COP 5536 Project Report

Anushka Sudhir Linge

UFID: 77530821

UF EMAIL: alinge@ufl.edu

I have used the Java programming language (version 8) and the NetBeans IDE to write the source code of this project. The algorithm that I developed to solve the problem statement is as follows:

1. Maintain a variable called 'global' that initially stores the value 0 (as specified in the problem statement
2. Create a MinHeap (using the MinHeap and the BuildingNode classes)
3. Create a Red Black Tree (using the RBTree and RBNode classes)
4. Insert the first building into the minheap and the rbtree
5. Increment the execution time of the root node of the minheap, either by 5 or the number of days left for the completion of the building, whichever is smaller
6. Increment the value of 'global' by the same amount that we used in step 5
7. Iterate through all the commands in the input file, for which the counter value is less than or equal to the value of 'global'. These commands are either an Insert command or a Print command. For the Insert command, do the insert into the minheap and the rbtree. For the Print command, print the triplet.
8. Go to step 5 and keep executing till either the minheap becomes empty or all the commands are read from the input file and executed.

The structure of my program is as follows:

1. The RisingCity class contains the main method, and this is where the execution of the program begins. It has four functions written in it, the first one is the main function. The others are the printOneBuilding, printBuilding, printMultipleBuildings functions, which prints the triplets of building number, execution time and total time of a building.
2. The BuildingNode class defines the structure of the basic unit of the minheap i.e. a node of the minheap. This class stores the three essential attributes i.e. building number, the execution time and the total time. It also stores a reference of the corresponding node in the red black tree. The fifth attribute is the start day. This

attribute is required to correctly calculate the number of days spent on building a particular building later, when a Print command is encountered in the input file.

3. The MinHeap class defines one of the data structures of this program, which is the minheap. The underlying data structure used is an array of type BuildingNode. Thus, each entry of the array stores a reference to a BuildingNode object. There is an attribute called ‘current’ which stores the value of the next empty position in the array where a BuildingNode can be inserted.
4. The RBNode class defines the basic unit of the red black tree. This node stores the attributes left, right and parent in order to maintain information about the parent-child relationships among the different nodes of the red black tree.
   The underlying data structure that has been used to implement the red black tree is a tree data structure. An essential and important attribute of the RBNode class is the ‘bNum’ attribute. This stores the building number of the various buildings that are passed to the program.
   This is an important attribute because the red black tree is structured on the basis of the values of these keys.
5. The RBTree class defines the second data structure of the program, which is the red black tree. The red black tree is required in our program because of its (log n) runtime to perform a search operation. A red black tree is essentially a binary search tree, whose structure is optimised for faster access times.
   The two main properties that a red black tree must satisfy are: 1) there cannot be two consecutive red nodes in any path from the root to an external node and 2) the number of black nodes on any path from the root to an external node should be the same.
   Every time an insert operation or a delete operation is performed on the red black tree, a violation of the above two properties may occur. In order to transform the tree back into a red black tree, we need to perform left and right rotations, which are similar to the rotations performed in an AVL tree.

The above are the five classes defined in the program.

The structure of the classes is as follows:

class BuildingNode {

Building number;

Executed time;

Total time;

Start day;

Corresponding Red Black Node;

// there is a constructor defined in this class to initialise values of all attributes

BuildingNode(int buildingNum, int executedTime, int totalTime, RBNode rbn)

}

class MinHeap {

// an array of the BuildingNode type of a predefined size of 2005 is created

BuildingNode Heap[] = new BuildingNode[2010];

// this stores the value of the next position of the Heap array where insertion should be performed

int current;

// this function accepts an array index and computes its parent index

public int parent(int index)

// this function accepts an array index and computes the index of its left child

public int left_child(int index)

// this function accepts an array index as an argument and computes the index of its right child

public int right_child(int index)

// this function accepts the three building arguments, creates a new BuildingNode, and inserts it at the correct position in the minheap by calling the buildMinHeap function

public BuildingNode insert(int buildingNum, int executedTime, int totalTime, RBNode rbn)

// this function pushes the newly inserted node upwards in the minheap, also taking into account the building numbers of this node and the parent under consideration, to tackle the ties that result due to duplicate execution times

public void buildMinHeap()

// this function extracts the root node, inserts the last leaf node in its place, and calls minheapify

public BuildingNode extractMin()

// a function to help visualise the structure of the minheap

public void printMinHeap()

// a function to go through the Heap array in order and print its elements

public void printHeap()

// the most important function in this class, it is called repeatedly at different points in the program to ensure that the minheap property is always satisfied

public void minHeapify(int i)

}

class RBNode{

// these attributes store the reference of parent, left and right children of a node

// the nullValue attribute is created to be used in place of null in the program

RBNode nullValue;

RBNode left;

RBNode right;

RBNode parent;

int bNum;

BuildingNode heapNode;

RBTree.COLOR color;

// three constructors have been defined to initialise an RBNode object in different ways

}

class risingCity{

// this function searches the red black tree, fetches the node with the desired building number and then prints the triplet

public static void printBuilding(BufferedWriter wr,RBTree tree,int bN,int counter,int global)

// this function is called for a PrintBuilding command with only one argument i.e. for one building

public static void printOneBuilding(String line,RBTree tree,BufferedWriter wr,int global)

// this function is called for a PrintBuilding command with two arguments i.e. for a range of buildings

public static void printMultipleBuildings(String line,RBTree tree,BufferedWriter wr,int global)

// this function encapsulates the main logic of the program

public static void main(String[] args)

}

class RBTree{

// the attribute ‘root’ stores the reference of the root node of the red black tree

RBNode nullValue;

RBNode root;

// this stores the two possible color values of a node of a red black tree, either red or black

public enum COLOR

// constructor to create and initialise the red black tree

public RBTree()

// this function searches whether a given key value i.e. a given building number exists in the red black tree or not

public RBNode search(int bNum)

// this private function is called by the above public ‘search’ function. Returns the RBNode if it is present, else returns null

private RBNode search(RBNode root, int bNum)

// this function searches between a range of building numbers i.e. bNum1 and bNum2

public List<RBNode> searchRange(int bNum1, int bNum2)

// this function searches the range of building numbers and is called by the above public function. Returns a list of RBNodes if there exist nodes in the given range

private void searchRange(RBNode root, List<RBNode> list, int bNum1, int bNum2)

// this rotation is similar to the right rotation of an AVL tree. The rotation is performed to correct the structure of the red black tree after an insert or delete operation

private RBNode rotateRight(RBNode rbn1)

// the left counterpart of the above right rotation function

private RBNode rotateLeft(RBNode rbn1)

// this function performs the insertion of RBNode into the red black tree and also fixes errors in structure, if any

public void insert(RBNode p)

// this function does the insertion in accordance with the binary search tree property

private void doBSTInsertion(RBNode root, RBNode p)

// function to correct the structure of the red black tree

private void correctInsertion(RBNode p)

// this function is run when nodes at two levels need to be swapped with one another

private void moveUp(RBNode a, RBNode b)

// function to delete a RBNode from the red black tree

public boolean delete(RBNode p)

// this function removes the RBNode from the red black tree

public boolean delete(int bNum)

// this function corrects any errors after deletion of node from red black tree

private void correctDeletion(RBNode py)

// this function returns the node with the smallest key value in the red black tree

private RBNode getMinimum(RBNode root)

// function to swap colours of the parent and the grandparent nodes

```
        private void colorChange(RBNode pp, RBNode gp)

}
```

The structure of the main method is as follows:

```
public static void main(String[] args){
        // create a minheap, a rbtree. Initialise global to 0 and line to the first line in the input
        file
        // insert the first building into the minheap and the rbtree
        while (line is not null or root node of minheap is not null){
                if (root of heap is not null){
                        if (total time – executed time of root > 5){
                                // increment global by 5
                                // increment execution time of root node by 5
                        }
                        else{
                                // increment global by total time – executed time of root
                                // increment execution time of the root node by same value
                                // if execution completes on same day as we get Print command
                                // print the building and then do extract min
                                // perform extract min
                        }
                }
                else{
                        // set global to the counter value in line
                }
                while(line is not null and counter in line is less than value of global){
                        if (line is a Print command){
```

```
                // do printing for PrintBuilding(_,_) and PrintBuilding(_)
                accordingly
            }
            else if (line is an Insert command){
                // check if duplicate building is being inserted. If yes, then exit
                program
                // insert the node into the minheap and the rbtree
            }
            // read a new line
        }
    }
        // close the writer object
}
```