Advanced Data Structures project report(COP5536)

Introduction

A software has been developed for Wayne Enterprises for helping them develop a new city to keep track of all buildings under construction. Each building has following fields:

1.buildingNum: unique integer identifier for each building.
2. executed_time: total number of days spent so far on this building.
3. total_time: the total number of days needed to complete the construction of the building.

To implement this, Java language has been used.

Structure of Solution:

There are five java files which comprise the whole functionality of this solution

Data Structures related files:

1.MinHeapNodeImpl

This class is a node implementation of min heap with the following fields
Here rbtNode is the pointer to corresponding RBT.

```
public int buildingNo;
public int totalTime;
public int executedTime;
public RbtNodeImpl rbtNode;
```

Here execution time is the priority on which we construct the heap however building num is used for tying case. Getters and setters for accessing the variables are implemented accessing the variables directly is not a good practice. Constructors is used to set node while creation.

2.MinHeap.

This contains implementation of min heap using Array of size 2000 named minArr.
Insert(MinHeapNodeImpl node) is used to insert the node into the minArr array and we fix the min heap property using **fixAfterInsert(int index, MinHeapNodeImpl array[])** which will return the fixed min heap array. Check here recursively if any node executed time value is less than its parent then swap it else if equal then check for the minimum building number among the two and rearrange the heap accordingly.

The **extractMin()** function removes the minimum execution time value node from the hap which is indexed at 0^{th} element of the array. After removing we run min heapify operation using **heapify(MinHeapNodeImpl arr[],int index)** function which arranges the element according to their execution time, which if ties is broken by building number value. We run this for half of the

elements which ensures that all elements are in their place. The **isEmpty()** function checks id the heap is empty or not.

3.RbtNodeImpl

This class is a node implementation of red black tree with the following fields
Here the heapNode is pointer to corresponding minheap.

```
    private int buildingNo;
    private int totalTime;
    private int executedTime;
    private RbtNodeImpl left = nil;
    private RbtNodeImpl right = nil;
    private RbtNodeImpl parent = nil;
    private String colour;
    private MinHeapNodeImpl heapNode;
    public static final RbtNodeImpl defaultNode = new
RbtNodeImpl(Integer.MIN_VALUE,"Black");
```

Here the key value is buildingNo. which should be unique .Getters and setters for accessing the variables are implemented accessing the variables directly is not a good practice. Constructors is used to set buildingNo and colour for node while creation.

The defaultNode is the an object for denoting default node equivalent to nil. We use a string variable named colour to store colour of node by having values “RED” or “BLACK”

```
  public RbtNodeImpl(int buildingNo, String colour){
        this.setBuildingNo(buildingNo);
        this.setColour(colour);
    }
```

4.Rbt.java

Constructor is used to instantiate Rbt root node and set it to defaultNode.

```
public Rbt(){
        defaultNode = RbtNodeImpl.defaultNode;
        root = defaultNode;
        root.setRight(defaultNode);
        root.setLeft(defaultNode);
    }
```

Search(int buildingNum) is used as an interface from risingCity.java to search the node with following buildingNum. If not found will return null. It use search(RbtNodeImpl root, int buildingNum) method internally which runs a binary search on the tree for the node.

SearchInRange(int buildingNum1. int buildingNum2) is used as an interface from risingCity.java to search the node with following buildingNum range from buildingNum1 to buildingNum2

RightRotate Rbt(NodeImpl node) and **leftRotate (RbtNodeImpl root)** methods are used for performing right and left rotations respectively on the nodes of tree while insertion or deletion and updates all pointers accordingly

insert(int buildingNum) is an interface for insert operation which in turn calls **insertNode(RbtNodeImpl node).** This function also throws error id we try to insert a duplicate node with following building number.

insertOpertaion(RbtNodeImpl root, RbtNodeImpl parent) compares the parent and node and inserts the element wherever it falls off the tree after search and add it as default red node which me lead to violations like double red nodes in a row which is corrected by **fixAfterInsertion(RbtNodeImpl child).**

The **delete(RbtNodeImpl node)** is an interface for delete node operation which is done in **delete(int buildingNum)** which searches for the node with following building number and removes it from the tree by getting its replacement and performing fixup operation to set the levels of the node properly. The violations are handled by **fixAfterDeleteion(RbtNodeImpl node)** where rotations are performed so that it maintains the property of ted black tree

Driver files:

risingCity.java

This is the entry point of the program where arguments are taken from command line which specifies the input file name. FileWriter and FileReader has been used to read and write input/output in .txt files respectively. BufferredReader is used to read the input line by line.
The input line has been split for getting parameters namely time, buildingNum and time required and also store the operation to be performed. For insert operation call **insertUtil(int buildingNum, Int TimeReq)**

insertUtil(int buildingNum,Int TimeReq):
Here new nodes are created for both RBT and MinHeap and set pointers to each other. This also throws any error while insertion (duplicate value error).

resetCounter()
CurrentBuilding value is reset here so that a new building can be initialized as currentBuilding.

Scheduler()

This is the main event runner in the program which executes work on buildings run in recursion. If building is present on which work has to be done, add 5 to its current working time and check if it is less than its work finish time(current working tim+5). If it is less then continue, else if it is equal to current building finish time write the output and delete the element from the data structures. If it elapses 5 counters then insert it back in the heap continue it for next job.

printBuilding(int buildingNum)
This is used to print the output in output file when there is buildingNum as input .

printBuilding(int buildingNum1, int buildingNum2)
This is used to print the output in output file when there is buildingNum range as input. The String containing the output is received from searchInRange (int b1,int b2). Since search takes logarithmic time this operation takes logarithmic time only.

finishWork()
This method is used to complete all the works still present in the data structures when there are no operations being performed on them.

Name:Himavanth Boddu
UFID:32451847
Email:himavantboddu@ufl.edu