ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 335E
ANALYSIS OF ALGORITHMS I
ASSIGNMENT II REPORT

STUDENT NAME	:	Fatma Sena Akçağlayan
STUDENT ID	:	150170085
ASSINGMENT DATE	:	17.12.2021

FALL 2021

Contents

1 Analysis of Heapsort Operations

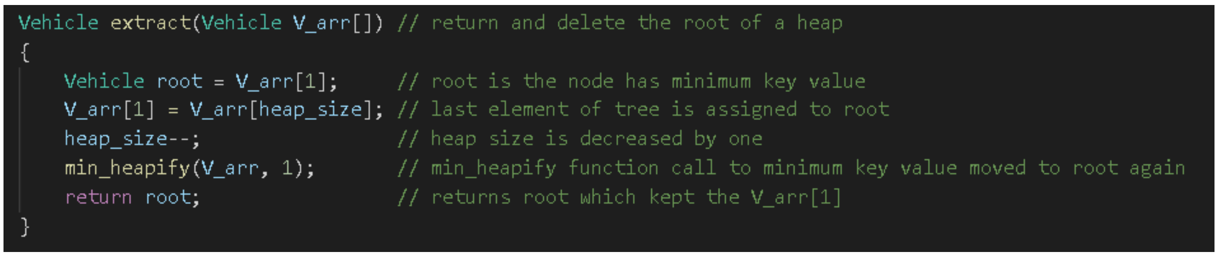
1.1 Extract

During the *Extract* operation, the root, that is, the smallest element, is returned. Then, the element in the last index of the heap is assigned to the first index that is root and the size of the heap is decreased by one. After this process, the $min\_heapify$ operation must be performed so that the element in the last index assigned to the root can reach the correct position.

For a heap with N elements, a maximum of tree height can be processed, which is equal to $logn$. In this process called $min\_heapify$, the key values of the right and left children of the root are checked. It is swapped as long as there is a child key value smaller then roots key value. As a result of the operation, the last index assigned to the root is placed in the correct position.

In *Extract* operation, processes other than heapify are constant time processes. So these processes can ignore when calculating time complexity. As a result, Extract operation takes $\Theta(logn)$ time.

The *Extract* function implemented for Assignment 2 appears in the Fig. 1. It has been implemented as the 1th indexed tree to make it more understandable. First index of heap is kept in *root* variable and returned. Before returning, the element in the last index is assigned to the first index and the $heap\_size$ value is decreased by one. The $min\_heapify$ function is then called.



```
Vehicle extract(Vehicle V_arr[]) // return and delete the root of a heap
{
    Vehicle root = V_arr[1];     // root is the node has minimum key value
    V_arr[1] = V_arr[heap_size]; // last element of tree is assigned to root
    heap_size--;                 // heap size is decreased by one
    min_heapify(V_arr, 1);       // min_heapify function call to minimum key value moved to root again
    return root;                 // returns root which kept the V_arr[1]
}
```

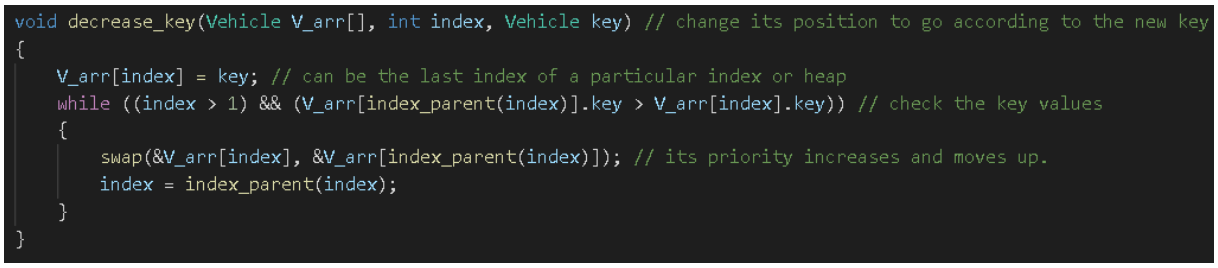
Figure 1: Extract Function Implementation

1.2 Decrease Key

In the $DecreaseKey$ operation, the key value of one of the elements is changed or decreased. Therefore, the tree should be $heapify$ according to the new key value. The element whose key value is changed should go to the correct position. To achieve this, it is checked whether the element is less than its parent values. If the parent key value is greater than the element's key value, the element and parent are replaced. This operation can be repeated up to the height of the tree, which corresponds to $nlogn$ operations.

It is very similar to the *heapify* operation, this time checking and relocating the tree from the bottom up. It already has the same time complexity as the *heapify* operation. Consequently, the worst time complexity is $\Theta(logn)$.

In the Figure 2, *decrease_key* function implemented for Assignment 2 can be seen. The new key value is assigned to the last element of the heap, then checked with its parent key values as mentioned above. As long as the new key value is less than the parent key values, it is swapped two elements. When the function is finished, the new key value is set to the correct position on min-heap tree.



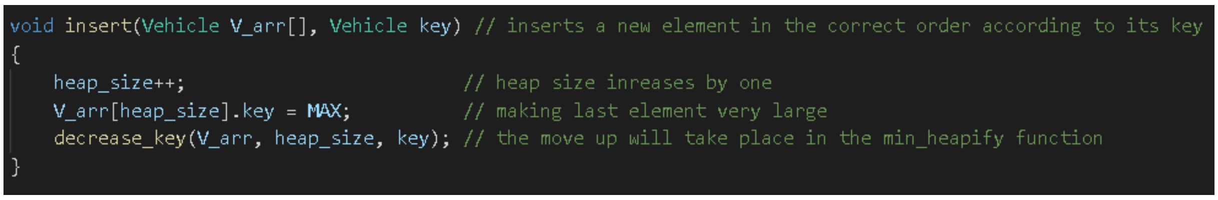
```
void decrease_key(Vehicle V_arr[], int index, Vehicle key) // change its position to go according to the new key
{
    V_arr[index] = key; // can be the last index of a particular index or heap
    while ((index > 1) && (V_arr[index_parent(index)].key > V_arr[index].key)) // check the key values
    {
        swap(&V_arr[index], &V_arr[index_parent(index)]); // its priority increases and moves up.
        index = index_parent(index);
    }
}
```

Figure 2: Decrease Key Function Implementation

1.3 Insert

Insert operation is used to add a new element into correct position on the heap according to the its key value. Since a new element will be added to the heap first, the size value must be increased by one. Since we are using min-heap, the key value of the last index is given a very large value. Then, with the *DecreaseKey* operation explained above, the new element is assigned to the last index and moved to the correct position. Since the *DecreaseKey* operation is called in the *Insert* operation, the time complexity is $\Theta(logn)$. Other actions such as increasing the heap size is ignored due to the constant time.

The Figure 3 shows the implementation of the *Insert* operation and the same steps is done as described above.



```
void insert(Vehicle V_arr[], Vehicle key) // inserts a new element in the correct order according to its key
{
    heap_size++;                          // heap size inreases by one
    V_arr[heap_size].key = MAX;           // making last element very large
    decrease_key(V_arr, heap_size, key); // the move up will take place in the min_heapify function
}
```

Figure 3: Insert Function Implementation

2 N – runtime Relation

	N=1000	N=10K	N=20K	N=50K	N=100K
average time	1,56ms	9,37ms	20,31ms	37,5ms	67,19ms

Table 1: Average running times for different N values

The N value entered from the command line determines how many *Extract*, *Decrease* or *Insert* operations will be performed. Average times calculated for different values of N (1000, 10K, 20K, 50K, 100K) are shown in the Table 1.

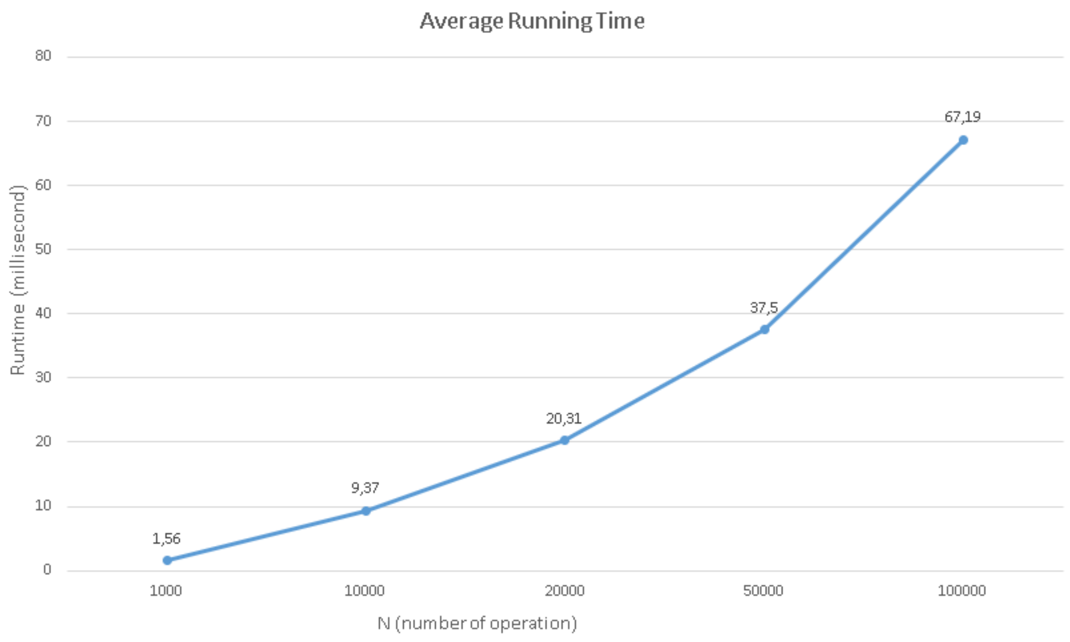


Figure 4: Average running times for different N values

In Part1, the time complexity of each of the *Insert*, *Extract* and *Decrase* operations was found to be $\Theta(logn)$. That is, when these operations are performed N times, it takes the $Nlogn$ time. This is the case for the general algorithm. In the homework implementation, since the Vehicle removed from the heap is added to the heap again, the logn value is the same at different N values (due to 1642). In the Figure 2, it is seen that the graph showing the average time values is similar to the linear graph although time complexity is $nlogn$.