第11章

二叉树和其他树

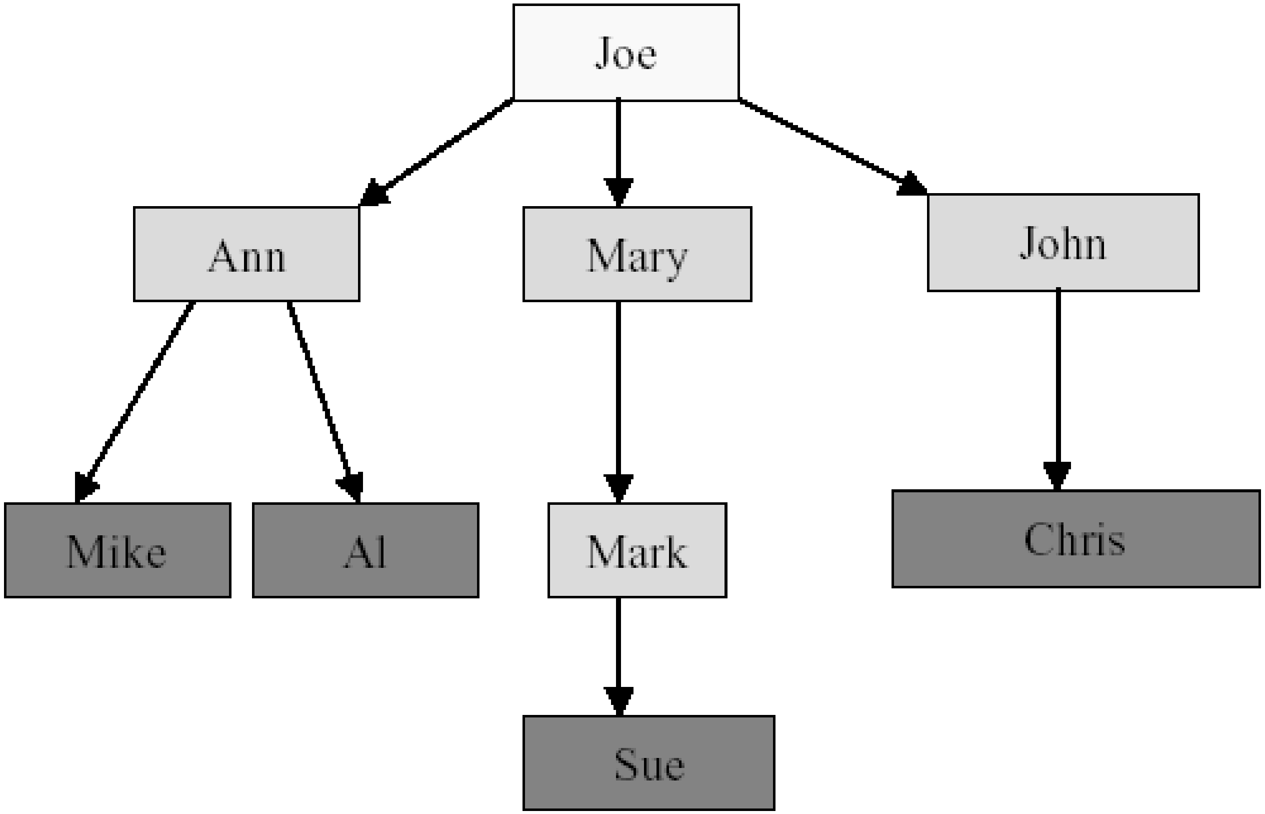
本章内容

- 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

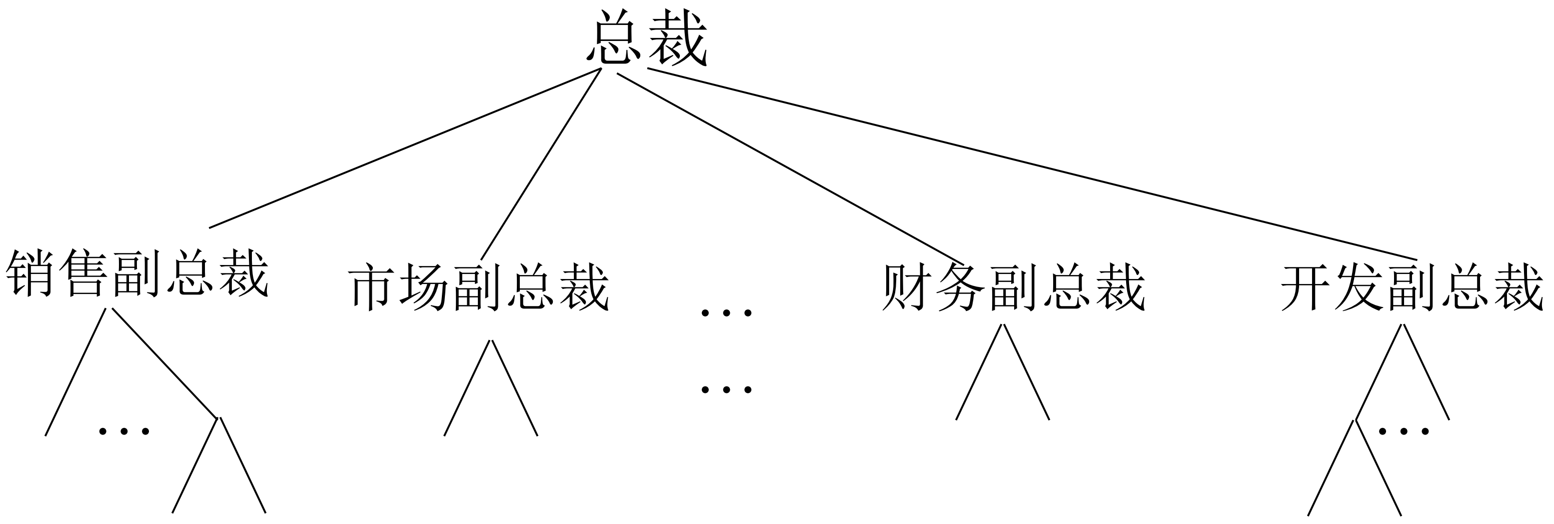
- 线性数据结构和表数据结构一般不适合于描述具有层次结构的数据。
- 例：层次结构的数据

例 11.1 Joe的后代



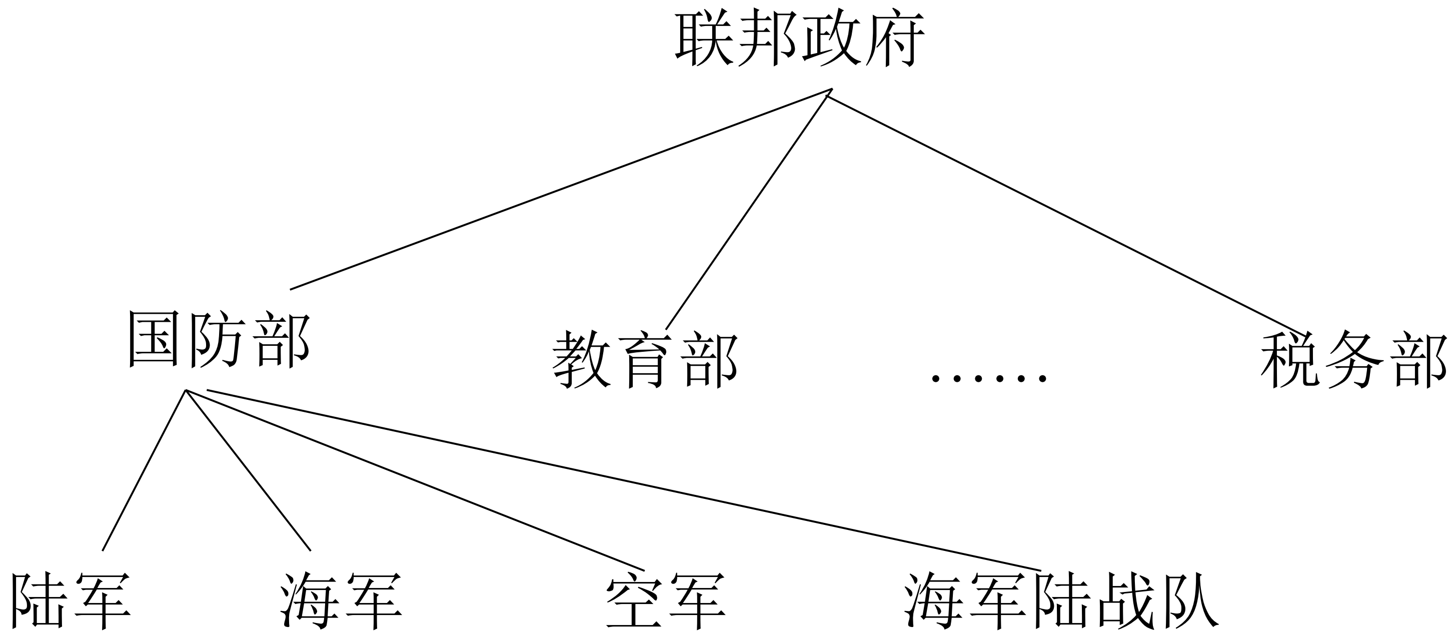
- 元素之间的关系是父母——子女

例 11.2 公司组织机构



- 元素之间的关系是上级——下级

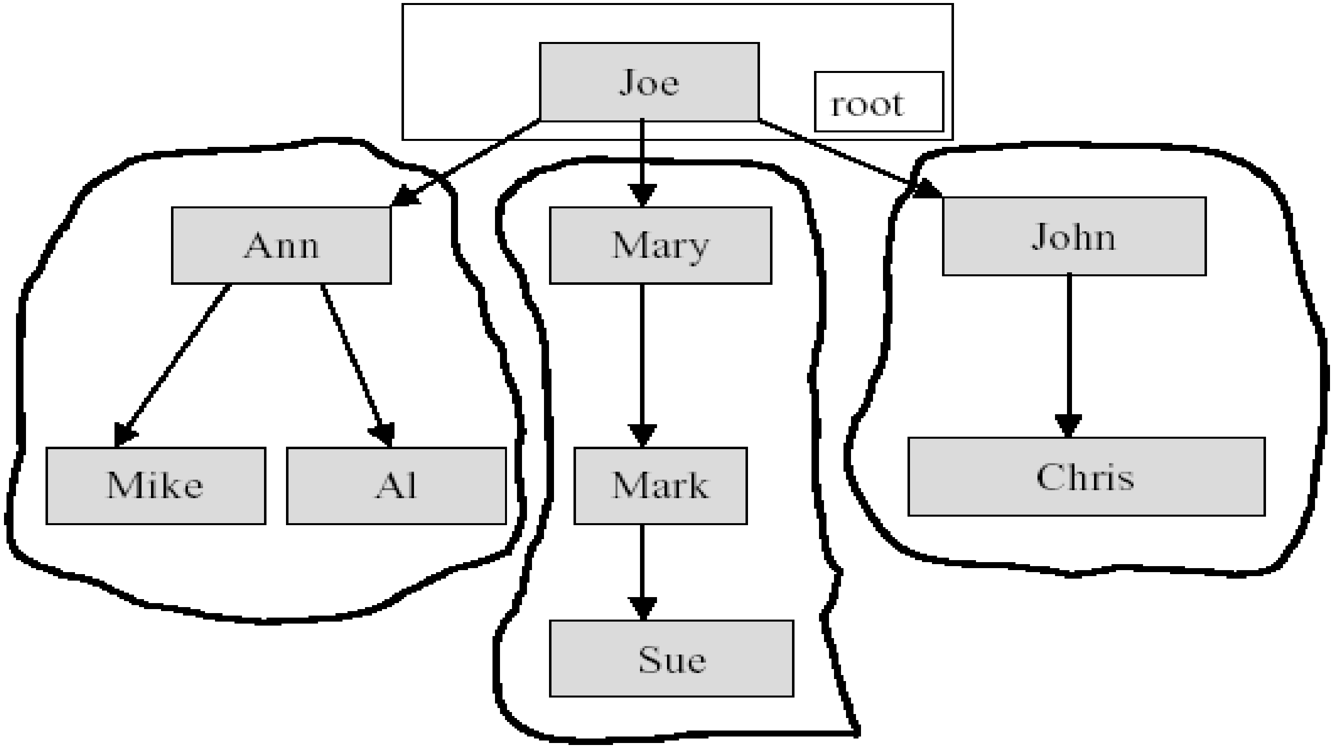
例 11.3 政府机构



- 元素之间的关系是整体——部分

树的定义

- 定义[树]：
  - 树(tree)t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).

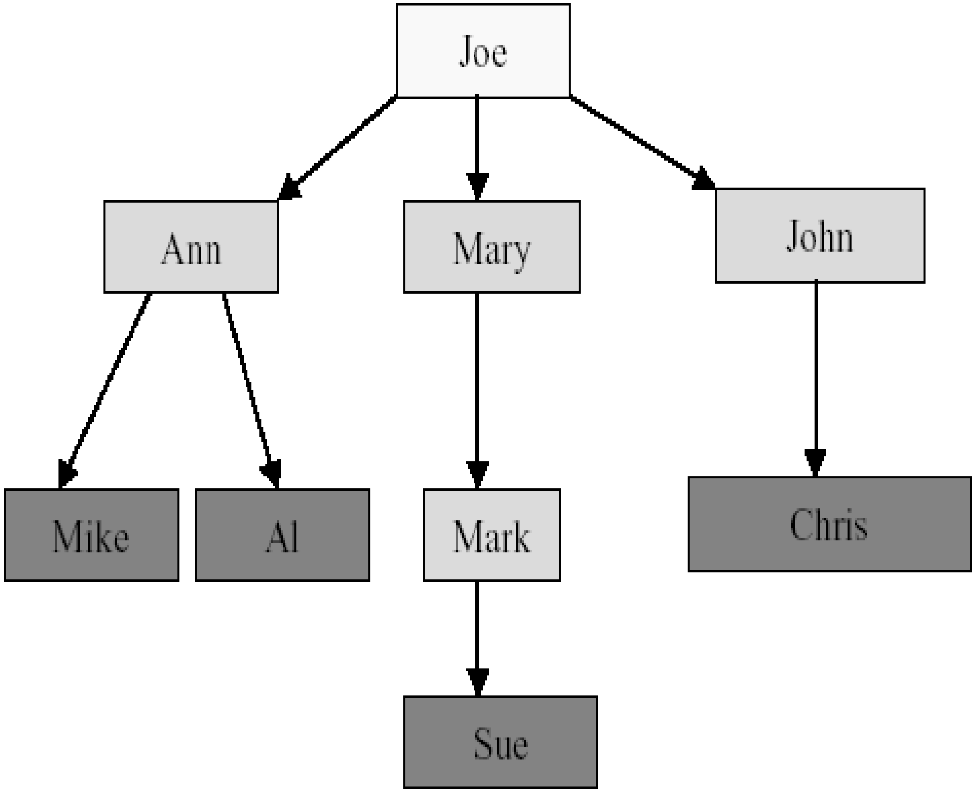


术语

- 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**

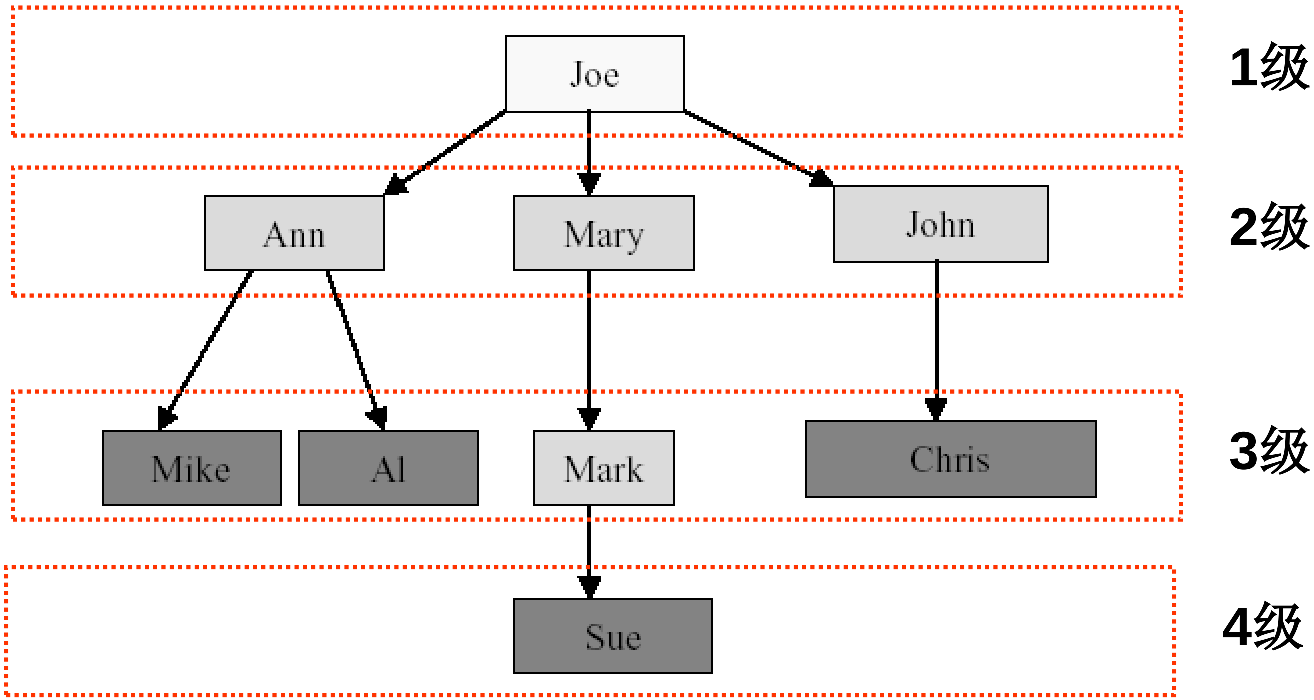
兄弟**(Mary) = {Ann,John}**

祖先**(Mike) = {Ann,Joe}**

后代**(Mary}={Mark,Sue}**

术语

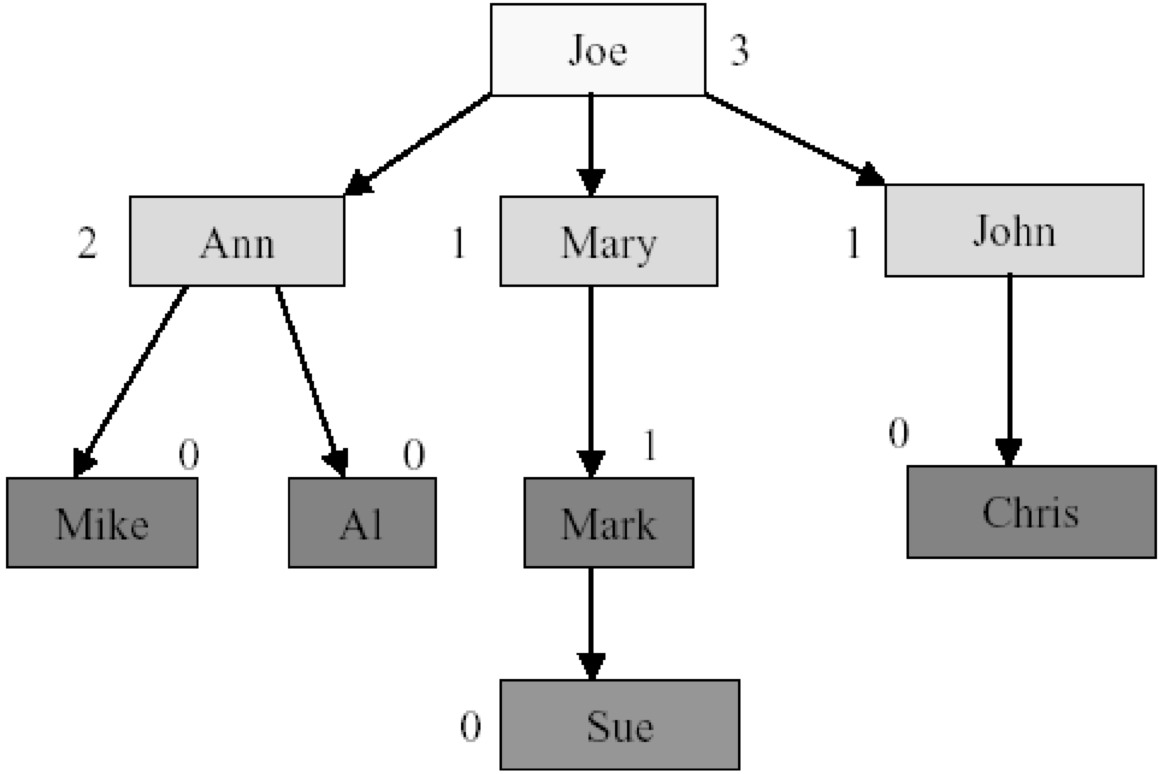
- 级(level)/层次：指定树根的级为1，其孩子(如果有)的级为2。 一个元素的级=其父母的级+1.



- 二叉树的高度(height)或深度(depth)是指该二叉树的级数(或层数)。

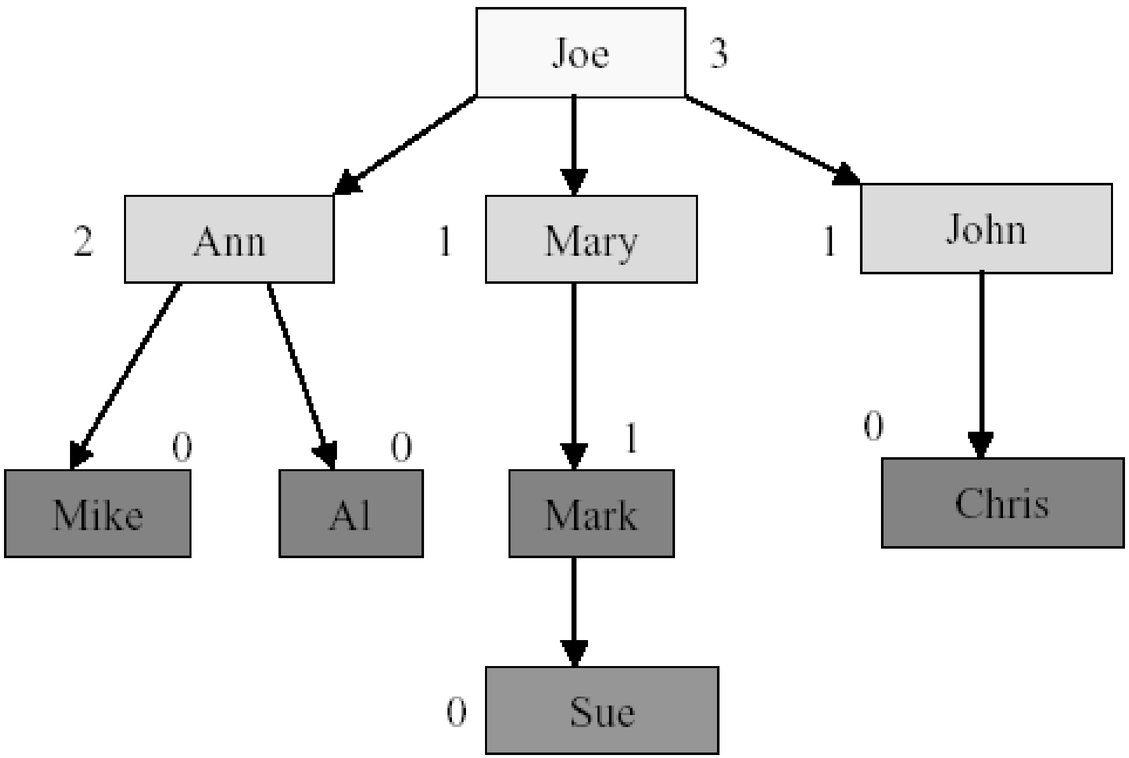
术语

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



术语

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



树的度 = 3

线性结构	树型结构
第一个数据元素 (无前驱)	根结点 (无前驱)
最后一个数据元素 (无后继)	多个叶子结点 (无后继)
其它数据元素 (一个前驱、 一个后继)	其它数据元素 (一个前驱、 多个后继)

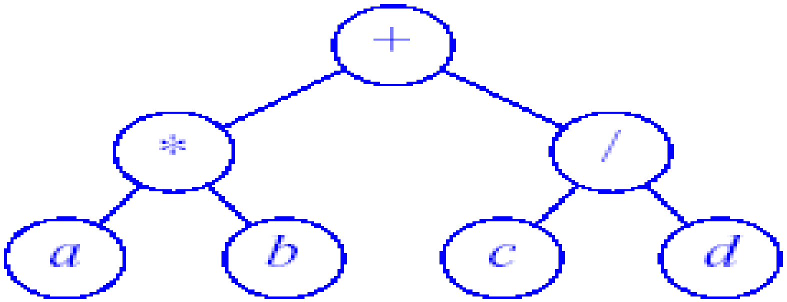
11.2 二叉树

- 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**.

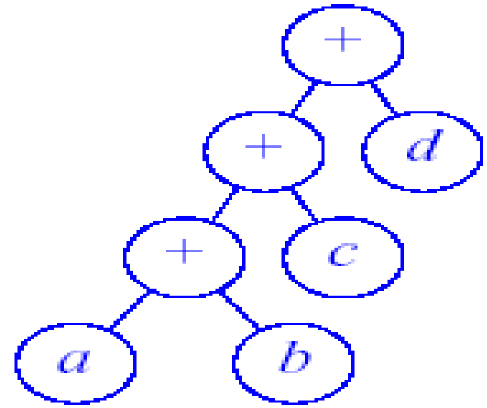
二叉树和树的区别

- 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**。

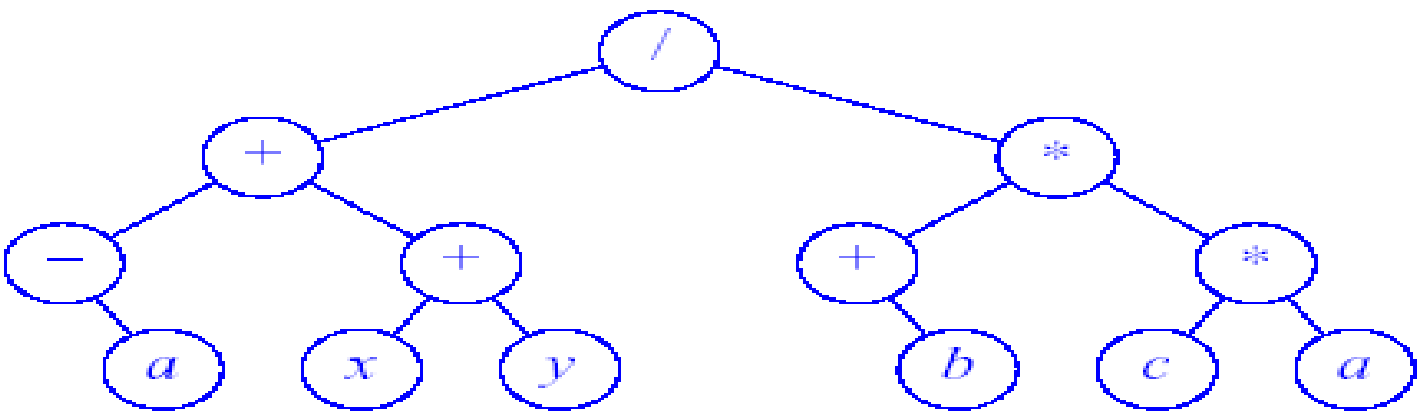
二叉树表示数学表达式—表达式树



(a) $(a * b) + (c / d)$



(b) $((a + b) + c) + d)$



(c) $((-a) + (x + y)) / ((+b) * (c * a))$

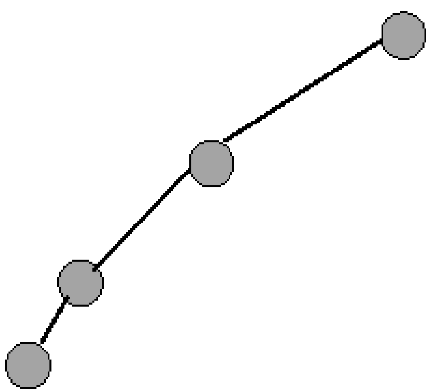
11.3 二叉树的特性

- 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。

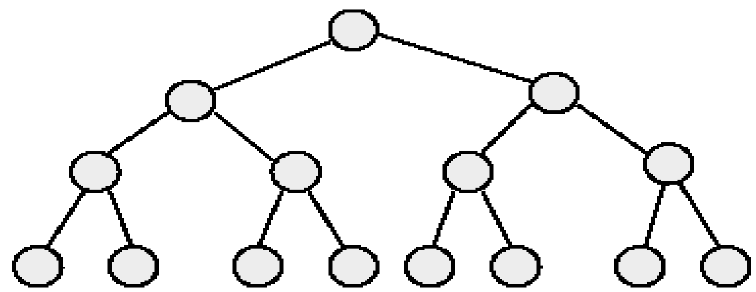
11.3 二叉树的特性

✓ 特性 2:

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



元素数最少为h



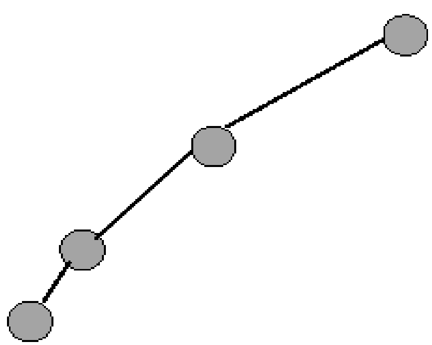
第i 层节点元素最多为2^{i-1}

元素的总数最多为$\sum 2^{i-1} = 2^h - 1$ 个元素

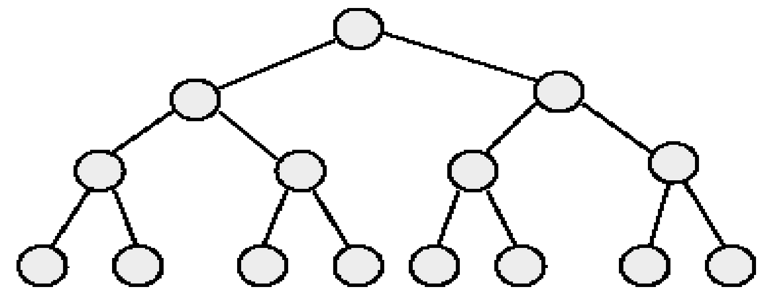
11.3 二叉树的特性

✓ 特性 3:

- 包含n(n≥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：



高度最大为*n*。



高度最小为：

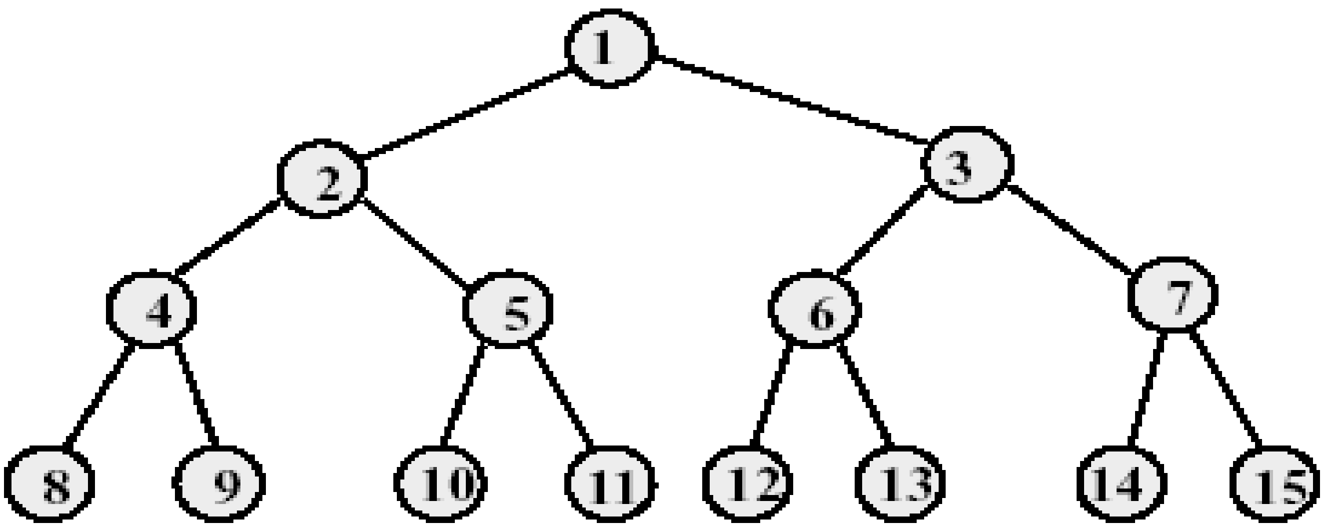
$$n \le 2^h-1$$

$$h \ge log_2(n+1)$$

$$h=\lceil (log_2(n+1)) \rceil$$

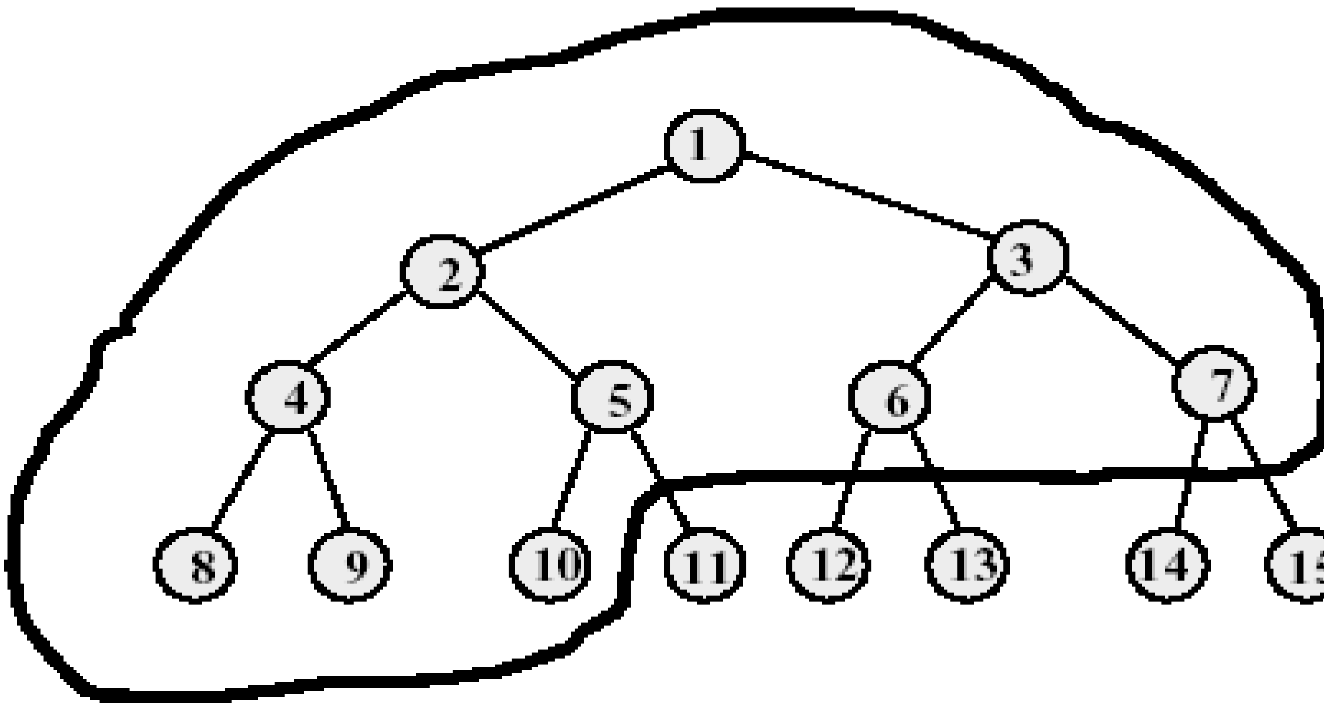
满二叉树

- 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-i，$1 \leqslant i \leqslant k$，所得到的二叉树被称为完全二叉树(complete binary tree)。

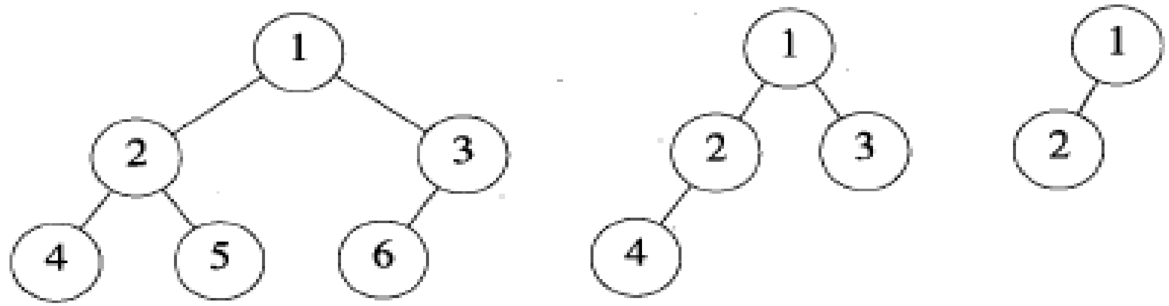


完全二叉树

- 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~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，$1 \le i \le n$。则有以下关系成立：

1. 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$\lfloor (i/2) \rfloor$.
2. 当$2i > n$时，该元素无左孩子。否则，其左孩子的编号为$2i$.
3. 若$2i+1>n$，该元素无右孩子。否则，其右孩子编号为 $2i+1$.

二叉树的特性

特性5. 度为0的结点数＝度为2的节点数 ＋ 1

证明:

设 二叉树上结点总数 $n = n_0 + n_1 + n_2$

又 二叉树上分支总数 $b = n_1 + 2n_2$

而 $b = n-1 = n_0 + n_1 + n_2 - 1$

由此， $n_0 = n_2 + 1$

思考题

- 一棵树，有n_1个度为1的节点，n_2个度为2 的节点，……，n_m个度为m的节点。有多少个叶节点？

- 总结点数 $n = n_0 + n_1 + n_2 + \ldots + n_m$
- 总分支数 $e = n\text{-}1 = n_0 + n_1 + n_2 + \ldots + n_m - 1$
- $= m*n_m + (m-1)*n_{m\text{-}1} + \ldots + 2*n_2 + n_1$
- 则有

$$n_0 = \left(\sum_{i=2}^{m} (i-1) n_i \right) + 1$$

- 叶节点=$1+n_2+2n_3+\ldots+(m-1)n_m$

思考题

一棵i层的k叉树，最多有多少个节点？

$N= k^{i}-1/k-1$

节点编号为i的节点，其第1个子节点若存在，编号为多少？

$(i-1)*k +2$

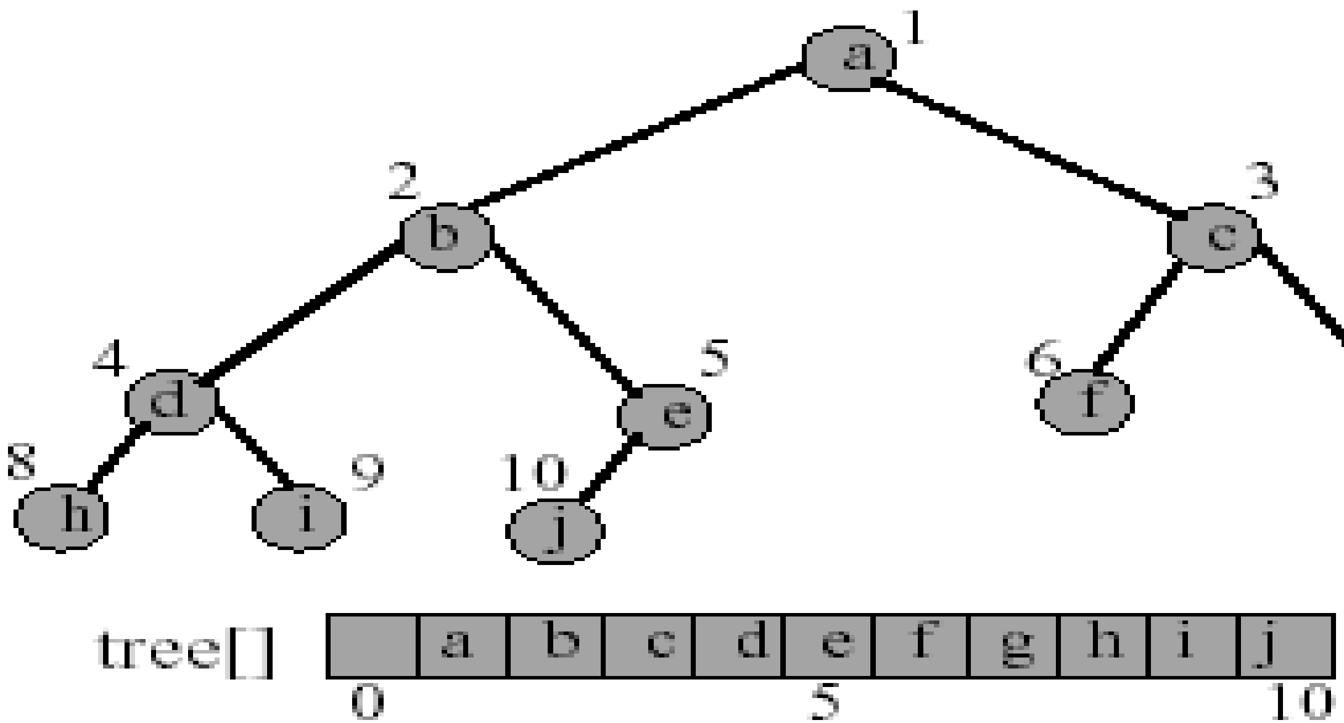
i若>1,他的父节点编号为 $(i+k-2)/k \rfloor$

11.4 二叉树描述

- 数组描述
- 链表描述

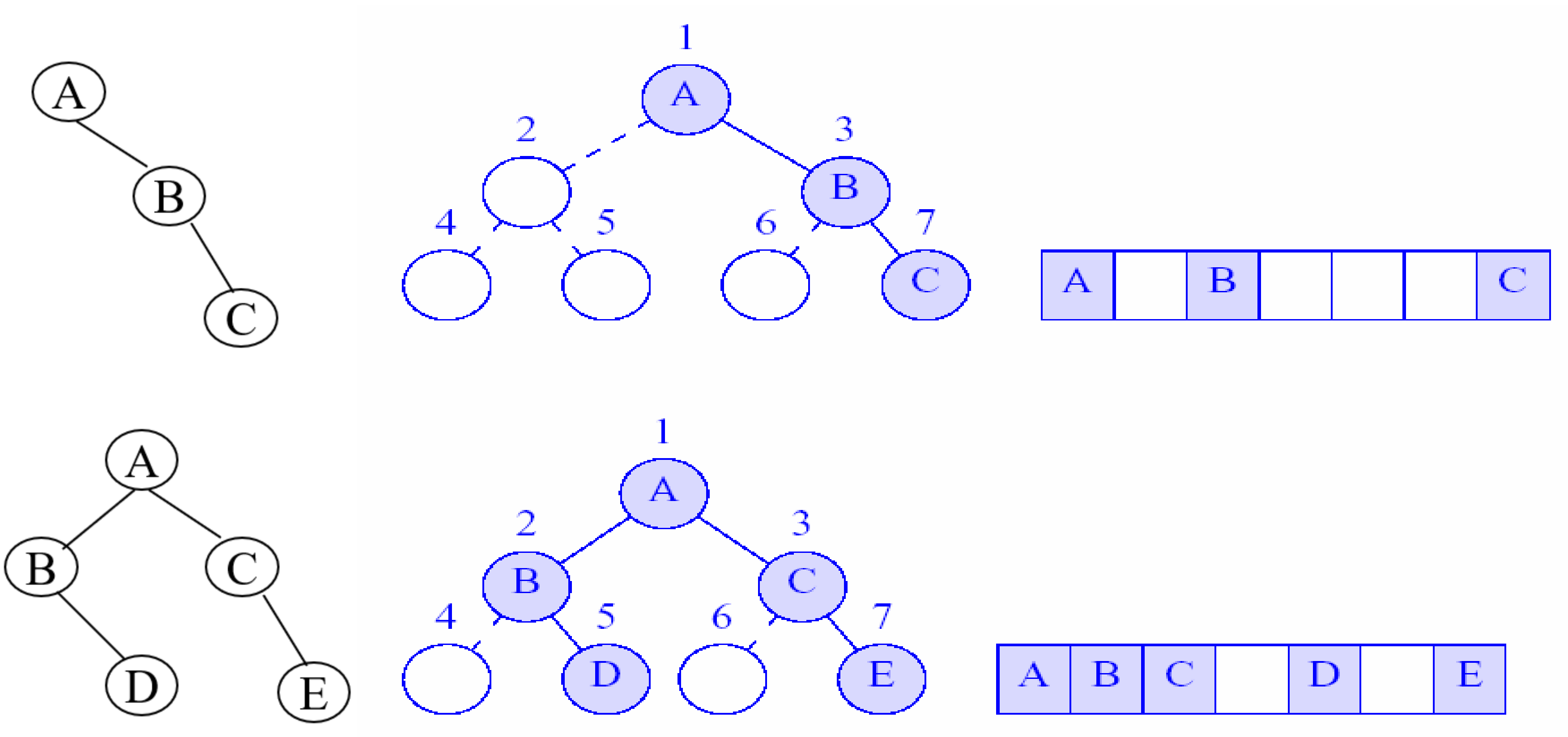
11.4.1 数组描述

- 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。



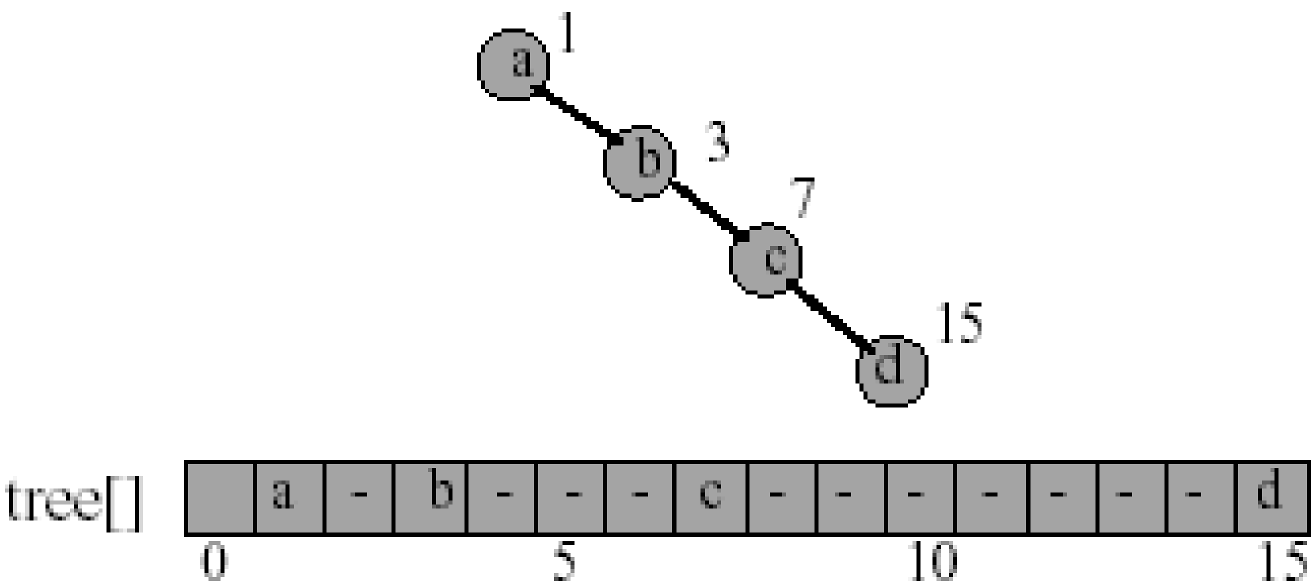
11.4.1 数组描述

- 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 。



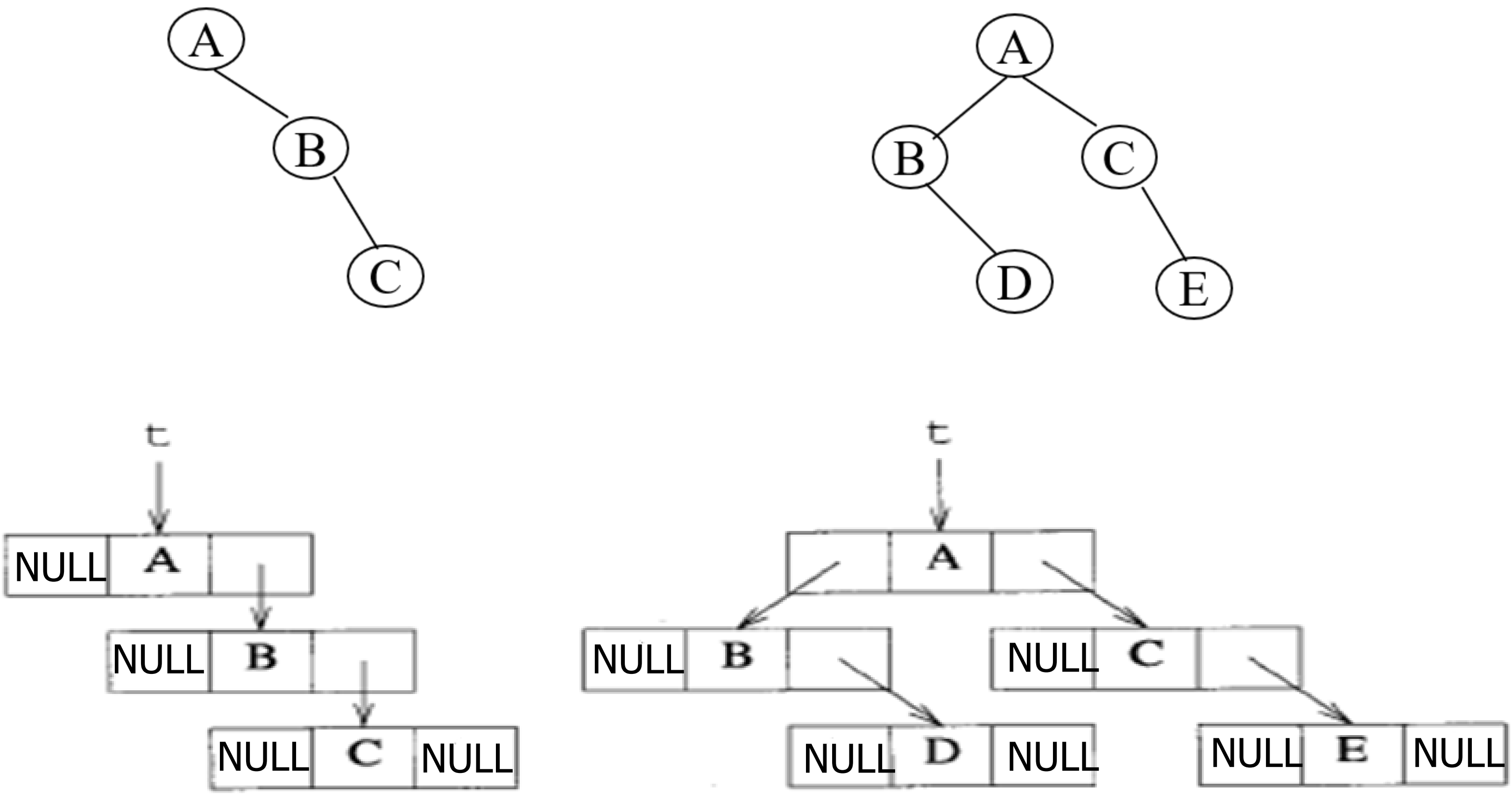
- 当缺少的元素数目比较少时，数组描述方法是有效的.

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


A
B
C
A
B
C
D
E
t
NULL
A
NULL
B
NULL
C
NULL
t
A
NULL
B
NULL
C
NULL
D
NULL
NULL
E
NULL

a) b)

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): element(theElement)
    //只有数据参数
        {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

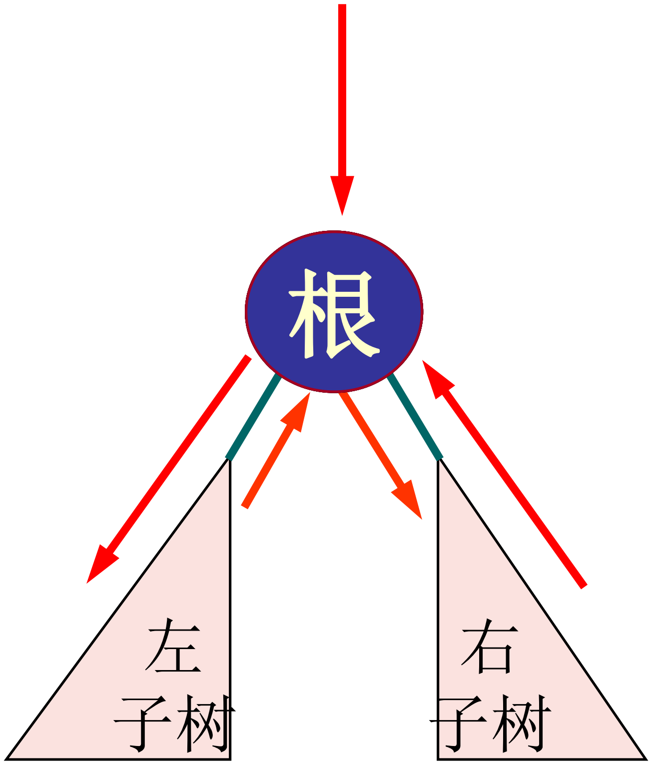
- 确定其高度
- 确定其元素数目
- 复制
- 在屏幕或纸上显示二叉树
- 确定两棵二叉树是否一样
- 删除整棵树
- 若为数学表达式树，计算该数学表达式
- 若为数学表达式树，给出对应的带括号的表达式

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)。

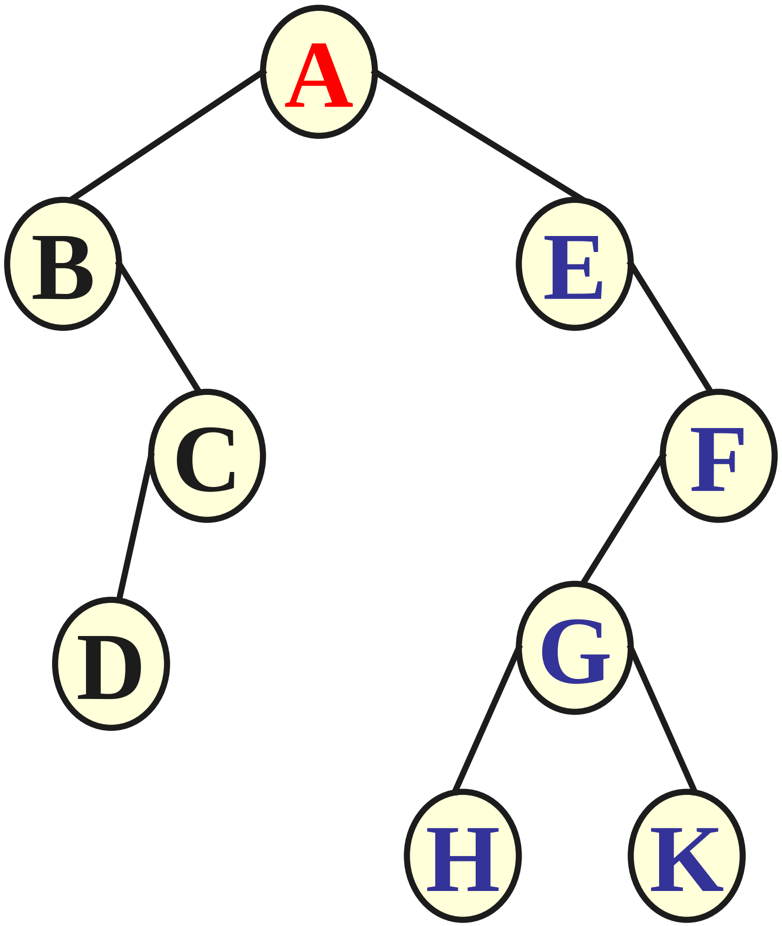
二叉树遍历方法

- 有四种遍历二叉树的方法：
  - 前序遍历
  - 中序遍历
  - 后序遍历
  - 逐层遍历



二叉树遍历方法

例如：



前序序列：

A B C D E F G H K

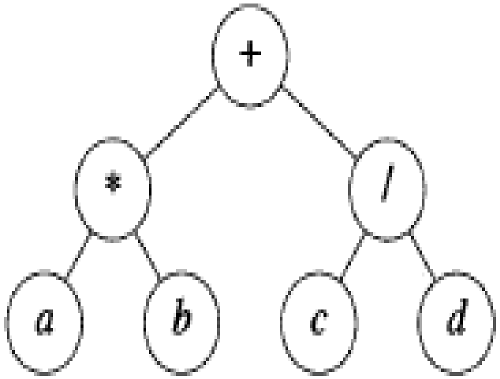
中序序列：

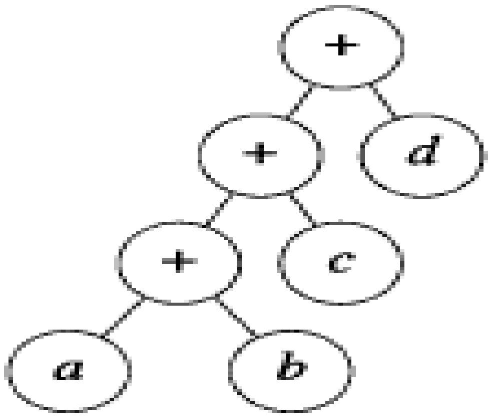
B D C A E H G K F

后序序列：

D C B H K G F E A

遍历示例 (visit(t)= cout(t ->element))





前序:+*ab/cd

中序:a*b+c/d

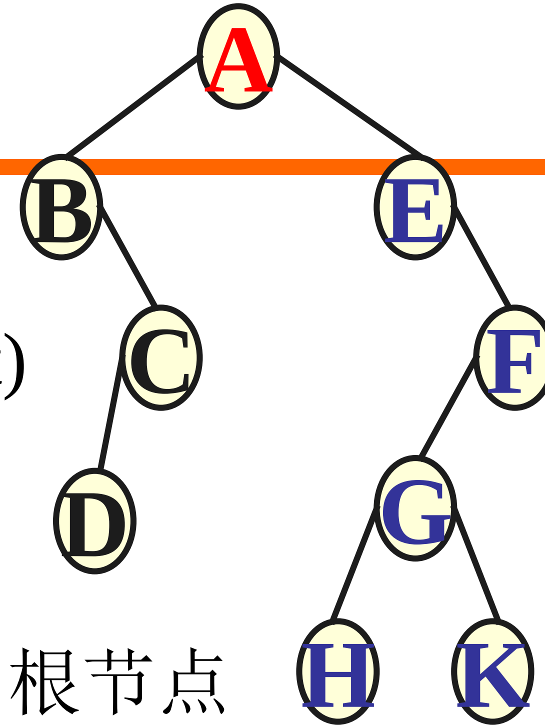
后序:ab*cd/+

前序:+++abcd

中序:a+b+c+d

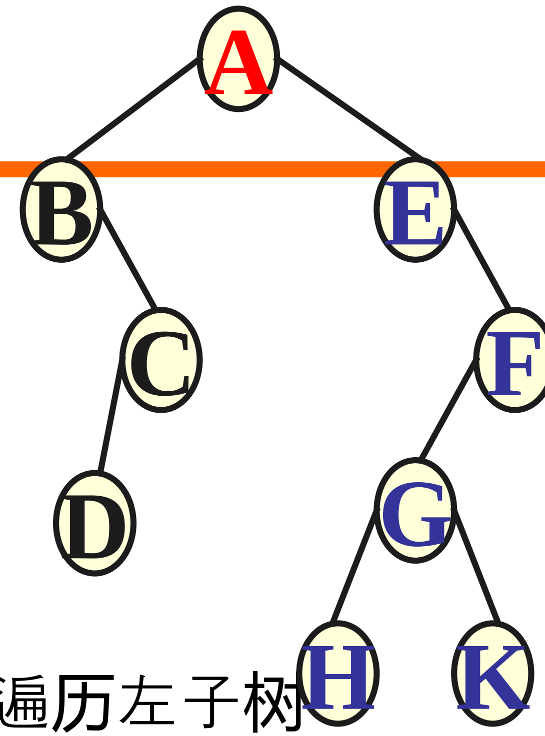
后序:ab+c+d+

前序遍历



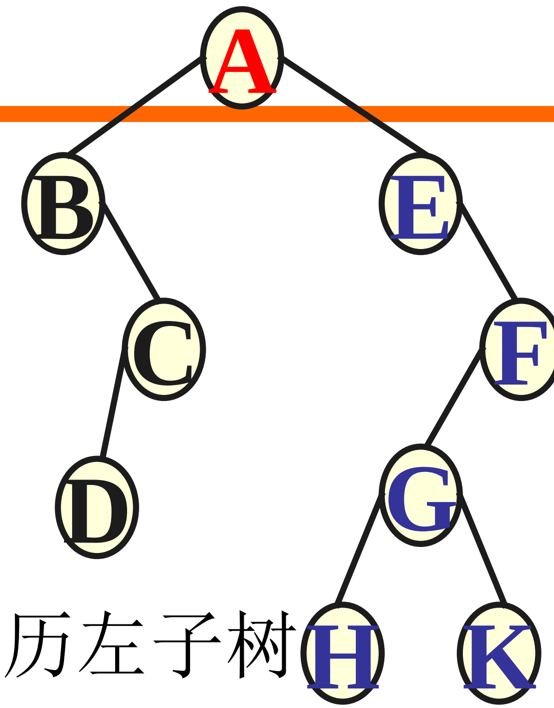

```
template<class E>
void preOrder (binaryTreeNode<E> *t)
{//前序遍历二叉树*t
   if (t != NULL)
     {visit(t);                    //访问根节点
      preOrder(t->leftChild);      //前序遍历左子树
      preOrder(t->rightChild);     //前序遍历右子树
     }
}
```

中序遍历



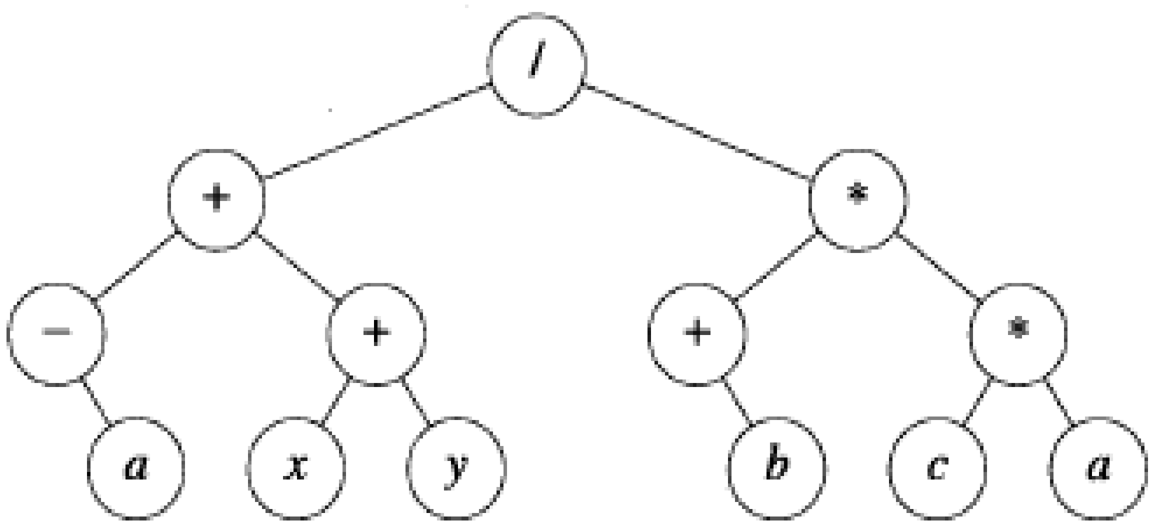

```
template<class E>
void inOrder(binaryTreeNode<E> *t)
{//中序遍历二叉树*t
   if (t != NULL)
    { inOrder(t->leftChild);        //中序遍历左子树
      visit(t);                     //访问根节点
      inOrder(t->rightChild);       //中序遍历右子树
    }
}
```

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



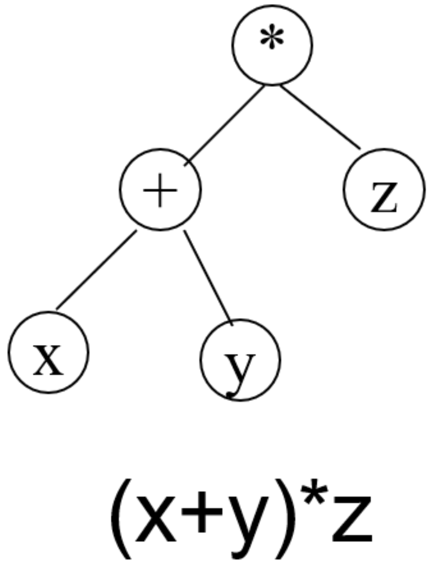
前序:/+-a+xy*+b*ca --表达式的前缀形式

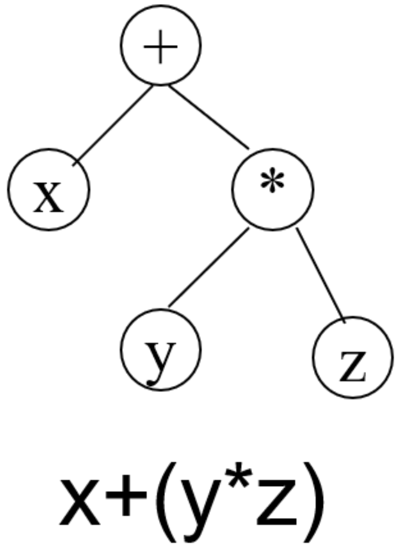
中序:-a+x+y/+b*c*d --表达式的中缀形式

后序:a-xy++b+ca**/ --表达式的后缀形式

表达式二叉树的遍历

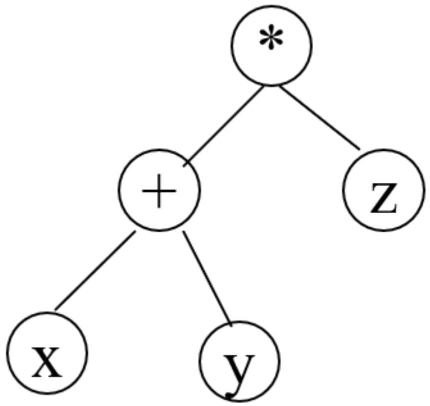
- 表达式的中缀形式存在歧义:
  - x+y*z



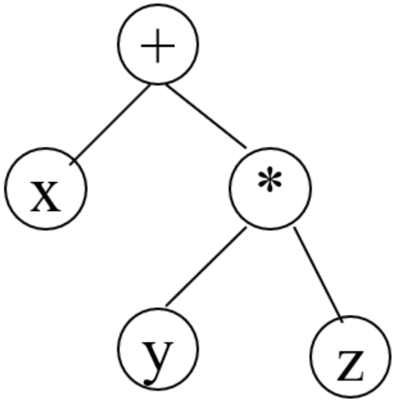


数学表达式二叉树的遍历

- 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。



(((x)+(y))*z)

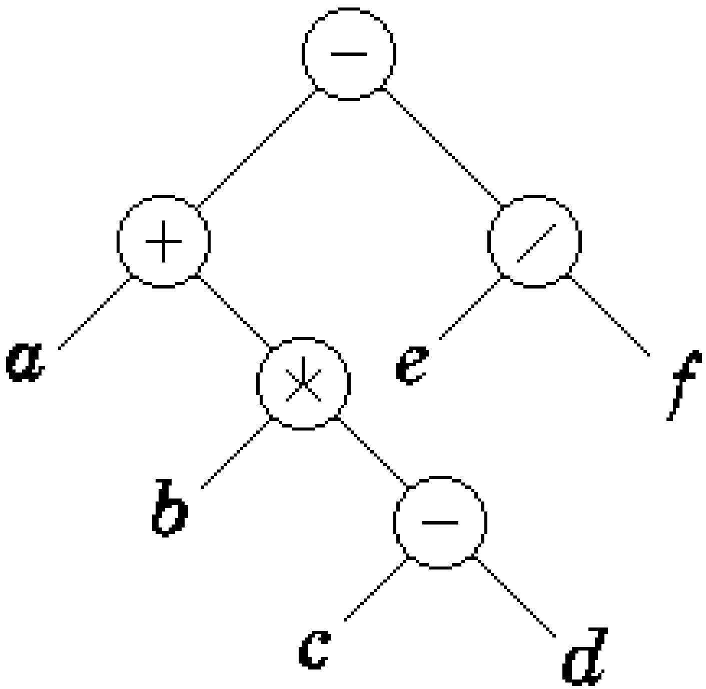


((x)+((y)*(z)))

输出完全括号化的中缀表达式

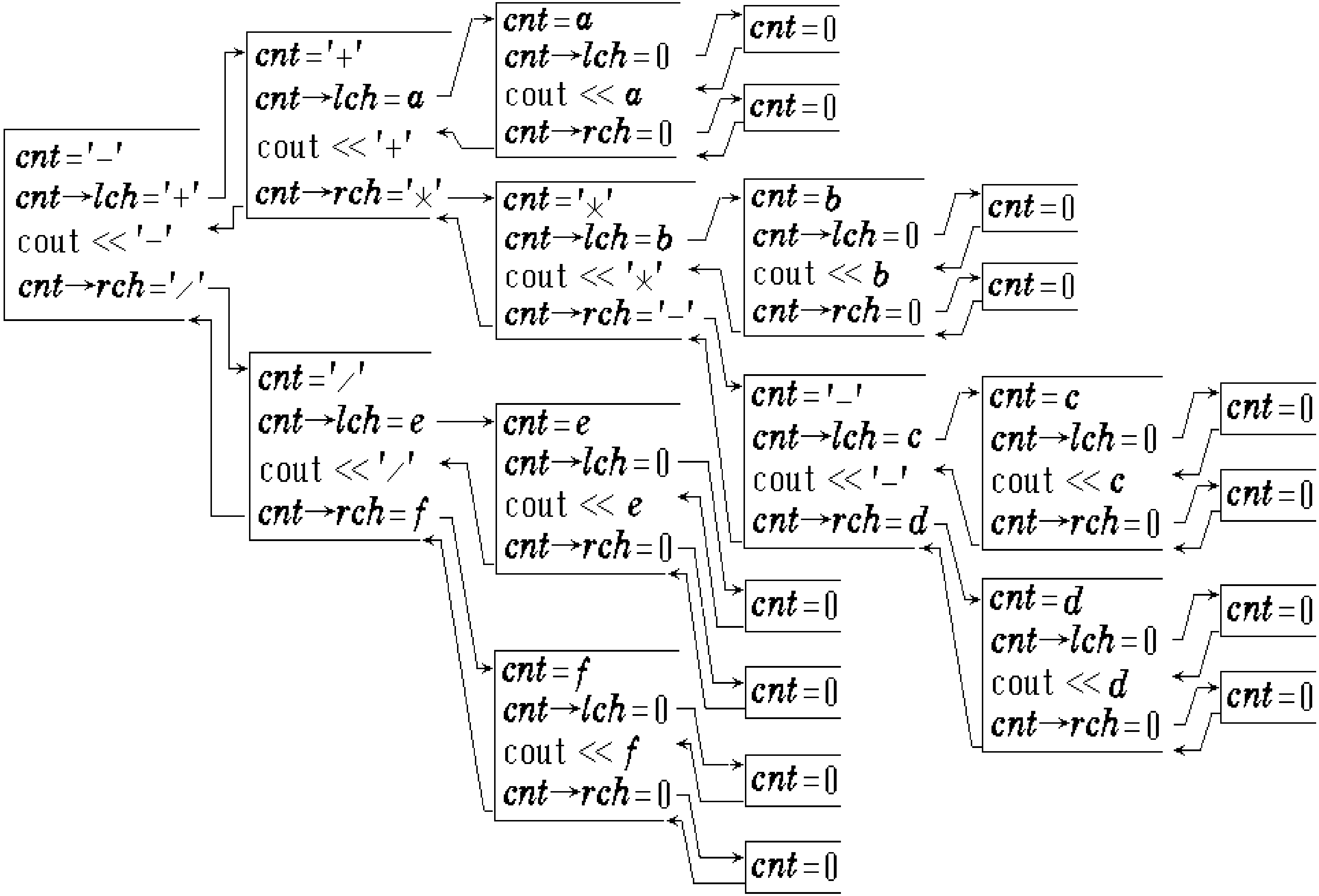
```
template <class E>
void infix(binaryTreeNode<E> *t)
{// 输出表达式的中缀形式
    if (t != NULL)
    {cout << '(';
     infix(t->leftChild);// 左操作数
     cout << t->element; // 操作符
     infix(t->rightChild);//右操作数
     cout << ')';}
}
```

- 前缀和后缀表达式中不会存在歧义。



中序遍历结果

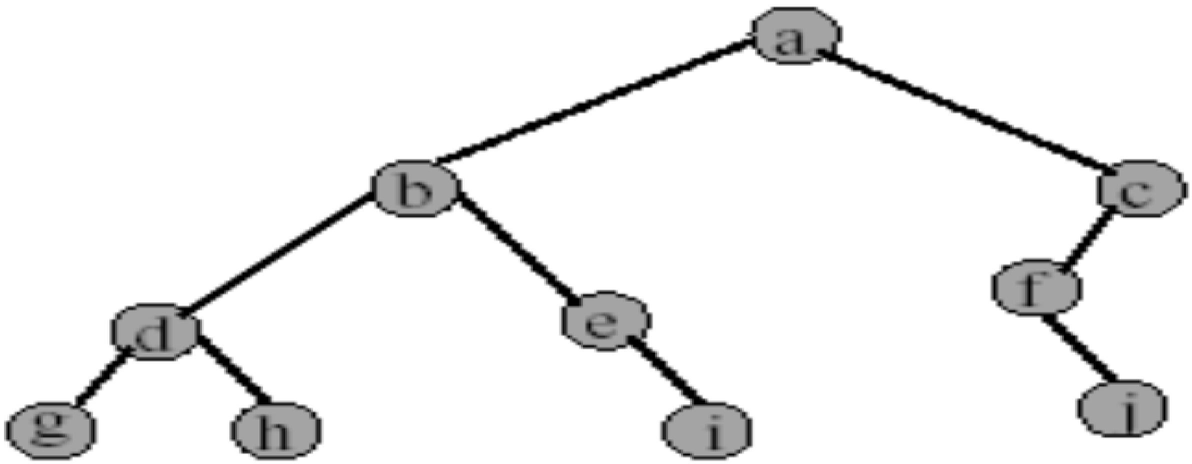
a + b * c - d - e / f



中序遍历二叉树的递归过程图解

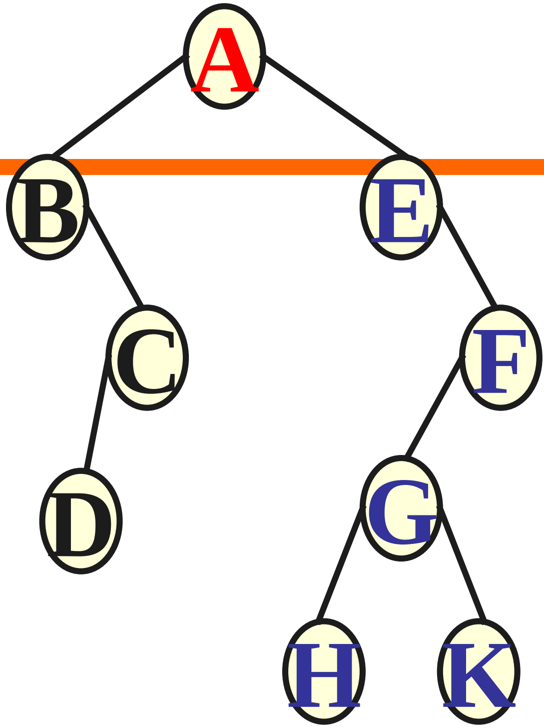
层次遍历

- 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

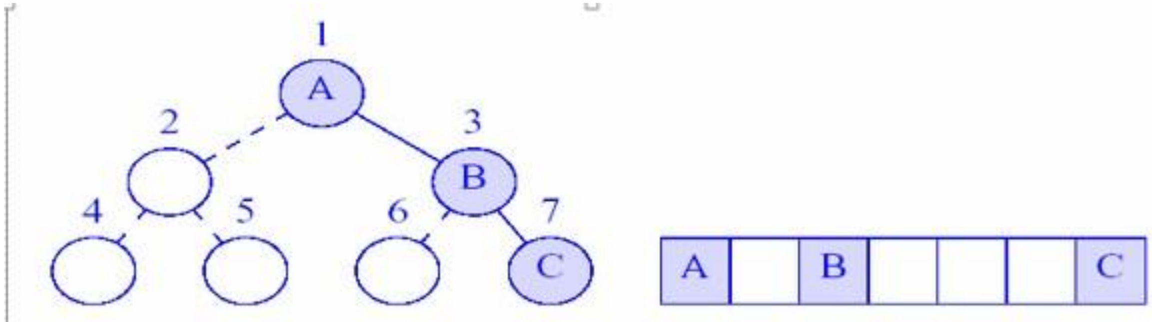
层次遍历




```
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();
      }
 }
```

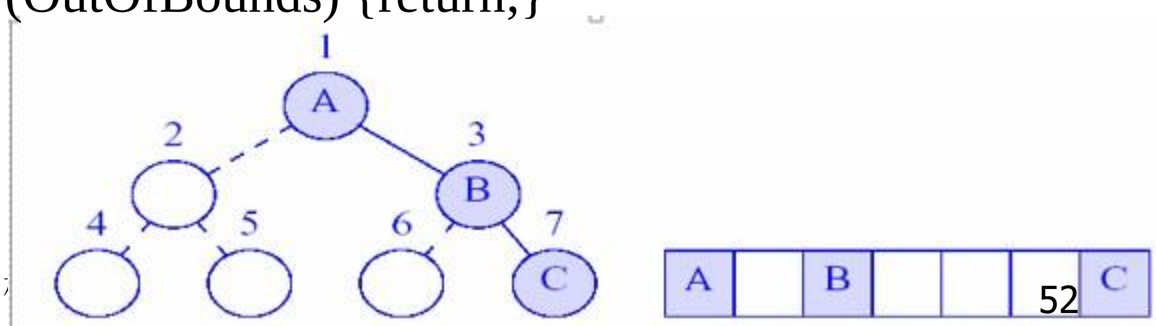
公式化描述的遍历

```
template <class T>
 void InOrder(T a[], int last, int i) {// Inorder traversal of binary tree
    with root a[i].
    if (i <= last && a[i]) {
     InOrder(a, last, 2*i); // do left subtree
     Visit(a, i); // visit tree root
     InOrder(a, last, 2*i+1); // do right subtree
}
 }
```



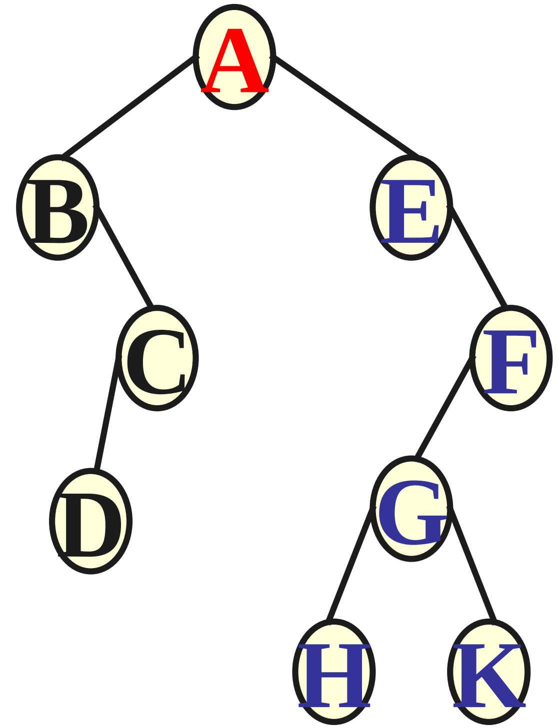
公式化描述的层次遍历

```
template <class T>
void LevelOrder(T a[], int last) {
// Level-order traversal of a binary tree. LinkedQueue<int> Q;

    if (!last) return; // empty tree int i = 1;
    // start at root
    while (true) {
        Visit(a, i);
        // put children on queue
        if (2*i <= last && a[2*i]) Q.Add(2*i); // add left child
        if (2*i+1 <= last && a[2*i+1]) Q.Add(2*i+1); // add right child
        // get next node to visit
        try {Q.Delete(i);} catch (OutOfBounds) {return;}
        }
    }
```



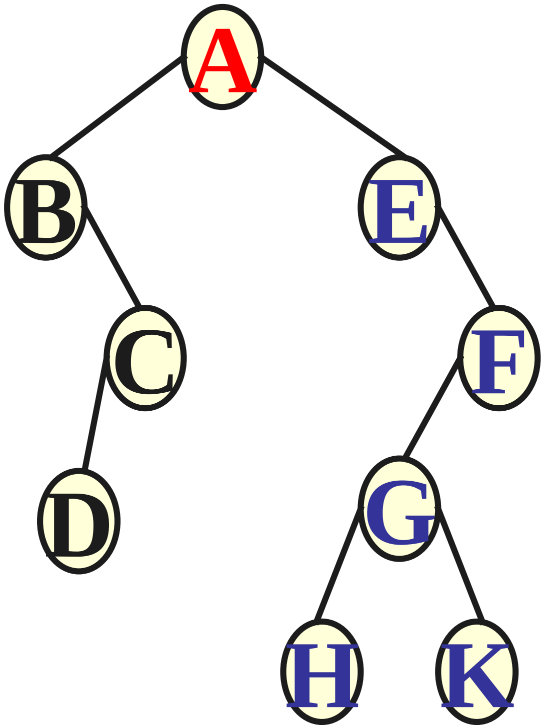
中序遍历的非递归实现

```
template<class T>
void InOrder(binaryTreeNode<T> *t)     //对*t进行中序遍历
{   stack  <BinaryTreeNode<T> *>   s(MaxLength);
    BinaryTreeNode<T> *p=t ;
   do {
          while (p)
                {s.push(p);   p=p->LeftChild;}
          if (!s.IsEmpty())
                { p=s.top();
                  s.pop();
                  Visit(p);
                  p=p->RightChild;
                }
      } while (p||!s.IsEmpty())
}
```



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Θ(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   {reurn 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();
     {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 << end1;}
    void inOrderOutput( )
        { inOrder(output); cout << end1;}
    void postOrderOutput( )
        {postOrder(output); cout<<end1;}
    void levelOutput( )
        { levelOrder(output); cout << end1;}
```

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;
}
```

统计二叉树中节点个数

算法基本思想:

先序(或中序或后序)遍历二叉树，在遍历过程中统计结点。

由此，需在遍历算法中增添一个“计数”的参数，并将算法中“访问结点” 的操作改为:计数器增1。

统计节点数（Size）

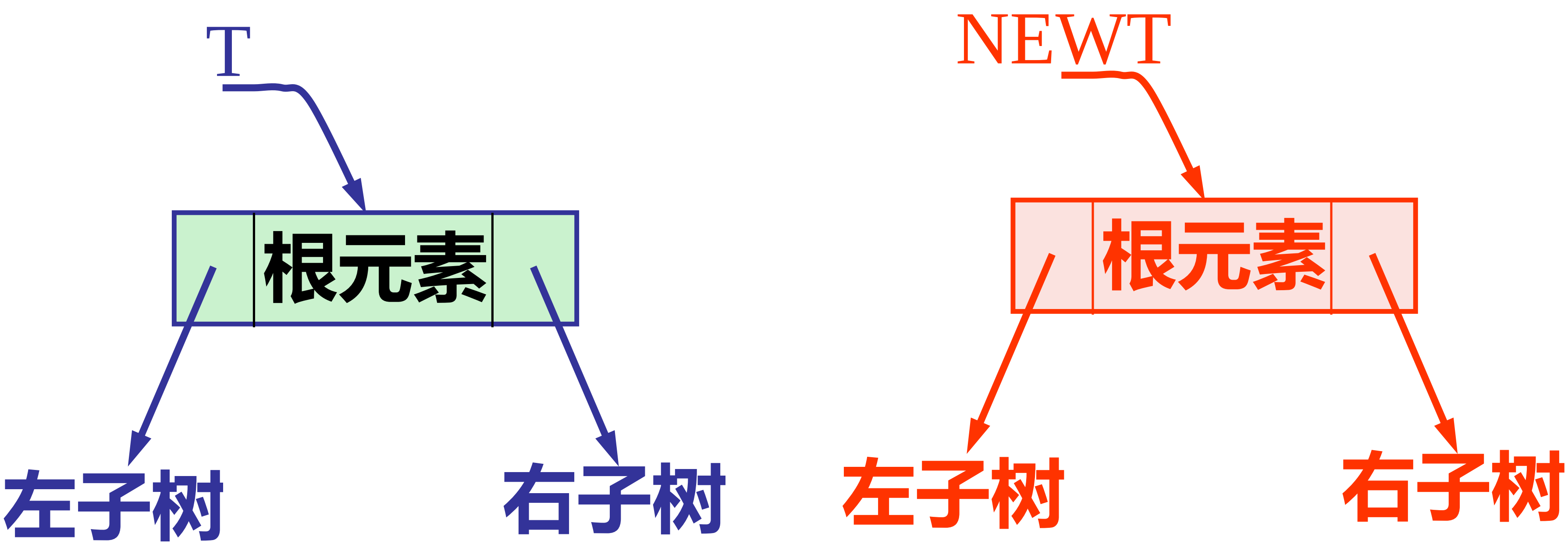
```
int    _count;//类定义之外定义的一个整型变量

Public:
    …….
    int Size ( )
    { _count = 0;
    PreOrder(Add1， root);
    return _count;}
private：
    ……
    static void Add1(binaryTreeNode<T> *t) { _count++;}
```

统计叶子节点数（Size）

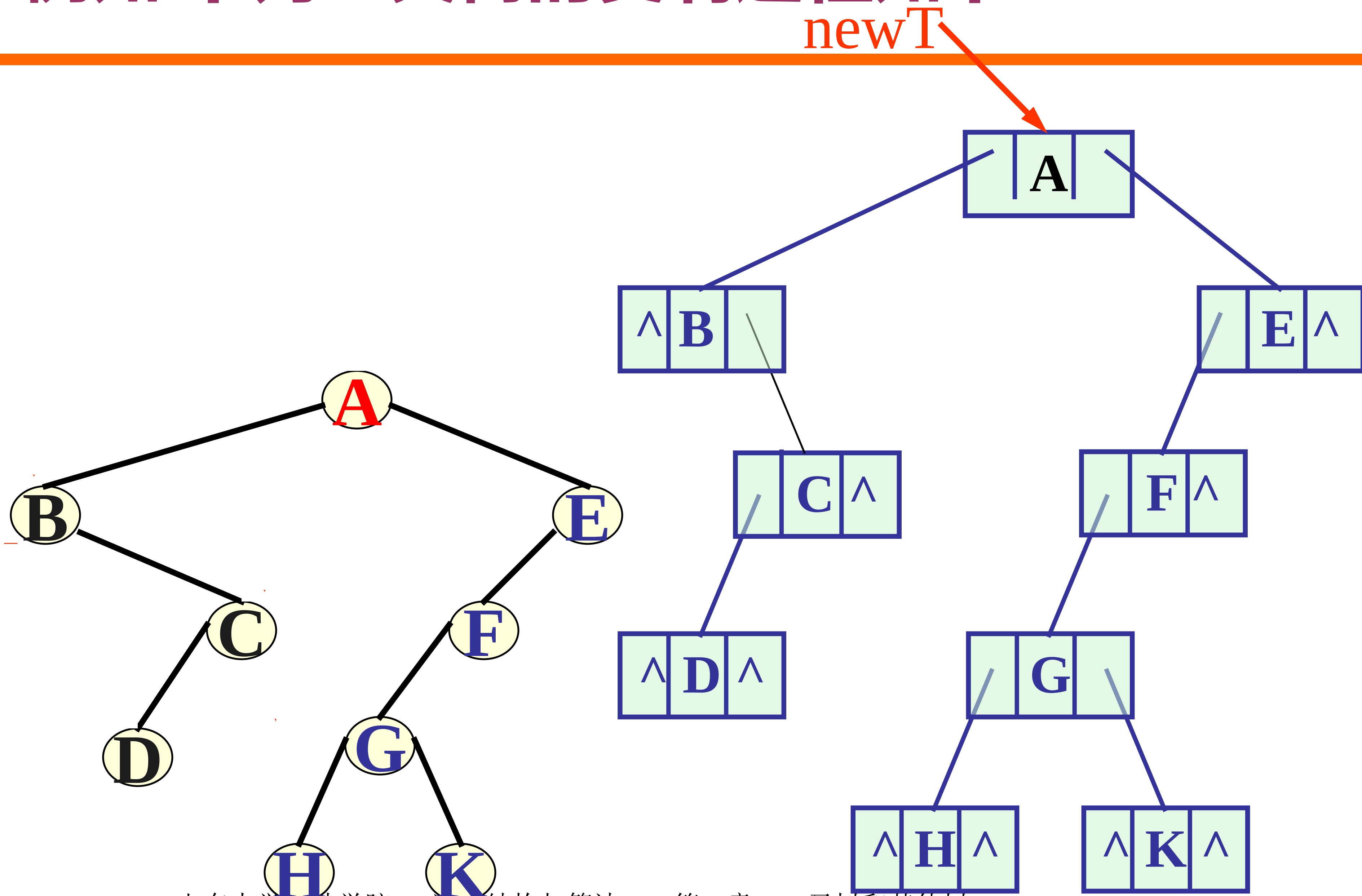
```
int   _count;//类定义之外定义的一个整型变量

Public:
    …….
    int Size ( )
    { _count = 0;
    PreOrder(Add2， root);
    return _count;}
private：
    ……
    static void Add2(binaryTreeNode<T> *t) {
    if (!t-> LeftChild && !t-> RightChild) _count++;}
```

复制二叉树

其基本操作为:生成一个结点。



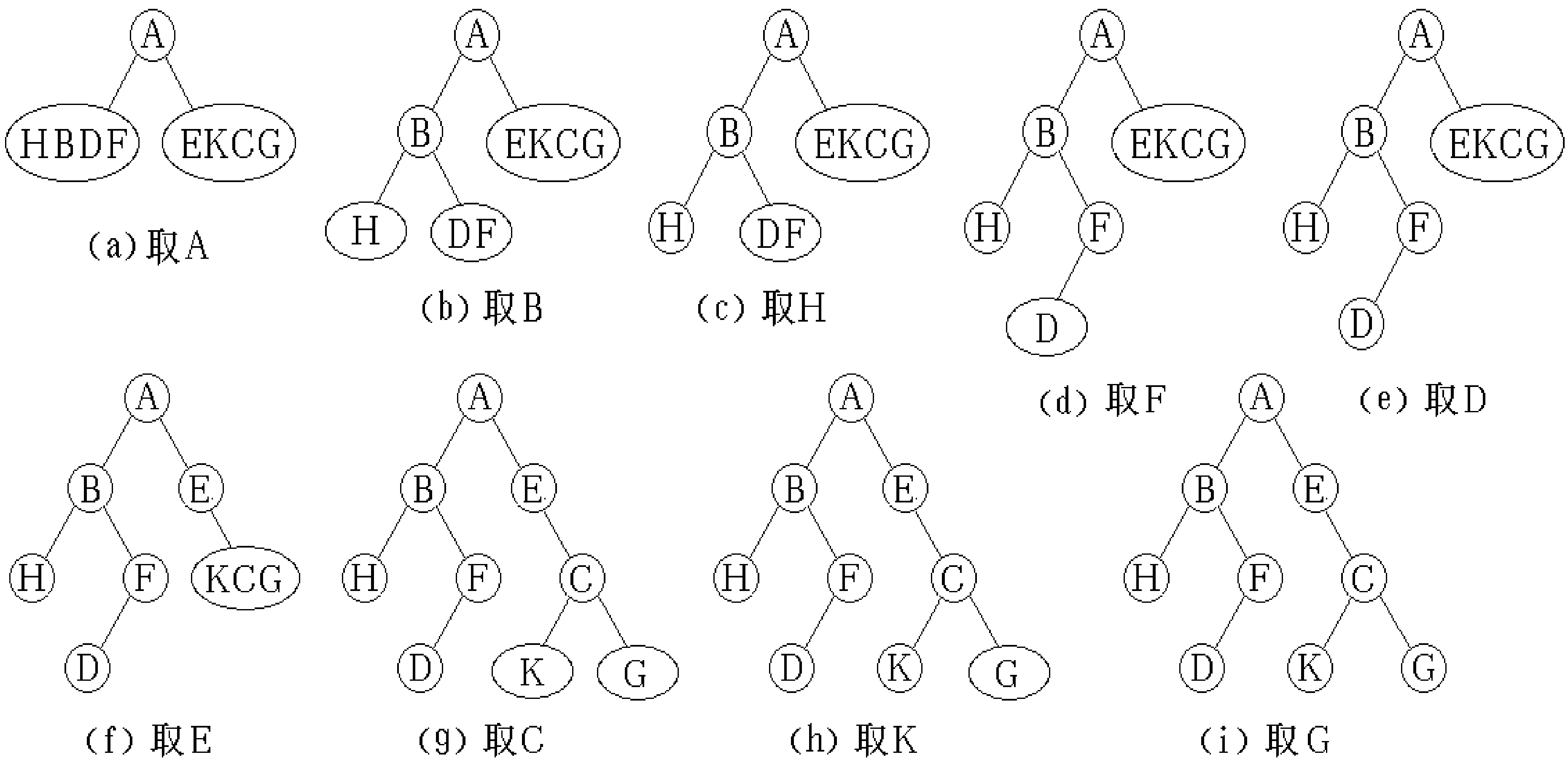
例如:下列二叉树的复制过程如下:



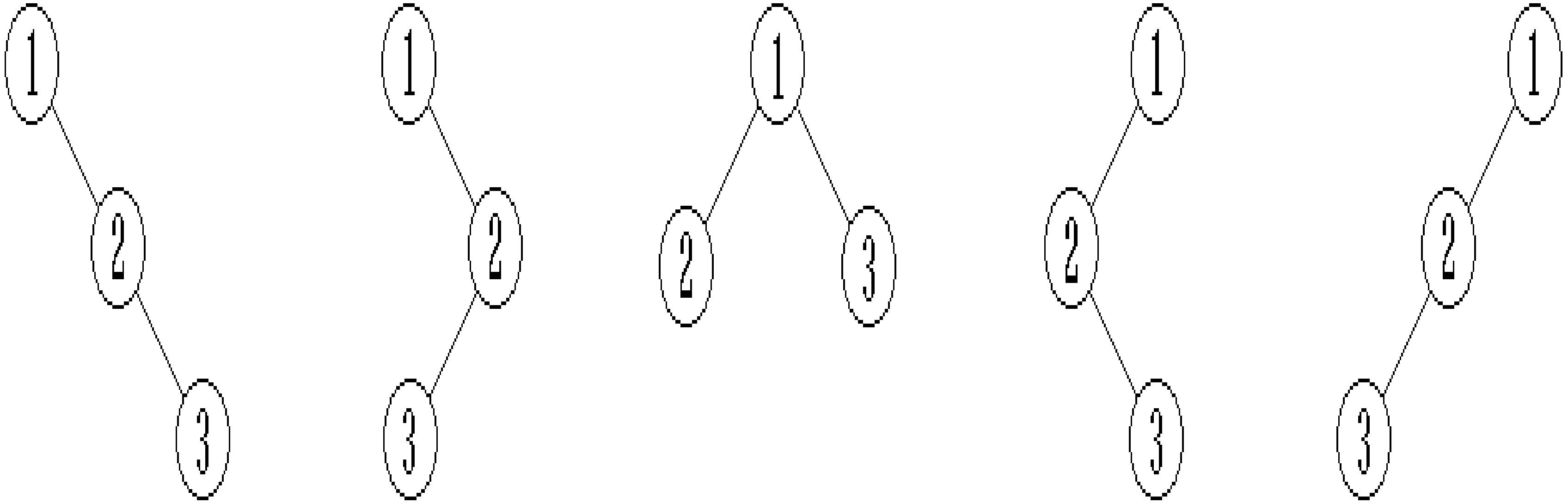
思考

二叉树的删除算法？

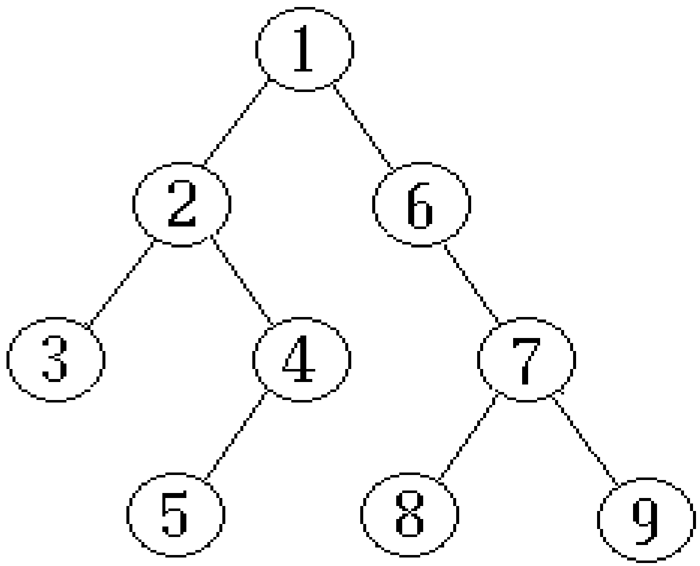
由二叉树的前序序列和中序序列可唯一地确定一棵二叉树。例，前序序列｛ ABHFDECKG ｝和中序序列｛ HBDFAEKCG ｝，构造二叉树过程如下：



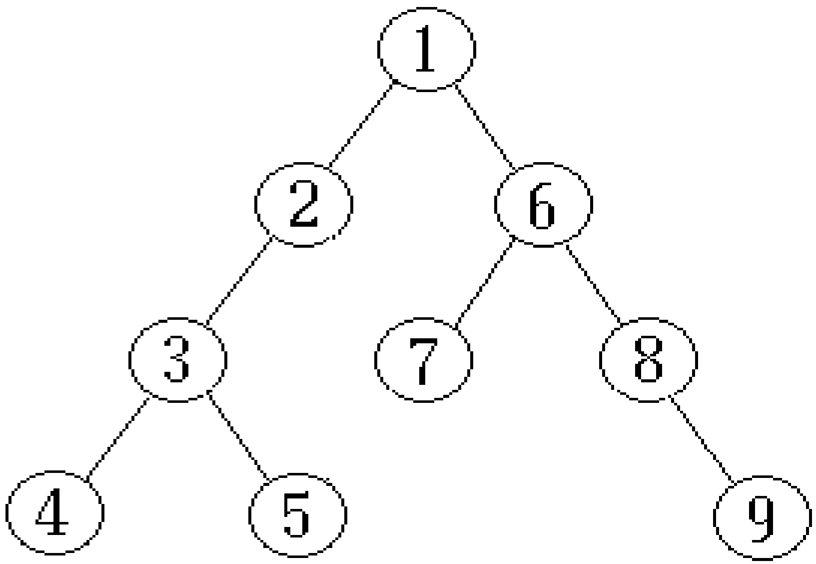
例如，有 3 个数据 { 1, 2, 3 }，可得5种不同的二叉树。它们的前序排列均为 123，中序序列可能是 123，132，213，231，321。



如果前序序列固定不变，给出不同的中序序列，可得到不同的二叉树。



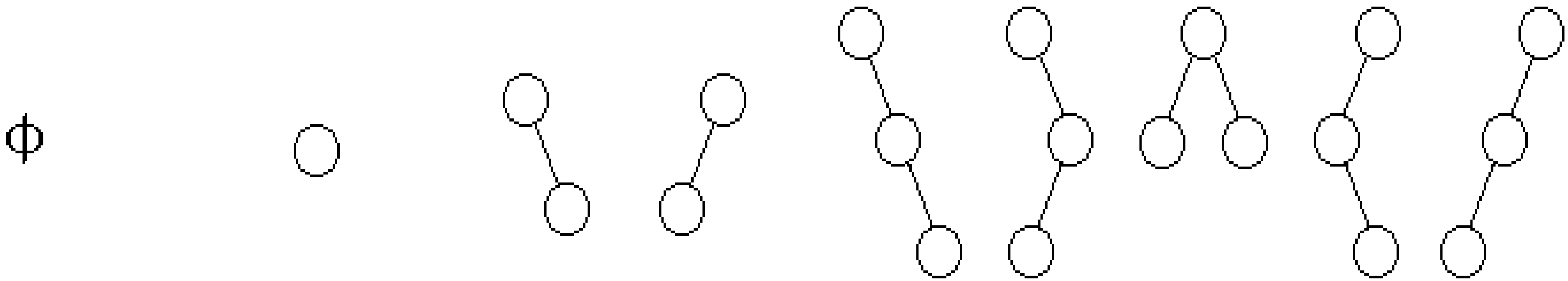
(a)



(b)

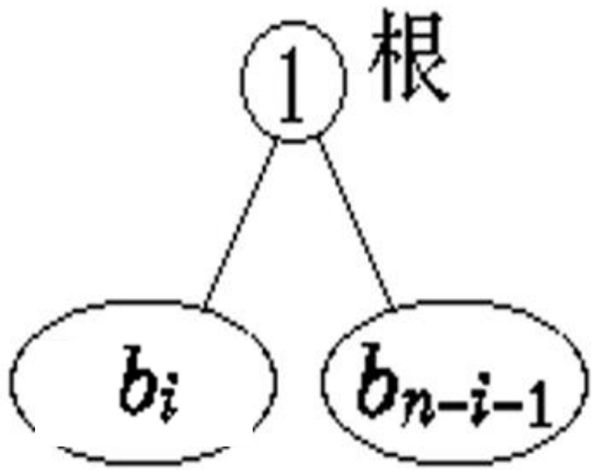
问题是有 n 个数据值，可能构造多少种不同的二叉树？我们可以固定前序排列，选择所有可能的中序排列。

有0个，1个，2个，3个结点的不同二叉树如下



(a) $b_0=1$　(b) $b_1=1$　(c) $b_2=2$　(d) $b_3=5$

有n个结点的不同二叉树如下



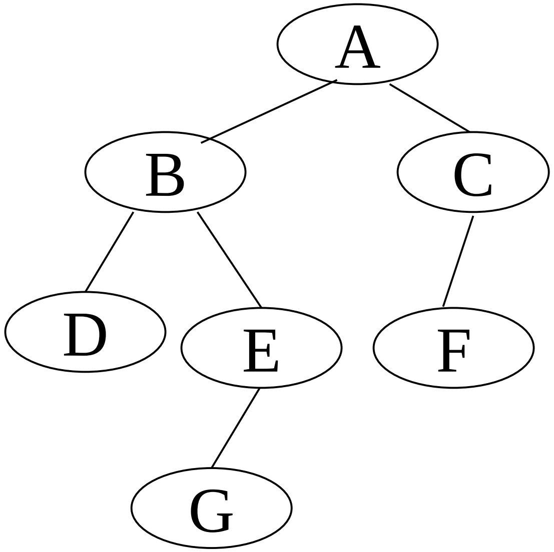
$$(e)\ b_n = \sum_{i=0}^{n-1} b_i \cdot b_{n-i-1}$$

$$b_n = b_0 \cdot b_{n-1} + b_1 \cdot b_{n-2} + b_2 \cdot b_{n-3} + \ldots + b_{n-1} \cdot b_0$$

结论

- 若二叉树中各节点的值均不相同
  - 由二叉树的前序序列和中序序列
  - 或由其后序序列和中序序列均能唯一地确定一棵二叉树
  - 但由前序序列和后序序列却不一定能唯一地确定一棵二叉树

练习



前序：ABDEGCF

中序：DBGEAFC

后序：DGEBFCA

逐层：ABCDEFG

练习

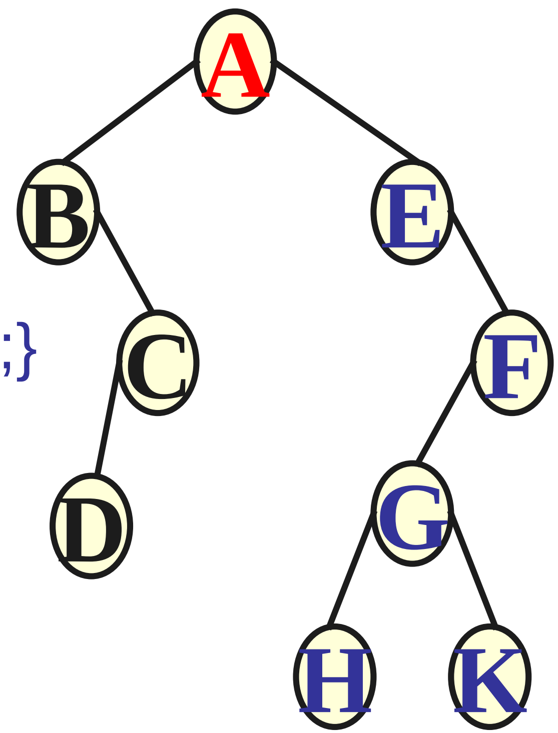
- 已知一棵二叉树的中序、后序序列分别如下：
- 中序：D C E F B H G A K J L I M
- 后序：D F E C H G B K L J M I A
- 要求：
- (1) 画出该二叉树；
- (2) 写出该二叉树的先序序列。

练习

- 前序遍历的非递归实现。
- 后序遍历的非递归实现。

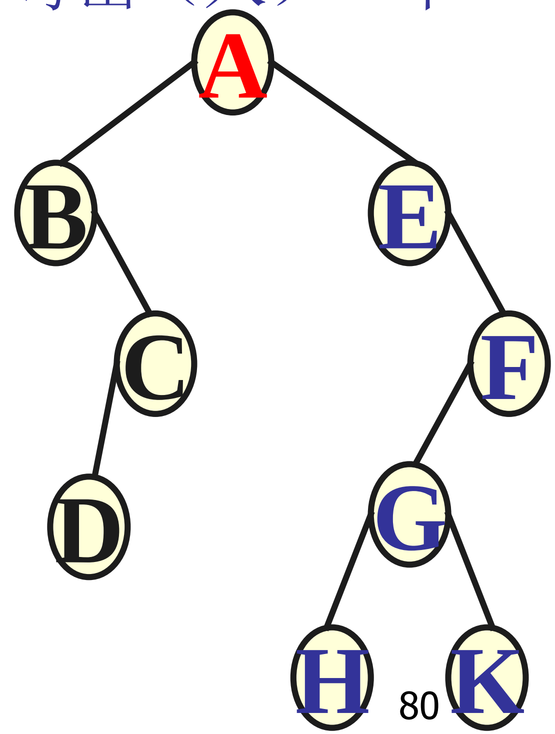
前序遍历的非递归实现

```
template<class T>
void PreOrder(binaryTreeNode<T> *t)     //对*t进行前序遍历
{   stack  <binaryTreeNode<T> *>    s(MaxLength);
    binaryTreeNode<T> *p=t ;
   do {
        while (p)
            {Visit(p);
             s.push(p);    p=p->LeftChild;}
        if (!s.IsEmpty())
             { p=s.top();
               s.pop();
               p=p->RightChild;
             }
   } while (p||!s.IsEmpty())
}
```



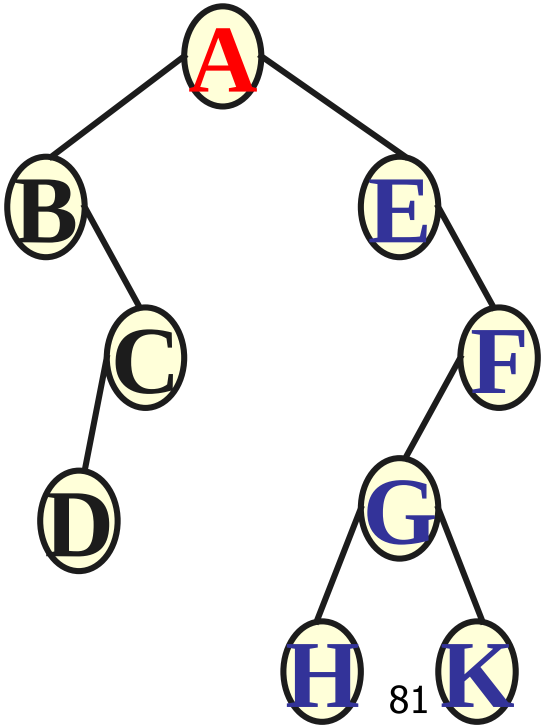
后序遍历的非递归实现

```
template<class T>
void PostOrder(binaryTreeNode<T> *t)    //对*t进行中序遍历
{   stack  <binaryTreeNode<T> *>   s(MaxLength);
     int flag;
     stack<int> flagStack; //存放标志位的栈，每出（入）一个
    节点指针，同步出（入）//一个标志位
    binaryTreeNode<T> *p=t；
   do {
         while (p)
             {s.push(p); flagStack. push(1);
              p=p->LeftChild;}
          if (!s.IsEmpty())
               {p=s.top(); flag=flagStack.top();
```



后序遍历的非递归实现

```
 if(p->RightChild&&flag==1){
                 flagStack.pop();
                 flagStack.push(2);
                 p=p->RightChild;
                 }
              else{ s.pop();
                     flagStack.pop();
                     Visit(p);
                      p=0;
                 }
                           }
     } while (p||!s.IsEmpty())
}
```



练习

- 设t是数据域类型为int 的二叉树，每个节点的数据都不相同。根据数据域的前序和中序排列构造二叉树。
- 指出函数的时间复杂性。

通过前序和中序构造二叉树

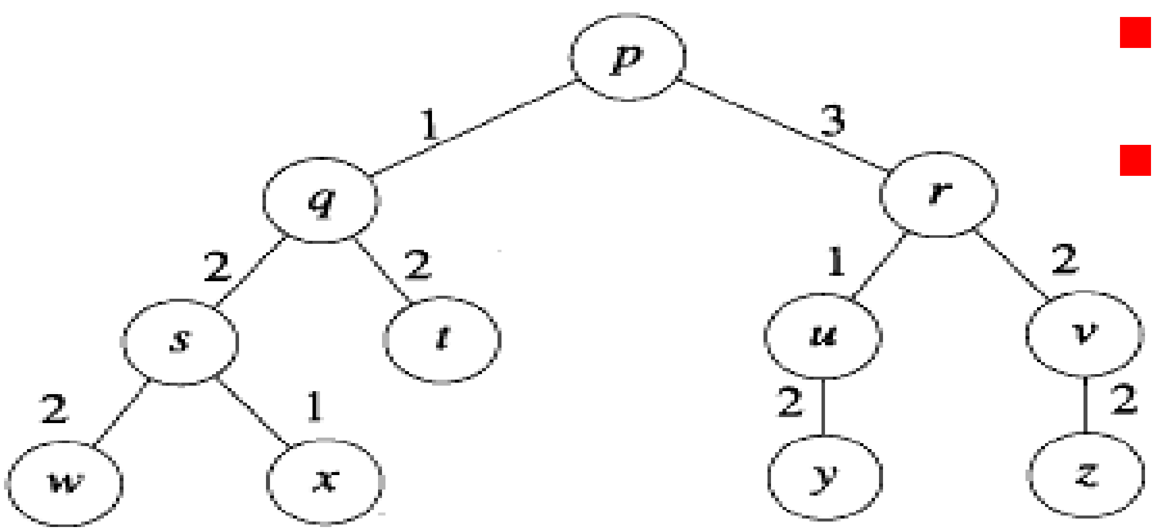
```
Template <class T>
BinaryTreeNode<T> *CreateBinary(T *Prelist, T *Inlist, int n) {
// Prelist是二叉树的前序序列, Inlist是二叉树的中序序列,函数返
   回二叉树根指针
  If （n= =0） return NULL;
  Int k=0;
  While(Prelist[0]!=Inlist[k]) k++;
  BinaryTreeNode<T> *t=new BinaryTreeNode<T> (Prelist[0]);
  t->LeftChild= CreateBinary(Prelist+1, Inlist, k);
  //从前序Prelist+1开始对中序的0~k-1左子序列的k个元素递归建立左子树
  t->RightChild= CreateBinary(Prelist+k+1, Inlist+k+1, n-k-1);
//从前序Prelist+k+1开始对中序的k+1~n-1右子序列的n-k-1个元素递归建立右
   子树
  return t;
}
```

11.9 应用

- 11.9.1 设置信号放大器
- 11.9.2 在线等价类

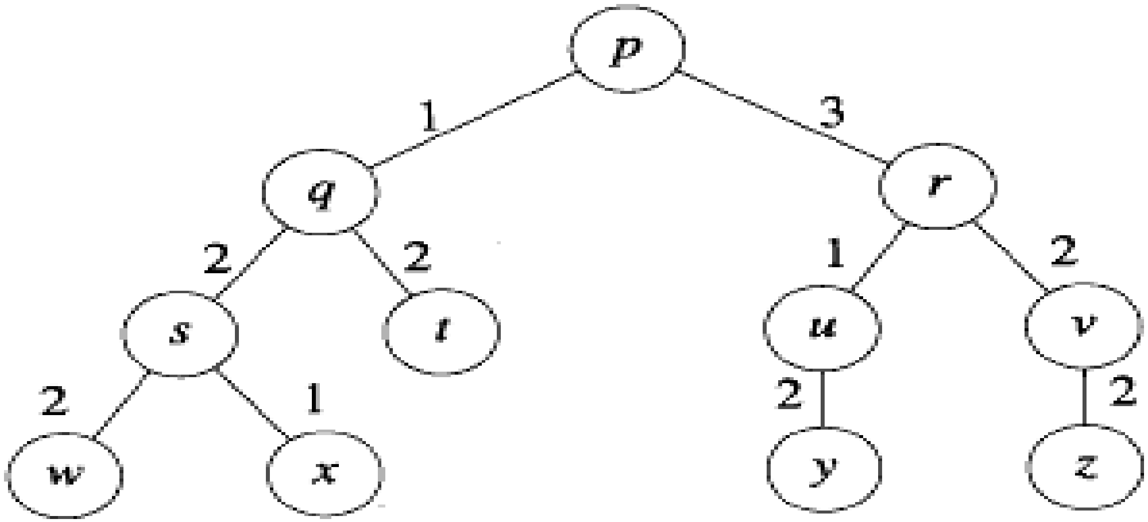
11.9.1 设置信号放大器

- 为简化问题，设分布网络是一树形结构
  - 源是树的根。
  - 每一节点(除了根)表示一个可以用来放置放大器的子节点。
  - 信号从一个节点流向其子节点。
  - 每条边上标出从父节点到子节点的信号衰减量。



- 从节点p 流到节点v 的衰减量为?。
- 从节点q 到节点x 的衰减量为?

11.9.1 设置信号放大器



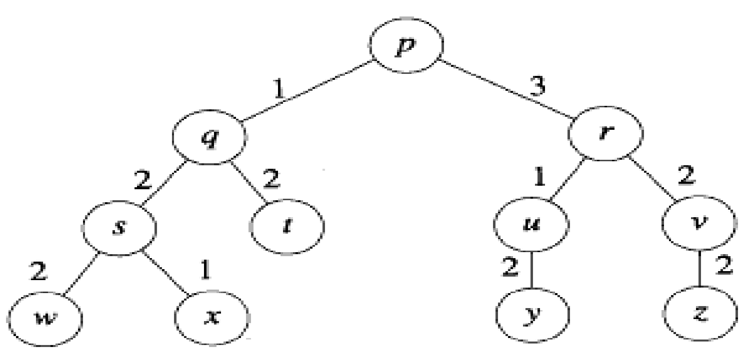
- 设计一个算法确定把信号放大器放在何处。
- 目标是要使所用的放大器数目最少并且保证信号衰减(与源端信号相关)不超过给定的容忍值。

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* (*j*) }

    j 是 i的孩子



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)

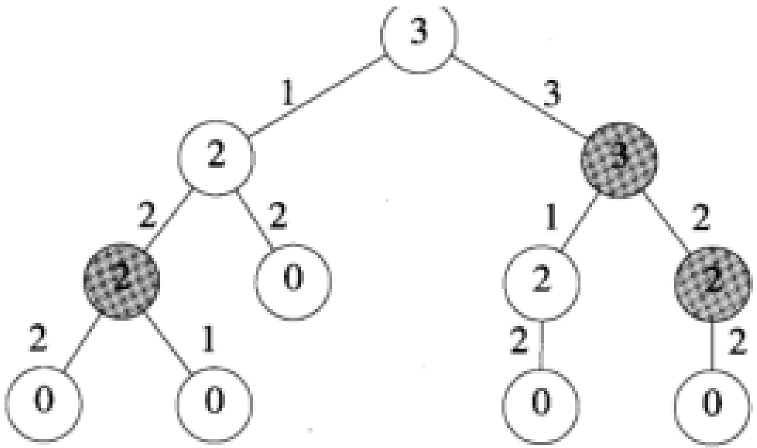
{在j 放置放大器;

degradeToLeaf(i) =

max{degradeToLeaf(i), degradeFromParent(j)};

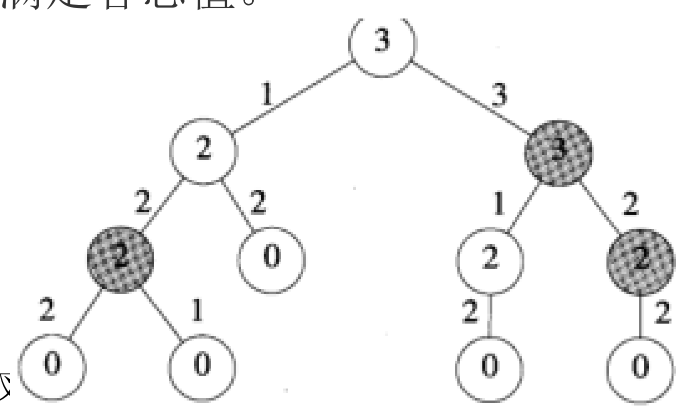
else degradeToLeaf(i) = max{degradeToLeaf(i),

degradeToLeaf(j) +degradeFromParent(j)};



证明：该方法所放置的放大器最少

可通过对树的节点数n 进行归纳来证明。
当n=1时，定理显然成立。
设n≤m 时，定理成立，其中m 为任意的自然数。
设t为有n+1个节点的树。令X为由算法所确定的放置放大器的节点的集合，W为满足容忍值限制的拥有最少放大器的节点集合，因此只需证X = W。
若X = 0,则X = W 。
若X>0，则设z为由算法给出的放置第一个放大器的节点,t(z)为树t中以z为根的子树。因为degradeToLeaf (z)+degradeFromParent (z)>容忍值,W至少需包含t(z)中的某个节点u。
若W中还包含了 u 以外的元素，则W一定不是最好的方案，因为通过在W－{所有的象u 这样的节点} + {z}集合中放置放大器也能满足容忍值。因此W中只包含节点u。
设W'=W－{u}，t'为从树t中除去子树t(z)（但保留z）而得到的树,则对于t'而言，W'为放大器数目最少的方案。而且,X'=X－{z}在树t'也满足容忍值。
因t'中的节点数小于m+1，X′ = W′。
因此 X = X′+1 = W′+1= W 。
证毕！



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:
    - {1, 2, 7, 8, 11, 12}
    - {3, 4, 5, 9, 13, 14}
    - {6, 10}

回顾第三章 第六种解决方案

```
void   Initialize(int n)
{//初始化n个类，每个类仅有一个元素
  E=new  int [n+1];
  For (int  e=1; e<=n; e++)
      E[e]=e;
}
void   Union(int i, int  j)
{//合并类i和类j
For (int k=1; k<=n; k++)
    if   (E[k]= =j)     E[k]=i;
}
int    Find(int  e)
{return  E[e]; //搜索包含元素i的类
}
```

E[] 1	2	3	4	5	…		n
1	2	3	4	5			

E[] 1	2	3	4	5	…		n
1	3	3	4	5			

回顾第六章 第二种解决方案

- 针对每个等价类设立一个相应的链表
- node[1:n]用于描述n个元素（每个元素都有一个对应的等价类链表）
- node[e]：EquivNode类私有数据成员（int）：
  - E：元素e所在的等价类(用等价类链表中的首节点位置表示)
  - Size：元素e所在等价类中的元素数目（等价类链表中的节点数，仅当e是链表的首节点时，才定义node[e].size ）
  - Link：链表指针（模拟指针）， 0表示空指针。

```
void Initialize(int n)
{// 初始化n个类，每个类仅有一个元素
    node = new EquivNode [n+1];
    for (int e = 1; e <= n; e++) {
        node[e].E = e;
        node[e].link = 0;
        node[e].size = 1;
    }
}
```

E[] 1 2 3 4 5 n

E	1	2	3	4	5					
LINK	0	0	0	0	0					
SIZE	1	1	1	1	1					

```
int Find(int e)
{ //搜索包含元素i 的类
return node[e].E;
}
```

E[]　1　2　3　4　5　n

E	1	2	3	4	5					
LINK	0	0	0	0	0					
SIZE	1	1	1	1	1					

```
void Union(int i, int j)
{ //合并类i 和类j;
 // 使i 代表较小的类
    if (node[i].size > node[j].size)
        swap ( i , j ) ;
    //改变较小类的E值
    int k;
    for (k = i; node[k].link; k = node[k].link)
        node[k].E = j;
    node[k].E = j; // 链尾节点
    //在链表j的首节点之后插入链表i; 并修改新链表的大小
    node[j].size += node[i].size;
    node[k].link = node[j].link;
    node[j].link = i;
}
```

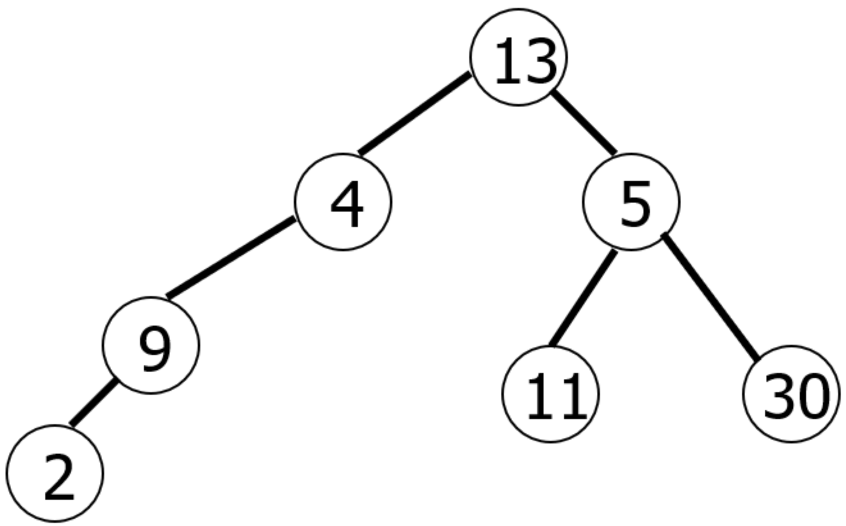
E[] 1 2 3 4 5

E	1	2	2	4	5		
LINK	0	3	0	0	0		
SIZE	1	2	1	1	1		

E	1	2	2	2	5		
LINK	0	4	0	3	0		
SIZE	1	3	1	1	1		

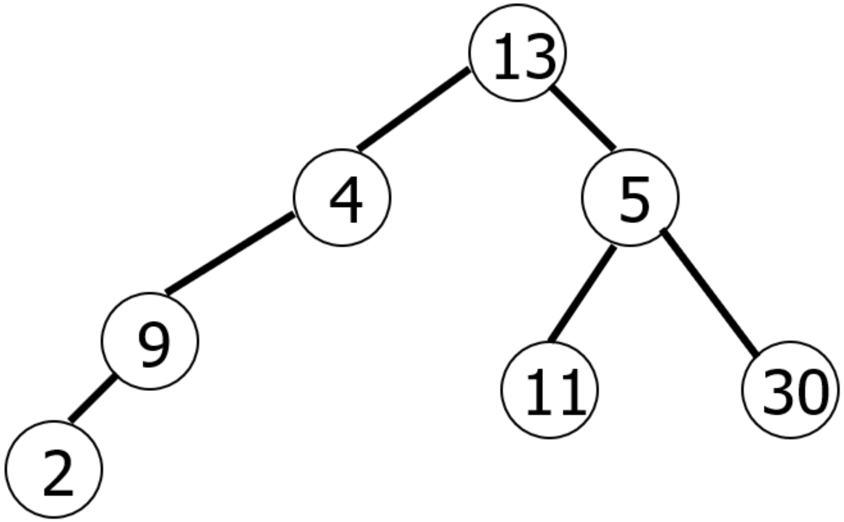
集合(类)的树形描述

- 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

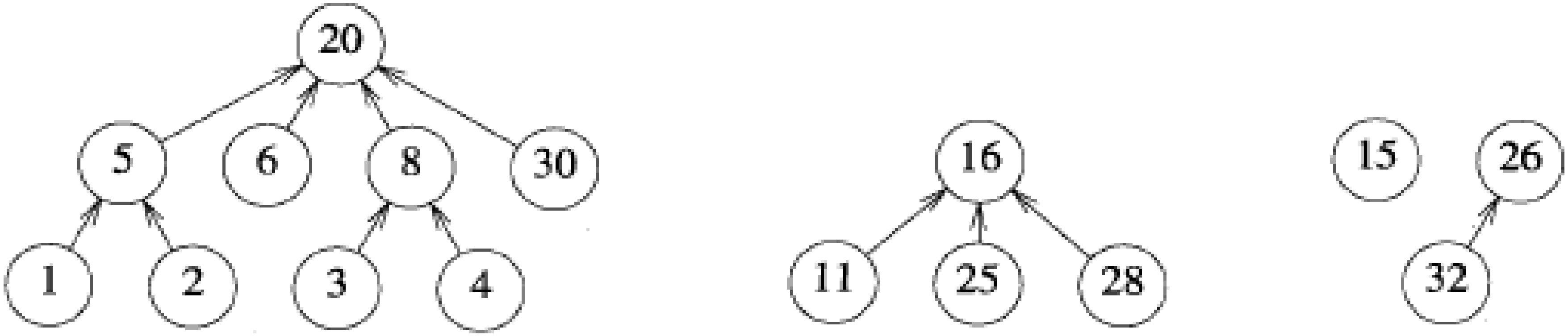
- find(i)： 返回i所在树的根元素.



- 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.

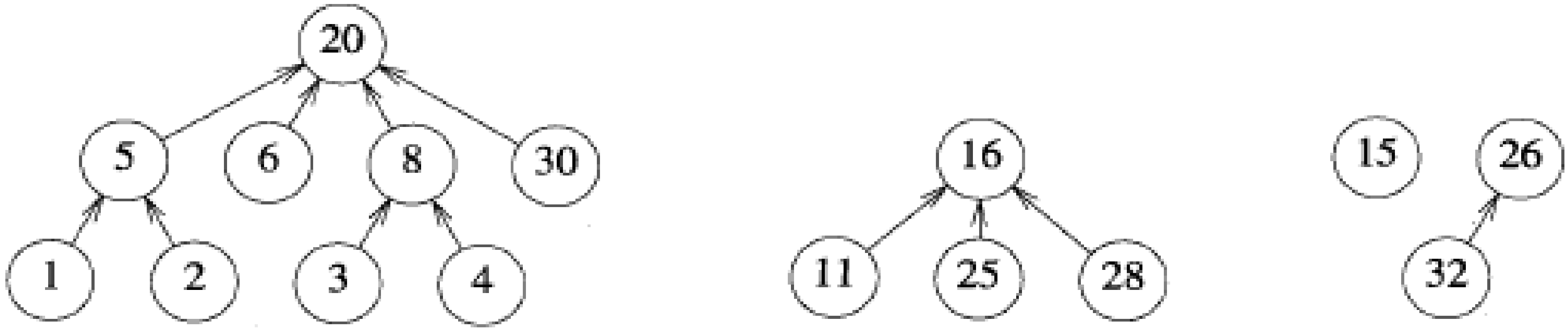
树的描述

- 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}



a) b) c)

树的描述



a) b) c)

- 使用模拟指针。
- 每个节点有一个parent域。
- 每个parent 域给出父节点的索引。

树的描述

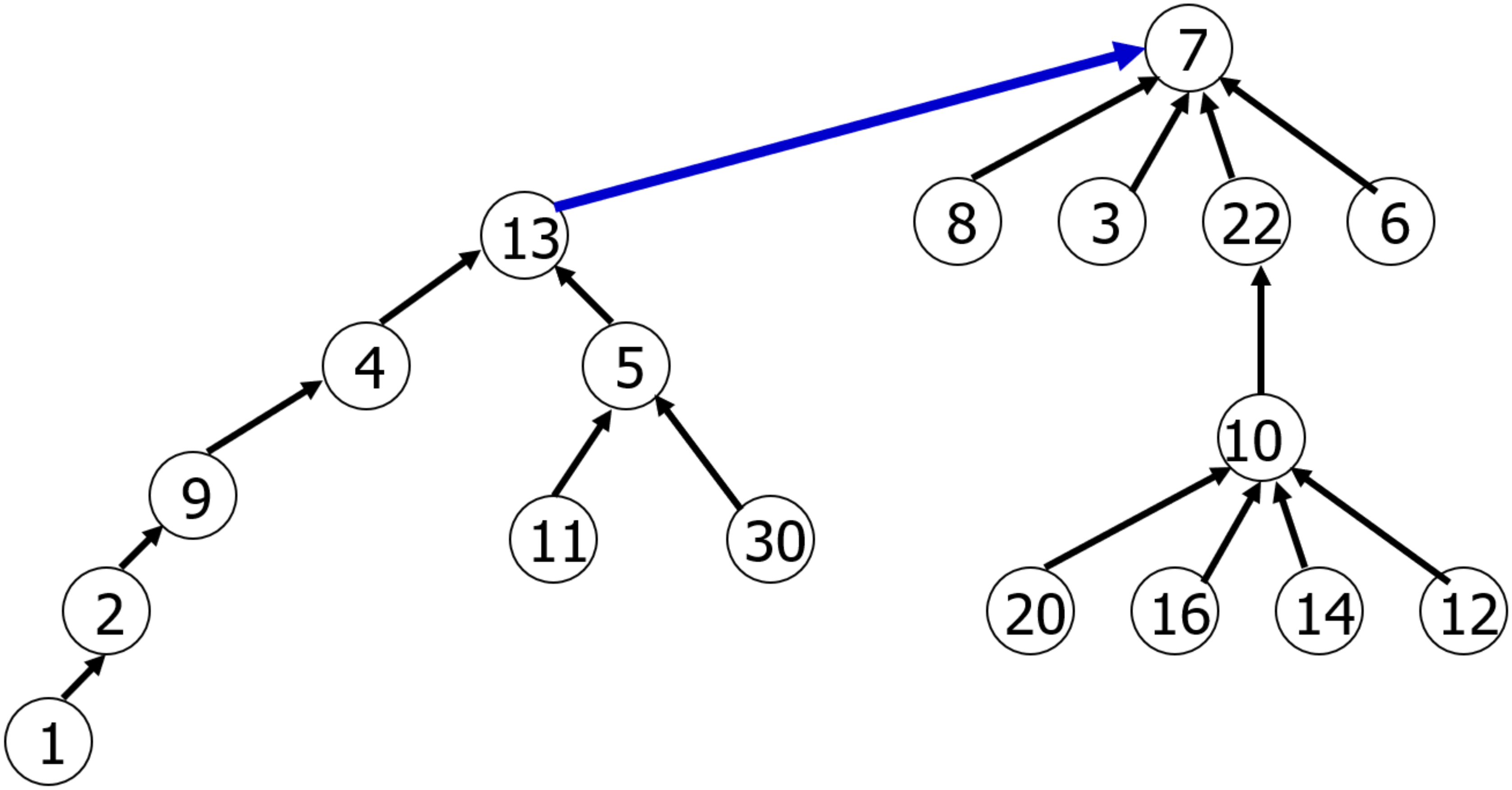
```
int    *parent;
```

Parent:

5	5	8	8	20	20	…….			
1	2	3	4	5	6	…….	e………		n

Unite 操作

- union(7,13)



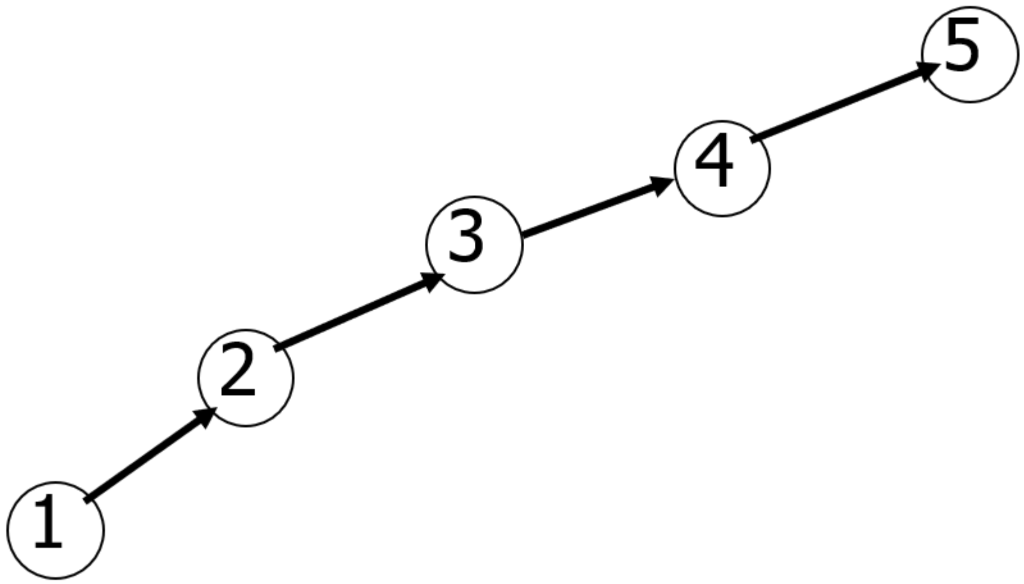
```
void initialize(int numberOfElements)
{// 初始化，每个类/树有一个元素
    parent = new int[numberOfElements+1];
    for (int e = 1; e <= numberOfElements; e++)
    parent[e] = 0;
}
int find(int theElement)
{ / /返回包含theElement的树的根节点
    while (parent[theElement]!=0)
        theElement = parent[theElement]; // 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$\Theta(1)$。

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

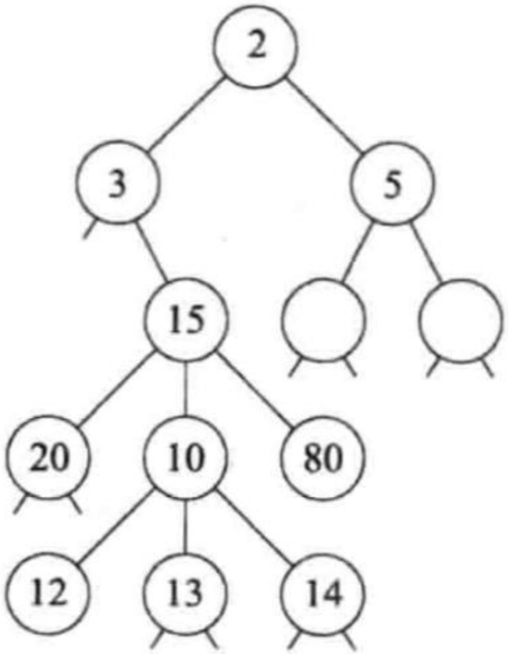
- 每一次查找需花费Θ(q)时间
  - q是在执行查找之前所进行的合并操作的次数
- 要执行一个系列操作(u 次合并和f 次查找)的时间为O（fu）

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

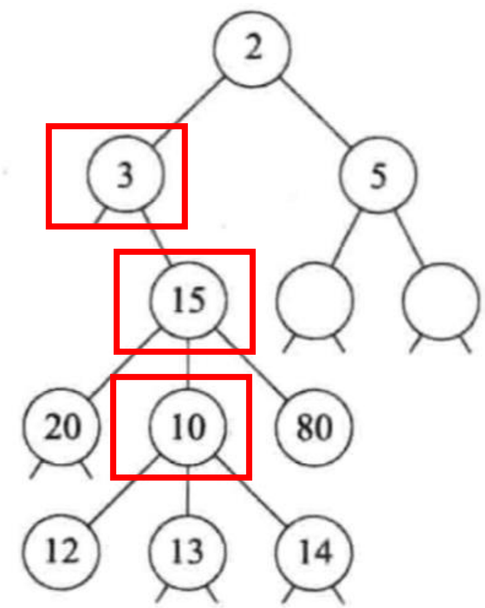
- 路径的缩短可以通过称为路径压缩(path compression) 的过程实现。

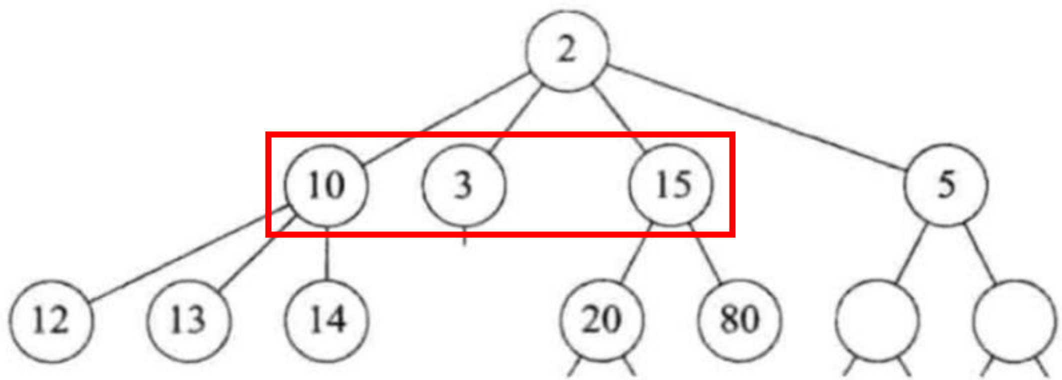


提高在最坏情况下的性能的方法

- 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。

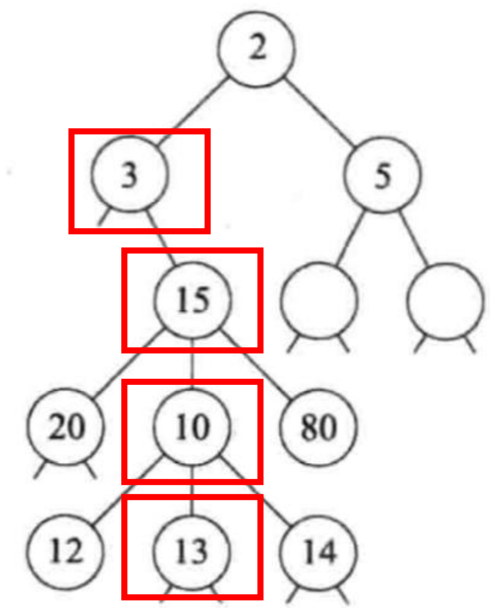


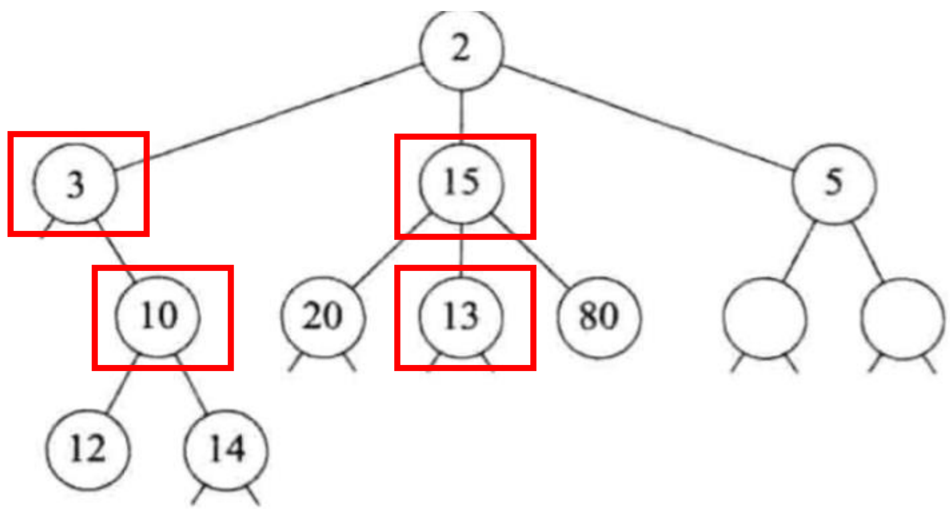


提高在最坏情况下的性能的方法

- 路径的缩短可以通过称为路径压缩(path compression) 的过程实现。

1. 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。

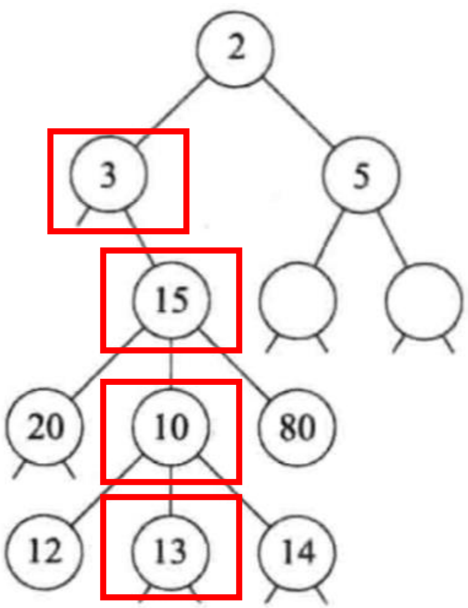


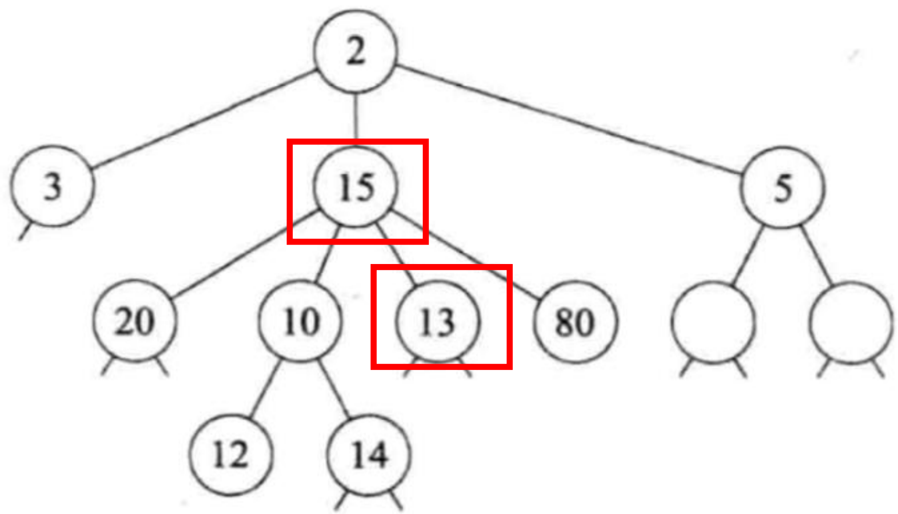


提高在最坏情况下的性能的方法

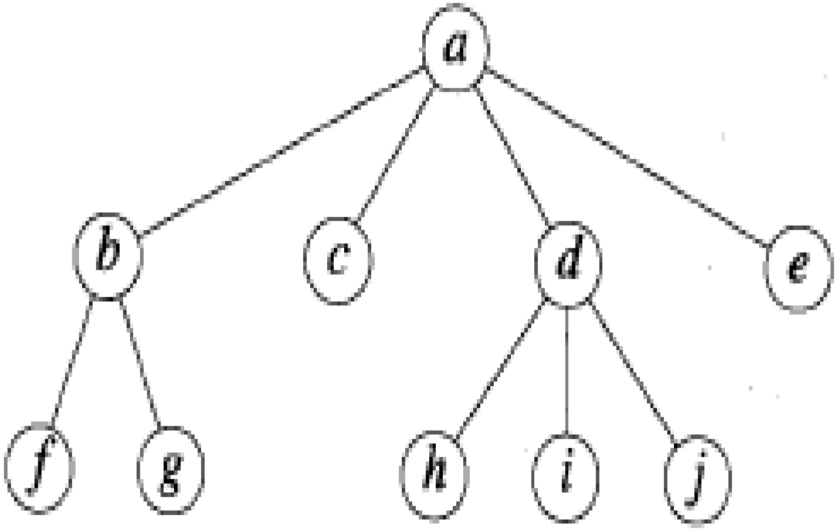
- 路径的缩短可以通过称为路径压缩(path compression) 的过程实现。

1. 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。

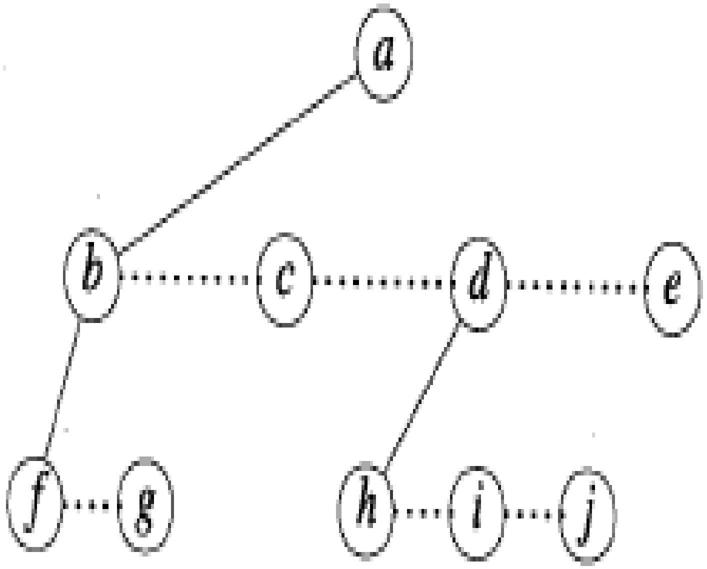




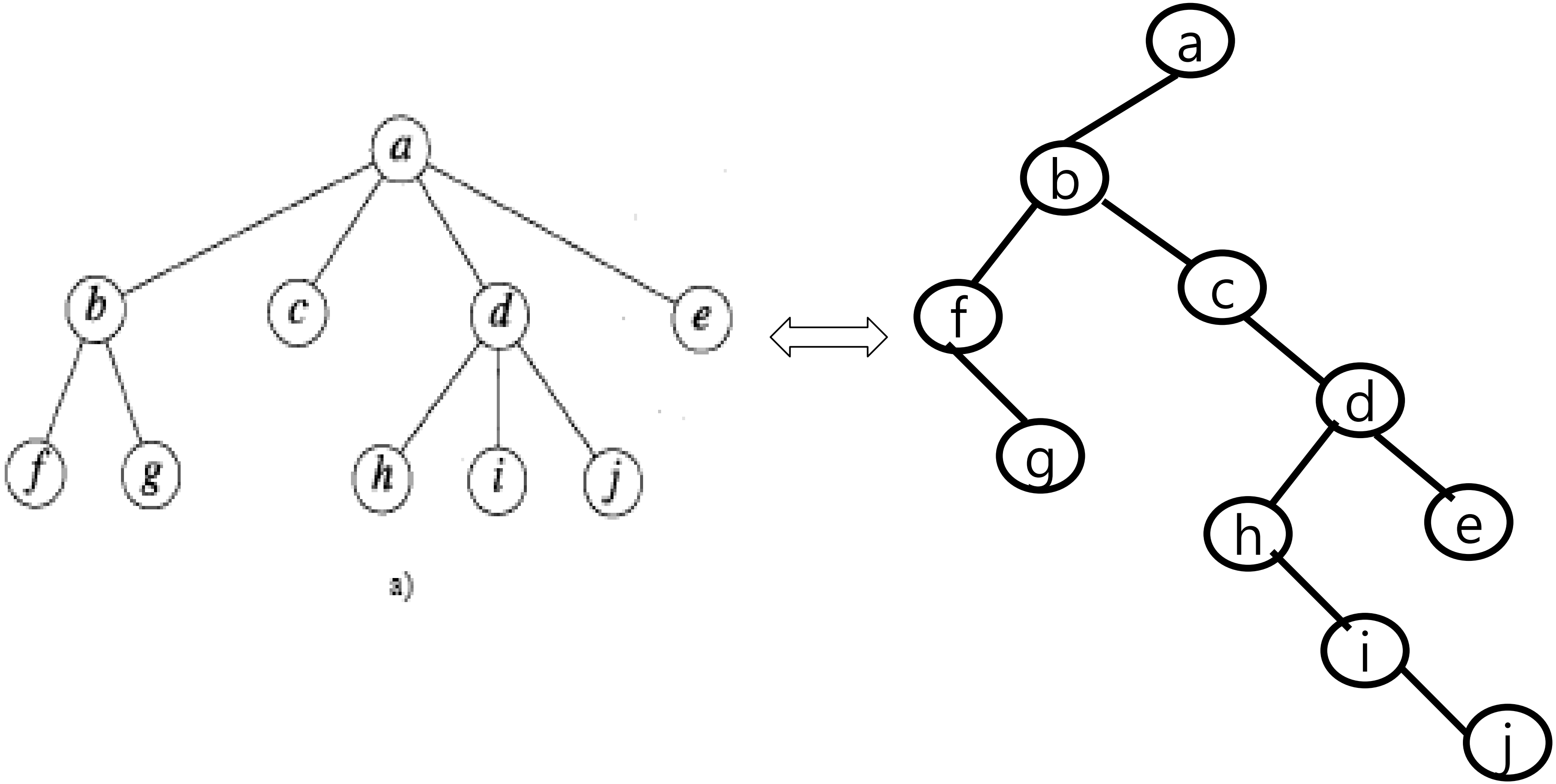
树的二叉树描述



- 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。

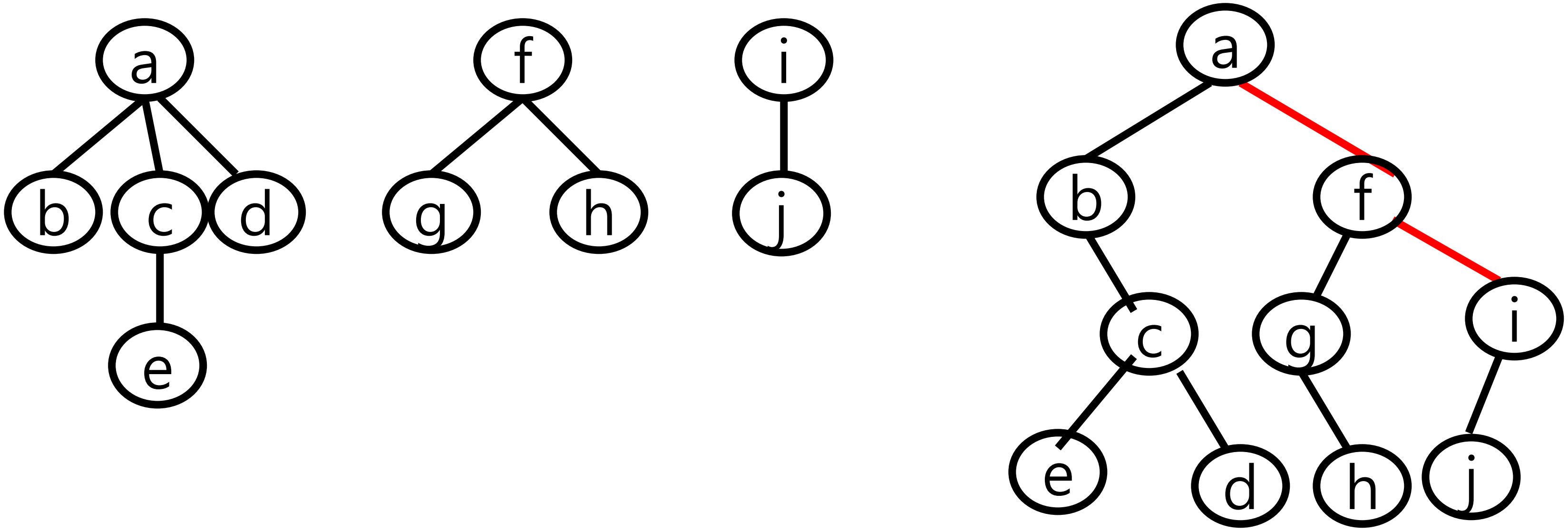


树的二叉树描述



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
  - 首先得到树林中每棵树(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≤i≤m).



树的遍历

- 对树的遍历操作是指按照某种次序系统地访问树中的每个子树，且对每个结点只访问一次。
- 遍历树的方法主要有深度优先遍历和广度优先遍历。

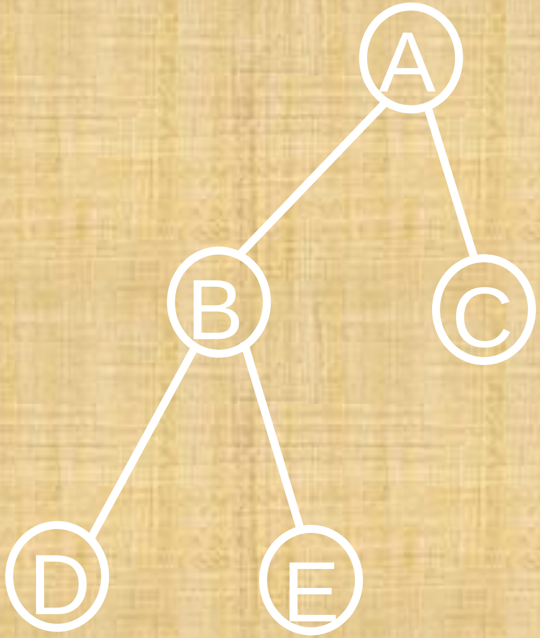
树的深度优先遍历

① 先根遍历

- ♣ 访问树的根结点
- ♣ 从左到右依次先根遍历根的各子树

对下图所示的树,按先根次序遍历后得到的结点序列为:

A B D E C



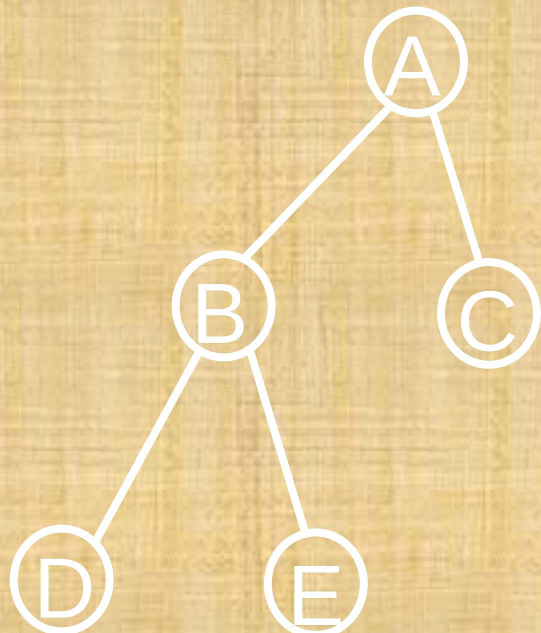
树的深度优先遍历

② 后根遍历

- ♣ 从左到右依次先根遍历根的各子树
- ♣ 访问树的根结点

对下图所示的树,按后根次序遍历后得到的结点序列为:

D E B C A

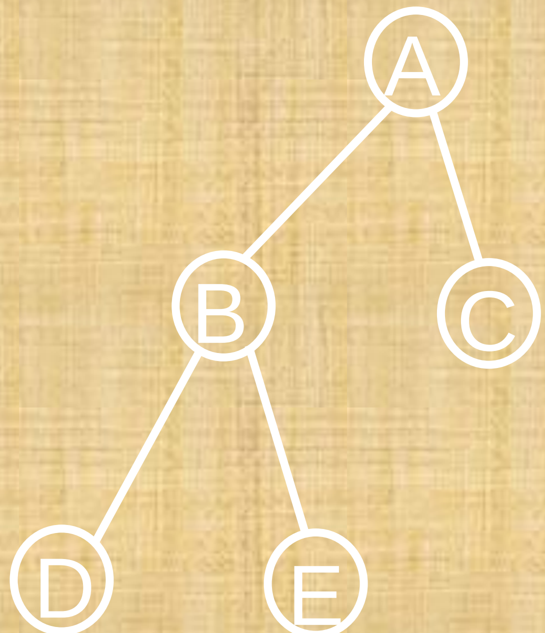


树的广度优先遍历

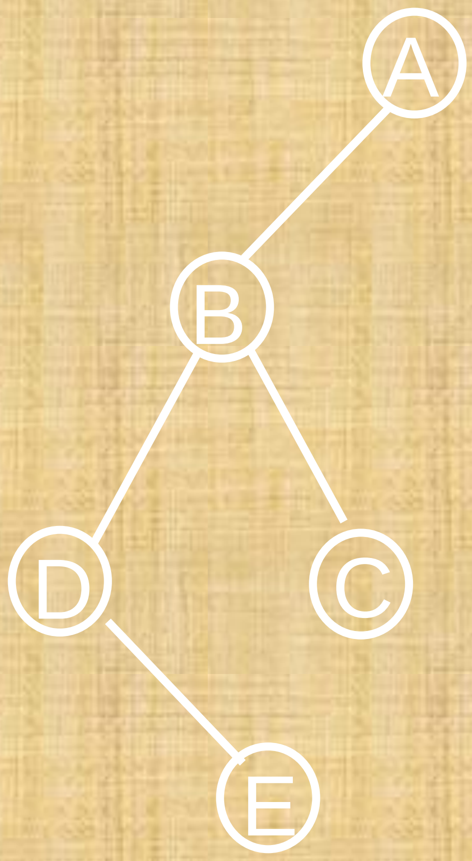
是一种按照层次遍历树的方法，具体的遍历过程是：首先从左到右访问层数为0的所有结点，然后再从左到右访问层数为1的所有结点，. . . ，直到访问完其余各层所有的结点。

对下图所示的树,按广度优先遍历后得到的结点序列为:

A B C D E



对下图所示的树,转化成二叉树并遍历



前序 A B D E C

中序 D E B C A

后序 E D C B A

层次 A B D C E

森林的遍历

- 对森林的遍历操作是指按照某种次序系统地访问树林中的每个结点，且对每个结点只访问一次。
- 遍历森林的方法主要有深度优先遍历和广度优先遍历。

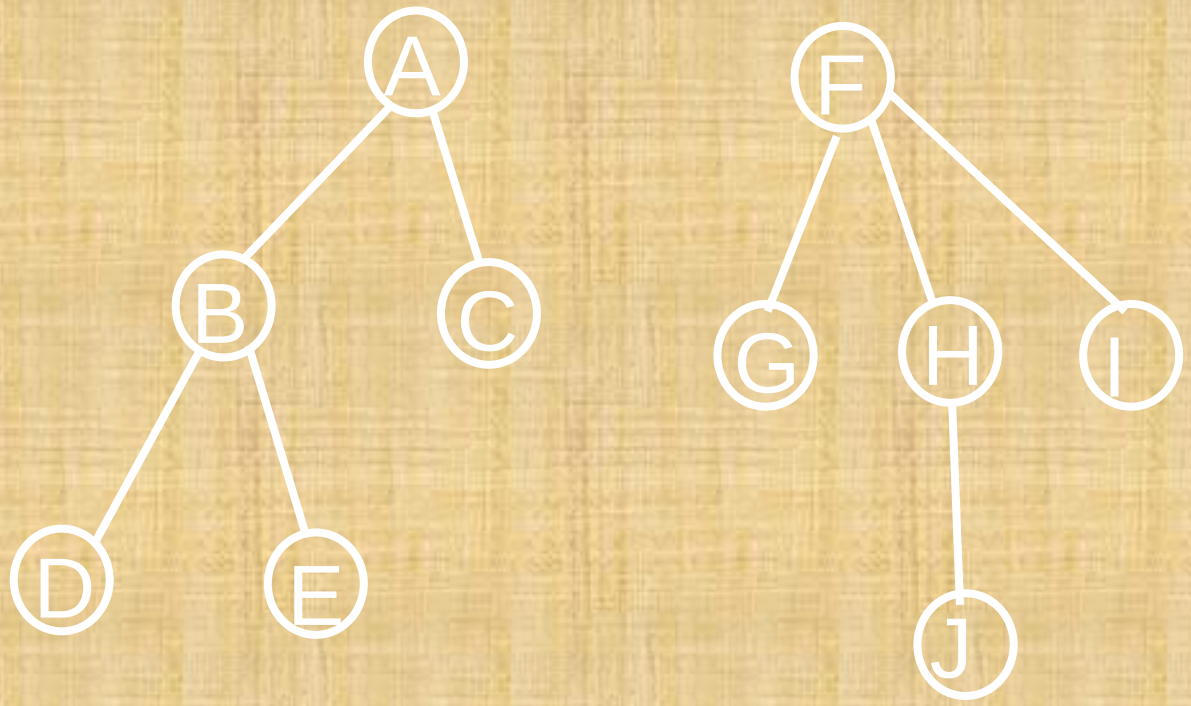
森林的深度优先遍历

① 先根遍历

- ♣ 访问头一棵树的根结点
- ♣ 从左到右先根遍历头一棵树树根的各子树
- ♣ 先根遍历其它的树

对下图所示的森林,按先根次序遍历后得到的结点序列为：

A B D E C F G H J I



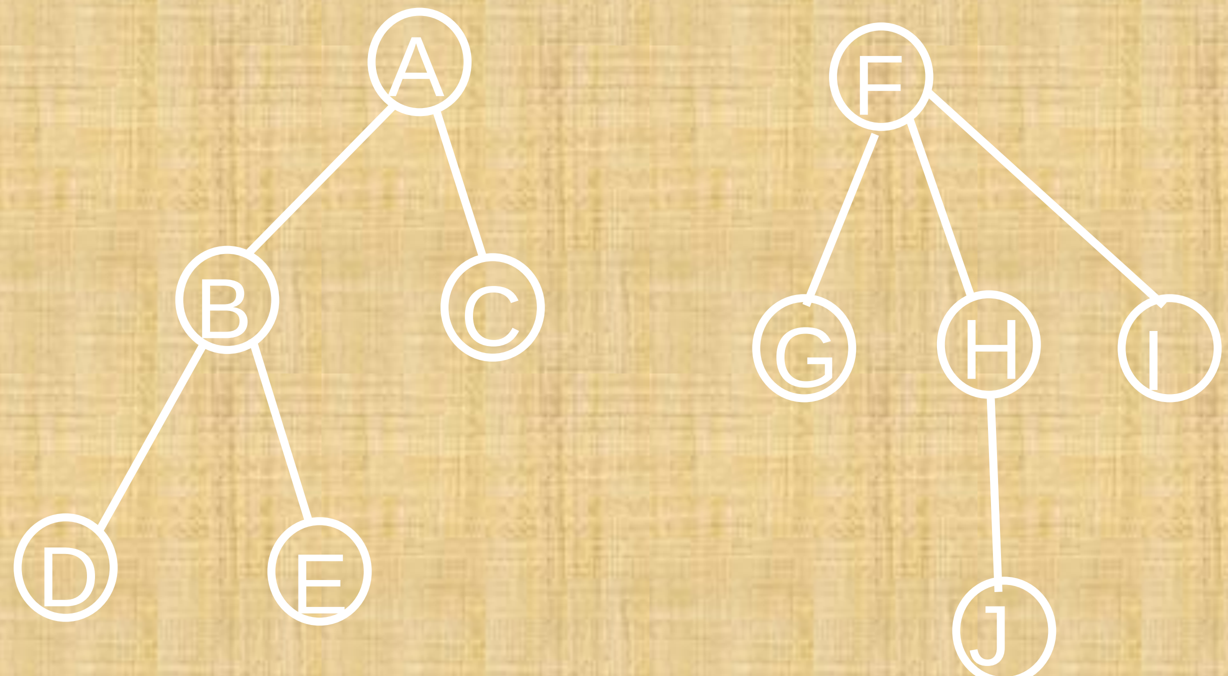
森林的深度优先遍历

② 中根遍历

- ♣ 从左到右中根遍历头一棵树树根的子树
- ♣ 访问头一棵树的根结点
- ♣ 后根遍历其它的树

对下图所示的森林,按中根遍历所得的结点序列为:

D E B C A G J H I F

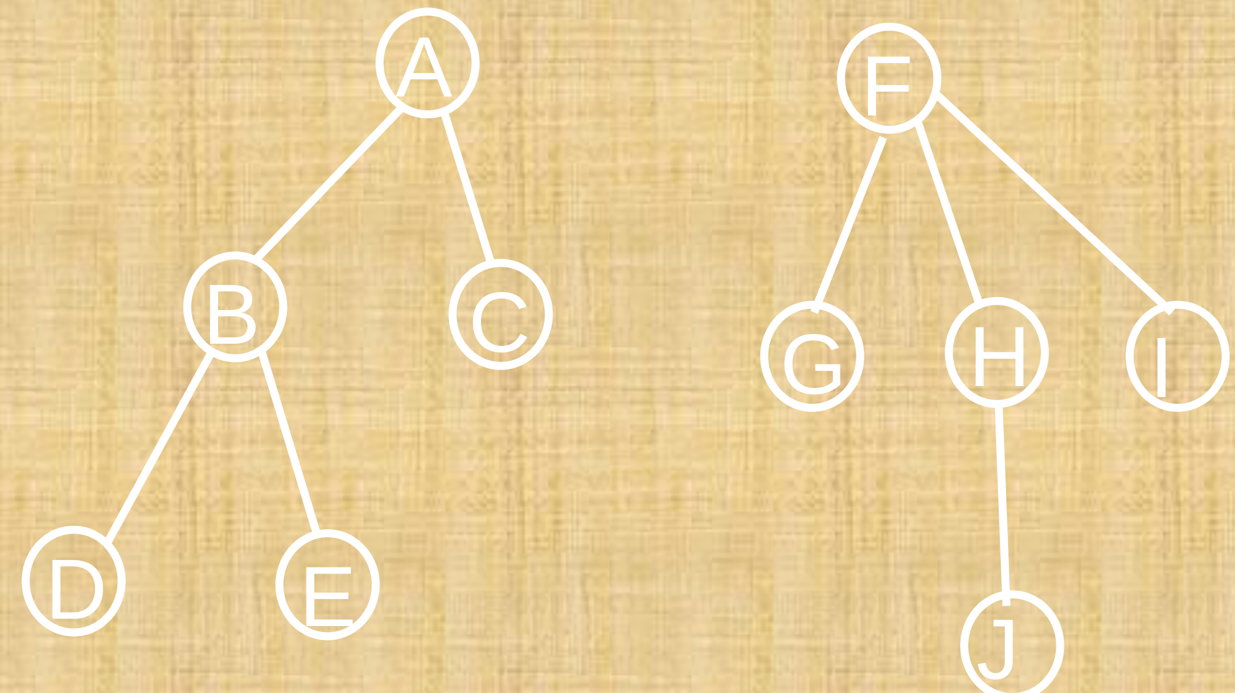


森林的广度优先遍历

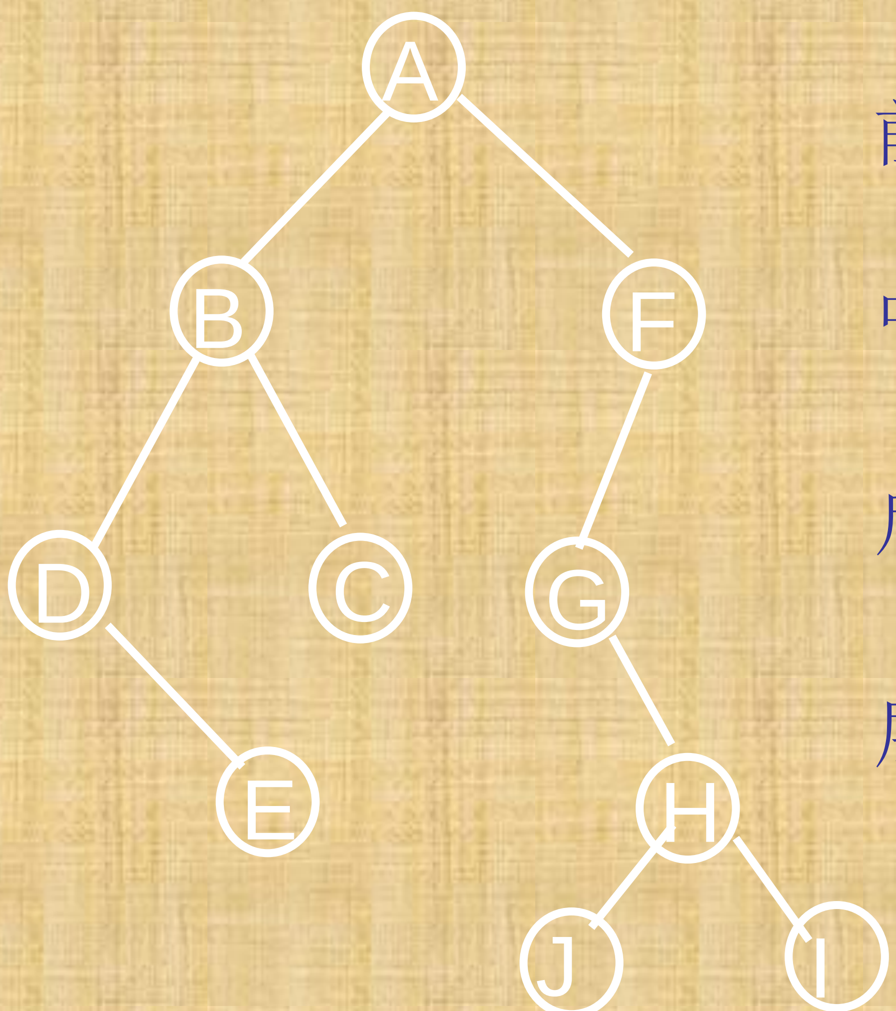
是一种按照层次遍历树林的方法，具体的遍历过程是：首先从左到右访问层数为0的所有结点，然后再从左到右访问层数为1的所有结点，. . . ，直到访问完其余各层所有的结点。

按广度优先遍历图示的森林 ,所得到的结点序列为:

AF BCGHI DEJ



对图示的森林，转换成二叉树：



前序 ABDECFGHJI

中序 DEBCAGJHIF

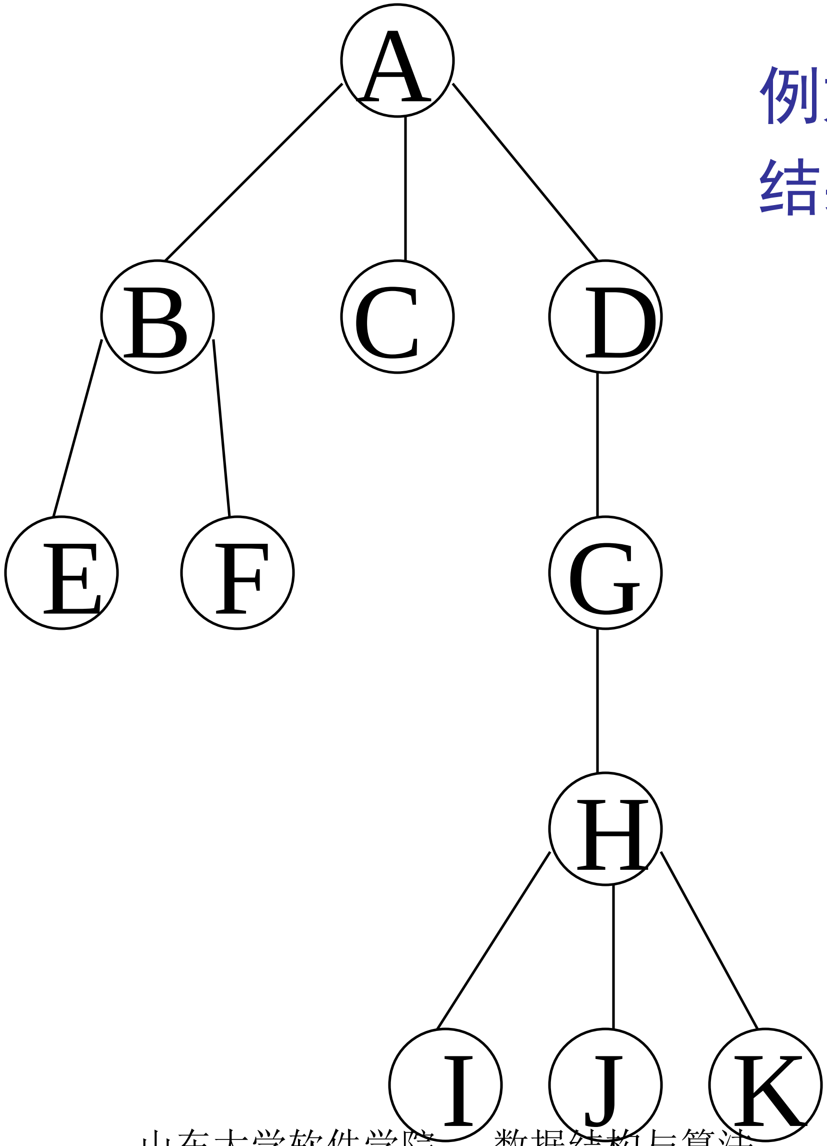
后序 EDCBJIHGFA

层次 ABFDCGEHJI

汇总

树的遍历	对应	森林的遍历	对应	二叉树的遍历
先根遍历	->	先序遍历	->	先序遍历
后根遍历	->	中序遍历	->	中序遍历

输出树中所有从根到叶子的路径的算法:



例如：对左图所示的树，其输出结果应为：

A B E

A B F

A C

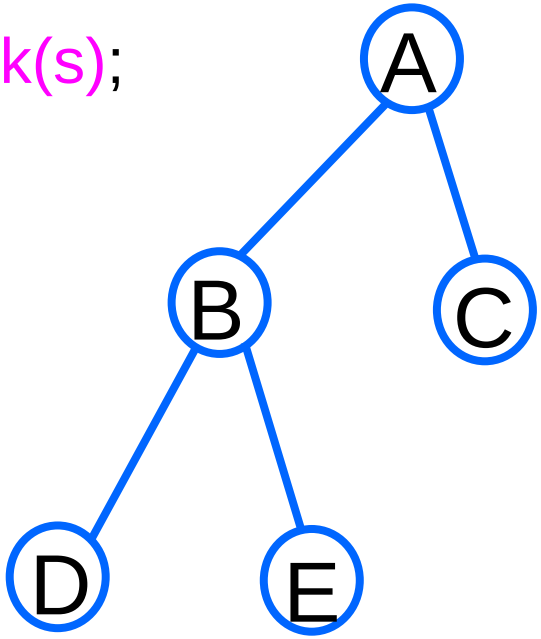
A D G H I

A D G H J

A D G H K

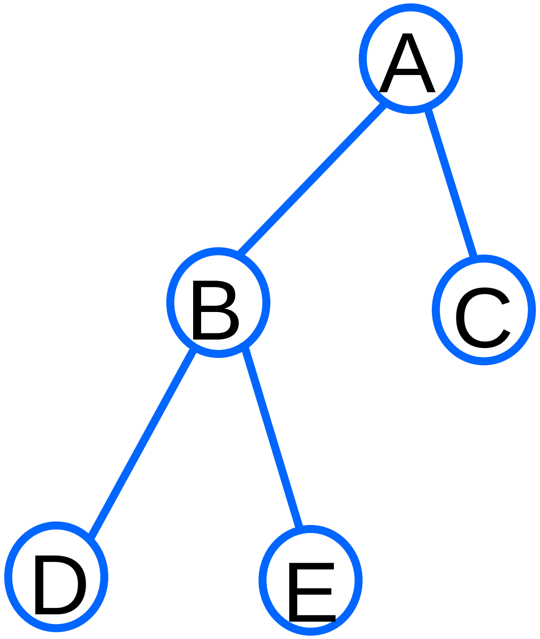
输出二叉树上从根到所有叶子结点的路径

```
void AllPath( BiTree T, Stack& s ) {
if (T) {
    s.push(T->data );
    if (!T->Lchild && !T->Rchild ) PrintStack(s);
    else {  AllPath( T->Lchild, s );
            AllPath( T->Rchild, s );
    }
    s.pop();
  } // if(T)
} // AllPath
```



输出树中所有从根到叶的路径

```
void OutPath( Bitree T, Stack& S ) {
 If ( T ) {
   S.push(T->data );
   if ( !T->firstchild ) Printstack(S);
   else
    { T=T->firstchild;
      While(T) {OutPath( T, S );
            T = T->nextsibling;)
        } // While
      }
S.pop();
} // OutPath
```



作业：

- 25， 45