浙江大學

《数据结构基础》

实验报告

周　　次	:	
上课时间	:	
授课教师	:	杨子祺
日　　期	:	2023/9/30

Chapter 1: Introduction

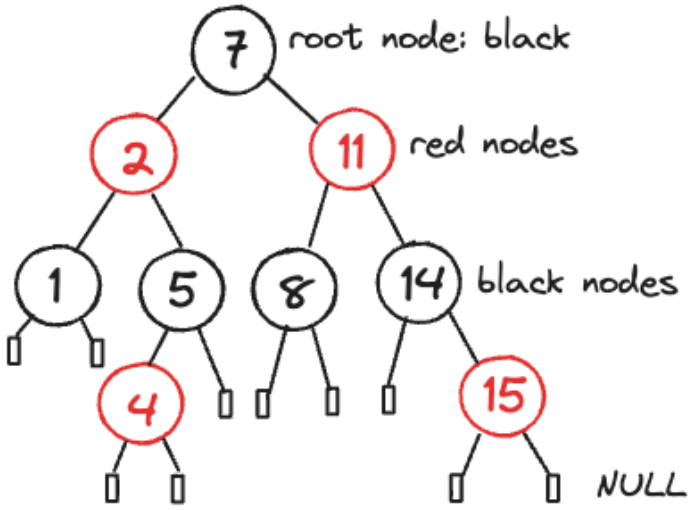
There is a kind of balanced binary search tree named red-black tree in the data structure. It has the following 5 properties:

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.

the property 4 and 5 need to be judged seriously.

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

> **Example**
>
> little squares present NULL leaf nodes)



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.

Chapter 2: Algorithm Specification

2.1 Build Tree according to input

> **Note**
>
> I used 2 algorithms to build tree, the first one is to bubble sort PreOrder traversal and get the InOrder traversal. This is a little clumsy so I use Insert function to build tree in the final version. Also I used PreOrder function to output the built-tree to check if it's correct.

2.1.1 the clumsy version

Because it is a binary search tree, the output obtained by an inorder traversal is monotonically increasing. Therefore, based on the preorder traversal, we can deduce the inorder traversal and construct the tree, similar to the method used in the previous homework to construct a binary tree using postorder and inorder. 因为是二叉搜索树，按照中序遍历得到的输出是单调递增的，所以根据先序遍历得出中序遍历，然后根据这两种遍历构建树，和上一次 homework 中利用 PostOrder 和 InOrder 构建二叉树类似。

- For inorder traversal, the left subtree of a root node is located to the left of the root, and the right subtree is located to the right of the root. 对于中序遍历，根节点的左子树位于根节点的左侧，右子树位于根节点的右侧。
- For preorder traversal, the root node is the first element in the sequence, and the left subtree is located to the left of the root, while the right subtree is located to the right of the root. 对于先序遍历，根节点位于序列的第一个元素，左子树位于根节点的左侧，右子树位于根节点的右侧。

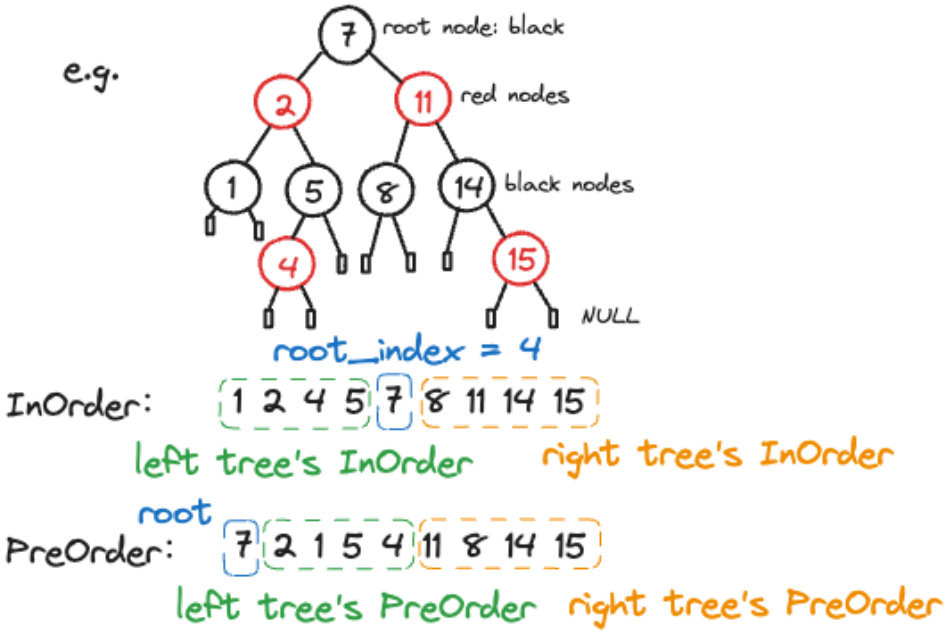
1. Get the value of the root node from the first element of the preorder traversal.从先序遍历的第一个元素中获取根节点的值。

2. Find the position of the root node in the inorder traversal and divide it into the inorder traversal sequences of the left subtree and the right subtree. 在中序遍历中找到根节点的位置，将其分为左子树的中序遍历序列和右子树的中序遍历序列。

3. Divide the preorder traversal sequence into the preorder traversal sequences of the left subtree and the right subtree based on the length of the inorder traversal sequence of the left subtree. 根据左子树中序遍历序列的长度，将先序遍历序列分为左子树的先序遍历序列和右子树的先序遍历序列。

4. Recursively perform the same operation on the left subtree and the right subtree to build them. 递归地对左子树和右子树进行相同的操作，构建左子树和右子树。

5. Connect the root node, the left subtree, and the right subtree to construct the complete binary tree. 将根节点、左子树和右子树连接起来，构建完整的二叉树。




```
/* in main: Bubble sort PreOrder to get InOrder */
        int *pre = (int *)malloc(n * sizeof(int));
        int *in  = (int *)malloc(n * sizeof(int));

    for (int j = 0; j < n; j++) {
        scanf("%d", &pre[j]); // get input PreOrder
        in[j] = pre[j];
    }

    for (int a = 0; a < n; a++) { // bubble sort get InOrder
        for (int j = 0; j < n - i - 1; j++) {
            if (in[j] > in[j + 1]) {
                    int temp = in[j];
                    in[j] = in[j + 1];
                    in[j + 1] = temp;
            }
```

```
        }
    }
        Tree t = (Tree)malloc(sizeof(struct Node));
        t = BuildTree(n, in, pre); // BuildTree function call
```

```
Tree BuildTree(int n, int* in, int* pre) {
    if (n <= 0) {
        return NULL;
    }

    Tree t = (Tree)malloc(sizeof(struct Node));
    t->left  = NULL; // initialize left child
    t->right = NULL; // initialize right child
    t->value = (pre[0] > 0 ? pre[0] : -pre[0]); // get abs(value)
    t->color = (pre[0] < 0 ? RED : BLACK); // get color

    int root_index = 0;
    for (int i = 0; i < n; i++) {
        if (in[i] == pre[0]) {
            root_index = i;
            break;
        }
    }

    t->left = BuildTree(root_index, in, pre + 1);
    t->right = BuildTree(n - root_index - 1, in + root_index + 1, pre + root_index + 1);

    return t;
}
```

2.1.2 the Insert version

1. Initialize an empty tree `t`. 初始化一个空树 `t`
2. Read the values of the nodes from input 从输入中读取节点的值
3. For each value, call the `Insert` function with the current tree `t` and the value as parameters. 对于每个值，调用 `Insert` 函数，将当前树 `t` 和该值作为参数传入
4. The `Insert` function will create a new node with the given value, determine its color (red or black) based on the value, and insert it into the tree `t` at the appropriate position. `Insert` 函数将创建一个具有给定值的新节点，根据符号正负确定其颜色（红色或黑色），并将其插入到树 `t` 的适当位置
5. The `Insert` function returns the updated tree `t`. `Insert` 函数返回更新后的树 `t`
6. Update the tree `t` with the returned value from `Insert`. 使用 `Insert` 返回的值更新树 `t`

```
Tree Insert(Tree t, int value) { // build tree using Insert
    if (t == NULL) {
        t = (Tree)malloc(sizeof(struct Node));
        t->value = abs(value);
        t->color = value < 0 ? RED : BLACK;
        t->left = t->right = NULL;
    } else if (abs(value) < t->value) {
        t->left = Insert(t->left, value);
    } else if (abs(value) > t->value) {
        t->right = Insert(t->right, value);
    }
    return t;
}
```

```
// in function main call:
        for (int j = 0; j < n; j++) { // build Tree
            scanf("%d", &pre[j]);
            t = Insert(t, pre[j]);
        }
```

2.2 Judge wether the tree is Red Black Tree

2.2.1 Judge whether there are adjacent red nodes

1. If the current node `t` is NULL, return true, indicating that the property is satisfied. 如果当前节点 `t` 为空，则返回真
2. If the color of the current node `t` is red, check its left and right child nodes. 如果当前节点 `t` 的颜色为红色，那么检查它的左子节点和右子节点
    - If the left child node is not NULL and its color is red, return false, indicating that the property is not satisfied.
    - If the right child node is not NULL and its color is red, return false, indicating that the property is not satisfied.
3. Recursively call the `Property4` function on the left subtree and right subtree to check if they satisfy the property. 递归地对左子树和右子树调用 `Property4` 函数，检查它们是否满足性质
4. Return the logical AND result of whether the left subtree and right subtree both satisfy the property. 返回左子树和右子树是否都满足性质

```
bool Property4(Tree t) { // there is no red node with red children
    if (t == NULL) {
        return true;
    }

    if (t->color == RED) {
        if (t->left != NULL && t->left->color == RED) {
            return false;
        }
        if (t->right != NULL && t->right->color == RED) {
            return false;
        }
    }

    return Property4(t->left) && Property4(t->right); // check recursively
}
```

2.2.2 Judge wether all paths from root to leaf have the same number of black nodes

1. If the current node `t` is NULL, indicating a leaf node, return 1 to represent one black node on the path. 如果当前节点 `t` 为空，表示到达叶子节点，返回 1，表示路径上有一个黑色节点
2. Recursively calculate the black height (the number of black nodes on the path) of the left subtree and assign it to `leftHeight`. 递归地计算左子树的黑高度（路径上的黑色节点数量）并赋值给 `leftHeight`。
3. Recursively calculate the black height of the right subtree and assign it to `rightHeight`. 递归地计算右子树的黑高度并赋值给 `rightHeight`。
4. If the black height of the left subtree is not equal to the black height of the right subtree or if either the left or right subtree has a black height of -1 (indicating inequality), return -1 to indicate that the number of black nodes on the path is not equal. 如果左子树的黑高度与右子树的黑高度不相等，或者左子树或右子树的黑高度为 -1（表示不相等），则返回 -1，表示路径上的黑色节点数量不相等。
5. If the color of the current node `t` is black, return the black height of the left subtree plus 1 (only need to return one side since they are equal). 如果当前节点 `t` 的颜色为黑色，返回左子树的黑高度加 1
6. If the color of the current node `t` is red, return the black height of the left subtree. 如果当前节点 `t` 的颜色为红色，返回左子树的黑高度。
7. During the recursive backtracking process, the calculated black height is continuously returned to the upper levels. Finally, the function determines whether the entire tree satisfies the property of having an equal number of black nodes on all paths. 递归回溯过程中，将计算得到的黑高度一直返回上层，最终判断整个树是否满足路径上的黑色节点数量相等的性质。

```
int blackHeight(Tree t) { // check if all paths from root to leaf have the same number of black nodes
    if (t == NULL) { // leaf
        return 1;
    }
    int leftHeight  = blackHeight(t->left);
    int rightHeight = blackHeight(t->right);

    if (leftHeight != rightHeight || leftHeight == -1 || rightHeight == -1) {
        return -1; // not equal
    }

    if (t->color == BLACK) {  // count black nodes
        return leftHeight + 1; // just need to return one side because they are equal
    } else {
        return leftHeight;
    }
}
```

Chapter 3: Testing Results

3.1 Sample Input and Output

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
```

My shell's Output:

```
└─ make redblack
mkdir -p build
gcc -Wall redblack.c -o build/redblack
build/redblack
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
```

For the sample input, my out put is `YES NO NO`, same as the sample output.

3.2 My own testcases:

3.2.1 Max case

```
30
30
10 -15 20 -25 30 -35 40 -45 50 -55 60 -65 70 -75 80 -85 90 -95 100 -105 110 -115 120 -125 130
-135 140 145 150 -155
...（重复 30 次含有 30 个元素的树的输入）
```

3.2.2 Min case

```
1
1
10
```

3.2.3 Root is Black

```
1
5
```

```
10 -5 15 -12 18
```

3.2.4 Adjecent Red Nodes

```
1
7
10 -5 -3 20 25 -22 -24
```

3.2.5 Different Height

```
1
6
10 5 -15 -12 25 30
```

Chapter 4: Analysis and Comments

4.1 Time Complexity of Build Tree using Insert function

The code you provided is for inserting a node into a binary tree. The `Insert` function is recursive and the depth of the recursion would be determined by the height of the tree.

Assuming that the tree is balanced, the height of the tree would be $(O(\log N))$ where N is the number of nodes in the tree. Each insertion would take $(O(\log N))$ time, therefore K insertions would take $(K \times O(\log N))$ time.

The height (h) of the tree varies from (1) to (log N) as nodes are inserted. Summing (log k) for k from 1 to N gives us:

$$\log 1 + \log 2 + \log 3 + \ldots + \log N = \log(1 \times 2 \times 3 \times \ldots \times N) = \log(N!)$$

According to Stirling's approximation, $(log(N!))$ is $(O(N \log N))$.

Therefore, the time complexity of building the tree using (K) inserts in a **balanced binary tree** would be (O(N \log N)) if (N) is the number of nodes I wish to insert. If the tree is not balanced, the **worst-case** time complexity could be as bad as $(O(N^2))$.

4.2 Time Complexity of Judge Functions

Property4 and blackHeighgt functions' time complexity are both $O(N)$ because they both need to traverse the whole tree.

Chapter 5: Appendix: Source Code in C

Run:

In command line input `make redblack` or `gcc redblack.c -o redblack; ./build/redblack`

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

#define RED 0
#define BLACK 1
#define maxnum 30

/*
(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.
*/

struct Node {
    int value;
    int color;
    struct Node *left, *right;
};
typedef struct Node *Tree;

Tree Insert(Tree t, int value);
void PreOrder(Tree T); // test function
int blackHeight(Tree t); // count if blacknodes are equal in every path
bool Property4(Tree t); // judge if there is a red node with red children


int main() {
    int k = 0, n = 0;
    bool *result = (bool *)malloc(k * sizeof(int));

    scanf("%d", &k); // number of cases

    for (int i = 0; i < k; i++) {

        scanf("%d", &n); // number of nodes
        int *pre = (int *)malloc(n * sizeof(int));
        Tree t = NULL; // initialize Tree t
```

```
        for (int j = 0; j < n; j++) { // build Tree
            scanf("%d", &pre[j]);
            t = Insert(t, pre[j]);
        }

        // PreOrder(t);
        result[i] = (t->color==BLACK) && Property4(t) && (blackHeight(t) != -1); // judge if it
is a red-black tree
        free(pre); // free memory
    }

    for (int i = 0; i < k; i++) { // print result
        if (result[i] == true) {
                printf("Yes\n");
            } else {
                printf("No\n");
            }
    }

    return 0;
}

Tree Insert(Tree t, int value) { // build tree using Insert
    if (t == NULL) {
        t = (Tree)malloc(sizeof(struct Node));
        t->value = abs(value);
        t->color = value < 0 ? RED : BLACK;
        t->left = t->right = NULL;
    } else if (abs(value) < t->value) {
        t->left = Insert(t->left, value);
    } else if (abs(value) > t->value) {
        t->right = Insert(t->right, value);
    }

    return t;
}

void PreOrder(Tree T) { // test if my BuildTree is correct (don't use this function in the
final version)
        if (T != NULL) {
        printf("%d ", T->value);
        PreOrder(T->left);
        PreOrder(T->right);
        }
}

bool Property4(Tree t) { // there is no red node with red children
    if (t == NULL) {
        return true;
    }

    if (t->color == RED) {
```

```
        if (t->left != NULL && t->left->color == RED) {
            return false;
        }
        if (t->right != NULL && t->right->color == RED) {
            return false;
        }
    }

    return Property4(t->left) && Property4(t->right); // check recursively
}


int blackHeight(Tree t) { // check if all paths from root to leaf have the same number of black
nodes
    if (t == NULL) { // leaf
        return 1;
    }
    int leftHeight  = blackHeight(t->left);
    int rightHeight = blackHeight(t->right);

    if (leftHeight != rightHeight || leftHeight == -1 || rightHeight == -1) {
        return -1; // not equal
    }

    if (t->color == BLACK) {  // count black nodes
        return leftHeight + 1; // just need to return one side because they are equal
    } else {
        return leftHeight;
    }
}
```

```
# 定义编译器和编译选项
CC = gcc
CFLAGS = -Wall

# 默认目标
all:
        @echo "Usage: make <source-file.c> to compile and run."

# 自定义规则：通过文件名编译和运行
%: %.c
        mkdir -p build
        $(CC) $(CFLAGS) $< -o build/$*
        build/$*

clean:
        rm -rf build/
```

- I hereby declare that all the work done in this project titled RedBlackTree is of my independent effort.