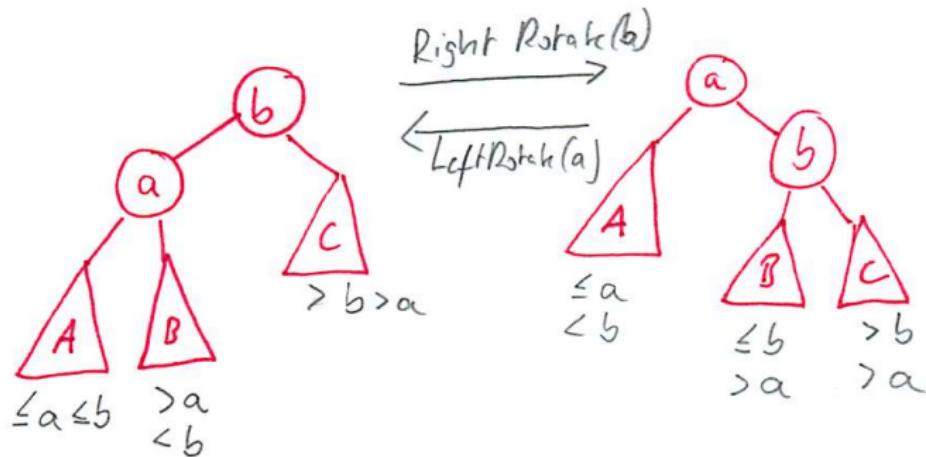
For searching, a balanced tree is pretty much optimal:

- ▶ Minimal height of a binary search tree is achieved for a full tree,  $h = \log n$ .
- ▶ A balanced search tree satisfies  $h \leq 2 \log n$ , which is nearly optimal (up to the factor of 2).
- ▶ We now want to maintain a balanced search tree.

WHICH ARE THE DIFFICULT OPERATIONS?

## Basic operation: rotation

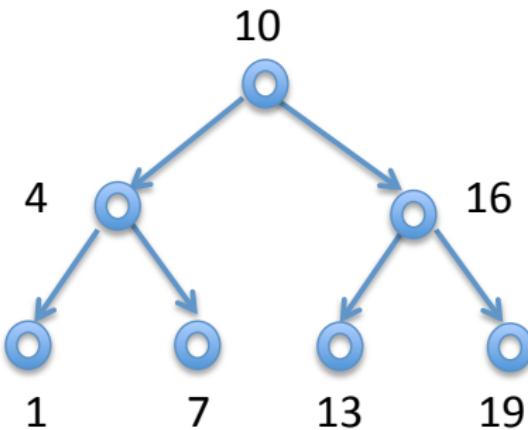
- ▶ Keeps the search tree property intact
- ▶ Changes the “balance” of the two subtrees



Note that one rotation operation just needs  $O(1)$  time.

## Inserting an element

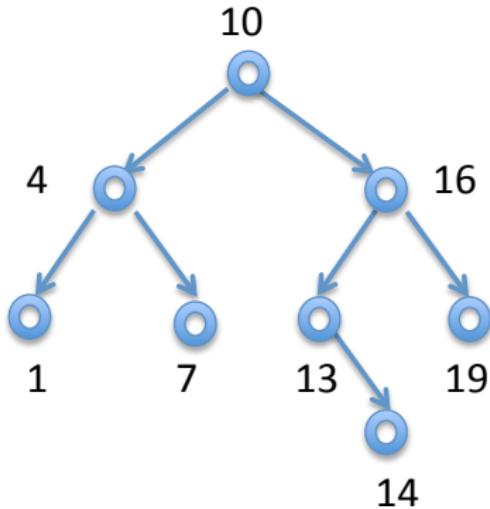
Start with a properly balanced tree.



- ▶ First insert an element “as usual”
- ▶ Now walk up towards the root and check whether the balancing conditions are still satisfied.

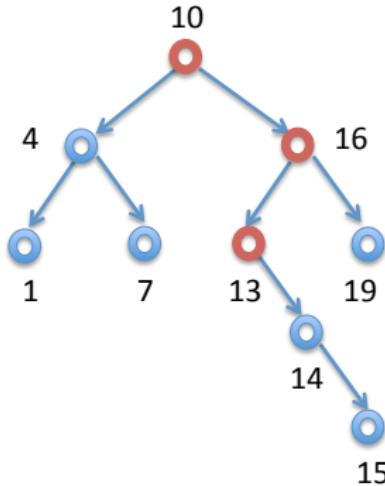
## Inserting an element (2)

**Good case:** all vertices up to the top are balanced. Then we are happy.



## Inserting an element (3)

**Bad case:** When we walk up towards the root, we encounter a vertex which is not balanced, that is we have height difference 2 between left and right subtree.



We want to fix this using rotations. Will see: two balancing operations are always enough.

## Fixing imbalance at one vertex

- ▶ Walk up the path from the inserted vertex to the root.
- ▶ Stop at the first imbalanced vertex.
- ▶ Now we can be in two different scenarios.

# Fixing imbalance at one vertex (2)

## Situation 1:

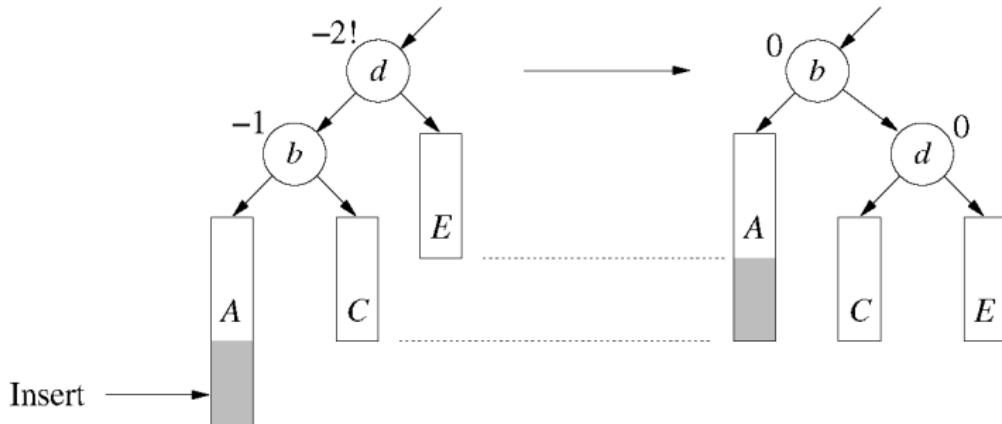


Figure 22: Single rotation.

# Fixing imbalance at one vertex (3)

## Situation 2:

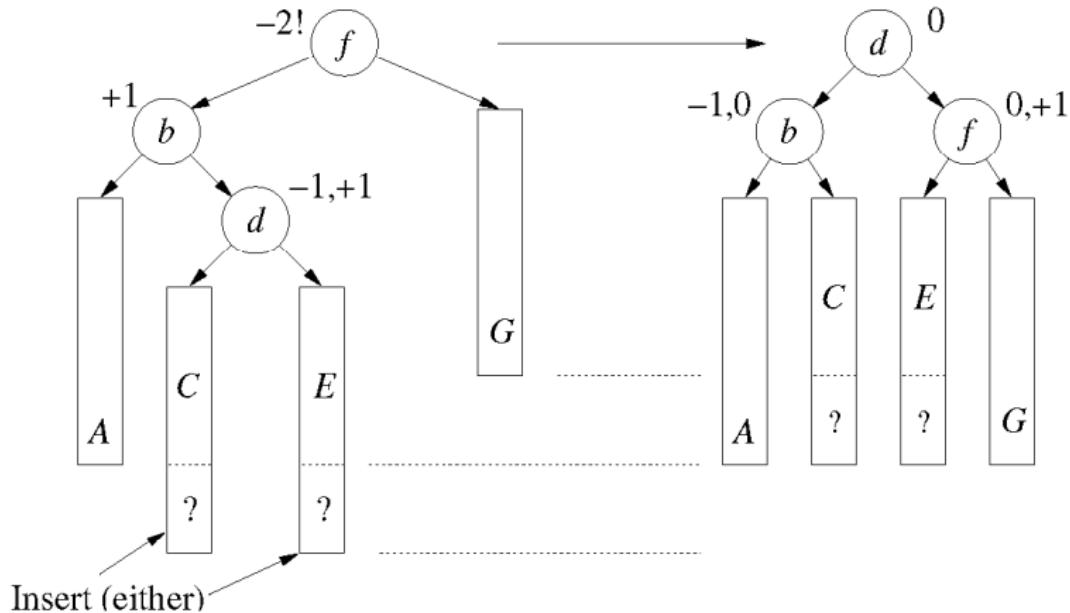


Figure 23: Left-right double rotation.

## Fixing the other imbalances

By the rotation operations just discussed before, we can fix the imbalance at the first imbalanced vertex we encounter.

Somewhat surprisingly, this also fixes any other imbalance that could have been on the path towards the root:

- ▶ Let  $h$  be the height of the subtrees below  $b$  before insertion. Then after insertion and rotation, the subtrees have again height  $b$ . So for any vertex that has  $b$  as child, the height differences did not change.
- ▶ Similarly, the height of the tree below  $d$  is the same before insertion and after insertion and rotation.
- ▶ So we do not need to walk up the tree and check for the balancedness of the other vertices on the path to the root, for them everything looks as it was before.

## Fixing the other imbalances (2)

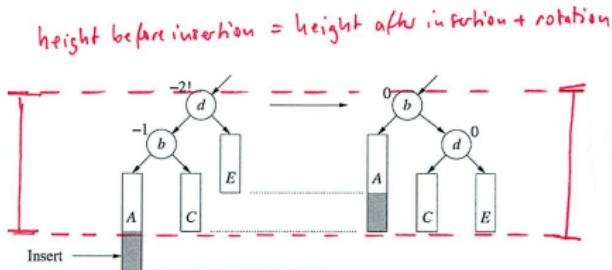


Figure 22: Single rotation.

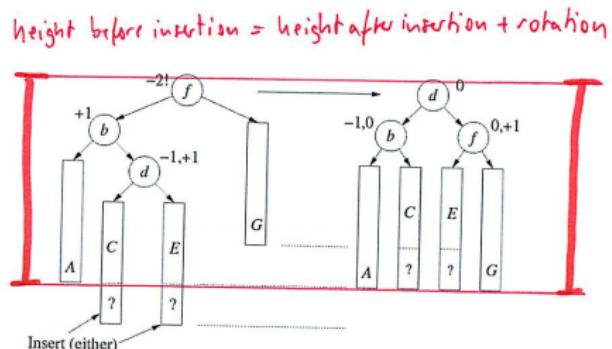
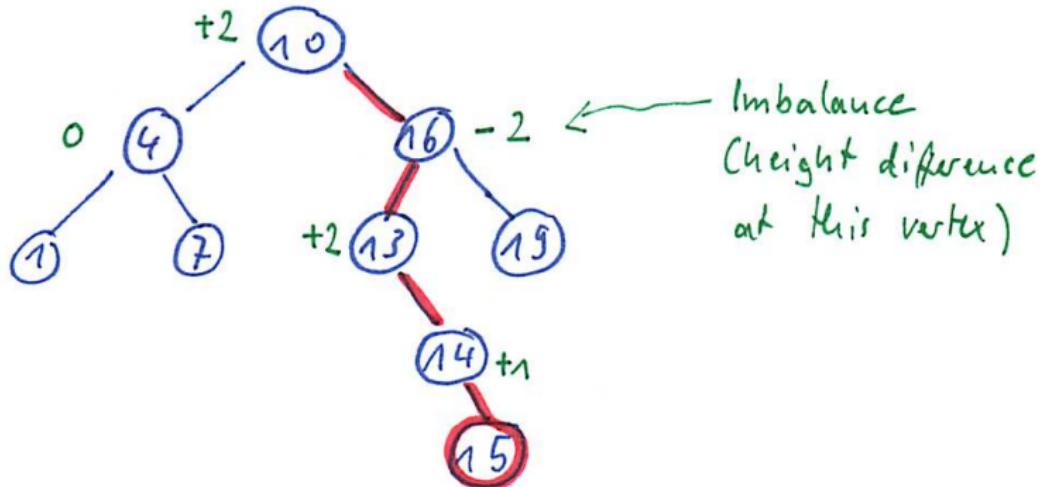


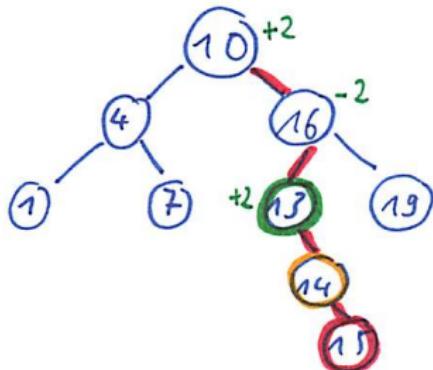
Figure 23: Left-right double rotation.

## Fixing the other imbalances (3)

Example:



## Fixing the other imbalances (4)

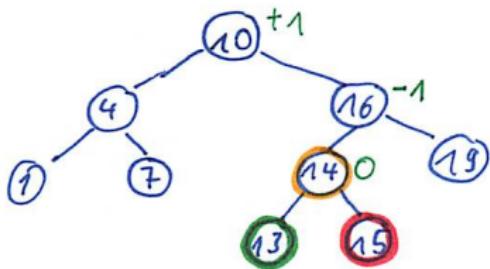


Walking up from 15 towards the root, 13 is the first imbalanced vertex we find.

## Fixing the other imbalances (5)

Note: at vertex 13 we are in situation 1 (with empty A, C, E).

## Fixing the other imbalances (6)



After performing the rotation,  
all imbalances are fixed  
(even at vertices above).

## Fixing the other imbalances (7)

Formal reason:

- ▶ After rebalancing, the corresponding subtree has the same height as it had been before.
- ▶ So the imbalance of any vertex above the current one is as it had been before insertion, hence it is correct.

## Running time of insertion

- ▶ Standard insertion to a search tree:  $O(\log n)$
- ▶ Walk up the tree to the check balancedness:  $O(\log n)$ .
- ▶ For the first imbalance you find, perform at most two rotations, each  $O(1)$ .
- ▶ Done.

So to insert an element to an AVL and maintain the balancing property takes  $O(\log n)$ .

# Deleting an element

Situation 1:

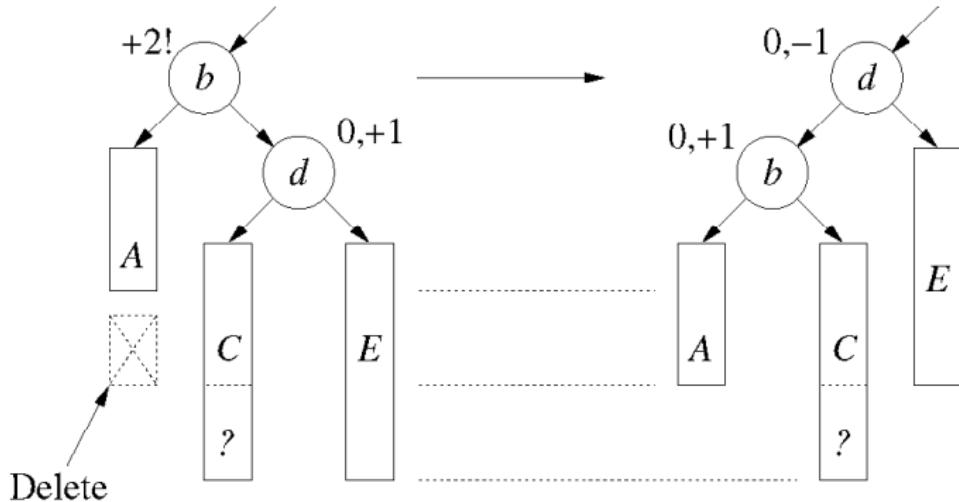


Figure 24: Single rotation for deletion.

# Deleting an element (2)

Situation 2:

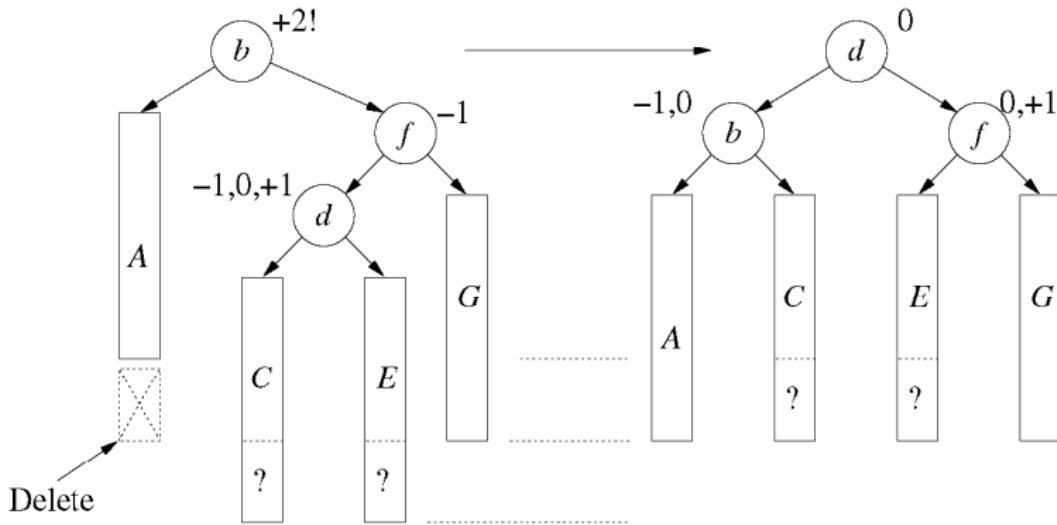


Figure 25: Double rotation for deletion.

## Deleting an element (3)

A whole deletion process can require rotations for all vertices along the insertion path. Example:

# Deleting an element (4)

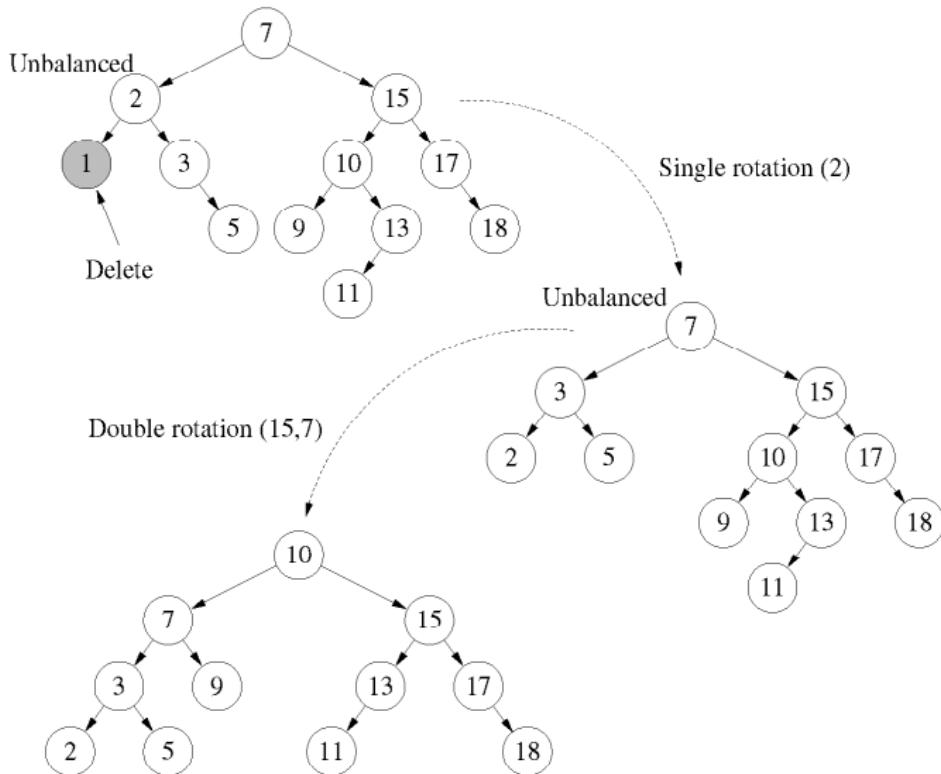


Figure 26: AVL Deletion example.

## Final comments

- ▶ We omit the formal proofs, they are not difficult but length (many different cases to distinguish).
- ▶ In order to implement AVL trees, each node stores as an additional attribute its height (or just the difference of the height of its subtrees, as an integer in  $\{-1, 0, 1\}$ ). Need to maintain it throughout all the rotations.

## History: AVL tree

The first tree data structure for searching was the “AVL” tree:

- ▶ **A**delson, **V**elskii, **L**andis: An algorithm for the organization of information. Soviet Mathematics Doklady, 1962.

Outlook: more search trees

## More search trees

- ▶ There exists a large variety of search trees, all with slightly different properties ... see Section 7.7. for some references.
- ▶ General trade-off:  
Harder balancing constraints  $\implies$  fast retrieval but slower insertion/removal
- ▶ To save costs, one might tolerate a certain unbalancedness, and just rebalance the tree if the unbalancedness becomes too large.

## More operations to support

Additionally to the basic operations, we might want to support some of the following operations:

- ▶ Find min / max
- ▶ Concatenate / merge two search trees
- ▶ Split two search trees
- ▶ ...

## Outlook: red black tree

- ▶ Is another balanced binary search tree architecture.
- ▶ Vertices are either “red” or “black”.
- ▶ A red vertex has only black children.
- ▶ At any vertex  $x$ : all paths from  $x$  to leaves contain the same number of black vertices.

One can prove:

- ▶ The height of such a tree is  $O(\log n)$ . Hence it is approximately balanced.
- ▶ Inserting and deleting vertices can be done in  $O(\log n)$  (but it is reasonably complicated because we have to make sure that the red-black-properties are fixed after changing the tree).

## Outlook: Splay tree

Invented by D. Sleator and R. Tarjan, 1985.

General idea:

- ▶ Whenever we access an element  $x$  in the tree, we perform a “splay operation”. This operation moves  $x$  to the root of the tree.
- ▶ In this way, elements that are often accessed sit close to the root (fast access when we need it the next time).
- ▶ We also call splay operations after inserting an element (to move it to the root) and after deleting an element (then we apply the splay operation to the parent of the deleted vertex).

If element  $e$  is accessed with probability  $p$ , then over time the tree will be reshaped to allow for access to  $e$  in time  $O(\log(1/p))$ .

## Excursion: Inverted index for document retrieval

### Literature:

- ▶ Büttcher, Clarke, Cormack: Information Retrieval, 2010.  
Chapter 4 (see paper repository)
- ▶ Zobel, Moffat: Inverted files for text search engines. ACM Computing Surveys, 2006. (see paper repository)

# The retrieval problem

Given a collection of documents (say, all html pages in the web) and a query for a search term (say, “Hamburg”), find all pages that contain the query term.

- ▶ Obviously, if the system is supposed to scale we cannot start searching through all documents at query time.
- ▶ We need an efficient data structure that did the “scanning” already.
- ▶ Building the data structure is allowed to be somewhat expensive, but once we have it, queries should be answered very fast.
- ▶ Ideally, we would like to update the data structure online as the contents of the document collection changes.

## Inverted index, idea

Like the index in a book:

- ▶ Maintain a dictionary of all words that occur in the whole collection of documents
- ▶ For each word, store a list of document ids of all documents that contain the term.

# Inverted index, idea (2)

Example:

## Document collection:

<u>Id</u>	<u>Document</u>
1	It is very cold today.
2	I came by bike.
3	There is a cold wind.

## Dictionary

<u>Terms</u>	<u>Occurrences</u>
It	1
is	1, 3
very	1
cold	1, 3
today	1
I	2
came	2
by	2
bike	2
there	3
:	

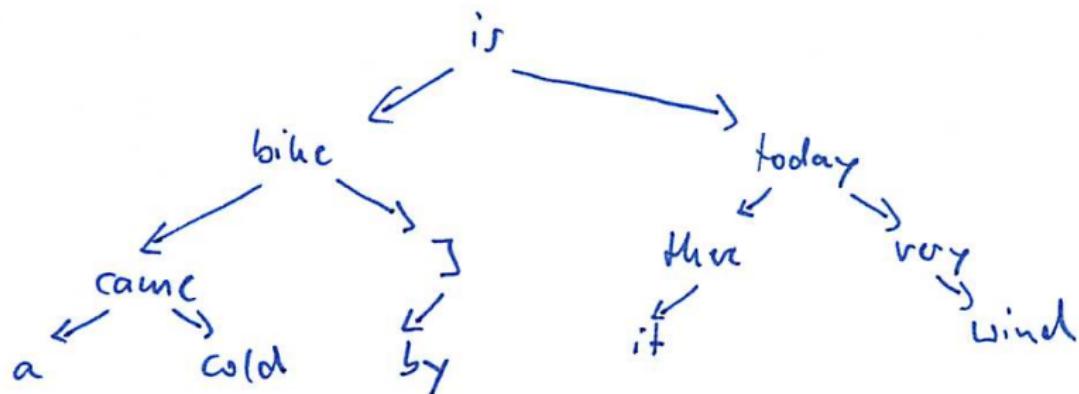
# Data structure for storing the index

ANY IDEAS?

## Data structure for storing the index (2)

### Sort-based dictionary:

- ▶ The terms are stored in a balanced search tree.
- ▶ At query time, we search the tree for the occurrence of the search term.
- ▶ Advantage: also supports prefix queries of the type “quer\*”.



## Data structure for storing the index (3)

### Hash-based dictionary:

- ▶ We organize the search terms in a hash table, where conflicts are resolved by chaining.
- ▶ At query time, we first evaluate which is the correct hash bin of the query, and then we look through all corresponding entries to find (or not find) the occurrences of the query term in the documents.
- ▶ As a rule of thumb, the hash table should grow linearly with the number of terms in the dictionary, to avoid too costly collisions.

## Data structure for storing the index (4)



## Data stored for each term occurrence

The actual list items encode (term, doc id), but usually store more information such as :

- ▶ term frequency: how often does the term occur in the document?
- ▶ where exactly does the term occur in the document (positions of the terms in the document)

# Answering queries

## **Single term query:**

Easy: just look in your data structure to see in which of the documents the term occurs.

## **Multiple term query: Term1 AND Term2:**

- ▶ Look up the ids for Term 1, and the ones for Term 2, as usual.
- ▶ Build the intersection of the two lists of ids  
Remark: this has to be done efficiently ...

## **Phrases like “inverted index”:**

- ▶ First proceed as in the multiple term query.
- ▶ Then also evaluate the positions at which the two query terms occur.  
Remark: if this has to happen efficiently, it is not straight forward.

## Answering queries (2)

### Ranking the search results:

- ▶ It is crucial to present the search results to the user such that the “most important” hits appear at the top of the list of results.
- ▶ The problem of rearranging the list of matches according to how important they are is called “ranking”.
- ▶ Google has been founded on the idea of a new ranking algorithm: pagerank.
- ▶ We won’t have time to discuss the details...

# Optimizing the index

To build an efficient system, it is crucial to **adapt the data structure to the type of data you are dealing with.**

- ▶ In natural language, the vast majority of terms comes from a list of, say, 10.000 words.
- ▶ So we need to make sure that access to these 10.000 words is very fast.
- ▶ If this is the case, the average performance of the system is going to be fast as well.

# Optimizing the index (2)

## Adapting hash tables:

- ▶ Want to make sure that frequent query terms are at the beginning of the list that contains the collisions.
- ▶ **Insert-at-back:** Whenever you insert an element, insert it to the end of the list. Rare words usually do not occur early in the index construction, so the rarer the word is, the more at the back of the list it tends to be.
- ▶ **Move-to-front:** Whenever a query finds a term, move the corresponding term to the beginning of the collision list. In this case, frequently asked queries remain at the beginning of the list, rare queries at the end of the list.

# Optimizing the index (3)

## Adapting search trees:

- ▶ Make sure that frequent search terms are at the top of the tree.
- ▶ Whenever a query term has been accessed, perform a sequence of operations that move it closer to the top of the tree.
- ▶ Not so obvious how to do this, look up “Splay trees” for an example.

## Challenge: scaling!

- ▶ Obviously, it is a huge challenge to build such an inverted index for huge systems such as the internet.
- ▶ All the components have to be saved and maintained in a highly distributed fashion.
- ▶ Details are out of the scope of this lecture. If you are interested, you could start to read the references I mentioned at the beginning of this section.

# Graph algorithms

## Recap: basic graph definitions

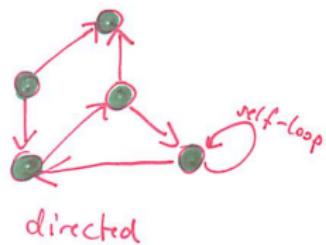
# Recap: graphs

A graph  $G = (V, E)$  consists of vertices  $v \in V$  and edges  $e \in E$ .

- edges can be **directed** or **undirected**
- edges can be **weighted**, that is each edge  $e = (u, v)$  has a weight  $w(u, v)$ .
- an edge  $e = (u, v)$  is called a self-loop if  $u = v$



undirected



directed

# Recap: graphs (2)

Notation:

- ▶ Often we treat unweighted graphs as a special case of weighted graphs where all edge weights  $w(u, v)$  are either 0 (no edge between  $u$  and  $v$ ) or 1 (undirected edge exists between  $u$  and  $v$ ).
- ▶ We say that  $u$  is **adjacent** to  $v$  if there exists an edge between  $u$  and  $v$ .
- ▶ In an undirected graph, we write  $u \sim v$  if  $u$  is adjacent to  $v$ .
- ▶ In a directed graph we write  $u \rightarrow v$  if there is a directed edge from  $u$  to  $v$ .

## Recap: graphs (3)

In an undirected graph, the **degree of a vertex**  $v$  is defined as

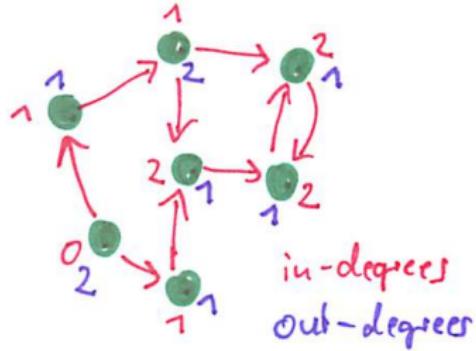
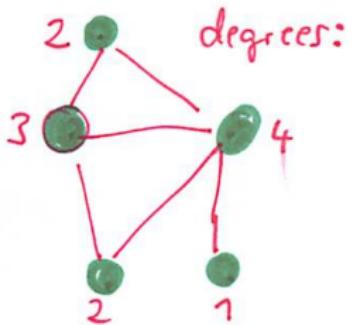
$$d_v := d(v) := \sum_{v \sim u} w_{vu}$$

In a directed graph, we define the **in-degree** and the **out-degree**:

$$d_{in}(v) = \sum_{\{u : u \rightarrow v\}} w(u, v)$$

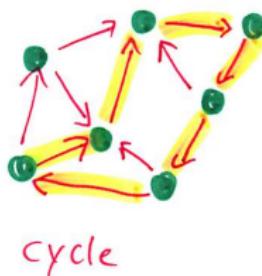
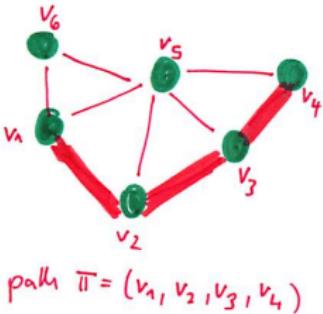
$$d_{out}(v) = \sum_{\{u : v \rightarrow u\}} w(v, u)$$

## Recap: graphs (4)



## Recap: graphs (5)

- ▶ A (directed) **path** in a graph is a sequence of vertices  $v_1, \dots, v_k$  such that there is a (directed) edge between  $v_i$  and  $v_{i+1}$  for all  $i = 1, \dots, k - 1$ .
- ▶ A path is called a **cycle** if  $v_1 = v_k$ , that is it ends in the vertex where it started from. Each edge is allowed to occur at most once (that is, walking an undirected edge back and forth is not considered a cycle).

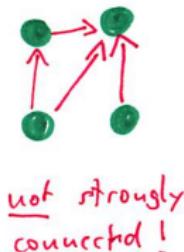
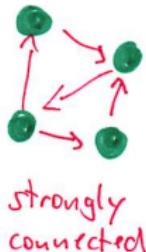
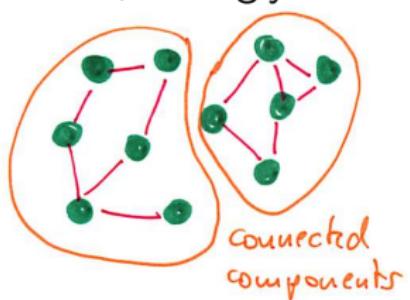


## Recap: graphs (6)

- ▶ A path is called **simple** if it contains no cycle (that is, each vertex appears at most once).

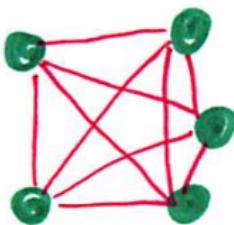
## Recap: graphs (7)

- ▶ An undirected graph is called **connected** if for all  $u, v \in V$ ,  $u \neq v$  there exists a path from  $u$  to  $v$ .
- ▶ A directed graph is called **strongly connected** if for all  $u, v \in V$ ,  $u \neq v$  there exists a directed path from  $u$  to  $v$  AND a directed path from  $v$  to  $u$ .
- ▶ A **connected component** of an undirected graph is a maximal, connected subset of  $V$ .
- ▶ A **strongly connected component** of a directed graph is a maximal, strongly connected subset  $A \subset V$ .



# Particular graphs

**Complete graph:** There exists an edge between all pairs of vertices  $u, v \in V$ .



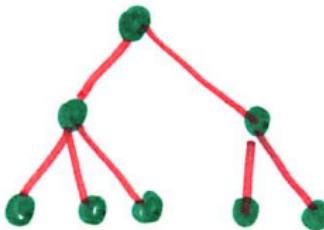
complete graph

Note: some people define the complete graph with self-loops, other people define it without self-loops.

## Particular graphs (2)

**Acyclic graph:** A graph is called acyclic if it does not contain any cycles.

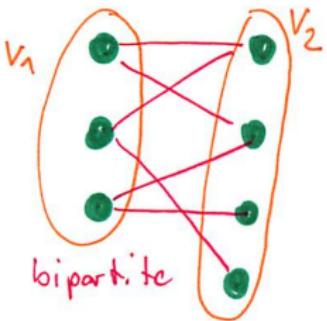
A **(undirected) tree** is an undirected, connected, acyclic graph.



Any acyclic undirected graph can be represented as a **forest**, a collection of trees.

## Particular graphs (3)

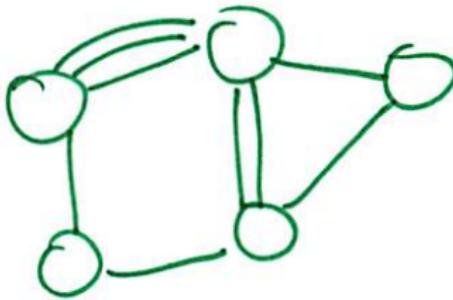
A graph is called **bipartite** if its vertex set  $V$  can be decomposed into two disjoint subsets  $V = V_1 \cup V_2$  such that all edges in  $E$  are only between  $V_1$  and  $V_2$ , but not within  $V_1$  or within  $V_2$ .



## Particular graphs (4)

Sometimes people consider graphs that can have several edges between vertices. Such graphs are called **multi-graphs**.

Usually, multi-graphs don't have edge weights, they can be directed or undirected.



In most cases, we can replace multi-graphs by weighted graphs.

# Particular graphs (5)

Two somewhat vague notions:

- ▶ A graph is called **dense** if it has “very many edges”. The definition of “very many” depends on the actual source one considers, often it is used if the number of edges is  $O(n^2)$ .
- ▶ A graph is called **sparse** if it has only “few edges”, for example each vertex has a very small degree compared to  $n$ . . .

# Particular graphs (6)

Finally, general conventions:

- ▶ In many papers and books, the authors use the convention that *n* denotes the number of vertices and *m* the number of edges.
- ▶ Sometimes authors also use  $|V|$  and  $|E|$ .

# Data structures for representing graphs

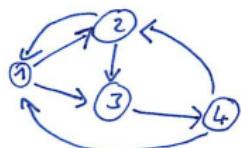
Literature: Mehlhorn Sec. 8

## Representing a graph

EVERYBODY: GIVEN A GRAPH, HOW WOULD YOU STORE IT???

# Adjacency matrix

- ▶ **Adjacency matrix  $A$ :** an  $n \times n$  matrix (where  $n$  is the number of vertices) that contains entries  $a_{ij} = 1$  if there is a directed edge from vertex  $i$  to vertex  $j$  and  $a_{ij} = 0$  otherwise.
- ▶ If the graph is weighted, then the adjacency matrix contains the weights of the edges, that is  $a_{ij} = w_{ij}$ . By default,  $w_{ij} = 0$  if there is no edge between  $i$  and  $j$ .
- ▶ To implement it, we simply use  $n$  arrays of length  $n$  each.



Adjacency matrix:

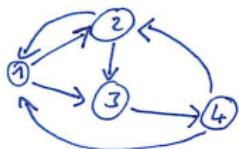
0	1	1	0
1	0	1	0
0	0	0	1
1	1	0	0

## Adjacency matrix (2)

- ▶ Needs  $n^2$  units of storage!
- ▶ This is only useful if the graph is dense (WHY???)
- ▶ But in some algorithms, this is useful after all, for example if we want to use tools from linear algebra.
- ▶ Advantage is also that we can test in time  $O(1)$  whether two particular vertices are adjacent.

# Adjacency arrays

- ▶ For each vertex  $v$ , we store a little array that contains the target vertices of all edges adjacent to  $v$
- ▶ We concatenate all these arrays to one long array of length  $|E|$ .
- ▶ To know which target vertices belong to which starting vertex, we additionally store the starting positions of the subarrays of all the vertices.



Adjacency array:	Edge array	Starting positions
[2   3   3   4   1   2]		
	[1   3   4   5]	

## Adjacency arrays (2)

- ▶ Needs  $|V| + |E| + O(1)$  units of storage
- ▶ Very space efficient
- ▶ Unsuitable if the graph is dynamic (WHY?)
- ▶ Can take up to  $O(n)$  time to test whether two particular vertices are adjacent (WHY?)

## Adjacency lists

For each vertex, we store a list of all outgoing edges or of all incoming edges, and sometimes also of all incoming and outgoing edges.

More sophisticated constructions exist.

## Adjacency lists (2)

- ▶ Needs about  $m$  units of storage.
- ▶ Easy to add / remove edges.
- ▶ Testing whether two particular vertices are adjacent can take up to  $O(n)$  time (WHY???)

# Walking through graphs

# The problem

We want to “walk” through a graph in a systematic way.

Example:

- ▶ We want to send a message to all sensors in a sensor network.
- ▶ We want to find out whether there exists a path from A to B in a road network.

EVERYBODY: TAKE A COUPLE OF MINUTES. TRY TO COME UP WITH A STRATEGY TO WALK THROUGH A GRAPH SO THAT YOU VISIT EACH VERTEX (AT LEAST / AT MOST) ONCE.

# Depth first search

Literature: Cormen 22.3; Mehlhorn Sec 9; (Dasgupta 3.2)

# Depth first search — idea

Given any (directed or undirected) graph  $G = (V, E)$ .

Goal: Explore and traverse the whole graph

## Algorithm: Depth first search (DFS)

**General idea:** we want to explore the graph with “long paths”. Starting at one arbitrary vertex, we jump to one of its neighbors, then one of his neighbors, and so on. We take care that we never visit a vertex twice. Once the current chain ends, we backtrack and walk along another chain.

# Depth first search — pseudo code

DFS( $G$ )

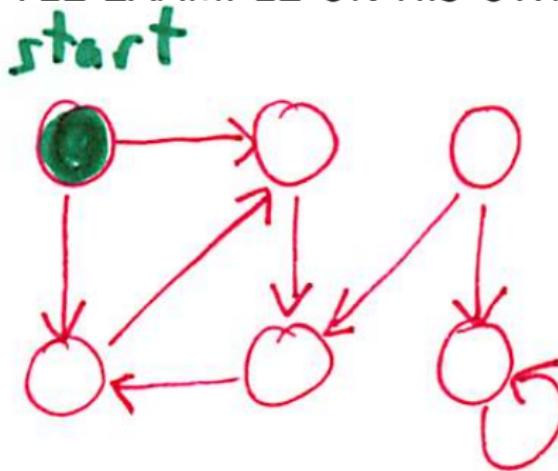
```
1 for all  $u \in V$ 
2    $u.\text{color} = \text{white}$  # not visited yet
3 for all  $u \in V$ 
4   if  $u.\text{color} == \text{white}$ 
5     DFS-Visit( $G, u$ )
```

DFS-Visit( $G, u$ )

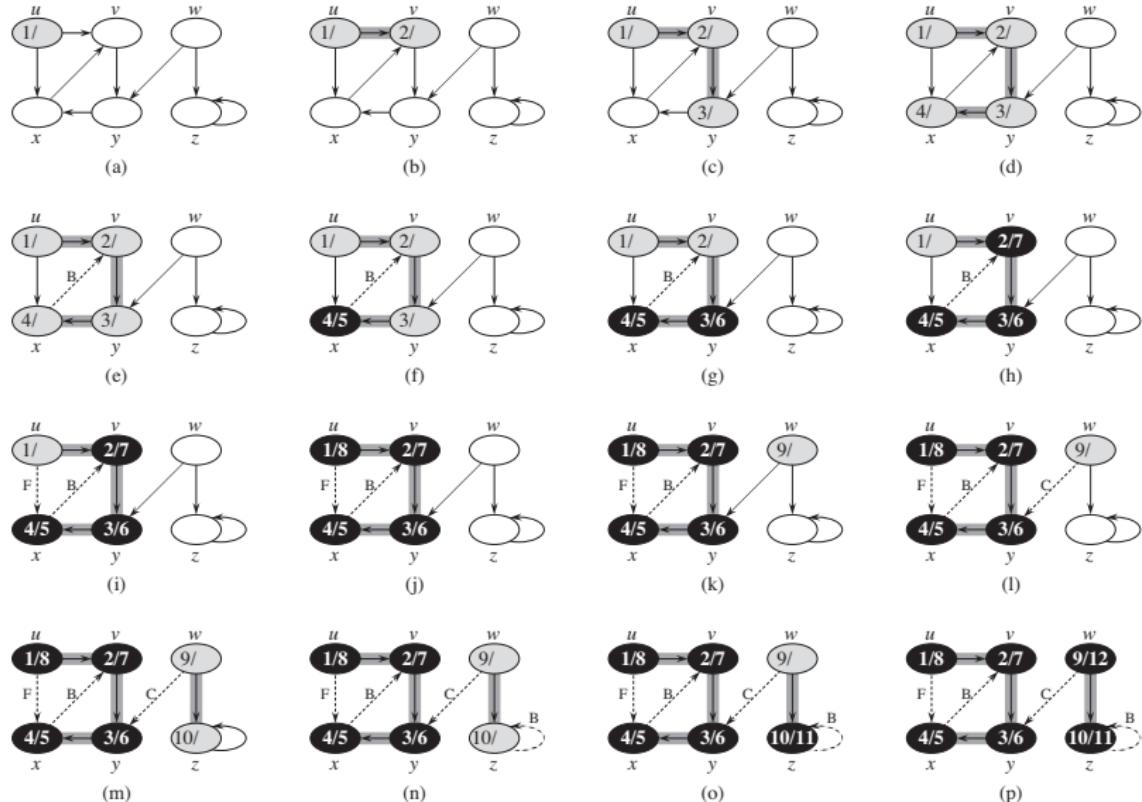
```
1  $u.\text{color} = \text{grey}$  # grey: in process
2 for all  $v \in \text{Adj}(u)$ 
3   if  $v.\text{color} == \text{white}$ 
4      $v.\text{pre} = u$  # Remember where we came from, just for analysis
5     DFS-Visit( $G, v$ )
6    $u.\text{color} = \text{black}$  # black: we are done
```

# DFS — an example

EVERYBODY SHOULD TRY TO RUN DFS ON THE FOLLOWING LITTLE EXAMPLE ON HIS OWN:



# DFS — an example (2)



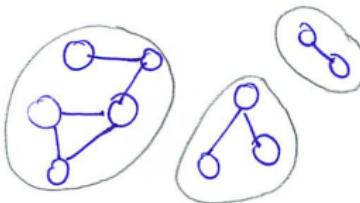
# DFS — properties

- ▶ The algorithm visits each vertex of the graph exactly once (why?)
- ▶ The algorithm travels along each edge of the graph at least once (why?).
- ▶ The running time of the algorithm is  $O(|V| + |E|)$  (why? )

## Application: Connected components

# Connected components

Recall: In an undirected graph, a **connected component** is a maximal subset  $A$  of vertices such that there exists a path between each two vertices  $u, v \in A$ .



Observe: if we start a DFS in a vertex  $u$ , then the tree discovered by  $\text{DFS}(G, u)$  contains exactly all vertices in the connected component of  $u$ .

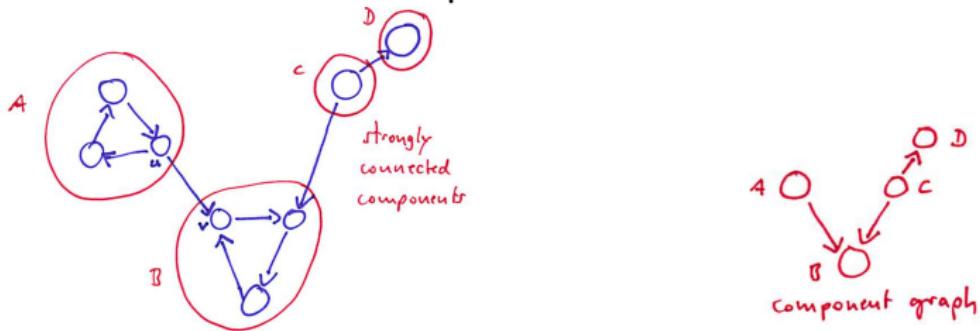
Consequence: each tree created by DFS exactly corresponds to a connected component of the graph.

## Application: Strongly connected components

Literature: Cormen 22.5; Dasgupta

# Strongly connected components (SCC)

Recap: in a directed graph, a **strongly connected component** is a maximal subset  $S$  of vertices such that for each two vertices  $u, v \in S$  there exists a directed path from  $u$  to  $v$  and vice versa.



We define the **component graph**  $G^{SCC}$  of a directed graph:

- ▶ Vertices of  $G^{SCC}$  correspond to the components of  $G$
- ▶ There exists an edge between vertices  $A$  and  $B$  in  $G^{SCC}$  if there exist vertices  $u$  and  $v$  in the connected components represented by  $A$  and  $B$  such that there is an edge from  $u$  to  $v$ .

# Strongly connected components (SCC) (2)

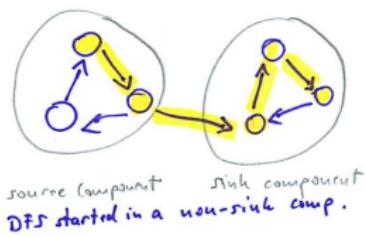
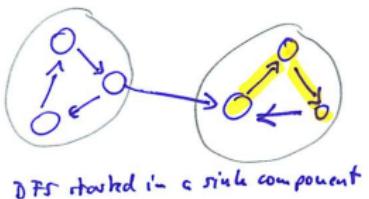
## Proposition 12

For any directed graph  $G$ , the graph  $G^{SCC}$  is a DAG.

**Proof:** If  $G^{SCC}$  would contain a cycle, then all corresponding vertices would be part of a larger SCC, which contradicts the definition of the SCCs.

# Observation: DFS when started in a sink comp.

- ▶ Consider a component  $B$  that corresponds to a sink in  $G^{SCC}$ .
- ▶ If we start DFS on  $G$  in a vertex  $u \in B$ , then the tree constructed in  $\text{DFS}(G,u)$  covers the whole component  $B$ .
- ▶ However, if we start with  $u$  in a non-sink component, then the tree constructed in  $\text{DFS}(G,u)$  covers more than this component.



- ▶ Idea: to discover the SCCs we start a DFS in a sink component and then work our way backwards along  $G^{SCC}$ .

# Finding sources

Consider a standard DFS on a directed graph.

Define the **discovery time**  $d(u)$  and **finishing time**  $f(u)$  of a vertex as the time when the DFS algorithm first visits  $u$  (i.e., it makes  $u$  grey) and the time when it is done with  $u$  (i.e., it makes  $u$  black).

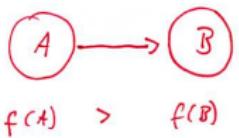
For a set  $A$  of vertices, define:

$$d(A) := \min_{u \in A} d(u) \quad \text{and} \quad f(A) := \max_{u \in A} f(u)$$

# Finding sources (2)

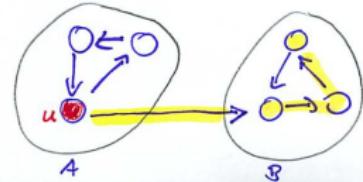
## Proposition 13

Let  $A$  and  $B$  be two SCCs of a graph  $G$  and assume that  $B$  is a descendent of  $A$  in  $G^{SCC}$ . Then  $f(B) < f(A)$  (no matter where we start the DFS).

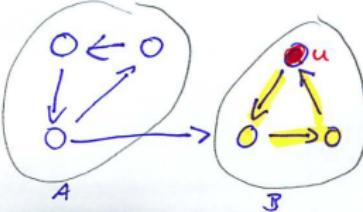


## Finding sources (3)

**Proof:** Case  $d(A) < d(B)$ . Denote by  $u$  the first vertex we visit in  $A$ . Then the DFS builds a tree starting from  $u$ , and this tree will cover all of  $B$  before it is done with  $u$ .



Case  $d(A) > d(B)$ . Denote by  $u$  the first vertex we visit in  $B$ . Then the DFS will first visit all of  $B$  before moving on.



□

# Finding sources (4)

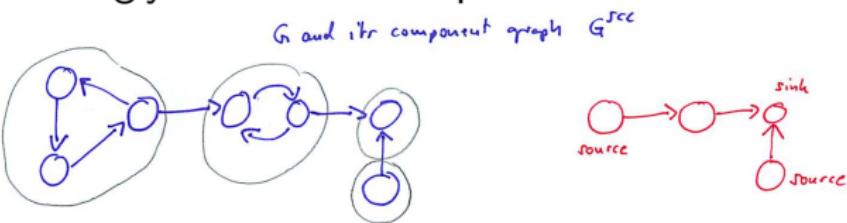
Consequence:

- ▶ Assume we run a normal DFS on  $G$  and for each vertex we store the finishing time  $f(u)$ .
- ▶ By Proposition ?? we now know that the vertex  $u$  with the largest finishing time has to be in a source component of  $G^{SCC}$ .

# Converting sources to sinks

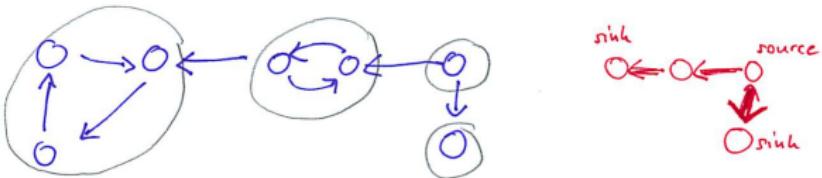
By now we know how to find a source. But we would like to have a sink!

Trick: We “reverse” the graph: we consider the graph  $G^t$  which has the same vertices as  $G$  but all edges with reversed directions. Note that  $G^t$  has the same strongly connected components as  $G$ .



# Converting sources to sinks (2)

Reversed graph  $G^t$  and its component graph



# SCC — final algorithm

General idea:

- ▶ Run a first DFS. The vertex  $u^*$  with the largest finishing time  $f(u)$  is in a source of  $G^{SCC}$ .
- ▶ Thus the vertex  $u^*$  is in a sink of  $(G^t)^{SCC}$ .
- ▶ Now start a second DFS on  $u^*$  in  $G^t$ . The tree discovered by  $DFS(G^t, u^*)$  is the first strongly connected component.
- ▶ Then continue with a DFS on the vertex  $V = v^*$  that has the highest  $f(u)$  among the remaining vertices.
- ▶ And so on.

# SCC — final algorithm (2)

- 1  $\text{SCC}(G)$
- 2 Call  $\text{DFS}(G)$  to compute the finishing times  $f(u)$
- 3 Compute the reverse graph  $G^t$
- 4 Call  $\text{DFS}(G^t)$ , where the vertices in the main loop are considered in order of decreasing  $f(u)$
- 5 Output the vertices of each tree of  $\text{DFS}(G^t)$  as a SCC

# Comments

Running time: as two DFSs, i.e. still  $O(|V| + |E|)$ .

## Application: Topological sort

Literature: Cormen 22.5; Dasgupta Sec. 3.3.2; Mehlhorn Sec. 9.2.1

# A scheduling problem to start with

A simple scheduling problem:

- ▶ You have to schedule a number of jobs.
- ▶ There are some dependencies like
  - ▶  $job_3$  can only happen after  $job_{13}$  is finished.
  - ▶  $job_7$  can only happen after  $job_1$  is finished.
  - ▶ ...
- ▶ Your task:
  - ▶ Figure out whether a correct scheduling is possible at all.
  - ▶ Find a schedule (in what order do we perform the tasks).

ANY IDEAS HOW TO DO THIS?

## A scheduling problem to start with (2)

Solution:

- ▶ Form a directed graph from your constraints: edge from  $job_i$  to  $job_j$  if  $job_i$  has to be finished before  $job_j$  can start.
- ▶ Then try to find a “topological sort”:

Definiton:

A **topological sort of a directed graph** is a linear ordering of its vertices such that whenever there exists a directed edge from vertex  $u$  to vertex  $v$ ,  $u$  comes before  $v$  in the ordering.

First thoughts: when is it possible at all?

DOES A TOPOLOGICAL SORT EXIST FOR ANY DIRECTED  
GRAPH?

First thoughts: when is it possible at all? (2)

No, only if the graph is a DAG. (WHY???)

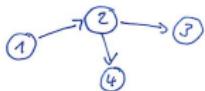
Exercise: prove it formally!

First thoughts: when is it possible at all? (3)

IN CASE THERE EXISTS A TOPOLOGICAL SORT, IS IT  
ALWAYS UNIQUE?

## First thoughts: when is it possible at all? (4)

No, in most cases not.



This graph is consistent with the orderings 1, 2, 3, 4 and 1, 2, 4, 3.

## First thoughts: when is it possible at all? (5)

### Proposition 14 (Uniqueness of topological sort)

The topological sort of a DAG is unique if and only if the DAG contains a Hamiltonian path (that is, a path that visits each vertex exactly once).

Proof: exercise.

# Topological sort: how to solve it

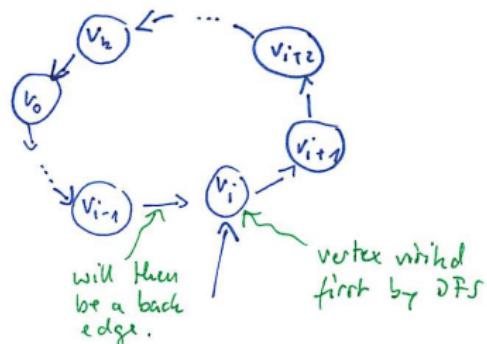
## Proposition 15 (DFS on a directed graphs)

A directed graph has a cycle if and only if its DFS reveals a back edge (that is, an edge that goes from the current vertex to a previously visited vertex).

Proof of " $\Leftarrow$ ":

Assume DFS reveals a back edge  $(u, v)$ . This means that the DFS has already visited  $u$ , some intermediate vertices  $v_1, \dots, v_k$ , and then  $v$ , and now sees a link back to  $u$ . Thus, the graph has the cycle  $u, v_1, \dots, v_k, v, u$ .

## Topological sort: how to solve it (2)

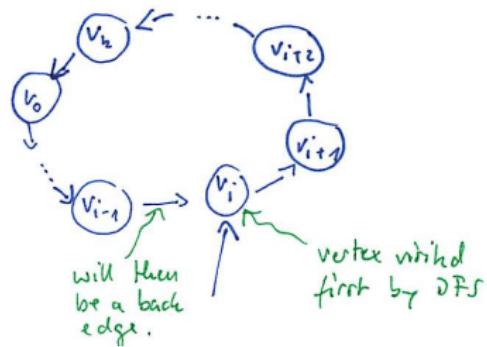


Proof of " $\implies$ :

- ▶ Assume there is a cycle  $v_0, v_1, \dots, v_k, v_0$ .
- ▶ Let  $v_i$  be the vertex with the smallest discovery time.
- ▶ All other  $v_j$  can be reached from  $v_i$  and are thus descendants in the DFS tree.

## Topological sort: how to solve it (3)

- In particular, this is true for  $v_{i-1}$ , which is going to reveal the edge  $(v_{i-1}, v_i)$ , which is a back edge.



# Topological sort: how to solve it (4)

## Proposition 16

On a DAG, all edges go from larger to smaller finishing times.

- ▶ DFS on a DAG does not have any back edges
- ▶ Then we always finish the descendants first before we finish the ancestors (this would not be true for back edges).

## Topological sort: how to solve it (5)

Algorithm for topological sort:

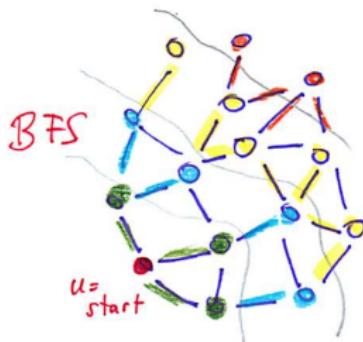
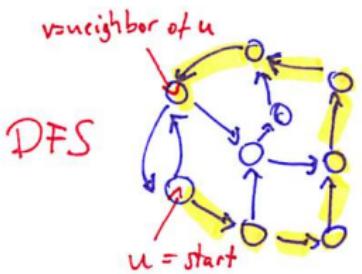
- ▶ Run DFS with an arbitrary starting vertex and record the finishing times.
- ▶ Then sort the vertices by decreasing finishing times.

# Breadth first search

Literature: Cormen 22.2 (Dasgupta 4.2)

# BFS — Intuition

In DFS, we can happen to take a very long path between two vertices that are very close indeed.



In BFS, we try to travel “uniformly” through the graph, making sure we first explore the local neighborhood of the starting point before going further.

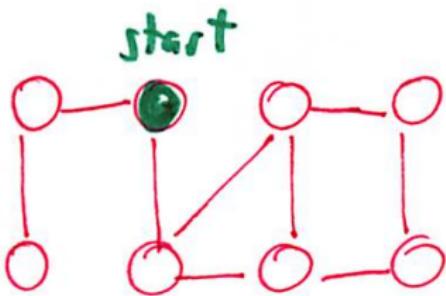
# BFS — pseudocode

*BFS( $G, s$ )*

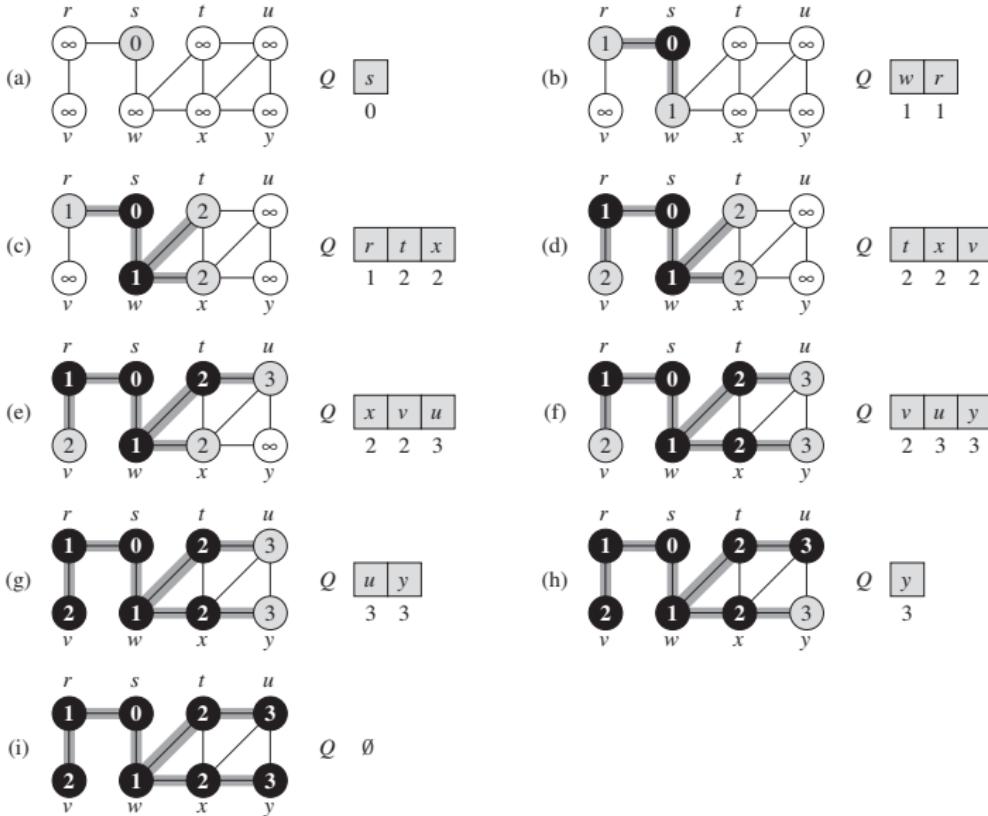
```
1 for all vertices  $u \in V \setminus \{s\}$  # initialize
2    $u.\text{color} = \text{white}$ 
3    $s.\text{color} = \text{grey}$ 
4    $Q = [s]$  # queue containing  $s$ 
5   while  $Q \neq \emptyset$ 
6      $u := \text{DEQUEUE}(Q)$ 
7     for all  $v \in \text{Adj}(u)$ 
8       if  $v.\text{color} == \text{white}$ 
9          $v.\text{color} = \text{grey}$ 
10         $\text{INQUEUE}(Q, v)$ 
11      $u.\text{color} = \text{black}$ 
```

# BFS — an example

EVERYBODY SHOULD TRY TO RUN BFS ON THE FOLLOWING LITTLE EXAMPLE ON HIS OWN:



# BFS — an example (2)



# BFS — running time

- ▶ Running time:  $O(|E| + |V|)$  (Why?)

# BFS vs. DFS

- ▶ DFS travels “deep”, BFS travels more “like a wave”
- ▶ DFS is “adventurous”: it only traces back if it has nowhere else to go.  
BFS is “timid”: it always traces back until it cannot trace back any more and has to move forward.
- ▶ On a high level: both are nearly the same, the main difference is that BFS uses a queue whereas DFS uses a stack.

## Application: shortest paths in unweighted graphs

## Replace your “Navi”

- ▶ You want to find the shortest path in a network of roads.
- ▶ Assume all roads have the same lengths.

ANY IDEAS HOW THIS COULD BE SOLVED USING DFS OR BFS?

# BFS finds shortest paths

In an unweighted graph, the length  $\ell(\pi)$  of a path in the graph is defined as the number of edges in the path.

The **shortest path distance** between two vertices is defined as

$$d(u, v) = \min\{\ell(\pi) \mid \pi \text{ path between } u \text{ and } v\}.$$

The shortest path distance satisfies the **triangle inequality**:

$$d(u, v) \leq d(u, w) + d(w, v) \text{ for all } u, v, w \in V.$$

In particular: if there is an edge from  $u$  to  $v$ , then

$$d(s, v) \leq d(s, u) + 1.$$



## BFS finds shortest paths (2)

Want to show: If  $G$  is an unweighted graph, then  $BFS(G, s)$  can compute the shortest path distances of all vertices to  $s$ .

Any idea how??

Let's insert a couple of lines in BFS:

# BFS finds shortest paths (3)

*BFS*( $G, s$ )

```
1 for all vertices  $u \in V \setminus \{s\}$ 
2    $u.\text{color} = \text{white}$ 
3    $u.\text{dist} = \infty$ 
4    $s.\text{dist} = 0$ 
5    $s.\text{color} = \text{grey}$ 
6    $Q = [s]$ 
7   while  $Q \neq \emptyset$ 
8      $u := \text{DEQUEUE}(Q)$ 
9     for all  $v \in \text{Adj}(u)$ 
10       if  $v.\text{color} == \text{white}$ 
11          $v.\text{color} = \text{grey}$ 
12          $v.\text{dist} = u.\text{dist} + 1$ 
13          $\text{INQUEUE}(Q, v)$ 
14        $u.\text{color} = \text{black}$ 
```

# BFS finds shortest paths (4)

We are now going to prove the following statement:

## Theorem 17

Let  $G$  be an unweighted graph. Upon termination of  $BFS(G, s)$  we have  $v.dist = d(s, v)$  for all vertices  $v$  that are reachable from  $s$ .

Even though it looks obvious, we are going to give a formal proof now. It requires several steps.

# BFS finds shortest paths (5)

## Proposition 18

Upon termination of  $BFS(G, s)$ , we have  $v.dist \geq d(s, v)$  for all  $v \in V$ .

**Proof:** by induction over the number of INQUEUE operations.

Base case: Queue only contains  $s$ . Obvious because of initialization

Inductive step:

- ▶ Consider a white vertex  $v$  that is discovered from predec.  $u$ .
- ▶ By the inductive hypothesis:  $u.dist \geq d(s, u)$

# BFS finds shortest paths (6)

- Then we get:

$$\begin{aligned}v.dist &= u.dist + 1 \text{ (by the pseudo-code)} \\&\geq d(s, u) + 1 \text{ (by the inductive hypothesis)} \\&\geq d(s, v) \text{ (by the triangle inequality)}\end{aligned}$$



# BFS finds shortest paths (7)

## Proposition 19

Suppose that during the execution of BFS the queue contains  $(v_1, \dots, v_r)$  where  $v_1$  is the head and  $v_r$  the tail. Then:

$$v_1.dist \leq v_2.dist \leq \dots \leq v_r.dist \leq v_1.dist + 1$$

**Proof:** Induction on queue operations.

Base case: The queue only contains  $s$ . Obviously true.

Inductive step: Have to consider two cases.

## BFS finds shortest paths (8)

- ▶ Assume we dequeue  $v_1$ . Obviously we still have  $v_2.dist \leq \dots \leq v_r.dist$ . And we also have  $v_r.dist \leq v_1.dist + 1 \leq v_2.dist + 1$  (using the inductive hypothesis  $v_1.dist \leq v_2.dist$ ).
- ▶ Assume we enqueue  $v$ , it is now  $v_{r+1}$ .  
Assume predecessor of  $v$  was  $u$ .  $u$  has just been dequeued.  
Because  $u$  was previously head in the queue, by ind.hyp. we have  $u.dist \leq v_1.dist$ . By the shortest path property above we know  $v.dist \leq u.dist + 1$ . Together:  
 $v.dist \leq u.dist + 1 \leq v_1.dist + 1$

Similarly can also conclude:  $v_r.dist \leq u.dist + 1 = v.dist$



## BFS finds shortest paths (9)

Finally, we are going to prove Theorem ??:

**Proof:** By Proposition ?? we know that for all vertices,  $v.dist \geq d(s, v)$ , that is BFS can only attain values that are too large.

Assume the theorem is wrong. Let  $v$  be vertex with the smallest  $d(v, s)$  that gets a wrong distance. Let  $u$  be the predecessor vertex of  $v$  in a shortest path from  $s$  to  $v$ . In particular:  
 $d(s, u) + 1 = d(s, v)$ .

Consider the time when we dequeue  $u$ .

1. CASE  $v.color = \text{white}$ : then code sets  $v.dist = u.dist + 1$ , which would be the correct distance. ↳
2. CASE  $v.color = \text{black}$ : then  $v$  was ejected from the queue before  $u$ , contradicting Proposition ??.

## BFS finds shortest paths (10)

3. CASE  $v.\text{color} = \text{grey}$ : this happened for dequeuing another vertex  $w$ . Have by Proposition ??:
- $$v.\text{dist} = w.\text{dist} + 1 \leq u.\text{dist} + 1. \checkmark$$



# Shortest path problems

# Shortest paths: definitions and properties

We are now going to treat various shortest path algorithms more deeply.

- ▶ In an unweighted graph, the **length  $\ell(\pi)$  of a path** in the graph is defined as the number of edges in the path. In a weighted graph, the length of a path is usually defined as the sum of the edge weights along the path.
- ▶ The **shortest path distance** between two vertices is defined as

$$d(u, v) = \min\{\ell(\pi) \mid \pi \text{ path between } u \text{ and } v\}.$$

- ▶ Observe: shortest paths are often not unique.
- ▶ Attention: in the presence of negative weights, funny things can happen. In particular, if the graph contains loops in which the sum of weights is negative, then the shortest path has length  $-\infty$  (hence it is not well-defined).

# Shortest paths: definitions and properties (2)

## Proposition 20

The shortest path distance satisfies the triangle inequality:

$$d(u, v) \leq d(u, w) + d(w, v) \text{ for all } u, v, w \in V.$$

**Proof:** Easy.

Consequence: In graphs with non-negative weights, the shortest path distance is a **metric**

- ▶  $d(a, b) \geq 0$  for all  $a, b$
- ▶  $d(a, b) = 0 \iff a = b$
- ▶  $d(a, b) \leq d(a, c) + d(c, b)$

# Shortest paths: definitions and properties (3)

## Proposition 21

Subpaths of shortest paths are shortest paths.

**Proof:** easy.

# Different shortest path problems

We are going to consider several types of problems:

- ▶ Single Source Shortest Paths: We want to know the shortest path distances of one particular vertex  $s$  to all other vertices.
- ▶ All Pairs Shortest Paths: We want to know the shortest path distance between all pairs of points.
- ▶ Point to Point Shortest Paths: We want to the shortest path distance between a particular source vertex  $s$  and a particular sink vertex  $t$

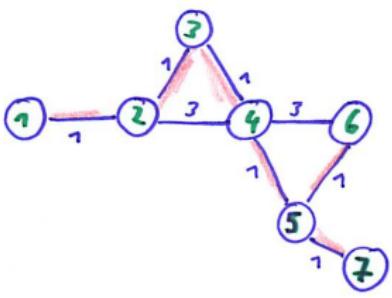
# Storing paths efficiently

Keep track of the predecessors in the shortest paths with the help of the predecessor matrix  $\Pi = (\pi_{ij})_{i,j=1,\dots,n}$ :

- ▶ If  $i = j$  or there is no path from  $i$  to  $j$ , set  $\pi_{ij} = \text{NIL}$
- ▶ Else set  $\pi_{ij}$  as the predecessor of  $j$  on a shortest path from  $i$  to  $j$ .

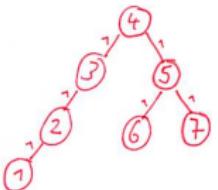
The subgraph induced by row  $i$  of the matrix  $\Pi$  induces a tree structure called the predecessor subgraph of  $G$  for  $i$ .

# Storing paths efficiently (2)



von	1	2	3	4	5	6	7
1	N	1	2	3	4	5	Γ
2	2	N	2	3	4	5	5
3	2	3	N	3	4	5	Γ
4	2	3	4	N	4	5	Γ
5	2	3	4	5	N	5	Γ
6	2	3	4	5	5	N	5
7	2	3	4	5	5	5	N

predecessor subgraph of ④ =  
tree corresponding to root ④:

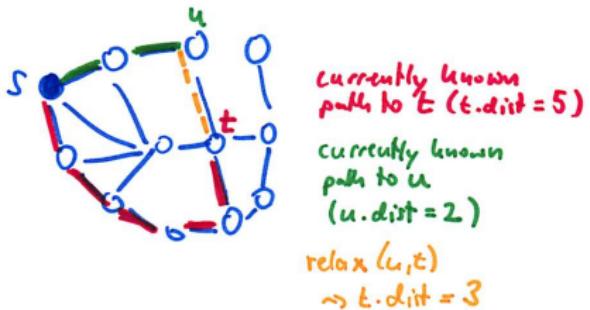


# Single Source Shortest Paths

# A general technique: Relaxation

... is going to be used by several of the following algorithms.

- ▶ For each vertex, keep an attribute  $v.dist$  that is the current estimate of the shortest path distance to the source vertex  $s$ .
- ▶ Initially, it is set to  $\infty$  for all vertices except the start vertex  $s$ .
- ▶ Relaxation step: figure out whether we can improve the shortest path from  $s$  to  $v$  by using an edge  $(u, v)$  and thus extending the shortest path of  $s$  to  $u$ .



## A general technique: Relaxation (2)

This kind of procedure has several properties that are easy to prove:

- ▶ Upper bound:  $v.dist$  is always an upper bound to the actual shortest path distance.
- ▶ No path property: If there is no path from  $s$  to  $v$ , then  $v.dist = \infty$ .
- ▶ Convergence: If  $u.dist$  contains the correct shortest path distance, this never changes any more during the course of the algorithm.

# A general technique: Relaxation (3)

Two functions we will need over and over again. Let  $G$  be a graph with weights  $w$  and  $s$  the starting point for our single source shortest path problem.

## InitializeSingleSource( $G, s$ )

```
1 for all  $v \in V$ 
2    $v.dist = \infty$  # Current distance estimate
3    $v.\pi = NIL$  # Predecessor on the best current path to  $v$ 
    $s.dist = 0;$ 
```

## Relax( $u, v$ )

```
1 if  $v.dist > u.dist + w(u, v)$ 
2    $v.dist = u.dist + w(u, v)$ 
3    $v.\pi = u$ 
```

## Bellman-Ford algorithm

Literature: Cormen 24.1; (Kleinberg 6.8)

Original sources are somewhat hard to track, commonly cited are:

- ▶ R. Bellman: On a routing problem. Quarterly of Applied Mathematics 16: 87–90, 1958
- ▶ L. R. Ford: Network flow theory, Paper P-923. The Rand Corporation, Santa Monica 1956
- ▶ E. F. Moore: The shortest path through a maze. In: Proceedings of the International Symposium on the Theory of Switching. 2/1959. Harvard University Press, S. 285–292

See also the notes by D. Walden (2003) on the history of the Bellman-Ford algorithm (is in the paper repository)

# Bellman-Ford algorithm: pseudo-code

Solves the single source shortest path problem for general weighted graphs (edges are allowed to be negative). Returns false in case of negative-weight-cycles.

**BellmanFord( $G, s$ )**

```
1 InitializeSingleSource( $G, s$ )
2 for  $i = 1, \dots, |V| - 1$ 
3   for all edges  $(u, v) \in E$ 
4     Relax( $u, v$ )
5   for all edges  $(u, v) \in E$ 
6     if  $v.dist > u.dist + w(u, v)$ 
7       return false
8 return true
```

# Analysis of the algorithm

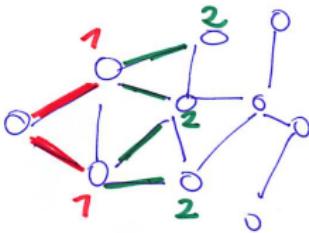
## Proposition 22

Assume that  $G$  contains no negative weight cycles. Then the Bellman-Ford algorithm returns the correct shortest path values.

**Proof :** (intuition, case unweighted graphs)

In the first iteration of the for-loop, the edges to the neighbors of  $s$  are relaxed, the the neighbors of  $s$  have the correct values. After the second iteration, their neighbors have the correct values. And so on.

# Analysis of the algorithm (2)



correct after iteration 1  
correct after iteration 2

# Analysis of the algorithm (3)

Formal proof:

Consider any vertex  $v$  in the graph, and a shortest path  $s, v_1, \dots, v_k, v$  from  $s$  to  $v$ .

- ▶ After the first iteration,  $v_1$  has the correct distance.
- ▶ After the second iteration,  $v_2$  has the correct distance.
- ▶ In general, after iteration  $i$ , all vertices whose shortest paths contain at most  $i$  edges have the correct distances.
  - ▶ To be very formal, use a proof by induction for this statement.
  - ▶ Note that this also holds for weighted graphs with arbitrary edge weights, as long as there is no negative cycle.
- ▶ Shortest paths in the graph can contain at most  $|V| - 1$  edges. So after the last iteration, each vertex in the graph has the correct distance.



# Analysis of the algorithm (4)

## Proposition 23

If  $G$  contains a negative-weight cycle, then the algorithm returns FALSE.

**Proof:** Denote the negative-weight cycle by  $v_0, \dots, v_k$  with  $v_0 = v_k$ . By definition

$$\sum_{i=0}^{k-1} w(v_i, v_{i+1}) < 0.$$

Now assume the algorithm returns TRUE. In this case, the IF-clause at the end of the code was false for all  $v_i$ , that is

$$v_i.dist \leq v_{i-1}.dist + w(v_{i-1}, v_i)$$

Summing up these inequalities over  $i = 0, \dots, k - 1$  and exploiting that  $v_0 = v_k$  leads to  $0 \leq \sum_{i=0}^{k-1} w(v_i, v_{i+1})$ .  $\lhd$



# Running time

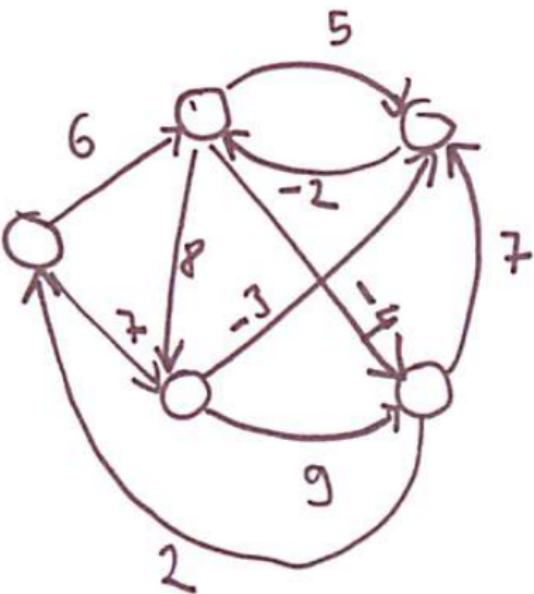
## Proposition 24

The running time of the algorithm is  $O(|V| \cdot |E|)$ .

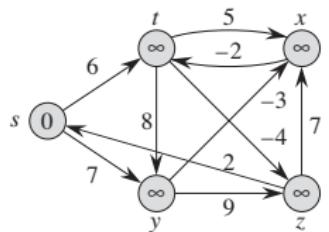
**Proof:** Easy

# Example

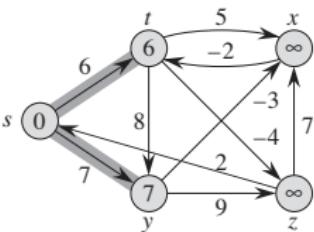
EVERYBODY: SOLVE THE FOLLOWING LITTLE EXAMPLE:



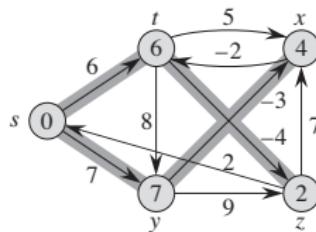
## Example (2)



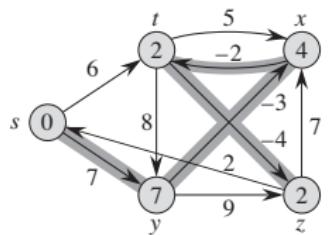
(a)



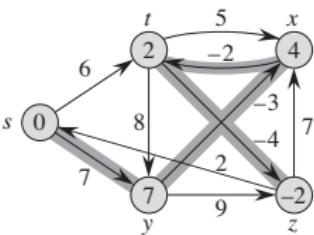
(b)



(c)



(d)



(e)

# Assumptions

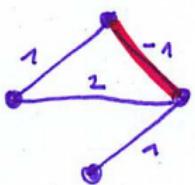
Note: The Bellman-Ford algorithm works on all general kinds of graphs: weighted or unweighted, directed or undirected, weights arbitrary.

However, note the following for undirected graphs:

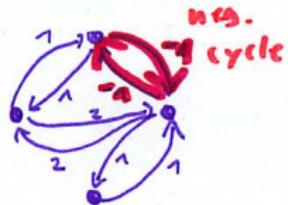
- ▶ If you run the Bellman-Ford algorithm on an undirected graph, you always have to relax the edges in both directions. The easiest is to convert the undirected graph in the equivalent directed graph.
- ▶ In particular, if there is an edge with a negative weight in an undirected graph, this induces a negative weight cycle in the directed version!!! So whenever an undirected graph has negative weights, shortest path is not defined.

# Assumptions (2)

Undirected graph



Corresponding directed graph



# Underlying paradigm: dynamic programming

The Bellman-Ford algorithm is an example of **Dynamic Programming**.

- ▶ decompose problem into smaller subproblems
- ▶ build up the overall solution by larger and larger subproblems (which are allowed to overlap, as opposed to divide and conquer)
- ▶ Store the solutions to all the subproblems solved so far in a table

In this particular case the crucial observation is as follows:

$$\ell(\text{shortest path of length } i \text{ from } s \text{ to } t)$$

$$= \min_u \ell(\text{shortest path of length } i - 1 \text{ from } s \text{ to } u) + w(u, t)$$

## Dijkstra's algorithm

Literature: Cormen 24.3; Kleinberg 4.4; Dasgupta 4.4.

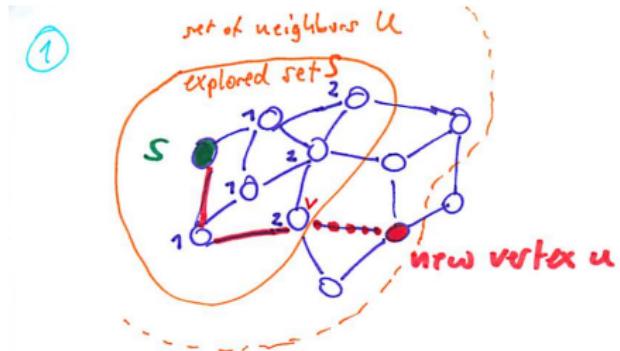
Original publication:

E. Dijkstra: A note on two problems in connexion with graphs.  
Numerische Mathematik 1: 269–271, 1959

# Dijkstra's algorithm: the naive way

This algorithm works on any weighted, directed (or undirected) graph **in which all edge weights  $w(u, v)$  are non-negative**.

- ▶ Algorithm maintains a set  $S$  of vertices for which it already knows the shortest path distances from  $s$ .
- ▶ Then it looks at neighbors  $u$  of  $S$  and assigns a guess for the shortest path by using a path through  $S$  and adding one edge.



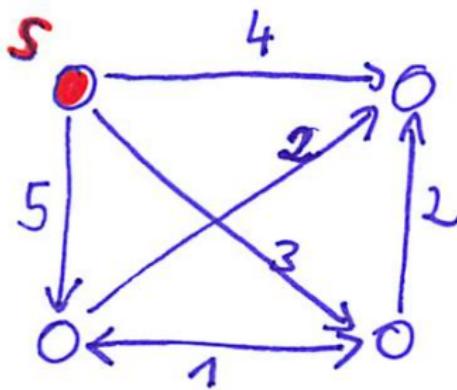
# Dijkstra's algorithm: the naive way (2)

DijkstraNaive( $G, w, s$ )

```
1  $S = \{s\}$  #  $S$  set of explored vertices
2  $d(s) = 0$ 
3 while  $S \neq V$ 
4    $U := \{u \notin S \mid u \text{ neigh. of a vertex } \in S\}$  # candidates
5   for all  $u \in U$ 
6     for all  $\text{pre}(u) \in S$  that are predecessors of  $u$ 
7        $d'(u, \text{pre}(u)) := d(\text{pre}(u)) + w(\text{pre}(u), u)$ 
         # candidate distances
8      $u^* := \operatorname{argmin}\{d'(u, \text{pre}(u)) \mid u \in U\}$  # choose best candidate
9      $d(u^*) = d'(u^*)$ 
10     $S = S \cup \{u^*\}$ 
```

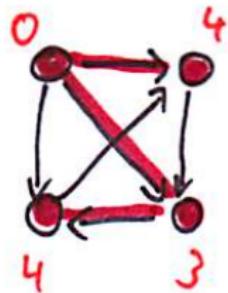
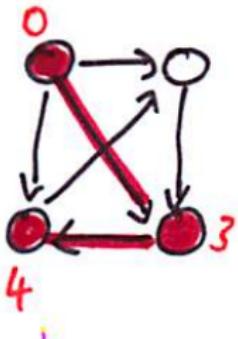
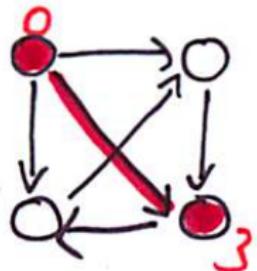
# Examples

EVERYBODY, PLEASE TRY THIS EXAMPLE:



## Examples (2)

Solution:



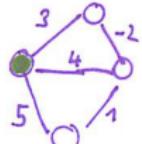
## Examples (3)

Important: Dijkstra's algorithm can go wrong if a graph has negative-weight edges!

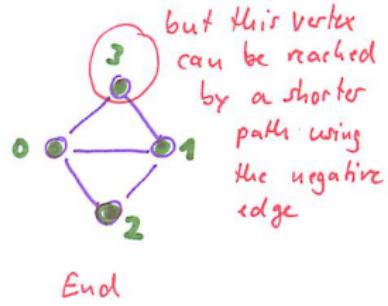
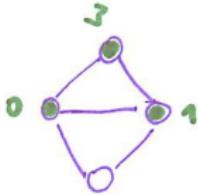
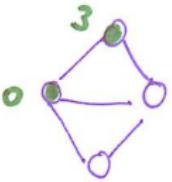
CAN YOU FIND AN EXAMPLE?

# Examples (4)

Example where Dijkstra gives the wrong result:



Start



End

# Analysis

## Theorem 25 ((Correctness of the algorithm))

Consider the set  $S$  at any point in the algorithm. For each  $t \in S$ , the value  $d(t)$  coincides with the shortest path distance from  $s$  to  $t$ .

**Proof.** Induction on the size  $|S|$  of  $S$ .

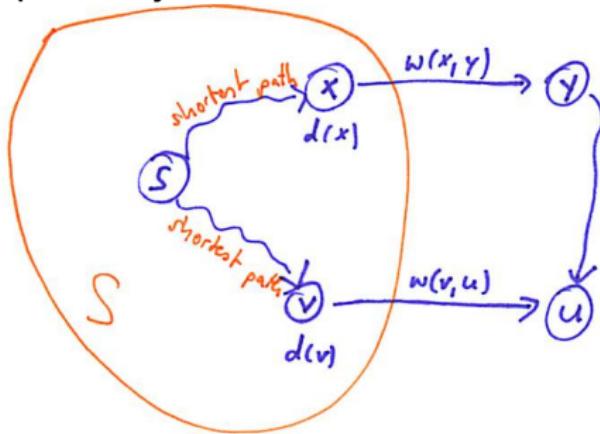
Base case:  $|S| = 1$ . Then  $S = \{s\}$ , and we assigned  $d(s) = 0$ . ☺

Inductive step from  $k$  to  $k + 1$ :

- ▶ Let  $u$  be the next vertex we add to  $S$ , denote its predecessor in  $S$  by  $v$ .
- ▶ Assume  $d(u) = d(v) + w(v, u)$  is not the shortest path distance.
- ▶ Instead, consider a correct shortest path to  $u$ .

# Analysis (2)

- It has to leave  $S$  somewhere, say at  $x$ , and say the next vertex on the shortest path is  $y$ .



- The path via  $x$  and  $y$  can only be shorter than the one via  $v$  if  $w(x, y) < w(v, u)$ .
- But as we added  $u$  and not  $y$  to  $S$  in this step,  $w(x, y) \geq w(v, u)$  according to the algorithm. ↵

# Analysis (3)



# Running time of the naive implementation

**Running time:** the naive implementation DijkstraNaive has running time  $O(|V| \cdot |E|)$ :

- ▶ while loop runs for all  $|V|$  vertices
- ▶ inside the while loop, we have to look at all neighbors of  $S$ , that is we have to look at all edges from  $S$  to  $V \setminus S$ , which is bounded by  $|E|$  edges.

This is pretty bad!

Luckily we can save a lot by using an appropriate data structure.

# Recap: min-priority queue

- ▶ Maintains a set  $S$  of elements
- ▶ Each element  $s$  has a key  $k(s)$  assigned to it
- ▶ The key of each element can be *decreased* at any point.
- ▶ When extracting an element, we extract the one with the smallest key.

Implementations: using arrays or various heaps, see later.

Implementation	extract element	insert element / decrease key
Array	$O( V )$	$O(1)$
Binary heap	$O(\log  V )$	$O(\log  V )$
d-ary hep	$O\left(\frac{d \log  V }{\log d}\right)$	$O\left(\frac{\log  V }{\log d}\right)$
Fibonacci heap	$O(\log  V )$	$O(1)$ (amortized)

Literature: Cormen 6.5; Kleinberg 2.5

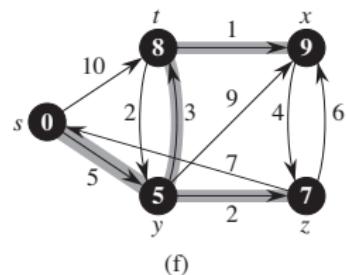
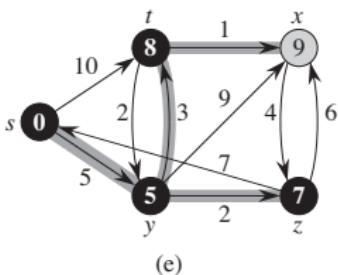
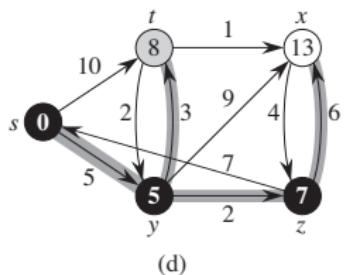
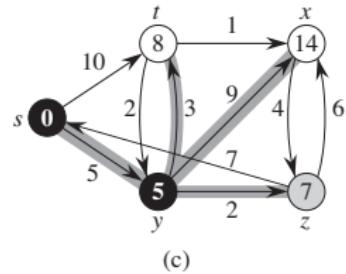
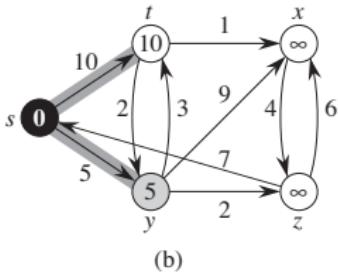
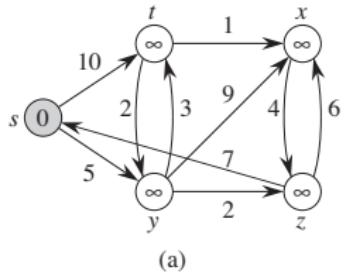
# Dijkstra's algorithm with min-priority queues

Dijkstra( $G, w, s$ )

- 1 InitializeSingleSource( $G, s$ ) *# initializes the  $v.dist$  values*
- 2  $Q = (V, V.dist)$  *# insert all vertices with the initial keys  $v.dist$*
- 3 **while**  $Q \neq \emptyset$
- 4    $u = \text{Extract}(Q)$
- 5   **for all**  $v$  adjacent to  $u$
- 6     Relax( $u, v$ ) and update the keys in  $Q$  accordingly

Note: Intuitively,  $Q$  is the complement of the set  $S$  we had in the naive Dijkstra algorithm:  $Q = V \setminus S$ .

# Dijkstra's algorithm with min-priority queues (2)



# Dijkstra's algorithm with min-priority queues (3)

Exercise: Formally prove that the two algorithm based on priority queues does exactly the same as the naive Dijkstra algorithm.

# Dijkstra's algorithm with min-priority queues (4)

## Running time:

- ▶  $|V|$  Insert operations of the queue
- ▶  $|V|$  Extract operations of the queue
- ▶  $|E|$  DecreaseKey operations of the queue (in the Relax subfunction)

Depending on how we implement the min-priority queue we get the following overall running times for Dijkstra's algorithm:

- ▶ Implementation by a naive array  $\sim O(V^2)$
- ▶ Implementation by a binary heap  $\sim O((V + E) \log V)$
- ▶ Implementation by a  $d$ -ary heap  $\sim O(|V| \cdot d + |E|) \frac{\log V}{\log d}$ );
- ▶ Implementation by a Fibonacci heap  $\sim O(V \log V + E)$

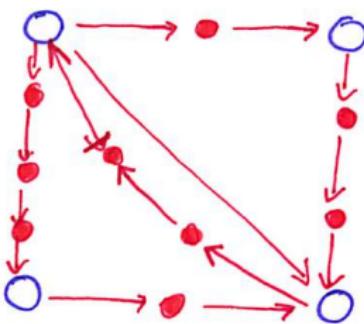
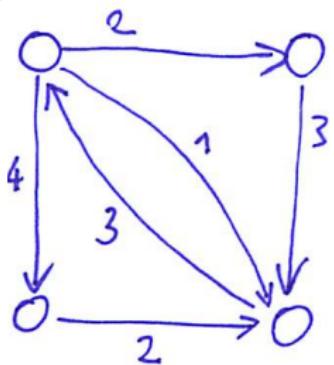
Which one to choose?

## Dijkstra's algorithm with min-priority queues (5)

- ▶ If the graph is **dense**, that is  $|E| = O(|V|^2)$ : prefer the array implementation
- ▶ If the graph is **sparse**, that is  $|E|$  is (much) smaller than  $|V|^2$ : prefer a heap implementation
- ▶ For a d-ary heap: choose  $d$  depending on graph:  $d = |E|/|V|$ . Then:
  - ▶ If graph is very sparse, i.e.  $|E| = \Omega(|V|)$  we get  $O(|V| \log |V|)$
  - ▶ If graph is dense, i.e.  $|E| = \Omega(|V|^2)$ , we get  $O(|V|^2)$

# Dijkstra's algorithm and BFS

One can interpret Dijkstra's algorithm as a generalization of BFS.  
For simplicity assume that edge weights are integers. Replace all edges of length  $k$  by a string of  $k - 1$  vertices connected by edges of length 1:



Then BFS on the new graph does the same as Dijkstra on the old graph.

## Remarks

- ▶ In some sense, Dijkstra's algorithm is a **greedy** algorithm. At each point in time, it does the “locally best” best action. In this particular case, the solution also turns out to be “globally optimal”.
- ▶ Dijkstra's algorithm requires global knowledge of the network (maintain set  $S$ ; make global decision about which node to add next). Unsuitable for many applications where global network is unknown or too big (e.g., routing problems in the www).

# Generic labeling method

# Generic labeling method

A convenient generalization of Dijkstra and Bellman-Ford:

- ▶ For each vertex, maintain a status variable  $S(v) \in \{\text{unreached}, \text{labelchanged}, \text{settled}\}$ .
- ▶ Repeatedly relax edges.
- ▶ Do this until nothing changes any more.

Pseudo-Code is as follows:

# Generic labeling method (2)

GenericLabelingMethod( $G, s$ )

```
1 for all  $v \in V$ 
2    $v.dist = \infty$  # current distance estimate
3    $v.parent = NIL$  # parent in current shortest path
4    $v.status = \text{unreached}$ 
5    $s.dist = 0$ 
6    $s.status = \text{labelchanged}$ 
7   while there exist vertices with status "labelchanged"
8     pick one such vertex  $v$ 
9     for all neighbors  $u$  of  $v$ 
10       Relax( $v, u$ )
11       if this relaxation changed the value  $u.dist$ 
12          $u.status = \text{labelchanged}$ 
13        $v.status = \text{settled}$ 
```

## Generic labeling method (3)

- ▶ Note that it can happen that a vertex gets scanned several times, the status can change back from “settled” to “labelchanged”.
- ▶ How often this is going to happen (and thus, how efficient this generic method is) depends on how exactly we choose the next vertex in line 8 of the code.
- ▶ If we manage to always select a vertex  $v$  such that  $v.dist$  is already the correct distance, then each vertex gets scanned exactly once.
- ▶ Dijkstra's algorithm is a special case of the labeling method: one can show that if we choose the vertex that has the smallest  $v.dist$  value among all vertices with status “labelchanged”, then the algorithm is equivalent to Dijkstra and each vertex gets scanned at most once.

## Generic labeling method (4)

- ▶ The Bellman-Ford algorithm can also be written in this framework (see later: decentralized Bellman-Ford)
- ▶ Another example we will see later:  $A^*$  search

# All pairs shortest paths on general graphs

# All pairs shortest paths

Goal: want to find the shortest paths between *all* pairs of vertices in a graph.

Naive approach: run Bellman-Ford or Dijkstra with all possible start vertices

Problem with this approach:

- ▶ Running time!
  - ▶ Bellman-Ford  $\sim O(V^2 \cdot E)$
  - ▶ Dijkstra, say with binary heaps  $\sim O(V(V + E) \log V)$
- ▶ Seems like a waste of efforts that we do not “re-use” the results we already have

# Floyd-Warshall algorithm

Literature: Cormen 25

Original papers:

- R. Floyd. Algorithm 97: Shortest Path. Communications of the ACM 5 (6), 1962
- S. Warshall. A Theorem on Boolean Matrices. J. ACM 9, 11-12, 1962.

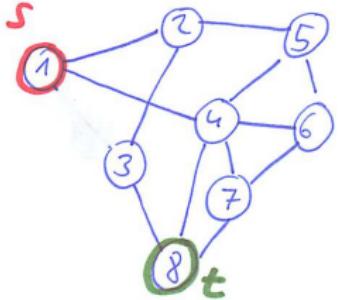
# Floyd-Warshall algorithm — idea

Dynamic programming approach:

- ▶ Assume all vertices are numbered from 1 to  $n$ .
- ▶ Fix two vertices  $s$  and  $t$ .
- ▶ Consider all paths from  $s$  to  $t$  that only use vertices  $1, \dots, k$  as intermediate vertices. Let  $\pi_k(s, t)$  be a shortest path *from this set*, and denote its length by  $d_k(s, t)$ .
- ▶ Want to find a recursive relation between the  $\pi_k$  and  $\pi_{k-1}$  to be able to construct the solution bottom-up.

# Floyd-Warshall algorithm — idea (2)

Example for the definition of  $\pi_k$  and  $d_k$ :



$$\begin{array}{lll}
 \pi_0(s, t) = \text{NIL} & d_0(s, t) = \infty \\
 \pi_1(s, t) = \text{NIL} & d_1(s, t) = \infty \\
 \pi_2(s, t) = \text{NIL} & d_2(s, t) = \infty \\
 \pi_3(s, t) = [1\ 2\ 3\ \delta] & d_3(s, t) = 3 \\
 \pi_4(s, t) = [1\ 4\ \varnothing] & d_4(s, t) = 2
 \end{array}$$

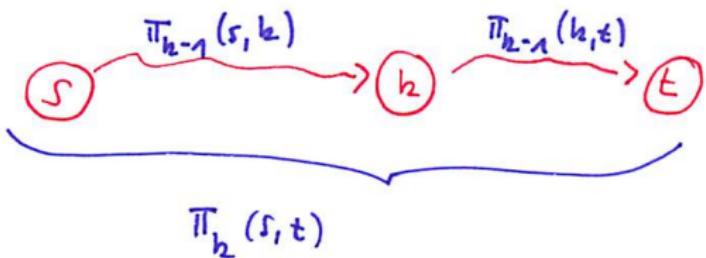
# Floyd-Warshall algorithm — idea (3)

Observe the following recursive relationships:

- ▶ If vertex  $k$  is not element of  $\pi_k(s, t)$ , then  
 $\pi_k(s, t) = \pi_{k-1}(s, t).$
- ▶ If vertex  $k$  is an element of  $\pi_k$ , it occurs at most once. Thus we can decompose

$$\underbrace{[sv_1 \dots v_i k v_{i+1} \dots v_j t]}_{\pi_k(s, t)} = \underbrace{[sv_1 \dots v_i k]}_{\pi_{k-1}(s, k)} \circ \underbrace{[kv_{i+1} \dots v_j t]}_{\pi_{k-1}(k, t)}$$

# Floyd-Warshall algorithm — idea (4)



## Floyd-Warshall algorithm — idea (5)

To exploit this insight bottom-up, we proceed as follows:

Case  $k = 0$  (“no intermediate vertices”): Define

$$d_0(s, t) = w(s, t)$$

Case  $k > 0$ : recursively define

$$d_k(s, t) = \min\{d_{k-1}(s, t) , d_{k-1}(s, k) + d_{k-1}(k, t)\}$$

In the algorithm below, we define the  $n \times n$  matrix  $D_k$  as the matrix with entries  $(d_k(s, t))_{s,t=1,\dots,n}$ .

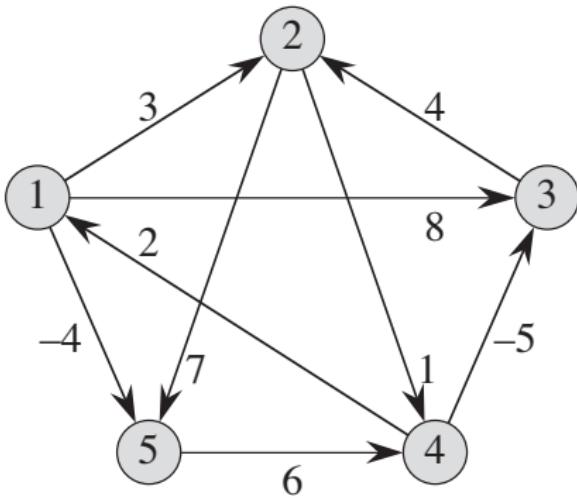
# Floyd-Warshall — pseudo code

FloydWarshall( $W$ )

```
1  $n :=$  number of vertices
2  $D^{(0)} = W$  # Matrix of size  $n \times n$ 
3 for  $k = 1, \dots, n$ 
4   Let  $D^{(k)}$  be a new  $n \times n$  matrix
5   for  $s = 1, \dots, n$ 
6     for  $t = 1, \dots, n$ 
7        $d_k(s, t) = \min\{d_{k-1}(s, t), d_{k-1}(s, k) + d_{k-1}(k, t)\}$ 
8       #  $d_k(s, t)$  is the entry at  $(s, t)$  in Matrix  $D^{(k)}$ 
8 return  $D^{(n)}$ 
```

# Example

Consider this graph:



On the next slide we see the corresponding  $D^{(k)}$ -matrices:

## Example (2)

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ 8 & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

# Floyd-Warshall — running time

Running time:  $O(|V|^3)$ .

- ▶ Hm, still does not look too good 😞
- ▶ Seems like it saves surprisingly little compared to other algorithms, even though it “re-uses” knowledge.

As a comparison: Using Dijkstra, say with binary heaps, leads to  $O(V(V + E) \log V)$ .

- ▶ If the graph is sparse, then Dijkstra is obviously preferable.
- ▶ If the graph is dense, Floyd-Warshall is preferable.

In general, Floyd-Warshall is simpler to implement and works reasonably on moderately sized graphs.

# And what if the graph has a negative-weight cycle?

We can detect this with Floyd-Warshall as follows:

- ▶ We have a negative cycle if there exists a path from  $i$  to  $i$  with negative length.
- ▶ Simply look at the values of the diagonal of the distance matrix that is output by Floyd-Warshall. If it contains negative entries, then the graph contains a negative cycle.
- ▶ Then all the results that were computed by the algorithm are meaningless.

## Johnson's algorithm

Literature: Cormen 25.3

Original paper:

D. Johnson. Efficient algorithms for shortest paths in sparse networks. Journal of the ACM 24 (1): 1-13, 1977

# Johnson's algorithm

Assume our graph has negative edges.

**Naive idea:** add weight to all edges such that all edge weights become positive: For some large constant  $a$  define

$$w'_{ij} = w_{ij} + a.$$

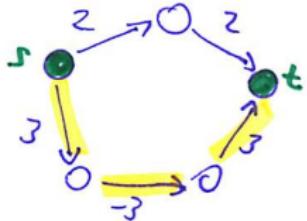
Then all weights are non-negative, and we can run Dijkstra without problems.

WHAT IS YOUR GUESS, IS THIS APPROACH VALID OR NOT?

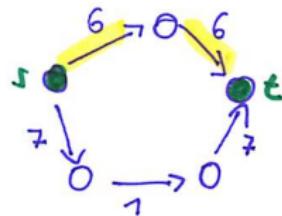
# Johnson's algorithm (2)

⚠️Attention⚠️: this naive approach can change the shortest paths.

Example:



shortest path in orig. graph  
(negative weights!)



shortest path after adding 4 to  
all edges

## Johnson's algorithm (3)

**Instead we want:**

Transform the edge weights such that

- (i) shortest paths don't change
- (ii) new edge weights are non-negative

If we manage to do this, we can proceed as follows:

- ▶ Given graph with arbitrary weights  $w_{ij}$
- ▶ Transform its weights to new weights  $\tilde{w}_{ij}$  that satisfy (i) and (ii)
- ▶ Then find the shortest paths according to the new edge weights

# Johnson's algorithm (4)

**First insight, deals with (i):**

Define new edge weights  $\tilde{w}_{ij}$  by

$$\tilde{w}_{ij} = w_{ij} + h(i) - h(j)$$

for some function  $h : V \rightarrow \mathbb{R}$ .

Theorem 26

## Johnson's algorithm (5)

Given a weighted graph  $G = (V, E)$  with arbitrary weight matrix  $W$ . Let  $h : V \rightarrow \mathbb{R}$  be any function. Define the new edge weight matrix  $\tilde{W} = (\tilde{w}_{ij})$  as above, and let  $\tilde{G}$  be the graph with the new edge weights.

1. Then  $\pi$  is a shortest path from  $s$  to  $t$  in  $G$  if and only if  $\pi$  is a shortest path from  $s$  to  $t$  in  $\tilde{G}$ .
2.  $G$  has a negative-weight cycle if and only if  $\tilde{G}$  has a negative weight cycle.

# Johnson's algorithm (6)

## Proof:

- ▶ Fix  $s$  and  $t$ . For any path  $\pi$  between  $s$  and  $t$  we have that  $\tilde{\ell}(\pi) = \ell(\pi) + h(s) - h(t)$
- ▶ Thus shortest paths between a fixed pair  $s$  and  $t$  of vertices don't change.
- ▶ In particular, for any cycle  $\pi$  we have  $\tilde{\ell}(\pi) = \ell(\pi)$  (just consider the case  $s = t$  above).



## Johnson's algorithm (7)

**But how to choose the function  $h$  to satisfy (ii)?**

If the graph has negative weight cycles:

- ▶ No matter what we choose as function  $h$ , we will always still have negative weight edges (because we still have negative weight cycles)!
- ▶ So we cannot get rid of negative cycles.  
Note that this is a good feature!

If we don't have negative weight cycles:

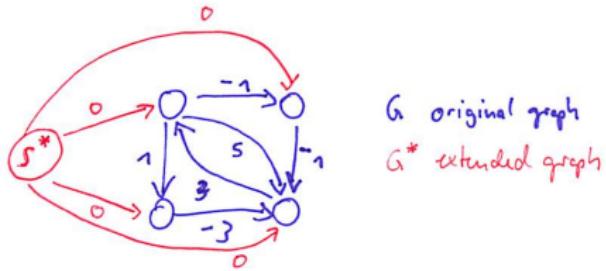
- ▶  $h(u)$  interferes with *all* neighbors of  $u$ .
- ▶ So we might start to turn one edge positive by assigning appropriate values to its end points  $u$  and  $v$ , but then these values interfere with all other neighbors of  $u$  and  $v$  as well...
- ▶ ... not obvious at all ...

# Johnson's algorithm (8)

Here is the solution how to deal with (ii):

First, we add a new vertex  $s^*$  to the graph as follows:

- ▶ Add a directed edge from  $s^*$  to all vertices  $v$  with edge weight  $w(s^*, v) := 0$ .
- ▶ This does not generate new cycles in the graph.
- ▶ All shortest paths in the old graph remain shortest paths in the new graph (because the in-degree of  $s^*$  is 0).
- ▶ Let's call the new graph  $G^*$ .



## Johnson's algorithm (9)

Now run the Bellman-Ford algorithm on  $G^*$  with source  $s^*$ :

- ▶ If the algorithm returns FALSE, then the graph  $G^*$  (and thus,  $G$ ) contains a negative weight cycle. Stop.
- ▶ Otherwise, let  $\delta(s^*, v)$  be the shortest path values for source  $s^*$  returned by Bellman-Ford. Note that  $\delta(s^*, v) \leq 0$  for all  $v \in V$  (why?).
- ▶ Now define  $h(v) := \delta(s^*, v)$ .

# Johnson's algorithm (10)

## Proposition 27

If the graph  $G$  does not contain any negative-weight cycle, then the edge weights  $\tilde{w}_{ij}$  generated with the function  $h$  from above are all non-negative.

**Proof:** By the triangle inequality, we have

$$h(v) := \delta(s^*, v) \leq \delta(s^*, u) + w(u, v) = h(u) + w(u, v),$$

that is

$$w(u, v) + h(u) - h(v) \geq 0.$$

## Johnson's algorithm (11)

Thus  $\tilde{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$ .

□

All in all, this leads to the following algorithm:

# Johnson's algorithm (12)

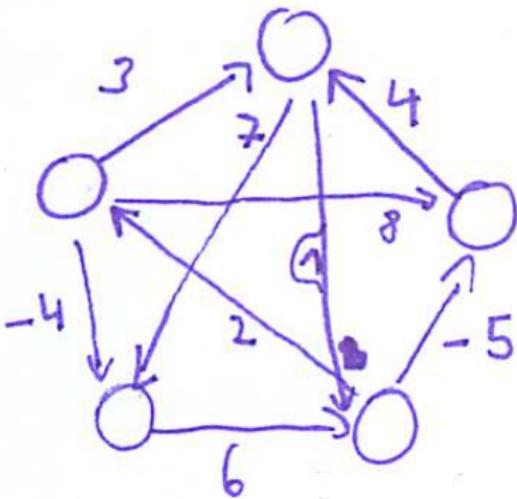
JohnsonAllPairsShortestPaths( $G$ )

- 1 Build graph  $G^*$  with the extra vertex  $s^*$  as above
- 2 **if**  $BellmanFord(G^*, s^*) == FALSE$
- 3     Return FALSE # no solution in case of neg. weight cycle
- 4 **else**
- 5     Let  $\delta(s^*, v)$  be the results by BellmanFord
- 6     **for all**  $v \in V$
- 7          $h(v) = \delta(s^*, v)$
- 8     **for all**  $(u, v) \in E$
- 9          $\tilde{w}(u, v) = w(u, v) + h(u) - h(v)$
- 10     **for all**  $u \in V$
- 11         run  $Dijkstra(\tilde{G}, \tilde{W}, u)$  to obtain  $\tilde{d}(u, v)$  for all  $v \in V$
- 12         **for all**  $v \in V$
- 13              $d(u, v) = \tilde{d}(u, v) - h(u) + h(v)$
- 14     Return  $D = (d(u, v))$

# Johnson's algorithm (13)

EVERYBODY: COMPUTE THE NEW WEIGHT OF ONE OF THE EDGES IN THE GRAPH:

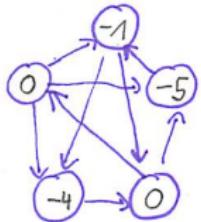
Original graph  $G$ :



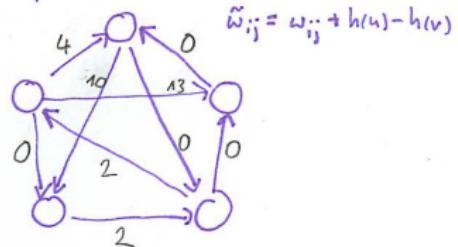
# Johnson's algorithm (14)

Solution:

Distances computed by Bellman-Ford:



Graph with new weights:



# Johnson's algorithm (15)

Running time of Johnson:

- ▶ One call to BellmanFord  $\sim O(V \cdot E)$
- ▶ One loop over vertices  $\sim O(V)$
- ▶ One loop over edges  $\sim O(E)$
- ▶ **V calls of Dijkstra**  $\sim O(VE + V^2 \log V)$  with Fibonacci heaps
- ▶ double loop (vertices and edges, line 12)  $\sim O(V \cdot E)$

Together:  $O(V \cdot E + V^2 \log V)$

- ▶ If graph is dense ( $E = \Theta(V^2)$ ), then this is pretty bad:  $O(V^3)$
- ▶ If graph is very sparse ( $E = \Theta(V)$ ), then this is reasonably good:  $O(V^2 \log V)$

## Johnson's algorithm (16)

Comment: Johnson's algorithm only makes sense if we want to solve the All-pairs shortest path problem, not for the single pair shortest path problem (why?)

# Point to Point Shortest Paths

# Point to Point Shortest Paths

Given a graph  $G$  and two vertices  $s$  and  $t$ , we want to compute the shortest path between  $s$  and  $t$  only.

First idea:

- ▶ Run  $Dijkstra(G, s)$  and stop the algorithm when we reached  $t$ .
- ▶ This has the same worst case running time as Dijkstra.
- ▶ However, in practice it is often faster because the running time is just in terms of the number of vertices and edges we actually visited.
- ▶ In general, one cannot do better in terms of worst case running time.

## Bi-directional Dijkstra

Literature:

I did not find any good textbook reference. A sketch of the algorithm can be found in the following paper:

A. Goldberg, C. Harrelson. Computing the shortest path:  $A^*$  search meets graph theory. In: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms, pages 156–165, 2005.

Original sources are somewhat hard to track. It has appeared in the literature in the 1970ies.

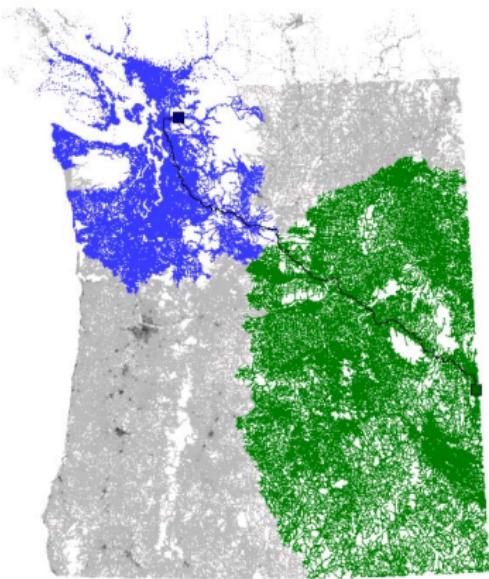
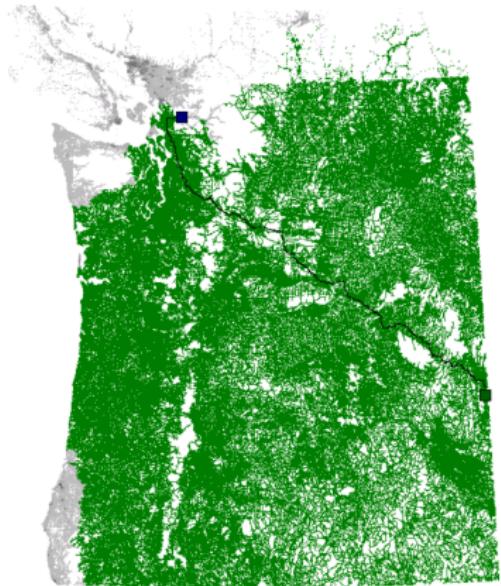
# Bi-directional Dijkstra

Idea:

- ▶ Instead of starting Dijkstra at  $s$  and waiting until we hit  $t$ , we start two copies of the Dijkstra algorithm, one at  $s$  (“forward search”) and one at  $t$  (“backward search”).
- ▶ We alternate between these two algorithms
- ▶ We stop when the two algorithms “meet”.

## Bi-directional Dijkstra (2)

Why should this help? Consider the following example  
(taken from slides of Andrew Goldberg):



Left: normal Dijkstra, right: bidirectional Dijkstra

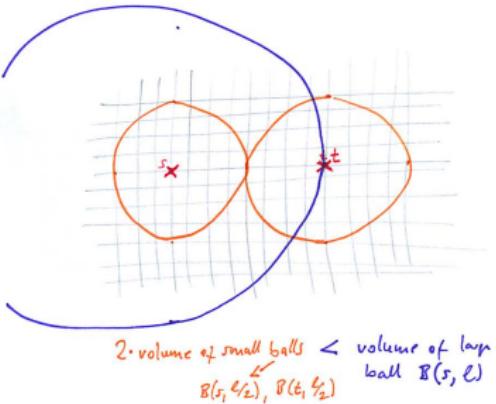
## Bi-directional Dijkstra (3)

Observe: Bidirectional  $\leadsto$  smaller portion of the graph.

# Bi-directional Dijkstra (4)

More formal example:

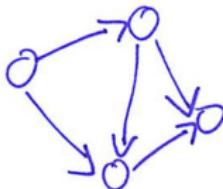
- ▶ Consider  $s$  and  $t$  as points on a  $k$ -dimensional grid, having distance  $d(s, t) := \ell$ .
- ▶ Standard-Dijkstra visits about  $(2k)^\ell$  vertices
- ▶ Bi-directional Dijkstra visits about  $2 \cdot k^\ell$  vertices
- ▶ Speedup of a factor  $2^{k-1}$



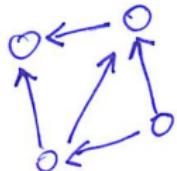
# Bi-directional Dijkstra (5)

How to perform bidirectional Dijkstra? General idea:

- If the graphs are directed: reverse all edges in the backward search starting from  $t$ .  $\sim G'$



original graph



reversed graph

- Run Dijkstra on  $G$ , starting from  $s$ , and Dijkstra on  $G'$  starting from  $t$ , until "they meet"

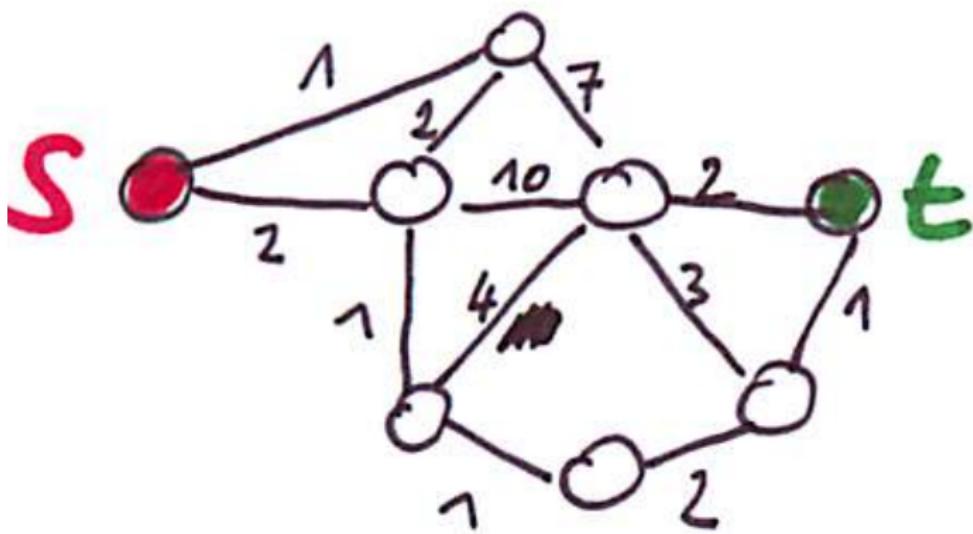
# Bi-directional Dijkstra (6)

## BidirectionalDijkstra( $G, s, t$ )

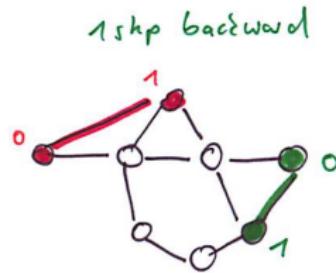
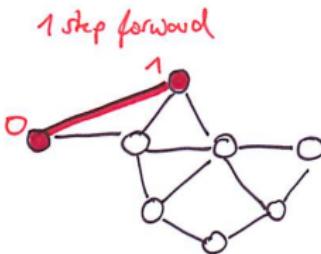
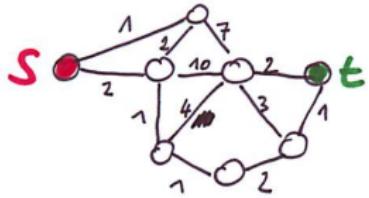
- ▶  $\mu = \infty$  (best path length currently known)
- ▶ Alternately run steps of  $Dijkstra(G, s)$  and  $Dijkstra(G', t)$ 
  - ▶ When an edge  $(v, w)$  is *scanned* by the forward search and  $w$  has already been visited by the backward search:
    - ▶ We have found a path between  $s$  and  $t$ , namely  $s \dots v \ w \dots t$ .
    - ▶ The length of this path is  $\ell = d(s, v) + w(v, w) + d(w, t)$
    - ▶ If  $\mu > \ell$ , set  $\mu = \ell$ .
  - ▶ Analogously for the backward search.
- ▶ We terminate when the search in one direction **selects** a vertex  $v$  that has already been selected in the other direction.
- ▶ We return  $\mu$  as the shortest path length between  $s$  and  $t$ .

## Bi-directional Dijkstra (7)

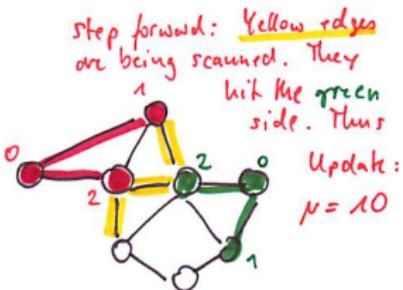
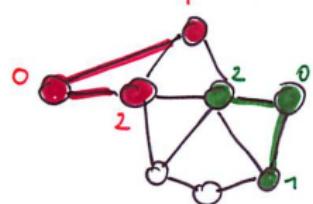
EVERYBODY PLEASE TRY THE FOLLOWING EXAMPLE:



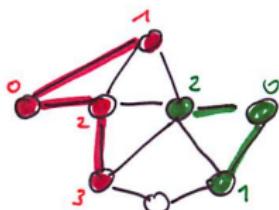
# Bi-directional Dijkstra (8)



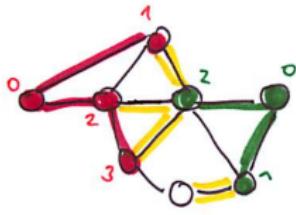
1 step forward, one step backward



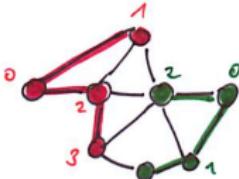
After this step:



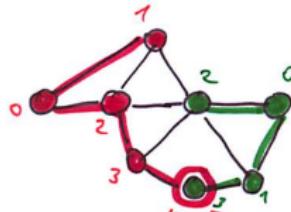
# Bi-directional Dijkstra (9)



step backward: Yellow edges  
are being examined.  
Update  $\mu = 9$   
(3rd yellow edge from top)



After this step:



Step forward: we select a  
vertex that has already  
been selected by the  
backward search.  
Update  $\mu = 7$  and stop.

# Bi-directional Dijkstra (10)

## Theorem 28

If  $t$  is reachable from  $s$ , then the algorithm

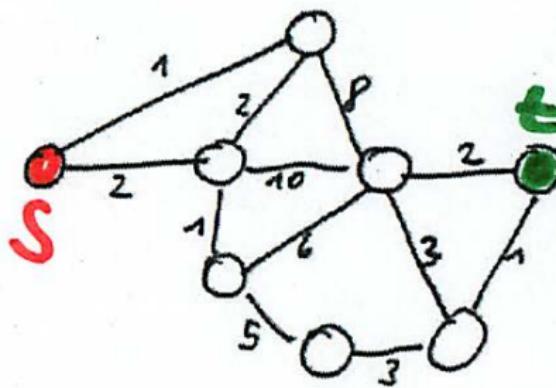
$\text{BidirectionalDijkstra}(G, s, t)$  finds an optimal path, and it is the path stored with  $\mu$ .

We omit the proof.

## Bi-directional Dijkstra (11)

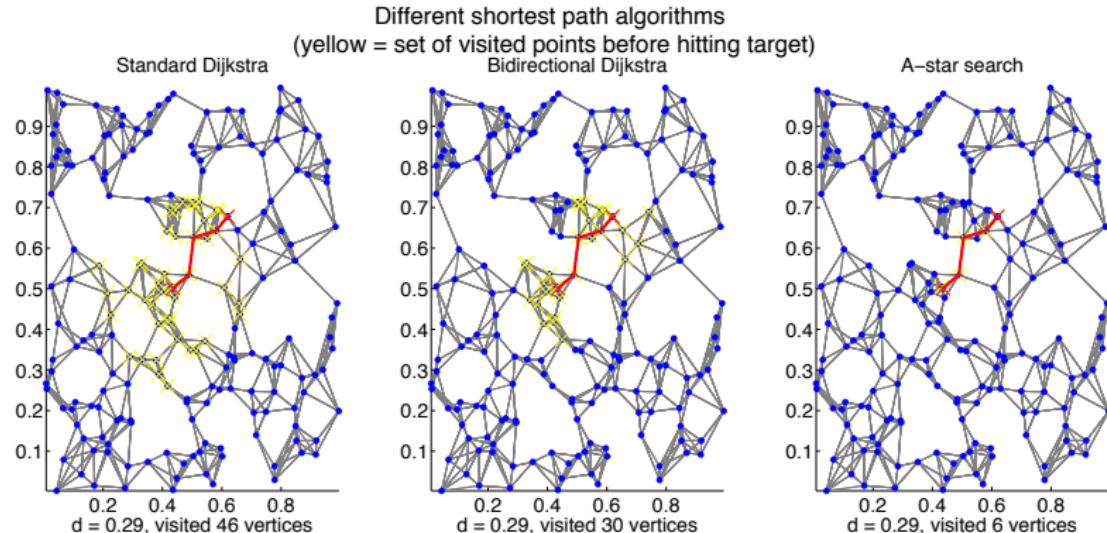
Important remark: It is not always true that if the algorithm stops at  $v$ , that then the shortest path between  $s$  and  $t$  has to go through  $v$ !!!

Here is an example graph where this is not the case (...exercise...):



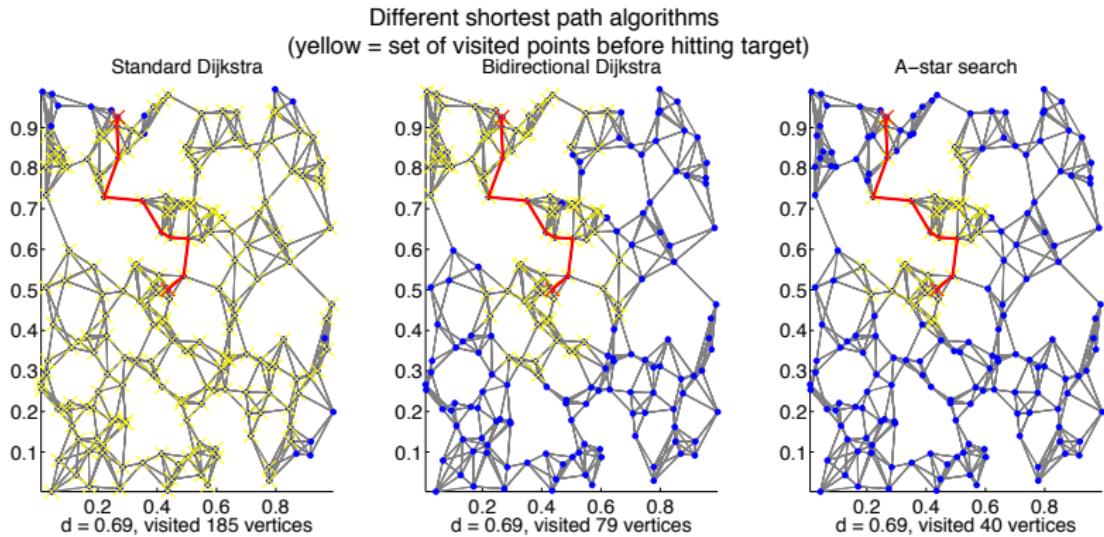
# Bi-directional Dijkstra (12)

Example 1: Comparing normal and bidirectional Dijkstra (for now, ignore the third fig.).



# Bi-directional Dijkstra (13)

Example 2: Comparing normal and bidirectional Dijkstra (for now, ignore the third fig.).



# More elaborate point to point algorithms

.... see  $A^*$  heuristics below ...

# Minimal spanning trees

Kleinberg Section 4.5; Mehlhorn/Sanders Section 11; Cormen Sec. 23













## History / Original sources

Kruskal's algorithm:

- ▶ J. Kruskal: On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society, 1956

Prim's algorithm:

- ▶ R. Prim. Shortest connection networks and some generalizations. Bell System Technical Journal, 1957.
- ▶ Apparently had already been invented by V. Jarnik in 1930.

Advanced methods were published in the 1980ies, e.g. by Tarjan and coworkers, and even more improvements in 2000 by Chazelle

# Hard problems

# Generic approaches to optimization

Mehlhorn/Sanders Section 12

# Greedy algorithms

# Local search

# Dynamic programming

# Branch and bound

# Outlook