Project – Rising City
COP 5536 – Fall 2019

Student details:
Ankur Sethi UFID: 9351-2951
ankur.sethi@ufl.edu

IMPLEMENTATION DETAILS:

The project utilizes the data structures Red-black tree and min-heap for maintaining building data and properties.
The ***MinHeap*** and ***RedBlackTree*** classes take care of the data structures, while the main processing of the building contruction happens in the ***risingCity*** class (specifically the ***risingCity.process(Scanner sc)*** method).
Following is the documentation of classes and their methods:

1. **Building**

   This is the fundamental class which on which the other data structures are built. It contains the elementary building data.

   Contains the following fields:

- ***int buildingNum***: integer field for building number, must be unique
- ***int executedTime***: integer field for current executed time, initially is 0
- ***int totalTime***: integer field for totalTime required to construct the building

  Methods:

- ***Building(int buildingNum, int totalTime)***
  Constructor method to set the buildingNum and totalTime as given and executedTime as 0.

- ***int compareTo(Building o)***
  Compare two buildings objects, first comparing their respective executed times, and then comparing the building numbers to break ties

2. **MinHeap**

   Class implementing the min-heap data structure. Uses the ***buildingObject.compareTo(Building anotherBuilding)*** to compare two building objects and place accordingly in the heap. Thus, the key for the heap is executed time, followed by the building number.

Contains the following fields:

- ***ArrayList<Building> heapArray****:* Array of element type as ***Building*** which maintains the min heap data with the index of the elements sorted as per the property of min heap.
- ***int heapSize****:* an interger to maintain the size of the heap

Methods:

- ***MinHeap()***
  Constructor method to initialize the ***heapArray*** as empty and ***heapSize*** as 0

- ***void insert(Building building)***
  Adds a building (with the initial executed time set to Max Integer) after the heap array's last element and then decreases its main key, i.e., the building's executed time to 0 by calling ***decreaseKey(heapSize++ , 0)*** where ***heapsize*** represents the position of the newly inserted element.

- ***void decreaseKey(int i, int key)***
  Decreases the key of the element at position *"i"* and makes relevant updates (swaps), moving the element up the heap. This is done by exchanging the building element with its parent element in the heap while the parent is larger than the current element. Main key here is the current executed time of the building, but the ***compareTo()*** method used here to compare two buildings also checks the building number in case of any ties. This maintains the min-heap property.

- ***void increaseKey(int i, int key)***
  Sets the **i**th element's key, i.e., the building's execution time to the value provided as parameter. Calls the ***heapify(i)*** method to ensure that the subtree rooted at position ***i*** satisfies the min-heap property and restores the same if it has been violated by increasing the element's key.

- ***void heapifyEntire()***
  This method is invoked when the min-heap may be violated at more than one positions. Builds the entire array into a min heap. This is done by calling the ***heapify(i)*** method on the range (heapSIze/2, heapSIze/2 -1 …1, 0).

- ***void heapify(int pos)***
  Restores the min-heap property at index ***pos.*** Its assumes that the subtrees rooter at ***left(pos) and right(pos)*** are min-heaps. When array[pos] is smaller than its children the method makes it float down such that the min-heap property is maintained.

- ***Building peek()***
  Returns the minimum element in the heap, without removing it.

- ***void swap(int i1, int i2).***
  Helper function to swap two heap elements at indices i1 and i2.

- ***void parentOf (int i)***
  Helper function to calculate index of the parent of ith element.

- ***void rightChildOf (int i)***
  Helper function to calculate index of the right child of ith element.

- ***void leftChildOf (int i)***
  Helper function to calculate index of the left child of ith element.

3. <u>Node (inner class)</u>

This is the class for red-black tree node elements, implemented as a nested inner class in the parent ***RedBlackTree*** class.

Contains the following fields:

- ***Node left***: Reference to the left child of the current node
- ***Node right***: Reference to the right child of the current node
- ***Node parent***: Reference to the parent of the current node
- ***int color***: Refers to the current node, can be black (0) or red (1)
- ***Building building***: Reference to the Building object contained in the current node

Methods:

- ***int compareTo(Node o)***
  Helper method to compare two nodes with respect to their building element, in essence comparing the building number of those elements. Returns a negative value if the this building is smaller than the building compare and positive value otherwise.

4. *RedBlackTree*

This is the class implements the red-black tree data structure and various operations on the tree.

Contains the following fields:

- ***Node root***: Points to the root node of the tree
- ***Node nil***: A nil node which represents sentinel node. This helps to find out where we fall of the tree when searching on checking children data. The color of this node always remains black. Used to represent all NILs, i.e., all the leaves and the root's parent.
- ***Node:*** Inner class for the node element.

Methods:

- ***void insert(Building buildingData)***
  Creates a new node with the building data and inserts into the red-black tree. The insertion is similar to the BST insertion but it may mess the red-black tree property of no two parent-child nodes being red, and the ***rebalance()*** method is called to fix that.

- ***void rebalance(Node node)***
  Called on the node where the red-black tree is being violated. Fixes the tree in either of the following cases by invoking ***rightRotate*** and/or ***leftRotate***.
  - LXr (where X can be L or R): Move the unbalance up the tree and continue rebalancing on the grand parent of the current node.
  - LLb: Perform a single left rotation to resolve the imbalance and continue rebalancing.
  - LRb: Perform a right rotation and then a left rotation.
  - RXr (where X can be L or R): Move the unbalance up the tree and continue rebalancing on the grand parent of the current node.
  - RRb: Perform a single right rotation to resolve the imbalance and continue rebalancing.
  - RLb: Perform a left rotation and then a right rotation.

- ***rightRotate(Node node)***
  Makes the left child of Node the root of the subtree rooted at ***node*** and makes other relevant pointer and color changes along the process.

- ***delete(Building building)***
- Wrapper function to delete the node with the building element. Calls ***delete(Node nodeToBeDeleted)*** for the actual deletion.

- ***delete(Node nodeToBeDeleted)***
  Actual function to perform deletion in a red black tree. Since it is a type of BST, for the deletion of a node with one or more children, we find an element to replace the node to be deleted and transplant it by calling the method ***transplantNode***. This is the child of the node to be deleted if has a single child or the minimum in its right subtree.
  Calls ***deleteRebalance()*** to resolve any imbalance as a result of the delete operation if the deleted node was black and returns true on successful deletion or false when the element is not found in the tree.

- ***deleteRebalance(Node x)***
  Called on the node where the red-black tree property is being violated. Fixes the tree in either of the cases by invoking the method(s) ***rightRotate*** and/or ***leftRotate***.

- ***void transplantNode(Node u, Node v)***
  Places the subtree rooted at node u with the subtree rooted at node v; node u's parent becomes node v's parent, and u's parent ends up having v as its appropriate child.

- ***Node findMinimumInSubtree(Node subTreeRoot)***
  Helper function to find the minimum in a subtree by following left child pointers.

- ***Node search(Node nodeToSearch)***
  Search method works as in a BST: going to the left child if the nodeToSearch is smaller than the current node or going to the right child if larger or else if return current node if both are equal. Returns null when node is not found in the tree

- ***String printBuilding(int buildingNum)***
  Given the buildingNum, this function to search and output the building data from the corresponding node element or returns (0,0,0) if the building is not found.

- ***String printRange(int b1, int b2)***
  Wrapper function to output buildings in the given range or return (0,0,0) if the building is not found in the given range.

- ***StringBuilder printRange(Node node, int b1, int b2, StringBuilder builder)***
  Recursive function to build output of printing the buildings in the given range initially called from root node and an empty stringbuilder. Searches the range, going into the left and right subtree only when their value is less than b1 and greater than b2 respectively.

5. <u>risingCity (main class)</u>

This is the main class and contains the logic to process input commands, pick buildings for construction and generate the relevant output.

Contains the following fields:

- ***MinHeap heap***: Reference to min-heap data structure used in the implementation
- ***RedBlackTree rbTree***: Reference to min-heap data structure used in the implementation
- ***int globalTime***: Counter to maintain the global time
- ***PrintWriter writer***: A Writer object to write the output generated to the desired text file
- ***ArrayList<Building> pendingHeapInserts***: An array list of type ***Building*** that helps maintain a list of buildings which have been added to out process and red-black tree but not to the min-heap.

Methods:

- ***public static void main(String[] args)***
  The default main function where the code execution starts from. Gets the input file name from the command line argument ***args[0],*** creates a Scanner object to read that input file and passes it on to the ***process()*** function.

- ***void process(Scanner sc)***
  The execution of all business logic operations related to the project happen in this function.
  It takes a scanner class as input, to help read the input file, line by line. The global time is maintained by the ***globalTime*** variable.
  The next command to process is read from the input file but not processed until its time matches the global timer. And the while loop in line 44 executes only when the command which was read last is executed and there is a next line to read from the input. This allows us to proceed without preprocessing and storing the input text commands.
  To work on a building there must one of the following two cases, and the two ***if*** conditions in lines 49 and 68 take care of that:
  1. There is no current building under construction and the heap is not empty. We pick a new building to work on and increase its execution time and reset the ***workOnLastBuilding*** to *false.*
  2. There is a building which is currently being constructed and its 5 days haven't been completed. The flag ***workOnLastBuilding*** helps to keep a check on this condition. In such as case we work on, we fall into the else if part of *line 68* and its execution time is increased.

  Common to both the cases, we check if the building's execution time has matched its total construction time and if true, we remove it from the min-heap and the red-black tree, output its completed time and move on to the next iteration. We also increase the global timer inside these if cases.
  We also to handle the scenario where the print command is encountered on the same day when the building construction completes. Here, we want to execute the ***print*** command before deletion from the data structures and the lines numbers 62 and 77 take care of that after increasing the **globalTime** counter.

  Coming out of these if conditions, if the **globalTime** hasn't been incremented we do so in line number 88.

- ***boolean processCommand(String command, boolean isLastPickedBuildingWorked)***
  This function processes the input command. The command given to it as an argument represents a single line of the input, which is parsed and executed based on the type of command which fall into one of the following categories:
  - Insert -> **insertBuilding()** method is invoked to insert the building data into the min-heap and the red-black tree.

- PrintBuilding -> When a single building number is given, the method ***String printBuilding(int buildingNum)*** from the ***RedBlackTree*** class is called and the output from this method is given as an argument to the ***void printOutput(String output)*** method for printing it to the specified output file.
- PrintBuilding -> When a range is given, the method ***String printRange(int b1, int b2)*** from the ***RedBlackTree*** class is called and the output from this method is given as an argument to the ***void printOutput(String output)*** method for printing it to the specified output file.

- ***void insertBuilding(int buildingNum, int totalTime, boolean isLastPickedBuildingWorked)***
  Inserts the building element into the red-black tree; but its added to the min heap if a building not being worked upon, i.e., if the construction of a building is in process new buildings to be inserted are not immediately added to the heap. This would mess the heap balance, and thus we add those buildings to a list of pending buildings, ***pendingHeapInserts,*** which is processed the heap only when the current building being constructed is done or 5 days are up (whichever is earlier)).

- ***void printOutput(String output)***
  Writes the string given to it as a parameter to the specified output text file. It uses the PrintWriter object initialized in the constructor.

- ***void completePendingHeapInserts ()***
  Helper function to insert (if any) pending building inputs to the min heap. Called when the construction of the previous building is completed. Adds the buildings in the list ***pendingHeapInserts*** and clears it for further iterations.

- ***Scanner readFile(String fileName)***
  Helper function to create a Scanner object for the input file.

6. **Constants**

   A class to maintain final constant values, which may be used across the project and whose values don't change during execution. Added the color values for the red-black tree here for easy change when needed.
   Fields:
   - final int RED = 0
   - final int BLACK = 1