

**Advanced Data Structures  
(COP 5536)  
Fall 2019  
Programming Project Report**

Vishisth Chaturvedi  
UFID 1787-6619  
[vchaturvedi@ufl.edu](mailto:vchaturvedi@ufl.edu)

# PROJECT DESCRIPTION

The project is emulating the workings of Wayne Enterprises, a construction company. The project handles an object Building. Running the building process according to the required constraints.

It consists of two main parts. The Min Heap and The Red Black Tree. These two are used in combination to perform the required three operations:

- A. Print Operation
- B. PrintRange Operation
- C. Insert Operation

The program reads the input file from the command prompt and then generates an output file named "output\_file.text".

# STRUCTURE OF THE PROJECT AND FUNCTION DESCRIPTION

The implementation is based on five classes which are:

1. risingCity.java
2. HNode.java
3. MinHeap.java
4. RBTree
5. RBNode

## Classes Description:

Below I have mentioned all the classes and some of their important functions:-

### 1. risingCity.java

This is the main driver class. It emulates the working of the project. It uses the MinHeap and RedBlackTree to store the present buildings. It keeps track of the global time.

#### Functions

- A. **private void run():**- This is the main coordinator function. It takes the input commands. Works in cycles of 1 global unit time. It gives preference to reading the input commands over the current process execution. It calls the builder function to work on the building. When it detects input command it calls the insertBuilding function. On detecting print command it runs both print and print in range command.
- B. **private void builder():**- This the command which works on the building. It checks if the work on the building is complete or it has been working for 5 days.
- C. **private void printBuilding(String[] values):**- It performs both the print and print in Range command.
- D. **private void insertBuilding(String[] values):**- It inserts the building. It does so by inserting the building into both the heap and the tree.

### 2. HNode.java

This is the basic node class. It is the node which is the basic structure of each node in the heap.

#### Member variables

- A. **public int executedTime** - The key in this heap is the executed time.
- B. **public RBNode rbNode** - Each node has a reference to the equivalent RBNode;

### 3. MinHeap.java

#### Functions

- A. **public int size()** - It return the size of the heap.
- B. **public boolean isEmpty()** - Return if the heap is empty.
- C. **public void add()** - Adds a new HNode to the heap.
- D. **public void upHeapify()** - Heapify function used to maintain the heap property.
- E. **public HNode remove()** - Remove function.
- F. **public void swap()** - Swaps two nodes in the heap.
- G. **public boolean isLarger()** - Returns the larger of the two nodes.

### 4. RBTree

#### Functions

- A. **public void insert()** - Inserts a new node into the red black tree. It then uses the fixup function to maintain the Red Black Properties.
- B. **public boolean delete()** - Deletes the node with the given key. Then uses the fixup function to correct the issue created due to the delete. Does this in a bottom to up manner.
- C. **public RBNode search()** - Looks for the node with the given key.
- D. **public List<RBNode> searchInRange()** - Looks for the nodes which are in the given range.
- E. **public void rightRotate()** - Basic rotate function similar to the AVL tree.
- F. **public void leftRotate()** - Left equivalent of the above function.

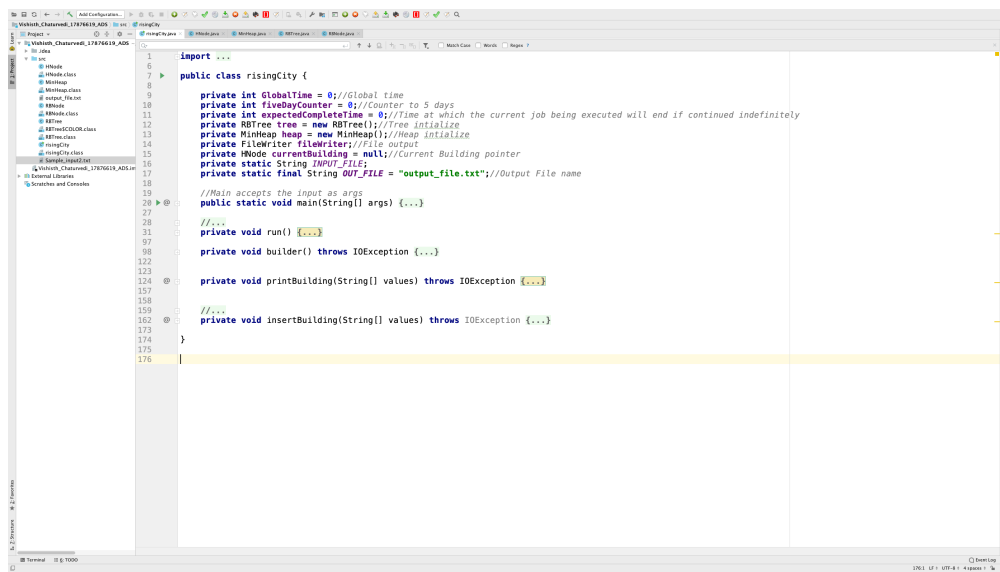
### 5. RBNode

This is the basic node structure of the RB Tree.

#### Member variables

- A. **int Building\_ID** - The unique id of each building.
- B. **RBNode left** - Points to the left child of the node.
- C. **RBNode right** - Points to the right child of the node.
- D. **RBNode parent** - Points to the parent of the node.
- E. **HNode heapnode** - Reference to the equivalent HNode in the heap.

# SCREENSHOTS



```
import java.util.*;

public class risingCity {

    private int GlobalTime = 0; //Global time
    private int fiveDayCounter = 0; //Counter to 5 days
    private int expectedCompleteTime = 0; //Time at which the current job being executed will end if continued indefinitely
    private RBTree tree = new RBTree(); //Tree initialize
    private MinHeap heap = new MinHeap(); //Heap initialize
    private FileWriter fileWriter; //File output
    private int currentBuilding = null; //Current Building pointer
    private static final String INPUT_FILE = "input_file.txt"; //Input File name
    private static final String OUTPUT_FILE = "output_file.txt"; //Output File name

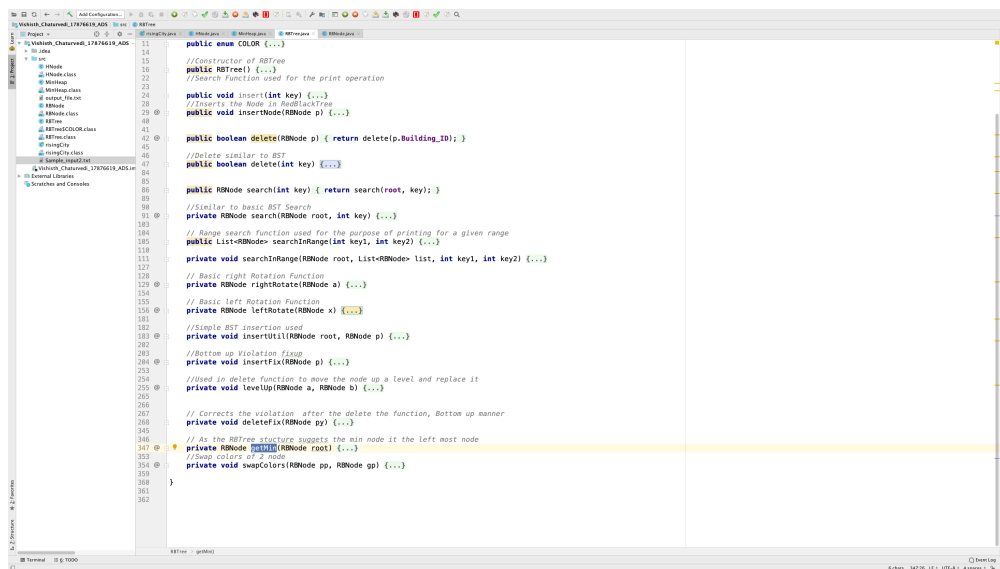
    //Main accepts the input as args
    public static void main(String[] args) { ... }

    //...
    private void run() { ... }
    private void builder() throws IOException { ... }

    private void printBuilding(String[] values) throws IOException { ... }

    //...
    private void insertBuilding(String[] values) throws IOException { ... }
}
```

risingCity.java



```
public enum COLOR { ... }

//Constructor of RBTree
public RBTree() { ... }

//Search Function used for the print operation
public void insert(int key) { ... }
public void insertNode(RBNode p) { ... }

public boolean delete(RBNode p) { return delete(p.Building_ID); }
public boolean delete(int key) { ... }

public RBNode search(int key) { return search(root, key); }

//Similar to basic BST Search
private RBNode search(RBNode root, int key) { ... }

//Range search function used for the purpose of printing for a given range
public List<RBNode> searchRange(int key1, int key2) { ... }

private void searchRange(RBNode root, List<RBNode> list, int key1, int key2) { ... }

// Basic right Rotation Function
private RBNode rightRotate(RBNode a) { ... }

// Basic left Rotation Function
private RBNode leftRotate(RBNode a) { ... }

//Simple BST insertion used
private void insertUtil(RBNode root, RBNode p) { ... }

//Bottom up Violation fixup
private void insertFix(RBNode p) { ... }

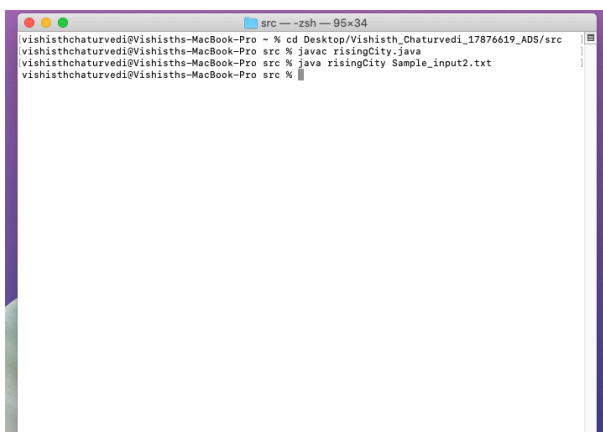
//Used in delete function to move the node up a level and replace it
private void levelUp(RBNode a, RBNode b) { ... }

// Corrects the violation after the delete function, Bottom up manner
private void deleteFix(RBNode p) { ... }

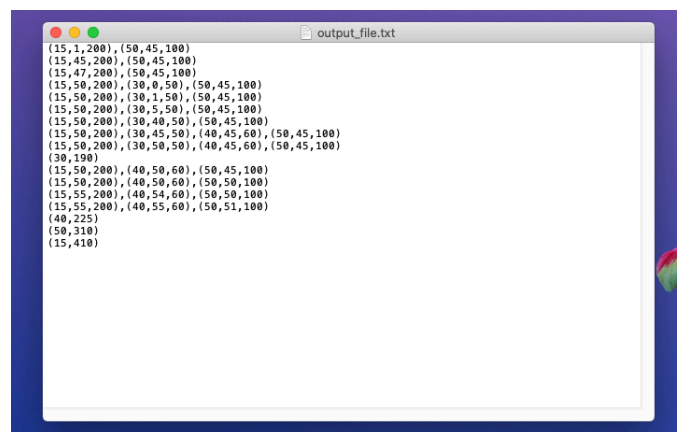
// As the RBTree structure supports the min node if the left most node
private RBNode minNode(RBNode root) { ... }

//Swap colors of 2 nodes
private void swapColors(RBNode pp, RBNode gp) { ... }
}
```

RBTree.java



```
src ~ zsh — 95x34
vishishchaturvedi@Vishishths-MacBook-Pro ~ % cd Desktop/Vishishth_Chaturvedi_17876619_ADS/src
vishishchaturvedi@Vishishths-MacBook-Pro src % javac risingCity.java
vishishchaturvedi@Vishishths-MacBook-Pro src % java risingCity Sample_input2.txt
vishishchaturvedi@Vishishths-MacBook-Pro src %
```



```
output_file.txt
(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)
(30,100)
(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)
```

```
import java.util.ArrayList;

//
// Min Heap Implementation using an ArrayList
//

public class MinHeap {

    private ArrayList<HNode> data;

    private boolean isMin; // Could be used as Max Heap as well

    // Constructors
    public MinHeap() {...}
    public MinHeap(ArrayList<HNode> arr) { this.data = arr; }
    // Constructors
    public MinHeap(boolean isMin) {...}
    // Constructors
    public MinHeap(HNode[] items, boolean isMin) {...}

    // Return the size of the heap
    public int size() { return this.data.size(); }
    // Return if the heap is empty
    public boolean isEmpty() { return this.size() == 0; }

    // Adds a new HNode
    public void add(HNode item) {...}
    // Heapify function used to maintain the heap property
    public void upHeapify(int ci) {...}
    // Remove function
    public HNode remove() {...}
    // Works to maintain the heap condition
    private void downHeapify(int pi) {...}
    // Swaps the HNode data
    private void swap(int i, int j) {...}
    // ...
    private boolean isLarger(int i, int j) {...}
}

// End of MinHeap.java
```

## MinHeap.java

```
// Basic Node Implementation for the heap
public class HNode {

    // Executed time
    public int executedTime;
    // Set in the RNode
    public RNode rNode;

    // Constructor
    public HNode(int executedTime) {...}

    // Override
    public String toString() { return Integer.toString(executedTime); }
}

// End of HNode.java
```

## HNode.java

```
public class RNode {

    public static final RNode nil = new RNode(-Integer.MIN_VALUE, RNode.COLOR.BLACK); // We always take the new added node as Black

    public int Building_ID; // Building_ID
    public int totalTime; // Total executed time of current job
    public RNode left = nil;
    public RNode right = nil;
    public RNode parent = nil;
    public RNode color; // RED/Black color
    public HNode heapNode; // Object reference to heapNode in MinHeap

    // Constructors
    public RNode(int building_number, HNode heapNode) {...}
    public RNode(int building_number) {...}
    public RNode(int building_number, RNode color) {...}

    // Override
    public String toString() { return "Key:"+this.Building_ID +",Color:"+this.color.name(); }
}

// End of RNode.java
```

# CONCLUSION

All the three required functions have been implemented accurately . All of them run in the required time complexities.

1. Insert =  $O(\log n)$
2. Print(key) =  $O(\log n)$
3. Print(key1, key2) =  $O(\log(n)+S)$

The insert function makes use of both the Min Heap implementation and the Red Black Tree implementation. Both of the Print Functions make use of the Red Black Tree.