



Red Black Tree Implementation

COP 5536 - Project 1

03.22.2016

Sudeep Reddy Gaddam

UFID: 4198 8919

sudeepgaddam@gmail.com

Overview

I have implemented the Red Black trees to store <id,counter> pairs. Supports search , Increase, Reduce, Next and Prev operations. Since, Red Black trees have logarithmic bound on the maximum height, Above operations run in logarithmic time. Inrange runs linearly proportional to keys(ID2-ID1).

Compiler and files provided:

I am using g++ -std=c++11 compiler. I am using some of the c++11 features in the code.

To run: do make; ./bbst <input-file> < command.txt > out.txt

Files: main.cc, MakeFile, red_black.h

main.cc has readCommands(), readSortedArraytoBst(), readInputIfStream(), readInputMMap() methods.

Finally, Only readSortedArraytoBst() is used and not readInputIfStream() and readInputMMap() because both of them are slower in reading inputs. Description about them can be found in comments section of code.

red_black.h has two classes declared.

1. RBNode
2. RBTree

RBTree has following methods. Important functions are given. Structure of these functions is discussed in next section.

- A. void RBInsertFixup(RBNode<T> *z);
- B. void LeftRotate(RBNode<T> *z);
- C. void RightRotate(RBNode<T> *z);
- D. RBNode<T> *RBMin(RBNode<T> *root);
- E. RBNode<T> *RBMax(RBNode<T> *root)
- F. void RBDelete(RBNode<T> *z);
- G. void RBTransplant(RBNode<T> *u, RBNode<T> *v);
- H. void RBDeleteFixup(RBNode<T> *z);
- I. RBNode<T> * RBSearchRecursive(const RBNode<T> *u,T key);
- J. RBNode<T> * RBSearchIter(T key);
- K. int setMaxHeight(); int getNodeCount();
- L. void InRangeHelper(RBNode<T> *node, int ID1, int ID2, int &InRangeCounter);
- M. void Increase(int ID, int m);
- N. void Reduce(int ID, int m);
- O. int Count(T ID);
- P. int InRange(int ID1, int ID2);
- Q. void Next(int ID);
- R. void Previous(int ID);
- S. void colorTree(RBNode<T> *node , int lvl);
- T. void initialise(vector<RBNode<T> *> &arr);
- U. void RBInsert(RBNode<T> *z);
- V. RBNode<T> *sortedArrayToBST(vector<RBNode<T> *> &arr, int start, int end);

Structure

1. Main() reads input from file. I build vector of RBNODE ptrs while reading the input and construct BST from this vector using Sorted Array property in the **readSortedArraytoBst()** function which in turn calls

```
RBTree::initialise -> RBTree::sortedArrayToBST();
                        RBTree::colorTree();
```

sortedArrayToBST() builds the BST from vector of ptrs.

colorTree() colors all nodes to Black except the nodes at the Last level(MaxHeight level)

2. After Building the BST, I call testRBProperties() to see if Red-Black properties are preserved or not, Only if its preserved, I go ahead and read commands from input one by one using readCommands() function . There are 6 kinds of inputs.
 - a. Increase
 - b. Reduce
 - c. Inrange
 - d. Count
 - e. Previous
 - f. next

For all the above commands, Depending on the command, Inputs are taken and corresponding functions are called. They are given below:

- | | |
|----|--------------------------------|
| A. | void Increase(int ID, int m); |
| B. | void Reduce(int ID, int m); |
| C. | int InRange(int ID1, int ID2); |
| D. | int Count(T ID); |
| E. | void Previous(int ID); |
| F. | void Next(int ID); |

3. Increase() calls RBInsert(). RBInsert() calls RBInsertFixup()
4. Reduce() calls RBDelete(). RBDelete() calls RBTransplant() and RBDeleteFixup().
5. Both LeftRotate() and RightRotate() are used in Increase() and Reduce() to preserve Red-Black properties depending on where the nodes are inserted.

6. More description for 3,4,5 is given in the comments section of the code.
7. testRBProperties() function checks the
 - a. BlackNode count for each and every root to leaf path and makes sure the Black Nodes count is same for all the paths.
 - b. No consecutive Red nodes in any path.
 - c. Node->Left < Node value and Node->right has >= Node value(BST Property)
8. Next(key) prints the key's next value in inorder traversal. Previous(key) prints the key's previous value in in-order traversal. If any of above is not present, Then 0 is printed.
9. Inrange(id1,id2) prints the sum of count of all nodes that have $id1 \leq \text{value} \leq id2$. Exact implementation details are given in the comments
10. RBsearchIter(key) function searches for the key iteratively. At every node, Depending on the value, It will go either left or right. If node's value == key. Then it returns that pointer to node.
11. RBTree() constructor sets the root and dummy based on parameters passed. ~RBTree() destructor calls FreeTree() which frees up the nodes allocated in a pre-order way of traversal.

Sample Output:

```
s@s:~/Desktop/ADS/project$ time ./bbst ../proj_data/test_1000000.txt <
../proj_data/commands.txt > ../proj_data/out_1m.txt
```

```
real 0m0.348s
```

```
user 0m0.336s
```


```
sys 0m0.008s
```

```
s@s:~/Desktop/ADS/project$ time ./bbst ../proj_data/test_10000000.txt <
../proj_data/commands.txt > ../proj_data/out_10m.txt
```

```
real 0m3.537s
```

```
user 0m3.336s
```

```
sys 0m0.176s
```



For 1M nodes, It is taking less than 1s. For 10M nodes, It is taking 3.6 seconds. I have also checked correctness with two sample output files given. It matches perfectly.

References

1. Chapter 13, Introduction to Algorithms, Thomas H Cormen et al.
2. <http://www.cplusplus.com>