Project 1 Is It A Red-Black Tree

Chapter 1 Introduction

There is a kind of balanced binary search tree named **red-black 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 descendant leaves contain the same number of black nodes.

  For example, the tree in Figure 1 is a red-black tree, while the ones in Figure 2 and 3 are not.

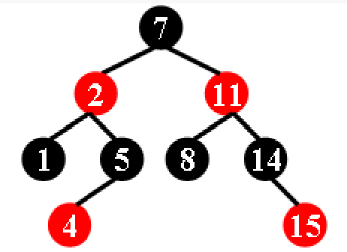


Figure 1

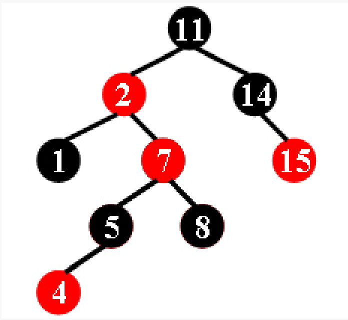


Figure 2

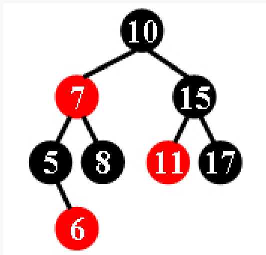


Figure 3

For each given **binary search tree**, you are supposed to tell if it is a legal red-black tree.

Input Specification:

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. While all the keys in a tree are positive integers, we use negative signs to represent red nodes. All the numbers in a line are separated by a space. The sample input cases correspond to the trees shown in Figure 1, 2 and 3.

Output Specification:

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

Sample Input:

```
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
```

Sample Output:

```
Yes
No
No
```

Chapter 2 Algorithm Specification

①数据结构及全局变量定义

```
//定义全局变量
int input[maxn];                        //输入数组存储
int output[maxn] = {0};                 //输出“Yes”“No”存储
//定义数据结构
struct node
{
    int value;                           //节点数值
    struct node *leftchild, *rightchild; //左右子节点
};
typedef struct node *Tree;
```

②建立二叉树函数

```
//最基本建树BuildTree
Tree Build(Tree root, int num)
{
    if (root == NULL)
    {
        root = (Tree)malloc(sizeof(struct node));
        root->value = num;
        root->leftchild = NULL;
        root->rightchild = NULL;
    }
    else if (abs(num) <= abs(root->value))
    {
        root->leftchild = Build(root->leftchild, num);
    }
    else
    {
        root->rightchild = Build(root->rightchild, num);
    }
}
```

利用递归进行最基本的二叉树建立，由于输入红色节点表示为负值，所以判断新节点存放位置需要调用**math**库中的**abs**函数；另外由于题干中已经说明"For each given **binary search tree**, you are supposed to tell if it is a legal red-black tree"，所以可以省去判断输入树是否为二叉搜索树的过程。

③对红黑树性质判断

性质1显然成立；

```
if(input[0]<0) return NO;
```

对性质2，由于输入为前序序列，只需要在输入时判断第一个数的正负即可(入上伪代码)；

对性质3，这里的"叶节点"主要指"NIL节点"，保证没有子节点(实际为两个子节点均为NIL节点)的红色节点的存在性，避免了与性质4的冲突；

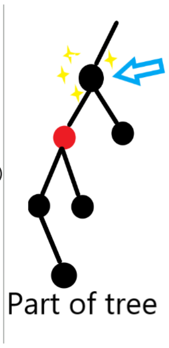
```
//函数判断性质4: If a node is red, then both its children are black.
int Judge_Properties_four(Tree root)
{
    if (root == NULL)
        return 1;
    if (root->value < 0)
    {
        if (root->leftchild != NULL && root->leftchild->value < 0)
            return 0;
        if (root->rightchild != NULL && root->rightchild->value < 0)
            return 0;
    }
    return Judge_Properties_four(root->leftchild) &&
Judge_Properties_four(root->rightchild);
}
```

对性质4，只需要判断每一个红色节点的左右子节点是否为正数即可；

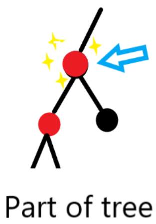
```
//计算对每个结点到叶(子)节点路径黑色节点个数
int Path_Black_Num(Tree root)
{
    if (root == NULL)
        return 0;
    int left_num = Path_Black_Num(root->leftchild);
    int right_num = Path_Black_Num(root->rightchild);
    if (root->value > 0)
        return left_num + 1;
    else
        return left_num;
}
//函数判断性质5: For each node, all simple paths from the node to descendant
leaves contain the same number of black nodes.
int Judge_Properties_five(Tree root)
{
    if (root == NULL)
        return 1;
    int left_path_num = Path_Black_Num(root->leftchild);
    int right_path_num = Path_Black_Num(root->rightchild);
    if (left_path_num != right_path_num)
        return 0;
```

```
    return Judge_Properties_five(root->leftchild) &&
Judge_Properties_five(root->rightchild);
}
```

性质5的判断是本题最大的难点，**Path_Black_Num** 函数利用递归计算某节点到叶(子)节点路径黑色节点个数，value>0(黑色节点)则返回值+1，否则直接返回**left_num**。按照正常逻辑，返回值应该选择**left_num**和**right_num**中的更大者的值，这里返回值选择**left_num**而不是**left_num**和**right_num**中的更大者的值，是因为

① 

这种情况下，在最外层**Judge_Properties_five**函数递归到蓝色箭头指示节点时，会直接返回0(FALSE);

② 

这种特殊情况过不了**Judge_Properties_four**函数，可以直接pass掉;

所以选择**left_num**代替**left_num**和**right_num**中的更大者的值并不会影响对性质5的判断，并且可以节省一些步骤，缩短时间;

Judge_Properties_five函数用来判断对每个结点性质5是否成立，如果不成立直接return 0;

④主函数(输入与输出)

```
//主函数
int main()
{
    int K, n, i, m;
    scanf("%d", &K);
    for (m = 0; m < K; m++)
    {
        scanf("%d", &n);
        memset(input, 0, sizeof(input));
        Tree root = NULL;
        for (i = 0; i < n; i++)
        {
            scanf("%d", &input[i]);
            root = Build(root, input[i]);
        }
        if (input[0] < 0 || !Judge_Properties_four(root) ||
!Judge_Properties_five(root))
            output[m] = 1;
    }
    for (m = 0; m < K; m++)
    {
        if (!output[m])
            printf("Yes\n");
        else
```

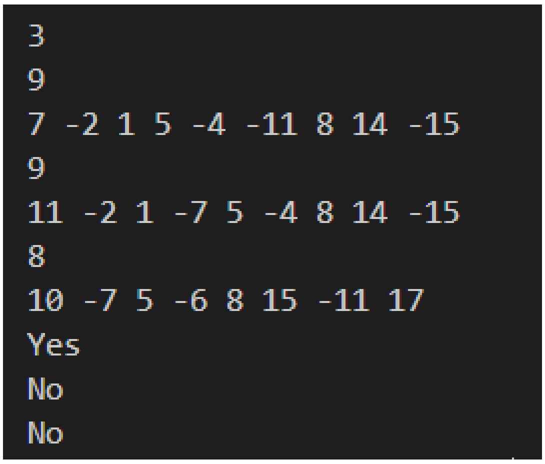
```
            printf("No\n");
    }
    return 0;
}
```

这里使用Input数组存储输入节点数据，每次新的输入前利用memset函数对input数组进行清零，并且将root变量清空，防止污染后续输入，并且可以避免将input声明为二维数组；output数组用来存储每一个树的判断情况，output[i]=0代表第i+1次输入的树是红黑树，反之亦反。

Chapter 3: Testing Results

Test 1 (Sample Input)

Input	Output	Description
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	验证样例



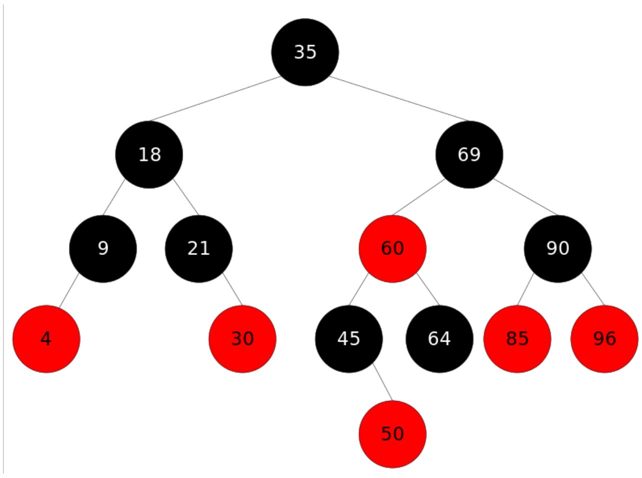
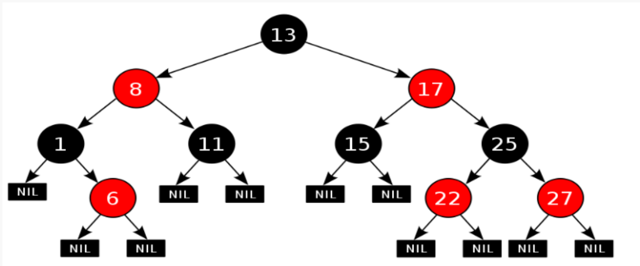
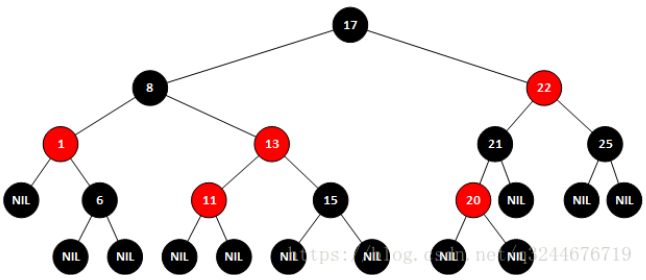
标准样例运行截图

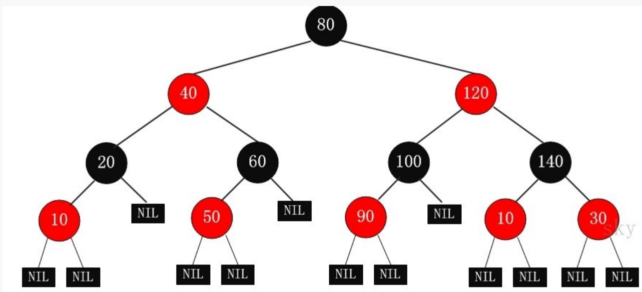
Test 2 Input Test((min_k,min_n),max_k,max_n,(max_k,max_n))

Input	Ideal Output	Description	
1 0	Yes	空树验证	√
1 1 1	Yes	K,N都取最小正值的边界验证	√
30 1 1 ... 1 1	Yes Yes ... Yes Yes	K取最大值的边界验证	√
1 30 35 18 9 -4 2 1 3 5 -15 10 16 21 -30 28 34 69 -60 45 39 50 64 61 67 90 -85 75 86 -96 92 100	No	N取最大值的边界验证	√
30 30 35 18 9 -4 2 1 3 5 -15 10 16 21 -30 28 34 69 -60 45 39 50 64 61 67 90 -85 75 86 -96 92 100 ... 35 18 9 -4 2 1 3 5 -15 10 16 21 -30 28 34 69 -60 45 39 50 64 61 67 90 -85 75 86 -96 92 100	No No ... No	K,N都取最大值的边界验证	√

注：运行截图尺寸过于奇怪，故将Test 2 的运行截图存储于文件夹“Test_2_sample_picture”中。

Test 3 Judge Test(general sample)

<table>
<tr><th>Input</th><th>Ideal Output</th><th>Picture</th><th></th></tr>
<tr><td>1
14
35 18 9 -4 21
-30 69 -60 45
-50 64 90 -85
-96</td><td>Yes</td><td>
</td><td>√</td></tr>
<tr><td>1
10
13 -8 1 -6 11
-17 15 25 -22
-27</td><td>Yes</td><td>
</td><td>√</td></tr>
<tr><td>1
11
17 8 -1 6 -13
-11 15 -22 21
-20 25</td><td>No</td><td>

(Reason : Property 4)</td><td>√</td></tr>
<tr><td>1
11
80 -40 20 -10
60 -50 100
-90 140 -10
-30</td><td>No</td><td>

(Reason : Property 3)</td><td>√</td></tr>
</table>

```
4
14
35 18 9 -4 21 -30 69 -60 45 -50 64 90 -85 -96
10
13 -8 1 -6 11 -17 15 25 -22 -27
11
17 8 -1 6 -13 -11 15 -22 21 -20 25
11
80 -40 20 -10 60 -50 100 -90 140 -10 -30
Yes
Yes
No
No
```

四种平凡样例的运行结果截图

Chapter 4 Analysis and Comments

本次实验中，代码运行时间占用最多的部分就是**Judge_Properties_five**函数的嵌套递归，这种方法的优点在于不需要开内存来记录每个节点到叶子节点的路径上黑色节点个数，缺点在于时间复杂度上升。与之对应的，可以通过另开容器来减少递归嵌套层数，缩短运行时间，缺点在于需要另外声明容器来储存节点深度数组，并且空间复杂度较大。而对于此题而言，重点在于树的多层遍历，那么可以考虑利用数组存储树，优点是除必要的递归(DFS)外对数组操作时间复杂度极低，缺点是数据离散分布对内存浪费大且节点间关联性差。

数组存储(示意)代码如下：

```
int Judge_Properties_five(int x)
{
    if(left[x])
    mleft[x] += Judge_Properties_five(left[x]);
    if(right[x])
    mright[x] += Judge_Properties_five(right[x]);
    if(mleft[x] != mright[x])
    output[x] = 1;//
    if(input[x] < 0)
    return mleft[x];
    else
    return mleft[x] + 1;
}
```

实现方式	时间复杂度	空间复杂度
嵌套递归	大	小
递归+深度数组+记录数组	中	中
递归+树数组	小	大

在本次实验中，题干对节点数目进行了限制(N<=30),这就说明红黑树的深度较小(考虑极端情况，对30个节点的AVL树，其最大深度不超过([log(30)]+1)=5),而对性质5判断函数的递归次数与二叉树的深度呈正相关，所以理论上这三种实现方式运行时间相差不大。

Appendix: Source Code (in C)

```c
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#define maxn 35
//定义全局变量
int input[maxn];
int output[maxn] = {0};
//定义数据结构
struct node
{
    int value;                          //结点数值
    struct node *leftchild, *rightchild; //左右子节点
};
typedef struct node *Tree;
Tree Build(Tree root, int num);       //最基本建树BuildTree
int Judge_Properties_four(Tree root); //函数判断性质4
int Path_Black_Num(Tree root);        //计算对每个结点到叶(子)节点路径黑色节点个数
int Judge_Properties_five(Tree root); //函数判断性质5
int main()
{
    int K, n, i, m;
    scanf("%d", &K);
    for (m = 0; m < K; m++)
    {
        scanf("%d", &n);
        memset(input, 0, sizeof(input)); //input数组清零
        Tree root = NULL;                //root清空
        for (i = 0; i < n; i++)
        {
            scanf("%d", &input[i]);
            root = Build(root, input[i]); //递归建树
        }
        if (input[0] < 0 || !Judge_Properties_four(root) ||
!Judge_Properties_five(root))
            //input[0]<0直接pass
            output[m] = 1;

            //输出“No”的标志
    }
    for (m = 0; m < K; m++)
    {
        if (!output[m])
            printf("Yes\n");
        else
            printf("No\n");
    }
    return 0;
}

Tree Build(Tree root, int num)
{
    if (root == NULL)
    {
        root = (Tree)malloc(sizeof(struct node)); //申请空间
        root->value = num;
        root->leftchild = NULL;
```

```
        root->rightchild = NULL;
    }
    else if (abs(num) <= abs(root->value))        //红节点为负值,利用abs函数还原
    {
        root->leftchild = Build(root->leftchild, num);
    }
    else
    {
        root->rightchild = Build(root->rightchild, num);
    }
}

int Judge_Properties_four(Tree root)
{
    if (root == NULL)
        return 1;
    if (root->value < 0)
    {
        if (root->leftchild != NULL && root->leftchild->value < 0)
            return 0;
        if (root->rightchild != NULL && root->rightchild->value < 0)
            return 0;
    }
    //判断红节点的左右子节点是否为黑节点
    return Judge_Properties_four(root->leftchild) && Judge_Properties_four(root-
>rightchild);
    //递归判断下一层
}

int Path_Black_Num(Tree root)
{
    if (root == NULL)
        return 0;              //到达底层得到值
    int left_num = Path_Black_Num(root->leftchild);
    int right_num = Path_Black_Num(root->rightchild);
    if (root->value > 0)
        return left_num + 1; //经过黑节点+1
    else
        return left_num;     //经过红节点直接进入下一层
}

int Judge_Properties_five(Tree root)
{
    if (root == NULL)
        return 1;
    int left_path_num = Path_Black_Num(root->leftchild);
    int right_path_num = Path_Black_Num(root->rightchild);
    if (left_path_num != right_path_num)
        return 0;

    //节点数不相等则直接返回0值
    return Judge_Properties_five(root->leftchild) && Judge_Properties_five(root-
>rightchild);
    //节点数相等进入下一层
}
```