COP5536: Advanced Data Structures, Fall 2019

Project Report

Shashank Chandrashekar Murigappa (UFID: 3701-3155)
smurigappa@ufl.edu

Project Description:

Wayne Enterprises is developing a new city. They are constructing many buildings and plan to use this software to keep track of all buildings under construction in this new city. The input file has a sequence of commands, including Insert, Print and PrintBuilding, When we get an insert, it means, we have a new building to construct, Print prints single building properties, if currently being worked on and and PrintBuilding prints all the buildings properties that are currently being worked on, retrieved from the Red black tree. We need to work on a building for 5 days. After that, we pick the building with least executed time and work on it for the next 5 days and so on until all the buildings are constructed and the city is complete. If we have 2 buildings with the same executed time, we break the tie using the building number. The building number is a unique key and each time a new building is inserted into the heap, its executed time would be 0.

Assumptions:

PrintBuilding()- definitely has 2 parameters. (range)

Print()- definitely has only one parameter. (single building)

Solution:

We start off by reading the first line and if it is insert, we simply insert it into the heap. We keep 2 global counters, local and global. Local counter keeps track of the number of days a building has been worked on after it has been picked the current time. The global counter keeps track of the number of days passed.

We start reading the input line by line. At each line, we decipher the command and if it is an insert, we store it in a buffer until the local counter value becomes 5. This means that we have a new insert, but we are already working on another building for which we have not completed 5 days of construction.

Each day, we call the UpdateExecutedTime function which increments the current minimum executed time building one day at a time. If the executed time becomes equal to total time for construction, ewe reset the local counter and start again by picking the next min element. Once the local counter becomes 5, we heapify and insert all the elements in the buffer into the minheap with executed time as 0. Then, we pick the next element with least executed time. Since we insert more than one element into the heap at a stretch, we might get multiple elements with same executed times, we handle all these cases to break ties in the Heapify function.

If print is found as a command, we call the search method of red black tree to locate the building and print it's values. If not found, we simply print (0,0,0). For the printBuilding, we call the printElements in range function in the red black tree, which calls a recursive function to print all values in the range. After input file is completely read, if we are still working on a building, we finish it for 5 days. Then we call the final update which runs on the existing elements in the heap updating their executed times and heapifying every 5 days until all the elements in the heap are removed which marks the completion of the city.

Programming Environment:

Code is written on Eclipse IDE using JAVA as a programming language.

Program Structure:

The project has four classes in total, "Building.java", "MinHeap.java", "RedBlackTree.java", "risingCity.java", where "risingCity.java" is the class that holds the Main method.

Building.java

This class describes the structure of a building which also acts as a node in a Red-Black Tree. It contains the following class variables:

- *public static int globalCounter* – Used to keep track of number of days passed.
- *public static int localCounter* – Used to keep track of number of days a building has been worked on.
- *protected int BuildingNum* – Stores the Building Number of a building inserted which is used as an ID for the red-black tree.
- *protected int executed_time* - Stores the number of days the building has been worked on.
- *protected int total_time* - Stores the time taken to complete the construction of the building.
- *protected Building left* – Pointer to the left child in RB-tree.
- *protected Building right* – Pointer to the right child in RB-tree.
- *protected Building parent* – Pointer to the parent in RB-tree.
- *protected int color* – Color Property of the RB-Node (0 represents 'black' and 1 represents 'red').

This class does not contain a constructor since the building values are being initialized in MinHeap and RedBlackTree classes.

MinHeap.java

The class MinHeap is where all the Heap operations have been defined. The operations of this class are performed on an array, which is defined in the same class.

The properties defined are as follows:

private Building[] heap – Array used to store the heap objects (in our case, buildings).

private int size – Stores the size of the MinHeap.

The methods defined in the class are given below:

public Building Insert (int, int) – public method that calls the private insert

private Building insert (int, int) – Takes 2 parameters, the building number and total time and initializes a building with executed time as 0 and inserts into the heap. It also restores the heap structure after the insert.

public void removeBuilding (RedBlackTree, int) – public method that calls private remove.

private void remove (RedBlackTree, int) – Takes 2 parameters, calls remove min and delete operation on Red-Black Tree for the int parameter. It also resets the local counter.

public Building getMin() – return the minimum element in the heap, which is heap[0]

public void removeMin() – returns minimum element from the heap and calls heapify() to restore the heap structure.

public int UpdateExecTime (RedBlackTree) – public method that calls private updateExecTime()

private int updateExecTime (RedBlackTree) – Updates execution time of the current building being worked on one day at a time. If a building is completed, it simply returns the building to be removed, else returns a negative value.

public void final_update (RedBlackTree) – public method that calls private FinalUpdate().

private void FinalUpdate (RedBlackTree) – This method is called when the input file is completely read, but all the buildings are not constructed yet. We work on the remaining buildings updating every 5 days.

private void Heapify(int) – Public method that calls private heapify(int).

public void heapify(int) – This method takes in the index value where the node from where we are supposed to heapify. For all nodes other than leaves, we compare it with it's left and right child.

Here we check a few cases:

- If right child exists and if it is smaller than the left child, we start swapping the parent node with its right child.
- If the right child is equal to the left, we pick the one with smallest building number and compare it's executed time with parent. If child has smaller executed time, simply swap. If child with smallest building number also has the same executed time as the parent, we compare the building numbers of that child and the parent. If required, we swap.
- If right child does not exist or it is larger than the left child, we swap with the parent if its executed time is smaller than its parent. If they are equal, we again compare the building numbers of the left child and its parent and swap if necessary.

private void swap(int, int) – Swaps 2 building objects.

This class also has a constructor that takes the maximum size of the Heap as parameter and creates instances of that many building objects.

RedBlackTree.java

The class RedBlackTree is where all the operations of a Red Black Tree have been defined. The operations of this class are performed on instances of the building object.

The methods defined in the class are given below:

private Building searchTree(Building, int) - private method that takes the root of RB-tree and the building number as parameters and recursively finds the building in the RB-tree.

private void restoreUponDelete(Building) – Rearranges the RB-tree after deletion to ensure the properties of RB-tree are not violated.

private void swapNodes(Building, Building) – Takes 2 building objects as parameters and swaps them.

private void delete(Building, int) – Takes the root and building number to be deleted as parameters and deletes the building from the RB-Tree. It calls the RestoreUponDelete method to restructure the tree.

private void RestoreUponInsert(Building k) - Rearranges the RB-tree after insertion to ensure the properties of RB-tree are not violated.

public Building search(int buildingNumber) – public search method that calls the private searchTree method.

public Building minimum(Building building) – public method that finds the building with the minimum buildingNum.

public Building maximum(Building building) - public method that finds the building with the maximum buildingNum.

public Building successor(Building) - finds the successor of the given building.

public Building predecessor(Building) - finds the predecessor of the given building

public void rotateLeft(Building) - rotates building node given to the left.

public void rotateRight(Building) - rotates building node given to the right.

public void insert(Building) - Inserts the building to the tree in its appropriate position according to its building number and restores RB-properties to the tree by calling the RestoreUponInsert method.
public void deleteBuilding(int) – public method that calls the delete method.

public void printBuildingsInRange(int, int) - public method that calls buildingsInRange method by passing the root node and a String to retrieve the output line from the function, so as to remove the last comma inserted. It takes the minimum and maximum building numbers as parameters and prints the buildings in the range.

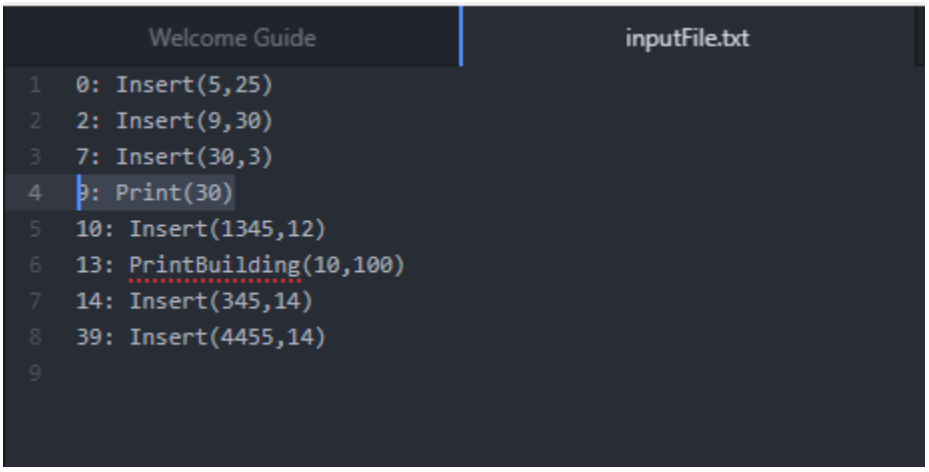
private void buildingsInRange(Building, StringBuilder, int, int) – It is a recursive function that finds all buildings in the given range and appends its properties to the string builder. Since we are passing the StringBuilder object, it would automatically be updated without having to return it.

risingCity.java

This is the class that contains the main method. It is the entry point of the program.

public static void main() – In this method, we read the input file using the BufferedReader and after reading each line, we split the string to retrieve parameters of insert and print commands. We call the methods accordingly. We also redirect our system out stream to a file by creating a new one if it doesn't exist. Once the file is completely read, we close the buffer and call the final Update method to process the buildings still in the heap.

Sample Input:



Sample Output: