Name: Başar Demir

Student Number: 150180080

BLG 335E – Analysis of Algorithms I

HOMEWORK-2

1) I have implemented priority queue using heap data structure on **array**. In priority queue, we need fundamental functions that handle some operations such as **adding** new element, **updating** existing element and **getting** top element. All required functions were implemented based on minimum-heap structure functions.

The implementation logic of the **minHeapify**, **addTaxi**(addition operation), **decreaseTaxiDistance**(update key), **getNearestTaxi**(gets top element) is given below.

As a last bullet of this part, I have described the simulation properties and main function of my code.

a. **minHeapify Function**

Description:

Minimun Heapify is a function that preserves heap property of the array. Its fundamental logic based on looking value of element that's index is given as a parameter to function and it compares this value with its children. Heap structure is represented in binary tree format. Therefore, it checks only right and left children of node. It finds smallest node and if it is not current node, it locates this smallest element to current node's position by swap operation and it calls itself with new position of element.

Complexity:

- We have to decide smallest node for each function call and it works at constant time.
- We have to call function recursively. Tree with half-filled leaves causes worst case condition. For each recursion, we can decrease size of the subtree to $\frac{2n}{3}$. Therefore, we are calling min-heapify for $\frac{2n}{3}$ sized part and it costs $T\left(\frac{2}{3}n\right)$.
- Final recurrence function is:

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

While determining the time complexity of the cases, I use Master Theorem that is given below. If the recurrence function is

$$T(n) = a * T\left(\frac{n}{b}\right) + O(n^d)$$

time complexity of our algorithm:

$$T(n) = \begin{cases} O(n^d \log(n)), & if\ a = b^d \\ O(n^d), & if\ a < b^d \\ O\left(n^{\log_b(a)}\right), & if\ a > b^d \end{cases}$$

We can observe that **a=1, b=3/2, d=0** in our recurrence function. Thanks to Master Theorem, the time complexity of the worst-case Min-Heapify is,

$$\boldsymbol{T(n) = O(\log(n))}$$

```
//MinHeapify function of PriorityQueue
void PriorityQueue::minHeapify(int index) {
    int smallest_index = index; //keeps index of smallest node
    double smallest_element = this->taxis[index]; //gets current node's
value
    unsigned left_index = this->getLeft(index); //gets index of left
element
    unsigned right_index = this->getRight(index); //gets index of right
element

    //if left node exists and left node smaller than smallest node value
    if (left_index < this->taxis.size() && smallest_element > this-
>taxis[left_index]) {
        smallest_index = left_index; //updates smallest node's index
        smallest_element = this->taxis[left_index]; //updates smallest
node's value
    }
    //if right node exists and right node smaller than smallest node value
    if (right_index < this->taxis.size() && smallest_element > this-
>taxis[right_index]) {
        smallest_index = right_index;//updates smallest node's index
        smallest_element = this->taxis[right_index]; //updates smallest
node's value
    }
    //Checks child nodes (left, right) smaller than current node or not
    if (smallest_index != index) {
        //Gets smaller node to the upper node
        swap(this->taxis[index], this->taxis[smallest_index]);
        //Calls MinHeapify for swapped node
        this->minHeapify(smallest_index);
    }
}
```

b. <u>getNearestTaxi Function</u>

Description:

This function provides operations that is combination of top() and pop() operations in standard library priority queue. It returns root of the heap and rearranges remaining tree using minheapify function, after assigning last element as a new root of heap.

Complexity:

- It takes first element, assigns last element to first element, decreases size of the array and these operations directly work in constant time.
- At the end of the function, it calls heapify function from first index that's complexity is calculated as $O(\log(n))$.
- Therefore, we can say that its complexity is directly equals to complexity of the heapify function.

$$\boldsymbol{T(n) = O(\log(n))}$$

```
//Function that get minimum element of heap
double PriorityQueue::getNearestTaxi() {
    //If heap is empty, throws error
    if (this->taxis.size() < 1) {
        cout<< "There is not any taxi!"<<endl;
        exit(1);
    }
    double min_element = this->taxis[0]; //Takes smallest value of the heap
    this->taxis[0] = this->taxis[taxis.size() - 1]; //Places last element
of heap as a root
    this->taxis.pop_back(); //Decreases size of heap
    this->minHeapify(0); //Calls MinHeapify for root
    return min_element; //Returns minimum element
}
```

c. <u>decreaseTaxiDistance Function</u>

Description:

This function allows us to decrease value of the random element from the heap. Firstly, it chooses random element from heap and decreases its value by 0.01. If it is smaller than 0.01, it directly gives 0 value to element. After decrement operation, we have to rearrange the heap. If our element's value is smaller than parent's value, we have to swap these elements to protect heap property. Until element reaches to smaller parent, it gets closer to root at every while iteration.

Complexity:

- Random function, decrease and reach operations directly work in constant time.
- At the end of the function, it iterates until it finds smaller parent node. For the worst case, we may choose element from deepest layer of the tree and after decrement operation, it can be smallest element of the heap. This case costs $O(\log(n))$ because depth of any binary tree is log(n).
- Therefore, we can say that its **worst-case** complexity is directly equals to depth of tree:

$$\boldsymbol{T(n) = O(\log(n))}$$

- For the **best case**, we can choose root as an element that will be updated. Therefore, complexity will be

$$\boldsymbol{T(n) = O(1)}$$

```
//Function for decrease node value of heap
void PriorityQueue::decreaseTaxiDistance() {
    int index = 0;
    if(this->taxis.size()<=0){
        return;
    }
    if(this->taxis.size()>1){
        //Randomly chooses index from heap
        index = rand() % (this->taxis.size());
    }
    //if node's value smaller than 0.01
    if(this->taxis[index]<0.01){
        //Directly sets as 0
        this->taxis[index] = 0;
    }else{
        //decreases value with 0.01
        this->taxis[index] -= 0.01;
    }
    //if it is not a root and it is smaller than its parent
    while (index > 0 && this->taxis[getParent(index)] > this->taxis[index])
{
        //Places to parent node's position
        swap(this->taxis[getParent(index)], this->taxis[index]);
        index = getParent(index); //Updates index
    }
}
```

d. **addTaxi Function**

Description:

This function allows us to add new element to priority queue. Its implementation logic is similar to decreaseTaxiDistance function. New element is pushed to end of the heap and it changes its position until it reaches smaller parent element.

Complexity:

- If we consider array size is enough for new element, adding new element handles in constant time.
- At the end of the function, it iterates until it finds smaller parent node. For the worst case, it can be smallest element of the heap. This case costs $O(\log(n))$ because depth of any tree is log(n). It should iterate all depth levels.
- Therefore, we can say that its complexity is directly equals to depth of tree:

$$\boldsymbol{T(n) = O(\log(n))}$$

```
//Function for adding new element to the heap
void PriorityQueue::addTaxi(double distance) {
    //Adds new value to the end of the heap
    this->taxis.push_back(distance);

    int index = taxis.size() - 1; //takes index value of the last element

    //if it is not a root and it is smaller than its parent
    while (index > 0 && this->taxis[getParent(index)] > this->taxis[index])
{
        //Places to parent node's position
        swap(this->taxis[getParent(index)], this->taxis[index]);
        index = getParent(index); //Updates index
    }
}
```

e. <u>Main Function and Simulation Features</u>

Description:

In main function, I handle all simulation features such as calculating passed time, calling necessary functions and prepare proper medium to simulation.

I have defined a queue that stores all taxi distances and called read file operation. To obtain better simulation results and directly observe the priority queue operations, I have handled all input operations before the simulation. Read file function reads first m+5 lines, converts position values to distance and pushes to taxi queue. If needed, I get top element of queue and I use it this way.

I have **initialized** my priority queue with **5 elements** to avoid errors that occurs because of empty priority queue while operating decreaseTaxiDistance function. With this method, we can prevent update operations that are tried to apply on empty priority queue.

I also noticed to avoid unnecessary time-consuming operations. Therefore, I defined called_taxis vector to **store distances of the called taxis** and **extract** effect of terminal **printing** time from simulation time. (cout effects total time significantly and decreases simulation reliability)

In simulation part, I have defined a while loop that provides required iteration number. In the loop for each **100th** operation, it calls getNearestTaxi function. To provide randomized function selection, I used rand function that generates number from [1-100] with modulo operation and I check the random value with **100*p**. If it is less than this value, I call decreaseTaxiDistance function and increment update_counter. If not, I call addTaxi function and increment add_counter. In simulation, I consider this part's execution time as a simulation execution time. Because fundamental heap operations are handled in this part.

Finally, I print necessary outputs and deallocate my priority queue.

Complexity:

- I have used one while loop that costs $\boldsymbol{O(n)}$
- For each iteration, I select one operation that are decreaseTaxiDistance, addTaxi. Their worst-case complexities are $\boldsymbol{O(log(n))}$
- Therefore, we can say upper bound of simulation is

$$\boldsymbol{T(n) = O(n * \log(n))}$$

```cpp
int main(int argc, char *argv[]) {

    srand(time(NULL)); //seeds random function
    int m; //keeps input size
    double p; //probability of update
    if(argc == 3){ //checks command line argument number
        m = atoi(argv[1]); //reads argument and transforms to integer
        p = atof(argv[2]); //reads argument and transforms to double
    }else{ //if argument is not in proper format
        cerr << "Command line argument does not given in right format!";
        exit(1); //exits with failure
    }

    queue<double> input_queue; //Stores inputs
    readFile(&input_queue, m+5); //reads input file

    PriorityQueue *taxi_pq = new PriorityQueue(); //Initialized priority
queue

    for(int i=0; i<5;i++){ //pq initialized with 5 taxis
        taxi_pq->addTaxi(input_queue.front()); //reads from input
        input_queue.pop(); //pops from input queue
    }

    int counter = 0; //counter keeps operation number

    int addition_counter = 0; //counter keeps addition number
    int update_counter = 0; //counter keeps update number

    vector<double> called_taxis; //vector that keeps distances of called
taxis
    time_t timer = clock(); //starts time
    while (m--) {
        if (counter == 100) { //if it is 100th operation
            double taxi = taxi_pq->getNearestTaxi();
            called_taxis.push_back(taxi); //gets taxi and pushes to array
            counter = 0; //sets counter as 0
        }
        if (rand() % 100+1 <= p * 100 ) {
            taxi_pq->decreaseTaxiDistance(); //calls decrease taxi distance
            update_counter++; //increments update counter by 1
            counter++; //increments counter by 1
        } else {
            taxi_pq->addTaxi(input_queue.front()); //reads from input
            input_queue.pop(); //pops from input queue
            addition_counter++; //increments addition counter by 1
            counter++; //increments counter by 1
        }
    }
    timer = clock() - timer; //calculates time that is passed
    //prints called taxi distances
    for(unsigned i=0; i<called_taxis.size();i++){
        cout<<"Taxi called. Distance: "<<called_taxis[i]<<endl;
    }
    //prints expected outputs
    cout <<"Number of addition operations: "<< addition_counter <<endl;
    cout <<"Number of update operations: " << update_counter << endl;
    cout <<"Time that is passed: "<<timer <<" ms"<<endl;
    delete taxi_pq; //deallocates priority queue
    return 0;
}
```

1) The Effect of m Value

In simulation, operations that are addTaxi, decreaseTaxiDistance are performed with rates that are determined in terms of probability value. And also, getNearestTaxi operation is called for each 100 operation. In this simulation our p value is constant and it is **0.2**. If we construct an equation that shows the complexity of the simulation:

Probability of addTaxi operation

Probability of decreaseTaxiDistance operation

Complexity of getNearestTaxi

$$T(n) = \sum_{i=0}^{n} (0.8) * \log(0.8 * i) + (0.2) * \log(0.8 * i) + \sum_{i=0}^{n/100} \log(100 * 0.8 * i)$$

N iteration

Complexity of addition operation

Worst-case complexity of update operation

For each 100 iteration

- In log functions, it is given $i * 0.8$ because we add new taxi to heap with 0.8 probability. Therefore, we have $0.8 * i$ taxis in i^{th} step.

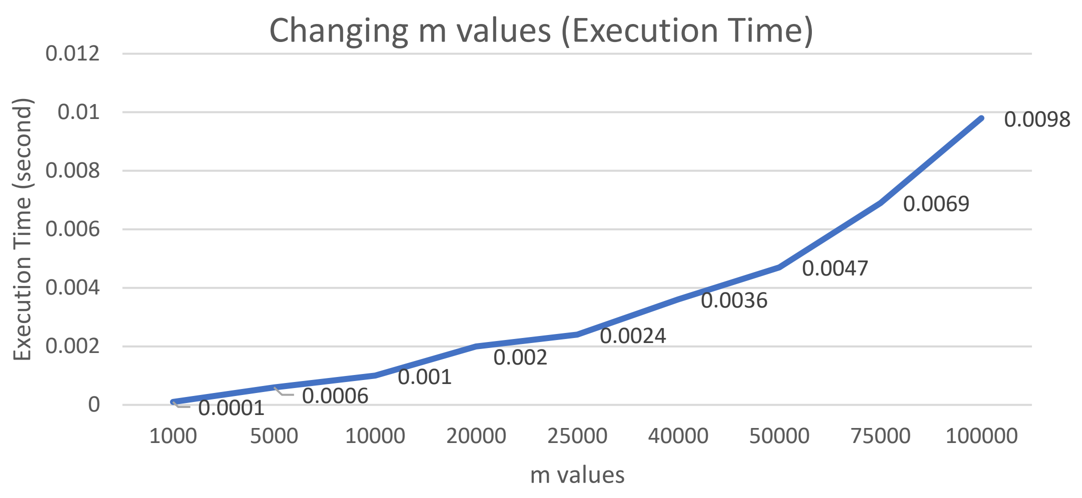
$$T(n) = \sum_{i=0}^{n} \log(0.8 * i) + \sum_{i=0}^{n/100} \log(80 * i) \leq \sum_{i=0}^{n} \log(i)$$

$$T(n) = \sum_{i=0}^{n} \log(i) \approx log(n!) < \boldsymbol{n * log(n)}$$

As we can observe in the calculations, theoretically **worst-case** of our simulation works with $\boldsymbol{n * log(n)}$ complexity.

My practical simulation table and plot are given below. For each value, I ran my code 10 times and I get their averages as execution time.

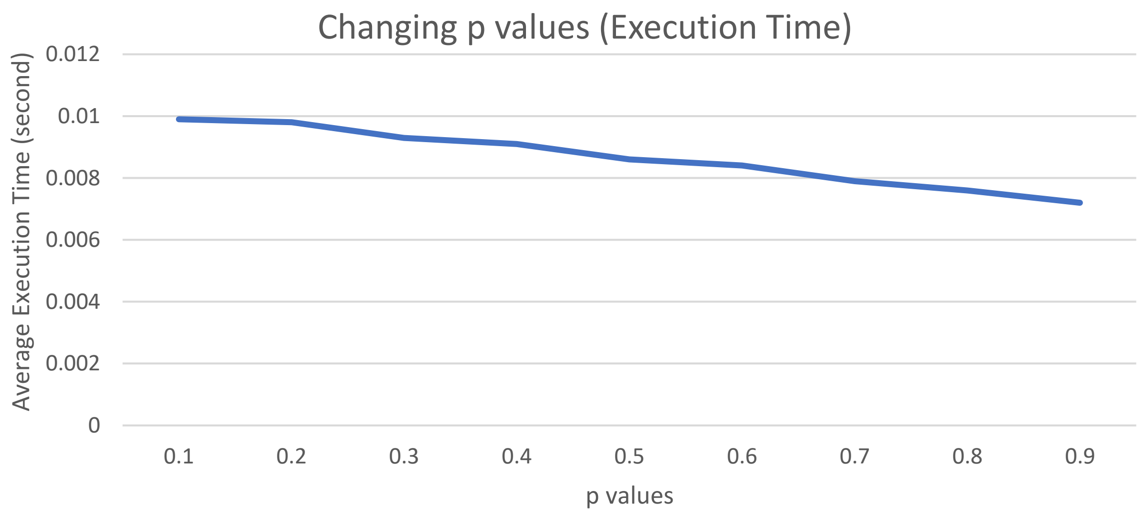
p	m	Average (N=10) Execution Time (seconds)
0.2	1K	0.0001
0.2	5K	0.0006
0.2	10K	0.001
0.2	20K	0.002
0.2	25K	0.0024
0.2	40K	0.0036
0.2	50K	0.0047
0.2	75K	0.0069
0.2	100K	0.0098



In my simulation, I have observed that my plot works **better** than worst-case approximation that is $O(\boldsymbol{n} * \boldsymbol{log}(\boldsymbol{n}))$. Actually, getting more faster result than worst-case is **expected** behavior because execution time can be affected from different conditions such as selected element for update operation, in which iteration the adding operations are performed etc. It does not have to be converge to worst-case.

If we examine bilateral relations between such as 1K-5K, 10K-20K or 50K-100K, we can observe that my implementation works in higher complexity than $O(n)$ and it has smaller complexity than $O(n * logn)$. But for some intervals such as 20K-40K, 25K-75K, it works with smaller complexities than $O(n)$. I have not expected these relations. I think that it can be caused because of precision errors of computer and difference between selected indexes for update operations.

2) The Effect of p Value



p	m	Average (N=10) Execution Time (seconds)
0.1	100K	0.0099
0.2	100K	0.0098
0.3	100K	0.0093
0.4	100K	0.0091
0.5	100K	0.0086
0.6	100K	0.0084
0.7	100K	0.0079
0.8	100K	0.0076
0.9	100K	0.0072

As seen in the plot that is given above, the execution time is affected by p value. As I have mentioned in the first part, worst cases for the insert and update operations have same complexity that is $O(\log(n))$. If we focus on only worst case-complexities, our expectation would be flat execution time graph because we are changing the execution rate of two functions that have same complexities. However, in the execution time plot, we have observed slightly decreasing graph. This situation can be explained depending on **heap size** that is directly proportional to addition operation and **execution time of update** operation.

For lower p values, we can say that addition operation is performed more times than higher p values. It causes to **faster grow of heap structure** that directly affects execution time of other add and update operations. For this reason, if we perform more addition operation, our execution time becomes **longer**. For this plot, the most significant factor is change in number of addition operations that affects execution time to observe decreasing graph.

Also, update operation usually runs faster than insert operation. As I stated in first part, worst case of update operation is updating element that locates in deepest level and because of this case, its complexity is $O(\log(n))$. But, if we choose elements from levels that are closer to root, then it runs faster than worst case complexity. On the other hand, insert operation always works with $O(\log(n))$, because it adds element to lower depth than it iterates through to root. Therefore, increase in number of update operations can provide faster running time than add operations.

In essence, if the number of updates increases, the execution time slightly (not significantly) decreases. It is same as what I have expected.