COP 5536 Programming Project Report – Spring 16

TARUN GUPTA AKIRALA - UFID 43394921

Compiler Details:

- Programming language : Java
- Java compiler used : 1.7
- JDK Compliance level : 1.7

Executable (make file)

```
#!/bin/sh

if [ $# -eq 0 ]
  then
    echo "No arguments supplied"
    exit
fi
javac bbst.java
java -Xss256k -Xms256m -Xmx8g -XX:-UseGCOverheadLimit bbst $1
```

Program Structure:

There is single java file *bbst.java* which has the public class "bbst". There is another class called EventCounter which holds the data of the current tree data structure. An instance of event counter is created in the main method of bbst class and this instance is used to perform a series of operations/commands. The output is printed to the standard output and the input is read from the standard input.

The input file format should look like:

number_of_nodes
id1 count1
id2 count2
id3 count3
.
.
and so on.

Main method will do the following steps:

- Build the tree in linear time (Recursively calculates)
- Execute the commands read from std input - {increase, reduce, count, inrage, next, previous, quit}
- When quit command is entered, the program exits with status code 0

Function Details

The bbst class has only main method in which it first builds the tree and then goes in to an infinite while loop which runs as long as the standard input is provided. It will only break when the command quit is given in to the standard input.

The other class, EventCounter holds the core data structure of Red Black Tree, performs the fix ups as and when needed, has the logic to perform rotations, fix ups etc., Below is the exhaustive list of functions along with their prototype definitions.

Class bbst

Has only one method main. This creates an instance of EventCounter class at the boot and uses the created instance to perform all the operations.

Class EventCounter

Nested classes

This class has one nested class **TreeNode** which carries the design of each node of the RBT. Below is the format of the TreeNode class.

```
EventCounter.TreeNode(int ID,
                      int count,
                      byte color)
```

Parameters:

`ID` - the ID value of the event. This is used as sorting key for the Red Black Tree
`count` - the count value associated with the ID
`color` - the color value of the node
The left, right & parent nodes are not initialized in the constructor.

This class has three filed parameters to store pointers to left, right and parent nodes in RBT Data structure.

Field Summary

Modifier and Type	Field and Description
`(package private) static byte`	**`BLACK`**
`(package private) static byte`	**`RED`** By convention, RED is 0 and BLACK is -1.
`private EventCounter.TreeNode`	**`root`** This class level variable holds the pointer to root. Used by helper functions.

Method Summary

Modifier and Type	Method and Description
(package private) void	**buildTree**(int[] keys, int[] vals) This builds the tree in linear time.
private **EventCounter.TreeNode**	**buildTree**(int low, int high, int[] keys, int[] vals) Using the keys and vals arrays for ID, count values, the node is build and then its left and right children and built recursively.
(package private) void	**count**(java.lang.Integer ID) Prints the event count associated with the passed ID. Performs a binary search **in O(log(n))** time.
private **EventCounter.TreeNode**	**createRedLeft**(**EventCounter.TreeNode** node) node - Red node.left and node.left.left are Black Create a red node in node.left or one of its children. This is used to address deficiency concern.
private **EventCounter.TreeNode**	**createRedRight**(**EventCounter.TreeNode** node) node - Red node.right and node.right.left - Black Create a red node in node.right or one of its children. This is used to address deficiency concern.
private **EventCounter.TreeNode**	**delete**(**EventCounter.TreeNode** node, java.lang.Integer ID) Deletes the node with the corresponding ID. Searching for deletion takes O(log(n)) time Rebalancing the newly formed unbalanced tree may take further O(log(n)) time.
private void	**delete**(java.lang.Integer ID) Deletes the node with the corresponding ID.
private **EventCounter.TreeNode**	**delMinimum**(**EventCounter.TreeNode** node) Deletes the key-value pair with the minimum key rooted at node. This is used as a helper function to the delete method.

private **EventCounter.TreeNode**	**findPredecessor**(**EventCounter.TreeNode** node, java.lang.Integer eventId) This is used by the previous command. Prints the ID and the count of the event with the greatest key that is less than the given eventID. Prints “0 0”, if there is no previous ID.
private **EventCounter.TreeNode**	**findSuccessor**(**EventCounter.TreeNode** node, java.lang.Integer eventId) This is used by the next command. Prints the ID and the count of the event with the smallest key that is greater than the given eventId. Prints “0 0”, if there is no next ID
private **EventCounter.TreeNode**	**fixViolations**(**EventCounter.TreeNode** node) Check the invariants of RB Tree. This checks for all the three cases given in CLRS. Performs rotations as applicable.
private int	**getLeftHeight**(**EventCounter.TreeNode** node) Returns the height along the left nodes
private **EventCounter.TreeNode**	**getNode**(java.lang.Integer ID) Search for an ID in O(log(n)) time - Binary search.
private int	**getRightHeight**(**EventCounter.TreeNode** n) Returns the height along the right nodes
(package private) void	**increase**(java.lang.Integer ID, java.lang.Integer count) If the node is not already present, this will insert a new node.
(package private) void	**inRange**(java.lang.Integer ID1, java.lang.Integer ID2) Print the total count for IDs between ID1 and ID2 inclusively.
(package private) void	**insert**(java.lang.Integer ID, java.lang.Integer count) Inserts the node into the tree.
private boolean	**isRed**(**EventCounter.TreeNode** node) Checks if a node color is red or not.

private **EventCounter.TreeNode**	**leftRotate**(**EventCounter.TreeNode** node) Performs left rotation on the current node
private void	**markRed**(**EventCounter.TreeNode** node, int curLevel, int lastLevel) Marks the current node as Red.
(package private) void	**next**(java.lang.Integer ID) Print the ID and the count of the event with the lowest ID that is greater that theID.
private void	**populateList**(**EventCounter.TreeNode** node, java.util.ArrayList<**EventCounter.TreeNode**> nodes, java.lang.Integer ID1, java.lang.Integer ID2) Add all nodes between ID1 and ID2 (inclusive) to list
(package private) void	**previous**(java.lang.Integer ID) Print the ID and the count of the event with the highest ID that is less that theID.
(package private) void	**reduce**(java.lang.Integer ID, java.lang.Integer count) If the node is not already present, this operation does not do anything.
private **EventCounter.TreeNode**	**rightRotate**(**EventCounter.TreeNode** node) Performs right rotation on the current node
private void	**swapColors**(**EventCounter.TreeNode** node) alters the colors of children and parent node.
private **EventCounter.TreeNode**	**upsert**(**EventCounter.TreeNode** node, java.lang.Integer ID, java.lang.Integer count, boolean doPrint) This will update / insert accordingly.
private void	**upsert**(java.lang.Integer ID, java.lang.Integer count, boolean doPrint) This will update / insert accordingly

Results

For the 1GB test file, default JVM options are not sufficient. 100,000,000 (Hundred million entries) = 200,000,000 Integers. Each Integer in java (primitive data type) will take up 4 Bytes. For 200 Million integers – 200000000 * 4B = 800000000B = 800000KB = 800MB. For 100 Million nodes, each node will have ID, count, a byte indication color – 9 Bytes per Node – This means that for 100 Million nodes, 900MB is consumed. Also we are storing references to left, parent and right child nodes. This will consume even more memory. Below table shows the results for the tests ran.

Input size	Time taken
100	Less than two seconds
1000000	Less than two seconds
10000000	Less than four seconds
100000000	Less than 40 seconds

```
Lavenger@LAPTOP-AA0UQ1QC MINGW64 ~/git/RedBlackTree/src
$ time ./bbst test_100.txt < commands.txt > output_100_43394921.txt

real    0m1.039s
user    0m0.045s
sys     0m0.185s

Lavenger@LAPTOP-AA0UQ1QC MINGW64 ~/git/RedBlackTree/src
$ time ./bbst test_1000000.txt < commands.txt > output_1000000_43394921.txt

real    0m1.393s
user    0m0.031s
sys     0m0.170s

Lavenger@LAPTOP-AA0UQ1QC MINGW64 ~/git/RedBlackTree/src
$ time ./bbst test_10000000.txt < commands.txt > output_10000000_43394921.txt

real    0m3.329s
user    0m0.015s
sys     0m0.139s

Lavenger@LAPTOP-AA0UQ1QC MINGW64 ~/git/RedBlackTree/src
$ time ./bbst test_100000000.txt < commands.txt > output_100000000_43394921.txt

real    0m37.456s
user    0m0.000s
sys     0m0.154s

Lavenger@LAPTOP-AA0UQ1QC MINGW64 ~/git/RedBlackTree/src
$ diff -b output_100_43394921.txt out_100.txt

Lavenger@LAPTOP-AA0UQ1QC MINGW64 ~/git/RedBlackTree/src
$ diff -b output_1000000_43394921.txt out_1000000.txt

Lavenger@LAPTOP-AA0UQ1QC MINGW64 ~/git/RedBlackTree/src
$ lsa output*
-rw-r--r-- 1 Lavenger 197609 106 Mar 20 04:05 output_100_43394921.txt
-rw-r--r-- 1 Lavenger 197609 111 Mar 20 04:05 output_1000000_43394921.txt
-rw-r--r-- 1 Lavenger 197609 111 Mar 20 04:05 output_10000000_43394921.txt
-rw-r--r-- 1 Lavenger 197609 114 Mar 20 04:06 output_100000000_43394921.txt
```

Above is the screenshot of terminal running given test cases. As required, the diff command does not show any difference between the desired and generated outputs. Note that instead of executable, we can also use the java bbst command to execute the test cases. Similar observations were found in Thunder server.

Attachments (inside zip file)

- Source code
- java docs
- Command file
- Output files (For 100, 1000000, 10000000, 100000000 entries)
- Executable file (bbst).

References

CLRS – Red Black Tree – 13th Chapter.