ADVANCED DATA STRUCTURES

COP 5536

FALL 2019

PROGRAMMING PROJECT

Submitted by:

Name: Sri kanth Juluru

UFID: 9279-1918

Email: srikanth.juluru@ufl.edu

Introduction:

MinHeap class is used to maintain the min heap data structure. I am using 3 arrays to store values of building number, execution time and total time in each array. We will be using variable size to find the size of min heap. Since we will have not more than 2000 active buildings, we assigned array size as 2000. If buildings construction is done then we will decrease the size of the min heap as well.

RedBlackTree handles the red black tree data structure. Upon each insertion and deletion we will maintain the number of black property by calling their fixup methods. We are using a model class Building whose object will be used as a node in data structure.

Function Prototypes:

1. **Class RisingCity**
   RisingCity is the main class containing the main method. Reading and Writing the input, output will be done in this class.

2. **Class MinHeap**
   MinHeap class implements the min heap data structure for the project.

3. **Class Building**
   Building class is a model class which assigns the values to its variables building_num, execution_time, total_time.

4. **Class RedBlackTree**
   RedBlackTree implements the Red-Black tree data structure. It contains methods which perform red black tree operations like insert(building_num), leftRotate(node), rightRotate(node) and deleteNode(node) etc.,

Methods of class RisingCity:

i. **public static void main(String[] args) throws Exception**
   main method is the point of entry into code. It takes file as an argument which is expected to be in ".txt" format. This method takes the global time and input command from input and passes it to buildingProcess(endTime, inputCounter, inputCommand) method. Output is stored in "Output_file.txt".

ii. public static void buildingProcess(int endTime, ArrayList<Integer> inputCounter, ArrayList<String> inputCommand)

endTime is the last global time for which we are taking input. inputCounter is the list of global times where we need to perform some operation. inputCommand is a list of input commands which needs to be performed at respective global time.

We are inserting the element at 0 counter and then starts the loop. We will take min{5, total_time-execution_time}at root and runs the inside while loop. We will be running 2 loops, outer loop for all commands and inner loop (construction loop)(i.e., 5 day counter) for building construction.
If any print operations comes during this construction phase then execution time will be updated in red black tree and node will be printed. If it is a printBuilding(build_num), we will only print building properties of that particular building and for ranged printing we will print all the buildings in that range by doing inorder traversal. During the construction phase we will increase the global counter value.

For any insert operation during the construction phase we will wait until the phase completes to insert in min heap but node will be added to red black tree during the phase itself. We will add these insert operations to a queue and retrieve back after construction phase.

After construction phase, we will update the execution in min heap and red black tree. If new execution time is equal to the total time, then we will delete the node from min heap and red black tree. If any print operation comes at the end of construction phase and also its updation will finish the construction of building, then the node will be printed according to the print condition and then it gets deleted.

After building completions and updations, we will perform the insert operations which are in queue. We will insert each building into min heap and red black tree and removes from the queue. Since we are using a queue, it will be a first come first serve basis.

Methods of class MinHeap:

i. **public void buildHeap()**
buildHeap() method is used to validated the min heap property across all the nodes of min heap. For every internal node we perform heapify to check and satisfy that its subtree is a min heap.

ii. **public boolean isLeaf()**
returns whether a node is a leaf node or internal node.

iii. **public void heapify(int pos)**
heapify method takes position of node as an arguments and check subtree rooted with that node is a min heap and if not then do necessary changes to satisfy the property.

iv. **public int getSize()**
returns the size of min heap.

v. **public int leftChild(int pos)**
Takes position of node as an argument and returns the left child of node at position pos. If there is no left child for that node then returns -1.

vi. **public int rightChild(int pos)**
Takes position of node as an argument and returns the right child of node at position pos. If there is no right child for that node then returns -1.

vii. **public int parent(int pos)**
parent method takes position of node as an argument and returns the parent of node.

viii. **public void swap(int node1, int node2)**
Takes position of two nodes to be swapped as arguments and swaps them. We will swap execution time, total time and building number of two nodes.

ix. **public int removeMin()**
returns the execution time of root node.

x. **public void insert_heap(int exe_time, int tot_time, int build_num)**

insert_heap takes execution time, total time and building number of new node and inserts new node into min heap. We will insert new node at the end of the heap and perform heapify operation.

xi. **public int getExecutionTime()**
returns the execution time of root node.

xii. **public int getTotalTime()**
returns the total time of root node.

xiii. **public int getBuildingNum()**
returns the building number of root node.

xiv. **public void updateExecutionTimeByX(int localCounter)**
takes localCounter variable as an argument and updates the execution time of root. Increases the execution time of root node by localCounter value. This method is used to updated execution time of root for every 5 global time change.

Methods of class RedBlackTree:

i. **private RedBlackNode doesNodeExist(RedBlackNode find, RedBlackNode node)**
doesNotExist method takes two nodes find and node as input arguments and searches whether we can get find in tree rooted with node. If found return the node if not return tnil (a sentinel node)

ii. **public void insert(Building item)**
insert method takes building object as an argument. Building object contains building number, execution time, total time.
We will find the right position to insert this object/node by calling the nearestPosition(node) method. After inserting as red node we will validate the red black property by calling RB_Insert_Fixup(node).

iii. **private void RB_Insert_Fixup(RedBlackNode node)**
we will check whether the parent to newly inserted node is black or red. If the parent is black then no need to make change since number of blacks wont change by insertion of new node. If not we have to maintain the number of

black property by left rotation, right rotations and color flips. To perform left and right rotations we will call leftRotate(node) and rightRotate(node) respectively.

iv. **private RedBlackNode nearestPostion(RedBlackNode target)**
nearestPostion takes target node as an argument searches for the right position to insert by calling nearestPostion(node, target). This method returns the nearest node which will be returned by the current method.

v. **private RedBlackNode nearestPostion(RedBlackNode node, RedBlackNode target)**
nearestPostion takes target node as an argument searches for the right position to insert in subtree rooted at node.

vi. **void leftRotate(RedBlackNode node)**
this method performs the left rotation based on that node.

vii. **void rightRotate(RedBlackNode node)**
this method performs right rotation based on that node.

viii. **void RB_transplant(RedBlackNode u, RedBlackNode v)**
RB_transplant method is used to replace one subtree as a child of its parent with another subtee.

ix. **boolean deleteNode(RedBlackNode z)**
deleteNode deletes the node which is passed as an argument. If the node does not exist then it will return false else true. It will call RB_Delete_Fixup method to fixed black property of red black tree.

x. **void RB_Delete_Fixup(RedBlackNode x)**
After node is deleted, this method will fixup the red black property of the tree rooted at node x.

xi. **RedBlackNode treeMinimum(RedBlackNode subTreeRoot)**
This method returns the building object whose building number is minimum in the subtree rooted at subTreeRoot.

xii. **public RedBlackNode findNodebyId(int building_num)**
findNodebyId takes building_num as an argument and finds the node with that building number. This method calls findNodebyId(node, theID) with root as node.

xiii. **private RedBlackNode findNodebyId(RedBlackNode node, int theID)**
This method takes node and building number from arguments and returns the root whose building number is same as the argument. If no node exists then we will return sentinel node.

xiv. **public void printTree(int lowest, int highest)**
If red black is an empty tree then prints “(0,0,0)” otherwise calls printTree method with 3 arguments (root, lowest, highest).

xv. **public void printTree(int building_num)**
This method takes the building_num as argument and prints the details of the node whose building number is the argument. We can find that node by calling findNodebyId(building_num). If there is no node with that building number then it will print “(0,0,0)”

xvi. **public void printTree(RedBlackNode node, int lowest, int highest)**
This method takes 3 values as arguments. Lowest and highest values are the range in which building number needs to be present. “node” is the root of the subtree in which search needs to be done. Inorder traversal will be done for searching the node.

xvii. **public void deleteKeyPair(int building_num)**
This method will call the findNodebyId(building_num) and calls delete(node) to delete the node with its building properties/values.

xviii. **public void updateExecutionTime(int building_num, int x)**
this method will take building number and x as input. It fill find the node with building_num and changes its execution time to x.

Programming Environment

Software:

Programming language – Java v12.0.2

Hardware:

Operating systems: Windows 10

RAM: 32GB

Processor: Intel core i7-4810MQ

Sample snapshots of Input and Output:

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)
```