Laboratory Project 2
Is It a Red-Black Tree

Programmer: Zecheng Qian

Tester: Jiahui Shi

Writer: Yingchu Sun

Date: 2017-11-06

1 Introduction

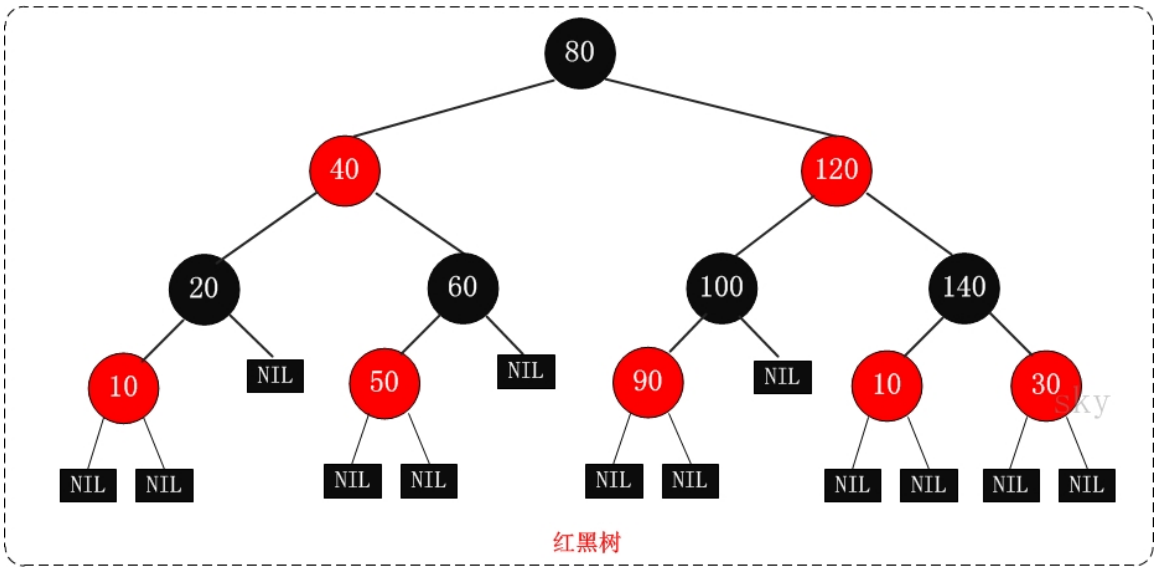
1.1 Problem Description

In this project, we are required to make a program which can tell if a tree is a Red-Black tree. Specifically, we can input file contains several test cases to the program. If the given tree is a red-black tree, the program will print in a line "Yes", or "No" if not.

1.2 Algorithm background

Red-Black Tree is a Self-balancing binary search tree, and its typical purpose is to implement associative arrays.

It was invented by Rudolf Bayer in 1972, when it was called the symmetric binary tree. Later, in 1978, Leo J. Guibas and Robert Sedgewick were revised to today's "Red-Black tree".



Red-Black tree has 5 properties:

(1) Every node is either red or black.

(2) The root is black.

(3) Every leaf (NULL) is black.

(4) If a node is red, then both its children are black.

(5) For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

For this problem, while all the keys in a tree are positive integers, we use negative signs to represent red nodes.

Besides, like AVL tree, the Red-Black tree is able to maintain the balance of the binary search tree by specific operations when inserting and deleting operations, in order to get higher efficiency.

Although it is complex, the running time for the worst case in the Red-Black tree is still very good. And in practice it is efficient: the time complexity for finding、inserting and deleting can be in the O (log n), where n is the number of elements in the tree.

2 Algorithm Specification

2.1 General idea

In line with the idea of modularization, our program is divided into several parts with the following functions:

1>build a binary search tree; (Tree Insert(ElementType a, Tree T);)

2>check if there are continuous red nodes in the binary tree;

(void CheckRedNode(Tree T,int *pflag);)

3> check rule 5 for root node(combine function "CheckPath" and function "PathGenerator");(void CheckRule5(Tree T, int *pflag,int *top, int *BlackNum);)

In general, firstly, the data which is entered by the operator is used for setting up a tree. Then, the program check if there are continuous red nodes in the binary

tree. If the answer is there are continuous red nodes, the flag will be equaled to no, so the program will print “No”. Otherwise, the program will check rule 5 for root node. If it’s not a Red-Black tree, the flag will also be equaled to no. In this way, our program achieve the function of judging whether it is a Red-Black tree.

2.2 pseudo-code

The pseudo-code of main function is following.

```
Function main()
{
    input   K,N;
    for(i from 0 to N-1) //run K times
        input data[i];
    input Ite;
    start=clock();          //begin testing time
    for(i from 0 to N-1) //run K times
        Insert(data[i], T);     //generate a Binary search tree
    CheckRedNode;               //check if there are continuous red nodes
    if flag=Yes         //if the check result is "Yes", continue checking rule 5
        CheckRule5;
       //check if all the paths to leaf node have the same amount of black nodes
    if flag=No or T=NULL or T->color=Red
       //also handle the case that the root's color is red or the tree is empty
        printf("No\n");
     else
        printf("Yes\n");
    stop=clock();                                   //finish one round
    Ticks=(double)(stop-start);
    Duration=Ticks/CLK_TCK;
    print
}
```

The pseudo-code of Tree Insert is following.

```
Tree Insert(ElementType a, Tree T)
{
    input ColorFlag;
    Absolute   a ;          //take the absolute value of a
    if T=NULL
    {               //every insertion the function handle an empty node
        malloc space for T
        input T->color, T->Element;
        T->Left=NULL;
        T->Right=NULL;
    }
    else if a>T->Element
                    //the tree already exists and find the appropriate position to insert
        T->Right=Insert(a, T->Right); //insert into the right subtree recursively
    else if a<T->Element
        T->Left=Insert(a, T->Left);          //insert into the left subtree recursively
    else                                    //the only case is that a==T->Element
        ;                                       //do nothing
    return T;
}
```

The pseudo-code of CheckRedNode is following.

```
CheckRedNode(Tree T,int *pflag)
    //this function only does not examine path and root(rule 5)
{   flag=Yes;                    //flag of judement
    if *pflag=No or T=NULL
    //if it has already confirmed the tree is not a Red Black Tree, then return
        return;
    if T->color==Red {           //handle red nodes
        if T->Left!=NULL and T->Left->color=Red
        {   //if it's left child is also red,return with no
            *pflag=No;
            return;
        }
        if T->Right!=NULL and T->Right->color=Red
        {       //if it's right child is also red,return with no
            *pflag=No;
            return;
        }
        else{                        //this 'else' indicate the case that left and
right nodes are both black or both NULL
            CheckRedNode(T->Left,pflag);
            CheckRedNode(T->Right,pflag);
        }
    }
    return;
}
```

The pseudo-code of CheckRule5 is following.

```
void CheckRule5(Tree T,int *pflag ,int *top, int *BlackNum)
{
    if *pflag=No                //only when flag==Yes needs further checking
        return;
    if T=NULL
        return;
    else{
        *top=0;
        PathGenerator(T,0,top,BlackNum);
            //generates numbers of black nodes of all paths for the root node
        CheckPath(pflag, top, BlackNum);
    }
    return;
}
```

3 Testing Results

3.1 Sample

```
3
9
7 -2 1 5 -4 -11 8 14 -15
Yes
9
11 -2 1 -7 5 -4 8 14 -15
No
8
10 -7 5 -6 8 15 -11 17
No
```

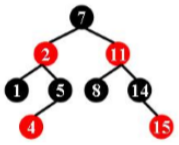


Figure 1

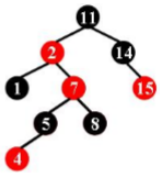


Figure 2

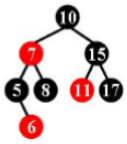


Figure 3

Testing the sample case is to make sure that the program has no obvious mistakes.

When testing the sample, it works well. The output is identical with sample output.

In the second case, a red node whose element is 2 has a red child node. It breaks rule 4.

In the third case, three simple path from root to descendant leaf contain 3 black nodes, while the other two paths contain 4 black nodes. It breaks rule 5.

3.2 There is only one node in the tree

```
3
1
1
Yes
1
-1
No
1
0
No
```

Testing this case is to make sure that rule 1 and 2 has been checked.

In the first case, there is only a black root node, it is a legal red-black tree.

In the second case, there is only a red root node, it breaks rule2.

In the third case, there is only node being neither red nor black, it breaks rule 1.

3.3 There is 30 nodes(maximum)

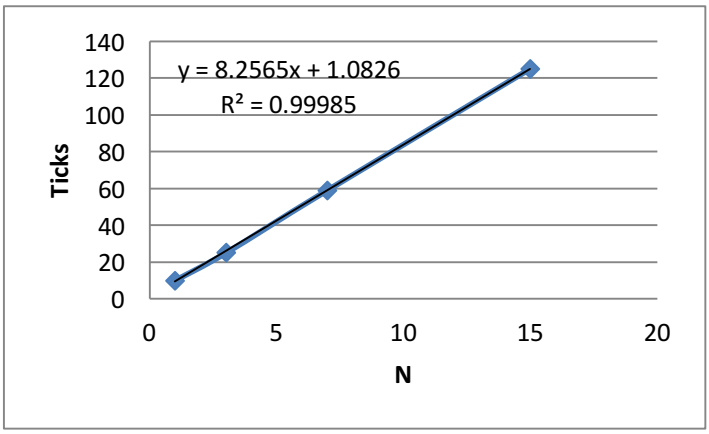
Testing this case is to make sure there is enough space allocated to variables, and the result shows it works well.

```
1
30
16 -8 4 -2 1 3 -6 5 7 12 -10 9 11 -14 13 15 -24 20 -18 17 19 -22 21 23 28 -26 25
 27 -30 29
No
```

3.4 Time complexity test

Theoretically checking whether input is a red-black tree program consumes O(n) time (the reason will be explained in next chapter), so when the inputs are all Red-Black Trees, T-N curve should be linear.

Iteration	N	Ticks
50000	1	10
	3	25
	7	50
	15	125



4 Analysis and Comments

4.1 Time complexity

1)Creating a tree

The program uses recursive method to create a tree. N elements are inserted one by one, and each iteration takes up constant time.

The worst case is Skewed Binary Tree, the ith element takes up i*O(1) time to be inserted into its place. Therefore, the total time of inserting all elements should be

$$T(N)=O(1)*O(1)*\sum_{i=1}^{N} i=O(N^2).$$

Considering the average case, the elements of d-depth take up (d+1)*O(1) time to find its place, and number of nodes of d-depth is O(2^d). Therefore, the time complexity should be

$$T(N)=\sum_{i=1}^{d}(i+1)*O(1)*O(2^i)=O(d*2^d)=O(N*\log N).$$

2)Judgment

The time complexity of checking the rule 4 is O(N), because function CheckRedNode traverses every node it uses once before loop breaks.

The time complexity of checking the rule 4 is O(N) too. Function CheckRule5 calls function PathGenerator and CheckPath. PathGenerator traverses every node in the tree, so it takes up O(N) time. CheckPath use iterative method to compare numbers of black nodes in each simple path, the number of path is O(log N), so it takes up O(log N) time.

If the input is a legal tree, it goes through function CheckRedNode and CheckRule5, time complexity is sum of that of two functions, that is, O(n).

4.2 Space complexity

1)Creating a tree

It allocates space for all the nodes of the tree, so its space complexity is O(N).

2)Judgment

This part uses recursive method to check rule 4 and 5, and the traversing way is post-order, so there are at most (d+1) nodes is stored at the same time for a tree of d-depth.

In the average case, d is O(log N) size. The worst case is Skewed Binary Tree, counting black node of every simple path takes up space of O(n).

4.3 Comments

It is wise to check rule 4 before rule 5 is wise, because function CheckRule5 takes O(N) time regardless of whether it is a red-black tree.

As for red-black tree, it is a kind of data structure which easily get balanced. the longest simple path is no longer than twice of the shortest simple path. So traversing a red-black tree is time saving, its time complexity is O(log n), as well as space complexity.

5 Appendix: Source Code in C

```
int main(void)
{
    double Duration;
    Tree T;                                 //point to the root of the tree
    ElementType data[MAXLENGTH];
    //this array stores datas that users input
    int i, j, K, N, flag;
//N represent the number of datas, K tree(s) to judge,flag:result of judging(Yes or No)
    int BlackNum[MAXSIZE];
//this array stores numbers of black nodes of all paths for one node
    int topofstack;                         //topofstack points to the top path elements
    int *top=&topofstack;
    int *pflag=&flag;
    scanf("%d", &K);
    while(K>0){
    scanf("%d", &N);
    for(i=0;i<N;i++){
        scanf("%d", data+i);
    }
    T=NULL;
    flag=Yes;                               //assume it is a red-black tree
    for(i=0;i<N;i++)
        T=Insert(data[i], T);               //generate a Binary search tree
   CheckRedNode(T,pflag);                       //check if there are continuous red nodes
    if(flag==Yes)
        //if the check result is "Yes", continue checking rule 5
        CheckRule5(T,pflag,top,BlackNum);
          //check if all the paths to leaf node have the same amount of black nodes
    if(flag==No||T==NULL||T->color==Red)
         //also handle the case that the root's color is red or the tree is empty
        printf("No\n");
    else
        printf("Yes\n");
    K--;
    }
    system("pause");
    return EndWithoutError;                         //end successfully
}
```

```
Tree Insert(ElementType a, Tree T)
{
    NodeColor ColorFlag;
    ElementType Absolute_a;
    Absolute_a=a<0? -a:a;
                        //take the absolute value of a
    ColorFlag = a>0?Black:Red;
                      //judge the color of node to be inserted
    if(T==NULL){
                        //every insertion the function handle an empty node
        T=(Tree) malloc(sizeof(struct TreeNode));
        if(T==NULL)                                  //exception handling
            printf("Out of space!");
        else{
            T->color=ColorFlag;                  //initialize the new node
            T->Element=Absolute_a;
            T->Left=NULL;
            T->Right=NULL;
        }
    }
    else if(Absolute_a>T->Element)
         //the tree already exists and find the appropriate position to insert
        T->Right=Insert(a, T->Right);        //insert into the right subtree
recursively
    else if(Absolute_a<T->Element)
        T->Left=Insert(a, T->Left);          //insert into the left subtree
recursively
    else                                    //the only case is that a==T->Element
        ;                                       //do nothing
    return T;
}
```

```
void CheckRedNode(Tree T,int *pflag)
                //this function only does not examine path and root(rule 5)
{
     static int flag=Yes;                    //flag of judement
     if(*pflag==No)
//if it has already confirmed that the tree is not a Red Black Tree, then return
          return;
     else if(T==NULL){
     //T==NULL indicates the current path has gone out,then return
          return;
     }
     else if(T->color==Red){               //handle red nodes
          if(T->Left!=NULL&&T->Left->color==Red){
        //if it's left child is also red,return with no
               *pflag=No;
               return;
          }
          else if(T->Right!=NULL&&T->Right->color==Red){
            //if it's right child is also red,return with no
               *pflag=No;
               return;
          }
          else{                          //this 'else' indicate the case that left and right
nodes are both black or both NULL
               CheckRedNode(T->Left,pflag);
               CheckRedNode(T->Right,pflag);
          }
     }
     else{
              //this 'else' indicate the case that the current node is black
               CheckRedNode(T->Left,pflag);
               CheckRedNode(T->Right,pflag);
     }
     return;
}
```

```
void PathGenerator(Tree T, int BlackCount, int *top, int *BlackNum)

            //generate all paths of T to all leaf nodes(black)

{
    if(T==NULL){                    //T==NULL indicates the current path has gone out
        BlackCount++;
                                //because 'NULL' represent a null black node,so it should be '++'
        BlackNum[*top]=BlackCount;    //write the number of black nodes in to the stack
        (*top)++;                        //finish one path and topofstack++
        return;
    }
    else if(T->color==Black)          //if it's a black node
        BlackCount++;
    PathGenerator(T->Left, BlackCount, top, BlackNum);      //recurse
    PathGenerator(T->Right, BlackCount, top, BlackNum);     //recurse
    return;
}

void CheckPath(int *pflag, int *top, int *BlackNum)
                    //examine if all numbers of black nodes in paths are the same(rule 5)
{
    int i;
    for(i=0;i<*top;i++){
        if(BlackNum[0]!=BlackNum[i])
            break;
    }
    if(i<*top)                  //if 'break' happens in advance,then return with flag=No
        *pflag=No;
    return;
}

void CheckRule5(Tree T,int *pflag ,int *top, int *BlackNum)
{
    if(*pflag==No)                              //only when flag==Yes needs further checking
        return;
    if(T==NULL)
        return;
    else{
        *top=0;
        PathGenerator(T,0,top,BlackNum);
                            //generates numbers of black nodes of all paths for the root node
        CheckPath(pflag, top, BlackNum);
    }
    return;
}
```

Declaration:

We hereby declare that all the work done in this project titled "Is It a Red-Black Tree " is of our independent ef fort as a group.

Duty Assignments:

Programmer: Zecheng Qian

Tester: Jiahui Shi

Writer: Yingchu Sun