PART 1 – To be completed by the student		
Student Name	Peter Kurtosi	
Student ID Number	LON01111239	
Module Name	Data Structures and Algorithms	
Course	Year 2, Semester B	
Assignment Title	Implementing Data Structures and Algorithms	
Module Lecturer	Rafiqul Islam	
Number of Words	5,977	
Assignment Due Date	06th May 2014	
Submission Date	06th May 2014	

Table of Contents

Introduction

This assignment is about data structures and algorithms as used in computer programming. Data structures are ways in which data is arranged in your computer's memory (or stored on disk). Algorithms are the procedures a software program uses to manipulate the data in these structures.

Almost every computer program, even a simple one, uses data structures and algorithms. For programs that handle even moderately large amounts of data, or that solve problems that are slightly out of the ordinary, more sophisticated techniques are necessary. Simply knowing the syntax of a computer language such as Java or C++ isn't enough.

This assignment is about what we need to know after we have learned a programming language.

Task 1: Algorithms

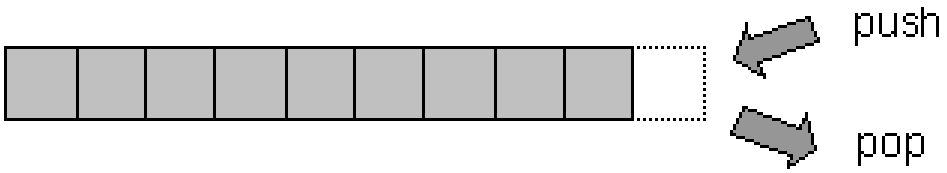
Task 1.1: Stack and Queue

In this section, two Abstract Data Types (ADTs), which are very close to each other, will be presented. These are Stack and Queue. Both of them are containers made for such objects that have two basic operations:

- insert a new item;
- remove an item.

The difference between stack and queue is in the rule of removing ONE element, according to the followings (Sedgewick & Wayne, 2011):

- Stack - LIFO (last in - first out) - Picture 1



Picture 1

- Queue - FIFO (first in - first out). - Picture 2

Picture 2

STACK:

With the help of the applied marking in Picture 3, the Pseudo-code of the operations that can be carried out on the Stack can be described. (In the case of Array-Based Stack N= maximum size of stack (array S), t = the index of the top cell)

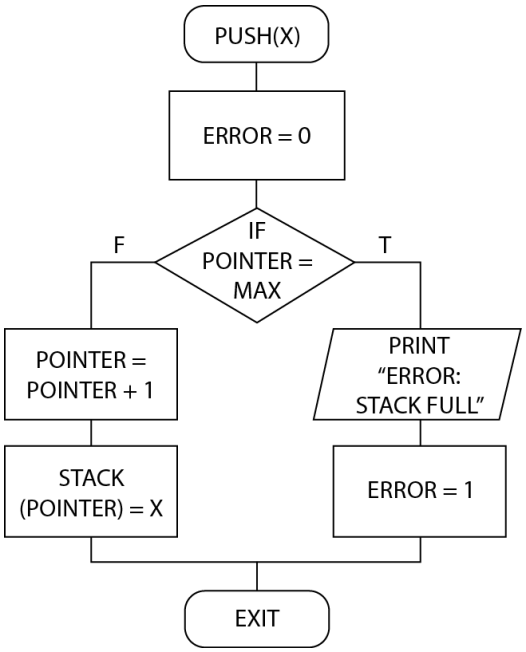
Picture 3

The two main methods of the Stack:

- push(o) - Inserts the object o -> onto the top of the stack.

```
Algorithm push(o):      // add an item to the stack
    if size() = N then  // the stack is MAX or not?
        throw a StackFullException // exception handling
    e <- S[t]           // the "t"th item -> e
    t <- t+1            // increase the size of the stack
    S[t] <- o           // the "t"th item <- o
```

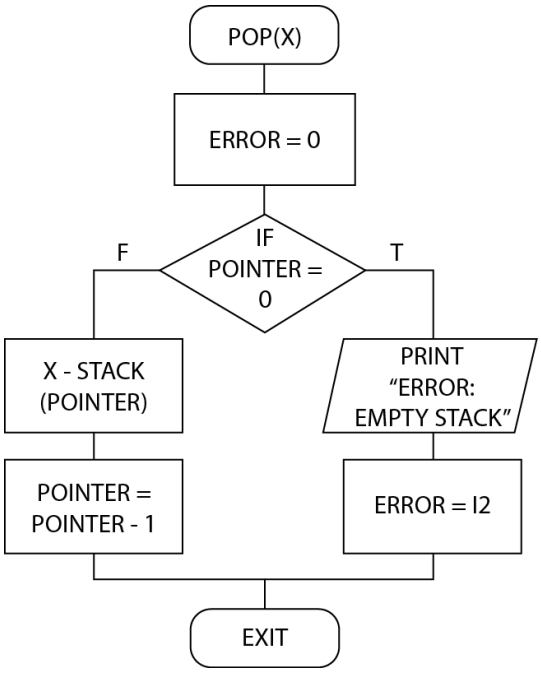
And the flow chart of push method:



- pop() – It removes the topmost element of the stack (which was inserted by the push() as the last one) and gives it back as a return value. If the stack is empty, an error occurs that must be dealt with.

```
Algorithm pop():    // remove an item from the stack
    if isEmpty() then  // the stack is empty or not?
        throw a StackEmptyException // exception handling
    e <- S[t]       // the "t"th item -> e
    S[t] <- null    // delete the "t"th item in the stack
    t <- t-1        // decrease the size of the stack
    return e        // return with the removed item
```

And the flow chart of pop method:



The support methods that facilitate the use of Stack:

- size() – The return value of the method is the number of objects in the stack.

```
Algorithm size():
    return t+1
```

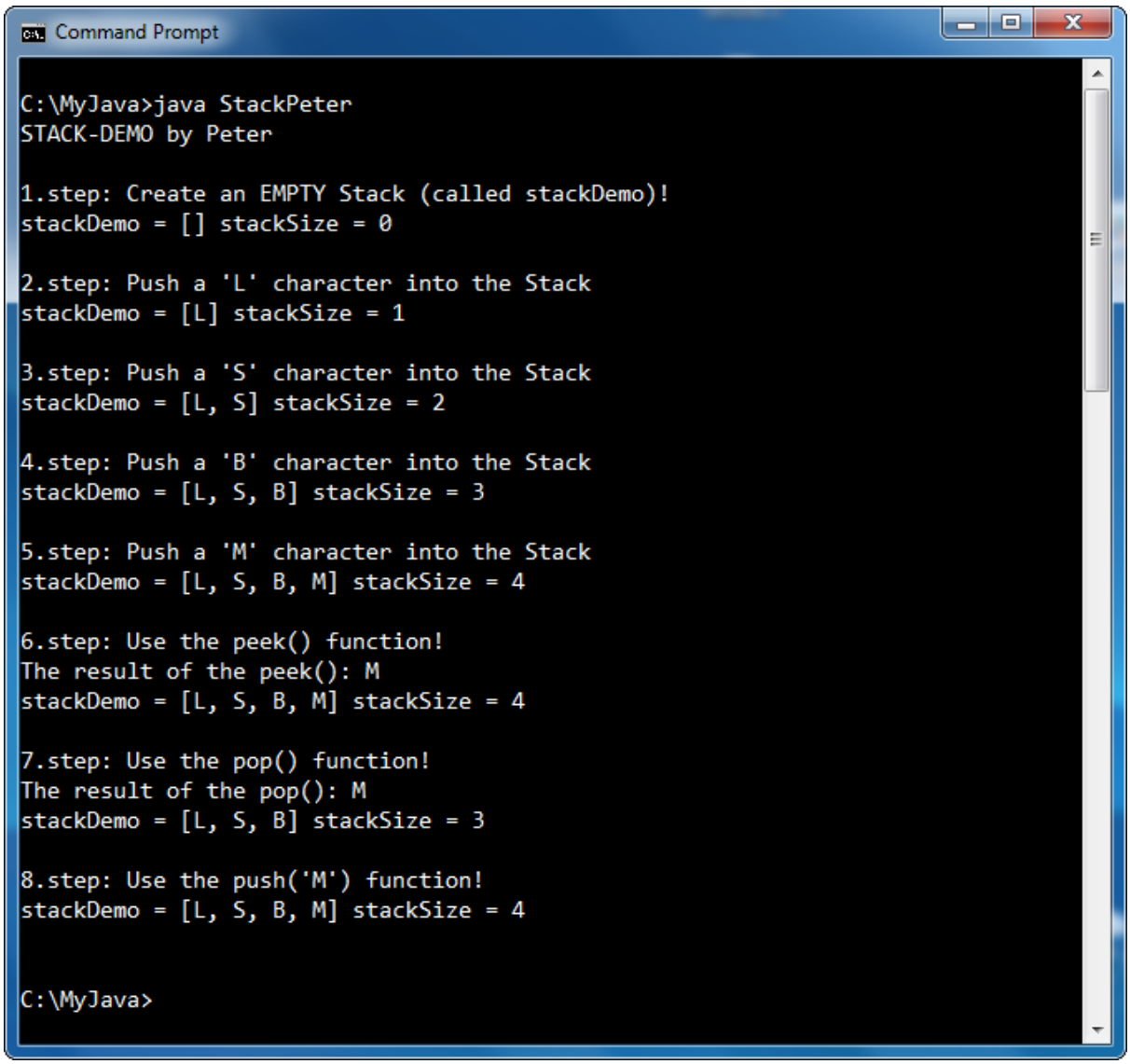
- isEmpty() – It checks whether the stack is empty. Its return value is a Boolean type variable that takes the value TRUE in the case of an empty stack.

```
Algorithm isEmpty():
    return (t<0)
```

- top() - peek() – Its return value is the top object of the stack. Does not remove the object just returns it. In case of an empty stack, error occurs that must be dealt with.

```
Algorithm top():
    if isEmpty() then
        throw a StackEmptyException
    return S[t]                          //return with the top item
```

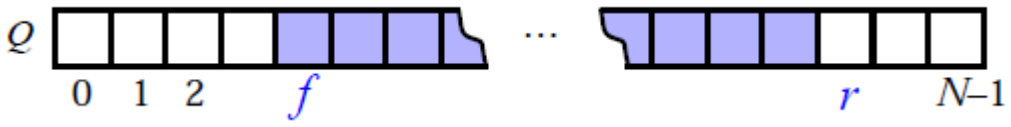
I have created a small sample program to demonstrate the operation of the Stack (Picture 4). The java source code of the program can be found in Appendix A.



Picture 4

QUEUE:

With the help of the marks used in Picture 5 the Pseudo code of the operations that can be carried out on Queue can be described. (In the case of an Array-Based Queue, N= maximum size of queue, f= the index of the front cell, r= the index of the cell after the last)



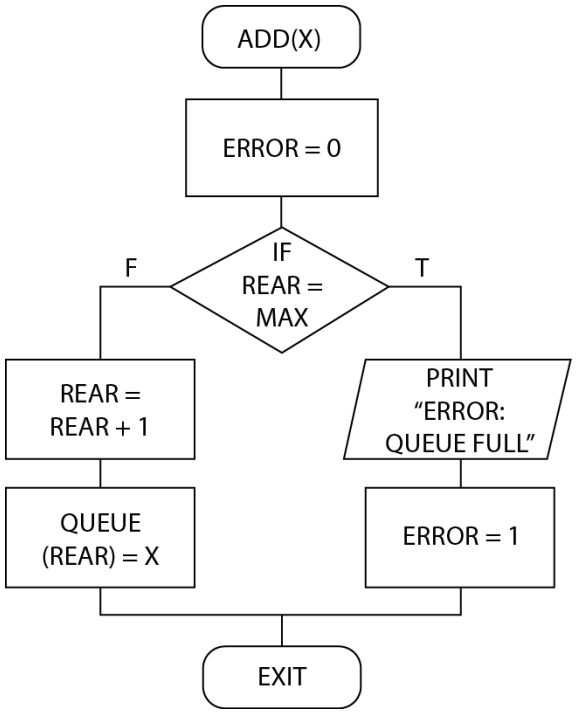
Picture 5

The two main methods of the Queue:

- add(o) - Inserts the object o -> onto the end of the Queue.

```
Algorithm add(o):       // add an item to the queue
    if size() = N-1 then  // the queue size is N-1 or not?
        throw a QueneFullException // exception handling
    Q[r]   <- o         // the "r"th item <- o
    r <- (r+1) mod N    // the value of new r
```

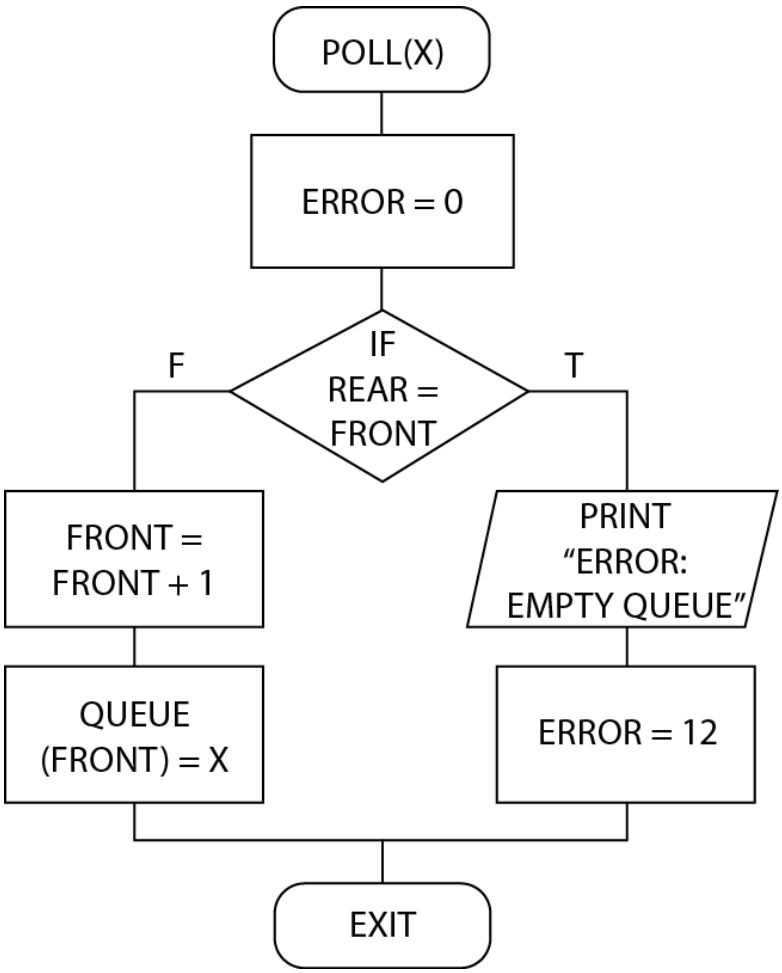
And the flow chart of add method:



- poll() – Removes the first element of the queue (which was inserted by the add() as the first one) and gives it back as a return value. If queue is empty, an error occurs that has to be dealt with.

```
Algorithm poll(o):      // remove an item from the queue
    if isEmpty() then   // the queue is empty or not?
        throw a QueneEmptyException // exception handling
    temp <- Q[f]        // the "front"th item -> temp
    Q[f] <-null         // delete the value
    f <- (f+1) mod N    // the new front is increased
    return temp         // return with the removed item
```

And the flow chart of poll method:



The support methods that help the use of Queue:

- size() – The return value of the method is the number of objects in a queue.

```
Algorithm size():
    return (N - f + r) mod N
```

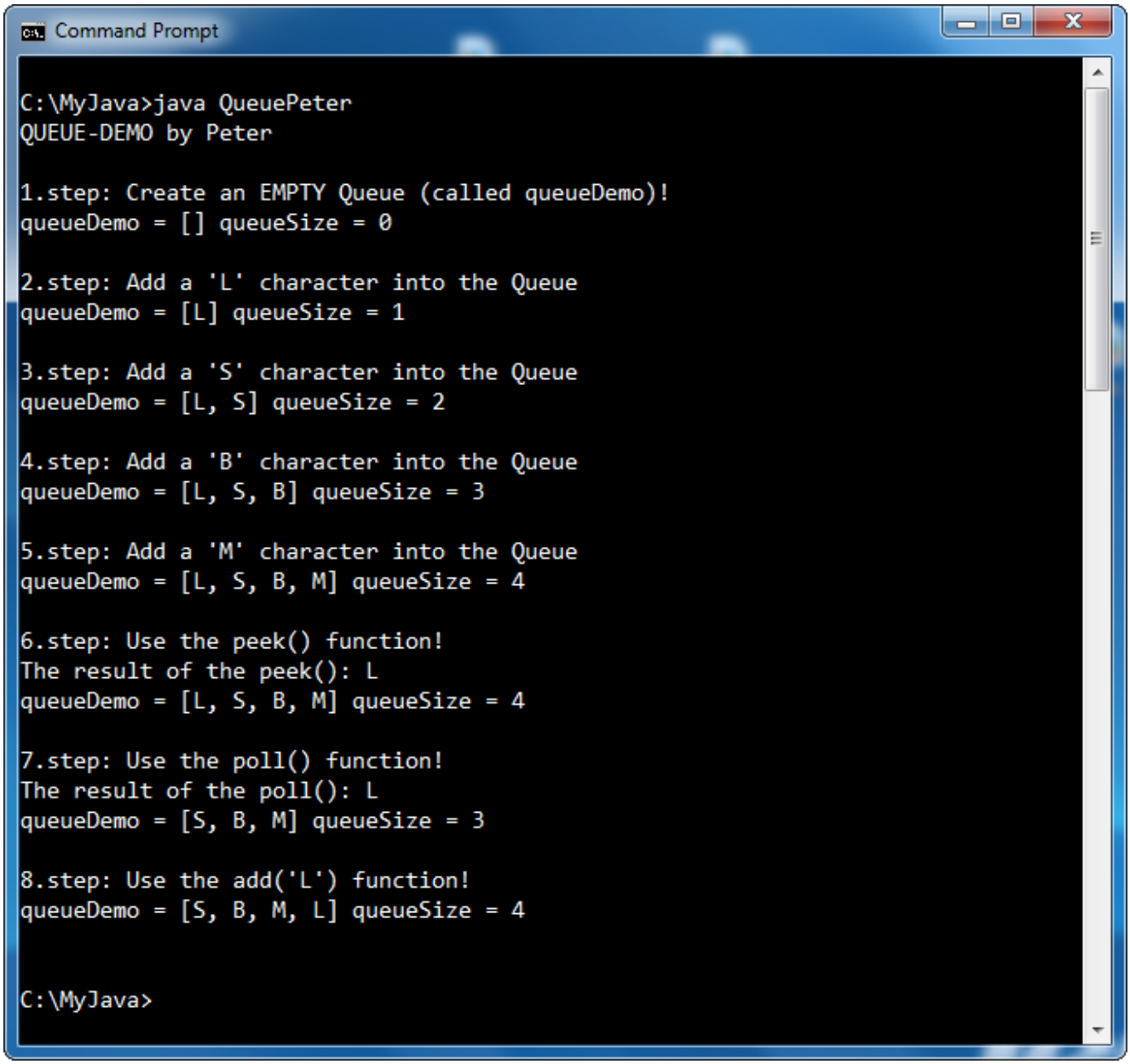
- isEmpty() – It checks whether the queue is empty. Its return value is a Boolean type variable, which takes the value TRUE if the queue is empty.

```
Algorithm isEmpty():
    return (f = r)
```

- front() - peek() – Its return value is the first object of the queue. It does not remove the object just returns it. In case of an empty queue an error occurs that has to be dealt with.

```
Algorithm front():
    if isEmpty() then
        throw a QueueEmptyException
    return Q[f]                          // return with the front item
```

I have made a small sample program to illustrate the operation of Queue (Picture 6). The java source code of the program can be found in Appendix B.



Picture 6

Critically review: Java language has its own implementation for handling stacks and queues. In my opinion, these implementations keep the practicality in mind and not fullness. That is why in cases where unique solutions are needed it is simpler and more expedient to make our own Stack or Queue Class.

Task 1.2: The Binary Search Algorithm

Definition of Binary Tree: That tree is called binary tree, in which the pointer points from one element to just two other elements.
Recursive definition: The Binary Tree is either an empty tree or such a tree where its left and right pointers each points to a binary tree (Puntambekar, 2008).
Binary Search Tree: The Binary Search Tree is a Binary Tree that has the following attributes:

- Let x be an arbitrary node of the Binary Search Tree.
- If y is in the left side subtree of x, then value[y] <= value[x].
- If y is in the right side subtree of x, then value[x] <= value[y].

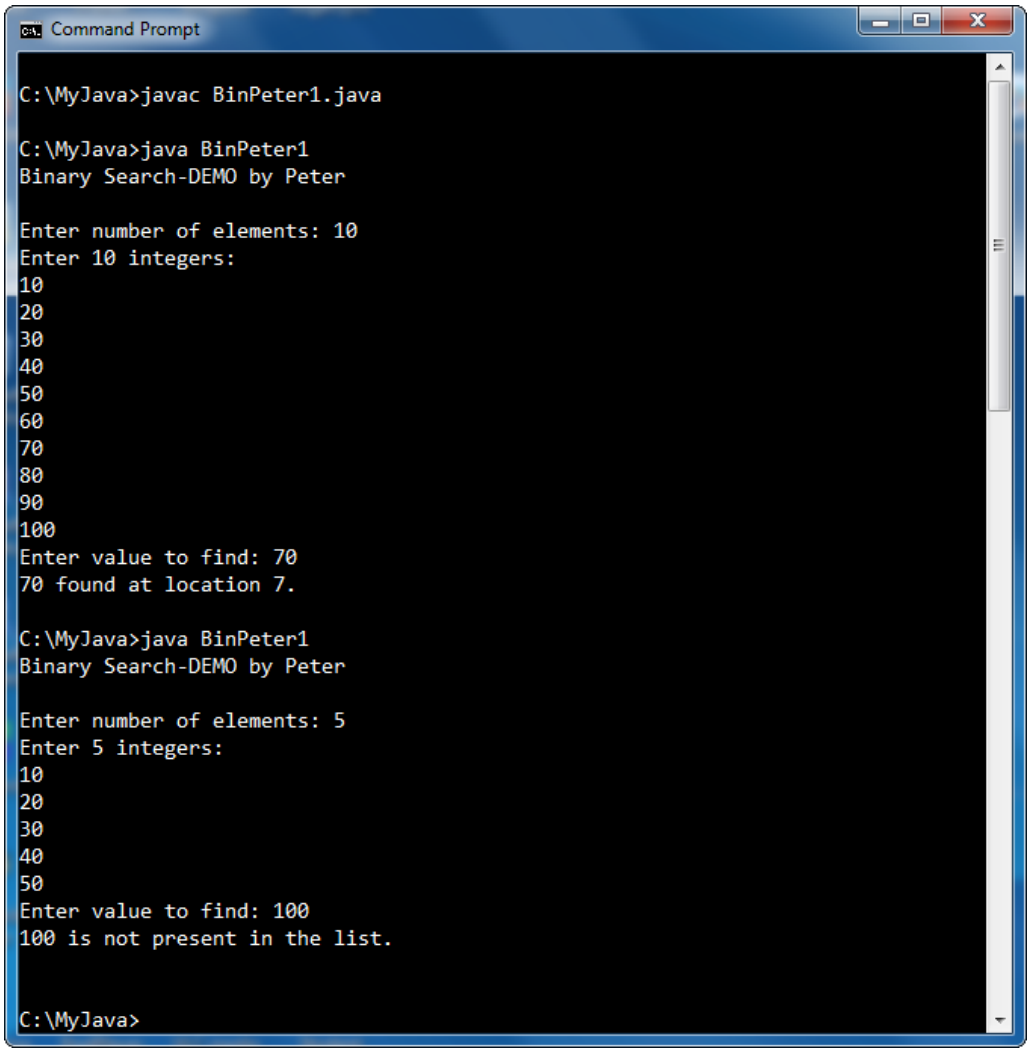
It follows from the above definition that the Binary Search Algorithm searches in a sorted data set. In addition to that, it is a necessary condition that the elements should be directly accessible, as the algorithm is jumping not stepping between the elements of the data set.

The operation of Binary Search Algorithm: Binary Search Algorithm is typically a “divide and conquer” type method. By finding the central element it divides the data set into two parts and checks whether this is the searched element. If it is, the search is successfully over. If it is not then the searched element is greater or less than the central element. This way it can be determined in which set is the searched element. So half of the elements has already been filtered, the searched element is surely not there. Repeating this procedure, the data set is constantly divided into half, until the searched element is found.

In practice, it is done by declaring a lower (l) and higher (h) index and the values of these indexes must be changed according to the above described rule. The central index will be the middle (m). Naturally, in the first step l=1 and h=n (where n is the number of nodes). In this case the central element is: m = (n-l)/2.

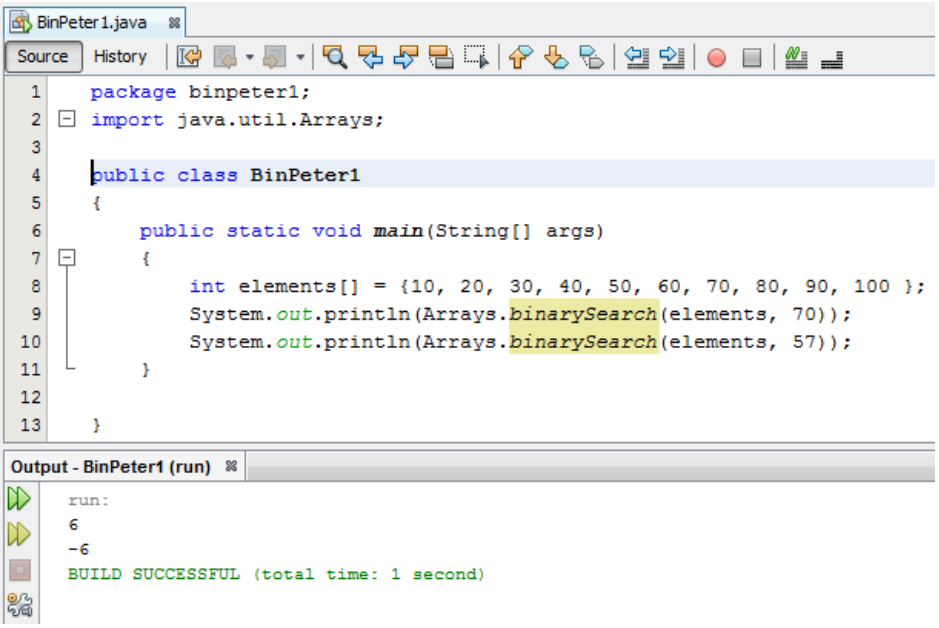
If the searched element is less than the central element, it receives the h = m-1 index. If it is greater, than the l = m+1 index.
The search is unsuccessful if the number of elements in the final set is 1 but this element does not match either with the searched one. Which means that l>h.

Critically review: I have made a small sample program to illustrate the operation of Binary Search Algorithm (Picture7).The java source code of the program can be found in Appendix C.



Picture 7

Java language contains a built-in implementation for Binary Search (Picture 8).



Picture 8

It can be seen that the **Arrays.binarySearch()** function returns with the array index of the found element in the case of a successful search and with a negative array index if it did not find the given element. In the **java.util.Arrays**, Picture 9 demonstrates the code of the standard binary search.

```
1:     public static int binarySearch(int[] a, int key) {
2:         int low = 0;
3:         int high = a.length - 1;
4:
5:         while (low <= high) {
6:             int mid = (low + high) / 2;
7:             int midVal = a[mid];
8:
9:             if (midVal < key)
10:                 low = mid + 1
11:             else if (midVal > key)
12:                 high = mid - 1;
13:             else
14:                 return mid; // key found
15:         }
16:         return -(low + 1);  // key not found.
17:     }
```

Picture 9

The problem is in the 6th row.

```
6:              int mid =(low + high) / 2;
```

The problem appears when the sum of **low** and **high** is greater than the maximum positive int value (2^{31}-1). In this case the sum overflows, takes a negative value that remains even after dividing by two.

Nowadays the processing of such a huge amount of data is required that it is easy to run into this bug. To avoid this bug, it is worth to rewrite the 6th row to the following:

```
int mid = low + ( ( high - low ) / 2 );
```

The performance of Binary Search Algorithm: The efficiency of the operation that can be performed by the binary search algorithm depends on the height of binary tree. And the height of the tree depends on how balanced the tree is. To find a randomly chosen element (which is in the data set) in a full binary tree that contains n number of nodes, in the **worst case** requires **$O(\log_2 n)$** number of operations. The number of unsuccessful steps is always $O(\log_2 n)$. Naturally, the **best case** is to find it at the first time: **O(1)**.

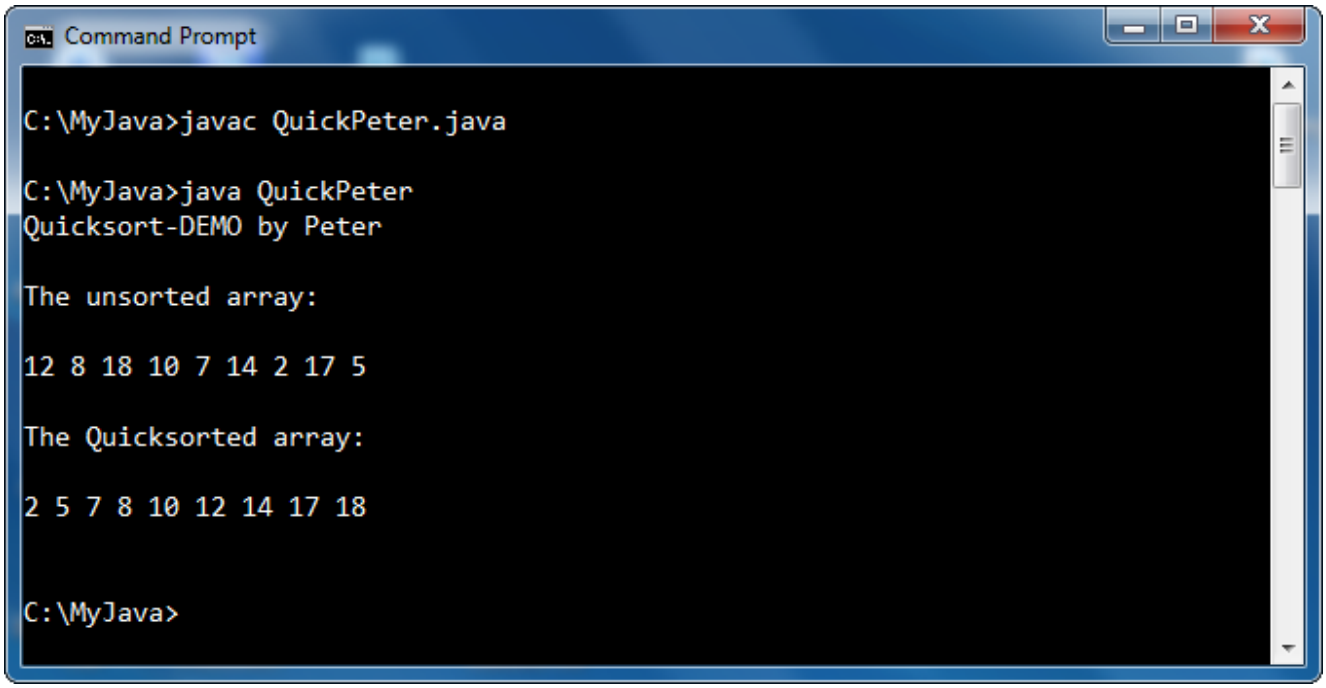
The **average case** is **$O(½ \log_2 n)$.**

In the case of random insertions the tree can “stretch”, deteriorating the efficiency of the search. If the tree has only one long branch (worst case), then the running time of the operations will be O(n). The AVL Tree (Adelson-Velskii and Landis' Tree) offers a solution to this problem.

Task 1.3: The Quicksort Algorithm

The operation of Quicksort Algorithm: The Quicksort algorithm also operates based on the “divide and conquer” principle. It divides the original array into two parts. One of the part-arrays is not bigger than a pivot element, while the other one contains the bigger elements. After sorting the two part-arrays and by stitching them together the original array is returned, sorted. Naturally, the sorting of the part-arrays must be solved as well, which is implemented by another dividing into 2-2 part-arrays. This partitioning to smaller arrays can be continued as long as we get part arrays with one element. The efficiency of the Quicksort depends on the goodness of the partitioning, namely the selection of the pivot element.

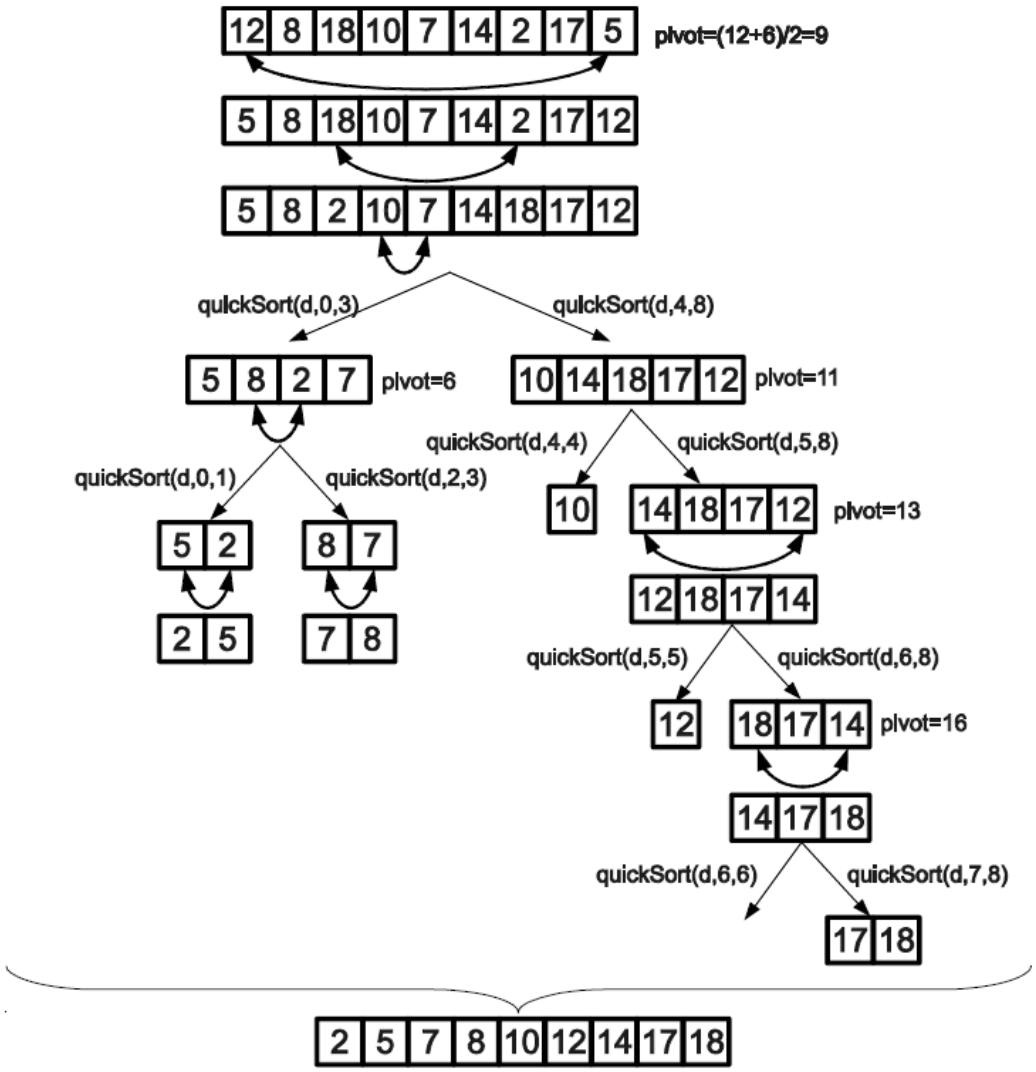
I have made a small sample program to illustrate the operation of Quicksort Algorithm (Picture 10). The program carries out the quick sorting of the program illustrated on Picture 11. The java source code of the program can be found in Appendix D.



Picture 10

A pivot value is good if the resulting part-arrays have nearly the same size. Usually, the best result is achieved if more array element averages of the pivot value are selected. The partitioning of the array, after the determination of the pivot, continues with the swapping of the elements. For this an index has to be started in the array both from left and from right. The right side index has to be decreased until we find a value in the array, which is less than the pivot or until we reach the left side index. The left side index is increased until we find a greater value than the pivot or we reach the other index. We change the array elements belonging to the index. The array has to be divided into two part-arrays based on the indexes.

Picture 11 illustrates the operation of Quicksort with the help of an example.



Picture 11

The performance of QuickSort Algorithm: The Quicksort is such a sorting algorithm the running time of which is **$O(n^2)$** in the **worst case**. (The worst case occurs if, after choosing the pivot element, the size of one of the part-arrays, during the running of the algorithm, is constantly 1.) The Quicksort keeps picking the worst pivot element and at each recursion, one sub array is empty.

$$T(0) = T(1)=O(1)$$
$$T(n) = T(n-1) + O(n) = T(n-2) + O(n-1) + O(n) = T(0) + O(1) + \dots + O(n) = O(n^2)$$

The running time in the **best case** is **$O(n \log_2 n)$**, Quicksort always chooses best pivot element and at each recursion keeps splitting sub arrays in half.

$$T(0) = T(1) = O(1) \text{ and } T(n) = 2T(n/2) + O(n)$$
$$T(n) = O(n \log_2 n)$$

The Quicksort is an interesting algorithm from point of view of average case analysis. The **average running time** is approximately equal with **O(n $\log_2$ n)**.

$$T(n) = \begin{cases} O(1) \text{ if } n \leq 1 \\ \frac{1}{n}\sum_{i=1}^{n}(T(i-1) + T(n-i) + O(n)) \end{cases}$$

The general case is further divided into cases, where the pivot is the ith smallest element in the array. The general case is:

$$T(n) = \frac{1}{n}\sum_{i=1}^{n}\big(T(i-1) + T(n-i) + O(n)\big) = \frac{2}{n}\sum_{i=0}^{n-1} T(i) + O(n)$$

For large enough n, it holds that

$$n\,T(n) - (n-1)\,T(n-1) = 2T(n-1) + O(n)$$

$$T(n) = \frac{n+1}{n}T(n-1) + O(1) = \frac{n+1}{n-1}T(n-2) + O\left(1 + \frac{n+1}{n}\right) =$$

$$\frac{n+1}{2}T(1) + O\left((n+1)\sum_{i=3}^{n+1}\frac{1}{i}\right) = O(n \log_2 n)$$

Task 1.4: The Recursive Algorithm

Definition of Recursion: Recursive is the procedure or the function, which directly or indirectly refers to itself. Recursion always consists of two parts:

1. static part: e.g. 1! = 1 - its purpose is to ensure the stopping of the recursion.
2. dynamic part: e.g. n! = n * (n-1)! - its purpose is to ensure further progress.

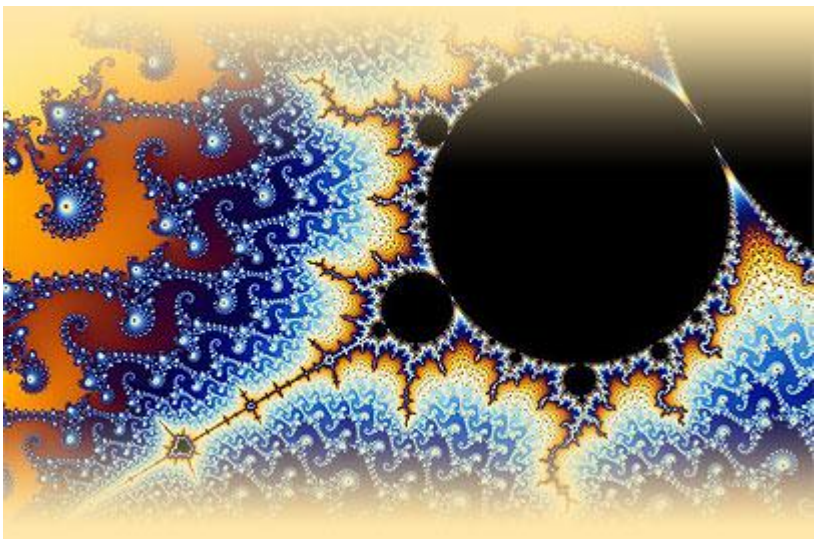
In other words: the definition of recursion consists of two parts:

1. Determination of the basic cases.
2. Defining of the reduction rules. With the help of the reduction rule the problem is led back to the solution of a smaller problem. When defining the reduction rule the concept to be defined is used as well (Neapolitan & Naimipour, 2004).

Sample: Recursive definition of n!:

1. Basic case:	n! = 1	ha	n = 1
2. Reduction rule:	n! = n * (n - 1)!	ha	n > 1

The power of the recursion is that is becomes possible to define infinite number of things with finite number of statements. The same way, infinite number of calculation steps can be expressed with one single finite recursive program, without including an explicit cycle.
The use of recursive algorithms is justified primarily where the problem to be solved is. The function to be calculated is recursively defined from the beginning. The function (or subroutine) is the natural tool of the use of recursion. There is direct recursion, when the recursive function calls itself. But there is indirect recursion as well, when e.g. f1() function calls an f2() function and the f2() function refers back to the f1() function.
The **advantage** of the Recursive Algorithm is that it is rather simple. Although, it is proven that every recursive algorithm is feasible without recursion (many times with the use of less memory and time), a shorter and more transparent solution can be made with recursion.
The **disadvantage** of the Recursive Algorithm is that the badly used recursive subroutines can fill up the stack, which can cause "stack overflow" runtime error.
In the case of recursive algorithms it is important to mention one of their most spectacular uses, the fractals. Picture 12 illustrates a recursive Mandelbrot fractal curve. (Mandelbrot, 2002)



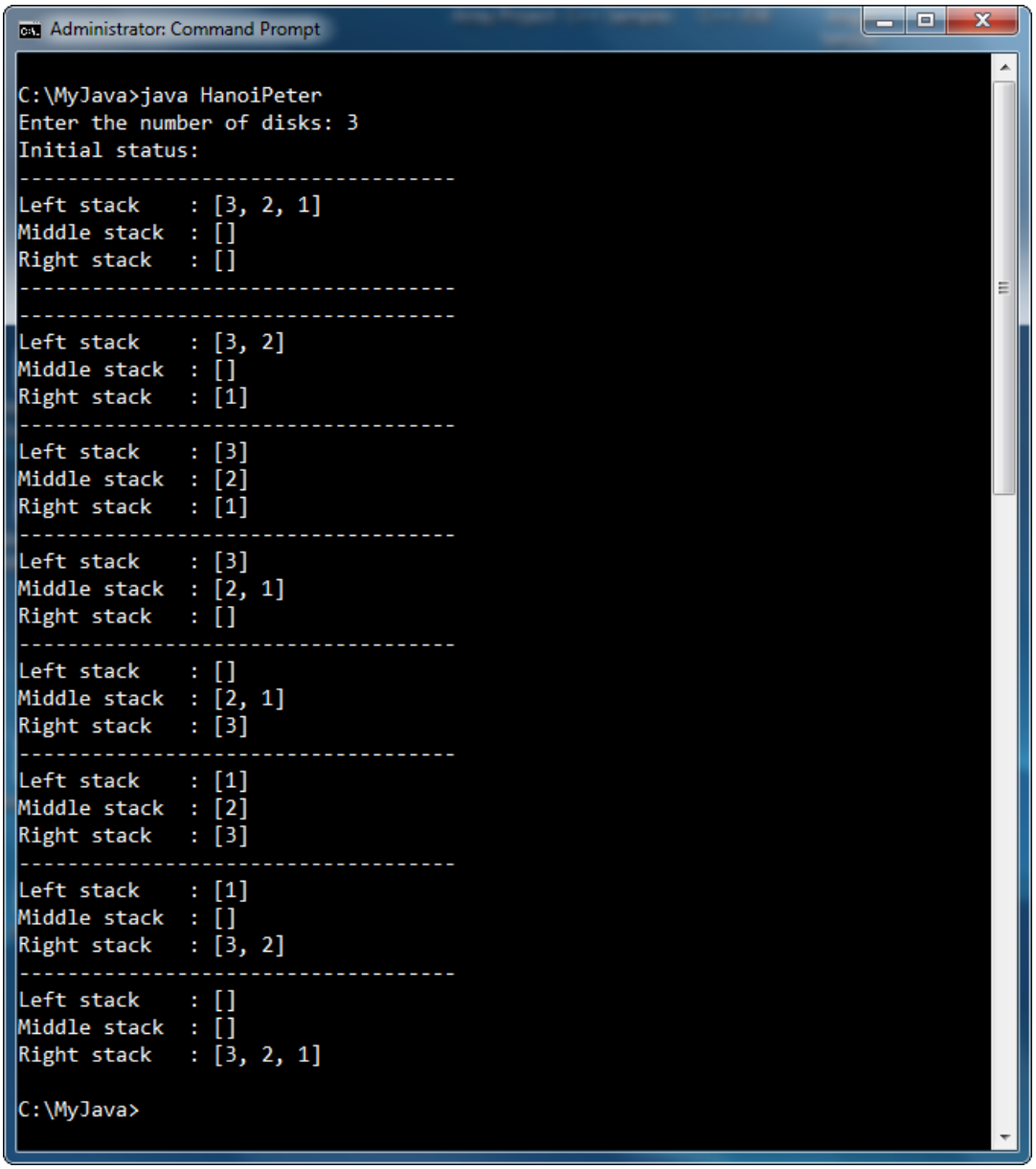
Picture 12

Examples for the use of recursive algorithms:

The Binary Search Algorithm, presented in Task 1.2, could also be solved with recursive algorithm. However, the code would be so complex that my sequential solution, which can be found in Appendix C, is more favourable in all aspects.

A great example for the use of the recursive algorithm is the Quicksort algorithm, presented in Task 1.3 ban. Based on my program that can be seen in Appendix D, it can be seen that the quickSort procedure calls itself recursively and this way it ensures the adequate number of implementation of the task that should be performed. Special feature of the procedure, that the quickSort function receives the changed parameters, which are necessary for its own recursion, from another function as a result.

To present the specific use of the recursive algorithm, I have created the famous **Tower of Hanoi** program. (Appendix E contains the Power Point presentation.) The screenshot of the operation of the program can be seen on Picture 13.



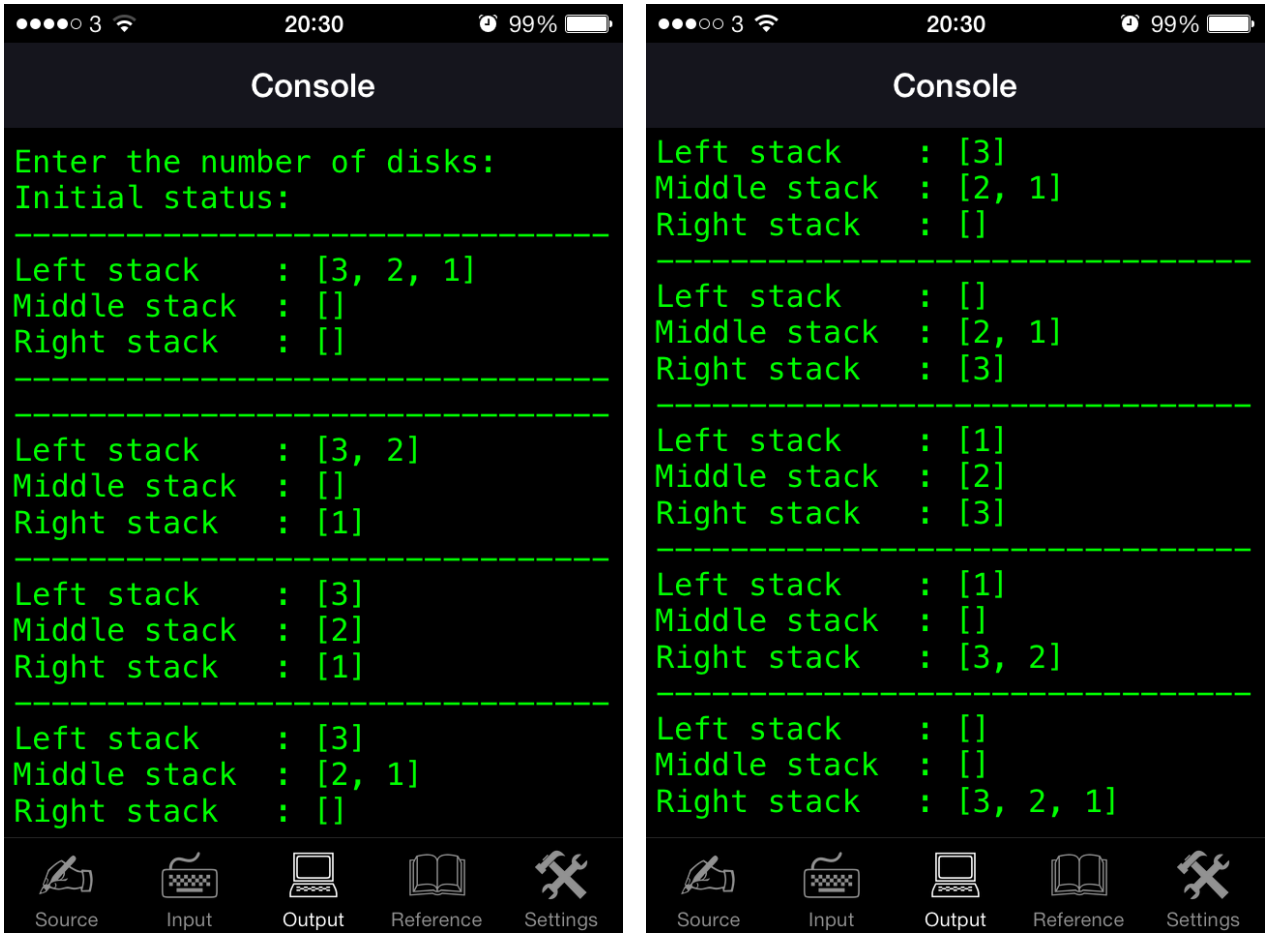
Picture 13

The complete code of the program can be found in Appendix F. However, the essence of the procedure is solved by the recursion of the small four-line function seen on Picture 14.

```
static void recurHanoi(int disknr, String stringLeft, String stringRight, String stringMiddle, Stack stL, Stack stM, Stack stR)
{
    if (disknr >0)
    {
        recurHanoi(disknr-1, stringLeft, stringMiddle, stringRight, stL, stM, stR);
        stMove(stringLeft, stringRight, stL, stM, stR);
        recurHanoi(disknr-1, stringMiddle, stringRight, stringLeft, stL, stM, stR);
    }
}
```

Picture 14

I have made the Tower of Hanoi program for iPhone (Ray, 2011) as well (Picture 15).



Picture 15

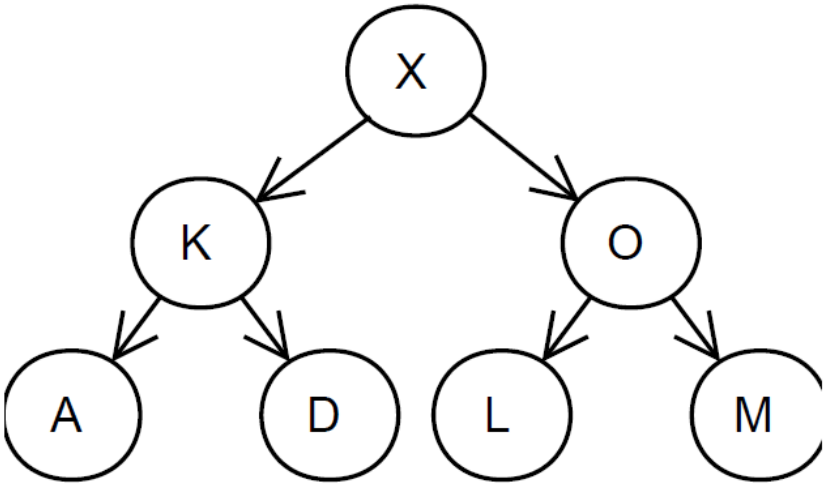
Task 2: Heap sort

In this chapter the introduction of the Heap Sort is my task. The assignment brief also mentions the **Insertion Sort**, which has the advantage to sort locally, although its running time is not too favourable and the **Merge Sort** which has a running time of O(n * log n) for n element but does not sort locally.
The **Heap Sort** combines the good properties of this two sorting methods as it sorts locally. Apart from the array of the input data, it needs only a few (constant number) variables and it has a running time of O(n * log n).

Definition of Heap property: An almost complete binary tree has heap property if it is empty or the value in its root is bigger than the value in both of its children and both of its sub-trees have heap property (Picture 16)

A binary tree is complete if its height is h and it has 2^h-1 nodes. A binary tree with a height of h is **almost complete**, if:

- it is empty;
- its height is h and the height of its left sub-tree is h-1 and it is almost complete and the height of its right sub-tree is h-2 and it is complete;
- it has a height of h, the height of its left sub-tree is h-1 and it is complete and the height of its right sub-tree is h-1 and it is almost complete.



Picture 16

The complete code of the HeapPeter program can be found in Appendix G.

Task 2.1: Heap Sort implementation in Java

Before starting the Heap sort it must be ensured that the input array heap, which consists of unsorted data, should have heap property, as Heap sort can be implemented only on an array that has heap property.
The doHeap() procedure is important for the handling of the heaps (Picture 17). Its input data are the heapArray[] and one if its “i” indexes.

```
public static void doHeap(int heapArray[], int i) //This function rearrange the order of the members
{
    leftSide=2*i;      // calculation of the left side
    rightSide=2*i+1;   // calculation of the right side
    if(leftSide <= heapSize && heapArray[leftSide] > heapArray[i])
        peak=leftSide; // the value of peak -> left
    else
        peak=i;        // the value of peak -> i
    if(rightSide <= heapSize && heapArray[rightSide] > heapArray[peak])
        peak=rightSide;// the value of peak -> right
    if(peak!=i)
    {
        changePlace(i,peak);     // i and peak change place in the array
        doHeap(heapArray, peak); // recursive calling of the doHeap
    }
}
```

Picture 17

When the doHeap() is called we assume that the sub-trees with the leftSide(i) and the rightSide(i) roots has heap structure but the heapArray[i] can be less than its children and this way can breach the heap property. The task of doHeap() is to “move down” the heapArray[i] value so that the sub-tree with the “i” root become a heap.

At every step, it determines the greatest of the heapArray[i], heapArray[leftSide(i)] and heapArray[rightSide(i)] and stores its index into the “peak” variable.
If any two examined value breach the heap property, then the changePlace() procedure exchanges them (Picture 18).

```
public static void changePlace(int i, int j) // This function is changing the values of necessary members
{
    int t=heapArray[i];        // temporary variable for ith element
    heapArray[i]=heapArray[j]; // value of "j"th -> "i"th place
    heapArray[j]=t;            // the original value of "i"th -> "j"th place
}
```

Picture 18

The recursive call of the doHeap() function and the call of the changePlace() at a necessary place, continues until the recursive call cannot find anything that could be changed in the data structure, which means peak=i.

In other words, if heapArray[i] is the greatest than the sub-tree with the “i” root already has heap property and the procedure has ended.
The doHeap() procedure can be used to form the heapArray[1...n] array into a heap bottom-up. (n=heapArray.lenght)

As the elements of the heapArray[] and the heapArray[((n/2)+1)...n] are leaves, they can be considered as single element heaps.
So the heapBuild() procedure only has to run through the other peaks and to execute the doHeap() procedure on every single peak (Picture 19).

```
public static void heapBuild(int heapArray[]) //This function is building a heap with the help of doHeap()
{
    heapSize=heapArray.length-1;   // calculation the size of the heap
    for(int i=heapSize/2;i>=0;i--) // doHeap on the half of the array
        doHeap(heapArray,i);
}
```

Picture 19

The processing order of the peaks guarantees that when the doHeap() procedure runs on an “i” peak its children already form a heap.

The heapSort() procedure (Picture 20) starts with the call of the heapBuild(), which change the heapArray[1...n] input array to a heap.

```
public static void heapSort(int a0[]) // This function manages the heap sorting
{
    heapArray=a0;
    heapBuild(heapArray);       // build heap from array

    for(int i=heapSize;i>0;i--)
    {
        changePlace(0, i);      // 0th and ith element of the arrey change place
        heapSize=heapSize-1;    // decrease the heapSize
        doHeap(heapArray, 0);   // call the doHeap
    }
}
```

Picture 20

The root element is the biggest element because of the heap property. So if we swap the heapArray[1] and the heapArray[n] elements the biggest element gets to its place based on sorting. Excluding the n-th element from the heap, the remaining heapArray[1...(n-1)] element can easily be changed to heap again, as the sub-trees belonging to the children of the root have heap properties, only the element that has got into the root can breach that.

With the call of the doHeap(heapArray, 0) the heap property can be restored in the heapArray[1...(n-1)] array. This is repeated decreasing the size of the heap from n-1 to 2.

Task 2.2: Error handling in Java

As every data that is asked by the HeapPeter.java program has to be an integer, therefore every user input could be solved with the help of one single function (Picture 21).

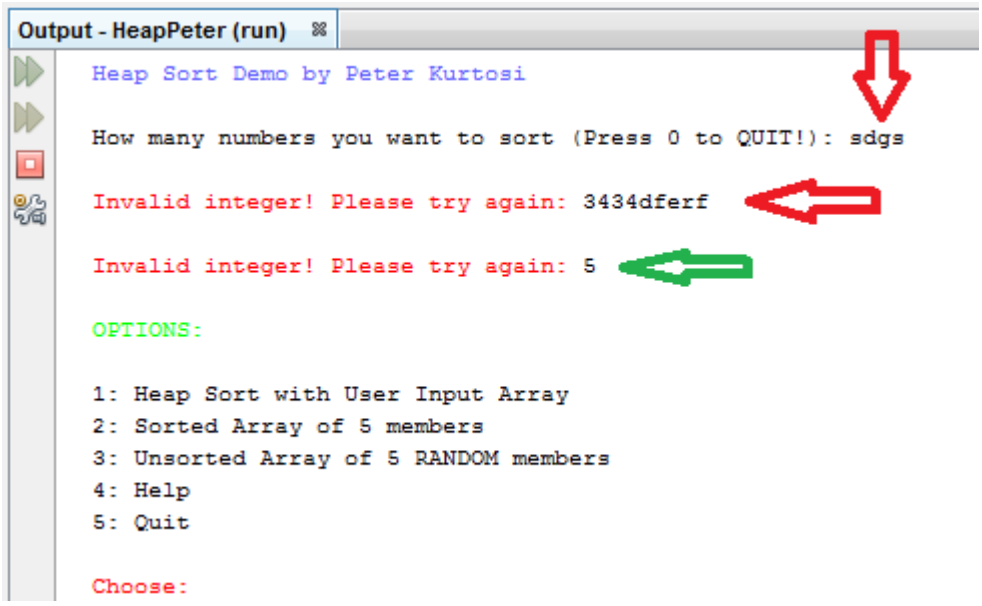
```
public static int onePositiveIntInput() //Input function for positive integer
{
    int oneInt;
    Scanner scan = new Scanner (System.in);
    try
    {
        oneInt = scan.nextInt();
        if(oneInt < 0)  //Error handling - negative value
        {
            System.out.print("\n" + (char)27 + RED +"This number is negative! Please try again: " + (char)27 + ENDCOLOR);
            oneInt = onePositiveIntInput();
        }
    }
    catch(InputMismatchException error)  //Error handling - integer
    {
        System.out.print("\n" + (char)27 + RED +"Invalid integer! Please try again: " + (char)27 + ENDCOLOR);
        oneInt = onePositiveIntInput();
    }

    return oneInt;
}
```

Picture 21

It can be seen on Picture 21 that the onePositiveIntInput() procedure contains the handling of two different errors.

One of the things to be examined is that the value entered by the user is really integer or not. This problem has been handled with the try{} - catch{} structure.

With the help of the try{} command, the program “tries to give an integer value” to the oneInt variable from the Scanner stream through the “scan” object. The catch{} method, called by an InputMismatchException parameter, “catches” the possible error and takes over the control of the program. It displays a report to the user about the “caught” error (Picture 22) then recalls the onePositiveIntInput() method. It repeats this until gets a valid integer value from the user.



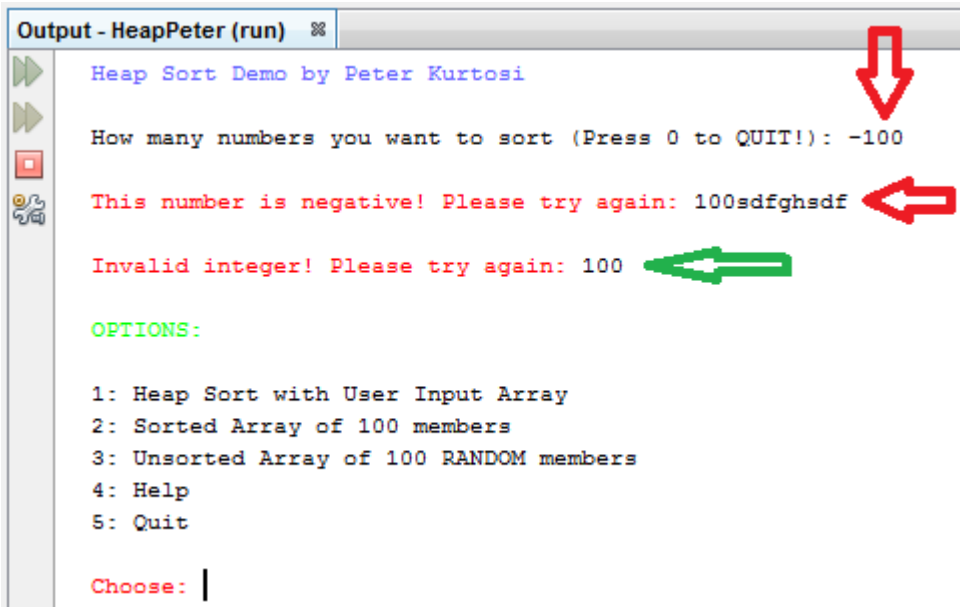
Picture 22

The onePositiveIntInput() method also has to examine whether the value of the incoming user input is less than zero (in this case the zero value is acceptable) (Picture 23).

```
oneInt = scan.nextInt();
if(oneInt < 0)  //Error handling - negative value
{
    System.out.print("\n" + (char)27 + RED +"This number is negative! Please try again: " + (char)27 + ENDCOLOR);
    oneInt = onePositiveIntInput();
}
```

Picture 23

As the catch{} method has already filtered the not valid input so it is enough to examine whether the user entered value is negative or not. If it is negative, then it displays a report in which it informs the user that the entered value is negative (although a valid integer) (Picture 24). As in this case it is not proper input, therefore it restarts the onePositiveIntInput() method.



Picture 24

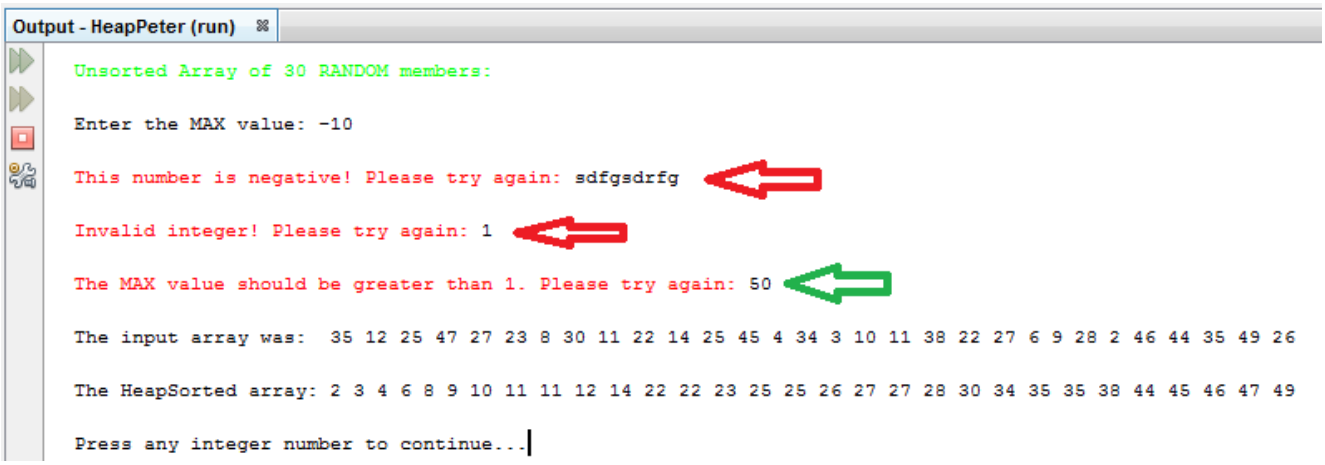
In Option 3, the program fills the array of user defined size with random numbers between 1 and maxX. For this it asks the maxX value, which is also implemented by the onePositiveIntInput() method. But as in this case either the zero or the 1 value does not make sense so further tests had to be inserted (Picture 25).

```
System.out.print("\nEnter the MAX value: ");
int maxX;
do
{
    maxX = onePositiveIntInput();
    if(maxX <=1)
        System.out.print("\n" + (char)27 + RED + "The MAX value should be greater than 1. Please try again: " + (char)27 + ENDCOLOR);
} while (maxX <= 1);
```

Picture 25

A do{}while() loop is perfect for this purpose. After maxX got a value from the onePositiveIntInput() method, the previously mentioned error handlings have already done their jobs. So the value of maxX will certainly be a positive integer (zero is also a positive integer). In this case, however, both the 0 and the 1 value have to be filtered out and a report

must be sent to the user about the error. The cycle stops if the value of maxX is a positive integer with a value of 2 or greater (Picture 26).



Picture 26

The HeapPeter program always displays the reports that describe the errors in red colour, even this way referring to the improper operation as well.

Task 2.3: Testing of the result

The operation principle and structure of the HeapSort algorithm has been presented in detail in Task 2.1. In this chapter the expected (calculated) running time of the part-algorithms that build-up the HeapSort algorithm will be defined. Afterwards, by testing the developed program with data, we examine whether the received test result correspond with the expected ones.

Let the height of one element of the tree be the number of the elements of that longest route, which leads from the given peak to a leaf. The height of the tree should be the height of the root element. As a heap with n elements is based on a complete binary tree, its height is $O(\log_2 n)$. It can be seen that the execution time of the basic operations implemented on the heap is proportional to the height of the tree, i.e. to $O(\log_2 n)$.
The part-algorithms used during the HeapSort algorithm are as follows:

- doHeap() method, with a running time of $O(\log_2 n)$. It is a key method for maintaining the heap property (Picture 17).
- heapBuild() method, with linear running time. It builds up a heap from arbitrary input data (Picture 19).
- heapSort() method, with a running time of $O(n \log_2 n)$ (Picture 20).

The running time of the **doHeap()** procedure, considering a sub-tree with size n and i root, is O(1) – during which the relations between the heapArray[i], heapArray[leftSide(i)] and heapArray[rightSide(i)] can be examined - plus the time during which the doHeap() is executed on the sub-tree from one of the children of the i peak.

```
doHeap(heapArray, i)
        l <- leftSide(i)
        r <- rightSide(i)
        if l ≤ heapSize[heapArray] and heapArray[l] > heapArray[i]
                then peak <- l
                else peak <- i
        if r ≤ heapSize[heapArray] and heapArray[r] > heapArray[peak]
                then peak <- r
        if peak != i
                then changePlace heapArray[i] <-> heapArray[peak]
                        doHeap(heapArray, peak)
```

The size of each of these sub-trees is at most is (2n/3). The worst case is when the lowest level of the tree is exactly half-filled. Based on these, the run time of the doHeap() method can be calculated with the following inequality:

$$T(n) \leq T(2n / 3) + O(1).$$

The solution of the recursion based on the second base case of the “master theorem” is: $T(n) = O(\log_2 n)$. (Cormen, et al., 2009)

A simple upper limit of the run time of the **heapBuild()** method can be obtained as follows: every single call of the doHeap() takes $O(\log_2 n)$ time and there are O(n) of such methods so its run time is $O(n \log_2 n)$. This is a fair estimation but not close asymptotically (Sedgewick & Wayne, 2011).

```
heapBuild(heapArray)
        heapSize[heapArray] <- length[heapArray]
        for i <- (length[heapArray]/2) downto 1
                do doHeap(heapArray, i)
```

Sharper limit can be obtained, if we observe that the run time of the doHeap() depends on the height of the peak in the tree, which for most of the peaks is small. According to Wayne, the stronger estimation is based on that property that the height of the n-element heap is $\log_2 n$ and in the case of any h height the number of h-height peaks is at most $[n/2^{h+1}]$.

The run time of the doHeap() on a h-height peak is O(h), so the complete run time of the heapBuild() is:

$$\sum_{h=0}^{\log_2 n} \left[\frac{n}{2^{h+1}}\right] O(h) = \left(n \sum_{h=0}^{\log_2 n} \left[\frac{h}{2^h}\right] \right)$$

From this the following upper limit can be calculated for the run time of the heapBuild:

$$O\left(n \sum_{h=0}^{\log_2 n} \left[\frac{h}{2^h}\right] \right) = O\left(n \sum_{h=0}^{\infty} \left[\frac{h}{2^h}\right] \right) = O(n)$$

So an unsorted heap can be formed into a heap in linear time.

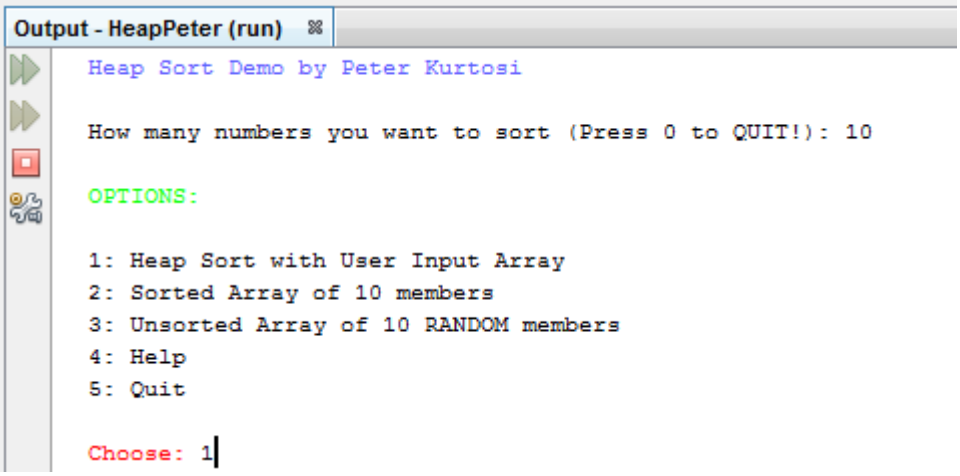
The run time of **heapSort()** is $O(n \log_2 n)$, as the heapBuild() runs in O(n) time and the (n-1) times run of the doHeap() method take $O(\log_2 n)$ time.

```
heapSort(heapArray)
        heapBuild(heapArray)
        for i <- length[heapArray] downto 2
                do changePlace heapArray[1] <-> heapArray[i]
                        heapSize[heapArray] <- heapSize[heapArray] - 1
                        doHeap(heapArray,1)
```

The 2nd line of the for loop shows the state of each heap that exists at the beginning of the iteration step.

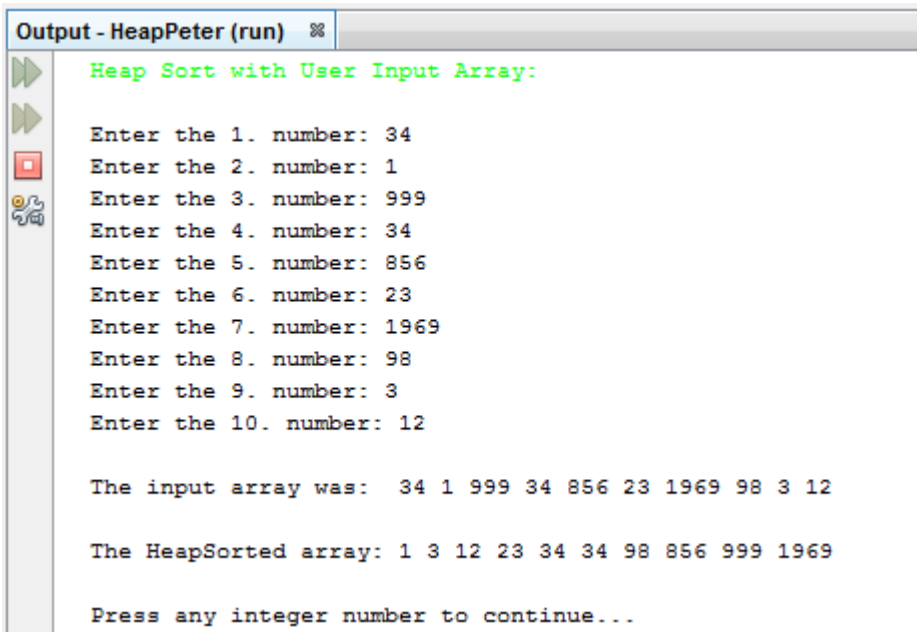
Hereinafter let us examine the implementation of the HeapSort algorithm in the HeapPeter program. For measuring the run time I will use the Analyze Performance function of the Net Beans 7.4 IDE environment.

At the start of the HeapPeter program it asks the number of data to be sorted (the size of the heapArray[]) (Picture 27).

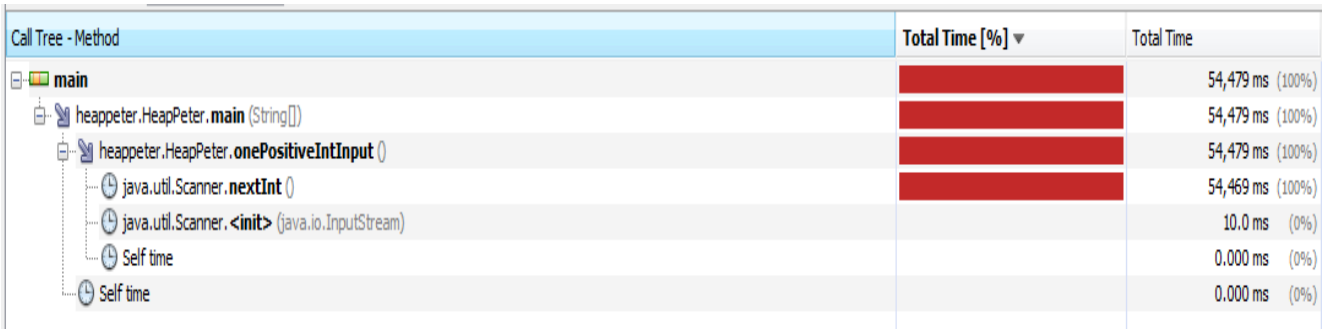


Picture 27

After this, by choosing option 1, the user can upload the elements to be sorted (Picture 28). The run time list made by the Analyze Performance can be seen on Picture 29.

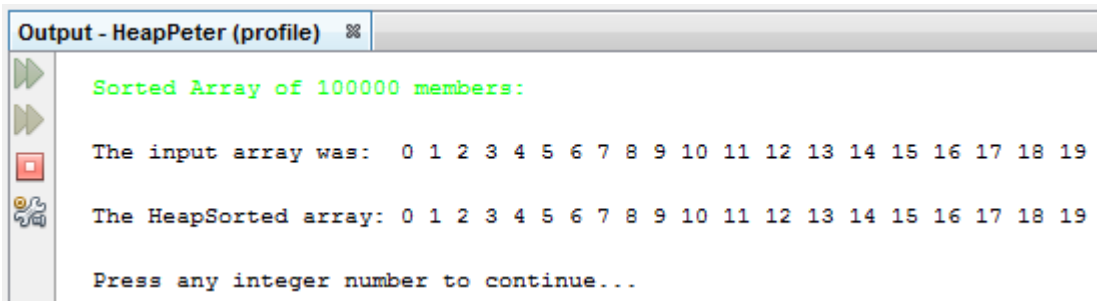


Picture 28



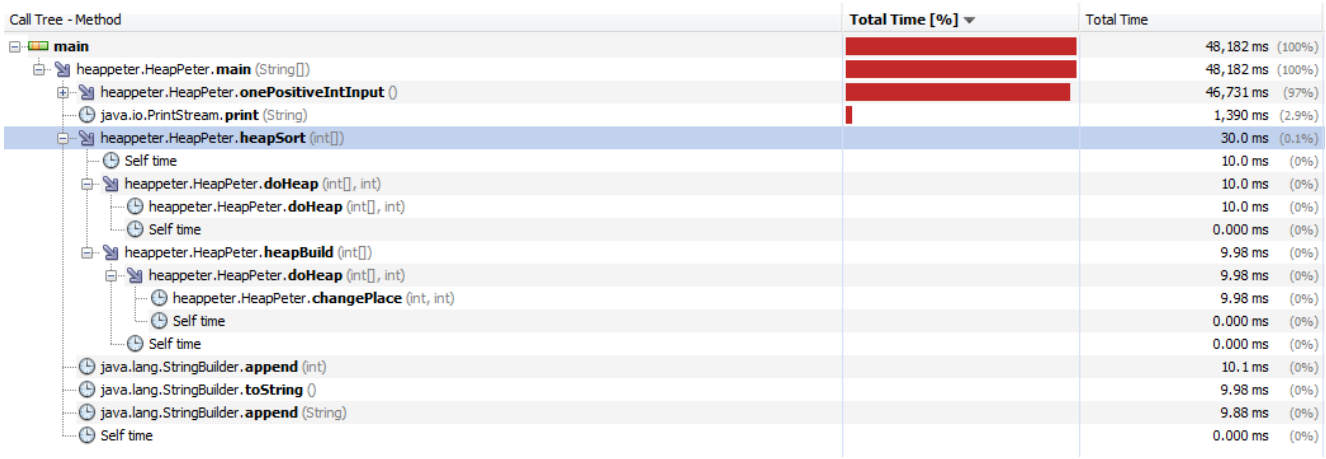
Picture 29

It can be read from Picture 29 that the measuring of such a small array does not produce any result. As the run time, which is smaller than 1ms is undetectable that is why I started to increase the number of elements that has to be sorted. As not even increasing by an order of magnitude had any result, so I have expanded the program with two more options. In the case of the 2nd and 3rd option, it is not the user who fills up the data.
At the next test case I have worked with an array of size 100.000. First, choosing the 2nd option, the program filled up the array with sorted integers from 1 to 100.000 (Picture 30).



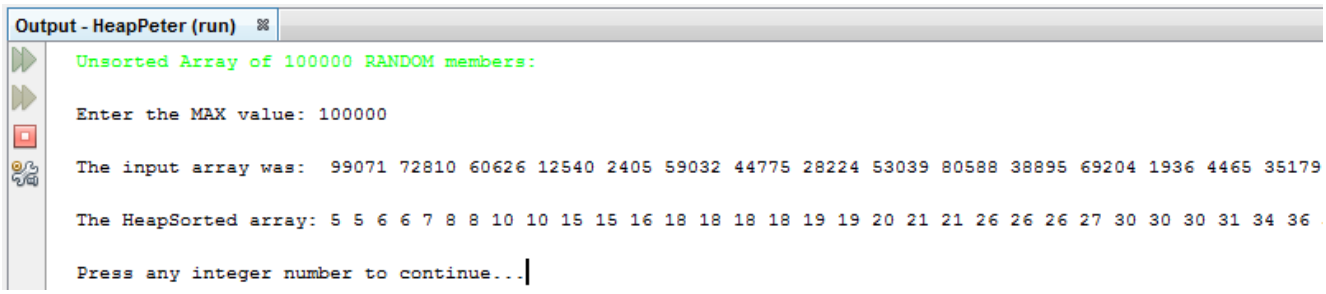
Picture 30

Picture 31 illustrates the run time of the single methods.



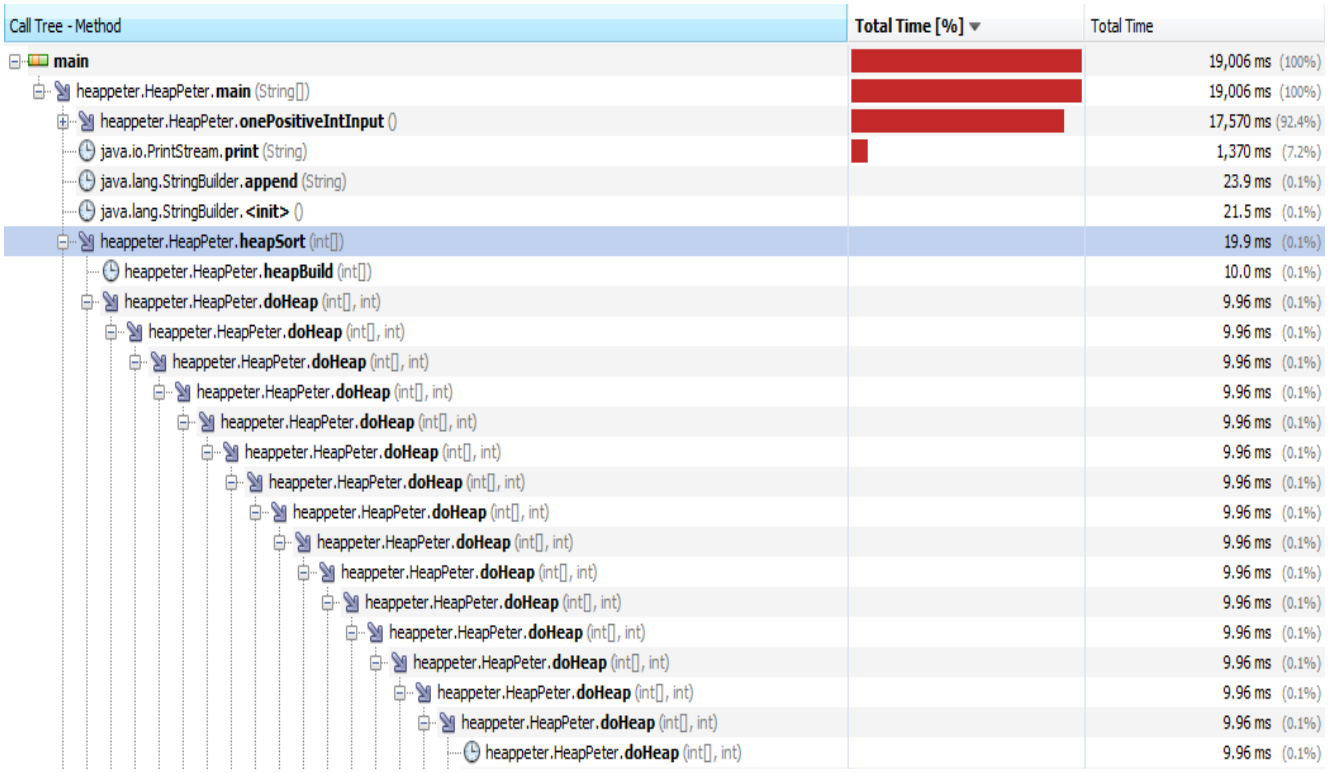
Picture 31

Let us now examine the run times with similar parameters. In this test case I have also been working with an array of 100.000 elements and I have filled up the array again with integers between 1 and 100.000 but in this case with the help of a random number generator (Picture 32).



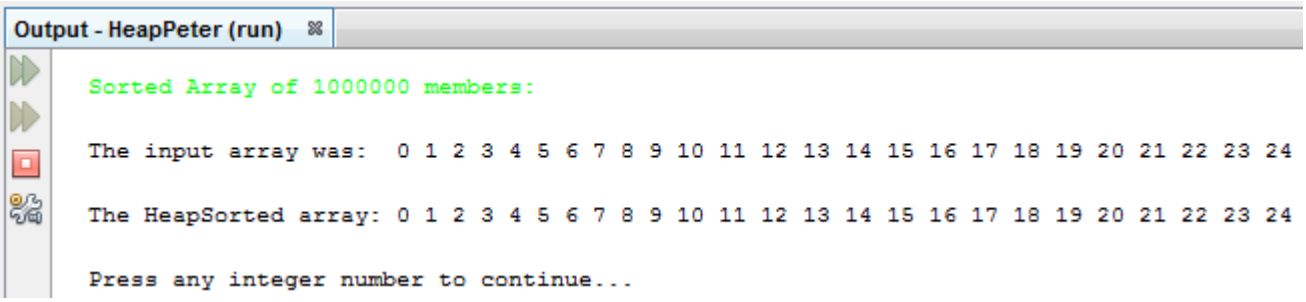
Picture 32

Picture 33 illustrates the measurements results.



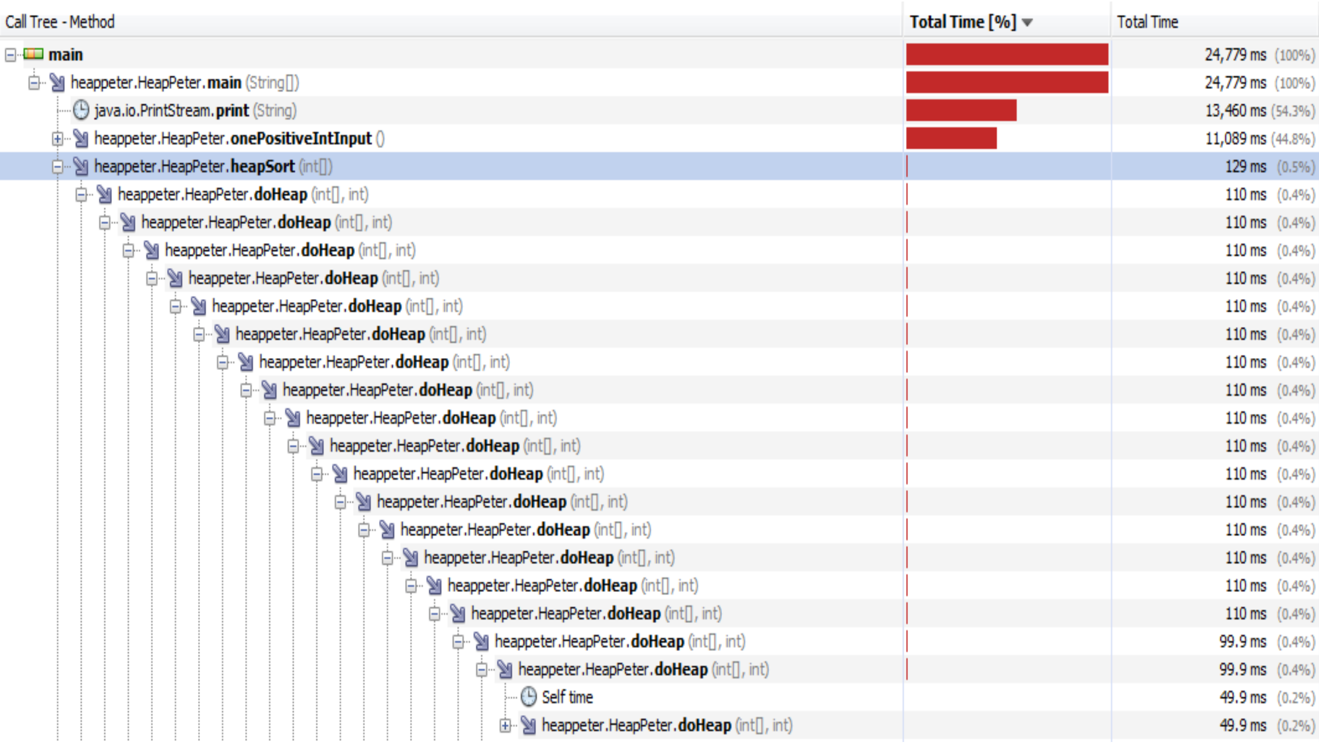
Picture 33

It can be seen from the run times that the run time of the method, which forms the heap property has significantly increased because of the random data.
In the next test case, I have increased the size of the array to 1.000.000 and I have filled it up with sorted integers from between 1 and 1.000.000 according to the 2nd option (Picture 34).

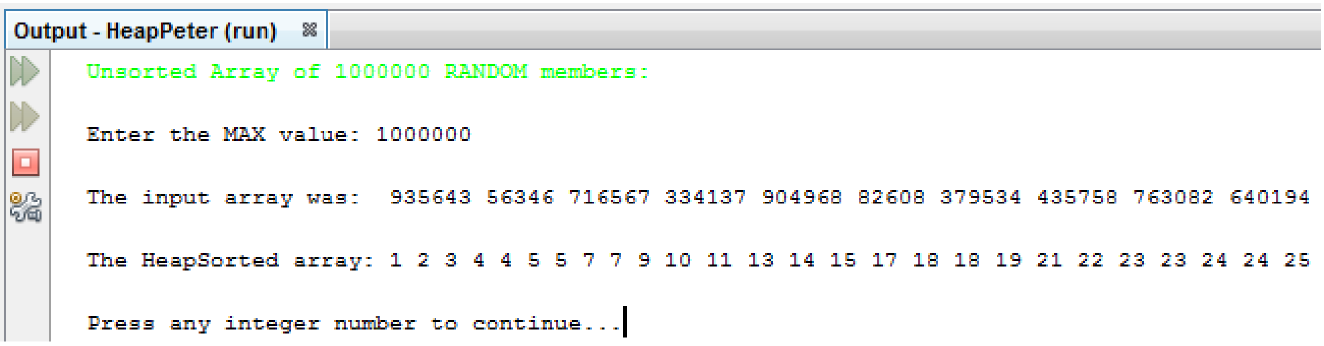


Picture 34

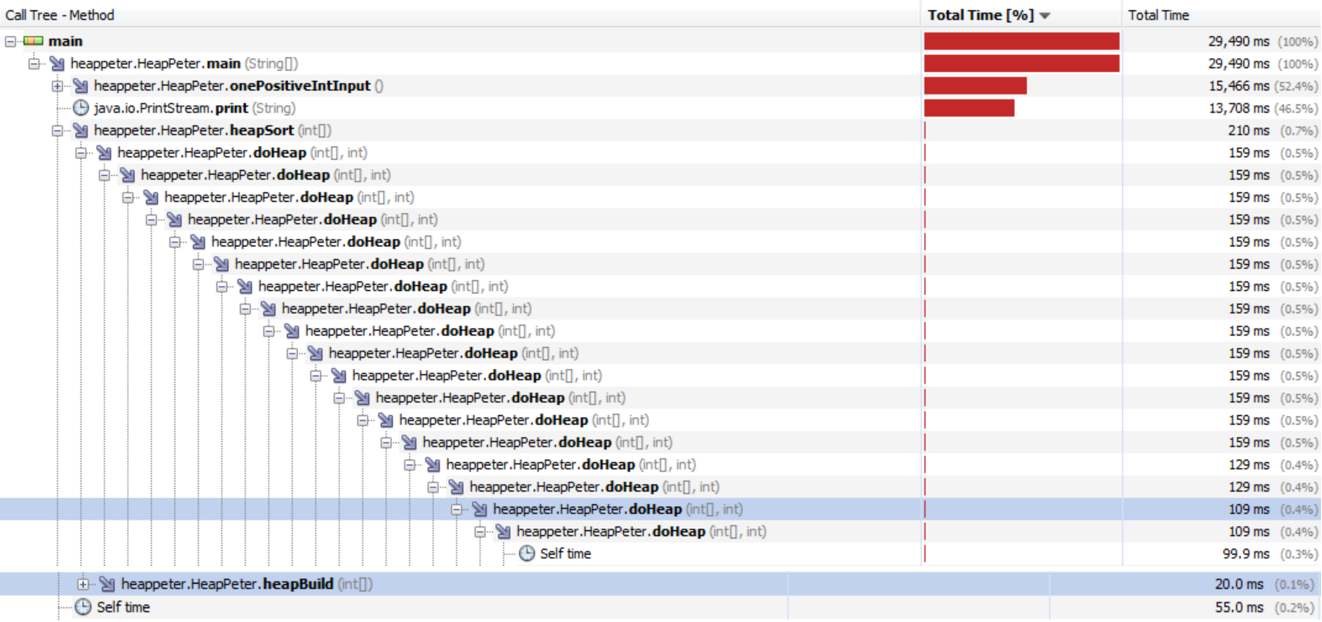
The result of the execution can be seen on Picture 35. For easier comparison, in the next test case I have worked again with an array of the size of 1.000.000 and with random integers in the interval between 1 and 1.000.000 (Picture 36). Picture 37 illustrates the results of this test.



Picture 35

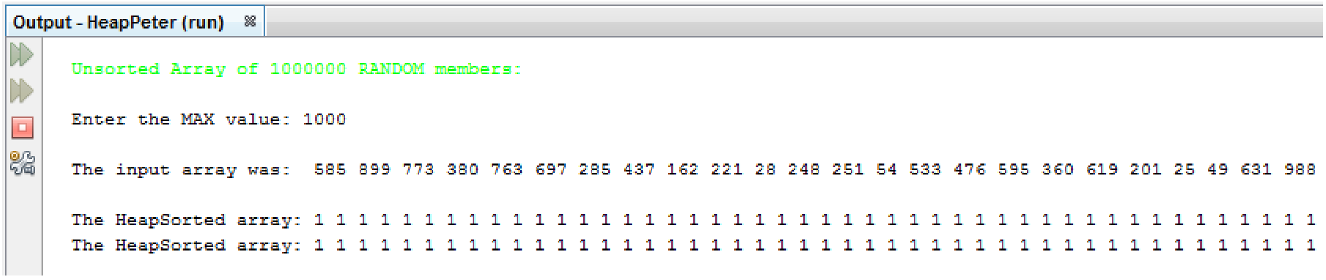


Picture 36



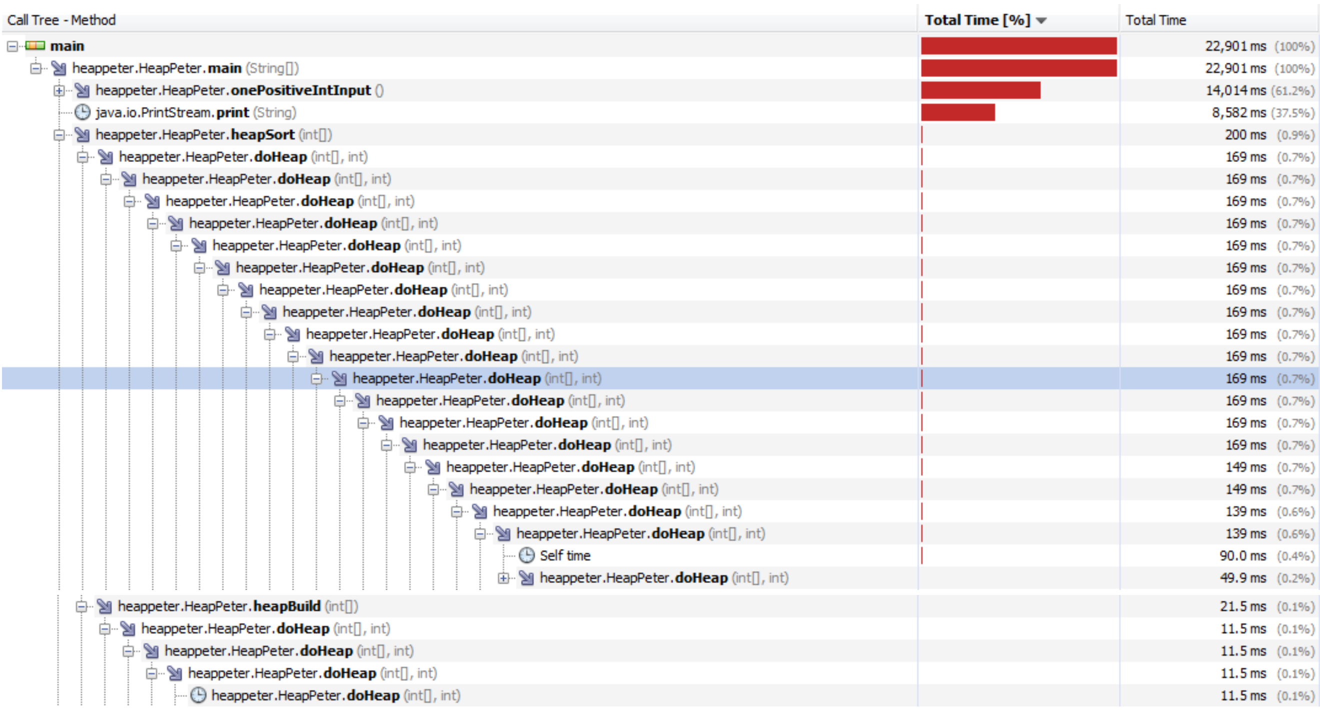
Picture 37

In the final test cast let the size of the array be 1.000.000 again but now I fill it up with random integers only between 1 and 1.000. As a result of this, the repetition of the numbers is much larger. According to my expectations, in run time a result between the completely sorted and in the 1-1.000.000 interval unsorted test cases can be expected (Picture 38).



Picture 38

Picture 39 demonstrates the test results.



Picture 39

The test shows run times in accordance with expectations. Naturally, the function of the NetBeans Analyze Performance is not suitable for the verification of the mathematical run time equations defined at the beginning of Task 2.3. However, it shows those tendencies well, which depend on the quantity of the data to be sorted and the sort rate of the input data set. To completely separate the subroutines of the HeapSort algorithm is impossible, since during their own run they call each other from time to time.

Task 3: String operations

There is no primitive type for strings in Java language. However, there is a "class String {}", which can be used to store and process characters that form a chain. This chapter presents the common operations offered by the "class String {}".

Task 3.1: The common string operations in Java

As soon as a String object is created neither the length of its value nor any of its characters can be changed. This means that a String object is immutable. At the same time, there are numerous methods in the String class that return with new String objects as a result of the transformation of the original string's value. Table 1 demonstrates some common string operations of the "class String". (Lewis, et al., 2011)

`String (String str)`
Constructor: creates a new string object with the same characters as `str`.

`char charAt (int index)`
Returns the character at the specified `index`.

`int compareTo (String str)`
Returns an integer indicating if this string is lexically before (a negative return value), equal to (a zero return value), or lexically after (a positive return value), the string `str`.

`String concat (String str)`
Returns a new string consisting of this string concatenated with `str`.

`boolean equals (String str)`
Returns true if this string contains the same characters as `str` (including case) and false otherwise.

`boolean equalsIgnoreCase (String str)`
Returns true if this string contains the same characters as `str` (without regard to case) and false otherwise.

`int length ()`
Returns the number of characters in this string.

`String replace (char oldChar, char newChar)`
Returns a new string that is identical with this string except that every occurrence of `oldChar` is replaced by `newChar`.

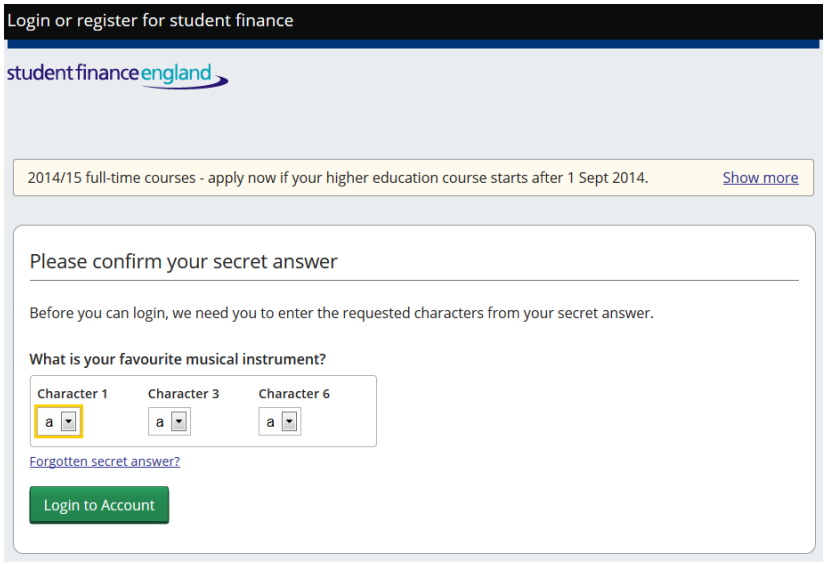
`String substring (int offset, int endIndex)`
Returns a new string that is a subset of this string starting at index `offset` and extending through `endIndex-1`.

`String toLowerCase ()`
Returns a new string identical to this string except all uppercase letters are converted to their lowercase equivalent.

`String toUpperCase ()`
Returns a new string identical to this string except all lowercase letters are converted to their uppercase equivalent.

Table 1

Almost the full range of the operations related to strings is used in practice by for example the word processing programs.
A great example for the use of the charAt() method is the Student Finance's Login procedure. It can be seen on Picture 40 that this time only 1st, 3rd and 6th characters of the password had to be entered.

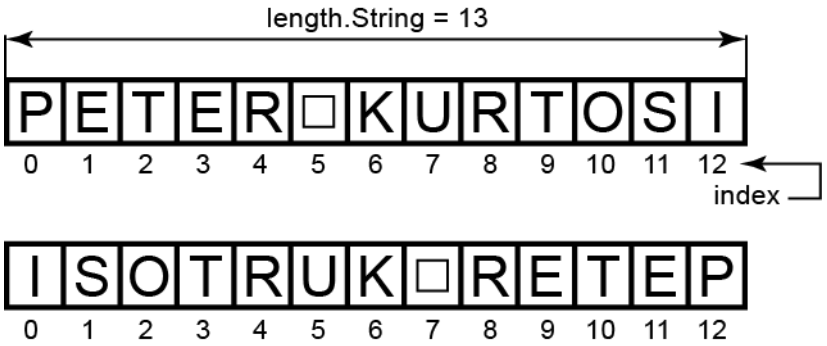


Picture 40

The comparative string operations play significant role in dictionary programs, search algorithms (e.g. Google), and so on.
I used the replace() method several times while I was writing the programs during for the assignment, when the NetBeans renamed e.g. one of the variables at all of its occurrence place.
The substring() type methods are suitable, among other things, to divide a properly structured code. For example, my LSBM student ID is LON01111239. It is easy to choose from this the substring that contains only the numbers.
The toLower-toUpper methods are very useful in the case of for example the hot-keys. The save function of the MS-WORD is Ctrl+s or Ctrl+S. The program does not have to examine both cases this time, it is enough if it converts every character that is entered as a hot-key into capital letters and only interprets these.

Task 3.2: How to reverse a string

In this task I present 2 two different algorithms for reversing a string. The full list of the ReversePeter program can be found in Appendix H. In both cases I used that feature of the string that it behaves as a character array. Its every element can be accessed with the help on an index (Picture 41.)



Picture 41

Reversing a String with the help of an ARRAY:
The main feature of the algorithm (Picture 42) is that I have created a character array, the length of which is equal to the length of the string. (The return value of the length() is the length of the given string.)

```
public static void reverseArray(String tmpString, int tmpLength)
{
    char reverseArray[] = new char [tmpLength]; // new array for reverse string
    String reverseString = "";
    // copy the original string to the reverseArray in reverse order
    for(int i=tmpLength-1; i>=0; i--)
        reverseArray[tmpLength-1-i]=tmpString.charAt(i);
    System.out.print("The reverse string with using of ARRAY: ");
    // copy the content of the reverseArray to a string
    for(int i=0; i<=tmpLength-1; i++)
        reverseString = reverseString + reverseArray[i];
    System.out.println(reverseString + "\n");
}
```

Picture 42

In the next step, I asked the elements of the string in reverse order, with the help of a for cycle. I used the charAt() for the query. After this, by using the same cycle variable, I asked the elements of the reverseArray in increasing order. With this I achieved that the last element of the string became the first element of the array, the penultimate element of the string became the second element of the array and so on. So the order of the elements of the string has reversed.

I read the elements of the array in order and interlinked them to one string again with another for cycle. This way the reverse string has been created.

Reversing a String with the help of a STACK:
In this algorithm (Picture 43) I used the FILO feature of the stack. With the help of the push() method, I put the elements of the string into the stack in order with the help a for cycle. With the help of another for cycle, which ran as many times as the length of the string, and with a pop() method I interlinked the elements of the stack to a string again. There was no need to know the indexes of the elements in the stack as the pop() method is only able to empty the stack in reverse order.

```
public static void reverseStack(String tmpString, int tmpLength)
{
    String reverseString = "";
    Stack reverseStack = new Stack(); // open a stack for string
    // copy the original string to a stack char by char
    for(int i=0; i<tmpLength; i++)
        reverseStack.push(tmpString.charAt(i));
    System.out.print("The reverse string with using of STACK: ");
    // take out the string from tha stack char by char and build up the reverse string
    for(int i=0; i<tmpLength; i++)
        reverseString = reverseString + reverseStack.pop();
    System.out.println(reverseString + "\n");
}
```

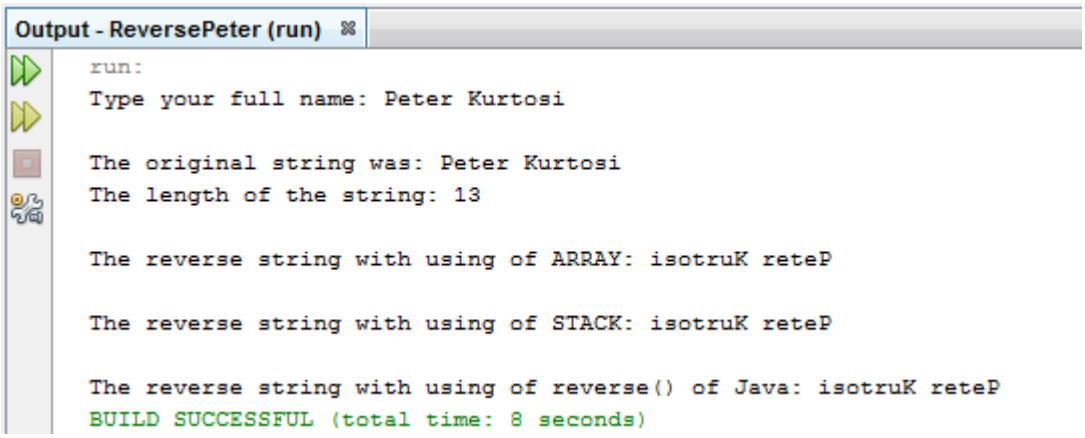
Picture 43

Reversing a String with the help of the class String{} - reverse() method:
As curiosity, I have presented the reverse() method of the String class implemented in Java language as well (Picture 44).

```
System.out.print("The reverse string with using of reverse() of Java: ");
String reverseString = new StringBuffer(lineInput).reverse().toString();
System.out.println(reverseString);
```

Picture 44

The execution of the ReversePeter program had the required result in the case of all three algorithms, as it can be seen on Picture 45.



Picture 45

Conclusion

The science of Data Structures and Algorithms was great evidence for that that the science of programming is much more than just the knowledge of the syntax and command set of a computer language. It requires unique vision and thinking mechanism.
Who once had a taste of this branch of the science will observe the surrounding world differently.

References

Cormen, T., Leiserson, C., Rivest, R. & Stein, C., 2009. *Introduction to Algorithms.* 3rd ed. United States of America: Massachusetts Institute of Technology.

Lewis, J., DePasquale, P. & Chase, J., 2011. *Java Foundations - Introduction to Program Design & Data Structures.* 2nd ed. Boston: Addison-Wesley - Pearson Education, Inc..

Mandelbrot, B. B., 2002. *The Fractal Geometry of Nature.* 2nd ed. New York: Henry Holt and Company.

Neapolitan, R. & Naimipour, K., 2004. *Foundations od Algorithms Using C++ Pseudocode.* 3rd ed. Boston: Jones and Bartlett Computer Science.

Puntambekar, A., 2008. *Analysis & Design of Algorithms.* 1st ed. Pune: Technical Publications Pune.

Ray, J., 2011. *iPhone Application Development.* 2nd ed. United States of America: Pearson Education, Inc..

Sedgewick, R. & Wayne, K., 2011. *Algorithms.* 4th ed. Boston: Pearson Education, Inc..

Appendix A

```
import java.util.*;

public class StackPeter
{
    public static void main(String[] args)
    {
        System.out.println("STACK-DEMO by Peter\n");
        System.out.println("1.step: Create an EMPTY Stack (called stackDemo)!");
        Stack stackDemo = new Stack();
        System.out.println("stackDemo = " + stackDemo + " stackSize = " + stackDemo.size() + "\n");

        System.out.println("2.step: Push a 'L' character into the Stack");
        stackDraw(stackDemo, 'L');
        System.out.println("3.step: Push a 'S' character into the Stack");
        stackDraw(stackDemo, 'S');
        System.out.println("4.step: Push a 'B' character into the Stack");
        stackDraw(stackDemo, 'B');
        System.out.println("5.step: Push a 'M' character into the Stack");
        stackDraw(stackDemo, 'M');
        System.out.println("6.step: Use the peek() function!");
        System.out.println("The result of the peek(): " + stackDemo.peek());
        System.out.println("stackDemo = " + stackDemo + " stackSize = " + stackDemo.size() + "\n");
        System.out.println("7.step: Use the pop() function!");
        System.out.println("The result of the pop(): " + stackDemo.pop());
        System.out.println("stackDemo = " + stackDemo + " stackSize = " + stackDemo.size() + "\n");
        System.out.println("8.step: Use the push('M') function!");
        stackDraw(stackDemo, 'M');
    }
    static void stackDraw(Stack stackDemo, char lsbm)
    {
        stackDemo.push(lsbm);
        int actSize = stackDemo.size();
        System.out.println("stackDemo = " + stackDemo + " stackSize = " + actSize + "\n");
    }
}
```

Appendix B

```
import java.util.*;

public class QueuePeter
{
    public static void main(String[] args)
    {
        System.out.println("QUEUE-DEMO by Peter\n");
        System.out.println("1.step: Create an EMPTY Queue (called queueDemo)!");
        Queue queueDemo = new LinkedList();
        System.out.println("queueDemo = " + queueDemo + " queueSize = " + queueDemo.size() + "\n");

        System.out.println("2.step: Add a 'L' character into the Queue");
        queueDraw(queueDemo, 'L');
        System.out.println("3.step: Add a 'S' character into the Queue");
        queueDraw(queueDemo, 'S');
        System.out.println("4.step: Add a 'B' character into the Queue");
        queueDraw(queueDemo, 'B');
        System.out.println("5.step: Add a 'M' character into the Queue");
        queueDraw(queueDemo, 'M');
        System.out.println("6.step: Use the peek() function!");
        System.out.println("The result of the peek(): " + queueDemo.peek());
        System.out.println("queueDemo = " + queueDemo + " queueSize = " + queueDemo.size() + "\n");
        System.out.println("7.step: Use the poll() function!");
        System.out.println("The result of the poll(): " + queueDemo.poll());
        System.out.println("queueDemo = " + queueDemo + " queueSize = " + queueDemo.size() + "\n");
        System.out.println("8.step: Use the add('L') function!");
        queueDraw(queueDemo, 'L');
    }
    static void queueDraw(Queue queueDemo, char lsbm)
    {
        queueDemo.add(lsbm);
        int actSize = queueDemo.size();
        System.out.println("queueDemo = " + queueDemo + " queueSize = " + actSize + "\n");
    }
}
```

Appendix C

```
import java.util.Scanner;
class BinPeter1
{
  public static void main(String args[])
  {
    int c, first, last, middle, n, search, array[];
    Scanner in = new Scanner(System.in);
    System.out.println("Binary Search-DEMO by Peter\n");
    System.out.print("Enter number of elements: ");
    n = in.nextInt();
    array = new int[n];
    System.out.println("Enter " + n + " integers:");
    for (c = 0; c < n; c++)
      array[c] = in.nextInt();
    System.out.print("Enter value to find: ");
    search = in.nextInt();
    first  = 0;
    last   = n - 1;
    middle = first + ((last - first) / 2);
    while( first <= last )
    {
      if ( array[middle] < search )
        first = middle + 1;
      else if ( array[middle] == search )
      {
        System.out.println(search + " found at location " + (middle + 1) + ".");
        break;
      }
      else
         last = middle - 1;
      middle = (first + last)/2;
   }
   if ( first > last )
      System.out.println(search + " is not present in the list.\n");
  }
}
```

Appendix D

```
public class QuickPeter
{
    public static int partition(int quickArray[], int left, int right)
    {
          int i = left, j = right;
          int tmp;
          int pivot = quickArray[(left + right) / 2];
          while (i <= j) {
                while (quickArray[i] < pivot)
                      i++;
                while (quickArray[j] > pivot)
                      j--;
                if (i <= j) {
                      tmp = quickArray[i];
                      quickArray[i] = quickArray[j];
                      quickArray[j] = tmp;
                      i++;
                      j--;
                }
          };
      return i;
}
    public static void quickSort(int quickArray[], int left, int right)
    {
          int index = partition(quickArray, left, right);
          if (left < index - 1)
                quickSort(quickArray, left, index - 1);
          if (index < right)
                quickSort(quickArray, index, right);
    }
    public static void main(String[] args)
    {
        int x;
        int quickArray[] = new int[] {12,8,18,10,7,14,2,17,5};
        System.out.println("Quicksort-DEMO by Peter\n");
        System.out.println("The unsorted array: \n");
        for(x=0; x< quickArray.length; x++)
            System.out.print(quickArray[x] + " ");
        quickSort(quickArray, 0, quickArray.length-1);
        System.out.println("\n\nThe Quicksorted array: \n");
        for(x=0; x< quickArray.length; x++)
            System.out.print(quickArray[x] + " ");
        System.out.println("\n");
    }
}
```

Appendix E

Tower of Hanoi

By Peter Kurtosi

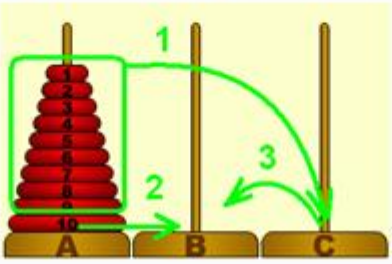
The rule

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- No disk may be placed on top of a smaller disk.

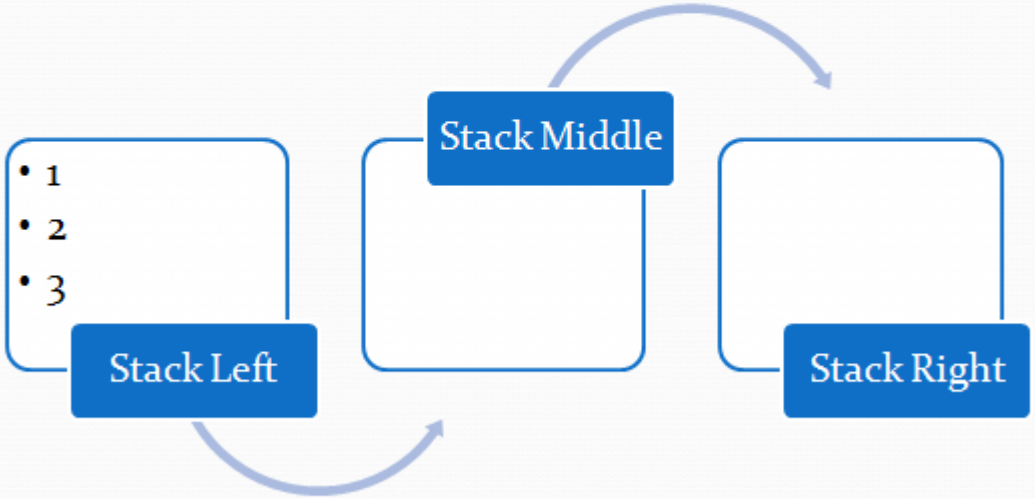
(The minimum number of moves required to solve a Tower of Hanoi puzzle is $2^n - 1$, where n is the number of disks.)

The solution

- To move *n* discs from peg A to peg C:
- move *n*–1 discs from A to B. This leaves disc *n* alone on peg A
- move disc *n* from A to C
- move *n*–1 discs from B to C so they sit on disc *n*



One stick is one Stack in Java



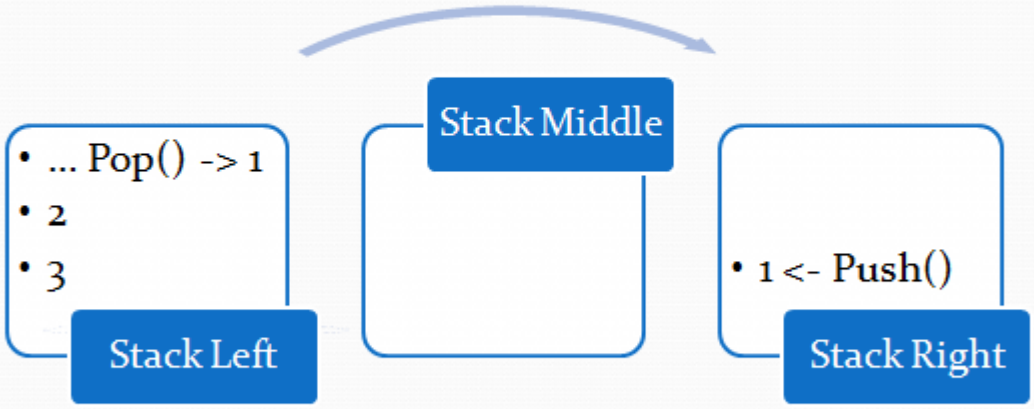
Stacks in Java

```
Stack stL = new Stack();
Stack stM = new Stack();
Stack stR = new Stack();
```

```
static void stackUpload(int disknr, Stack stL)
{
    for(int i=disknr; i>0; i--)
        stL.push(new Integer(i));
}
```

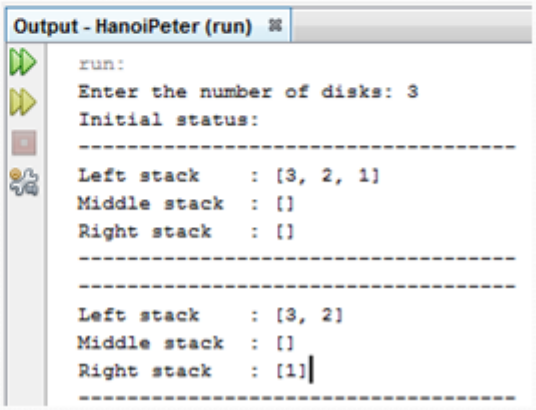
```
Output - HanoiPeter (run)
run:
Enter the number of disks: 3
Initial status:
-----------------------------------
Left stack    : [3, 2, 1]
Middle stack  : []
Right stack   : []
-----------------------------------
```

Push() and Pop()



Push() and Pop() in Java

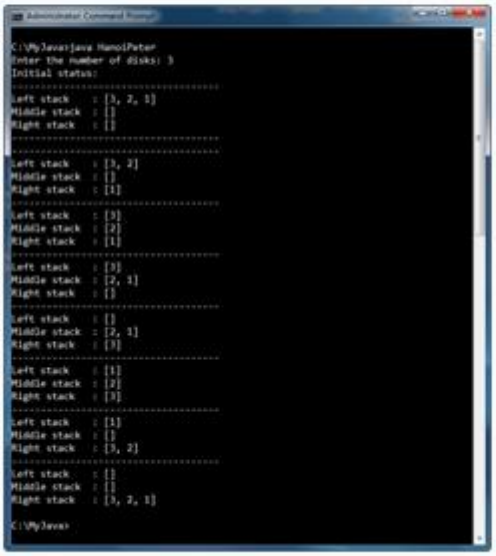
```
Integer temp = (Integer) stL.pop();
stR.push(new Integer(temp));
System.out.println("-----------------------------------");
System.out.println("Left stack    : " + stL);
System.out.println("Middle stack  : " + stM);
System.out.println("Right stack   : " + stR);
```



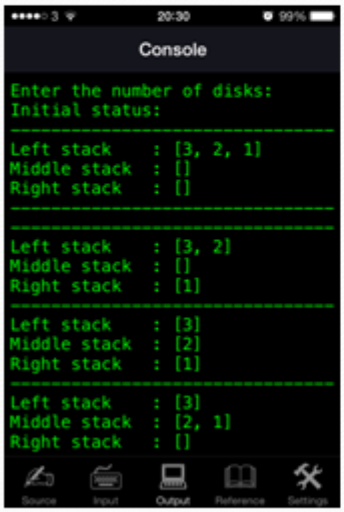
Recursive function call in Java

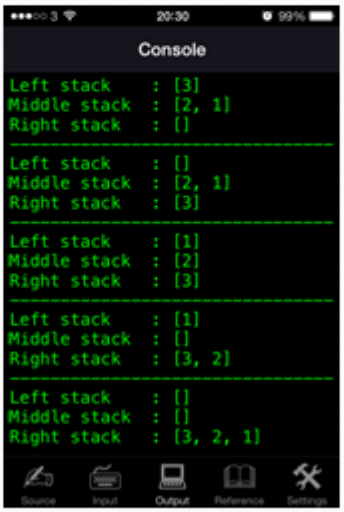
```
static void recurHanoi(int diskNr, String stringLeft, String stringRight, String stringMiddle, Stack stL, Stack stM, Stack stR)
{
    if (diskNr >0)
    {
        recurHanoi(diskNr-1, stringLeft, stringMiddle, stringRight, stL, stM, stR);
        stMove(stringLeft, stringRight, stL, stM, stR);
        recurHanoi(diskNr-1, stringMiddle, stringRight, stringLeft, stL, stM, stR);
    }
}
```

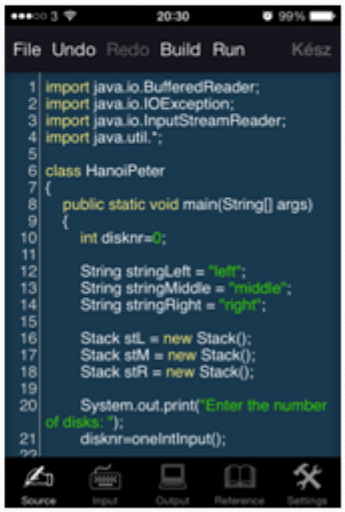
The solution on console

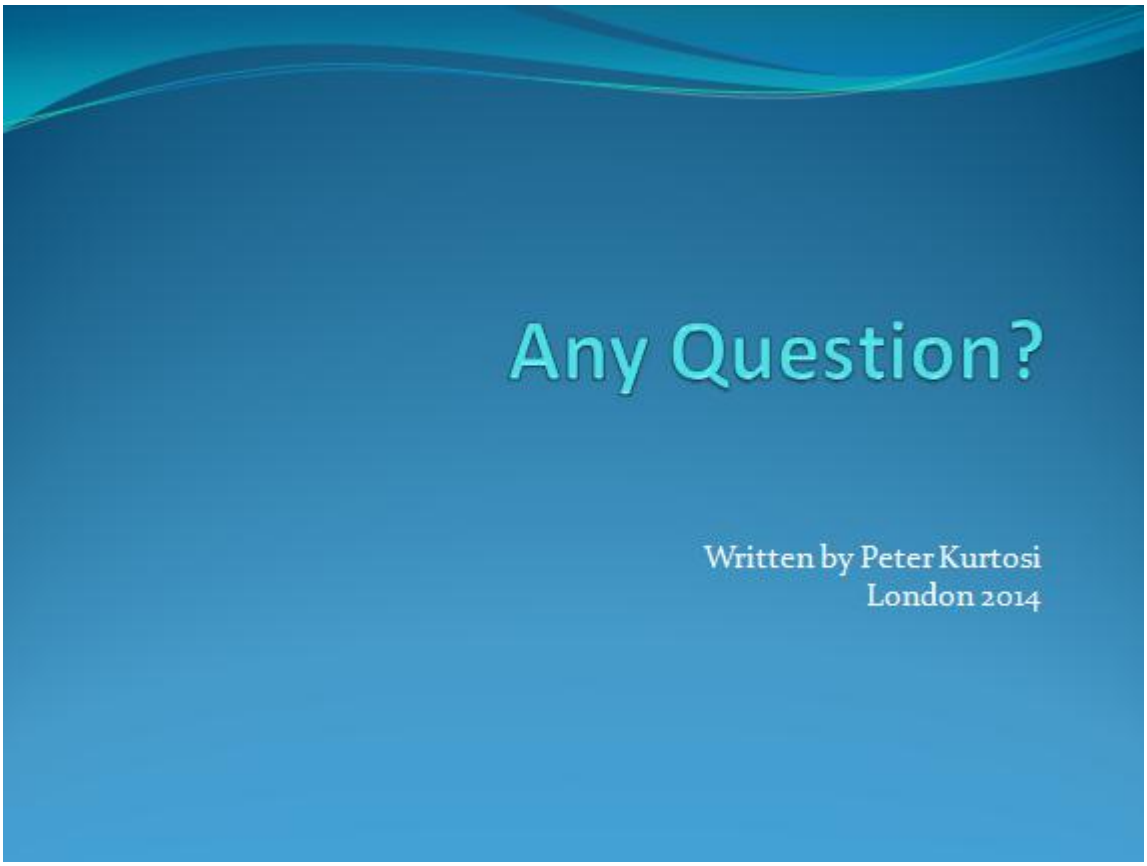


The solution on iPhone








Any Question?
Written by Peter Kurtosi
London 2014

Appendix F

HanoiPeter.java

Source | History

```
1  package hanoipeter;
2
3  import java.io.BufferedReader;
4  import java.io.IOException;
5  import java.io.InputStreamReader;
6  import java.util.*;
7
8  public class HanoiPeter
9  {
10     public static void main(String[] args)
11     {
12         int disknr=0;
13
14         String stringLeft = "left";
15         String stringMiddle = "middle";
16         String stringRight = "right";
17
18         Stack stL = new Stack();
19         Stack stM = new Stack();
20         Stack stR = new Stack();
21
22         System.out.print("Enter the number of disks: ");
23         disknr=oneIntInput();
24
25         System.out.println("Initial status:");
26         System.out.println("-----------------------------------");
27
28         stackUpload(disknr, stL);
29
30         System.out.println("Left stack    : " + stL);
31         System.out.println("Middle stack  : " + stM);
32         System.out.println("Right stack   : " + stR);
33         System.out.println("-----------------------------------");
34
35         recurHanoi(disknr, stringLeft, stringRight, stringMiddle, stL, stM, stR);
36     }
37
38     public static int oneIntInput()
39     {
40         String lineInput = "";
41         int oneInteger = 0;
42
43         BufferedReader reader = new BufferedReader(new InputStreamReader (System.in));
44         try
45         {
46          lineInput = reader.readLine();
47          oneInteger = Integer.parseInt(lineInput);
48         }
49
50         catch(IOException e){ }
51         return oneInteger;
52     }
53
54     static void recurHanoi(int disknr, String stringLeft, String stringRight, String stringMiddle, Stack stL, Stack stM, Stack stR)
55     {
56         if (disknr >0)
57         {
58             recurHanoi(disknr-1, stringLeft, stringMiddle, stringRight, stL, stM, stR);
59             stMove(stringLeft, stringRight, stL, stM, stR);
60             recurHanoi(disknr-1, stringMiddle, stringRight, stringLeft, stL, stM, stR);
61         }
62     }
63
```

cont. on next page...

```
64      static void stMove(String fromSt, String toSt, Stack stL, Stack stM, Stack stR)
65      {
            if(fromSt=="left" && toSt=="middle")
67          {
68              Integer temp = (Integer) stL.pop();
69              stM.push(new Integer(temp));
70              System.out.println("-------------------------------------");
71              System.out.println("Left stack    : " + stL);
72              System.out.println("Middle stack  : " + stM);
73              System.out.println("Right stack   : " + stR);
74          }
            else if(fromSt=="left" && toSt=="right")
76          {
77              Integer temp = (Integer) stL.pop();
78              stR.push(new Integer(temp));
79              System.out.println("-------------------------------------");
80              System.out.println("Left stack    : " + stL);
81              System.out.println("Middle stack  : " + stM);
82              System.out.println("Right stack   : " + stR);
83          }
            else if(fromSt=="right" && toSt=="middle")
85          {
86              Integer temp = (Integer) stR.pop();
87              stM.push(new Integer(temp));
88              System.out.println("-------------------------------------");
89              System.out.println("Left stack    : " + stL);
90              System.out.println("Middle stack  : " + stM);
91              System.out.println("Right stack   : " + stR);
92          }
            else if(fromSt=="right" && toSt=="left")
94          {
95              Integer temp = (Integer) stR.pop();
96              stL.push(new Integer(temp));
97              System.out.println("-------------------------------------");
98              System.out.println("Left stack    : " + stL);
99              System.out.println("Middle stack  : " + stM);
100             System.out.println("Right stack   : " + stR);
101         }
101         }
            else if(fromSt=="middle" && toSt=="left")
103         {
104             Integer temp = (Integer) stM.pop();
105             stL.push(new Integer(temp));
106             System.out.println("-------------------------------------");
107             System.out.println("Left stack    : " + stL);
108             System.out.println("Middle stack  : " + stM);
109             System.out.println("Right stack   : " + stR);
110         }
            else if(fromSt=="middle" && toSt=="right")
112         {
113             Integer temp = (Integer) stM.pop();
114             stR.push(new Integer(temp));
115             System.out.println("-------------------------------------");
116             System.out.println("Left stack    : " + stL);
117             System.out.println("Middle stack  : " + stM);
118             System.out.println("Right stack   : " + stR);
119         }
120
121     }
122
123     static void stackUpload(int disknr, Stack stL)
124     {
125         for(int i=disknr; i>0; i--)
126             stL.push(new Integer(i));
127      }
128
129 }
130
```

Appendix G

HeapPeter.java

Source | History

```
1  package heappeter;
2
3  /**
4   *
5   * @author Peter Kurtosi - LON01111239
6   * Heap Sort Demo
7   *
8   **/
9
10 import java.util.*;
11
12 public class HeapPeter
13 {
14     private static final String ENDCOLOR = "[00;00m"; //ANSI escape sequence: end of colours
15     private static final String BLUE = "[01;34m"; //ANSI escape sequence: BLUE
16     private static final String GREEN = "[01;32m"; //ANSI escape sequence: GREEN
17     private static final String RED = "[01;31m"; //ANSI escape sequence: RED
18
19     private static int heapArray[]; //the variables of the Heap sort
20     private static int heapSize;
21     private static int leftSide;
22     private static int rightSide;
23     private static int peak;
24
25     public static void cls()  //The clear screen function
26     {
27         int i;
28         for (i=1; i<20; i++)
29             System.out.println("\n");
30     }
31
32     public static int onePositiveIntInput() //Input function for positive integer
33     {
34         int oneInt;
35         Scanner scan = new Scanner (System.in);
36         try
37         {
38             oneInt = scan.nextInt();
39             if(oneInt < 0)  //Error handling - negative value
40             {
41                 System.out.print("\n" + (char)27 + RED +"This number is negative! Please try again: " + (char)27 + ENDCOLOR);
42                 oneInt = onePositiveIntInput();
43             }
44         }
45         catch(InputMismatchException error)  //Error handling - integer
46         {
47             System.out.print("\n" + (char)27 + RED +"Invalid integer! Please try again: " + (char)27 + ENDCOLOR);
48             oneInt = onePositiveIntInput();
49         }
50
51         return oneInt;
52     }
53
54     public static void helpHeapsort() //The help function
55     {
56         System.out.println("First the program expects a positive integer, which specifies the size");
57         System.out.println("of the array of the elements that should be sorted.");
58         System.out.println("In the case of 0 the program quits as intended.");
59         System.out.println((char)27 + RED +"Option 1:" + (char)27 + ENDCOLOR);
60         System.out.println("The program asks for the numbers that should be sorted one by one.");
61         System.out.println("Exactly as much, as we entered at the start of the program.");
62         System.out.println("The program expects integers.");
63         System.out.println((char)27 + RED +"Option 2:" + (char)27 + ENDCOLOR);
64         System.out.println("The program fills up the array, which size was specified at the start,");
65         System.out.println("with sorted integers (from 0 to ArraySize-1).");
66         System.out.println((char)27 + RED +"Option 3:" + (char)27 + ENDCOLOR);
67         System.out.println("The program fills up the array, which size was specified at the start,");
68         System.out.println("with random numbers. The maximum value of the random numbers");
69         System.out.println("must be entered. It expects a positive integer that is bigger than 1.");
70     }
71
72     public static int randomNumbers(int maxNum) //This function is generating a random number
73     {
74         int minNum = 1;
75         return (int)(maxNum*Math.random())+minNum; //Type coercion to integer
76     }
77
78     public static void heapBuild(int heapArray[]) //This function is building a heap with the help of doHeap()
```

```
    {
        heapSize=heapArray.length-1;   // calculation the size of the heap
        for(int i=heapSize/2;i>=0;i--) // doHeap on the half of the array
            doHeap(heapArray,i);
    }

    public static void doHeap(int heapArray[], int i) //This function rearrange the order of the members
    {
        leftSide=2*i;      // calculation of the left side
        rightSide=2*i+1;   // calculation of the right side
        if(leftSide <= heapSize && heapArray[leftSide] > heapArray[i])
            peak=leftSide; // the value of peak -> left
        else
            peak=i;        // the value of peak -> i
        if(rightSide <= heapSize && heapArray[rightSide] > heapArray[peak])
            peak=rightSide;// the value of peak -> right
        if(peak!=i)
        {
            changePlace(i,peak);     // i and peak change place in the array
            doHeap(heapArray, peak); // recursive calling of the doHeap
        }
    }

    public static void heapSort(int a0[]) // This function manages the heap sorting
    {
        heapArray=a0;
        heapBuild(heapArray);       // build heap from array

        for(int i=heapSize;i>0;i--)
        {
            changePlace(0, i);      // 0th and ith element of the arrey change place
            heapSize=heapSize-1;    // decrease the heapSize
            doHeap(heapArray, 0);   // call the doHeap
        }
    }

    public static void changePlace(int i, int j) // This function is changing the values of necessary members
    {
        int t=heapArray[i];        // temporary variable for ith element
        heapArray[i]=heapArray[j]; // value of "j"th -> "i"th place
        heapArray[j]=t;            // the original value of "i"th -> "j"th place
    }

    public static void main(String[] args) //The MAIN function
    {
        int menuItem = 0;          // variable for the menu
        while (menuItem < 5)       // loop of the menu
        {
        cls();                     // the clear screen function
        String message = "Heap Sort Demo by Peter Kurtosi";
        System.out.println((char)27 + BLUE + message + (char)27 + ENDCOLOR);

        System.out.print("\nHow many numbers you want to sort (Press 0 to QUIT!): ");
        int maxNumber = onePositiveIntInput(); // the size of the array from user input
        int inputArray[] = new int[maxNumber]; // create the array
        System.out.print("\n");
        if ( maxNumber != 0)
        {
        System.out.println((char)27 + GREEN + "OPTIONS:" + (char)27 + ENDCOLOR);

        System.out.println("\n1: Heap Sort with User Input Array");
        System.out.println("2: Sorted Array of " + maxNumber + " members");
        System.out.println("3: Unsorted Array of " + maxNumber + " RANDOM members");
        System.out.println("4: Help");
        System.out.println("5: Quit");
        System.out.println("");
        System.out.print((char)27 + RED + "Choose: " +(char)27 + ENDCOLOR);

        menuItem = onePositiveIntInput();
        }
        else
            menuItem = 5;

        switch (menuItem)
        {
            case 1:
                cls();
                System.out.println();
```

```
                System.out.println((char)27 + GREEN + "Heap Sort with User Input Array:\n" + (char)27 + ENDCOLOR);

                for(int i=0;i<inputArray.length;i++)
                {
                    System.out.print("Enter the " + (i+1) + ". number: " );
                    inputArray[i] = onePositiveIntInput();
                }
                System.out.print("\nThe input array was:  ");
                for(int i=0;i<inputArray.length;i++)
                    System.out.print(inputArray[i] + " ");

                heapSort(inputArray);

                System.out.print("\n\nThe HeapSorted array: ");
                for(int i=0;i<inputArray.length;i++)
                    System.out.print(inputArray[i] + " ");

                System.out.print("\n\nPress any integer number to continue...");
                onePositiveIntInput();
                break;
            case 2:
                cls();
                System.out.println();
                System.out.println((char)27 + GREEN + "Sorted Array of " + maxNumber + " members:" + (char)27 + ENDCOLOR);
                for(int i=0;i<inputArray.length;i++)
                    inputArray[i] = i;
                System.out.print("\nThe input array was:  ");
                for(int i=0;i<inputArray.length;i++)
                    System.out.print(inputArray[i] + " ");

                heapSort(inputArray);

                System.out.print("\n\nThe HeapSorted array: ");
                for(int i=0;i<inputArray.length;i++)
                    System.out.print(inputArray[i] + " ");

                System.out.print("\n\nPress any integer number to continue...");
                onePositiveIntInput();
                break;
            case 3:
                cls();
                System.out.println();
                System.out.println((char)27 + GREEN + "Unsorted Array of " + maxNumber + " RANDOM members:" + (char)27 + ENDCOLOR);

                System.out.print("\nEnter the MAX value: ");
                int maxX;
                do
                {
                    maxX = onePositiveIntInput();
                    if(maxX <=1)
                        System.out.print("\n" + (char)27 + RED + "The MAX value should be greater than 1. Please try again: " + (char)27 + ENDCOLOR);
                } while (maxX <= 1);

                for(int i=0;i<inputArray.length;i++)
                    inputArray[i] = randomNumbers(maxX);

                System.out.print("\nThe input array was:  ");
                for(int i=0;i<inputArray.length;i++)
                    System.out.print(inputArray[i] + " ");

                heapSort(inputArray);

                System.out.print("\n\nThe HeapSorted array: ");
                for(int i=0;i<inputArray.length;i++)
                    System.out.print(inputArray[i] + " ");

                System.out.print("\n\nPress any integer number to continue...");
                onePositiveIntInput();
                break;
            case 4:
                cls();
                System.out.println();
                System.out.println((char)27 + RED + "Heap Sort Demo by Peter Kurtosi:" + (char)27 + ENDCOLOR);
                helpHeapsort();
                System.out.print("\nPress any integer number to continue...");
                onePositiveIntInput();
                break;
            default:
                System.out.println("\nGood By!!!\n");
        }
        }
    }
}
```

Appendix H

ReversePeter.java

Source | History

```
1  package reversepeter;
2
3  /**
4   *
5   * @author Peter Kurtosi - LON01111239
6   * Reverse String Demo
7   *
8   **/
9
10 import java.io.*;
11 import java.util.Stack;
12
13 public class ReversePeter
14 {
15     public static void main(String[] args)
16     {
17         String lineInput = "";
18
19         System.out.print("Type your full name: ");
20         BufferedReader reader = new BufferedReader(new InputStreamReader (System.in));
21         try
22         {
23          lineInput = reader.readLine(); // read a whole line from user input
24         }
25         catch(IOException e){ }          // error handling
26
27         System.out.print("\nThe original string was: " + lineInput + "\n");
28
29         int lengthString = lineInput.length(); // calculate the lenght of string
30         System.out.println("The length of the string: " + lengthString + "\n");
31
32         reverseArray(lineInput, lengthString); // reverse string with array
33         reverseStack(lineInput, lengthString); // reverse string witj stack
34
35         System.out.print("The reverse string with using of reverse() of Java: ");
36         String reverseString = new StringBuffer(lineInput).reverse().toString();
37         System.out.println(reverseString);
38     }
39
40     public static void reverseArray(String tmpString, int tmpLength)
41     {
42         char reverseArray[] = new char [tmpLength]; // new array for reverse string
43         String reverseString = "";
44         // copy the original string to the reverseArray in reverse order
45         for(int i=tmpLength-1; i>=0; i--)
46             reverseArray[tmpLength-1-i]=tmpString.charAt(i);
47         System.out.print("The reverse string with using of ARRAY: ");
48         // copy the content of the reverseArray to a string
49         for(int i=0; i<=tmpLength-1; i++)
50             reverseString = reverseString + reverseArray[i];
51         System.out.println(reverseString + "\n");
52     }
53
54     public static void reverseStack(String tmpString, int tmpLength)
55     {
56         String reverseString = "";
57         Stack reverseStack = new Stack(); // open a stack for string
58         // copy the original string to a stack char by char
59         for(int i=0; i<tmpLength; i++)
60             reverseStack.push(tmpString.charAt(i));
61         System.out.print("The reverse string with using of STACK: ");
62         // take out the string from tha stack char by char and build up the reverse string
63         for(int i=0; i<tmpLength; i++)
64             reverseString = reverseString + reverseStack.pop();
65         System.out.println(reverseString + "\n");
66     }
67 }
68
```