Advanced Data Structures (COP 5536)

Fall 2019

Project Report

Name : Mohit Kalra
UFID : 13906151
Email ID : mohit.kalra@ufl.edu

Problem Statement Understanding

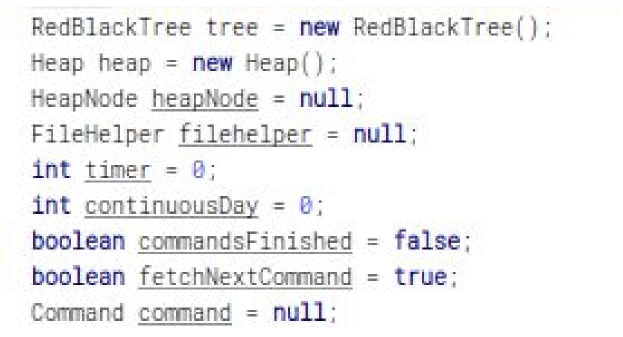
The problem statement revolves around the construction of a city. At maximum a building can only be worked on for 5 consecutive days. We'll be working on buildings on the basis of time which was given to them. We'll consider the building for which least work was done as the next potential building to work on. We'll utilize Min Heap Data Structure for this. In case there's a tie in the total time worked upon a building, we'll utilize the building number of the building to break the tie. Building with smaller number will be worked upon first. For keeping the search of building information fast we'll be utilizing a Red Black Tree supporting search in O(logn) time and ensuring that the search tree is balanced at all times. We also have a print in range functionality which will print all buildings in a range of numbers provided in the input.

The java application consists of the following classes:

1. risingCity.java - Main driver class of the project. Utilizes other classes for running the aforementioned problem statement logic.
2. Building.java - Defines attributes and methods for a building. A building object is passed as a reference to a node of red black tree.
3. Command.java - Helps define a command and it's characteristics like the type of command - insert, print, print in range etc.
4. FileHelper.java - Helps tackle all file operations like reading commands from input file and writing to output files etc.
5. HeapNode.java - Heap Node defines nodes for the heap tree. Heap Node holds reference to corresponding red black node in the red black tree.
6. TreeNode.java - TreeNode defines a red black tree node. TreeNode holds reference to a building object which holds the building information, inturn.
7. RedBlackTree.java - RedBlackTree class perform red black insertion and deletion. It also has other methods which help search a building in O(logn) time.
8. Heap.java - Class defining behaviors of a min heap. Performs push, pop operations on heap nodes with respect to a custom comparison function.

risingCity.java

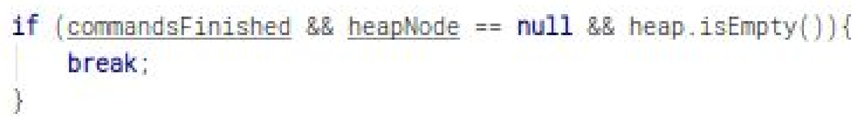
- Class defines a heap, a red black tree, a heap node representing a building we'll be working on.
- We also have a FileHelper object to help us with the file operations
- Read a new command from the file returns a command object
- We parse the command object when it's time for its execution



```java
RedBlackTree tree = new RedBlackTree();
Heap heap = new Heap();
HeapNode heapNode = null;
FileHelper filehelper = null;
int timer = 0;
int continuousDay = 0;
boolean commandsFinished = false;
boolean fetchNextCommand = true;
Command command = null;
```

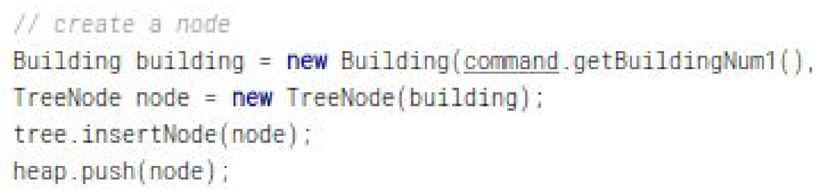
    -

- We'll keep reading until the commands have finished or there is no building left to work on.
- We keep track of the timer using the timer variable.
- We keep track of the continuous working day variables using the continuousDay variable
- The following condition breaks the driving program loop and represents completion of the city.



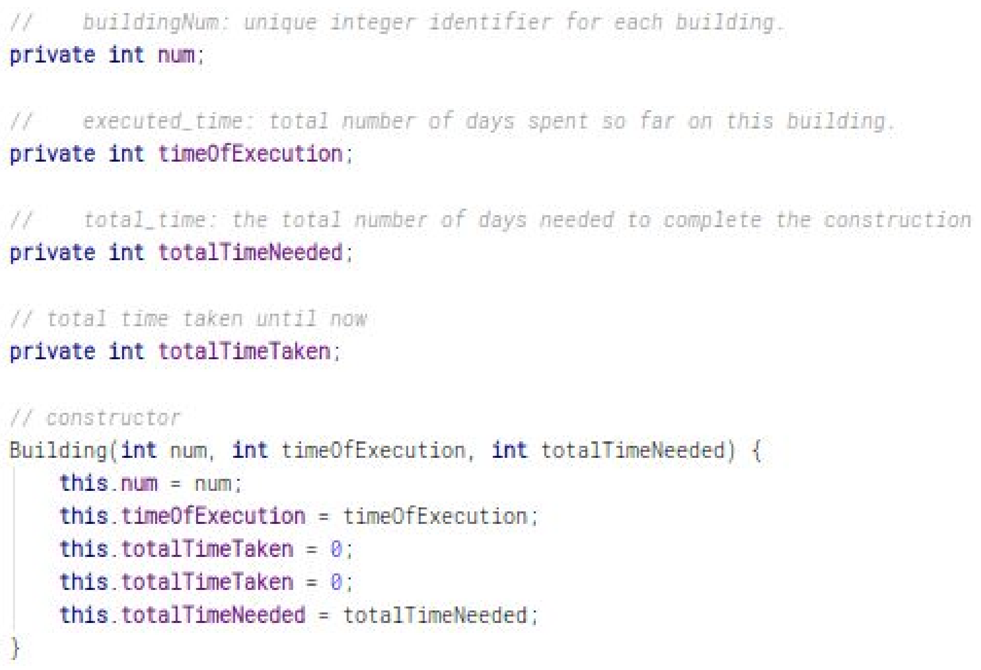
```java
if (commandsFinished && heapNode == null && heap.isEmpty()){
    break;
}
```

    - For insert command, we create a new building object given the command variables
    - We create a new tree node and pass building's reference into it
    - We pass the red black tree node reference in the heap object.
    - For printing we call the inorder search function defined in the RedBlack Tree class.
    - Insertion of a new building into the city ecosystem



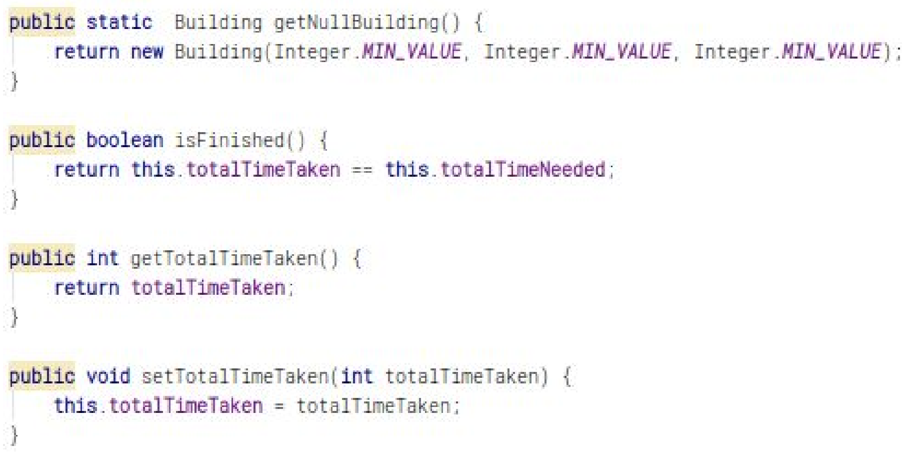
```java
// create a node
Building building = new Building(command.getBuildingNum1(),
TreeNode node = new TreeNode(building);
tree.insertNode(node);
heap.push(node);
```

Building.java



```java
//    buildingNum: unique integer identifier for each building.
private int num;

//    executed_time: total number of days spent so far on this building.
private int timeOfExecution;

//    total_time: the total number of days needed to complete the construction
private int totalTimeNeeded;

// total time taken until now
private int totalTimeTaken;

// constructor
Building(int num, int timeOfExecution, int totalTimeNeeded) {
    this.num = num;
    this.timeOfExecution = timeOfExecution;
    this.totalTimeTaken = 0;
    this.totalTimeTaken = 0;
    this.totalTimeNeeded = totalTimeNeeded;
}
```

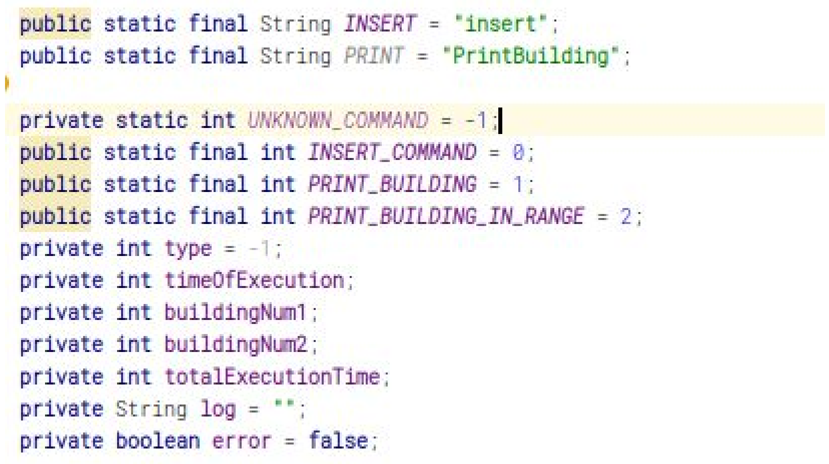
- All the attributes related to a building are packed into this class.
- num - Building Number
- timeOfExecution - time associated with command using the building
- totalTimeNeeded - number of days needed to complete the building
- totalTimeTake - number of days already given to the building’s construction



```java
public static  Building getNullBuilding() {
    return new Building(Integer.MIN_VALUE, Integer.MIN_VALUE, Integer.MIN_VALUE);
}

public boolean isFinished() {
    return this.totalTimeTaken == this.totalTimeNeeded;
}

public int getTotalTimeTaken() {
    return totalTimeTaken;
}

public void setTotalTimeTaken(int totalTimeTaken) {
    this.totalTimeTaken = totalTimeTaken;
}
```

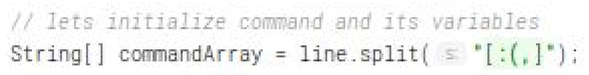
- getNullBuilding - Helps defines a null node of a red black tree.
- isFinished - Tells whether a building has completed its construction or not
- getTotalTimeTaken - Getter for number of working days given to the building until now.
- setTotalTimeTaken - Setter function for working days of a given building

Command.java



```java
public static final String INSERT = "insert";
public static final String PRINT = "PrintBuilding";

private static int UNKNOWN_COMMAND = -1;
public static final int INSERT_COMMAND = 0;
public static final int PRINT_BUILDING = 1;
public static final int PRINT_BUILDING_IN_RANGE = 2;
private int type = -1;
private int timeOfExecution;
private int buildingNum1;
private int buildingNum2;
private int totalExecutionTime;
private String log = "";
private boolean error = false;
```

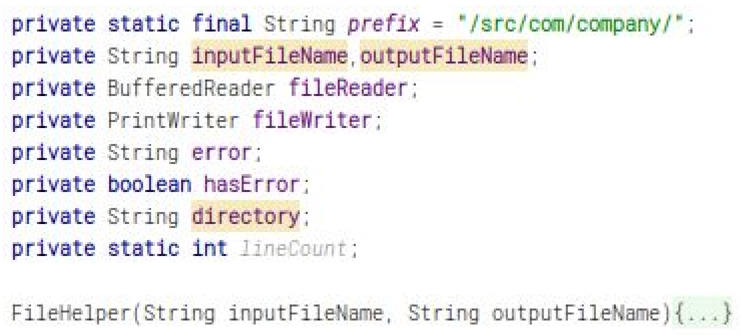
- INSERT and PRINT define constant strings to compare to during reading in of commands.
- we have constants for each of the commands - 0 - INSERT, 1 PRINT BUILDING, 2 - PRINT ALL BUILDINGS in a given range.
- buildingNum1 - represents num of a building in case of search of only that building
- buildingNum1, buildingNum2 - defines range in case we need to print nodes in a given range.
- Log and error are used for developer output logging.



```java
// lets initialize command and its variables
String[] commandArray = line.split( s: "[:(,]");
```

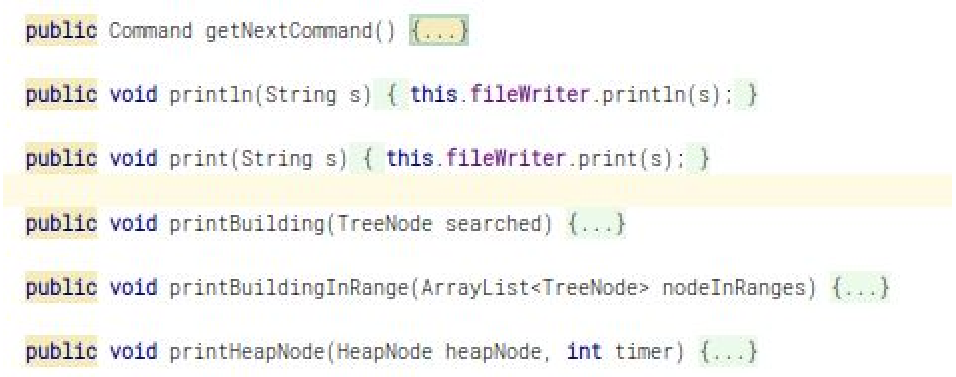
- The above split statement splits the command on the basis of tokens provided in the regex. Depending on the type of command we'll then have an array of size 2,3 and 4 in the array.
- We'll use the array size to define the commands accordingly.

FileHelper.java



```java
private static final String prefix = "/src/com/company/";
private String inputFileName,outputFileName;
private BufferedReader fileReader;
private PrintWriter fileWriter;
private String error;
private boolean hasError;
private String directory;
private static int lineCount;

FileHelper(String inputFileName, String outputFileName){...}
```

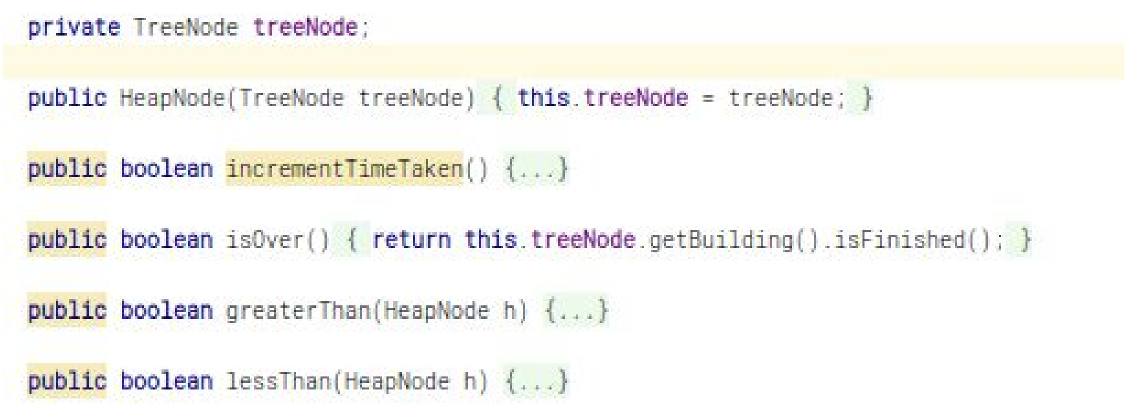
- inputFileName and OutputFilename hold input and output file names respectively.
- fileReader and fileWriter are objects for reading from /writing into a file.
- Directory holds the value of the current working directory



```
public Command getNextCommand() {...}

public void println(String s) { this.fileWriter.println(s); }

public void print(String s) { this.fileWriter.print(s); }

public void printBuilding(TreeNode searched) {...}

public void printBuildingInRange(ArrayList<TreeNode> nodeInRanges) {...}

public void printHeapNode(HeapNode heapNode, int timer) {...}
```

- getNextCommand() is utilized by the driver program to fetch the next command in the file.
- print() and println()- print in same / next line into the output file
- printBuilding prints the searched building for the command
- printBuildingInRange() - prints all the buildings in range of given min and max values
- printHeapNode() - prints output for building which completed its construction.

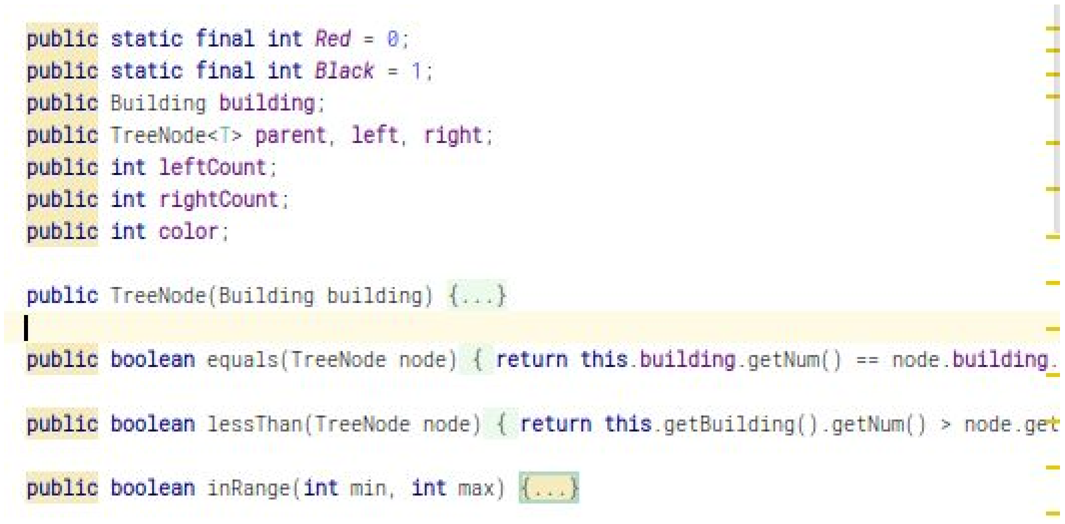
HeapNode.java



```
private TreeNode treeNode;

public HeapNode(TreeNode treeNode) { this.treeNode = treeNode; }

public boolean incrementTimeTaken() {...}

public boolean isOver() { return this.treeNode.getBuilding().isFinished(); }

public boolean greaterThan(HeapNode h) {...}

public boolean lessThan(HeapNode h) {...}
```

- HeapNode holds a reference to a corresponding treeNode in Red Black Tree.
- incrementTimeTaken increases the number of days given to building by 1
- isOver tells whether the construction of the building the heapNode holds is over or not
- greaterThan and lessThan are special comparators for comparing two heap nodes
- The heap is minified on a combined comparison of number of days given to a building and the building number.
    - If the number of days given to a building are equal, then we consider the building with less building number as the min element for heapification.

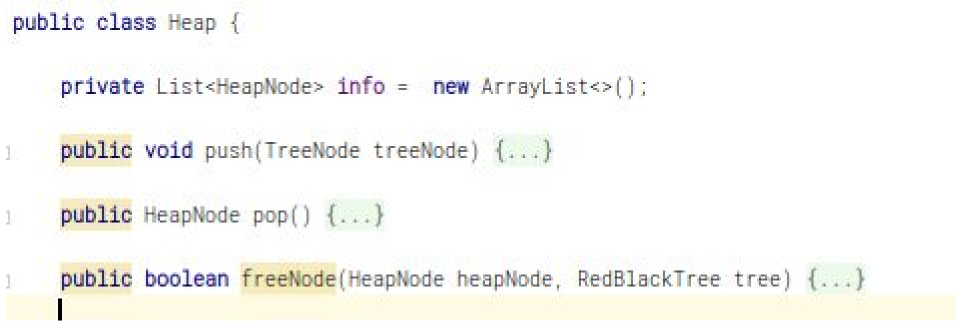
TreeNode.java



```
public static final int Red = 0;
public static final int Black = 1;
public Building building;
public TreeNode<T> parent, left, right;
public int leftCount;
public int rightCount;
public int color;

public TreeNode(Building building) {...}

public boolean equals(TreeNode node) { return this.building.getNum() == node.building.

public boolean lessThan(TreeNode node) { return this.getBuilding().getNum() > node.get

public boolean inRange(int min, int max) {...}
```

- TreeNode class defines the behavior of a red black node
- Red, Black are color constants defined for a TreeNode
- A TreeNode holds reference to a building
- Color attribute holds the color of the TreeNode

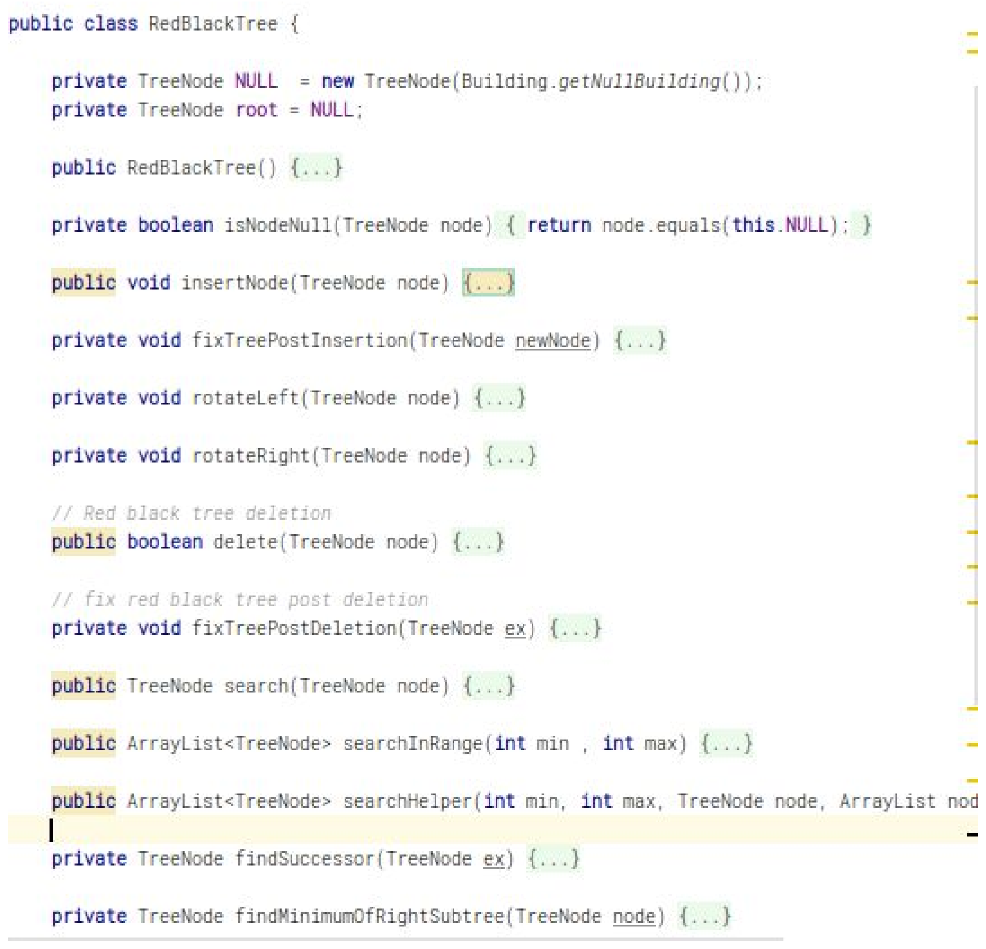
Heap.java



```
public class Heap {

    private List<HeapNode> info =  new ArrayList<>();

    public void push(TreeNode treeNode) {...}

    public HeapNode pop() {...}

    public boolean freeNode(HeapNode heapNode, RedBlackTree tree) {...}
```

- Heap class defines the behavior of min heap
- We maintain an arraylist of heap nodes called info, which holds all the heap nodes we'll use for heapification.
- Push and Pop methods are typical push and pop operations of a heap but utilized the custom comparison of a heap node to heapify the data structure.
- **FreeNode** is an important function which implicitly calls the delete node of a red black tree of the treeNode - reference of which is held by the given heap node. This function is called when the work of a building is finished.

RedBlackTree.java



```java
public class RedBlackTree {

    private TreeNode NULL  = new TreeNode(Building.getNullBuilding());
    private TreeNode root = NULL;

    public RedBlackTree() {...}

    private boolean isNodeNull(TreeNode node) { return node.equals(this.NULL); }

    public void insertNode(TreeNode node) {...}

    private void fixTreePostInsertion(TreeNode newNode) {...}

    private void rotateLeft(TreeNode node) {...}

    private void rotateRight(TreeNode node) {...}

    // Red black tree deletion
    public boolean delete(TreeNode node) {...}

    // fix red black tree post deletion
    private void fixTreePostDeletion(TreeNode ex) {...}

    public TreeNode search(TreeNode node) {...}

    public ArrayList<TreeNode> searchInRange(int min , int max) {...}

    public ArrayList<TreeNode> searchHelper(int min, int max, TreeNode node, ArrayList nod

    private TreeNode findSuccessor(TreeNode ex) {...}

    private TreeNode findMinimumOfRightSubtree(TreeNode node) {...}
```

- NULL TreeNode is reference to a null node of the tree. It's to be noted that it's initialized by a null reference of the Building.
- isNodeNull method helps us check whether a given red black tree node is null or not.
- insertNode inserts a new red black node into the tree as we would have done in a BST. For rules we might have violated, we follow it up with fixTreePostInsertion method. The tree rotations are done using the rotateLeft and rotateRight methods defined in the class.
- fixTreePostInsertion - All the red black tree properties violated by the insertion of a new node is fixed by this method. We cover each of the 6 cases of insertion of a Red black Tree Node
- Delete - We pass a node representing a node to be deleted to this method. Depending on whether the node deleted has degree 0,1,2 - appropriate action is defined for the same. This is followed by fixTreePostDeletion() to fix rules that would have been violated during the deletion. LeftRotation, RightRotation around a node is provided by rotateLeft and rotateRight methods.
- fixTreePostDeletion - Helps tree fix violation of Red Black Tree. I've done this using 8 cases of Deletion as per standard CLRS textbook.

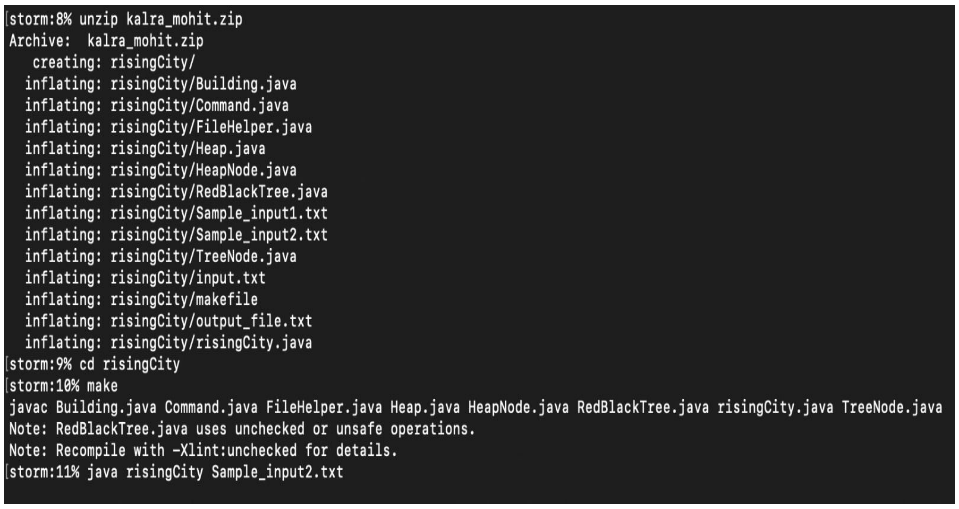
- findSuccessor - Helps find inorder successor as if in a BST. If left child is null, we search for the successor in the right subtree which is done using the findMinimumOfRightSubtree method.

Conclusion

The above classes define all the operations of that problem statement wanted for implementation. Min Heap ensured that we always process the building which is less worked on. We could know about such a building in O(logn) time by a normal pop operation of a heap. We could also search for the information of building with a given number or a range in O(logn) / O(n) time respectively. The output was thus, written to the output_file.txt.

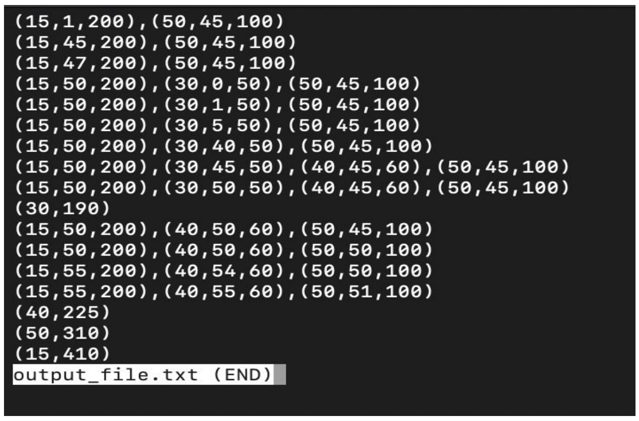
Please refer to the following screenshots from the thunderbird server which was used to test this program :

Sample input:



```
storm:8% unzip kalra_mohit.zip
Archive:  kalra_mohit.zip
   creating: risingCity/
  inflating: risingCity/Building.java
  inflating: risingCity/Command.java
  inflating: risingCity/FileHelper.java
  inflating: risingCity/Heap.java
  inflating: risingCity/HeapNode.java
  inflating: risingCity/RedBlackTree.java
  inflating: risingCity/Sample_input1.txt
  inflating: risingCity/Sample_input2.txt
  inflating: risingCity/TreeNode.java
  inflating: risingCity/input.txt
  inflating: risingCity/makefile
  inflating: risingCity/output_file.txt
  inflating: risingCity/risingCity.java
storm:9% cd risingCity
storm:10% make
javac Building.java Command.java FileHelper.java Heap.java HeapNode.java RedBlackTree.java risingCity.java TreeNode.java
Note: RedBlackTree.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
storm:11% java risingCity Sample_input2.txt
```

Sample Output



```
(15,1,200),(50,45,100)
(15,45,200),(50,45,100)
(15,47,200),(50,45,100)
(15,50,200),(30,0,50),(50,45,100)
(15,50,200),(30,1,50),(50,45,100)
(15,50,200),(30,5,50),(50,45,100)
(15,50,200),(30,40,50),(50,45,100)
(15,50,200),(30,45,50),(40,45,60),(50,45,100)
(15,50,200),(30,50,50),(40,45,60),(50,45,100)
(30,190)
(15,50,200),(40,50,60),(50,45,100)
(15,50,200),(40,50,60),(50,50,100)
(15,55,200),(40,54,60),(50,50,100)
(15,55,200),(40,55,60),(50,51,100)
(40,225)
(50,310)
(15,410)
output_file.txt (END)
```