二叉树

20123211 高雨晴

0. 概述

- 实验内容：定义二叉树，完成二叉树的前序遍历、中序遍历、后序遍历以及层次遍历（广度遍历）算法。
- 实验方法：C++语言，通过MinGW-w64 C/C++编译器。
- 实验结果：测试结果与预期相符。

1. 实验设计

二叉树由根节点、左子树、右子树组成。每一个节点最多连接两个子节点，即左子节点和右子节点。若节点没有子节点，则是叶节点。

1.1 算法实现原理

1. 前序遍历：先输出根元素，再对左子树递归，最后对右子树递归；
2. 中序遍历：顺序改为左子树-根-右子树；
3. 后序遍历：顺序改为左子树-右子树-根；
4. 层次遍历：设置一个队列，从根节点开始遍历，首先将根节点指针入队列，然后从队头取出一个元素，访问该元素指向的节点，若该元素所指节点的左右子节点非空，则将该元素所指节点的左孩子指针和右孩子指针按顺序入队，直到队列为空。

1.2 算法设计与分析

0. 定义节点

```
typedef struct node
{
    struct node *lchild;       // 左子节点
    struct node *rchild;       // 右子节点
    char data;
}BiTreeNode, *BiTree;          // BiTree为指向节点的指针
```

1. 建立树：输入一个满二叉树，按前序顺序建立二叉树，即将输入的字符按根-左子树-右子树顺序放入二叉树。

```
void createBiTree(BiTreeNode* &T)
{//输入一个用#补满的满二叉树
    char c;
    cin >> c;
    if ('#' == c)              //若输入的是#，则令（子）树的根节点为NULL，结束该分支的递归
        T = NULL;
    else
    {
        T = new BiTreeNode;
        T->data = c;                // 先储存进根节点
```

```
        createBiTree(T->lchild);    // 再将数据放入左子树
        createBiTree(T->rchild);    // 最后将数据放入右子树
    }
}
```

2. 前序遍历

```
void Preorder(BiTree T)
{
    if (T)
    {
        cout << T->data << " ";      // 输出根节点
        Preorder(T->lchild);         // 从左子树递归
        Preorder(T->rchild);         // 从右子树递归
    }
}
```

3. 中序遍历

```
void Inorder(BiTree T)
{
    if (T)
    {
        Inorder(T->lchild);
        cout << T->data << " ";
        Inorder(T->rchild);
    }
}
```

4. 后序遍历

```
void postorder(BiTree T)
{
    if (T)
    {
        postorder(T->lchild);
        postorder(T->rchild);
        cout << T->data << " ";
    }
}
```

5. 层次遍历

5.1 定义队列

```
typedef struct Quene {       // 定义顺序队
    int     front;           // 队头指针
    int     rear;            // 队尾指针
    BiTreeNode* data[128];   // 存放队中元素
} SqQueue;
```

5.2 初始化队列，判断是否为空

```
void initQueue(SqQueue** q) {
    if (!((*q) = (SqQueue*)malloc(sizeof(SqQueue)))) {
        printf("error");
        exit(-1);
    }
    (*q)->front = (*q)->rear = -1;
}

bool emptyQueue(SqQueue* q) {
    // 首指针和尾指针相等，说明为空
    if (q->front == q->rear) {
        return true;
    }
    return false;
}
```

5.3 进出队列

```
bool enQueue(SqQueue* q, BiTreeNode* node) {
    // 判断队列是否满了。满（插入失败）-返回假，不满（插入成功）-返回真
    if (q->rear == 128) {
        return false;
    }
    q->rear++;                // 头指针加 1
    q->data[q->rear] = node; // 传值
    return true;
}

bool deQueue(SqQueue* q, BiTreeNode** node) {
    // 判断是否空了。空（取出失败）-返回假，不空（取出成功）-返回真
    if (q->front == q->rear) {
        return false;
    }
    q->front++;                 // 尾指针加 1
    *node = q->data[q->front]; // 取值
    return true;
}
```

5.4 层次遍历

```
void levelorder(BiTreeNode* T) {
    SqQueue* q;        // 定义队列
    initQueue(&q);     // 初始化队列
    if (T != NULL) {  // 根节点指针进队列
        enQueue(q, T);
    }
    // 一层一层的把节点存入队列，当没有孩子节点时就不再循环
    while (!emptyQueue(q)) {       // 队不为空循环
        deQueue(q, &T);            // 出队时的节点
        cout << T->data << " ";    // 输出节点存储的值
        if (T->lchild != NULL) {  // 有左孩子时将该节点进队列
            enQueue(q, T->lchild);
        }
        if (T->rchild != NULL) {  // 有右孩子时将该节点进队列
```

```
            enQueue(q, T->rchild);
        }
    }
}
```

2. 实验结果说明与分析

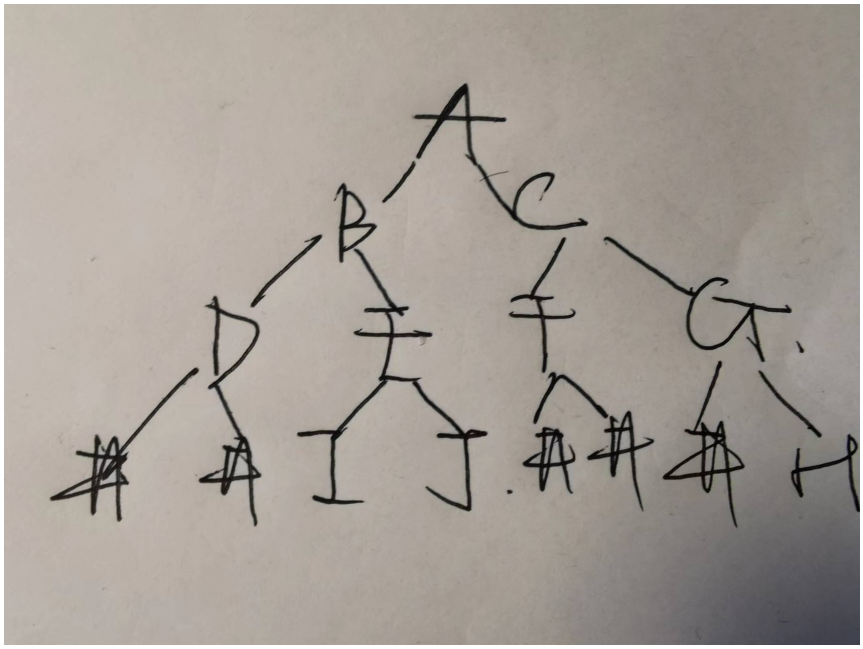
2.1 实验设置

```
#include<iostream>
using namespace std;

//定义节点
typedef struct node
{
    struct node *lchild;
    struct node *rchild;
    char data;
}BiTreeNode, *BiTree;

// 定义队列
typedef struct Quene {
    int     front;
    int     rear;
    BiTreeNode* data[128];
} SqQueue;

// 初始化队列
void initQueue(SqQueue** q) {
    if (!((*q) = (SqQueue*)malloc(sizeof(SqQueue)))) {
        printf("error");
        exit(-1);
    }
    (*q)->front = (*q)->rear = -1;
}

// 判断队列是否为空
bool emptyQueue(SqQueue* q) {
    // 首指针和尾指针相等，说明为空
    if (q->front == q->rear) {
        return true;
    }
    return false;
}

// 进队列
bool enQueue(SqQueue* q, BiTreeNode* node) {
    if (q->rear == 128 - 1) {
        return false;
    }
    q->rear++;
    q->data[q->rear] = node;
```

```
    return true;
}

// 出队列
bool deQueue(SqQueue* q, BiTreeNode** node) {
    if (q->front == q->rear) {
        return false;
    }
    q->front++;
    *node = q->data[q->front];
    return true;
}

// 层次遍历
void levelorder(BiTreeNode* T) {
    SqQueue* q;
    initQueue(&q);
    if (T != NULL) {
        enQueue(q, T);
    }

    while (!emptyQueue(q)) {
        deQueue(q, &T);
        cout << T->data << " ";
        if (T->lchild != NULL) {
            enQueue(q, T->lchild);
        }
        if (T->rchild != NULL) {
            enQueue(q, T->rchild);
        }
    }
}

//按照前序顺序建立二叉树
void createBiTree(BiTreeNode* &T)
{
    char c;
    cin >> c;
    if ('#' == c)
        T = NULL;
    else
    {
        T = new BiTreeNode;
        T->data = c;
        createBiTree(T->lchild);
        createBiTree(T->rchild);
    }
}

//前序遍历输出
void Preorder(BiTree T)
{
    if (T)
    {
        cout << T->data << " ";
```

```
        Preorder(T->lchild);
        Preorder(T->rchild);
    }
}
//中序遍历输出
void Inorder(BiTree T)
{
    if (T)
    {
        Inorder(T->lchild);
        cout << T->data << " ";
        Inorder(T->rchild);
    }
}
//后序遍历输出
void postorder(BiTree T)
{
    if (T)
    {
        postorder(T->lchild);
        postorder(T->rchild);
        cout << T->data << " ";
    }
}
int main()
{
    BiTree T;                //声明一个指向二叉树根节点的指针
    createBiTree(T);
    cout << "Preorder traversal:" << endl;
    Preorder(T);
    cout << endl;
    cout << "Inorder traversal:" << endl;
    Inorder(T);
    cout << endl;
    cout << "Postorder traversal:" << endl;
    postorder(T);
    cout << endl;
    cout << "Level traversal:" << endl;
    levelorder(T);

    system("pause");
    return 0;
}
```

用于测试的树：



2.2 实验结果以及相应的分析

输入：

```
ABD##EI##J##CF##G#H##
```

输出结果：

```
Preorder traversal:
A B D E I J C F G H
Inorder traversal:
D B I E J A F C G H
Postorder traversal:
D I J E B F H G C A
Level traversal:
A B C D E F G I J H
```

与手动排序结果是相符的。

3. 总结

1. 树本身是一种非线性结构，在二叉树的基本操作的算法中多次利用到递归思想，二叉树本身的定义即为递归定义，调用自身定义左右孩子指针；
2. 前中后序遍历算法的递归算法不同之处仅在于访问根结点和遍历左、右子树的先后关系。任何一棵二叉树的叶子结点在先序，中序，后序遍历中，其访问的相对次序不变。这三种遍历算法的时间复杂度均为O(n)，其中n为节点数；
3. 和前三种遍历方式不同的是，层次遍历依靠队列进行节点的储存和打印，而不是利用系统栈，但时间复杂度是一样的，为O(n).