Advanced Data Structure

Project Overview:

Table of Contents

Introduction…………………………………………………………………………………..2 Associated File ………………………………………………………………………………2 Program Structure………………………………………………………………………….4

Compiler Information……………………………………………………………………..5 Results Observation………………………………………………………………………..6 Conclusion……………………………………………….……………………………………..6 References…………………………………...………………………………………………..6

**Introduction**

The assignment is implemented in java programming language. The main aim of this assignment was to implement an event counter. The event counter had two fields ID and count where each count is number of active events with given ID. The event counter was to be implemented using Red Black Tree. Red black tree is a type of Balanced Binary Search tree. But it differs from Balanced Binary Search tree in a way that each node in red black tree contains an extra bit of information. This extra bit of information is the color associated with each node. The properties of red black tree are as follows:

1. A node is either red or black.
2. The root of tree is black.
3. All Leaves are black.
4. Children of red node are black

**Associated files**

The file in which the program is implemented is bbst.java.

**bbst.java:** The file contains the main function as well as all the required methods required toimplement the event counter.

**class bbst:** This class contains the main method which take the input files from the user throughcommand line and console and the outputs to another file. The main function of this class is to implement the red black tree that implements the event counter. This class runs all the methods which will execute the different commands of event counter. The list of functions in this class are as follow:

* **Red\_Black\_Node**: This is a private inner class required to implement the node of the redblack tree. It is implemented as an inner class so as not to expose the details of the node. It has a constructor Red\_Black\_Node (int ID, Int count) to construct the node i.e. the event upon receiving its ID and count.
* **Main ():** Main method is used to take the input from the user and execute the program.Upon receiving the first file containing all the sorted data of IDs and counter, It stores the data into two arrays arr1 and arr2 storing ID and counts in them respectively.it calls the Initial\_RBT\_Creation function to create the Red Black Tree.
* **Initial\_RBT\_Creation ():** This function is used to create the binary balanced tree. It takesinput from two arrays and recursively creates the binary balanced tree. After the binary balanced tree is created it calls the BST\_TO\_RBT function.

* **BST\_TO\_RBT ():** This function assigns color to the tree which Is created. It assigns blackcolor to the root and then recursively colors the children node.
* **Count ():** This function is called when the user gives the command Count(theID).Thisfunction the passes theID to RBT\_Count and prints the count of theID returned by RBT\_Count function.
* **RBT\_Count ():** This function returns the node with specified ID. It takes as input the rootnode of the tree and theID whose count is to be returned. It traverses the tree till node with ID which is equal to theID is found and then returns that node.
* **InRange ():** This function is called when the user gives the command InRange (ID1, ID2).This function the passes both ID1 and ID2 as well as root node of tree to RBT\_InRange and prints the total count between the range.
* **RBT\_InRange ():** This function is called by InRange function. It finds the closest node toboth the IDs given (for left range it finds node with same ID or smallest ID which is greater than the Left ID and right range it finds node with same ID or Greatest ID which is smaller than given right ID) by calling the ClosestTotheID () function and then counts the total number of counts between the 2 Ids.
* **ClosestTotheID ():** This function finds the node with ID which is equal to the input theIDor node with ID which is closest to theID if node with equal ID is not present. It recursively traverses the tree till it reaches the required node.
* **Increase ():** This function calls the RBT\_Count function. Once the node is returned by theRBT\_Count, the Increase function adds the new count to available count and prints the new count. If the node with ID is not present It calls Red\_Black\_Insert () to insert the node with ID and count in given tree.
* **Red\_Black\_Insert ():** This function inserts the the node in given tree. It then callsRed\_Black\_Insert\_Fix function to fix the tree so that it satisfies all properties of red black tree.
* **Red\_Black\_Insert\_Fix ():** This function is to call recolor nodes and perform rotations sothat all the properties of red black tree are maintained after insertion. It may call Left\_Rotate and Right\_Rotate function for rotation of nodes.
* **Left\_Rotate ():** This function is called to change the configuration of two nodes fromright into left. These functions are to satisfy the properties of red black tree.
* **Right\_Rotate ():** This function is called to change the configuration of two nodes fromleft into right. These functions are to satisfy the properties of red black tree.
* **Reduce ():** This function calls the RBT\_Count function. Once the node is returned by theRBT\_Count, the Increase function subtracts the new count from available count and prints the new count. If the count of node with ID reduces less than 0, It calls Red\_Black\_Delete () to delete the node with ID and count in given tree.
* **Red\_Black\_Delete ():** This function deletes the the node in given tree. It then callsRed\_Black\_Delete\_Fix function to fix the tree so that it satisfies all properties of red black tree.
* **Red\_Black\_Delete\_Fixup ():** This function is to call recolor nodes and perform rotationsso that all the properties of red black tree are maintained after insertion. It may call Left\_Rotate and Right\_Rotate function for rotation of nodes.

* **Red\_Black\_Transplant ():** This function is called to called to transplant the the twonodes in a given tree. It is called by Red\_Black\_Delete\_Fixup to satisfy the properties of red black tree if there is any violation.
* **Next ():** This function is to find the next of the given theID i.e. node with smallest IDgreater then given theID. It first calls ClosestTotheID. If the returned node’s ID is greater than given theID, it prints the ID else it calls the RBT\_Successor () function to find the next node.
* **Previous ():** This function is to find the previous of the given theID i.e. node withgreatest ID smaller then given theID. It first calls ClosestTotheID. If the returned node’s ID is smaller than given theID, it prints the ID else it calls the RBT\_Predecessor () function to find the next node.
* **RBT\_Successor ():** This function finds the Successor of the given node i.e. node withsmallest ID greater then given node's ID.
* **RBT\_Predecessor ():** This function finds the predecessor of the given node i.e. nodewith greatest ID smaller then given node's ID.
* **RBT\_Tree\_Minimum ():** This function finds the node with minimum ID in a given subtree.
* **RBT\_Tree\_Maximum ():** This function finds the node with maximum ID in a given subtree.

**Program Structure**

When project is compiled and executed it takes as input, the input file as well as the commands. From the input file, two array are created. Then the **Initial\_RBT\_Creation** function is called which creates the Red Black Trees. Then depending upon the commands, that respective function is called.

If the command is:

* **count(theID):** This calls the method **count(theID),** which in turns passes theID and rootof tree to **RBT\_Count**. The RBT\_Count then executes and returns the the node, whose ID is equal to given theID, back to count method. The count method than prints count of that node. If node with ID is not present it prints 0.
* **inrange (ID1, ID2):** This calls the method **InRange (ID1, ID2)**, which in turns passesID1and ID2 and root of tree to **RBT\_InRange**. The RBT\_InRange then executes and returns the the total of counts between the range.
* **increase (theID, m):** This calls the method **Increase (theID ,m)** which in turns passestheID and root of tree to **RBT\_Count**. The RBT\_Count then executes and returns the the node, whose ID is equal to given theID, back to increase method. The Increase method

than adds m to the count and prints new count of that node. If node with ID is not present it calls **Red\_Black\_Insert** function to insert the node in tree.

* **reduce (theID, m):** This calls the method **Reduce (theID, m)** which in turns passes theIDand root of tree to **RBT\_Count**. The RBT\_Count then executes and returns the the node, whose ID is equal to given theID, back to reduce method. The reduce method than subtracts m to the count and prints new count of that node. If node with ID is not present it prints 0. If the subtraction of m causes the count to go below 0, then it calls

**Red\_Black\_Delete** to delete the node.

* **next(theID):** This calls the method **Next (theID)** which in turns passes theID and root oftree to **ClosestTotheID**. The **ClosestTotheID** then executes and returns the node, whose ID is equal to given theID and if no such node exists, it returns the node whose ID is closest in difference to theID, back to next method. The next method than checks If the returned node’s ID is greater than given theID, if its than, it prints the ID else it calls the

**RBT\_Successor** function to find the next node.

* **previous(theID):** This calls the method **Previous (theID)** which in turns passes theID androot of tree to **ClosestTotheID**. The **ClosestTotheID** then executes and returns the node, whose ID is equal to given theID and if no such node exists, it returns the node whose ID is closest in difference to theID, back to previous method. The previous method than checks If the returned node’s ID is less than given theID, if its than, it prints the ID else it calls the **RBT\_Predecessor** function to find the previous node.

**Compiler Information**

This assignment has been compiled and tested under following platforms:

|  |  |  |
| --- | --- | --- |
| Operating System | Compiler | Test Result |
| Windows 7 | IntelliJ IDEA | Pass |
| OS X | javac | Pass |
| OS X | Eclipse IDE | Pass |

Steps to Execute using IDE:

1. Open the file bbst.java and compile the project.
2. Build the project
3. Input in console present in IDE.

Steps to Execute using javac without makefile:

1. Copy the java file in a separate directory
2. Copy the input and command file that have to be input by the user into the same directory
3. Type in javac bbst.java
4. java bbst input.txt<commands.txt>output.txt to print in output file

Steps to Execute using javac with makefile:

1. Copy the java file and makefile in a separate directory
2. Copy the input and command file that have to be input by the user into the same directory
3. Type in make
4. java bbst input.txt<commands.txt>output.txt to print in output file

**Results Observation**

All the input files test\_100, test\_1000000, test\_10000000, test\_10000000 and command file provided to check whether the event counter was implemented gave the correct output. For the input with more than 10^8 data though the heap size has to be increased so that it does not run out of memory. All the methods as implemented in given time complexity did not take much time to execute the program.

**Conclusion**

The Red Black tree can successfully implement the event counter. Its Self balancing features allows new events to be added and old events to be deleted without affecting the balancing of the tree. Also It allows to find the required event and update them in O (log n) time complexity. This saves time to find the required event.

**References**

1. (Code Reference) Introduction to Algorithms 3rd Edition
2. Algorithms 4th Edition Robert Sedgewick