COP 5536 Advanced Data Structures

Spring 2016

Programming Project Report

Implementation of event counter using red-black tree

Submitted by

Name: Karthik Maharajan Sankara Subramanian

UFID: 41984439 Email: skarthikmaharaja@ufl.edu

Working Environment and Compiler Description

Working Environment

Hardware Requirement
Hard Disk Space: 256 GB
RAM: 8GB

Operating System
Mac OS X

Compiler Description

Platform/ Operating System	Compiler	Test Result
Mac OS X (Local Machine)	Netbeans	Pass
Linux (thunder.cise.ufl.edu)	Javac	Pass

Instructions for compiling and running the program

The project has been compiled and tested on the following environments,

1) Netbeans IDE
2) thunder.cise.ufl.edu

All test cases for the following sample input files provided have passed,

1) test_100.txt
2) test_1000000.txt

Perform the following steps to compile and run the program

1) Remotely access the server using the following command from the terminal
   ssh thunder.cise.ufl.edu -l <username>
2) Provide the correct password for successful login
3) Ensure that all of the following files are present in the working directory,
   i) Input text files containing the initial IDs and counts of the event counter
   ii) Files with commands to be executed on the event counter
   iii) Java file (Source program)
   iv) Makefile
   v) Compile the program using the makefile as follows
   MacBook-Pro:src KarthikMaharajan$ **make**
   vi) Run the program as follows
   MacBook-Pro:src KarthikMaharajan$ java **mainclassname inputtextfilename**< **commandstextfilename** > **outputtextfilename**

see below for examples,

MacBook-Pro:src KarthikMaharajan$ java bbst test_100.txt < commands.txt > output100.txt
MacBook-Pro:src KarthikMaharajan$ java bbst test_1000000.txt < commands.txt > output1000000.txt
MacBook-Pro:src KarthikMaharajan$ java bbst test_10000000.txt < commands.txt > output10000000.txt

> **Note:** while running the program for the input file with 100000000 events, increase the heap size of the Java Virtual Machine to 8GB using the following command
>
> MacBook-Pro:src KarthikMaharajan$ java -Xmx8g **classname inputtextfilename**< **commandstextfilename** > **outputtextfilename**
>
> Use a similar approach for other large files as well.

Example,

MacBook-Pro:src KarthikMaharajan$ java -Xmx8g bbst test_100000000.txt < commands.txt > output100000000.txt

vii) Find the results in **outputtextfilename.txt**

Please find below the output and the running time for the input files provided,

test_100.txt

100
50
50
50
156
206
0
50
50
350 50
350 50
0 0
350 50
271 8
0 0
2
271 2
0
267 8
147 2

real 0m0.214s
user 0m0.106s
sys 0m0.052s

test_1000000.txt

104
54
54
1363
192
1555
101
54
61
303 6
350 54
363 8
359 5
349 7
0 0
0
349 7
0
349 7
146 2

real 0m0.951s
user 0m1.428s
sys 0m0.171s

test_10000000.txt

109
59
59
1363
185
1548
103
59
59
301 6
350 59
363 5
358 2
346 8
0 0
0
346 8
0
346 8
147 9

real 0m9.189s
user 0m19.118s
sys 0m0.648s

test_100000000.txt

106
56
56
1344
168
1512
93
56
66
303 3
350 56
362 8
358 10
349 10
0 0
0
349 10
0
349 10
149 7

real 1m32.881s
user 2m5.847s
sys 0m22.535s

Function Prototypes and Program Structure

The source program contains the two classes, **1) bbst 2) TreeNode. bbst** is the class containing the main function and **TreeNode** is present within **bbst.**

Function Prototypes

Class TreeNode contains the following function,

TreeNode(int key, int count)

This function is the constructor of the TreeNode class and is used to create and initialize each node in the red-black tree. It sets the ID attribute to the value passed in "key" and sets the count attribute to the value passed in "count". It sets the color of every node to black when created by setting ISRed attribute to false. It also assigns null value to the leftchild, rightchild and parent.

Class bbst contains the following functions,

public static void main(String[] args)

The main function gets as input, the input text file with the initial event IDs and counts. An array with all IDs and an array with the corresponding counts is created by reading from the input file. The arrays are passed to **arraytobst** function to create the red-black tree. After the tree is created, the commands are read from the standard input stream and the operations are performed by invoking the appropriate functions. The results of the operations are sent to the standard output stream.

public TreeNode arraytobst(int[] id,int[] count,int start, int end,int height)

This function is used to create the red black tree from the id array, count array and the height of the tree in O(n) time. The middle event is made the root. Then the left subtree and right subtree of the root are recursively constructed. After the balanced binary search tree is created, it is converted to a red black tree by converting the color of all the levels in the last node from black to red.

public int count(int id)
This function gets the event id as input and returns the count of that event if it is present in the event counter. Otherwise, it returns 0.

public TreeNode iterativeSearch(int id)

This function is used to iteratively search the red-black tree for the event with the given id. It returns the treenode with the given id.

public int increase(int id,int m)

This function is used to increase the count of the given event id by "m". If the event is present in the counter, it increases the count and returns the count value. Otherwise, it inserts the event into the red-black tree and returns "m".

public void insertNode(int id,int count)

This function gets as input the id and count of the event not present in the counter. It then creates a new node for this event and inserts it into the red-black tree. If the insertion violates the red-black tree properties, then **insertfixup** function is called to fix the properties.

public void insertfixup(TreeNode z)

This function gets as input the node inserted and fixes the red-black tree properties. It implements this by checking the color of node y which is the uncle of z and by checking if z is

the left child or right child of its parent. As discussed in class, three cases arise and the violations are fixed using left and right rotations.

public void leftRotate(TreeNode x)

This function is used to perform left rotation on the nodes involved, i.e. nodes x and y. x is passed to the function and y is x's rightchild.

public void rightRotate(TreeNode y)

This function is used to perform right rotation on the nodes involved, i.e. nodes x and y. y is passed to the function and x is y's leftchild.

public int decrease(int id,int m)

This function is used to decrease the count of the given event id by "m". If the event is present in the counter, it decreases the count and returns the count value. Otherwise, it returns 0. If the count of the event is less than or equal to 0 after the decrease then, the node is deleted from the tree by calling **deleteNode** function.

public void deleteNode(int id,int m)

This function is used to delete the node with the given id. If the deletion violates the red-black tree properties, then **deletefixup** function is called to fix the properties.

public void deletefixup(TreeNode x)

This function gets as input node x which is neither red nor black and hence the red-black properties should be restored. It handles 4 cases depending on the attributes of node w and its children. w is the sibling of x. As discussed in class, the 4 cases are fixed using left and right rotations.

public TreeNode treeMinimum(TreeNode x)

This function is used by other functions in the bbst class to determine the minimum element in the red-black tree

public TreeNode treeMaximum(TreeNode x)

This function is used by other functions in the bbst class to determine the maximum element in the red-black tree

public void transplant(TreeNode u,TreeNode v)

This function is used to perform transplant on the nodes u and v. Transplant operation replaces one subtree as a child of its parent with another subtree.

public int inRange(int id1,int id2)

This function is a wrapper for the **rangeImp** function that implements the range search operation.

public int rangeImp(TreeNode rootnode,int id1,int id2)

This function is used to find the sum of the counts of all events between id1 and id2 inclusive. Here the argument "rootnode" is the root of the tree. The function computes the required sum using recursion and returns the final sum.

public void next(int id)

This function is used to print the id and count of the event with the lowest id that is greater than the given id. It returns "0 0" if there is no next id. The case when the given id is not present in the tree is handled by the function **nextNoKey.**

public void previous(int id)

This function is used to print the id and count of the event with the greatest id that is lesser than the given id. It returns "0 0" if there is no previous id. The case when the given id is not present in the tree is handled by the function **previousNoKey.**

public TreeNode nextNoKey(int id)

This function handles the functionality of finding the next id when the given id is not present in the tree. It returns the node with the lowest id that is greater than the given id by determining whether the search for the given id falls off the tree from left or right.

public TreeNode previousNoKey(int id)

This function handles the functionality of finding the previous id when the given id is not present in the tree. It returns the node with the greatest id that is lesser than the given id by determining whether the search for the given id falls off the tree from left or right.

Program Structure

The program structure for the operations on the red-black tree are as follows,

1) Increase (the ID, m)
   Time Complexity: O(log n)

   Function **increase** is invoked to increase the count of the event by m. If the ID is present in the counter, the function increase returns the increased count value. If the ID is not

present, then the ID is inserted by calling the **insertNode** function and the m value is returned as count by **increase.**

2) Reduce (the ID, m)
   Time Complexity: O(log n)

   Function **decrease** is invoked to decrease the count of the event by m. If the ID is present in the counter, the function **decrease** returns the decreased count value. If the ID is not present, then 0 is printed. If the decreased value is less than or equal to zero, then the node is deleted by calling **deleteNode** function and zero is returned by **decrease.**

3) Count (the ID)
   Time Complexity: O(log n)

   Function **count** is invoked to get the count value of the given ID. If the ID is not present, **count** returns 0.

4) InRange (ID1, ID2)
   Time Complexity: O(log n + s) where s is the number of IDs in the range.

   Function **inRange** is invoked to get the sum of the count of the events with IDs between ID1 and ID2 inclusive. **inRange** is a wrapper function which invokes **rangeImp** which implements the functionality and returns the required sum.

5) Next (the ID)
   Time Complexity: O(log n)

   Function **next** is invoked to print the id and count of the event with the lowest id that is greater than the given id. To handle the case when the given ID is not present in the event counter **next** invokes **nextNoKey** to get the next event. If there is no next ID, it prints "0 0".

6) previous (the ID)
   Time Complexity: O(log n)

   Function **previous** is invoked to print the id and count of the event with the greatest id that is lesser than the given id. To handle the case when the given ID is not present in the event counter **previous** invokes **previousNoKey** to get the previous event. If there is no previous ID, it prints "0 0".

Observations

As observed from the running times of the program the red-black tree is a highly efficient data structure for the implementation of the event counter. This is because of the efficient time complexities of tree creation step and the operations shown above. The tree creation takes O(n) time. The algorithms for search, insert and delete in a red-black tree used by the operations take O(log n) time because of the balanced nature of the tree.

Conclusion

Thus the event counter was successfully implemented by using the red-black tree data structure.

REFERENCES

Introduction to Algorithms, Third Edition By Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein