Hard Performance Measurement(MSS)

2023-10-7

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.

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.

To complete this program, I should build a tree through the preorder traversal and the quality of binary search tree. And to judge the fifth requirement, I record the number of black nodes of each traversal.

Chapter 2: Algorithm Specification

Build tree

Because the project gives the preorder traversal and it's a binary search tree. We can easily know that if we transform all the numbers to positive. Using a for circulation, I sign the first number which is more than the first number. The sign number is the root of right tree, and the second number to last number of the sign is the left number. And through the recursion, we can build the tree. It is worth noting that the end of recursion. I set the start and end of each recursion, and when `start==end`, it means that it has only one node. And when `start>end+1`, it means that I reach the NULL, then `return NULL`.

Another important point is the parameters that need to be passed in the recursion function. I set 4 parameters. The first is the number of tree. The same tree has the same number. The second is the root of each node. The third is the start of the child tree. The fourth is the end of the child tree. For example, the start of recursion is `Recursion(i, keep[i][0], 1, flag[i] - 1)`, and the left tree is `T_->Left = Recursion(i, keep[i][start], start + 1, kk - 1);`, the right tree is `T_->Right = Recursion(i, keep[i][kk], kk + 1, end);`

Identify a node' color

I use `scanf("%d", &keep[i][j]);` to read numbers. And when building tree, I build a node while judging it is positive or negative. When it is positive, I keep it directly and the color is black. When it is negative, I keep its opposite and the color is red. And when judging the start of right tree, I judge them through transforming all the numbers into positive. That can compare their abs easily.

Judge if it is a red-black tree

After building a tree, I judge it using `JudgeTree(T)`.The function returns `sign`. And in the `JudgeTree` function, I change sign when there is something not satisfying the condition. Finally, I print yes or no through sign. Additionally, I preorder the tree again, because I need record the number of black nodes of each path.

Every node is either red or black.

This can be ignored.

The root is black

This is easy. Before the recursion, judging the root node is black. That's all.

Every leaf (NULL) is black.

This can be ignored too.

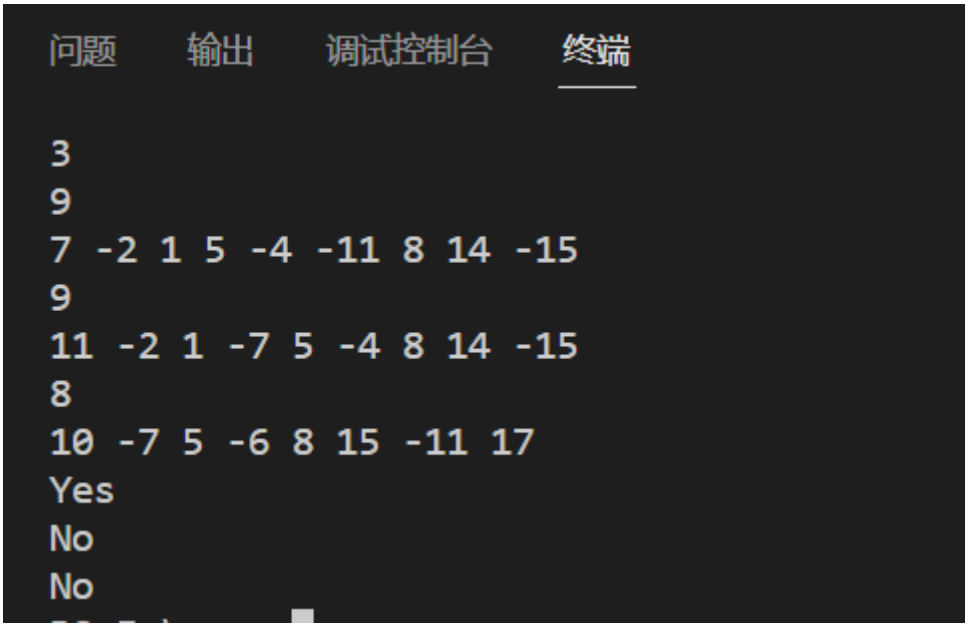
If a node is red, then both its children are black.

In the recursion, if a node is red, I judge the left tree and the right tree. If it is not NULL, then the color of node must be black, or the sign is 0.

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

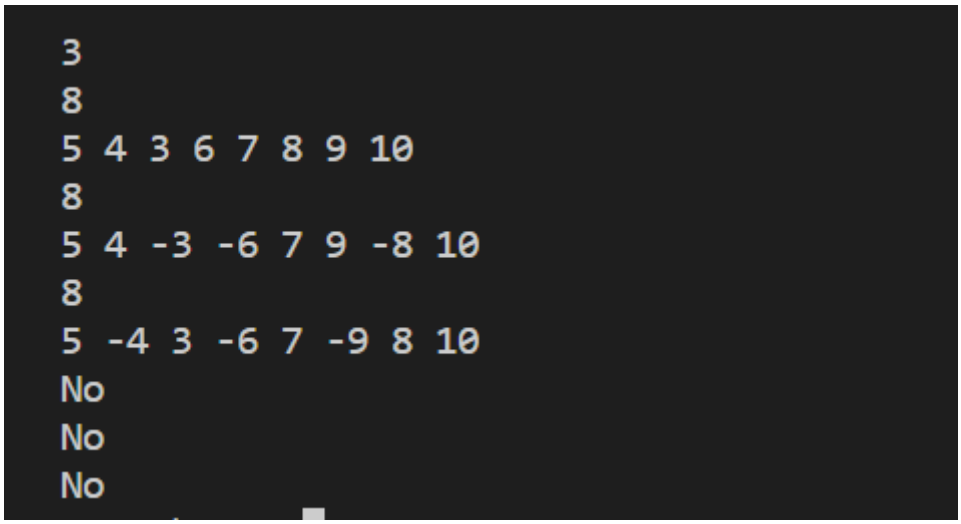
This is the most complex requirement. When each recursion, I record the number of black node. And when it comes to the end node, if it is the first path, I record the number of black nodes in an array. Otherwise, I compare the number with which kept in array. And if the result of comparing is false, I change the sign. Additionally, I add a parameter in recursion function to record the recursion level of black numbers.

Chapter 3: Testing Results



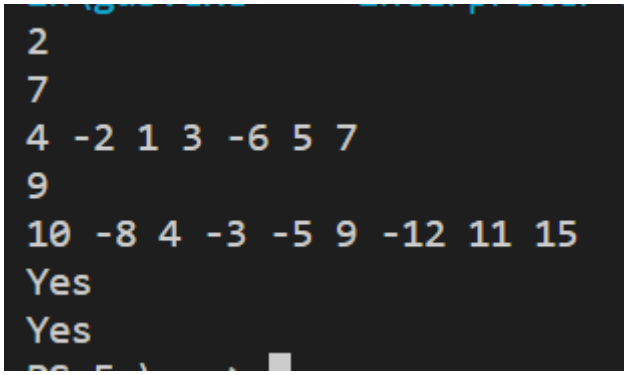
First, I test the samples on PTA. We can notice that the output answer is correct.

Then, I design 3 trees, which are all not Red-Black Tree. The inputs and outputs are as follows.



The answer is right.

More than, I design 2 trees, which are all Red-Black Tree. The inputs and outputs are as follows.



The answer is right.

Finally, I design a special tree, which is not Red-Black Tree. The inputs and output are as follows.

Chapter 4: Analysis and Comments

Through the program, we can get whether the tree is a Red-Black tree. The time complexity seems to be O(N) and the space complexity is O(N).

All in all, I think my advantage is the algorithm is easily to understand and the solving way is easy and clear. And the functions are divided so they seem clear too.

The disadvantage is that I use too many global variables. If a variable is frequently used in a program, such as a loop variable, the system must access the unit in memory several times, affecting the efficiency of program execution.

Appendix: Source Code (in C)

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

#define RED 0   // define RED is 0
#define BLACK 1 // define BLACK is 1

int keep[100][100]; // keep all the numbers of all the treeds
int flag[100];      // keep how many nodes each tree has
int sign;           // prompt viable to sign YES or NO
int lev[100] = {};  // keep each way's black node's number
int count;          // count the tree

typedef struct TreeNode *Tree;
struct TreeNode
{
    int Element; // the number of each node
    Tree Left;
    Tree Right;
    int color; // the color of each node
};

void Read(int total) // read the input
{
    int i, j;
    for (i = 0; i < total; i++)
    {
        scanf("%d", &flag[i]);
        for (j = 0; j < flag[i]; j++) // there are flag[i] numbers in the tree
        {
            scanf("%d", &keep[i][j]); // the numbers are kept in array keep
        }
    }
}
```

```
Tree Recursion(int i, int root, int start, int end) // Recursion to build tree
{
    if (start > end + 1) // it means that the child treenode is NULL
        return NULL;
    Tree T_;
    T_ = (Tree)malloc(sizeof(struct TreeNode)); // if not NULL, malloc space for
the new node
    if (root > 0)                               // root>0, the color is BLACK,
element = root
    {
        T_->Element = root;
        T_->color = BLACK;
    }
    else // root<0, the color is RED, element = -root
    {
        T_->Element = -root;
        T_->color = RED;
    }
    // printf("%d ", T_->Element);
    T_->Left = NULL;  // init the left node
    T_->Right = NULL; // init the right node
    int kk, temp;
    for (kk = start; kk <= end; kk++) // find the first number more than root
    {
        if (keep[i][kk] < 0)
        {
            temp = -keep[i][kk]; // temp should be positive
        }
        else
            temp = keep[i][kk];
        if (temp > abs(root))
            break; // use kk to record
    }
    T_->Left = Recursion(i, keep[i][start], start + 1, kk - 1); // recursion the
left tree
    T_->Right = Recursion(i, keep[i][kk], kk + 1, end);         // recursion the
right tree
    return T_;
}

Tree BuildTree(int i)
{
    Tree T0;
    T0 = Recursion(i, keep[i][0], 1, flag[i] - 1); // start the recursion
    return T0;
}

int PreTree(Tree T, int level)
{
    // printf("%d ", T->Element);
    if (T->Left == NULL && T->Right == NULL) // end the preorder
    {
        if (T->color == BLACK) // notice the last element
            level += 1;
        if (lev[count] == 0)
```

```
        {
            lev[count] = level; // record the level
        }
        if (lev[count] != 0)
        {
            if (lev[count] != level) // compare the level with first level to
judge whether they're equal
                sign = 0;            // change the sign if not equal
        }
    }
    if (T->color == RED) // judge the two children's color
    {
        if (T->Left != NULL)
        {
            if (T->Left->color == RED)
                sign = 0; // if RED, sign = 0
        }
        if (T->Right != NULL)
        {
            if (T->Right->color == RED)
                sign = 0; // if RED, sign = 0
        }
    }
    if (T->color == BLACK) // recursion child node: black: level += 1
    {
        if (T->Left != NULL && T->Right != NULL) // deal with special situation
        {
            PreTree(T->Left, level + 1);
            PreTree(T->Right, level + 1);
        }
        if (T->Left != NULL && T->Right == NULL) // deal with special situation
        {
            if (lev[count] == 0)
            {
                lev[count] = level; // record the level
            }
            if (lev[count] != 0)
            {
                if (lev[count] != level + 1) // compare the level with first
level to judge whether they're equal
                    sign = 0;                // change the sign if not equal
            }
            PreTree(T->Left, level + 1);
        }
        if (T->Left == NULL && T->Right != NULL) // deal with special situation
        {
            if (lev[count] == 0)
            {
                lev[count] = level; // record the level
            }
            if (lev[count] != 0)
            {
                if (lev[count] != level + 1) // compare the level with first
level to judge whether they're equal
                    sign = 0;                // change the sign if not equal
```

```
            }
            PreTree(T->Right, level + 1);
        }
    }
    else // recursion child node: red: level doesn't change
    {
        if (T->Left != NULL && T->Right != NULL) // deal with special situation
        {
            PreTree(T->Left, level);
            PreTree(T->Right, level);
        }
        if (T->Left != NULL && T->Right == NULL) // deal with special situation
        {
            if (lev[count] == 0)
            {
                lev[count] = level; // record the level
            }
            if (lev[count] != 0)
            {
                if (lev[count] != level) // compare the level with first level
to judge whether they're equal
                    sign = 0;            // change the sign if not equal
            }
            PreTree(T->Left, level);
        }
        if (T->Left == NULL && T->Right != NULL) // deal with special situation
        {
            if (lev[count] == 0)
            {
                lev[count] = level; // record the level
            }
            if (lev[count] != 0)
            {
                if (lev[count] != level) // compare the level with first level
to judge whether they're equal
                    sign = 0;            // change the sign if not equal
            }
            PreTree(T->Right, level);
        }
    }
    return sign; // return the number of sign
}

int JudgeTree(Tree T) // judge if the tree is a red black tree
{
    sign = 1;
    if (T->color == RED) // the first node must be BLACK
    {
        sign = 0;
    }
    PreTree(T, 0); // PreTree function also changes sign
    return sign;   // return sign
}

int main()
```

```
{
    int total;
    scanf("%d", &total); // read the number of trees
    Read(total);
    for (count = 0; count < total; count++)
    {
        Tree T;
        T = BuildTree(count); // deal with each tree
        if (JudgeTree(T))
        {
            printf("Yes"); // print YES
        }
        else
        {
            printf("No"); // print NO
        }
        if (count != total - 1) // if not the end string, print "\n"
        {
            printf("\n");
        }
    }
    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.