BLG 335E – ANALYSIS OF ALGORITHMS I

ASSIGNMENT 2

Student Name: MUHAMMET ASLAN

No: 150160031

Deadline : 25/12/2020

PART 2

1-

min_heapify : This function takes two arguments. First argument is “array”, second is “index”. This function is used to heapify the element which position number is equal to the “index”. First, I took of children indexis of index. Then I defined a variable which named “smallest” and set it equal to the position of the “index” in the beginning. Secondly, I compared the children with their parent, if one of them smaller than the parent, the “smallest” variable is updated the current position. This operation is done for the both children. Then we check if the argument we take as “index” is equal to the smallest variable we found. If not, then we swap the them and call recursively heapify (with the array and the position of the smallest) the affected sub-tree.

delete_root : This function takes array as arguments. Then I threw the last element to the first element. Now our new root has become the last member. And we reduced the array size by 1. Then we made the tree suitable for heap again by sending this new root to the min_heapify function. Heapifying operation the tree again after deletion of root takes O(log(n)) time and all other operations take constant time.

shiftUp : Whenever we change the key of an element, it must change its position to go in a place of correct order according to the new key. If the new key is bigger than any of its children, then it is violating the heap property, so swap them. If there is no smaller value in children of the node, no operation is needed. If there are, we need to swap small values up to root of the heep which takes log(n) time for the worst case which equals to O(log(n)).

insert : This function adds a new element to heap. Then we just change the key of the element to call the shiftUp function to find the correct index. This operation's time complexity is O(log(n)).

Simulation Feature : Total operation is summation of update, addition and calling taxi. Running time contains these three operations and output of called taxis, total update and addition operations on the output screen.

2-

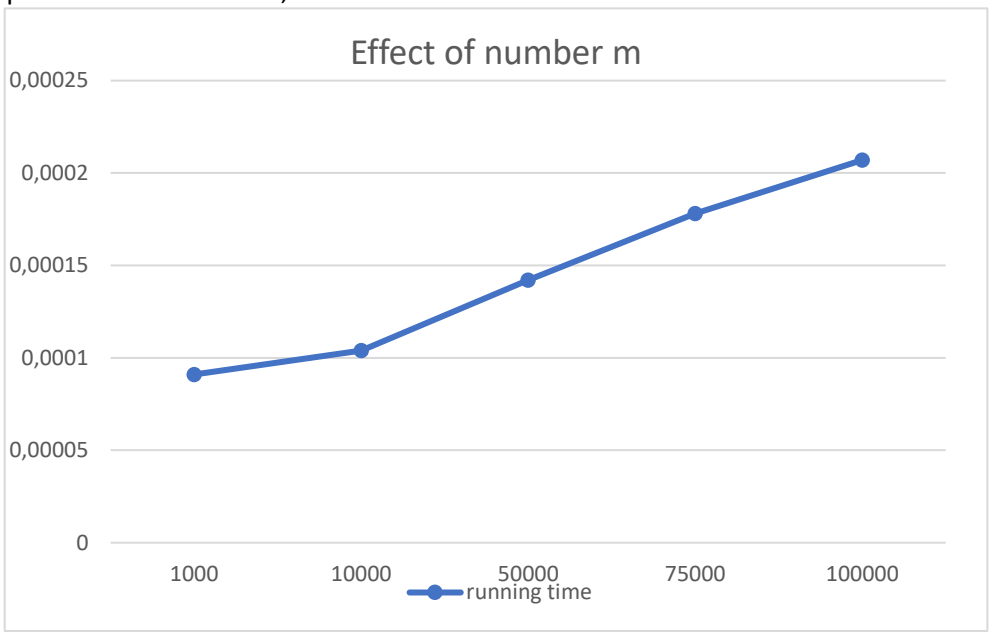
In the first case, we expected the runtime to increase as the number of m increased, because the depth of the binary tree increased as the m increased. It can also be seen that the running time increases while the number of m increases. Because time complexity is O (log(n)). So, the results substantially matched with the theoretical running time.

As the graph, running time increased with m.It is similar to O(log(n))

X directions are number of m.

Y directions are running time type in milliseconds.

p is constant which is 0,2.



3-

Running time is affected by number p. Because when the number p is small, the addition operation is most likely taking place. And this increases the depth of the heap tree. Increasing depth means increasing running time. In other words, increasing the p number while the m number was constant decreased the running time at each step because the possibility of an add operation new node to tree is lower. As the rate of adding new taxi to the tree decreases, the depth gradually decreases. And this reduces running time.

As shown in the graph, running time decreased with p.

X directions are p.

Y directions are running time type in milliseconds.

m is constant which is 100000.