

**BLG 335E ANALYSIS OF ALGORITHMS**

**HW#2**

**NAME: Ece Çınar**

**STUDENT ID: 150150138**

**DATE: 03.12.2019**

## A. INTRODUCTION

Second assignment of Analysis of Algorithms was a scheduler implementation. To explain briefly, I implemented a class named “*Heap*” which encapsulates several methods and data. I also implemented a struct named “*Node*” to represent the events properly. Heap class has three variables, six methods and a default constructor. Node struct contains three variables which are name of an event, type of it and time of its start or end depending on the type of node. Heap class has an array involving all nodes which are read from a text file and uses this array in its methods with other variables which will be explained in detail.

## B. LIBRARIES

I used *iostream* library in order to perform input/output operations and *fstream* to perform file operations. I included *string* library to store type and name of an event using string variables. I also used *vector* library of STL since size of the heap array should be dynamic. I used *push\_back*, *pop\_back* and *size* functions of the library to insert the nodes that are read from the text file, to remove the scheduled node from heap and to obtain size of the heap array respectively.

## C. NODE STRUCT

As explained above, this struct has three variables which are time, event name and event type. Each event corresponds to two nodes in the tree, this is why this struct is named as “Node”. Type of a node can either be “START” or “END”, and the time variable represents time of that action type. Names of the events are read from a given text file, two nodes are created for each event to represent start time and end time of that event.

```
typedef struct Node{    //struct node holds data of an event
    int time;
    string event_name;
    string event_type;
} Node;
```

Figure 1: Struct Node

## D. HEAP CLASS

Heap class has three variables. “*latest\_schedule*” holds time of the uttermost scheduled node. “*virtual\_clock*” starts from zero and counts one by one. “*array*” is actually a vector consisting of nodes.

```

class Heap{
private:
    vector<Node> array;    //heap array
    int virtual_clock;
    int latest_scheduled; //holds the latest scheduling time
public:
    Heap(){ virtual_clock = 0; } //virtual clock initialized to 0
    void setArray(Node element){ array.push_back(element); } //setArray is used to fill the array node by node
    void display();
    void minHeapify(int);
    void buildMinHeap();
    void scheduleEvents();
    bool isEventScheduled();
};

```

Figure 2: Heap Class

Default constructor of the class initializes virtual clock with zero. “*setArray*” takes a node struct as parameter and adds that node to the array. “*display*” can be used to view the array content instantaneously but it is not actually included in the program flow. “*minHeapify*” takes an index parameter and converts the subtree starting from that index into a minimum heap. “*buildMinHeap*” converts the whole tree into a minimum heap by calling *minHeapify* multiple times. “*isEventScheduled*” compares current value of *virtual\_clock* to the current top element of heap, returns 1 if they are equal and 0 if they aren’t. “*scheduleEvents*” calls *isEventScheduled*, checks if the top node can be scheduled, takes action according to the state and pops the top node if an action was taken. This process continues until heap is emptied. It also prints a message describing the current scheduling for each *virtual\_clock* tick.

Some of these functions are explained in detail below. Implementations of the default constructor, *setArray* function and *display* function are not included since they are less complex compared to the others and can be understood from the code itself.

## 1. minHeapify

This function converts a subtree into a tree with minimum heap property. A tree has minimum heap property if parent node value is smaller than its’ children always. The function takes a node index parameter and calculates its left and right children’s indexes. It compares time values of the parent and its’ children to choose the one with the smallest time value and carries it to the parent node. This is a recursive function since it calls itself at the end with the node index which had its’ value exchanged with the parent node. Since this exchange can harm the other subtrees ‘ minimum heap property, recursion has to be done.

```

void Heap::minHeapify(int idx){ //function to min heapify any node
    int l = 2 * idx + 1;    //left index calculation formula
    int r = 2 * idx + 2;    //right index calculation formula
    int smallest = idx;
    Node temp;
    int size = array.size();

    if(l < size && array[l].time < array[idx].time)
        smallest = l;
    if(r < size && array[r].time < array[smallest].time)
        smallest = r;
    if(smallest != idx){
        temp = array[idx];
        array[idx] = array[smallest];
        array[smallest] = temp;
        minHeapify(smallest);
    }
}

```

Figure 3: minHeapify

## 2. buildMinHeap

This function converts the entire tree into a tree with minimum heap property. This is done by calling minHeapify function for all interior nodes and the root node of the tree. Leaf nodes are not included since they don't have any children.

```

void Heap::buildMinHeap(){ //function to build min heap
    int size = array.size();
    for(int i = size / 2 - 1; i >= 0; i--)
        minHeapify(i);
}

```

Figure 4: buildMinHeap

## 3. isEventScheduled

This function compares top node of the array (I use array interchangeably with tree in this report since we act as if the array is a tree) with the virtual\_clock. If time value of the top node is equal to the virtual\_clock, this means that there is a node to be scheduled right now. Function stores current time using a variable called latest\_scheduled and returns 1. Returning 1 indicates that a node is to be scheduled in scheduleEvents function. The variable latest\_scheduled will be used to distinguish between two different states which occur when time value of the top node is greater than the virtual\_clock: one is if an event was scheduled during this clock tick before, and the other is when an event wasn't scheduled during this clock tick. Returning 0 means that a node isn't to be scheduled in scheduleEvents function.

```

bool Heap::isEventScheduled(){
    if(array[0].time == virtual_clock){
        latest_scheduled = virtual_clock;
        return 1;    //top event is scheduled
    }
    return 0;    //top event is not scheduled yet
}

```

Figure 5: isEventScheduled

#### 4. scheduleEvents

This function calls all the functions above directly or indirectly. First, it increments virtual clock by one and makes it one. Secondly, it calls buildMinHeap to obtain minimum heap property. Then, a for loop which will be executed size times is started. At the beginning of the loop, there is a while loop which checks the return value of isEventScheduled continuously. If an event isn't to be scheduled, isEventScheduled returns zero. Then, while loop checks if latest\_scheduled value is equal to the virtual\_clock. If so, this means that a node was scheduled in this virtual\_clock time before but now there is another node to be scheduled in another clock tick, so no message is printed. If they are not equal, this means that there wasn't and there won't be any nodes being scheduled in this time\_clock. So a "NO EVENT" message is printed on the screen. After incrementing virtual\_clock, while loop continues checking the return value of isEventScheduled.

If isEventScheduled returns 1, while loop terminates and prints a message describing the action taken. After, top node is swapped with the last node and the last node is popped. This is how a scheduled node is removed from the array. Then, minHeapify is called with index 0 since top node is exchanged with the last node and the tree must be heapified again to get the node with the smallest time value to the top. During each execution of the for loop, a node is scheduled and popped from the array. So the heap is empty when the loop ends. At the end, a last message is printed to inform the user that there aren't any nodes left to be scheduled.

```

void Heap::scheduleEvents(){    //function to schedule the nodes
    int size = array.size();
    Node temp;
    virtual_clock++;    //virtual clock starts ticking, currently = 1
    buildMinHeap();
    for(int i = size - 1; i >= 0; i--){ //for loop will be executed #size times, each time smallest element will be popped
        while(!isEventScheduled()){ //while execution time of top node(smallest node) is larger than virtual clock
            if(latest_scheduled!=virtual_clock)
                cout << "TIME " << virtual_clock << ": " << "NO EVENT" << endl;
            virtual_clock++;    //clock is incremented
        }
        cout << "TIME " << virtual_clock << ": " << array[0].event_name << " " << array[0].event_type << "ED" << endl;
        temp = array[0];
        array[0] = array.back();
        array.back() = temp;
        array.pop_back();    //smallest node is moved to the end scheduled and popped from the array
        if(i)
            minHeapify(0); //heapify the root node
    }
    cout << "TIME " << virtual_clock << ": NO MORE EVENTS, SCHEDULER EXITS" << endl; //heap empty, nothing more to be scheduled
}

```

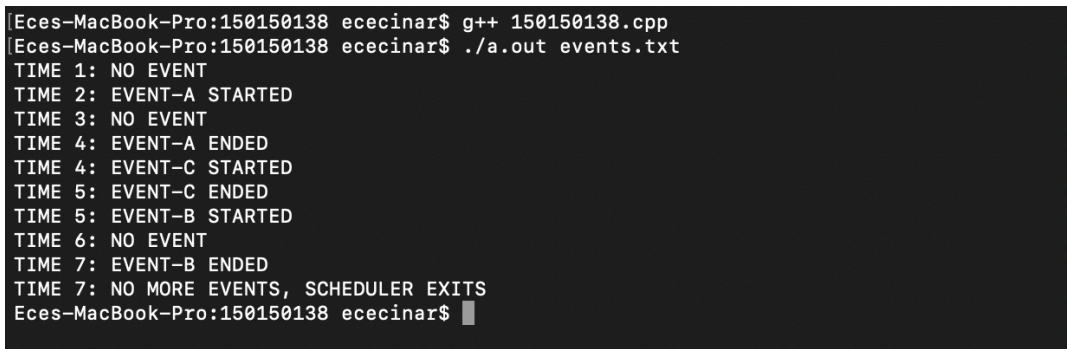
Figure 6: scheduleEvents

## E. MAIN

In main, I created a Heap object, a Node instance and some other variables to perform read operations on. User should supply a file name parameter from the command line. Then, main tries to open that file in read mode and throws exception if the file cannot be opened or if it doesn't exist. If the file can be opened, program reads event name, event start time and event end time from the file till EOF. From each line, two nodes are created. First node represents start of an event and the second represents end of that event. These two nodes are created using the instance of Node struct and then they are pushed to the array of Heap object which was created at the beginning of main. SetArray function is used to push all nodes to the array one by one. After that, scheduleEvents is called to schedule the nodes of array.

## F. CONCLUSION

This program is capable of reading events from a file, creating two nodes for each and scheduling those nodes correctly. It can also handle the cases in which two nodes have to be scheduled at the same clock tick. The image below shows program output for the text file given in the homework.

A terminal window screenshot showing the execution of a C++ program. The prompt is 'Eces-MacBook-Pro:150150138 ececinar\$'. The first command is 'g++ 150150138.cpp' and the second is './a.out events.txt'. The output shows a sequence of events over time: 'TIME 1: NO EVENT', 'TIME 2: EVENT-A STARTED', 'TIME 3: NO EVENT', 'TIME 4: EVENT-A ENDED', 'TIME 4: EVENT-C STARTED', 'TIME 5: EVENT-C ENDED', 'TIME 5: EVENT-B STARTED', 'TIME 6: NO EVENT', 'TIME 7: EVENT-B ENDED', and 'TIME 7: NO MORE EVENTS, SCHEDULER EXITS'. The prompt returns to 'Eces-MacBook-Pro:150150138 ececinar\$' with a cursor.

```
Eces-MacBook-Pro:150150138 ececinar$ g++ 150150138.cpp
Eces-MacBook-Pro:150150138 ececinar$ ./a.out events.txt
TIME 1: NO EVENT
TIME 2: EVENT-A STARTED
TIME 3: NO EVENT
TIME 4: EVENT-A ENDED
TIME 4: EVENT-C STARTED
TIME 5: EVENT-C ENDED
TIME 5: EVENT-B STARTED
TIME 6: NO EVENT
TIME 7: EVENT-B ENDED
TIME 7: NO MORE EVENTS, SCHEDULER EXITS
Eces-MacBook-Pro:150150138 ececinar$
```

Figure 7: Sample Execution