BLG 335E Project 2 Report

Ahmed Burak Gulhan
150160903

Compilation: use command g++ ./main.cpp ./heap_numbers.cpp ./node.cpp ./heap.cpp

a) Implementation

To implement the heapsort algorithm in order to use sort the day#.csv files I made and used two different classes, class codes and class heap.

Class node:

```
using namespace std;

class node{
//public:
    int id;
    int calls;
    int positive_feedback;
    int negative_feedback;
    int score;
  public:
    node(int, int, int, int);
    node();
    void add_calls(int, int, int, int);
    int get_id();
    int get_calls();
    friend bool operator> (const node &n1, const node &n2){
        return n1.score > n2.score;
    }
    friend bool operator< (const node &n1, const node &n2){
        return n1.score < n2.score;
    }
    node& operator+=(const node& rhs){
        this->calls += rhs.calls;
        this->positive_feedback += rhs.positive_feedback;
        this->negative_feedback += rhs.negative_feedback;
        score = 2*calls + positive_feedback - negative_feedback;
        return *this;
    }
};
```

The class node is used to store each employee id, number of calls, number of positive feedback, number of negative feedback and performance score. This class has operator overloading, which are used to compare the performance scores (int score). This class has getters in order to compare the amount of calls. When any value of the node is updated, the score automatically updates as well.

Class Heap:

```
1  #include "node.h"
2  using namespace std;
3  class Heap{
4  //public:
5      node *heap;
6      int heap_size = 0;
7      int array_size = 0;
8      int max_array_size;
9      int heap_type = 1;  ///0 min heap, 1 max heap
10     int max_id = 0;
11   public:
12     Heap(int,int);//
13     ~Heap();//
14     int get_array_size();
15     int get_id(int);
16     int get_max_id();
17     void max_heapify( int);/// according to performance
18     void min_heapify( int);/// according to performance
19     void build_max_heap();/// build heap according to performance
20     void build_min_heap();//
21     void heapsort( int, int);/// sorts according to performance
22     void insert(node);//
23     void extract_max();//
24     void extract_min();//
25     void increase_key(int, node);
26     int find_node(int);
27     void reset_heap(); ///adds removed items back to heap (maked heap_size = array_size)
28
29     void heapsort_calls( int, int);/// sorts according to number of calls
30     void build_max_heap_calls();/// build heap according to number of calls
31     void build_min_heap_calls();/// build heap according to number of calls
32     void max_heapify_calls( int);/// according to number of calls
33     void min_heapify_calls( int);/// according to number of calls
34     void increase_key_calls(int, node);
35 };
```

The heap class is used to make a heap, and is where the procedures required for sorting are located as methods. The heap is made up of an array of 'node's (the class node). When the array is being initialized it takes two inputs, one is the size of the heap and the other is the heap type (1 for max heap, 0 for min heap). The respective inputs are also stored in int max_array_size and int heap_type. The array also stores the values, int array_size and int heap_size. The value array size is total number of items in the array, this value increases with each insert. The heap_size is the length of the heap, this value is the same as array_size for unsorted heaps and decreases as the heap gets sorted. There is also a value int max_id. This value stores the newest employee ID (which are sequential) so that we can know if a node being inserted already exists or nor.

The class also contains some getters, which are self explanatory. The class contains the methods (which we were asked to implement in the homework): max_heapify, min_heapify, build_max_heap, build_min_heap, heapsort, insert, increase_key, extract_max and extract_min. These functions do the comparisons according to the *performance score* (int score). There are functions with similar names (lines 29-34) with '_calls' at the end of the names. These methods are the exact same as the previous methods, with the only difference being that the comparisons are done according to the *number of calls* (int calls). I will not explain these methods as they have the same algorithms as the previous methods. There are also two extra methods that I implemented, find_node and reset heap. Find_node finds the location of an ID. Reset_heap makes that value heap_size equal to array_size, which is used when a new day#.csv is being read.

Methods max_heapify() and min_heapify()

```
void Heap::max_heapify( int item){
    int l = (2*item)+1;
    int r = (2*item)+2;
    int largest = item;
    if (l < heap_size && heap[l] > heap[item]){
        largest = l;
    }
    else {
        largest = item;
    }

    if(r < heap_size && heap[r] > heap[largest]){
        largest = r;
    }

    if(largest != item){
        node temp(0,0,0,0);
        temp = heap[item];
        heap[item] = heap[largest];
        heap[largest] = temp;
        max_heapify(largest);
    }
}
```

```
void Heap::min_heapify( int item){
    int l = (2*item)+1;
    int r = (2*item)+2;
    int small = item;
    if (l < heap_size && heap[l] < heap[item]){
        small = l;
    }
    else {
        small = item;
    }

    if(r < heap_size && heap[r] < heap[small]){
        small = r;
    }

    if(small != item){
        node temp(0,0,0,0);
        temp = heap[item];
        heap[item] = heap[small];
        heap[small] = temp;
        min_heapify(small);
    }
}
```

Max_heapify and min_heapify are used to shift a node down into it's correct location in a min/max heap. These methods compare a parent node with it's two children, the smallest child (for max heap) or the largest child (for min heap) is exchanged with the parent node until the parent node is in the correct location in the heap.

Methods build_max_heap() and build_min_heap()

```
void Heap::build_max_heap(){
    for(int i = ((heap_size-1)/2); i>=0; i--){
        max_heapify(i);
    }
    heap_type = 1;
}

void Heap::build_min_heap(){
    for(int i = ((heap_size-1)/2); i>=0; i--){
        min_heapify(i);
    }
    heap_type = 0;
}
```

These two methods build a max or min heap. To do this the methods calls max_heapify (for max heap) or min_heapify(for min heap) for every inner node (not a leaf) inside of the array. After this the method sets the value heap_type accordingly.

Methods extract_max() and extract_min()

```
void Heap::extract_max(){
    heap_size--;
    node temp = heap[0];
    heap[0] = heap[heap_size];
    heap[heap_size] = temp;
}

void Heap::extract_min(){
    heap_size--;
    node temp = heap[0];
    heap[0] = heap[heap_size];
    heap[heap_size] = temp;
}
```

These two methods are exactly the same. The reason for two same methods with different names is to increase the readability of the code.

This method switches the first element, of the heap which is the max or min value of the heap depending on the heap type, and swaps it with the last value of the heap. Then it decreases the heap_size, since that value is now considered 'extracted' from the heap (it is still in the same array).

Method heapsort()

```
void Heap::heapsort( int amount_to_sort, int type){
    int end_key = 1;
    if (amount_to_sort <= array_size){
        end_key = heap_size-1 - amount_to_sort;
    }
    if (type == 0){
            build_min_heap();
        for(int i = heap_size-1; i>=end_key; i--){
            extract_min();
            min_heapify(0);
        }
    }
    else{
            build_max_heap();
        for(int i = heap_size-1; i>=end_key; i--){
            extract_max();
            max_heapify(0);
        }
    }
}
```

This method sorts the heap. It first checks the heap type, then it sorts according to the heap type. If the heap is a max_heap, the method calls build_heap then it calls extract_max and min_heapify for the first node. The method does this for all nodes, starting from the last node and up to the second last node. If the heap is a min heap, the algorithm is the same except with the min variants of the methods.

Method insert()

```
void Heap::insert(node item){

    //node new_node(item.get_id(), item.calls, item.positive
    if(array_size == max_array_size){
        cout << "array is full" << endl;
        //cout << "full " << item.get_id() << endl;
    }
    else{
        heap[array_size] = item;
        array_size++;
        heap_size = array_size; ///list loses heap property
        if (item.get_id() > max_id){
            max_id = item.get_id();
        }
    }
}
```

This method adds a new item to the and of the heap. The value of array_size is increased and heap_size is set as array_size, since the heap property is lost after this operation. This method also checks if the id of the node being added has been added before, by checking max_id. If not, max_id is updated.

Method increase_key(int)

```
void Heap::increase_key(int key, node increase){
    node new_node = heap[key];
    new_node += increase;
    if(heap_type == 1){/// if max heap
        if(new_node < heap[key]){
            heap[key] += increase;
            max_heapify(key);
        }
        else{
            heap[key] += increase;
            node temp;
            while (key > 0 && (heap[(key-1)/2] < heap[key]) ){ /// (key-1)/2 is parent node
                heap[(key-1)/2] = temp;
                heap[(key-1)/2] = heap[key];
                heap[key] = temp;
                key = (key-1)/2;
            }
        }
    }

    else{/// if min heap
        if(new_node > heap[key]){
            heap[key] += increase;
            min_heapify(key);
        }
        else{
            heap[key] += increase;
            node temp;
            while (key > 0 && (heap[(key-1)/2] > heap[key]) ){ /// (key-1)/2 is parent node
                heap[(key-1)/2] = temp;
                heap[(key-1)/2] = heap[key];
                heap[key] = temp;
                key = (key-1)/2;
            }
        }
    }
}
```

This method updates an existing key and places it in the correct position so that the heap property is maintained. If the heap is a max heap and the updated value is smaller than the previous value, then max_heapify is called and the node is moved down into the correct position. If the value is larger than the original value, then the node is moved up into the correct position. If the heap is a min heap a similar procedure is applied. If the new value is larger than the previous value, then min_heapify is called for that node, and it is moved downwards into the correct position. If the value is smaller, then the value is moved upwards into the correct position.

Method find_node(int id)

```
int Heap::find_node(int node_id){ ///search for node id, return key of node if found, else reutrn -1
    for(int i=0; i<array_size; i++){
        if(heap[i].get_id() == node_id){
            return i;
        }
    }
    return -1;
}
```

This method finds the location of an ID in the heap. If the id is not found, then the method returns -1. This method is used to find the location of an existing node to update.

Method reset_heap()

```
void Heap::reset_heap(){
    heap_size = array_size;
}
```

This method is used to make the heap size equal to the array size. This is used when the list becomes unsorted, such as a new day#.csv file being read.

b)Run-time calculations

Run time for max_heapify and min_heapify = O(lgn)

-max_heapify and min_heapify have similar algorithms, so proving one method is O(lgn) should also prove that the other is O(lgn)

```
43 void Heap::max_heapify( int item){
44     int l = (2*item)+1;
45     int r = (2*item)+2;
46     int largest = item;
47     if (l < heap_size && heap[l] > heap[item]){
48         largest = l;
49     }
50     else {
51         largest = item;
52     }
53 
54     if(r < heap_size && heap[r] > heap[largest]){
55         largest = r;
56     }
57 
58     if(largest != item){
59         node temp(0,0,0,0);
60         temp = heap[item];
61         heap[item] = heap[largest];
62         heap[largest] = temp;
63         max_heapify(largest);
64     }
65 }
```

```
void Heap::min_heapify( int item){
    int l = (2*item)+1;
    int r = (2*item)+2;
    int small = item;
    if (l < heap_size && heap[l] < heap[item]){
        small = l;
    }
    else {
        small = item;
    }

    if(r < heap_size && heap[r] < heap[small]){
        small = r;
    }

    if(small != item){
        node temp(0,0,0,0);
        temp = heap[item];
        heap[item] = heap[small];
        heap[small] = temp;
        min_heapify(small);
    }
}
```

Analyzing max_heapify

In max_heapify there is no loop, as we can see in lines 44 to 62. These steps are independent from n (heap size), so these lines are of the complexity O(1). However there is a recursive call is line 63.

The worst case for max_heapify would be if we analyzed the topmost node (root) and had to move it down to the leaf. This would mean we have to call this procedure h times, where h is the height of the tree. The worst case for h would be if the binary tree is not a full binary tree, instead the height is higher on the left side of the tree. In general we can say h <= 2n/3.

We can write the recurrence equation of max_heapify as: $T(n) <= T(2n/3)+O(1)$
by using the master method we find $T(n) = O(lgn)$

This is also true for min heapify.

Run time for build_max_heap and build_min_heap

```
void Heap::build_max_heap(){
    for(int i = ((heap_size-1)/2); i>=0; i--){
        max_heapify(i);
    }
    heap_type = 1;
}

void Heap::build_min_heap(){
    for(int i = ((heap_size-1)/2); i>=0; i--){
        min_heapify(i);
    }
    heap_type = 0;
}
```

These two procedures are O(n)

Since these two procedures call min_heapify and max_heapify for the same amount of nodes, and max_heapify and min_heapify have the same complexity we can say that build_max_heap and build_min_heap have the same complexity.

In max_heapify, we call max_heapify for all inner nodes. Since max_heapify is O(lgn) and the inner nodes linearly depend on the heap size n, it seems as if the complexity is O(nlgn), however this is not a tight bound. max_heapify is O(lgn) for the worst case, but when we are calling max_heapify for all inner nodes, there will not be a worst case for every node. Since most inner elements are the parent of a leaf, and have a height of 1.

By solving the equation:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

We find that build_max_heap is O(n).

Run time for extract_max and extract_min

```
void Heap::extract_max(){
    heap_size--;
    node temp = heap[0];
    heap[0] = heap[heap_size];
    heap[heap_size] = temp;
}

void Heap::extract_min(){
    heap_size--;
    node temp = heap[0];
    heap[0] = heap[heap_size];
    heap[heap_size] = temp;
}
```

These procedures do not depend of the size of a heap. All that is being done is that two elements in the heap are being swapped. Therefore we can say that these procedures have a O(1) complexity.

Run time for heapsort

```
105 void Heap::heapsort( int amount_to_sort, int type){
106         int end_key = 1;
107         if (amount_to_sort <= array_size){
108             end_key = heap_size-1 - amount_to_sort;
109         }
110         if (type == 0){
111                 build_min_heap();
112             for(int i = heap_size-1; i>=end_key; i--){
113                 extract_min();
114                 min_heapify(0);
115             }
116         }
117         else{
118                 build_max_heap();
119             for(int i = heap_size-1; i>=end_key; i--){
120                 extract_max();
121                 max_heapify(0);
122             }
123         }
124 }
```

This heapsort method works for both min heaps and max heaps, which has the same complexity for both. Lines 106 to 109 have a complexity of O(1) since they do not depend on n. Assuming we are sorting a max_heap, line 118 has a complexity of O(lgn). On line 119 we have a for loop, which will execute ‘k’ times. Inside the loop we have extract_max and max_heapify which have complexities O(1) and O(lgn) respectively. So we have:
O(1) + O(lgn) + O(1) + O(lgn) + k*(O(1) + O(lgn)) complexity. If we are sorting the entire heap, then k is equal to n (actually it is n-1, but say n for simplicity). Then our complexity is:
O(1) + O(lgn) + O(1) + O(lgn) + n*(O(1) + O(lgn)), and is simplified into O(nlgn).

If our k value is chosen small, such as 3 (in order to find the smallest 3 elements, or largest for min heap), then the complexity is
O(1) + O(lgn) + O(1) + O(lgn) + 3*(O(1) + O(lgn)) which has a complexity of O(lgn).

Runtime for insert

```
void Heap::insert(node item){

    //node new_node(item.get_id(), item.calls, item.positive
    if(array_size == max_array_size){
        cout << "array is full" << endl;
        //cout << "full " << item.get_id() << endl;
    }
    else{
        heap[array_size] = item;
        array_size++;
        heap_size = array_size; ///list loses heap property
        if (item.get_id() > max_id){
            max_id = item.get_id();
        }
    }
}
```

This procedure does not depend on n, the size of the heap. Also there are no loops in the code. Therefore it is O(1) complexity.

Runtime for increase_key

```
143 void Heap::increase_key(int key, node increase){
144     node new_node = heap[key];
145     new_node += increase;
146     if(heap_type == 1){/// if max heap
147         if(new_node < heap[key]){
148             heap[key] += increase;
149             max_heapify(key);
150         }
151         else{
152             heap[key] += increase;
153             node temp;
154             while (key > 0 && (heap[(key-1)/2] < heap[key]) ){ /// (key-1)/2 is parent node
155                 heap[(key-1)/2] = temp;
156                 heap[(key-1)/2] = heap[key];
157                 heap[key] = temp;
158                 key = (key-1)/2;
159             }
160         }
161     }
162
163     else{/// if min heap
164         if(new_node > heap[key]){
165             heap[key] += increase;
166             min_heapify(key);
167         }
168         else{
169             heap[key] += increase;
170             node temp;
171             while (key > 0 && (heap[(key-1)/2] > heap[key]) ){ /// (key-1)/2 is parent node
172                 heap[(key-1)/2] = temp;
173                 heap[(key-1)/2] = heap[key];
174                 heap[key] = temp;
175                 key = (key-1)/2;
176             }
177         }
178     }
179 }
```

This procedure should have a O(lgn) run time.

There are 4 different conditions which can happen. First is when the heap is a max heap, and after updating our key, the value is smaller then the previous value. In this case we call max_heapify to move the node down into the correct position, which has a complexity of O(lgn). The second case is when the heap is a max heap and after updating our key, the key is larger than the previous value, in this case we need to move the node up. To do this we execute the lines 152 to 158. Lines 152-153 are O(n). On line 154 there is a while loop, which will execute, in the worst case h times, where h is the distance of the current node from the root. If our node is a leaf, then h would be lgn. The while loop has a complexity of O(1) on the inside. So the total complex is O(1) + lgn(O(1)) which is simplified into O(lgn). For the third case our heap is a min heap and after updating our key, the value is larger than the previous key. In this case we call max heapify to move the node downwards, which has a complexity of O(lgn). The fourth case is when our heap is a min heap and after updating our key, the value is smaller than the previous key. In this case we execute the lines 169 to 175. These line are similar to the lines in case two and have the same complexity. So for this case we have a O(lgn) complexity.

For all cases we find a O(lgn) complexity, so this procedure is O(lgn)

Run time of find_node

```
int Heap::find_node(int node_id){ ///search for node id, return key of node if found, else reutrn -1
    for(int i=0; i<array_size; i++){
        if(heap[i].get_id() == node_id){
            return i;
        }
    }
    return -1;
}
```

This procedure iterates over the heap until the correct id is found. The worst case would be is the node is located at the end of the list. Therefore this procedure has a O(n) complexity.

Run time of reset_heap

```
void Heap::reset_heap(){
    heap_size = array_size;
}
```

This algorithm only changes a single value. Therefore it has a run time of O(1).

c)Sorting numbers.csv

Heap size: 2000000
Time taken to sort block 1 of 200k: 0.23613s
Heap size: 1800000
Time taken to sort block 2 of 200k: 0.188883s
Heap size: 1600000
Time taken to sort block 3 of 200k: 0.162743s
Heap size: 1400000
Time taken to sort block 4 of 200k: 0.155438s
Heap size: 1200000
Time taken to sort block 5 of 200k: 0.147405s
Heap size: 1000000
Time taken to sort block 6 of 200k: 0.138198s
Heap size: 800000
Time taken to sort block 7 of 200k: 0.126168s
Heap size: 600000
Time taken to sort block 8 of 200k: 0.115249s
Heap size: 400000
Time taken to sort block 9 of 200k: 0.105314s
Heap size: 200000
Time taken to sort block 10 of 200k: 0.0866062s

For this part a new class called heap_numbers was made. This class is similar to the heap class but it is more simple, as the numbers can be stored directly in the array without needing the class node.

The sorted list is saved to numbers_sorted.csv