Is It A Red-Black Tree

Zhang Youchao

3170100125

Date:2020-10-25

Chapter 1: Introduction

There is a kind of balanced binary search tree named red-bla ck tree in the data structure. It has the following 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 descend ant leaves contain the same number of black nodes.

For each given binary search tree, you are supposed to tell if it is a legal red-black tree. My algorithm is based on the property (2)(4)(5).

According to (3) The root should be black node.

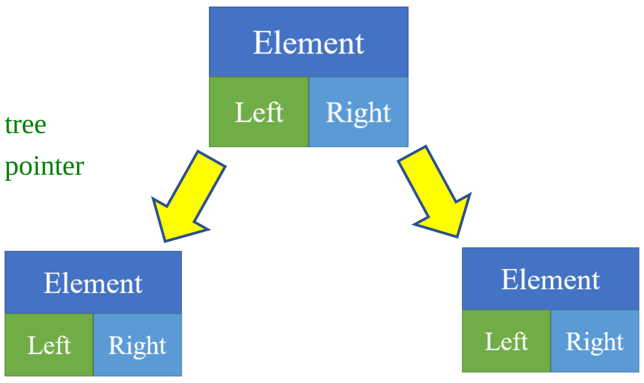
According to (4) : If a node is red, then both its children must be black.

According to (5) :For each node, all simple paths from the node t o descendant leaves contain the same number of black nodes.A red blac k tree can make the black node balanced , which means the left child a nd the right child must have the same depth(we just care about the blac k node)

Chapter 2: Algorithm Specification

1.the data structure

```
/*define the node*/
typedef struct Node *tree;//define the tree
typedef tree ptr;          //define the pointer
struct Node
{
    int Element;
    ptr Left;
    ptr Right;
};
```



2.According to (3) The root should be black node.

```
Algorithm1:Check root
Input  :  tree T
Output :  0→wrong   1→right
1:  function checkroot( tree T )
2:      if T==NULL return 1
3:      else if T→element>0 return 1
4:      else return 0
5:  end
```

```c
/*According to 3 The root should be black node.*/
int checkroot(tree T)
{
    if (!T)return 1;
    else if (T->Element > 0) return 1;
    else return 0;
}
```

3.According to (4) : If a node is red, then both its children must be black.

```
Algorithm2:Check children
Input  :  tree T
Output :  0→wrong   1→right
1:   function red_with_2black( tree T )
2:       if T!=NULL
3:           if T→element<0
4:               if T→left! =NULL and T→left→element<0
5:                   return 0
6:               else if T→right! =NULL and T→right→element<0
7:                   return 0
8:               end if
9:               return red_with_2black(T→Left)//recursion
10:              return red_with_2black(T→Right)
11:      else return 1
12:  end
```

```c
/*According to 4 : If a node is red, then both its children must be black.*/
int red_with_2black(tree T)
{
    if (T)
    {
        if (T->Element < 0)
        {
            if (T->Left)
            {
                if (T->Left->Element < 0)return 0;
            }
            if (T->Right)
            {
                if (T->Right->Element < 0)return 0;
            }
        }
        return red_with_2black(T->Left);//check leftchild
        return red_with_2black(T->Right);//check rightchild
    }
    else
        return 1;
}
```

4.According to (5) :For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.A red black tree can make the black node balanced, which means the left child an d the right child must have the same depth(we just care about the black n ode)

```
Algorithm3:  equal_black
Input    :   tree  T
Output  :   0→wrong    1→right
   1:   function equal_black(tree T)
   2:       if (!T)
   3:           return 1
   4:       int left_black = find_blackNode_depth(T->Left)
   5:       int right_black = find_blackNode_depth(T->Right)
   6:       if (left_black == right_black)  return 1
   7:       else  return 0
   8:       /*Recursion*/
   9:       if (T->Left)
  10:           return equal_black(T->Left)
  11:       if (T->Right)
  12:           return equal_black(T->Right)
  13:   end
```

/*According to 5 :For each node, all simple paths from the node to descendant leaves contain the s ame number of black nodes. A red black tree can make the black node balanced, which means th e leftchild and the rightchild must have the same depth(we just care about the black node)
*/

```
int equal_black(tree T)
{
    if (!T)
        return 1;
  /*find the leftchild and rightchild black node depth*/
    int left_black = find_blackNode_depth(T->Left);
    int right_black = find_blackNode_depth(T->Right);
   /*the black node depth must equal to each other*/
    if (left_black == right_black)
        return 1;
    else
        return 0;
    /*Recursion*/
    if (T->Left)
        return equal_black(T->Left);
    if (T->Right)
        return equal_black(T->Right);
}
```

5.Find the depth of node T to the leaf .And the depth is calculate by the black node.UsePostorder traversal, find the black node height of the current tree.

Algorithm4: find_blackNode_depth

```
Input  :    tree T
Output :    the number of black node on the deepth
  1:    function find_blackNode_depth(tree T)
  2:        if (!T)
  3:            return 0
  4:        int left_black = find_blackNode_depth(T->Left)
  5:        int right_black = find_blackNode_depth(T->Right)
  6:        if (T->Element > 0) /*Black Node*/
  7:            if (left_black > right_black)
  8:                return left_black + 1
  9:            else
 10:                return right_black + 1
 11:        else /*red node*/
 12:            if (left_black > right_black)
 13:                return left_black
 14:            else  return right_black
 15:    end
```

```
/*Find the deepth of node T to the leaf And the deepth is calculate by the black node.
Post-order traversal, find the black node height of the current tree.*/
int find_blackNode_depth(tree T)
{
    if (!T)
        return 0;
    int left_black = find_blackNode_depth(T->Left);
    int right_black = find_blackNode_depth(T->Right);

    if (T->Element > 0) /*Black Node*/
    {
        if (left_black > right_black) /*The bigger number is the depth,rather than t
he smaller,so we should return the bigger one*/
        {
            return left_black + 1;
        }
        else
        {
            return right_black + 1;
        }
    }
    else /*red node*/
    {
        if (left_black > right_black)
        {
            return left_black;
        }
        else
        {
            return right_black;
        }
    }
}
```

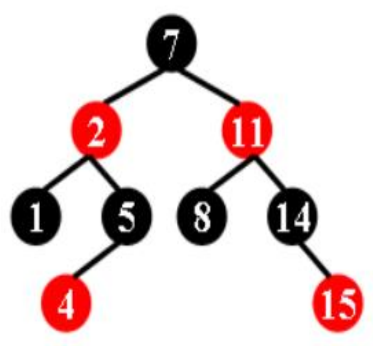
6.The main loop:

First read the number of binary trees, then read the number of nodes of the first binary tree, and then read the traversal sequence of the binary tree. Then read the number and sequence of nodes in turn and judge whether the binary tree is a legal red-black tree in real time. If it is, print "YES", if it is illegal, print "NO".

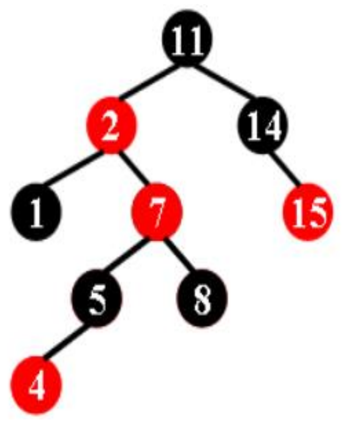
```
int main()
{
    int num = 0;//The number of testing tree
    scanf("%d", &num);
    for (int i = 0; i < num; i++)
    {
        tree T = NULL;
        int number_of_node = 0;
        scanf("%d", &number_of_node);
        for (int j = 0; j < number_of_node; j++)
        {
            int data = 0;
            scanf("%d", &data);
            T = Insert(data, T);
        }
        if (checkroot(T) && red_with_2black(T) && equal_black(T))
//To judge the temp tree use the 3 algorithm.
            printf("YES\n");
        else
            printf("NO\n");
        FreeTree(T); //To free the memory
    }
    system("pause");
    return 0;
}
```

Chapter 3: Testing Results

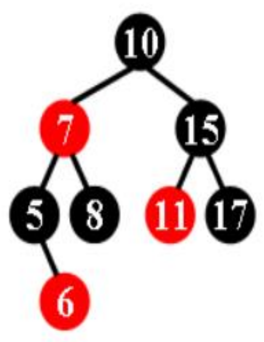
Example:



Yes



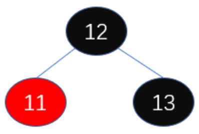
No

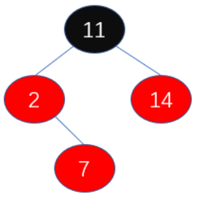


No

Result table:

Test case	a brief description of the purpose	Expected result	Actual behavior	Possible cause of a bug	Current status
3 9 7 -2 1 5 -4 -11 8 14 -1	To test whether the program can detect the input given	Yes No	Yes No	No	Pass

5 9 11 -2 1 -7 5 -4 8 14 -1 5 8 10 -7 5 -6 8 15 -11 17	by the project	No	No		
1 1 -1	To test whether th e program can jud ge by the root (property 2)	No The root m ust be black	No	No	Pass
30 30 1 2 3 4 …30 …. 30 1 2 3 4 … 30	To test whether th e program will out of memory.	No … No	No … No	No	Pass
1 0 	To test whether th e program can jud ge by the root (property 2)	Yes NULL is a kind of b lack node	Yes	No	Pass
1 3 12 -11 13 	To test whether th e code can judge by the Property 5: For each node, all simple paths from the node to desce ndant leaves contai n the same numbe r of black nodes.	No 12 -11 13 the path from the node 12 to the leaf has different black node 12→-11→N has 2 black nodes 12→13→N has 3 black nodes	No	No	Pass

1 4 11 -2 -14 -7 	To test whether the code can judge by the Property 4: If a node is red, then both its children are black.	No In the tree, 11→2→7→NULL both 2 and 7 are the red node, so according to property 4, the tree is not legal	No	NO	Pass
1 10 1 2 3 4 5 -6 -7 -8 -9 -10		No	No	No	Pass
2 5 1 2 -3 -4 5 7 1 -4 -6 2 5 3 7		No No	No No	No	Pass
2 5 1 2 -3 -4 5 7 -1 -4 -6 2 5 3 7		No No	No No	No	Pass
1 8 -1 -4 -6 2 5 3 7 -9 -8		No	No	No	Pass
1 11 1 2 3 4 5 6 7 8 9 10 11		No	No	No	Pass
1 11 11 10 9 8 7 6 5 4 3 2 1		No	No	No	Pass
1 10 -1 2 -3 4 -5 6 -7 8 -9 -10		No	No	No	Pass

Chapter 4: Analysis and Comments

Analysis of the time and space complexities of the al gorithms. Comments on further possible improvements.

1. **Algorithm1** Check root, we can see that both the time complexity and space com plexity is O(1), because we just need to judge the root of the tree.
2. **Algorithm2,** Check children, after analysis the code we can find that the function Check children is a kind of preorder traversal, we need to check every node, so t he time complexity is O(N). And we don't need extra space, so the space comple xity is O(1)
3. **Algorithm3** equal_black and **Algorithm4** find_blackNode_depth are used together i n function equal_black. In the function we used find_blackNode_depth function for every node, and the time complexity of it is O(logN), so the total time complexi ty is **O(NlogN)**, And we don't need extra space, so the space complexity is O(1). When we build the tree the space complexity is O(N)
4. All in all the total time complexity is **O(NlogN)**, the space complexity is O(N)

Further possible improvements

1. If the node→element is 0，we don't know whether it is red node or black node. So we should define it as black node.
2. The writing of function names and other code specifications can be improved, and when time permits, other methods can be devised to solve the practical problem.
3. Algorithm is implement by the recursion method , we can reimplement them by s ome basic data structure such as stack and queue. Because the efficiency of recurs ive programs is lower than that of loop programs

Appendix: Source Code (in C)

At least 30% of the lines must be commented. Otherwise th e code will NOT be evaluated.

```
#include <stdio.h>
#include <stdlib.h>

/*define the node*/
typedef struct Node *tree;//define the tree
typedef tree ptr;         //define the pointer
struct Node               //define the node
{
    int Element;
    ptr Left;
    ptr Right;
};

/*calculate the absolute value */
int abs(int a)
{
    if (a < 0)
        a = -1 * a;
    return a;
}

/*to build the tree*/
tree Insert(int X, tree T)
{
    if (T == NULL)
    { /* Create and return a one-node tree */
        T = malloc(sizeof(struct Node));
        if (T == NULL)
            printf("Out of space!!!\n");
        else
        {
            T->Element = X;
            T->Left = T->Right = NULL;
        }
    }   /* End creating a one-node tree */
    else /* If there is a tree */
    {
        if (abs(X) < abs(T->Element))
        {
            T->Left = Insert(X, T->Left);
```

```c
        }
        else if (abs(X) > abs(T->Element))
        {
            T->Right = Insert(X, T->Right);
        }
    }
    /* Else X is in the tree already; we'll do nothing */
    return T;
}

/*free tree by postorder*/
void FreeTree(tree T)
{
    if (T)
    {
        FreeTree(T->Left);
        FreeTree(T->Right);
        free(T);
        T = NULL;//let the pointer point to NULL
    }
}

/*
*****************************************************
************Use 2,4,5 to check the tree**************
*****************************************************
*/
/*According to 3 The root should be black node.*/
int checkroot(tree T)
{
    if (!T)//black node
        return 1;
    else if (T->Element > 0)//black node
        return 1;
    else//red node
        return 0;
}
/*
According to 4 : If a node is red, then both its children must be blac
k.
*/
int red_with_2black(tree T)
{
    if (T)
    {
        if (T->Element < 0)//red node
        {
            if (T->Left)
            {
                if (T->Left->Element < 0)//red node
                    return 0;
            }
            if (T->Right)
            {
                if (T->Right->Element < 0)//red node
                    return 0;
            }
        }
        return red_with_2black(T->Left);
        return red_with_2black(T->Right);
    }
    else//NULL black node
        return 1;
}

/*According to 5 :For each node, all simple paths from the node to desc
endant leaves contain the same number of black nodes.
So a red black tree can make the black node balanced, the leftchild an
d the rightchild must have the same depth(we just care about the blac
k node)*/
int equal_black(tree T)
{
    if (!T)
        return 1;

    /*find the leftchild and rightchild black node depth*/
    int left_black = find_blackNode_depth(T->Left);
    int right_black = find_blackNode_depth(T->Right);
    /*the black node depth must equal to each other*/
    if (left_black == right_black)
        return 1;
    else
        return 0;

    /*Recursion*/
    if (T->Left)
        return equal_black(T->Left);
    if (T->Right)
```

```
        return equal_black(T->Right);
}
/*Find the deepth of node T to the leaf And the deepth is calculate b
y the black node.
Post-order traversal, find the black node height of the current tree.*/
int find_blackNode_depth(tree T)
{
    if (!T)
        return 0;
    int left_black = find_blackNode_depth(T->Left);
    int right_black = find_blackNode_depth(T->Right);

    if (T->Element > 0) /*Black Node*/
    {
        if (left_black > right_black) /*The bigger number is the depth,
rather than the smaller,so we should return the bigger one*/
        {
            return left_black + 1;
        }
        else
        {
            return right_black + 1;
        }
    }
    else /*red node*/
    {
        if (left_black > right_black)
        {
            return left_black;
        }
        else
        {
            return right_black;
        }
    }
}
/*
******************************************************
************************END**************************
******************************************************
*/
/* to judge the tree */
int judge(tree T)
{
```

```c
    if (checkroot(T) && red_with_2black(T) && equal_black(T))
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

int main()
{
    int num = 0;//The number of testing tree
    scanf("%d", &num);
    for (int i = 0; i < num; i++)
    {
        tree T = NULL;
        int number_of_node = 0;
        scanf("%d", &number_of_node);
        for (int j = 0; j < number_of_node; j++)
        {
            int data = 0;
            scanf("%d", &data);
            T = Insert(data, T);
        }
        if (judge(T))//To judge the temp tree
        {
            printf("YES\n");
        }
        else
        {
            printf("NO\n");
        }
        FreeTree(T); //To free the memory
    }
    system("pause");
    return 0;
}
```

Declaration

I hereby declare that all the work done in this project titled " Is It A Red-Black Tree " is of my independent effort.