BLG335E Algorithm Analysis Assignment2 Report

Mihriban Nur Koçak - 150180090

17.12.2021

Question 1

heapExtractMin() function does corresponding heap operation for the extract operation in the implementation:

```
Vehicle* heapExtractMin(Vehicle** arr, int* vehicle_heap_size){
    if(*vehicle_heap_size < 1){
        //Heap size should not be lower than 1
        cout << "Heap underflow" << endl;
    }
    Vehicle* min = arr[0]; //as a heap property element with sma
    arr[0] = arr[(*vehicle_heap_size)-1];//move last element of
    *vehicle_heap_size = *vehicle_heap_size - 1;//to remove smal
    minHeapify(arr,0,vehicle_heap_size);//call minHeapify for el
    return min;//return smallest element
}
```

Figure 1: Extract Function in the Implementation

- Check whether heap has a proper size or not = $\mathbf{O(1)}$
  - If heap doesn't have a proper size, emphasize this by via output = $\mathbf{O(1)}$
- Since first element of the array is the element with smallest key according to heap property, assign it to min variable to get element with smallest key = $\mathbf{O(1)}$
- Change first element of the array as the last element of the array = $\mathbf{O(1)}$
- Decrease size of the heap by 1 = $\mathbf{O(1)}$
- Call **minHeapify()** for first element of the array to maintain the heap property = $\mathbf{O(log(n))}$ (will be explained in detail)

- Return extracted minimum element = **O(1)**

Since there are not any loops or recursions, the times can be just summed up to obtain the total time:

Total time = O(1)+O(1)+O(1)+O(1)+O(1)+O(1)+O(log(n))= O(6) + O(log(n))

O(6) + O(log(n)) = **O(log(n))** since O(6) this very small compare to O(log(n)), O(log(n)) dominates the time

Furthermore, **minHeapify()** should be explained since it is used in heapExtractMin() function. **minHeapify()** is crucial part of heap since it is responsible for maintaining heap property:

```cpp
void minHeapify(Vehicle** arr, int index, int* vehicle_heap_size){
    int leftchild = 2 * (index+1); //using formula to get index of left child
    leftchild = leftchild - 1;
    //since array index starts from 0, 1 is added to index for proper calculation then substracted
    int rightchild = (2 * (index+1)) + 1; //using formula to get index of right child
    rightchild = rightchild - 1;
    //since array index starts from 0, 1 is added to index for proper calculation then substracted
    int smallest = index; //assume index of the heap's smallest element is current index

    //if both index of left child is smaller than heap size and left child's key is smaller than i
    if(leftchild < *vehicle_heap_size && arr[leftchild]->get_key() < arr[index]->get_key()){
        smallest = leftchild;
    }


    //if both index of rightchild is smaller than heap size and rightchild's key is smaller than b
    if(rightchild < *vehicle_heap_size && arr[rightchild]->get_key() < arr[smallest]->get_key()){
        smallest = rightchild;
    }

    //if smallest of parent,leftchild,rightchild is not parent, exchange parent with the smallest
    if(smallest != index){
        Vehicle* buffer = arr[index];
        arr[index] = arr[smallest];
        arr[smallest] = buffer;
        //call minHeapify for smallest's index to maintain the heap property
        minHeapify(arr,smallest,vehicle_heap_size);
    }
}
```

Figure 2: Minimum Heapify Function in the Implementation

- Obtain the index of left child = **O(2)**
- Obtain the index of right child = **O(2)**
- Assume index of the smallest is parent's index = **O(1)**
- Check if whether both index of left child is smaller than heap size and key value of left child is smaller than parent's key value or not = **O(1)**

  - If both conditions holds, assign index of the smallest element as left child's index = **O(1)**

- Check if whether both index of right child is smaller than heap size and key value of right child is smaller than both parent's key and left child's key values or not = **O(1)**
  - If both conditions holds, assign index of the smallest element as right child's index = **O(1)**

- Check if index of smallest element is equal to parent's index or not = **O(1)**
  - If it is not equal, exchange parent with the one of its children whose index corresponds to index of smallest element = **O(1)**
  - Call **minHeapify()** recursively for parent node with its exchanged new index(will be explained in detail)

Since there are not any loops or recursions until recursive call of function **minHeapify()**, the times can be just summed up to obtain the total time up to that point:

Total time up to recursive call = O(2)+O(2)+O(1)+O(1)+O(1)+O(1)+O(1)+O(1)+O(1)= **O(11)**

To decide the running time of recursion, we need to decide how many nodes might be involved to recursion in the worst case scenario. Since it is stated that each children has a subtree with at most 2/3 of current tree's size at each recursive call, the following equation can be obtained:

T(n) $\leq$ T(2n/3) + O(11)

We can solve this recurrence using Master Theorem:

- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Then

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

Figure 3: Master Theorem from Lecture Slides

$$d = 0,\ a = 1,\ b = 3/2$$
$$1 = (3/2)^1$$
$$a = b^d$$
$$\mathbf{T(n) = O(log(n))}$$

heapDecreaseKey() function does corresponding heap operation for the decrease operation in the implementation:

```cpp
void heapDecreaseKey(Vehicle** arr, double key, int index){
    if(key > arr[index]->get_key()){
        //to decrease key the given key value should be smaller than cu
        cout << "New key is larger than current key";
    }
    arr[index]->set_key(key);//set key value of element at (index)th in
    int parent = ((index+1)/2)-1;//use formula to reach parent of the (
    while(index > 0 && arr[parent]->get_key() > arr[index]->get_key()){
        //if parent's key value is larger than (index)th node's key val
        //continue this operation until index becomes zero or parent's
        Vehicle* buffer = arr[parent];
        arr[parent] = arr[index];
        arr[index] = buffer;
        index = parent;
        parent = ((index+1)/2)-1;
    }
}
```

Figure 4: Decrease Function in the Implementation

- Check whether the given key is larger than the current key or not = $\mathbf{O(1)}$
  - If yes, emphasize this by via output = $\mathbf{O(1)}$
- Assign given key to element with given index = $\mathbf{O(1)}$
- Obtain index of the parent of element with given index = $\mathbf{O(1)}$
- Execute loop if both index is not index of the root and parent's key is larger than child's (element with given index) key = $\mathbf{O(log(n))}$ (will be explained in detail)
- Exchange child(element with given index) and its parent = $\mathbf{O(3)}$
- New given index is assigned as parent's index = $\mathbf{O(1)}$
- Obtain new index of the parent of element with given index = $\mathbf{O(1)}$

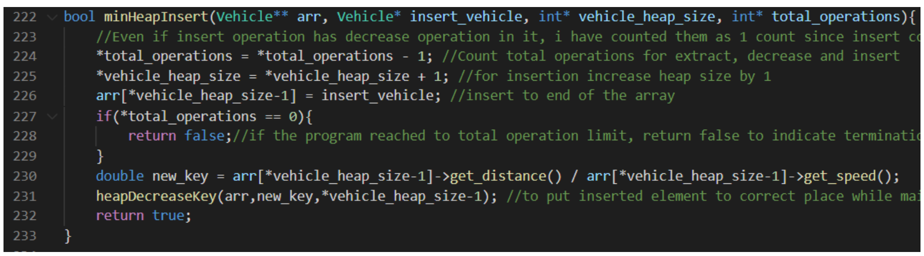
Since the while loop is executed as many times we goes up from a child to its parent level by level, it will be executed at most depth of the three times which is equal to O(log(n)) for a binary tree. At each execution of while loop the amount of time is O(5), so that the total time for while loop is O(5log(n)).

Since there are not any loops or recursions other than while loop, the times can be just summed up to obtain the overall total time:

Overall total time = O(1)+O(1)+O(1)+O(1)+O(5log(n)) = O(4) + O(5log(n))

O(4) + O(5log(n)) = **O(log(n))** since O(log(n)) dominates the time

minHeapInsert() function does corresponding heap operation for the insert operation in the implementation:



```
bool minHeapInsert(Vehicle** arr, Vehicle* insert_vehicle, int* vehicle_heap_size, int* total_operations){
    //Even if insert operation has decrease operation in it, i have counted them as 1 count since insert co
    *total_operations = *total_operations - 1; //Count total operations for extract, decrease and insert
    *vehicle_heap_size = *vehicle_heap_size + 1; //for insertion increase heap size by 1
    arr[*vehicle_heap_size-1] = insert_vehicle; //insert to end of the array
    if(*total_operations == 0){
        return false;//if the program reached to total operation limit, return false to indicate terminatio
    }
    double new_key = arr[*vehicle_heap_size-1]->get_distance() / arr[*vehicle_heap_size-1]->get_speed();
    heapDecreaseKey(arr,new_key,*vehicle_heap_size-1); //to put inserted element to correct place while mai
    return true;
}
```

Figure 5: Insert Function in the Implementation

- Decrease total operations by 1 since insert is counted as operation = **O(1)**
- For insertion, increase heap size by 1 to arrange space for element to be inserted = **O(1)**
- Insert desired element to the end of the array = **O(1)**
- Check whether the program reached total operation limit or not = **O(1)**
  - If yes, terminate by returning false = **O(1)**
- Calculate key value for inserted element = **O(1)**
- Decrease key value of inserted element as calculated key to put it its correct place by calling heapDecreaseKey() function = **O(log(n))** (it is explained previously)
- Return true to indicate the program didn't reach total operation limit = **O(1)**

Since there are not any loops or recursions, the times can be just summed up to obtain the total time:

Total time = O(1)+O(1)+O(1)+O(1)+O(1)+O(1)+O(log(n))+O(1) = O(7) + O(log(n))

O(7) + O(log(n)) = **O(log(n))** since O(log(n)) dominates the time

Question 2

Here is the calculation of execution times for different N values at SSH environment:

N	Execution 1(ms)	Execution 2(ms)	Execution 3(ms)	Execution 4(ms)	Execution 5(ms)	Average Execution Time (ms)
1000	0	0	0	0	0	0
10000	0	0	10	0	10	4
20000	10	10	10	0	10	8
50000	20	20	20	20	20	20
100000	40	40	50	40	30	40

Figure 6: Table of the Calculated Execution Times For Different N Values

Here is the plot of the above table which shows the relation between N value and execution time:

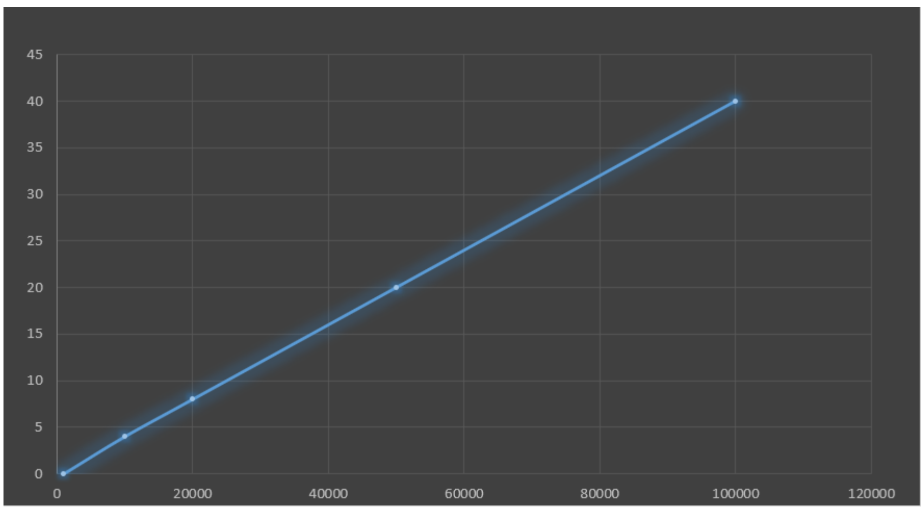


Figure 7: Plot of the Calculated Execution Times For Different N Values

As we obtained at Question 1, each of the extract, decrease and insert operations takes O(log(n)) time. So that, since total number of operations is n, overall total time should be O(nlog(n)).

Therefore, as can be seen from the plot, the results which I have obtained is suitable for O(nlog(n)). It can be clarified by following calculations:

$20000xlog(20000)/10000xlog(10000) >= 8/4$
$50000xlog(50000)/20000xlog(20000) >= 20/8$
$100000xlog(100000)/50000xlog(50000) >= 40/20$

In conclusion, it can be claimed that the theoretical result we obtained at Question 1 match with the practical result we obtained at Question 2.