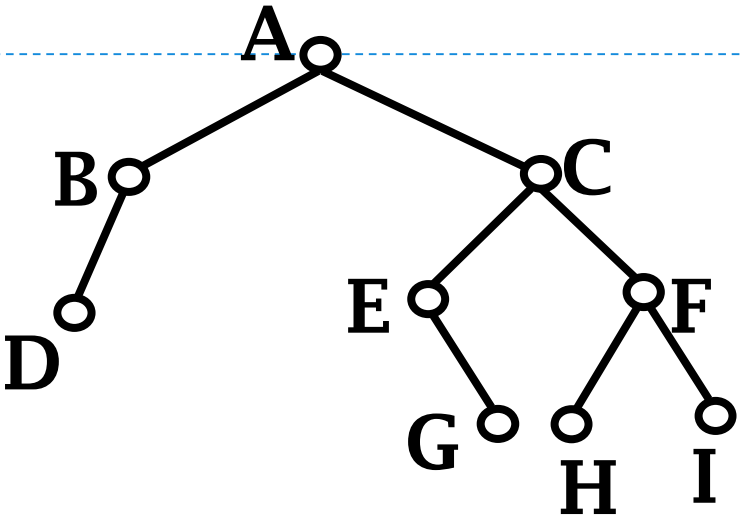
第五章　二叉树

张铭　主讲
采用教材：张铭，王腾蛟，赵海燕 编写
高等教育出版社，2008. 6 （“十二五”国家级规划教材）

http://jpk.pku.edu.cn/course/sjjg/
https://www.icourse163.org/course/PKU-1002534001

第五章 二叉树

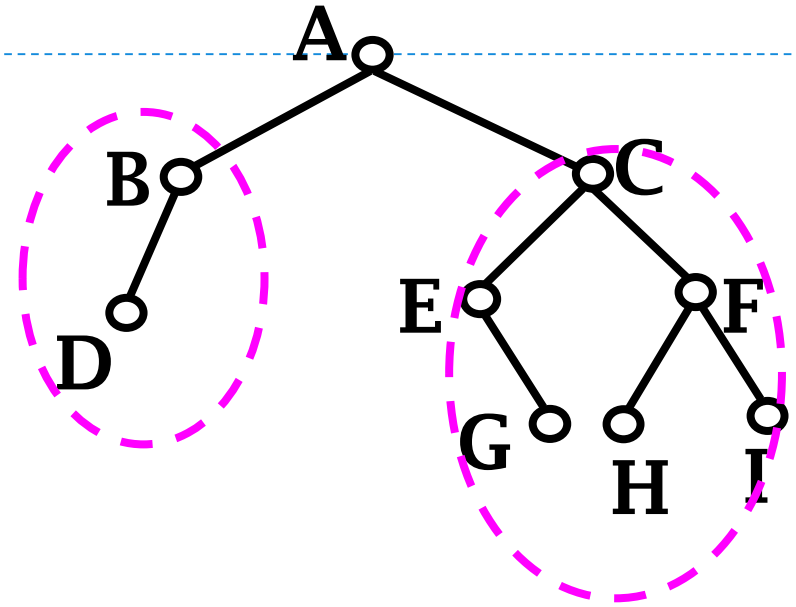


- 二叉树的概念
- 二叉树的抽象数据类型
  - 深度优先搜索
  - 宽度优先搜索
- 二叉树的存储结构
- 二叉搜索树
- 堆与优先队列
- Huffman树及其应用

二叉树的概念

- **二叉树(binary tree)的定义**
  - 二叉树由**结点的有限集合**构成
  - 这个有限集合或者为**空集** (empty)
  - 或者为由一个**根结点** (root) 及两棵互不相交、分别称作这个根的**左子树** (left subtree) 和**右子树** (right subtree) 的二叉树组成的集合

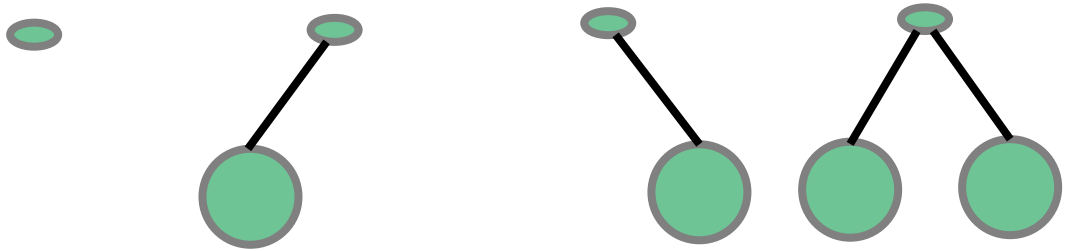
注意：每个结点至多两棵子树，且有左右之分，不能随意交换



二叉树有哪几种基本形态呢?

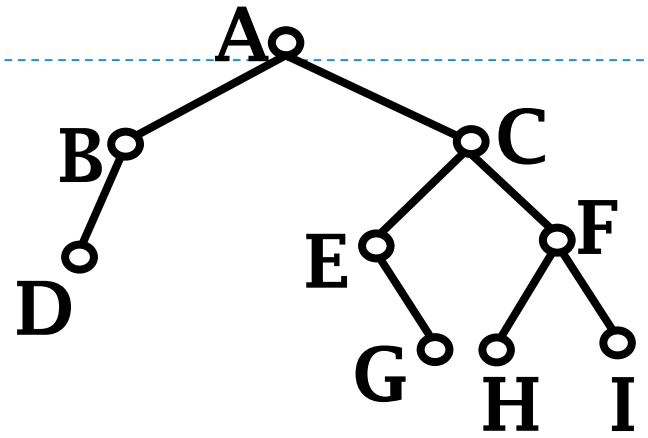
二叉树的五种基本形态

- **二叉树可以为空**，结点的左、右子树也可空



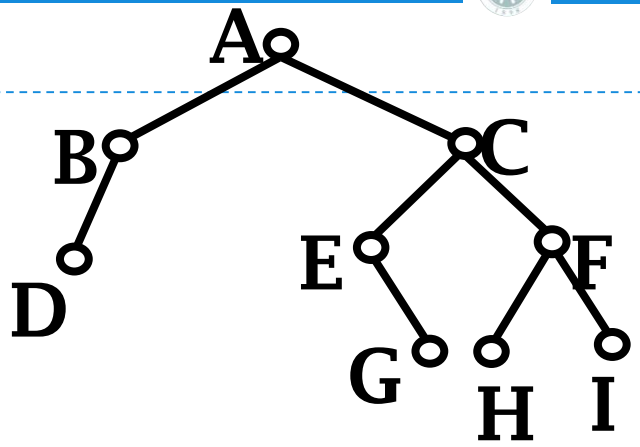
(a)空　(b)独根　(c)空右　(d)空左　(e)左右都不空

二叉树相关术语



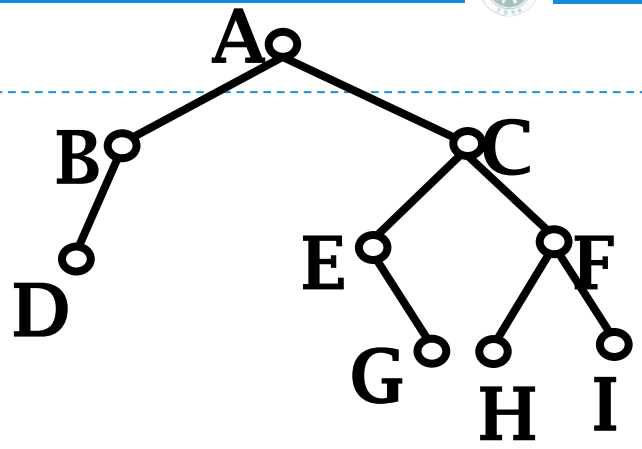
- **结点**
  - **子结点、父结点、最左子结点**
    - 若 <k，k′ >∈ r，则称 k 是 k′ 的父结点(或“父母”)，而 k′ 则是 k 的 子结点(或“儿子”、“子女”)
  - **兄弟结点、左兄弟、右兄弟**
    - 若有序对 <k，k′ >及 <k，k″ >∈ r，则称 k′ 和 k″ 互为兄弟
  - **分支结点、叶结点**
    - 没有子树的结点称作 叶结点（或树叶、终端结点）
    - 非终端结点称为分支结点

二叉树相关术语



- **边：** 两个结点的有序对，称作 **边**
- **路径、路径长度**
  - **除结点 k_0外的任何结点 $k \in K$，都存在一个结点序列 k_0，k_1，…，k_s，使得 k_0 就是树根，且 $k_s=k$，其中有序对 $<k_{i-1}, k_i> \in r$ $(1 \le i \le s)$。这样的结点序列称为从根到结点 k 的一条路径，其路径长度为 s (包含的边数)**
- **祖先、后代**
  - 若有一条由 k 到达 k_s 的路径，则称 k 是 k_s 的 **祖先**，k_s 是 k 的 **后代（或称子孙）**

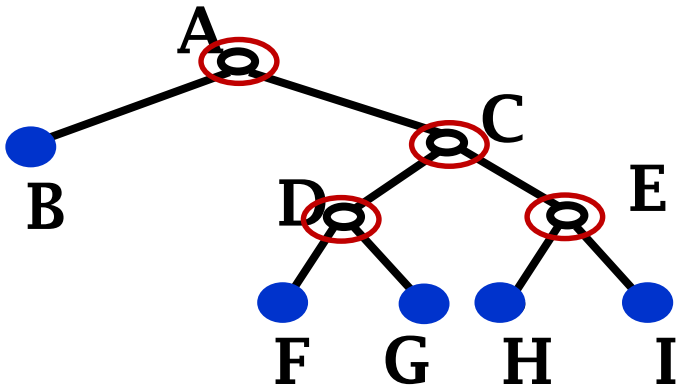
二叉树相关术语



- **层数**：根为第 0 层
  - 其他结点的层数等于其父结点的层数加 1
- **深度**：层数最大的叶结点的层数
- **高度**：层数最大的叶结点的层数加 1

满二叉树（NIST标准定义）

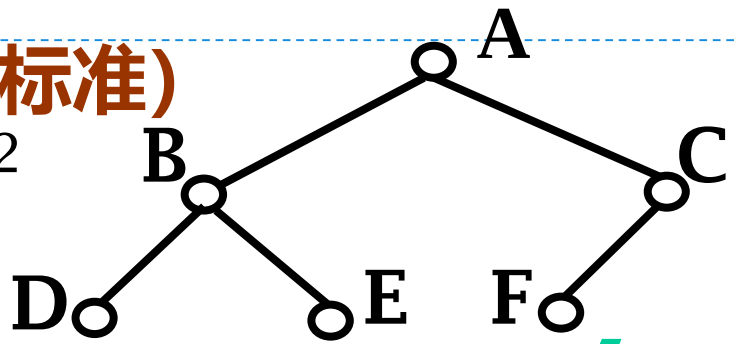
- 满二叉树的所有结点，或为叶结点，或恰有两棵非空子树



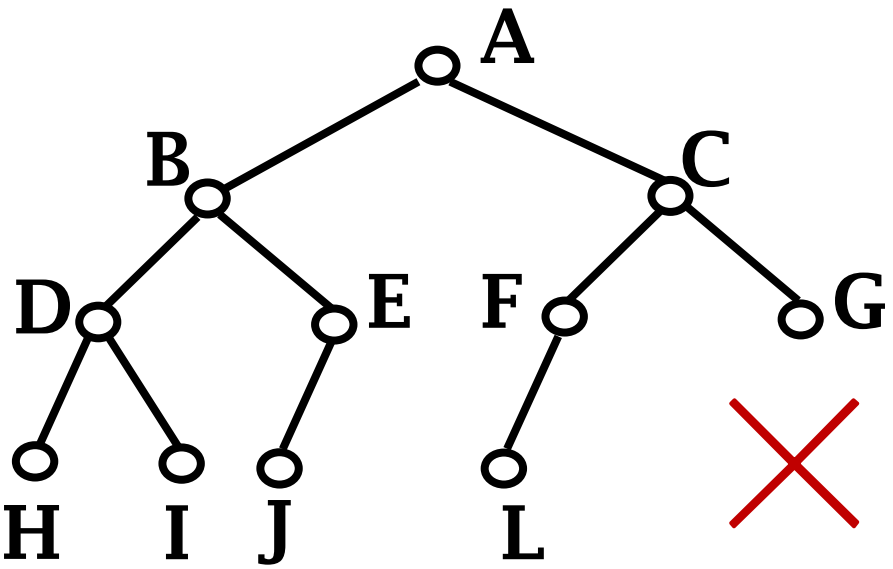
5.1 二叉树的概念

完全二叉树（NIST标准）

- 最多只有最下面的两层结点度数可以小于2
- 最下一层的结点都集中最左边



√

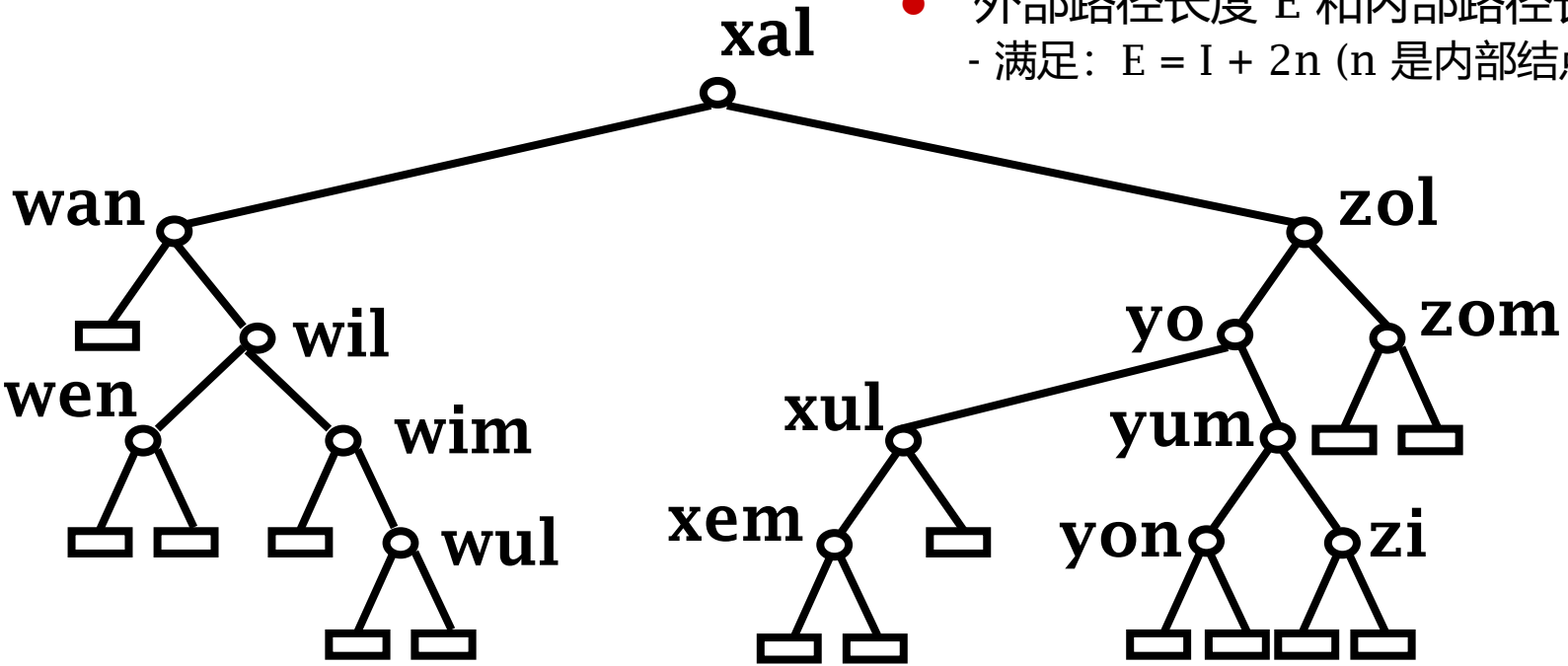


×

5.1 二叉树的概念

扩充二叉树

- **所有空子树，都增加空树叶**
  - 不满的 分支结点 增加 1 个 空叶结点
  - 叶结点 增加 2 个 空叶结点
- 外部路径长度 E 和内部路径长度 I
  - 满足：E = I + 2n (n 是内部结点个数)



二叉树的主要性质

- 性质1. 在二叉树中，第i层上最多有 2^i 个结点（i≥0）
- 性质2. 深度为 k 的二叉树至多有 $2^{k+1}-1$ 个结点（k≥0）

  其中深度(depth)定义为二叉树中层数最大的叶结点的层数
- 性质3. 一棵二叉树，若其终端结点数为 n_0，度为2的结点数为 n_2，

  则 $n_0=n_2+1$
- 性质4. **满二叉树定理**：非空满二叉树树叶数目等于其分支结点数加1
- 性质5. 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
- 性质6. 有n个结点（n>0）的完全二叉树的高度为 $\lceil \log_2 (n+1) \rceil$

  (深度为 $\lceil \log_2 (n+1) \rceil - 1$)

5.2 二叉树性质与ADT

二叉树性质

性质1. 非空二叉树的第 i（i≥0）层 至多有 2^i 个结点

证明（归纳法）

1. 归纳基础：i = 0， 结点数 1 = 2^0；
2. 归纳假设：假设对于所有的 j (0≤j ≤i)， 命题成立，即， j 层结点数至多有2^j；
3. 归纳结论：对于 i=j+1，结点数至多为

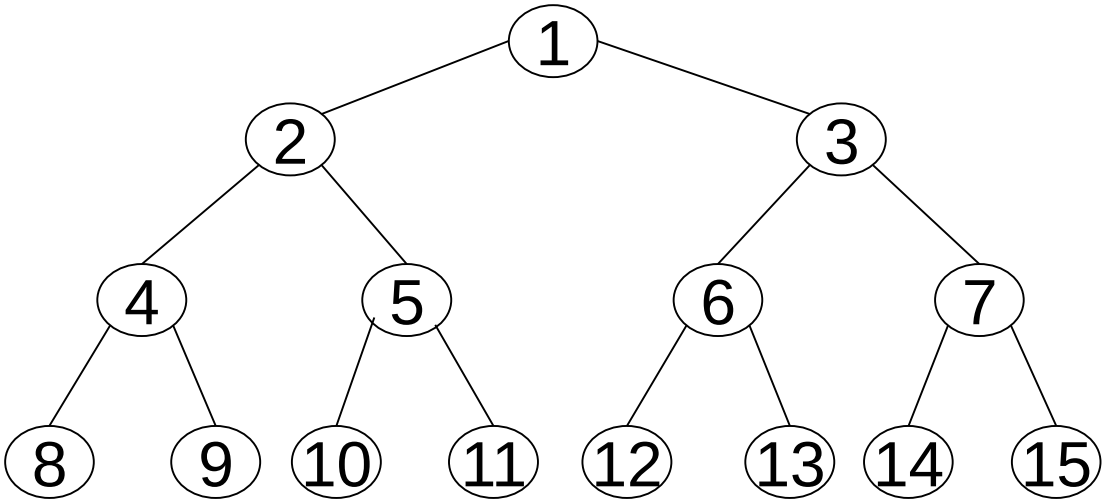
$$2*2^j=2^{j+1}$$

5.2 二叉树性质与ADT

高度为4的二叉树结点数目示例

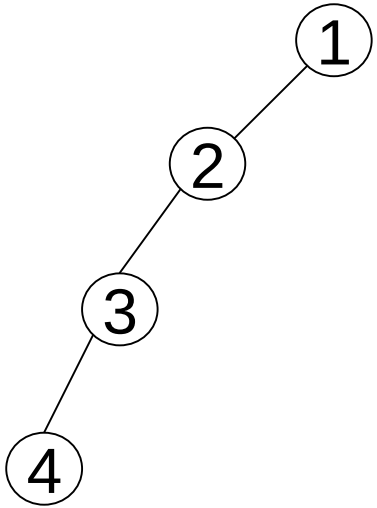
结点最多的二叉树

$2^0+2^1+2^2+2^3$



结点最少二叉树

1+1+1+1



5.2 二叉树性质与ADT

二叉树的性质

性质2. 深度为 k 的二叉树至多有 $2^{k+1}-1$ 个结点 $(k \geq 0)$

证明: 假设第 *i* 层上的最大结点个数为m_i，由性质 1 可知，深度为 *k* 的二叉树中最大的结点个数

$$M = \sum_{i=0}^{k} m_i \leq \sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

5.2 二叉树性质与ADT

二叉树的性质

性质3. 一棵二叉树中度为 0 的结点数目 n_0 比度为 2 的结点数目 n_2 多1，即 $n_0 = n_2 + 1$

证明： n_0，n_1，n_2分别表示n个结点中度为0、1、2的结点数，则有

$$n = n_0 + n_1 + n_2 \quad (1)$$

若用e 表示树的边数， 则有 $n = e + 1$

边均由度为 1 和 2 的结点射出的，故有 $e = n_1 + 2 \cdot n_2$， 即，

$$n = e + 1 = n_1 + 2 \cdot n_2 + 1 \quad (2)$$

由（1）和（2）得

$$n_0 + n_1 + n_2 = n_1 + 2 \cdot n_2 + 1$$

即， $n_0 = n_2 + 1$

5.2 二叉树性质与ADT

二叉树的性质

性质4. 满二叉树定理：非空满二叉树的叶结点数目为其分支结点数加 1

证明：设二叉树结点数为 n，其中叶结点数为 m、分支结点数为 b，则有

$$n\text{（总结点数）} = m\text{（叶）} + b\text{（分支）} \quad (1)$$

∵ 每个分支恰有两个子结点（满），故有2*b条边；n个结点， 故

$$n - 1 = 2b \quad (2)$$

∴ 由(1) 和 (2) 可得 n-1= m+b -1 = 2b，即

$$m(\text{叶}) = b\text{（分支）} + 1$$

5.2 二叉树性质与ADT

二叉树的性质

性质5. 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：　设二叉树为T，将其所有空子树替换为叶结点，所得的扩充满二叉树为T’，亦即，T 的原有结点变为T’ 的分支结点

根据**满二叉树定理**，T’ 的叶结点数目等于T 的结点个数 加 1；每个新添加的叶结点对应T的一个空子树

∴ T 中空子树数目等于T 中结点数加1

5.2 二叉树性质与ADT

二叉树的性质

性质6. 具有 n 个结点的完全二叉树的 高度 和 深度 分别为 $\lceil \log_2 (n+1) \rceil$ 和 $\lceil \log_2 (n+1) \rceil - 1$

证明：设其深度为k，则有

$$n = 2^0 + 2^1 + 2^2 + \ldots + 2^{k-1} + m_k = 2^k - 1 + m_k \qquad (1)$$

故 $2^k - 1 < n \le 2^{k+1} - 1$ (性质2) (2)

$$2^k < n+1 \le 2^{k+1}$$

$$k < \log_2(n+1) \le k+1$$

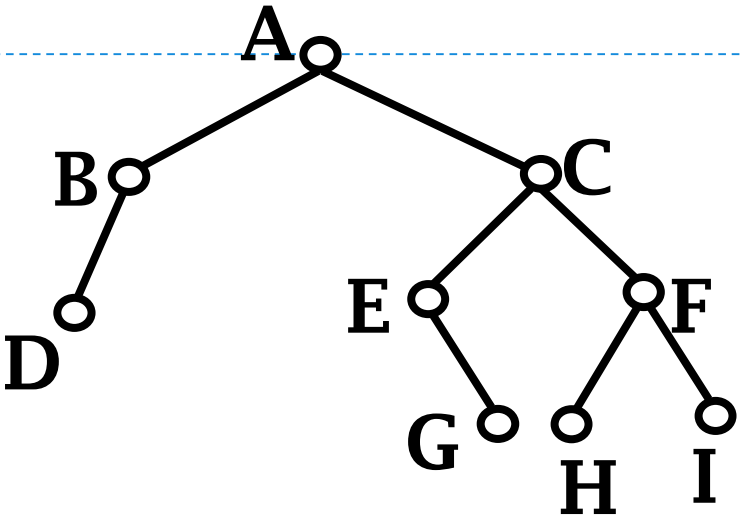
$$\therefore \quad k = \lceil \log_2 (n+1) \rceil - 1$$

// 性质2. 深度为 k 的二叉树至多有 $2^{k+1}-1$ 个结点

思　考

- 扩充二叉树和满二叉树的关系
- 二叉树主要六个性质的关系

第五章 二叉树



- 二叉树的概念
- 二叉树的抽象数据类型
  - 深度优先搜索
  - 宽度优先搜索
- 二叉树的存储结构
- 二叉搜索树
- 堆与优先队列
- Huffman树及其应用

5.2 二叉树性质与ADT

二叉树抽象数据类型

- **逻辑结构 + 运算**
- 针对整棵树的运算
  - 初始化二叉树
  - 合并两棵二叉树
  - 遍历（周游）二叉树
- 围绕结点的运算
  - 访问某个结点的左子结点、右子结点、父结点
  - 访问结点存储的数据

5.2 二叉树性质与ADT

二叉树结点的ADT

```
    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
    BinaryTreeNode<T>* Parent();                       // 返回父结点
    BinaryTreeNode<T>* LeftSibling();                  // 返回左兄结点
    BinaryTreeNode<T>* RightSibling();                 // 返回右兄结点
    BinaryTreeNode<T>& operator =                      // 重载赋值操作符
            (const BinaryTreeNode<T>& Node);
    bool  isLeaf() const;                              // 判断是否为叶结点
    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
};
```

5.2 二叉树性质与ADT

二叉树的ADT

```
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
```

5.2 二叉树性质与ADT

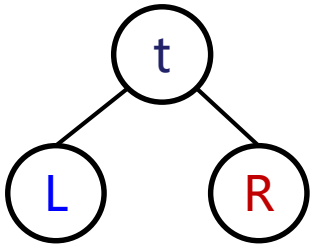
二叉树的ADT

```
  void CreateTree(const T& info,
      BinaryTree<T>& leftTree, BinaryTree<T>&  rightTree);          // 构造新树
   void PreOrder(BinaryTreeNode<T> *root);                  // 前序遍历二叉树或其子树
   void InOrder(BinaryTreeNode<T> *root);                   // 中序遍历二叉树或其子树
   void PostOrder(BinaryTreeNode<T> *root);                 // 后序遍历二叉树或其子树
   void LevelOrder(BinaryTreeNode<T> *root);                // 按层次遍历二叉树或其子树
   void DeleteBinaryTree(BinaryTreeNode<T> *root);          // 删除二叉树或其子树
};
```

5.3 二叉树的遍历

深度优先遍历二叉树

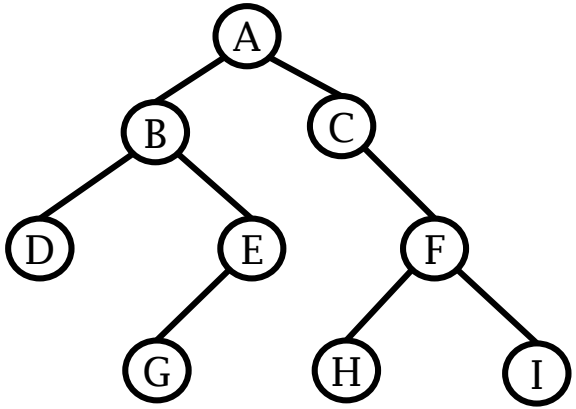
- 一棵二叉树由三部分组成
  - 根结点；根结点的左子树；根结点的右子树
- 通过**变换根结点**的遍历顺序，可有**3种方案**：
  - **前序遍历** (**tLR**次序, preorder traversal)
    - 访问根结点；前序遍历左子树；前序遍历右子树
  - **中序遍历** (**LtR**次序, inorder traversal)，也称 对称序
    - 中序遍历左子树；访问根结点；中序遍历右子树
  - **后序遍历** (**LRt**次序, postorder traversal)
    - 后序遍历左子树；后序遍历右子树；访问根结点



5.3 二叉树的遍历

深度优先遍历二叉树

- 前序遍历序列： A B D E G C F H I
- 中序遍历序列： D B G E A C H F I
- 后序遍历序列： D G E B H I F C A



结点在**某种序列**下的前驱/后继

深度优先遍历二叉树（递归）

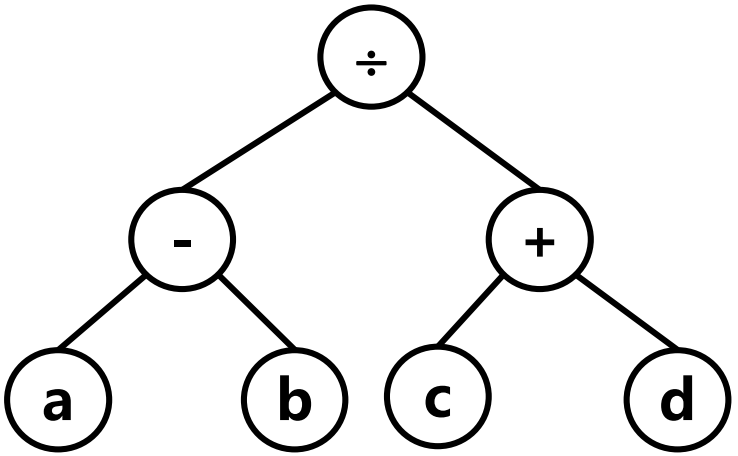
```
// 简洁，推荐使用； 当前编译系统优化效率很好
template<class T>
void BinaryTree<T>::DepthOrder (BinaryTreeNode<T>* root)  {
  if(root!=NULL) {
        Visit(root);                          // 前序
        DepthOrder(root->leftchild());        // 递归访问左子树
        Visit(root);                          // 中序
        DepthOrder(root->rightchild());       // 递归访问右子树
        Visit(root);                          // 后序
    }
}
```

5.3 二叉树的遍历

表达式二叉树

- 前序、后序和中序次序对图周游可分别得到如下的结点序列：

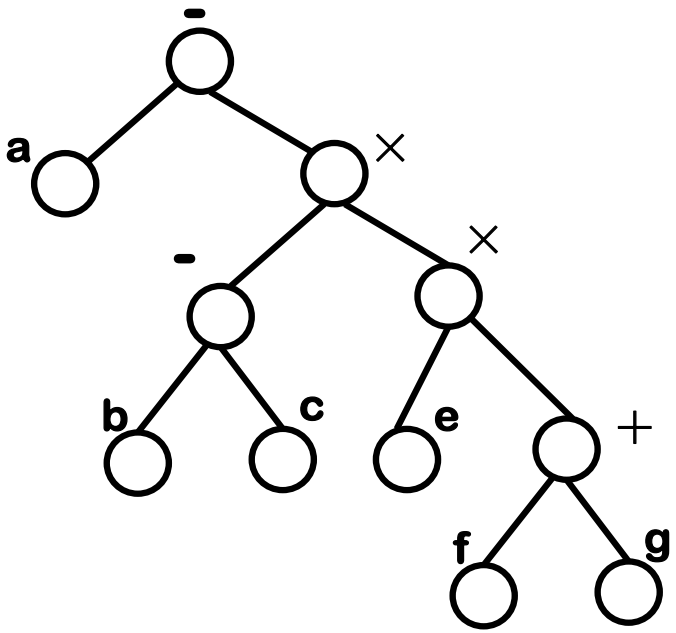
前序序列：　÷ - a b + c d
　　前缀表示（波兰表示法）
后序序列：　a b – c d + ÷
　　后缀表示（逆波兰表示法）
对称序列：　(a – b) ÷ (c + d)
　　中缀表示



补充：二叉树遍历框架讨论

问题：打印一棵表达式二叉树对应的中缀形式的表达式

要求：打上必要的括号以保持原有的运算顺序，但括号不要冗余



例如，下面是右图对应的表达式

a – (b – c) × (e × (f + g))

5.3 二叉树的遍历

二叉树深度优先遍历

- 实现方式：非递归算法
  - 有些资源受限的应用环境
  - 理解栈的工作原理
  - 掌握深度优先遍历的回溯特点

- 关键：设置一个栈，记录尚待遍历的结点，以备后续访问

非递归前序遍历二叉树

- **思想：**
  - 遇到一个结点，访问之，并将其非空右子结点压栈；
  - 下降去前序遍历其左子树；
  - 按前序方式遍历完左子树后，从栈顶弹出一个结点，访问之，并继续按前序方式遍历其右子树

5.3 二叉树的遍历

非递归前序遍历二叉树

```
template<class T> void BinaryTree<T>::PreOrderWithoutRecursion(BinaryTreeNode<T> *root) {
    using std::stack;                                    // 使用STL中的栈
    stack<BinaryTreeNode<T>* > aStack;
    BinaryTreeNode<T> *pointer = root;
    aStack.push(NULL);                                   // 栈底监视哨
    while (pointer) {                                    // 或者!aStack.empty()
        Visit(pointer->value());                         // 访问当前结点
        if (pointer->rightchild() != NULL)               // 非空右子入栈
            aStack.push(pointer->rightchild());
        if (pointer->leftchild() != NULL)
            pointer = pointer->leftchild();              // 左路下降
        else {                                           // 左子树访问完毕，转向访问右子树
            pointer = aStack.top();                      // 获得栈顶元素
            aStack.pop();                                // 栈顶元素退栈
        }
} }
```

5.3 二叉树的遍历

非递归中序遍历二叉树

- **基本思想**
  - 遇到一个结点，将其压栈后，遍历其左子树；
  - 遍历完左子树后，从栈顶弹出该结点并访问之；
  - 然后下降到其右子树再去遍历其右子树

```
template<class T>  void
BinaryTree<T>::InOrderWithoutRecusion(BinaryTreeNode<T>*
root) {
    using std::stack;                          // 使用STL中的stack
    stack<BinaryTreeNode<T>* > aStack;
    BinaryTreeNode<T>* pointer = root;
    while (!aStack.empty() || pointer) {
       if (pointer ) {
         // Visit(pointer->value());      // 前序访问点
          aStack.push(pointer);           // 当前结点地址入栈
         // 当前链接结构指向左孩子
          pointer = pointer->leftchild();
```

```
    } //end if
    else {          //左子树访问完毕，转向访问右子树
        pointer = aStack.top();
        aStack.pop();                          //栈顶元素退栈
        Visit(pointer->value());               //访问当前结点
        //当前链接结构指向右孩子
        pointer=pointer->rightchild();
    } //end else
  } //end while
}
```

5.3 二叉树的遍历

非递归后序遍历二叉树

- **基本思想**
  - 遇到一个结点，将其压栈，遍历其左子树
  - 遍历结束后，尚不能马上访问处于栈顶的该结点，须向右下降去遍历其右子树
  - 遍历完右子树后方可从访问栈顶元素并弹出
- **问题：**
  - 从栈中取出一个结点时， **如何判断**

    从 左子树 还是 右子树 返回?

5.3 二叉树的遍历

非递归后序遍历二叉树

- **解决方案**：栈中的元素增加一个**特征位**，以**区别**栈顶弹出结点时是从其左子树（仍需遍历右子树），还是从右子树返回
  - ***L*** 表示进入左子树并从其返回
  - ***R*** 表示进入右子树并从其返回

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) {
   while (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();
   }
   element = aStack.top();  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 {                                       // 如果从右子树回来
      Visit(pointer->value());                  // 访问当前结点
      pointer = NULL;                           // 置point指针为空，以继续弹栈
   }
 }
}
```

5.3 二叉树的遍历

深度优先遍历二叉树分析

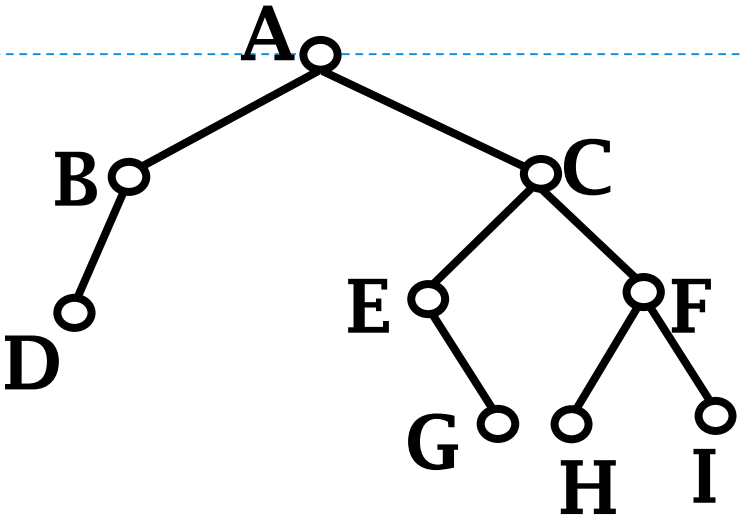
- 时间代价：遍历具有 n 个结点的二叉树 需 O(n)时间
  - 前序和中序，某些结点入栈/出栈一次，不超过 O(n)
  - 后序，每个结点分别从左、右边各入栈/出栈一次 O(n)
  - 前提：各结点处理（函数Visit的执行）时间为常数
- 空间代价： 遍历过程中栈的最大容量，与树的高度成正比
  - **最坏情况**：高度为 n，所需空间复杂度为O(n)
  - **最好情况**：O(1)
  - **平均情况**：O(log n)

5.3 二叉树的遍历

思考

- 非递归遍历的意义何在?
  - 后序遍历时，栈中结点有何规律?
  - 栈中存放了什么?
- 前序、中序、后序遍历算法框架的通用性
  - 例如， 后序遍历框架是否支持前序、中序访问?
  - 若支持，怎么改动?
- 若已知前序、中序、后序之一可恢复二叉树的结构吗？如若不能，那么哪几种结合可以恢复二叉树的结构?
- 已知某二叉树的中序序列为 {A, B, C, D, E, F, G}，后序序列为 {B, D, C, A, F, G, E}；请给出其前序序列。

第五章 二叉树

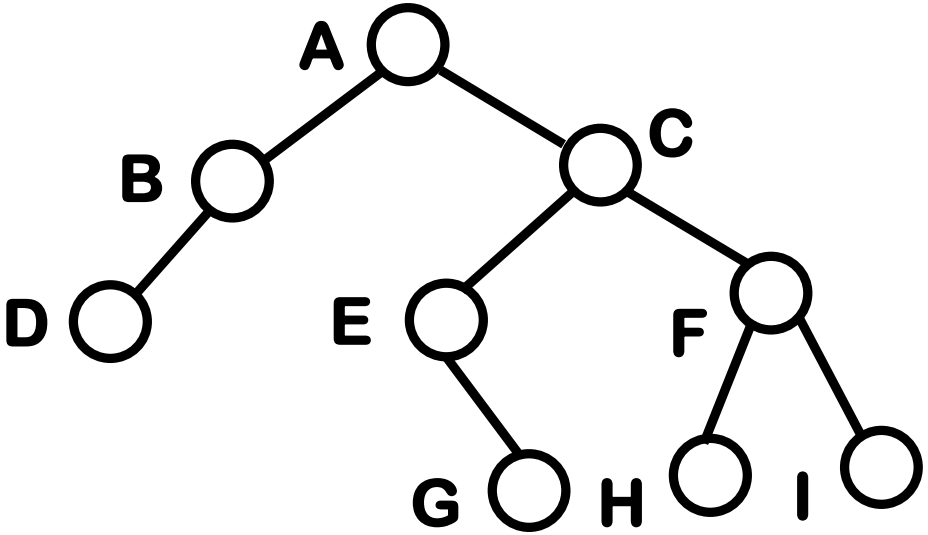


- 二叉树的概念
- 二叉树的抽象数据类型
  - 深度优先搜索
  - 宽度优先搜索
- 二叉树的存储结构
- 二叉搜索树
- 堆与优先队列
- Huffman树及其应用

5.3 二叉树的遍历

广度优先遍历二叉树

- 从二叉树的第0层（根结点）开始，自上至下逐层遍历；在同一层中，按照从左到右的顺序对结点逐一访问



ABCDEFGHI

使用中间数据结构?

宽度优先遍历二叉树算法

```
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.front();                // 取队列首结点
          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
 } }
```

5.3 二叉树的遍历

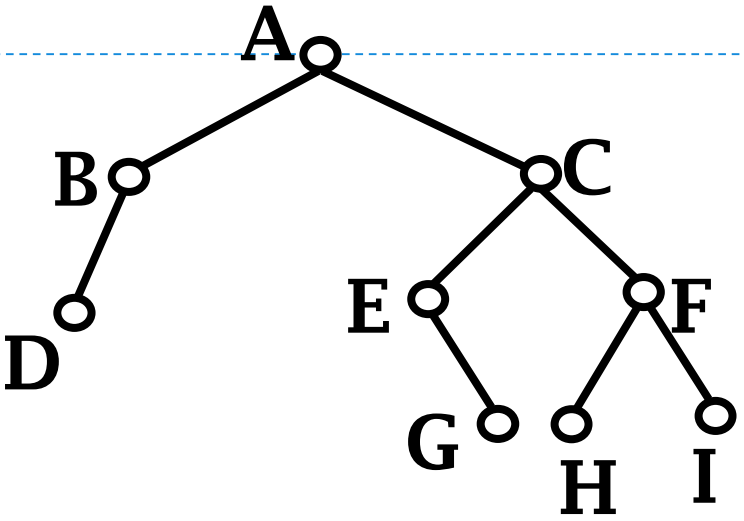
广度优先遍历二叉树分析

- 时间代价
  - 每个结点都被访问且只被访问一次，时间代价为 O(n)
  - 正好每个结点入/出队一次，O(n)
- 空间代价与树的最大宽度相关
  - 最佳 O(1)
  - 最坏 O(n)

思　考

- 试比较宽搜与非递归前序遍历算法框架

第五章 二叉树



- 二叉树的概念
- 二叉树的抽象数据类型
  - 深度优先搜索
  - 宽度优先搜索
- 二叉树的存储结构
  - 用数组实现完全二叉树（顺序存储）
  - 用指针实现二叉树 （链式存储）
- 二叉搜索树
- 堆与优先队列
- Huffman树及其应用

5.4 二叉树的存储

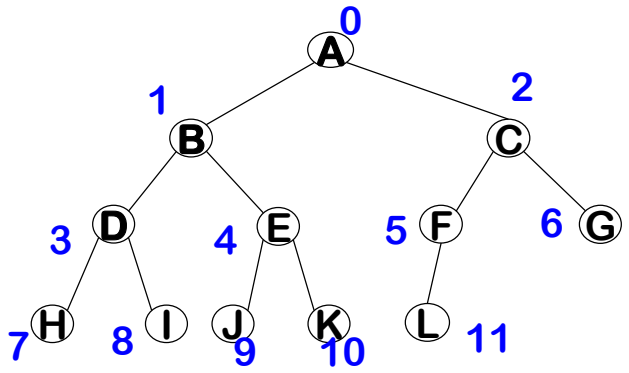
二叉树的顺序存储

- 当要求一个二叉树按 **紧凑结构** 存储，且在应用过程中二叉树的 **大小** 和 **形状 很少**发生**剧烈变化**时，可采用**顺序存储**方法
  - 所有结点按一定的次序顺序存储到**一片事先申请的连续**存储单元中
  - 适当安排结点的线性序列，使结点在序列中的**相互位置**反映出二叉树**结构的部分信息**
  - 一般二叉树，仅有这样的信息**不足以**刻画整个结构，需在结点中附加一些其他的必要信息才能完全反映整个结构

5.4 二叉树的存储

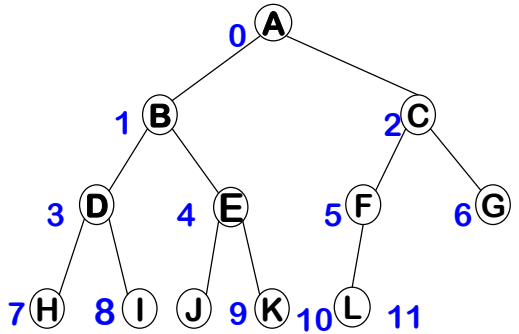
用数组实现完全二叉树

- 有 **n** 个结点的完全二叉树的结点从 **0** 到 **n-1** 编号，得到结点按层次顺序将一棵具的一个线性序列
- 从结点编号 i 可推知其父结点、子结点、兄弟结点的编号
  - 当 2i+1<n 时，其左子结点编号为 2i+1，否则没有左子
  - 当 2i+2<n 时，其右子结点编号为 2i+2，否则没有右子
  - 当 0<i<n 时， 其父结点编号⌊(i-1)/2⌋
  - 当 i 为偶数且 0<i<n 时，其左兄结点编号为i-1，否则没有左兄
  - 当 i 为奇数且 i+1<n 时，其右弟结点编号为 i+1，否则没有右弟



5.4 二叉树的存储

完全二叉树顺序存储示例



位置	0	1	2	3	4	5	6	7	8	9	10	11
父结点	-	0	0	1	1	2	2	3	3	4	4	5
左子结点	1	3	5	7	9	11	-	-	-	-	-	-
右子结点	2	4	6	8	10	-	-	-	-	-	-	-
左兄结点	-	-	1	-	3	-	5	-	7	-	9	-
右弟结点	-	2	-	4	-	6	-	8	-	10	-	-

5.4 二叉树的存储

完全二叉树顺序存储

- 完全二叉树的层次序列**足以反映**其结构
  - 按**层次序列顺序**存储，根据结点的存储地址可计算其左、右子结点、父结点、兄弟结点的存储地址
  - 简单、节省空间的存储方式
- 完全二叉树的顺序存储，**存储结构**上是**线性的**，但依然完整反映了二叉树的**逻辑结构**

二叉树的链式存储结构

二叉树的各结点随机地存储在内存空间中，结点之间的逻辑关系用指针来链接。

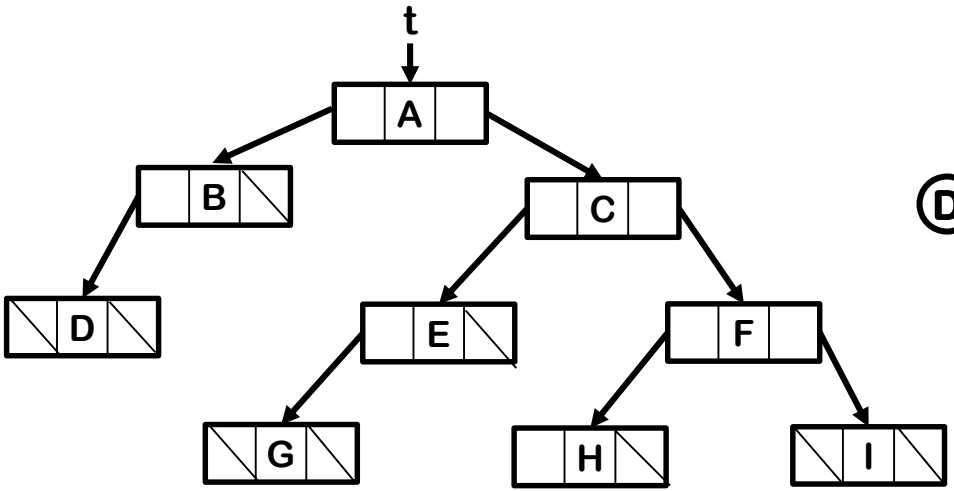
- 二叉链表
  - 指针 left 和 right，分别指向结点的左孩子和右孩子

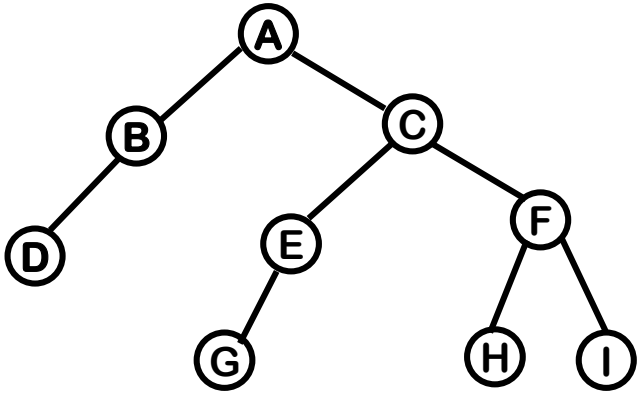
- 三叉链表
  - 指针 left 和 right，分别指向结点的左孩子和右孩子
  - 增加一个父指针

5.4 二叉树的存储

二叉树的链式实现

- 特点
  - n 个结点的二叉链表含有 n+1 个空指针





5.4 二叉树的存储

BinaryTreeNode类中增加两个私有数据成员

```
private:
    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> 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
    ....
}
```

5.4 二叉树的存储

递归框架寻找父结点——注意返回

```
template<class T>
BinaryTreeNode<T>* BinaryTree<T>::
Parent(BinaryTreeNode<T> *rt,  BinaryTreeNode<T> *current)  {
    BinaryTreeNode<T> *tmp;
    if (rt == NULL) return(NULL);
    if (current == rt ->leftchild() || current == rt->rightchild())
        return rt;                          // 如果子结点是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;
}
```

5.4 二叉树的存储

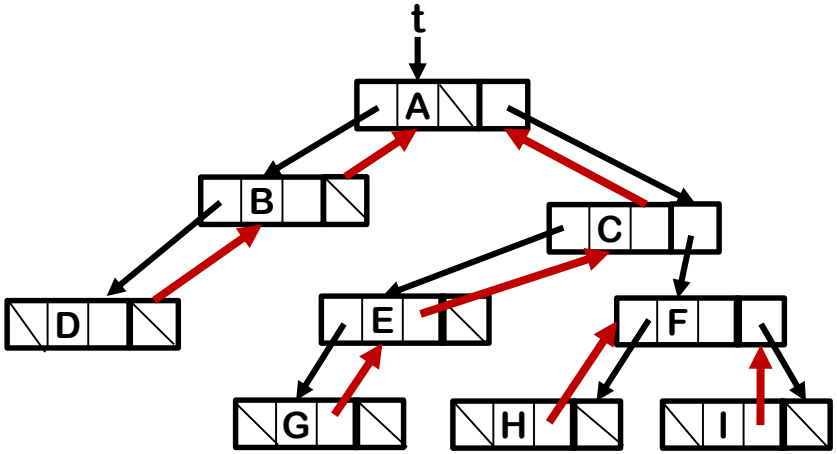
非递归框架找父结点

```
BinaryTreeNode<T>* BinaryTree<T>::Parent(BinaryTreeNode<T> *current) {
    using std::stack;                                          // 使用STL中的栈
    stack<BinaryTreeNode<T>* > aStack;
    BinaryTreeNode<T> *pointer = root;
    aStack.push(NULL);                                         // 栈底监视哨
    while (pointer) {                                          // 或者!aStack.empty()
        if (current == pointer->leftchild() || current == pointer->rightchild())
            return pointer;                                    // 若current为pointer子结点则返回parent
        if (pointer->rightchild() != NULL)                     // 非空右子结点入栈
            aStack.push(pointer->rightchild());
        if (pointer->leftchild() != NULL)
            pointer = pointer->leftchild();                    // 左路下降
        else                                                   // 访问完左子转右子
            pointer = aStack.top(); aStack.pop();              // 取栈顶并退栈
    }
}
```

5.4 二叉树的存储

二叉树的链式实现

- 链接的其他表示方式：**三叉链表**
  - 增加一个指向其父结点的指针域
  - 提供“向上”访问的能力，在某些经常要回溯到父结点的应用中很有效



left	info	parent	right

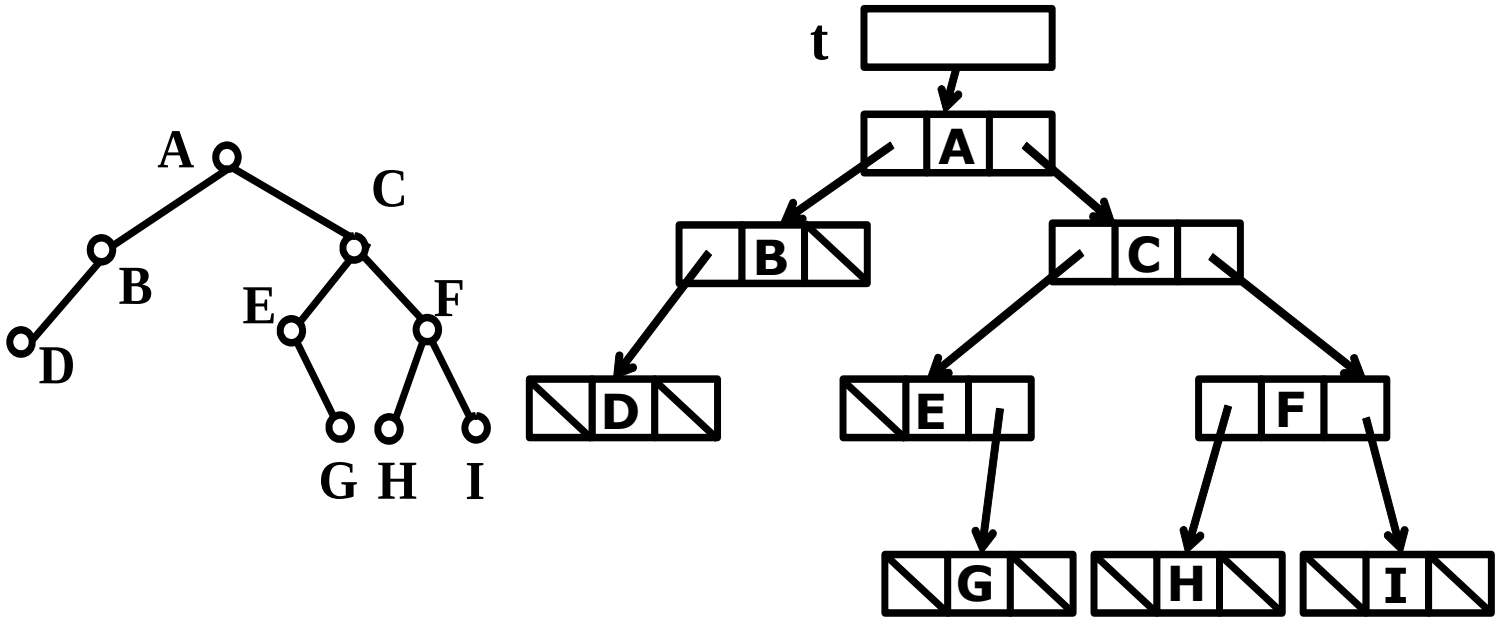
5.4 二叉树的存储

链式存储特点

- 优点
  - 运算方便，通过指针可直接访问相关结点

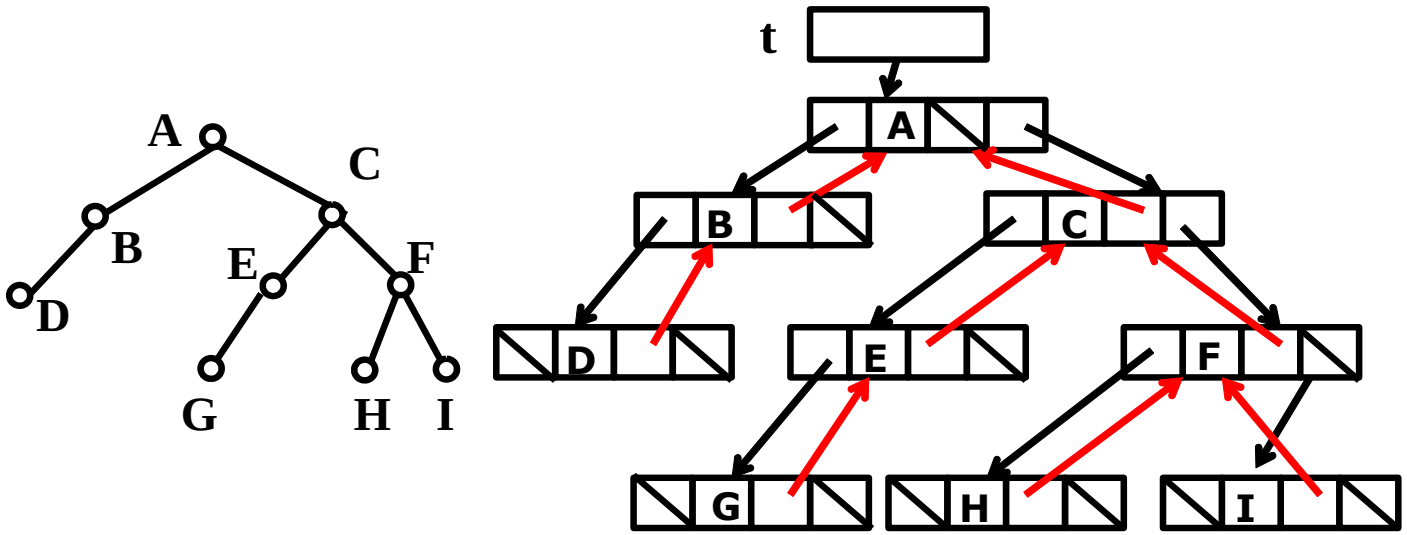
- 问题
  - 根据满二叉树定理：一半的指针是空的
    - 空指针 n+1，存储密度低
    - 结构性开销较大

二叉链表



“三叉链表”

- 指向父母的指针parent, “向上”能力



5.4 二叉树的存储

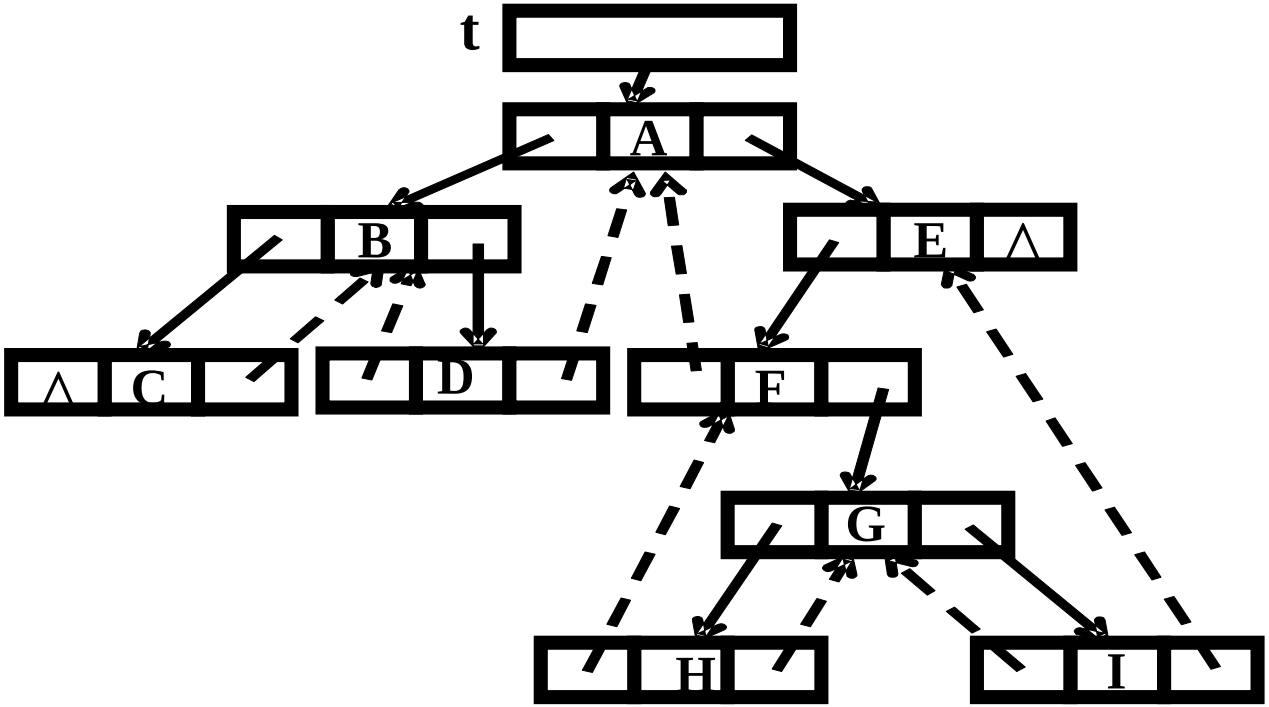
二叉树的存储总结

- 采用不同的存储实现，不仅空间开销有差异，其上的运算和操作的实现也不同

  - 具体应用中采取何种存储结构，除根据二叉树的形态外，还应考虑运算

    - 时间复杂度
    - 空间复杂度
    - 算法简洁性

穿线二叉树



中序线索化二叉树：递归实现

```
template <class T> void
ThreadBinaryTree<T>::InThread
            (ThreadBinaryTreeNode<T>*root,
             ThreadBinaryTreeNode<T>* &pre) {
    if (root!=NULL) {
            InThread(root->leftchild(), pre);                    //中序线索化左子树
            if (root->leftchild() == NULL) {
                root->left=pre;                                  //建立前驱线索
                 if (pre != NULL) root->lTag=1; }
            if ((pre!=NULL) && (pre->rightchild()==NULL)) {  //建立后继线索
                pre->right=root;  pre->rTag=1; }//end if
            pre=root;
            InThread(root->rightchild(),pre);                    //中序线索化右子树
    }//end if
}
```

left	lTag	info	rTag	right

空间开销分析

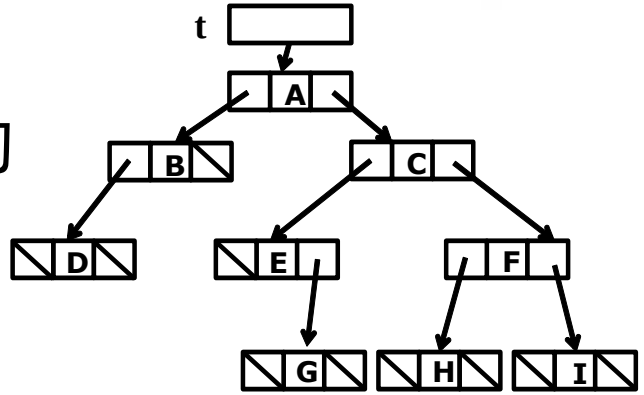
- 存储密度 α (≤1) 表示数据结构存储的效率

$$\alpha(\text{存储密度}) = \frac{\text{数据本身存储量}}{\text{整个结构占用的存储总量}}$$

- 结构性开销 γ，α + γ = 1
  - 实现数据结构需占用的辅助空间
  - 这些辅助空间并非用来存储数据，而是保存数据的逻辑特性或方便运算
  - 存储空间全部用于数据本身的存储结构 称为**紧凑结构**，否则**非紧凑结构**

5.4 二叉树的存储

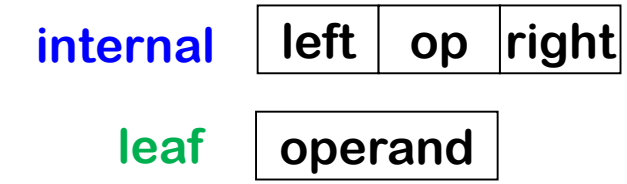
空间代价分析



- 根据满二叉树定理：一半的指针是空的
- 若每个结点占用两个指针、一个数据域
  - 总空间 (2p + d)n
  - 结构性开销：2pn
    - 若 $p = d$，则 2p/(2p + d) = 2/3
- 若只有叶结点存储数据，分支结点为内部结构，则开销取决于二叉树满的程度
  - 越满存储效率越高

空间开销分析

- C++ 可以用两种方法来实现不同的分支与叶结点：
  - union联合类型定义
  - C++的子类分别实现分支结点与叶结点
    并用虚函数isLeaf来区别分支与叶结点



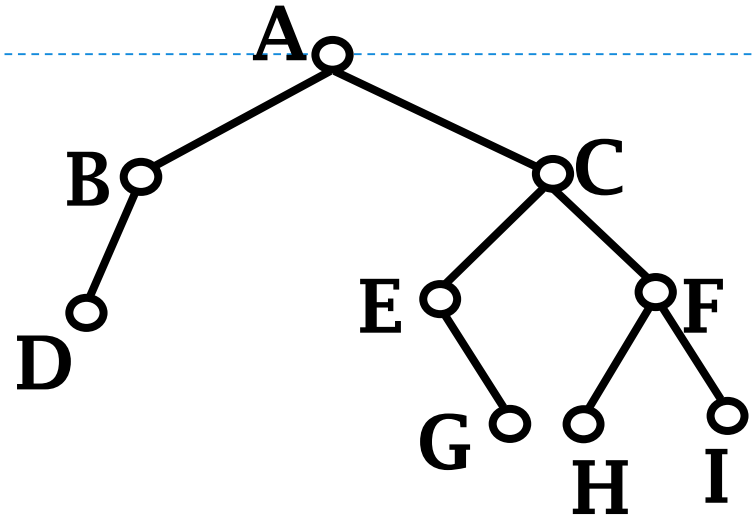
- 早期节省内存资源
  - 利用结点指针的一个空闲位（一个bit）来标记结点所属的类型
  - 利用指向叶的指针或者叶中的指针域来存储该叶结点的值

5.4 二叉树的存储

思考

- 非完全的二叉树可以采用顺序存储吗？若可以，如何存储？
- 完全二叉树的编号规则可推广至完全三叉树，试给出完全三叉树的下标公式
- 从前面给出的寻找给定结点的父结点算法出发，怎样寻找给定结点的兄弟结点？

第五章 二叉树



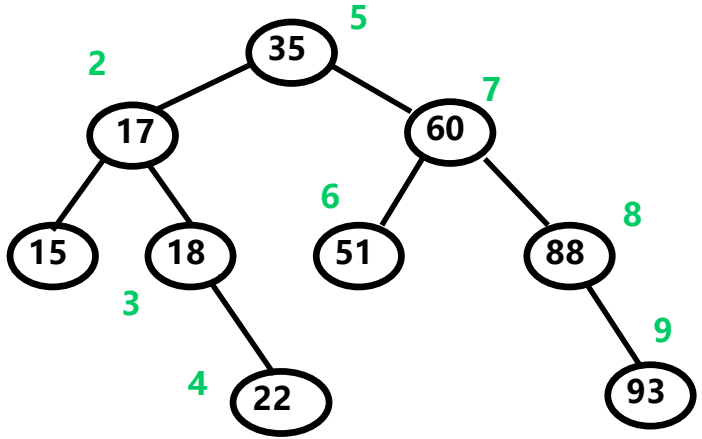
- 二叉树的概念
- 二叉树的抽象数据类型
  - 深度优先搜索
  - 宽度优先搜索
- 二叉树的存储结构
- 二叉搜索树
- 堆与优先队列
- Huffman树及其应用

5.5 二叉搜索树

二叉搜索树

- 二叉树的一个**主要用途**是提供对数据（包括索引）的**快速检索**，而一般二叉树对此并不具有性能优势

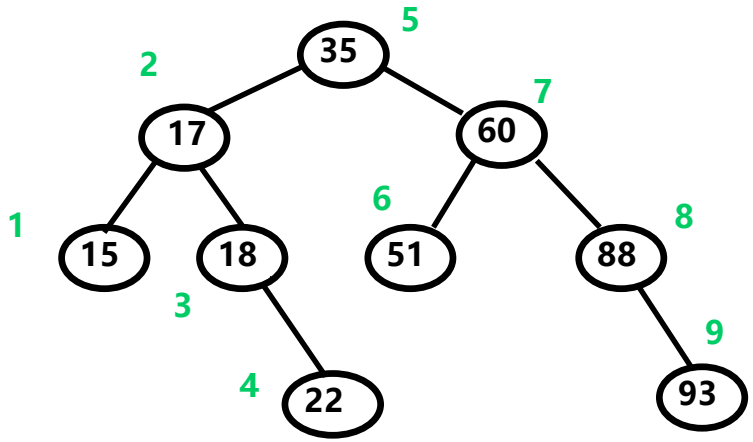
- **常用名称（同义词）**
  - 二叉搜索树（Binary Search Tree，简称BST）
  - 二叉查找树
  - 二叉检索树
  - 二叉排序树



5.5 二叉搜索树

二叉搜索树：定义

- 按结点的**检索码K** 组织，具有如下性质：
  - ✓ 或为 **一棵空树**；
  - ✓ 或 任何一个**检索码为 K** 的结点满足
    1. 其左子树（若非空）任一结点的检索码均**小于 K**；
    2. 其右子树（若非空）任一结点的检索码均**大于或等于 K**；
    3. 其左、右子树分别为二叉搜索树



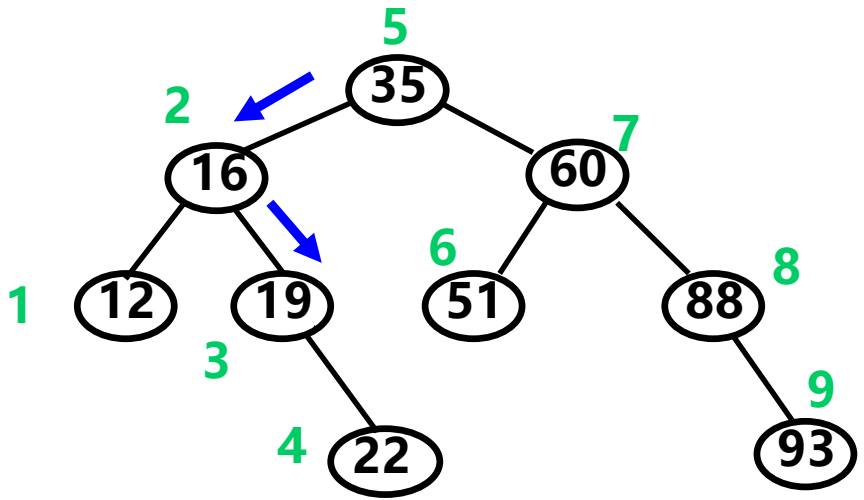
特点: 中序遍历是正序的（由小到大的排列）

5.5 二叉搜索树

BST的检索

- **步骤**
  - 从根结点开始，在二叉搜索树中查找检索码为K 的结点。若根结点检索码为 K，则检索成功并结束
  - 若 K 小于根结点的值，则只需检索左子树
  - 若 K 大于根结点的值，只检索右子树

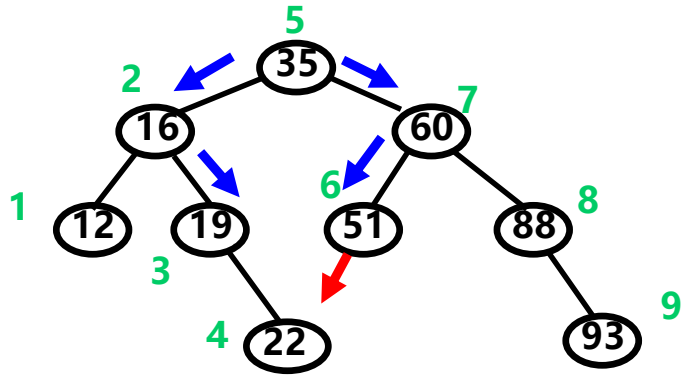
  如此， 一直持续到 K 被**找到** 或
  - 遇上叶结点仍没发现 K ， 则检索失败
    - 说明不存在满足条件的结点



5.5 二叉搜索树

BST检索示例

- 查找检索码为 35 的结点
- 查找检索码为 19 的结点
- 查找检索码为 40 的结点



比较次数?

时间复杂度?

5.5 二叉搜索树

BST的检索

- 检索代价
  - 基本操作： 比较
  - 只需检索两个子树之一：每比较1 次，树的高度降 1
    - **O(h)**　树高 **h**

5.5 二叉搜索树

BST的插入

- 保证结点插入后仍保持BST性质，

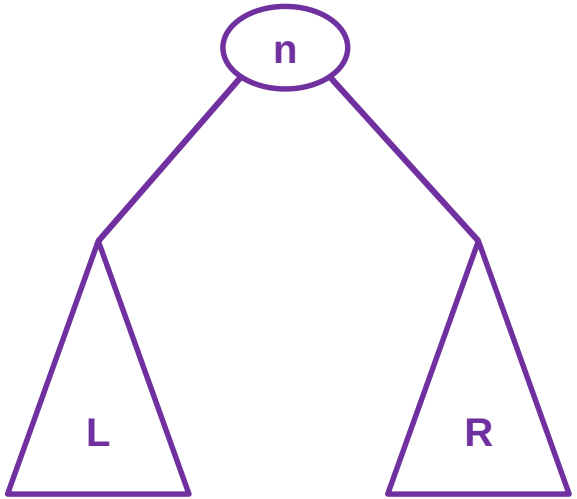
  即，**BST的分割不变量：中序有序性**

  L.key < n.key

  R.key > n.key

- 待插入结点的**检索码为 K**

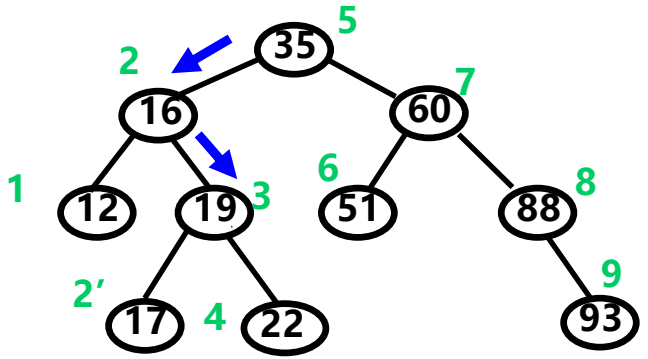
  1. 从根结点开始，若根结点为空，则将 K 结点作为根插入，操作结束
  2. 若 K 小于根结点的值，将其插入左子树
  3. 若 K 大于根结点的值，将其插入右子树



5.5 二叉搜索树

BST插入示例

- 首先是检索，若找到则不允许插入
- 若失败，则在该位置插入一个新叶
- 保持BST性质和性能！



5.5 二叉搜索树

BST的插入分析

- 一次失败的检索
- 新结点作为叶结点插入，改动相关特定结点的空指针即可
- 时间复杂度与 根到插入位置的 **路径长度** 相关

5.5 二叉搜索树

BST上的删除

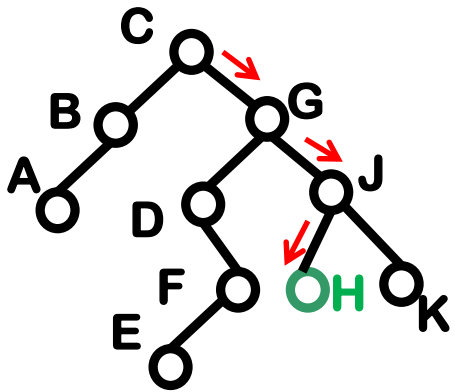
- 删除一个结点后仍能保持BST性质，且树高变化较小
- 首先检索到待删除结点，根据其在树中所处位置分情况处理

5.5 二叉搜索树

BST上的删除 #1

- **情况1．叶结点**
  - ✓ 直接删除，其父结点的相应指针置为空

删除 H

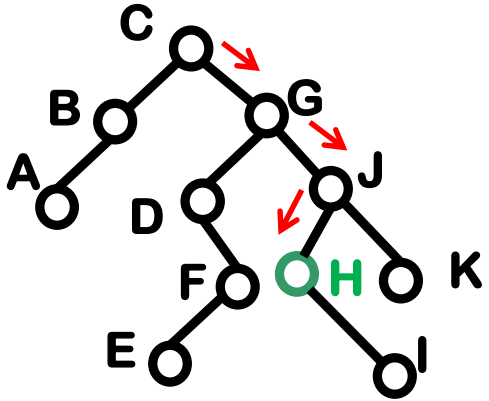


5.5 二叉搜索树

BST上的删除 #2

- **情况2.　只有一个子结点的结点**
  - ✓ 直接用该子结点代替

删除 H



5.5 二叉搜索树

BST上的删除 #3

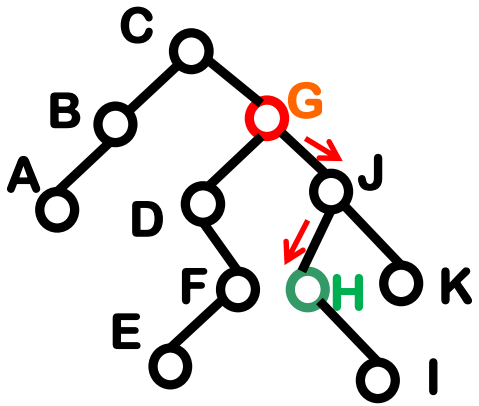
- **情况3. 被删结点 p 的左右子结点皆不空**

  - 根据BST性质，寻找可替换 **p** 的结点：比 p 的左子树中所有结点大，比p的右子树中所有结点小（或不大于）
    - 左子树中**最大者**
    - 右子树中**最小者**
  - 二者都至多只有一个子结点 （归结为情况#1 和情况#2）
    - **Why?**

5.5 二叉搜索树

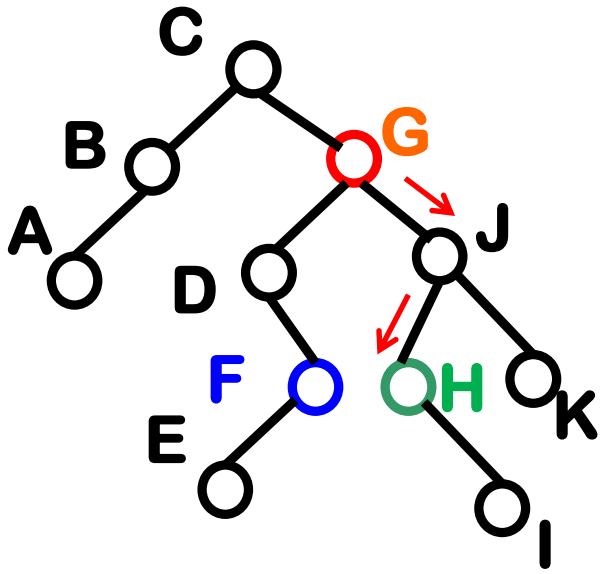
删除示例 #3

删除 G



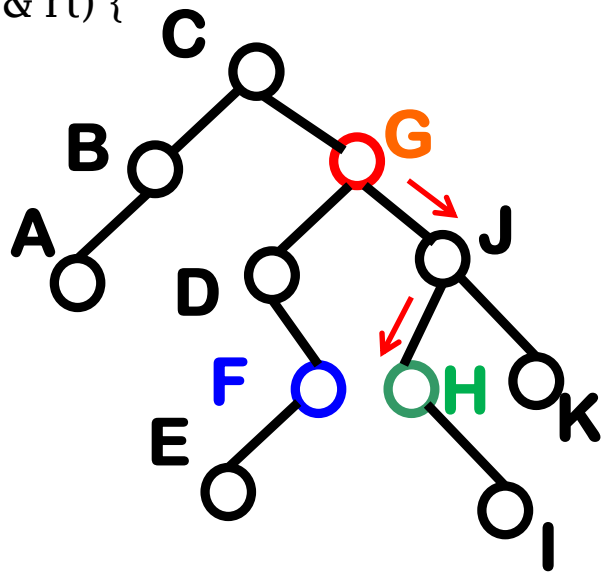
递归找到值的位置，准备替换

```
void BinarySearchTree<T>:::removehelp(BinaryTreeNode <T> *& rt, const T val) {
  if (rt==NULL) cout<<val<<" is not in the tree.\n";
  else if (val < rt->value())
     removehelp(rt->leftchild(), val);
  else if (val > rt->value())
     removehelp(rt->rightchild(), val);
  else {                                  // 真正的删除
     BinaryTreeNode <T> * temp = rt;
     if (rt->leftchild() == NULL)  rt = rt->rightchild();
     else if (rt->rightchild() == NULL)  rt = rt->leftchild();
     else {
        temp = deletemin(rt->rightchild());
        rt->setValue(temp->value());
     }
     delete temp;
  }
}
```



找rt右子树中最小结点，并删除

```
template <class T>
BinaryTreeNode* BST::deletemin(BinaryTreeNode <T> *& rt) {
  if (rt->leftchild() != NULL)
        return deletemin(rt->leftchild());
  else {    // 找到右子树中最小，删除
      BinaryTreeNode <T> *temp = rt;
      rt = rt->rightchild();
      return temp;
  }
}
```



5.5 二叉搜索树

BST的删除算法

- 树高h，时间代价O(h)
  - 查找待删除结点
  - 查找替代结点

5.5 二叉搜索树

有关BST的讨论

- 所有操作都围绕BST的性质，并保持这个性质
- 操作的效率取决于树的高度，树形平衡时二叉搜索树的效率相当高
  - 多次插入和删除之后，BST的形状变化，失去平衡
    - 结合AVL 树，红黑树等平衡树技术

二叉搜索树总结

- 组织内存索引
  - 二叉搜索树是适用于内存储器的一种重要的树形索引
    - 常用红黑树、伸展树等，以维持平衡
  - 外存常用B/B+树
- 保持性质 vs 保持性能
  - 插入新结点或删除已有结点，要保证操作结束后仍符合二叉搜索树的定义

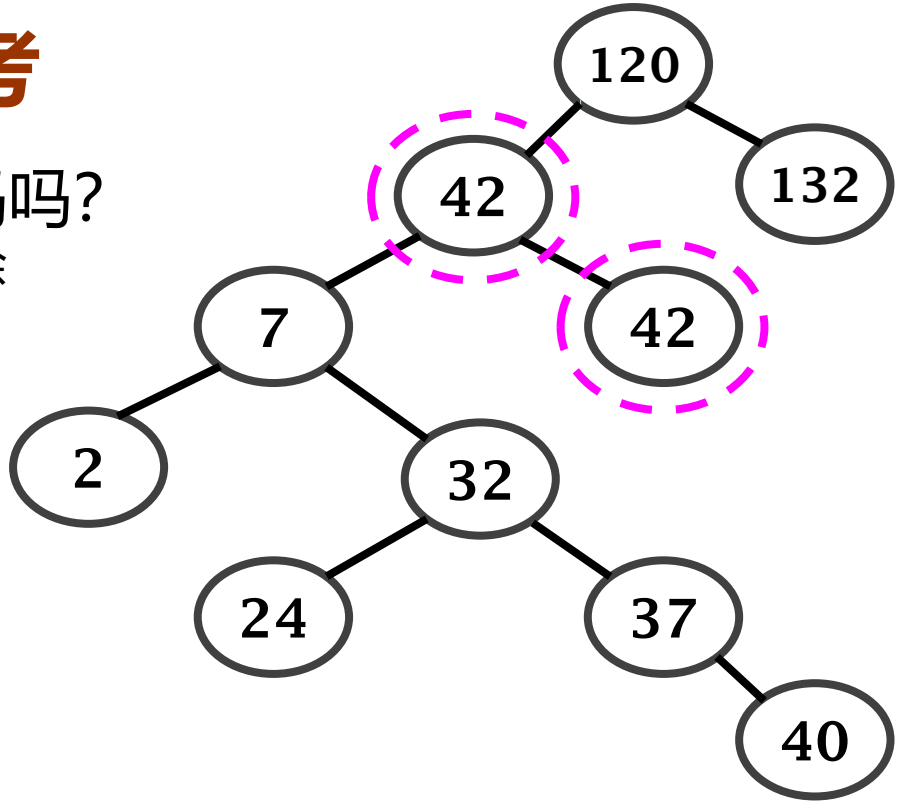
5.5 二叉搜索树

BST总结

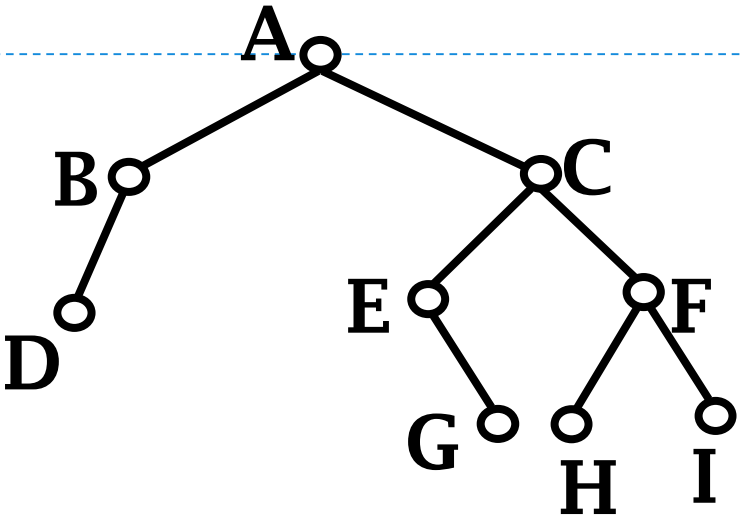
- 树结构的**应用之一**是**组织索引**
  - **BST** 适于组织**内存索引**
    - 常结合红黑树、AVL、伸展树等平衡技术
  - 外存常用B/B+树
- 保持性质 vs 保持性能
  - 往BST插入新结点或删除已有结点时须保证BST性质
  - BST的插入和删除**代价与树高成正比**

思 考

- 允许重复关键码吗?
  - 插入、检索、删除



第五章 二叉树



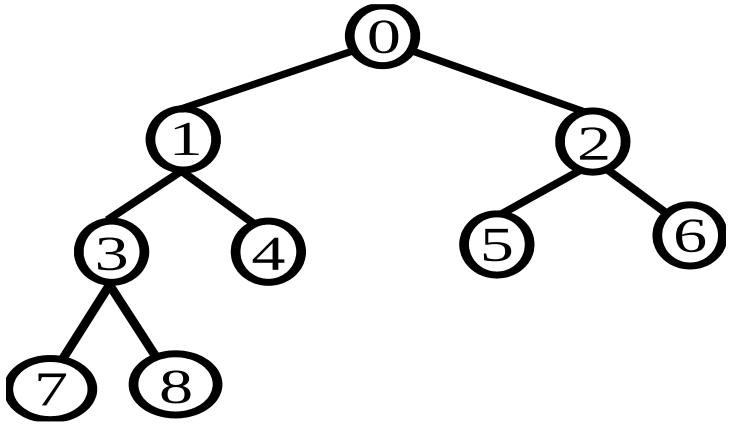
- 二叉树的概念
- 二叉树的抽象数据类型
  - 深度优先搜索
  - 宽度优先搜索
- 二叉树的存储结构
- 二叉搜索树
- 堆与优先队列
- Huffman树及其应用

完全二叉树的顺序存储结构

- 顺序方法存储二叉树
  - 把结点按一定的顺序存储到一片连续的存储单元
  - 使结点在序列中的位置反映出相应的结构信息
- 存储结构上是线性的

0	1	2	3	4	5	6	7	8

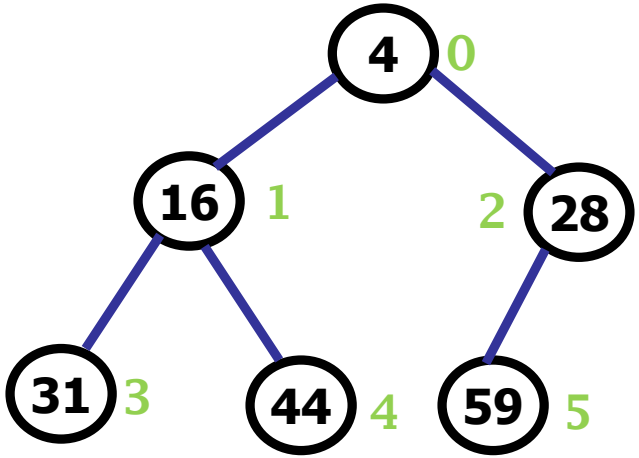
- 逻辑结构上它仍然是二叉树形结构



5.6 堆与优先队列

堆：满足某种特性的完全二叉树

- **最小值堆（min-heap）**
  - **完全二叉树**
  - 任一结点的值**小于或等于**其子结点的值
  - 根结点存储着树中所有结点的**最小值**



5.6 堆与优先队列

堆的定义 （heap）

- 一个关键码序列{ $K_0, K_1, \ldots, K_{n-1}$} 若具有下述特性：

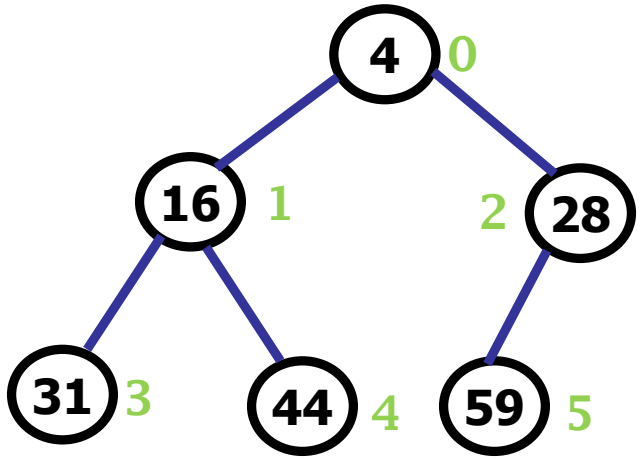
$$K_i \le K_{2i+1} \quad / \quad K_i \ge K_{2i+1},$$

$$K_i \le K_{2i+2} \quad / \quad K_i \ge K_{2i+2},$$

$$(i = 0, 1, \ldots, \lfloor n/2 \rfloor)$$

则称其为**堆**。 即，

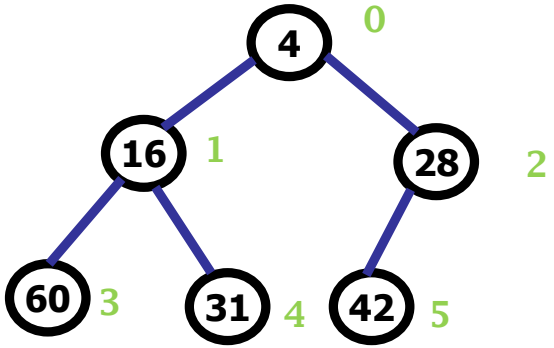
- 最小值**堆** / 最大值**堆**



5.6 堆与优先队列

堆的性质

- 从逻辑角度看，堆是一种树型结构
  - 完全二叉树的层次序列，可采用顺序数组表示
- 堆 **局部有序** ，堆**不唯一**
  - 结点的值与其子结点的值之间存在某种约束
  - 堆中任一结点与其兄弟之间没有约束
  - 最小堆 并非BST 那样实现关键码的完全排序，而是局部有序，只有父子结点的大小关系可以确定



4	16	28	60	31	42

5.6 堆与优先队列

如何建堆?

- 一棵完全二叉树如何调整成堆
  - 假设根的左、右子树都已是堆，且根为T，则有 两种可能:
    1. T 的值小于或等于其两个子结点的值，此时为堆，无须调整;
    2. T 的值大于两个子结点至少一个的值，此时 T 应与两个子结点中较小者交换，若 T 仍大于其新的子结点，需 将 T 继续往下“筛”（*siftdown*），直至到某一个层使 T 不再大于其子结点，或已为叶结点

筛选法 （1964年Floyd提出）

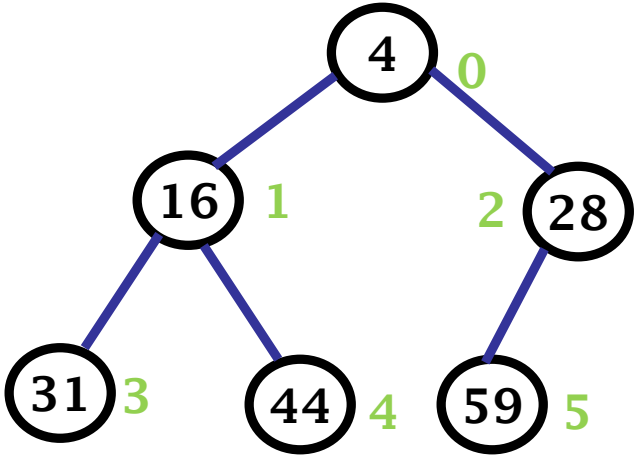
5.6 堆与优先队列

筛选法建堆

- 将 n 个关键码组织到一维数组中
  - 可能整体并不满足堆的特性
  - 以叶结点为根的子树 都已满足堆的特性，亦即，当 $i \geq \lfloor n/2 \rfloor$ 时，以关键码 K_i 为根的子树已为堆
  - 从 最后一个分支结点 $i = \lfloor n/2 \rfloor - 1$ 开始，采用 siftdown 从右向左、自底向上 将 以各个分支结点为根的子树 **调整** 成堆，直到树根为止

建最小堆过程

- 首先，将 n 个关键码放到一维数组中
  - 整体不是最小堆
  - 所有叶结点子树本身是堆
    - 当 $i \geq \lfloor n/2 \rfloor$ 时，
      以关键码 K_i 为根的子树已经是堆
- 从倒数第二层，$i = \lfloor n/2 \rfloor - 1$ 开始
  从右至左依次调整
- 直到整个过程到达树根
  - 整棵完全二叉树就成为一个堆

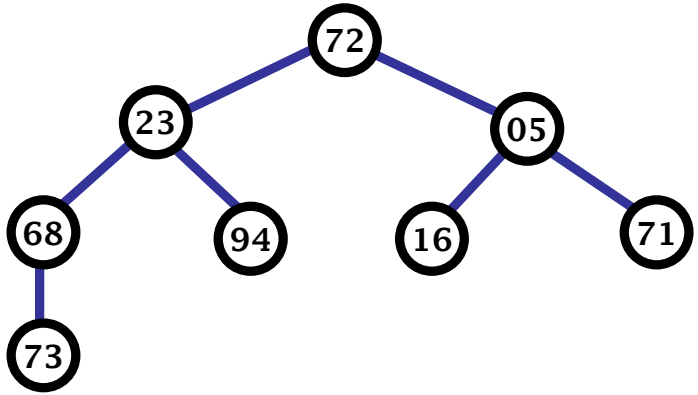


堆的类定义

```
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
   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);                 // 删除给定下标的元素
   bool Insert(const T& newNode);                 // 向堆中插入新元素newNode
   T& RemoveMin();                                // 从堆顶删除最小值
   void SiftUp(int position);                     // 从position向上开始调整，使序列成为堆
   void SiftDown(int left);                       // 筛选法函数，参数left表示开始处理的数组下标
}
```

对最小堆用筛选法 SiftDown 调整

```
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
```



对最小堆用筛选法 SiftDown 调整

```
    while (j < CurrentSize) {
        if ((j < CurrentSize-1)  &&  (heapArray[j] > heapArray[j+1]))
            j++;                          // j 指向数值较小的子结点
        if (temp > heapArray[j]) {// 判断结点与其子结点关键码大小
            heapArray[i] = heapArray[j];
            i = j;    j = 2*j + 1;        // 向下继续
        }
        else break;                       // 调整结束
    }
    heapArray[i] = temp;
}
```

对最小堆用筛选法 SiftUp 向上调整

```
template<class T>
void MinHeap<T>::SiftUp(int position) {
    // 从position向上开始调整，使序列成为堆
    int temppos=position;
    // 不是父子结点直接swap
    T temp=heapArray[temppos];
    while((temppos>0) && (heapArray[parent(temppos)] > temp))  {
            heapArray[temppos]=heapArray[parent(temppos)];
              temppos=parent(temppos);
    }
    heapArray[temppos]=temp;// 找到最终位置
}
```

建最小堆

从第一个分支结点 heapArray[CurrentSize/2-1]
开始，自底向上逐步把以子树调整成堆

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

5.6 堆与优先队列

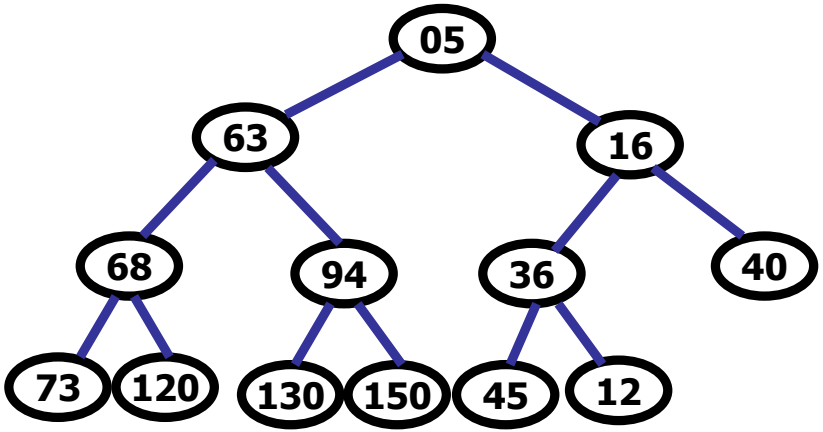
建堆过程小结

1. 先将所有元素组织成一维数组，此时所形成的完全二叉树可能尚不具备最小堆的特性，只有叶结点满足堆的性质
2. 从最后一个分支结点（在完全二叉树倒数第二层）开始，从右至左依次通过筛选法调整
3. 一层调整完后，继续调整上一层，直到调整完树根，整棵完全二叉树就成为一个堆

5.6 堆与优先队列

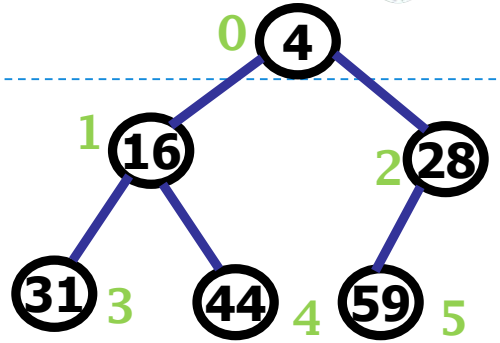
堆中插入新元素

- 插入元素12



- 插在堆的最后 (why?)
- 与其父结点比较，若不满足堆的性质则往上“拉” (SiftUp)
- 逐步向上，直到与其父结点满足堆的性质

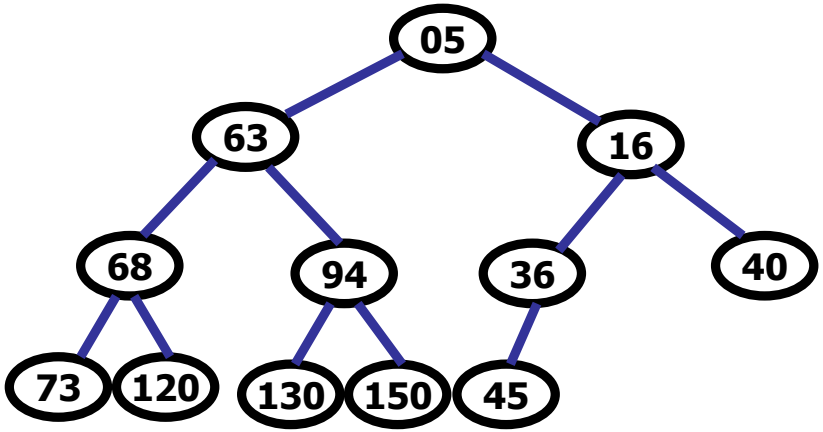
最小堆插入新元素




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

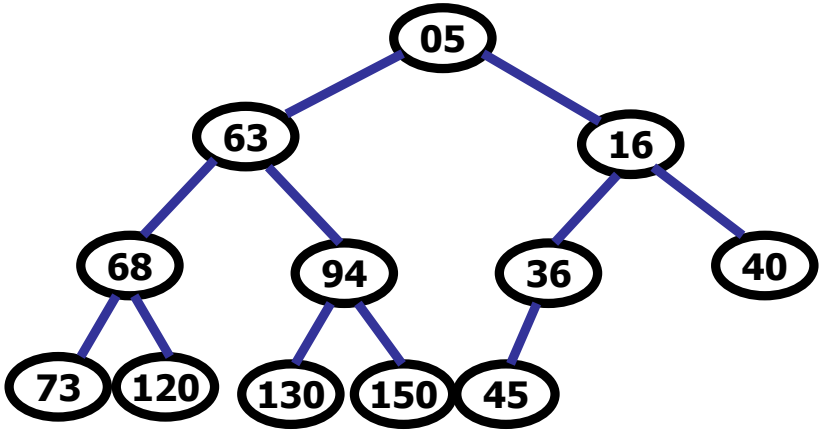
5.6 堆与优先队列

根结点删除示意



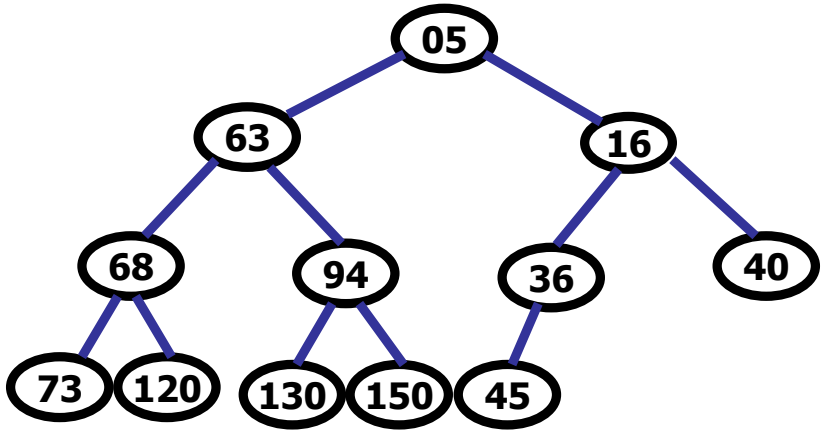
删除堆中任一元素

- 如何删除根以外的元素？是否可以沿用删除根结点的方法？
- e.g., 删除 68



删除堆中任一元素

- 如何删除根以外的元素？是否可以沿用删除根结点的方法？
- e.g., 删除16



最小堆删除元素操作

```
template<class T>
bool MinHeap<T>::Remove(int pos, T& node) {
    if((pos<0)||(pos>=CurrentSize))
        return false;
    T temp=heapArray[pos];
    heapArray[pos]=heapArray[--CurrentSize];
    if (heapArray[parent(pos)]> heapArray[pos])
        SiftUp(pos);          //上升筛
    else SiftDown(pos);   // 向下筛
    node=temp;
    return true;
}
```

5.6 堆与优先队列

删除根结点

- 堆最常用的操作 （删除后须维护和保持堆的特性）

```
template <class T>  T MinHeap<T>:: RemoveRoot() {
    if (CurrentSize == 0)  exit(1);

    Item tmpItem = heapArray[0];
    heapArray[0] = heapArray[CurrentSize - 1];
    heapArray[CurrentSize - 1] = tmpItem;

    CurrentSize --;
    SiftDown(0);
    return tmpItem;
}
```

SiftDown的时间代价

- 基本操作
  - 比较：2 次，判断子结点的大小，及结点是否需要筛选
  - 交换：1 次，最差情况每层都需调整 （不满足堆的特性）
- 一个 n 个结点的完全二叉树高度⌈log(n+1)⌉
  - 每循环一次把目标结点下移一层，故循环最多为
    ⌈ log(n+1) ⌉次

∴ **最差情况下** SiftDown 的时间代价为 **O(log n)**

5.6 堆与优先队列

建堆的时间代价

- 一个 n 个结点的完全二叉树，若同时为满二叉树，则筛选的层数达到最大，此时有 $\mathbf{n = 2^d - 1}$，$\mathbf{d = \lceil \log (n+1) \rceil}$

  - 第 k 层至多2^k个结点，且离叶结点的距离为 d-k-1 层

- 构建具有 n 个结点的堆需要的比较次数为：

$$2\sum_{k=0}^{d-1} 2^k(d-k-1) = 2\left[(d-1)\sum_{k=0}^{d-1} 2^k - 2\sum_{k=0}^{d-1} k2^{k-1}\right]$$

$$= 2\left[2^d - d - 1\right] = 2\left[n - \lceil \log n \rceil\right]$$

∴ 构建具有 n 个结点的堆的时间复杂度为 O(n)

5.6 堆与优先队列

堆运算分析

- 建堆算法的时间复杂度是 O(n)
  - 线性时间内把一个无序的序列转化成堆序
- 堆的深度为log n
  - 插入、删除元素 的平均时间代价和最差时间代价都是O(log n)
- 适合于既经常查找最小值，又经常增删数据的场景
- 查找任意值的效率不高

5.6 堆与优先队列

堆小结

- 二叉树所表示的二叉堆是常用的一种堆；由于完全二叉树良好的性质，常采用数组来存储堆
- 堆是基于树的满足一定约束的重要数据结构，存在许多变体：二项式堆、斐波那契堆
- 堆的基本操作均依赖于两个重要的函数siftUp和siftDown
  - 堆的**插入操作**在堆尾插入新元素并通过**siftUp调整堆**
  - 堆的**删除操作**根据被删元素的位置和大小 进行**siftDown 或 siftUp二者之一**

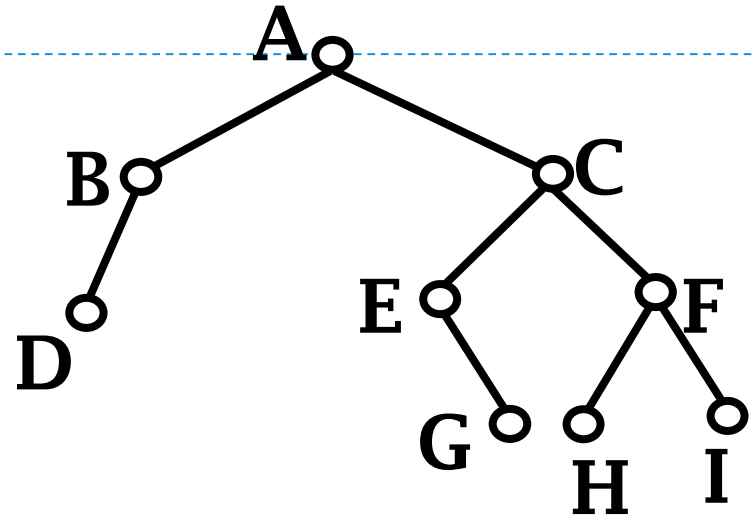
堆的应用

- 堆排序
- 优先队列(Priority Queue)
  - 根据需要释放具有最小 / 大值的对象
  - 最大树、左高树 (HBLT、WBLT、MaxWBLT)
  - 改变已存储于优先队列中对象的优先权
    - 辅助数据结构帮助找到对象

思考

- SiftDown操作时，一旦发现逆序对就交换会怎么样?
- 能否在一个数据结构中同时维护最大值和最小值? （提示：最大最小堆）

第五章 二叉树



- 二叉树的概念
- 二叉树的抽象数据类型
  - 深度优先搜索
  - 宽度优先搜索
- 二叉树的存储结构
- 二叉搜索树
- 堆与优先队列
- Huffman树及其应用

编码

- **程序设计**、**数据通信**等领域常需对某些给定字符集进行编码：即，采用一组无歧义的规则将字符集中每个字符编码为唯一可标识的代码串
  - 定长编码 (fixed-length coding scheme)
  - 变长编码 (variable-length coding scheme)

5.7 Huffman树及其应用

定长编码

- 所有字符的编码长度均相同，编码一个具有 n 个字符的字符集最少需要$\log_2 n$ 位
  - ASCII码就是一种定长编码（8位）
  - 中文编码（双字节）
- 若字符集中每个字符使用频率大致相同，定长编码的空间效率最高
- 具有**简单、解码容易**的优点

5.7 Huffman树及其应用

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

- 不等长编码
  - ✓ 根据字符出现频率编码，常出现字符的编码较短，使用频率低的字符编码较长
  - ✓ 不等长编码是文件压缩技术的核心
    - 数据压缩既能节省磁盘空间，又能提高传输速度

（**外存时空权衡的规则**）

5.7 Huffman树及其应用

不等长编码

- 不等长编码关键：任何一个字符的编码 都不能是 另外一个字符编码的前缀
  - ✓ 例如，对于字符集{Z, K, F, C, U, D, L, E}，若编码为

    Z(0), K(1), F(00), C(01), U(10), D(11), L(000), E(001)，
  - ✓ 解码歧义　串“000110”可有多种解释

    “ZZZDZ”，或

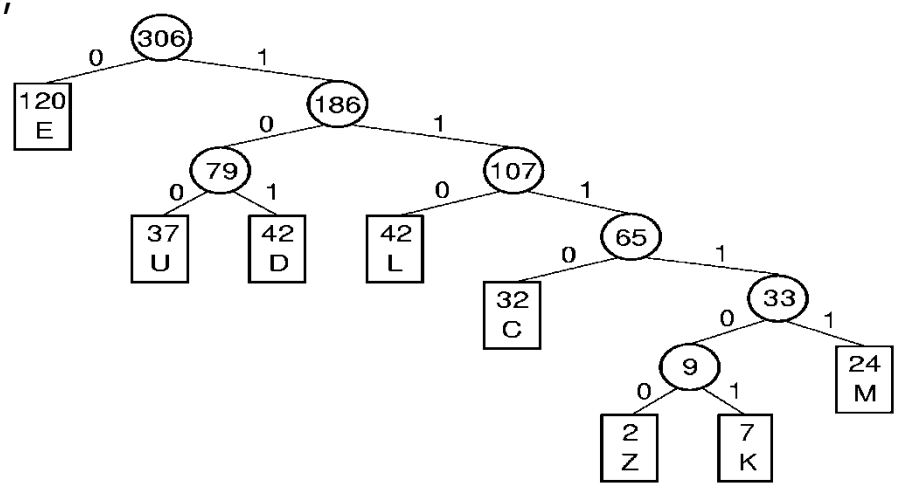
    “LDZ”，或

    “FCU”

5.7 Huffman树及其应用

前缀编码

- 一个字符集中，任何一个字符的编码都不是另外一个字符编码的前缀
- **前缀特性**保证了代码串被解码时，不会出现歧义
  - ✓ 例如，将字符集 {Z, K, F, C, U, D, L, E}的 8 个字符分别编码为
    Z(111100), K(111101), F(11111), C(1110),
    U(100), D(101), L(110), E(0)
    为一种前缀编码
  - ✓ 代码串“000110” 可解码为
    唯一的字符串 “ EEEL”



5.7 Huffman树及其应用

二叉树 vs 前缀编码

- 可用二叉树来设计和表示前缀编码
  - ✓ 约定叶结点代表字符
  - ✓ 一个结点的 左分支 标记 ‘0’ ，右分支标记 ‘1’
  - ✓ **根结点到叶结点的路径上所有分支标记**所组成的**代码串**作为该叶结点所代表字符的**编码**

  这样的编码一定是**前缀编码** (why?)
- 如何保证这样的编码树所得到的编码总长度最小?
  - ✓ Huffman算法解决了这个问题

5.7 Huffman树及其应用

Huffman树

- 待编码的字符集D = {d_0，…，d_{n-1}}，D中各个字符的出现频率为W = {w_0，…，w_{n-1}}，对字符集D进行二进制编码，使得：
  - ✓ 通信编码总长最短
  - ✓ ∀i, j, $d_i \neq d_j$ 时 d_i 的编码不是 d_j 编码的前缀，反之亦然
- Huffman编码的基本思想
  - ✓ 将 d_i 作为 **外部结点**，w_i为外部结点的权，构造具有 最小带权外部路径长度 的扩充二叉树

5.7 Huffman树及其应用

Huffman树

- 一个具有 n 个外部结点的扩充二叉树
  - ✓ 每个外部结点 d_i 有一个与之对应的 权 w_i
  - ✓ 这个扩充二叉树的带权外部路径长度总和

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

最小

- ✓ 权越大的叶结点 离根越近

5.7 Huffman树及其应用

建立Huffman树

步骤：

1. 按照“权”(诸如，频率) 将字符组成有序序列；
2. 取走当前序列的前两个字符（“权”最小），将其标记为Huffman树的叶结点，并将这两个叶结点作为一个（新生成）**分支结点** 的两个子结点，该**分支结点的权**为两叶结点的**权之和**；将所得子树的“权”放回序列中适当位置，保持“权”的有序性；
3. 重复上述步骤直至序列中只剩一个元素，则Huffman树建立完毕

5.7 Huffman树及其应用

Huffman树

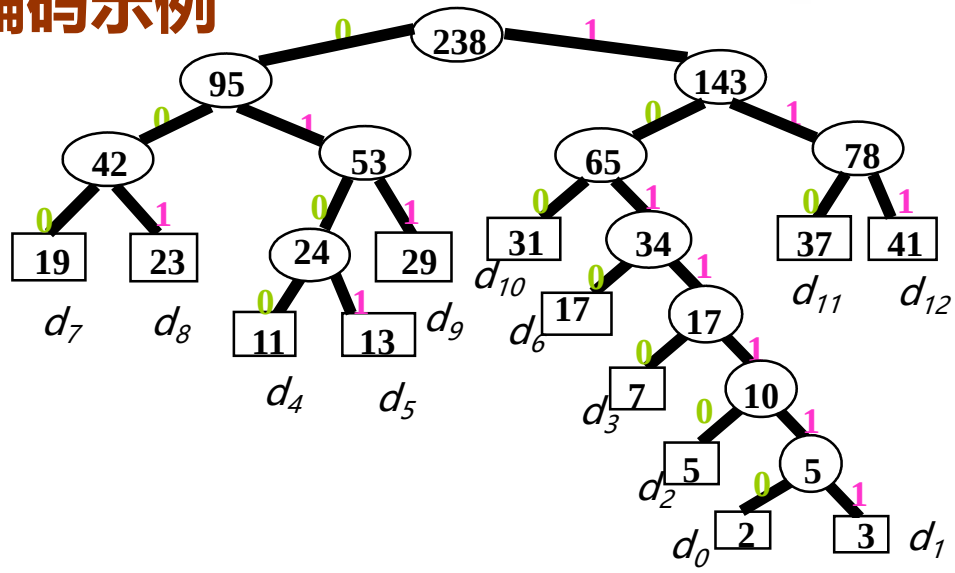
- 从根到叶结点的路径上的0、1标记组成该叶结点所代表字符的编码
  - ✓ 结点到 其左子结点 的 边标记为 0
  - ✓ 结点到 其右子结点 的 边标记为 1

5.7 Huffman树及其应用

Huffman编码示例

- 各字符的二进制编码为：

d_0：1011110　d_1：1011111
d_2：101110　d_3：10110
d_4：0100　d_5：0101
d_6：1010　d_7：000
d_8：001　d_9：011
d_{10}：100　d_{11}：110
d_{12}：111



- 出现频率越大的字符，编码越短

5.7 Huffman树及其应用

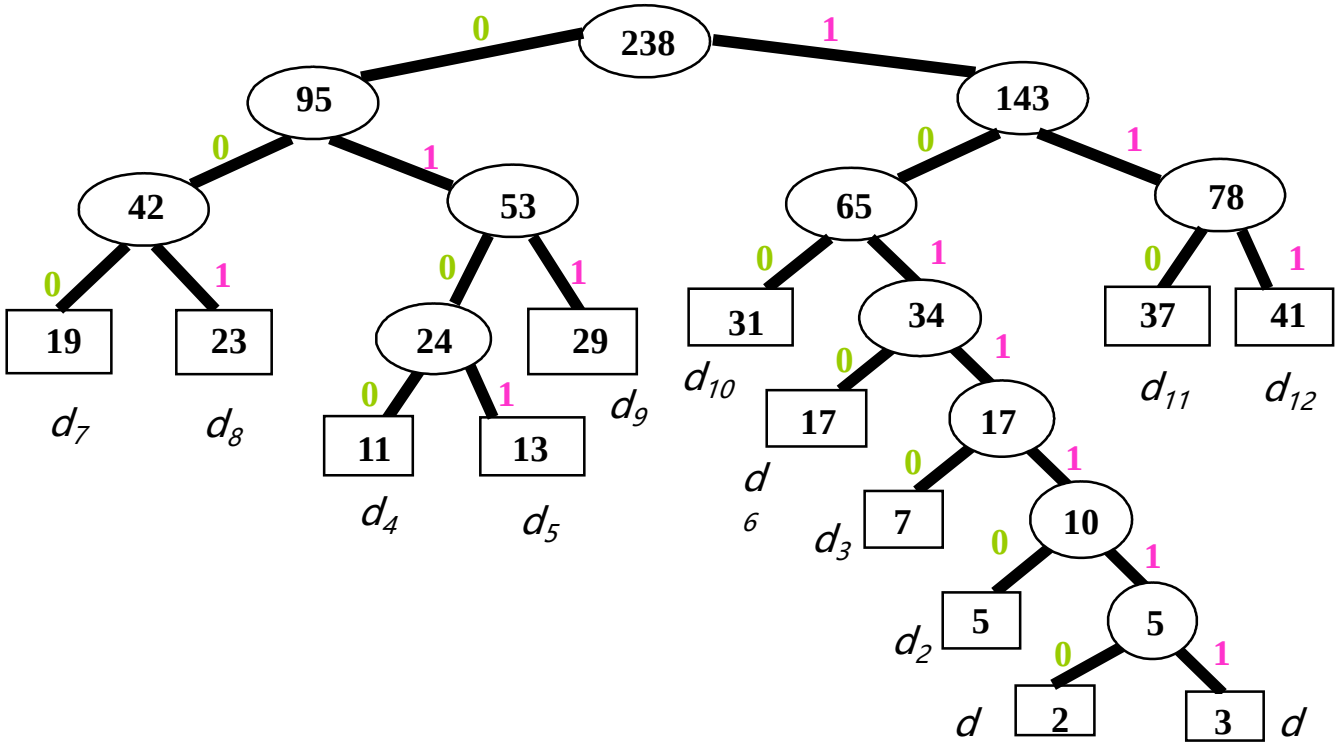
Huffman编码及其解码

- 采用Huffman算法构造的扩充二叉树既可编码字符集，也用来解码/译码二进制代码串
- 译码与编码过程相逆
  - ✓ 从树的根结点开始
    - 沿 0 下降到左分支，沿 1 下降到右分支
    - 直到一个**叶结点**，译出了一个字符
  - ✓ 连续译码
    - 回到树根
    - 从二进制代码串中的下一位开始继续译码

5.7 Huffman树及其应用

Huffman译码示例

111 101110



5.7 Huffman树及其应用

Huffman树的类定义

```
template<class T>  class HuffmanTree  {
private:
    HuffmanTreeNode <T> *root;                                  // Huffman树的根结点
    //  将以ht1和ht2为根的两棵Huffman树合并成一棵以parent为根的二叉树
    void MergeTree(HuffmanTreeNode<T> &ht1, HuffmanTreeNode<T>
                   &ht2，HuffmanTreeNode<T> *parent);
    // 删除Huffman树或其子树
    void DeleteTree(HuffmanTreeNode<T> *root);
public:
    // 构造Huffman树，参数weight为权值数组，n为数组长度
    HuffmanTree(T weight[], int n);
    virtual ~HuffmanTree() {DeleteTree(root);};    // 析构函数
}
```

5.7 Huffman树及其应用

Huffman树的构造

```
template<class T> HuffmanTree<T>::HuffmanTree(T weight[], int n)  {
    MinHeap< HuffmanTreeNode<T> > heap(n);          // 最小值堆
    HuffmanTreeNode<T> *parent, firstchild, secondchild;
    HuffmanTreeNode<T> *NodeList = new HuffmanTreeNode<T>[n];
    for (int i = 0; i < n; i++) {                                   // 初始化
      NodeList[i].info = weight[i];
      NodeList[i].parent = NodeList[i].left = NodeList[i].right = NULL;
      heap.Insert(NodeList[i]);                                     // 向堆中添加元素
    }
    for (i = 0; i < n-1; i++) {                                     // 通过n-1次合并建立Huffman树
      parent = new HuffmanTreeNode<T>;              // 申请一个分支结点
      firstchild = heap.RemoveMin();                                // 选择权值最小的结点
      secondchild = heap.RemoveMin();                               // 选择权值次小的结点
      MergeTree(firstchild, secondchild, parent);   // 将权值最小的两棵树合并到parent树
      heap.Insert(*parent);                                         // 把parent插入到堆中去
      root = parent;                                                // Huffman树的根结点赋为parent
    }
    delete [] NodeList;
}
```

5.7 Huffman树及其应用

Huffman方法的正确性证明

- 是否前缀编码?
- 是否最优解?
  - ✓ 贪心法的典型例子： Huffman树建立的每一步，“权”最小的两个子树被合并为一棵新子树

5.7 Huffman树及其应用

Huffman正确性证明

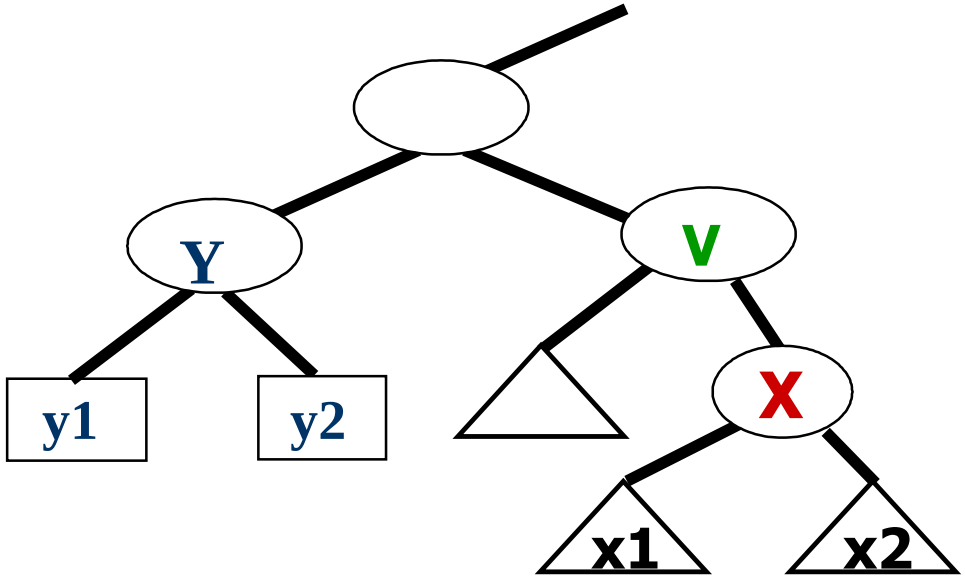
- 引理：**Huffman性质**

  一棵含有两个以上结点的Huffman 树中，使用频率最小的两个字符是兄弟结点，而且其深度不比树中其他任何叶结点浅

5.7 Huffman树及其应用

Huffman正确性证明

- 记使用频率最低的两个字符为 y1 和 y2
- 假设 x1, x2 是最深的结点
  - ✓ y1 和 y2 的父结点 Y 一定会有比 X 更大的“权”
  - ✓ 否则，会选择 Y 而不是 X 作为结点 V 的子结点
- 然而，由于 y1 和 y2 是频率最小的字符，这种情况不可能发生



5.7 Huffman树及其应用

Huffman正确性证明

- 定理：对于给定的一组字符，函数HuffmanTree实现了“最小外部路径权重”
- 证明：对字符个数 n 归纳
  - ✓ 归纳基础：令n = 2, Huffman树一定有最小外部路径权重
    - 只可能有成镜面对称的两种树
    - 两种树的叶结点加权路径长度相等
  - ✓ 归纳假设：
    - 假设由函数 HuffmanTree产生的具有n-1个叶结点的 Huffman 树有最小外部路径权重

5.7 Huffman树及其应用

Huffman正确性证明

- 归纳步骤：
  - ✓ 设一棵由函数HuffmanTree产生的树 T 有 n 个叶结点，n>2，并假设字符的“权” $w_0 \le w_1 \le \ldots \le w_{n-1}$
    - 记 V 是频率为 w_0 和 w_1 的两个字符的父结点，据引理，它们已是树 T 中最深的结点
    - T 中结点 V 换为一个叶结点 V′ （权等于$w_0 + w_1$） 得到另一棵树 T′
  - ✓ 根据归纳假设，T′ 具有最小的外部路径长度，将 V′ 展开为 V（$w_0 + w_1$），T′ 还原为 T， 则

    T 也应有最小的外部路径长度

因此，根据归纳原理，定理成立

5.7 Huffman树及其应用

Huffman树编码效率

- Huffman编码的空间效率：字符的平均编码长度
  - ✓ 各字符的编码长度c_i乘以其出现概率 p_i，即:

$$c_0p_0 + c_1p_1 + \ldots + c_{n-1}p_{n-1},$$

or

  - ✓ f_i表示第i个字符的出现频率，f_T表示所有字符出现的总次数

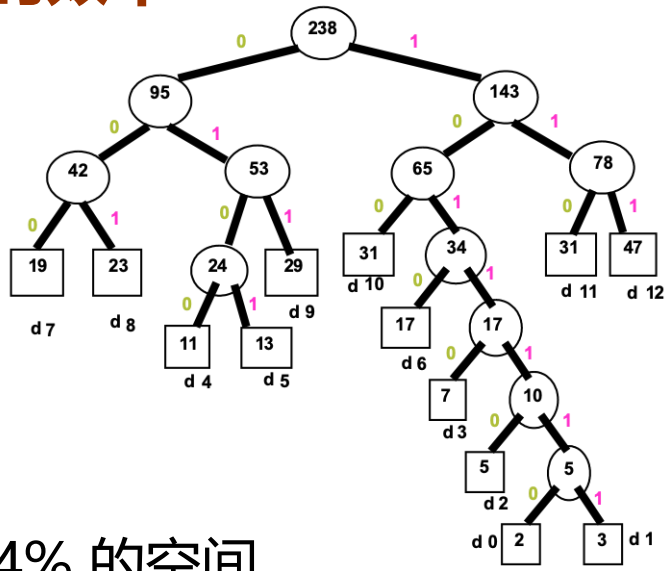
$$(c_0f_0 + c_1f_1 + \ldots + c_{n-1}f_{n-1}) / f_T$$

5.7 Huffman树及其应用

Huffman树编码效率

编码的平均长度为：

(3*(19+23+29+31+31+47)
+ 4*(11+13+17)
+ 5*7
+ 6*5
+ 7*(2+3)) / 238
= 804/238 ≈ 3.38



- Huffman编码只需等长编码3.38/4≈ 84% 的空间
  - ✓ 对于这13个字符，等长编码每个字符需要⌈log 13⌉= 4位
  - ✓ Huffman编码只需3.38位

5.7 Huffman树及其应用

Huffman树的应用

- Huffman编码适合于
  - ✓ 字符 频率不等、差别较大 字符集
  - ✓ 不同的频率分布，会有不同的压缩比率
  - ✓ 大多数商业压缩软件都是采用几种编码方式以因应各种类型的文件
    - zip 压缩就是 LZ77 与 Huffman 结合
- 外排序 归并顺串

思考

- 编制一个将百分制转换成五分制的程序，怎样才能使得程序中的比较次数最少？
- 成绩分布如下：

分数	0 - 59	60 - 69	70 - 79	80 - 89	90 - 100
比例数	0.05	0.15	0.40	0.30	0.10

数据结构与算法

谢谢倾听

国家精品课“数据结构与算法”

http://jpk.pku.edu.cn/course/sjjg/

https://www.icourse163.org/course/PKU-1002534001

张铭，王腾蛟，赵海燕

高等教育出版社，2008. 6。“十二五”国家级规划教材