BLG335E: Analysis of Algorithms I Report #2

Uğur Ali Kaplan - 150170042

December 10, 2019

1 Assignment

In this homework, we are asked to implement an event-scheduler, using min-heap. User provides the name of the file that contains the event names and associated starting and ending times of those events.

An example of how inputs are structured is here:

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

Listing 1: Text File

2 Programming Language and Compilation Process

I have used C++11 in this assignment. To compile the program, the following command can be run:

`g++ -std=c++11 -Wall -Werror 150170042.cpp -o scheduler`
where "scheduler" is the name of the executable.

Program can be executed in the following manner after compiling:

`./scheduler events.txt`
where "events.txt" is the name of the input file. Users can specify other names as name of the input file.

3 Code and Explanation

Our first goal is to read the inputs, and create the nodes of the heaps accordingly. In order to do that, I have utilized `fstream` and `vector` libraries. Also, in case of an empty file, or maybe some kind of a reading error on the software side, I have used `cstdlib` to exit the program. To print the error messages, I have used `iostream` and we also specify that we are using the `standard` namespace with `using namespace std`.

```
int main(int argc, char * argv[]) {
    if(argc != 2) {cerr << "One file name must be specified." << endl; exit(1);}
    ifstream in_file(argv[1]); // Open the input file
    int size = 0; // Heap Size

    if(!in_file) { // If were unable to open the file
        cerr << "File could not be read" << endl;
        exit(1);
    }

    vector<Node> nodeList; // Number of events is unknown, therefore we use vectors

    string name; // name will hold the name of the event

    while(in_file >> name){ // Read lines from the file
        int start, end; // Start and end times
        in_file >> start;
        in_file >> end;
        if(start < 1 || end < 1 || start > end){
            cerr << name << " cannot be used, times are incorrect" << endl;
            continue;
        }
        nodeList.push_back(Node(name, true, start)); // Append a start node
        nodeList.push_back(Node(name, false, end)); // Append an end node
        size += 2; // Keep track of how many nodes we have added
    }

    in_file.close(); // Clean after yourself

    if (size == 0) {cerr << "File is empty!" << endl; exit(1);} // If there are no nodes, exit
```

Listing 2: Snippet 1

As previously stated, name of the input file is provided by the user as a commandline argument. We create a `input file stream` using the filename which is provided to us with `argv[1]`.

Then, since we have to know how many nodes there are in the heap, we create a `size` variable that will count how many nodes do we have.

If we have failed to create an input file stream, we send an error message and exit the program. If we are successful in reading the name of the file, we continue with creating a vector of `Node`s, which will be examined in more detail in the rest of the report. The reason I am using a vector is that, there is no information provided to us on how many events there are before we have read the file. Since we do not want to read the file twice, we are using `vector`s which dynamically increase their capacity as more elements are added to them.

Then, we create a variable of type `string`, which is called `name`. This variable will hold the names of the events, and will be passed to the constructor of Node class in the future.

We use a `while` loop to read the names and times of the events, as it can be seen from the snippet above. Then, if given times are less than 1, or start time is later than the end time, we print an error message and skip that event. If everything seems to be in order, we create two nodes: one with the start

time as key and the other with the end time as key. Then, we also increase the size as we have added 2 nodes.

After we are finished with reading the file, we close input file stream. Then, we check the size and if there are no nodes created, we print an error message indicating that the file is empty and we exit the program.

So far, we have managed to read the file, and create the necessary nodes for the heap.
Rest of the `main` function is as follows:

```
    Scheduler schedule(&nodeList[0], size); // Initialize the scheduler object

    schedule.tickTock(); // Run the timer

    return 0;
}
```

Listing 3: Snippet 2

We initialize a `Scheduler` class, and pass the first element of the vector and size of the heap. Since elements of vectors are stored in contiguous memory as specified in the standard, we are free to act like we get the first element of an array.
Then, we use the `tickTock` method of the `schedule` object, and then the program finishes.

Now, we can examine how `Node`s are structured. Here is the definition of a `Node` class and its members:

```
class Node{
private:
    string name;
    bool start; // If true, this is a "START" node, else this is an "END" node
    unsigned int time;
public:
    Node(string name, bool start, unsigned int time);
    unsigned int getTime() {
        return time;
    }
    void printNode();
};

void Node::printNode() {
    /*
     * Prints the name of the event and type of the event (e.g. EVENT-A STARTED, EVENT-B ENDED ...)
     */

    cout << name << " ";
    if (start) cout << "STARTED";
    else cout << "ENDED";
    cout << endl;
}

Node::Node(string name, bool start, unsigned int time) {
    this->name = name;
    this->start = start;
```

```
28      this->time = time;
29  }
```

Listing 4: Node Class

`Node` class has 3 private attributes and 3 public methods one of which is a constructor. `name` holds the name of the event(e.g. "EVENT-A"), `start` is a `boolean` variable which indicates if this node is the start or end of an event, and `time` is the key of the node, which is either the start or end time of the event.

Constructor takes 3 arguments and previously mentioned attributes get initialized here.

`getTime()` returns the value of the `time` attribute.

`printNode()` prints the name of the event and if it is a start node or end node. For example, if this node is an end note, and name attribute has `"EVENT-A"`, `"EVENT-A ENDED"` will be printed on screen.

Backbone of our program is in the `Scheduler` class. Here it is:

```
1   class Scheduler{
2   private:
3       Node *heap; // Hold an array of nodes, called heap
4       int heap_size; // How many elements do we have?
5       unsigned int T; // Timer
6   public:
7       Scheduler(Node *heap, int size);
8       unsigned int heapMinimum();
9       void heapExtractMin();
10      void minHeapify(int index);
11      void buildMinHeap();
12      void printHeap();
13      void tickTock();
14  };
15
16  Scheduler::Scheduler(Node *heap, int size) : T(1){
17      this->heap = heap;
18      this->heap_size = size;
19      buildMinHeap();
20  }
21
22  unsigned int Scheduler::heapMinimum() {
23      /*
24       * Returns the key of the minimum element
25       */
26      return heap[0].getTime();
27  }
28
29  void Scheduler::heapExtractMin() {
30      /*
31       * Prints the minimum element and discards it out of the heap.
32       */
33      Node value = heap[0];
34      Node temp = heap[heap_size - 1];
```

```
35      heap[heap_size - 1] = heap[0];
36      heap[0] = temp;
37      heap_size -= 1;
38      minHeapify(0);
39      value.printNode();
40  }
41
42  void Scheduler::minHeapify(int index) {
43      /*
44       * Changes the heap so that index it is called on satisfies the min-heap property.
45       */
46      int left = (2 * (index + 1)) - 1;
47      int right = left + 1;
48      int min = index;
49      if (left <= (heap_size - 1))
50      {if (heap[left].getTime() < heap[min].getTime()) min = left;}
51      if (right <= (heap_size - 1))
52      {if (heap[right].getTime() < heap[min].getTime()) min = right;}
53      if(min != index){
54          Node temp = heap[index];
55          heap[index] = heap[min];
56          heap[min] = temp;
57          minHeapify(min);
58      }
59  }
60
61  void Scheduler::buildMinHeap() {
62      /*
63       * Builds a min-heap out of the unordered array it is given.
64       */
65      int start_index = floor(heap_size - 1 / 2.0);
66      for(int i = start_index; i >= 0; i--){
67          minHeapify(i);
68      }
69  }
70
71  void Scheduler::printHeap() {
72      /*
73       * Prints all elements of the heap, can be used for debugging purposes.
74       */
75      for(int i = 0; i < heap_size; i++){
76          heap[i].printNode();
77      }
78  }
79
80  void Scheduler::tickTock() {
81      /*
82       * Runs the timer, prints the events that are happening.
83       * Stops the timer if there are no more events.
84       */
85      while (heap_size > 0){
86          if (T != heapMinimum()) cout << "TIME " << T << ": NO EVENT" << endl;
87          while (T == heapMinimum() && heap_size > 0){
88              cout << "TIME " << T << ": ";
```

```
89              heapExtractMin();
90          }
91          T++;
92      }
93      T--;
94      cout << "TIME " << T << ": NO MORE EVENTS, SCHEDULER EXITS" << endl;
95  }
```

Listing 5: Scheduler Class

`Scheduler` has 3 private attributes and 7 public methods one of which is its constructor.

`heap` is a pointer to `Node` type of object, which will be used as an array.

`heap_size` holds the number of elements in the heap.

`T` is here to run the timer. It gets initialized as 1 when an object of class `Scheduler` is created.

In the constructor, we initialize the other 2 attributes with the information provided to us and then we call another method of the Scheduler, called `buildMinHeap()`.

`heapMinimum()` returns the key of the node in the root of the heap, which is also the minimum element since it has to satisfy the min-heap property.

`heapExtractMin()` calls the `printNode()` method of the root and discards it from the heap. Since we have another node in the root now, we call another method of Scheduler, called `minHeapify()` to make sure root element satisfies the min-heap property after the function ends.

`minHeapify(int index)` takes an index as an argument, and then determines if it is indeed smaller than its left and right nodes. If is smaller than both of them, nothing happens. But if it is not, it is swapped with the minimum of the three nodes and then, `minHeapify` is recursively called on the newly swapped node until it satisfies the min-heap property.

`buildMinHeap()` is used to build a min heap out of the unordered array given to Scheduler class when it is first initialized. To use the `floor` function, I have used `cmath` library. To do that, it calls `minHeapify` on the nodes of the heap, except for the leaves of the heap. Since leaves do not have left and right nodes, it would be a waste of time to call minHeapify on them.

Then we have `printHeap()`, which prints the elements of the heap. It was used for debugging purposes when the program was being developed but it has no uses in the end-program.

`tickTock()` starts the timer, and runs until there are no nodes left in the heap. When the key value of a node matches the timer value, it calls `heapExtractMin()`.

4 Running Time

If we have n events, we create $2n$ nodes. Building the minimum heap takes $O(n)$ time. Priority queue operations take $O(\log n)$ time. Since we use a timer until there are no more nodes left in the heap, we have

to run the loop until $T = finaltime$.

1. $O(n)$ from the construction of the min-heap.
2. $O(finaltime)$ from the timer loop.
3. $O(n \log n)$ from the fact we call minExtractHeap() n times.

So, the running time depends on the final time. If there is a limit on the greatest value final time can take, we can say running time is $O(n \log n)$. Else, there is no upper bound that depends on the input size but rather, running time depends on the greatest key value.