BLG 335E – Analysis of Algorithms I
Fall2020 Homework 2 Report
Oguzhan Karabacak 150170021

1.

In this homework, we build a priority queue(PQ) for a taxi application simulation using the binary heap data structure. For Priority Queue operations that we built using binary heap tree, we write 4 functions except swap function.

These are min_heapify, decrease_key_value, extract_min, and insert functions.

Let's start with min_heapify first

1.1 min_heapify() function

```
//min_heapify function, Vector array, i value, Vector array size
void min_heapify (vector<double> &distances , int i, int N)
{
   double left = 2*i+1; //left child
   double right = 2*i+2; //right child
   double min;

   //if left is smaller than N and left element is smaller than ith elements ,
      min is equal left child
   if(left < N && distances[left] < distances[i] ) min = left;
   else min = i; //else min is equal i;

   //if right is smaller than N and right element value is smaller than min
      value
   if(right < N && distances[right] < distances[min] ) min = right;

   if(min != i) //if min is not equal to ith index
   {
       swap (&distances[i],& distances[min]); //swap parent and child
       min_heapify (distances, min,N); //rebuild again binary heap tree
   }
}
```

The min_heapify function is not a function we use directly inside the main function.
We use the min_heapify function to make the array a binary heap tree when we extract an element from the array.
The min_heapify function takes 3 parameters. These are Vector array(&distances), parent element's index(i) and Vector array size(N).
First we assign the indexes of the children of this parent element to the left and right variables and create a min variable to keep the index of the minimum value.

First, we compare the index of the left child with the array size to check whether the parent element has a left child. Also, if there are left children we take the value of it and compare it with the value of the parent element.
Because in a min binary heap tree, parent must always have a smaller value than its children.
If the value of the left child is smaller than the value of parent, the new min value is the index of the left child. If not, the value of min is parent.
We repeat the same process for the right child. We compare the index of the right child with the size of the Vector array and compare the value of the parent with the value of the right child.
If the index of the right child is smaller than the size of the Vector array and the value of the right child is smaller than the value of its parent, our new min value is the index of the right child.
At the end of the block, we check whether the min value is equal to the parent index.
If it is equal, it means that the tree is a binary heap tree, but if not, if the min value is equal to one of the children's indexes, here we first swap the parent and the child and call the min_heapify function with the new min value using recursive.

Time Complexity

The time complexity of all operations up to the last operation in the code block is O(1) because we are only doing a comparison or assign operations.
But since the min_heapify function at the end of the code is a recursive function, its time complexity is not a constant number.
The height of binary heap tree determines time complexity in the recursive function here.
Considering that the Vector array has n elements and each parent has 2 child, then time complexity is logn.
So the complexity of the min_heapify function is logn.

1.2 decrease_distance() function

```
//decrease_distance function , ith elements, new_distance is new value
void decrease_distance(vector<double> &distances,int i,double new_distance) {
   //if new _distance is bigger current_distance
   if(new_distance > distances[i]) {
      cout << "new distance is bigger than current distance" << endl;
      return;
   }
   distances[i] = new_distance; //new_distance -> distances[i]
   if(distances[i]<0) distances[i]=0; //set to zero if distance is negative
   //compare ith elements value and its parent until the root
   while (i != 0 && distances[ (i-1) / 2] > distances[i] ){
      //swap ith element and its parent
      swap(&distances[(i-1)/2],&distances[i]);
      i = (i-1)/2; //new i value is i's parent
   }
}
```

The decrease_distance function is a function that we use for the update operation in main function. decrease_distance function takes 3 parameters.
The first parameter is Vector array(distances), the second parameter is which element's value will be updated (i), and the third parameter is the new value of the element (new_distance).
Since we do not want the update value to be bigger than the old value in the update operations, we first check whether the update value is bigger or smaller than the old value.
If the update value is bigger than the old value, we finish the function.
If the update value is smaller than the old value, we assign the update value to the index of the old value.
Then we check minimum value is negative, actually the distance cannot be negative but If we do too many updates over the same distance, the value may be negative, so we check if distance is negative,we assign 0 to distance.
After the new value has been assigned, we find the location of the update value in the tree to make the Vector Array a Binary Heap Tree again.
We use the while loop for this and do two checks.
The first check is whether the i(update value's index) value is different to 0, that is, not root and the updated index's parent value is bigger than the updated value.
If the updated index is not equal to 0 and the updated value parent is smaller than its parent, we swap the place of the updated value with the its parent and then we assign parent index to the i value. This process continues until root.

Time Complexity

The time complexity of operations up to while loop in the code block is O(1) because we are only doing a comparison or assign operations. But since the time complexity of while loop is not constant.
If the Vector array has n elements, the depth of a binary heap tree is O(logn) so the time complexity of this function is O(logn).

1.3 extract_min() function

```
double extract_min(vector<double> &distances,int N){ //function to delete root
   if (N == 0) { //if array is empty
      cout << " heap underflow " << endl;
      return -1; //exit function
   }
   double min=distances[0]; //distances[0] -> min
   distances[0] = distances[N-1]; //last element -> distance[0]
   distances.pop_back(); //delete last element
   min_heapify(distances,0,distances.size()); //rebuild binary heap tree
   return min; //and return min
}
```

The extract_min() function is used to extract the smallest element from the binary heap tree, in other words the root of the tree. extract_min() function has 2 parameters Vector array and the size of this array.
First we check if the array is empty. If the array is empty, it is a heap underflow and the function returns -1. If array is not empty. We assign the value of the root of the tree to the min variable, ie its minimum value.
Then, we assign the last element of the array to index 0 and remove the last element from the array.
After doing this, we call the min_heapify function to rebuild the binary heap tree structure. And min_heapify parameters are Vector Array, the roof of tree and new array size.
At the end of the Code Block, we return the min value.

Time Complexity

The time complexity of operations up to min_heapify function in the code block is O(1) because we are only doing a comparison or assign operations.
But since the time complexity of min_heapify is not constant.
As we calculated above, Time complexity of min_heapify is O(logn) so the time complexity of extract_min() function is O(logn)

1.4 insert() function

```
void insert(vector<double> &distances,double key,int N){ //insert function
   distances.push_back(key); //first add the last of array
   decrease_distance(distances,N,key); //then update the tree
}
```

The insert() function is used to insert new element to array.This function has 3 parameters: Vector Array , value which will be inserted and Array size.
The insert function is very short.
First we add the new value to Vector array then we call decrease_distance() because the new value(key) find the place in Binary heap tree.

Time Complexity

First line has constant time complexity and in second linet here is decrease_key_value() function and its time complexity is logn so time complexity of insert() function is O(logn).

1.5 Loop in Main function

```
for (int i=0;i<m;i++){
  file >> longitude; //get longitude
  file >> latitude; //get latitude
  double euclid=pow( pow(longitude-hotel_long,2) +
      pow(latitude-hotel_lati,2 ) , 0.5); //apply euclid
   //and determine the type if what_operation = 0, operation is insert,
       else update
  what_operation=(rand()%100) < p*100;
  if(what_operation==0) { //insert operation
     insert(distances,euclid,distances.size()); //insert to array
     insert_counter+=1; //increment insert_counter
  }
  else{
       // if array is empty , start loop again
     if(distances.size() == 0) continue;
      //select random index to update
     update_random=rand() % distances.size();
       //call decrease_distance , distances[update_random]-0.01 : Random
          element minus 0.01
     decrease_distance(distances,update_random,distances[update_random]-0.01);
     update_counter += 1; //increment update_counter
  }
  if((i+1)%100 == 0 && i != 0){ //if i is 100x , and i is not equal to zero
     double extract_value=extract_min(distances,distances.size());
     if (extract_value != -1){
        delete_distances.push_back(extract_value); //add returns of
           extract_min function to delete_distances array
     }
  }
}
```

Most of our simulation operations are done in the loop block in Main.
First we get the longitude and latitude values of the taxi from the file, then we apply the Euclid process by using the longitude and latitude values in the hotel. This gives us the distance between the hotel and the taxi.

Then we pick a random number between 0 and 100, then compare this number with p * 100 and assign it to what_operation variable.

If the number is small, the variable what_operation takes a value of 0 if not 1.
Here, if what_operation equals 0, it means that we will make an insert. In the insert operation, we call the insert () function first, then we increment the insert_counter by 1.

If what_operation is equal to 1, it means we will do an update process.For this, we first check if the Vector Array is empty, if it is empty, the loop returns to the beginning. If it is not empty, we select a random element from the array and call decrease_distance().

Here we give the Vector array as the parameter, the index of the element and the new value less than 0.01 as the old value, then we increment the update_counter by 1.

At the end of the loop block, we check whether the number of operations is multiple of 100 and whether the operation is the first operation.

If it is not a multiple of 100 and the first action, we subtract the minimum value from the Vector Array and check whether the return value is -1. If not, then we add the extract_value to the delete_distances vector Array.

After the loop block, we print the delete_distances value on the screen with the help of another loop.

Time Compexity of Simulation

In general, the time complexity of the 4 functions we use for PQ operations is logn.

These functions usually worked for just one element. For example, we insert one element, extract one element, or update one element.

However, in Loop in Main function, we call these functions for each element. Therefore, since the number of elements in the operation is n, the time complexity of the entire simulation is O(**nlogn).**

2. In simulation m operations: insert + update operations

In this part i ran my code for different values of m for a constant p(0,2). This m values are 1000,2000,5000,10000,20000,50000,100000.

I ran the simulation 10 times for each value of m to make the results more accurate and i used average value for each value of m. The all results are in the table 1. Result's unit is milisecond.

N	1000	5000	10000	20000	50000	75000	100000
1	1,88	8,99	14,44	28,89	75,51	106,97	141,422
2	1,6	8,05	14,16	30,05	68,44	109,84	146,12
3	1,73	7,36	15,76	29,66	77,46	109,4	140,2
4	1,55	7,37	14,01	31,71	68,41	105,37	170,45
5	2,06	7,44	13,92	29,96	68,02	108,65	151,37
6	1,61	7,08	13,88	28,76	72,81	106,39	164,73
7	1,6	8,15	15,21	27,8	69,7	112,74	138,1
8	1,53	7,34	14,31	27,9	71,8	105,57	160,63
9	1,6	7,06	14,62	30,7	76,04	102,12	150,53
10	1,58	7,22	15,26	28,6	77,54	108,36	145,85

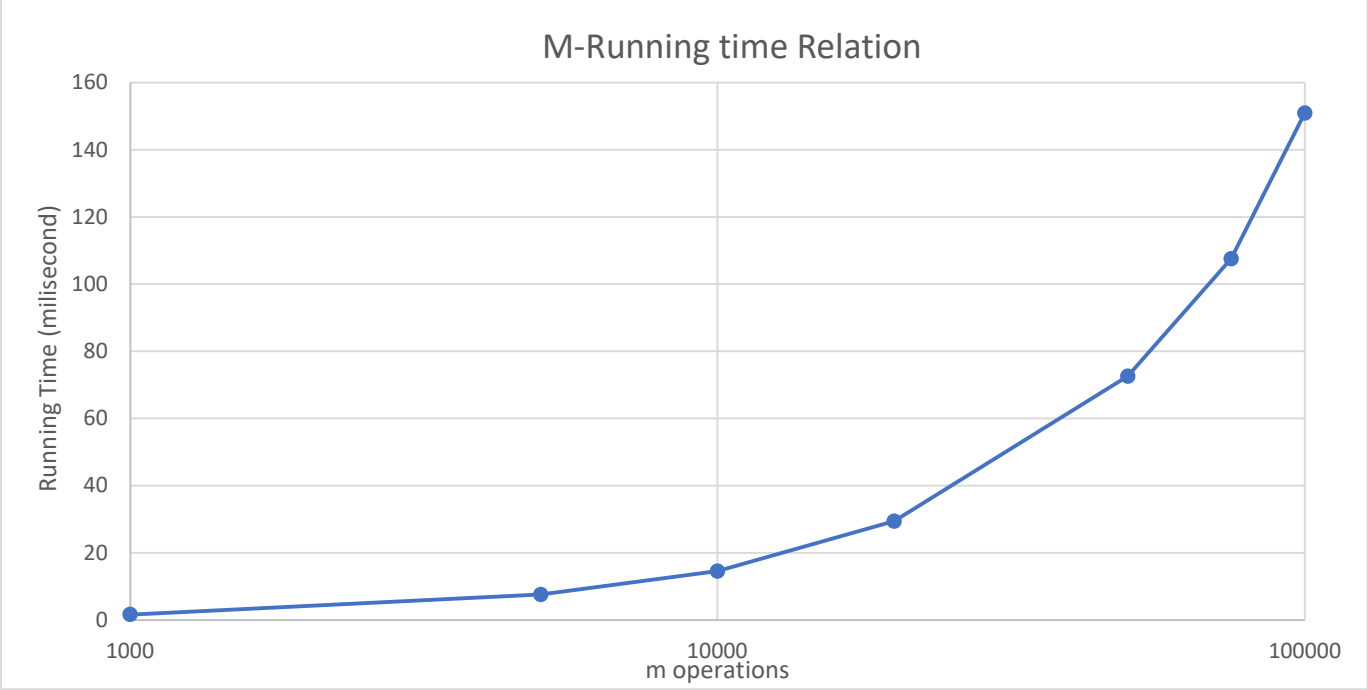
(Table 1: Result's unit is milisecond)

The average results are in the table 2.

M	Running Time
1000	1,674
5000	7,606
10000	14,557
20000	29,403
50000	72,573
75000	107,541
100000	150,9405

(Table 2:Average Results)

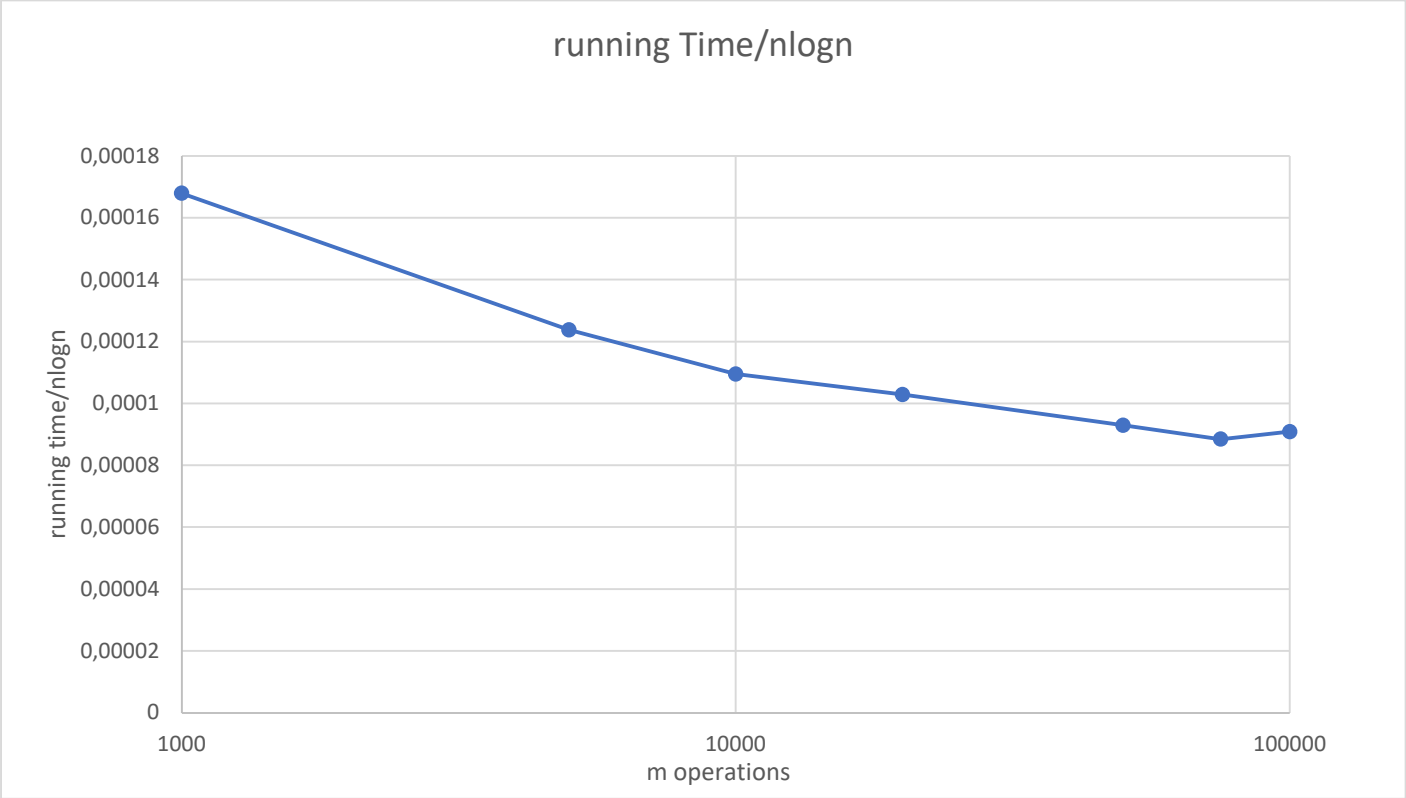
And graph of this results is :



Yes, the graph came as expected. In theory I was expecting the nlogn graph and the graph generated according to the experiment results in the simulation looks like a nlogn graph. It is better to look at the running time/nlogn graph for each m value to see the result better.

M	running time/nlogn
1000	0,000167976
5000	0,000123799
10000	0,000109552
20000	0,000102896
50000	9,29848E-05
75000	8,84678E-05
100000	9,08752E-05

And graph of this table:



When we look at the runnign time/nlogn graph, we can say that the similar results are obtained for m values. (The first m values are slightly different.)

3.

In this part, when we think theoretically, as the p value, in other words the update operation probability increases, the insert operation probability decreases because it has a 1-p value.

The lower the probability to insert operation , the fewer elements we will add to the array, and therefore the smaller the size of the array. The small size of the Array means that the binary heap tree has lower height.

Since the time complexity of travelling on a binary heap tree is logn, the running time is shorter because the binary heap tree has lower height so **the running time is affected by p value.**

For example, when m = 100000, let's compare p = 0.1 with p = 0.9.

In the case of p = 0,1, approximately 90000 is added to the locations array and there are 10000 update operations.

In the last case the number of elements of the array is 90000 and the height of the binary heap tree is log90000 (≈ 16,458).

When p = 0.9, approximately 10000 locations are added to the array and 90000 updates will occur, and the number of elements of the array will be 10000 and the binary heap height will be log10000 (≈ 13,288).

As we can see, log10000 value is smaller than log90000

In the experiment, i ran my code for different values of p for a constant m(100000).This p values are 0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.7,0.8,0.9.

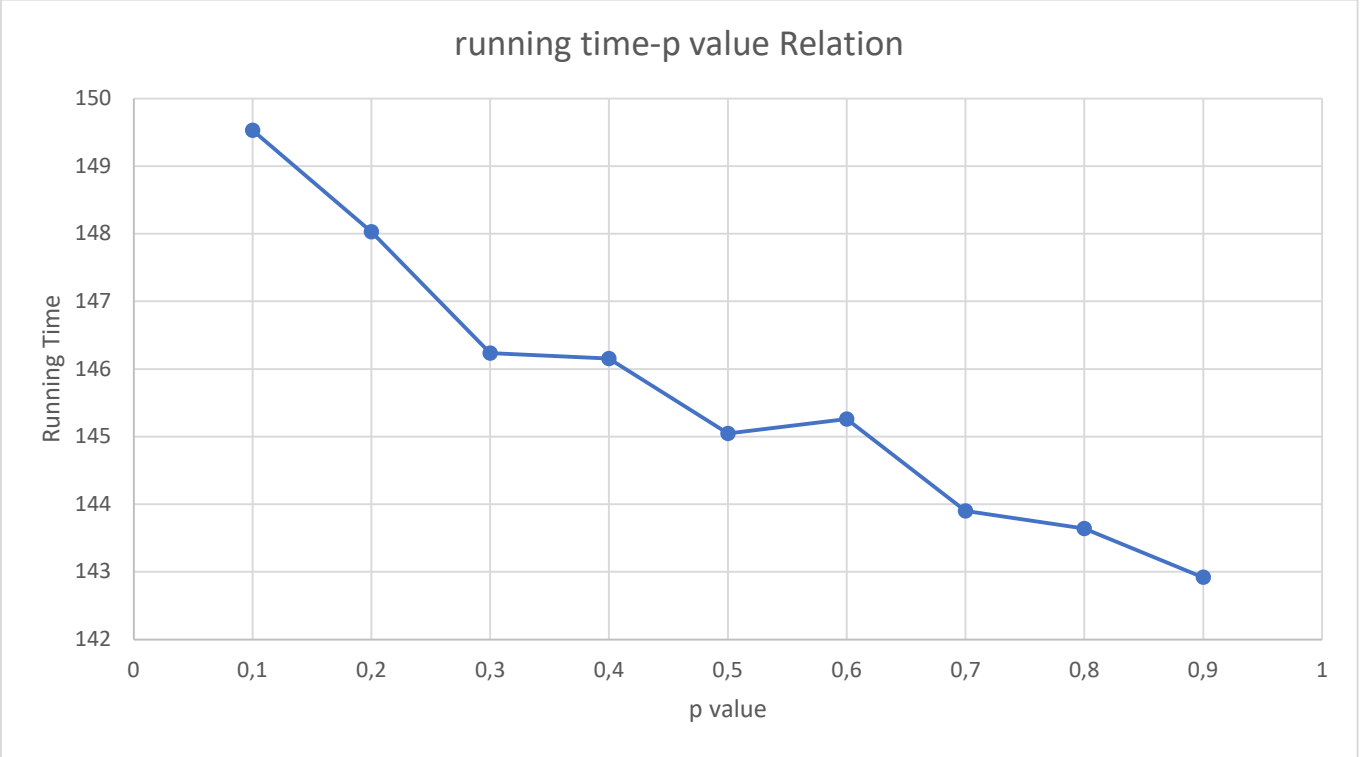
I ran the simulation 5 times for each value of p to make the results more accurate and i used average value for each value of p. Result's unit is milisecond.

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
1	151,17	141,4	141,24	142,88	152,35	149,58	148,96	138,444	150,14
2	140,37	146,53	141,63	152,56	137,44	140,4	143,52	142,4	135,65
3	151,15	147,69	142,2	140,57	154,02	145,52	150,62	140,437	132,58
4	153,26	153,95	150,18	149,74	139,44	140,3	143,26	141,375	143,94
5	162,712	150,58	155,92	145,02	141,99	153,493	143,13	155,54	142,28

And the average of this values:

p value	running time
0,1	151,7324
0,2	148,03
0,3	146,234
0,4	146,154
0,5	145,048
0,6	145,8586
0,7	145,898
0,8	143,6392
0,9	140,918

And the graph of the average values :



When we look at the experiment separately for each p value , we see that some p values(p=0.6) theoretically do not work correctly because we expect running-time to decrease continuously as the p value increases.

But when we look at the graph in general, we see that when the p value increases, running-time decreases in general.

In summary, the p value affects the size of the array, which affects the binary heap height, so **the p value affects the running time**.