*Image taken from Brewminate*

# BLG 335E

# Homework 2 Report

## Priority Queue (Heap)

**Berke Dönmez**

**150170045**

**December 25, 2020**

# 1 Implementation and Running Time

Firstly, I implemented a **Min Heap** (a heap where every non-root node has parent with a smaller or equal value). My Min Heap uses the **STL `vector<double>`** as its main data structure because the storage should be dynamic since there are insertions (adding a new taxi) and deletions (calling a taxi).

The heap has two attributes: `vector<double> taxis` and `int _size` (number of nodes). Initially, I inserted 0 to `taxis` in order to handle with 1-based indexing.

There are 5 operations the heap can handle:

```
1 double getMin() const
2 void heapifyUP(int startIndex)
3 void insert(const double newTaxi)
4 void removeMin()
5 void update(int index)
```

In addition, I implemented some helper functions: `getLeftChild(i)`, `getRightChild(i)`, `getParent(i)`. Their names are self-explanatory. They return `2*i`, `2*i+1`, `i/2`, respectively.

## 1.1 `getMin`

It returns the minimum distance (value of root of the Min Heap):

```
1 double getMin() const{
2     return taxis[1];
3 }
```

Since it only accesses an element of a `vector` (i.e. array), it runs in $\mathcal{O}(1)$.

## 1.2 heapifyUP(startIndex)

It is called whenever something is changed in Min Heap. When a node's value gets smaller than its parent, it should be ascended in order to keep the heap healthy.

```
1 void heapifyUP(int startIndex){
2     int& index = startIndex;
3     int parent = getParent(index);
4     while(parent >= 1 && taxis[index] < taxis[parent]){
5         swap(taxis[index], taxis[parent]);
6         index = parent;
7         parent = getParent(index);
8     }
9 }
```

First two lines run in $\mathcal{O}(1)$, while loop is repeated at most $log_2(parent)$ (height of the parent) times (because `parent` is divided by 2 (floored) each time) and operations inside the loop run in $\mathcal{O}(1)$. In total, this function runs in $2 * \mathcal{O}(1) + log_2(parent) * 3 * \mathcal{O}(1) = \mathcal{O}(log_2(parent))$ time. The worst case takes place when initial value of $parent$ is maximized: $parent = \left\lfloor \frac{SIZE}{2} \right\rfloor$ where $SIZE$ is the number of nodes in the heap. Therefore, the worst case time complexity is $\mathcal{O}(log_2(SIZE))$.

## 1.3 insert(newTaxi)

It is called when a new taxi is added to heap.

```
1 void insert(const double newTaxi){
2     taxis.push_back(newTaxi);
3     _size++; //size of the heap
4     int curIndex = _size;
5     heapifyUP(curIndex);
6 }
```

First three lines run in $\mathcal{O}(1)$ (the time complexity of push back operation is linear when resizing happens but we may not consider this special case) and last line runs in $\mathcal{O}(log_2(SIZE))$. In total, insertion is done in $\mathcal{O}(log_2(SIZE))$.

## 1.4 `removeMin`

It is called when a taxi is called. It removes the root of the heap (nearest taxi) and re-heapifies the heap.

```
void removeMin(){
    if(_size == 0) return;
    swap(taxis[1], taxis[_size]);
    taxis.pop_back();
    _size--;
    if(_size == 0) return;
    int index = 1;
    while(true){
        int leftChild = getLeftChild(index);
        int rightChild = getRightChild(index);
        double leftChildVal = leftChild <= _size ?
        taxis[leftChild] : INF;
        double rightChildVal = rightChild <= _size ?
        taxis[rightChild] : INF;
        int minChild = -1;
        if(leftChildVal >= taxis[index]
           && rightChildVal >= taxis[index]){
            break;
        }
        if(leftChildVal < rightChildVal){
            minChild = leftChild;
        }else{
            minChild = rightChild;
        }
        swap(taxis[index], taxis[minChild]);
        index = minChild;
    }
}
```

Firstly, the root and the last node are swapped in order to pop the minimum value. Then, the heap should be heapified because the root is changed. While the value of the minimum valued child is smaller than the current node (initially root), current node should be swapped with this child. Loop is ended when there is not any child (node is a leaf) or the minimum valued child is bigger than or equal to the node.

The determinant of time complexity is the while loop because all operations run in $\mathcal{O}(1)$ (pop back has a special case similor to push back but generally, it runs in constant time). This loop is very similar to the `heapifyUP` function. The only difference is that it heapifies **down**. Therefore, removal is also done in $\mathcal{O}(log_2(SIZE))$ (height of the heap) with a little bigger constant factor.

## 1.5 `update(index)`

It is called whenever a random taxi's distance is decreased.

```
void update(int index){
    if(_size == 0) return;
    taxis[index] -= 0.01;
    if(taxis[index] < 0) taxis[index] = 0;
    heapifyUP(index);
}
```

Firstly, value of the randomly selected node is decreased by $0.01$ and when it gets below zero, it gets back to zero. Since there is a possibility that the node's value becomes smaller than its ancestors' values, heap is heapified starting from the current node. Update is done in $\mathcal{O}(log_2(SIZE))$ because of the heapify.

## 1.6 Simulation Phase

In every operation, I calculate the probability of the operation being an update or an addition by the following way: I get a random real number from the range $[0, 1]$. If it is smaller than $p$, the update probability, distance of a random taxi is updated. Else, a new taxi is called.

```
double which = (double) myRand() / RAND_MAX;
if(which <= p){
    //update
}else{
    //call
}
```

Update is done as follows: Pick a random integer between 1 and heap size, then call the update function with this integer (target index). When there is no available taxi, update cannot be done. Therefore, it is not counted.

```
int size = heap.size();
if(size == 0) continue;
int randomTaxi = myRand() % size + 1;
heap.update(randomTaxi);
update++; //total number of updates
```

Addition is done by reading the location of new taxi from the file, calculating its distance from the hotel and inserting this distance to the heap.

```
double longitude, latitude;
file >> longitude >> latitude;
double dist = distance(longitude, latitude);
heap.insert(dist);
addition++; //total number of additions
```

Lastly, if a new taxi is called (removal), root's value in heap is added to a vector `calledTaxis` and `removeMin()` function is called. Below, `i` keeps the operation order. Similar to update, if there is no taxi available, no taxi is called.

```
if(i % 100 == 0){
    if(heap.size() == 0) continue;
    calledTaxis.push_back(heap.getMin());
    heap.removeMin();
}
```

All operations are done in $\mathcal{O}(log_2(SIZE))$ since in every operation, a logarithmic-timed function of heap is called.

In total, program runs theoretically in $\mathcal{O}(m*log_2(m))$ where $m$ is the number of operations. It is actually not the tightest upper bound because it is assumed that before every operation, the size of the heap is $m$ and every function works in the worst case. In detail, we can say that runtime is $\mathcal{O}(log_2(s_1)) + \mathcal{O}(log_2(s_2)) + \mathcal{O}(log_2(s_3)) + ... + \mathcal{O}(log_2(s_m))$ where $s_i$ is the size of heap before the $i$-th operation. Also, the runtime depends on $p$ because addition increases the size while update does not affect the size. When $p$ gets bigger, runtime gets smaller as the number of updates is expected to get bigger and the number of addition is expected to get smaller. Similarly when $p$ gets smaller, runtime gets bigger. **It is important to note that I calculated the total running time (e.g. file reading included).**

# 2 Runtimes for different m (p = 0.2)

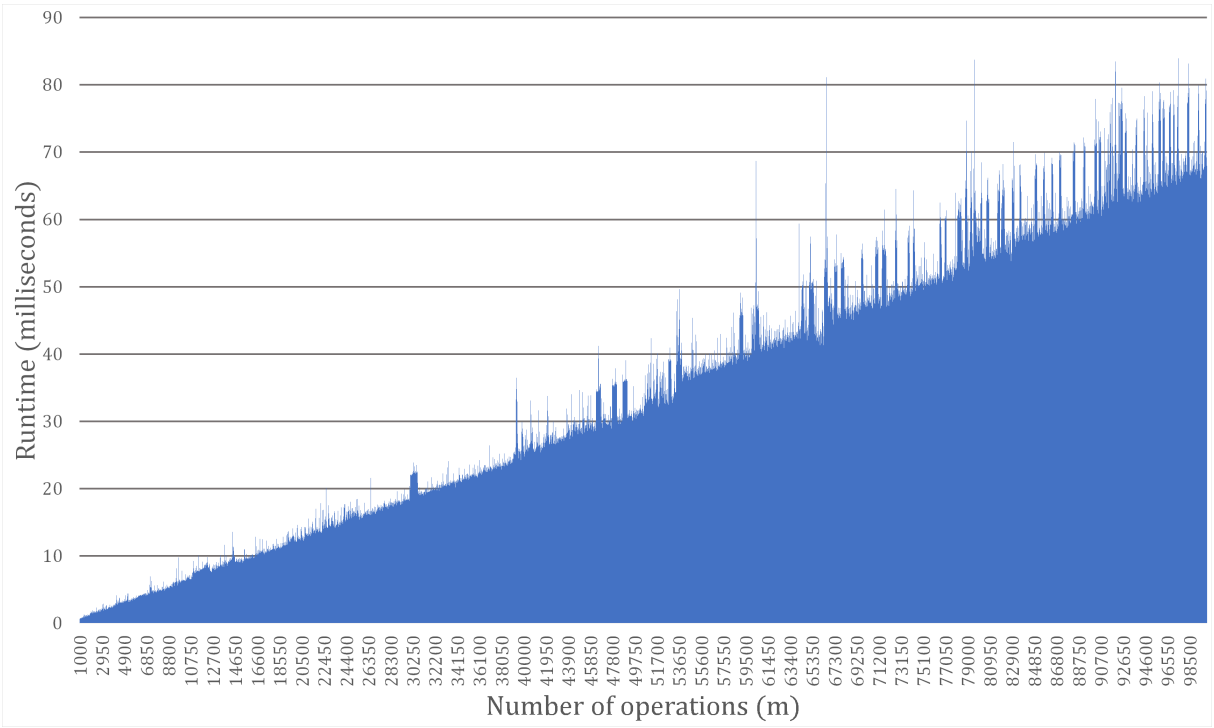


Figure 1: Runtimes for $m \in [10^3, 10^5]$

I have run the program 5 times (taking the average runtime) for each multiple of 25 between $[10^3, 10^5]$ i.e. 1025, 1050, 1075... and got the graph above, using Excel. I believe that since the runtime depends on many factors (the computer being imperfect, $p$, order of taxis in the file, file reading etc.), the graph is not that perfect. The theoretical runtime was $\mathcal{O}(m * log_2(m))$ but the graph is close to being linear. Here, I want to say that it is a "nice" $\mathcal{O}(m * log_2(m))$.

# 3 Runtimes for different p ($m = 10^5$)

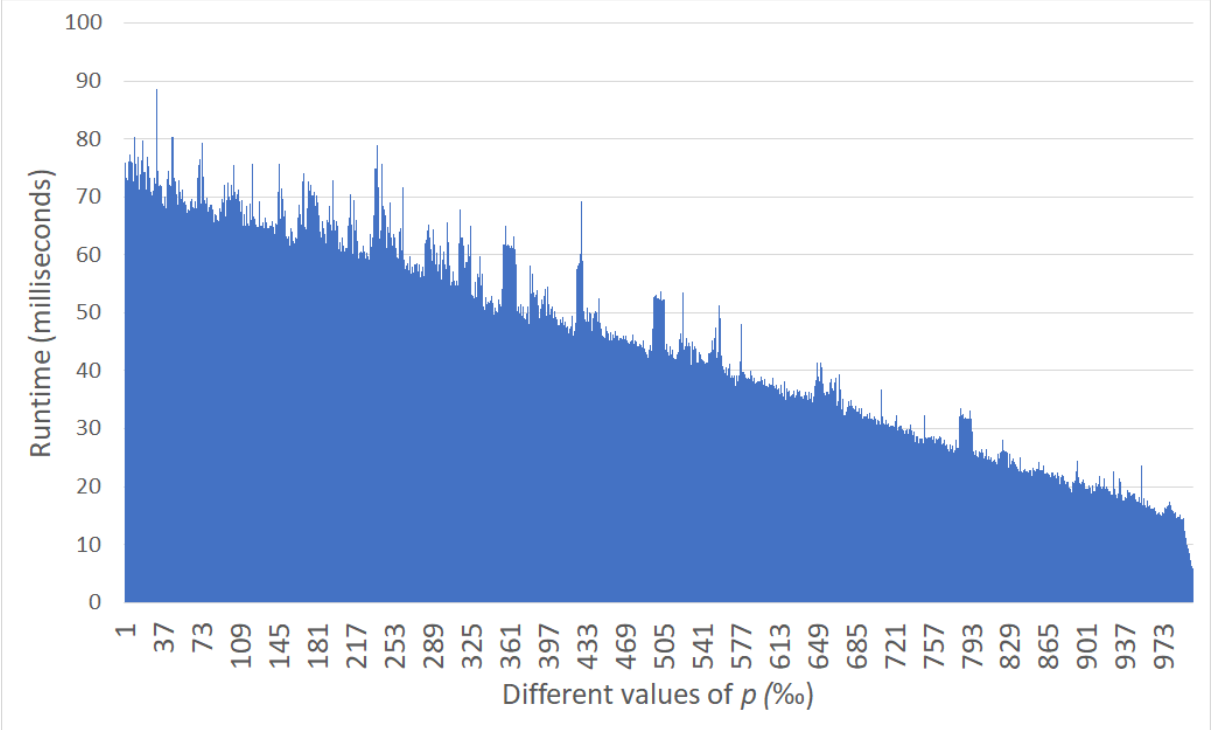


Figure 2: Runtimes for $p \in [0, 1]$

$p$ is a determinant on the runtime but not that effective when $m$ is large. Every operation runs in $\mathcal{O}(log_2(SIZE))$. As p increases, total number of updates increases and total number of additions decreases. Addition increments the total number of nodes i.e. $SIZE$ and update does not affect $SIZE$. Let's call the average $SIZE$s among all the operations as $SIZE_{avg}$. When $p$ is close to 0, say 0.1, assume that $a$ additions are done. When $p$ is close to 1, say 0.9, number of additions is expected to be $\frac{a}{9}$ because number of additions is proportional to $1 - p$. Thus, ratio of $SIZE_{avg}$s is expected to be $\approx 9$. Program runs ins $\mathcal{O}(m * log_2(SIZE_{avg}))$. Thus, ratio of runtimes is ratio of logarithms of average sizes. Average sizes are expected to be proportional to $m * (1 - p)$ (expected final size of heap). There are also removals but not that much ($\frac{m}{100}$). Finally, ratio of runtimes would be

$$\frac{m * log_2(m * 0.9)}{m * log_2(m * 0.1)} = log_{10^4}(9 * 10^4) \approx 1.2386$$

But on graph, the ratio is larger. This is because I included the file readings. This affects the total runtime considerably much.