树和二叉树

树的概念

树的概念和术语说明如下：

1. 树(Tree)：是由$n(n \geq 0)$个结点构成的有限集合T。若$n = 0$，则称为空树；否则，一个非空树需要满足以下条件：
   - 有且只有一个特定的称为根(Root)的结点。
   - 除了根结点以外的其他结点被分成$m(m \geq 0)$个互不相交的有限集合$T_1, T_2, \ldots, T_m$，其中每个集合又是一颗树，树$T_1,T_2,\dots,T_m被称为根结点的子树(Subtree)。

   由此可见，树的定义是一个递归定义，即在树的定义中又用到了树的概念。
2. 结点(Node)：它包含数据项以及指向其他结点的分支。
3. 结点的度(Degree)：结点所拥有的子树的树木。
4. 叶结点(Leaf)，也叫终端结点：度为0的结点。叶结点没有后继。
5. 分支结点(Brand)，也叫非终端结点：度不为0的点，即除了叶结点以外的其他结点。
6. 孩子结点(Child)，也叫儿子结点：一个结点的直接后继称为该结点的孩子结点（简称孩子）。
7. 双亲结点(Parent)，也叫父结点：一个结点的直接前驱称为该结点的双亲结点（简称双亲）。
8. 兄弟结点(Sibling)：同一双亲的孩子结点互称为兄弟结点（简称兄弟）。
9. 堂兄弟(Cousin)：双亲互为兄弟的结点互称为堂兄弟结点（简称堂兄弟）。
10. 祖先结点(Ancestor)：从根结点到达一个结点的路径上的所有结点称为该结点的祖先结点（简称祖先）。
11. 子孙结点(Desendant)：以某个节点为根的子树中的任意一个结点都称为该结点的子孙结点（简称子孙）。
12. 结点的层次(Level)：将根结点的层次设为1，其余结点的层次等于它双亲的层次加1。
13. 树的高度(Depth)，也叫树的深度：树中结点的最大层次。
14. 有序树(Ordered Tree)：如果一颗树中各结点的子树从左到右是依次有序的，不能交换，则称为有序树。
15. 无序树(Unordered Tree)：若树中各结点的子树的次序是不重要的，可以交换，则称为无序树。
16. 森林(Forest)：$m(m \geq 0)$棵互不相交的树的集合称为森林。删除一个树的根结点就会得到森林；反之，若给森林增加一个统一的根结点，森林就变成了一颗树。

二叉树的概念和性质

二叉树的概念和抽象数据类型

二叉树是树结构的一种重要类型。在二叉树中，每个结点最多只可以有两个孩子，因此二叉树的存储实现较容易也更具有实际意义。二叉树的概念和术语说明如下：

1. 二叉树(Binary Tree)的递归定义：二叉树是$n(n \geq 0)$个结点的有限集合，该集合或者为空$(n = 0)$，或者由一个根结点以及两个互不相交的左、右子树构成，而其左、右子树又都是二叉树。

   由上述定义可知：
   - 二叉树可以为空，即不含任何结点的空集为空二叉树

- 二叉树的特点是每个结点最多有两个孩子，分别称为该结点的左孩子和右孩子。也就是说，二叉树中所有结点的度都小于等于2.
- 二叉树的子树有左右之分，其子树的次序不能颠倒，即使只有一棵子树，也必须说明是左子树还是右子树。交换一棵二叉树的左、右子树后得到的是另一棵二叉树。
- 度为2的有序树并不是二叉树，因为在有序树中，删除某个度为2的结点的第一子树后，第二子树自然顶替称第一子树。而在二叉树中，若删除某个结点的左子树，则左子树为空，右子树仍然为右子树。

2. 满二叉树(Full Binary Tree)：如果一棵二叉树中任意一层的结点个数都达到了最大值，则此二叉树称为满二叉树。一棵高度为k的满二叉树具有2^k-1个结点。
3. 完全二叉树(Complete Binary Tree)：如果一棵二叉树之有最下面两层结点的度可以小于2，并且最下面一层的结点都集中在该层最左边的连续位置上，则此二叉树称为完全二叉树。对于深度为k，有n个结点的完全二叉树，除了第k层以外，其他各层($1\sim k-1$层)的结点个数都达到最大个数，第k层的所有结点都集中在该层的最左边的连续位置上。

   完全二叉树的特征如下：

   - 叶结点只可能在层次最大的两层出现。
   - 任意一个结点，若其左分支下的子孙的最大层次为l，则其由分支下的最大层次为l或$l-1$，即若某个结点没有左子树，则该结点一定没有右子树。
   - 满二叉树必定为完全二叉树，而完全二叉树不一定为满二叉树。
4. 正则二叉树(Proper Binary Tree)，也称严格二叉树：如果一棵二叉树的任意结点，或者是叶结点，或者恰有两棵非空子树，则这棵二叉树为正则二叉树。即正则二叉树中不存在度为1的结点，除了度为0的叶结点以外，所有分支结点的度都为2。
5. 扩充二叉树(Extended Binary Tree)：在二叉树里出现空子树的位置增加空的叶结点（也称为外部结点），所形成的二叉树称为扩充二叉树。扩充二叉树是严格二叉树。

   构造一棵扩充二叉树的方法如下：

   1. 在原二叉树中度为1的分支结点下面增加一个外部结点。
   2. 在原二叉树的叶结点下面增加两个外部结点。
   3. 原二叉树中度为2的分支结点保持不变。

下面给出二叉树的抽象数据类型定义：

```
template <class elemType> // 二叉树的元素类型
class binaryTree{
public:
    virtual void clear() = 0; // 清空
    virtual bool empty() const = 0; // 判空
    virtual int height() const = 0; // 树的高度
    virtual int size() const = 0; // 树的节点数
    virtual void preOrderTraverse() const = 0; // 前序遍历
    virtual void inOrderTraverse() const = 0; // 中序遍历
    virtual void postOrderTraverse() const = 0; // 后序遍历
    virtual void levelOrderTraverse() const = 0; // 层次遍历
    virtual ~binaryTree() {}; // 虚析构函数
};
```

二叉树的性质

性质一：一个非空二叉树的第i层上最多有$2^{i-1}(i\geq 1)$个结点。

证明：当$i=1$时，二叉树只有一个结点即根结点，$2^{i-1}=2^0=1$，命题成立。假设$i=k$时结论成立，即第k层上最多有2^{k-1}个结点。由归纳假设可知，第$k+1$层上最多有2^k个结点，因为二叉树的每个结点的度最大是2，所以第$k+1$层上的最大结点个数为第k层上的最大结点个数的2倍，即$2\times 2^{k-1}=2^k$。因此，命题成立。

性质二：深度为k的二叉树最多有2^k-1个结点。

证明：深度为k的二叉树，只有每层的结点个数达到最大值时，二叉树的结点总数才会最多。根据性质1，第i层的结点个数最多为2^i，此时二叉树的总结点个数为：$\sum_{i=1}^{k}2^{i-1}=2^k-1$。因此，命题成立。

推论一：深度为k且具有2^k-1个结点的二叉树一定是满二叉树。

性质三：任何一棵二叉树中，若叶结点的个数为n_0，度为2的结点个数为n_2，则$n_0=n_2+1$。

证明：假设二叉树中总结点数为n，度为1的结点个数为n_1，二叉树的总结点数等于度分别为0，1，2的结点个数之和，即$n=n_0+n_1+n_2$。假设二叉树中的边（分支）数为B。除了根结点之外，其余结点都有一条边进入，所以有：$B=n-1$。又由于每个度为2的结点均发出两条边，每个度为1的结点均发出一条边，度为0的结点不发出边，因此有：$B=n_1+2n_2$。由以上三个等式可以推断出$n_0+n_1+n_2=n_1+2n_2+1\Rightarrow n_0=n_2-1$。因此，命题成立。

推论二：扩充二叉树中新增外部结点的个数等于原二叉树的结点个数加1。因为在扩充二叉树中，新增外部结点都是叶结点，而原二叉树中的结点都变成了度为2的结点，根据性质三，该推论成立。

性质四：具有n个结点的完全二叉树的深度为$\left \lceil \log(n+1) \right \rceil $。

证明：假设n个结点的完全二叉树的深度为k，则n的值应该大于深度为$k-1$的满二叉树的结点个数$2^{k-1}-1$，而应该小于深度为k的满二叉树的结点个数2^k-1，即$2^{k-1}-1<n\leq 2^k-1$。将不等式各部分都加1，可得：$2^{k-1}<n+1\leq 2^k$。两边同时取对数，有：$k-1<\log(n+1)\leq k$。由于k是整数，所以$k=\lceil \log(n+1) \rceil$。因此，结论成立。

性质五：如果对一棵有n个结点的完全二叉树按照层次自上而下（每层自左而右）对结点从1到n进行编号，则对任意一个结点$(1\leq i\leq n)$有：

1. 若$i=1$，则结点i为根结点，无双亲；若$i<1$，则结点i的双亲的编号是$\left \lfloor \frac{i}{2} \right \rfloor$。
2. 若$2i\leq n$，则i的左孩子的编号是$2i$，否则i无左孩子。
3. 若$2i+1\leq n$，则i的右孩子的编号是$2i+1$，否则i无右孩子。

二叉树的表示和实现

二叉树的存储结构

顺序存储结构

二叉树的顺序存储结构就是，一组地址连续的存储单元依次自上而下，自左而右地存储二叉树中的结点，并且在存储结点的同时，结点的存储位置（下标）应能体现结点之间的逻辑关系。

对于普通的二叉树，为了能够方便地体现结点之间双亲、孩子、兄弟等逻辑关系，需要将二叉树先扩充一些空结点使之成为完全二叉树，新增的空结点记为$\emptyset$，然后按照完全二叉树的编号将每个结点存储在一维数组的相应分量中。

普通二叉树也可以不经过扩充直接编号，编号方法是：根结点若岑在，则编号为1；编号为i的结点的左孩子若存在，则编号为$2i$；右孩子若存在，则编号为$2i+1$。

这种顺序存储结构比较适用于完全二叉树，因为在最坏情况下，一个深度为k且只有k的结点的右单支二叉树需要2^k-1个存储单元，这显然会造成存储空间的极大浪费。因此，顺序存储结构一般只用于静态的完全二叉树或接近完全二叉树的二叉树。

链式存储结构

采用链式存储结构存储二叉树时，链表结点除了存储元素本身的信息以外，还要设置指示结点间逻辑关系的指针。由于二叉树的每个结点最多有两个孩子，因此可以设置两个指针域left和right，分别指向该结点的左孩子和右孩子。当结点的某个孩子为空时，相应的指针置为空指针。这种结点结构称为二叉链表结点。

若二叉树中经常进行的操作时寻找结点的双亲，每个结点还可以增加一个指向双亲的指针域parent，根结点的parent指针置为空指针。这种结点结构称为三叉链表结点。

利用这两种结点结构所构成的二叉树的存储结构分别称为二叉链表和三叉链表。下面给出二叉树的二叉链表表示和实现：

```
template <class elemType>
class BinaryLinkList:public binaryTree<elemType>{
    struct Node{
        Node *left, *right; // 左右孩子指针
        elemType data; // 节点数据
        Node():left(nullptr), right(nullptr){} // 默认构造函数
        Node(elemType value, Node *l = nullptr, Node *r = nullptr){
            data = value; left = l; right = r;
        } // 带参数的构造函数
        ~Node(){}
    };
    Node* root;
    void clear(Node* t); // 私有，清空
    int size(Node* t) const; // 私有，二叉树的结点总数
    int height(Node* t) const; // 私有，二叉树的高度
    int leafNum(Node* t) const; // 私有，二叉树的叶结点数
    void preOrder(Node* t) const; // 私有，递归前序遍历
    void inOrder(Node* t) const; // 私有，递归中序遍历
    void postOrder(Node* t) const; // 私有，递归后序遍历
    void preOrderCreate(elemType flag, Node* &t); // 私有，创建二叉树
public:
    BinaryLinkList():root(nullptr){} // 构造空二叉树
    ~BinaryLinkList(){clear();} // 析构函数，清空二叉树
    bool empty() const {return root == nullptr;} // 判空
    void clear() {if(root) clear(root); root = nullptr; } // 清空二叉树
    int size() const {return size(root);} // 返回二叉树的结点总数
    int height() const {return height(root);} // 返回二叉树的高度
    void preOrderTraverse() const {if(root) preOrder(root);} // 前序遍历
    void inOrderTraverse() const {if(root) inOrder(root);} // 中序遍历
    void postOrderTraverse() const {if(root) postOrder(root);} // 后序遍历
    void levelOrderTraverse() const; // 层次遍历
    void preOrderWithStack() const; // 非递归前序遍历
    void inOrderWithStack() const; // 非递归中序遍历
    void postOrderWithStack() const; // 非递归后序遍历
    void levelOrderCreate(elemType flag); // 利用带外部结点的层次序列创建二叉树
    void preOrderCreate(elemType flag){
        preOrderCreate(flag, root); // 利用带外部结点的前序序列创建二叉树
    }
};
```

二叉树的遍历运算

深度优先遍历

广度优先遍历

二叉树的其他基本运算

按带外部结点的前序序列建立二叉树

求二叉树的结点总数

求二叉树的高度

求叶结点个数

清空二叉树

树和森林

树的存储结构

双亲表示法

孩子表示法

孩子兄弟表示法

树、森林和二叉树的相互转换

树到二叉树的转换

森林到二叉树的转换

二叉树到森林的转换

树和森林的遍历运算

树的深度优先遍历

森林的深度优先遍历

树的广度优先遍历

森林的广度优先遍历

树和森林的其他基本运算

按带外部结点的前序序列建立树

求树的高度

求结点总数

求叶结点个数

清空树