算法设计-树

目录

1. 二叉树的链式存储结构

```
typedef struct BiTNode{
    ElemType data;
    struct BiTNode *lchild, *rchild;
}BiTNode, *BiTree;
```

2. 二叉树的遍历

（1）递归先序遍历：

```
void PreOrder(BiTree T){
    if (T != NULL){
        visit(T);
        PreOrder(T ->lchild);
        PreOrder(T ->rchild);
    }
}
```

（2）非递归先序遍历

```
void preOrder(BiTree b){
    BiTree *stack[20], *p;
    int top;
    if (b != NULL){
        top = 1;
        stack[top] = b; // 根结点入栈
        while (top > 0){
            p = stack[top]; // 出栈访问
            top--;
            visit(p);
            if (p ->rchild != NULL) // 右孩子入栈
                stack[++top] = p ->rchild;
            if (p ->lchild != NULL) // 左孩子入栈
                stack[++top] = p ->lchild;
        }
    }
}
```

（3）非递归中序遍历

```
void InOrder(Bitree T){
    InitStack(S); // 初始化栈
    BiTree p = T; // p 是遍历指针
    while (p || !IsEmpty(S)){
        if (p){ // 根指针入栈，遍历左子树
            Push(S, p);
            p = p ->lchild;
        } else { // 根指针出栈，访问根结点，遍历右子树
            Pop(S, p);
            visit(p);
            p = p ->rchild;
```

```
        }
    }//while
}
```

（4）非递归后续遍历（应用访问很广，极其重要）

```
void PostOrder(BiTree T){
    InitStack(S);
    BiTree p = T, r = NULL;
    while (p || !IsEmpty(S)){
        if (p){ // 走到最左边
            push(S, p);
            p = p ->lchild;
        } else {
            GetTop(S, p);
            // 右子树存在，且未被访问
            if (p ->rchild && p ->rchild != r){
                p = p ->rchild; // 转向右子树
                push(S, p);
                p = p ->lchild; // 再走向最左
            } else {
                pop(S, p);
                visit(p); // 访问该结点
                r = p;    // 记录最近访问过的结点
                p = NULL; // 结点访问完后，重置 p 指针
            }
        }// else
    }// while
}
```

（5）层次遍历(很重要)

```
void LevelOrder(BiTree T){
    InitQueue(Q);  // 初始化队列
    Bitree p;      // p 为遍历结点
    EnQueue(Q, T); // 根结点入队
    while (!IsEmpty(Q)){
        DeQueue(Q, p);
        visit(p);
        if (p ->lchild != NULL) // 左孩子入队
            EnQueue(Q, p ->lchild);
        if (p ->rchild != NULL) // 右孩子入队
            EnQueue(Q, p ->rchild);
    }
}
```

层次遍历的应用：

（1） 求某一层的叶子数 / 分支结点。

（2） 求树的宽度。

（3） 非递归求高度。

（4） 判断是否是完全二叉树。

（5） 删除 以元素值等于 x 的的结点 为根结点的子树。
（6） 也可用于求树的 WPL（也可用先序求）。

3. 求二叉树的深度

（1）递归求深度

```
int Depth(BiTree T){
    if (T == NULL)
        return 0;
    else{
        m = Depth(T ->lchild);
        n = Depth(T ->rchild);
        return m > n ? m+1 : n+1;
    }
}
```

（2）非递归求深度（层次遍历）

```
int Btdepth(BiTree T){
    if (T == NULL)
        return 0;
    int front = -1, rear = -1;
    int last = 0;  // last 指向下一层的第一个结点的位置
    int level = 0; // level 记录当前深度
    BiTree p;
    BiTree Q[MaxSize]; // 设置队列 Q，存放二叉树结点指针
    Q[++rear] = T; // 根结点入队
    while (front < rear){
        p = Q[++front]; // 队列元素出队，即正在访问的结点
        if (p ->lchild)
            Q[++rear] = p ->lchild;
        if (p ->rchild)
            Q[++rear] = p ->rchild;
        if (front == last){ // 处理到该层的最右结点
            level++; // 层数加一
            last = rear; // last 指向下层最左元素的位置
        }
    }// while
    return level;
}
```

4. 求二叉树的宽度

```
int BtWidth(BiTree T, int k){
    BiTree Q[], p = T; // 队列，遍历指针
    int front = 0, rear = 0; // 队列的头、尾
    int last = 1 // last 指向下一层的第一个结点的位置
    int num = 0, maxWidth = 0; // 某层结点数，最大宽度
    Q[++rear] = T; // 根结点入队
    while (front < rear){
        p = Q[++front]; // 队列元素出队，即正在访问的结点
```

```
            num++;
            if (p ->lchild)
                Q[++rear] = p ->lchild;
            if (p ->rchild)
                Q[++rear] = p ->rchild;
            if (front == last){ // 处理到该层的最右结点
                last = rear;    // last 指向下一层
                if (num > maxWidth)
                    maxWidth = num;
                num = 0;
            }
        }// while
        return maxWidth;
    }
```

5. 求两个结点的共同祖先

（1）二叉链表存储

```
    typedef struct{
        BiTree t;
        int tag; // tag = 0 表示左子女已被访问，tag = 1 表示右子女已被访问
    }Stack;

    stack s[], s1[];

    BiTree Ancestor(Bitree Root, BiTree *p, BiTree *q){
        top = 0;
        bt = Root;
        while (bt != NULL || top > 0){
            while (bt != NULL && bt != p && bt != q){ // 结点入栈
                while (bt != NULL){ // 沿左分支向下
                    s[++top].t = bt;
                    s[top] .tag = 0;
                    bt = bt ->lchild;
                }
            }
            // 假定 p 在 q 的左侧，遇到 p 时，栈中元素均为 p 的祖先
            while (top != 0 && s[top].tag == 1){
                if (s[top].t == p){ // 将栈中元素转入辅助数组
                    for (i = 1; i <= top; i++)
                        s1[i] = s[i];
                    top1 = top;
                }
                if (s[top].tag == q) // 找到 q 结点
                    for (i=top; i>0; i--){//将栈中元素的树结点到 s1 中去匹配
                        for (j = top1; j > 0; j--)
                            if (s1[j].t == s[i].t)
```

```
                    return s[i].t; // 返回公共祖先
                }
            top--;
        } // while
        if (top != 0){
            s[top].tag = 1;
            bt = s[top].t ->rchild; // 沿右分支向下遍历
        }
    } // while
    return NULL; // 无公共祖先
}
```

（2）顺序表存储

```
ElemType Comm_Ancestor(SqTree T, int i, int j){
    if (T[i] != '#' && T[j] != '#'){ // 结点存在
        while (i != j){ // 两个编号不同时循环
            if (i > j)
                i /= 2; // 向上找 i 的祖先
            else
                j /= 2; // 向上找 j 的祖先
        }
        return T[i];
    }
}
```

6. 打印值为 x 的结点的所有祖先

采用非递归后序遍历，最后访问根结点，当访问到值为 x 的结点时，栈中所有元素均为该结点的祖先，依次出栈打印即可。

```
typedef struct{
    BiTree t;
    int tag; // tag = 0 表示左子女已被访问，tag = 1 表示右子女已被访问
}stack;

void Search(BiTree bt, ElemType x){
    stack s[];
    top = 0;
    while (bt != NULL || top > 0){
        while (bt != NULL && bt ->data != x){
            s[++top].t = bt;
            s[top].tag = 0;
            bt = bt ->lchild; // 沿左分支向下
        }
        // 找到，输出祖先
        if (bt != NULL && bt ->data == x){
            printf("所有祖先节点为：\n");
            for (i = 0; i <= top; i++)
                printf("%d ", a[i].t ->data);
```

```
                break;
            }
            while (top != 0 && s[top].tag == 1) // 退栈，空遍历(其子树已被访问完)
                top--;
            if (top != 0){
                s[top].tag = 1;
                bt = s[top].t ->rchild; // 沿右分支向下
            }
        }
}
```

7. 输出根结点到每个叶子结点的路径

思路同第 6 题，判断遍历到叶子结点，输出栈内元素。

【注】：输出栈内元素，但不能清空栈。

8. 输出根结点到叶子结点最长一枝路径的所有结点

思路同第 7 题，不同的地方在于，本题遍历到叶子结点时，将栈内元素保存到辅助栈中，并增设一个变量保存路径长度，当遇到更长的路径，则更新辅助栈。最后输出结果。

9. 给二叉树结点增设一个指针域，不适用堆栈，不破坏树的结构，先序遍历二叉树

遍历的同时进行 pre 指针域的更新。

```
void PreOrder(BiTree T){
    // p 表示遍历指针；f 指向双亲
    BiTree p = T, f = NULL;
     while(p){
        while (p){
                visit(p);
                p ->pre = f; // 指向双亲
                f = p;
                p = p ->lchild; // 沿左侧向下
            }
            p = f; // 回退
            while (p && p ->rchild == NULL)
                p = p ->pre; // 回退
            if (p && p ->rchild){
                f = p ->pre; // 避免从右侧返回后，重复访问左侧
                p = p ->rchild;
            }
      }// while
  }
```

10. 二叉树增设双亲域（parent）和标志域（flag，取值为 0-2），不用堆栈，进行后序遍历。

查找结点的后继通过双亲结点，因此设置结点结构为（lchild，data，rchild，parent，flag）。遍历时，当 flag=0 时置 flag 为 1，并遍历左子树；当 flag=1 时置 flag 为 2，并遍历右子树；当 flag=2 时访问结点，同时恢复 flag=0，再去查找其双亲。

```
void PostOrder(BiTree T){
    BiTree p = T;
```

```
        while (p){
            switch (p ->flag){
                case 0: // 遍历左子树
                    p ->flag = 1;
                    if (p ->lchild)
                        p = p ->lchild;
                    break;
                case 1: // 遍历右子树
                    p ->flag = 2;
                    if (p ->rchild)
                        p = p ->rchild;
                    break;
                case 2: // 访问根结点
                    p ->flag = 0;
                    visit(p);
                    p = p ->parent;
            }
        }
}
```

11. 二叉树删除 以元素值等于 x 的结点为根 的子树

删除以元素值 x 为根的子树，只要能删除其左、右子树，就可以释放值为 x 的根结点，因宜采用**后序遍历**删除结点。删除值为 x 的结点，意味着应将其父结点的左（右）子女指针置空，用**层次遍历**易于找到某结点的父结点。本题要求删除树中每一个元素值为 x 的结点的子树。因此要遍历整棵二叉树。

```
// 删除以 bt 为根结点的子树
void DeleteXTree(BiTree bt){
    if (bt){
        DeleteXTree(bt ->lchild);
        DeleteXTree(bt ->rchild);
        free(bt);
    }
}
// 层次遍历
// 在二叉树上查找所有以 x 为元素值的结点，并删除以其为根的子树
void Search(BiTree T, ElemType x){
    BiTree Q[]; // 队列
    if (T){
        if (T ->data == x){ // 根结点符合，删掉
            DeleteXTree(T);
            exit(0); // 若值为 x 的结点不唯一，则删掉此句
        }
        Init Queue(Q);
        EnQueue(Q, T);
        while (!IsEmpty(Q)){
            Dequeue(Q, p);
```

```
                if (p ->lchild){
                    if (p ->lchild ->data == x){ // 左子树符合，删除左子树
                        DeleteXTree(p ->lchild);
                        p ->lchild = NULL; // 记得置空
                    } else {
                        EnQueue(Q, p ->lchild);
                    }
                }
                if (p ->rchild){
                    if (p ->rchild ->data == x){ // 右子树符合，删除右子树
                        DeleteXTree(p ->rchild);
                        p ->rchild = NULL; // 记得置空
                    } else {
                        EnQueue(Q, p ->rchild);
                    }
                }
            }
        }// if
    }
```

12. 求某一层上的叶子树

```
//层次遍历进行求解，思路与求树的宽度相同。
int Btdepth(BiTree T, int k){
    BiTree Q[], p = T; // 队列，遍历指针
    int front = 0, rear = 0; // 队列的头、尾
    int last = 1, level = 1; // last 指向下一层的第一个结点的位置
    int leaf = 0; // 叶子数
    Q[++rear] = T; // 根结点入队
    while (front < rear){
        p = Q[++front]; // 队列元素出队，即正在访问的结点
        if (level == k && !p ->lchild && !p ->rchild)
            leaf++; // 叶子数加一
        if (p ->lchild)
            Q[++rear] = p ->lchild;
        if (p ->rchild)
            Q[++rear] = p ->rchild;
        if (front == last){ // 处理到该层的最右结点
            level++; // 层数加一
            last = rear; // last 指向下层
        }
        if (level > k)
            return leaf;
    }// while
    return 0;
}
```

13. （1）求后序遍历的第一个结点。

思想同下

（2）前序遍历的最后一个结点。

```
// 实际所求为二叉树的最右叶子结点
BiTree getPreLast(BiTree T){
    BiTree p = T;
    while (p){
        if (p ->rchild)
            p = p ->rchild;
        else if (p ->lchild)
            p = p ->lchild;
        else
            return p;
    }
    return NULL; // 空树
}
```

14. 交换左右子树

（1）递归（后序遍历）

```
void exchange(BiTree T){
    if (T){
        exchange(T ->lchild);
        exchange(T ->rchild);
        BiTree p = T ->lchild;
        T ->lchild = T ->rchild;
        T ->rchild = p;
    }
}
```

（2）非递归

```
void exchange(BiTree T){
Bitree s[], p, t = T;
int top = 0;
s[++top] = t;
while (top > 0){
    t = s[top--];
    if (t ->lchild || t ->rchild){ // 交换
        p = t ->lchild;
        t ->lchild = t ->rchild;
        t ->rchild = p;
    }
    if (t ->lchild)
        s[++top] = t ->lchild;
    if (t ->rchild)
        s[++top] = t ->rchild;
}
}
```

15. 判断一个二叉树是否是正则二叉树（结点的度只能为 0 或 2）

只需在遍历的过程中判断有无度为 1 的结点即可。

16. 复制一棵二叉树

（1）递归（先序遍历思想）

```
    Bitree CopyTree(BiTree T){
    Bitree bt;
    if (T == NULL)
        bt == NULL;
    else {
        bt = new(BiTNode);
        bt ->data = T ->data;
        bt ->lchild = CopyTree(T ->lchild);
        bt ->rchild = CopyTree(T ->rchild);
    }
    return bt;
}
```

（2）非递归（层次遍历思想）

```
    void CopyTree(BiTree T){
    BiTree<BiTree, Bitree> Q[];
    EnQueue(Q, (t, bt));
    while (!QueueEmpty(Q)){
        (t, bt) = DeQueue(Q);
        bt = new(BiTNode);
        bt ->data = t ->data;
        if (t ->lchild)
            EnQueue(Q, (t ->lchild, bt ->lchild));
        else
            bt ->lchild = NULL;
        if (t ->rchild)
            EnQueue(Q, (t ->rchild, bt ->rchild));
        else
            bt ->rchild = NULL;
    }
}
```

17. 判断两棵树是否相似

```
// 相似：树的结构相同即可
bool similar(BiTree T1, Bitree T2){
    bool lefts, rights;
    if (T1 == NULL && T2 == NULL) // 两树皆空
        return true;
    else if (T1 == NULL || T2 == NULL) // 只有一树为空
        return false;
    else { // 两树皆不空，递归判断
        lefts = similar(T1 ->lchild, T2 ->lchild);
        rights = similar(T1 ->rchild, T2 ->rchild);
        return lefts && rights;
```

```
    }
}
```

18. 判断两棵树是否等价

在 17 题基础上，加上数据是否相等的判断即可。

19. 求中序线索二叉树的后序遍历的前驱（PS：求中序线索二叉树的前序遍历的后继）

```
BiTree InPostPre(BiTree T, BiTree p){
    BiTree q;
    if (p ->rtag == 0) // 若 p 有右子女，则右子女是其前驱
        q = p ->rchild;
    else if (p ->ltag == 0) // 若 p 只有左子女，则左子女是其前驱
        q = p ->lchild;
    else if (p ->lchild == NULL) // p 是中序的第一个结点，无后序前驱
        q = NULL;
    else { // 顺左线索向上找 p 的祖先，若存在，再找祖先的左子女
        while (p ->ltag == 1 && p ->lchild != NULL)
            p = p ->lchild;
        if (p ->ltag == 0)
            q = p ->lchild; // p 结点的祖先的左子女是其后序前驱
        else
            q = NULL;
    }
    return q;
}
```

20. 中序线索二叉树中找某结点的父亲结点

在中序线索树中找结点的双亲，最简单情况是结点有左右子女，且左右子女都是叶子。结点的左子女的右线索和右子女的左线索都指向双亲。对于有左右子女且左右子女不是叶子的子树中，中序遍历最左下的结点，p 的左线索指向 q：若结点 p 是结点 q 左子树上中序遍历最右下的结点，p 的右线索指向是 q。反过来，通过祖先找子女就容易了。另外，若结点 q 的后继是中序线索树的头结点，则应特殊对待。

21. 后序遍历一个中序线索二叉树

22. 完全二叉树 bt[1…n]，求前序遍历

23. 满二叉树，根据其先序遍历输出后序遍历（王道-4.3.3-15）

24. 先序+中序->后序 | 后序+中序->先序 | 层次+度->构造树

25. 求二叉树的 WPL(带权路径长度之和)（王道-4.3.3-19）

层次遍历 / 先序遍历

26. 孩子兄弟表示法的树，递归求树的深度（王道-4.4.5-6）

27. 孩子兄弟表示法的树，求叶子结点数（王道-4.4.5-5）