ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 335E

ANALYSIS OF ALGORITHMS I
PROJECT 2 REPORT

CRN : 15175

DATE : 25.12.2020

NAME : ALI KEREM YILDIZ

STUDENT ID : : 150170013

FALL 2020-2021

Q1)

In this part of question, we will first explain used classes for storing taxis and taxi's information. In the source code files, it can be seen, there is a "Taxi" class which contains informations such that longitude, latitude and distance. According to given document, distance calculated with doing some operations on longitude and latitude attributes. Vector was used for storing this instances. In order to build the asked implementation, "MinHeap" structure was used instead of "MaxHeap". The reason for that is clerk calls the root node everytime because it is nearest. Which means taxi with smallest distance to hotel must be on the root. A class called "MinHeap" was created which contains Taxis in a vector and several operations. These operations explained below:

- ***getSize()***

⇨ Actually, it is not a fundamental Priority Queue operation. It is just emphasize that size of the heap stored on the "MinHeap" class as a private attribute which is initially has no value. With using getSize() function, according to its returning value some randomly determined processes will occur later on.

- ***insertTaxi(Taxi &newTaxi)***

⇨ This operation is a fundamental operation in Priority Queue. It takes only one argument that is taxi to add. Since the "TaxiVect" and this function are in same class, it is not necessary to give the "TaxiVect" as a parameter. In order to avoid some empty heap update operation and using the nearest taxi as a first index not zeroth index. After the checking attribute size and size of vector, a trivial Taxi which is Taxi(0,0) (*longitude*=0, *latitude*=0 it is the furthest taxi.) After that, its value assigned to newTaxi as last element. So far, new taxi added to heap but as the last element. We need to set it to a proper place with shiftUp() function. In this function the dominant process for running time is shiftUp() function call because other operations are O(1) which means they will omitted when finding the time complexity. Since shiftUp is O(logn) (we discuss it in shiftUp function), it dominates other operations. O(1)+O(1)+O(logn) means O(logn)

- ***shiftUp(int i)***

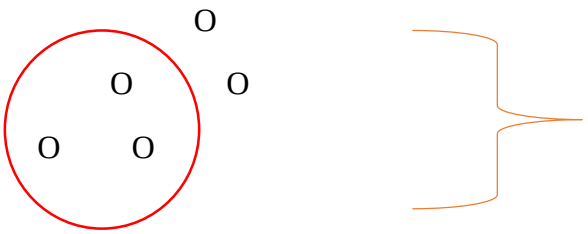
⇨ This is a one of the fundamental operation of Priority Queue. It works kind of a HEAP-DECREASE-KEY. In that function index i has passed as an argument. Unlike the heapify function, (in heapify we are going to check downwards, check parent is greater than childs or not) we check the elements with index i, whether they are smaller than its parent or not. The reason for that is new added taxis are added to last index of vector. Thus, it is needed to check parents of them. p(i) function gives the parent index. Via using that function it is checked. Then, if parent is greater, swap function is called. And then recursively call itself with parent index, because at the beginning Taxi has index i and after the swap operation it has index p(i). Therefore shiftUp(p(i)) recursively called until it reachs to root and we check if i is greater than size although it is an impossible situation. In this function since it is recursively call itself. In every call index i has divided by 2. Because it calls the parent index which means i/2. Therefore time complexity is O(logn).

- ***extractNearestTaxi()***

⇨ This is a fundamental Priortiy Queue operation too. It called when clerk calls a taxi which means after every 100 opearation according to given document. Until now, we know that vector is satisfy heap rules thanks to insert, shiftUp and heapify functions. (we will discuss heapify later) That means the element which has index 1 is root and it has the nearest taxi to the hotel. In order to extract it, we swap it with the last element and then decrease the size. That means now the first element is the furthest taxi and nearest taxi is not in the heap anymore, it has been extracted. Since root element is not nearest Taxi right now, we must call heapify function with index 1. In this function we have not any loop statement. Every process except calling heapify function is O(1). heapify() function is O(logn)(we will discuss it later.) It dominates other terms. So, just sum up the times: O(3)+O(logn) simply written as O(logn).

- ***heapify(int i)***

⇨ This is a fundamental Priortiy Queue operation too. In this function at first we assume that parent has smallest distance among their childrens with index i. “smallest” variable assigned to parent. Then it is checked whether it is smaller than its left and right child, we also check if it has got a child. (when l(i) or r(i) is greater than size, that means it has no such child). So, the smallest distance among parent and childrens is found. If parent has the smallest value, we do not need to do any swap or heapify operation anymore. But if one of its child has smallest value, we must swap the smallest children with parent and heapify it again recursively. This part determines the time complexity. The worst case of heapify is called by root node and the last row of the tree is half full. The subtrees of the children of our current node have size at most 2n/3.
For example :



Approximately 2/3 of the tree might be involved.

Now we can describe current situation as

$$T(n) \leq T\left(\frac{2n}{3}\right) + O(1)$$

According to Master Theorem which shown below[1]

Theorem 1 *The recurrence*

$$\begin{aligned} T(n) &= aT(n/b) + cn^k \\ T(1) &= c, \end{aligned}$$

where a, b, c, and k are all constants, solves to:

$$\begin{aligned} T(n) &\in \Theta(n^k) \text{ if } a < b^k \\ T(n) &\in \Theta(n^k \log n) \text{ if } a = b^k \\ T(n) &\in \Theta(n^{\log_b a}) \text{ if } a > b^k \end{aligned}$$

T(n) = O(logn). We could also describe the running time of heapify for a node of height h as O(h).

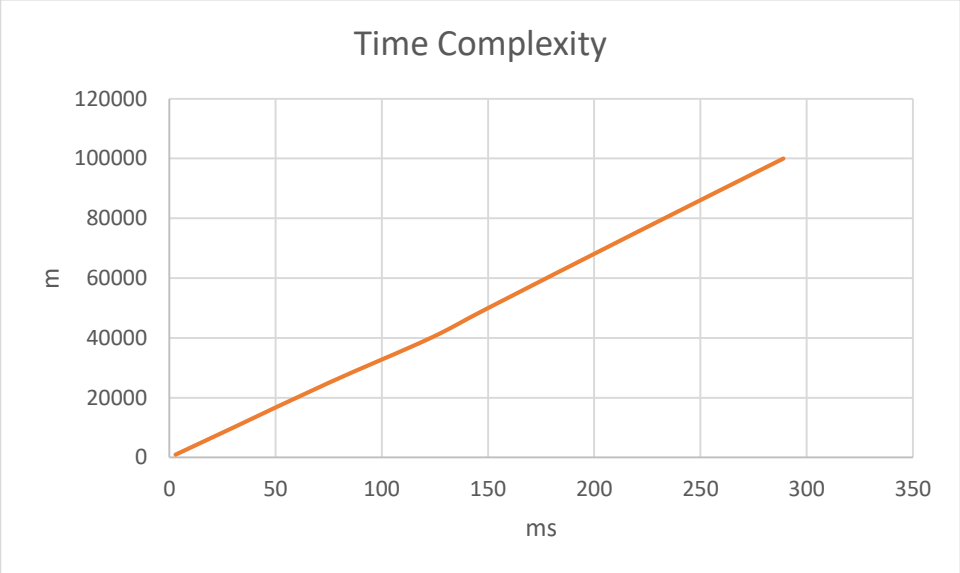
- ***updateTaxiLocation(int i)***

⇨ This is a fundamental operation too. It is similar to HEAP-DECREASE-KEY. In this function, we get the taxi's distance with given index i. Then according to document, decrease it by 0.01. Since we decreased the distance of a taxi, we need to check again whether it is smaller than its parent or not after the update it. We check it and if it has smaller value it is smalled with its parent. Until reach root or not smaller its parent, this process continues. We have a loop to execute which is important when calculating time complexity. Loop statement dominates other O(1) operations and we run at most logn times because in each loop we call the parent index which means i/2. Thus time complexity is O(logn)

Q2)

First, we will draw the table and then skectch the graph according to it.

p	m	run-time
0.2	1000	3ms
0.2	10000	30ms
0.2	25000	75ms
0.2	40000	123ms
0.2	50000	150ms
0.2	75000	219ms
0.2	100000	289ms

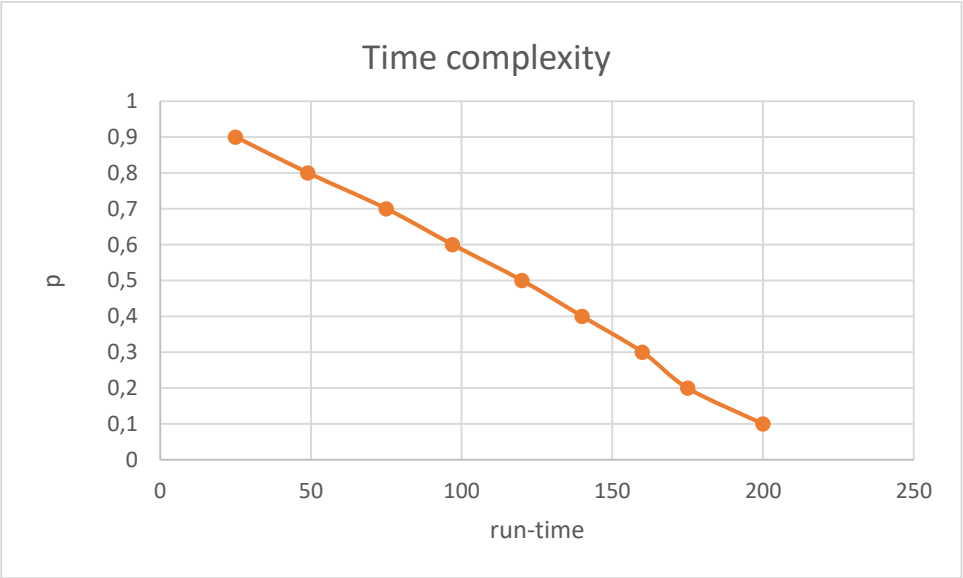


We observe that the Priority Queue operations takes O(logn). Since we perform n operations, we expect that run time is O(nlogn). In our example we except O(mlogm). It is upper bound but not tight upper bound because when we calculate the time complexity of functions, we assume the height-based operations' time complexity as O(logn) which means we consider all cases as root or leaf depends on operation. In brief we calculate as the worst case consideration. But since we are not making operations on root everytime, actually most of the time we will doing operations on the nodes which are close to leaf nodes. Therefore tighter upper bound can be evaluated for insert and update. As shown in graph with time complexity O(m). In brief, it does not match the theoretical running time.

Q3)

In this question, we need to observe the effect of p choice on run time. To demonstrate it better below table and graph created:

p	m	run-time
0.1	60000	200ms
0.2	60000	175ms
0.3	60000	160ms
0.4	60000	140ms
0.5	60000	120ms
0.6	60000	97ms
0.7	60000	75ms
0.8	60000	49ms
0.9	60000	25ms



The number of update and insert operations depends on p. When p is lower that means there are more update operation than insert operation and vice versa. Both operations depends on height of tree (logn). So if we do more insert operation that which means decreasing p, height of the tree will decrease but when p is increasing that means more update operation, height of tree will not increase. Thus, when decreasing, time cost increase a bit.

NOTE: NEXT PAGE

NOTE: I consider call taxi operation for counting which means

m = call_taxi + update + insert + exception

Where exception is doing update operation on empty heap. I am not considering it as update operation because it is not updating anything. In output format, it is not asked but i wrote this print line commented. If you want you can uncomment it and see how many time this situation exist.

References :

[1] https://randerson112358.medium.com/master-theorem-909f52d4364