Advances Data Structures (COP 5536)
Spring 2016
Programming Project Report

Implementing an Event Counter using Red-Black Tree

Manmeet Singh Khurana
UFID 6331-4786
mkhurana@ufl.edu

1. Working Environment:

Visual Studio Express (in **Windows 10**) was used to write the code in **C++**. The code has been testing in Linux using **g++ compiler**. The project has been compiled and tested on thunder.cise.ufl.edu.

To execute the program, you can remotely access the server using SSH client username@thunder.cise.ufl.edu .

2. Compile Process:

Makefile can be found in the zip file submitted on canvas.

A. To compile the program, run the command:-
   $ make
   This statement will execute the command “g++ -o bbst source.cpp”
B. The above command will generate an object file named “bbst”.
C. To run the object file, run the command:-
   $ ./bbst file-name
   Where file-name can be “test_100.txt” which will contain the initial sorted list of IDs with their respective count. As posted in the announcements the code will be tested by running the commands like:-
   $ ./bbst test_1000000.txt < commands.txt > out_1000000.txt
   Here the “test_1000000.txt” will be containing 1000000 IDs and their respective counts. “commands.txt” will be containing the commands to be passed to the code to test the functionality of the code. And finally “out_1000000.txt” will contain the program output.
D. After the successful testing of the code, the object file can be removed using:-
   $ make clean
   This statement will execute the command “rm bbst”

3. Problem Description:

We have to implement an event counter using red-black tree. Each event will have two fields: *ID* and *count*, where *count* is the number of active events with the given *ID*. The event counter stores only those *ID*'s whose *count* is > 0. Once a *count* drops below 1, that *ID* is removed. Initially, your program must build red black tree from a sorted list of *n* events (i.e., *n* pairs (*ID*, *count*) in ascending order of *ID*) in O(*n*) time. Our counter should support the following operations:

A. Increase(theID, m)
B. Reduce(theID, m)
C. Count(theID)
D. InRange(ID1, ID2)
E. Next(theID)
F. Previous(theID)

4. Program structure and Function Prototypes:

All the source code of the file is contained in one file only, "source.cpp". The execution of the program starts from the "**main()**". First it opens a file in read mode and then it is used to read the IDs and their respective counts.

These pairs of (ID,count) are saved in the **vector**. Then "**obj.initial_insert()**" is called to initialize the tree with **O(n)** complexity. After the Initialization the program gets the commands from the "**commands.txt**" file. "**While**" is used to get the commands with the parameters until it receives "**quit**".

There can be six types of commands and they are the same as the operations which we need to support for our counter i.e **increase, reduce, count, inrange, next, previous**. Depending on the command fetched, the numbers of parameters may be 2 or 1 and also the control of the program will be shifted to the corresponding functions. To understand how the control continues, we need to have a look at the function prototypes and what each one of them does.

The following are the functions used in the program with their prototype and function:-

1. **Function**: void Increase(long long int ID, long long int count)
   Arguments: ID of the event and increase value of the count.
   Return value: Void.
   Description: Increase the count of the event ID by count. If ID is not present, insert it. Print the count of ID after the addition.

2. **Function**: void Reduce(long long int ID, long long int value)
   Arguments: ID of the event and decrease value of the count.
   Return value: Void.
   Description: Decrease the count of ID by count. If ID's count becomes less than or equal to 0, remove ID from the counter. Print the count of ID after the deletion, or 0 if ID is removed or not present.

3. **Function**: void Count(long long int ID)
   Arguments: ID of the event.
   Return value: Void.
   Description: Print the count of ID. If not present, print 0.

4. **Function**: void InRange(long long int ID1, long long int ID2)
   Arguments: ID of the event1 and event2.
   Return value: Void.
   Description: Print the total count for IDs between ID1 and ID2 inclusively. Note, ID1 ≤ ID2.

5. **Function**: void Next(long long int ID)
   Arguments: ID of the event.
   Return value: Void.
   Description: Print the ID and the count of the event with the lowest ID that is greater that ID. Print "0 0", if there is no next ID.

6. **Function**: void Previous(long long int ID)
   Arguments: ID of the event.
   Return value: Void.
   Description: Print the ID and the count of the event with the greatest key that is less that ID. Print "0 0", if there is no previous ID.

7. **Function**: void Insert(long long int ID, long long int count)
   Arguments: ID and Count of the event.
   Return value: Void.
   Description: Used to insert a node in a RB Tree.

8. **Function**: void insertfix(node *t)
   Arguments: Pointer to a node in the RB Tree.
   Return value: Void.
   Description: Used to fix any issue after Insert function.

9. **Function**: void leftrotate(node *p)
   Arguments: Pointer to a node in the RB Tree.
   Return value: Void.
   Description: Used to rebalance the tree after delete or insert.

10. **Function**: void rightrotate(node *p)
    Arguments: Pointer to a node in the RB Tree.
    Return value: Void.
    Description: Used to rebalance the tree after delete or insert.

11. **Function**: node* successor(node *p)
    Arguments: Pointer to a node in the RB Tree.
    Return value: Pointer to a node in the RB Tree.
    Description: Used to know the successor of a node in RB Tree and also used in Next() and Previous().

12. **Function**: node *search(long long int ID)
    Arguments: ID of the event.
    Return value: Pointer to a node in the RB Tree.
    Description: Used to search the RB Tree for a given ID.

13. **Function**: node *searchnf(long long int ID)
    Arguments: ID of the event.
    Return value: Pointer to a node in the RB Tree.

Description: Similar to search() but will return immediate(next/previous) node if ID not found, whereas search() returns NULL is ID not found.

14. **Function**: node *sibling(node *x)
    Arguments: Pointer to a node in the RB Tree.
    Return value: Pointer to a node in the RB Tree.
    Description: Returns the pointer of the sibling corresponding to the node passed into this function.

15. **Function**: void replace_node(node *oldn, node *newn)
    Arguments: Pointer to new and old node in the RB Tree.
    Return value: Void.
    Description: Used to replace old node(oldn) with new node(newn).

16. **Function**: void delete_prog2(node *n)
    Arguments: Pointer to a node in the RB Tree.
    Return value: Void.
    Description: Main delete function of the program, every node which needs to be deleted, enters this function.

17. **Function**: void delete_case_noChild1(node *n)
    Arguments: Pointer to a node in the RB Tree.
    Return value: Void.
    Description: Will be called when the node being deleted has no child.

18. **Function**: void delete_case1(node *n)
    Arguments: Pointer to a node in the RB Tree.
    Return value: Void.
    Description: Will be called if node being deleted has 1 child.

19. **Function**: void delete_case2(node *n)
    Arguments: Pointer to a node in the RB Tree.
    Return value: Void.
    Description: Called from delete_case1() if node being deleted has a parent.

20. **Function**: void delete_case3(node *n)
    Arguments: Pointer to a node in the RB Tree.
    Return value: Void.
    Description: This function will check for sibling's children.

21. **Function**: void delete_case4(node *n)
    Arguments: Pointer to a node in the RB Tree.
    Return value: Void.
    Description: This function is used to check if parent, sibling and sibling’s children of the node being deleted are black in color or not.

22. **Function**: void delete_case5(node *n)
    Arguments: Pointer to a node in the RB Tree.
    Return value: Void.
    Description: It will check if both of sibling's children are not black i.e if one of them is red in color.

23. **Function**: void delete_case6(node *n)
    Arguments: Pointer to new and old node in the RB Tree.
    Return value: Void.
    Description: This will be called when further changes after delete_case 5 are needed.

24. **Function**: long long int inRange(long long int ID1, long long int ID2)
    Arguments: ID of the even1 and event2.
    Return value: long long Integer.
    Description: This function has the actual implementation of inrange function.

25. **Function**: node * next_IDptr(long long int inID)
    Arguments: ID of the event.
    Return value: Pointer to a node in the RB Tree.
    Description: Will return the pointer to the next ID.

26. **Function**: node *min(node *x)
    Arguments: Pointer to a node in the RB Tree.
    Return value: Pointer to a node in the RB Tree.
    Description: Returns minimum of the node in its subtree. If no min, returns the node.

27. **Function**: node *max(node *x)
    Arguments: Pointer to a node in the RB Tree.
    Return value: Pointer to a node in the RB Tree.
    Description: Returns maximum of the node in its subtree. If no max, returns the node.

28. **Function**: int bin_count(long long int m)
    Arguments: Integer (size of the vector).
    Return value: Integer.
    Description: Finding the biggest index of form 2^n which is less than size of the vector(containing all IDs).

29. **Function**: node *createNode(long long int ID, long long int count, char color)
    Arguments: ID, count and color of a particular node.
    Return value: Pointer to a node in the RB Tree.
    Description: Creates nodes of the RB tree structure to initialize it.

30. **Function**: node *sortedArrayToBST(node *r, long long int start, long long int end, int flag)
    Arguments: Pointer to a node in the RB Tree, Start, end and level of the current RB tree portion.
    Return value: Pointer to a node in the RB Tree.
    Description: This function initializes the Red Black tree using sorted input IDs.

31. **Function**: void initial_insert()
    Arguments: No argument.
    Return value: Void.

Description: This function calls the other functions to start the initialization of the Red Black tree.

32. **Function**: int main(int argc, char* argv[])
Arguments: Argument count and arguments from the command line.
Return value: Int.
Description: Execution of the program starts from this function.

Since now we have seen what each function takes in as parameters, what they perform and what do they return, lets take an example and see how the flow moves from one function to other.

Let's say we are currently in "Main()" and we got a command "inrange 300 349". After getting this command, "InRange()" will be called from the main() and flow will now go to "InRange()". From there the same inputs i.e 300 and 349 will be passed on to "inRange()" which has a return type of "**long long int**". After the flow reaches "inRange()" we will find the node of the RB Tree corresponding to 300 and 349. Now flow will go to "search()" function twice to search for node with IDs 300 and 349. If node corresponding to 300 is not found, then we will find its immediate next node in the tree for which flow/control will move to "next_IDptr()". Similarly if we don't find node corresponding to 349, the flow/control will go to "previous_IDptr" to get the immediate previous of 349.

After we get these nodes or their immediate next/previous then we will start from node 300(or its immediate next node) and we keep calling "next_IDptr()" and simultaneously we keep on adding all the counts along the way until we reach the node with ID 349(or its immediate previous node). Finally we return the final count value to "InRange()" function, which prints it and then sends the control/flow to the main function from where it was called in the first place and then the main will wait for the next command to be fetched.

So we can see that flow changes a lot and that too so rapidly. This was the flow description for only "InRange()" function. Similar concept applies to all other functions.

5. Result Summary

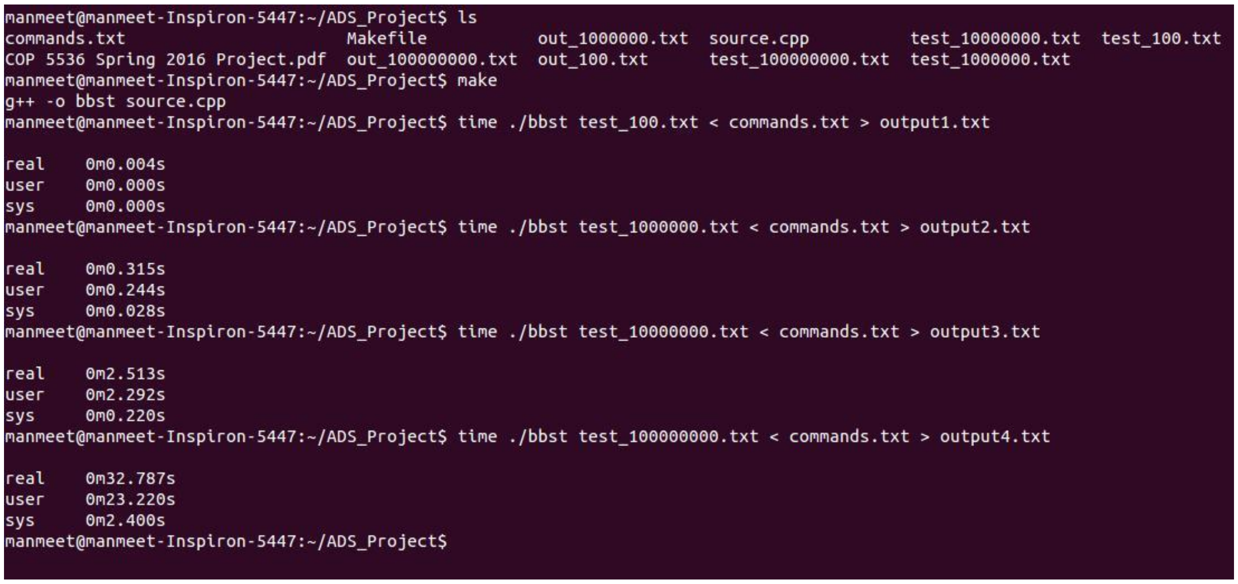
The code was tested with 4 different input files:-

1. test_100.txt
2. test_1000000.txt
3. test_10000000.txt
4. test_100000000.txt

which contained 100, 1000000, 10000000, 100000000 pairs of ID and count and these inputs were sorted according to IDs where ID(i)<ID(i+1).

The commands file used for these all inputs was the same, “commands.txt”. The command file contained 20 commands followed by the “quit” command. The code was compiled using “**make**” and executed using “**$ ./bbst test_1000000.txt < commands.txt > output1.txt**”.

The screenshots below shows the running time of the code for different input files which all use the same commands file.



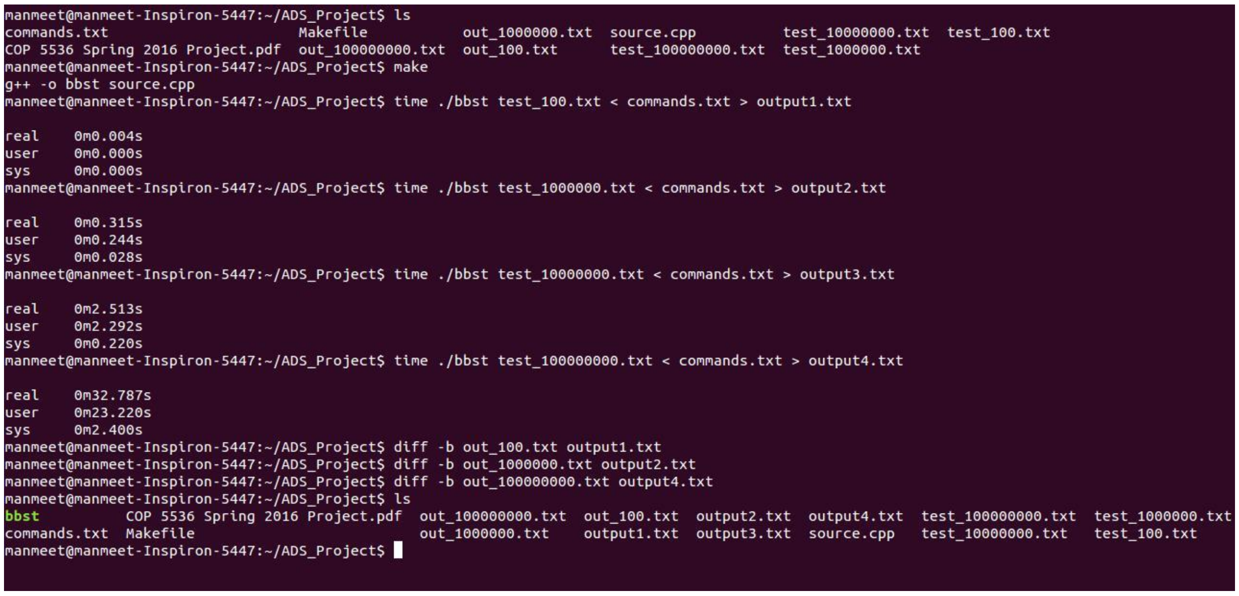
```
manmeet@manmeet-Inspiron-5447:~/ADS_Project$ ls
commands.txt                   Makefile          out_1000000.txt  source.cpp          test_10000000.txt  test_100.txt
COP 5536 Spring 2016 Project.pdf  out_100000000.txt  out_100.txt      test_100000000.txt  test_1000000.txt
manmeet@manmeet-Inspiron-5447:~/ADS_Project$ make
g++ -o bbst source.cpp
manmeet@manmeet-Inspiron-5447:~/ADS_Project$ time ./bbst test_100.txt < commands.txt > output1.txt

real	0m0.004s
user	0m0.000s
sys	0m0.000s
manmeet@manmeet-Inspiron-5447:~/ADS_Project$ time ./bbst test_1000000.txt < commands.txt > output2.txt

real	0m0.315s
user	0m0.244s
sys	0m0.028s
manmeet@manmeet-Inspiron-5447:~/ADS_Project$ time ./bbst test_10000000.txt < commands.txt > output3.txt

real	0m2.513s
user	0m2.292s
sys	0m0.220s
manmeet@manmeet-Inspiron-5447:~/ADS_Project$ time ./bbst test_100000000.txt < commands.txt > output4.txt

real	0m32.787s
user	0m23.220s
sys	0m2.400s
manmeet@manmeet-Inspiron-5447:~/ADS_Project$
```

From the screenshots we can see that the execution time for different input files were:-

1.	test_100.txt	~	0minutes 0.004seconds
2.	test_1000000.txt	~	0minutes 0.315seconds
3.	test_10000000.txt	~	0minutes 2.513seconds
4.	test_100000000.txt	~	0minutes 32.787seconds

Finally the output was verified using the diff command:-

$ diff -b out_1000000.txt output1.txt.



```
manmeet@manmeet-Inspiron-5447:~/ADS_Project$ ls
commands.txt                 Makefile            out_1000000.txt  source.cpp          test_10000000.txt  test_100.txt
COP 5536 Spring 2016 Project.pdf  out_100000000.txt  out_100.txt      test_100000000.txt  test_1000000.txt
manmeet@manmeet-Inspiron-5447:~/ADS_Project$ make
g++ -o bbst source.cpp
manmeet@manmeet-Inspiron-5447:~/ADS_Project$ time ./bbst test_100.txt < commands.txt > output1.txt

real    0m0.004s
user    0m0.000s
sys     0m0.000s
manmeet@manmeet-Inspiron-5447:~/ADS_Project$ time ./bbst test_1000000.txt < commands.txt > output2.txt

real    0m0.315s
user    0m0.244s
sys     0m0.028s
manmeet@manmeet-Inspiron-5447:~/ADS_Project$ time ./bbst test_10000000.txt < commands.txt > output3.txt

real    0m2.513s
user    0m2.292s
sys     0m0.220s
manmeet@manmeet-Inspiron-5447:~/ADS_Project$ time ./bbst test_100000000.txt < commands.txt > output4.txt

real    0m32.787s
user    0m23.220s
sys     0m2.400s
manmeet@manmeet-Inspiron-5447:~/ADS_Project$ diff -b out_100.txt output1.txt
manmeet@manmeet-Inspiron-5447:~/ADS_Project$ diff -b out_1000000.txt output2.txt
manmeet@manmeet-Inspiron-5447:~/ADS_Project$ diff -b out_100000000.txt output4.txt
manmeet@manmeet-Inspiron-5447:~/ADS_Project$ ls
bbst          COP 5536 Spring 2016 Project.pdf  out_100000000.txt  out_100.txt  output2.txt  output4.txt  test_100000000.txt  test_1000000.txt
commands.txt  Makefile                          out_1000000.txt    output1.txt  output3.txt  source.cpp   test_10000000.txt   test_100.txt
manmeet@manmeet-Inspiron-5447:~/ADS_Project$ 
```

The ouput was verified for 3 input files “test_100.txt”, “test_1000000.txt”, “test_100000000.txt” which can be seen in the screenshot above.