Is It A Red-Black Tree

Name

Date

Chapter 1：Introduction

1 Red-black tree

It’s a kind of balanced binary search tree and has the following 5 properties:

- Every node is either red or black.
- The root is black.
- Every leaf(NULL) is black.
- If a node is red, then both its children are black.
- For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

2 Problem description

Each **input** file contains several test cases:

- The first line gives a positive integer **K (≤30)** which is the total number of cases.
- For each case, the first line gives a positive integer **N (≤30)**, the total number of nodes in the binary tree. The second line gives the **preorder** traversal sequence of the tree (numbers are separated by a space). While all the keys in a tree are positive integers, we use **negative signs to represent red nodes.**

Output: for each test case, print in a line "Yes" if the given tree is a red-black tree, or "No" if not.

3 Meaning of Red-black Tree

In AVL tree, because AVL tree is absolutely balanced, in order to maintain its absolute balance when inserting and deleting, sometimes node modification operations need to be carried out to the root node, and the number of rotations is relatively large, so there is a red-black tree. When data is not static data but **dynamic data**, There is no need to maintain absolute balance when inserting and deleting, which reduces the number of rotations, but also **improves efficiency**, and the average search efficiency of red-black trees is **log2 (n)**.

Chapter 2: Algorithm Specification

1 main data structure

I defined a structure named **TreeNode**, where **flag** is used to mark its color (1:black; 0:red), and **count** is for the number of the black nodes from this node to leaf (if the number of different leaves are distinct, then this tree is not a red-black tree, marked in **EN**, and then count make no sense).

```
typedef struct TreeNode *Tree;
struct TreeNode {
    int value;  /* the value of this node */
    int flag;   /* if the flag == 1, it's a black node, else it's a red node */
    int count;  /* to count the number of black nodes from leaves to this node */
    Tree  Left;
    Tree  Right;
};
```

2 main function

My program consists of a **main** function and **five functions**, where the main function reads the total number of cases (K), builds and determines whether it is a red-black tree (implemented by the function), and prints the results uniformly.

```
function main():
    read(K)

    for i from 0 to k-1:
        T <- NULL
        EN <- 1
        createTree()
        JudgeRBT()

    for i from 0 to k-1:
        print(result[i])

    return 0
```

3 createTree & addNode

These two functions are to **create a binary tree** with given n numbers. In **createTree**, the program assigns the new node an address and initializes it, then reads in the number and determines whether it is red or black (marked with a flag), and finally uses the **addNode** function to insert the new node into the binary tree. The **addNode** function, on the other hand, uses a **recursive** algorithm to continuously compare the size of the current node with that of the node to be added until it finds a vacant location.

```
function createTree():
    read(N)

    for j from 0 to N-1:
        node <- createNode()
        node.flag <- 0
        node.Left <- NULL
        node.Right <- NULL
        node.count <- 0

        read(number)
        if number > 0:
            node.flag <- 1
        else:
            number <- -number
        node.value <- number

        T <- addNode(T, node)
```

```
function addNode(T, node):
    if T is NULL:
        return node
    else:
        if node.value < T.value:
            T.Left <- addNode(T.Left, node)
        else:
            T.Right <- addNode(T.Right, node)
    return T
```

4 JudgeRBT & Blackson & SameBN

These three functions are to **judge whether the binary tree is a red-black tree**. In function **JudgeRBT**, I use **three conditions** to determine whether it is a red-black tree: all sons of red nodes are black nodes, for each node all simple paths from the node to descendant leaves contain the same number of black nodes, and the root is black. The function **Blackson** uses a **recursive** algorithm to determine the first condition: for a red node, if there is a child node that is not black, it is determined that this tree is not a red-black tree (empty nodes, i.e. leaves, are black nodes). The function **SameBN** also uses a **recursive** algorithm. First calculate the **count** of current node with its left subtree (if it has no left subtree, it has a black leaf, namely NULL). Then compare it with that of right subtree. If they are different, change **EN** to zero (which means it disagrees with 2nd condition).

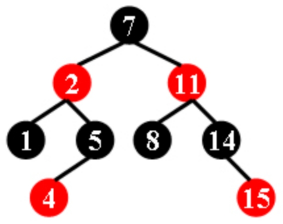
```
function JudgeRBT():
    if not BlackSon(T):
        c[i] <- "No"
        return

    if not (SameBN(T) and EN):
        c[i] <- "No"
        return

    if not T.flag:
        c[i] <- "No"
        return

    c[i] <- "Yes"
```

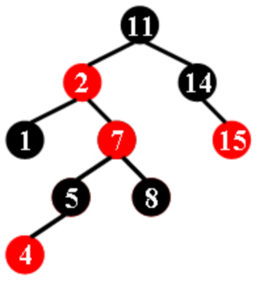
```
function BlackSon(T):
    if T is NULL:
        return 1

    if not T.flag:
        if T.Left is NULL:
            return 1
        if not T.Left.flag:
            return 0
        if T.Right is NULL:
            return 1
        if not T.Right.flag:
            return 0

    if BlackSon(T.Left) and BlackSon(T.Right):
        return 1

    return 0
```

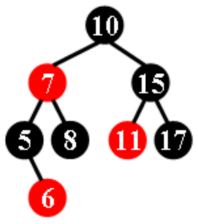
```
function SameBN(T):
    if T.Left is NULL:
        T.count <- T.flag + 1
    else:
        T.count <- T.flag + SameBN(T.Left)

    if T.Right is NULL:
        if T.count != T.flag + 1:
            EN <- 0
    else:
        if T.count != T.flag + SameBN(T.Right):
            EN <- 0

    return T.count
```

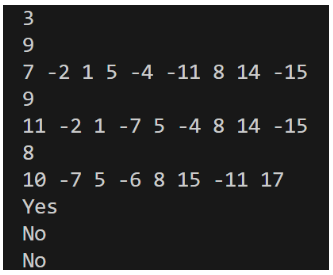
Chapter 3: Testing Results

Here are the three test samples given by the PTA (omit black NULL leaf nodes), and the **expected** outputs are **Yes**, **No** (red node 2 has red son 7), and **No** (the number of black nodes from node –such as 15– to leaves are distinct) respectively. Besides, I add some cases (the binary tree of a case with N=30 is posted, and the expected output is **No**). And the **running results** are posted.

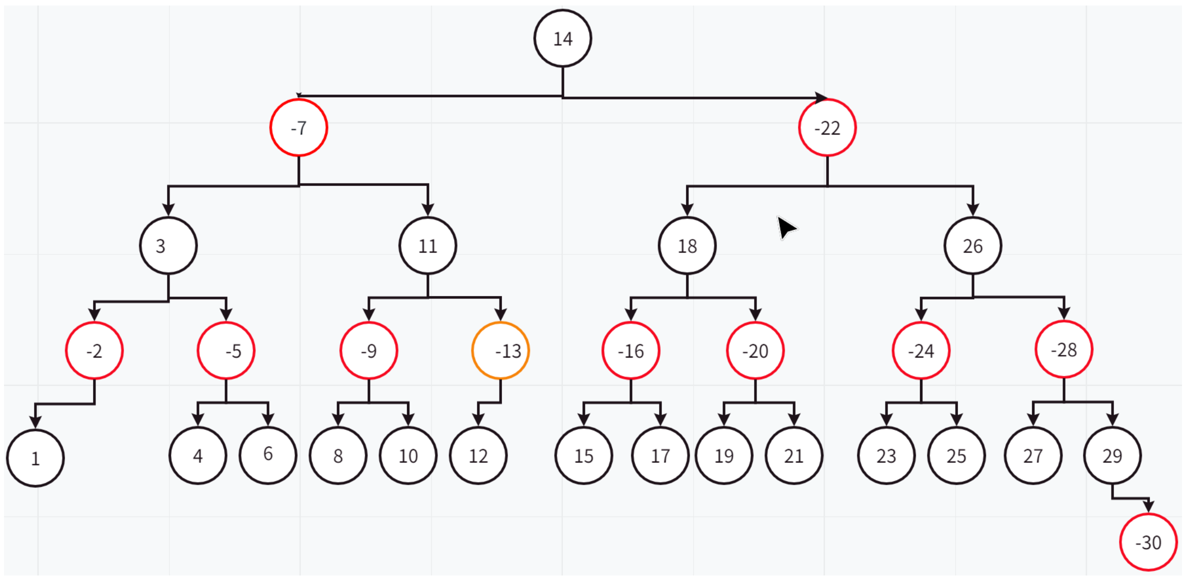


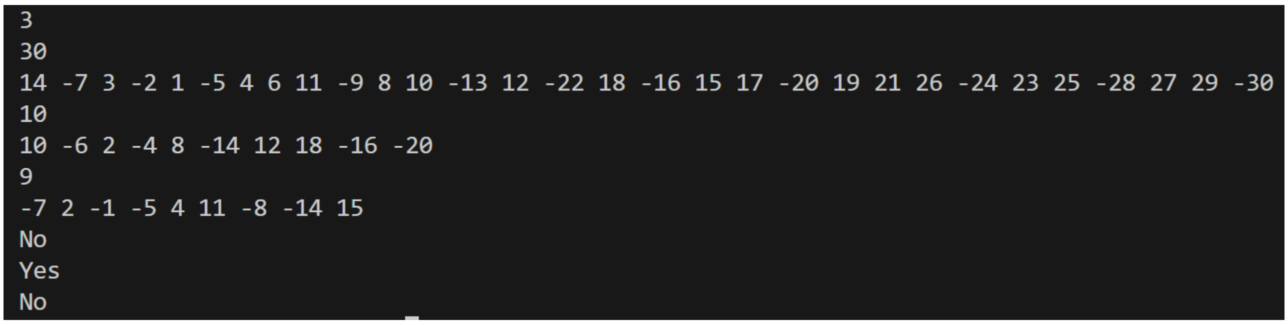






```
3
9
7 -2 1 5 -4 -11 8 14 -15
9
11 -2 1 -7 5 -4 8 14 -15
8
10 -7 5 -6 8 15 -11 17
Yes
No
No
```





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

Chapter 4: Analysis and Comments

1 time complexities

createTree function:

The number of iterations of the loop is N, so the time complexity of the loop is O(N).
In each iteration, the node insertion operation (**addNode** function) may need to traverse the entire tree in the worst case, with a time complexity of O(N).

Therefore, the total time complexity of createTree is O(N^2).

JudgeRBT function:

Both **BlackSon** function and **SameBN** function have a time complexity of O(N) because they may need to traverse the entire tree.

main function:

In the main function, first iterate K times, each time calling the createTree and JudgeRBT functions, so the time complexity of the main function is O(K * N^2).

Total time complexity:

O(K * N^2), where K is the number of cases and N is the number of nodes in the binary tree.

2 space complexities

createTree function:

A dynamically allocated node is used, so the space complexity depends on the number of dynamically allocated nodes, and it's O(N).

Other function:

There is no significant additional space footprint, except stack frames.

Total space complexity:

O(N), where N is the number of binary tree nodes.

3 possible improvement

The binary tree created in my program contains many unused null Pointers (actually N+1). You can use these Pointers by creating a **Threaded Binary Tree**.

Chapter 5: Source Code (in C)

```c
#include <stdio.h>
#include <stdlib.h>
#define MAX 35
typedef struct TreeNode *Tree;
struct TreeNode {
    int value;  /* the value of this node */
    int flag;   /* if the flag == 1, it's a black node, else it's a red node */
    int count;  /* to count the number of black nodes from leaves to this node */
    Tree  Left;
    Tree  Right;
};
Tree T;     /* the root of binary tree */
int i;      /* the number of the trees */
char *c[MAX];   /* result of judge */
int EN;     /* flag for condition 2 */

void createTree(); /* function: create the binary tree with N numbers given */
Tree addNode(Tree T, Tree node); /* function: add new node into the binary tree */
void JudgeRBT(); /* function: judge whether it's a red-black tree and store the result */
int BlackSon(Tree T); /* condition 1: for each red node, both its children are black */
int SameBN(Tree T); /* condition 2: For each node, all paths from it to leaves contain the same number of black nodes.*/

int main(void)
{
    int K;      /* the total number of cases, K <= 30 */
    scanf("%d",&K);
    for(i = 0;i < K;i++){
        T = NULL;       /* initialize the head pointer */
        EN = 1;
        createTree(); /* create the binary tree */
        JudgeRBT();     /* judge whether it's a red-black tree and store the result */
    }
    for(i = 0;i < K;i++)/* print the result */
        printf("%s\n",c[i]);
    return 0;
}
void createTree() /* function: create the binary tree with N numbers given */
{
    int j, number;
    int N;      /* the total number of nodes in binary tree, N <= 30 */

    scanf("%d",&N);
    // printf("N:%d\n",N);//    all codes for test are marked by '//'
    for(j = 0;j < N;j++){
        Tree node = (Tree)malloc(sizeof(struct TreeNode));
        node->flag = 0;     /* assume it's a red node */
        node->Left = NULL;  /* initialization */
        node->Right = NULL;
        node->count = 0;
        scanf("%d",&number);/* put in the number */
        // printf("put in:%d\n",number);//
        if(number > 0)  node->flag = 1;     /* flag == 1 means it's the black node */
        else            number = -number;   /* keep 'number' positive */
        node->value = number;
        T = addNode(T, node);   /* add this node into the binary tree */
    }
}
Tree addNode(Tree T, Tree node) /* function: add new node into the binary tree */
{
    if(T == NULL)   /* this position is empty, and add node here */
        return T = node;
    else{   /* search the position for the number */
        if(node->value < T->value) T->Left = addNode(T->Left, node);
        else    T->Right = addNode(T->Right, node);
    }
    return T;
}
void JudgeRBT() /* function: judge whether it's a red-black tree and store the result */
{
    if(!BlackSon(T)){
        c[i] = "No";
        return;
    }
    // printf("condition 1: done.\n");//
    if(!(SameBN(T) && EN)){
        c[i] = "No";
        return;
    }
    if(!T->flag){   /* condition 3: the root is black node */
        c[i] = "No";
        return;
    }
    c[i] = "Yes";
}
int BlackSon(Tree T) /* condition 1: for each red node, both its children are black */
{
    if(T == NULL) return 1;
    if(!T->flag){
        // printf("judge:%d\n",T->value);//
        if(T->Left == NULL)     return 1;   /* leaf(NULL) is black node */
        if(!T->Left->flag) return 0;
        if(T->Right == NULL)    return 1;
        if(!T->Right->flag)return 0;
    }//else printf("non_judge:%d\n",T->value);//
    if(BlackSon(T->Left) && BlackSon(T->Right)) return 1;
    return 0;
}
int SameBN(Tree T) /* condition 2: For each node, all paths from it to leaves contain the same number of black nodes.*/
{
    if(T->Left == NULL) T->count = T->flag + 1;
    else                T->count = T->flag + SameBN(T->Left);
    if(T->Right == NULL){
        if(T->count != T->flag + 1) EN = 0; /* once count_left != count_right, EN = 0 */
    }else if(T->count != T->flag + SameBN(T->Right)) EN = 0;
    // printf("value:%d,count:%d\n",T->value,T->count);//
    return T->count;
}
```

Declaration

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