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

Introduction

Jiří Dvorský, Ph.D.

Presentation status to date February 24, 2025

Department of Computer Science
VSB – Technical University of Ostrava

Lecture outline

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.

$$\begin{tikzcd}
 & \text{Problem} \arrow[d] & \\
 & \text{Algorithm} \arrow[d] & \\
\text{Input} \arrow[r] & \boxed{\text{Computer}} \arrow[r] & \text{Output}
\end{tikzcd}$$

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$(m, n)$
while $n \neq 0$ do
    $r \leftarrow m$ mod $n$;
    $m \leftarrow n$;
    $n \leftarrow r$;
end
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 $2p, 3p, \dots, (p-1)p$ 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 $\leq n$

```
for $p \leftarrow 2$ to $n$ do
    $A[p] \leftarrow p$
end
for $p \leftarrow 2$ to $\lfloor\sqrt{n}\rfloor$ do
    if $A[p] \neq 0$ then          // $p$ has not been excluded yet
        $j \leftarrow p^2$;
        while $j \leq n$ do
            $A[j] \leftarrow 0$;
            $j \leftarrow j + p$;
        end
    end
end
```

The Sieve of Eratosthenes (cont.)

```
// Numbers that were not excluded from array $A$ are
    copied to array $L$
$i \leftarrow 0$;
for $p \leftarrow 2$ to $n$ do
    if $A[p] \neq 0$ then
        $L[i] \leftarrow A[p]$;
        $i \leftarrow i + 1$;
    end
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…

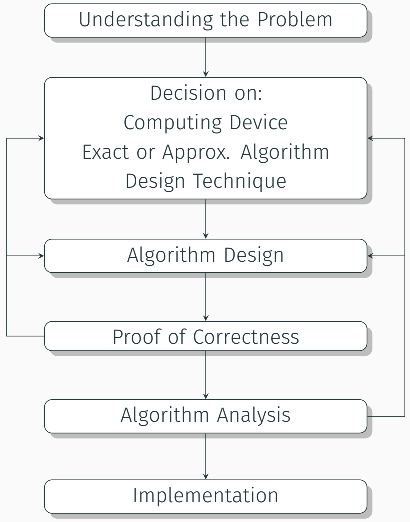
Introduction

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 $\Rightarrow$ 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.

Techniques of Algorithm Design (cont.)

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.)

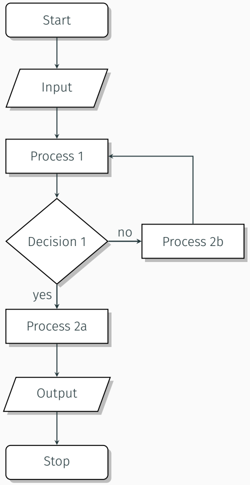
Programming Language

- 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

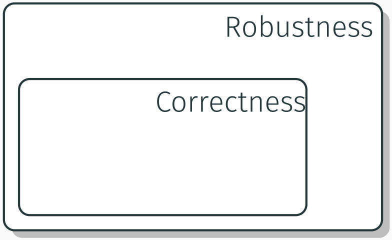
Algorithm Coding

- 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

$$\begin{tikzcd}
& \{a,b\} & \\
\{a\} \arrow[ur, no head] & & \{b\} \arrow[ul, no head] \\
& \emptyset \arrow[ul, no head] \arrow[ur, no head] \arrow[uu, dashed, no head] &
\end{tikzcd}$$

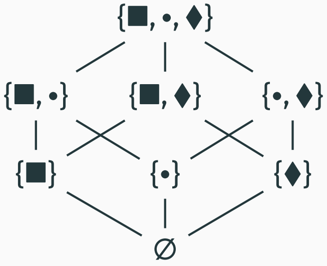
Hasse diagram for the ordering relation “to be a subset” on the set $\{a,b\}$. A transitive edge, which is normally not shown, is displayed dashed.

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 $A = a_1, a_2, \dots a_n$. 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

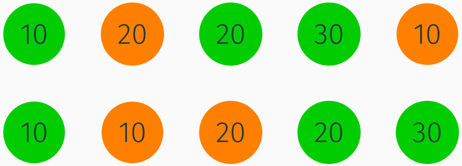
$$a_{\pi_1} < a_{\pi_2} < a_{\pi_3} < a_{\pi_4} < a_{\pi_5} < a_{\pi_6}$$
$$a_6 < a_2 < a_4 < a_5 < a_1 < a_3$$
$$a < b < c < d < e < f$$

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 $1 \dots n$.
- 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

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

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

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

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

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”

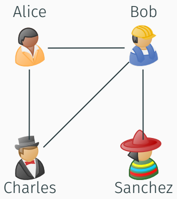
- symmetrical algorithms – the pattern is traversed in the same direction
- asymmetrical algorithms – the pattern is traversed in the opposite direction.

Searching in Text – Notation

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

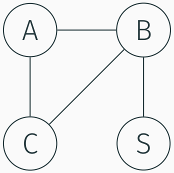
- p the searched pattern, $p = p_0 p_1 \dots p_{m-1}$, where $|p| = m$ is the length of the pattern,
- t the searched text, $t = t_0 t_1 \dots t_{n-1}$, where $|t| = n$ is the length of the text,
- Σ – the alphabet from which the pattern and the text are composed,
- σ – the size of the alphabet Σ ($\sigma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

Graph Problems



Ostravsko
Racibórz, Rybnik
Bohumín
Dětmarovice
Petrovice u Karviné
Karviná hl.n.
Krnov Opava
Ostrava hl.n.
Ostrava-Stodolní
Ostrava střed
Ostrava-Svinov
Polanka nad Odrou
Ostrava-Kunčice
Havířov
O.-Vítkovice
O.-Bartovice
Albrechtice u Č.Těšína
Chotěbuz
Katowice Warszawa Kraków
Studénka, Hranice na Mor. Mošnov, Ostrava Airport
Frýdek-Místek Valašské Meziříčí
Český Těšín Třinec, Žilina
270
320
321
323
326

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 $0, \dots, n - 1$

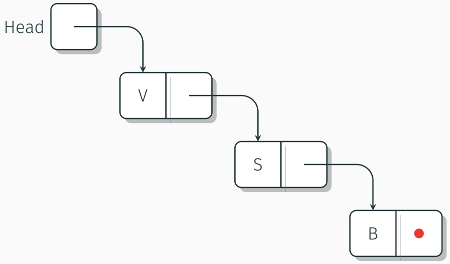
`a[0]`	`a[1]`	...	`a[n-1]`

- 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.)

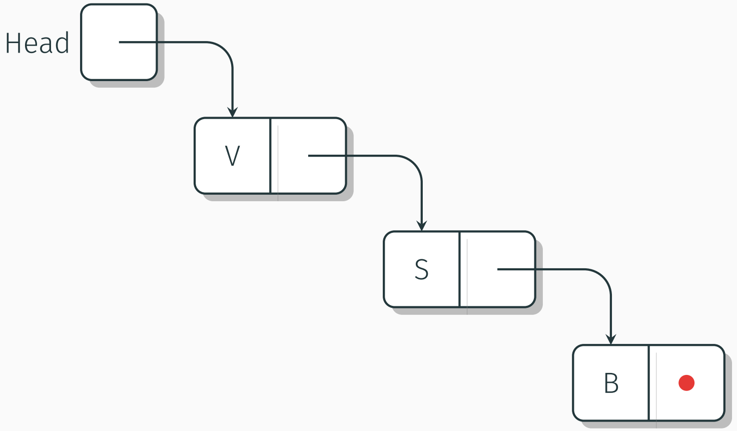
List Variants

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

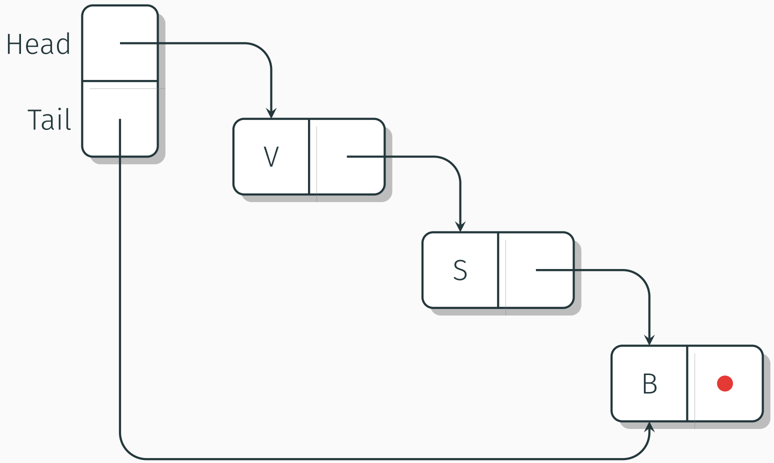
Singly Linked List

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

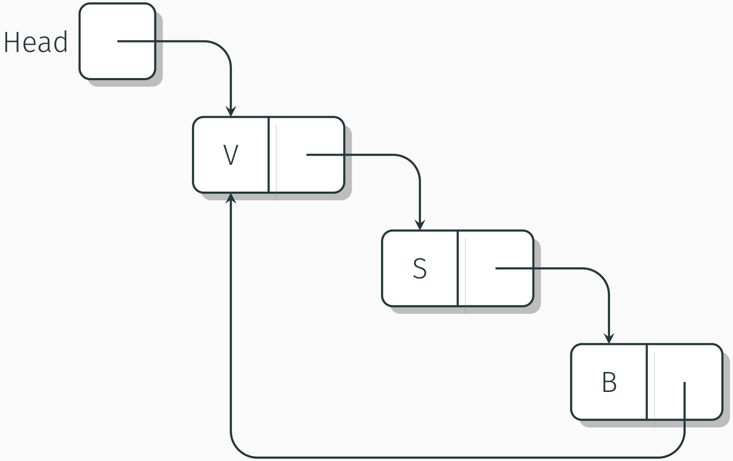
Singly Linked List (Head Only)



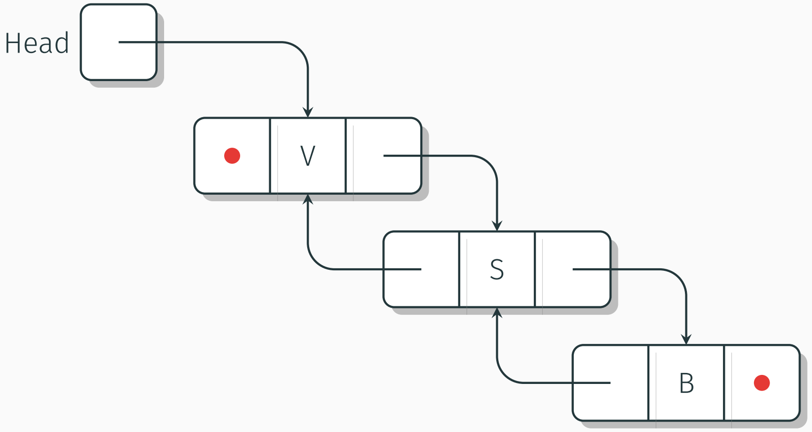
Singly Linked List (Head & Tail)



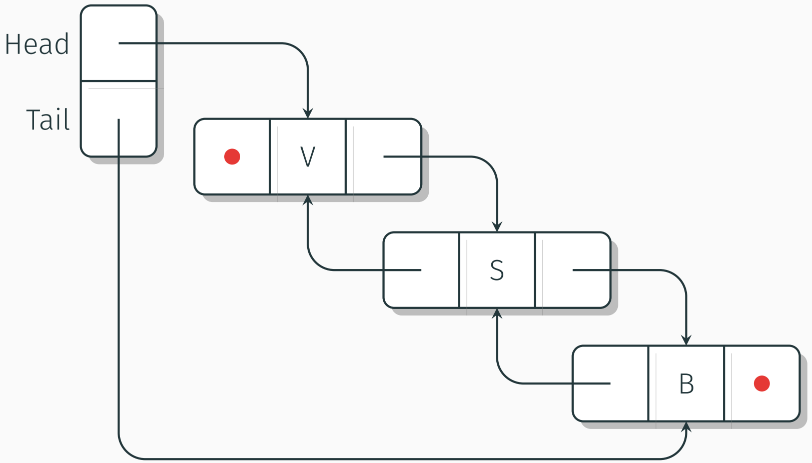
Circular Singly Linked List



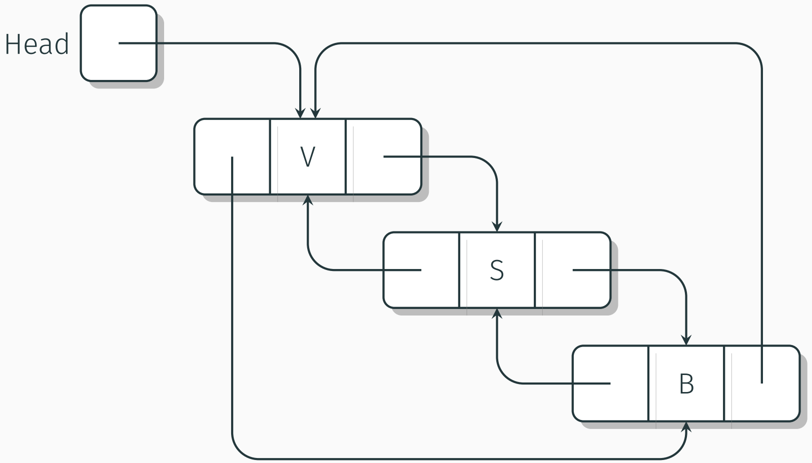
Doubly Linked List (Head Only)



Doubly Linked List (Head & Tail)



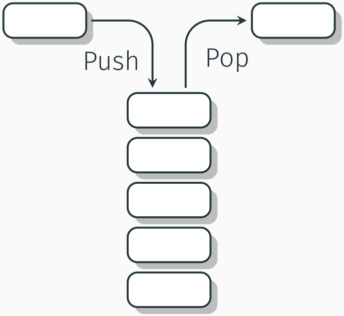
Circular Doubly Linked List



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

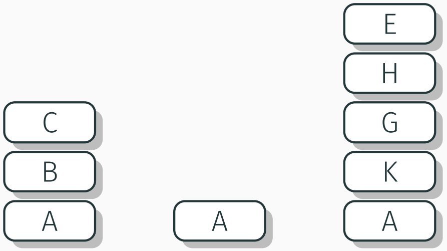
- **Push** – add element to top
- **Pop** – remove top element
- **IsEmpty** – check emptiness
- **Top** – peek top element

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

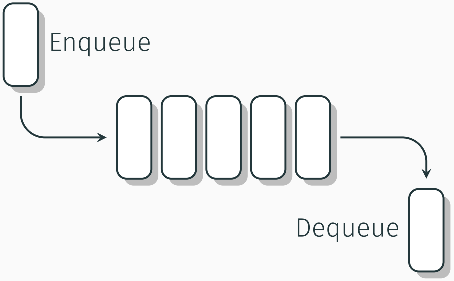
Stack Applications

- 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

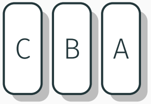
- **Enqueue** – add to tail
- **Dequeue** – remove from head
- **Peek** – inspect head element
- **IsEmpty** – check emptiness

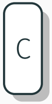
Additional Operations

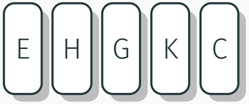
- **Init** – initialize
- **Clear** – empty queue
- IsFull – check capacity (limited capacity queues)

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

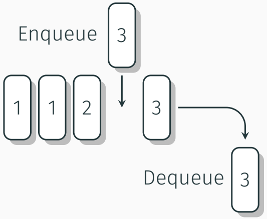
Queue Applications

- 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.



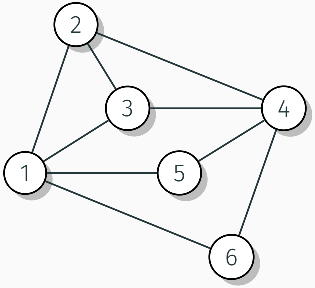
Implementations

- 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

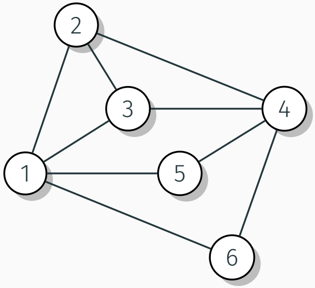
An edge that connects a vertex to itself is called a **loop**.

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.

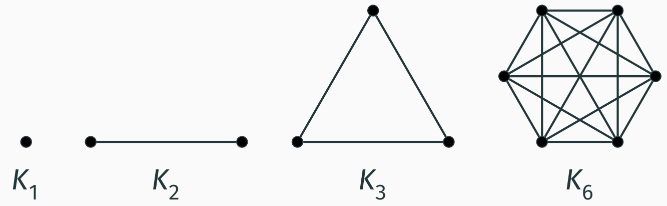
The maximum number of edges in a graph is achieved by connecting each of the $|V|$ vertices with all other vertices, which are $|V| - 1$. The product $|V|(|V| - 1)$ must be divided by two because each edge is counted twice. □

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.

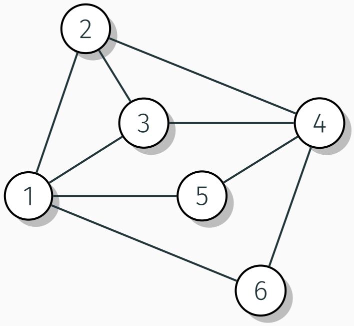
Subgraph (cont.)

Remarks

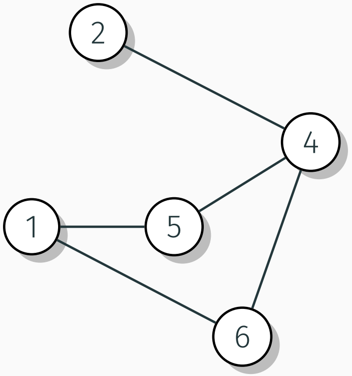
- In other words, a subgraph is obtained by deleting some vertices of the original graph, all edges incident to these vertices, and possibly some additional edges.
- The term subgraph is used in graph theory as a kind of analogy to the concept of a subset.

Subgraph (cont.)

Example



Graph ***G***

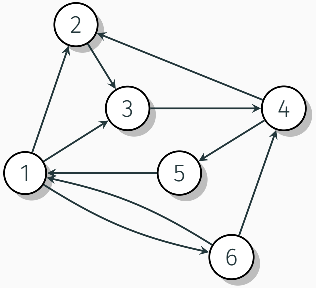


Subgraph ***H***

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

$G = (V, E)$

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

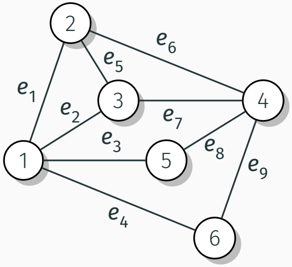
$E = \{(1, 2), (1, 3), (1, 6), (2, 3), (3, 4), (4, 2), (4, 5), (5, 1),$
$\{(6, 1), (6, 4)\}$

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

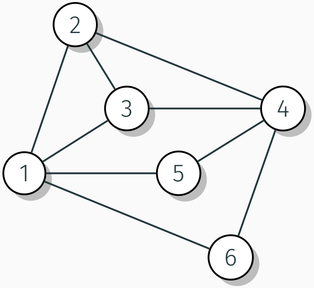
- The number of rows in the matrix corresponds to the number of vertices, and the number of columns corresponds to the number of edges.
- If a vertex is incident with an edge, there is a 1 at the given position; otherwise, there is a 0.



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

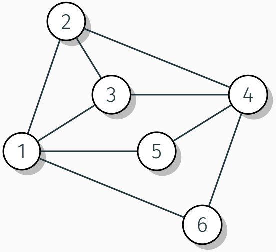
Adjacency Matrix

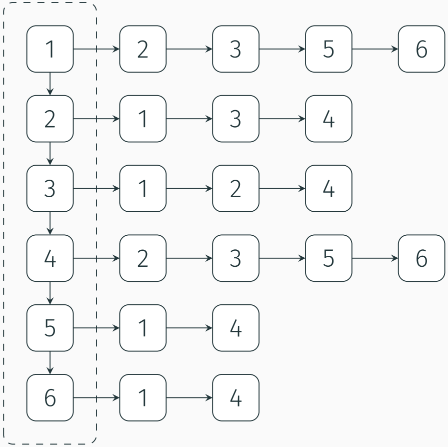
- Square matrix where the number of rows and columns corresponds to the number of vertices.
- Contains **1** if vertices are adjacent, **0** otherwise.



$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 2 & 1 & 0 & 1 & 1 & 0 & 0 \\ 3 & 1 & 1 & 0 & 1 & 0 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 & 1 \\ 5 & 1 & 0 & 0 & 1 & 0 & 0 \\ 6 & 1 & 0 & 0 & 1 & 0 & 0 \end{array}$$

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

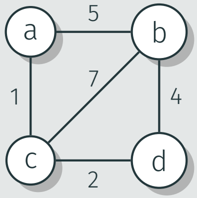
- Suitable for dense graphs.
- Vertex insertion/deletion is complex, while edge operation are easy.

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



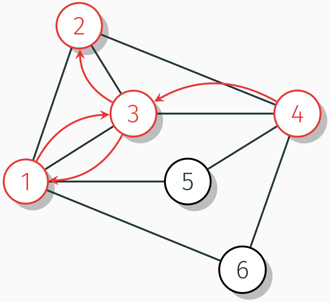
Adjacency Matrix

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

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

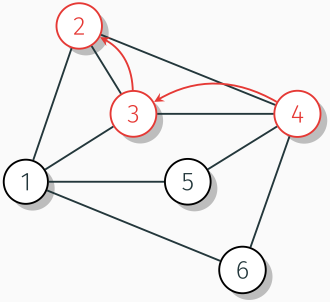
4 {4, 3} 3 {3, 1} 1 {1, 3} 3 {3, 2} 2

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

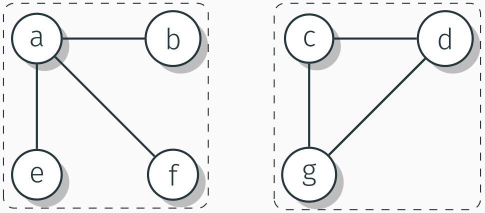
A graph is called **connected** if there exists a path between every pair of vertices.

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 $|E| \geq |V| - 1$.

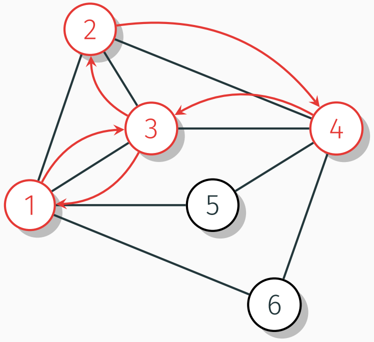
Proof.

Obvious. □

Closed trail

Definition

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



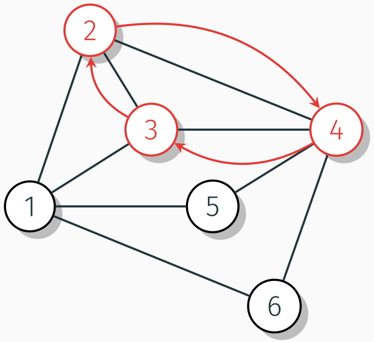
Closed trail

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

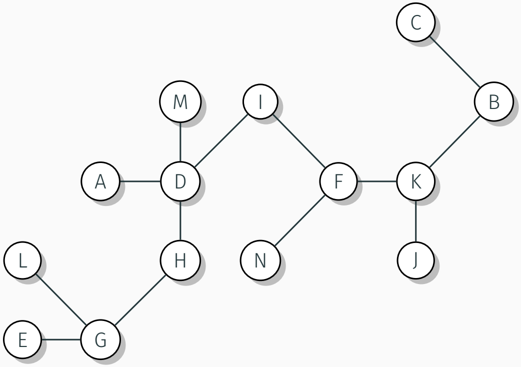
Definition

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

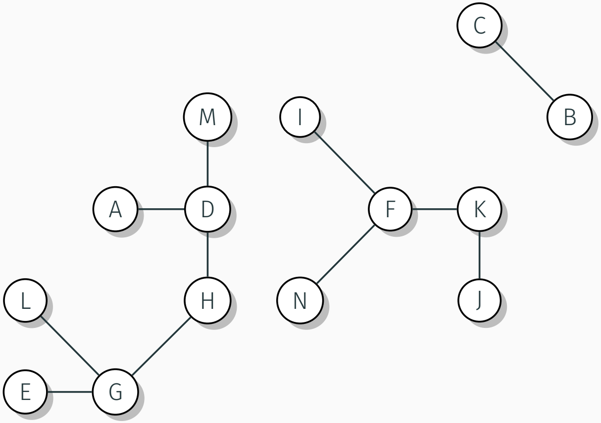
- In graph theory, the objects connected by edges are usually called vertices.
- When discussing trees, the term **node** can also be used for a vertex.
- The terms vertex and node are equivalent; it is more a matter of convention.

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

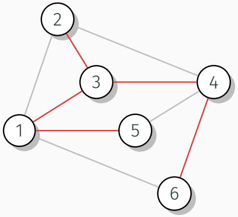
Let $G = (V, E)$ be an undirected graph, then the following statements are equivalent

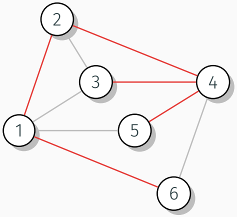
1. *G is a free tree.*
2. *Every two vertices in G are connected by exactly one path.*
3. *G is connected, but if we remove any edge, we obtain a disconnected graph.*
4. *G is connected, and $|E| = |V| - 1$.*
5. *G is acyclic, and $|E| = |V| - 1$.*
6. *G is acyclic. Adding a single edge to the set of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

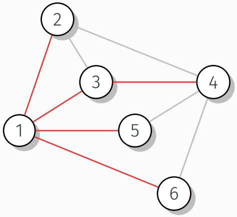
Spanning Tree

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.







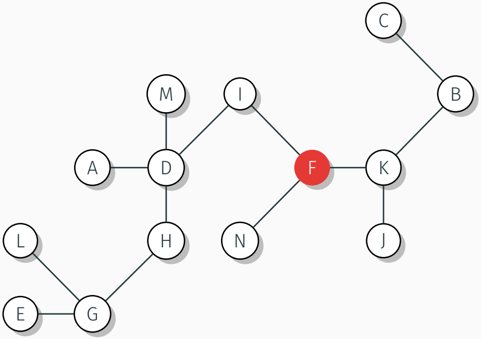
Remarks

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

Rooted Tree

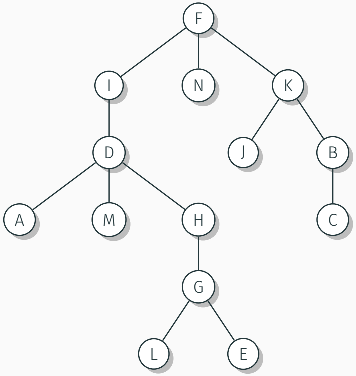
Definition

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

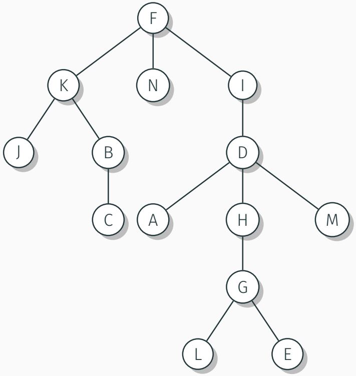


Rooted tree – a Common Visualization

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.

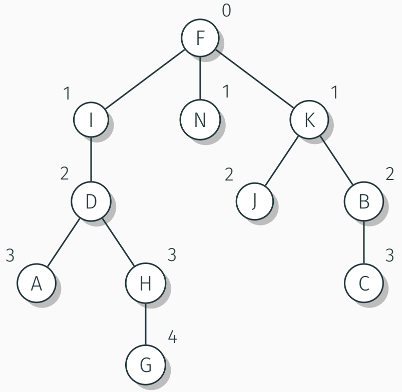
Depth of a Vertex – Height of a Tree

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.

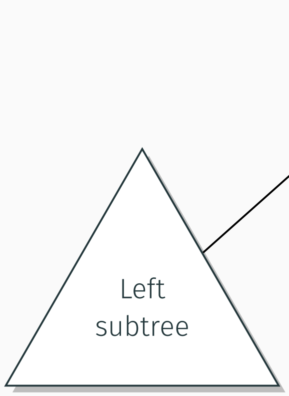
Binary Tree

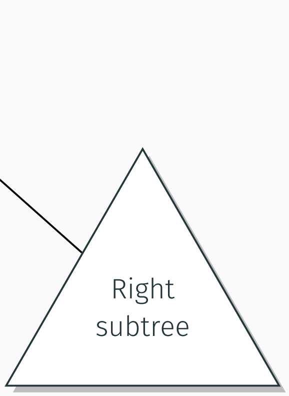
Definition

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

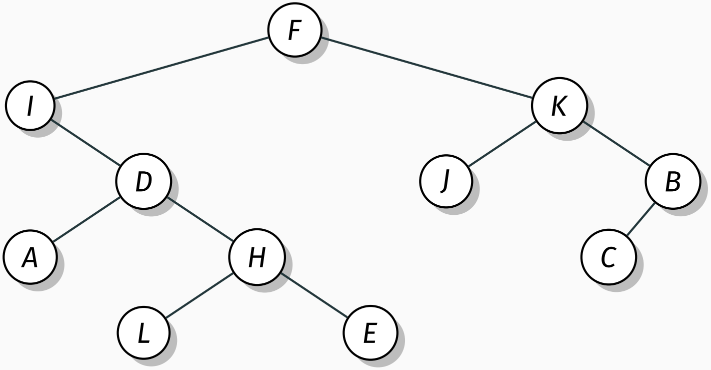
- **Rule 1**
  contains no nodes, i.e., $M = \varnothing$, or
- **Rule 2**
  is composed of three disjoint sets of nodes L, R, and $\{r\}$, $L \cup R \cup \{r\} = M$:
  - the root of the tree r,
  - a binary tree over set L, called the left subtree, and
  - a binary tree over set R, called the right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

Binary Tree – Graphical Representation of Recursive Definition

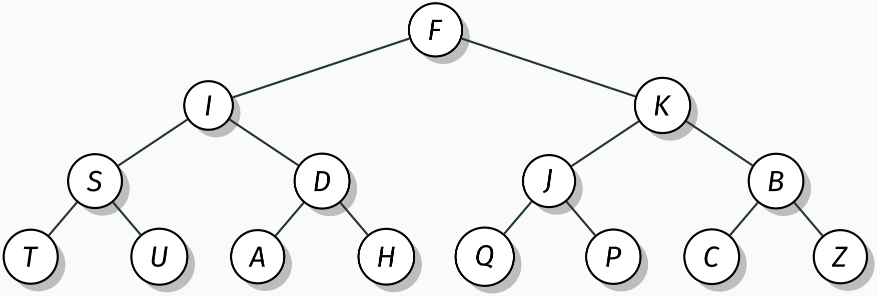




Binary Tree – Example



Complete Binary Tree

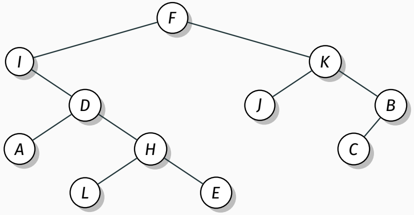


Complete Binary Tree – every internal node has exactly two children.

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:

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

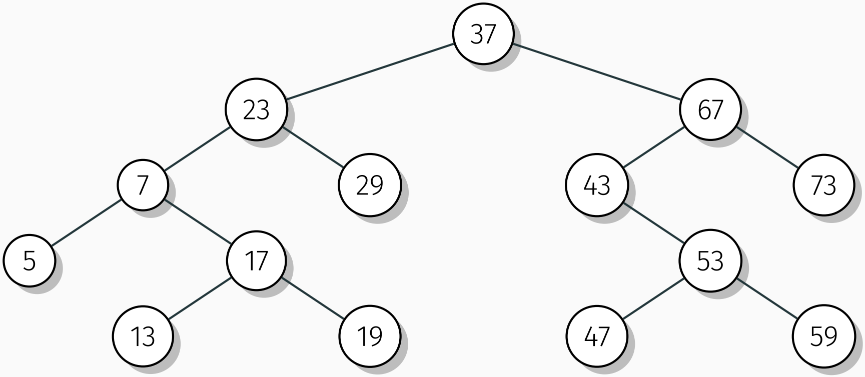
A binary tree, in which this rule applies to all its nodes, is called a **binary search tree**.

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.

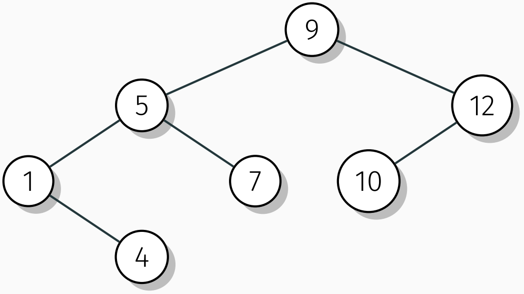
For the height h of a binary tree with n nodes, the inequality holds:

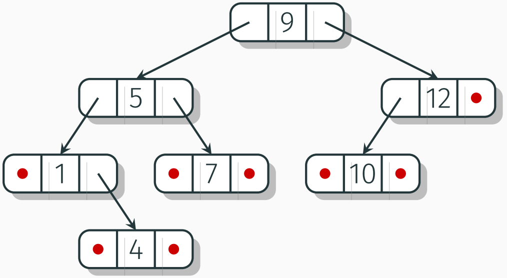
$$\lfloor \log_2 n \rfloor \leq h \leq n - 1$$

Binary Search Tree – Insertion

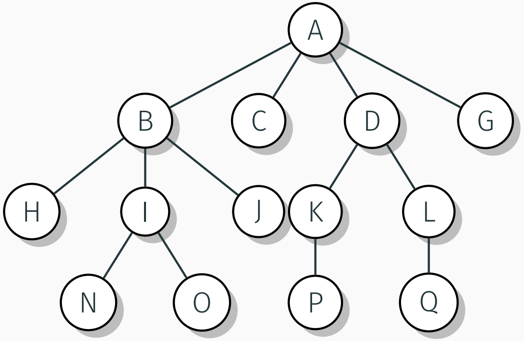
- The insertion of a key must correspond to the search algorithm.
- First, we must attempt to find the key being inserted in the tree.
- If it is not found, then the place where the search ended unsuccessfully corresponds to where this key should be in the tree.
- This follows from the uniqueness of 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



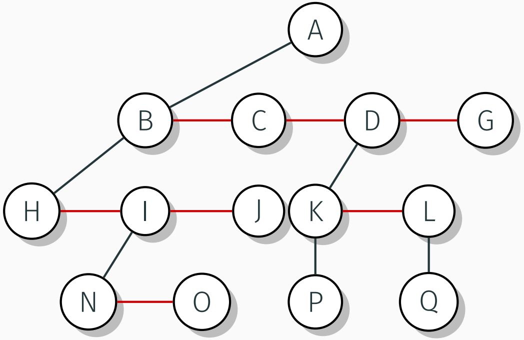


Ordered Tree Representation



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

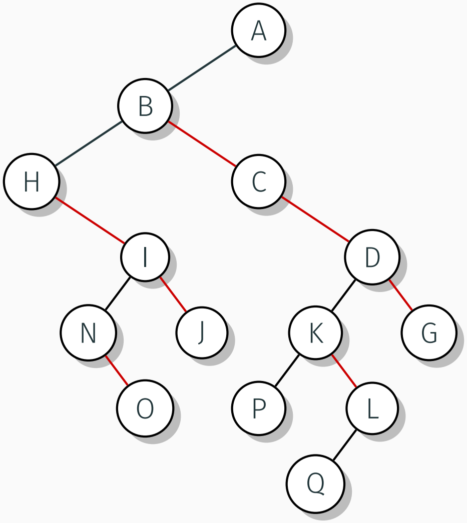
Ordered Tree Representation – first child – next sibling



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., $M = \{n, \text{prime numbers less than } 10\}$.
- 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