树的介绍

> by ReActor

1. 树的概念

1.0 图的概念：

图是点和边共同组成的集合，首先有一些点，然后是一部分边，这些边像道路一样，把点连接在一起。

当然边可能是单向的，也可能是双向的。

$$\text{设有图 } G=(V,E)$$
$$V\text{为点集，}E\text{为边集}$$
$$|V|,|E| \text{ 分别表示点、边的数目}$$

> // Graph,Vertex,Edge

1.1 重要概念

- **路径**：顶点序列，使得其中每个顶点到该序列的下一个顶点有连边
- **简单路径**：一个没有重复顶点的路径称为**简单路径**
- **连通**： 两个顶点之间存在**路径**相连
- **环**：存在结点同时是**路径**的起点与终点

自由树：一个连通的、无环的，无向图。

1.2 自由树性质：

> 令 $G=(V,E)$ 为无向图，下面的描述是等价的。

1. G 是自由树;
2. G 中任意两点由唯一简单路径相连;
3. G 连通，且 $|E|=|V|-1$;
4. G 无环，且 $|E|=|V|-1$;
5. G 无环，但向E 中添加任意一条边成环;

总结:

自由树：一个连通的、无环的，无向图。

- $|E|=|V|-1$
- 任意两点由唯一简单路径相连，因此加边成环

1.3 有根树与有序树

任何树都是在自由树的基础上定义的。

有根树 是一颗自由树，其顶点中存在一个与其他顶点不同的点，称作根$(Root)$

可以任选一个自由树上的结点作为根节点得到一颗有根树，有根树的形状和我们平常见到的树就很相似了。

重要概念:

> 这部分内容和人的宗族关系有很大相近

- 父节点

- 子节点
- 祖先、后代、兄弟；

度：

- **结点的度：**结点拥有的子树(孩子)的数目；
- **树的度：**树中各结点的度最大者；

深度

- **结点的层(深度)：**以根结点为第一层计数；
- **树深度：**最深结点的层数；

叶子

- **叶子结点：**度为零，即没有子树的结点；
- **分支结点：**度不为零的结点，非终端结点；
- **森林：**若干棵互不相交的树构成森林；

有序树： 有序树是一颗有根树，其每个结点的孩子是有顺序的。

二叉树： 不只是有序树。结点的度数不超过二。（位置树）

- 满二叉树：除最后一层无叶子结点外，所有节点度为二

2. 树的存储：

> 树是递归定义的（基于结点），存储和遍历也通常都采用递归的方式。

2.1 邻接表与链式存储

- 邻接表：自由树存储
- 链式存储：二叉树或k叉树的存储

对于一般的自由树，其首先是联通的无向图，实现上一般用图的存储方法，这个我们等到图的部分一并介绍，否则课程容量太大了，大家不太好接受，哈哈。

但在孩子结点数目一定的时候，非叶子结点的结构都一致，可以采用类似链表的存储方式。

2.2 定义结点：

```
typedef struct NODE{
    char data;
    struct NODE *lch,*rch;
}BTNode, BTREE;
```

2.3 树的创建：

取决于数据给出的方式，这里以广义表形式为例给出形式化的说明。

可能有同学没听说过广义表，在二叉树语境下，介绍如下：

> ① 广义表中的一个字母代表一个结点的数据信息。
> ② 每个根结点作为由子树构成的表的名字放在广义表的前面。
> ③ 每个结点的左子树与右子树之间用逗号分开。若结点只有右子树而无左子树，则该逗号不能省略。
> ④ 在整个广义表的末尾加一个特殊符号(如“@”)作为结束标志。
> 例如，对于图 7.17 所示的二叉树，其广义表形式为
> A(B(D,E(G)),C(F(,H)))@

这里统一参考北航出版社的数据结构教程，算法描述如下：

① 若当前取得的元素为字母，则按如下规则建立一个新的(链)结点。

a) 若该结点为二叉树的根结点，则将该结点的地址送 T。

b) 若该结点不是二叉树的根结点，则将该结点作为左孩子(若标志 flag 为 1)或者右孩子(若标志 flag 为 2)链接到其双亲结点上(此时双亲结点的地址在栈顶位置)。

② 若当前取得的元素为左括号“(”，则表明一个子表开始，将标志 flag 置为 1，同时将前面那个结点的地址进栈。

③ 若当前取得的元素为右括号“)”，则表明一个子表结束，做退栈操作。

④ 若当前取得的元素为逗号，则表明以左孩子为根的子树处理完毕，接着应该处理以右孩子为根的子树，将标志 flag 置为 2。

如此处理广义表中的每一个元素，直到取得广义表的结束符号“@”为止。算法如下。

```
typedef struct NODE{
    char data;
    struct NODE *lch,*rch;
}BTNode, BTREE;

BTREE CREATEBT()
{
    BTREE STACK[MAXSIZE], p, T = NULL;//初始化栈、当前指针、根节点;
    char ch;
    int flag, top = -1;
    while (1)
    {
        ch = getchar();
        switch (ch)
        {
        case '@':
            return (T);
        case '(':
            STACK[++top] = p;
            flag = 1;
            break;
        case ')':
            top--;
            break;
        case ',':
            flag = 2;
            break;
        default:
            p = (BTREE)malloc(sizeof(BTNode));
            p->data = ch;
            p->lch = NULL;
            p->rch = NULL;

            if (T == NULL)//将当前结点与栈顶结点链接，特别地处理根节点
                T = p;
            else if (flag == 1)
                STACK[top]->lch = p;
            else
                STACK[top]->rch = p;
        }
    }
}
```

递归建树：

在处理满二叉树的情况下，会使用递归方法建树，极大地简化代码，如初始化线段树的情况，这里先暂缓这部分的介绍。

3. 树的遍历：

3.1 深度优先遍历(DFS)

> Depth-First-Search

通过**递归**，或者是依赖于**栈**的方式来访问树上节点。

这种遍历方式的特征是优先访问深度更高的节点。

自由树的DFS与图的DFS如出一辙，我们暂且按下不表，在图的介绍中一并说。

3.1.1 三种遍历方式

对于二叉树，根据访问当前结点、其左子树、其右子树的顺序，分别由前、中、后序遍历三种方式：

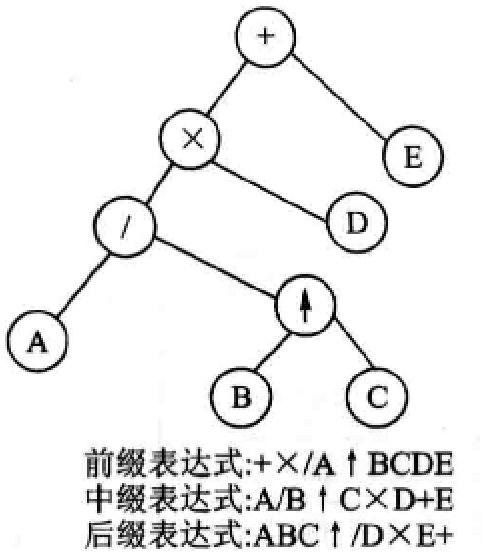
```
void PreOrder(BTREE T){
    if(T!=NULL){
        VISIT(T);
        PreOrder(T->lch);
        PreOrder(T->rch);
    }
}
void InOrder(BTREE T){
    if(T!=NULL){
        InOrder(T->lch);
        VISIT(T);
        InOrder(T->rch);
    }
}
void PosOrder(BTREE T){
    if(T!=NULL){
        PosOrder(T->lch);
        PosOrder(T->rch);
        VISIT(T);
    }
}
```

之所以区分这三者，本质上是关心计算的依赖性。

当一个节点信息的计算需要依赖其子节点的信息时，应当优先完成对子节点的访问，所以选择后序遍历。

相应的，当计算当前节点的信息依赖于父节点的信息的时候，则应当先结束对祖先的访问，选择先序遍历。

3.1.2 表达式树



当把二元运算符的符号作为根节点，参与计算的数值置于子节点时，会得到一棵表达式树。

通过对该表达式树的不同遍历顺序，我们有不同的表达式形式。

同样的，我们也能根据表达式还原出树本身。

3.2 广度优先遍历(BFS)

> Breadth First Search，又作层次优先遍历

有时候我们希望按层次遍历树上结点，这时采用BFS。

BFS中，我们通过**队列**来遍历树上节点。

```
void BFS_TREE(BTREE T)
{
    BTREE QUEUE[MAXSIZE],p;
    int front,rear;
    if(T!=NULL){
        QUEUE[0]=T;//根节点入队初始化
        front=-1;
        rear=0;
        while(front<rear){
            p=QUEUE[++front];
            VISIT(p);
            if(p->lch!=NULL)//将子节点入队
                QUEUE[++rear]=p->lch;
            if(p->rch!=NULL)
                QUEUE[++rear]=r->lch;
        }
    }
}
```

4. 总结