第五章 二叉树

宋国杰

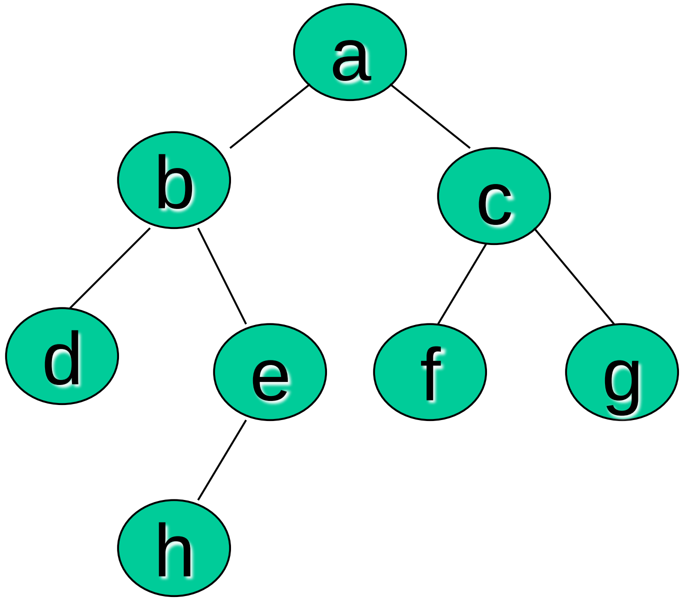
北京大学信息科学技术学院

课程内容

- 5.1 基本概念
- 5.2 周游二叉树
- 5.3 二叉树的存储结构
- 5.4 二叉搜索树
- 5.5 堆与优先队列
- 5.6 Huffman编码树及其应用

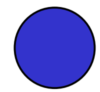
5.1 二叉树的概念

- 二叉树（**Binary Tree**）由结点的有限集合构成:
  - 或者为空集（**NIL**）
  - 或者由一个根结点及两棵不相交的分别称作左子树和右子树的二叉树组成
- 递归定义
  - 二叉树或为空集
  - 或左子树为空，或右子树为空
  - 或左右子树皆空

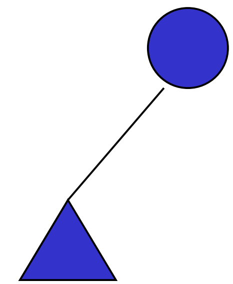


五种基本形态

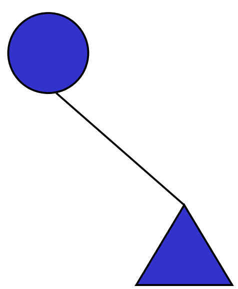
（a）空二叉树



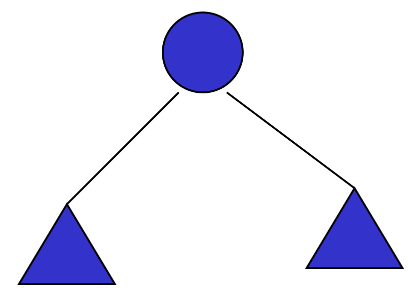
（b）根和空的左、右子树



（c）根和非空左子树、空右子树



（d）根和空左子树、非空右子树



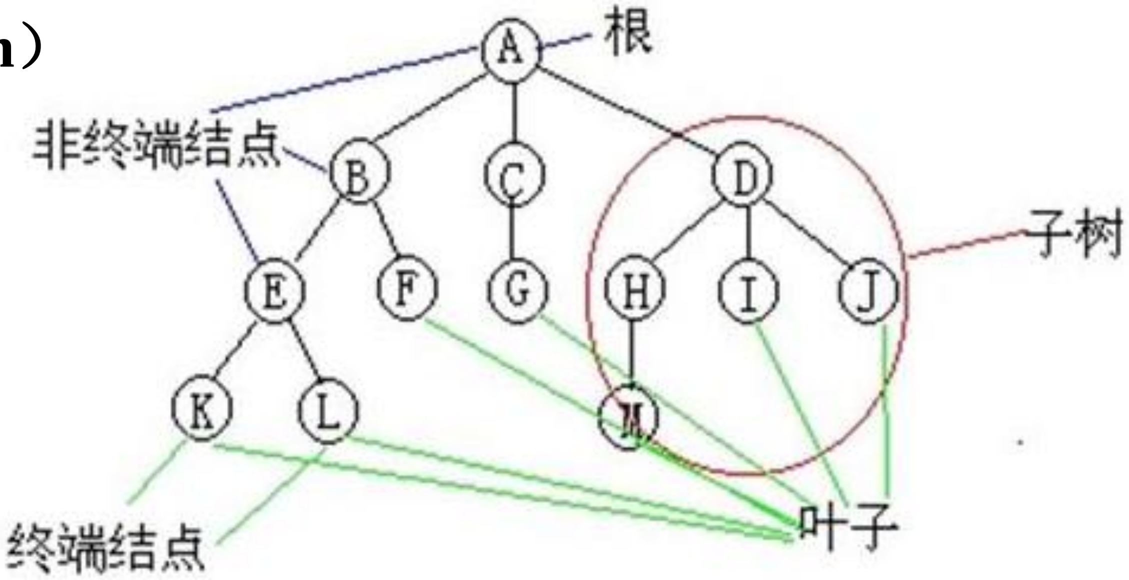
（e）根和非空的左、右子树

考虑一下

N个节点的树有多少种?

相关概念

- 父母（parent）
- 子女（孩子）（children）
- 边(edge)
- 兄弟(sibling)
- 路径(path)
- 祖先(ancestor)
- 子孙(descendant)
- 树叶(leaf)
- 内部节点或分支节点(internal node)
- 度数(degree)：节点子树的数目
- 层数(level)：根结点层数为0，其它节点层数等于父母层数加1

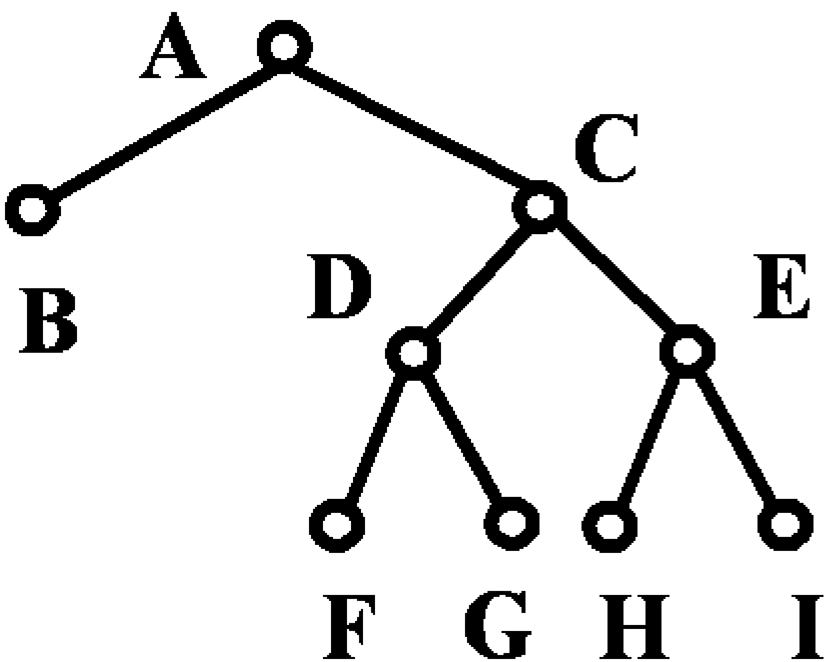


来个小测试

N个节点的树有多少条边?

满二叉树

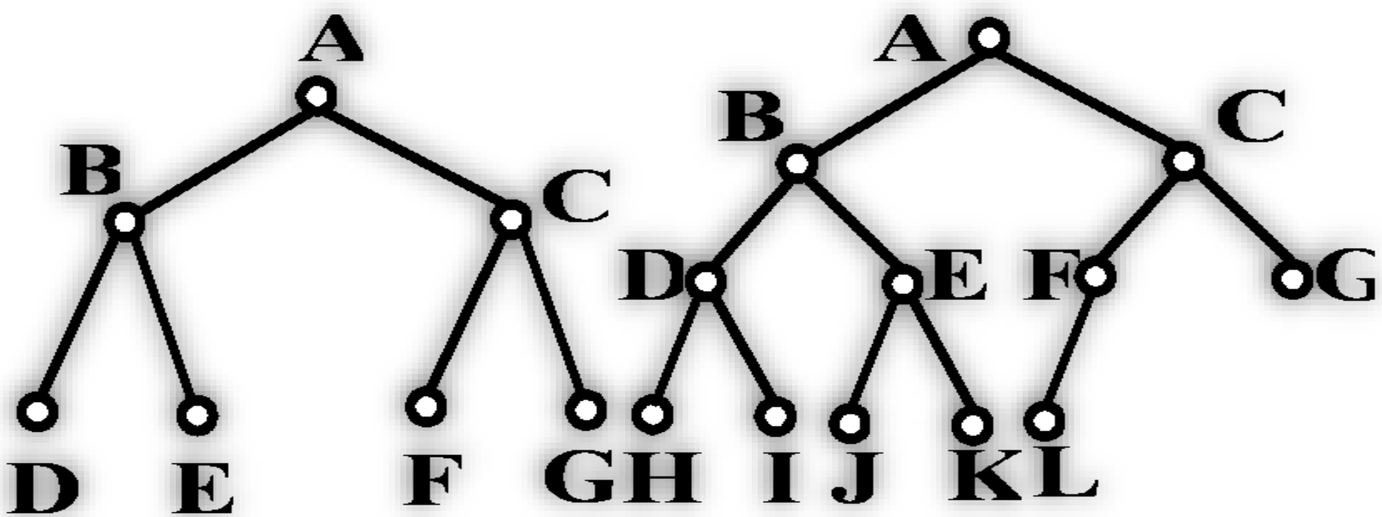
- 如果一棵二叉树的结点，或为树叶（0度节点），或有两棵非空子树（2度节点），则称满二叉树



1度节点个数为0

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
  - 最下面一层的结点都集中在该层最左边、连续的位置上
- 则称此二叉树为完全二叉树



完全二叉树的特点

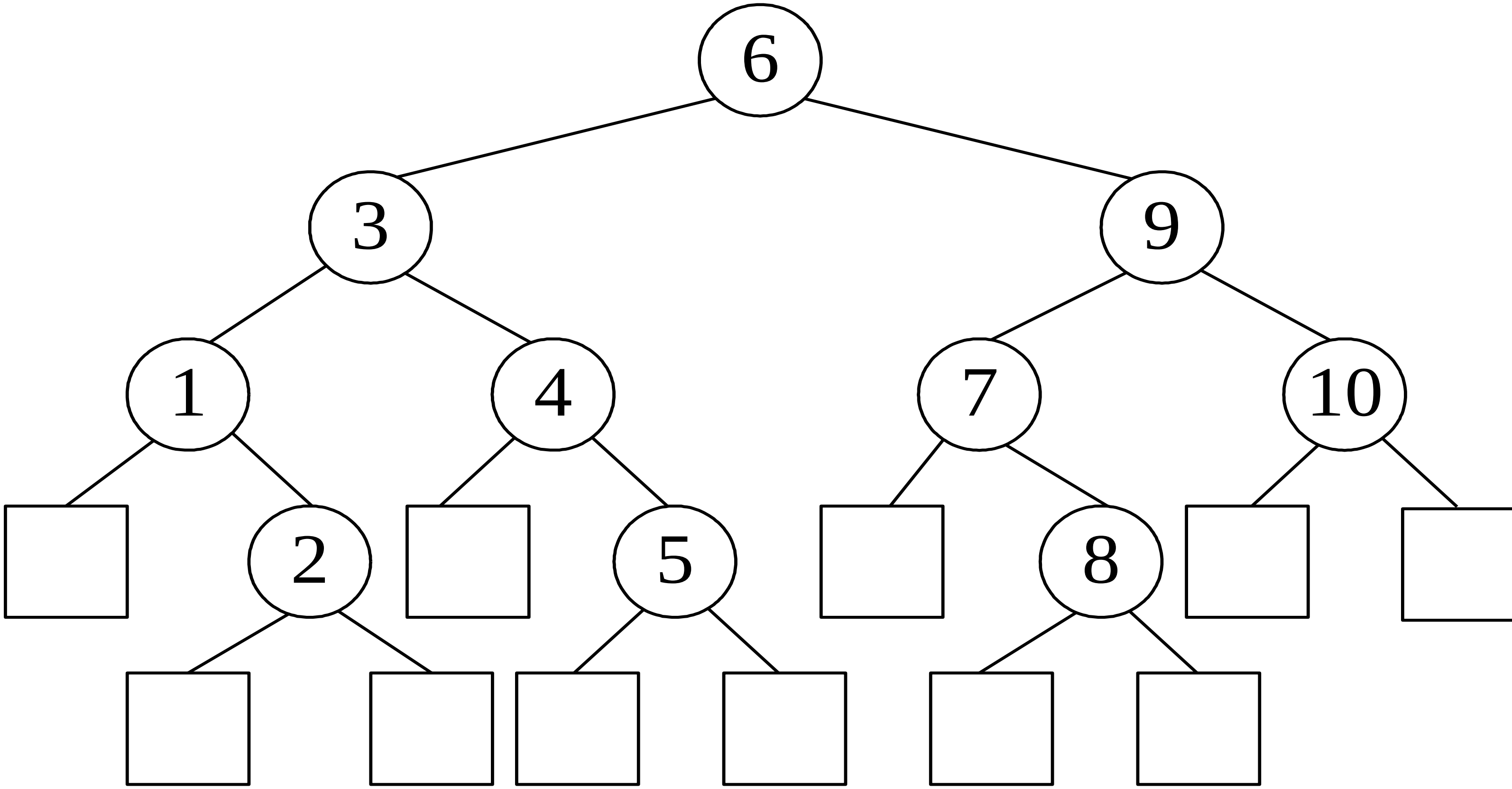
- 叶结点只可能在最下面两层出现
- 路径长度和最短（满二叉树不具有此性质）
  - **由根结点到二叉树中各结点路径长度的总和，在具有同样结点数的二叉树中最小**
  - 任意一棵二叉树中根结点到各结点的最长路径一定不短于结点数目相同的完全二叉树中的路径长度

扩充二叉树

- 当二叉树节点出现空指针时，就增加一个特殊结点——空树叶
  - 度为1的分支结点，在它下面增加1个空树叶
  - 度为0的叶结点，在它下面增加2个空树叶
- **扩充二叉树是满二叉树**
  - 新增加空树叶（外部结点N_0）的个数等于原来二叉树结点个数（内部结点N_2）加1

（后面证明：$N_0=N_2+1$）

扩充二叉树示例



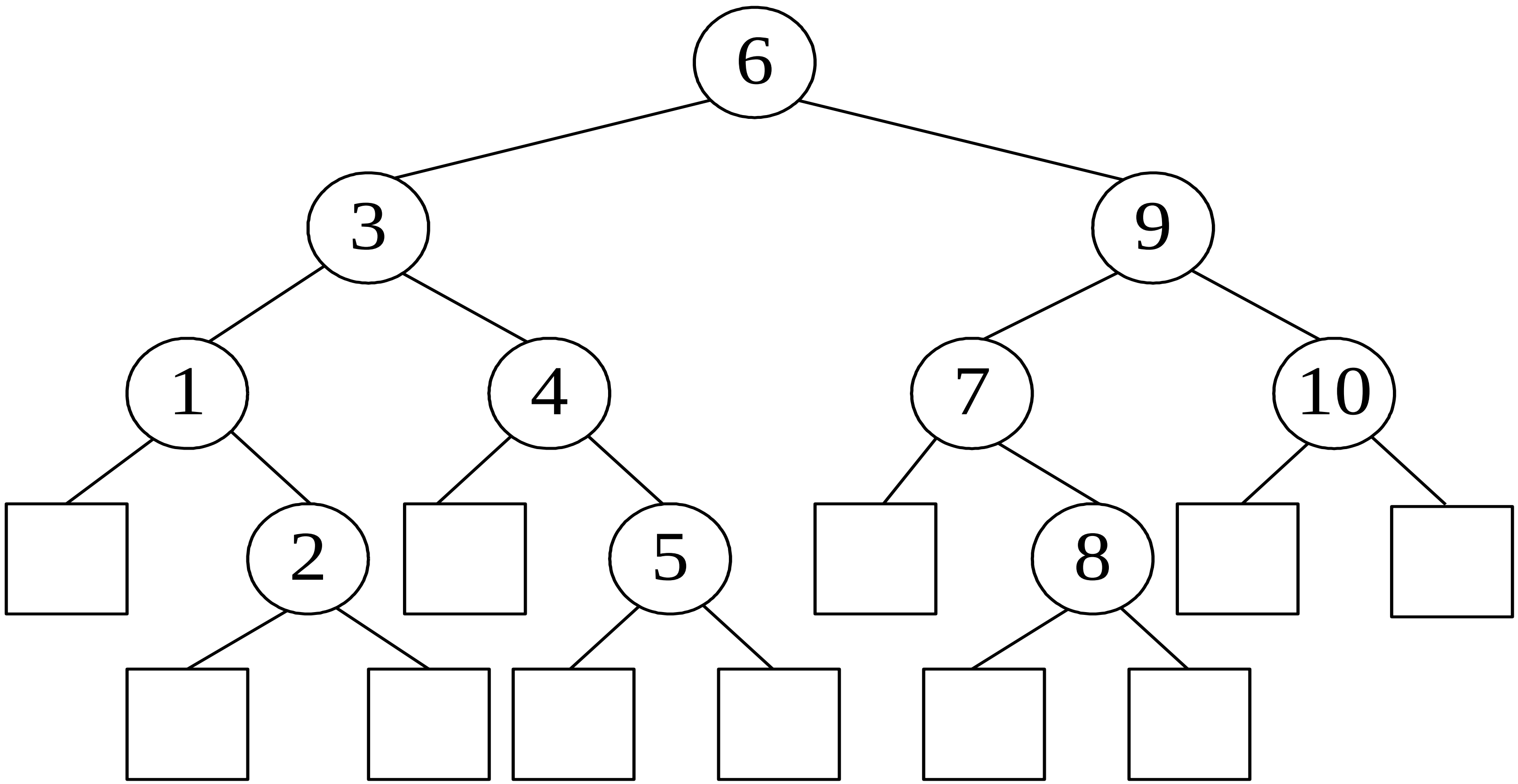
扩充二叉树性质

- 外部路径长度 E
  - 从扩充二叉树的根➡每个外部结点的路径长度之和
- 内部路径长度 I
  - 从扩充二叉树的根➡每个内部结点的路径长度之和
- E和I两个量之间的关系

$$E=I+2n$$

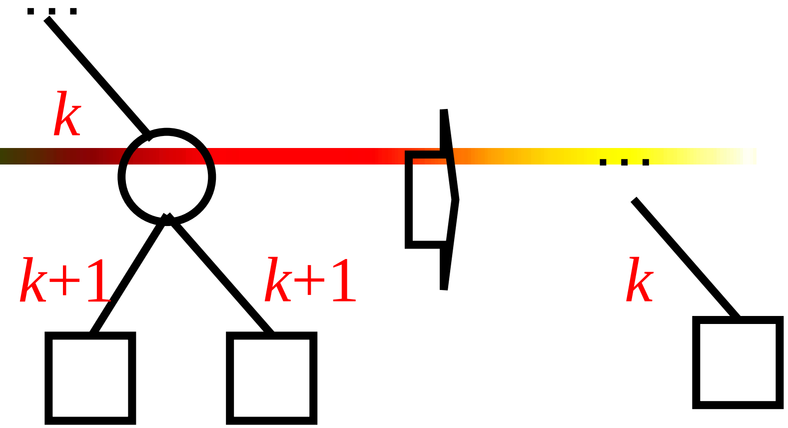
  - 其中，n是内部节点的个数

示例



- $E = 3 + 4 + 4 + 3 + 4 + 4 + 3 + 4 + 4 + 3 + 3 = 39$
- $I = 0 + 1 + 2 + 3 + 2 + 3 + 1 + 2 + 3 + 2 = 19$

证明



- 归纳法证明：E=I+2n
  - 当n=1时，I=0，E=2，等式成立
  - 若对于具有n个内部节点的扩充二叉树此等式也成立，即 $E_n=I_n+2n$。考虑有n+1个内部节点的扩充二叉树，删去一个作为原来二叉树树叶的路径长度为k的内部节点，使之成为一个有n个节点的二叉树。则有

$$I_n=I_{n+1}-k \Rightarrow I_{n+1}=I_n+k$$

$$E_n=E_{n+1}-2(k+1)+k=E_{n+1}-k-2 \Rightarrow E_{n+1}=E_n+k+2$$

  - 把前面等式带入进来得：

$E_{n+1}=(I_n+2n)+k+2=(I_n+k)+2n+2=I_{n+1}+2(n+1)$ 得证E=I+2n

二叉树的主要性质

- **性质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，即$n_0 = n_2 + 1$
- **性质2（满二叉树定理推论）**：一个非空二叉树的空子树(指针)数目等于其结点数加1
- **性质3**：任何一棵二叉树，度为0的结点n_0比度为2的结点n_2多1个，即$n_0 = n_2 + 1$
- **性质4**：二叉树的第i层（根为第0层，$i \geq 1$）最多有2^i个结点

满二叉树定理

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：设二叉树结点数为n，叶结点数为n_0，分支结点数为n_2，则

$$n = n_0 + n_2 \qquad \text{(公式1)}$$

∵ 边由分支节点产生，每个分支节点对应两条边，故有$2*n_2$条边。

而且，除根结点外，每个结点都恰有一条边连接父结点，故共有n-1条边，则有

$$n-1 = 2*n_2 \qquad \text{(公式2)}$$

∴由(公式1、公式2)得 $n-1= n_0+ n_2 -1 = 2*n_2$，得出

$$n_0 = n_2 + 1$$

满二叉树定理推论

性质2 满二叉树定理推论：一个非空二叉树的空子树数目等于其结点数加1

证明：

1. 设二叉树T，将其所有空子树换为树叶，记新的扩充满二叉树为T’。所有原来T的结点现在是T’的分支结点

2. 根据满二叉树定理，新添加的树叶数目等于T结点个数加1。而每个新添加的树叶对应T的一个空子树。 因此T中空子树数目等于T中结点数加1

再来个小测试

试证明，在具有n(n>=1)个结点的k叉树中，有n(k-1)+1个指针是空的。

$n_0 = n_2 + 1$

性质3 任何一颗二叉树，度为0的结点比度为2的结点多1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n_0，n_1，n_2，则有

$$n = n_0 + n_1 + n_2 \qquad \text{(公式3)}$$

设边数为e。除根节点外，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的的结点射出的，因此 $e = n_1 + 2*n_2$，于是

$$n = e + 1 = n_1 + 2 \cdot n_2 + 1 \qquad \text{(公式4)}$$

因此由公式3、4得$n_0 + n_1 + n_2 = n_1 + 2 \cdot n_2 + 1$，即

$$n_0 = n_2 + 1$$

性质5

- **二叉树高度定义为树的层数**
  - 即树中层数最大的叶结点的层数加1
- **二叉树深度定义为树中最长路径的长度**
  - 即树中层数最大的叶结点的层数
- **只有一个根结点的二叉树的高度为1，深度为0**
- **性质5 高度为 k 的二叉树至多有 2^k-1 个结点**

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

性质6

- **性质6：有n个节点（n>0）的完全二叉树的高度为 $\lceil \log_2 (n+1) \rceil$（深度为 $\lceil \log_2 (n+1) \rceil - 1$）**

证明： 设完全二叉树的高度为k，由定义可得：高为k的完全二叉树的前k-1层(0 ~ k-2)共有$2^{(k-1)}$-1 个结点。由于高度为k，故第k-1层上还有若干个结点，因此该完全二叉树的结点个数：

$$n > 2^{k-1} - 1$$

另一方面，由性质5可得：

$$n \le 2^{k} - 1$$

性质6(续)

由此可推出：$2^k \geq n+1 > 2^{k-1}$，取对数后有：

$$k \geq \log(n+1) > k-1$$

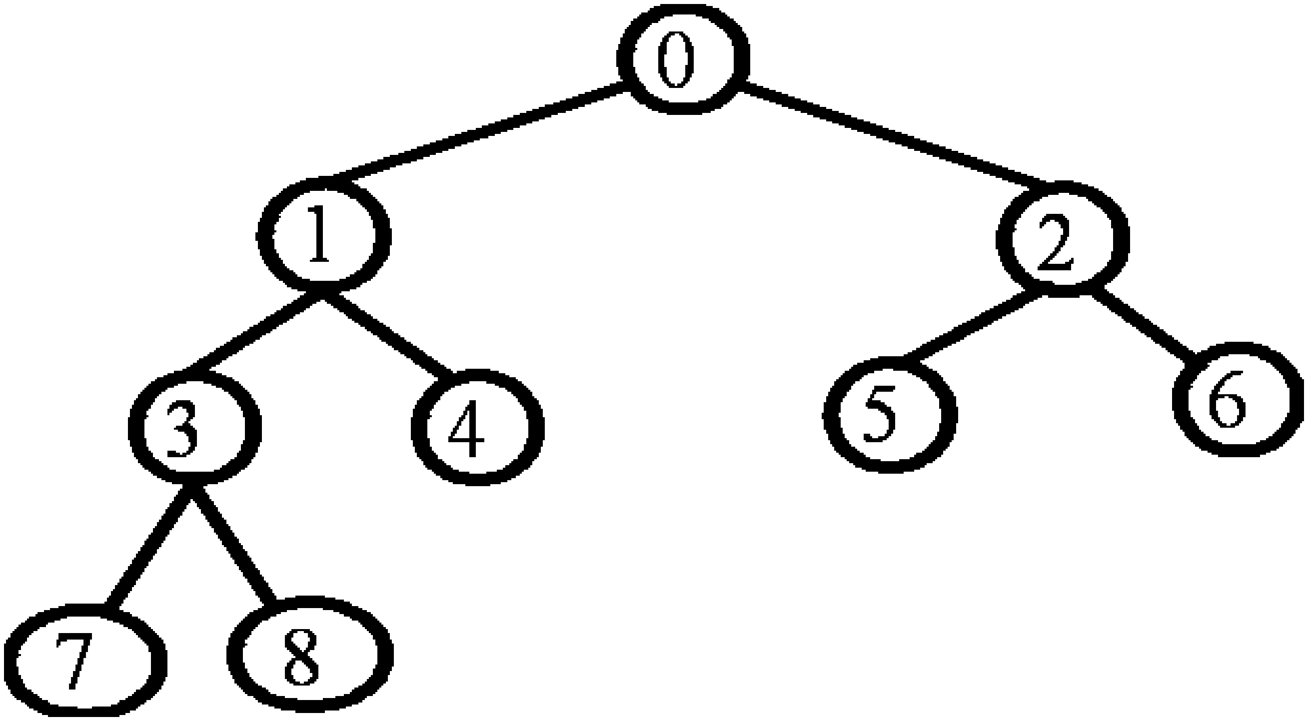
又因 **k-1** 和 **k** 是相邻的两个整数，故有

$$k = \lceil \log(n+1) \rceil$$

性质7

- **性质7. 对于具有n个结点的完全二叉树，结点按层次由左到右编号，则对任一结点 i（$0 \le i \le n-1$）有**
  - （1）如果i = 0，则结点i是二叉树的根结点；若i＞0，则其父结点编号是$\lfloor (i-1)/2 \rfloor$
  - （2）当$2i+1 \le n-1$时，结点i的左子结点是$2i+1$，否则结点i没有左子结点；当$2i+2 \le n-1$时，结点i的右子结点是$2i+2$，否则结点i没有右子结点
  - （3）当i为偶数且$0 < i < n$时，结点i的左兄弟是结点$i-1$，否则结点i无左兄弟；当i为奇数且$i+1 < n$时，结点i的右兄弟是结点$i+1$，否则结点i无右兄弟

- 按层次顺序将一棵有n个结点的完全二叉树的所有结点从0到n-1编号，就得到结点的一个线性序列



0	1	2	3	4	5	6	7	8

5.2 二叉树的周游

- 5.2.1 抽象数据类型ADT
- 5.2.2 深度优先周游二叉树
- 5.2.3 广度优先周游二叉树

5.2.1 抽象数据类型

- 结点存储数据信息，边保持结构信息
- 操作运算集中在访问二叉树的结点信息上
  - 例如：访问结点、左右子结点、父结点等
- 通过遍历实现对二叉树结点信息的访问

ADT: BinaryTreeNode

```
template <class T>
    class BinaryTreeNode  {
    friend class BinaryTree<T>;                    // 声明二叉树类为友元类
    private:
        T  info;                                   // 二叉树结点数据域
    public:
        BinaryTreeNode();                          // 缺省构造函数
        BinaryTreeNode(const T& ele);              // 给定数据的构造
        BinaryTreeNode(const T& ele, BinaryTreeNode<T> *l,
                       BinaryTreeNode<T> *r);      // 子树构造结点
```

```
    T  value() const;                              // 返回当前结点数据
    BinaryTreeNode<T>* leftchild() const;          // 返回左子树
    BinaryTreeNode<T>* rightchild() const;         // 返回右子树
    void  setLeftchild (BinaryTreeNode<T>*);       // 设置左子树
    void  setRightchild (BinaryTreeNode<T>*);      // 设置右子树
    void  setValue (const T& val);                 // 设置数据域
    bool  isLeaf() const;                          // 判断是否为叶结点
    BinaryTreeNode<T>& operator =
        (const BinaryTreeNode<T>& Node);           // 重载赋值操作符
};
```

ADT: BinaryTree

```
template <class T>
class BinaryTree  {
private:
    BinaryTreeNode<T>* root;                    //二叉树根结点
public:
     BinaryTree() {root = NULL;};               //构造函数
     ~BinaryTree() {DeleteBinaryTree(root);};   //析构函数
     bool isEmpty() const;                      //判定二叉树是否为空树
     BinaryTreeNode<T>* Root() {return root;};  //返回根结点
```

```
    BinaryTreeNode<T>* Parent(BinaryTreeNode<T> *current);    //返回父
    BinaryTreeNode<T>* LeftSibling(BinaryTreeNode<T> *current);//左兄
    BinaryTreeNode<T>* RightSibling(BinaryTreeNode<T>*current);//右兄
    void CreateTree(const T& info,
         BinaryTree<T>& leftTree, BinaryTree<T>&  rightTree);   // 构造树
    void PreOrder(BinaryTreeNode<T> *root);    // 前序遍历二叉树
    void InOrder(BinaryTreeNode<T> *root);     // 中序遍历二叉树
    void PostOrder(BinaryTreeNode<T> *root);   // 后序遍历二叉树
    void LevelOrder(BinaryTreeNode<T> *root);  // 按层次遍历二叉树
    void DeleteBinaryTree(BinaryTreeNode<T> *root); // 删除二叉树
};
```

遍历二叉树

- **遍历（Traversal）,也称“周游”**
  - 按照**一定的次序（规律）**访问二叉树中的结点
  - 每个结点被**访问（输出，修改等）**一次
- **二叉树的线性化**
  - 实质是把二叉树结点放入一个线性序列的过程
  - **“非线性”→“线性”**的过程
- **线性化方式，是对非线性结构的访问方式**
  - **深度优先**：一棵一棵子树的**纵深遍历**
  - **广度优先**：一层一层的自左而右的**逐层遍历**

5.2.2 深度优先周游二叉树

- 变换根结点的周游顺序，可得到以下六种方案：
- **前序周游**
  - 访问根结点 → 前序周游左子树 → 前序周游右子树
  - 访问根结点 → 前序周游右子树 → 前序周游左子树
- **中序周游**
  - 中序周游左子树 → 访问根结点 → 中序周游右子树
  - 中序周游右子树 → 访问根结点 → 中序周游左子树
- **后序周游**
  - 后序周游左子树→ 后序周游右子树 → 访问根结点
  - 后序周游右子树→ 后序周游左子树 → 访问根结点

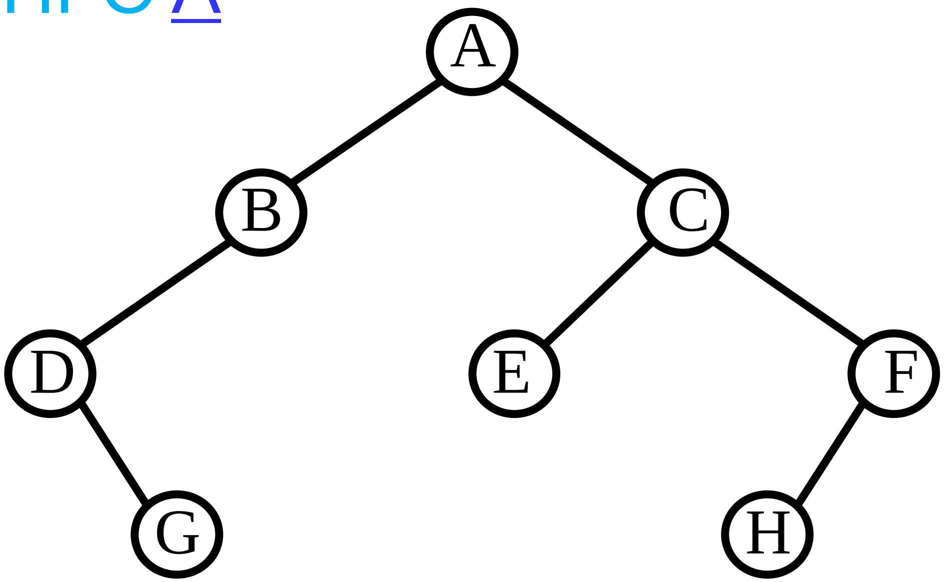
根节点遍历时机的决定性

子树遍历结果的连续性

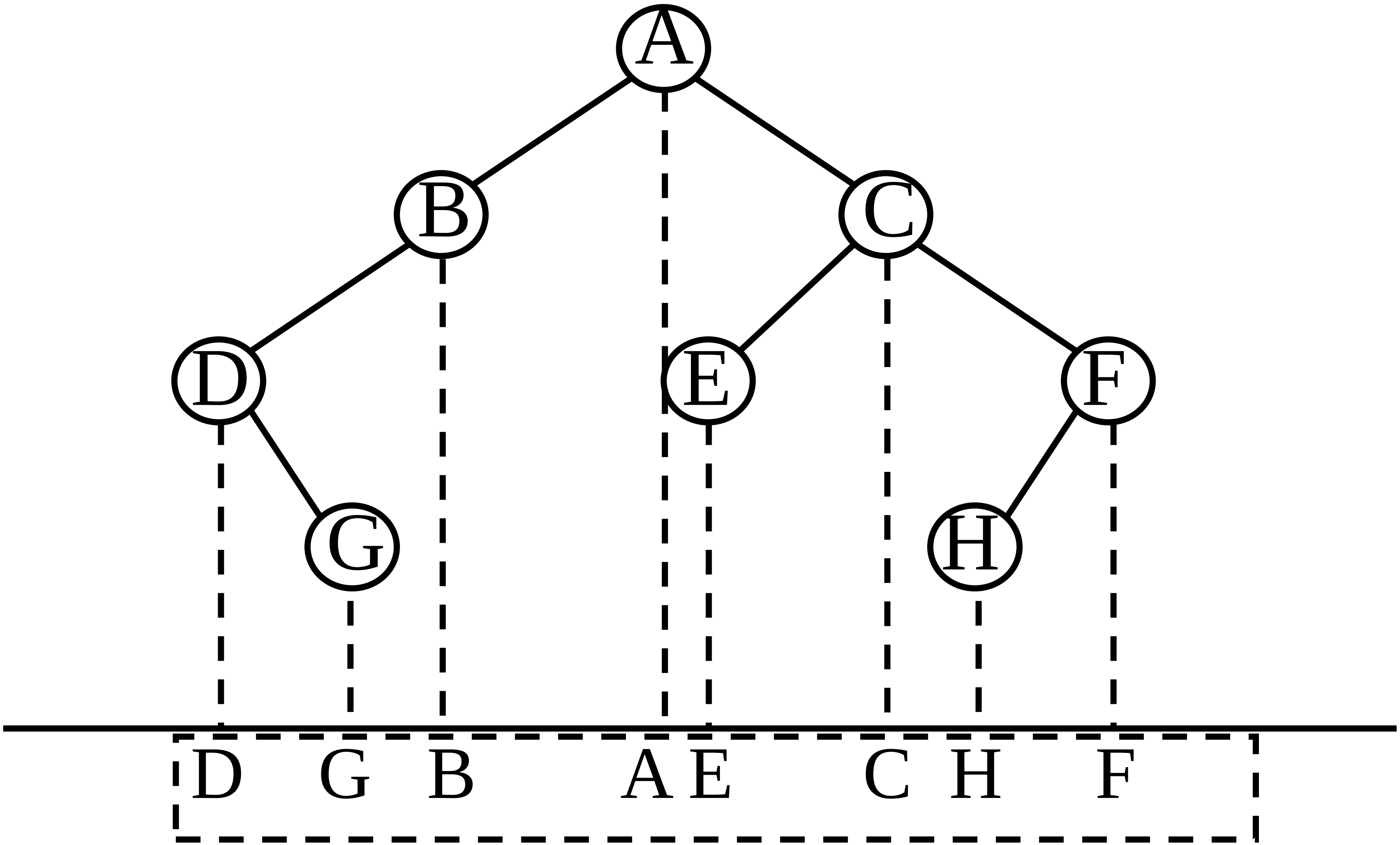
遍历过程的递归性

举例

- **前序周游：** A BDG CEFH
- **中序周游：** DGB A ECHF
- **后序周游：** GDB EHFC A



中序周游

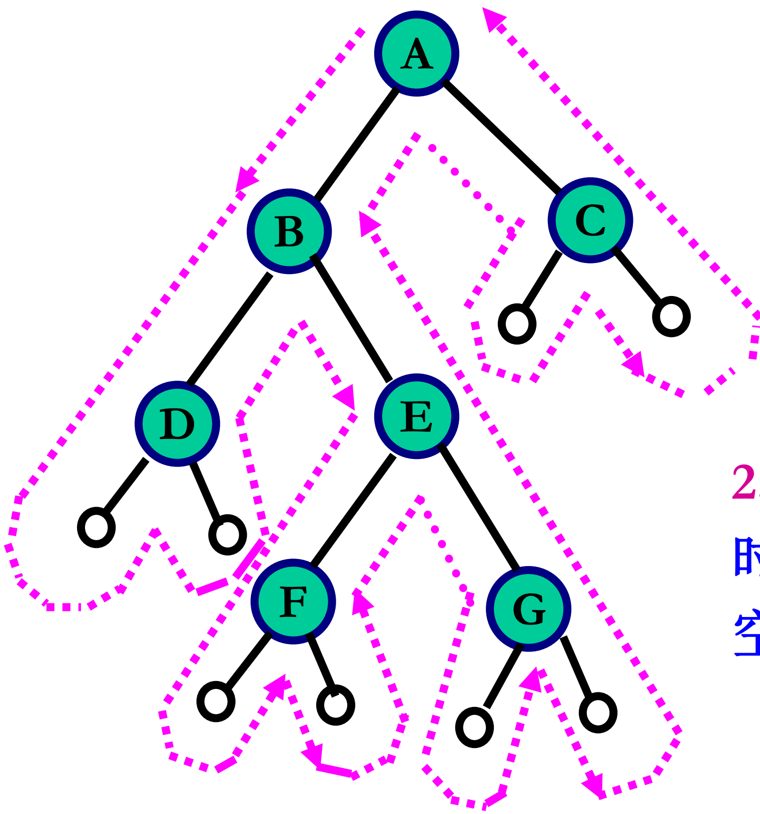


深度优先周游二叉树（递归实现）

```
template<class T>
void BinaryTree<T>::DepthOrder (BinaryTreeNode<T>* root)
 {
   if(root!=NULL) {
            Visit(root);                          // 前序
            DepthOrder(root->leftchild());        // 递归访问左子树
            Visit(root);                          // 中序
            DepthOrder(root->rightchild());       // 递归访问右子树
            Visit(root);                          // 后序
       }
}
```

对遍历的分析

1. 从递归遍历算法可知：如果将Visit(root)语句抹去，从递归的角度看，这三种算法完全相同，或者说三种遍历算法的访问路径是相同的，只是访问结点的时机不同



从虚线出发到终点的路径上，每个结点经过3次。

第1次经过时访问，是先序遍历

第2次经过时访问，是中序遍历

第3次经过时访问，是后序遍历

2. 二叉树遍历的时间\空间复杂性

时间复杂性：O(n) //结点线性访问次数

空间复杂性：O(logn) //栈占用的辅助空间

精确值：树深为k的递归遍历需要k+1个辅助单元

二叉树遍历的性质

- **性质1：已知二叉树的先序序列和中序序列，可以唯一确定一棵二叉树**
  - 推论：已知二叉树的后序序列和中序序列，可以唯一确定一棵二叉树
  - 请给出实现程序
- **性质2：已知二叉树的先序序列和后序序列，不能唯一确定一棵二叉树**

重构树：已知前序和中序遍历序列

I. 确定树的根节点

- 树根是当前树中所有元素在前序序列中的第一个元素

II. 求解树的子树

- 找出根节点在中序序列中的位置，根左边的所有元素就是左子树，根右边的所有元素就是右子树。
  - ✓ 若根节点左边或右边为空，则该方向子树为空
  - ✓ 若根节点左边和右边都为空，则根节点为叶节点

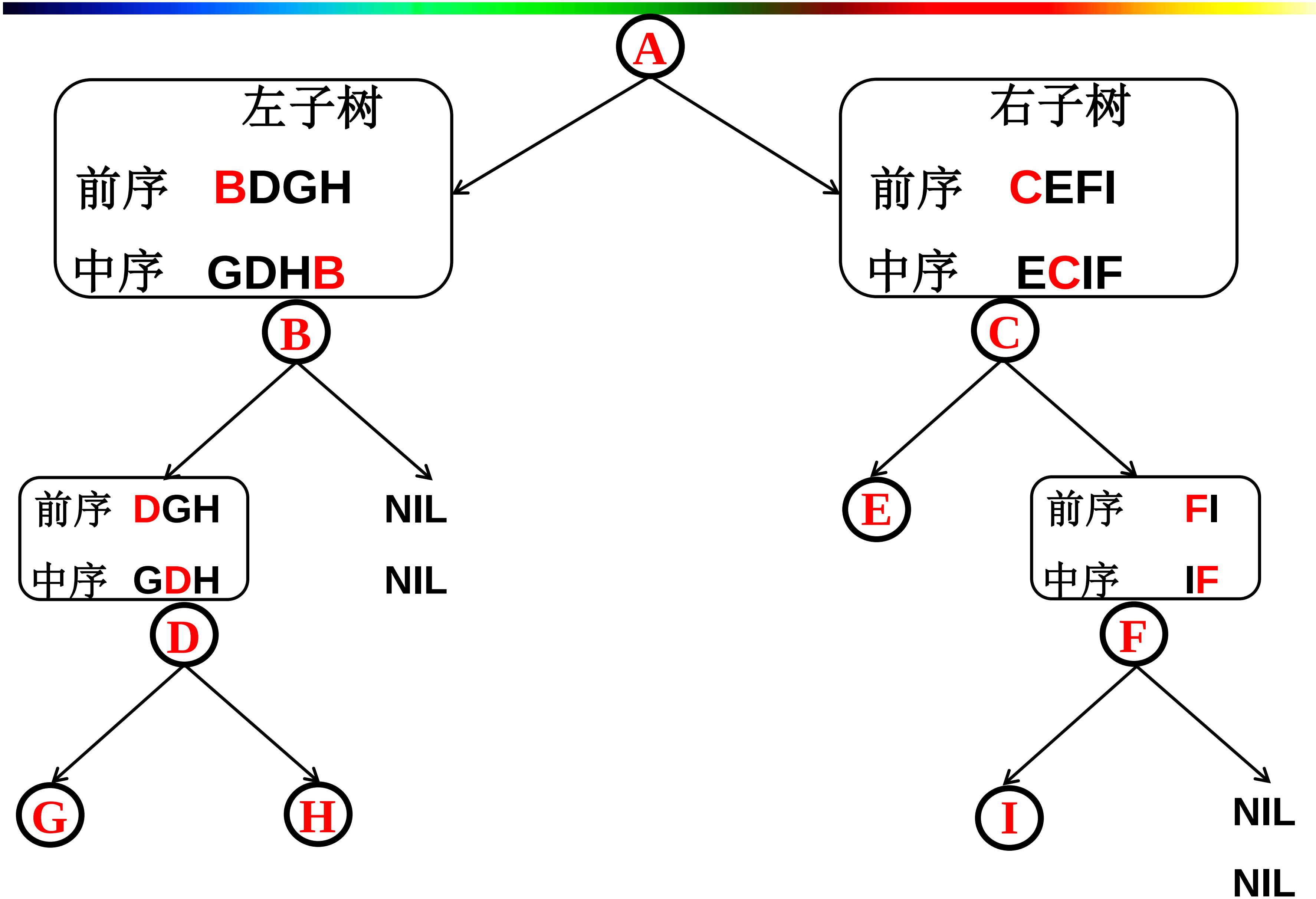
III. 递归求解树：将左、右子树分别看成一棵二叉树，重复上述步骤，直到所有节点完成定位。

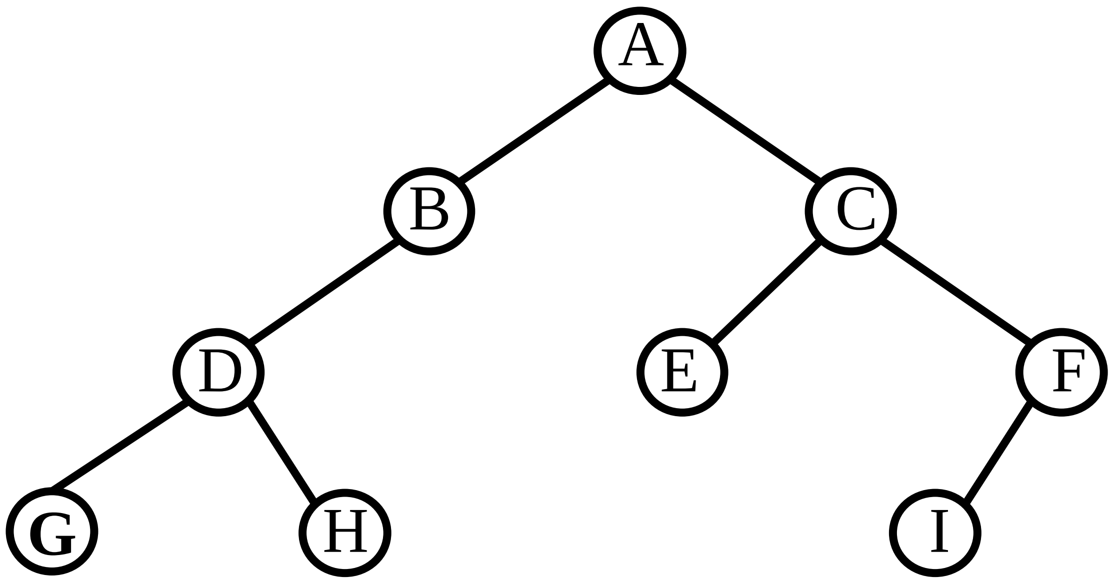
举例

- (1)已知一棵二叉树的前序序列和中序序列分别为ABDGHCEFI和GDHBAECIF，请画出此二叉树。
- (2)已知一棵二叉树的中序序列和后序序列分别为BDCEAFHG和DECBHGFA，请画出此二叉树。
- (3)已知一棵二叉树的前序序列和后序序列分别为AB和BA，请画出这两棵不同的二叉树。

前序 ABDGHCEFI

中序 GDHBAECIF



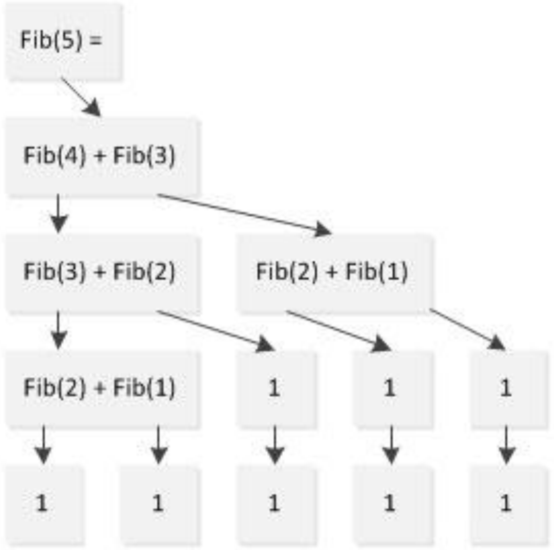

A
B
C
D
E
F
G
H
I

递归 ➡ 非递归

- 递归带来大量函数调用，有许多额外的时间开销
- 理论上所有的递归都是可以转换成非递归
  - 可以用数据归纳法来证明！
- 使用递归注意两点
  - 递归就是在过程或函数里调用自身
  - 必须有明确的递归结束条件，称为递归出口
- 实现算法的非递归转换，需要借助栈来实现
  - 在递归的入口处，用栈保存返回地址！

非递归深度优先遍历

- 递归算法简洁，但也有不足之处，这时就存在如何把递归算法转化成等价的非递归算法的问题
  - F(1)=1，F(2)=1
  - F(n)=F(n-1)+F(n-2)（n>=3，n∈N*）



- 问题的关键是设置栈结构，使其仿照递归算法执行过程中编译栈的工作原理，完成非递归遍历算法

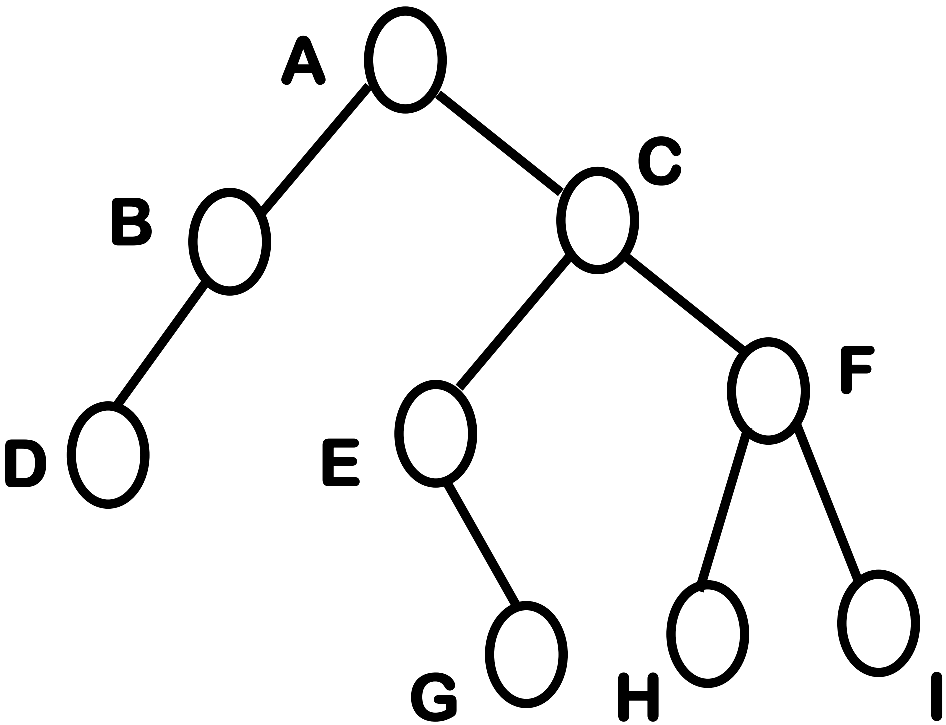
1、前序遍历非递归算法

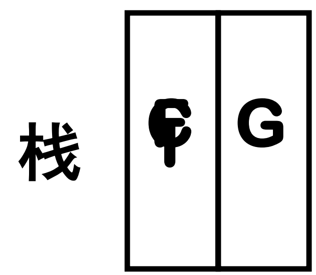
- 看到一个结点，访问他，并把非空右子结点压栈，然后深度遍历其左子树（走之前右孩子先入栈）
- 左子树遍历完毕，弹出结点并访问之，继续遍历（左子树完毕就出栈）
- 开始推入一空指针作为监视哨，作为遍历结束标志（遇到监视哨就结束）

1、前序遍历非递归算法

前序序列 A B D E H

入栈序列 C F G I





访问结点

栈中结点

已访问结点

非递归前序遍历二叉树算法

- 看到一个结点，访问他，并把非空右子结点压栈，然后深度遍历其左子树（走之前右孩子先入栈）。
- 左子树遍历完毕，弹出结点并访问之，继续遍历（左子树完毕就出栈）。
- 开始推入一空指针作为监视哨，作为遍历结束标志（遇到监视哨就结束）。

```
template<class T> void BinaryTree<T>::PreOrderWithoutRecu.
(BinaryTreeNode<T>* root) {
   using std::stack;                         // 使用STL中的stack
   stack<BinaryTreeNode<T>*> aStack;
   BinaryTreeNode<T>* pointer=root;
   aStack.push(NULL);                        // 栈底监视哨
   while (pointer) {                         //或者!aStack.empty()
      Visit(pointer);                        // 访问当前结点
      if (pointer->rightchild() != NULL)     // 右孩子入栈
         aStack.push(pointer->rightchild());
      if (pointer->leftchild() != NULL)
         pointer = pointer->leftchild();     //左路下降
      else      pointer = aStack.pop();   //左子树访问完毕转向右子树
   }
}
```

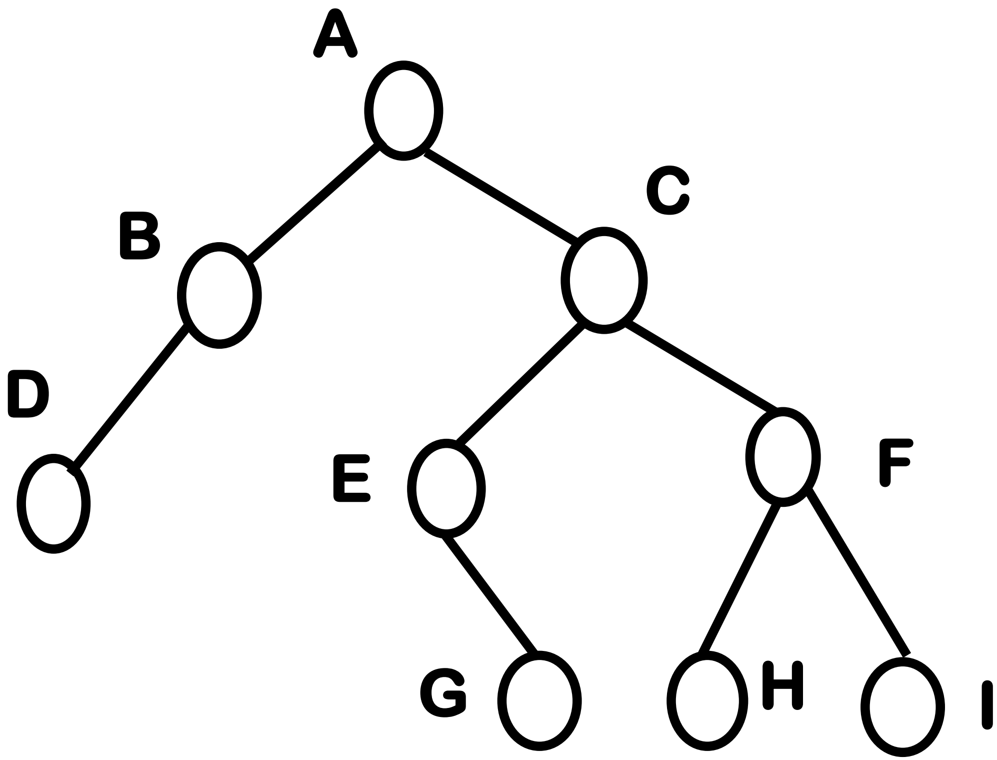
2、中序遍历非递归算法

- 遇到一个结点
  - 入栈
  - 遍历其左子树
- 遍历完左子树
  - 出栈并访问之
  - 遍历右子树

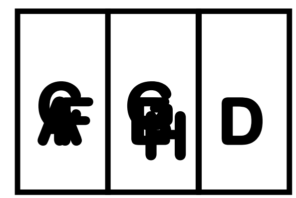
示例

中序序列

进栈序列　A B D C E G F H I



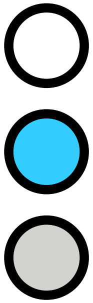
栈



未访问结点

栈中结点

出栈结点



非递归中序遍历

```
template<class T>  void BinaryTree<T>:: InOrderWithoutRecu.
(BinaryTreeNode<T>* root) {
    using std::stack;                        // 使用STL中的stack
    stack<BinaryTreeNode<T>* > aStack;
    BinaryTreeNode<T>* pointer = root;
    while ( !aStack.empty() || pointer ) {   // 注意循环条件判断
        if ( pointer ) {
            aStack.push(pointer);            // 当前结点地址入栈
            pointer = pointer->leftchild();  // 当前链指向左孩子
        } //end if
```

```
        else {
            pointer = aStack.pop();                        //栈顶元素退栈
            Visit(pointer);                                //访问当前结点
            pointer=pointer->rightchild();   //当前链接结构指向右孩子
        } //end else
    } //end while
}
```

非递归后序遍历

基本思想

- 遇到一个结点，将其入栈，遍历其左子树
- 左子树遍历结束后，不能马上访问栈顶结点，而是要按照其右链去遍历其右子树
- 右子树遍历后才能从栈顶托出该结点访问

非递归后序遍历(续)

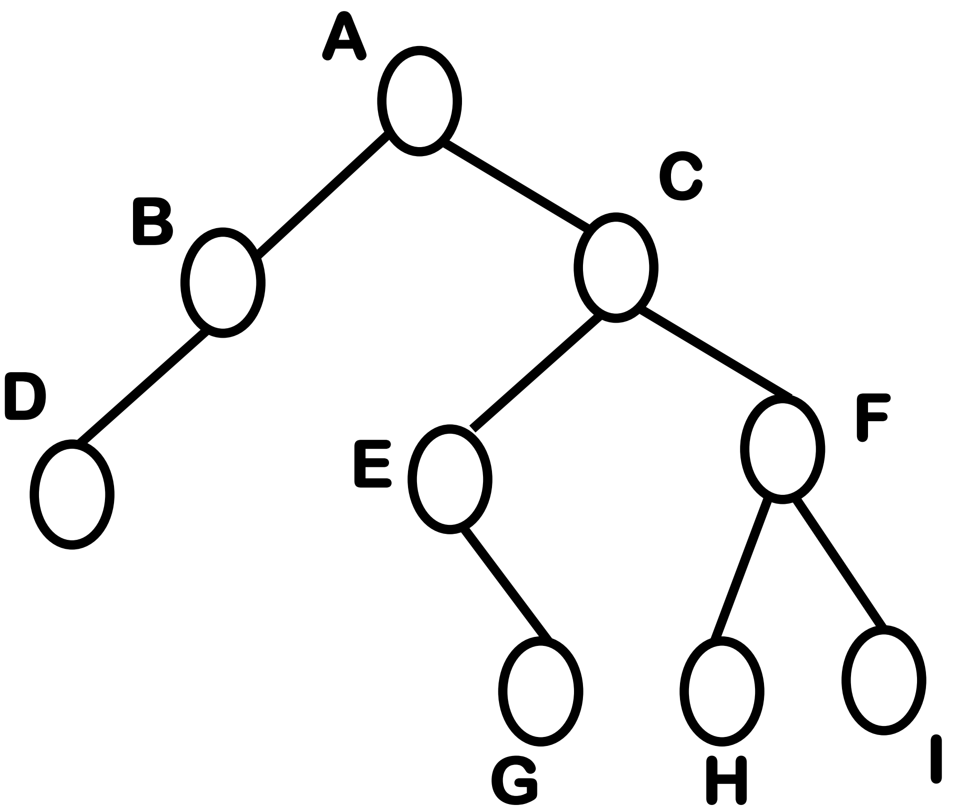
- 需给栈中每个元素附加一个特征位，以便当从栈顶托出一个结点时区分是从左边回来(**则要继续遍历右子树**)，还是从右边回来(**该结点的左、右子树均已遍历**)

- 特征位表示
  - **Left**表示进入的是该结点的左子树，从左边回来
  - **Right**表示进入的是该结点的右子树，从右边回来

非递归后序遍历二叉树

后序序列　D

出栈序列



栈

(D,B)
(B,L)
(A,L)

未访问结点

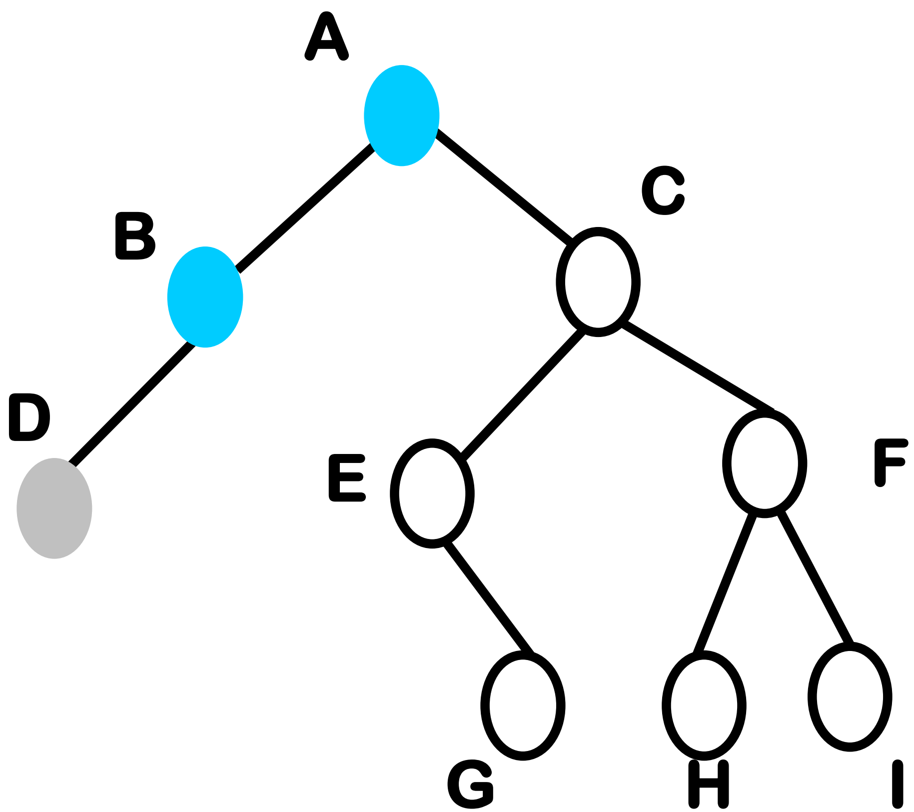
栈中结点

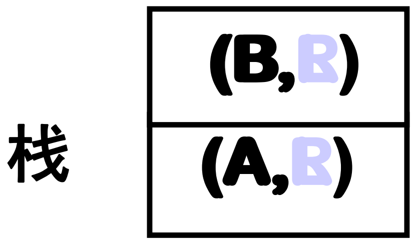
出栈结点

示例

后序序列 D B

出栈序列 (D,L) (D,R)





未访问结点

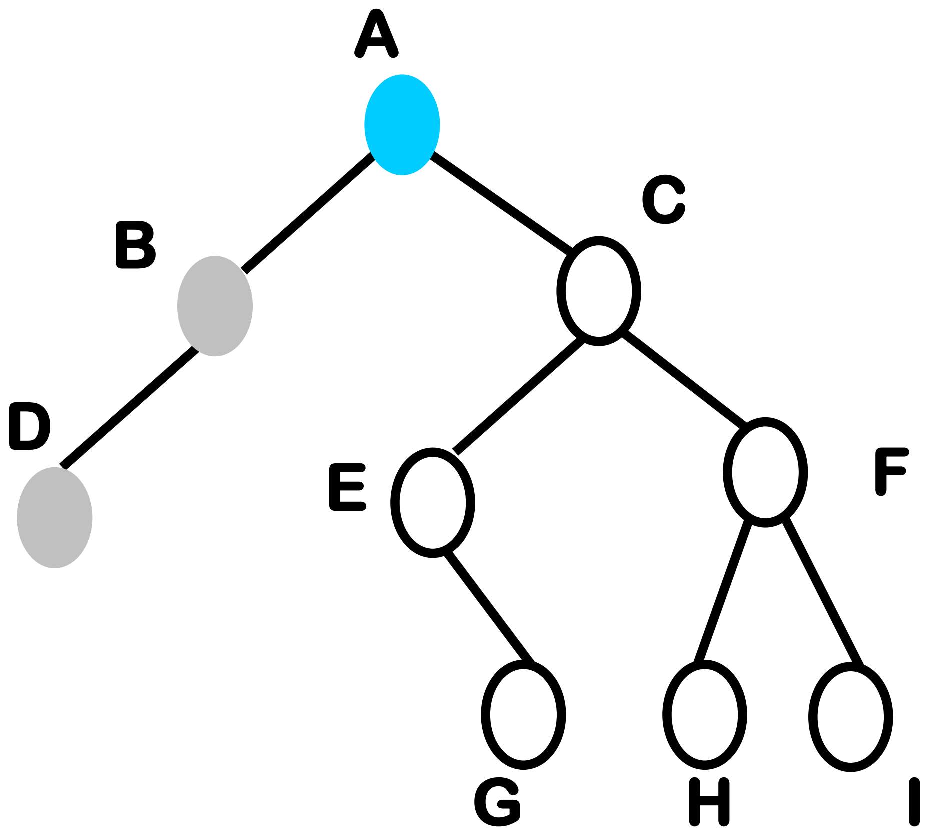
栈中结点

出栈结点

示例

后序序列 D B G

出栈序列 (D,L) (D,R) (B,L) (B,R) (A,L)



栈

(G,R)
(E,R)
(C,L)
(A,R)

未访问结点

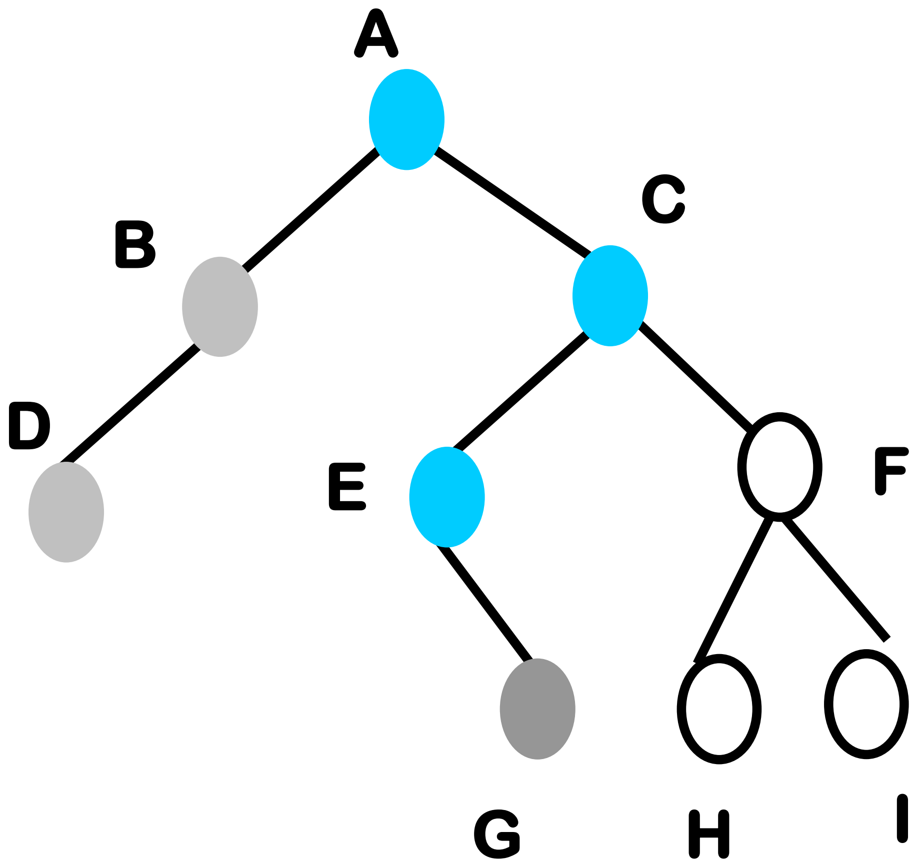
栈中结点

出栈结点

示例

后序序列 D B G E H

出栈序列 (D,L) (D,R) (B,L) (B,R) (A,L) (E,L) (G,L) (G,R)



栈

(H,R)
(E,R)
(C,R)
(A,R)

未访问结点

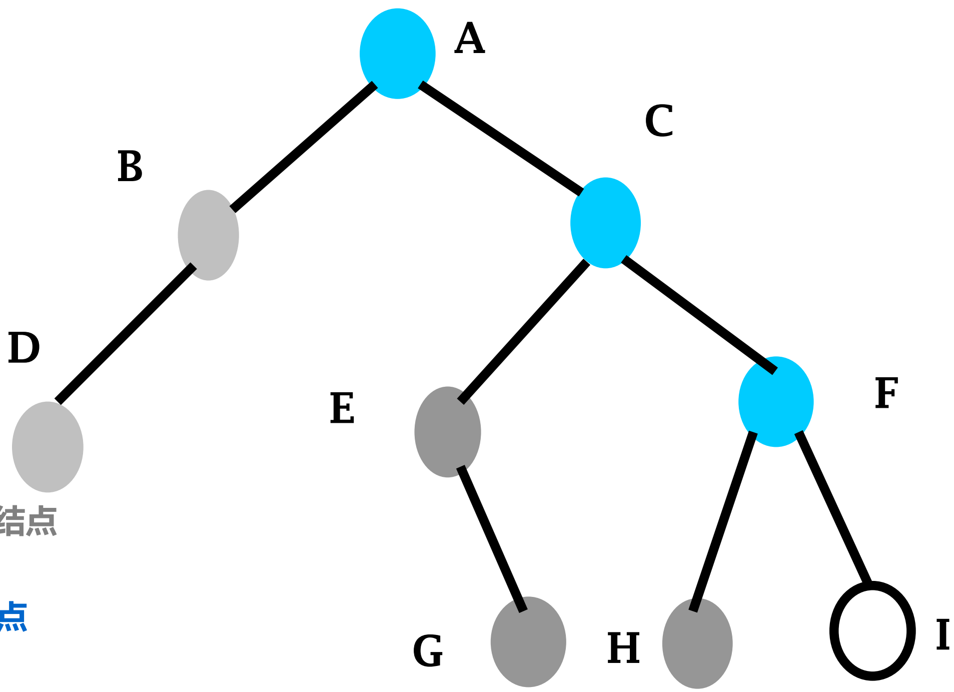
栈中结点

出栈结点

示例

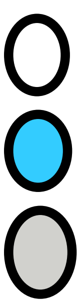
后序序列 D B G E H I F C A

出栈序列(D,L) (D,R) (B,L) (B,R) (A,L) (E,L) (G,L) (G,R) (E,R) (C,L) (H,L) (H,R)



栈

(I,R)
(F,R)
(C,R)
(A,R)



非递归后序遍历二叉树算法

```
enum Tags { Left, Right };                       // 定义枚举类型标志位
template <class T>
class StackElement  {                            // 栈元素的定义
public:
    BinaryTreeNode<T>* pointer;                  // 指向二叉树结点的指针
    Tags tag;                                    // 标志位
};
template<class T>
void BinaryTree<T>::PostOrderWithoutRecursion(BinaryTreeNode<T>* root) {
    using std::stack;                            // 使用STL的栈
    StackElement<T> element;
    stack<StackElement<T > > aStack;
    BinaryTreeNode<T>* pointer;
    pointer = root;
```

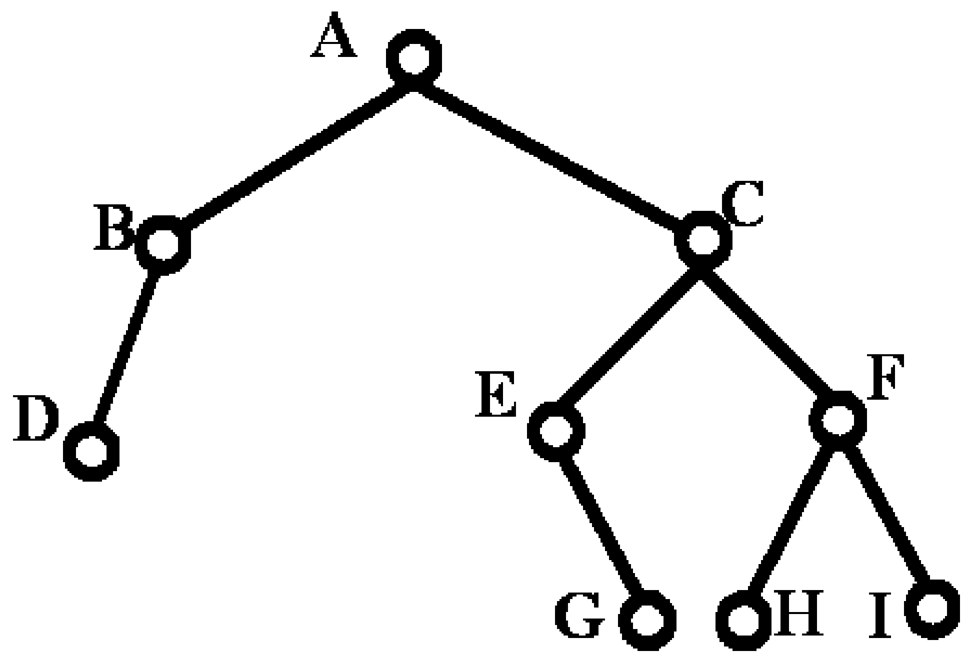
```
    while (!aStack.empty() || pointer) {
      if (pointer != NULL) {                              // 沿非空指针压栈，并左路下降
          element.pointer = pointer;  element.tag = Left;
          aStack.push(element);                           // 把标志位为Left的结点压入栈
          pointer = pointer->leftchild();
      }
      else{  element = aStack.pop();                      // 获得栈顶元素，并退栈
              pointer = element.pointer;
              if (element.tag == Left) {                  // 如果从左子树回来
                  element.tag = Right; aStack.push(element); //置标志位为Right
                  pointer = pointer->rightchild();
              }
              else {  Visit(pointer);                     // 如从右子树回来访问当前结点
                  pointer = NULL;                         // 置point为空，以继续弹栈
              }
        }
    }//end while
}
```

复杂性分析

- 在各种遍历中，每个结点都被访问且只被访问一次，时间代价为O(n)
- 非递归保存入出栈时间
  - 前序、中序，某些结点入/出栈一次，不超过 O(n)
  - 后序，每个结点分别从左、右边各入/出一次，O(n)
- 深搜：栈的深度与树的高度有关
  - 最好 O(log n)
  - 最坏 O(n)

5.2.3 广度优先遍历二叉树

- 从二叉树的根结点开始，自上而下逐层遍历；
- 同层节点，按从左到右的顺序对结点逐一访问
- 例如：**ABCDEFGHI**

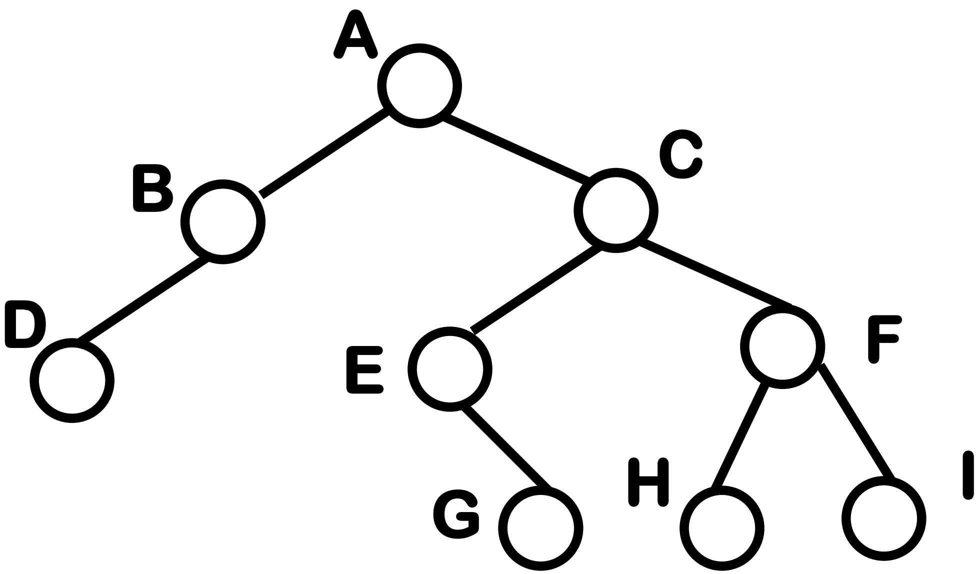


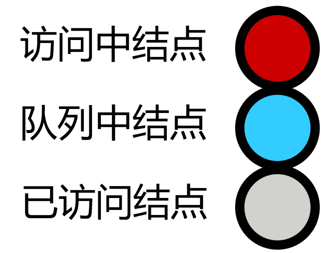
示例

BFS序列

队列

A	B	C	D	E	F	G	H	I





算法实现

```
void BinaryTree<T>::LevelOrder(BinaryTreeNode<T>* root){
    using std::queue;                                   // 使用STL的队列
    queue<BinaryTreeNode<T>*> aQueue;
    BinaryTreeNode<T>* pointer = root;                  // 保存输入参数
    if (pointer) aQueue.push(pointer);                  // 根结点入队列
    while (!aQueue.empty())  {                          // 队列非空
        pointer = aQueue.pop();                         // 当前结点出队列
        Visit(pointer->value());                        // 访问当前结点
        if(pointer->leftchild())
            aQueue.push(pointer->leftchild());          // 左子树进队列
        if(pointer->rightchild())
            aQueue.push(pointer->rightchild());         // 右子树进队列
    }
}
```

//算法的空间复杂性如何?

5.3 二叉树的存储结构

- 动态存储结构
  - 链式存储
- 静态存储结构
  - 顺序存储（完全二叉树）

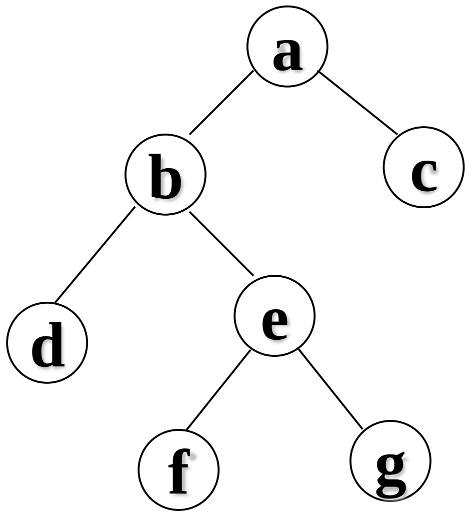
1、动态链式存储结构

- 各结点随机存储在内存空间，结点之间关系用指针表示
- 除存储结点本身数据外，每个结点再设置两个指针字段left和right，分别指向左孩子和右孩子
- 子女为空时指针为空指针
- 这种存储结构称为二叉链表表示法
- 结点的形式为

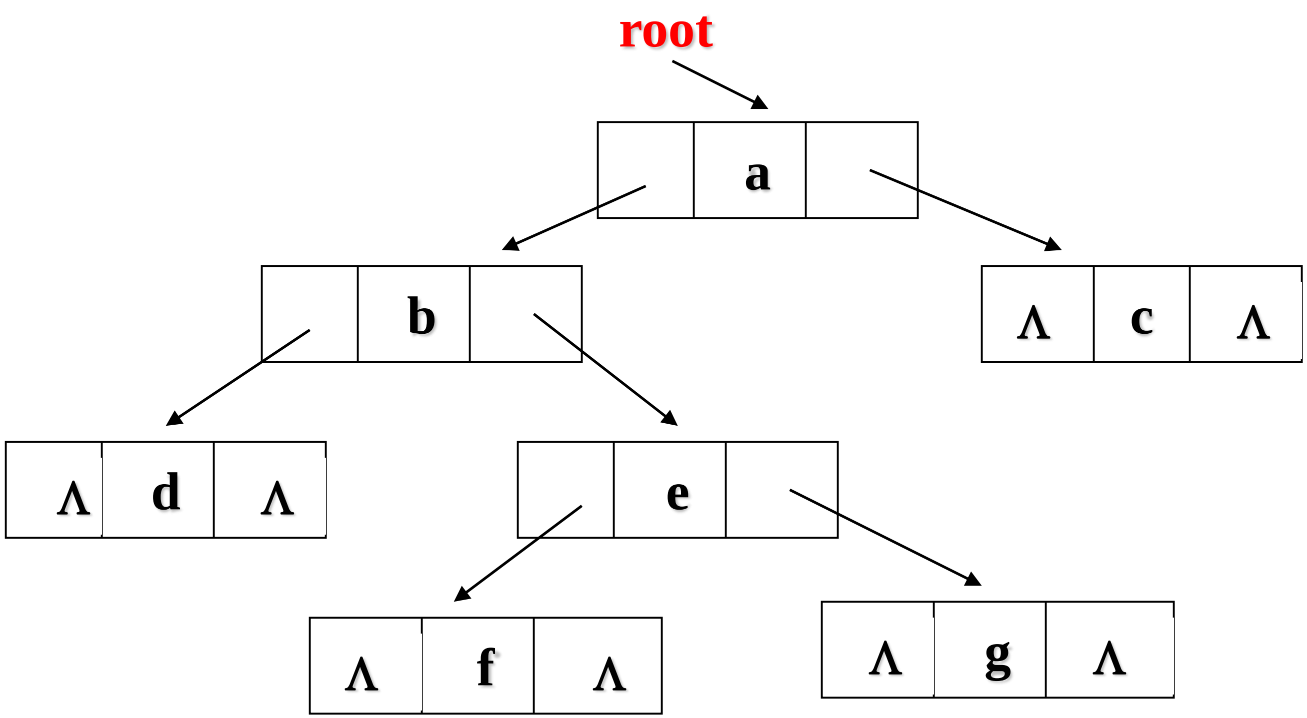
left	info	right

有两个指针域的结点结构

二叉树与二叉链表



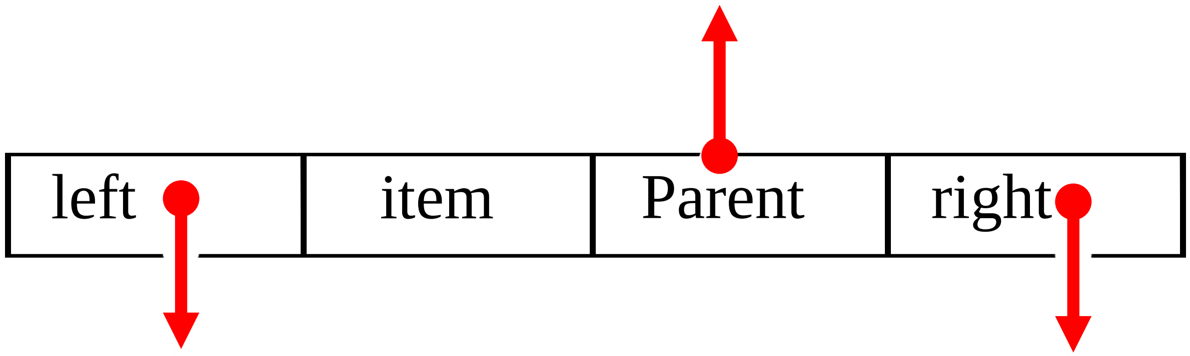
(a)二叉树



(b) 二叉树的链式存储

三叉链表

- 三叉链表
  - 除left和right指针外，每个结点再增加一个指向父节点的指针parent，形成“三叉链表”
  - 提供了“向上”访问的能力



二叉树部分成员函数的实现

```
private:                                //在BinaryTreeNode类中增加两个私有成员
    BinaryTreeNode<T> *left;        // 指向左子树的指针
    BinaryTreeNode<T> *right;       // 指向右子树的指针

template<class T>    // 判二叉树是否为空
bool BinaryTree<T>:: isEmpty() const  {
        return ( root != NULL ? false : true);
}
```

二叉树部分成员函数的实现(续)

```
template<class T>                                   //删除二叉树
void BinaryTree<T>:: DeleteBinaryTree (BinaryTreeNode<T>
    *root)  {
    if (root != NULL) {
        DeleteBinaryTree(root->left);          //递归删除左子树
        DeleteBinaryTree(root->right);         //递归删除右子树
        delete root;                           // 删除根结点
    }
}
```

寻找节点的父节点

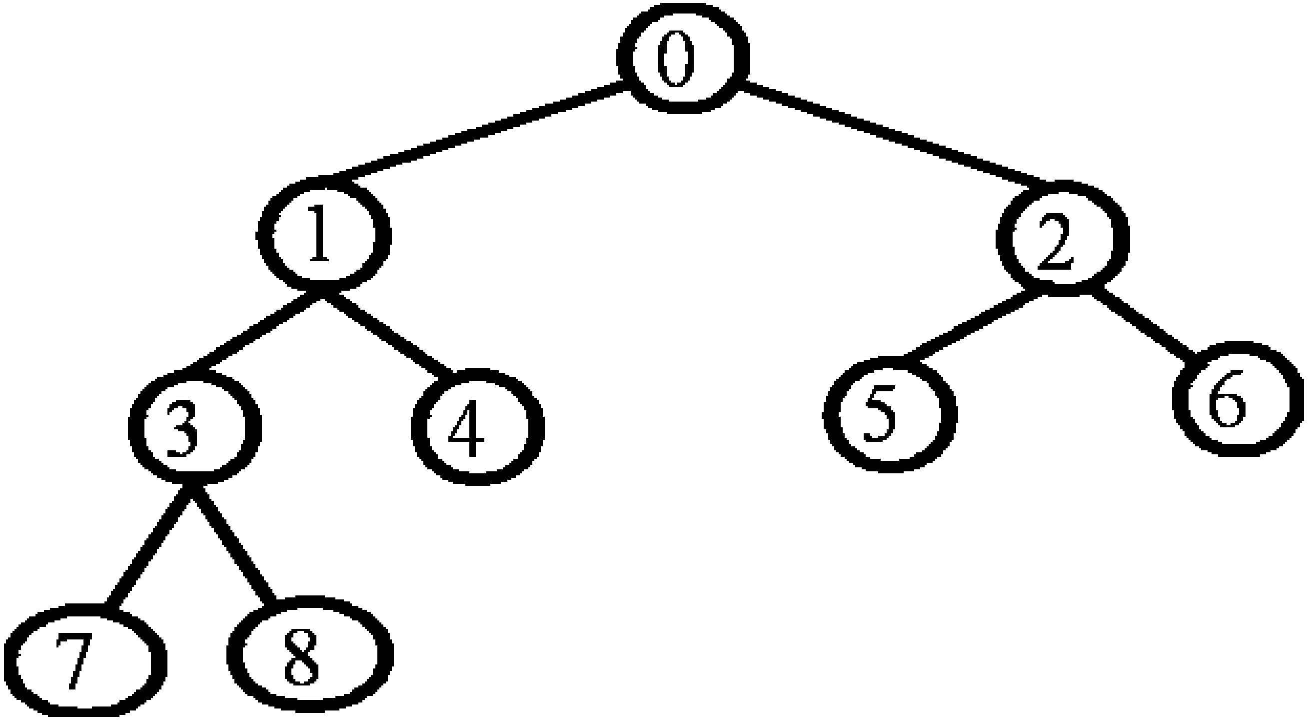
```
template<class T> BinaryTreeNode<T>* BinaryTree<T>::
Parent(BinaryTreeNode<T> *rt, BinaryTreeNode<T> *current)  {
    BinaryTreeNode<T> *tmp,
    if (rt == NULL) return NULL;
    if (rt ->leftchild() == current || rt->rightchild() == current)
        return rt;                         //如果孩子是current则返回parent
    if ((tmp =Parent(rt- >leftchild(), current) != NULL)
        return tmp;
    if ((tmp =Parent(rt- > rightchild(), current) != NULL)
        return tmp;
    return NULL;                           //没找到，携带空指针返回
}
```

2、静态数组存储(完全二叉树）

- 按照一定次序，用一组地址连续的存储单元存储二叉树上的各个结点元素
- 二叉树是非线性结构，因此必须将二叉树的结点排成一个线性序列，使得通过结点在序列中的相对位置确定结点间的逻辑关系

完全二叉树的顺序存储

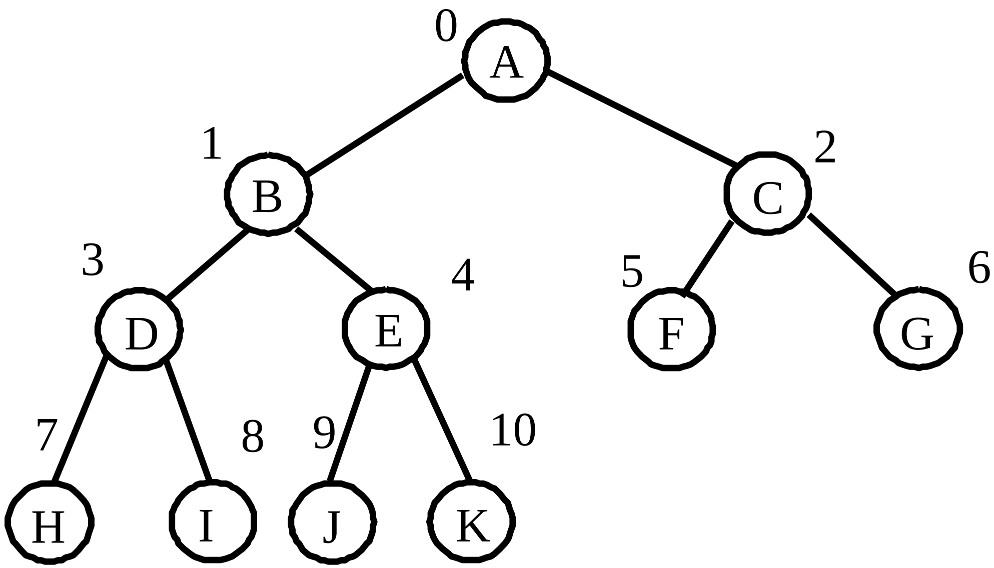
- 一棵具有n个结点的完全二叉树，可以从根结点起自上而下，从左至右地把所有的结点编号，得到一个足以反映整个二叉树结构的线性序列。



完全二叉树的下标公式

- 完全二叉树中除最下面一层外，各层都被结点充满，每一层结点个数恰是上一层结点个数的两倍。因此，从一个结点的编号就可以推知它的父母，左、右子女，兄弟等结点的编号
  - 当$2i+1<n$时，结点i的左子女是结点$2i+1$，否则结点i没有左子女
  - 当$2i+2<n$时，结点i的右子女是结点$2i+2$，否则结点i没有右子女
  - 当$0<i<n$时，结点i的父母是结点$\lfloor(i-1)/2\rfloor$
  - 当i为偶数且$0<i<n$时，结点i的左兄弟为 $i-1$，否则结点i没有左兄弟
  - 当i为奇数且$i+1<n$时，结点i的右兄弟为 $i+1$，否则结点i没有右兄弟

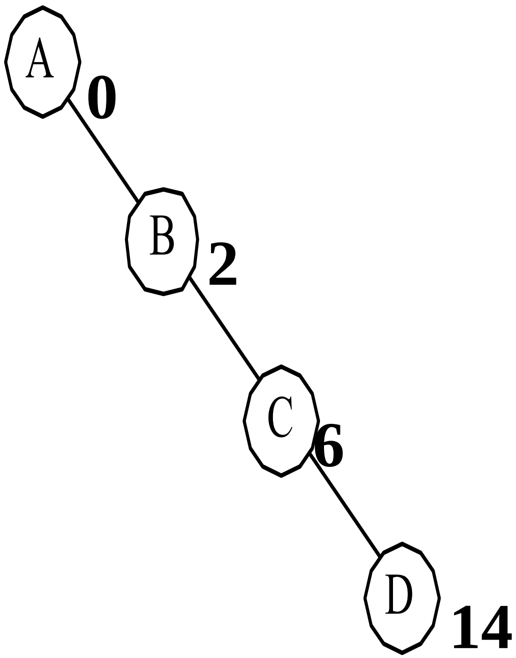
示例



完全二叉树

A	B	C	D	E	F	G	H	I	J	K				

非完全二叉树顺序存储

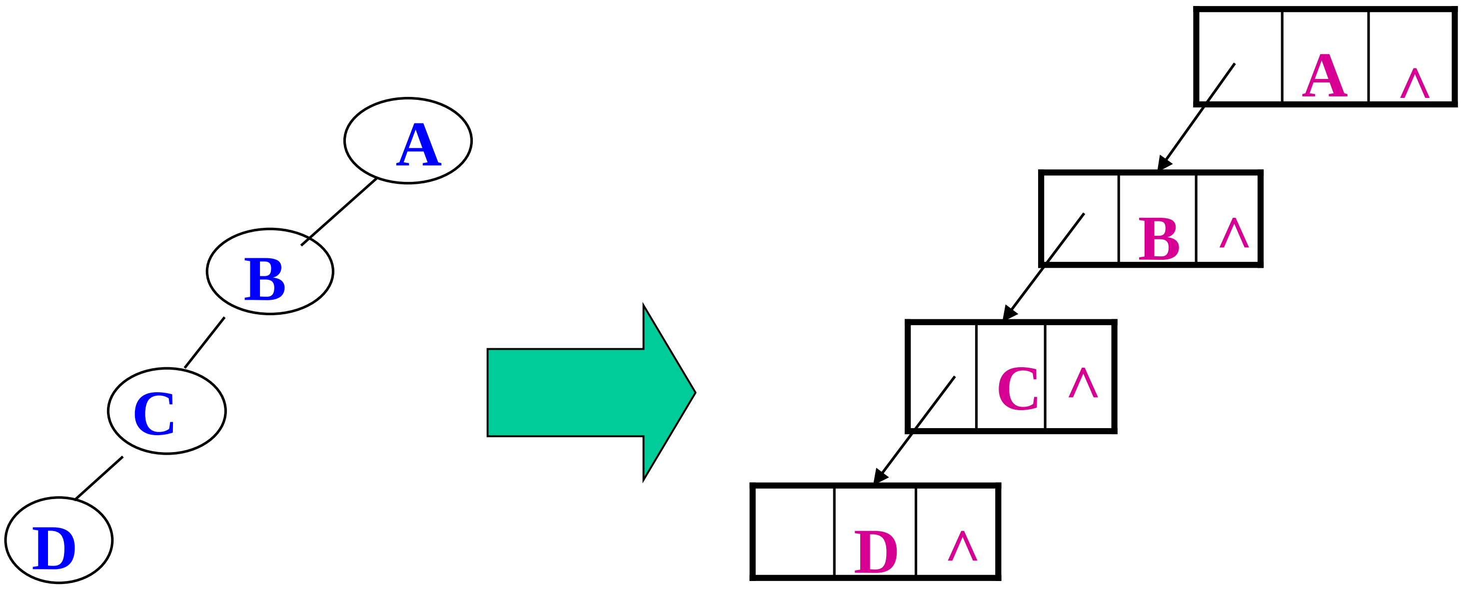


(a) 单支二叉树

A	∧	B	∧	∧	∧	C	∧	∧	∧	∧	∧	∧	∧	D

(b) 顺 序 存 储 结 构

非完全二叉树链式存储



优点：①不浪费空间；②插入、删除方便

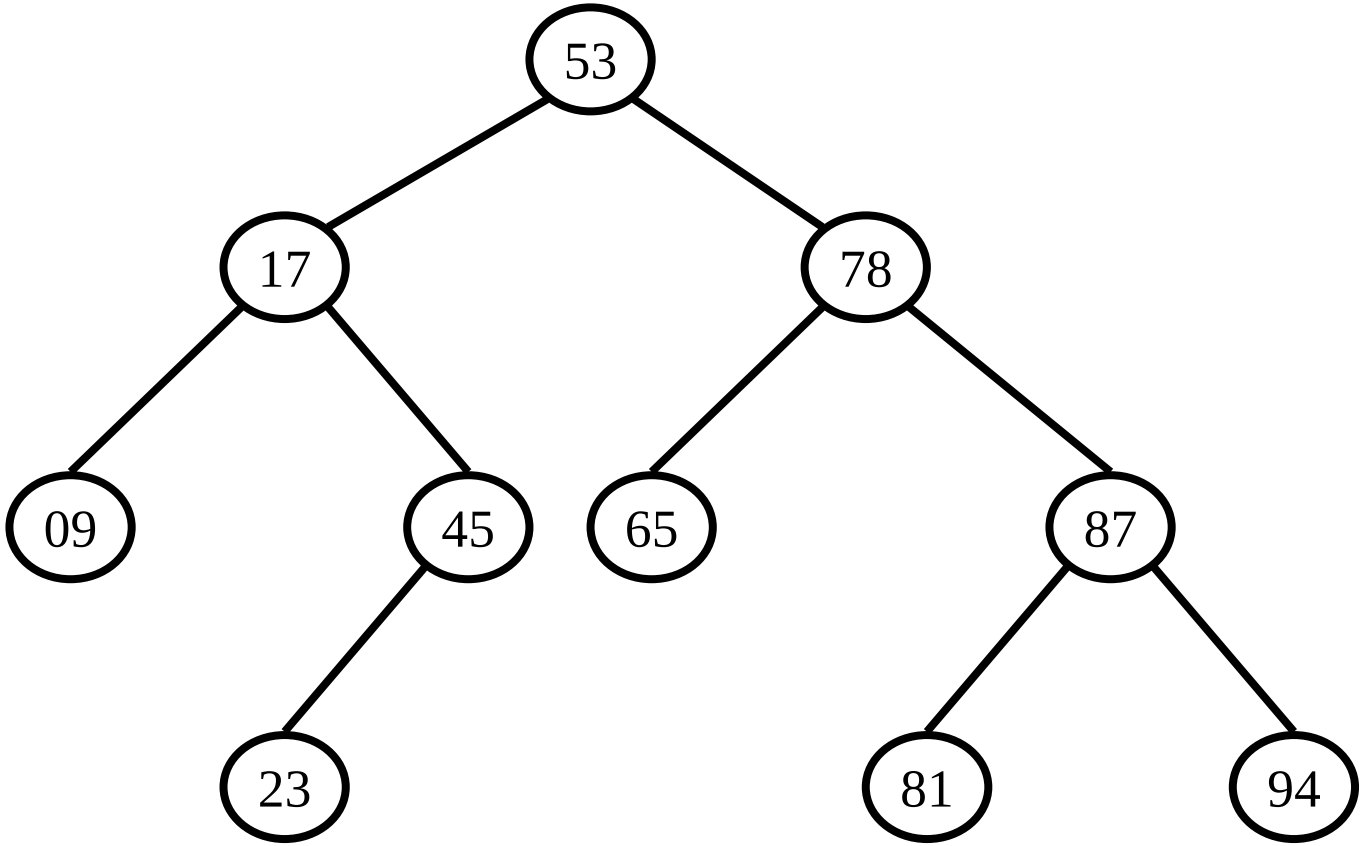
顺序存储总结

- 完全二叉树结点的层次序列足以反映二叉树的结构
  - 所有结点按层次顺序依次存储在一片连续的存储单元中，则根据一个结点的存储地址就可算出它的左右子女，父母的存储地址
  - 数组下标标识了指针的指向关系
  - 存储完全二叉树的最简单，最节省空间的存储方式
- 完全二叉树的顺序存储，在存储结构上是线性的，但在逻辑结构上它仍然是二叉树型结构

5.4 二叉搜索树

- 二叉搜索树（Binary Search Tree, BST），也称二叉排序树
  - 或者是空树
  - 或者具有下列性质
    - 对于任何值为K的结点，该结点的左子树中的结点的值都小于K；
    - 该结点右子树的结点值都大于K；
    - 左右子树也分别为二叉搜索树
- BST树的性质
  - **树中结点的值唯一**
  - 按照中序周游将各结点打印出来，将得到由小到大的排列

BST图示



- 中序遍历结果：<u>**09，17，23，45，53，65，78，81，87，94。**</u>

  显然中序遍历是有序的，故又称二叉排序树

二叉搜索树的搜索过程

- 从根结点开始，在二叉搜索树中检索值K
  - 如果根结点值为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
  - 一直持续到K被找到或者到达树叶（搜索失败）
- **二叉搜索树的效率就在于只需检索二个子树之一**

二叉搜索树的插入

- **新结点插入后仍是二叉搜索树，值不重复!**
- 插入过程
  - 将待插入结点的码值与树根的码值比较
    - 若插入结点值小于根结点值，则进入左子树，否则进入右子树；
    - 若相等则直接返回
  - 递归进行下去，直到遇到空指针，把新结点插入到该位置
  - 成功的插入，首先要执行一次**失败的查找**，再执行插入！

插入算法

```
template<class T>
void BinarySearchTree<T>::InsertNode( BinaryTreeNode<T>* root , *
    newpointer){        //向二叉搜索树插入新结点newpointer
    BinaryTreeNode<T>* pointer;
    if (root==NULL) {  //用指针newpointer初始化二叉搜索树树根，赋值实现
        Initialize(newpointer);   return;
    } else pointer=root;
    while(1){
        if (newpointer->value()==pointer->value())  return; //=则不处理
        else if (newpointer->value() < pointer->value()) { //左子树
            if (pointer->leftchild()==NULL){              //作为左孩子
                pointer->left=newpointer;
                return;
            } else   pointer=pointer->leftchild();        //否则向左遍历
    }
```

插入算法

```
        else {    //作为右子树
            if  (pointer->rightchild()==NULL) { //右孩子空，则作为右孩子
                pointer->right=newpointer;    return;
            } else    pointer=pointer->rightchild();
        }
    }
}
```

性能分析

- BST树的检索，每次只需与结点的一棵子树比较
- 插入操作时，不像线性表插入元素移动大量数据，只需改动某个结点的空指针插入一个叶结点即可
- 插入一个新结点操作的时间复杂度是根到插入位置的路径长度，因此在树形比较平衡时二叉搜索树的效率相当高

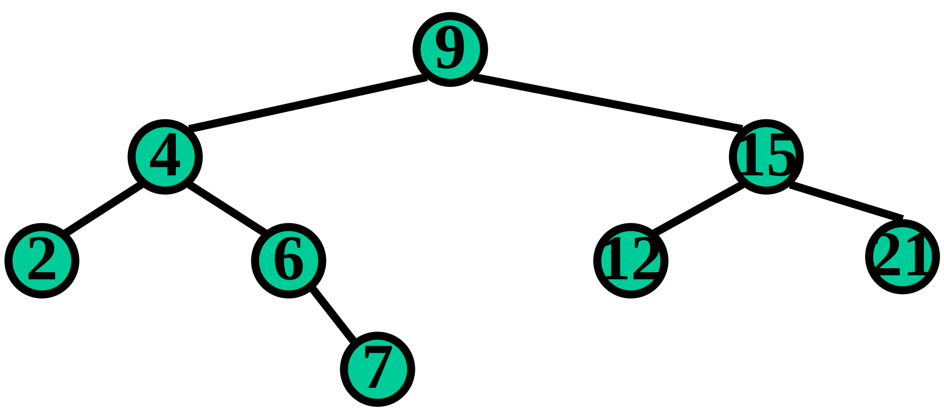
二叉搜索树的建立

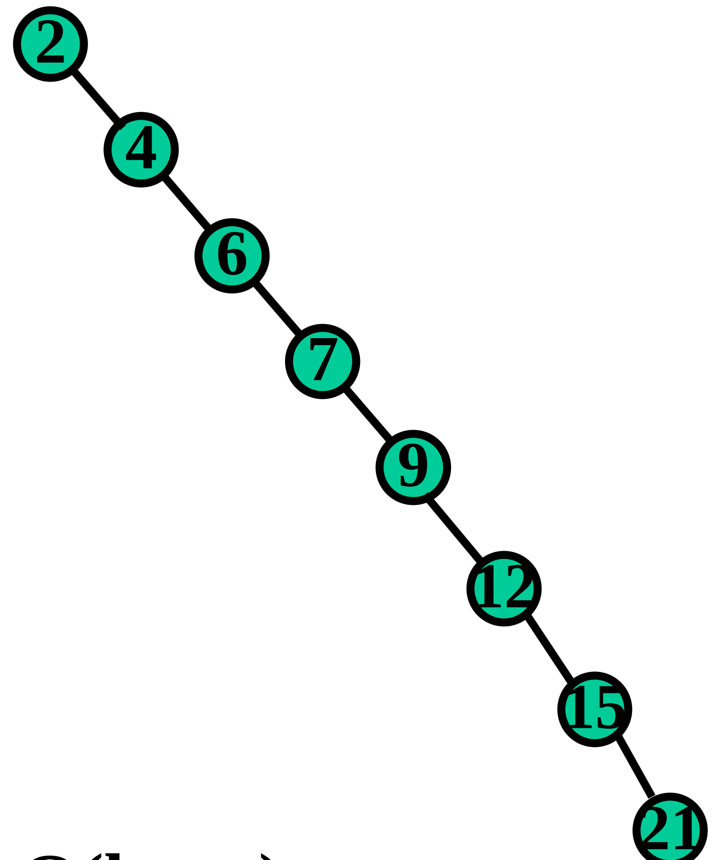
- 对于给定的关键码集合，为建立二叉搜索树，可以从一个空的二叉搜索树开始，将关键码一个个插进去

- 将关键码集合组织成二叉搜索树，实际上起了对集合里的关键码进行排序的作用，按中序周游二叉搜索树，就能得到排好的关键码序列

BST树的平衡问题

- 输入：9,4,2,6,7,15,12,21
- 输入：2,4, 6,7, 9, 12,15, 21



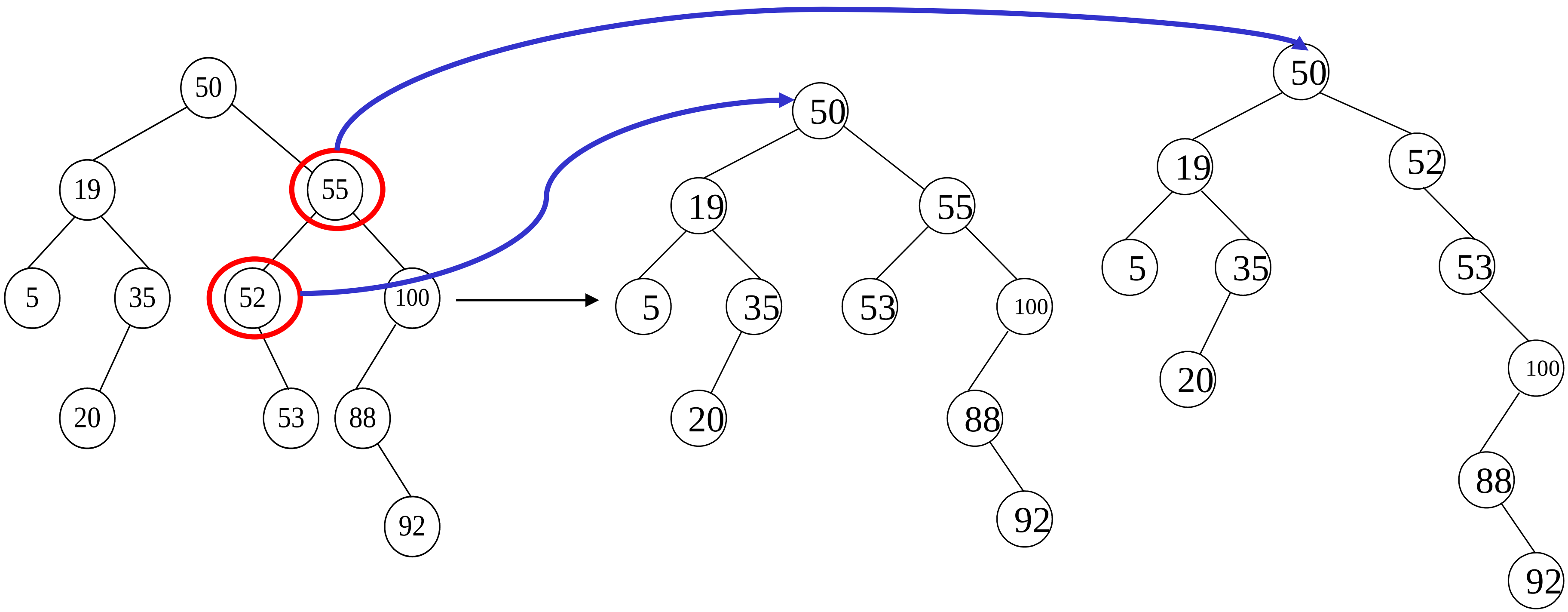


- 希望保持理想状况
- 插入、删除、查找时间代价为O(log*n*)

二叉搜索树的删除

- 首先找到待删除的结点pointer，删除该结点的过程如下（temppointer是指针变量）：
  - 若结点pointer没有左子树：则用pointer右子树的根代替被删除的结点pointer;
  - 若结点pointer有左子树：则在左子树里找到按中序周游的最后一个结点temppointer（右子树必为空），把temppointer的右指针置成pointer右子树的根，然后用结点pointer左子树的根代替被删除的结点pointer。

二叉搜索树的删除示例



二叉搜索树

没有左子树的情况（节点52）

有左子树的情况（节点55）

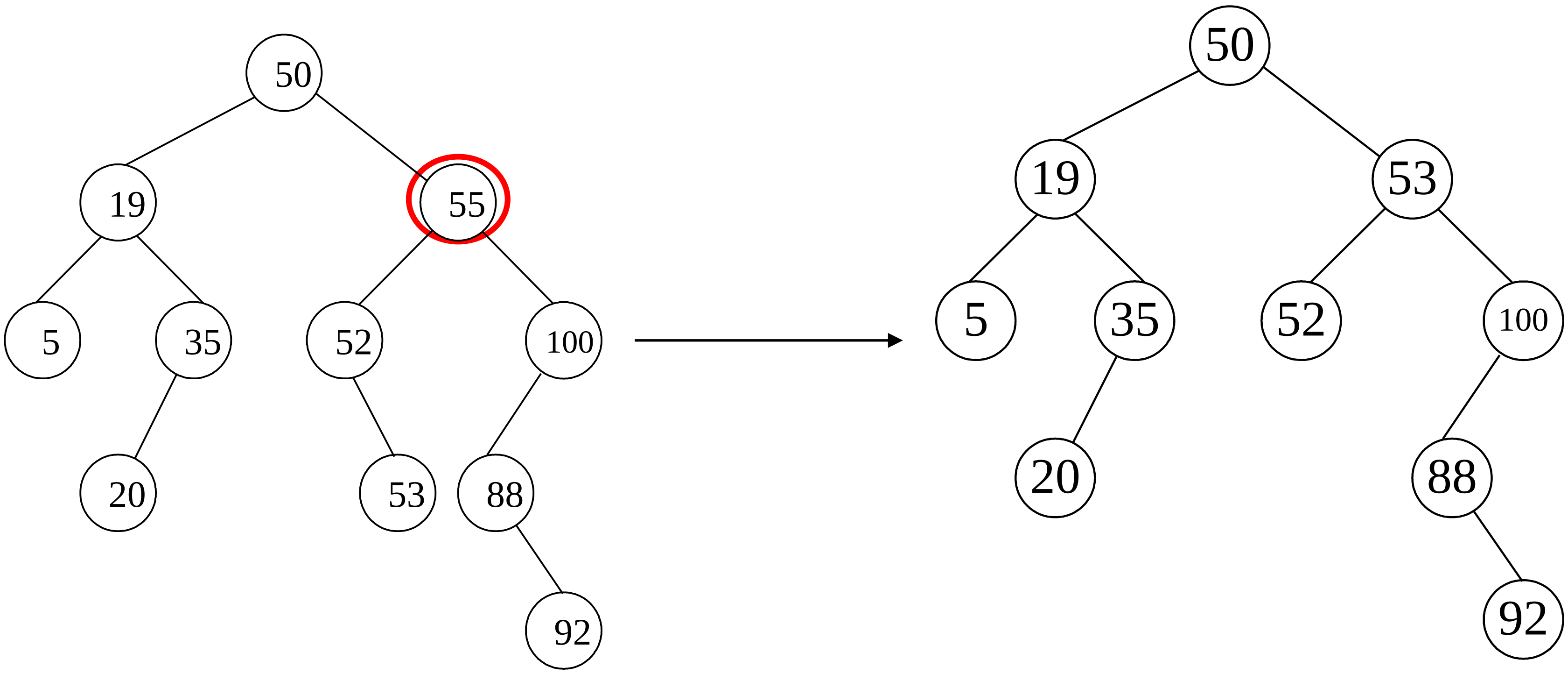
有什么问题？？

高度失衡→效率降低

改进

- 改进的二叉搜索树结点删除算法的思想为：
  - 若结点pointer没有左子树：则用pointer右子树的根代替被删除的结点pointer
  - 若结点pointer有左子树：则在左子树里找到按中序周游的最后一个结点replpointer（即左子树中的最大结点）并将其从二叉搜索树里删除
  - 由于replpointer没有右子树，删除该结点只需用replpointer的左子树代替replpointer，然后用replpointer结点代替待删除的结点pointer

示例



二叉搜索树的删除算法

```
template <class T>
void BinarySearchTree<T>::DeleteNodeEx(BinaryTreeNode<T>
    *delpointer) {
    BinaryTreeNode<T>  *replpointer;                //替换结点
    BinaryTreeNode<T>  *replparent = NULL;          //替换结点的父结点
    BinaryTreeNode<T>  *delparent = Parent(delpointer)//待删结点父结点
    //若待删除结点的左子树为空，就将其右子树代替它
    if ( delpointer->leftchild() == NULL )
        replpointer = delpointer->rightchild();
```

// 待删除结点左子树不空，在左子树中寻找最大结点替换待删除结点

```
else {   replpointer = delpointer->leftchild();
         while (replpointer->rightchild() != NULL ) {
               replparent = replpointer;
               replpointer = replpointer->rightchild();
         }
          //替换结点就是被删结点的左子结点，左子树挂接为其父（被删）的左子树
         if (replparent == NULL)
               delpointer->left = replpointer->leftchild();
         // 替换结点的左子树挂接为其父的右子树
         else replparent->right = replpointer->leftchild();
         replpointer->left = delpointer->leftchild();  //继承待删结点左子树
         replpointer->right=delpointer->rightchild();//继承待删结点右子树
}
```

```
// 用替换结点去替代真正的删除结点
if (delparent == NULL)
        root = replpointer;
else if ( delparent->leftchild() == delpointer )
    delparent->left = replpointer;
else delparent->right = replpointer;
delete delpointer;                                // 删除该结点
delpointer = NULL;
return;
}
```

5.5 堆与优先队列

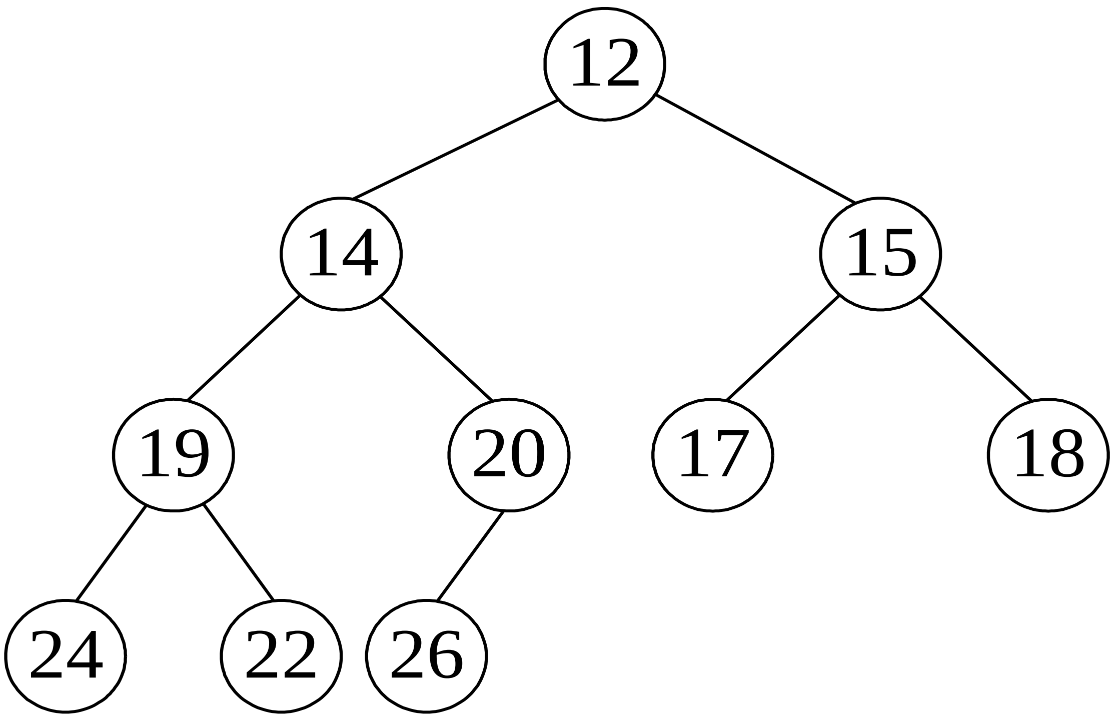
- **5.5.1** 堆的定义及其实现
- **5.5.2** 优先队列

堆的定义

- **最小值堆**：是一个关键码序列$\{K_0, K_1 \ldots K_{n-1}\}$，由完全二叉树表示，有如下**特性**：
  - $K_i \le K_{2i+1}$　(i=0，1，…，<u>n/2-1</u>)
  - $K_i \le K_{2i+2}$
- **最大值堆**
  - 类似可以定义

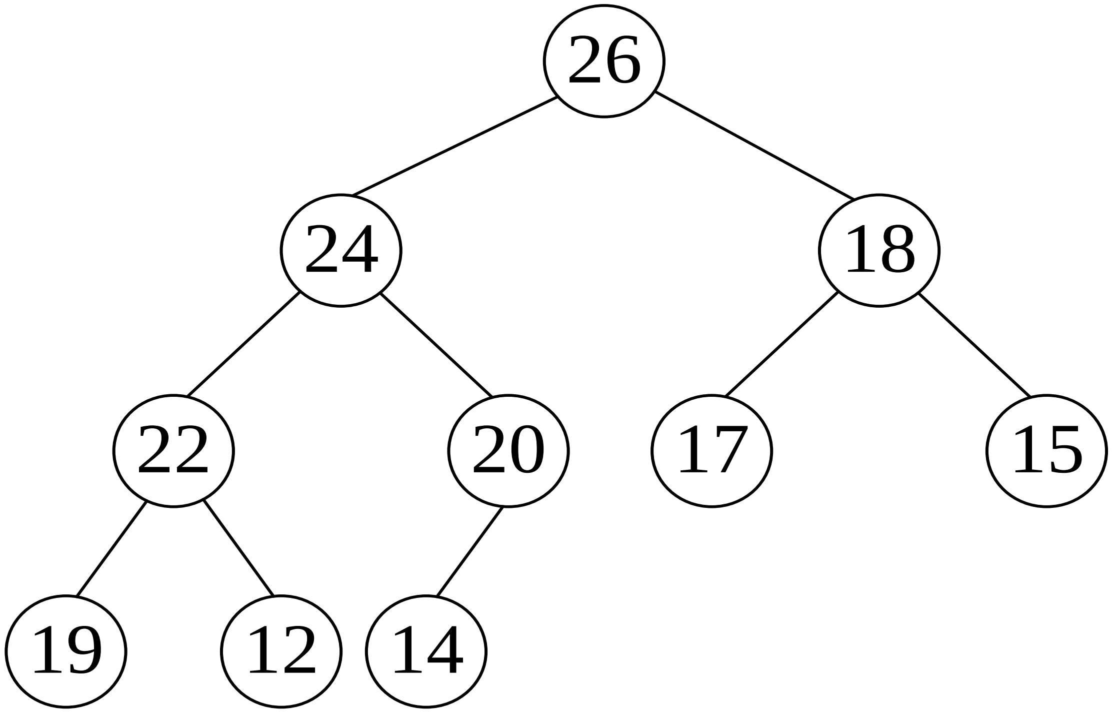
最小值堆示例

- 关键码序列K = {12，14，15，19，20，17，18，24，22，26}所对应的最小堆形成的完全二叉树形式为下图所示：



最大值堆示例

- 关键码序列K = {12，14，15，19，20，17，18，24，22，26}所对应的最大值堆形成的完全二叉树形式为下图所示：



堆的性质

- 堆中数据**局部有序**（与BST树不同，全局有序）
  - 结点与其子女值之间存在大小比较关系
  - 两种堆（最大、最小）
  - 兄弟之间没有限定大小关系
- 堆不唯一
  - 从逻辑角度看，堆实际上是一种树型结构
- **堆是一个可用数组表示的完全二叉树**

堆的类定义

```
template <class T>
class MinHeap  {                    // 最小堆ADT定义
private:
    T* heapArray;                   // 存放堆数据的数组
    int CurrentSize;                // 当前堆中元素数目
    int MaxSize;                    // 堆所能容纳的最大元素数目
    void BuildHeap();               // 建堆
public:
    MinHeap(const int n);           // 构造函数,n为最大元素数目
    virtual ~MinHeap(){delete []heapArray;};      // 析构函数
    bool isLeaf(int pos) const;        // 如果是叶结点，返回TRUE
```

堆的类定义

```
int leftchild(int pos) const;          // 返回左孩子位置
int rightchild(int pos) const;         // 返回右孩子位置
int parent(int pos) const;             // 返回父结点位置
bool Remove(int pos, T& node);         // 删除给定下标元素
bool Insert(const T& newNode);         //向堆中插入新元素
T& RemoveMin();                        // 从堆顶删除最小值
void SiftUp(int position);             // 从position向上调整堆
void SiftDown(int left);               // 筛选法
}
```

堆成员函数

```
template<T>                                    //构造函数
MinHeap<T>::MinHeap(const int n) {
    if (n<=0)
        return;
    CurrentSize=0;
    MaxSize=n;                                 //初始化堆容量为n
    heapArray=new T[MaxSize];                  //创建堆空间
    BuildHeap();                               //此处进行堆元素的赋值工作
}
```

第一个叶子结点的位置

```
template<class T>                              //判断是否叶结点
bool MinHeap<T>::isLeaf(int pos) const{
    return (pos>=CurrentSize/2)&&(pos<CurrentSize);
}
```

```
template<class T>
int MinHeap<T>::leftchild(int pos) const{
    return 2*pos+1;                          //返回左孩子位置
}

template<class T>                            //返回右孩子位置
int MinHeap<T>::rightchild(int pos) const {
    return 2*pos+2;
}

template<class T>                            //返回当前结点的父结点位置
int MinHeap<T>::parent(int pos) const {
    return  (pos-1)/2 ;                      //DIV
}
```

建堆过程

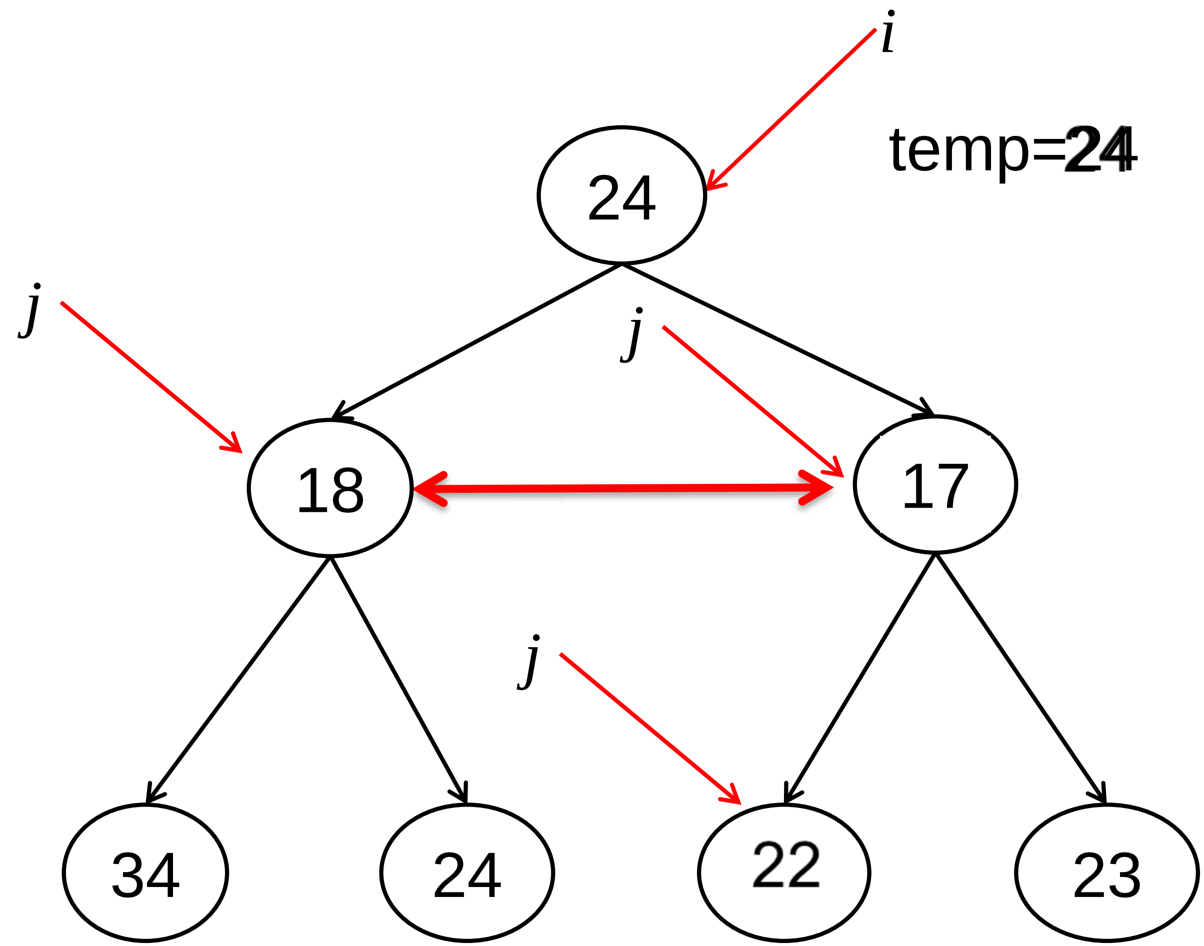
一. 将关键码放到一维数组形成完全二叉树，但并不具备最小堆的特性

- 仅叶子结点代表的子树已经是堆

二. 从完全二叉树的倒数第二层的i（n/2-1）位置开始，从右至左，从下至上依次调整

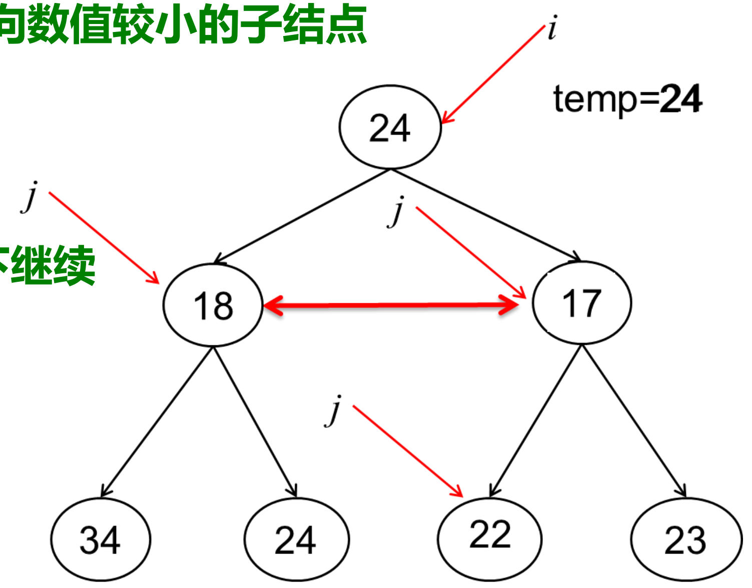
三. 直到树根，整棵完全二叉树就成为一个堆

建堆过程示意



筛选法

```
template <class T>
void MinHeap<T>::SiftDown(int position) {
    int i=position;                              //标识父结点
    int j=2*i+1;                                 //标识关键值较小的子结点
    T    temp=heapArray[i];                      //保存父结点
    while (j<CurrentSize){                       //过筛
        if ((j<CurrentSize-1)&&(heapArray[j]>heapArray[j+1]))
            j++;                                 //j指向数值较小的子结点
        if (heapArray[j] < temp ){
            heapArray[i]=heapArray[j];
            i=j;  j=2*j+1;                       //向下继续
        }  else break;
    }
    heapArray[i]=temp;
}
```

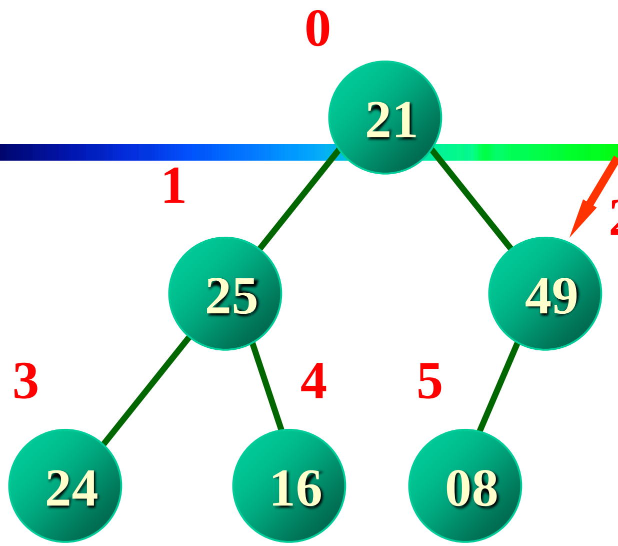


建堆

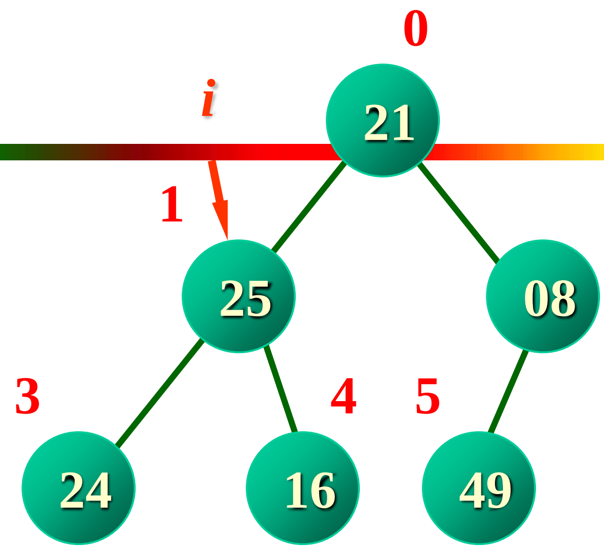
- 从堆的最后一个分支结点heapArray[CurrentSize/2-1]开始，自底向上、自右向左逐步把以各分支结点为根的子树调整成堆

```
template<class T>
void MinHeap<T>::BuildHeap() {
    for (int i=CurrentSize/2-1; i>=0; i--)        //反复调用筛选函数
        SiftDown(i);
}
```

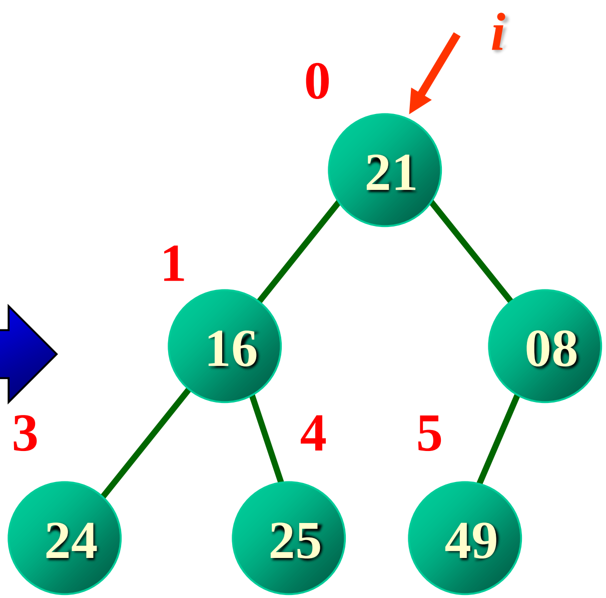
最后一个分支结点位置



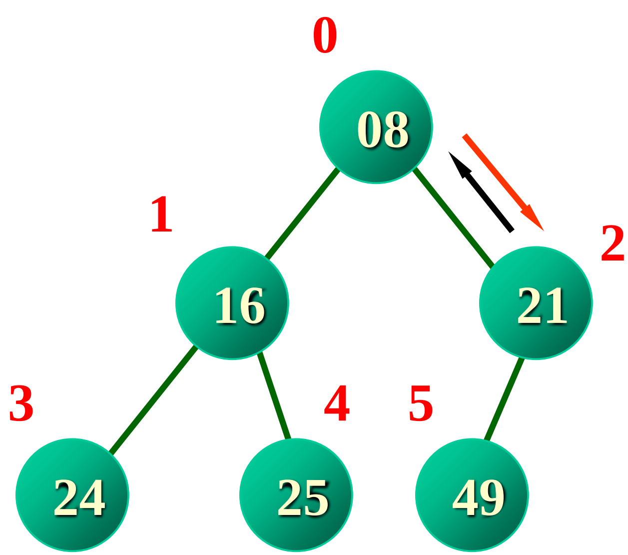
初始排序码集合



i =2时的局部调整



i = 1 时的局部调整



i = 0 时的局部调整形成最小堆

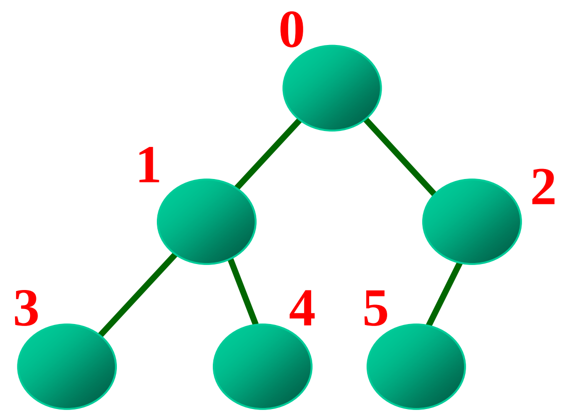
插入新元素

```
template <class T>                          //向堆中插入新元素
bool MinHeap<T>::Insert(const T& newNode) {
    if (CurrentSize==MaxSize)               //堆空间已经满
        return FALSE;
    heapArray[CurrentSize]=newNode;
    SiftUp(CurrentSize);                    //向上调整
    CurrentSize++;
}
```

先把新节点插入堆的最后位置，然后向上调整Siftup，使之成堆

向上筛选调整堆

```
template<class T>   //从position向上开始调整，使序列成为堆
void MinHeap<T>::SiftUp(int position) {
    int temppos=position;
    T temp=heapArray[temppos];
    //请比较父子结点直接swap的方法
    while ((temppos>0)&&(heapArray[parent(temppos)]>temp)) {
        heapArray[temppos]=heapArray[parent(temppos)];
        temppos=parent(temppos);
    }
    heapArray[temppos]=temp;
}       //从叶子节点到根的路径上执行有序插入算法!
```

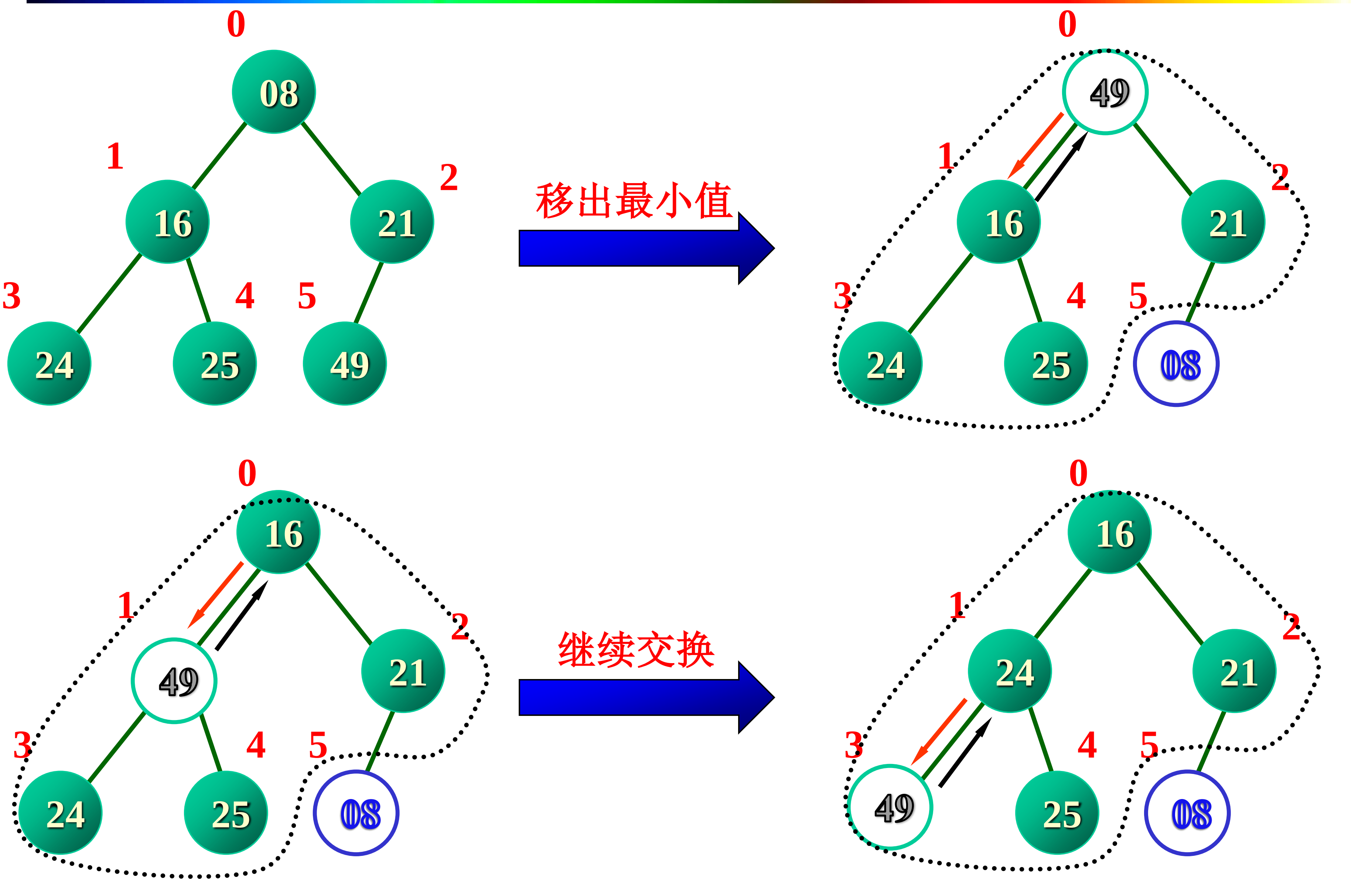


移出最小值

- 移出最小值(根结点)后，剩下的n-1个结点仍要求符合堆性质
- 将堆中最后一个位置上的元素移到根节点，利用siftdown调整

```
template<T>  T& MinHeap<T>::RemoveMin() { //从堆顶删除最小值
    if (CurrentSize==0){                          //空堆
        cout<<"Can't Delete";   exit(1);
    }
    else {
        swap(0,--CurrentSize);                    //交换堆顶和最后一个元素
        if (CurrentSize>1)                        // <=1就不要调整了
            SiftDown(0);                          //从堆顶开始筛选
        return heapArray[CurrentSize];
    }
}
```

示例



0
08
1
16
2
21
3
24
4
25
5
49
移出最小值
0
49
1
16
2
21
3
24
4
25
5
08
0
16
1
49
2
21
3
24
4
25
5
08
继续交换
0
16
1
24
2
21
3
49
4
25
5
08

删除元素

```
template<class T>                          // 删除给定下标元素
bool MinHeap<T>::Remove(int pos, T& node){
    if ((pos<0)||(pos>=CurrentSize))
        return false;
    T temp=heapArray[pos];                 //指定元素置于最后
    heapArray[pos]=heapArray[--CurrentSize];
    SiftUp(pos);                           //上升筛
    SiftDown(pos);                         //向下筛
    node=temp;
    return true;
}
```

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

建堆效率

- n个结点堆的高度为log$n$，第i层上的结点数最多为$2^i (i \geq 0)$
- 建堆过程中，每个非叶子结点都调用一次SiftDown算法，而每次最多向下调整到最底层，即第i层上的结点向下调整到最底层的调整次数为log$n$ – i。
- 因此，建堆的计算时间为

$$\sum_{i=0}^{\log n} 2^i \cdot (\log n - i)$$

令j = logn – i，代入上式得

$$\sum_{i=0}^{\log n} 2^i \cdot (\log n - i) = \sum_{j=0}^{\log n} 2^{\log n - j} \cdot j = \sum_{j=0}^{\log n} n \cdot \frac{j}{2^j} < 2n$$

建堆效率

- 建堆算法的时间复杂度是**O(n)**，即线性时间内把一个无序序列转化成堆。
- 由于堆有**log n**层深，插入结点、删除普通元素和删除最小元素的平均时间代价和最差时间代价都是**O(log n)**

5.5.2 优先队列

- 优先队列(priority queue)是0个或多个元素的集合，每个元素有一个关键码值，执行查找、插入和删除操作。
- 优先队列的主要特点是**从一个集合中快速地查找并移出具有最大值或最小值的元素。**
  - 最小优先队列，适合查找和删除最小元素；
  - 最大优先队列中，适合查找和删除最大元素。
- 堆是优先队列的一种自然的实现方法

5.6 Huffman树及其应用

- 计算机二进制编码
  - ASCII 码，中文编码等
- 等长编码
  - 假设所有编码都等长，**表示 n 个不同的字符需要 $\log_k n$ 位**
  - 字符的使用频率相等
- 频率不等的字符，可以利用字符的出现频率来编码
  - 经常出现的字符的编码较短，不常出现的字符编码较长

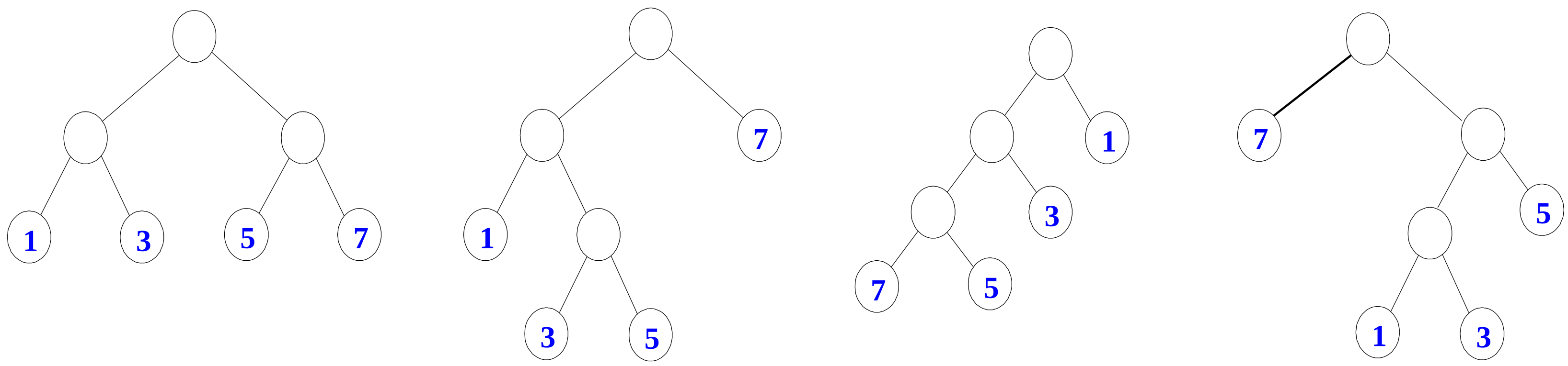
Z	K	F	C	U	D	L	E
2	7	24	32	37	42	42	120

5.6.1 Huffman编码树

- 回顾：扩充二叉树、外部路径长度
- 一个具有n个外部结点的扩充二叉树
  - 每个外部结点K_i有一个w_i与之对应，称为该外部结点的权
  - 带权外部路径长度：二叉树叶结点带权外部路径长度总和

$$WPL = \sum_{i=0}^{n-1} w_i * l_i$$

下图所示二叉树带权路径长度分别为:



(a) (b) (c) (d)

（a）WPL = 1×2+3×2+5×2+7×2 = 32

（b）WPL = 1×2+3×3+5×3+7×1 = 33

（c）WPL = 7×3+5×3+3×2+1×1 = 43

（d）WPL = 1×3+3×3+5×2+7×1 = 29

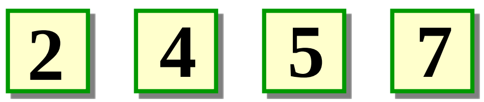
- **定义**：具有最小带权路径长度的二叉树称作**哈夫曼（Huffman）树（或称最优二叉树）**
- 图（d）的二叉树是一棵哈夫曼树

建立Huffman编码树

- 首先，按照“权重”（例如频率）将字母排为一个有序序列
- 接着，拿走前两个字母（“权”最小的两个字母），再将它们标记为Huffman树的树叶，将这两个树叶标为一个分支结点的两个子女，而该结点的权即为两树叶的权之和。将所得“权”放回序列中适当位置，使“权”的顺序保持
- 重复上述步骤直至序列中剩一个元素，则Huffman树建立完毕

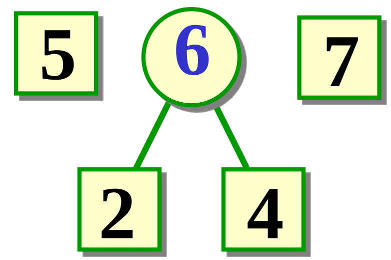
举例：Huffman树的构造

H : {2} {4} {5} {7}



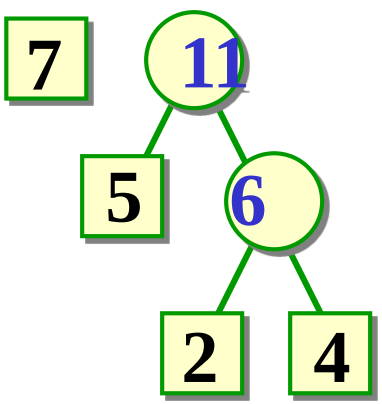
初始

H : {5} {6} {7}



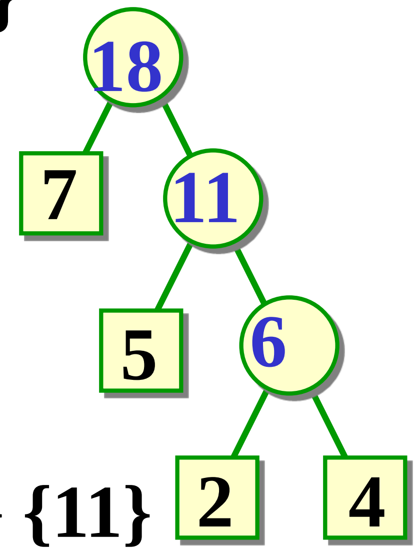
合并{2} {4}

H : {7} {11}



合并{5} {6}

H : {18}



合并{7} {11}

5.6.2 Huffman编码

- Huffman树的一个重要应用是解决数据通信中的二进制编码问题。

  设　　$D=\{d_0, \ldots, d_{n-1}\}$,

  $W=\{W_0, \ldots, W_{n-1}\}$

  D为需要编码的字符集合，W为D中各字符出现的频率，要对D里的字符进行二进制编码，使得：

  $$\sum_{i=0}^{n-1} w_i l_i$$

  最小，其中，li为第i个字符的二进制编码长度。

- 由此可见，设计电文总长度最短的编码问题就转化成了设计字符出现频率作为外部结点权值的Huffman树的问题。

编码过程

- **Huffman编码过程如下：**
  - 用d_0，d_1，…，d_{n-1}作为外部结点构造具有最小带权外部路径长度的扩充二叉树
  - 把从每个结点**引向其左子结点的边标上号码0，引向其右子结点的边标上号码1**
  - 从根结点到每个叶结点路径上的编号（**0 or 1**）连接起来，就是这个外部结点所代表字符的编码。得到的二进制前缀码就称作**Huffman编码**

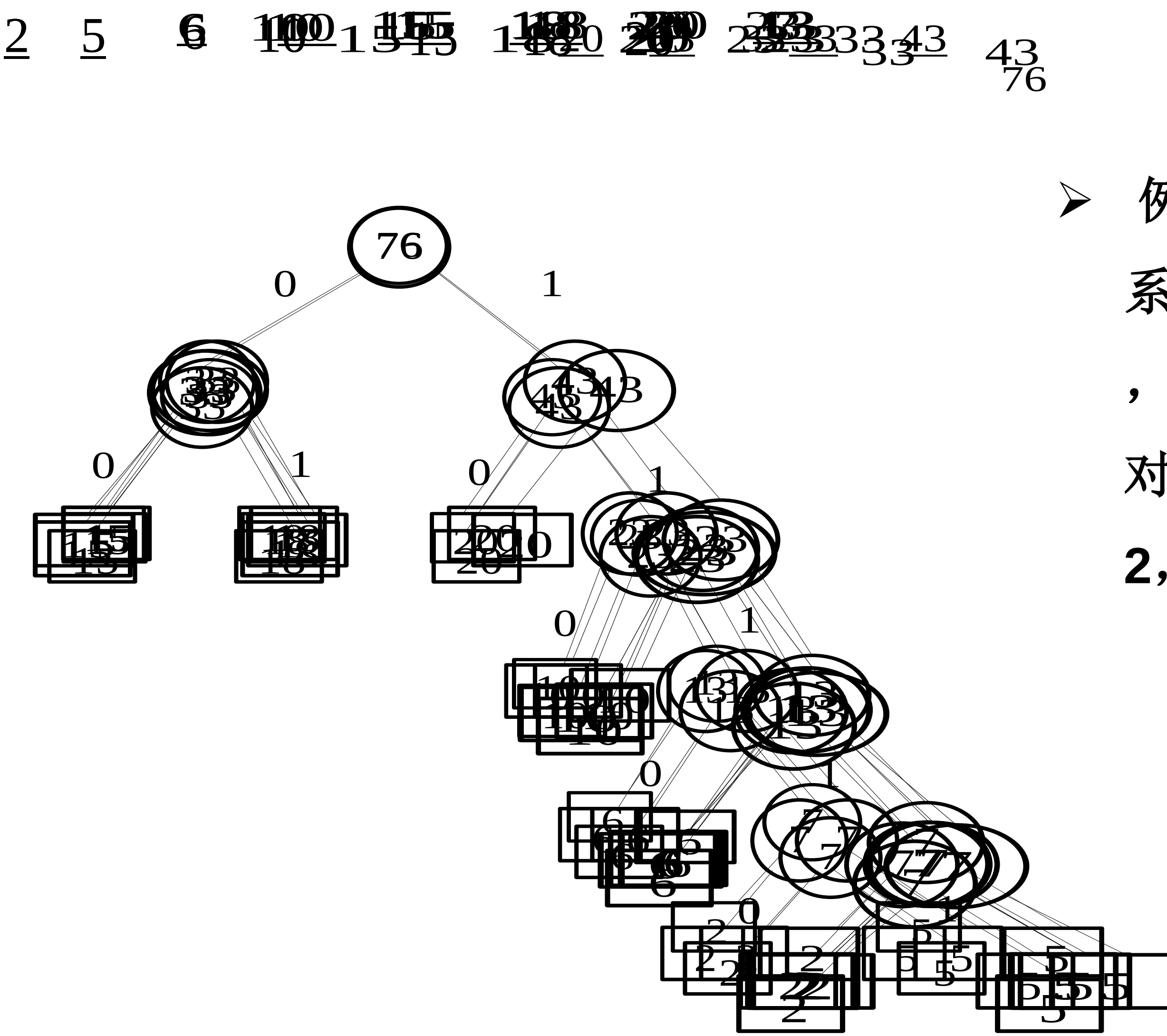
Huffman编码性质

- Huffman编码将代码与字符相联系
  - 不等长编码
  - 代码长度取决于对应字符的相对使用频率或“**权重**”
- **任何一个字符的编码都不是另一个字符编码的前缀**
  - 前缀特性保证了代码串被反编码时，不会有多种可能。

译码

- 用Huffman算法构造出的扩充二叉树给出了各字符的编码，同时也用来**译码**
- 从二叉树的根开始，把二进制编码每一位的值与Huffman树边上标记的0，1相匹配，确定选择左分支还是右分支，直至确定一条到达树叶的路径。
- 一旦到达树叶，就译出了一个字符。然后继续用这棵二叉树继续译出其它二进制编码

示例：编码



- 例如，在一个数据通信系统中使用的字符是**a，b，c，d，e，f，g，**对应的频率分别为**15，2，6，5，20，10，18**

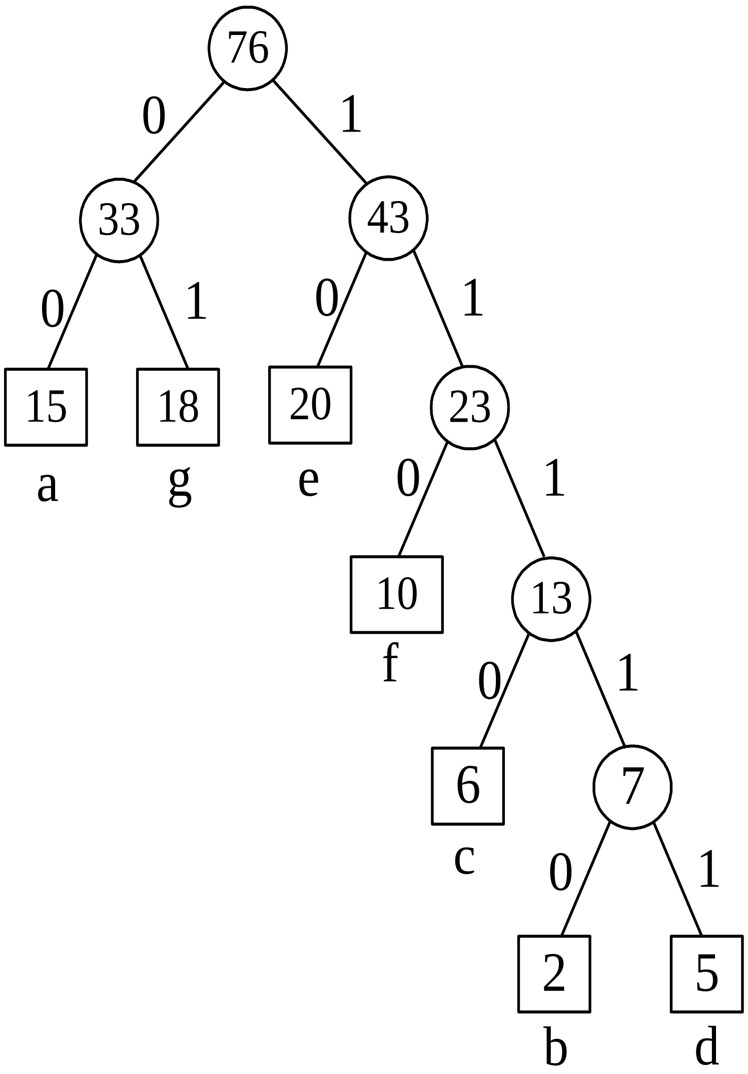
示例

各字符的二进制编码为：

a：00　　b：11110　　c：1110

d：11111　e：10　　　f：110

g：01



示例

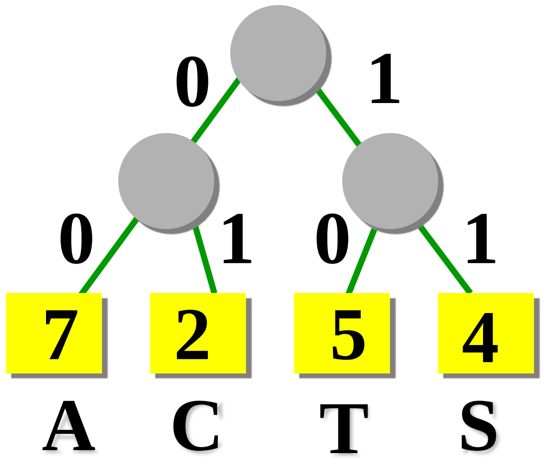
- 设给出一段报文：

  CAST CAST SAT AT A TASA

- 字符集合是 { C, A, S, T }，各个字符出现的频度(次数)是 W＝{ 2, 7, 4, 5 }

- 若给每个字符以等长编码

  A : 00　T : 10　C : 01　S : 11



  - 则总编码长度为 (2+7+4+5) * 2 = 36

- 若按各个字符出现的概率不同而给予不等长编码，可望减少总编码长度

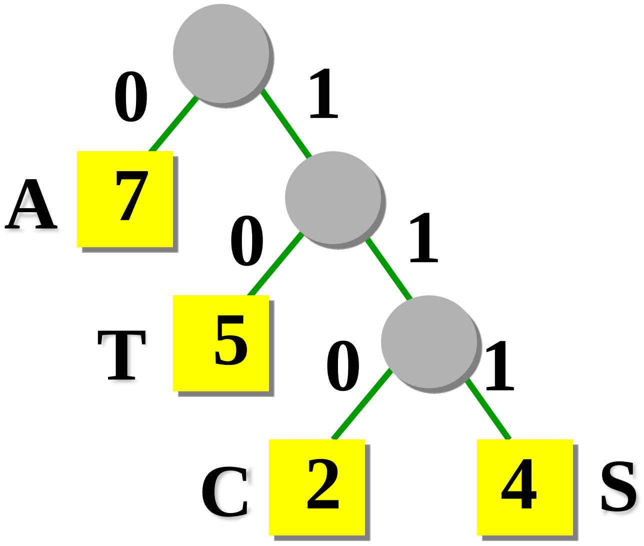
示例

- 以它们为各叶结点上的权值, 建立Huffman树。左分支赋 0，右分支赋 1，得Huffman编码(变长编码)。

  CAST CAST SAT AT A TASA

  A : 0　T : 10　C : 110　S : 111

- 编码长度：7*1+5*2+(2+4)*3 = 35
- 霍夫曼编码: A:0 T:10 C:110 S:111
  - 110011110 1100111101110100100100110
- 等长编码: A:00 T:10 C:01 S:11
  - 0100111001001110110010001000010001100



Huffman树类

```
template <class T> class HuffmanTree  {
private:
    HuffmanTreeNode<T>* root;          //Huffman树的树根
    //把ht1和ht2为根的合并成一棵以parent为根的Huffman子树
    void MergeTree(HuffmanTreeNode<T> &ht1,
             HuffmanTreeNode<T> &ht2, HuffmanTreeNode<T>* parent);
public:
    //构造Huffman树，weight是存储权值的数组，n是数组长度
    HuffmanTree(T weight[],int n);
    virtual ~HuffmanTree()  { DeleteTree(root); }  //析构函数
}
```

Huffman树的构造算法

```
template<class T>
HuffmanTree<T>::HuffmanTree(T weight[], int n) {
    MinHeap<HuffmanTreeNode<T>> heap;          //定义最小值堆
    HuffmanTreeNode<T> *parent,&leftchild,&rightchild;
    HuffmanTreeNode<T> *NodeList = new
    HuffmanTreeNode<T>[n];
    for(int i=0; i<n; i++) {                   //向堆中添加初始元素
        NodeList[i].element = weight[i];
        NodeList[i].parent = NodeList[i].left
                           = NodeList[i].right = NULL;
        heap.Insert(NodeList[i]);
    } //end for
```

```
   for(i=0;i<n-1;i++) {          //通过n-1次合并建立Huffman树
     parent=new HuffmanTreeNode<T>;
     firstchild=heap. RemoveMin();          //选值最小的结点
     secondchild=heap. RemoveMin();      //选值次小的结点
     MergeTree(firstchild,secondchild,parent); //权值最小树合并
     heap.Insert(*parent);                  //把parent插入到堆中去
     root=parent;                           //建立根结点
    }//end for
   delete []NodeList;
}
```

K叉Huffman树

- 如何提高Huffman编码和译码的效率?

- 如何构造K叉Huffman树?

- 如果字符个数不是K的整数倍怎么办?

本 章 小 结

二叉树

应用 ↓

Huffman树

↓

Huffman编码

- 1、定义和性质　　性质有3+2条
- 2、存储结构
  - 顺序结构
  - 链式结构
    - 二叉链表
    - 三叉链表
- 3、遍历
  - 先序遍历
  - 中序遍历
  - 后序遍历
  - 层次遍历
- 4、二叉搜索树

再见…

联系信息：

电子邮件：gjsong@pku.edu.cn

电　　话： 62754785

办公地点：理科2号楼2307室