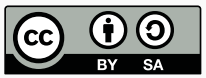
VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Algorithms – lecture presentations

Jiří Dvorský, Ph.D.

Presentation status to date April 3, 2025

Department of Computer Science
VSB – Technical University of Ostrava



Current version of presentations

The presentations are continuously, according to the needs of teaching, supplemented and updated. The current version of the presentations can always be found on the course website

www.cs.vsb.cz/dvorsky/Algorithms_Slides.html

Algorithms I

Algorithms I – Subject Syllabus

Introduction

Fundamentals of the Analysis of Algorithm Efficiency

Brute Force and Exhaustive Search

Decrease and Conquer

Brief outline of all lectures (cont.)

Reduction by a constant factor

Reduction by a variable factor

Divide and Conquer

Algorithms II

Algorithms II – Subject Syllabus

Transform and Conquer

Brief outline of all lectures (cont.)

Space and Time Trade-Offs

Dynamic Programming

Greedy Technique

Iterative Improvement

Limitations of Algorithm Power

Coping with Limitations of Algorithm Power

Other slides

Appendices

Algorithms I

Overall outline of all lectures (cont.)

- Basics of algorithmic problem solving
- Important Types of Problems
- Fundamental Data Structures
  - Linear Data Structures
  - Graphs
  - Trees
  - Sets and dictionaries

Fundamentals of the Analysis of Algorithm Efficiency

- Basics of algorithm complexity analysis
- Worst, Best and Average Case

Overall outline of all lectures (cont.)

- Asymptotic Notation of Complexity
- Analysis of Non-Recursive Algorithms
- Analysis of Recursive Algorithms

Brute Force and Exhaustive Search

- Sorting Algorithms
  - SelectSort
- Sequential search
- Brute force string matching
- Closest pair problem

Overall outline of all lectures (cont.)

- Convex hull of a set
- Exhaustive search
  - Traveling Salesman Problem
  - Knapsack problem
- Graph traversal
  - Depth-first graph traversal
  - Breadth-first graph traversal

Decrease and Conquer

- Sorting by insertion – Insertion Sort
- Topological sorting

Overall outline of all lectures (cont.)

- Generating combinatorial objects
  - Generating permutations
  - Generating subsets

Reduction by a constant factor

Reduction by a variable factor

Divide and Conquer

- Multiplication of Large Integers
- Strassen's Matrix Multiplication
- Closest Pair Problem
- Convex hull of a set

Algorithms II

Algorithms II – Subject Syllabus

- About Algorithms II
- Software
- Study Literature

Transform and Conquer

- Presorting
  - Unity of elements in the array
  - Module Calculation
  - Search

Overall outline of all lectures (cont.)

- Gaussian Elimination Method
  - *LU*-decomposition of a matrix
- Balanced Search Trees
  - AVL Trees
  - 2-3 trees
- Heap and Heap Sorting
- Horner's Scheme
- Problem Reduction

Space and Time Trade-Offs

- B-trees

Overall outline of all lectures (cont.)

- Searching for a key in a B-tree
- Inserting a key into a B-tree
- Deletion of a key from a B-tree

Dynamic Programming

- Warshall's algorithm

Greedy Technique

- Minimum Spanning Tree of a Graph
  - Prime's algorithm
  - Kruskal's algorithm
- Dijkstra's algorithm

Overall outline of all lectures (cont.)

Huffman code

Iterative Improvement

Limitations of Algorithm Power

Coping with Limitations of Algorithm Power

Other slides

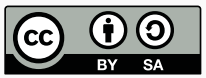
Appendices

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Algorithms I – Subject Syllabus

Jiří Dvorský, Ph.D.

Department of Computer Science
VSB – Technical University of Ostrava



Algorithms I – Subject Syllabus

About Algorithms I

Attention

For all the latest information on the subject, please see

http://www.cs.vsb.cz/dvorsky/

This presentation is for introductory lecture purposes only and will not be updated.

About Algorithms I

- The subject covers basic algorithmic problem solving strategies (brute force, divide and conquer, etc.) and typical examples of their use.
- Lectures are focused on **theory**.
- Seminars are focused on problem solution **implementation** using a given strategy in C or C++.
- Algorithms I are related to other subjects:
  - Introduction to programming – C language,
  - Functional programming – recursion and
  - Object-oriented programming – probably no commentary needed.

Time Allocation, Evaluation of the subject

Time Allocation

- Subject is taught in the summer semester of the first year of the bachelors study.
- There are
  - 2 hours of lectures and 2 hours of exercises per week in full-time form and
  - 6 tutorials in the combined form of study.

Evaluation – marked credit

- Marked credit is not an exam, it follows different rules.
- Please read the Study and Examination Regulations for Study in Bachelor'S and Master'S Degree Programmes at VSB - Technical University of Ostrava, Article 12.

Subject Guarantor

doc. Mgr. Jiří Dvorský, Ph.D.

Office: EA441

Email: *jiri.dvorsky@vsb.cz*

Web: *www.cs.vsb.cz/dvorsky*

What's the subject guarantor for?

The guarantor is responsible for the course of the entire subject, is responsible for teaching and correct evaluation of the assignments.

Prerequisities

- Prerequisites are a set of requirements that must be met in order for a student to enrol in a subject. Prerequisites are either formal or substantive.
- Formal prerequisites – none
- Substantive prerequisites:
  - knowledge from Introduction to Programming,
  - high school mathematics and
  - general orientation in IT.
- The subject Algorithms I is a mandatory prerequisity of the follow-up subject Algorithms II.

Attendance

Lectures

- Attendance at the lectures is **highly recommended**.

Seminars

- Attendance is **mandatory**.
- Attendance and activity at the seminars are evaluated.
- Sufficient scores must be obtained.

Consultation

- If you don’t understand something in class, need help with something or solve a problem with a lecture, seminars, tests, your absence from class, etc. it is possible to arrange a **individual consultation**.
- The consultation must be arranged in advance, for example by e-mail.
- If you need help with the material, prepare the materials you have studied on the topic, write down what is clear to you and where you are “stuck” and need advice.
- You don’t risk anything by consulting the teacher – at most you will learn what you need.

Algorithms I – Subject Syllabus

Fulltime Study

Lecture topics

1. General information about the subject
2. **Introduction**
   - 2.1 What Is an Algorithm?
   - 2.2 Fundamentals of Algorithmic Problem Solving
   - 2.3 Important Problem Types
   - 2.4 Fundamental Data Structures
3. **Fundamentals of the Analysis of Algorithm Efficiency**
   - 3.1 The Analysis Framework
   - 3.2 Asymptotic Notations and Basic Efficiency Classes
   - 3.3 Mathematical Analysis of Nonrecursive Algorithms
   - 3.4 Mathematical Analysis of Recursive Algorithms
4. **Brute Force and Exhaustive Search**

Lecture topics (cont.)

- 4.1 Selection Sort and Bubble Sort
- 4.2 Sequential Search and Brute-Force String Matching
- 4.3 Closest-Pair and Convex-Hull Problems by Brute Force
- 4.4 Exhaustive Search
- 4.5 Depth-First Search and Breadth-First Search

5. **Decrease-and-Conquer**
   - 5.1 Insertion Sort
   - 5.2 Topological Sorting
   - 5.3 Algorithms for Generating Combinatorial Objects
   - 5.4 Decrease-by-a-Constant-Factor Algorithms
   - 5.5 Variable-Size-Decrease Algorithms
6. **Divide-and-Conquer**
   - 6.1 Mergesort

Lecture topics (cont.)

6.2 Quicksort
6.3 Binary Tree Traversals and Related Properties
6.4 Multiplication of Large Integers and Strassen's Matrix Multiplication
6.5 The Closest-Pair and Convex-Hull Problems by Divide-and-Conquer

Seminars

- Semminars corresponds to lectures.
- In the seminar, students implement given tasks in C++ language.
- It is also possible to consult the lecture material.
- The seminar is not a substitute for lecture!
  - The seminars are not a “brief lecture” for those who do not attend lectures.
  - It is necessary to be prepared for the seminars.
  - The purpose of the seminar is not to prepare for the final exam.

Tasks

- The assessment consists of three parts:
  1. **Ongoing activities on seminars**
  2. **Project defense**
  3. **Final written test**
- All assignments are mandatory.
- A minimum grade is required for each assignment.

Tasks – Ongoing activities on seminars

- This part of the assessment is done **ongoing throughout the semester**.
- At each exercise, your activity is evaluated by the teacher. The activity is graded using a colour code:
  - green – the student actively participated in the seminar, was familiar with the material, he/she was able to carry out the assigned tasks,
  - orange – the student was rather passive in the seminar, he/she was not very well prepared for the seminar, his/her knowledge was limited, he/she had problems with the implementation of the tasks, and

Tasks – Ongoing activities on seminars (cont.)

  - **red** – the student was rather passive in the seminar, he/she was unable to complete the assignments. Unexcused absence from the exercise also falls into this category.

- Each colour code corresponds to a certain weight, that is reflected in the overall evaluation of all seminars.

Color code	Weight
green	1
orange	0.5
red	0

- At the end of the semester, an average weight is calculated, multiplied by the maximum number of points possible (30), and the result is your score.

Tasks – Ongoing activities on seminars (cont.)

- It is clear that all green codes correspond to the maximum number of points (30), while all red codes correspond to zero points.
- Activity points cannot be redeemed.

Example

The student A received a green rating on five seminars, orange on three and red on two ones. The average weight is calculated as:

$$\frac{5 \times 1 + 3 \times 0.5 + 2 \times 0}{5 + 3 + 2} = \frac{6.5}{10} = 0.65.$$

So the final score is $0.65 \times 30 = 19.5 \approx 20$ points.

Tasks – Project defense

- The project assignment will be published on the subject website at the beginning of April.
- Deadline for submission will be around credit week. The exact date will be published in the project assignment.
- The method of submission will be determined later.
- Project defenses will take place during the credit week and the exam period.
- Regardless of when the project defences take place, the version that has been submitted by the deadline is defended.
- The project defence cannot be repeated and the project will not be returned for revision.

Tasks – Final written test

- The test will take place during the exam period.
- All test dates will be announced in Edison system.
- Only students who have scored at least 10 points on their first attempt will be allowed to retake the test.

Number of points in the first attempt	Retake the test
0 to 9	no
10 to 20	yes
more than 21	not necessary

- You are allowed to write the final test a total of **two times**, in other words you are entitled to **one correction**. The course is completed with a marked credit not an exam – the rules are different.

Algorithms I – Subject Syllabus

Software

Software

Primary Software

- C++ Development Environment
- C++ Documentation

Additional Software

- Doxygen Documentation System, *www.doxygen.org*
- Typography System LaTeX, *www.ctan.org*

Development Environment for C++

- Microsoft Visual Studio Community 2022 is available for classroom use.
- I recommend this development environment for home study.
- In general, any development environment with a compiler that supports at least the **C++17** specification can be used.

Development Environment for C++ (cont.)

Remarks

1. The **Microsoft Visual C++** compiler and the **C++17** language specification will be used to evaluate your projects.
2. The C language is not identical to C++!
3. Beware of non-standard C++ language extensions implemented in the GNU C++ compiler.
   - For example, a variable length array is such an extension.
   - It is recommended to compile with the `-pedantic-errors` option enabled, see Options to Request or Suppress Warnings.

Algorithms I – Subject Syllabus

Study Literature

Study Literature

The study literature can be divided into two groups:

- **mandatory literature** – strategies of algorithmic problems solving and
- **recommended literature** – C++ programming language.

The study literature is shared across Algorithms I and Algorithms courses.

Mandatory Study Literature

1. LEVITIN, Anany. *Introduction to the Design and Analysis of Algorithms*. 3rd ed. Boston: Pearson, 2012. ISBN 978-0-13-231681-1.
2. CORMEN, Thomas H., Charles Eric LEISERSON, Ronald L. RIVEST a Clifford STEIN, 2022. *Introduction to algorithms*. Fourth edition. Cambridge, Massachusetts: The MIT Press. ISBN 978-026-2046-305.
3. SEDGEWICK, Robert, 1998. *Algorithms in C++*. 3rd ed. Reading, Mass: Addison-Wesley. ISBN 978-020-1350-883.

Recommended study literature

1. STROUSTRUP, Bjarne., 2013. The C++ programming language. Fourth edition. Upper Saddle River, NJ: Addison-Wesley. ISBN 978-0321563842.
2. CADENHEAD, Rogers a Jesse LIBERTY, 2017. Sams teach yourself C in 24 hours. Sixth edition. Indianapolis, Indiana: Pearson Education. ISBN 978-0672337468.

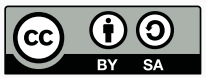
Thanks for your attention

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Introduction

Jiří Dvorský, Ph.D.

Department of Computer Science
VSB – Technical University of Ostrava



Introduction

What is an algorithm?

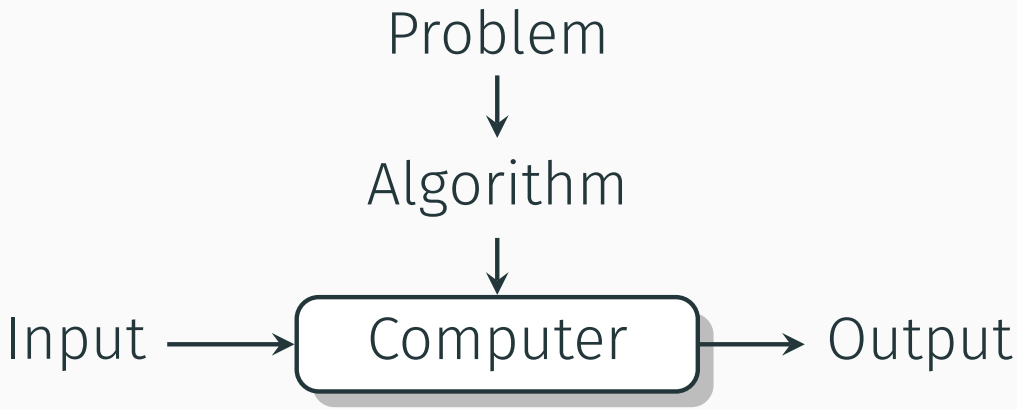
Why study algorithms?

- A professional developer/informatician should know standard algorithms for solving basic problems, be able to design new algorithms, and analyze the effectiveness of algorithms.
- Algorithms lead to the development of analytical thinking – it's about finding a precise and formal procedure for solving a problem.
- It's a universally applicable mental tool – **a person does not fully understand a problem until they can explain it to anyone else, let alone explain it to a computer**.
- The ability to formalize solutions leads to a much deeper understanding of the issue than if we simply tried to solve the problem, say, in an ad-hoc way.

What is an algorithm?

Algorithm

An algorithm is understood as a finite sequence of unambiguous instructions leading to the solution of a problem, i.e., leading to obtaining the desired output for any correct input in a finite time.



What is an algorithm? (cont.)

- The previous description of the concept of an algorithm is not a definition in the mathematical sense.
- We assume that there is something or someone who can understand “unambiguous instructions” and is able to follow them.
- For a correct definition, we would have to first clearly define what an unambiguous instruction is.
- A formal definition of an algorithm does not at all exist!

What is an algorithm? (cont.)

Remarks

- Automatic assumption – the algorithm will be executed by an electronic computer.
- The word **computer** means:
  1. today – electronic device,
  2. formerly – **calculator**, a person involved in numerical calculations.
- Although we will further assume that we will implement algorithms on an electronic computer, the concept of an algorithm itself does not depend on electronic computers.

Example of an Algorithm

- Three algorithms for solving the same problem – finding the greatest common divisor of two integers.
- Demonstration of several important facts:
  - adherence to the requirement of uniqueness of instructions,
  - the range of input values must be precisely specified,
  - the same algorithm can be represented in several different ways,
  - there can be multiple algorithms for solving one problem and
  - algorithms solving the same problem can be based on entirely different ideas, principles, and can differ significantly in the speed of solving the given problem.

Greatest Common Divisor (GCD)

- Let's have two non-negative integers m and n, of which at least one is also different from zero.
- The **greatest common divisor** $\gcd(m, n)$ is defined as the largest integer that divides both numbers m and n without a remainder.
- An algorithm for finding it was described in the book "Elements" by Euclid of Alexandria around the third century before our era.

Euclid's Algorithm

The algorithm is based on the repeated application of the relationship

$$\gcd(m, n) = \gcd(n, m \bmod n), \tag{1}$$

until the remainder $m \bmod n$ is equal to 0.

Because $\gcd(m, 0) = m$, the last value of m is equal to the desired greatest common divisor.

Euclid's Algorithm – Example

$$
\begin{aligned}
\gcd(60, 24) &= \gcd(24, 12) = \gcd(12, 0) = 12 \\
\gcd(24, 60) &= \gcd(60, 24) = \gcd(24, 12) = \gcd(12, 0) = 12 \\
\gcd(7, 3) &= \gcd(3, 1) = \gcd(1, 1) = \gcd(1, 0) = 1 \\
\gcd(3, 7) &= \gcd(7, 3) = \gcd(3, 1) = \gcd(1, 1) = \gcd(1, 0) = 1 \\
\gcd(13, 0) &= 13 \\
\gcd(0, 13) &= \gcd(13, 0) = 13
\end{aligned}
$$

Euclid's Algorithm – Stepwise Description

Step 1 If $n = 0$ then return the value m as the result and finish; otherwise continue with Step 2.

Step 2 Divide the number m by the number n, assign the remainder to r.

Step 3 Assign the value of the number n to m, the value of the number r to n. Continue with Step 1.

Euclid's Algorithm – Pseudocode

Input : Two non-negative integers m and n, at least one of which is non-zero

Output: The greatest common divisor of the numbers m and n, $\gcd(m, n)$

```
1 while n ≠ 0 do
2 |   r ← m mod n;
3 |   m ← n;
4 |   n ← r;
5 end
6 return m;
```

Algorithm of Successive Division

- The algorithm is based directly on the definition of GCD – GCD divides both given numbers m and n without a remainder.
- GCD cannot be greater than the smaller of the given numbers, so we can write $t = \min(m, n)$.
- If t divides both numbers m and n without a remainder, then $\gcd(m, n) = t$, otherwise the number t is decreased by 1 and the process is repeated.
- When does the algorithm stop?

Algorithm of Successive Division (cont.)

Example

For $m = 60$ and $n = 24$, we have $t = \min(60, 24) = 24$.
The algorithm first tries $t = 24$, then $t = 23$, and so on until it finally stops at $t = 12$.

Algorithm of Successive Division – Stepwise Description

Step 1 Assign to t the value of $\min(m, n)$.

Step 2 Divide the number m by the number t. If the remainder is equal to 0, proceed to Step 3; otherwise proceed to Step 4.

Step 3 Divide the number n by the number t. If the remainder is equal to 0, return the number t as the result and finish; otherwise proceed to Step 4.

Step 4 Decrease the value of the number t by 1 and proceed to Step 2.

Algorithm of Successive Division – Stepwise Description (cont.)

Error in the Algorithm

- The algorithm in this form does not work correctly if one of the numbers m and n is equal to 0. The number t would have a value of 0 and division by zero would occur.
- **Requirements for values entering the algorithm must be carefully specified!**

GCD – algorithm by prime factorization

Step 1 Perform the prime factorization of the number m.

Step 2 Perform the prime factorization of the number n.

Step 3 Find all common prime factors in the decompositions obtained in Step 1 and Step 2. The number of occurrences of a common prime factor p is equal to

$$\min(p_m, p_n),$$

where p_m and p_n are the numbers of occurrences of p in the decompositions of m and n, respectively,

GCD – algorithm by prime factorization (cont.)

Step 4 Calculate the product of all common prime factors and return this product as the result.

GCD – algorithm by prime factorization (cont.)

Example

For $m = 60$ and $n = 24$, the algorithm will proceed as follows:

$$\begin{aligned} 60 &= 2^2 \cdot 3^1 \cdot 5^1 \\ 24 &= 2^3 \cdot 3^1 \\ \gcd(60, 24) &= 2^2 \cdot 3^1 \\ &= 12 \end{aligned}$$

GCD – algorithm by prime factorization (cont.)

Problems

- The described algorithm is computationally much more demanding than the Euclidean algorithm.
- Finding GCD using prime factorization is not an algorithm – prime factorization of a number is not a “unique instruction”.
- Prime factorization requires a list of primes.
- Step 3 is also unclear – how to find common elements in the prime factorization? How to find common elements in two sorted lists of numbers?

The Sieve of Eratosthenes

- Solution to the problem of finding all prime numbers less than or equal to a number n, where $n > 1$.
- Origin in Greece, around 200 years before our era.
- First, we create a list of all natural numbers from 2 to n.
- Then, we take the numbers that remain in the list and exclude their multiples.
- We continue this way until no more numbers can be excluded from the list.
- The numbers that remain in the list are the desired prime numbers.

The Sieve of Eratosthenes (cont.)

Example

For $n = 25$ we get

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3		5		7		9		11		13		15		17		19		21		23		25
2	**3**		5		7				11		13				17		19				23		25
2	3		**5**		7				11		13				17		19				23		

The Sieve of Eratosthenes (cont.)

Stopping the Algorithm

- In the example, we last excluded multiples of the number 5.
- What will be, for a given n, the largest number p whose multiples we will exclude from the list?
- The first multiple will be $p \cdot p$, i.e., p^2.
- All lower multiples $2p, 3p, \ldots, (p-1)p$ have already been eliminated as multiples of other numbers: $2p$ as a multiple of 2, $3p$ as a multiple of 3, and so on.
- Furthermore, it is clear that $p^2 \leq n$ and thus $p = \lfloor \sqrt{n} \rfloor$, where $\lfloor x \rfloor$ denotes the nearest smaller natural number to x.

The Sieve of Eratosthenes (cont.)

Input: A natural number n

Output: The array of prime numbers $\leq n$

```
1  for p ← 2 to n do
2  |   A[p] ← p
3  end
4  for p ← 2 to ⌊√n⌋ do
5  |   if A[p] ≠ 0 then          // p has not been excluded yet
6  |   |   j ← p^2;
7  |   |   while j ≤ n do
8  |   |   |   A[j] ← 0;
9  |   |   |   j ← j + p;
10 |   |   end
11 |   end
12 end
```

The Sieve of Eratosthenes (cont.)

```
13 // Numbers that were not excluded from array A are
      copied to array L
14 i ← 0;
15 for p ← 2 to n do
16 |   if A[p] ≠ 0 then
17 |   |   L[i] ← A[p];
18 |   |   i ← i + 1;
19 |   end
20 end
```

NSD – algorithm for decomposition into prime factors

- By incorporating the Sieve of Eratosthenes, we obtain a regular algorithm for calculating the greatest common divisor using prime factorization.
- It remains to solve the problem when one or both numbers, for which we are calculating the greatest common divisor, is equal to 1...

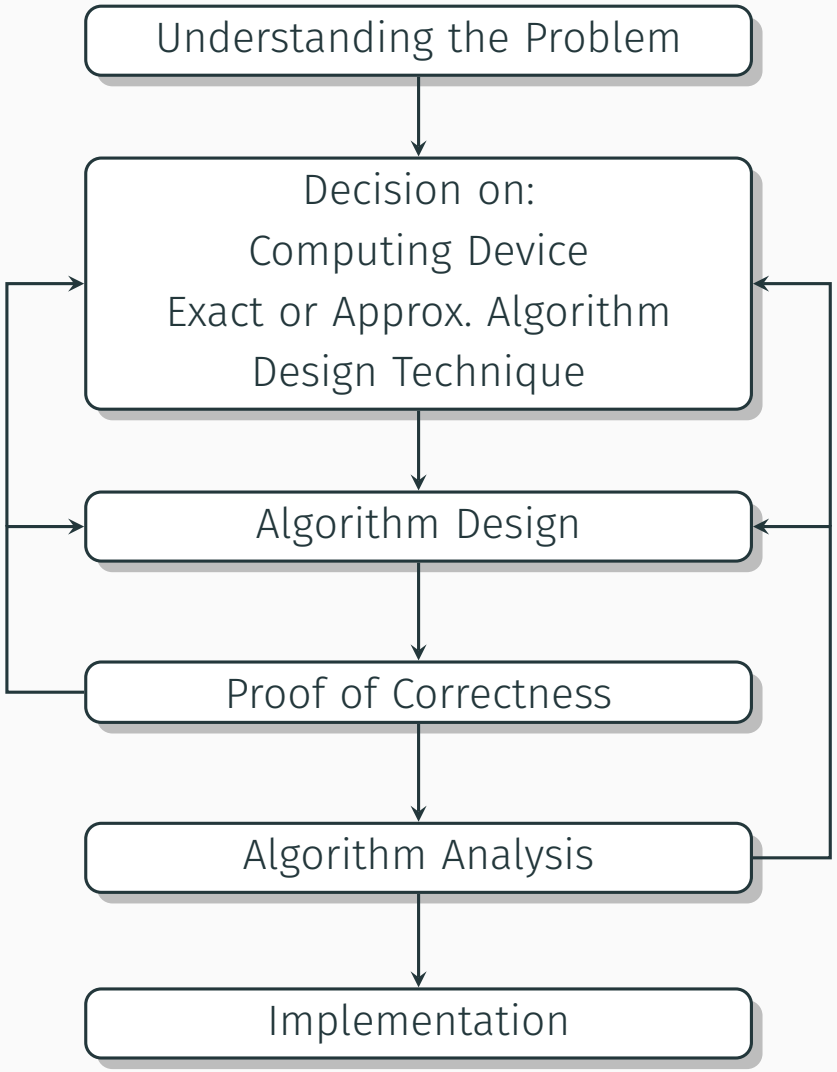
Introduction

Basics of algorithmic problem solving

Basics of algorithmic problem solving

- We consider algorithms as an **procedural, constructive** way to solve a given problem.
- Algorithms are not the solution to the problem themselves, but are instructions on how to obtain the solution.
- Computer science vs. mathematics – no existence of "infinitely small ε", "limits for n approaching infinity".
- Similarity between computer science and ancient Greek concept of geometry – solving using "ruler and compass", finite number of steps.

Process of Algorithm Design and Analysis



Understanding the Problem

- At first glance, a banality – incorrect understanding can backfire ⇒ necessity to rework the algorithm.
- Solving sample cases, special cases of solutions.
- Input data define an **instance of the problem**. Definition of permissible input data.
- A **correct algorithm** must **work correctly** for **all** permissible input data, not just for the **majority**.
- Knowledge of professional literature is an advantage – typical problems and their typical solutions.
- It's not always necessary to "reinvent the wheel".
- To select a suitable algorithm, it's good to know its strong and weak points.

Computing Devices

- Computing devices – a computer doesn't have to be just a "laptop".
- Parallel computing devices – multi-core processors, CUDA accelerators, parallel supercomputers.
- So far, the **von Neumann architecture** (John von Neumann 1946) prevails.
- In the following explanation, we will deal with **sequential algorithms** on the von Neumann architecture.
- **Random Access Machine** (RAM) – a theoretical model of the von Neumann computer architecture.

Computing Devices (cont.)

- For designing an algorithm and examining its effectiveness, it is suitable to use RAM – HW and SW independence.
- Practical implementation – it is necessary to take into account the HW and SW limitations of a specific computer.
- Assumption of sufficient performance of the used computer. Computer “stone age”.

Computing Devices (cont.)

Warning

"The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times;

premature optimization is the root of all evil

(or at least most of it) in programming."

Donald Knuth, The Art of Computer Programming

Exact vs. Approximate Solution of the Problem

- **Exact algorithm** – provides an exact solution.
- **Approximation algorithm** – provides an approximate solution.
  Use of approximation algorithms:
  1. There are important problems that we do not know how to solve exactly, e.g., aerodynamic and hydrodynamic problems.
  2. Exact algorithms are inherently unacceptably slow due to the enormous number of possible solutions, not due to a poor algorithm or implementation.
  3. An approximation algorithm is part of a sophisticated exact algorithm.

Remark

If we do not need to strictly distinguish between an algorithm for an exact and an approximate solution of the problem, we usually omit the adjective ”exact”.

Techniques of Algorithm Design

- We have everything we need: we understood the given problem, chose a computing device, and decided whether to use an exact or approximate algorithm.
- How do we proceed with designing an algorithm? What technique should we use for algorithm design?

Definition

Algorithm design technique (algorithm design strategy or paradigm) is a general approach to algorithmic problem-solving that can be applied to a wide range of problems from various areas of computer science.

Usefulness of Algorithm Design Techniques

1. they provide guidance on how to design algorithms for new problems, for which no satisfactory algorithm is known, and
2. allow for a clear classification of various algorithms according to their basic idea.

Techniques of Algorithm Design (cont.)

However, Keep in Mind That

- designing a specific algorithm for solving a specific problem can be a very challenging task,
- not all algorithm design techniques can be applied to a specific problem; sometimes it is necessary to combine techniques,
- it can be difficult to recognize which design technique an algorithm is based on,
- even if the technique is clear, assembling the algorithm often requires non-trivial effort and ingenuity, but

Techniques of Algorithm Design (cont.)

- with increasing developer experience, everything becomes easier and easier, although rarely easy.

Importance of Data Structures

- a suitable data structure has fundamental importance for the designed algorithm – Eratosthenes' sieve versus linked list
- some algorithm design techniques strongly depend on the structure or reorganization of the data that determine the instance of the problem being solved,
- Niklaus Wirth: "Algorithms + Data Structures = Programs"

Methods of Recording an Algorithm During its Design

Natural Language

- does not have to be a written record – an orally formulated idea
- possible ambiguities – extreme case “The woman beats the machine with a stick.”
- ability to precisely formulate thoughts, formulate them logically correctly, define concepts describing the problem, classify concepts into a thought schema etc.

Methods of Recording an Algorithm During its Design (cont.)

Pseudocode

- a mix of natural language and constructs similar to programming languages.
- usually more precise and concise than natural language
- more concise notation of the proposed algorithm
- there are many mutually similar “dialects” of pseudocode

Methods of Recording an Algorithm During its Design (cont.)

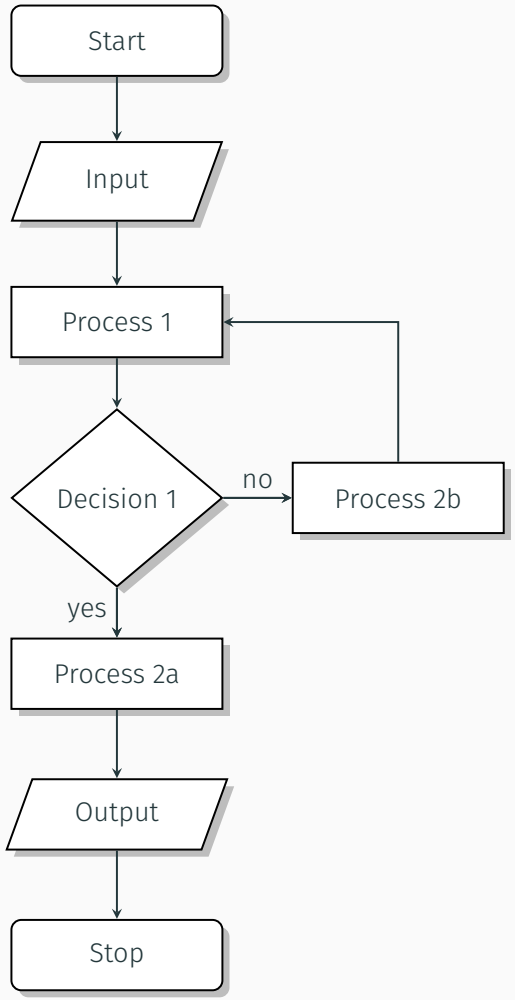
Programming Language

- another possible way of recording
- this record is considered more as an implementation

Methods of Recording an Algorithm During its Design (cont.)

Development Diagram

- Engl. flowchart
- graphical form of algorithm recording
- no longer used today



Proof of Algorithm Correctness

Definition

An algorithm is considered **correct**, if for every correct input it provides a correct result in finite time. For incorrect input, the behavior of a correct algorithm is not defined.

- The usual method of proof is mathematical induction.
- Proof of correctness vs. incorrectness of an algorithm
  - For a correct proof of algorithm correctness, it is not enough to prove correctness for **some** instance of the problem, we must be able to prove correctness for **all** instances of the problem, and vice versa

Proof of Algorithm Correctness (cont.)

    - as a proof of incorrectness of an algorithm, it is enough to find **one** instance of the problem, so that we can declare the algorithm faulty.
- Correctness of an approximation algorithm – the error of the algorithm's result does not exceed a predefined limit.

Analysis of the Algorithm – Examined Properties

Correctness – already solved

Time Complexity (English: **time complexity**)

- ”how fast the algorithm works”
- speed is not measured in time units, but by the amount of instructions performed by the algorithm (the same algorithm on faster and slower HW)

Space Complexity (English: **space complexity**)

- ”how much memory the algorithm needs”
- measured in bytes and multiples

Analysis of the Algorithm – Examined Properties (cont.)

Simplicity (English: **simplicity**)

- cannot be exactly defined, unlike complexity,
- rather a subjective matter – beauty, elegance (NSD Euclid's algorithm vs. prime factorization),
- simpler algorithm – easier to understand, implement, likely fewer errors,
- simpler algorithm – does not necessarily have lower complexity,
- use – typically software prototype. If it does not meet the requirements – transition to an algorithm with lower complexity. But! "Premature optimization..."

Analysis of the Algorithm – Examined Properties (cont.)

- terminology – the opposite of simplicity is not ”complexity” of the algorithm, but rather complicatedness, incomprehensibility, inappropriateness of design.

Generality

1. generality of the proposed algorithmic solution – solve the problem very generally or take into account possible simplifications in a specific case?
    - solving a more general problem is easier than a specific one – e.g. inseparability of two numbers, solution via NSD, NSD is a more general problem

Analysis of the Algorithm – Examined Properties (cont.)

   - solving a more general and specific problem at the same level – e.g. finding the median, solution via sorting (more general) and a specific algorithm
   - solving a more general problem is significantly more difficult – e.g. quadratic equation $ax^2 + bx + c = 0$ versus a general algebraic equation of degree n.

2. generality of the problem instance – the algorithm design should handle all reasonably expected, natural instances of the problem.
   - For NSD, it is not natural to exclude the number 1, but
   - for a quadratic equation, we usually assume that a, b, and c are real numbers – more generally, we can also consider complex numbers.

Analysis of the Algorithm – Examined Properties (cont.)

If we are not satisfied with the complexity or simplicity of the design or generality of the algorithm?

There is nothing else to do but go back to the beginning, sit down at the table, take a pencil and paper in hand, and think, draw, search in literature, and so on.

"The designer knows he has achieved perfection when he can no longer add or remove anything."

Antoine de Saint-Exupéry

"Keep It Simple, Stupid!"

Kelly Johnson

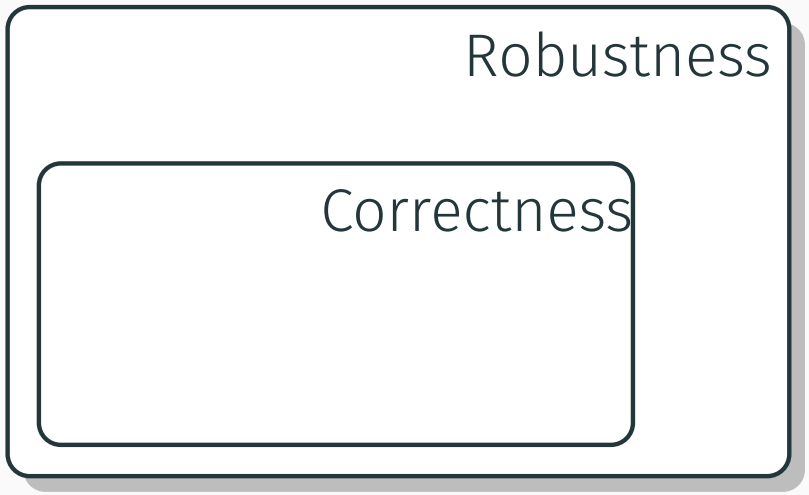
Algorithm Coding

- Again, an underestimated phase – “We have the algorithm figured out, so now we just rewrite it on the computer and we’re done.”
- We implement the algorithm either incorrectly or inefficiently, or even both options occur at the same time.
- In real life – the correctness of programs is verified by testing.
- Program testing is “an artistic craft”.
- Another critical point – data entry.
  - School – input data define a correct instance of the solved problem.
  - Practice – the question of controlling input data needs to be addressed.

Algorithm Robustness

Definition

We consider an algorithm to be **robust** if it is correct and for every incorrect input, it issues an error report and is able to recover from the error.



Efficiency of Implementation

- Correctness of algorithm implementation is a necessity.
- But even a correct implementation can be done inefficiently, the computer's performance is not utilized as it could be.
- Code optimization:
  1. manual – calculation of loop invariant, replacement of common subexpressions with a variable.
  2. automatic – optimization algorithms built into compilers, e.g. register allocation.
- By optimizing the code, the program's efficiency can be improved by some constant factor, e.g. 10%.

Efficiency of Implementation (cont.)

- For radical, order-of-magnitude improvement, it is necessary to implement an algorithm with lower complexity.
- Searching for a better and better algorithm is an interesting mental adventure...
- The question is when to stop. Perfection is an expensive luxury. Engineering approach – resources allocated for the project.
- Academic question of **algorithm optimality**: "What is the smallest possible complexity of any algorithm that solves a given problem?"

Efficiency of Implementation (cont.)

- For example, a sequential algorithm for sorting an array with n elements – at least $n \log_2 n$ comparisons.
- Can every problem be solved by an algorithm?
  Undecidable problems – cannot be solved by any algorithm.
- Fortunately, most practical problems can be solved algorithmically.

  A good algorithm is the result of repeated effort and multiple reworkings.

Introduction

Important Types of Problems

Important Types of Problems

- Sorting
- Searching
- String Processing
- Graph Problems
- Combinatorial Problems
- Geometric Problems
- Numerical Problems

Sorting

- Sorting in computer science – rearranging elements into a non-decreasing sequence. Compare with waste sorting.
- A **relation of order** must be defined between the elements, i.e., the relation ”less than or equal to”, $\leq$.
- In practice, we sort numbers, strings, or structured records.
- For a record, we must define a **key**, i.e., the part of the record that we sort by, for which an order is defined. The key does not have to be defined explicitly, e.g., for numbers, it is the number itself.

Arrangement

Definition

Let us have a binary homogeneous relation $\rho \subseteq A \times A$ on the set A.

- The relation ρ is called (non-strict) **partial ordering**, if it is simultaneously reflexive, antisymmetric and transitive.
- The relation ρ is called (non-strict) **total ordering**, if it is simultaneously reflexive, antisymmetric, transitive and total.
- The relation ρ is called (partial) **strict ordering**, if it is simultaneously asymmetric (and therefore also antisymmetric and irreflexive) and transitive.
- The relation ρ is called **total strict ordering**, if it is simultaneously asymmetric (and therefore also antisymmetric and irreflexive), transitive and connected.

Arrangement – notes

- We standardly denote a non-strict arrangement by $\leq$, and a strict arrangement by $<$.
- Instead of the term **partial arrangement**, we sometimes use just **arrangement**.
- Instead of the term **complete arrangement**, we also use the terms **total** or **linear** arrangement.
- If $\leq$ is an arrangement on a set A, then we call the relational system $(A, \leq)$ an **ordered set** (Eng. **poset** – partially ordered set). A completely ordered set is called a **chain** (Eng. **chain**).

Arrangement – notes (cont.)

- Two different elements x, y are **comparable** in the arrangement $\leq$, if $(x \leq y) \vee (y \leq x)$ holds. Otherwise, the elements are **incomparable**. In a complete arrangement, all pairs of elements are comparable.
- The intersection of arrangements is again an arrangement. The union of arrangements does not have to be an arrangement in general.
- The relationship between strict and non-strict arrangements can be written as follows: " $\leq$ " = " $<$ " $\cup$ " = ", i.e., by adding the identity relation ("equality") to the strict arrangement.

Used properties of binary homogeneous relations

Used properties of relation $\rho \subseteq A \times A$ $\forall x, y, z \in A$:

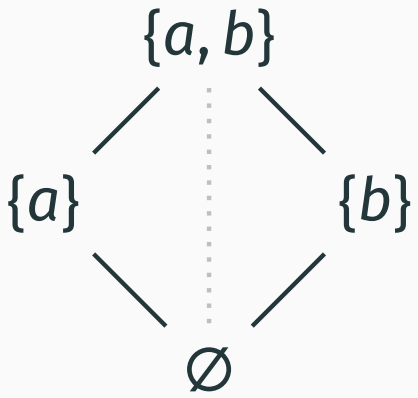
- reflexivity: $x \rho x$,
- irreflexivity: $\neg(x \rho x)$,
- asymmetry: $x \rho y \Rightarrow \neg(y \rho x)$,
- antisymmetry: $x \rho y \wedge y \rho x \Rightarrow x = y$,
- transitivity: $x \rho y \wedge y \rho z \Rightarrow x \rho z$,
- connectivity: $[x \neq y \Rightarrow x \rho y \vee y \rho x]$,
- completeness: $x \rho y \vee y \rho x$.

Hasse Diagram

The ordering relation is typically represented using a **Hasse diagram**, which

- represents the relation of immediate precedence without transitive edges, which is the same for both strict and non-strict orderings and which
- corresponds to a directed graph, where all edges are oriented from bottom to top.

Example



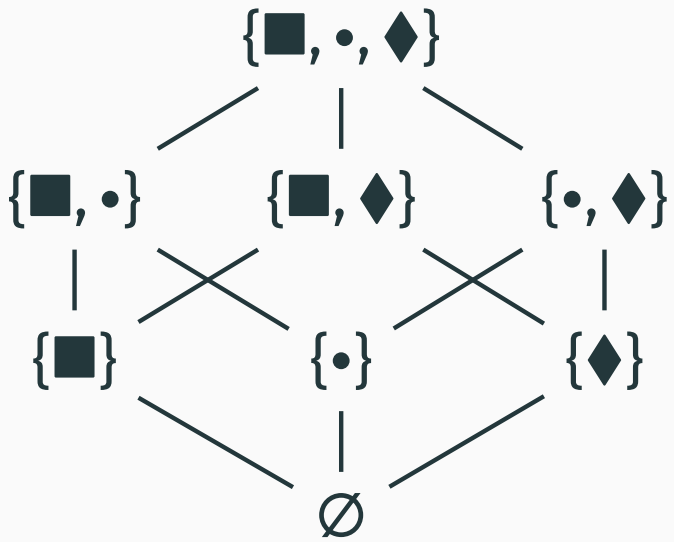
Hasse diagram for the ordering relation "to be a subset" on the set $\{a, b\}$. A transitive edge, which is normally not shown, is displayed dashed.

Partial ordering – example

For any set A we can define an ordering $\leq$ of inclusion on the set of its subsets $P(A)$: $X \leq Y$ if $X \subseteq Y$, where $X, Y \in P(A)$.

The ordering defined in this way is not complete but only partial, because it contains incomparable elements.

Example



In $A = \{■, •, ♦\}$ the incomparable elements are

- all one-element subsets among themselves and
- all two-element subsets among themselves.

Total Ordering – Example

- The usual relation $<$ on the set of natural, integer, rational, and real numbers is a total ordering.
- Alphabetical, lexicographical ordering of strings is also a total ordering.
- Properly nested matryoshka dolls are totally ordered using the relation “being inside”. But only under the condition that no more than one smaller doll can fit inside another at a time – otherwise, we get only a partial ordering.

Sorting – utilization

- A sorted list of values is the desired output – a race result list, internet search results.
- For some tasks, it is better to solve for a sorted input – typically **search**. Phone book. Geometric tasks. Data compression. Greedy algorithms.

Definition of the Sorting Problem

- Let's assume a sequence of elements $A = a_1, a_2, \ldots a_n$. The task of sorting is to find a permutation $\pi : \mathbb{N} \to \mathbb{N}$ such that $a_{\pi_i} \leq a_{\pi_{i+1}}$ for all $1 \leq i < n$.
- The permutation π cannot be found directly, as there are $n!$ permutations of n elements.
- We will understand sorting algorithms as algorithms that construct the permutation π step by step, for example by comparing and swapping elements.

Definition of the sorting problem – example

Let us have a sequence $A = ebfcda$ and the usual alphabetical ordering of letters. The sought permutation is

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 2 & 4 & 5 & 1 & 3 \end{pmatrix}$$

Then

$$a_{\pi_1} < a_{\pi_2} < a_{\pi_3} < a_{\pi_4} < a_{\pi_5} < a_{\pi_6}$$

$$a_6 < a_2 < a_4 < a_5 < a_1 < a_3$$

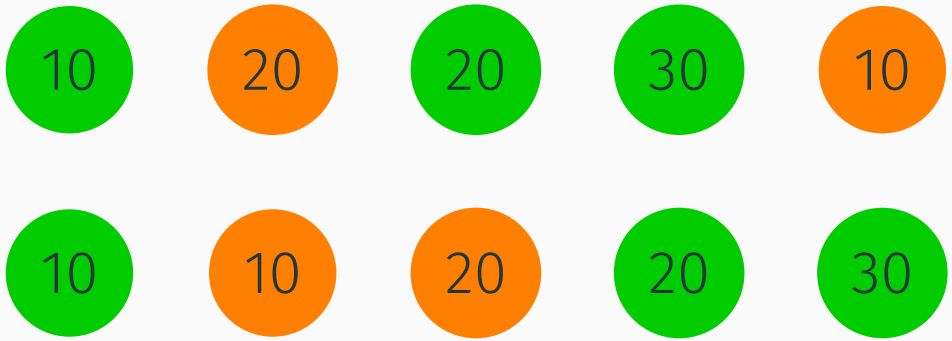
$$a < b < c < d < e < f$$

Sorting

- A number of sorting algorithms have been developed. There is no single universal algorithm for all situations.
  - simple and slow vs. complex and fast,
  - random vs. nearly sorted sequence on input
  - internal memory vs. external memory.
- Given n elements, the minimum number of comparisons is $n \log_2 n$ for serial algorithms based on comparison and swapping.

Sorting (cont.)

- **Stable sorting** – preserves the relative positions of elements. If we have two elements with the same key in positions i and j, where $i < j$, then after sorting, these elements will be in positions i' and j', where $i' < j'$.



Hint: follow the relative positions of orange and green numbers.

Algorithms that sort using exchanges over long distances are usually faster, but not stable.

Sorting (cont.)

- **In-situ** sorting – a sorting algorithm only needs memory for storing elements plus additional memory of constant scope, i.e., this memory does not depend on the number of sorted elements, typically variables for loop iteration, logical flags, etc.
- **Natural** sorting – the complexity of the sorting algorithm increases with the degree of unsortedness of the input data.

Degree of disorder of a data sequence

- The goal is to find a measure of disorder, ”messiness”, of a sequence of n elements that we need to sort.
- Sorted sequences should correspond to **zero** disorder.
- Sequences sorted in reverse order should correspond to **maximum** disorder.
- Other sequences should fall between these extreme possibilities

Rate of non-monotonicity of a permutation

- Permutation of numbers $1 \ldots n$.
- Identity permutation – zero non-monotonicity

$$\pi_{id} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}$$

- Reverse permutation – maximum non-monotonicity

$$\pi_{rev} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 5 & 4 & 3 & 2 & 1 \end{pmatrix}$$

- Non-monotonicity of a permutation will be measured by the **number of inversions** of the given permutation.

Inverse in Permutation

Definition

Let's have a permutation $\pi : \mathbb{N} \to \mathbb{N}$. The inverse in the permutation π is a pair of elements i, j such that $i < j$ and simultaneously $\pi_i > \pi_j$.

The inverses in the permutation can be freely interpreted as "a larger element is at a smaller index and at the same time a smaller element is at a larger index".

Example

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 3 & 2 \end{pmatrix}$$

All permutations in π

$1 < 2 \wedge 4 > 1$ $\quad$ $1 < 4 \wedge 4 > 2$

$1 < 3 \wedge 4 > 3$ $\quad$ $3 < 4 \wedge 3 > 2$

Number of inversions in a permutation

- Identity permutation – the total number of inversions is zero
- Reverse permutation

Element	Inversions with elements	Number of inversions
n	$n-1, n-2, n-3, \dots, 1$	$n-1$
$n-1$	$n-2, n-3, \dots, 1$	$n-2$
$\vdots$	$\vdots$	$\vdots$
3	2, 1	2
2	1	1
1	–	0

The total number of inversions is equal to

$$(n-1) + (n-2) + \cdots + 2 + 1 + 0 = \frac{1}{2} n(n-1)$$

Average number of inversions in a permutation

- Let's denote C_n as the total number of inversions in all permutations of n elements. First, we derive a relationship between C_n and C_{n-1}.
- Consider all permutations of $n - 1$ elements. To all these permutations, we add n after the last element of the permutation. The number of inversions does not increase and will be equal to C_{n-1}.
- To the permutations of $n - 1$ elements, we add n after the second-to-last element of the permutation. The number of inversions increases by one for each permutation, so $C_{n-1} + 1 \cdot (n - 1)!$

Average number of inversions in a permutation (cont.)

- Finally, to the permutations of $n - 1$ elements, we add n before the first element of the permutation. The number of inversions increases by $n - 1$ for each permutation, so $C_{n-1} + (n - 1)(n - 1)!$

Therefore

$$\begin{aligned} C_n = {} & C_{n-1} + 0 \cdot (n-1)! + \\ & C_{n-1} + 1 \cdot (n-1)! + \\ & C_{n-1} + 2 \cdot (n-1)! + \\ & \vdots \qquad\qquad \vdots \\ & C_{n-1} + (n-1)(n-1)! \end{aligned}$$

Average number of inversions in a permutation (cont.)

From this

$$\begin{aligned} C_n &= nC_{n-1} + \big[0 + 1 + \cdots + (n-1)\big](n-1)! \\ &= nC_{n-1} + \left[\frac{1}{2}n(n-1)\right](n-1)! \\ &= nC_{n-1} + \frac{1}{2}(n-1)n! \end{aligned}$$

The average number of inversions I_n is equal to

$$I_n = \frac{C_n}{n!}.$$

Average number of inversions in a permutation (cont.)

From this, we substitute $C_n = n!I_n$ and $C_{n-1} = (n-1)!I_{n-1}$ to get

$$\begin{aligned} n!I_n &= n(n-1)!I_{n-1} + \frac{1}{2}(n-1)n! \\ &= n!I_{n-1} + \frac{1}{2}(n-1)n! \end{aligned}$$

After cancelling $n!$ we get

$$I_n = I_{n-1} + \frac{1}{2}(n-1)$$

Average number of inversions in a permutation (cont.)

Expanding the expression for I_n

$$\begin{aligned} I_n &= I_{n-2} + \frac{1}{2}(n-2) + \frac{1}{2}(n-1) \\ &= I_{n-3} + \frac{1}{2}(n-3) + \frac{1}{2}(n-2) + \frac{1}{2}(n-1) \\ &\vdots \\ &= I_{n-i} + \frac{1}{2}(n-i) + \cdots + \frac{1}{2}(n-2) + \frac{1}{2}(n-1) \end{aligned}$$

Furthermore, we know that a one-element permutation cannot have an inversion, so $I_1 = 0$.

Average number of inversions in a permutation (cont.)

Now, we are looking for such i, so that the expression $n - i$ in the index I_{n-i} equals 1. Obviously, $i = n - 1$ and therefore

$$
\begin{aligned}
I_n &= I_{n-(n-1)} + \frac{1}{2}\big[n - (n-1)\big] + \frac{1}{2}\big[n - (n-2)\big] + \cdots + \frac{1}{2}(n-2) + \frac{1}{2}(n-1) \\
&= I_1 + \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 2 + \cdots + \frac{1}{2}(n-2) + \frac{1}{2}(n-1) \\
&= I_1 + \frac{1}{2}\big[1 + 2 + \cdots + (n-2) + (n-1)\big] \\
&= I_1 + \frac{1}{2}\left[\frac{1}{2}n(n-1)\right] \\
&= I_1 + \frac{1}{4}n(n-1)
\end{aligned}
$$

Average number of inversions in a permutation (cont.)

And since $I_1 = 0$, we finally get

$$I_n = \frac{1}{4} n(n-1)$$

Summary – number of inversions in a permutation of n elements

Minimum	0
Average	$\frac{1}{4} n(n-1)$
Maximum	$\frac{1}{2} n(n-1)$

Searching

- Basic task – finding an element a in a given set M, or multiset.
- Mathematically – does $a \in M$ hold, or $a \notin M$?
- Mathematics does not deal with the complexity of this operation.
- There are numerous search algorithms – sequential, interval halving, hashing...
- There is no optimal algorithm for all situations, algorithms have different assumptions – more memory for faster work, sorted array...
- Important aspects:
  - the mutual ratio of search, insert, and delete operations on the set – does searching prevail or is the ratio balanced?
  - organization of very large data.

String Processing

- String – a sequence of characters from a given alphabet.
- Typical examples of strings:
  - text strings, alphabet composed of letters, digits, and punctuation,
  - bit strings composed of 0 and 1 or
  - genetic strings composed of the characters A, C, G, and T
- Applications
  - text processing,
  - data compression,
  - programming languages and compilers or
  - string searching (pattern matching) – finding one string, pattern, or patterns in another string. A trivial example – `Ctrl+F` in a text editor.

Searching in Text – Pattern Matching

- When **searching in text**, we determine whether a given **pattern/patterns** matches, coincides with, a part of a given **text**. We can also say that we are looking for **occurrences of the pattern in the text**.
- Applications:
  - in text editors (moving in edited text, replacing strings),
  - in utilities like *`grep`*, which allow finding all occurrences of specified patterns in a set of text files,
  - web search,
  - when examining DNA,
  - when analyzing images, sound, etc.

Text Search – Classification of Search Algorithms

		Text Preprocessing	
		no	yes
Sample Preprocessing	no	brute force search	index-based methods, typically web search engines, generally known as Information Retrieval Systems
	yes	advanced search algorithms	signature-based search methods

Text Search – Additional Division Criteria

Number of searched patterns – one, finite number or infinite number of patterns

Number of occurrences – first occurrence, all occurrences

Comparison method – exact search versus approximate search, where deviations between the pattern and text are allowed, e.g., one character may differ

Search direction – in text, we usually proceed from lower indices to higher, “from left to right”

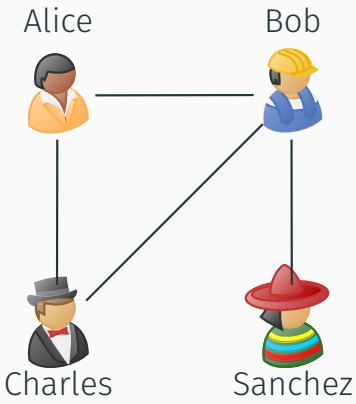
- symmetrical algorithms – the pattern is traversed in the same direction
- asymmetrical algorithms – the pattern is traversed in the opposite direction.

Searching in Text – Notation

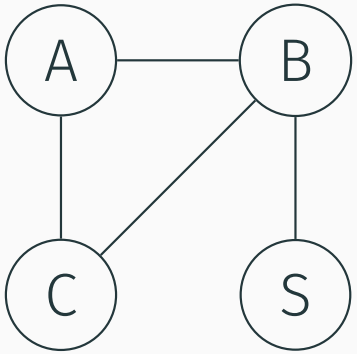
In the following text, we will use the following notation:

- p the searched pattern, $p = p_0 p_1 \ldots p_{m-1}$, where $|p| = m$ is the length of the pattern,
- t the searched text, $t = t_0 t_1 \ldots t_{n-1}$, where $|t| = n$ is the length of the text,
- Σ – the alphabet from which the pattern and the text are composed,
- σ – the size of the alphabet Σ ($\sigma = |\Sigma|$),
- $\bar{C}_n$ – the expected number of comparisons needed to find the pattern in a text of length n.

Graph Problems



Taken from [1]



- **Graph** – informally, a set of points, **vertices**, some of which are connected by line segments, **edges**.
- Applications – representation of transportation networks, project management, social networks, electrical networks, etc.
- Basic problems:
  - graph traversal – can we reach all vertices in the graph?
  - shortest path – the shortest path between two cities

Graph Problems (cont.)

    - topological sorting – organization of a project, activities must depend on each other, can something be done in parallel?
- Computationally complex problems
    - **Traveling Salesman Problem** (TSP) – the task is to find the shortest path between n cities, visiting each one exactly once. Logistics, microchip manufacturing.
    - **Graph Coloring Problem** – the task is to find the smallest number of colors for vertices such that no two adjacent vertices have the same color. Planning – events correspond to vertices, edges connect events that cannot be performed simultaneously, solving the graph coloring problem provides an optimal schedule.

Combinatorial Problems

- The essence of problems – finding a **permutation**, **combination** or **subset** from a given set of objects that satisfies certain **constraints** and possibly has some other property, such as minimizing or maximizing some function.
- The Traveling Salesman Problem – the order of visited cities is a permutation, the minimized function is the total distance.
- Perhaps the most complex problems in computer science from both theoretical and practical perspectives:
  - the number of possible candidate solutions (e.g., permutations) grows very rapidly and reaches enormous values even for moderately sized problems

Combinatorial Problems (cont.)

  - no algorithm is known to find an exact solution in an acceptable amount of time, and
  - it is not even known whether such an algorithm exists; it is assumed that it does not.

- However, some combinatorial problems **can** be solved efficiently – for example, finding the shortest path.

Geometric Problems

- They process points, line segments, polygons and similar objects.
- These are actually the first algorithms – Euclidean geometry, constructions with ”ruler and compass”.
- Applications:
  - computer graphics,
  - computer games,
  - robotics,
  - medicine.
- In our subject:
  - closest pair problem – a set of points in a plane, find two points with minimum distance,
  - convex hull of a set of points – find the smallest convex polygon containing the given points.

Numerical Tasks

- Solving systems of equations, calculating function values, definite integrals, etc.
- Most of these tasks require calculations with real numbers. Typical problems:
  - The computer can only capture a limited range of numbers (not ∞) and with limited precision ($\frac{1}{3}, \pi$) and
  - Accumulation of rounding errors.
- Scientific and technical calculations – the classic application of early computers. Engineering applications.
- Today – data storage and analysis, navigation, logistics...
- In our subject – several typical tasks, solving a system of equations, matrices.

Introduction

Fundamental Data Structures

Fundamental Data Structures

- A **data structure** can be defined as a way of organizing interrelated data.
- The choice of data structure strongly depends on the problem being solved.
- Several particularly important data structures exist:
  - linear data structures – **array**, **linked list**, **stack**, **queue**, **priority queue**
  - **graph**
  - **tree**
  - **set**
  - **dictionary**

Array

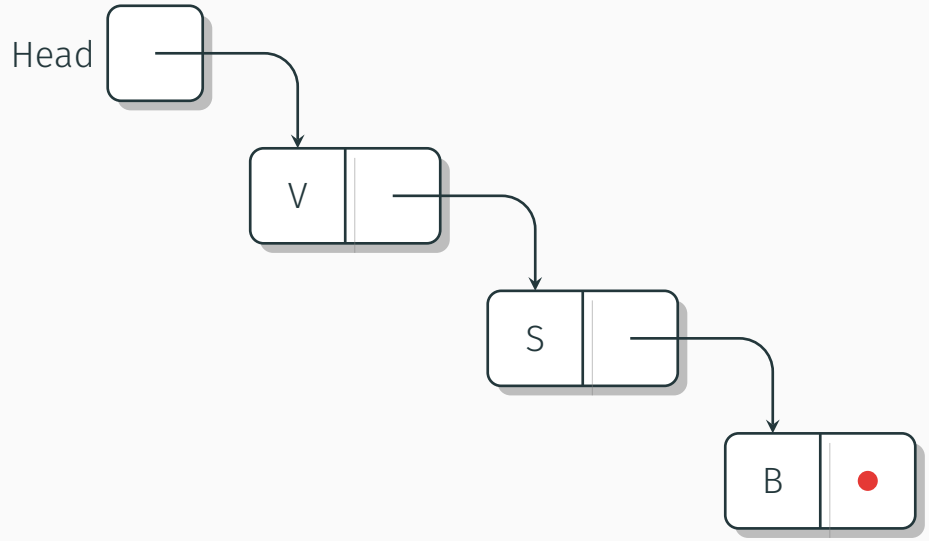
- Finite sequence of n values stored in a contiguous memory block
- Access via index with constant time complexity
- Index:
  - Non-negative integer
  - Array with n elements always has index range $0, \ldots, n - 1$

- Applications:
  - Direct use – vectors, buffers
  - Foundation for other data structures – strings, matrices etc.

Linked List

Characteristics

- Most general linear data structure
- Operations not strictly defined
- Many variants exist



Attributes

- List attributes depend on implementation
- Simplest case: single reference to first element (head)

Linked List (cont.)

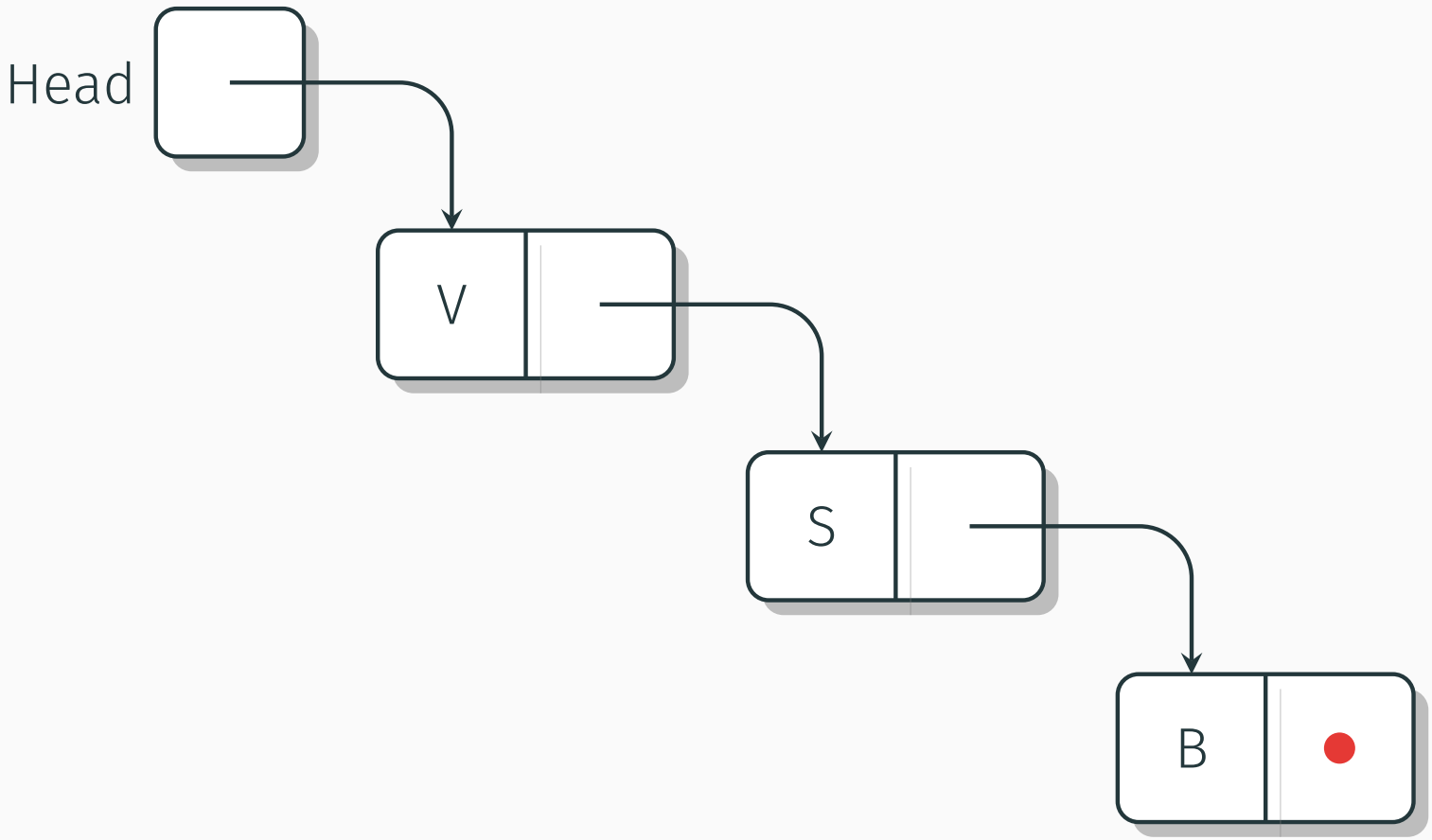
List Variants

- **Singly linked list** – nodes contain next pointer
- **Doubly linked list** – nodes contain prev/next pointers
- **Circular list** – head and tail coincide

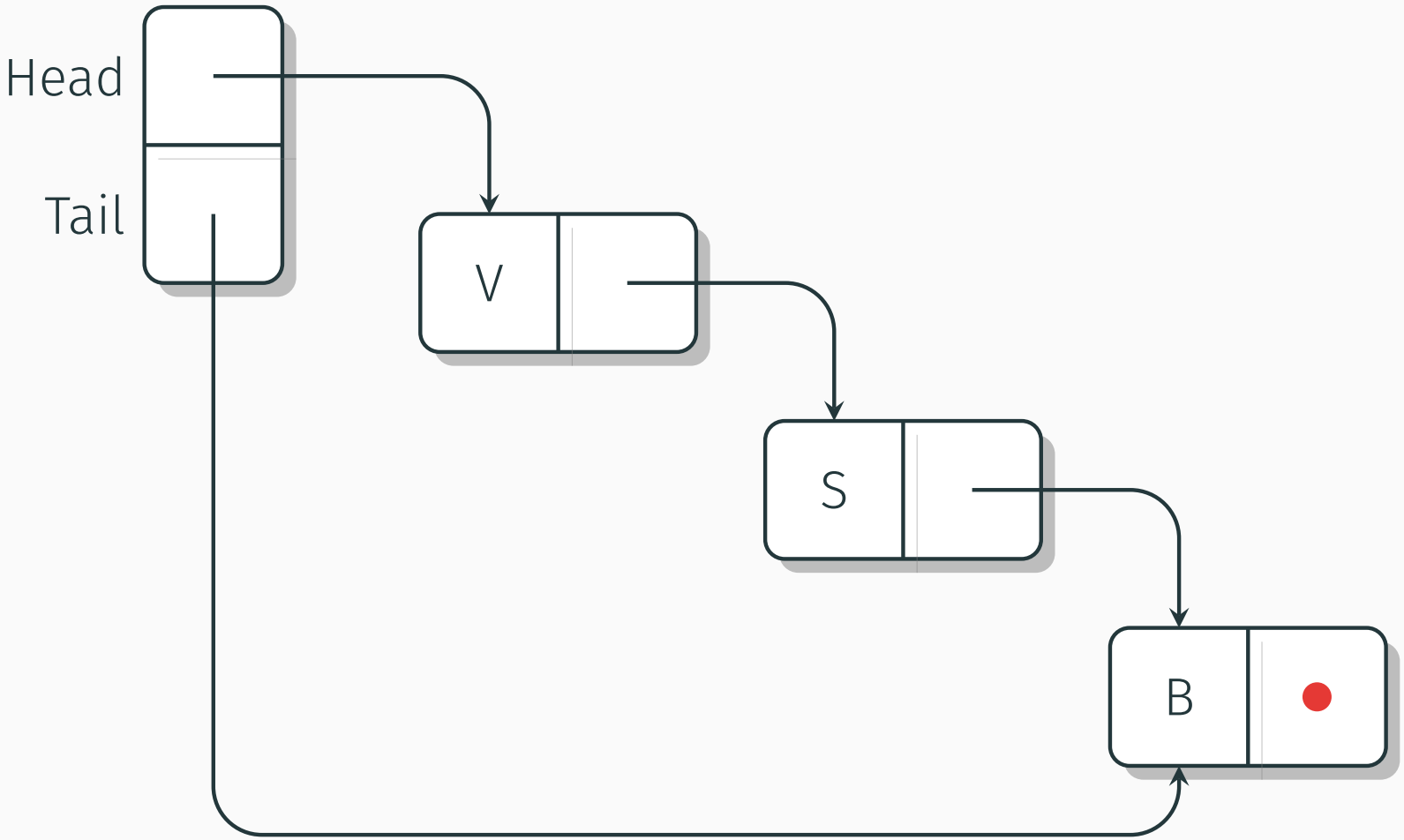
Singly Linked List

- Composed of nodes containing data + next pointer
- Sequential access only
- Direct index access requires traversal
- End marked with special nil/null pointer

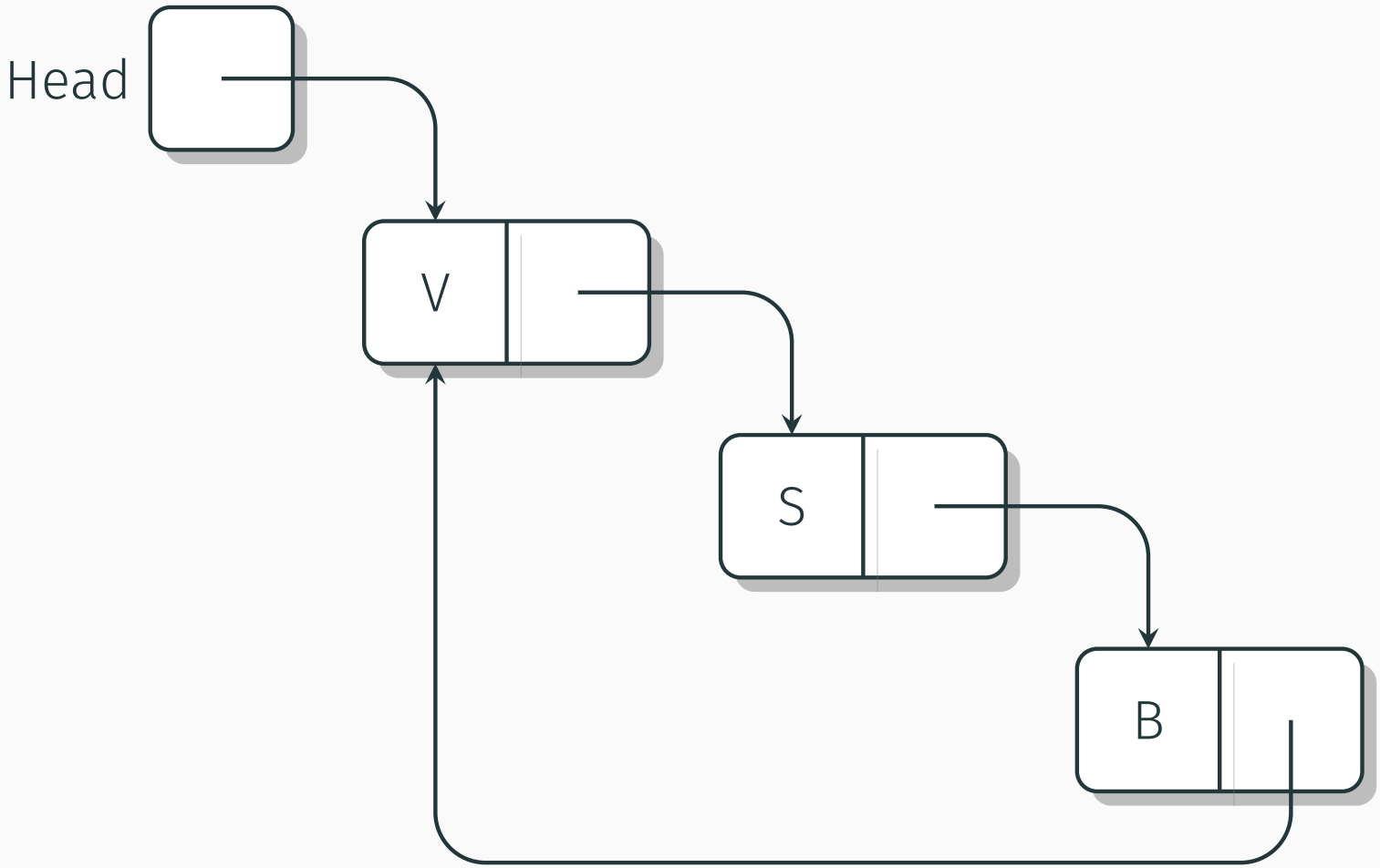
Singly Linked List (Head Only)



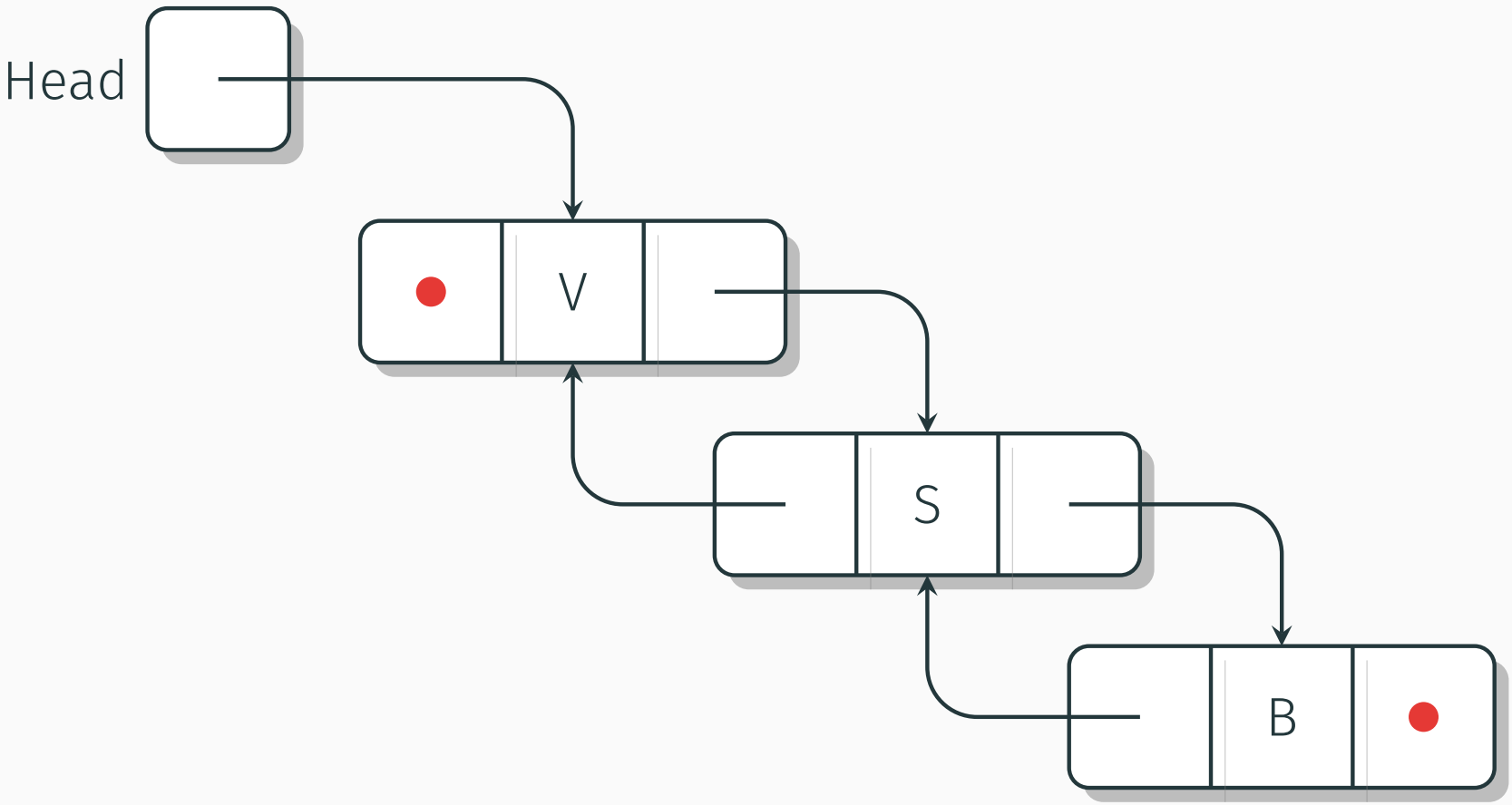
Singly Linked List (Head & Tail)



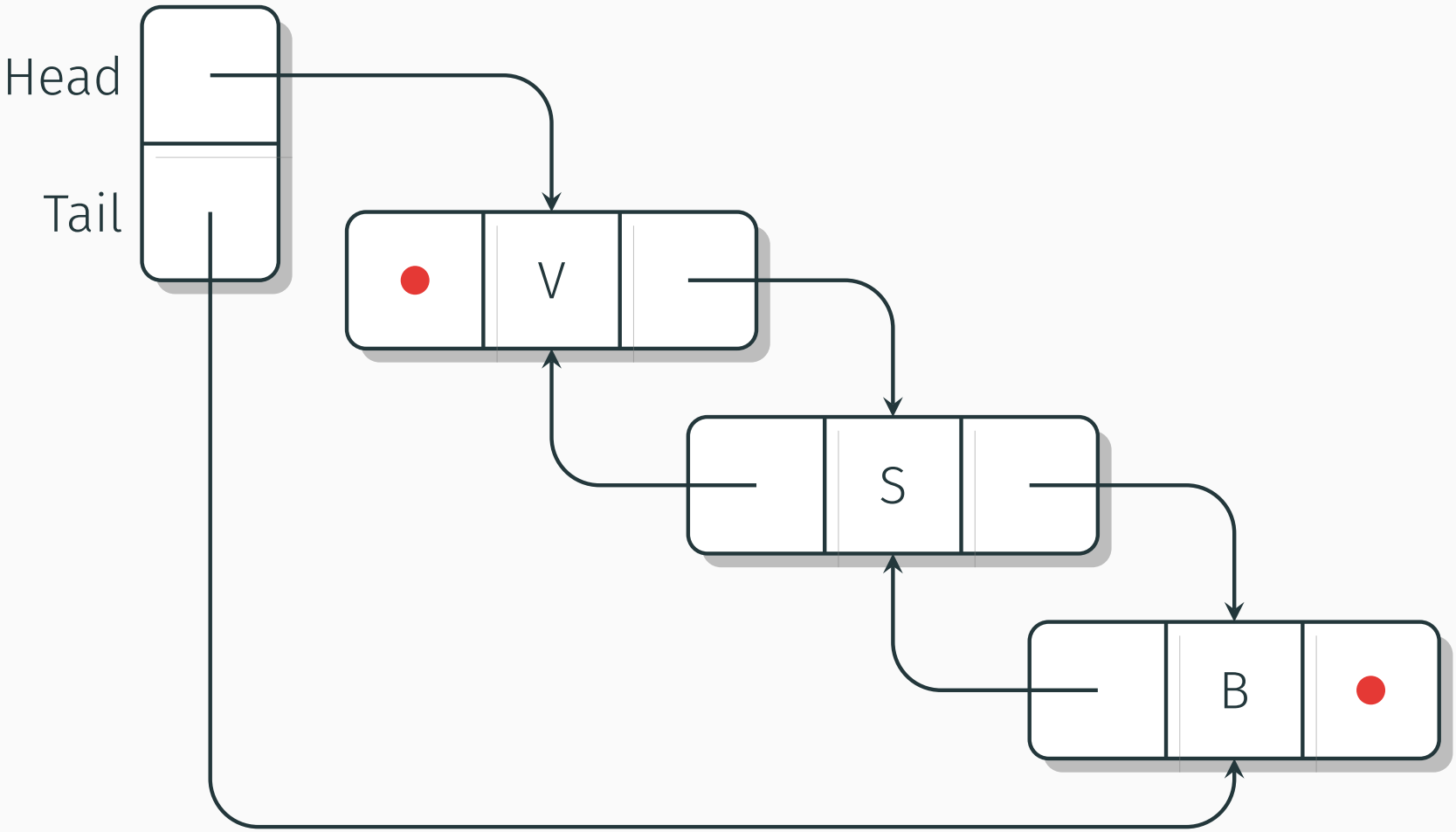
Circular Singly Linked List



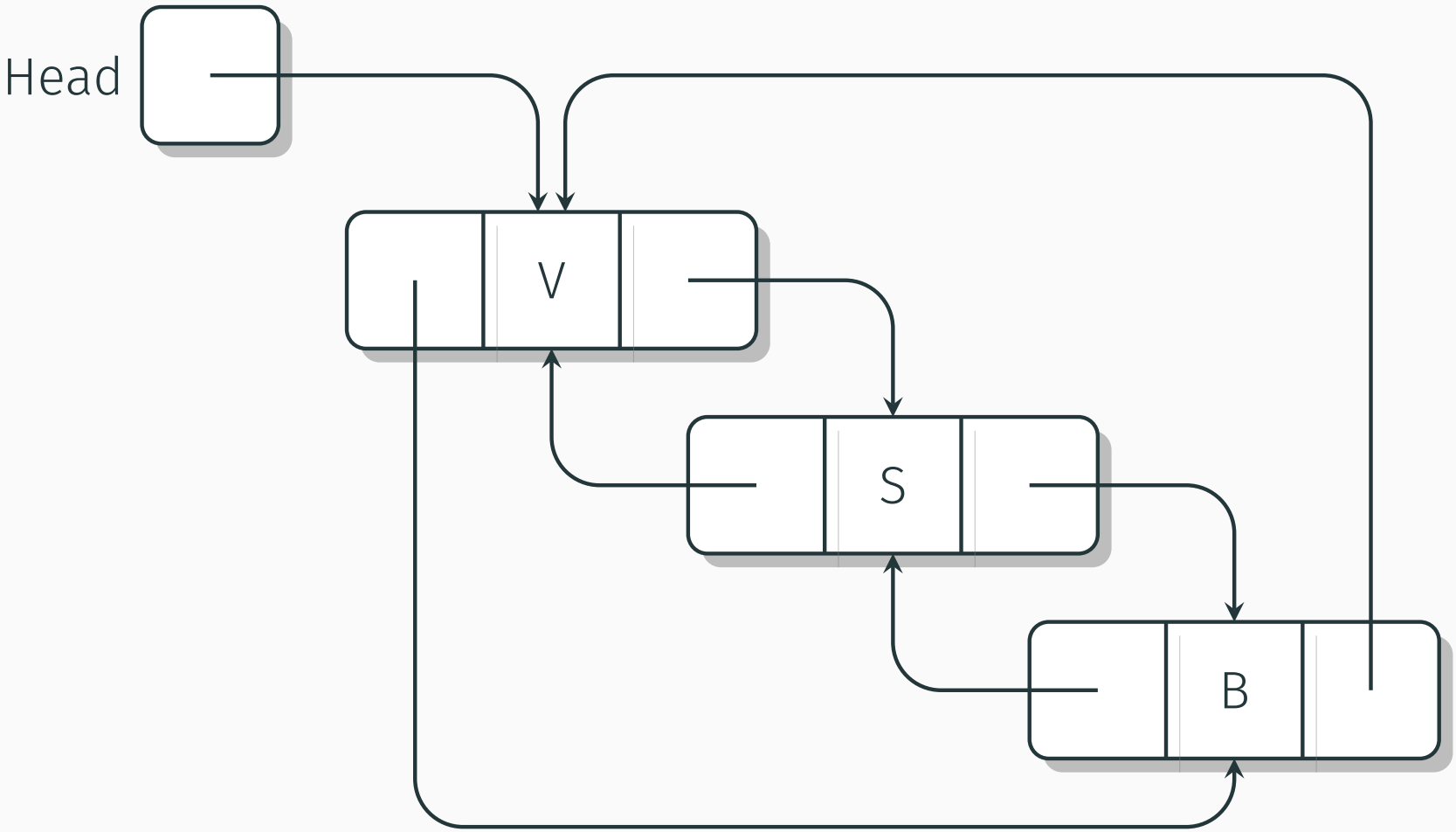
Doubly Linked List (Head Only)



Doubly Linked List (Head & Tail)



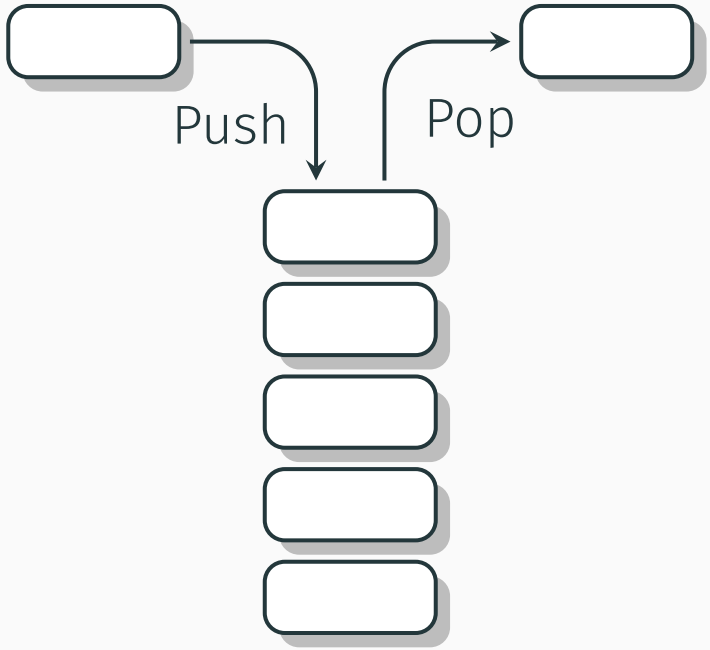
Circular Doubly Linked List



Stack

Characteristics

- LIFO (Last-In, First-Out) principle
- Most recently pushed element is first popped



Attributes

- Elements added/removed at stack top
- First inserted element – stack bottom

Stack Operations

Core Operations

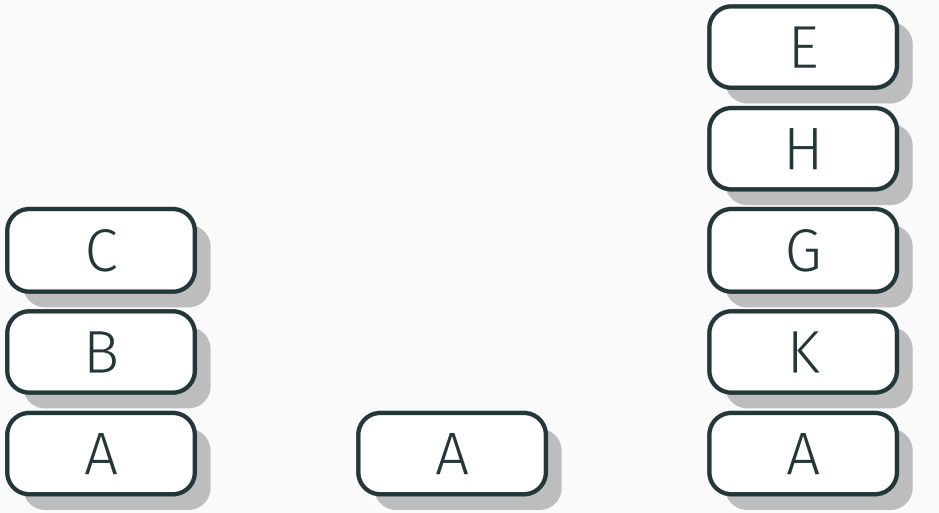
- **Push** – add element to top
- **Pop** – remove top element
- **IsEmpty** – check emptiness
- **Top** – peek top element

Additional Operations

- **Init** – initialization
- **Clear** – empty stack
- **IsFull** – check capacity (limited capacity stacks)

Properly implemented operations have **constant** time complexity $O(1)$, i.e., their time complexity does not depend on the number of elements in the stack.

Stack Visualization




```
Push(A)
Push(B)
Push(C)
```

```
Pop()
Pop()
```

```
Push(K)
Push(G)
Push(H)
Push(E)
```

Stack

Stack Error States

- **Underflow** – popping empty stack
- **Overflow** – pushing full stack

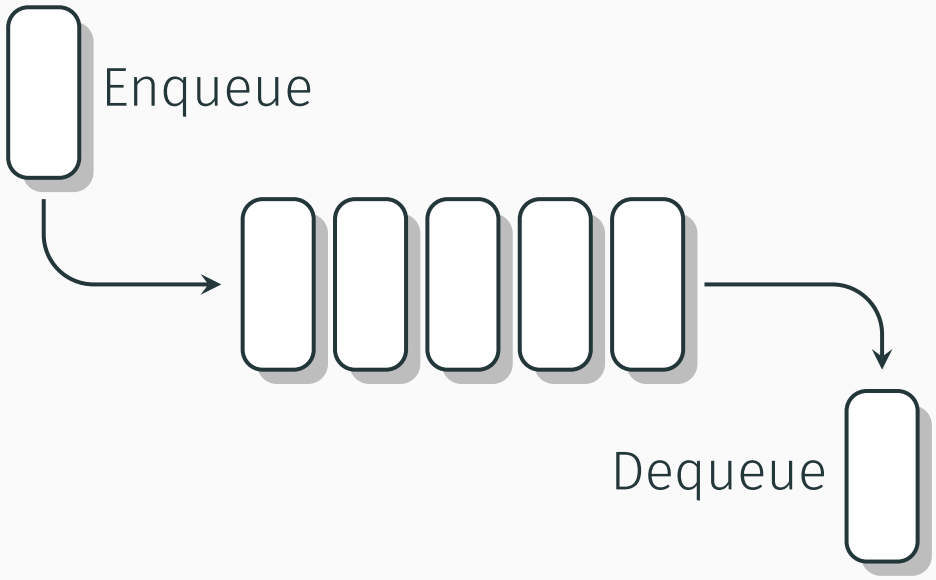
Stack Applications

- Function call management
- Expression evaluation
- Recursion elimination
- Parenthesis/XML tag validation

Queue

Characteristics

- Follows **First-In, First-Out (FIFO)** principle
- First element inserted is first removed



Attributes

- First element – **head**
- Last element – **tail**

Queue Operations

Core Operations

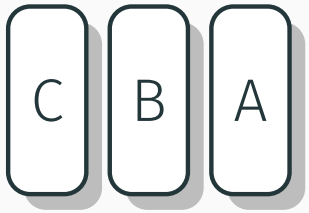
- **Enqueue** – add to tail
- **Dequeue** – remove from head
- **Peek** – inspect head element
- **IsEmpty** – check emptiness

Additional Operations

- **Init** – initialize
- **Clear** – empty queue
- IsFull – check capacity (limited capacity queues)

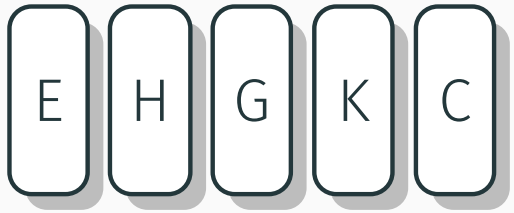
Properly implemented operations have **constant** time complexity $O(1)$, i.e., their time complexity does not depend on the number of elements in the queue.

Queue Visualization



Enqueue(A)
Enqueue(B)
Enqueue(C)

Dequeue()
Dequeue()



Enqueue(K)
Enqueue(G)
Enqueue(H)
Enqueue(E)

Queue

Queue Error States

- **Underflow** – dequeuing empty queue
- **Overflow** – enqueuing full queue, if queue capacity is limited

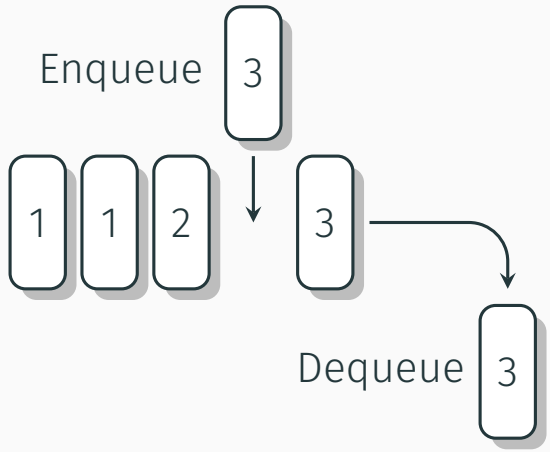
Queue Applications

- Print job scheduling
- OS process scheduling
- Server request handling

Priority Queue

Characteristics

- solving the task “Remove the largest element from the set and process it.”
- unlike a regular queue, elements are also associated with a priority,
- for elements with the same priority, FIFO applies,
- an element with higher priority overtakes those with lower priority and leaves the queue earlier.



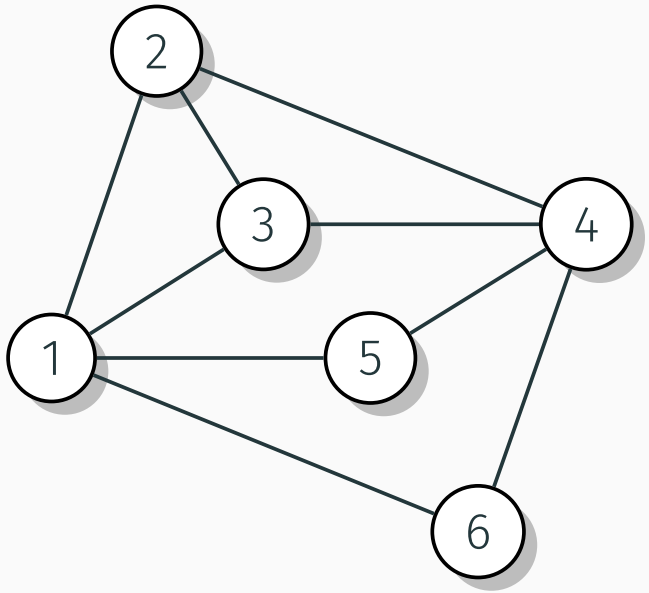
Implementations

- using an array or a sorted array,
- more efficiently using a data structure called a **heap**.

Undirected Graph

Definition

An **undirected graph** is a pair $G = (V, E)$ where V is a finite non-empty set of **vertices**, and E is a set of one-element or two-element subsets of V. Elements of set E are called **edges** of the graph.



Example

$G = (V, E)$

$V = \{1, 2, 3, 4, 5, 6\}$

$E = \{\{1, 2\}, \{1, 3\}, \{1, 5\}, \{1, 6\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{4, 5\}, \{4, 6\}\}$

Edges in an Undirected Graph

Let us have an edge $e \in E$, where $e = \{u, v\}$.

- We say that the edge e **connects** the vertices u and v.
- The vertices u and v are called the **end vertices** of the edge e.
- Furthermore, we say that the vertices u and v are **incident** (or that they **incide**) with the edge e. Similarly, we say that the edge e is incident to the vertices u and v.
- Since the edge e connects the vertices u and v, we say that they are **adjacent** (neighboring) vertices.

Definition

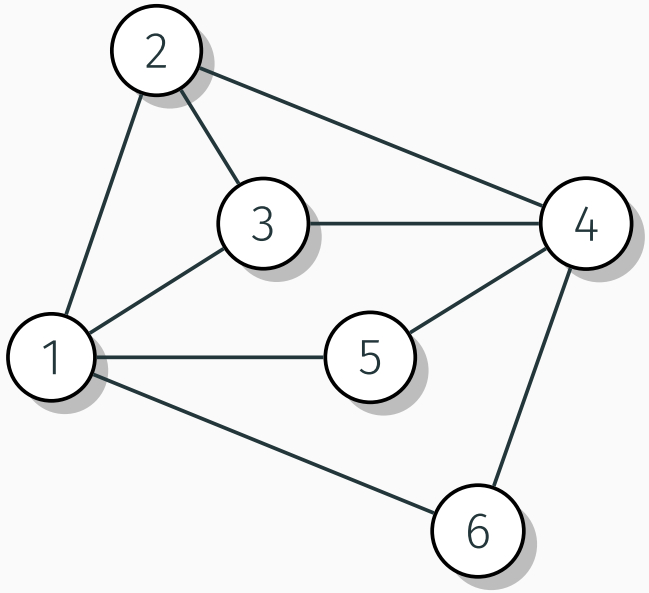
An edge that connects a vertex to itself is called a **loop**.

Undirected graph – Vertex Degree

Definition

The **degree of a vertex** in an undirected graph is the number of edges incident to the vertex, i.e., $d(v) = |\{e \in E \mid v \in e\}|$.

Example



v	$d(v)$
1	4
2	3
3	3
4	4
5	2
6	2

Undirected graph – Vertex Degree (cont.)

Theorem

The sum of the degrees of the vertices of any undirected graph $G = (V, E)$ *is equal to twice the number of its edges.*

$$\sum_{v \in V} d(v) = 2|E|$$

Proof.

Obvious (each edge is counted twice in the sum). □

Number of Edges in an Undirected Graph

Theorem

For any undirected graph $G = (V, E)$ without loops, the following holds:

$$0 \leq |E| \leq \frac{1}{2}|V|\big(|V| - 1\big)$$

Proof.

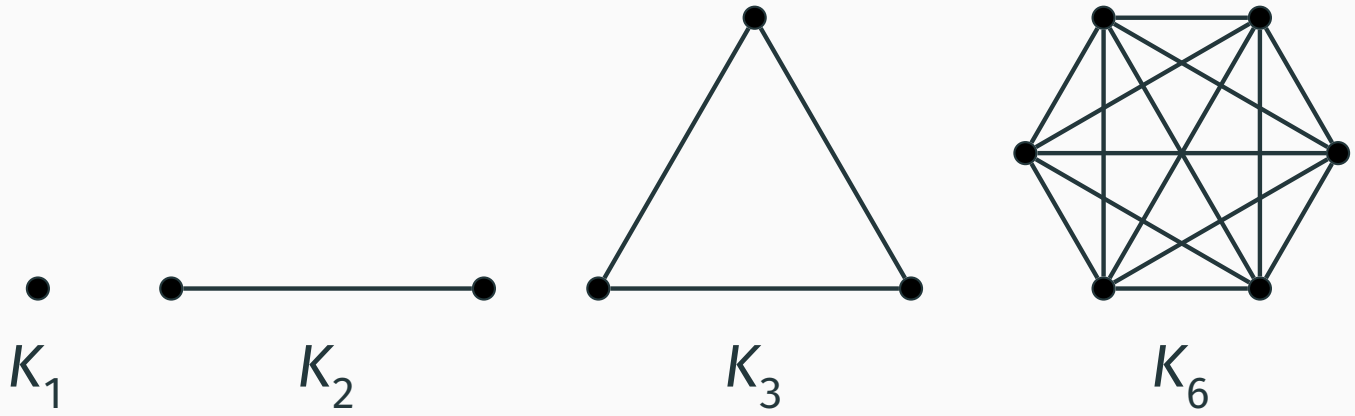
The maximum number of edges in a graph is achieved by connecting each of the $|V|$ vertices with all other vertices, which are $|V| - 1$. The product $|V|(|V| - 1)$ must be divided by two because each edge is counted twice. □

Complete Graph

Definition

An undirected graph $G = (V, E)$ in which for every pair of vertices u and v there exists an edge is called a **complete graph** and is denoted by $K_{|V|}$

Example



Dense vs. Sparse Graph

- **Dense graph** – a graph that is “almost” complete, missing only a “relatively” small number of edges to reach the maximum number.
- **Sparse graph** – a graph with a “very small” number of edges, where a “relatively” large number of edges do not exist.
- There is no precise definition; terms like “almost”, “relatively”, or “very small” are subjective.
- It always depends on the specific situation.
- When choosing a graph representation in a computer, it is necessary to consider whether the graph is dense or sparse. This subsequently affects the time complexity of the implemented algorithms.

Subgraph

Definition

Graph $H = (V_H, E_H)$ is a **subgraph** of $G = (V_G, E_G)$ if:

1. $V_H \subseteq V_G$
2. $E_H \subseteq E_G$
3. The edges of graph H have both vertices in H.

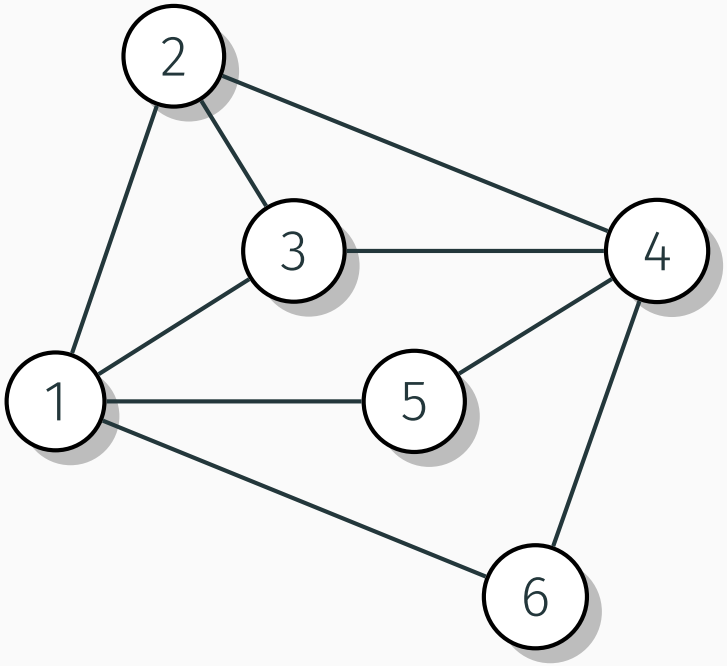
Subgraph (cont.)

Remarks

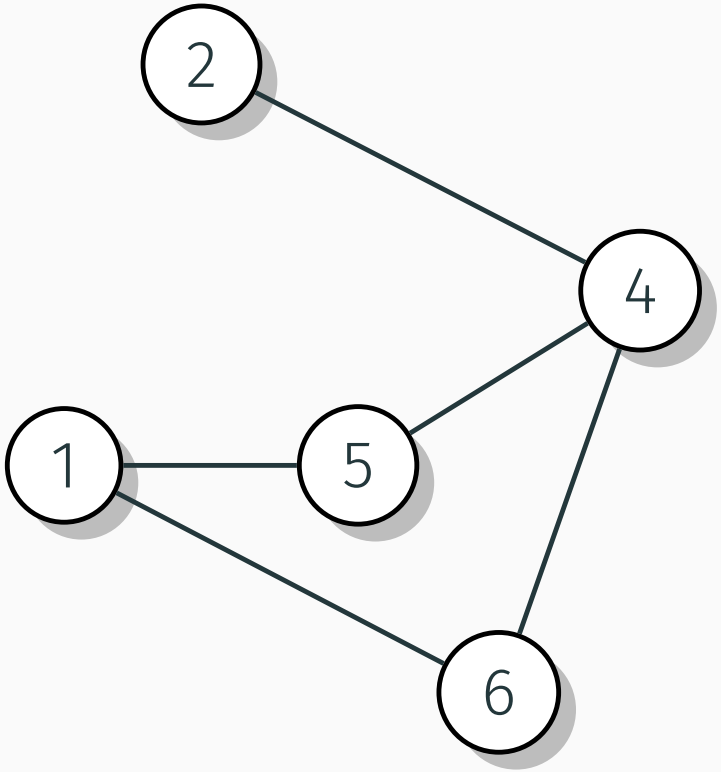
- In other words, a subgraph is obtained by deleting some vertices of the original graph, all edges incident to these vertices, and possibly some additional edges.
- The term subgraph is used in graph theory as a kind of analogy to the concept of a subset.

Subgraph (cont.)

Example



Graph G

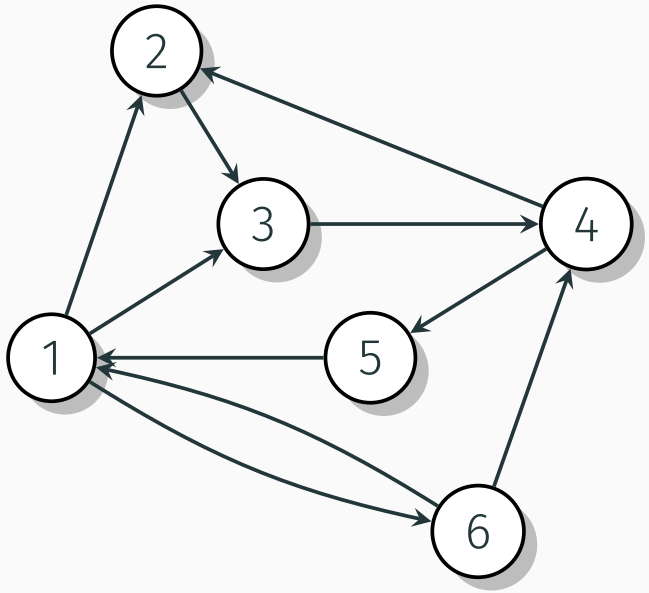


Subgraph H

Directed Graph

Definition

A directed graph is defined as a pair $G = (V, E)$, where V is a finite non-empty set of **vertices**, E is a set of ordered pairs (u, v), **edges**, from the Cartesian product $V \times V$, i.e., $(u, v) \in V \times V$.



Example

$G = (V, E)$

$V = \{1, 2, 3, 4, 5, 6\}$

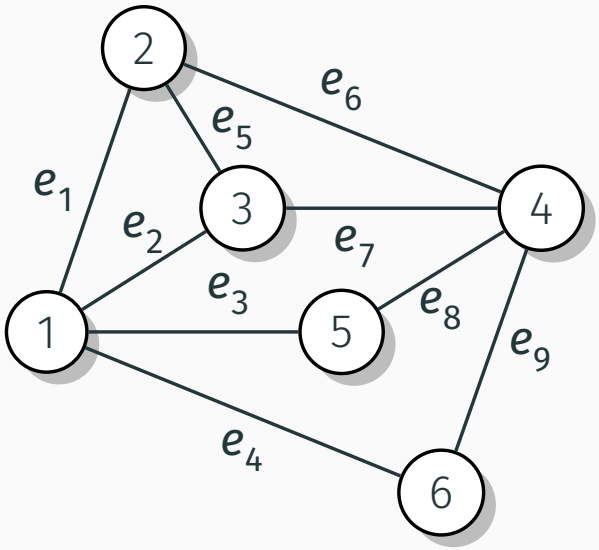
$E = \{(1, 2), (1, 3), (1, 6), (2, 3), (3, 4), (4, 2), (4, 5), (5, 1),$
$\{(6, 1), (6, 4)\}$

Methods of Representing a Graph

- Graphical form
  - simply as a picture,
  - probably the most understandable form for humans,
  - suitable for graphs with a small number of vertices,
  - practically impossible to use for computer processing.
- Matrix
- Lists of adjacent vertices

Incidence Matrix

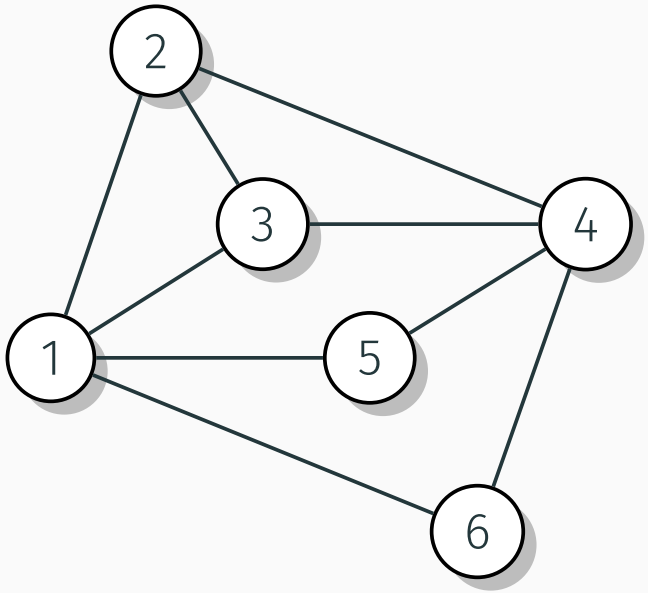
- The number of rows in the matrix corresponds to the number of vertices, and the number of columns corresponds to the number of edges.
- If a vertex is incident with an edge, there is a **1** at the given position; otherwise, there is a **0**.



$$
\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array}
\begin{array}{c}
\begin{array}{ccccccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 & e_9 \end{array} \\
\begin{pmatrix}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
\end{array}
$$

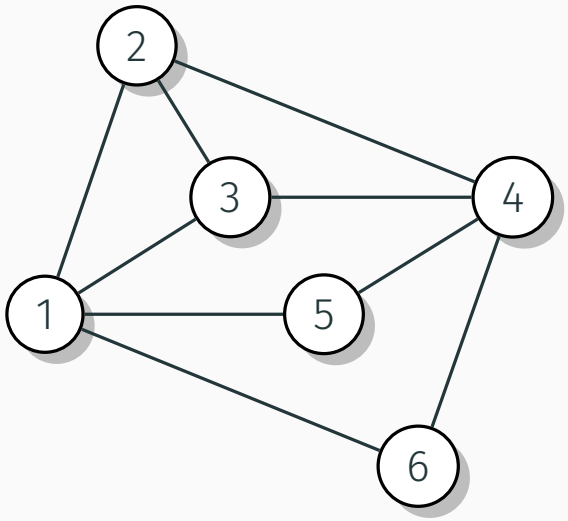
Adjacency Matrix

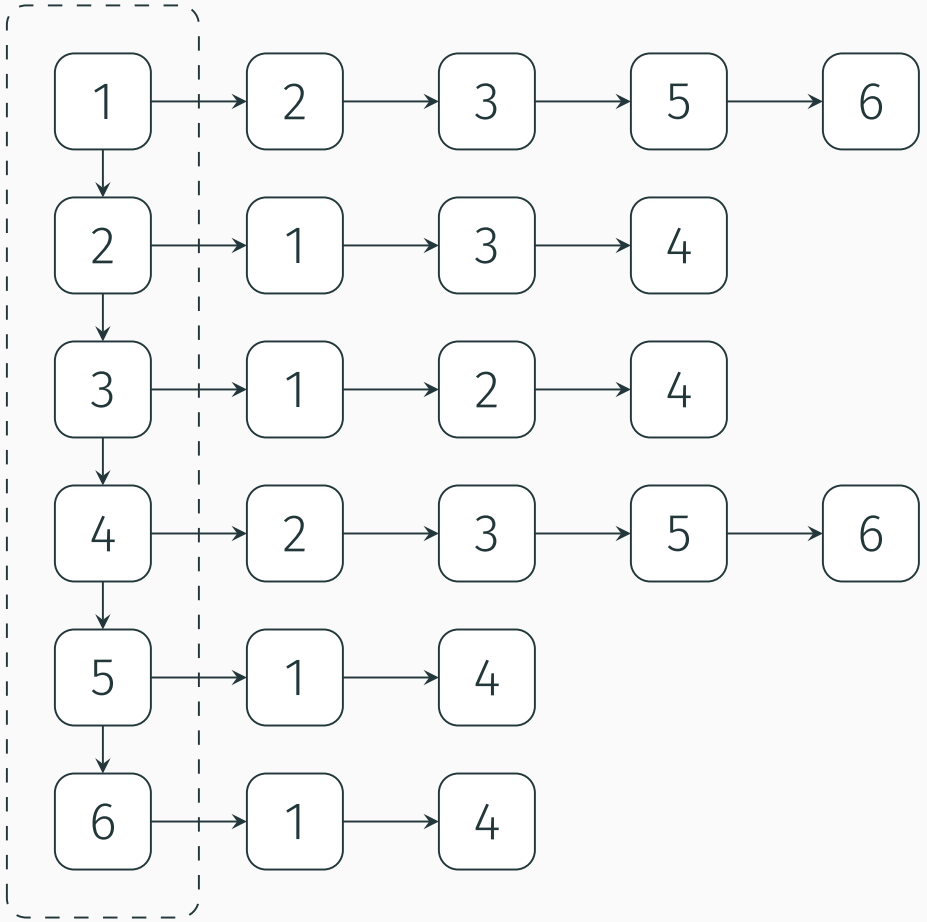
- Square matrix where the number of rows and columns corresponds to the number of vertices.
- Contains **1** if vertices are adjacent, **0** otherwise.



	1	2	3	4	5	6
1	0	1	1	0	1	1
2	1	0	1	1	0	0
3	1	1	0	1	0	0
4	0	1	1	0	1	1
5	1	0	0	1	0	0
6	1	0	0	1	0	0

Lists of Neighboring Vertices





Dense vs. Sparse Graph and their representations

Lists of Neighboring Vertices

- Pointers in lists take up additional memory.
- Suitable for sparse graphs.
- More convenient modifications of the graph structure (insertion or deletion of a vertex, as well as an edge).

Matrix Representation

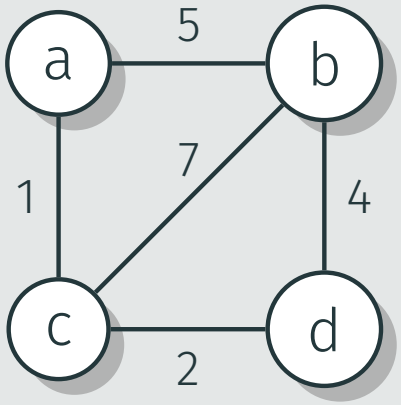
- Suitable for dense graphs.
- Vertex insertion/deletion is complex, while edge operation are easy.

Weighted Graphs

- Each edge is assigned a number referred to as the **weight** or **cost** of the edge.
- Real-world motivation – length of a path, capacity of a data link, etc.
- Weighted graphs can be directed or undirected.
- Representation:
  - adjacency matrix – the value in the matrix indicates the weight of the edge or a special value for a non-existent edge, e.g. ∞
  - adjacency list – the weight of a specific edge is also stored in the list of neighbors.

Weighted Graphs – Example

Weighted Graph



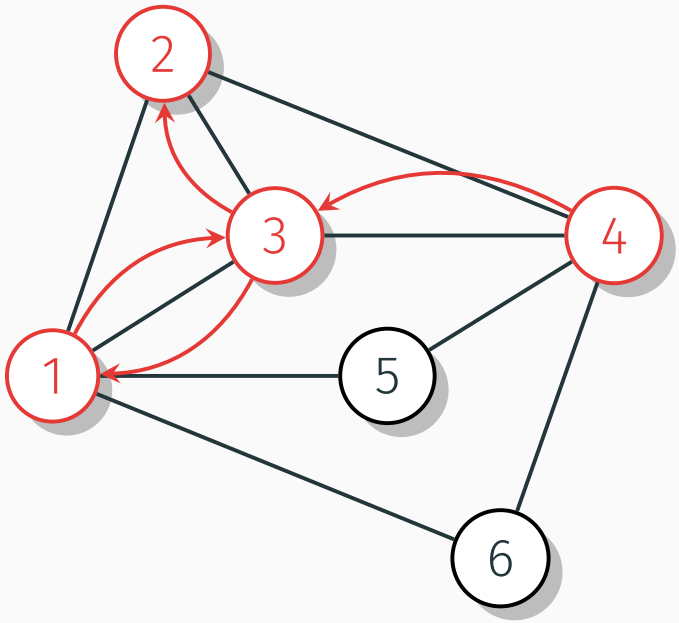
Adjacency Matrix

$$\begin{pmatrix} \infty & 5 & 1 & \infty \\ 5 & \infty & 7 & 4 \\ 1 & 7 & \infty & 2 \\ \infty & 4 & 2 & \infty \end{pmatrix}$$

Trail

Definition

A sequence of consecutive vertices and edges $v_1, e_1, v_2, \ldots, v_n, e_n, v_{n+1}$, where $e_i = \{v_i, v_{i+1}\}$ for $1 \leq i \leq n$, is called an (undirected) **trail**.



Trail

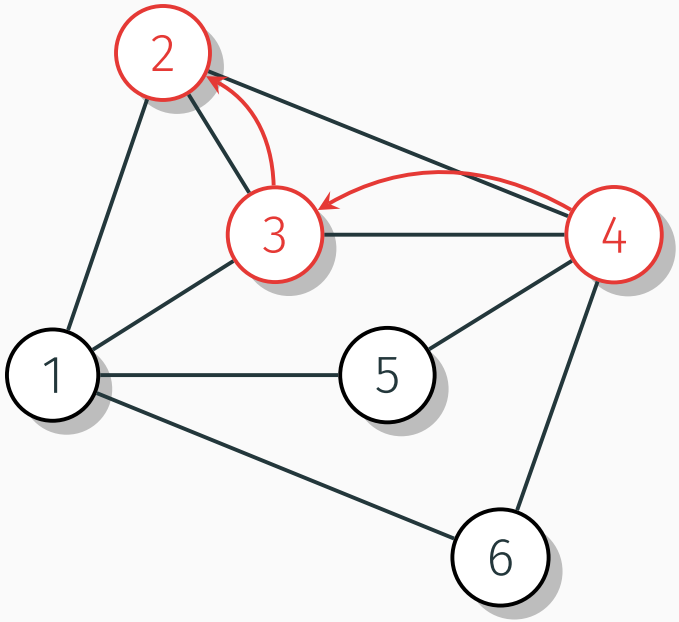
4 {4, 3} 3 {3, 1} 1 {1, 3} 3 {3, 2} 2

In oriented graphs these are called oriented trails.

Path

Definition

A trail in which no vertex is repeated is called a **path**. That is, $v_i \neq v_j, \forall\ 1 \leq i \leq j \leq n$. The number n is then called the **length of the path**.



Path

4 3 2

From the fact that vertices do not repeat in a path, it follows that edges do not repeat either. Therefore, every path is also a trail.

Graph Connectivity

Definition

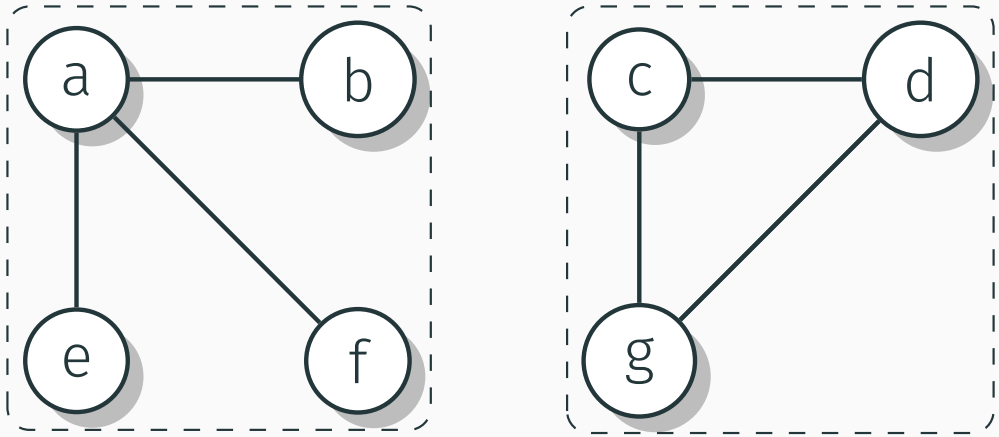
A graph is called **connected** if there exists a path between every pair of vertices.

A disconnected graph consists of several connected parts, called connected components.

Definition

A **connected component of a graph** is the maximal connected subgraph of the given graph.

Graph Connectivity (cont.)



Theorem

Let $G = (V, E)$ be a connected graph. Then it holds that $|E| \geq |V| - 1$.

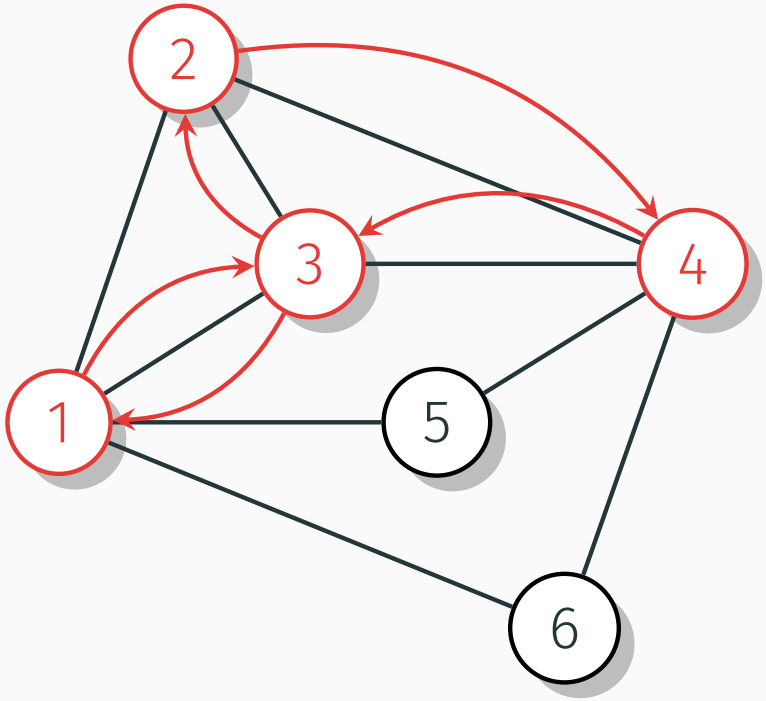
Proof.

Obvious. □

Closed trail

Definition

A trail that has at least one edge and whose starting and ending vertices coincide is called a **closed trail**.



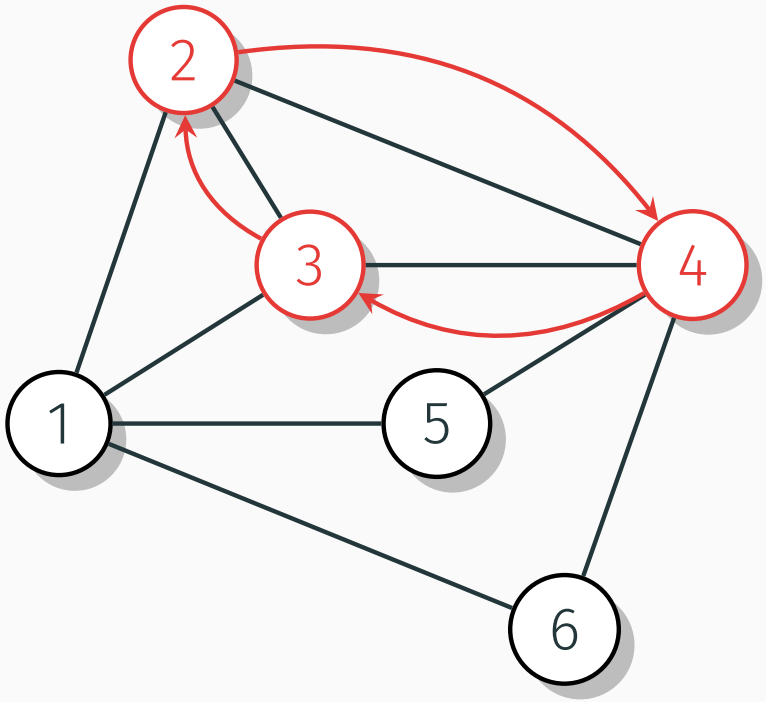
Closed trail

4 {4, 3} 3 {3, 1} 1 {1, 3} 3 {3, 2} 2 {2, 4} 4

Cycle

Definition

A **closed path** is a closed trail in which neither vertices nor edges are repeated. A closed path is also called a **cycle**.



Cycle

4 3 2

In the definition of a cycle, we had to prohibit not only the repetition of vertices but also the repetition of edges to ensure that the sequence v_1, e_1, v_2, e_1, v_1 cannot be considered a cycle.

Acyclicity of a Graph

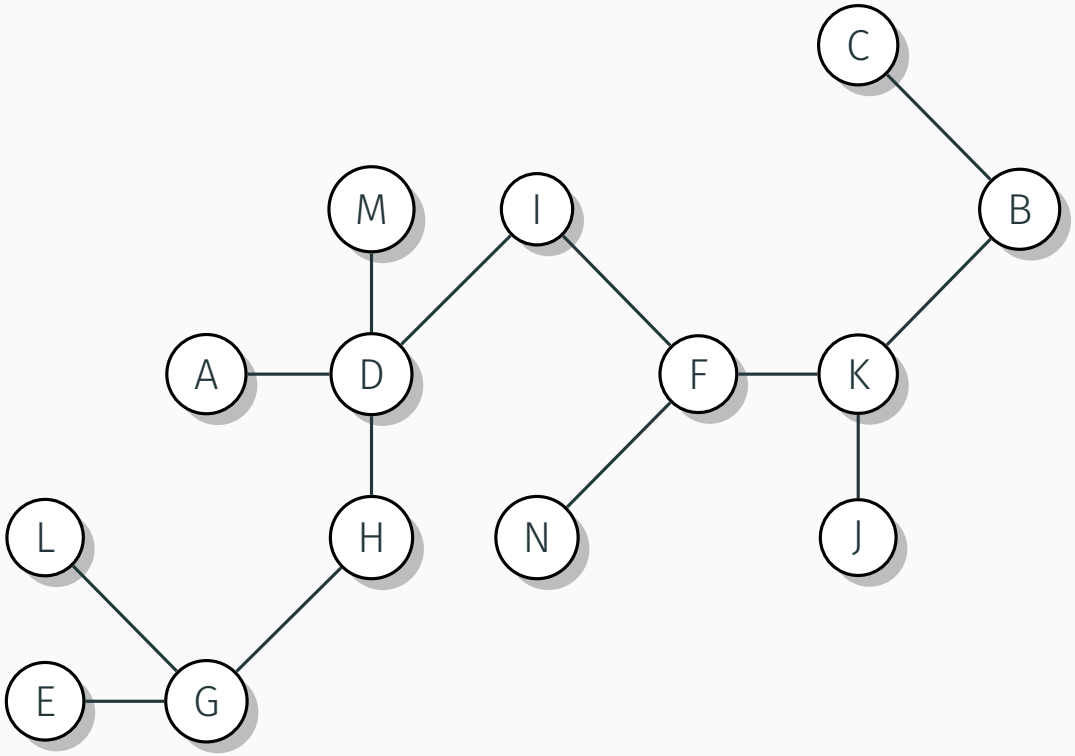
Definition

A graph is called **acyclic** if it does not contain a cycle.

Free Tree

Definition

A connected, acyclic, undirected graph is called a **free tree**.



Remark

An empty graph can be considered a tree, known as an empty tree.

Free Tree (cont.)

Terminology

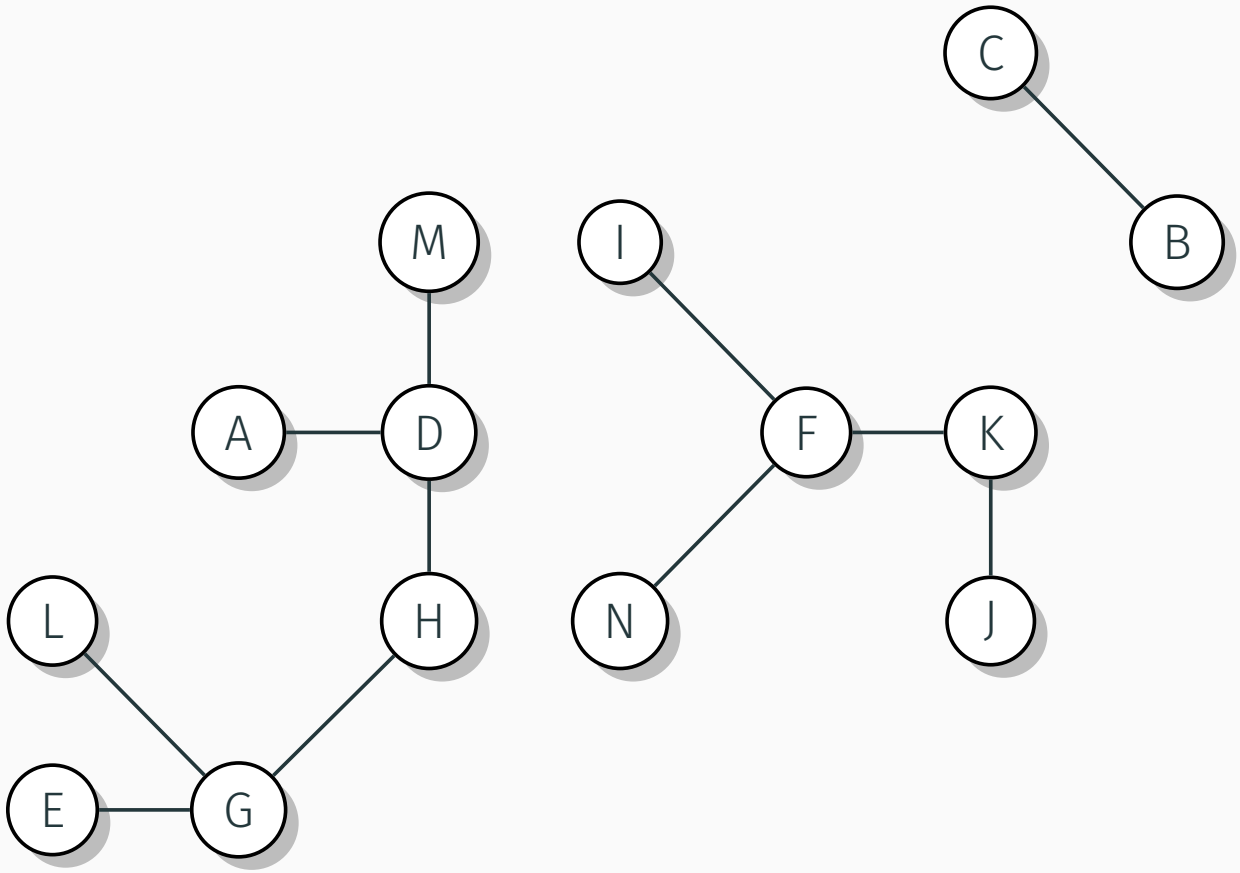
- In graph theory, the objects connected by edges are usually called vertices.
- When discussing trees, the term **node** can also be used for a vertex.
- The terms vertex and node are equivalent; it is more a matter of convention.

Forest

Definition

An acyclic graph that is not connected is called a **forest**.

Each connected component of a forest is a free tree.



Free tree properties

Theorem

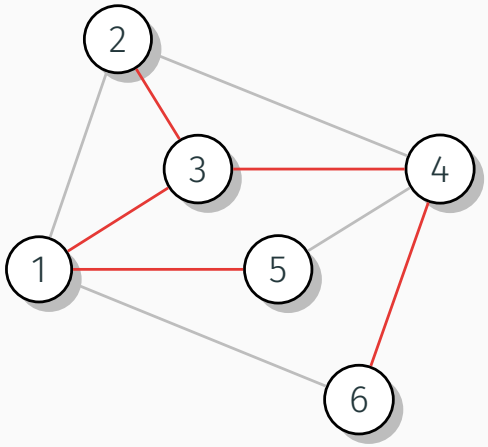
Let $G = (V, E)$ be an undirected graph, then the following statements are equivalent

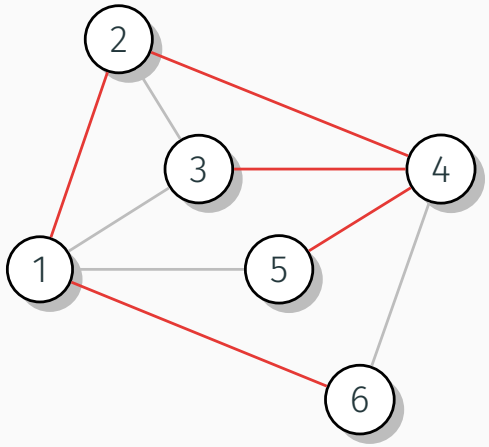
1. *G is a free tree.*
2. *Every two vertices in G are connected by exactly one path.*
3. *G is connected, but if we remove any edge, we obtain a disconnected graph.*
4. *G is connected, and $|E| = |V| - 1$.*
5. *G is acyclic, and $|E| = |V| - 1$.*
6. *G is acyclic. Adding a single edge to the set of edges E will result in a graph containing a cycle.*

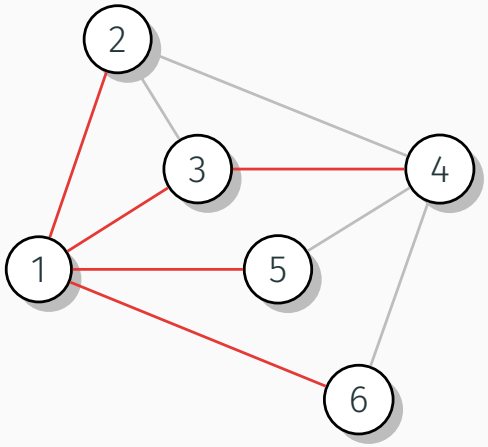
Spanning Tree

Definition

A spanning tree of a connected graph G is called a subgraph of G on the set of all its vertices that is a tree.







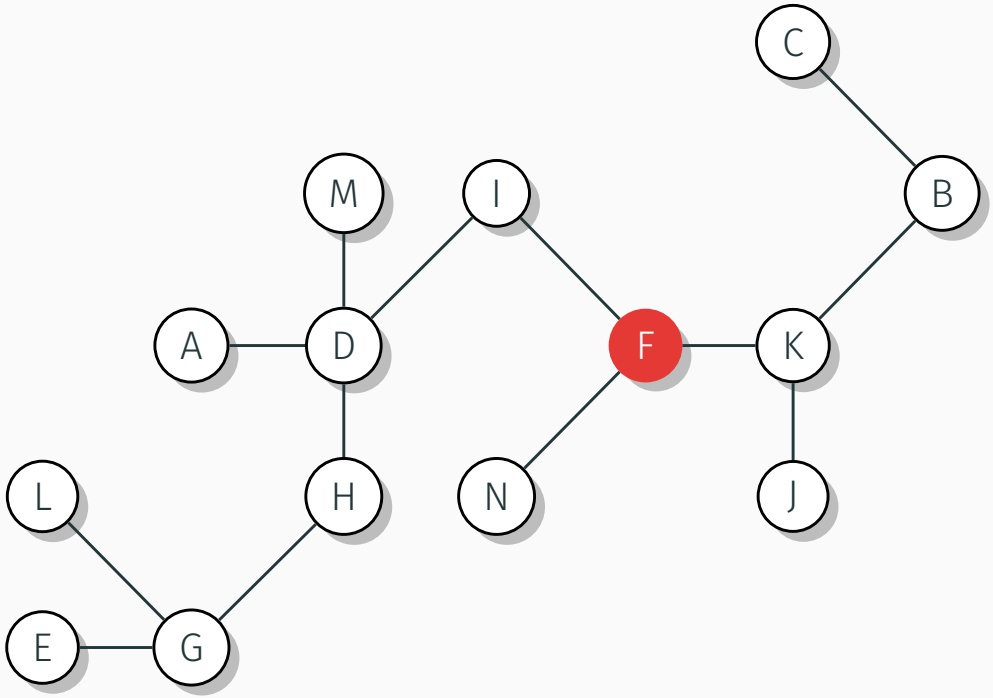
Remarks

- A spanning tree must contain all the vertices of the original graph G.
- A graph can have multiple spanning trees.

Rooted Tree

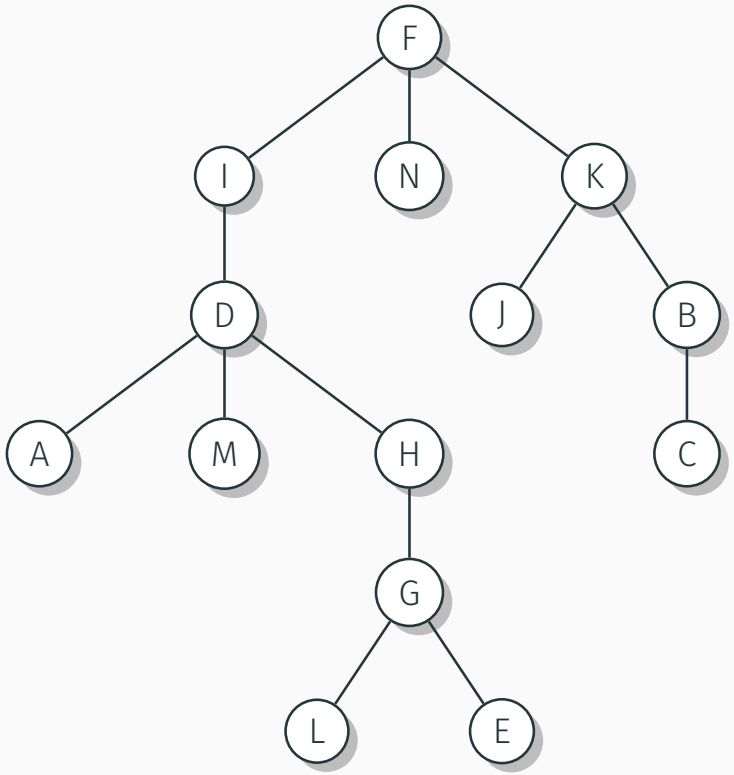
Definition

A free tree that contains one distinguished vertex is called a **rooted tree**. The distinguished vertex is called the **root** of the tree.

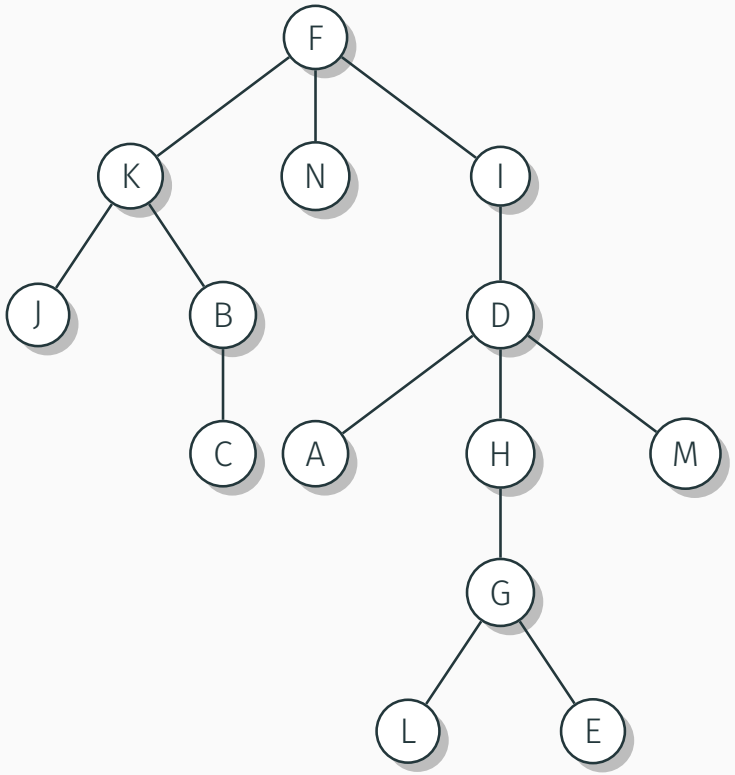


Rooted tree – a Common Visualization

Visualization 1



Visualization 2



Both visualizations are **equivalent** rooted trees! There is no "left" or "right".

Rooted Tree – Basic Concepts

Consider a vertex x in a rooted tree T with root r.

- Any vertex y on the unique path from the root r to the vertex x is called a **predecessor** of the vertex x.
- If y is a predecessor of x, then x is called a **successor** of the vertex y.
- If the last edge on the path from the root r to the vertex x is the edge (y, x), then the vertex y is called the **parent** of the vertex x, and the vertex x is a **child** of the vertex y.
- Two vertices that have the same parent are called **siblings**.
- A vertex without children is called an external vertex or a **leaf**.

Rooted Tree – Basic Concepts (cont.)

- A non-leaf vertex is called an **internal** vertex of the tree.

Remarks

- Every vertex is, of course, a predecessor and successor of itself.
- If y is a predecessor of x and at the same time $x \neq y$, then y is a proper predecessor of the vertex x, and x is a proper successor of the vertex y.
- The root of the tree is the only vertex in the tree without a parent.
- A vertex is a general concept. Every leaf and internal vertex is also a (generic) vertex. Compare: human, woman, man.

Vertex Degree

Definition

The number of children of a vertex x in a rooted tree is called the **degree of the vertex x**.

Remarks

- The method of calculating the degree of a vertex in a rooted tree differs from that in a free tree.
- In a rooted tree, the parent is not counted.
- In a free tree, the concept of a parent does not exist; there are only neighboring vertices, so all vertices are counted.

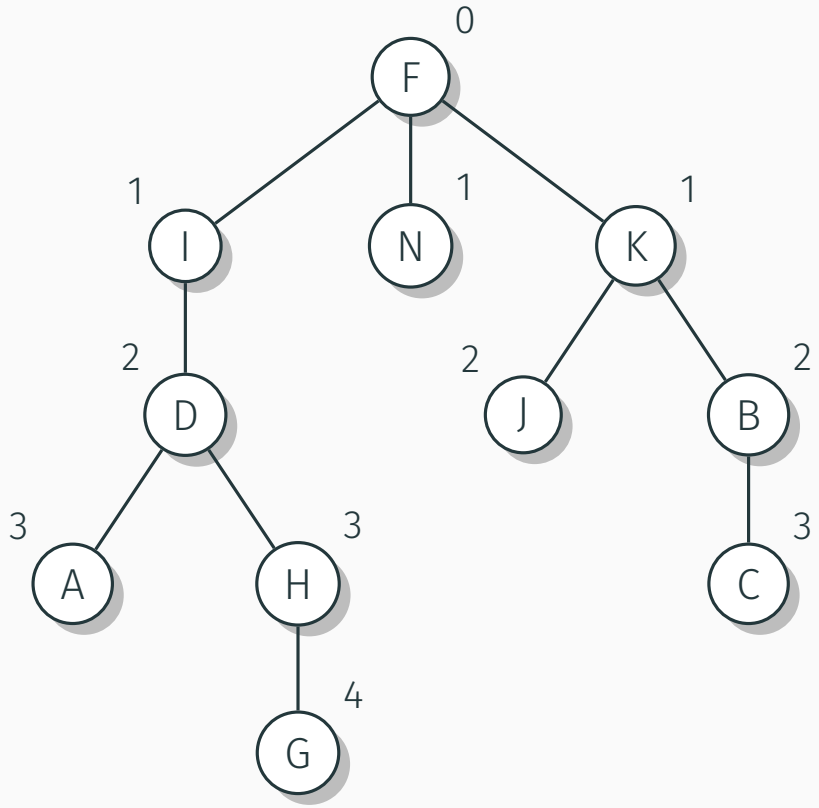
Depth of a Vertex – Height of a Tree

Definition

The length of the path from the root of the tree to a vertex x is called the **depth of the vertex x** in the tree T.

Definition

The greatest depth of any vertex is called the **height of the tree T**.



The height of the tree is 4.

Ordered Tree

Definition

A rooted tree in which the order of children is specified is called an **ordered tree**.

Remarks

- Thus, if a vertex has k children, it is possible to determine the first child, second child, up to the k-th child.
- However, if, for example, we remove the first child, the remaining children shift! The second child becomes the first, the third becomes the second, and so on. There cannot be an “empty position” among children.

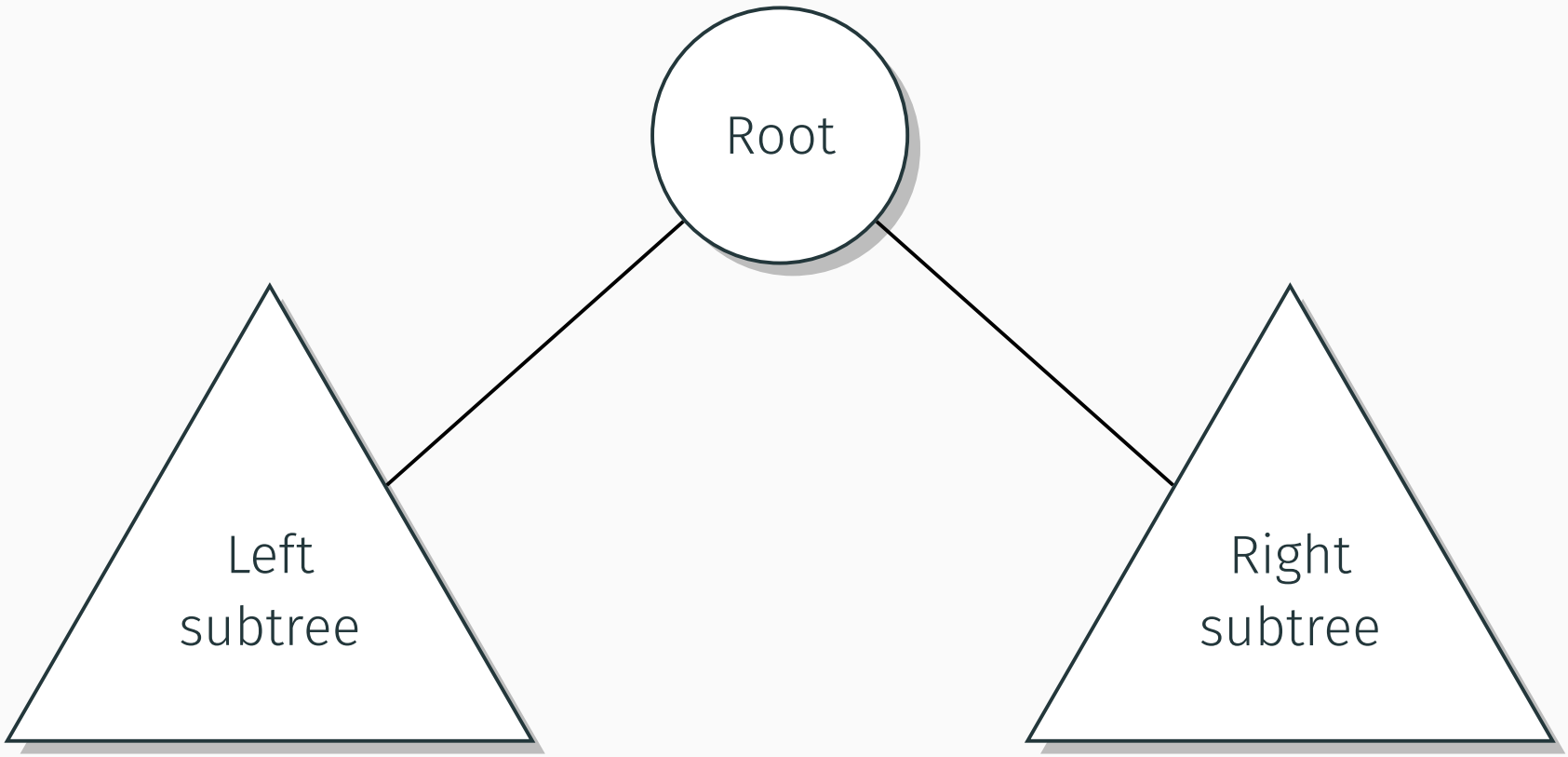
Binary Tree

Definition

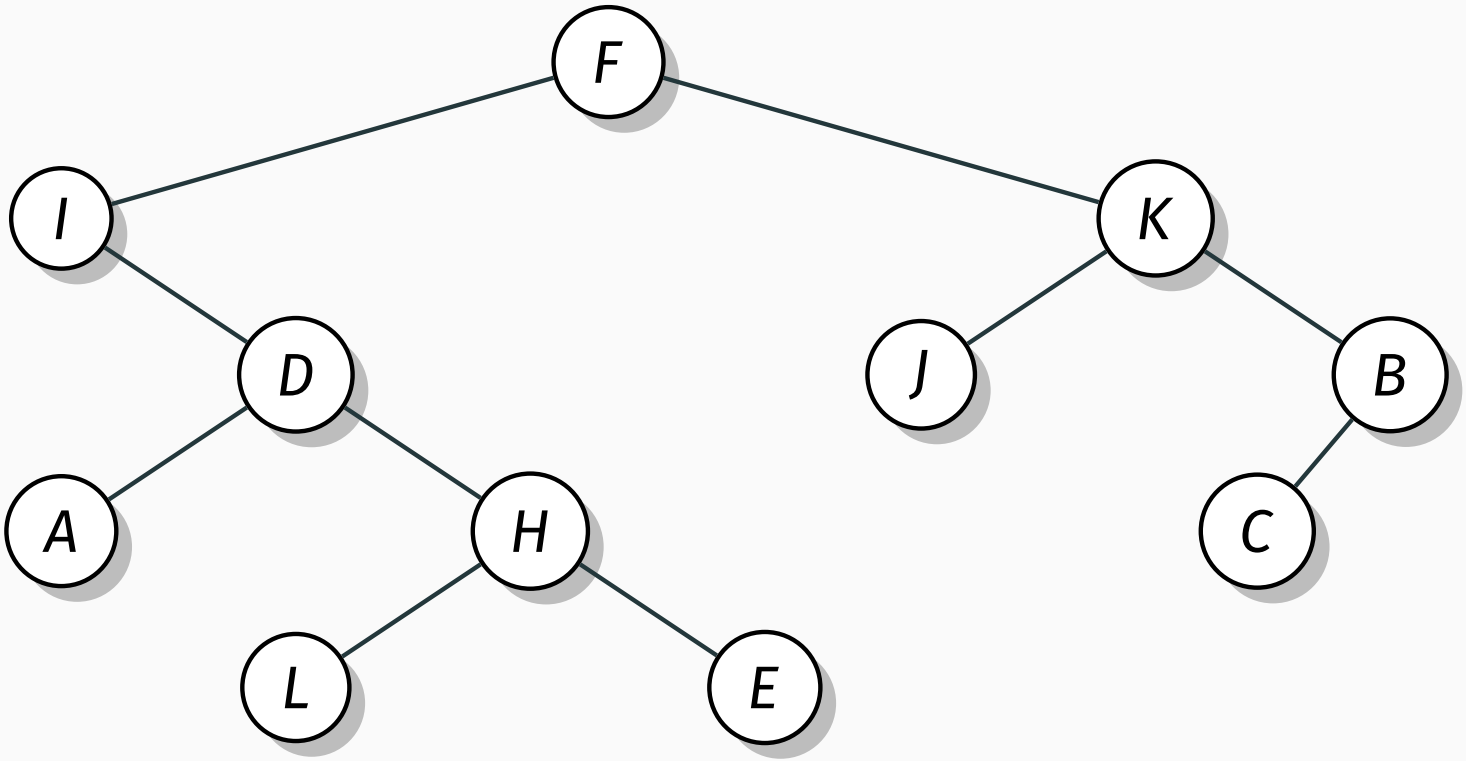
A **binary tree** is a structure defined over a finite set of nodes M, which:

- **Rule 1**
  contains no nodes, i.e., $M = \emptyset$, or
- **Rule 2**
  is composed of three disjoint sets of nodes L, R, and $\{r\}$, $L \cup R \cup \{r\} = M$:
  - the root of the tree r,
  - a binary tree over set L, called the left subtree, and
  - a binary tree over set R, called the right subtree.

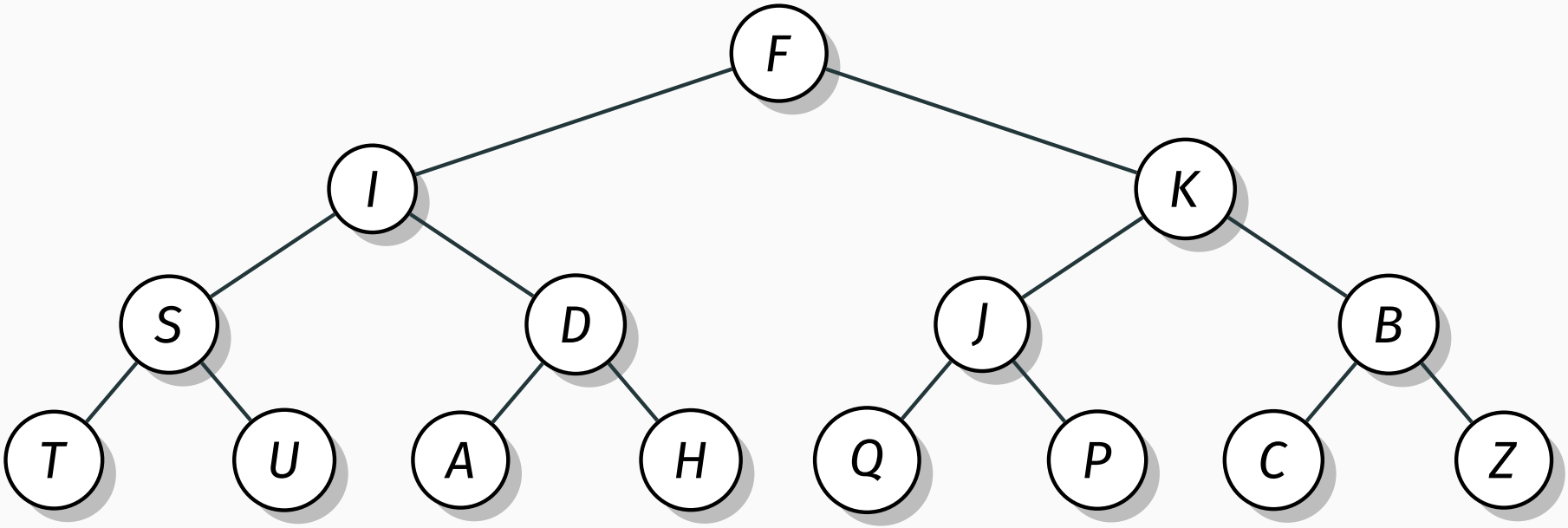
Binary Tree – Graphical Representation of Recursive Definition



Binary Tree – Example



Complete Binary Tree

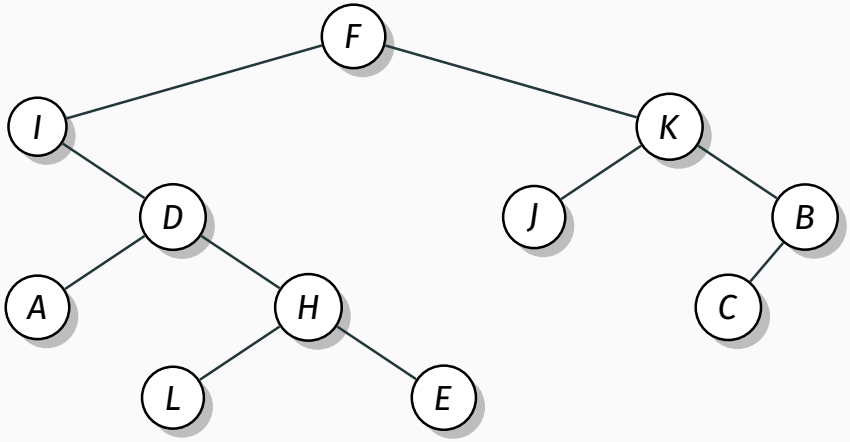


Complete Binary Tree – every internal node has exactly two children.

Binary Search Tree

How to use binary trees as a data structure? How to organize data within them?

Arbitrarily? Nonsense – it would be an unnecessarily complicated list!



The solution is to use the properties of the tree (connectivity and uniqueness of the path from node to node) and complement them with an appropriate "**navigation rule**".

Binary Search Tree – “Navigation Rule”

Let y be a node in a binary tree. Then for every node x in the left subtree of node y and every node z in the right subtree of node y, the following holds:

$$x_{key} < y_{key} < z_{key}.$$

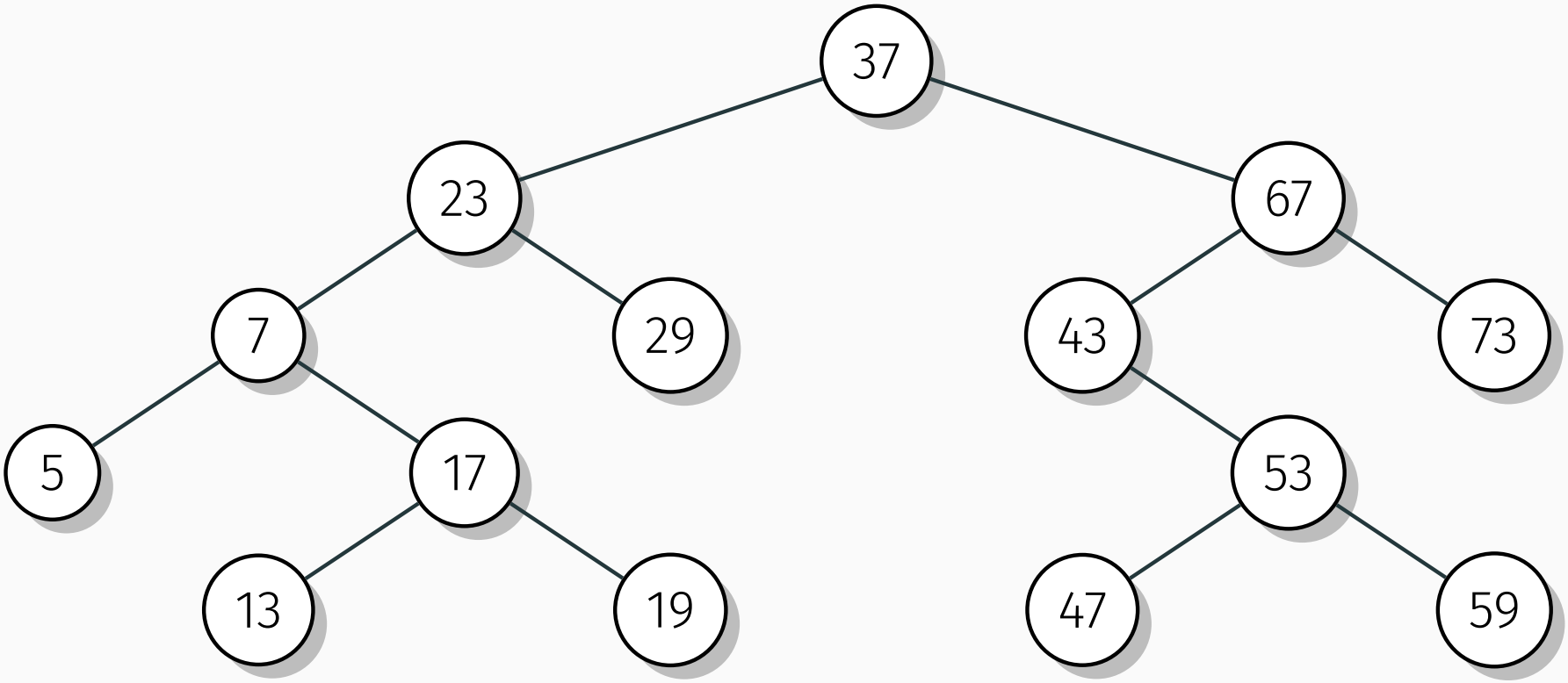
A binary tree, in which this rule applies to all its nodes, is called a **binary search tree**.

Binary Search Tree – “Navigation Rule” (cont.)

Remarks

- The navigation rule thus determines how data should be arranged in the binary search tree.
- Knowledge of data arrangement in the tree is used when searching for them.
- Algorithms for insertion and deletion from the tree are tied to the search algorithm.
- A binary search tree is therefore built from the outset with this rule in mind.

Binary Search Tree



Binary Search Tree – Searching

Searching for a value a begins at the root of the tree r. Then, the following possibilities may occur:

1. The tree with root r is empty; in this case, the tree cannot contain a node with key a, and the search ends unsuccessfully.
2. Otherwise, we compare the key a with the key of the root r. In the case that:
   - 2.1 $a = r_{key}$, the tree contains a node with key a, and the search ends successfully;
   - 2.2 $a < r_{key}$, all nodes with keys smaller than r_{key} are in the left subtree, so we continue recursively in the left subtree;
   - 2.3 $a > r_{key}$, all nodes with keys greater than r_{key} are in the right subtree, so we continue recursively in the right subtree.

Binary Search Tree

The efficiency of many algorithms that generally work with binary trees, such as searching in a binary search tree, depends on the height of the binary tree.

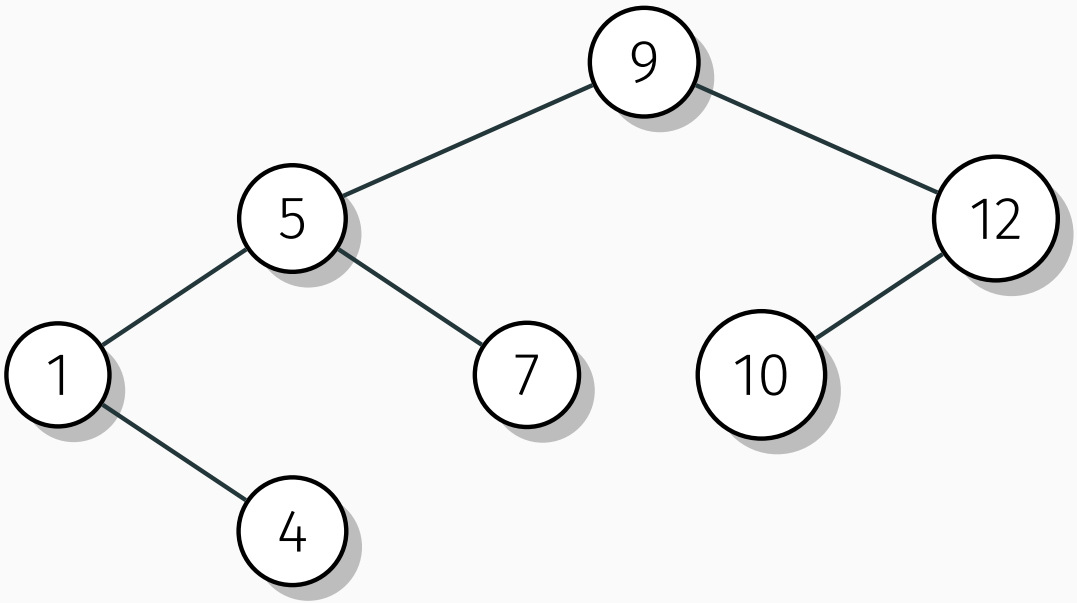
For the height h of a binary tree with n nodes, the inequality holds:

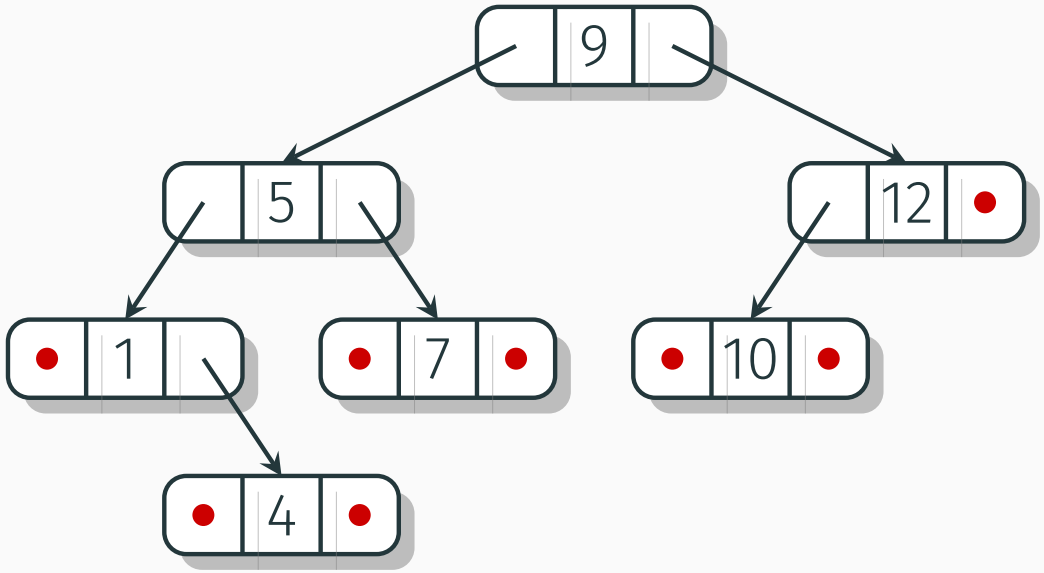
$$\lfloor \log_2 n \rfloor \leq h \leq n - 1$$

Binary Search Tree – Insertion

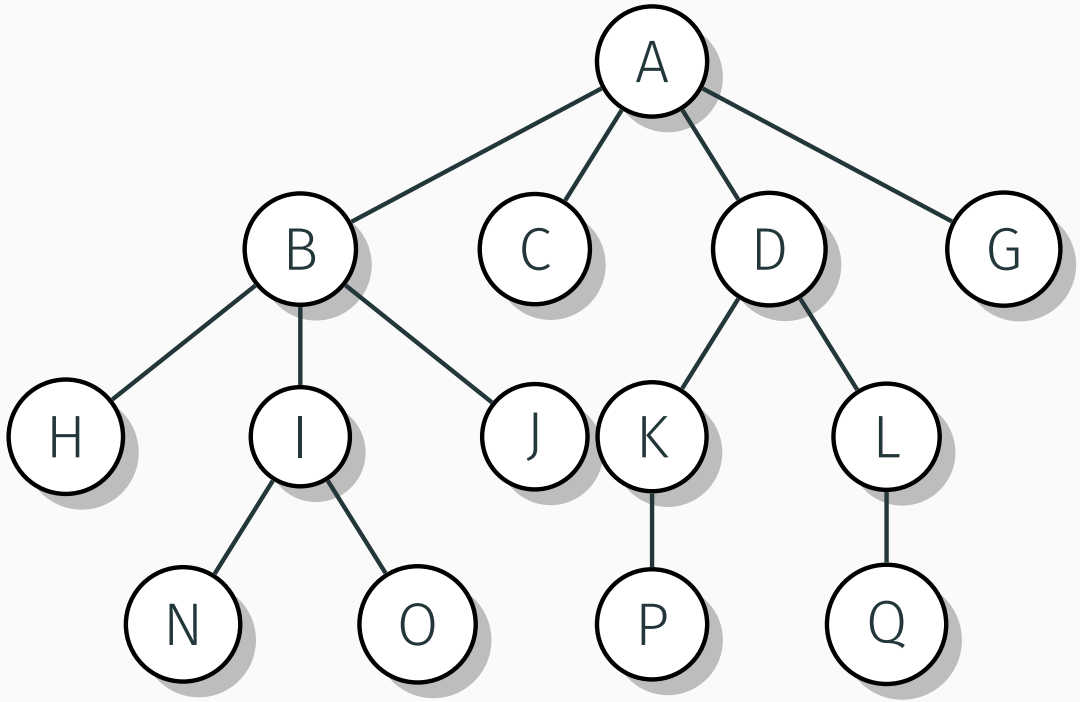
- The insertion of a key must correspond to the search algorithm.
- First, we must attempt to find the key being inserted in the tree.
- If it is not found, then the place where the search ended unsuccessfully corresponds to where this key should be in the tree.
- This follows from the uniqueness of the path between the root and any node.
- The new node is attached as a new leaf to the tree – the tree grows through its leaves.
- The question is what to do with duplicates? The solution depends on the nature of the specific problem being solved.

Binary Tree – the standard implementation



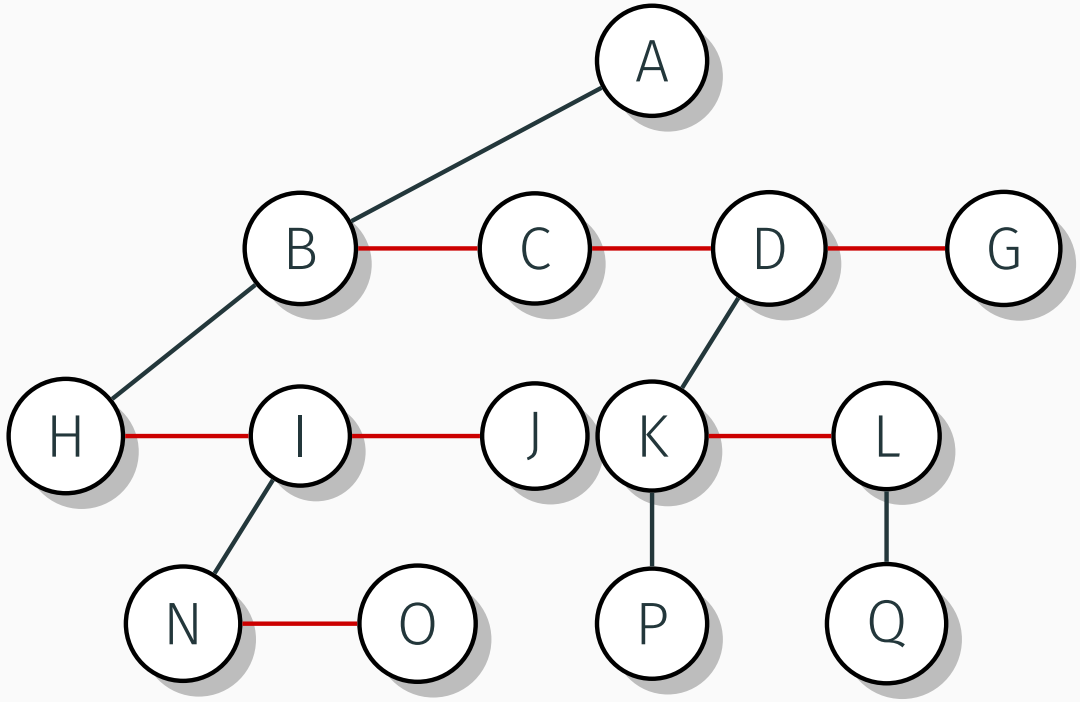


Ordered Tree Representation



- Each node can have any number of children.
- Complex representation of a node – list of children

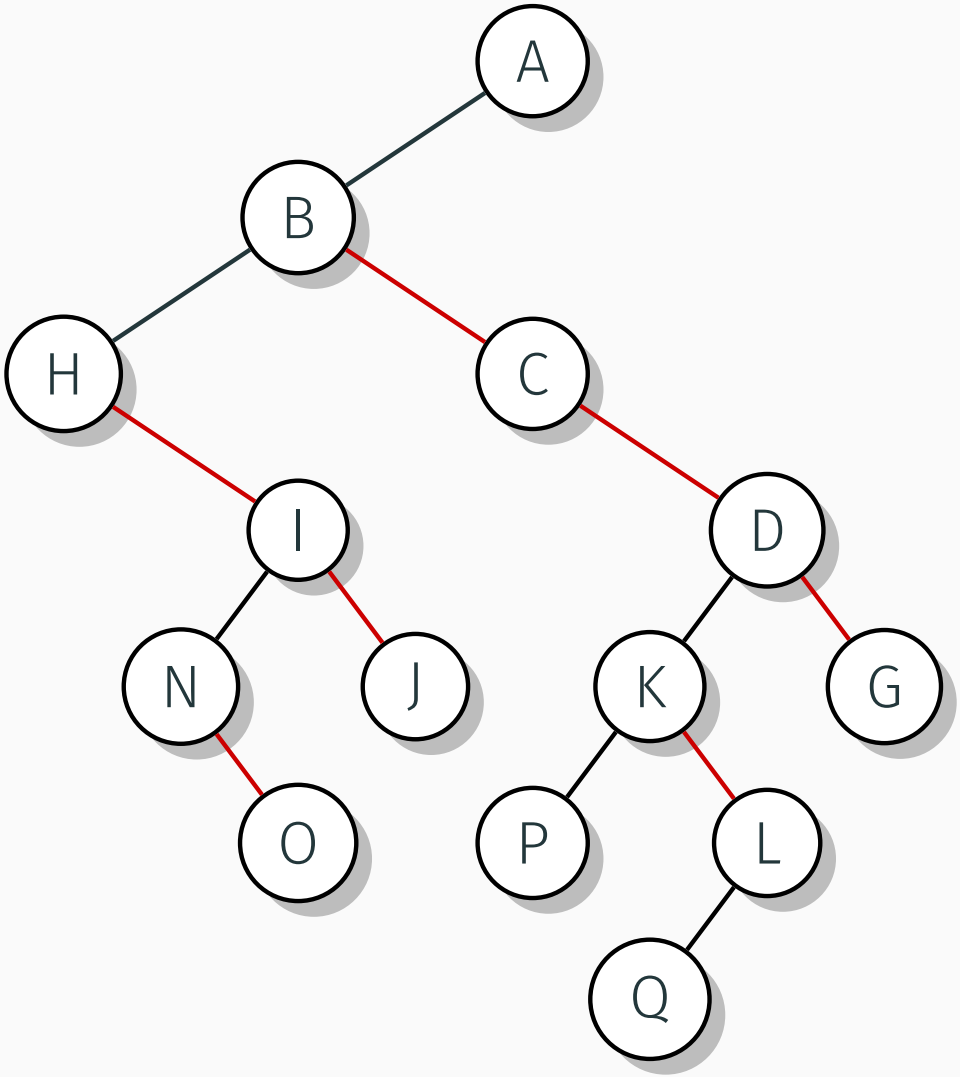
Ordered Tree Representation – first child – next sibling



First child – next sibling representation – each node contains two pointers:

1. pointer to the first child, and
2. pointer to the sibling.

Ordered Tree Representation – Knuth's transformation



First child – next sibling representation rotated by **45**° clockwise.

Set

- We will understand the data structure set as an unordered collection (even empty) of mutually distinct elements.
- A set can be defined in two ways:
  1. **by listing elements**, e.g., $M = \{2, 3, 5, 7\}$ or
  2. **by properties** that the elements must satisfy, e.g., $M = \{n,$ prime numbers less than $10\}$.
- The most important set operations:
  - membership query, i.e., the question "Is x an element of M?",
  - union of two sets, and
  - intersection of two sets.

Set – implementation

Bit vector

- universe $U = \{u_0, u_1, \ldots, u_{n-1}\}$, $|U| = n$
- any set M is considered a subset of the universe U
- bit vector $\vec{b}$ of dimension n, where

$$\vec{b}_i = \begin{cases} 1 & u_i \in M \\ 0 & \text{otherwise} \end{cases}$$

Example

$U = \{0, 1, 2, \ldots, 8, 9\}$

$M = \{2, 3, 5, 7\}$

$\vec{b} = (0, 0, 1, 1, 0, 1, 0, 1, 0, 0)$

Set – implementation

Listing elements

- a set is represented by listing the elements it contains
- depending on the circumstances, we can use arrays, linked lists, binary search trees, hash tables, etc., to store the elements
- it always depends on the specific problem which operations are essential: maintaining order or other considerations

Dictionary

- If an additional piece of information is associated with an element of a set, we then talk about a **dictionary**.
- A dictionary maintains pairs (key, value), where the key must be unique in the dictionary.
- Mathematically, it is a **mapping**.
- The most important operations:
  - inserting a pair into the dictionary
  - deleting a pair from the dictionary
  - modifying a value in the dictionary
  - finding a value for a given key
- For implementation, arrays, linked lists, binary search trees, hash tables, etc., can be used.

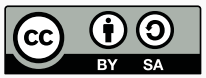
Thanks for your attention

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Fundamentals of the Analysis of Algorithm Efficiency

Jiří Dvorský, Ph.D.

Department of Computer Science
VSB – Technical University of Ostrava



Fundamentals of the Analysis of Algorithm Efficiency

Basics of algorithm complexity analysis

Algorithm analysis

What to analyze?

- correctness
- time complexity
- space complexity
- optimality

Possible approaches

- empirical and
- theoretical

Time and Space Complexity of an Algorithm

- **Time complexity** – how long the algorithm will run.
- **Space complexity** – how much extra memory the algorithm will need in addition to the storage required for the data itself.
- Previously, both resources were critical.
- Thanks to advances in computing technology, memory is relatively abundant.
- We will examine time complexity – significant progress can be made here.
- It turns out that space complexity can be studied using the same apparatus as time complexity.

Measuring Input Size

- Trivial observation – larger data usually takes an algorithm longer to process.
- We introduce the parameter n denoting the size of the input data, which represents for example:
  - searching in a list, array – length of the array
  - evaluating the polynomial $p(x) = a_n x^n + \cdots + a_1 x + a_0$ at point x – degree of the polynomial
  - multiplying matrices of type $n \times n$ – dimension of the matrix. The actual number of input numbers is n^2, but this still depends on n
  - spell checking – number of characters or number of words, depending on what the algorithm works with

Measuring Input Size (cont.)

- primality testing – the input is always a single number, the running time depends on the size of the number (compare testing 2^3 and 2^{64}), the input size will be the number of bits required to write the number

$$n = \lfloor \log_2 a \rfloor + 1 \tag{2}$$

- graph problems – number of vertices and/or number of edges – here we already have two parameters

Empirical Measurement of Complexity

- We provide suitable input data and measure the program's running time in standard units of time.
- Disadvantages:
  - Dependence on specific hardware, implementation method, and compiler.
  - We want to measure algorithm complexity – we do not have the means to capture the aforementioned influences.
  - Hardware development – does this mean algorithms are accelerating? No, they remain the same.
  - The number of operations performed by the program can be difficult to determine.
  - We want to avoid implementation – after all, we are examining algorithms.

Time complexity of the algorithm

Time complexity of the algorithm will be expressed (measured) by **the number of performed basic operations** with respect to (as a function of) **the size of the input n**:

$$T(n) \approx c_{op} C(n),$$

where

- n is the size of the input,
- $T(n)$ is the running time of the algorithm,
- c_{op} is the time to perform one basic operation and
- $C(n)$ is the number of basic operations.

Basic Operations

Typical operations for a given algorithm that significantly contribute to the overall "running time" of the algorithm.

Problem	Input size	Basic operation
Searching for an element in a list	Number of elements in the list	Comparing elements
Matrix multiplication	Matrix dimensions	Arithmetic operations (multiplication)
Primality testing	Number of bits of the number	Dividing numbers
Graph problems	Number of vertices and/or edges	Processing a vertex or traversing an edge

Order of Growth of Complexity

Regarding the relationship

$$T(n) \approx c_{op} C(n),$$

we must approach it with caution, because

1. $C(n)$ does not take into account the influence of operations other than basic ones and
2. c_{op} cannot be reliably determined.

We understand this relationship as a **reasonable estimate of the algorithm's running time**, except for extremely small n.

Order of Growth of Complexity (cont.)

Problem

How many times faster will my algorithm run on a computer that is **10×** faster than my current computer?

Solution

Of course, **10×**, c_{op} is one-tenth.

Order of Growth of Complexity (cont.)

Problem

How many times longer will my algorithm run for a twice-as-large input when $C(n) = \frac{1}{2}n(n-1)$?

Solution

We approximate from above the number of operations $C(n)$

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

However, in the context of order of growth, lower-order terms like $-\frac{1}{2}n$ are typically ignored. Thus,

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

Order of Growth of Complexity (cont.)

and

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

Therefore,

$$\frac{C(2n)}{C(n)} = \frac{2n^2}{\frac{1}{2}n^2} = 4$$

Order of Growth of Complexity (cont.)

Remarks

- The base of the logarithm is not significant: $\log_a n = \log_a b \cdot \log_b n$.
- A computer with a speed of 10^{12} (one trillion) operations per second would take approximately 40 billion years to perform $2^{100} \approx 1.3 \cdot 10^{30}$ operations. The age of the Earth is approximately 4.4 billion years.
- We will not even consider performing **100!** operations...

Algorithms with exponential or factorial order of complexity are only usable for very small input sizes!

Order of Growth of Complexity (cont.)

Problem

How many times longer will my algorithm run for a twice-as-large input, for algorithms with different orders of growth?

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
$2n$	$+1$	$2\times$	$\approx 2\times$	$4\times$	$8\times$	$(\dots)^2$	n/a

because

$$\log_2(2n) = \log_2 2 + \log_2 n = 1 + \log_2 n$$

$$2^{2n} = (2^n)^2$$

Fundamentals of the Analysis of Algorithm Efficiency

Worst, Best and Average Case

Worst, Best and Average Case

- The number of basic operations is expressed as a function with one parameter n, the input size.
- Some algorithms may have different numbers of basic operations even for the same n, such as the linear search algorithm.

Input : Array $A[0 \dots n-1]$ and the target element x
Output: Index of the first occurrence of element x in array A, otherwise -1

```
1 for i ← 0 to n - 1 do
2 |   if A[i] = x then
3 |   |   return i;
4 |   end
5 end
6 return -1;
```

Worst, Best and Average Case (cont.)

Significant numbers of basic operations:

- $C_{worst}(n)$ – worst case, highest number of operations
- $C_{best}(n)$ – best case, lowest number of operations
- $C_{avg}(n)$ – average case, average number of operations.

Worst-case scenario $C_{worst}(n)$

- We analyze the algorithm and look for an input of size n that results in the maximum possible number of operations.
- The worst-case scenario provides an upper bound on complexity, all other cases are either the same or better.
- A low number of operations in the worst case – good news.

Example

Linear search: element x in array A is not found or is found at the end, thus $C_{worst}(n) = n$.

Best-case scenario $C_{best}(n)$

- Generally, we seek an input of size n for which the algorithm performs the smallest number of operations.
- The average best-case scenario is not as crucial as the worst-case scenario.
- Inputs are ”similar” and ”close” to the best case. Sorting nearly sorted sequences.
- A best-case scenario with a ”frightening” number of operations – generally bad news and ”final” for the algorithm. But for an encryption algorithm, a ”frightening” number of cryptanalysis operations is necessary even in the best case.

Example

Linear search: element x is the first element in array A, $C_{best}(n) = 1$.

Average case $C_{avg}(n)$

- Number of operations in the average, “typical”, “random” case (best and worst cases are extremes).
- It is not the average of the best and worst case!
- We must take into account the probabilities of individual possible inputs of size n.
- Analysis of the average case is thus more complicated than the previous two.
- There are algorithms where the worst and average number of operations differ significantly, for example QuickSort.

Average Case $C_{avg}(n)$ – Linear Search

Assumptions

1. probability of successful search p, where $0 \leq p \leq 1$
2. probability of finding at all positions in the array is the same and equals $\frac{p}{n}$

Average Case $C_{avg}(n)$ – Linear Search (cont.)

Successful Search

- finding at the first position – one comparison with probability $\frac{p}{n}$,
- finding at the second position – two comparisons with probability $\frac{p}{n}$, and so on, thus

$$1\frac{p}{n} + 2\frac{p}{n} + \cdots + i\frac{p}{n} + \cdots + n\frac{p}{n}$$

Average Case $C_{avg}(n)$ – Linear Search (cont.)

Unsuccessful Search

- probability of failure is $1 - p$ and we perform n comparisons, i.e., $n(1 - p)$

From this

$$\begin{aligned} C_{avg}(n) &= \left(1\frac{p}{n} + 2\frac{p}{n} + \cdots + i\frac{p}{n} + \cdots + n\frac{p}{n}\right) + n(1 - p) \\ &= \frac{p}{n}(1 + 2 + \cdots + i + \cdots + n) + n(1 - p) \\ &= \frac{p}{n}\left[\frac{1}{2}n(n + 1)\right] + n(1 - p) \\ &= \frac{1}{2}p(n + 1) + n(1 - p) \end{aligned}$$

Average Case $C_{avg}(n)$ – Linear Search (cont.)

Analysis

- always successful search, $p = 1$ and thus $C_{avg}(n) = \frac{1}{2}(n + 1)$
- unsuccessful search, $p = 0$ and thus $C_{avg}(n) = n$

Amortized Complexity

- We do not examine a single, isolated run of the algorithm, but rather examine a “set” of runs with different inputs of the same size.
- We are interested in the total number of operations for the set.
- The number of operations for one input from the set may be high, but it is balanced, “amortized” by a significantly smaller number of operations for other inputs from the set.
- For example, one of the inputs causes a significant change in the data structure, making the processing of subsequent inputs easier.
- In industry, for example, the purchase of an expensive machine is amortized by cheaper production of products.

Sources for Independent Study

- Book [2], chapter 2.1, pages 42 – 51
- Book [3], chapter 2.2, pages 25 – 34 partially

Fundamentals of the Analysis of Algorithm Efficiency

Asymptotic Notation of Complexity

Big O Notation

Definition

Let us have functions $t(n)$ and $g(n)$, where $t(n), g(n) : \mathbb{N} \to \mathbb{N}$. We say that function $t(n)$ belongs to $O(g(n))$, if there exists a positive non-zero real constant c and a natural number $n_0 \geq 0$ such that

$$t(n) \leq cg(n)$$

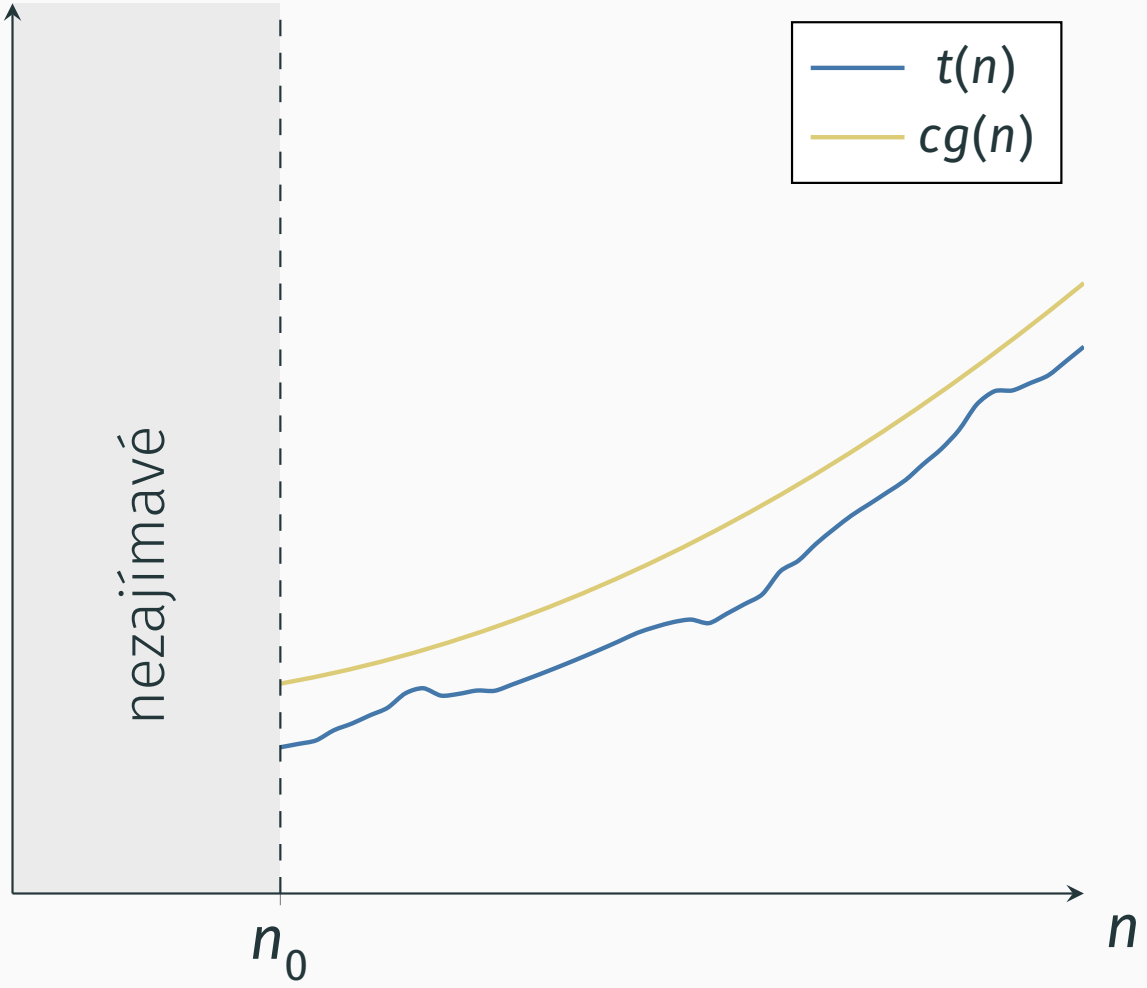
for all $n \geq n_0$.

Remark

Instead of saying "$t(n)$ belongs to $O(g(n))$", we can say that "$t(n)$ is of order $O(g(n))$".

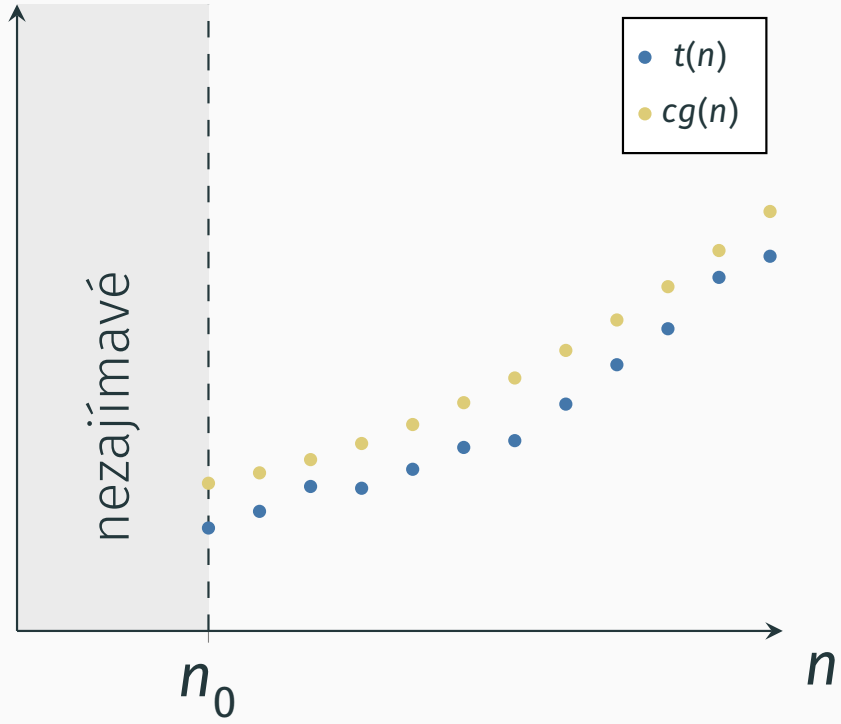
Big O notation graphically

$$t(n) \in O(g(n))$$



Big O notation – formally correct graph

Formally, the domain of definition and the range of values of functions $t(n)$ and $g(n)$ are natural numbers ⇒ the graph should consist only of points, not curves.



If we interpolate the points with a curve ⇒ we obtain continuous functions ⇒ we can use mathematical analysis (limits, derivatives, etc.) for calculations.

Big O notation – example 1

Problem statement

Prove that $3n + 7 \in O(n)$.

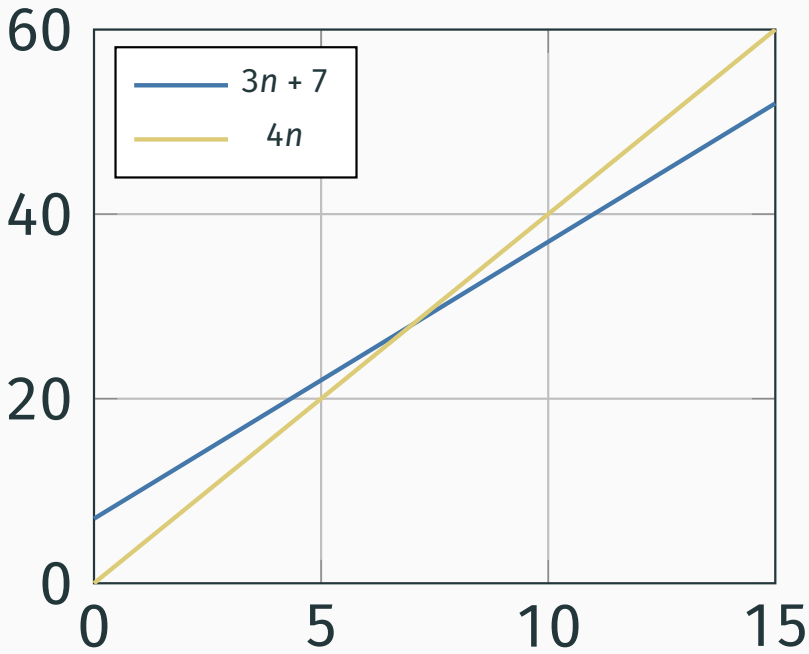
Solution

1. We seek constants c and n_0 such that

$$3n + 7 \leq cn$$

holds for all $n \geq n_0$.

2. It is clear that necessarily $c > 3$. If we choose, for example, $c = 4$, then $n_0 = 7$.



Big O notation – example 2

Problem statement

Prove that $3n + 7 \in O(n^2)$.

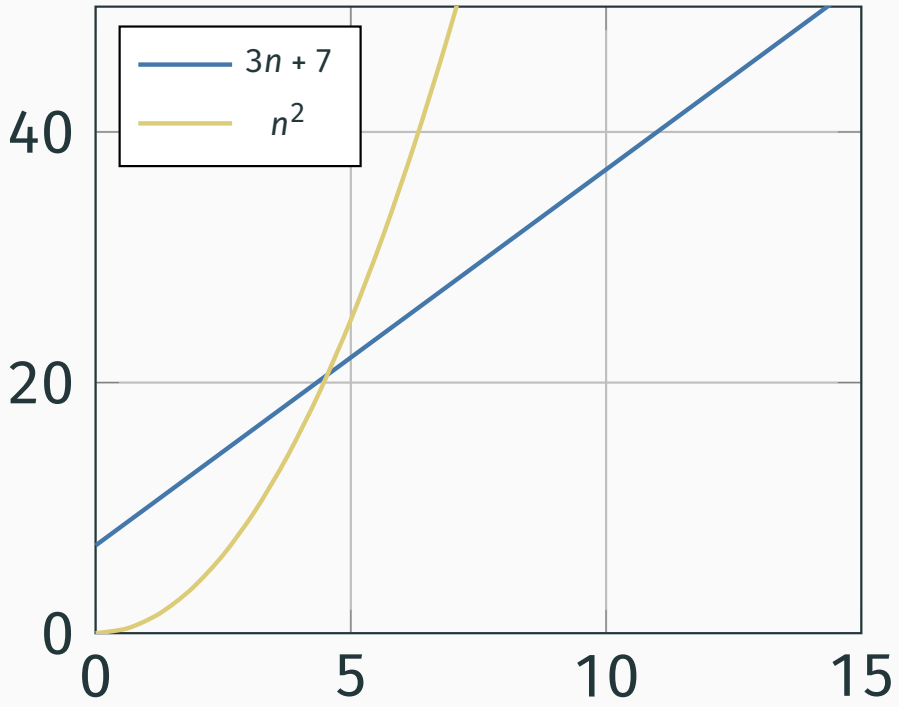
Solution

1. We are looking for constants c and n_0 such that

$$3n + 7 \leq cn^2$$

holds for all $n \geq n_0$.

2. If we choose $c = 1$, then $n_0 = 5$.



Big O notation – example 3

Problem statement

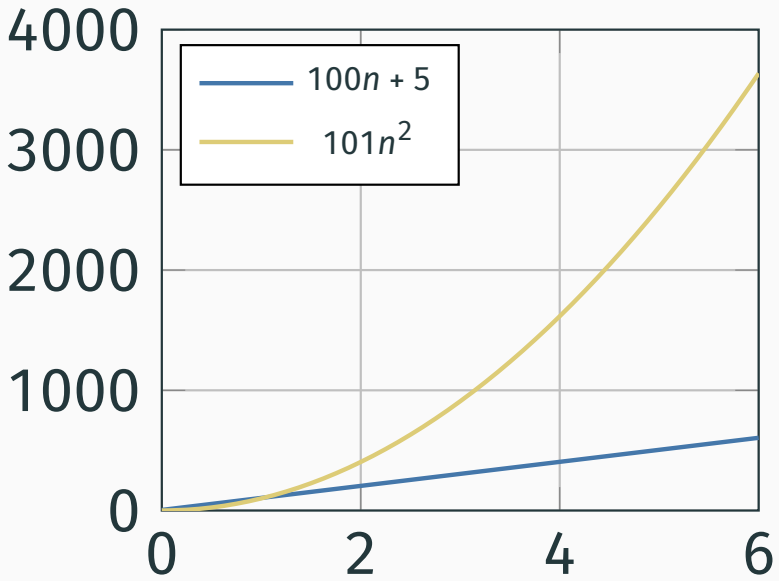
Prove that $100n + 5 \in O(n^2)$.

Solution

1. It holds that $100n + 5 \leq 100n + n$ for all $n \geq 5$.
2. Furthermore, it holds that $101n \leq 101n^2$.
3. From this

$$100n + 5 \leq 101n \leq 101n^2$$

and thus $c = 101$ and $n_0 = 5$.



Big O notation – example 3 (cont.)

The proof can also be conducted as follows:

$$100n + 5 \leq 100n + 5n = 105n$$

for all $n \geq 1$. This implies that

$$105n \leq 105n^2$$

and thus $c = 105$ and $n_0 = 0$.

The definition of Big O notation does not say anything about the uniqueness of the values c and n_0, it only requires their existence.

Omega notation

Definition

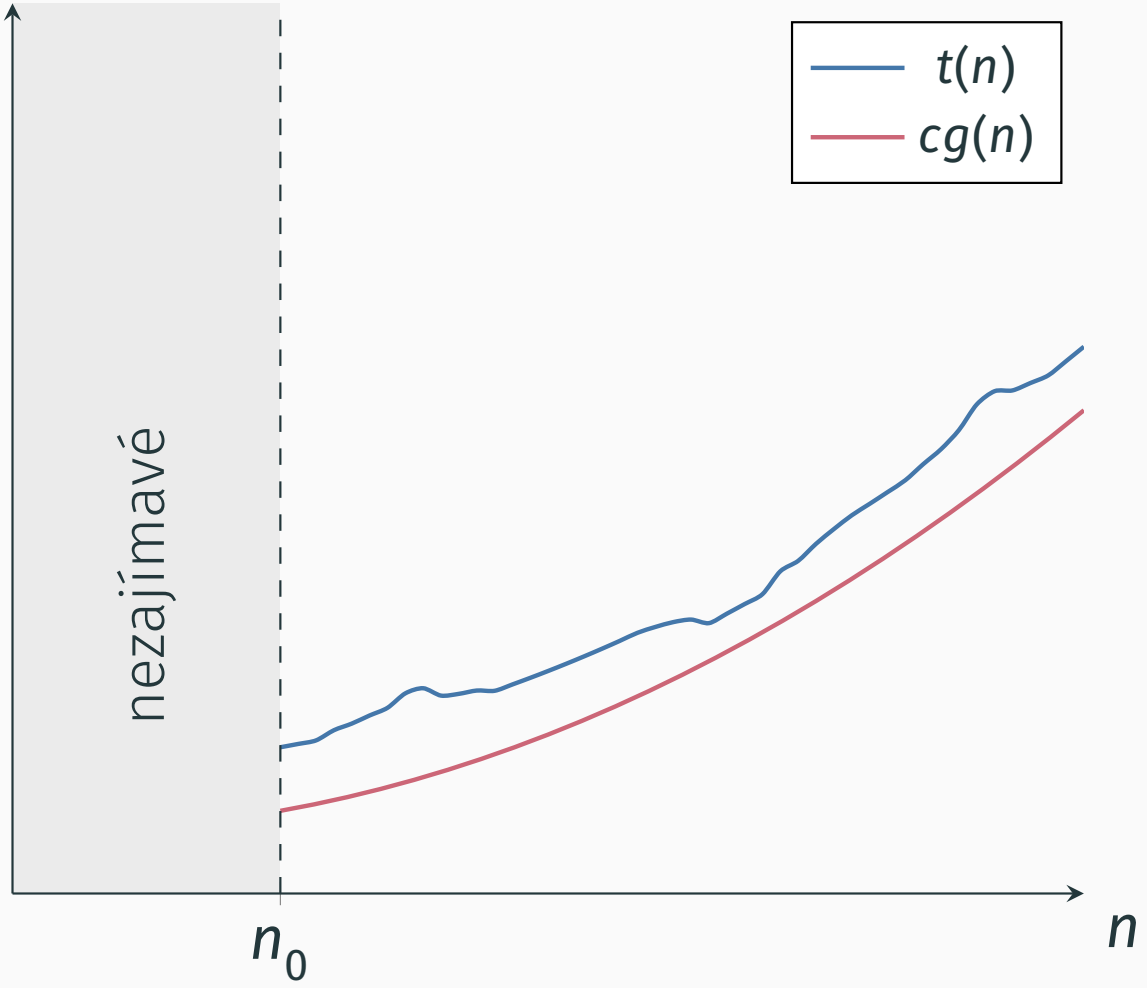
Given functions $t(n)$ and $g(n)$, where $t(n), g(n) : \mathbb{N} \to \mathbb{N}$. We say that function $t(n)$ belongs to $\Omega(g(n))$, if there exists a positive non-zero real constant c and a natural number $n_0 \geq 0$ such that

$$t(n) \geq cg(n)$$

for all $n \geq n_0$.

Lower bound notation graphically

$$t(n) \in \Omega(g(n))$$



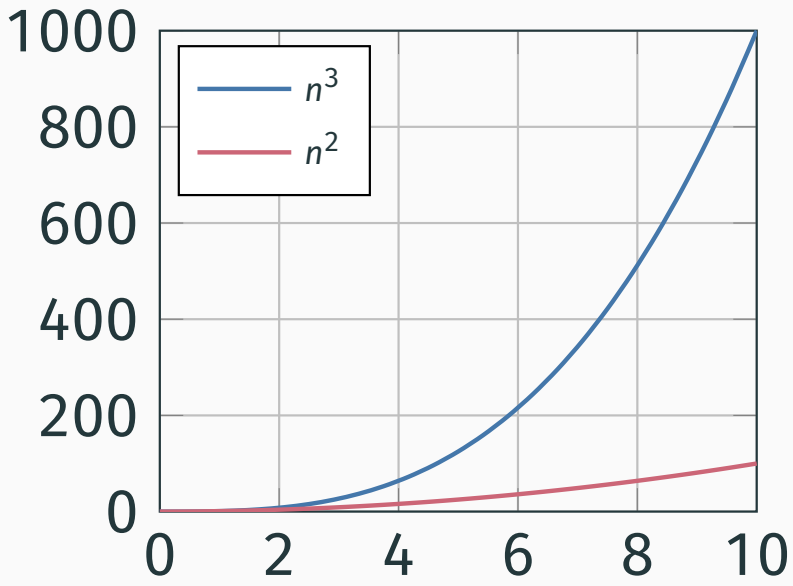
Ω-notation – example 1

Problem statement

Prove that $n^3 \in \Omega(n^2)$.

Solution

1. Clearly, it holds that $n^3 \geq n^2$ for all $n \geq 0$.
2. Thus we can choose $c = 1$ and $n_0 = 0$.



Omega notation – example 2

Problem statement

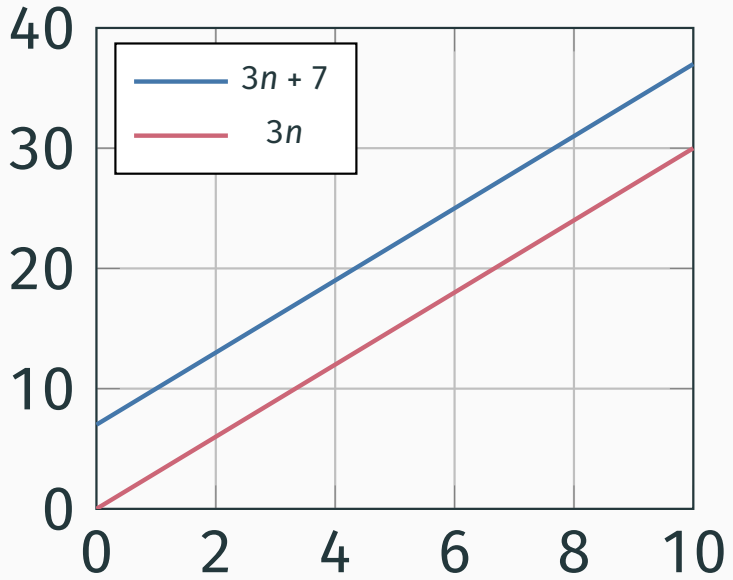
Prove that $3n + 7 \in \Omega(n)$.

Solution

1. We are looking for constants c and n_0 such that

$$3n + 7 \geq cn$$

   holds for all $n \geq n_0$.
2. The expression $3n + 7 \geq 3n$ is valid for all $n \geq 0$, so $c = 3$ and $n_0 = 0$.



Theta notation

Definition

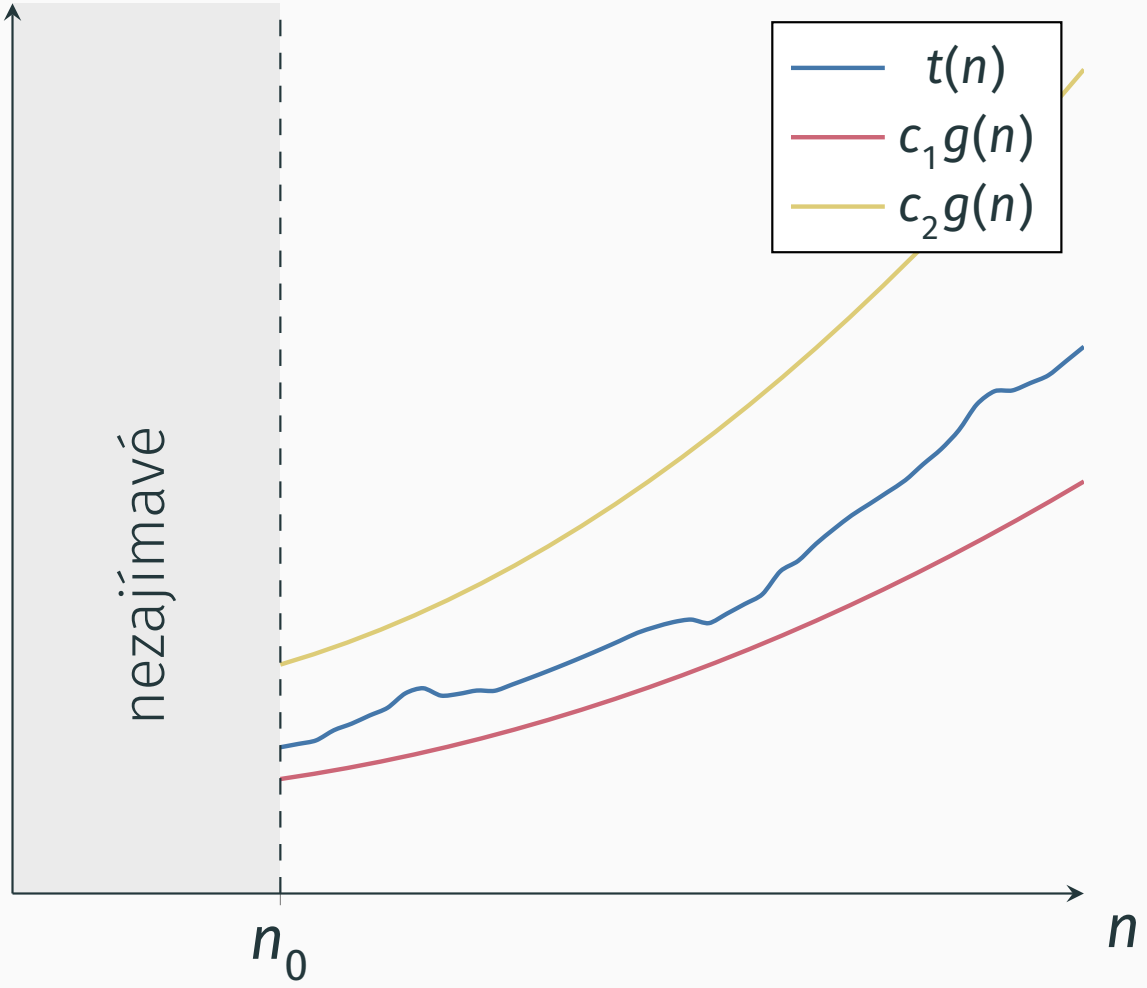
Given functions $t(n)$ and $g(n)$, where $t(n), g(n) : \mathbb{N} \to \mathbb{N}$. We say that the function $t(n)$ belongs to $\Theta(g(n))$, if there exist positive nonzero real constants c_1, c_2 and a natural number $n_0 \geq 0$ such that

$$c_1 g(n) \leq t(n) \leq c_2 g(n)$$

for all $n \geq n_0$.

Theta notation graphically

$$t(n) \in \Theta(g(n))$$



Θ-notation – example

Problem Statement

Prove that $\frac{1}{2}n(n-1) \in \Theta(n^2)$.

Solution

1. First, we prove the right inequality $t(n) \leq c_2 g(n)$ (upper bound)

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2$$

   for all $n \geq 0$.

Θ-notation – example (cont.)

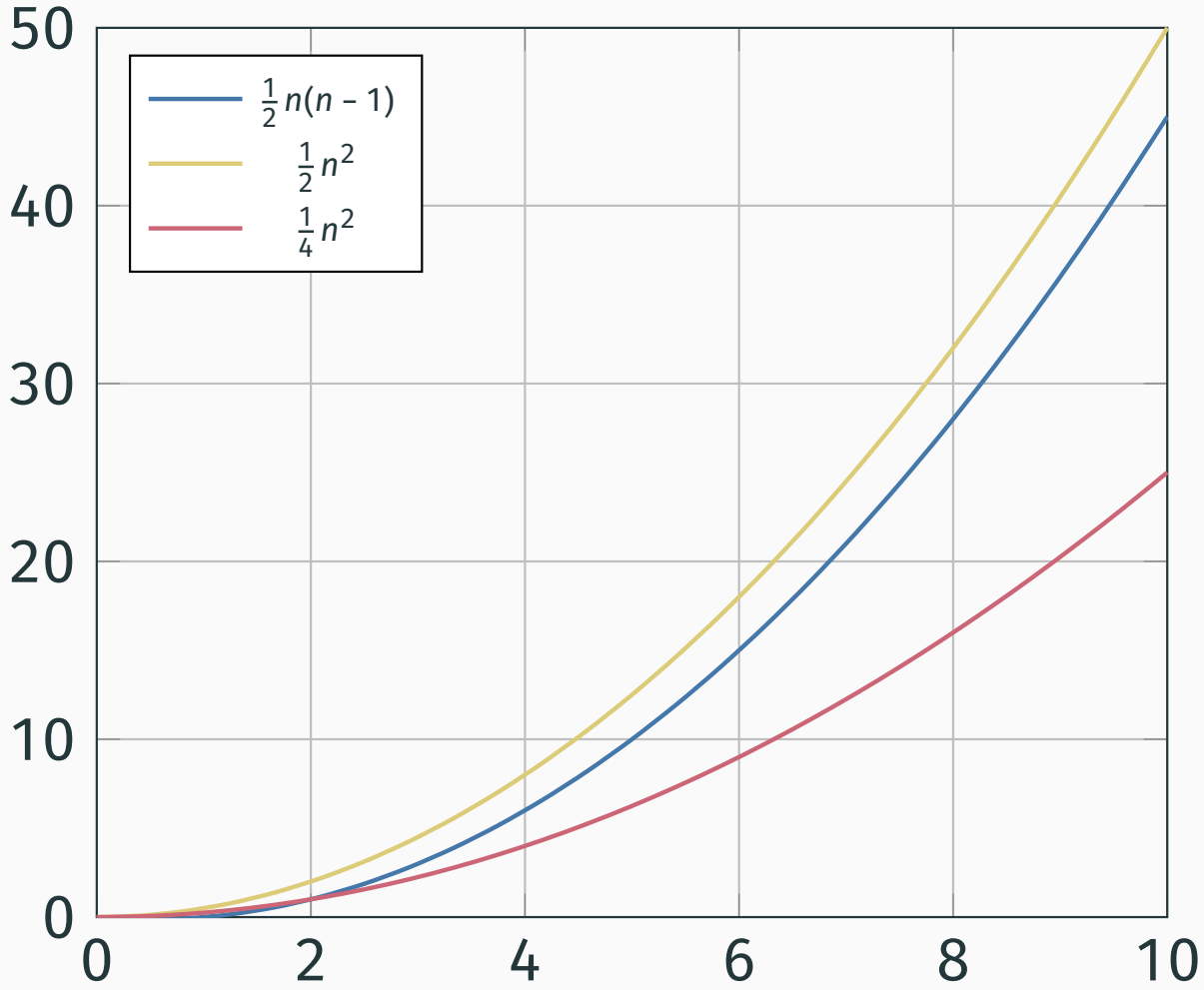
2. The left inequality $c_1 g(n) \leq t(n)$ (lower bound) can be proven as follows:

$$\begin{aligned} t(n) = \frac{1}{2}n(n-1) &= \frac{1}{2}n^2 - \frac{1}{2}n \\ &\geq \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n \\ &\geq \frac{1}{2}n^2 - \frac{1}{4}n^2 \\ &\geq \frac{1}{4}n^2 \end{aligned}$$

   In summary, $\frac{1}{4}n^2 \leq \frac{1}{2}n(n-1)$ for all $n \geq 2$.

3. From the previous inequalities, it follows that $c_1 = \frac{1}{4}$, $c_2 = \frac{1}{2}$, and $n_0 = 2$.

Θ-notation – example (cont.)



Properties of asymptotic notation

Basic properties:

1. $f(n) \in O(f(n))$
2. $f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n))$
3. $f(n) \in O(g(n)) \land g(n) \in O(h(n)) \implies f(n) \in O(h(n))$
4. $\Theta(f(n)) = O(f(n)) \land \Omega(f(n))$

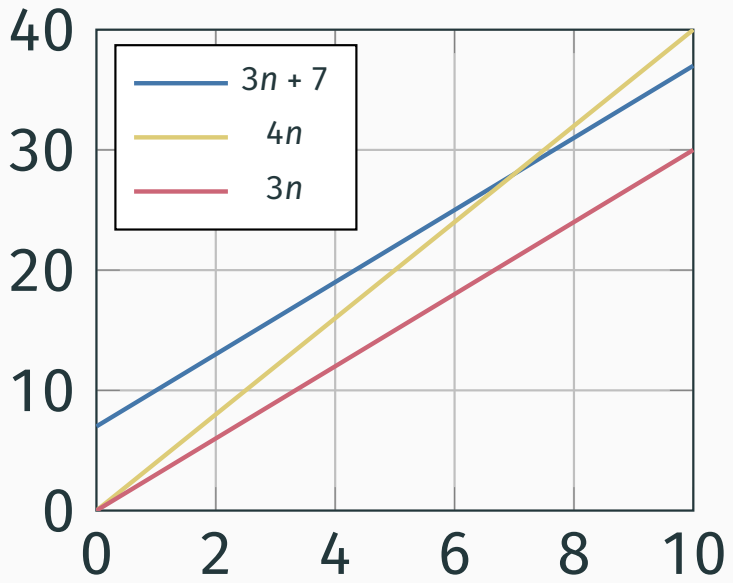
Properties of Asymptotic Notation – Application

Task

Prove that $3n + 7 \in \Theta(n)$.

Solution

1. From previous examples, we know that $3n + 7 \in O(n)$ and simultaneously $3n + 7 \in \Omega(n)$.
2. Therefore, it holds that $3n + 7 \in \Theta(n)$.
3. Specifically, $c_1 = 3$, $c_2 = 4$ and $n_0 = 7$.



Properties of Asymptotic Notation – Computing Complexity

- Algorithm A consists of parts A_1 and A_2.
- The parts of the algorithm are executed sequentially, i.e., after completing A_1, A_2 begins execution.
- The complexity of part A_1 is $t_1(n) \in O(g_1(n))$, the complexity of part A_2 is $t_2(n) \in O(g_2(n))$.
- The question is – what is the overall complexity of algorithm A?

Properties of asymptotic notation – auxiliary lemma

Lemma

Let us have arbitrary real numbers a_1, a_2, b_1, b_2. *Then the following holds:*

$$a_1 \leq b_1 \wedge a_2 \leq b_2 \implies a_1 + a_2 \leq 2 \max(b_1, b_2).$$

Properties of asymptotic notation – auxiliary lemma (cont.)

Proof.

From the assumption, we know that

$$\begin{array}{rcl} a_1 & \leq & b_1 \\ a_2 & \leq & b_2 \\ \hline a_1 + a_2 & \leq & b_1 + b_2. \end{array}$$

Furthermore, it holds that

$$b_1 + b_2 \leq 2 \max(b_1, b_2).$$

From this, we obtain

$$a_1 + a_2 \leq b_1 + b_2 \leq 2 \max(b_1, b_2).$$

□

Properties of asymptotic notation – computation of complexity

Theorem

If $t_1(n) \in O(g_1(n))$ *and simultaneously* $t_2(n) \in O(g_2(n))$*, then*

$$t_1(n) + t_2(n) \in O(\max(g_1(n), g_2(n))).$$

Remark

The same statement can be expressed for Ω and Θ notation.

Properties of asymptotic notation – computation of complexity (cont.)

Proof.

Since $t_1(n) \in O(g_1(n))$, there exists a positive non-zero constant c_1 and a non-negative constant n_1 such that

$$t_1(n) \leq c_1 g_1(n) \qquad \forall n \geq n_1.$$

Similarly,

$$t_2(n) \leq c_2 g_2(n) \qquad \forall n \geq n_2.$$

Properties of asymptotic notation – computation of complexity (cont.)

Proof.

Let $c_3 = \max(c_1, c_2)$ and $n_0 \geq \max(n_1, n_2)$. Then,

$$\begin{aligned} t_1(n) + t_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\leq 2c_3 \max(g_1(n), g_2(n)). \end{aligned}$$

Thus, $t_1(n) + t_2(n) \in O(\max(g_1(n), g_2(n)))$, since there exist constants $c = 2c_3 = 2\max(c_1, c_2)$ and $n_0 = \max(n_1, n_2)$. □

The overall complexity of an algorithm is determined by the part with the highest complexity.

Properties of asymptotic notation – complexity calculation, example

Problem statement

Test whether two identical values occur in the array.

Solution

1. Sorting the array requires no more than $\frac{1}{2}n(n-1)$ comparisons, i.e., a complexity of class $O(n^2)$.
2. Comparing all pairs of adjacent elements will require $n-1$ comparisons, i.e., a complexity of class $O(n)$.

The overall complexity of the algorithm is therefore $O(\max(n^2, n)) = O(n^2)$.

Utilization of limits for computations

The growth rate of functions can be more easily calculated using limits:

$$\lim_{n\to\infty}\frac{t(n)}{g(n)} = \begin{cases} 0 & t(n)\ grows slower than\ g(n) \\ c & t(n)\ grows at the same rate as\ g(n) \\ \infty & t(n)\ grows faster than\ g(n) \end{cases}$$

It is clear that:

$t(n) \in O(g(n)) \quad \Leftrightarrow \quad t(n)$ grows slower or at the same rate as $g(n)$

$t(n) \in \Omega(g(n)) \quad \Leftrightarrow \quad t(n)$ grows at the same rate or faster than $g(n)$

$t(n) \in \Theta(g(n)) \quad \Leftrightarrow \quad t(n)$ grows at the same rate as $g(n)$

Utilization of limits for computations (cont.)

Some useful formulas

L'Hospital's rule

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

Stirling's formula

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Using limits for calculations – example I

Compare the growth rate of functions $\frac{1}{2}n(n-1)$ and n^2.

$$\begin{aligned}
\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} &= \frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2} \\
&= \frac{1}{2}\lim_{n\to\infty}\left(1-\frac{1}{n}\right) \\
&= \frac{1}{2}\left(\lim_{n\to\infty}1-\lim_{n\to\infty}\frac{1}{n}\right) \\
&= \frac{1}{2}(1-0) = \frac{1}{2} > 0
\end{aligned}$$

The functions $\frac{1}{2}n(n-1)$ and n^2 grow at the same rate, so

$$\frac{1}{2}n(n-1) \in \Theta(n^2)$$

.

Utilization of limits for computations – example II

Compare the growth rate of functions $\log_2 n$ and $\sqrt{n}$.

$$\begin{aligned}
\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} &= \lim_{n\to\infty} \frac{\left(\log_2 n\right)'}{\left(\sqrt{n}\right)'} \\
&= \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = (\log_2 e) \lim_{n\to\infty} \frac{\frac{1}{n}}{\frac{1}{2\sqrt{n}}} \\
&= \log_2 e \lim_{n\to\infty} \frac{2\sqrt{n}}{n} \\
&= 2\log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0
\end{aligned}$$

The function $\log_2 n$ therefore **grows more slowly** than $\sqrt{n}$.

Using limits for computations – example III

Compare the growth rate of the functions $n!$ and 2^n.

$$\begin{aligned}\lim_{n\to\infty}\frac{n!}{2^n} &= \lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}\\ &= \sqrt{2\pi}\lim_{n\to\infty}\sqrt{n}\frac{n^n}{2^n e^n}\\ &= \sqrt{2\pi}\lim_{n\to\infty}\sqrt{n}\left(\frac{n}{2e}\right)^n = \infty\end{aligned}$$

Remarks

- The function $n!$ therefore grows faster than 2^n.
- The definition of Θ-notation does not exclude that $n! \in \Omega(2^n)$, but the limit calculation clearly states that $n!$ grows faster than 2^n

Basic Complexity Classes

Although theoretically there are infinitely many complexity classes, the complexity of most algorithms falls into a few classes.

Class	Name	Note
1	constant	complexity does not depend on the size of the input; very few algorithms
$\log n$	logarithmic	typically algorithms reducing the size of the input by a constant factor; interval halving search
n	linear	algorithms processing a list of n elements; e.g. sequential search
$n \log n$	linearithmic	divide and conquer algorithms; average complexity of QuickSort, MergeSort

Basic Complexity Classes (cont.)

Class	Name	Note
n^2	quadratic	generally algorithms with two nested loops; elementary sorting methods, summing $n \times n$ matrices
n^3	cubic	generally algorithms with three nested loops; multiplying $n \times n$ matrices
2^n	exponential	typically generating all subsets of an n-element set
$n!$	factorial	typically generating all permutations of an n-element set

Influence of the Multiplicative Constant

- The complexity class is given up to a multiplicative constant, which is usually not precisely specified.
- Could an algorithm with a higher complexity class therefore run faster than an algorithm from a better class for some reasonable n? For example:

Algorithm	Running Time
A	n^3
B	$10^6 n^2$

A will be better than B for $n < 10^6$.

- Multiplicative constants usually take on similar, relatively small values.
- **It can be expected that algorithms with lower complexity will be better than those with higher complexity already for moderately large inputs.**

Sources for Independent Study

- Book [2], chapter 2.2, pages 52 – 61
- Book [3], chapters 3.1 and 3.2, pages 49 – 63

Fundamentals of the Analysis of Algorithm Efficiency

Analysis of Non-Recursive Algorithms

Finding the Largest Element in an Array of n Numbers

```
   Input  : Array A[0 ... n - 1] of integers
   Output: Largest element of array A
1  max ← A[0];
2  for i ← 1 to n - 1 do
3  |   if A[i] > max then
4  |   |   max ← A[i];
5  |   end
6  end
7  return max;
```

Finding the Largest Element in an Array of n Numbers (cont.)

Working Procedure

1. Input size – size of array n
2. Basic operation:
   - most frequently performed operations are inside the loop – comparison $A[i] > max$ and assignment $max \leftarrow A[i]$
   - basic operation will be **comparison**, because it
     - is performed in each iteration of the loop,
     - is the key operation for the algorithm, "How many pairs of elements must I compare to find the maximum?"
3. Number of comparisons is the same for all inputs of size n, it is not necessary to distinguish between the best, average, and worst case

Finding the Largest Element in an Array of n Numbers (cont.)

4. Number of basic operations, comparisons, $C(n)$ will be equal to

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

5. **Conclusion**: Finding the largest element in an array of n numbers is a **linear algorithm**.

Finding the largest element in an array of n numbers, all operations

Number of operations	Description
1	assignment $max \leftarrow A[0]$
1	assignment $i \leftarrow 1$
$n - 1$	comparison $i \leq n - 1$
$n - 1$	increment i by 1
$n - 1$	comparison $A[i] > max$
$n - 1$	assignment $max \leftarrow A[i]$
1	return result **return** max
$4(n - 1) + 3 = 4n - 1 \in \Theta(n)$	

Conclusion: Finding the largest element in an array of n numbers is a **linear algorithm**.

General procedure for determining the time complexity of non-recursive algorithms

1. Selection of a parameter, or parameters, representing the size of the input n.
2. Identification of the basic operations of the algorithm (these are the ones in the most nested loop!).
3. Does the number of basic operations depend only on the size of the input? If it depends on something else as well, we must examine the worst, best, and average cases separately.
4. Establishment of a relationship, or relationships, (i.e., "formulas") expressing the number, or numbers, of executions of the basic operations.
5. Simplification of the established relationships and, at least, determination of the order of growth.

Useful Summation Formulas

$$\sum (a_i \pm b_i) = \sum a_i \pm \sum b_i \tag{3}$$

$$\sum c a_i = c \sum a_i \tag{4}$$

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{m} a_i + \sum_{i=m+1}^{n} a_i \tag{5}$$

$$\sum_{i=l}^{u} 1 = 1 + 1 + \cdots + 1 = u - l + 1 \tag{6}$$

Specifically

$$\sum_{i=1}^{n} 1 = n \in \Theta(n) \tag{7}$$

Useful Summation Formulas (cont.)

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{1}{2}n(n+1) \approx \frac{1}{2}n^2 \in \Theta(n^2) \tag{8}$$

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{1}{6}n(n+1)(2n+1) \approx \frac{1}{3}n^3 \in \Theta(n^3) \tag{9}$$

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1}, \text{ for } a \neq 1 \tag{10}$$

Specifically

$$\sum_{i=0}^{n} 2^i = 2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1 \in \Theta(2^n) \tag{11}$$

Uniqueness of elements in an array

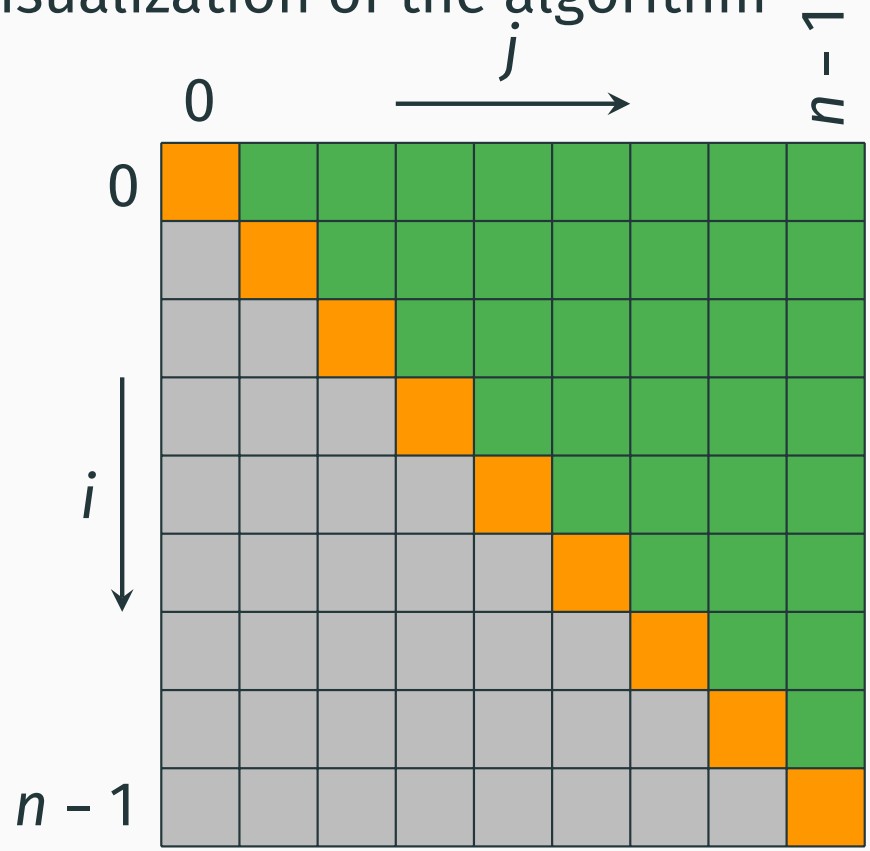
Given is an array of n elements. Our task is to analyze the algorithm that determines whether all elements in the array are mutually distinct, i.e., unique.

Input : Array $A[0 \ldots n-1]$
Output: Returns true if all elements are unique, otherwise returns false

```
1 for i ← 0 to n − 2 do
2 |   for j ← i + 1 to n − 1 do
3 |   |   if A[i] = A[j] then
4 |   |   |   return false;
5 |   |   end
6 |   end
7 end
8 return true;
```

Uniqueness of elements in an array (cont.)

Visualization of the algorithm



Legend

pairs that **must** be tested

an element with itself does not need to be tested

pairs already tested in previous iterations of the cycle

Uniqueness of elements in an array (cont.)

Procedure

1. Input size – size of the array n
2. Basic operation – the most nested cycle contains a single operation, comparison $A[i] = A[j]$
3. Dependence only on n? No, the number of basic operations depends also on whether a duplicate element appears in the array. Thus, we perform analysis of the **worst**, best, and average case.
4. Establishing relationships. For the worst case, it is clear from the inner cycle that premature termination of the cycle must not occur, either:
   4.1 because all elements are unique or

Uniqueness of elements in an array (cont.)

4.2 a duplicate appears only in the last pair.

Thus, we perform:

- one comparison for each iteration of the inner cycle, i.e., $j = i + 1, \ldots, n - 1$
- the outer cycle iterates $n - 1$ times

Establishing relationships. For the worst case, it is clear from the inner cycle that premature termination of the cycle must not occur. Thus, we will perform:

- one comparison for each iteration of the inner cycle, i.e., $j = i + 1, \ldots, n - 1$
- the outer cycle iterates $n - 1$ times

Thus, we perform:

Uniqueness of elements in an array (cont.)

- one comparison for each iteration of the inner cycle, i.e., $j = i + 1, \ldots, n - 1$
- outer cycle runs $n - 1$ times

Uniqueness of elements in an array (cont.)

$$= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i \qquad \text{by (3)}$$

$$= (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \qquad \text{by (4) and (8)}$$

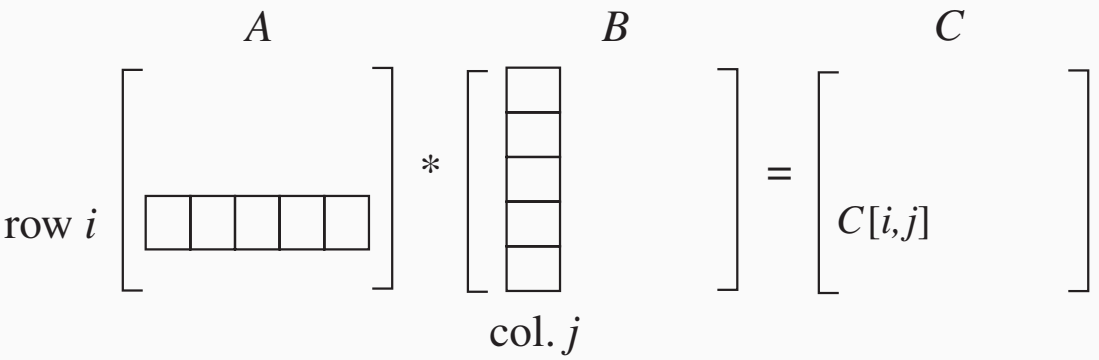
$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} \qquad \text{by (6)}$$

$$= \frac{1}{2} n(n-1) \approx \frac{1}{2} n^2 \in \Theta(n^2)$$

Multiplication of Square Matrices

Our task is to perform an analysis of the algorithm for computing the product $C = AB$ of two square matrices A and B of order n.

By definition, the elements of the matrix are equal to the scalar products of the rows of matrix A with the columns of matrix B.



$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} b_{k,j}$$

for all

$0 \leq i, j \leq n - 1$

Multiplication of Square Matrices (cont.)

$$C[i,j] = A[i,0] \times B[0,j] + \cdots + A[i,k] \times B[k,j] + \cdots + A[i,n-1] \times B[n-1,j]$$

Input : Two square matrices A and B of order n
Output: Square matrix C of order n

```
1 for each element c_{i,j} of matrix C do
2 |   c_{i,j} = 0;
3 |   for k from 0 to n – 1 do
4 |   |   c_{i,j} = c_{i,j} + a_{i,k} × b_{k,j};
5 |   end
6 end
```

1. The algorithm must compute $n \times n$ elements of matrix C

Multiplication of Square Matrices (cont.)

2. Each element of matrix C is computed as the scalar product of the i-th row of matrix A and the j-th column of matrix B
3. The rows and columns have n elements that must be multiplied
4. Therefore, there are a total of $n^2 \times n = n^3$ multiplications

Multiplication of Square Matrices (cont.)

Informal Procedure

1. The algorithm must compute $n \times n$ elements of matrix C
2. Each element of matrix C is computed as the scalar product of the i-th row of matrix A and the j-th column of matrix B
3. The rows and columns have n elements that must be multiplied
4. Therefore, there are a total of $n^2 \times n = n^3$ multiplications

Multiplication of Square Matrices (cont.)

The running time of the algorithm on a specific computer

$$T(n) \approx c_m M(n) = c_m n^3$$

if we also count additions

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3,$$

where c_m and c_a are the times required for multiplication and addition, respectively, and $A(n)$ is the number of additions, which satisfies $A(n) = M(n)$.

Multiplication of Square Matrices (cont.)

Summary

The running time of the algorithm may vary depending on the specific computer, but **the order of complexity of the algorithm (n^3) remains the same**.

Number of bits in the binary representation of a number

Our task is to analyze the algorithm that for a given natural number n calculates the number of bits necessary for writing the number n in binary.

Input : Natural number n
Output: Number of bits in the binary representation of the number n

```
1 count ← 1;
2 while n > 1 do
3 |   count ← count + 1;
4 |   n ← ⌊n/2⌋;
5 end
6 return count;
```

Number of bits in the binary representation of a number (cont.)

- Input size – one number?
- Basic operation – addition, division, comparison with 1?
- Most importantly, in this case, we need to determine the number of loop iterations. The number of comparisons is one more than the number of loop iterations.
- The value of the number n decreases by half with each loop iteration, leading to the relation

$$\lfloor \log_2 n \rfloor + 1$$

  and which corresponds to the relation (2).
- To derive this, we will need to be able to solve recursive equations...

Sources for Independent Study

- Book [2], chapter 2.3, pages 61 – 70

Fundamentals of the Analysis of Algorithm Efficiency

Analysis of Recursive Algorithms

Calculation of Factorial

Our task is to analyze the recursive algorithm that calculates the factorial $n!$ for a given natural number n.

$$n! = \begin{cases} 1 & \text{for } n = 0 \\ n(n-1)! & \text{otherwise} \end{cases}$$

```
1 Function F(n)
  |   Input: Natural number n
  |   Result: Result
2 |   if n = 0 then
3 |   |   return 1;
4 |   end
5 |   else
6 |   |   return n · F(n − 1);
7 |   end
8 end
```

Calculation of Factorial (cont.)

- The size of the input is n.
- We need to find a function $M(n)$ that represents the number of multiplications performed by the algorithm.
- The algorithm has a recursive structure, so we can write a recurrence relation for $M(n)$.

Remark

To solve the recurrence relation, we need to find an explicit expression for $M(n)$. We will use the method of backward substitution to solve the recurrence relation.

- The recurrence relation is $M(n) = M(n-1) + 1$ for $n > 0$.

Calculation of Factorial (cont.)

- We need to find an initial condition to make the recurrence relation unique.
- From the algorithm, we can see that when $n = 0$, no multiplications are performed, so $M(0) = 0$.
- Therefore, the complete recurrence relation is

$$\begin{aligned} M(n) &= M(n-1) + 1 \text{ for } n > 0 \\ M(0) &= 0 \end{aligned}$$

Calculation of Factorial (cont.)

- We will solve the recurrence relation using backward substitution. Substituting $M(n-1) = M(n-2) + 1$ into $M(n) = M(n-1) + 1$, we get

$$M(n) = [M(n-2) + 1] + 1 = M(n-2) + 2$$

  Substituting $M(n-2) = M(n-3) + 1$ into the previous equation, we get

$$M(n) = [M(n-3) + 1] + 2 = M(n-3) + 3.$$

Calculation of Factorial (cont.)

We can see a pattern emerging: $M(n) = M(n - i) + i$. Using this formula, we can find an explicit expression for $M(n)$ by setting $i = n$, which gives

$$M(n) = M(0) + n = 0 + n = \boxed{n}.$$

Calculation of Factorial (cont.)

Summary

1. The result $M(n) = n$ was more or less expected.
2. An iterative algorithm performs the same number of multiplications as a recursive algorithm, without the overhead of function calls.
3. However, the approach used to solve the recurrence relation is important and can be applied to other problems.

General procedure for determining the time complexity of recursive algorithms

1. Selection of a parameter, or parameters, representing the size of the input n.
2. Identification of the basic operations of the algorithm.
3. Does the number of basic operations depend only on the size of the input? If it depends on something else as well, we must examine the worst, best, and average cases separately.
4. Construction of a recursive relation and suitable initial conditions, expressing the number of executions of basic operations.

General procedure for determining the time complexity of recursive algorithms (cont.)

5. Simplification of the constructed relations and, at least, determination of the order of growth.

Resources for Independent Study

- Book [2], chapter 2.4, pages 70 – 79

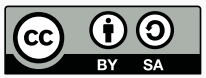
Thanks for your attention

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Brute Force and Exhaustive Search

Jiří Dvorský, Ph.D.

Department of Computer Science
VSB – Technical University of Ostrava



Brute Force Strategy

Characteristics

Brute force is a straightforward approach to solving a problem, usually **directly based on the problem definition** and definitions of the concepts involved.

Brute Force Strategy – examples

Exponentiation problem

Compute an for a nonzero number a and a nonnegative integer n. By the definition of exponentiation

$$a^n = \underbrace{a \times a \times \cdots \times a}_{n \text{ times}}$$

Brute force solution – simply computing an by multiplying 1 by a n times.

More examples

- the consecutive integer checking algorithm for computing GCD, and
- the definition-based algorithm for matrix multiplication from previous presentation.

Brute Force Strategy

1. general strategy – it is difficult to find a problem where "doesn't work",
2. does not generally lead to efficient algorithms, but for some problems, e.g., matrix multiplication, pattern matching, algorithms based on this strategy are applicable to larger inputs,
3. is an acceptable strategy when it is not worthwhile to deal with more sophisticated algorithms – ad hoc problem solving,
4. it's always a useful strategy for solving problems with small input sizes, and
5. it's also important as a benchmark against which to compare more efficient algorithms solving the same problem.

Brute Force and Exhaustive Search

Sorting Algorithms

Sorting Algorithms

- We are given an array of n elements for which an ordering relation is defined (i.e. the relation “less than”), for example let’s take integers.
- The task is to rearrange the elements of the array into a non-decreasing sequence – the element of the array at the lower index must be less than or equal to the element at the higher index.
- The question is: is there a sorting algorithm that solves the problem in a brute force, completely straightforward way?

SelectSort – Initial consideration

Question

Which element in the array do we know exactly where it belongs?

Answer

The smallest! It belongs at the beginning of the array, at the lowest index! (Note: The same applies to the largest element.)

SelectSort – Algorithm Principle

1. Select the smallest element of the n array elements and replace it with the first element of the array.
2. Select the smallest element from the remaining $n - 1$ elements of the array and replace it with the second element of the array.
3. In general, in the i-th step, select the smallest element from the remaining $n - i$ elements and replace it with the i-th element.
4. After $n - 1$ steps, the array is sorted.

SelectSort – Example

89	45	68	90	29	34	17
17	45	68	90	29	34	89
17	29	68	90	45	34	89
17	29	34	90	45	68	89
17	29	34	45	90	68	89
17	29	34	45	68	90	89
17	29	34	45	68	89	90

SelectSort – pseudocode

Input : Array $A[0 \ldots n-1]$ with ordering defined on array elements
Output: Sorted array A

```
1 for i ← 0 to n − 2 do
2 |   min ← i;
3 |   for j ← i + 1 to n − 1 do
4 |   |   if A[j] < A[min] then
5 |   |   |   min ← j;
6 |   |   end
7 |   end
8 |   Swap (A[i], A[min]);
9 end
```

Exchanging values of two variables

```
1 Procedure Swap(x, y)
  |   Input  : Parameters x and y of the same data type
  |   Output: Interchanged values of x and y
2 |   aux ← x;
3 |   x ← y;
4 |   y ← aux;
5 end
```

SelectSort – Analysis

1. Input size – number of elements n.
2. Basic operations – element comparison (sometimes number of element swaps).
3. The number of basic operations depends only on n – worst, best and average cases merge.
4. Constructing equations

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1]$$

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{1}{2} n(n-1) \approx \frac{1}{2} n^2 = \Theta(n^2)$$

SelectSort – Analysis (cont.)

Remarks

- The detailed calculation of the sum can be found in the example “Uniqueness of elements” in the previous lesson.
- The complexity of the algorithm does not depend on the unorderedness of the input array. So the algorithm is **not natural**.
- But the algorithm is **stable** – the first of several identical elements is always taken as the minimum.
- The algorithm is **in situ** – only a constant amount of extra memory is needed.

Animation

An animation of the SelectSort algorithm is available in a separate file.

Brute Force and Exhaustive Search

Sequential search

Sequential search

- Typical example of a brute force strategy solution.
- Strong side of the algorithm – simplicity.
- Weak side of the algorithm – high complexity.

Input : Array $A[0 \ldots n-1]$ and searched element x
Output: Index of the first occurrence of element x in array A, otherwise -1

```
1 for i ← 0 to n – 1 do
2 |   if A[i] = x then
3 |   |   return i;
4 |   end
5 end
6 return -1;
```

The algorithm is also referred to as **linear search**.

Linear search – complexity

	Number of comparisons
Worst-case scenario	n
Best-case scenario	1
Average-case scenario	$\frac{1}{2}p(n+1) + n(1-p)$

where p is the probability of successful search

Linear search – using a sentinel

Input : Array $A[0 \ldots n]$ and searched element x
Output: Index of the first occurrence of element x in array A, otherwise -1

```
1 A[n] ← x;
2 i ← 0;
3 while A[i] ≠ x do
4 |   i ← i + 1;
5 end
6 if i < n then return i;
7 return -1;
```

Brute Force and Exhaustive Search

Brute force string matching

Brute force string matching

Task

Find the pattern p in the text t.

Brute Force Solution

1. Attach the pattern to the beginning of the text.
2. Start comparing characters in the pattern and the text.
3. If all characters of the pattern match the text – found.
4. If we find a mismatch, move the pattern one position forward and continue with point 2

$$\begin{array}{ccccccccc} t_0 & \cdots & t_i & \cdots & t_{i+j} & \cdots & t_{i+m-1} & \cdots & t_{n-1} \\ & & \updownarrow & & \updownarrow & & \updownarrow & & \\ & & p_0 & \cdots & p_j & \cdots & p_{m-1} & & \end{array}$$

Brute force search – pseudocode

Input : Pattern p, text t and starting position s
Output: Position of the first occurrence of p in text t or -1

```
1 for i ← s to |t| - |p| do
2 |   j ← 0;
3 |   while j < |p| do
4 |   |   if p[j] ≠ t[i + j] then break;
5 |   |   j ← j + 1;
6 |   end
7 |   if j = |p| then return i;
8 end
9 return -1;
```

Brute force search – example

First attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
1	2	3	4																				
G	C	A	G	A	G	A	G																

Shift by 1 character

Second attempt

G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
	1																						
	G	C	A	G	A	G	A	G															

Shift by 1 character

Brute force search – example (cont.)

Third attempt

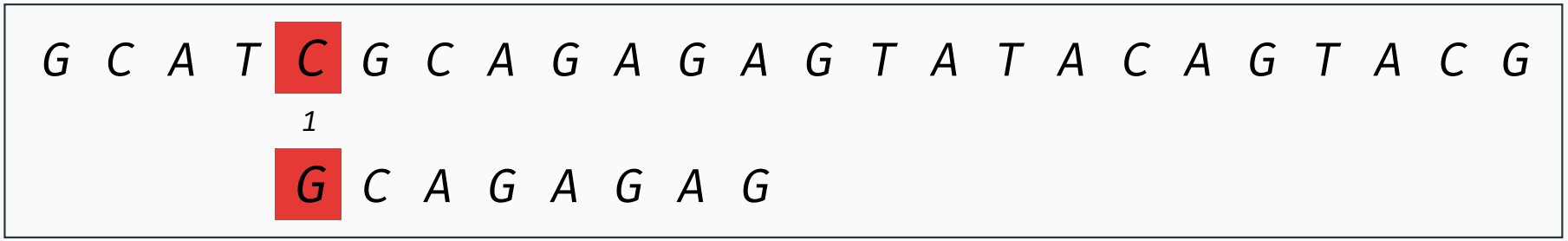
Shift by 1 character

Fourth attempt

Shift by 1 character

Brute force search – example (cont.)

Fifth attempt



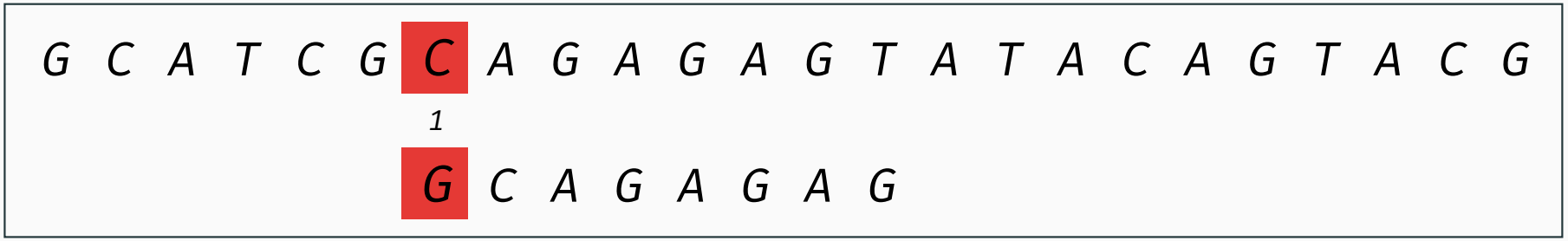
Shift by 1 character

Sixth attempt

Shift by 1 character

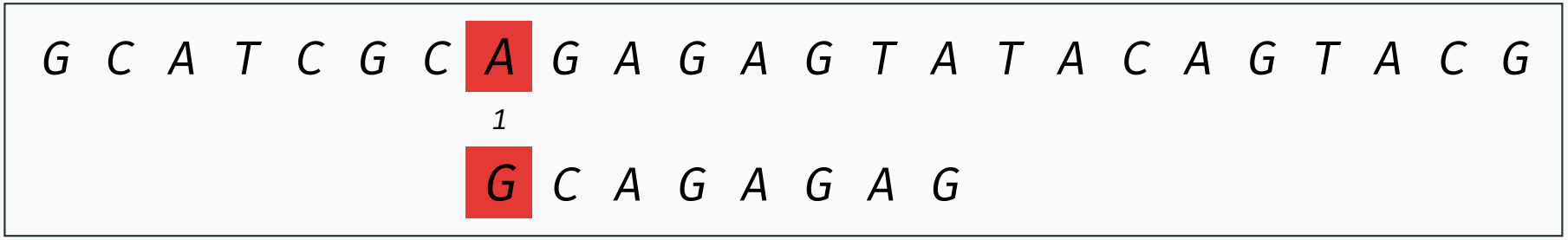
Brute force search – example (cont.)

Seventh attempt



Shift by 1 character

Eighth attempt



Shift by 1 character

Brute force search – example (cont.)

Ninth attempt

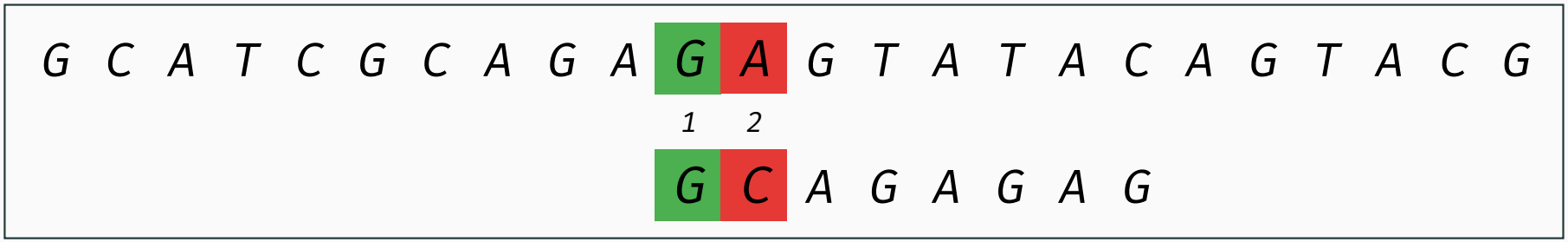
Shift by 1 character

Tenth attempt

Shift by 1 character

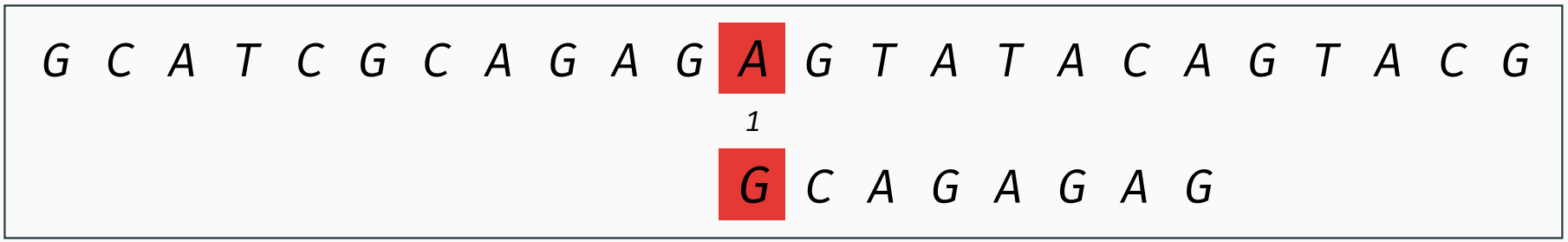
Brute force search – example (cont.)

Eleventh attempt



Shift by 1 character

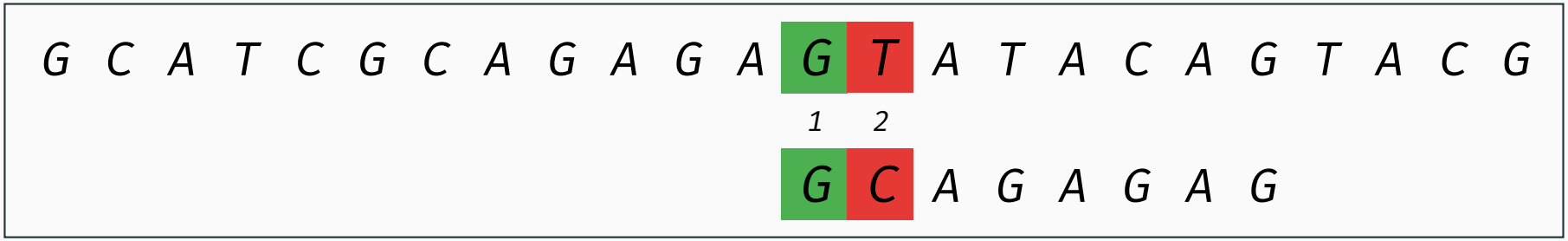
Twelfth attempt



Shift by 1 character

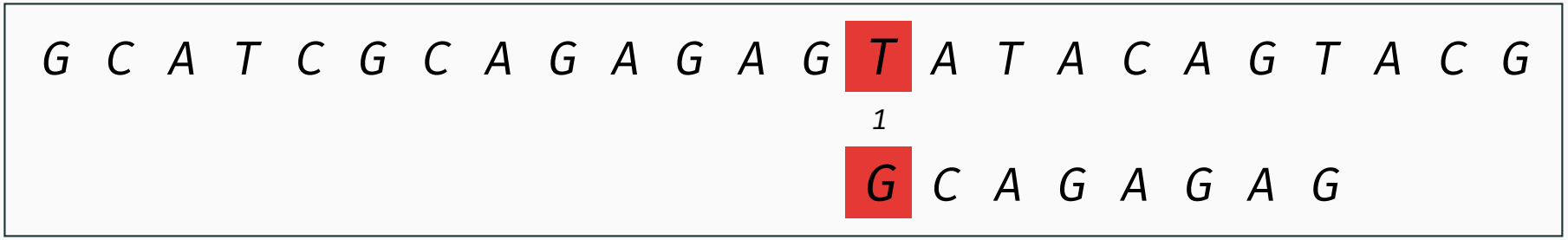
Brute force search – example (cont.)

Thirteenth attempt



Shift by 1 character

Fourteenth attempt



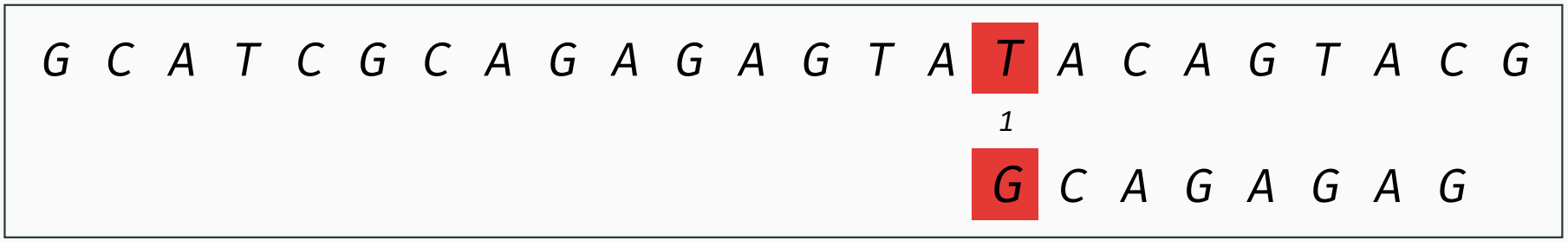
Shift by 1 character

Brute force search – example (cont.)

Fifteenth attempt

Shift by 1 character

Sixteenth attempt



Shift by 1 character

Brute force search – example (cont.)

Seventeenth attempt

Shift by 1 character

The algorithm performed a total of 30 character comparisons.

Brute force string matching – algorithm complexity

- Input size – length of the text n and length of the sample m.
- Basic operation – comparison of the sample character and the text character.
- Dependency only on input size – no, it also depends on when the first mismatch is found.
- Worst-case scenario – text $a^{n-1}b$, sample $a^{m-1}b$
  - in each attempt we perform all m comparisons of the sample with the text
  - at the same time, we make all $n - m + 1$ attempts.
  - In total, we perform $m(n - m + 1)$ comparisons, the algorithm falls into $O(mn)$.
- Best-case scenario – sample is found at the beginning of the text, complexity $O(m)$.
- Natural languages – shift occurs after several (k_L) comparisons, worst-case complexity $O(k, n) = O(n)$.

Brute Force and Exhaustive Search

Closest pair problem

Closest pair problem

Problem definition

Find two mutually closest points from a set of n points.

- This is one of the problems of **computational geometry**.
- Points can lie in a plane or generally in some multidimensional space.
- Points can represent real-world objects or records in a database, texts...
- Examples of applications:
  - Air traffic control – we are looking for the two closest aircraft in airspace.
  - Clustering – hierarchical clustering algorithms gradually merge the closest clusters into one larger cluster.

Closest pair problem – assumptions

Let's assume a set of n points $\{P_1, \ldots, P_n\}$, to each point P_i corresponds a vector $\vec{p}_i$ with components

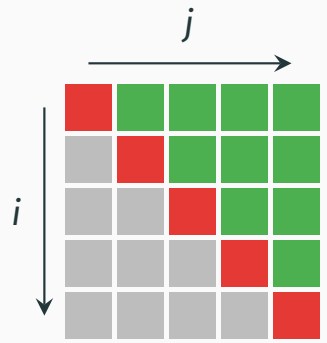
$$\vec{p}_i = (x_i, y_i)$$

in the usual Cartesian coordinates.

The distance of points $\vec{p}_i$ and $\vec{p}_j$ will be calculated using the Euclidean distance

$$d(\vec{p}_i, \vec{p}_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Closest pair problem – solution by brute force



- Calculate the distance of all pairs of points $\vec{p}_i$ and $\vec{p}_j$ and find the minimum.
- It is sufficient to calculate only pairs of points for $j = i + 1, \dots n$.

Input : Set of points $\{\vec{p}_1, \vec{p}_2, \dots, \vec{p}_n\}$
Output: Distance of the two closest points

```
1 Minimum distance ← ∞;
2 for i ← 1 to n − 1 do
3 |   for j ← i + 1 to n do
4 |   |   Minimum distance ←
  |   |     min(Minimum distance, sqrt((x_i − x_j)^2 + (y_i − y_j)^2));
5 |   end
6 end
7 return Minimum distance;
```

Closest pair problem – solution by brute force, complexity

1. Input size – number of points n
2. Basic operation
   - Calculation of square root – not a trivial matter[1].
   - Calculation of square root can be avoided – it is an increasing function, we can look for the minimum of "squares" of distances.
   - We will take the exponentiation of differences of coordinates as the basic operation.
3. Number of basic operations depends only on n – worst, best and average case are the same.

Closest pair problem – solution by brute force, complexity (cont.)

4. Establishment of relationships

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2 = 2 \sum_{i=1}^{n-1} (n - i)$$
$$= 2[(n-1) + (n-2) + \cdots + 1] = (n-1)n \in \Theta(n^2)$$

5. Removal of square root – reduction of complexity by a constant factor, no improvement in asymptotic complexity, still $\Theta(n^2)$ algorithm.
6. Later we will show an algorithm with linear logarithmic complexity.

[1] *https://en.wikipedia.org/wiki/Methods_of_computing_square_roots*

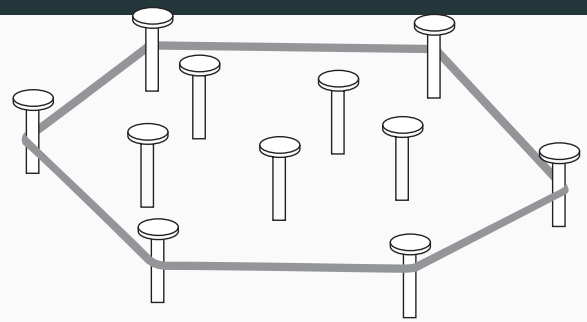
Brute Force and Exhaustive Search

Convex hull of a set

Convex Hull

Problem Statement

The task is to find the convex hull of a set of points in space.



- This is one of the problems of **computational geometry**.
- Points can lie in a plane or generally in some multidimensional space.
- Examples of applications:
  - Collision detection – computer graphics, autonomous vehicles,
  - GIS – point sensors, creating an area from this data,
  - Optimization tasks – the vertices of the convex hull are extreme in some way; a convex polygon is created as the intersection of a finite number of half-spaces; a half-space is defined by an inequality...

Convex set of points

Definition

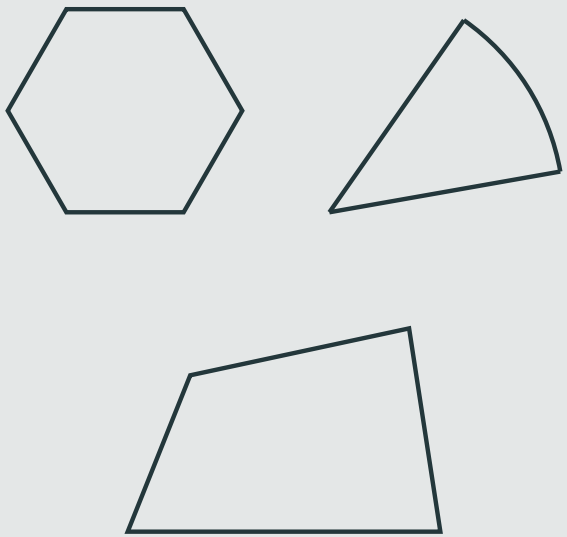
A set of points M in a plane is called **convex**, if for any pair of points $p, q \in M$ the line segment connecting points p and q belongs to the set M.

A set that is not convex is called **non-convex**.

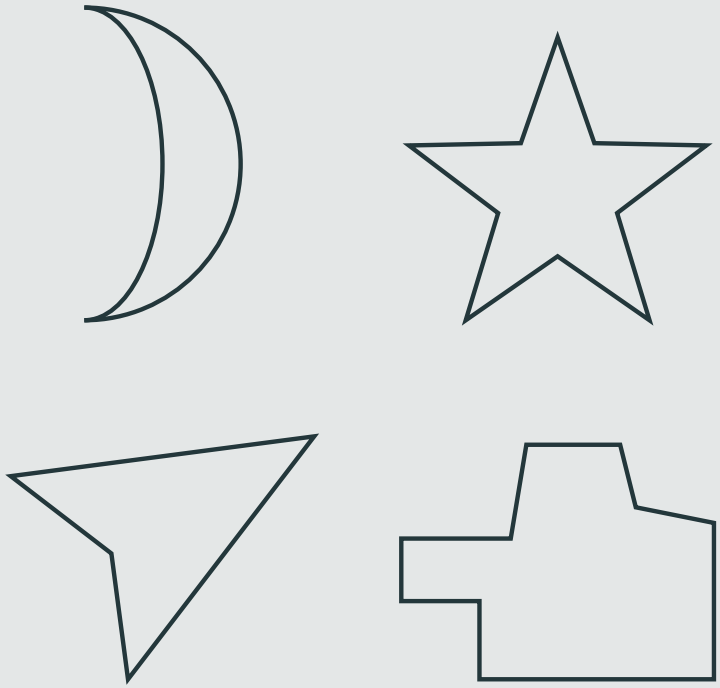
If we imagine the boundary of the set as opaque, the convexity of the set means intuitively that from each of its points every point is visible.

Convex set of points – examples

Convex shapes



Nonconvex shapes



Convex hull

Definition

The convex hull of a set of points M is called the smallest convex set that contains M.

The expression ”smallest” means that the convex hull of the set M must be a subset of any other convex subset containing the set M.

Convex hull

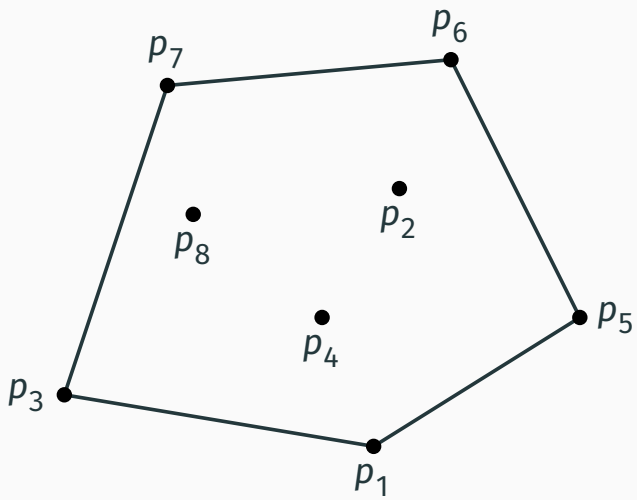
- two-element set – line segment connecting both points.

Convex hull (cont.)

- three-element set – triangle, if the points do not lie on a line, otherwise it is a line segment connecting the most distant points.

Theorem

The convex hull of a set of points M with more than two points that do not lie on one line is a convex polygon, whose vertices belong to M.



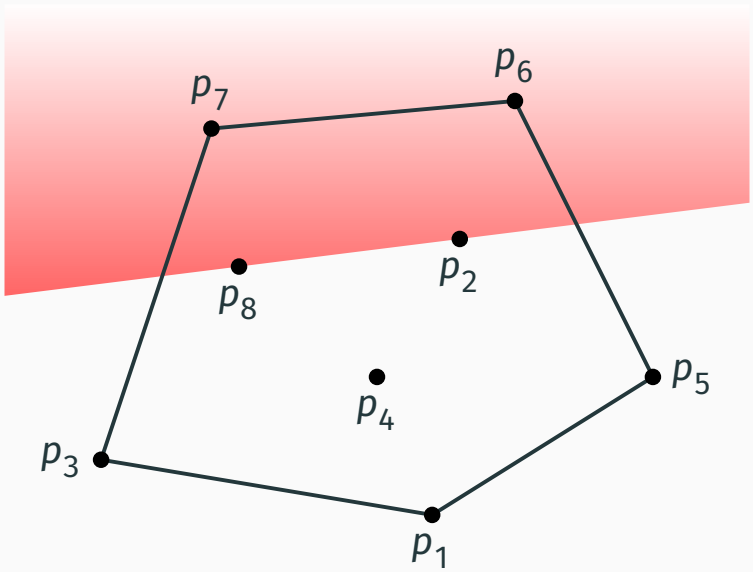
The points of the set M that specify the convex hull M are called **extreme points**.

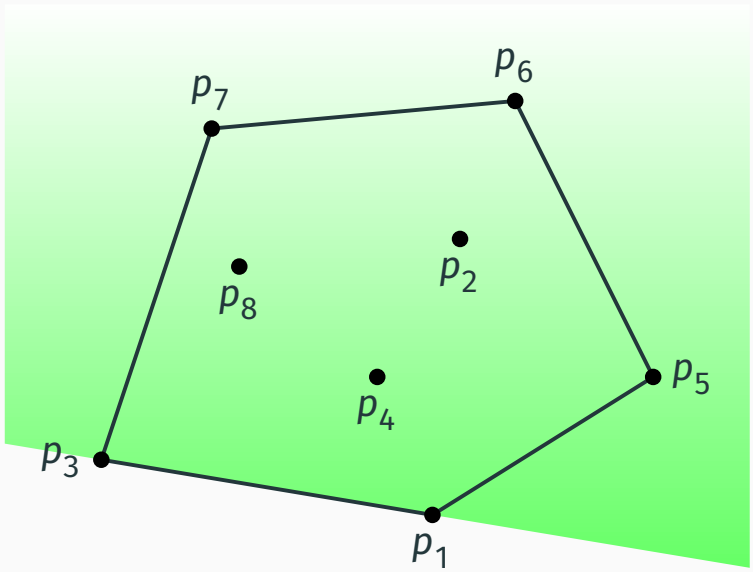
Convex hull (cont.)

The problem of finding the convex hull of a set M is reduced to finding the extreme points.

Convex hull – brute force solution

The line segment $\vec{p}_i\vec{p}_j$ belongs to the convex hull of the set M if and only if all other points from M lie in one of the half-planes defined by the line $\vec{p}_i\vec{p}_j$.





Convex hull – brute force solution (cont.)

The general equation of the line passing through points $\vec{p}_i$ and $\vec{p}_j$ can be written as

$$ax + by + c = 0,$$

where

$$\begin{aligned} a &= y_j - y_i \\ b &= x_i - x_j \\ c &= y_i x_j - x_i y_j \end{aligned}$$

Convex hull – brute force solution (cont.)

The line defines two half-planes:

$$ax + by + c < 0 \tag{12}$$

$$ax + by + c > 0 \tag{13}$$

It is therefore sufficient to verify that for the remaining $n - 2$ points, either inequality (12) or (13) holds.

Convex hull – complexity analysis

- We must check all $\frac{1}{2}n(n-1)$ pairs of points and simultaneously
- for each line defined by one pair of points we must verify the validity of inequalities (??) and (??) for the remaining $n-2$ points.
- Overall, therefore $\left[\frac{1}{2}n(n-1)\right](n-2) \in O(n^3)$.

Brute Force and Exhaustive Search

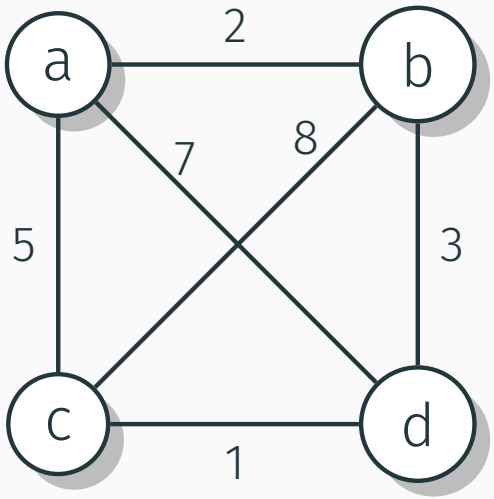
Exhaustive search

Exhaustive search

- Part of solving many problems – finding **one element**, with some **specific property**, from a set of elements, so-called domain, which **exponentially** or faster grows with the problem size.
- The searched element is typically **of combinatorial nature** – permutation, combination, subset.
- Typically it is about **optimization tasks** – typically we search for maximum, minimum. For example, we minimize path length, maximize profit.

Exhaustive search is a problem-solving strategy based on brute force, consisting of testing all elements of the considered domain.

Traveling Salesman Problem – example



Route	Route length l
$a \to b \to c \to d \to a$	$2 + 8 + 1 + 7 = 18$
$a \to b \to d \to c \to a$	$2 + 3 + 1 + 5 = 11$
$a \to c \to b \to d \to a$	$5 + 8 + 3 + 7 = 23$
$a \to c \to d \to b \to a$	$5 + 1 + 3 + 2 = 11$
$a \to d \to b \to c \to a$	$7 + 3 + 8 + 5 = 23$
$a \to d \to c \to b \to a$	$7 + 1 + 8 + 2 = 18$

Exhaustive search

Traveling Salesman Problem

Knapsack problem

Brute Force and Exhaustive Search

Graph traversal

Depth first and breadth first graph traversal

Algorithm for depth-first graph traversal

```
   Input  : Graph G(V, E), initial vertex s ∈ V
   Output: DF-tree
 1 Init(V, s);
 2 while stack ≠ ∅ do
 3 |   u ← Top(stack);
 4 |   switch state [u] do
 5 |   |   case discovered do
 6 |   |   |   ProcessDiscoveredVertex(u);
 7 |   |   end
 8 |   |   case current do
 9 |   |   |   ProcessCurrentVertex(u);
10 |   |   end
11 |   end
12 end
```

Algorithm for depth-first graph traversal (cont.)

```
1  Procedure Init(V, s)
       Input  : Vertices set V, initial vertex s ∈ V
2      foreach v ∈ V do
3          state [v] ← unknown;
4          d [v] ← undefined;
5          f [v] ← undefined;
6          π [v] ← undefined;
7      end
8      stack ← ∅;
9      state [s] ← discovered;
10     Push(stack, s);
11     time ← 0;
12 end
```

Algorithm for depth-first graph traversal (cont.)

```
1  Procedure ProcessDiscoveredVertex(u)
       Input  : Vertex u ∈ V
2      state [u] ← current;
3      d [u] ← time ← time + 1;
4      foreach v ∈ Adj(G, u) do
5          if state [v] = unknown then
6              state [v] ← discovered;
7              π [v] ← u;
8              Push(stack, v);
9          end
10     end
11 end
```

Algorithm for depth-first graph traversal (cont.)

```
1 Procedure ProcessCurrentVertex(u)
  |   Input  : Vertex u ∈ V
2 |   state[u] ← finished;
3 |   f[u] ← time ← time + 1;
4 |   Pop(stack);
5 end
```

Animation of depth-first graph traversal – legend

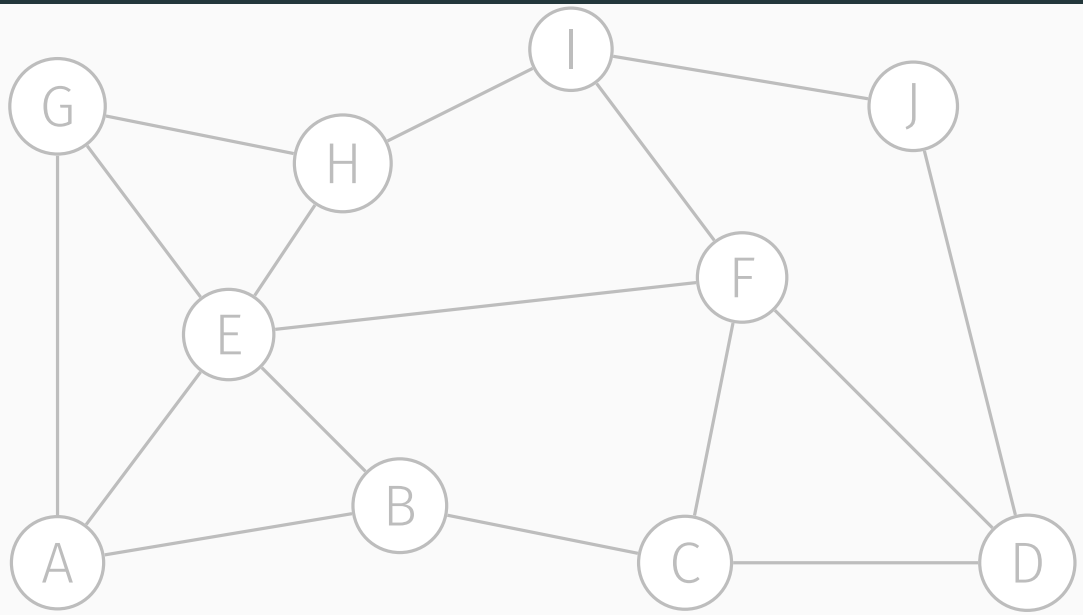
Graph vertices

gray	vertex in unknown state
yellow	vertex in discovered state
red	vertex in current state
blue	vertex in finished state

Graph edges

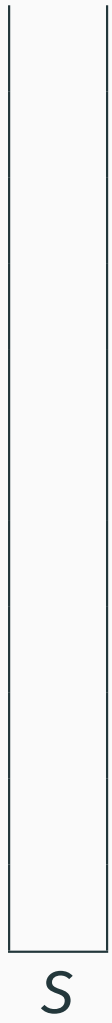
gray	edge between vertices in unknown state or edge not belonging to the DF-tree
yellow	edge incident with vertices in discovered state
red	edge incident with vertex in current state
blue	edge between vertices in finished state

Depth-first graph traversal, step 1

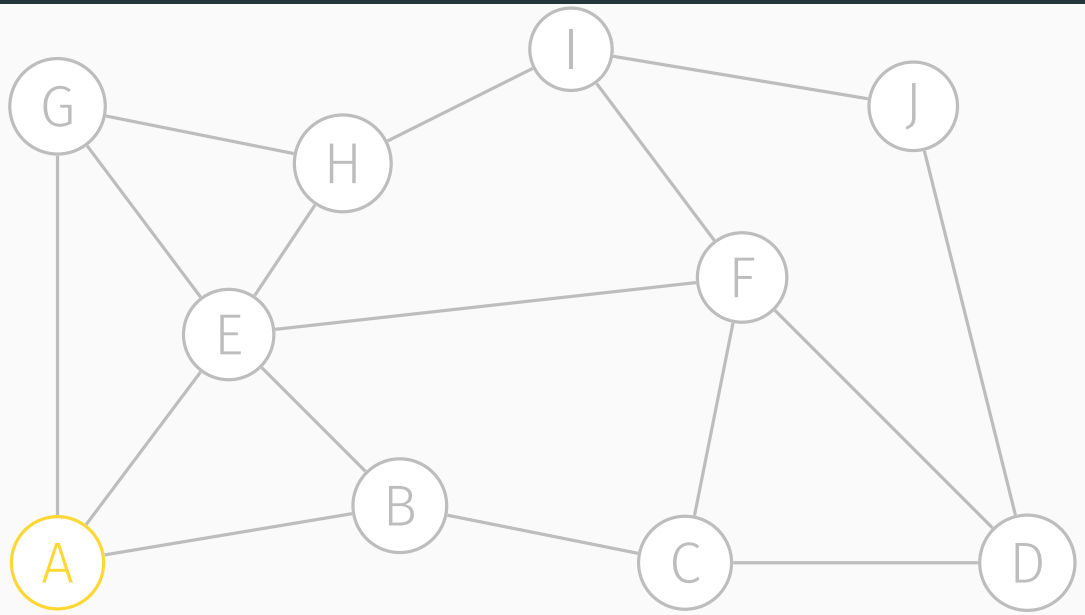


v	$d[v]$	$f[v]$	$\pi[v]$
A	∞	∞	/
B	∞	∞	/
C	∞	∞	/
D	∞	∞	/
E	∞	∞	/

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	∞	∞	/
H	∞	∞	/
I	∞	∞	/
J	∞	∞	/



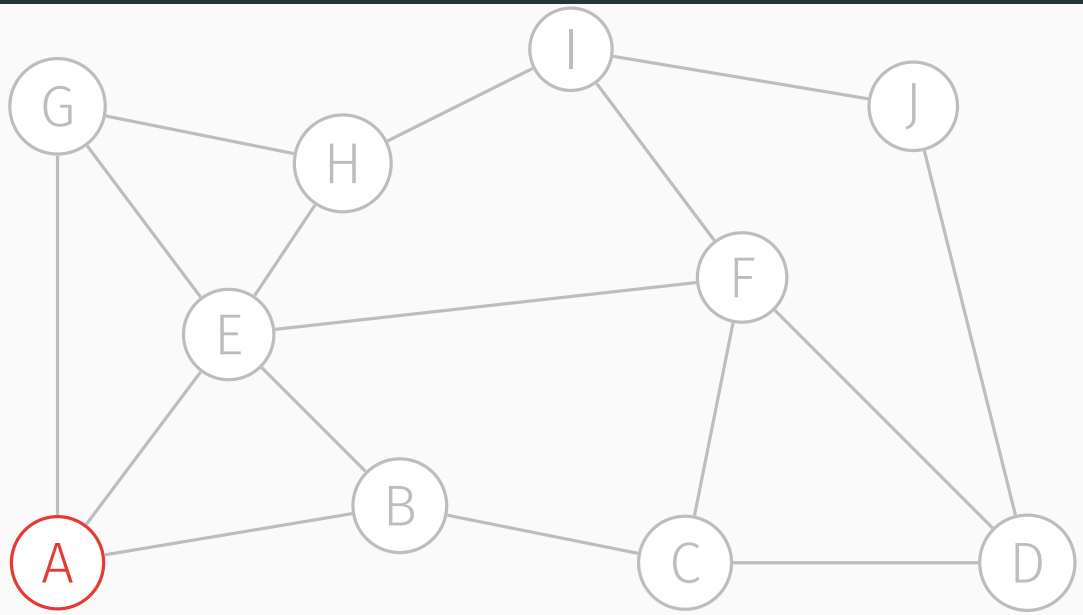
Depth-first graph traversal, step 2



v	$d[v]$	$f[v]$	$\pi[v]$
A	∞	∞	/
B	∞	∞	/
C	∞	∞	/
D	∞	∞	/
E	∞	∞	/

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	∞	∞	/
H	∞	∞	/
I	∞	∞	/
J	∞	∞	/

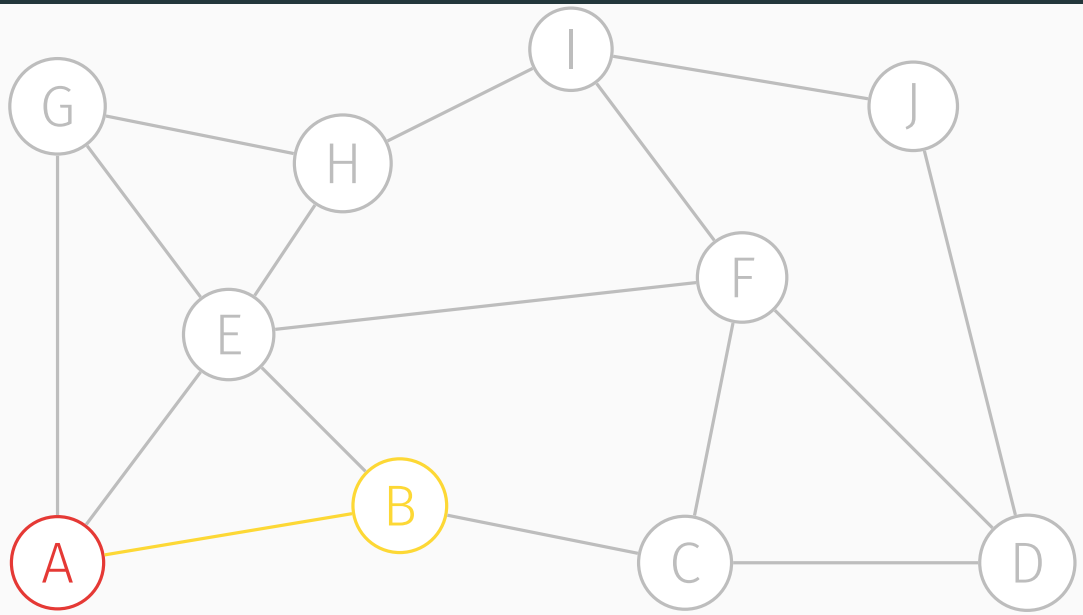
Depth-first graph traversal, step 3



v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	/
C	∞	∞	/
D	∞	∞	/
E	∞	∞	/

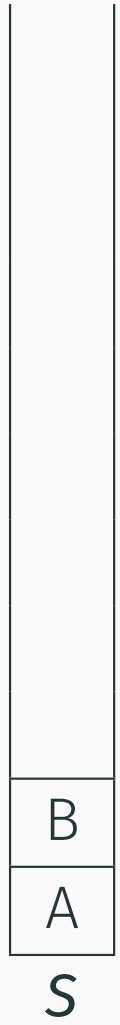
v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	∞	∞	/
H	∞	∞	/
I	∞	∞	/
J	∞	∞	/

Depth-first graph traversal, step 4

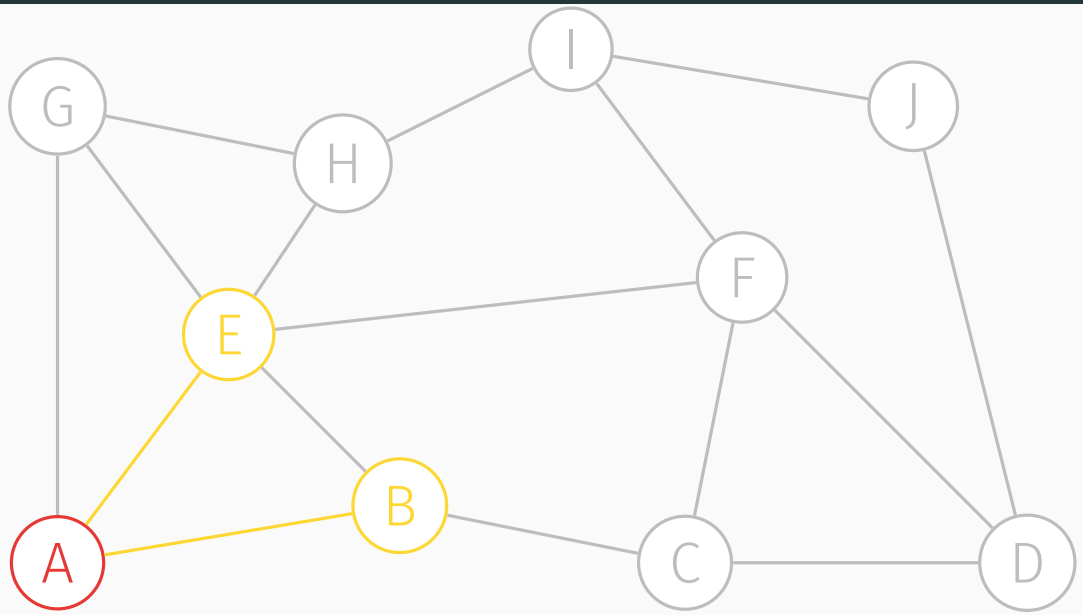


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	/

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	∞	∞	/
H	∞	∞	/
I	∞	∞	/
J	∞	∞	/

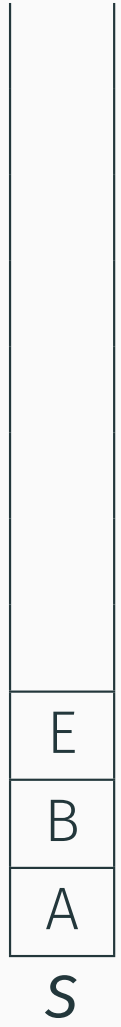


Depth-first graph traversal, step 5

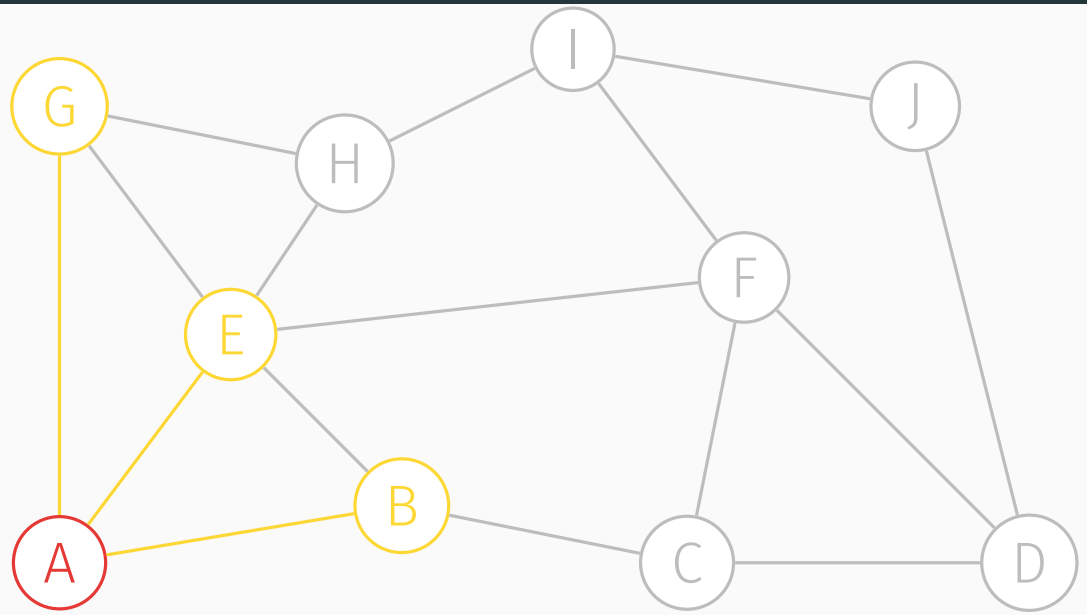


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	∞	∞	/
H	∞	∞	/
I	∞	∞	/
J	∞	∞	/



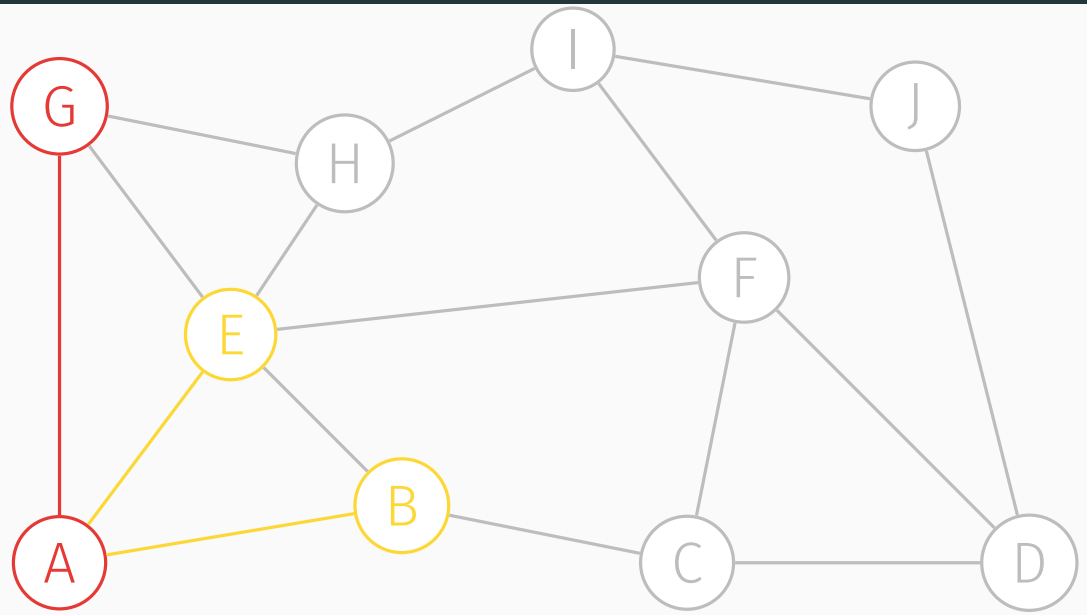
Depth-first graph traversal, step 6



v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	∞	∞	A
H	∞	∞	/
I	∞	∞	/
J	∞	∞	/

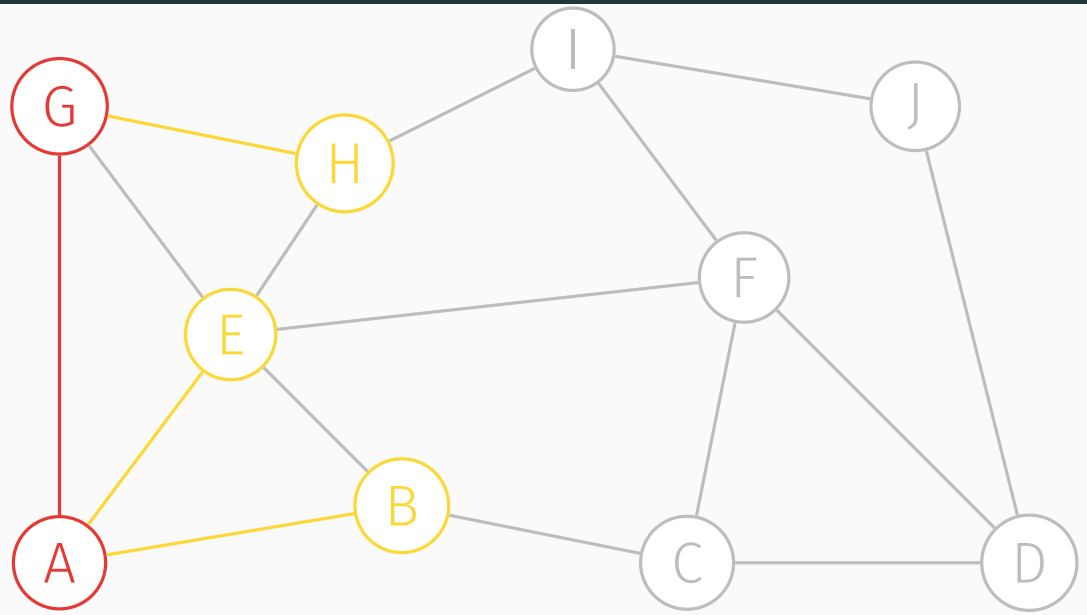
Depth-first graph traversal, step 7



v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

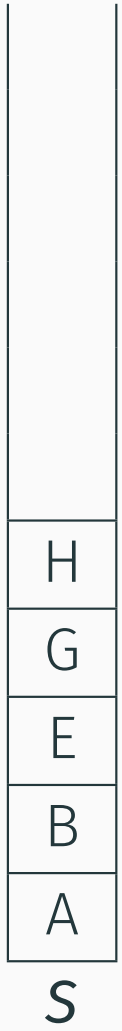
v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	2	∞	A
H	∞	∞	/
I	∞	∞	/
J	∞	∞	/

Depth-first graph traversal, step 8

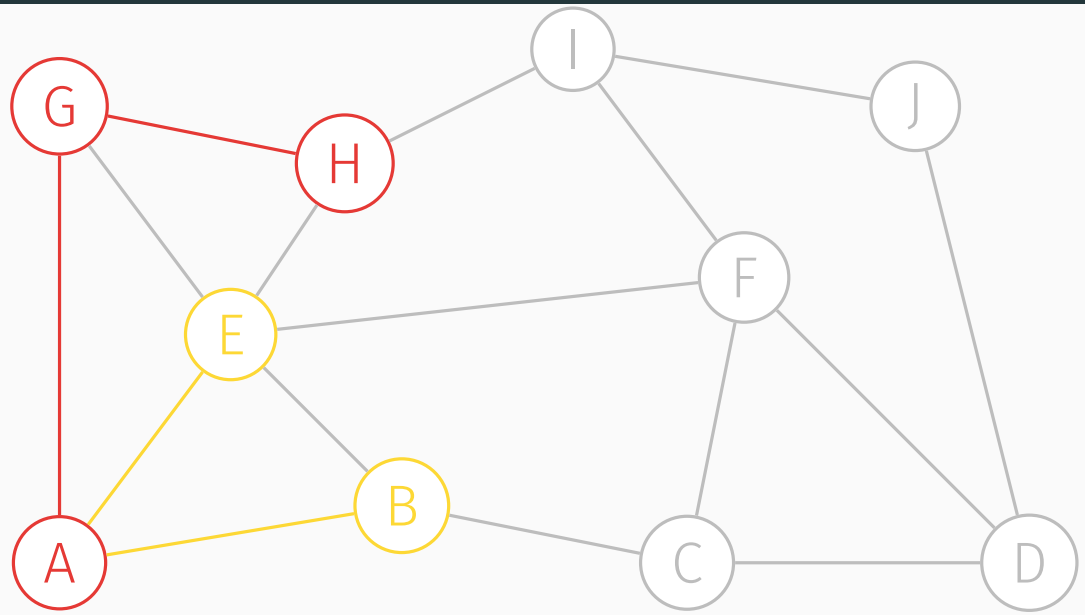


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	2	∞	A
H	∞	∞	G
I	∞	∞	/
J	∞	∞	/

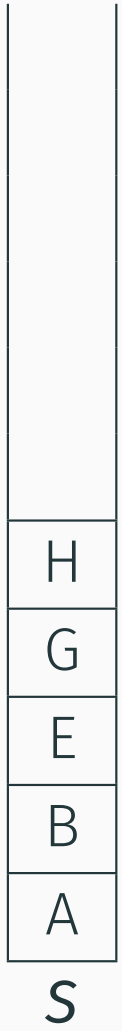


Depth-first graph traversal, step 9

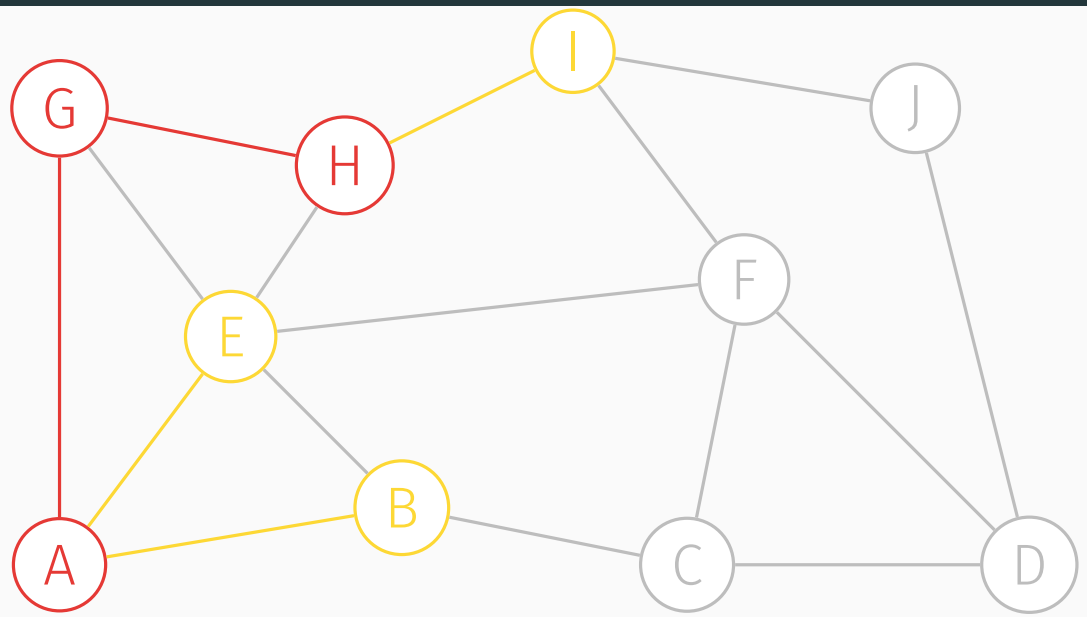


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	2	∞	A
H	3	∞	G
I	∞	∞	/
J	∞	∞	/

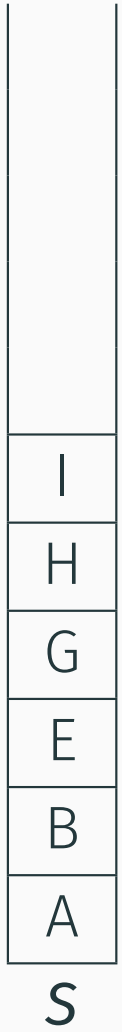


Depth-first graph traversal, step 10

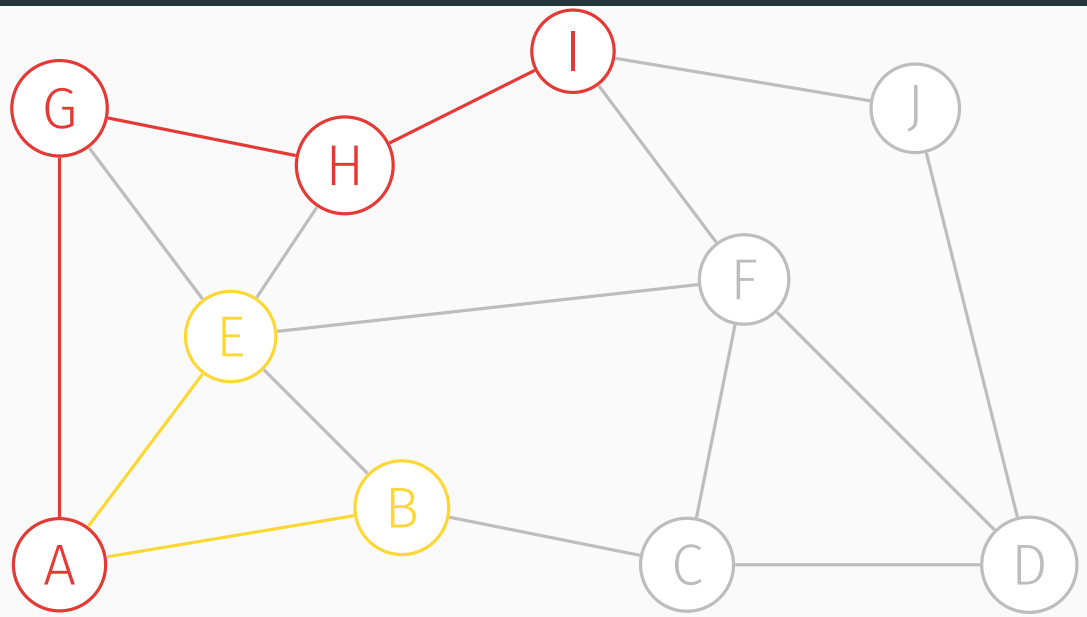


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	2	∞	A
H	3	∞	G
I	∞	∞	H
J	∞	∞	/

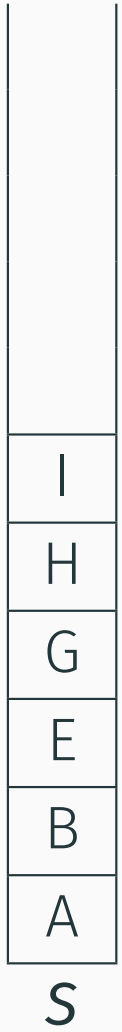


Depth-first graph traversal, step 11

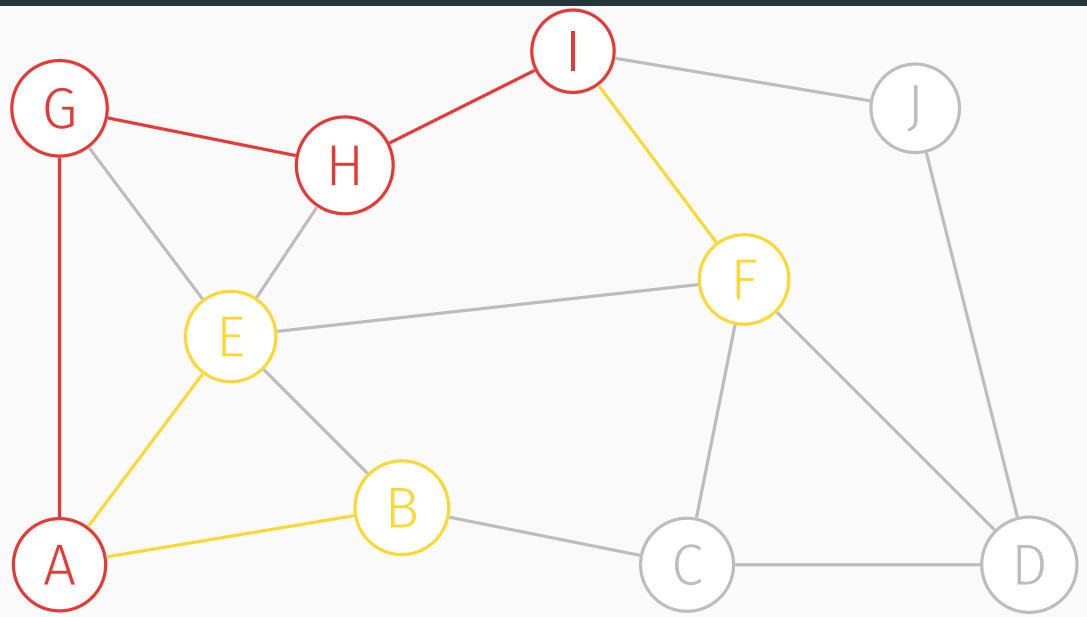


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	/
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	∞	∞	/

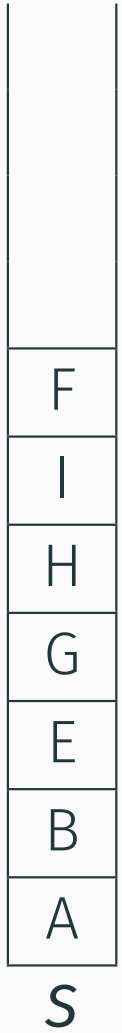


Depth-first graph traversal, step 12

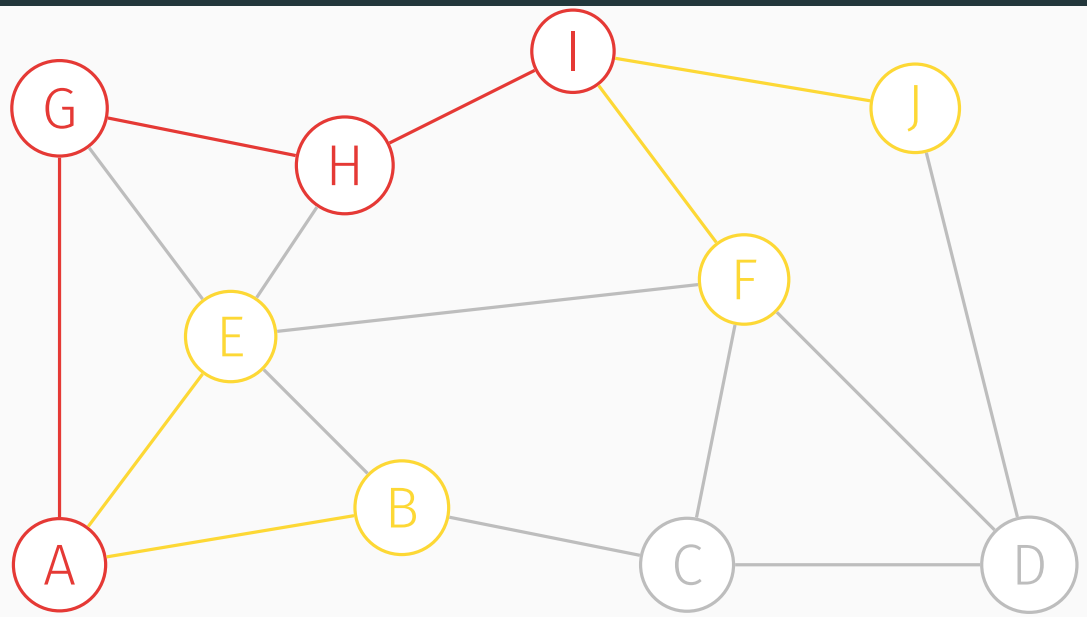


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	∞	∞	/

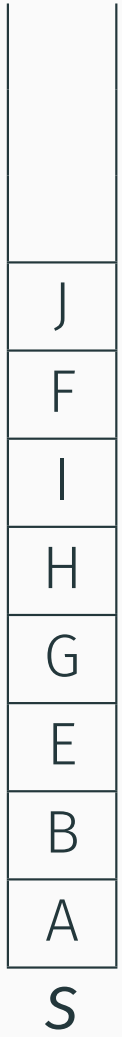


Depth-first graph traversal, step 13

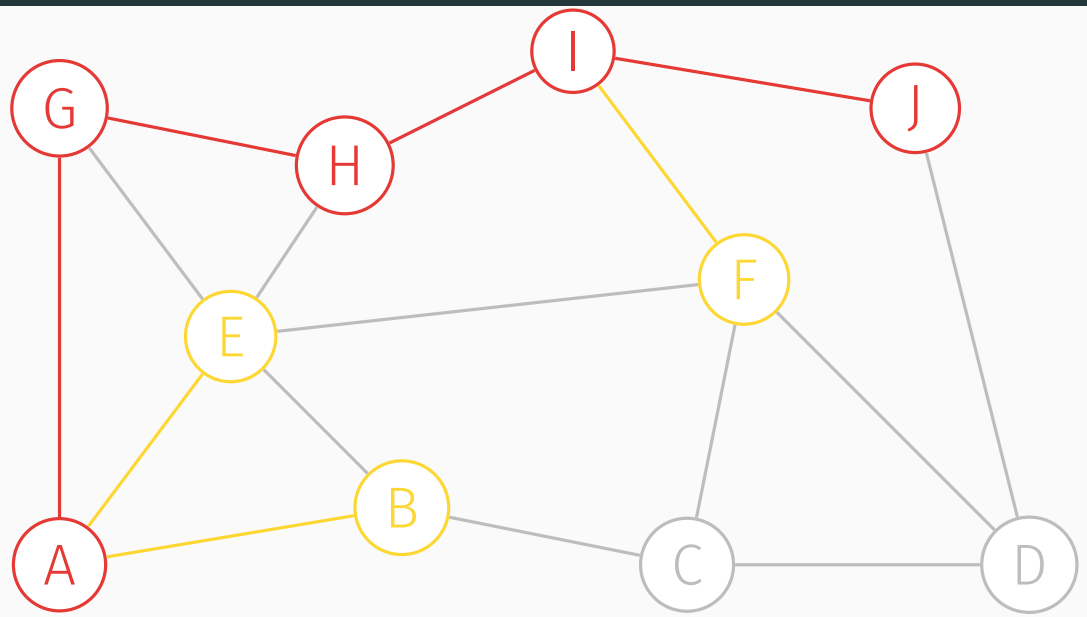


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	∞	∞	I

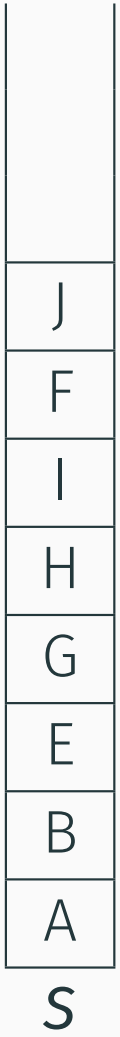


Depth-first graph traversal, step 14

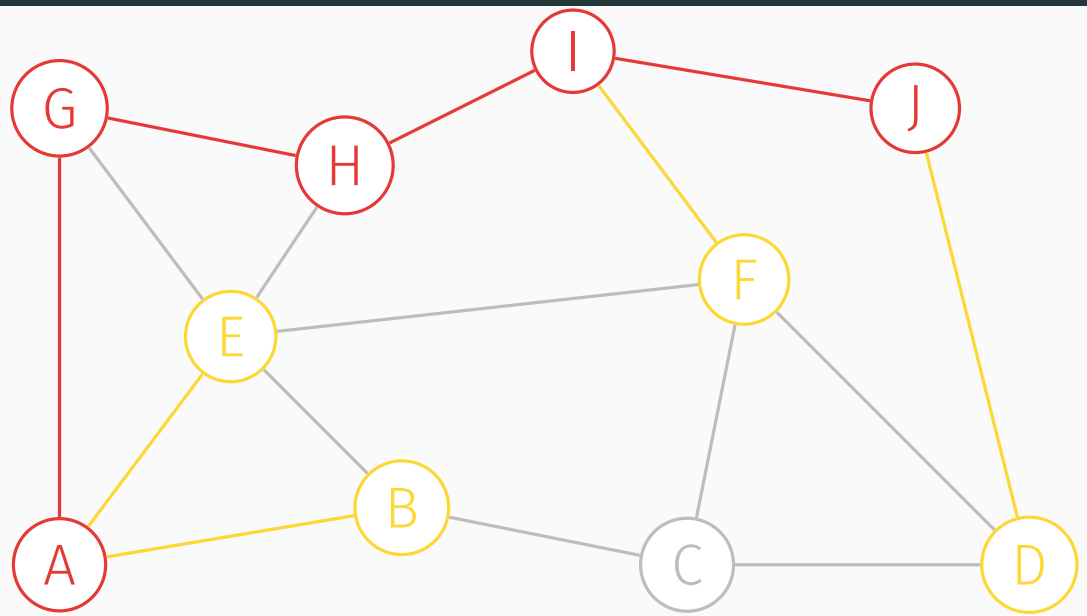


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	/
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	5	∞	I

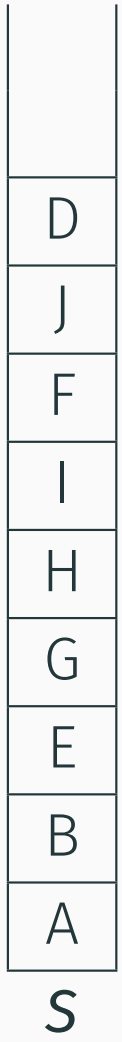


Depth-first graph traversal, step 15

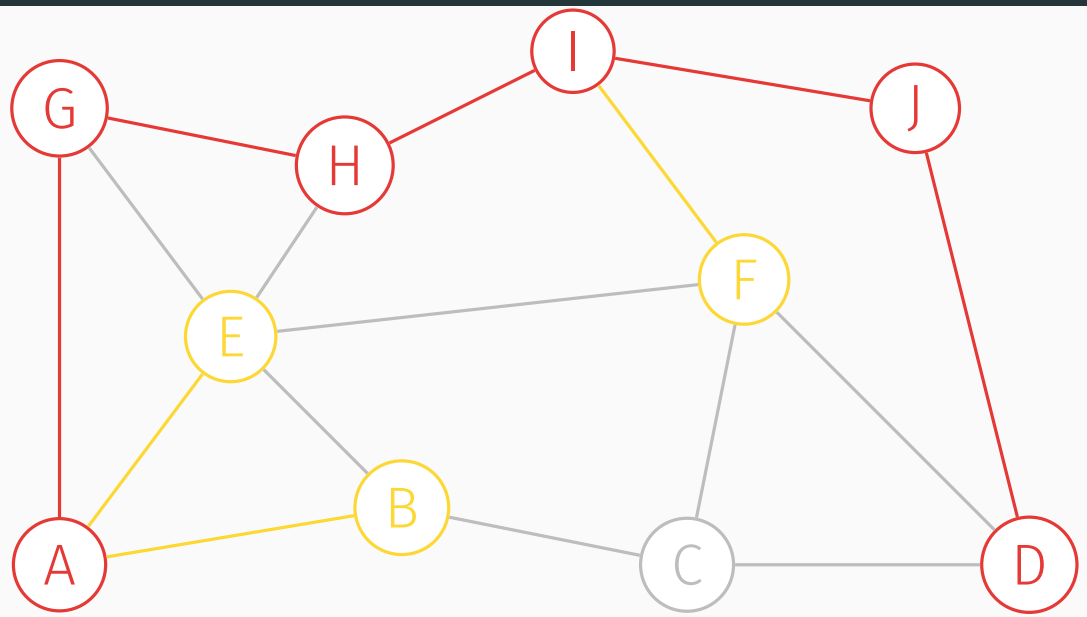


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	∞	∞	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	5	∞	I

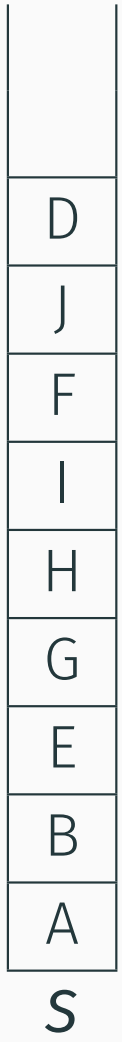


Depth-first graph traversal, step 16

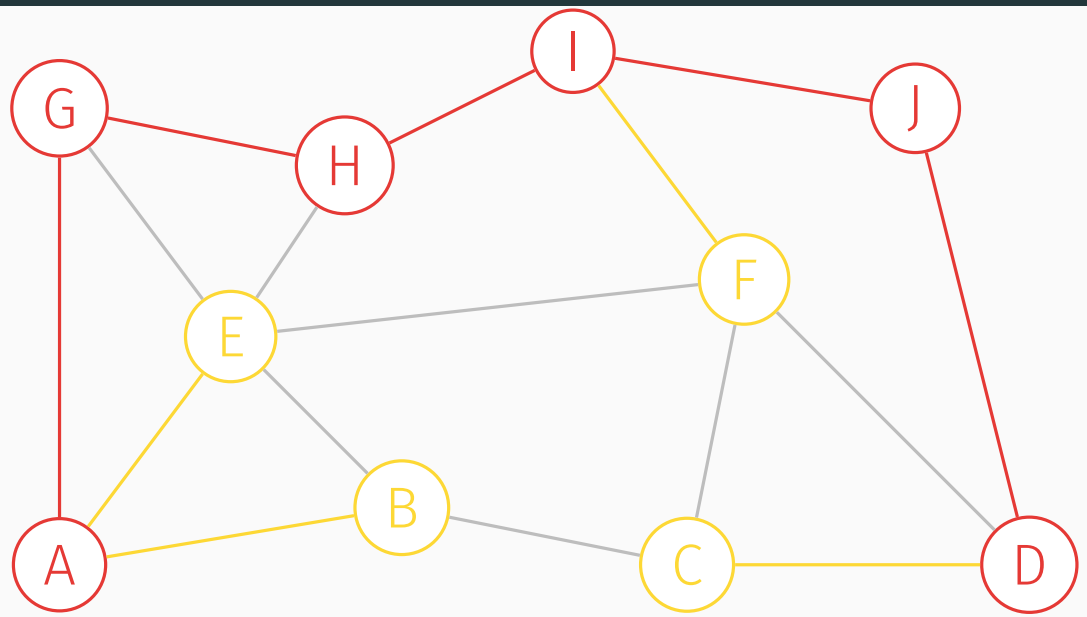


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	/
D	6	∞	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	5	∞	I

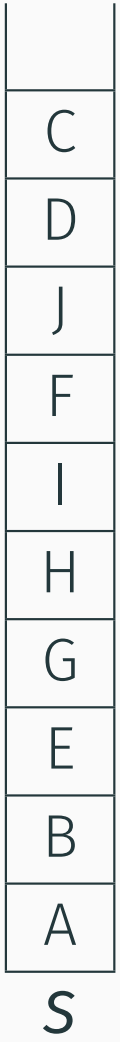


Depth-first graph traversal, step 17

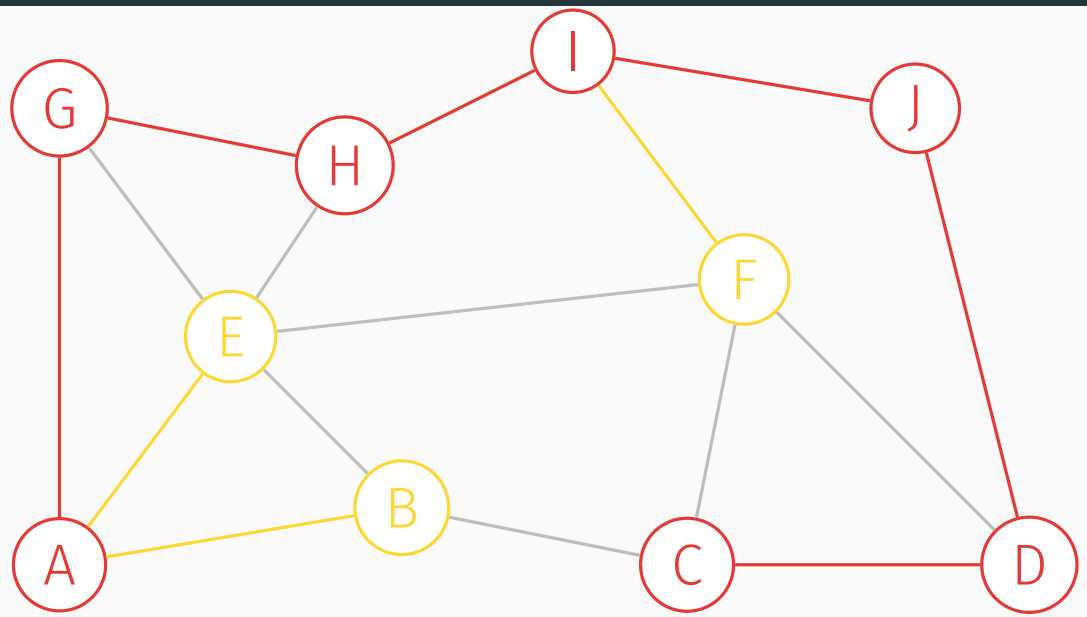


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	∞	∞	D
D	6	∞	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	5	∞	I

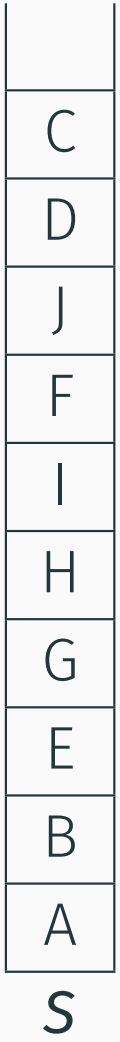


Depth-first graph traversal, step 18

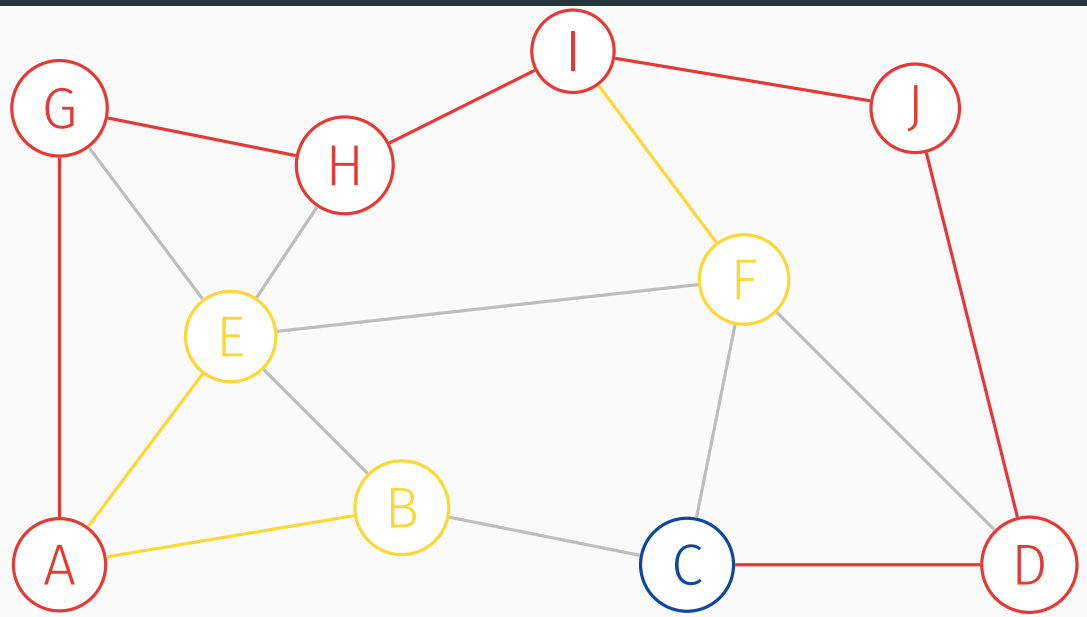


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	∞	D
D	6	∞	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	5	∞	I

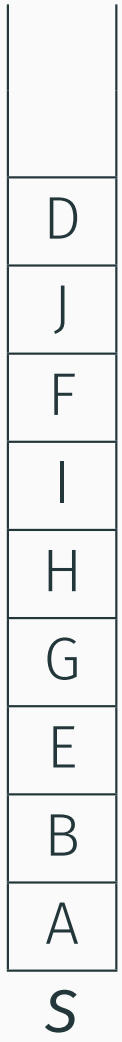


Depth-first graph traversal, step 19

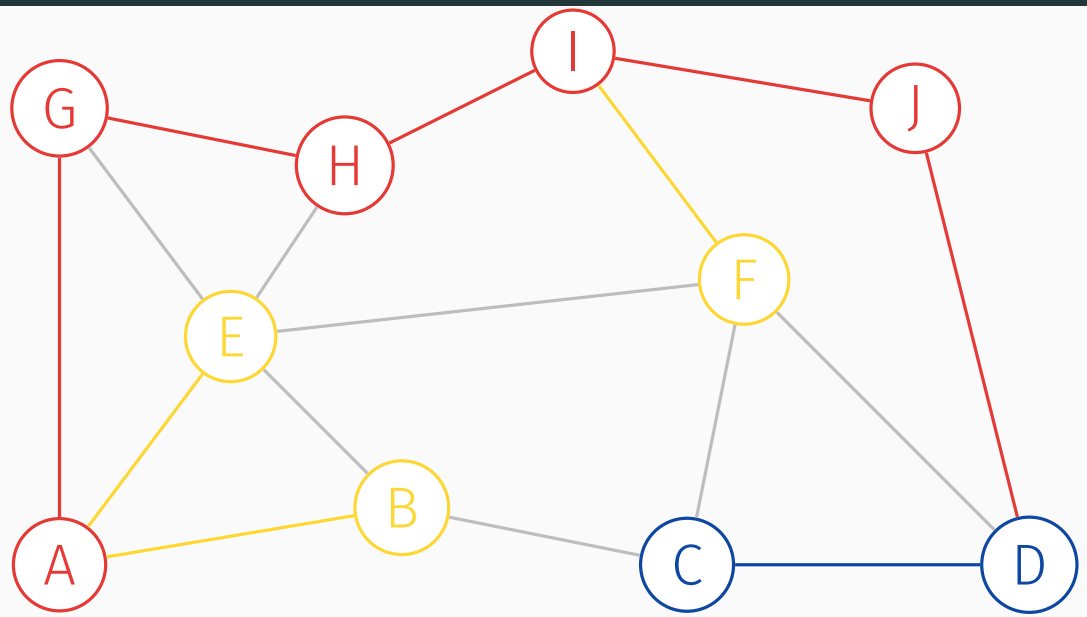


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	∞	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	5	∞	I

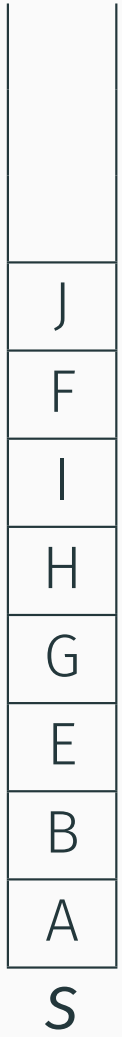


Depth-first graph traversal, step 20

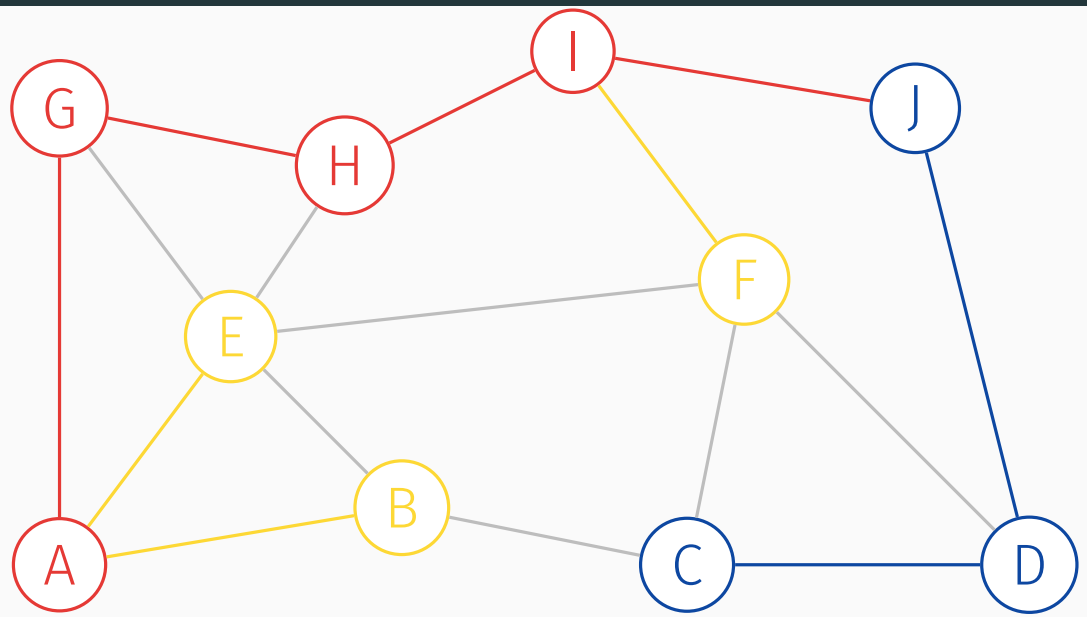


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	5	∞	I

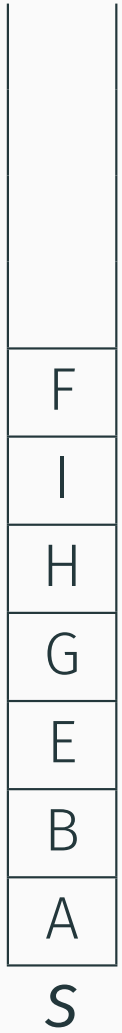


Depth-first graph traversal, step 21

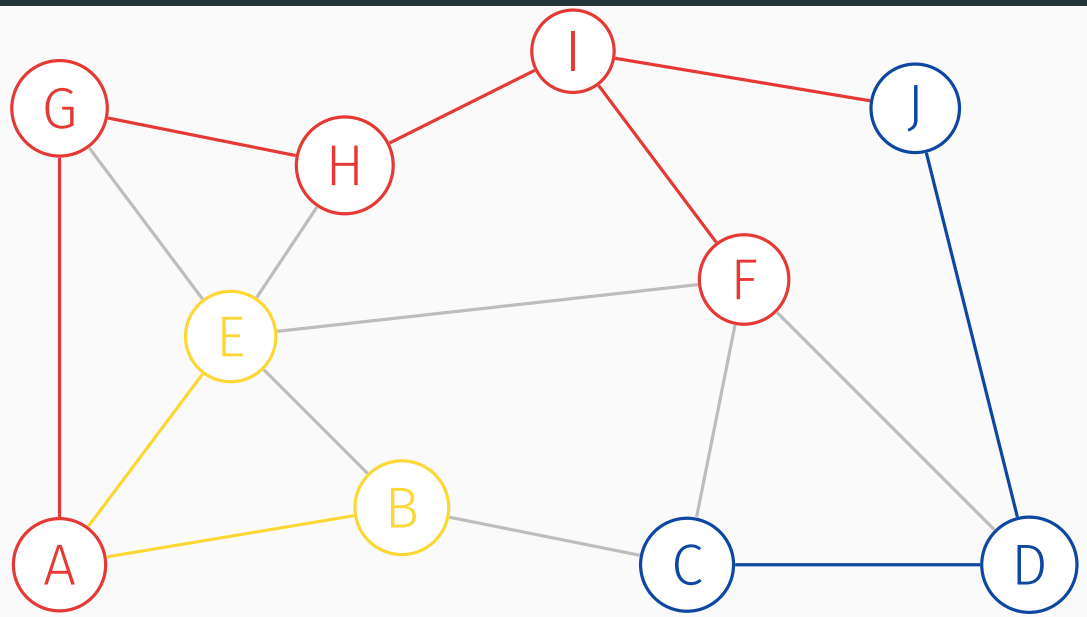


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	∞	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	5	10	I

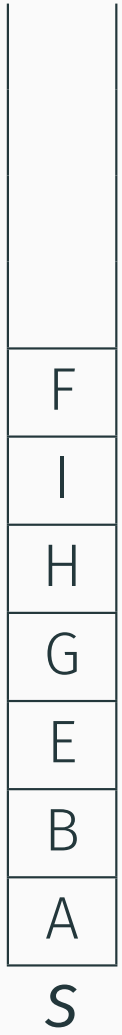


Depth-first graph traversal, step 22

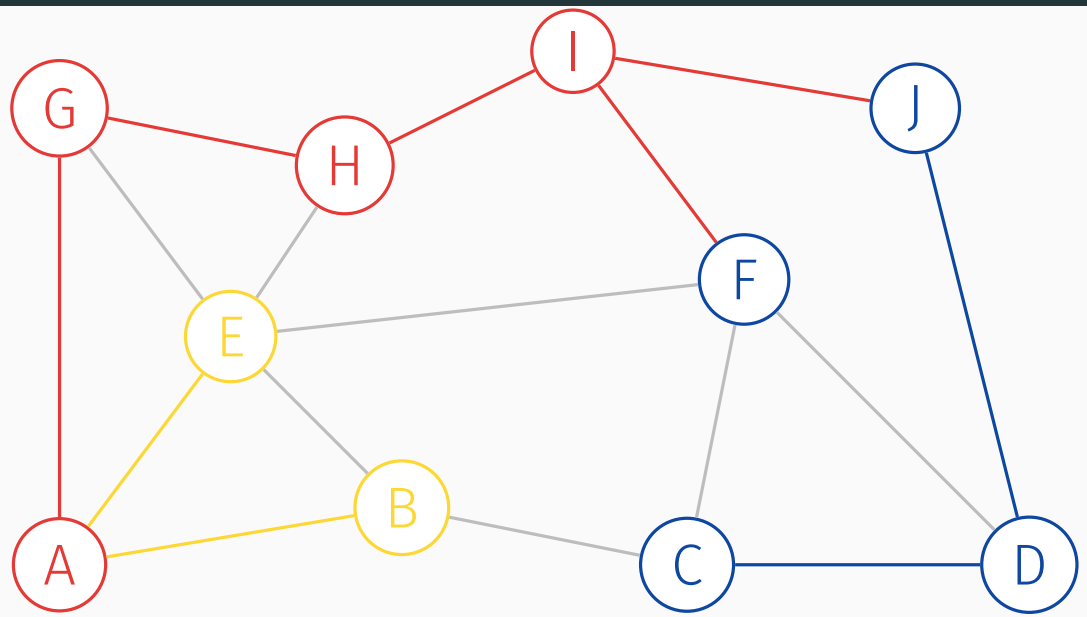


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	11	∞	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	5	10	I

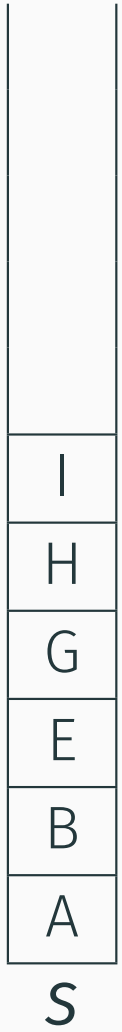


Depth-first graph traversal, step 23

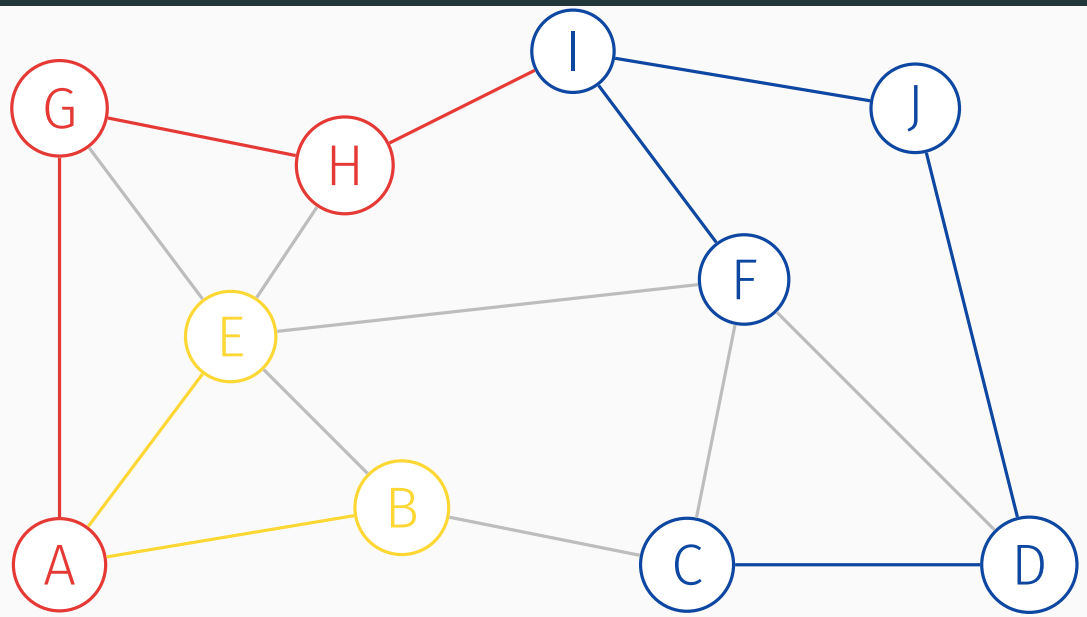


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	11	12	I
G	2	∞	A
H	3	∞	G
I	4	∞	H
J	5	10	I

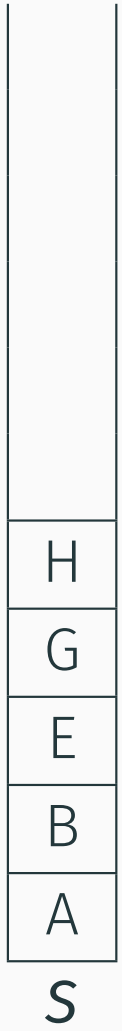


Depth-first graph traversal, step 24

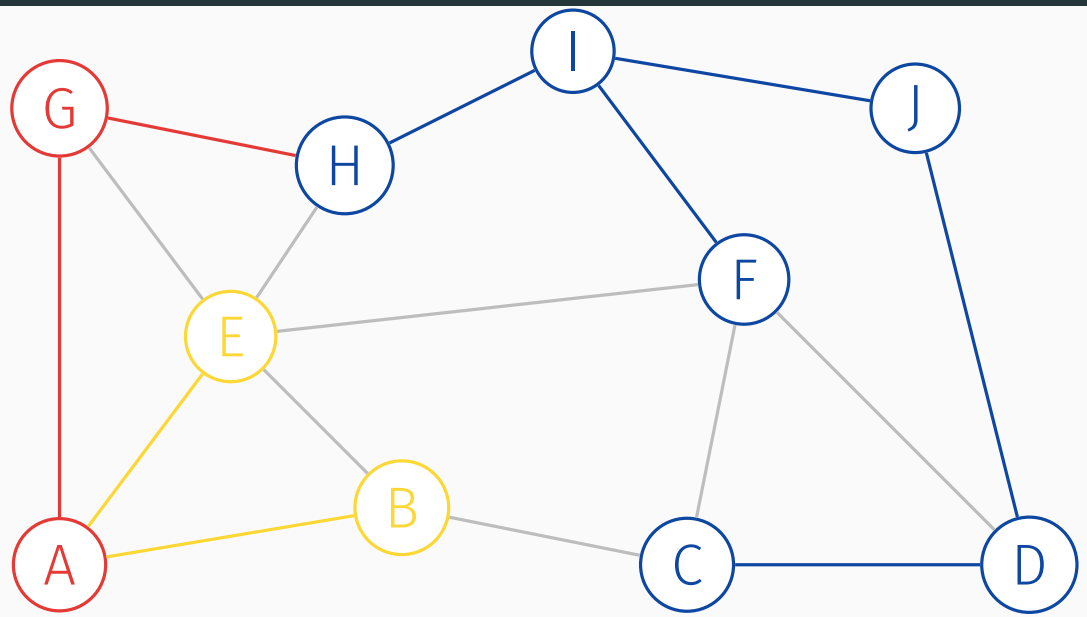


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	11	12	I
G	2	∞	A
H	3	∞	G
I	4	13	H
J	5	10	I



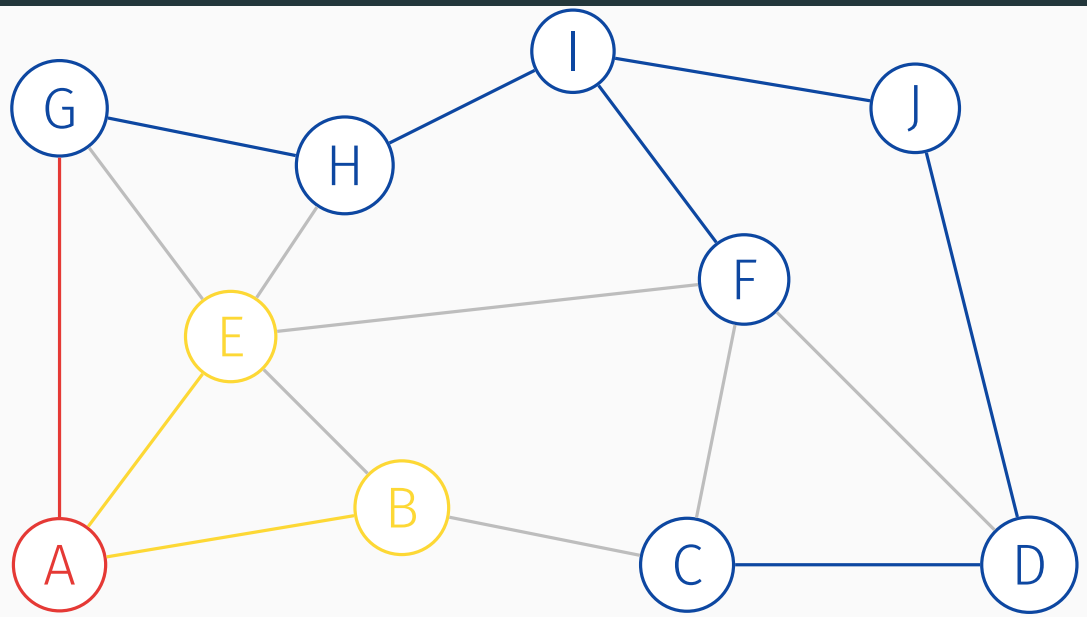
Depth-first graph traversal, step 25



v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

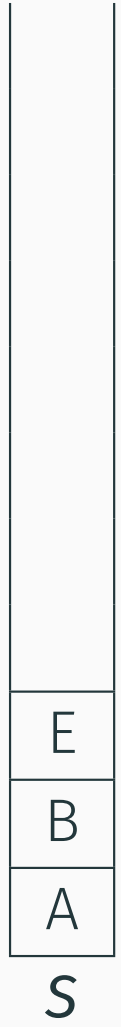
v	$d[v]$	$f[v]$	$\pi[v]$
F	11	12	I
G	2	∞	A
H	3	14	G
I	4	13	H
J	5	10	I

Depth-first graph traversal, step 26

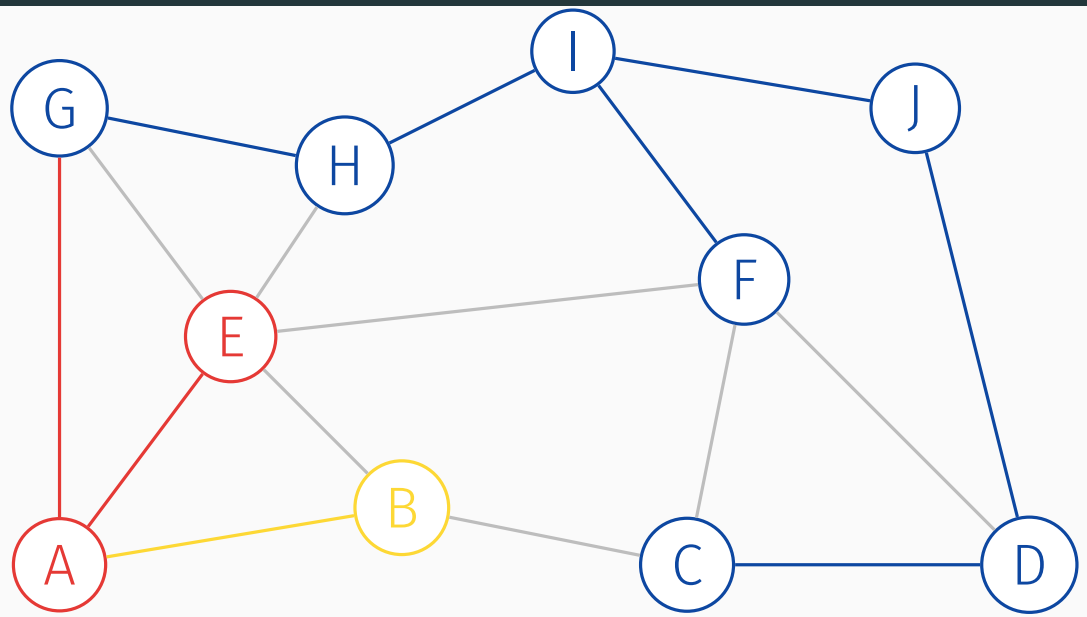


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	∞	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	11	12	I
G	2	15	A
H	3	14	G
I	4	13	H
J	5	10	I

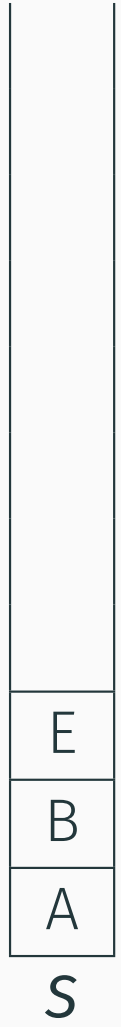


Depth-first graph traversal, step 27

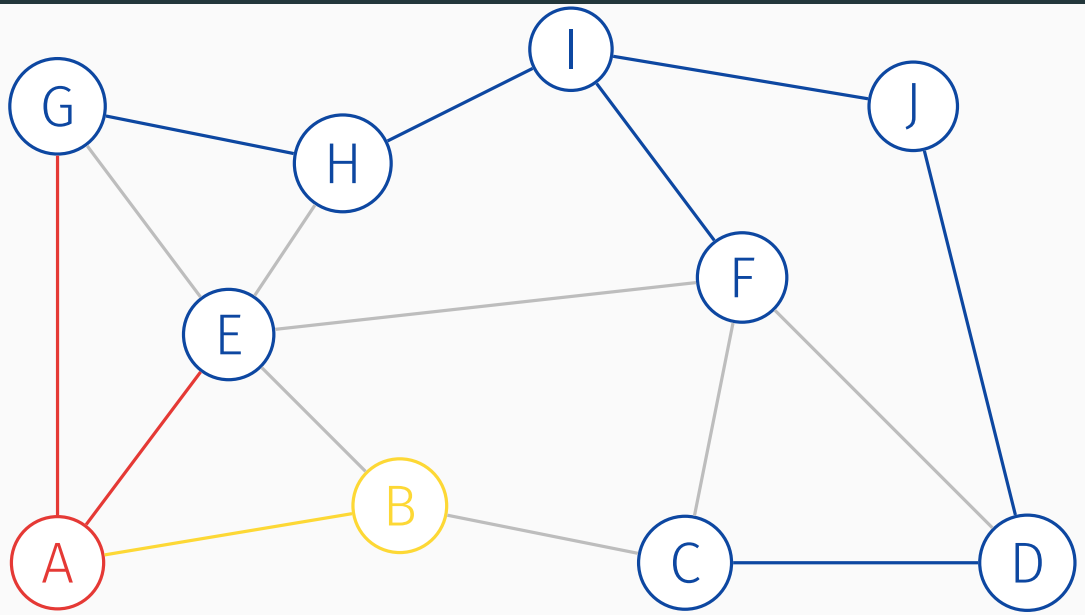


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	16	∞	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	11	12	I
G	2	15	A
H	3	14	G
I	4	13	H
J	5	10	I

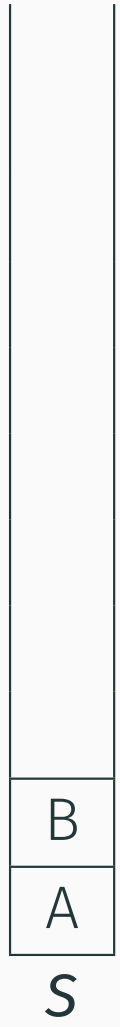


Depth-first graph traversal, step 28

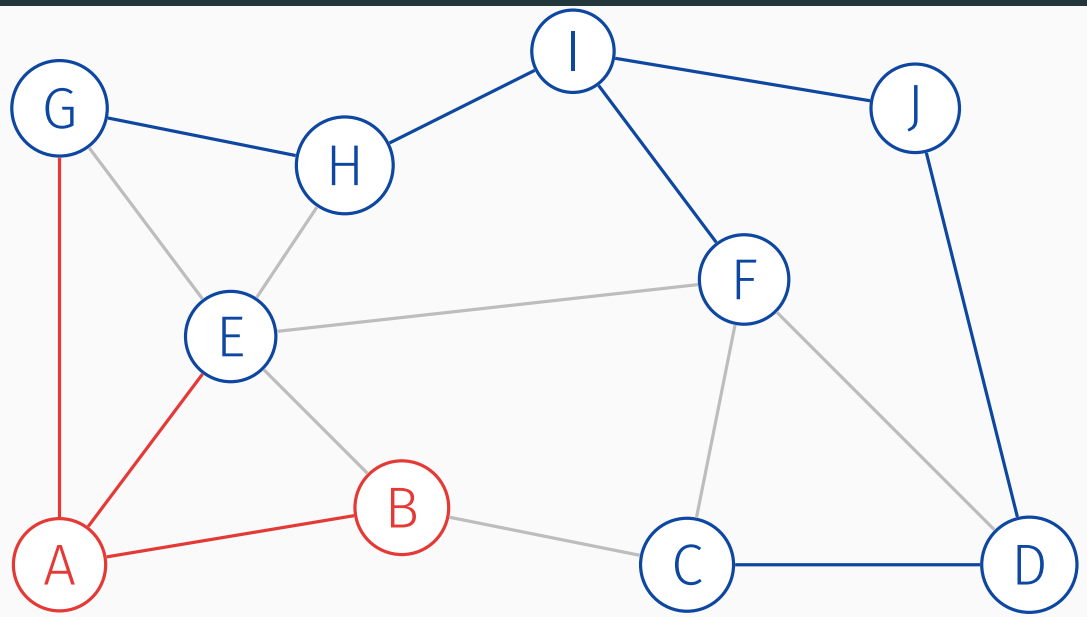


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	∞	∞	A
C	7	8	D
D	6	9	J
E	16	17	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	11	12	I
G	2	15	A
H	3	14	G
I	4	13	H
J	5	10	I

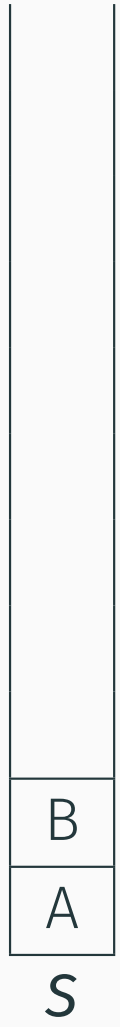


Depth-first graph traversal, step 29

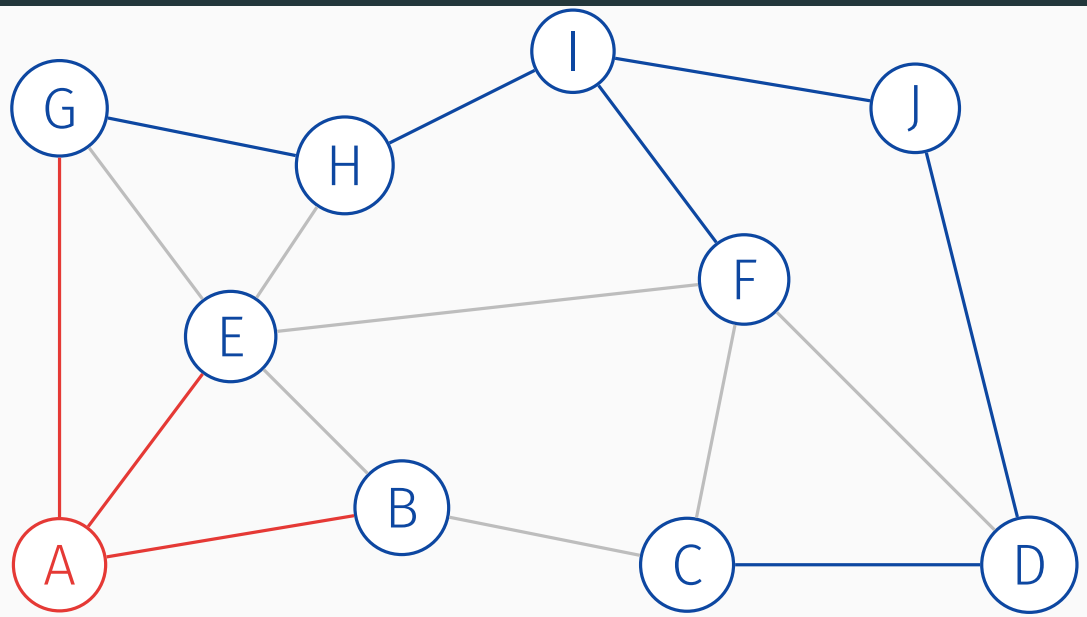


v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	18	∞	A
C	7	8	D
D	6	9	J
E	16	17	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	11	12	I
G	2	15	A
H	3	14	G
I	4	13	H
J	5	10	I



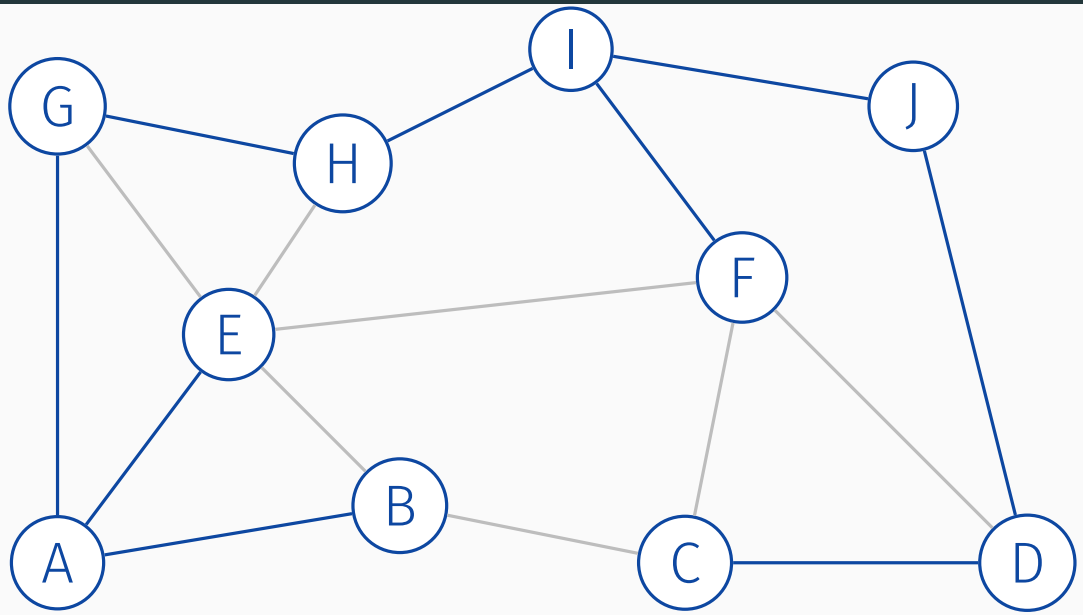
Depth-first graph traversal, step 30



v	$d[v]$	$f[v]$	$\pi[v]$
A	1	∞	/
B	18	19	A
C	7	8	D
D	6	9	J
E	16	17	A

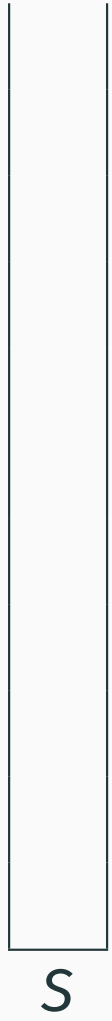
v	$d[v]$	$f[v]$	$\pi[v]$
F	11	12	I
G	2	15	A
H	3	14	G
I	4	13	H
J	5	10	I

Depth-first graph traversal, step 31



v	$d[v]$	$f[v]$	$\pi[v]$
A	1	20	/
B	18	19	A
C	7	8	D
D	6	9	J
E	16	17	A

v	$d[v]$	$f[v]$	$\pi[v]$
F	11	12	I
G	2	15	A
H	3	14	G
I	4	13	H
J	5	10	I



Algorithm for breadth-first graph traversal

Input : Graph $G(V, E)$, initial vertex $s \in V$
Output: BF tree

```
1  foreach u ∈ V/{s} do
2  |   state [u] ← unknown;
3  |   d [u] ← ∞;
4  |   π [u] ← nothing;
5  end
6  Q ← ∅;
7  state [s] ← discovered;
8  d [s] ← 0;
9  π [s] ← nothing;
10 Enqueue(Q, s);
```

Algorithm for breadth-first graph traversal (cont.)

```
11 while Q ≠ ∅ do
12 |   u ← Dequeue(Q);
13 |   foreach v ∈ Adj(G, u) do
14 |   |   if state [v] = unknown then
15 |   |   |   state [v] ← discovered;
16 |   |   |   d [v] ← d [u] + 1;
17 |   |   |   π [v] ← u;
18 |   |   |   Enqueue (Q,v);
19 |   |   end
20 |   end
21 |   state [u] ← finished;
22 end
```

Animation of breadth-first graph traversal – legend

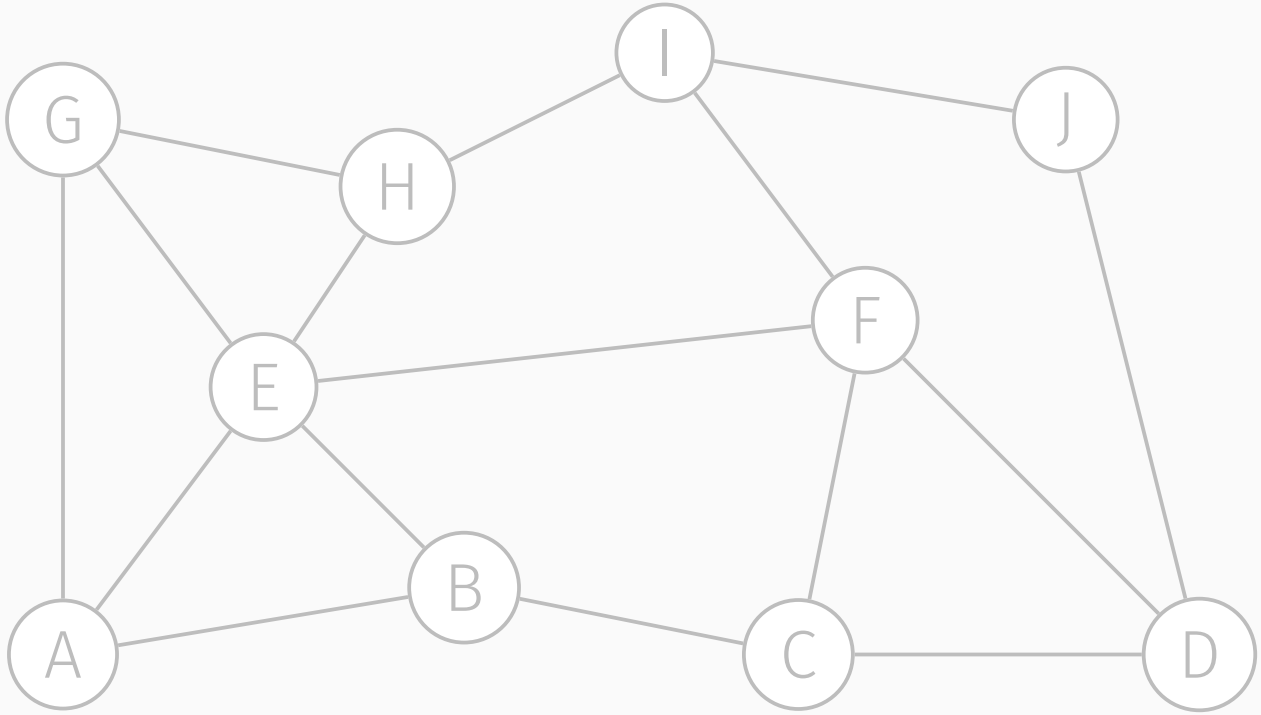
Graph vertices

grey	vertex in unknown state
yellow	vertex in discovered state
red	currently processed vertex
blue	vertex in finished state

Graph edges

grey	edge between vertices in unknown state or edge not belonging to the BF-tree
yellow	edge incident with vertices in discovered state
red	edge incident with currently processed vertex
blue	edge between vertices in finished state

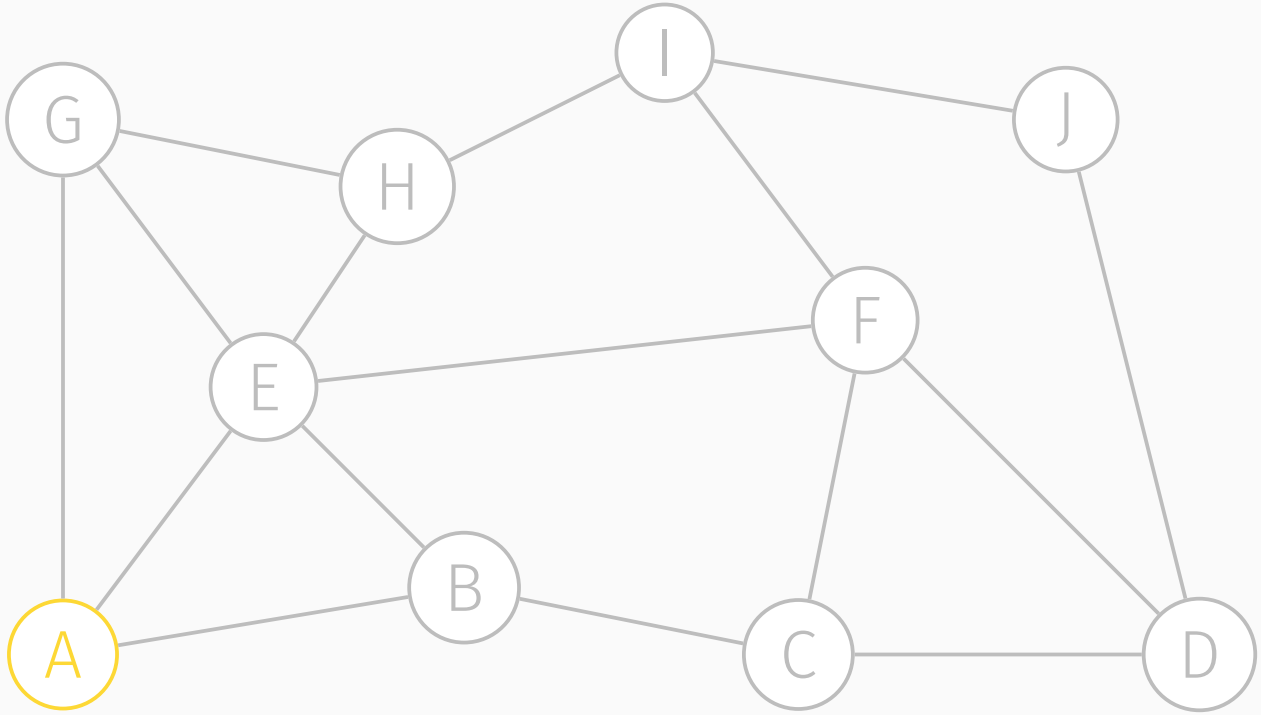
Breadth-first graph traversal, step 1



Q [ ]

v	$d[v]$	$\pi[v]$
A	∞	/
B	∞	/
C	∞	/
D	∞	/
E	∞	/
F	∞	/
G	∞	/
H	∞	/
I	∞	/
J	∞	/

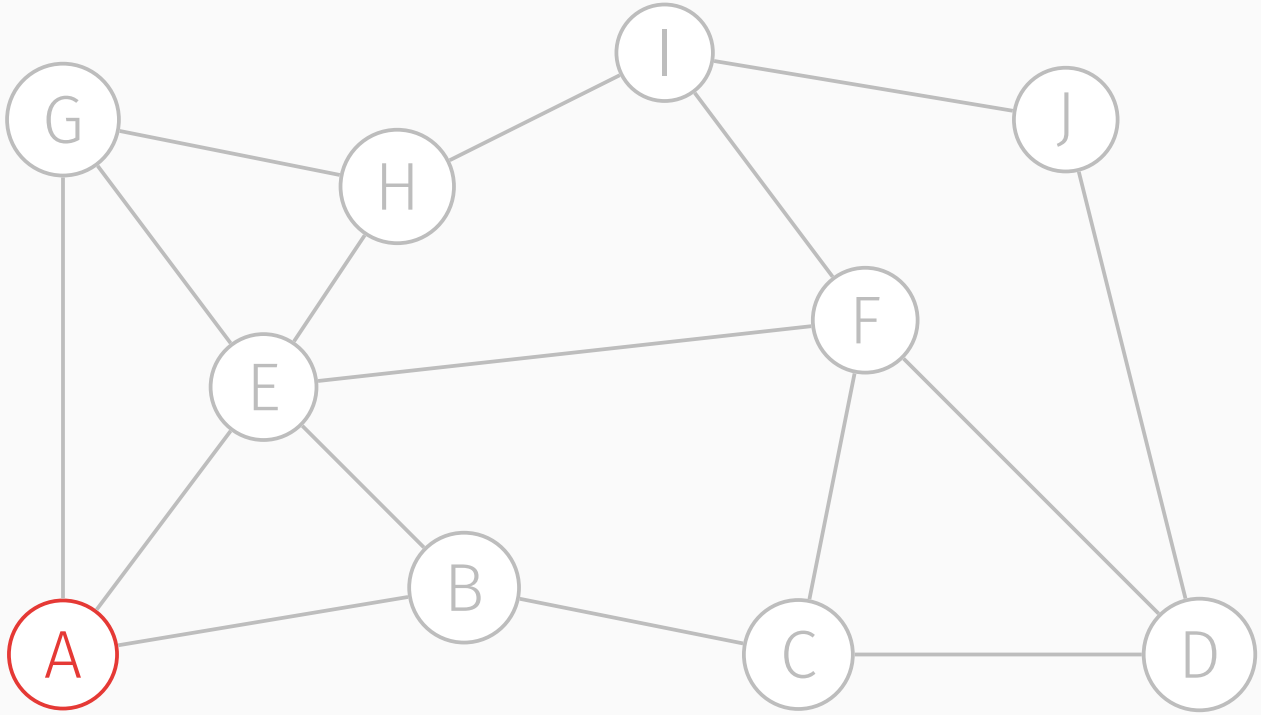
Breadth-first graph traversal, step 2



Q | A |

v	$d[v]$	$\pi[v]$
A	0	/
B	∞	/
C	∞	/
D	∞	/
E	∞	/
F	∞	/
G	∞	/
H	∞	/
I	∞	/
J	∞	/

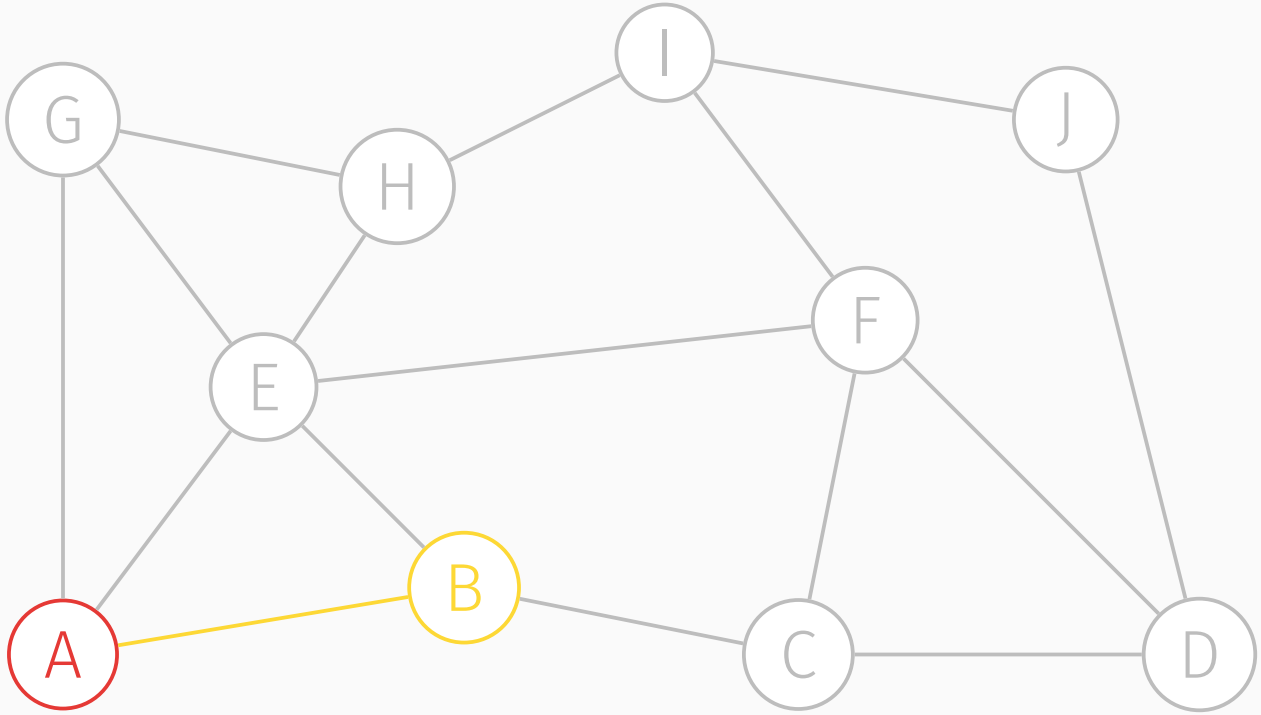
Breadth-first graph traversal, step 3



Q [ ]

v	$d[v]$	$\pi[v]$
A	0	/
B	∞	/
C	∞	/
D	∞	/
E	∞	/
F	∞	/
G	∞	/
H	∞	/
I	∞	/
J	∞	/

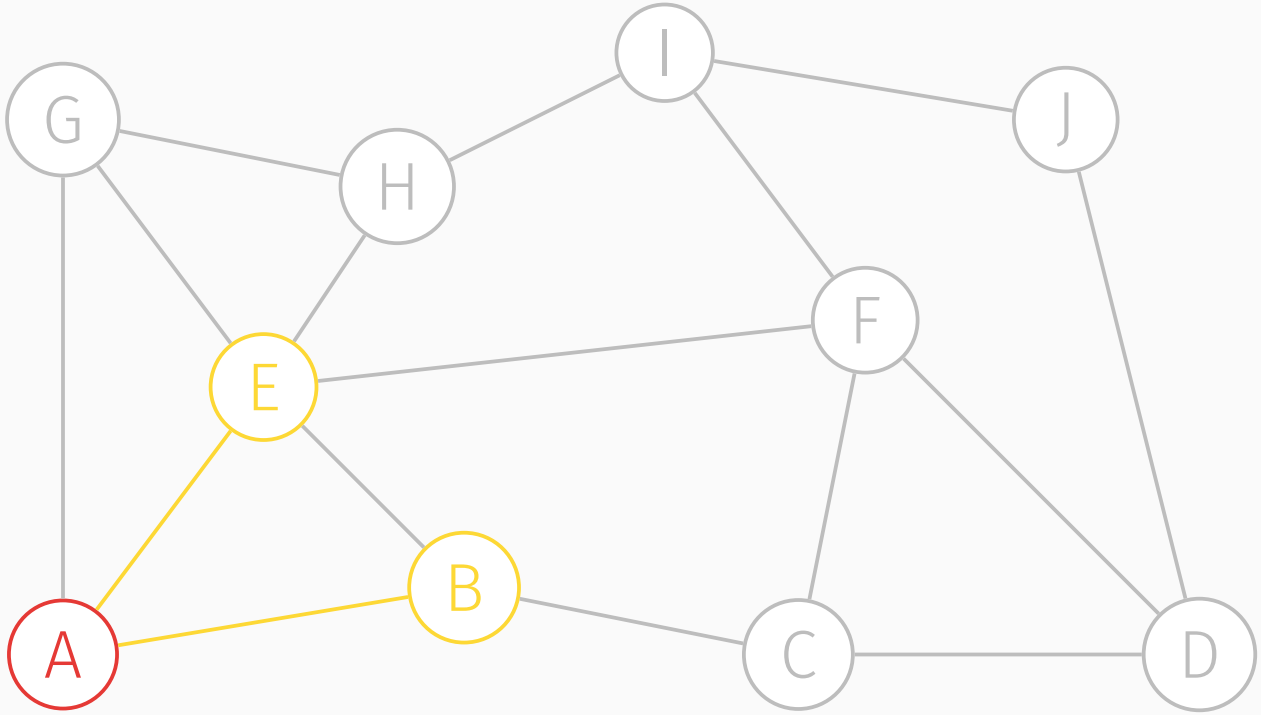
Breadth-first graph traversal, step 4



Q | B |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	∞	/
D	∞	/
E	∞	/
F	∞	/
G	∞	/
H	∞	/
I	∞	/
J	∞	/

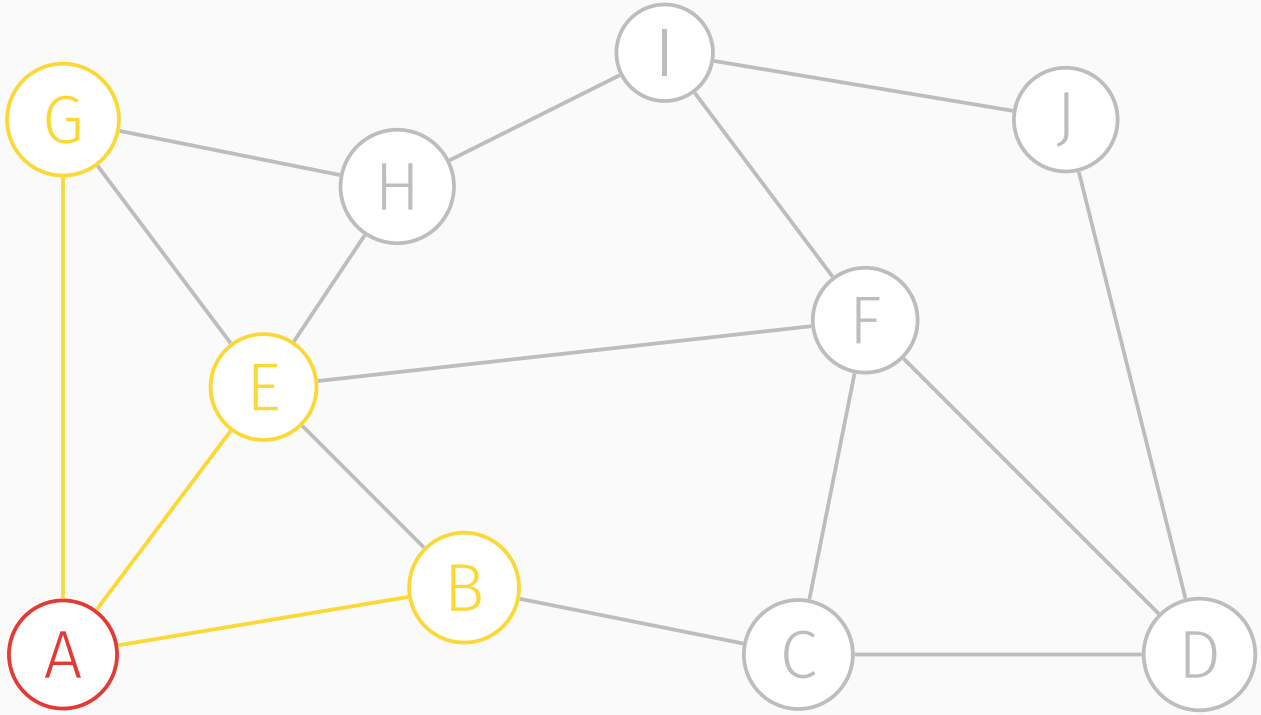
Breadth-first graph traversal, step 5



Q | B | E |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	∞	/
D	∞	/
E	1	A
F	∞	/
G	∞	/
H	∞	/
I	∞	/
J	∞	/

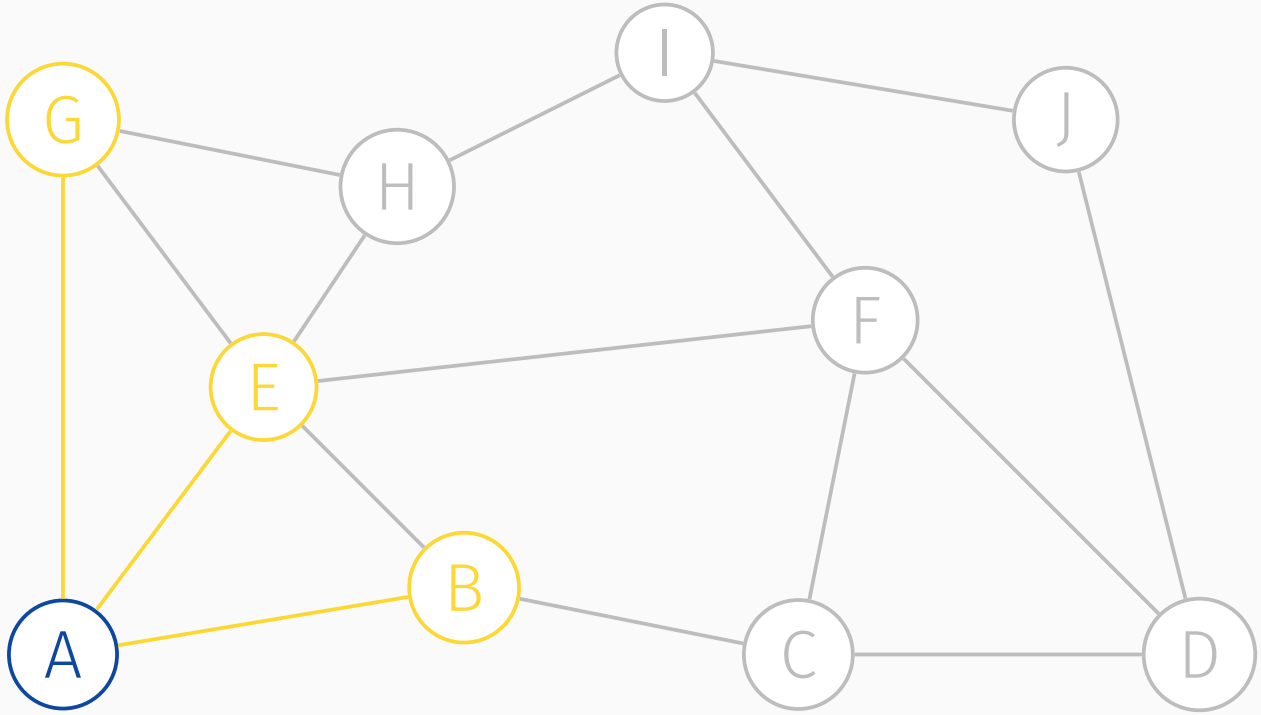
Breadth-first graph traversal, step 6



Q | B | E | G |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	∞	/
D	∞	/
E	1	A
F	∞	/
G	1	A
H	∞	/
I	∞	/
J	∞	/

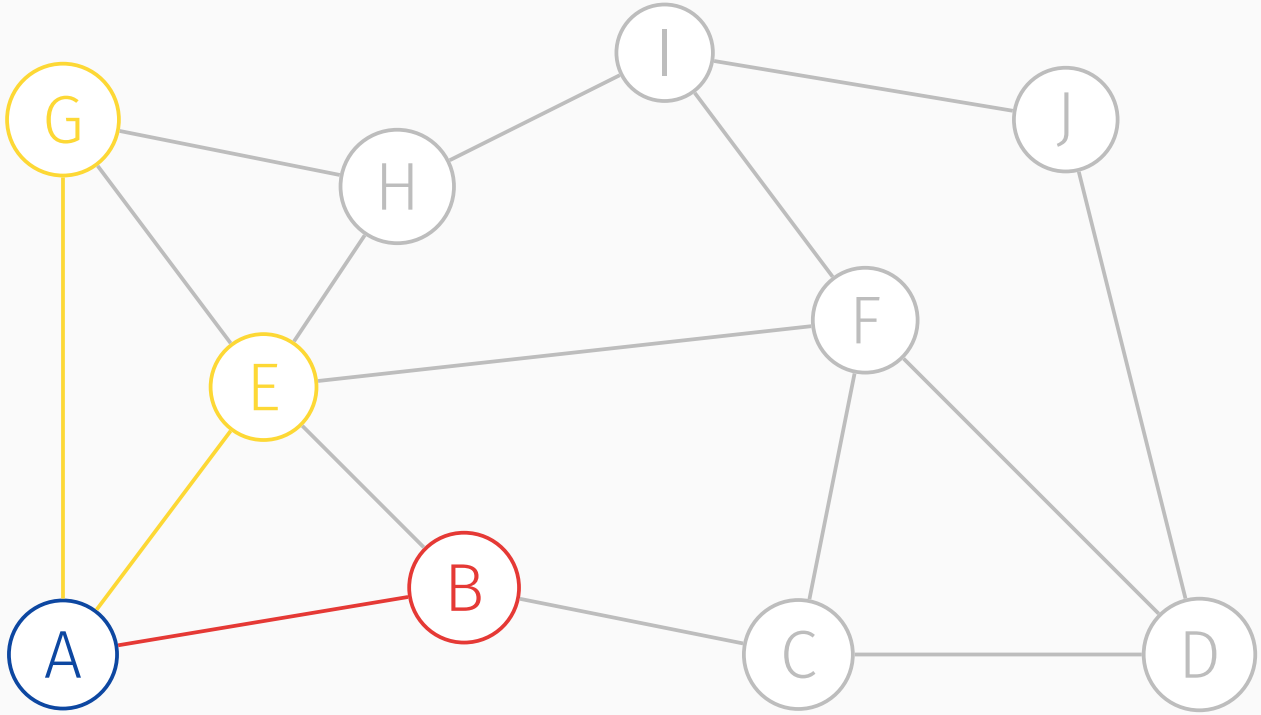
Breadth-first graph traversal, step 7



Q | B | E | G |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	∞	/
D	∞	/
E	1	A
F	∞	/
G	1	A
H	∞	/
I	∞	/
J	∞	/

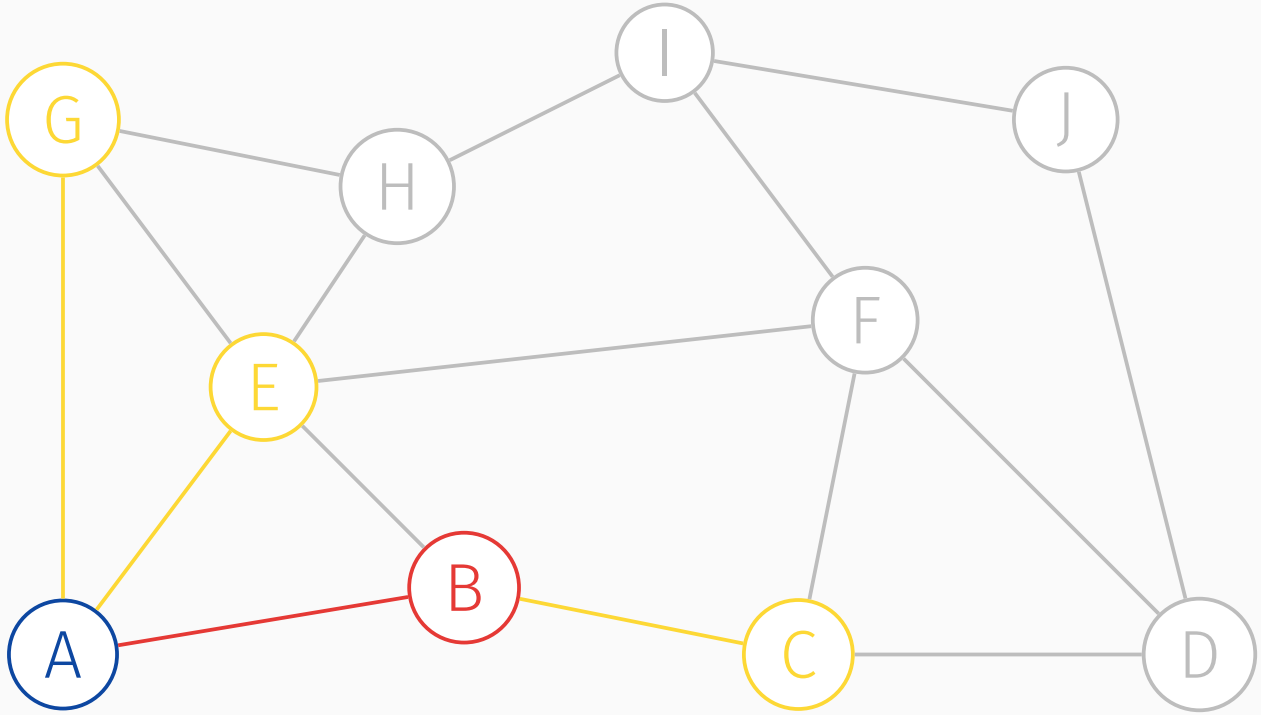
Breadth-first graph traversal, step 8



Q | E | G |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	∞	/
D	∞	/
E	1	A
F	∞	/
G	1	A
H	∞	/
I	∞	/
J	∞	/

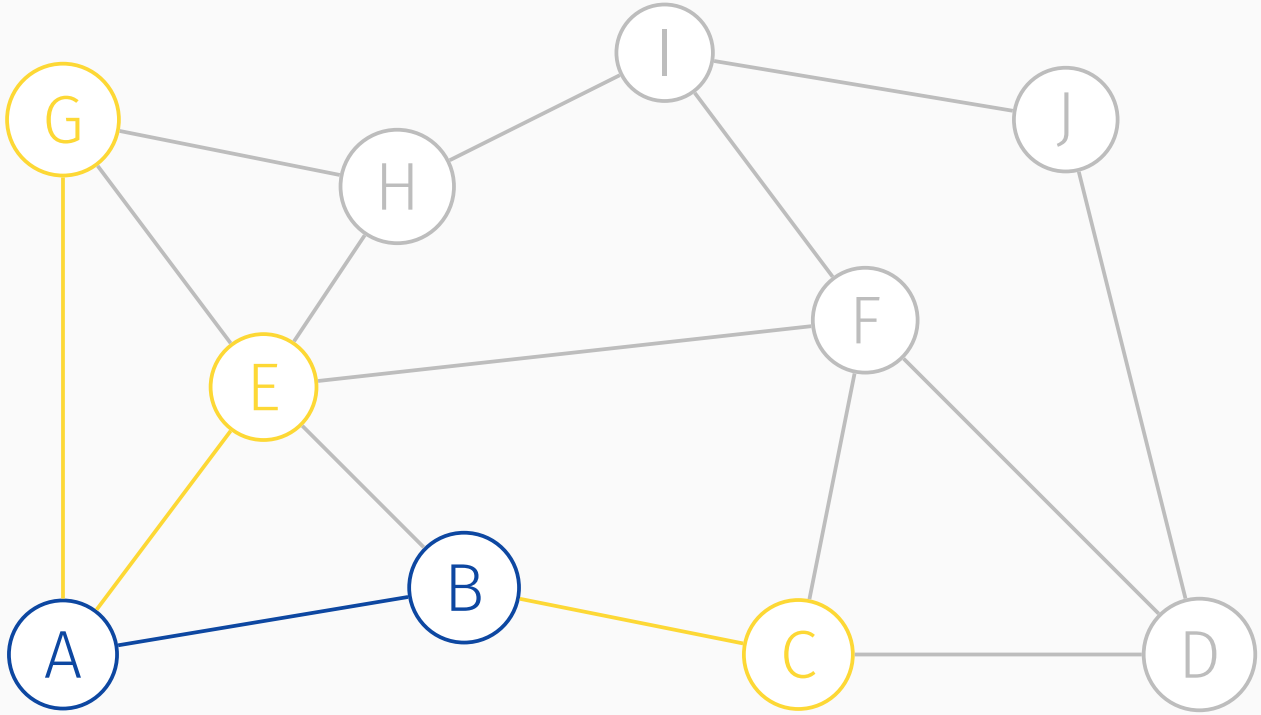
Breadth-first graph traversal, step 9



Q | E | G | C |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	∞	/
E	1	A
F	∞	/
G	1	A
H	∞	/
I	∞	/
J	∞	/

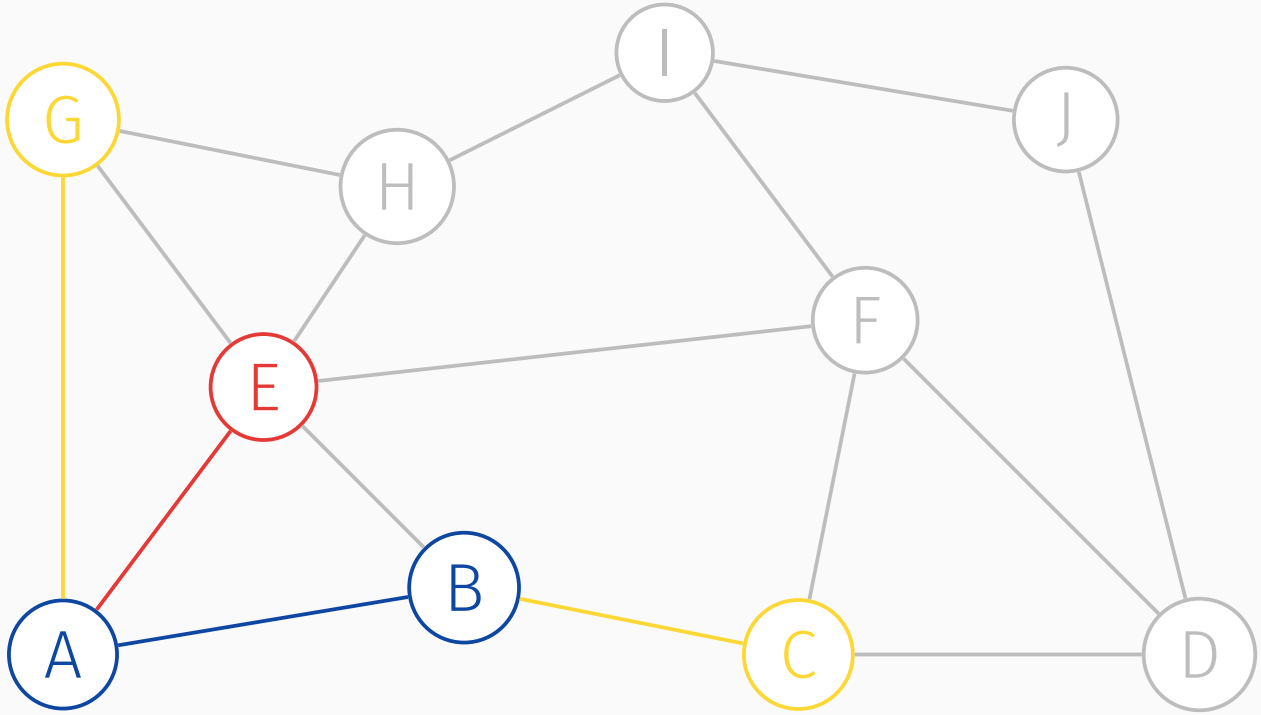
Breadth-first graph traversal, step 10



Q | E | G | C |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	∞	/
E	1	A
F	∞	/
G	1	A
H	∞	/
I	∞	/
J	∞	/

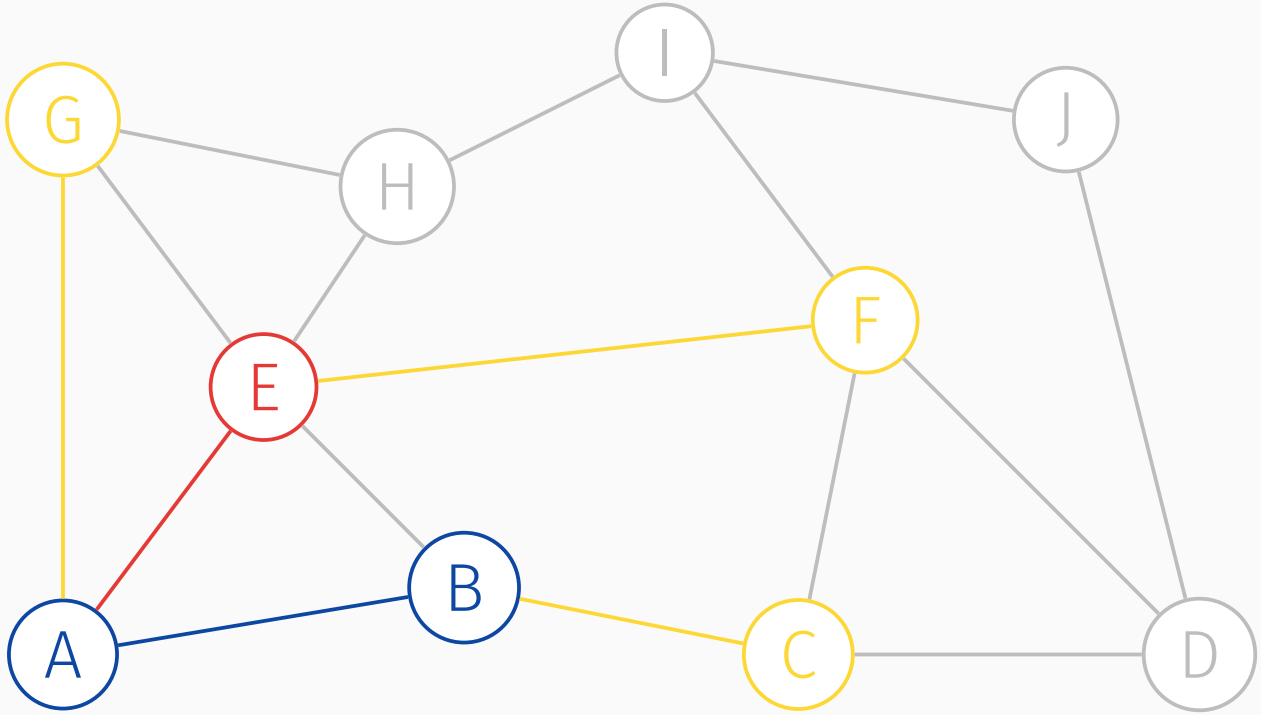
Breadth-first graph traversal, step 11



Q | G | C |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	∞	/
E	1	A
F	∞	/
G	1	A
H	∞	/
I	∞	/
J	∞	/

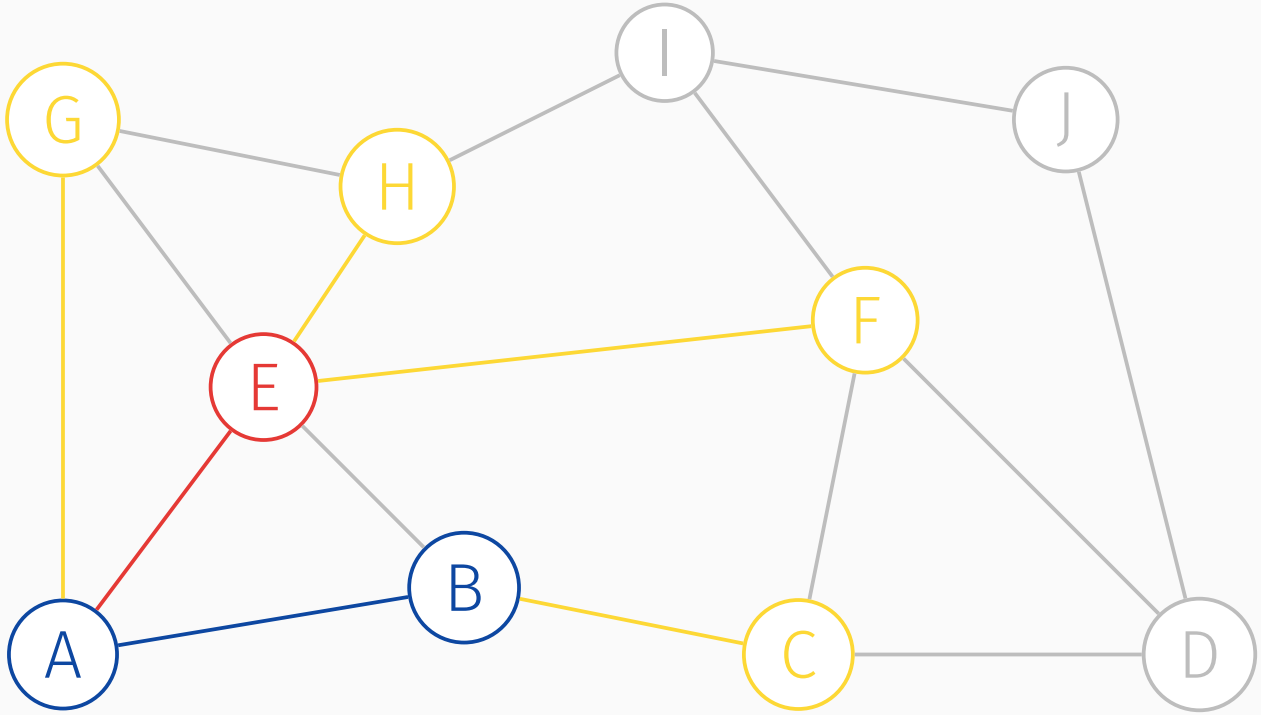
Breadth-first graph traversal, step 12



Q | G | C | F |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	∞	/
E	1	A
F	2	E
G	1	A
H	∞	/
I	∞	/
J	∞	/

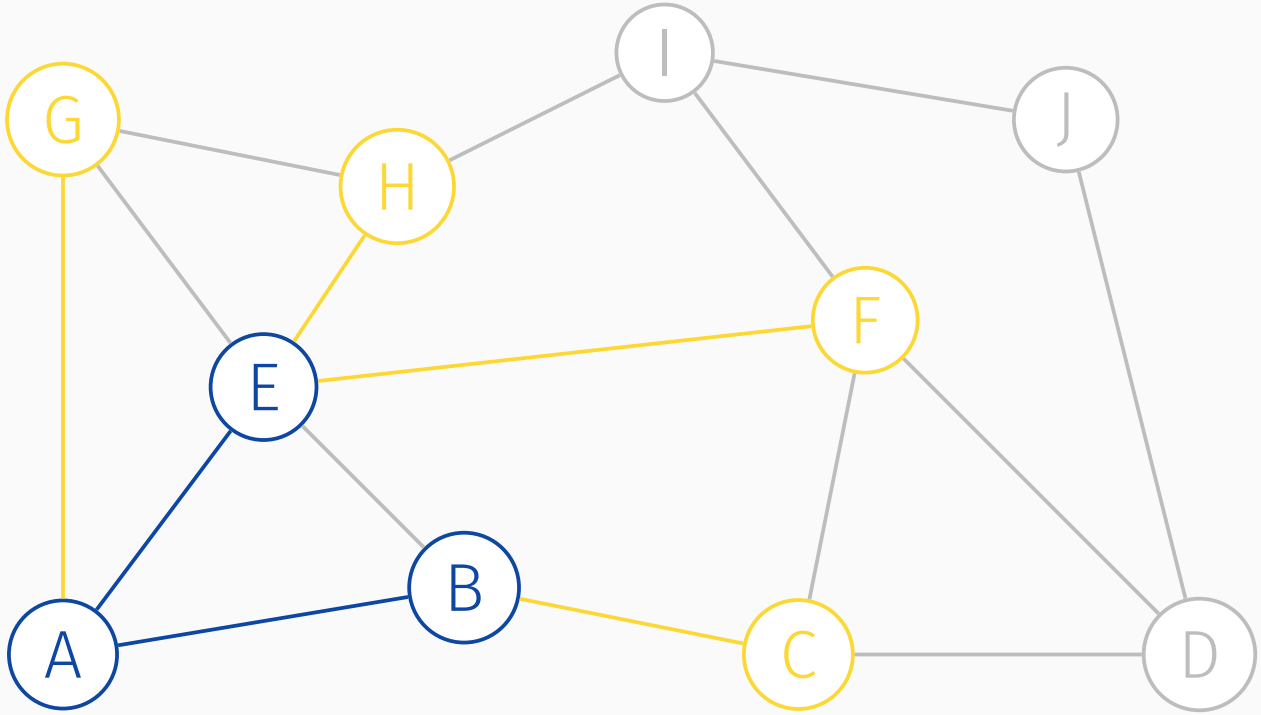
Breadth-first graph traversal, step 13



v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	∞	/
E	1	A
F	2	E
G	1	A
H	2	E
I	∞	/
J	∞	/

Q | G | C | F | H |

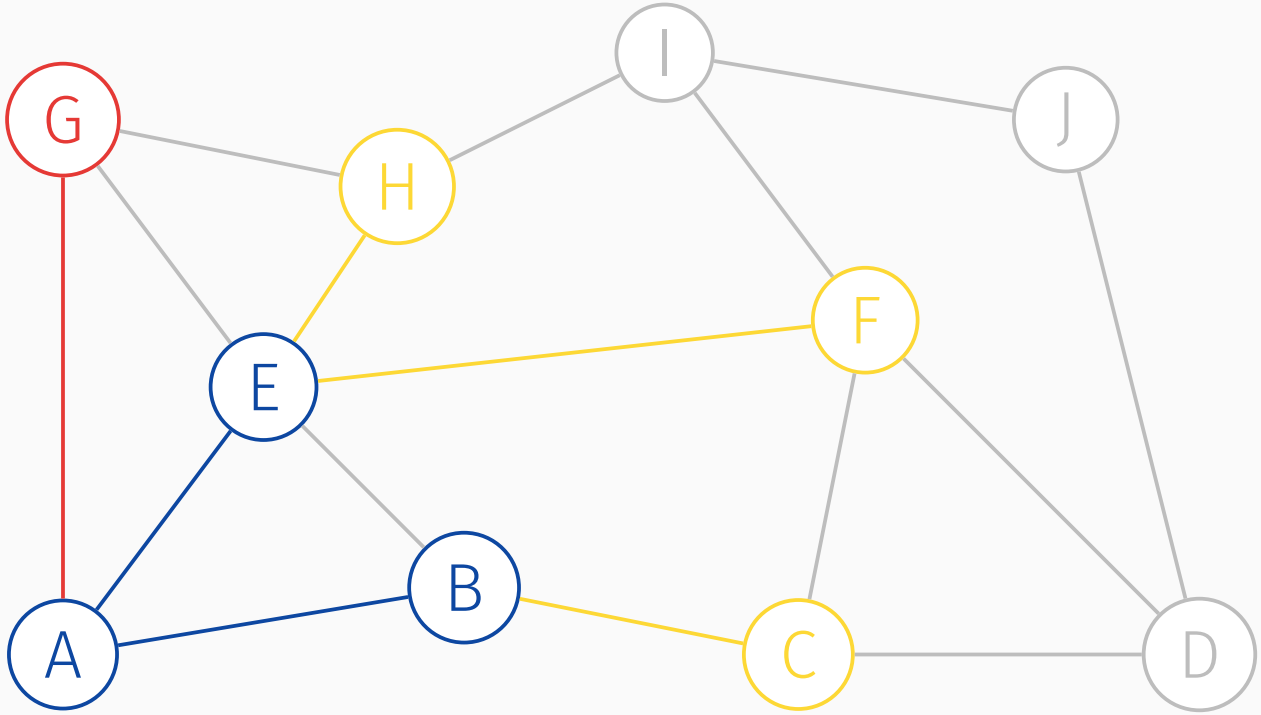
Breadth-first graph traversal, step 14



v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	∞	/
E	1	A
F	2	E
G	1	A
H	2	E
I	∞	/
J	∞	/

Q | G | C | F | H |

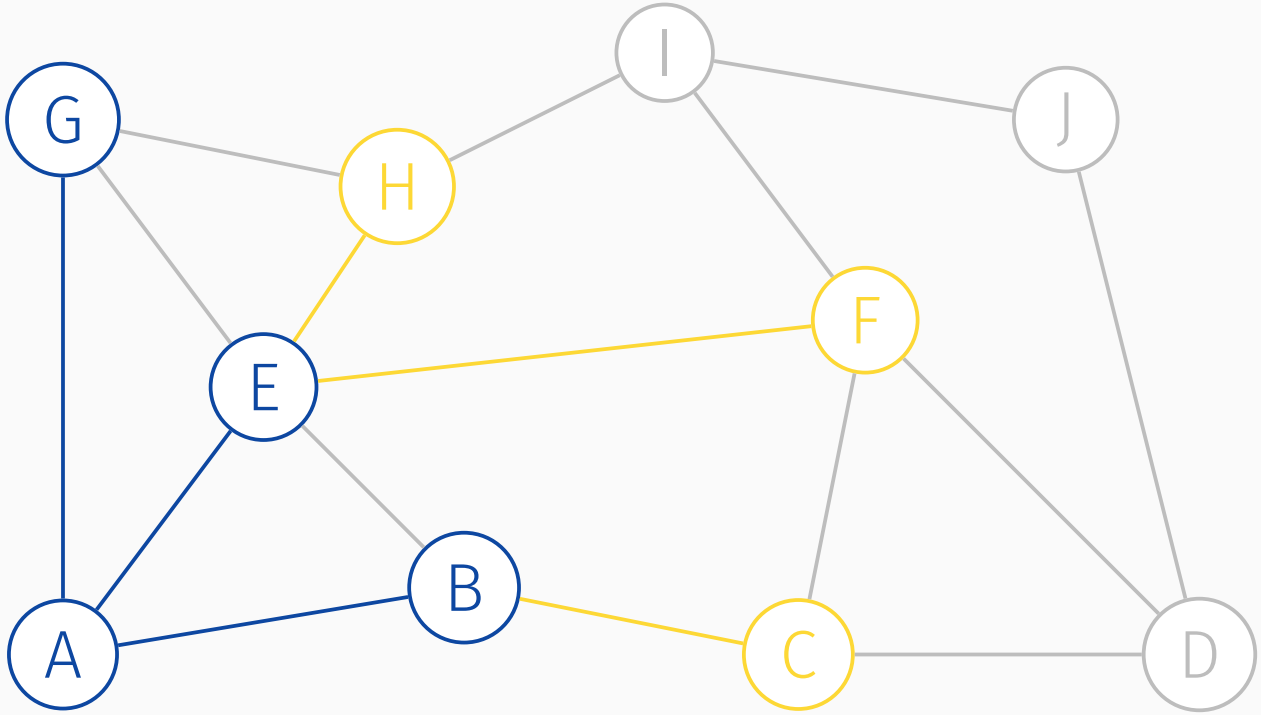
Breadth-first graph traversal, step 15



Q | C | F | H |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	∞	/
E	1	A
F	2	E
G	1	A
H	2	E
I	∞	/
J	∞	/

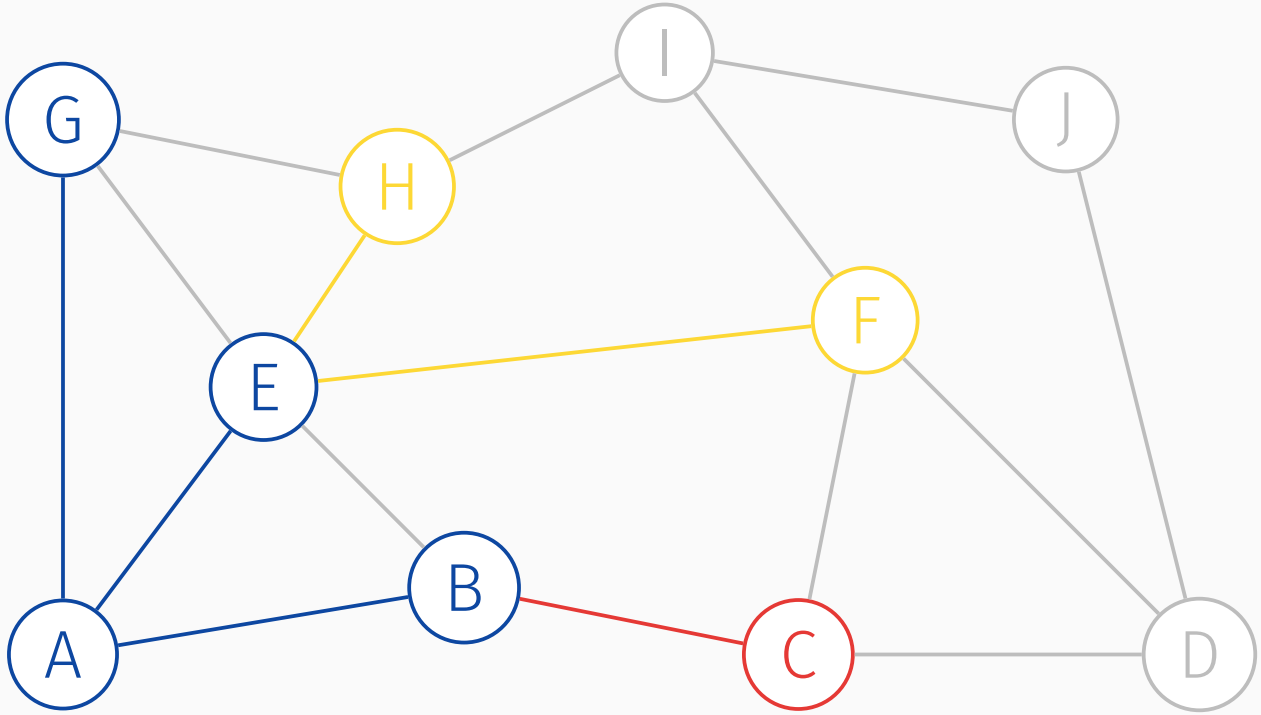
Breadth-first graph traversal, step 16



Q | C | F | H |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	∞	/
E	1	A
F	2	E
G	1	A
H	2	E
I	∞	/
J	∞	/

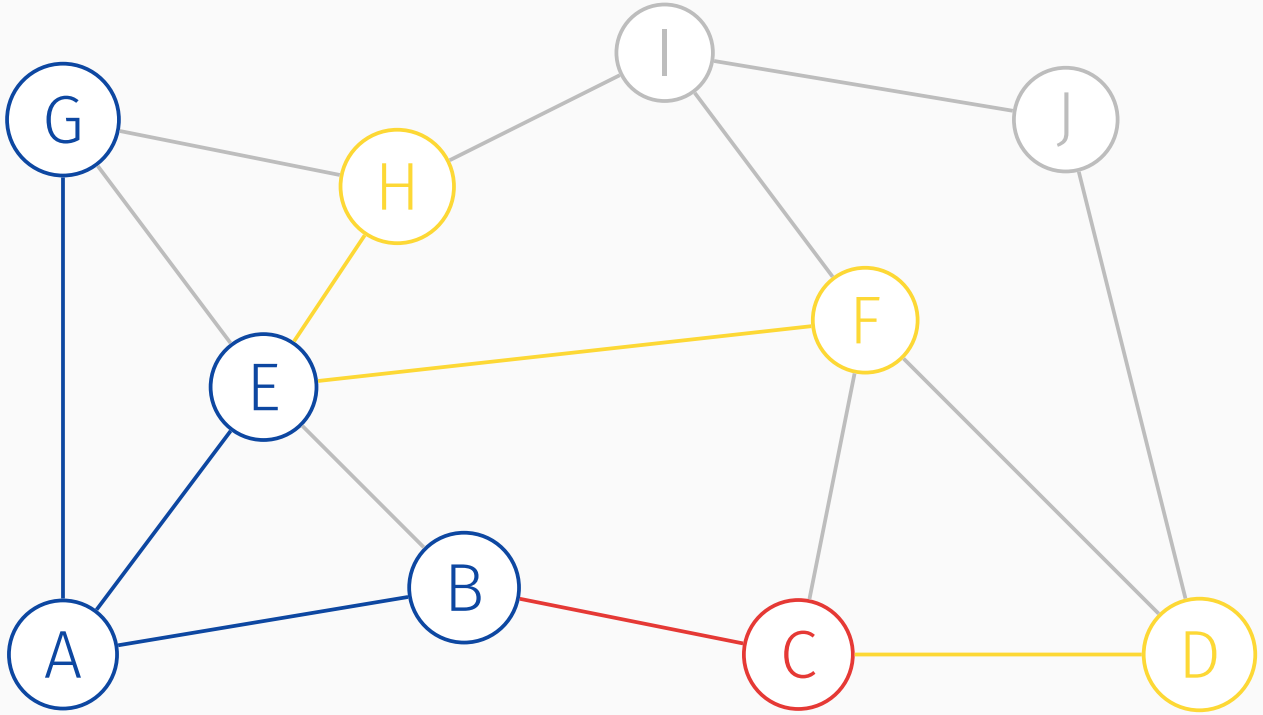
Breadth-first graph traversal, step 17



Q | F | H |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	∞	/
E	1	A
F	2	E
G	1	A
H	2	E
I	∞	/
J	∞	/

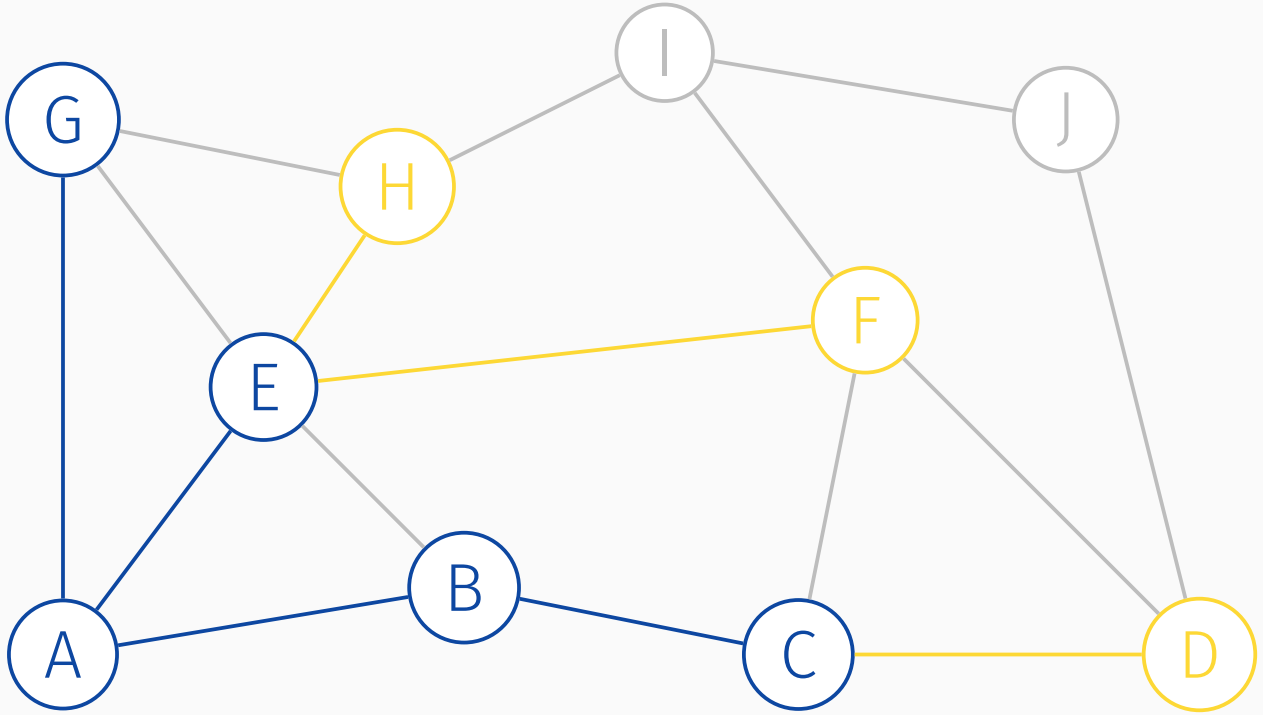
Breadth-first graph traversal, step 18



Q | F | H | D |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	∞	/
J	∞	/

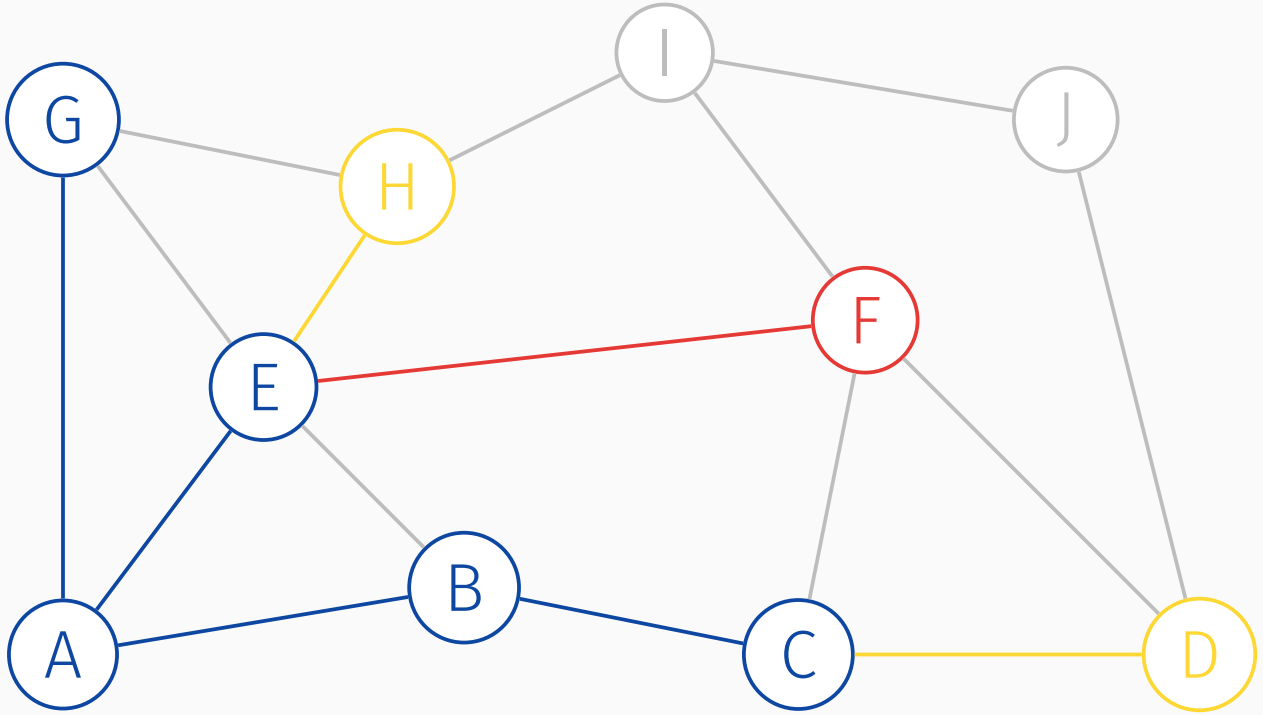
Breadth-first graph traversal, step 19



Q | F | H | D |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	∞	/
J	∞	/

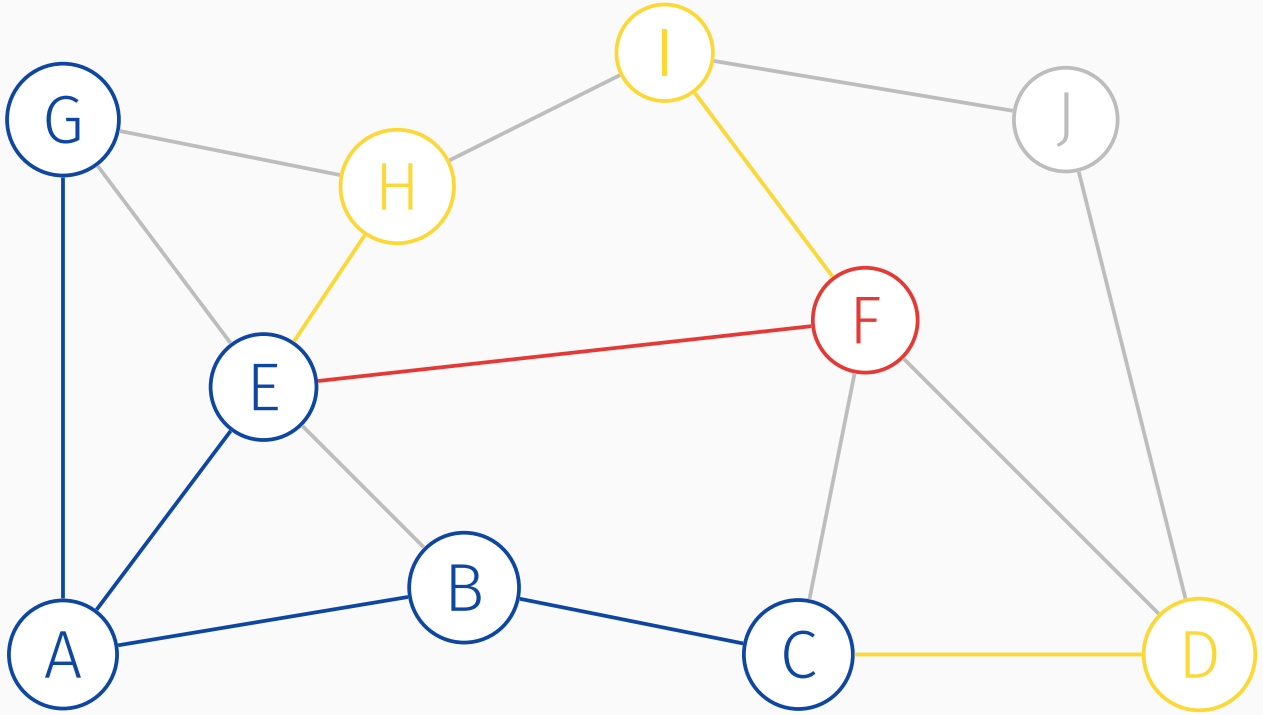
Breadth-first graph traversal, step 20



Q | H | D |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	∞	/
J	∞	/

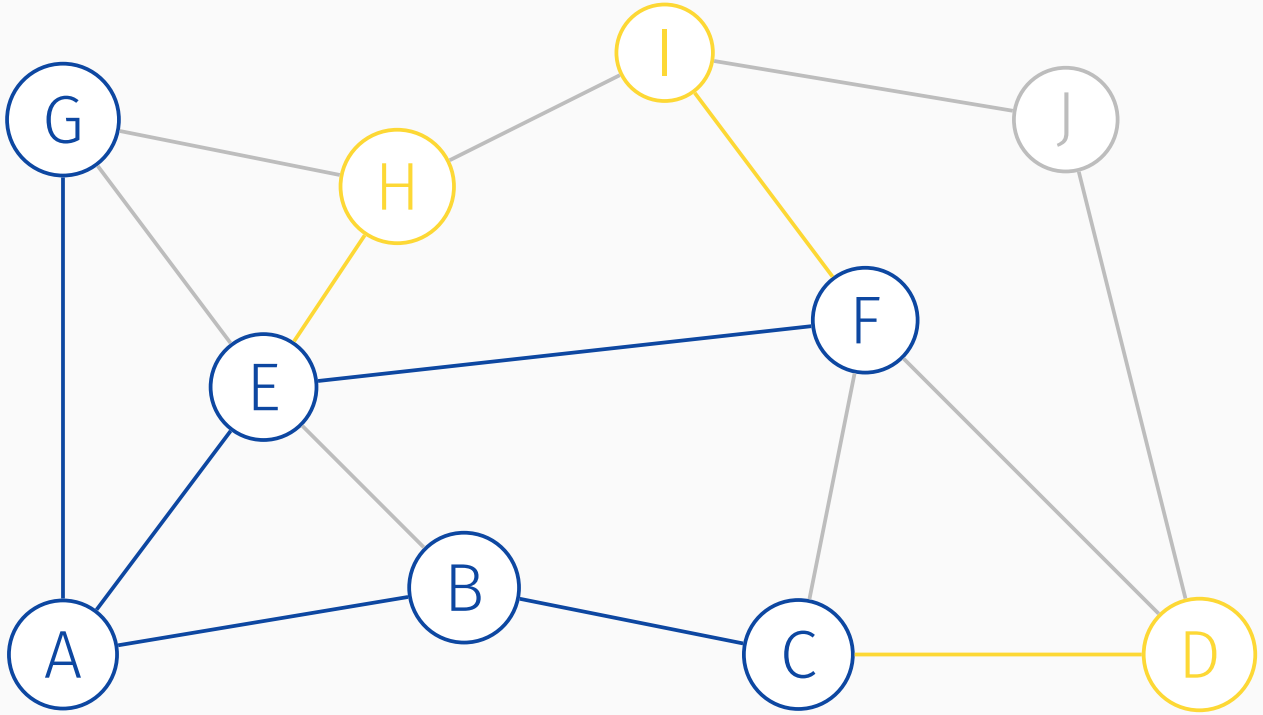
Breadth-first graph traversal, step 21



Q | H | D | I |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	∞	/

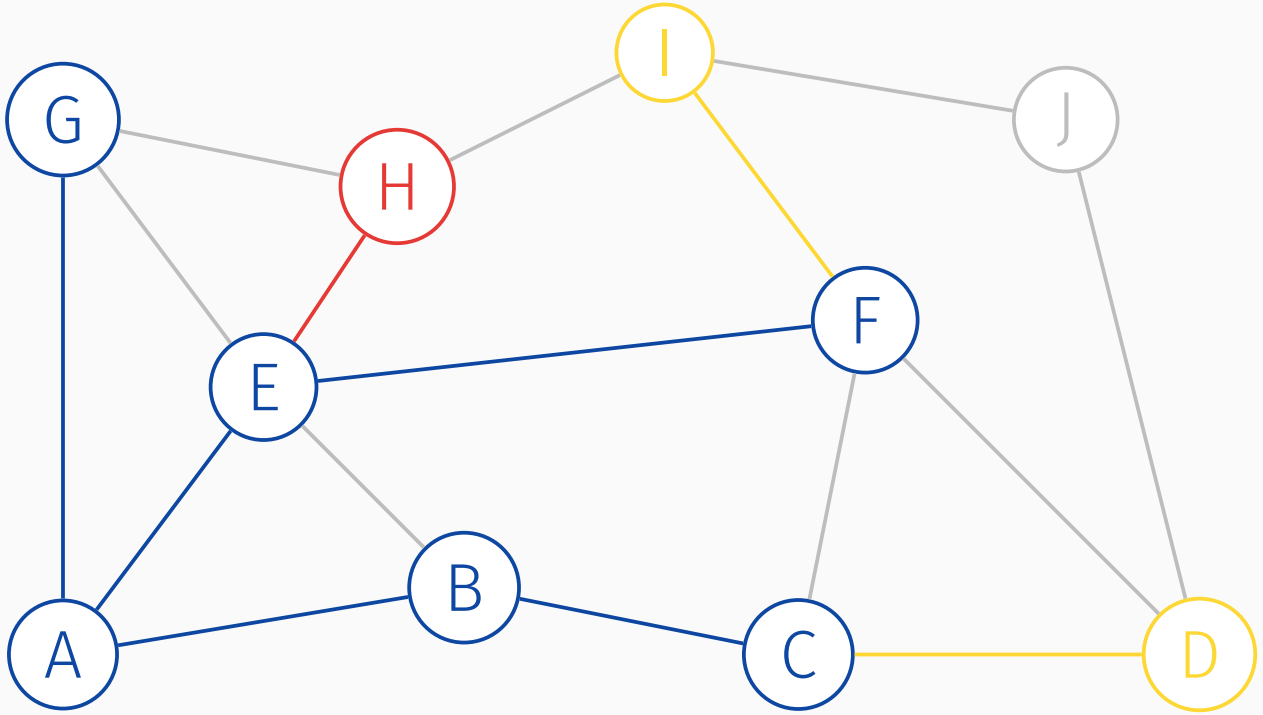
Breadth-first graph traversal, step 22



v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	∞	/

Q | H | D | I |

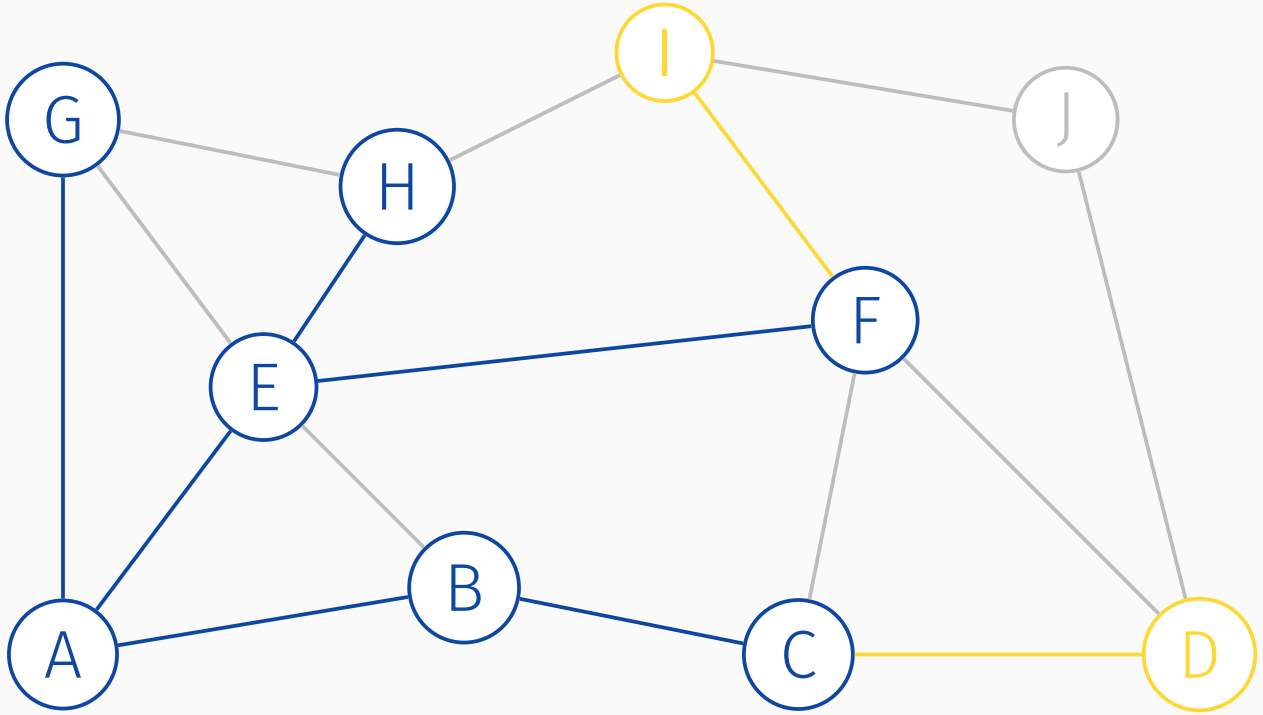
Breadth-first graph traversal, step 23



Q | D | I |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	∞	/

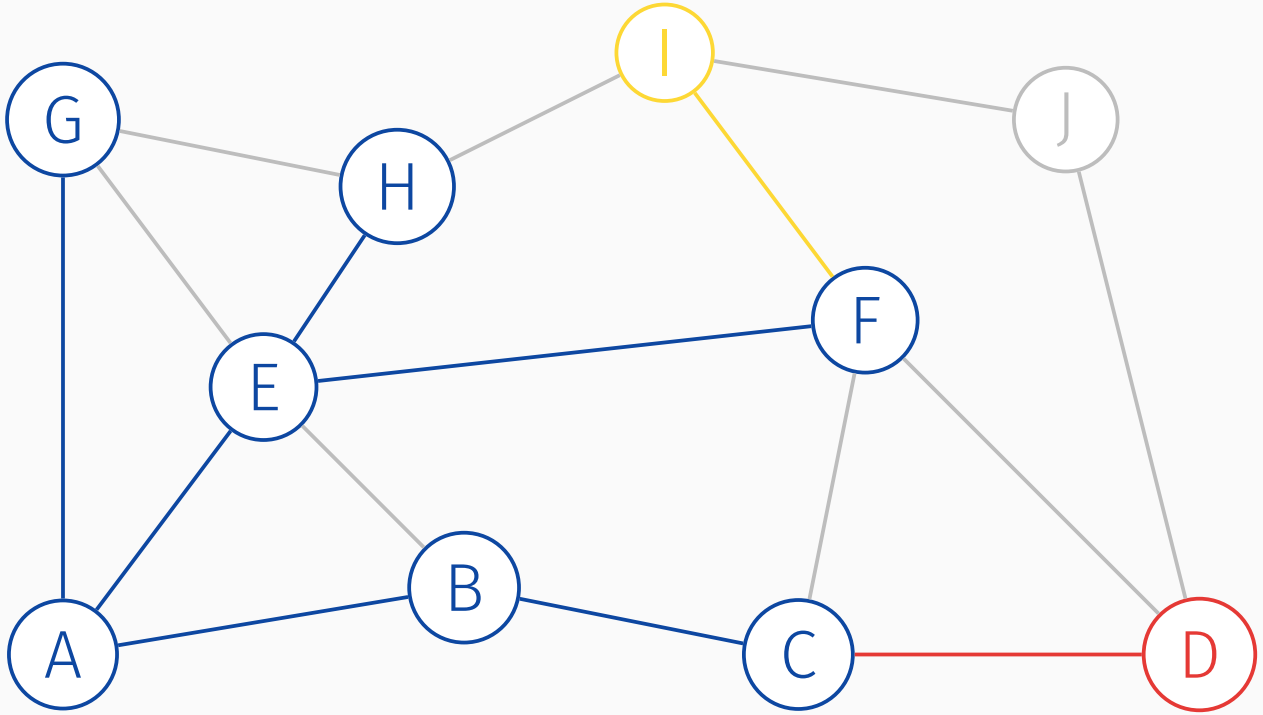
Breadth-first graph traversal, step 24



Q | D | I |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	∞	/

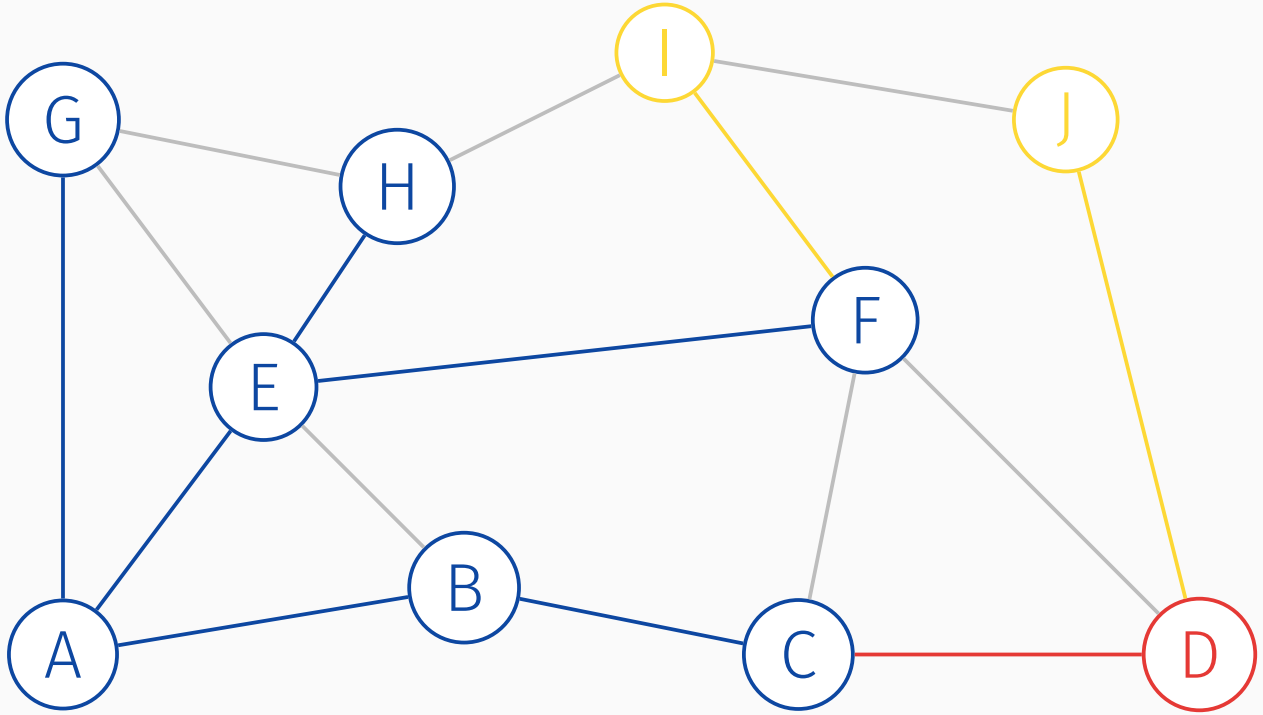
Breadth-first graph traversal, step 25



Q | I |

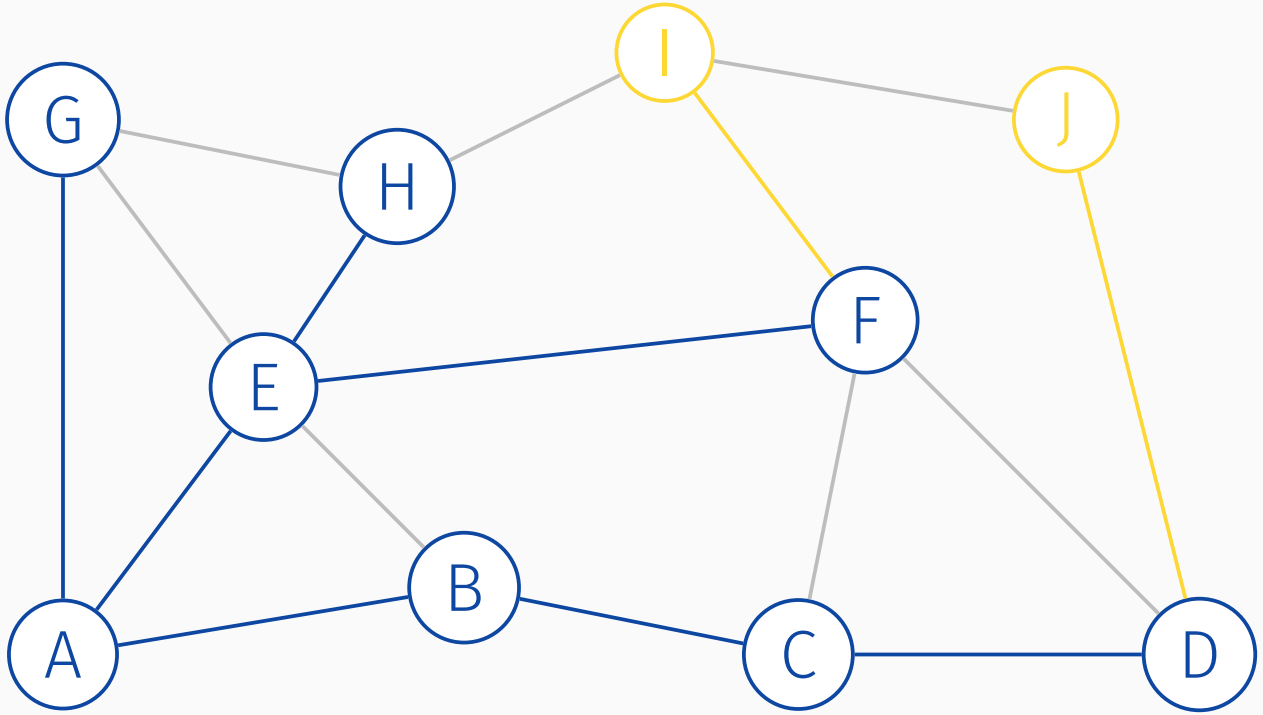
v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	∞	/

Breadth-first graph traversal, step 26



v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	4	D

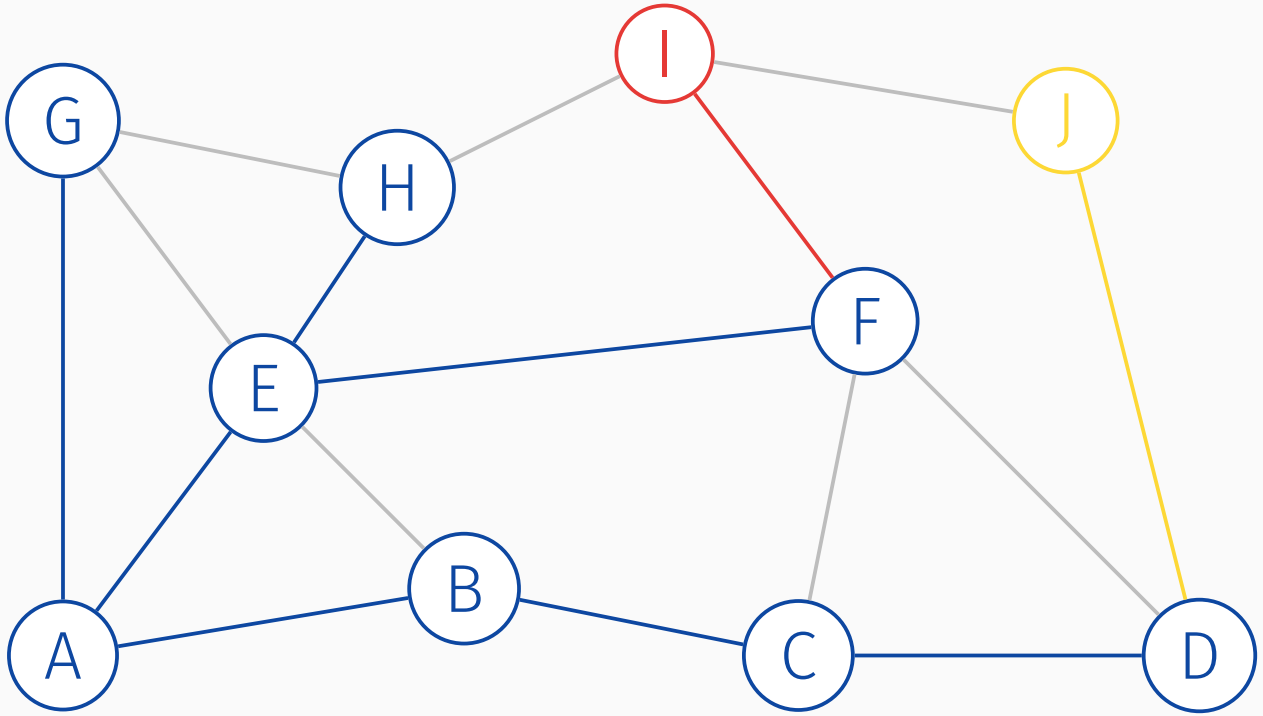
Breadth-first graph traversal, step 27



Q | I | J |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	4	D

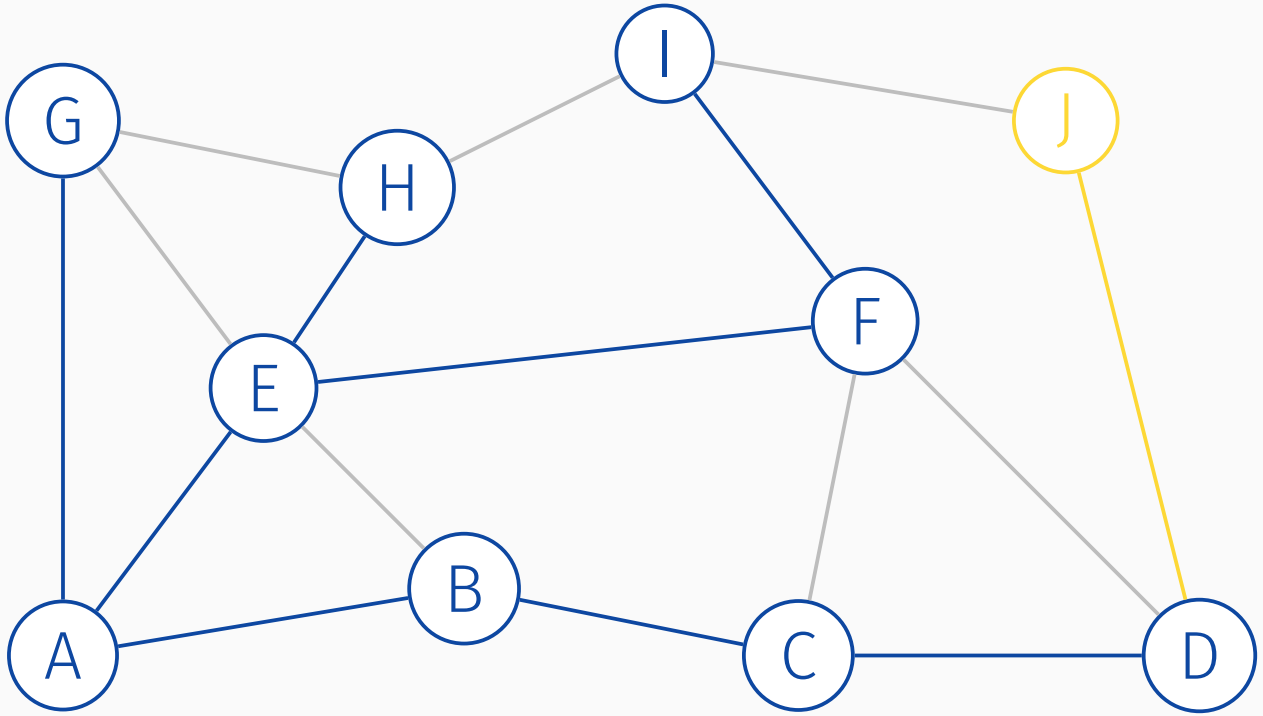
Breadth-first graph traversal, step 28



Q | J |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	4	D

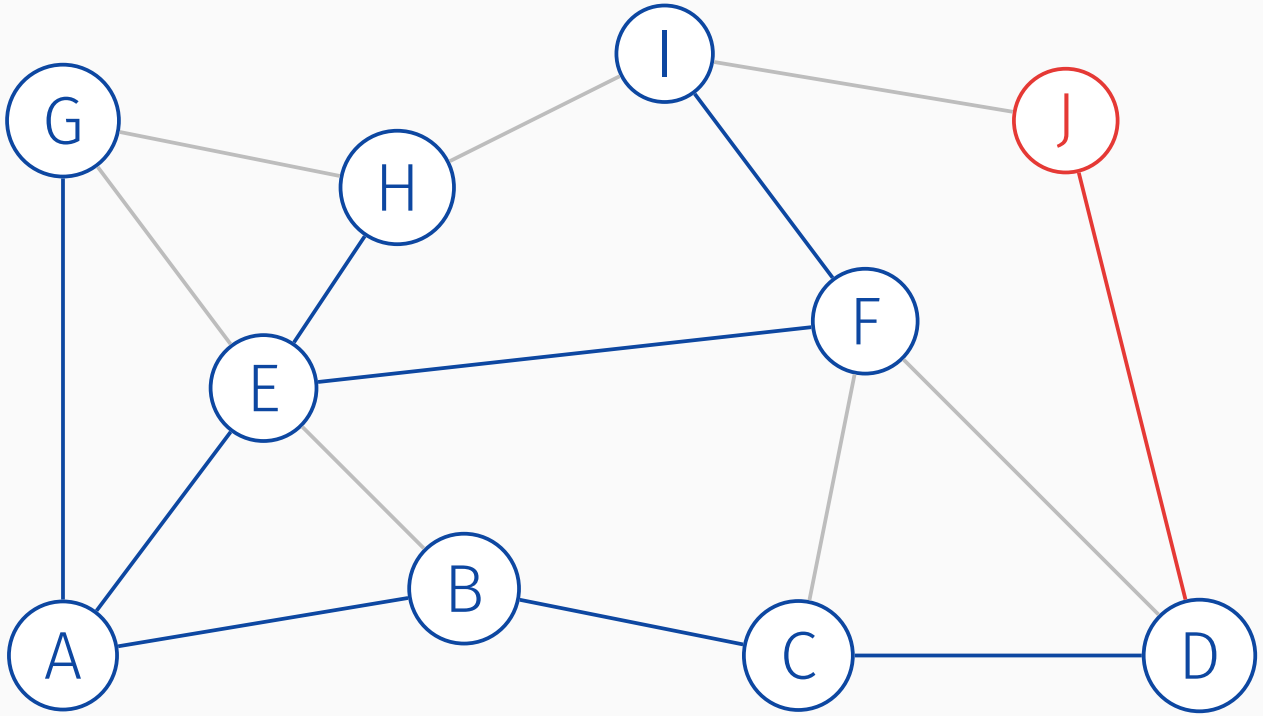
Breadth-first graph traversal, step 29



Q | J |

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	4	D

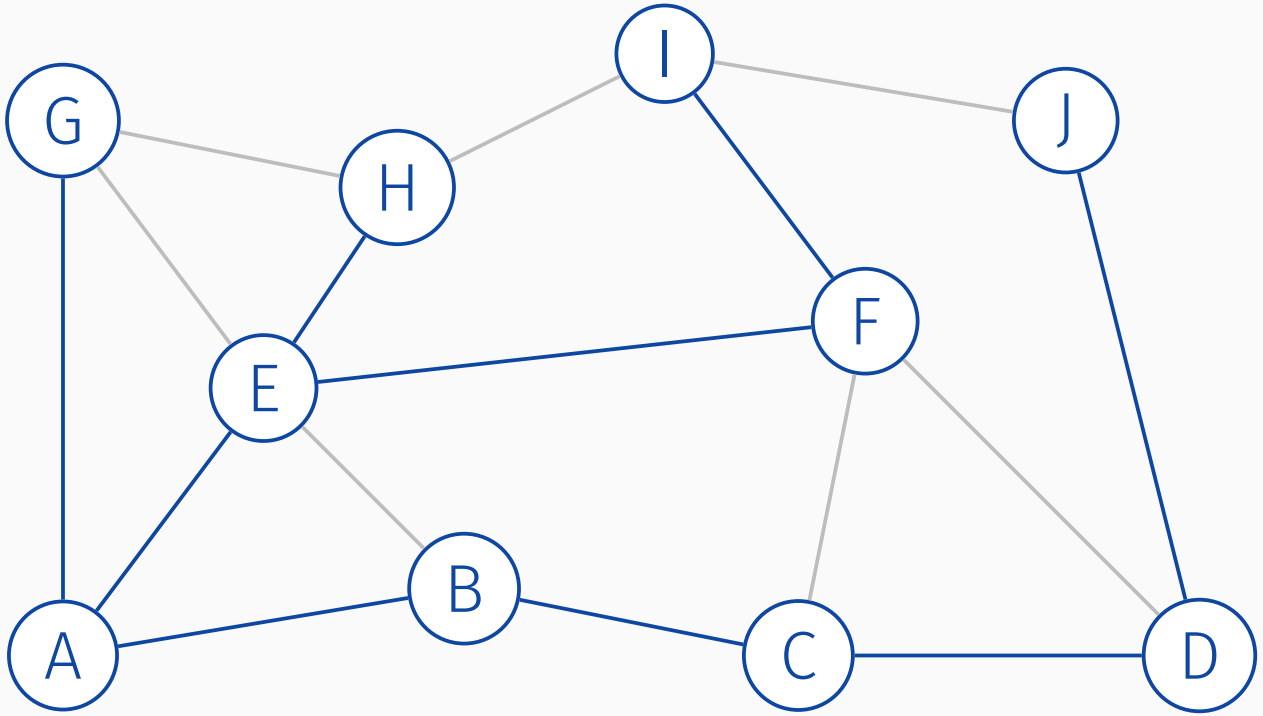
Breadth-first graph traversal, step 30



Q 

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	4	D

Breadth-first graph traversal, step 31



Q 

v	$d[v]$	$\pi[v]$
A	0	/
B	1	A
C	2	B
D	3	C
E	1	A
F	2	E
G	1	A
H	2	E
I	3	F
J	4	D

Depth-first and breadth-first graph traversal

Animation

For both graph traversal algorithms, two animations are available:

- path search in a maze and
- application in computer graphics – area filling.

The animations are available as separate file animations.

Sources for independent study

- Brute force problem solving strategies
  - book [2], chapter 3, pages 97 – 98
- Selection sort
  - book [2], chapter 3.1, pages 98 – 100
- Bubble sort
  - book [2], chapter 3.1, pages 100 – 101
- Shaker sort
  - book [4], chapter 4, pages 78 – 79
- Sequential search
  - book [2], chapter 3.2, pages 104 – 104
- Brute force string matching
  - book [2], chapter 3.2, pages 105 – 106

Sources for independent study (cont.)

- Closest pair problem
  - book [2], chapter 3.3, pages 108 – 109
- Convex hull problem
  - book [2], chapter 3.3, pages 109 – 113
- Exhaustive search
  - book [2], chapter 3.4, page 115
- Traveling salesman problem
  - book [2], chapter 3.4, page 116
- Knapsack problem
  - book [2], chapter 3.4, pages 116 – 117
- Depth-first graph traversal
  - book [2], chapter 3.5, pages 122 – 125
- Breadth-first graph traversal
  - book [2], chapter 3.5, pages 125 – 128

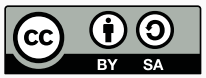
Thanks for your attention

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Decrease and Conquer

Jiří Dvorský, Ph.D.

Department of Computer Science
VSB – Technical University of Ostrava



Decrease and Conquer

Sorting by insertion – Insertion Sort

Decrease and Conquer

Topological sorting

Decrease and Conquer

Generating combinatorial objects

Generating combinatorial objects

- Generating combinations, variations, permutations, subsets is part of various algorithms.
- Typically it involves selecting some alternative, option, setting parameters...
- Examples – Traveling salesman problem, Knapsack problem.
- Mathematics is more interested in counting these objects, while computer science seeks algorithms to generate them.
- The number of these objects grows exponentially or even faster!

Generating permutations

- We will generate permutations of integers $1, 2, \ldots, n$.
- More generally, we can generate permutations of elements $\{a_1, a_2, \ldots, a_n\}$.
- Using the decrease and conquer strategy:
  1. Generating $n!$ permutations for n elements is reduced to generating $(n-1)!$ permutations of $n-1$ elements.
  2. Once we have solved the problem for $n-1$, we insert element n into all n possible positions in each of the $(n-1)!$ permutations.
  3. In other words, we have $(n-1)!$ permutations, and for each of them, we generate n additional ones. Overall, we obtain $n(n-1)! = n!$ permutations.

Generating permutations – example

permutation of element 1	1	
insertion of 2 into permutation 1 from right to left	1**2**	**2**1
insertion of 3 into permutation 12 from right to left	12**3**	1**3**2
insertion of 3 into permutation 21 from left to right	**3**21	2**3**1

What is evident from the example?

- All permutations are mutually distinct.
- Minimal change between permutations – two consecutive permutations differ by swapping a single pair of elements and even adjacent elements.

Johnson-Trotter algorithm

- Is there a possibility to generate permutations of n elements? Without the need to generate permutations for $n - 1$? Yes, there is.
- We assign an **arrow** (direction) to each of the n elements of the permutation, either to the left or to the right.
- We say that an element k is **mobile** in a given permutation if the neighboring element in the direction of the arrow of element k is smaller than k.

Example

Permutation with arrows

$$\vec{3}\ \overleftarrow{2}\ \vec{4}\ \vec{1}$$

Elements 3 and **4 are mobile**, elements 2 and **1 are not mobile**.

Johnson-Trotter algorithm

Input : Natural number n

Output: List of all permutations of numbers $\{1, \ldots, n\}$

1 $\pi \leftarrow \overleftarrow{1}\ \overleftarrow{2}\ \ldots\ \overleftarrow{n}$;

2 **while** π contains a mobile element **do**

3 $\quad k \leftarrow$ largest mobile element in π;

4 $\quad$ swap in π the element k with its neighbor in the direction of the arrow;

5 $\quad$ change the direction of the arrow for all elements greater than k;

6 $\quad$ insert the newly created permutation (step 4) π into the resulting list;

7 **end**

8 **return** list of all permutations;

Johnson-Trotter algorithm – example

Example of generating permutations for $n = 3$

$$
\begin{array}{ccc}
\overleftarrow{1} & \overleftarrow{2} & \overleftarrow{3} \\
\overleftarrow{1} & \overleftarrow{3} & \overleftarrow{2} \\
\overleftarrow{3} & \overleftarrow{1} & \overleftarrow{2} \\
\overrightarrow{3} & \overleftarrow{2} & \overleftarrow{1} \\
\overleftarrow{2} & \overrightarrow{3} & \overleftarrow{1} \\
\overleftarrow{2} & \overleftarrow{1} & \overrightarrow{3}
\end{array}
$$

We say that an element k is **mobile** in a given permutation if the neighboring element in the direction of the arrow of element k is smaller than k.

Johnson-Trotter algorithm – example, $n = 4$

$\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{3}\,\overleftarrow{4}$

$\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{4}\,\overleftarrow{3}$

$\overleftarrow{1}\,\overleftarrow{4}\,\overleftarrow{2}\,\overleftarrow{3}$

$\overleftarrow{4}\,\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{3}$

$\overrightarrow{4}\,\overleftarrow{1}\,\overleftarrow{3}\,\overleftarrow{2}$

$\overleftarrow{1}\,\overrightarrow{4}\,\overleftarrow{3}\,\overleftarrow{2}$

$\overleftarrow{1}\,\overleftarrow{3}\,\overrightarrow{4}\,\overleftarrow{2}$

$\overleftarrow{1}\,\overleftarrow{3}\,\overrightarrow{2}\,\overleftarrow{4}$

$\overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{4}$

$\overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{4}\,\overleftarrow{2}$

$\overleftarrow{3}\,\overleftarrow{4}\,\overleftarrow{1}\,\overleftarrow{2}$

$\overleftarrow{4}\,\overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{2}$

$\overrightarrow{4}\,\overrightarrow{3}\,\overleftarrow{2}\,\overleftarrow{1}$

$\overrightarrow{3}\,\overrightarrow{4}\,\overleftarrow{2}\,\overleftarrow{1}$

$\overrightarrow{3}\,\overleftarrow{2}\,\overrightarrow{4}\,\overleftarrow{1}$

$\overrightarrow{3}\,\overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{4}$

$\overleftarrow{2}\,\overrightarrow{3}\,\overleftarrow{1}\,\overleftarrow{4}$

$\overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{3}\,\overleftarrow{4}$

$\overleftarrow{2}\,\overleftarrow{4}\,\overrightarrow{3}\,\overleftarrow{1}$

$\overleftarrow{4}\,\overleftarrow{2}\,\overrightarrow{3}\,\overleftarrow{1}$

$\overrightarrow{4}\,\overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{3}$

$\overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{4}\,\overrightarrow{3}$

$\overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{3}\,\overrightarrow{4}$

Johnson-Trotter algorithm

- One of the most efficient algorithms for generating permutations.
- The time complexity of the algorithm is $\Theta(n!)$.
- The ”fearsome” complexity of the algorithm, however, is not caused by the algorithm itself, which works very quickly. It is caused by the enormous number of permutations that must be generated...

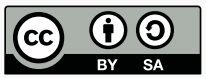
Thanks for your attention

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Divide and Conquer

Jiří Dvorský, Ph.D.

Department of Computer Science
VSB – Technical University of Ostrava



Divide and Conquer Solution Strategy

Master Theorem

Divide and Conquer

Multiplication of Large Integers

Multiplication of Large Integers

- Multiplication of ”ordinary” integers is handled by the processor.
- What about multiplying much larger numbers, with hundreds of digits? For example, in cryptography.
- Certainly, it would be possible to implement an algorithm similar to manual multiplication.
- Its implementation requires n^2 digit multiplications, where n is the number of digits.

$$\begin{array}{rrr} & 2 & 3 \\ & 1 & 4 \\ \hline & 9 & 2 \\ 2 & 3 & 0 \\ \hline 3 & 2 & 2 \end{array}$$

Question to Solve

Can this be done faster? Or is this the best possible algorithm?

Multiplication of large integers – multiplication of 23 and 14

We determine the decimal expansion of numbers

$$\begin{aligned} 23 &= 2 \cdot 10^1 + 3 \cdot 10^0 \\ 14 &= 1 \cdot 10^1 + 4 \cdot 10^0 \end{aligned}$$

And multiply both expansions with each other

$$\begin{aligned} 23 \times 14 &= \left(2 \cdot 10^1 + 3 \cdot 10^0\right) \times \left(1 \cdot 10^1 + 4 \cdot 10^0\right) \\ &= (2 \times 1) \cdot 10^2 + (2 \times 4 + 3 \times 1) \cdot 10^1 + (3 \times 4) \cdot 10^0 \\ &= 322 \end{aligned}$$

For the computation, we needed 4 multiplications (denoted by $\times$), i.e., n^2 multiplications.

Multiplication of large integers – multiplication of 23 and 14 (cont.)

The middle term (tens) can also be evaluated as follows

$$2 \times 4 + 3 \times 1 = (2 + 3) \times (1 + 4) - 2 \times 1 - 3 \times 4$$

Have we not seen the expressions 2×1 and 3×4 somewhere else?

More generally, let $a = a_1 a_0$ and $b = b_1 b_0$ then

$$c = a \times b = c_2 \cdot 10^2 + c_1 \cdot 10^1 + c_0,$$

where

- $c_2 = a_1 \times b_1$ is the product of the first digits,

Multiplication of large integers – multiplication of 23 and 14 (cont.)

- $c_0 = a_0 \times b_0$ is the product of the second digits and
- $c_1 = (a_1 + a_0) \times (b_1 + b_0) - (c_2 + c_0)$ is the product of the sums of digits a and b minus the digits c_2 and c_0.

Multiplication of large integers – divide and conquer

Let us have two n-digit numbers a and b, where n is an even natural number.

We will denote the first half of the digits of the number a as a_1, the second half as a_0. The notation $a = a_1 a_0$ will be understood as

$$a = a_1 a_0 = a_1 \cdot 10^{n/2} + a_0$$

A similar relationship holds for $b = b_1 b_0$.

Multiplication of large integers – divide and conquer

The product $c = a \times b$ can be written as

$$\begin{aligned} c &= \left(a_1 \cdot 10^{n/2} + a_0\right) \times \left(b_1 \cdot 10^{n/2} + b_0\right) \\ &= (a_1 \times b_1) \cdot 10^n + (a_1 \times b_0 + a_0 \times b_1) \cdot 10^{n/2} + (a_0 \times b_0) \\ &= c_2 \cdot 10^n + c_1 \cdot 10^{n/2} + c_0, \end{aligned}$$

where

- $c_2 = a_1 \times b_1$ is the product of the first halves,
- $c_0 = a_0 \times b_0$ is the product of the second halves and
- $c_1 = (a_1 + a_0) \times (b_1 + b_0) - (c_2 + c_0)$ is the product of the sums of halves of numbers a and b minus the sum of c_2 and c_0.

The numbers c_2, c_1 and c_0 are computed in the same way – recursive algorithm.

Termination of recursion: $n = 1$ or numbers a, b can be multiplied using hardware.

Multiplication of large integers – number of multiplications

- The number of multiplications necessary for computing the product of two n-digit numbers will be denoted as $M(n)$.
- Computing the product requires 3 multiplications of numbers of half the size. Multiplication of numbers for $n = 1$ requires one multiplication. Thus

$$\begin{aligned} M(n) &= 3M\left(\frac{n}{2}\right) \text{ for } n > 1 \\ M(1) &= 1 \end{aligned}$$

Multiplication of large integers – number of multiplications (cont.)

- By the method of backward substitution for $n = 2^k$ we get

$$\begin{aligned} M\left(2^k\right) &= 3M\left(2^{k-1}\right) = 3\left[3M\left(2^{k-2}\right)\right] = 3^2 M\left(2^{k-2}\right) \\ &\vdots \\ &= 3^i M(2^{k-i}) \\ &\vdots \\ &= 3^k M(2^{k-k}) = 3^k \end{aligned}$$

Multiplication of large integers – number of multiplications (cont.)

- Since $k = \log_2 n$ we further get

$$M(n) = 3^{\log_2 n} = n^{log_2 3} \approx n^{1,585}$$

Remarks

1. For logarithms, the property $a^{\log_b c} = c^{\log_b a}$ holds.
2. The recursion does not necessarily have to continue until $n = 1$, it can be stopped earlier and for small n the standard algorithm can be used.

Multiplication of large integers – number of additions and subtractions

- But what about addition and subtraction? Is the lower number of multiplications offset by a higher number of additions and multiplications?
- Let us denote $A(n)$ as the number of additions and subtractions when multiplying two n-digit numbers.
- In addition to $3A\left(\frac{n}{2}\right)$ operations necessary for the recursive computation of c_2, c_1 and c_0, we need 5 additions and 1 subtraction (marked in color on slide 435), so

$$\begin{aligned} A(n) &= 3A\left(\frac{n}{2}\right) + cn \text{ for } n > 1 \\ A(1) &= 1 \end{aligned}$$

Multiplication of large integers – number of additions and subtractions (cont.)

- According to the relation (**??**), the **Master theorem**, we get

$$A(n) \in \Theta\left(n^{\log_2 3}\right)$$

- The total number of additions and subtractions grows asymptotically at the same rate as the number of multiplications.

Multiplication of large integers – history

- The author of the algorithm is Soviet mathematician Anatolij Alexejevič Karacuba (1937 – 2008), who presented it in 1960.
- Until then, the prevailing opinion was that the traditional algorithm is asymptotically optimal.
- So it makes sense to deal with already “resolved” problems :-)
- The question is when to use the standard algorithm and when to use the algorithm based on the divide and conquer strategy.

Divide and Conquer

Strassen's Matrix Multiplication

Strassen's Matrix Multiplication

- Is brute force matrix multiplication the best possible strategy?
- The complexity of brute force multiplication is $\Theta(n^3)$.
- An asymptotically better algorithm was introduced by Volker Strassen in 1969.
- The initial "discovery" – multiplying square matrices of order 2 can be done with 7 multiplications, unlike 8 for brute force.

Strassen's matrix multiplication of order 2

$$\begin{pmatrix} c_{0,0} & c_{0,1} \\ c_{1,0} & c_{1,1} \end{pmatrix} = \begin{pmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{pmatrix} \times \begin{pmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{pmatrix}$$

$$= \begin{pmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{pmatrix}$$

$$\begin{aligned} m_1 &= (a_{0,0} + a_{1,1})(b_{0,0} + b_{1,1}) \\ m_2 &= (a_{1,0} + a_{1,1})b_{0,0} \\ m_3 &= a_{0,0}(b_{0,1} - b_{1,1}) \\ m_4 &= a_{1,1}(b_{1,0} - b_{0,0}) \end{aligned} \qquad \begin{aligned} m_5 &= (a_{0,0} + a_{0,1})b_{1,1} \\ m_6 &= (a_{1,0} - a_{0,0})(b_{0,0} + b_{0,1}) \\ m_7 &= (a_{0,1} - a_{1,1})(b_{1,0} + b_{1,1}) \end{aligned}$$

Strassen's Matrix Multiplication

- Operation counts for 2×2 matrices:

	Brute Force	Strassen
Number of multiplications	8	7
Number of additions and subtractions	4	18

- Multiplying 2×2 matrices in this way is obviously nonsense. But!

Strassen's Matrix Multiplication (cont.)

- We can reformulate the relationships to convert matrix multiplication of $n \times n$ matrices into submatrices of order $\frac{n}{2} \times \frac{n}{2}$ as follows:

$$\left(\begin{array}{c|c} C_{0,0} & C_{0,1} \\ \hline C_{1,0} & C_{1,1} \end{array}\right) = \left(\begin{array}{c|c} A_{0,0} & A_{0,1} \\ \hline A_{1,0} & A_{1,1} \end{array}\right) \times \left(\begin{array}{c|c} B_{0,0} & B_{0,1} \\ \hline B_{1,0} & B_{1,1} \end{array}\right)$$

- The submatrix $C_{0,0}$ can be computed either as

$$C_{0,0} = A_{0,0} \times B_{0,0} + A_{0,1} \times B_{1,0}$$

  or as

$$C_{0,0} = M_1 + M_4 - M_5 + M_7$$

- The matrices $M_1, \ldots M_7$ are computed in the same recursive manner.

Strassen's matrix multiplication – complexity analysis

The number of multiplications $M(n)$ for $n \times n$ matrices is given by the recursive equation:

$$\begin{aligned} M(n) &= 7M\left(\frac{n}{2}\right) \text{ for } n > 1 \\ M(1) &= 1 \end{aligned}$$

Assuming $n = 2^k$, we obtain

$$\begin{aligned} M\left(2^k\right) &= 7M\left(2^{k-1}\right) = 7\left[7M\left(2^{k-2}\right)\right] = 7^2 M\left(2^{k-2}\right) \\ &\vdots \\ &= 7^i M\left(2^{k-i}\right) \\ &\vdots \\ &= 7^k M(2^{k-k}) = 7^k. \end{aligned}$$

Since $k = \log_2 n$ and thus

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807} < n^3$$

Strassen's matrix multiplication – complexity analysis, addition

- But does the number of additions $A(n)$ for $n \times n$ matrices not grow too quickly?
- For multiplying $n \times n$ matrices we need:
  1. to compute 7 submatrices of order $\frac{n}{2} \times \frac{n}{2}$ and
  2. to perform 18 additions/subtractions of submatrices of order $\frac{n}{2} \times \frac{n}{2}$.

  So

$$\begin{aligned} A(n) &= 7A\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 \text{ for } n > 1 \\ A(1) &= 0 \end{aligned}$$

- According to the relation (??), **Master theorem**, we get

$$A(n) \in \Theta\left(n^{\log_2 7}\right)$$

- It follows that Strassen's matrix multiplication has an asymptotic complexity of $\Theta\left(n^{\log_2 7}\right)$, which is less than the brute force solution.

Divide and Conquer

Closest Pair Problem

Divide and Conquer

Convex hull of a set

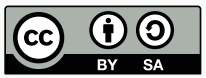
Thanks for your attention

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Algorithms II – Subject Syllabus

Jiří Dvorský, Ph.D.

Department of Computer Science
VSB – Technical University of Ostrava



Algorithms II – Subject Syllabus

About Algorithms II

Algorithms II – Subject Syllabus

Software

Software

Primary Software

- C++ Development Environment
- C++ Documentation

Additional Software

- Doxygen Documentation System, *www.doxygen.org*
- Typography System LaTeX, *www.ctan.org*

Development Environment for C++

- Microsoft Visual Studio Community 2022 is available for classroom use.
- I recommend this development environment for home study.
- In general, any development environment with a compiler that supports at least the **C++17** specification can be used.

Development Environment for C++ (cont.)

Remarks

1. The **Microsoft Visual C++** compiler and the **C++17** language specification will be used to evaluate your projects.
2. The C language is not identical to C++!
3. Beware of non-standard C++ language extensions implemented in the GNU C++ compiler.
   - For example, a variable length array is such an extension.
   - It is recommended to compile with the `-pedantic-errors` option enabled, see Options to Request or Suppress Warnings.

Algorithms II – Subject Syllabus

Study Literature

Study Literature

The study literature can be divided into two groups:

- **mandatory literature** – strategies of algorithmic problems solving and
- **recommended literature** – C++ programming language.

The study literature is shared across Algorithms I and Algorithms courses.

Mandatory Study Literature

1. LEVITIN, Anany. *Introduction to the Design and Analysis of Algorithms*. 3rd ed. Boston: Pearson, 2012. ISBN 978-0-13-231681-1.
2. CORMEN, Thomas H., Charles Eric LEISERSON, Ronald L. RIVEST a Clifford STEIN, 2022. *Introduction to algorithms*. Fourth edition. Cambridge, Massachusetts: The MIT Press. ISBN 978-026-2046-305.
3. SEDGEWICK, Robert, 1998. *Algorithms in C++*. 3rd ed. Reading, Mass: Addison-Wesley. ISBN 978-020-1350-883.

Recommended study literature

1. STROUSTRUP, Bjarne., 2013. The C++ programming language. Fourth edition. Upper Saddle River, NJ: Addison-Wesley. ISBN 978-0321563842.
2. CADENHEAD, Rogers a Jesse LIBERTY, 2017. Sams teach yourself C in 24 hours. Sixth edition. Indianapolis, Indiana: Pearson Education. ISBN 978-0672337468.

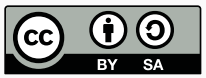
Thanks for your attention

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Transform and Conquer

Jiří Dvorský, Ph.D.

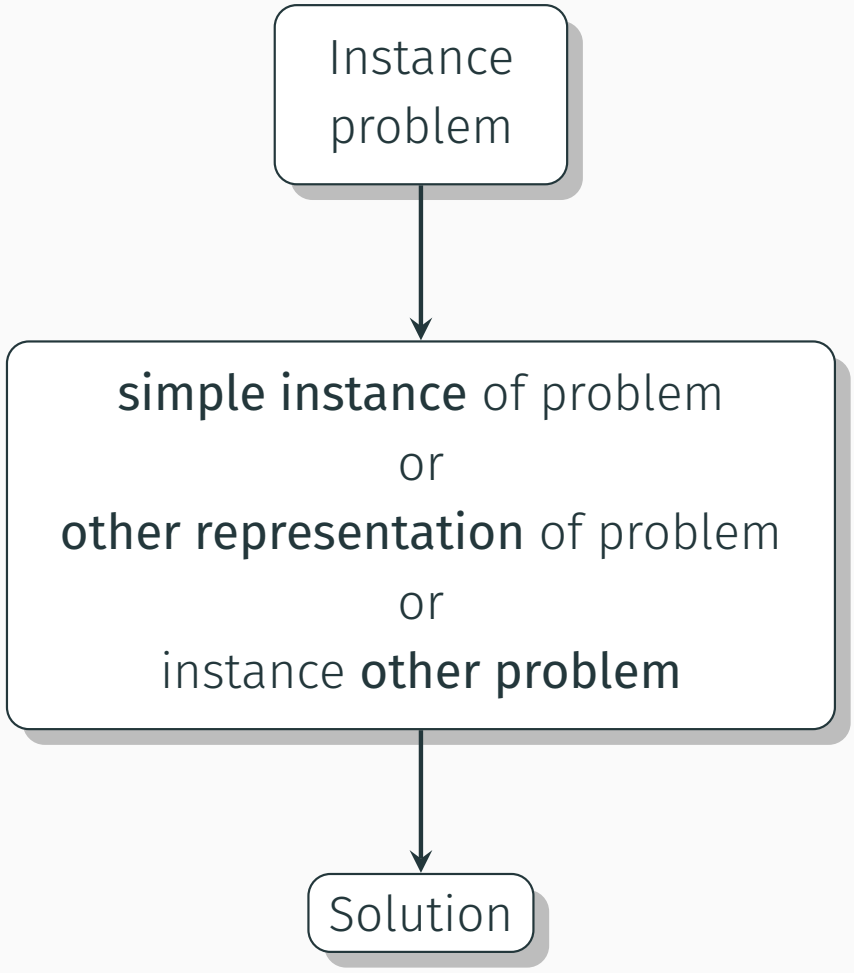
Department of Computer Science
VSB – Technical University of Ostrava



Solution strategy transform and solve

Biphasic strategy

1. transformation
2. solution



Transform and Conquer

Presorting

Data sorting

- A relatively old idea that motivated, among other things, research into sorting algorithms.
- Sorted data lead to significantly simpler algorithms, “order must be”.
- Prerequisites:
  1. data is stored in an array – sorting an array is easier than sorting a list **for s do**

l o

end

rting we use an algorithm with complexity $\Theta(n \log n)$ – typically QuickSort, MergeSort.
- Usage: geometric algorithms, graph algorithms, caustic algorithms.

Unity of elements in the array

Background

We are given an array A with n elements. We have to determine whether each element occurs exactly once in the array A.

Rough force solution – compare all pairs of elements until:

1. does not find a pair of the same elements or
2. tested all pairs of elements.

The time complexity is in the worst case $\Theta(n^2)$.

Unity of elements in the array

ALGORITHM *PresortElementUniqueness*$(A[0..n-1])$

//Solves the element uniqueness problem by sorting the array first
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Returns "true" if A has no equal elements, "false" otherwise
sort the array A
for $i \leftarrow 0$ **to** $n-2$ **do**
    if $A[i] = A[i+1]$ **return false**
return true

Algorithm time complexity

$$T(n) = T_{sort}(n) + T_{scan}(n) \in \Theta(n \log n) + \Theta(n) = \Theta(n \log n)$$

Module count

Background

We are given an array A with n elements. We have to determine which element occurs most often in the array. This element is called **modus**.

For simplicity, we will assume that there is only one modus in the array A.

Rough force solution

For each element $a_i \in A$, search the auxiliary list L:

1. If we find a match, we increment the corresponding frequency,
2. otherwise, insert the element a_i at the end of the list with frequency 1.

Mod calculation – time complexity of brute force solution

- Worst case – all elements in array A are different.
- For a_i we have to do $i - 1$ comparison with elements in the list L before we add a new element to the end of it.
- The number of comparisons is equal to

$$C(n) = \sum_{i=1}^{n} (i - 1) = 0 + 1 + \cdots + (n - 1) = \frac{1}{2} n(n - 1) \in \Theta(n^2)$$

- Finding the maximum requires $n - 1$ comparisons, which does not affect the quadratic complexity of the algorithm.

Mod calculation – data presort

- If we sort the array A, the identical elements in the array A will be next to each other.
- To calculate the mode, it is enough to find the longest run of identical elements in A.
- Time complexity

$$T(n) = T_{sort}(n) + T_{scan}(n) \in \Theta(n \log n) + \Theta(n) = \Theta(n \log n)$$

ALGORITHM *PresortMode*$(A[0..n-1])$

//Computes the mode of an array by sorting it first
//Input: An array $A[0..n-1]$ of orderable elements
//Output: The array's mode
sort the array A
$i \leftarrow 0$ //current run begins at position i
$modefrequency \leftarrow 0$ //highest frequency seen so far
while $i \leq n-1$ **do**
    $runlength \leftarrow 1$; $runvalue \leftarrow A[i]$
    while $i + runlength \leq n-1$ **and** $A[i+runlength] = runvalue$
        $runlength \leftarrow runlength + 1$
    if $runlength > modefrequency$
        $modefrequency \leftarrow runlength$; $modevalue \leftarrow runvalue$
    $i \leftarrow i + runlength$
return $modevalue$

Search for element x in array A of length n

- The brute force solution leads to an algorithm requiring n comparisons in the worst case.
- After sorting the array, the interval halving algorithm can be used, which requires $\lfloor \log_2 n \rfloor + 1$ comparison in the worst case.
- The time complexity of the algorithm will then be

  $$T(n) = T_{sort}(n) + T_{search}(n) = \Theta(n \log n) + \Theta(\log n) = \Theta(n \log n),$$

  which is **more** than the complexity of sequential search!!!
- But for **repeated** searches it is already worth sorting the A field.

Resources for self-study

- Book [2], chapter 6.1, pages 202 – 205

Transform and Conquer

Gaussian Elimination Method

Gaussian Elimination Method – Motivation

A system of two equations with two unknowns

$$\begin{aligned} a_{11}x + a_{12}y &= b_1 \\ a_{21}x + a_{22}y &= b_2 \end{aligned}$$

can be solved relatively easily – for example, we can express the variable x as a function of y, substitute it into the second equation, and solve the equation.

Problem

How to solve a system of n equations with n unknowns? In the same way?

Gaussian elimination method

System of n linear equations with n unknowns

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

is transformed into an equivalent system of equations, where all coefficients below the main diagonal are zero

$$\begin{aligned} a'_{11}x_1 + a'_{12}x_2 + \cdots + a'_{1n}x_n &= b'_1 \\ a'_{22}x_2 + \cdots + a'_{2n}x_n &= b'_2 \\ &\vdots \\ a'_{nn}x_n &= b'_n \end{aligned}$$

Gaussian Elimination Method – Matrix Notation

$$\mathbf{A}\vec{x} = \vec{b} \quad \Longrightarrow \quad \mathbf{A}'\vec{x} = \vec{b}'$$

where

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \qquad \vec{b} = \begin{pmatrix} b_{11} \\ b_{21} \\ \vdots \\ b_{n1} \end{pmatrix}$$

$$\mathbf{A}' = \begin{pmatrix} a'_{11} & a_{12} & \cdots & a'_{1n} \\ 0 & a'_{22} & \cdots & a'_{2n} \\ \vdots & & & \vdots \\ 0 & 0 & \cdots & a'_{nn} \end{pmatrix} \qquad \vec{b}' = \begin{pmatrix} b'_{11} \\ b'_{21} \\ \vdots \\ b'_{n1} \end{pmatrix}$$

$\mathbf{A}'$ is called the **upper triangular matrix**.

Gaussian Elimination Method – Advantages of Representation Change

A system given by an upper triangular matrix can be easily solved using **back substitution**:

1. From the equation
$$a'_{nn} x_n = b'_n$$
we compute the unknown x_n.
2. We substitute the value of the unknown x_n into the equation
$$a'_{n-1\,n-1} x_{n-1} + a'_{n-1\,n} x_n = b'_{n-1}$$
and compute the unknown x_{n-1}.
3. We proceed in this manner until we compute the unknown x_1.

The complexity of this algorithm is $\Theta(n^2)$.

Gaussian elimination method – elementary operations

The matrix of the system $\mathbf{A}$ is transformed into an upper triangular matrix $\mathbf{A}'$ using **elementary operations**:

- swapping two equations in the system,
- multiplying an equation by a non-zero coefficient and
- adding or subtracting a multiple of another equation to the given equation, i.e. a linear combination with another equation.

Elementary operations do not change the solution of the system of equations – the transformed system has the same solution as the original system.

Gaussian Elimination Method – Matrix Transformation

1. We choose a_{11} as the **pivot** and "nullify" all coefficients in the first column, except for a_{11}.

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

   "Nullification" – from the second equation, we subtract $\frac{a_{21}}{a_{11}}$ times the first equation, from the third equation, we subtract $\frac{a_{31}}{a_{11}}$ times the first equation, and so on.
2. We choose a_{22} as the pivot and repeat the same procedure.

Remark

Of course, we also perform changes for the vector of right-hand sides $\vec{b}$.

Gaussian Elimination Method – Example

Let us have a system of equations

$$\begin{aligned} 2x_1 - x_2 + x_3 &= 1 \\ 4x_1 + x_2 - x_3 &= 5 \\ x_1 + x_2 + x_3 &= 0 \end{aligned}$$

The augmented matrix of the system

$$\left(\begin{array}{ccc|c} 2 & -1 & 1 & 1 \\ 4 & 1 & -1 & 5 \\ 1 & 1 & 1 & 0 \end{array}\right)$$

Gaussian Elimination Method – Example (cont.)

Forward Elimination

From the second row, we subtract $\frac{4}{2}$ times the first row, from the third row, we subtract $\frac{1}{2}$ times the first row

$$\left(\begin{array}{ccc|c} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & \frac{3}{2} & \frac{1}{2} & -\frac{3}{2} \end{array}\right)$$

From the third row, we subtract $\frac{\frac{3}{2}}{3} = \frac{1}{2}$ times the second row

$$\left(\begin{array}{ccc|c} 2 & -1 & 1 & 1 \\ 0 & 3 & -3 & 3 \\ 0 & 0 & 2 & -2 \end{array}\right)$$

Gaussian Elimination Method – Example (cont.)

Back Substitution

$$
\begin{aligned}
x_3 &= \frac{-2}{2} = -1 \\
x_2 &= \frac{3 - (-3)x_3}{3} = \frac{3 - (-3)(-1)}{3} = 0 \\
x_1 &= \frac{1 - x_3 - (-1)x_2}{2} = \frac{1 - (-1)}{2} = 1
\end{aligned}
$$

Gaussian elimination method – forward elimination

Input : Matrix $\mathbf{A}$ of type $n \times n$ and column vector $\vec{b}$ of dimension n

Output: Equivalent triangular matrix $\mathbf{A}$ and vector $\vec{b}$

```
1 for i ← 1 to n - 1 do
2 |   for j ← i + 1 to n do
3 |   |   temp ← A[j, i]/A[i, i];
4 |   |   for k ← i to n do
5 |   |   |   A[j, k] ← A[j, k] - A[i, k] * temp;
6 |   |   end
7 |   |   b[j] ← b[j] - b[i] * temp;
8 |   end
9 end
```

Partial Pivoting

- In the forward elimination algorithm, there is an error. If $a_{ii} = 0$, then division by zero occurs.
- The problem can be solved by swapping equations (elementary operation) so that $a_{ii} \neq 0$.
- It is also possible to simultaneously address potential rounding errors – the pivot is chosen such that it is the largest of all elements a_{ii} to a_{ni} in absolute value.

Gaussian elimination method – partial pivoting

ALGORITHM *BetterForwardElimination*$(A[1..n, 1..n], b[1..n])$

//Implements Gaussian elimination with partial pivoting
//Input: Matrix $A[1..n, 1..n]$ and column-vector $b[1..n]$
//Output: An equivalent upper-triangular matrix in place of A and the
//corresponding right-hand side values in place of the $(n+1)$st column
for $i \leftarrow 1$ **to** n **do** $A[i, n+1] \leftarrow b[i]$ //appends b to A as the last column
for $i \leftarrow 1$ **to** $n-1$ **do**
    $pivotrow \leftarrow i$
    for $j \leftarrow i+1$ **to** n **do**
        if $|A[j, i]| > |A[pivotrow, i]|$ $pivotrow \leftarrow j$
    for $k \leftarrow i$ **to** $n+1$ **do**
        $swap(A[i, k], A[pivotrow, k])$
    for $j \leftarrow i+1$ **to** n **do**
        $temp \leftarrow A[j, i] / A[i, i]$
        for $k \leftarrow i$ **to** $n+1$ **do**
            $A[j, k] \leftarrow A[j, k] - A[i, k] * temp$

Gaussian Elimination Method – Time Complexity

- Input size – number of equations in the system, i.e., dimension of matrix n.
- Basic operation – arithmetic operations, for historical reasons multiplication. In the innermost cycle, the number of multiplications corresponds to the number of subtractions, it's just a multiple of a constant 2.
- We will be interested in the number of multiplications $C(n)$ depending on the number n.

Gaussian Elimination Method – Time Complexity (cont.)

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

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

The last sum is expanded for individual i

$$\begin{array}{rrcl}
i = 1 & (n - 1 + 1)(n - 1) & = & n(n - 1) \\
i = 2 & (n - 2 + 1)(n - 2) & = & (n - 1)(n - 2) \\
\vdots & \vdots & \vdots & \vdots \\
i = n - 2 & (n - n + 2 + 1)(n - n + 2) & = & 3 \cdot 2 \\
i = n - 1 & (n - n + 1 + 1)(n - n + 1) & = & 2 \cdot 1
\end{array}$$

Gaussian Elimination Method – Time Complexity (cont.)

From the last column, it is clear that this is a sum of a series

$$1 \cdot 2 + 2 \cdot 3 + \cdots + (n-2)(n-1) + (n-1)n = \sum_{l=1}^{n-1} l(l+1)$$

$$\begin{aligned}
\sum_{l=1}^{n-1} l(l+1) &= \sum_{l=1}^{n-1} l^2 + \sum_{l=1}^{n-1} l \\
&= \frac{1}{6}n(n-1)(2n-1) + \frac{1}{2}n(n-1) \\
&= \frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n + \frac{1}{2}n^2 - \frac{1}{2}n \\
&= \frac{1}{3}n^3 - \frac{1}{3}n
\end{aligned}$$

Gaussian Elimination Method – Time Complexity (cont.)

And therefore

$$C(n) = \frac{1}{3}n^3 - \frac{1}{3}n \approx \frac{1}{3}n^3 \in \Theta(n^3)$$

Since the complexity of back substitution is $\Theta(n^2)$, the complexity of the entire Gaussian elimination method is $\Theta(n^3)$.

LU-decomposition of a matrix

Let us have the matrix $\mathbf{A}$ of the system of linear equations from the previous example

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 1 \\ 4 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix}$$

Further, let us consider two matrices:

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{pmatrix}$$

Coefficients from Gaussian elimination

$$\mathbf{U} = \begin{pmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{pmatrix}$$

Result of Gaussian elimination

LU-decomposition of a matrix

Definition

Let $\mathbf{A}$ be a regular square matrix with elements from the real numbers, for which it is not necessary to swap rows during Gaussian elimination. Then there exist regular matrices $\mathbf{L}$ and $\mathbf{U}$, which are uniquely determined and satisfy the following statement

$$\mathbf{A} = \mathbf{LU},$$

where $\mathbf{L}$ is a lower triangular matrix with ones on the entire main diagonal and $\mathbf{U}$ is an upper triangular matrix with non-zero elements on the main diagonal.

Solution of a system of equations by *LU* decomposition

Let us have a system of linear equations

$$\mathbf{A}\vec{x} = \vec{b}$$

We replace matrix $\mathbf{A}$ with its *LU* decomposition

$$\mathbf{L}\mathbf{U}\vec{x} = \vec{b}$$

Furthermore, let us denote the product $\mathbf{U}\vec{x} = \vec{y}$. After substitution, we obtain a system of equations

$$\mathbf{L}\vec{y} = \vec{b}$$

This system can be easily solved because $\mathbf{L}$ is a lower triangular matrix. And finally, we can also easily solve the system

$$\mathbf{U}\vec{x} = \vec{y},$$

because $\mathbf{U}$ is an upper triangular matrix.

Solution of a system of equations by *LU* decomposition, example

We have a system of equations

$$\begin{aligned} 2x_1 - x_2 + x_3 &= 1 \\ 4x_1 + x_2 - x_3 &= 5 \\ x_1 + x_2 + x_3 &= 0 \end{aligned}$$

We perform the ***LU*** decomposition of the system matrix $\mathbf{A}$

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 1 \\ 4 & 1 & -1 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{pmatrix} \begin{pmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{pmatrix}$$

Solution of a system of equations by *LU* decomposition, example (cont.)

First, we solve the system $\mathbf{L}\vec{y} = \vec{b}$

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 5 \\ 0 \end{pmatrix}$$

$$\begin{aligned} y_1 &= 1 \\ y_2 &= 5 - 2y_1 = 3 \\ y_3 &= 0 - \frac{1}{2}y_1 - \frac{1}{2}y_2 = -2 \end{aligned}$$

Solution of a system of equations by *LU* decomposition, example (cont.)

Subsequently, we solve the system $\mathbf{U}\vec{x} = \vec{y}$

$$\begin{pmatrix} 2 & -1 & 1 \\ 0 & 3 & -3 \\ 0 & 0 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ -2 \end{pmatrix}$$

$$\begin{aligned} x_3 &= \frac{-2}{2} = -1 \\ x_2 &= \frac{3 - (-3)x_3}{3} = \frac{3 - (-3)(-1)}{3} = 0 \\ x_1 &= \frac{1 - x_3 - (-1)x_2}{2} = \frac{1 - (-1)}{2} = 1 \end{aligned}$$

LU-decomposition of a matrix, notes

- In practice, *LU*-decomposition is used to solve systems of linear equations.
- Using *LU*-decomposition, it is possible to efficiently solve multiple systems of equations with the same system matrix.
- The matrices $\mathbf{L}$ and $\mathbf{U}$ can be stored together in one matrix – from the matrix $\mathbf{L}$ we store only the elements below the diagonal. Why?
- If it is necessary to perform partial pivoting in the matrix $\mathbf{A}$, i.e., to swap rows, then the decomposition has the form

$$\mathbf{PA} = \mathbf{LU}$$

and from this

$$\mathbf{A} = \mathbf{P}^{-1}\mathbf{LU},$$

where $\mathbf{P}$ is a permutation matrix.

Permutation Matrix

- Represents a permutation of n elements as a matrix
- A square binary matrix of order n, with one 1 in each row and column, and the rest 0
- For every permutation matrix $\mathbf{P}$ applies:
  - left multiplication, $\mathbf{PM}$, results in a permutation of the rows of matrix $\mathbf{M}$, where $\mathbf{M}$ is a matrix with n rows
  - right multiplication, $\mathbf{MP}$, results in a permutation of the columns of matrix $\mathbf{M}$, where $\mathbf{M}$ is a matrix with n columns
  - $\mathbf{P}$ is orthogonal, i.e. its inverse matrix is equal to its transpose, $\mathbf{P}^{-1} = \mathbf{P}^{T}$

Permutation matrix, example

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 2 & 4 & 1 \end{pmatrix} \quad \leftrightarrow \quad R_\pi = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

$$\updownarrow \qquad\qquad \updownarrow$$

$$C_\pi = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad \leftrightarrow \quad \pi^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 2 & 1 & 3 \end{pmatrix}$$

Transform and Conquer

Balanced Search Trees

Binary Search Trees – review

- Fundamental data structure for implementing sets, dictionaries etc.
- Each node contains one key; a total order must be defined over the keys.
- For each node, all keys in the left subtree are smaller than the key in the given node and in the right subtree are all keys greater.
- Average time complexity of search, insertion, and deletion operations is $\Theta(\log_2 n)$.
- Worst-case scenario is however still $\Theta(n)$ – the tree degenerates into a list.

Balanced Search Trees

Possible solution for the worst case:

Proactive Measures

- transformation into a balanced binary tree using rotations
- various definitions of balance
- AVL trees, red-black trees, splay trees.

Representation Change

- multiple keys in one node,
- 2-3 trees, 2-3-4 trees, B-trees.

AVL Trees

Authors

- Georgij Maximovič Adelson-Velskij and
- Jevgenij Michajlovič Landis

First published in 1962.

Definition

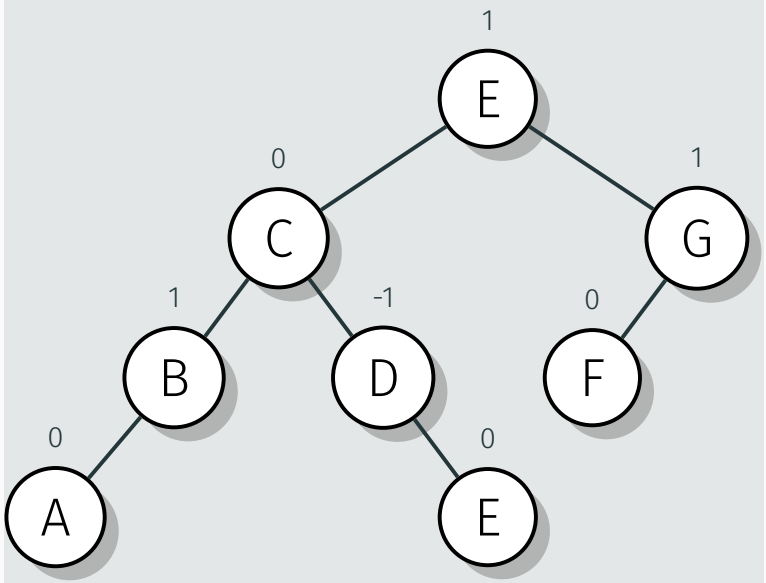
The **balance factor** of a node u is the difference between the heights of its left and right subtrees. The height of an empty tree is defined as -1.

Definition

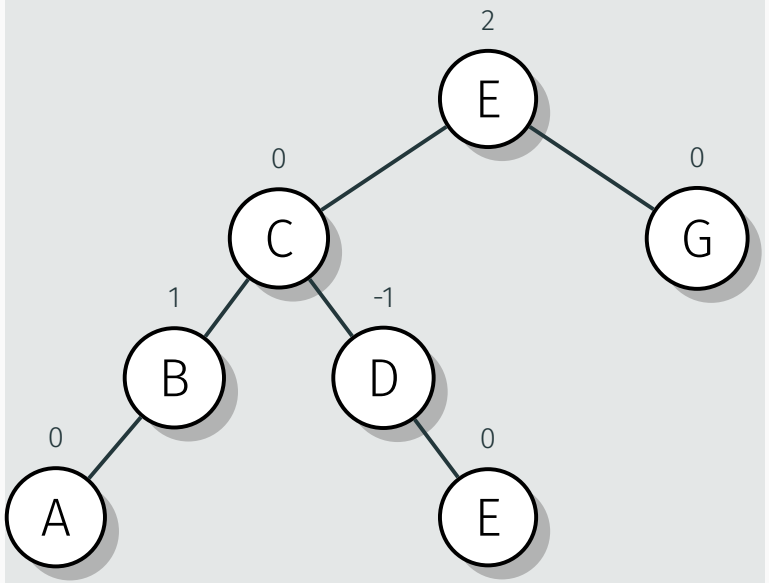
A binary search tree is called an **AVL tree** if and only if the balance factor for each node in the tree is either -1, 0, or +1.

AVL trees – example

AVL tree



This is not an AVL tree

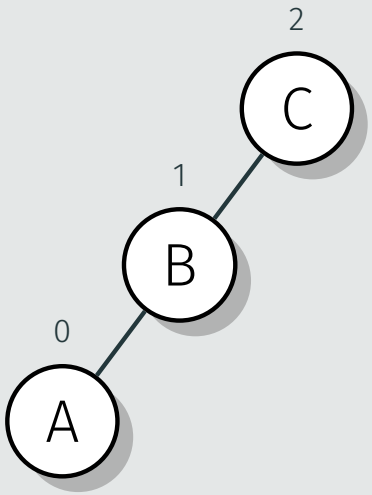


AVL trees – maintaining balance

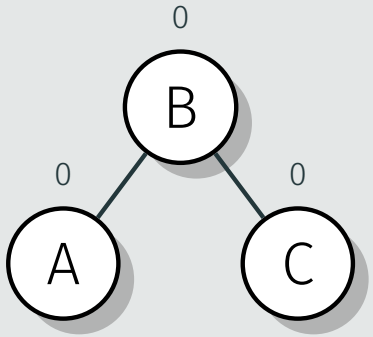
- Insertion of a new node, or deletion of an existing one, can cause imbalance in the AVL tree.
- Balance must be restored after each such operation.
- Balance is restored using **rotations**.
- Rotation is a local transformation of the tree at those nodes where the balance factor reaches a value of -2 or 2.
- If there are multiple such nodes, we always start with the node at the lowest level (closest to the leaves of the tree) and proceed upwards towards the root of the tree.
- There are a total of four rotations – two pairs of mutually mirror-symmetric rotations.

Simple rotations

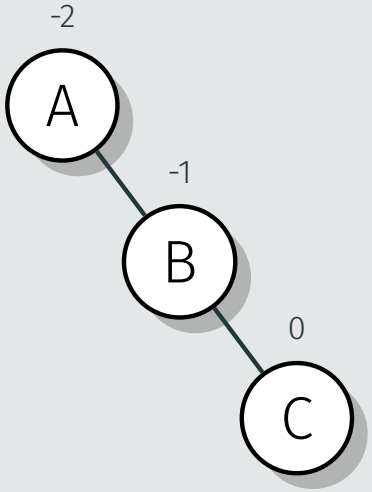
Right rotation



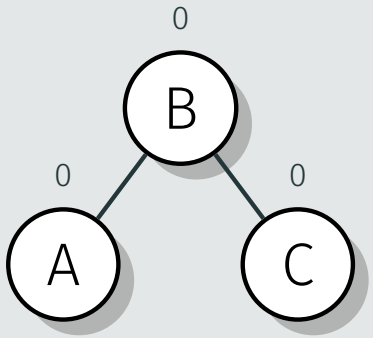
Operation result



Left rotation

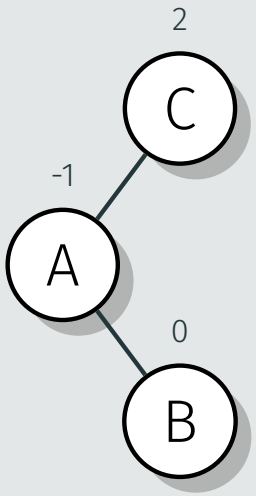


Operation result

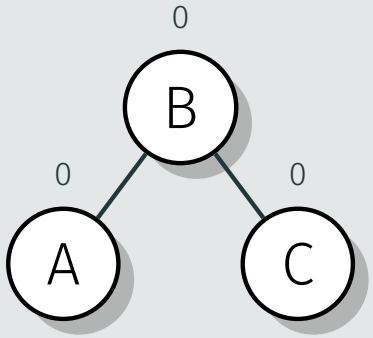


Double rotations

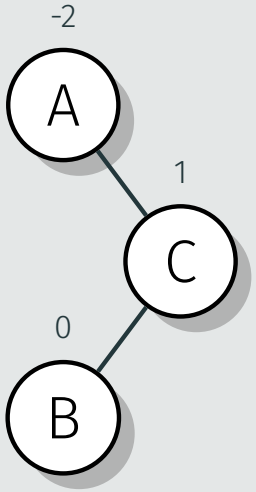
Left-Right rotation



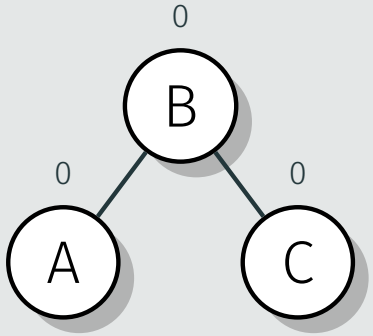
Operation result



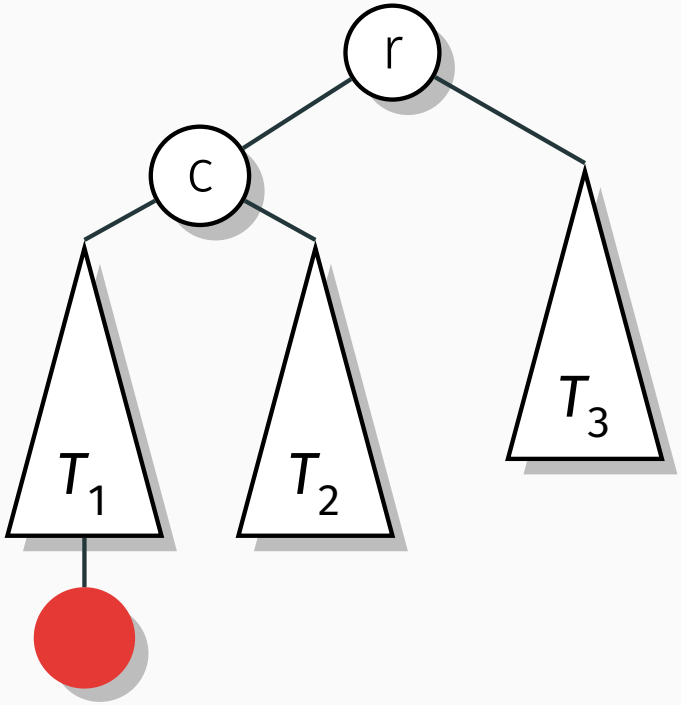
Right-Left rotation



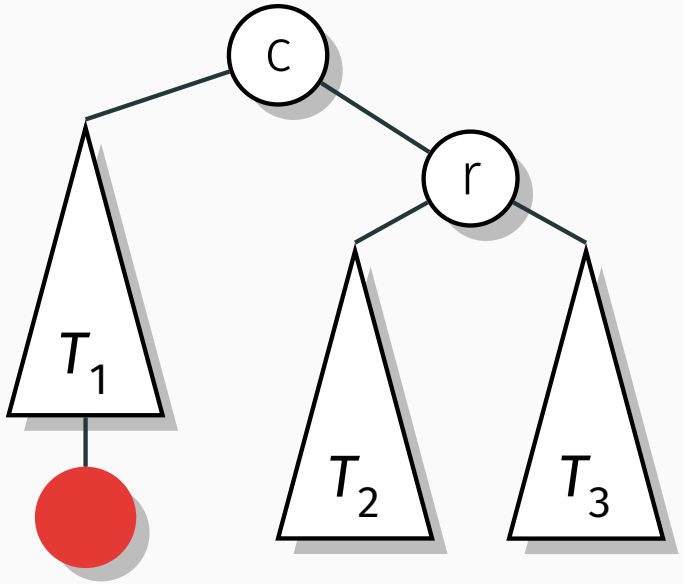
Operation result



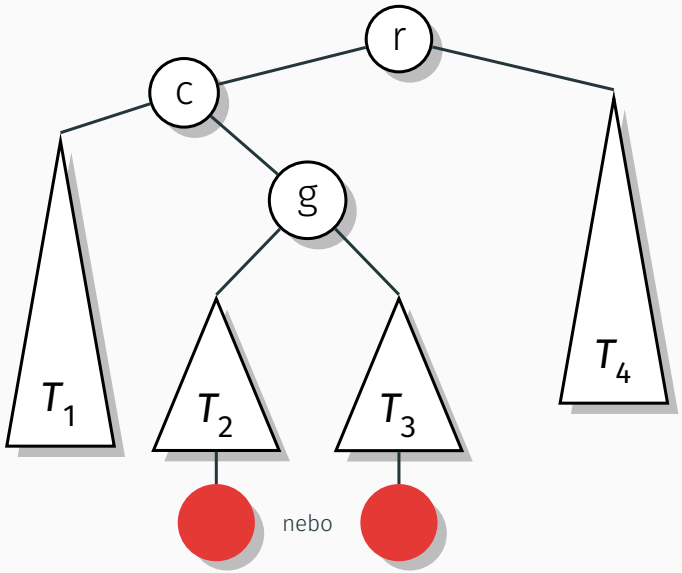
AVL trees – general scheme of right rotation



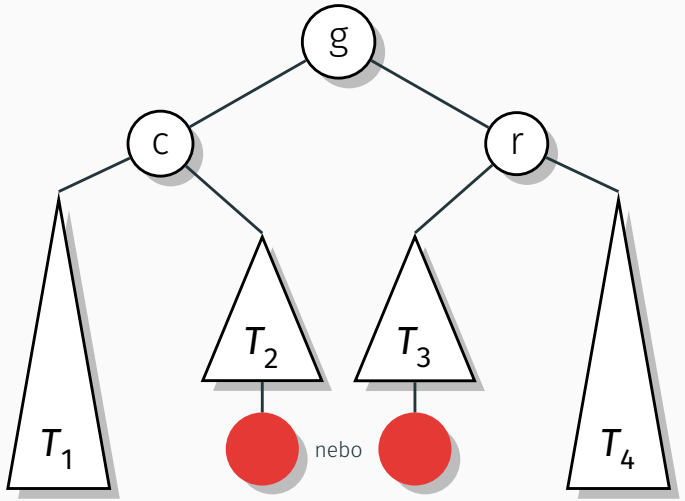
Operation result



AVL trees – general scheme of LR rotation



Operation result

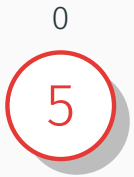


AVL trees – properties of rotations

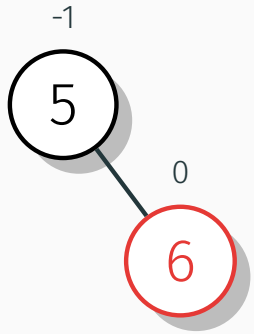
- Constant time complexity – only pointers between nodes are moved, not data.
- Rotations preserve the ordering of keys in the tree – after completing a rotation, the “left” side always contains smaller keys, the “right” side always contains larger keys.

AVL Trees – Sequential Construction of the Tree

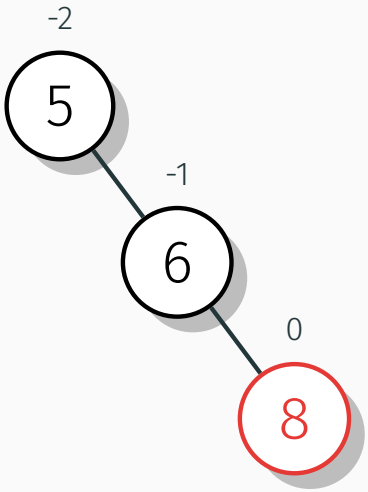
Insertion of 5



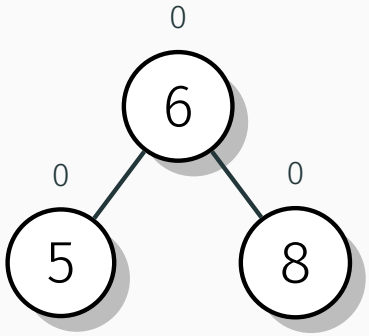
Insertion of 6



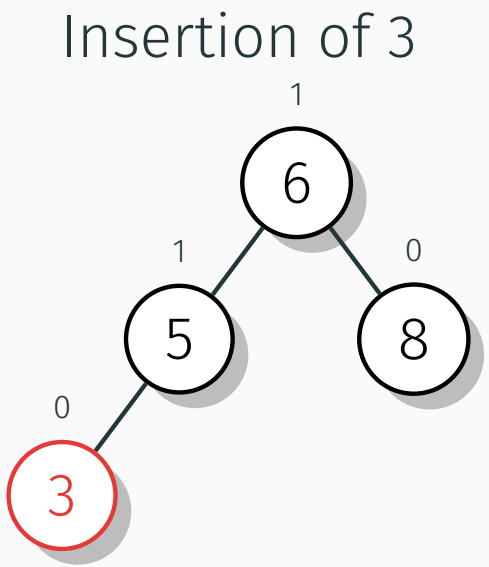
Insertion of 8



Left Rotation of 5

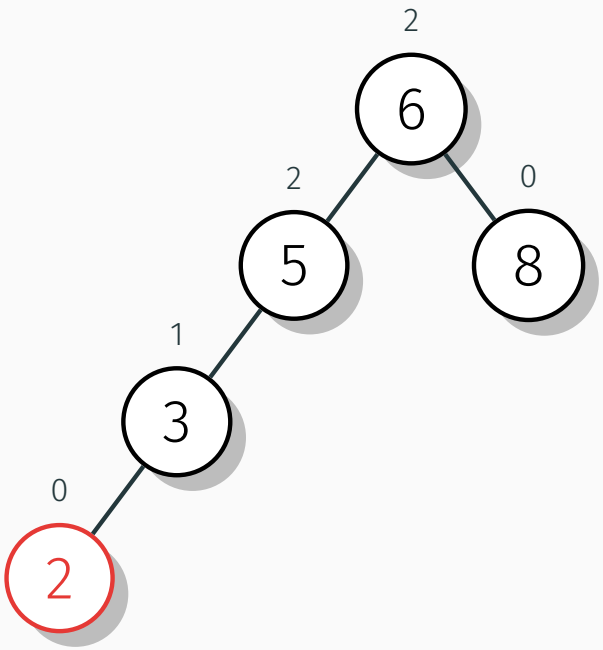


AVL Trees – Sequential Construction of the Tree (cont.)

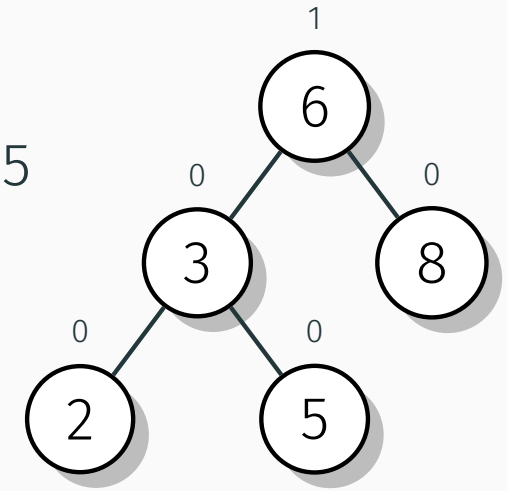


AVL Trees – Sequential Construction of the Tree (cont.)

Insertion of 2

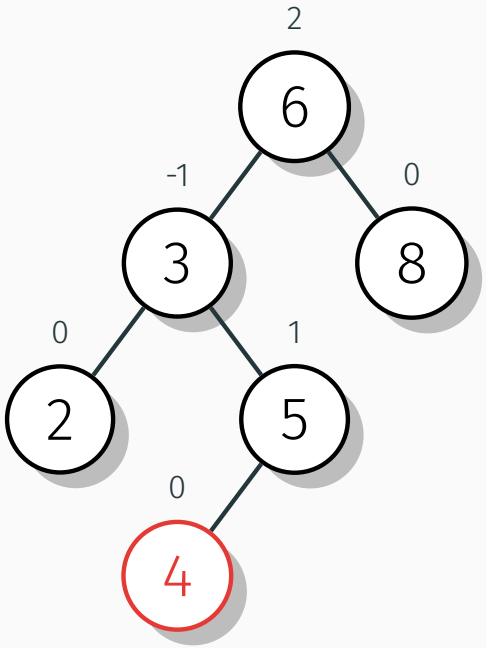


Right Rotation of 5

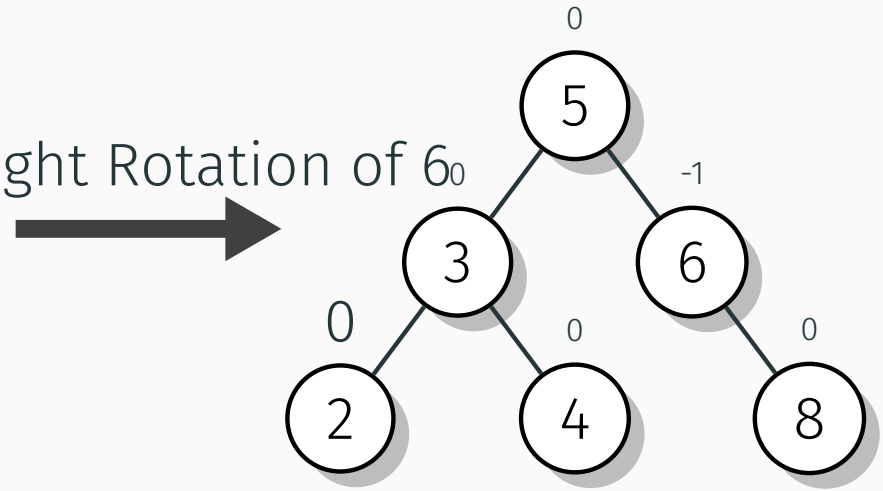


AVL Trees – Sequential Construction of the Tree (cont.)

Insertion of 4

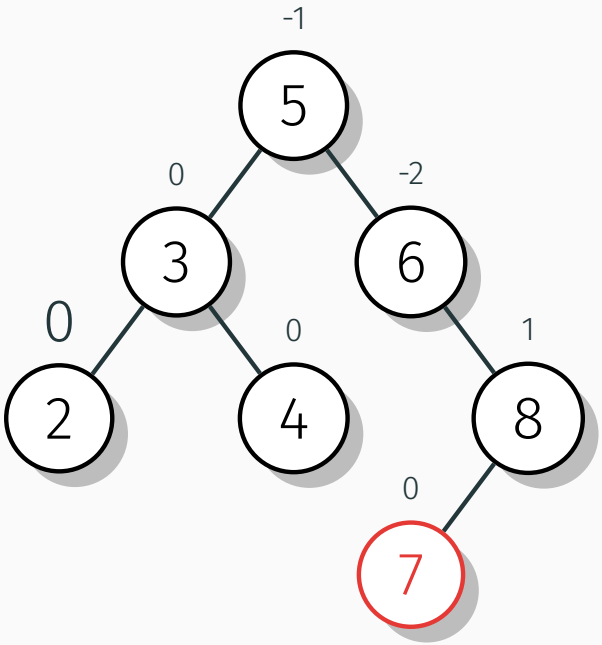


Left-Right Rotation of 6

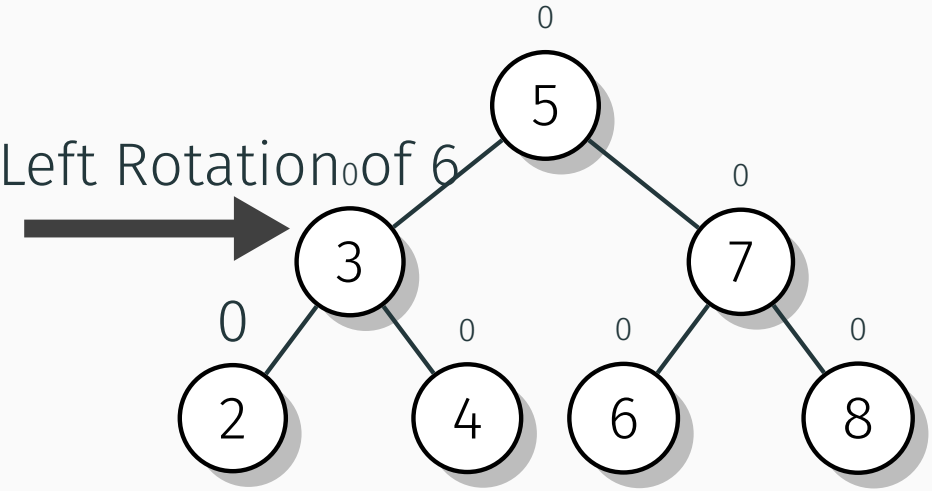


AVL Trees – Sequential Construction of the Tree (cont.)

Insertion of 7



Right-Left Rotation of 6



AVL trees – properties

- The height of an AVL tree with n nodes is bounded by

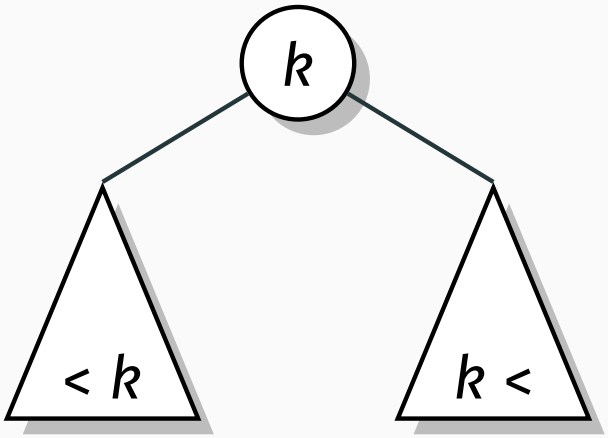
$$\lfloor \log_2 n \rfloor \leq h < 1.4405 \log_2(n+2) - 1.3277$$

- Search and insertion operations therefore proceed with a complexity of $\Theta(\log_2 n)$ even in the worst case.
- The average height of an AVL tree constructed from a random sequence of n keys is $1.01 \log_2 n + 0.1$.
- Node deletion is more complicated, but still falls within the logarithmic complexity class.
- Disadvantages – a large number of rotations during tree balancing.

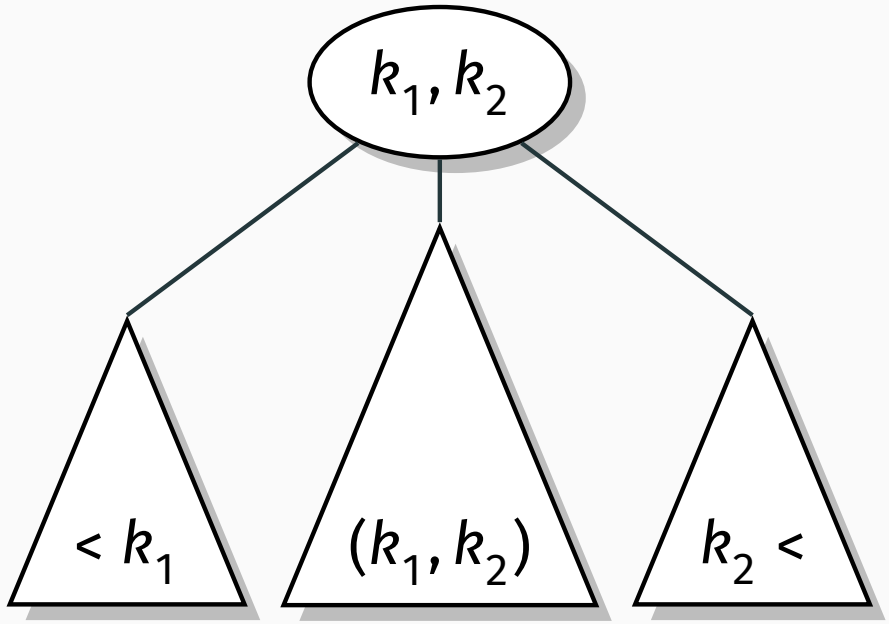
2-3 trees

2-3 trees – types of nodes

2-node



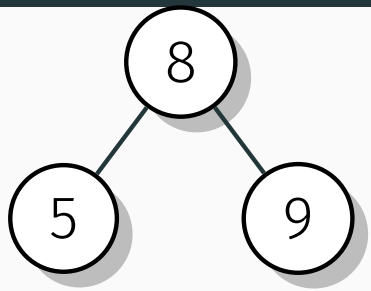
3-node

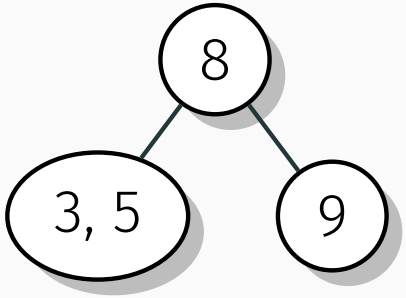


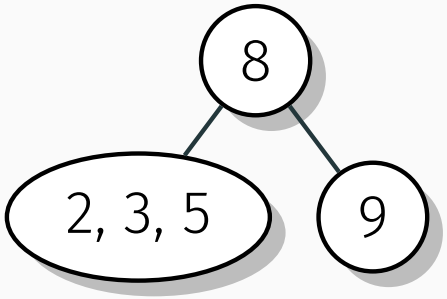
Construction of a 2-3 tree from the sequence 9, 5, 8, 3, 2, 4, 7

Operation

Result

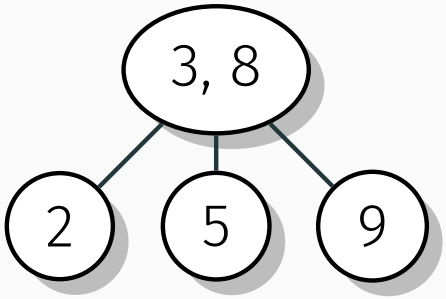


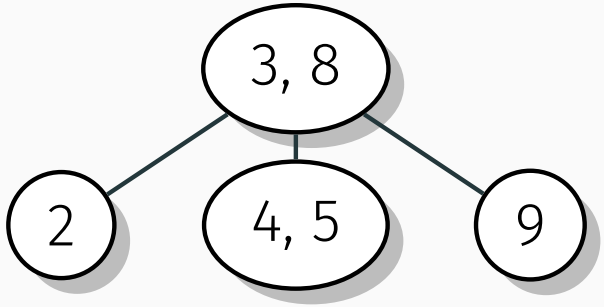




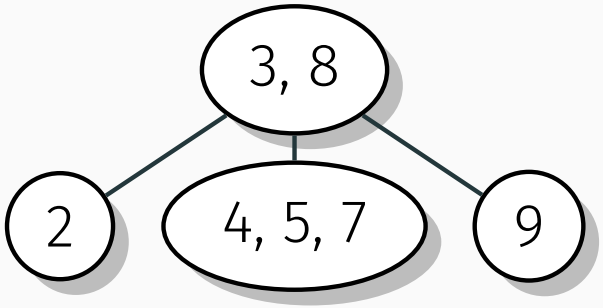
Operation

Result

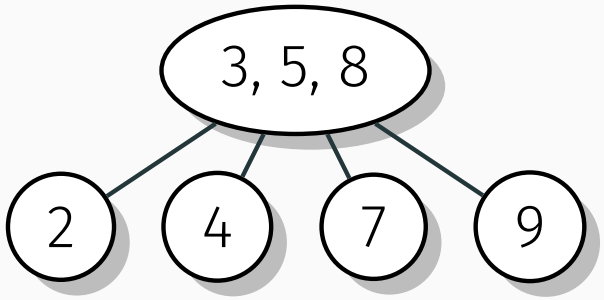




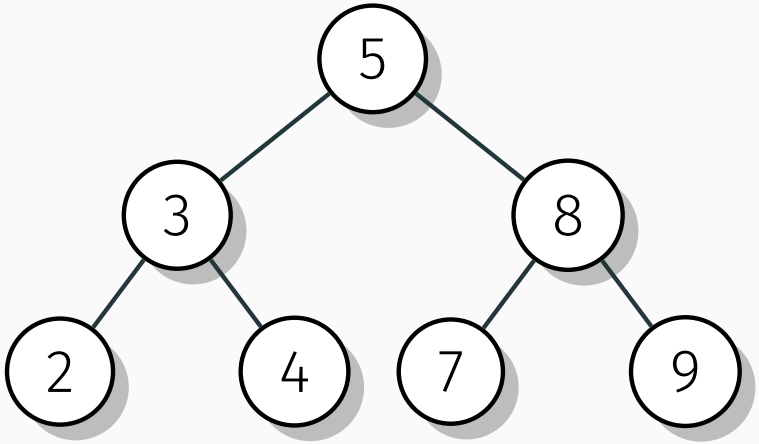
Construction of a 2-3 tree from the sequence 9, 5, 8, 3, 2, 4, 7 (cont.)



Operation
Result



Operation
Result



Sources for independent study

- Book [2], chapter 6.3, pages 218 – 225
- Book [5], chapters 4.4.6, 4.4.7 and 4.4.8, pages 296 – 310

Transform and Conquer

Heap and Heap Sorting

Heap

Heap – a partially sorted data structure, especially suitable for implementing a priority queue.

Priority Queue – a data structure understood as a multiset, where elements are ordered according to **priority** and supporting operations:

- finding the element with the highest priority,
- removing the element with the highest priority and
- inserting a new element into the queue.

Usage of Priority Queue :

- task scheduling in OS
- graph algorithms such as Prim's, Dijkstra's etc.
- heap sorting – **HeapSort**
- and others...

Heap – distinction of terminology

The term heap in computer science is used to denote:

- a data structure and
- a part of the operating memory during program execution.

In further explanation, we will deal with the heap exclusively as a **data structure**.

Heap

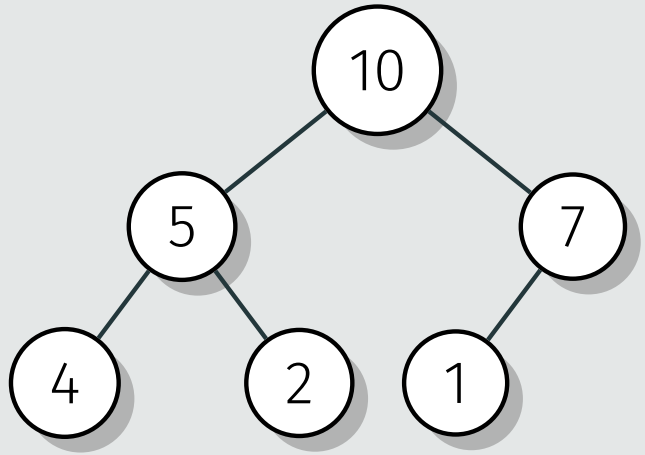
Definition

A **heap** is defined as a binary tree with one key in each node, which satisfies the following two properties:

1. **completeness**, i.e., all levels of the tree are filled, except for the last. In the last level, several leaves may be missing from the right and
2. **parent dominance**, i.e., the key in each node is always greater than or equal to the keys in all its children. In leaves, any key is always considered greater than the keys in non-existent children.

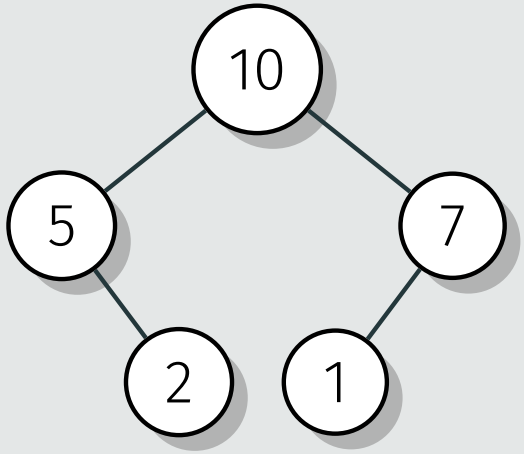
Heap – example

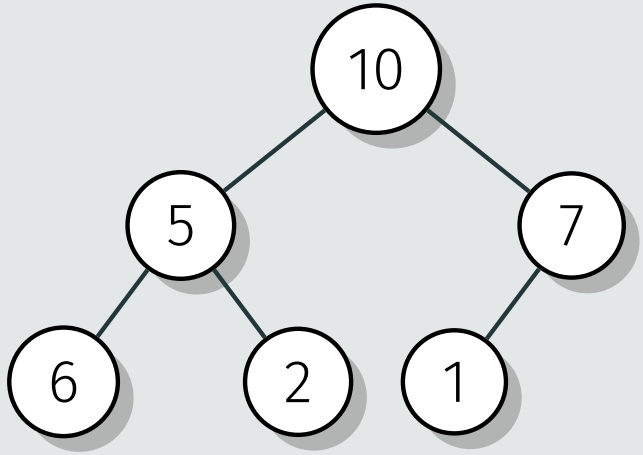
Heap



Not every binary tree is a heap!

These are not heaps – why?





Heap – additional properties

For all heaps, it can be proven that:

1. The keys on each path from the root to a leaf form a **non-increasing** sequence. Otherwise, there are no relationships between the keys, e.g., smaller keys in the left subtree than in the right etc.
2. For n keys, there exists only one complete binary tree. Its height is $\lfloor \log_2 n \rfloor$.
3. The largest key is always at the root of the heap.
4. Each node in the heap is always the root of a heap formed by this node and its descendants.

Heap – array representation

In an array, we store the heap from the root to the leaves and from left to right: Then:

1. internal nodes – the first $\left\lfloor \frac{n}{2} \right\rfloor$, leaves are the remaining $\left\lceil \frac{n}{2} \right\rceil$,
2. the children of a node at position i, where $1 \leq i \leq \left\lfloor \frac{n}{2} \right\rfloor$, are located at positions $2i$ and $2i + 1$. And conversely, the parent of a node at position j, for $2 \leq j \leq n$, is located at position $\left\lfloor \frac{j}{2} \right\rfloor$.

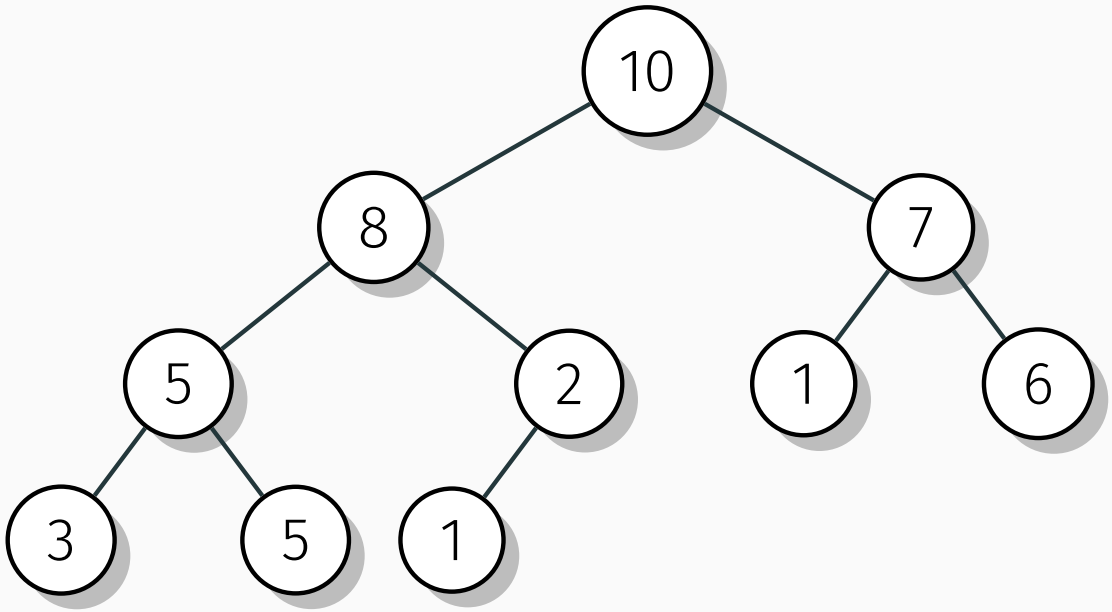
Remark

A heap can be defined as an array $H[1 \ldots n]$ in which for each element at index i holds

$$H[i] \geq max\{H[2i], H[2i + 1]\}$$

for all $i = 1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor$.

Heap – representation in an array, example



index	1	2	3	4	5	6	7	8	9	10
key	10	8	7	5	2	1	6	3	5	1
	internal nodes					leaves				

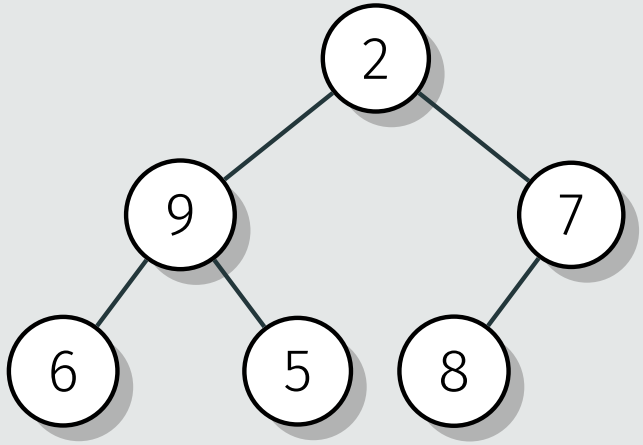
Construction of a heap

A heap can be constructed in two ways:

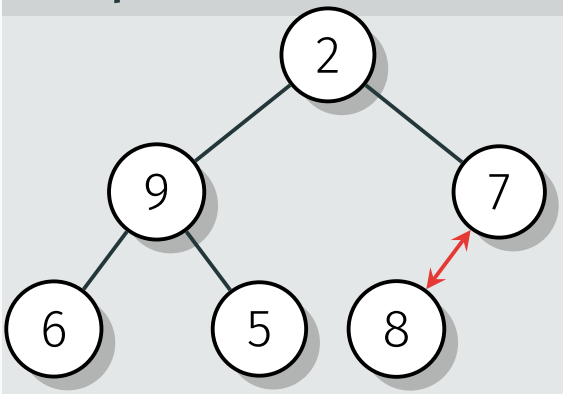
1. **bottom-up** and
2. top-down.

Construction of a heap from the bottom up – example

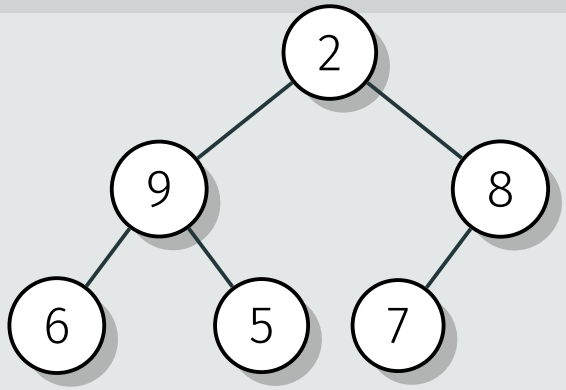
Initial state of the heap



Step 1

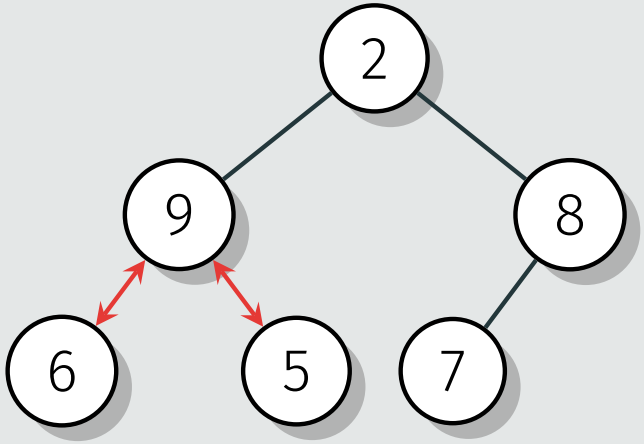


Operation result

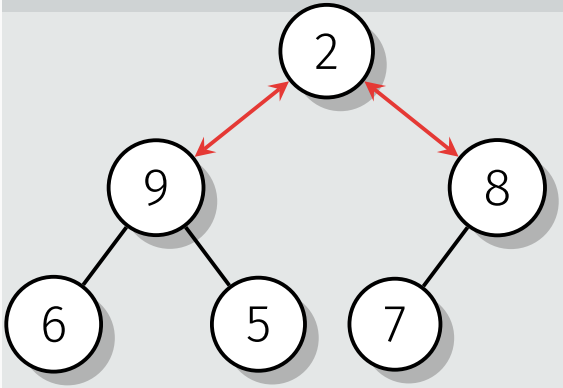


Construction of a heap from the bottom up – example (cont.)

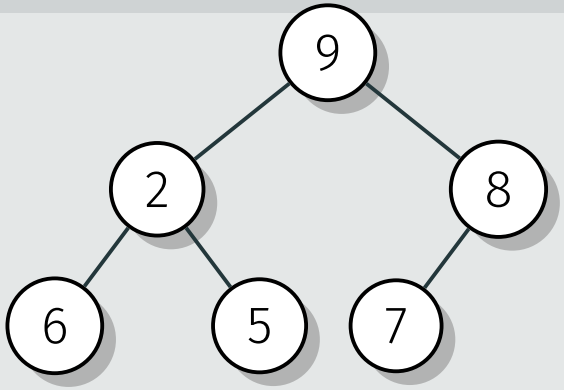
Step 2



Step 3a

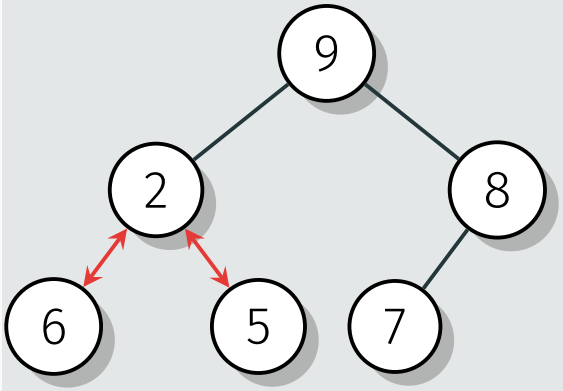


Operation result

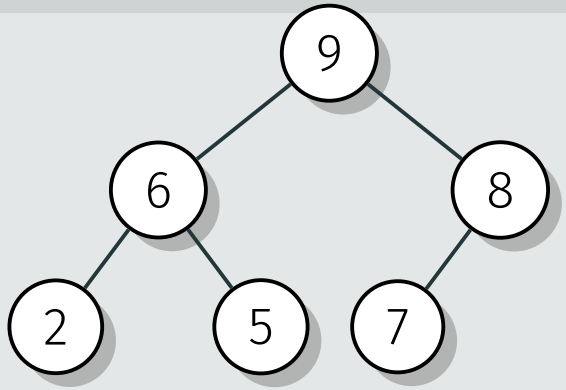


Construction of a heap from the bottom up – example (cont.)

Step 3b



Operation result



Finished heap

Construction of a heap from the bottom up

Input : Array $A[0 \ldots n-1]$ with a defined ordering on the array elements, i root of the heap being constructed

Output: Heap with the root at index i

```
1  procedure Heapify(A, n, i)
2      largest ← i;
3      l ← 2 * i + 1;
4      r ← 2 * i + 2;
5      if l < n ∧ A[l] > A[largest] then largest ← l ;
6      if r < n ∧ A[r] > A[largest] then largest ← r ;
7      if largest ≠ i then
8          Swap (A[i], A[largest]);
9          Heapify (A, n, largest);
10     end
11 end
```

Construction of a heap from the bottom up

Input : Array $A[0 \ldots n-1]$ with a defined ordering on the array elements

Output: Heap in the array A

```
1 procedure MakeHeap(A, n)
2 |   for i ← ⌊n/2⌋ - 1 down to 0 do
3 |   |   Heapify (A, n, i);
4 |   end
5 end
```

Heap Construction from Bottom to Top – Time Complexity

For simplicity, let us assume that $n = 2^k - 1$, i.e., the heap forms a complete binary tree.

The height of the heap is then $h = \lfloor \log_2 n \rfloor$, which can be written as

$$\begin{aligned} \lceil \log_2(n+1) \rceil - 1 &= \lceil \log_2(2^k - 1 + 1) \rceil - 1 \\ &= \lceil \log_2(2^k) \rceil - 1 \\ &= k - 1 \end{aligned}$$

Heap Construction from Bottom to Top – Time Complexity (cont.)

Remark

The expression $\lceil \log_2(n+1) \rceil$ can be interpreted as the "height of the heap with $n+1$ elements". We assumed a complete binary tree $\Rightarrow$ the tree with $n+1$ elements definitely has one more level than the tree with n elements.

Each key from level i will be shifted, in the worst case, to the leaf, i.e., to level h.

Shifting by one level requires two comparisons:

1. finding the larger of both children and
2. testing whether an exchange with the parent is necessary.

Heap Construction from Bottom to Top – Time Complexity (cont.)

The number of comparisons is therefore $2(h - i)$.

The total number of comparisons will be, in the worst case, equal to

$$\begin{aligned} C(n) &= \sum_{i=0}^{h-1} \sum_{\text{keys of level } i} 2(h - i) \\ &= \sum_{i=0}^{h-1} 2(h - i)2^i = 2h \sum_{i=0}^{h-1} 2^i - 2 \sum_{i=0}^{h-1} i2^i \\ &= 2n - 2 \log_2(n + 1) \end{aligned}$$

Heap Construction from Bottom to Top – Time Complexity (cont.)

Constructing a heap with n elements requires, in the worst case, less than $2n$ comparisons.

Remark

In the derivation, we used the formulas:

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

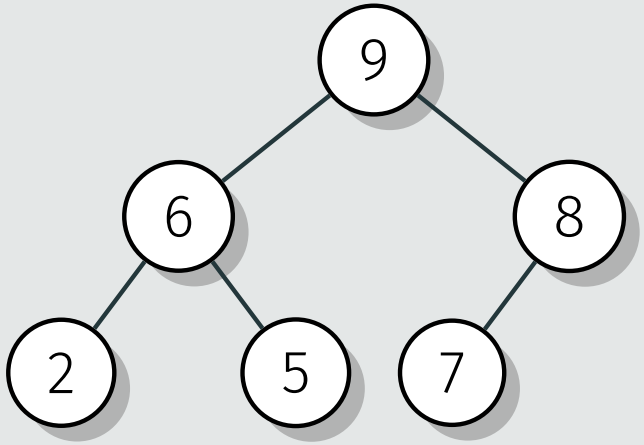
$$\sum_{i=1}^{n} i2^i = 1 \cdot 2 + 2 \cdot 2^2 + \cdots + n2^n = (n-1)2^{n+1} + 2$$

Construction of a heap from top to bottom

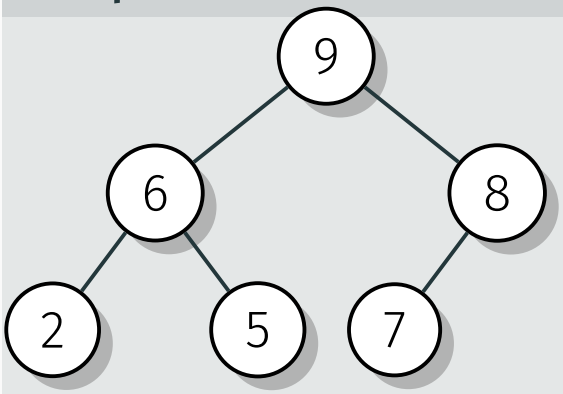
- Repeated insertion of a new key into an existing heap.
  1. We insert the new key at the end of the heap.
  2. We compare the new key with its parent and potentially move the new key up one level.
  3. We continue this process until we encounter a larger parent or reach the root of the heap.
- The height of a heap with n elements is $\approx \log_2 n$, thus the complexity of inserting a key into the heap is $O(\log n)$.
- Construction from top to bottom is therefore more complex than construction from bottom to top.

Construction of a heap from top to bottom – example

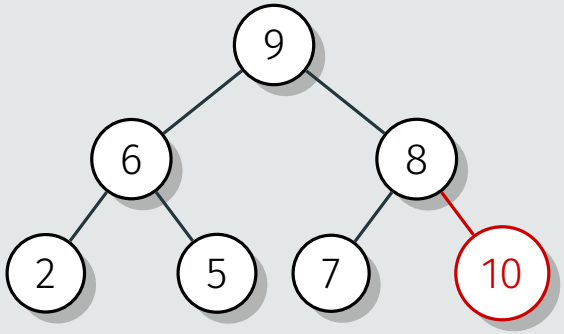
Initial state of the heap



Step 1 – insertion of key 10 at the end of the heap

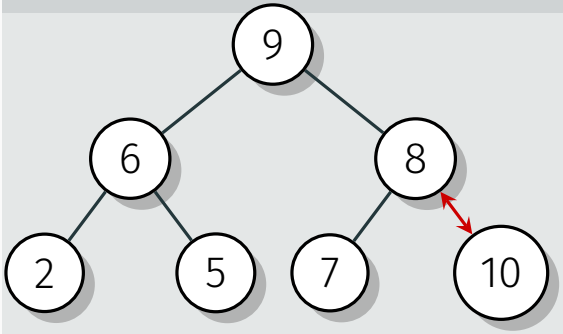


Operation result

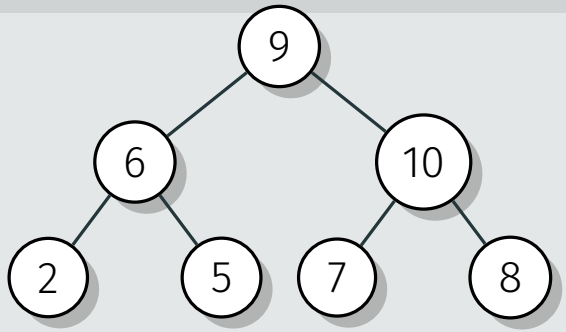


Construction of a heap from top to bottom – example (cont.)

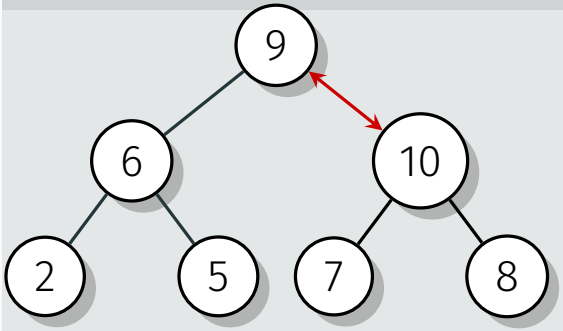
Step 2a – comparison of key 10 with parent



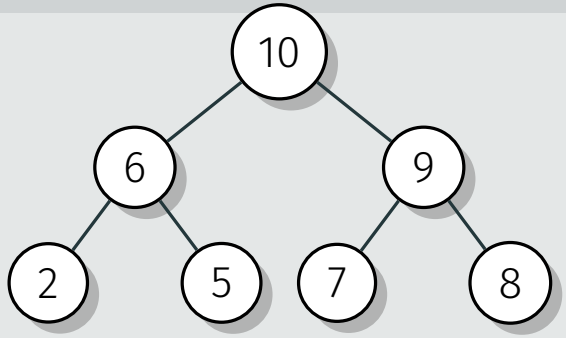
Operation result



Step 2b – comparison of key 10 with parent



Operation result



Removal of the largest key from the heap

Algorithm principle:

1. Swapping the key in the root with the key at the end of the heap.
2. Reducing the heap by one.
3. Heap restoration – testing whether the parent key is greater than the keys in both children and, if necessary, performing a swap. This process is repeated until the parent key is greater than the keys in the children.

Remark

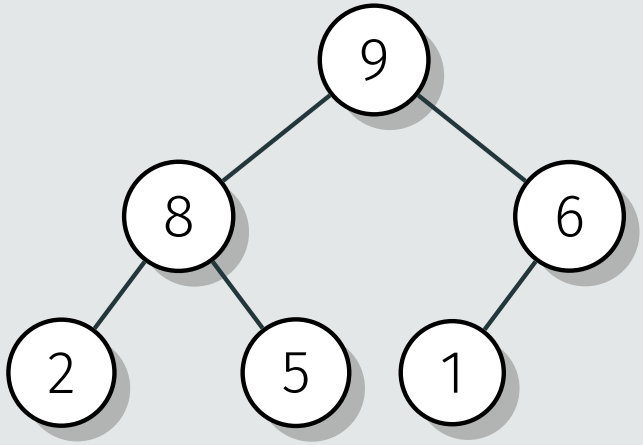
In principle, any key can be removed from the heap. But this operation has no practical significance.

Removal of the largest key from the heap – algorithm complexity

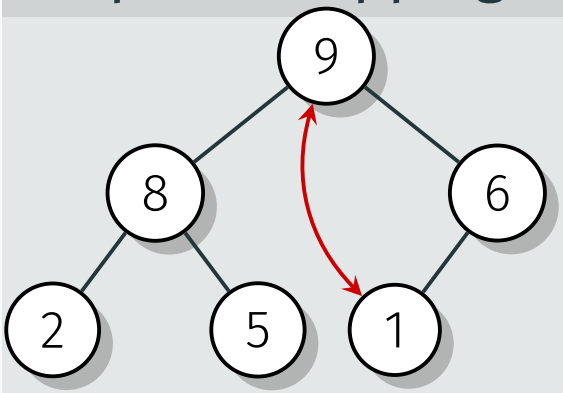
- The number of comparisons necessary to restore the heap is proportional to the height of the heap – we “move” the key from the root down through the levels.
- We always compare the parent with both children – we must find the largest of the given trio.
- The height of the heap is $h \approx \log_2 n$, so the number of comparisons will not be greater than $2h$.
- The complexity of the algorithm is therefore $O(\log n)$.

Removal of the largest key from the heap – example

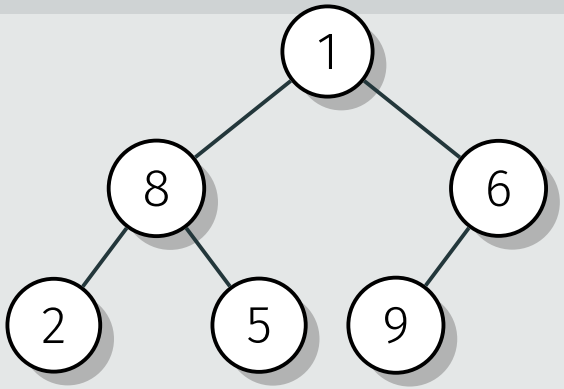
Initial state of the heap



Step 1 – swapping the root with the last element

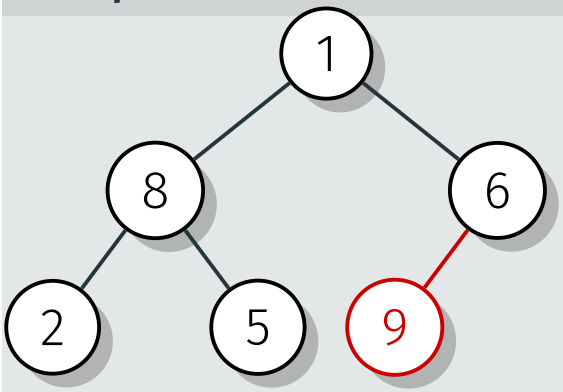


Operation result

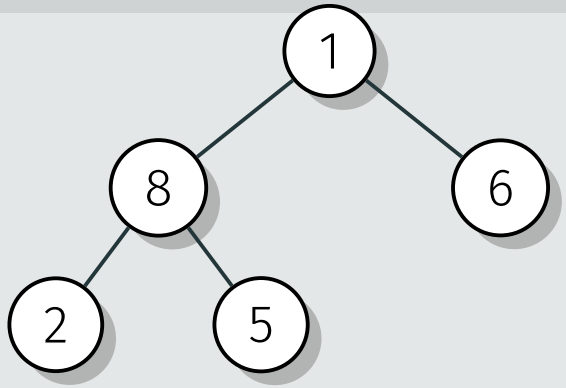


Removal of the largest key from the heap – example (cont.)

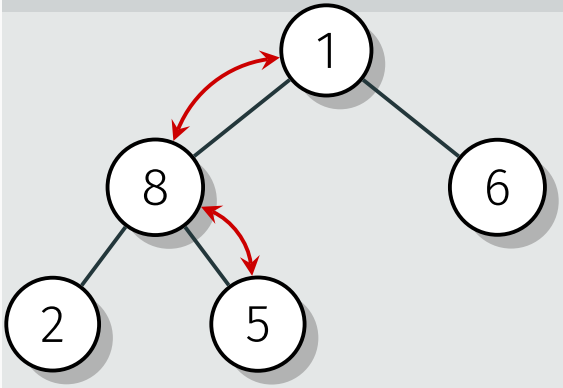
Step 2 – removal of the last node



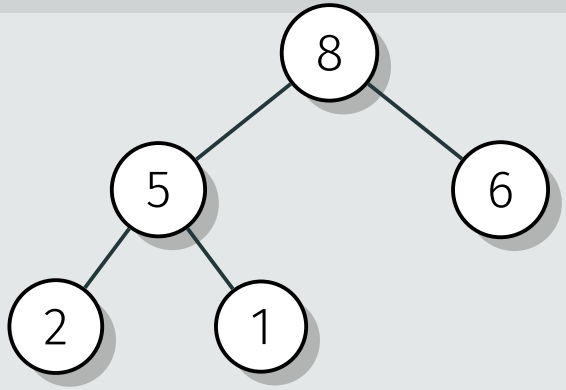
Operation result



Step 3 – heap restoration



Operation result



Heap Sorting – HeapSort

The algorithm works in two phases:

Heap Construction : for a given array, a heap is constructed.

Removal of Maximum : the algorithm for removing the largest key from the progressively decreasing heap is applied $(n - 1)$ times.

Heap Sorting – HeapSort

Input : Array $A[0 \ldots n-1]$ with a defined ordering on the array elements

Output: Sorted array A

```
1 procedure HeapSort(A, n)
2 |   BuildHeap (A, n);
3 |   for i ← n - 1 downto 0 do
4 |   |   Swap (A[0], A[i]);
5 |   |   Heapify (A, i, 0);
6 |   end
7 end
```

Heap sorting – algorithm complexity

- The complexity of the first phase is $O(n)$.
- In the second phase, we progressively remove the largest key from the heap of decreasing size $n, n-1, \ldots, 2$. The number of comparisons $C(n)$ is

$$\begin{aligned} C(n) &\leq 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \\ &\leq 2\sum_{i=1}^{n-1} \log_2 i \\ &\leq 2\sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1)\log_2(n-1) \leq 2n \log_2 n \end{aligned}$$

Thus, $C(n) \in O(n \log n)$.

Heap sorting – algorithm complexity (cont.)

- For both phases, we get $O(n) + O(n \log n) = O(n \log n)$.
- Further complexity analysis can prove that the same complexity applies to the average case as well. Therefore, $\Theta(n \log n)$.
- Heap sorting is comparable to merge sorting.
- However, in practice, it is slower than QuickSort.

Sources for Independent Study

- Book [2], chapter 6.4, pages 226 – 232
- Book [3], chapters 6.1 through 6.4, pages 161 – 172

Transform and Conquer

Horner's Scheme

Value of a Polynomial at a Point

Problem Statement

Given is a polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0.$$

Our task is to compute the value of the polynomial $p(x)$ at the point x_0.

Motivation

- Polynomials are used for function approximation, namely
  1. How does a processor calculate the value of the function $\sin(x)$?
  2. Where do the values of the function $\sin(x)$ in mathematical tables come from?

  Using the Taylor series expansion of a function, which is a polynomial!

Taylor expansion of the function $y = f(x)$

The function $f(x)$, which has finite derivatives up to order $n + 1$ at point a, can be expressed in the vicinity of point a as an expansion

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n + R_{n+1}^{f,a}(x)$$

For $a = 0$, the expansion is called Maclaurin

$$f(x) = f(0) + \frac{f'(0)}{1!}x + \frac{f''(0)}{2!}x^2 + \cdots + \frac{f^{(n)}(0)}{n!}x^n + R_{n+1}^{f,0}(x)$$

Taylor expansion of the function $y = \sin(x)$ at point 0

$$\sin(x) = \sin(0) + \frac{\sin'(0)}{1!}x + \frac{\sin''(0)}{2!}x^2 + \cdots + \frac{\sin^{(n)}(0)}{n!}x^n + R_{n+1}^{\sin,0}(x)$$

Derivatives

$$\sin^{(1)} 0 = \cos 0 = 1 \qquad \sin^{(2)} 0 = -\sin 0 = 0$$

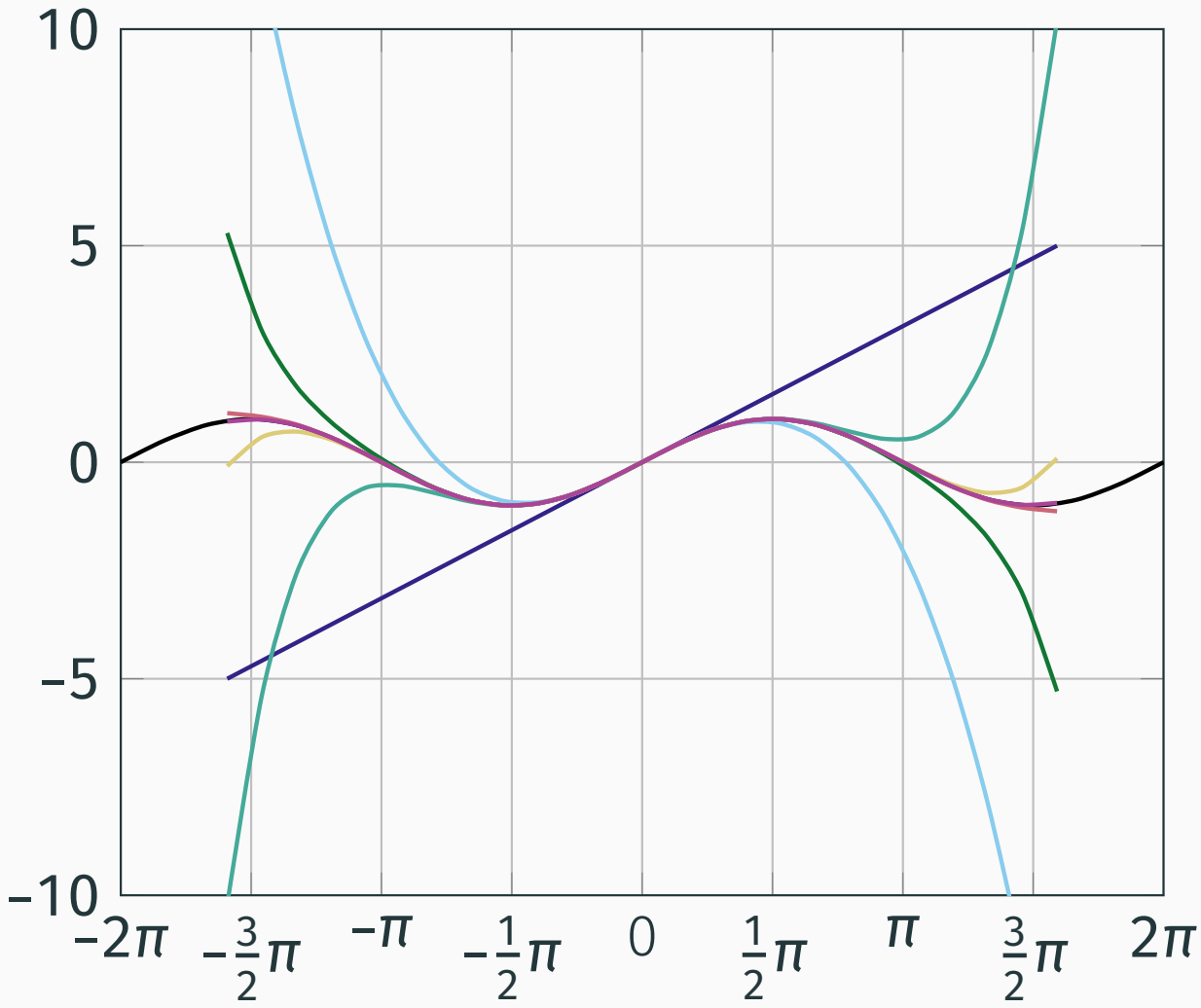
$$\sin^{(3)} 0 = -\cos 0 = -1 \qquad \sin^{(4)} 0 = \sin 0 = 0$$

$$\sin(x) = 0 + \frac{1}{1!}x + \frac{0}{2!}x^2 + \frac{-1}{3!}x^3 + \frac{0}{4!}x^4 + \cdots + R_{n+1}^{\sin,0}(x)$$

Approximation by a 13th-degree polynomial

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \frac{x^{13}}{13!}$$

Taylor series expansion of the function $y = \sin(x)$ at point 0

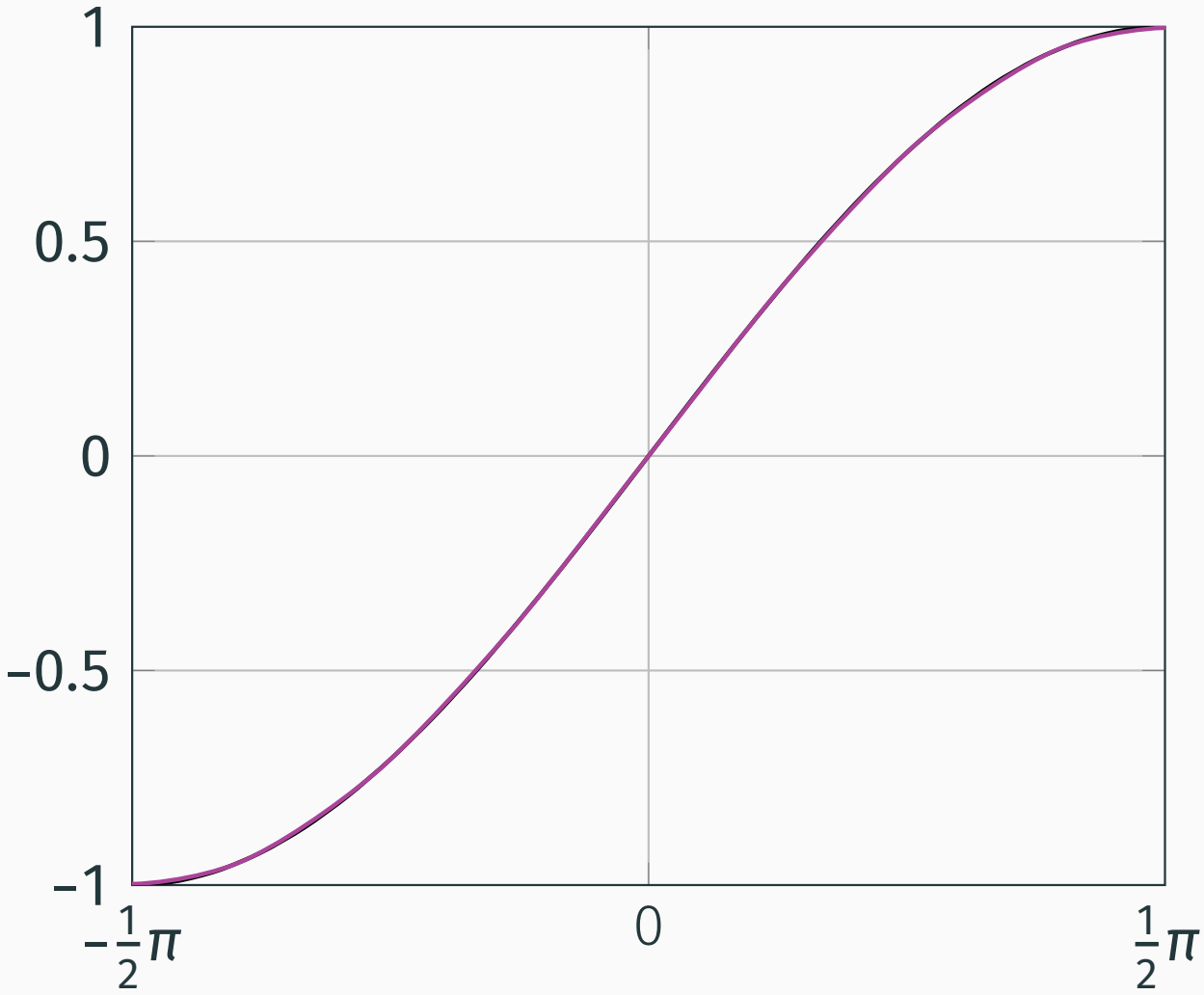


Taylor series expansion:

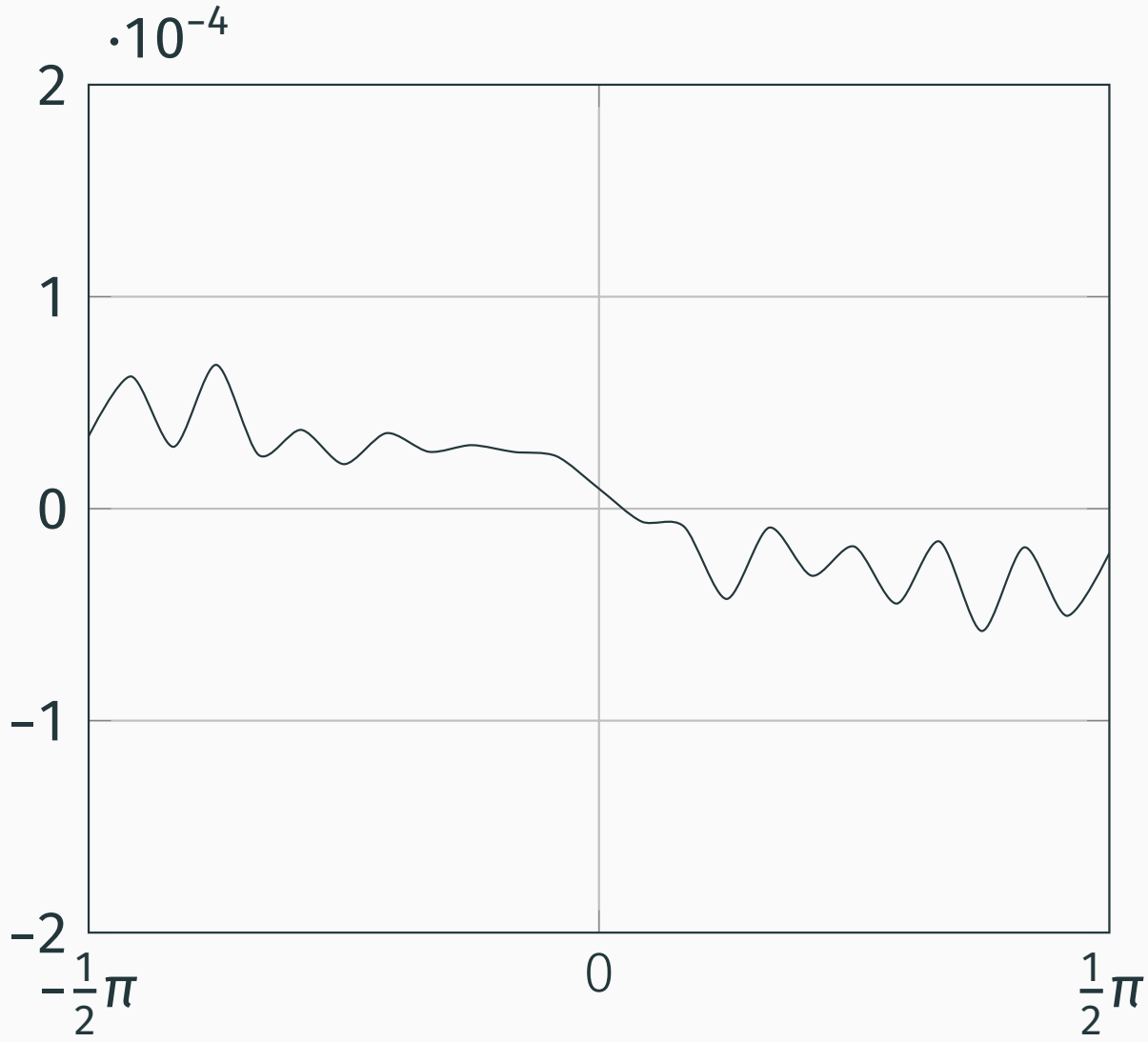
- degree 1
- degree 3
- degree 5
- degree 7
- degree 9
- degree 11
- degree 13

The function $y = \sin(x)$ is displayed in black.

Taylor series expansion of the function $y = \sin(x)$ of degree 13 at point 0

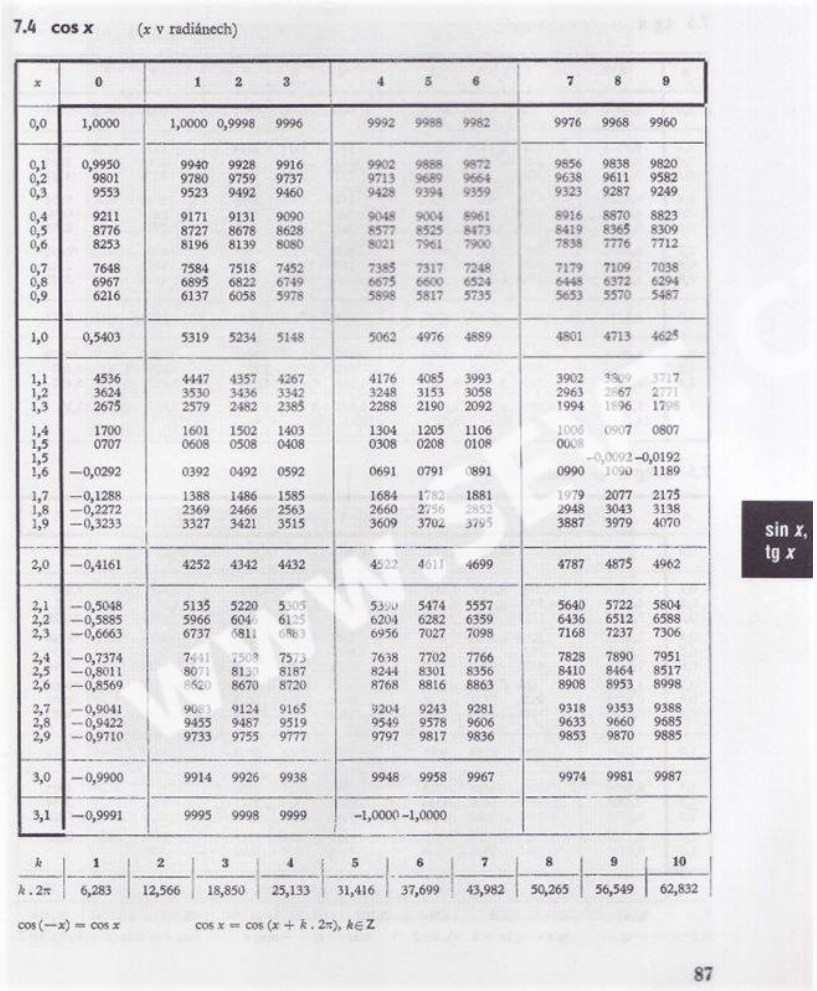


Taylor series expansion of the function $y = \sin(x)$ at point 0, approximation error



Tables of function values

- Using Taylor series expansion, we can approximate the value of the desired function and construct tables.
- Manual calculation – laborious and prone to a vast number of errors.
- Breakthrough idea – numerical computations do not require intelligence! They can be performed **mechanically**!



7.4 cos x (x v radiánech)

x	0	1	2	3	4	5	6	7	8	9
0,0	1,0000	1,0000	0,9998	9996	9992	9988	9982	9976	9968	9960
0,1	0,9950	9940	9928	9916	9902	9888	9872	9856	9838	9820
0,2	9801	9780	9759	9737	9713	9689	9664	9638	9611	9582
0,3	9553	9523	9492	9460	9428	9394	9359	9323	9287	9249
0,4	9211	9171	9131	9090	9048	9004	8961	8916	8870	8823
0,5	8776	8727	8678	8628	8577	8525	8473	8419	8365	8309
0,6	8253	8196	8139	8080	8021	7961	7900	7838	7776	7712
0,7	7648	7584	7518	7452	7385	7317	7248	7179	7109	7038
0,8	6967	6895	6822	6749	6675	6600	6524	6448	6372	6294
0,9	6216	6137	6058	5978	5898	5817	5735	5653	5570	5487
1,0	0,5403	5319	5234	5148	5062	4976	4889	4801	4713	4625
1,1	4536	4447	4357	4267	4176	4085	3993	3902	3809	3717
1,2	3624	3530	3436	3342	3248	3153	3058	2963	2867	2771
1,3	2675	2579	2482	2385	2288	2190	2092	1994	1896	1798
1,4	1700	1601	1502	1403	1304	1205	1106	1006	0907	0807
1,5	0707	0608	0508	0408	0308	0208	0108	0008	−0,0092	−0,0192
1,6	−0,0292	0392	0492	0592	0691	0791	0891	0990	1090	1189
1,7	−0,1288	1388	1486	1585	1684	1782	1881	1979	2077	2175
1,8	−0,2272	2369	2466	2563	2660	2756	2852	2948	3043	3138
1,9	−0,3233	3327	3421	3515	3609	3702	3795	3887	3979	4070
2,0	−0,4161	4252	4342	4432	4522	4611	4699	4787	4875	4962
2,1	−0,5048	5135	5220	5305	5390	5474	5557	5640	5722	5804
2,2	−0,5885	5966	6046	6125	6204	6282	6359	6436	6512	6588
2,3	−0,6663	6737	6811	6883	6956	7027	7098	7168	7237	7306
2,4	−0,7374	7441	7508	7573	7638	7702	7766	7828	7890	7951
2,5	−0,8011	8071	8130	8187	8244	8301	8356	8410	8464	8517
2,6	−0,8569	8620	8670	8720	8768	8816	8863	8908	8953	8998
2,7	−0,9041	9083	9124	9165	9204	9243	9281	9318	9353	9388
2,8	−0,9422	9455	9487	9519	9549	9578	9606	9633	9660	9685
2,9	−0,9710	9733	9755	9777	9797	9817	9836	9853	9870	9885
3,0	−0,9900	9914	9926	9938	9948	9958	9967	9974	9981	9987
3,1	−0,9991	9995	9998	9999	−1,0000	−1,0000				

k	1	2	3	4	5	6	7	8	9	10
k . 2π	6,283	12,566	18,850	25,133	31,416	37,699	43,982	50,265	56,549	62,832

$\cos(-x) = \cos x$ $\cos x = \cos(x + k \cdot 2\pi),\ k \in Z$

87

sin x, tg x

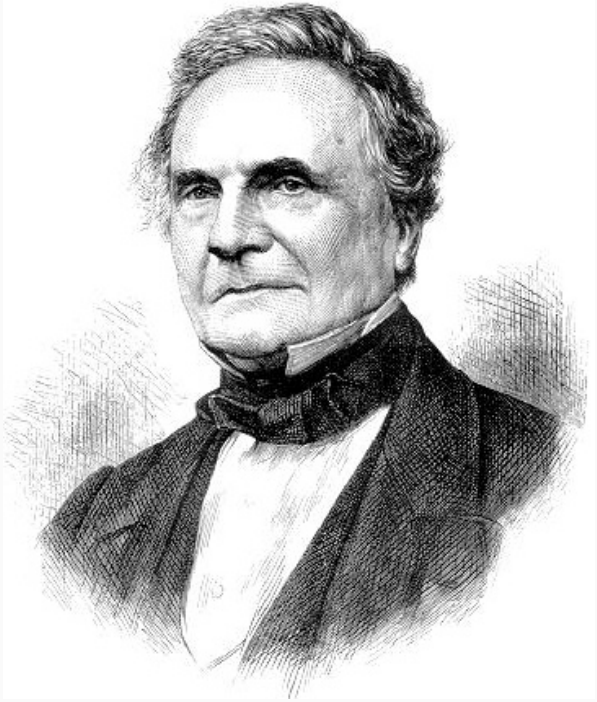
Charles Babbage – Difference Engine

Difference Engine

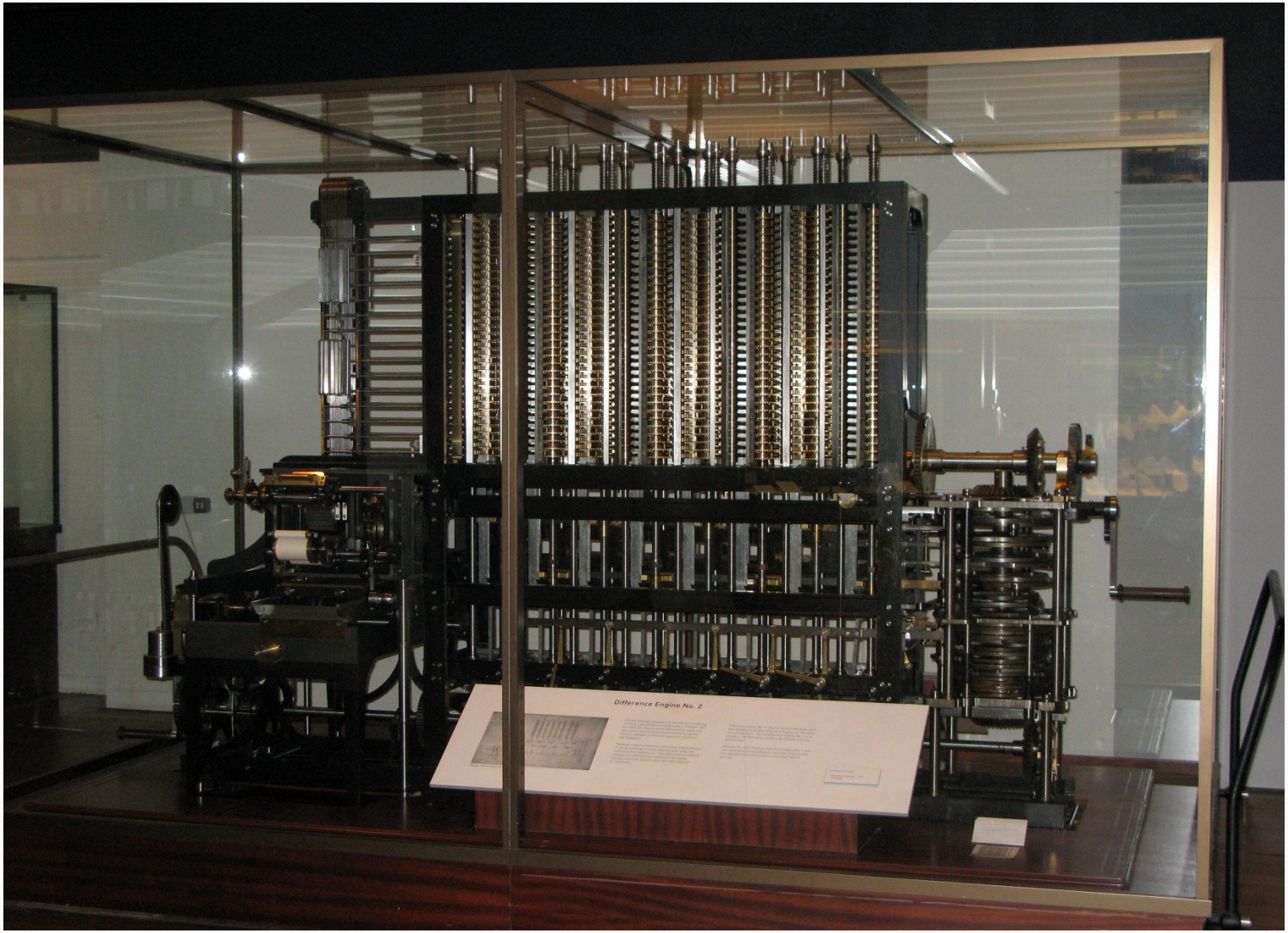
- first programmable computer in the world
- 1819 – commencement of work
- 1822 – prototype completed
- 1823 – work begun on large machine
- 1833 – work halted
- 1842 – government support terminated, 17 thousand pounds spent on project, machine never completed
- 1991 – functional replica!

Charles Babbage

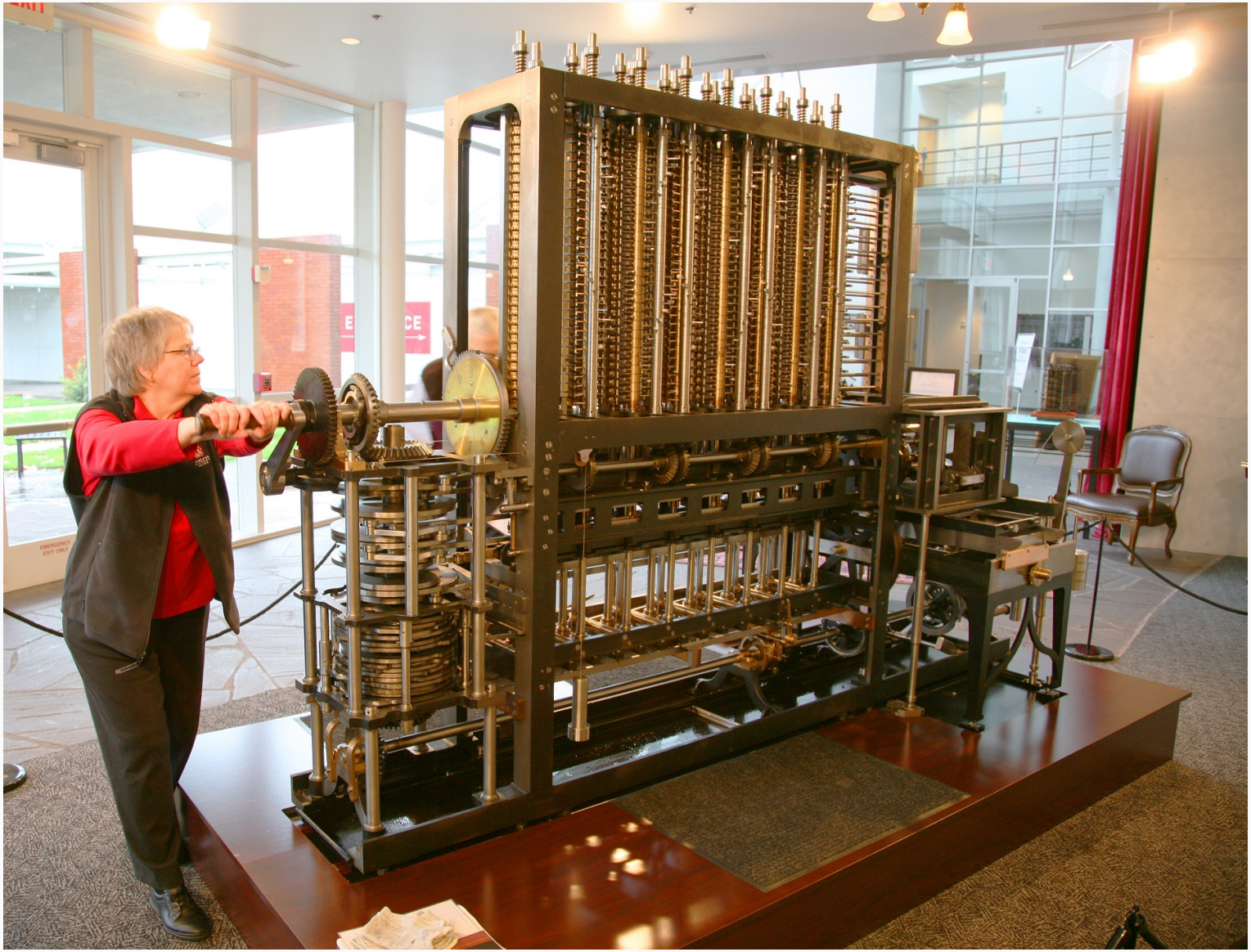
1791 – 1871



Difference Engine



Difference Engine



First programmer in the world?!

Augusta Ada King, Countess of Lovelace (1815 – 1852)

Programmer of the **Analytical Engine**, (Babbage 1837), which was the first general-purpose Turing-complete computer.

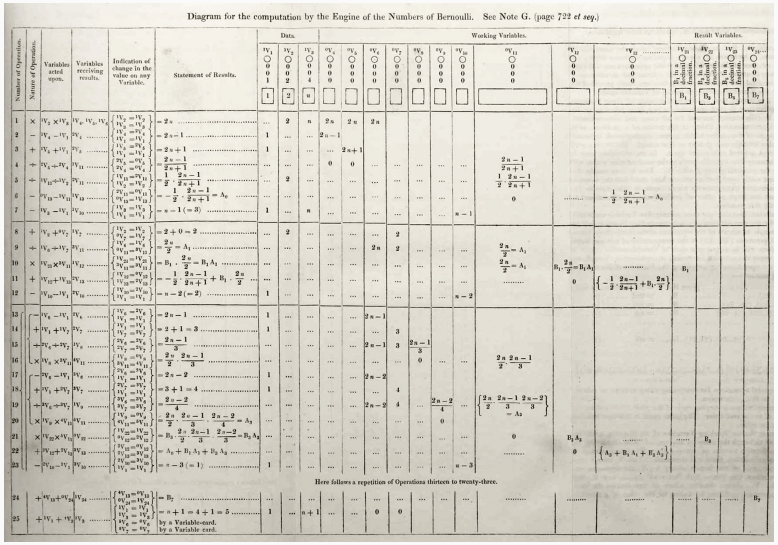


Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 *et seq.*)

Horner's scheme – transformation

Basic idea:

- transformation of a polynomial into another form,
- we gradually extract the variable x from parts of the polynomial.

$$
\begin{aligned}
p(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n \\
&= a_0 + x\left(a_1 + a_2 x + \cdots + a_{n-1} x^{n-2} + a_n x^{n-1}\right) \\
&= a_0 + x\left(a_1 + x\left(a_2 + \cdots + a_{n-1} x^{n-3} + a_n x^{n-2}\right)\right) \\
&\vdots \\
&= a_0 + x\Big(a_1 + x\Big(a_2 + \cdots + x\big(a_{n-1} + a_n x\big)\ldots\Big)\Big)
\end{aligned}
$$

It is easy to see that this equality holds by successive multiplication of all parentheses.

Horner's scheme – computation

The value of $p(x_0)$ is computed "from the inside" of the parentheses, progressively calculating the values of b_i

$$\begin{aligned} b_n &= a_n \\ b_{n-1} &= a_{n-1} + b_n x_0 \\ b_{n-2} &= a_{n-2} + b_{n-1} x_0 \\ &\vdots \\ b_0 &= a_0 + b_1 x_0 \end{aligned}$$

The value of b_0 is then equal to $p(x_0)$, since

$$p(x_0) = a_0 + x_0\Big(a_1 + x_0\big(a_2 + \cdots + x_0\big(a_{n-1} + a_n x_0\big)\ldots\big)\Big)$$

Horner's scheme – computation (cont.)

and by progressively substituting b_i, we obtain

$$
\begin{aligned}
p(x_0) &= a_0 + x_0\Big(a_1 + x_0\Big(a_2 + \cdots + x_0\big(a_{n-1} + b_n x_0\big)\ldots\Big)\Big) \\
p(x_0) &= a_0 + x_0\Big(a_1 + x_0\Big(a_2 + \cdots + x_0\big(b_{n-1}\big)\ldots\Big)\Big) \\
p(x_0) &= a_0 + x_0(b_1) \\
p(x_0) &= b_0
\end{aligned}
$$

Horner's scheme – manual calculation

Calculate the value of the polynomial $p(x) = 2x^3 - 6x^2 + 2x - 1$ at the point $x_0 = 3$.

x_0	x^3	x^2	x^1	x^0
3	2	-6	2	-1
		6	0	6
	2	0	2	5

Standard calculation

$$\begin{aligned} p(3) &= 2 \times 3^3 - 6 \times 3^2 + 2 \times 3 - 1 \\ &= 2 \times 27 - 6 \times 9 + 2 \times 3 - 1 \\ &= 54 - 54 + 6 - 1 = 5 \end{aligned}$$

Horner's scheme

ALGORITHM $Horner(P[0..n], x)$

//Evaluates a polynomial at a given point by Horner's rule
//Input: An array $P[0..n]$ of coefficients of a polynomial of degree n,
// stored from the lowest to the highest and a number x
//Output: The value of the polynomial at x
$p \leftarrow P[n]$
for $i \leftarrow n-1$ **downto** 0 **do**
    $p \leftarrow x * p + P[i]$
return p

Horner's scheme – time complexity of the algorithm

It is clear that the number of multiplications $M(n)$ and the number of additions $A(n)$ equals

$$M(n) = A(n) = \sum_{i=0}^{n-1} 1 = n \in \Theta(n)$$

Computation by brute force

Just for computing $a_n x^n$, the following is needed:

- $n - 1$ multiplications to compute the power
- 1 multiplication to multiply by a_n.

For the same number of multiplications, Horner's algorithm can also compute the remaining $n - 1$ terms of the polynomial!!!

Sources for independent study

- Book [2], chapter 6.5, pages 234 – 239
- Book [3], chapter 30.1, pages 879 – 880

Transform and Conquer

Problem Reduction

Problem Reduction

The purpose of reduction is to transform the problem being solved into another problem that we know how to solve.

Reduction Procedure

1. **Problem 1** – what we want to solve
2. Reduction of **Problem 1** to **Problem 2**
3. **Problem 2** – solvable by algorithm *A*
4. Execution of algorithm *A*
5. **Solution to Problem 2**

Least Common Multiple

The least common multiple $lcm(m, n)$ of two natural numbers m and n is defined as the smallest natural number that is divisible by both m and n.

Solution using Prime Factorization

$$\begin{aligned} 24 &= 2^3 \cdot 3^1 \\ 60 &= 2^2 \cdot 3^1 \cdot 5^1 \\ \text{lcm}(24, 60) &= 2^3 \cdot 3^1 \cdot 5^1 = 120 \end{aligned}$$

Solution using Greatest Common Divisor

It can be proven that

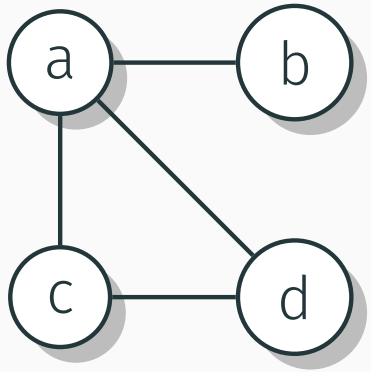
$$\text{lcm}(m, n) = \frac{mn}{\gcd(m, n)}$$

$\gcd(m, n)$ can be computed efficiently using the Euclidean algorithm

Number of walks in a graph

Problem statement: Calculate the number of walks between pairs of vertices in a given graph G.

Solution: It can be proven that the number of different walks of length k between vertices i and j is equal to the element a_{ij} of the matrix $\mathbf{A}^k$, where A is the adjacency matrix of graph G.



$$\mathbf{A} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \end{array} \qquad \mathbf{A}^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{pmatrix} \end{array}$$

From a to a, there are three walks of length 2: $a - b - a$, $a - c - a$, $a - d - a$

From a to c, there is one walk of length 2: $a - d - c$

Reduction of Optimization Problems

Maximization Problem – finding the maximum of function $f(x)$

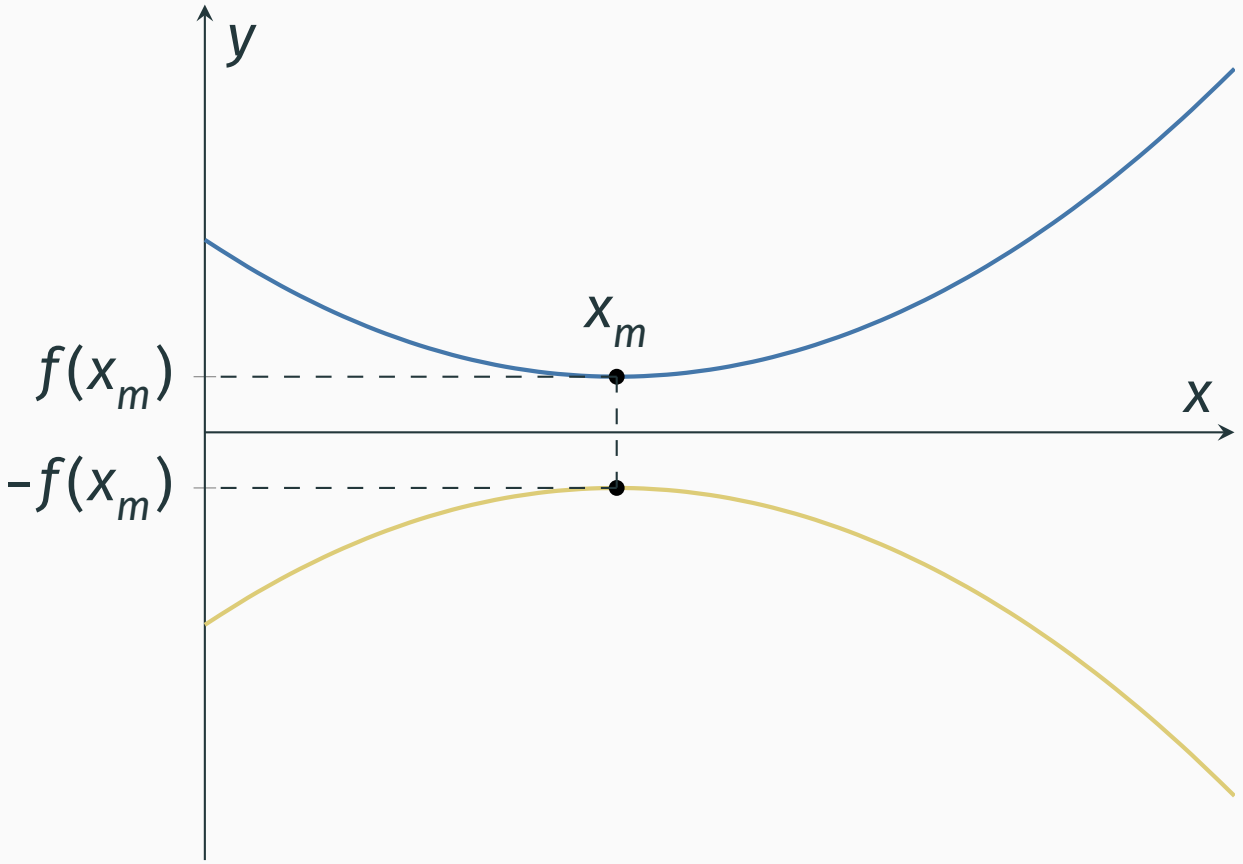
Minimization Problem – finding the minimum of function $f(x)$

How to Solve the Situation?

- We need to minimize function $f(x)$, but
- we only have a maximization algorithm available.

Can we use a maximization algorithm for a minimization problem? Or vice versa?

Reduction of Optimization Problems



$$\min f(x) = -\max\left[-f(x)\right]$$
$$\max f(x) = -\min\left[-f(x)\right]$$

Goat, wolf and cabbage

- On the riverbank, there is a ferryman, a goat, a wolf, and cabbage.
- The ferryman must transport the goat, the wolf, and the cabbage to the other bank using a boat.
- The boat can hold at most one of the entities being transported, in addition to the ferryman.
- On the same bank, the pairs goat and cabbage and wolf and goat cannot be left together without the ferryman's supervision.
- The task is to devise a transportation plan or prove that no solution exists.

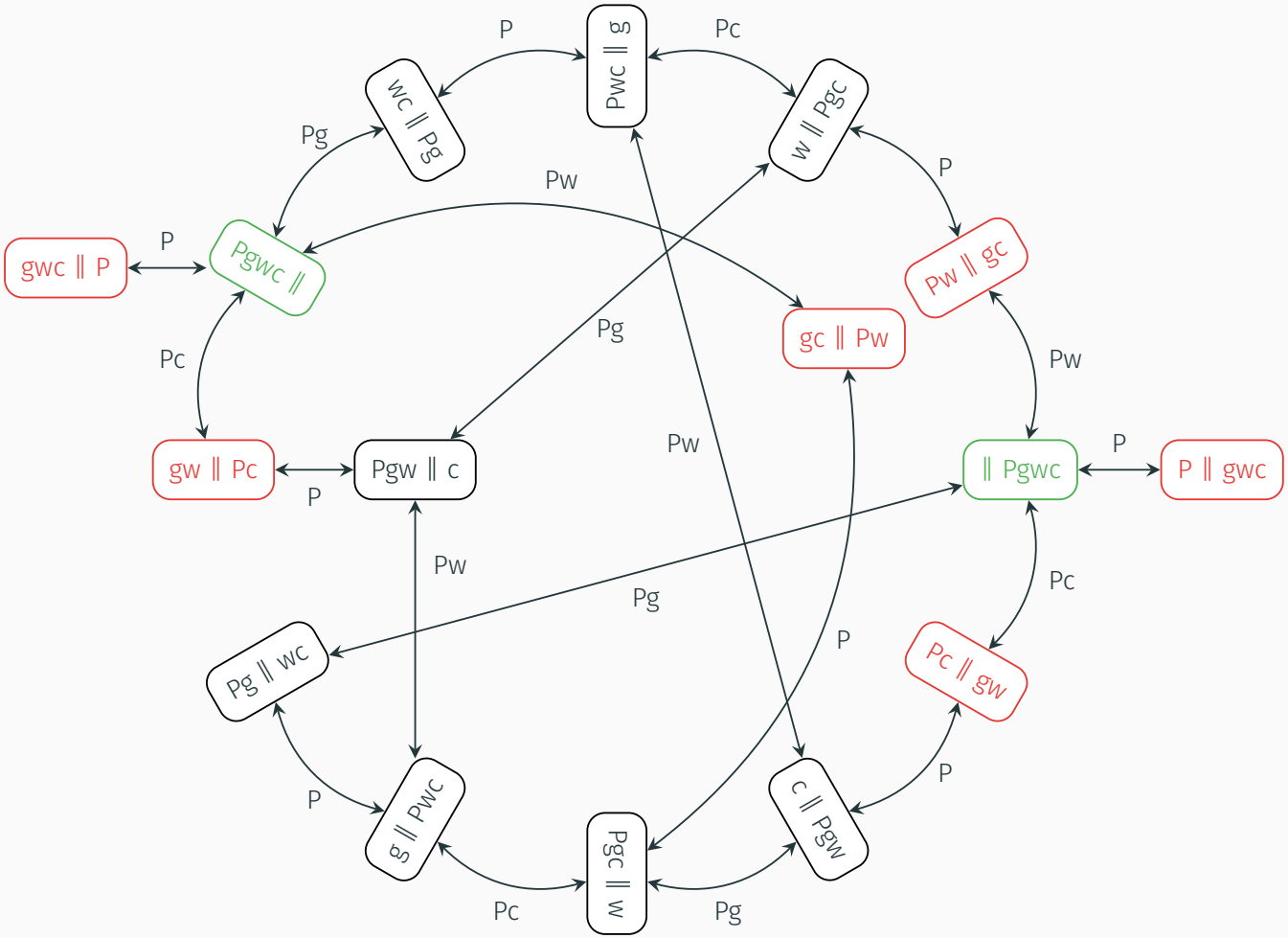
The oldest written form of the problem dates back to the 9th century...

Goat, wolf and cabbage – state space

State – represents the occupancy of both riverbanks, e.g. Gw||c

Transition between states – path from one riverbank to the other, with possible transportation

Goat, wolf and cabbage – state space graph



Solution to the problem – finding a directed path from the initial state to the final state through breadth-first traversal.

Sources for Independent Study

- Book [2], chapter 6.6, pages 240 – 248

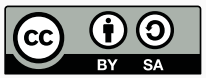
Thanks for your attention

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Space and Time Trade-Offs

Jiří Dvorský, Ph.D.

Department of Computer Science
VSB – Technical University of Ostrava



Space and Time Trade-Offs

B-trees

B-trees – motivation

- Processing a large amount of structured records (that can be identified by a unique key) that exceeds the available operating memory.
- Data must be stored in external memory, so-called ”on disk”.
- The disk offers only a sequential file.
- We are looking for a data structure that allows efficient searching, inserting, and deleting records in such a file.
- The answer is to trade off memory complexity for time complexity, in other words, we increase memory complexity (we sacrifice extra memory) to reduce the time complexity of operations.

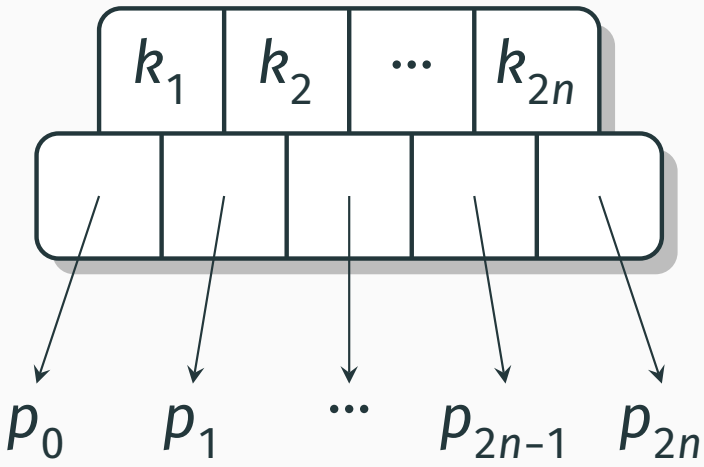
B-trees

B-tree of order n is a $(2n + 1)$-tree that satisfies the following criteria:

1. Each page contains at most $2n$ keys.
2. Each page, with the exception of the root, contains at least n keys.
3. Each page is either a leaf page, i.e. it has no children, or it has $m + 1$ children, where m is the current number of keys in the page.
4. All leaf pages are on the same level. In other words, the tree is perfectly balanced.

Published by Rudolf Bayer in 1972 [6].

B-trees – page schema

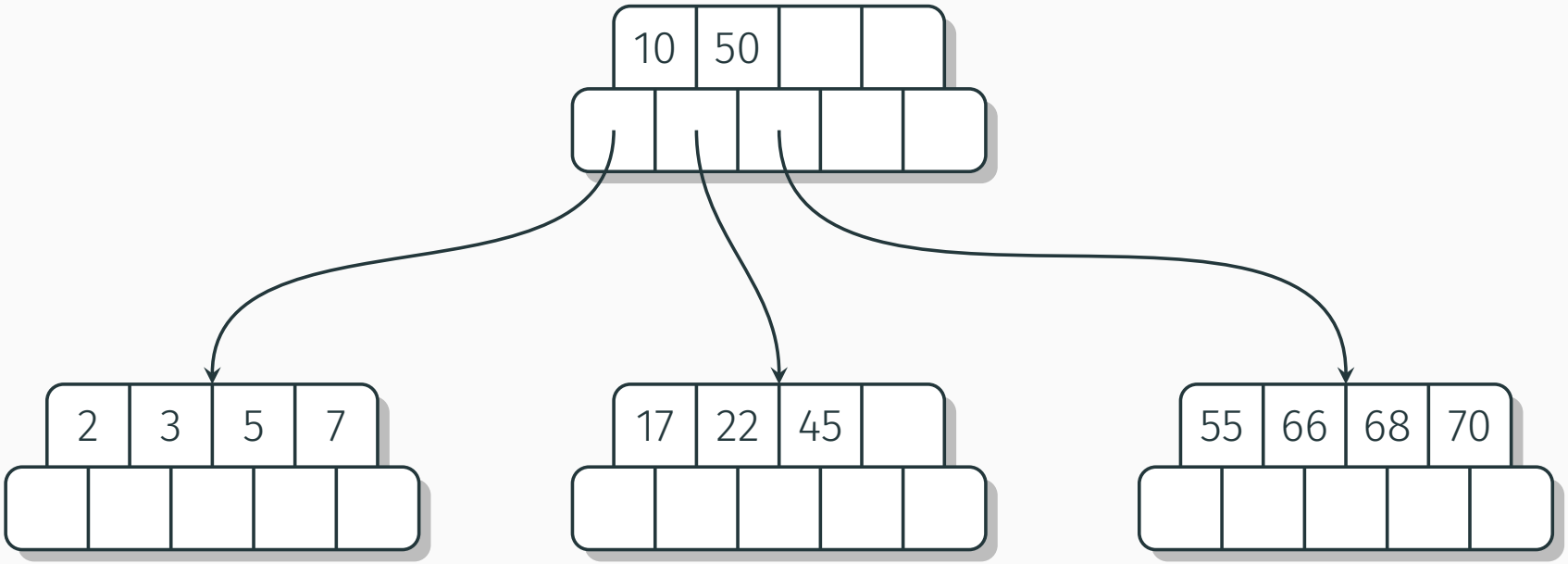


- Nodes in a B-tree are traditionally called **pages**.
- The number of keys in a page ranges from n to $2n$, with the exception of the root node.
- Keys in a page are sorted, i.e., $k_1 \leq k_2 \leq \cdots \leq k_{2n}$.
- For keys in the subtrees referenced by pointers $p_0, \ldots, p_{2n}$, the following holds

$$K_{p_0} \leq k_1 \leq K_{p_1} \leq k_1 \leq \cdots K_{p_{2n-1}} \leq k_{2n} \leq K_{p_{2n}},$$

where K_{p_i} is the set of all keys in the subtree rooted at the page referenced by p_i.

B-trees – example



B-trees – notes

- From the definition, it is clear that a B-tree does not have to be completely filled. The fill factor varies from 50 % to 100 %.
- Free space in the tree allows for easy insertion of additional keys.
- Thanks to the tree structure, search, insert, and delete key operations in a B-tree can be performed with logarithmic time complexity.
- B-tree algorithms are a generalization of binary search tree algorithms.

B-trees – alternative definition, variants

- The above definition only allows B-trees with a maximum capacity of $2n$ keys, i.e., an even number.
- However, the maximum capacity can be any number, even odd.
- Some definitions denote the maximum capacity using the number n.
- The number of keys in a page thus varies from $\left\lceil \frac{n}{2} \right\rceil$ to n.
- The goal of our definition is easy understandability of the B-tree operation principles and simple notation.
- Sometimes, one can also encounter a definition where the number n denotes the maximum number of children, not the number of keys in a page.

B-trees – variants

B^+-tree

- all keys are stored only in leaves
- leaves are mutually linked by pointers – faster operation with contiguous key ranges, ”find all keys between 100 and 200”
- in Levitin’s book [2] is as a B-tree described precisely this variant.

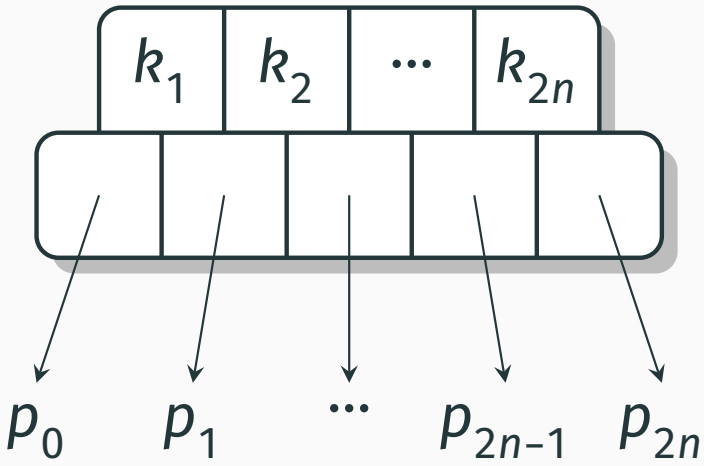
B^*-tree

- a page must be filled to at least two thirds,
- when inserting a key into a full page, keys are first moved between siblings,
- results in a smaller number of page splits.

B-trees – searching for a key x

1. At the beginning of the algorithm, we mark the root page as the current page P.
2. If page P does not exist, the search ends in failure.
3. Otherwise, assume that page P contains m keys $k_1, \ldots, k_m$ and corresponding child pointers $p_0, \ldots p_m$. Then:
   - 3.1 If $x = k_i$, for some $1 \leq i \leq m$, then the search ends in success.
   - 3.2 If $x < k_1$, then $P = p_0$ and back to point 2.
   - 3.3 If $x > k_m$, then $P = p_m$ and back to point 2.
   - 3.4 Otherwise, we find such i, $1 \leq i < m$, for which it holds that $k_i < x < k_{i+1}$. Then $P = p_i$ and back to point 2.

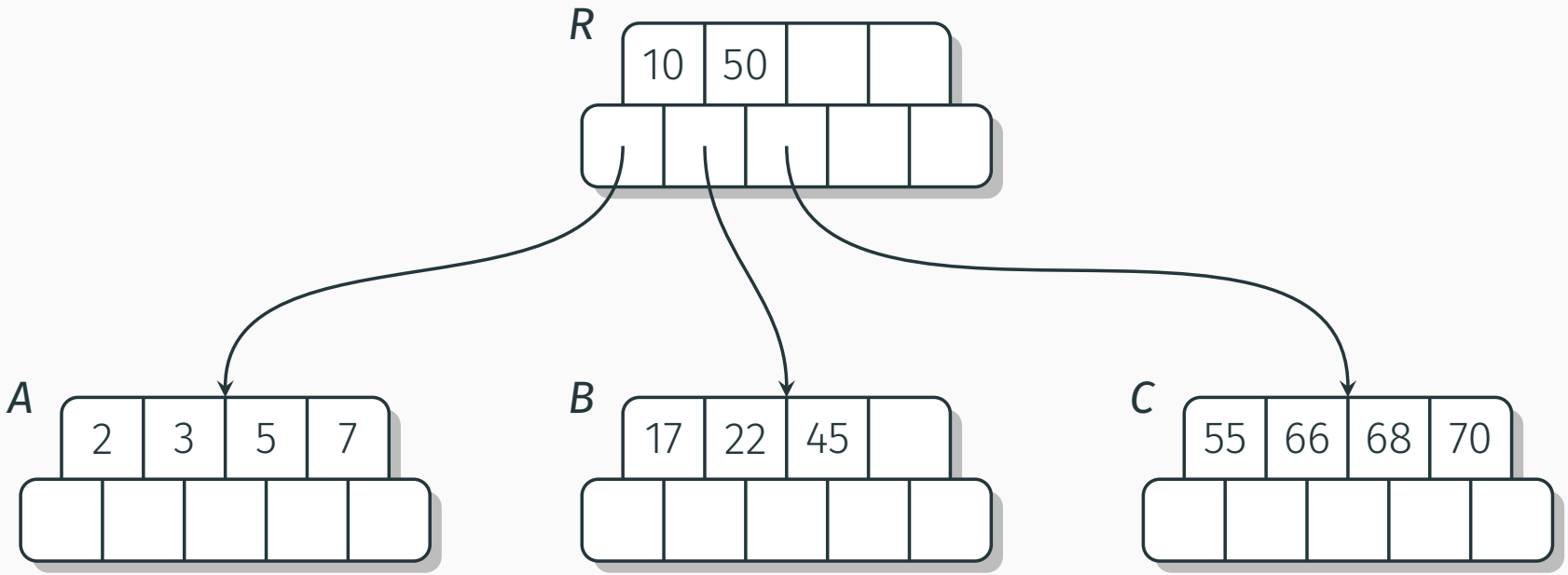
Examples of B-tree operations



In examples we will use B-tree for $n = 2$, meaning each page contains at least 2 and at most 4 keys.

Furthermore, each page refers to at least 3 and at most 5 children. The exception is the root of the tree.

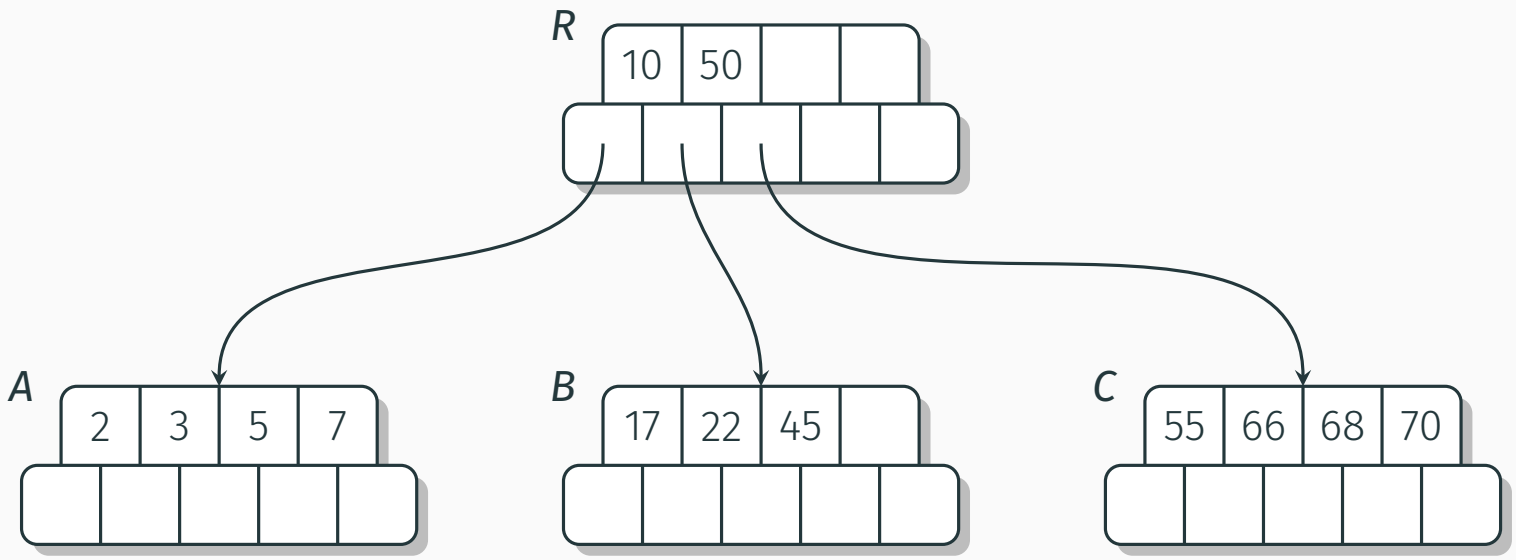
Example of searching in a B-tree – finding the key 50



Procedure

1. We start the search in the root R, so $P = R$.
2. Page P exists, we proceed to the next point.
3. Because $x = k_2$ the search ends in success.

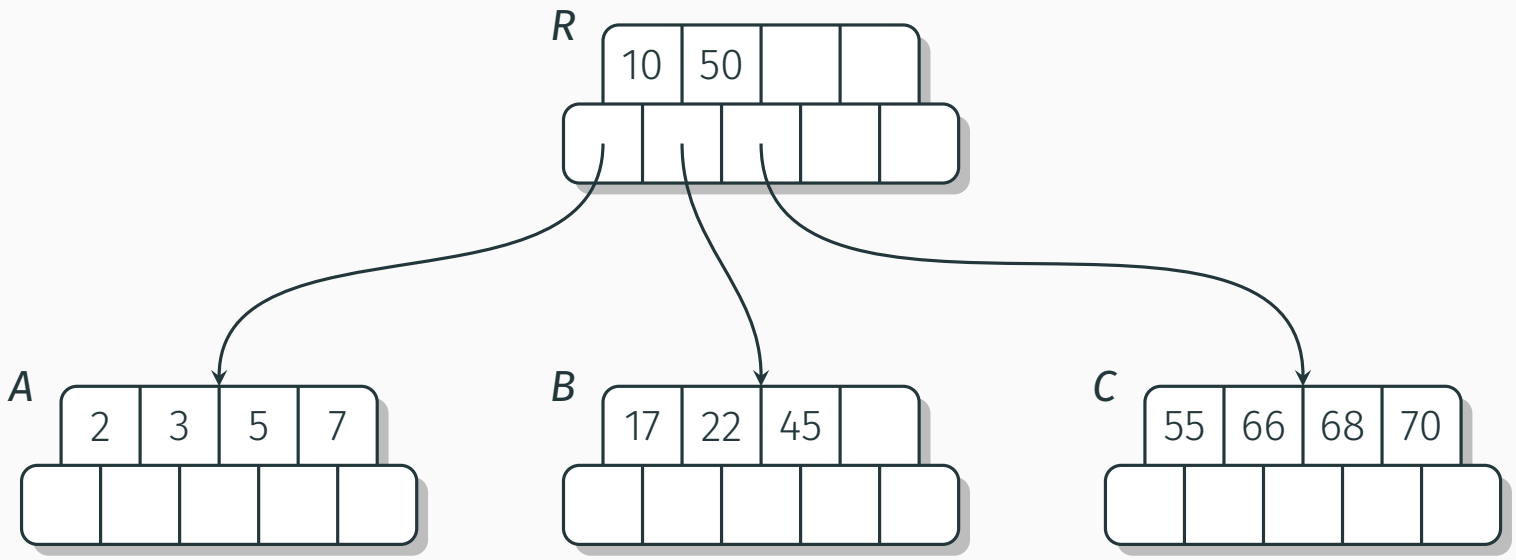
Example of searching in a B-tree – search for 3



Procedure

1. We start the search in the root R, thus $P = R$.
2. Page P exists, we proceed to the next point.
3. Since $x < k_1$, then $P = p_0 = A$.
4. Page P exists, we proceed to the next point.
5. Since $x = k_2$ the search ends in success.

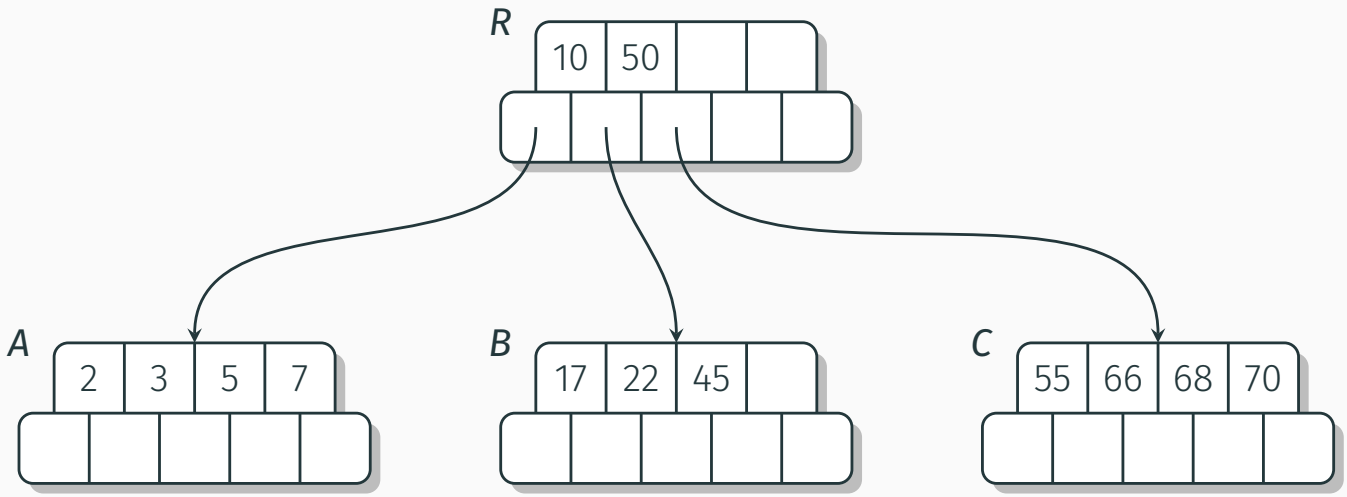
Example of searching in a B-tree – search for 45



Procedure

1. We begin the search at the root R, thus $P = R$.
2. Page P exists, we proceed to the next point.
3. Since $k_1 < x < k_2$, then $P = p_1 = B$.
4. Page P exists, we proceed to the next point.
5. Since $x = k_3$ the search ends in success.

Example of searching in a B-tree – search for 57



Procedure

1. We begin the search at the root R, thus $P = R$.
2. Page P exists, we proceed to the next point.
3. Since $x > k_2$, then $P = p_2 = C$.
4. Page P exists, we proceed to the next point.
5. Since $k_1 < x < k_2$, then $P = p_1 = null$.
6. Since P does not exist, the search ends in **failure**.

B-trees – inserting key x

1. First, it is necessary to determine, using the search algorithm, the leaf page L where the key x will be inserted.
2. Two cases can occur:
   - Page L is not completely filled – key x is inserted into the page so that the ordering of keys is preserved.
   - Page L is completely filled, then
     2.1 key x is sorted (e.g., in an auxiliary array) among the keys from page L so that the ordering of keys is preserved. We obtain a sequence of $2n+1$ keys $k'_1 < k'_2 < \cdots k'_{2n+1}$
     2.2 a new page P is created, with the same parent R as L
     2.3 distribution of keys to pages

Keys	Action
$k'_1, \ldots, k'_n$	remain in page L
k'_{n+1}	insert into parent page R
$k'_{n+2}, \ldots k'_{2n+1}$	insert into new page P

B-trees – insertion algorithm, notes

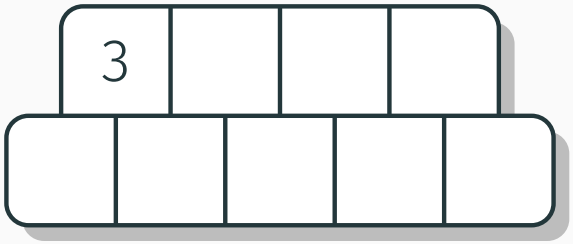
- The process of creating a new page and redistributing keys is called **page splitting**.
- By inserting the key k'_{n+1} into the parent page R, the number of keys in this page increases, which in turn increases the number of references to the child pages of this page. Without moving k'_{n+1}, the page R would lack a free reference for attaching the page P.
- The insertion of the key k'_{n+1} into page R is performed using the same algorithm as the insertion of key x into L. The insertion of k'_{n+1} can cause the page R to split.
- Page splitting can lead to the creation of a new root of the entire tree, which is the only way for a B-tree to increase its height.

Example of Insertion into a B-Tree

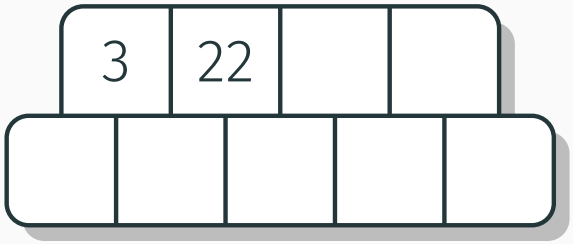
- In this more extensive example, we will gradually build a B-tree with the same parameters as in the search example.
- We will gradually insert the keys 3, 22, 10, 2, 17, 5, 66, 68, 50, 7, 55, 45, 70, 44, 6, 21, 67, 1, 4, 8, 9, 12, and 15 into the tree.

Example of insertion into a B-tree – insertion of keys 3, 22 and 10

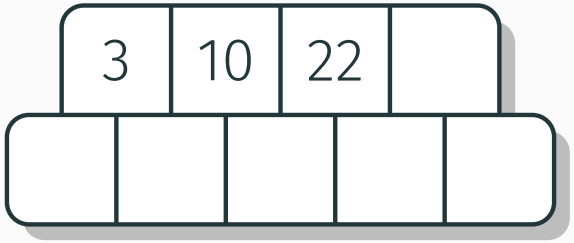
Insertion of key 3



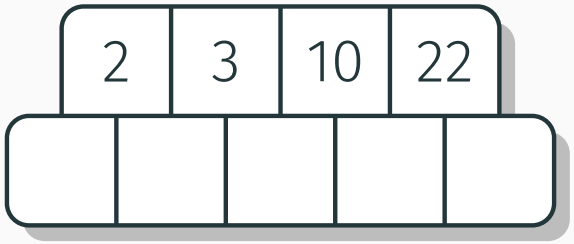
Insertion of key 22



Insertion of key 10

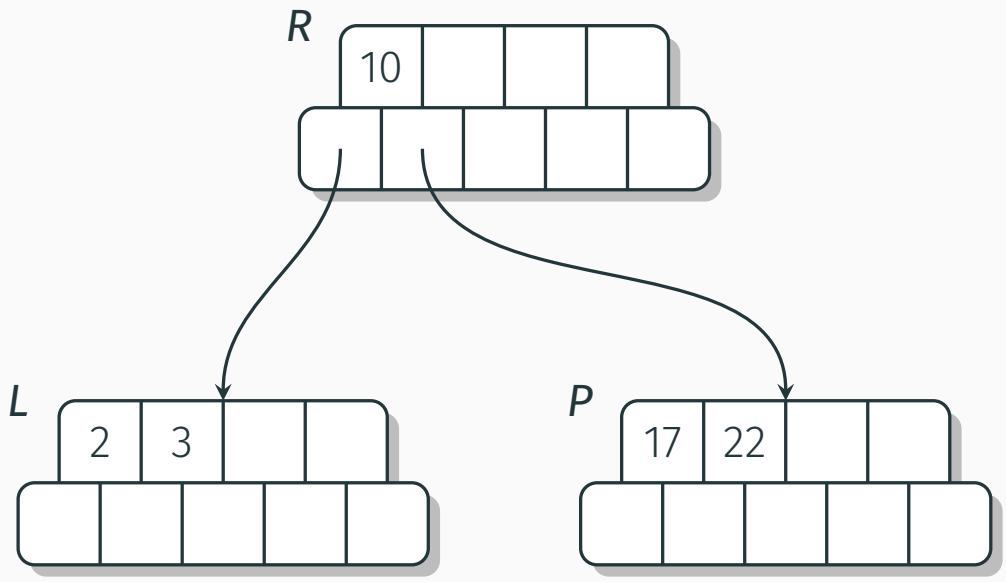


Example of insertion into a B-tree – insertion of key 2



The page is completely full, inserting any additional key will cause a change in the structure of the B-tree.

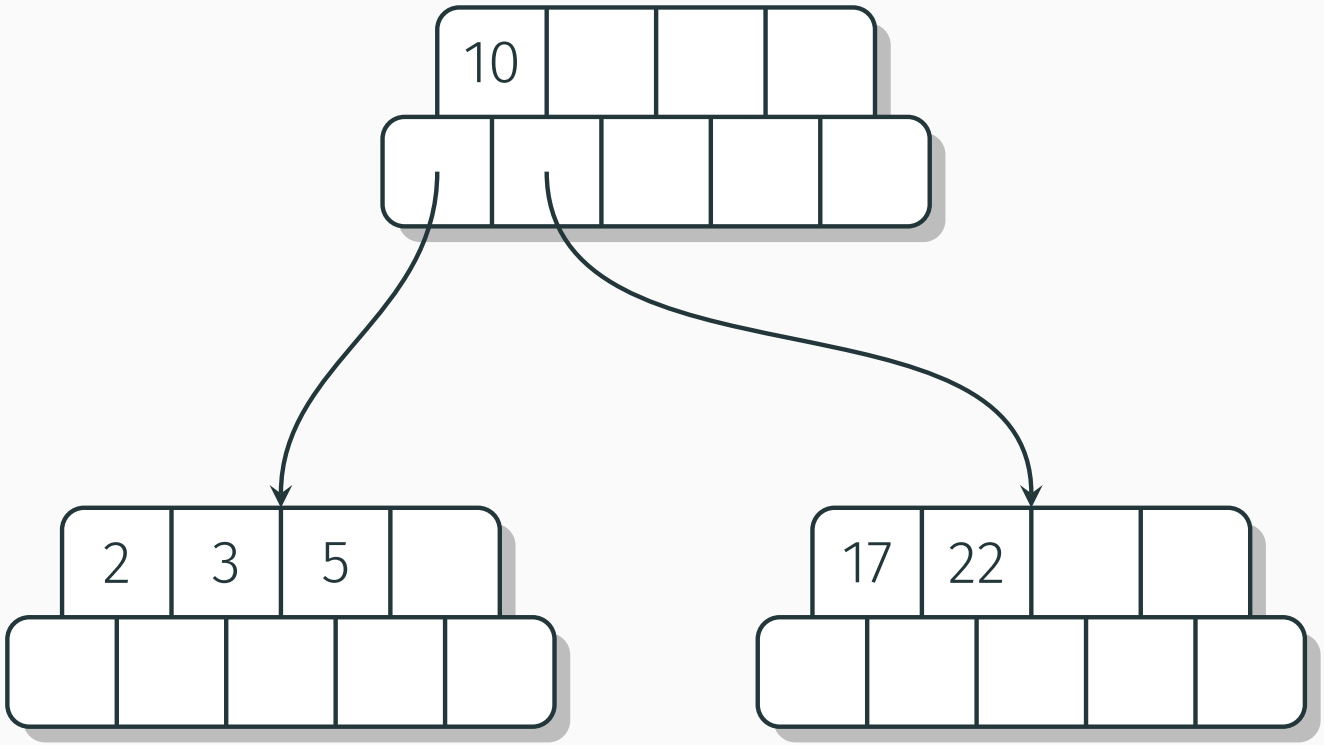
Example of insertion into a B-tree – insertion of key 17



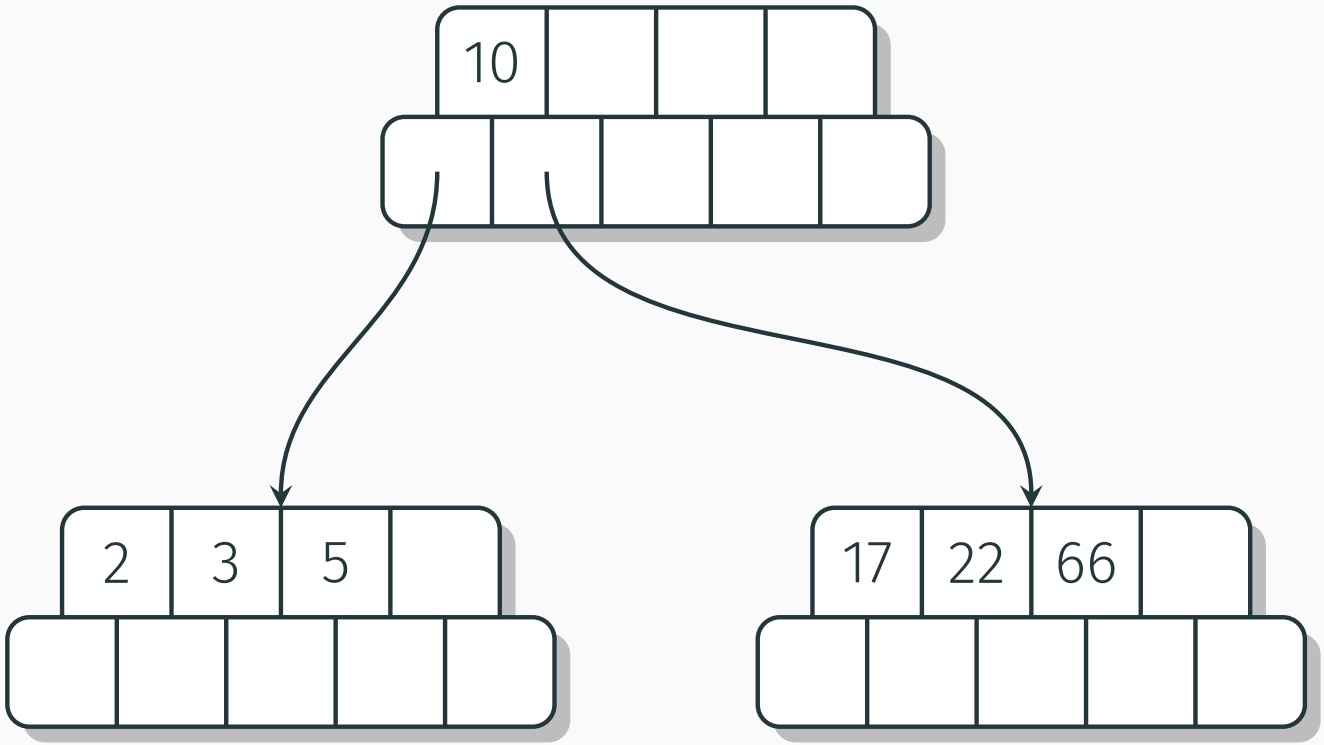
By inserting the key 17, the following occurred:

1. the page *L* was split and half of the keys were moved to a new page *P*,
2. a new root page *R* was created and the key 10 was moved to the new root.

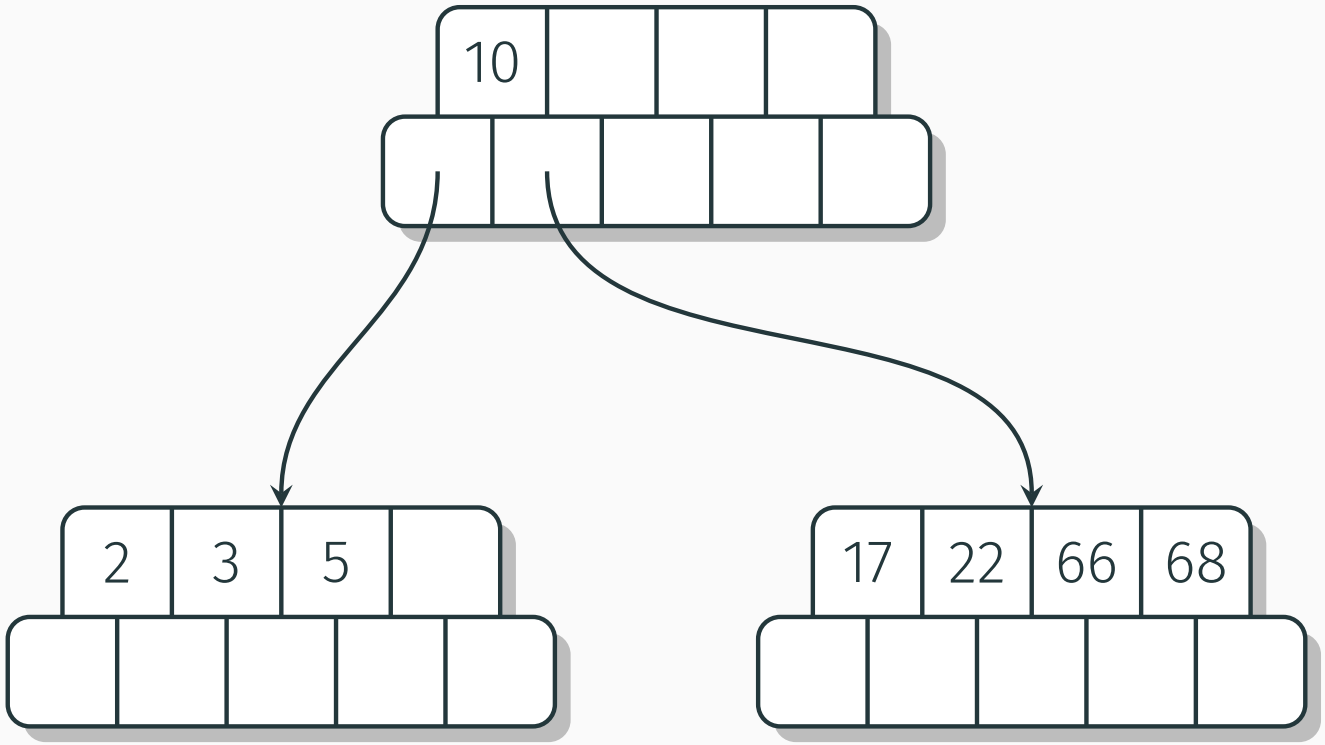
Example of insertion into a B-tree – insertion of key 5



Example of insertion into a B-tree – insertion of key 66

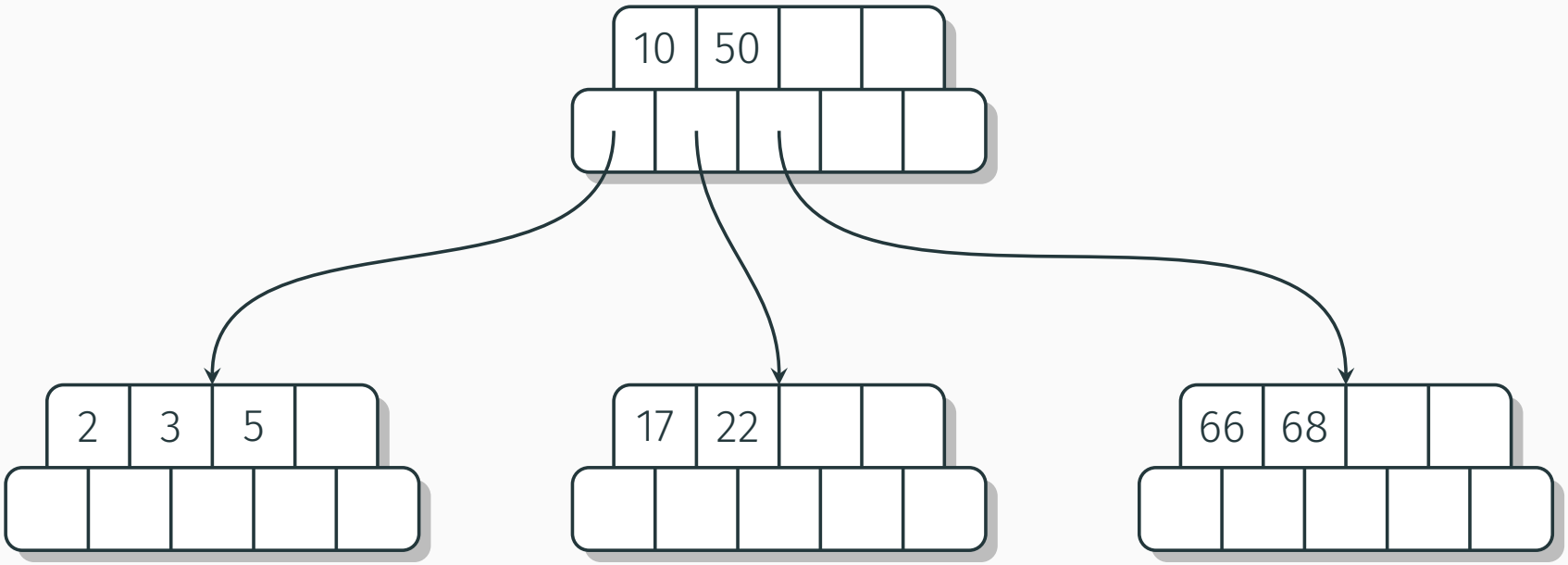


Example of insertion into a B-tree – insertion of key 68



The page with keys 17 to 68 is completely full, inserting another key into this page will cause a change in the structure of the B-tree.

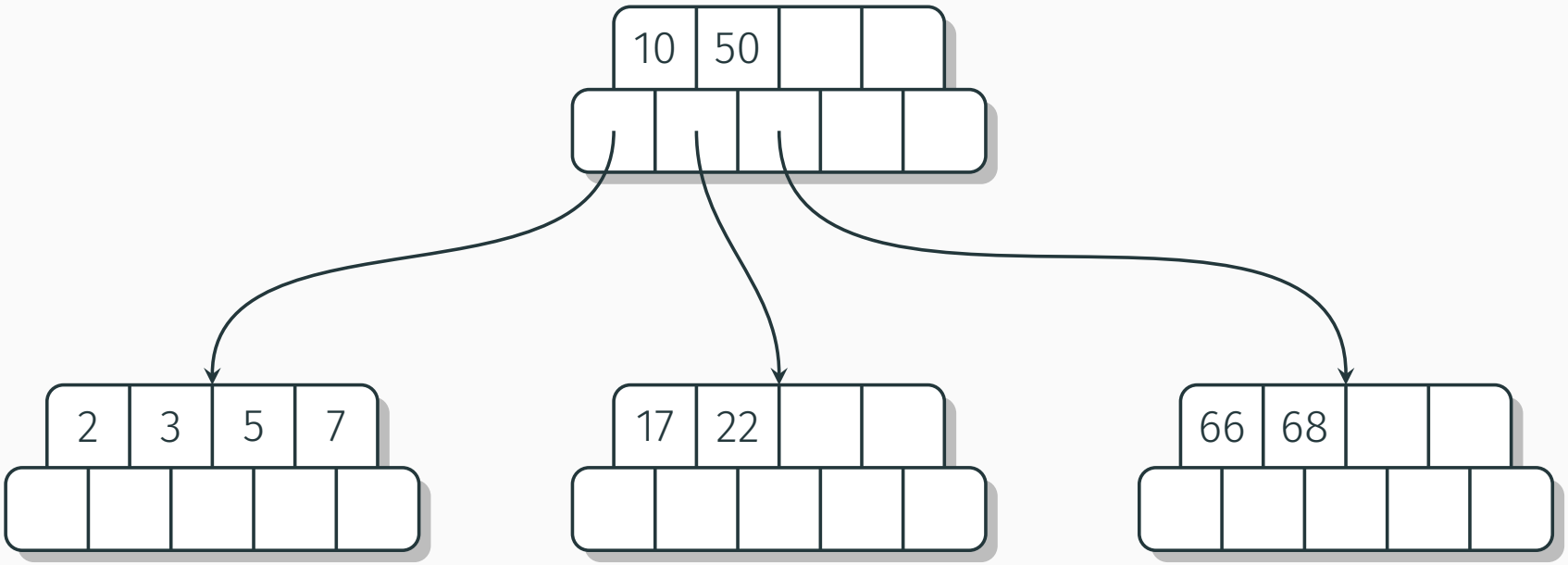
Example of insertion into a B-tree – insertion of key 50



By inserting the key 50, the following occurred:

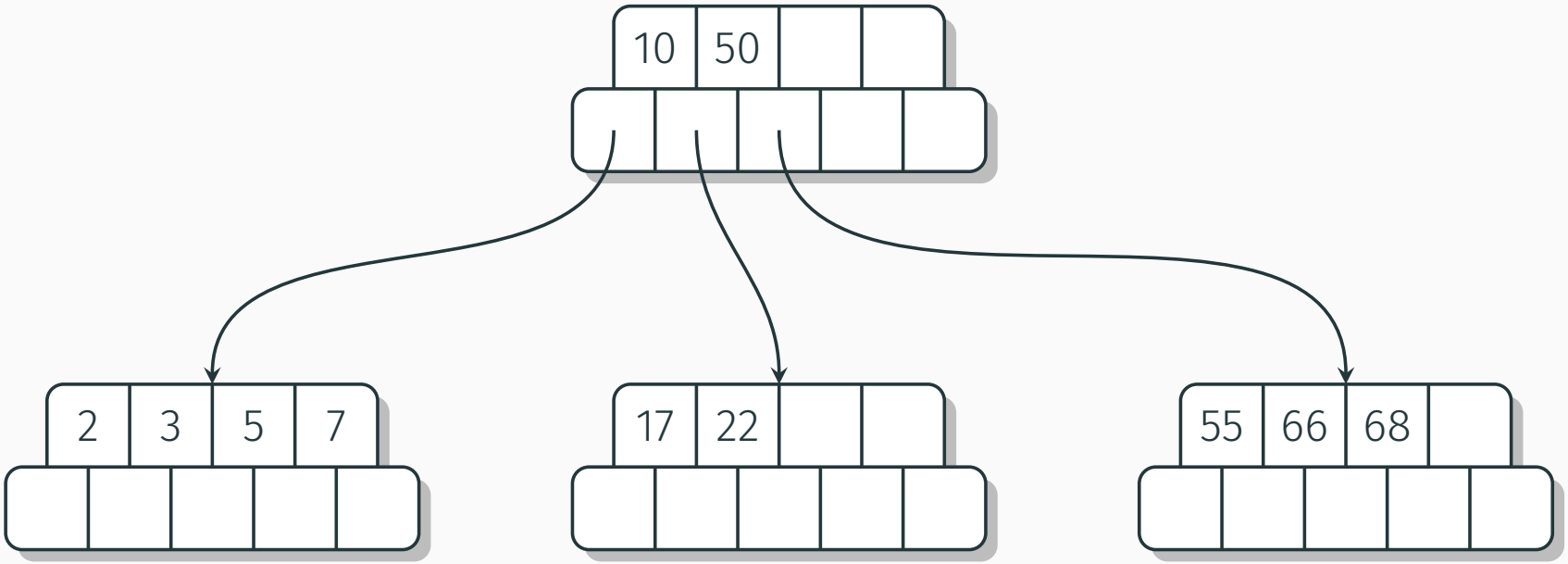
1. the page split and half of the keys were moved to a new page, and
2. at the same time, the newly inserted key 50, being the median of the values in the original page, was moved to the root page.

Example of insertion into a B-tree – inserting key 7

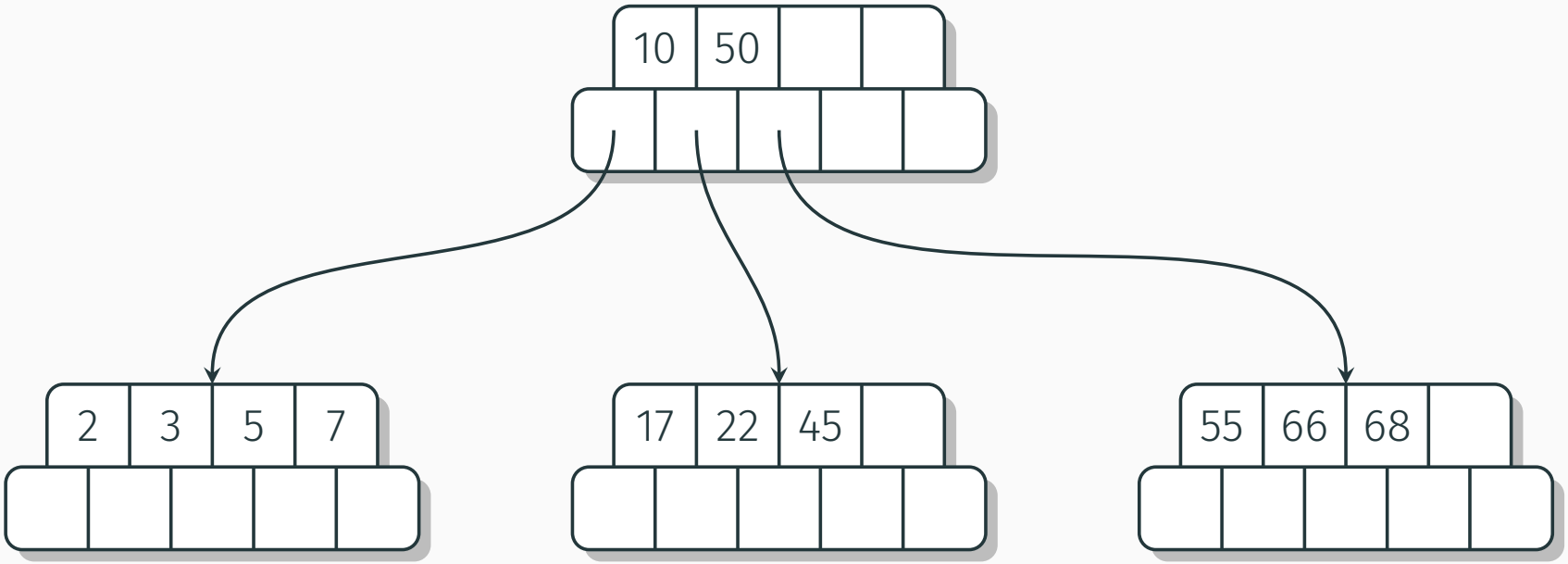


The page with keys 2 to 7 is completely full, inserting another key into this page will cause a change in the structure of the B-tree.

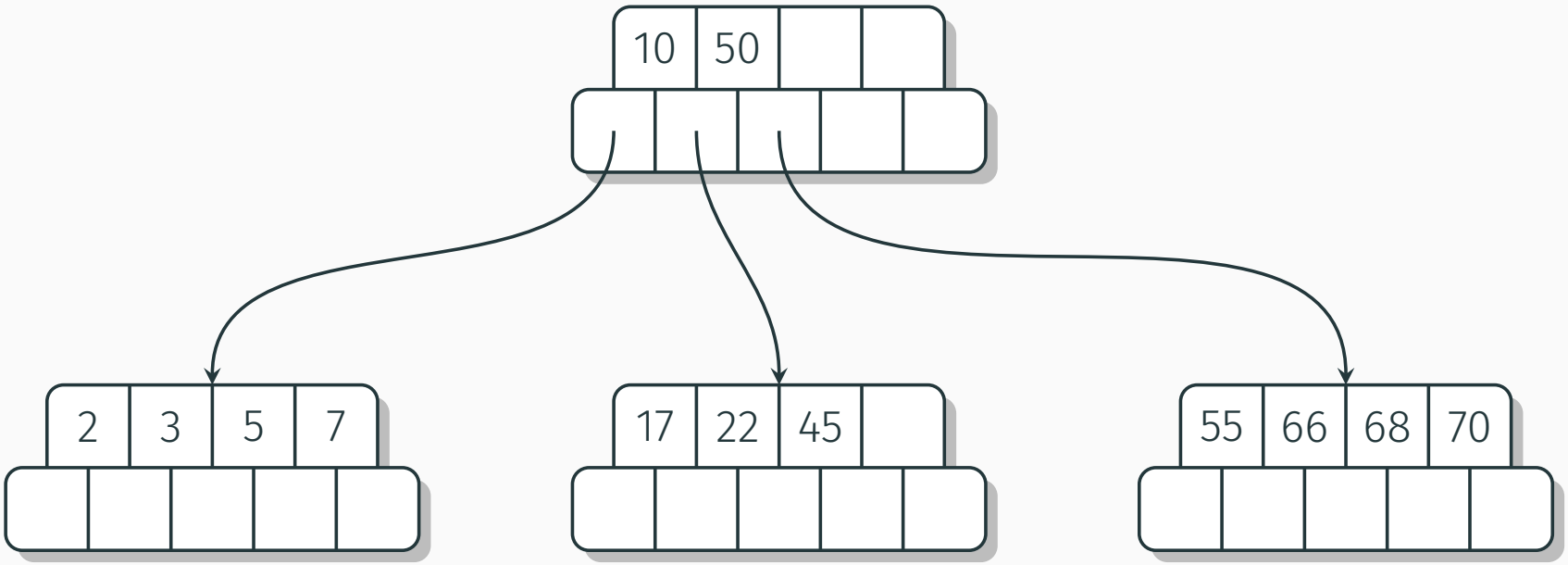
Example of insertion into a B-tree – insertion of key 55



Example of insertion into a B-tree – insertion of key 45

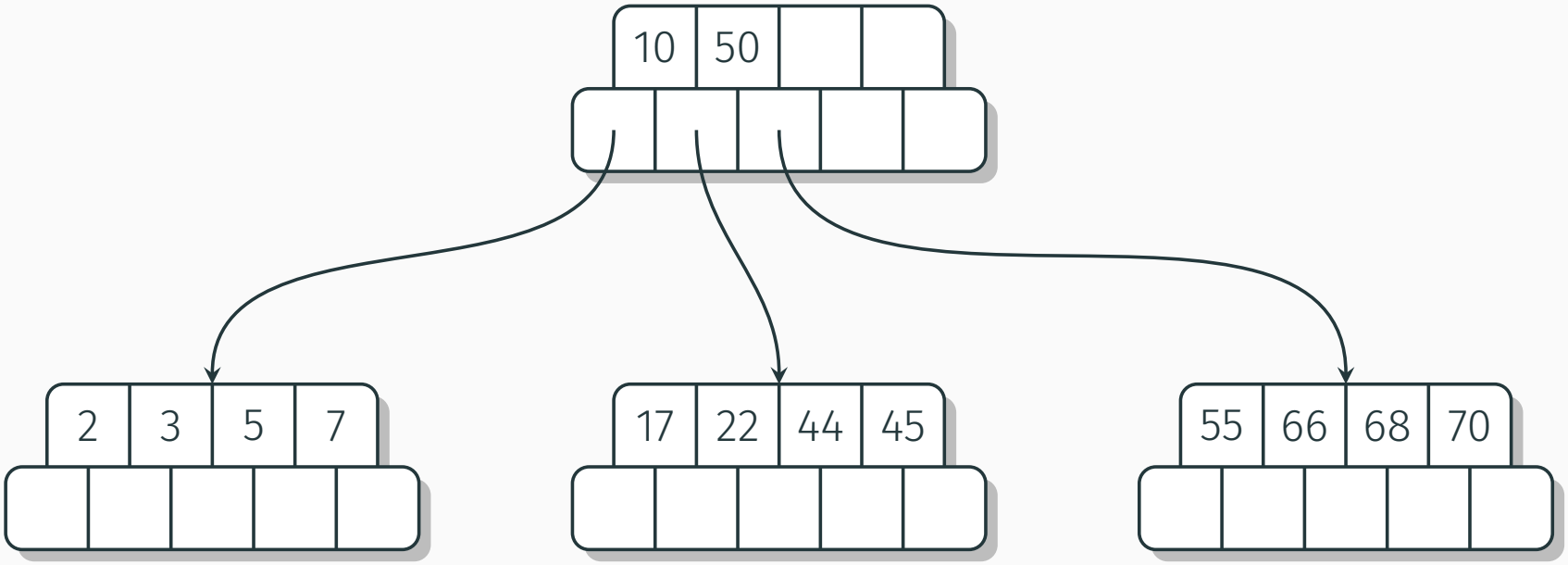


Example of insertion into a B-tree – insertion of key 70



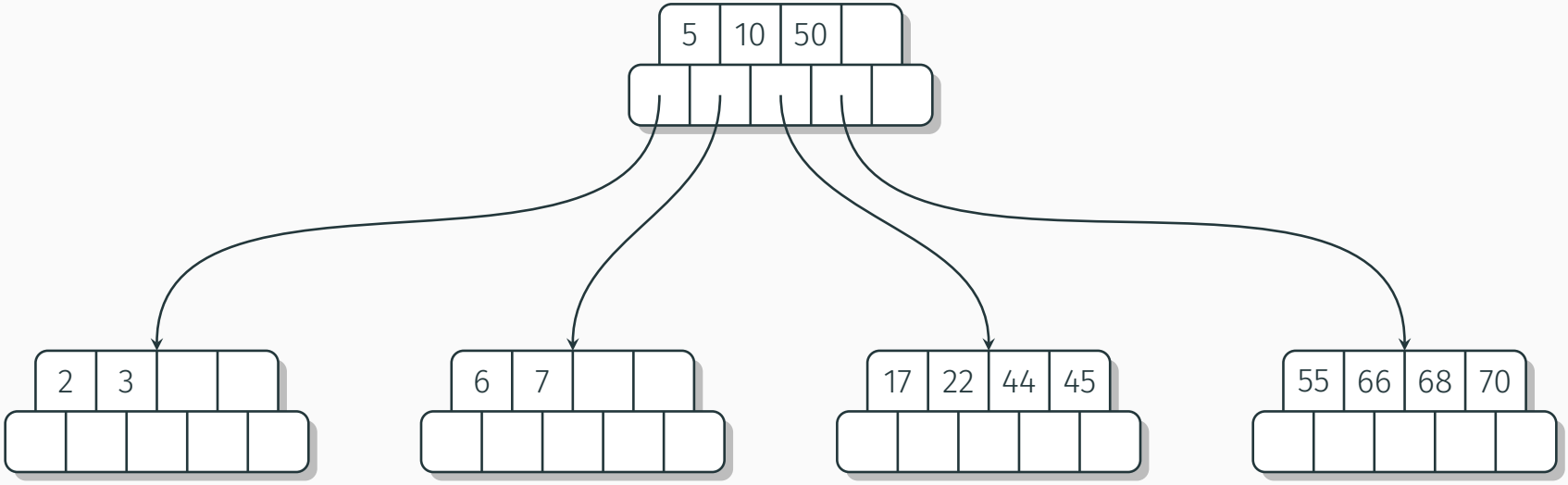
The page with keys 55 to 70 is completely full, inserting another key into this page will cause a change in the structure of the B-tree.

Example of insertion into a B-tree – insertion of key 44



All leaf pages are completely filled, inserting any additional key will cause a change in the structure of the B-tree.

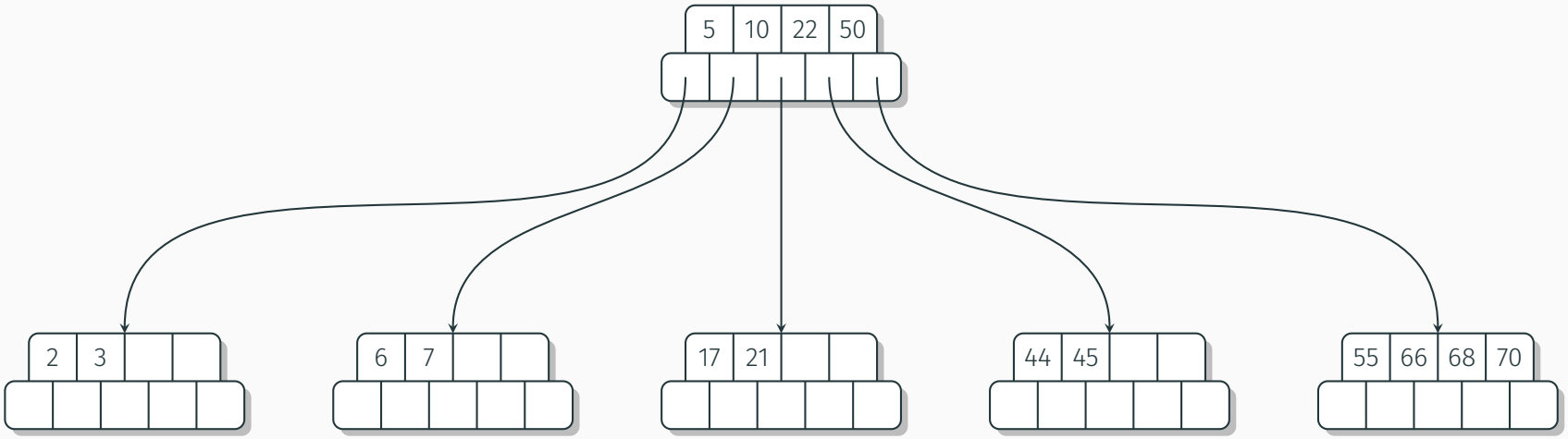
Example of insertion into a B-tree – insertion of key 6



Upon inserting key 6, the following occurred:

1. the page split and half of the keys were moved to a new page, and
2. simultaneously, key 5 was moved to the root page.

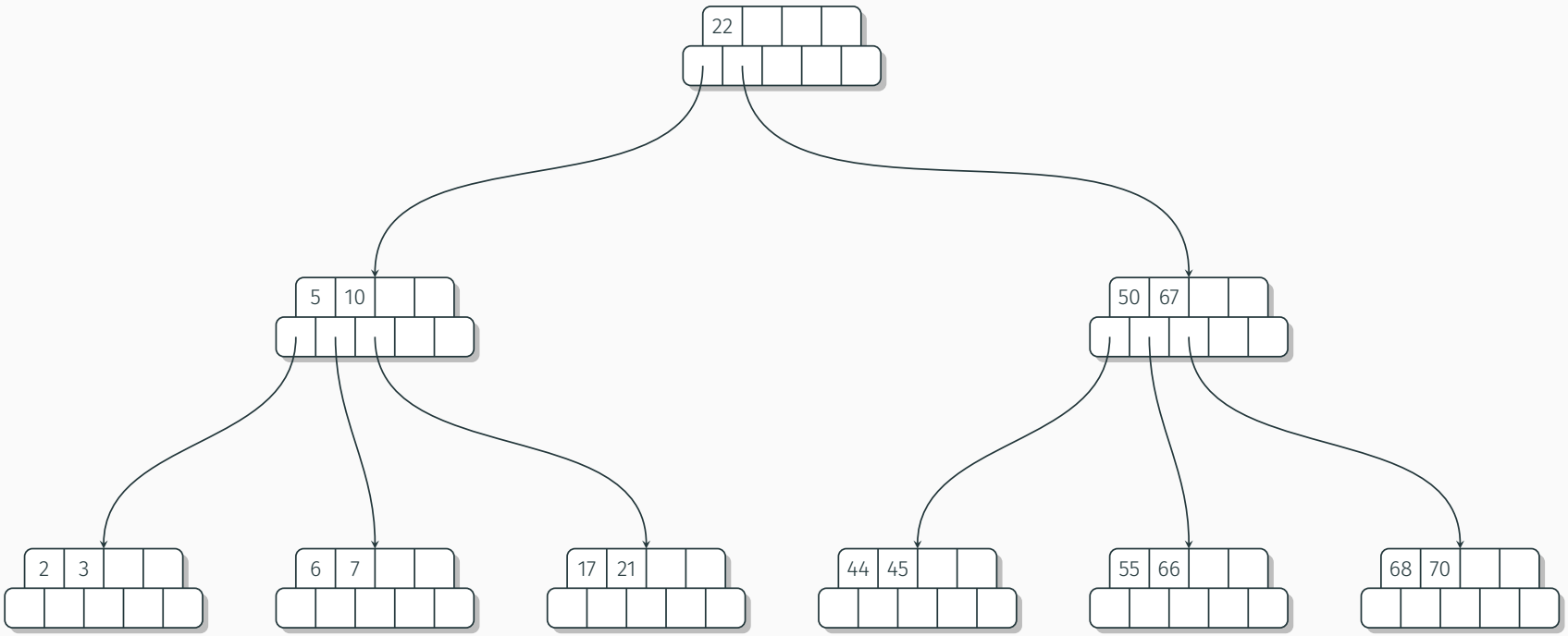
Example of insertion into a B-tree – insertion of key 21



Upon the insertion of key 21, the following occurred:

1. the page split and half of the keys were moved to a new page, and
2. key 22 was moved to the parent page.
3. At the same time, the root page of the tree became full.

Example of insertion into a B-tree – insertion of key 67



Example of insertion into a B-tree – insertion of key 67 (cont.)

Upon inserting key 67, the following occurred:

1. the page split and half of the keys were moved to a new page, and
2. simultaneously, the newly inserted key 67, being the median value in the original page, was moved to the parent page.
3. Since this page was also fully occupied, it split, resulting in the creation of a new root page with a single key 22.

Example of insertion into a B-tree – insertion of key 67 (cont.)

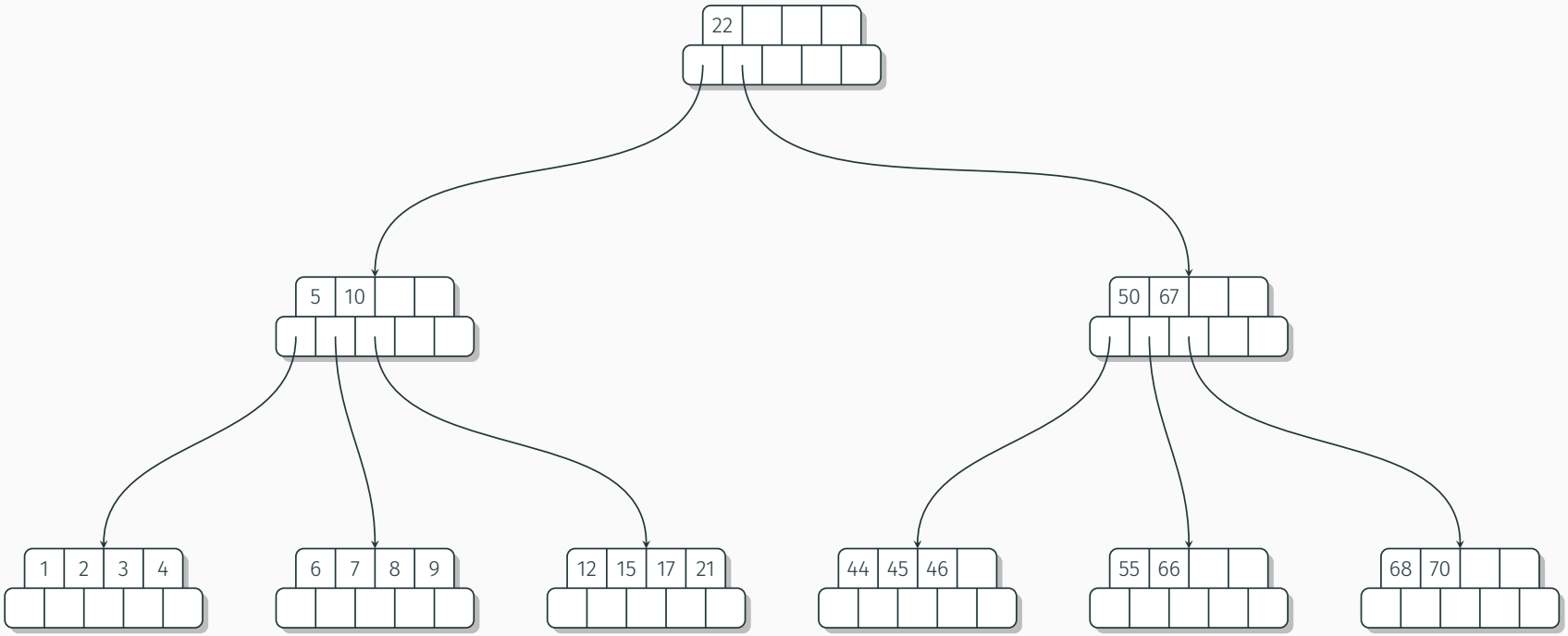
Remarks

- At this point, the B-tree's fill factor reaches its minimum value of approximately 50%. The B-tree has maximum free space for inserting additional keys.
- However, the minimal filling of the B-tree will cause the removal of any key to result in page merging, including the cancellation of the root and a subsequent decrease in the height of the B-tree[2].

[2] See the next part of the presentation

Example of insertion into a B-tree – insertion of additional keys

Into the tree were further inserted keys 1, 4, 8, 9, 12, 15 and 46.
The order of key insertion, in this case, does not matter.



B-trees – deletion of key x

1. First, it is necessary to find the key x in the tree.
2. Let us denote the page with key x as P.
3. Two cases can occur:
   - page P is an **internal page** of the tree or
   - page P is a **leaf page**.

B-trees – deletion of key x from internal page P

1. We replace key x in page P with the closest larger key y to it.
2. Key y must be located in the subtree with keys greater than x and, at the same time, is the smallest among these keys, so it must be located in a leaf page.
3. We have thus reduced the deletion of key x from an internal page of the tree to the deletion of key y from a leaf page of the tree.

B-trees – deleting key x from leaf page P

1. We delete key x from page P.
2. If page P still contains at least n keys after deletion, the deletion process is terminated.
3. If P then contains only $n - 1$ keys, we must replenish the missing key.
   - 3.1 We determine the number of keys in the sibling page of P. We denote the sibling as S. The common parent of pages P and S is denoted as R.
   - 3.2 If there are **more than** n keys in S, then
     - 3.2.1 we move the nearest larger key than x from R to P and
     - 3.2.2 we move the smallest key from S to R.
   - 3.3 If there are **exactly** n keys in S, then

B-trees – deleting key x from leaf page P (cont.)

3.3.1 we move keys from page S to page P and obtain one page with $2n - 1$ keys.

3.3.2 We eliminate page S.

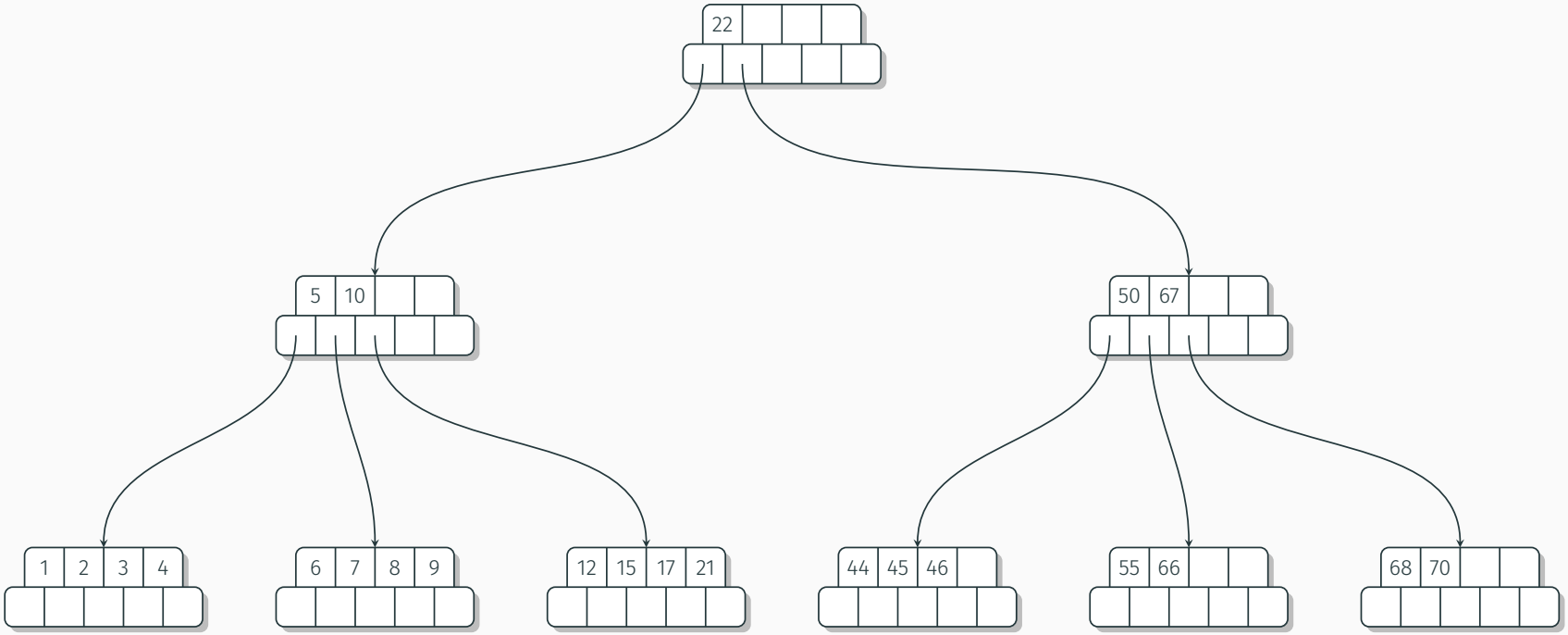
3.3.3 In page R, there is now one redundant pointer to a page. We move the nearest larger key than x from R to page P, which now contains exactly $2n$ keys.

B-trees – deleting key x from leaf page P (cont.)

Remarks

- Usually, we choose a sibling with larger keys than x, i.e., the sibling to the "right" of P. In the previous explanation, we assumed this choice.
- However, it is possible to choose a sibling with smaller keys, i.e., the one to the "left" of P. The further procedure is a mirror image of the "right" sibling.
- The process of moving keys from S to P and subsequent elimination of page S is called **page merging**.
- The process of page merging can continue progressively up to the root of the tree and may lead to the extinction of the current root of the tree. The new root of the tree will then be the page resulting from the merging process. The B-tree thus reduces its height.

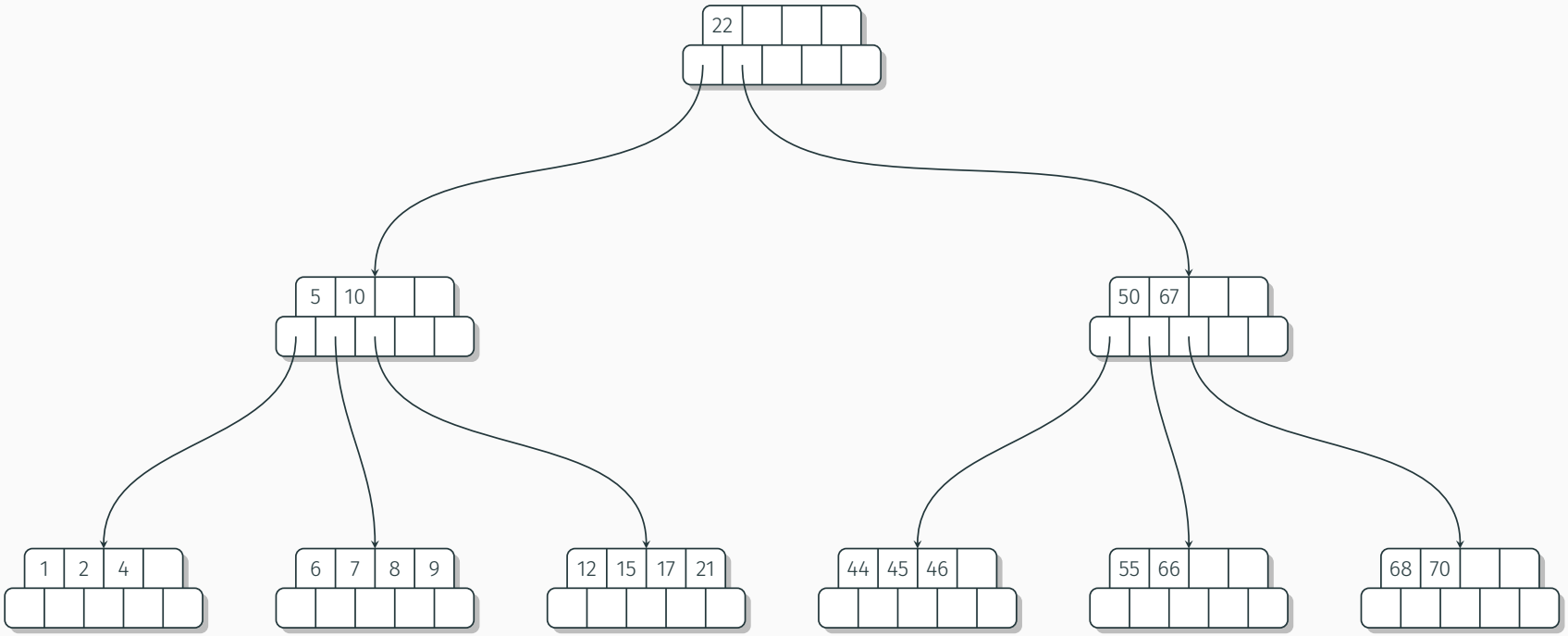
Example of deletion in a B-tree – initial B-tree



In this state, we have left the B-tree at the end of the example of inserting keys into the B-tree. Now we will gradually delete keys from the B-tree.

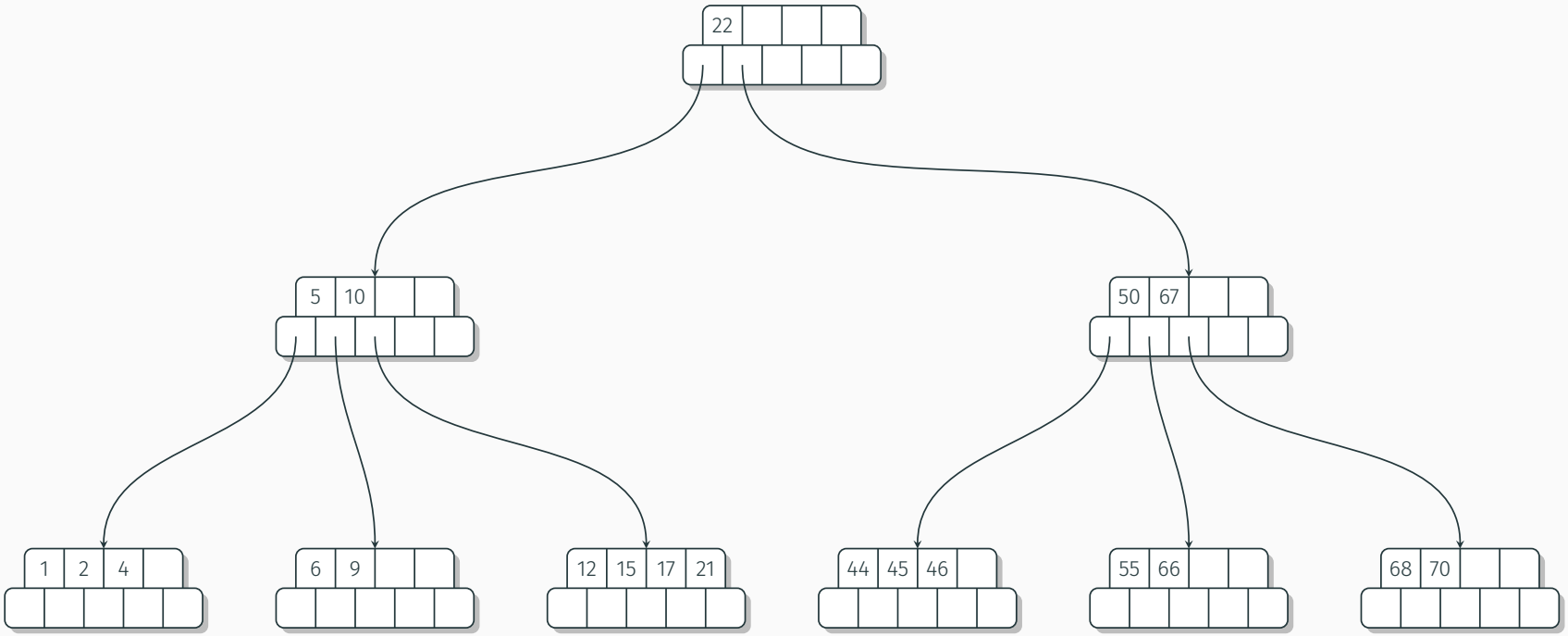
Example of deletion in a B-tree – deletion of key 3

Key 3 is located in a leaf page, where there are enough keys to simply delete key 3. This results in the following B-tree.



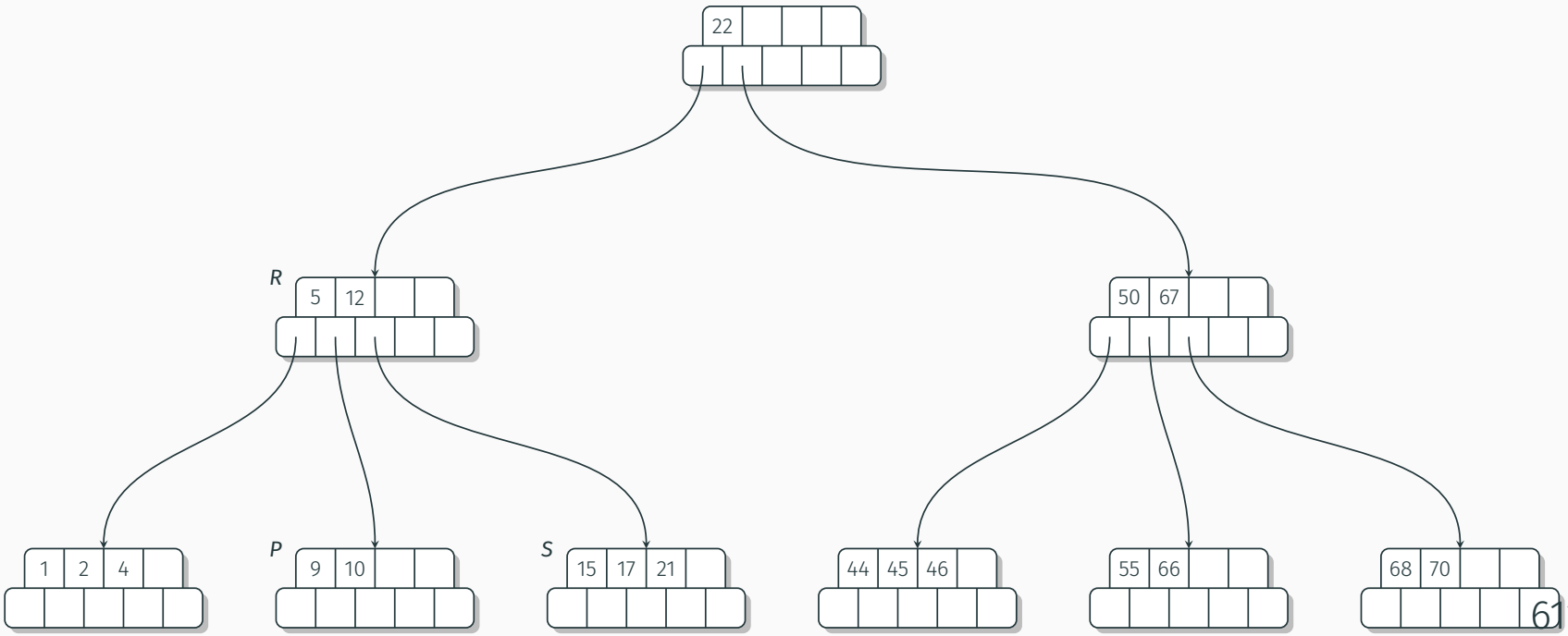
Example of deletion in a B-tree – deletion of keys 7 and 8

In the same way, we delete keys 7, 8 and obtain



Example of deletion in a B-tree – deletion of key 6

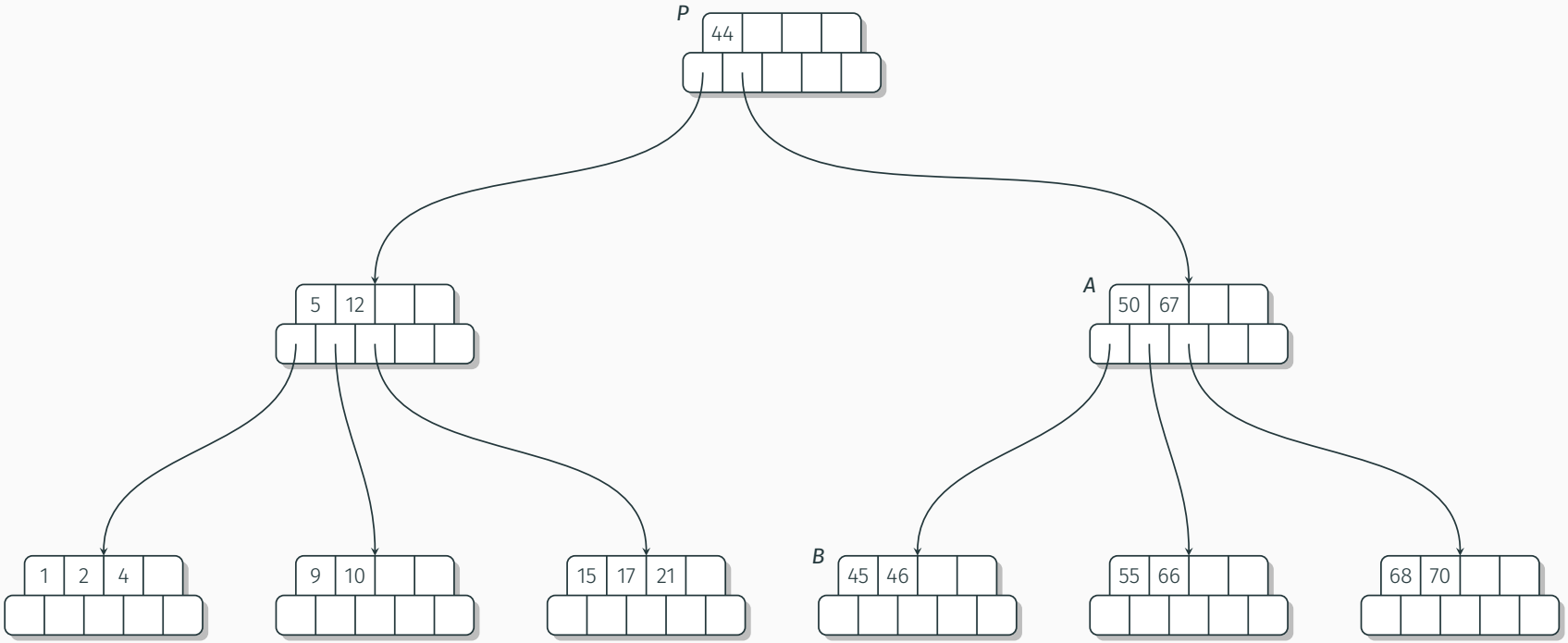
1. After deleting key 6, page P contains $n - 1 = 1$ key, number 9.
2. Sibling S contains more than n keys.
3. The nearest larger key than 6, i.e. 10, is moved from R to P.
4. The smallest key from S, i.e. 12, is moved to R.



Example of deletion in a B-tree – deleting key 22

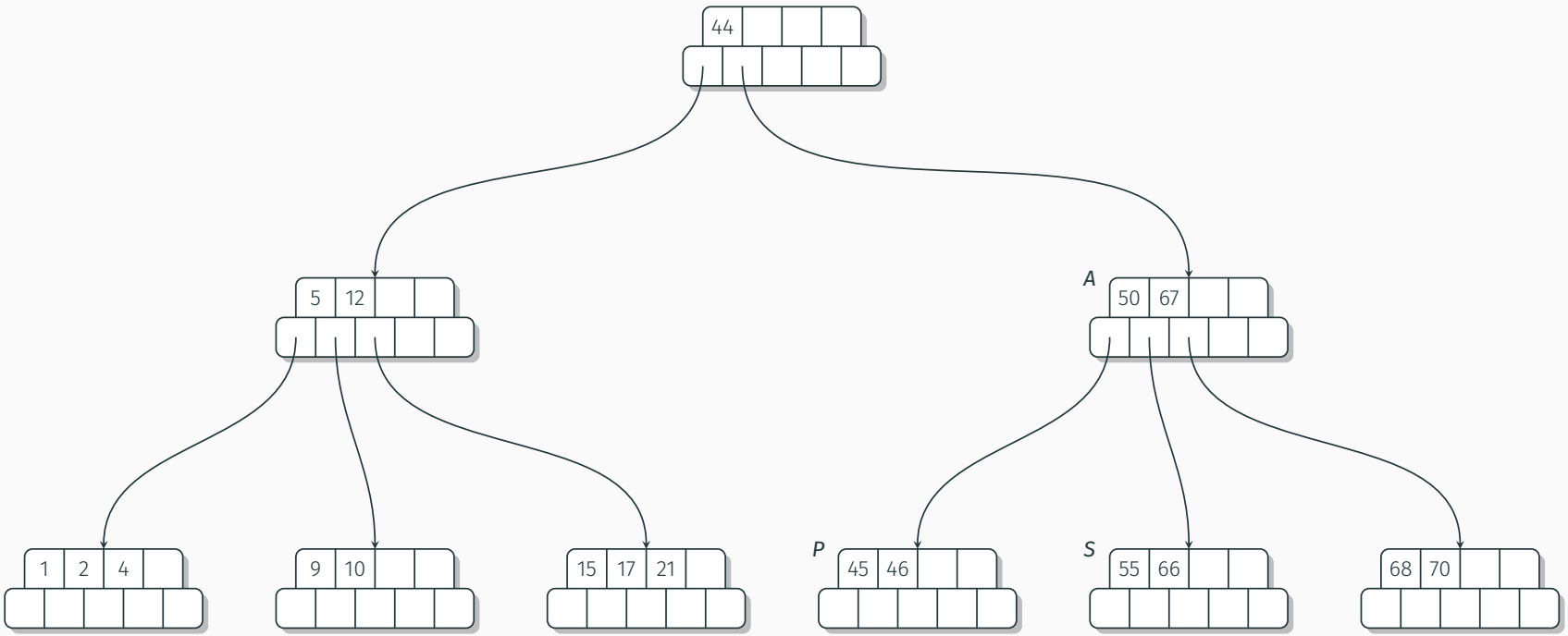
1. Key 22 is located in the internal page *P*, see the following figure.
2. We replace it with the nearest larger key – larger keys than 22 are in the subtree rooted at page *A*. From there, we proceed to the leftmost leaf page, in our case to *B*.
3. We select the smallest key in *B*, i.e., 44.
4. We have thus reduced the deletion of key 22 to the deletion of key 44.
5. After performing all operations corresponding to the deletion of 44 (in this case, it only involves deleting 44 from page *B*), we replace key 22 with key 44.

Example of deletion in a B-tree – deleting key 22 (cont.)



Example of deletion in a B-tree – deletion of key 46, phase I

State of the B-tree before deletion begins

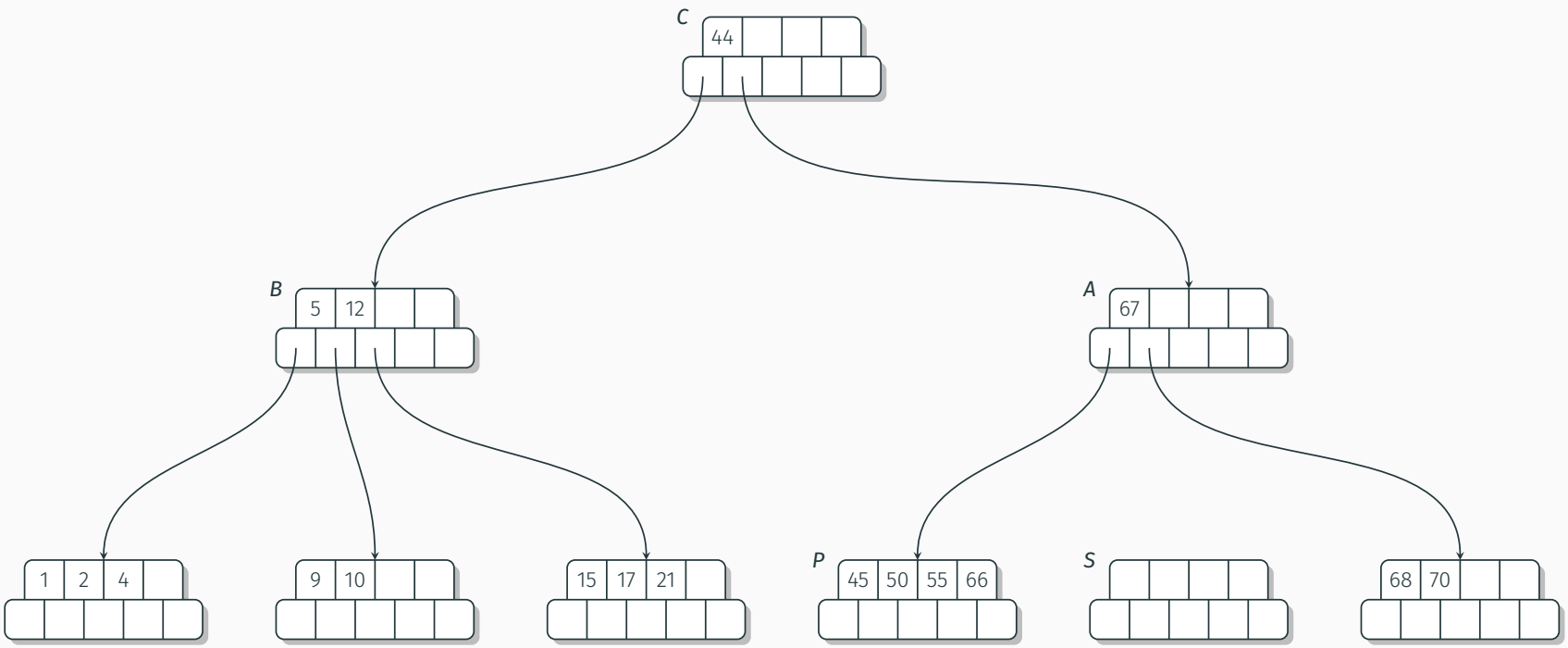


Example of deletion in a B-tree – deletion of key 46, phase I (cont.)

1. After deleting key 46, page P contains $n - 1$ keys, i.e., only key 45.
2. Sibling S contains exactly less than n keys, we must merge pages.
3. We move all keys from S to P.
4. Page P is the first child of page A, so we also move the first key from A to P.

Example of deletion in a B-tree – deletion of key 46, phase I (cont.)

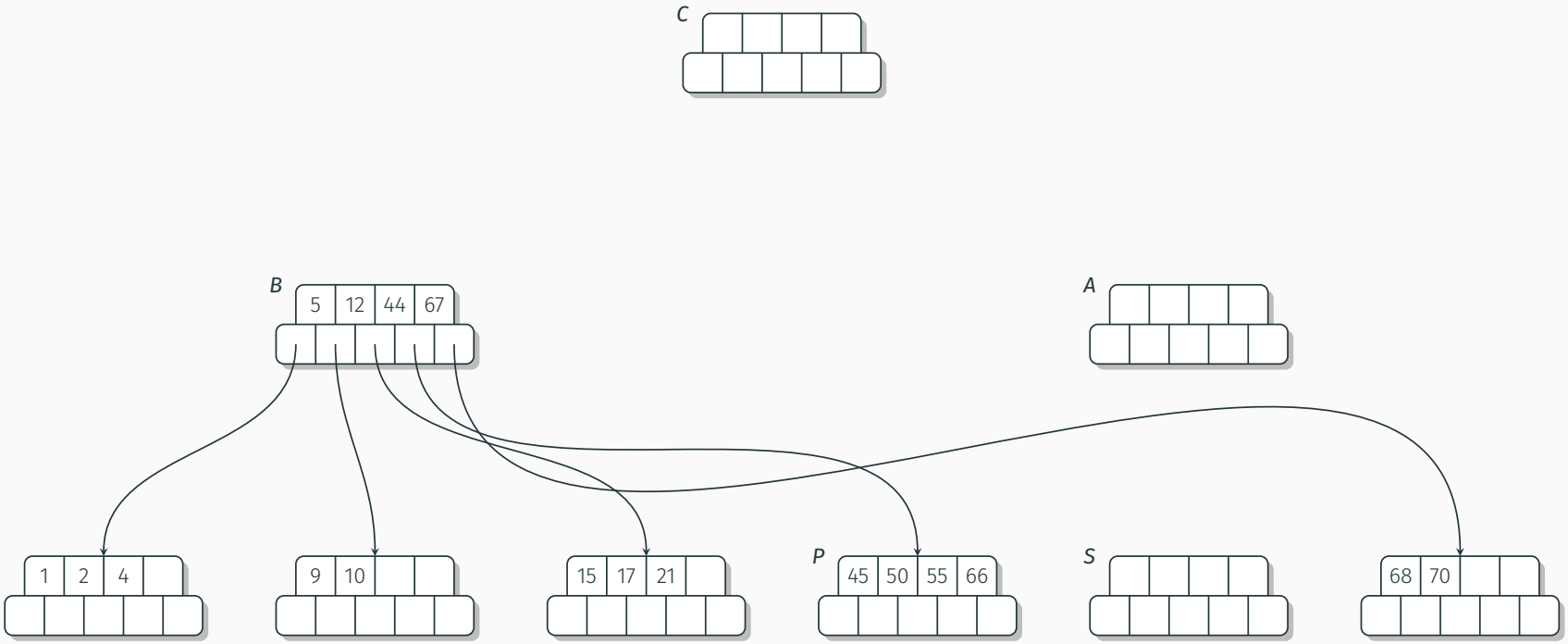
Result of phase 1



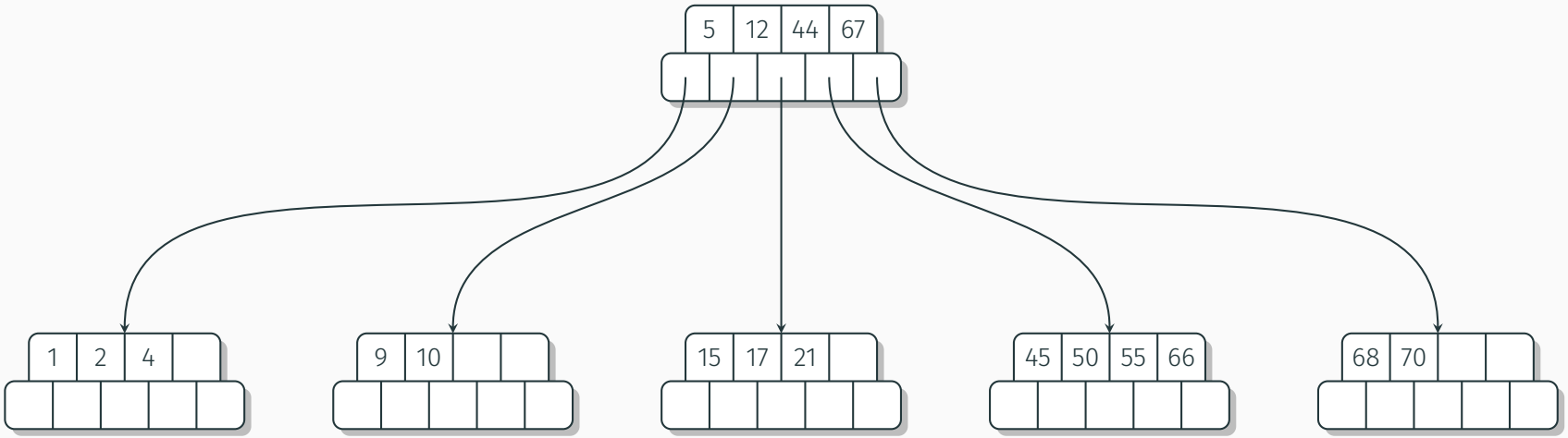
Example of deletion in a B-tree – deletion of key 46, phase II

1. In page A, only $n - 1$ keys remain, which contradicts the definition of a B-tree.
2. Sibling B contains exactly n keys, so we must also perform page merging at this level.
3. We move key 67 from page A to page B.
4. And similarly, we also move one key from the parent page C to B.
5. This results in the elimination of the root page and a decrease in the height of the B-tree.

Example of deletion in a B-tree – deletion of key 46, phase II (cont.)

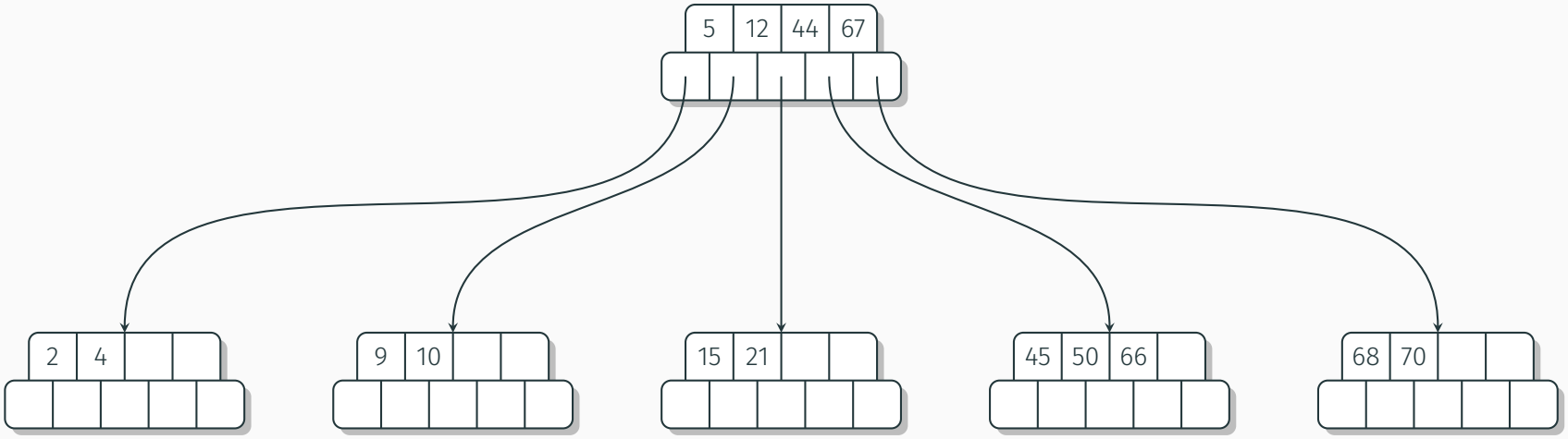


Example of deletion in a B-tree – deletion of key 46, result



Example of deletion in a B-tree – deletion of keys 1, 17, and 55

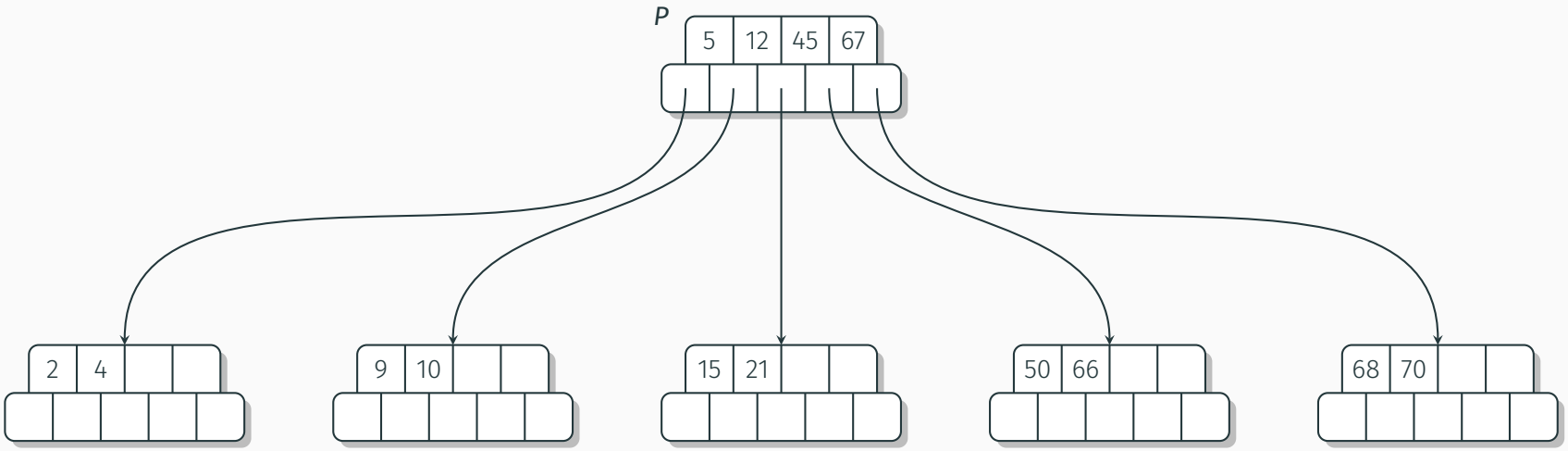
.Keys 1, 17, and 55 are located in leaf pages, where there is a sufficient number of keys to simply delete them.



Example of deletion in a B-tree – deletion of key 44

1. Key 44 is located on an internal page.
2. We replace it with the nearest larger key, i.e. key 45.

Resulting tree

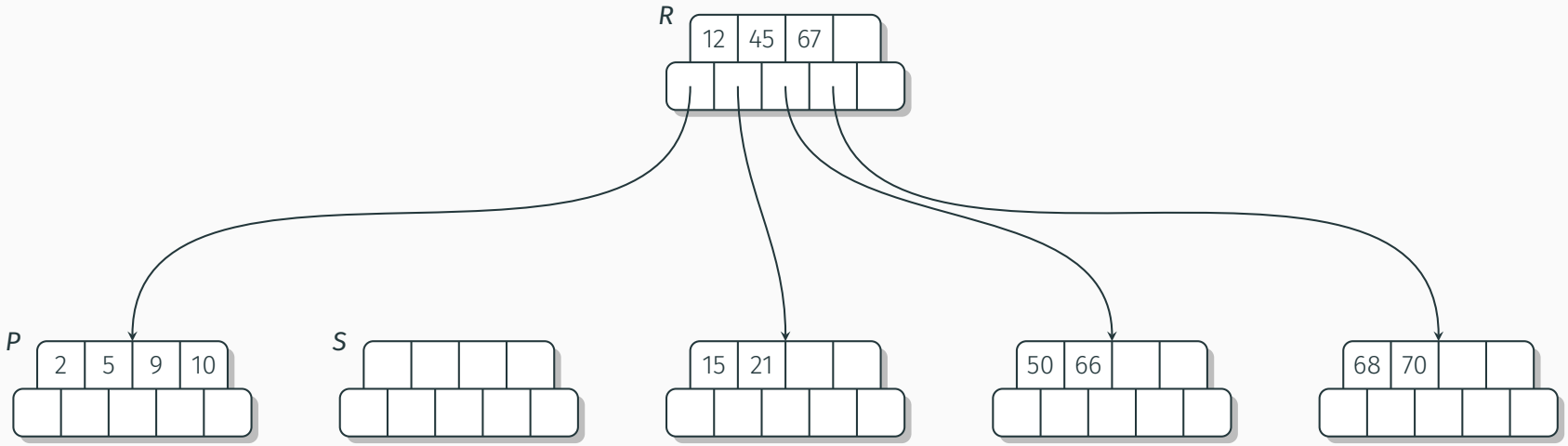


The B-tree is now in a state where the leaf pages are filled to the minimum acceptable level. Deletion of any key will cause page merging.

Example of deletion in a B-tree – deletion of key 4

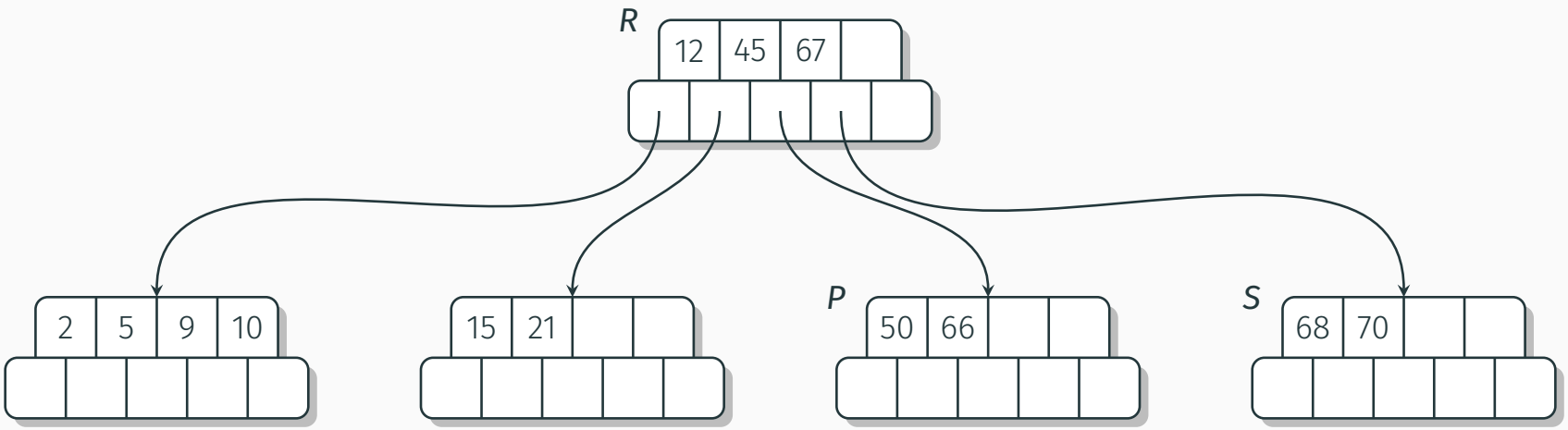
1. After deleting key 4, page P contains only $n - 1$ keys.
2. Sibling S contains exactly n keys.
3. We move the keys from S to P.
4. We also move key 5 from the parent page R to P, because otherwise one child pointer in R would be redundant.

Resulting tree



Example of deletion in a B-tree – deletion of key 45

The tree before deleting key 45



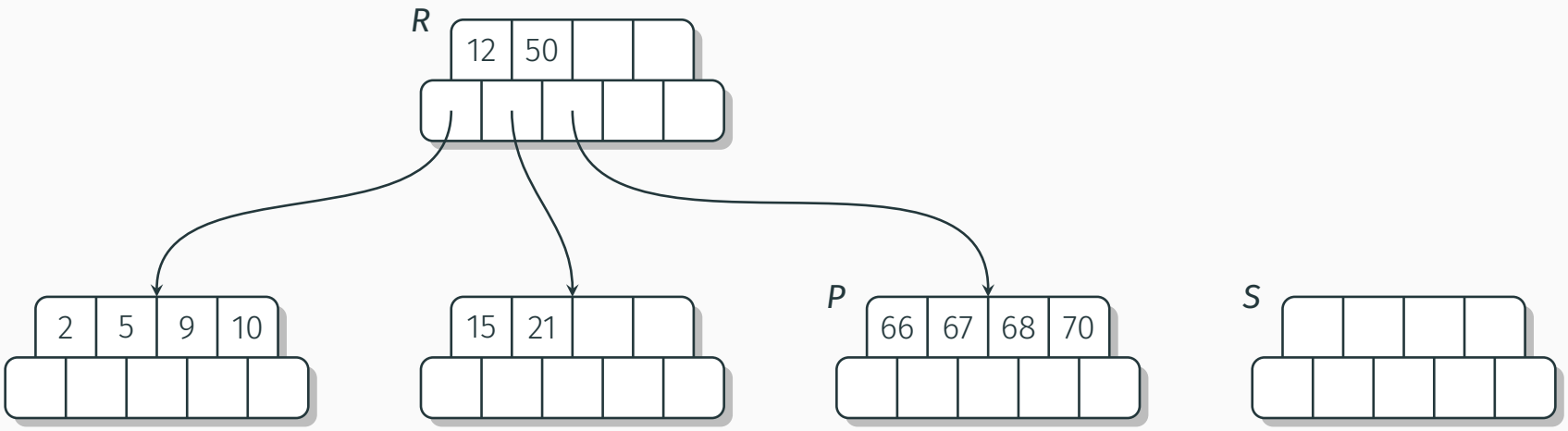
1. Key 45 is located in the internal page ***R***.
2. We replace it with the next larger key, i.e., key 50.

Example of deletion in a B-tree – deletion of key 45 (cont.)

3. This reduces the deletion of key 45 to the deletion of key 50.
4. After deleting key 50, page P contains only $n - 1$ keys.
5. Sibling S contains exactly n keys.
6. We move keys from S to P.
7. We also move the next larger key than 45, i.e., key 57, from parent page R to P, because otherwise one child pointer in R would be left over.

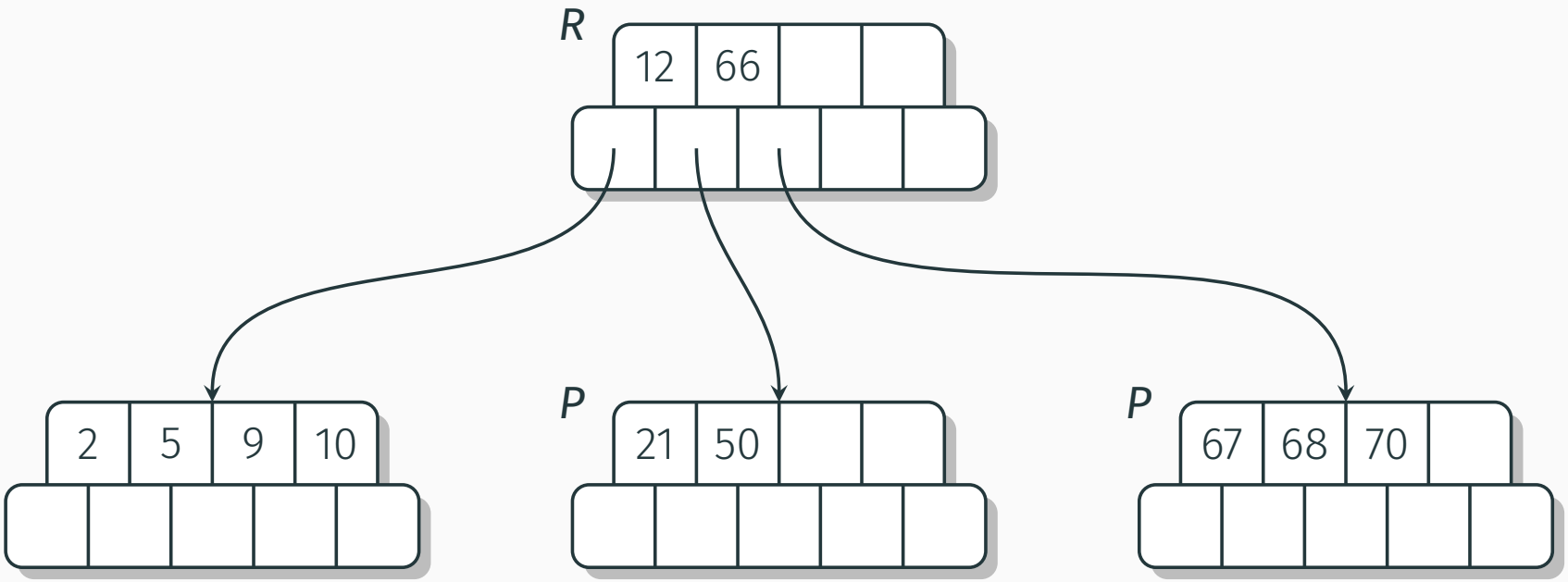
Example of deletion in a B-tree – deletion of key 45 (cont.)

Resulting tree



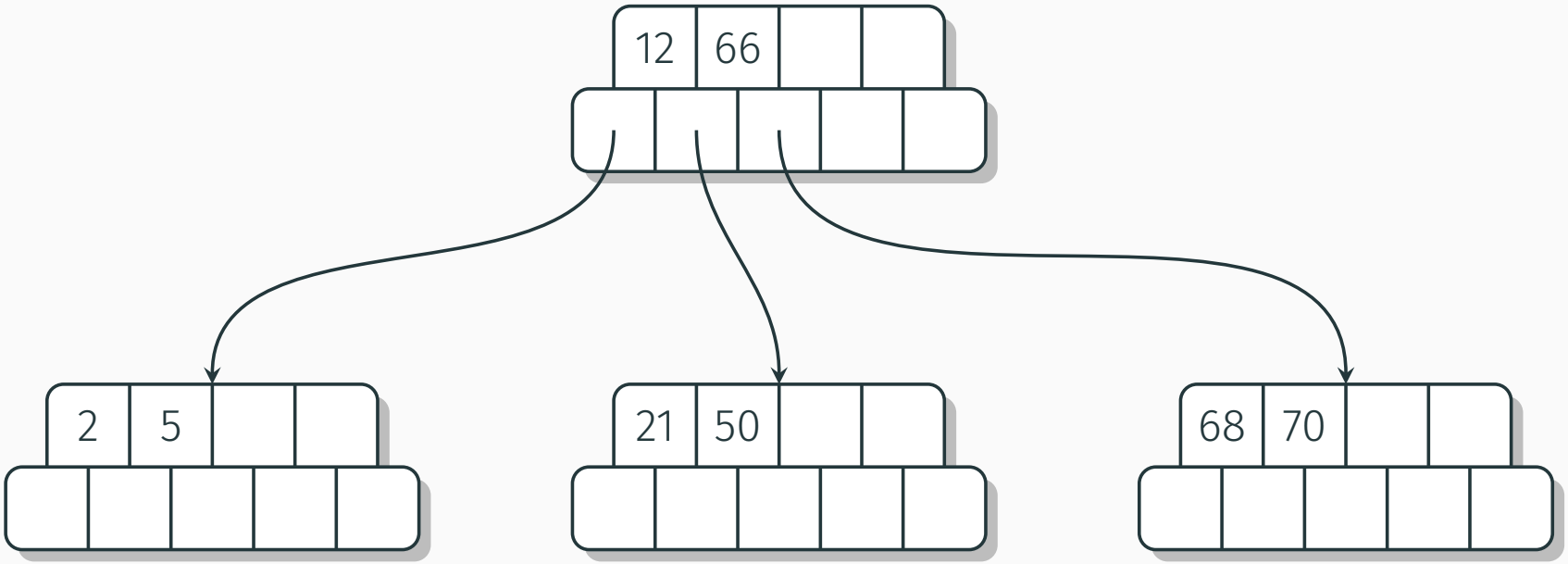
Example of deletion in a B-tree – deletion of key 15

After deleting 15, there remained only $n - 1$ keys in page P. We must therefore move one key from page S through page R.



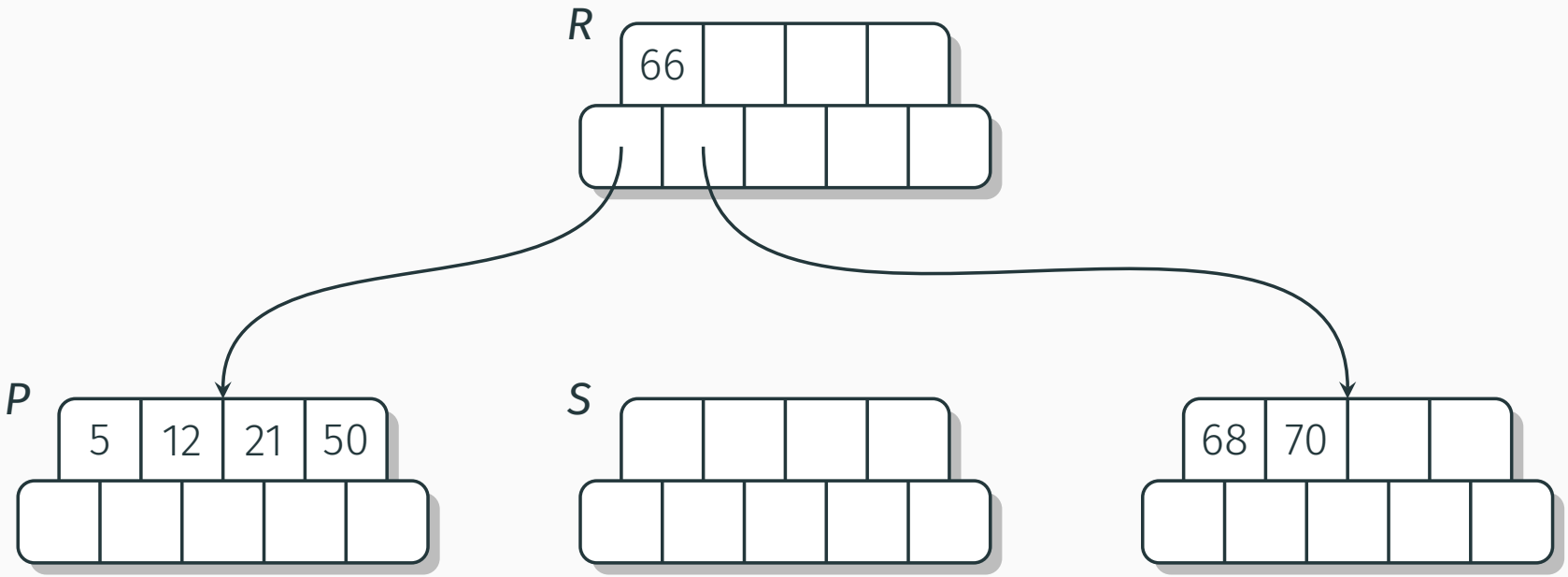
Example of deletion in a B-tree – deletion of keys 9, 10 and 67

Deletion of keys 9, 10 and 67 is very simple.

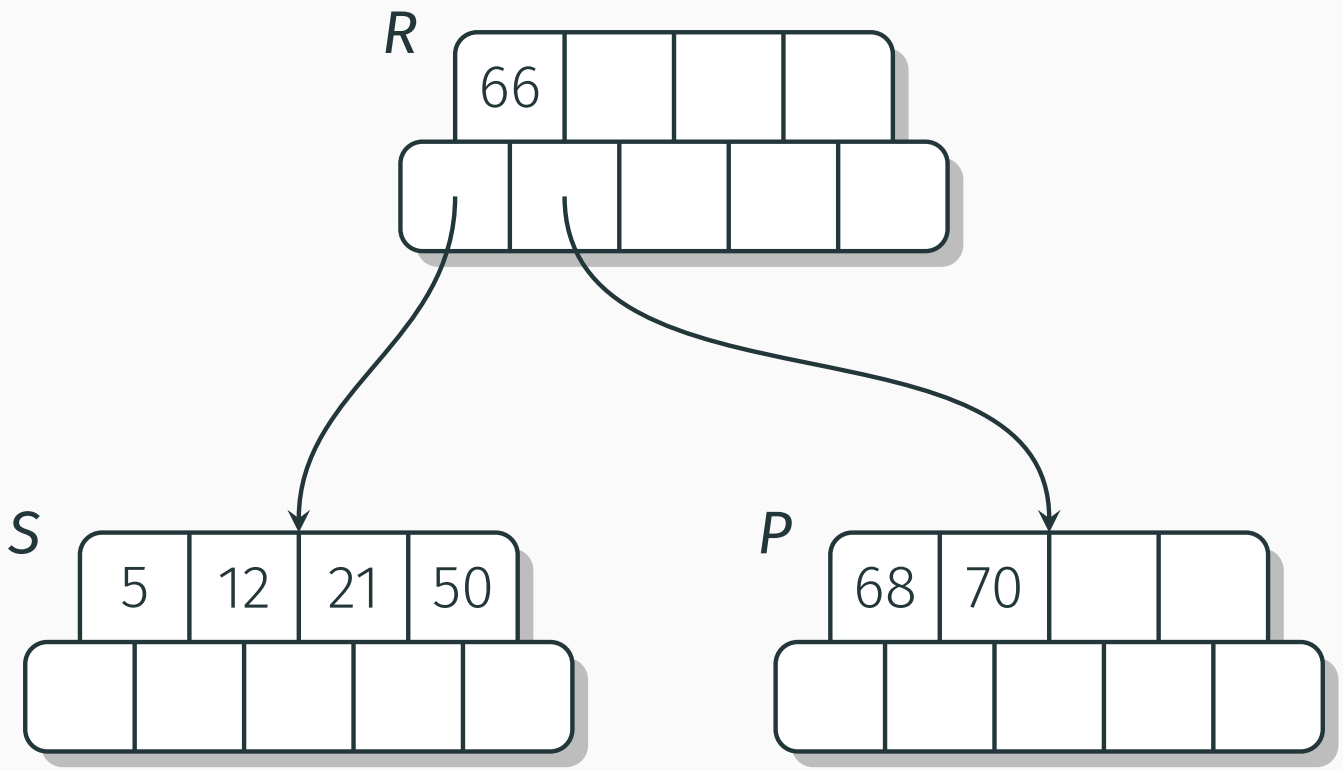


Example of deletion in a B-tree – deletion of key 2

After deleting 2, there are only $n - 1$ keys left in page P. The sibling S contains n keys, so page merging occurs.

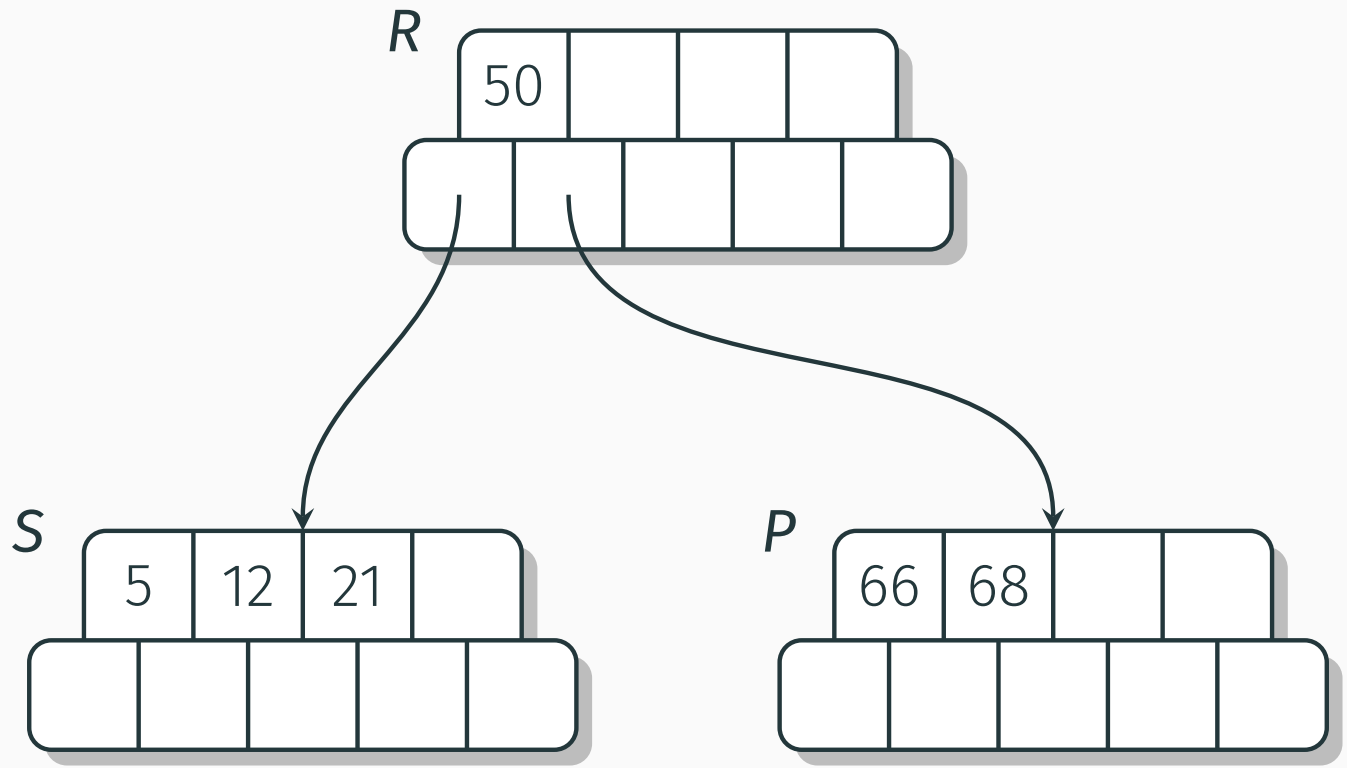


Example of deletion in a B-tree – deletion of key 70



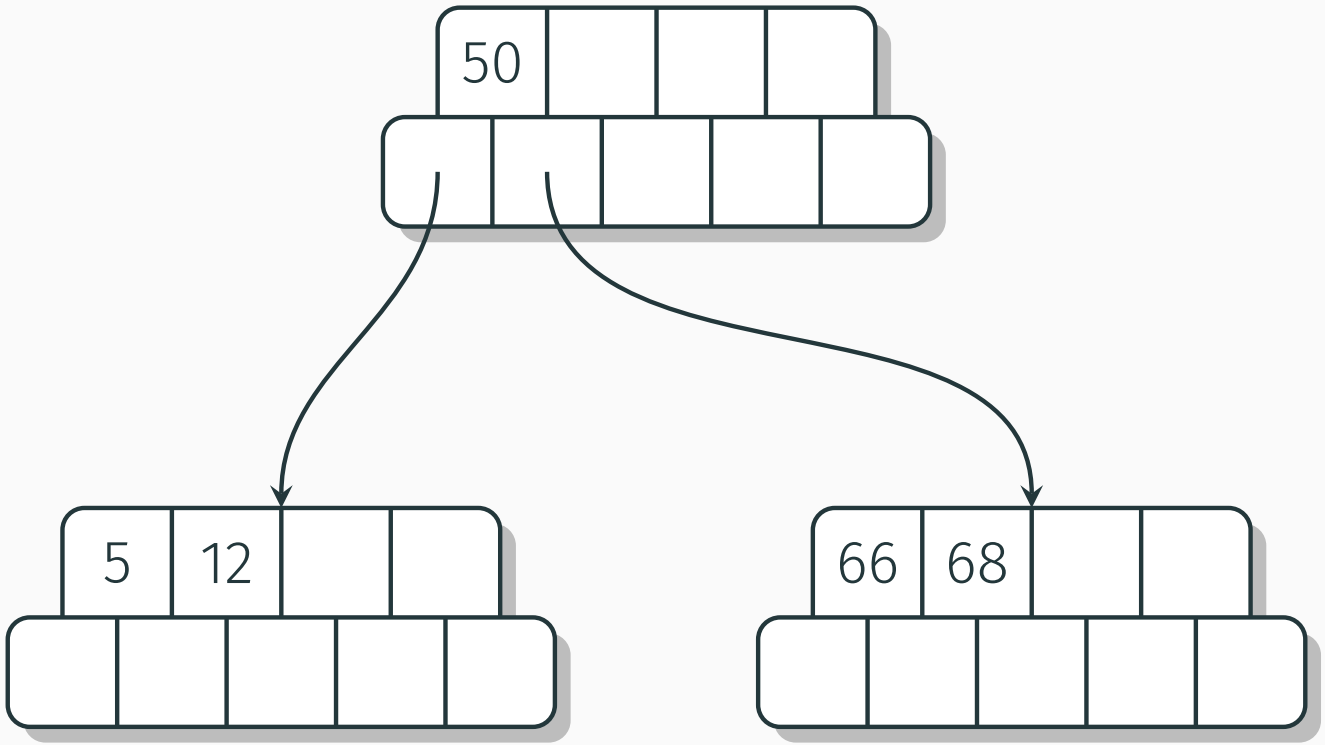
Example of deletion in a B-tree – deletion of key 70 (cont.)

After deleting 70, page P is left with only $n - 1$ keys. Sibling S contains more than n keys, so a shift of 66 from R to P occurs and the nearest smaller key from S to R.



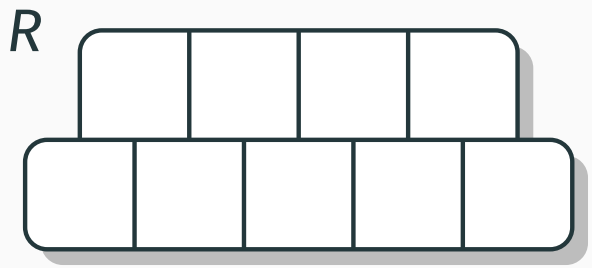
Example of deletion in a B-tree – deletion of key 21

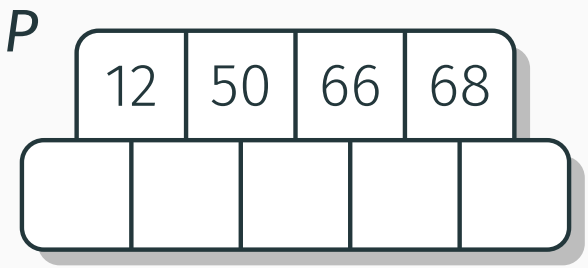
Deletion of key 21 is very simple.

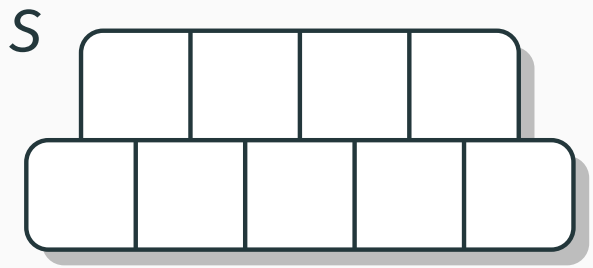


Example of deletion in a B-tree – deletion of key 5

Deletion of key 5 is evident.

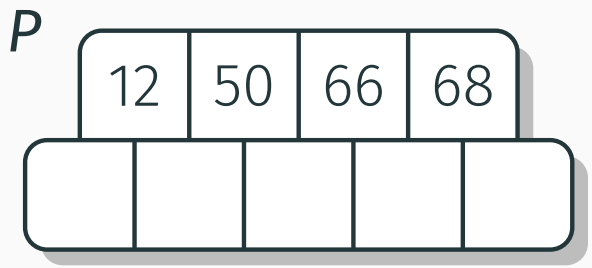






Example of deletion in a B-tree – deletion of key 5 (cont.)

Page P has become the new root, and simultaneously the only page, of the B-tree.



Deletion of keys 12, 50, 66, and 68 is now a trivial matter.

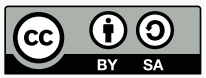
Thanks for your attention

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Dynamic Programming

Jiří Dvorský, Ph.D.

Department of Computer Science
VSB – Technical University of Ostrava



Dynamic Programming

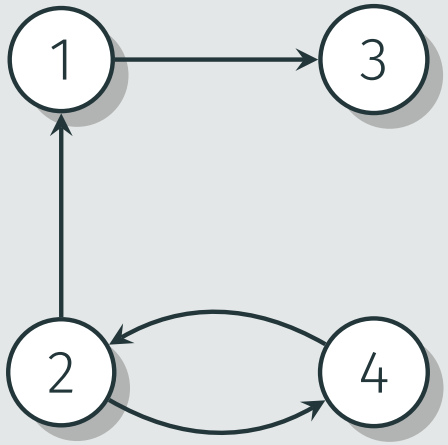
Warshall's algorithm

Warshall's algorithm

$$\mathbf{R}_{i,j}^{(k)} = \begin{cases} \mathbf{R}_{i,j}^{(k-1)} & \text{path through vertices } 1, \ldots, k-1 \\ \vee & \\ \mathbf{R}_{i,k}^{(k-1)} \wedge \mathbf{R}_{k,j}^{(k-1)} & \text{paths from } i \text{ to } k \text{ and from } k \text{ to } j \text{ through vertices } 1, \ldots, k-1 \end{cases}$$

Warshall's algorithm – example

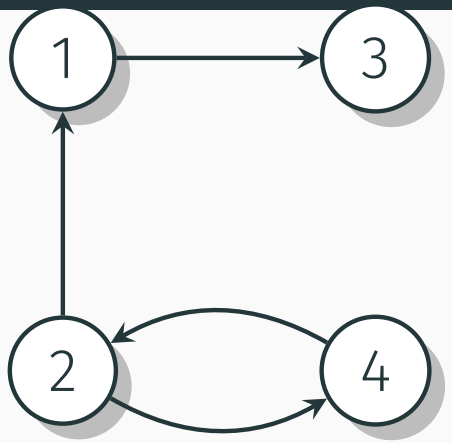
Sample graph G

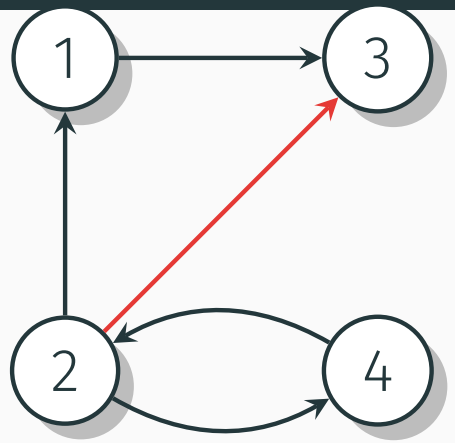


Adjacency matrix

$$\mathbf{A}_G = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Warshall's algorithm – example, $k = 1$



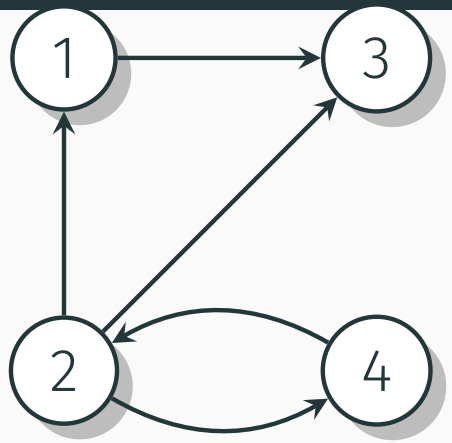


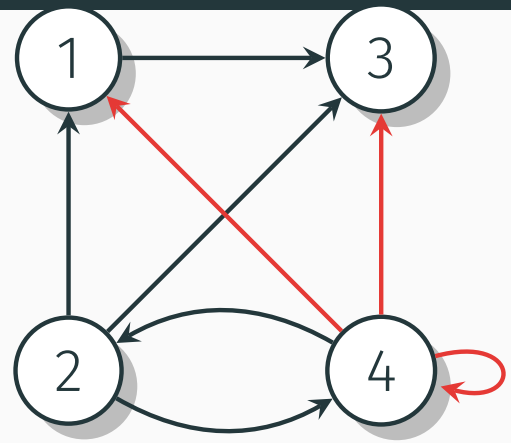
$$\mathbf{R}^{(0)} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

$$\mathbf{R}^{(1)} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Edges for paths leading through vertex 1 have been added to the graph.

Warshall's algorithm – example, $k = 2$



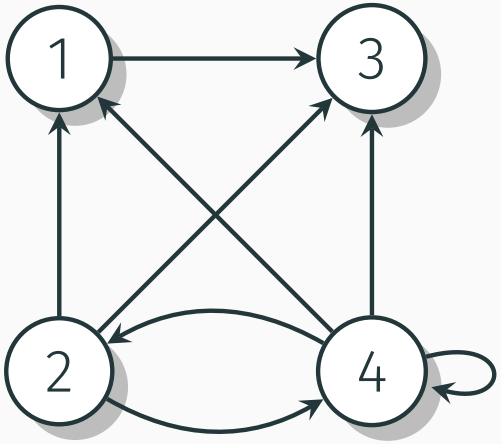


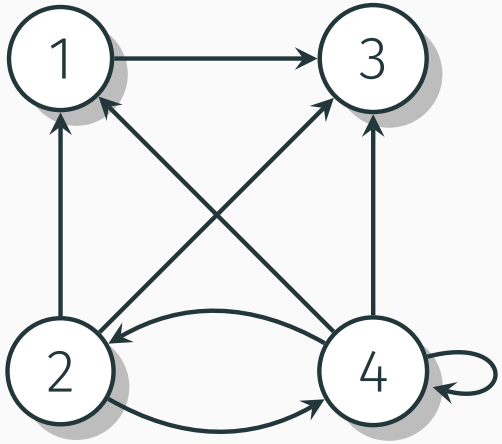
$$\mathbf{R}^{(1)} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

$$\mathbf{R}^{(2)} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Edges for paths leading through vertex 2 have been added to the graph.

Warshall's algorithm – example, $k = 3$

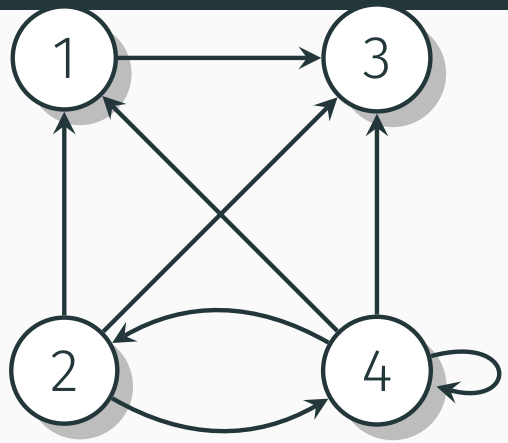


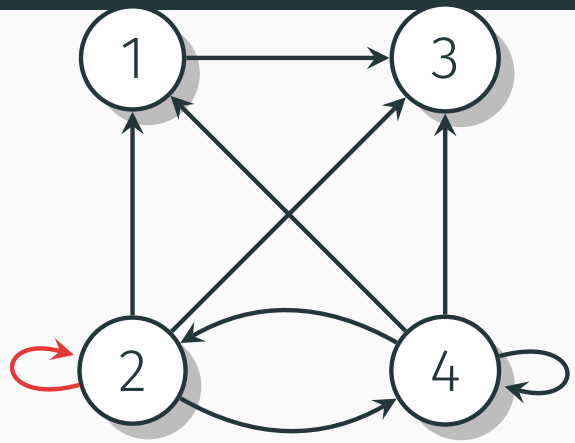


$$\mathbf{R}^{(2)} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \qquad \mathbf{R}^{(3)} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

No edge was added to the graph – **through** vertex 3 no path leads, edges lead only **to** vertex 3.

Warshall's algorithm – example, $k = 4$





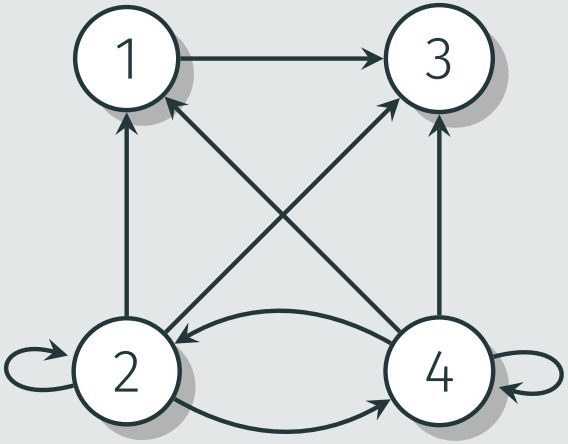
$$\mathbf{R}^{(3)} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

$$\mathbf{R}^{(4)} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Edges for paths leading through vertex 4 have been added to the graph.

Warshall's algorithm – example

Resulting transitive closure T



Adjacency matrix

$$\mathbf{A}_T = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

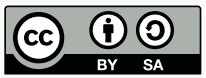
Thanks for your attention

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Greedy Technique

Jiří Dvorský, Ph.D.

Department of Computer Science
VSB – Technical University of Ostrava



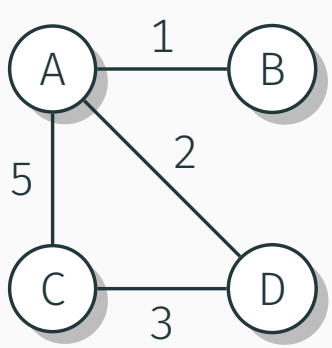
Greedy Technique

Minimum Spanning Tree of a Graph

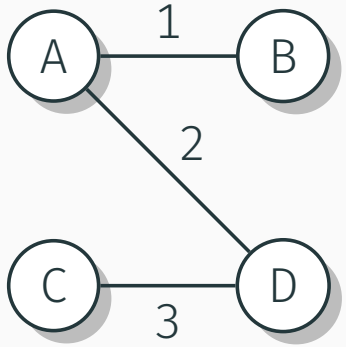
Minimum Spanning Tree of a Graph

Minimum Spanning Tree – Example

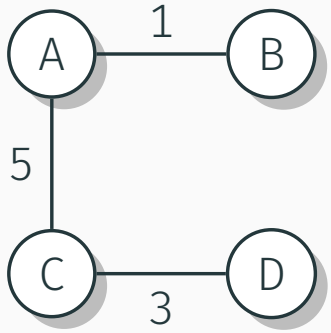
Graph G has a total of 4 spanning trees



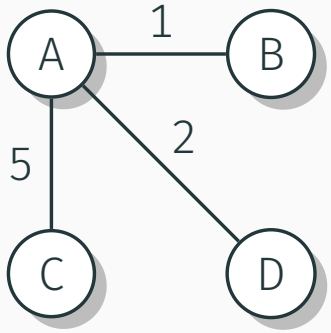
Graph G



$w(K_1) = 6$



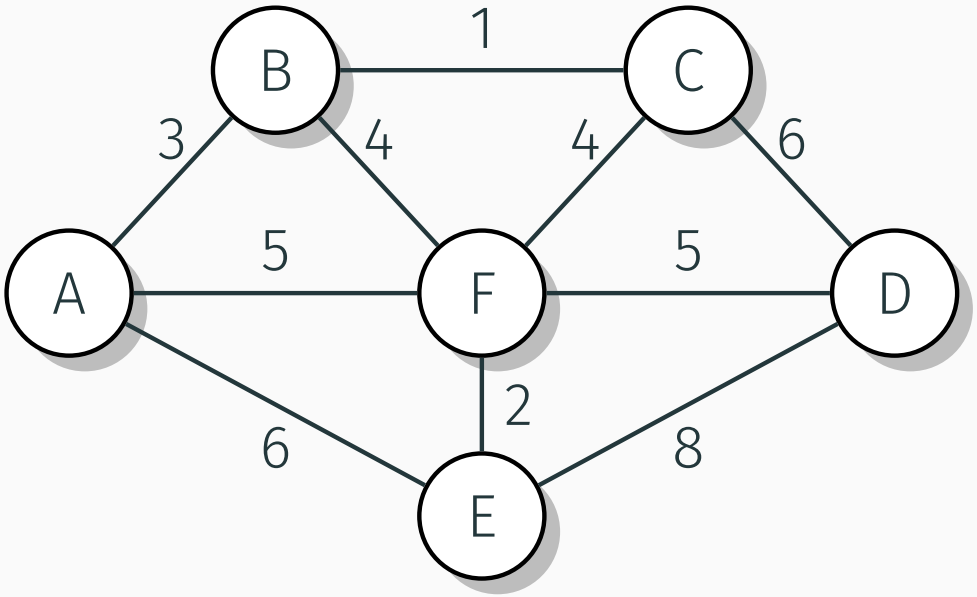
$w(K_2) = 9$



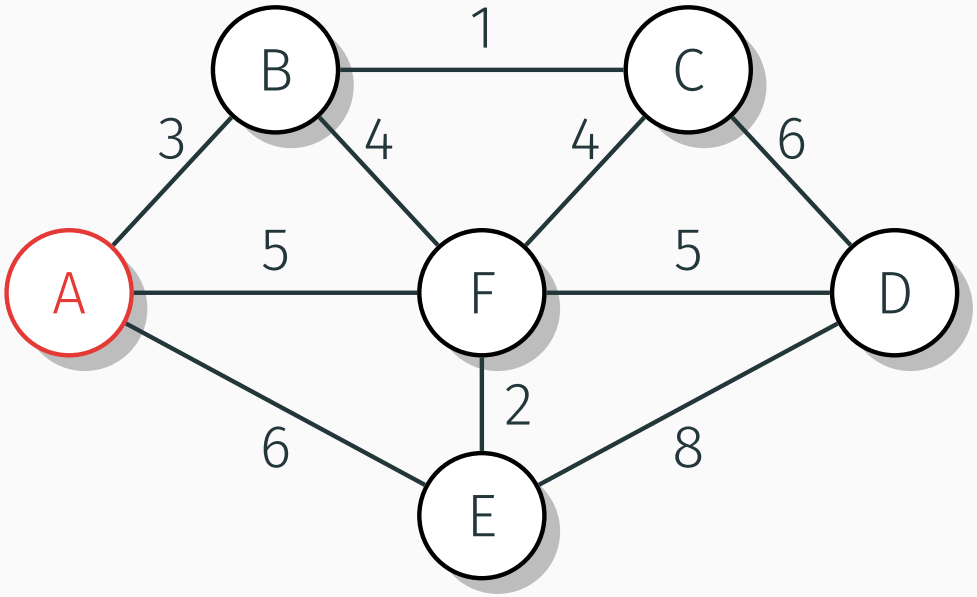
$w(K_3) = 8$

The minimum spanning tree is spanning tree K_1.

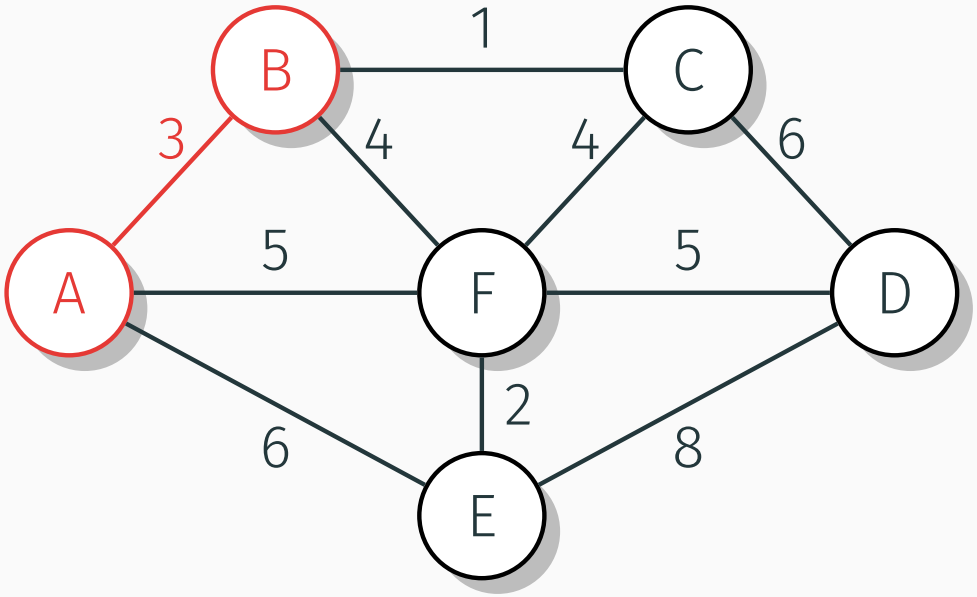
Minimum spanning tree of a graph



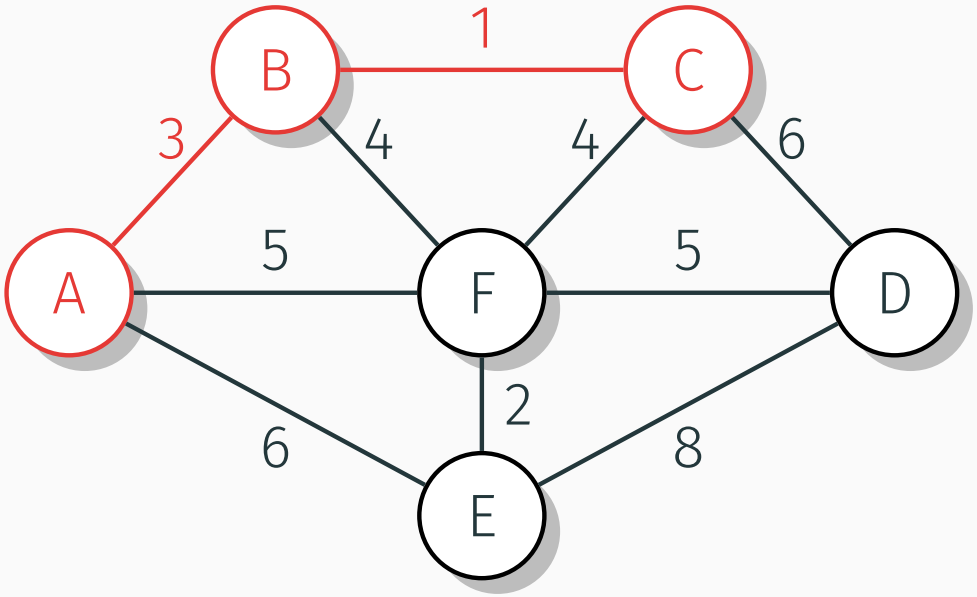
Minimum spanning tree of a graph (cont.)



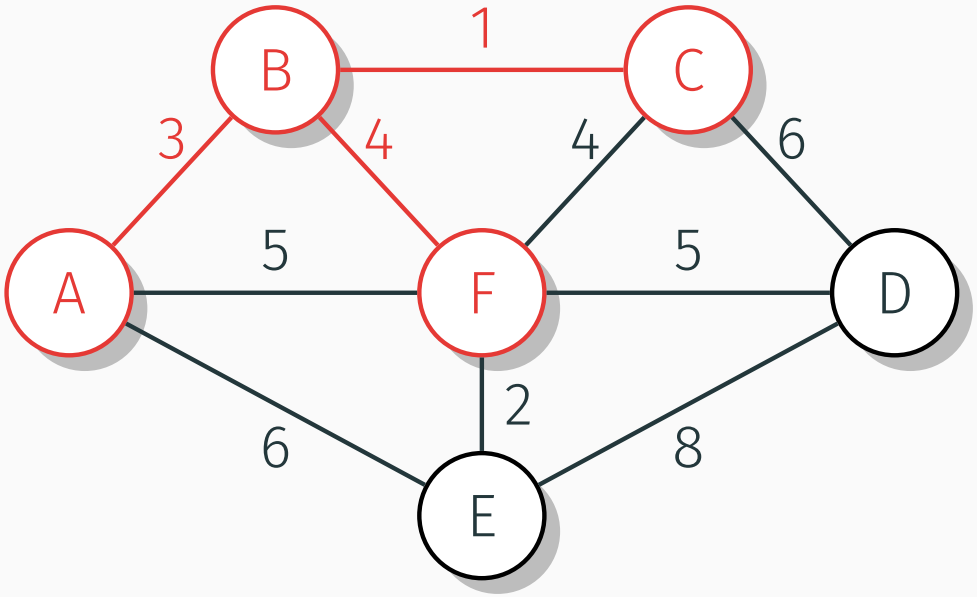
Minimum spanning tree of a graph (cont.)



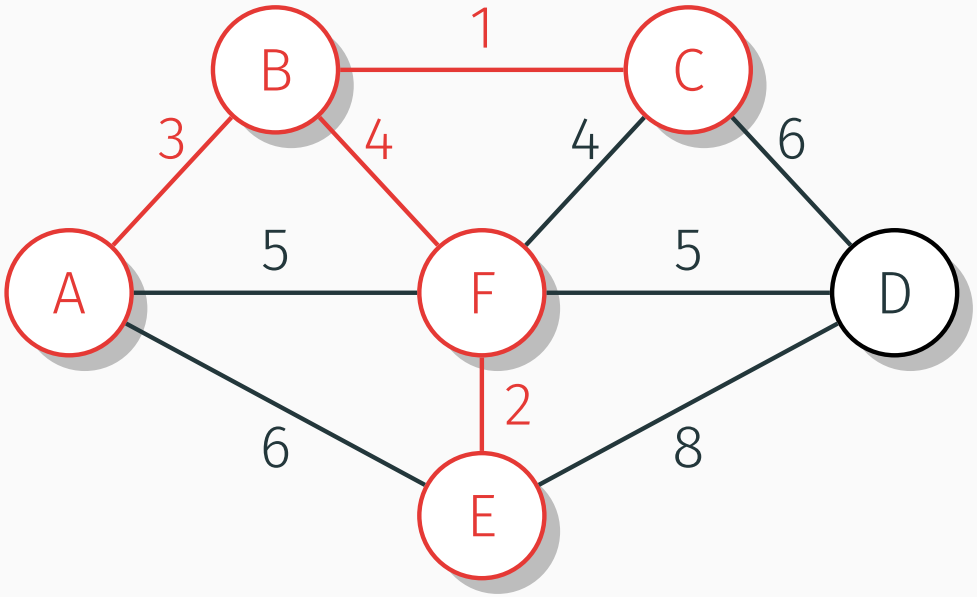
Minimum spanning tree of a graph (cont.)



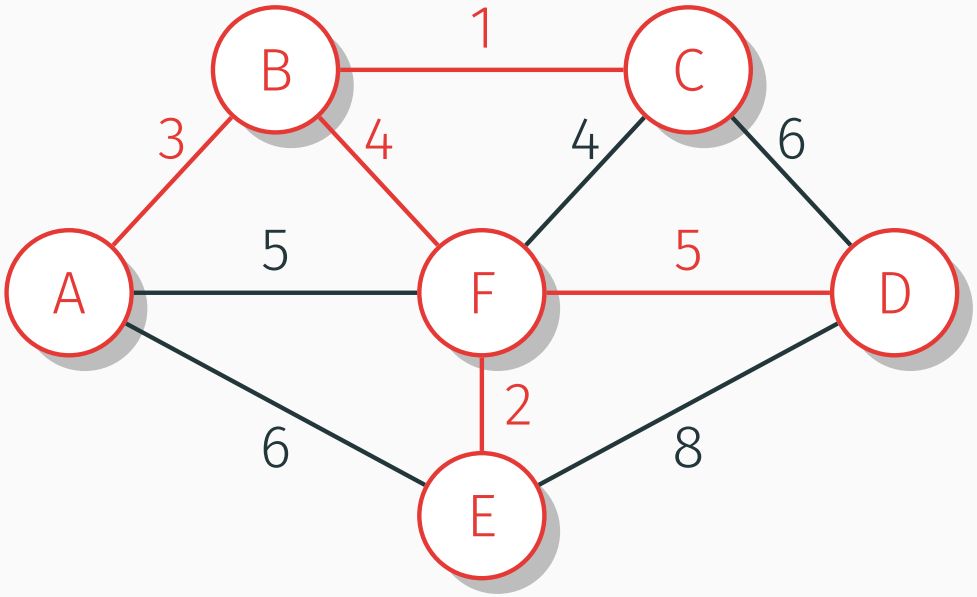
Minimum spanning tree of a graph (cont.)



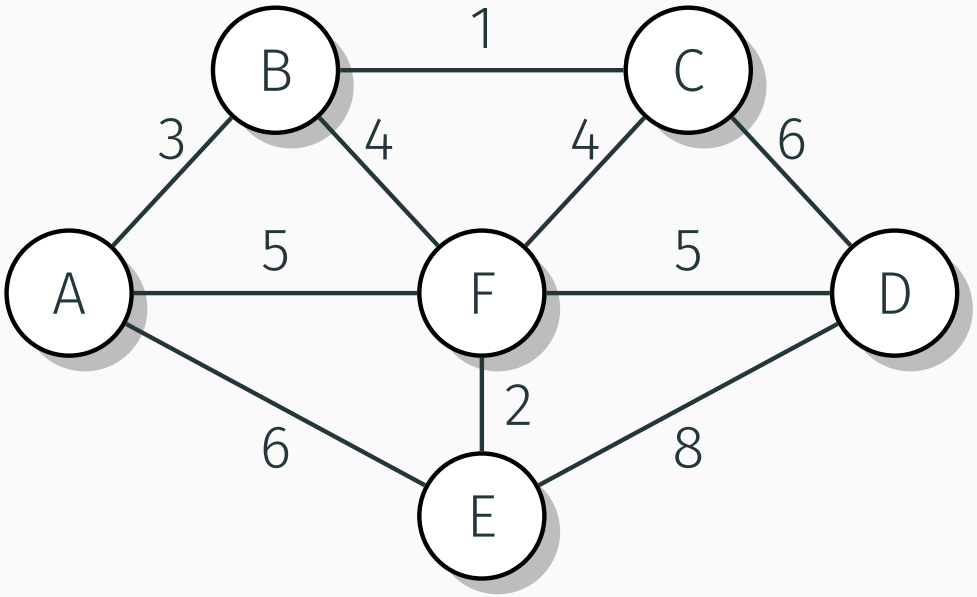
Minimum spanning tree of a graph (cont.)



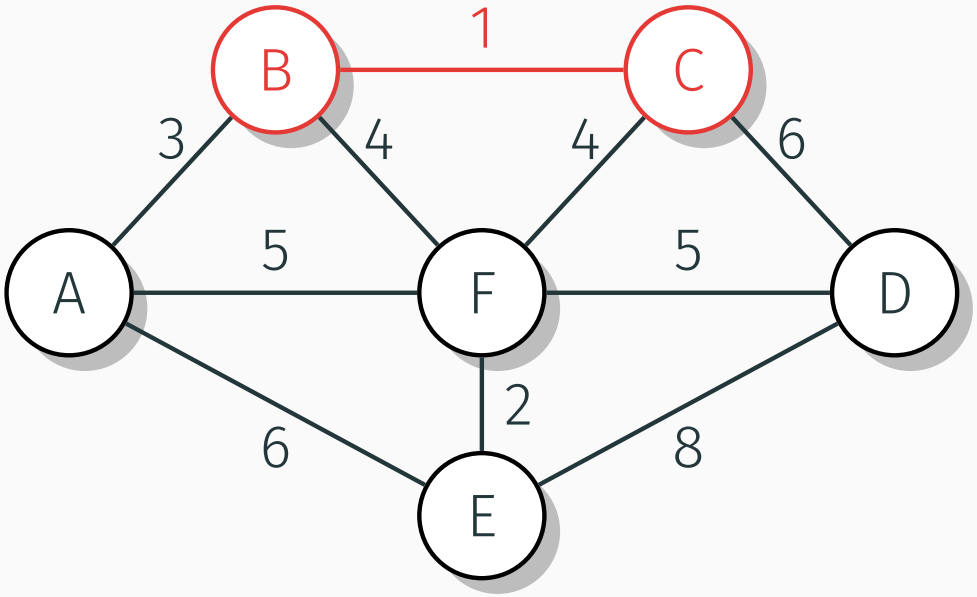
Minimum spanning tree of a graph (cont.)



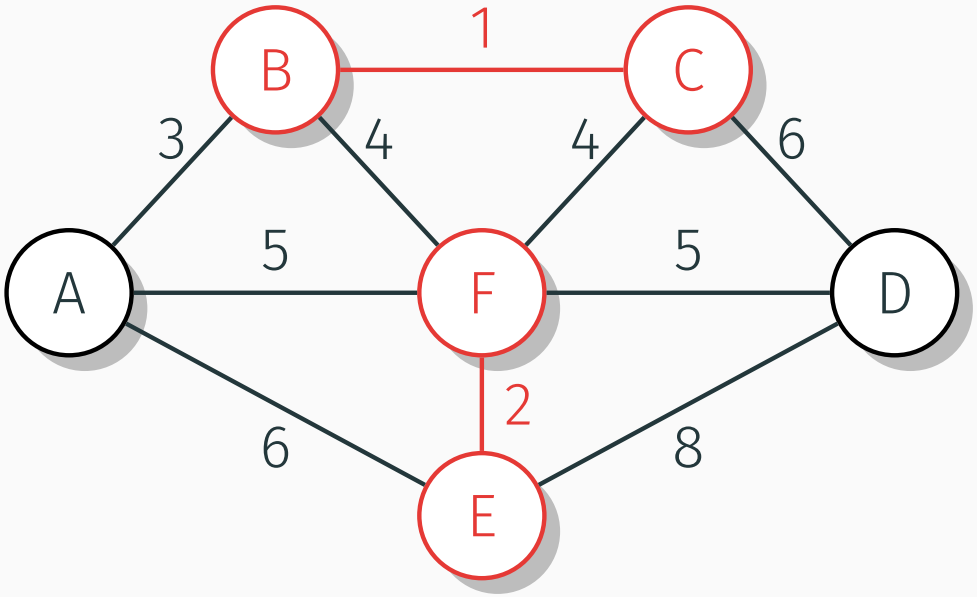
Minimum spanning tree of a graph



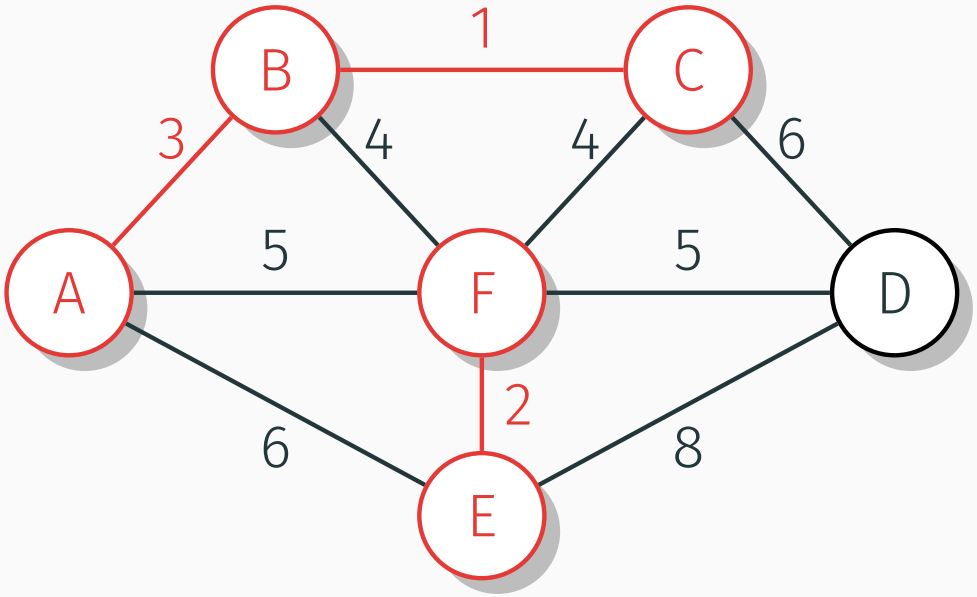
Minimum spanning tree of a graph (cont.)



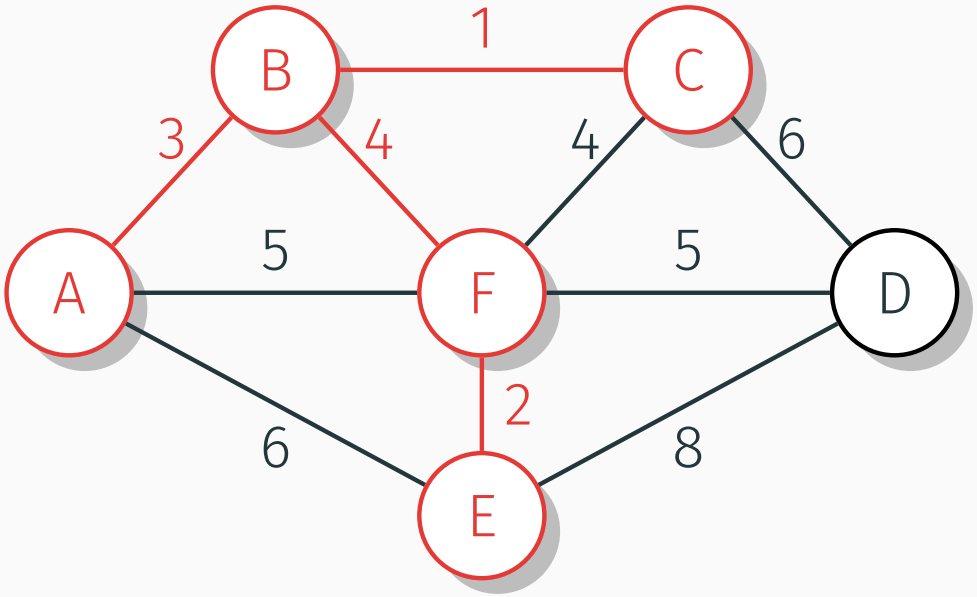
Minimum spanning tree of a graph (cont.)



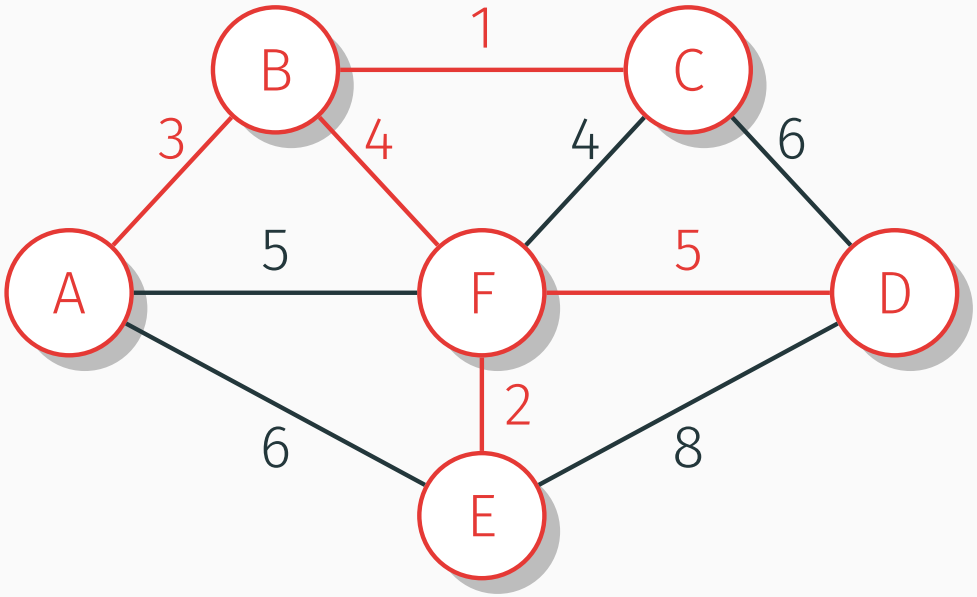
Minimum spanning tree of a graph (cont.)



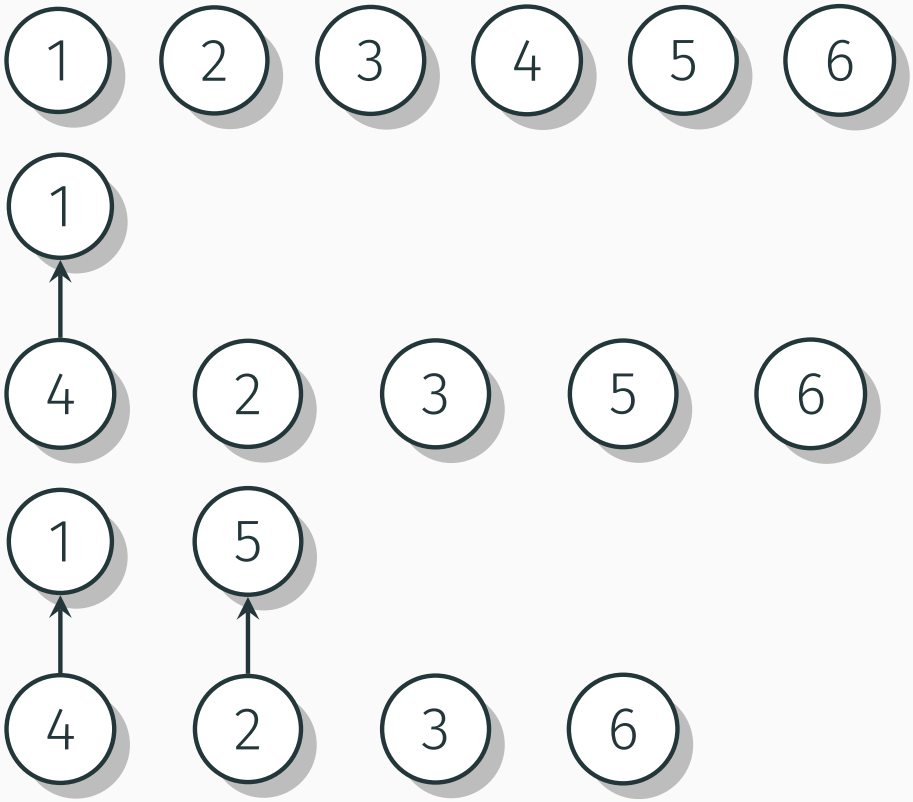
Minimum spanning tree of a graph (cont.)



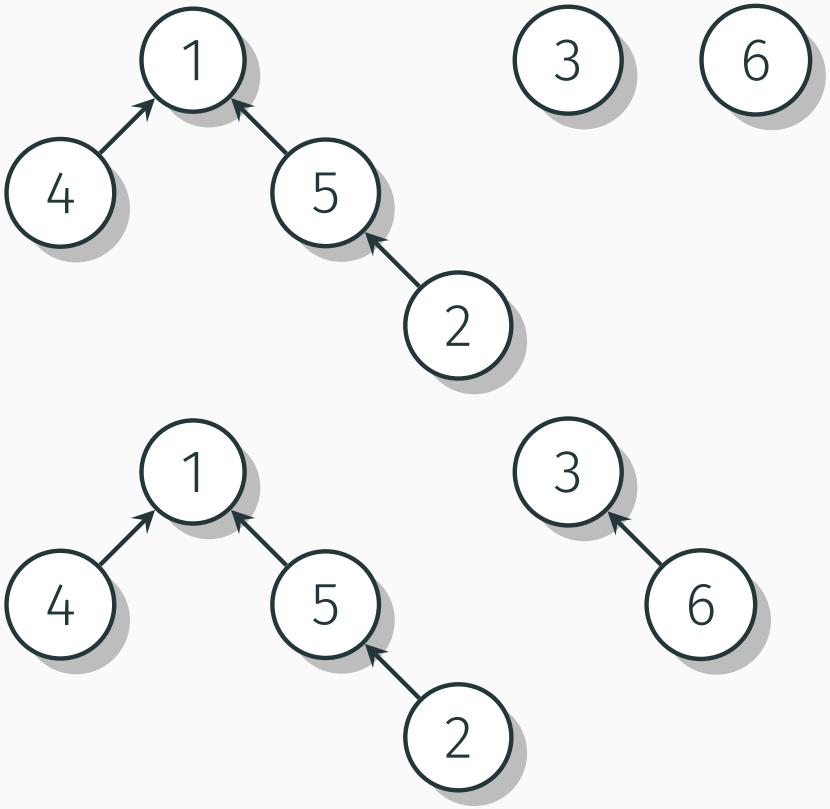
Minimum spanning tree of a graph (cont.)



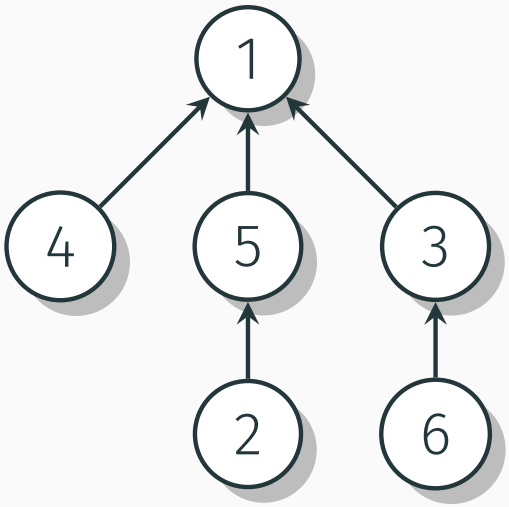
Quick Union



Quick Union (cont.)



Quick Union (cont.)



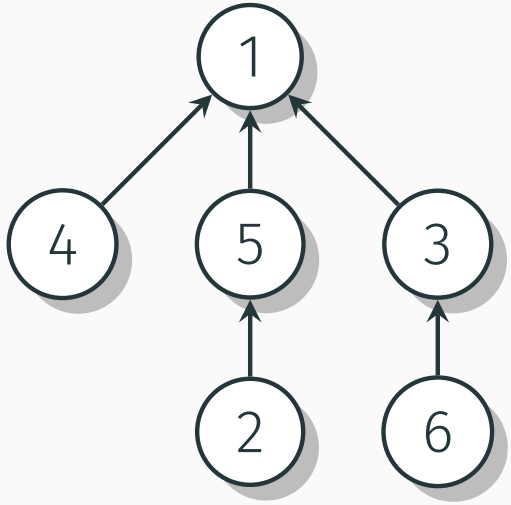
QuickUnion, representation of the tree in an array

Initial state after performing *makeset*(1), ... , *makeset*(6)

Element	Parent
1	null
2	null
3	null
4	null
5	null
6	null

QuickUnion, representation of the tree in an array (cont.)

Final state after performing all *union* operations



Element	Parent
1	null
2	5
3	1
4	1
5	1
6	3

Greedy Technique

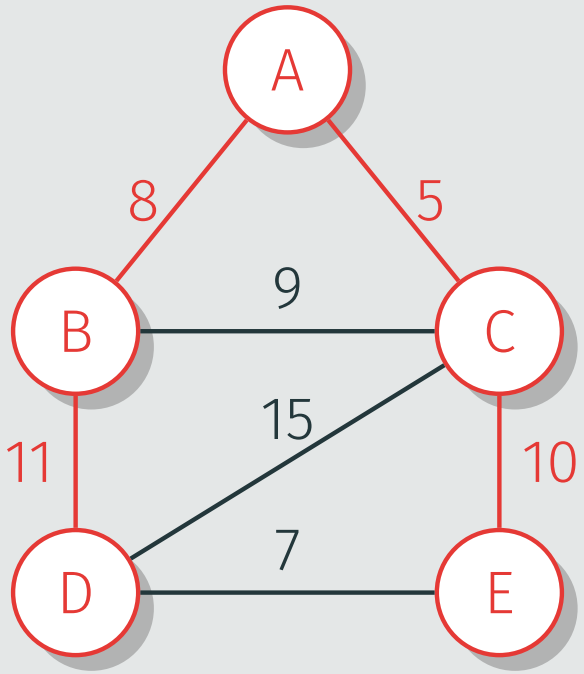
Dijkstra's algorithm

Shortest path tree vs. minimum spanning tree of the graph

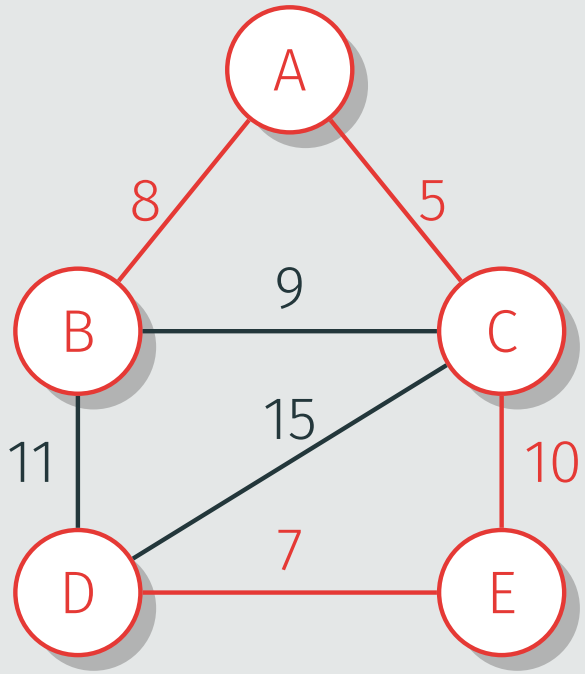
- The result of Dijkstra's algorithm is not the minimum spanning tree of the graph.
- The result is **the shortest path tree** from a given initial vertex.
- The shortest path tree is just one of the graph's spans, but it does not have to be minimal.
- The minimum spanning tree of the graph minimizes **the sum of edge weights** in the span.
- The shortest path tree minimizes **path length** from a given initial vertex. The shape of the tree **depends** on the initial vertex.

Shortest paths tree vs. minimum spanning tree of the graph, example

Shortest paths tree from A



Minimum spanning tree of the graph



Greedy Technique

Huffman code

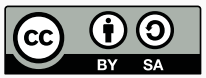
Thanks for your attention

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Iterative Improvement

Jiří Dvorský, Ph.D.

Department of Computer Science
VSB – Technical University of Ostrava



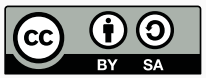
Thanks for your attention

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Limitations of Algorithm Power

Jiří Dvorský, Ph.D.

Department of Computer Science
VSB – Technical University of Ostrava



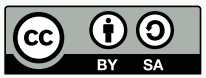
Thanks for your attention

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Coping with Limitations of Algorithm Power

Jiří Dvorský, Ph.D.

Department of Computer Science
VSB – Technical University of Ostrava



Thanks for your attention

Bibliography

1. **Železniční mapy ČR** [online]. Praha: SŽDC, 2019 [visited on 2019-11-15]. Available from: *http://provoz.szdc.cz/portal/Show.aspx?path=/Data/Mapy/kjr.pdf*.
2. LEVITIN, Anany. **Introduction to the Design and Analysis of Algorithms**. 3rd ed. Boston: Pearson, 2012. ISBN 978-0-13-231681-1.
3. CORMEN, Thomas H.; LEISERSON, Charles Eric; RIVEST, Ronald L.; STEIN, Clifford. **Introduction to algorithms**. Fourth edition. Cambridge, Massachusetts: The MIT Press, 2022. ISBN 978-026-2046-305.

Bibliography (cont.)

4. WRÓBLEWSKI, Piotr. **Algoritmy**. 1. vyd. Brno: Computer Press, 2015. ISBN 978-80-251-4126-7.
5. WIRTH, Niklaus. **Algoritmy a štruktúry údajov**. 1st ed. Bratislava: Alfa, 1988. ISBN 063-030-87.
6. BAYER, Rudolf; MCCREIGHT, Edward Meyers. Organization and maintenance of large ordered indexes. **Acta Informatica**. 1972, vol. 1, no. 3, pp. 173–189. ISSN 1432-0525. Available from DOI: *10.1007/BF00288683*.

VSB TECHNICAL UNIVERSITY OF OSTRAVA | FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE | DEPARTMENT OF COMPUTER SCIENCE

Appendices

Jiří Dvorský, Ph.D.

Department of Computer Science
VSB – Technical University of Ostrava