**Student Name： 代翔 StudentID:2023337621159**

**Experiment 2:** Tree

**Purpose:**

1. Master the concepts and properties of different Trees;

2. Master the representation of different Trees;

3. Understand the way of using Tree structures to solve problems.

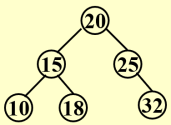
**Problems and requirements:**

【Problem 1】: Collecting statistics of a binary search tree (BST).

**Requirements:**

**Input**: a sequence of integers inserted one by one for constructing a BST T;

**Output**: depth(T), number of leaves; number of nodes having one child; number of full nodes; parent-children relationships

Sample input: 20 15 18 10 25 32 //the BST tree will be 

Sample output:

Depth of the tree: 2

Number of leaves: 3

Number of full nodes: 2

Number of nodes having one child: 1

Node 20: 15, 25

Node 15: 10, 18

Node 25: ,32

【Problem 2】: Red-Black Tree.

**Requirements:** A red-black tree is a special type of balanced binary search tree. It has the following 5 properties:

(1) Every node is either red or black.

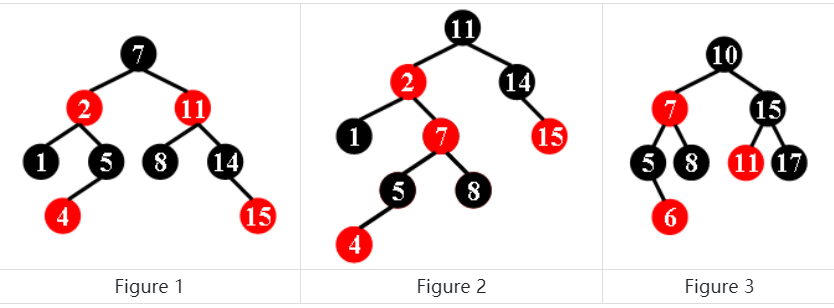
(2) The root is black.

(3) Every NULL node 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.



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

**Input**: the preorder traversal sequence of the tree.

**Output**: yes // indicating that the tree is a red-black tree

No //indicating that the tree is not a red-black 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.

Sample Input:

7 -2 1 5 -4 -11 8 14 -15

Sample Output: Yes

Sample Input: 11 -2 1 -7 5 -4 8 14 -15

Sample Output: No

Sample Input: 10 -7 5 -6 8 15 -11 17

Sample Output: No

**[Task 1]**

1. **Task description:** *//clarify your understanding of the tasks to solve problem1*

*The solution to problem 1 is to achieve a deep view of a BST, a leaf node, a full node, and a node with only one child, which has a parent-child relationship*

1. **Method:** *//describe your ideas to solve the problem, you can describe your idea by text, or depict your algorithm.*

*First, you need to create a data structure of a leaf node, and then you need to write an insertion operation to ensure that the left subnode is always smaller than the right subnode. This is followed by calculating the depth, using recursion, adding one at a time. Count the number of leaf nodes, and add one as long as the two child nodes below him are empty. To calculate the full sub-nodes, you only need to add one if the following two sub-nodes are not empty, and recursively the following two sub-nodes. A child only needs to add a child node and recursion that child node. The parent-child relationship can be directly traversed and printed*

1. **Implementation:** *//present and explain your key code here*

#include<stdio.h>

#include<stdlib.h>

typedef struct TreeNode

{

int val;

struct TreeNode\* left;

struct TreeNode\* right;

}TreeNode;

TreeNode\* MakeNode(int val)

{

TreeNode\* Node = (TreeNode\*)malloc(sizeof(TreeNode));

Node->val = val;

Node->left = NULL;

Node->right = NULL;

return Node;

}

TreeNode\* Insert(int val, TreeNode\* T)

{

TreeNode\* Node = MakeNode(val);

if (T == NULL)

{

T = Node;

}

else

{

if ((Node->val) < (T->val))

{

T->left = Insert(val, T->left);

}

else

{

T->right = Insert(val, T->right);

}

}

return T;

}

int CountDepth(TreeNode\* T)

{

if (T == NULL)

return 0;

int Ldepth = CountDepth(T->left);

int Rdepth = CountDepth(T->right);

return (Ldepth > Rdepth ? Ldepth : Rdepth) + 1;

}

int CountLeaves(TreeNode\* T)

{

if (T == NULL)

return 0;

if (T->left == NULL && T->right == NULL)

return 1;

return CountLeaves(T->left) + CountLeaves(T->right);

}

int CountFull(TreeNode\* T)

{

if (T == NULL)

return 0;

if (T->left != NULL && T->right != NULL)

return 1 + CountFull(T->left) + CountFull(T->right);

return CountFull(T->left) + CountFull(T->right);

}

int Count\_oneChild(TreeNode\* T)

{

if (T == NULL)

return 0;

if (T->left != NULL && T->right == NULL)

return 1 + Count\_oneChild(T->left);

if (T->right != NULL && T->left == NULL)

return 1 + Count\_oneChild(T->right);

return Count\_oneChild(T->left) + Count\_oneChild(T->right);

}

void printParentChild(TreeNode\* T)

{

if (T == NULL) return;

else

{

printf("Node %d: ", T->val);

if (T->left) printf("%d, ", T->left->val);

else printf(", ");

if (T->right) printf("%d", T->right->val);

printf("\n");

}

printParentChild(T->left);

printParentChild(T->right);

}

int main()

{

int a[20];

int len = 0;

scanf\_s("%d", &a[len]);

while (a[len] != -1)

{

len++;

scanf\_s("%d", &a[len]);

}

TreeNode\* head = MakeNode(a[0]);

for (int i = 1; i < len ; i++)

{

Insert(a[i], head);

}

printf\_s("Depth of the tree:%d\n", CountDepth(head)-1);

printf\_s("Number of leaves:%d\n", CountLeaves(head));

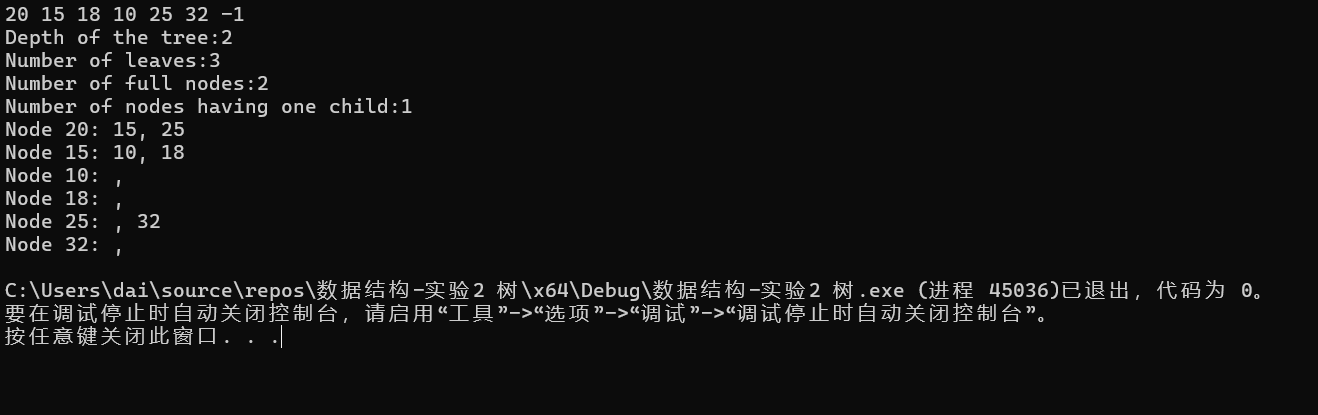
printf\_s("Number of full nodes:%d\n", CountFull(head));

printf\_s("Number of nodes having one child:%d\n", Count\_oneChild(head));

printParentChild(head);

}

1. **Results:** *//run your implementation with some input, then show and explain your outputs.*

****

**[Task 2]**

1. **Task description:** *//clarify your understanding of the tasks to solve problem2*

*Problem 2 is mainly related to the calculation of the number of black nodes on the path.*

1. **Method:** *//describe your ideas to solve the problem, you can describe your idea by text, or depict your algorithm.*

*The first is to interpret whether the root node is black or not, which is judged by returning 0,1. The second is to determine whether it is a balance tree, and one more function is needed to calculate the height of each node, and the height difference is within two is the balance tree, and then recursively. To determine whether the red node child is black, you can judge it by using positive and negative. To determine whether the black nodes in each path are the same, the difference will not be greater than two after the balance tree judgment, so we only need to judge that the black nodes under each node are equal.It should be noted that because the red nodes entered are negative, but the sorting is still positive, we need to insert them with absolute values first, store them in the array, and then change these nodes to negative numbers*

1. **Implementation:** *//present and explain your key code here*

typedef struct TreeNode

{

int val;

struct TreeNode\* left;

struct TreeNode\* right;

}TreeNode;

TreeNode\* MakeNode(int val)

{

TreeNode\* Node = (TreeNode\*)malloc(sizeof(TreeNode));

Node->val = val;

Node->left = NULL;

Node->right = NULL;

return Node;

}

TreeNode\* Insert(int val, TreeNode\* T)

{

TreeNode\* Node = MakeNode(val);

if (T == NULL)

{

T = Node;

}

else

{

if ((Node->val) < (T->val))

{

T->left = Insert(val, T->left);

}

else

{

T->right = Insert(val, T->right);

}

}

return T;

}

int IsRootBlack(TreeNode\* T)

{

return T->val > 0 ? 1 : 0;

}

int GetHeight(TreeNode\* T) {

if (T == NULL)

return 0;

int Lheight = GetHeight(T->left);

int Rheight = GetHeight(T->right);

int Height = (Lheight > Rheight ? Lheight : Rheight) + 1;

return Height;

}

int IsBalance(TreeNode\* T) {

if (T == NULL)

return 1;

int Lheight = GetHeight(T->left);

int Rheight = GetHeight(T->right);

int Lflag = IsBalance(T->left);

int Rflag = IsBalance(T->right);

if (abs(Lheight - Rheight) > 1) {

return 0;

}

return (Lflag & Rflag);

}

int IsChildBlack(TreeNode\* T)

{

if (T == NULL) {

return 1;

}

if (T->val < 0)

{

if ((T->left && T->left->val < 0) || (T->right && T->right->val < 0))

{

return 0;

}

}

int Lflag = IsChildBlack(T->left);

int Rflag = IsChildBlack(T->right);

return (Lflag & Rflag);

}

int CountBlacknode(TreeNode\* T)

{

if (T == NULL)

return 0;

int count = T->val > 0 ? 1 : 0;

count += CountBlacknode(T->left);

count += CountBlacknode(T->right);

return count;

}

int IsBlackEq(TreeNode\* T)

{

if (T == NULL)

return 1;

int Lblack = CountBlacknode(T->left);

int Rblack = CountBlacknode(T->right);

return Lblack == Rblack & IsBlackEq(T->left) & IsBlackEq(T->right);

}

TreeNode\* searchNode(TreeNode\* T, int val)

{

if (T == NULL || T->val == val)

{

return T;

}

if (val < T->val)

{

return searchNode(T->left, val);

}

else

{

return searchNode(T->right, val);

}

}

void IsRDT(TreeNode\* T)

{

if (IsRootBlack(T) & IsChildBlack(T) & IsBalance(T) & IsBlackEq(T))

printf\_s("Yes\n");

else

printf\_s("No\n");

}

int main()

{

int a[20];

int len = 0;

scanf\_s("%d", &a[len]);

while (a[len] != -1)

{

len++;

scanf\_s("%d", &a[len]);

}

TreeNode\* head = MakeNode(a[0]);

for (int i = 1; i < len ; i++)

{

Insert(a[i], head);

}

printf\_s("Depth of the tree:%d\n", CountDepth(head)-1);

printf\_s("Number of leaves:%d\n", CountLeaves(head));

printf\_s("Number of full nodes:%d\n", CountFull(head));

printf\_s("Number of nodes having one child:%d\n", Count\_oneChild(head));

printParentChild(head);

}

int main()

{

int a[20];

int len = 0;

scanf\_s("%d", &a[len]);

while (a[len] != -1)

{

len++;

scanf\_s("%d", &a[len]);

}

TreeNode\* head = MakeNode(a[0]);

for (int i = 1; i < len ; i++)

{

int flag = 1;

if (a[i] < 0)

{

flag = 0;

}

Insert(abs(a[i]), head);

if (flag == 0)

{

TreeNode\* temp = searchNode(head, abs(a[i]));

temp->val = a[i];

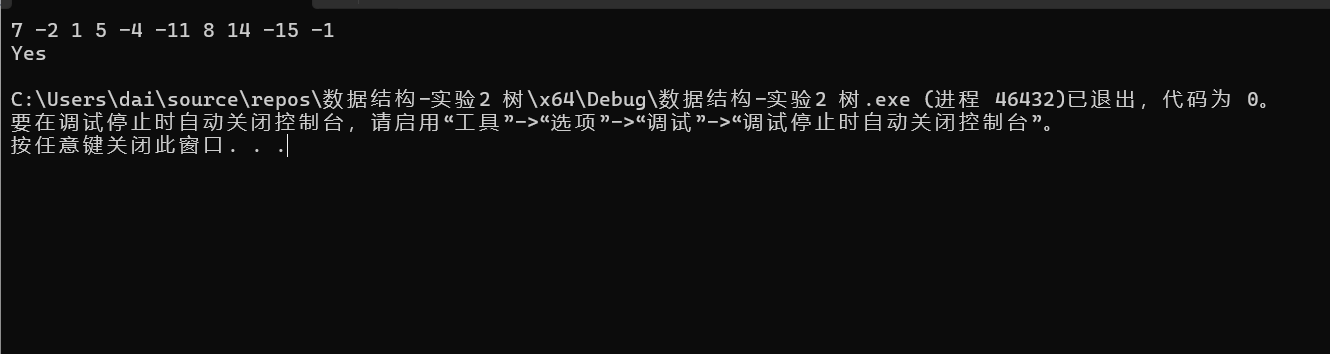
}

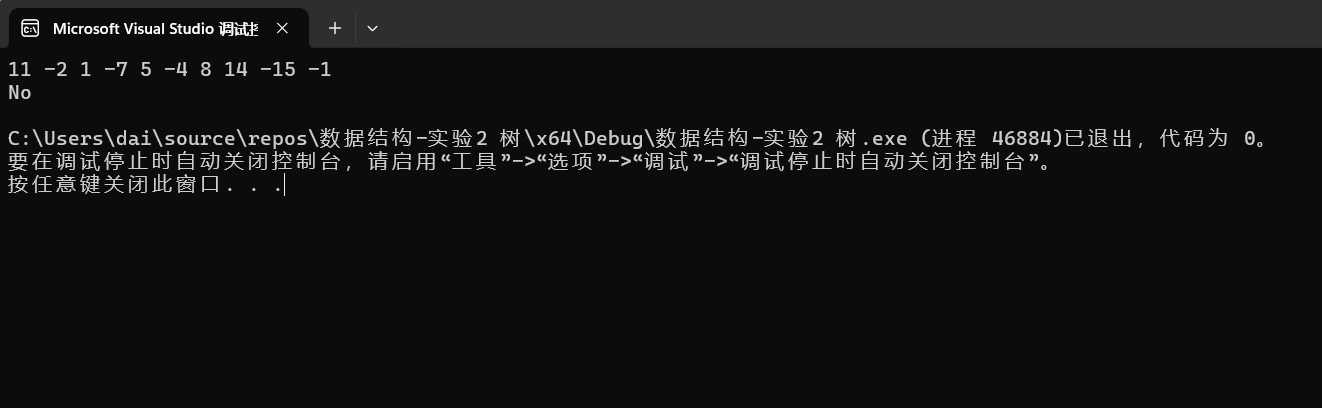
}

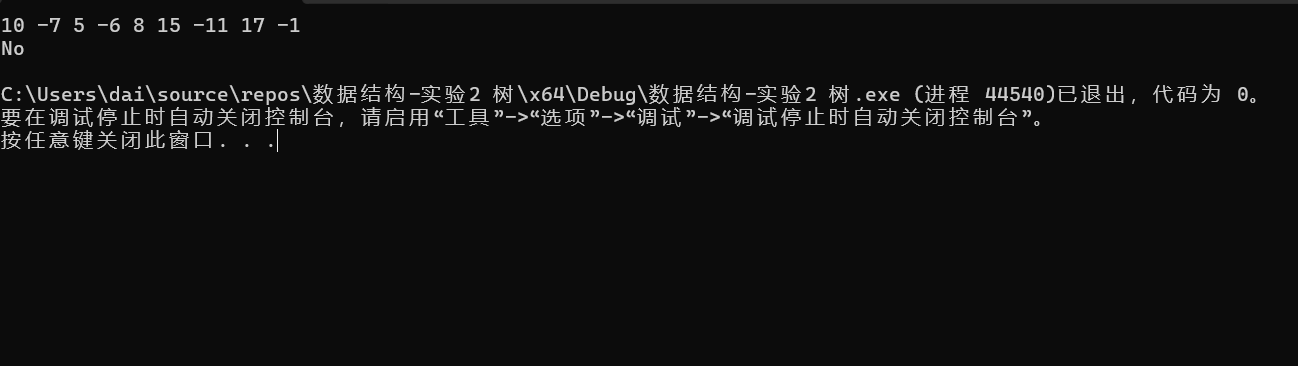
IsRDT(head);

}

1. **Results:** *//run your implementation with some input, then show and explain your outputs.*

****

****

****

**[Discussions]:** //*talk about your observations from this experiments, for example, what kind of experiences you have learned? Are there any key points that you should pay attention to in your study or programming exercises in future?*

I learned some operations about balancing binary trees, mainly recursive operations are important, and secondly, I was exposed to red-black trees for the first time, and the complex problems of red-black trees further stimulated my ability to think independently, and I gained a lot this time.

**Evaluation:**

Functionality

Completeness

Readability

======================================================================