重庆交通大学信息科学与工程学院

《数据结构 A》实验报告

实验名称	二叉树实验
课程名称	数据结构 A
专业班级	曙光 2101 班
学　　号	632107060510
姓　　名	王靖
指导教师	鲁云平

2022 年 11 月

《数据结构 A》综合性实验评分标准

评分等级	综合性实验评分标准
优	程序演示完全正确，界面美观，能正确回答 90%及以上的问题；报告规范，分析清楚，严格按照要求条目书写，阐述清楚。能针对综合性题目进行分析，并设计合适的解决方案，同时使用合适的编程平台进行编程、调试、测试和实现。
良	按要求完成 80%及以上功能，界面尚可，能正确回答 80%及以上的问题；报告规范，分析清楚，个别条目书写不完全符合要求，阐述基本清楚。能针对综合性题目进行分析，并设计较合适的解决方案，同时使用合适的编程平台进行编程、调试、测试和实现。
中	按要求完成 70%及以上功能，能回答 70%及以上的问题；报告基本规范，分析基本清楚，存在 30%以内条目书写不完全符合要求。在教师的指导下，能针对综合性题目进行分析，并设计较合适的解决方案，同时使用合适的编程平台进行编程、调试、测试和实现。
及格	按要求完成 60%及以上功能，能回答老师多数问题；报告基本规范，存在 40%以内条目书写不完全符合要求。在教师的指导下，能针对综合性题目进行分析和设计较合适的解决方案，并能使用合适的编程平台进行编程、调试、测试和实现，成功调试功能达 60%以上。
不及格	存在 40%以上功能未完成或抄袭；报告不规范或存在 40%以上条目书写不完全符合要求。无法采用正确的方法完成项目分析和解决方案设计或成功调试测试功能超过 40%未完成。

教师签名	
综合性实验成绩	

说明:

1、综合性实验报告总体要求为格式规范，内容丰富、全面、深入；严禁大量拷贝。

2、请提交 word 文档和 PDF 文档，文件命名格式：学号-姓名。

3、报告主要内容参考下页示例

目录

一、实验目的

1、实现二叉树的链式存储结构

2、熟悉二叉树基本术语的含义

3、掌握相关操作的编程实现

4、实现二叉树的一些基本操作

二、实验内容

采用面向对象和泛型编程实现整个二叉树

1.建立二叉树

2.统计结点数量和叶结点数量

3.计算二叉树的高度

4.计算结点的度

5.找结点的双亲和子女

6.二叉树前序、中序、后序遍历的递归实现和非递归实现及层次遍历

7.二叉树的输出

8.求二叉树的最大深度和最小深度

9.翻转二叉树

10.通过前序遍历和中序遍历生成二叉树

11.通过中序遍历和后序遍历生成二叉树

三、系统分析

（1）数据方面：采用标准数据和引用数据

（2）功能方面：

1.建立二叉树

2.统计结点数量和叶结点数量

3.计算二叉树的高度

4.计算结点的度

5.找结点的双亲和子女

6.二叉树前序、中序、后序遍历的递归实现和非递归实现及层次遍历

7.二叉树的输出

8.求二叉树的最大深度和最小深度

9.翻转二叉树

10.通过前序遍历和中序遍历生成二叉树

11.通过中序遍历和后序遍历生成二叉树

四、系统设计

（1）数据结构的设计

- 二叉树节点类

```
template<class T>
class TreeNode
{
public:
    T data; //数据
    TreeNode<T>* left; //左指针
    TreeNode<T>* right;//右指针
public:
    TreeNode();//构造函数
    TreeNode(T data);//带参构造函数
    ~TreeNode();//析构函数
};
```

- 二叉树类

```
template<class T>
class Tree
{
private:
    TreeNode<T>* root; //树根
    int numOfNode; //节点数量
public:
    TreeNode<T>* stk[N]; //模拟栈
    TreeNode<T>* que[N]; //模拟队列
    Tree();//构造函数
    Tree(char s[]);//带参构造函数
    ~Tree();//析构函数
    void createTree(char s[]);//创建二叉树
    //根据前序和中序遍历序列生成二叉树
    TreeNode<T>* createTreeByPreIn(char pre[], char in[], int n);
    void createTreeByPreAndIn(char pre[], char in[], int n);
    //根据中序和后序遍历序列生成二叉树
    TreeNode<T>* createTreeByInPost(char in[], char post[], int n);
    void createTreeByInAndPost(char in[], char post[], int n);
    TreeNode<T>* getRoot();//获取根节点
    int getNumOfNode();//获取节点个数
    void preOrder(TreeNode<T>* node);//前序遍历
    void inOrder(TreeNode<T>* node);//中序遍历
    void postOrder(TreeNode<T>* node);//后序遍历
    void preOrderByStack();//前序遍历（栈）
    void inOrderByStack();//中序遍历（栈）
```

```
    void postOrderByStack();//后序遍历（栈）
    void levelOrder();//层序遍历
    TreeNode<T>* leftChild(TreeNode<T>* node);//找左子节点
    TreeNode<T>* rightChild(TreeNode<T>* node);//找右子节点
    int duOfNode(TreeNode<T>* node);//节点的度
    int heigh(TreeNode<T>* node);//高度
    TreeNode<T>* invertTree(TreeNode<T>* node);//翻转二叉树
    int maxDepth(TreeNode<T>* node);//二叉树的最大深度
    int minDepth(TreeNode<T>* node);//二叉树的最小深度
    void dispTree(TreeNode<T>* node);//输出二叉树
};
```

（2）基本操作的设计

关键算法的设计思路和算法流程图。

一、建立二叉树

鉴于二叉树的特殊结构，我们采用 root（left,right）此类的字符串建立二叉树，采用顺序栈的结构，分一下几步完成整个二叉树建立。

- 如果当前的字符是字母，建立结点，如果 root 为空则为空结点，否则如果 k = 1,栈顶左子树为当前字符，k = 2, 栈顶右子树为当前字符。
- 如果当前字符为左括号，当前新建的结点入栈，k = 1。
- 如果当前字符为右括号，栈顶结点出栈。
- 如果当前字符为逗号，k = 2。

二、统计结点数量

考虑使用递归来完成，对于每个结点，递归计算左子树和右子树，返回两者之和再加入加一，递归结束条件，当前节点为空节点则返回 0。

三、计算树的高度

考虑使用递归实现，递归思路是计算左子树和右子树的树高度，返回当前左子树和右子树当中更高的数量加一，递归结束条件为当前节点为 0。

四、查找结点、查找结点的双亲

这个两个方法采用的递归思想极度的类似，故将其放在一起说明。首先如果当前节点为空或当前节点的数据与待查找的一致则返回当前节点。其次，查找当前节点的左子树，如果返回值不为空则返回左子树递归结果，反之放回右子树结果。

五、前序、中、后序遍历递归版

考虑到这三种遍历的递归版几乎一样，此处一起说明。三种遍历的定义便不

再赘述，拿前序遍历为例，递归思路是先判断当前节点是否为空如果为空，便返回，其次先输出当前节点的值，再分别递归左子树、右子树。其他两个只是输出节点的顺序不一样。

六、树的前、中序遍历非递归版

由于前、中序遍历的非递归版的算法相似，同样一起分析。先分析前序遍历，我们采用顺序栈来存放节点，先访问根节点，随后一直访问节点的左子树，不断的入栈，直到为空。并访问节点的右子树，直到栈空。重复这种操作，直到节点为空并且栈为空。后序遍历只需要在前序便利的基础上，将输出值放在出栈时即可。

七、树的后序遍历非递归版

该算法在前、中序遍历的非递归版上，加了一些改动，在二阶段处理右子树时，我们要保存上次出栈的节点，如果上次出栈的结点为栈顶结点的右子树，则可以将栈顶元素输出，并出栈。具体实现比较繁琐，在系统实现上有具体代码。

八、二叉树的输出

采用递归算法，实现输出如构建二叉树时字符串形式 root(left,right),当前节点为空则放回，输出当前节点，如果左右子树都为空则放回，输出“(”，递归左子树，如果右子树不空，输出“，”，否则递归右子树，输出 “)”。

九、二叉树的翻转

采用递归算法，实现左右子树的交换形成翻转之后的二叉树，依托于二叉树的二叉链式存储特性

五、系统实现

主要功能的实现代码

- 创建二叉树

```
template<class T>
void Tree<T>::createTree(char s[])
{
    root = NULL;
    TreeNode<T>* p = NULL;
    int tp = -1, k;
    for (int i = 0; s[i] != '\0'; i++)
    {
        if (s[i] == '(')
        {
            stk[++tp] = p;
            k = 1;
```

```
        }
        else if (s[i] == ')') tp--;
        else if (s[i] == ',') k = 2;
        else
        {
            p = new TreeNode<T>(s[i]);
            if (root == NULL) root = p;
            else
            {
                if (k == 1) stk[tp]->left = p;
                if (k == 2) stk[tp]->right = p;
            }
            numOfNode++;
        }
    }
}
```

- 二叉树的前序遍历

```
//前序遍历
template<class T>
void Tree<T>::preOrder(TreeNode<T>* node)
{
    if (node != NULL)
    {
        cout << node->data;
        preOrder(node->left);
        preOrder(node->right);
    }
}
//前序遍历（栈）
template<class T>
void Tree<T>::preOrderByStack()
{
    if (root == NULL) return;
    TreeNode<T>* p = root;
    int tp = -1;
    while (tp > -1 || p != NULL)
    {
        while (p != NULL)
        {
            cout << p->data;
            stk[++tp] = p;
            p = p->left;
```

```
        }
        if (tp > -1)
        {
            p = stk[tp--];
            p = p->right;
        }
    }
    cout << endl;
}
```

- 二叉树的中序遍历

```
//中序遍历
template<class T>
void Tree<T>::inOrder(TreeNode<T>* node)
{
    if (node != NULL)
    {
        inOrder(node->left);
        cout << node->data;
        inOrder(node->right);
    }
}
//中序遍历（栈）
template<class T>
void Tree<T>::inOrderByStack()
{
    if (root == NULL) return;
    TreeNode<T>* p = root;
    int tp = -1;
    while (tp > -1 || p != NULL)
    {
        while (p != NULL)
        {
            stk[++tp] = p;
            p = p->left;
        }
        if (tp > -1)
        {
            p = stk[tp--];
            cout << p->data;
            p = p->right;
        }
    }
    cout << endl;
```

```
}
```

- 二叉树的后序遍历

```
//后序遍历
template<class T>
void Tree<T>::postOrder(TreeNode<T>* node)
{
    if (node != NULL)
    {
        postOrder(node->left);
        postOrder(node->right);
        cout << node->data;
    }
}
//后序遍历（栈）
template<class T>
void Tree<T>::postOrderByStack()
{
    if (root == NULL) return;
    TreeNode<T>* p = root;
    TreeNode<T>* r;
    bool flag;
    int tp = -1;
    do
    {
        while (p != NULL)
        {
            stk[++tp] = p;
            p = p->left;
        }
        r = NULL;
        flag = true;
        while (tp > -1 && flag)
        {
            p = stk[tp];
            if (p->right == r)
            {
                cout << p->data;
                tp--;
                r = p;
            }
            else
            {
                p = p->right;
```

```
            flag = false;
        }
    }
} while (tp > -1);
cout << endl;
}
```

- 二叉树的层序遍历

```
template<class T>
void Tree<T>::levelOrder()
{
    if (root == NULL) return;
    TreeNode<T>* tmp;
    int hh = 0, tt = -1;
    que[++tt] = root;
    while (hh <= tt)
    {
        tmp = que[hh++];
        cout << tmp->data;
        if (tmp->left != NULL) que[++tt] = tmp->left;
        if (tmp->right != NULL) que[++tt] = tmp->right;
    }
    cout << endl;
}
```

- 找二叉树的左右子节点

```
//找左子节点
template<class T>
TreeNode<T>* Tree<T>::leftChild(TreeNode<T>* node)
{
    return node->left;
}
//找右子节点
template<class T>
TreeNode<T>* Tree<T>::rightChild(TreeNode<T>* node)
{
    return node->right;
}
```

- 求节点的度

```
template<class T>
int Tree<T>::duOfNode(TreeNode<T>* node)
{
    return (node->left != NULL) + (node->right != NULL);
}
```

- 获取根节点

```
template<class T>
TreeNode<T>* Tree<T>::getRoot()
{
    return root;
}
```

- 获取节点个数

```
template<class T>
int Tree<T>::getNumOfNode()
{
    return numOfNode;
}
```

- 求二叉树的高度

```
template<class T>
int Tree<T>::heigh(TreeNode<T>* node)
{
    if (node == NULL) return 0;
    int hl = 0, hr = 0;
    hl = heigh(node->left);
    hr = heigh(node->right);
    return max(hl, hr) + 1;
}
```

- 求二叉树的最大深度

```
template<class T>
int Tree<T>::maxDepth(TreeNode<T>* node)
{
    if (node == NULL) return 0;
    return max(maxDepth(node->left), maxDepth(node->right)) + 1;
}
```

- 求二叉树的最小深度

```
template<class T>
int Tree<T>::minDepth(TreeNode<T>* node)
{
    if (node == NULL) return 0;
    if (node->left == NULL && node->right == NULL) return 1;
    int ans = INF;
    if (node->left != NULL) ans = min(minDepth(node->left), ans);
    if (node->right != NULL) ans = min(minDepth(node->right), ans);
    return ans + 1;
}
```

- 根据前序遍历和后序遍历生成二叉树

```
template<class T>
TreeNode<T>* Tree<T>::createTreeByPreIn(char pre[], char in[],
int n)
{
    int k;
    char* p;
    if (n <= 0) return NULL;
    TreeNode<T>* node = new TreeNode<T>(*pre);
    numOfNode++;
    for (p = in; p < in + n; p++)
    {
        if (*p == *pre) break;
    }
    k = p - in;
    node->left = createTreeByPreIn(pre + 1, in, k);
    node->right = createTreeByPreIn(pre + k + 1, p + 1, n - k - 1);
    return node;
}
template<class T>
void Tree<T>::createTreeByPreAndIn(char pre[], char in[], int n)
{
    root = createTreeByPreIn(pre, in, n);
}
```

- 根据中序遍历和后序遍历生成二叉树

```
template<class T>
TreeNode<T>* Tree<T>::createTreeByInPost(char in[], char post[],
int n)
{
    int k;
    char* p;
    char r;
    if (n <= 0) return NULL;
    r = *(post + n - 1);
    TreeNode<T>* node = new TreeNode<T>(r);
    numOfNode++;
    for (p = in; p < in + n; p++)
    {
        if (*p == r) break;
    }
    k = p - in;
    node->left = createTreeByInPost(in, post, k);
    node->right = createTreeByInPost(p + 1, post + k, n - k - 1);
    return node;
}
```

```
template<class T>
void Tree<T>::createTreeByInAndPost(char in[], char post[], int
n)
{
    root = createTreeByInPost(in, post, n);
}
```

- 翻转二叉树

```
template<class T>
TreeNode<T>* Tree<T>::invertTree(TreeNode<T>* node)
{
    if (node == NULL) return NULL;
    TreeNode<T>* pLeft = invertTree(node->left);
    TreeNode<T>* pRight = invertTree(node->right);
    node->left = pRight;
    node->right = pLeft;
    return node;
}
```

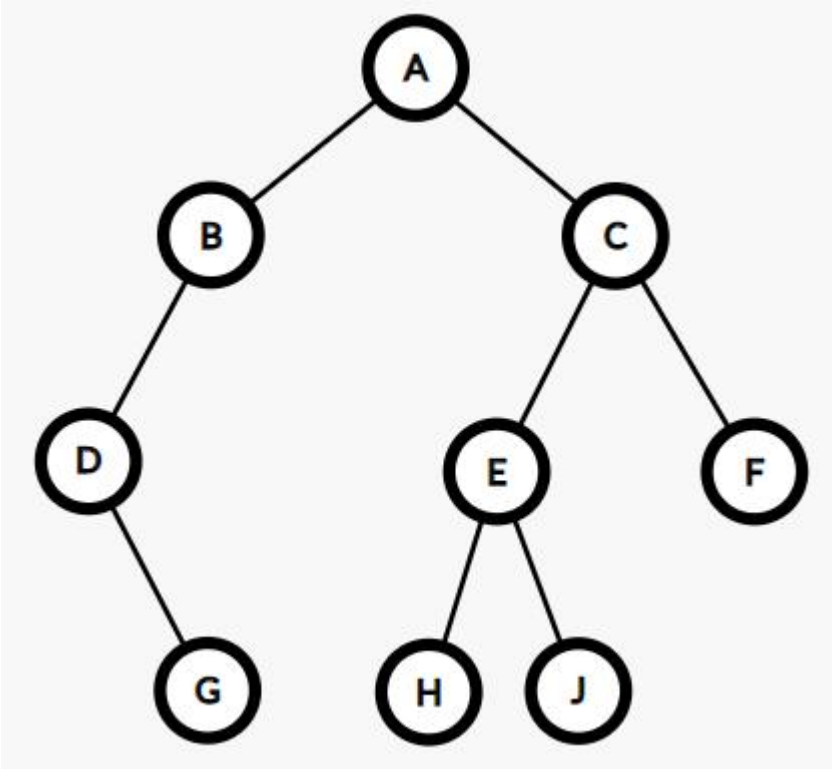
- 输出二叉树

```
template<class T>
void Tree<T>::dispTree(TreeNode<T>* node)
{
    if (node != NULL)
    {
        cout << node->data;
        if (node->left != NULL || node->right != NULL)
        {
            cout << '(';
            dispTree(node->left);
            if (node->right != NULL) cout << ',';
            dispTree(node->right);
            cout << ')';
        }
    }
}
```

六、系统测试与结果

至少完成功能测试，使用测试数据测试相关功能是否符合设计要求。

- 生成如下所示的二叉树



- 根据字符串生成

测试函数

```
void test_01()
{
    cout << "根据字符串生成二叉树" << endl;
    Tree<char> t1;
    char s[] = "A(B(D(,G),),C(E(H,J),F))";
    t1.createTree(s);
    cout << "节点个数：";
    cout << t1.getNumOfNode() << endl;
    cout << "前序遍历：";
    t1.preOrder(t1.getRoot());
    cout << endl << "中序遍历：";
    t1.inOrder(t1.getRoot());
    cout << endl << "后序遍历：";
    t1.postOrder(t1.getRoot());

    cout << endl << "前序遍历（非递归）：";
    t1.preOrderByStack();
    cout << "中序遍历（非递归）：";
    t1.inOrderByStack();
    cout << "后序遍历（非递归）：";
    t1.postOrderByStack();
```

```
    cout << "层序遍历：";
    t1.levelOrder();
    cout << "输出二叉树：";
    t1.dispTree(t1.getRoot());

    cout << endl << "翻转二叉树：";
    t1.invertTree(t1.getRoot());
    cout << endl;

    cout << "前序遍历：";
    t1.preOrder(t1.getRoot());
    cout << endl << "中序遍历：";
    t1.inOrder(t1.getRoot());
    cout << endl << "后序遍历：";
    t1.postOrder(t1.getRoot());

    cout << endl << "前序遍历（非递归）：";
    t1.preOrderByStack();
    cout << "中序遍历（非递归）：";
    t1.inOrderByStack();
    cout << "后序遍历（非递归）：";
    t1.postOrderByStack();

    cout << "层序遍历：";
    t1.levelOrder();
    cout << "输出二叉树：";
    t1.dispTree(t1.getRoot());

    cout << endl << "最大深度：";
    cout << t1.maxDepth(t1.getRoot()) << endl;
    cout << "最小深度：";
    cout << t1.minDepth(t1.getRoot()) << endl;
}
```

运行结果

```
根据字符串生成二叉树
节点个数：9
前序遍历：ABDGCEHJF
中序遍历：DGBAHEJCF
后序遍历：GDBHJEFCA
前序遍历（非递归）：ABDGCEHJF
中序遍历（非递归）：DGBAHEJCF
后序遍历（非递归）：GDBHJEFCA
层序遍历：ABCDEFGHJ
输出二叉树：A(B(D(,G)),C(E(H,J),F))
翻转二叉树：
前序遍历：ACFEJHBDG
中序遍历：FCJEHABGD
后序遍历：FJHECGDBA
前序遍历（非递归）：ACFEJHBDG
中序遍历（非递归）：FCJEHABGD
后序遍历（非递归）：FJHECGDBA
层序遍历：ACBFEDJHG
输出二叉树：A(C(F,E(J,H)),B(,D(G)))
最大深度：4
最小深度：3
```

- 根据前序遍历和中序遍历生成

测试函数

```
void test_02()
{
    cout << "根据前序遍历和中序遍历生成二叉树" << endl;
    char s1[] = "ABDGCEHJF";
    char s2[] = "DGBAHEJCF";
    Tree<char> t1;
    t1.createTreeByPreAndIn(s1, s2, 9);

    cout << "节点个数：";
    cout << t1.getNumOfNode() << endl;
    cout << "前序遍历：";
    t1.preOrder(t1.getRoot());
    cout << endl << "中序遍历：";
    t1.inOrder(t1.getRoot());
    cout << endl << "后序遍历：";
    t1.postOrder(t1.getRoot());
```

```
    cout << endl << "前序遍历（非递归）：";
    t1.preOrderByStack();
    cout << "中序遍历（非递归）：";
    t1.inOrderByStack();
    cout << "后序遍历（非递归）：";
    t1.postOrderByStack();

    cout << "层序遍历：";
    t1.levelOrder();
    cout << "输出二叉树：";
    t1.dispTree(t1.getRoot());

    cout << endl << "翻转二叉树：";
    t1.invertTree(t1.getRoot());
    cout << endl;

    cout << "前序遍历：";
    t1.preOrder(t1.getRoot());
    cout << endl << "中序遍历：";
    t1.inOrder(t1.getRoot());
    cout << endl << "后序遍历：";
    t1.postOrder(t1.getRoot());

    cout << endl << "前序遍历（非递归）：";
    t1.preOrderByStack();
    cout << "中序遍历（非递归）：";
    t1.inOrderByStack();
    cout << "后序遍历（非递归）：";
    t1.postOrderByStack();

    cout << "层序遍历：";
    t1.levelOrder();
    cout << "输出二叉树：";
    t1.dispTree(t1.getRoot());

    cout << endl << "最大深度：";
    cout << t1.maxDepth(t1.getRoot()) << endl;
    cout << "最小深度：";
    cout << t1.minDepth(t1.getRoot()) << endl;
}
```

运行结果

```
根据前序遍历和中序遍历生成二叉树
节点个数：9
前序遍历：ABDGCEHJF
中序遍历：DGBAHEJCF
后序遍历：GDBHJEFCA
前序遍历（非递归）：ABDGCEHJF
中序遍历（非递归）：DGBAHEJCF
后序遍历（非递归）：GDBHJEFCA
层序遍历：ABCDEFGHJ
输出二叉树：A(B(D(,G)),C(E(H,J),F))
翻转二叉树：
前序遍历：ACFEJHBDG
中序遍历：FCJEHABGD
后序遍历：FJHECGDBA
前序遍历（非递归）：ACFEJHBDG
中序遍历（非递归）：FCJEHABGD
后序遍历（非递归）：FJHECGDBA
层序遍历：ACBFEDJHG
输出二叉树：A(C(F,E(J,H)),B(,D(G)))
最大深度：4
最小深度：3
```

■ 根据中序遍历和后序遍历生成

测试函数

```
void test_03()
{
    cout << "根据中序遍历和后序遍历生成二叉树" << endl;
    char s1[] = "GDBHJEFCA";
    char s2[] = "DGBAHEJCF";
    Tree<char> t1;
    t1.createTreeByInAndPost(s2, s1, 9);

    cout << "节点个数：";
    cout << t1.getNumOfNode() << endl;
    cout << "前序遍历：";
    t1.preOrder(t1.getRoot());
    cout << endl << "中序遍历：";
    t1.inOrder(t1.getRoot());
    cout << endl << "后序遍历：";
    t1.postOrder(t1.getRoot());
```

```
    cout << endl << "前序遍历（非递归）：";
    t1.preOrderByStack();
    cout << "中序遍历（非递归）：";
    t1.inOrderByStack();
    cout << "后序遍历（非递归）：";
    t1.postOrderByStack();

    cout << "层序遍历：";
    t1.levelOrder();
    cout << "输出二叉树：";
    t1.dispTree(t1.getRoot());

    cout << endl << "翻转二叉树：";
    t1.invertTree(t1.getRoot());
    cout << endl;

    cout << "前序遍历：";
    t1.preOrder(t1.getRoot());
    cout << endl << "中序遍历：";
    t1.inOrder(t1.getRoot());
    cout << endl << "后序遍历：";
    t1.postOrder(t1.getRoot());

    cout << endl << "前序遍历（非递归）：";
    t1.preOrderByStack();
    cout << "中序遍历（非递归）：";
    t1.inOrderByStack();
    cout << "后序遍历（非递归）：";
    t1.postOrderByStack();

    cout << "层序遍历：";
    t1.levelOrder();
    cout << "输出二叉树：";
    t1.dispTree(t1.getRoot());

    cout << endl << "最大深度：";
    cout << t1.maxDepth(t1.getRoot()) << endl;
    cout << "最小深度：";
    cout << t1.minDepth(t1.getRoot()) << endl;
}
```

运行结果

```
根据中序遍历和后序遍历生成二叉树
节点个数：9
前序遍历：ABDGCEHJF
中序遍历：DGBAHEJCF
后序遍历：GDBHJEFCA
前序遍历（非递归）：ABDGCEHJF
中序遍历（非递归）：DGBAHEJCF
后序遍历（非递归）：GDBHJEFCA
层序遍历：ABCDEFGHJ
输出二叉树：A(B(D(,G)),C(E(H,J),F))
翻转二叉树：
前序遍历：ACFEJHBDG
中序遍历：FCJEHABGD
后序遍历：FJHECGDBA
前序遍历（非递归）：ACFEJHBDG
中序遍历（非递归）：FCJEHABGD
后序遍历（非递归）：FJHECGDBA
层序遍历：ACBFEDJHG
输出二叉树：A(C(F,E(J,H)),B(,D(G)))
最大深度：4
最小深度：3
```

七、实验分析

（1）算法的性能分析

鉴于二叉树多数操作比较简单，算法多数采用递归，基本上每一个节点都会访问并且只访问一次，所以时间复杂度均为$O(n)$

（2）数据结构的分析

二叉树这种结构比较特殊，由于插入、删除等操作容易破坏树的结构，本身应用不常见，但是二叉树是二叉树搜索树、哈夫曼树等数据结构的基础。二叉树这种结构特性可能适用于一些比较特殊的场景，如生物遗传图谱等。

总结一下，二叉树这种数据结构适用场景较少，但不可缺少。在特殊的场景以及一些更特殊的数据结构需要二叉树，其次二叉树的算法时间复杂度不高，经过特殊的改进，可以应用更加广泛。

八、实验总结

这次实验成功的完成了二叉树的创建以及一些操作。首先，必须要说的是，这次实验与往常的有非常大的不同，采用了大量的递归和栈的应用，经过这次实验之后，对这种晦涩的递归，产生了更深的理解。其次是对于二叉树产生了更多

的可能，能不能通过一定的限制条件，来使二叉树具备插入、删除等操作，又或者能让搜索更加的快等等。最后是递归，在前、中、后序的递归和非递归的实现对比中发现，递归的代码往往更加简洁和简单。

总结一下，经过这次实验，第一是收获了对于递归的理解和应用，第二是对于“二叉树是不是具有扩展？”的想象。为后续学习二叉搜索树、红黑树等打下了良好的基础。

参考文献：

[1]李春葆.《数据结构教程（第6版）》.北京：清华大学出版社.
[2]陈越,何钦铭.《数据结构（第2版）》.北京：高等教育出版社.
[3]殷人昆.《数据结构（用面向对象与C++语言描述）（第3版）》.北京：清华大学出版社
[4]严蔚敏.《数据结构（C语言版）》.北京.清华大学出版社.