

# 第11章

## 二叉树和其他树

### 简要回顾

- 线性数据结构：一对一
  - 线性表：数组描述、链式描述（第五、六章）
  - 数组和矩阵（第七章）
  - 栈：先进后出（第八章）
  - 队列：先进先出（第九章）
- 一对多的数据结构？



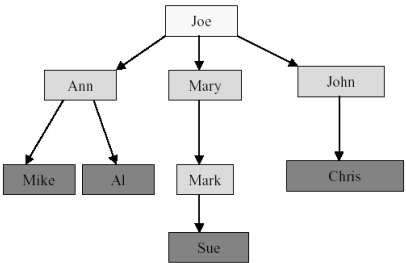
### 本章内容

- 11.1 树
- 11.2 二叉树
- 11.3 二叉树的特性
- 11.4 二叉树的描述
- 11.5 二叉树常用操作
- 11.6 二叉树遍历
- 11.7 抽象数据类型BinaryTree
- 11.8 类linkedBinaryTree
- 11.9 应用

### 11.1 树

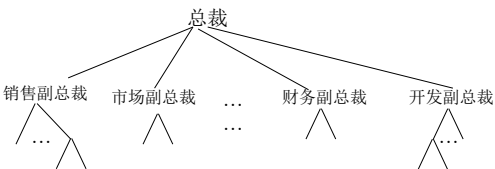
- 线性数据结构和表数据结构一般不适合于描述具有层次结构的数据。
- 例：层次结构的数据

### 例 11.1 Joe的后代



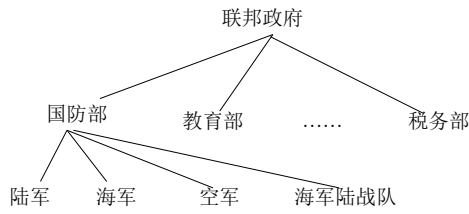
- 元素之间的关系是父母——子女

### 例 11.2 公司组织机构



- 元素之间的关系是上级——下级

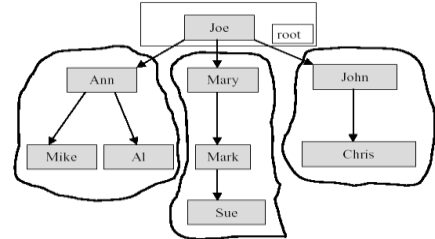
## 例 11.3 政府机构



- 元素之间的关系是整体——部分

## 树的定义

- 定义[树]：
  - 树(tree)是一个**非空**的有限元素的集合。
  - 其中一个元素为根(root)。
  - 其余的元素(如果有的话)组成t的子树(subtrees)。

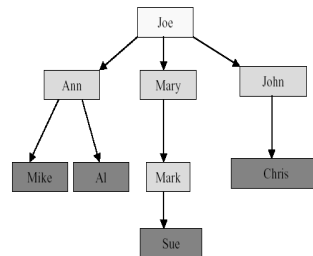


## 术语

- 层次中最高层的元素为**根 (root)**。
- 其余的元素分成不相交的集合。根的下一级的元素是**根的孩子 (children)**。是余下元素所构成的子树的根。
- 树中没有孩子的元素称为**叶子(leaves)**

## 术语

- 父母(Parent), 孙子(grandchildren), 祖父(Grandparent),
- 兄弟(Sibling), 祖先(Ancestors), 后代(Descendent)



叶子 = {Mike, Al, Sue, Chris}

父母(Mary) = Joe

祖父(Sue) = Mary

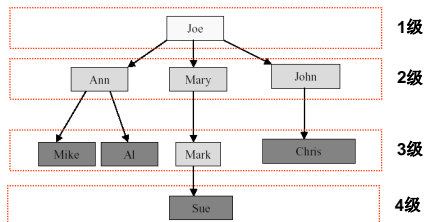
兄弟(Mary) = {Ann, John}

祖先(Mike) = {Ann, Joe}

后代(Mary) = {Mark, Sue}

## 术语

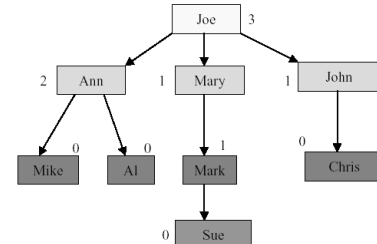
- 级(level)/层次: 指定树根的级为1, 其孩子(如果有的)的级为2。一个元素的级=其父母的级+1。



- 一棵树的高度或深度是树中级数的个数。图中树高为4。

## 术语

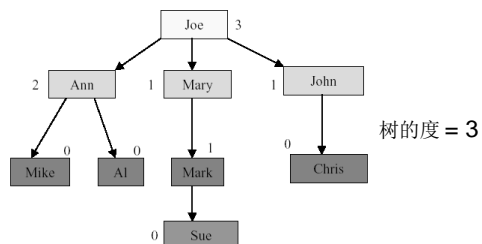
- 元素的度(Degree of an element)
  - 是指其孩子的个数。



- 叶节点的度为0

## 术语

- 树的度(The degree of a tree)
  - 是其元素度的最大值。



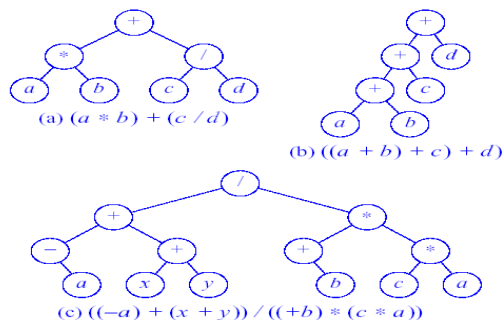
## 11.2 二叉树

- 定义[二叉树]:
- 二叉树(binary tree)  $t$  是有限个元素的集合(可以为空)。
- 当二叉树非空时, 其中有一个称为**根(root)**的元素, 余下的元素(如果有的话)被组成2个二叉树, 分别称为 $t$ 的左子树和右子树。

## 二叉树和树的区别

- 二叉树可以为空, 但树不能为空。
- 二叉树中每个元素都恰好有**两棵子树**(其中一个或两个可能为空)。而树中每个元素可有**任意多个子树**。
- 在二叉树中每个元素的子树都是**有序的**, 也就是说, 可以用左、右子树来区别。而树的子树间是**无序的**。

## 二叉树表示数学表达式—表达式树

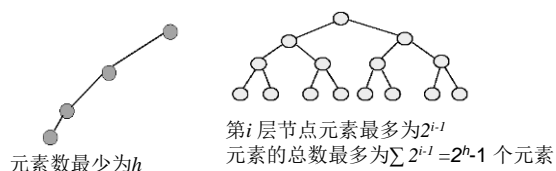


## 11.3 二叉树的特性

- ✓ 特性 1:
  - 包含 $n$  ( $n > 0$ ) 个元素的二叉树边数为 $n-1$ 。
  - 证明:
    - 二叉树中每个元素(除了根节点) 有且只有一个父节点
    - 在子节点与父节点间有且只有一条边
    - 因此, 边数为 $n-1$ 。

## 11.3 二叉树的特性

- ✓ 特性 2:
  - 若二叉树的高度为 $h$ ,  $h \geq 0$ , 则该二叉树最少有 $h$ 个元素, 最多有 $2^h - 1$  个元素。
  - 证明:

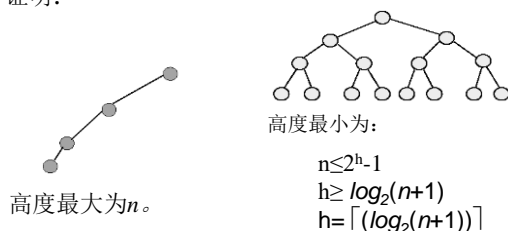


## 11.3 二叉树的特性

✓ 特性 3:

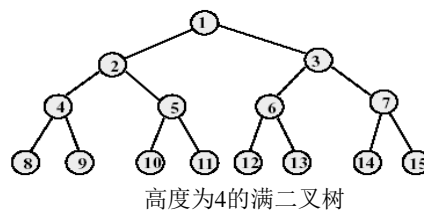
- 包含  $n(n \geq 0)$  个元素的二叉树的高度最大为  $n$ ，最小为  $\lceil \log_2(n+1) \rceil$ 。

■ 证明:



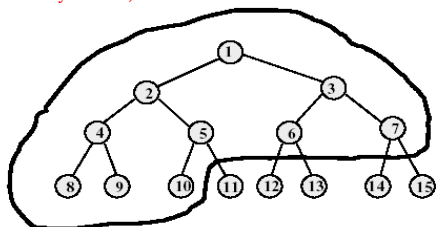
## 满二叉树

- 当高度为  $h$  的二叉树恰好有  $2^h - 1$  个元素时，称其为 **满二叉树(full binary tree)**。
- 对高度为  $h$  的满二叉树中的元素按从上到下，从左到右的顺序从 1 到  $2^h - 1$  进行编号。



## 完全二叉树

- 从满二叉树中删除  $k$  个元素，其编号为  $2^{h-1} - i$ ,  $1 \leq i \leq k$ ，所得到的二叉树被称为 **完全二叉树(complete binary tree)**。

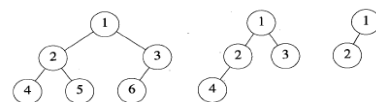


## 完全二叉树

- 深度为  $k$  具有  $n$  个节点的二叉树是一颗完全二叉树，当且仅当它与  $k$  层满二叉树前  $1 \sim n$  个节点所构成的二叉树结构相同。
- $k$  层完全二叉树:
  - 1) 前  $k-1$  层为满二叉树;
  - 2) 第  $k$  层上的节点都连续排列于第  $k$  层的左端。

## 完全二叉树

- 满二叉树是完全二叉树的一个特例。
- 有  $n$  个元素的完全二叉树的深度为  $\lceil \log_2(n+1) \rceil$



✓ 特性 4:

- 设完全二叉树中一元素的序号为  $i$ ,  $1 \leq i \leq n$ 。则有如下关系成立:
  - 当  $i = 1$  时，该元素为二叉树的根。若  $i > 1$ ，则该元素父节点的编号为  $\lfloor i/2 \rfloor$ 。
  - 当  $2i > n$  时，该元素无左孩子。否则，其左孩子的编号为  $2i$ 。
  - 若  $2i+1 > n$ ，该元素无右孩子。否则，其右孩子编号为  $2i+1$ 。

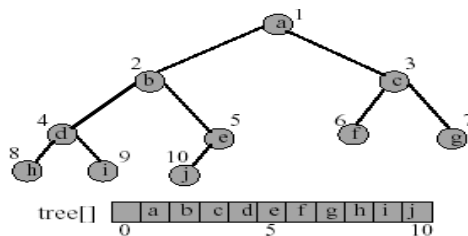
## 11.4 二叉树描述

### ➤数组描述

### ➤链表描述

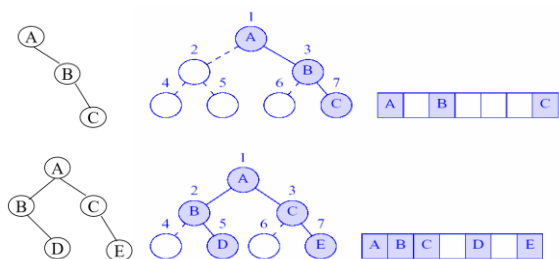
## 11.4.1 数组描述

- 完全二叉树：
- 按照二叉树对元素的编号方法，将二叉树的元素存储在数组中。



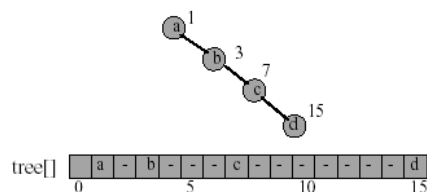
## 11.4.1 数组描述

- 二叉树可以看作是缺少了部分元素的完全二叉树。



## 11.4.1 数组描述

- 一个有 $n$ 个元素的二叉树需要存储空间： $n+1$  到  $2^n$  (或： $n$  到  $2^n - 1$ ).
- 右斜(Right-skewed)二叉树存储空间达到最大。



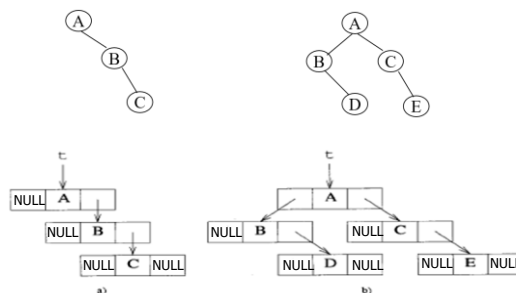
- 当缺少的元素数目比较少时，数组描述方法是有效的。

## 11.4.2 链表描述

- 二叉树最常用的描述方法。
- 每个元素都存储在一个节点内，每个节点：

leftChild | element | rightChild

## 11.4.2 链表描述



## 链表二叉树的节点结构

```
template<class T>
struct binaryTreeNode
{
    T element;
    binaryTreeNode<T> *leftChild; //指向左孩子节点的指针
    *rightChild; //指向右孩子节点的指针
    // 3 个构造函数
    binaryTreeNode() // 没有参数
    {
        leftChild = rightChild = NULL;
    }
    binaryTreeNode(const T& theElement) // 只有数据参数
    {
        element(theElement); leftChild = rightChild = NULL;
    }
    binaryTreeNode(const T& theElement, // 数据 + 指针参数
        binaryTreeNode *theLeftChild,
        binaryTreeNode *theRightChild)
    {
        element(theElement);
        leftChild = theLeftChild;
        rightChild = theRightChild;
    }
}
```

## 11.5 二叉树常用操作

- 确定其高度
- 确定其元素数目
- 复制
- 在屏幕或纸上显示二叉树
- 确定两棵二叉树是否一样
- 删除整棵树
- 若为数学表达式树，计算该数学表达式
- 若为数学表达式树，给出对应的带括号的表达式

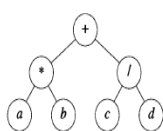
## 11.6 二叉树遍历

- 许多二叉树操作是通过对二叉树进行遍历来完成。
- 在二叉树的遍历中，**每个元素都被访问到且仅被访问一次**。
- 在访问时执行对该元素的相应操作(复制、删除、输出等)。

## 二叉树遍历方法

- 二叉树遍历方法：
  - **前序遍历**
    - **访问根元素**；前序遍历左子树；前序遍历右子树。
    - (根、左、右)
  - **中序遍历**
    - 中序遍历左子树；**访问根元素**；中序遍历右子树。
    - (左、根、右)
  - **后序遍历**
    - 后序遍历左子树；后序遍历右子树；**访问根元素**。
    - (左、右、根)
  - 层次遍历
- 注：“访问”可以代表对节点操作的函数操作，如输出、复制、删除

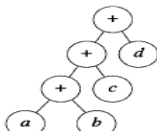
## 遍历示例 (visit(t)= cout(t->element))



前序: +\*ab/cd

中序: a\*b+c/d

后序: ab\*cd/+



前序: +++abcd

中序: a+b+c+d

后序: ab+c+d+

## 前序遍历

```
template<class E>
void preOrder (binaryTreeNode<E> *t)
{ //前序遍历二叉树*t
    if (t != NULL)
    {
        visit(t); //访问根节点
        preOrder(t->leftChild); //前序遍历左子树
        preOrder(t->rightChild); //前序遍历右子树
    }
}
```

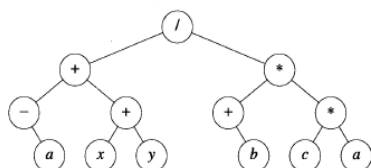
## 中序遍历

```
template<class E>
void inOrder(binaryTreeNode<E> *t)
{ //中序遍历二叉树*t
  if (t != NULL)
  { inOrder(t->leftChild);    //中序遍历左子树
    visit(t);                //访问根节点
    inOrder(t->rightChild);   //中序遍历右子树
  }
}
```

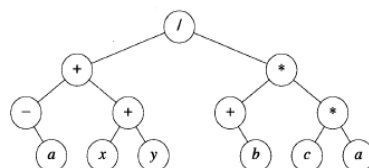
## 后序遍历

```
template<class E>
void postOrder(binaryTreeNode<E> *t)
{ //后序遍历二叉树*t
  if (t != NULL)
  { postOrder(t->leftChild);  //后序遍历左子树
    postOrder(t->rightChild); //后序遍历右子树
    visit(t);                //访问根节点
  }
}
```

## 表达式二叉树的遍历



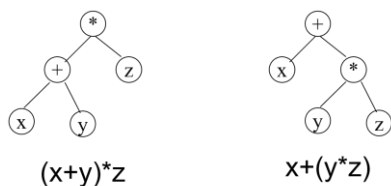
## 表达式二叉树的遍历



前序: /+-a+xy\*+b\*ca      --表达式的前缀形式  
中序: -a+x+y/+b\*c\*d      --表达式的中缀形式  
后序: a-xy++b+ca\*\*/      --表达式的后缀形式

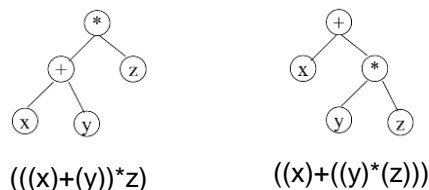
## 表达式二叉树的遍历

- 表达式的中缀形式存在歧义:
  - $x+y*z$  有下面两种解释, 对应两棵不同的二叉树



## 数学表达式二叉树的遍历

- 完全括号化的中缀表达式:
  - 每个操作符和相应的操作数都用一对括号括起来。更甚者把操作符的每个操作数也都用一对括号括起来。



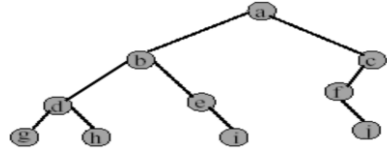
## 输出完全括号化的中缀表达式

```
template <class E>
void infix(binaryTreeNode<E> *t)
{// 输出表达式的中缀形式
    if (t != NULL)
    {cout << '(';
        infix(t->leftChild); // 左操作数
        cout << t->element; // 操作符
        infix(t->rightChild); // 右操作数
        cout << ')';}
}
```

- 前缀和后缀表达式中不会存在歧义。

## 层次遍历

- 在层次遍历过程中，按从顶层到底层的次序访问树中元素，在同一层中，从左到右进行访问。
- 逐层示例 (visit(t)=cout(t->element))



■ 输出: a b c d e f g h i j

## 层次遍历

```
template <class E>
void levelOrder(binaryTreeNode<E> *t)
{//层次遍历二叉树*t
    arrayQueue<binaryTreeNode<E>*> q;
    while (t != NULL)
    { visit(t); //访问t
        //将t的左右孩子放入队列
        if (t->leftChild) q.push(t->leftChild);
        if (t->rightChild) q.push(t->rightChild);
        try {t=q.front();} //访问下一个节点
        catch (queueEmpty) {return;}
        q.pop();
    }
}
```

## 遍历算法性能

- 设二叉树中元素数目为 $n$ 。四种遍历算法：
  - 空间复杂性均为 $O(n)$ 。
  - 时间复杂性均为 $\Theta(n)$ 。

## 11.7 抽象数据类型binaryTree

抽象数据类型 binaryTree

{

实例:

元素集合: 如果不空, 则被划分为根、左子树和右子树;  
每个子树仍是一个二叉树;

操作:

empty: 如果二叉树为空, 则返回true, 否则返回false  
Size(): 返回二叉树的节点/元素个数  
preorder(visit): 前序遍历二叉树, visit是访问函数  
inOrder(visit): 中序遍历二叉树  
postOrder(visit): 后序遍历二叉树  
LevelOrder(visit): 层次遍历二叉树

}

## 二叉树抽象类 binaryTree

```
template<class T> //T:节点类型
class binaryTree {
{
    public:
        virtual ~ binaryTree() {}
        virtual bool empty() const = 0;
        //二叉树为空时返回true, 否则返回false
        virtual int size() const = 0;
        //返回二叉树中元素的个数
        virtual void preOrder( void (*) (T *) ) = 0; //前序遍历二叉树
        virtual void inOrder( void (*) (T *) ) = 0; //中序遍历二叉树
        virtual void postOrder( void (*) (T *) ) = 0; //后序遍历二叉树
        virtual void levelOrder( void (*) (T *) ) = 0; //层序遍历二叉树
}
```



## 11.8 类 linkedBinaryTree(1/3)

```
template<class E>
class linkedBinaryTree: public binaryTree<binaryTreeNode<E>>
{public:
    linkedBinaryTree() {root = NULL; treeSize = 0;};
    ~BinaryTree() {erase();};
    bool empty() const {return treeSize==0;};
    void preOrder( void(*theVisit) (binaryTreeNode<E> *) )
        {visit= theVisit; preOrder(root);};
    void inOrder( void(*theVisit) (binaryTreeNode<E> *) )
        {visit= theVisit; inOrder(root);};
    void postOrder( void(*theVisit) (binaryTreeNode<E> *) )
        {visit= theVisit; postOrder(root);};
    void levelOrder(void(*) (binaryTreeNode<E> *));};
```

## 类 linkedBinaryTree(2/3)

```
void erase();//在后序遍历中删除二叉树所有节点
{postOrder(dispose);
 root=NULL;
 treeSize=0;
}
private:
    binaryTreeNode<E>*root;//根节点指针
    int treeSize;//数的元素个数
    static void (*visit) (binaryTreeNode<E> *) ;//访问函数
    static void preOrder(binaryTreeNode<E> *t);
    static void inOrder(binaryTreeNode<E> *t);
    static void postOrder(binaryTreeNode<E> *t);
    static void dispose(binaryTreeNode<E> *t)
        {delete t};//删除t指向的节点
}
```

## 私有preOrder

```
template<class E>
void linkedBinaryTree<E>::preOrder (binaryTreeNode<E> *t)
{//前序遍历二叉树*t
    if (t != NULL)
    {linkedBinaryTree<E>:: visit(t);    //访问根节点
      preOrder(t->leftChild);    //前序遍历左子树
      preOrder(t->rightChild);    //前序遍历右子树
    }
}
```

## linkedBinaryTree<E> :: levelOrder

```
template <class E>
void linkedBinaryTree<E> :: levelOrder(
    void(*theVisit) (binaryTreeNode<E> *))
{//层次遍历二叉树
    binaryTreeNode<E> * t=root;
    arrayQueue<binaryTreeNode<E>*> q;
    while (t != NULL)
    { theVisit(t); //访问t
      //将t的左右孩子放入队列
      if (t->leftChild) q.push(t->leftChild);
      if (t->rightChild) q.push(t->rightChild);
      try {t=q.front();} //访问下一个节点
      catch (queueEmpty) {return;}
      q.pop();
    }
}
```

## 类linkedBinaryTree的扩充

- 增加如下操作:
  - preOrderOutput(): 按前序方式输出二叉树中的元素。
  - inOrderOutput(): 按中序方式输出二叉树中的元素。
  - postOrderOutput(): 按后序方式输出二叉树中的元素。
  - levelOrderOutput(): 逐层输出二叉树中的元素。
  - height(): 返回树的高度。

## 输出(Output)

```
Private:
    .....
    static void output(binaryTreeNode<E> *t)
        { cout << t->element << ' ';}
Public:
    .....
    void preOrderOutput()
        { preOrder(output); cout << endl;}
    void inOrderOutput()
        { inOrder(output); cout << endl;}
    void postOrderOutput()
        {postOrder(output); cout<<endl;}
    void levelOutput()
        { levelOrder(output); cout << endl;}
}
```

## 计算高度(Height)

```
Public:
.....
int Height( ) const {return height (root);}
private:
.....
int height (binaryTreeNode<E> *t) const;
.....
template <class E>
int linkedBinaryTree<E>::height(BinaryTreeNode<E> *t)
{ //返回二叉树*t的高度
  If (t==NULL) return 0; // 空树
  int hl = height(t->leftChild); // 左子树的高度
  int hr = height(t->rightChild); // 右子树的高度
  if (hl > hr) return ++hl;
  else return ++hr;
}
```

## 测验

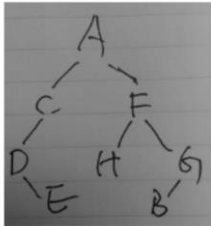
已知一个二叉树的前序遍历结果是 (ACDEFHGB)，中序遍历结果是 (DECAHFGB)，请问后续遍历结果是 ( )。

- A. HGFEDCBA
- B. EDCBHGFA
- C. BGFHEDCA
- D. EDCBHGFA
- E. BEGHDFCA
- F. BGHFEDCA

## 测验

选B

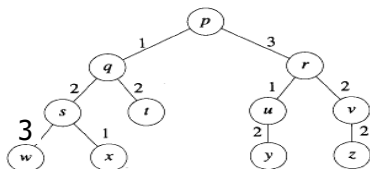
根据已知条件得到树结构如下:



## 11.9 应用

- 11.9.1 设置信号放大器
- 11.9.2 并查集

### 11.9.1 设置信号放大器



- 设计一个算法确定把信号放大器放在何处。
- 目标是要使所用的放大器数目最少并且保证信号衰减(与源端信号相关)不超过给定的容忍值。

### 树形分布网络信号放大器的放置

- $degradeFromParent(i)$  —— 节点*i* 与其父节点间的衰减量
- If  $degradeFromParent(i) >$  容忍值, 则不可能通过在节点*i*处放置放大器来使信号的衰减不超过容忍值。
- $degradeToLeaf(i)$  —— 从节点*i* 到以*i* 为根节点的子树的任一叶子的衰减量的最大值。
  - 若 *i* 为叶节点, 则  $degradeToLeaf(i) = 0$
  - 对其它节点*i*,  $degradeToLeaf(i) = \max\{degradeToLeaf(j) + degradeFromParent(i)\}$    
  $j$  是 *i* 的孩子

必须遍历整棵树, 先访问子节点然后访问父节点。

这样当访问一个节点时, 就可同时计算其 D 值

这种遍历方法是高度大于 2 的树的后序遍历的一种自然扩充

## 计算 $degradeToLeaf$ 和放置放大器的伪代码

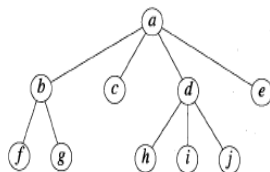
```

degradeToLeaf(i) = 0 ;
for (i 的每个孩子j)
if (degradeToLeaf(j) + degradeFromParent(j)) > 容忍值)
{ 在j 放置放大器;
  degradeToLeaf(i) =
    max { degradeToLeaf(i), degradeFromParent(j) };
}
else
degradeToLeaf(i) = max { degradeToLeaf(i),
  degradeToLeaf(j) + degradeFromParent(j) };

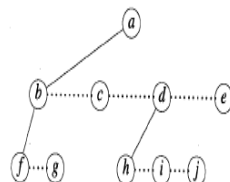
```



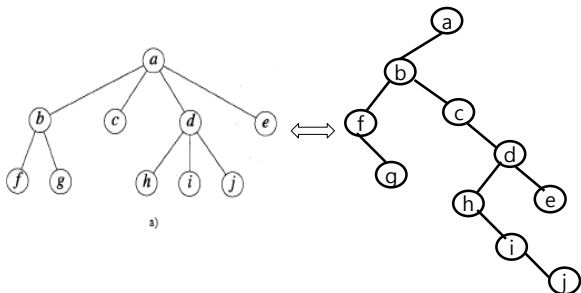
## 树的二叉树描述



- 对于树 $t$ 的每个节点 $x$ ,
- $x$ 节点的 $leftChild$ 指针指向 $x$ 的第一个孩子。
- $x$ 节点的 $rightChild$ 域指向 $x$ 的下一个兄弟。

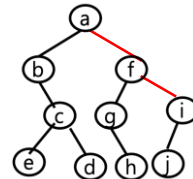
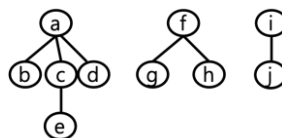


## 树的二叉树描述



## 森林的二叉树表示

- 森林(forest)是0棵或多棵树的集合。
- 森林的二叉树表示
  - 首先得到森林中每棵树(设有 $m$ 棵树)的二叉树描述。
  - 然后, 第 $i$ 棵作为第 $i-1$ 棵树的右子树( $2 \leq i \leq m$ )。

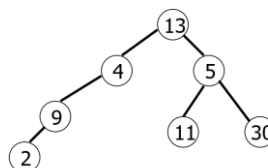


## 11.9.2 并查集

- 并查集示例:
  - $n = 14$ ;  $n$ 个元素从1号到 $n$ 号
  - $R = \{ (1, 11), (7, 11), (2, 12), (12, 8), (11, 12), (3, 13), (4, 13), (13, 14), (14, 9), (5, 14), (6, 10) \}$
  - 等价类: 相互等价的元素的最大集合
    - $\{1, 2, 7, 8, 11, 12\}$
    - $\{3, 4, 5, 9, 13, 14\}$
    - $\{6, 10\}$

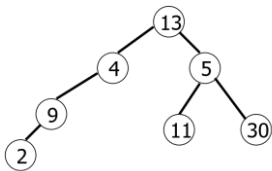
## 集合(类)的树形描述

- 每个集合(类)描述为一棵树
- 用根元素作为集合标识符。
- 例: 13:  $\{2, 4, 5, 9, 11, 13, 30\}$



### find 操作

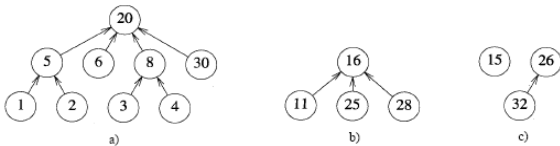
- find(i) : 返回i所在树的根元素。



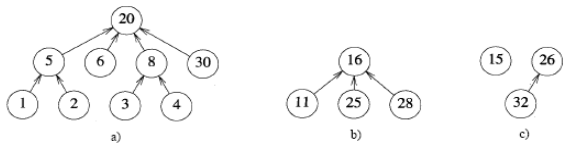
- 从i 所在的节点开始，沿着节点到其父节点移动，直到到达根节点位置。
- 返回根元素。

### 树的描述

- 每个非根节点都指向其父节点。
- 20: {1,2,3,4,5,6,8,30,20}
- 16: {11,25,28,16}
- 15: {15}
- 26: {32,26}



### 树的描述

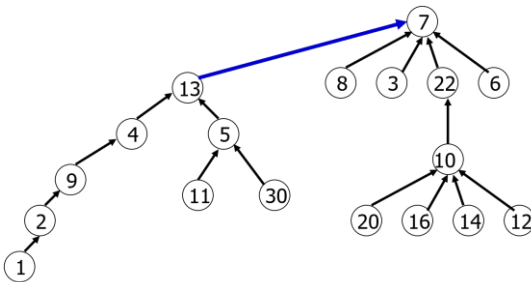


树的描述

int \*parent;

### Unite 操作

- union(7,13)



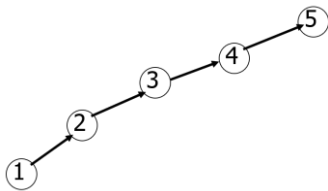
```
void initialize(int numberOfElements)
{ // 初始化，每个类/树有一个元素
    parent = new int[numberOfElements+1];
    for (int e = 1; e <= numberOfElements; e++)
        parent[e] = 0;
}
int find(int theElement)
{ // 返回包含theElement的树的根节点
    while (parent[theElement] != 0)
        theElement = parent[theElement]; // 上移一层
    return theElement;
}
void unite(int rootA, int rootB)
{ // 将根为rootA 和rootB的两棵树进行合并
    parent[rootB] = rootA;
}
```

### 时间复杂性

- 假设一个系列操作：要执行u 次合并和f 次查找。
- 每次合并前都必须执行两次查找，
  - 可假设f > u。
- 每次合并所需时间为Θ(1)。

### 时间复杂性

- 每次查找所需时间由树的高度决定。
- 在最坏情况下，有 $m$ 个元素的树的高度为 $m$ 。
  - 当执行以下操作序列时，即可导致最坏情况出现：  
 $Unite(2,1)$ ,  $Unite(3,2)$ ,  $Unite(4,3)$ ,  $Unite(5,4)$ , ...



### 时间复杂性

- 要执行一个系列操作( $u$  次合并和 $f$  次查找)的时间为  $O(fu)$

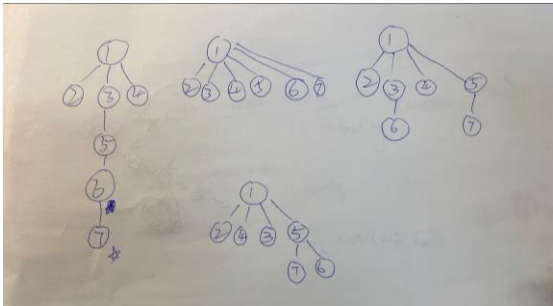
### 性能改进-重量规则

- [重量规则]若树 $i$ 节点数少于树 $j$ 节点数，则将 $j$ 作为 $i$ 的父节点。否则，将 $i$ 作为 $j$ 的父节点。
- [高度规则]若树 $i$ 的高度小于树 $j$ 的高度，则将 $j$ 作为 $i$ 的父节点，否则将 $i$ 作为 $j$ 的父节点。

### 提高在最坏情况下的性能的方法

- 路径的缩短可以通过称为路径压缩(path compression)的过程实现。
- 1. 紧凑路径法(path compaction)
  - 改变从 $e$ 到根节点路径上所有节点的parent指针，使其指向根节点。
- 2. 路径分割法(path splitting)
  - 改变从 $e$ 到根节点路径上每个节点(除了根和其子节点)的parent指针，使其指向各自的祖父节点。
- 3. 路径对折法(path halving)
  - 改变从 $e$ 到根节点路径上每隔一个节点(除了根和其子节点)的parent域，使其指向各自的祖父节点。

### 提高在最坏情况下的性能的方法



### 作业

- P280. 练习12
- P295. 练习53