BLG335E ANALYSIS OF ALGORITHMS I FALL 2021

ASSIGNMENT #2 HEAP REPORT

ŞERİFE DAMLA KONUR
040160433

CRN : 11458
Date of Delivery : 17.12.2021

1. INTRODUCTION

In this assignment, we are expected to implement minimum priority queue using heap structure keeping the vehicles in vehicles.txt, and for each request in requests.txt file we are expected to;

- If no lucky number → extract the vehicle in the root
- If there is a lucky number → first decrease the key value of the given vehicle in other words make it first priority vehicle then extract from the heap
- After the extraction update the vehicle's distance, location and estimated time values and insert to the heap again

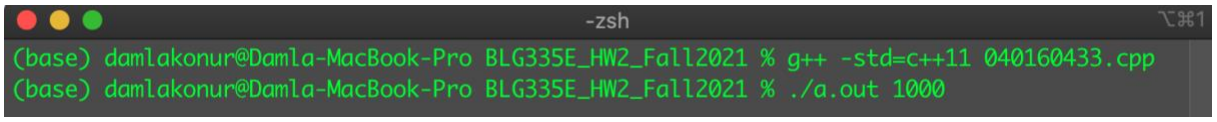
These above operations extract, insert and decrease are repeated untill the operation counter reaches the bound.

In my project, all the codes are written in 040160433.cpp file. Also the compilation and execution commands as follows :

Compilation : g++ -std=c++11 040160433.cpp

Execution : ./a.out numberOfOperations

Example :



```
(base) damlakonur@Damla-MacBook-Pro BLG335E_HW2_Fall2021 % g++ -std=c++11 040160433.cpp
(base) damlakonur@Damla-MacBook-Pro BLG335E_HW2_Fall2021 % ./a.out 1000
```

In the code, I created the Vehicle struct to hold vehicle data. This struct vehicle has members id, location, distance, speed and estimated time(timeToReach). In order to create min-heap I used zero-indexed vector of struct. Firstly, the input file "vehicles.txt" is read and each vehicle is stored in minHeapArr vector. The buildMinHeap function builds a min-heap from an unordered vector of struct according to the estimated time value of vehicles. After building the min-heap, for each request in requests.txt file I did necessary extract, insert, decrease operations. Also, in order to exclude I/O operation times from total execution time I created the call history vector and I stored the called vehicles' ids in it. Moreover, in order to exit the program when the N operations are executed I wrote a while loop, because in the each operation counter is increased by 1 program will run untill the number of operation bound.

2. Questions

Q- 1 : Write down the corresponding heap operations for the **extract, decrease and insert operations** in the implementation. Then, give the asymptotic upper bounds for each operation and explain these bounds with your own words.

For extraction I wrote the following function ;

```
struct Vehicle extractRoot() :
    1.  if(lastIndex < 1)                                 O(1)
    2.      then error                            O(1)
    3.  if(lastIndex == 1)                        O(1)
    4.      lastIndex --;                         O(1)
    5.      return minHeap[0];                    O(1)
    6.  root = minHeap[0];                        O(1)
    7.  minHeap[0] = minHeap[lastIndex];          O(1)
    8.  lastIndex--;                              O(1)
    9.  minHeapify(0);                                    O(lgn)
    10. return root;                              O(1)
```

This function returns the root (vehicle in node with index 0), it also takes the vehicle in the last index and puts in root. Then in order to maintain min-heap property the minHeapify function is called. There is no loop in the function and all operations except minHeapify takes O(1) time, in total T(n) = O(9) + O(lgn) and O(lgn) dominates other terms. If we look at the minHeapify function :

```
void minHeapify(int i):
    1.  r = right(i)                                                          O(1)
    2.  l = left(i)                                                           O(1)
    3.  min = i                                                       O(1)
    4.  if(l <= lastIndex and minHeapArr[l].keyValue < minHeapArr[min].keyValue    O(1)
    5.      then min = l;                                             O(1)
    6.  else    min = i;                                              O(1)
    7.  if(r <= lastIndex and minHeapArr[r].keyValue < minHeapArr[min].keyValue    O(1)
    8.      then min = r;                                             O(1)
    9.  if(min != i)                                                       O(1)
    10.     then exchange minHeapArr[min] minHeapArr[i];          O(lgn)
    11.     minHeapify(min);
```

This function maintains the heap property, In the pseudo code, there is no loop and number of

steps do not depend upon on N(number of node). This function takes parameter which is called index. This index represents the heapifyed index. So, the function put the ith element in the correct position. It looks key values of the left and right childs and if any of the left and right child's key value is smaller than than given index's value then swaps these two node. Lastly, minHeapify is called for current smallest element's index. The last part is recursion and it determines the running time. In the worst case is when the last row of the tree is half full on the left side and minHeapArr[i] is their ancestor. The subtrees of the children of our current node have size at most 2n/3. Also the running time of the minHeapify for a node is described with the heigth of the node. The running time of MAX_HEAPIFY can be described by the recurrence:

$T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$ with the help of the master theorem(case-2) T(n) = O(lgn)

For insertion I wrote the following function ;

```
void insertVehicle(Vehicle newVehicle):
    1.  lastIndex++;                                                       O(1)
    2.  i = lastIndex                                                      O(1)
    3.  minHeapArr[i] = newVehicle                                         O(1)
    4.  while(i!=0 and minHeapArr[parent(i)].keyValue > minHeapArr[i].keyValue)O(lgn)
    5.    then exchange minHeapArr[i] minHeapArr[parent(i)];         O(1)
    6.    i = parent(i);                                                   O(1)
```

This function inserts a new element to the heap. First, it increases the lastIndex value and puts the new element at the last index of the heap. After that, maintain the heap property it repeatedly compare the parent's key value and current element's key value untill the reaches the root or parant's key value is smaller than current element's key value. This while loop takes O(lgn) time.

For decrease key operation I wrote the following function ;

```
void decreaseKey(int i)
    1.  minHeapArr[i].keyValue = 0;                                   O(1)
    2.  while(i!=0 and minHeapArr[parent(i)].keyValue > minHeapArr[i].keyValue)
        O(lgn)
    3.    then exchange minHeapArr[i] minHeapArr[parent(i)];      O(1)
    4.    i = parent(i);                                              O(1)
```

The aim of this function is make the node in given index to first priority node. Therefore, I set the key value of the node in given index as 0. After that, in order to maintain the min-heap property it repeatedly compare the parent's key value and current element's key value untill the reaches the root or parant's key value is smaller than current element's key value. This while loop takes O(lgn) time.

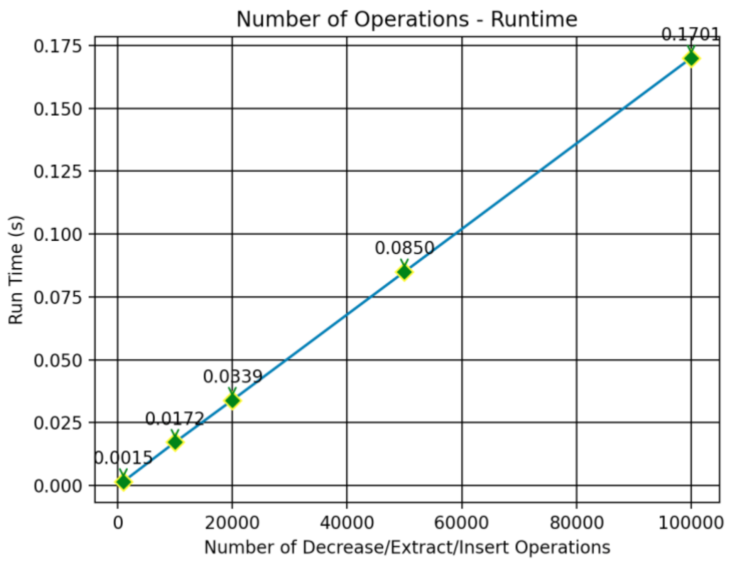
Operation	**Asymptotic Upper Bound**
Insert Node	O(lgn)
Extract Node	O(lgn)
Decrease Key	O(lgn)

Q- 2: For different values of N (1000, 10K, 20K, 50K, 100K) calculate the average time of multiple executions (at least 5 times). Report the average execution times in a table and prepare an Excel plot which shows the N – runtime relation of the algorithm. Comment on the results in detail considering the asymptotic bounds that you gave in Question 1.

Table 1. Relation between Number of Operations and Run Time

N	Rate of Increase	Running Time	Rate of Increase
1000		0.0015338 (s)	
10K	10	0.017173 (s)	11.196375
20K	2	0.033913 (s)	1.97479
50K	2.5	0.084976 (s)	2.5057
100K	2	0.170061 (s)	2.00128

In the assignment we are expected to the measure the running time for executing N operations. Also, the asymptotic upper bounds of the each operation is found in the previous question and these bounds are depend on number of nodes in the tree. But in this question total number of vehicle/number of node/height of the tree is constant therefore lgn is constant. As can be seen in above table, increasing ratio of the input values and increasing ratio of running times is so close. The relationship between number of operation and running time is linear O(N).


Number of Operations - Runtime
Run Time (s)
Number of Decrease/Extract/Insert Operations