Interreg
EUROPEAN UNION
Austria-Czech Republic
European Regional Development Fund

INFORMATICS

Algorithms and data structures

UNIVERSITY OF APPLIED SCIENCES UPPER AUSTRIA

VŠTE

EUROPEAN UNION

Contents

1. ALGORITHM

By the term algorithm is meant a precise instruction or procedure that can solve the given job type. It describes the theoretical principle of the solution, unlike the exact (specific) entry in the given programming language.

The algorithm should have certain properties:

Finality - Each algorithm must end in the final number of steps. There can be any number of steps, but it must be final for each individual input. A procedure that does not meet this condition cannot be called an algorithm, but only a computational method.

Generality - The algorithm does not address one particular problem, but a general class of similar problems.

Determination - Each step of the algorithm must be unambiguously and precisely defined. In each situation, it must be clear what to do and how to do it, how the algorithm should continue. Some algorithms are not determined - they contain a random element (eg: genetic algorithms)

Output (or Resultativity) - The algorithm has at least one output, the quantity that is in the required relation to the inputs. The output is the answer to the problem.

Elementarity - the algorithm consists of a finite number of simple (elementary) steps.

1.1. Algorithm development process

There are several procedures that are used to design algorithms.

Top-down method - process solutions are decomposed into simpler operations until it reaches the elementary steps.

Bottom-up method - from elementary steps, we create resources that will eventually make it possible to handle the specific problem.

Alternatively, a **combination of the two methods**, when the top-down method is completed by "a partial bottom-up step" (usin the function library, high-level programming language or system programming ...).

Usual procedures for algorithms are the following methods: Divide and conquer, greedy algorithms, dynamic programming and backtracking.

- **The Divide and conquer method** divides the problem into sub-tasks that must be independent of each other. Then these sub-tasks are solved. It is often implemented recursively or iteratively.
- **Greedy algorithm** is mostly used for solving optimization problems. It always selects a local minimum (maximum) in an attempt to find a global minimum (maximum) . This process is not always ideal and may not reach the global minimum (maximum)

Dynamic programming divides the problem into subtasks, just like the Divide and conquer method, but in such a case these parts may be dependent.

Backtracking, or search return, is a way to solve algorithmic problems based on searching the status tree of the problem. It is a brute-force search.

1.2. Types of algorithms

Algorithms can be divided into several types.

- **Recursive algorithms** that use (call) themselves.
- **Randomized algorithms** that makes some decisions of random (pseudo-random) choices.
- **Parallel algorithms** that allocate tasks between more computers (processors, threads).
- Another type is **genetic algorithms** based on the imitation of biological evolutionary processes.
- **Heuristic algorithms** that do not search for a precise specific solution, but just some appropriate approximation. They are used in such situations where available resources are insufficient to be used for exact algorithms or no suitable exact algorithms are known at all.

2. ABSTRACT DATA TYPE – ADT

2.1. Description

Abstract Data Type - ADT - is an expression used for data types that are independent of their own implementation.

Using ADT, we try to simplify and streamline the program that performs operations with the given data type.

Each ADT can be deployed using basic algorithmic operations such as assignment, addition, multiplication, conditional jump,

2.1. Properties of ADT

ADT should have the following properties:

- **Generality of implementation -** Once designed, ADT can be embedded and run smoothly in any program.
- **Exact description** - The link between the implementation and the interface must be definite and complete.
- **Simplicity** - The user should not be involved in internal implementation and administration of ADT in memory.
- **Encapsulation** - The interface as a closed part, the user knows what ADT does, but not how it works
- **Integrity** - The user cannot intervene in the internal data structure
- **Modularity** - The "modular" programming principle is well-arranged and allows easy code parts exchange. When searching for errors, the individual modules can be considered compact units. There is no need to intervene in the whole program to improve ADT.

If ADT is programmed object-oriented, then these properties are typically met by default.

2.1. Operation in ADT

Using ADT, basic operations can be performed. These include the Constructor, Selector, and Modifier:

- The **constructor** creates a new ADT, assembles a valid internal representation of the value based on the given parameters.
- The **selector** obtains the values that make up the components or properties of a specific ADT value.
- A **modifier** performs changes to data type values.

3.ALGORITHMS ANALYSIS

3.1. Process efficiency comparison

Since one problem can usually be solved using several different procedures (algorithms), it is necessary to have a tool that allows us to compare the effectiveness of the process. Algorithms can be compared either experimentally or theoretically.

Experimental analysis is time consuming. Of course, time required increases with the amount and size of inputs. This type of analysis requires a specific implementation of the algorithm, which obviously requires additional knowledge. It's hard to find an average case. Therefore, it is better to concentrate on the worst case possible that is easy to analyze and is critical for most applications, whether it's games, finance, robotics, and automated operations.

Experimental analysis of time required takes place in the environment where the algorithm (program) is being implemented mostly using an internal time measurement function. The program run depends on the inputs and their composition, and not all inputs are included in each program run. Comparing two algorithms requires the same hardware and software and the same memory occupancy.

Instead of experimental analysis, certain theoretical procedures can be used. The theoretical analysis uses the description of the algorithm by means of operations instead of a specific implementation. It takes into account all inputs and enables to rate the algorithm speed independently of hardware or software.

One of these tools is the so-called Pseudocode. Pseudocode allows to use higher levels of algorithm description. The description is more structured than a commonly written text, but less detailed than a specific implementation. It is a preferred type of writing for describing algorithms. Its advantage and disadvantage is that it hides problems of a particular implementation.

The pseudocode uses keywords to describe the algorithm:

For run control: -If...then...else (condition), while...do, repeat...until, for...do (cycles)

Header: Algorithmus Name(arg1, arg2...), input, output

Procedure call (Methoden, Algorithmus): var.Name(arg1,arg2,...)

Value return: return *Expression*

Expressions:	←	Zuschreibung
	=	Gleichheit
	+, -, n^2,...	Mathematische Operationen

3.2. Primitive operations

Primitive operation is a basic operation performed by algorithm, identifiable in a pseudo-code, independent of the programming language, and should be precisely defined. Such a primitive operation may nr expression evaluation, assigning a value to a variable, index it in a field, procedure call or returned.

Similarly, we can use asymptotic notation (big O, Bachmann-Landau notation). It determines the operational demandingness of the algorithm by determining how the algorithm behavior will change depending on the size of the input data. Asymptotic time and space complexity is usually used. The type of the write used means that the algorithm demandingness is less than A+B · f(N), where A and B are appropriately selected constants, and N is the variable describing the size of the input data. The multiplicative and additive constants, ie. O(N+1000)=O(1000·N)=O(N) are disregarded. We are only interested in the behavior of a large N values.

To determine the time required for the algorithm using the big O notation, we must find the largest possible number of primitive operations, which we then express by means of the big O notation.

4. QUEUES AND STACKS

4.1. Stack

A stack is a data structure that serves to store data. It is characterized by a way of data manipulation - it accesses the data using the LIFO (Last In First Out) principle. It can be imagined as a plate container.

The ADT stack must contain at least:

- operations for inserting an object,
- returning and removing the last object,
- querying on the top of the stack,
- its size,
- whether the stack is empty.

If we try to perform a pop or top operation on an empty stack, we will get an Empty-StackException exception.

Stack application is, for example, the history of the web browser, the Undo sequence in the editors, or the individual procedures call. The stack can be used as an auxiliary data structure for other algorithms or as part of other data structures.

The simplest way to implement the sis by means of an array. We add the elements from left to right and the auxiliary variable holds the index of the last element.

Thanks to the array properties we get the following properties:

- ***n -*** *number of elements in the stack*
- Memory requirements - ***O (n)***
- Time requirement of each operation ***- O (1)***

Array also brings some limitations:

- At the beginning, we must define the stack size
- stack size cannot be changed at will
- Adding an element to the full stack will cause an implementation-specific exception

4.2. Fronta

The queue data structure FIFO (First In First Out). The minimum queue implementation must include operations for:

- inserting an item at the end of the queue,
- selecting an item from the beginning of the queue,
- query on the beginning of the queue,
- its length and occupancy.

Llike the stack, the queue can throw an exception during the dequeue operation or queues over a blank stack?queue - EmptyStackException.

Queue application is for example waitlist, queue, access to shared resources (printers), multiprogramming. The queue can also be used as an auxiliary data structure for other algorithms or as a part of other data structures.

The queue can be implemented using an array. To improve its properties, a circular array is used. We thus have two variables, *f* - index of the first element, *r* - index of the last element plus one (pointing to the first free space).

5. VECTORS, LISTS, SEQUENCES

5.1. Vectors

The vector extends the concept of array by storing the sequence of any objects. The element in the vector can be read, inserted and removed by determining its order The vector enables basic operations:

- elements in a specific order,
- replacing the element at a specific location, inserting it into a particular location,
- removing an element from a particular location,
- enables to determine the size and whether the vector is empty

Vectors operations may throw an exception in case of an incorrect index (usually negative). Vector application is a sorted object collection (basic database). Vectors can be implemented using an array. This brings the following properties:

- *The variable n indicates the length of the vector*
- *Operation isEmpty () elemAtRank (r) replaceAtRank (r, O) - time complexity O(1)*
- *Operation insertAtRank (r, O) - Time complexity O (n)*
- *Operation removeAtRank (R) - Time complexity O (n)*

5.2. Lists

Another data structure is the list. The list is a sequence of positions storing any data. It introduces relationships before / after between positions.

General operations are a query on size, and a blank list query. Other operations are finding out whether the given element is the first or the last one, obtaining the first and the last element, the preceding or the following element.

The ADT list contains:

- replaceElement(p, o),
- swapElements(p, q),
- insertBefore(p, o),
- insertAfter(p, o),
- insertFirst(o),
- insertLast(o),

- remove(p).

Lists can be divided into single linked list and double linked list. In a single linked list, the element contains a reference to the next node; in the case of a double linked list, the element also contains a reference to the preceding node.

5.3. Sequence

ADT sequence is a combination of a vector and a list, the elements can thus be accessed using both the position and sequence. Besides vector and list operations, it also includes interconnecting operations **atRank (r)** and **rankOf (p).**

Sequence is a general basic type that can be used to store an arranged set of elements. It is a general replacement for a stack, queue, vector or list. It can also be used as a small database.

6.TREES

Trees represent a model of a hierarchical structure that consists of nodes with a parent/child relationship between them.

Trees can be used as an organizational chart, file systems, or programming environment.

The following terminology is used to describe trees and their parts:

6.1. Types of nodes

- Root (root)
- Inner node - a node that is neither a root nor leaf
- List (leaf node, external node) - a node that has no offsprings
- Parent node - the node directly preceding the node on the path from leaf to root
- Child node - the node that follows directly after the node on the path from the root to the leaf
- Sibling - siblings refer to as the nodes with the same parent
- Ancestor node, predecessor node - a node that lies in front ofthe given node on the path to the root (the nearest ancestor is the parent)
- Successor node - a node that lies behind the given node on the path from the root to any leaf (the next descendant is a descendant)
- Depth - tree depth is the length of the longest path from root to sheet, with an empty tree defined as -1
- Level - is usually used for a set of nodes located at the same distance from the root, counted by the number of nodes

6.2. Structure

- Subtree – is a subgraph of a tree, which is also a tree (most frequently, there are subtrees formed by taking a tree node as a new root and retaining the rest of the structure,
- Branch - Each path from root to leaf.

6.3. Operations for Trees manipulation

General operations:

- integer size(),
- boolean isEmpty(),
- objectIterator elements(),
- positionIterator position().

Access operations:

- position root(),
- position parent(),
- positionIterator children(p),
- Dotazovací operace,
- boolean isInternal(p),
- boolean isExternal(p),
- boolean isRoot(p).

Actualization operations

- swapElements(p, q),
- object replaceElement(p, o).

Since tree is a hierarchical structure, it can be tracked in several ways.

Pre-order traversal

- Checking whether the node is empty or null
- Displaying the current node data
- The left subtree throughput by recursive pre-order function call
- The right subtree throughput by recursive pre-order functions call

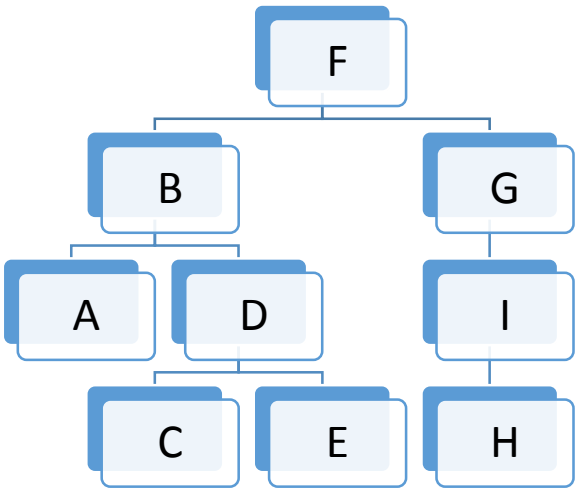
In-order throughput

- Checking whether the node is empty or null
- The left subtree throughput by recursive pre-order function call
- Displaying the current node data The right subtree throughput by recursive pre-order functions call

Post-order throughput

- Checking whether the node is empty or null
- The left subtree throughput by recursive pre-order function call
- The right subtree throughput by recursive pre-order functions call

- Displaying the current node data



Each type of throughput gives different results.

- Pre-order: F, B, A, D, C, E, G, I, H
- In-order: A, B, C, D, E, F, G, H, I
- Post-order: A, C, E, D, B, H, I, G, F

Using the ADT Tree, it is also possibleto define a Binary Tree or other types.

The binary tree extends the definition of the tree by the fact that each node has no more than two children, forming an ordered pair (left descendant, true descendant).

The binary tree adds additional operations:

- **position leftChild(p)**
- **position rightChild(p)**
- **position sibling(p)**

7.PRIORITY QUEUE AND HEAP

7.1. Priority queue

Priority queue stores a collection of items where the item is ordered pair key (priority) -the value.

Basic implementation should include:

- **insertItem(k, o),**
- **removeMin(),**
- **minKey(k, o),**
- **minElement(),**
- **size(),**
- **isEmpty().**

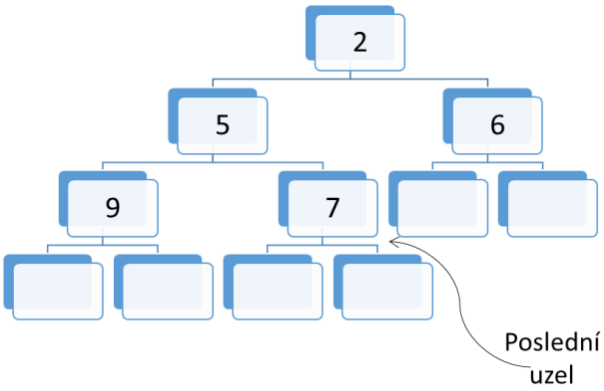
The priority queue application is for example auctions and stock exchanges. The priority queue key can be any object with a defined order and which can be arranged. Two different elements (values) can have the same key (priority).

To use the priority queue, the ADT Comparator must be implemented, which enables to compare two objects.

7.2. Heap

Heap is a binary tree that stores keys as internal nodes. For each tree node outside the root it holds true that the node key is larger than the parent key. For each heap a complete binary tree is defined. If h is the height of the tree, then for i from 0 to $h-1$ there is 2^i nodes of i depth.

The last heap node is an internal node, which is located at the rightmost at the $h-1$ level.



Heap can be used to implement the priority queue. Then we store an item (key, value) in each internal node, and keep a reference to the position of the last element.

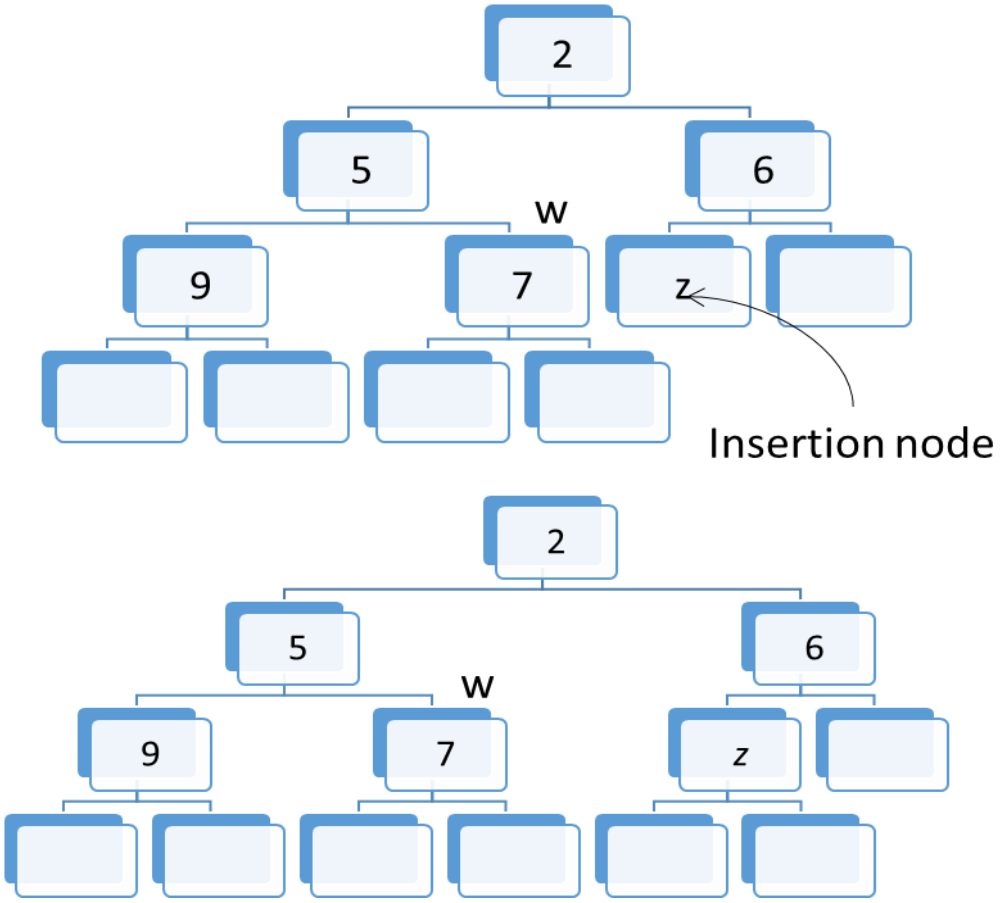
7.3. Handling a heap

Operace insertItem (k,o)

- Finding the node where it will be inserted
- Saving the k key into the z node, changing z node into internal node
- Recovering the heap order (checking properties) - upheap () operation

Operace removeMin ()

- It corresponds to removing the root from the heap (node 2)
- Changing the root for the last node (2-7)
- Changing the w node and its children into the list



upheap()

- inserting a node may affect the arrangement/layout
- Upheap algorithm restores the arrangement by swapping *k* key upwards from the inserted node
- It ends when the internal node becomes the root or parent key is less than or equal *k*

downheap()

- removing a node may affect the arrangement/layout
- downheap algorithm restores the arrangement by swapping k key downwards from the node
- it ends when the internal node becomes a leaf or the child key is greater than or equal *k*

8.DICTIONARIES AND HASH TABLES

8.1. Dictionary

Dictionary refers to the ADT containing a key-value collection that can be searched for. Operations that can be performed with the dictionary:

- **findElement(k),**
- **insertItem(k, o),**
- **removeElement(k),**
- **size(),**
- **isEmpty(),**
- **keys(),**
- **elements()**

Dictionary application is a directory, credit card authorization, dictionary, domain name translation to IP address.

Log File - dictionary implemented as a random sequence (double linked list). We store the objects in any order.

Complexity of operations:

- Vložení objektu O(1)
- Nalezení prvku, odebrání prvku O(n)

The log file is suitable for small dictionaries or applications where the most frequent operation is inserting, while searching and removing is rarely done.

The **findElement (k**) operation on a dictionary implemented by a sequence based on a key-based array is performed as a binary search. At each step, the candidate number is divided by two, ending after the logarithmic number of steps.

Lookup table is a Dictionary implemented using an arranged sequence. We store the dictionary entries in a sequence based on a key-based array; an external key comparator is required.

Complexity of operations:

- Finding the element O(log(n))
- Inserting element, removing element O(n)

This is effective for small dictionaries or applications where searching is a frequent operation.

Binary search tree is a binary tree for which the following rules apply:

- u, v and w are such nodes for which it holds true that u is in the left v subtree and w is in the right v subtree
- $key(u) \leq key(v) \leq key(w)$
- External nodes do not store any items.
- The in-order throughput gives the keys in the ascending order

8.2. Hash table

Hash function h is a function that allocates an integral value from the 0 and N-1 interval to the key of a given type between 0 and N-1. The purpose of this function is to split the keys uniformly at a given interval.

A hash table for a given key type contains a hash function and a N-size array (table).

The key is replaced by a hash value. However, it may happen that for the two keys the same hash value is generated - a collision occurs. This can be solved in two ways:

- by chaining - conflicting items are stored as sequences
- by open addressing - the item is saved elsewhere in the table.

9. SORTING ALGORITHMS

9.1. Explanation

Sorting algorithms are used to sort the data file in a specific order, either alphabetical or numerical. The key-value pair is ranked by the key and the value is not taken into account.

Sorting algorithms can be divided into stable and unstable, depending on whether they keep the order of items with the same key, natural and unnatural - natural work faster with a partially arranged set.

They can also be broken down by the sorting type:

- By selecting
- By inserting
- By replacing
- By merging

The best-known algorithms:

- Bubble sort
- Heap sort
- Insertion sort
- Merge sort
- Quicksort
- Selection sort

Other algorithms based on a different principle

- Bucket sort
- Radix sort
- Counting sort

9.2. Bubble sort

- Easy to implement.
- Universal, local (in-place, no need of extra memory).
- The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass.

Algorithm:

```
procedure bubbleSort( A : list sortable items )
    n = length(A)
    repeat
        swapped = false
        for i from 1 to n-1 inclusive do
     if A[i-1] > A[i] then
                        swap( A[i-1], A[i] )
                swapped = true
            end if
        end for
    until not swapped
end procedure
```

9.3. Heap sort

- A comparison-based sorting algorithm.
- Not stable.
- Uses the heap data structure and its properties.

Algorithm:

```
procedure heapsort(a, count) is
    input: an unordered array a of length count
    heapify(a, count)
   end ← count - 1
    while end > 0 do
         swap(a[end], a[0])
        (the heap size is reduced by one)
        end ← end - 1
        (the swap ruined the heap property, so restore it)
        shiftDown(a, 0, end)
```

If the smallest element is the root – placed on the first place in the array and the root is removed.

Downheap() – recovering the heat following the rulesWe repeat removing the root and heap recovery until the heap is empty.

9.4. Insertion sort

- A simple sorting algorithm that builds the final sorted array (or list) one item at a time
- Simple implementation
- Efficient for (quite) small data sets
- Efficient for data sets that are already partly sorted
- Stable, on-line, in-place

Algorithm:

```
for i = 1 to length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
    end while
end for
```

9.5. Merge sort

- is an efficient, general-purpose, comparison-based sorting algorithm
- the divide and conquer method
- Stable, divide and conquer algorithm, easy to paralelized
- Worst and average time: O(NlogN)
- Extra memory needs: array of N size

Algoritmus

```
mergesort(m)
      var list left, right
      if length(m) ≤ 1
            return m
      else
           middle = length(m) / 2
           for each x in m up to middle
                add x to left
           for each x in m after middle
                add x to right
left = mergesort(left)
           right = mergesort(right)
           result = merge(left, right)
      return result
```

9.6. Quicksort

- an efficient sorting algorithm
- It takes: O(NlogN) – O(N2)
- Divide and conquer algorithm, not stable, in-place
- Recursive
- The steps are:
    - Pick a pivot from the array.
    - Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
    - Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.
- Pivot selection –median is ideal
    - First element (any fix position) – not suitable for partly arranged sets
    - Random element – actually pseudo-random
    - Median of three (five...) – or other number of elements from the fix or random positions

When pivot is selected properly, it does not need extra memory. Quicksort is an unstable algorithm. The pivot selection method affects sorting, but on average, it is the fastest universal array sort algorithm in the computer operational memory.

9.7. Selection sort

A simple algorithm whose time complexity is O (N 2),It is suitable for small data volumes. It is universal, local and unstable.

Procedure:

1. Dividing the sequence into an arranged and unarranged unassigned part.
2. Finding the element with the smallest value in the unarranged part of the sequence.
3. Replacing it with the element in the first position of the unarranged part.
4. The first element of the unarranged part is included in the arranged part and at the same time the unarranged part is reduced by 1 element from the left.
5. The remainder of the sequence is arranged by repeating steps 2 - 5 for the remaining unarranged part.
6.

Comparison of sorting algorithms:

Name	Time complexity			Extra memory	Stable	Natural	Method
	Minimum	Average	Maximum				
Bubble sort	*O(n)*	$O(n^2)$	$O(n^2)$	*O*(1)	yes	yes	Exchange
Heapsort	*O*(*n* log *n*)	*O*(*n* log *n*)	*O*(*n* log *n*)	*O*(1)	no	no	Heap, exchange
Insertion sort	*O(n)*	$O(n^2)$	$O(n^2)$	*O*(1)	yes	yes	Inserting
Merge sort	*O*(*n* log *n*)	*O*(*n* log *n*)	*O*(*n* log *n*)	*O*(log n)	yes	yes	Merging
Quicksort	*O*(*n* log *n*)	*O*(*n* log *n*)	$O(n^2)$	*O*(log *n*)	no	no	Exchanging
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	*O*(1)	no	no	Selection

9.8. Bucket sort

It divides data into several buckets, its time requirement is O (n * k), where k = n / m, input n, number of bucket is m.

To use Bucket sort, prerequisites are required:

- Suitable for evenly distributed input data values.
- The algorithm for arranging the buckets must be stable

Procedure:

- Input data are divided into a predefined number of buckets.
- For each bucket a stable sorting algorithm is called.
- The individual buckets are gradually copied into the output array.

The benefits of the bucket sort are that it is well parallelizable and does not require having all the data in memory at the same time.

9.9. Radix sort

It sorts integers by scanning through all digits. There are two approaches:

- LSD (Least Significant Digit)
- MSD (Most Significant Digit)

Time required: O((z+n)*$\log_z u$), where z is the basis of the selected number system, *n* are the numbers at the input and *u* is the maximum range of numbers at the input

It is not suitable for unlimited input size.

Example of LSD radix: 170, 45, 75, 90, 802, 2, 24, 66 ⇒ 170, 90, 802, 2, 24, 45, 75, 66 ⇒ 802, 2, 24, 45, 66, 170, 75, 90 ⇒2, 24, 45, 66, 75, 90, 170, 802

9.10. Counting sort

Suitable for large files with a small amount of discrete values; stable. The time required is *O(N+M)* and extra memory needs *O(M.)*

Prerequisites:

- Number of different values *(M)* is significantly smaller than the total number of elements *(N).*
- Auxiliary array - writing and reading at a constant time (array indexed by value or hash values).

Algorithm:

- Passing the input array from the left (or from the right).
- For each element, the frequency of occurrence of this element is increased in the auxiliary array.
- To each item, the number of occurrences of all previous items (it gets the exact position of the border) is added.
- Starting to pass the unarranged array from the right.
- For each element, it looks into the auxiliary array at the top of the placement boundary.
- Placing the element into that border and reduces it by one.
- Repeating until the entire array is passed.

10. PATTERN MATCHING

Pattern matching, is searching for a specific pattern in a given sequence, usually searching for a substring in a string

First, a string must be defined. String is a sequence of characters from the given alphabet. Alphabet is a set of all possible characters - Ascii, Unicode, {0,1}, {A, C, G, T}

If the *P* string length is *m,* then the substring *P [i..j]* of *P* string consists of characters between *i* and *j.* The string in front of the index *i* is prefix. The string located behind the index *j* is a suffix.

Application - text editors, search tools, biological research.

There are several algorithms for pattern matching.

The basic one is the **Brute-Force** (brute force).

It passes the text from left to right

It compares the *P* pattern with the T text *,* for every possible position until:

a match is found

all possible positions have been tested

Time required for this algorithm is *O (nm).*

Boyer-Moore algorithm goes through text from the end (right to left).

We define:

- The index i indicates the position in text T, the index j points to the P

During the search, there are 4 possible situations:

- *T (i)* is not in *P;* then i will be moved bythe *P* length *(P* aligned to the next letter *T,* that is, *T (i + 1))*
- *T (i)* corresponds to *P (j)* – we move in both to the left and repeat (as in the brute force)
- *T (i)* is *P (j) T (i)* is in front of the index *j P -> P* aligned to the right so that *T (i)* corresponds to the occurrence of *P* and repeat
- *T (i)* is *P (j) T (i) P* is an index *j* -> move to the right by 1 and repeat (not recurring, but not move more below)

Return *i* - if the whole pattern is found.

The algorithm is faster than the Brute-Force, however, its complexity can be *O (mn + A),* where A is the size of the alphabet.

Preprocessing is required to apply this algorithm. It finds out the position of the letters (from the left)

- If the patern is eg: "ABRAKADABRA"
- A gets index 10, B gets index 8, K = 4, D = 6 and R = 9. For other letters, we assign -1.
- We implement the function Last (char input) which returns the index according to the letter, then, for cases 3 and 4, we compare the last *(T (i))* and *j,* and thus we know whether to move it tothe Last *(T (i))* (if the Last is *(T (i)) <j)* or only *i ++*

Knuth-Morris-Pratt (KMP) algorithm

It searches thetext from left to right, unlike brute-force it does not make all comparisons. If a disagreement is found, it moves by more than one letter. If we find a P (from the beginning, a prefix), the characters of this prefix match the text, so no need to check them again. The end of the found substring can also be included at the beginning of this substring. Such a match is, of course, the whole of the P found, so we are looking for P + 1. So we go from the end of the found P piece from the left and right, and when we can not find a match, we know how much we can move. This can be computed into a table - then all is O (1).

Trie

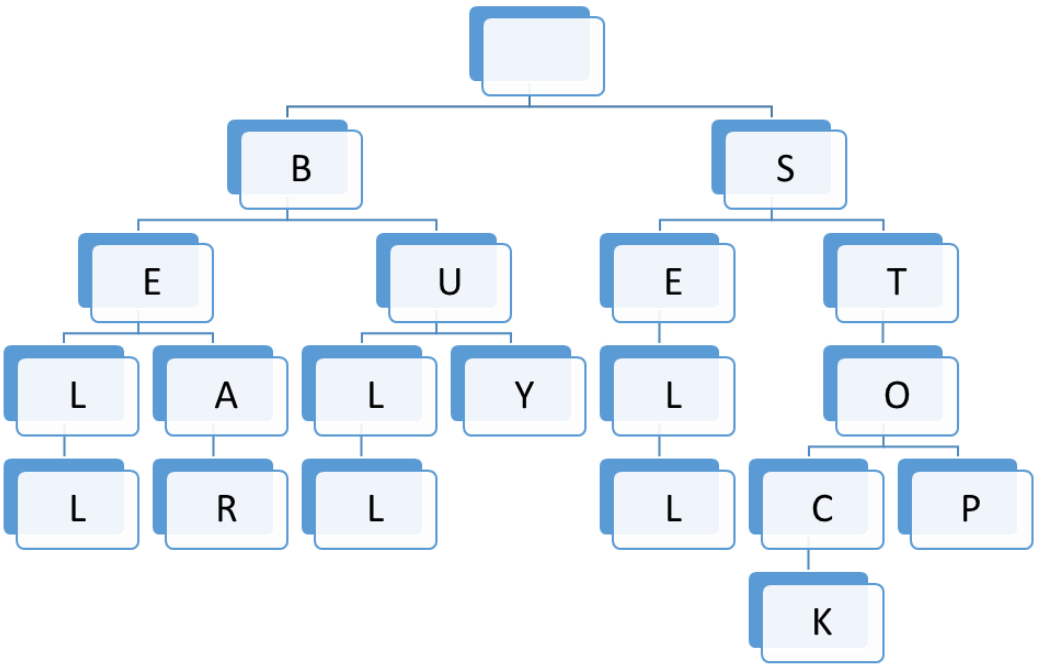
Trie is a tree structure for text pre-processing. Each node has one letter. The length of the path from the node to the top determines the position of the letter in the word.

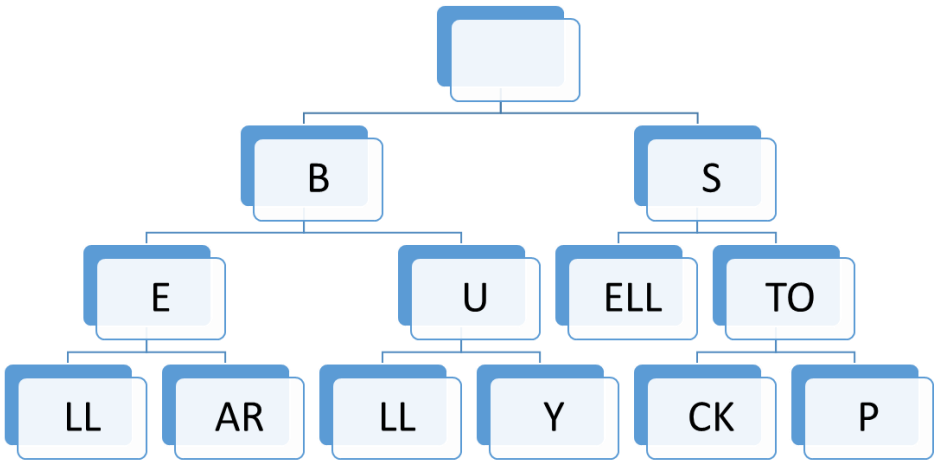
The search has a time requirement O (dm), where d is the alphabet size, m is the length of the word.

It is possibleto save all text in the Trie structure, then each node contains one word.

For saving,the so-called compressed Trie can be defined. The tree then contains nodes of at least degree two (two letters per node).

Example:S={BELL, BEAR, BULL, BUY, SELL, STOCK, STOP}





11. GRAPH THEORY

Graph is an ordered pair (V, E) where V is the set of vertices and E is the set of edges. Each edge is determined by just two vertices, optionally by the direction or weight.

11.1. Types of edges

There are several types of edges:

- Oriented - ordered pair of vertices *(u, v)* where *u* is the beginning, *v* is the end/ target
- Unoriented - ordered pair of vertices *(u, v)*
- Loops - the edge begins and ends at the same vertex
- Multiple edge (multiple, parallel) there are more edgesbetween vertices *(u, v)*

11.2. Types of graphs

Similarly, there are several types of graphs.

- Oriented - all edges are oriented
- Un-oriented - all edges are not oriented
- Multigraph - Contains multiple edges

11.3. Terminology

- End vertices (or endpoints) of an edge
- Edges incident on a vertex
- Adjacent vertices
- Degree of a vertex
- Parallel edges
- Self-loop
- Path
    - sequence of alternating vertices and edges
    - begins with a vertex
    - ends with a vertex
    - each edge is preceded and followed by its endpoints
- Simple path
- path such that all its vertices and edges are distinct
- Cycle

    - circular sequence of alternating vertices and edges
    - each edge is preceded and followed by its endpoints
- Simple cycle
    - cycle such that all its vertices and edges are distinct

11.4. ADT Graf supports operations

Access operations

- aVertex()
- incidentEdges(v)
- endVertices(e)
- isDirected(e)
- origin(e)
- destination(e)
- opposite(v,e)
- areAdjecent(v,w)

Update operations

- insertVertex(o)
- insertEdge(v, w, o)
- insertDirectedEdge(v, w, o)
- removeVertex(v)
- removeEdge(e)

General Operations

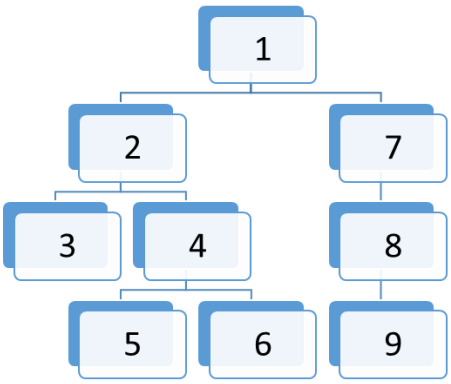
- numVertices()
- numEdges()
- vertices()
- edges()

The graph structure needs to be searched in some way. There are two options - DFS (Depth-First Search) and BFS (Breadth-First Search).

11.5. Depth-First Search

Depth-first search is a complete algorithm (passes through each node). Its principle consists in the fact that it expands the first successor of each peak if it has not visited it yet. If a peak is encountered from which it is not possible to continue (there are no successors or all of them have already been visited), it goes back by backtracking.

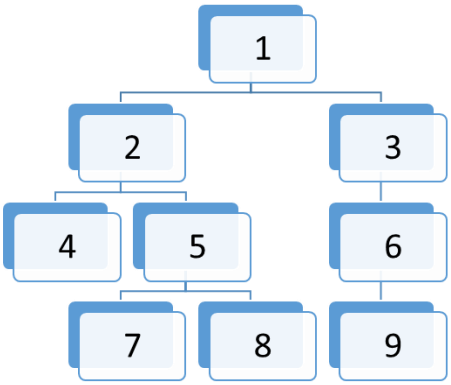
It passes the nodes in the following order:



11.6. Breadth-First Search

Breadth-First Search is a similar algorithm, it passes through all the neighbors of the starting peak, then the neighbors of neighbors etc. to pass through the entire connection component

It passes the nodes in the following order:



The fundamental question of graph theory is finding the shortest path between two nodes. There are several algorithms for that.

Dijkstra's algorithm

- non-negative weights on the edges
- $O(|V|^2+|E|)$ – V number of vertices, E number of edges

Bellman-Ford algorithm

- Graph can have negative edges
- *O(V·E)* – slower than Dijsktra's alg.

Floyd-Warshall algorithm

- Directed/oriented graph with non-negative edges
- It finds the shortest path between all vertices
- Time required – $O(V^3)$, memory required - $O(V^2)$

Johnson's algorithm

- oriented graph, it may contain negative edges
- find the shortest paths between all pairs of vertices in a sparse, edge weighted, directed graph. It allows some of the edge weights to be negative numbers
- $O(V^2 \log_2(V) + VE)$

12. GENETIC ALGORITHMS

12.1. Explanation

Genetic algorithms belong among the evolutionary algorithms and are parts of artificial intelligence. They are a class of heuristic algorithms.

They use the knowledge of evolutionary biology to solve complex problems for which there is no exact algorithm.

It imitates the techniques of evolutionary biology:

- Heredity
- Mutation
- Natural selection
- Crossover

The principle of genetic algorithms works according to the scheme:

1. Initialization – generate the 0 generation
2. Begining of cycle – Choose (randomly) several individuals from whole population
3. Create a new generation using the following methods:
    i. crossover - „swap“ parts of few individuals
    ii. mutation – randomly change of some genes
    iii. reproduction – copy individuals without changes
4. Calculate the capability of the new generation
5. Termination of the cycle - Repeat from point 2 until the termination condition is reached
6. The end of algorithm – the individual with the best capability is the main algorithm output and represents the best possible solution

12.2. Terminologie

- Phenotype - an individual's designation
- Genotype, genome, chromosome - representation of phenotype
- Chromosome - divided into different linearly arrayed genes (i -th chromosomal gene of the same type representing the same characteristic)
- Alleles - Various gene values
- Fitness value - ranging from 0-1, expresses the quality of each individual

Each individual can be encoded (genetically described) in a different way. The description method can be important for the success or failure of solving a specific task.

12.3. Example:

0th Generation (fitness value # "1"):

- 0100011011 *f*=0,5
- 0101000100 *f*=0,3
- 1010110000 *f*=0,4
- 1110111000 *f*=0,6

Selection

$$p_i = \frac{f_i}{\sum_1^N f_i}$$

- Weighted roulette:
    - Probability of being a parent
- Tournament method
    - Random selection of groups from each parent group becomes the person with the highest fitness value
- Trimming
    - We sort all the individuals according to the f value, cut the low value part, select the parents from the rest
- Random selection
    - The simplest method, f value does not play a role in selecting an individual for parenting
- Crossover
    - Parents exchange parts of their genetic code
    - Simplest method– one point crossover
    - Place for cutting – randomly chosen
        - X: 010001|1011
        - Y: 111011|1000
    - P: 0100011000 *f*=0,3
    - Q: 111011 1011 *f=0,8*
    - More-point crossover , possibility of more than two parents

- Mutation
  - Random change of the random gene in an individual
  - Very low probability

    1. 0100011011 ⇒ 0101011011
    2. 0101000100 ⇒ 0101100100
    3. 1010110000 ⇒ 1010110100
    4. 1110111000 ⇒ 1010111000

  - It is possible to reach properties which are not in the original generation
- Termination
  - This generational process is repeated until a termination condition has been reached. Common terminating conditions are:
  - A solution is found that satisfies minimum criteria
  - Fixed number of generations reached
  - Allocated budget (computation time/money) reached
  - The highest ranking solution's fitness is reaching or has reached a plateau such that successive iterations no longer produce better results
  - Manual inspection
  - Combinations of the above