第五章 树和二叉树

内容提要：

前面学习了线性数据结构，从本章开始学习非线性数据结构。本章学习一种常见的非线性结构—树结构。

- 树数据结构的特点
- 二叉树数据结构
  - 二叉树数据结构的特性
  - 二叉树ADT的定义
  - 二叉树ADT的实现
    - （1）顺序存储
    - （2）链式存储

5.1 树

5.1.1 树ADT的定义

1. 树逻辑结构

[树 Tree]:是一个非空的有限元素的集合，元素之间具有如下关系：有且仅有一个特殊元素，它没有前驱 (称为树根Root)，其余元素都有且仅有一个前驱，所有元素可以有零个或多个后继。

树的递归定义：

树T是一个非空数据元素的有限集合，其中有且仅有一个特定元素称为树T的根，剩余的元素（若有的话）可被划分为 m 个互不相交的集合 T_1，T_2，...，T_m，而每个集合又都是树，称为T的子树(Subtree)。

5.1 树

5.1.1 树ADT的定义

1. 树逻辑结构

特点：除根外，每个元素有且仅有一个前驱，有零个或多个后继。

设树T的根为root，子树$T_1,T_2,\ldots T_m$的根分别为 $r_1,r_2,\ldots,r_m$ ，则root是$r_1,r_2,\ldots,r_m$的前驱，$r_1,r_2,\ldots r_m$是root的后继。

<root, r_1>

<root,r_2>

……

<root,r_m>

5.1 树

5.1.1 树ADT的定义

1. 树逻辑结构

树数据结构的形式化定义：

Tree=(D,R)

$D=\{a_i \mid i=1,2,...,n\ n\geq 1 \quad a_i \in D_0\}$

$=\{ root \} \cup D_f \quad root \in D_0 \quad D_f=\bigcup_{i=1}^{m} D_i \quad D_i \cap D_j=\varnothing$

$$R=\begin{cases} < root, r_i > \mid T_i=(D_i, R_i), D_i=\{r_i\} \cup D_{fi}, 1\leq i\leq m & m\geq 1 \\ \varnothing & m=0 \end{cases}$$

5.1 树

5.1.1 树ADT的定义

1. 树逻辑结构

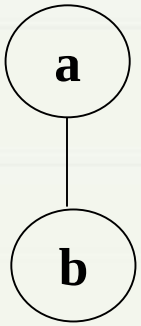
举例：

T_1=(D,R)
D={a}
R= ∅

T_1={a}

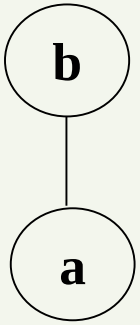
T_2=(D,R)
D={a,b}
R={<a,b>}

T_2={a,{b}}



T_3=(D,R)
D={a,b}
R={<b,a>}

T_3={b,{a}}



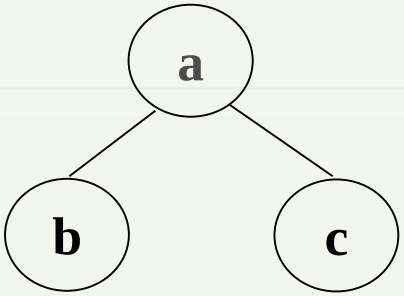
5.1 树

5.1.1 树ADT的定义

1. 树逻辑结构

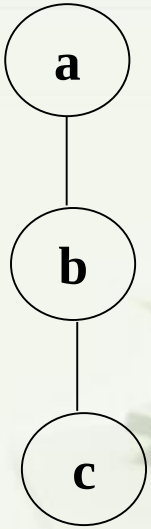
T_4=(D,R)
D={a,b,c}
R={<a,b>,<a,c>}

T_4={a,{b},{c}}



T_5=(D,R)
D={a,b,c}
R={<a,b>,<b,c>}

T_5={a,{b,{c} } }



5.1 树

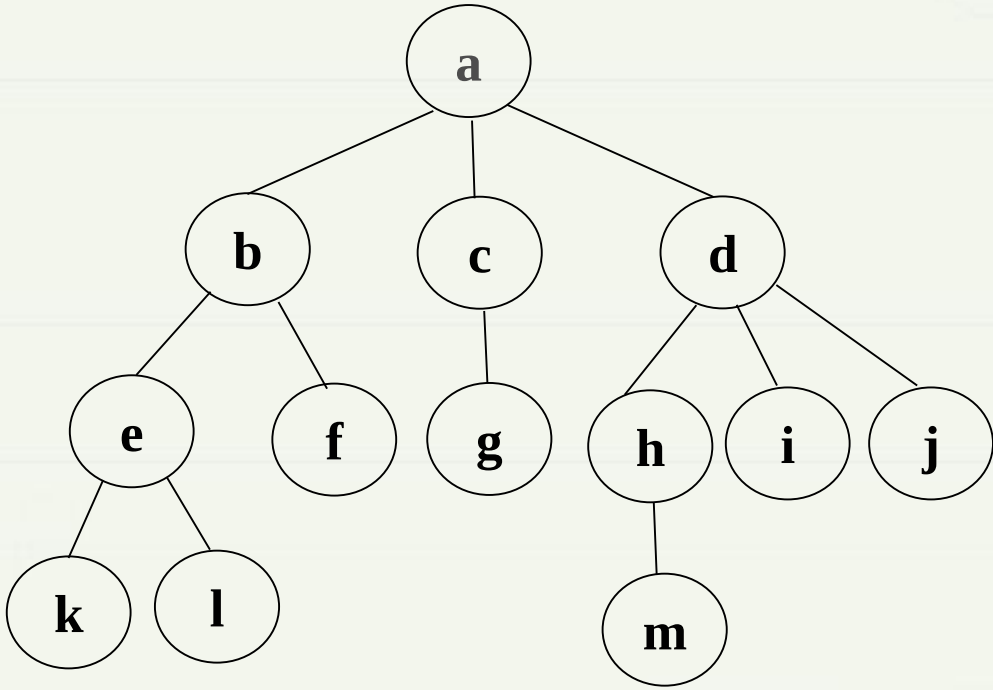
5.1.1 树ADT的定义

1. 树逻辑结构

T_6=(D,R)

D={a,b,c,d,e,f,g,h,i,j,k,l,m}

R={<a,b>,<a,c>,<a,d>,
<b,e>,<b,f>,
<c,g>
<d,h>,<d,i>,<d,j>,
<e,k>,<e,l>,
<h,m> }



T_6={a,
{b, {e,{k},{l}}, {f} },
{c,{g} },
{d,{h,{m}},{i},{j} }
}

5.1 树

5.1.1 树ADT的定义

1. 树逻辑结构

树数据结构的术语：

结点	树中的一个数据元素
结点的度(degree)	任一结点子树的数目，记作 d(v)
叶子(leaf)	度为0的结点，又称为末端结点。
分支结点	度不为0的结点，又称为非末端结点。
分支	结点之间的关系。
内部结点	除根之外的分支结点。
树的度	树中结点度的最大值。K叉树

5.1 树

5.1.1 树ADT的定义

1. 树逻辑结构

树数据结构的术语（续）：

孩子（儿子）	结点的子树的根称为该结点的孩子（后继）。
双亲（父亲）	一个结点是它各子树的根结点的双亲（前趋）。
兄弟(sibling)	具有相同双亲的结点。
祖先	从根结点到该结点路径上所有结点都称为该结点的祖先
子孙	该结点所有子树上的结点都称为该结点的子孙
结点的层次：	根结点定义为第1层，根的儿子定义为第2层，…，依次类推。记作 L(v)

5.1 树

5.1.1 树ADT的定义

1. 树逻辑结构

树数据结构的术语（续）：

树的深度（高度）	各个结点层次的最大值。
堂兄弟	双亲在同一层上的结点。
有序树	结点的子树（后继）是有顺序的（兄弟有大小、先后之分）。
路径	树中的k个结点$n_1,n_2,\ldots,n_k$，满足 n_i是 n_{i+1}的双亲，称n_1到n_k有一条路径。
路径长度	分支数=路径上结点个数-1

5.1 树

5.1.1 树ADT的定义

1. 树逻辑结构

- 根没有双亲，叶子没有孩子；
- v_i是v_j的双亲，则 $L(v_i)=L(v_j)-1$；

注意：

- 堂兄弟的双亲是兄弟关系吗？
- 有序树和无序树的区别；
- 树有n个数据元素，则它有多少分支？

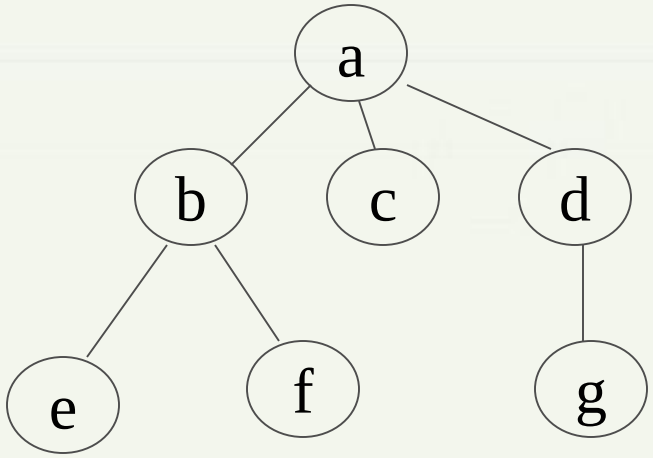
5.1 树

5.1.1 树ADT的定义

2. 树逻辑结构表示方法（树的画法）

（1）直观表示法：

用圆圈表示结点，元素写在圆圈中，连线表示元素之间的关系根在上，叶子在下（即树向下生长）。



5.1 树

5.1.1 树ADT的定义

2. 树逻辑结构表示方法（树的画法）

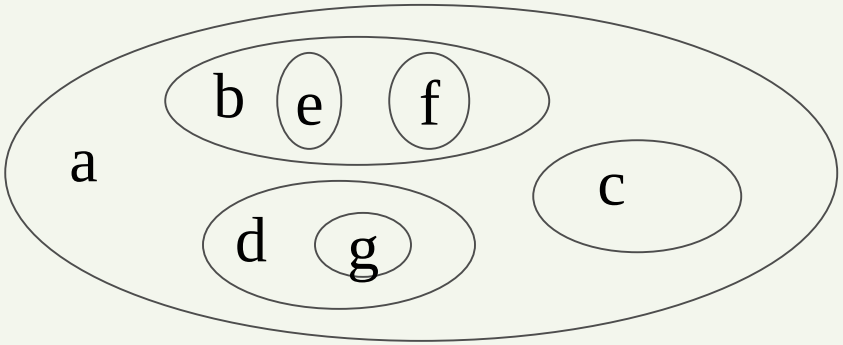
（2）集合表示法：

根据树的集合定义，写出集合划分。

{ a, **{b,{e},{f}},** {c}, **{d,{g}}** }

（3）文氏图表示法：

集合表示的一种直观表示，用图表示集合。



5.1 树

5.1.1 树ADT的定义

2. 树逻辑结构表示方法（树的画法）

（4）目录表示法（凹入表示法）：

将一棵树描述为一本书，书-章-节-小节-

1a

2b

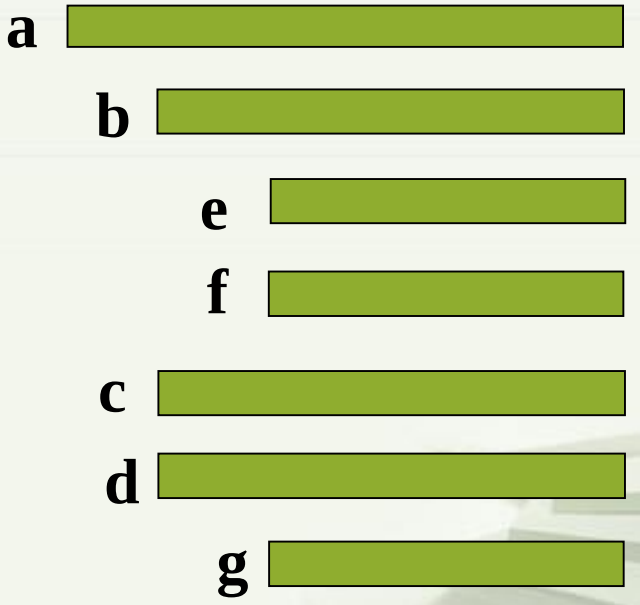
3e

3f

2c

2d

3g



5.1 树

5.1.1 树ADT的定义

2. 树逻辑结构表示方法（树的画法）

（5）广义表表示法：

将一棵树描述为一个广义表，树根为单元素，子树就对应子表。

(a, (b,(e),(f)), (c), (d,(g)))

人们最常用的是第1种，
但不适合计算机！

5.1 树

5.1.1 树ADT的定义

3. 树逻辑结构的性质

性质1： 树中的结点个数等于所有结点的度加1。

性质2： 度为k的树（k叉树）中第i层上最多有k^{i-1}个结点。（$i \geq 1$）

性质3： 深度为h的 k叉树最多有 $\frac{k^h-1}{k-1}$ 个结点。

性质4： 有n个结点的k叉树的最小深度为$\lceil \log_k(n(k-1)+1) \rceil$。

具有n个结点的k叉树的最大深度是多少？

5.1 树

5.1.1 树ADT的定义

4. 树逻辑结构定义的操作

（1）初始化Create(t)：创建空树

（2）求树的根结点 Root(t)

（3）求某结点的双亲 Parent(t,x)

（4）求某结点的最左孩子 Left_Child(t,x)

（5）求某结点的右兄弟 Right_Sibling(t,x)

（6）遍历 Traver(t)：将树中结点访问而且仅访问一遍。

（7）求深度 Depth(t)

（8）插入

（9）删除

最重要的操作

5.1 树

5.1.1 树ADT的定义

5. 树ADT 请同学们自己写出！

树的抽象类定义：

```
class Tree
{  public:
   Tree ();
   ~Tree ();
   BuildRoot (const ElemType& value);
   position FirstChild(position p);
   position NextSibling(position p);
   position Parent(position p);
   ElemType getData(position p);
   bool InsertChild(position p, ElemType& value);
   bool DeleteChild (position p, int i);
   void DeleteSubTree (position t);
   bool IsEmpty ();
   void Traversal (void (*visit)(position p));
};
```

5.1 树

5.1.2 树ADT的实现

1. 树的存储结构

存储元素，表示关系。可以顺序存储，也可以链式存储。
存储元素简单，表示关系困难！

■ 顺序存储

分配连续空间，依次存放树的各个元素。如何依次存放？

可约定：从上至下、从左至右将树的结点依次存入连续内存空间。

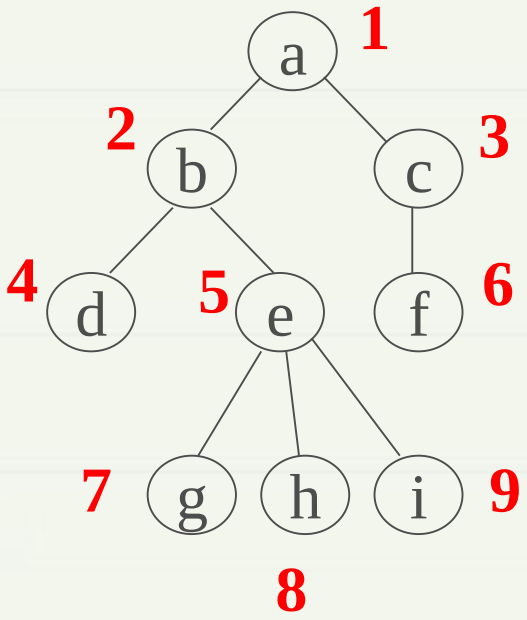
优点：简单
缺点：不能很好地表示出关系！

5.1 树

5.1.2 树ADT的实现

1. 树的存储结构

- 顺序存储



1	2	3	4	5	6	7	8	9			
a	b	c	d	e	f	g	h	i			

5.1 树

5.1.2 树ADT的实现

1. 树的存储结构

■ 链式存储

用任意的存储空间，存放数据元素，在存储元素的同时存放其相关元素的地址。

数据元素	前驱指针	后继指针$_1$	后继指针$_2$	……	后继指针$_k$

存储一个元素的空间

问题：需要开辟几个指针？

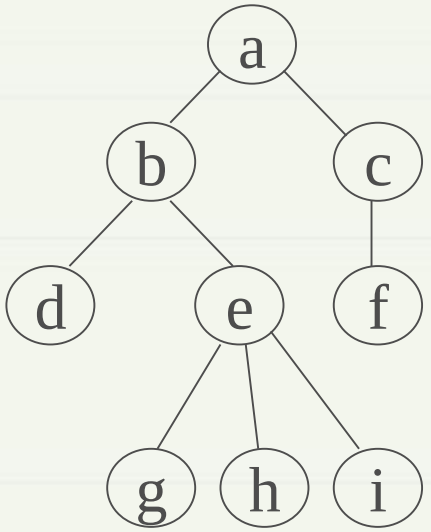
定长结点
不定长结点

5.1 树

5.1.2 树ADT的实现

1. 树的存储结构

- 链式存储



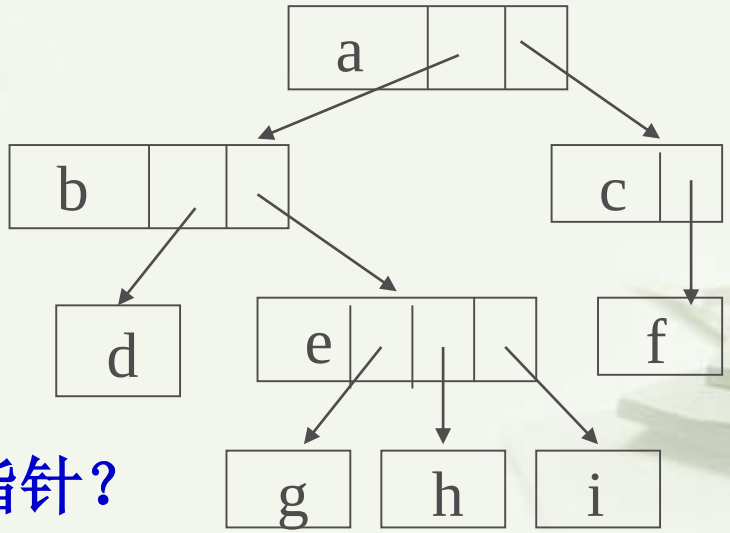
不定长结点：

data	c_1	c_2	… …	c_{di}

data： 数据元素

$c_1, c_2, \ldots, c_{di}$：指针，指向孩子结点

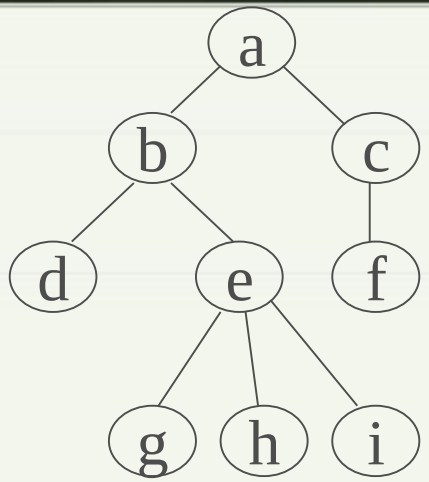
d_i：结点的度



存储有n个结点的树，有多少个指针？

5.1 树

5.1.2 树ADT的实现



1. 树的存储结构

- 链式存储

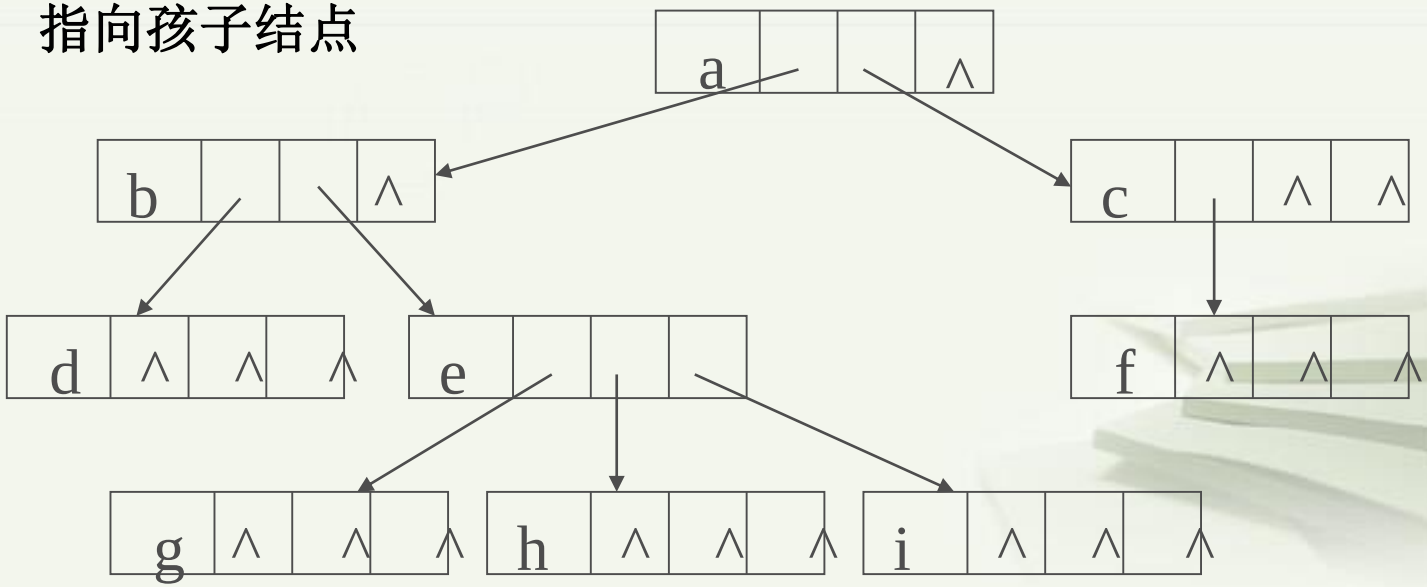
定长结点：

data	c_1	c_2	… …	c_d

data： 数据元素

$c_1, c_2, \ldots, c_d$: 指针，指向孩子结点

d： 树的度



存储有n个结点的树，有多少个指针？空指针有多少？

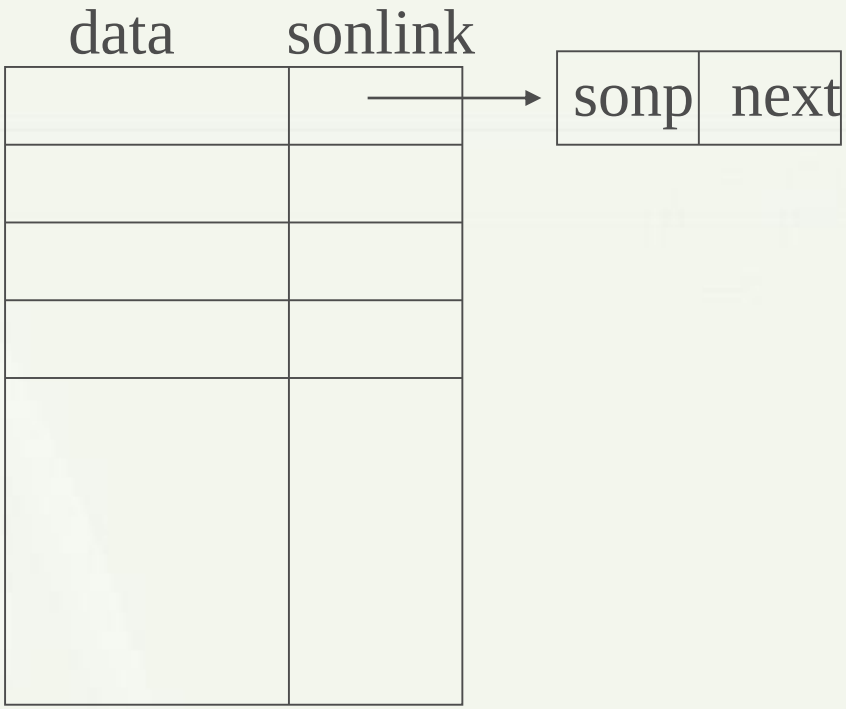
5.1 树

5.1.2 树ADT的实现

1. 树的存储结构

- 其他存储

存储方式1：用连续空间单元来存放树的各个元素，在存放元素的同时，还存放该元素的所有孩子的存储地址（链表，后继）。



其中：

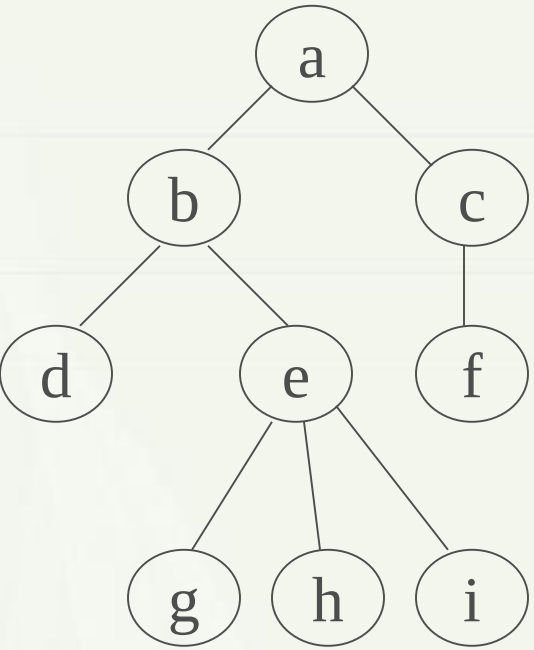
data: 数据元素值
sonlink: 指针，第1个孩子结点
sonp: 孩子的存储位置
next: 下一个孩子结点

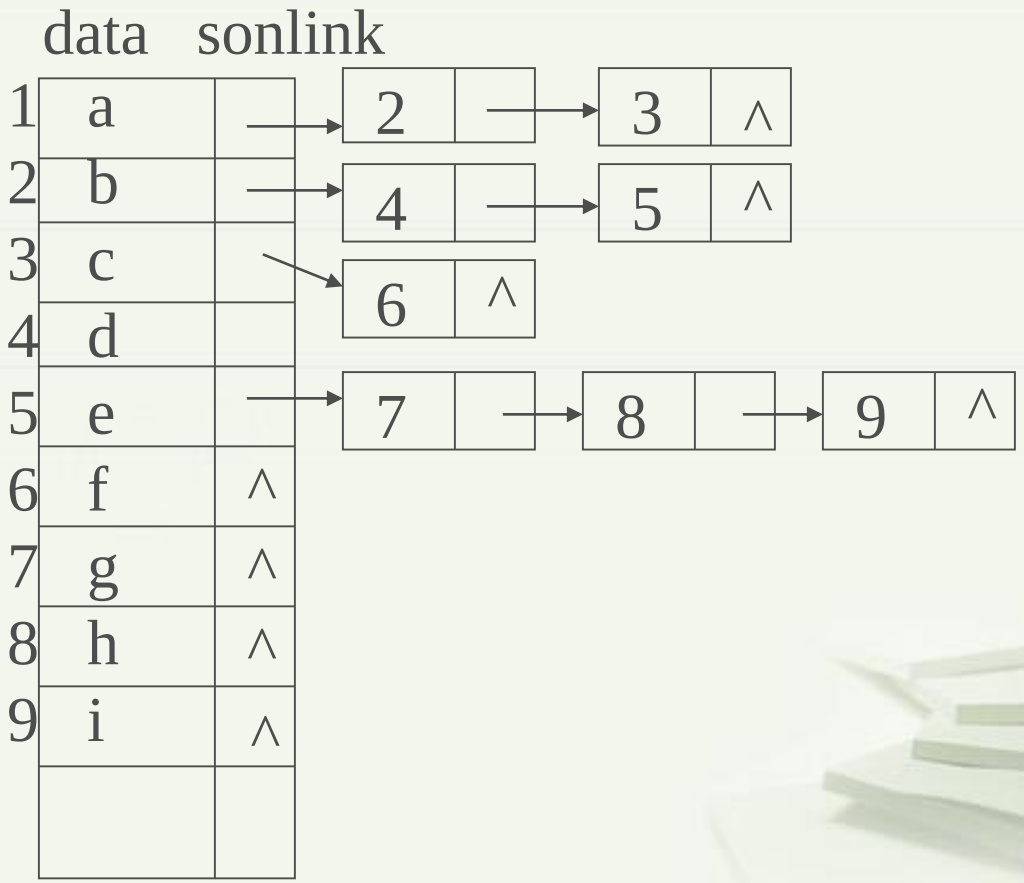
5.1 树

5.1.2 树ADT的实现

1. 树的存储结构

- 其他存储





5.1.2 树ADT的实现

1. 树的存储结构

- 其他存储

存储方式2：用任意空间单元来存放树的各个元素，在存放元素的同时，还存放该元素的第1个孩子的存储地址和右兄弟的存储地址。

data	fch	nsib

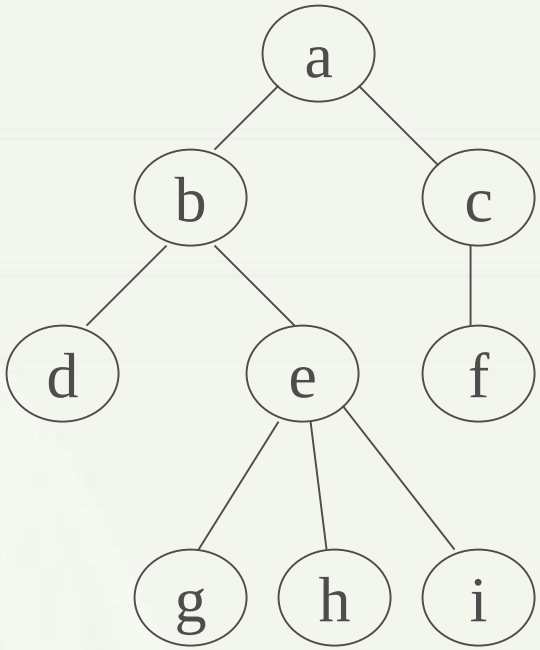
data: 数据元素值
fch: 指针，指向其第1个儿子结点
nsib: 指针，指向其右兄弟结点

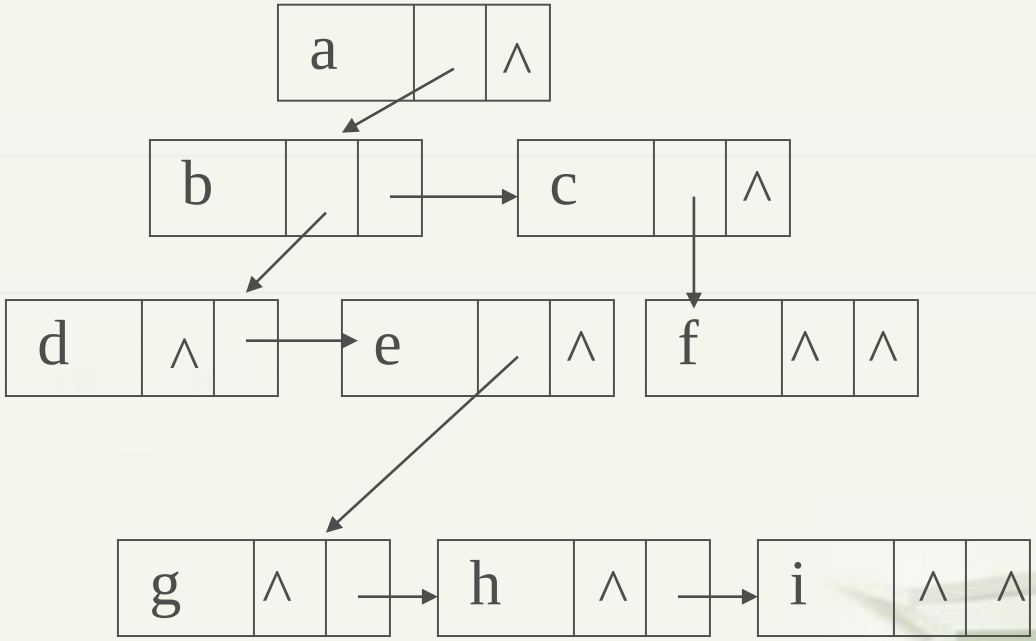
5.1 树

5.1.2 树ADT的实现

1. 树的存储结构

- 其他存储





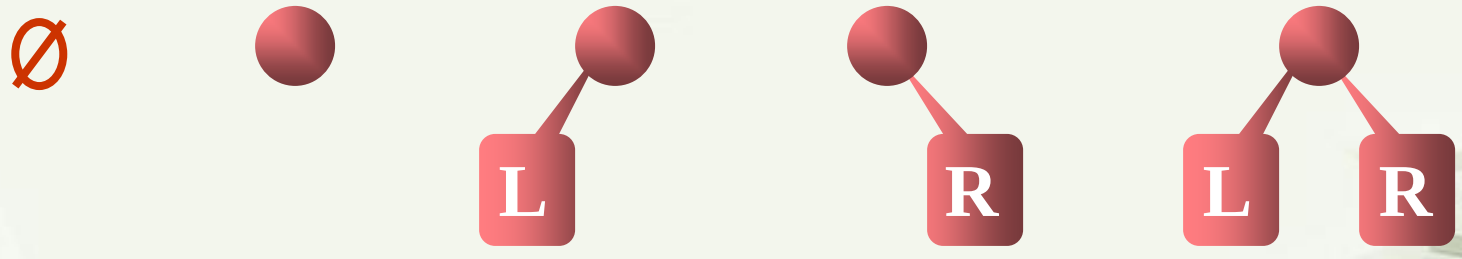
5.2 二叉树

5.2.1 二叉树ADT的定义

1. 二叉树数据结构

[二叉树 Binary Tree] 是一个有限元素的集合(可以空)，它或者是空（空二叉 树），或者由一个称为根的元素以及分别称为左子树和右子树的两个互不相交的集合组成，这个两个集合又都是二叉树。

问：二叉树有几种可能的形态?



5.2 二叉树

5.2.1 二叉树ADT的定义

1. 二叉树数据结构

Binary-Tree=(D,R)

$D=\{a_i \mid i=1,2,...,n\ n\geq 0 \quad a_i \in D_0\}$

$=\{ root \} \cup D_l \cup D_r \quad root \in D_0 \qquad D_l \cap D_r = \varnothing$

R={ LH,RH }

$$LH=\begin{cases} <root, r_l> \quad | \quad T_l=(D_l, R_l)\ r_l是T_l的根 \\ \varnothing \end{cases}$$

$$RH=\begin{cases} <root, r_r> \quad | \quad T_r=(D_r, R_r)\ r_r是T_r的根 \\ \varnothing \end{cases}$$

二叉树的有关术语：略！

5.2 二叉树

5.2.1 二叉树ADT的定义

2. 二叉树数据结构上定义的操作

同树结构，略

3. 二叉树ADT的定义

同树结构类似，略

4. 二叉树的性质

性质1：二叉树的第 i 层上至多有 2^{i-1} 个结点(i≥1)　最少？

性质2：深度为 k 二叉树至多有 2^k -1个结点(k≥1)　最少？

$$\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

5.2 二叉树

5.2.1 二叉树ADT的定义

4. 二叉树的性质

*性质3：*对任意一棵二叉树T，如果其叶子数为n_0,度为2的结点数为n_2，则 $n_0=n_2+1$

证明：设度为1的结点数为n_1，总结点数为n

$\therefore\ n=n_0+n_1+n_2$

设分支数为B，则 $B=n-1=n_0+n_1+n_2-1$

又 $B=n_0\times0+n_1\times1+n_2\times2$

$\therefore\ n_0\times0+n_1\times1+n_2\times2=n_0+n_1+n_2-1$

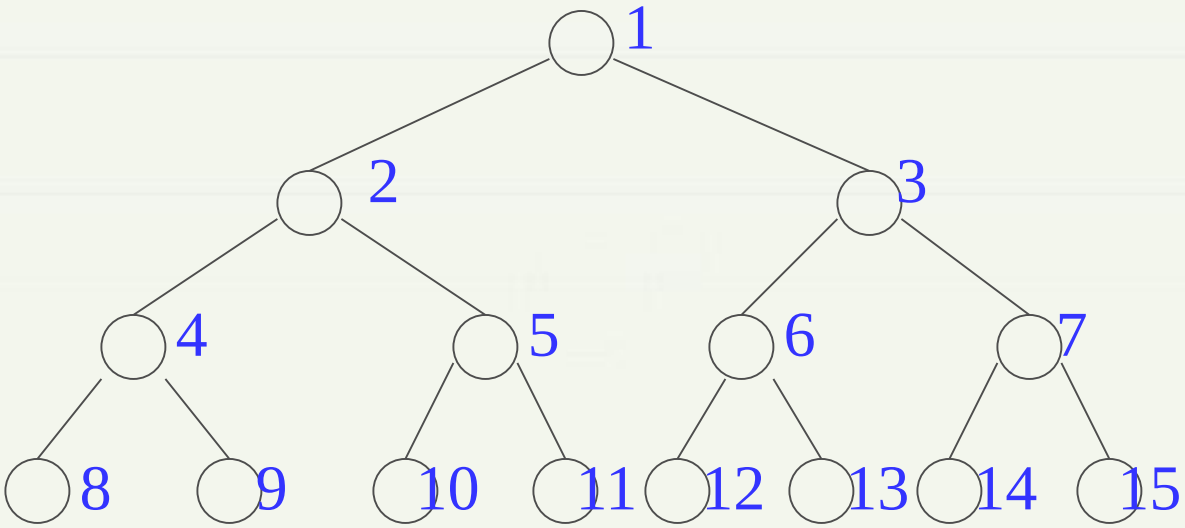
$\therefore\ n_0=n_2+1$

5.2 二叉树

5.2.1 二叉树ADT的定义

4. 二叉树的性质

[满二叉树 full binary tree] 高度为 k 的二叉树，若具有 2^k-1 个结点，称为满二叉树。



对二叉树的结点按逐层从上到下、每层从左向右进行编号，于是每个结点都有一个序号。

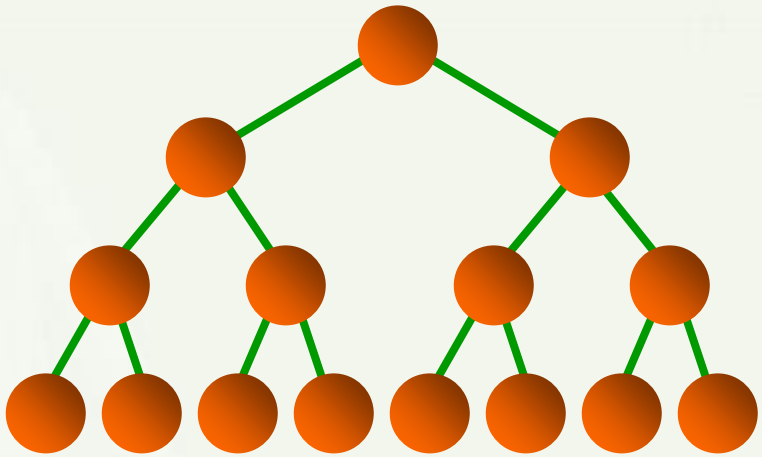
5.2 二叉树

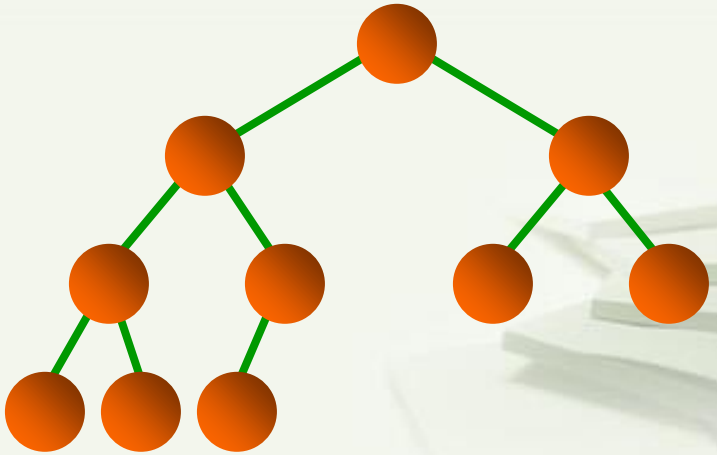
5.2.1 二叉树ADT的定义

4. 二叉树的性质

[完全二叉树 complete binary tree] 假设二叉树具有 n 个结点，对二叉树的结点进行编号，若二叉树各结点与深度相同的满二叉树中编号相同的 1~n 个各结点一一对应，则称为完全二叉树。

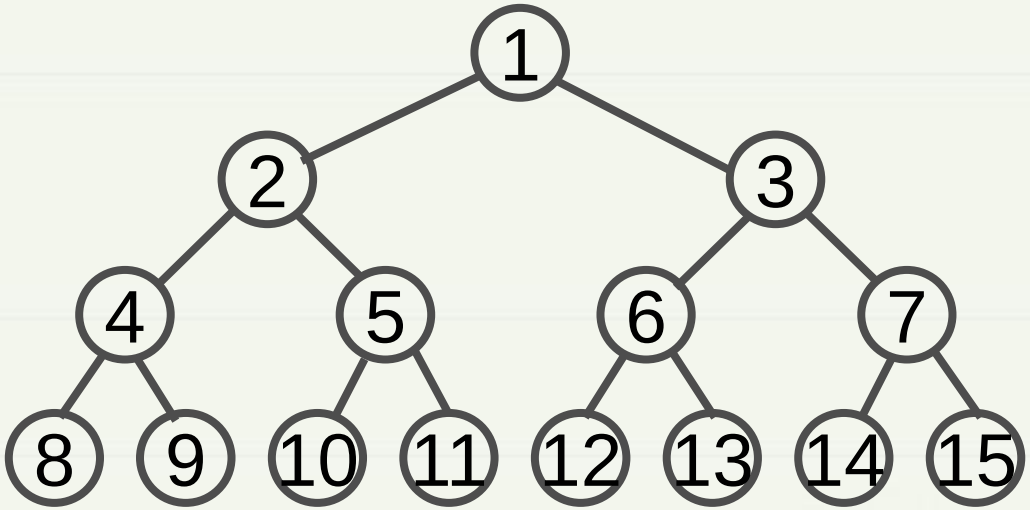
若设二叉树的深度为 *k*，前k-1层是满的，不满的最后一层，结点首先出现在左边！

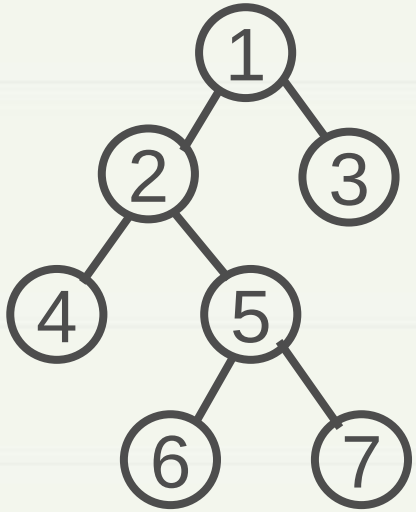


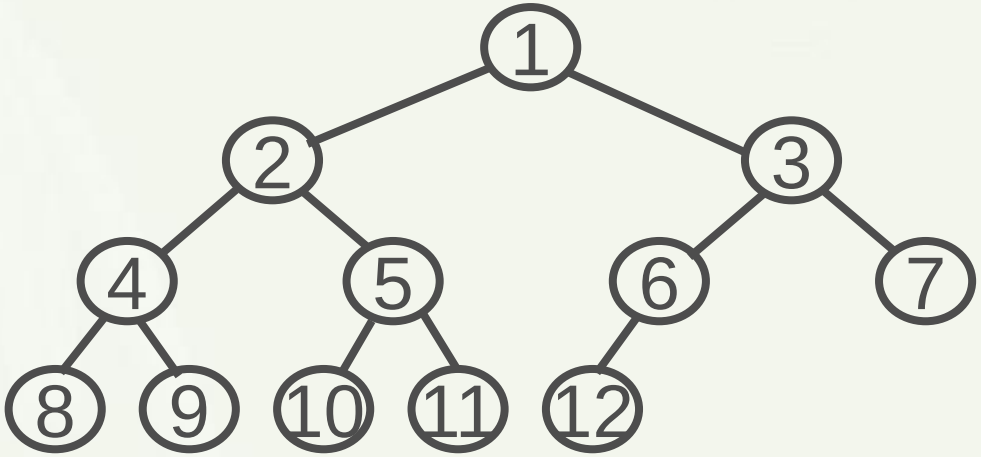


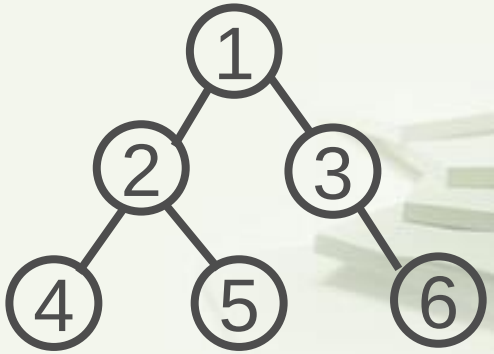
5.2 二叉树

5.2.1 二叉树ADT的定义









5.2 二叉树

5.2.1 二叉树ADT的定义

4. 二叉树的性质

性质4: 具有 n ($n\geq 0$) 个结点的完全二叉树的深度为$\lceil \log_2(n+1) \rceil$ 或$\lfloor \log_2 n \rfloor +1$

证明: 设其深度为 k , 于是， 该二叉树最大

$\therefore$ $n \leq 2^k-1$ //深度为k的满二叉树

又根据：深度为k-1的满二叉树，

$2^{k-1}-1 < n \leq 2^k-1$

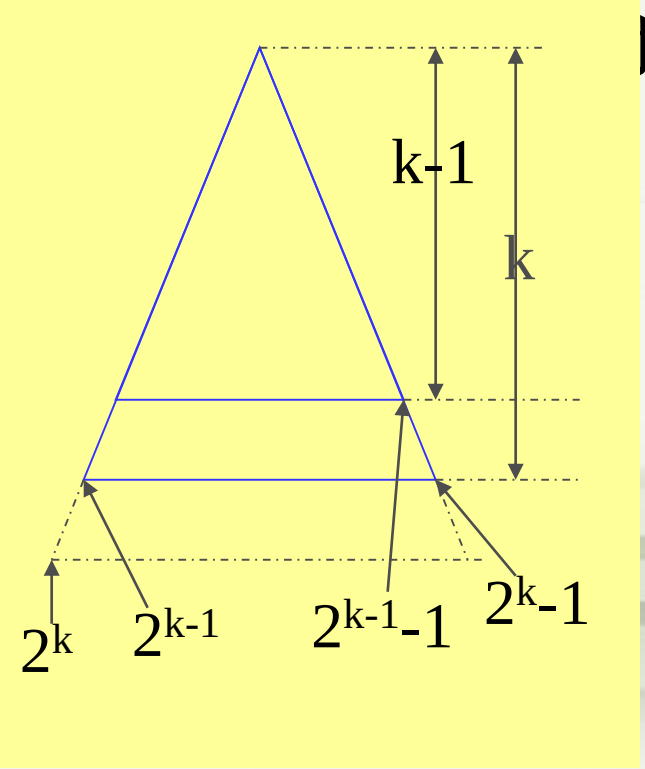
$\therefore$ $2^{k-1}-1+1 \leq n < 2^k-1+1$ 即 $2^{k-1} \leq n <$

两边取对数 $\therefore$ $k-1 \leq \log_2 n < k$

由于 k 是整数 $\therefore$ $k-1 \leq \log_2 n$

$\therefore$ $k-1 = \lfloor \log_2 n \rfloor$

$\therefore$ $k = \lfloor \log_2 n \rfloor +1$



5.2 二叉树

5.2.1 二叉树ADT的定义

4. 二叉树的性质

性质5： 如果对一棵具有 n 个结点的完全二叉树的结点进行编号，则对任一结点 i (1≤ i≤ n) 有：

（1） 若 i =1 ，则该结点是二叉树的根，无双亲；
否则，其双亲结点的编号为⌊ i/2 ⌋ 或⌈ (i-1)/2 ⌉；

（2）如果 2i>n 则结点 i 没有左孩子，
否则，其左孩子的编号为 2i

（3）如果 2i+1>n 则结点 i 没有右孩子，
否则，其右孩子的编号为 2i+1

（4）若 *i* 为奇数, 且 *i*≠1，则其左兄弟为*i*-1,
若 *i* 为偶数, 且*i*≠*n*, 则其右兄弟为*i*+1

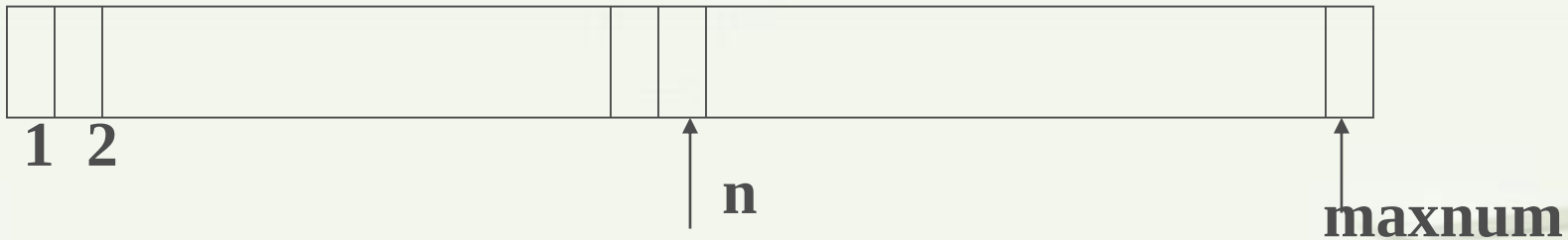
5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.1 基于顺序存储的实现

1. 存储结构

存储方式：用地址连续的空间单元来存储二叉树的各个元素，但为了表示关系，利用二叉树的性质，元素存放时，首先确定一个序号，该序号是对二叉树*按完全二叉树形式*编号而得，编号为 i 的存放在第i 个位置。



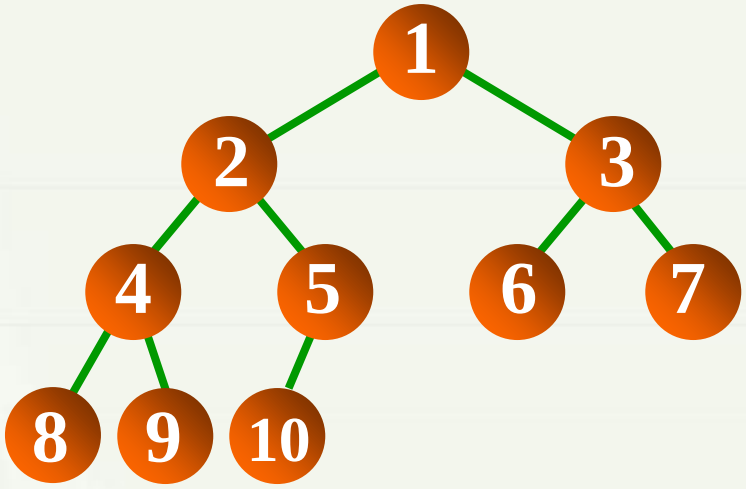
特点：存储位置体现层次关系，*双亲 i/2 ,孩子 2i, 2i+1*

但是，插入、删除需移动元素；空间效率低。

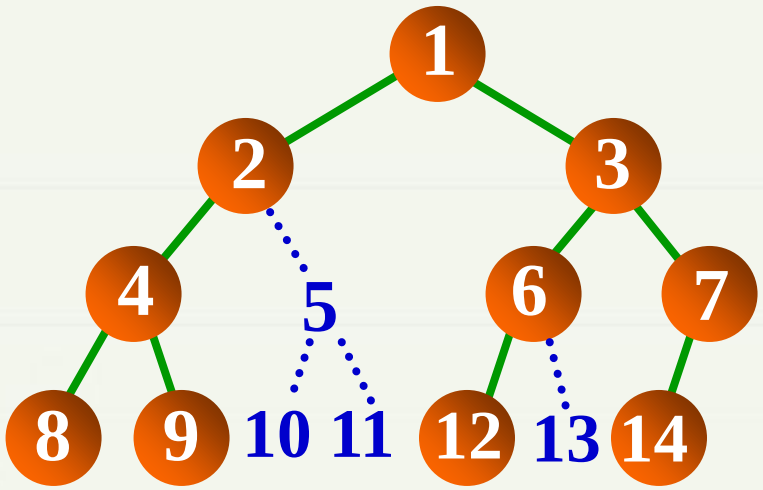
5.2 二叉树

5.2.2 二叉树ADT的实现

5. 2. 2. 1 基于顺序存储的实现



完全二叉树
的顺序存储



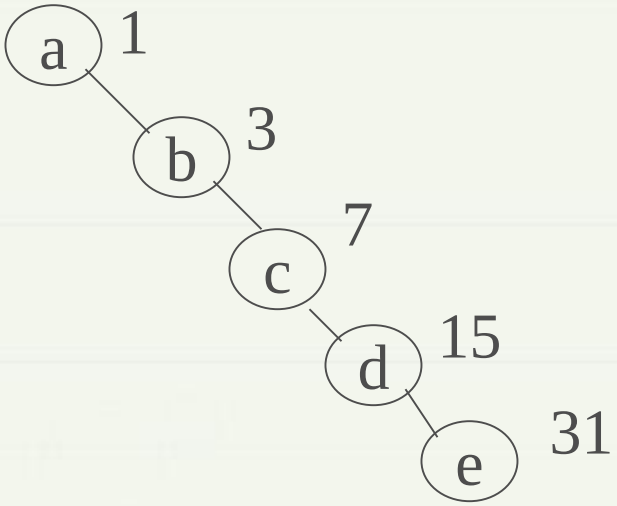
一般二叉树
的顺序存储

5.2 二叉树

5.2.2 二叉树ADT的实现

5. 2. 2. 1 基于顺序存储的实现

极端情况：



1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5
a		b				c								d																3						

分析空间占用情况：

5.2 二叉树

5.2.2 二叉树ADT的实现

5. 2. 2. 1 基于顺序存储的实现

高级语言实现：

```
#define maxdepth 二叉树的最大深度
#define maxlen  2^maxdepth-1   //maxlen连续空间的最大值
typedef  用户数据类型  ElemType;      //定义数据元素类型
struct sqbitree
 {     ElemType elem[1..maxlen];     //连续存储空间
       nd:0..maxlen          //二叉树占用空间大小
 }
```

2. 操作的实现：略！

5.2 二叉树

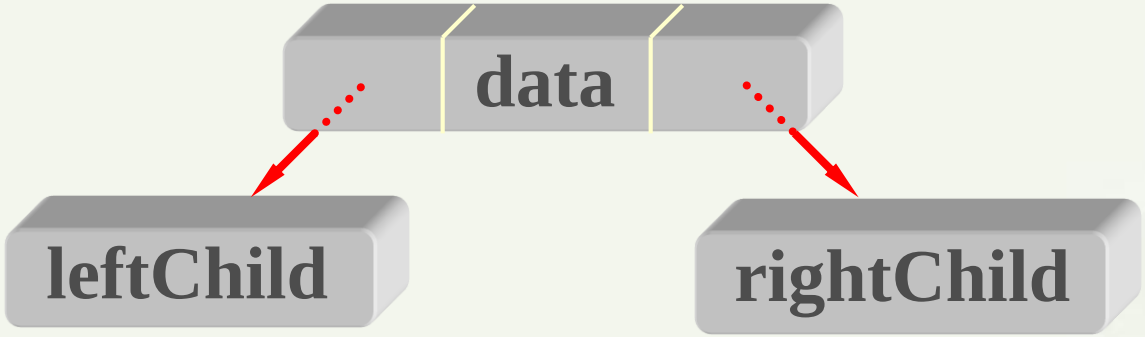
5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

1. 存储结构

存储方式：用任意的空间单元来存储二叉树的各个元素，在存储元素时，同时也存储其相关元素的地址（左、右孩子、双亲）。

二叉链式：



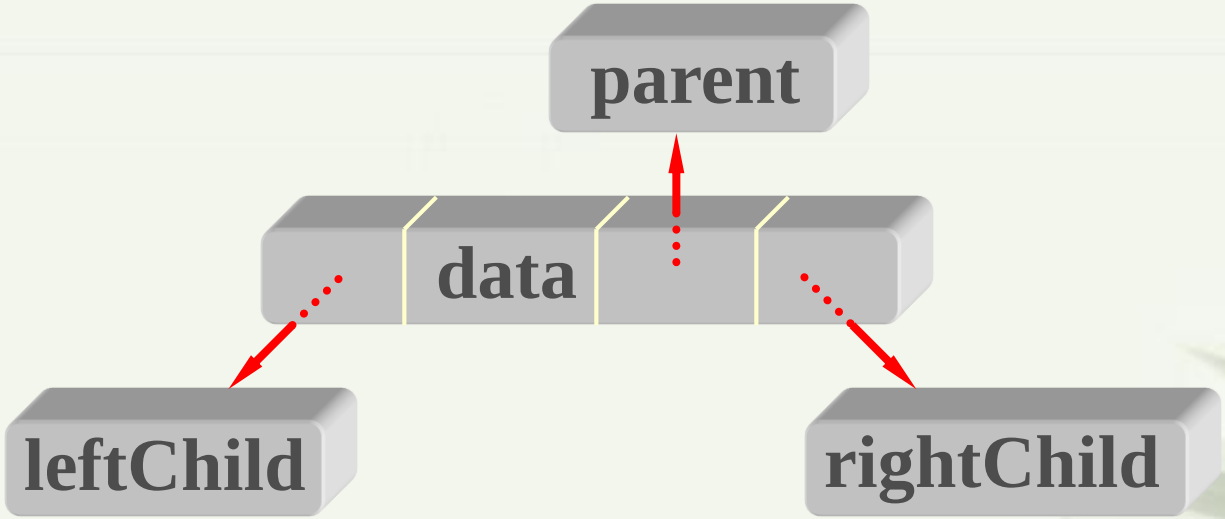
5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

1. 存储结构

三叉链式：



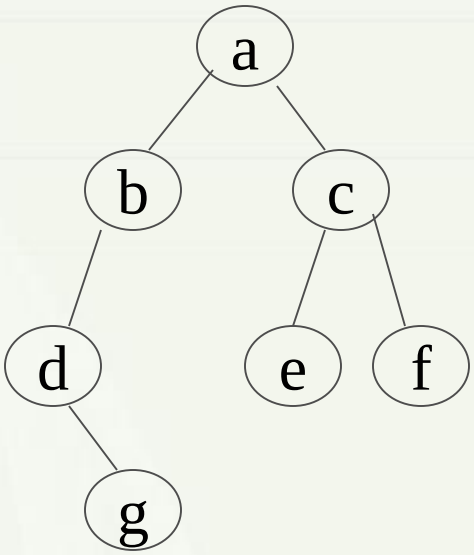
5.2 二叉树

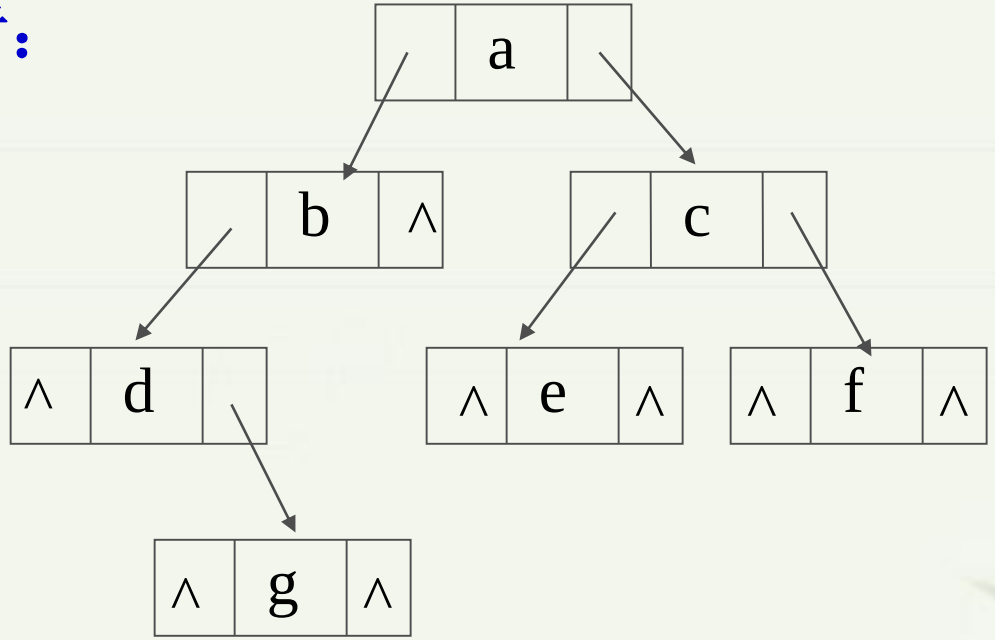
5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

1. 存储结构

例如下面的二叉树：





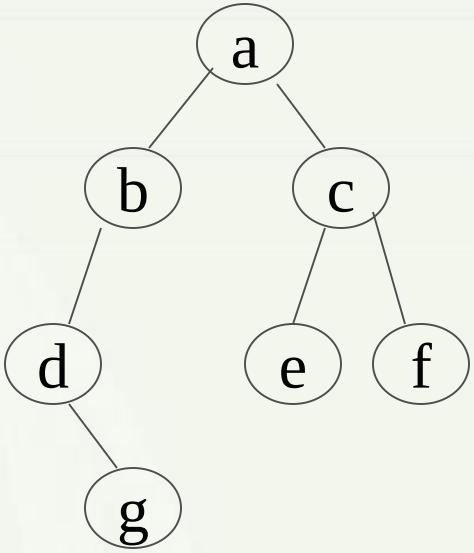
二叉链式存储

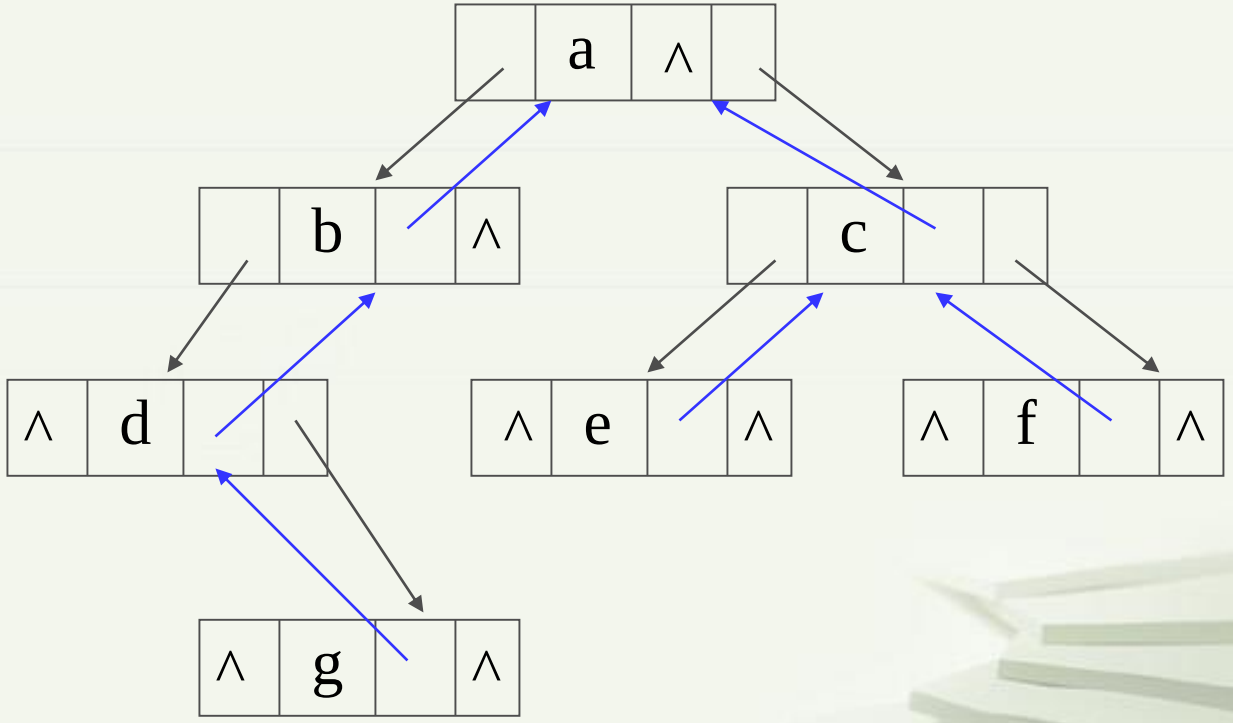
5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

1. 存储结构





三叉链式存储

5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

1. 存储结构

特点：

- ♣ 占用空间不随树的形态而变化
- ♣ n 个结点的二叉树，占用空间为：
  n ×(存储一个元素的空间+存储指针的空间)
- ♣ 对求孩子（双亲）操作易。
- ♣ 插入、删除元素不需移动，但调整指针多；
- ♣ 空指针多；

2n-(n-1)=n+1

5.2 二叉树

5.2.2 二叉树ADT的实现

5. 2. 2. 2 基于链式存储的实现

2. 二叉树基于链式存储的类定义

```
#include <iostream.h>
class BinTreeNode  //结点类的定义
{  public:
     BinTreeNode ( ) { leftChild =NULL, rightChild =NULL }
     BinTreeNode ( DataType x, BinTreeNode *left = NULL,
                 BinTreeNode *right = NULL ) : data (x),  leftChild (left),
    rightChild(right) { }              //构造函数
    ~BinTreeNode ( ) { }           //析构函数
  private:
    BinTreeNode *leftChild,  *rightChild;  //左、右子女链域
    DataType data;                                    //数据域
};
```

5.2 二叉树

5.2.2 二叉树ADT的实现

5. 2. 2. 2 基于链式存储的实现

2. 二叉树基于链式存储的类定义

```
class BinaryTree    //二叉链式二叉树类
{ public:
    BinaryTree(): root (NULL) { }
    BinaryTree ( DataType value ) { RefValue =value), root =NULL; }
    ~BinaryTree(){ destroy ( root );}
    void CreateBinTree( ) { CreateBinTree(root);}   //建立二叉树
   int IsEmpty ( ) { return (root == NULL) ? true : false; }
   BinTreeNode  *Parent ( BinTreeNode  *current )
   { return (root == NULL || root == current)?NULL : Parent ( root,
   current ); }
```

5.2 二叉树

5.2.2 二叉树ADT的实现

5. 2. 2. 2 基于链式存储的实现

```
BinTreeNode  *LeftChild (BinTreeNode  *current )
{ return (current != NULL )? current->leftChild :NULL; }
BinTreeNode  *RightChild (BinTreeNode  *current )
{ return ( current!= NULL) ? current->rightChild : NULL; }
int Height( ){return Height(root);}
int Size( ){return Size(root);}
BinTreeNode  *GetRoot ( ) const { return root; }
void preOrder( )  {preOrder(root);}          //前序遍历
void inOrder( )    {inOrder(root);}          //中序遍历
void postOrder( ) {postOrder(root);}         //后序遍历
 void levelOrder( ) {levelOrder(root);}      //层序遍历
```

5.2 二叉树

5.2.2 二叉树ADT的实现

5. 2. 2. 2 基于链式存储的实现

```
private:
    BinTreeNode  *root;                    //二叉树的根指针
    DataType RefValue;                     //数据输入停止标志
    void CreateBinTree( BinTreeNode * & subTree)
    BinTreeNode  *Parent ( BinTreeNode * subTree, BinTreeNode
    *current );
    int Height(BinTreeNode  *subTree);
    int Size(BinTreeNode  *subTree);
    void preOrder(BinTreeNode &subTree );     //前序遍历
    void inOrder(BinTreeNode &subTree );      //中序遍历
    void postOrder(BinTreeNode &subTree );    //后序遍历
    void levelOrder(BinTreeNode &subTree);    //层序遍历
    void destroy(BinTreeNode* &subTree);           };
```

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

部分成员函数的实现：

```
BinTreeNode *BinaryTree::Parent (BinTreeNode *subTree,
    BinTreeNode *current)
{//私有函数: 从结点 subTree 开始, 搜索结点 current 的
 //双亲, 若找到则返回双亲结点地址, 否则返回NULL
    if (subTree == NULL) return NULL;
    if (subTree->leftChild ==current || subTree->rightChild == current )
        return subTree;        //找到, 返回父结点地址
     BinTreeNode *p;
    if ((p = Parent (subTree->leftChild, current)) != NULL)
        return p;              //递归在左子树中搜索
    else return Parent (subTree->rightChild, current);
                               //递归在右子树中搜索
};
```

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

部分成员函数的实现：

```
void BinaryTree::destroy (BinTreeNode* subTree)
{//私有函数: 删除根为subTree的子树
    if (subTree != NULL)
    {    destroy (subTree->leftChild);     //删除左子树
         destroy (subTree->rightChild);    //删除右子树
         delete subTree;                   //删除根结点
    }
};
```

5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

★ 遍历操作

遍历(Traversal)：把二叉树中的结点访问且仅访问一次，又称为扫描。

遍历方式：由于元素之间的关系复杂了，按什么样的顺序访问数据元素？ 人们约定一个原则。

- 深度优先遍历
- 广度优先遍历

5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

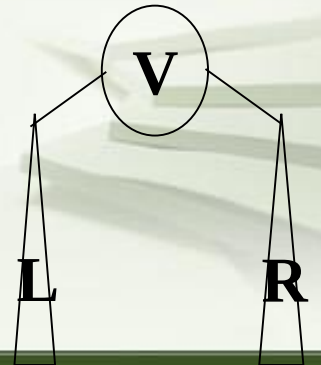
3. 二叉树基于链式存储下的典型操作的实现

（1）深度优先遍历：

基本思想是： 按某种原则访问一个元素，由于它与多个元素有关系（1个前趋、2个后继），按同样原则选择一个元素继续访问。

假设一棵树表示为根（V）、左子树（L）、右子树（R）则，它们的顺序关系有3！=6种：

VLR　　LVR　　LRV



如果规定人们习惯的先左后右，则剩下3种！

5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

（1）深度优先遍历：

前序遍历（VLR）：访问根；
前序遍历左子树；
前序遍历右子树；

Preorder Traversal

中序遍历（LVR）：中序遍历左子树；
访问根；
中序遍历右子树；

Inorder Traversal

后序遍历（LRV）：后序遍历左子树；
后序遍历右子树；
访问根；

Postorder Traversal

5.2 二叉树

5.2.2 二叉树ADT的实现

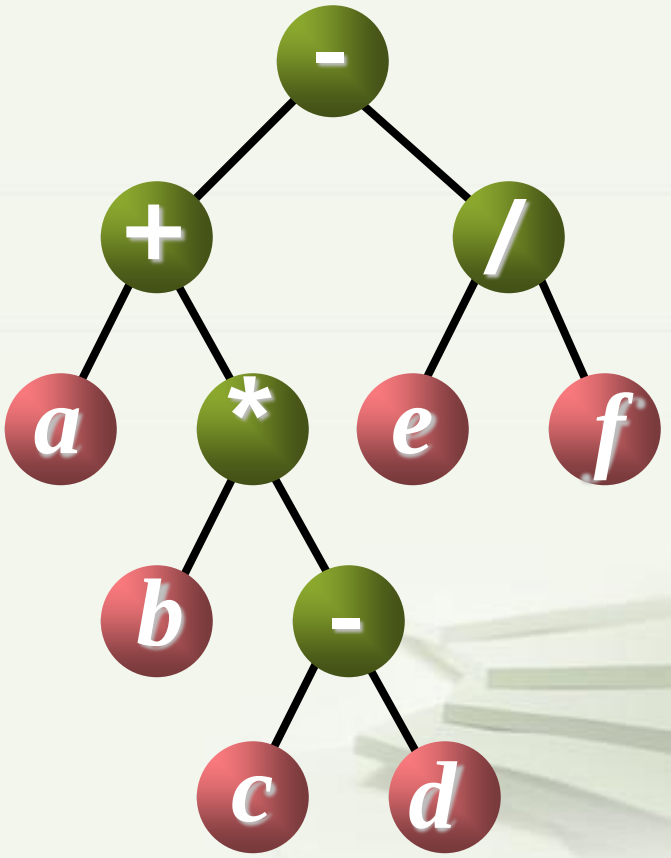
5.2.2.2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

（1）深度优先遍历：

遍历序列：按某种方式遍历二叉树的元素，得到的元素序列。

例，有二叉树如右，写出遍历序列。



前序遍历序列：-+a*b-cd/ef

中序遍历序列：a+b*c-d-e/f

后序遍历序列：abcd-*+ef/-

5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

（1）深度优先遍历：

中序遍历

```
void BinaryTree ::InOrder( BinTreeNode  *subTree )
 { //私有函数，中序遍历以subTree为根的二叉树
   if (subTree != NULL)
   {     //终止递归的条件
         InOrder (subTree->leftChild);
         cout << subTree->data;
         InOrder (subTree->rightChild );
   }
}
```

5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

（1）深度优先遍历：

前序遍历

```
void BinaryTree ::PreOrder( BinTreeNode  *subTree )
   {   //私有函数，前序遍历以subTree为根的二叉树
   if (subTree != NULL )
    {                  //终止递归的条件
        cout << subTree->data;
        PreOrder (subTree->leftChild );
        PreOrder (subTree->rightChild );
   }
}
```

5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

（1）深度优先遍历：

后序遍历

```
void BinaryTree ::PostOrder( BinTreeNode *subTree )
   {//私有函数，后序遍历以subTree为根的二叉树
     if (subTree != NULL )
     {  //终止递归的条件
        PostOrder (subTree->leftChild );
        PostOrder (subTree->rightChild );
        cout << subTree->data;
   }
}
```

5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

（1）深度优先遍历：

深度遍历的非递归算法：借助栈数据结构可以实现！

教材P203-206。略！

5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

（2）广度优先遍历：

广度优先遍历：又称为层次遍历，从第1层开始，逐层访问二叉树的元素，每一层从左向右。

先访问的元素，它们的孩子也先访问！

因此：用队列数据结构辅助来完成层次边历

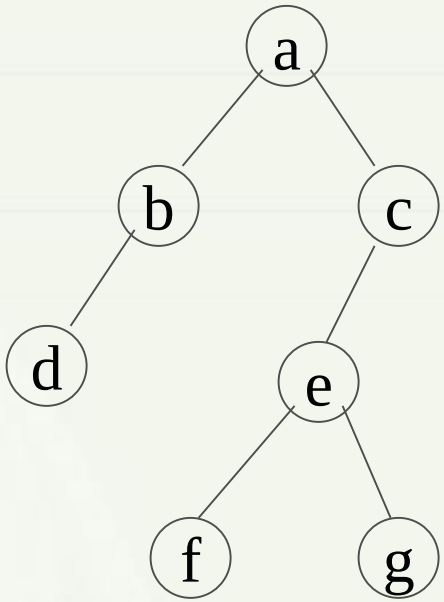
5.2 二叉树

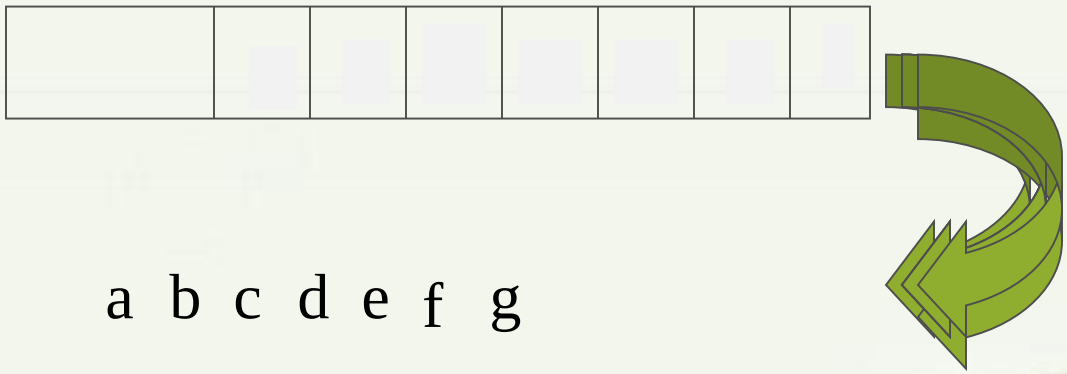
5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

（2）广度优先遍历：





5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

（2）广度优先遍历：

广度遍历

```
void BinaryTree::levelOrder (BinTreeNode *t)
 {   if (root == NULL) return;
     SeqQueue  Q;
     BinTreeNode *p = root;
     Q.EnQueue (p);
     while (!Q.IsEmpty ())
     {    Q.DeQueue (p); cout<<p->data;
          if (p->leftChild != NULL)  Q.EnQueue (p->leftChild);
          if (p->rightChild != NULL) Q.EnQueue (p->rightChild);
     }
};
```

5.2 二叉树

5.2.2 二叉树ADT的实现

5. 2. 2. 2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

特别强调：遍历是二叉树最最重要的操作之一，二叉树的很多操作可以借助遍历操作完成。

计算二叉树深度或高度操作：

```
int BinaryTree ::Depth( const BinTreeNode  * subTree ) const
{ //私有函数，计算根指针为subTree的二叉树的深度或高度
    if (subTree == NULL ) return 0;  //空二叉树的深度为0
    else return 1+ Max(Depth(subTree->leftChild ), Depth
(subTree->rightChild ) );
}
```

5.2 二叉树

5.2.2 二叉树ADT的实现

5. 2. 2. 2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

计算二叉树结点个数操作：

```
int BinaryTree ::Size ( const BinTreeNode  * subTree ) const { //
私有函数，计算根指针为subTree的二叉树中的结点个数
    if (subTree == NULL ) return 0;
                                        //空二叉树的结点个数为0
    else return 1 + Size (subTree->leftChild )
                  + Size (subTree->rightChild );
}
```

5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

计算二叉树的叶子数：

复制二叉树：

判断是否是满二叉树；

判断是否是完全二叉树；

判断二叉树是否相等；

……

5.2 二叉树

5.2.2 二叉树ADT的实现

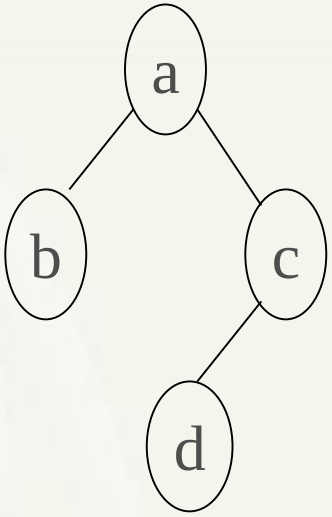
5. 2. 2. 2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

★ 创建二叉树操作

（1）根据二叉树的结构直接生成

方法：逐个生成各元素结点，根据元素关系连接指针；




```
New(p1);  p1->data='a';
New(p2);  p2 -> data='b';
New(p3);  p3 -> data='c';
New(p4);  p4 -> data='d';
```

```
p1 -> lchild=p2;    p1 -> rchild=p3;
p2 -> lchild =NULL;  p2 -> rchild = NULL;
p3 -> lchild =p4;   p3 -> rchild = NULL;
p4 -> lchild = NULL;  p4 -> rchild = NULL;
```

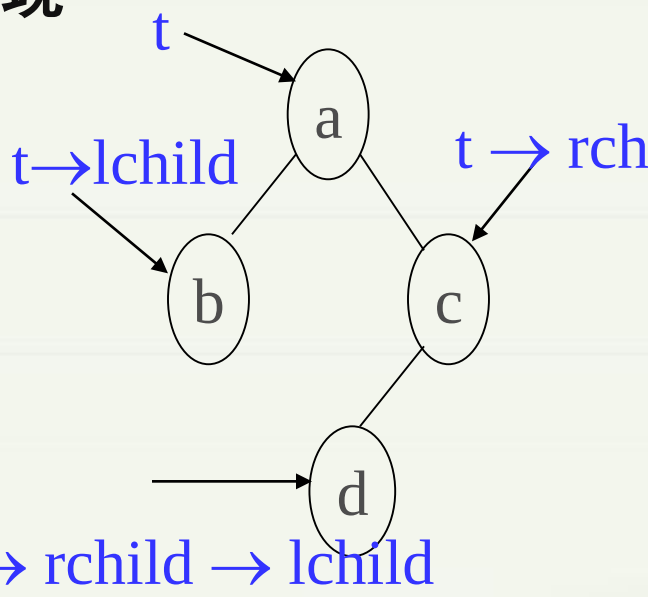
5.2 二叉树

5.2.2 二叉树ADT的实现

5. 2. 2. 2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

创建二叉树操作



或：

New(t); t->data='a';
New(t -> lchild); t -> lchild -> data='b';
New(t -> rchild); t -> rchild -> data='c';
t -> lchild -> lchild=NULL;
t -> lchild -> rchild= NULL;
New(t -> rchild -> lchild); t -> rchild -> lchild -> data='d';
t -> rchild -> rchild= NULL;
t -> rchild -> lchild -> lchild= NULL;
t -> rchild -> lchild -> rchild= NULL;

5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

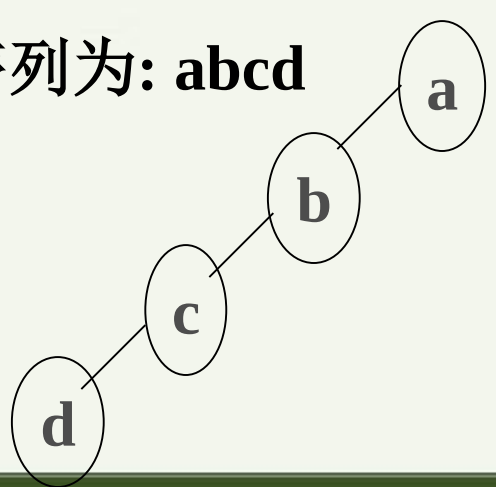
创建二叉树操作

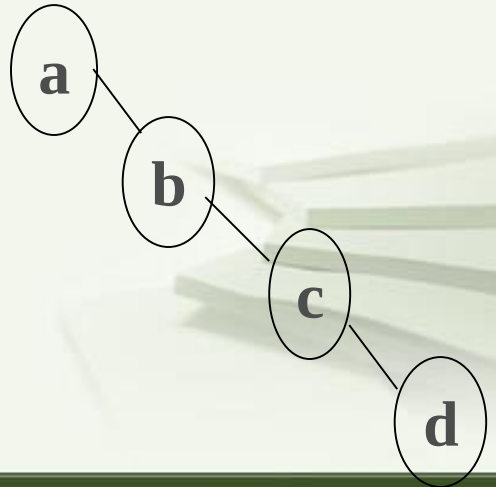
（2）已知能唯一确定一棵二叉树的元素序列，创建二叉树。

并不是任何的一个元素序列能够唯一确定一棵二叉树，因为，序列中不能反映出元素之间的关系！

例如：已知二叉树的前序序列为: **abcd**

那么，什么样的元素序列能唯一确定二叉树呢？





5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

- 已知完全前序序列，唯一确定一棵二叉树；

二叉树的完全前序序列定义为：

若二叉树为空，则序列为： ∅ （可以用其他符号表示）

否则，序列为：

<根元素><左子树的完全前序序列><右子树的完全前序序列>

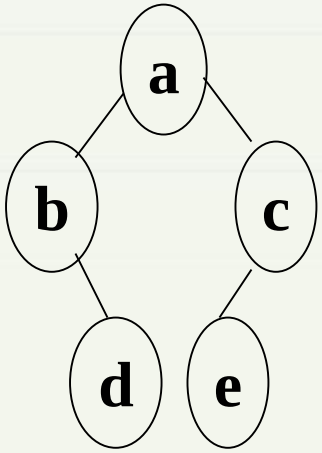
5.2 二叉树

5.2.2 二叉树ADT的实现

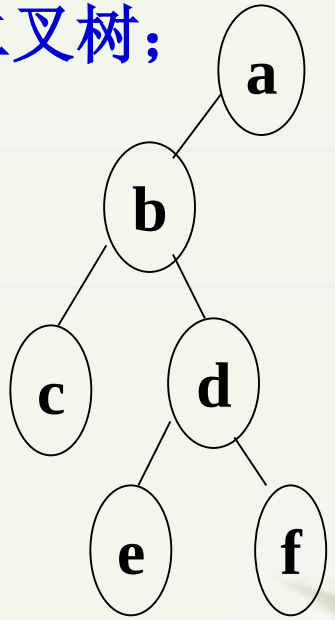
5.2.2.2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

- 已知完全前序序列，唯一确定一棵二叉树；



a b ∅ d ∅ ∅ c e ∅ ∅ ∅



a b c ∅ ∅ d e ∅ ∅ f ∅ ∅ ∅

5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

- 已知完全前序序列，确定一棵二叉树；

创建二叉树基本思想：

若完全前序序列为∅，则二叉树为空；

否则，完全前序序列的第1个元素是根，生成根结点，
左子树的完全前序序列生成左子树；
右子树的完全前序序列生成右子树

递归！

5.2 二叉树

5.2.2 二叉树ADT的实现

5. 2. 2. 2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

```
void BinaryTree ::CreateBinTree(BinTreeNode* & subTree)
 {//私有函数: 建立根为subTree的子树
    DataType item;          cin>> item;
     if (item != RefValue)
     {  subTree = new   BinTreeNode (item);
        if( subTree == NULL ) {cerr<<"存储分配错!" <<endl;        exit(1);}
       CreateBinTree( subTree->leftChild);
       CreateBinTree( subTree->rightChild);
        }
     else subTree = NULL;
};
```

P202 程序5.12

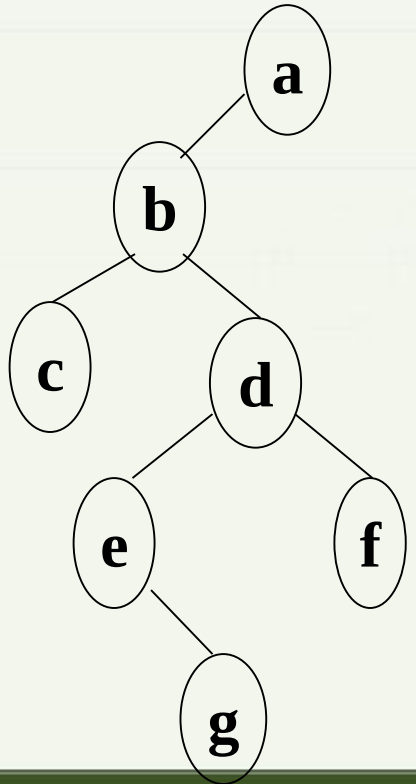
5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

举例： a b c ∅ ∅ d e ∅ g ∅ ∅ f ∅ ∅ ∅



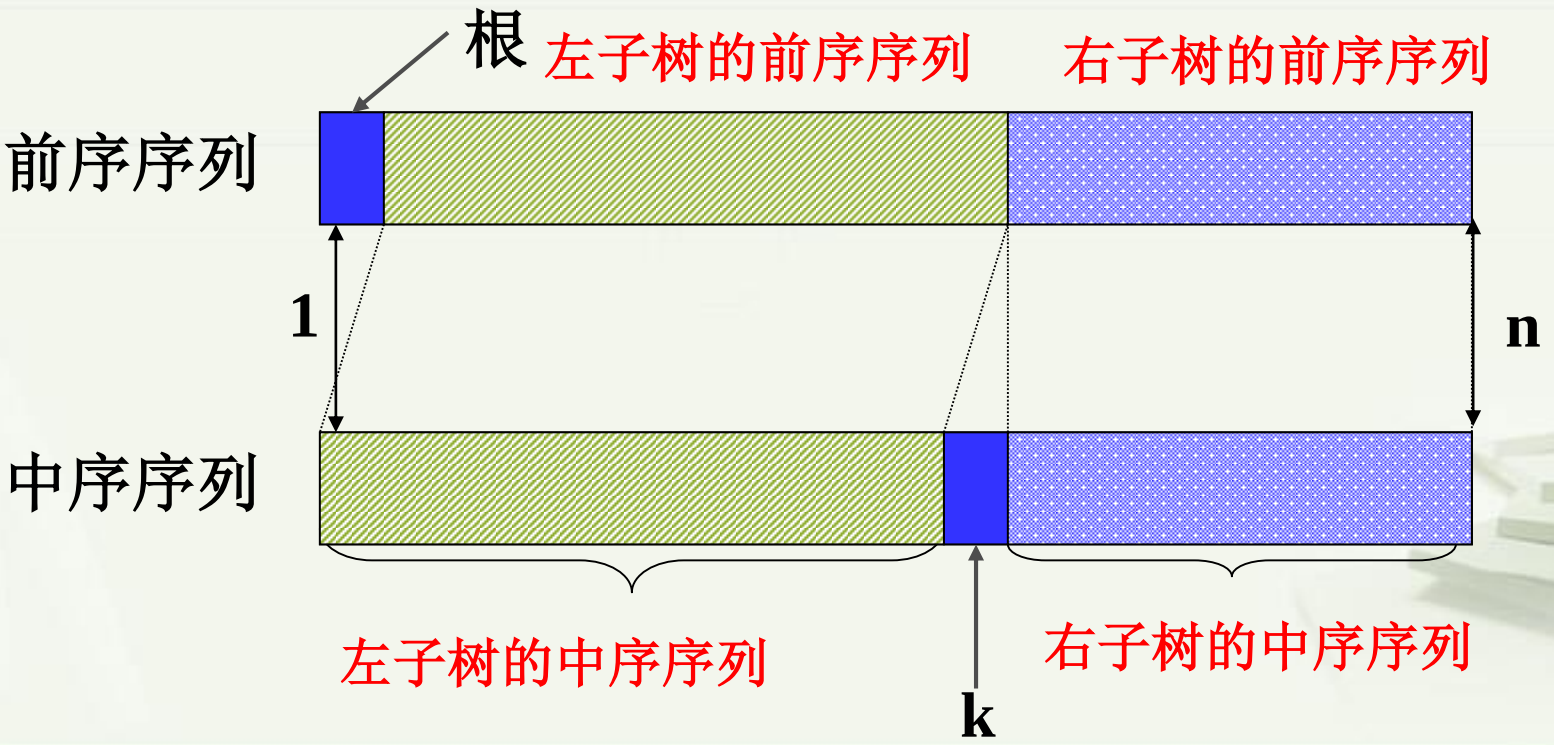
5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

◆ 已知二叉树的前序遍历序列和中序遍历序列，唯一确定一棵二叉树；



5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

已知后序、中序序列能否唯一确定二叉树？

已知前序、后序序列能否唯一确定二叉树？

证明你的结论

对数据元素有什么要求？

5.2 二叉树

5.2.2 二叉树ADT的实现

5.2.2.2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

例如，已知二叉树的前序和中序遍历序列如下：

前序： A B H F D E C K G

中序： H B D F A E K C G

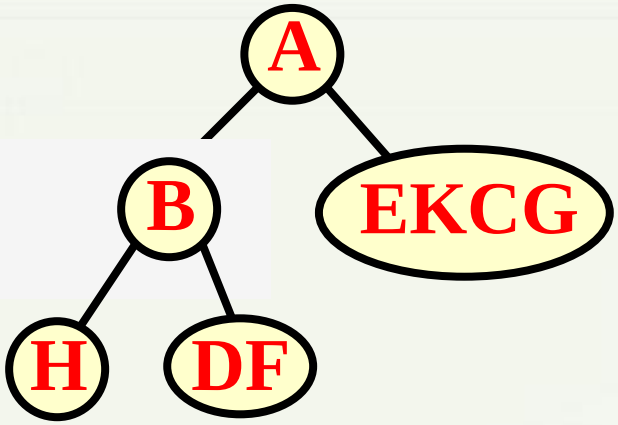
试构造出二叉树

A B H F D E C K G

H B D F A E K C G

B H F D

H B D F



5.2 二叉树

5.2.2 二叉树ADT的实现

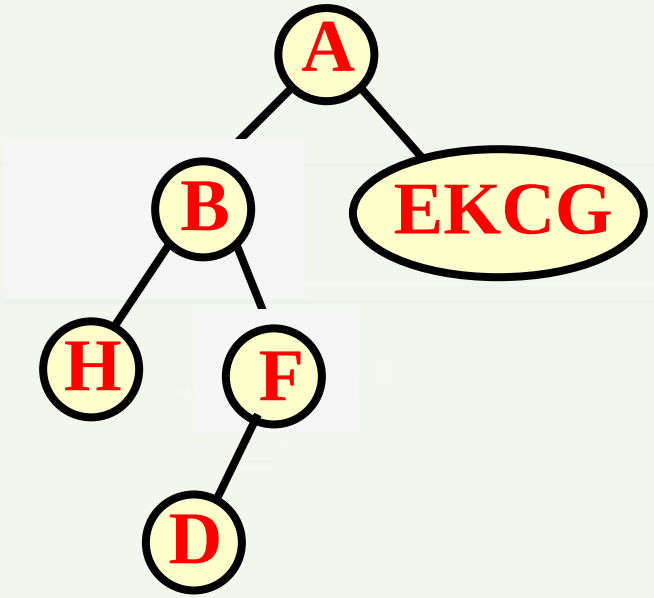
5. 2. 2. 2 基于链式存储的实现

3. 二叉树基于链式存储下的典型操作的实现

H
H

F D
D F

D
D



5.2 二叉树

5.2.2 二叉树ADT的实现

5. 2. 2. 2 基于链式存储的实现

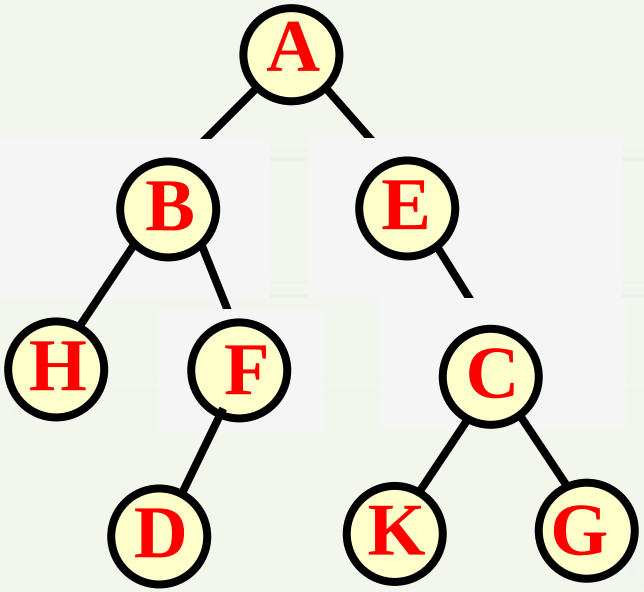
3. 二叉树基于链式存储下的典型操作的实现

E C K G
E K C G

C K G
K C G

K
K

G
G



5.3 线索二叉树

为什么讨论线索二叉树？

二叉树的数据元素具有*非线性关系*，但是访问元素时是要按某种顺序去访问，即通常会确定一种遍历方式，把二叉树的元素排成一种顺序，即得到二叉树元素的一个*线性序列*（具有线性关系），但是，这种*线性关系是依赖于遍历方式*的！

显然，对于一棵二叉树，如果遍历方式相同，每次调用遍历算法得到的结果序列（线性序列）是一样的。这也就意味着多次调用同一遍历算法没有必要，只要能把这种遍历方式下元素的顺序关系（线性关系）记录下来，以后对相同的遍历就不用去调用遍历算法，而可以直接用记录的线性关系。

这样，我们可以记录二叉树某种遍历方式下元素之间的线性关系，从而有了线索二叉树

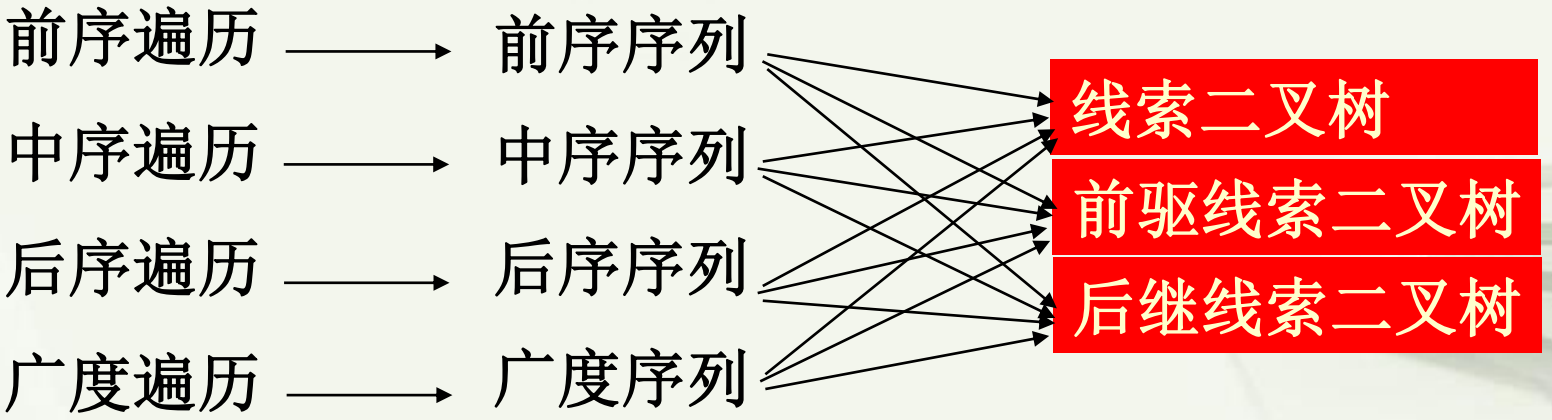
5.3 线索二叉树

[线索]：元素之间的线性信息，即前驱、后继信息；又分为前驱线索、后继线索

前驱线索
- 前序前驱线索
- 中序前驱线索
- 后序前驱线索
- 广度前驱线索

后继线索
- 前序后继线索
- 中序后继线索
- 后序后继线索
- 广度后继线索

[线索二叉树]：记录了线索信息的二叉树；



5.3 线索二叉树

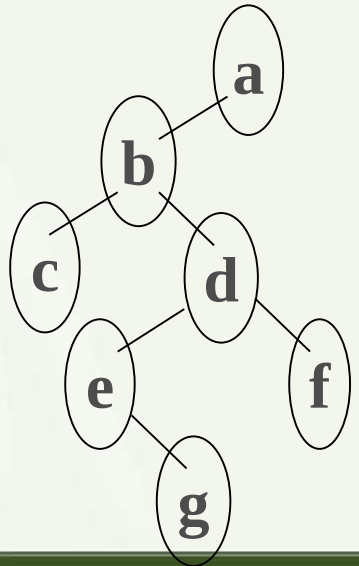
如何记录线索？

记录线索就是要存储二叉树元素之间的线性关系。我们前面已经学习过，存储线性关系可以顺序存储，也可以链式存储。

■ 顺序存储：用地址连续的空间把遍历得到的元素序列依次存放，用物理上相邻来存储线索！

优点：使用线索方便；

缺点：二叉树元素又存储了一份；



前序线索：	a	b	c	d	e	g	f
中序线索：	c	b	e	g	d	f	a
后序线索：	c	g	e	f	d	b	a

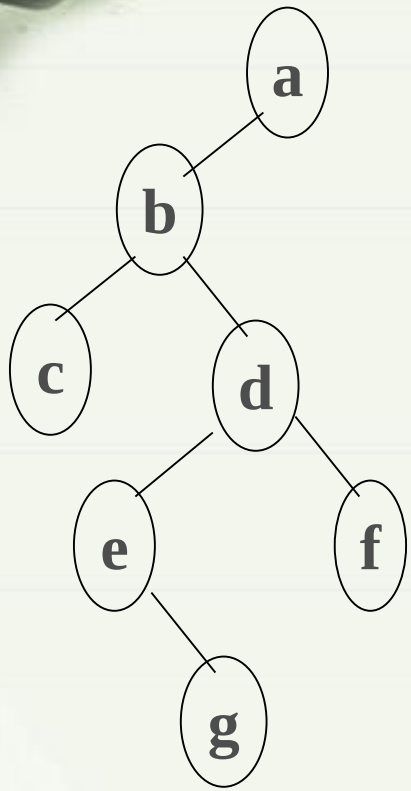
5.3 线索二叉树

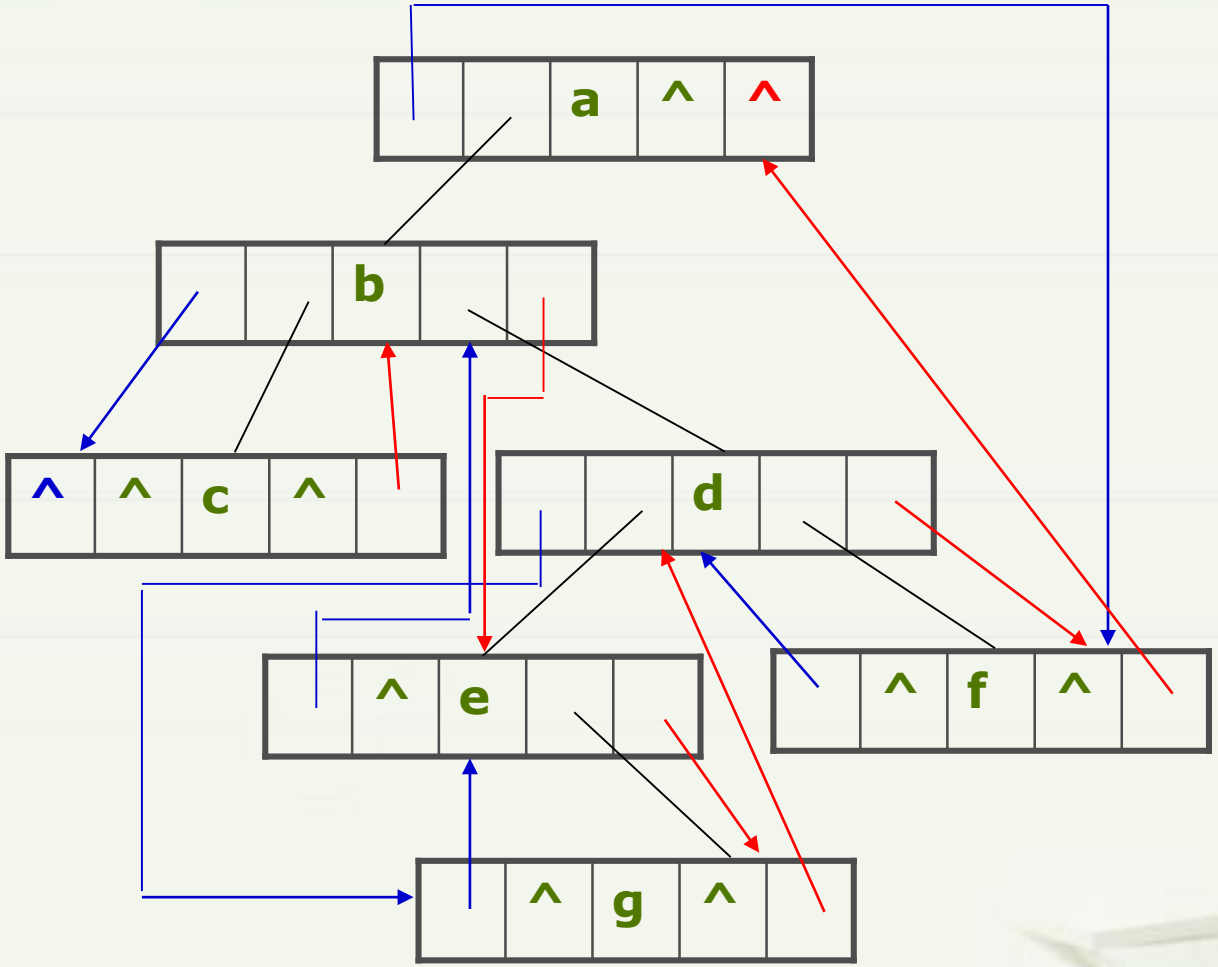
如何记录线索？

■ 链式存储：用指针把二叉树中元素在遍历得到的元素序列关系存储起来。

优点：直接在二叉树上存储了线索；
缺点：增加了指针（空间）；

5.3 线索二叉树





中序线索二叉树：

c b e g d f a

5.3 线索二叉树

如何记录线索？

无论是顺序还是链式存储线索的方式，都是把每个元素的线索都存储起来（存储的是完全线索），显然，使用线索时就很方便。但是付出的代价是增加了空间！

有没有节省空间存储线索的方式呢？ 这让我们想起了二叉树链式存储时的 n+1 空指针！ 如果能用这些空指针存储线索，显然可以节省空间。但是，存在问题：

第一，只有n+1个空指针，而存储完全线索需要2n个指针。因此，指针不够用，只能存部分线索！

第二，空指针存储了线索后，怎么知道它存储的是线索（而不是左右孩子？），因此，需要把线索指针与左右孩子指针区分开来。——可以用标志位。

5.3 线索二叉树

如何记录线索？

- 如果二叉树结点的左子树空，则左指针可以记录该结点的前驱线索；
- 如果二叉树结点的右子树空，则右指针可以记录该结点的后继线索；

通过给二叉树左右孩子指针增加标志位区分是左右孩子还是线索。

ltag	leftChild	data	rightChild	rtag

ltag=0: 指向左孩子
ltag=1: 指向前驱

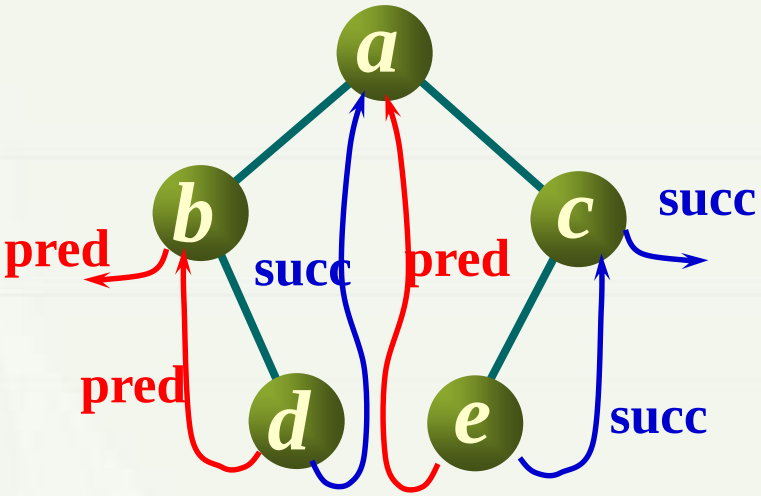
rtag=0: 指向右孩子
rtag=1: 指向后继

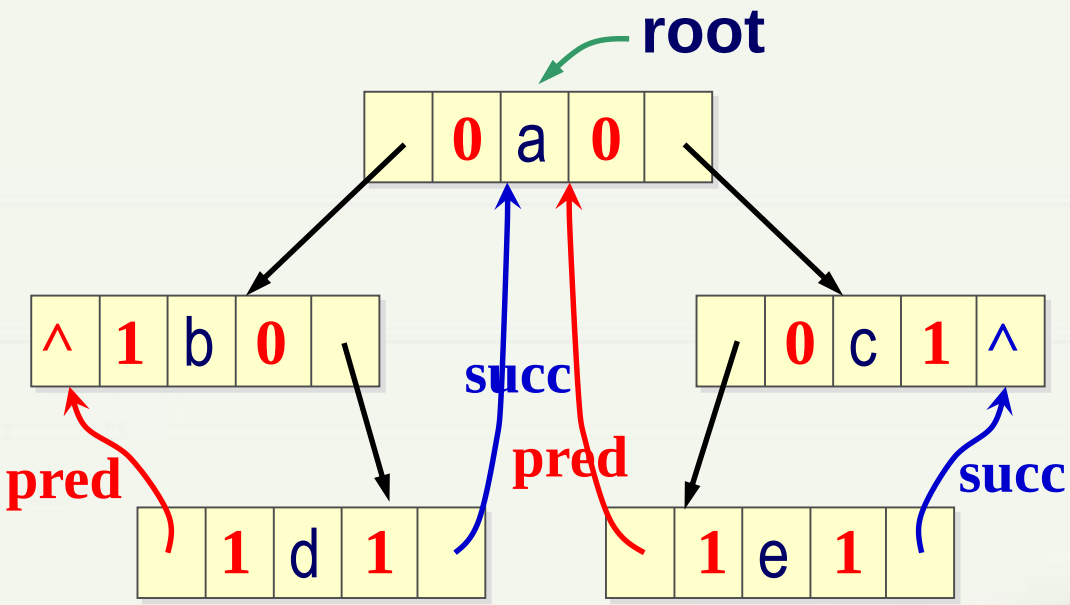
特点：空间利用率高，但记录线索不完全；

5.3 线索二叉树

如何记录线索？

例：画出下面二叉树的中序线索二叉树





线索二叉树的类定义：P212。 略！

上节课内容回顾

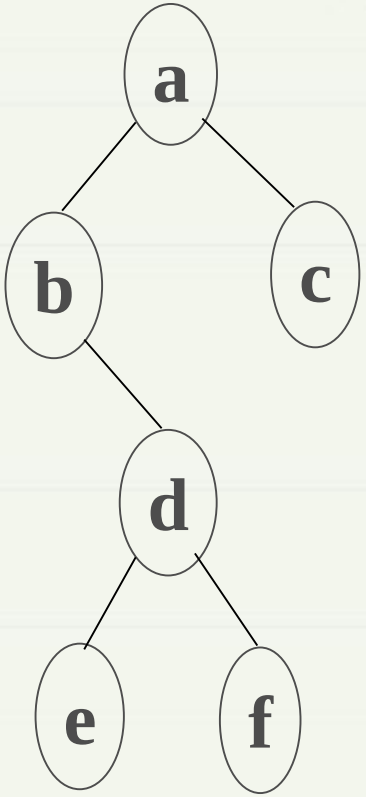
■ 线索二叉树

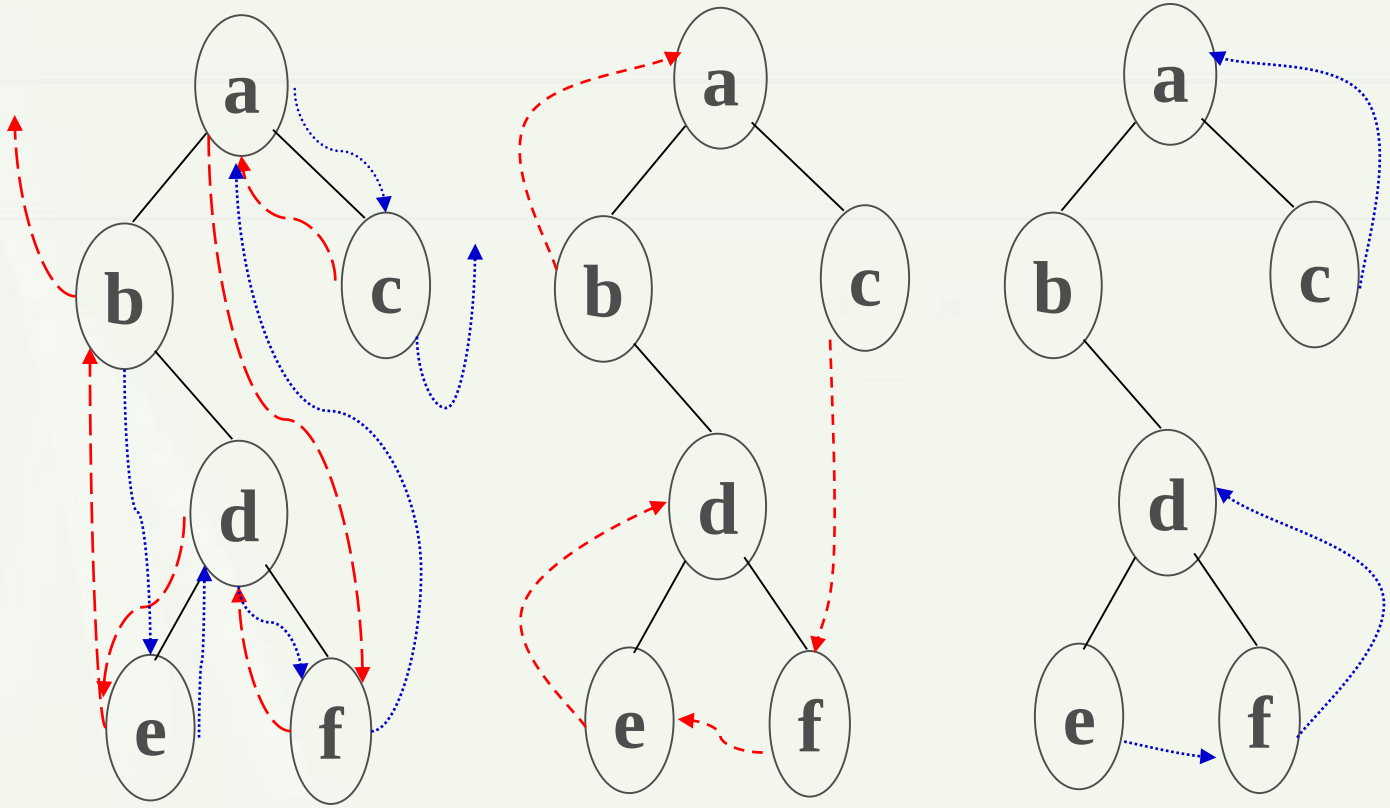
有下面的二叉树，完成：

（1）画出中序全线索二叉树；

（2）画出前序不完全前驱线索二叉树；

（3）画出后序不完全后继线索二叉树；





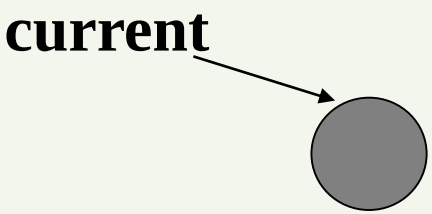
5.3 线索二叉树

如何使用线索？

如果记录的是完全线索，线索的使用非常简单：任何一个元素都可以通过记录的线索找到其前驱和后继（在某种遍历方式下！）。

但是，很遗憾！ 现在记录的是不完全线索，那么一个元素的前驱和后继就有可能不好得到。关键看是不是有线索记录着！

假设current指向当前元素，那么它的前驱和后继如何得到？



前驱：即前一个访问的元素

后继：即后一个访问的元素

5.3 线索二叉树

如何使用线索？

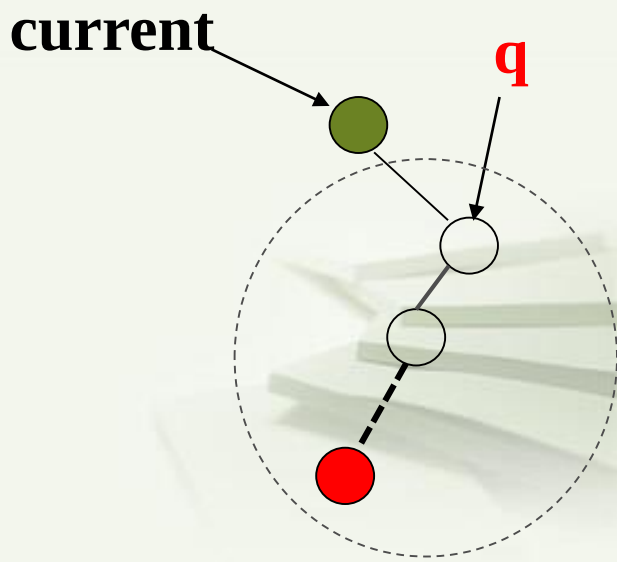
如果current所指元素记录了线索，很简单。怎么做？

如果current所指元素没记录线索，就麻烦了。怎么做？

（1）下面以中序线索二叉树为例，讨论如何得到前驱和后继：

- 寻找当前结点在中序下的后继

```
if (current->rtag ==1)
    return current->rightChild
else   //current->rtag == 0
    { q=current->rightChild;
      while(q->ltag==0)
          q=q->leftChild;
      rerturn q ; }
```

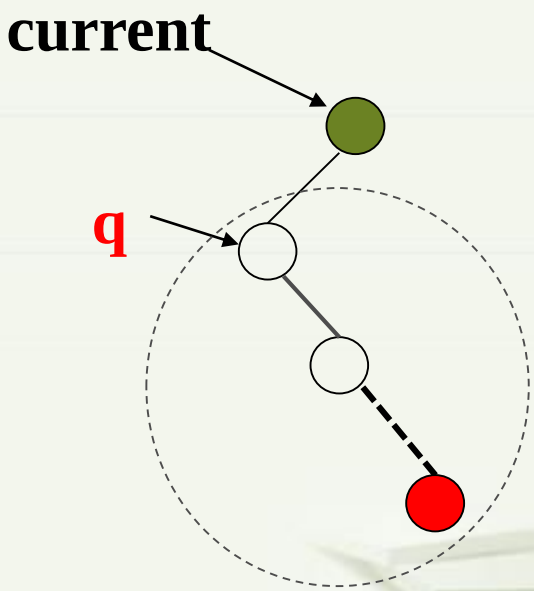


5.3 线索二叉树

如何使用线索？

- 寻找当前结点在中序下的前驱

```
if (current->ltag ==1)
    return current->leftChild
else   //current->ltag == 0
    { q=current->leftChild;
      while(q->rtag==0)
          q=q->rightChild;
      rerturn q ; }
```



5.3 线索二叉树

如何使用线索？

（2）前序线索二叉树，如何找当前结点的前驱和后继：

（3）后序线索二叉树，如何找当前结点的前驱和后继：

参阅教材：P219。 略！

注意：记录了某种线索，显然就可以利用线索进行这种方式的遍历。例如利用中序线索就可以通过找前驱（或后继）实现中序遍历。 也可以进行其他方式的遍历，例如利用中序线索进行前序或后序遍历（当然要麻烦一些！）

参阅教材：P215-216 略！

5.3 线索二叉树

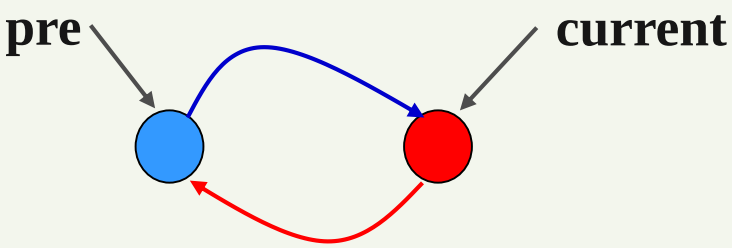
如何建立线索？

[线索化] 建立（记录）线索的过程。

基本思想：

确定遍历方式，在遍历过程中：

- 记住刚刚访问过的结点pre和正在访问的结点current
- pre->rightChild==NULL，则可以记录其后继线索：
  pre->rtag=1; pre->rightChild=current;
- current->leftChild==NULL，则可以记录其前驱线索：
  current->ltag=1; current->leftChild=pre;

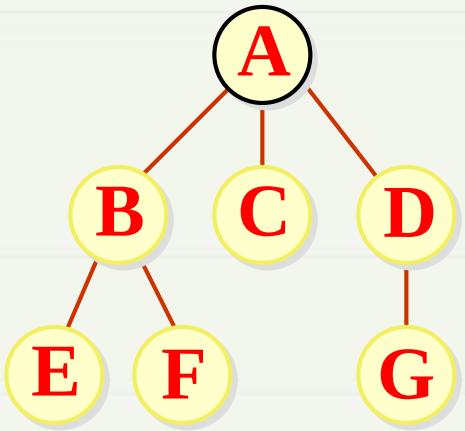


教材：P214 程序5.24

5.4 树、森林与二叉树

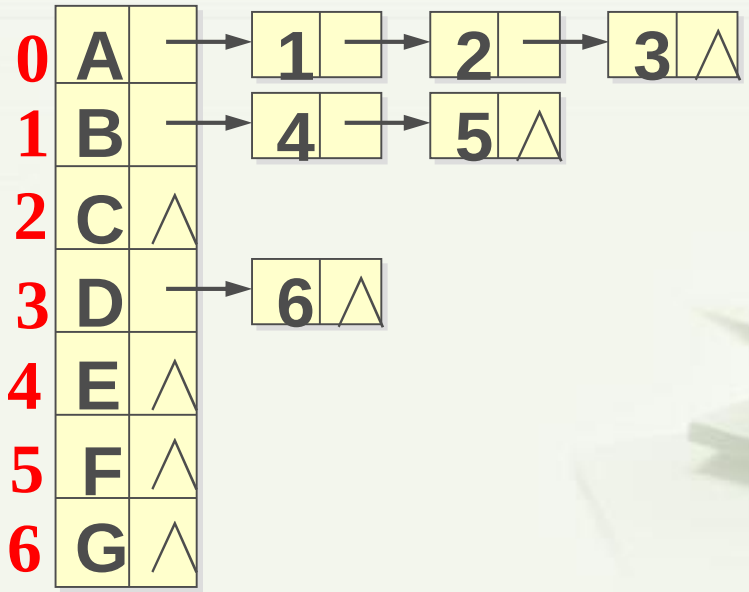
5.4.1 树的存储

1. 双亲存储



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

2. 子女链表存储之一

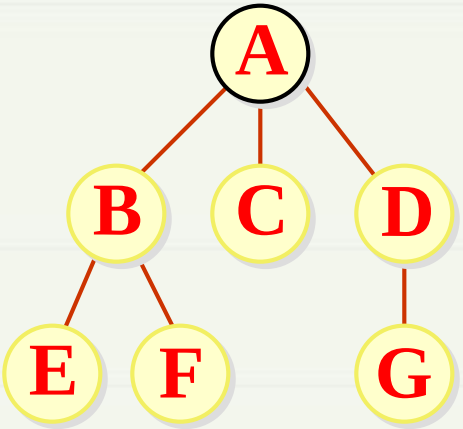


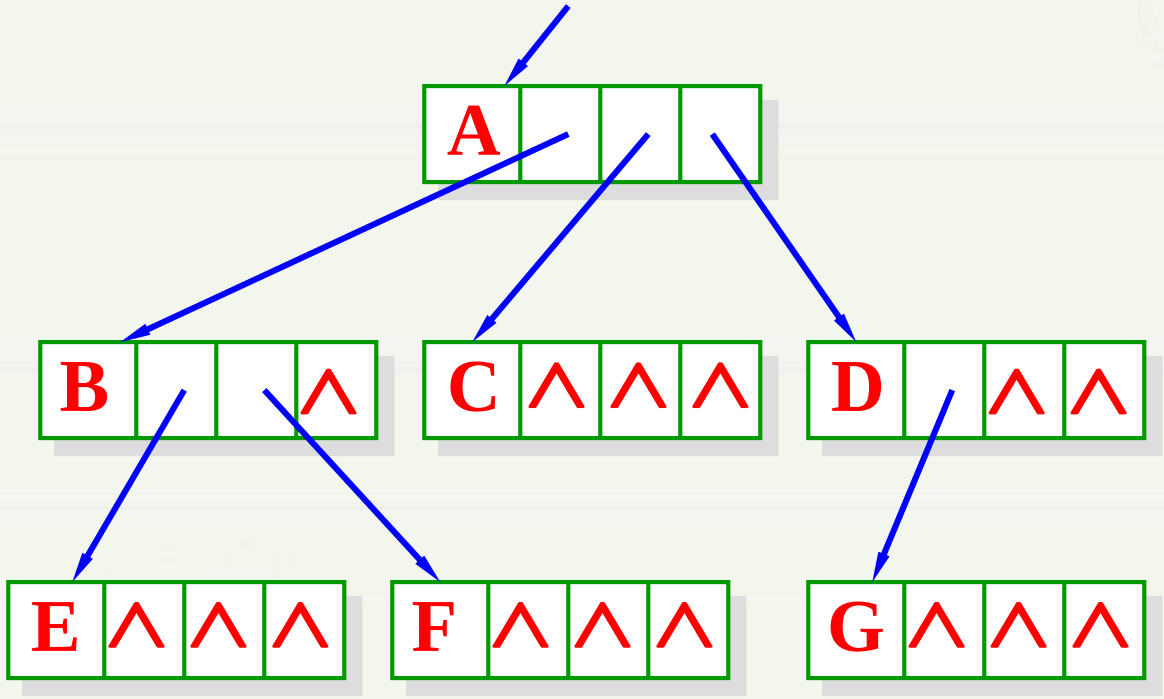
5.4 树、森林与二叉树

5.4.1 树的存储

空指针个数=n*d-(n-1)=(d-1)*n+1

3.子女链表存储之二





data	$child_1$	$child_2$	$child_3$	·····	$child_{di}$

不定长结点

data	$child_1$	$child_2$	$child_3$	·····	$child_d$

定长结点

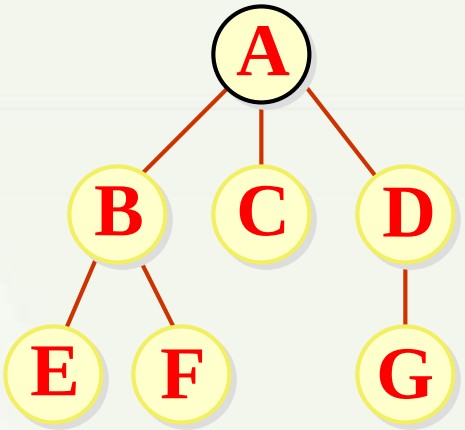
5.4 树、森林与二叉树

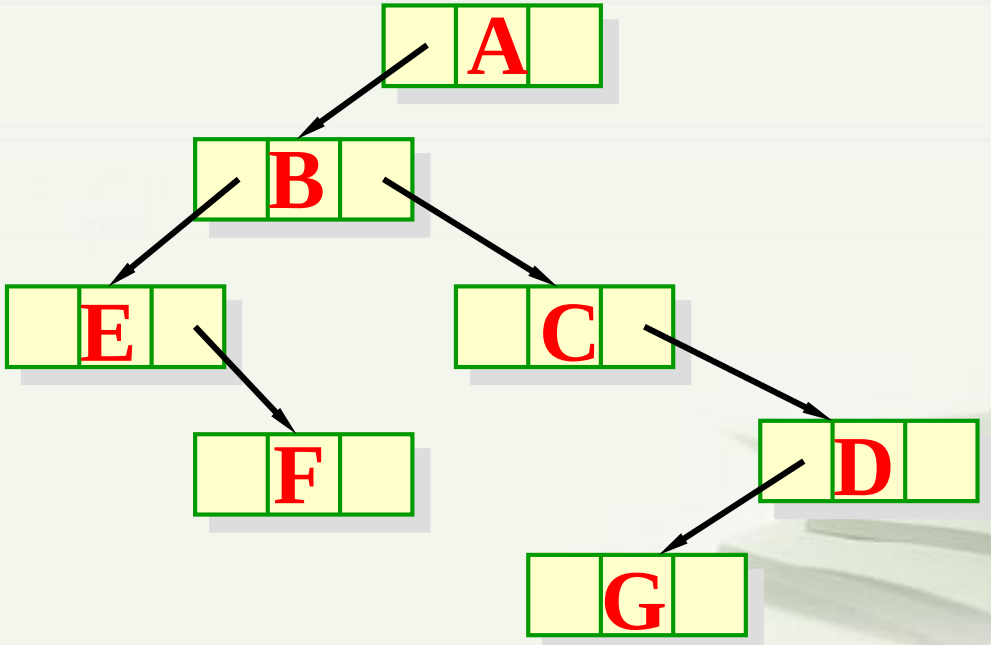
5.4.1 树的存储

4.子女链表存储之三：左孩子右兄弟

存储结点：

data	firstChild	nextSibling





5.4 树、森林与二叉树

5.4.2 树、森林与二叉树的转换

1、树 ⟺ 二叉树

树 ⟹ 二叉树

树的左儿子、右兄弟存储，即：凡是兄弟就连起来（右指针），然后只保留双亲到第一个孩子的分支连线而去掉到其他孩子的分支连线。

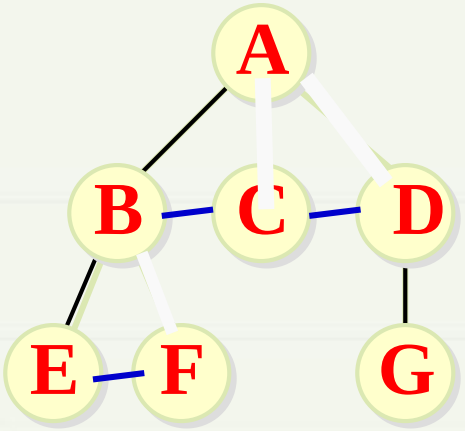
二叉树 ⟹ 树

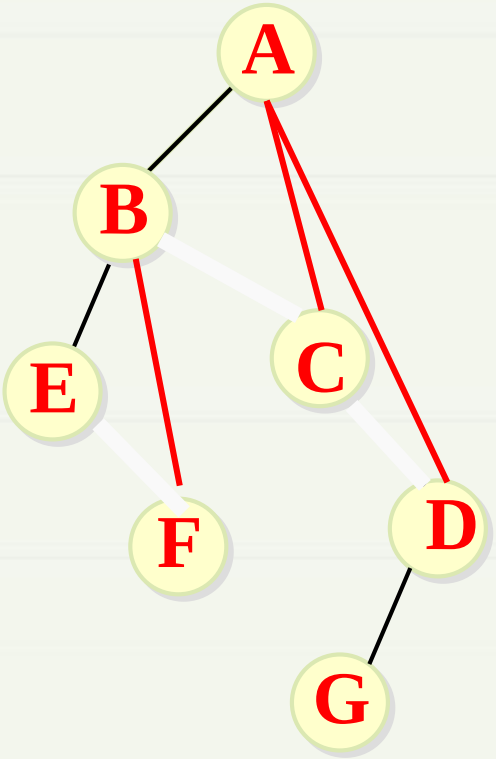
如果一个结点是双亲的左儿子，则将该结点的右儿子，右儿子的右儿子，...，均与其双亲相连，然后去掉双亲到右儿子的分支连线。

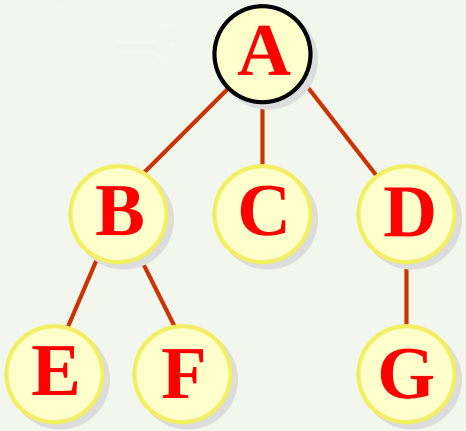
5.4 树、森林与二叉树

5.4.2 树、森林与二叉树的转换

1、树 ⟺ 二叉树







5.4 树、森林与二叉树

5.4.2 树、森林与二叉树的转换

2、森林 ⟺ 二叉树

森林 ⟹ 二叉树

$F=\{T_1,T_2,\dots,T_m\}$，

若m=0，则二叉树为空；

否则，二叉树的根为T_1的根，T_1的子树森林转化为二叉树的左子树，森林$\{T_2,T_3,\dots,T_m\}$转化为二叉树的右子树。（递归）

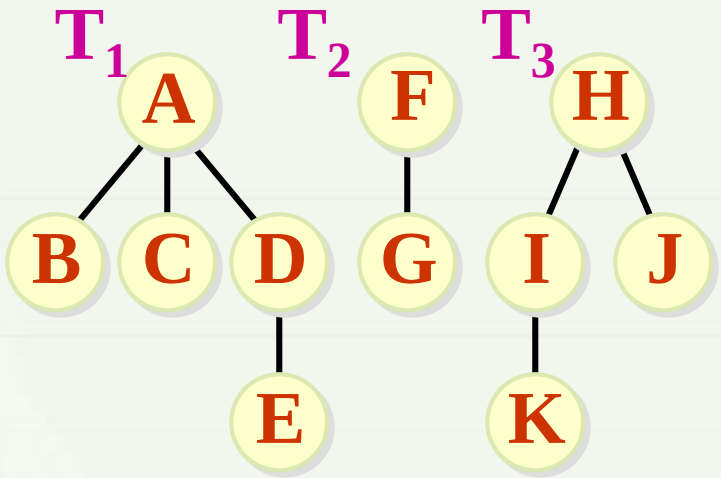
二叉树 ⟹ 森林

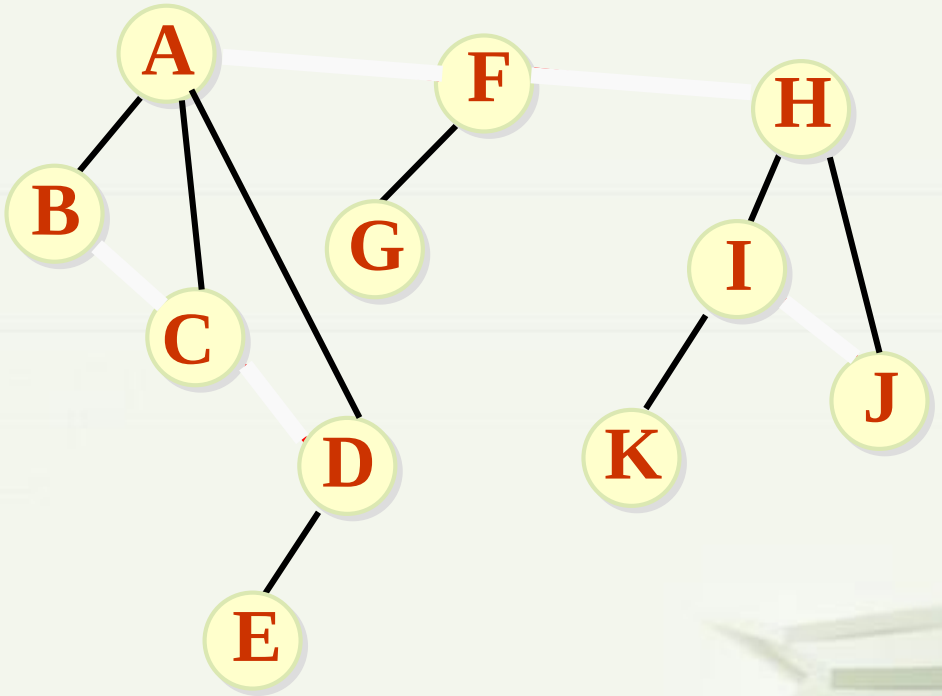
同二叉树还原为树一样！

5.4 树、森林与二叉树

5.4.2 树、森林与二叉树的转换

2、森林 ⟺ 二叉树





5.4 树、森林与二叉树

5.4.3 树、森林的遍历

1.树的广度遍历：略。

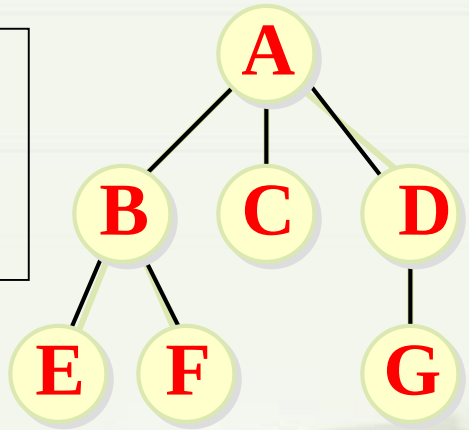
2.树的深度遍历：

◆ 先根次序遍历

当树非空时，

（1） 访问根结点

（2）依次先根遍历根的各棵子树



遍历序列：ABEFCDG

等价于其对应二叉树表示的前序遍历序列！

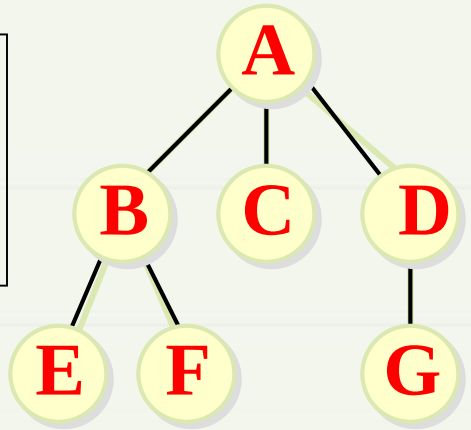
5.4 树、森林与二叉树

5.4.3 树、森林的遍历

2.树的深度遍历：

- 后根次序遍历

当树非空时
（1）依次后根遍历根的各棵子树
（2）访问根结点



遍历序列：EFBCGDA

等价于其对应二叉树表示的中序遍历序列！

5.4 树、森林与二叉树

5.4.3 树、森林的遍历

3.森林的广度遍历：略

4.森林的深度遍历：

- 森林的先根次序遍历

若森林F = Ø，返回；
否则：

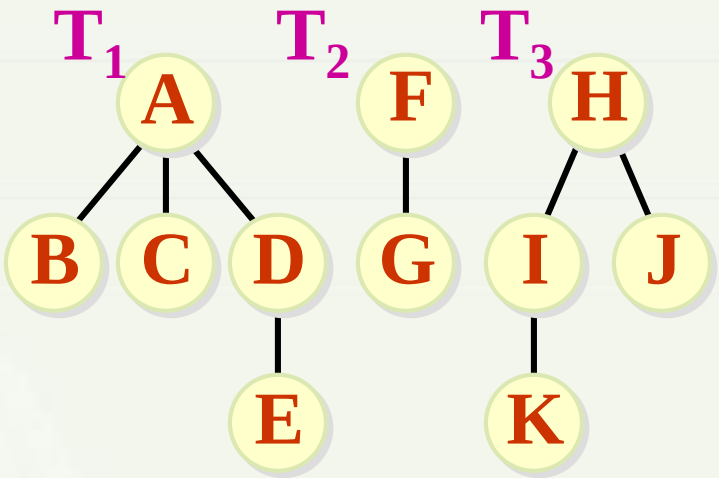
（1）访问森林的第一棵树的根$r1$；
（2）先根遍历森林第一棵树的根的子树森林
$\{T_{11}, \ldots, T_{1k}\}$；
（3）先根遍历森林中除第一棵树外其他树组成的森林
$\{T_2, \ldots, T_m\}$。

5.4 树、森林与二叉树

5.4.3 树、森林的遍历

4.森林的深度遍历：

- 森林的先根次序遍历



先根遍历序列：**ABCDEFGHIKJ**

等价于对应二叉树的前序遍历序列

5.4 树、森林与二叉树

5.4.3 树、森林的遍历

4.森林的深度遍历：

- 森林的后根次序遍历

若森林 $F = \emptyset$，返回；
否则：

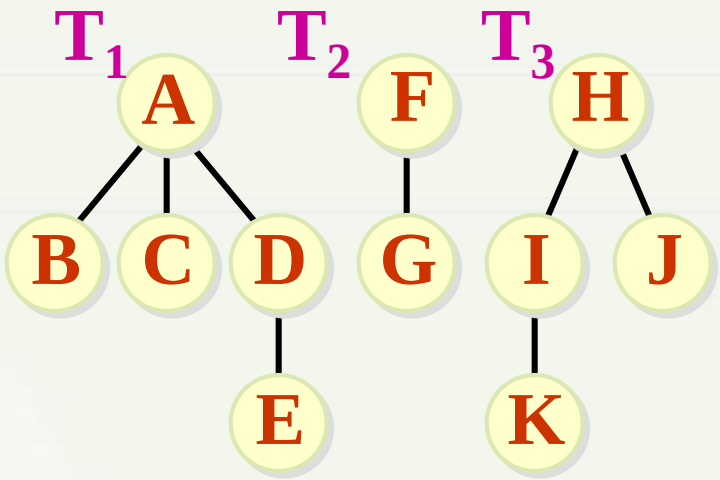
（1）后根遍历森林 F 第一棵树的根结点的子树森林 $\{T_{11}, \ldots, T_{1k}\}$；
（2）访问森林第一棵树的根结点 $r1$；
（3）后根遍历森林中除第一棵树外其他树组成的森林 $\{T_2, \ldots, T_m\}$。

5.4 树、森林与二叉树

5.4.3 树、森林的遍历

4.森林的深度遍历：

- 森林的后根次序遍历



后根遍历序列：**BCEDAGFKIJH**

等价于对应二叉树的中序遍历序列

5.5 特殊二叉树

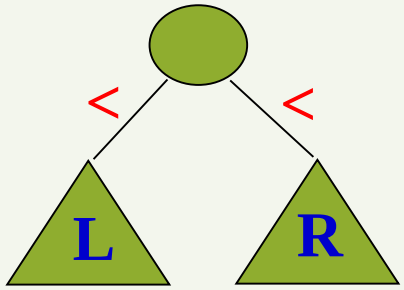
5.5.1 二叉排序树

5.5.1.1 二叉排序树的定义及特点

[二叉排序树 Binary Sort Tree，BST] 二叉树中，任何结点均满足条件：“大于其左子树上的所有结点，小于其右子树上的所有结点（若存在的话）”。又称为**二叉查找树（Binary Search Tree），或二叉搜索树。**

递归定义：若二叉树是空，则是二叉排序树；否则：

- 若左子树不空，它的结点均小于根结点；
- 若右子树不空，它的结点均大于根结点；
- 左、右子树又都是二叉排序树！



5.5 特殊二叉树

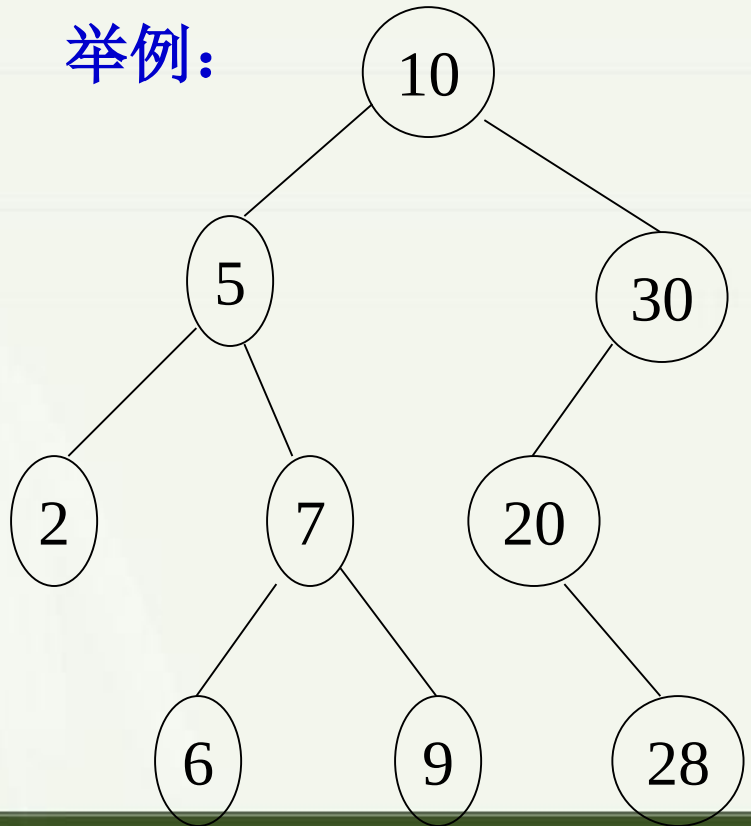
5.5.1 二叉排序树

5.5.1.1 二叉排序树的定义及特点

特点：（1）可以实现类似折半思想的查找，有较高的查找性能

（2）中序遍历二叉排序树，可得到一个有序序列！

举例：



中序遍历序列：

2 5 6 7 9 10 20 28 30

查找元素 7：找到，3次比较

查找元素 18：没找到，3次比较

5.5 特殊二叉树

5.5.1 二叉排序树

5.5.1.2 二叉排序树的构造（创建）

前面我们介绍了一般二叉树的建立，是比较难的，原因是根据结构建立二叉树，如果不考虑结构，仅仅是考虑二叉树满足某种性质，就简单多了。如二叉排序树的建立就是这样。

■ 基本思想：只要将数据元素链到合适的位置（满足性质）。

开始二叉树为空，然后重复在二叉树中插入元素，即：

读入数据元素；

（1）若二叉树为空，则插入到空二叉树中，即该数据元素就是根；显然是二叉排序树。

（2）若二叉树不是空，则与根比较，若比根大则*在右子树中插入*，否则*在左子树中插入*。

5.5 特殊二叉树

5.5.1 二叉排序树

5.5.1.2 二叉排序树的构造（创建）

```
void  Insert (Btreenode *BST ,ElemType  x )
{    if(BST==NULL)
          {     Btreenode    *p= new  Btreenode;
                p->data = x;
                p->leftChild=NULL；  p->rightChild=NULL;
               BST=p;  }
     else if (BST->data >= x )
              Insert ( BST-> leftChild,x); //在左子树中插入
          else
              Insert  (BST->rightChild,x);    //在右子树中插入
 }
```

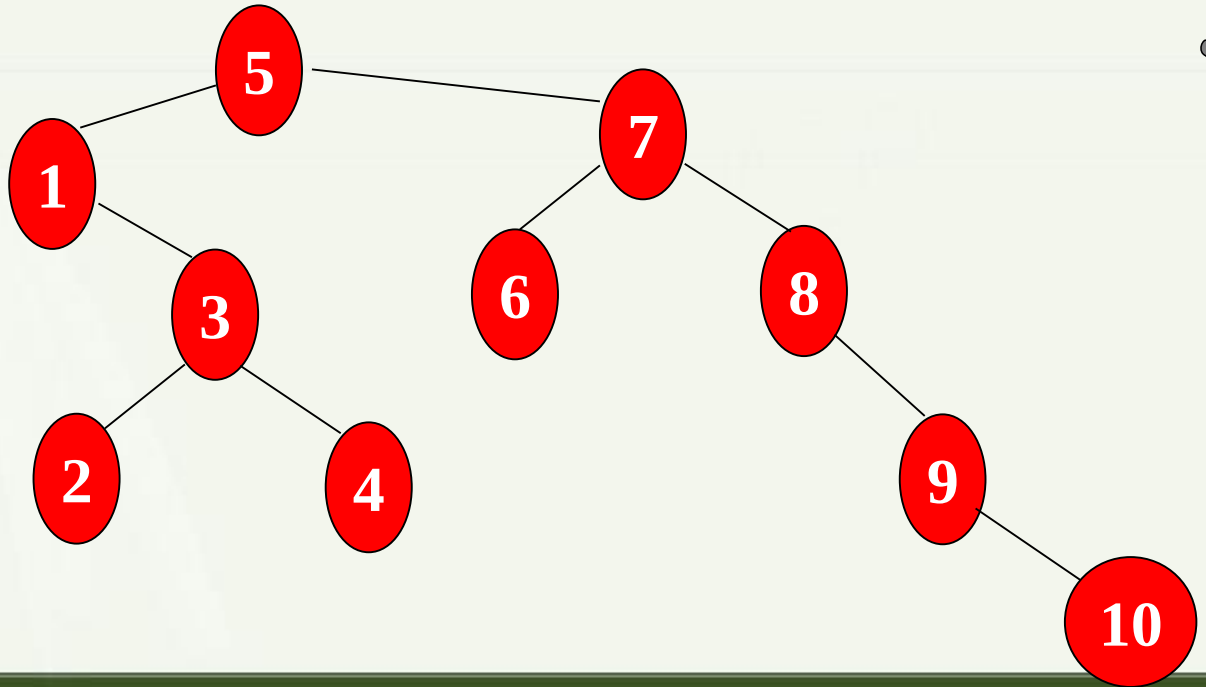
5.5 特殊二叉树

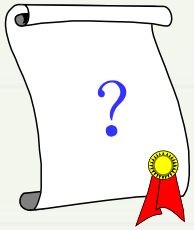
5.5.1 二叉排序树

5.5.1.2 二叉排序树的构造（创建）

例，给定数据元素： 5 1 3 7 8 9 4 10 2 6
请建立二叉排序树

5 1 3 7 8 9 4 10 2 6





元素集合不变但读入顺序改变，得到的树是否一样？

5.5 特殊二叉树

5.5.1 二叉排序树

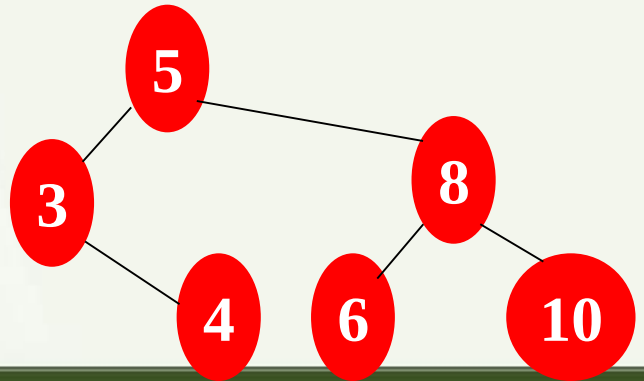
5.5.1.3 二叉排序树的插入删除操作

1. 插入：在二叉排序树中插入一个数据元素x。
   （插入后仍然是二叉排序树）

   容易，同二叉排序树的建立。略！

2. 删除：在二叉排序树中删除元素y。
   （删除后仍然是二叉排序树）

   在保持二叉排序树特性的前提下，用容易删除的结点去替换被删除的结点，然后把容易删除的结点删除即可。



删除元素5：

插入、删除的算法略！

5.5 特殊二叉树

5.5.2 堆

5.5.2.1 堆的定义及特点

在许多应用中（例如优先队列），需要从数据集中挑选具有最小或最大关键码的元素。找最小或最大元素的常用算法的思想是“打擂台”，其时间复杂性为O(n)。那么有没有更好的算法呢？

“堆”是一种特殊的二叉树数据结构，在它上面可以有高效的求最小（或最大）值操作。

5.5 特殊二叉树

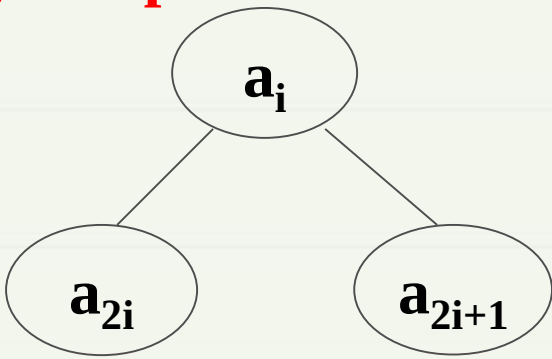
5.5.2 堆

5.5.2.1 堆的定义及特点

[堆 Heap] n个数据元素的序列$a_1,a_2,a_3,\ldots,a_n$，当且仅当满足下列关系时称为堆。又称为二叉堆，Binary Heap

$a_i \le a_{2i}$　$a_i \le a_{2i+1}$ ——最小堆

或者　$a_i \ge a_{2i}$　$a_i \ge a_{2i+1}$ ——最大堆



堆可以用一棵完全二叉树来表示，更直观！

即以n个数据元素的序列$a_1,a_2,a_3,\ldots,a_n$按照从第一层开始，每层从左向右构造出完全二叉树，如果任一结点的元素值都不大于（或不小于）它的左右孩子的元素值，则该完全二叉树就是最小堆（最大堆）。

5.5 特殊二叉树

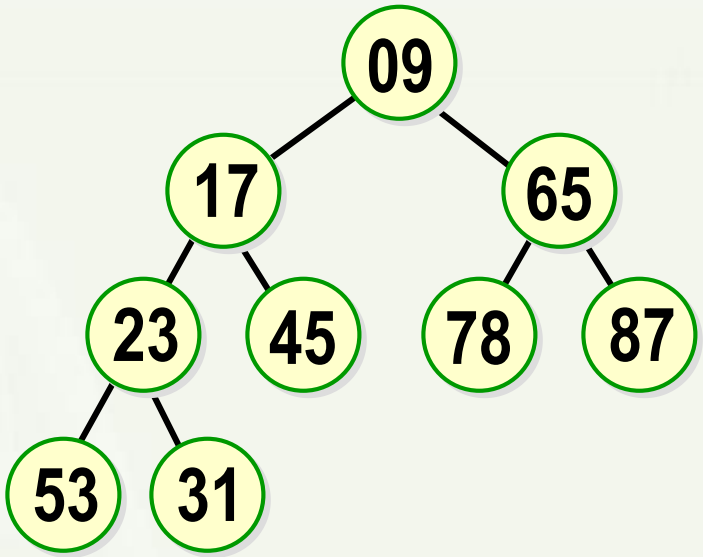
5.5.2 堆

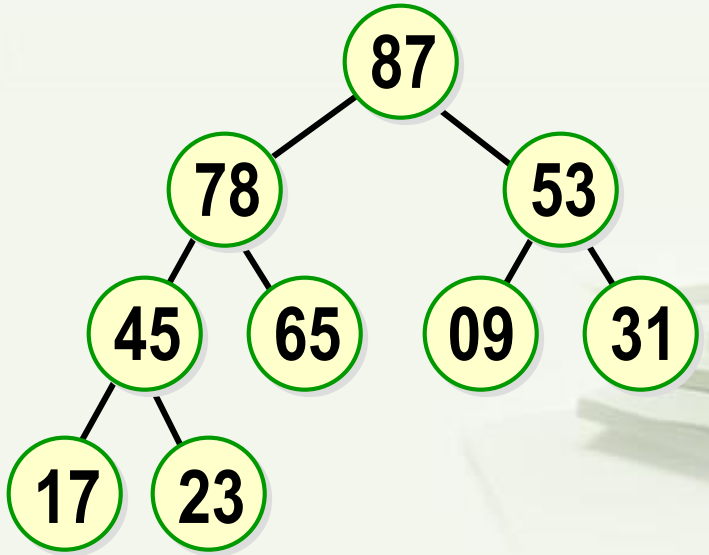
5.5.2.1 堆的定义及特点

特点：

在最小堆中，a_1一定是最小的元素。最大元素可能在哪里？

在最大堆中，a_1一定是最大的元素。最小元素可能在哪里？





5.5 特殊二叉树

5.5.2 堆

5.5.2.1 堆的定义及特点

堆的用途：

（1）高效求最小（或最大）值。如果一个序列（完全二叉树）不是堆，则通过调整把它变成堆，于是就得到了最小（或最大）元素。——在堆顶

效率体现在调整过程上！

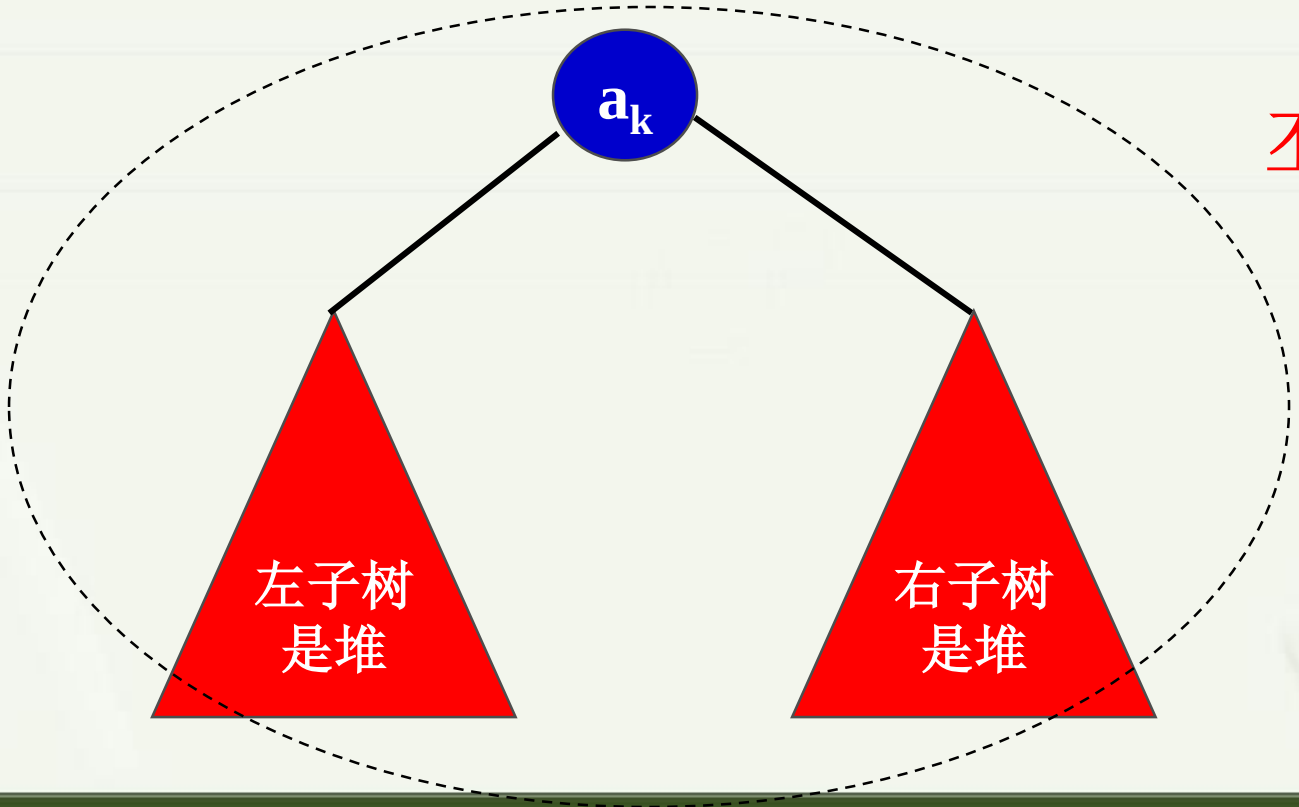
（2）不断地重复求最小（或最大）值，可以实现排序。——堆排序

5.5 特殊二叉树

5.5.3 堆

5.5.2.2 堆的构造（创建）

(1) 假设 $a_{k+1}..a_m$满足堆的性质，即它是堆，而$a_k a_{k+1}..a_m$不是堆，如何调整为堆？

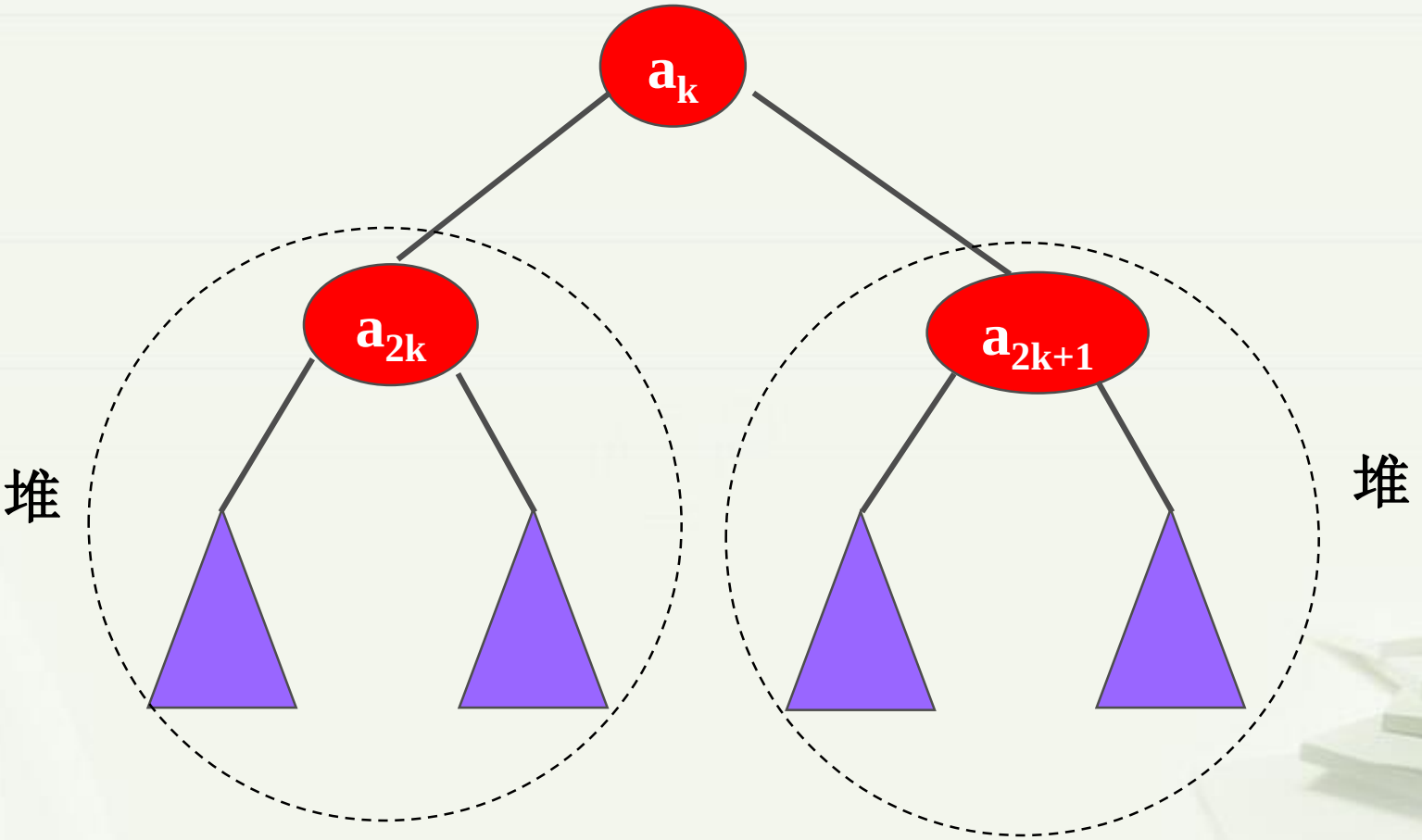


不是堆！

5.5 特殊二叉树

5.5.3 堆

5.5.2.2 堆的构造（创建）



5.5 特殊二叉树

5.5.3 堆

5.5.2.2 堆的构造（创建）

具体地，按“自上向下”的顺序调整：

堆顶元素a_k与左、右子树的根结点(a_{2k} ,a_{2k+1})比较（实际上只要与小根比较即可）：

若a_k比小根小，则是堆（最小堆），结束；

否则，即a_k比小根大，则与小根交换。但可能会导致小根所在的子树的堆性质被破坏，如果被破坏，就用同样的方式进行调整，直到调整的子树已经是堆或为叶子为止。

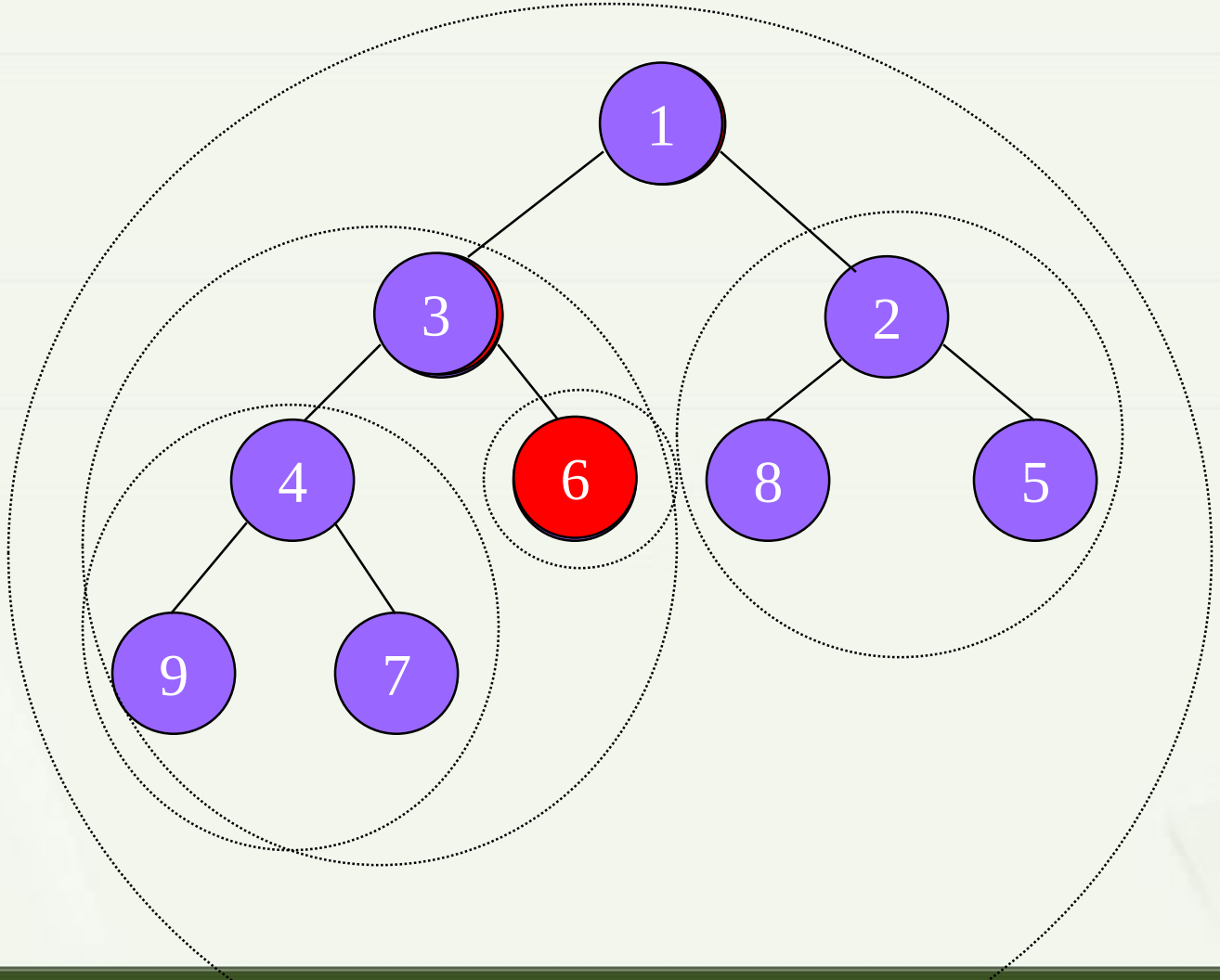
效率：二叉排序树的高度。

教材P238：siftDown

5.5 特殊二叉树

5.5.3 堆

5.5.2.2 堆的构造（创建）



5.5 特殊二叉树

5.5.3 堆

5.5.2.2 堆的构造（创建）

(2) 假设 $a_1..a_n$是任意一个序列，如何调整为堆呢？

重复调用siftDown，自下向上逐步调整为最小堆：

从最底层开始，显然，如果二叉树只有树根，则一定是堆！

而，从第n个至第⌊n/2⌋+1个元素都是叶子（即是只有树根的二叉树），即都是堆。

但是，以第⌊n/2⌋个元素为根的二叉树不一定是堆，但是这种情况是我们前面介绍的（1）情况，

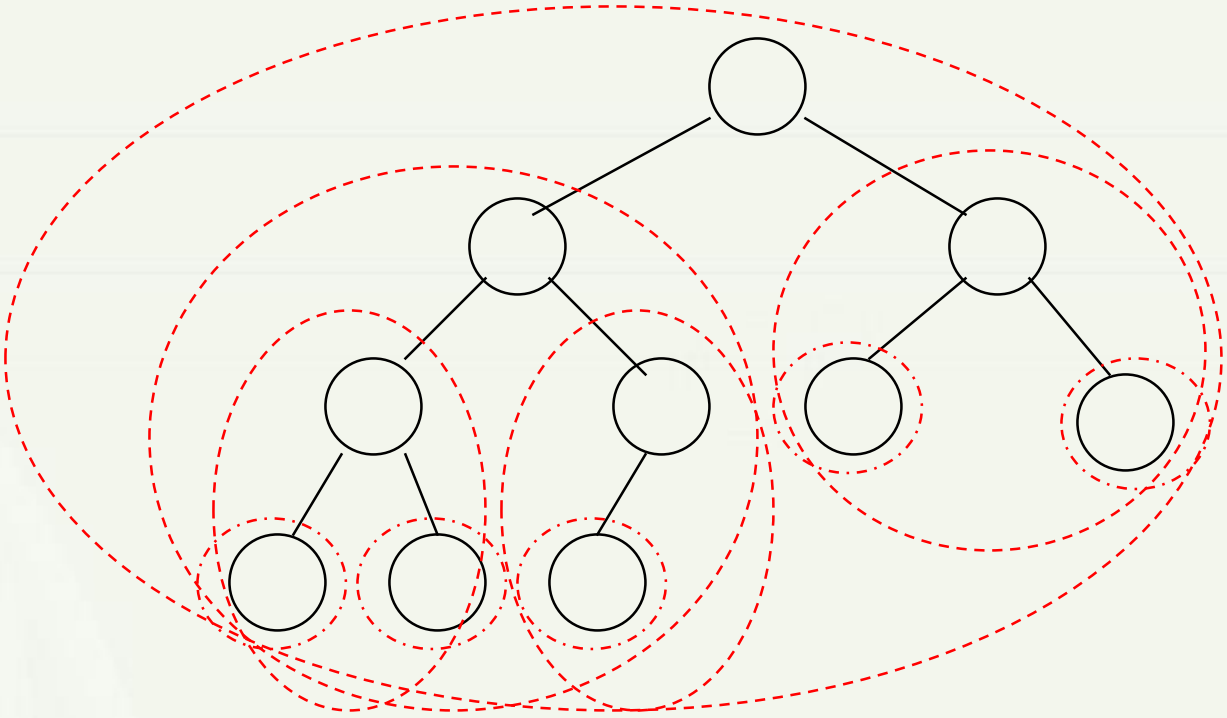
使用siftDown方法很容易调整为堆。同样，继续向前看第⌊n/2⌋-1，第⌊n/2⌋-2，...3,2,1个元素为根的二叉树，它们都是前面的（1）情况。用同样的方法进行调整。

5.5 特殊二叉树

5.5.3 堆

5.5.2.2 堆的构造（创建）

(2) 假设 $a_1..a_n$是任意一个序列，如何调整为堆呢？



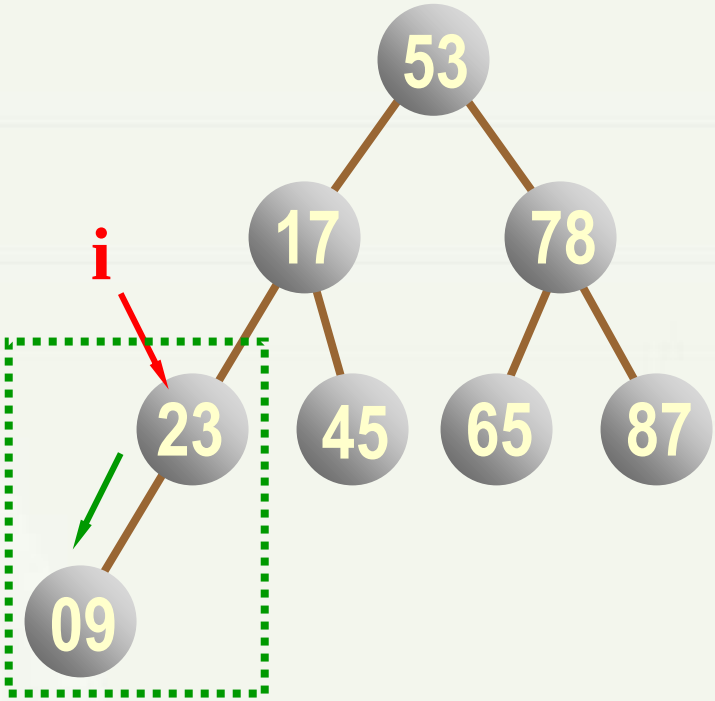
这样得到的堆称为初始堆。

5.5 特殊二叉树

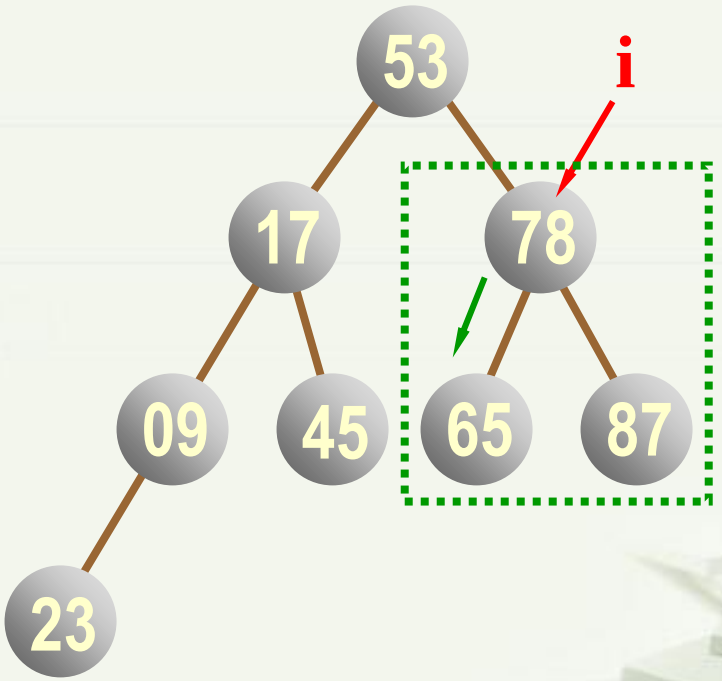
5.5.3 堆

5.5.2.2 堆的构造（创建）

(2) 假设 $a_1..a_n$是任意一个序列，如何调整为堆呢？



currentPos = i = 4

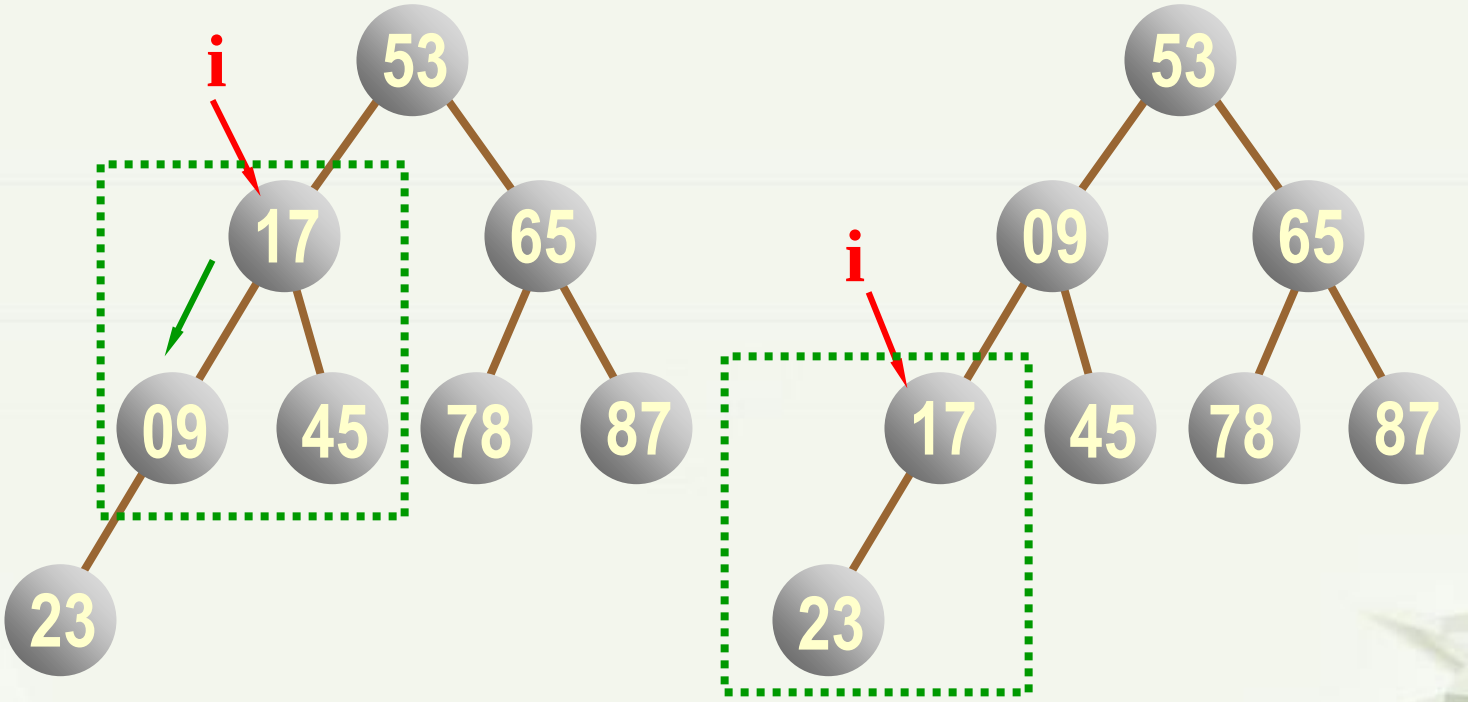


currentPos = i = 3

5.5 特殊二叉树

5.5.3 堆

5.5.2.2 堆的构造（创建）

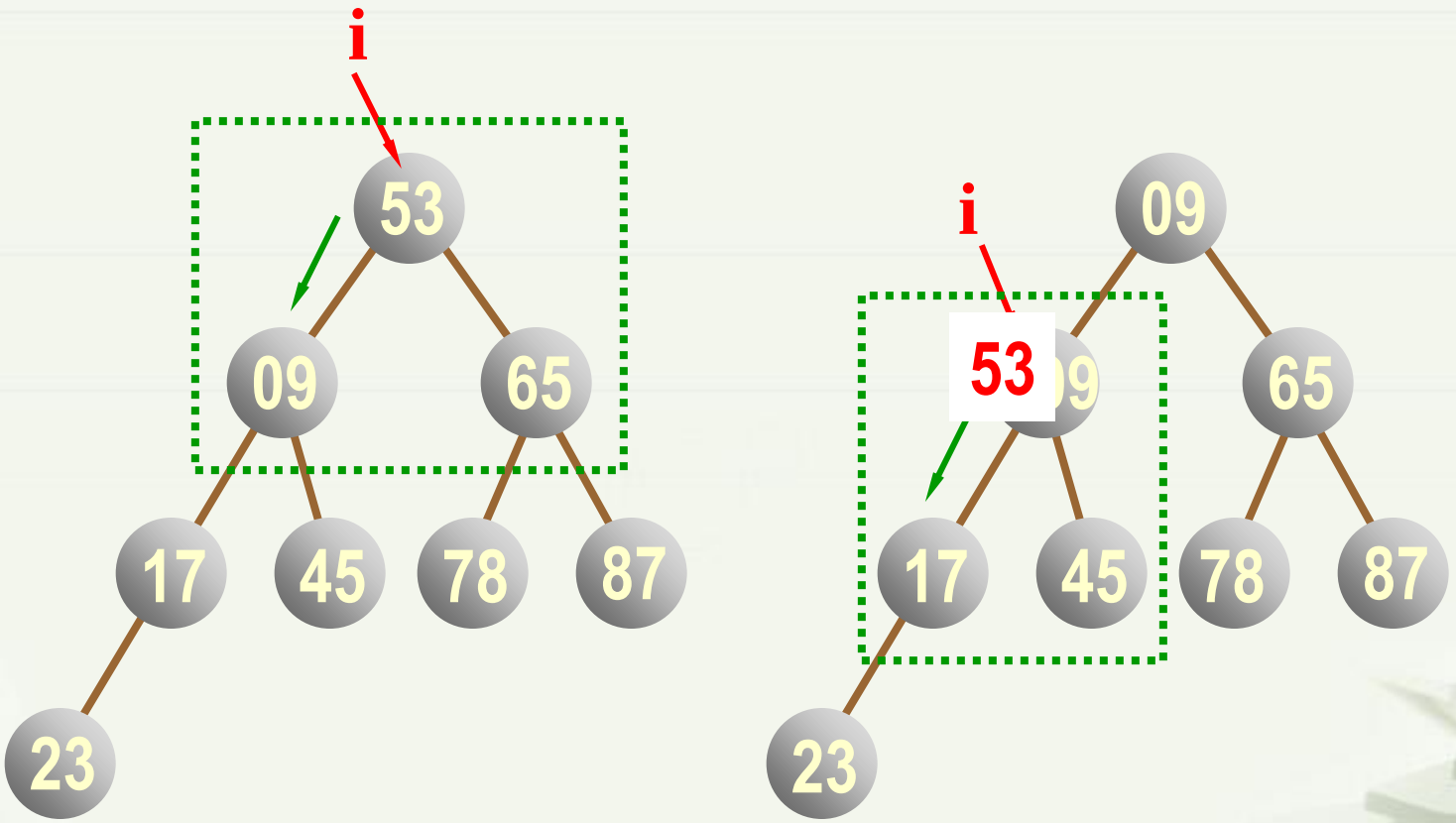


currentPos = i = 2

5.5 特殊二叉树

5.5.3 堆

5.5.2.2 堆的构造（创建）

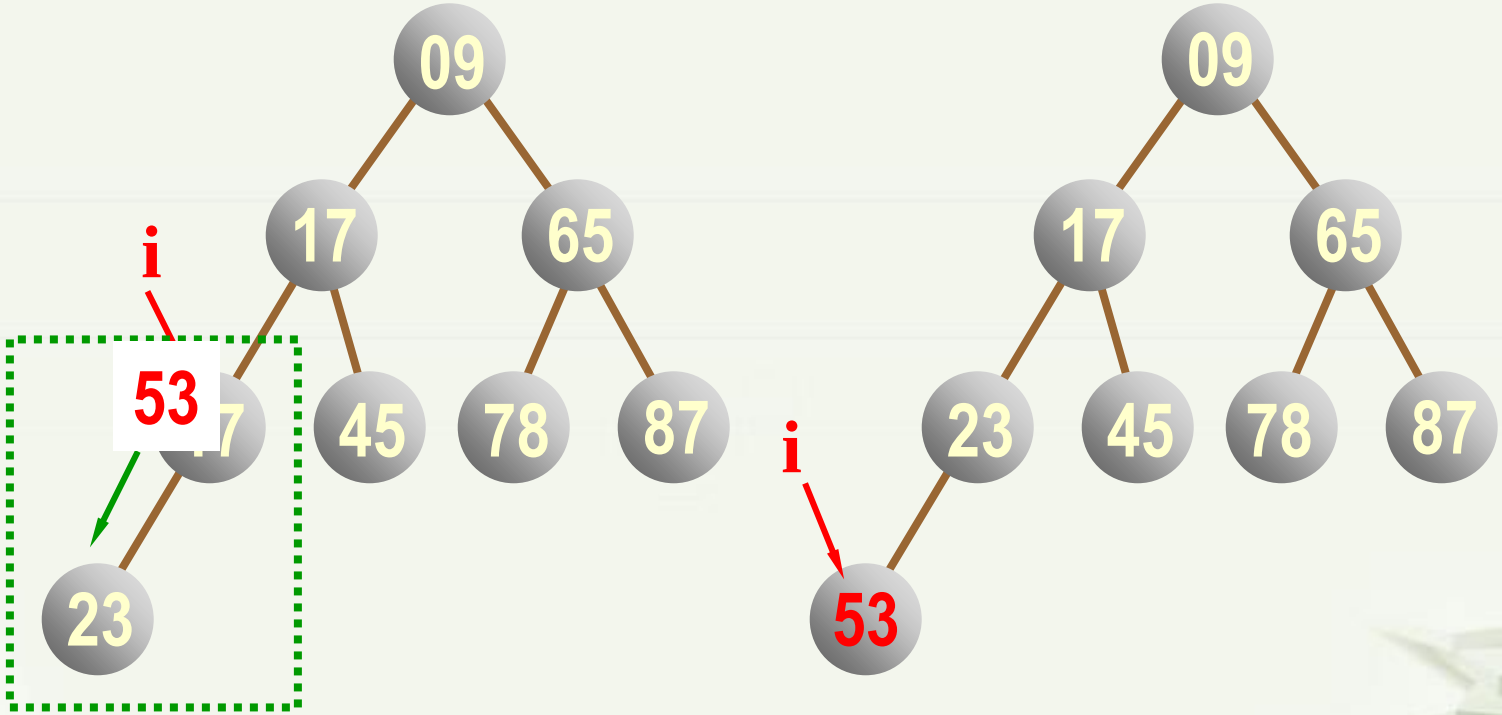


currentPos = i = 1

5.5 特殊二叉树

5.5.3 堆

5.5.2.2 堆的构造（创建）



5.5 特殊二叉树

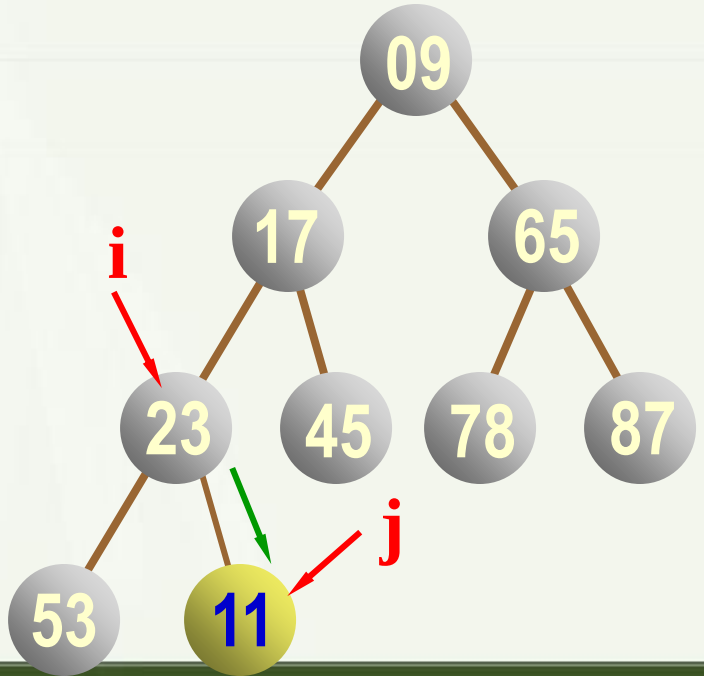
5.5.3 堆

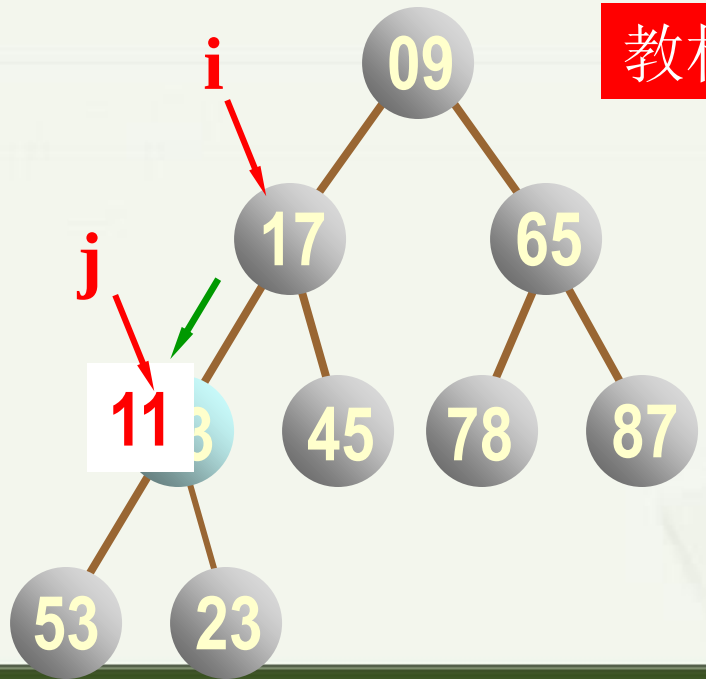
5.5.2.3 堆的插入、删除

(1) 在堆中插入： 插入到序列的最后（完全二叉树最后一层的最后一个元素后面）

处理：插入元素后，可能会破坏其堆（不是堆了！），则沿父结点路径“自下向上”调整，直到满足是堆为止。

教材P239：siftUp



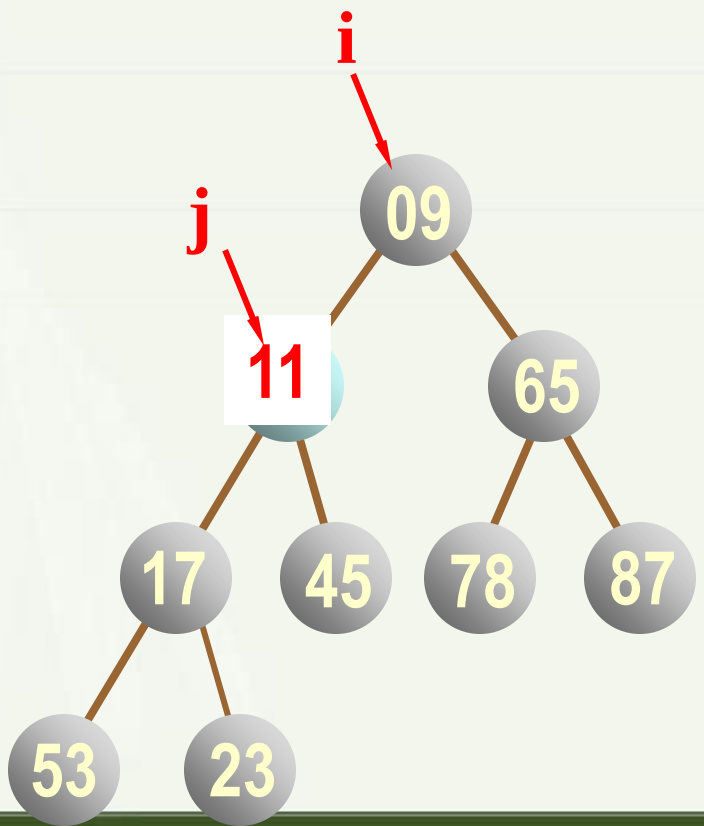


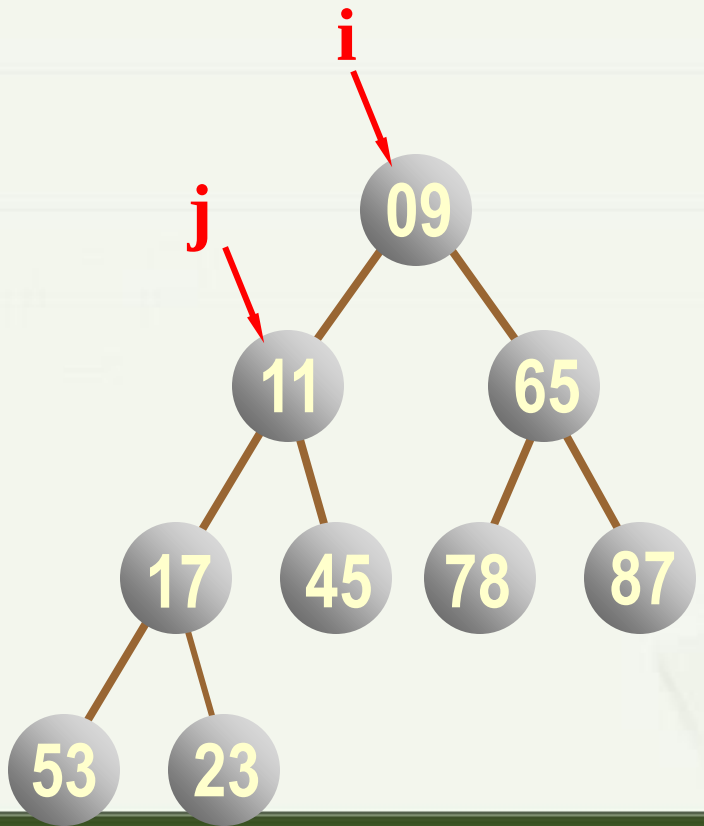
5.5 特殊二叉树

5.5.3 堆

5.5.2.3 堆的插入、删除

(1) 在堆中插入： 插入到序列的最后（完全二叉树最后一层的最后一个元素后面）





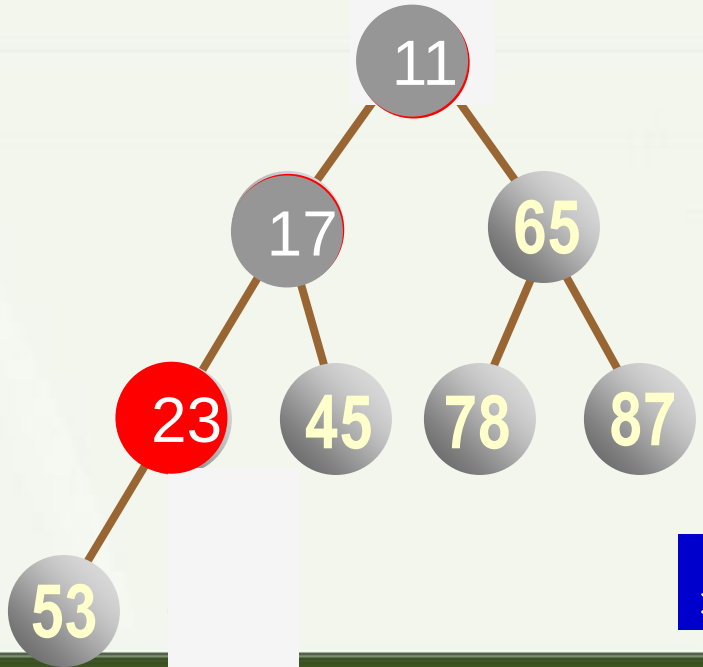
5.5 特殊二叉树

5.5.3 堆

5.5.2.3 堆的插入、删除

(2) 删除堆顶元素： 一般是用最后一个元素来替换堆顶元素(a_1)。

处理：替换后，可能会破坏堆。如果破坏了堆，可以按照前面“自上向下”方式进行调整。siftDown

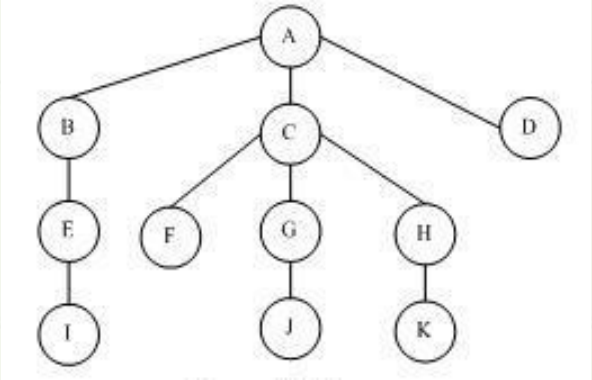


具体算法参阅教材：P238-239。略！

上节课内容回顾

■ 树、森林与二叉树的转换

- 树与二叉树的转换
- 森林与二叉树转换

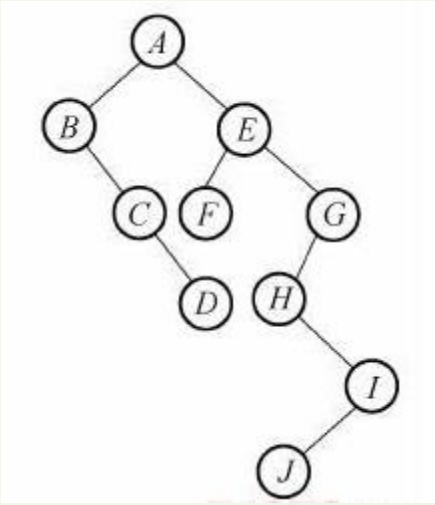


(a)

练习：

（1）将树(a) 转化为二叉树

（2）将二叉树(b)还原为树或森林



(b)

上节课内容回顾

■ 特殊二叉树之 二叉排序树

- 定义
- 特点
- 创建（构造）
- 用途

练习：

（1）假设数据元素为整数，输入元素顺序为5,7,3,4,1,8,6,2，建立二叉排序树。假设输入元素顺序为1,2,3,4,5,6,7,8，二叉排序树又是什么样子？

（2）分别计算上面两棵二叉排序树查找时的平均比较次数。

上节课内容回顾

■ 特殊二叉树之 二叉堆（堆二叉树）

- 定义
- 特点
- 调整（创建）
- 用途

练习：

已知有元素序列为25,13,8,45,34,18,30,50,28。完成：

（1）它是不是堆？大堆还是小堆？

（2）如果不是堆，请调整成堆（指出你调整成堆的类型）

5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.1 最优二叉树的定义及特点

基本概念：

路径： 结点序列 $n_1\ n_2\ \ldots\ n_k$ 满足 n_i 是 n_{i+1} 的双亲。

路径长度： 路径上的分支数。 $l=k-1$

扩充二叉树： 在一般二叉树中，将原来的每个空指针都指向一个特殊的结点——外结点（原来二叉树中的结点称为内结点），这样的二叉树称为扩充二叉树（真二叉树）。

特点： 在扩充二叉树中没有度为1的结点。

外结点个数 S = 内结点个数 n +1

5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.1 最优二叉树的定义及特点

树的内路径长度：从根结点到各个内结点的路径长度之和。

$$I=\sum_{i=1}^{n} l_i$$

I 是二叉树的内路径长度
l_i 是一个内结点的路径长度

树的外路径长度：从根结点到各个外结点的路径长度之和。

$$E=\sum_{j=1}^{n+1} h_j$$

E 是二叉树的外路径长度
h_j 是一个外结点的路径长度

$$E = I + 2n$$

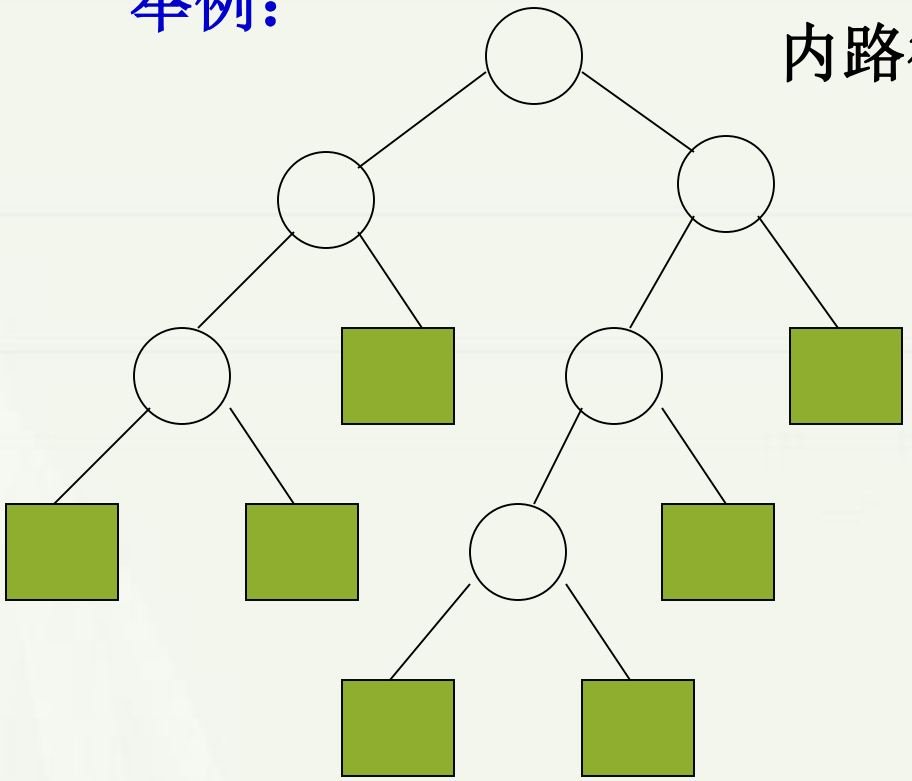
可以用归纳法证明：略！

5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.1 最优二叉树的定义及特点

举例：



内路径长度 I=0+2*1+2*2+1*3=9

E=21=9+2*6=I+2n

外路径长度 E=2*2+3*3+2*4=21

5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.1 最优二叉树的定义及特点

结点的权：有时为了表示某种含义，赋予结点一个数值。

结点加权路径长度：从根结点到该结点的路径长度与权值之积， 即 $w_i * l_i$ 。

扩充二叉树的加权路径长度(Weighted Path Length, WPL)：假设给扩充二叉树的外结点（扩充后的叶子）赋予权值，则二叉树中所有加权结点的加权路径长度之和称为扩充二叉树的加权路径长度。

$$\mathbf{WPL}=\sum_{k=1}^{n} w_k l_k$$

- n是外结点个数
- w_k是外结点的权值
- l_k是外结点的路径长度

5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.1 最优二叉树的定义及特点

? 树的路径长度最小的是什么树？

最优二叉树(Huffman树)： 假设有 n 个数据元素，它们的权值为 $w_1,w_2,...,w_n$，以它们为叶子构造具有 n 个叶子的（真）二叉树（可以构造很多棵）。加权路径长度最小的二叉树称为 最优二叉树（Huffman树）。

特点：权值大的应该尽可能靠近根！

? 这样的二叉树具有多少个结点？：

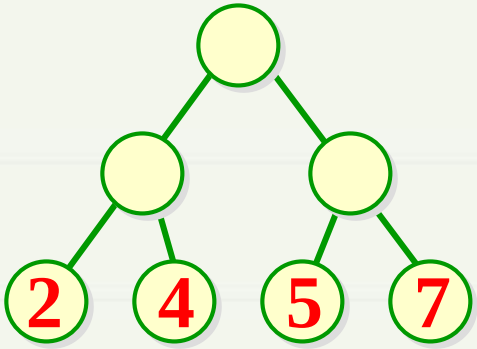
2n-1=n个外结点+n-1 个内结点

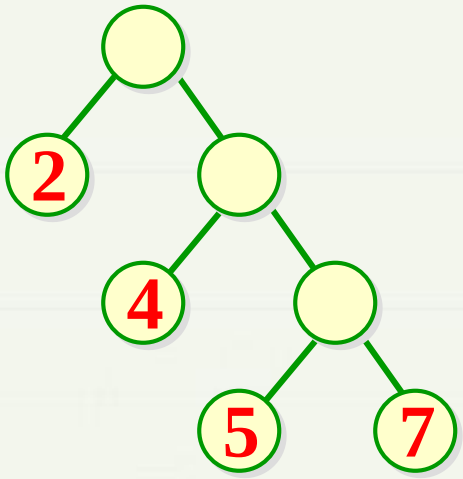
5.5 特殊二叉树

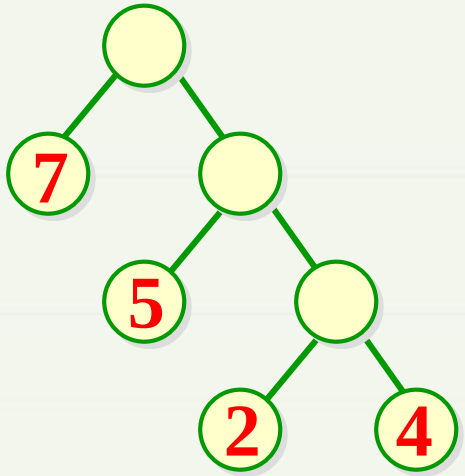
5.5.3 最优二叉树

5.5.3.1 最优二叉树的定义及特点

例如，有4个外结点（叶子），权值为2，4，5，7







WPL=2*2+
4*2+
5*2+
7*2=36

WPL=2*1+
4*2+
5*3+
7*3=46

WPL=7*1+
5*2+
2*3+
4*3=35

5.5 特殊二叉树

5.5.3 最优二叉树

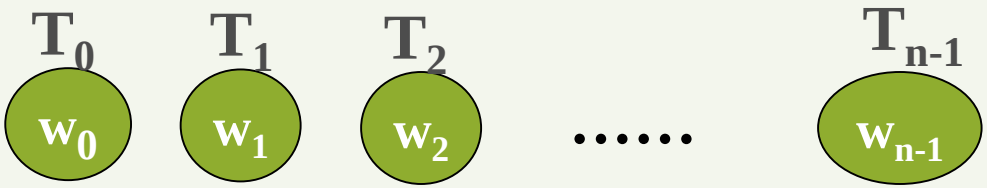
5.5.3.2 最优二叉树的构造（创建）

1. 基本思想：

选择权值小的叶子，让它离根的距离尽可能远（贪心算法）。

2. 算法描述

(1)由给定 n 个权值 $\{w_0, w_1, w_2, \ldots, w_{n-1}\}$，构造 具有 n 棵扩充二叉树的森林 $F = \{T_0, T_1, T_2, \ldots, T_{n-1}\}$，其中每棵扩充二叉树 T_i 只有一个带权值 w_i 的根结点, 其左、右子树均为空。



5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.2 最优二叉树的构造（创建）

2. 算法描述

(2)重复以下步骤, 直到 F 中仅剩一棵树为止:

- 在 F 中选取两棵根结点的权值最小的扩充二叉树， 做为左、右子树构造一棵新的二叉树。置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和。
- 在 F 中删去这两棵二叉树。
- 把新的二叉树加入 F。

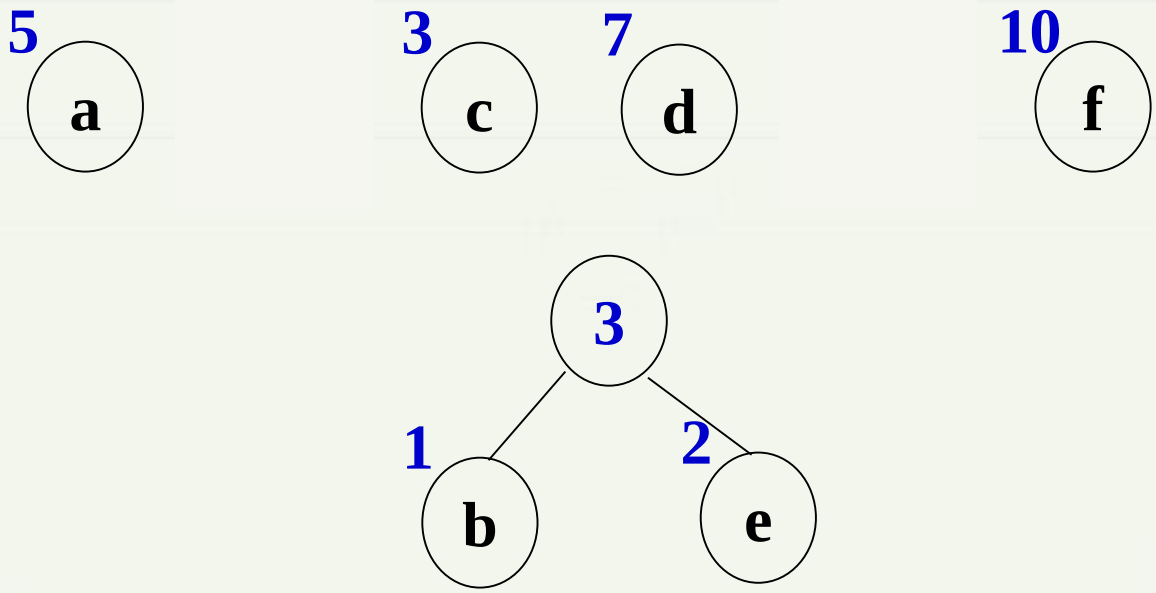
? 共重复多少次？ **n-1**

5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.2 最优二叉树的构造（创建）

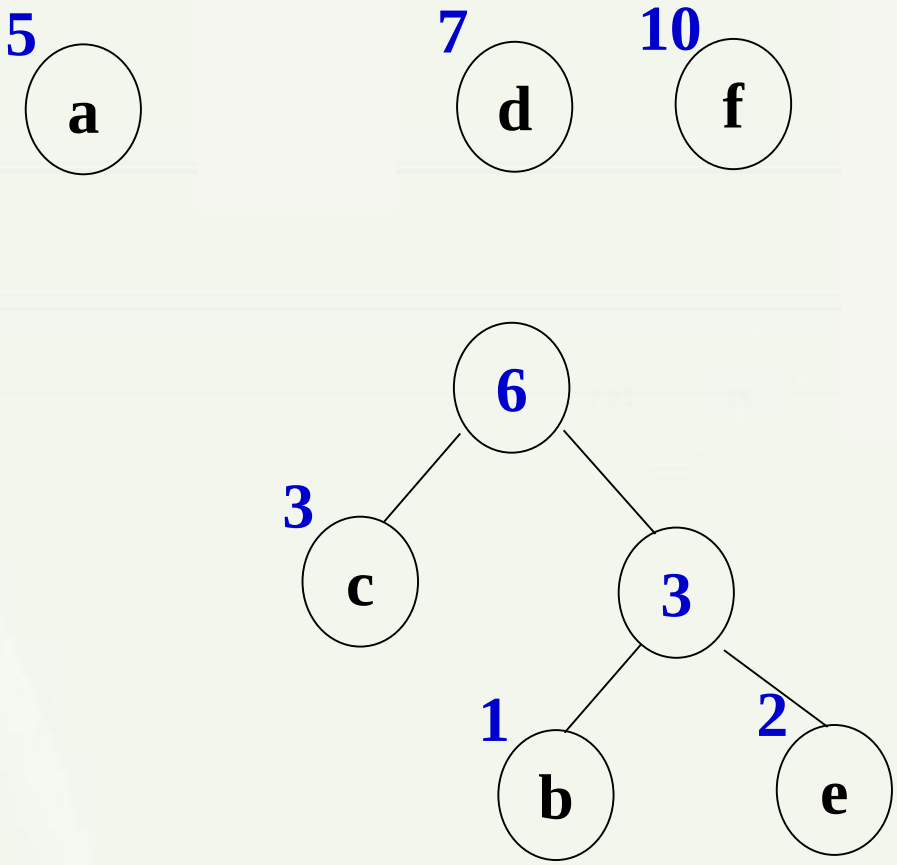
举例：有a,b,c,d,e,f 6个数据元素，它们的权值分别为5,1,3,7,2,10, 构造最优二叉树（哈夫曼树）



5.5 特殊二叉树

5.5.3 最优二叉树

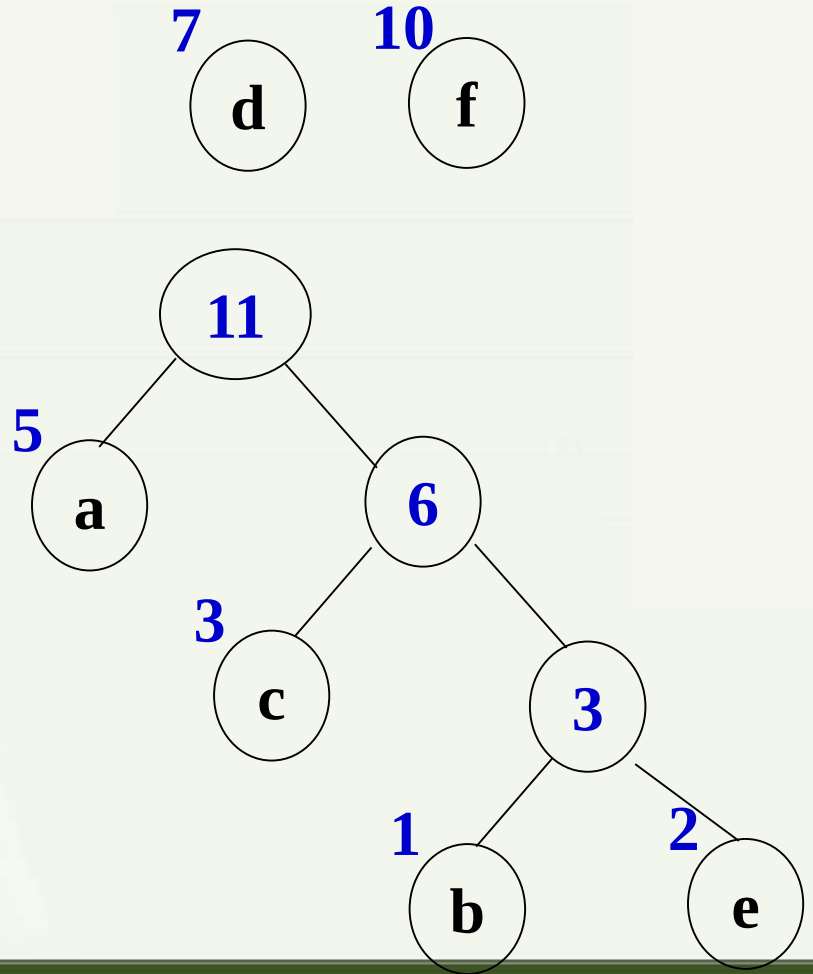
5.5.3.2 最优二叉树的构造（创建）



5.5 特殊二叉树

5.5.3 最优二叉树

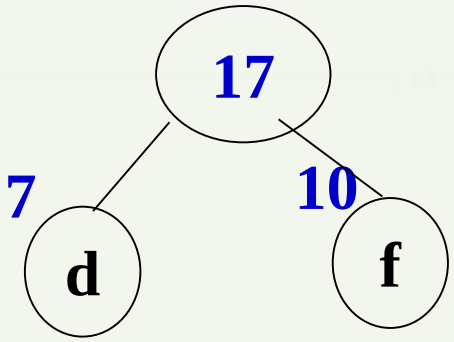
5.5.3.2 最优二叉树的构造（创建）

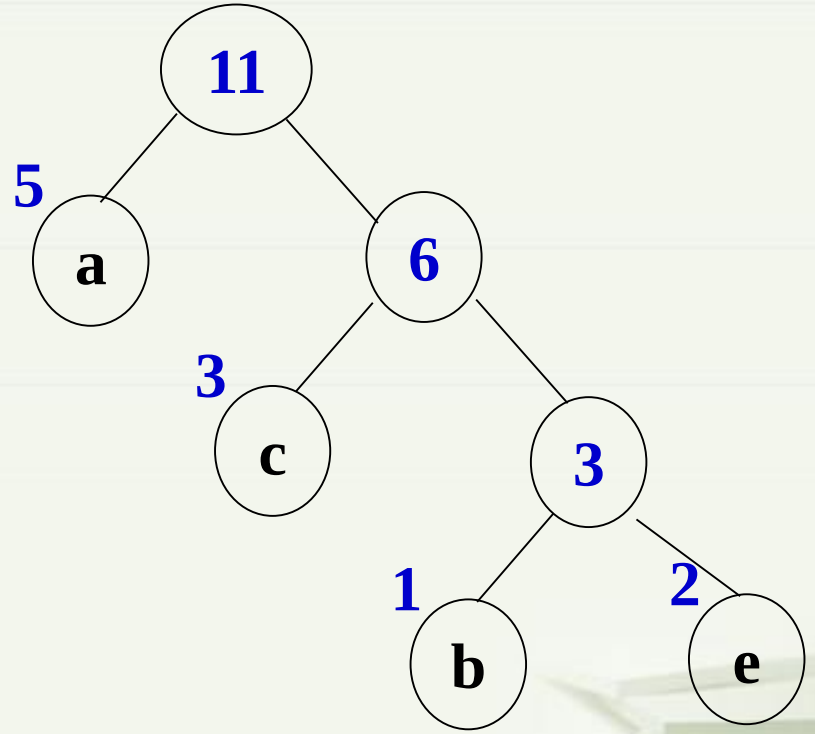


5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.2 最优二叉树的构造（创建）

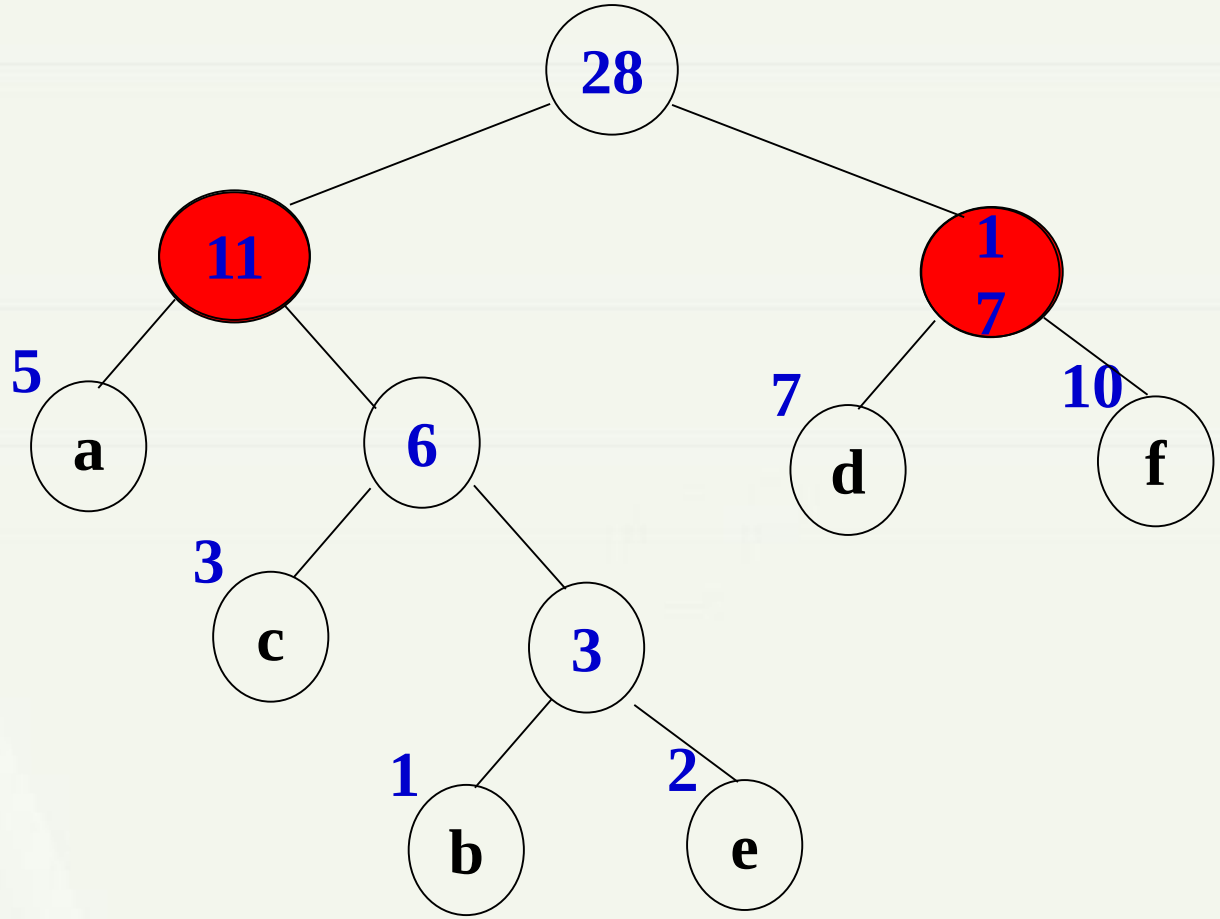




5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.2 最优二叉树的构造（创建）



问：这种方法构造最优二叉树唯一吗？为什么？

权值可能相同；
左、右子树互换；

5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.2 最优二叉树的构造（创建）

3. 算法实现

首先为二叉树（森林）选择存储结构： 三叉链式

data	weight	parent	leftChild	rigthChild

（1）parent为空，则为树根；

（2）每次循环，挑选两棵根的权值最小的二叉树first,second作为左右子树，构造一棵新的二叉树，即生成一个新的根结点p。

```
new(p);
p->weight=first->weight+second->weight;
p->leftChild=first; p->rightChild=second;
first->parent=p;second->parent=p;
```

5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.2 最优二叉树的构造（创建）

3. 算法实现

```
typedef 数据元素类型 ElemType;
struct HuffmanNode  //树结点类的定义
{     ElemType data;   //数据元素
      float weight;        //权值
      HuffmanNode *parent，  *leftChild, *rightChild; //孩子及双亲指针
      HuffmanNode () : data(0),Parent(NULL), leftChild(NULL),
rightChild(NULL) { }      //构造函数
      HuffmanNode (ElemType elem, HuffmanNode *pr = NULL,
        HuffmanNode *left = NULL, HuffmanNode*right = NULL)
           : data (elem), parent (pr), leftChild (left),
             rightChild (right) { } //构造函数
};
```

5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.2 最优二叉树的构造（创建）

3. 算法实现

```
class HuffmanTree  //Huffman树类定义
 {  public:
       HuffmanTree (ElemType elem[],float w[], int n);          //构造函数
       ~HuffmanTree() { deleteTree(root);}    //析构函数
    protected:
      HuffmanNode *root;                        //树的根
      void deleteTree (HuffmanNode *t);        //删除以 t 为根的子树
      void mergeTree (HuffmanNode & ht1, HuffmanNode& ht2,
        HuffmanNode*& parent);
};
```

5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.2 最优二叉树的构造（创建）

3. 算法实现

```
HuffmanTree::HuffmanTree (ElemType elem[],float w[], int n)
//给出 n 个数据元素elem[0] ～ elem[n-1]及权值w[0]～w[n-1] 构造Huffman树
{   minHeap hp;   //使用最小堆存放森林
    HuffmanNode*parent, first, second;
    HuffmanNode*NodeList = new HuffmanNode[n];  //森林中树的根指针数组
    for (int i = 0; i < n; i++)   //初始化森林
   {        NodeList[i].data = elem[i];
            NodeList[i].weight=w[i];
            NodeList[i].leftChild = NULL;
            NodeList[i].rightChild = NULL;
            NodeList[i].parent = NULL;
            hp.Insert(NodeList[i]);     //插入最小堆中
    }
```

5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.2 最优二叉树的构造（创建）

3. 算法实现

```
    //重复n-1趟, 建Huffman树
    for (i = 0; i < n−1; i++)
    {    hp.Remove (first);              //根权值最小的树
         hp.Remove (second);             //根权值次小的树
         mergeTree (first, second, parent);    //合并
         hp.Insert (*parent);                  //重新插入堆中
         root = parent;                  //建立根结点
     }
};
```

5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.2 最优二叉树的构造（创建）

3. 算法实现

```
//合并两棵二叉树
void HuffmanTree::mergeTree (HuffmanNode & bt1,
                             HuffmanNode & bt2,  HuffmanNode*& parent)
{
    parent = new HuffmanNode;    //生成一个结点
    parent->leftChild = &bt1;      //链接左子树
    parent->rightChild = &bt2;    //链接右子树
    parent->weight=bt1.root->weight+bt2.root->weight; //根的权值
    bt1->parent = parent;  // 子树的双亲
    bt2->parent = parent;  // 子树的双亲
};
```

参见教材：P242-243

5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.3 最优二叉树的应用——Huffman编码

早期，电报是进行快速、远距离通讯的重要手段！

实现电报通讯的两个要素：

- 一套编码，可以将电文编码为待发送的电文，还可以把收到的电文译码为原电文。
- 发送的电文应尽可能短，节省占用线路费用。

5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.3 最优二叉树的应用——Huffman编码

编码的原则：

首先，译码要唯一，即对字符进行编码后，能够唯一地翻译成原来的字符。

其次，各个字符的编码要尽可能短，只有这样才能使编码后最短。

编码的方法：

定长编码方法—特点：易达到译码唯一，但长度较大；

ASCII编码是定长编码

不定长编码方法—特点：长度较短，但达到译码唯一较难。

5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.3 最优二叉树的应用——Huffman编码

1、哈夫曼编码的方法——不定长编码方式

[前缀码]：任何一个字符的编码都不是另外字符编码的前缀。

构造方法：用被编码的字符作为叶子，构造二叉树，然后在二叉树的左分支上标“0”，右分支标“1”（或反过来），每个字符的编码就是从根到该字符叶子所经路径上的0、1序列。

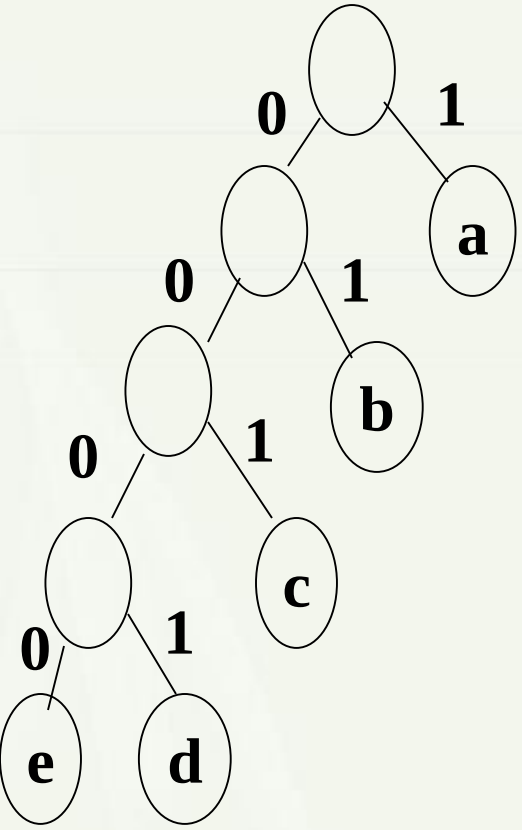
这样构造出来的一套编码一定是前缀码，译码唯一！

5.5 特殊二叉树

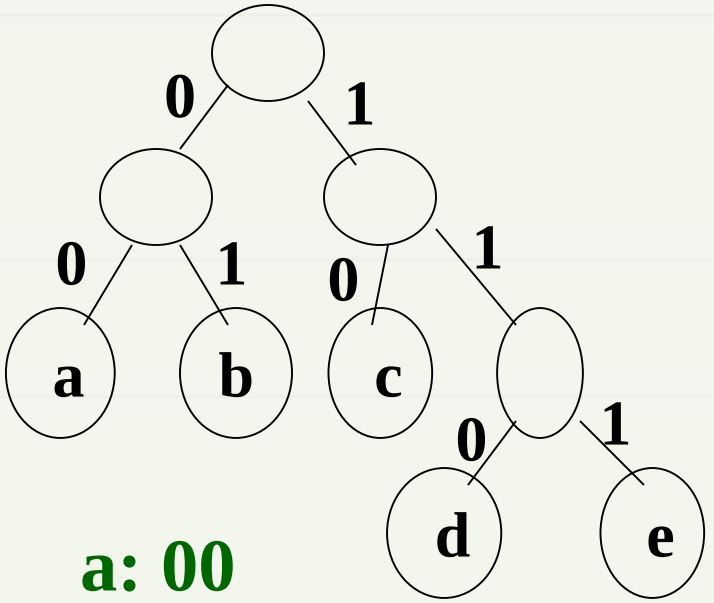
5.5.3 最优二叉树

5.5.3.3 最优二叉树的应用——Huffman编码

例如：有a,b,c,d,e 5个字符



a: 1
b: 01
c: 001
d: 0001
e: 0000



a: 00
b: 01
c: 10
d: 110
e: 111

5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.3 最优二叉树的应用——Huffman编码

2、使编码最短的方法——哈夫曼树

要使电文的编码长度最短，应该是使用频率高的字符，编码应该短，可以以使用频率作为权值，构造编码字符为叶子的哈夫曼树（二叉树），使用频率高的字符，离根近，编码短从而，电文编码后达到最短（而且是前缀码，由1知道）

因此，一棵最优二叉树就同时解决了译码唯一和编码长度最短的问题，由Huffman树得到的这套编码称为Huffman编码。

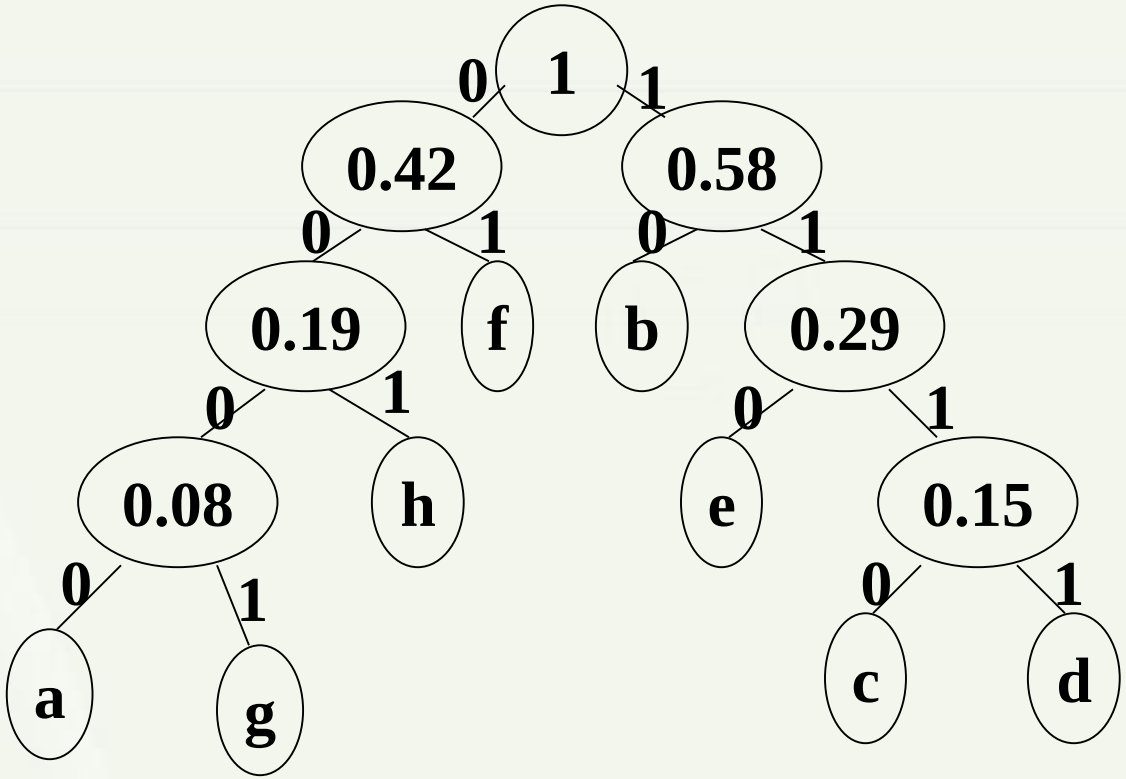
即：用被编码字符作为叶子（扩展结点），以其出现的频率作为权值，构造最优二叉树，然后左分支标“0”，右分支标“1”。从根到叶子的路径上的“0-1”就组成了该叶子字符的编码。

5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.3 最优二叉树的应用——Huffman编码

举例： 有电文用到8个字符 a,b,c,d,e,f,g,h，它们的使用频率为 0.05, 0.29, 0.07, 0.08, 0.14, 0.23, 0.03, 0.11，



a: 0000
b: 10
c: 1110
d: 1111
e: 110
f: 01
g: 0001
h: 001

5.5 特殊二叉树

5.5.3 最优二叉树

5.5.3.3 最优二叉树的应用——Huffman编码

那么，如何根据建立的哈夫曼树，进行：

编码？

译码？

具体算法请参阅有关文献，略！

5.5 特殊二叉树

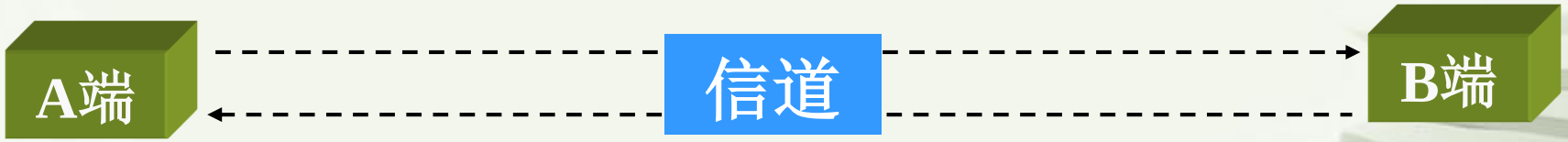
5.5.3 最优二叉树

上机实习作业（6）：

设计完成一个Huffman编/译码系统，实现双工通信。

[问题描述]

利用哈夫曼编码进行信息通讯可以大大提高信道利用率，缩短信息传输时间，降低传输成本。但是，这要求在发送端通过一个编码系统对待传数据预先编码；在接收端将传来的数据进行译码(复原)。对于双工信道 (即可以双向传输信息的信道)，每端都需要一个完整的编/译码系统。试为这样的信息收发站写一个哈夫曼码的编译码系统。



5.5 特殊二叉树

5.5.3 最优二叉树

[具体要求]

一个完整的系统应具有以下功能：

（1）建立编码：读入待传送文件F，统计各字符出现的频率；建立哈夫曼树，并将它存于文件hfmtree中（可以得到Huffman编码）。

（2）编码(Coding)：利用已建好的哈夫曼树(如不在内存，则从文件hfmtree中读入)，对待传送文件进行编码，形成已编码文件F1。

（3）发送、接收

（4）译码 (Decoding)：利用已建好的哈夫曼树将接收的文件进行译码，形成译码后的文件F2。

（5）校验：比较文件F和F2

（6）输出：以直观的形式输出哈夫曼树，输出各个字符的哈夫曼编码

本章小结

重点和难点：

1. 树、二叉树数据结构的定义、特点
2. 有关的术语（度、深度、满二叉树、完全二叉树……）
3. 二叉树的性质
4. 二叉树ADT的定义
5. 二叉树ADT实现
   顺序存储：方式、优缺点
   链式存储：方式、特点
6. 二叉树的重要操作
   遍历：方式
   创建：
7. 线索二叉树
   线索的目的、线索的存储、线索的使用

本章小结

重点和难点：

8. 树、森林与二叉树的关系
   转换
   遍历的对应
9. 特殊二叉树
   二叉排序树：特点、创建、应用——查找
   堆树：特点、创建（调整）、应用——最大最小
   最优二叉树：特点、构造、应用——哈夫曼编码

二叉树的性质、二叉树各种操作的实现是重点！


END