

第五章 树型结构



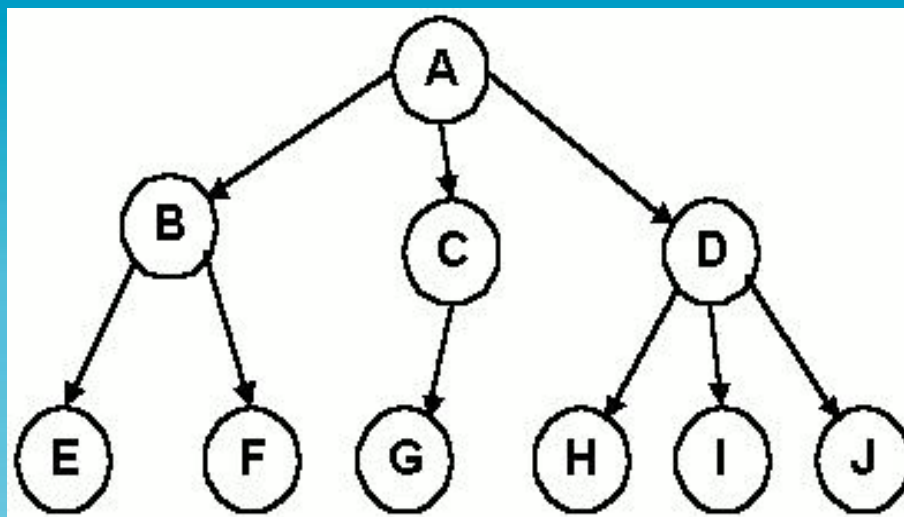
- 线性结构：反映相邻关系
- 层次关系？
- 树（Tree）型结构。
- 以分支关系定义的层次结构。



树的基本概念



- 1 树的定义
- 树（递归的描述）
- 空树
- 子树



总结出树结构的特点：

树的形式定义——抽象



- 用二元组表示:

$$\text{Tree} = (D, R)$$

若 $D = \phi$, 则 Tree 为空树, 否则

$D = \{\text{root}\} \cup D_f$, root 为树的根结点;

$D_f = D_1 \cup D_2 \cup \dots \cup D_m$, 且 $D_i \cap D_j = \phi$ ($i \neq j$, $1 \leq i, j \leq m$);

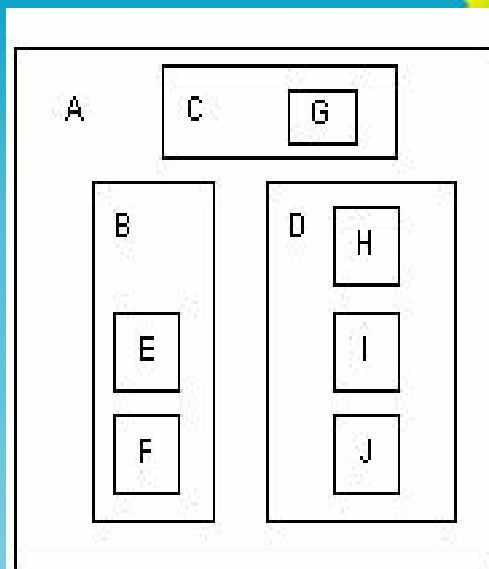
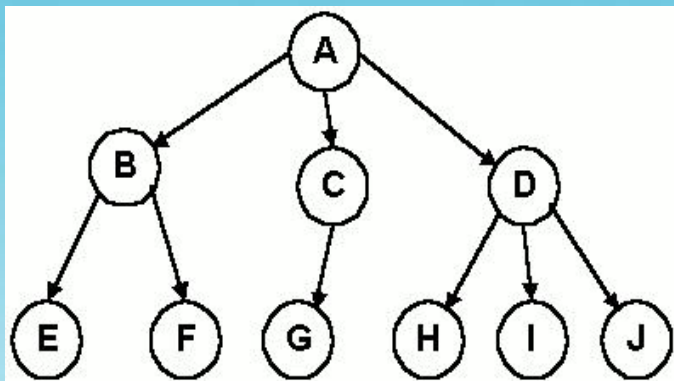
$R = \{r\}$

$$r = \begin{cases} \langle \text{root}, r_i \rangle, i=1, 2, \dots, m, \text{其中 } r_i \text{ 是树的根结点 root 的子树 } T_i \text{ 的根结点} & (m > 0) \\ \phi & m=0 \end{cases}$$

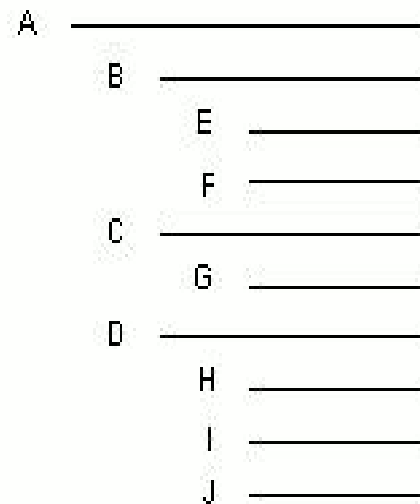
树的逻辑表示方法



- (1) 树形表示方法。
- (2) 文氏图表示方法。
- (3) 凹入表示方法。
- (4) 广义表表示方法。



(a)



(b)

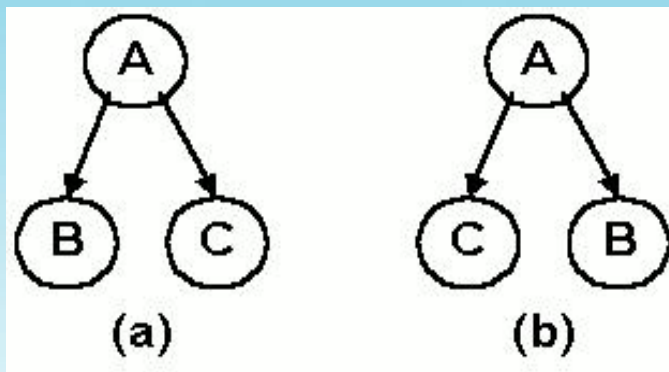
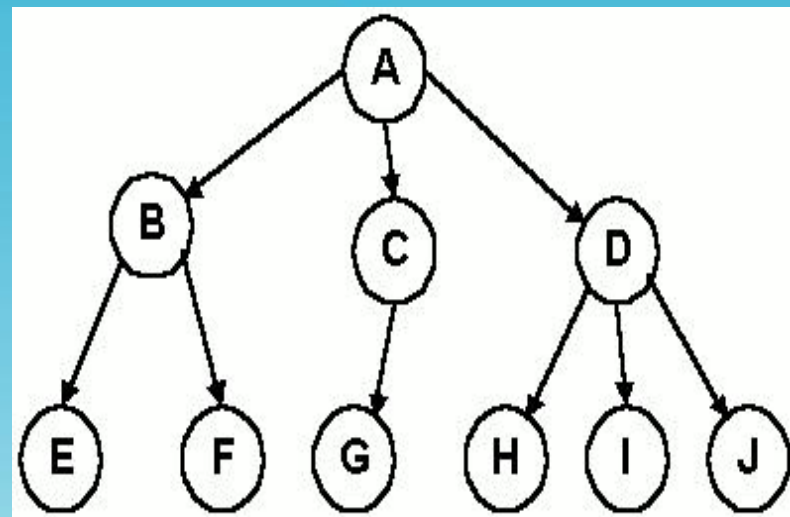
$A(B(E, F), C(G), D(H, I, J))$

(c)

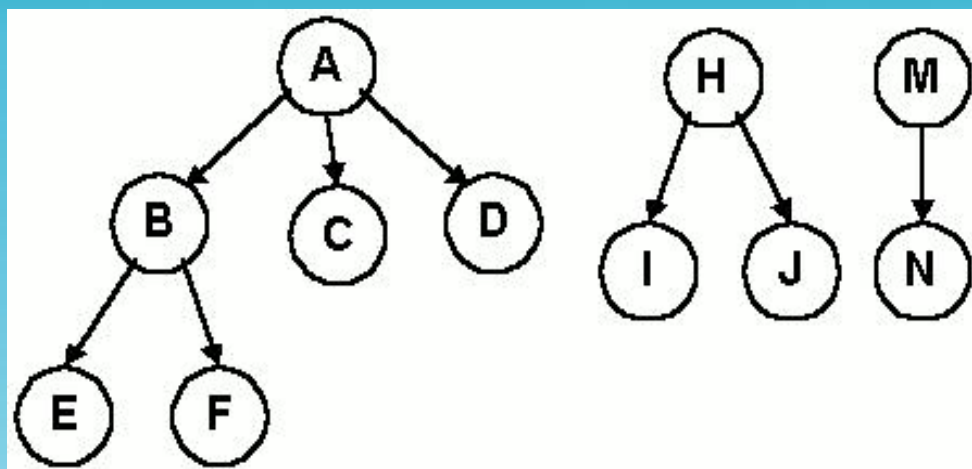
树的基本术语



- 1.根
- 2.孩子、双亲和兄弟、祖先、子孙
- 3.分支结点和叶子结点
- 4.结点的度和树的度
- 5.结点的层数和树的深度
- 6.路径和路径长度
- 7.有序树和无序树



- 8.森林



树的操作（实际生活的抽象）



- (1) 求根操作 $\text{Root}(x)$ 求出当前树中结点 x 的根结点；
- (2) 求双亲操作 $\text{Parent}(x)$ 求出当前树中结点 x 的双亲结点；
- (3) 求孩子操作 $\text{Child}(x, i)$ 求出当前树中结点 x 的第 i 个孩子结点；
- (4) 插入子树操作 $\text{Insert}(x, i, s)$ 当前树中插入根结点为 s 的子树，并作为结点 x 的第 i 棵子树；
- (5) 删除子树操作 $\text{Delete}(x, i)$ 在当前树中删除结点 x 的第 i 棵子树；
- (6) 树的遍历操作 $\text{Travel}()$ 按某种次序依次访问当前树中的各个结点，并使每个结点只被访问一次；
- (7) 清除操作 $\text{Clear}()$ 将当前树置为空树；
- (8) 求树高 $\text{Depth}()$ 求出树的高度；
- (9) 树建立操作 $\text{Create}()$ 建立一棵非空的树并成为当前树；

树的抽象数据类型



```
template <class T>
```

```
class Tree {
```

```
//对象：树是由 $n (\geq 0)$  个结点组成的有限集合。在  
//类界面中的 position 是树中结点的地址。在顺序  
//存储方式下是下标型,在链表存储方式下是指针  
//型。T 是树结点中存放数据的类型,要求所有结  
//点的数据类型都是一致的。
```

```
public:
```

```
    Tree ();
```

```
    ~Tree ();
```

BuildRoot (const T& value);

//建立树的根结点

position FirstChild(position p);

//返回 p 第一个子女地址, 无子女返回 0

position NextSibling(position p);

//返回 p 下一兄弟地址, 若无下一兄弟返回 0

position Parent(position p);

//返回 p 双亲结点地址, 若 p 为根返回 0

T getData(position p);

//返回结点 p 中存放的值

bool InsertChild(position p, T& value);

//在结点 p 下插入值为 value 的新子女, 若插

//入失败, 函数返回false, 否则返回true





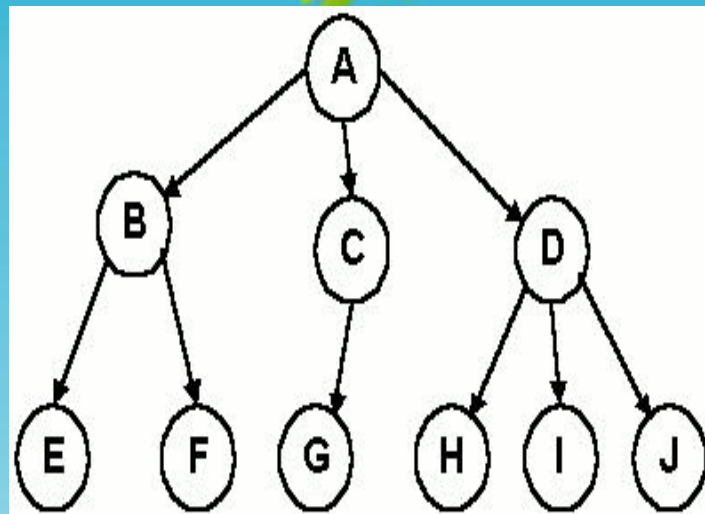
```
bool DeleteChild (position p, int i);  
    //删除结点 p 的第 i 个子女及其全部子孙结  
    //点, 若删除失败, 则返回false, 否则返回true  
void DeleteSubTree (position t);  
    //删除以 t 为根结点的子树  
bool IsEmpty ();  
    //判树空否, 若空则返回true, 否则返回false  
void Traversal (void (*visit)(position p));  
    //遍历以 p 为根的子树  
};
```

树的性质



- 性质1:

树中的结点总数
与分支数
与所有结点的度数之和
有何关系?



树的性质



- 性质2:

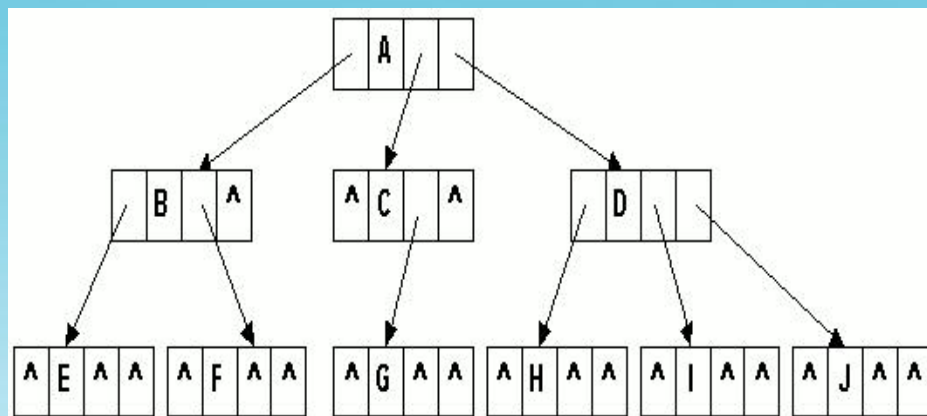
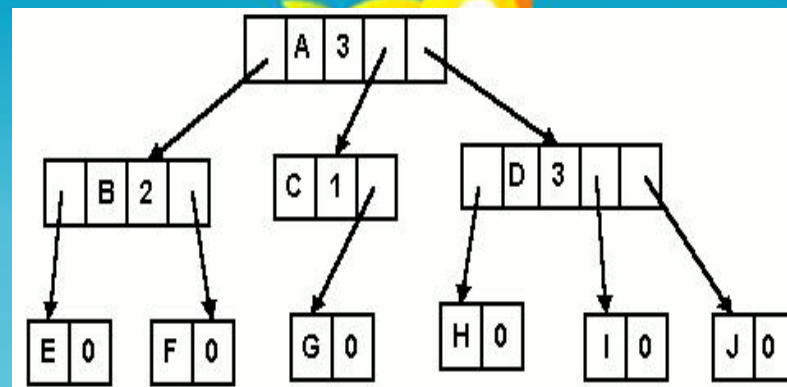
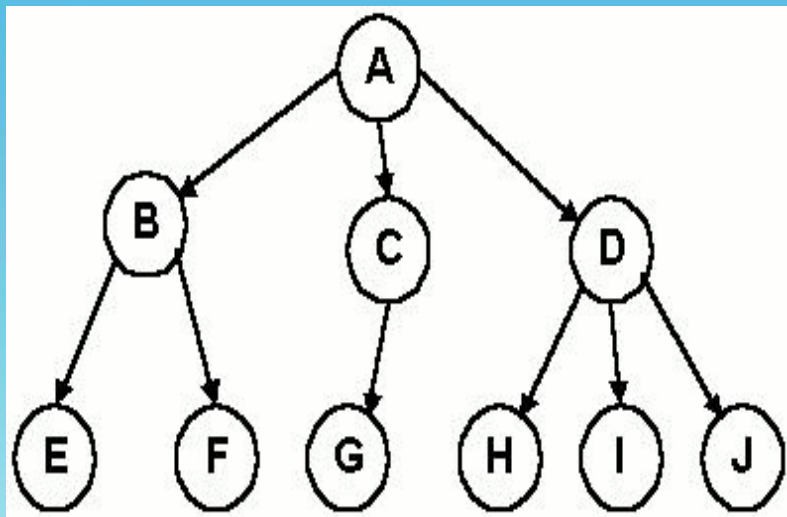
度为 k 的树中第 i 层上至多有 k^{i-1} 个结点
($i \geq 1$)。

树的存储结构

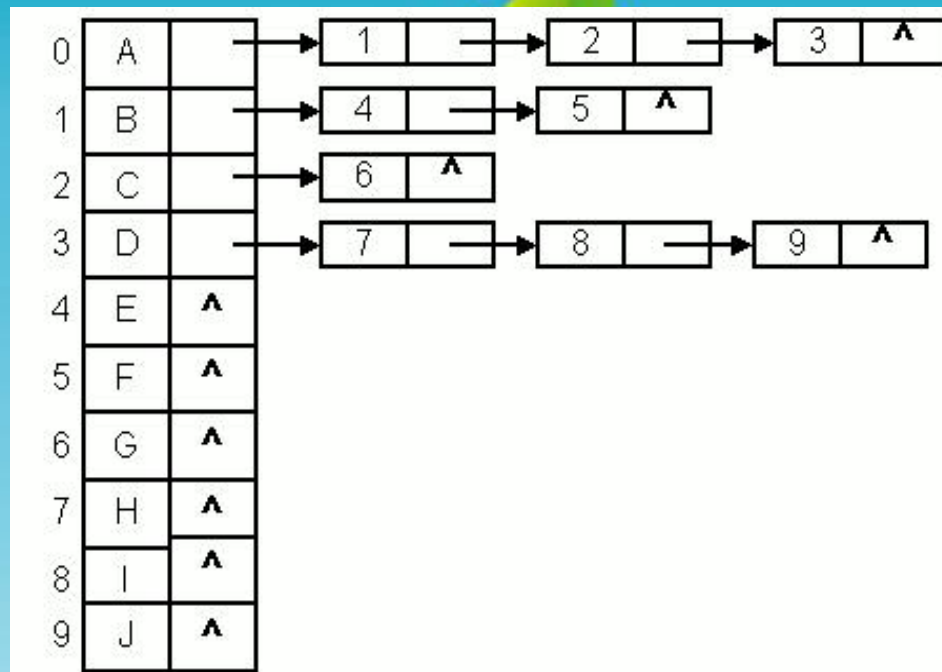
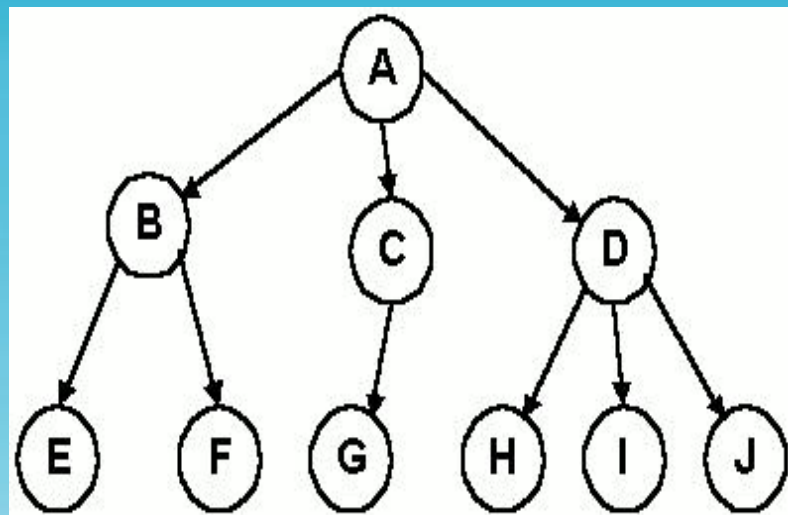


链式结构

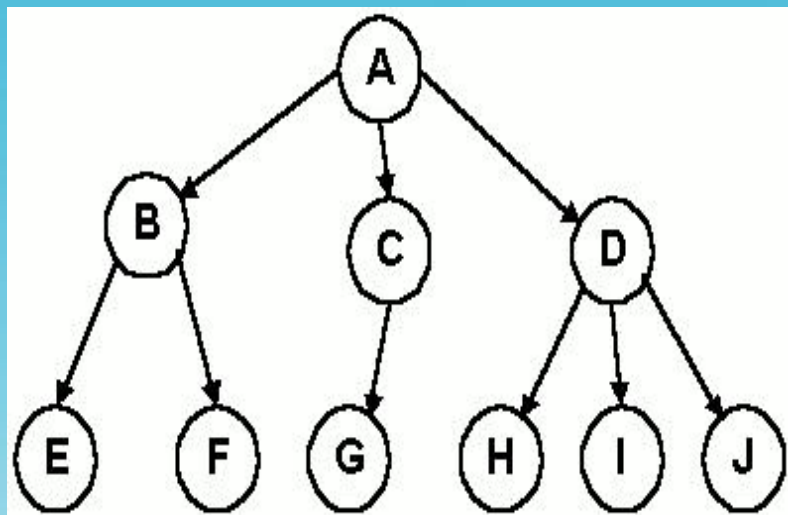
- (1) 不定长的多重链式结构
- (2) 定长的多重链式结构



• (3) 孩子链表示方法



• 双亲表示法



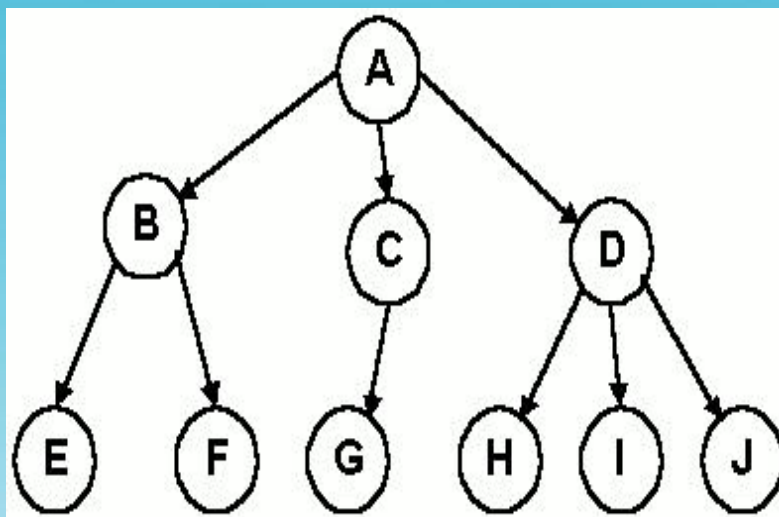
| 结点序号 | data | parent |
|------|------|--------|
| 0 | A | -1 |
| 1 | B | 0 |
| 2 | C | 0 |
| 3 | D | 0 |
| 4 | E | 1 |
| 5 | F | 1 |
| 6 | G | 2 |
| 7 | H | 3 |
| 8 | I | 3 |
| 9 | J | 3 |

问题的分析及解决



树的存储结构有问题？

如何解决？

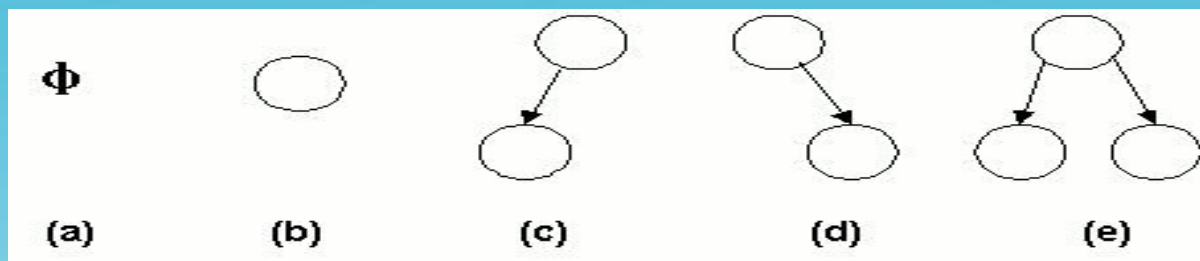


二叉树



1. 二叉树的定义

2. 二叉树的基本形态:



注意事项:

- (1) 二叉树与无序树不相同
- (2) 二叉树与度为2的有序树不同

树、森林与二叉树的转换



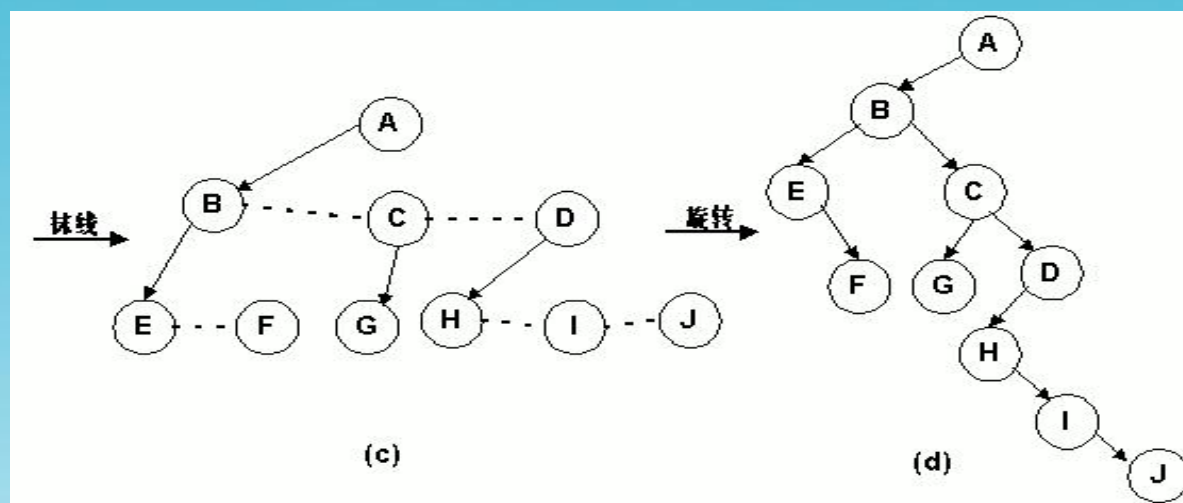
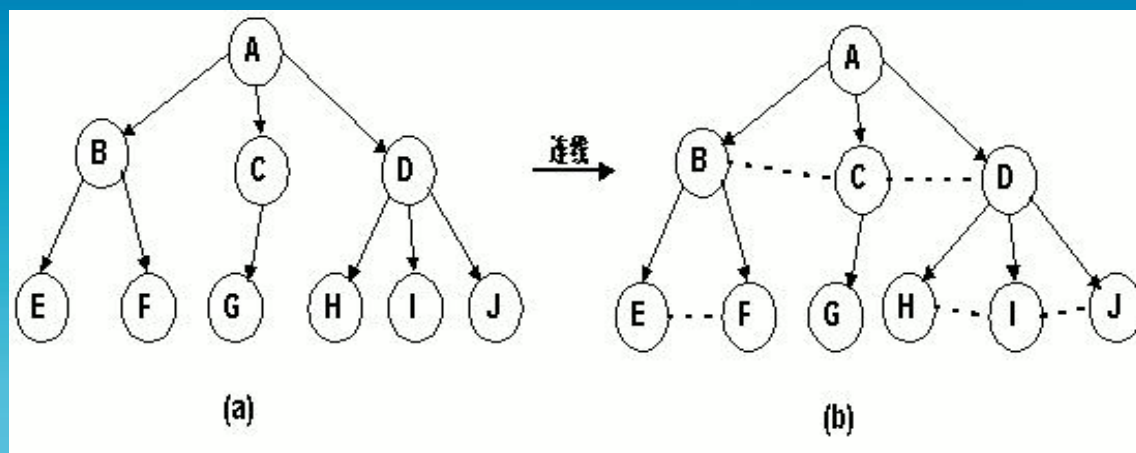
- 1.为什么要将树或森林转换为二叉树?

一般树转化为二叉树



转换步骤:

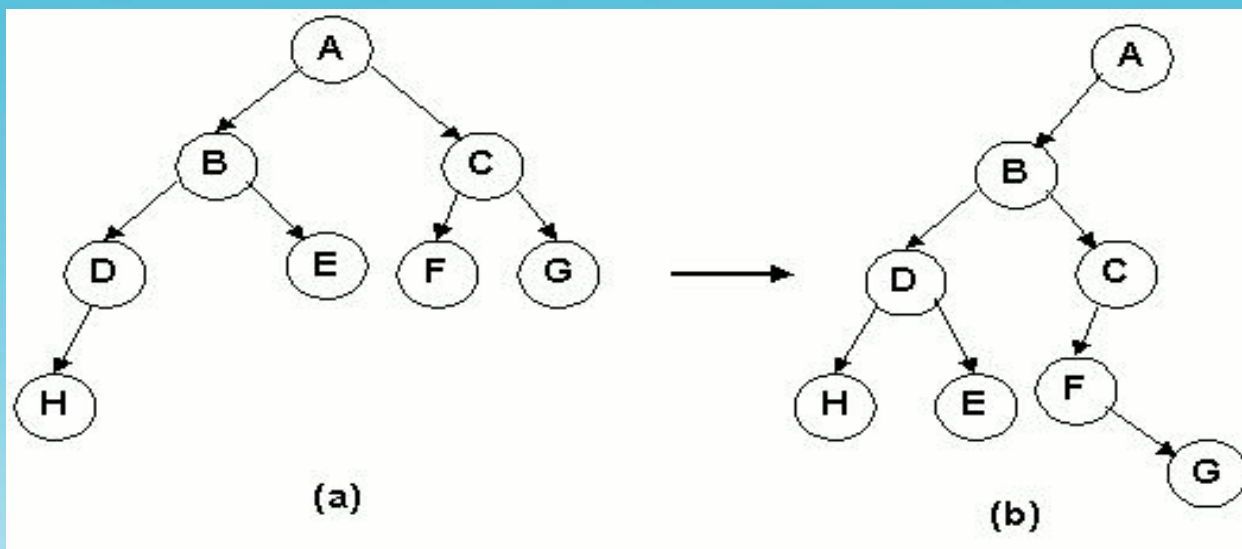
- (1) 加线: 在个兄弟结点之间加一连线。
- (2) 抹线: 对每一结点, 除了其最左的一个孩子以外, 抹掉该结点原先与其余孩子之间的连线。
- (3) 旋转: 以树根为轴心, 将整棵树按顺时针旋转 45° 。



总结特点:

- 注意事项:

- 一棵形状像二叉树的树，仍然需转换才能得到二叉树，而不能说不用转换了。



• 2. 二叉树还原为一般树

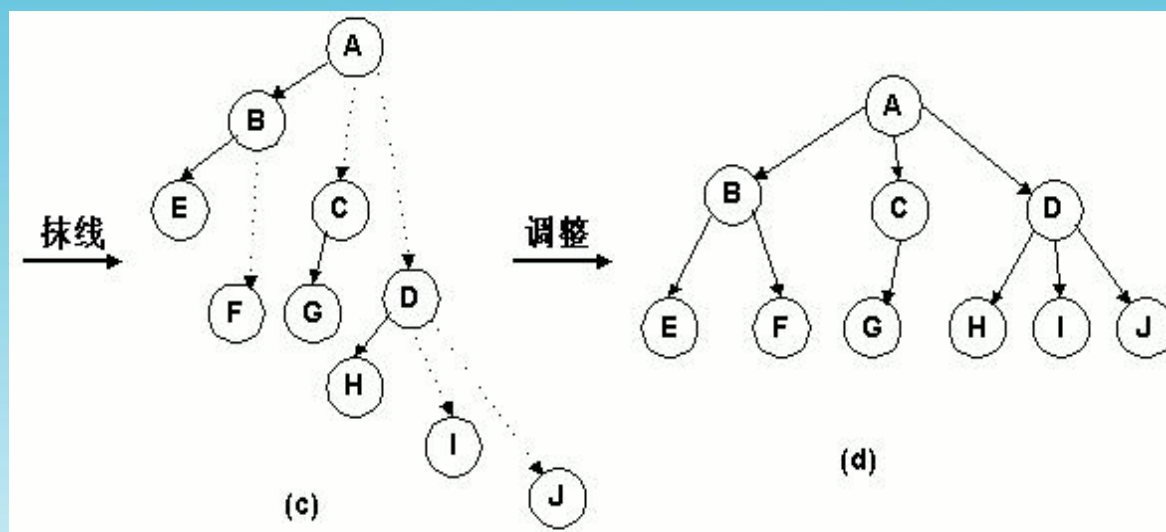
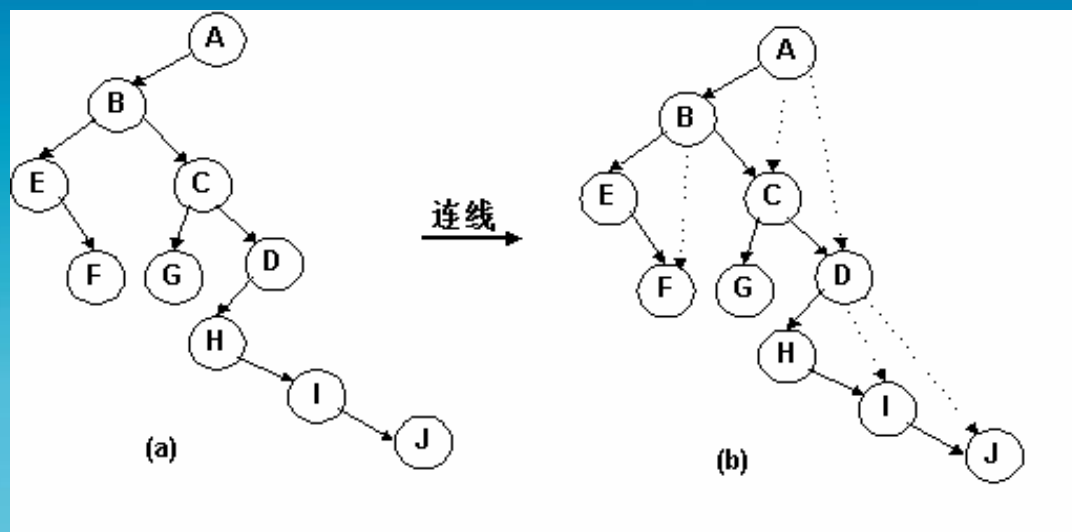
步骤:



(1) 加线:若某结点是双亲结点的左孩子,则将该结点的右孩子以及当且仅当连续地沿着此右孩子的右子树方向不断地搜索到的所有右孩子,都分别与该结点的双亲结点用连线连接起来

(2) 抹线:删去原二叉树中所有双亲结点与右孩子的连线。

(3) 调整:把属于同一层的结点调整到同一水平线上。



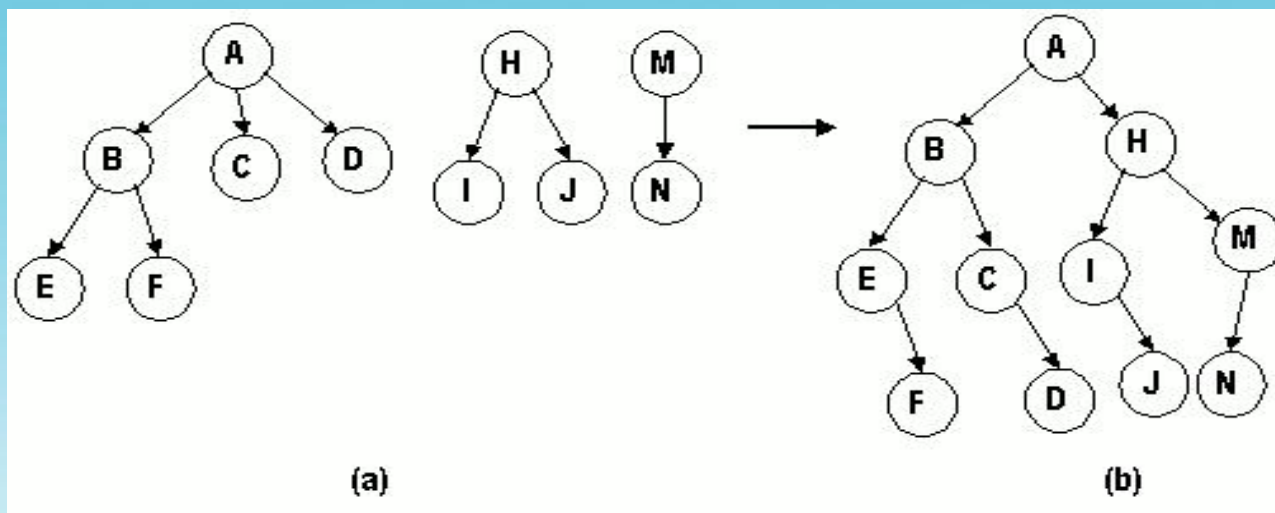
3.森林转换为二叉树



转换方法：

(1) 将森林中每棵树转换为二叉树。

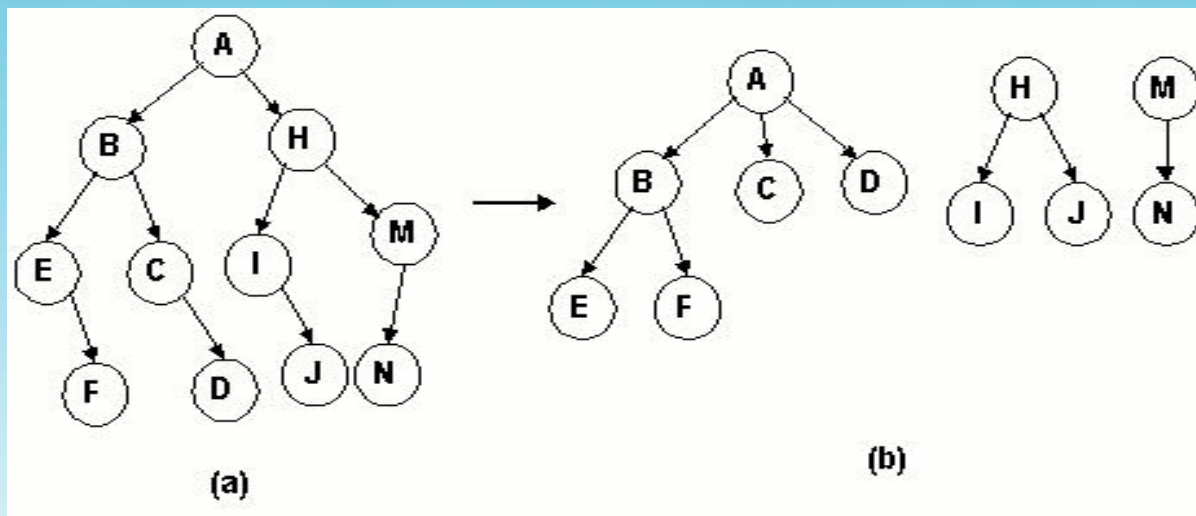
(2) 将森林中每棵转换所得的二叉树的树根用线相连。即第一棵二叉树不动，从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树根结点的右孩子。



• 4. 二叉树还原为森林

• 转换步骤:

- (1) 抹线: 将二叉树的根结点与其右孩子 i 的连线以及当且仅当连续地沿着结点 i 的右分支不断搜索到的所有右孩子间的连线删去, 这样就可以得到若干棵独立的二叉树。
- (2) 还原: 分别将各棵独立的二叉树还原为树。



二叉树的性质



- 性质1 二叉树第 i 层的结点数最多有 2^{i-1} 个 ($i \geq 1$)

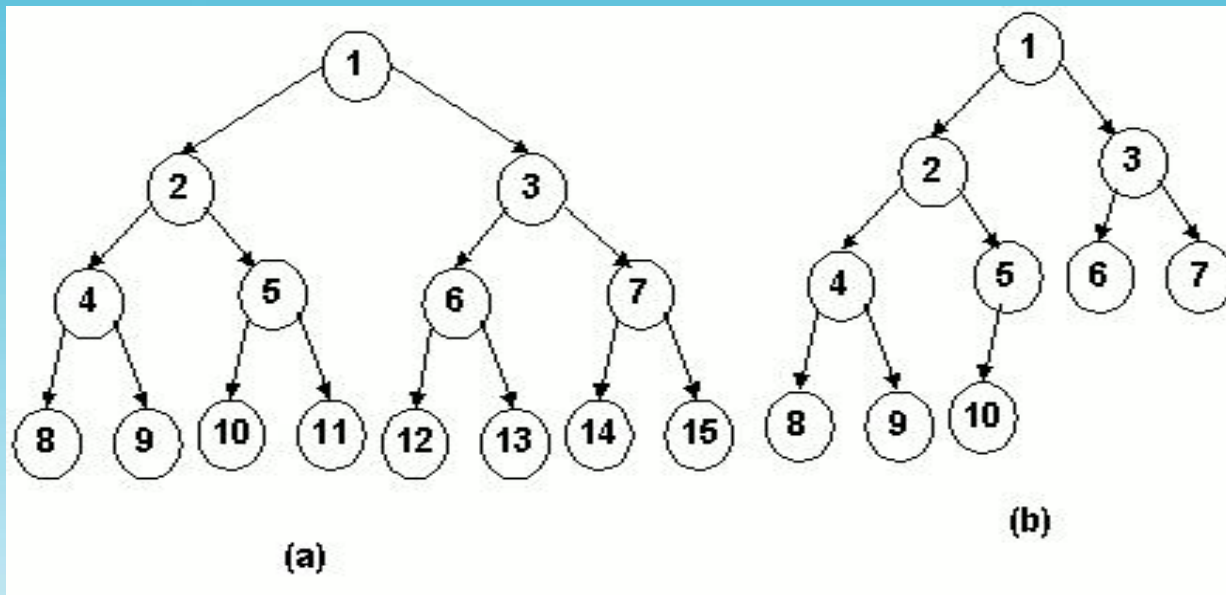




- 性质2:深度为 h 的二叉树至多有 2^h-1 个结点 ($h \geq 1$)

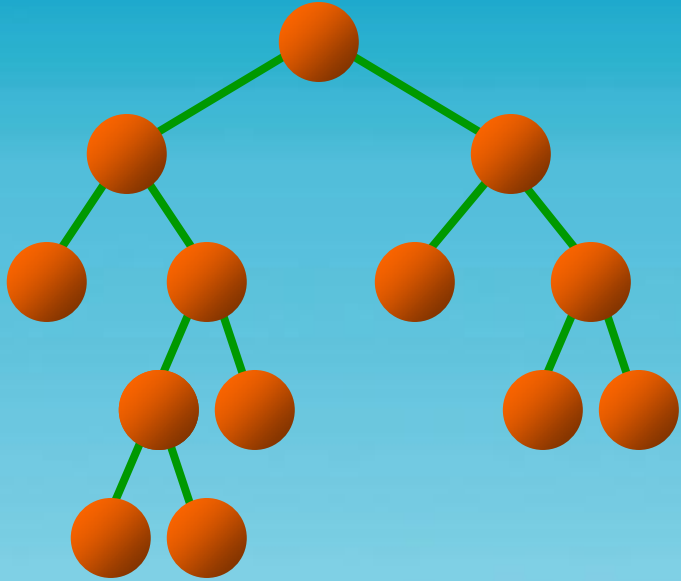


- 满二叉树
- 完全二叉树

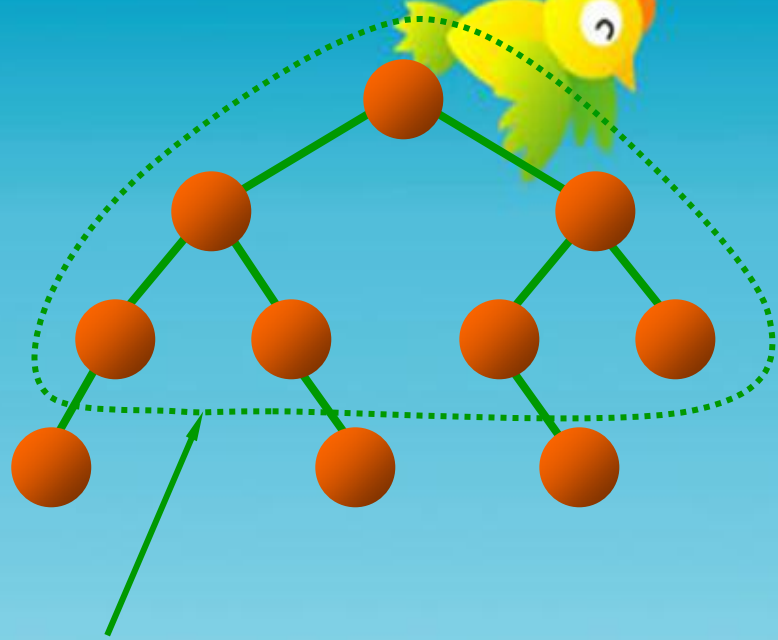




正则二叉树



理想平衡二叉树



满的

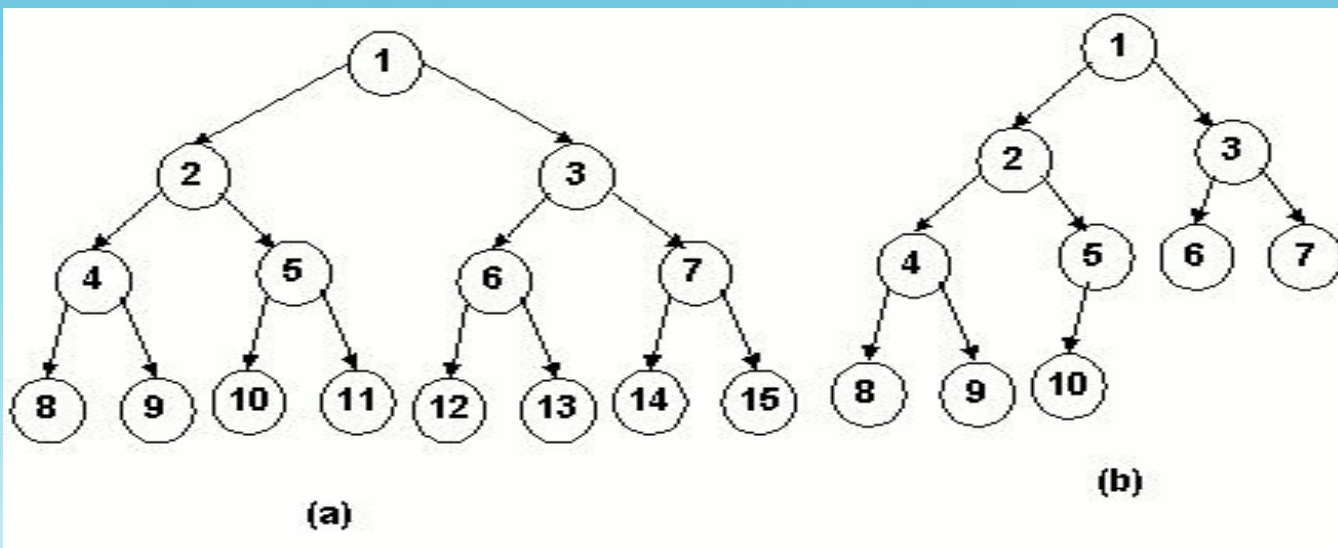
- 性质3 具有n个结点的完全二叉树的深度为多少?

$$\lfloor \log_2 n \rfloor + 1 \text{ 或 } \lceil \log_2^{(n+1)} \rceil$$

- 性质5:对于任意一棵非空的二叉树, 如果叶子总数为 n_0 个, 度为2的结点总数为 n_2 个, 度为1的结点总数为 n_1 个, 则 n_0 、 n_1 、 n_2 有何关系?

性质4: 对于具有 n 个结点的完全二叉树,对其结点进行编号, 则对于编号为 i 的结点,有:

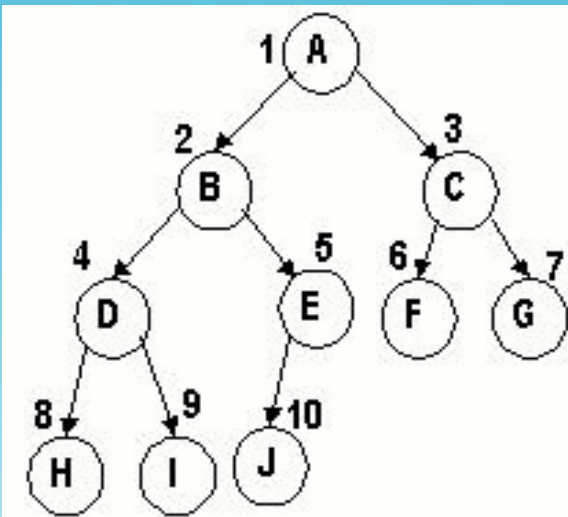
- (1)双亲的编号为多少? 存在的条件?
- (2)左孩子的编号为多少? 存在的条件?
- (3)右孩子的编号为多少? 存在的条件?



二叉树的存储结构



- 1.顺序存储结构
- 如何用单元之间的关系来反映结点之间的层次关系？



(a)

单元编号:

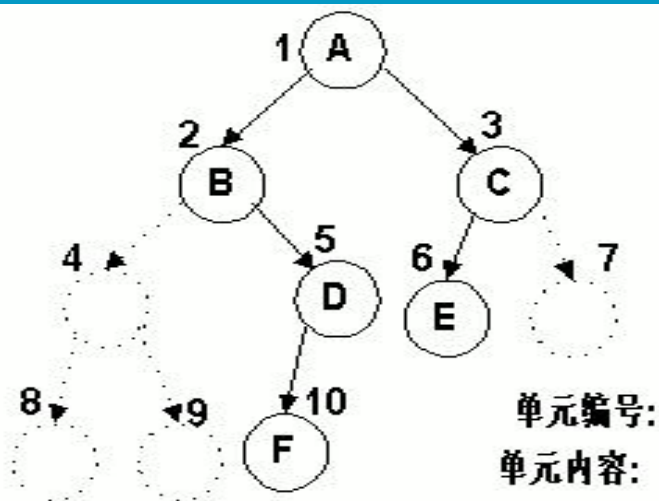
1 2 3 4 5 6 7 8 9 10

单元内容:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I | J |
|---|---|---|---|---|---|---|---|---|---|

(b)

顺序存储的缺点

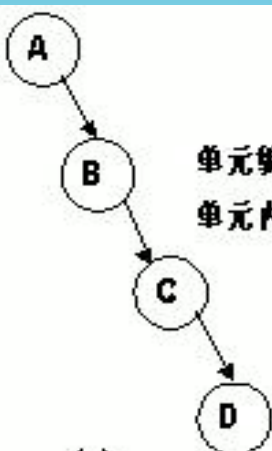


单元编号:
单元内容:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|----|
| A | B | C | | D | E | | | | F |

(a)

(b)



单元编号:
单元内容:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| A | | B | | | | C | | | | | | | | D |

(b)

(a)

• 2.链式存储结构

用指针的关系来反映结点之间的层次关系。

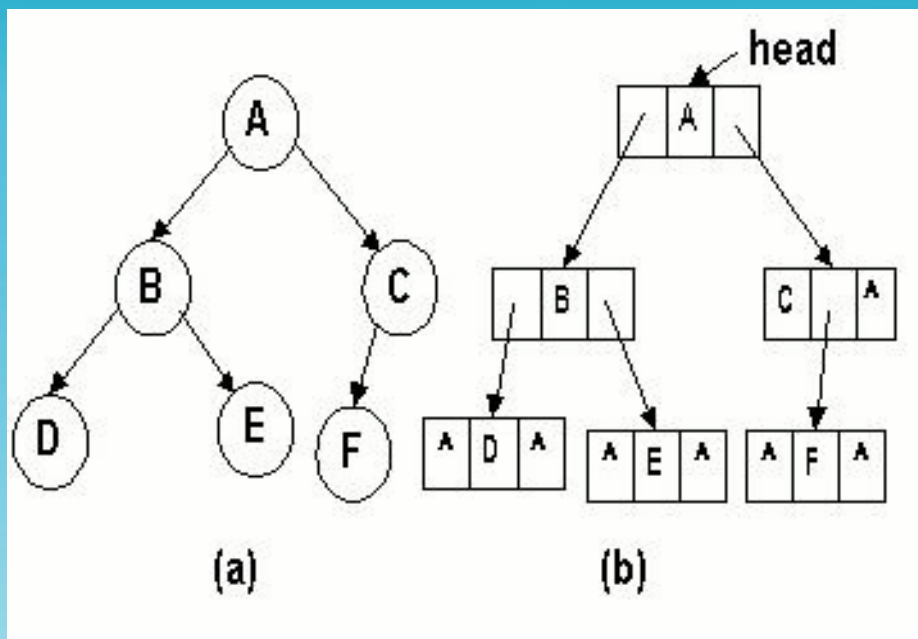


结点结构：

```
struct BinTreeNode
{   T data;
    BinTreeNode *leftchild, *rightchild;
};
```



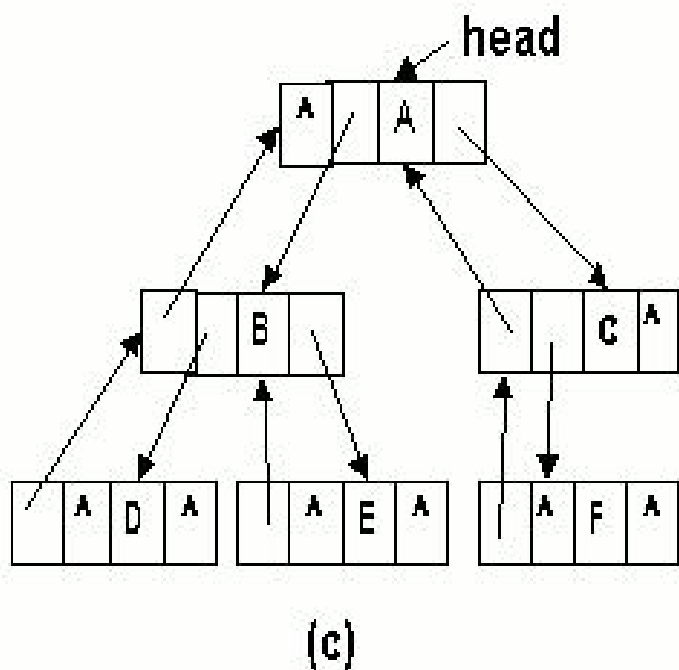
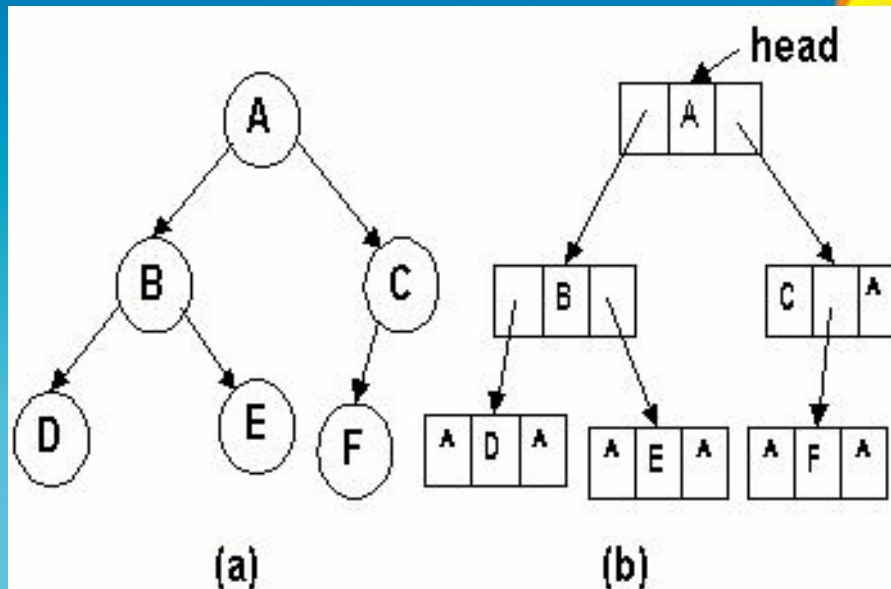
链式结构



链式结构的缺点?

二叉链表

三叉链表



三叉链表存储结构:

```
struct BinTreeNode
```

```
{ T data;
```

```
    BinTreeNode *parent;
```

```
    BinTreeNode *leftchild,*rightchild;
```

```
};
```

- 二叉树顺序存储与链式存储的比较：
- (1) 顺序存储适用范围为完全二叉树，否则会浪费空间
- 插入和删除需移动结点
- 找子孙双亲都容易。
- (2) 链式存储则便于插入和删除结点
- 浪费空间，查找费时
- 链式存储子孙容易，找双亲难
- 解决方法：三叉链式存储，浪费空间。



二叉树操作的抽象



二叉树操作



- (1) 求根操作 $\text{Root}(x)$ 求出当前二叉树中结点 x 的根结点;
- (2) 求双亲操作 $\text{Parent}(x)$ 求出当前二叉树中结点 x 的双亲结点;
- (3) 求左孩子操作 $\text{leftchild}(x)$ 求出当前二叉树中结点 x 的左孩子结点;
- (4) 求右孩子操作 $\text{rightchild}(x)$ 求出当前二叉树中结点 x 的右孩子结点;
- (5) 插入左子树操作 $\text{Linsert}(x,s)$ 当前二叉树中插入根结点为 s 的子树, 并作为无左子树的结点 x 的左子树;
- (6) 插入右子树操作 $\text{Rinsert}(x,s)$ 当前二叉树中插入根结点为 s 的子树, 并作为无右子树的结点 x 的右子树;
- (7) 删除左子树操作 $\text{Ldelete}(x)$ 在当前二叉树中删除结点 x 的左子树;
- (8) 删除右子树操作 $\text{Rdelete}(x)$ 在当前二叉树中删除结点 x 的右子树;
- (9) 遍历操作 $\text{Travel}()$ 按某种次序依次访问当前二叉树中的各个结点, 并使每个结点只被访问一次;
- (10) 清除操作 $\text{Clear}()$ 将当前二叉树置为空二叉树;
- (11) 求树高 $\text{Depth}()$ 求出二叉树的高度;
- (12) 建立二叉树操作 $\text{Create}()$ 建立一棵非空的二叉树并成为当前二叉树;

二叉树的抽象数据类型



```
template <class T>
```

```
class BinaryTree {
```

```
//对象: 结点的有限集合, 二叉树是有序树
```

```
public:
```

```
    BinaryTree ();                //构造函数
```

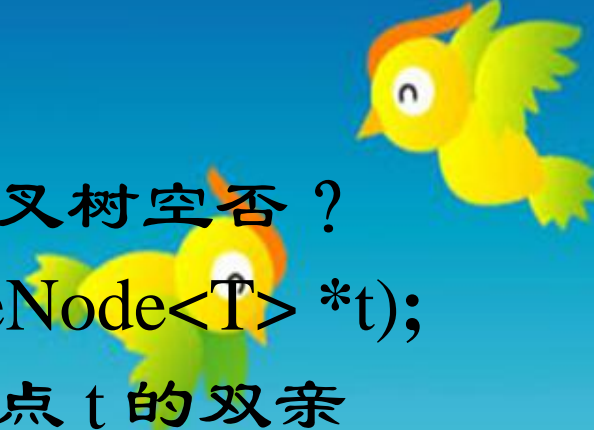
```
    BinaryTree (BinTreeNode<T> *lch,  
                BinTreeNode<T> *rch, T item);
```

```
    //构造函数, 以item为根, lch和rch为左、右子
```

```
    //树构造一棵二叉树
```

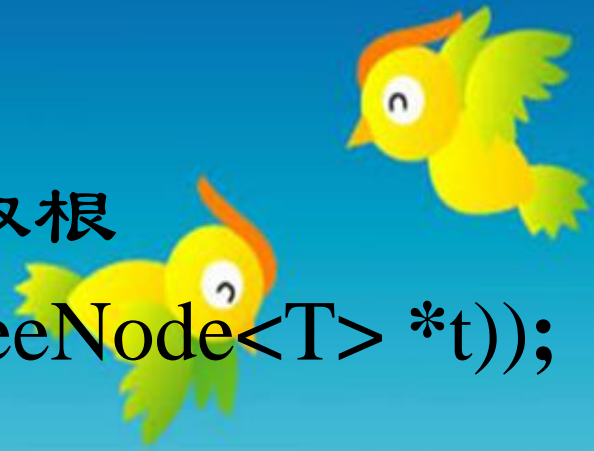
```
    int Height ();                //求树深度或高度
```

```
    int Size ();                  //求树中结点个数
```



```
bool IsEmpty ();           //判二叉树空否 ?
BinTreeNode<T> *Parent (BinTreeNode<T> *t);
                           //求结点 t 的双亲
BinTreeNode<T> *LeftChild (BinTreeNode<T> *t);
                           //求结点 t 的左子女
BinTreeNode<T> *RightChild (BinTreeNode<T> *t);
                           //求结点 t 的右子女
bool Insert (T item);      //在树中插入新元素
bool Remove (T item);      //在树中删除元素
bool Find (T& item);       //判断item是否在树中
bool getData (T& item);    //取得结点数据
```

```
BinTreeNode<T> *getRoot ();//取根  
void Travel(void (*visit) (BinTreeNode<T> *t));  
//遍历, visit是访问函数  
};
```



二叉树的类定义



```
template <class T>
struct BinTreeNode {           //二叉树结点类定义
    T data;                    //数据域
    BinTreeNode<T> *leftChild, *rightChild;
                                //左子女、右子女链域
    BinTreeNode ()             //构造函数
    { leftChild = NULL; rightChild = NULL; }
    BinTreeNode (T x, BinTreeNode<T> *l = NULL,
                  BinTreeNode<T> *r = NULL)
    { data = x; leftChild = l; rightChild = r; }
};
```

```
template <class T>
```

```
class BinaryTree {
```

```
public:
```

```
    BinaryTree () : root (NULL) { }      //构造函数
```

```
    BinaryTree (T value) : RefValue(value), root(NULL)  
    { }                                  //构造函数
```

```
    BinaryTree (BinaryTree<T>& s);      //复制构造函数
```

```
    ~BinaryTree () { destroy(root); }   //析构函数
```

```
    bool IsEmpty () { return root == NULL;}
```

```
                                //判二叉树空否
```

```
    int Height () { return Height(root); } //求树高度
```

```
    int Size () { return Size(root); }    //求结点数
```

//二叉树类定义

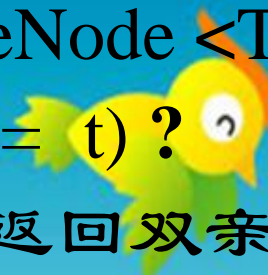


```
BinTreeNode<T> *Parent (BinTreeNode <T> *t)
{ return (root == NULL || root == t) ?
    NULL : Parent (root, t); } //返回双亲结点
```

```
BinTreeNode<T> *LeftChild (BinTreeNode<T> *t)
{ return (t != NULL) ? t->leftChild : NULL; }
//返回左子女
```

```
BinTreeNode<T> *RightChild (BinTreeNode<T> *t)
{ return (t != NULL) ? t->rightChild : NULL; }
//返回右子女
```

```
BinTreeNode<T> *getRoot () const { return root; }
//取根
```



int Insert (const T item); //插入新元素

BinTreeNode<T> *Find (T item) const; //

搜索

void Travel(void (*visit) (BinTreeNode<T> *t));

//遍历, visit是访问函数



protected:

```
BinTreeNode<T> *root;    //二叉树的根指针  
T RefValue;              //数据输入停止标志
```

```
void CreateBinTree (istream& in,  
                    BinTreeNode<T> *& subTree);  
                    //从文件读入建树
```

```
bool Insert (BinTreeNode<T> *& subTree, T& x);  
//插入
```

```
void destroy (BinTreeNode<T> *& subTree);  
//删除
```

```
bool Find (BinTreeNode<T> *subTree, T& x);  
//查找
```




```
BinTreeNode<T> *Copy (BinTreeNode<T> *r);
```

//复制

```
int Height (BinTreeNode<T> *subTree);
```

//返回树高度

```
int Size (BinTreeNode<T> *subTree);
```

//返回结点数

```
BinTreeNode<T> *Parent (BinTreeNode<T> *  
    subTree, BinTreeNode<T> *t);
```

//返回父结点

```
BinTreeNode<T> *Find (BinTreeNode<T> *  
    subTree, T& x) const;    //搜寻x
```





```
friend ostream& operator >> (ostream& out,  
    BinaryTree<T>& Tree); //重载操作：输入  
friend ostream& operator << (ostream& out,  
    BinaryTree<T>& Tree); //重载操作：输出  
};
```

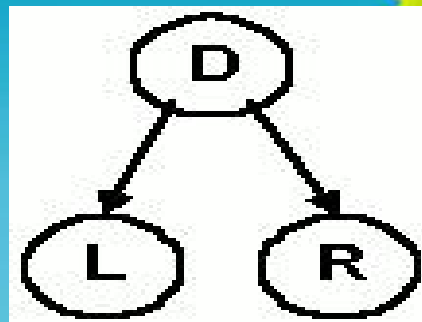
二叉树的遍历



- 二叉树的遍历就是按照某种顺序访问二叉树中的每个结点，并使每个结点被访问一次且只被访问一次。
- 访问的理解？
- 对结点的增加、删除、修改、查阅或加工
- 先简化化为结点数据域值的输出。
- 作用：将非线性结构变为线性结构

二叉树的遍历

DLR, DRL,
LDR, RDL
LRD, RLD。



中根（序）遍历
先根（序）遍历
后根（序）遍历。
层次遍历



• 因此遍历函数有四个，分别为：



- `void preOrder(BinTreeNode<T>*subTree, void(*visit)(BinTreeNode<T> *p));`
- `void preOrder(void (*visit)(BinTreeNode<T> *p));` //公有成员函数

// 对二叉树进行前序遍历

- `void inOrder(BinTreeNode<T>*subTree, void (*visit)(BinTreeNode<T> *p));`
- `void inOrder(void (*visit)(BinTreeNode<T> *p));` //公有成员函数
- // 对二叉树进行中序遍历

- `void postOrder(BinTreeNode<T> *subTree, void (*visit)(BinTreeNode<T> *p));`
- `void postOrder(void (*visit)(BinTreeNode<T> *p));` //公有成员函数

// 对二叉树进行后序遍历

- `void levelOrder(void (*visit)(BinTreeNode<T> *p));`

// 对二叉树进行层次遍历

- `void BinaryTree<T>::preOrder(void(*visit)(BinTreeNode<T> *p));`
- `//公有成员函数`

`// 对二叉树进行前序遍历`

```
{ preOrder( root,visit ); }
```

- `void BinaryTree<T>::inOrder(void (*visit)(BinTreeNode<T> *p));`
- `//公有成员函数`

`// 对二叉树进行中序遍历`

```
{ inOrder( root,visit ); }
```

- `void BinaryTree<T>::postOrder(void(*visit)(BinTreeNode<T> *p));`
- `//公有成员函数`

`// 对二叉树进行后序遍历`

```
{ postOrder( root,visit ); }
```



```
void preOrder( void (*visit)(BinTreeNode<T> *p) )  
//调用的方法说明
```

```
void f ( BinTreeNode<T> *p )  
{ cout<<p->data; }
```

调用preOrder的方法

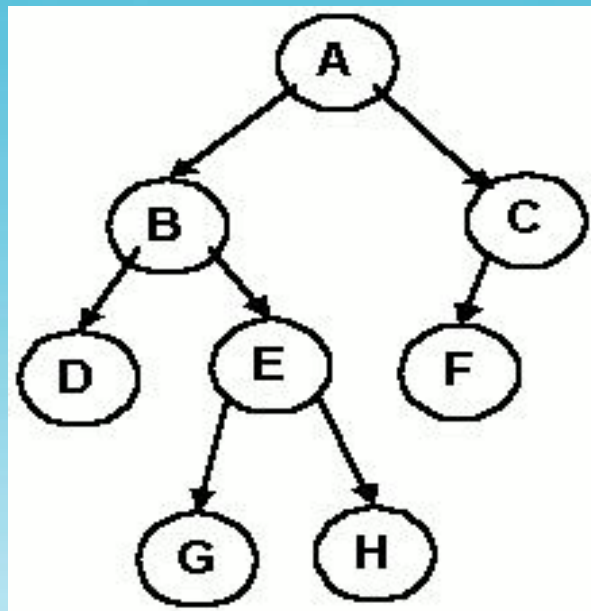
```
MyBTree.preOrder( f );
```



• 1.前序遍历

过程：若二叉树非空，则做如下的操作：

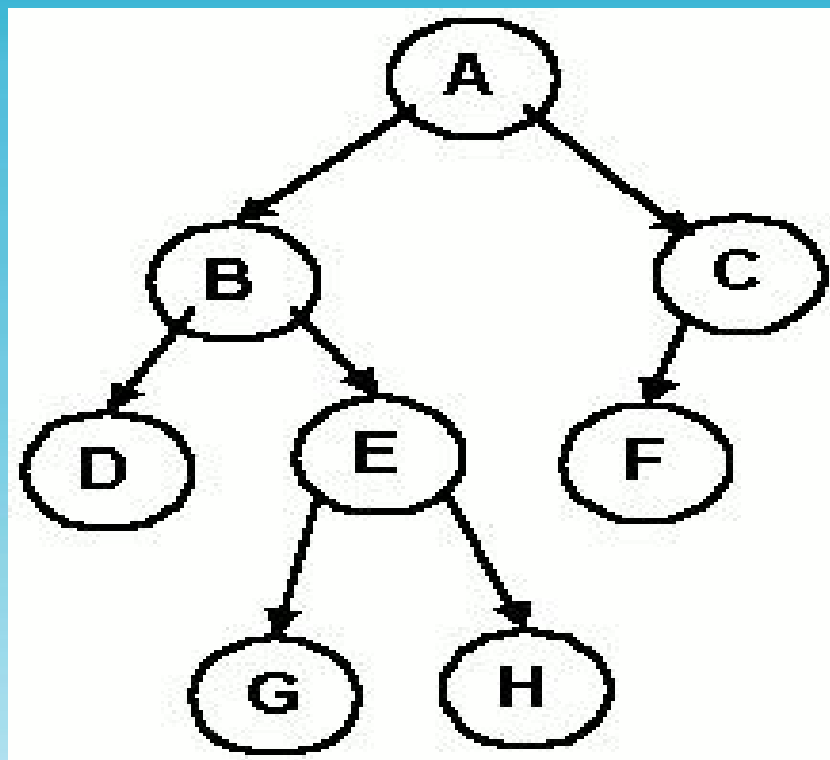
- (1) 访问根结点；
- (2) 按前序遍历次序遍历根结点的左子树；
- (3) 按前序遍历次序遍历根结点的右子树；




```
template <class T>
void BinaryTree<T>::preOrder(BinTreeNode<T>*subTree, void (*visit)(BinTreeNode<T> *p) )
{
    if ( subTree!=NULL ) // 二叉树非空
    {
        visit(subTree); // 对根结点进行访问
        preOrder(subTree->leftchild,visit);//对左子树进行遍历
        preOrder(subTree->rightchild,visit);//对右子树进行遍历
    }
} // preorder
```



如何把上述递归算法转化为等价的非递归算法？



方法一:



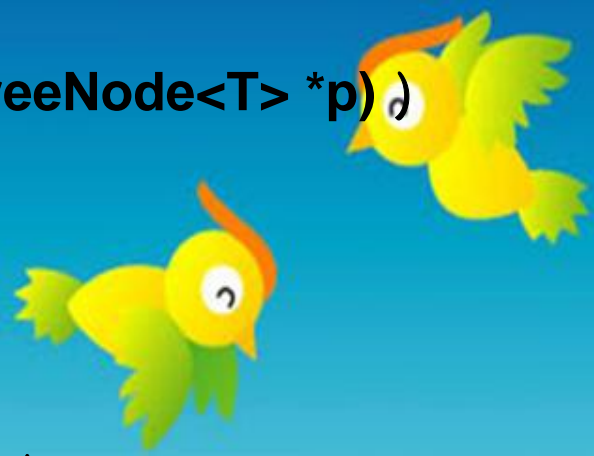
如果二叉树非空，则

- (1) 将二叉树的根结点作为当前结点;
- (2) 若当前结点非空，则先访问该结点，并将该结点进栈，再将其左孩子结点作为当前结点，重复步骤(2)，直到当前结点为空为止;
- (3) 若栈非空，则将栈顶结点出栈，并将当前结点的右孩子结点作为当前结点;
- (4) 重复步骤(2)、(3)，直到栈为空且当前结点为空为止。



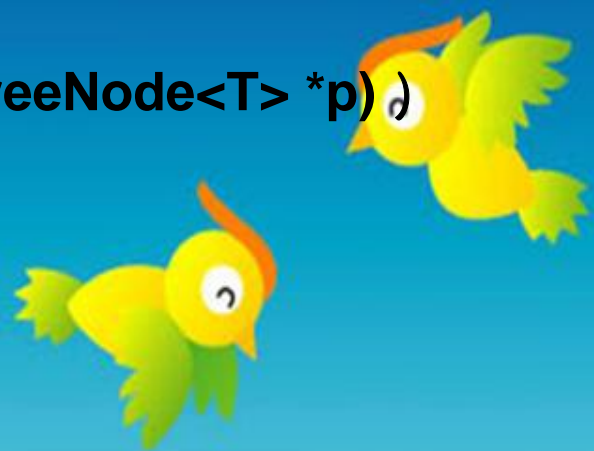
算法的实现如下：（方法一）

```
void BinaryTree<T>::preOrder( void (*visit)(BinTreeNode<T> *p) )
//前序遍历的非递归算法
{
    stack< BinTreeNode<T>*> s;
    // 定义一个以二叉树结点总数为容量的顺序栈对象
    t=root;
    while( ! s.IsEmpty() || t != NULL) //栈或当前指针非空
    {
        while ( t!=NULL) //当前指针非空
        {
            visit(t); s.Push( t );
            t=t->leftchild;
        }
        if( ! s.IsEmpty( ) )
        { s.Pop( t ); t=t->rightchild; }
    } // while
} // PreOrder
```



算法的实现如下: (方法一) STL stack

```
void BinaryTree<T>::preOrder( void (*visit)(BinTreeNode<T> *p) )  
//前序遍历的非递归算法  
{  
    stack<BinTreeNode<T>*>s;  
    //定义一个以二叉树结点总数为容量的顺序栈对象  
    t=root;  
    while( !s.empty() || t!=NULL) //栈或当前指针非空  
    {  
        while ( t!=NULL) //当前指针非空  
        {  
            visit(t); s.push(t);  
            t=t->leftchild;  
        }  
        if( !s.empty( ) )  
        { t=s.top();s.pop(); //出栈  
          t=t->rightchild;  
        }  
    } // while  
} // PreOrder
```



方法二:



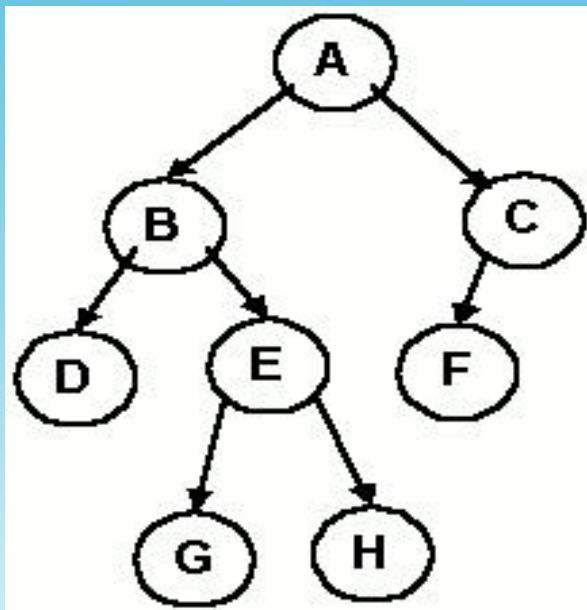
- (1) 如果二叉树非空，首先从二叉树的根结点作入口并先对其做访问，然后判断当前结点是否存在有左子树，如果有左子树，则以该左子树的根结点作为入口，重复步骤(1)，直到当前结点无左子树为止；
- (2) 接着判断该结点是否存在右子树，如果存在则以该右子树的根结点作为入口，重复步骤(1)；
- (3) 如果该结点的右子树遍历完或不存在右子树，则再退回到该结点的双亲结点，重复步骤(2)；
- (4) 如此反复，直到所有的结点都被进行了有且只有一次的访问。

算法过程可描述如下：

(方法二)

如果二叉树非空，则

- (1) 将二叉树的根结点进栈；
- (2) 若栈非空，则将栈顶结点出栈并对其实施访问；
- (3) 若该结点有右子树，则将右子树的根结点进栈；
- (4) 若该结点有左子树，则将左子树的根结点进栈；
- (5) 重复步骤 (2)、 (3)、 (4)，直到栈为空为止。



算法的实现如下： （方法二）

```
void BinaryTree<T>::preOrder( void (*visit)(BinTreeNode<T> *p) )  
//前序遍历的非递归算法  
{  
    stack<BinTreeNode<T>*> s;  
    //定义一个以二叉树结点数为容量的顺序栈对象  
    t=root;  
    s.Push(t); // 把根结点进栈  
    while ( ! s.IsEmpty( ) ) // 栈非空  
    { t=s.Pop(); // 栈顶结点出栈  
      visit(t);  
      if ( t->rightchild !=NULL )  
          s.Push(t->rightchild );  
      if ( t->leftchild !=NULL )  
          s.Push(t->leftchild );  
    } // while  
} // PreOrder
```

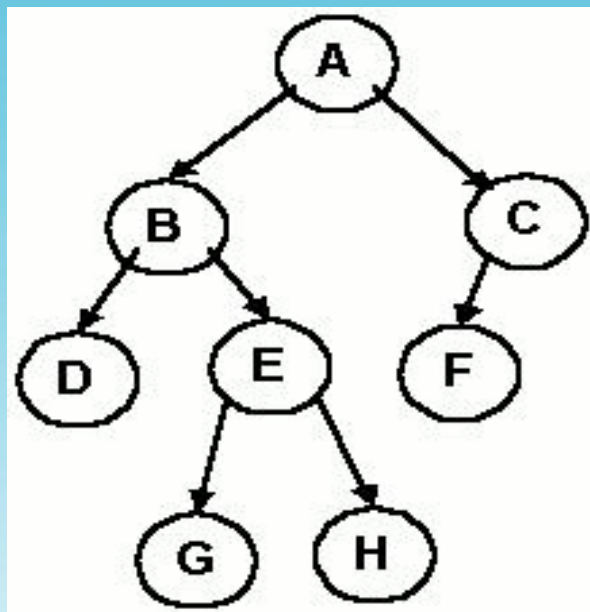


• 2.中序遍历

过程是：

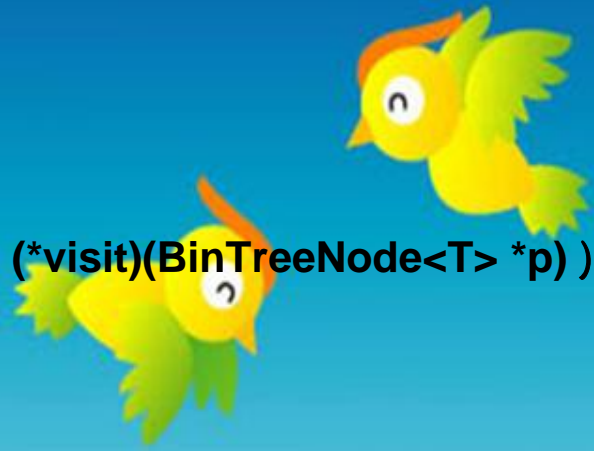
若二叉树非空，则做如下的操作：

- (1) 按中序遍历次序遍历根结点的左子树；
- (2) 访问根结点
- (3) 按中序遍历次序遍历根结点的右子树；

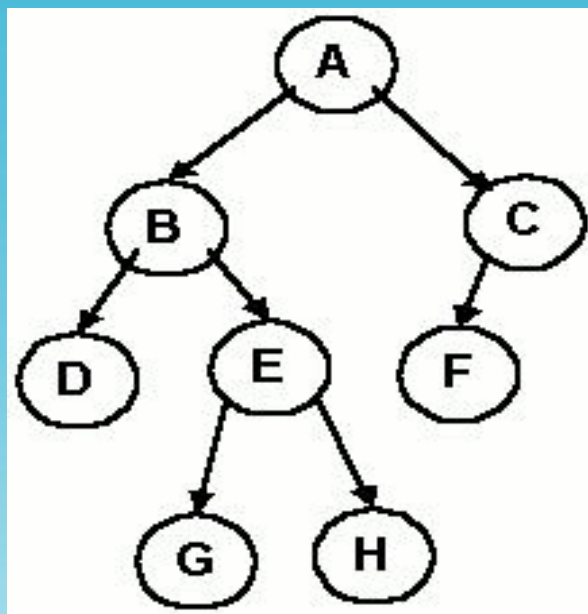


算法实现如下：

```
template <class T>
void BinaryTree<T>::inOrder( BinTreeNode<T> *subTree, void (*visit)(BinTreeNode<T> *p) )
{
    if ( subTree!=NULL ) // 二叉树非空
    {
        inorder(subTree->leftchild,visit);//对左子树进行遍历
        visit(subTree); // 对根结点进行访问
        inorder(subTree->rightchild,visit);//对右子树进行遍历
    }
} // InOrder
```



中序遍历二叉树的非递归算法:



中序遍历二叉树非递归过程可描述如下：

如果二叉树非空，则

- (1) 将二叉树的根结点作为当前结点；
- (2) 若当前结点非空，则该结点进栈并将其左孩子结点作为当前结点，重复步骤(2)，直到当前结点为空为止；
- (3) 若栈非空，则将栈顶结点出栈并作为当前结点，接着访问当前结点，再将当前结点的右孩子结点作为当前结点；
- (4) 重复步骤(2)、(3)，直到栈为空且当前结点为空为止。



Template <class T>

void BinaryTree<T>::inOrder(void (*visit)(BinTreeNode<T> *p))

//中序遍历的非递归算法

{

stack<BinTreeNode<T>*> s;

// 定义一个以二叉树的结点总数作为容量的顺序栈对象

t=root;

while(! s.IsEmpty() || t!=NULL) //栈或当前指针非空

{

while (t!=NULL) //当前指针非空

{

s.Push(t); t=t->leftchild;

}

if(! s.IsEmpty())

{

s.Pop(t);

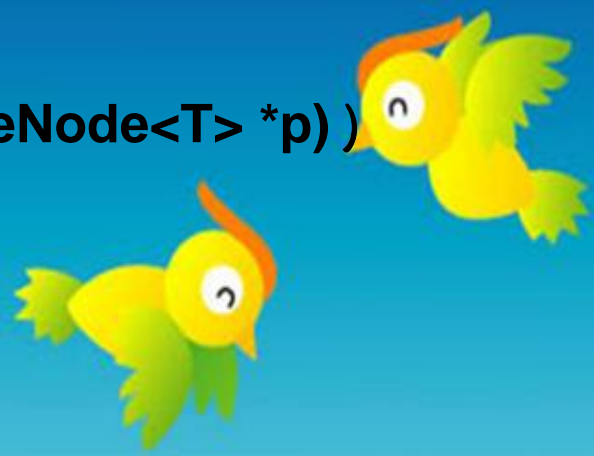
visit(t);

t=t->rightchild;

}

} // while

} // InOrder



Template <class T>

STL stack

```
void BinaryTree<T>::inOrder( void (*visit)(BinTreeNode<T> *p) )
```

```
//中序遍历的非递归算法
```

```
{
```

```
    stack<BinTreeNode<T>*> s;
```

```
// 定义一个以二叉树的结点总数作为容量的顺序栈对象
```

```
    t=root;
```

```
    while( ! s.empty() || t!=NULL) //栈或当前指针非空
```

```
{
```

```
    while ( t!=NULL) //当前指针非空
```

```
{
```

```
        s.push(t);    t=t->leftchild;
```

```
}
```

```
    if( ! s.empty( ))
```

```
{
```

```
        t=s.top( ); s.pop( );
```

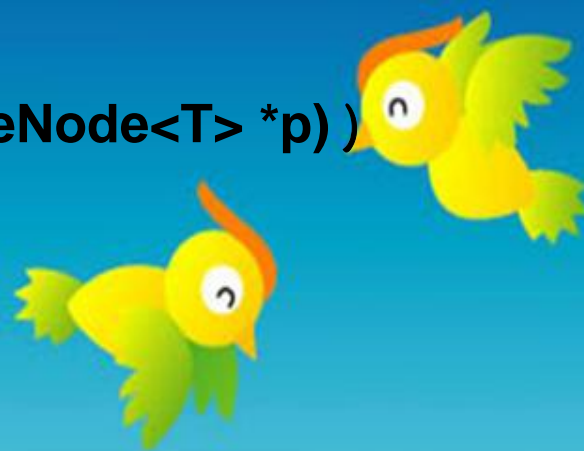
```
        visit(t);
```

```
        t=t->rightchild;
```

```
}
```

```
} // while
```

```
} // InOrder
```

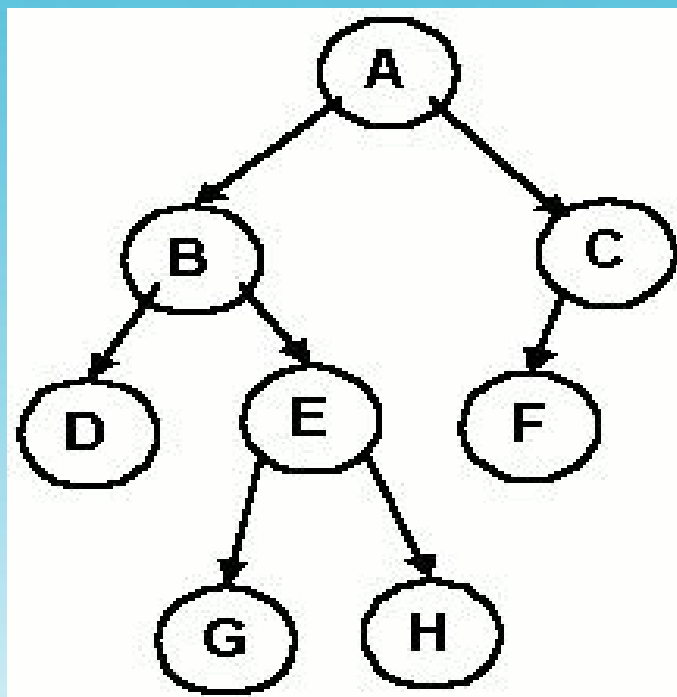


• 3.后序遍历

过程是：

若二叉树非空，则做如下的操作：

- (1) 按后序遍历次序遍历根结点的左子树；
- (2) 按后序遍历次序遍历根结点的右子树；
- (3) 访问根结点



Template <class T>

```
void BinaryTree<T>::postOrder( BinTreeNode *subTree, void (*visit)(BinTreeNode<T> *p) )
```

```
{
```

```
if ( subTree!=NULL ) // 二叉树非空
```

```
{
```

```
    postOrder( subTree->leftchild,visit) ;//对左子树进行遍历
```

```
    postOrder( subTree->rightchild,visit );//对右子树进行遍历
```

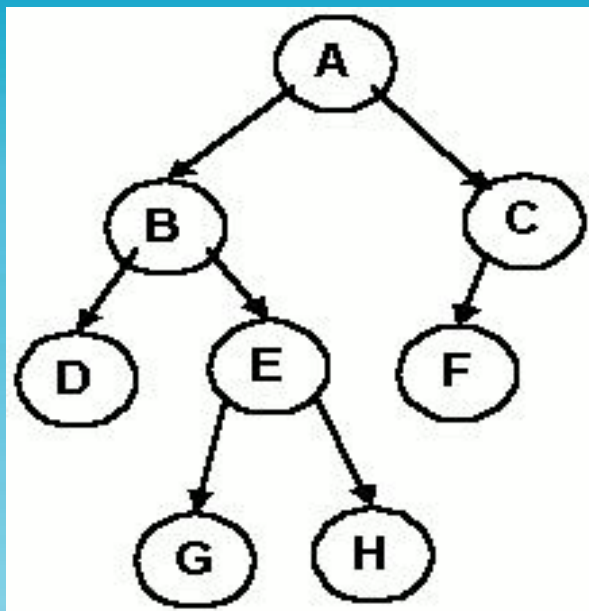
```
    visit(subTree); // 对根结点进行访问
```

```
}
```

```
} // PostOrder
```



- 后序遍历二叉树的非递归执行过程：



- 特点：每个结点要进两次栈。



结论:



用非递归算法进行后序遍历二叉树时，一个结点需要进栈、出栈各两次。

为了区别同一个结点的两次进栈，需要设置一个标志(flag)。

```
flag= { 0 第一次进栈，表示该结点出栈之后不能访问  
      {  
      | 1 第二次进栈，表示该结点出栈之后可以访问
```

```
void BinaryTree<T>::PostOrder(void (*visit)(BinTreeNode<T> *p) )
```

```
//后序遍历的非递归算法
```

```
{
```

```
    stack< BinTreeNode<T>*> s1;    stack<int> s2;
```

```
    // 定义一个以二叉树的结点总数作为容量的顺序栈对象,
```

```
    //栈结点结构应该能保存要返回的结点及该结点的进栈标志值。
```

```
    t=root;
```

```
    while( (! s1.IsEmpty()) || ( t!=NULL)) // 栈非空或当前结点非空
```

```
{ while ( t!=NULL) // 当前结点非空
```

```
    { s1.Push( t ); s2.Push( 0 ); //当前结点和第一进栈标志进栈s
```

```
        t=t->leftchild; //以当前结点的左孩子作为当前结点
```

```
    }
```

```
    if ( ! s.IsEmpty() ) // 栈s非空
```

```
    { s1.Pop( t ); s2.Pop( flag ); // s1,s2栈顶结点出栈
```

```
        if ( flag== 1 ) //该结点是第二进栈而出来的
```

```
        { visit(t);    t=NULL; } // 访问结点
```

```
        else
```

```
        { s1.Push( t ); s2.Push( 1 ); // 结点第二次进栈s
```

```
            t=t->rightchild; //以当前结点的右孩子作为当前结点
```

```
        } // else
```

```
    } // if
```

```
} // while
```

```
} // PostOrder
```



```
void BinaryTree<T>::postOrder(void (*visit)(BinTreeNode<T> *p))
```

STL stack

//后序遍历的非递归算法

```
{ struct node { BinTreeNode<T> *tp; bool flag; } x;
```

stack<struct node> s; // 定义一个以二叉树的结点总数作为容量

//的顺序栈对象,栈结点结构应该能保存要返回的结点及该结点的进栈标志值。

```
t=root;
```

```
while( (! s.empty()) || ( t!=NULL)) // 栈非空或当前结点非空
```

```
{ while ( t!=NULL ) // 当前结点非空
```

```
{ x.tp=t; x.flag=0;
```

```
s.push( x );//当前结点和第一进栈标志进栈s
```

```
t=t->leftchild;//以当前结点的左孩子作为当前结点
```

```
}
```

```
if ( ! s.empty() ) // 栈s非空
```

```
{ x=s.top(); s.pop(); // s栈顶结点出栈
```

```
t=x.tp;
```

```
if ( x.flag== 1 ) //该结点是第二进栈而出来的
```

```
{ visit(t); t=NULL; }// 访问结点
```

```
else
```

```
{ x.flag=1; s.push(x); // 结点第二次进栈s
```

```
t=t->rightchild;//以当前结点的右孩子作为当前结点
```

```
} // else
```

```
} // if
```

```
} // while
```

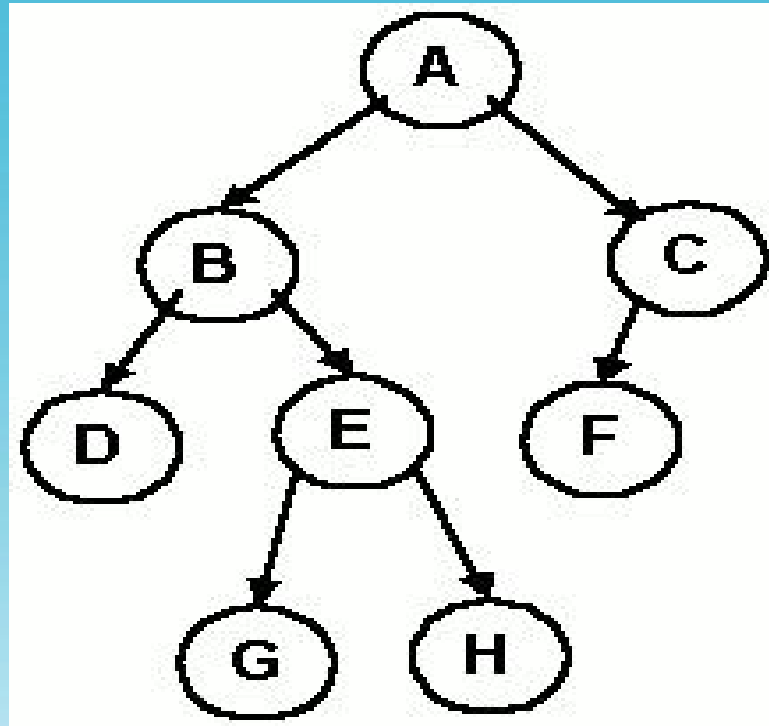
```
} // PostOrder
```



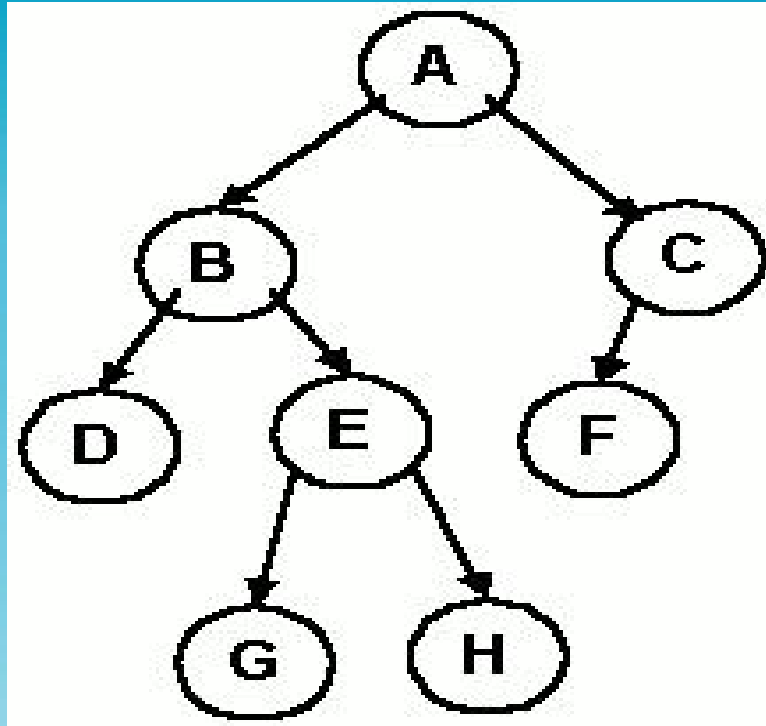
非递归遍历算法的总结



- 算法的统一



- 4 层次遍历



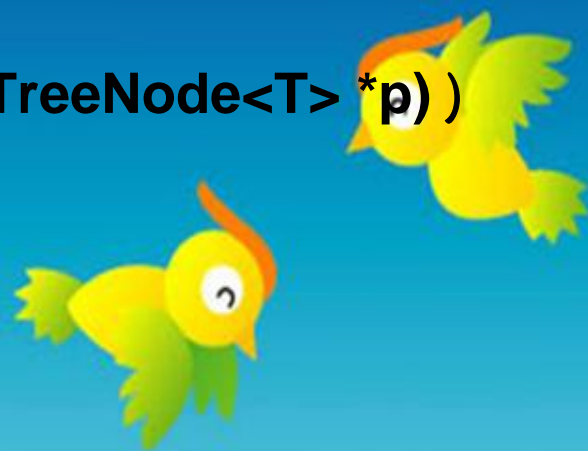
层次遍历的过程



- (1) 遍历进行之前先将二叉树的根结点存入队列中，
- (2) 然后依次从队列中取出队头结点，每取出一个结点，都先访问该结点，
- (3) 接着分别检查该结点是否存在左、右孩子，若存在则先后入列，
- (4) 如此反复，直到队列为空为止。

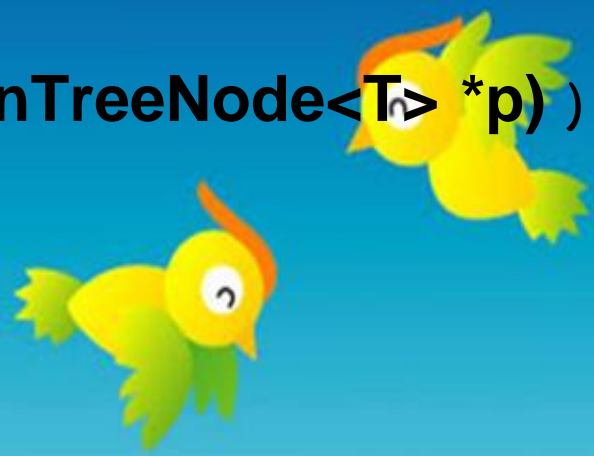
层次遍历的算法实现如下:

```
void BinaryTree<T>::levelOrder(void (*visit)(BinTreeNode<T> *p))  
// 对二叉树进行层次遍历  
{  
    queue< BinTreeNode<T>*> q;  
    //定义一个以二叉树结点数为容量的顺序队列对象  
    t=root;  
    if ( t!=NULL )  
        q.Enqueue(t ); // 二叉树的根结点入列  
    while ( ! q.IsEmpty() ) // 队列非空  
    { t=q.DeQueue(); // 队头结点出队  
      visit(t); // 对结点t进行访问  
      if(t->leftchild!=NULL)//左孩子非空, 则入列  
          q.Enqueue(t->leftchild);  
      if(t->rightchild!=NULL)//右孩子非空, 则入列  
          q.Enqueue(t->rightchild);  
    } // while  
} // LevelOrder
```



层次遍历的算法实现如下: **STL queue**

```
void BinaryTree<T>::levelOrder(void (*visit)(BinTreeNode<T> *p) )  
// 对二叉树进行层次遍历  
{  
    queue<BinTreeNode<T>*>q;  
    //定义一个以二叉树结点数为容量的队列对象  
    t=root;  
    if ( t!=NULL )  
        q.push(t); // 二叉树的根结点入列  
    while ( ! q.empty() ) // 队列非空  
    { t=q.front(); q.pop(); // 队头结点出队  
      visit(t); // 对结点进行访问  
      if(t->leftchild!=NULL)//左孩子非空, 则入列  
          q.push( t->leftchild );  
      if(t->rightchild!=NULL)//右孩子非空, 则入列  
          q.push(t->rightchild);  
    } // while  
} // LevelOrder
```



- 5 二叉树遍历的应用

- 访问的重新理解

- 对该结点的增加、删除、修改、查阅或加工等各种操作的抽象。

- 因此，二叉树的遍历经常用于解决实际问题。



(1) 以二叉链表作存储结构，编写一个算法
将二叉树左、右子树进行交换。

```
void PreOrder( BinTreeNode *t)
```

```
{
```

```
    if ( t!=NULL ) // 二叉树非空
```

```
    {
```

```
        x=t->leftchild; t->leftchild=t->rightchild; t->rightchild=x;
```

```
        PreOrder( t->leftchild); //对左子树进行遍历
```

```
        PreOrder( t->rightchild); //对右子树进行遍历
```

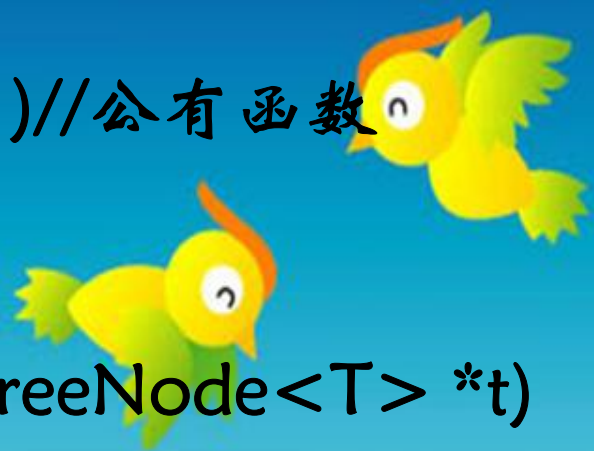
```
    }
```

```
} // PreOrder
```



```
void BinaryTree<T>::Exchange( void )//公有函数
{ Exchange(root); }
```

```
void BinaryTree<T>::Exchange( BinTreeNode<T> *t)
{ //私有函数
  if ( t!=NULL ) // 二叉树非空
  {
    x=t->leftchild; t->leftchild=t->rightchild; t->rightchild=x;
    Exchange( t->leftchild);//对左子树进行遍历
    Exchange( t->rightchild);//对右子树进行遍历
  }
} //exchange
```



(2)以二叉链表作存储结构，编写一个算法以求出该二叉树的叶子总数。



```
void PreOrder( BinTreeNode *t)
```

```
{
```

```
    if ( t!=NULL ) // 二叉树非空
```

```
    {
```

```
        if ( t->leftchild==NULL && t->rightchild==NULL) counter++;
```

```
        PreOrder( t->leftchild);//对左子树进行遍历
```

```
        PreOrder( t->rightchild);//对右子树进行遍历
```

```
    }
```

```
} // PreOrder
```

//算法1

```
void BinaryTree<T>::Leaves( void ) //公有函数  
{ Leaves(root); }
```

int counter=0; //全局变量

```
void BinaryTree<T>:: Leaves( BinTreeNode<T> *t)  
{  
    if ( t!=NULL ) // 二叉树非空  
    {  
        if ( t->leftchild==NULL && t->rightchild==NULL)  
            counter++;  
        Leaves( t->leftchild); //对左子树进行统计  
        Leaves( t->rightchild); //对右子树进行统计  
    }  
} //Leaves
```



//算法2

```
int BinaryTree<T>::Leaves( void ) //公有函数
{ int n=0; Leaves(root,n); return n; }
```

```
void BinaryTree<T>::Leaves( BinTreeNode<T> *t, int &counter)
```

```
//私有
```

```
{
```

```
if ( t!=NULL ) // 二叉树非空
```

```
{
```

```
    if ( t->leftchild==NULL && t->rightchild==NULL)
```

```
        counter++;
```

```
    Leaves( t->leftchild,counter);//对左子树进行统计
```

```
    Leaves( t->rightchild,counter);//对右子树进行统计
```

```
}
```

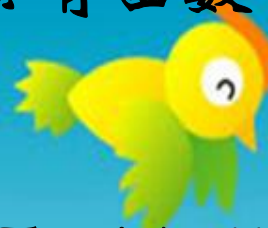
```
} //Leaves
```



//算法3

```
int BinaryTree<T>::Leaves( void ) // 公有函数  
{ return Leaves(root); }
```

```
int BinaryTree<T>::Leaves( BinTreeNode<T> *t) // 私有函数  
{  
    if ( t!=NULL ) // 二叉树非空  
    {  
        if ( t->leftchild==NULL && t->rightchild==NULL)  
            return 1;  
        else  
            return Leaves( t->leftchild)+Leaves( t->rightchild);  
    }  
} //Leaves
```



(3) 树中结点个数的统计

```
void PreOrder( BinTreeNode *t)
```

```
{
```

```
    if ( t!=NULL ) // 二叉树非空
```

```
    {
```

```
        cout<<t->data;  // 对根结点进行访问
```

```
        PreOrder( t->leftchild); //对左子树进行遍历
```

```
        PreOrder( t->rightchild); //对右子树进行遍历
```

```
    }
```

```
} // PreOrder
```



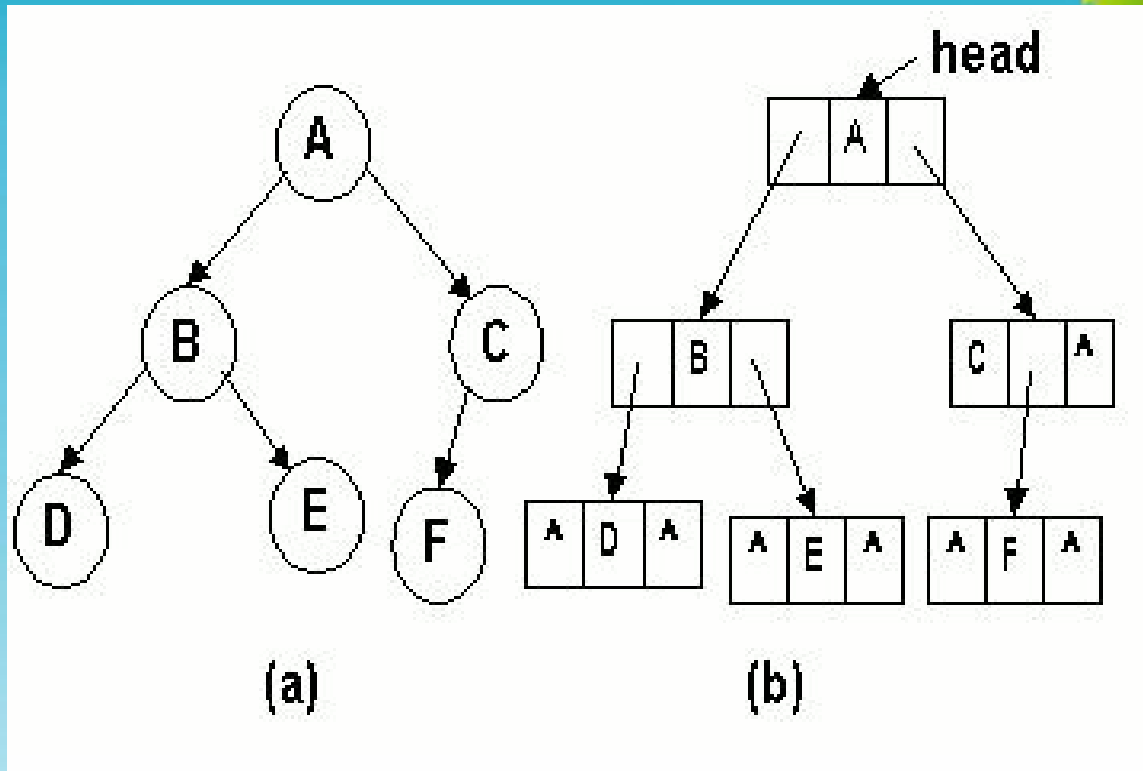
(3) 结点个数的统计

```
int BinaryTree<T>::Size( void )  
{ return Size(root); }
```

```
int BinaryTree<T>::Size (BinTreeNode<T> *T)  
{  
    if (T == NULL) return 0;    // 空树  
    else  
        return 1+Size(T->leftChild)+Size(T->rightChild);  
}
```



- (4) 二叉树的生成 (二叉链式结构的建立)



面向对象的算法

```
Void BinaryTree<T>::Create(void) //public成员函数
{
    root=CreateBTree();
}
```

```
BinTreeNode<T> *BinaryTree<T>::CreateBTree(void) //private成员函数
{ char ch;
  BinTreeNode<T> *current;
  cin>>ch;
  if (ch=='.') // 输入的是否为符号'.'
      return NULL; // 建立空树
  else
  { current=new BinTreeNode<T>; // 分配结点空间
    current->data=ch; // 填入结点值
    current->leftchild=CreateBTree();//建立左子树
    current->rightchild=CreateBTree();//建立右子树
    return current;
  }
} // Create
```



中序建立算法



```
BinTreeNode<T> *BinaryTree<T>::CreateBTree(void)
```

```
//private成员函数
```

```
{ char ch;
```

```
BinTreeNode<T> *q,*current;
```

```
cin>>ch;
```

```
if (ch=='.' ) // 输入的是否为符号'.'
```

```
return NULL; // 建立空树
```

```
else
```

```
{ q=CreateBTree();//建立左子树
```

```
current=new BinTreeNode<T>; // 分配结点空间
```

```
current->data=ch; // 填入结点值
```

```
current->leftchild=q; //建立左子树
```

```
current->rightchild=CreateBTree(); // 建立右子树
```

```
return current;
```

```
}
```

```
} // Create
```



后序建立算法

```
BinTreeNode<T> *BinaryTree<T>::CreateBTree(void)
```

```
//private成员函数
```

```
{ char ch;
```

```
  BinTreeNode<T> *tl,*tr,*current;
```

```
  cin>>ch;
```

```
  if (ch=='.' ) // 输入的是否为符号'.'
```

```
    return NULL; // 建立空树
```

```
  else
```

```
  {  tl=CreateBTree(); //建立左子树
```

```
    tr=CreateBTree(); // 建立右子树
```

```
    current=new BinTreeNode<T>; // 分配结点空间
```

```
    current->data=ch; // 填入结点值
```

```
    current->leftchild=tl;
```

```
    current->rightchild= tr;
```

```
    return current;
```

```
  }
```

```
} // Create
```

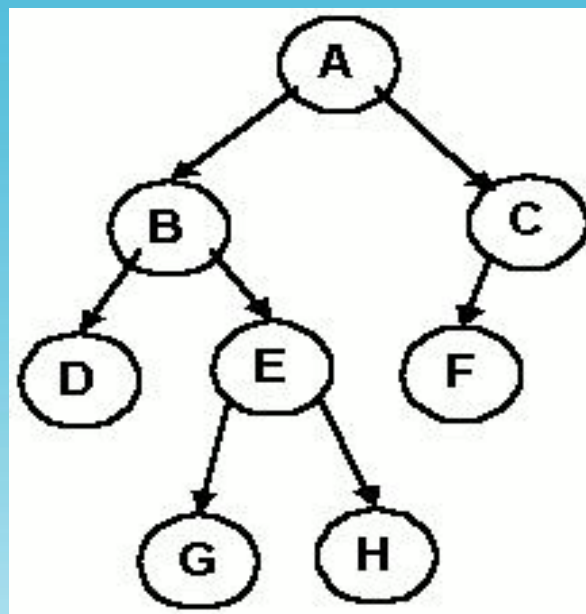


层次式建立算法



广义表 \rightarrow 二叉树

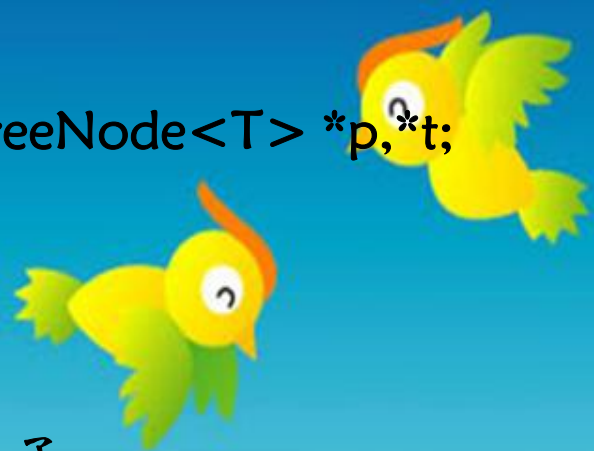
- $A(B(D,E(G,H)),C(F))$



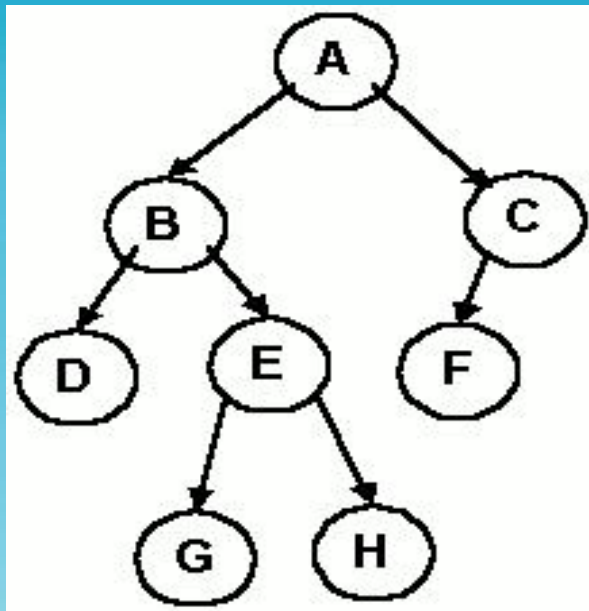

```

Void BinaryTree<T>::Create2(void) //public成员函数
{ stack<BinTreeNode<T>*> s; int k; char ch; BinTreeNode<T> *p,*t;
  root=NULL;
  cin>>ch; // 读入第一个符号
  while ( ch!='.' )
  {   switch(ch)
      { case '(': s.Push(p); k=1; break; // 左孩子准备来了
        case ',': k=2;break; // 右孩子准备来了
        case ')': s.Pop(t); break; // 这个结点的孩子处理完了
        default: p=new BinTreeNode<T>; p->data=ch; // 是结点值就生成
                  p->leftchild=p->rightchild=NULL;
                  if ( root==NULL) root=p; // 第一个结点吗？ 是则为根
                  else // 不是第一个结点
                      if ( k==1 ) // 作为左孩子吗？
                          { s.getTop(t); t->leftchild=p; } //做左孩子
                      else
                          { s.getTop(t); t->rightchild=p; } //做右孩子
                  }
      }
  cin>>ch;
}
}

```



二叉树 \rightarrow 广义表



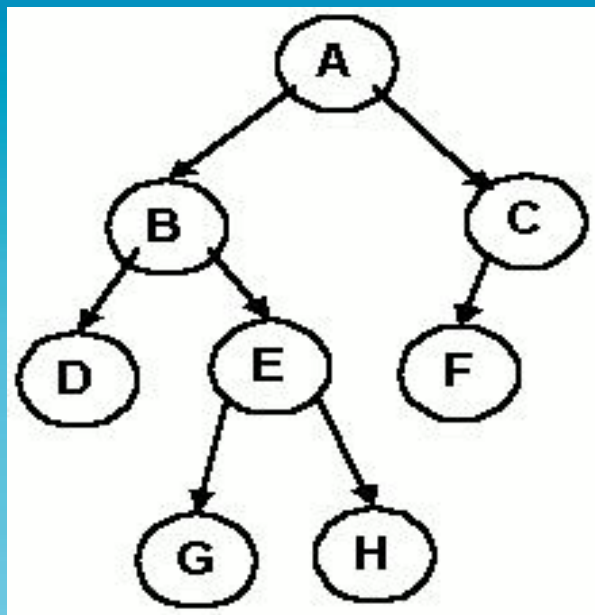
- $A(B(D, E(G, H)), C(F))$

```
void BinaryTree<T>::Print( void ) //public成员函数
{   print( root ); }
```

```
int BinaryTree<T>::Print( BinTreeNode<T> *t) //private成员函数
{
    if ( t!=NULL)
    {   cout<<t->data;    // 先输入当前结点值
        if ((t->leftchild!=NULL)|| (t->rightchild==NULL)) //它有孩子吗?
        {   cout<<'('; // 有孩子就要先输出(
            Print( t->leftchild);    // 输出左子树
            if (t->leftright!=NULL)           // 有右子树吗?
            {   cout<<',';                // 有右子树才输出,
                Print(t->rightchild); // 输出右子树
            }
            cout<<')';    // 输出括号)
        }
    }
}
```



- (5) 求二叉树深度



- **方法一**: 根据深度的概念, 可知道对**层次遍历**算法进行修改便可得到问题的解决。
- **方法二**: 若一棵二叉树为空, 则它的深度0, 否则求二叉树深度的递归定义为: 它等于左子树和右子树中最大深度加1。 (**后序遍历**)

//面向对象算法1

```
int BinaryTree<T>::Depth( void ) //public成员函数
{
    return Depth(root);
}
```

```
int BinaryTree<T>::Depth( BinTreeNode<T> *t) //private成员函数
{
    int CurrentDepth,LeftDepth,RightDepth;
    if ( t==NULL )
        return 0; // 二叉树为空，则深度为0
    else {
        LeftDepth=Depth( t->leftchild );//求左子树的深度
        RightDepth=Depth( t->rightchild);//求右子树的深度
        if ( LeftDepth > RightDepth ) //比较左、右子树的深度
            return LeftDepth + 1;//改变当前结点的深度
        else
            return RightDepth + 1;
    }
} // Depth
```



与二叉树遍历应用相关的习题



- 1.设计一个实现一棵二叉树复制的算法。

二叉树的建立算法（前序）

```
BinTreeNode<T> *BinaryTree<T>::CreateBTree(void)
//private成员函数
{ char ch;
  BinTreeNode<T> *current;
  cin >> ch;
  if (ch == '.') // 输入的是否为符号'.'
    return NULL; // 建立空树
  else
  { current = new BinTreeNode<T>; // 分配结点空间
    current->data = ch; // 填入结点值
    current->leftchild = CreateBTree(); // 建立左子树
    current->rightchild = CreateBTree(); // 建立右子树
    return current;
  }
} // Create
```



- 2. 设具有 n 个结点的完全二叉树采用顺序存储结构，试写一个算法将该顺序存储结构转换为二叉链式存储结构。



二叉树的建立算法（前序）

```
BinTreeNode<T> *BinaryTree<T>::CreateBTree(void)
//private成员函数
{ char ch;
  BinTreeNode<T> *current;
  cin >> ch;
  if (ch == '.') // 输入的是否为符号'.'
    return NULL; // 建立空树
  else
  { current = new BinTreeNode<T>; // 分配结点空间
    current->data = ch; // 填入结点值
    current->leftchild = CreateBTree(); // 建立左子树
    current->rightchild = CreateBTree(); // 建立右子树
    return current;
  }
} // Create
```



- 3. 一棵具有 n 个结点的完全二叉树存放在二叉树的顺序存储结构中，试编写非递归算法对该树进行前序遍历。
- 4. 试编写算法判别两棵二叉树是否等价。如果 $T1$ 和 $T2$ 都是空二叉树，或 $T1$ 和 $T2$ 的根结点的值相同，并且 $T1$ 的左子树与 $T2$ 的左子树是等价的； $T1$ 的右子树与 $T2$ 的右子树是等价的。



- 5.试写一个算法，在一棵二叉链式作存储结构的二叉树中求这样的结点，它在前序遍历序列中处于第 j 个位置。
- 6.设具有 n 个结点的二叉树采用二叉链式存储结构，试写出一个算法将该二叉链式存储结构转换为顺序存储结构。





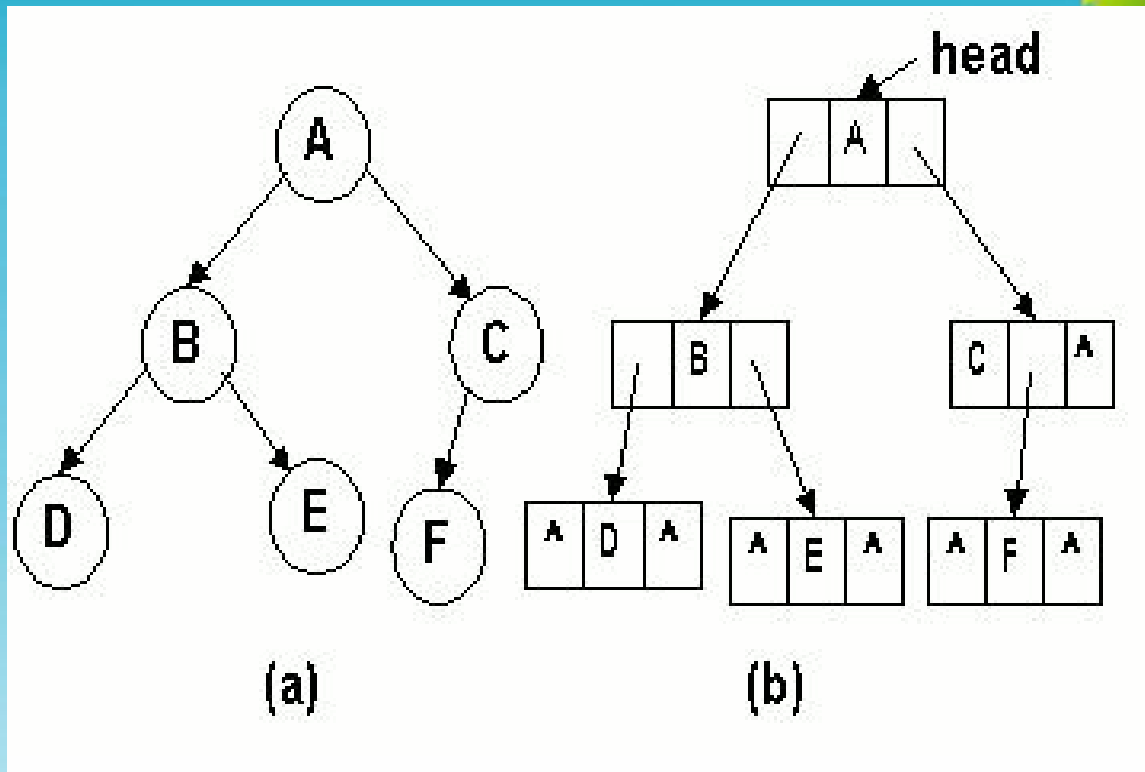
- 7. 现有以下按前序和中序遍历二叉树的结果，问这样能否唯一地确定这棵二叉树的形状？为什么？

前序序列：ABCDEFghi

中序序列：BCAEDGHFI

- 线索二叉树

- 为什么需要引入线索二叉树?



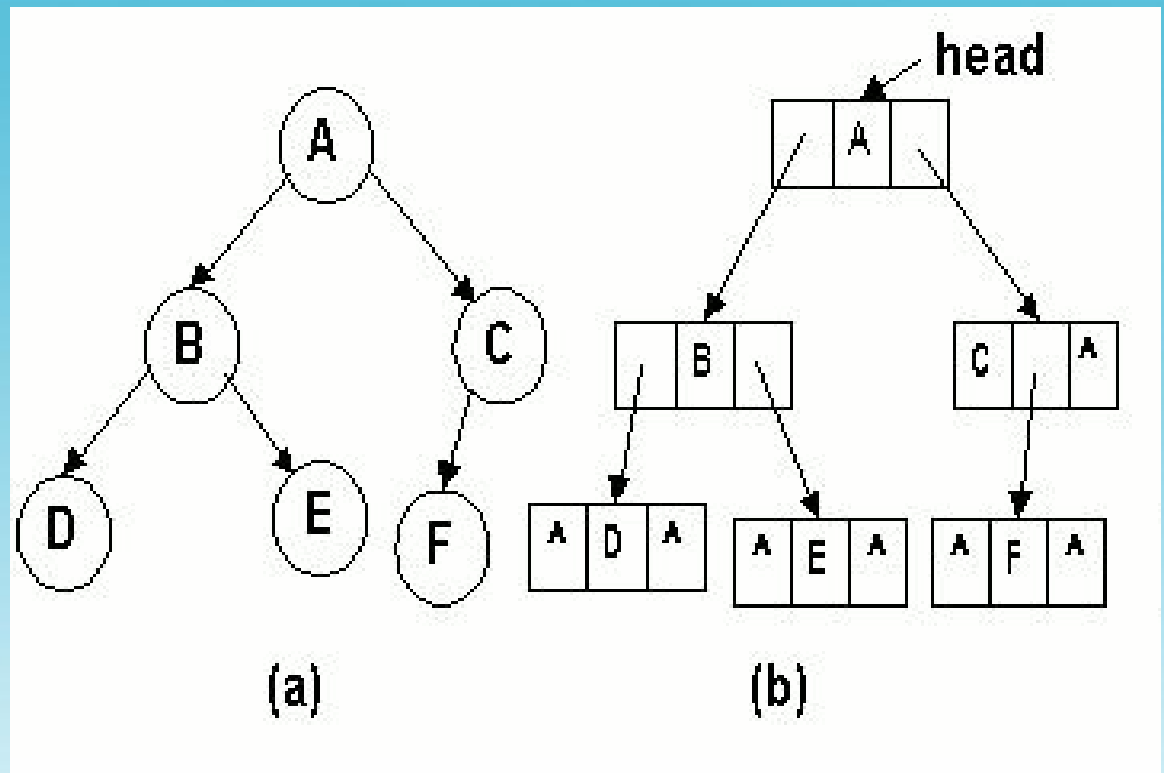


- 在二叉树中,从任一结点出发只能直接找到该结点的左、右孩子,而在一般情况下是无法直接找到该结点在某种遍历序列中的直接前趋和直接后继。——**查找不方便**
- 具有 n 个结点的二叉链表中必定存在 $n+1$ 个空指针域——**浪费空间**

基本概念



- 左线索
- 右线索
- 线索二叉树



线索二叉树的存储结构



方法:在二叉链式结构的基础上做适当的修改。



即在每个结点上增加两个线索标志域：一个是左线索标志域（用ltag来表示），一个是右线索标志域（用rtag来表示）。

ltag= {
 0 左指针域指向的结点是左孩子
 1 左指针域指向的结点是在某种遍历次序下的直接前趋
 }
 rtag= {
 0 右指针域指向的结点是右孩子
 1 右指针域指向的结点是在某种遍历次序下的直接后继
 }
 因此线索二叉树中结点存储结构如下:

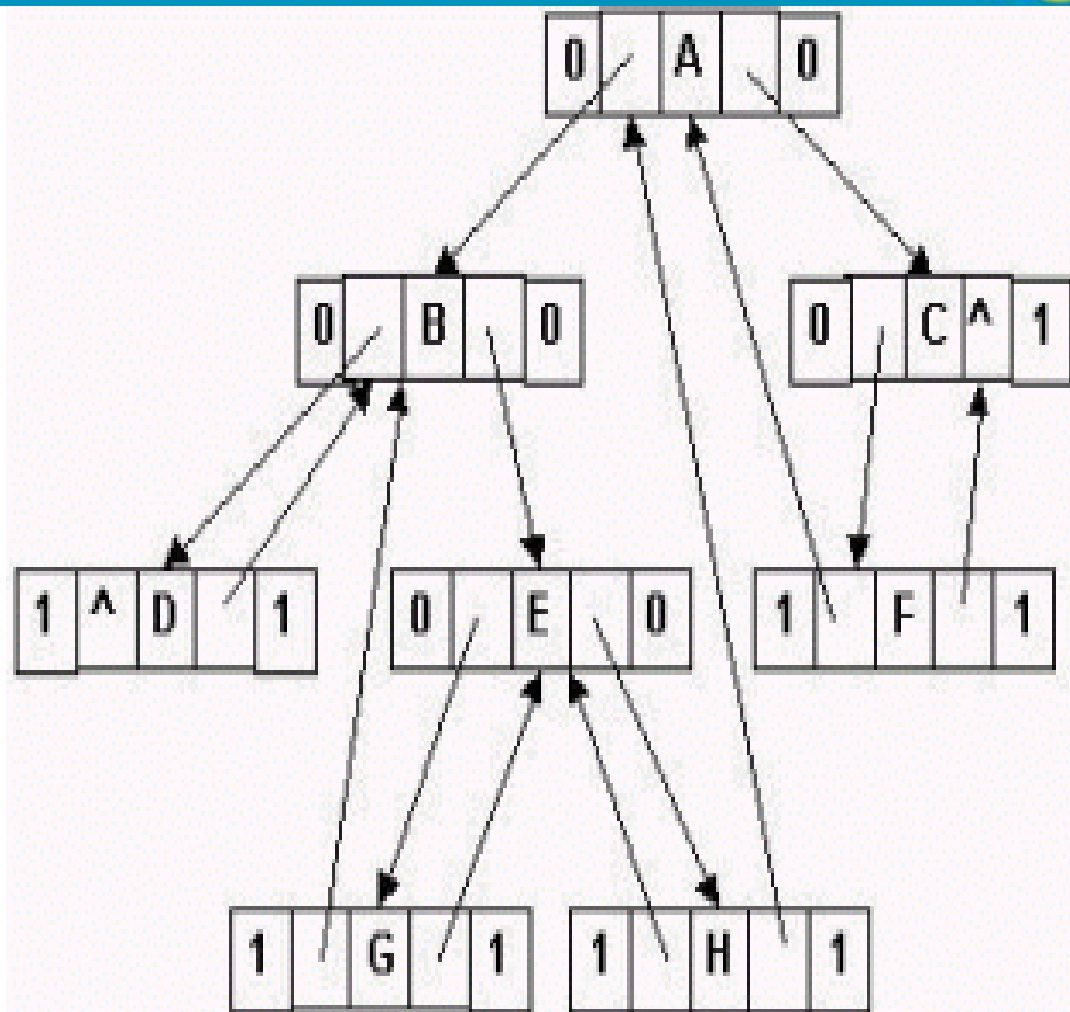
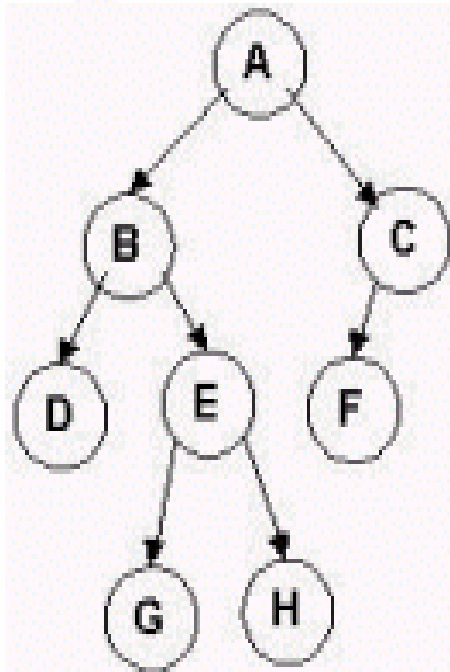
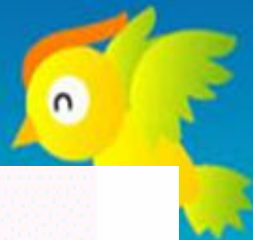
| | | | | |
|--------|------|------|------|--------|
| lchild | ltag | data | rtag | rchild |
|--------|------|------|------|--------|

于是线索二叉树中的结点的存储结构可定义如下:

```

typedef enum { Link, Thread } PointerTag; // =0 指针; =1 线索
typedef struct BiThrNode
{
    TElemType data;
    PointerTag ltag;
    struct BiThrNode *lchild;
    PointerTag rtag;
    struct BiThrNode *rchild;
}BiThrNode,*BiThrTree;
  
```





树和森林的遍历



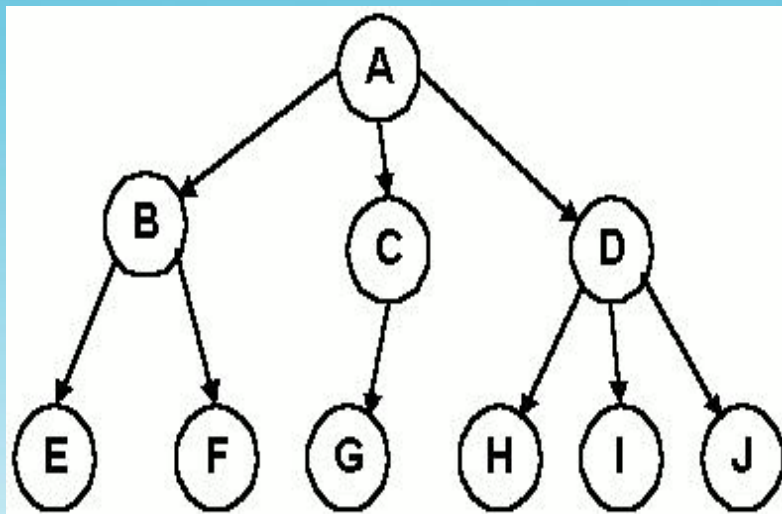
• 一、树的遍历

1. 前序遍历

若树非空，则

(1) 访问根结点；

(2) 按照从左到右的顺序依次前序遍历每棵子树；



• 2. 后序遍历

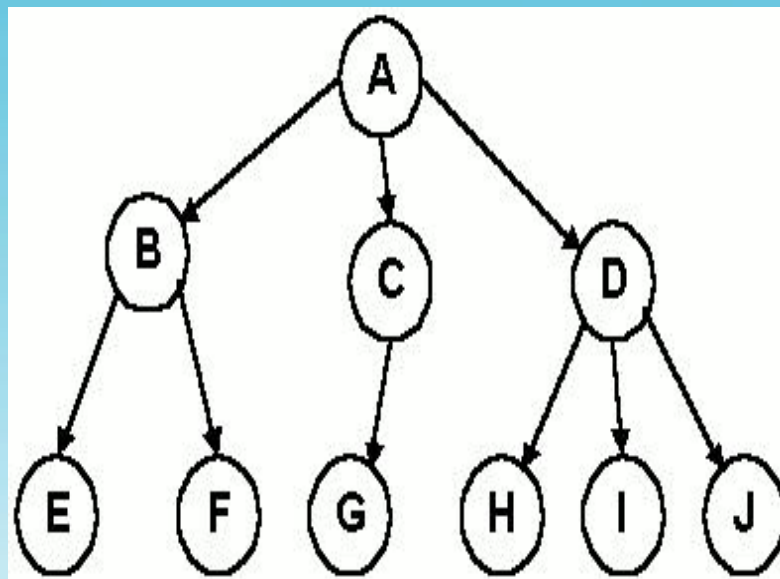
- 若树非空，则

(1) 按照从左到右的顺序依次后序遍历每棵子树；

(2) 访问根结点。

• 3. 层次遍历

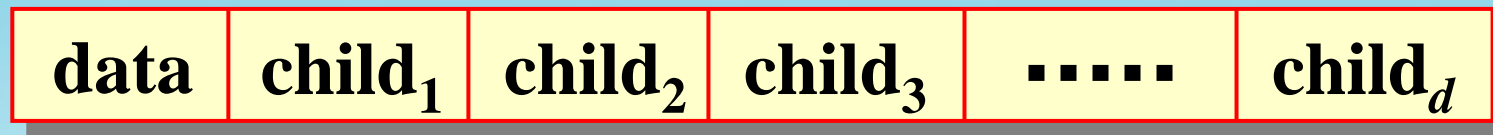
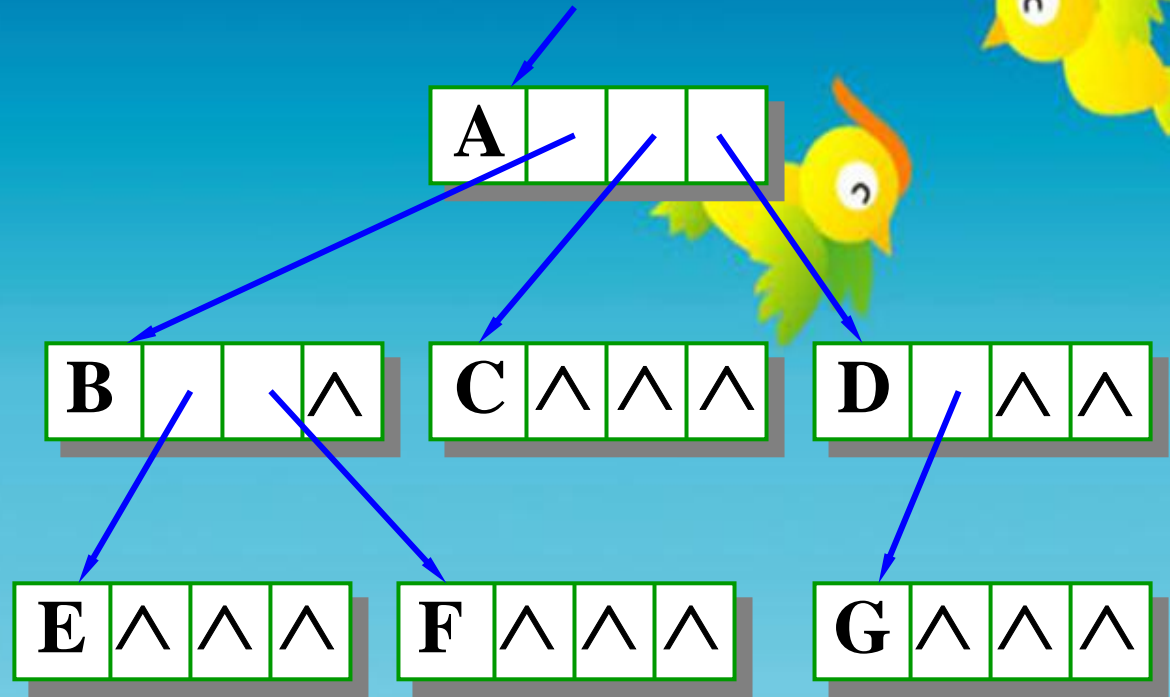
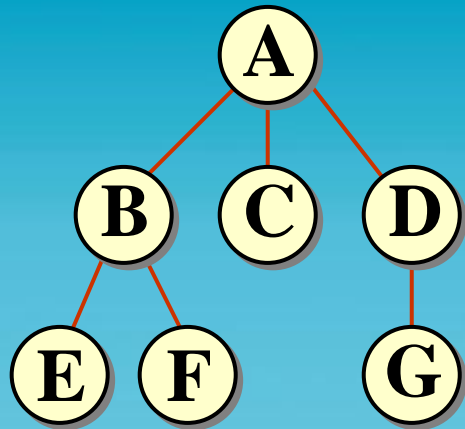
- 若树非空，则先访问根结点（第一层上的结点），再按从左到右的顺序访问第二层上的结点，依次按层访问，直到树中的全部结点都被访问为止。



算法的实现

- (1) 存储结构
- (2) 算法



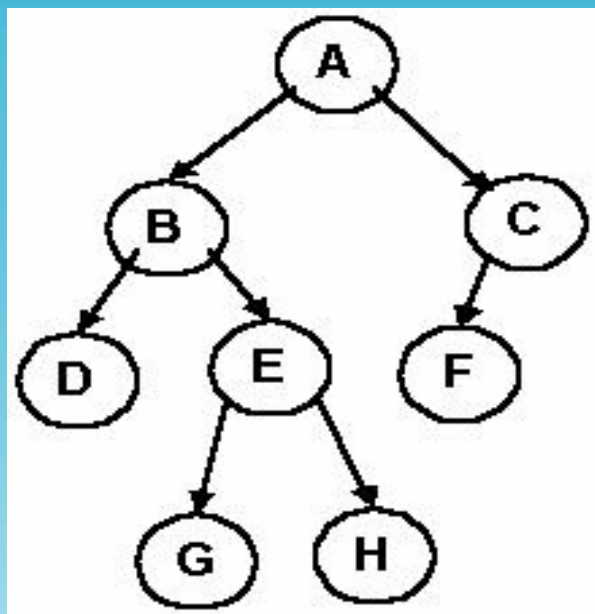


存储结构



```
struct node
{
    int data;
    struct node *child[3];
};
```

二叉树前序遍历算法



存储结构



```
struct node  
{  
    int data;  
    struct node *child[2];  
};
```

前序遍历算法

```
void PreOrder(struct node *root)
{
    int i;

    if (root!=0 )
    {
        cout<<root->data<<" ";
        for (i=0;i<2;i++)
            DFS(root->child[i]);

        //DFS(root->child[0]);
        //DFS(root->child[1]);
    }
}
```



层次遍历算法



二叉树层次遍历算法



```
void BFS(struct node *root)
{
    queue<struct node *> qu;    struct node *temp;

    qu.push(root);
    while(!qu.empty())
    {
        temp=qu.front();
        qu.pop();
        cout<<temp->data<<" ";

        if (temp->child[0]!=0)
            qu.push(temp->child[0]);

        if (temp->child[1]!=0)
            qu.push(temp->child[1]);
        // 可以改写为 for 循环
    }
}
```

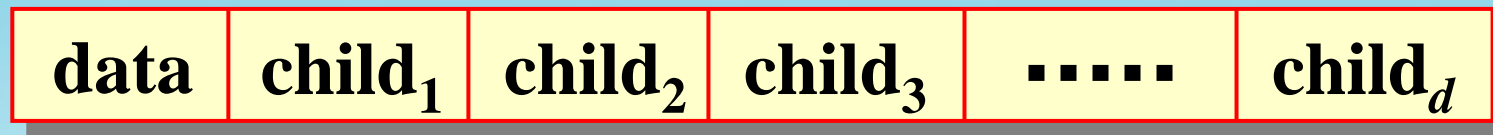
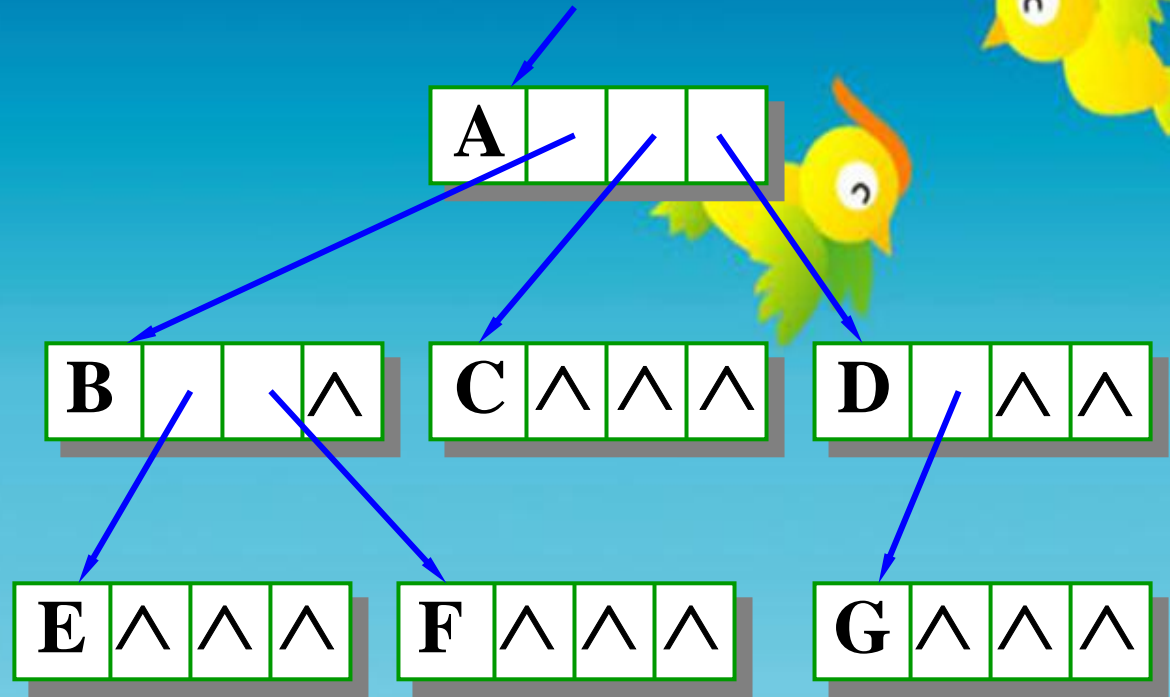
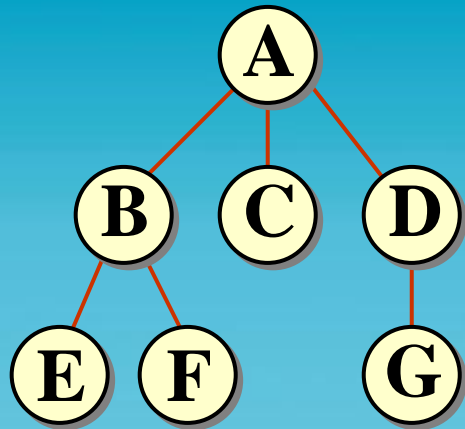
二叉树的层次遍历算法



```
void BFS(struct node *root)
{
    queue<struct node *> qu;
    struct node *temp;

    qu.push(root);
    while(!qu.empty())
    {
        temp=qu.front();
        qu.pop();
        cout<<temp->data<<" ";

        for(i=0;i<2;i++)
            if (temp->child[i]!=0)
                qu.push(temp->child[i]);
    }
}
```



存储结构



```
struct node
{
    int data;
    struct node *child[3];
};
```

树的前序遍历算法



```
void DFS(struct node *root)
{
    int i;

    if (root!=0 )
    {
        cout<<root->data<<" ";

        for (i=0;i<3;i++)
            DFS(root->child[ i ]);  ////////////

    }

}
```


树的层次遍历算法

```
void BFS(struct node *root)
{
    int i;
    queue<struct node *> qu;
    struct node *temp;

    qu.push(root);
    while(!qu.empty())
    {
        temp=qu.front();
        qu.pop();
        cout<<temp->data<<" ";

        for (i=0;i<3;i++)
            if (temp->child[i]!=0)
                qu.push(temp->child[i]);
    }
}
```



存储结构→算法

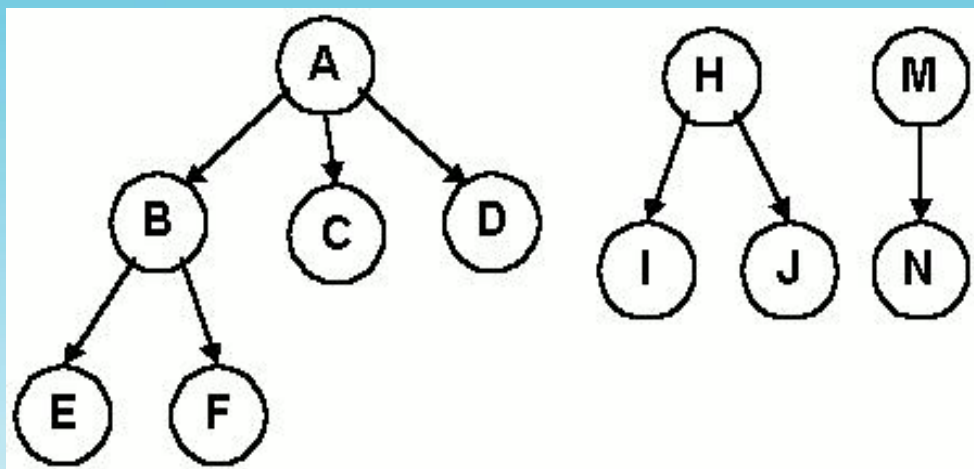


• 二、森林的遍历

1. 前序遍历

若森林非空，则

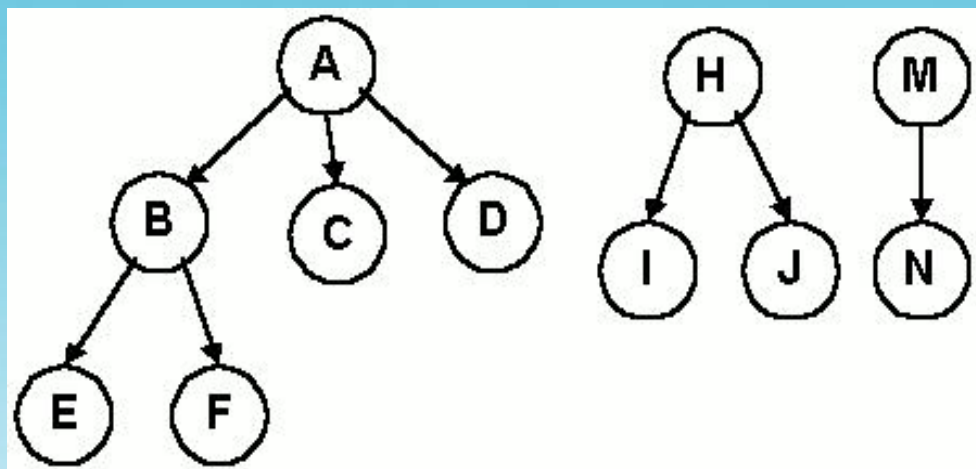
- (1) 访问森林中第一棵树的根结点；
- (2) 前序遍历森林中第一棵树的根结点的各子树所构成的森林；
- (3) 前序遍历除第一棵树外其它树所构成的森林。



2. 后序遍历

若森林非空，则

- (1) 后序遍历森林中第一棵树的根结点的各子树所构成的森林；
- (2) 访问第一棵树的根结点；
- (3) 后序遍历除第一棵树外其它树所构成的森林



结 论



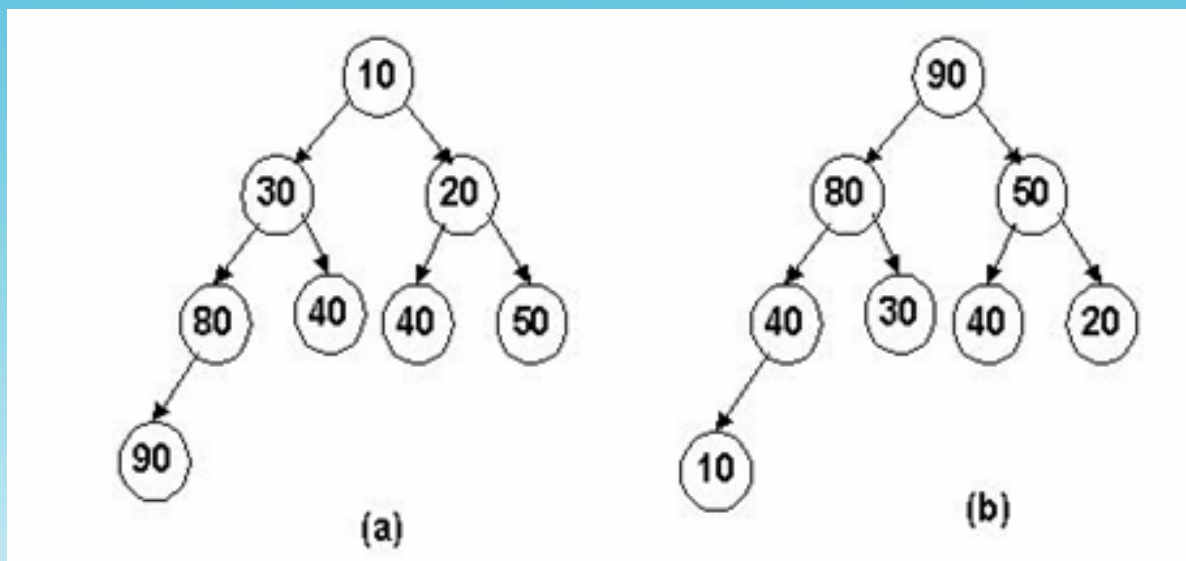
- 由于树、森林的存储结构相对较复杂，因此，其遍历操作对应算法的实现较为困难。
- 但事实上，根据树、森林和二叉树的转换关系以及它们的遍历定义可知：
 - (1) 树的前序遍历和后序遍历与相应二叉树的前序遍历和中序遍历的结果序列相同；
 - (2) 森林的前序遍历和中序遍历则与相应二叉树的前序遍历和中序遍历的结果序列相同。
- 因此，在实际应用中，树、森林的遍历算法经常借用二叉树的相应遍历算法来实现。

树形结构的应用



- 树形结构的特点：
 - 可以拥有多个直接后继
- (1) 分类
- (2) 贪心方法的改进

- **堆的定义**：设有n个数据元素组成的序列 $\{a_1, a_2, \dots, a_n\}$ ，若它满足下面的条件：
- (1) 这些数据元素是一棵**完全二叉树**中的结点，且 a_i ($i=1, 2, \dots, n$) 是该完全二叉树中编号为i的结点；
- (2) 若 $2i \leq n$ ，有 $a_{2i} \geq a_i$ ；
- (3) 若 $2i+1 \leq n$ ，有 $a_{2i+1} \geq a_i$ ；
- 则称该序列为一个堆。

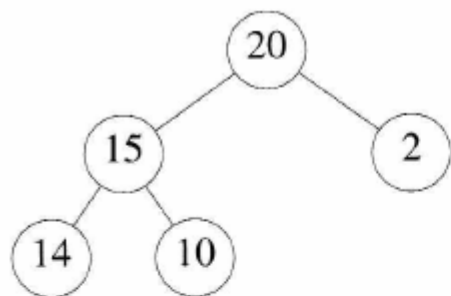


小根堆(最小堆)

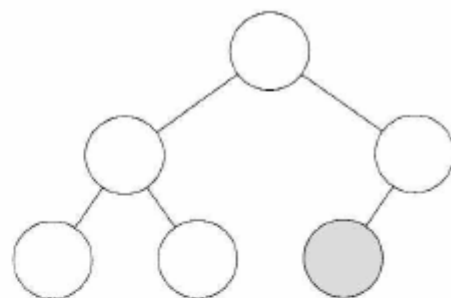
大根堆 (最大堆)

最大堆的插入

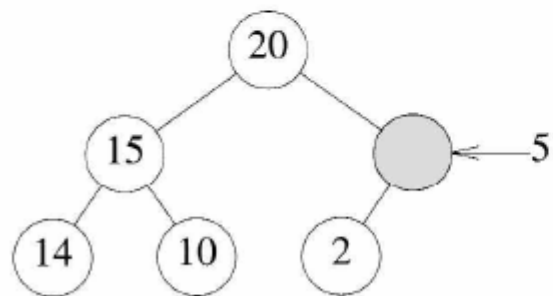
插入元素值为21



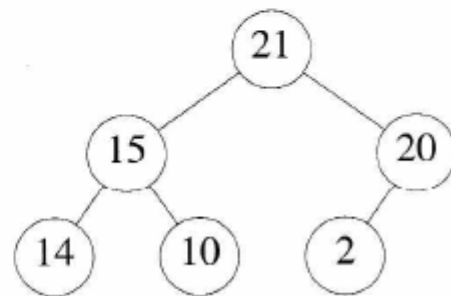
a)



b)

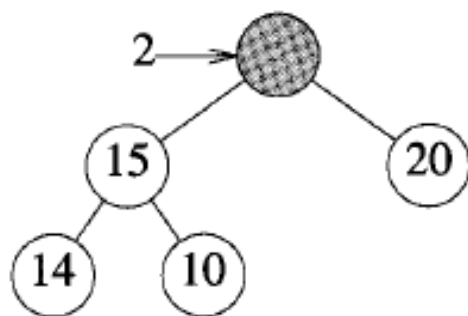
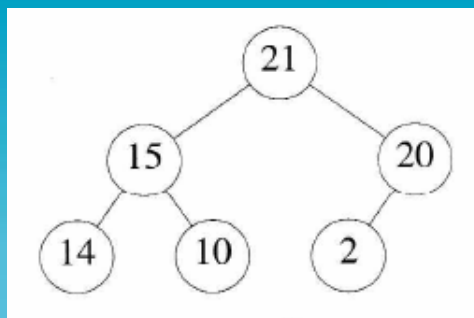


c)

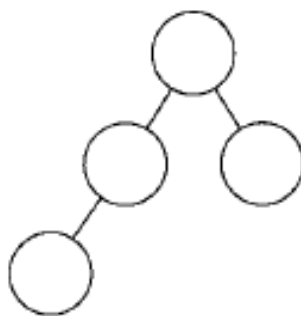


d)

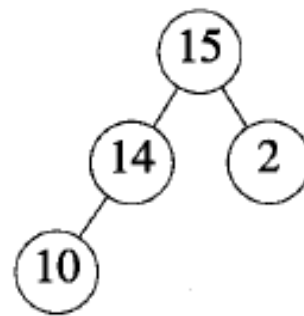
最大堆的删除



a)



b)



c)

操作的归纳



(1) 初始堆的建立

(2) 插入一个元素

(3) 删除一个元素

如何实现这些操作？

堆的存储结构



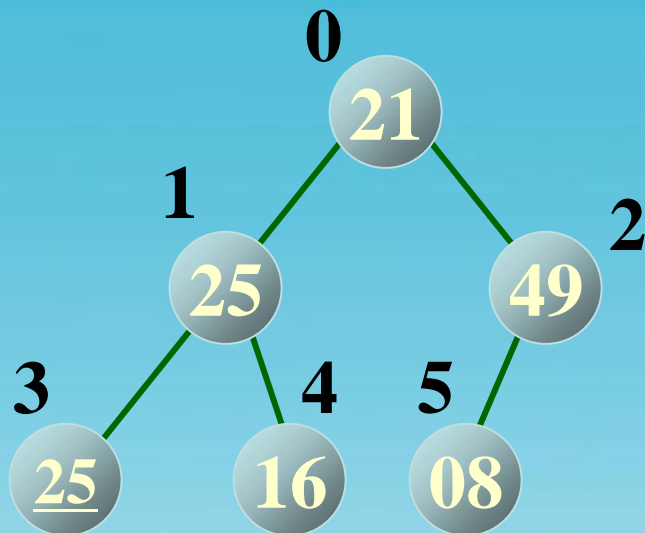
- 由于堆是完全二叉树，因此我们可以采用二叉树的顺序存储结构。
- 一维数组 \rightarrow 动态数组

数组存储时结点间的关系分析



- 如,输入 21,25,49,25,16,08

| | | | | | |
|----|----|----|-----------|----|----|
| 21 | 25 | 49 | <u>25</u> | 16 | 08 |
| 0 | 1 | 2 | 3 | 4 | 5 |



- n 个元素则存放单元为 $0 \sim n-1$.

最小堆的类定义

```
#define DefaultSize = 10
template <class T>
class MinHeap
{
public:
    MinHeap ( int sz=DefaultSize );
    MinHeap ( T arr[ ], int n );
    ~MinHeap ( ) { delete [ ] heap; }
    const MinHeap<T> & operator =
        ( const MinHeap &R );
    bool Insert ( const T &x );
    bool RemoveMin ( T &x );
```





```
bool IsEmpty ( ) const
    { return CurrentSize == 0; }
bool IsFull ( ) const
    { return CurrentSize == MaxHeapSize; }
void MakeEmpty ( ) { CurrentSize = 0; }
private:
    T *heap;
    int CurrentSize;
    int MaxHeapSize;
    void siftDown ( int start, int m );
    void siftUp ( int start );
}
```

初始化操作 → 生成空堆



```
template <class T>
```

```
MinHeap <T> ::MinHeap ( int sz )
```

```
{
```

```
    //根据给定大小maxSize,建立堆对象
```

```
    MaxHeapSize=DefaultSize<sz ?maxSize :DefaultSize ;//确定堆大小
```

```
    heap = new T[MaxHeapSize]; //创建堆空间
```

```
    CurrentSize = 0;                //初始化
```

```
}
```

初始化操作——建立一个堆

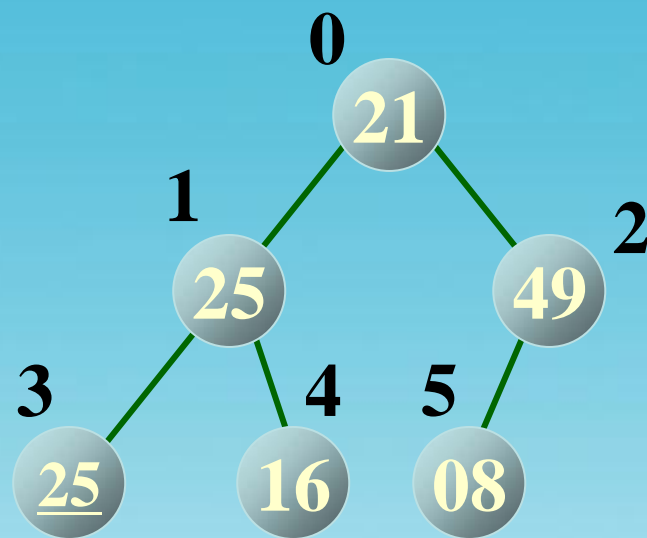


- 初始堆

条件?

- 如,输入 21,25,49,25,16,08

| | | | | | |
|----|----|----|-----------|----|----|
| 21 | 25 | 49 | <u>25</u> | 16 | 08 |
| 0 | 1 | 2 | 3 | 4 | 5 |



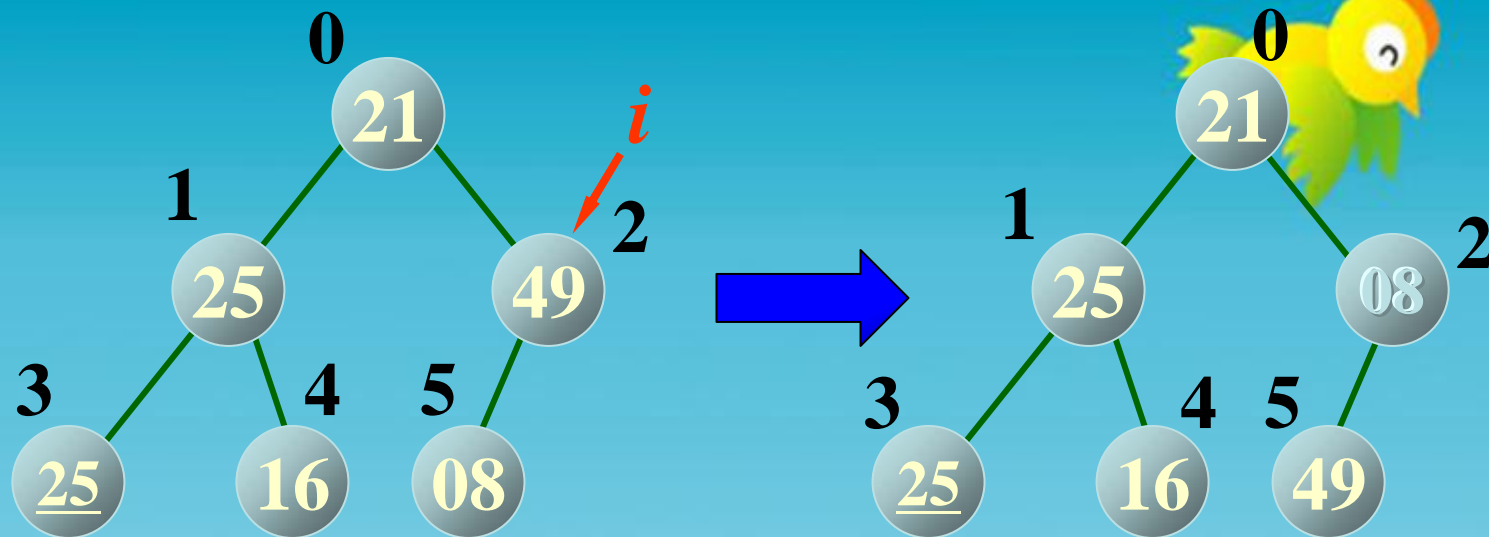
初始堆的建立



- 1964年Floyd给出了一种称作**筛选法**的算法。
- **具体做法**：（根据小根堆的概念）
 - (1) 让 n 个结点成为完全二叉树
 - (2) 让其成为小根堆

从最后一个拥有孩子的结点开始调整(即 $\text{heap}[(n-2)/2]$, $\text{heap}[(n-2)/2-1]$,, $\text{heap}[0]$ 为根的树调整为小根堆)

建立初始的小根堆



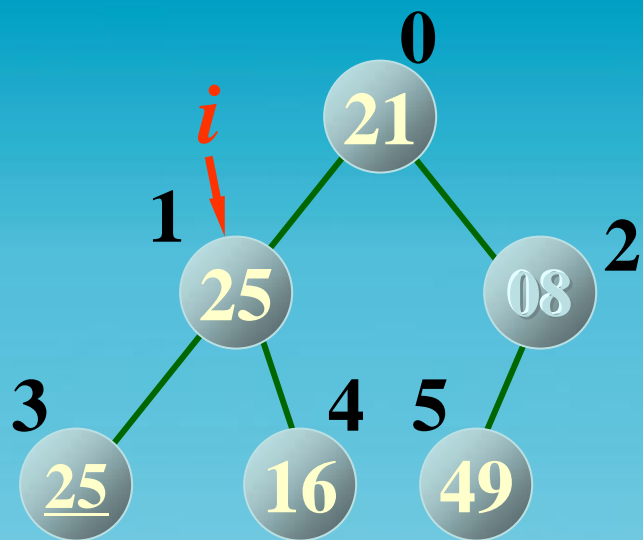
| | | | | | |
|----|----|----|-----------|----|----|
| 21 | 25 | 49 | <u>25</u> | 16 | 08 |
|----|----|----|-----------|----|----|

初始状态

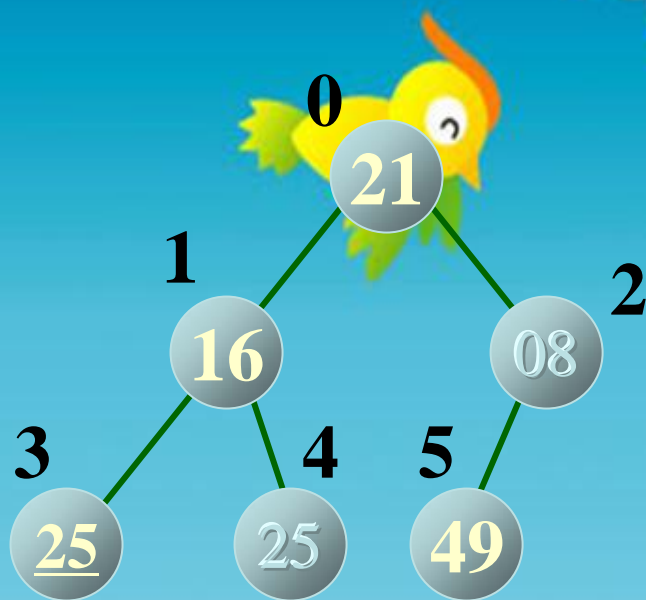
| | | | | | |
|----|----|----|-----------|----|----|
| 21 | 25 | 08 | <u>25</u> | 16 | 49 |
|----|----|----|-----------|----|----|

$i=2$ 的调整结果

建立初始的小根堆

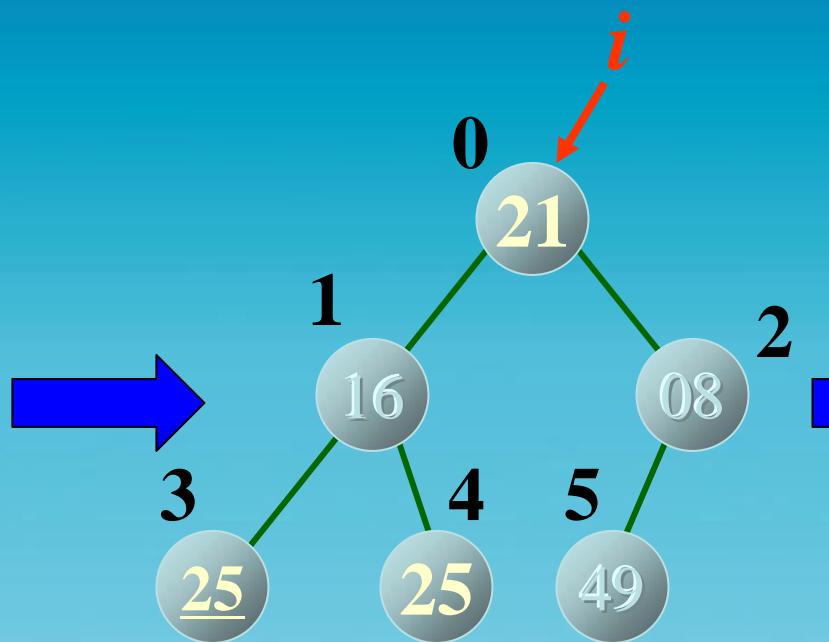


| | | | | | |
|----|----|----|-----------|----|----|
| 21 | 25 | 08 | <u>25</u> | 16 | 49 |
|----|----|----|-----------|----|----|

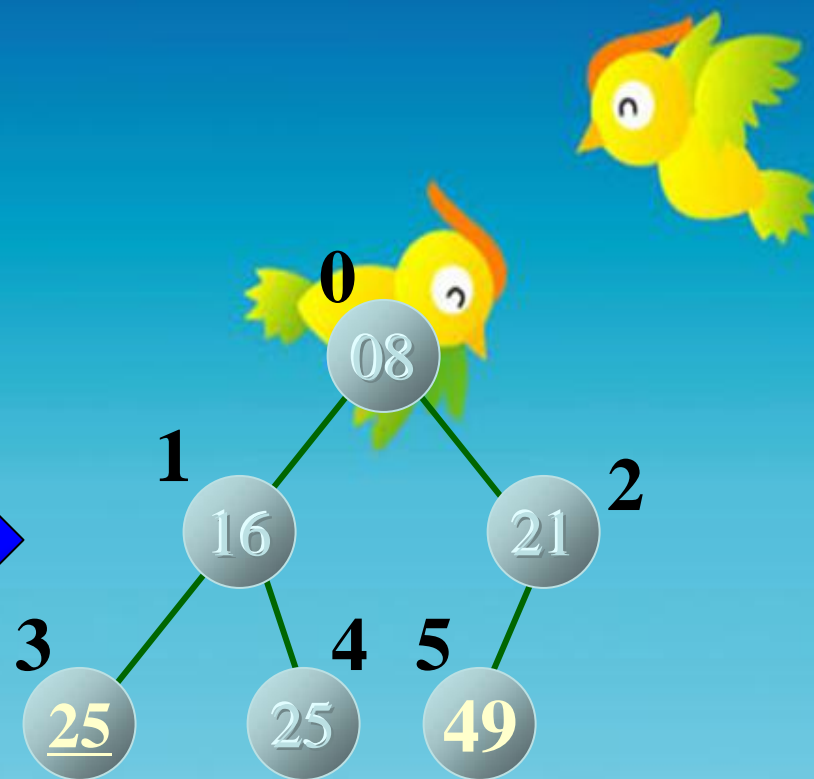


| | | | | | |
|----|----|----|-----------|----|----|
| 21 | 16 | 08 | <u>25</u> | 25 | 49 |
|----|----|----|-----------|----|----|

$i = 1$ 时的调整结果



| | | | | | |
|----|----|----|-----------|----|----|
| 21 | 16 | 08 | <u>25</u> | 25 | 49 |
|----|----|----|-----------|----|----|

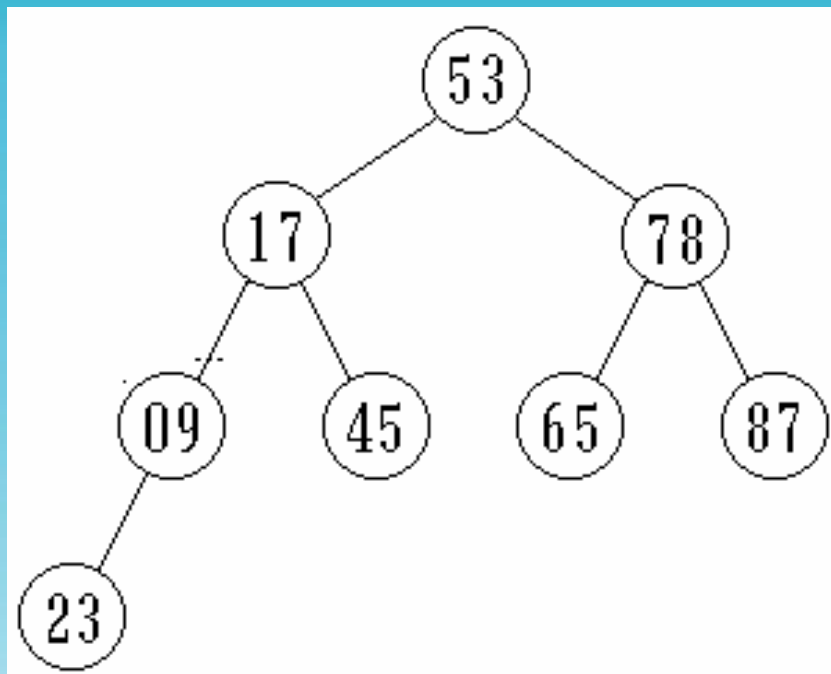


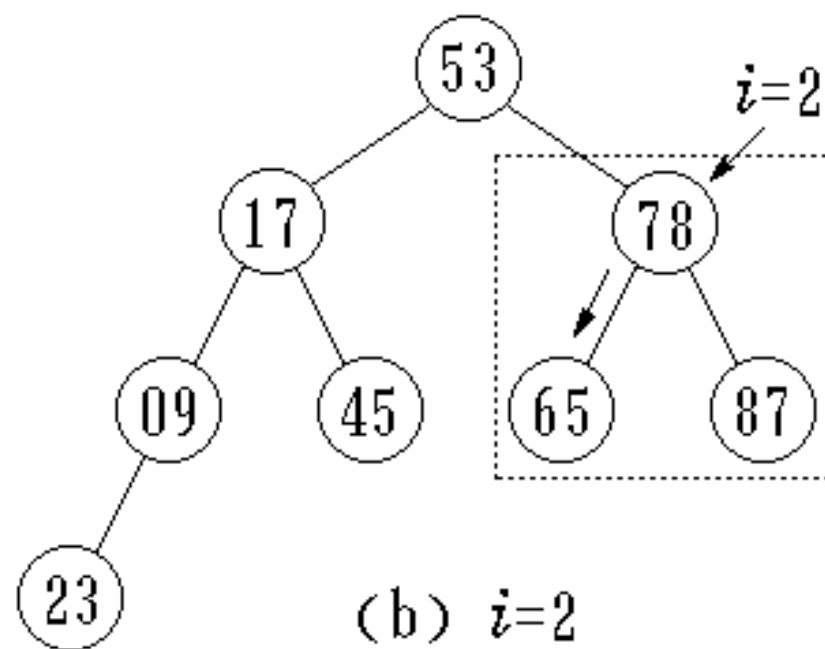
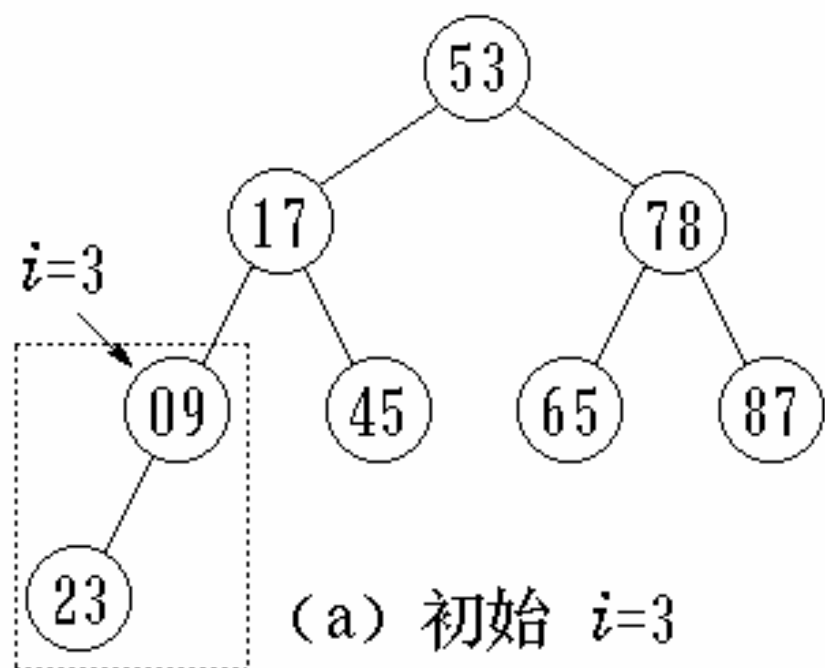
| | | | | | |
|----|----|----|-----------|----|----|
| 08 | 16 | 21 | <u>25</u> | 25 | 49 |
|----|----|----|-----------|----|----|

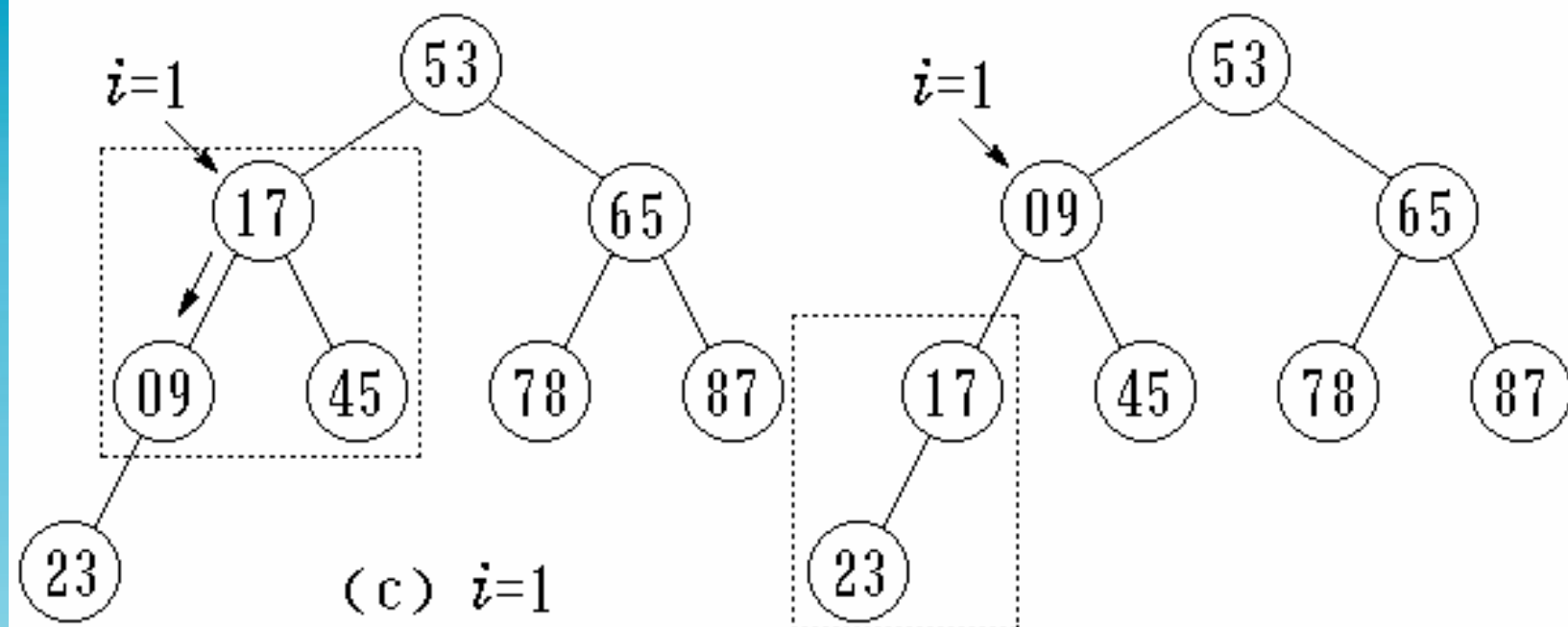
初始堆（小根堆）

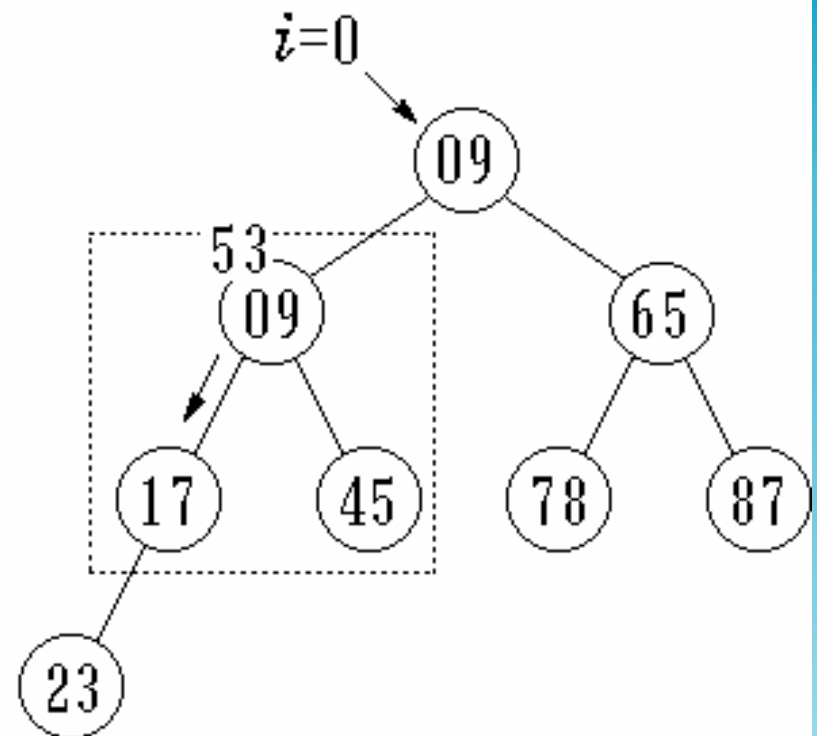
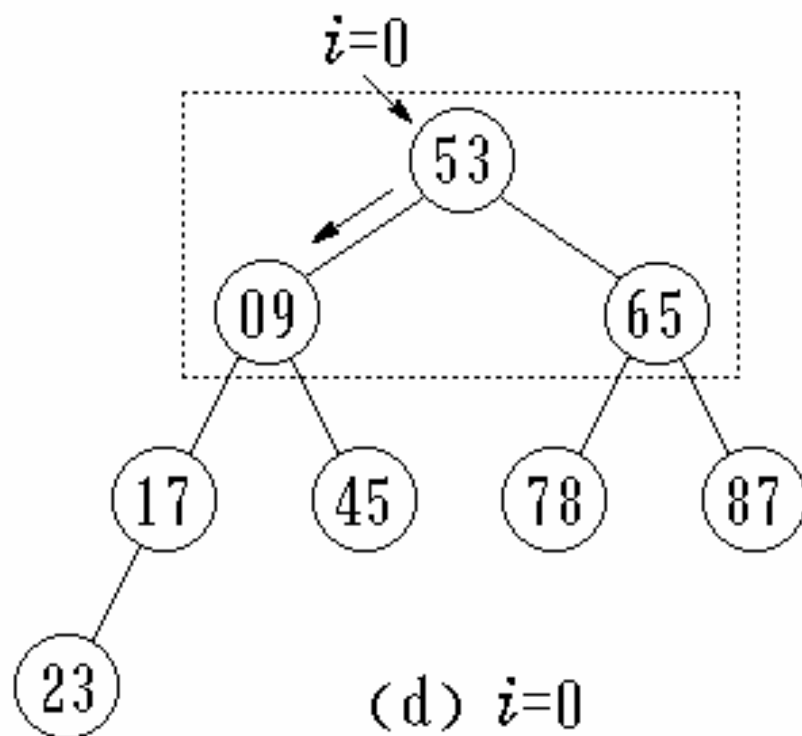
例子2

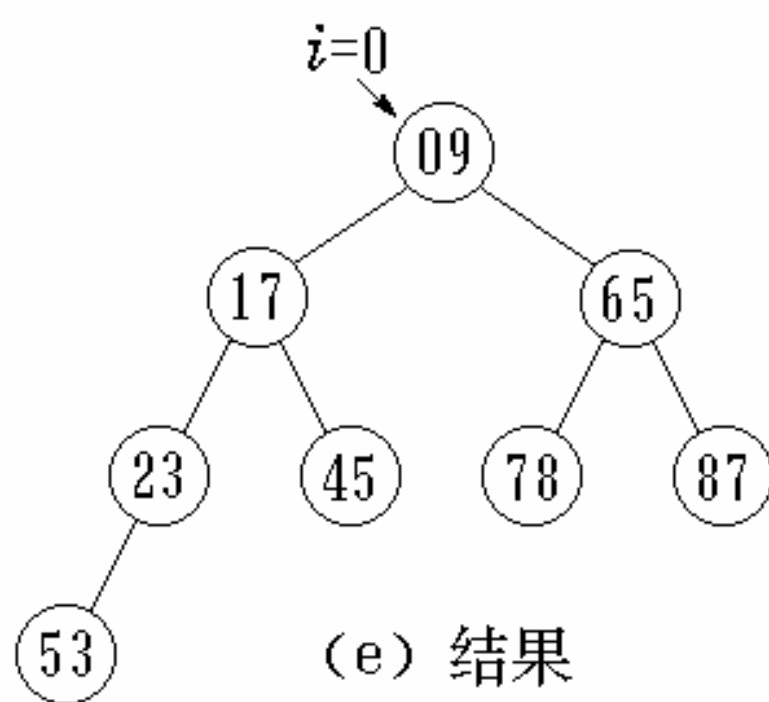
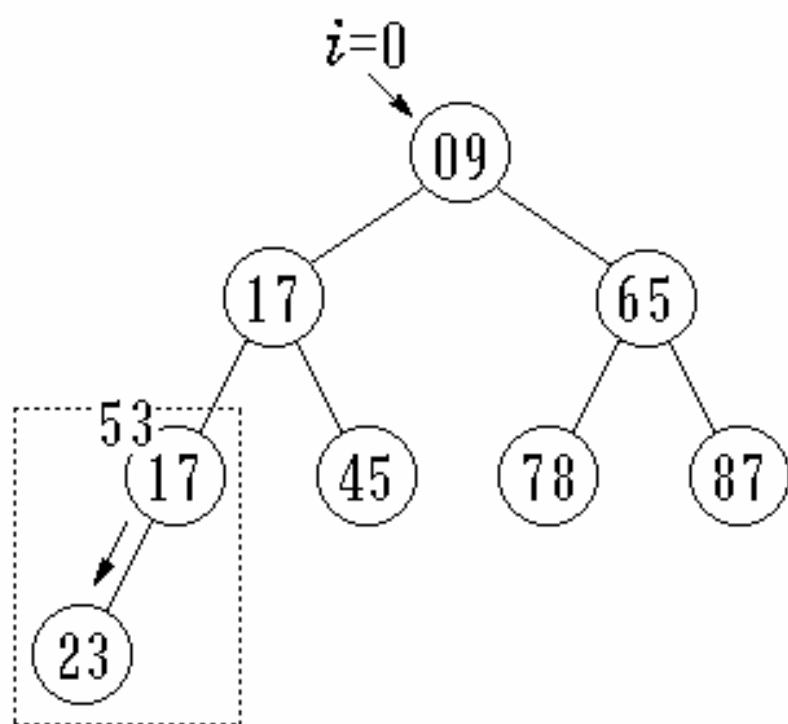
- 例2，输入 53,17,78,09,45,65,87,23











(e) 结果

操作的归纳

- 向下调整的操作



结点向下调整的代码



```
template <class T>
void MinHeap<T> :: siftDown ( int start, int m )
{
    int i = start, j = 2*i+1;    // j 是 i 的左子女
    T temp = heap[i];
    while ( j <= m )
    {
        if ( j < m && heap[j] > heap[j+1] )
            j++;    // 两子女中选小者
        if ( temp <= heap[j] ) break;
        else { heap[i] = heap[j]; i = j; j = 2*j+1; }
    }
    heap[i] = temp;
}
```

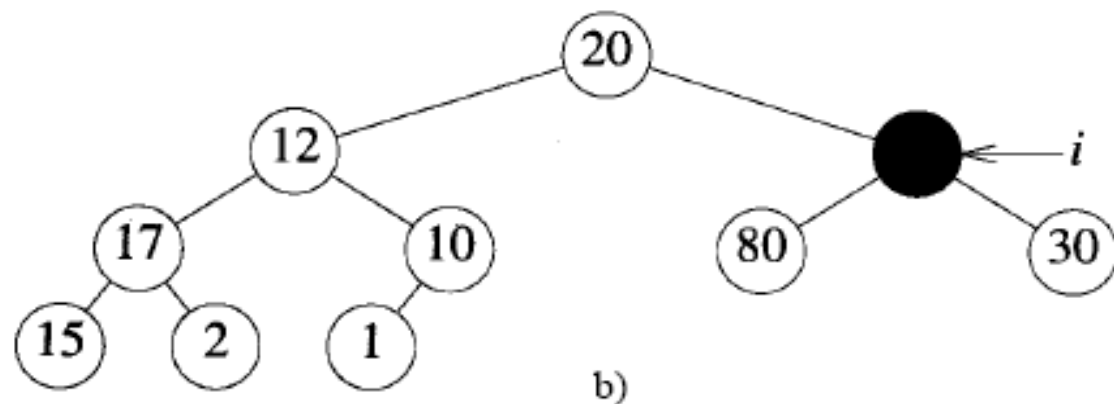
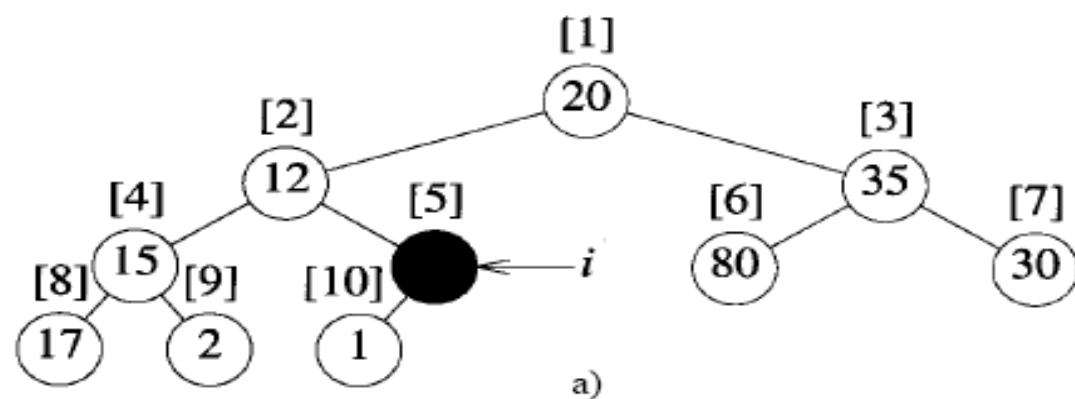


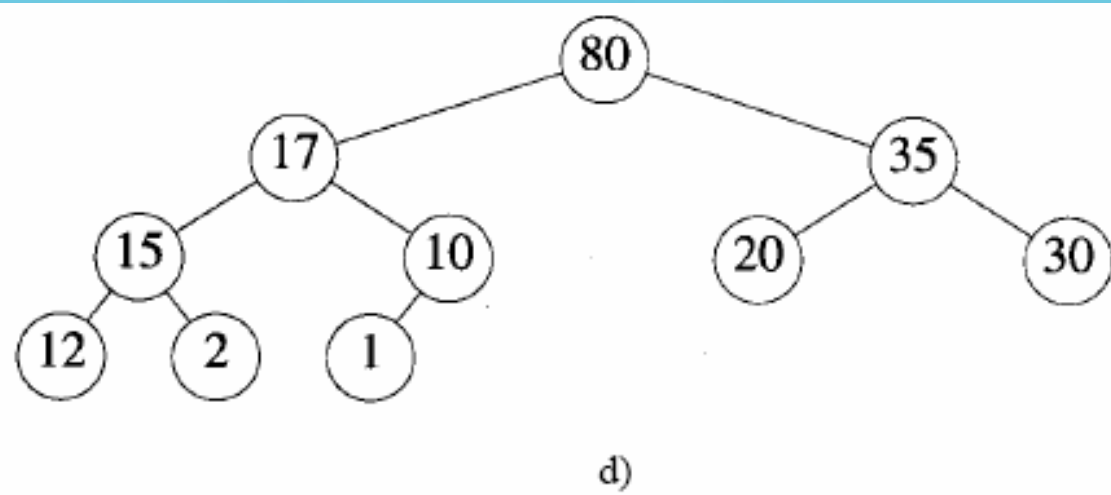
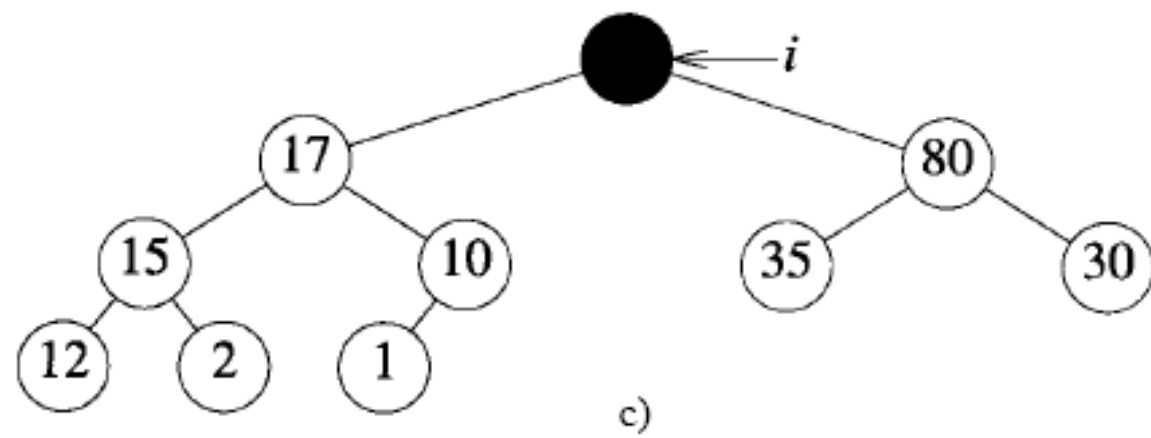
初始化一个非空最小堆

```
template<class T>
void MinHeap<T>:: MinHeap(T arr[], int n)
{ // 把最小堆初始化为数组arr .
    delete [ ] heap;
    MaxHeapSize = DefaultSize < n ? n : DefaultSize;
    heap = new Type [MaxHeapSize];
    for ( int i=0;i<n;i++) heap[i] = arr[i];    //数组传送
    CurrentSize = n;    //当前堆大小
    int currentPos = (CurrentSize-2)/2; //最后非叶
    while ( currentPos >= 0 )
    { //从下到上逐步扩大,形成堆
        siftDown ( currentPos, CurrentSize-1 );
        //从currentPos开始,到CurrentSize为止,调整
        currentPos--;
    }
}
```



最大堆的初始化——建立最大堆

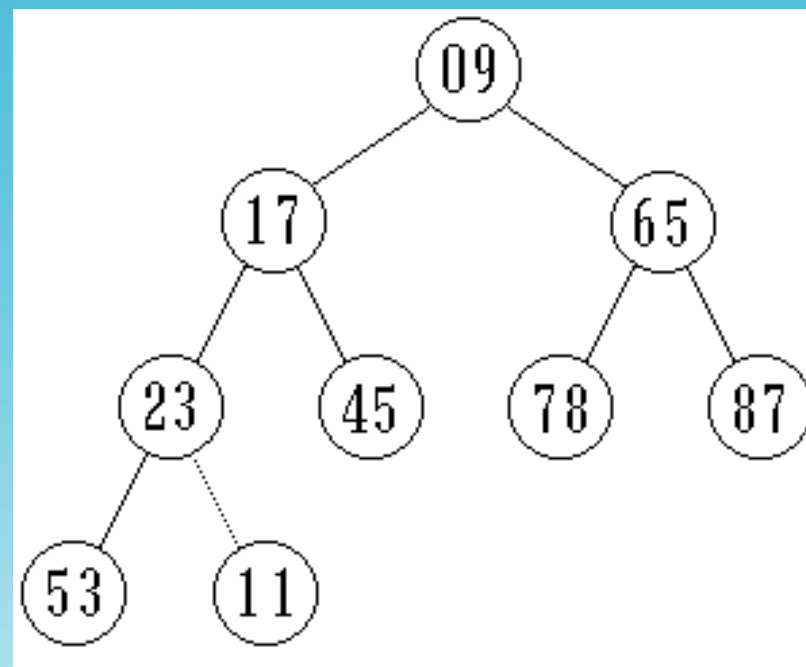
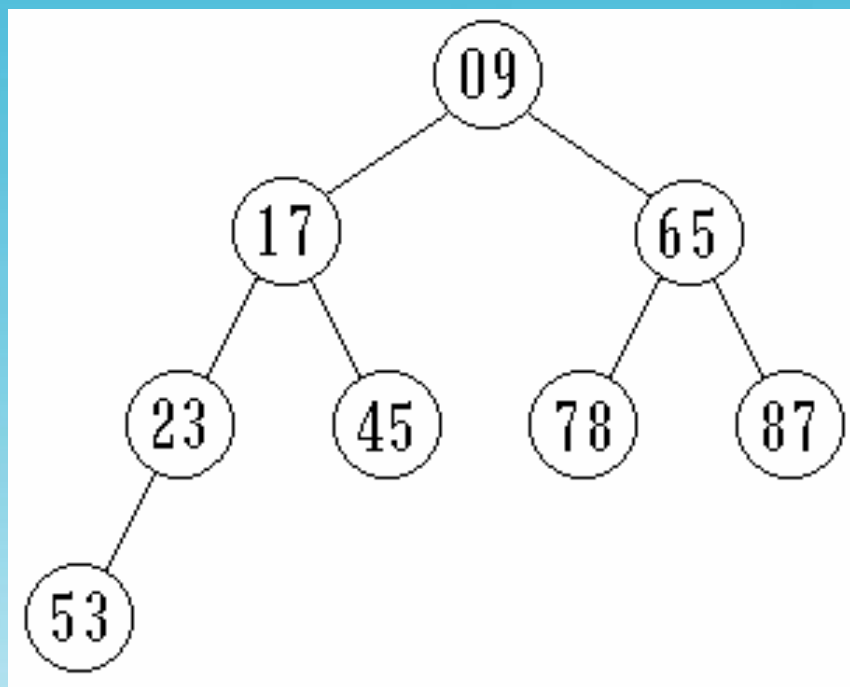


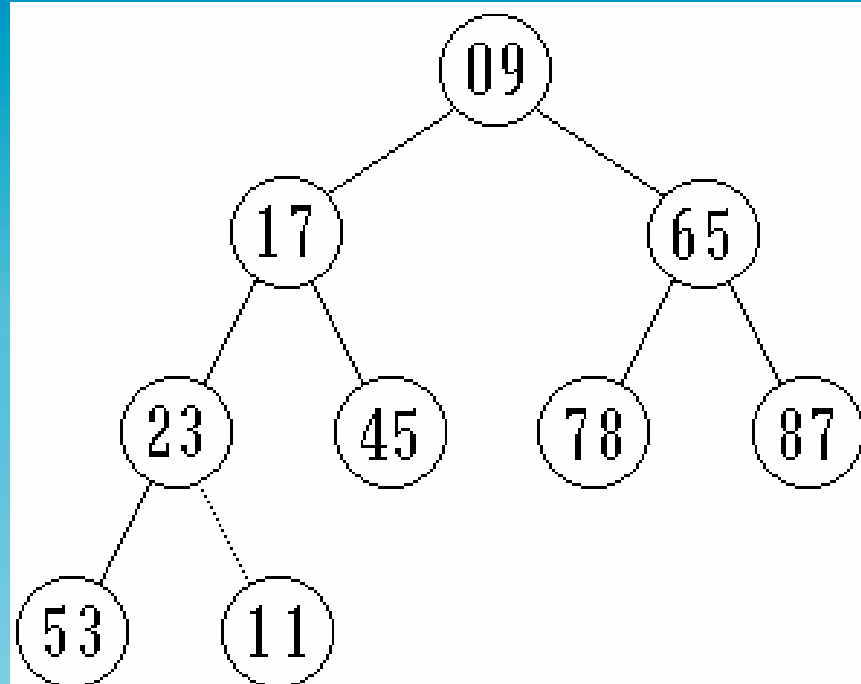


小根堆的插入操作

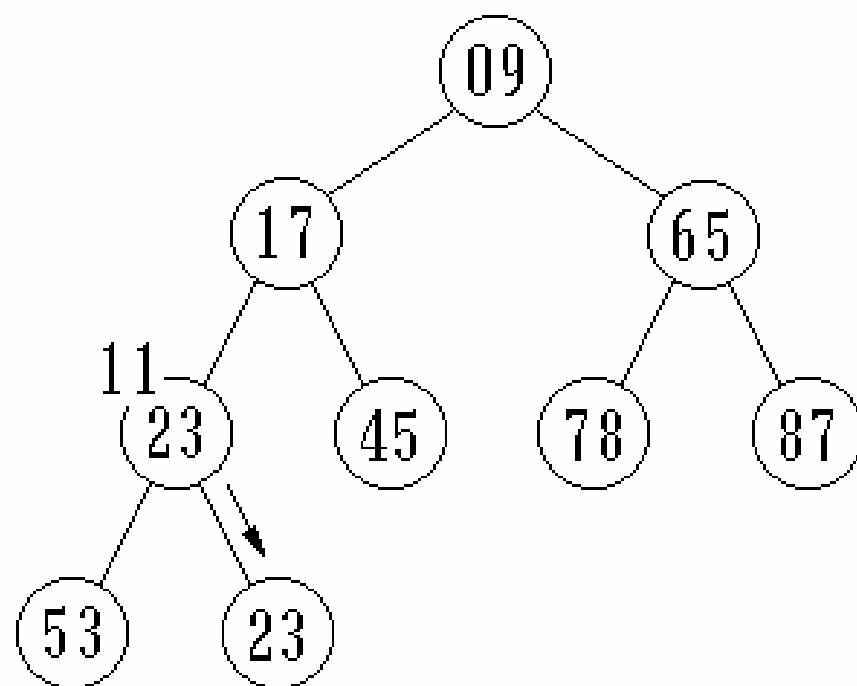


- 插入11

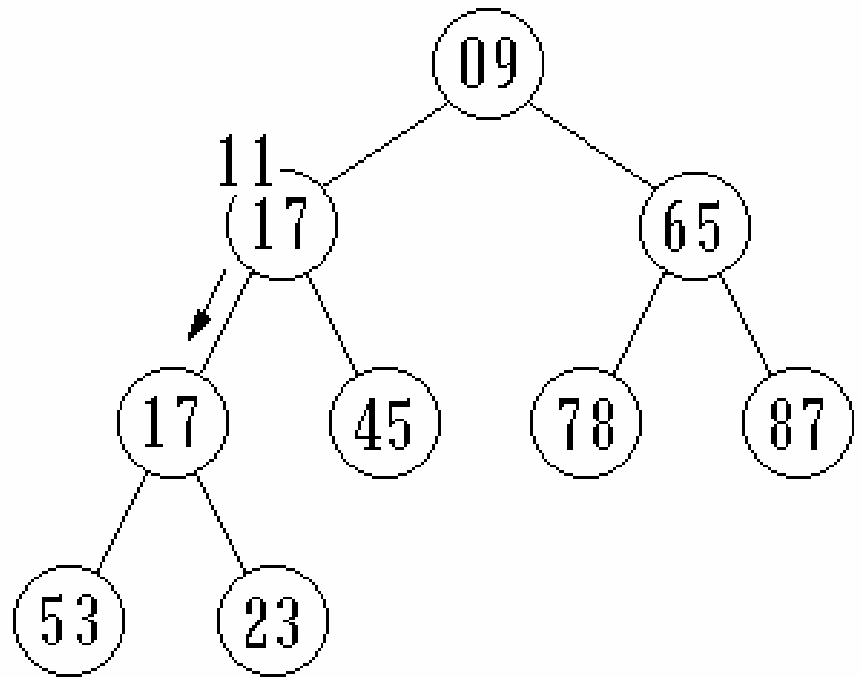
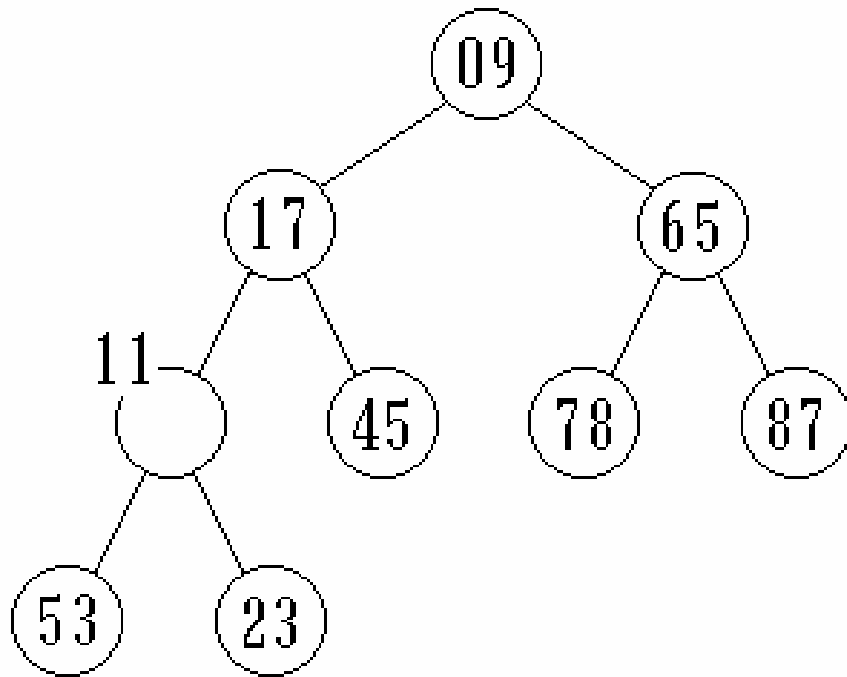




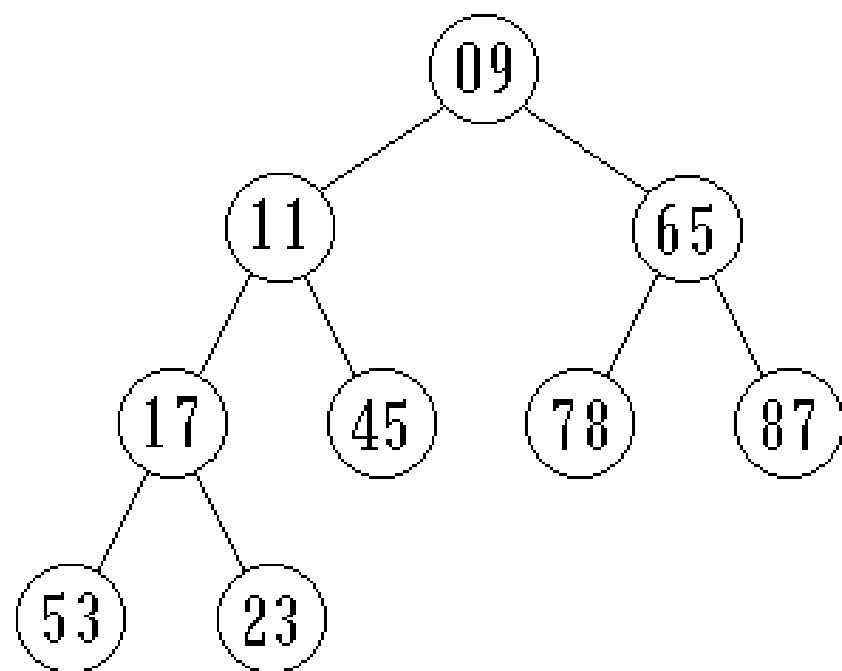
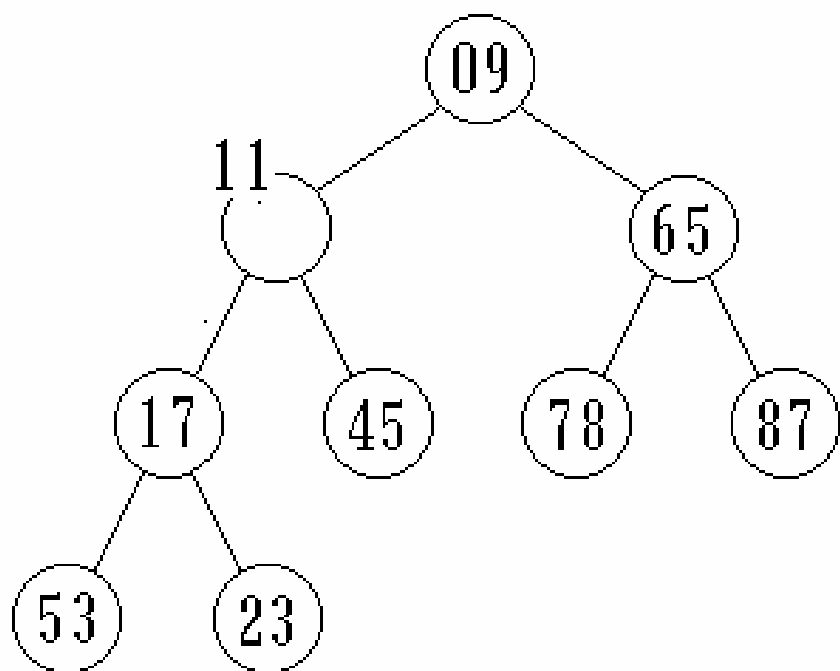
(a) 初始 尾部加11



(b) 双亲关键码23下降



(c) 双亲关键码17下降



(d) 11 回填 调整完成


操作的归纳

- 向上调整的操作



最小堆的向上调整算法

```
template <class T>
void MinHeap<T> ::siftUp ( int start )
{    //从 start 开始,向上直到0,调整堆
    int j = start, i = (j-1)/2;  // i 是 j 的双亲
    T temp = heap[j];
    while ( j > 0 )
    {
        if ( heap[i] <= temp ) break;
        else { heap[j] = heap[i]; j = i; i = (i -1)/2; }
    }
    heap[j] = temp;
}
```

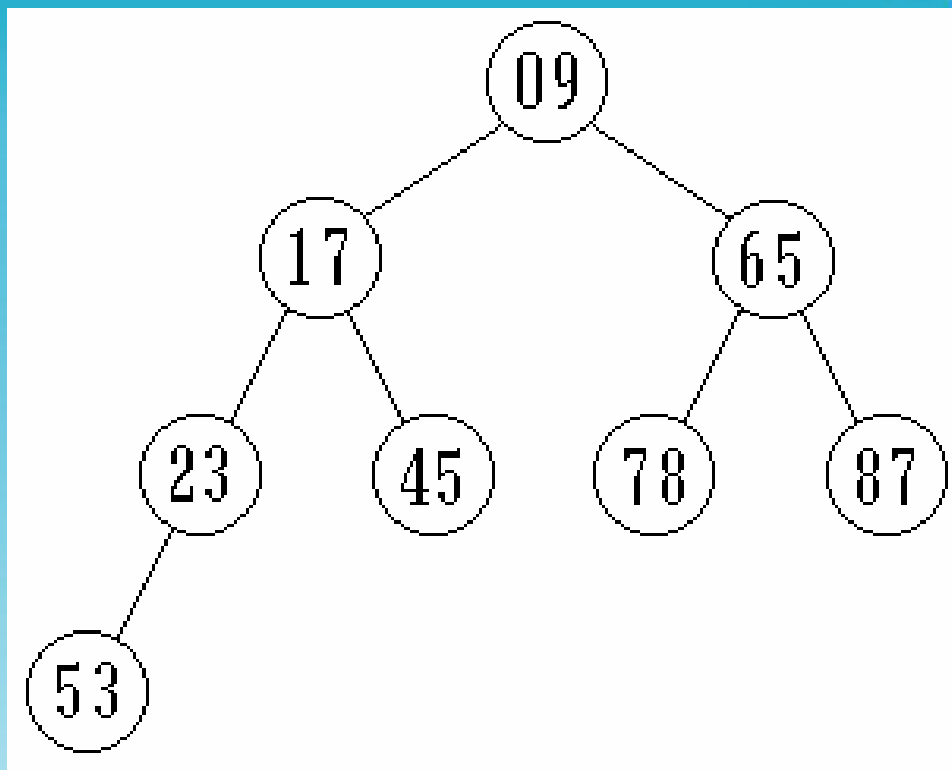


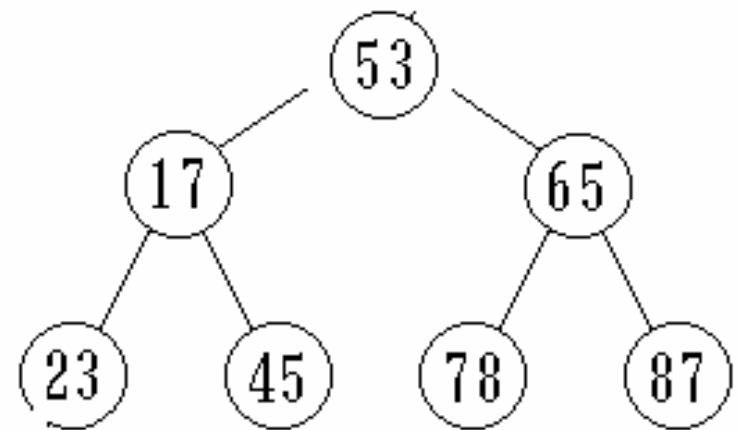
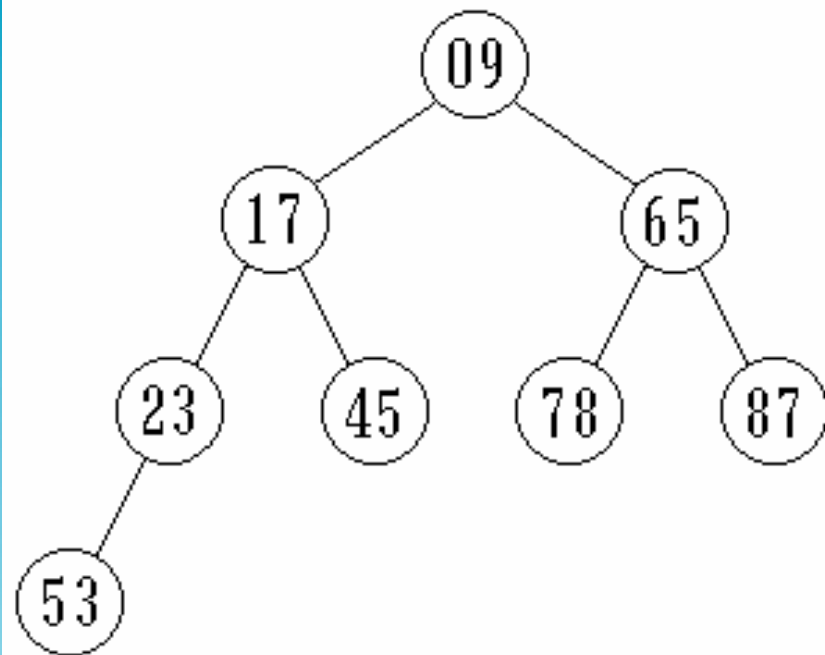
堆插入算法



```
template <class T>
bool MinHeap<T> ::Insert ( const T &x )
{ //在堆中插入新元素 x
    if ( CurrentSize == MaxHeapSize ) //堆满
    { cout << "堆已满" << endl; return 0; }
    heap[CurrentSize] = x;           //插在表尾
    siftUp (CurrentSize);           //向上调整为堆
    CurrentSize++;                  //堆元素增一
    return 1;
}
```

小根堆的删除操作





操作的归纳

- 向下调整的操作



最小堆的删除算法

```
template <class T>
int MinHeap <T> ::RemoveMin ( T &x )
{
    if ( !CurrentSize )
    {   cout << " 堆已空 " << endl; return 0; }
    x = heap[0];           //最小元素出队列
    heap[0] = heap[CurrentSize-1];
    CurrentSize--;        //用最小元素填补
    siftDown ( 0, CurrentSize-1 );
    //从0号位置开始自顶向下调整为堆
    return 1;
}
```



树的应用



- 2. 哈夫曼树
- 问题的提出：如何将一篇文章的全部内容以最短的时间从一端发送到另一端。
- (1) 计算机是如何传送信息的。
- (2) 如何才能提高传送效率？
- 解决方法：
- (1) 硬件设备的升级；
- (2) 信息重新编码。

- 涉及到的几个概念:



- (1) 结点的带权路径长度
- (2) 树的带权路径长度
- $$WPL = w_1 * l_1 + w_2 * l_2 + \dots w_i * l_i + \dots + w_n * l_n$$
- (3) 哈夫曼树或最优树。

- 如果给出一组权值，如何才能构造出一棵哈夫曼树呢？



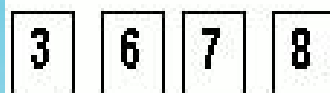
- 哈夫曼 (Huffman) 在1952年提出的一种算法解决了这个问题 (贪心法)。
- 其基本思路是：

哈夫曼树建立例子

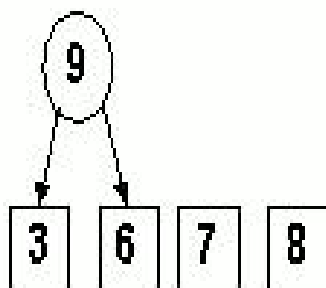
例1, 4,2,6,8,3,2,1



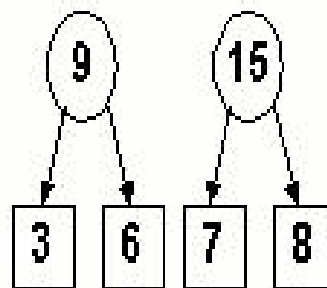
- 例2, $\{3, 7, 8, 6\}$



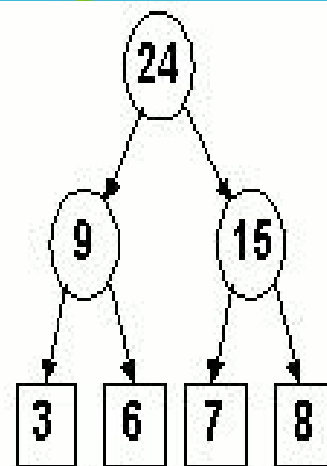
(a)



(b)



(c)



(d)

问题发现



- (1) 如何找出最小值和次小值
- (2) 如何实现合并工作

- 如何在计算机上哈夫曼树的构造方法？
- (1) 存储结构——如何存储哈夫曼树



每个结点的存储结构应如下：

| | | | |
|--------|--------|--------|--------|
| lchild | parent | weight | rchild |
|--------|--------|--------|--------|

其中，lchild、rchild分别用来存放该结点的左、右孩子在顺序存储结构中的单元编号，如果该结点为叶子结点，则 lchild=-1, rchild=-1。

于是可以给出哈夫曼树类的描述如下：

```
struct HuffmanNode {  
    int  weight; //存放结点的权值，假设只考虑处理权值为整数的情况  
    int  parent ; // -1 表示为根结点，否则表示为非根结点  
    int  lchild, rchild; // 分别存放该结点的左、右孩子的所在单元的编号  
};
```


图 6-30 所示是对于给定的权值 { 3, 7, 8, 6 }, 利用上述算法构造哈夫曼树的过程。

| 单元编号 | | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|-----------|--------|----|----|----|----|----|----|----|
| 初始 状态 | lchild | -1 | -1 | -1 | -1 | | | |
| | parent | -1 | -1 | -1 | -1 | | | |
| | weight | 3 | 7 | 8 | 6 | | | |
| | rchild | -1 | -1 | -1 | -1 | | | |
| 第一次 合并 | lchild | -1 | -1 | -1 | -1 | 0 | | |
| | parent | 4 | -1 | -1 | 4 | -1 | | |
| | weight | 3 | 7 | 8 | 6 | 9 | | |
| | rchild | -1 | -1 | -1 | -1 | 3 | | |
| 第二次 合并 | lchild | -1 | -1 | -1 | -1 | 0 | 1 | |
| | parent | 4 | 5 | 5 | 4 | -1 | -1 | |
| | weight | 3 | 7 | 8 | 6 | 9 | 15 | |
| | rchild | -1 | -1 | -1 | -1 | 3 | 2 | |
| 第三次 合并 | lchild | -1 | -1 | -1 | -1 | 0 | 1 | 4 |
| | parent | 4 | 5 | 5 | 4 | 6 | 6 | -1 |
| | weight | 3 | 7 | 8 | 6 | 9 | 15 | 24 |
| | rchild | -1 | -1 | -1 | -1 | 3 | 2 | 5 |

说明: 带阴影部分表示下次选出来的权值最小和次小的结点。

- (2) 实现算法

- 算法核心：贪心法

- 缺陷：

- 改进方案：堆



新实现方案——用堆



(1) 存储结构

三叉链式结构

(2) 操作实现

- 1) 用堆来找出最小值和次小值
- 2) 如何实现合并工作

哈夫曼树结点的类定义

```
#define DefaultSize 20
struct HuffmanNode
{
    float data;
    HuffmanNode *leftChild, *rightChild;
    HuffmanNode *parent;

    ....

};
```



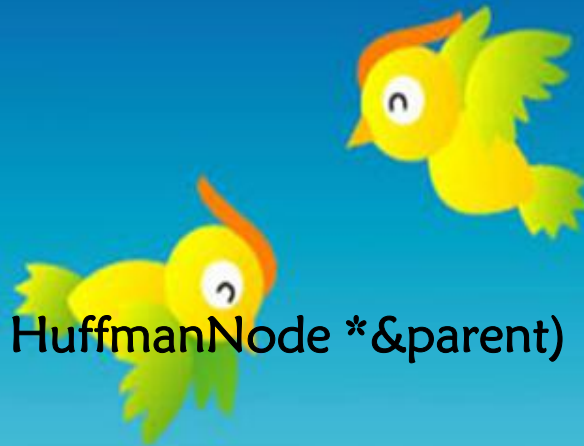
哈夫曼树的类定义

```
class HuffmanTree
{
    HuffmanNode *root;
    void mergeTree(HuffmanNode &ht1, HuffmanNode&ht2, HuffmanNode *&parent);
    // 合并操作

public:
    HuffmanTree(float w[ ], int n ); // 建立哈夫曼树
    ~HuffmanTree( );
};
```



合并操作的实现



```
void mergeTree(HuffmanNode &ht1, HuffmanNode&ht2, HuffmanNode *&parent)
{
    parent=new HuffmanNode;
    parent->leftchild=&ht1;
    parent->rightchild=&ht2;
    parent->data = ht1.root->data+ht2.root->data;
    ht1->root->parent= ht2->root->parent=parent;
}
```

建立霍夫曼树的算法



```
HuffmanTree::HuffmanTree (float w[ ], int n)
```

```
{ minheap hp;
```



```
    HuffmanNode *parent , first, second, work;
```

```
    for ( int i = 0; i < n; i++ )
```

```
    {
```

```
        work.data=w[i];
```

```
        work.leftChild =work.rightChild = work.parent=NULL;
```

```
        hp.Insert(work);
```

```
    }    //传送初始权值
```



```
for ( int i = 0; i < n-1; i++ )  
{
```

```
    //建立霍夫曼树的过程，做n-1趟
```

```
    hp.removeMin ( first );    //选根权值最小的树
```

```
    hp.removeMin ( second ); //选根权值次小的树
```

```
    mergeTree( first, second, parent ); // 合并
```

```
    hp.Insert ( *parent );      //重新插入到小根堆中
```

```
}
```

```
root=parent; //最后的结点为根结点
```

```
}
```


- 将一篇文章的全部内容以最短的时间从一端发送到另一端的解决方法：

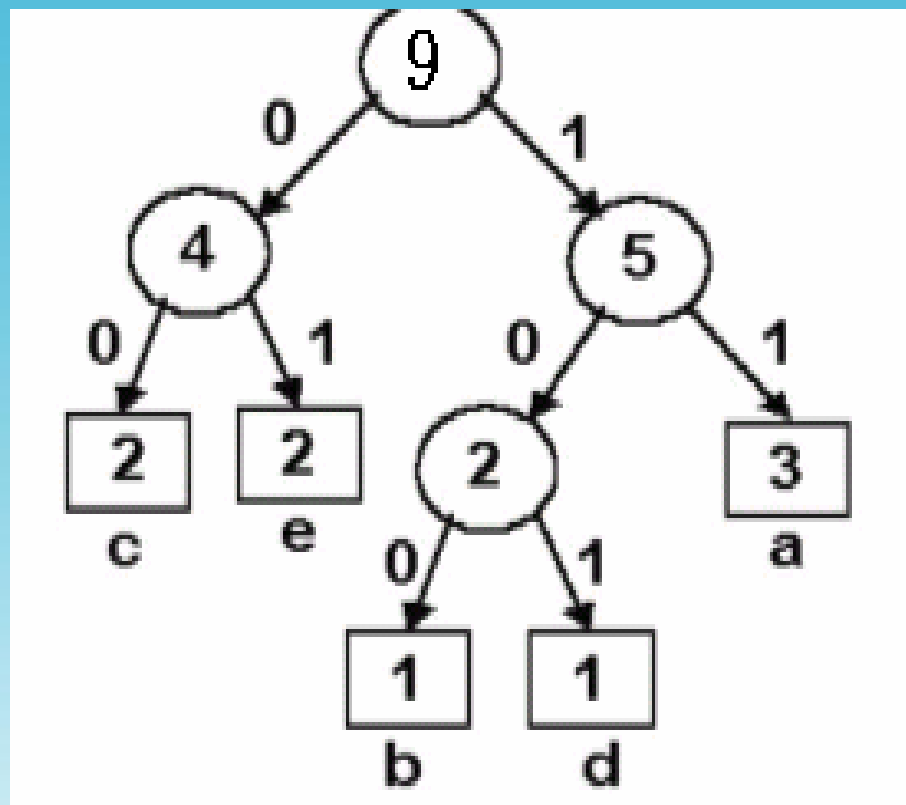


- (1) 以文字出现频率——→权值——→叶子
——→哈夫曼树；
- (2) 为每个文字（叶子）进行编码；
- (3) 发送；
- (4) 解码；

- 例如，文字信息“acbcdeaea”，

- {a,b,c,d,e}的出现频率分别为3，1，2，1，2，

- 构造哈夫曼树



编码

各字母的二进制编码为：

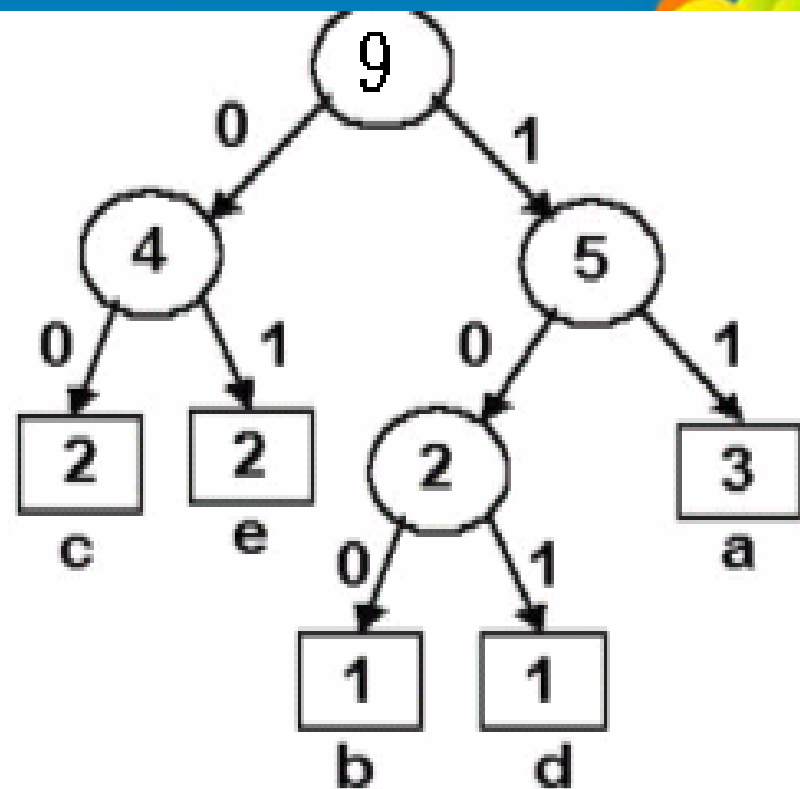
字母a: 11

字母b: 100

字母c: 00

字母d: 101

字母e: 01

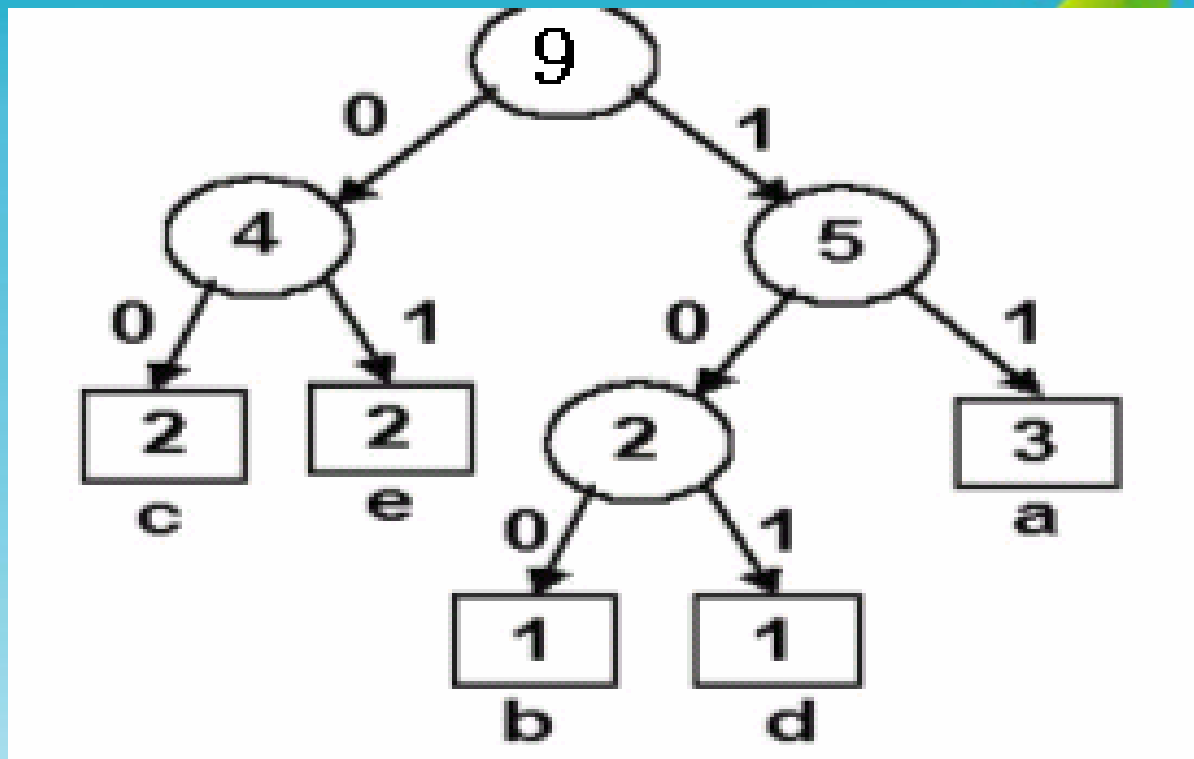


文字信息“acbcdeaea”，其编码为：

11001000010101110111

解码

- 接收的二进制串为：11001000010101110111



- 二叉树的遍历

- 注意事项:

- 为了确保接收方和发送方所用的哈夫曼树是一样的，因此双方所采用的哈夫曼树构造算法是一样的。



• 习题



- 30. 现有以下按前序和中序遍历二叉树的结果，问这样能否唯一地确定这棵二叉树的形状？为什么？

前序序列：ABCDEFGHI 中序序列：BCAEDGHFI

- 37. 一个深度为 h 的满 k 叉树有如下性质：第 h 层上的结点都是叶子结点，其余各层上每个结点都有 k 棵非空子树。如果按层次顺序从1开始对全部结点编号，问：
 - (1) 各层的结点数目是多少？
 - (2) 编号为 n 的结点的父结点（若存在）的编号是多少？
 - (3) 编号为 n 的结点的第 i 个儿子结点（若存在）的编号是多少？
 - (4) 编号为 n 的结点有右兄弟的条件是什么？其右兄弟的编号是多少？