数据结构:二叉树

Data Structure

主讲教师：杨晓波

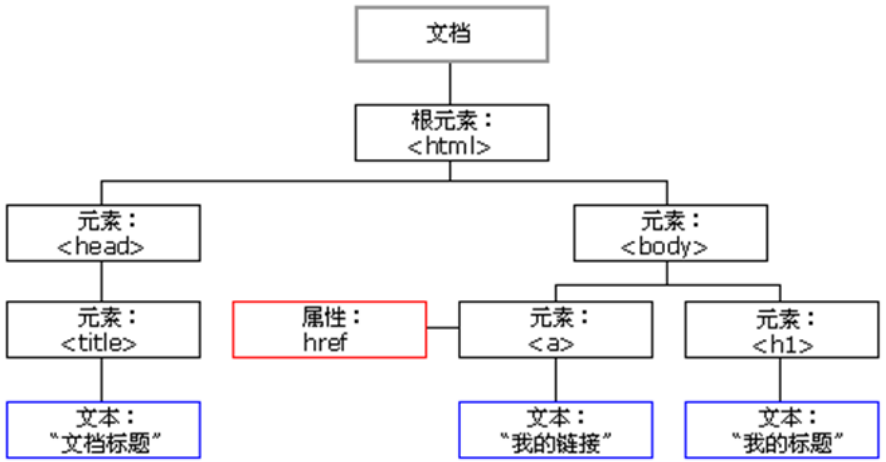
E-mail: 248133074@qq.com

层次结构的应用举例

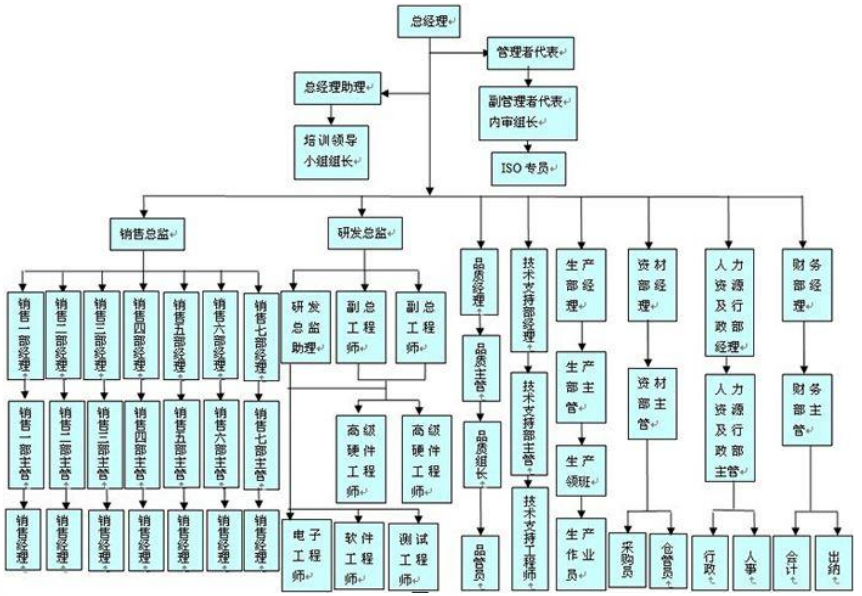
有些应用中数据具有典型的一对多特性，如：

- **家族谱**
- **组织结构图**
- **文件目录**
- **语法树**
- **楼栋的网络拓扑**
- **……**

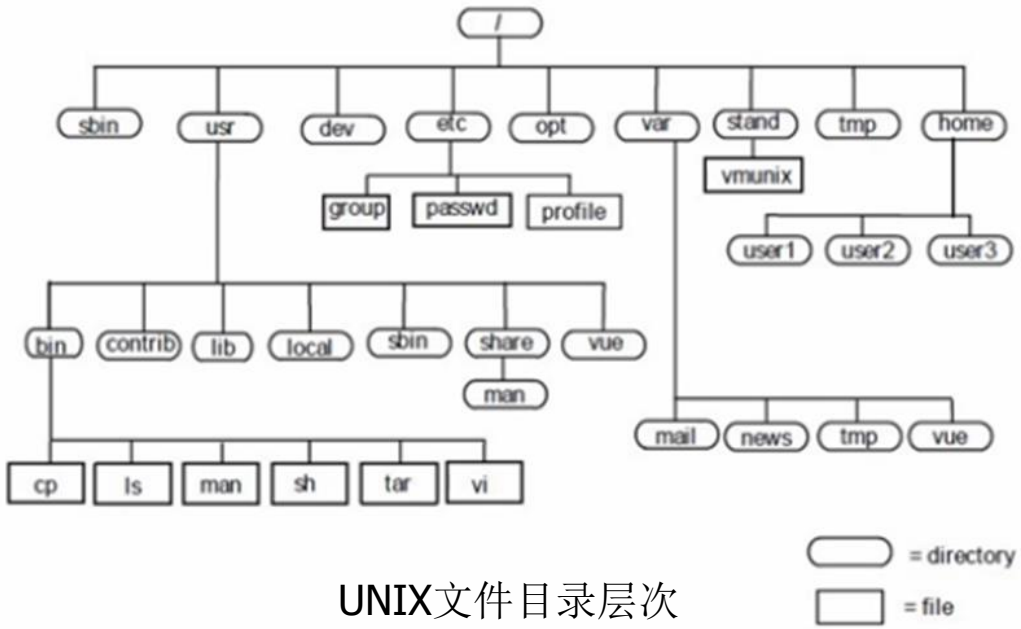
层次结构的应用举例



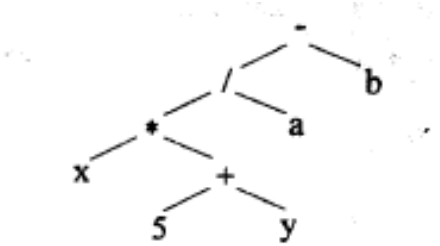
HTML文档的层次结构图



软件公司组织架构图



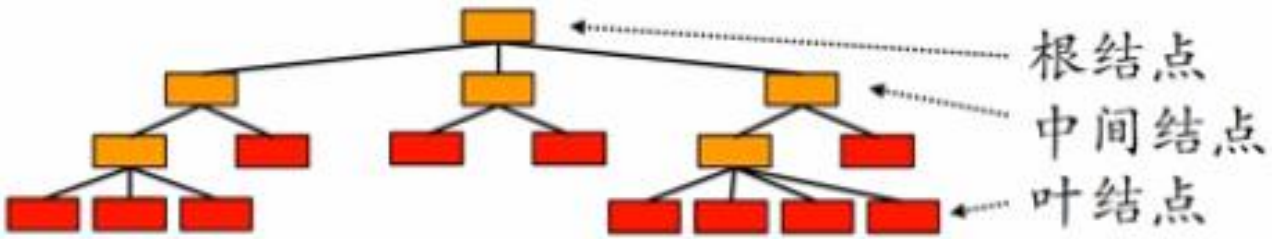
UNIX文件目录层次



抽象语法树

树型结构特征

- **树型逻辑结构的特点是：**
  - 集合中存在唯一的“无前驱”数据元素—根结点；
  - 集合中存在多个的“无后继”数据元素—叶结点；
  - 集合中其余每个数据元素均有唯一的直接前驱元素和多个的直接后继元素—中间结点。



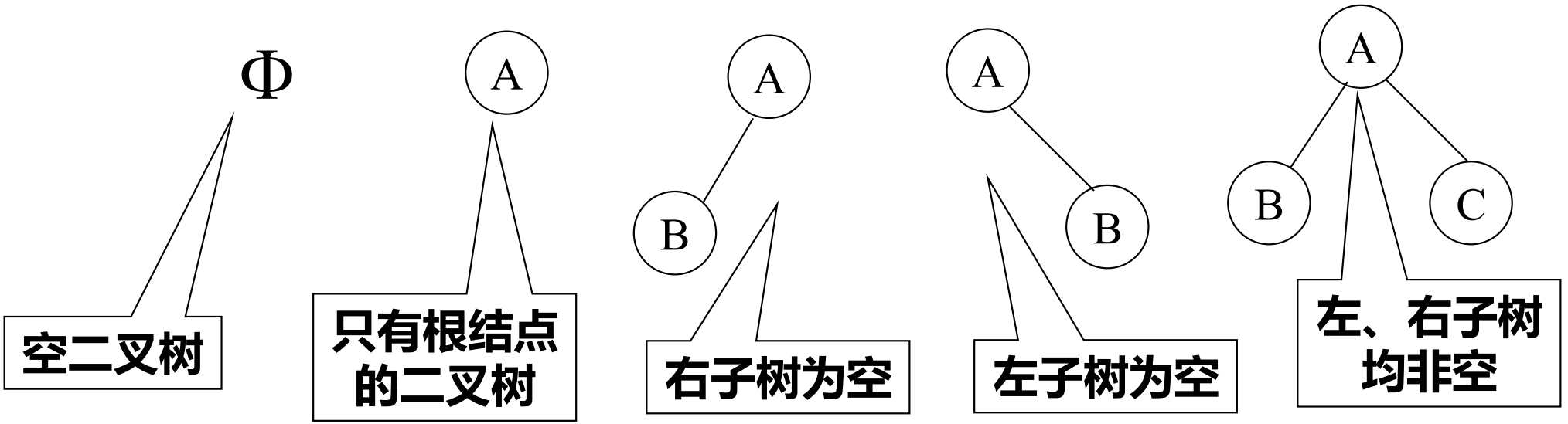
二叉树定义及主要特性

- **递归定义:**

  二叉树由结点的有限集合组成，这个集合或者为空，或者由一个根结点及两棵不相交的，分别称作这个根的左子树和右子树的二叉树组成。

- **特点:**
  - **每个结点至多有二棵子树。**
  - **二叉树的子树有左、右之分，且其次序不能任意颠倒。**

二叉树的基本形态 （5种）



树的类型定义

数据对象 D：

D是具有相同特性的数据元素的集合。

数据关系 R：

若D为空集，则称为空树 。

否则:

(1) 在D中存在唯一的称为**根**的数据元素root；

(2) 当n>1时，其余结点可分为m (m>0)个**互不相交**的有限集T_1, T_2, …, T_m，其中每一棵子集本身又是一棵符合本定义的树，称为根root的子树。

树的定义和术语

森林——一棵或者更多棵树的集合

- **结点:数据元素+若干指向子树的分支**
- **没有非空子树的结点称为叶结点(leaf)或终端结点。**
- **至少有一个非空子树的结点称为分支结点或内部结点(internal node)。**
- **结点的度：树结点的子结点数。**
  - **叶节点的度等于0，内部节点的度>0.**
- **树的度：树中所有结点的度的最大值**

树的术语

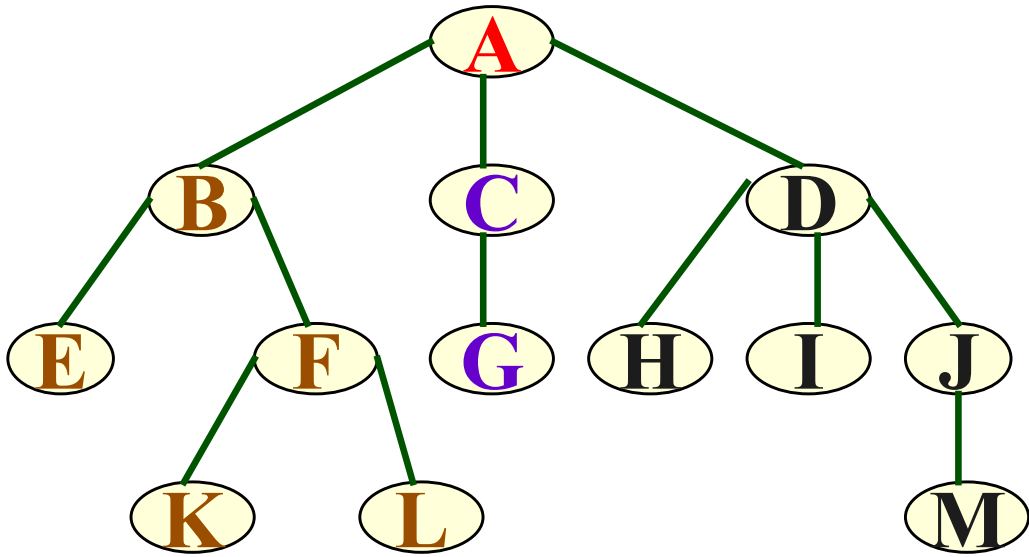
- 从一个结点到它的子结点都有边(edge)相连，这个结点称为它的子结点的父结点(parent)。
- 如果一棵树的一串结点$n_1, n_2, \ldots, n_k$有如下关系: 结点n_i是n_{i+1}的父结点$(1 \le i < k)$, 就把$n_1, n_2, \ldots, n_k$称为一条由n_1至n_k的路径(path)。
- 路径长度(length)是路径上边的数目。
- 如果有一条路径从结点R至结点M, 那么R就称为M的祖先(ancestor), 而M称为R的子孙(descendant)。

树的术语

- 结点M的深度(depth)就是从根结点到M的路径的长度。
- 树的高度(height)等于最深的结点的深度+1。任何深度为d的结点的层数(level)都为d。根结点深度为0，层数也为0。 (严版根结点深度和层次为1)

(从根到结点的)**路径**:

由从**根**到该结点所经分支和结点构成



孩子结点、**双亲**结点
兄弟结点、堂兄弟
祖先结点、**子孙**结点

结点的层次: **假设根结点的层次为0，第l层的结点的子树根结点的层次为$l+1$**

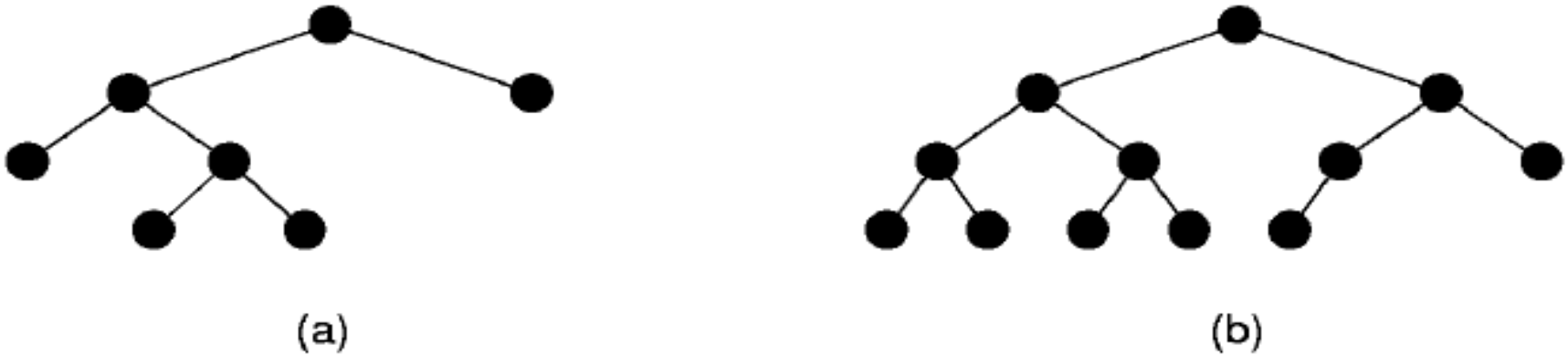
树的深度: **树中叶子结点所在的最大层次**

二叉树的相关术语

- **满二叉树**

  如果一棵二叉树的任何结点，或者是树叶，或者恰有两个非空子女的分支结点，则此二叉树称为满二叉树。

注：严版教材中满二叉树是各层结点都到了最大值。



(a)满二叉树(非完全二叉树) (b)完全二叉树(非满二叉树)

二叉树的相关术语

- **完全二叉树**

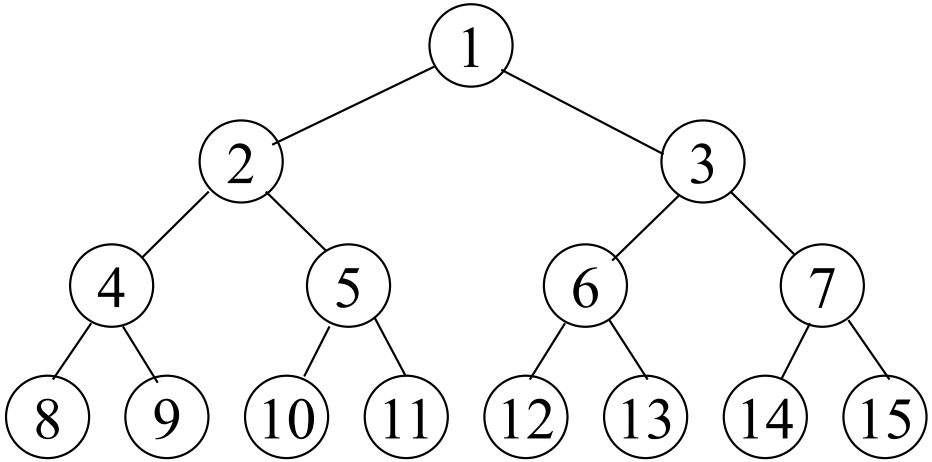
若一棵二叉树最多只有最下面的两层结点度数可以小于2，并且最下面一层的结点都集中在该层最左边的若干位置上，则称此二叉树为完全二叉树。

形状要求：

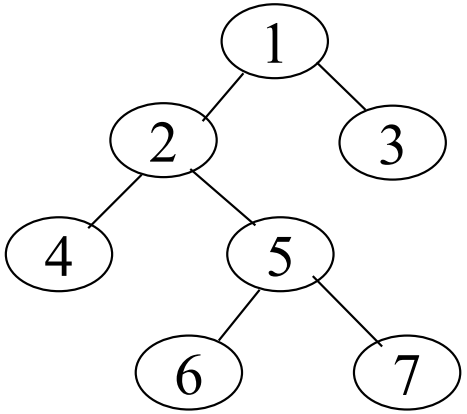
- **自根结点起每一层从左至右地填充。**
- **一棵完全二叉树（高度为d）除了最后一层（d-1层）外，每一层都是满的。**
- **底层叶结点集中在左边的若干位置上。**

满二叉树和完全二叉树习题

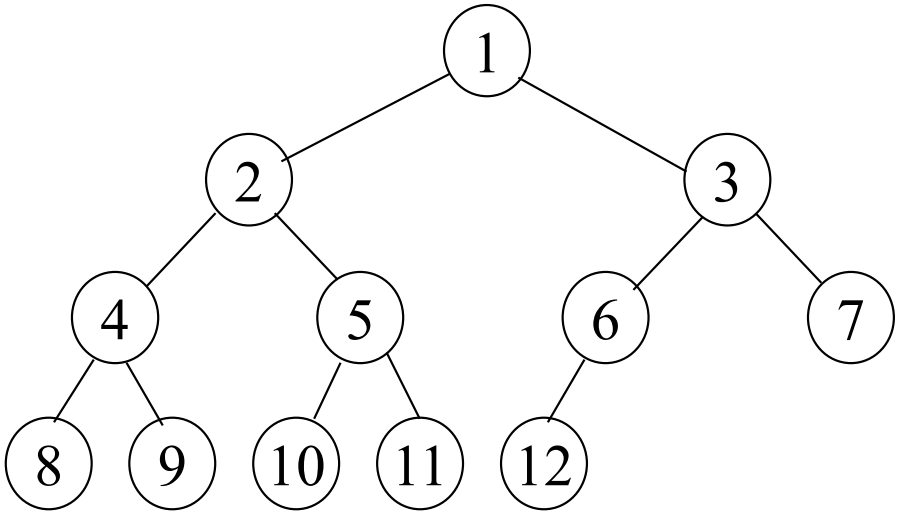
完成慕课上的满二叉树和完全二叉树的习题



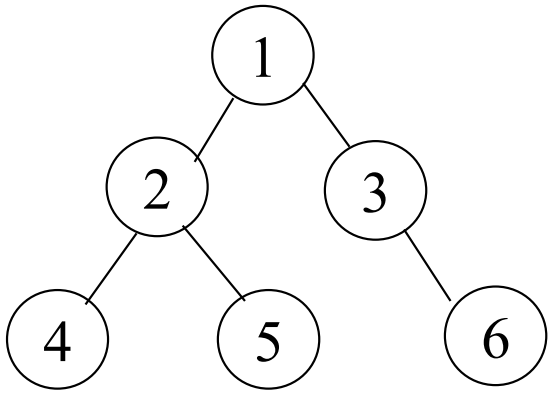
（a）



（b）



（c）



（d）

二叉树性质

- 补1.二叉树的第i层（根为第0层）最多有2^i个结点
- 补2.高度为k的二叉树至多有2^k-1个结点
- 补3. 具有 n 个结点的完全二叉树的高度为 $\lfloor log_2 n \rfloor +1$ 。

二叉树的重要特性

- **性质1**

  在二叉树的第 i 层上至多有 2^i 个结点。($i \geq 0$)

用科学归纳法证明：

归纳基础： $i = 0$ 层时，只有一个根结点：

$$2^i = 2^0 = 1;$$

归纳假设： 假设对所有的 j，$0 \leq j < i$，命题成立；则第i-1层的至多有 2^{i-1} 个结点。

归纳证明： 二叉树上每个结点至多有两棵子树，则第 i 层的结点数 $= 2^{i-1} \times 2 = 2^i$ 。

二叉树的重要特性

- **性质2**

  高度为 k 的二叉树上至多含 2^k-1 个结点（k≥0）。

证明：

根据性质1，高度为 k 的二叉树上的结点数至多为

$$2^0+2^1+\cdots\cdots+2^{k-1}=2^k-1$$ 。

二叉树的重要特性

- **性质3**

  具有 n 个结点的完全二叉树的高度为 $\lfloor \log_2 n \rfloor + 1$ 。

证明：

设完全二叉树的高度为 k

则根据第二条性质得 $2^{k-1} \leq n < 2^k$

即 $k-1 \leq \log_2 n < k$

因为 k 只能是整数，因此， $k = \lfloor \log_2 n \rfloor + 1$ 。

二叉树性质

- **性质4**

任何一棵二叉树，度为0的结点比度为2的结点多一个。

证明：

设有n个结点的二叉树的度为0、1、2的结点数分别为n_0，n_1，n_2，

则有：

$$n=n_0+n_1+n_2 \qquad \text{(公式1)}$$

设边数为e。因为除根以外，每个结点都有一条边进入，故n=e+1。

由于这些边是有度为1和2的结点射出的，因此$e=n_1+2*n_2$，于是

$$n=e+1=n_1+2*n_2+1 \qquad \text{(公式2)}$$

因此由公式（1）（2）得：

$$n_0+n_1+n_2=n_1+2*n_2+1$$

即$n_0=n_2+1$

二叉树的重要特性

- **性质5**（完全二叉树的父子节点位置可知，举例）

若对含 n 个结点的完全二叉树从上到下、从左至右进行 1 至 n 的编号，则对完全二叉树中任意一个编号为 i 的结点：

(1) 若 $i=1$，则该结点是二叉树的根，无双亲，

否则，编号为 $\lfloor i/2 \rfloor$ 的结点为其**双亲**结点；

(2) 若 $2i>n$，则该结点无左孩子，

否则，编号为 $2i$ 的结点为其**左孩子**结点；

(3) 若 $2i+1>n$，则该结点无右孩子结点，

否则，编号为 $2i+1$ 的结点为其**右孩子**结点。

二叉树性质

定理5.1 满二叉树定理：非空满二叉树树叶数等于其分支结点数加1。

证明：设非空满二叉树结点数为n，叶结点数为m，分支结点数为b，则有

$$n = m + b \qquad (1)$$

∵每个分支结点，恰有两个子结点（满），故一棵满二叉树有2*b条边；除根结点外，树中每个结点都恰有一条边联接父结点，故共有n-1条边。

$$即 n-1 = 2*b \qquad (2)$$

∴由(1)(2)得n-1=m+b-1=2*b，

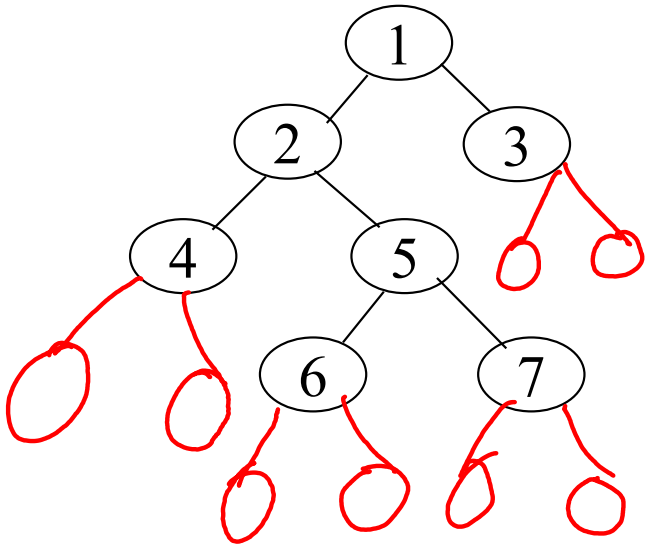
得m=b+ 1，定理得证。

二叉树性质

定理5.2、满二叉树定理的推论: 一棵非空二叉树空子树的数目等于其结点数目加1。

证明1： (构造法)

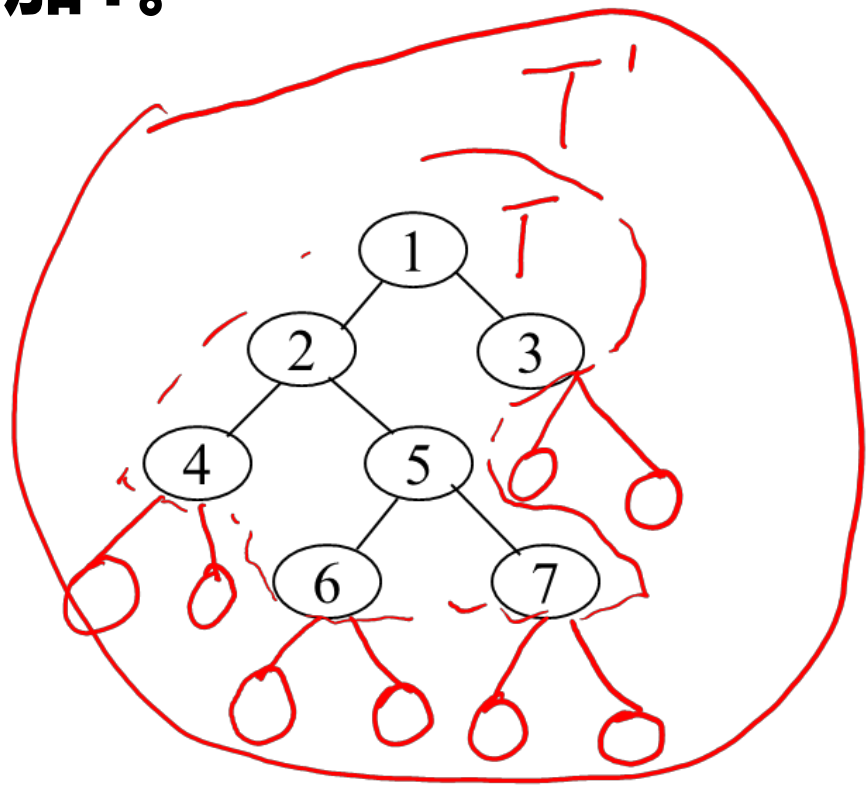
设满二叉树T，将其所有空子树换成叶结点，把新的满二叉树记为T ‘。



二叉树性质

定理5.2、满二叉树定理的推论: 一棵非空二叉树空子树的数目等于其结点数目加1。

证明1： (构造法)



设二叉树T，将其所有空子树换成叶结点，把新的二叉树记为T‘。

根据满二叉树定理，在T’中的叶结点数目等于其分支结点数目加1。
树T’的分支结点恰好是所有原来树T的结点；
而T’的叶结点都是构造时新添加的，每个新添加的叶结点对应树T的一棵空子树；
因此，树T中空子树的数目等于树T中结点数目加1。定理得证

二叉树性质

证明2：（计算法）

根据定义，二叉树T中每个结点都有两个子结点指针（空或非空）。

因此一个有n个结点的二叉树有2n个子结点指针。

除根结点外，共有n-1个结点，它们都是由其父结点中相应指针指引而来的，换句话说就有n-1个非空子结点指针。

既然子结点指针数为2n，则其中有n+1个为空(指针)。

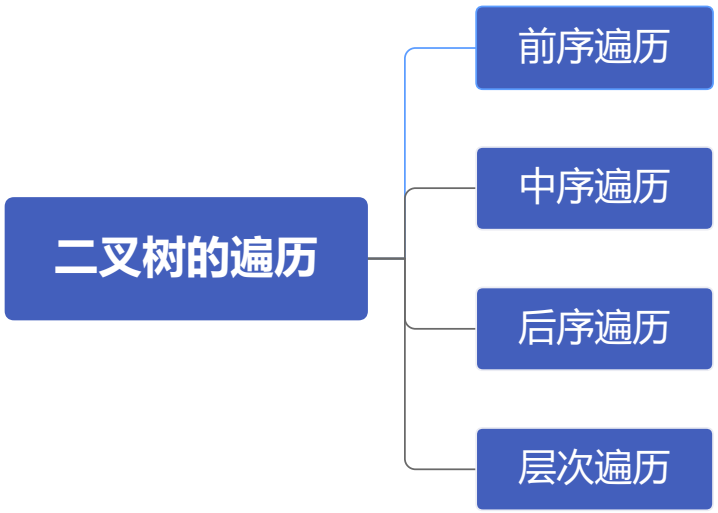
命题得证。

二叉树结点的抽象数据类型(BinNode.h)

```
template <typename E> class BinNode {
public:
   virtual ~BinNode() {} // Base destructor
   virtual  Elem& val( ) = 0 ;//取元素值
   virtual void setVal (const Elem&) = 0;//设置元素值
  virtual BinNode* left() const = 0; //返回左孩子指针
  virtual BinNode* right() const = 0;//返回右孩子指针
 virtual void setLeft(BinNode* ) = 0;//设置左孩子指针
  virtual void setRight(BinNode* ) = 0;//设置右孩子指针
  virtual bool isLeaf() = 0;//叶节点标识
};
```

二叉树的遍历

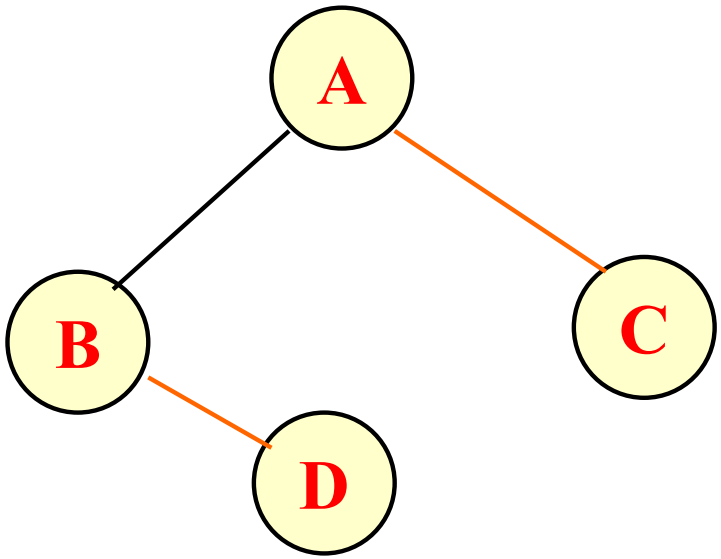
- **二叉树的遍历：系统地访问二叉树中的每个结点一次且仅一次。**

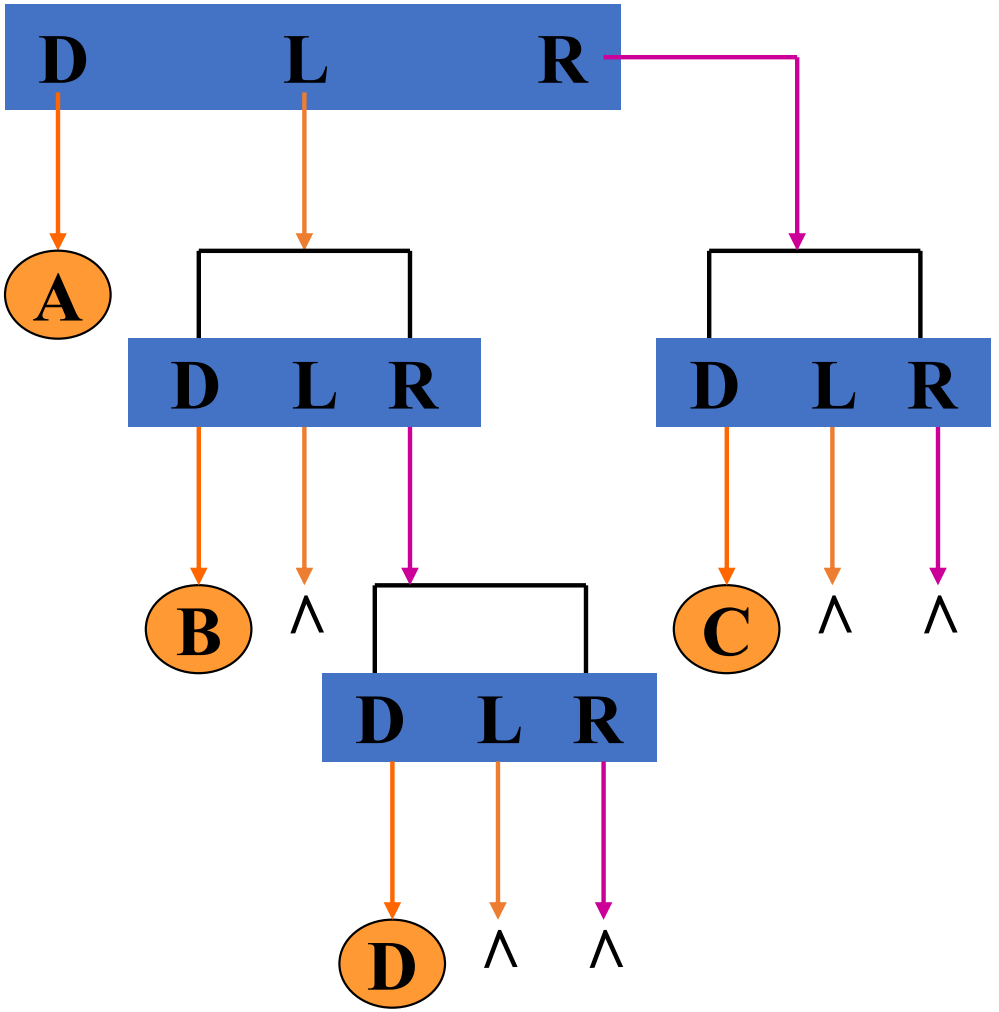


二叉树的遍历

- 前序遍历(preorder traversal)：访问根结点；前序遍历左子树;前序遍历右子树。 (DLR)
- 中序遍历(inorder traversal)：中序遍历左子树;访问根结点;中序遍历右子树。(LDR)
- 后序遍历(postorder traversal)：后序遍历左子树;后序遍历右子树;访问根结点。(LRD)
- 层次遍历：对二叉树自上而下逐层遍历，同层结点按照从左向右循序遍历

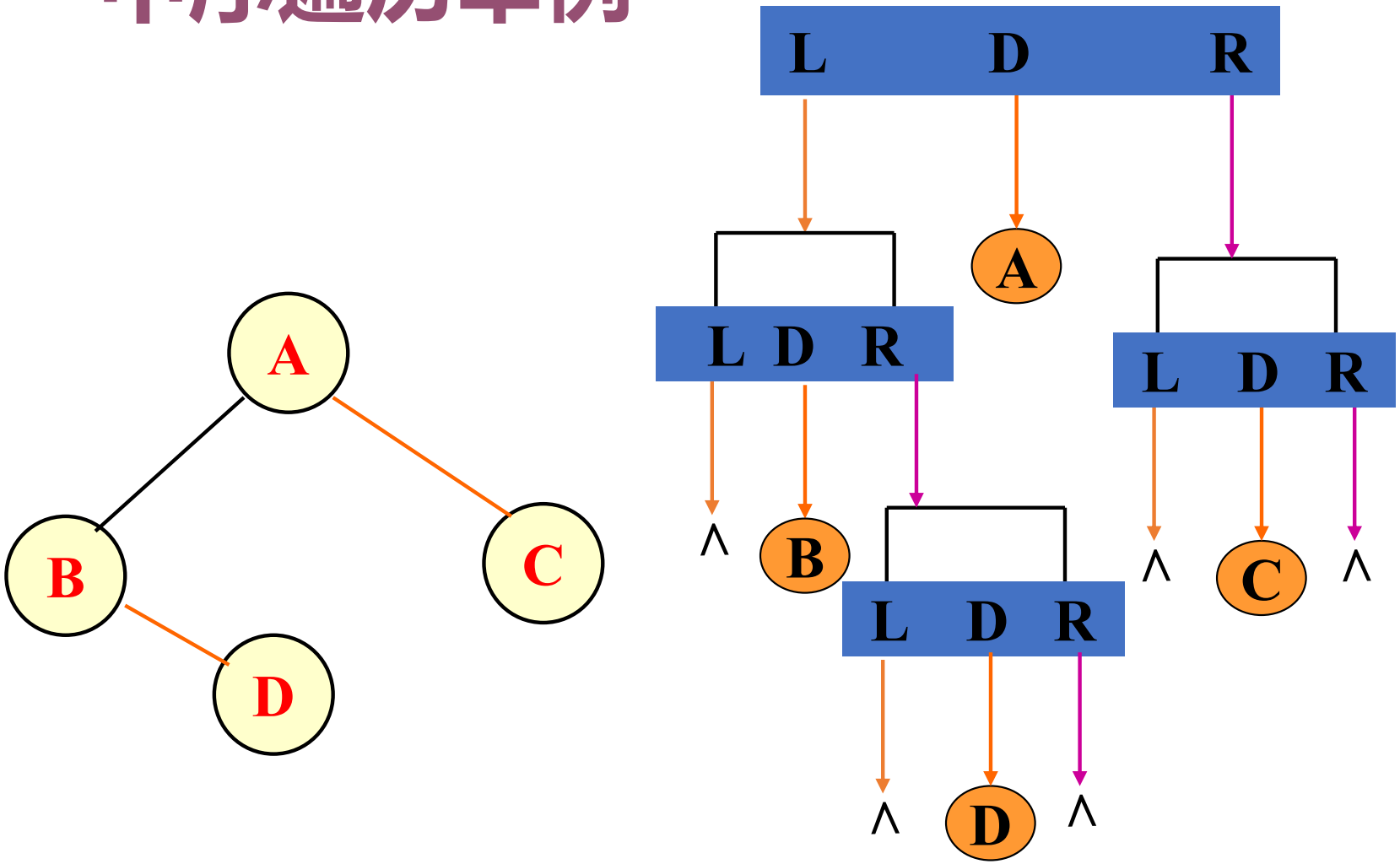
先序遍历举例





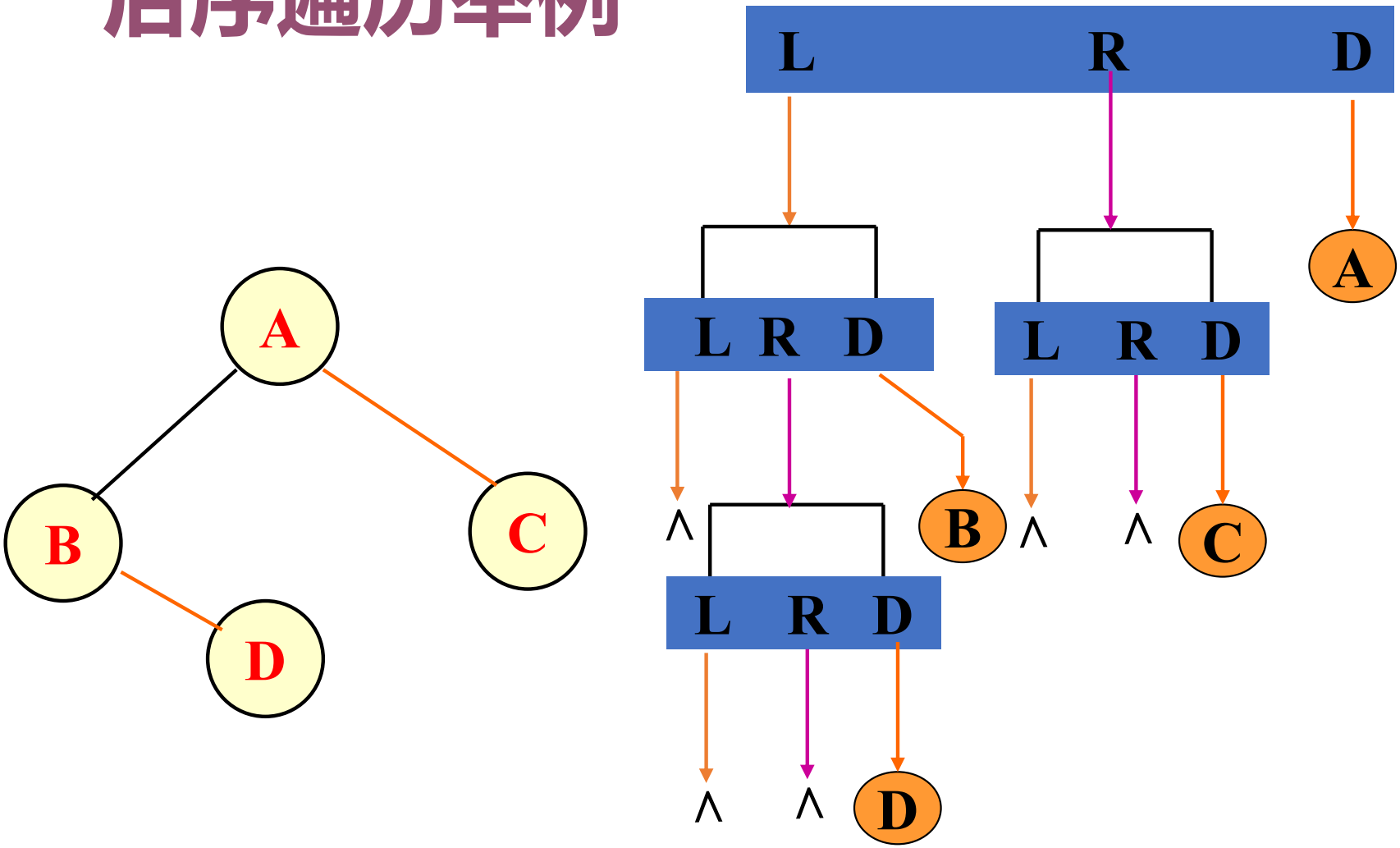
先序遍历序列：A B D C

中序遍历举例



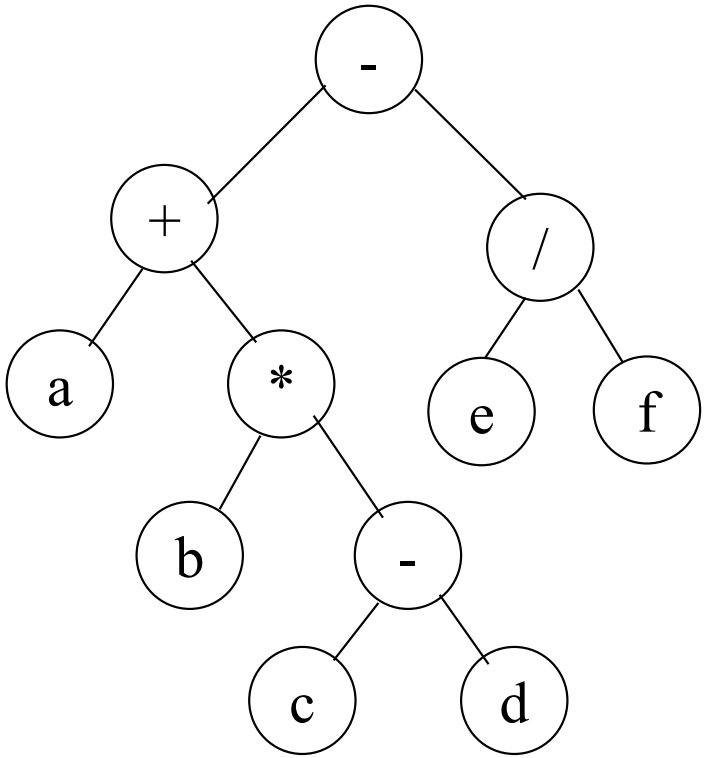
中序遍历序列：B D A C

后序遍历举例



后序遍历序列： D B C A

遍历举例——抽象语法树



先序遍历： - + a * b - c d / e f

中序遍历： a + b * c - d - e / f

后序遍历： a b c d - * + e f / -

层次遍历： - + / a * e f b - c d

前序遍历算法

```
template <typename E>
void preorder(BinNode<E>* root) {
  if (root == NULL) return; // Empty subtree, do nothing
  visit(root);              // Perform desired action
  preorder(root->left());
  preorder(root->right());
}
```

课堂练习：学习通上完成二叉树的遍历

二叉树的ADT

```
//Binary General tree ADT
template <class Elem> class BinTree {
public:
  BinTree();          //constructor
  ~BinTree();         //Destructor
  Binode* root();     //Return the root
  BiNode* CreateBiTree() ;          //创建二叉树
  bool BiTreeEmpty(BiNode* rt) ;

  Void preorder(BiNode* rt);
  void Inorder(BiNode* rt) ;
  void postorder(BiNode* rt) ;
  void LevelOrderTraverse(BiNode* rt) ;

  int BiTreeDepth(BiNode* rt) ;
};
```

问题 能根据二叉树的遍历序列还原出二叉树吗？

1.只知道二叉树的一个 遍历序列能构造出唯一的一棵二叉树吗？

例:

（1）仅知二叉树的先序序列“abcdefg” 不能唯一确定一棵二叉树，只知道树根标记为a，不知道哪些结点属于左子树，哪些属于右子树；

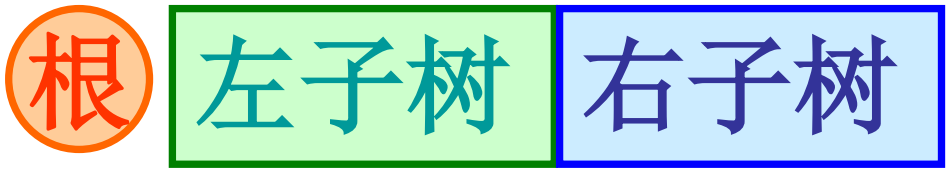
（2）后序序列情况与先序序列类似；

（3）中序序列呢?

由二叉树的先序和中序序列建树

如果同时已知二叉树的二叉树的先序序列“abcdefg”和中序序列“cbdaegf”，则会如何?

二叉树的先序序列



二叉树的中序序列

分析：先序序列中第一个结点对应根，在中序序列中出现在根左边的是左子树，出现在其右边的是右子树；

策略：先找根，再分左右

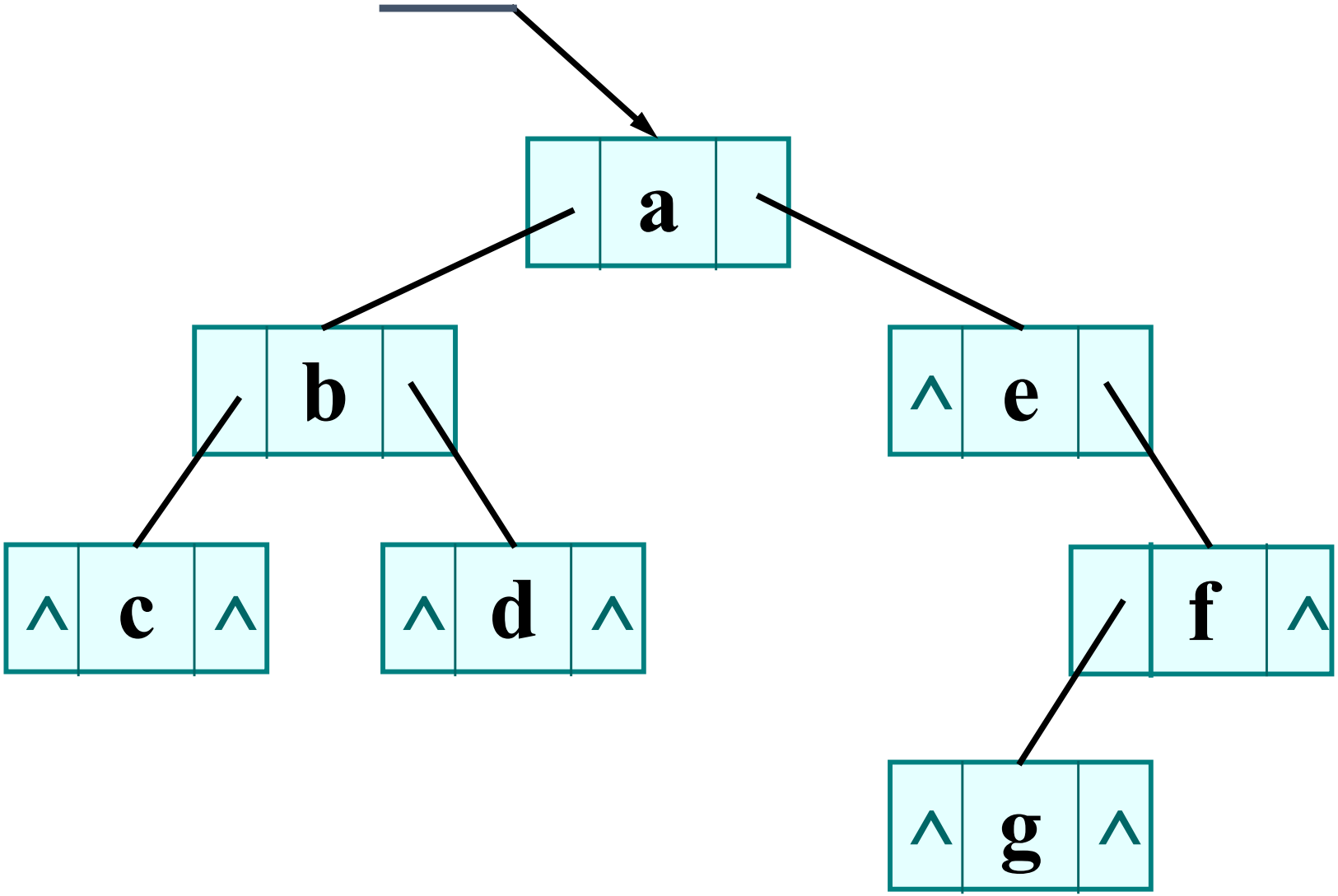
例如:

a b c d e f g

c b d a e g f

先序序列

中序序列



一起画

思考题

根据其他两种遍历序列是否也可以构造出唯一一棵二叉树?

- 已知后序遍历序列和中序遍历序列;
  - 例：已知某二叉树的后序遍历序列是dabec，中序遍历序列是debac，试构造出这棵树。

- 课后思考：若已知前序遍历序列和后序遍历序列，能构造出唯一的一棵二叉树吗?

遍历算法的应用举例

1. 统计二叉树中结点个数、叶子结点个数
2. **求二叉树的高度**
3. **复制二叉树**
4. **建立二叉树的存储结构**

遍历算法应用1

例5.4 计算二叉树的结点数：

算法思想：

采用递归的方法对二叉树进行先序遍历计算结点数。对于空树，返回结点数0；对于非空二叉树，其结点数为1加上其左右子树的结点值（递归地对其左、右子树进行先序遍历返回的结点值）。

```
template <typename E>
int count(BinNode<E>* root) {
  if (root == NULL) return 0;  // 空树
  return 1 + count(root->left())
           + count(root->right());
}
```

什么是递归?

- 递归是极强大的问题解决技术。
- 当一个函数用它自己来定义时就是递归。
- 递归将一个复杂问题分解为一些更小的问题。

- 优势：把复杂的问题变简单；通过这样的不断分解，最终得到的问题可能直接可以获得解决，从而解决原来的复杂问题。
- 注：递归不是数据结构；而是数据结构中常用的一种算法设计方法。

递归解决方案的一般形式

- 怎样按同类型的更小的问题来定义问题
- 各个递归调用怎么减小问题规模
- 哪个问题实例可用做基例
- 随着问题规模的减小，最终能否到达基例

举例1：n的阶乘

- 怎样按同类型的更小的问题来定义问题
  - n的阶乘等于n乘以n-1的阶乘
- 各个递归调用怎么减小问题规模
  - n的阶乘等于n乘以n-1的阶乘
  - n-1的阶乘等于n-1乘以n-2的阶乘
- 哪个问题实例可用做基例
  - n等于0时，0的阶乘等于1
- 随着问题规模的减小，最终能否到达基例
  - 可以。根据问题2的答案，n可以减小到基例

举例1：n的阶乘

```
public static int fact(int n){
//compute the factorial of a nonnegative integer
//precondition:n must be greater than or equal to 0
//postcondition:returns the factorial of n
//-----------------------
   if (n == 0) {
      return 1;
   }
   Else {
      return n*fact(n-1);
   } //end if

}//end fact
```

遍历算法应用2

例 计算二叉树的叶子结点数:

算法思路: 基于递归

- 怎样按同类型的更小的问题来定义问题
  - 二叉树叶子节点个数 =
    左子树叶子节点个数 + 右子树叶子节点个数
- 各个递归调用怎么减小问题规模
  - 二叉(子)树可以不断分解为根结点，左右子树
- 哪个问题实例可用做基例
  - 空二叉树，叶子结点数为0，子问题结束

-只有一个根结点的树，叶子结点数为1，子问题结束；

- 根据问题2的答案，最终可以到达基例。

遍历算法应用2

例 计算二叉树的叶子结点数：

算法思想：

采用递归的方法对二叉树进行先序遍历计算叶子结点数。对于空树，返回0；对于只有1个根结点的树，返回1；对于其他二叉树，其叶子结点数为其左右子树的叶子结点值之和（递归地对其左、右子树进行先序遍历计算返回的叶子结点值）。

遍历算法应用2

例 计算二叉树的叶子结点数：

伪代码：

```
template <typename E>
int count(BinNode<E>* root) {
  if (root == NULL) return 0;  // Nothing to count
  if(root->isleaf()) return 1;//根结点就是叶子结点
  else return count(root->left())
          + count(root->right());
}
```

算法的时空复杂度分析：

时间复杂度为O(n)（因为每个结点都被访问一次）

空间复杂度为O(logn)（因为递归需要存储空间，所以空间开销为最大同时活跃的递归调用数，即二叉树的高度）

遍历算法应用3——求二叉树的高度

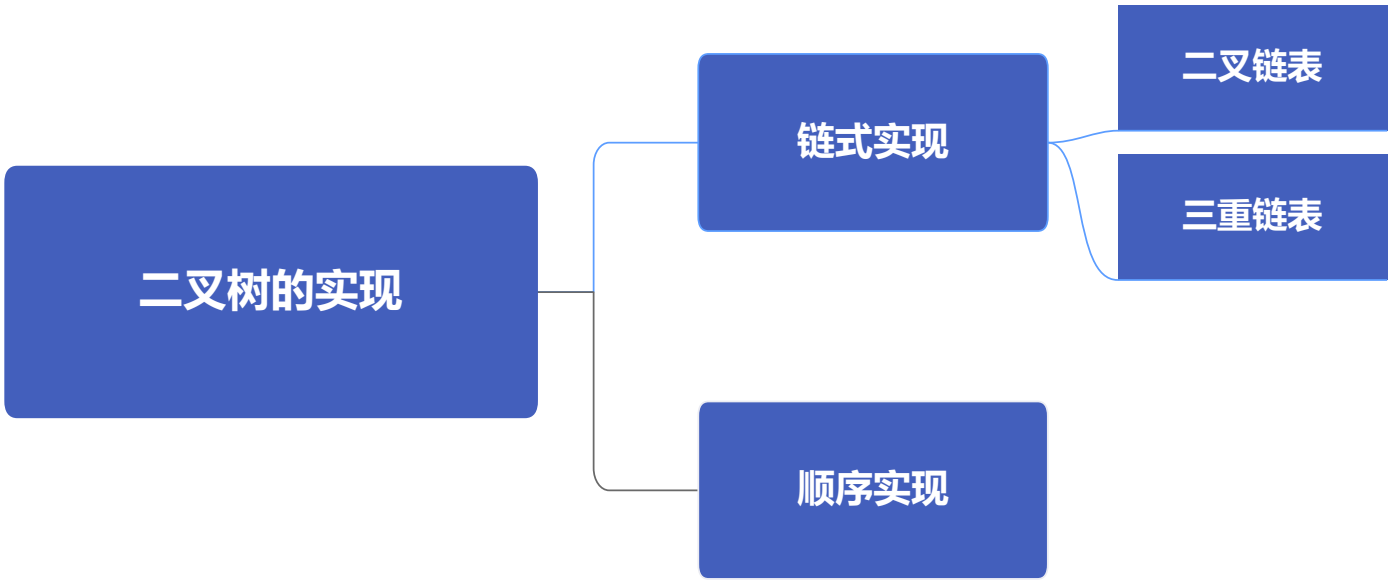
算法思想：

从二叉树高度的定义可知，**二叉树的高度应为其左、右子树高度的最大值加1。**由此，**需先分别求得左、右子树的高度，**算法中“访问结点”的操作为：空树的高度为0，对于非空二叉树递归调用求高度函数**求得左、右子树高度的最大值，然后加 1 。**

求二叉树的高度的伪代码

```
int Depth (BiTree T ){ // 返回二叉树的高度
   if ( !T )    depthval = 0;//空树
   else  {
     depthLeft = Depth( T->lchild );
     depthRight= Depth( T->rchild );
     depthval = 1 + (depthLeft > depthRight  ?
                          depthLeft : depthRight);
   }
   return depthval;
}
```

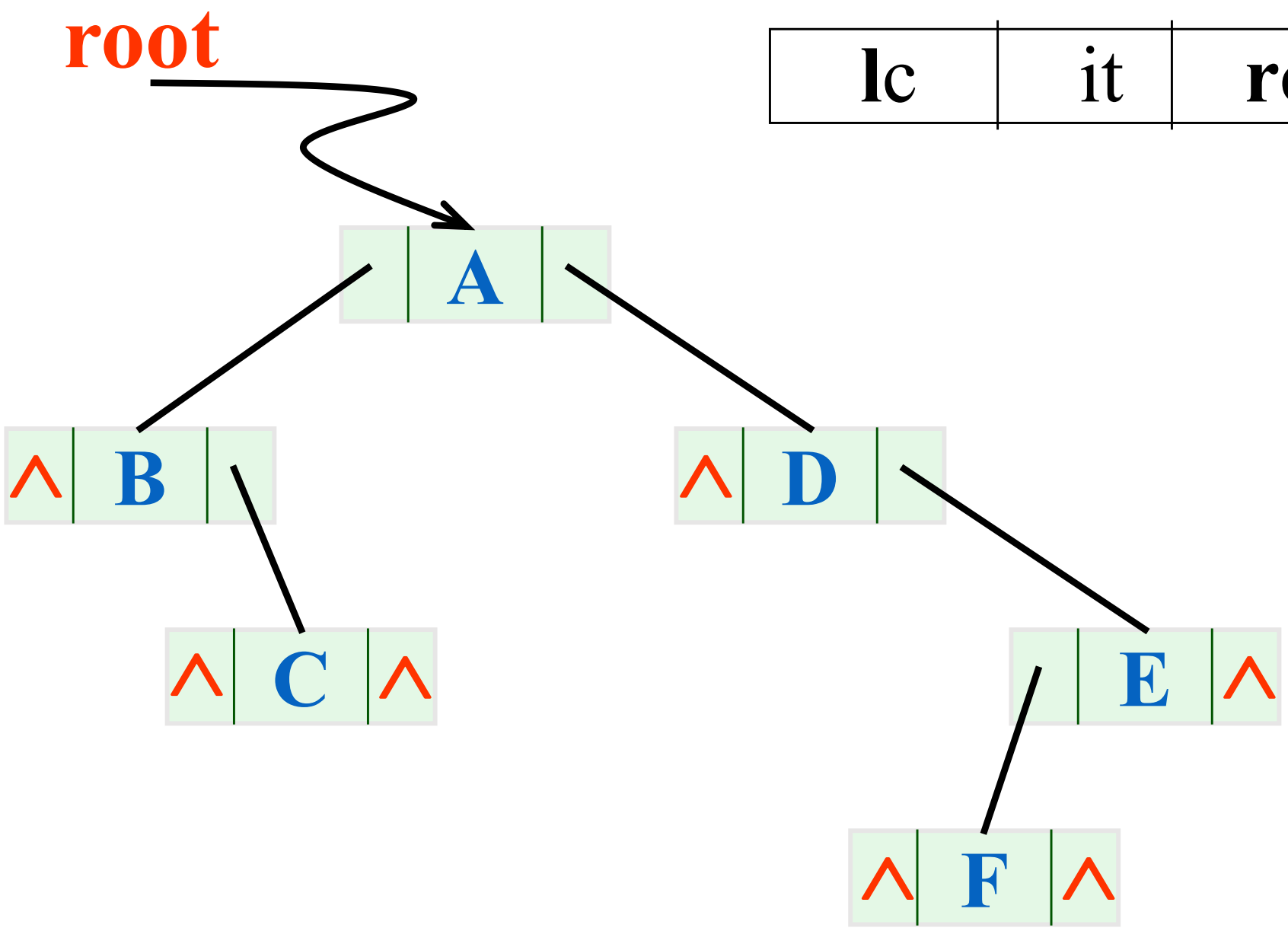
二叉树的实现



二叉链表

结点结构:

lc	it	rc

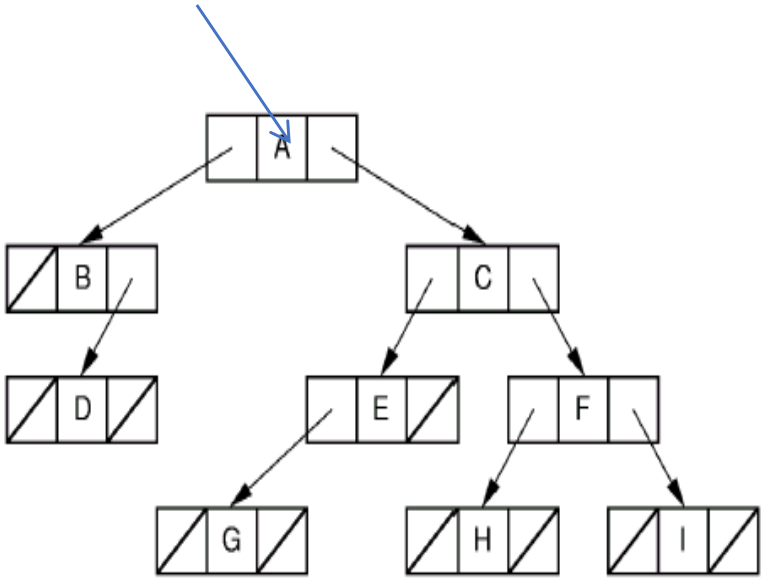


二叉树的实现

使用指针实现二叉树

- **二叉链表(最常用)（BSTNode.h二叉查找树结点的头文件)**

```
#include "BinNode.h"
// Simple binary tree node implementation
template <typename Key, typename E>
class BSTNode : public BinNode<E> {
private:
  Key k;              // The node's key
  E it;               // The node's value
  BSTNode* lc;        // Pointer to left child
  BSTNode* rc;        // Pointer to right child
```



二叉树的实现——二叉链表2

```
public:
  // Two constructors -- with and without initial
   values
  BSTNode() { lc = rc = NULL; }
  BSTNode(Key K, E e, BSTNode* l =NULL,
   BSTNode* r =NULL)
    { k = K; it = e; lc = l; rc = r; }
  ~BSTNode() {}              // Destructor
```

二叉树的实现——二叉链表3

```
// Functions to set and return the value and key
E& element() { return it; }
void setElement(const E& e) { it = e; }
Key& key() { return k; }
void setKey(const Key& K) { k = K; }
```

二叉树的实现

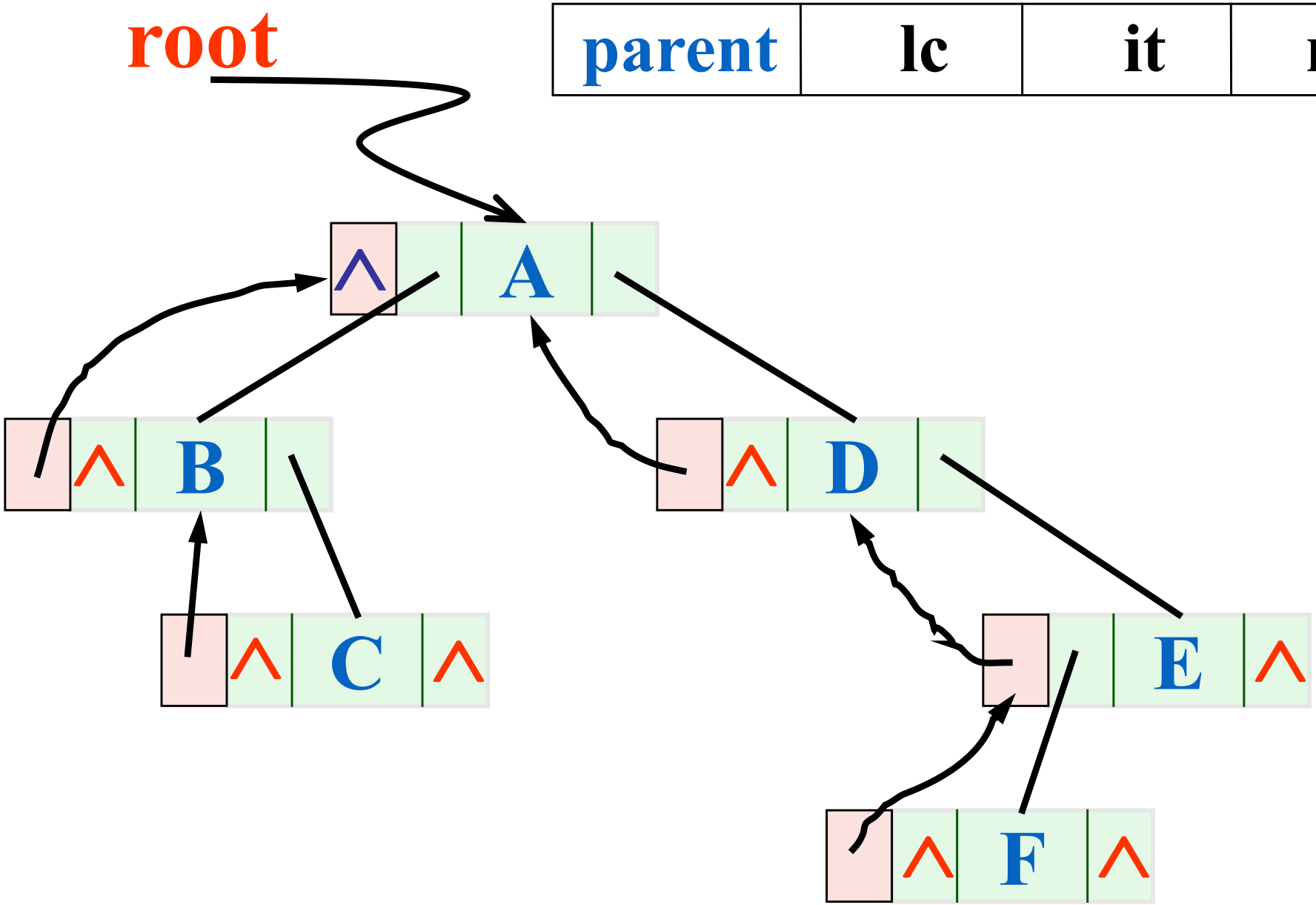
```
// Functions to set and return the children
  inline BSTNode* left() const { return lc; }
  void setLeft(BinNode<E>* b) { lc = (BSTNode*)b; }
  inline BSTNode* right() const { return rc; }
  void setRight(BinNode<E>* b) { rc =
   (BSTNode*)b; }

  // Return true if it is a leaf, false otherwise
  bool isLeaf() { return (lc == NULL) && (rc ==
   NULL); }
};
```

好处：运算方便;问题：空指针太多

三重链表

结点结构:

parent	lc	it	rc



二叉树的存储——三重链表1

- 三重链表的变化：加了父指针
- 在某些经常要回溯到父结点的应用中很有效。

```
class BSTNode:public BinNode<E> {
  private:
    Key k;
    E it;
    BSTNode* lc;
    BSTNode* rc;
     BSTNode* father;//父指针
  …};
```

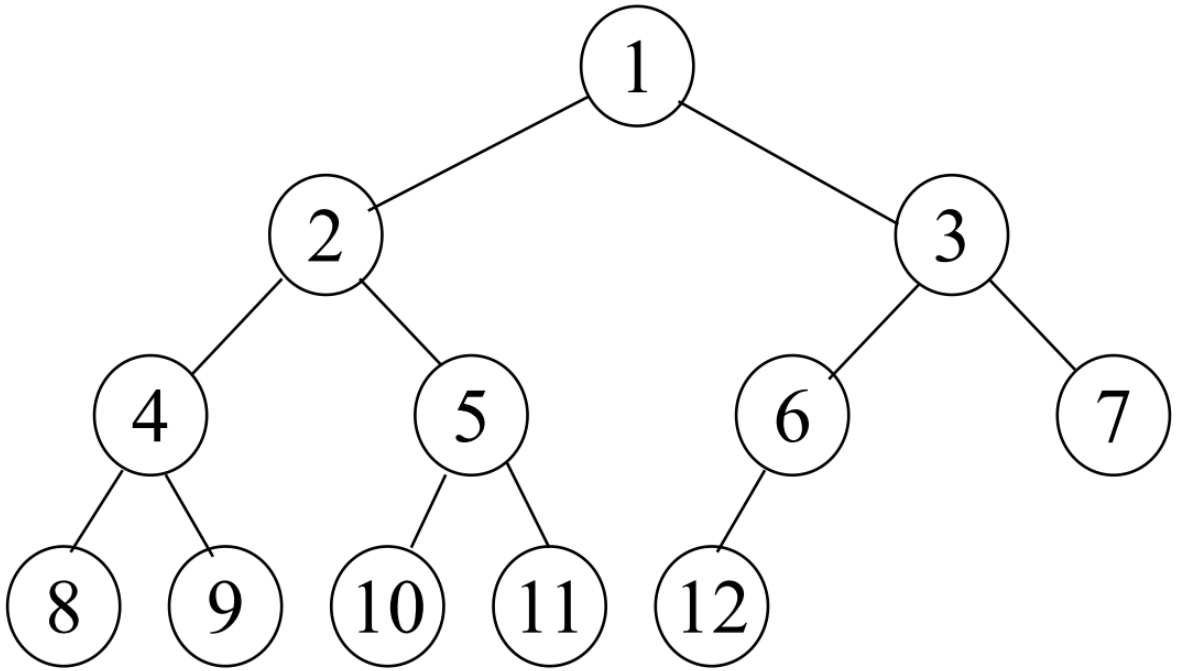
思考

问题：

二叉树只用链式存储就够了吗？

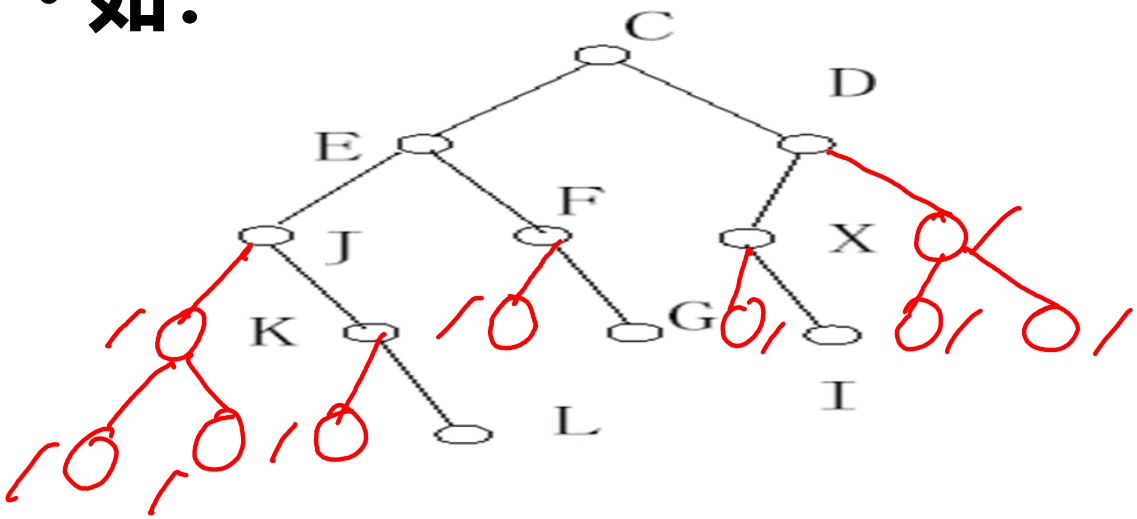
二叉树的实现3——
使用数组实现完全二叉树

- 在完全二叉树中，父子之间的关系可以通过索引的数学关系计算出来（参见二叉树的性质5）
- 完全二叉树的顺序存储，按照二叉树的层次遍历次序存储在一个数组中：　　1 2 3 4 5 6 7 8 9 10 11 12
- 简单，省空间



二叉树的顺序存储——非完全二叉树

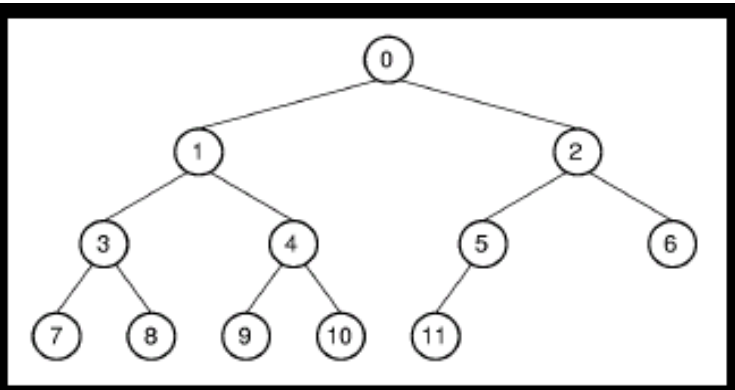
- 非完全二叉树可通过将空指针置空值后转换为完全二叉树存储 （补全法）
- 如：



上图中的树可存储为：

CEDJFX//K/G/I/////L

完全二叉树的下标对应关系



Position	0	1	2	3	4	5	6	7	8	9	10	11
Parent	--	0	0	1	1	2	2	3	3	4	4	5
Left Child	1	3	5	7	9	11	--	--	--	--	--	--
Right Child	2	4	6	8	10	--	--	--	--	--	--	--
Left Sibling	--	--	1	--	3	--	5	--	7	--	9	--
Right Sibling	--	2	--	4	--	6	--	8	--	10	--	--

由于编码从0开始，编号为r的结点的左孩子编号为2（r+1)-1=2r+1，右孩子编号为2（r+1)+1-1=2r+2，双亲结点编号为$\lfloor (r+1)/2 \rfloor -1$

完全二叉树的下标公式

公式中r表示结点的索引， n表示二叉树结点总数。

Parent(r) = $\lfloor (r-1)/2 \rfloor$，当r!=0时。

Leftchild(r) =2r +1，当2r+1<n时。

Rightchild(r) = 2r+ 2 ，当2r+2 < n 时。

Leftsibling(r) =r-1，当r为偶数且0<=r<=n-1。

Rightsibling(r) =r+1，当r为奇数且r+1<n。

思考

二叉树的ADT是以二叉树结点为基础的。

通用的树ADT的抽象跟树的结点也密切相关。

通用的树的结点跟二叉树的结点有什么不一样？

通用的树的度可能大于2；

可能是一个很大的值。

因此用孩子指针的方式就不现实

General Tree Node的类定义

```
template <class Elem> class GTNode {
public:
  GTNode(const Elem&); // Constructor
  ~GTNode();           // Destructor
  Elem value();        // Return value
  bool isLeaf();       // TRUE if is a leaf
  GTNode* parent();    // Return parent
  GTNode* leftmostChild(); // First child
  GTNode* righSibling();  // Right sibling
  void setValue(Elem&);     // Set value
  void insertFirst(GTNode<Elem>* n);//插入第一个孩子结点
  void insertNext(GTNode<Elem>* n);//插入下一个兄弟结点
  void removeFirst(); // Remove first child
  void removeNext();  // Remove right sibling
};
```

树的ADT

```
// General tree ADT
template < class Elem > class GenTree {
private:
  void printhelp(GTNode*); // Print helper function
public:
  GenTree();              // Constructor
  ~GenTree();             // Destructor
  void clear();           // Send nodes to free store
  GTNode* root();         // Return the root
  // Combine two subtrees
  void newroot(Elem&, GTNode<Elem>*,
    GTNode<Elem>*);
  void print();           // Print a tree
};
```

树的遍历

先根(次序)遍历

若树不空，则先访问根结点，然后依次先根遍历各棵子树。

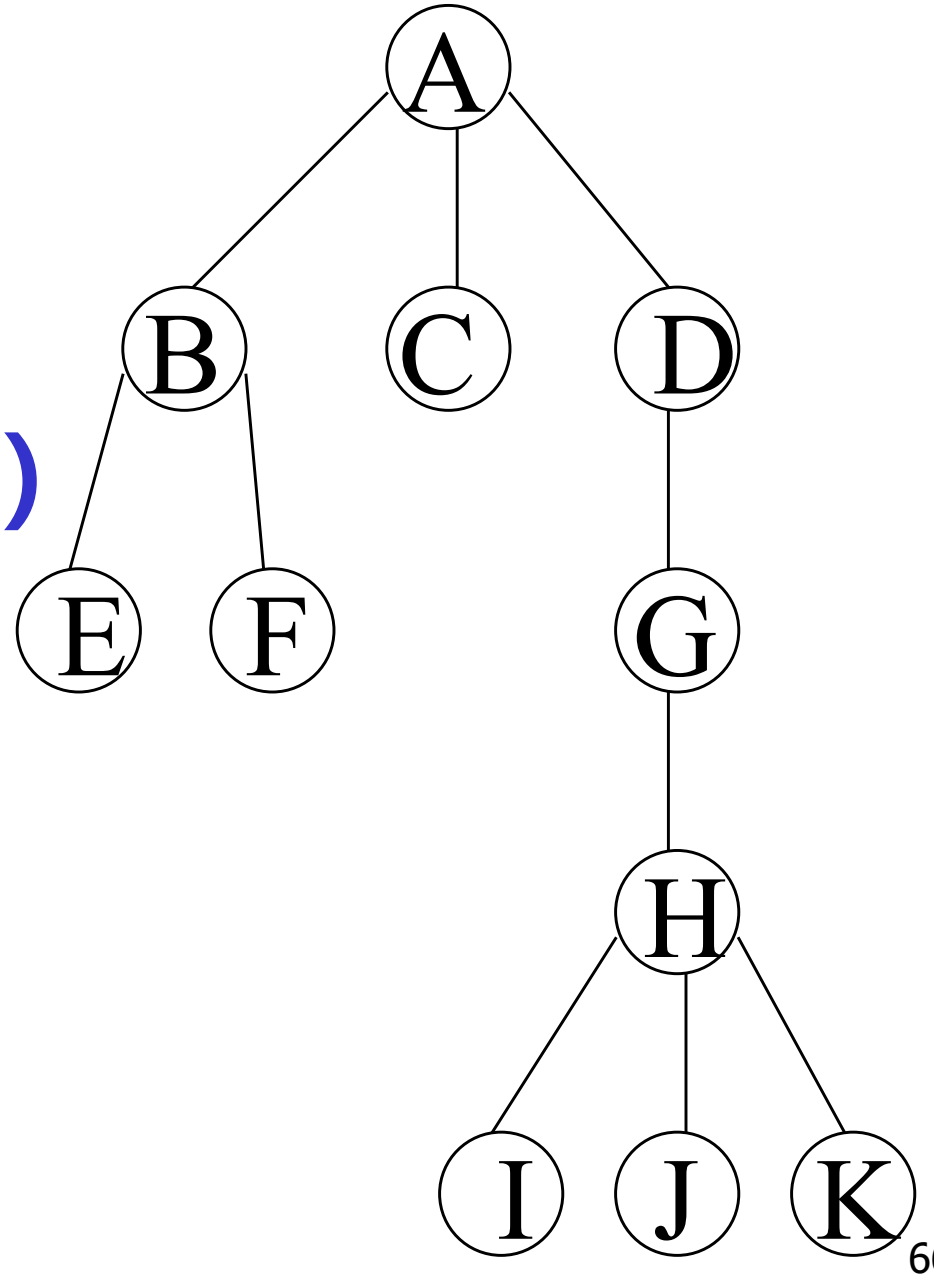
后根(次序)遍历

若树不空，则先依次后根遍历各棵子树，然后访问根结点。

按层次(广度)遍历

若树不空，则自上而下自左至右访问树中每个结点。

对右图分别写出
先根遍历、后根遍历
和层次遍历的时顶点
的访问次序：（5分钟）



先根遍历时顶点的访问次序：

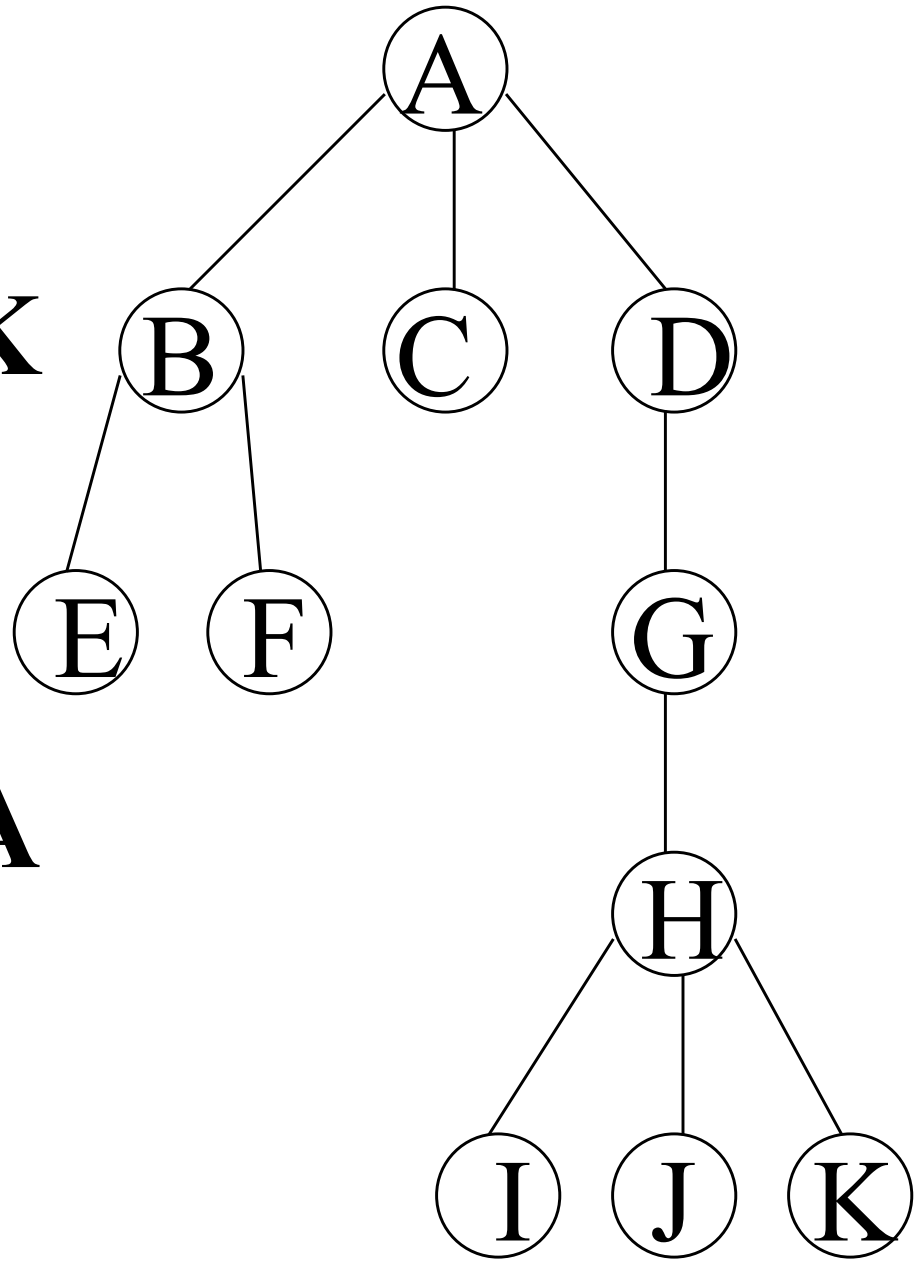
A B E F C D G H I J K

后根遍历时顶点的访问次序：

E F B C I J K H G D A

层次遍历时顶点的访问次序：

A B C D E F G H I J K



先根遍历算法

```
template < class Elem >
void GenTree<Elem>::
printhelp(GTNode<Elem>* root) {
  if (root->isLeaf()) cout << "Leaf: "; //叶子结点
  else cout << "Internal: "; //中间结点
  cout << root->value() << "\n";
  for (GTNode<Elem>* temp =
       root->leftmostChild();
       temp != NULL;
       temp = temp->rightSibling())//遍历每颗子树
    printhelp(temp);
}
```

5.4 二叉检索树
(BST，又称为二叉查找树)

定义：二叉检索树或者为空，或者是满足下列条件的非空二叉树：

(1)若它的左子树非空，则左子树上所有结点的值均小于根结点的值；

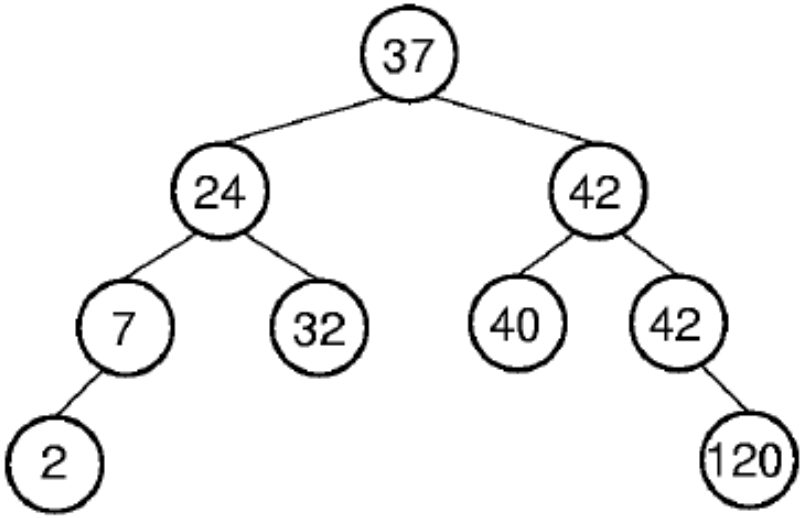
(2)若它的右子树非空，则右子树上所有结点的值均大于或等于根结点的值；

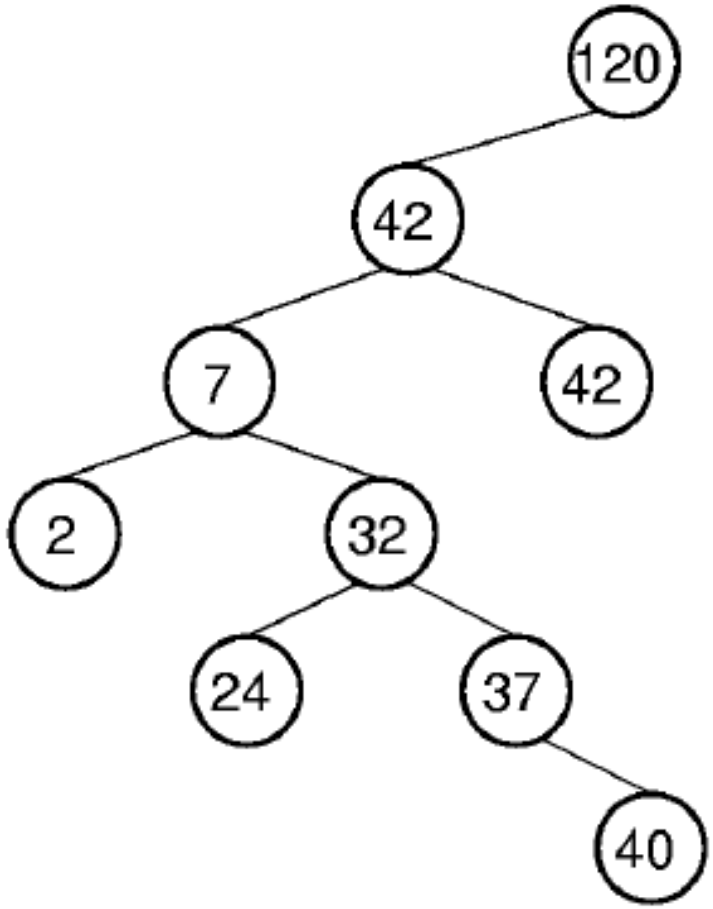
(3)左右子树本身又各是一棵二叉检索树。

性质：按照中序遍历将各结点打印出来，将得到按照由小到大的排列。

注：有序性是BST的显著特征，对它的插入或删除等操作均需维持其有序性。（完全有序）

BST图示





思考:

从上面两幅图，大家可以得到一些什么结论?

二叉检索树的检索过程

二叉检索树的效率就在于只需检索二个子树之一。

- **从根结点开始，在二叉检索树中检索值K。**
  - **如果根结点的值为K，则检索结束。**
  - **如果K小于根结点的值，则只需检索左子树**
  - **如果K大于根结点的值,就只检索右子树**
- **这个过程一直持续到K被找到或者遇到一个树叶。**
- **如果遇到树叶仍没有发现K，那么K就不在该二叉检索树中。**

分析：

检索过程最多经过从树根到树叶的一条路径，因此效率与二叉检索树的高度有关。

二叉检索树类定义（BST.h)

```
// Binary Search Tree implementation for the Dictionary ADT
template <typename Key, typename E>
class BST : public Dictionary<Key,E> {
private:
  BSTNode<Key,E>* root;   // Root of the BST
  int nodecount;          // Number of nodes in the BST
 // Private "helper" functions
  void clearhelp(BSTNode<Key, E>*);//清空
  BSTNode<Key,E>* inserthelp(BSTNode<Key, E>*,
                             const Key&, const E&);//插入结点，返回插入后BST根结点指针
  BSTNode<Key,E>* deletemin(BSTNode<Key, E>*);
     //删除最小值结点，返回删除后BST根结点指针
  BSTNode<Key,E>* getmin(BSTNode<Key, E>*);//获取指向最小值结点的指针
  BSTNode<Key,E>* removehelp(BSTNode<Key, E>*, const Key&);
    //删除结点，返回删除后BST根结点指针
  E findhelp(BSTNode<Key, E>*, const Key&) const;//查找关键字所在结点
  void printhelp(BSTNode<Key, E>*, int) const;//打印
```

参数为BST树跟结点的指针和索引值

二叉检索树类定义

```
public:
  BST() { root = NULL;nodecount=0; } //Contructor
  ~BST() { clearhelp(root); }                  //Destructor

  void clear()//重新初始化树
 {clearhelp(root);root=NULL; nodecount=0;}

  // Insert a record into the tree，  k Key value of the record.
  // e The record to insert.
  void insert(const Key& k, const E& e) {
    root = inserthelp(root, k, e);
    nodecount++;
  }
```

二叉检索树类定义

```
// Remove a record from the tree.
  // k Key value of record to remove.
  // Return: The record removed, or NULL if there is none.
  E remove(const Key& k) {
                //从二叉树中删除关键字值为k的一条记录并返回元素值
    E temp = findhelp(root, k);   // First find it
    if (temp != NULL) {
      root = removehelp(root, k);
      nodecount--;
    }
    return temp;  }
  // Remove and return the root node from the dictionary.
  // Return: The record removed, null if tree is empty.
  E removeAny() {  // Delete min value，实际为删除根节点
    if (root != NULL) {
      E temp = root->element();
      root = removehelp(root, root->key());
      nodecount--;
      return temp;    }
    else return NULL;  }
```

二叉检索树类定义

```
// Return Record with key value k, NULL if none exist.
  // k: The key value to find. */
  // Return some record matching "k".
  // Return true if such exists, false otherwise. If
  // multiple records match "k", return an arbitrary one.
  E find(const Key& k) const { return findhelp(root, k); }

  // Return the number of records in the dictionary.
  int size() { return nodecount; }

  void print() const { // Print the contents of the BST
    if (root == NULL) cout << "The BST is empty.\n";
    else printhelp(root, 0);
  }
};
```

二叉检索树类定义——检索

```
// Find a node with the given key value
template <typename Key, typename E>
E BST<Key, E>::findhelp(BSTNode<Key, E>* root,
                        const Key& k) const {
  if (root == NULL) return NULL;          // Empty tree
  if (k < root->key())
    return findhelp(root->left(), k);   // Check left
  else if (k > root->key())
    return findhelp(root->right(), k);  // Check right
  else return root->element();  // Found it
}
```

二叉检索树类定义

```
/ Clean up BST by releasing space back free store
template <typename Key, typename E>
void BST<Key, E>::
clearhelp(BSTNode<Key, E>* root) {     //此处使用了后序遍历，递归清空BST所占空间
  if (root == NULL) return;
  clearhelp(root->left());
  clearhelp(root->right());
  delete root;
}
// Insert a node into the BST, returning the updated tree
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::inserthelp(
    BSTNode<Key, E>* root, const Key& k, const E& it) {
  if (root == NULL)  // Empty tree: create node
    return new BSTNode<Key, E>(k, it, NULL, NULL);
  if (k < root->key())
    root->setLeft(inserthelp(root->left(), k, it));
//inserthelp将（k,it）对应结点插入根结点的左子树，并返回左子树根结点指针
  else root->setRight(inserthelp(root->right(), k, it));
  return root;       // Return tree with node inserted}
```

BST插入图示

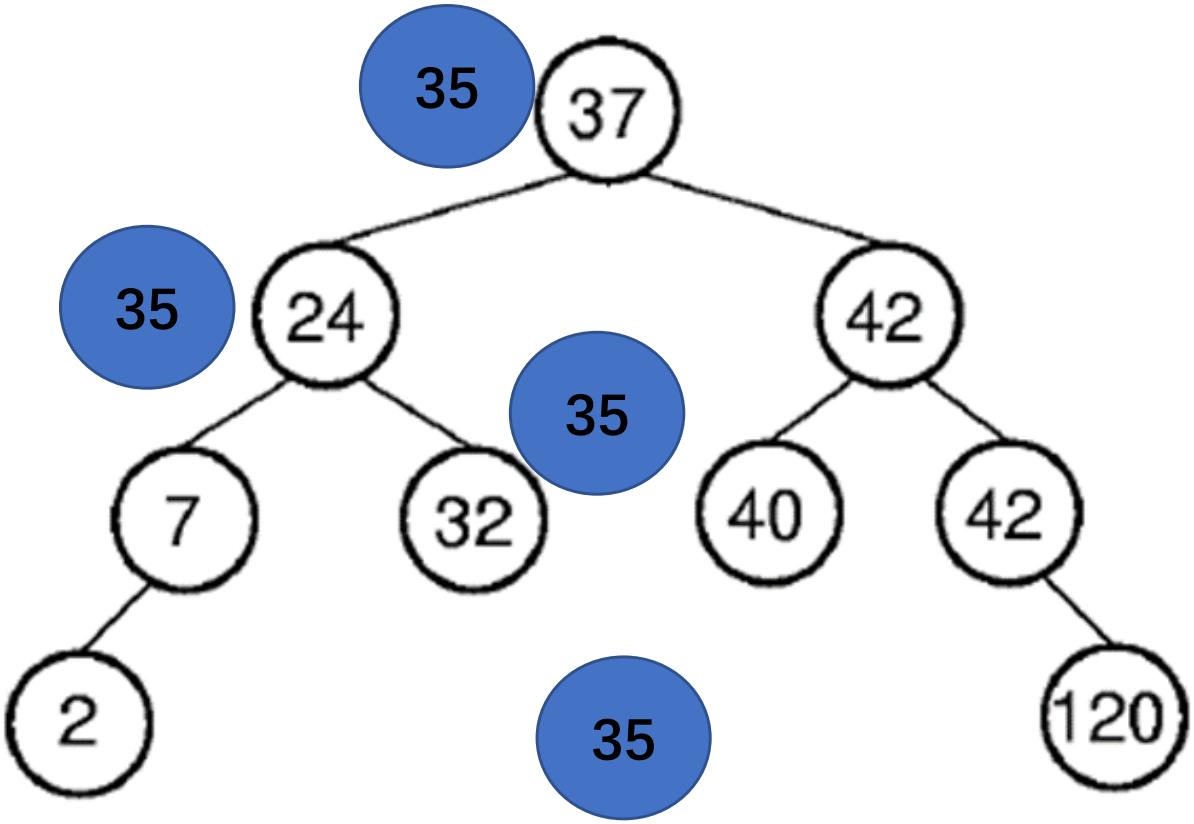


图5.15 在BST中插入值为35的结点

删除

从二叉检索树中删除一个任意的结点R，首先必须找到R，接着将它从二叉树中删除掉。分情况考虑：

（1）如果R是一个叶结点(没有儿子)，那么只要将R的父结点指向它的指针改为NULL就可以了。

（2）如果R是一个分支结点，就不能简单地删除这个结点，因为这样做会破坏树的连通性。

- 如果R只有一个儿子，就将R的父结点指向它的指针改为指向R的子结点就可以了。
- 如果R有两个儿子，为了保持二叉检索树的性质，可以用R的中序后继结点来代替它。该结点应该是其右子树中值最小的结点（在BST中是其右子树中最左的结点）。

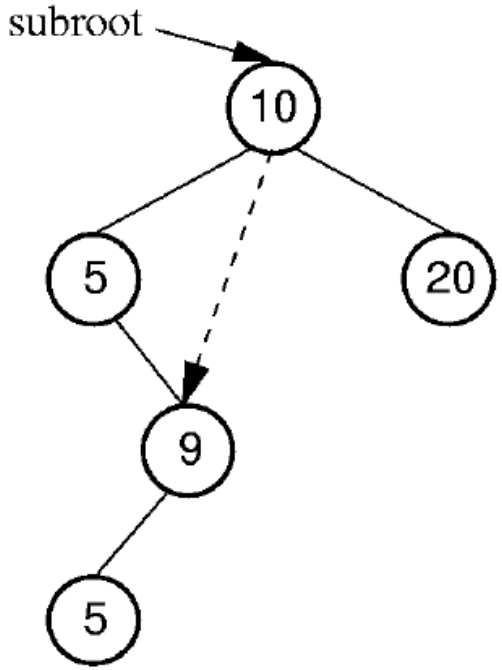
查找二叉检索树中最小值

```
// get the minimum value from the BST, 返回指向 BST最小值结点的指针
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::
getmin(BSTNode<Key, E>* rt) {
  if (rt->left() == NULL)//左子树为空时，BST中最小值在根结点
    return rt;
  else return getmin(rt->left());//分析：左子树不为空时，BST中最小值在左
   子树的最左的结点
}
```

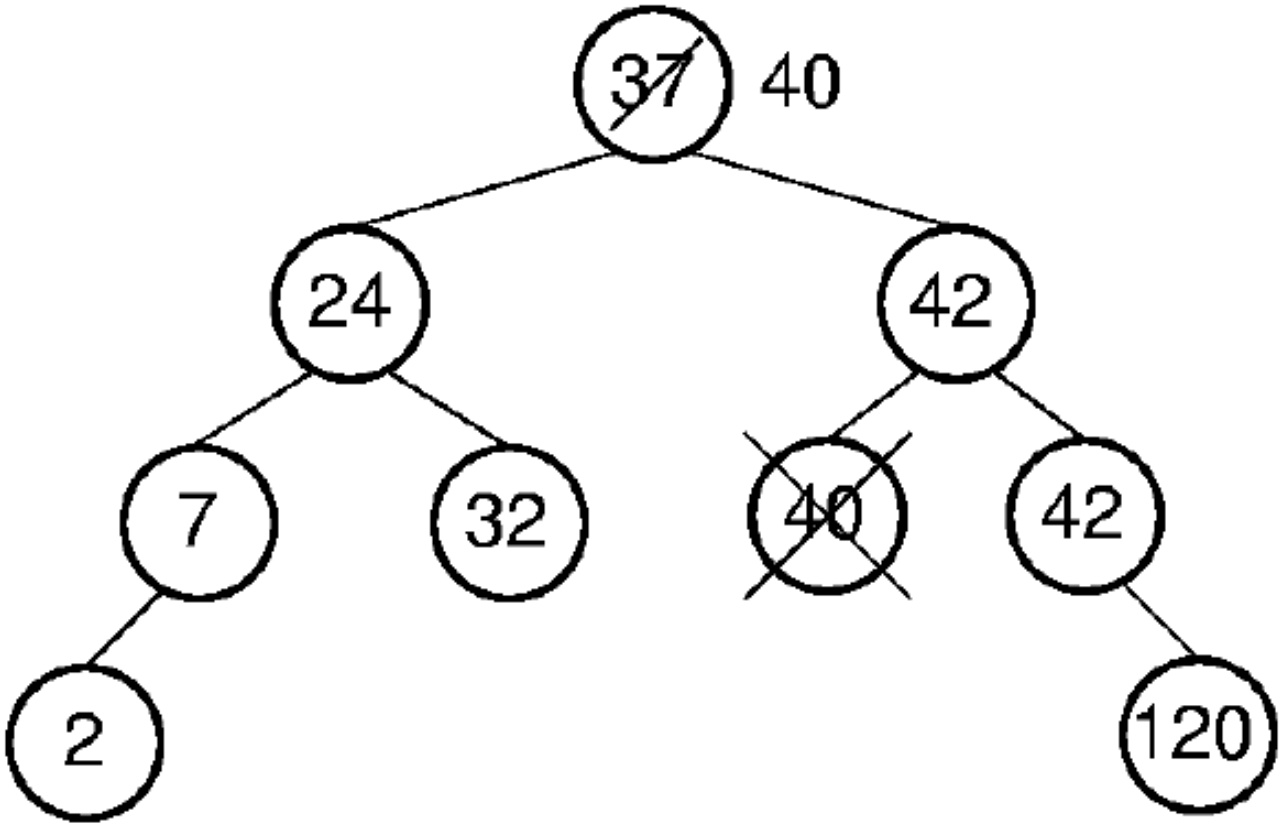
删除子树中最小值图示

```
// Delete the minimum value from the BST, returning the revised BST
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::
deletemin(BSTNode<Key, E>* rt) {
  if (rt->left() == NULL) // Found min
    return rt->right();//最小值在根结点 删除根结点
  else {                  // Continue left
    rt->setLeft(deletemin(rt->left()));
//左子树不为空时，BST中最小值在左子树的最左的结点
    return rt;  }
}
```

设置左孩子



删除37的图示



37有左右子树，删除后用其中序遍历的后继取代它，即右子树的最左的结点。

左子树最右的节点

BST Remove (1)

```
// Remove a node with key value k，  Return: The tree with the node removed

template <typename Key, typename E>

BSTNode<Key, E>* BST<Key, E>::

removehelp(BSTNode<Key, E>* rt, const Key& k) {

  if (rt == NULL) return NULL;    // k is not in tree

  else if (k < rt->key())

    rt->setLeft(removehelp(rt->left(), k));

//递归地在左子树上删除关键码为k的一个结点，并修改左子树根结点指针

  else if (k > rt->key())

    rt->setRight(removehelp(rt->right(), k));

//递归地在左子树上删除关键码为k的一个结点，并修改左子树根结点指针
```

BST Remove (2)

```
else {                          // Found: remove it
  BSTNode<Key, E>* temp = rt;
  if (rt->left() == NULL) {     // Only a right child
    rt = rt->right();           //  so point to right
    delete temp;
  }
  else if (rt->right() == NULL) { // Only a left child
    rt = rt->left();            //  so point to left
    delete temp;    }
```

BST Remove (3)

```
else {                    // Both children are non-empty
  BSTNode<Key, E>* temp = getmin(rt->right());
      //找到右子树的最小值结点
  rt->setElement(temp->element());//修改当前结点元素和关键字值为找到的最小值结点对应的元素值和键值
  rt->setKey(temp->key());
  rt->setRight(deletemin(rt->right()));
  //在右子树中删除最小值结点，并更新右子树根节点指针
  delete temp;    } }
return rt;}
```

输出BST

```
    // Print out a BST

template <typename Key, typename E>

void BST<Key, E>::

printhelp(BSTNode<Key, E>* root, int level) const {

  if (root == NULL) return;          // Empty tree

  printhelp(root->left(), level+1);   // Do left subtree

  for (int i=0; i<level; i++)         // Indent to level

    cout << "   ";

  cout << root->key() << "\n";        // Print node value

  printhelp(root->right(), level+1);  // Do right subtree}
```

课堂练习：BST的插入删除操作

（1）画出从空树开始，依次插入15、20、25、16、5 、3、7 和18的BST树。

（2）画出在BST中删除20后的BST树。

5.5 堆与优先队列

在实际应用中，存在许多需要从一群人、一系列任务或一些对象中寻找“下一位最重要”目标的情况，如：医院急诊的接诊、多任务操作系统的调度等。

一些按照重要性或优先级来组织的对象称为**优先队列**。普通队列不能有效实现优先队列，其插入和删除代价为Θ（n）；二叉检索树的插入和删除代价为Θ（logn），但有平衡性问题，极差情况下代价为Θ（n）。针对这种特殊的应用，为实现有效操作引入数据结构——堆。

堆的特性

1.堆是一棵**完全二叉树**，因此可用**数组**表示完全二叉树的方法来表示堆；

2.堆是**局部有序**的（不同于BST），结点的值与其子结点的值之间存在某种关系，分为**最小堆**和**最大堆**两种。

最大值堆

定义:

对于一个关键码序列{ K_0, K_1, …, K_{n-1}}，如果满足$K_i \geq K_{2i+1}$，$K_i \geq K_{2i+2}$，(i=0,1,…,n/2-1),则称其为堆，而且这是最大值堆。

性质:

是任意一个结点的值都大于或者等于其任意一个子结点存储的值。

分析:

由于根结点包含大于或等于其子结点的值，而其子结点又依次大于或等于各自子结点的值，所以根结点存储着该树所有结点中的最大值。（对每棵子树也如此）

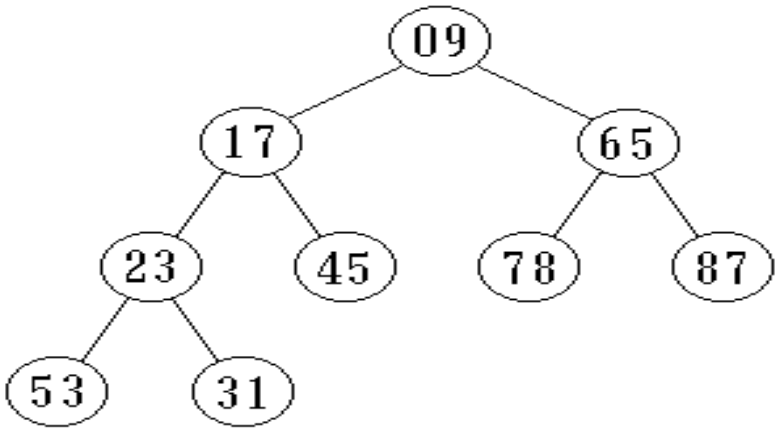
应用：7.6堆排序

最小值堆

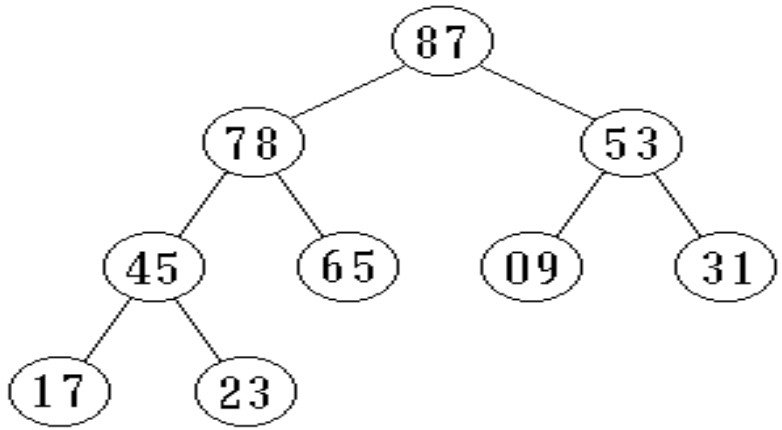
定义:

对于一个关键码序列{ K_0, K_1, …, K_{n-1}}，如果满足$K_i \le K_{2i+1}$，$K_i \le K_{2i+2}$，(i=0,1,…,n/2-1),则称其为堆，而且这是最小值堆。

- **最小值堆(min-heap)的性质是每一个结点存储的值都小于或等于其子结点存储的值。由于根结点包含小于或等于其子结点的值，而其子结点又依次小于或等于各自子结点的值，所以根结点存储了该树所有结点的最小值。（对每棵子树也如此）**
- **[illegible]**



（a）最小堆



（b）最大堆

堆

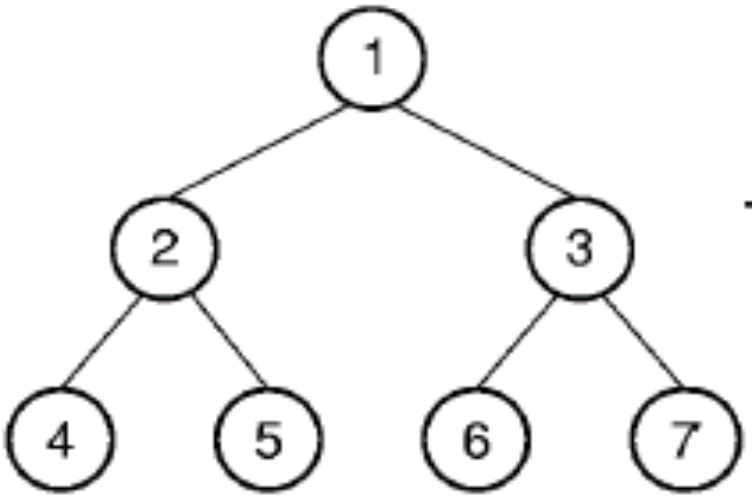
- 逻辑上是一种树形结构，是一棵完全二叉树；
- 典型的实现形式：
  - 数组
    - 通过数组的下标来表示结点在堆中的逻辑位置

最大值堆的实现

```
// Heap class
template <typename E, typename Comp> class heap {
private:
  E* Heap;         // Pointer to the heap array
  int maxsize;       // Maximum size of the heap
  int n;             // Number of elements now in the heap

  // Helper function to put element in its correct place
  void siftdown(int pos) {//下推
    while (!isLeaf(pos)) { // Stop if pos is a leaf
      int j = leftchild(pos);  int rc = rightchild(pos);
      if ((rc < n) && Comp::prior(Heap[rc], Heap[j]))
        j = rc;           // Set j to greater child's value
      if (Comp::prior(Heap[pos], Heap[j])) return; // 找到合适位置
      swap(Heap, pos, j);
      pos = j;           // 下移下推元素的位置
    }
```

不断跟左右孩子中值最大的那个交换位置直至找到合适的位置或者已经下推到叶子为止

下推举例



若需要构建最大堆，在上图中依次对结点3，2和1进行下推的过程如何？

最大值堆的实现

```
public:
heap(E* h, int num, int max)     // Constructor
  { Heap = h;  n = num;  maxsize = max;  buildHeap(); }
int size() const       // Return current heap size
  { return n; }
bool isLeaf(int pos) const // True if pos is a leaf，最后一个分支结点下标为n/2-1
  { return (pos >= n/2) && (pos < n); }
int leftchild(int pos) const
  { return 2*pos + 1; }    // Return leftchild position
int rightchild(int pos) const
  { return 2*pos + 2; }    // Return rightchild position
int parent(int pos) const  // Return parent position
  { return (pos-1)/2; }
void buildHeap()          // Heapify contents of Heap
  { for (int i=n/2-1; i>=0; i--) siftdown(i);//从最后一个分支结点到根结点逐一下推 }
```

最大值堆的实现

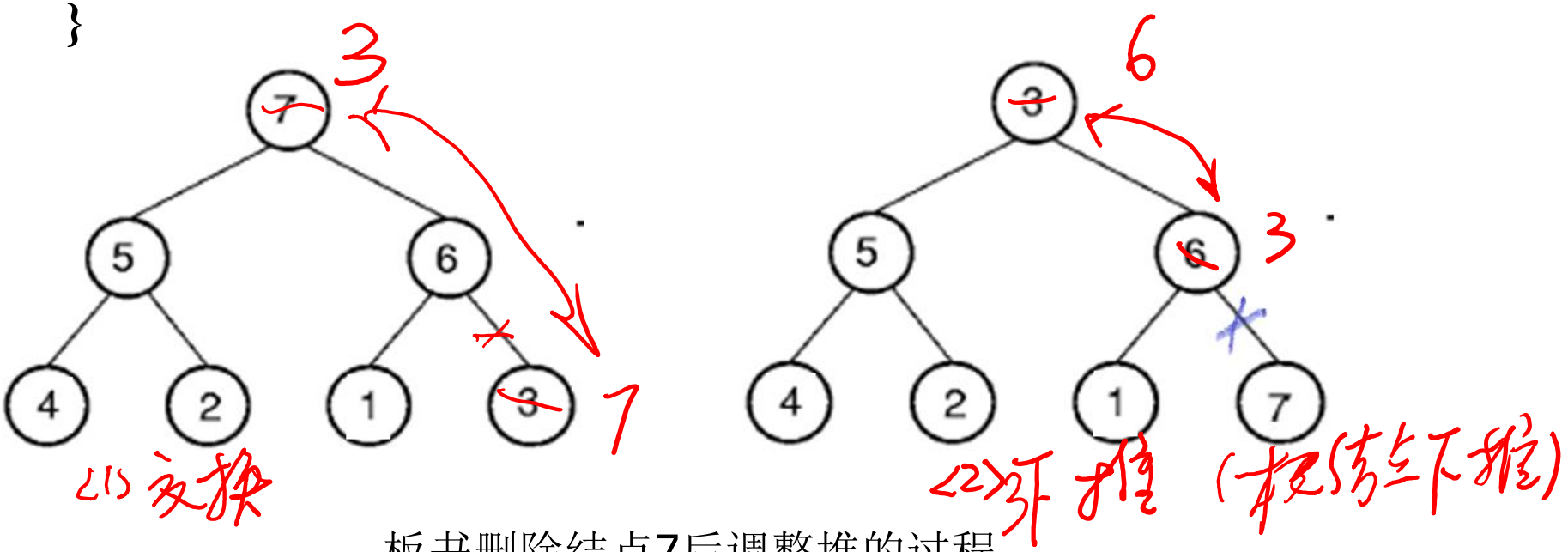
```
// Insert "it" into the heap
void insert(const E& it) {
  Assert(n < maxsize, "Heap is full");//参见附录的函数
  int curr = n++;  //指示当前插入元素位置，最后一个元素后
  Heap[curr] = it;              // 先插入在堆的末尾
  // 其他位置结点原来已构成了堆，只有新结点加入后，可能会违反堆的
  定义，需上推到 合适位置，即curr's parent > curr时或至根结点
  while ((curr!=0) &&
         (Comp::prior(Heap[curr], Heap[parent(curr)]))) {
    swap(Heap, curr, parent(curr));
    curr = parent(curr);
  }
}
```

注：逐一插入是一种建堆方法

插入结点8?

最大值堆的实现

```
// Remove first value
E removefirst() {//删除根结点的最大值
  Assert (n > 0, "Heap is empty");
  swap(Heap, 0, --n);       // Swap first with last value
  if (n != 0) siftdown(0);  // Siftdown new root val
  return Heap[n];           // Return deleted value
}
```



板书删除结点7后调整堆的过程

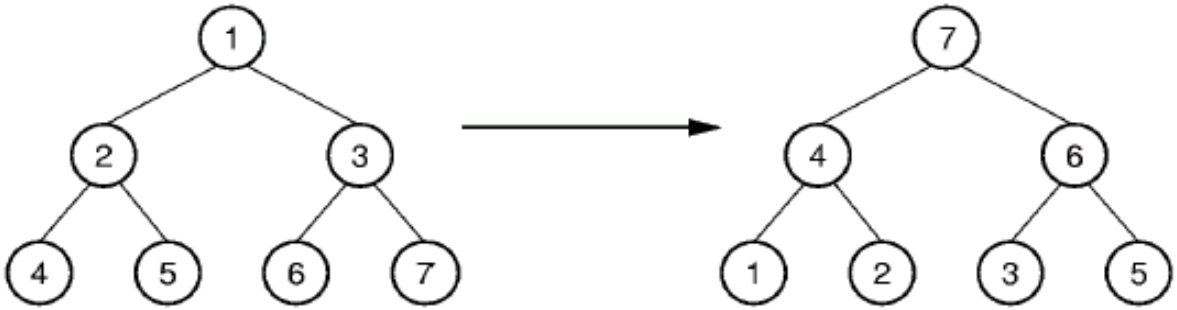
最大值堆的实现

```
// Remove and return element at specified position
E remove(int pos) {
  Assert((pos >= 0) && (pos < n), "Bad position");
  if (pos == (n-1)) n--; // Last element, no work to do，因为没有破坏堆
  else
  {
    swap(Heap, pos, --n);          // Swap with last value
    while ((pos != 0) &&
           (Comp::prior(Heap[pos], Heap[parent(pos)]))) {
      swap(Heap, pos, parent(pos)); // Push up large key
      pos = parent(pos);
    }
    if (n != 0) siftdown(pos);     // Push down small key
  }
  return Heap[n]; }};
```

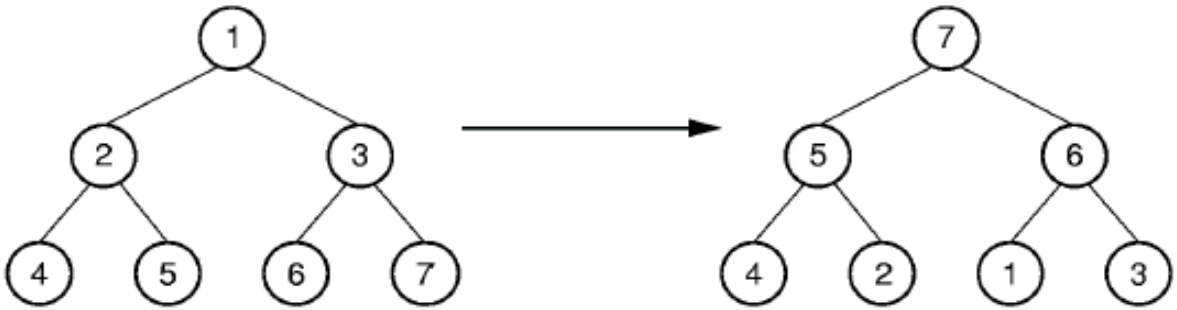
有可能出现原来的最后一个值较大的情况

思考:
这里为什么要先上推再下推?
上推和下推可能同时发生吗?

如何交换效率更高?
——建堆图示



(a)



(b)

(a) (4-2) (4-1) (2-1) (5-2) (5-4) (6-3) (6-5) (7-5) (7-6)
(b) (7-3), (5-2), (7-1), (6-1)

分析：

- 对于给定的一组数，建的堆不唯一；
- （b）比（a）的交换次数少

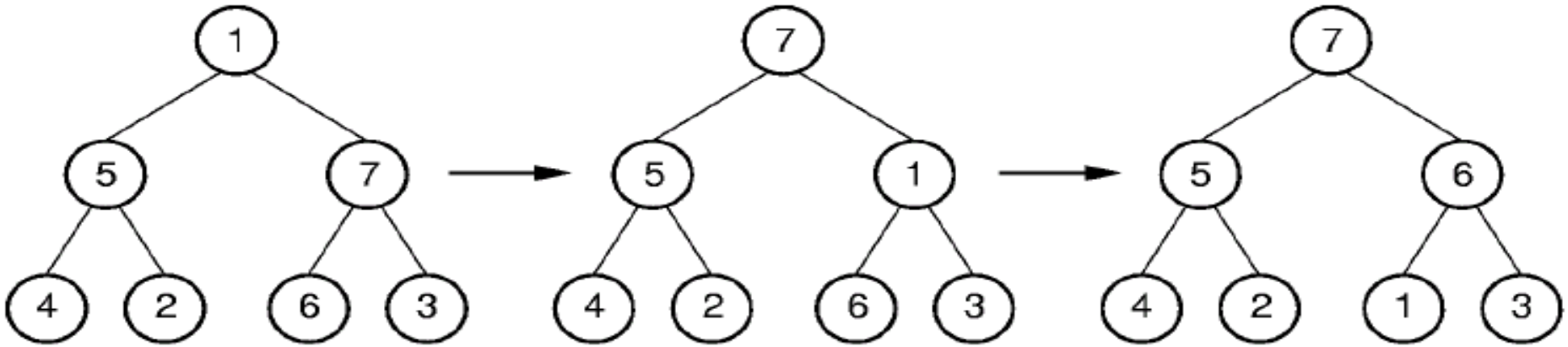
堆的形成

- 不必将值一个个地插入堆中，通过交换形成堆。
- 假设根的左、右子树都已是堆，并且根的元素名为R。能这种情况下，有两种可能：

  (1) R的值大于或等于其两个子女，此时堆已完成

  (2) R的值小于其某一个或全部两个子女的值，此时R应与两个子女中值较大的一个交换，结果得到一个堆，除非R仍然小于其新子女的一个或全部的两个。这种情况下，我们只需简单地继续这种将R“拉下来”的过程，直至到达某一个层使它大于它的子女，或者它成了叶结点。

Siftdown操作



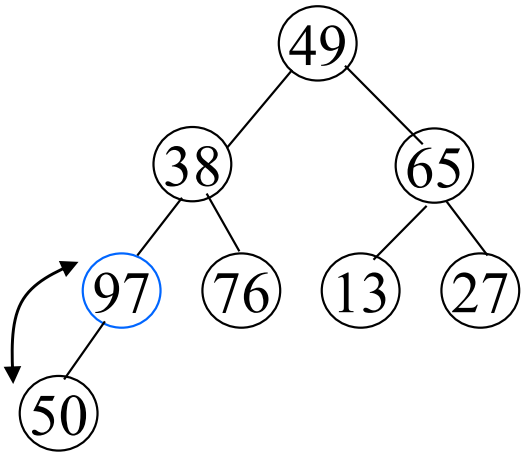
(a) (b) (c)

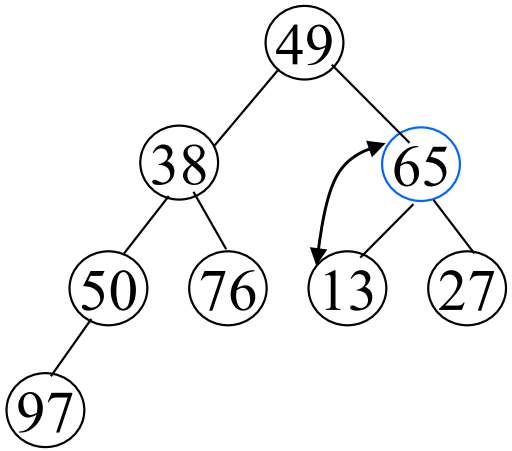
从堆的第一个分支结点（**heap[n/2-1]**，自底向上逐步把以各分支结点为根的子树调整成堆。

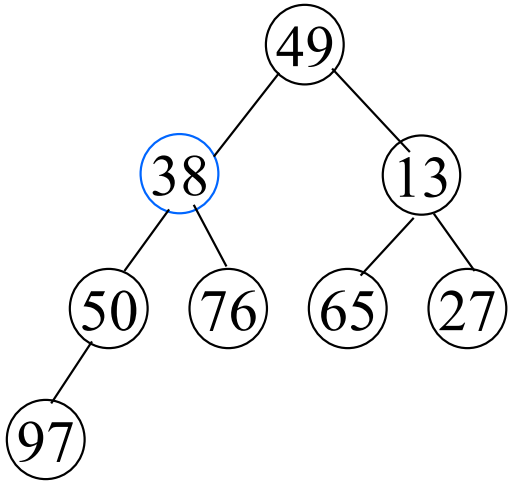
当调整到某一个位置**heap[pos]**时，由于其子树都已经是堆，

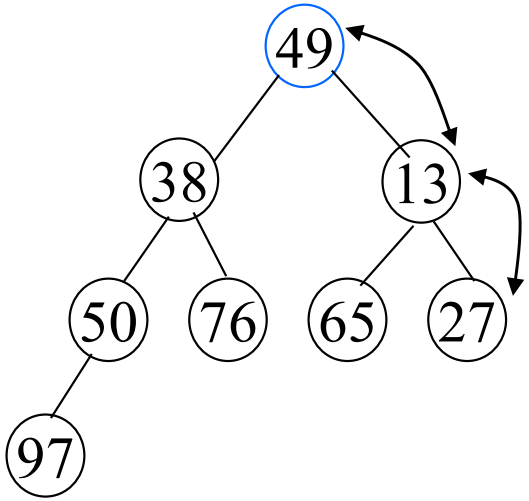
- 如果它也正好满足堆定义，则不用调整；
- 否则取其子树结点中较大的与**heap[pos]**交换；
- 交换后此子树根可能不满足堆定义（但其子树还是堆）；这样，可以继续一层层交换下去，最多到叶停止（过筛）

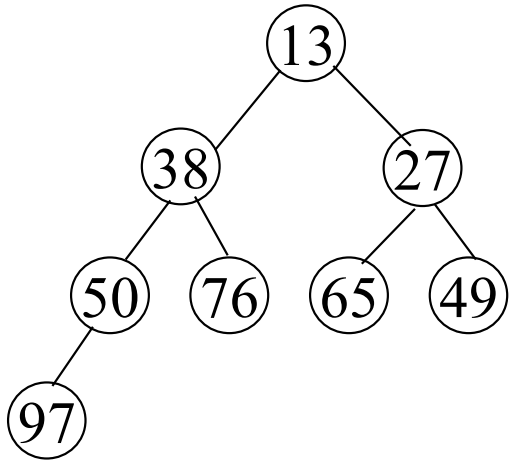
最小堆形成过程举例











最小堆的形成过程类似最大堆；
只不过下推时每次跟左右子结点中最小值交换

建堆操作的效率

n个结点的堆，高度d = floor($\log_2 n$ + 1)。根为第0层，则第i层结点个数为2^i，

考虑一个元素在堆中向下移动的距离。

- 大约一半的结点深度为d-1，不移动（叶）。
- 四分之一的结点深度为d-2，而它们至多能向下移动一层。
- 树中每向上一层，结点的数目为前一层的一半，而子树高度加一。因而元素移动的最大距离的总数为

$$\sum_{i=1}^{\log n} (i-1)\frac{n}{2^i} = o(n)$$

所以，这种算法时间代价为O(n)。

由于堆有log n层深，插入结点、删除普通元素和删除最大元素的平均时间代价和最差时间代价都是O(log n)。

Huffman编码树

哈夫曼编码——一种变长编码(VLC)，由Huffman于1952年提出，该方法依据字符出现概率来构造前缀不同的平均长度最短的码字。

1.固定长度编码

- 设所有代码都等长，则表示n个不同的代码需要$\log_2 n$位称为固定长度编码(a fixed-length coding scheme)。
- ASCII 码就是一种固定长度编码。
- 如果每个字符的使用频率相等的话，固定长度编码是空间效率最高的方法。

2.数据压缩和不等长编码

- 频率不等的字符

Z	K	F	C	U	D	L	E
2	7	24	32	37	42	42	120

数据压缩和不等长编码

- 可以利用字母的出现频率来编码，使得经常出现的字母的编码较短，反之不常出现的字母编码较长。
- 数据压缩既能节省磁盘空间，又能提高运算速度。（外存时空权衡的规则）
- 不等长编码是今天广泛使用的文件压缩技术的核心
- Huffman 编码是最简单的文件压缩技术，它给出了这种编码方法的思想。

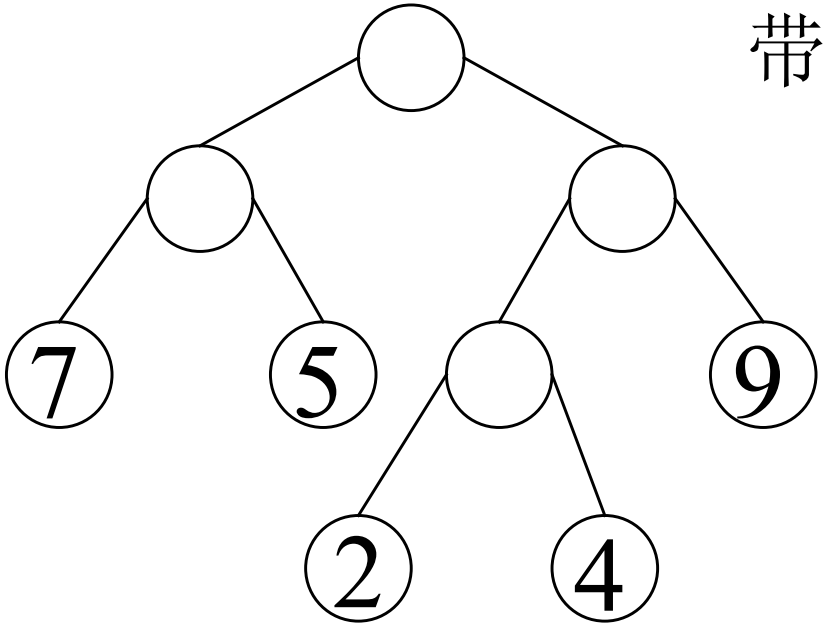
建立Huffman编码树

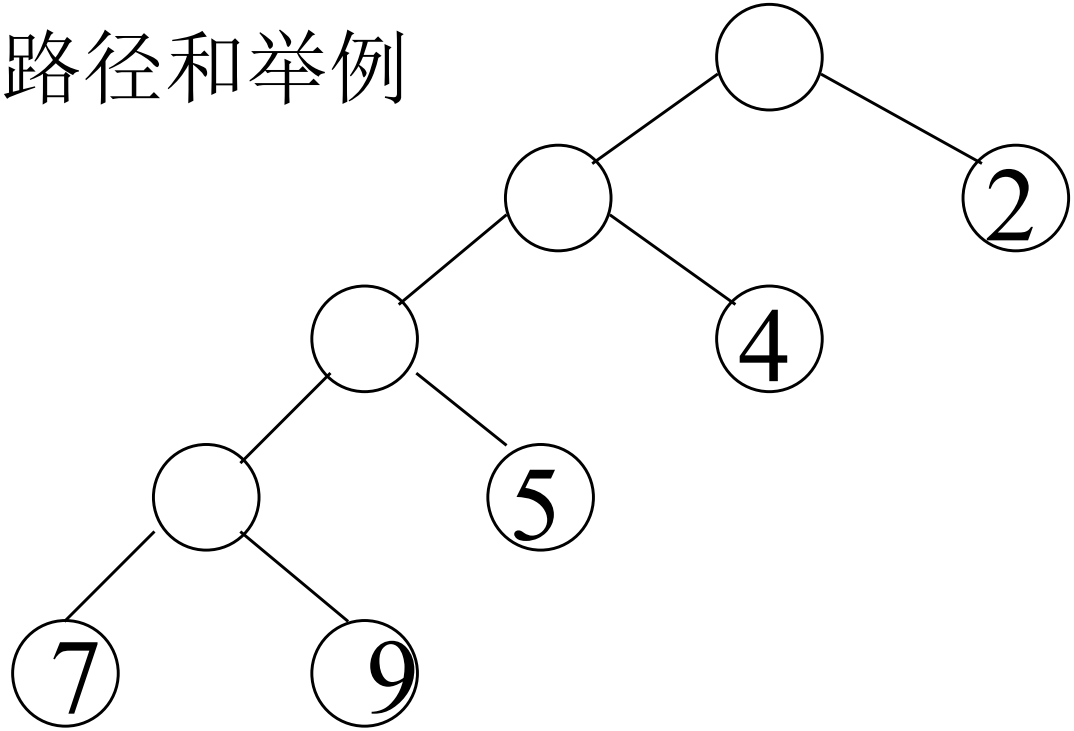
- 对于n个字符K_0，K_1，…，K_{n-1}，它们的使用频率分别为w_0，w_1，…，w_{n-1}，请给出它们的编码，使得总编码效率最高。
- 定义一个树叶的**带权路径长度**(Weighted path length)为权乘以它的路径长度(即树叶的深度）。
- 这个问题其实就是要求给出一个具有n个外部结点的扩充二叉树， 该二叉树每个外部结点K_i有一个w_i与之对应，作为该外部结点的权

这个扩充二叉树的叶结点带权外部路径长度总和 $\sum_{i=0}^{n-1} w_i \cdot l_i$ 最小(注意不管内部结点，也不用有序)。

权越大的叶结点离根越近；如果某个叶的权较小，可能就会离根较远。

带权路径和举例





WPL(T)=

7×2+5×2+2×3+

4×3+9×2

=60

不同的二叉树，带权路径和不一样

WPL(T)=

7×4+9×4+5×3+

4×2+2×1

=89

如何构造最优树

(哈夫曼算法)以二叉树为例：

(1) 根据给定的 n 个权值 $\{w_1, w_2, \cdots, w_n\}$，

构造 n 棵二叉树的集合

$F = \{T_1, T_2, \cdots, T_n\}$，

其中每棵二叉树中均只含一个带权值

为 w_i 的根结点，其左、右子树为空树；

如何构造最优树（续）

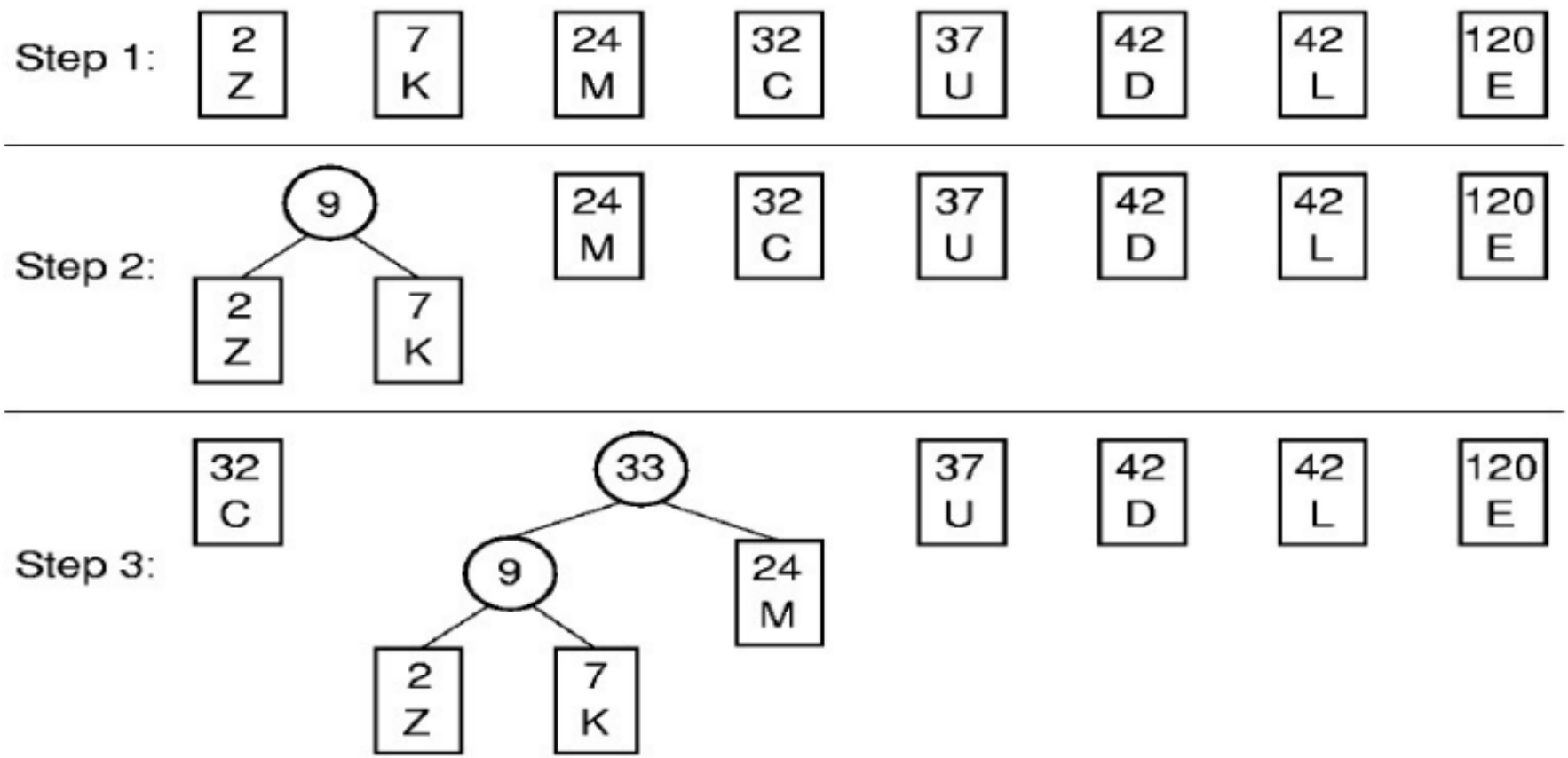
(2) 在F中选取其根结点的**权值为最小**的两棵二叉树，分别作为左、右子树构造一棵新的二叉树，并置这棵新的二叉树根结点的权值为其左、右子树根结点的权值之和；

（隐含了**排序**，最小的放左边，次小的放右边）

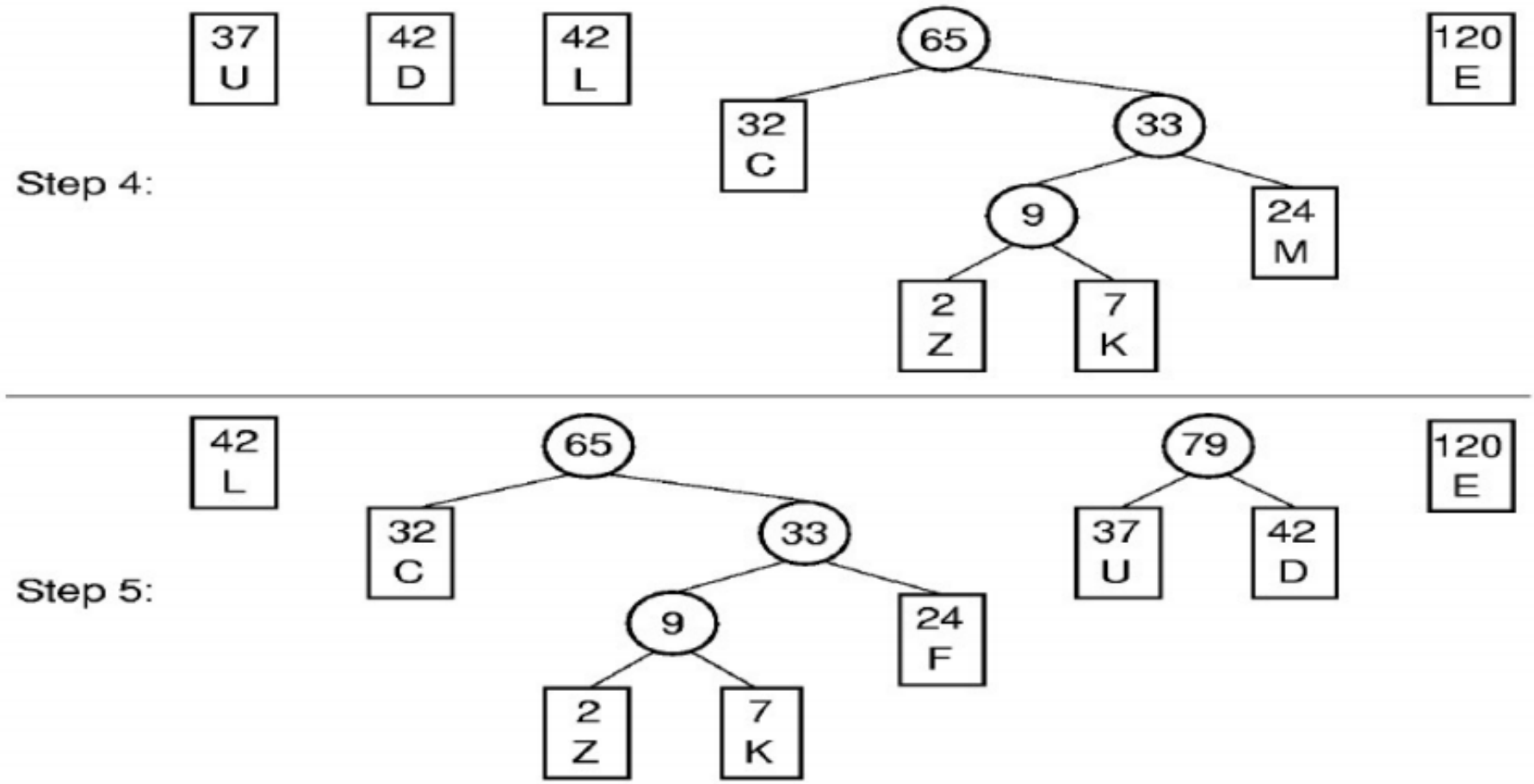
(3) 从F中删去这两棵树，同时加入刚生成的新树；

(4) 重复(2)和(3)两步，直至F中只含一棵树为止。

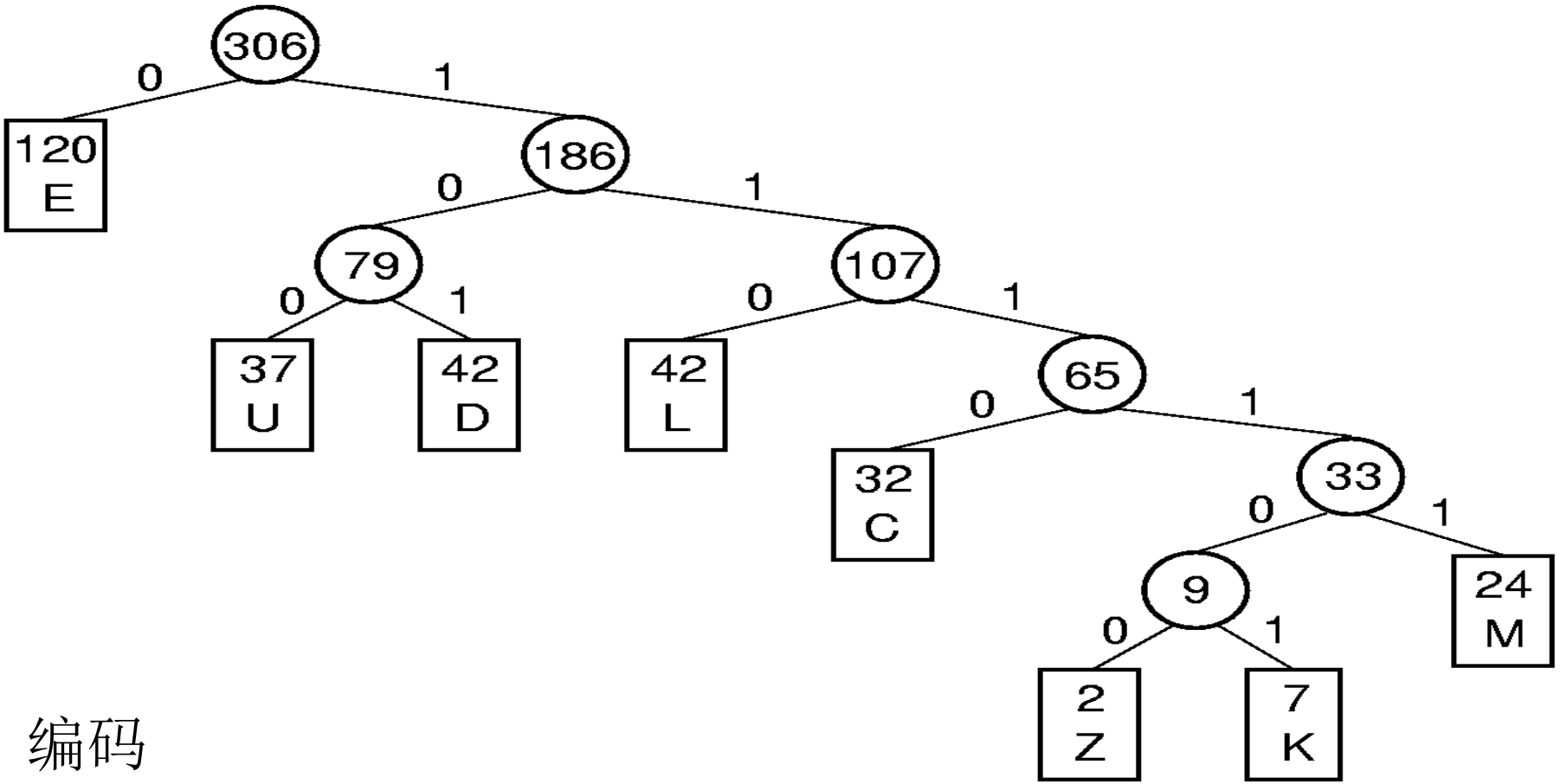
Huffman建树图示



Huffman建树图示



Huffman建树图示

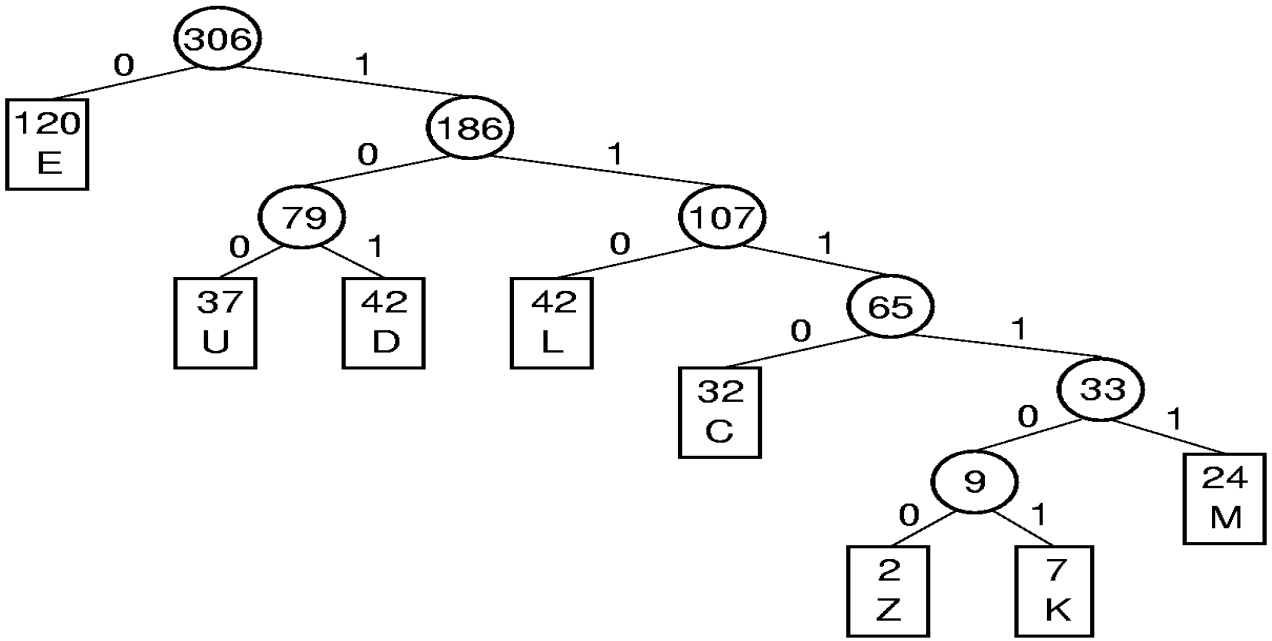


编码

1.给霍夫曼树的所有左分支标'0'，右分支标'1'。

2.从树根至树叶依次连接所有标记的字母可得相应字母的编码。

Huffman编码



字母	E	U	D	L	C	Z	K	M
编码	0	100	101	110	1110	111100	111101	11111

平均码长为=（1*120+3*（37+42+42）+4*32+5*24+6*（2+7））/306≈2.57

Huffman树结点的实现

```
template <typename E>
class HuffNode {          //Node abstract base class
public:
 virtual ~ HuffNode() { }
  virtual int weight() = 0;  //返回频率
  virtual bool isLeaf() = 0;
};};
```

Huffman树结点的实现——叶结点

```
template <typename Elem>   // Leaf node subclass
class LeafNode : public HuffNode<E>{
private:
  E it;
  int wgt;
public:
  LeafNode(const E& val, int freq)   // Constructor
    { it = val; wgt=freq); }
  int weight() { return wgt; }
  E val()  { return it; }
  bool isLeaf() { return true; }
};
```

Huffman树结点的实现——中间结点

```
template <typename E>  //Internal node subclass
class IntlNode : public HuffNode<E>{
private:
  HuffNode<E>* lc;    // Left child
  HuffNode<E>* rc;   //Right child
  int wgt;                        //Subtree weight,子树权值和
public:
  IntlNode(HuffNode<E>* l , HuffNode<E>* r)
     { wgt = l->weight( ) + r->weight( ) ; lc = l; rc = r; }
  int weight( ) { return wgt; }    //Return frequency
  bool isLeaf() { return false; }
  HuffNode<E>* left ( ) const { return lc; }
  void setLeft (HuffNode<E>* b) { lc = (HuffNode*)b; }
  HuffNode<E>* right( ) const { return rc; }
  void setRight (HuffNode<E>* b) { rc = (HuffNode*)b; }
};
```

Huffman树

```
template <typename E>
class HuffTree {
private:
  HuffNode<E>* Root;
public:
  HuffTree(E& val, int freq)
    { Root = new LeafNode<E>(val, freq); }
  HuffTree(HuffTree<E>* l, HuffTree<E>* r)
    { Root = new IntlNode<E>(l->root(), r->root()); }
  ~HuffTree() {}
  HuffNode<E>* root() { return Root; }
  int weight() { return Root->weight(); }
};
```

Huffman树的构建

用最小堆排序

```
template <typename E>HuffTree<E>*
buildHuff(HuffTree<E>** TreeArray, int count) {
   heap< HuffTree <E>*,minTreeComp>* forest=new heap
 < HuffTree <E>*,minTreeComp>(TreeArray,count,count);
  HuffTree<char> *temp1,*temp2,*temp3=NULL;
   while(forest.size( )>1){ //while at least two items left
         temp1 = forest ->removefirst();  //Pull first two trees
         temp2 =forest->removefirst();  //off the list
         temp3 = new HuffTree<E>(temp1, temp2 );
         forest ->insert(temp3);  //put the new tree back on  list
         delete temp1;  //Must delete the remnants
         delete templ2;  //of the trees we created
   }
  return temp3;//返回得到的哈夫曼树
}
```