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$\lceil\log(n+1)\rceil$。

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$\lfloor\frac{i}{2}\rfloor$。
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

遍历二叉树，指按照一定的规则和顺序访问二叉树的所有节点，使得每个结点都被访问一次，而且只被访问一次。由于二叉树是非线性结构，因此，二叉树的遍历实质上是将二叉树的各个结点排列长一个线性序列。遍历的含义包含输出、读取、修改等。

深度优先遍历

深度优先遍历(Depth First Traverse)是指沿着二叉树的深度遍历二叉树的结点，尽可能深地访问二叉树的分支。若分别用L表示遍历左子树、D表示访问根结点、R表示遍历右子树，则有DLR、LDR、LRD、DRL、RDL、RLD共6中遍历方案。如果限定先左后右，则二叉树的遍历方式有三种：DLR、LDR、LRD。这三种方式按照访问根结点次序的不同分别称为：前序遍历、中序遍历、后序遍历。

前序递归遍历

```
template <class elemType>
void BinaryLinkList<elemType>::preOrder(Node* t) const {
    if (t){
        cout << t->data << " "; // 访问根结点
        inOrder(t->left); // 访问左子树
        inOrder(t->right); // 访问右子树
    }
}
```

中序递归遍历

```
template <class elemType>
void BinaryLinkList<elemType>::inOrder(Node* t) const {
    if (t){
        inOrder(t->left); // 访问左子树
        cout << t->data << " "; // 访问根结点
        inOrder(t->right); // 访问右子树
    }
}
```

后序递归遍历

```
template <class elemType>
void BinaryLinkList<elemType>::postOrder(Node* t) const {
    if (t){
        postOrder(t->left); // 访问左子树
        postOrder(t->right); // 访问右子树
        cout << t->data << " "; // 访问根结点
    }
}
```

递归算法的形式简便、可读性好，而且其正确性容易得到证明，但是其消耗的时间与空间多，运行效率低。因此可以仿照递归算法执行过程中递归工作站的工作原理写出其相应的非递归算法。利用一个栈来记下待遍历的结点或子树，以备以后访问，可以将递归的深度优先遍历改为非递归的算法。

非递归前序遍历

算法思想：每到一个结点，先访问该结点，并把该结点压入栈中，然后下降去访问它的左子树。遍历完它的左子树后，从栈顶弹出这个结点，并按照它的`right`域再去遍历该结点的右子树。

```
template <class elemType>
void BinaryLinkList<elemType>::preOrderWithStack() const {
```

```
    stack<Node*> s; // 创建一个栈
    Node* p = root; // 工作指针
    while(!s.empty() || p){ // 栈非空或者p非空
        if(p){
            cout << p->data << " "; // 访问根结点
            s.push(p); // 将根结点入栈
            p = p->left; // 访问左子树
        } else {
            p = s.top(); // 栈顶元素
            s.pop(); // 弹出栈顶元素
            p = p->right; // 访问右子树
        }
    }
}
```

非递归中序遍历

算法思想：每到一个结点就把该结点压入栈中，然后下降去访问它的左子树。遍历完它的左子树后，从栈顶弹出这个结点并访问该结点，并按照它的right域再去遍历该结点的右子树。

```
template <class elemType>
void BinaryLinkList<elemType>::inOrderWithStack() const {
    stack<Node*> s; // 创建一个栈
    Node* p = root; // 工作指针
    while(!s.empty() || p){ // 栈非空或者p非空
        if(p){
            s.push(p); // 将根结点入栈
            p = p->left; // 访问左子树
        } else {
            p = s.top(); // 栈顶元素
            s.pop(); // 弹出栈顶元素
            cout << p->data << " "; // 访问根结点
            p = p->right; // 访问右子树
        }
    }
}
```

非递归后序遍历

算法思想：每到一个结点就把该结点压入栈中，然后下降去访问它的左子树。遍历完它的左子树后，按照它的right域再去遍历该结点的右子树，最后从栈顶弹出该结点并访问它。在后续非递归遍历过程中，需要给栈中的每个元素加上一个特征位，以便区分从栈顶弹出的结点是从栈顶结点的左子树回来的，还是从右子树回来的。

```
template <class elemType>
void BinaryLinkList<elemType>::postOrderWithStack() const{
    enum ChildType{Left, right}; // 定义子树类型
    struct StackElem{
        Node* pointer;
        ChildType flag;
    };
    StackElem elem;
    stack<StackElem> s; // 创建一个栈
    Node* p = root; // 工作指针
    while(!s.empty() || p){ // 栈非空或者p非空
```

```
        while (p != nullptr){
            elem.pointer = p;
            elem.flag = Left; // 标记为左子树
            s.push(elem);
            p = p->left;
        }
        elem = s.top();
        s.pop(); // 弹出栈顶元素
        p = elem.pointer;
        if (elem.flat == Left){ // 从左边回来，已经遍历完左子树
            elem.flag = Right;
            s.push(elem);
            p = p->right;
        }else{
            cout << p->data << " "; // 访问根结点
            p = nullptr;
        }
    }
}
```

广度优先遍历

广度优先遍历(Breadth First Traverse)，又叫宽度优先遍历，或层次遍历，是指沿着二叉树的宽度遍历二叉树的结点，即从上至下，从左至右依次逐层遍历二叉树的结点。

层次遍历的过程是：首先访问根结点，然后从左向右依次访问根结点的非空的左、右孩子，在完成一层结点的访问之后，按照先访问的结点其左、右孩子也要先访问的顺序访问下一层结点，这样一层一层地访问，直至二叉树中所有结点都被访问到。由于队列具有先进先出的特点，因此可以借助队列实现算法。

1. 初始化一个队列，并把根结点入队。
2. 若队列非空，则循环执行步骤3~5，否则遍历结束。
3. 出队一个结点，并访问该结点。
4. 若该结点的左子树非空，则将它的左子树入队。
5. 若该结点的右子树非空，则将它的右子树入队。

```
template <class elemType>
void BinaryLinkList<elemType>::levelOrderTraverse() const {
    queue<Node*> que; // 创建一个队列
    Node* p = root;
    if(p) que.push(p); // 根结点入队
    while (!que.empty()){ // 队列非空
        p = que.front(); // 队首元素
        que.pop(); // 出队
        cout << p->data << " "; // 访问根结点
        if (p->left) que.push(p->left); // 左子树入队
        if (p->right) que.push(p->right); // 右子树入队
    }
}
```

无论是递归遍历算法还是非递归遍历算法，因为要访问每个结点，因此时间复杂度都是$O(n)$。

二叉树遍历的规律

前序序列的第一个结点必然是二叉树的根结点。若根结点的左子树非空，则第二个结点必然是左子树的根，否则第二个结点必然是右子树的根。

根结点将中序序列分割成两个子序列，根结点左边的子序列是根结点的左子树的中序序列，根结点右边的子序列是根结点的右子树的中序序列。

后序序列的最后一个结点必然是二叉树的根结点。若根结点的右子树非空，则倒数第二个结点必然是右子树的根，否则倒数第二个结点必然是左子树的根。

层次序列的第一个结点必然是二叉树的根结点。若根结点的左、右子树都是非空的，则第二个结点必然是左子树的根，第三个结点必然是右子树的根。若根结点的左子树为空，则第二个结点必然是右子树的根。

通过上面的分析，我们可以得出以下结论：

1. 已知二叉树的前序序列和中序序列，可以唯一确定一棵二叉树。
2. 已知二叉树的后序序列和中序序列，可以唯一确定一棵二叉树。
3. 已知二叉树的前序序列和后序序列，不能唯一确定一棵二叉树。
4. 已知二叉树的层次序列和中序序列，可以唯一确定一棵二叉树。

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

算法思想：递归创建二叉树，先创建根结点再创建其左右子树。我们已经知道，对于一般的二叉树，根据某种遍历序列无法确定结点间的关系，也就无法唯一确定一棵二叉树，但是用带外部结点的前序序列可以唯一确定一棵二叉树，外部结点标识了空子树。

```
template <class elemType>
void BinaryLinkList<elemType>::preOrderCreate(elemType flag, Node* &t){
    elemType value;
    cin >> value; // 输入结点值
    if (value == flag) { // 如果是外部结点
        t = nullptr; // 置空
    } else {
        t = new Node(value); // 创建新结点
        preOrderCreate(flag, t->left); // 创建左子树
        preOrderCreate(flag, t->right); // 创建右子树
    }
}
```

求二叉树的结点总数

基于前序递归遍历的思想，求二叉树的结点总数的算法描述如下：

1. 若为空子树，则该子树结点个数为0。
2. 若子树非空，则该子树的结点总数=1(当前结点)+左子树的结点个数+右子树的结点个数。

```
template <class elemType>
int BinaryLinkList<elemType>::size(Node* t) const {
    if (t == nullptr) return 0; // 空树
    return 1 + size(t->left) + size(t->right); // 递归计算
}
```

求二叉树的高度

基于前序递归遍历的思想，求二叉树的高度的算法描述如下：

1. 若为空子树，则该子树高度为0。
2. 若子树非空，则该子树的高度为：左、右子树高度大者+1。

```
template <class elemType>
int BinaryLinkList<elemType>::height(Node* t) const {
    if (t == nullptr) return 0; // 空树
    int lh = height(t->left); // 左子树高度
    int rh = height(t->right); // 右子树高度
    return ((lh > rh) ? lf:rh) + 1; // 返回最大高度
}
```

求叶结点个数

基于前序递归遍历的思想，求二叉树的叶结点个数的算法描述如下：

1. 若为空子树，则该子树叶结点个数为0。
2. 若当前结点没有左、右孩子，那么该结点是叶结点，当前子树的叶结点个数为1。
3. 若当前结点为分支结点，则当前子树的叶结点个数=左子树的叶结点个数+右子树的叶结点个数

```
template <class elemType>
int BinaryLinkList<elemType>::leafNum(Node* t) const {
    if (t == nullptr) return 0; // 空树
    if (t->left == nullptr && t->right == nullptr) return 1; // 叶结点
    return leafNum(t->left) + leafNum(t->right); // 递归计算
}
```

清空二叉树

删除其左、右子树之后再删除根结点自身。

```
template <class elemType>
void BinaryLinkList<elemType>::clear(Node* t){
    if (t->left) clear(t->left); // 清空左子树
    if (t->right) clear(t->right); // 清空右子树
    delete t; // 删除当前结点
}
```

树和森林

下面给出树的抽象数据类型定义：

```
template <class elemType> // 二叉树的元素类型
class Tree{
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 ~Tree() {}; // 虚析构函数
};
```

树的存储结构

树的分支结点可以有很多孩子，实现树的存储结构的关键是如何表示树中结点之间的逻辑关系。树的存储结构有很多种，下面重点介绍最常用的三个：

双亲表示法

基本思想：树具有$1:n$的关系，根结点无双亲，其他任何一个结点的双亲都只有一个，这是由树的定义决定的。双亲表示法正是利用了树的这种性质，用一维数组来存储树的各个结点，通常按层存储，数组中的一个元素对应树中的一个结点，结点的信息包含数据域`data`和结点双亲在数组中的下标`parent`。根结点无双亲，其双亲域用-1表示。因为在存储每个结点的数据信息的同时还存储了该结点的双亲的数组下标，所以这种表示方法对求指定结点的双亲和祖先是非常方便的，可以反复调用求双亲的操作直到根结点。但查找该结点的孩子或兄弟，需要遍历整个数组。

孩子表示法

1. 树的每个结点都可能有多个孩子。我们可以参考二叉链表结构，在每个结点中设置若干指针指向该结点的孩子，每个结点的指针域的个数等于树的度d，即每个结点包含一个数据域和d个指针域。这种链表中的结点是同构的，称为多重链表。其优点是结点结构定长，易于管理。但是由于树中有很多结点的度小于d，许多指针域是空的，因此其缺点是造成存储空间的浪费。在这种结构中，具有n个结点的树总共有$n \times d$个指针，因为树只有$n-1$个分支，因此只有$n-1$个指针有用，浪费了$n(d-1)+1$个指针域。
2. 按照每个结点的度分配指针域的个数，并且在结点中设置`degree`域，保存该结点的度。这种链表中的结点是非同构的，即各个结点的结构不等长，这种存储结构的优点是可以节约空间，缺点是难以实现，运算也不方便。
3. 孩子链表表示法：将每个结点的孩子构成一个单链表，称为孩子链表。叶结点的孩子链表为空。链表中增加一个头结点，为了便于管理，将各个头结点放在一个一维数组中，构成孩子链表的表头数组。

   表头数组中每个元素（结点）包含两个域：数据域`data`用于存放该结点的数据信息，指针域`first`用于存放该结点的第一个孩子的地址。

   孩子链表结点也有两个域，数据域`child`用于存放该结点在顺序表的下标，指针域`next`用于存放其双亲的下一个孩子的地址。用孩子链表表示法查找某个结点的孩子很容易，但是查找结点的双亲比较困难。
4. 双亲孩子表示法：为了查找双亲和查找孩子一样方便，我们把双亲表示法和孩子表示法结合起来，在表头结点中增加指示双亲的`parent`域，形成双亲孩子表示法。

孩子兄弟表示法

孩子兄弟表示法也称二叉树表示法，即用二叉链表作为树的存储结构。其原理是，结点的第一个孩子若存在，则它是唯一的，结点的右兄弟若存在，则它也是唯一的。因此，链表中结点的两个指针域firstChild和nextSibling，分别指向该结点的第一个孩子和下一个兄弟。用这种存储结构很容易实现树的某些操作。下面仅介绍孩子兄弟表示法的实现，给出树（森林相同）的类型定义。

```
template <class elemType>
class childSiblingTree : public Tree<elemType> {
    struct Node{
        elemType data; // 节点数据
        Node* firstChild; // 指向第一个子节点
        Node* nextSibling; // 指向下一个兄弟节点
        Node(const elemType& value) : data(value), firstChild(nullptr), nextSibling(nullptr) {}
        Node() : firstChild(nullptr), nextSibling(nullptr) {}
        ~Node(){}
    };
    Node* root; // 根节点
    void clear(Node* t); // 清空子树
    int size(Node* t) const; // 计算子树的节点数
    int height(Node* t) const; // 计算子树的高度
    int leafNum(Node* t) const; // 计算子树的叶子节点数
    void preOrder_1(Node* t) const; // 前序遍历（递归）
    void preOrder_2(Node* t) const; // 前序遍历（非递归）
    void postOrder_1(Node* t) const; // 后序遍历（递归）
    void postOrder_2(Node* t) const; // 后序遍历（非递归）
    void preOrderCreate(elemType flag, Node* &t); // 前序创建树
public:
    childSiblingTree(): root(nullptr) {} // 默认构造函数
    ~childSiblingTree() { clear(); } // 析构函数
    void clear(){if (root) clear(root); root = nullptr; } // 清空树
    bool empty() const { return root == nullptr; } // 判空
    int height() const { return height(root); } // 树的高度
    int size() const { return size(root); } // 树的节点数
    void preOrderTraverse() const {if(root) preOrder_1(root); } // 前序遍历（递归）
    void postOrderTraverse() const {if(root) postOrder_1(root); } // 后序遍历（递归）
    void levelOrdertraverse() const; // 层次遍历
    void preOrderCreate(elemType flag){ // 前序创建树
        preOrderCreate(flag, root);
    }
};
```

树、森林和二叉树的相互转换

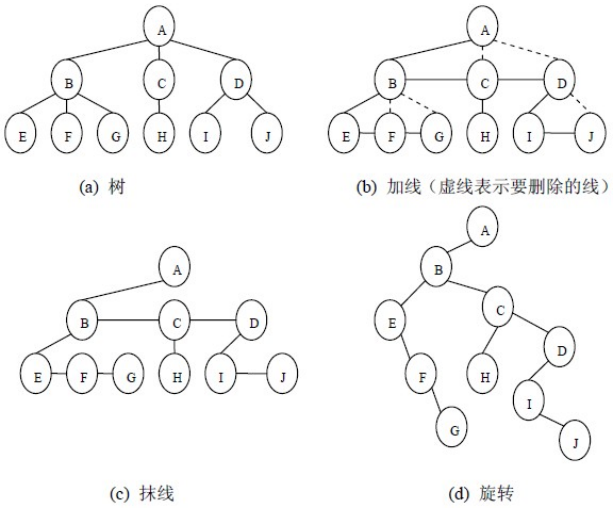
二叉树是一种结构相对简单，运算容易实现的树结构。但是对于很多实际问题，其自然的描述形态是树或者森林，树的孩子兄弟表示法就是将一棵树表示称二叉树的形态，这样可以将二叉树中的许多方法用在树的处理中。因此任何树都可以采用二叉链表作为存储结构，树可以转换为二叉树。森林是树的有限集合，森林也可以转换成二叉树。

树到二叉树的转换

树到二叉树的转换可以分为以下三步：

1. 连线：在所有互为相邻兄弟的结点之间加一条连线。
2. 删线：对于每个结点，除保留与其最左孩子的连线之外，删除该结点与其他孩子之间的连线。

3. 旋转：将按以上方法形成的二叉树，沿着顺时针方向旋转45^o，就可以得到一棵结构清晰的二叉树



森林到二叉树的转换

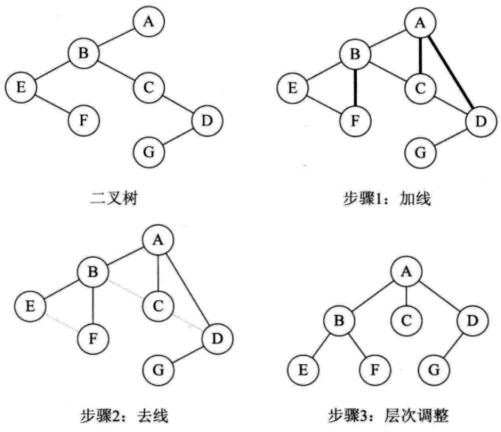
我们认为森林中所有的树具有兄弟关系，森林到二叉树的转换和树到二叉树的转换一样：

1. 连线：在所有互为相邻兄弟的结点之间加一条连线，包括森林中所有树的根结点。
2. 删线：对于每个结点，除保留与其最左孩子的连线之外，删除该结点与其他孩子之间的连线。
3. 旋转：将按以上方法形成的二叉树，沿着顺时针方向旋转45^o，就可以得到一棵结构清晰的二叉树

二叉树到森林的转换

二叉树到森林（树）的转换即为森林（树）到二叉树的转换的逆过程，分为以下三步：

1. 连线：如果某个节点的左孩子有右子树，则在该结点与其左孩子的右子树的右链上各个结点间增加连线。
2. 删线：删去二叉树中所有的双亲与右孩子的连线。
3. 将按以上方法形成的森林（树），沿着逆时针方向旋转45^o，就可以得到一棵结构清晰的森林（树）。



树和森林的遍历运算

树的深度优先遍历

1. 前序：若树非空，访问根结点——前序遍历根结点的各个子树
2. 后序：若树非空，后序遍历根结点的各个子树——访问根结点

树的遍历也可以借助二叉树的遍历运算来实现。

森林的深度优先遍历

1. 前序：若森林非空，访问第一颗树的根结点——前序遍历根结点的子树森林——前序遍历除第一颗树之外的树构成的森林
2. 后序：若森林非空，后序遍历第一棵树的子树森林——访问第一颗树的根结点——后序遍历除第一颗树之外剩余的树构成的森林

前序遍历树、森林：

```
template <class elemType>
void childSiblingTree<elemType>::preOrder_1(Node* t) const {
    if (t != nullptr) {
        cout << t->data << " "; // 访问当前节点
        preOrder_1(t->firstChild); // 递归访问第一个子节点
        t = t->nextSibling; // 移动到下一个兄弟节点
    }
}
```

后序遍历树、森林：

```
template <class elemType>
void childSiblingTree<elemType>::postOrder_1(Node* t) const {
    if (t != nullptr) {
        postOrder_1(t->firstChild); // 递归访问第一个子节点
        cout << t->data << " "; // 访问当前节点
        t = t->nextSibling; // 移动到下一个兄弟节点
    }
}
```

利用二叉树的前序遍历实现树、森林的前序遍历：

```
template <class elemType>
void childSiblingTree<elemType>::preOrder_2(Node* t) const {
    if (t != nullptr){
        cout << t->data << " "; // 访问当前节点
        preOrder_2(t->firstChild); // 递归访问第一个子节点
        preOrder_2(t->nextSibling); // 递归访问下一个兄弟节点
    }
}
```

利用二叉树的后序遍历实现树、森林的后序遍历：

```
template <class elemType>
void childSiblingTree<elemType>::postOrder_2(Node* t) const {
    if (t != nullptr){
        postOrder_2(t->firstChild); // 递归访问第一个子节点
        cout << t->data << " "; // 访问当前节点
        postOrder_2(t->nextSibling); // 递归访问下一个兄弟节点
    }
}
```

树的广度优先遍历

广度优先遍历即层次遍历，其过程是：首先访问根结点，然后从左向右依次访问根结点的非空的孩子，在完成一层结点的访问之后，按照“先被访问的结点，其孩子也要被先访问”的顺序访问下一层结点，这样一层一层访问，直至树中所有结点都被访问到。广度优先遍历的基本原则是，按层次顺序，自顶而下，同一层自左而右。

森林的广度优先遍历

层次遍历森林：若森林非空，层次遍历森林中的第一棵树——层次遍历森林中剩余的树构成的森林。

层次遍历树、森林：

```
template <class elemType>
void childSiblingTree<elemType>::levelOrderTraverse() const {
    queue<Node*> Q; // STL队列
    Node* p = root; // 工作指针
    while (p != nullptr){
        Q.push(p); // 将当前节点入队
        p = p->nextSibling; // 指向下一个兄弟节点
    }
    while (!Q.empty()){
        p = Q.front(); // 获取队首元素
        Q.pop(); // 出队
        cout << p->data << " "; // 访问当前节点
        p = p->firstChild; // 指向第一个子节点
        while (p != nullptr) {
            Q.push(p); // 将子节点入队
            p = p->nextSibling; // 指向下一个兄弟节点
        }
    }
}
```

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

树的带外部结点的前序序列是可以唯一确定树所对应的二叉树的，而且树的前序序列与其对应的二叉树的前序序列相同，因此利用前序序列创建二叉树算法创建的二叉链表也就是树的兄弟链表。

```
template <class elemType>
void childSiblingTree<elemType>::preOrderCreate(elemType flag, Node* &t) {
    elemType value;
    cin >> value; // 输入节点值
    if(value != flag){
        t = new Node(value); // 创建新节点
        preOrderCreate(flag, t->firstChild); // 递归创建第一个子节点
        preOrderCreate(flag, t->nextSibling); // 递归创建下一个兄弟节点
    } else {
        t = nullptr; // 如果输入值为标志值，则设置为nullptr
    }
}
```

求树的高度

1. 若为空子树，则该子树的高度为0

2. 若子树非空，则比较1（当前结点）加其左孩子树的高度与其右兄弟树的高度，选取大者为该子树的高度。

```
template <class elemType>
int childSiblingTree<elemType>::height(Node* t) const {
    if (t == nullptr) return 0; // 空节点高度为0
    else{
        int lh = height(t->firstChilde), rh = height(t->nextSibling) // 计算子节点和兄弟节点的高度
        return (1+lh > rh ? 1+lh : rh); // 返回较大高度
    }
}
```

求结点总数

1. 若为空子树，则该子树的结点个数为0
2. 若子树非空，则该子树的结点树为1+左孩子树的结点个数+右兄弟树的结点个数

```
template <class elemType>
int childSiblingTree<elemType>::size(Node* t) const {
    if (t == nullptr) return 0; // 空节点返回0
    return 1 + size(t->firstChild) + size(t->nextSibling); // 当前节点+子节点+兄弟节点
}
```

求叶结点个数

1. 若为空子树，则该子树的叶结点个数为0
2. 若当前结点没有第一个孩子，那么它也不会有其他孩子，该结点是叶结点，因此，当前子树的叶结点个数=1+兄弟树的叶结点个数
3. 若当前结点右孩子，即分支结点，那么当前子树的叶结点个数=孩子树的叶结点个数+兄弟树的叶结点个数

```
template <class elemType>
int childSiblingTree<elemType>::leafNum(Node* t) const {
    if (t == nullptr) return 0; // 空节点返回0
    if (t->firstChild == nullptr) return 1 + leafNum(t->nextSibling); // 如果没有子节点，则是叶子节点
    return leafNum(t->firstChild) + leafNum(t->nextSibling); // 递归计算子节点和兄弟节点的叶子节点数
}
```

清空树

删除其左孩子树和右兄弟树之后再删除根结点自身

```
template <class elemType>
void childSiblingTree<elemType>::clear(Node* t) {
    if (t->firstChild) clear(t->firstChild); // 清空第一个子节点
    if (t->nextSibling) clear(t->nextSibling); // 清空下一个兄弟节点
    delete t; // 删除当前节点
}
```