Table of Contents

Table of Figures

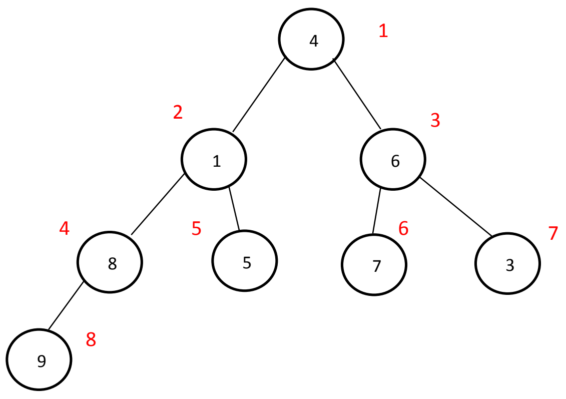
1 Research

Heap-sort is an in place sorting algorithm with an *O(n log n)* complexity. Heaps can be visualised as a binary tree. Heap-sort is made up of three functions; Build-Heap, Heapify and Heap-sort. Heap-sort has an efficiency of O(1), however, due to variables in xsl being innumerable, the implementation is recursive.

1.1 Build-Heap

The first step of a heap-sort is the build-heap function. This takes a list and creates a max-heap. A heap can be seen in a binary tree structure. A heap data structure seen as a binary tree must be "near complete". This means that every parent node must be larger than its children (In the case of max-heap). This is purpose of the build-heap function.

A binary tree is a tree structure in which each parent node can have a maximum of two child nodes. This does not mean that each node must have 2 children. The last section of the tree can have a single child or even no children. An example of this is given in Figure 1.



When visualising a heap as a binary tree you must first index the heap. Following this, the binary tree is filled with structure shown (in red), such that it is filled starting with the left child node. Also, a parent node must have two child nodes before a child node can have its own child node.

For example the list [4 1 6 8 5 7 3 9] which is processed by the build-heap function would produce the tree shown here.

Figure 1 Binary Tree Structure

In a build heap each node is checked to see if it is the root of a valid heap. This is done iteratively. The check starts at the first node that is not a heap in its own right. This node will always be the node length/2, as any node after this will not have any child nodes. The node that is being checked is compared to its respective child nodes. For the node to be a valid heap root it must be larger than its children and so, if either of the children are larger than the current node the two swap positions. If both the children are larger than the current node, then the node swaps with the bigger of the two children. This is done to all the nodes in turn going from node length/2 to ((length/2)-1) etc until the root node is reached, which results in a valid heap. This can be seen in figure 2. This swap is known as the "Heapify" function which is explained in section 1.2.

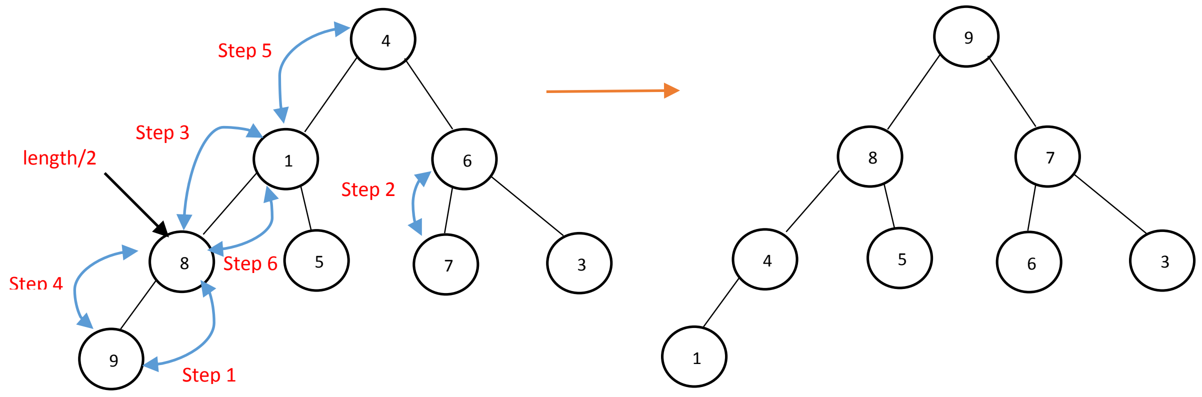


Figure 2 Build-Heap Step by Step Example

1.2 Heapify

From the observation of the tree structure provided above, it can be seen that assuming that a current node has index i, then:

- its Parent node has index (i/2);
- its Left child has index (2i);
- its Right child has index (2i + 1).

For the simplicity of notation, under the name Heapify we will use the Max-Heapify algorithm, as only the notion of a max heap will be considered under this assignment as a mean of sorting the list of numbers. The Heapify algorithm, which is needed by both Build-Heap and Heap-sort algorithms can be then implemented as following:

```
Heapify (A,i)
1 l = Left (i)
2 r = Right (i)
3 if  l < = A.heap-size and A[l] > A[i]
4         largest = l
5 else largest = i
6 if r  < = A.heap-size and A[r] > A[largest]
7         largest = r
8 if largest ! =  i
9         exchange A[i] with A[largest]
10        Heapify (A, largest)
```

Transforming the algorithm according with the node indexing rules provided above results in:

```
Heapify (A,i)
1 l = (2*i)
2 r = (2*i) + 1
3 if  l < = A.heap-size and A[l] > A[i]
4         largest = l
5 else largest = i
6 if r  < = A.heap-size and A[r] > A[largest]
7         largest = r
8 if largest ! =  i
9         exchange A[i] with A[largest]
10        Heapify (A, largest)
```

This algorithm takes a list of numbers (A) and an index of the current node (i) as its two arguments. First, it finds the left and right children of the node and then determines the largest among the node and its children. If the node itself holds the largest value, then no action is performed and the algorithm is terminated. Otherwise, if one of its children is larger than the other child and the node itself, then its value is exchanged with the current node and the algorithm is repeated with the index of the largest child. To sum up, the Heapify algorithm takes a node and a tree as its arguments and pushes the node down to its correct position if it is not already in place.

The Heapify algorithm can only be applied if left and right branches are already max-heaps, but when A[i] might be smaller than its children.

So, for instance, if we take a list of numbers “4 1 6 8 5 7 3 9” and its 4th element - 8, then 8 and 9 (being the left child of 8, bigger than 8) will be exchanged, which will result in: “4 1 6 9 5 7 3 8”. The algorithm will stop at this point, as there is nothing else that 8 can be replaced with, because now, being a leaf, it has no children.

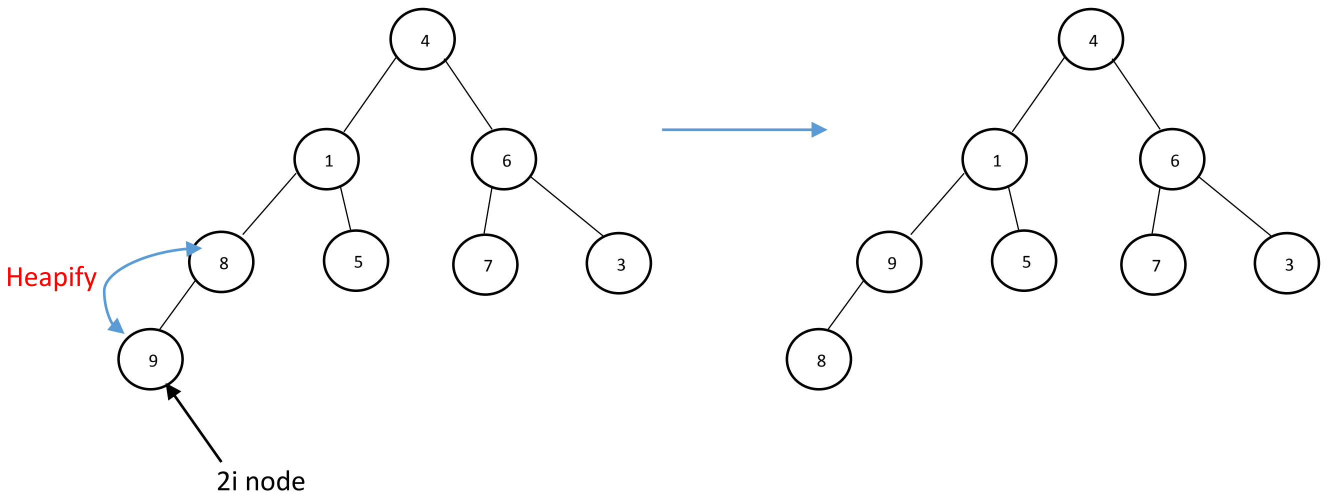


Figure 3 Heapify Example

1.3 Heap-Sort

Heap-sort algorithm plays a major role in the actual sorting of the list of numbers:

```
Heap-sort (A)
1 Build-Heap (A)
2 for i = A.length downto 2
3         exchange A[1] with A[i]
4         A.heap-size = A.heap-size - 1
5         Heapify(A, 1)
```

This algorithm takes a list of numbers as an argument. Firstly, it builds a max-heap by applying Build-Heap function, then by iterating through nodes, starting from the rightmost leaf node and going back to its first/root node’s left child, during each iteration it exchanges its first and *i*-th node and calls Heapify.

So, for example, we start with a list of numbers: “4 1 6 8 5 7 3 9”. As it was shown above, application of a Build-Heap will result in “9 8 7 4 5 6 3 1”. Then by consecutively applying exchange with root node and Heapify in each iteration, the result will be as following: “1 3 4 5 6 7 8 9“.

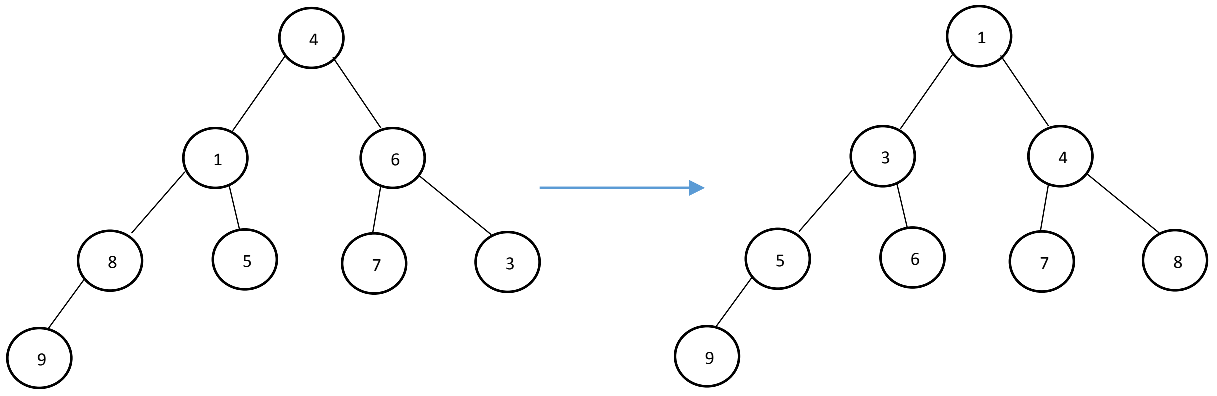


Figure 4 Heap-sort Example

2 Implementation Decisions

This assignment was developed and tested under Linux system (ITL) in oXygen v.15.0. The code itself is fully commented through in order to make following it easier. Below, only major implementation decisions will be described.

Our implementation sorts a list of number through building a max-heap first. Then the result is passed to Heap-sort, which then transforms a max-heap into a list of sorted numbers.

As it was mentioned above, due to limitations of xsl, recursion was used in place of loops in all three algorithms. In order to control the looping index for both Build-Heap and Heap-sort, a couple of helper functions – Iterators were implemented. Exchange (nodes) function was extracted in order to avoid duplication as it is actively used in both Build-Heap and Heap-sort algorithms.

Some of the calls of external templates were wrapped into a variable in order to retrieve the return value of the template and use it further. For instance, within the Build-Heap template, the call-template of Heapify was wrapped in a variable, as its return value needs to be passed back to Build-Heap to be further processed.

3 Testing

For testing purposes we have created a number of test cases:

- a list of positive integers;
- a list of odd length, containing zeros;
- a very long list;
- a list of just one number;
- ordered list;
- list in a reverse order;
- list containing negative numbers;
- list containing duplicates

Input	**Build-Heap**	**HeapSort**
4 1 6 8 5 7 3 9	9 8 7 4 5 6 3 1	1 3 4 5 6 7 8 9
0 1 0 5 7 8 2	8 7 2 5 1 0 0	0 0 1 2 5 7 8
10 13 5 21 7 1 87 3 5 4 24 2 12 5 6 7 9 10 5 7 3 4 8 3 25 10 13 5 21 7 1 87 3 5 4 24 2 12 5 6 7 9 10 5 7 3 4 8 3 25 10 13 5	87 25 87 24 24 25 87 21 13 10 8 25 21 21 24 9 10 10 12 7 9 7 7 8 10 13 13 5 5 7 12 7 7 9 7 8 10 5 5 6 7 4 3 5 4 3 4 3 3 2 1 10	1 1 1 2 2 2 3 3 3 3 3 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 5 6 6 6 7 7 7 7 7 7 7 7 7 8 8 8 9 9 9 10 10 10 10 10 10 12 12 12 13 13 13 21 21 21 24 24 24 25 25 25 87 87 87

21 7 1 87 3 5 4 24 2 12 5 6 7 9 10 5 7 3 4 8 3 25	5 12 7 1 5 3 5 4 6 2 1 5 6 3 3 5 5 4 3 4 5 3 2	
1	1	1
1 2 3 4 5 6	6 5 3 4 2 1	1 2 3 4 5 6
6 5 4 3 2 1	6 5 4 3 2 1	1 2 3 4 5 6
-1 43 3 6 -12 5 6	43 6 6 -1 -12 5 3	-12 -1 3 5 6 6 43
-100 -20 -35 -47 -5	-5 -20 -35 -47 -100	-100 -47 -35 -20 -5
1 1 1 1 1 1 1 1 1 2	2 1 1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1 1 2
1 2 1 2 1 2	2 2 2 1 1 1	1 1 1 2 2 2

4 Submitted Documents

File name	**Purpose**
heap.xsd	Schema defining the tree structure provided for the coursework
heap.xml	Document containing a sample listOfNumbers
heap-sort.xsl	XSL Transformation stylesheet presenting the implemented functions Heap-sort, Build-Heap and Heapify together with helper functions
HeapSort Report.pdf	Algorithm description, explanation of code and implementation decisions, test cases and list of files submitted

5 References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. 2009. Introduction to Algorithms (3rd ed)

2. w3shools.com. XSL languages.
   http://www.w3schools.com/xsl/xsl_languages.asp

3. w3.org. XSL Transformations (XSLT). Version 1.0.
   http://www.w3.org/TR/xslt#xslt-namespace