# 1. Implementation

For the implementation of the Heap data structure, and the other functions, classes and other abstractions like generic programming and comparators are used. By doing this, any kind of data can be sorted effectively without altering the implementation ever. For instance, a user can define a structure and create an array of objects out of it. By providing a comparator or overloading the less than operator, those user defined objects can be sorted correctly. Note that comparisons are made between keys. Therefore, users have to specify key fields of their types in another class similar to a comparator.

## 1.1. MAX-HEAPIFY

The implementation of this procedure starts with calculating indices of the left and right children of the given node. Then, key values of these three nodes are compared with each other. If the parent node has not got the largest key, then it is swapped with the bigger child. Then, the procedure calls itself recursively until the parent has the largest key. Since the number of levels inside a complete binary tree is $log(n)$, the number of recursive calls will also be $log(n)$. Knowing that the other operations like assignment and comparison takes constant time, it can be concluded that this procedure is $O(log(n))$. For the leaf nodes, there is no need of taking an action. So, the best case running time is $\Omega(1)$

## 1.2. BUILD-MAX-HEAP

This function utilizes MAX-HEAPIFY procedure to turn an unordered array of elements into a max-heap. Beginning the half of the array, MAX-HEAPIFY is called for each index until the beginning of the array. Workload of the nodes at the bottom of the heap take $\Theta(1)$ time to process, whereas the nodes closer to the root take $\Theta(log(n))$ time to place correctly inside the heap. By summing the time spent on each node, the running time of this function can be calculated as $O(n)$.

## 1.3. HEAPSORT

First, the BUILD-MAX-HEAP function is called if it is not called before. Then, starting from the last node, elements are swapped with the root node and the heap size is decremented. After that, MAX-HEAPIFY is called for the new root node to preserve the heap property. By using this process, the array is sorted *in-place*, since no new array is created. The running time of the HEAPSORT algorithm is $\Theta(nlog(n))$. This is because the MAX-HEAPIFY is called $n$ times for the root node that takes $O(log(n))$ time to process. Even if the number of nodes are decreased in each iteration, the height of the tree is not decreased that fast. So the running time is strictly $\Theta(nlog(n))$.

## 1.4. Priority Queue Operations

In the first few versions of the implementation, the priority queue operations were served under their own class that was inherited from MaxHeap class. In the later versions, the methods are placed under the MaxHeap class and the PriorityQueue class was discarded, due to code repetitions.

Priority queues are useful in many real world applications such as

- **Task Scheduling:**

  In an operating system, a priority queue can be used to schedule tasks or processes based on their priority levels. In job scheduling systems, tasks with higher priority might get processed before those with lower priority.

- **Job Scheduling in Multiprocessing Systems:**

  In multiprocessing systems, jobs are scheduled for execution based on their priority levels. A priority queue helps efficiently select the next job to be executed.

- **Dijkstra's Shortest Path Algorithm:**

  Priority queues are used in Dijkstra's algorithm for finding the shortest path in a graph with weighted edges. Nodes are processed based on their current distances from the source, with the priority queue ensuring efficient retrieval of the node with the smallest distance.

- **Huffman Coding for Data Compression:**

  Huffman coding, used in data compression algorithms, assigns shorter codes to more frequent symbols. Priority queues help efficiently build the Huffman tree by repeatedly combining the two least frequent symbols.

### 1.4.1. MAX-HEAP-INSERT

Firstly, the element is pushed back of the array. Then, the heap size is incremented, and the HEAPIFY-UP method is called to preserve the heap property, which is similar to the MAX-HEAPIFY, but from bottom to the up instead of up to down. HEAPIFY-UP method compares the keys of the element with the key of its parent. If the parent has a smaller key, then they are swapped. This method is the bottle-neck of the insertion and takes $O(log(n))$, which is the overall running time of the procedure.

### 1.4.2. HEAP-EXTRACT-MAX

The root is swapped with the last element of the heap. The size of the heap is decremented by one, and the MAX-HEAPIFY method is called on the new root to preserve the heap property. Then, the old root is popped from the back of the array and returned. The time spent on this method is $O(log(n))$ due to the call for the MAX-HEAPIFY method on the root.

### 1.4.3. HEAP-INCREASE-KEY

If the new key is greater then the old one at the specified index, then the key is updated with the new key. Then, the HEAPIFY-UP method is called to preserve the heap property. The running time is $O(log(n))$ due to the preservation of the heap property.

### 1.4.4. HEAP-MAXIMUM

A reference to the root node, which is in the beginning of the array, is returned. The running time is $O(1)$.

## 1.5. Implementation of d-ary Heap Operations

A class called DaryHeap is implemented to serve the methods below. D-ary versions of heap operations such as MAX-HEAPIFY is implemented.

### 1.5.1. Height Calculation

Starting from the root, the height is counted by visiting the left-most child of the current node until reaching a leaf. Off-by-one errors are prevented by visiting the left-most child. The running time complexity is strictly $\Theta(log_d(n))$ since the each level of the d-ary tree must be visited.

### 1.5.2. EXTRAXT-MAX Implementation

Similar to the binary EXTRACT-MAX method, except DARY-HEAPIFY is called MAX-HEAPIFY. Thus, the running time is $O(dlog_d(n))$. The $d$ stands for the comparisons made between the current node and its children. Whenever d = 1, then the running time is $O(1log_1(n)) = O(n)$, and if d = n, then the running time is again $O(nlog_n(n)) = O(n)$. For d = 2, the running time is same as MAX-HEAPIFY and is $O(2log_2(n)) = O(log(n))$. Table 1.1 shows execution time measurements of the method on different $d$ values. According to the table, values of $d = 1$, and $d = n - 1 = 13806$ are showing a similar pattern.

| **d** | 1 | 2 | 4 | 16 | 256 | 1024 | 13806 |
|---|---|---|---|---|---|---|---|
| **Population4** | 487.842 | 1.062 | 852 | 1.082 | 5.300 | 20.139 | 265.303 |

**Table 1.1:** Comparison of different d values on the DARY-EXTRACT-MAX on input data. (in nanoseconds)

### 1.5.3. INSERT Implementation

Push the new element to the back of the array. Increase the size of the heap by one. Then, call DARY-HEAPIFY-UP method to keep heap property. The running time of the method is equals to the running time of DARY-HEAPIFY-UP method and is $O(log_d(n))$. DARY-HEAPIFY-UP method compares the node with its parent and swaps them id necessary until the root is reached. Since there is only one comparison between levels, there is no $d$ coefficient inside the time complexity expression.

### 1.5.4. INCREASE-KEY Implementation

Compare the old and new key. If the new key is bigger, then update the key with the new value. Then, call DARY-HEAPIFY-UP method to keep heap property. The running time is $O(log_d(n))$, due to keeping the heap property.

## 1.6. HeapSort vs. QuickSort

HeapSort and QuickSort take different approaches to sort objects. However, they both run in approximately equal time at the end. Table 1.2 shows the execution times of QuickSort and HeapSort. According to table, they both require roughly the same amount of time to sort the given input data. Indeed, the average running time of randomized QuickSort is $\Theta(nlog(n))$, whereas the running time of HeapSort is $\Theta(nlog(n))$.

| | **Population1** | **Population2** | **Population3** | **Population4** |
|---|---|---|---|---|
| **Randomized QuickSort** | 11.560.344 | 8.363.778 | 11.585.467 | 10.984.347 |
| **Binary HeapSort** | 9.123.343 | 11.284.451 | 10.331.873 | 11.696.193 |
| **4-ary HeapSort** | 10.206.748 | 11.059.643 | 8.264.736 | 11.190.667 |

**Table 1.2:** Comparison of execution time of QuickSort and HeapSort on input data (in nanoseconds).

On most cases, an efficient QuickSort implementation beats HeapSort. This is because comparisons made during the construction of the heap, and during the maintenance of it adds an overhead to the execution time. Since, QuickSort usually requires less comparisons, it performs slightly better than Heapsort.

Advantages of using QuickSort are in-place sorting and good average-case performance, and some advantages of using are again in-place sorting and good worst-case time complexity. An advantage of QuickSort over Heapsort is performing faster on smaller arrays. On the other hand, HeapSort has a better worst-case complexity of $O(nlog(n))$ over QuickSort which has $O(n^2)$.