ANALYSİS of ALGORİTHMS – 1

BLG – 335E

ASSIGNMENT 2 REPORT

Name: Hakan SARAÇ

ID: 150140061

Introduction

In this project, it is asked to implement heap sort using min heapify. Heap sort is used for calculate when events will be happened respectively. Events are pulled from text file and sorted according to start and end time of events. Lastly, the project prints out events according to a virtual clock time with their name and type. If at the virtual clock there is no event, it prints out “no event” message or all of events have happened it prints out “no more events” message.

Min-Heapify, build min and heapsort methods are used to sort events time.

Min-Heapify

When implementing Min-Heapify, first of all the elements of array are placed to a tree based on their index. After that, Min-Heapify(A, index) function (A parameter is array, index parameter is index of an element of array) sorts the parent which is given index parameter and its children. When the function sorting these elements, the parent of children cannot be bigger than its own children.

An example of Min-Heapify(A, index):

Array Index	0	1	2	3	4	5	6
Input		40	60	10	20	50	30

Figure 1

Firstly, we implement array(Figure 1) as a tree like Figure 2. After that the Min-Heapify(A, 1) function sorts the A[1] element and its children. The rule is that parent node cannot be bigger than its children.

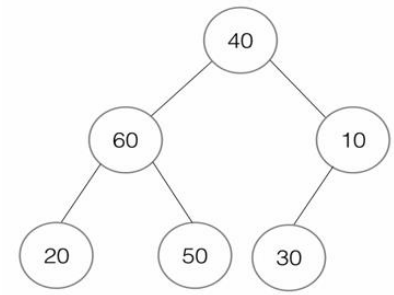

Figure 2

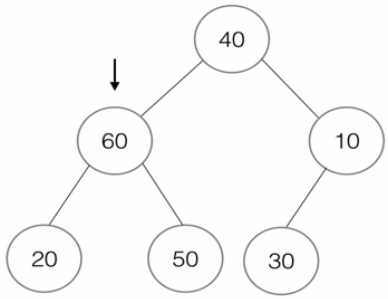

Figure 3

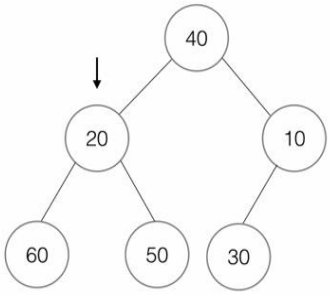

Figure 4

After implement Min-Heapify(A,1) function Figure 3 turns into Figure 4.

In this assignment, our events have name, type and time. Therefore, we have a struct which consists of event_name, event_type and event_time (Figure 5). We sort the events based on their time value.

```
typedef struct event_node{
    int event_time;
    string event_name;
    string event_type;
}event_node;
```

Figure 5

To find left and right children of the parent node, we use this formalization in Min-Heapify function:

Left child index = (2 * parent index) + 1

Right child index = (2 * parent index) + 2

```
int smallest = i;
if(left<size && elements[left].event_time < elements[i].event_time) smallest = left;
if(right<size && elements[right].event_time < elements[smallest].event_time) smallest = right;
```

Figure 6

In Figure 6, the Min-Heap method calculating the smallest element among a parent and its children. If smallest event_time value is not parent's, change the parent and smallest one.

Running time of Min-Heap

In the worst case, there must be 2/3 of the tree might be involved. The parent node have at most 2n/3 children. Therefore T(n) = O($\log_2 n$)

Build Min

Build min method uses a for loop to sort all nodes of tree by using Min-Heap method. For loop begins from Floor(array size / 2) and stops at first element of array. To continue our example:

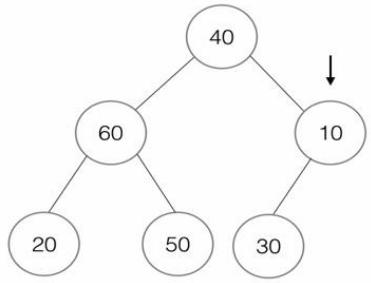

Figure 7

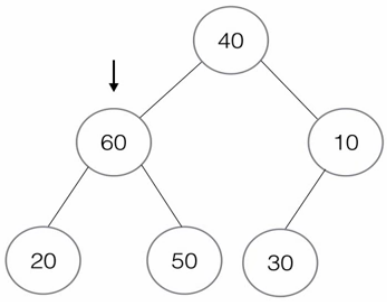

Figure 8

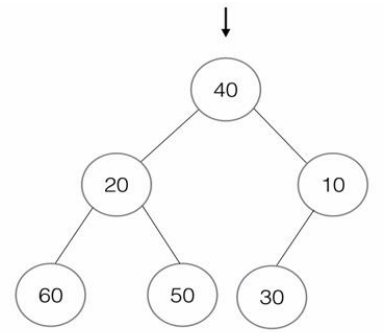

Figure 9

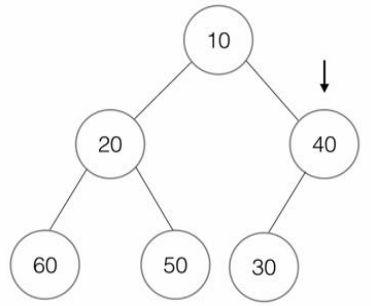

Figure 10

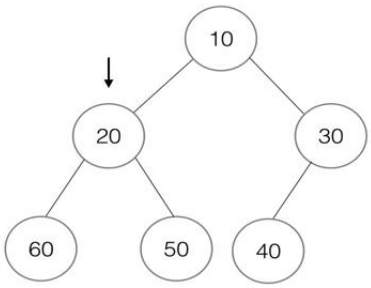

Figure 11

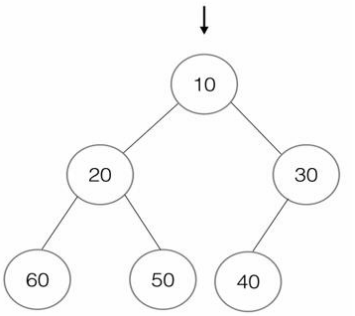

Figure 12

For each loop turn, build min calls the Min-Heap function and when for loop finished, all of the tree is sorted. We can see that in tree Figure 12, each parents are smaller than their children.

```
void event_array::Build_Min(){  //this function is :
    int i;
    int heap_size = size;
    for(i=heap_size/2; i>=0; i--)   Min_Heapify(i);
}
```

Figure 13

In the method I started loop from size/2 because size and 2 are integer values and int/int gives the result as its floor value.

Running time of Build Min

Each Min-Heap call costs O(logn) and there are n nodes in a tree so that running time must be at most O(nlogn). But we do not call the leaves nodes. Therefore, tighter bound of Build Min is lower than O(nlogn).

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

Figure 14

Result of Figure 14 is O(n) which value the tighter bound of Build Min.

Heapsort

In Heapsort method build min called firstly. After this call, all tree nodes are sorted with Min-Heap method according to time of each event. And before control of event times, virtual clock is increased since virtual clock begins from 0. After that for loop is started and virtual clock is increased until the next event is happened. Since the smallest event_time is at top of the tree, we compare the virtual clock with first element of array. When an event is happened, it is removed from array and we call the Min-Heap function to sort array again to find next event to happen. The Heapsort function is doing these steps until there is no event anymore.

```
event_node temp;
int flag = 0;       //this flag is to control multiple events at same virtual time
                    //if there are an event equal to virtual time flag is 1, ow 0
Build_Min();
int temp_size = size;
virtual_clock++;
for(int i=temp_size-1; i>=0 ; i--){
    if(!flag){      //do not enter the function if there is an event equal to virtual time to control there may be another event at same vitual time
        while(virtual_clock<elements[0].event_time){
            cout << "TIME " << virtual_clock << ": NO EVENT" << endl;
            virtual_clock++;
        }
    }
    if(virtual_clock==elements[0].event_time){
        cout << "TIME " << virtual_clock << ": " << elements[0].event_name << " " << elements[0].event_type << "ED" <<endl;
        flag = 1 ;
    }
    else{
        flag = 0;
        virtual_clock++;
        while(virtual_clock<elements[0].event_time){
            cout << "TIME " << virtual_clock << ": NO EVENT" << endl;
            virtual_clock++;
        }
        if(virtual_clock==elements[0].event_time){
        cout << "TIME " << virtual_clock << ": " << elements[0].event_name << " " << elements[0].event_type << "ED" <<endl;
        flag = 1;
        }
    }
    temp = elements[i];
    elements[i] = elements[0];
    elements[0] = temp;
    size--;
    Min_Heapify(0);
    if(i==0) cout << "TIME " << virtual_clock << ": NO MORE EVENTS, SCHEDULER EXITS" << endl;    //last event happened
}
}
```

Figure 15

In addition, when heapsort finds an event_time which equals to virtual time, gives a message “its time, its name and its type”. If there is no event_time which equals to virtual time, the virtual time is increased until an event happenes and in this condition program prints out “no event” message.

Running time of Heapsort

Pseudocode of Heapsort and time complexity of each line are given blow:

Steps	Time complexities
Build Min(A)	O(n)
for i =length of A to downto 2 do	O(n-1)
exchange A[1] <-> A[i]	O(1)
size <- size-1	O(1)
Min-Heap(A, 1)	O(logn)

Table 1

T(n) = O(n) + O(n-1)*[O(1) + O(1) + O(logn)] or just **O(nlogn)**

Conclusion

In conclusion, when each call Min-Heap from heapsort, we get smallest value of tree at top of tree which means first element of array. In this assignment, I sort event based on event_time and when an event happened at virtual time, it is printed out with its time, name and type and this event is removed from tree and heapsort call the Min-Heap again. These steps happens until all steps have done. An example from the hw2.pdf(Figure 16) and its print out(Figure 17) are given blow:

events.txt

EVENT-A	2	4
EVENT-B	5	7
EVENT-C	4	5

Figure 16

```
hakan@hakan:~/Desktop/algo1$ g++ 150140061.cpp -o a
hakan@hakan:~/Desktop/algo1$ ./a 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
hakan@hakan:~/Desktop/algo1$
```

Figure 17