

实验五实验报告

2112514 辛浩然

实验内容

实现对于二叉搜索树的如下操作：

1. 给定序列，使用逐点插入法构建二叉搜索树；
2. 使用非递归的方法按照升序序列输出上述二叉搜索树的关键字；
3. 判断上述二叉搜索树是否为AVL搜索树，若不是，则将其转化为AVL搜索树。

备注：输出树时将结点的左右孩子结点在括号内输出，若为空则输出#

实验代码

```
#include <iostream>
#include <queue>
#include <stack>
using namespace std;
class SortBinaryTree;
class BinaryTreeNode
{
    int data;
    int bf;
    BinaryTreeNode* left, * right;
    friend class SortBinaryTree;

public:
    BinaryTreeNode()
    {
        data = 0;
        bf = 0;
        left = nullptr;
        right = nullptr;
    }
    BinaryTreeNode(int x)
    {
        data = x;
        bf = 0;
        left = nullptr;
        right = nullptr;
    }
    BinaryTreeNode(int x, BinaryTreeNode* l, BinaryTreeNode* r)
    {
        data = x;
        bf = 0;
        left = l;
        right = r;
    }
};
class SortBinaryTree
{
private:
    BinaryTreeNode* root;
    int getHeight(BinaryTreeNode* tmp);
```

```

public:
    SortBinaryTree() { root = nullptr; }
    void createSortBinaryTree();
    void OutPut();
    void OutPutData();
    void toAVL();
    bool isAVL();
};
// 逐点插入法构建二叉搜索树
void SortBinaryTree::createSortBinaryTree()
{
    int val;
    while (cin >> val)
    {
        if (!root)
        {
            root = new BinaryTreeNode(val);
        }
        else
        {
            BinaryTreeNode* tmp = root;
            while (tmp)
            {
                if (val == tmp->data)
                {
                    break;
                }
                else if (val > tmp->data)
                {
                    if (tmp->right)
                    {
                        tmp = tmp->right;
                    }
                    else
                    {
                        tmp->right = new BinaryTreeNode(val);
                        break;
                    }
                }
                else if (val < tmp->data)
                {
                    if (tmp->left)
                    {
                        tmp = tmp->left;
                    }
                    else
                    {
                        tmp->left = new BinaryTreeNode(val);
                        break;
                    }
                }
            }
        }
        if (cin.get() == '\n')
            break;
    }
}
//输出二叉搜索树
void SortBinaryTree::OutPut()
{
    int curNum = 1, nextNum = 0;

```

```

if (root)
{
    queue<BinaryTreeNode*> s;
    s.push(root);
    while (!s.empty())
    {
        BinaryTreeNode* head = s.front();
        s.pop();
        cout << head->data << '(';
        curNum--;
        if (head->left)
        {
            cout << head->left->data << ',';
            s.push(head->left);
            nextNum++;
        }
        else
        {
            cout << "#,";
        }
        if (head->right)
        {
            cout << head->right->data << ")" ";
            s.push(head->right);
            nextNum++;
        }
        else
        {
            cout << "#) ";
        }
        if (!curNum)
        {
            cout << endl;
            curNum = nextNum;
            nextNum = 0;
        }
    }
}

//中序输出
void SortBinaryTree::OutPutData()
{
    stack<BinaryTreeNode*> s;
    BinaryTreeNode* tmp = root;
    while (tmp || (!s.empty()))
    {
        if (tmp)
        {
            s.push(tmp);
            tmp = tmp->left;
        }
        else
        {
            tmp = s.top();
            s.pop();
            cout << tmp->data << ' ';
            tmp = tmp->right;
        }
    }
    cout << endl;
}

```

```

int SortBinaryTree::getHeight(BinaryTreeNode* tmp)
{
    int h = 0, hl = 0, hr = 0;
    if (!tmp)
        return 0;
    hl = getHeight(tmp->left);
    hr = getHeight(tmp->right);
    h = (hr > hl) ? hr : hl;
    return h + 1;
}
//判断是否为二叉平衡树
bool SortBinaryTree::isAVL()
{
    bool rightBF = 1;
    queue<BinaryTreeNode*> s;
    s.push(root);
    while (!s.empty())
    {
        BinaryTreeNode* head = s.front();
        s.pop();
        head->bf = getHeight(head->left) - getHeight(head->right);
        if (head->bf != 1 && head->bf != 0 && head->bf != -1)
            rightBF = 0;
        if (head->left)
        {
            s.push(head->left);
        }
        if (head->right)
        {
            s.push(head->right);
        }
    }
    return rightBF;
}
//转化为二叉平衡树
void SortBinaryTree::toAVL()
{
    while (!isAVL())
    {
        queue<BinaryTreeNode*> s;
        stack<BinaryTreeNode*> BadNode;
        stack<BinaryTreeNode*> BadParent;
        //建立两个栈，一个存放平衡因子异常的结点，一个存放平衡因子异常结点的父母。
        s.push(root);
        if (root->bf != 1 && root->bf != 0 && root->bf != -1)
        {
            BadNode.push(root);
            BadParent.push(nullptr);
        }
        while (!s.empty())
        {
            BinaryTreeNode* head = s.front();
            s.pop();
            if (head->left)
            {
                if (head->left->bf != 1 && head->left->bf != 0 && head->left->bf != -1)
                {
                    BadNode.push(head->left);
                    BadParent.push(head);
                }
                s.push(head->left);
            }
        }
    }
}

```

```

    }
    if (head->right)
    {
        if (head->right->bf != 1 && head->right->bf != 0 && head->right->bf != -1)
        {
            BadNode.push(head->right);
            BadParent.push(head);
        }
        s.push(head->right);
    }
}

BinaryTreeNode* badnode = BadNode.top();
BinaryTreeNode* badparent = BadParent.top();
//获取两个栈顶元素，得到最后一个异常结点和其父母。
if (badnode->bf > 0) //如果其平衡因子大于0，需要处理其左子树使之平衡
{
    if (badnode->left->bf > 0) // LL
    {
        BinaryTreeNode* tmp = badnode->left->right;
        BinaryTreeNode* tmp2 = badnode->left;
        badnode->left->right = badnode;
        badnode->left = tmp;
        if (badparent)
        {
            if (badparent->left == badnode)
            {
                badparent->left = tmp2;
            }
            else
            {
                badparent->right = tmp2;
            }
        }
        else
        {
            root = tmp2;
        }
    }
    else // LR
    {
        BinaryTreeNode* tmp = badnode->left->right->left;
        BinaryTreeNode* tmp2 = badnode->left->right->right;
        BinaryTreeNode* newRoot = badnode->left->right;
        badnode->left->right->left = badnode->left;
        badnode->left->right->right = badnode;
        badnode->left->right = tmp;
        badnode->left = tmp2;
        if (badparent)
        {
            if (badparent->left == badnode)
            {
                badparent->left = newRoot;
            }
            else
            {
                badparent->right = newRoot;
            }
        }
        else
        {
            root = newRoot;
        }
    }
}

```

```

    }
}
}
else if (badnode->bf < 0) //如果其平衡因子小于0，需要处理其右子树使之平衡
{
    if (badnode->right->bf < 0) // RR
    {
        BinaryTreeNode* tmp = badnode->right->left;
        BinaryTreeNode* tmp2 = badnode->right;
        badnode->right->left = badnode;
        badnode->right = tmp;
        if (badparent)
        {
            if (badparent->left == badnode)
            {
                badparent->left = tmp2;
            }
            else
            {
                badparent->right = tmp2;
            }
        }
        else
        {
            root = tmp2;
        }
    }
    else // RL
    {
        BinaryTreeNode* tmp = badnode->right->left->left;
        BinaryTreeNode* tmp2 = badnode->right->left->right;
        BinaryTreeNode* newRoot = badnode->right->left;
        badnode->right->left->left = badnode;
        badnode->right->left->right = badnode->right;
        badnode->right->left = tmp2;
        badnode->right = tmp;
        if (badparent)
        {
            if (badparent->left == badnode)
            {
                badparent->left = newRoot;
            }
            else
            {
                badparent->right = newRoot;
            }
        }
        else
        {
            root = newRoot;
        }
    }
}
}
}

int main()
{
    SortBinaryTree a;
    a.createSortBinaryTree();
    cout << "BST:" << endl;
    a.OutPut();
}

```

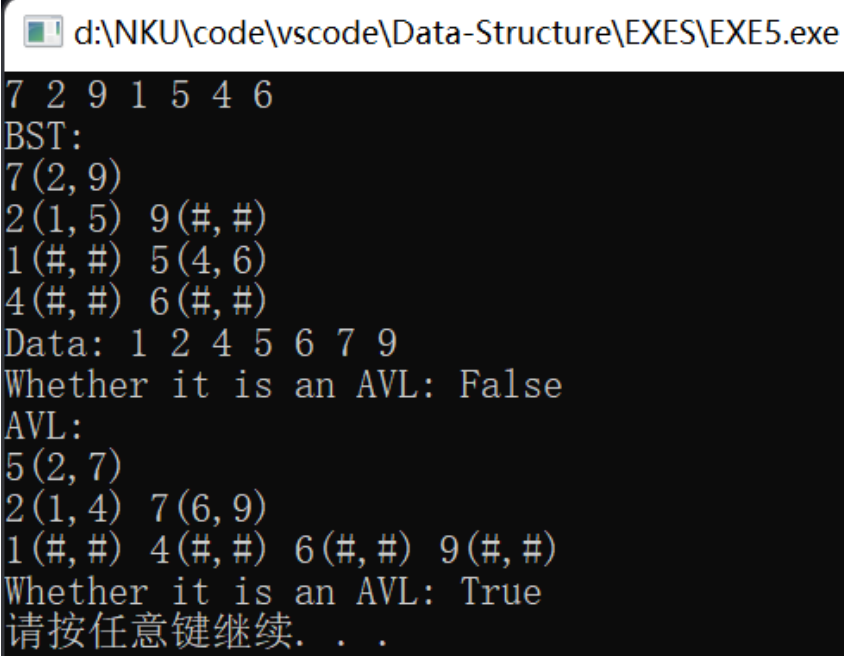
```

cout << "Data: ";
a.OutPutData();
cout << "Whether it is an AVL: " << (a.isAVL() ? "True" : "False") << endl;
if (!a.isAVL())
{
    cout << "AVL:" << endl;
    a.toAVL();
    a.OutPut();
    cout << "Whether it is an AVL: " << (a.isAVL() ? "True" : "False") << endl;
}
system("pause");
}

```

测试用例

测试用例一

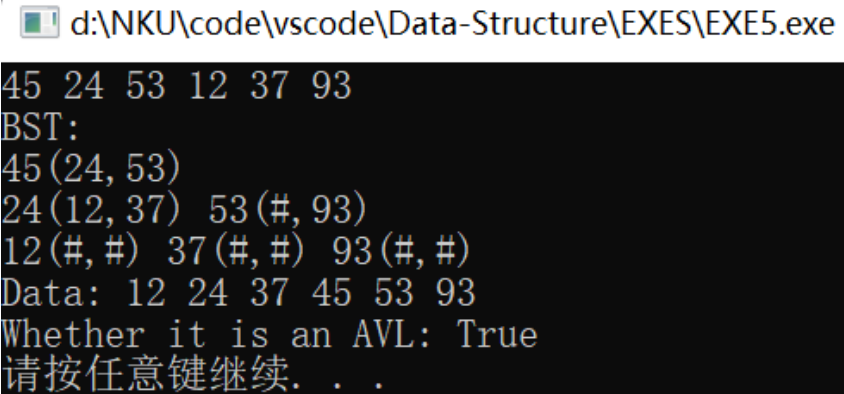


```

d:\NKU\code\vscode\Data-Structure\EXES\EXE5.exe
7 2 9 1 5 4 6
BST:
7(2, 9)
2(1, 5) 9(#, #)
1(#, #) 5(4, 6)
4(#, #) 6(#, #)
Data: 1 2 4 5 6 7 9
Whether it is an AVL: False
AVL:
5(2, 7)
2(1, 4) 7(6, 9)
1(#, #) 4(#, #) 6(#, #) 9(#, #)
Whether it is an AVL: True
请按任意键继续. . .

```

测试用例二



```

d:\NKU\code\vscode\Data-Structure\EXES\EXE5.exe
45 24 53 12 37 93
BST:
45(24, 53)
24(12, 37) 53(#, 93)
12(#, #) 37(#, #) 93(#, #)
Data: 12 24 37 45 53 93
Whether it is an AVL: True
请按任意键继续. . .

```

测试用例三

d:\NKU\code\vscode\Data-Structure\EXES\EXE5.exe


```
18 39 56 78 79 91 671 990
BST:
18(#, 39)
39(#, 56)
56(#, 78)
78(#, 79)
79(#, 91)
91(#, 671)
671(#, 990)
990(#, #)
Data: 18 39 56 78 79 91 671 990
Whether it is an AVL: False
AVL:
78(39, 91)
39(18, 56) 91(79, 671)
18(#, #) 56(#, #) 79(#, #) 671(#, 990)
990(#, #)
Whether it is an AVL: True
请按任意键继续. . .
```

测试用例四

d:\NKU\code\vscode\Data-Structure\EXES\EXE5.exe

```
1024 675 454 381 281 4 3 2 1
BST:
1024(675, #)
675(454, #)
454(381, #)
381(281, #)
281(4, #)
4(3, #)
3(2, #)
2(1, #)
1(#, #)
Data: 1 2 3 4 281 381 454 675 1024
Whether it is an AVL: False
AVL:
281(3, 675)
3(2, 4) 675(454, 1024)
2(1, #) 4(#, #) 454(381, #) 1024(#, #)
1(#, #) 381(#, #)
Whether it is an AVL: True
请按任意键继续. . .
```

测试用例五

 d:\NKU\code\vscode\Data-Structure\EXES\EXE5.exe

```
98 87 76 65 453 451 1 2 5 4 3 1 871 10
BST:
98(87, 453)
87(76, #) 453(451, 871)
76(65, #) 451(#, #) 871(#, #)
65(1, #)
1(#, 2)
2(#, 5)
5(4, 10)
4(3, #) 10(#, #)
3(#, #)
Data: 1 2 3 4 5 10 65 76 87 98 451 453 871
Whether it is an AVL: False
AVL:
76(5, 98)
5(3, 65) 98(87, 453)
3(2, 4) 65(10, #) 87(#, #) 453(451, 871)
2(1, #) 4(#, #) 10(#, #) 451(#, #) 871(#, #)
1(#, #)
Whether it is an AVL: True
请按任意键继续. . .
```

思路分析

逐点插入法构建BST

- 输入元素：
 - 根结点为空，将元素值赋值给根结点的 data；
 - 根结点不为空，与根结点的 data 值比较：
 - 若小于根结点的 data 值，插入到根结点的左子树；
 - 若大于根结点的 data 值，插入到根结点的右子树；
 - 与左子树/右子树的根结点进行比较，循环上述操作，直到左子树/右子树为空，将元素插入该位置。

非递归方式升序输出关键字

- 根据BST的定义，左子树值<根结点值<右子树值。升序输出关键字，即对BST进行中序遍历。
- 中序遍历的遍历顺序：
 - 访问当前节点的左子树；
 - 访问根节点；
 - 访问当前节点的右子树。
- 从根节点一直往下寻找左子结点，经过的结点都存到栈中，直到某个结点左孩子为空，那么该结点就是输出的第一个节点。
- 从已经输出的最后一个结点的右节点开始，重复上步。
- 创建栈，定义 `BinaryTreeNode *` 型指针 `tmp` 指向 `root`：
 - 如果栈非空或者 `tmp` 非空
 - 如果 `tmp` 非空
 - `tmp` 压入栈，保存该结点；
 - `tmp` 指向该结点左孩子；
 - 如果 `tmp` 为空

- 输出栈顶 data 值;
- tmp 指向该结点右孩子;
- 循环上述操作, 直到栈空且 tmp 为空。

判断是否为AVL

- 层序遍历二叉搜索树, 对于每个结点, 将其左子树高度减去其右子树高度, 得到其平衡因子。
 - 通过递归得到树的高度。对于每个结点, 求其左子树高度和右子树高度, 返回较高的高度+1。
- 如果存在结点的平衡因子不为1、0、-1, 则该树不为AVL; 如果所有结点的平衡因子都为1/0/-1, 则该树为AVL。

转化为AVL

- 首先找到**最后一个**平衡因子异常的结点:
 - 利用队列层序遍历树:
 - 建立两个栈, 一个存放平衡因子异常的结点, 一个存放平衡因子异常结点的父母。
 - 对于每个结点:
 - 如果根结点平衡因子异常, 将根结点入栈,
 - 如果其左孩子平衡因子异常, 将左孩子存入异常栈, 将该结点存入异常父母栈。
 - 如果其右孩子平衡因子异常, 将右孩子存入异常栈, 将该结点存入异常父母栈。
 - 获取两个栈栈顶元素, 得到最后一个异常结点和其父母。
 - 处理最后一个平衡因子异常的结点A:
 - 如果其平衡因子大于0, 需要处理其左子树使之平衡:
 - 如果其左结点平衡因子大于0, 使用LL转换方法:
 - A的左孩子B代替A成为根结点, A结点成为B的右孩子, 而B的原右孩子成为A的左孩子。
 - 如果其左结点平衡因子小于等于0, 使用LR转换方法:
 - A结点的左孩子B的右孩子C成为根结点, 其左孩子为B, 右孩子为A, C原本的左孩子成为B的右孩子, C原本的右孩子成为A的左孩子。
 - 如果其平衡因子小于0, 需要处理其右子树使之平衡:
 - 如果其右结点平衡因子大于0, 使用RR转换方法:
 - A的右孩子B代替A成为根结点, A结点成为B的左孩子, 而B的原左孩子成为A的右孩子。
 - 如果其右结点平衡因子小于0, 使用RL转换方法:
 - A结点的右孩子B的左孩子C成为根结点, 其左孩子为A, 右孩子为B, C原本的左孩子成为A的右孩子, C原本的右孩子成为B的左孩子。
 - 循环上步操作, 直到其为AVL。

心得体会

1. 通过这次实验, 对BST和AVL的理解更加深入。
2. 尤其是AVL的插入过程, 通过课上讲解虽然理解但还有些混乱, 通过自己写将BST转换为AVL的操作, 理清了旋转过程。