I. Problem Statement

Wayne Enterprises is developing a new city. They are constructing many buildings and plan to use software to keep track of all buildings under construction in this new city. A building record has the following fields:

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

The needed operations are:
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.

In order to complete the given task, you must use a min-heap and a Red-Black Tree (RBT). You must write your own code the min heap and RBT. Also, you may assume that the number of active buildings will not exceed 2000.A min heap should be used to store (buildingNums,executed_time,total_time) triplets ordered by executed_time. You will need a suitable mechanism to handle duplicate executed_times in your min heap. An RBT should be used store (buildingNums,executed_time,total_time) triplets ordered by buildingNum. You are required to maintain pointers between corresponding nodes in the min-heap and RBT.

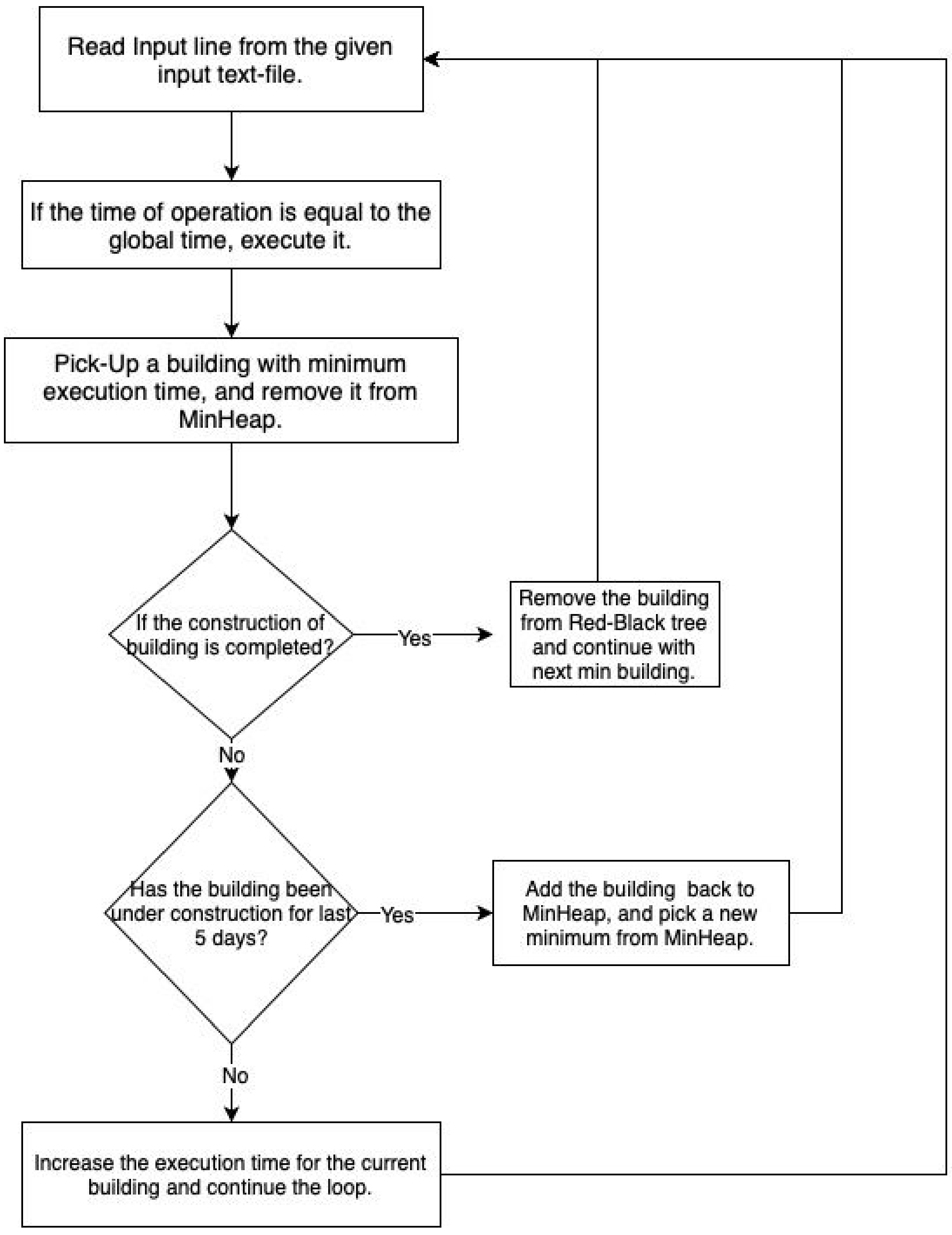
II. Instructions for Execution

- ❏ Unzip the folder Avula_VenkataGowtham.zip.
- ❏ Navigate into the unzipped folder using the ‘cd’ command in the terminal.
- ❏ Execute the ’make’ command in the terminal to compile all the java classes needed for the project and generate the required .class files.
- ❏ Now, run the project using the command:

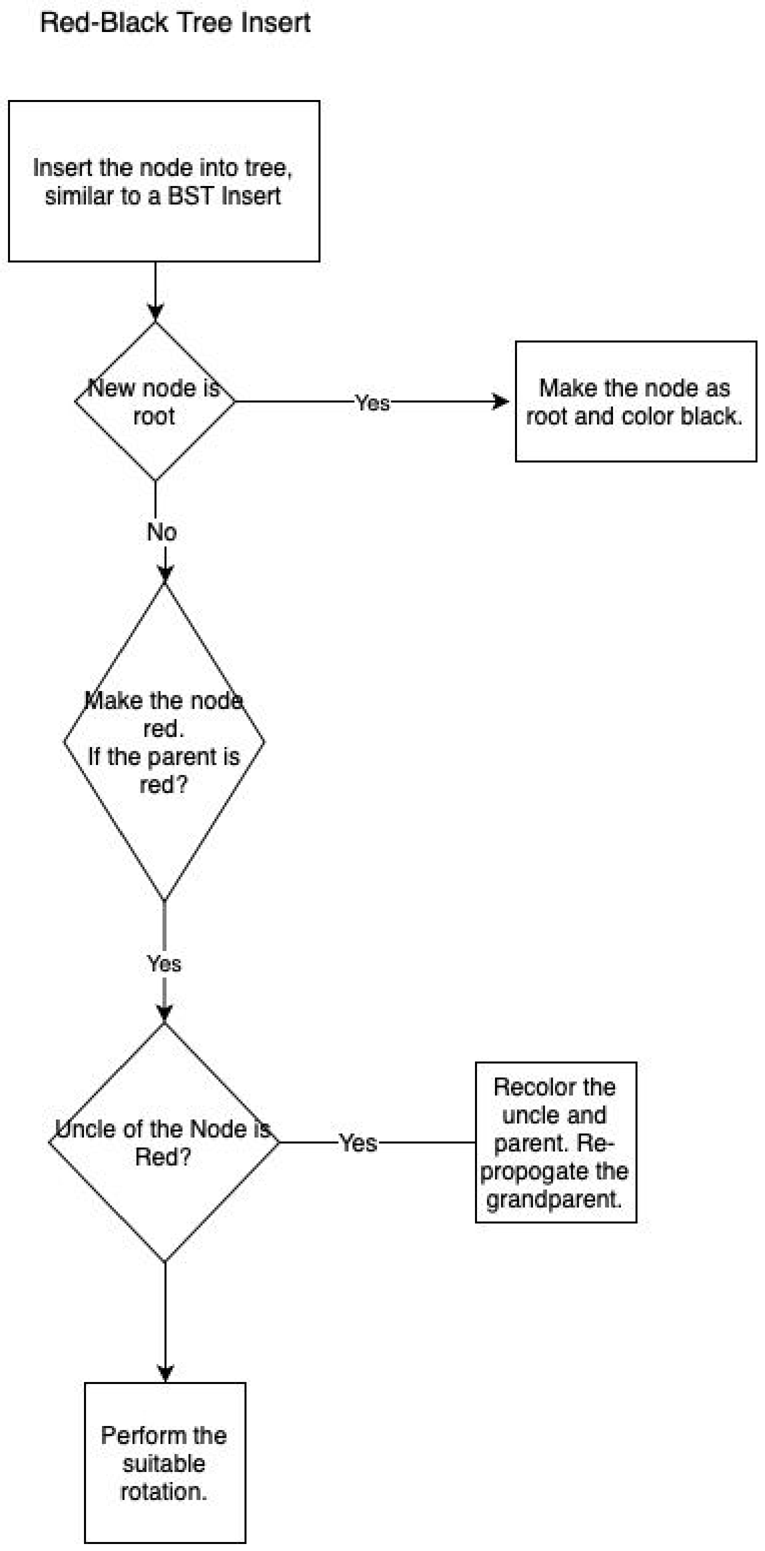
```
java risingCity <Path_to_Input_file>
```

III. Code Structure and Description of the project:

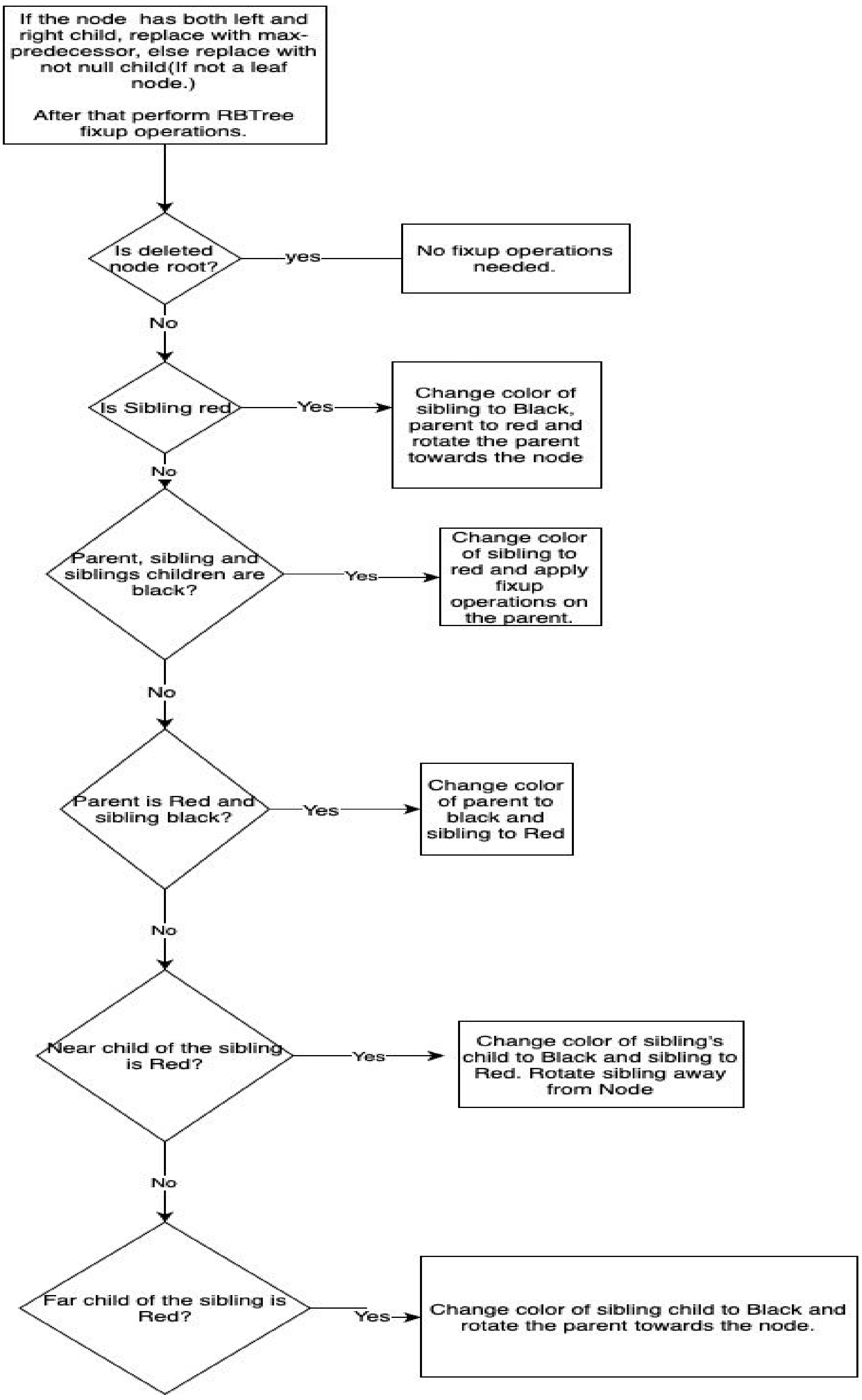
- ❏ risingCity
- ❏ CommandParser
- ❏ RBNode
- ❏ RBTree
- ❏ MinHeapNode
- ❏ MinHeapTree


Read Input line from the given input text-file.
If the time of operation is equal to the global time, execute it.
Pick-Up a building with minimum execution time, and remove it from MinHeap.
If the construction of building is completed?
Yes
Remove the building from Red-Black tree and continue with next min building.
No
Has the building been under construction for last 5 days?
Yes
Add the building back to MinHeap, and pick a new minimum from MinHeap.
No
Increase the execution time for the current building and continue the loop.

Logic for the risingCity Class


Red-Black Tree Insert
Insert the node into tree, similar to a BST Insert
New node is root
Yes
Make the node as root and color black.
No
Make the node red. If the parent is red?
Yes
Uncle of the Node is Red?
Yes
Recolor the uncle and parent. Re-propogate the grandparent.
Perform the suitable rotation.

Insertion Logic for the Red-Black Tree.


If the node has both left and right child, replace with max-predecessor, else replace with not null child(If not a leaf node.)
After that perform RBTree fixup operations.
Is deleted node root?
yes
No fixup operations needed.
No
Is Sibling red
Yes
Change color of sibling to Black, parent to red and rotate the parent towards the node
No
Parent, sibling and siblings children are black?
Yes
Change color of sibling to red and apply fixup operations on the parent.
No
Parent is Red and sibling black?
Yes
Change color of parent to black and sibling to Red
No
Near child of the sibling is Red?
Yes
Change color of sibling's child to Black and sibling to Red. Rotate sibling away from Node
No
Far child of the sibling is Red?
Yes
Change color of sibling child to Black and rotate the parent towards the node.

Deletion logic for the Red-Black Tree.

- ❑ **Class risingCity:**

risingCity is the main class that makes use of the instances of RBTree and MinHeap to solve the risingCity problem.

public static void main(java.lang.String[] args) throws java.lang.Exception:

The main method consists of all logic to read the input from a given file, schedule the construction of the buildings as per the given conditions and write the output to a file.

- ❑ **Class CommandParser:**

A utility class to parse the lines from the input file and return command object consisting of the Insertion time, Command and the arguments.

CommandParser parse(java.lang.String command)

To parse a line read from a file into a command object, that consists of the command name, it's arguments and time.

Parameters:
command - : The line read from the input file.

Returns: An instance of the commandParser object with the set values of command, time and arguments.

- ❑ **Class RBNode:**

RBNode is the Red-Black node which is used in RBTree, which consists of the Building Number, Total time & Executed time.

RBNode(int buildNum,int totalTime)

Constructor to set the building number and total time for a newly created red-black node.

Parameters:
buildNum - : Building number to be set.
totalTime - : Total time for the completion of building.

void setBlack()

To set a given node color to be black.

void setRed()

To set a given node color red.

boolean isBlack()

To check if given node is black.

boolean isRed()

To check if given node is red.

boolean isRoot()

To check if the given node is root.

RBNode getGrandparent()

To get the grandparent of a given node.

RBNode getUncle()

To get the uncle of a given node.

RBNode getSibling()

To get the sibling of a given node.

public void setMinHeapNode(MinHeapNode minHeapNode)

To set the corresponding Minheap Node

Parameters:
minHeapNode - : Corresponding minheap node.

public int getTotalTime()

To get the total time for a given node.

public int getExecTime()

To get the executed time for a given node.

- **Class RBTree:**

Class for the Red-Black tree, with the insert, delete and find functionalities. The RBTree consists of RBNode's.

setRoot
public void setRoot(RBNode node)

To set the root of RBTree.
Parameters:
node - : The node to be set as the root.

isBlackOrNull

public boolean isBlackOrNull(RBNode node)

To determine if a given node is black or null.

Parameters:

node - : Given node.

return: Returns true if the given node is Red and not Null

isRedAndNotNull

public boolean isRedAndNotNull(RBNode node)

To determine if a given node is red and not null.

Parameters:

node - : Given node

return: Returns true if the given node is Red and not Null.

setBlackIfNotNull

public void setBlackIfNotNull(RBNode node)

Set the color of the given node to black if it's not null.

Parameters:

node - : Given node.

isLeftChild

public boolean isLeftChild(RBNode node)

To check if the given node is a left-child of it's parent.

Parameters:

node - : The given node to be checked.

return: True if the node is a left child of it's parent.

isRightChild

public boolean isRightChild(RBNode node)

To check if the given node is a right-child of it's parent.

Parameters:

node - : The given node to be checked.

return: True if the node is a right child of it's parent.

findNode

public RBNode findNode(int buildNum)

To find a node of the given building-Number in the Tree.

Parameters:

buildNum - : Building-Number of the node to be discovered.

return: Node with the given building number.

leftRotate
private void leftRotate(RBNode node)

To rotate a given node towards left

Parameters:
node - : Node to be rotated left.

rightRotate
private void rightRotate(RBNode node)

To rotate a given node towards right.

Parameters:
node - : Node to be rotated right.

transplant
private void transplant(RBNode node, RBNode replacementNode)

To replace the given node with a replacement node.

Parameters:
node - : The node to be replaced.
replacementNode - : The node we are replacing with.

addNode
public RBNode addNode(int buildNum, int totalTime)

To insert a new building into the Red-Black tree.

Parameters:
buildNum - : Building number of a newly added building.
totalTime - : Total time needed for the construction of new building.

remove
public boolean remove(int buildNum)

To delete a building from red-black tree.

Parameters:
buildNum - : Building number of the node to be deleted.
return: True if node is deleted successfully.

insertBST
private RBNode insertBST(RBNode parent, RBNode node)

TO insert the new node into an appropriate position in the tree and set it's parent.

Parameters:
parent - : root of the tree
node - : new Node
return: new Node inserted in appropriate position with it's parent set.

rbInsertFixup
public void rbInsertFixup(RBNode node)

To perform Red-Black tree validations on the newly inserted node and make necessary changes such that it obeys Red-Black tree properties.
Parameters:
node - : The newly inserted node.

insertNewNode
public void insertNewNode(RBNode node)

To assign color to the newly inserted node and perform the required operations.
Parameters:
node - : newly inserted node

insertIsParentRed
public void insertIsParentRed(RBNode node)

To check if the parent of newly inserted node is Black. If yes, we need not make any changes, else we need to make appropriate changes.
Parameters:
node - : newly inserted node.

insertWithRedParent
public void insertWithRedParent(RBNode node)

If the parent of the node is Red, we check for the color of uncle node. If it's red, we recolor the parent and the uncle. Then we recolor the grandparent and recheck.
Parameters:
node - : newly inserted node

insertWithBlackUncle
public void insertWithBlackUncle(RBNode node)

If the parent in Red and uncle is Black/Null, then we perform the required rotations. In this method we perform RL and LR cases.
Parameters:
node - : newly inserted node.

insertWithBlackUncleLLAndRRRotations
public void insertWithBlackUncleLLAndRRRotations(RBNode node)

If the parent is Red and uncle is Black/Null, then we perform the required rotations. In this method we perform RR and LL cases.
Parameters:
node - : newly inserted node.

deleteBST
public boolean deleteBST(int buildNum)

We remove the node with given building number from the tree.
1. If the node has both and left and right children, then we replace it with max-predecessor.
2. If either left/right child is null, we replace it with the non-null child.
3. If both children are null, we then mark the node for deletion. While copying the node, we also copy the color of the replacing node.

Parameters:
buildNum - : Building number of the node to be deleted.
return: Returns true if the delete operation is successful.

rbDeleteFixup
private void rbDeleteFixup(RBNode node)

To fix-up the Red-Black tree property after deleting the node from Red-Black tree

Parameters:
node - : The node that's getting deleted. If the node getting deleted is root we just continue.

deleteWithRedSibling
private void deleteWithRedSibling(RBNode node)

The node getting deleted has a Red sibling

Parameters:
node - : The node that's getting deleted. In this case, we change the color of Sibling to Black and parent to Red and rotate the parent towards the deleted node.

deleteWithParentAndSiblingAndSiblingChildrenBlack
private void deleteWithParentAndSiblingAndSiblingChildrenBlack(RBNode node)

The node getting deleted has black parent, black sibling and even the children of the sibling are black.

Parameters:
node - : The node that's getting deleted. In this case, we change the color of the sibling to red and apply the Red-Black fix-up operation on the parent of the node.

deleteWithRedParentAndBlackSibling
private void deleteWithRedParentAndBlackSibling(RBNode node)

The node getting deleted had Red parent and Black sibling.

Parameters:
node - : The node that's getting deleted. In this case, we change the color of the parent to Black and the Sibling to Red.

deleteWithNearSiblingChildRed
private void deleteWithNearSiblingChildRed(RBNode node)
The node getting deleted has Black parent, black sibling and the sibling has the child which is near to the deleted node of red color.
Parameters:
node - : The node that's getting deleted. In this case, we change the color of the child of sibling to Black and sibling to Red and rotate the sibling away from the node getting deleted.

deleteWithFarSiblingChildRed
private void deleteWithFarSiblingChildRed(RBNode node)
The node getting deleted has Black parent, black sibling and the sibling has the child which is far away from the deleted node and has red color.
Parameters:
node - : The node that's getting deleted. In this case, we change the color of the child of sibling to Black and rotate the parent towards the node that's getting deleted.

replaceWithMaxPredecessor
private RBNode replaceWithMaxPredecessor(RBNode node)
To copy the value of the max predecessor to the given node.
Parameters:
node - : The node to which we want to find the max-predecessor and replace it's values by max-predecessor.
return: returns the max predecessor node which should now be deleted.

maxPredecessor
private RBNode maxPredecessor(RBNode node)
To find the max predecessor for a given node, so that we can replace it when we delete it.
Parameters:
node - : Node whose max predecessor is required.
return: Max predecessor of the given node.

replaceColor
private void replaceColor(RBNode node, RBNode replacement)
To swap the color of given node with that of the replacement node.
Parameters:
node - : The node whose color is to be replaced.
replacement - : The node with whose color we should replace.

print

public java.lang.String print(int buildNum)

Function to print the building node tuple of a given building number.

Parameters:
buildNum - : Building number
return: String consisting of the building details tuple.

print

public java.lang.String print(int buildNum1, int buildNum2)

Function to print the building node tuples between a set of building numbers.

Parameters:
buildNum1 - : First building number.
buildNum2 - : Second building number.
return: String containing the building tuples to be printed.

printRange

public void printRange(RBNode root, int buildNum1, int buildNum2, java.lang.StringBuilder ans)

An util function to print the building nodes between a set of building numbers.

Parameters:
root - : root of the red-black tree
buildNum1 - : First building number.
buildNum2 - : Second building number.
ans - : The string to which we append the values of building nodes.

isEmpty

public boolean isEmpty()

To check if the Red-Black is empty.

return: True if the tree is empty else returns false.

- **Class MinHeapNode:**

MinHeapNode is the node which is used in MinHeap, which consists of the Building Number, Total time & Executed time and also a link to the corresponding Red-Black Node.

MinHeapNode(int execTime)

To set the execution time for a newly created minheap node.

Parameters:
execTime - : The execution time that we should be setting for the new node.

public void setRBNode(RBNode rbNode)

To set the Red Black node for a given Minheap node.

Parameters:

rbNode - : The Red black node that should be set.

public int getExecTime()

To get the execution time of a given Minheap node.

returns: execution time of the Minheap node.

public int getBuildingNum()

To get the building number of the given Minheap node.

returns: Building Number for the given minheap node from the corresponding Red black node.

public int getTotalTime()

To get the total time of the given Minheap node.

returns: Total time for the given minheap node from the corresponding Red black node.

- **Class MinHeap:**

Class for the MinHeap, with the insert(), get_minimum() and remove() functionalities. The MinHeap consists of MinHeapNodes.

public int parent(int ind)

To return the parent of the given Node in minheap.

Parameters:

ind - : The index of the node for which we need to find the parent.

return: The index of the parent of given node.

private void swap(int a, int b)

To swap 2 minheap Nodes in the minheap.

Parameters:

a - : Index of first node.

b - : Index of second node.

public MinHeapNode addNode(int execTime, RBNode rbNode)

To add a new node to the Minheap.

Parameters:

execTime - : Executed time of the newly added node.

rbNode - : Corresponding Red Black node of the newly created node.

return: newly created minheap node.

public MinHeapNode getMin()

To find the minimum Node in minheap

return: returns the first node in Minheap if it's size is greater than 0.

public void remove(int ind)

To remove the node of given index from Minheap.

Parameters:
ind - : The index of node to be removed from Minheap.

private void minHeapify(int pos)

To perform Min Heapify operation at the given position in minheap.

Parameters:
pos - : The position at which we need to perform Min heapify operation.

private int getLeftChild(int ind)

To get the index of left child of given Node.

Parameters:
ind - : Index of the given node.
return: Index of the left child of the given node.

private int getRightChild(int ind)

To get the index of right child of given Node.

Parameters:
ind - : Index of the given node.
return: Index of the right child of the given node.