ANALYSIS OF ALGORITHMS 1
BLG335E
PROJECT 2

ANALYSIS OF HEAP SORT

PREPARED BY:

150150058

HASAN H. EROGLU

Table of Contents

Explanation of Heap Implementation

Heap Class

Heap class contains an array, an integer, 2 constructors, 1 destructor and 20 functions. Heap class, that I created, uses template class T as an array. For this project this T was Employee class (which is another class I created). Later, if someone wants to use this code to sort (using heap) custom classes, he/she can use it easily.

MaxHeapify/MinHeapify (O(lgn))

MaxHeapify/MinHeapify functions check children of the current element and swap with the largest child for MaxHeapify (if children are larger than the current element) and swap with the smallest child for MinHeapify (if children are smaller than the current element). When recursion of MaxHeapify/MinHeapify excluded, it will cost O(1), because there are no loops or anything else. When recursion included, it will cost O(lgn). For example, if it starts from the root, the worst possible case ending up on the furthest leaf, which has height of lgn. Therefore, its cost is O(lgn).

```
void MaxHeapify(int _numbers[], int _index){

    if(_index > heapSize || _index < 0)
        return;

    int current = _numbers[_index];
    int leftChild = -1;
    int rightChild = -1;
    bool leftChildExists = GetLeftChild(_numbers, _index, leftChild);
    bool rightChildExists = GetRightChild(_numbers, _index, rightChild);
    int nextHeapifyIndex = -1;

    if(current >= leftChild && current >= rightChild)
        return;

    if(leftChildExists && (leftChild >= rightChild))
        nextHeapifyIndex = SwapWithLeftChild(_numbers, _index);

    else if(rightChildExists && (rightChild > leftChild))
        nextHeapifyIndex = SwapWithRightChild(_numbers, _index);

    MaxHeapify(_numbers, nextHeapifyIndex);
}
```

Figure 1: MaxHeapify Function

```
void MinHeapify(int _numbers[], int _index){

    if(_index >= heapSize || _index < 0)
        return;

    int current = _numbers[_index];
    int leftChild = -1;
    int rightChild = -1;
    bool leftChildExists = GetLeftChild(_numbers, _index, leftChild);
    bool rightChildExists = GetRightChild(_numbers, _index, rightChild);
    int nextHeapifyIndex = -1;

    if(current >= leftChild && current >= rightChild)
        return;

    if( leftChildExists && (leftChild <= rightChild))
        nextHeapifyIndex = SwapWithLeftChild(_numbers, _index);

    else if( rightChildExists && (rightChild < leftChild))
        nextHeapifyIndex = SwapWithRightChild(_numbers, _index);

    MinHeapify(_numbers, nextHeapifyIndex);
}
```

Figure 2: MinHeapify Function

BuildMaxHeap/BuildMinHeap (O(n))

BuildMaxHeap/BuildMinHeap uses MaxHeapify/MinHeapify to build heap. Since half of the heap will be leaves (of the binary tree), it starts from the middle of the heap and calls MaxHeapify/MinHeapify for all elements until the root (root included). It is known that MaxHeapify/MinHeapify costs O(lgn). BuildMaxHeap/BuildMinHeap loops from the middle of the heap(n/2) to the root. That means n/2 times MaxHeapify/MinHeapify will be called. Therefore, cost for BuildMaxHeap/BuildMinHeap is O(nlgn) (This is not tight bound for them.O(n) is the tight bound.Check course slides to see simplifications).

```
void BuildMaxHeap(int _numbers[]){
    int startIndex = (heapSize - 1)/2;

    for(int i = startIndex; i >= 0; i--)
        MaxHeapify(_numbers, i);
}
```

Figure 3: BuildMaxHeap Function

```
void BuildMinHeap(int _numbers[]){
    int startIndex = (heapSize - 1)/2;

    for(int i = startIndex; i >= 0; i--)
        MinHeapify(_numbers, i);
}
```

Figure 4: BuildMinHeap Function

HeapSort (O(nlgn))

HeapSort first calls BuildMaxHeap (costs O(n)), then loops from the end to the beginning of the heap and on every loop, it swaps the first and the last element of the heap and call MaxHeapify for the root. It calls MaxHeapify (costs O(lgn)) n times. As a result, it costs O(nlgn) + O(n), which is equal to O(nlgn).

```
void HeapSort(int _numbers[]){
    int startIndex = heapSize - 1;
    int tempHeapSize = heapSize;

    BuildMaxHeap(_numbers);

    for(int i = startIndex; i >= 1; i--){
        Swap(_numbers[0], _numbers[heapSize - 1]);
        Swap(heap[0], heap[heapSize - 1]);
        heapSize--;
        MaxHeapify(_numbers, 0);
    }

    heapSize = tempHeapSize;
}
```

Figure 5: HeapSort Function

IncreaseKey (O(lgn))

Increase key function increases desired element's value in the heap and then puts that element to right position in the heap (to preserve its heap condition). It checks the desired element's parent if it is smaller than its parent function stops running, else it swaps the desired element with its parent until the desired element and its parent hold heap condition. The worst case for this function is increasing the furthest (according to the root) leaf's value. Since binary tree's height is lgn, this function will cost O(lgn).

```
void IncreaseKey(int _numbers[], int _index, int newNumber, T _newElement){

    if(_index <= 0 || _index > heapSize)
        return;

    int parentIndex = GetParentIndex(_index);

    if(parentIndex <= 0 || parentIndex > heapSize)
        return;

    heap[_index] += _newElement;

    numbers[_index]  += newNumber;

    while(_index > 0 && _numbers[_index] > _numbers[parentIndex]){

        Swap(heap[_index], heap[parentIndex]);
        Swap(_numbers[_index], _numbers[parentIndex]);

        _index = parentIndex;
        parentIndex = GetParentIndex(parentIndex);
    }
}
```

Figure 6: IncreaseKey Function

Insert (O(lgn) for my function O(n))

Insert increases heap size and calls the increase key to put the new element to the heap. Since it puts the new element to the end of the heap, it will be the worst case for increase key function and cost will be O(lgn).

Note: My function increases heap's memory. To increase memory, it loops n times to fill the new array. Therefore, for my insert function cost becomes O(n).

```
int Insert(int *& _numbers, int _newNumber, T _newElement, int _heapSize){

    int oldHeapSize = _heapSize;
    _heapSize++;
    heapSize = oldHeapSize + 1;

    IncreaseArrayMemory(_numbers, oldHeapSize, heapSize);
    IncreaseArrayMemory(heap, oldHeapSize, heapSize);

    _numbers[heapSize - 1] = 0;
    heap[heapSize - 1] = T();
    IncreaseKey(_numbers, heapSize - 1, _newNumber, _newElement);

    return _heapSize;
}
```

Figure 7: Insert Function

ExtractMax/ExtractMin (O(lgn))

ExtractMax/ExtractMin stores the root of the heap (largest/smallest element), then swaps the first element with the last element and calls MaxHeapify/MinHeapify on the root. Since MaxHeapify/MinHeapify costs O(lgn) and there is no loop or anything else, it will cost O(lgn).

```
int ExtractMax(int numbers[]){
    int max = numbers[0];

    numbers[0] = numbers[heapSize - 1];
    heapSize--;

    MaxHeapify(numbers, 0);

    return max;
}
```

Figure 8: ExtractMax Function

```
int ExtractMin(int numbers[]){
    int min = numbers[0];

    numbers[0] = numbers[heapSize - 1];
    heapSize--;

    MinHeapify(numbers, 0);

    return min;
}
```

Figure 9: ExtractMin Function

Employee Class

Employee class contains 5 integers, 3 constructors, a function and an overridden operator "+=". These integers store main attributes, which are id, number of calls, number of positive feedbacks, number of negative feedbacks and performance, of an employee. Evaluate function calculates performance of an employee according to this formula:

PS = 2 * (number of calls) + (number of positive feedback) – (number of negative feedback)

Figure 10: Formula of the performance

For further details, please check “employee.h” header file.

```
class Employee {
public:
    int id;
    int callCount;
    int posCount;
    int negCount;
    int performance;

    Employee(){…
    }
    Employee(int _id, int _callCount, int _posCount, int _negCount, int index){…
    }
    Employee(const Employee &_employee){…
    }
    void Evaluate(){…
    }
    Employee & operator +=(const Employee &_employee){…
    }
};
```

Figure 11: Structure of Employee

Numbers / ExtractMax Analysis

For this part of the project 2,000,000 numbers are sorted by extracting max. Numbers used on the tables are rounded. Heights are calculated before steps are executed. Real values are:

	Time Elapsed(ms)	Height of the Heap(lgn)
Step 1	278.666	20.9316
Step 2	271.371	20.7796
Step 3	257.301	20.6096
Step 4	242.230	20.4170
Step 5	225.863	20.1946
Step 6	214.575	19.9316
Step 7	190.453	19.6096
Step 8	175.712	19.1946
Step 9	154.587	18.6096
Step 10	122.023	17.6096

Table 1: Real values of time elapsed and height of the heap

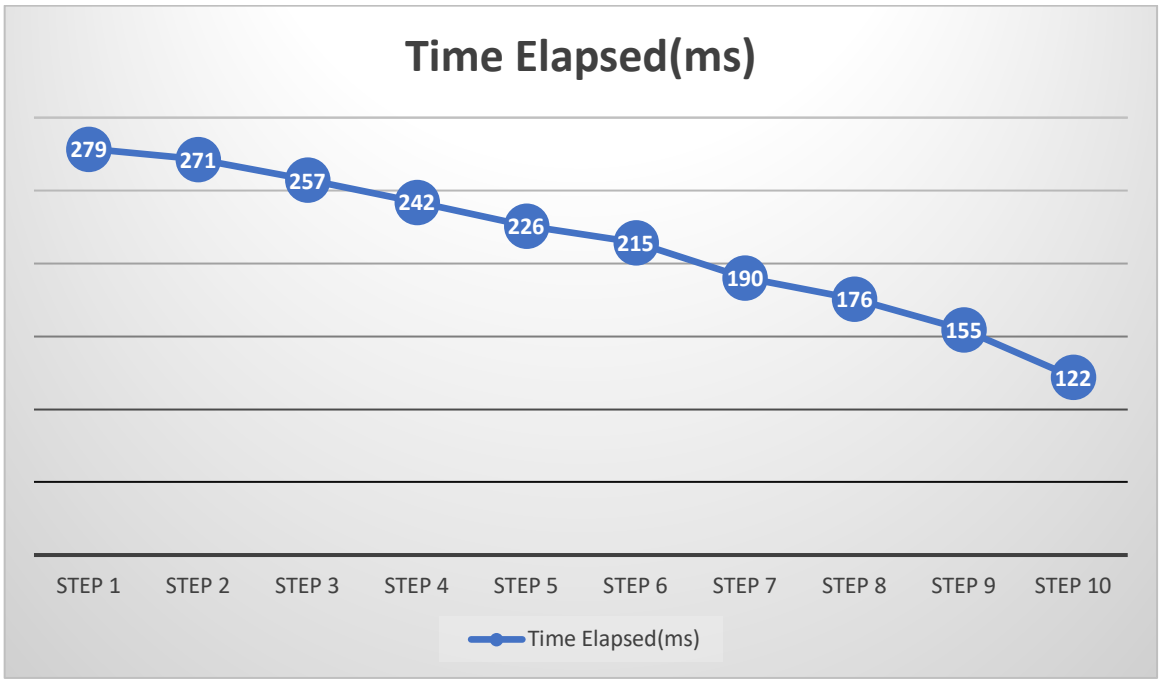

Figure 12: Time elapsed on every step

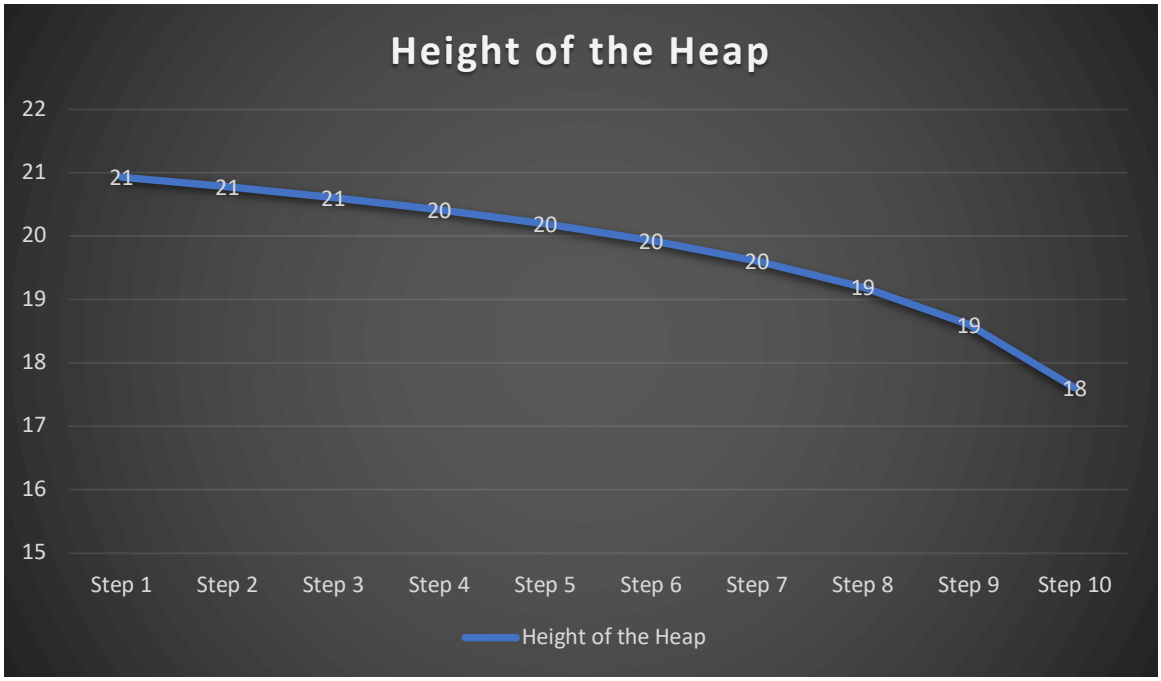

Figure 13: Height of the heap on every step (calculated before step is executed)

Notes for the code

I have used some functions, that are supported with C++11. Hence, you should use this command to compile with no errors and warnings:

```
g++ main.cpp -std=c++11
```