Assignment 2 - Report

Muhammed Sefa Aydoğan

150150124

Part 2. Report

A) I implemented a PQ using binary heap data structure that supports the necessary operations for this assignment. My implementation for the homework and their explanations is below.

Since the heap is maintained in form of a complete binary tree, because of this fact the heap can be represented in the form of an array. So, I implemented a priority queue so that it stores its data in a specific order according to the value of the elements. So, inserting a new data must go in a place according to the specified order. This is what the insert operation does. The entire point of the priority queue is to get the minimum value and the 'extract_minimum_value' does this job. My order of the parents, left child and the right child is shown in Figure 1.

```
27  int get_left_child(int i) // left child can be accessed by 2*i
28  {
29      if((2*i) < PQ_array_size))
30          if(i >= 1)
31              return 2*i;
32      return -1;
33  }
34  int get_right_child(int i) // right child can be accessed by (2*i)+1
35  {
36      if(((2*i)+1) < PQ_array_size))
37          if(i >= 1)
38              return (2*i)+1;
39      return -1;
40  }
41  int get_parent(int i)  // One element at index i in array, then its parent will be stored at index i/2. Index of root will be 1 in an array.
42  {
43      if((i > 1))
44          if(i < PQ_array_size)
45              return i/2;
46      return -1; //if its a root, root has no parent
47  }
```

Figure 1

What this function does and the order is commented on the code.

```
bool flag;
int update_counter = 0;
int addition_counter = 0;

for(int i = 0; i<m; i++){
    flag = randomize(p); // if randomize function returns false, operation is update.
    if(!flag)
    { //If an operation is an update, a random taxi's distance value will be decreased by 0.01.
        if(N >0) // if heap has elements in it which means does not empty.
        {
            int random_value = rand() % N;
            update_value(PQ,random_value);  //  Time complexity is O(N)
            update_counter ++; //Counter is increased because we need to know how many times does my program updated an information from the queue.
        }
    }
    else if(flag)
    { //If an operation is an addition, new information will be read from the provided file, and the distance value of the new taxi will be added to the PQ.
        file >> longitude;
        file >> latitude;
        float distance = distance_to_hotel(longitude,latitude);
        insert(PQ,distance); // For N values, Time Complexity will become O(NlogN). N is the heap size.
        addition_counter++; //Counter is increased because we need to know how many times does my program added a new information to the queue.
    }
    if((i+1)%100==0)
    {
        cout << "The distance of the called taxi : " << extract_minimum_value(PQ) << endl;
    }
}
```

Figure 2

My main calls for the functions is shown in Figure 2. Their jobs are commented on the code. With the given p value, I implemented a function called "randomize" and it produce a random value according to the p value. If p value is false, it means that operation is an update, if it is true, operation is addition. Update operation time complexity is O(1) as it can be seen in Figure 2 and how many times it runs is shown with the 'update_counter' variable. It runs "update_counter*O(1)". So, its time complexity is become O(N).

```
118  void update_value(float PQ[], int random_value) // A random taxis distance to the hotel will be decreases 0.01
119  {
120      PQ[random_value] -= 0.01f;
121  }
122
```

Figure 3

If the 'flag' is true, operation is addition. After reading data from the file, distance value of the readed taxi is added to the PQ with the 'insert' function. 'Insert' and 'decrease_value' functions can be seen in Figure 4 and 5.

```
96   void insert(float PQ[], float value) // Complexity is O(Log N). Because decrease_value function's complexity is O(Log N).
97   {
98       N++;
99       PQ[N] = INFINITY;
100      decrease_value(PQ, N, value);
101  }
102
```

Figure 4

```
86   void decrease_value(float PQ[], int index, float value) // Complexity is O(Log N).
87   {
88       PQ[index] = value;
89       while((index>1) && (PQ[index] < PQ[get_parent(index)])) // if the current nodes value is bigger than its parents,
90       {
91           swap(&PQ[index], &PQ[get_parent(index)]); // They have to be swapped.
92           index = get_parent(index);
93       }
94   }
95
```

Figure 5

Because the time complexity of the "decrease_value" function that is called from the insert function, my insert function time complexity is O(logN). I called it N times from my main function thus my total time complexity is O(NlogN). What functions do is commented on the code.

In my main for loop for the simulation, in every 100 operations, "extract minimum_value" function is called. This function is shown in Figure 6. This functions job and time complexity

is commented on the code. It calls “min_heapify” function and this function is shown in Figure 7. This functions job and time complexity is commented on the code.

```
77  float extract_minimum_value(float PQ[]) // Complexity is O(Log N). Because min_heapify function's complexity is O(Log N).
78  {
79      float minimum = PQ[1];  // Index of root will be 1 in an array according to the our design.
80      PQ[1] = PQ[N]; // Last element of the PQ will be the first element of the PQ so that we can heapify after the removal of the smallest number.
81      N--; // number of elements in the heap will be decreased by one.
82      min_heapify(PQ, 1);
83      return minimum;
84  }
```

Figure 6

```
57  void min_heapify(float PQ[], int index){    // Complexity is O(LogN). Goal is to find smallest value among index, left child and right child
58      int right_child_index = get_right_child(index);
59      int left_child_index = get_left_child(index);
60
61      int smallest = index;
62
63      if ((left_child_index <= N) && (left_child_index>0) && (PQ[smallest] > PQ[left_child_index]) )
64          smallest = left_child_index;
65
66      if ((right_child_index <= N) && (right_child_index>0) && (PQ[smallest] > PQ[right_child_index]) )
67          smallest = right_child_index;
68
69      // If smallest is changed in the previous steps, then smallest element and the element that index shows has to swap
70      if (smallest != index)
71      {
72          swap(&PQ[index], &PQ[smallest]);
73          min_heapify(PQ, smallest);
74      }
75  }
```

Figure 7

B) I run the simulation for different values of m between 1000 and 100000 for a constant p (0.2). I used the values that are shown in Table 1.

‘m’ Values	Running Time
1000	0
10000	10
20000	21
30000	31
40000	41
50000	50
60000	67
70000	73
80000	88
90000	96
100000	112

Table 1

I used this values to draw my graph. My graph is shown in Figure 8. As I expected, I get a graph that increases linearly. My programs time complexity is O(N*logN). So, according to the ‘Big-O Complexity Chart’ in Figure 9(I took it from the course slides), graph that I obtained is act as what I have expected. It matched with the theoretical running times, almost. I run my program 10 times for every m values and got average running time value of them.

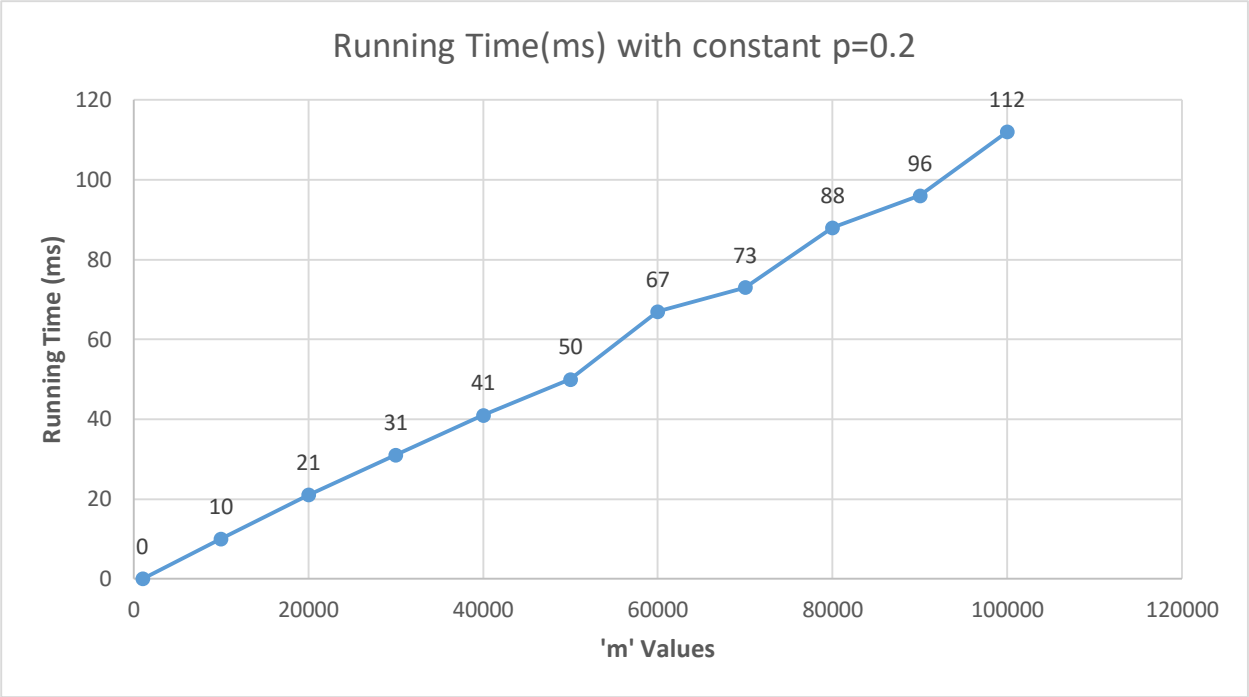

Figure 8

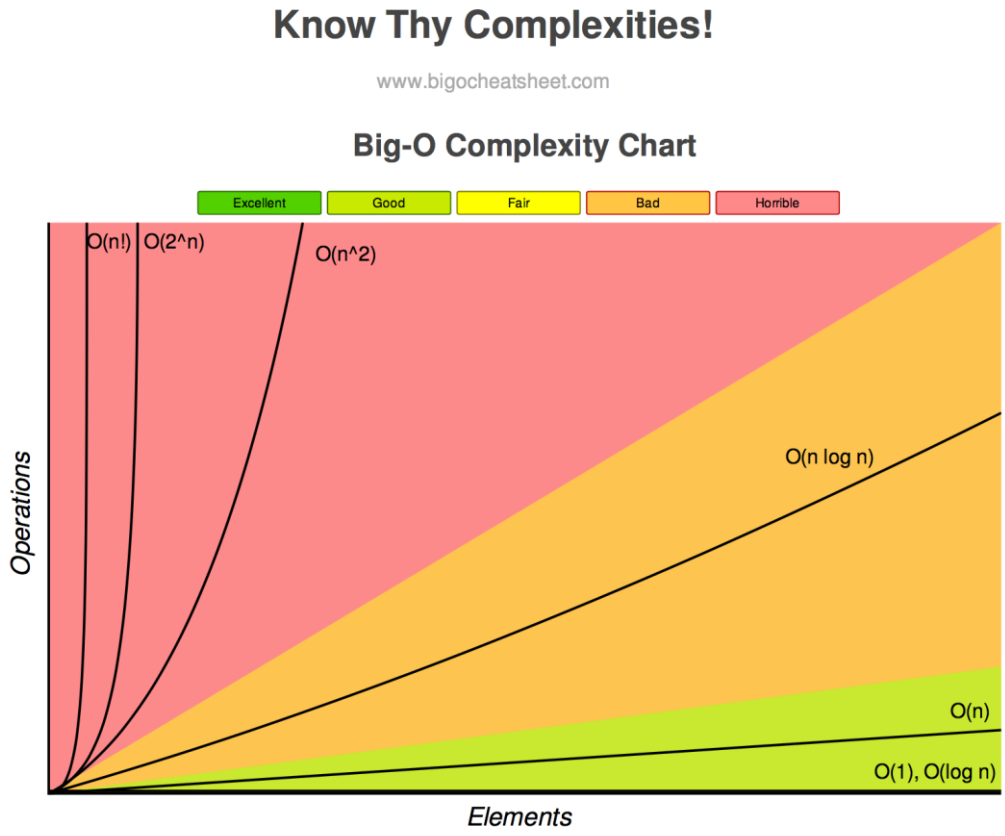

Figure 9

C) I run the simulation for different values of p {0.1, 0.2, ..., 0.9} for a constant m equals to 100000. According to the founding, I draw the graph in Figure 3. Running time is affected by p. Because, as it can be seen a piece of my program in Figure 11, according to p values,

probability of an operation is an update is increased for every increment in the p value. Because of the time complexity of the update operation is O(N), If probability of the update operation increases, my program will run faster and running time will decrease. Because when p has a small value, the probability of running the insert function is bigger and its time complexity is O(NlogN); therefore it runs slower when the leading operation is addition. As it expected, it happened exactly like this in my simulation. I run my program 5 times for every p values and got average running time value of them.

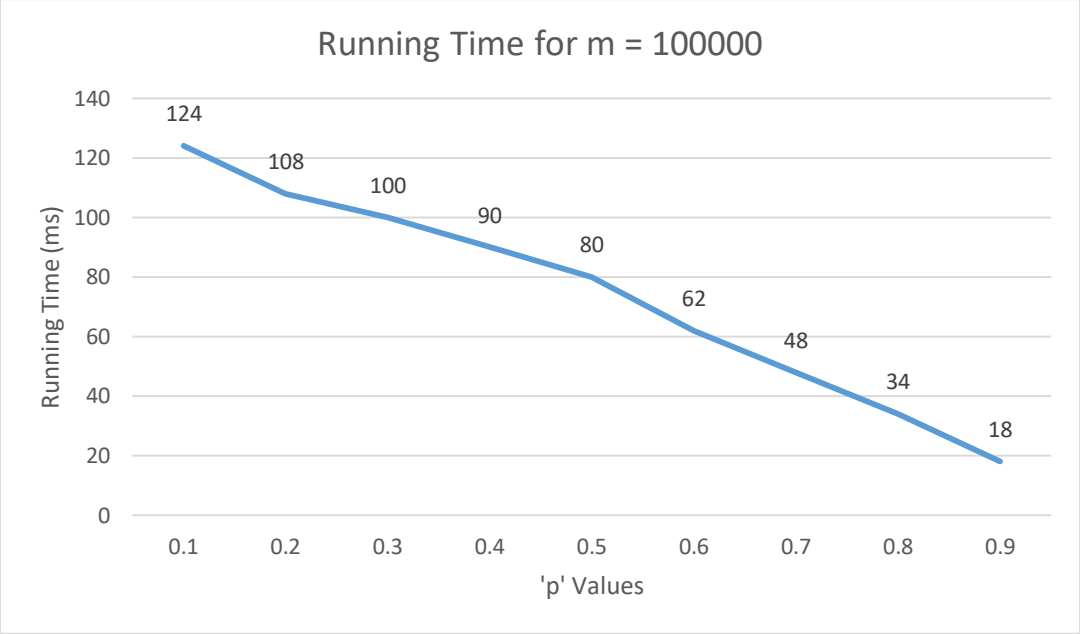

Figure 10

```
int randomize(float p) // update with a probability of p or an addition of a new taxi with a probability of 1 - p.
{
    p = p*10;
    float a;
    a = rand() % 10 + 1;
    if(a > p) // addition
        return true;
    else if(a <= p) // update
        return false;
}
```

Figure 11