ADVANCED DATA STRUCTURES

PROJECT REPORT
(COP 5536)

NIKHILESH REDDY TUMMALA
tummalanikhilesh@ufl.edu
(8350 – 1593)

1. Introduction
This project is written in JAVA 1.8.0 version. The classes I wrote in this project are as follows:

- RisingCity.java
- Building.java
- MinHeap.java
- RedBlackTree.java
- RedBlackNode.java
- ConvertingInputCommand.java

Each of these java files have single java class whose method implementations are comprehensively described in this report.

2. RisingCity.java
This file has the below method implementations:

2.1. public static void main(String[] args)
In this method, the input file goes through a function called "convertInputCommands" and returns a hash-map which has the day number as key and its value is an object which has the input command and its arguments. If the command is 'Insert', then the "ConvertingInputCommand" object stores this command and the building number and total time of the building that follows the command. I then create an object of Red-Black Tree and an object of Min-Heap with maximum capacity of 2000 buildings. Later, we have two temporary data structures, a list of type Buildings (Only to store the under constructed buildings) and a hash-map whose key is the building number and its value is of type Building class.

We initialize the global counter to -1 which acts like a day counter to use in a loop such that in every iteration of the loop, this counter is increased by 1 (just like each iteration in the loop is like each day). And we set it to -1 so that when we start the process, the counter begins with 0. I changed the properties of 'out' such that every time I write "System.out.print();" it prints into an output file "output_file.txt" rather than on console. Another counter called 'No_of_Days_Since_Previous_Building_began', is to know how many days it has been since the building's construction began. We have a waitlist which is essentially a hash-map containing the key as building number and value of type Building. The usage of these structures will be mentioned further.

There is another counter to keep track of how many insertion commands are given to us. Therefore, I ran a for loop across the hash-map containing all the input commands to find out the number of insert statements. This data is stored in "InsertCommmandCounter". A while loop begins with a condition that the loop ends when this counter becomes zero. After every insert command is encountered, this variable is reduced by 1. In this loop we start the global counter and check if there is any building which is already taken under construction. Since this is the start of the program, there won't be any. So, the

program moves to checking whether the key in the hash-map "commandMap" is same as the global counter. If it is, then we read the command and direct it to its corresponding functionality using switch-case operation. If the command is 'PrintBuilding', then from the 'commandMap hash-map we get the arguments that were given to us along with 'PrintBuilding'. Those arguments are two building numbers. We need to print the details of all the buildings whose building number lies in the range of those two given building numbers.

We store the first building number in 'start' variable and second in 'sec' variable. If they both are same, we just display the building details of only 'start'. But if they aren't, then we run a for-loop which searches through the red-black tree and returns those nodes(buildings) and we display all their details like building number, execution time and total time. 'buildingCount' and 'FlagVariable1' are used only for the output to appear in a desired format. If we don't find the nodes we are looking for, then we display zeroes instead.

In other case, if the command we encounter when the global counter equals the key in the 'commandMap' is an insert statement, then we create a new building object, enter its building number, execution time and total time into the object and add it into the hash-map 'waitList'. If the min-heap size is zero, then I insert the new building into the min-heap. Or else, I put it in 'List_Of_Under_Constructed_Buildings' and insert this new building's building number into the red-black tree. Instead of creating another building object to insert in red-black tree, I have inserted only the building number into the tree.

If the days since last building started is still zero and the min-heap size is one and there is no building on which we are currently working on, then we take the root node of min-heap and start out construction work on it. If there is already a building that we are currently working on and it's execution time equal total time, then that means the building construction is complete. So, we check what the min-heap size is. If it is 1, then we remove it and dump all the buildings from the 'List_Of_Under_Constructed_Building' into the min-heap and call heapify. Now that we have finished working on one building, we need to move to another and that would be the root of the min-heap. Since the previous building is completed, we remove it's building number from the red-black tree and the 'waitList as well and update the insert command counter by reducing it by 1. Later, we display the details of the completed building.

In another case, if we reach No_of_Days_Since_Previous_Building_began'to 5, then we re-insert all the buildings from the 'waitList' into the min-heap, changing the No_of_Days_Since_Previous_Building_began' counter back to zero, clear the temporary list and take the root of the min-heap and start constructing it.

2.2. private static MinHeap heapInsert(HashMap<Integer, Building> waitList)

This method takes the Hash-Map as input parameter, dump all the buildings inside it into the heap and return that heap.

2.3.public static HashMap <Integer, ConvertingInputCommand> convertInputCommands (String inputFileName)

In this method, we create a new hash-map to store the day count as key and it's corresponding command along with it's arguments as value. These values are stored in an object called 'ConvertingInputCommand' which is the value stored in the map. This function mainly focuses on string manipulation to format the given input which is read from the file into another format which is easy for programming.

3. ConvertingInsertCommand.java

This class contains only a constructor to transfer the given command along with it's arguments into the variables and the process is explained in public static HashMap<Integer,ConvertingInputComand> convertInputCommands (String inputFileName).

4. Building.java

When we create a new building in the main class, we send the building number, execution time and total time along with it. The constructor stores these details into its object.

5. RedBlackNode.java

I have made this class generic so that we can store any type of data we want. Therefore, the parent, left and right nodes are also made generic. But in the nodes, we are only storing the building number and not the whole building. While creation of each node, the constructor initializes these variables and among those, the node colour is set to black by default. Later in the algorithm we change it to red if needed. The constructor with the key as parameter stores the building number in the node. The whole red-black tree is built, based on comparing the key (Building Number).

6. RedBlackTree.java

We create a node 'nil' and set it to root. While creating each node, the constructor initializes the parent, left and right nodes to nil.

6.1. public void insert(T InsertKey)

This method takes the key (building number) as input and calls the insert function which takes the input parameter an object of RedBlackNode. This function doesn't return anything.

6.2. private void insert(RedBlackNode<T> nodeTobeInserted)

First, we directly insert the node into the tree without checking the red-black tree rules. It just performs the binary search tree insertion process regardless of Red-Black Tree rules. This function doesn't return anything.

6.3. private void insertFixup(RedBlackNode<T> insertedNode)

In this method, we check on the newly added node in the tree whether it is following all the rules of Red-Black Tree. If there are any necessary corrections required, like left or right rotations or color-flip, etc are done here. This function doesn't return anything.

6.4. public RedBlackNode<T> treeSuccessor(RedBlackNode<T> a)

We find out the successor for a node during deletion process, and this function is called during that time. This method will go through the children of the given node to find out the successor of that node and return it.

6.5. public RedBlackNode<T> treeMinimum(RedBlackNode<T> node)

This function is called by trueSuccessor(RedBlackNode<T> node) to help in the process of finding out the successor of the given node and return it.

6.6. public void remove(RedBlackNode<T> nodeToBeRemoved)

This function will remove the required node that needs to be deleted from the tree, but it may not obey all the rules of red-black tree by the end of the process. Therefore, we call removeFixup(RedBlackNode<T> a) to make all the necessary corrections. This function doesn't return anything.

6.7. private void removeFixup(RedBlackNode<T> a)

After deleting the node from the tree and replacing with its successor, the end result may not satisfy all of red-black tree rules. This method will do all those required corrections. It won't return anything.

6.8. public RedBlackNode<T> searchNode(T keyToBeSearched)

This node will iterate through the tree to check if the given input key is same as any key in the tree. If it is, then it returns that particular node. Else, it returns null.

6.9. public int size()
This function returns the integer value of the total number of nodes in the tree.

6.10. private boolean isNil(RedBlackNode node)
This function will check if the given node is nil or not. If it is then it returns 'true'. Else, it returns 'false'.

6.11. private void fixNodeData(RedBlackNode<T> a, RedBlackNode<T> b)
This function fixes the details inside the node by updating the number of left nodes and number of right nodes by assuming the given node as a root to this subtree. This function doesn't return anything.

6.12. private void rightRotate(RedBlackNode<T> NeedRightRotate)
This function does RL or RR rotations around the given input node. This method doesn't return anything.

6.13. private void rightRotateFixup(RedBlackNode RightRotateFix)
This function is called by rightRotate(RedBlackNode<T> NeedRightRotate) to make all the necessary changes or corrections after performing the required rotations. This function doesn't return anything.

6.14. private void leftRotate(RedBlackNode<T> NeedLeftRotate)
This function does LR or LL rotations around the given node, depending upon the scenario and calls the leftRotateFixup(RedBlackNode LeftRotateFix) function to clean up if needed. This function doesn't return anything.

6.15. private void leftRotateFixup(RedBlackNode LeftRotateFix)
This function is called by leftRotate(RedBlackNode<T> NeedLeftRotate) to perform the required corrections after performing the appropriate rotations. This function doesn't return anything.

7. MinHeap.java
This has a size variable to know how many nodes are present in the tree and the capacity is the maximum size the nodes can fit in the tree. The tree here is stored in the form of an array called 'heap' where if we are on index- i, we can find the parent of 'i' in heap[i/2]. And we can find the left and right child of node present in i by going to heap[2*i] and heap[(2*i) +1 /2] respectively. For this to happen I stored a dummy building node in heap[0]. The constructor initializes the capacity to the capacity received, size is zero, heap array capacity is set to 1 more than the value received so that we can put that dummy building in the initial position. This dummy building will not affect our min-heap in anyway. The heap is built by comparing the execution time of the building. Below are the methods in the MinHeap class.

7.1. private int parent(int location)
After receiving the location of the current node, we return its parent's location from the heap array.

7.2. private int leftChild(int location)
After receiving the current location of the node, we locate its left child as described above and return it.

7.3. private int rightChild(int location)
After receiving the current location of the node, we locate its right child as described above and return it.

7.4. private boolean ifLeaf(int location)
This function will return true if the given location of the node is leaf. Or false if it isn't. This method will not return anything.

7.5. private void swap(int first, int second)
This function will swap two integer numbers and won't return anything.

7.6. public void heapify(int location)
After deletion heapify is called to do all the necessary corrections to satisfy min-heap properties. This method will not return anything.

7.8. public void insert(Building element)
This function will add a new building into the array and does all the swapping to make sure the min-heap property satisfies. If the execution time of the buildings are same, then comparison is based on building number. The one with lesser building number goes up and the bigger one becomes the child. This method will not return anything.

7.9. public Building remove()
This function will delete the root node building by overriding it with the last node of min-heap and performing heapify operation. Size is reduced by 1 and the deleted node is returned.