Project Report - Implementation of an Event Counter using Red Black Tree

(Spring 2016)

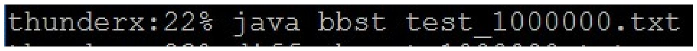
Working environment details:

Language used: Java
Compiler: javac version: "1.8.0_65"
Java runtime environment: java version: "1.8.0_73"

Details on how to execute the program:

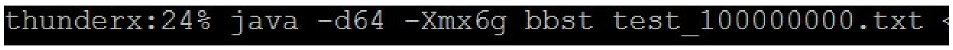
Once the program (bbst.java) is compiled using the makefile, given below is how we can execute the program from command line:

1. Pass in one command line argument (the file which has all data to create the initial tree)



```
thunderx:22% java bbst test_1000000.txt
```

2. In case the file being passed is too large (like test_100000000.txt file provided), then use the below method to execute to allocate heap memory



```
thunderx:24% java -d64 -Xmx6g bbst test_100000000.txt <
```

List of methods involved in the program along with their method signatures/prototypes:

- public static void main(String[] args)
- public void handleIO()
- Node arrayToRedBlackTreeRoot(int[] keys, int[] counts, int i, int j, int redLevel)
- int calculateRedLevel(int size)
- public void buildInitialRBT(String filename)
- int inRange(int id1, int id2, Node temp)
- public Node predecessor(int value, Node node)
- public Node successor(int value, Node node)
- void deleteFixUp(Node x)
- Node treeMaximum(Node subtree)
- Node treeMinimum(Node subtree)
- void replaceNodeWith(Node target, Node with)
- void delete(Node z)
- private Node findNode(int value, Node node)
- void rotateRight(Node node)
- void rotateLeft(Node node)
- private void insertFixUp(Node node)
- private void insert(Node node)

- detailed description of what each function does is provided as comments on top of the respective functions in the source code

Commands available:

- increase ID count
- reduce ID count
- count ID
- inrange ID1 ID2
- next ID
- previous ID
- quit

Structure of the program:

Below is an explanation of the structure of the program (program flow) through the descriptions of the given set of commands:

➢ *public static void main(String[] args)*
It builds an initial Red Black Tree from the file that is passed in to the program as a command line argument and handles the complete set of operations (all commands). Calls the below 2 methods:

main → buildInitialRBT(filename) --- > to build initial Red Black Tree from the input file
main → handleIO() --- > to handle the complete set of operations

➢ *public void buildInitialRBT(String filename)*
Builds the initial Red Black Tree by accessing the file that was passed into the program as a command line argument. Reads the file and forms arrays to hold key and count values. Calls the method *calculateRedLevel* to get the height of the tree that will be constructed using the size of the tree. Then calls the method *arrayToRedBlackTreeRoot* to actually construct the tree (in O(n) time) using the arrays (sorted input) that were populated in this method.

buildInitialRBT → int calculateRedLevel(int size) --- > to get the height of the tree
buildInitialRBT→ Node arrayToRedBlackTreeRoot(int[] keys, int[] counts, int i, int j, int redLevel)
--- > a recursive method to construct the tree using the arrays. Returns the root of the tree that was built.

➢ *public void handleIO()*
To handle all the interactive part of the commands entered on the standard input. Commands are as follows: *increase, reduce, count, inrange, next, and previous.* This method keeps running and keeps scanning for input to be entered on the standard input. This will exit only when *"quit"* is encountered on the standard input. After performing each of the commands the control returns back to this method and the next input is read in.
Given below is the description of what happens when each of these commands are entered on the standard input:

- ***increase*** – the ID and the count entered on the standard input are read in and in turn a call is made to the recursive method *findNode(int value, Node node)* by passing in the ID that was read and also the root of the tree.

private Node findNode(int value, Node node)- This method searches the tree and returns the searched node if found. If not it returns null. If the node was found, then its count is increased by the amount mentioned in the command and the new count is printed onto the output.
If null was returned, then a new node is inserted into the existing tree with the ID and count as what was read from the input and the count is printed. To insert the new node the method *insert(Node node)* is called.

increase → *findNode* then, if null was returned *increase* → *insert*

private void insert(Node node) - inserts a single new node that has been passed into this method to the existing Red Black Tree. If the node being inserted is not the root, then its color is set as RED. Determines the position of insertion by comparing the key value of the passed in node with the key values of the existing nodes in the tree. The new node is always being inserted as a leaf node. Finally after insertion, fix the tree by calling *insertFixUp* (adjusts the RedBlackTree after a node has been inserted. This method further calls *rotateLeft* and *rotateRight* methods to adjust the RedBlackTree) method for the new node

insert → *insertFixUp* → *rotateLeft* and *insert* → *insertFixUp* → *rotateRight*

- ***reduce –*** As in *increase, findNode* is called.
  reduce → *findNode* then, if null was returned print 0. Else, if node found, reduce the count by the amount mentioned on the input. If count goes less than 1, then delete that node by calling *delete(Node z)* function.

  void delete(Node z) - To delete a single new node that has been passed into this method from the existing Red Black Tree. Handles 3 cases. Calls *replaceNodeWith* method to actually take out the node to be deleted and brings in the appropriate node to the deleted position. Finally after deletion, fix the tree by calling *deleteFixUp* method for the appropriate node.

  reduce → *delete* → *rotateLeft* and *reduce* → *delete* → *rotateRight*

- ***count –*** Read in the ID from the standard input. As in the vase of the command *increase,* the function *findNode* is called by passing in the ID and the *root*.

  count → *findNode* then, if null was returned print 0. Else, if node found, print on to output the count of the returned node.

- ***Inrange -*** Read in the 2 IDs from the standard input and call the function int *inRange(int id1, int id2, Node temp)* by passing in the 2 read in IDs along with the *root.*

  inRange - This function sums up all the count values of all the keys which lie in between the lower and upper IDs which form the range. It returns the sum of these counts. Print this sum of counts on to the output.

- ***next –*** Read in the ID from the input. Call the function *successor* with the ID and the *root* as parameters.
  public Node successor(int value, Node node) – it's a recursive function. Searches for the node with the ID that's passed in. If found, its successor is returned. Even if the node is not found, if there is a successor, then return that. If there is no successor, return null.
  If found, print the successor's ID and count, else print 0 0.
  The function *successor* in turn calls 2 more functions:

  successor →Node treeMinimum(Node subtree)
  --- > To find the minimum most element in the subtree rooted at the node that is being passed into this method.

  successor →Node treeMaximum(Node subtree)
  --- > To find the maximum most element in the subtree rooted at the node that is being passed into this method.

- ***previous –*** Read in the ID from the input. Call the function *predecessor* with the ID and the *root* as parameters.
  public Node predecessor(int value, Node node) – it's a recursive function. Searches for the node with the ID that's passed in. If found, its predecessor is returned. Even if the node is not found, if there is a predecessor, then return that. If there is no predecessor, return null.
  If found, print the predecessor's ID and count, else print 0 0.
  The function *predecessor* in turn calls 2 more functions:

  predecessor →Node treeMinimum(Node subtree)
  --- > To find the minimum most element in the subtree rooted at the node that is being passed into this method.

  predecessor →Node treeMaximum(Node subtree)
  --- > To find the maximum most element in the subtree rooted at the node that is being passed into this method.

<u>**References:**</u>

- Lecture videos available on CANVAS
- Lecture slides available on Course web page
- Text book "Introduction to Algorithms" by CLRS – 3rd Edition