Report for the Programming Project

Name: Zhixin Pan UFID:35554386

1. Abstract

This is a simple programming project to implement red-black trees in C++.
The complier used is g++, and the IDE being Qt, under ubuntu 13.04 operation system.
In this program, I implement the class design and functions implement for classic red-black trees as well as tree nodes.
And to test the efficiency of this bbst , the program will continue reading commands from standard input stream and output the results to standar output, while redirection supportted.

2.How to apply

In the folder, there is already a “Makefile” , so

→ type “make” in this directory to comply and build executable file named “bbst”

→ type “make cl” to delete all “.o” files and “bbst ”

When the program is running, it has to get one argument of an input file, where we initialize the red-balck tree from. Then it will continuously reading commands from stdin until meet “quit” in commands, while outputting results to stdout.

As redirection mentioned above, to test the program , bash commands in terminal could be like this:

```
./bbst test_100.txt <commands.txt> out_100.txt
```

“test_100.txt” is input file name. Output would be shown in “out_100.txt”, if there is no “out_100.txt”, it will newly create a document named that.

3. Structure of Main Function

In main function, there are 4 steps to follow shown below (Fig 1) :

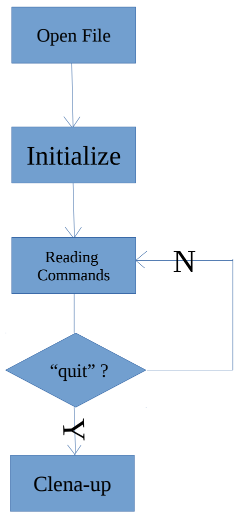


Fig 1

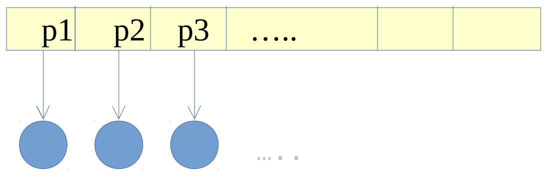


Fig 2

①To open file, we use file stream in C++, create an inputfile stream object named inutFile, and initialize it with the argument provided in terminal.

```
ifstream inputFile(argv[1]);
```

② After that, we read this file, and build treenode for each record pairs. And for each node, create a pointer to it, push it into a vector to store them all. The vector would be named “input”. Now, in the vector, we got N pointers to N objects. (Fig 2)

```
 while(inputFile >> temp1 >> temp2)
    {

        treeNode *t = new treeNode(temp1, temp2,UNDEFINED,NULL,NULL,NULL);
        input.push_back(t);
    }
```

We will use following codes to create an RB_Tree object, and initialize it with the vector

```
 RBTree*   T = new RBTree();
 T->initialize(input);
```

Details for functions will be told in next section.

③ Then there is a while loop, endlessly reading commands and decide which operation to be performed until “quit” read.

④ The final step is clean-up, where all space should be released, and the vector is free by codes:

```
vector<treeNode *>().swap(input);
```

4. Structure of RB Tree

① Class Design

What shown below is the design of the RB_tree and its nodes, for each node, we got 6 fields: the “ID” &”count” completes a pair in a search tree, while color is designed as emum type for convenience. Then , 3 pointers, to its parent, and 2 subtrees are included.

```
class treeNode
{

public:

    int ID;
    int count;
    t_color      color;
    treeNode *   parent;
    treeNode *   lChild;
    treeNode *   rChild;

    treeNode( int , int , t_color, treeNode* , treeNode* , treeNode* );
};
```

```
class RBTree
{
public:
/*----------------Member Var------------------*/
    treeNode * root;
    treeNode * NIL;  // A pointer to the NIL node for an RB tree
    int N;           // Number of treeNodes

/*----------------Constructor------------------*/
    RBTree();
    treeNode* BuildSubTree( vector<treeNode*> & , int , int , treeNode* , int );
    void initialize( vector<treeNode *> & );

/*----------------Unit_Funcs------------------*/
    void Left_Rotate ( treeNode*);
    void Right_Rotate( treeNode*);

    treeNode* Min ( treeNode* );
    treeNode* Max ( treeNode* );
    treeNode* Pred( treeNode* );
    treeNode* Succ( treeNode* );
    void Transplant( treeNode*, treeNode* );

/*----------------Basic_Funcs------------------*/
    treeNode* Search( int );

    void Insert( treeNode *);
    void Insert_Fixup( treeNode *);

    void Delete( treeNode *);
    void Delete_Fixup( treeNode *);

/*---------------Advanced_Funcs-----------------*/
    void Increase( int , int );
    void Reduce( int, int );
    void Count (int );
    int  Report( treeNode* );
    int  Inrange( int , int );
    void Next( int );
    void Previous( int );
};

#endif // RBTREE_H
```

② Class Structure

The crucial structure is the design of RB_Tree and the functions.

For an RB_Tree, we got a number N to record the total number of nodes, and a pointer which will pointerd to a NIL node. And the root pointer. Then , all functions are divided into 3 levels:

→ Unit functions are tiny ones which will be applied by other advanced operations

→ Basic functions are classic functions , like Insert&Delete, finding predecessor or successor.

→ Advanced functions are functions which will called by main function directly

Most functions have been discussed in class, so we mainly focus on the constructor and advanced functions

③ Initialize(vector<treeNode*>)

This function will receive a vector as described in main function, and use the vector to build a red-black tree in O(n) time.

Note that we have to finish initialization in O(n) time, so apply insertion n times is not suitable,since that will cost O(nlogn).

So our algorithm is , find the median, then let the median to become the root of the tree, then recursively build the left subtree and right subtree.

Assuming the time complexity of building a tree of size n is T(n)

We got: $T(n) = 2T(n/2) + O(1) \Rightarrow T(N) = O(N)$

And this is where function BuildSubTree(vector<treeNode*> & , int , int , treeNode* , int) come to use.

It got 5 parameters, one is the vector, then 2 integers marking the left boundary and right boundary of nodes we are dealing. For example, after we build the root node, for left subtree, these 2 boundaries should be 0 & (median-1). The pointer to a treeNode is the one pointed to the parent of this subtree, while the last integer is used to marking the depth of the root for this subtree, which will be crucial coloring nodes.

After building the structure of the tree, we have to color all nodes. Note that we only need to color all internal nodes at lowest level red, while the others being black.

To determine whether red or black for a node, we use an extra integer to mark the level. If

pow(2,level+1)-1 > N, then the node must be located at the lowest level, which means they shall be put as red.

④ Inrange(ID1, ID2)

This is actually a 1-D range search, like the Fig 3 shown below.

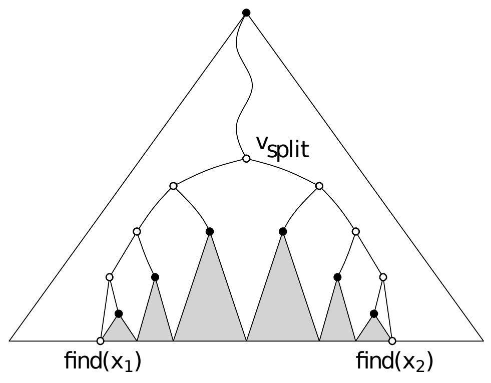


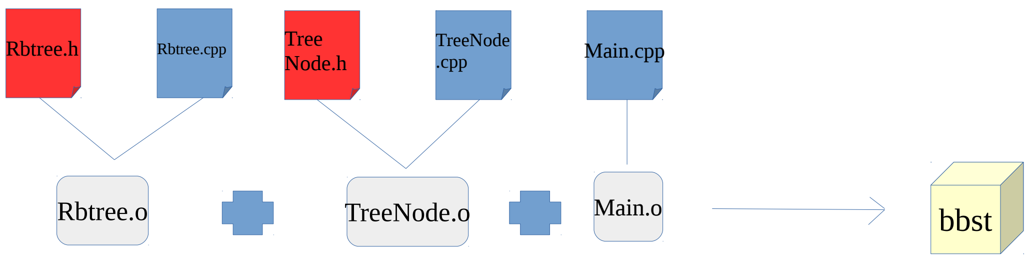
Fig 3 1-D range tree

The Inrange function is actually performed like the pseudocode for 1-D range search:

```
Algorithm 1DRangeQuery(T,[x : x 0 ])
1.  vsplit ←FindSplitNode(T,x,x 0 )
2.  if vsplit is a leaf
3.      then Check if the point in vsplit must be reported.
4.  else v ← lc(vsplit)
5.      while v is not a leaf
6.      do if x ≤ xv
7.          then ReportSubtree(rc(v))
8.          v ← lc(v)
9.      else v ← rc(v)
10.     Check if the point stored in v must be reported.
11. Similarly, follow the path to x 0 , and …
```

5. Conclusion

This is a c++ program, where 2 major class desined, treeNode and RB_Tree.


Rbtree.h
Rbtree.cpp
Tree Node.h
TreeNode .cpp
Main.cpp
Rbtree.o
TreeNode.o
Main.o
bbst