数据结构与算法（A）-W05/二叉树

北京大学 陈斌

2024.09.27

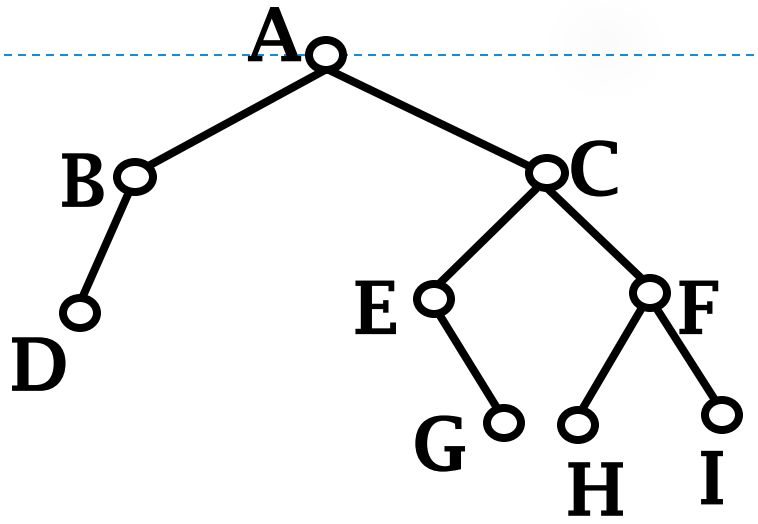
第五章 二叉树

赵海燕 主讲

采用教材： 《数据结构与算法》，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）

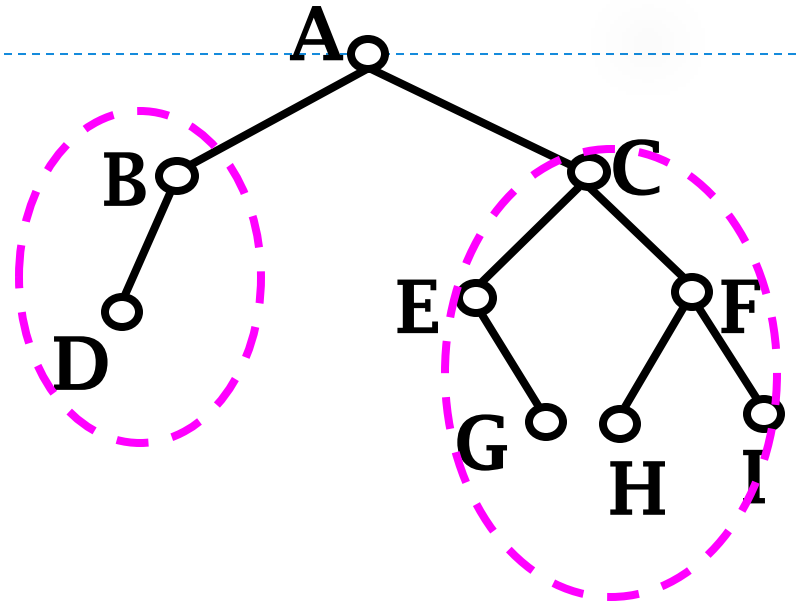
http://jpk.pku.edu.cn/course/sjjg/
https://www.icourse163.org/course/PKU-1002534001

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

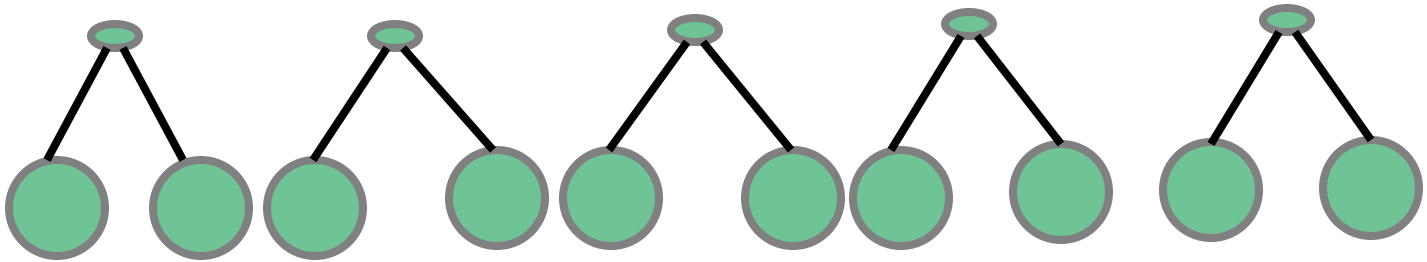
二叉树的概念



- **二叉树的定义**
  - 二叉树 (binary tree) 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

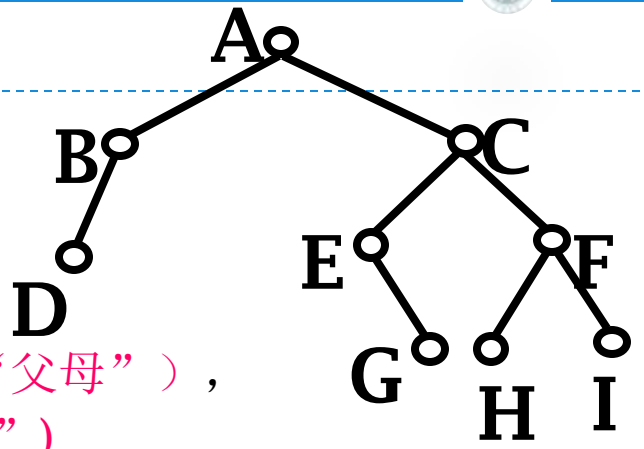
二叉树的五种基本形态

- 二叉树可以是空集合，因此根可以有空的左子树或右子树，或者左右子树皆为空



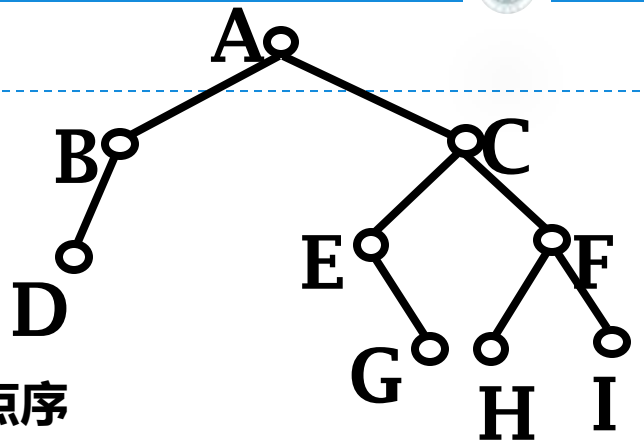
(a)空　(b)独根　(c)空右　(d)空左　(e)左右都不空

二叉树相关术语



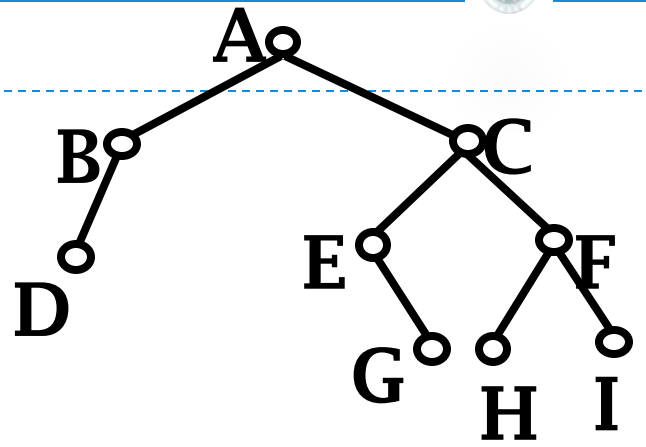
- **结点**
  - **子结点、父结点、最左子结点**
    - 若 <k，k′>∈ r，则称 k 是 k′ 的父结点（或“父母”），而 k′ 则是 k 的 子结点 (或“儿子”、“子女”)
  - **兄弟结点、左兄弟、右兄弟**
    - 若有序对 <k，k′>及 <k，k″>∈ r，则称 k′和 k″互为兄弟
  - **分支结点、叶结点**
    - 没有子树的结点称作 叶结点 (或树叶、终端结点)
    - 非终端结点称为分支结点

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 **子孙**

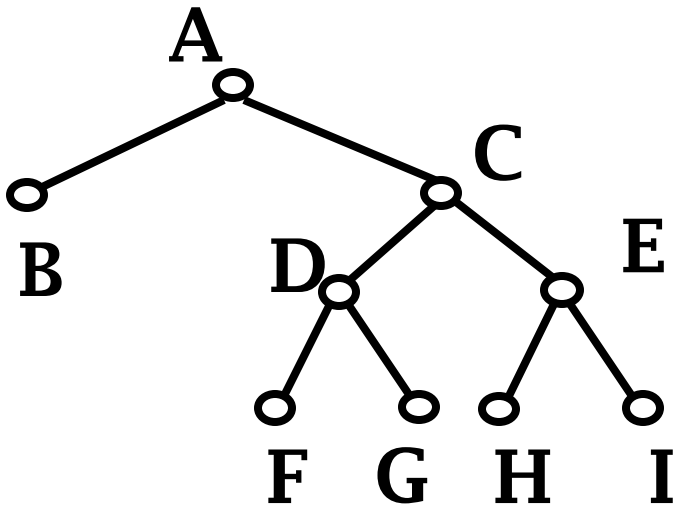
二叉树相关术语



- **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

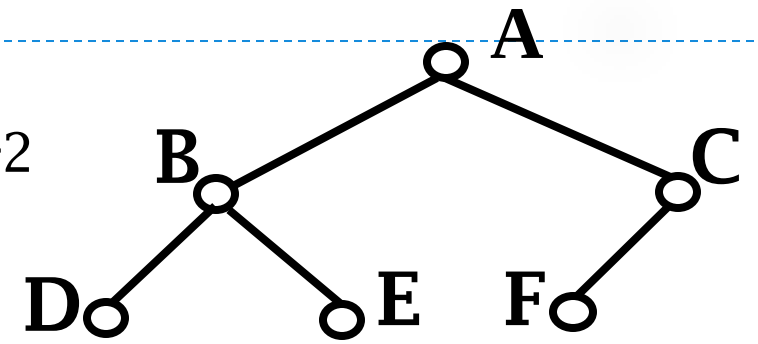
- 如果一棵二叉树的 **任何** 结点，或者是树叶，或者恰有两棵非空子树，则此二叉树称作 **满二叉树**

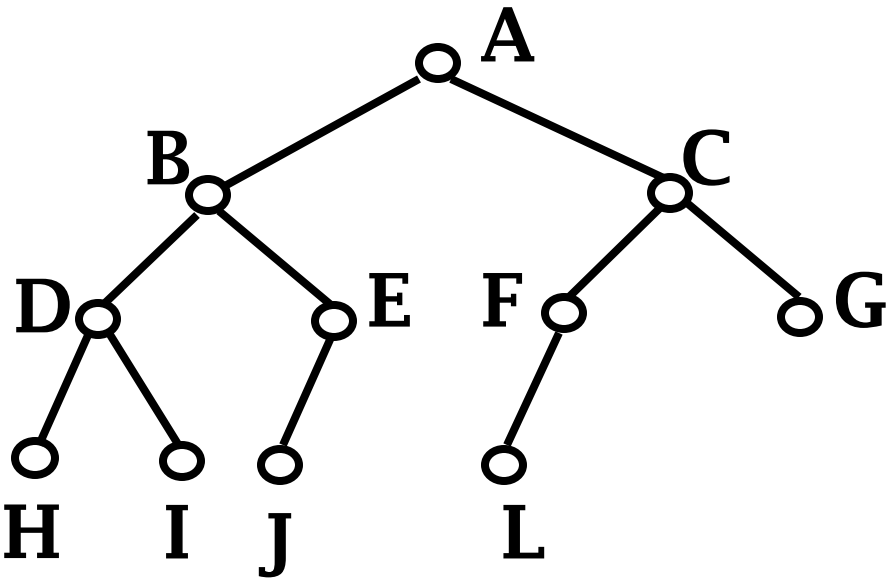


5.1 二叉树的概念

完全二叉树

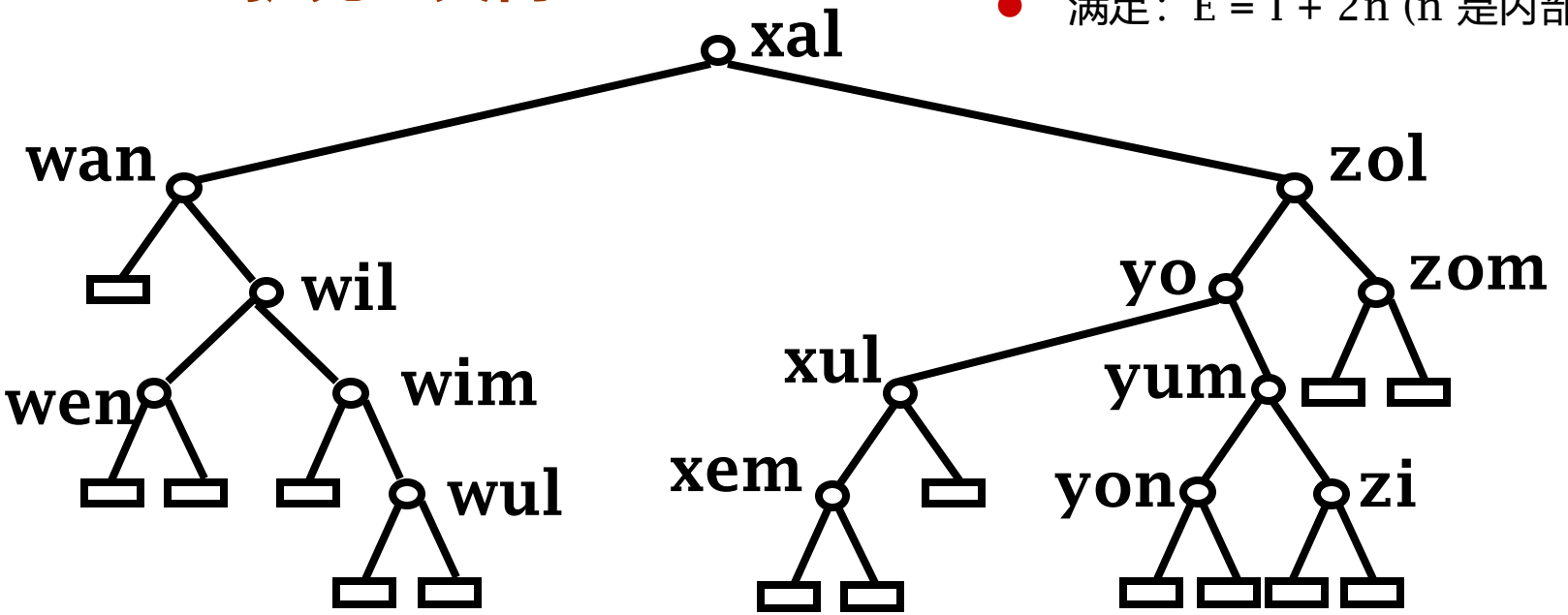
- 最多只有最下面的两层结点度数可以小于2
- 最下一层的结点都集中最左边





- 所有空子树，都增加空树叶
- 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)

扩充二叉树



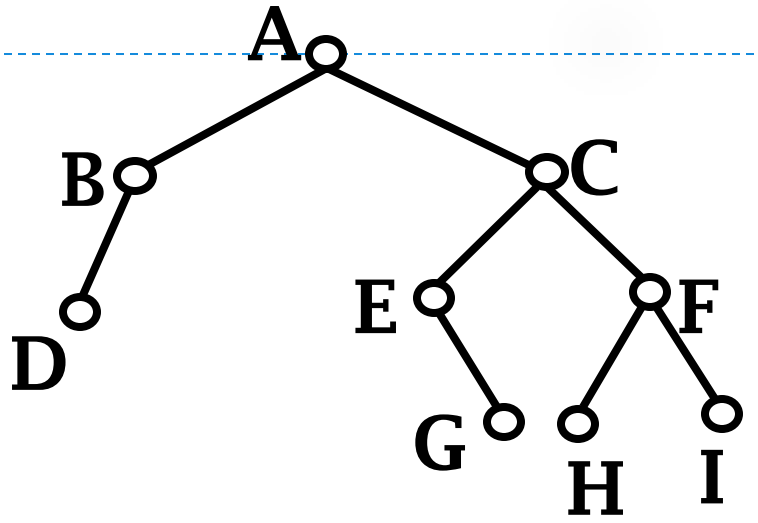
二叉树的主要性质

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

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

抽象数据类型

- **逻辑结构 + 运算：**
- **针对整棵树**
  - 初始化二叉树
  - 合并两棵二叉树
- **围绕结点**
  - 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

二叉树结点ADT

```
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
```

5.2 二叉树的抽象数据类型

```
    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
};
```

二叉树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 二叉树的抽象数据类型

```
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
```

遍历二叉树

- **遍历** (或称**周游**，**traversal**)
  - **系统地访问数据结构中的结点**
  - **每个结点都正好被访问到一次**
- 二叉树的结点的 **线性化**

深度优先遍历二叉树

三种深度优先遍历的递归定义：

(1) **前序法 (tLR次序，preorder traversal)。**

访问根结点；　　按前序遍历左子树；　按前序遍历右子树。

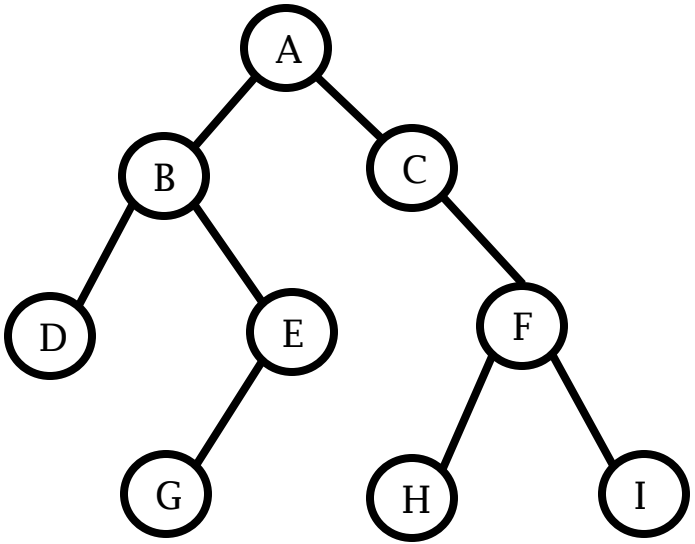
(2) **中序法 (LtR次序，inorder traversal)。**

按中序遍历左子树；　访问根结点；　　按中序遍历右子树。

(3) **后序法 (LRt次序，postorder traversal)。**

按后序遍历左子树；　按后序遍历右子树；　访问根结点。

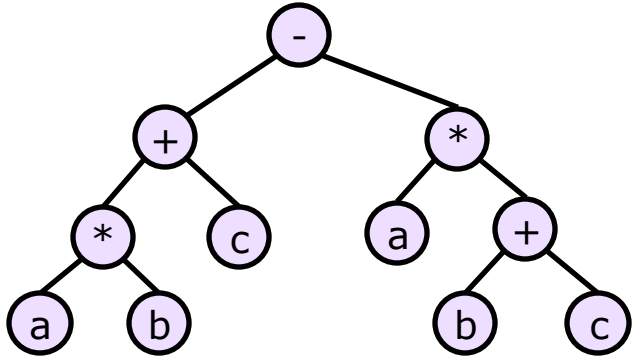
深度优先遍历二叉树



- 前序序列是：A B D E G C F H I
- 中序序列是：D B G E A C H F I
- 后序序列是：D G E B H I F C A

表达式二叉树

- 前序(前缀)： - + * a b c * a + b c
- 中序： a * b + c - a * (b + c)
- 后序(后缀)： a b * c + a b c + * -



深度优先遍历二叉树（递归）

```
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
      }
}
```

思 考

- 前、中、后序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};
    则其前序序列为 _____。

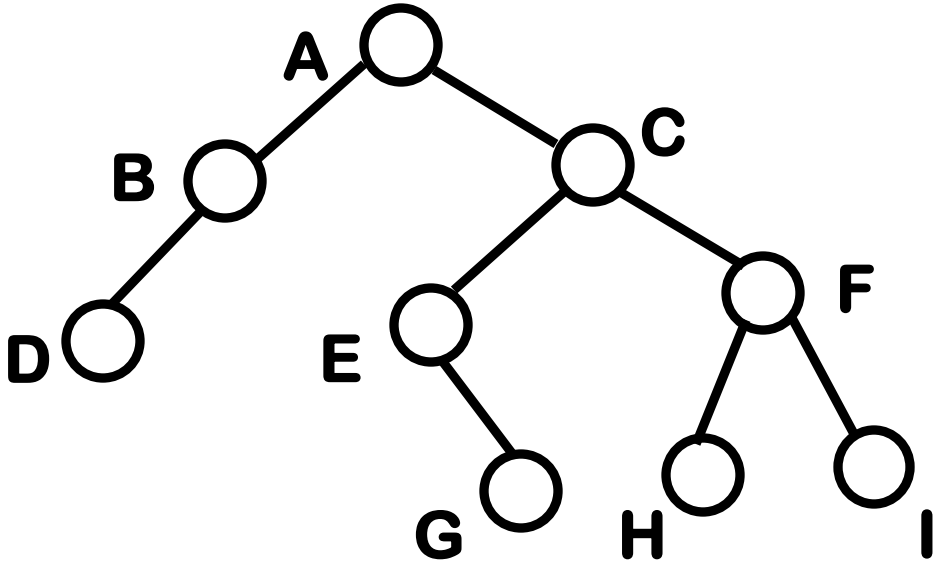
DFS遍历二叉树的非递归算法

- 递归算法非常简洁——推荐使用
  - 当前的编译系统优化效率很不错了
- 特殊情况用栈模拟递归
  - 理解编译栈的工作原理
  - 理解深度优先遍历的回溯特点
  - 有些应用环境资源限制不适合递归

5.2 二叉树的抽象数据类型

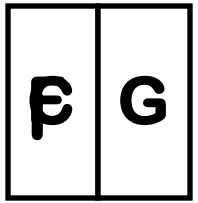
前序序列 A B D E H

入栈序列 C F G I



非递归前序遍历

栈



访问结点

栈中结点

已访问结点

非递归前序遍历二叉树

- **思想：**
  - 遇到一个结点，就访问该结点，并把此结点的非空右结点推入栈中，然后下降去遍历它的左子树；
  - 遍历完左子树后，从栈顶托出一个结点，并按照它的右链接指示的地址再去遍历该结点的右子树结构。

```
template<class T>  void
BinaryTree<T>::PreOrderWithoutRecusion
(BinaryTreeNode<T>* root) {
```

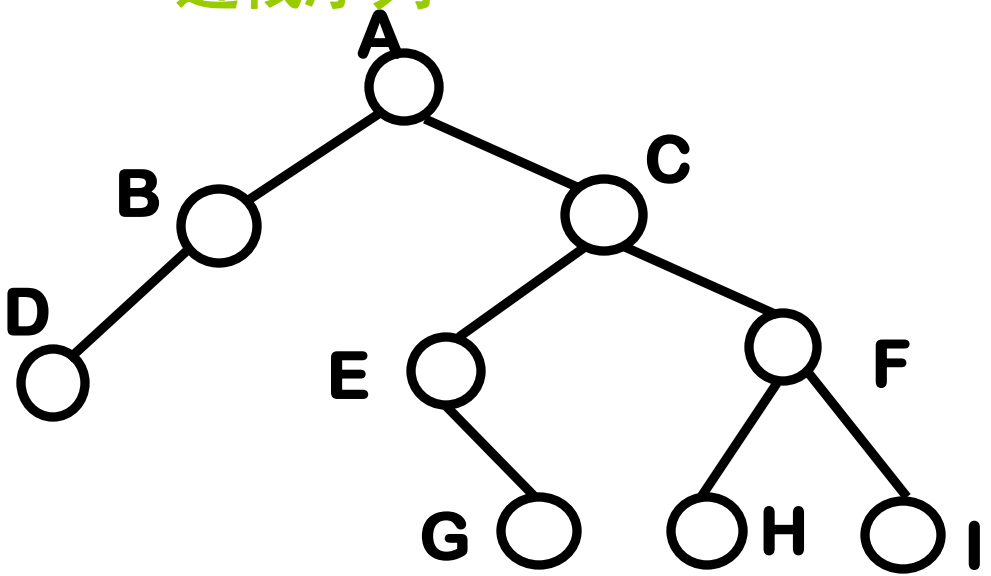
5.2 二叉树的抽象数据类型

```
    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->value());         // 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  {                      // 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 }
} }
```

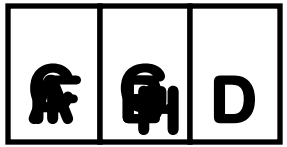
5.2 二叉树的抽象数据类型

中序序列

进栈序列 A B D C E G F H I



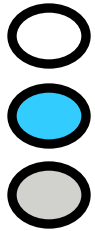
栈



未访问结点

栈中结点

出栈结点



非递归中序遍历二叉树

- 遇到一个结点
  - 把它推入栈中
  - 遍历其左子树
- 遍历完左子树
  - 从栈顶托出该结点并访问之
  - 按照其右链地址遍历该结点的右子树

5.2 二叉树的抽象数据类型

```
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
}
```

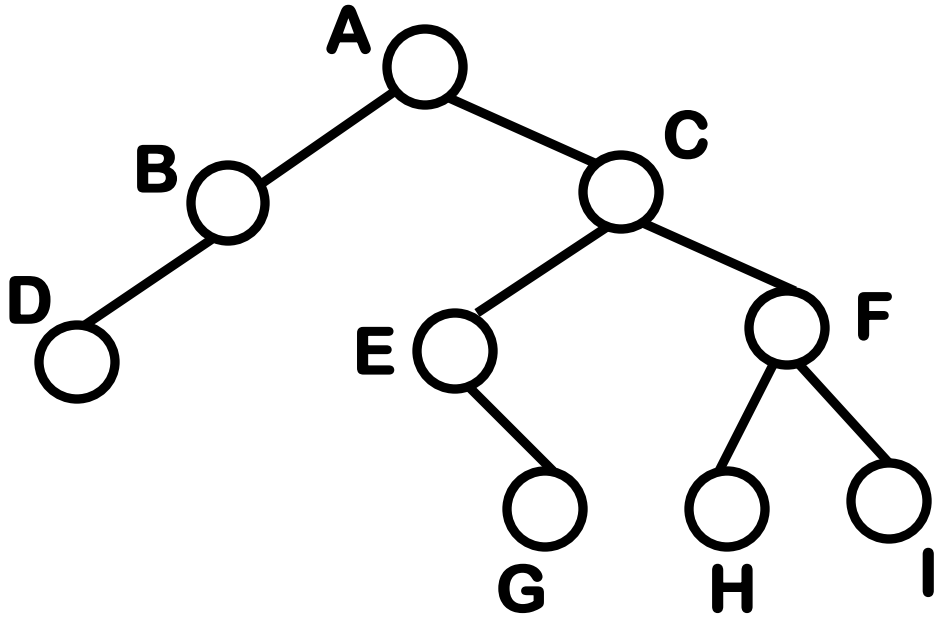
非递归后序遍历二叉树

- 左子树返回 vs 右子树返回？
- 给栈中元素加上一个特征位
  - Left 表示已进入该结点的左子树，
    将从左边回来
  - Right 表示已进入该结点的右子树，
    将从右边回来

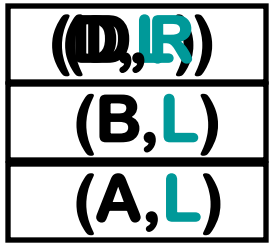
非递归后序遍历二叉树

后序序列　D

出栈序列



栈



未访问结点

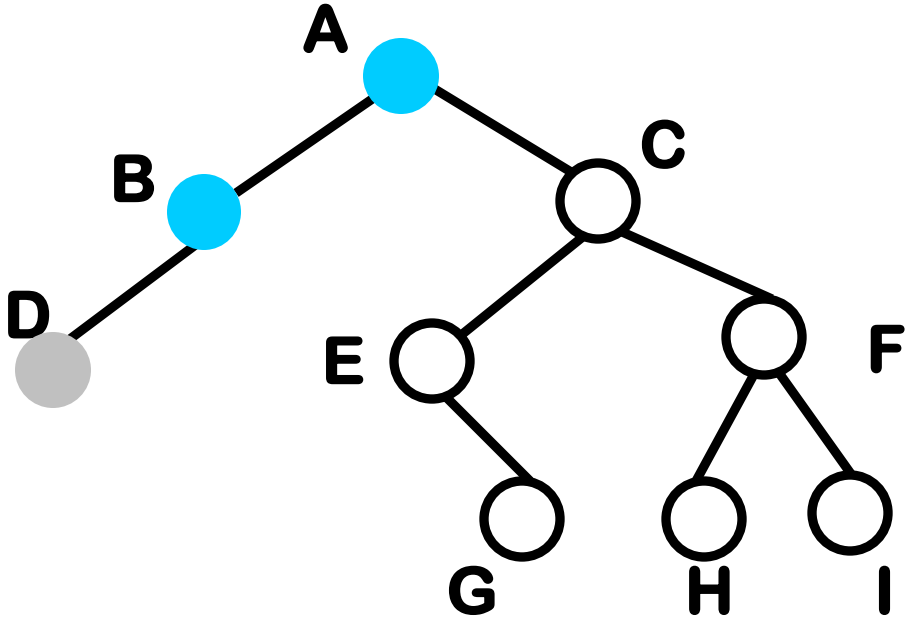
栈中结点

出栈结点

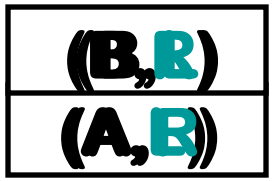
5.2 二叉树的抽象数据类型

后序序列 D B

出栈序列 (D,L) (D,R)



栈



未访问结点

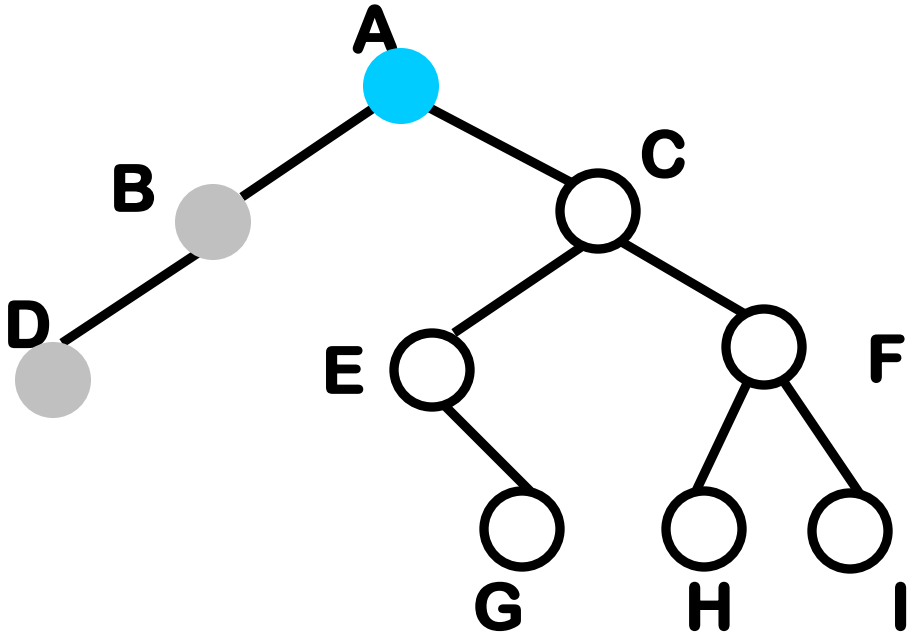
栈中结点

出栈结点

5.2 二叉树的抽象数据类型

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

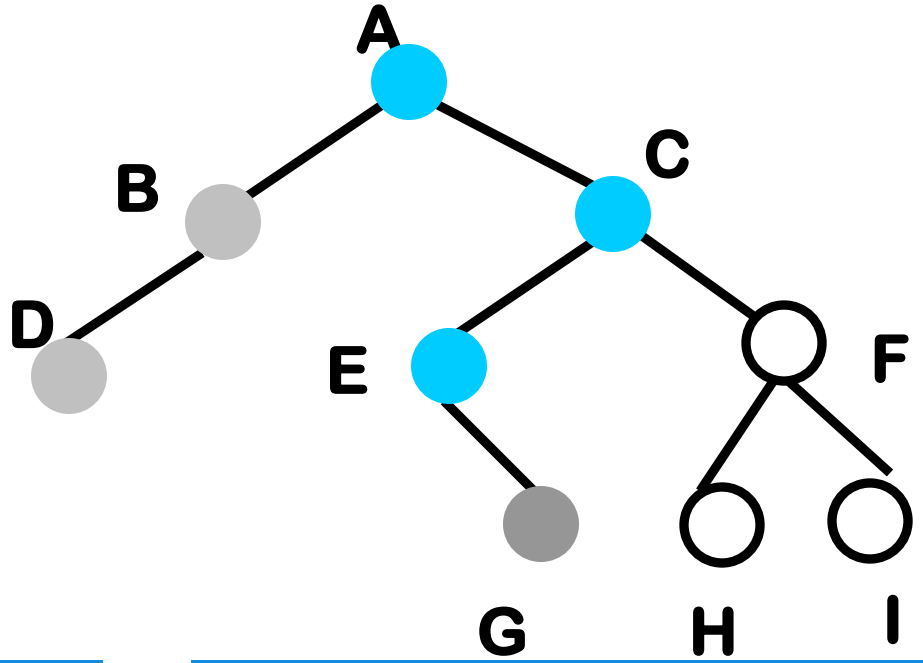
栈中结点

出栈结点

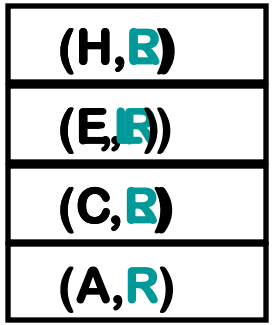
5.2 二叉树的抽象数据类型

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)



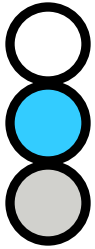
栈



未访问结点

栈中结点

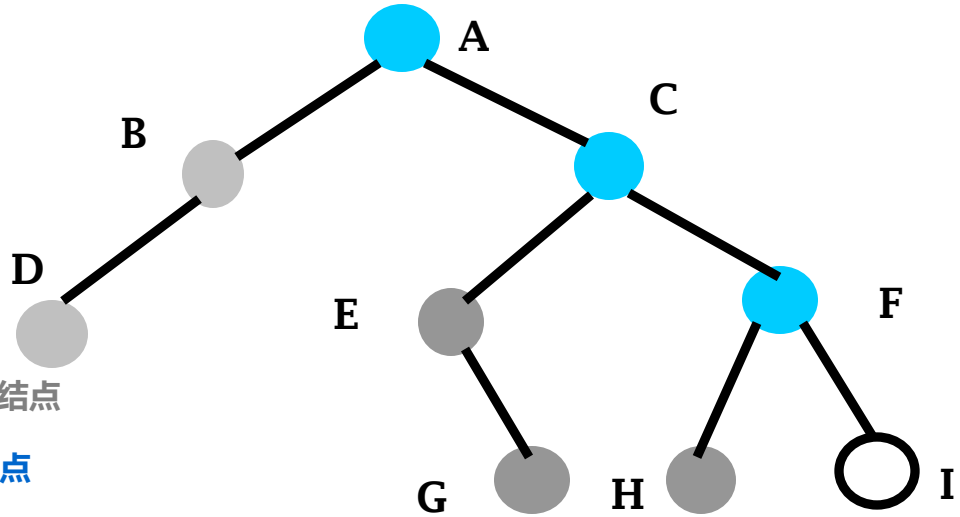
出栈结点



5.2 二叉树的抽象数据类型

后序序列 D B G E H I F C A

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



栈

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

未访问结点

栈中结点

出栈结点

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;
```

5.2 二叉树的抽象数据类型

```
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
   }
 }
}
```

二叉树遍历算法的时间代价分析

- 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（或队列）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)

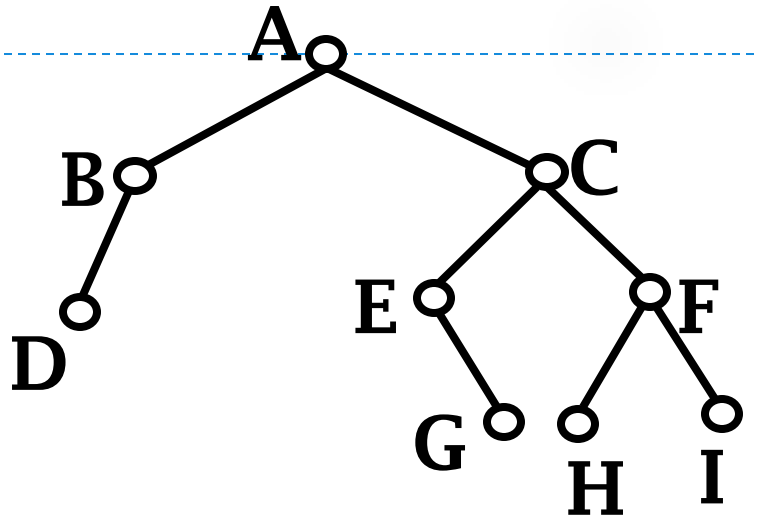
二叉树遍历算法的空间代价分析

- 深搜：栈的深度与树的高度有关
  - 最好 O(log n)
  - 最坏 O(n)

思　考

- 非递归遍历的意义？
  - 后序遍历时，栈中结点有何规律？
  - 栈中存放了什么？
- 前序、中序、后序框架的算法通用性？
  - 例如后序框架是否支持前序、中序访问？
  - 若支持，怎么改动？

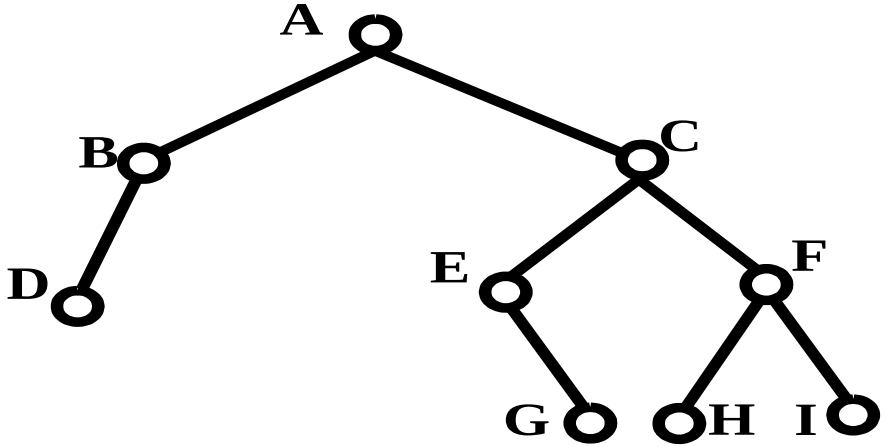
第五章 二叉树



- 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

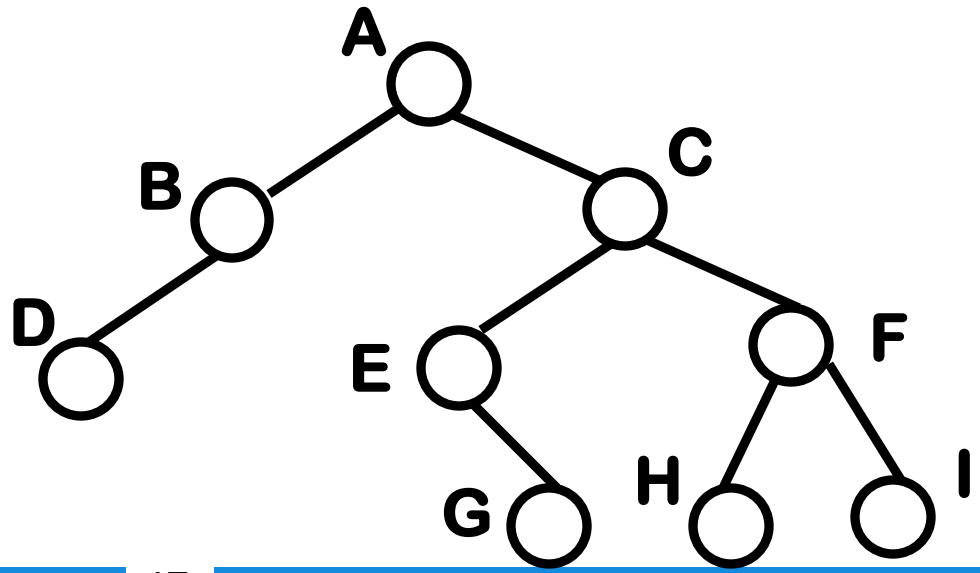
宽度优先遍历二叉树

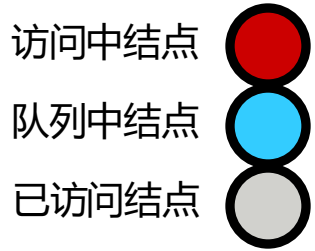
- 从二叉树的第 0 层（根结点）开始，自上至下 逐层遍历；在同一层中，按照 从左到右 的顺序对结点逐一访问。
- 例如：A B C D E F G H I



BFS序列

队列





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
 } }
```

二叉树遍历算法的时间代价分析

- 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（或队列）时间
  - 宽搜，正好每个结点入/出队一次，O(n)

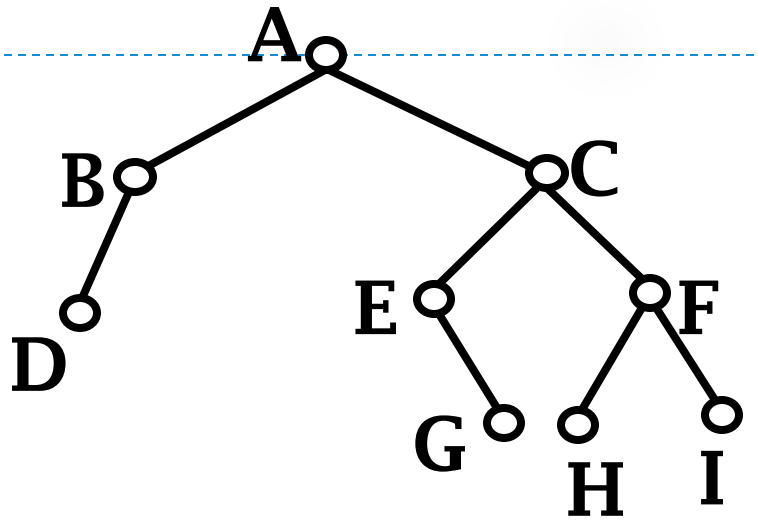
二叉树遍历算法的空间代价分析

- 宽搜：与树的最大宽度有关
  - 最好 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
- 二叉搜索树
- 堆与优先队列
- Huffman树及其应用

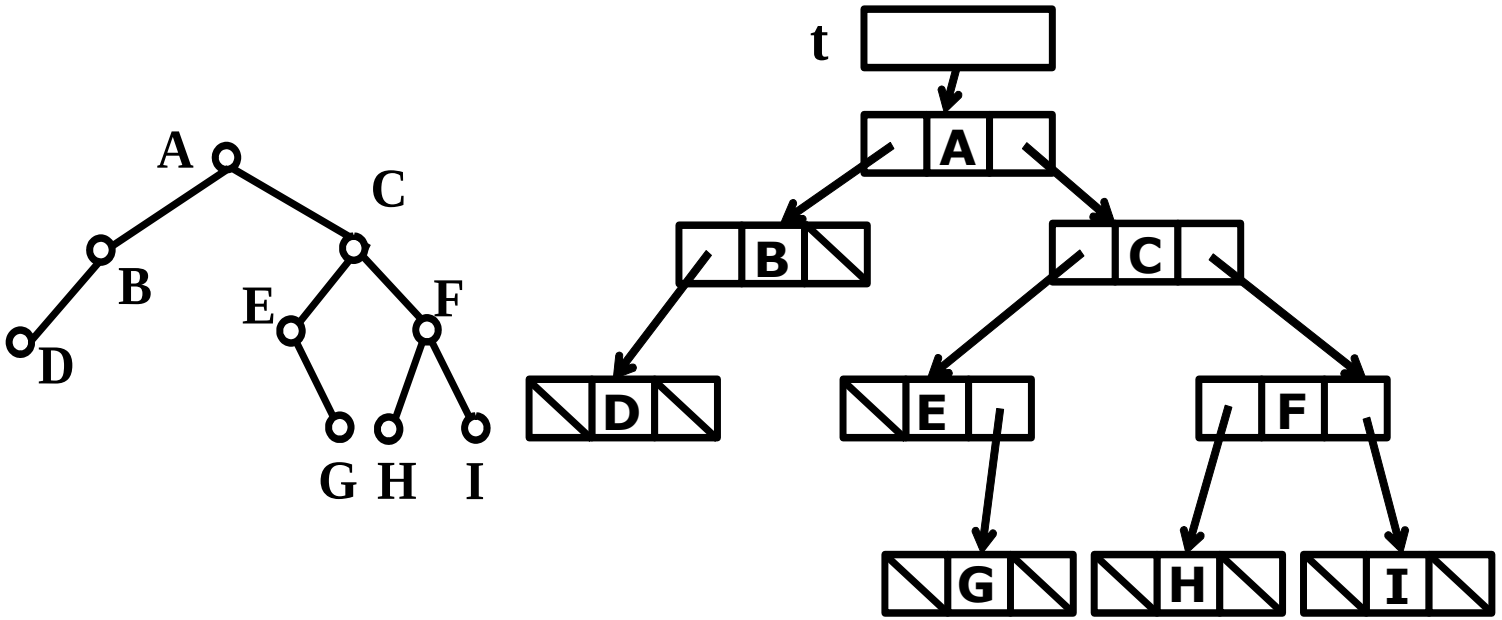
二叉树的链式存储结构

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

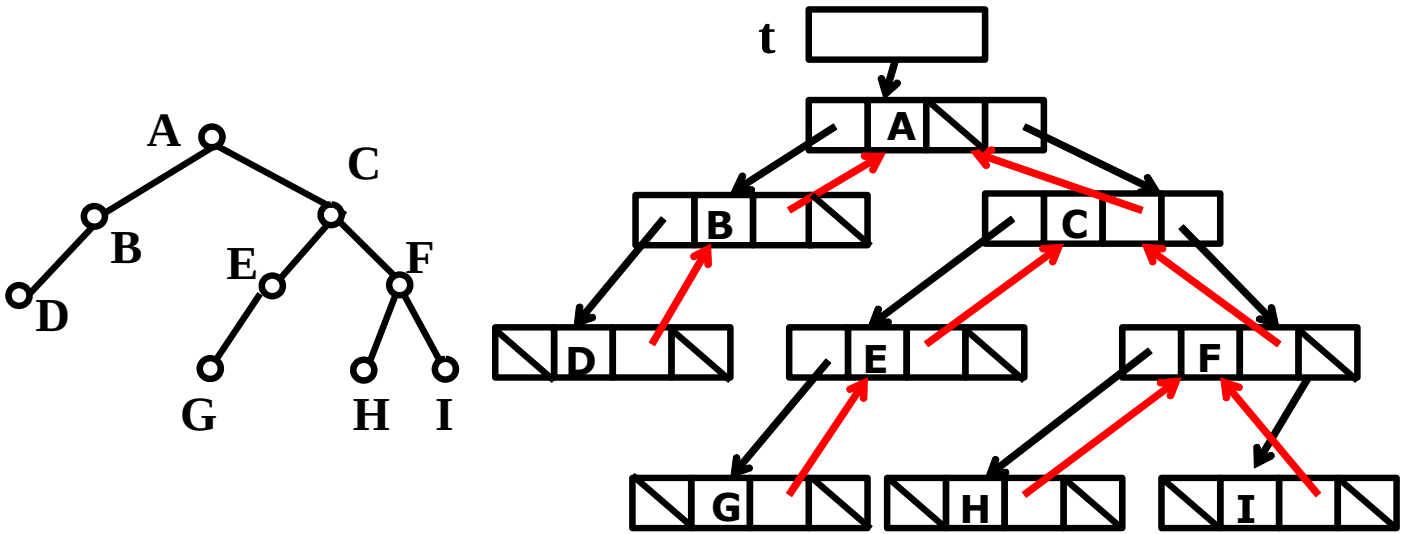
- 三叉链表
  - 指针 left 和 right，分别指向结点的左孩子和右孩子
  - 增加一个父指针

二叉链表



“三叉链表”

- 指向父母的指针parent, “向上”能力



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
.... }
```

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;
}
```

思考

- 该算法是什么框架?
- 该算法是什么序遍历?
- 可以怎样改进?
  - 可以用非递归吗?
  - 可以用BFS吗?
- 怎样从这个算法出发，寻找兄弟结点

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;                 // 如果pointer的孩子是current则返回parent
        if (pointer->rightchild() != NULL)              // 非空右孩子入栈
            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(); aStack.pop();   // 获得栈顶元素，并退栈
}   }   }
```

空间开销分析

- 存储密度 α (≤1) 表示数据结构存储的效率

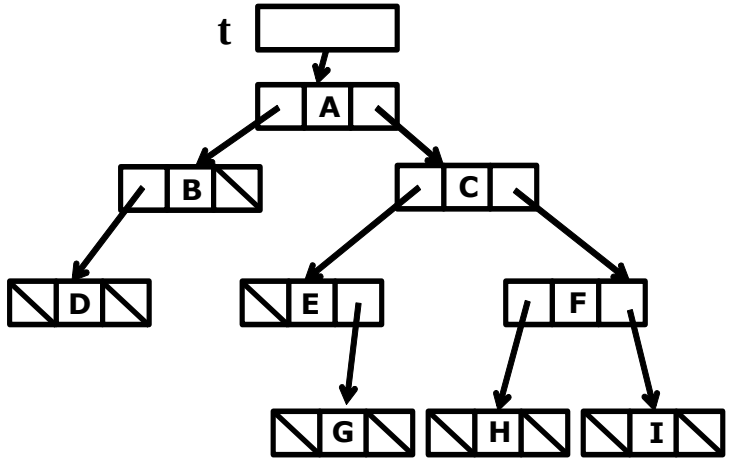
$$\alpha(\text{存储密度}) = \frac{\text{数据本身存储量}}{\text{整个结构占用的存储总量}}$$

- 结构性开销 γ = 1- α

空间开销分析

根据满二叉树定理：一半的指针是空的

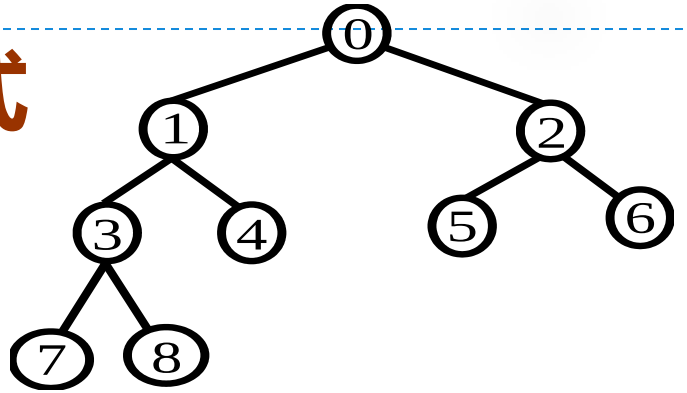
- 每个结点存两个指针、一个数据域
  - 总空间 $(2p + d)n$
  - 结构性开销：$2pn$
  - 如果 $p = d$, 则结构性开销 $2p/(2p + d) = 2/3$



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：
  - 用union联合类型定义
  - 使用C++的子类来分别实现分支结点与叶结点，
    并采用虚函数isLeaf来区别分支结点与叶结点
- 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

完全二叉树的下标公式

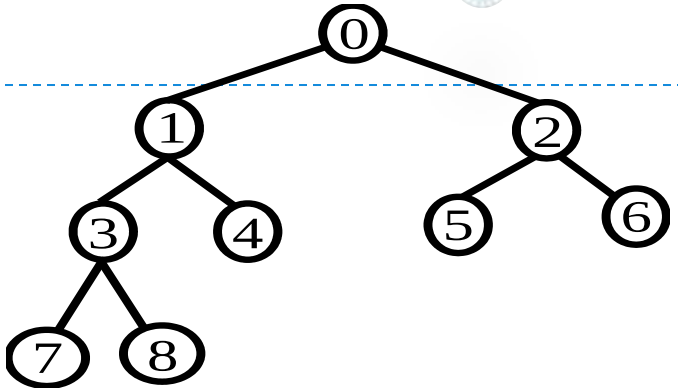


- 从结点的编号就可以推知

其父母、孩子、兄弟的编号

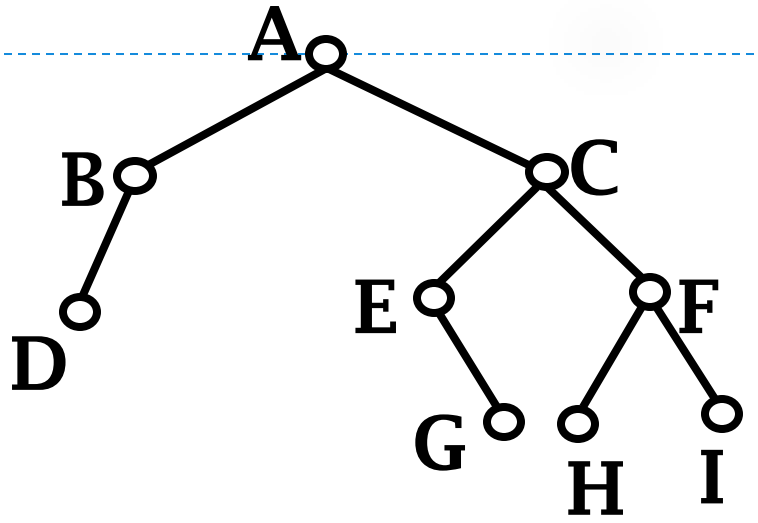
- 当 2i+1<n 时，结点 i 的左孩子是结点 2i+1，
  否则结点i没有左孩子
- 当 2i+2<n 时，结点 i 的右孩子是结点 2i+2，
  否则结点i没有右孩子

完全二叉树的下标公式



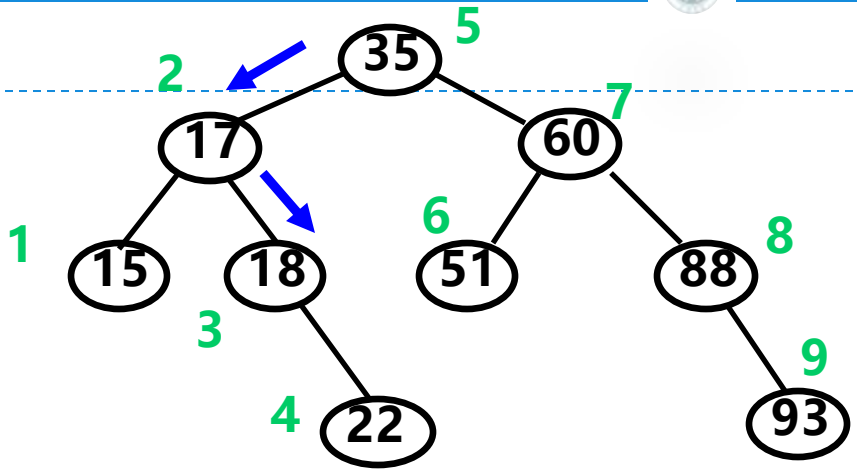
- 当 0<i<n 时，结点 i 的父亲是结点 ⌊(i-1)/2⌋
- 当 i 为偶数且 0<i<n 时，结点 i 的左兄弟是结点 i-1，
  否则结点 i 没有左兄弟
- 当 i 为奇数且 i+1<n 时，结点i的右兄弟是结点 i+1，
  否则结点i没有右兄弟

第五章 二叉树



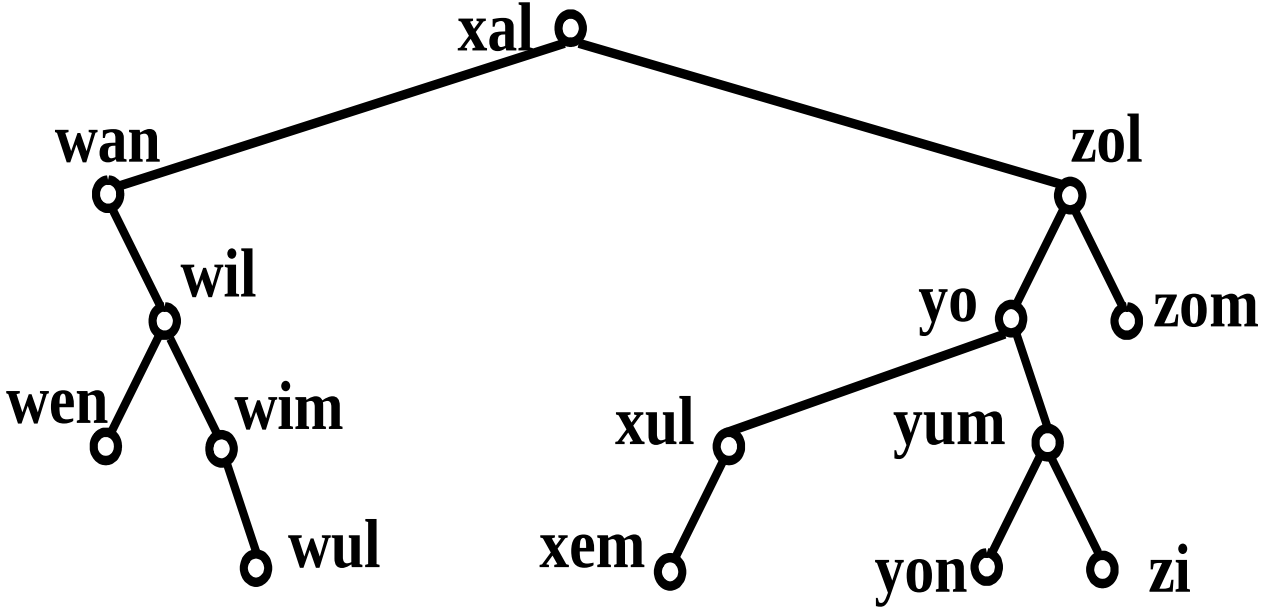
- 二叉树的概念
- 二叉树的抽象数据类型
  - 深度优先搜索
  - 宽度优先搜索
- 二叉树的存储结构
- 二叉搜索树
- 堆与优先队列
- Huffman树及其应用

二叉搜索树

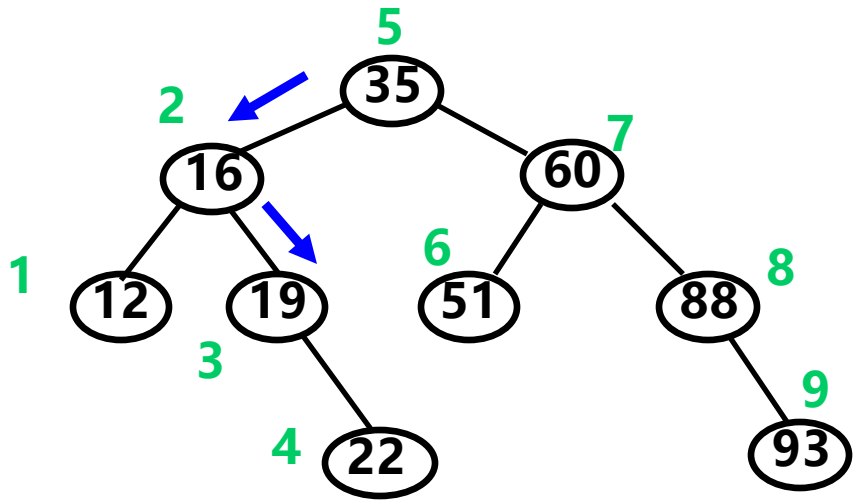


- Binary Search Tree（**BST**）
  - **或者是一棵空树；**
  - **或者是具有下列性质的二叉树：**
    - **对于任何一个结点，设其值为K**
    - **则该结点的 左子树(若不空)的任意一个结点的值都 小于 K；**
    - **该结点的 右子树(若不空)的任意一个结点的值都 大于 K；**
    - **而且它的左右子树也分别为BST**
- 性质: 中序遍历是正序的（由小到大的排列）

BST示意图



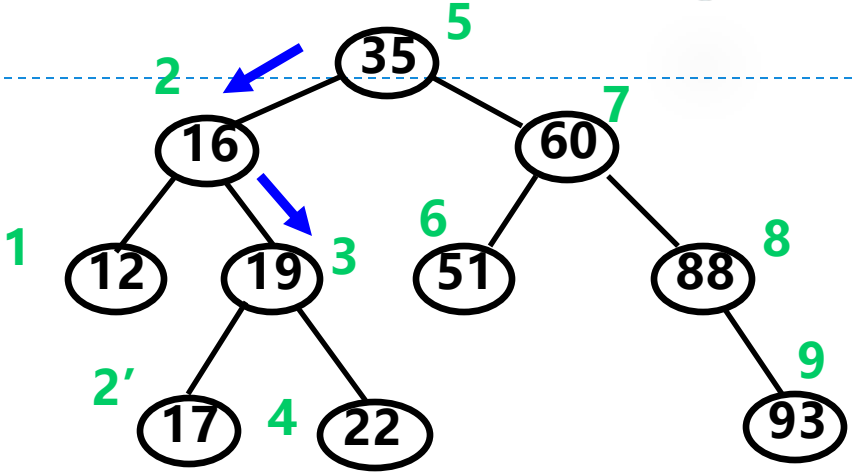
检索 19

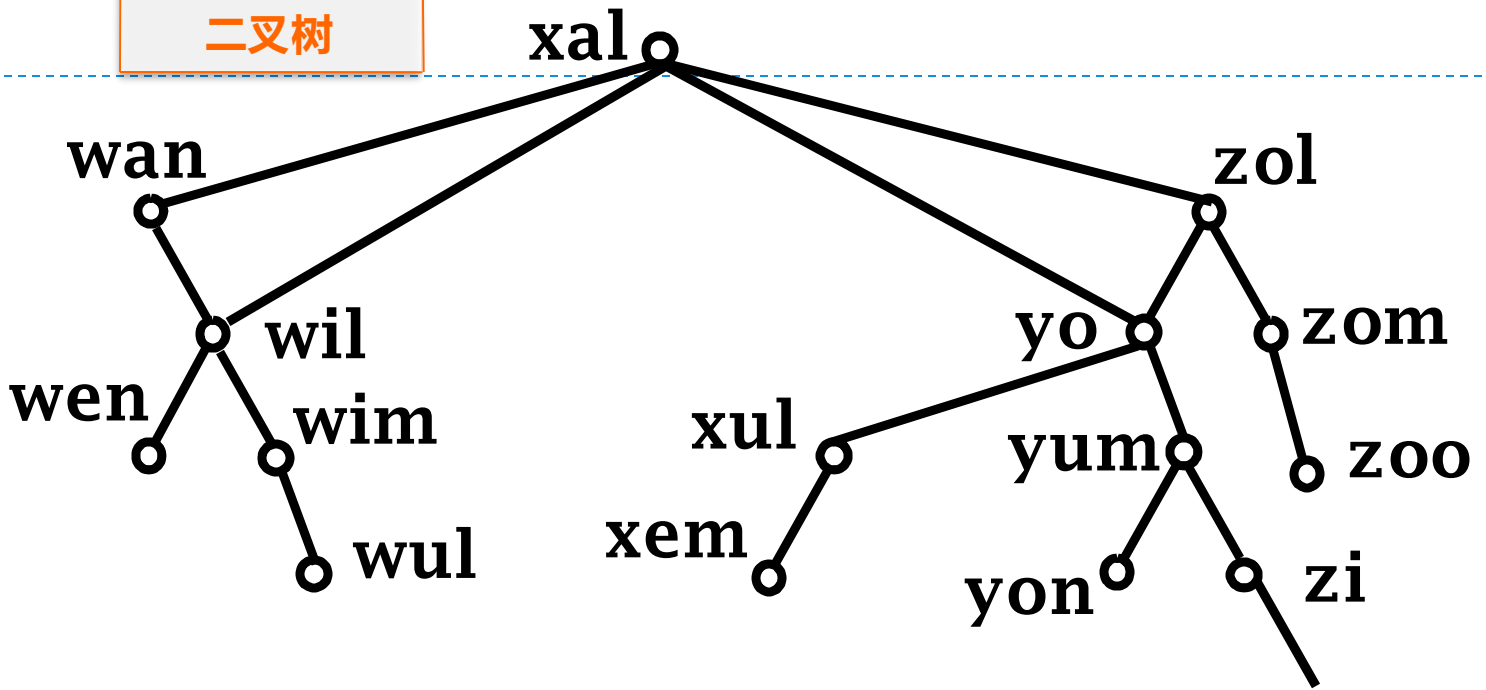


- 只需检索二个子树之一
  - 直到 K 被找到
  - 或遇上树叶仍找不到，则不存在

插入17

- 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！

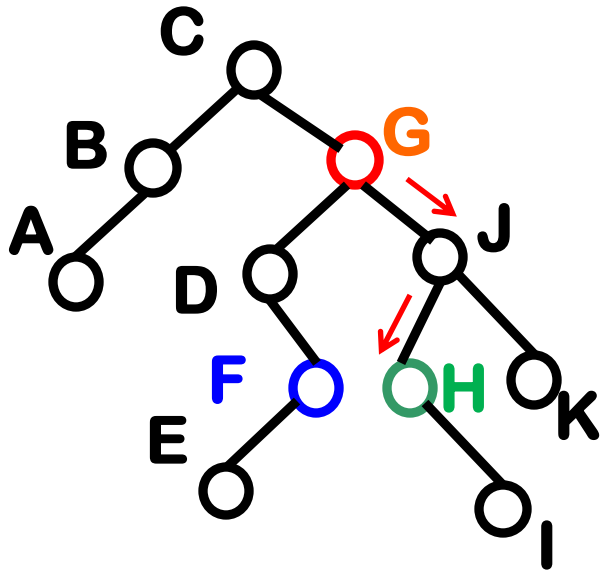




- 删除 wan
- 删除 zol

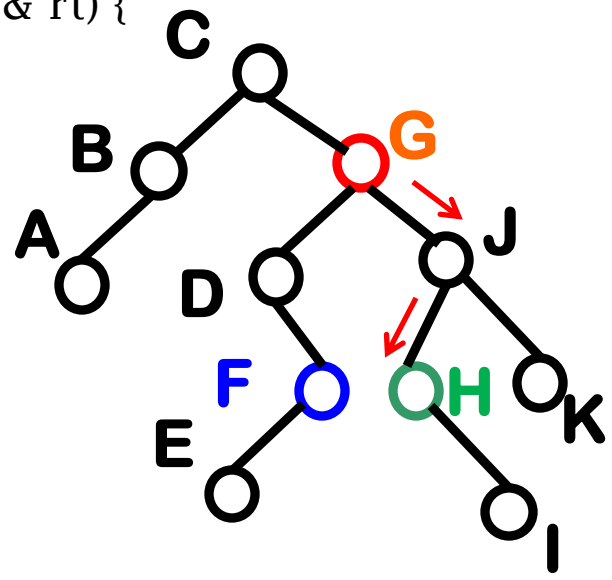
BST删除(值替换)

```
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;
  }
}
```

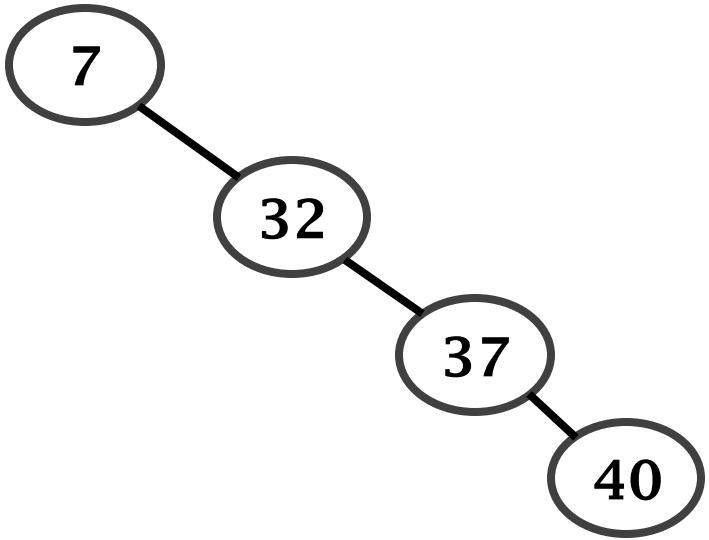


二叉搜索树总结

- 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

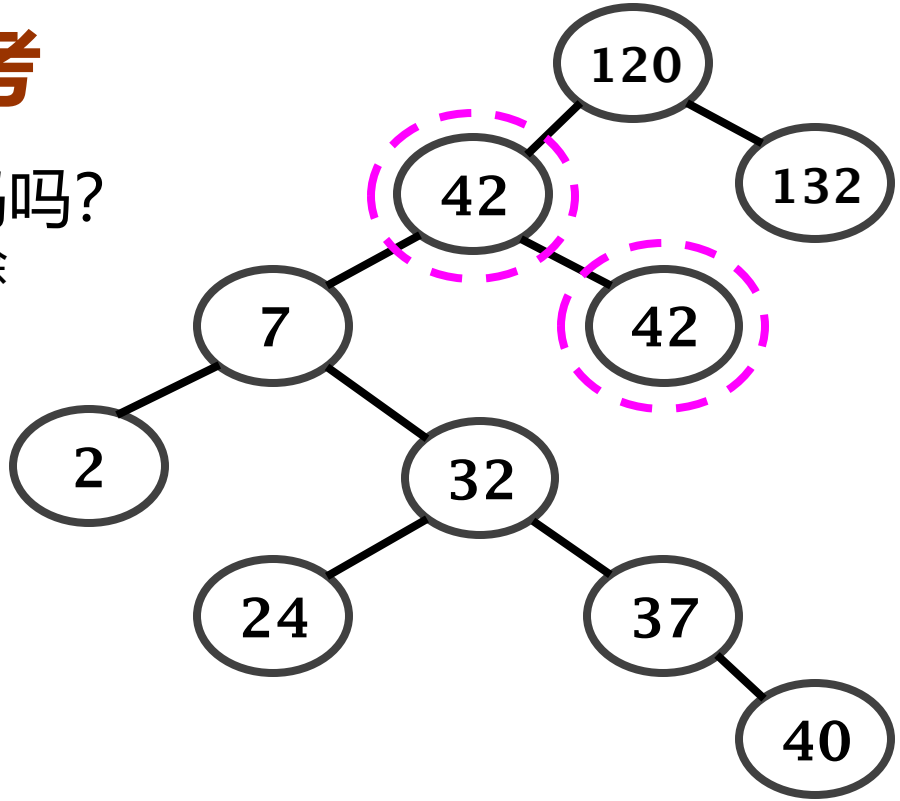
思考

- 怎样防止BST退化为线性结构?

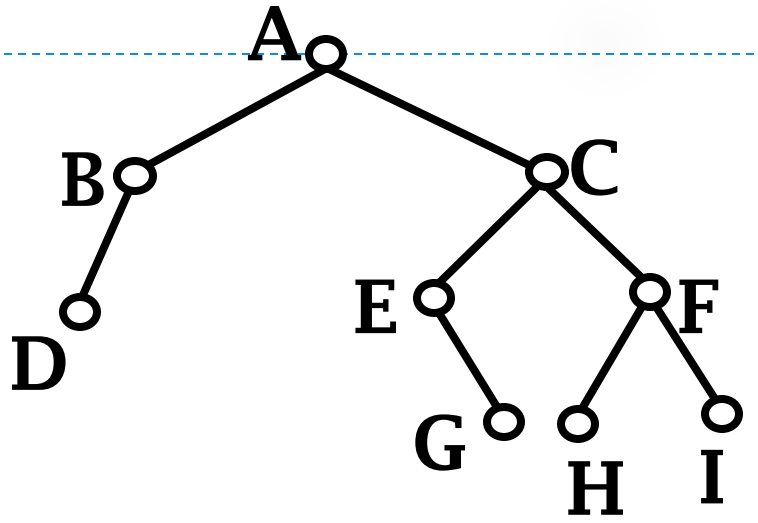


思 考

- 允许重复关键码吗?
  - 插入、检索、删除



第五章 二叉树

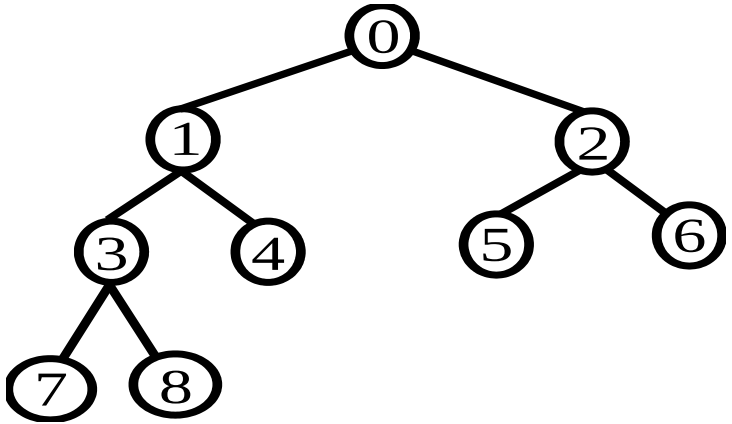


- 二叉树的概念
- 二叉树的抽象数据类型
  - 深度优先搜索
  - 宽度优先搜索
- 二叉树的存储结构
- 二叉搜索树
- 堆与优先队列
- Huffman树及其应用

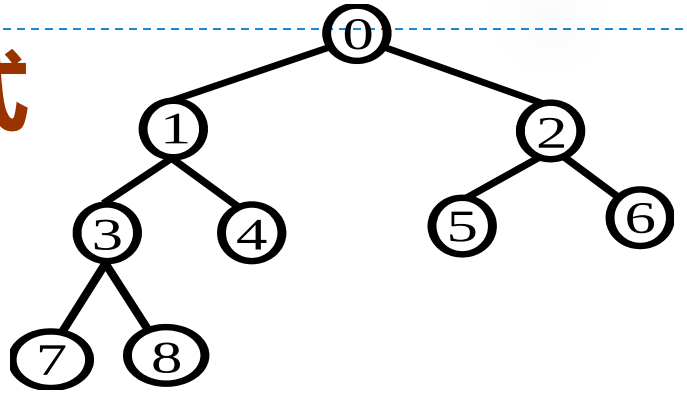
完全二叉树的顺序存储结构

- 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

- 逻辑结构上它仍然是二叉树形结构



完全二叉树的下标公式

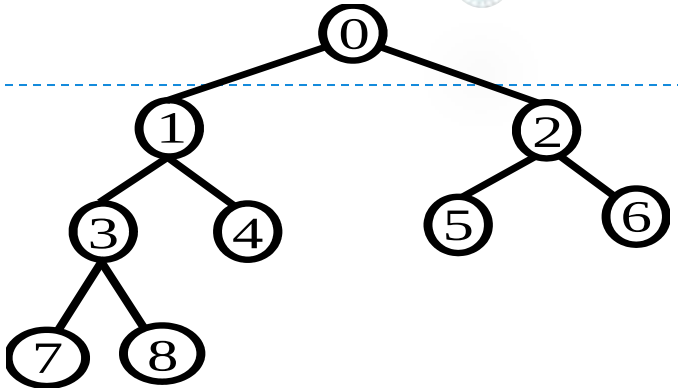


- 从结点的编号就可以推知

其父母、孩子、兄弟的编号

- 当 2i+1<n 时，结点 i 的左孩子是结点 2i+1，
  否则结点i没有左孩子
- 当 2i+2<n 时，结点 i 的右孩子是结点 2i+2，
  否则结点i没有右孩子

完全二叉树的下标公式



- 当 0<i<n 时，结点 i 的父亲是结点 ⌊(i-1)/2⌋
- 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 没有左兄弟
- 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没有右兄弟

思考

- 用三叉链的存储形式修改二叉树的相应算法。特别注意插入和删除结点，维护父指针信息。
- 完全三叉树的下标公式?

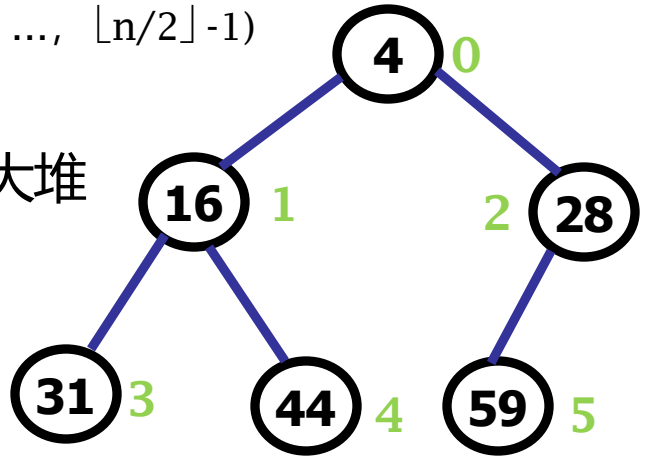
堆的定义及其实现

- 最小堆：最小堆是一个关键码序列

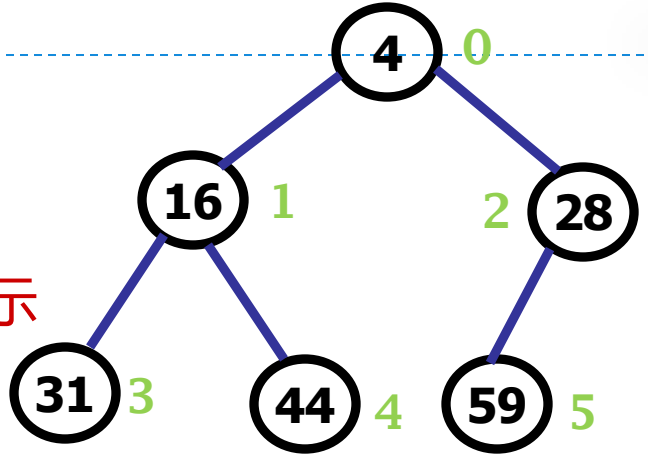
  $\{K_0, K_1, \ldots K_{n-1}\}$，它具有如下特性：

  - $K_i \le K_{2i+1}$　　$(i=0, 1, \ldots, \lfloor n/2 \rfloor -1)$
  - $K_i \le K_{2i+2}$

- 类似可以定义最大堆



堆的性质



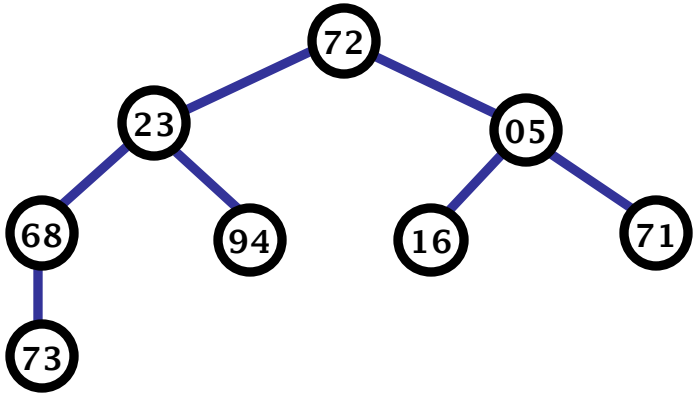
- 完全二叉树的层次序列，可以用数组表示
- 堆中储存的数是局部有序的，堆不唯一
  - 结点的值与其孩子的值之间存在限制
  - 任何一个结点与其兄弟之间都没有直接的限制
- 从逻辑角度看，堆实际上是一种树形结构

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
    Ttemp = heapArray[i];          // 保存父结点
```



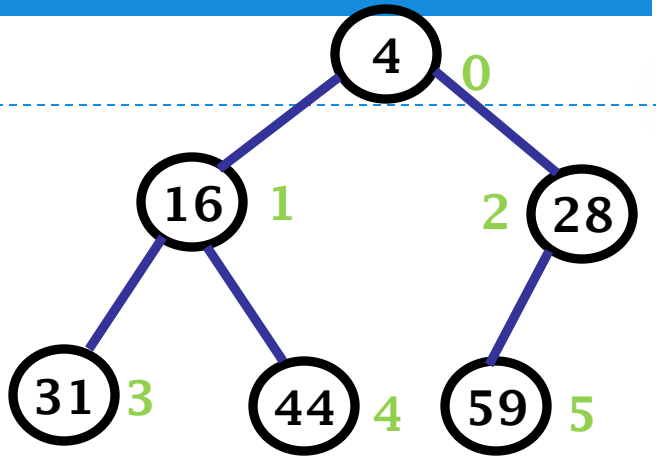
对最小堆用筛选法 SiftDown 调整

```
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]) {
        heapArray[i] = heapArray[j];
        i = j;
        j = 2*j + 1;            // 向下继续
    }
    else break;
}
heapArray[i]=temp;
}
```

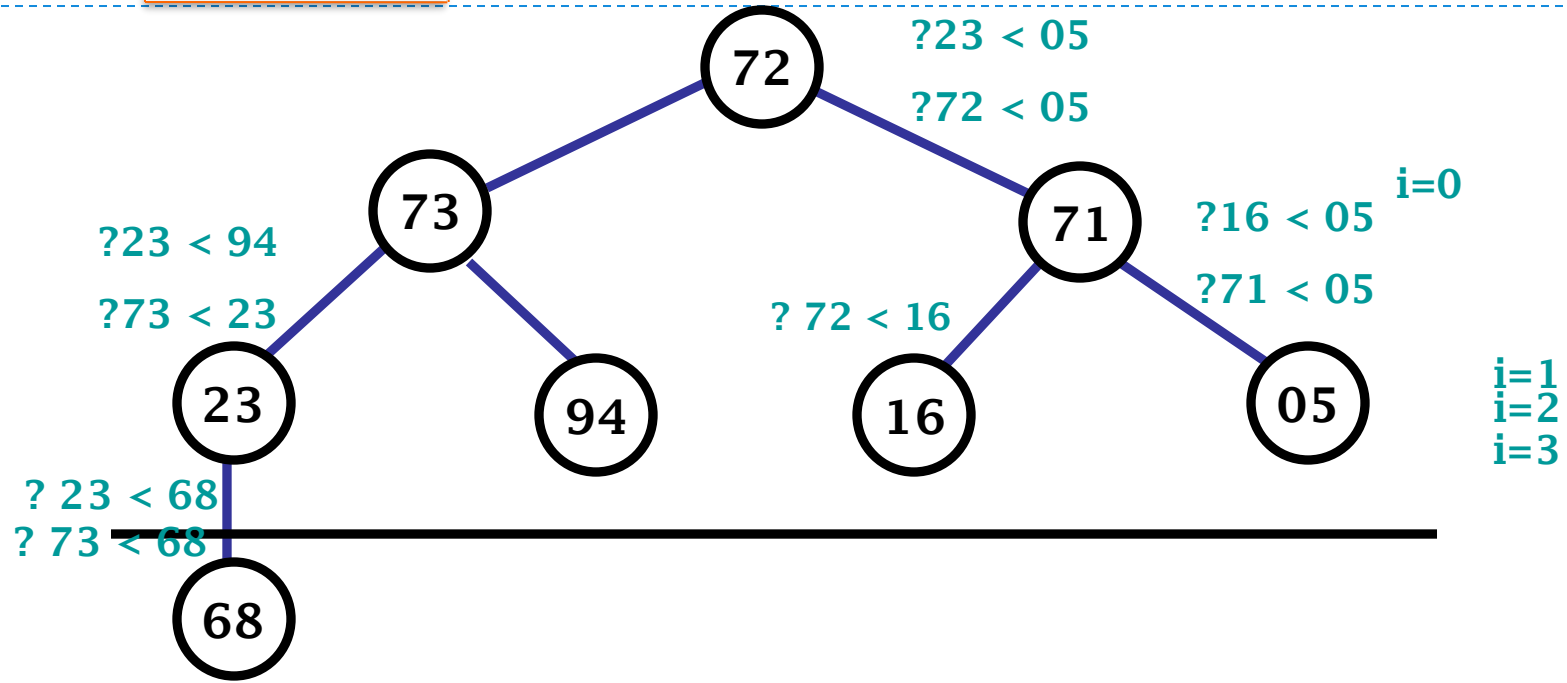
对最小堆用筛选法 SiftUp 向上调整

```
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


72
?23 < 05
?72 < 05
73
71
i=0
?23 < 94
?73 < 23
?16 < 05
?71 < 05
? 72 < 16
23
94
16
05
i=1
i=2
i=3
? 23 < 68
? 73 < 68
68

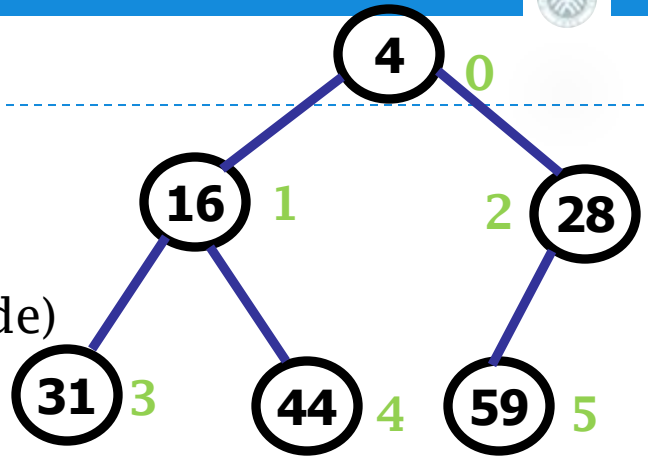
建最小堆过程示意图

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);
}
```

最小堆插入新元素

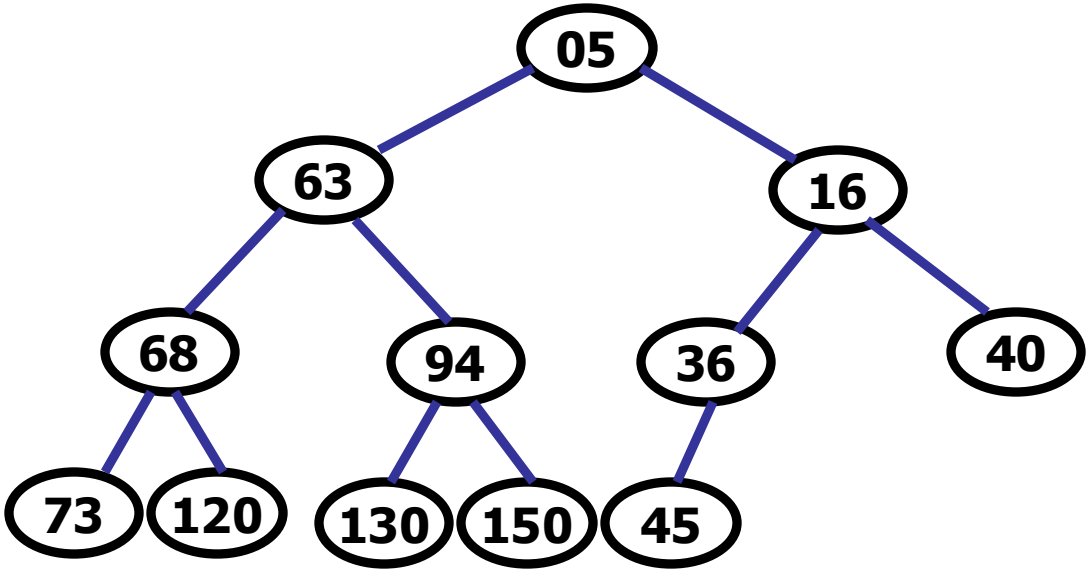



```
template <class T>
bool MinHeap<T>::Insert(const T& newNode)
//向堆中插入新元素newNode
{
    if(CurrentSize==MaxSize)      // 堆空间已经满
        return false;
    heapArray[CurrentSize]=newNode;
    SiftUp(CurrentSize);          // 向上调整
    CurrentSize++;
}
```

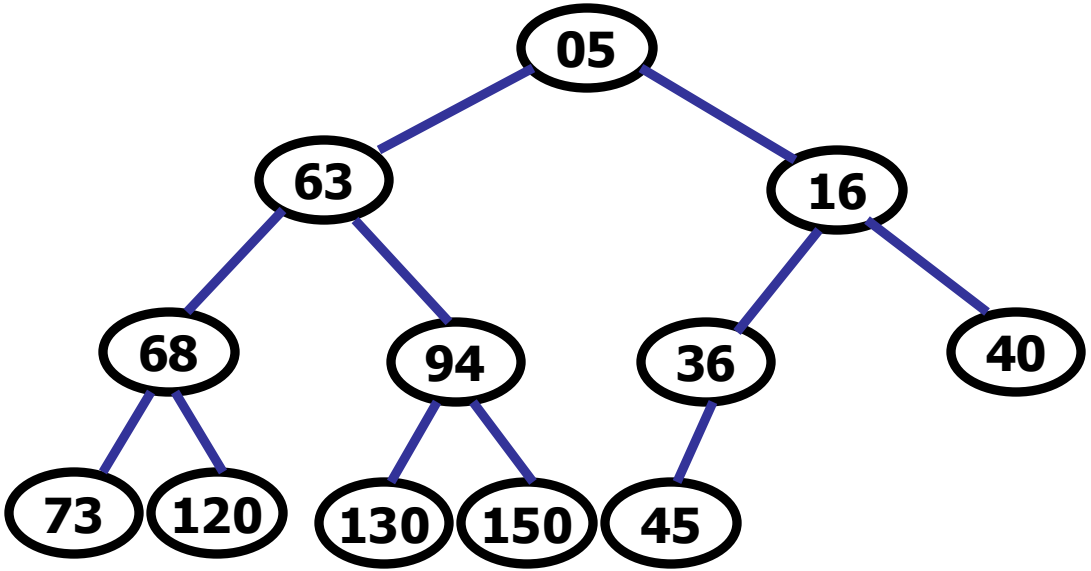
最小堆删除元素操作

```
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;
}.
```

删除68



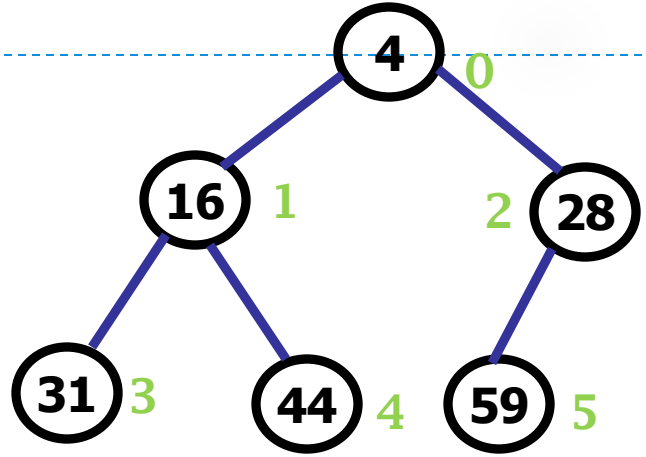
删除16



建堆效率分析

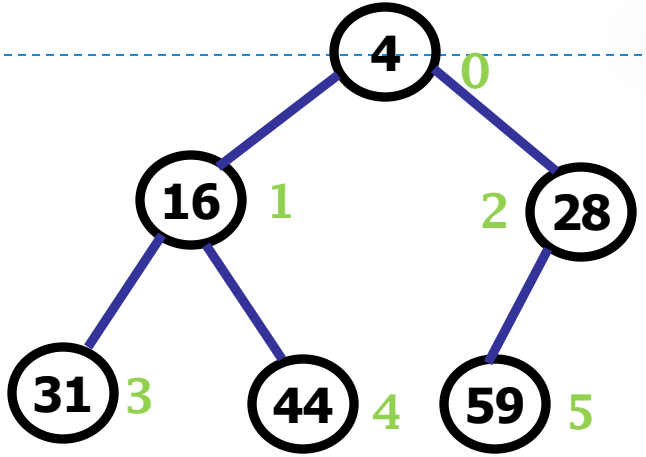
- n 个结点的堆，高度 $d = \lfloor \log_2 n + 1 \rfloor$。根为第 0 层，则第 i 层结点个数为 2^i，
- 考虑一个元素在堆中向下移动的距离。
  - 大约一半的结点深度为 d-1，不移动（叶）。
  - 四分之一的结点深度为 d-2，而它们至多能向下移动一层。
  - 树中每向上一层，结点的数目为前一层的一半，而子树高度加一。因而元素移动的最大距离的总数为

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



最小堆操作效率

- 建堆算法时间代价为 O(n)
- 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)



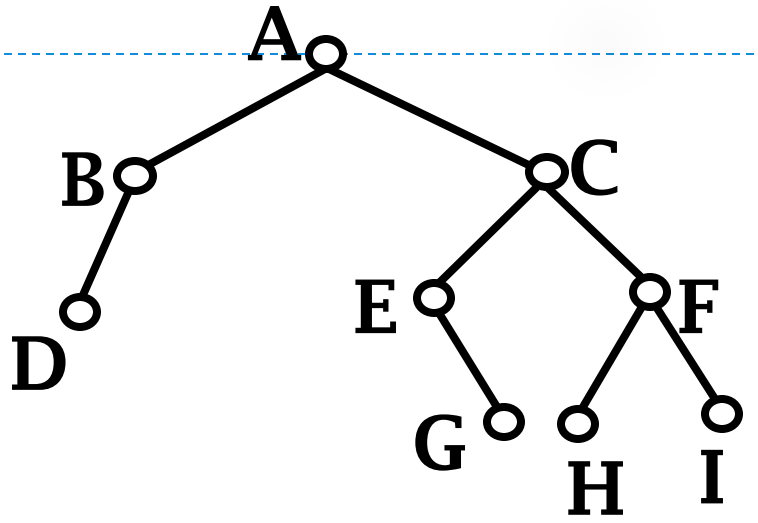
优先队列

- 堆可以用于实现优先队列
- 优先队列
  - 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

- 在向下筛选SiftDown操作时，若一旦发现逆序对，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

等长编码

- 计算机二进制编码
  - ASCII 码
  - 中文编码
- 等长编码
  - 假设所有编码都等长
    表示 n 个不同的字符需要 $\log_2 n$位
  - 字符的使用频率相等
- 空间效率

数据压缩和不等长编码

- 频率不等的字符

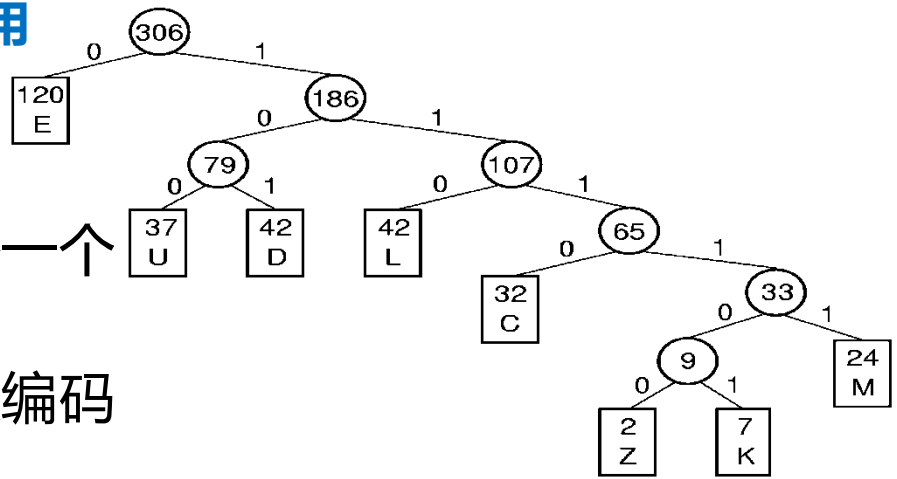
Z	K	F	C	U	D	L	E
2	7	24	32	37	42	42	120

- 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
- 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**）

5.6 Huffman树及其应用

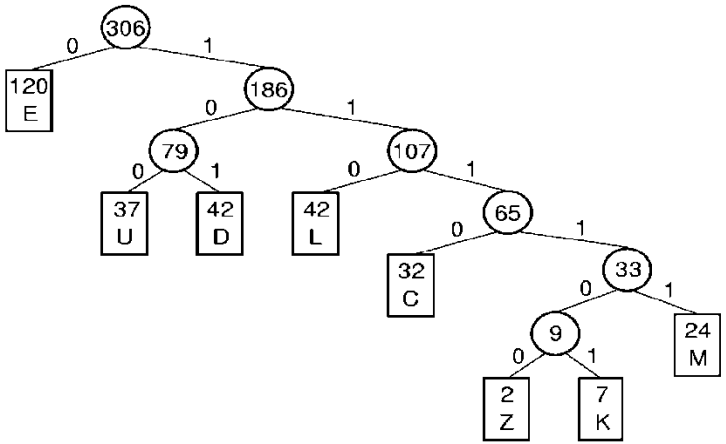
前缀编码



- 任何一个字符的编码都不是另外一个字符编码的前缀
- 这种前缀特性保证了代码串被反编码时，不会有多种可能。例如
  - 右图是一种前缀编码，对于 “000110”，可以翻译出唯一的字符串“EEEL”。
  - 若编码为Z(00), K(01), F(11), C(0), U(1), D(10), L(110), E(010)。则对应：”ZKD”,”CCCUUC”等多种可能
  - 编码 Z(111100), K(111101), F(11111), C(1110), U(100), D(101), L(110), E(0)

Huffman树与前缀编码

- 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”



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}，给出它们的前缀编码，使得总编码效率最高

- 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 有一个权 w_i 外部路径长度为 l_i
  - **这个扩充二叉树的叶结点带权外部路径长度总和**

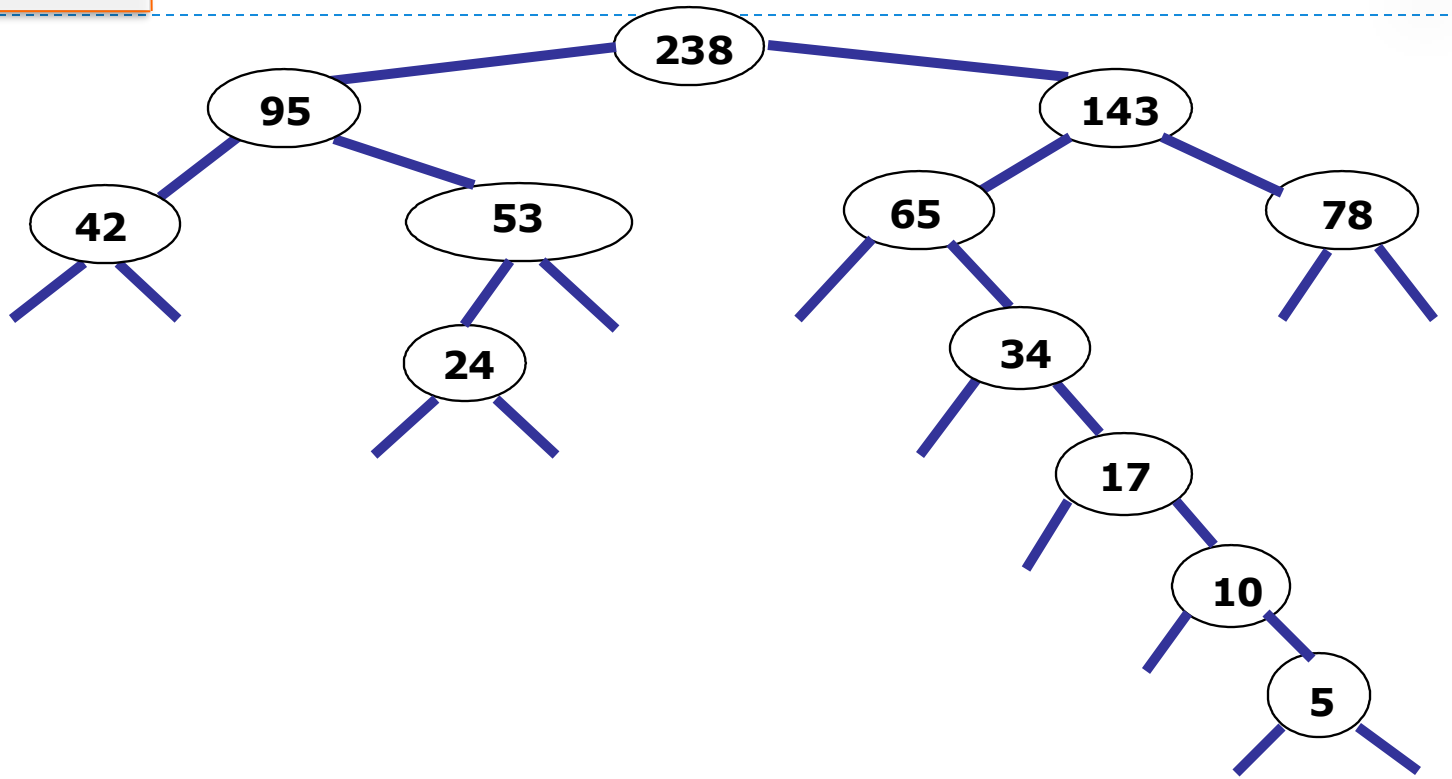
$$\sum_{i=0}^{n-1} w_i \cdot l_i$$

  - **权越大的叶结点离根越近**

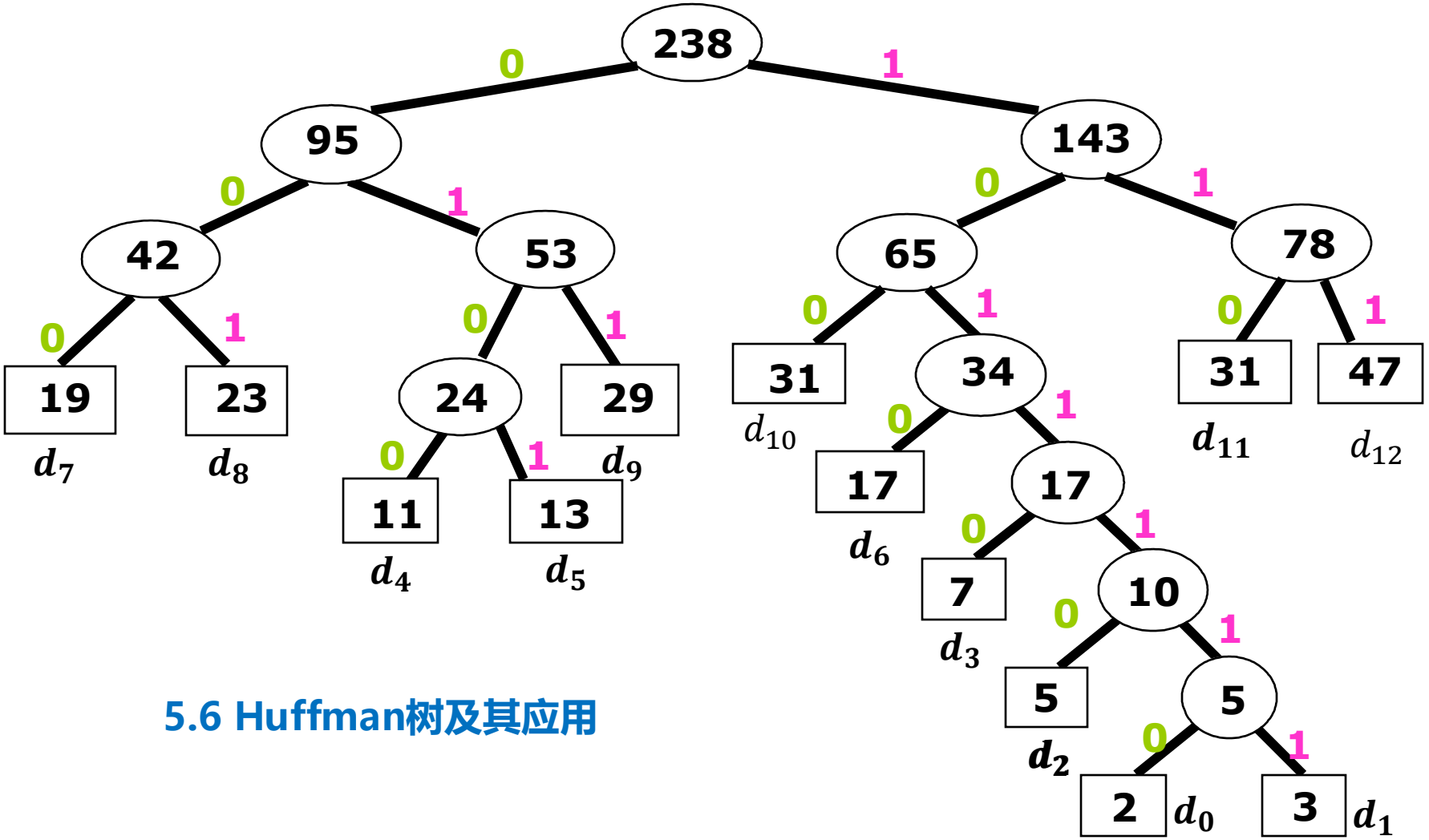
建立Huffman编码树

- 首先，按照“权”(例如频率)将字符排为一列
  - 接着，拿走**前两个字符**(“权”最小的两个字符)
  - 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孩子，而该结点的权即为两树叶的权之和
- 将所得“权”放回序列，使“权”的顺序保持
- 重复上述步骤直至序列处理完毕

5.6 Huffman树及其应用

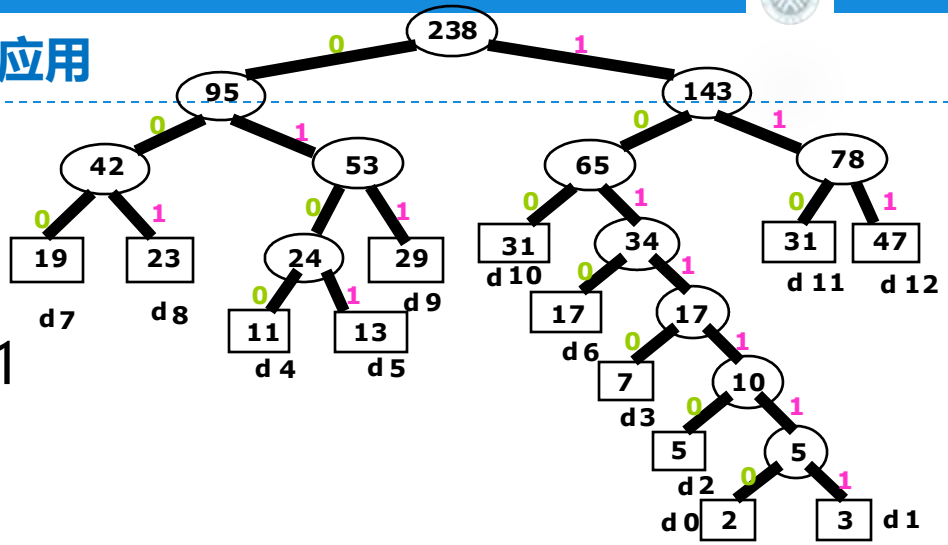


2	3	5	7	11	13	17	19	23	29	31	37	41



5.6 Huffman树及其应用

频率越大其编码越短

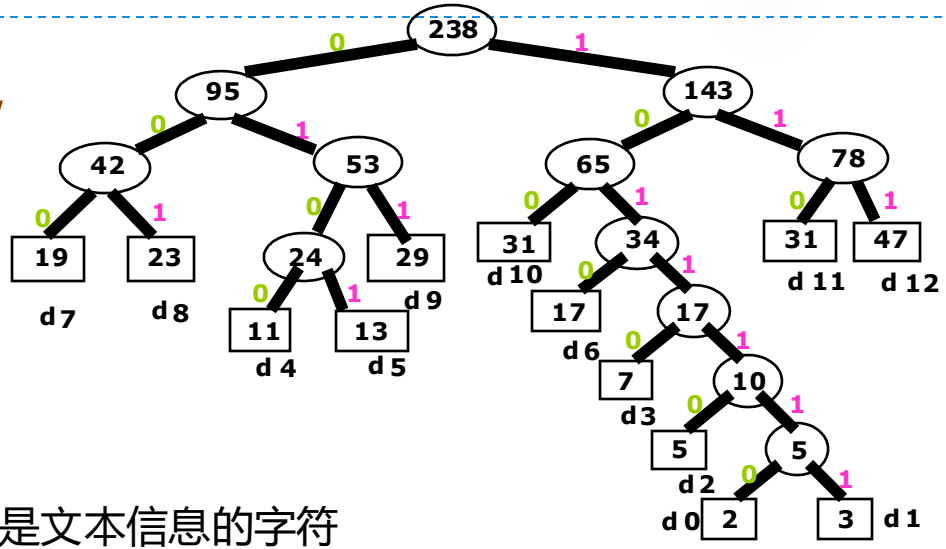


- 各字符的二进制编码为：

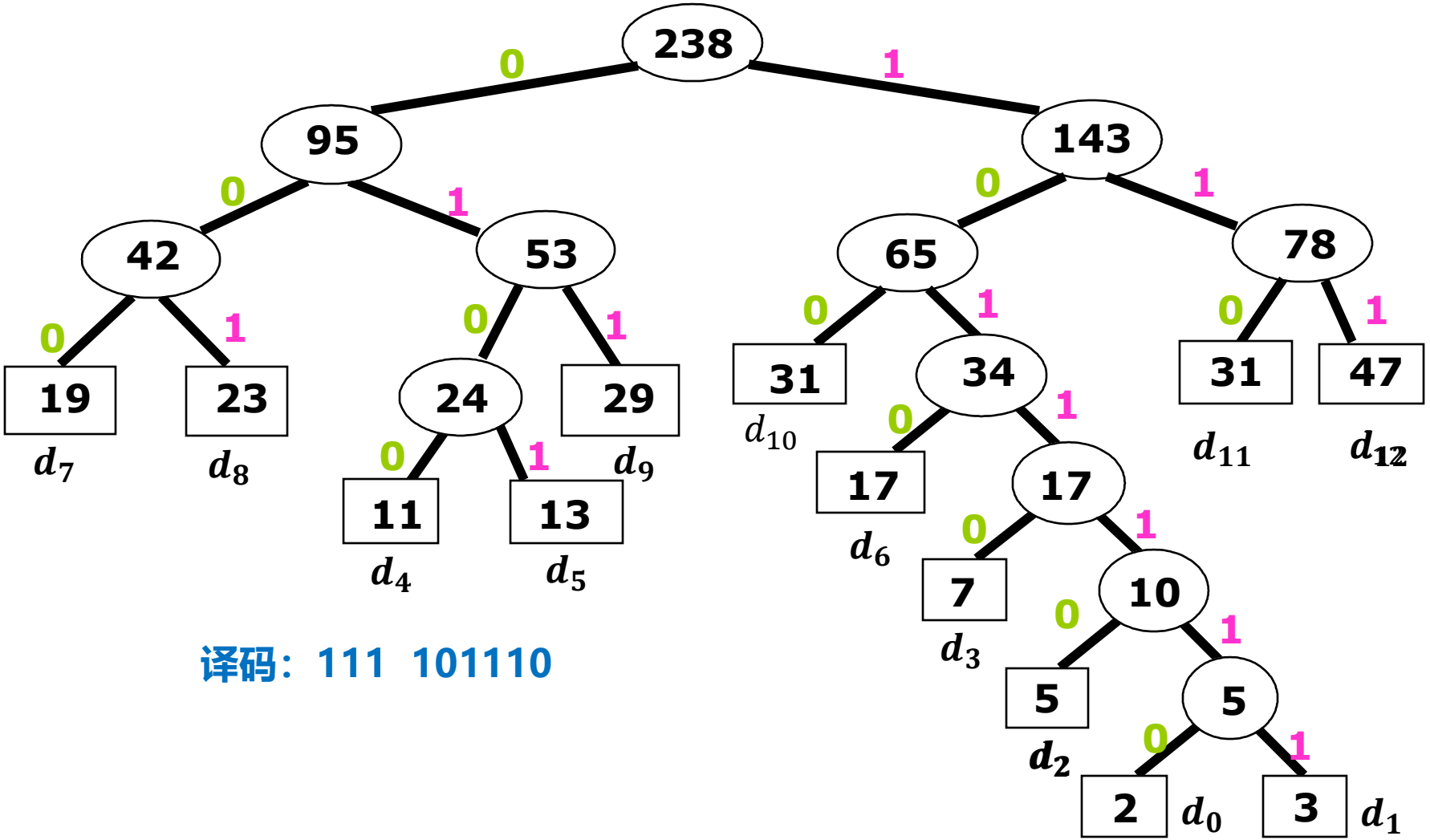
d_0 : 1011110　　d_1 : 1011111
d_2 : 101110　　d_3 : 10110
d_4 : 0100　　d_5 : 0101
d_6 : 1010　　d_7 : 000
d_8 : 001　　d_9 : 011
d_{10}: 100　　d_{11} : 110
d_{12}: 111

5.6 Huffman树及其应用

译码：　从左至右逐位判别代码串，直至确定一个字符



- 与编码过程相逆
  - 从树的根结点开始
    - “0”下降到左分支
    - “1”下降到右分支
    - 到达一个树叶结点，对应的字符就是文本信息的字符
- 连续译码
  - 译出了一个字符，再回到树根，从二进制位串中的下一位开始继续译码



译码：111 101110

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
}
```

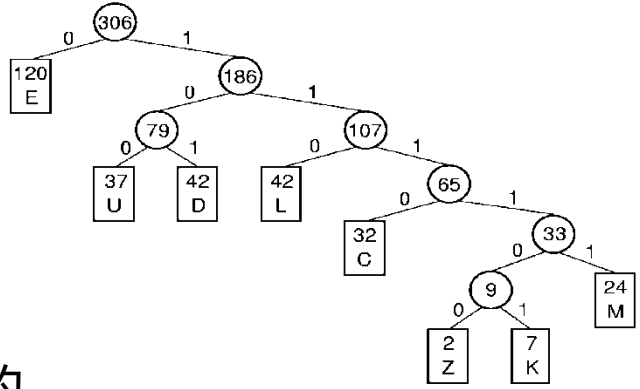
Huffman树的构造

```
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++) {
        NodeList[i].element =weight[i];
        NodeList[i].parent = NodeList[i].left
                                = NodeList[i].right = NULL;
        heap.Insert(NodeList[i]);                    //向堆中添加元素
    } //end for
```

Huffman树的构造

```
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); //合并权值最小的两棵树
      heap.Insert(*parent);                           //把parent插入到堆中去
      root=parent;                                    //建立根结点
     }//end for
delete []NodeList;
}
```

Huffman方法的正确性证明



- 是否前缀编码?
- 贪心法的一个例子
  - Huffman树建立的每一步，“权”最小的两个子树被结合为一新子树
- 是否最优解?

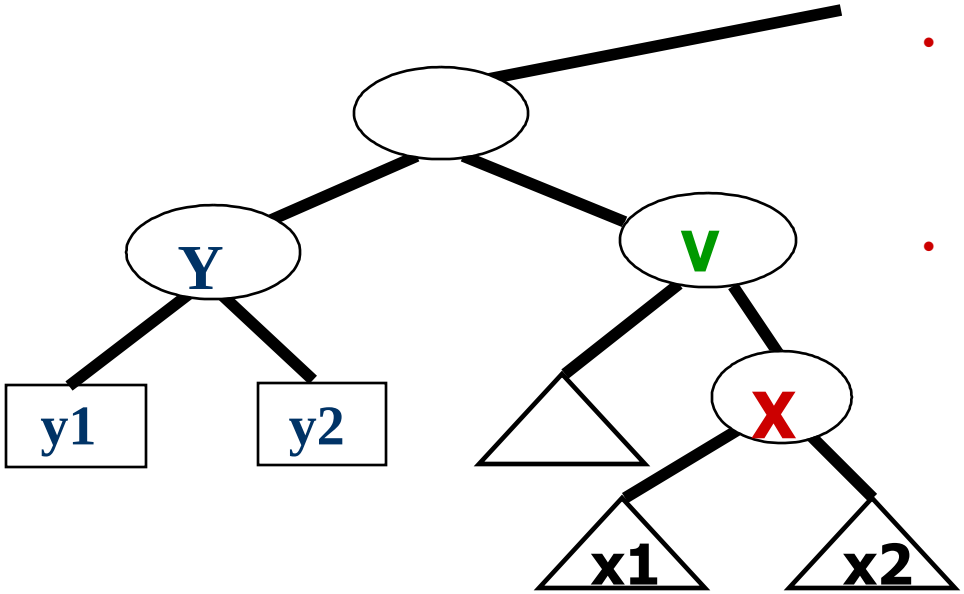
Huffman性质

- 引理

含有两个以上结点的一棵 Huffman 树中，字符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

5.6 Huffman树及其应用

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

5.6 Huffman树及其应用

- 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作归纳进行证明
- 初始情况：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：

假设有n-1个叶结点的由函数HuffmanTree产生的Huffman树有最小外部路径权重

5.6 Huffman树及其应用

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。根据引理，它们已经是树 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
- 把 V′展开为V（$w_0 + w_1$），T′还原为 T，

  则 T 也应该有最小的外部路径长度
- 因此，根据归纳原理，定理成立

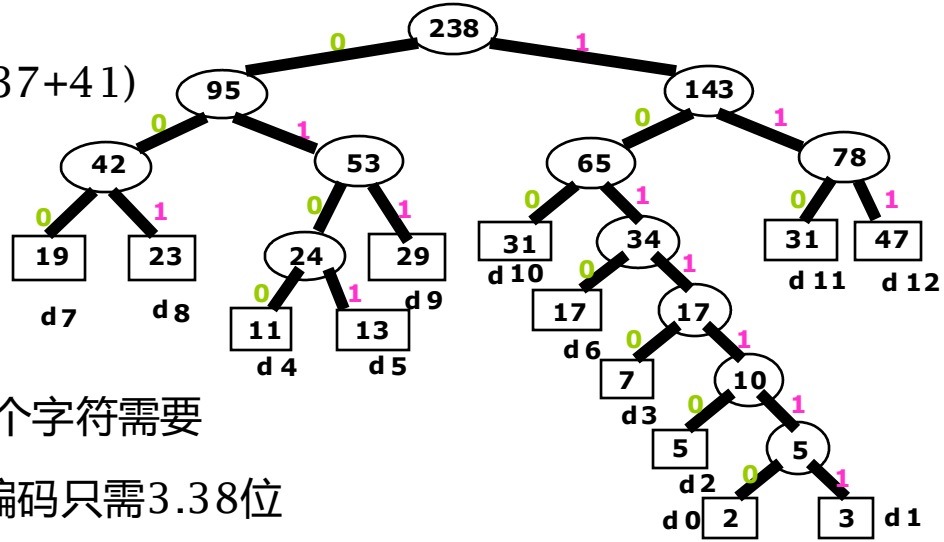
Huffman树编码效率

- 估计Huffman编码所节省的空间
  - 平均每个字符的代码长度等于每个代码的长度 c_i 乘以其出现的概率 p_i ，即:
  - $c_0p_0 + c_1p_1 + \ldots + c_{n-1}p_{n-1}$
    或 $(c_0f_0 + c_1f_1 + \ldots + c_{n-1}f_{n-1}) / f_T$

这里f_i为第i个字符的出现频率，而f_T为所有字符出现的总次数

Huffman树编码效率（续）

- 图中，平均代码长度为：
  (3*(19+23+24+29+31+34+37+41)
  + 4*(11+13+17)
  + 5*7
  + 6*5
  + 7*(2+3)) / 238
  = 804/238 ≈ 3.38



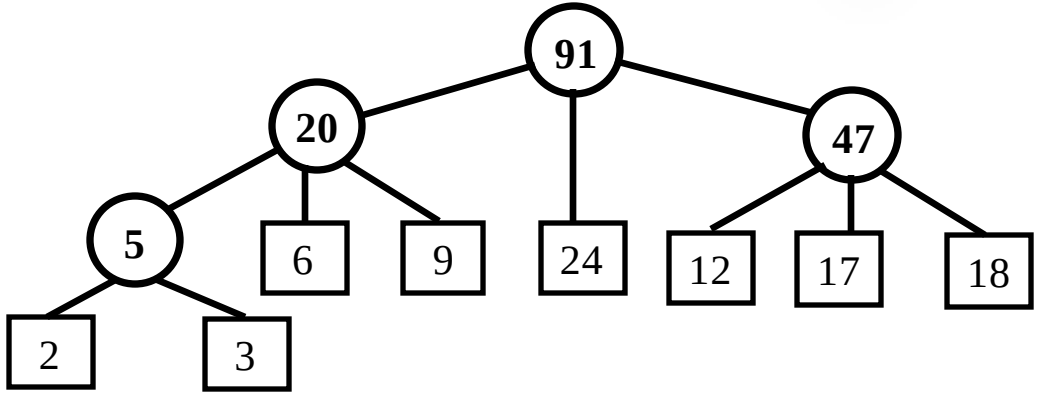
- 对于这13个字符，等长编码每个字符需要
  ⌈log 13⌉ = 4位，而Huffman编码只需3.38位
- Huffman编码预计只需要等长编码3.38/4≈ 84% 的空间

Huffman树的应用

- Huffman编码适合于
  字符 **频率不等，差别较大的情况**
- 数据通信的二进制编码
  - 不同的频率分布，会有不同的压缩比率
  - 大多数的商业压缩程序都是采用几种编码方式以应付各种类型的文件
    - Zip 压缩就是 LZ77 与 Huffman 结合
- 归并法外排序，合并顺串

5.6 Huffman树及其应用

思考



- 当外部的数目不能构成满 b叉 Huffman 树时，需附加多少个权为 0的“虚”结点？请推导
- R 个外部结点，b叉树
  - 若（r-1）% (b-1)==0，则不需要加“虚”结点
  - 否则需要附加 b -（r-1）% (b-1) - 1个“虚” 结点
    - 即第一次选取（r-1）% (b-1) + 1个非0权值
- 试调研常见压缩软件所使用的编码方式

数据结构与算法

感谢倾听

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