Advanced Data Structures (COP 5536)
Project Report

Darshan Dilip Kakwani
UFID: 5311 7117 d.kakwani@ufl.edu

Description

The aim of the project was to develop a program that is used to develop a city. A city is constructed by constructing multiple buildings. A building record consists of the following fields:

buildingNum: a unique integer for each building
executed_time: number of days that the building has been worked on for
total_time: number of days that the building needs to be worked on for completion.

There are three different operations that are supported:

Insert(buildingNum, total_time): A new building that is to be constructed
Print(buildingNum): Print the building record of the building with buildingNum = buildingNum
Print(buildingNum1, buildingNum2): Print all the building records with buildingNum between buildingNum1 and buildingNum2.

The input consists of a sequence of the operations stated above.
Each building should be executed for either 5 days or until it is completely constructed, i.e. executed_time is equal to total_time. The order in which the buildings are to be executed is determined by their executed_times. If the executed_time of multiple buildinds is equal, then they should be ordered by buildingNum.

The data structures that would be used are a Min Heap and a Red Black Tree.
The Min Heap is used to keep track of every building until it is completely constructed. It stores the building records ordered by executed_time. The Minimum Heap is minHeapified whenever the building at the root of the Minimum Heap completes either 5 days of execution or it finishes its construction. [Note: The Minimum Heap is not minHeapified at every insert].
The Red Black Tree is used for printing. Pointers have to be maintained between corresponding nodes in the Min Heap and the Red Black Tree.

The project has been implemented in JAVA. The input is provided in a .txt file. The output has to be written into a .txt file named output_file.txt.

Running Instructions

In order to execute the program, the files need to be unzipped first from the .zip file names "Kakwani_DarshanDilip.zip".
The java file needs to be compiled. This can be done using the following command:

```
javac risingCity.java
```

This will create the class files that can now be used to execute the JAVA code.
To execute the code, run the following command:

```
java risingCity filename.txt
```

This will run the code for the input file named filename.txt and give the required output in another file named "output_file.txt".

[Note: Two different input files were included in the .zip file. If the code needs to be tested on another input, it needs to be imported into the same directory as the .java and .class files.]

Program Structure

The program has only 1 .java file named risingCity.java. This file has 5 different classes.

<u>class Record</u>
Description: The class used to store the building records.
Parameters:

- int buildingNum
- int executed_time
- int total_time

<u>class MinHeap</u>
Description: The class used for the implementation of the Minimum Heap.
Parameters:

- Record[] Heap: Array used to maintain the Minimum Heap
- int size: Current size of the Minimum Heap
- int max_poss_size: Maximum possible size of the Minimum Heap [Given: 2000]

Functions:

- int l_child
    - *Description:* A function used to return the left child of the node passed as a parameter.
    - *Parameters:*
        - int current: Position of current node
- int r_child
    - *Description:* A function used to return the right child of the node passed as a parameter.
    - *Parameters:*
        - int current: Position of the current node

- boolean check_leaf
    - *Description:* A function used to check if the current node is a leaf node. It returns true if current node is a leaf. Else, it returns false.
    - *Parameters:*
        - int current: Position of the current node
- void exchange
    - *Description:* A function used to swap two different nodes at two different positions.
    - *Parameters:*
        - *int pos1:* Position of the first node
        - *int pos2:* Position of the second node
- void minHeapify
    - *Description:* A function used to heapify the Minimum Heap. It brings the node with the lowest executed_time (buildingNum, in case of equal executed_time) to the top of the Minimum Heap using a series of swaps going down the tree.
    - *Parameters:*
        - int current: Position of the current node
- void insert
    - *Description:* A function used to insert a new node into the Minimum Heap.
    - *Parameters:*
        - Record element: The building record that has to be inserted into the Minimum Heap
- void minHeapFunc
    - *Description:* A function used to minHeapify the entire tree. It calls the minHeapify function on every non leaf node.
    - *No parameters*
- Record remove
    - *Description:* A function used to remove an existing node from the Minimum Heap. It also returns the Record that was removed from the Heap.
    - *No parameters*

<u>class rbt_element</u>
Description: The class used to store the Node structure used in the Red Black tree.
Parameters:

- Record record_element: The building record containing the triplet (buildingNum, executed_time, total_time).
- int color: An integer value to indicate the color of the Node.
- rbt_element l_child: The Red Black Tree Node that indicates the left child of the current Node. Default Value: external.
- rbt_element r_child: The Red Black Tree Node that indicates the right child of the current Node. Default Value: external.
- rbt_element parent_node: The Red Black Tree Node that indicates the parent of the current Node. Default Value: external.

class RedBlackTree

Description: The class used for the implementation of the Red Black Tree.

Parameters:

- int COLOR_RED: An integer value to indicate that the color of a node is RED.
- int COLOR_BLACK: An integer value to indicate that the color of a node is BLACK.
- rbt_element external: A Red Black Tree Node that denotes external nodes. Children of leaf nodes are external Node.
- rbt_element root: The Red Black Tree Node that is the Root of the Tree. Parent of root node is external Node.

Functions:

- boolean inRange
  - *Description:* A function to check if a value passed to it is between a given range. Returns true if value is between given range. Else, returns false.
  - *Parameters:*
    - int bNo: The buildingNum passed to the function to check if it is between the given range.
    - int lower: The lower bound of the range.
    - int higher: The higher bound of the range.
- String printRecursively
  - *Description:* A function to print all nodes in the Red Black Tree that have their buildingNum between the given range. Returns the string.
  - *Parameters:*
    - rbt_element current: The current Node.
    - int lower: The lower bound of the range.
    - int higher: The higher bound of the range.
- String printInRange
  - *Decsription:* A function to print all nodes in the Red Black Tree that have their buildingNum between the given range. Returns the string. If no such Nodes exist, the function returns "(0,0,0)".
  - *Parameters:*
    - int lower: The lower bound of the range.
    - int higher: The higher bound of the range.
- String printNode
  - *Description:* A function to print the specific node in the Red Black Tree. Returns the Node building record. If no such Node exists, the function returns "(0,0,0)".
  - *Parameters:*
    - int bNo: The buildingNum passed to the function.

- rbt_element nodeSearch
    - *Description:* A function to find a particular Node in the Red Black Tree. Returns the Node if found. Else, the function returns null.
    - *Parameters:*
        - rbt_element currNode: The Node in the Red Black Tree where the search should start from.
        - rbt_element node: The Node in the Red Black Tree that is to be searched for.
- void insert
    - *Description:* A function used to insert a new Node into the Red Black Tree.
    - *Parameters:*
        - rbt_element node: The Node that is to be inserted into the Red Black Tree.
- void insertFix
    - *Description:* A function to fix the Red Black Tree after insertion of a new Node.
    - *Parameters:*
        - rbt_element node: The same node that was inserted into the red Black Tree.
- void leftRotation
    - *Description:* A function that left rotates the Red Black Tree around a given Node.
    - *Parameters:*
        - rbt_element node: The Node around which the Red Black Tree has to be rotated.
- void rightRotation
    - *Description:* A function that right rotates the Red Black Tree around a given Node.
    - *Parameters:*
        - rbt_element node: The Node around which the Red Black Tree has to be rotated.
- void transplant
    - *Description:* A function that connects the Red Black Tree after any Node is deleted.
    - *Parameters:*
        - rbt_element target: The Node that is to be deleted from the Red Black Tree.
        - rbt_element with: The Node that is to be used to connect the disconnected tree back to the original Red Black Tree.
- boolean delete
    - *Description:* A function used to delete a particular Node from the Red Black Tree. The function returns true if the delete was successful. Else, it returns false.
    - *Parameters:*
        - rbt_element c: The Node that is to be deleted from the Red Black Tree.

- void deleteFix
    - *Description:* A function that is used to fix the Red Black Tree after the deletion of a Node.
    - *Parameters:*
        - rbt_element a: The Node that becomes the new root of the subtree after deletion of a Node.
- rbt_element findMinimumNode
    - *Description:* A function that finds the minimum value node in the Red Black Tree. This would be the left most Node of the Red Black Tree. The function returns this Node.
    - *Parameters:*
        - rbt_element subTreeRoot: The root of the subtree of the Red Black Tree from where the minimum value node has to be found.
- void redBlackTreeHelper
    - *Description:* A function that is a driver code for the implementation of the Red Black Tree.
    - *Parameters:*
        - int choice: An integer value that indicates whether the insert or the delete function of the Red Black Tree is called.
        - Record item: The Node that is to be inserted into / deleted from the red Black Tree.

class risingCity
Description: The main driver class of the JAVA program that connects everything together.
Functions:

- void main
    - *Description:* The main function of the driver class. This is the first function that is executed when the code is ran.
    - *Functions:*
        - Initialize the Minimum Heap and the Red Black Tree.
        - Get the Input File.
        - Write into the Output File.
        - Insert, Delete and Update into the Minimum Heap and the Red Black tree based on the problem description.
    - *Parameters:*
        - MinHeap minHeap: The MinHeap class.
        - RedBlackTree redBlackTree: The Red Black Tree class.
        - File inputFile: The input file to be used passed as an argument of the main function.
        - String line: Individual lines in the inputFile.
        - String contents[]: line split into word and non word characters.
        - Global_counter: The integer value that indicates how many days of execution have been completed in the building of the city.

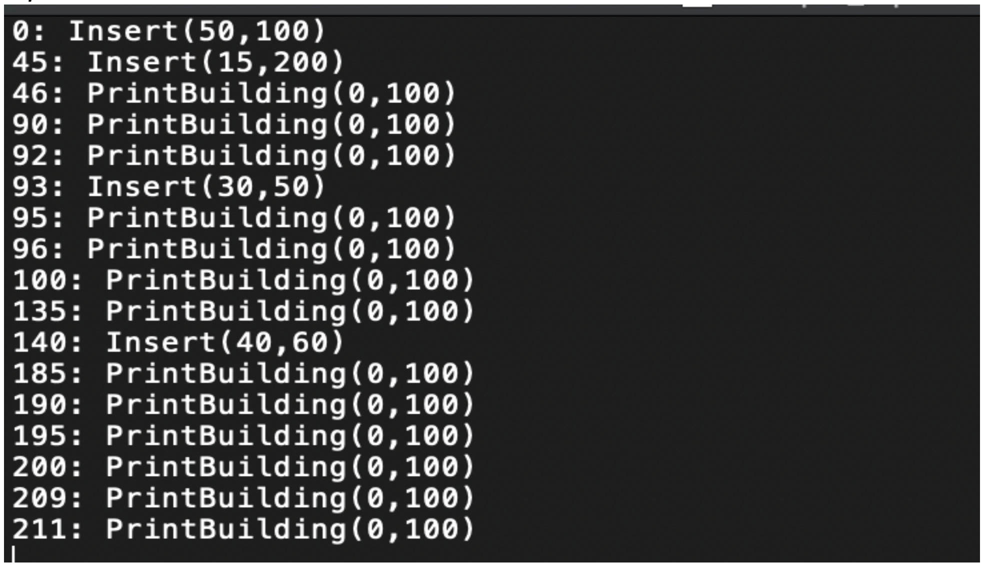
- List<Record> waitList: The ArrayList that stores all the insert commands until the Node can be inserted into the Minimum Heap.
- BufferedWriter bufferedWriter: The writer that is used to write all the output into the output file.

Test Cases

The JAVA program has been tested for the following and more test cases.

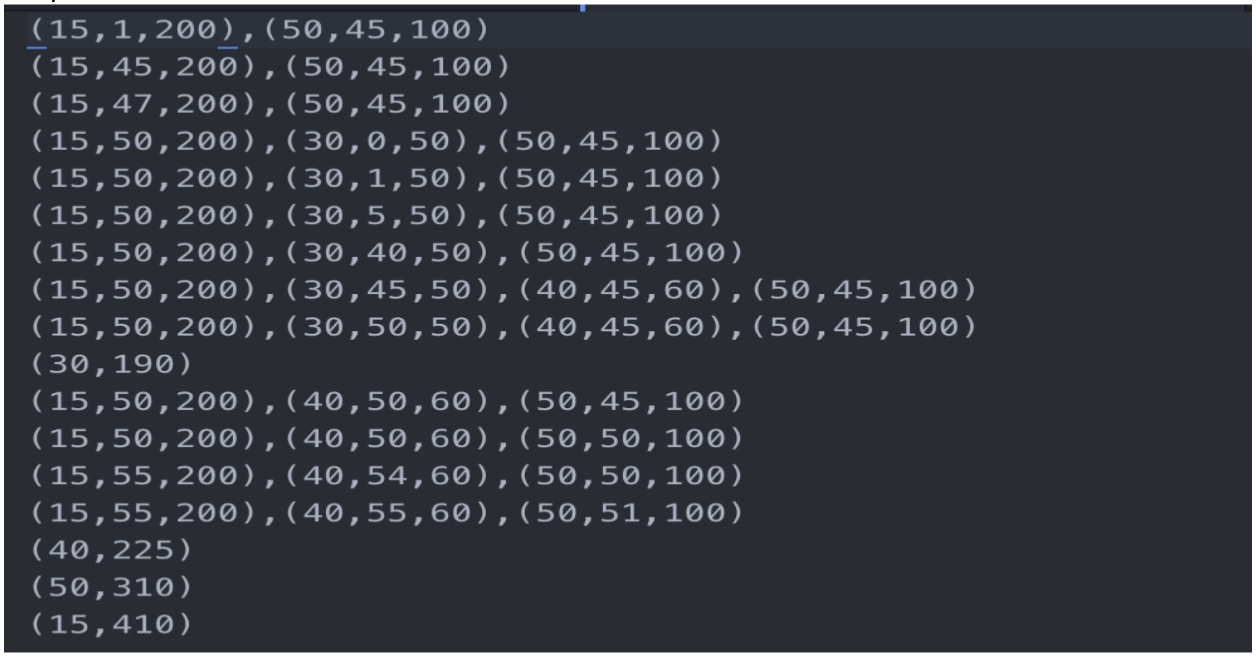
Case 1

Input:



```
0: Insert(50,100)
45: Insert(15,200)
46: PrintBuilding(0,100)
90: PrintBuilding(0,100)
92: PrintBuilding(0,100)
93: Insert(30,50)
95: PrintBuilding(0,100)
96: PrintBuilding(0,100)
100: PrintBuilding(0,100)
135: PrintBuilding(0,100)
140: Insert(40,60)
185: PrintBuilding(0,100)
190: PrintBuilding(0,100)
195: PrintBuilding(0,100)
200: PrintBuilding(0,100)
209: PrintBuilding(0,100)
211: PrintBuilding(0,100)
```

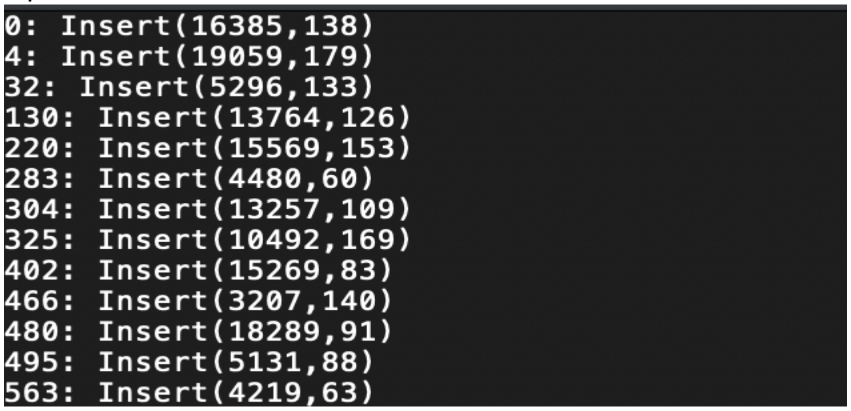
Output:



```
(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)
```

<u>Case 2</u>

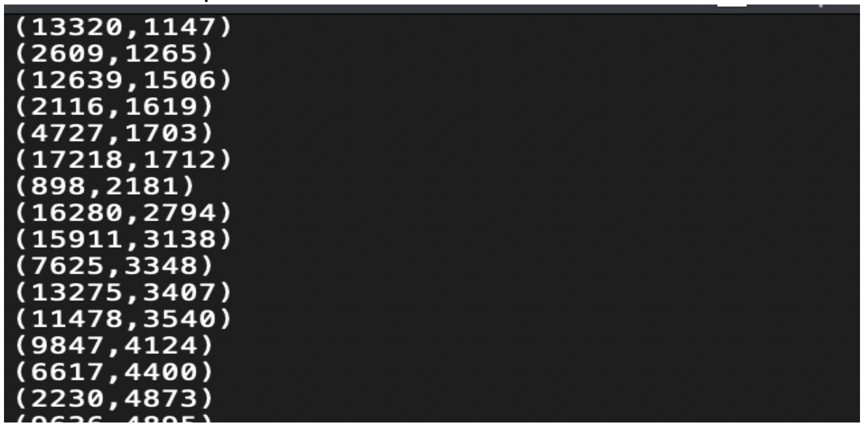
Input:



```
0: Insert(16385,138)
4: Insert(19059,179)
32: Insert(5296,133)
130: Insert(13764,126)
220: Insert(15569,153)
283: Insert(4480,60)
304: Insert(13257,109)
325: Insert(10492,169)
402: Insert(15269,83)
466: Insert(3207,140)
480: Insert(18289,91)
495: Insert(5131,88)
563: Insert(4219,63)
```

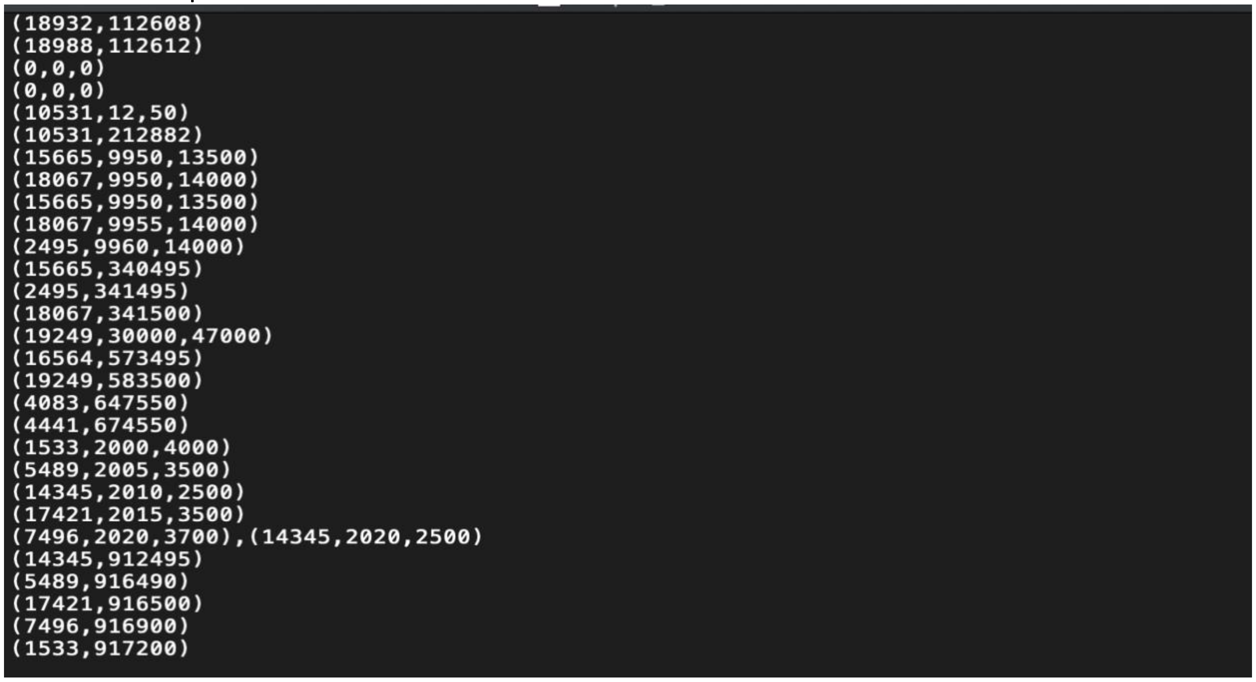
Output:

Start of the output



```
(13320,1147)
(2609,1265)
(12639,1506)
(2116,1619)
(4727,1703)
(17218,1712)
(898,2181)
(16280,2794)
(15911,3138)
(7625,3348)
(13275,3407)
(11478,3540)
(9847,4124)
(6617,4400)
(2230,4873)
```

End of the output



```
(18932,112608)
(18988,112612)
(0,0,0)
(0,0,0)
(10531,12,50)
(10531,212882)
(15665,9950,13500)
(18067,9950,14000)
(15665,9950,13500)
(18067,9955,14000)
(2495,9960,14000)
(15665,340495)
(2495,341495)
(18067,341500)
(19249,30000,47000)
(16564,573495)
(19249,583500)
(4083,647550)
(4441,674550)
(1533,2000,4000)
(5489,2005,3500)
(14345,2010,2500)
(17421,2015,3500)
(7496,2020,3700),(14345,2020,2500)
(14345,912495)
(5489,916490)
(17421,916500)
(7496,916900)
(1533,917200)
```