Department of Computer Science
Faculty of Science
Palacký University Olomouc

BACHELOR THESIS

Visualization of Sorting Algorithms

2019

Mykhailo Klunko

Supervisor: Mgr. Tomáš Kühr, Ph.D.

Study field: Computer Science, full-time form

Bibliografické údaje

Autor:	Mykhailo Klunko
Název práce:	Vizualizace třídicích algoritmů
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2019
Studijní obor:	Informatika, prezenční forma
Vedoucí práce:	Mgr. Tomáš Kühr, Ph.D.
Počet stran:	19
Přílohy:	1 CD/DVD
Jazyk práce:	anglický

Bibliograhic info

Author:	Mykhailo Klunko
Title:	Visualization of Sorting Algorithms
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2019
Study field:	Computer Science, full-time form
Supervisor:	Mgr. Tomáš Kühr, Ph.D.
Page count:	19
Supplements:	1 CD/DVD
Thesis language:	English

Anotace

Cílem práce bylo vytvořit software pro podporu výuky třídících algoritmů pomocí vizualizace průběhu třídění nejznámějšími algoritmy a jejich variantami. Program byl vytvořen s podporou názorné vizualizaci vybraných algoritmů na zadaném či vygenerovaném vstupním poli a krokování průběhu výpočtu se souběžným zobrazením pseudokódu použitého algoritmu a aktuálních hodnot použitých proměnných.

Synopsis

The main goal of the thesis was to create a learning support software with visualization of the most known sorting algorithms and their variations. The application has to support a graphic visualization of selected algorithms on randomly generated or manually created array, step-by-step execution possibility, pseudocode and current state of variables.

Klíčová slova: třídící algoritmus; třídění; vizualizace; program

Keywords: sorting algorithm; sorting; visualization; software

Děkuji, děkuji, děkuji.

I hereby declare that I have completed this thesis including its appendices on my own and used solely the sources cited in the text and included in the bibliography list.

date of thesis submission author's signature

Contents

List of Figures

List of Tables

List of theorems

List of source codes

1 Introduction

Nowadays sorting algorithms are widely used in software. For example, if you open file explorer on your PC, you may see files sorted in different ways. Searching in sorted data is more efficient than not sorted. Students of computer science start learning different algorithms in the first year of studies and sorting algorithms are among them.

Since I faced the problems of sorting during the course of algorithm design in the first year of my studies, there is an understanding that the visual representation is a vital part of the studying process. During working on the thesis it was very exciting to learn different techniques of sorting algorithms into the depth.

The main goal of the thesis was to create a program which would serve as a tool for understanding how most known sorting algorithms work. There was an attempt to make the best possible user experience. The demonstration software is made in a user-friendly and easy-to-use style. To gain maximal benefit from learning you can try each sorting algorithm on your own data.

The text of the thesis describes principles of the most known sorting algorithms which are demonstrated in the computer program. It might be used as a source for learning algorithms by students. Also, the program might be easily used as a demonstration by lecturers and tutors during classes. In addition, there is programmer documentation and user guide to the provided software.

Readers of this text are expected to have some programming experience to know basic data structures such as arrays, lists, trees and understand recursive procedures. Also, knowledge of some simple algorithms and their implementations could be helpful. In order to understand the topic better, knowledge of linear algebra and calculus is involved.

I want to apologize to the reader for possible mistakes and typing errors in this text. Thank you for understanding.

1.1 What is an algorithm?

Before moving on to the main part of the text which describes algorithms and the software, we need to assure that there is an understanding of the basics. We shall begin with the algorithm definition.

What is an algorithm? How can we define it? We may say simply that an algorithm is a sequence of steps or instructions that solves some kind of problem. Although it may be a bit imprecise because we have not defined what does problem mean and what does instruction mean.

Problem is a kind of task that we need to solve. We are facing different problems every day: finding the fastest route to work or home, etc. However, not all of them are problems that may fit our algorithm definition. A suitable definition of a problem has some limitations: problem should be specified by its inputs and all inputs have to be mapped to some outputs.

Step or instruction is some action that is clear to its executor. In our case, it could be a PC.

To solve a problem means to find a solution for each input.

Here is more precise definition by Thomas Cormen: "Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output." [1]

5 features of an algorithm according to D. Knuth[2]:

- *finiteness* - an algorithm should end in a finite number of steps.
- *definiteness* - each step of an algorithm should have precise definition. And it means that for the same inputs we will obtain the same results.
- *input* - an algorithm may have inputs, they are taken from some set of objects.
- *output* - an algorithm may have outputs which should be in some relation with inputs.
- *effectiveness* - we may expect an algorithm to be effective. It means "its operations must all be sufficiently basic that they can in principle be done exactly and in a finite length of time by someone using pencil and paper."[2]

1.2 Types of algorithms

1.3 Complexity

Frequently happen situations when we need to know the performance of some algorithm. There might be a necessity to compare performance of some algorithms, for example. Or we might need to know the time for an algorithm to

run with some input. And here we are moving to the concept of the algorithm complexity.

There is a time complexity and a space complexity. In general, complexity gives the amount of time or space needed to run an algorithm. To be more precise: time complexity is a function $f : \mathbb{N} \to \mathbb{N}$ that maps each input to the maximum number of steps for this input needed for an algorithm to complete a task. We will be talking here only about the time complexity and will call it just complexity.

Analyzing algorithms we might want to investigate different sides of the algorithm performance. Usually, it could be a *worst-case analysis* or an *average-case analysis*. The worst-case analysis implies determining the longest running time of all inputs with the same length. In the average-case analysis, we take the average running time of inputs with the same length.

Often complexity of an algorithm is measured on the large inputs. Since complexity is expressed by a polynomial, we might want to have it in a more convenient form. This type of estimation is called an *asymptotic analysis*. Such form takes only the highest order term of the polynomial.

To be more clear, let us have an example. $f(n) = 5n^3 + 10n^2 + 20n + 4$, the highest degree term here is $5n^3$. In some cases we can omit the coefficient 5, now f is asymptotically at most n^3.

For such approximation there exists **O notation**:

1. *Asymptotic upper bound*(or O). For the function $f(n)$, $O(f(n))$ means that the running time or complexity of an algorithm grows as much as the $f(n)$ but may grow more slowly. It represents a worst-case complexity.

2. *Asymptotic lower bound*(or Ω). If the complexity of some algorithm is in $\Omega(f(n))$, it means that there exists some large n such that the function $f(n)$ is a lower bound for the algorithm running time. It represents the best-case complexity.

3. *Asymptotic tight bound*(or Θ). This notation combines both lower and upper bounds. This way, it shows an average-case complexity.

Also, there is a *small-o* notation. It slightly differs from the *Big-O* notation. "Big-O notation says that one function is asymptotically no more than another. To say that one function is asymptotically less than another we use small-o notation."[3]

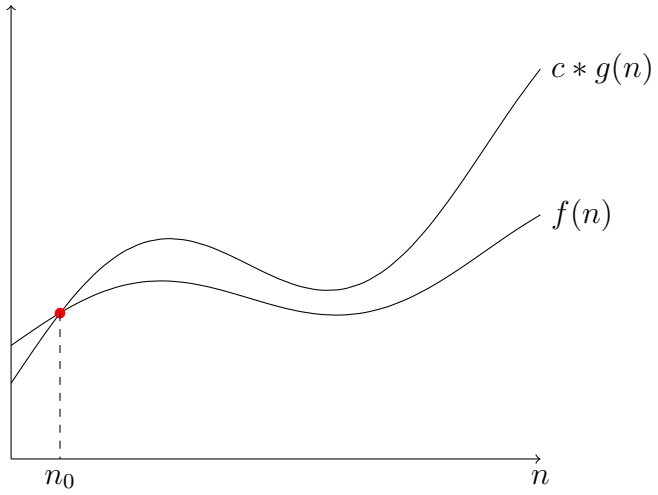

Figure 1: $f(n)$ in $O(g(n))$

For the asymptotic analysis of an algorithm mostly the worst-case complexity is used. Algorithm is considered for practically solvable if its running time is in $O(n^c)$, c is usually a small constant.

Here are some complexities (from the slowest to the fastest growing):

1. $O(1)$ - constant complexity
2. $O(\log(n))$ - logarithmic
3. $O(n)$ - linear
4. $O(n \log(n))$ - linear-logarithmic
5. $O(n^2)$ - quadratic
6. $O(2^n)$ - exponential
7. $O(n!)$ - factorial

1.4 Sorting

As was already said that sorting is used for solving a wide range of problems. It may be used for further searching or, for example, as part of different complex tasks. We were talking about sorting. But what actually sorting is?

Simply said, sorting is a process of rearranging of items, which are possible to compare, in ascending or descending order. In the text we are meaning only ascending order if not stated in a different way.

Ascending order means that items in a sequence are arranged from the smallest to the largest item. On the contrary, descending order means positioning from the largest to the smallest item.

Sorting algorithms are divided into two main types:

1. *Algorithms of internal sorting* - all the data to sort is stored in the internal memory during the sorting process.

2. *Algorithms of external sorting* - all the data to sort is stored outside the internal memory (e.g. on a hard disk) and is loaded to the internal memory by small parts.

Here, in the text, we are talking only about the algorithms of internal sorting. Talking about the algorithms of the internal sorting, here are five main techniques which are usually used[4]:

1. *Sorting by Insertion*

2. *Sorting by Exchanging*

3. *Sorting by Selection*

4. *Sorting by Merging*

5. *Sorting by Distribution*

2 Algorithms

This section describes algorithms represented in the program.

2.1 Insertion Sort

Insertion Sort algorithm has a simple idea. Assume an array with items to be sorted. We divide the array into two parts: sorted one and unsorted one. At the beginning sorted part consists of the first element. Then, for each item that we have in the unsorted part, we take element and insert it into the right place among the sorted items.

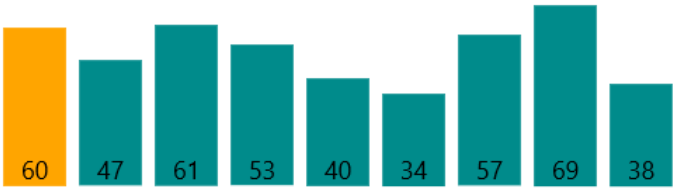

Figure 2: Insertion Sort: sorted and unsorted parts at the beginning

In order to insert element into the right place in the sorted part, we compare each item from the unsorted part with each item from the sorted part in the direction from left to right. Comparing continues until smaller or equal element is found or no elements to compare left. After each comparison, if current item in the sorted part is greater, we move that current item one position right. Finally, when the right position is found, we insert an item into the sorted part.

Complexity of Insertion Sort is $\Theta(n^2)$.

2.2 Selection Sort

Selection Sort algorithm is based on the repeated selection. Here we consider finding minimal key from the unsorted part and swapping it with the first unsorted key. As well as in the Insertion Sort, sorted part grows from the beginning of the sequence.

Assume an array of items to sort. At the beginning of the sorting process unsorted part is represented by the whole array. Then, the first item of the unsorted part is set as the smallest key and is compared with the follow-up elements. When smaller item is found, it is set as a new smallest key. After the end of the array is reached the smallest item is swapped with the first element of the unsorted part and it becomes the sorted part of the array. This step is repeated till the array is sorted.

Complexity of this sorting algorithm is $\Theta(n^2)$.

2.3 Bubble Sort

Bubble Sort is based on the idea of exchanging two adjacent elements if they have wrong order. The algorithm works stepping through all elements from left to right, so the largest elements tend to move or "bubble" to the right. That is why the algorithm is called Bubble Sort.

Now we are going to the details. Let us have an unsorted array. The algorithm does iterations through the unsorted part which is the whole array at the beginning. And with each iteration through the array the range of inspected items is decreased by one till only two elements left. After this two elements are compared and possibly swapped, the array is considered as sorted.

Bubble Sort complexity is $\Theta(n^2)$.

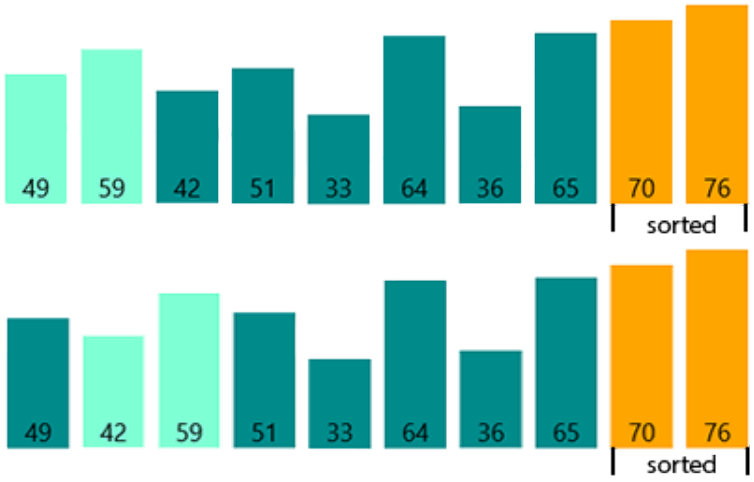

Figure 3: Bubble Sort in progress

2.4 Cocktail Sort

Cocktail Sort or also known as Bidirectional Sort. This algorithm similarly to Bubble Sort uses the idea of exchanging unordered adjacent items of array.

Assume array that needs to be sorted in ascending order. Above we described Bubble Sort and this algorithm has a significant problem. It iterates through array only in one direction. This way, smaller items which are closer to the end of array reach its right positions slowly.

Solution is to make Bubble Sort iterate left-to-right and right-to-left. Cocktail Sort uses two cycles inside a big one:

1. Iterate from a to b, compare adjacent elements and swap if they are not ordered.

2. Iterate from b to $a + 1$ same way as in the step 1

3. Repeat steps 1. and 2. but with a bit different range from $a = a + 1$ to $b = b - 1$

Cocktail Sort works better than Bubble Sort in some situations, e.g. array is already sorted. However, number of swaps remains the same. This way, worst case complexity remains $\Theta(n^2)$.

2.5 Quick Sort

Quick Sort works on the principle "divide and conquer". It recursively applies itself on smaller parts of array until it is not sorted.

Algorithm takes one item at unsorted array or its part, usually it is the leftmost or the rightmost element of array. Then this item also known as pivot is moved to its final position in the array that is should occupy. While determining pivot's position, other elements of array are rearranged the way that no bigger elements on the right and no smaller elements are on the left.

This way, it is enough to apply Quick Sort on each part of array not including pivot until array is not sorted.

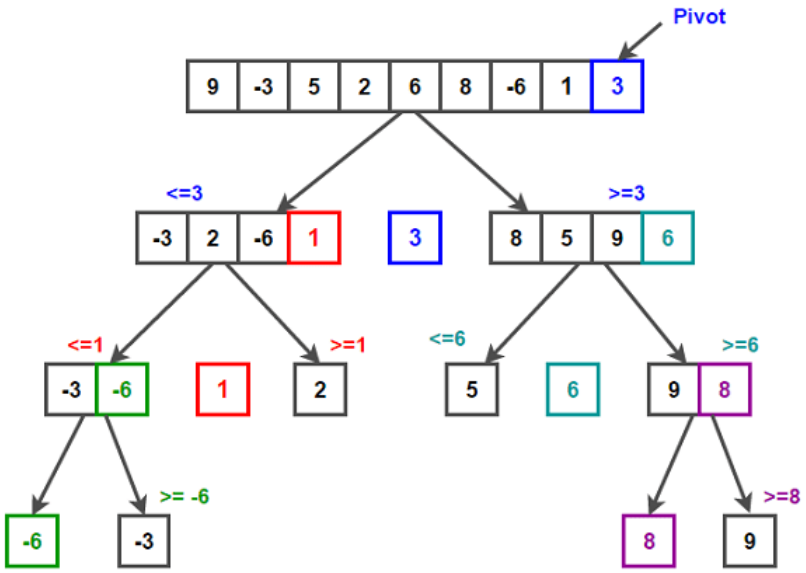

Figure 4: Quick Sort principle

There are several methods of partitioning of array into two parts, here I want to describe one that is demonstrated in the software part of this work.

Firstly, a pivot and index item are selected on the unsorted array or its part. Assume pivot is the last item and index is the first. Next each item of array except pivot is compared with the pivot. If current item is lesser or equal to pivot, it is swapped with the index item, next in order item becomes index. Finally, index and pivot are swapped and this way pivot is on its final position.

Quick Sort is counted as an effective algorithm because its average complexity is $\Theta(n \log n)$. However, when array is maximally unbalanced it may show worst performance. Worst case complexity is $\Theta(n^2)$.

2.6 Merge Sort

Merge Sort as well as Quick Sort is an algorithm of type "divide and conquer". The logic of it is simple: recursively divide data into two parts till it becomes indivisible then merge parts back.

Merge procedure itself takes items from each previously divided part one by one, compares them and moves the smallest to the output, repeats previous step.

Merge Sort complexity is $\Theta(n \log n)$

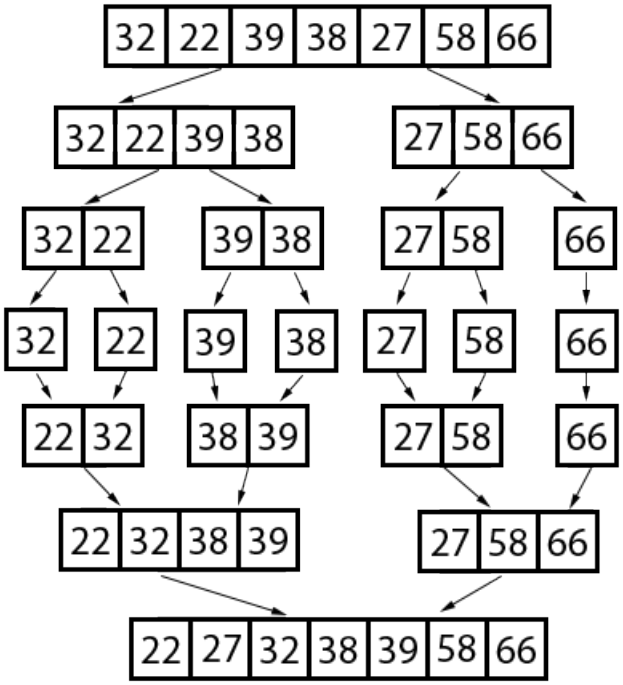

Figure 5: Merge Sort principle

2.7 Heap Sort

Heap Sort is a selection based algorithm and it offers another interesting approach to sorting. In comparison with the Selection Sort it has optimized selection by using binary heap data structure.

Binary heap is a complete binary tree; it means that all levels of tree, except the last one, must be completely filled with nodes. Also, this data structure satisfies the *heap condition*: each node key is greater than or equal to its child keys (this heap type is called *max-heap*).

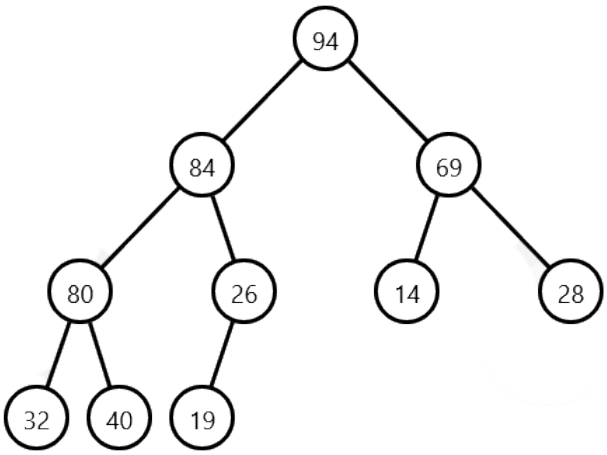

Figure 6: Max-Heap

Binary heap may be implemented by simple array. Item at position zero is a root node, items at position one and two are respectively left and right children of the root. From that representation it is easy to find children of each node (if they exist). Assume a node at position k then its left child is at $2k + 1$ and its right child is at $2k + 2$.

Figure 7: Children positions in heap

Heap Sort itself works as follows:

1. Build max-heap
2. Swap root and the last node, reduce size of heap by one
3. Build max-heap without the node on a reduced position
4. Repeat steps 2 and 3 until the range of array is one

To build *max-heap* from current node we need to assure that right and left child comprise max-heaps. This way, in the first step procedure for building max-heap is recursively applied for each node that has at least one child from bottom to top.

After each swap of the root node and the node at last considered position last node takes its final place. This way it joins the sorted part of array.

Worst and average case complexity of Heap Sort are both $\boldsymbol{\Theta(n \log(n))}$.

2.8 Counting Sort

Counting Sort is usually used for sorting integer keys in the range from 0 to k. Algorithm is based on counting keys of distinct values. Final positions of keys are calculated from the previous computations. It means that the position of some key x depends on the count of keys that are less than x.

For better understanding let us have an example. Assume we have integers in the range from 0 to k and an empty output array. For this range we create helping array that will keep counters for each number from the range. Then all counters from the helping array are set to 0. To obtain final positions for items to be sorted we need to make further computations.

Firstly, for each element from the initial array we increase the respective counter which is determined from the element value. That means value of the element is its counter position in the helping array. Secondly, we need to sum each counter with previous. Finally, for each item from the initial array the respective counter is decreased by one and the value of counter now is the final position, we move item to the output array.

Worst case complexity for Counting Sort is $\Theta(k + n)$, where k is number of items in range and n is size of array to be sorted.

2.9 Radix Sort

The Radix sorting algorithm is based on the idea of using separate digits to determine final number positions. It works with the help of some stable sorting algorithm, it can be for example Counting Sort, taking digits one by one from least significant to most significant.

For better understanding let us go to the details. Assume the array of integers. Let d be the number of digits of the largest item from the array. If there are numbers from the array that have less digits than the d is, then insufficient digits are counted as digits with zero values.

This way, for each digit from least significant to most we sort the input array according to the current digit. Here we use Counting Sort or another stable algorithm.

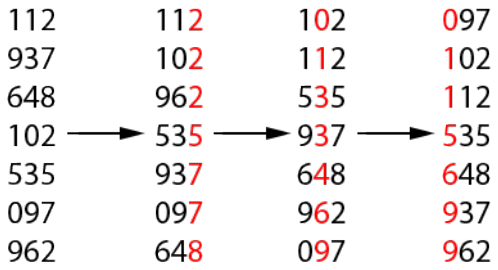

Figure 8: Radix Sort example

2.10 Bucket Sort

Bucket Sort as well as Counting Sort requires to know range of sorted data in advance. It is suitable when data is uniformly distributed over the range.

At the beginning algorithm creates n intervals of same size. For each interval there is created a dynamic structure which will held items from the input. This dynamic structures are called buckets.

Firstly, Bucket Sort distributes data to buckets. Then, there is need to sort each non-empty bucket. This task can be done by using, for example, Insertion Sort. Finally, items from each bucket one by one are moved to the output array.

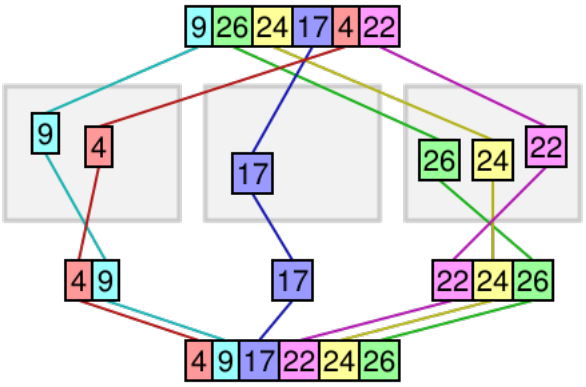

Figure 9: Bucket Sort

3 Documentation

4 User Guide

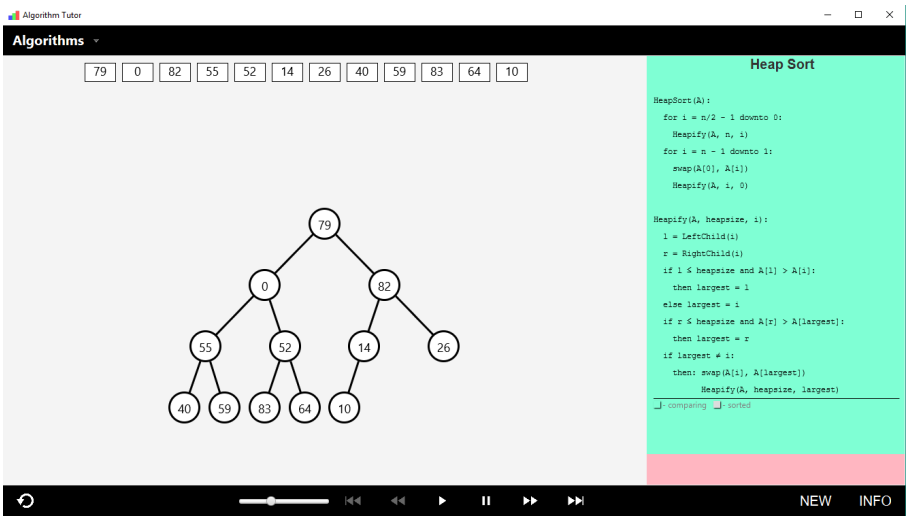

Figure 10: Main window

Conclusions

Závěr práce v "českém" jazyce.

Conclusions

Thesis conclusions in “English”.

A První příloha

Text první přílohy

B Druhá příloha

Text druhé přílohy

C Obsah přiloženého CD/DVD

Na samotném konci textu práce je uveden stručný popis obsahu přiloženého CD/DVD, tj. jeho závazné adresářové struktury, důležitých souborů apod.

bin/
: Instalátor INSTALATOR programu, popř. program PROGRAM, spustitelné přímo z CD/DVD. / Kompletní adresářová struktura webové aplikace WEBOVKA (v ZIP archivu) pro zkopírování na webový server. Adresář obsahuje i všechny runtime knihovny a další soubory potřebné pro bezproblémový běh instalátoru a programu z CD/DVD / pro bezproblémový provoz webové aplikace na webovém serveru.

doc/
: Text práce ve formátu PDF, vytvořený s použitím závazného stylu KI PřF UP v Olomouci pro závěrečné práce, včetně všech příloh, a všechny soubory potřebné pro bezproblémové vygenerování PDF dokumentu textu (v ZIP archivu), tj. zdrojový text textu, vložené obrázky, apod.

src/
: Kompletní zdrojové texty programu PROGRAM / webové aplikace WEBOVKA se všemi potřebnými (příp. převzatými) zdrojovými texty, knihovnami a dalšími soubory potřebnými pro bezproblémové vytvoření spustitelných verzí programu / adresářové struktury pro zkopírování na webový server.

readme.txt
: Instrukce pro instalaci a spuštění programu PROGRAM, včetně všech požadavků pro jeho bezproblémový provoz. / Instrukce pro nasazení webové aplikace WEBOVKA na webový server, včetně všech požadavků pro její bezproblémový provoz, a webová adresa, na které je aplikace nasazena pro účel testování při tvorbě posudků práce a pro účel obhajoby práce.

Navíc CD/DVD obsahuje:

data/
: Ukázková a testovací data použitá v práci a pro potřeby testování práce při tvorbě posudků a obhajoby práce.

`install/`
: Instalátory aplikací, runtime knihoven a jiných souborů potřebných pro provoz programu PROGRAM / webové aplikace WEBOVKA, které nejsou standardní součástí operačního systému určeného pro běh programu / provoz webové aplikace.

`literature/`
: Vybrané položky bibliografie, příp. jiná užitečná literatura vztahující se k práci.

U veškerých cizích převzatých materiálů obsažených na CD/DVD jejich zahrnutí dovolují podmínky pro jejich šíření nebo přiložený souhlas držitele copyrightu. Pro všechny použité (a citované) materiály, u kterých toto není splněno a nejsou tak obsaženy na CD/DVD, je uveden jejich zdroj (např. webová adresa) v bibliografii nebo textu práce nebo v souboru `readme.txt`.

References

[1] CORMEN, T. H.; LEISERSON, C. E.; RIVEST, D. L.; STEIN, C. *Introduction to algorithms*. Second Edition. 2001. ISBN 0-262-03293-7.

[2] KNUTH, D. *The Art of Computer Programming: Fundamental Algorithms*. Third Edition. 2004. ISBN 0-201-89683-4.

[3] SIPSER, M. *Introduction to the Theory of Computation*. Boston, MA: PWS Publishing Company, 1997. ISBN 0-534-94728-X.

[4] KNUTH, D. *The Art of Computer Programming: Sorting and Searching*. Second Edition. 2004. ISBN 0-201-89685-0.

[5] SEDGEWIK, R. *Algoritmy v C: základy, datové struktury, třídění, vyhledávání*. 2003.

[6] BĚLOHLÁVEK, R. *Algoritmická matematika 1*. Available also from: ⟨http://belohlavek.inf.upol matematika-1-1.pdf⟩.

[7] *Wikipedia*. Available from: ⟨https://www.wikipedia.org/⟩.

[8] *stackoverflow*. Available from: ⟨https://stackoverflow.com/⟩.

[9] *GeeksforGeeks*. Available from: ⟨https://www.geeksforgeeks.org/⟩.

[10] *Java documentation*. Available from: ⟨https://docs.oracle.com/javase/8/⟩.