COP5536 - Advanced Data Structures

Fall 2019

Assignment Project

RisingCity

Project Report

Name: Shaishav Harshadbhai Shah

Email: shah.sh@ufl.edu

UF ID: 1136-3317

Index:

Class Definitions with Function Prototypes

Building.java - A pojo class for storing building details.
Properties:
- buildingNumber - unique integer identifier for each building
- executedTime - total number of days spent so far working on this building
- totalTime - total number of days needed to complete the construction of the building

Methods:
- Contains getter and setter methods for the above mentioned properties.
- toString() - Returns formatted string to be printed in the output file. e.g (1,1,1)

HeapNode.java - A pojo class for storing heap node details. It inherits Building class.
Properties:
- rbtNodeReference - Pointer to the corresponding RedBlackNode instance.

Methods:
- Contains getter and setter methods for rbtNodeReference.

NodeColor.java - An enum class containing color options for RedBlackNode instance.
Color option constants: RED, BLACK

RedBlackNode.java - A pojo class for storing red black node details. It inherits Building class.
Properties:
- NIL - Represents null external black node of a red black tree.
- left - Pointer to the left child node. Set to NIL initially in constructor.
- right - Pointer to the right child node. Set to NIL initially in constructor.
- parent - Pointer to the parent node. Set to NIL initially in constructor.
- color - Represents node color with NodeColor enum. Set to red in constructor initially when a node is inserted.
- heapNodeReference - Pointer to the corresponding HeapNode instance.

Methods:
- Contains getter and setter methods for the above mentioned properties.
- toString() - Returns formatted string to be printed in the output file. e.g (1,1,1)

RedBlackNode.java - A pojo class for storing red black node details. It inherits Building class.
Properties:
- NIL - Represents null external black node of a red black tree. static and final variable.
- left - Pointer to the left child node. Set to NIL initially in constructor.
- right - Pointer to the right child node. Set to NIL initially in constructor.
- parent - Pointer to the parent node. Set to NIL initially in constructor.
- color - Represents node color with NodeColor enum. Set to red in constructor initially when a node is inserted.
- heapNodeReference - Pointer to the corresponding HeapNode instance.

Methods:
- Contains getter and setter methods for the above mentioned properties.
- toString() - Returns formatted string to be printed in the output file. e.g (1,1,1)

TestCommand.java - An enum class containing all the different types of operations required to be performed by the program.
Operation constants: INSERT, PRINT, PRINTBUILDING

TestCase.java - A generic class containing information regarding different types of test case operations.

Properties:

- testCommand - Contains enum value representing the type of operation to be performed.
- inputTime - time when the test case has to be executed.
- buildingId - Unique integer identifier for the building.
- totalConstructionTime - Total time required for constructing the entire building.
- startBuildingNum, endBuildingNum - Range of building numbers to be printed. endBuildingNum would be 0 in case if only one building has to be printed using the PRINT operation.

Methods:

- Contains getter and setter methods for the above mentioned properties.

InputParser.java - A utility class for reading test case data from the input and creating TestCase object from it.

Static Method:

- TestCase getParsedTestCase(String testCaseStr) - Parses the input file data string given by testCaseStr parameter and converts it into into TestCase object using regular expression and returns the TestCase instance.

OutputParser.java - A utility class that collects and writes the output required to be printed to the specified output file.

Properties:

- output - A static StringBuilder instance that contains the entire text to be printed in the output file. We'll append all output strings into this StringBuilder instance until City development is running and will make only one file write at the end when City development is completed. This will keep the connection to be open only when needed and also limits the frequency of costly write operation to 1 only.
- emptyNodeTuple - A final String representing value of empty node tuple which is to be printed when the requested building in Print command is not found. Value: "(0,0,0)".
- outputFilename - A final String representing value of output file name. Value: "output_file.txt".

Static Method:

- addBuilding(RedBlackNode node) - Appends the details of the building to be printed to the output when Print operation is to be performed. Takes as it's input parameter.
- addFinishedBuilding(RedBlackNode node, int finishTime) - Appends the details of the building to be printed to the output when building construction finishes. Takes node whose details to be printed and the day when it finishes as it's input parameter.
- addMultipleBuildings(List<RedBlackNode> nodes) - Appends the details of the buildings to be printed to the output when PrintBuilding operation is to be performed. Takes all the nodes whose details to be printed as it's input parameter.
- addErrorMessage(String error) - Appends the error message to the output when any error occurs. Takes error message to be printed as it's input parameter.
- print() - Writes the final output to the output file.

MinHeap.java - A class representing and containing all the functionalities of min heap data structure. This min heap is ordered by executed time of the building.

Properties:

- heap - Array of HeapNode.
- capacity - Maximum capacity for the heap array.
- size - Number of currently inserted items into the heap array.
- TOP - final integer representing index of the top-most item of heap array.

Constructor:

- MinHeap(int capacity) - Initializes the heap array with the given capacity. Here we'll put null at 0th index of the array to simplify our parent and children index calculations and increment size by 1.

Methods:

- isEmpty() - Return true if the heap is empty, false otherwise.
- hasParent(int position) - Returns true if the given position has parent, false otherwise.
- isLeaf(int position) - Returns true if the item at the given position is a leaf node, false otherwise.
- getParentPosition(int position) - Returns the index of the parent of the given position.
- getLeftChildPosition(int position) - Returns the index of the left child of the given position.
- getRightChildPosition(int position) - Returns the index of the right child of the given position.
- swap(int i, int j) - Swap the items at index i and j in the heap array.
- insert(HeapNode node) - Inserts the given node into the heap array at the end. Throws exception if array is already full. Calls heapifyUp operation to rebalance the heap property after insertion.
- heapifyUp(int position) - Rearranges the heap array as per heap property whenever a new item is inserted by performing swap operations whenever necessary. Increments the size by one at the end.
- extractMin() - Removes and returns the minimum item from the heap. Throws exception is the heap array is empty. In min heap the item with the minimum value will always be at the top, so it removes it, copies the item at the last index to the top, calls heapifyDown operation to rebalance top, removes last item, and then decrements the size by 1. It returns the min item at the end.
- heapifyDown(int position) - Set the node at the given position to it's appropriate index by recursively performing heapify operation.

RedBlackTree.java - A class representing and containing all the functionalities of red black tree data structure. This red black tree is ordered by building number.

Properties:

- root - Root node of the red black tree.
- nil - Null external black node.

Constructor:

- RedBlackTree() - Initializes the nil to reference the static final variable RedBlackNode.NIL. Initialize root node to reference nil and set left and right child of root node to nil.

Methods:

- isLeftChild(RedBlackNode node) - Return true if the node is left child of its parent, false otherwise.
- isRightChild(RedBlackNode node) - Return true if the node is right child of its parent, false otherwise.
- updateParentChildLink(RedBlackNode parent, RedBlackNode oldChild, RedBlackNode newChild) - Updates child link for the given parent node, where oldChild is the current child which is to be replaced by newChild. It also sets parent as the new parent of the newChild.
- rotateLeft(RedBlackNode node) - Executes left rotation on the given node.
- rotateRight(RedBlackNode node) - Executes right rotation on the given node.
- insert(RedBlackNode node) - Adds the given node into the tree. Throws an exception if the node with same building number already exists. Node is added as per binary search tree rules, and then rebalanced to maintain red black tree property.
- insertionRebalance(RedBlackNode node) - Re-establishes the RBT property if there are two consecutive red nodes after insertion. Called after every insert operation. It performs

necessary color change or rotation operations to rebalance as per red black tree insertion fix rules.
- delete(RedBlackNode node) - Removes the given node from the tree. Calls deletionRebalance to rebalance the tree after removal.
- deletionRebalance(RedBlackNode node) - Re-establishes the RBT property if a black node is deleted. It performs necessary rotation operations to rebalance as per red black tree deletion fix rules.
- getMinimumNode(RedBlackNode node) - Returns the node with min value in the subtree rooted at node. Used in finding successor node in deletion operation.
- search(int buildingNumber) - Finds and returns a node with buildingNumber equal to the given buildingNumber by using recursive binary search.
- searchInRange(int startBuildingNumber, int endBuildingNumber) - Find and returns a list of nodes for which following condition "startBuildingNumber <= node.buildingNumber <= endBuildingNumber" satisfies. Uses recursive binary search.

risingCity.java - An entry class of the application containing main method.
Properties:
- testCases - A queue object containing all the test case operations to be performed read from the input test file.

Methods:
- main(String[] args) - Entry point of the application. Expects name of the input file as the first argument in the command line arguments array args[]. The idea is to read the entire input file at once and store the input test case data into a Queue, so the file does not remain open throughout the life of a program. Queue makes sense here because the input is sorted according to the input time. Hence, we read one line at a time from the input file, parses it by calling InputParses.getParsedTestCase(), and adds it into the testCases Queue. After reading all the input data, we instantiate a CityBuilder object and pass our testCases queue to it for execution, and then we'll call CityBuilder.build() method to start city building process.

CityBuilder.java - A driver class of the application to execute city building process.
Properties:
- testCases - A queue object containing all the test case operations to be performed.
- presentDay - Global counter representing the present day starting from 0.
- minHeap - MinHeap instance.
- currentBuilding - HeapNode instance representing the current building we are working upon.
- redBlackTree - RedBlackTree instance.
- nodeToBeDeleted - RedBlackNode instance representing the node to be deleted after building construction has been finished for that node.
- MAX_DAYS_TO_WORK - static final integer representing the maximum number of days allowed to be worked upon one building at a time. Set to 5.
- MAX_BUILDINGS - static final integer representing the maximum number of buildings allowed to be inserted into the data structures. Set to 2000.
- daysLeftToWorkInSession - The idea is to use session to maintain how many days we should work on a building before moving onto the next. This represents the number of days left to be worked in a particular session on currentBuilding.

Constructor:
- CityBuilder(Queue<TestCase> testCases) - Initializes MinHeap and RedBlackTree instances.

Methods:
- build() - Starts city building process. The core logic lies here.

- finish() - Called when either City building has been finished or any error has occured to print the output to the file and stop program execution.
- insertBuilding(int buildingNumber, int executedTime, int totalTime) - Inserts new building to be worked in Min heap and Red black tree. It also sets corresponding node reference in min heap and rbt.

Program flow explained

Below is the explanation of core program logic that resides in build() method. The variables used here are explained in the above explanation of CityBuilder.java.

```
// Work until either there are requests to be processed from queue or there are buildings left to be built Or if there's currentBuilding to work upon. Would be not-null in case of last building.
While (testCasesQueue != empty OR minHeap != empty OR currentBuilding != null)
	// Checks if there are test case operations left to be performed.
	If (testCasesQueue != empty)
		// Peek the top element of queue.
		T = testCasesQueue.peek()

		// Work request can only be taken when the input time of the request matches the present day.
		If (T.inputTime == presentDay)
		// Execute whatever operation needs to be performed given by T.testCommand

		// Remove from queue as the operation has been executed.
		testCasesQueue.remove();

		// Check if there is any finished building available.
		If (nodeToBeDeleted != null)
			// Add finished building to the output.

			// Delete node from the red black tree.
			redBlackTree.delete(nodeToBeDeleted)

			// Reset nodeToBeDeleted to null.
			nodeToBeDeleted = null

		// Check if last building construction session is finished.
		If (currentBuilding == null)
			// If min heap contains any buildings to be built.
			If (minHeap != empty)
				// Extract min.
				currentBuilding = minHeap.extractMin()

				// Update daysLeftToWorkInSession. It will be a minimum of workLeft and MAX_DAYS_TO_WORK.
				workLeft = currentBuilding.totalTime - currentBuilding.executedTime
				daysLeftToWorkInSession = Math.min(workLeft, MAX_DAYS_TO_WORK)
			Else // Else there are no more buildings left to be built.
				// Just increment the present day global counter
				presentDay++
				Continue

		// Update executedTime in our data structures for min-heap and rbt.
		currentBuilding.executedTime++

		presentDay++ // Update present day counter.

		// Check if building construction is finished.
		If (currentBuilding.executedTime == currentBuilding.totalTime)
			// We defer node deletion from RBT to avoid mishandling the scenario when
			// next input comes for the same building on immediate next day.
```

```
                // Example: (8,0,10) finishing on day 40 and on the same day there's a print
                // command for the same building. i.e 40: Print(8)
                nodeToBeDeleted = currentBuilding.getRbtReference()

        // Decrement daysLeftToWorkInSession
        daysLeftToWorkInSession--

        // If work for current session is done, then add it again to the min-heap if it has
        // not been completed yet.
        If (daysLeftToWorkInSession == 0)
                // Check if construction is not finished.
                If (currentBuilding.executedTime != currentBuilding.totalTime)
                        // Re-insert with updated executed time.
                        minHeap.insert(currentBuilding)

                // Reset currentBuilding to null.
                currentBuilding = null

End While

// Final city building node.
If (nodeToBeDeleted != null)
        // Add finished building to the output.

        // Delete node from the red black tree.
        redBlackTree.delete(nodeToBeDeleted)

        // Reset nodeToBeDeleted to null.
        nodeToBeDeleted = null


// Finish execution. Print output to the file.
finish()
```