COP5536: Advance Data Structure

Project Report

Subham Agrawal
UFID – 7949-7379
subham.agrawal@ufl.edu

Aim – Implementing desired software for Wayne Enterprises to keep track of all buildings under construction in the new city using Red Black Tree and Min Heap implementation.
Also, to be able to output based on the following inputs -

1. **Print (buildingNum)** prints the triplet buildingNume,executed_time,total_time.
2. **Print (buildingNum1, buildingNum2)** prints all triplets bn, executed_tims, total_time for which buildingNum1 <= bn <= buildingNum2.
3. **Insert (buildingNum,total_time)** where buildingNum is different from existing building numbers and executed_time = 0.

Introduction –

A **min-heap** is a binary tree in which the data contained in each node is less than (or equal to) the data in that node's children.
Efficiency of min-heap: For a min-heap with N nodes. It has ⌈log2(N+1)⌉ levels.

1. Insert - Since the insert swaps at most once per level it takes O(log N).
2. Remove - Since the remove also swaps at most once per level, the time complexity is same as insert, O(log N).

A **red-black tree** is a binary search tree in which each node has a color (red or black) associated with it. It has the following 3 properties:

1. The root of the red-black tree is always black.
2. There cannot be two consecutive red nodes in any path from root to an external node.
3. For each node with at least one null child, the number of black nodes on the path from the root to the null child is the same.

Efficiency of RBT:

1. In the worst-case, we end up fixing a double-red situation along the entire path from the added node to the root. So, in the worst-case, the recoloring that is done during insert takes O(log N).
2. Remove takes the same time complexity as insertion, O(log N).

Implementation –
Below are my implementation details for this project, including the java classes used and important function prototypes.

Building.java

```
public class Building {
   int buildingNum;      // unique integer identifier for each building.
   int executedTime;     // total number of days spent so far on this building
   int totalTime;        // total number of days needed to complete the construction
                 // of the building
   . . .
}
```

A pojo class containing all the three building records.
Single building reference is stored in both the Red-Black-Tree and MinHeap node, reducing the space complexity of the program.

MinHeap.java

```
class MinHeapNode {
    Building building;                 //building object reference
    RedBlackTreeNode redBlackTreeNode; // Red-Black-Tree reference

    ...
    public int compareTo(MinHeapNode otherMinHeap) {
        ... ...
        }
}
```

A Min Heap implementation based on executedTime, contains two classes

1. **MinHeapNode**
2. **MinHeap**

MinHeapNode also has a **compareTo** implementation – which compares two MinHeapNode based on the executedTime. It is also used to break ties, comparing buildingNum when executedTime are equal.
MinHeap class is implemented using Arraylist<MinHeapNode>.

Function Prototypes

```
public void insert(MinHeapNode node)
```

Adds an element at the end of the arraylist, recursively compares itself with its parent element. If parent is greater, swaps the element and its parent. Continues the same until parent is smaller than child.

```
public MinHeapNode extractMin()
```

Removes the 0^{th} element (smallest executedTime), replace it with the last element of the arraylist, calls heapify(0). Also, returns the removed Node.

```
private void heapify(int i)
```

Starts at input i, find the smallest element between the current node and its children. If the smallest key is not the current key, then swap it with the smallest, and calls heapify(smallest). Bubble down until the min heap property is maintained.

RedBlackTree.java

```
class RedBlackTreeNode {

    Building building;                    //building object reference
    MinHeapNode minHeapNode;              //min-heap node reference
    int color = RedBlackTree. BLACK_NODE; //node color
    RedBlackTreeNode left = RedBlackTree.nil, right = RedBlackTree.nil, parent =
RedBlackTree.nil;
    ... ...
    public int compareTo(RedBlackTreeNode otherRedBlackTreeNode) {
        ... ...
    }
}
```

A Red Black Tree implementation based on buildingNum, contains two classes

1. **RedBlackTreeNode**
2. **RedBlackTree**

RedBlackTreeNode also has a **compareTo** implementation – which compares two RedBlackTreeNode based on the buildingNum.

Function Prototypes

```
public RedBlackTreeNode findNode(int buildingNum, RedBlackTreeNode node) {
```

Takes two parameters buildingNum (building to find) and node (root) and returns the RedBlackTreeNode with same building number as buildingNum. Finds a node in O(log(n)) time complexity.

```
public void findNode(RedBlackTreeNode node, int buildingFrom, int buildingTo,
ArrayList<RedBlackTreeNode> redBlackTreeNodes) {
```

Takes four parameters buildingFrom < buildingTo (range of building to find) and node (root) and reference of the Arraylist< RedBlackTreeNode>. Updates the arraylist whenever a building is found in the given range. Finds a range of nodes in O(log(n)+S) time, where n is number of active buildings and S is the number of triplets printed.

```
public void insert(RedBlackTreeNode newNode)
```

if root is nil - inserts the node as black root ; else, insert it as a red node, like a binary search tree insertion and calls rBTFixAfterInsert () on the inserted node.

```
void LeftRotation(RedBlackTreeNode nodeToRotate) {
```

Takes a RedBlackTreeNode and rotates the corresponding Red-Black tree to left.

```
void rightRotation(RedBlackTreeNode nodeToRotate) {
```

Takes a RedBlackTreeNode and rotates the corresponding Red-Black tree to right.

```
boolean delete(RedBlackTreeNode nodeToDelete) {
```

Takes a RedBlackTreeNode z, replace it with the minimum element in its right subtree and calls delete() on the replaced node.

```
private void rBTFixAfterInsert(RedBlackTreeNode nodeToFix) {
```

Takes as argument newly inserted RedBlackTreeNode node and evaluate two cases if node's parent is red –

Case 1. If node's parent's sibling is black or null

Case 2. If node's parent's sibling is red

```
void rBTFixAfterDelete(RedBlackTreeNode redBlackTreeNode) {
```

Is called when a black node is deleted from the RBT. And is called on the replaced RBT node x. Recursively fixes until RBT property is maintained in the whole tree.

DEPQ.java

```
public class DEPQ {
    public RedBlackTree redBlackTree;
    public MinHeap minHeap;
}
```

This class is used to instantiate single instances of RBT and MinHeap both simultaneously. And is also used for simultaneous insertion (keeps correspondence between nodes of RBT and MinHeap containing the same building). **Note:** Its naming is just based on the correspondence structure.

```
public void insert(MinHeapNode minHeapNode) {
```

Function is used for the simultaneous insertion of RBT and MinHeap nodes (as mentioned above). Also checks and stops the program in case of a duplicate building number.

ReadFile.java

```
public class ReadFile {
    public static Map<Integer,String> readFile(String fileName) {
        BufferedReader reader;
        . . .
        while (line != null) {
        int eventDay = Integer.parseInt(line.substring(0,
line.indexOf(":")).trim());
        map.put(eventDay,line.substring(line.indexOf(":")+1).trim());
        . . .
}
```

Reads the specified input file and stores the input days and operation in a HashMap<Integer,String>.

WriteFile.java

```
public class WriteFile {
   public static int printPriority = 1;
   public static BufferedWriter out = null;
   . . .
}
```

Used to write the required outputs to a file.

```
public static void writeLineWithNewLine(String output,int p) {
```

Writes to the output file with a new line (if priority is set as 1).

```
public static void writeLine(String output,int p) {
```

Writes to the output file without a new line (if priority is set as 1).

```
public static void close() throws IOException {
```

Closes the file.

risingCity.java

```
public class risingCity {
    public static int globalTime = 0;
    public static Building buildingSelected;

    . . .
}
```

It is the entering point of the program. Maintains the state of two variables, globalTime and the buildingSelected (the current building being worked upon).

```
public static void run(Map<Integer,String> lines, DEPQ doubleEndedPQ) {
```

Firstly, inserts the buildings available before the work starts, i.e., building on day 0.

Below is my algorithm:

While (minheap is not empty)
    globalTime = globalTime +1
    //if insert found & no building is selected, then inserts it in 1st half of the day
    if (there is an insert on day = globalTime)
        insert into MinHeap and RBT
    buildingSelected = MinHeap.extractMin()
    workStartedOn = globalTime
    continueFor = Min (5, total_time-executed_time)
    while (globalTime < workStartedOn + continueFor)
        buildingSelected.executedTime = buildingSelected.executedTime + 1
        if (there is an operation on day = globalTime)
            do required insert/print //do required insert/print on the 2nd half
        globalTime = globalTime +1
    If (buildingSelected.totalTime -buildingSelected. executedTime)
        Insert back the same building
    Else
        Remove the selected tree from RBT
    //If the input file contains any discontinued day, detect any discontinued day and increment the global time to that day & do the desired operation for that day.(where discontinued day is when no work is being done)

```
public static void insert(String line, DEPQ doubleEndedPQ) {
```

Creates a new Building() object, inserts into Red-Black-Tree and Min Heap simultaneously with executedTime = 0.

```
public static void print(String line, DEPQ doubleEndedPQ) {
```

Calls the findNode(int buildingNum, RedBlackTreeNode node) implantation of RedBlackTree class to get the building details of the corresponding building number and prints it to the output file. If there is no building with the associated building number prints (0,0,0).

```
private static void printFromTo(String line, DEPQ doubleEndedPQ) {
```

Like the print function above, prints all the buildingNum in increasing order between the specified range. Calls the RedBlackTreeNode node, int buildingFrom, int buildingTo, ArrayList<RedBlackTreeNode> redBlackTreeNodes) implantation of RedBlackTree class. If no building is found prints (0,0,0).

```
public static void main(String[] args) throws IOException {
```

Creates an instance of DEPQ, and calls the risingCity.run() method.

References

1. https://www.cise.ufl.edu/~sahni/cop5536/presentations.htm
2. http://pages.cs.wisc.edu/~skrentny/cs367-common/readings/Red-Black-Trees/
3. https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap13c.pdf
4. http://www.cs.toronto.edu/~wgeorge/csc265/2013/09/26/tutorial-3-red-black-tree-deletion.html