

Chapter 4

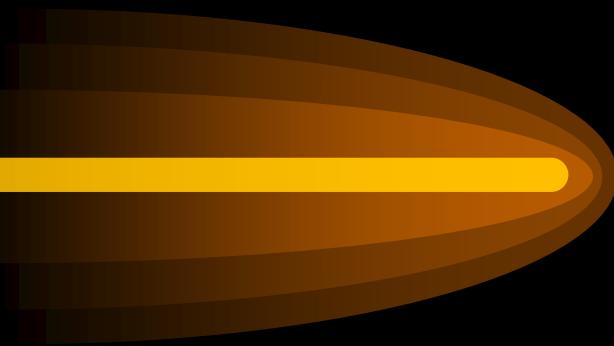
Tree

Two kinds of data structure

A decorative graphic consisting of three overlapping curved bands. The top band is dark red, the middle is orange, and the bottom is yellow. They curve from the left side of the slide towards the right, ending in a point.

- Linear: list, stack, queue, string
- Non-linear: tree, graph

4.1 Tree



1. Definition:

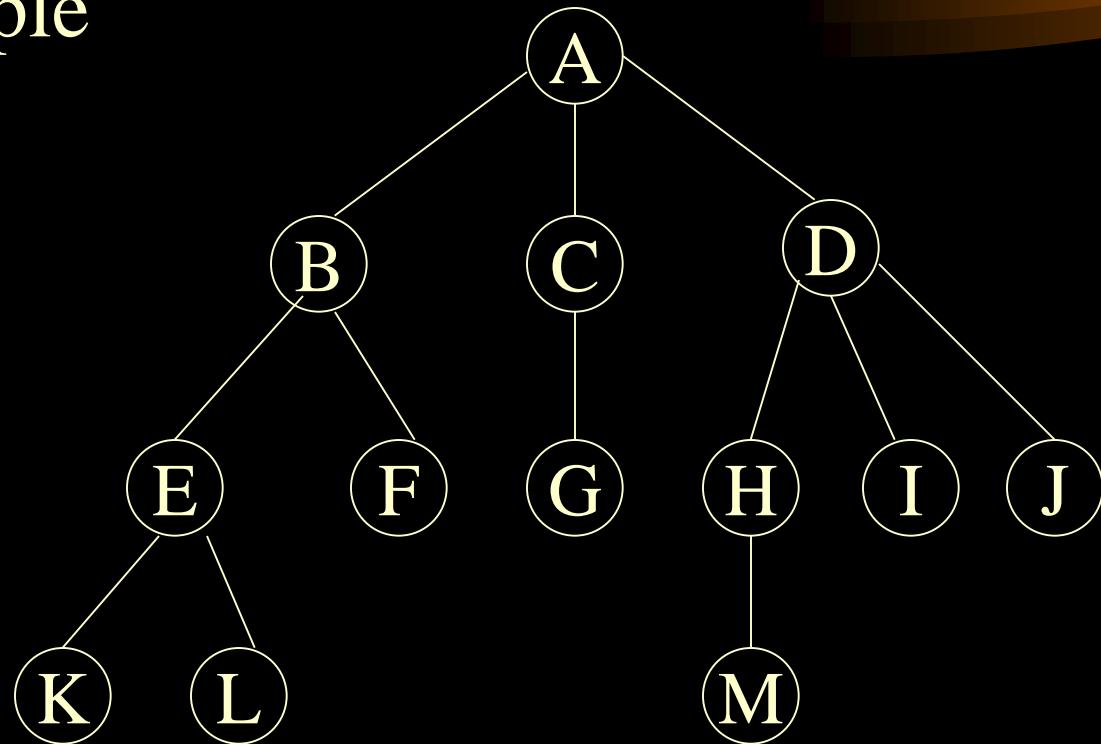
A tree T is a collection of nodes(element).

The collection can be empty;

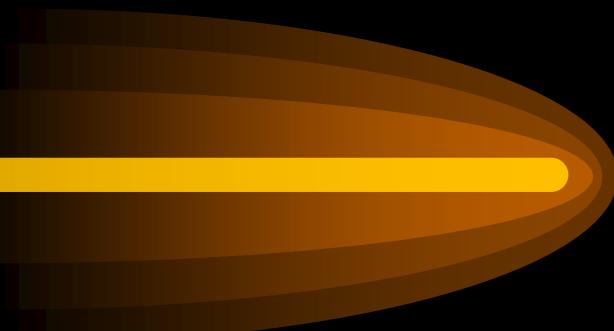
otherwise, a tree consists of a distinguished node r ,
called the root, and zero or more nonempty(sub)trees T_1 ,
 T_2 ,, T_k

4.1 Tree

example



4.1 Tree



2.Terminology

Degree of an elements(nodes): the number of
节点的度数.
children it has.

Degree of a tree: the maximum of its element
树的度数.
degrees

Leaf: element whose degree is 0
树叶.

Branch: element whose degree is not 0
分支节点.

4.1 Tree

Level:

the level of root is 0 (1) 树根的层号

the level of an element =

the level of its parent + 1

Depth(Height) of a tree: 树高

the maximum level of its elements

注意题目要求中
的“层”和“高度”

4.2 Binary Trees 二叉树.

1. **Definition:** A binary tree t is a finite (possibly empty) collection of elements.

{ empty
When the binary tree is not empty:

- It has a root element
- The remaining elements(if any) are partitioned into two binary trees, which are called the left and right subtrees of t .

度数最多为2.

子树有顺序.

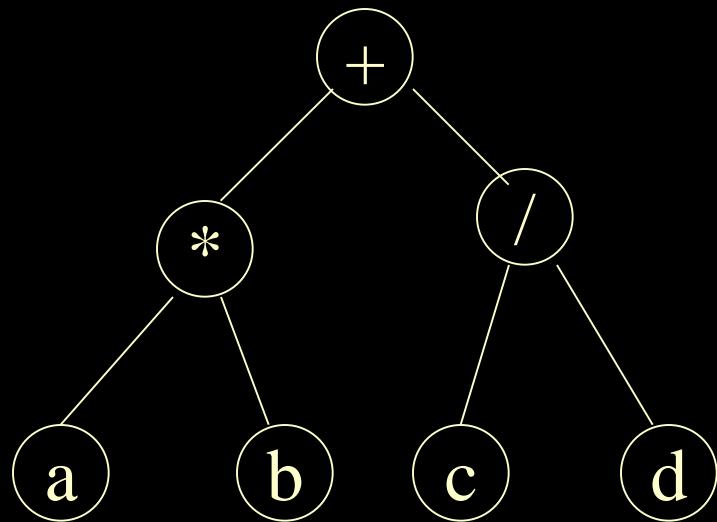
5种基本构型：

4.2 Binary Trees

2. The essential differences between a binary tree and a tree are:
 - 1) Each element **in a binary tree** has exactly two subtrees(one or both of these subtrees may be empty).
Each element **in a tree** can have any number of subtrees.
 - 2) The subtrees of each element **in a binary tree** are ordered.
That is, we distinguish between the left and the right subtrees.
The subtrees **in a tree** are unordered.

4.2 Binary Trees

Example of a binary tree



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

4.3 Properties of binary trees

Property 1. The drawing of every binary tree with n elements($n > 0$) has exactly $n-1$ edges.

通用。1个节点 \Rightarrow 0条边。

Property 2. The number of elements at level i is at most 2^i ($i \geq 0$).
原因：树中无环。
二叉树 k层 至多 2^k 个元素。

4.3 Properties of binary trees

↑说明树根是第0层.

Property 3. A binary tree of height h , $h \geq 0$, has at least $h+1$ and at most $2^{h+1} - 1$ elements in it.

$$ht \leq \leq 2^0 + 2^1 + \dots + 2^h = 2^{h+1} - 1$$

proof of property 3:

$$\sum_{i=0}^h 2^i = 2^0 + 2^1 + \dots + 2^h = 1 * (1 - 2^{h+1}) / (1 - 2) = 2^{h+1} - 1$$

4.3 Properties of binary trees



Property 4. If number of leaves is n_0 , and the number of the 2 degree elements is n_2 , then $n_0 = n_2 + 1$.

Proof: 证法一 数学归纳法.

设: 度为1的结点数是 n_1 个

$$n = n_0 + n_1 + n_2$$

总结点数的2种算法: $n = B+1$ 这里B为分支数

$$\begin{aligned} \textcircled{1} \quad n_0 + n_1 + n_2 &= 1 * n_1 + \underbrace{2 * n_2}_{\text{根}} + 1 \\ n_0 &= n_2 + 1 \quad \textcircled{2} \quad \frac{1}{\text{的B结点}} \end{aligned}$$

4.3 Properties of binary trees

性质3的反向

Property 5. The height of a binary tree that contains n ($n \geq 0$) element is at most $n-1$ and at least

向上取整
大于或等于
 $\lceil \log_2(n+1) \rceil - 1$ $-1 \leq h+1 \leq \log_2(n+1)$

proof: Since there must be at least one element at each level, the height cannot exceed $n-1$.

From property 3, we know $n \leq 2^{h+1} - 1$,

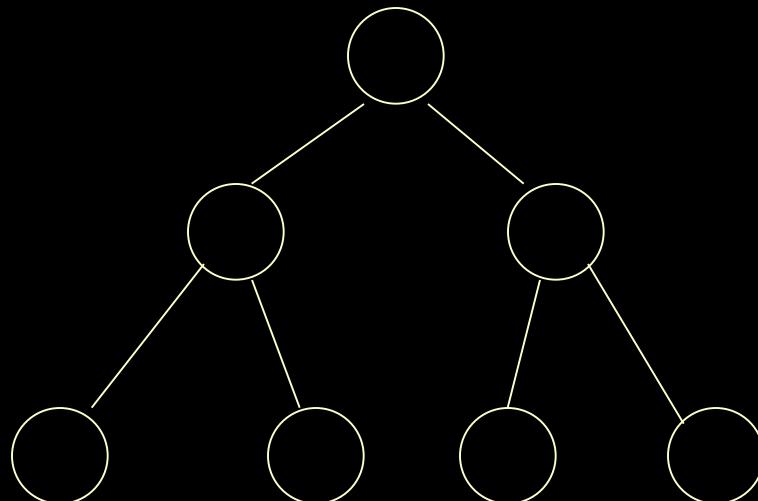
so, $h \geq \log_2(n+1) - 1$, since h is an integer, we get

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

4.3 Properties of binary trees

Definition of a full binary tree :

A binary tree of height h that contains exactly $2^{h+1}-1$ elements is called a full binary tree.



4.3 Properties of binary trees

完全二叉树.

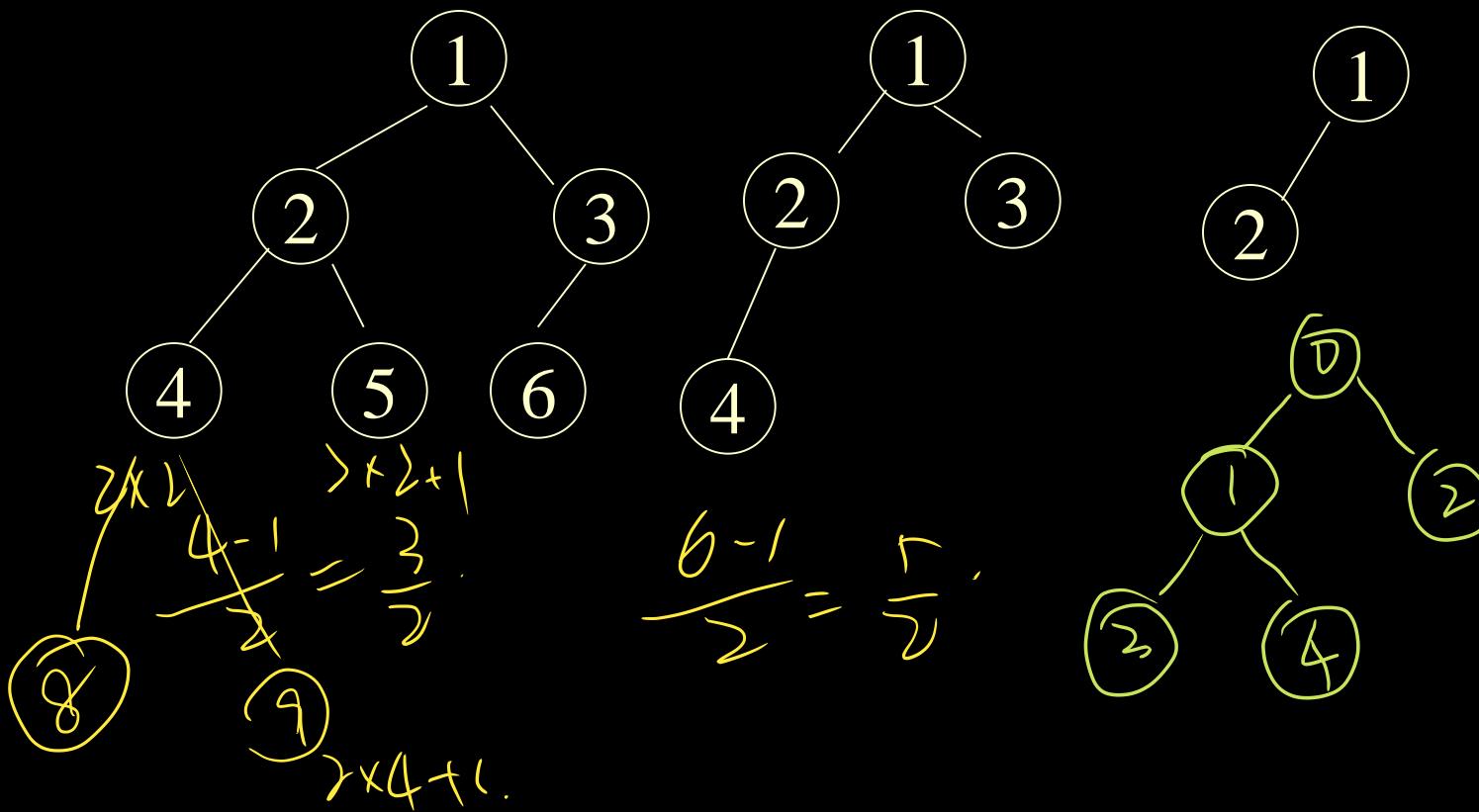
Definition of a complete binary tree:

Suppose we number the elements in a full binary tree of height h using the number 1 through $2^{h+1}-1$. We began at level 0 and go down to level h . Within levels the elements are numbered left to right.

Suppose we delete the k elements numbered $2^{h+1}-i$, $1 \leq i \leq k$, the resulting binary tree is called a complete binary tree.

4.3 Properties of binary trees

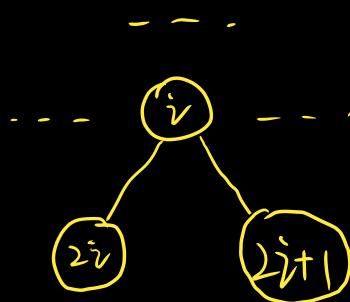
Example of complete binary trees



4.3 Properties of binary trees

Property 6. Let $i, 0 \leq i \leq n-1$, be the number assigned to an element of a complete binary tree. The following are true.

- 1) if $i=0$, then this element is the root of the binary tree.
if $i>0$, then the parent of this element has been assigned the number $\lfloor (i-1)/2 \rfloor$ 大于或等于的最小整数.
- 2) if $2*i+1 > n$, then this element has no left child. Otherwise, its left child has been assigned the number $2*i+1$.



若根为 i
将上边公式替换为 $i-1$
 $i-1$ i
 j

4.3 Properties of binary trees

- 3) if $\underline{2*i+2 >= n}$, then this element has no right child,
Otherwise its right child has been assigned the number
 $2*i+2$.

4.4 Representation of binary tree

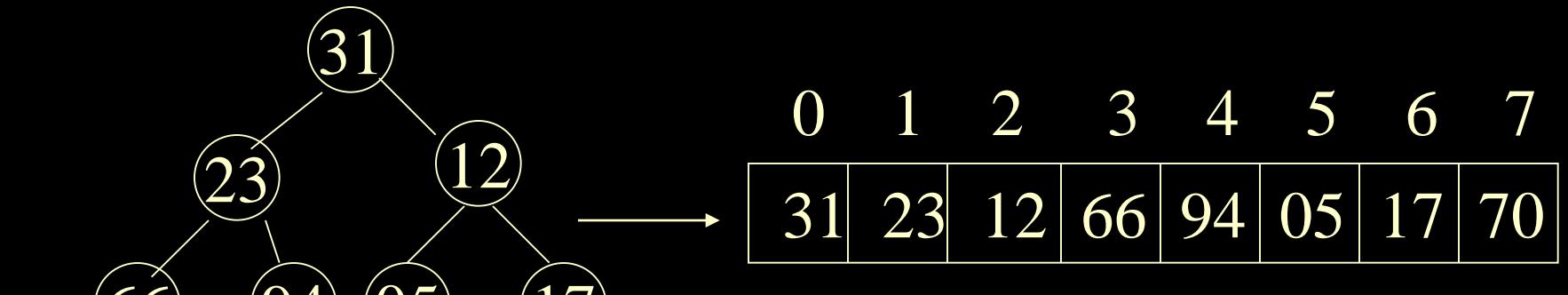
物理实验

1. Formula-Based Representation (array representation)

The binary tree to be represented is regarded as a complete binary tree with some missing elements.

4.4 Representation of binary tree

Example of a complete binary tree (array representation)



逐层操作：找父节点

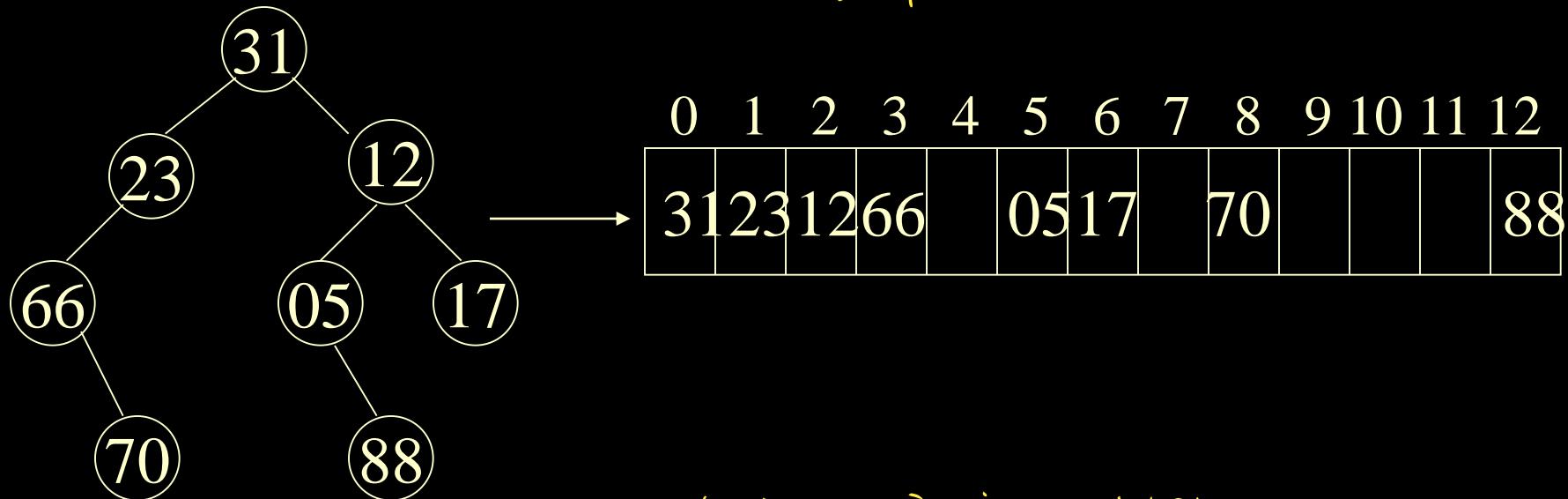
找左/右子节点

用性质6实现

4.4 Representation of binary tree

Example of a common binary tree (array representation)

空部分数组



当树 比较接近完全二叉树时
此物理层实现协议效率比较好。

4.4 Representation of binary tree

树的链接表示法.

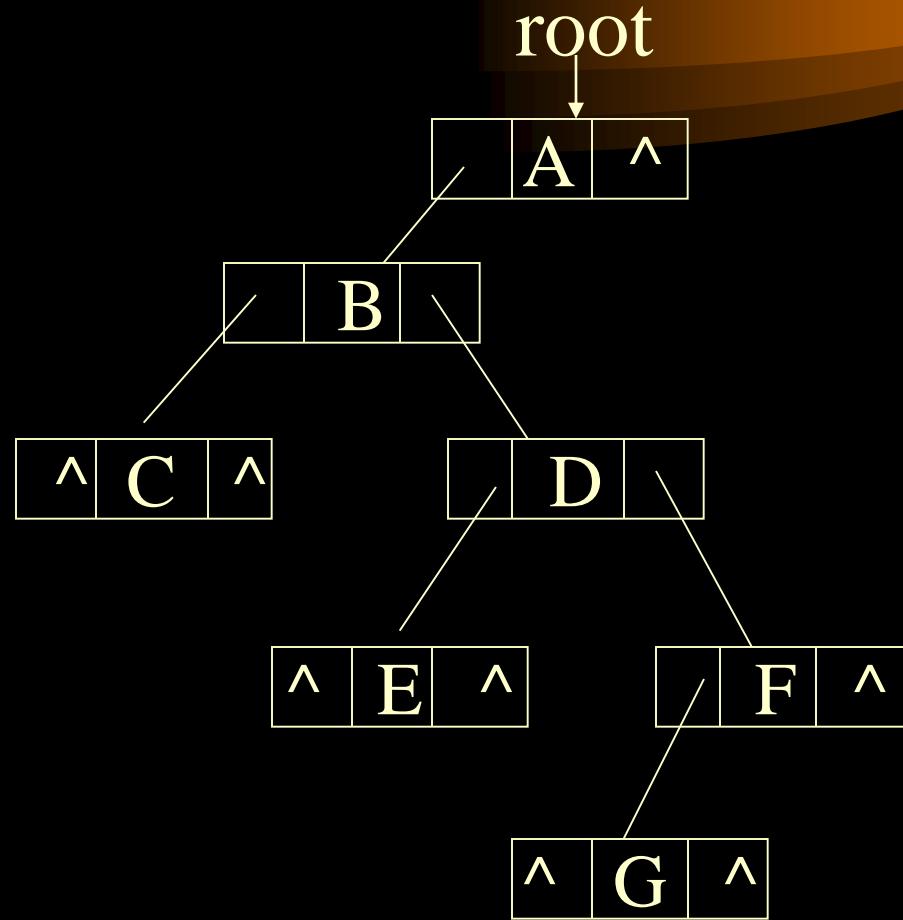
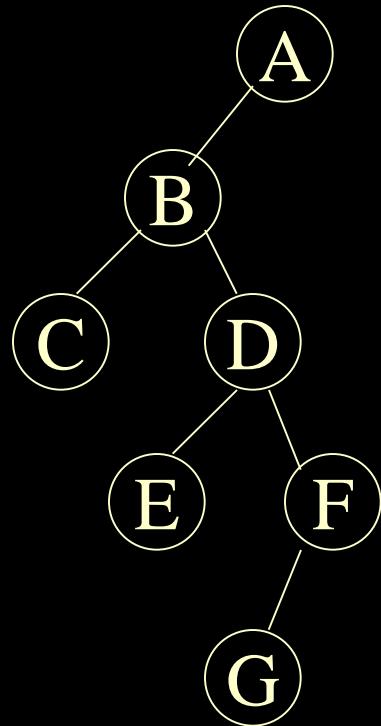
2. Linked representation(also called L-R linked storage)
不能叫单链表.

The node structure:

| | | |
|-----------|------|------------|
| LeftChild | data | RightChild |
|-----------|------|------------|

4.4 Representation of binary tree

Example



4.4 Representation of binary tree

3. Represented by Cursor 静态数组实现

| | data | leftchild | rightchild |
|---|------|-----------|------------|
| 0 | A | 1 | -1 |
| 1 | B | 2 | 3 |
| 2 | C | -1 | -1 |
| 3 | D | 4 | 5 |
| 4 | E | -1 | 6 |
| 5 | F | -1 | -1 |
| 6 | G | -1 | -1 |

-1表示空指针，不能用0.

4.4 Representation of binary tree

Node class for linked binary trees

```
class BianryNode
```

```
{
```

```
    BinaryNode(){Left=Right=0;}
```

```
    BinaryNode(Object e)
```

```
        {element=e; Left=Right=0;}
```

```
    BinaryNode(Object e, BinaryNode l, BinaryNode r)
```

```
        {element=e; Left=l; Right=r; }
```

```
};
```

{ Object element;
 BinaryNode left; //left subtree
 BinaryNode right; //right subtree

4.5 Common binary tree operations

the abstract data type binary tree

- Create()
- IsEmpty()
- Root(x)
- MakeTree(root, left, right)
- BreakTree(root, left, right)
- PreOrder
- InOrder
- PostOrder
- LevelOrder

} 遍历

4.5 Common binary tree operations

The Class BinaryTree

1. Binary tree class

```
template<class T>class BinaryTree  
{  
public: ] C++ 的写法  
    BinaryTree(){root=0;};  
    ~BinaryTree();  
    bool IsEmpty()const  
    {return ((root)?false:true);};  
    bool Root(T& x)const;  
    void MakeTree(const T& data,  
                  BinaryTree<T>& leftch, BinaryTree<T>& rightch);
```

4.5 Common binary tree operations

```
void BreakTree(T& data , BinaryTree<T>& leftch,  
               BinaryTree<T>& rightch);
```

```
{ void PreOrder(void(*visit)(BinaryNode<T>*u))  
    { PreOrder(visit, root); } 内部巡回  
    void InOrder(void(*visit)(BinaryNode<T>*u))  
    { InOrder(visit, root); }  
    void PostOrder (void(*visit)(BinaryNode<T>*u))  
    { PostOrder(visit, root); }  
    void LevelOrder  
    (void(*visit)(BinaryNode<T> *u));
```

4.5 Common binary tree operations

```
private:
```

```
    BinaryNode<T>* root;    // 树里 只维护树根这一棵树
```

```
    void PreOrder(void(*visit)(BinaryNode<T> *u),  
                  BinaryNode<T>*t);
```

```
    void InOrder(void(*visit)(BinaryNode<T> *u),  
                  BinaryNode<T>*t);
```

```
    void PostOrder(void(*visit) (BinaryNode<T> *u),  
                  BinaryNode<T>*t);
```

```
};
```

4.5 Common binary tree operations

- In this class ,we employ a linked representation for binary trees.
- The function **visit** is used as parameter to the traversal methods,so that different operations can be implemented easily

4.5 Common binary tree operations

2.Implementation of some member functions

Template<class T>



```
void BinaryTree<T>::MakeTree(const T& data,  
                                BinaryTree<T>& leftch, BinaryTree<T>& rightch)
```

```
{ root=new BinaryNode<T>(data, leftch.root,  
                           rightch.root);
```

```
leftch.root=rightch.root=0; } 原来的清零.
```

4.5 Common binary tree operations

```
template<class T>
void BinaryTree<T>::BreakTree(T& data,
                                BinaryTree<T>& leftch, BinaryTree<T>& rightch)
{ if(!root)throw BadInput(); //tree empty
  data=root. element;
  leftch.root=root. Left;
  rightch.root=root. Right;
  delete root;    释放      O
  root=0;        返回      `
```

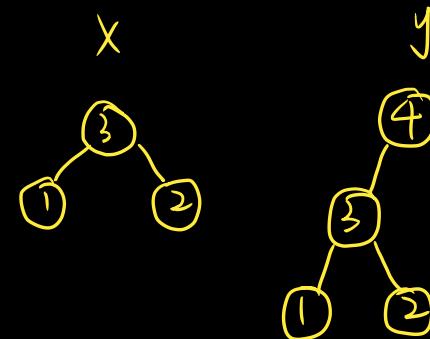
4.5 Common binary tree operations

3. Application of class BinaryTree(Create BinaryTree)

```
#include<iostream.h>
#include "binary.h"
int count=0;  BinaryTree<int>a,x,y,z;
```

```
template<class T>
void ct(BinaryTreeNode<T>*t){count++;} 当前访问的节点 回调函数
```

```
void main(void)
{
    a.MakeTree(1,0,0);
    z.MakeTree(2,0,0);
    x.MakeTree(3,a,z);
    y.MakeTree(4,x,0);
    →y.PreOrder(ct); 回调 序列用函数
    cout<<count<<endl;
}
```



4.6 Binary Tree Traversal

Each element is visited exactly once

V-----表示访问一个结点

L-----表示访问V的左子树

R-----表示访问V的右子树

VLR LVR LRV ~~VRL~~ ~~RVL~~ ~~RLV~~ (L一定在R前)

这些没有定义

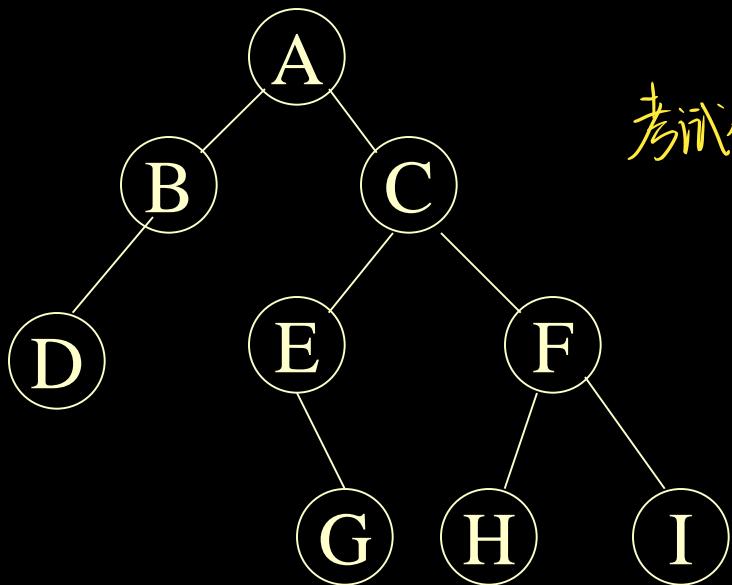
如果需要实现

可以构造一个所有左右节点
不报的二叉树

- Preorder VLR
- Inorder LVR
- Postorder LRV
- Level order 按层次

4.6 Binary Tree Traversal

Example of binary tree traversal



考試公考~ {

Preorder : ABDCEGFHI

Inorder : DBAEGCHFI

Postorder : DBGEHIFCA

Level order: ABCDEFGHI

遍历次序 应用： 例如 先放子节点还是父结点？ 的问题。
(双子节点串者(根据)次结点。)

4.6 Binary Tree Traversal

Preorder traversal

```
template<class T>
void PreOrder(BinaryNode<T>* t)
{// preorder traversal of *t.
    if(t){  visit(t); 这是回溯
            PreOrder(t→Left);
            PreOrder(t→Right);
    }
}
```

4.6 Binary Tree Traversal

Inorder traversal

```
template<class T>
void InOrder(BinaryNode<T>* t)
{ if(t){ InOrder(t->Left);
    visit(t);
    InOrder(t->Right);
}
}
```

4.6 Binary Tree Traversal

Postorder traversal

```
template<class T>
void PostOrder(BinaryNode<T>* t)
{ if(t){
    PostOrder(t→Left);
    PostOrder(t→Right);
    visit(t);
}
}
```

4.6 Binary Tree Traversal

Example:

Inorder traversal:

4.6 Binary Tree Traversal

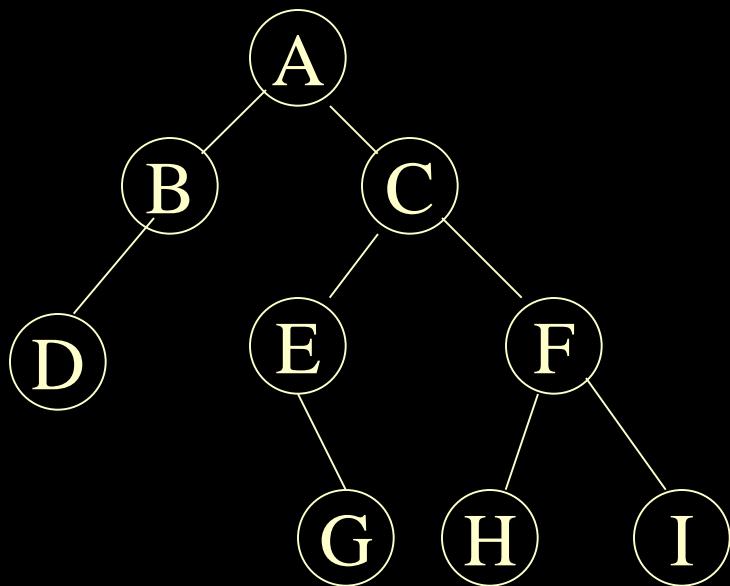
{ 前面是 深度优先遍历

Level order 广度优先遍历 和杨辉三角类似 !!

不是递归

用队列 先进先出.

it is a non-recursive function and a ~~queue~~ is used.



4.6 Binary Tree Traversal

Level order

```
template<class T>
void LevelOrder(BinaryNode<T>* t)
{ LinkedQueue<BinaryNode<T>*> Q;
  while(t){
    visit(t); //visit t
    if(t->Left) Q.Add(t->Left);
    if(t->Right) Q.Add(t->Right);
    try{Q.Delete(t);}
    catch(OutOfBounds){return;}
  }
}
```

註解：

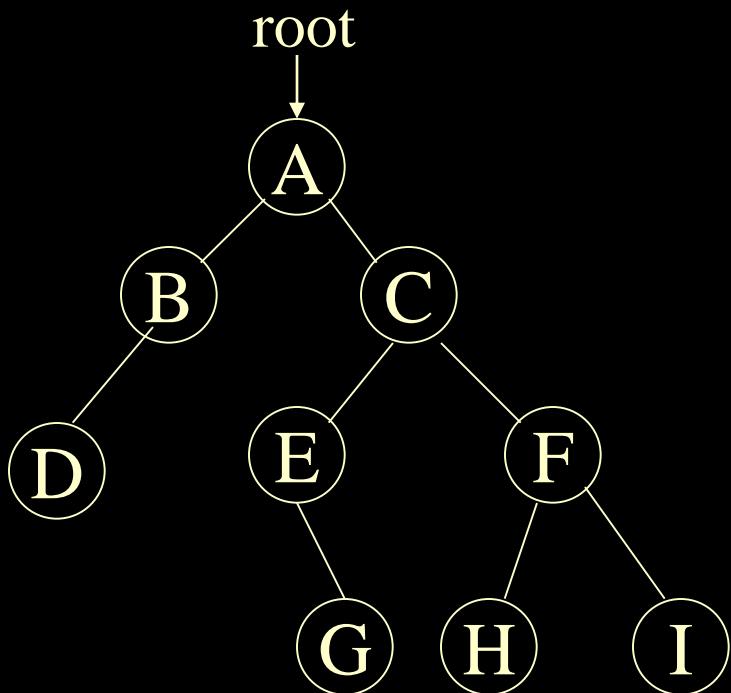
- A. $t = A$. $\cancel{B} \cancel{C}$
- B. \cancel{A} \cancel{D}
- C. \cancel{B} \cancel{E}
- D. \cancel{C} \cancel{F}
- E. \cancel{D} \cancel{G}
- F. \cancel{E} H
- G.

H
I

Inorder, Postorder non-recursive algorithm

- Inorder non-recursive algorithm

用栈



中序

Inorder non-recursive algorithm

非递归实现

```
void Inorder(BinaryNode <T> * t)
```

```
{ Stack<BinaryNode<T>*> s(10);
```

```
BinaryNode<T> * p = t;
```

```
for ( ; ; )
```

```
{ 1) while(p!=NULL)
```

```
{ s.push(p); P进栈  
p = p->Left; }
```

```
2) if (!s.IsEmpty())
```

```
{ p = s.pop(); P出栈  
cout << p->element;
```

```
p = p->Right; } 右子树为空的情况
```

```
}
```

```
else return; 戈为完了
```

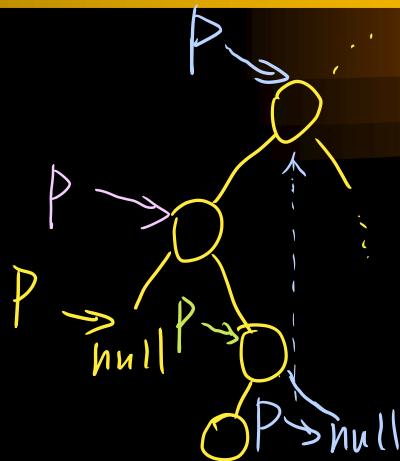
```
}
```

```
}
```

栈的作用：缓存树根

先遍历左子树

遍历完左子树，队根退栈

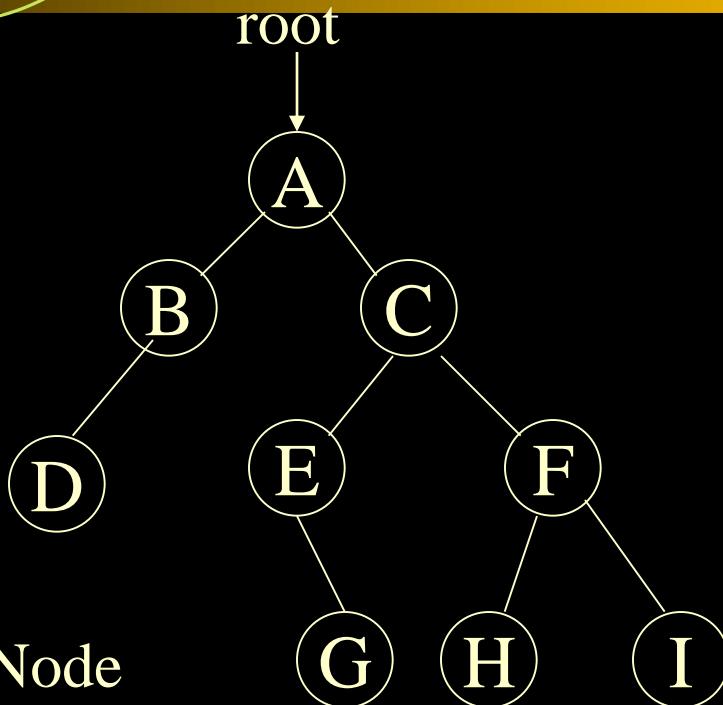


跳出 while 循环

(visit(p)) 圆圈函数是打印

Inorder, Postorder non-recursive algorithm

• Postorder non-recursive algorithm



struct StkNode

```
{ BinaryNode <T> * ptr;  
    int tag; 询问第几次访问  
}
```

根结点会“回”两次

Postorder non-recursive algorithm

看递归
和非递归

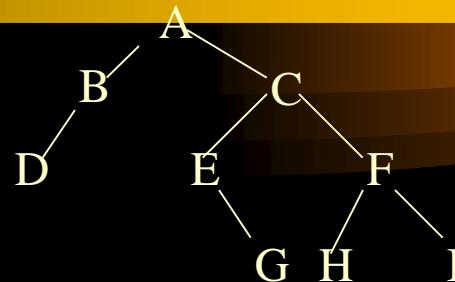
```
void Postorder(BinaryNode <T> * t)
{ Stack <StkNode<T>>s(10);
  StkNode<T> Cnode;
  BinaryNode<T> * p = t;
  for( ; ; )
  { 1)while (p!=NULL)      ?
     { Cnode.ptr = p;  Cnode.tag = 0; s.push(Cnode);
       p = p->Left;
     }  
    }  
    P为底  
  2)Cnode = s.pop();  p = Cnode.ptr;  
  3)while ( Cnode.tag == 1) //从右子树回来 该visit根了
     { cout << p->element;
       if ( !s.IsEmpty() )
         { Cnode = s.pop();  p = Cnode.ptr;  }
       else return; 遍历树的右子树了
     }
  4)Cnode.tag = 1; s.push(Cnode);  p = p->Right; //从左子树回来
} //for
}
```



建立一棵二叉树的方法

- 利用MakeTree函数
- 利用先序、中序唯一的构造一棵二叉树

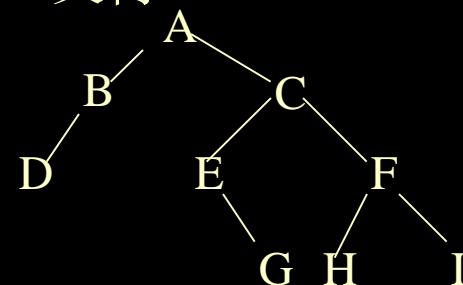
先序: ABDCEGFHI
中序: DBAEGCHFI



*3.
课程不考

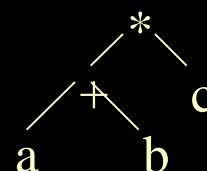
利用二叉树的广义表表示来构造一棵二叉树

A(B(D), C(E(,G), F(H,I))) →



- 利用二叉树的后缀表示来构造一棵二叉树

(a+b)*c → ab+c* →

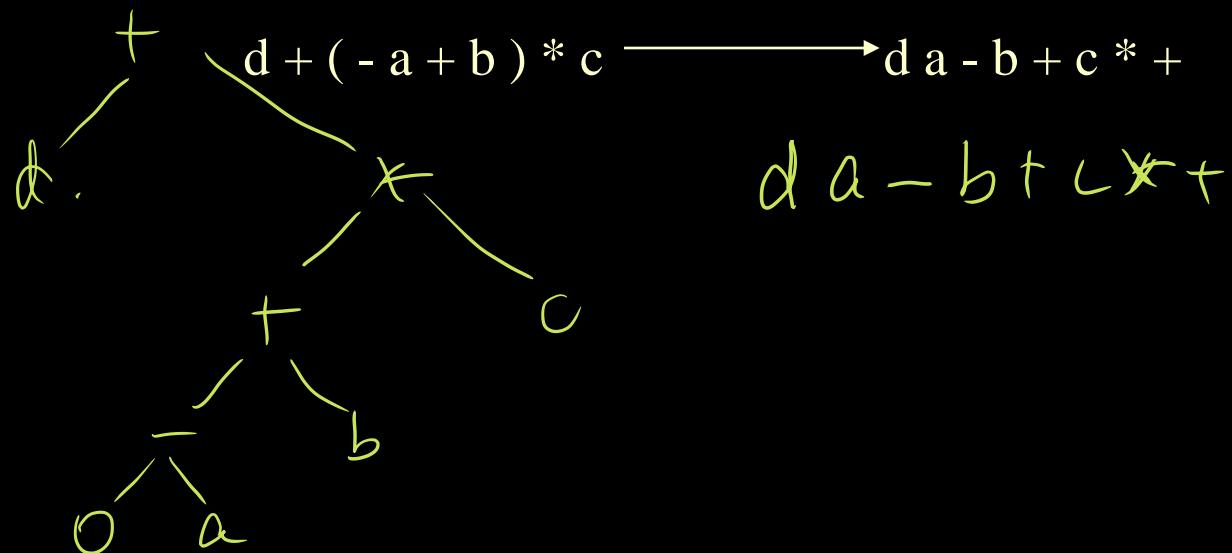


建立一棵二叉树的方法

4. 利用二叉树的后缀表示来构造一棵二叉树(习题)

$$(a + b) * c \longrightarrow a b + c *$$

$$(-a + b) * c \longrightarrow a - b + c *$$



考但简单

2. 利用先序、中序唯一的构造一棵二叉树

string

不考

1. 字符串的有关概念
串的定义、术语、基本操作
2. 字符串的类说明
3. 部分成员函数的实现
4. 利用前序、中序序列建立一棵树

string

1.字符串（简称串）的定义以及一些术语

*串：是n ($n \geq 0$) 个字符的一个有限序列，开头结尾用双引号“”括起来。

例如：B= “ structure ” (B为串名)

*串的长度：串中所包含的字符个数n (不包括分界符‘’‘’，也不包括串的结束符‘\0’)

*空串：长度为0的串。或者说只包含串结束符‘\0’的串

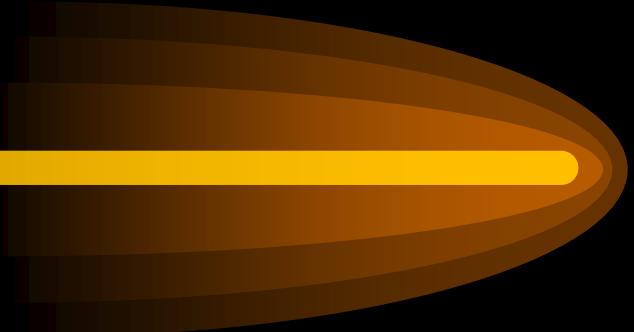
注意：“\0”不等于“\0”，空串不等于空白串

*子串：串中任一连续子序列

例子：B= “ peking ” 则 空串“”、“ki”、“peking”都是B的子串

但“pk”不是B的子串

string



*串的基本操作：

构造一个空串；

求串长；

两个串的连接（并置）；

取子串；

求一个子串在串中第一次出现的位置等。

string

Java与C/C++的不同处：

Java语言的字符串不是字符数组，所以不能以字符数组方式进行一些操作。如，`str[1] = "a"`是错误的，而只能通过方法（函数）来进行操作。

`int length()`

`boolean equals(Object obj)`

`char charAt(int index)`

`String substring (int beginIndex)`

`String substring (int beginIndex, int endIndex)`

string

2. 字符串的类说明

```
const int maxlen=128;
```

```
class String
```

```
{ public:
```

```
    String(const String & ob);
```

```
    String(const char * init);
```

```
    String( );
```

```
    ~String( ) {delete[ ] ch;}
```

```
    int Length( )const {return curlen;}
```

```
    String & operator( )(int pos, int len); //取子串
```

```
    int operator ==(const String & ob) const
```

```
        { return strcmp(ch, ob.ch)==0; } //判别相等否?
```

```
    int operator !=(const String & ob) const
```

```
        { return strcmp(ch, ob.ch)!=0; }
```

```
    int operator ! ( ) const {return curlen==0;}
```

string

```
String & operator = (const String & ob); //串赋值  
String & operator +=(const String & ob); //并置运算  
char & operator[ ](int i);  
int Find(String pat) const;  
private:  
    int curLen;  
    char * ch;  
}
```

string

3. 部分成员函数的实现

Taking Substring

String & String::operator()(int pos, int len)

```
{ String *temp=new String;
```

```
if (pos<0||pos+len-1>=maxlen ||len<0)
{   temp->curLen=0; temp->ch[0]='\0';}
```

```
else{ if (pos+len-1>=curLen) len=curLen-pos;
```

```
    temp->curLen=len;
    for(int i=0, j=pos; i<len; i++, j++)
        temp->ch[i]=ch[j];
    temp->ch[len]='\0';
```

```
}
```

```
return *temp;
```

```
}
```

string

Assigning Operate

```
String & String ::operator=(const String &ob)
```

```
{ if (&ob!=this)  
{delete [ ] ch;  
ch=new char[maxLen+1];  
if(!ch) {cerr<< “Out Of Memory! \n”;exit(1);}  
curLen=ob.curLen;  
strcpy(ch, ob.ch);  
}  
else cout<<“Attempted assignment of a String to itself!  
\n”;  
return * this;  
}
```

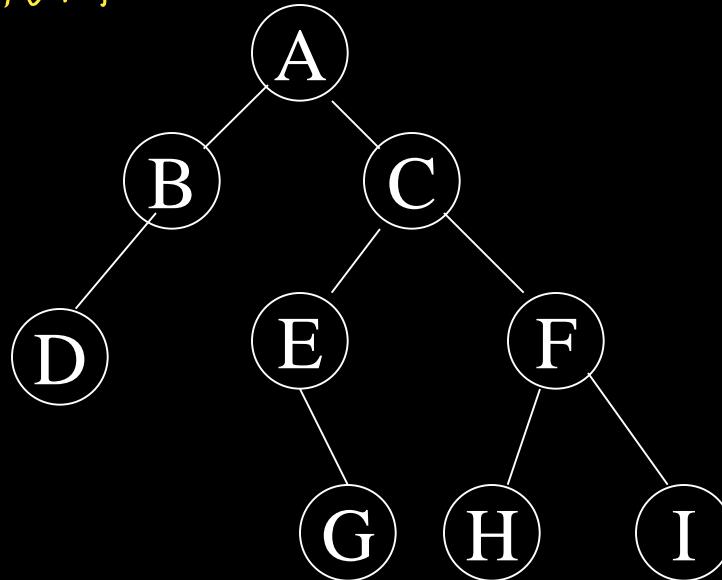
4. Create BinaryTree recursive algorithm

preorder: A B D C E G F H I

inorder: D B A E G C H F I

左子树 根 右子树

$$| \cup | \cup | = 3$$



Create BinaryTree recursive algorithm 1

```
void CreateBT(String pres, ins ; BinaryNode <Type>* & t)
{ int inpos;
```

String prestemp, instemp ;

if (pres.length()== =0) t=NULL;

else { t=new BinaryNode;

t->element=pres.ch[0]; inpos=0;

while (ins.ch[inpos]!=t->element) inpos++; 在中序遍历字符串中找树根

前序的第一个是树根

prestemp=pres(1,inpos);

instemp=ins(0,inpos-1);

CreateBT(prestemp, instemp, t->left);

] 左子树

prestemp=pres(inpos+1, pres.length()-1);

instemp=ins(inpos+1, pres.length()-1);

CreateBT(prestemp, instemp, t->right);

] 右子树

}

}

Create BinaryTree recursive algorithm 1

public:

BinaryTree(string pre, string In)

```
{  createBT( pre, In, root ) ;  
}
```

.....

main()

```
{  BinaryTree t1( "ABHFDECKG", " HBDFAEKCG" );
```

.....

```
}
```

如果已知后序与中序，能否唯一构造一棵二叉树呢？

后序： DBGEHIFCA

中序： DBAEGCHFI

如果已知先序与后序呢？ 特殊情况下 树不唯一

二义性

先序： AB

后序： BA



4.7 ADT and Class Extensions

- **PreOutput():output the data fields in preorder**
- **InOutput():output the data fields in inorder**
- **PostOutput():output the data fields in postorder**
- **LevelOutput():output the data fields in level order**
- **Delete():delete a binary tree,freeing up its nodes**
- **Height():return the tree height**
- **Size():return the number of nodes in the tree**

4.7 ADT and Class Extensions

The height of the tree is determined as:

$$\max\{hl, hr\} + 1$$

```
template<class T>
int BinaryTree<T>::Height(BinaryNode<T> *t) const
{
    if(!t) return 0;
    int hl=Height(t->Left);
    int hr=Height(t->Right);
    if(hl>hr) return ++ hl;
    else return ++hr;
}
```

4.8 Application

1. Binary-Tree Representation of a Tree

树的存储方式: 三种

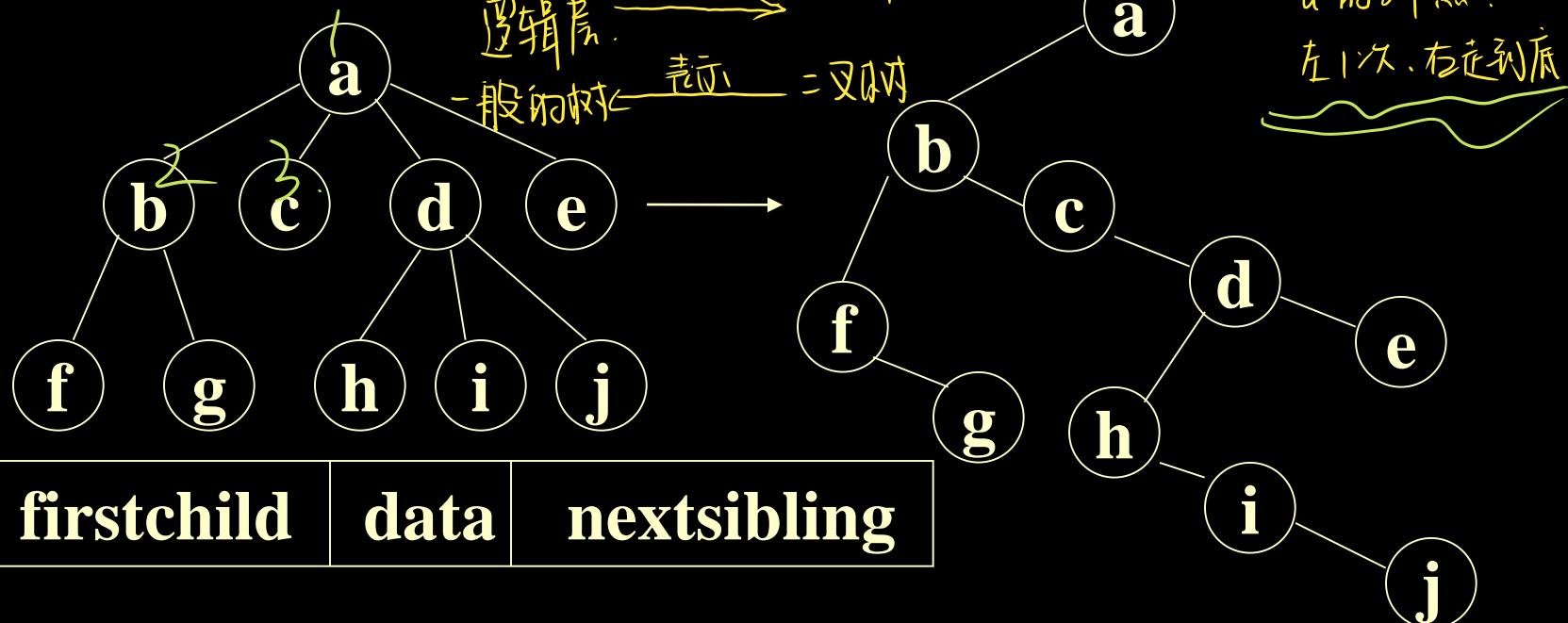
• 广义表表示: $a(b(f,g),c,d(h,i,j),e)$

• 双亲表示法 记录父节点 在有亲子女节点, 不在亲子女节点时适用

• 左子女—右兄弟表示法

| | | | | | |
|---|---|---|---|-------|--|
| a | b | f | g | | |
| 0 | 1 | 2 | 2 | | |

1) Take a tree as a binary tree



4.8 Application

```
class TreeNode:
```

```
    T data;
```

```
    TreeNode *firstchild, *nextsibling;
```

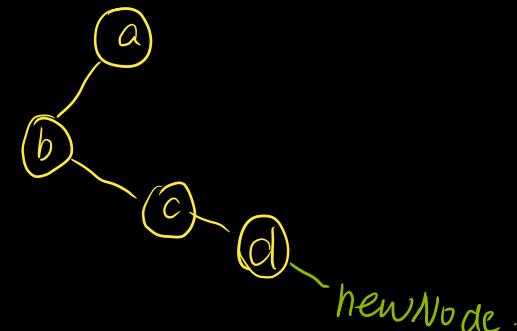
```
class Tree:
```

```
    TreeNode * root, *current;
```

4.8 Application

insert new node in tree

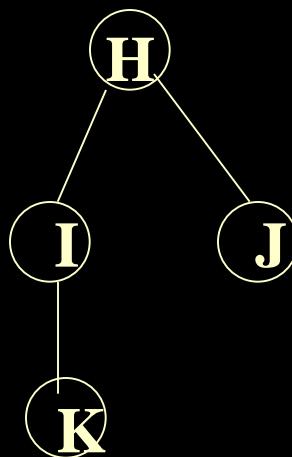
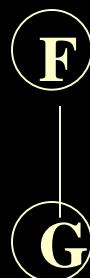
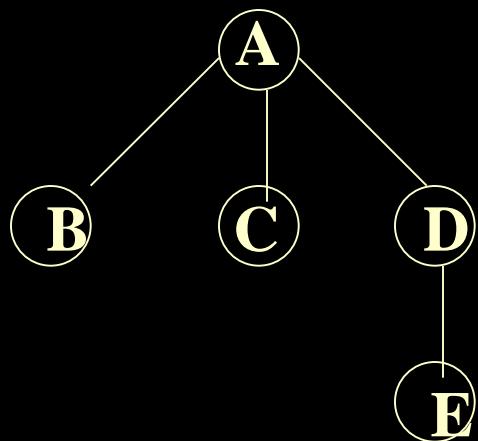
```
template <class T> void Tree <T>::Insertchild(T value)
{ TreeNode<T>*newnode = new TreeNode<T>(value);
  if(current->firstchild == NULL)
    current->firstchild = newnode;      @
  else
    { TreeNode<T>*p = current->firstchild;
      while ( p->nexstsibling!=NULL) p = p->nexstsibling;
      p->nexstsibling = newnode;
    }
}
```



4.8 Application

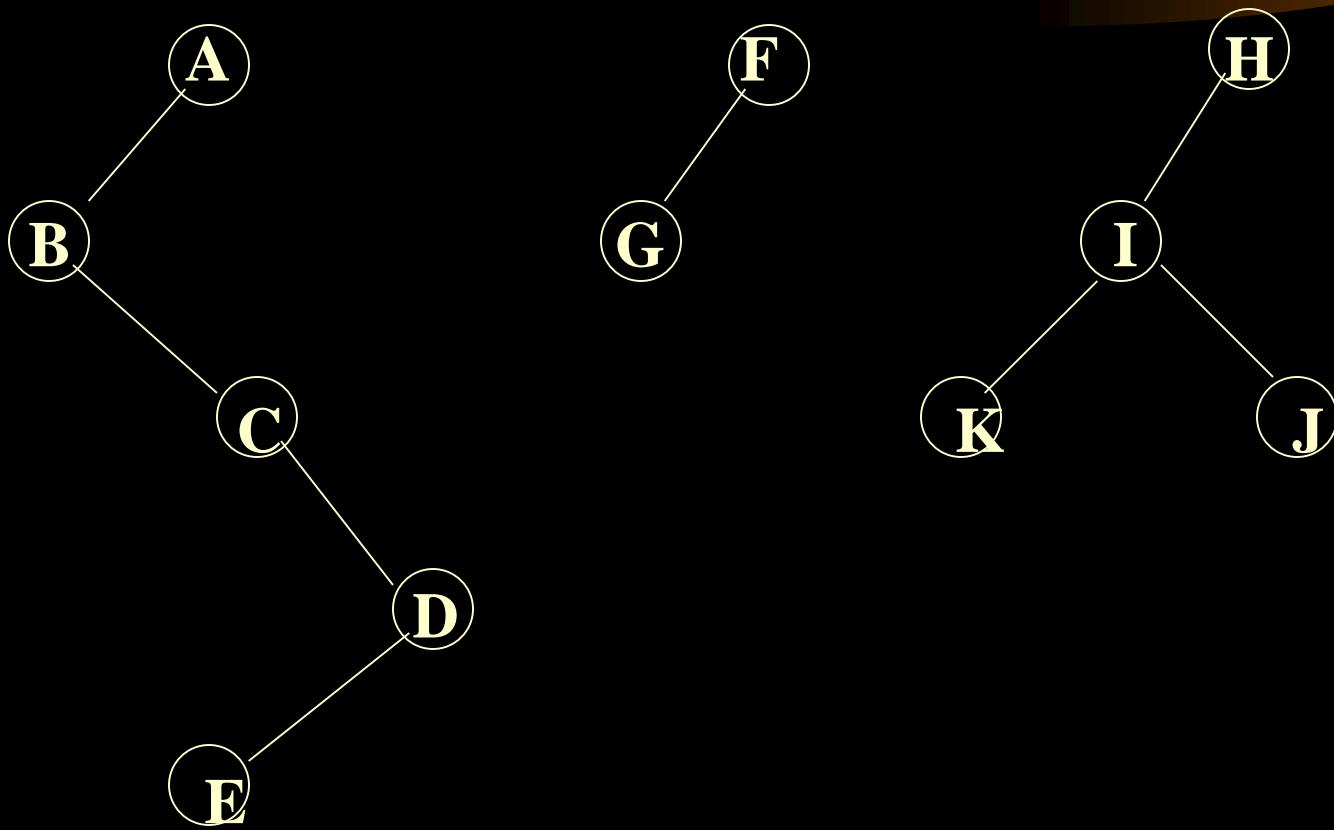
2) Forest ←→ Binary tree

- Forest → Binary tree

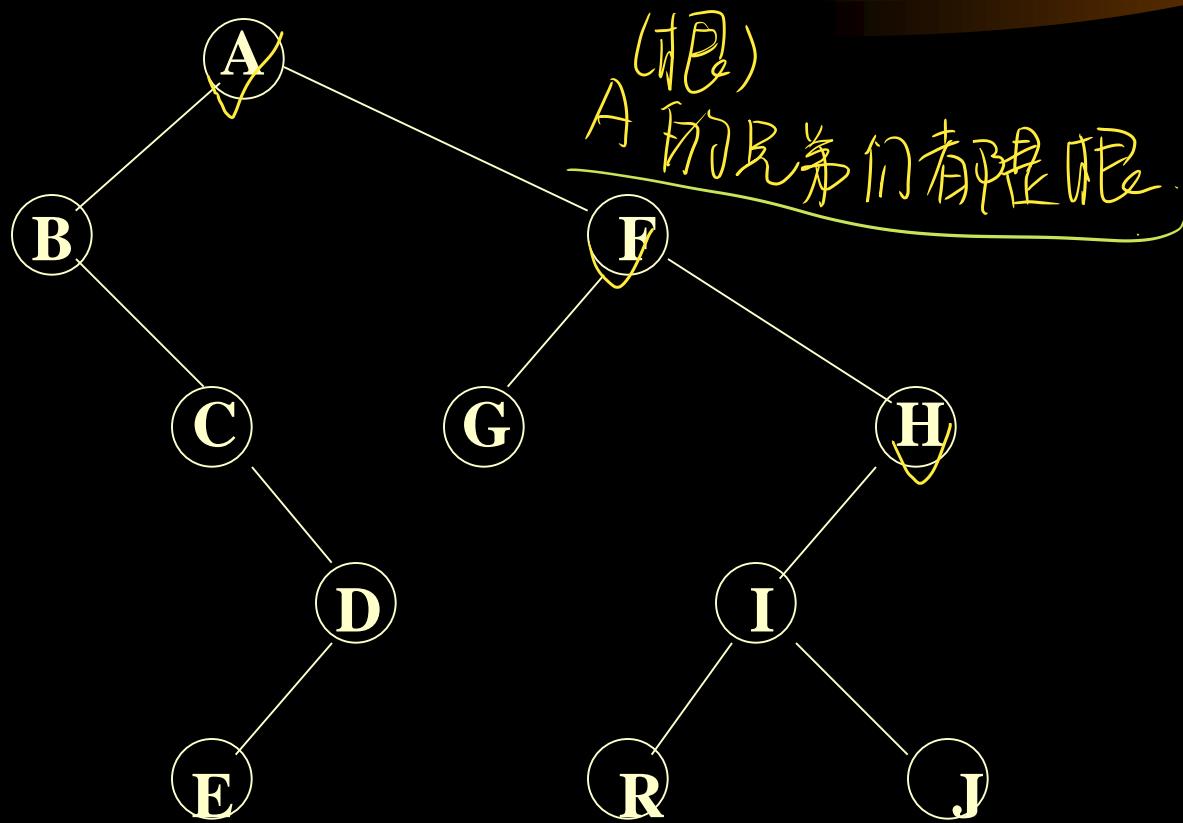


每棵树转为二叉树

用左子甘·右兄弟之法

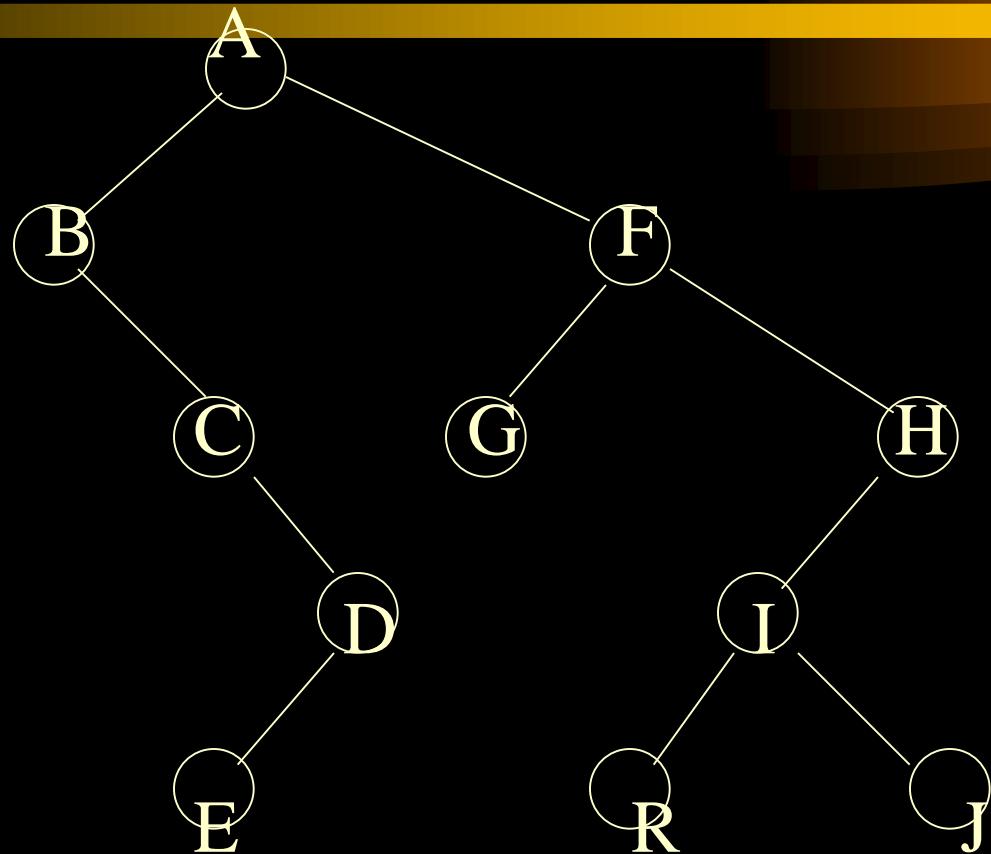


把每棵二叉树根用右链相连



- Binary tree → Forest

用二叉树表示森林



4.8 Application

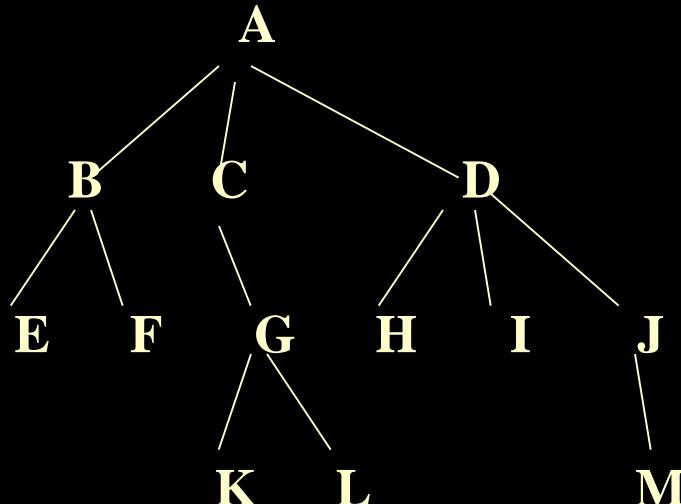
- 3) 树的遍历：深度优先遍历，广度优先遍历
• 深度优先遍历

先序次序遍历（先序）

访问树的根 ——> 按先序遍历根的第一棵子树，第二棵子树，……等。

后序次序遍历（后序）

按后序遍历根的第一棵子树，第二棵子树，……等 ——>
访问树的根。



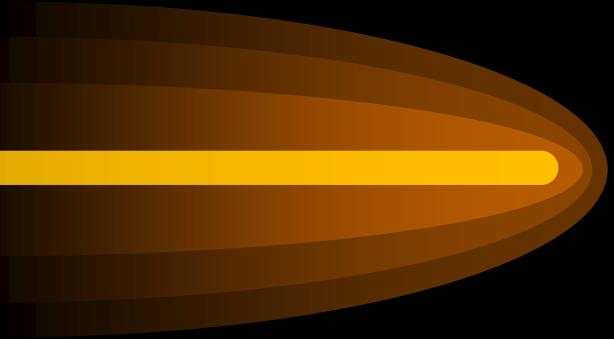
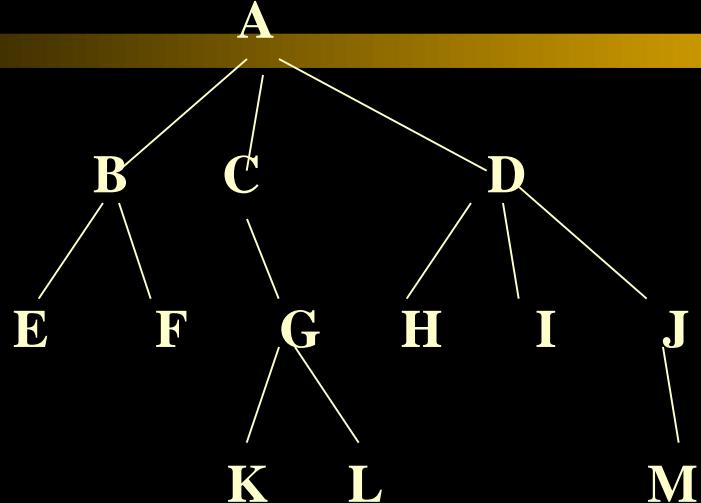
没有中根之说

先根：ABEFCGKLDHIJM与
对应的二叉树的先序一致

后根：EFBKLGCHIMJDA与
对应的二叉树的中序一致

4.8 Application

- 广度优先遍历



分层访问: ABCDEFGHIJKLM

4) 森林的遍历

深度优先遍历

* 先根次序遍历

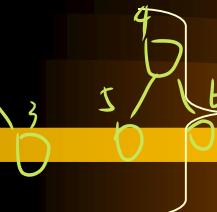
访问F的第一棵树的根

按先根遍历第一棵树的子树森林

按先根遍历其它树组成的森林

注意：这时不是在左子女右兄弟表示中讨论

而是分开的



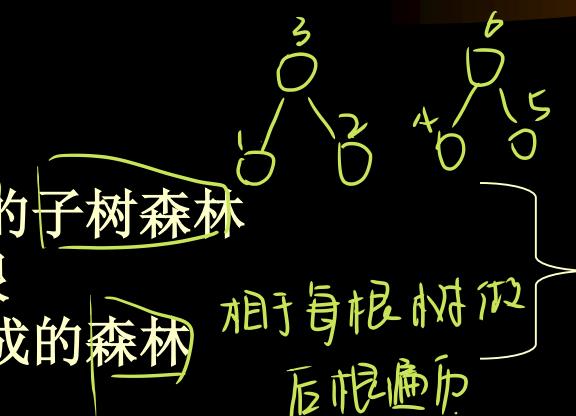
二叉树的先序

* 中根次序遍历

按中根遍历第一棵树的子树森林

访问F的第一棵树的根

按中根遍历其它树组成的森林



二叉树的中序

* 后根次序遍历

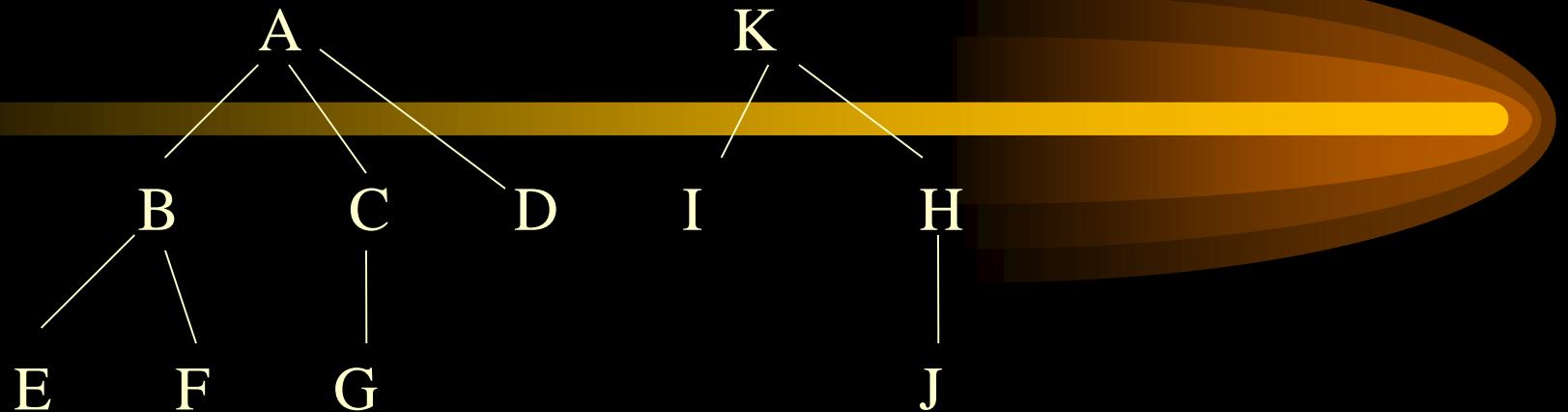
按后根遍历第一棵树的子树森林

按后根遍历其它树组成的森林

访问F的第一棵树的根

奇怪的定义

二叉树的后序



先根: **ABEFCGDKIHJ**

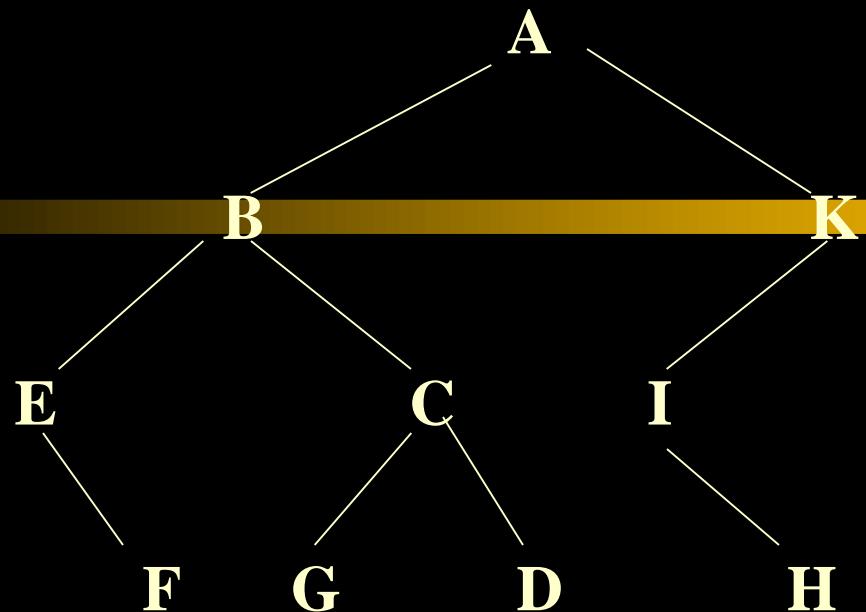
中根: **EFBGCDAIJHK**

后根: **FEGDCBJHIKA**

A B E F C G D K I H J

E F B G C D A I J H K

E F G D C B I J H K / A



后序: FEGDCBJHIKA

广度优先遍历（层次遍历）

AKBCDIHEFGJ

森林的后根遍历

开在子树右兄弟表示森

得利的树 后根遍历

一样

Thread Tree

1. Purpose:

2. Thread Tree Representation

left Thread Tree and right Thread Tree

3. Thread Tree class

1. Purpose:

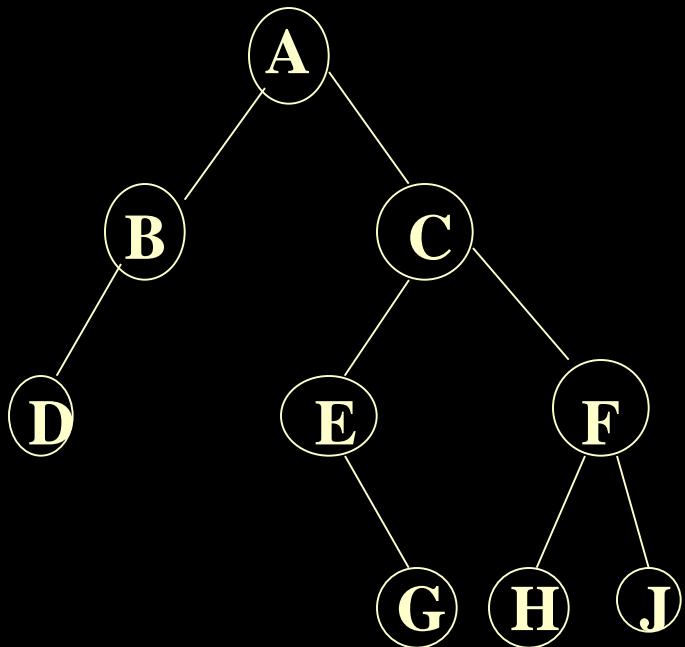
n 个结点的二叉树有 $2n$ 个链域，

其中真正有用的是 $n - 1$ 个，其它 $n + 1$ 个都是空域。

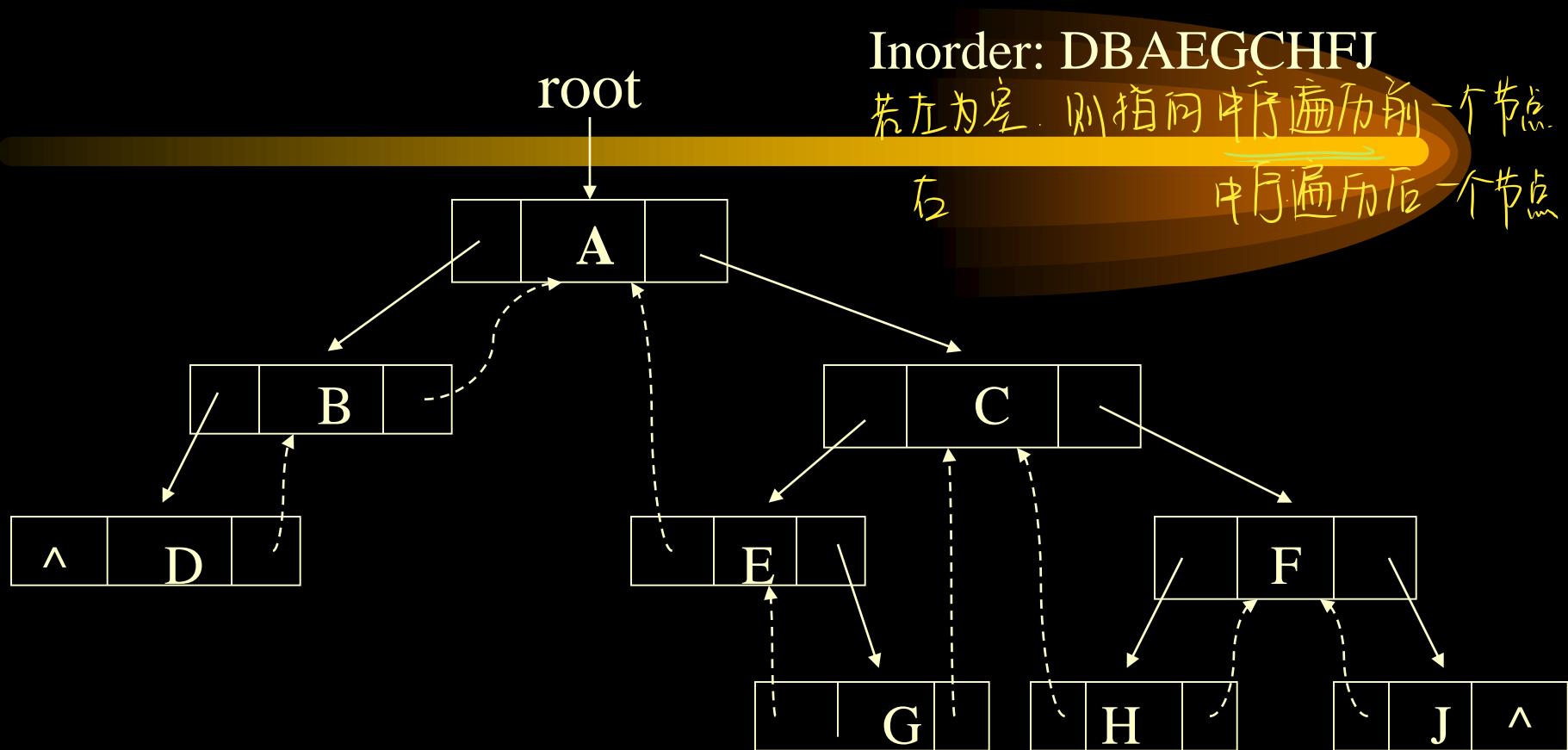
为了充分利用结点中的空域，使得对某些运算更快，如前驱或后继等运算。

Thread Tree

Example:



Thread Tree 中序遍历线索化



preorder ABDCEGFIHJ

2. 机内如何存储

一个结点增加两个标记域:

左行为父

说明 child 的实际指向内存 .



leftThread == { 0 **leftchild** 指向左子女
 1 **leftchild** 指向前驱 (某线性序列)

rightThread == { 0 **rightchild** 指向右子女
 1 **rightchild** 指向后继

3. 线索化二叉树的类声明。

```
template< class Type> class ThreadNode
```

```
{
```

```
    friend class ThreadTree;
```

```
private:
```

```
    int leftThread, rightThread;
```

```
    ThreadNode<Type>* leftchild, *rightchild;
```

```
    Type data;
```

```
public:
```

```
    ThreadNode(const Type item): data(item), leftchild(0),
```

```
        rihgtchild(0), leftThread(0), rightThread(0) { }
```

```
};
```

```
template< class Type> class ThreadTree
{
public:
    // 线索二叉树的公共操作
private:
    ThreadNode<Type> * root;
    ThreadNode<Type> *current
};
```

讨论线索化二叉树的几个算法

1) 按中序遍历中序线索树

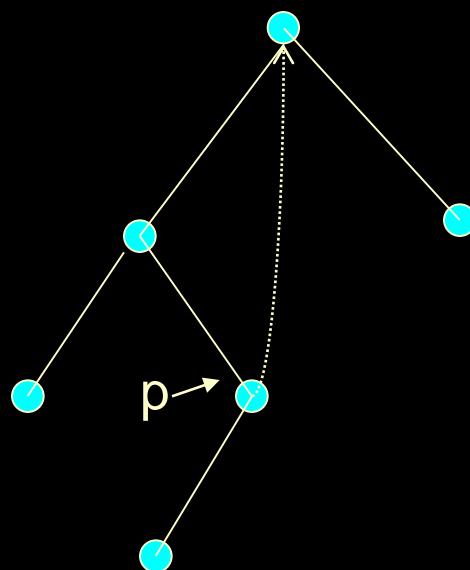
遍历算法（以中序为例）：

递归， 非递归（需用工作栈）。

这里前提是中序线索树， 所以既不要递归， 也不要栈。

遍历算法： * 找到中序下的第一个结点（first）从根向左走到头
* 不断找后继（Next）

如何找后继？



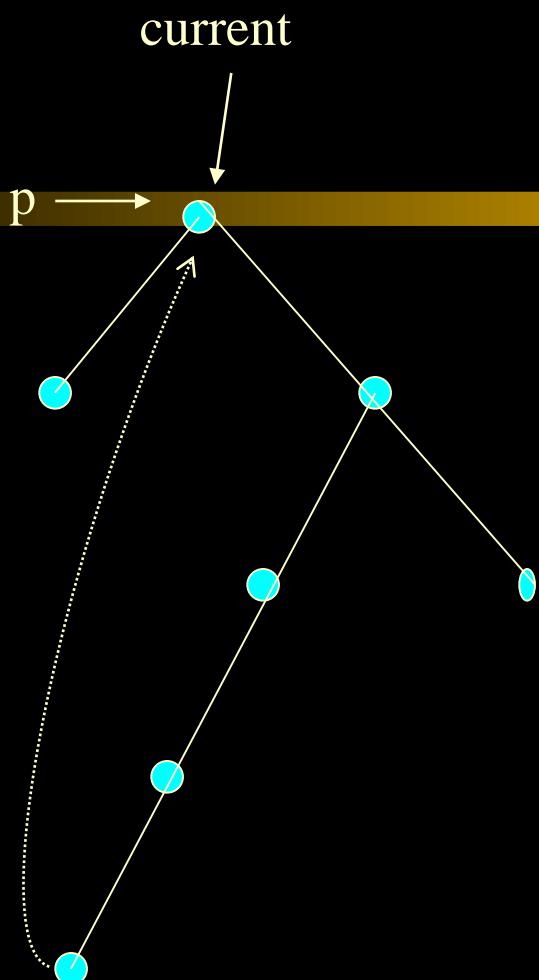
中序

p指结点没有右子树

($p \rightarrow \text{rightthread} = -1$)

则 $p = p \rightarrow \text{rightchild}$

(右链就是后继)



p有右子树

($p->rightThread == 0$)

则 (1) $p=p->rightchild$

(2) $while(p->leftThread == 0)$

$p=p->leftchild;$

左-次 + 向左走到头

先序 while $P \neq \text{根}$

{ $P_{左}tag == 0$ 有左子树

$P=P->leftchild$

无左子树

$P=P->rightchild$

后序 要为每个节点添加父节点字段

① 左庄村

② 无右子树

③

P->leftchild

P->rightchild

P->parent

④ 重标①②③

```
template<class Type>
```

```
ThreadNode<Type>* ThreadInorderIterator<Type>::First()
```

```
{
```

```
    while (current->leftThread==0) current=current->leftchild;
```

```
    return current;
```

```
}
```

```
template<class Type>
```

```
ThreadNode<Type>* ThreadInorderIterator<Type>::Next( )
```

```
{
```

```
    ThreadNode<Type>* p=current->rightchild;
```

```
    if(current->rightThread==0)
```

```
        while(p->leftThread==0) p=p->leftchild;
```

```
    current=p;
```

```
}
```

```
template<class Type>
void ThreadInorderIterator<Type>:: Inorder( )
{
    ThreadNode<Type> *p;
    for ( p=First( ); p!=NULL; p=Next( ))
        cout<< p->data<<endl;
}
```

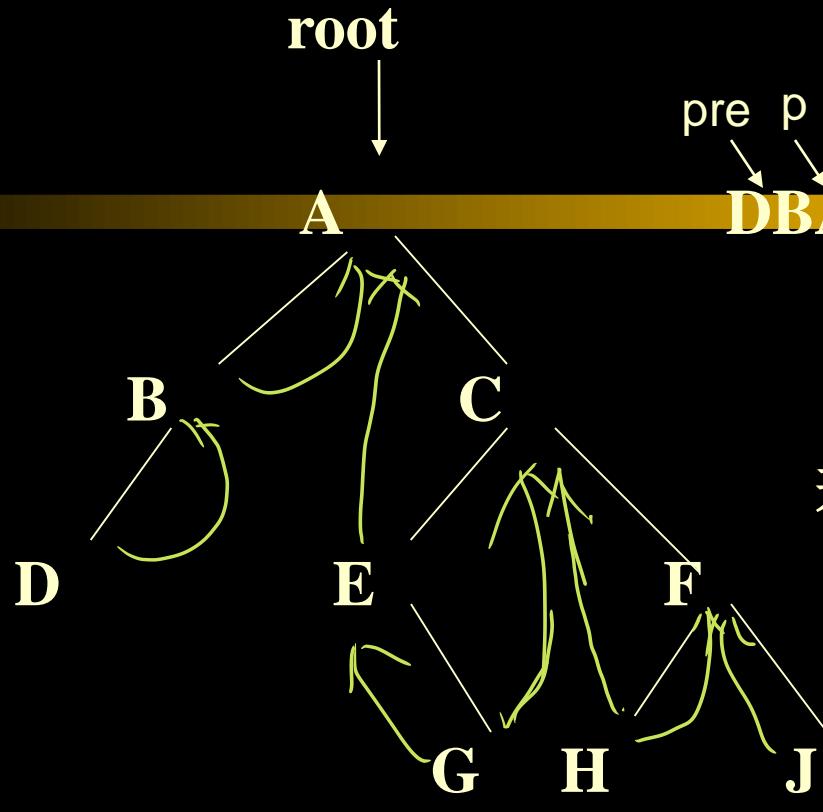
2) 构造中序线索树

对已存在的一棵二叉树建立中序线索树

分析：与中序遍历算法差不多，但是要填左空域右空域的前驱、后继指针。所以除了流动指针p外，还要加一个pre指针，它总是指向遍历指针p的中序下的前驱结点。

例如：

J
历前一个节点
历后一个节点



即 pre 为 p 的中序下的前驱，
若 pre 的左链为空，则可填 pre

即 p 为 pre 的中序下的后继
若 pre 的右链为空，则 可填 p

前中后序遍历

Thread Tree

Create inorder threadTree:

```
Void Inthread(threadNode<T> * T)
{ stack <threadNode <T> *> s (10)
  ThreadNode <T> * p = T ; ThreadNode <T> * pre = NULL; 线程是在成树的结构的基础上  
新增的代码
  for ( ; ; )
    { 1.while (p!=NULL)
      { s.push(p); p = p ->leftchild; }

      2.if (!s.IsEmpty( ))
        { 1) p = s.pop;
          2) if (pre != NULL)
            → if (pre ->rightchild == NULL)
              { pre ->rightchild = p; pre ->rightthread = 1; }
            → if ( p -> leftchild == NULL)
              { p -> leftchild = pre ; p ->leftthread = 1; }

            3) pre = p ; p = p -> rightchild ;
          }

        else return;
    } //for
}
```

4.8 Application

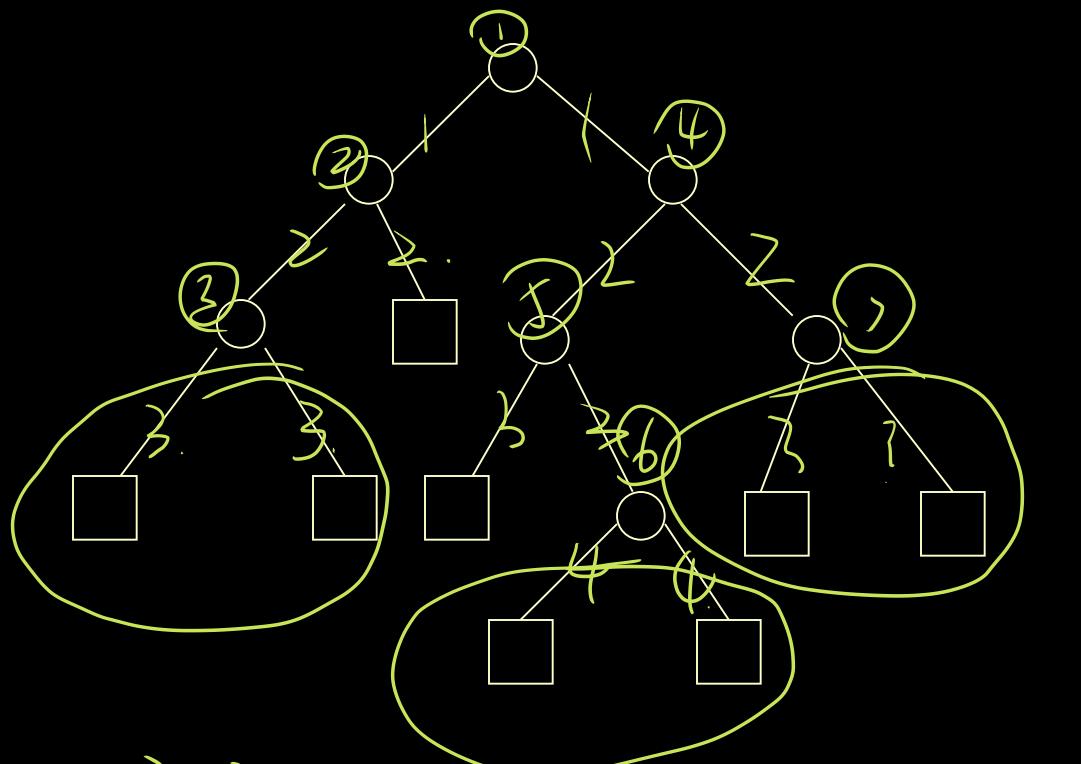
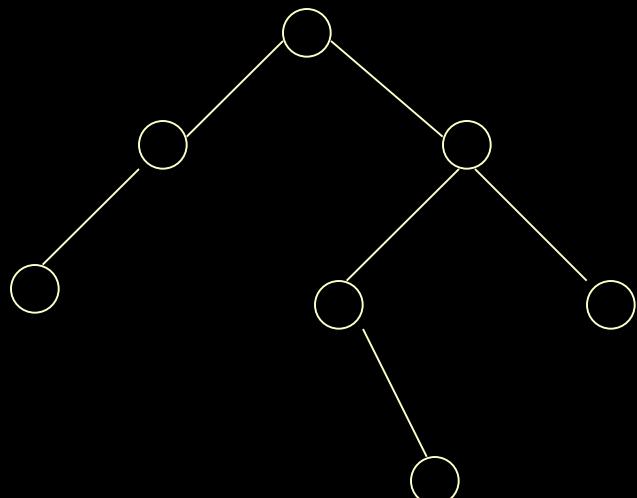
2. 霍夫曼树(Huffman Tree)

- ## • 增长树的概念

1) 增长树

对原二叉树中度为1的结点，增加一个空树叶

对原二叉树中的树叶，增加两个空树叶



3+3+2+3+4+4+3=23.

2) 外通路长度(外路径) E: 根到每个外结点(增长树的叶子)
的路径长度的总和(边数)

$$E = 3+3+2+3+4+4+3+3 = 25$$

不止是“头”，内部也算

2 1 1 2 3 2
(节点 - 叶)个

3) 内通路长度(内路径) I: 根到每个内结点(非叶子)的路径长度的总和(边数)。

$$I = 2+1+0+3+2+2+1 = 11$$

不仅路径长度变短 路经个数也减少了

4) 结点的带权路径长度: 一个结点的权值与结点的路径长度的乘积。

5) 带权的外路径长度: 各叶结点的带权路径长度之和。

6) 带权的内路径长度: 各非叶结点的带权路径长度之和。

• 霍夫曼树

1) 给出m个实数 w_1, w_2, \dots, w_m ($m \geq 2$) 作为m个外结点的权构造一棵增长树，使得带权外路径长度

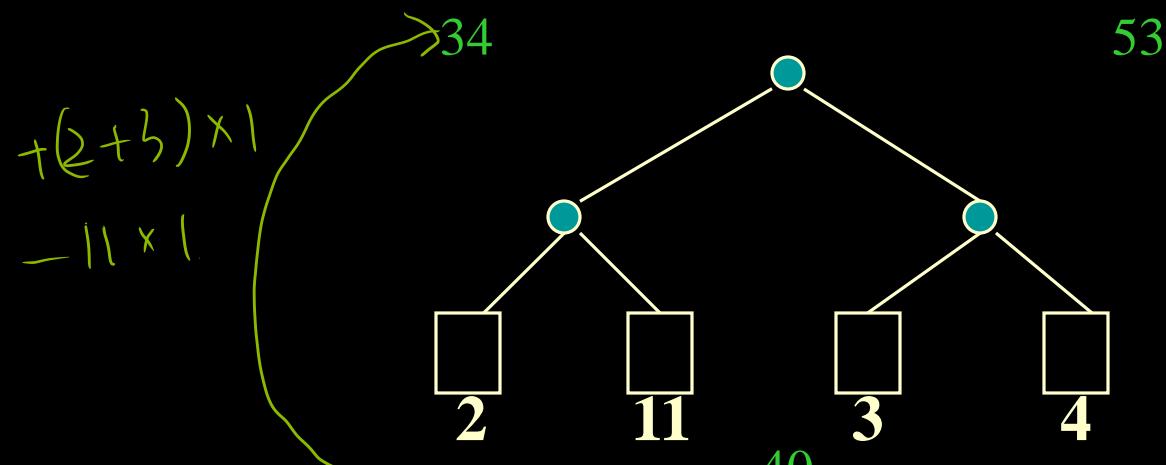
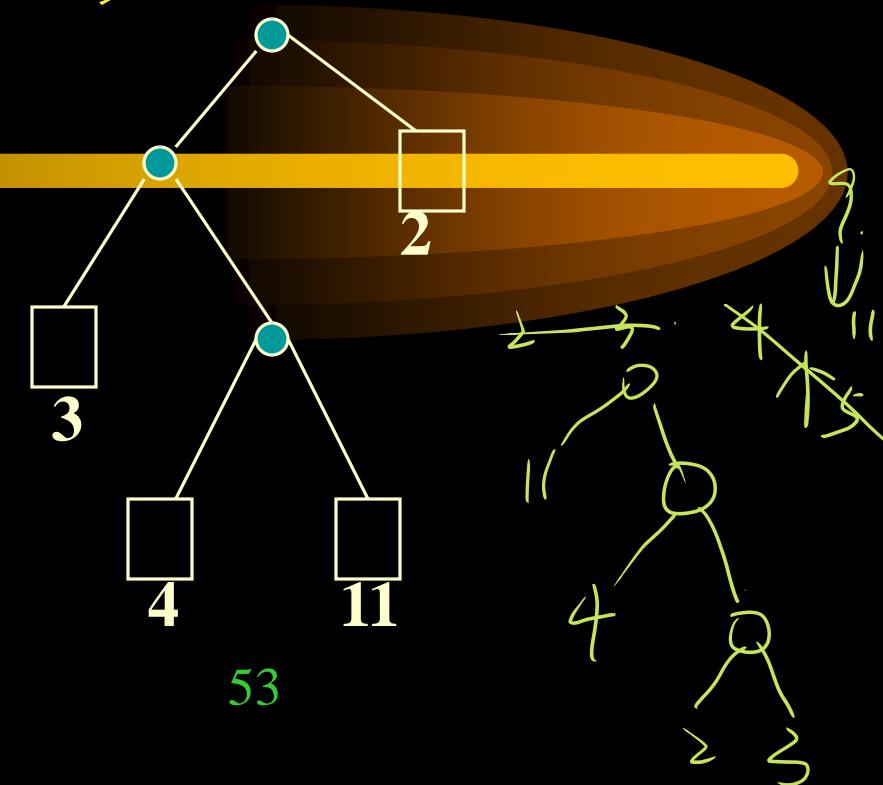
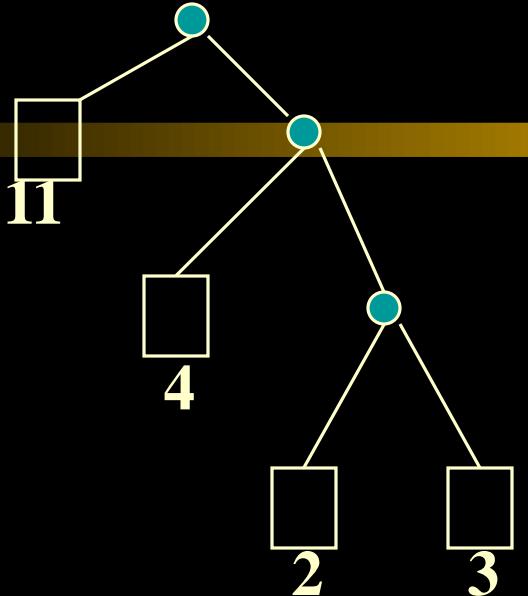
m

$\sum_i^m w_i l_i$ 最小。

其中 l_i 为从根结点出发到具有权为 w_i 的外结点的通路长。

2) 例子：外结点权为 2, 3, 4, 11 则可构造

$$1 \times 1 + 2 \times 4 + 3 \times 2 + 3 \times 3.$$

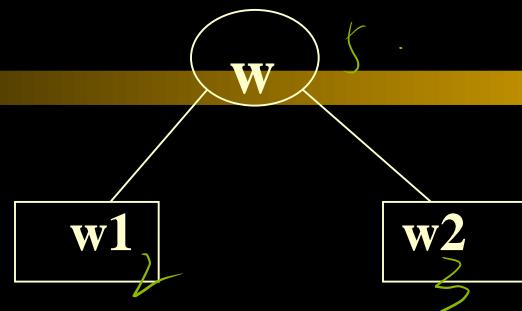


$$2 \times 2 + 2 \times 11 + 2 \times 3 + 2 \times 4.$$

3) Huffman 算法

思想： 权大的外结点靠近根， 权小的远离根。

算法： 从m个权值中找出两个最小值W1, W2构成



$W = W_1 + W_2$ 表示通过该结点的频度。

然后对m-1个权值W, W3, W4, ...Wm 经由小到大排序，求解这个问题。

例子： 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41
 ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
 5 10 15 20 25 30 35 40 45 50 55

注意：当内结点的权值与外结点的权值相等的情况下，内结点应排在外结点之后。除了保证 $\sum W_i l_i$ 最小外，还保证 $\max I_j, \sum I_j$ 也有最小值

例如： 7, 8, 9, 15

- 霍夫曼编码

是霍夫曼树在数据编码中一种应用。具体的讲用于通信的二进制编码中。设一电文出现的字符为 $D=\{M, S, T, A, Q, K\}$ ，每个字符出现的频率为 $W=\{10, 29, 4, 8, 15, 7\}$ ，如何对上面的诸字符进行二进制编码，使得

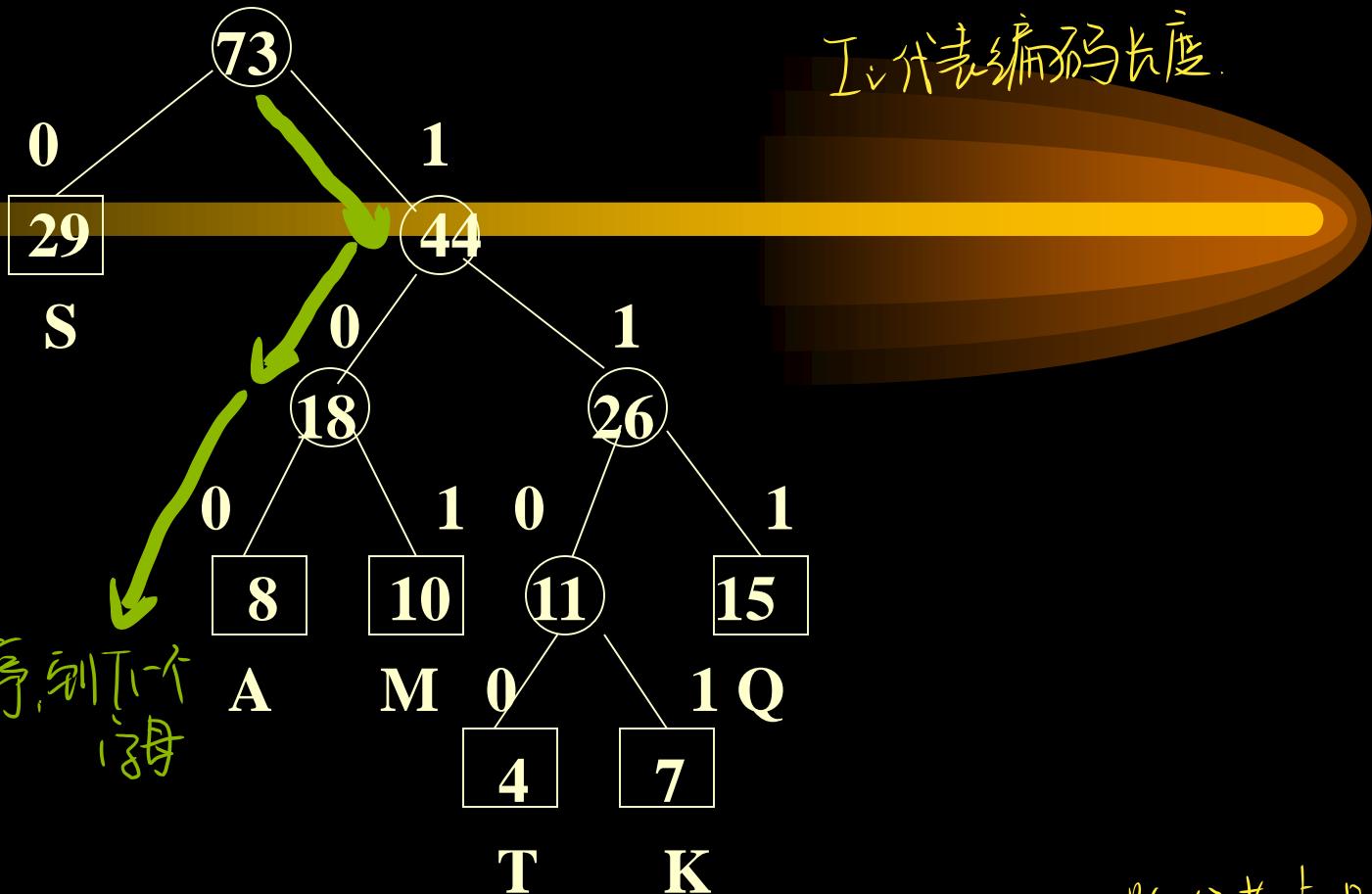
- 1) 该电文的总长度最短。
- 2) 为了译码，任一字符的编码不应是另一字符的编码的前缀

算法：利用Huffman算法，把 $\{10, 29, 4, 8, 15, 7\}$ 作为外部结点的权，构造具有最小带权外路径长度的扩充二叉树，把每个结点的左子女的边标上0，右子女标上1。这样从根到每个叶子的路径上的号码连接起来，就是外结点的字符编码。

左0右1

W_i 代表频率

T_i 代表编码码长



编码: S: 0 A: 100 M: 101 T: 1100 K: 1101 Q: 111

路径节点号
→ 编码

已知二进制编码，请译码。

方法是从根结点开始确定一条到达叶结点的路径。

| | | | | | | | | |
|-----|------|-----|---|-----|---|------|---|---|
| 100 | 1100 | 101 | 0 | 111 | 0 | 1101 | 0 | 0 |
| A | T | M | S | Q | S | K | S | S |

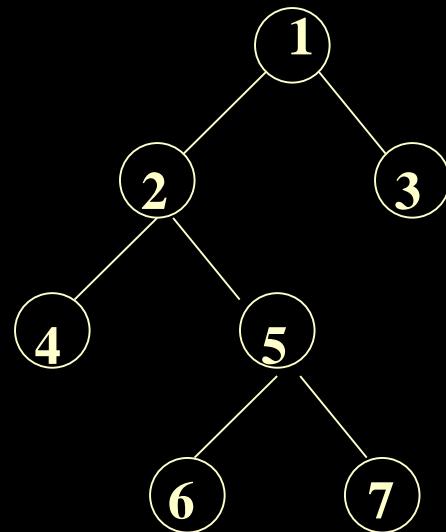
Chapter 4

2009年统考题(单项选择):

3. 给定二叉树如下图所示. 设N 代表二叉树的根, L 代表二叉树的左子树, R 代表根结点的右子树. 若遍历后的结点序列为 3, 1, 7, 5, 6, 2, 4, 则其遍历方式是

- A. LRN
- B. NRL
- C. RLN
- D. RNL

D.



Chapter 4

2009年统考题(单项选择):

- A. ~~已知一棵完全二叉树的第6层(设根为第1层)有8个叶结点, 则该完全二叉树的结点个数最多是~~
- C. ~~8个结点是叶结点~~
- A. 39 B. 52 C. 111 D. 119

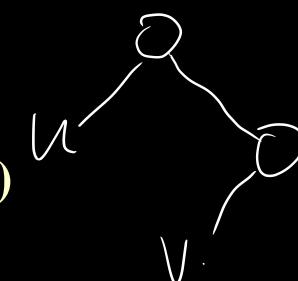
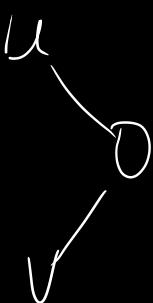
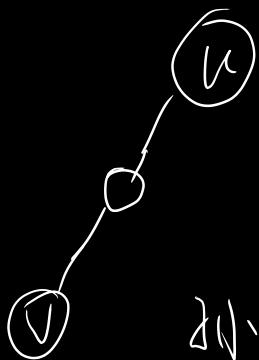
- B. 将森林转换为对应的二叉树, 若在二叉树中, 结点u是结点v的父结点的父结点, 则在原来的森林中, u 和 v 可能具有的关系是

- 1) 父子关系
2) 兄弟关系
3) u 的父结点与v 的父结点是兄弟关系

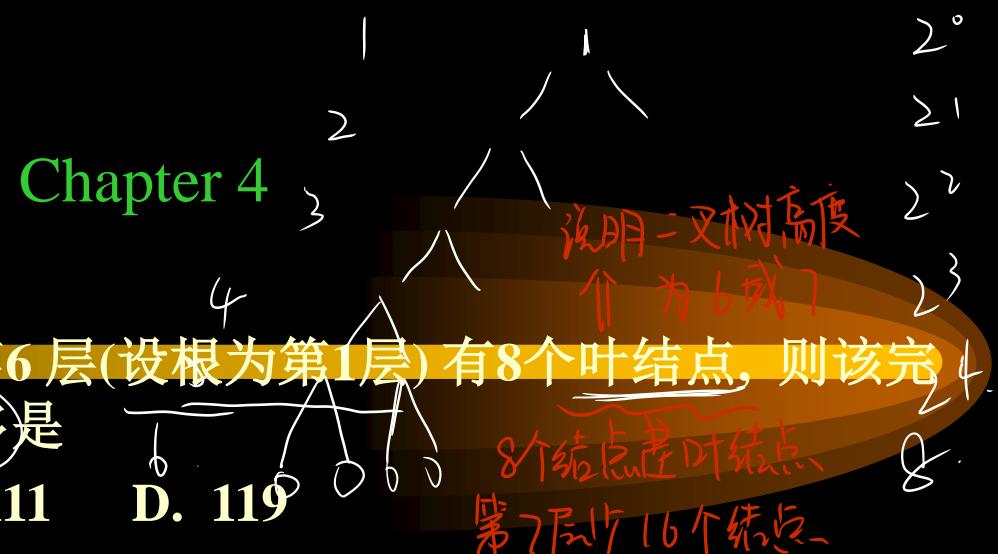
物理层

逻辑层

- A. 只有2)
B. 1)和2)
C. 1)和3)
D. 1), 2)和3)



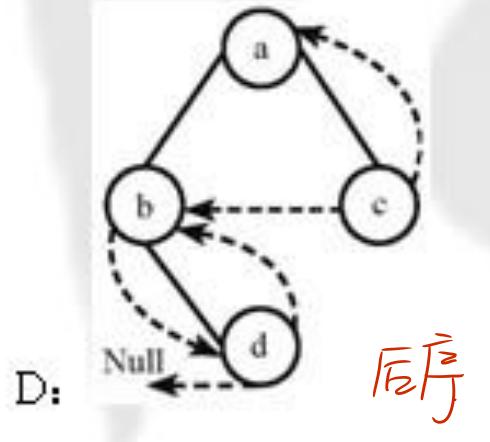
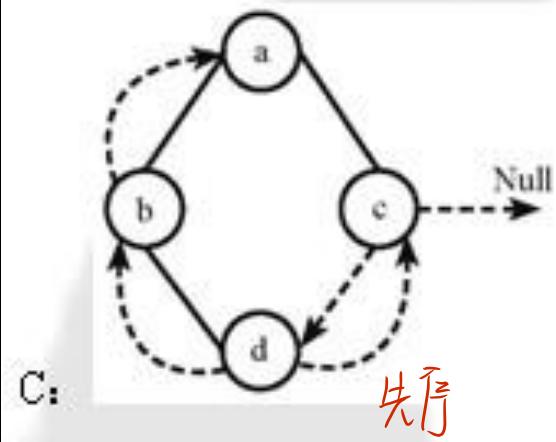
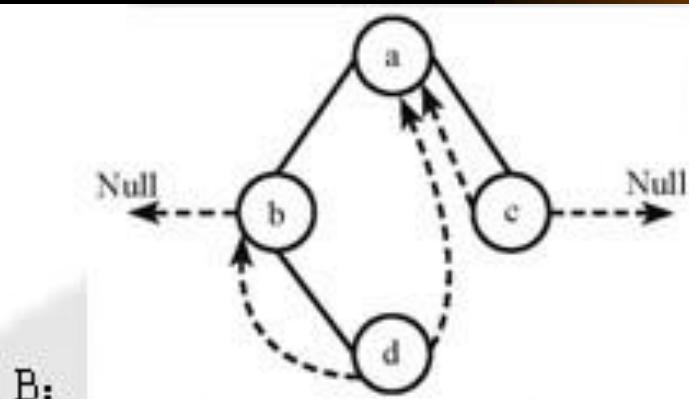
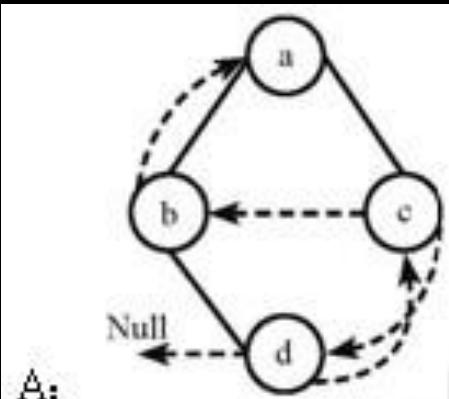
小



Chapter 4

2010年全国考研题

3、下列线索二叉树中（用虚线表示线索），符合后序线索树定义的是（ ）



Chapter 4

10.

$$20 \times 4 + 10 \times 3 + (1 \times 2 + 10 \times 1) = 122$$

- 5、在一棵度为4的树T中，若有20个度为4的结点，10个度为3的结点，1个度为2的结点，10个度为1的结点，则树T的叶节点个数是 **B) 123** ~~123 - 20 - 10 - 1~~

A: 41 B: 82 C: 113 D: 122

节点: 123

- 6、对n(n大于等于2)个权值均不相同的字符构成哈夫曼树，关于该树的叙述中，错误的是()

- A: 该树一定是一棵完全二叉树 **X**
- B: 树中一定没有度为1的结点
- C: 树中两个权值最小的结点一定是兄弟结点
- D: 树中任一非叶结点的权值一定不小于下一任一结点的权值

Chapter 4

1. 给出如下各表达式的二叉树：

1) $(a+b)/(c-d \cdot e)+e+g \cdot h/a$

2) $-x-y \cdot z+(a+b+c/d \cdot e)_x$

3) $((a+b)>(c-d)) \parallel a < f \&\& (x < y \parallel y > z)$

2. 如果一棵树有 n_1 个度为 1 的结点，有 n_2 个度为 2 的结点，……， n_m 个度为 m 的结点，试问有多少个度为 0 的结点？写出推导过程。

3. 分别找出满足以下条件的所有二叉树：

1) 二叉树的前序序列与中序序列相同

2) 二叉树的中序序列与后序序列相同

3) 二叉树的前序序列与后序序列相同

4. 若用二叉链表作为二叉树的存储表示，试对以下问题编写递归算法。

1) 统计二叉树中叶结点的个数。

2) 以二叉树为参数，交换每个结点的左子女和右子女

Chapter 4

5. 已知一棵二叉树的先序遍历结果是 ABECDFGHIJ,

中序遍历结果是 EBCDAFHIGJ

试画出这棵二叉树。

6. 编写一个Java函数，输入后缀表达式，构造其二叉树表示。设每个操作符有一个或两个操作数。

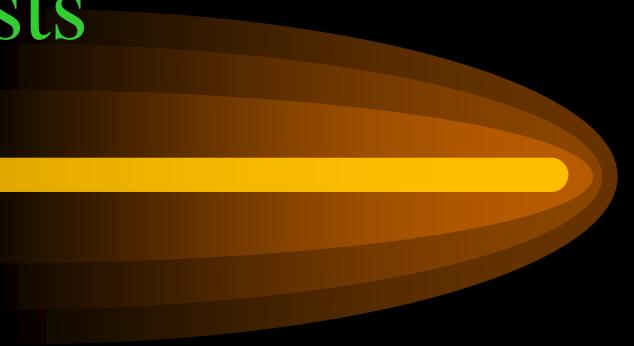
7. 给定权值{ 15, 03, 14, 02, 06, 09, 16, 17 }, 构造相应的霍夫曼树，并计算它的带权外路径长度。 2 3 6 9 14 15 16 17

8. c1, c2, c3, c4, c5, c6, c7, e8 这八个字母的出现频率分别 { 5, 25, 3, 6, 10, 11, 36, 4, } 为这八个字母设计不等长的Huffman编码，并给出该电文的总码数。 3 25 6 10 11 4 36 18

实习题：

6. 建立一棵二叉树，并输出前序、中序、后序遍历结果。

** General Lists

- 
1. 广义表的概念
 2. 广义表的性质
 3. 广义表的操作
 4. 广义表的存储结构
 5. 广义表的类声明
 6. 输入二叉树的广义表表示来建立一棵树

** General Lists

1. 广义表的概念(LS)

* 定义为 $n(n \geq 0)$ 个表元素 $a_0, a_1, a_2, \dots, a_{n-1}$ 组成的有限序列, 记作:

$$LS = (a_0, a_1, a_2, \dots, a_{n-1})$$

其中每个表元素 a_i 可以是原子, 也可以是子表.

原子: 某种类型的对象, 在结构上不可分(用小写字母表示).

子表: 有结构的 .(用大写字母表示)

example:

$$L = (3, (), ('b', 'c'), (((d))))$$

* 广义表的长度: 表中元素的个数

** General Lists

*广义表的表头(head),表尾(tail)

head= a_0 ;

tail=(a_1, a_2, \dots, a_{n-1})

*广义表的深度：表中所含括号的最大层数

1) A=();

2) B=(6, 2)

3) C=(‘a’, (5, 3, ‘x’)) 表头为 ‘a’, 表尾为((5, 3, ‘x’))

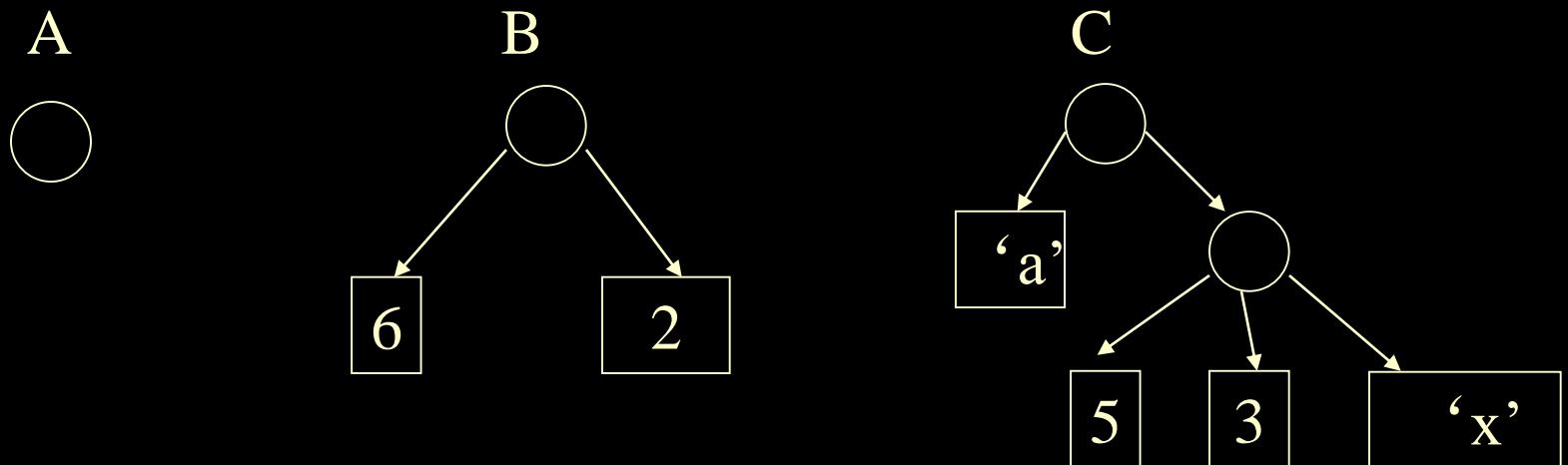
4) D=(B, C, A)

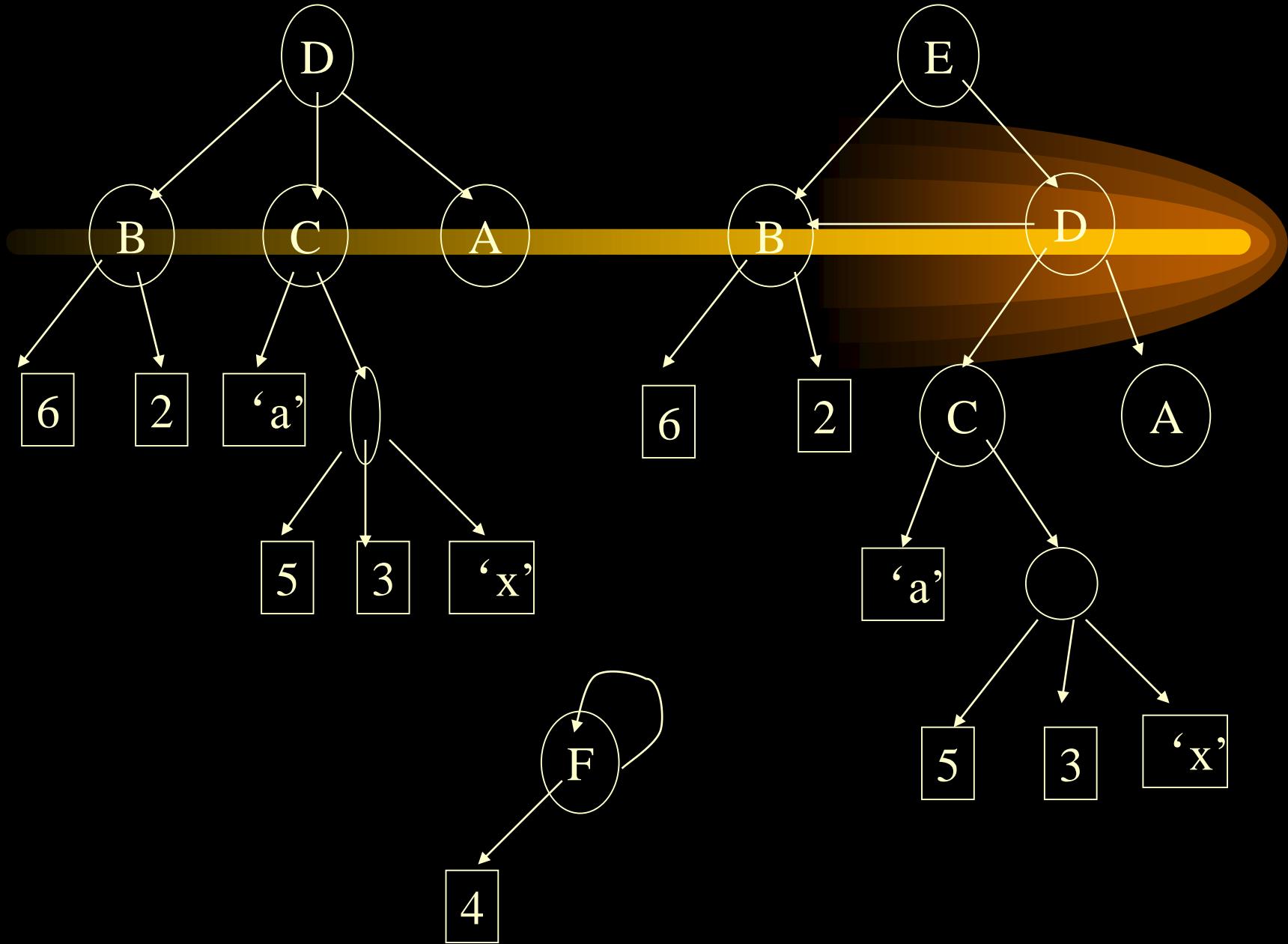
5) E=(B, D)

6) F=(4, F) 递归的表

2. 广义表的性质(特点)

- 1)有序性
- 2)有长度,有深度
- 3)可递归,如上面例6
- 4)可共享,如E中B为E, D所共享
- 5)各种广义表都可用一种示意图来表示,用 ○ 表示表元素,用 [] 表示原子





** General Lists

3. General Lists operate:

- | | |
|---------------------------|--------------------|
| 1) head (list) | 2) tail (list) |
| 3) first (list) | 4) info (elem) |
| 5) next (elem) | 6) nodetype (elem) |
| 7) push (list,x) | 8) addon (list,x) |
| 9) setinfo (elem,x) | |
| 10) sethead (list,x) | |
| 11) setnext (elem1,elem2) | |
| 12) settail (list1,list2) | |

** General Lists

4. General Lists representation:

| | | |
|-------|-------|-------|
| utype | value | tlink |
|-------|-------|-------|

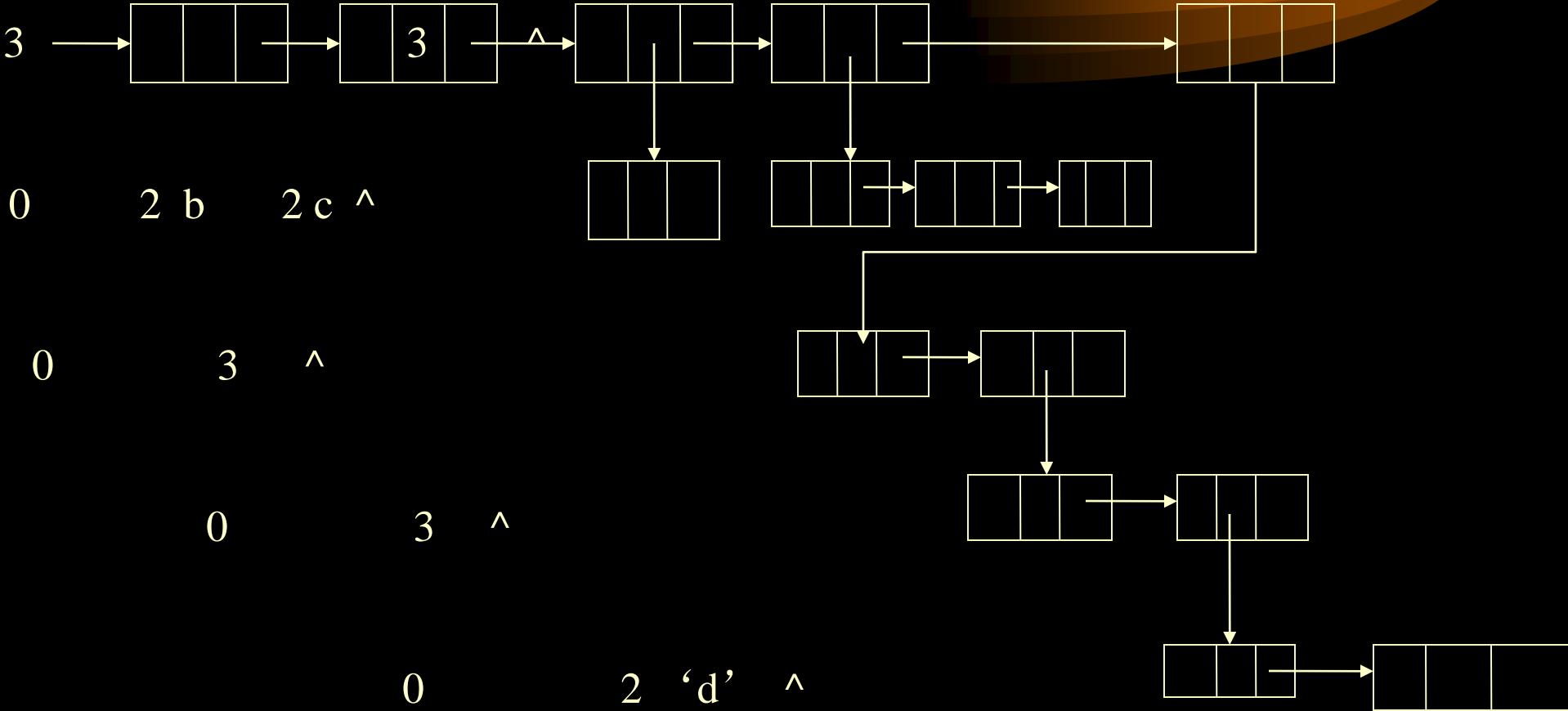
utype=0 ref

utype=1 intgrinfo

utype=2 charinfo

utype=3 hlink

((‘d’))))



** General Lists

5. 广义表的类声明

```
#define HEAD 0
```

```
#define INTGR 1
```

```
#define CH 2
```

```
#define LST 3
```

```
class GenList;
```

```
class GenListNode
```

```
{ friend class GenList;
```

```
private:
```

```
    int utype;
```

```
    GenListNode * tlink;
```

```
    union { int ref;
```

```
        int intgrinfo;
```

```
        char charinfo;
```

```
        GenListNode * hlink;
```

```
    } value;
```

** General Lists

public:

```
    GenListNode & info (GenListNode * elem);  
    int nodetype (GenListNode * elem) {return elem->utype;}  
    void setinfo (GenListNode * elem,GenListNode & x);  
};
```

class GenList

{ **private:**

```
    GenListNode * first;  
    GenListNode * Copy (GenListNode * ls);  
    int depth (GenListnode * ls);  
    int equal (GenlistNode * s, Genlistnode * t);  
    void Remove (GenlistNode * ls);
```

public:

```
    GenList ( );
```

** General Lists

```
~GenList ( );
GenListNode & Head ( );
GenListNode * Tail ( );
GenlistNode * First ( );
GenlistNode * Next (GenListNode * elem);
void Push (GenListNode & x);
GenList & Addon ( GenList & list, GenListNode & x);
void setHead (GenListNode & x);
void setNext (GenlistNode * elem1, GenlistNode * elem2);
void setTail(GenList & list);
void Copy (const GenList & l);
int depth ( );
int Createlist (GenListNode * ls, char * s);
}
```

6. 广义表的递归算法

递归算法:1)递归函数的外部调用-----公有函数 界面

2)递归函数的内部调用-----私有函数 真正实现部分

1)求广义表的深度

广义表的深度为广义表中最大括号的重数

广义表 $LS=(a_0, a_1, a_2, \dots, a_{n-1})$, 其中 $a_i (0 \leq i \leq n-1)$ 或者是原子或者子表.

$$\text{Depth}(LS)=\begin{cases} 0 & \text{当 } LS \text{ 为原子时} \\ 1 & \text{当 } LS \text{ 为空表时} \\ 1+\max\{\text{depth}(a_0), \text{depth}(a_1), \dots, \text{depth}(a_{n-1})\} & \end{cases}$$

递归结束条件

公共函数:

```
int GenList::depth()  
{  return depth(first);  
}
```

私有函数:

```
int GenList::depth(GenListNode*ls)  
{  if( ls-->tlink==NULL)  return 1;  
  GenListNode*temp=ls-->tlink; int m=0;  
  while( temp!=NULL)  
  {    if( temp-->utype==LST)  
      {      int n=depth(temp-->value.hlink);  
          if(m<n)m=n;  
      }  
    temp=temp-->tlink;  
  }  
  return m+1;  }
```

2) 判断两个广义表相等否

相等的条件: 具有相同的结构

对应的数据元素具有相等的值

if(两个广义表都为空表)return相等

else if(都为原子^值相等)递归比较同一层的后面的表元素

else return 不相等.

int operator== (const GenList&l,const GenList&m)//假设是友元

```
{ return equal(l.first, m.first); }
```

```
int equal( GenListNode*s, GenListNode*t)//假设是友元
```

```
{ int x;
```

```
if(s-->tlink==NULL&&t-->tlink==NULL) return 1;
```

```
if((s-->tlink!=NULL&&t-->tlink!=NULL&&s-->tlink-->utype==t-->tlink-->utype)
```

```
{ if(s-->tlink-->utype==INTGR)
```

```
    if(s-->tlink-->value.intgrinfo== t-->tlink-->value.intgrinfo) x=1;
```

```
    else x=0;
```

```
else if(s-->tlink-->utype==CH)
```

```
    if(s-->tlink-->value.charinfo==t-->tlink-->value.charinfo)x=1;
```

```
    else x=0;
```

```
else x=equal(s-->tlink-->value.hlink, t-->tlink-->value.hlink);
```

```
if(x) return equal(s-->tlink, t-->tlink);
```

```
}
```

```
return 0;
```

3) 广义表的复制算法

分别复制表头,表尾,然后合成

前提是广义表不可以是共享表或递归表

公共函数:

```
void GenList::copy(const GenList&l)  
{first=copy(l.first);  
}
```

私有函数:

```
GenListNode*GenList::copy(GenListNode*ls)  
{  
GenListNode*q=NULL;  
if(ls!=NULL)  
{q=new GenListNode;  
q-->utype=ls-->utype;
```

```
Switch(ls-->utype)
```

```
{  case HEAD: q-->value.ref=ls-->value.ref; break;//表头结点  
    case INTGR: q-->value.intgrinfo=ls-->value.intgrinfo; break;  
    case CH: q-->value.charinfo=ls-->value.charinfo; break;  
    case LST: q-->value.hlink=ls-->value.hlink; break;  
}  
  
q-->tlink=copy(ls-->tlink);  
  
}  
  
return q;  
}
```

4) 广义表的析构函数----- ~GenList()

公共函数:

```
GenList::~GenList( )
```

```
{ remove(first); }
```

私有函数:

```
void GenList::remove(GenListNode*ls)
```

```
{ ls->value.ref--;
```

```
if (!ls->value.ref)
```

```
{ 1) GenListNode*y=ls;
```

```
2) while(y-->tlink!=NULL)
```

```
{ y=y-->tlink;
```

```
if(y-->utype==LST)remove(y-->value.hlink);
```

```
}
```

```
3) y-->tlink=av; av=ls; //回收顶点到可利用空间表中
```

```
}
```

```
}
```

2005. 六. 1 广义表

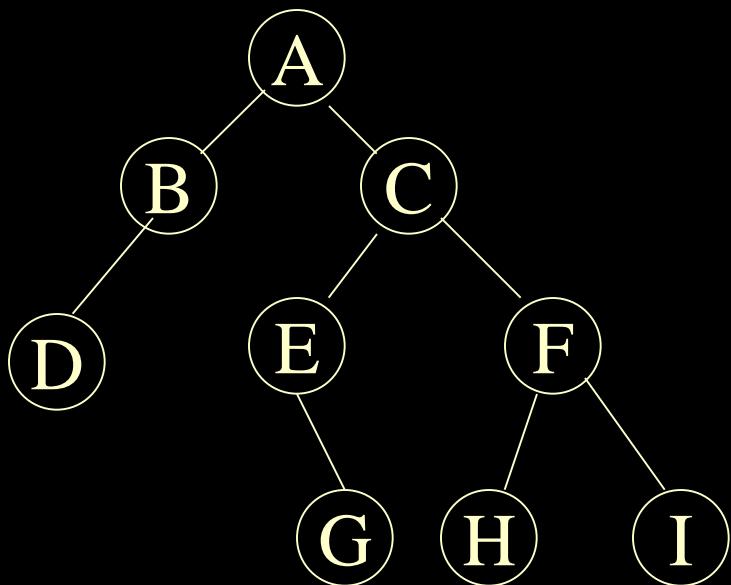
对广义表的回收算法如下, 请回答在回收广义表ls后, 在可利用空间表av中原广义表中的结点是按何种次序链接起来的(请用原广义表结点上方的字母组成一个序列来表示链接的顺序: 序列的头部是新的av所指向的地方, 序列的尾部连接原来av的空间).

```
GenList :: ~GenList( )
{
    Remove(first);
}

Void GenList :: Remove(GenListNode *ls)
{
    ls->value.ref--;
    if ( !ls -> value.ref)
    {
        GenListNode * y = ls;
        while ( y ->tlink != NULL)
        {
            y = y ->tlink;
            if ( y -> utype == LST) Remove( y ->value.hlink);
        }
        y -> tlink = av; av = ls;
    }
}
```

7. Create BinaryTree another method:

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



Create Binary Tree another method:

```
void CreateBTree(BTreeNode * & BT, char *a)
```

```
{ BTreeNode * s[10];
```

```
int top = -1;
```

```
BT = NULL;
```

```
BTreeNode * p;
```

```
int k;
```

```
istrstream ins (a);
```

```
char ch;
```

```
ins >> ch;
```

Create Binary Tree another method:

```
while( ch != '@')  
{  switch (ch)  
    { case '(': top++; s[top ]= p; k = 1; break;  
     case ')':top--; break;  
     case ',': k = 2; break;  
     default:  
        p = new BTreeNode;  
        p->data = ch; p->left = p->right = NULL;  
        if(BT == NULL) BT = p ;  
        else { if (k == 1) s[top]->left = p;  
               else s[top]->right = p;  
               }  
    } //switch end  
    ins >>ch;  
} //while end  
}
```