BLG335E HOMEWORK2 REPORT Burak Atlamaz

150180067

1) Implememtations on PQ operations and their theoretical running times

void insert(double): Goal is to put the i'th element in the correct place in a portion of the vector that "almost" has the heap property.

```
void taxi::insert(double key) { //Insert in slide does not worked, so I write different version
    int x = size++;                                                    // O(2)
    heap.push_back(key);    //Add to the last, then swap between parent-child      // O(1)

    /*If I call decrease_key, the func will give "larger" error each time
    due to heap[size] = +infinite line, so I write similar func again*/

    while (x > 0 && heap[parent(x)] > heap[x]) {                       // O(lg n)
        //Exchange
        double a = heap[x];                                            // O(1)
        heap[x] = heap[parent(x)];                                     // O(1)
        heap[parent(x)] = a;                                           // O(1) -> Exchange : O(3)

        //From child to parent until parent is smaller
        x = parent(x);                                                 // O(1)
    }
}//Cost is O(lg n)
```

Any loops? Yes. How many times will the loop execute? As many times as node x has ancestors, which = the depth of the tree. The depth of a binary tree is O(log *n*). We do a constant amount of work in the loop.

Cost is: O(7) + O(k. log *n*). k is constant number so that is equal to O(log *n*)

double extract_min(): remove and return element of heap with the smallest key

```
double taxi::extract_min() {
    //Error Message
    if (size < 1) {                         // O(1)
        cout << "heap_underflow" << endl;   // O(1)
        return -1;                          // O(1), not in the slide but have to return something
    }

    double max = heap[0];                   // O(1)
    heap[0] = heap[--size];                 // O(2)
    min_heapify(0);                         // O(lg n)
    return max;                             // O(1)
}//Cost is O(lg n)
```

Any loops? No. So just sum up the times: O(7) + O(log *n*). The dominant term is O(log *n*)

void min_heapify(int): Goal is to put the i'th element in the correct place in a portion of the vector that "almost" has the heap property.

```
void taxi::min_heapify(int x) { //In extract_min func, deletes smallest element
    int left = left_child(x);                                   //O(1)
    int right = right_child(x);                                 //O(1)
    int smallest = x;                                           //O(1)
    if (left < size && heap[left] < heap[x]) {                  //O(2)
        smallest = left;                                        //O(1)
    }
    else {                                                      //O(1)
        smallest = x;                                           //O(1)
    }
    if (right <= size && heap[right] < heap[smallest]) {        //O(2)
        smallest = right;                                       //O(1)
    }
    if(smallest != x) {                                         //O(1)
        //Exchange
        double a = heap[x];                                     //O(1)
        heap[x] = heap[smallest];                               //O(1)
        heap[smallest] = a;                                     //O(1) -> Exchange : O(3)

        //Recursive
        min_heapify(smallest);                                  //O(lg n)
    }
}//Cost is O(lg n)
```

Before the recursive, there is no loop and number of the steps does not depends on *n*. So recursive will be determine the time complexity.

When we call min_heapify(int), we know the process is before we call it again. So main problem in here is how many times we call recursive.

If the binary tree is full, above half of the three will be involved. For example, if the three has 7 nodes, (7-1)/2 of the nodes(2, 4, 5) will be involved which is just a little smaller than half of the three.

Else if the last row of the three is half full, more than ablove half of the three will be involved. For example, if three has 5 nodes, 3 of it(2, 4, 5) will be involved. As can be seen, number of involved nodes are the same but this time they has larger percent. In general, 2/3 of the three might be involved and this is worst case.

So, $T(n) \leq T(2n/3) + O(k)$. k is constant, does not depend on *n*

By using Master Theorem, $T(n) = O(\log n)$

void decrease_key(): remove and return element of heap with the smallest key

```
void taxi::decrease_key(int x, double key) {
    if (key > heap[x]) {                                    // O(1)
        cout << "new_key_is_larger_than_current_key" << endl;   // O(1)
        return;                                             // O(1)
    }
    heap[x] = key;                                          // O(1)
    while (x > 0 && heap[parent(x)] > heap[x]) {            // O(lgn)
        //Exchange
        double a = heap[x];                                 // O(1)
        heap[x] = heap[parent(x)];                          // O(1)
        heap[parent(x)] = a;                                // O(1) -> Exchange : O(3)

        //From child to parent until parent is smaller
        x = parent(x);                                      // O(1)
    }
}//Cost is O(lg n)
```

Any loops? Yes. How many times will the loop execute? As many times as node x has ancestors, which = the depth of the tree. The depth of a binary tree is O(log *n*). We do a constant amount of work in the loop.

Cost is: O(8) + O(k. log *n*). k is constant number so that is equal to O(log *n*)

NOTE: This code is compiled as g++ -Wall –Werror main.cpp

2) p is constant 0.2

m = 1000, Running time = 1.815 millisecond

m = 2000, Running time = 2.934 millisecond

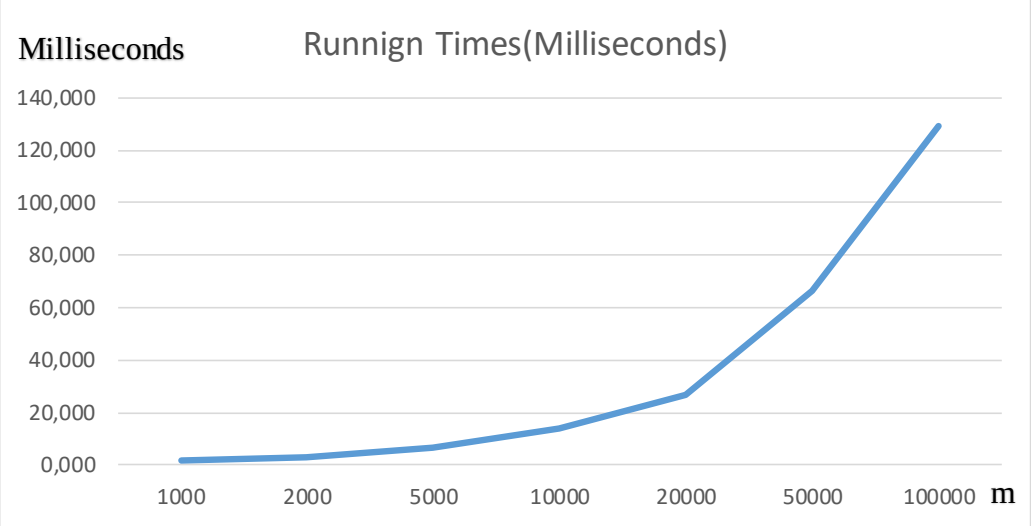
m = 5000, Running time = 6.855 millisecond

m = 10000, Running time = 13.994 millisecond

m = 20000, Running time = 26.737 millisecond

m = 50000, Running time = 66,136 millisecond

m = 100000, Running time = 129.273 millisecond



While adding, updating or extracting node; the cost is O(log *n*). In the whole program, if we implement n nodes in the PQ, we will do one of the implementation operations n time. So total cost of implementations on PQ is n. O(log *n*) = O(*n.* log *n*).

Graph shape is similar to O(*n.* lg *n*). Between 20000-10000 looks more curved but it is because of the difference increases but place keeps same size.

Ratio of measured running time between 2000-5000 = 2,335

Ratio of theoretical running time = 5000.log(5000) / 2000.log(2000)

= 2,5.log(5).log(1000) / log(2).log(1000)

= 2,5.log(2,5)

Ratio of measured time between 20000-50000 = 2,

Ratio of theoretical running time = 500000.log(50000) / 20000.log(20000)

= 2,5.log(5).log(10000) / log(2).log(10000)

= 2,5.log(2,5)

The measuered ratio is same and we expect that the calculated running times also must be same. As can be seen, the calculated times are similar with little margin error so we can say the theoretical running times matches with measured running times.

3) m is constant 10000

p = 0.1, Running time = 15,684 millisecond

p = 0.2, Running time = 13.994 millisecond

p = 0.3, Running time = 11.949 millisecond

p = 0.4, Running time = 10.234 millisecond

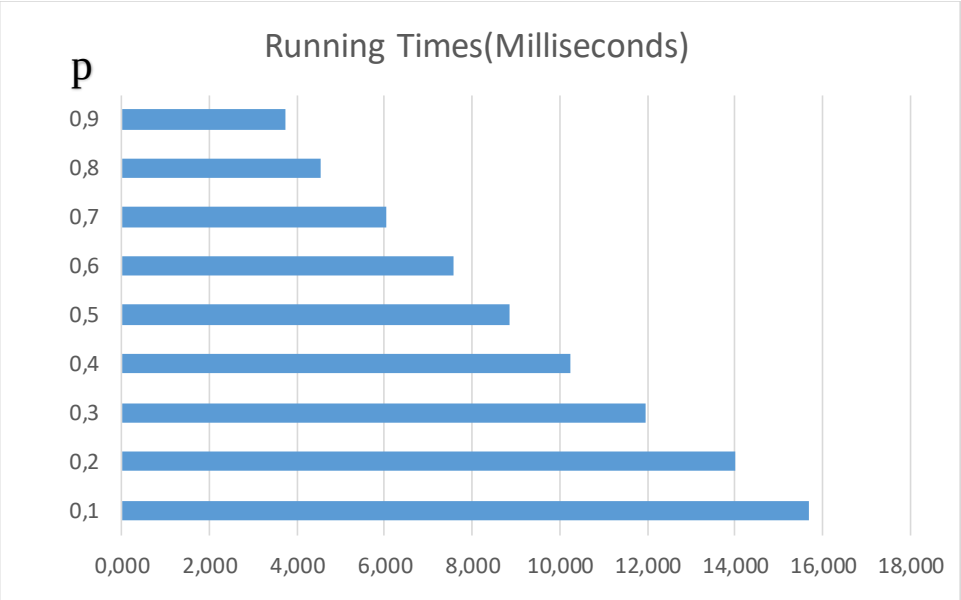
p = 0.5, Running time = 8.838 millisecond

p = 0.6, Running time = 7.558 millisecond

p = 0.7, Running time = 6.025 millisecond

p = 0.8, Running time = 4.534 millisecond

p = 0.9, Running time = 3.726 millisecond



While m is constant, while p is increased running time is decreased. In the homework, p is probability of update and 1 – p is probability of addition; so while number of update increases and number of addition decreases running time is decreased. But why?

In every addition, total node number is also increases. For example if p is 0.1, approximately after all the file reading done we have 9000 nodes; in the other hand, if p is 0.9, we will have approximately 1000 nodes (m is 10000). This situation increases depth of the tree and number of parents and owing to that the functions that have exchange check more nodes. *n* and O(lg *n*) increases. Total running time increases.

And also, reading numbers from locations.txt file and calculating distance with euclidian function have effect on total running time.

NOTE: While writing code, calling taxis does not counted as operation, so total operation number is equal to addition + update(decrease random distance 0.01).