数据结构与算法

第 5 章 二叉树

主讲：赵海燕

北京大学信息科学技术学院

“数据结构与算法”教学组

国家精品课“数据结构与算法”

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

张铭，王腾蛟，赵海燕

高等教育出版社，2008. 6，“十一五”国家级规划教材

内容提要

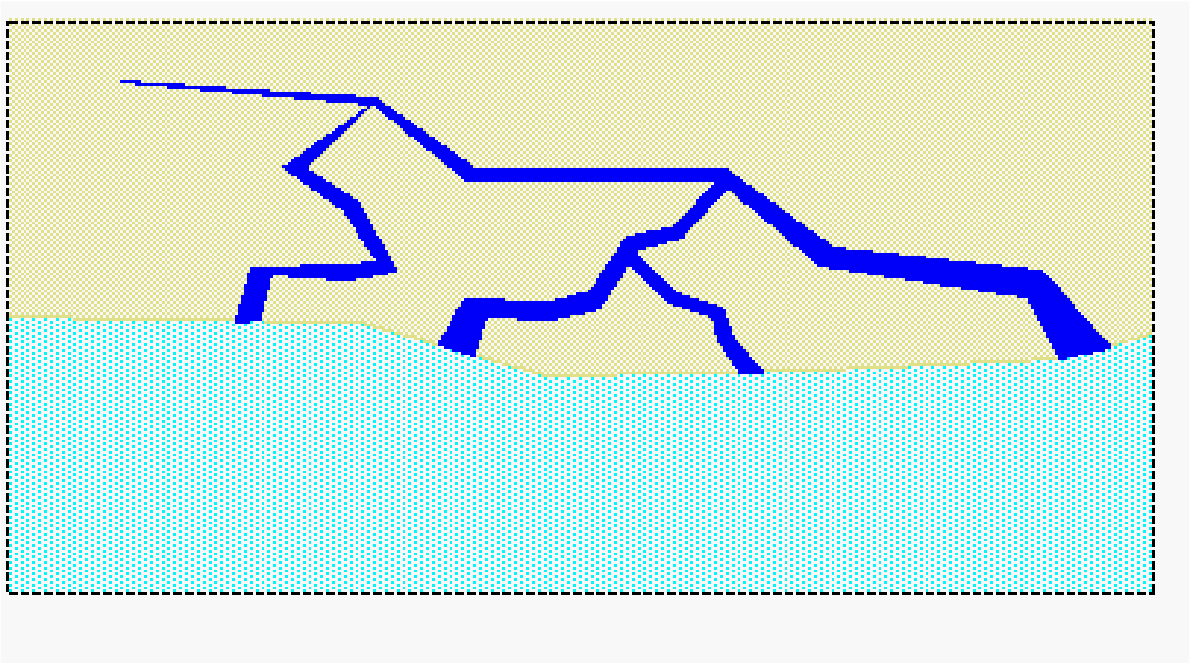
- 二叉树的概念及主要性质
  - 二叉树的抽象数据类型
  - 二叉树的存储结构
- 二叉树的应用
  - 二叉搜索树
  - 堆与优先队列
  - Huffman树及其应用
- 二叉树知识点总结

线性结构 vs. 非线性结构

- **线性结构**
  - **基本特点**：元素之间满足线性关系，每个内部结点（元素）都有且仅有一个前驱结点、一个后继结点
- **非线性结构**
  - **至少存在**一个数据元素，具有两个或两个以上的前驱或后继
    - 树结构（前驱唯一，后继不限）
    - 图结构（前驱、后继均不限）

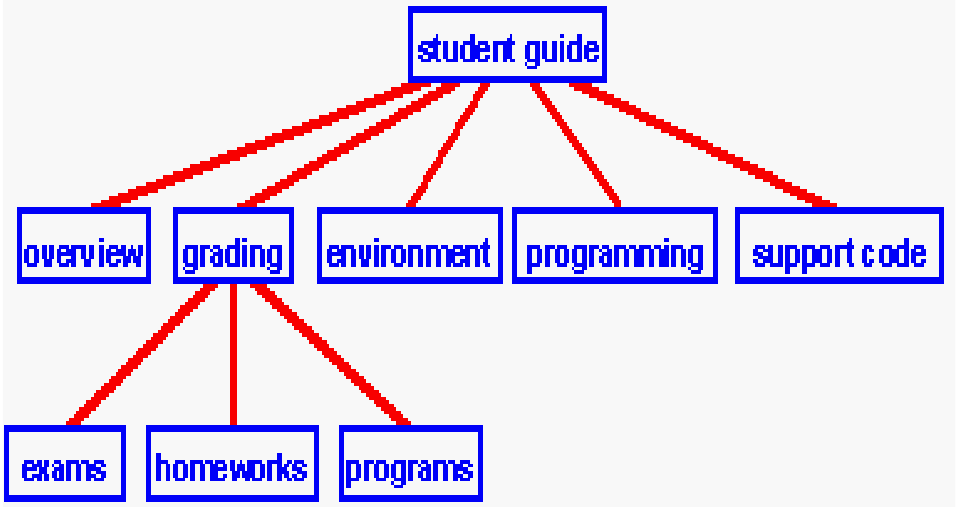
树结构

- 应用广泛的非线性结构
- 一个数据元素可有**零到多个**直接后继，与线性结构相比

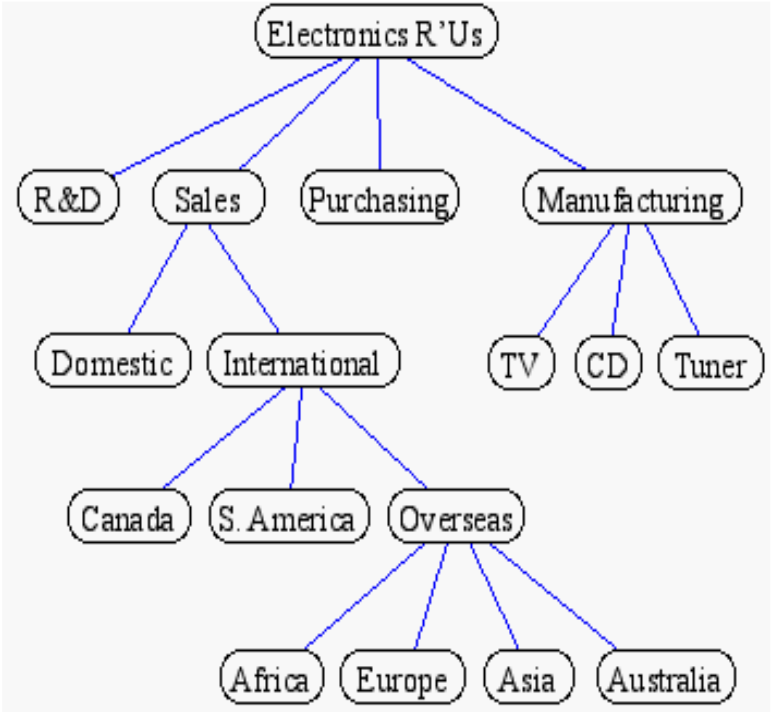


树结构

- 一棵树表示一个层次结构

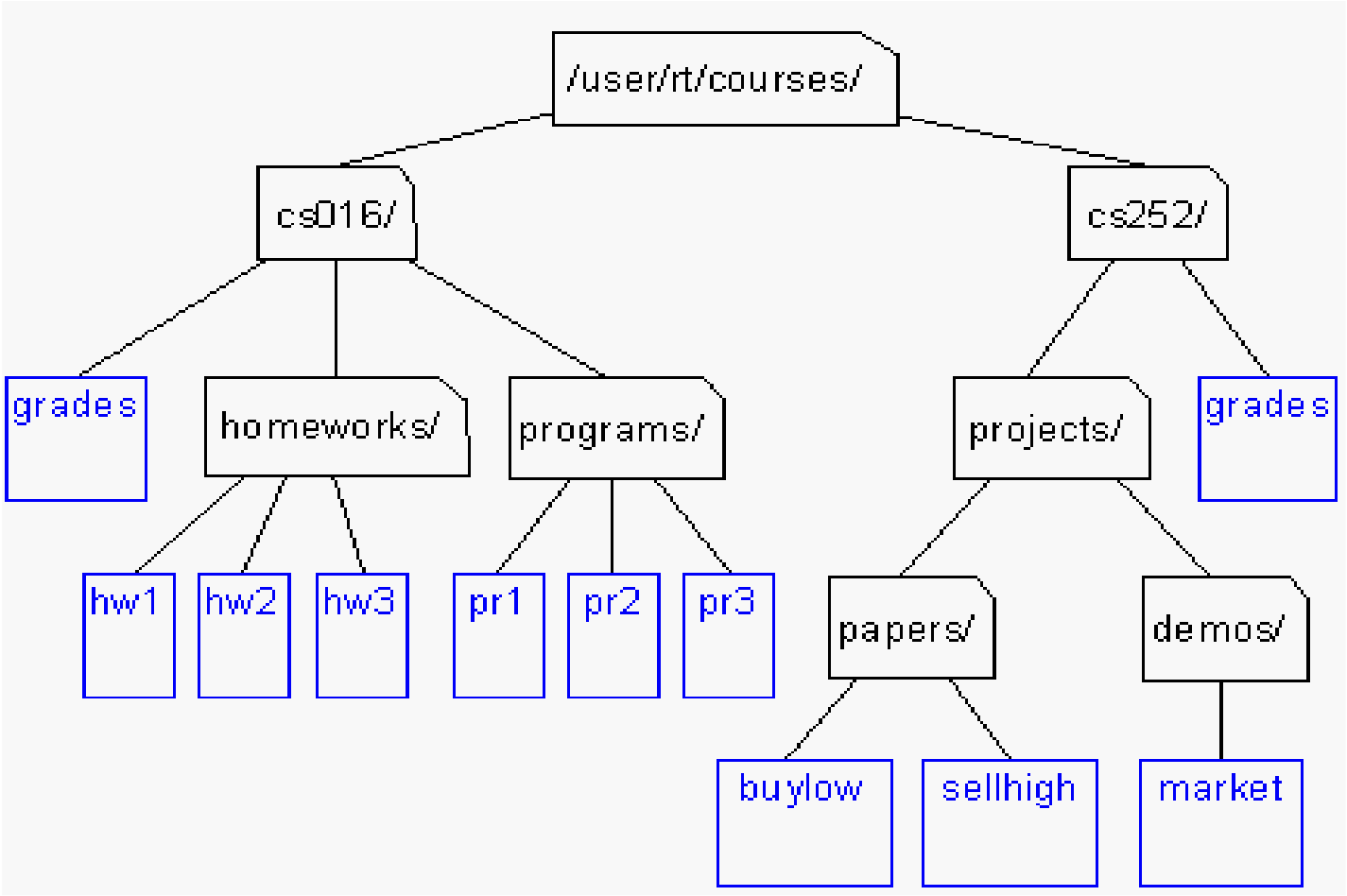


Ex 1: *table of contents of a book*



Ex 2: *organization structure of a corporation*

树结构


/user/rt/courses/
cs016/
cs252/
grades
homeworks/
programs/
projects/
grades
hw1
hw2
hw3
pr1
pr2
pr3
papers/
demos/
buylow
sellhigh
market

树的逻辑结构

- 包含n个结点的有穷集合K (n>0)，且在K上定义了一个满足以下条件关系 r：
  - 有且仅有一个结点 $k_0 \in K$，对于关系 r 来讲没有前驱：结点 k_0 称做树根(root)；
  - 除结点 k_0 外，K中所有结点对于关系 r 来说都 有且仅有一个前驱；
  - 除结点 k_0 外的任何结点 $k \in K$，都存在一个结点序列 $k_0, k_1, \ldots, k_s$，使得 k_0 就是树根，且 $k_s = k$，有序对$<k_{i-1}, k_i> \in r$ $(1 \le i \le s)$；该结点序列称为从根到结点k的一条路径（path）

相关概念和术语

- **结点**(node)
  - **根结点**(root)、 **父结点**(parent)、**子结点**(children) 、**祖先**(ancestor)、**后代**(descendant)
    - 若<k, k′>∈r，则称 k 是k′的父结点，k′是k的 子结点
  - **兄弟**（sibling)
    - 若有序对<k, k′>及<k, k″>∈r，则k′和k″互称为兄弟
  - **分支结点**(internal node)、**叶结点**(leaf/external node)
    - 没有子树的结点称作叶结点或终端结点 /外部结点
    - 分支结点（branch node）也称为非终端结点/内部结点（internal node)

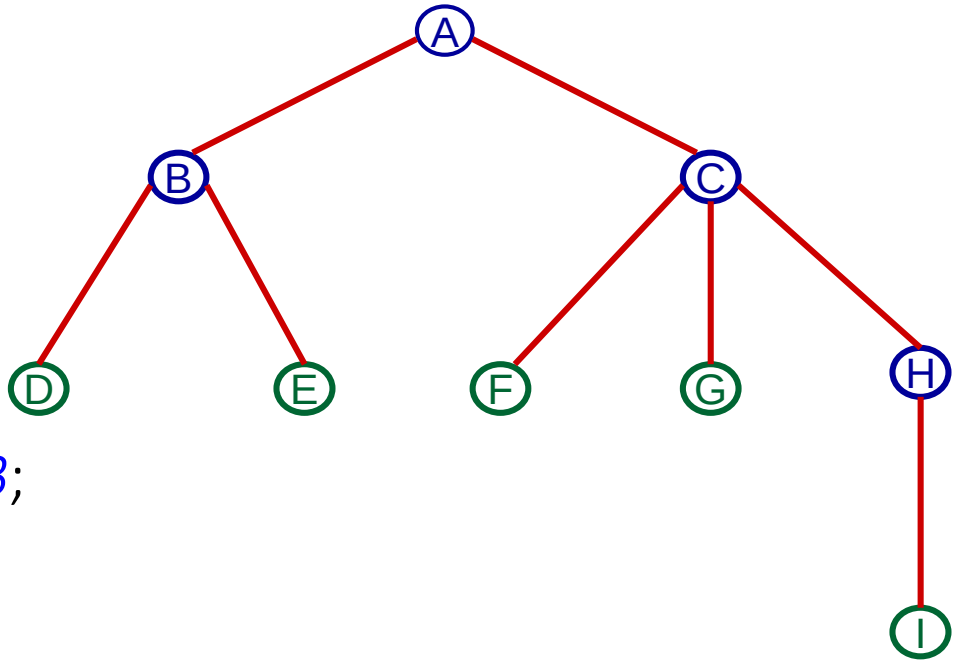
相关概念和术语

- **边** (edge)
  - 两个结点组成的有序对 <k，k′>
  - 路径(path) 及 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 $k_0, k_1, \ldots, k_s$，称为从根k_0 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

相关概念和术语

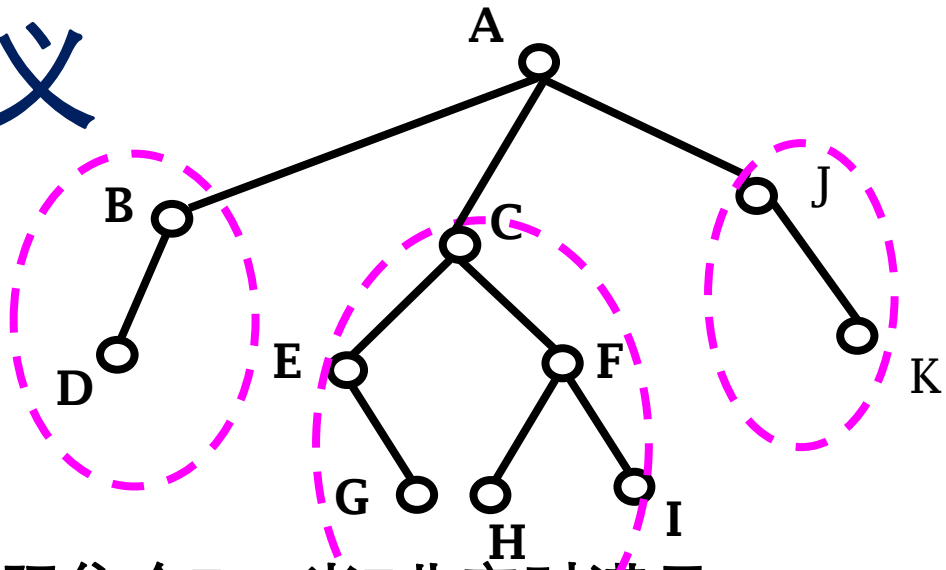
- 其他
  - **度数**(degree）、**层数**（levels）、**深度**(depth)、**高度**(height)
  - 一个结点所拥有的子树的个数为其**度数**
  - 根结点的**层数**为0，其它任何结点的层数为其父结点结点的层数加1
  - 树的**深度**：最大层数
  - 树的**高度**：深度+1

术语示例



- *A* is the *root* node;
- *B* is the *parent* of D and E;
- *C* is *sibling* of *B*;
- *D* and *E* are the *children* of *B*;
- *D,E,F,G* and *I* are *external nodes*, or *leaves*;
- *A,B,C,H* are *internal nodes*;
- The *degree* of node *C* is *3*;
- The *level* (*depth*) of node *E* is *2*;
- The *depth* of the tree is *3*
- The *height* of the tree is *?*

树结构的递归定义



- **树是包含 n (n≥0) 个结点的有限集合T，当T非空时满足：**
  - 有且仅有一个特别标出称作根的结点（r）；
  - 除根以外其他结点被分成m个(m≥0)不相交的集合$T_1, T_2, \ldots, T_m$，其中每一个又均为树。$T_1, T_2, \ldots, T_m$称作该根 (r) 的子树

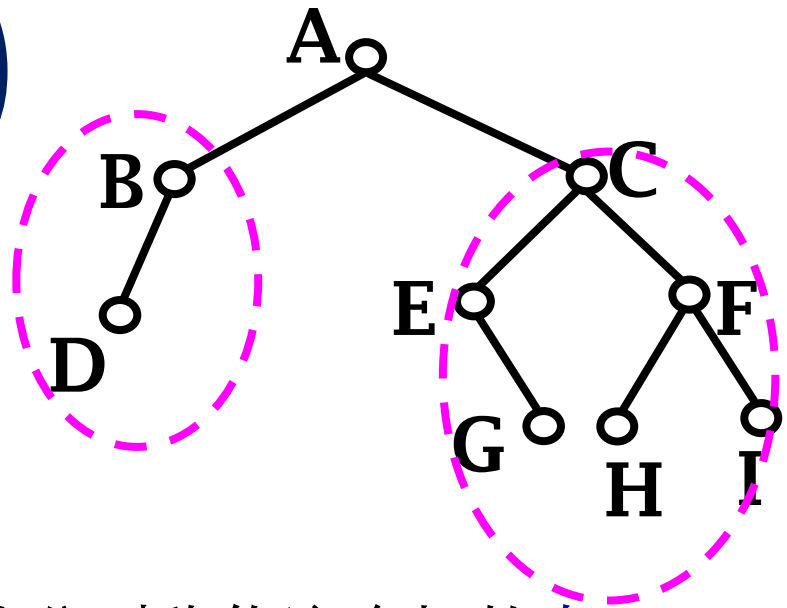
该定义是**递归的**，由其子树来构造和定义树：只包含一个结点的树必然仅由根组成，包含n>1个结点的树借助于少于n个结点的树来定义

特例：不包括任何结点的树称为空树

二叉树的概念

- 二叉树的定义及基本术语
- 满足一定特性的二叉树
  - 满二叉树
  - 完全二叉树
  - 扩充二叉树
- 二叉树的主要性质

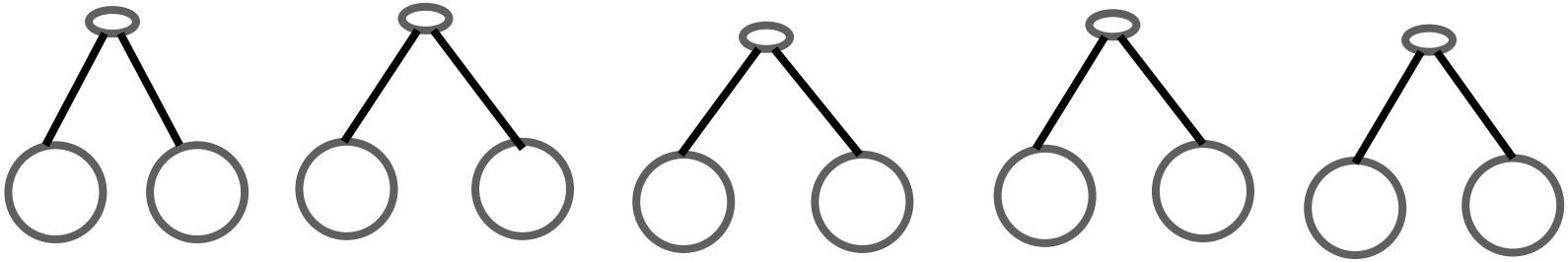
二叉树(Binary tree)



- **由结点的有限集合构成**
  - 或为空集(empty)
  - 或由一个根结点及两棵不相交的分别称作这个根的左子树 (left subtree) 和右子树 (right subtree) 的二叉树 (也是结点的集合) 组成

- 递归版本
  - 二叉树可以是空集合，根可有空的左子树或右子树，或者左右子树皆为空。也即，每个结点至多有两棵子树，且其子树有左右之分，不能随意交换

二叉树的基本形态



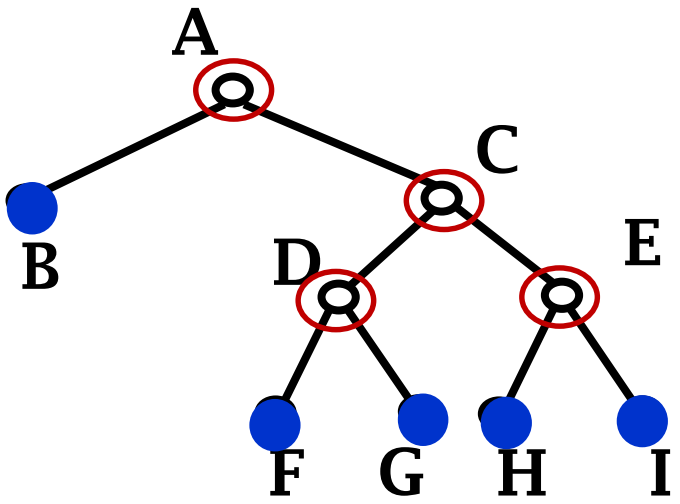
(a)空　(b)独根　(c) 右空　(d) 左空　(e)左右都不空

二叉树 vs 树

- 二叉树不是树的特殊情形：两种数据结构
- 树和二叉树之间的主要区别
  - 二叉树中结点的子树要区分为**左子树**和**右子树**，即使在结点只有一棵子树的情况下也要**明确指出**该子树是左子树还是右子树
  - (c)和(d) 是两棵不同的二叉树，但作为树则相同
- 二叉树的基本概念和术语与树中的相应部分类似

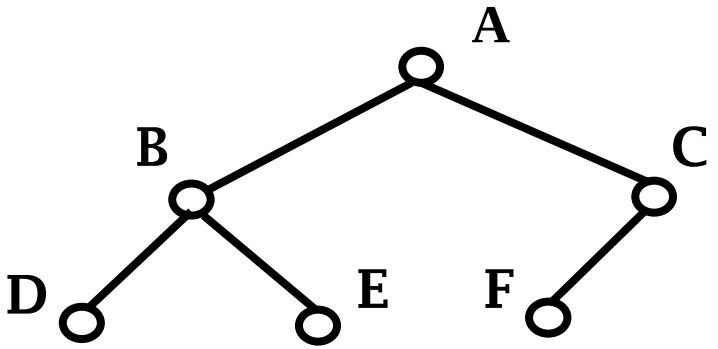
满二叉树（Full Binary Tree）

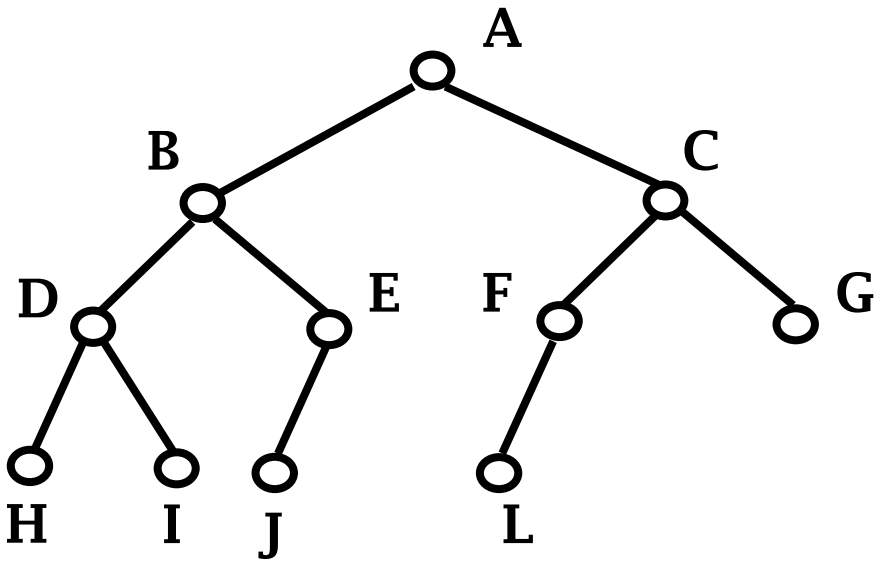
- 一棵二叉树的任何结点，或为**叶结点**，或恰有两棵非空子树



完全二叉树 (Complete Binary Tree)

- 一棵二叉树
  - 最多只有最下面的两层结点度数可以小于 2
  - 最下面一层的结点都集中在该层最左边的若干位置上
- 许多算法和算法分析中都或明显或隐含地用到完全二叉树的概念





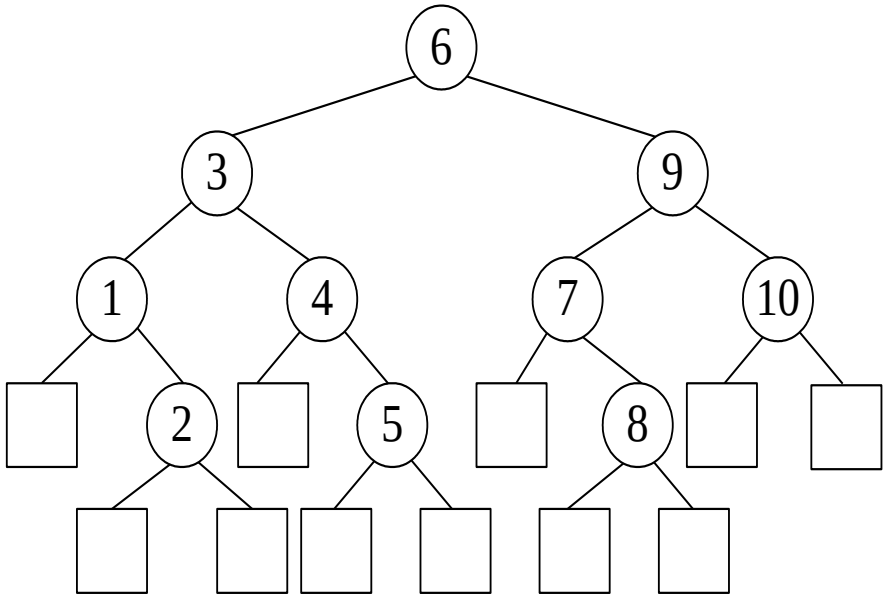
完全二叉树一定是满二叉树？

完全二叉树的特点

- 叶结点只可能在层次最大的两层出现
- 由根结点到各个结点的**路径长度总和**在具有同样结点个数的二叉树中达到了最小
  - 即，任意一棵二叉树中根结点到各结点的最长路径一定不短于结点数目相同的完全二叉树中的路径长度

扩充二叉树（Extended Binary Tree）

- **当二叉树里出现空的子树时，就增加新的、特殊的结点：空叶结点**
  - 对于原来二叉树里度数为 1的分支结点，其下增加一个空叶结点
  - 对于原来二叉树的叶结点，其下增加两个空叶结点
  - 扩充的二叉树是满二叉树，新增加的空叶结点（外部结点）的个数等于原来二叉树的结点（内部结点）个数加1



扩充二叉树

- 外部路径长度E
  - 从扩充的二叉树的根到每个外部结点的路径长度之和
- 内部路径长度I
  - 扩充的二叉树里从根到每个内部结点的路径长度之和

$$E = I+2n$$

其中，n 是内部结点的个数

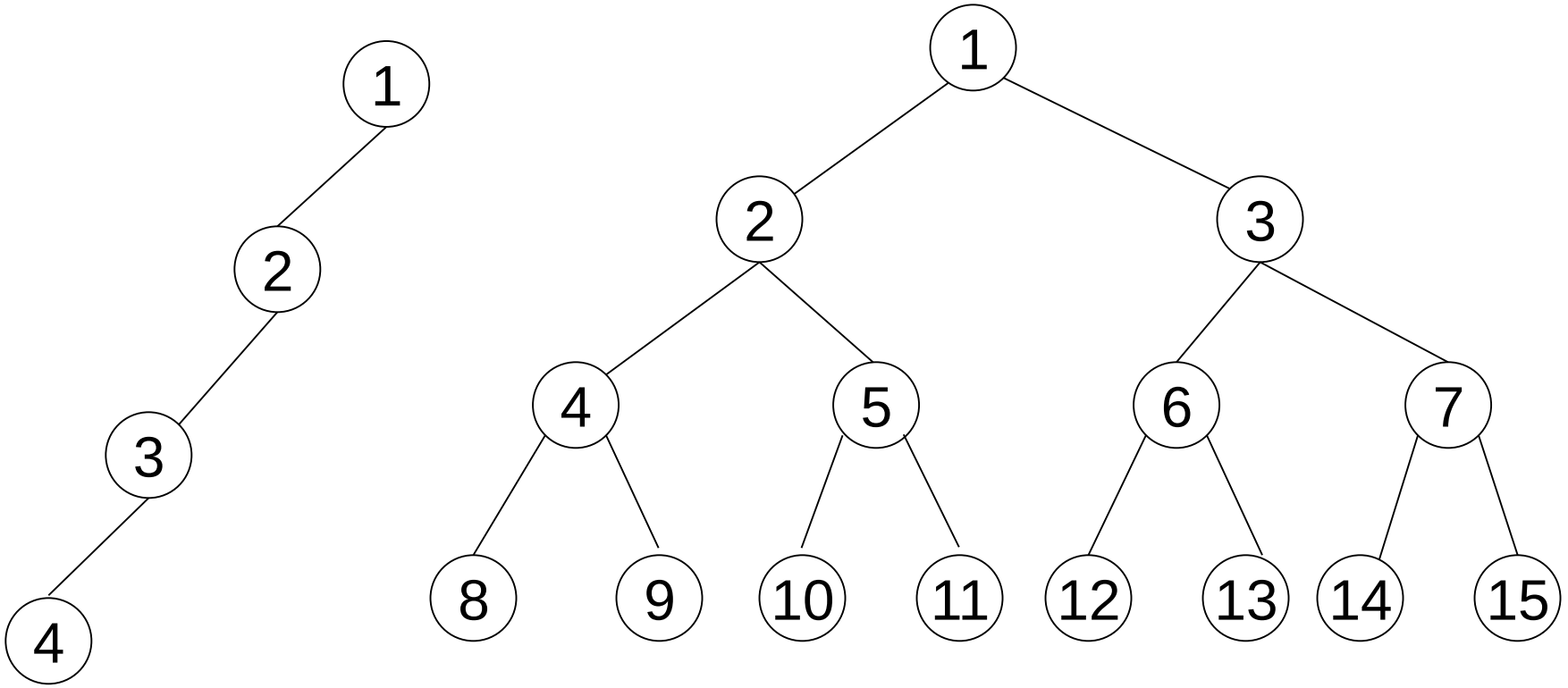
二叉树性质

性质1. 一棵非空二叉树的第i（i≥0）层上至多有2^i个结点

证明（采用归纳法）：

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，也即此层结点数至多有2^j；
3. 归纳结论：对于i=j+1，结点数至多为 $2*2^j=2^{j+1}$

结点数目示例



结点最少的二叉树：

1+1+1+1

结点最多的二叉树：

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

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为

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

二叉树的性质

性质3. 任何一棵二叉树，度为 0 的结点比度为 2 的结点多一个，即$n_0 = n_2 + 1$，其中n_0表示度为 0的结点数，n_2表示度为 2 的结点数

证明： 设有n个结点的二叉树的度为0、1、2的结点数分别为 n0， n1，n2，则有　　n = n0 + n1 + n2　　（公式5.1）

设边数为 e。因为除根以外，每个结点都有一条边进入，故有

n = e + 1 成立。

由于这些边是由度为 1 和 2 的结点射出的，故有

e = n1+ 2·n2，于是

n = e + 1= n1 + 2·n2 + 1　　（公式5.2）

由公式（5.1）（5.2）得

n0 + n1 + n2 = n1 + 2·n2 + 1

即 n0 = n2 + 1

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等于其分支结点数加 1

证明：设二叉树结点数为 n，叶结点数为m，分支结点数为b，则有

n（总结点数）= m（叶）+ b（分支） (公式5.3)

∵ 每个分支恰有两个子结点（满），故有2*b条边；除根结点外，其他结点都恰有一条边连接父结点，故共有n-1条边。即， *n - 1 = 2b* (公式5.4)

∴ 由(公式5.3)，(公式5.4) 得 n-1= m+b -1 = 2b，得出

m(叶) = b（分支）+ 1

还可怎么证明?

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，将其所有空子树换为叶结点，所得的扩充满二叉树为T'。所有原来 T 的结点变为T'的分支结点。根据**满二叉树定理**，新添加的叶结点数目等于T的结点个数加1。而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**

二叉树的性质

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

证明：设其深度为k，则有

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

$$= 2^k - 1 + m_k$$

故有 $2^k\text{-}1 < n \leq 2^{k+1}\text{-}1$ （性质2）

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

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

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

二叉树的性质

性质7. 对于具有n个结点的完全二叉树，结点按层次由左到右编号，则对任一结点i（$0 \le i \le n-1$）有

1. 若$i=0$，则结点 i 是二叉树的根结点；若$i>0$，则其父结点编号是$[(i-1)/2]$
2. 当$2i+1 \le n-1$时，结点 i 的左子结点是$2i+1$，否则结点 i 没有左子结点；　当$2i+2 \le n-1$时，结点 i 的右子结点是$2i+2$，否则结点 i 没有右子结点
3. 当 i 为**偶数**且$0<i<n$时，结点 i 的左兄是结点$i-1$，否则结点 i 没有左兄；　当 i 为**奇数**且$i+1<n$时，结点 i 的右弟是结点$i+1$，否则结点 i 没有右弟

二叉树的性质

性质7 证明：这里证明(2)，(1) 和(3)即可由结论(2)推得。对于i = 0，由完全二叉树定义，其左子的编号是1，若1>n-1，即不存在编号为1的结点，此时结点i没有左子；右子的编号只能是2，如果2>n-1，此时结点i没有右子

对于i>0 **分两种情况**讨论：

① 设第 j 层的第一个结点编号为i(此时有i = 2j-1)，则其左子必为第j + 1层的第一个结点，其编号为2j+1 - 1 = 2i + 1，如果2i + 1 > n - 1，那么i没有左子；其右子必为第j + 1层第二个结点，其编号为2i + 2

② 假设第 j 层的 某个结点编号为i，若左子存在，则其左子必然是第 j + 1层中的第 2[i - (2j - 1)]个，即左子的编号为[2j + 1 - 1] + 2[i - (2j - 1)] = 2i + 1；若右子存在，则其右子的编号必是2i + 2

二叉树抽象数据类型

- 定义了二叉树的逻辑结构后需考虑其上可能实施的各种运算，以适合于二叉树的各种应用
  - 某些运算是针对整棵树的
    - 初始化二叉树
    - 合并两棵二叉树
    - 遍历（周游）二叉树
  - 大部分运算都是围绕结点进行的
    - 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

二叉树抽象数据类型

- 抽象数据类型中定义了二叉树基本操作的集合，在具体应用中可根据实际情况以此为基础进行适当的**扩充**和**删减**
- 由于抽象数据类型与存储的无关性，故抽象数据类型定义中没有具体规定该抽象数据类型的存储方式

二叉树结点的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
   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
};
```

二叉树的ADT

```
BinaryTreeNode<T>* Parent(BinaryTreeNode<T> *current);          // 返回父
BinaryTreeNode<T>* LeftSibling(BinaryTreeNode<T> *current);     // 左兄
BinaryTreeNode<T>* RightSibling(BinaryTreeNode<T> *current);    // 右弟
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
```

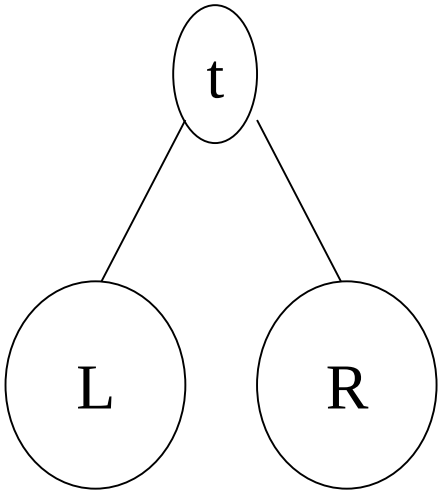
遍历二叉树

- 系统地访问数据结构中的每个结点，每个结点都正好被“访问”且**仅被访问到一次**
  - 也称周游（traversal）
  - “访问”是指对结点期望的操作，可理解成对二叉树结点数据成员的处理，比如输出、修改结点的信息等
  - 遍历一棵二叉树实际上是把二叉树的结点放入一个线性序列的过程，即对**二叉树的线性化**过程

遍历二叉树

- **深度优先遍历**
  - depth-first search/traversal(DFS/DFT)
- **广度优先遍历**
  - breadth-first search/traversal(BFT/BFS)

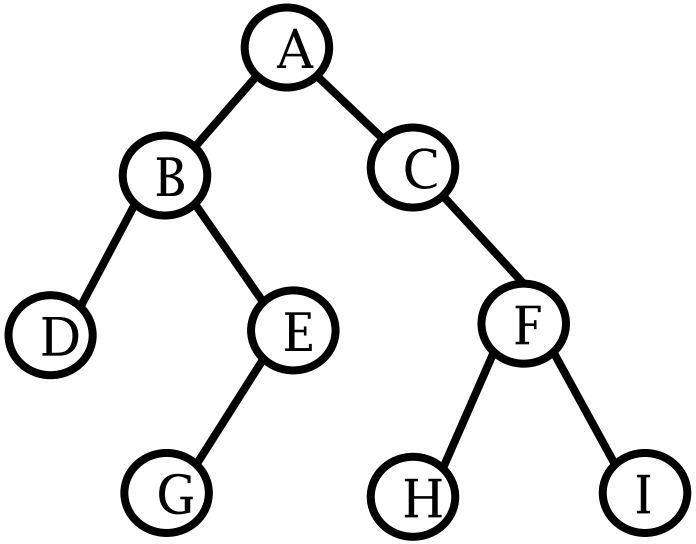
深度优先遍历二叉树



- 由定义知，一棵二叉树包括三部分：
  - 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)：
    - 中序遍历左子树；访问根结点；中序遍历右子树。也称 对称序
  - **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

深度优先遍历二叉树



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

某种序列下的前驱/后继

表达式二叉树

- 对右图进行前序、后序和中序次序周游可分别得到如下的结点序列：

前序序列：÷ - a b + c d

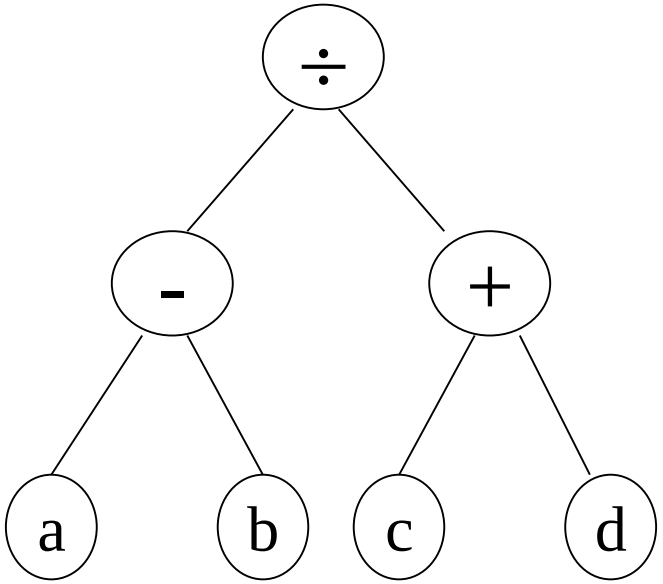
前缀表示（波兰表示法）

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



思考

- 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？

二叉树的遍历算法

- 按实现方式分为：
  - 递归算法 ——非常简洁，推荐使用

    void visit(BinaryTreeNode p);

    /* p是被遍历的二叉树的结点 */
  - 非递归算法
    - 理解编译栈的工作原理
    - 理解深度优先遍历的回溯特点
    - 有些资源受限的应用环境不适合递归

深度优先遍历二叉树

基本框架：

```
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
    }
}
```

二叉树DFS的非递归实现

- 如何把递归程序转化成等价的非递归算法的问题
  - 栈是实现递归的最常用的结构
  - 解决这个问题的关键是设置一个栈结构，利用一个栈来记下尚待遍历的结点或子树，以备以后访问

非递归前序遍历二叉树

- **基本思想**
  - 遇到一个结点，访问之，并将其压栈，再下降去遍历其左子树；
  - 遍历完左子树后，从栈顶弹出该结点，并按其右子地址再去遍历右子树

有无优化可能?

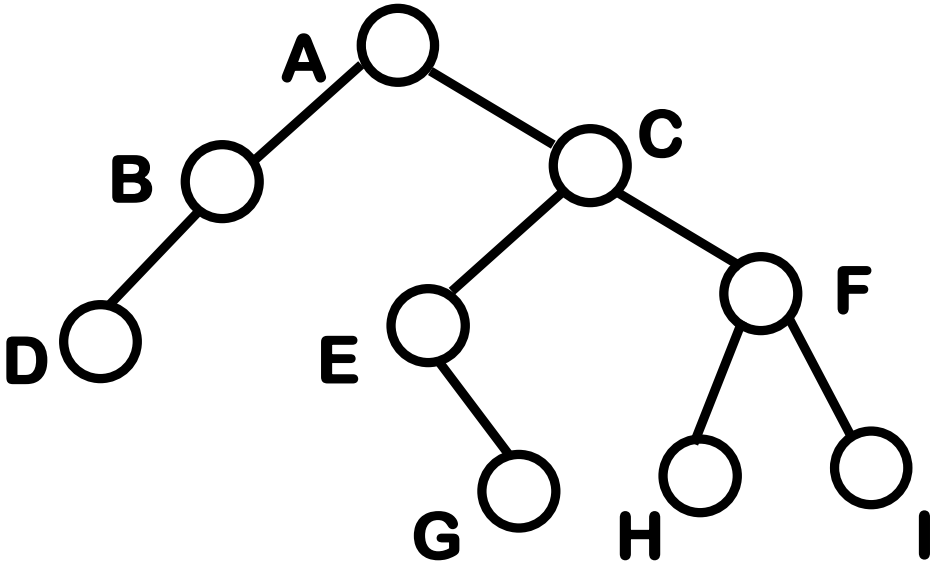
非递归前序遍历二叉树：优化版

- **基本思想**
  - 遇到一个结点，访问之，并将其非空右子结点压栈，然后下降去遍历其左子树；
  - 按前序方式遍历完左子树后，从栈顶弹出一个结点，访问之，并继续按照前序方式遍历其子树

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
        }
    }
}
```

前序非递归示例

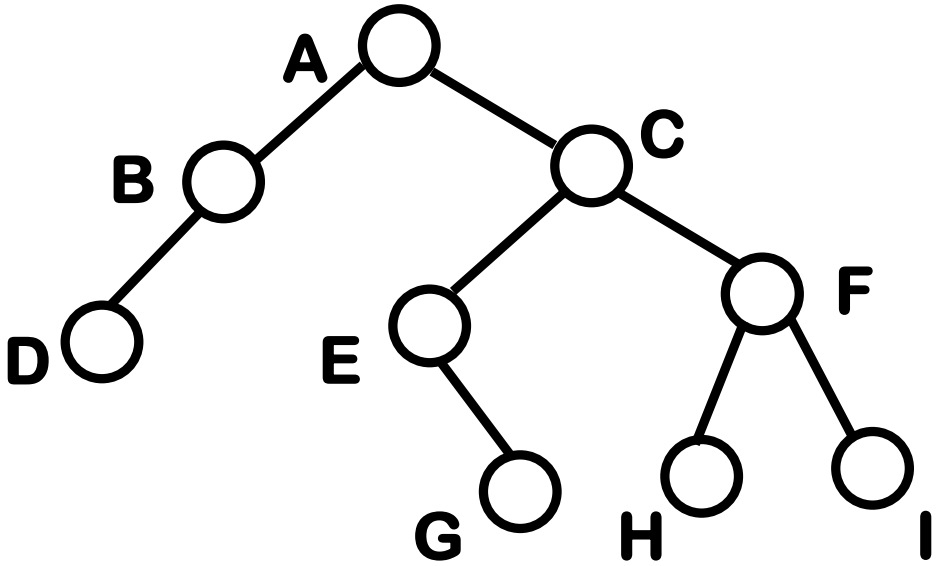


栈中变化?

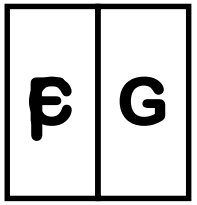
前序非递归示例

前序序列　A　B　D　　E　　H

入栈序列　C　F　G　I



栈



访问结点

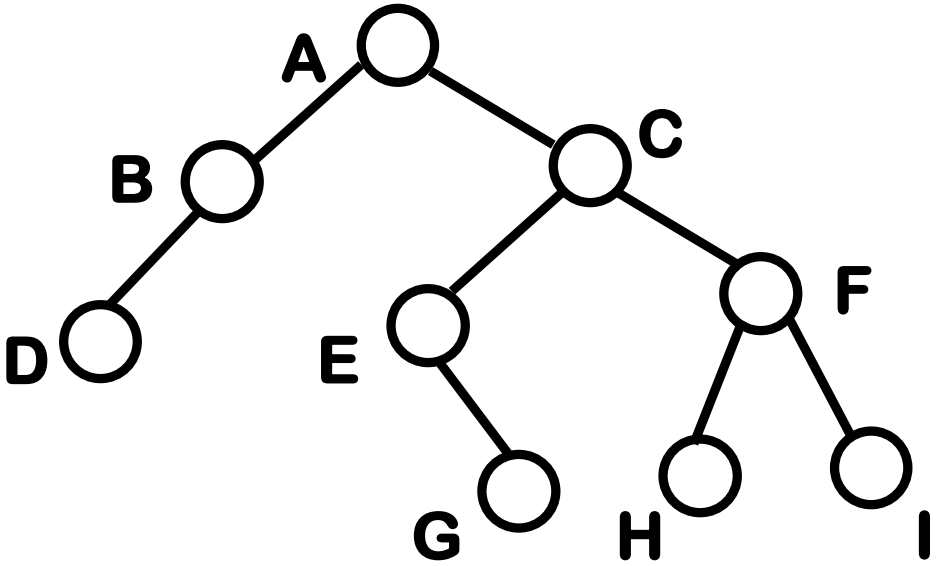
栈中结点

已访问结点

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；然后按其右链指示的地址再去遍历其右子树

中序非递归示例

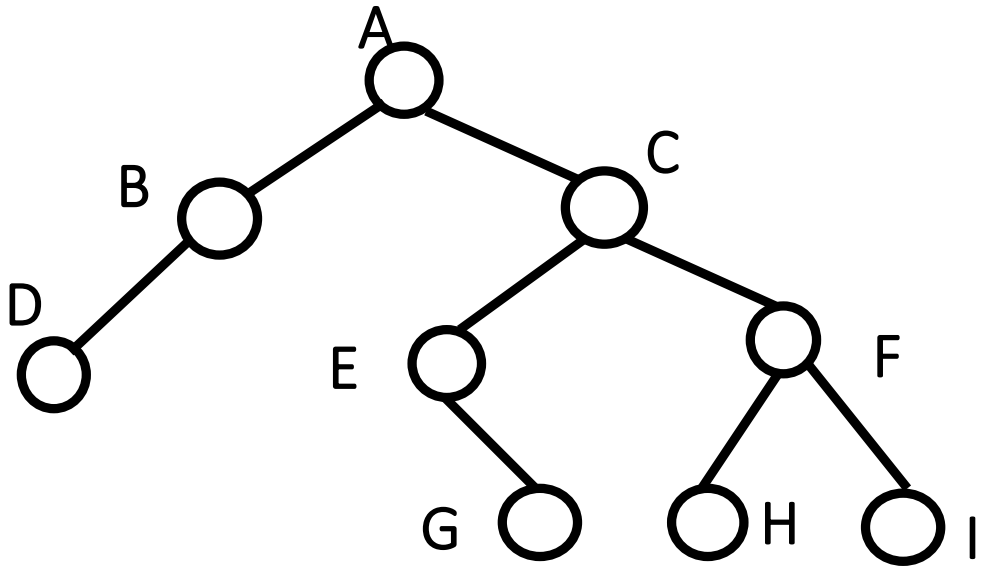


栈中变化?

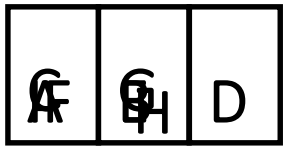
中序非递归示例

中序序列

进栈序列　　A　B　D　C　E　G　F　H　I



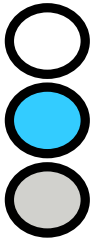
栈



未访问结点

栈中结点

出栈结点



非递归后序遍历二叉树

- **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，须再按其右链结构指示的地址去遍历其右子树
  - 遍历完右子树后才可从栈顶弹出该结点并访问之

问题： 当从栈中取出某结点时**如何判断**是继续访问其右子树还是直接访问该结点?

非递归后序遍历二叉树

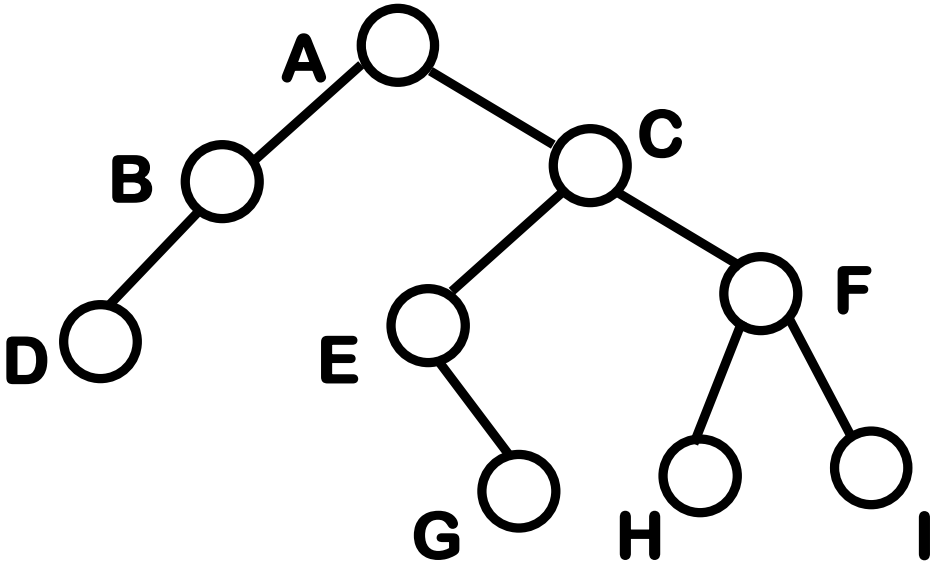
- **解决方案**
  - 需要给栈中的元素增加一个**特征位**，以便从栈顶弹出一个结点时可**区别**是从栈顶的左边（仍需遍历右子树），还是从右边（该结点的左、右子树均已遍历）返回
    - *Left* 表示已进入该结点的左子树，将从左边回来
    - *Right* 表示已进入该结点的右子树，将从右边回来

非递归后序遍历二叉树

栈中的元素类型定义 StackElement

```
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
};
```

后序非递归示例

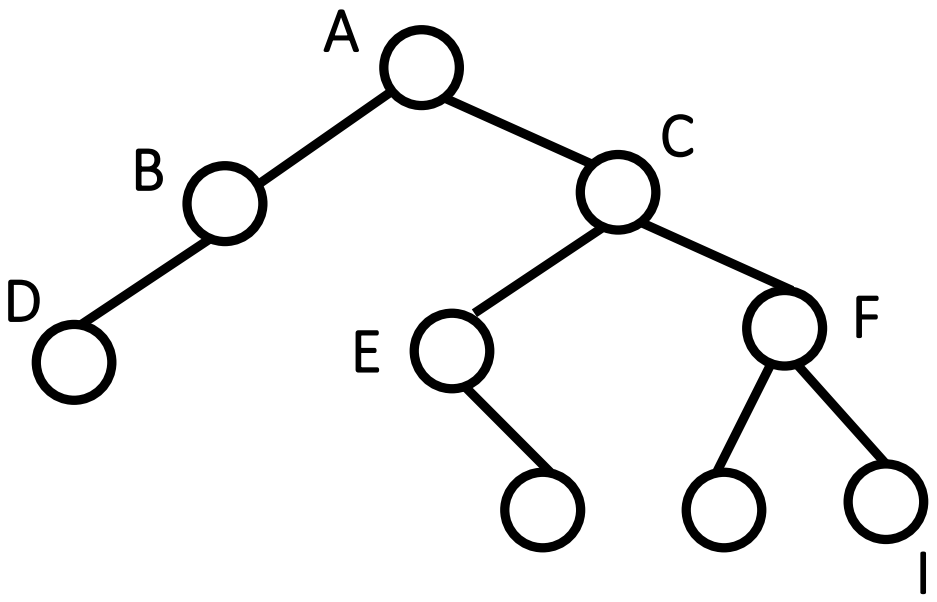


栈中变化?

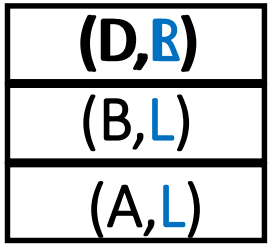
后序非递归示例

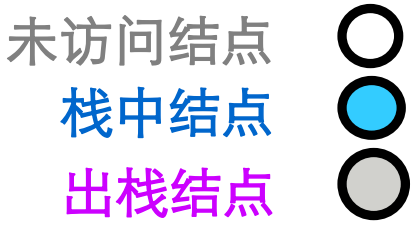
后序序列　　D

出栈序列



栈

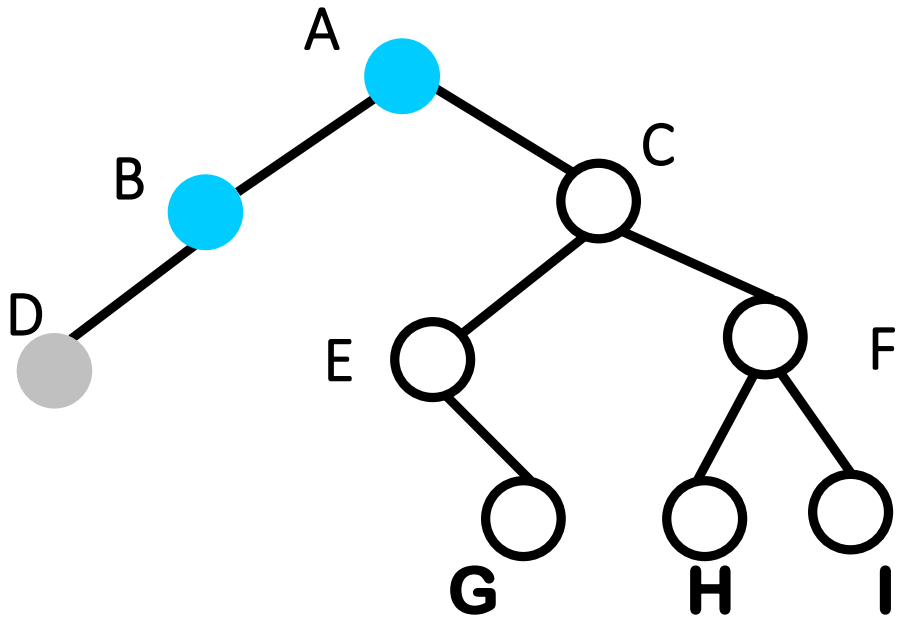




后序非递归示例

后序序列　　D　B

出栈序列　　(D,L)　(D,R)



栈

未访问结点

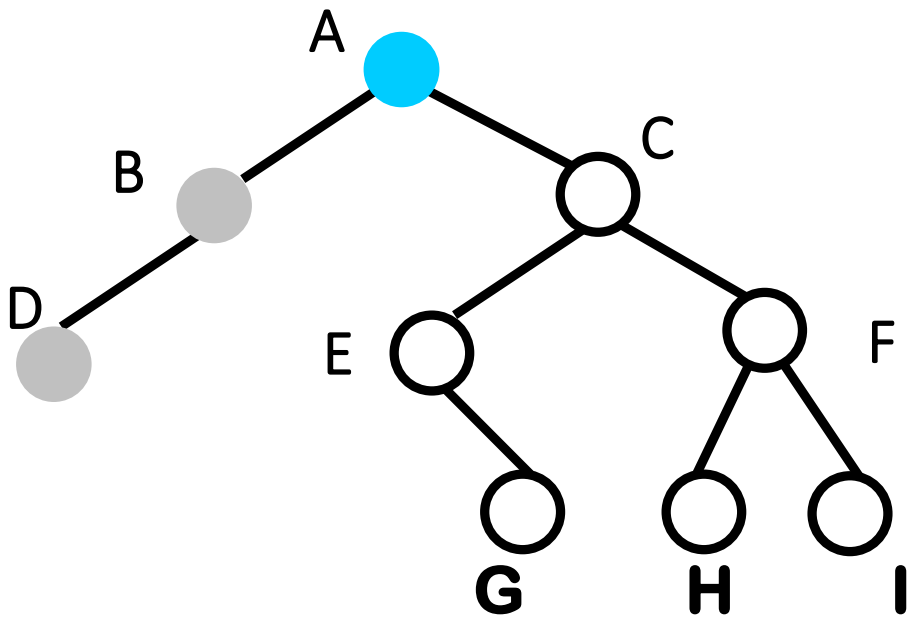
栈中结点

出栈结点

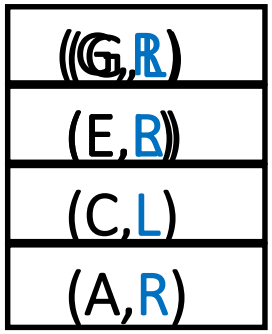
后序非递归示例

后序序列　D　B　G

出栈序列　(D,L)　(D,R)　(B,L)　(B,R)　(A,L)



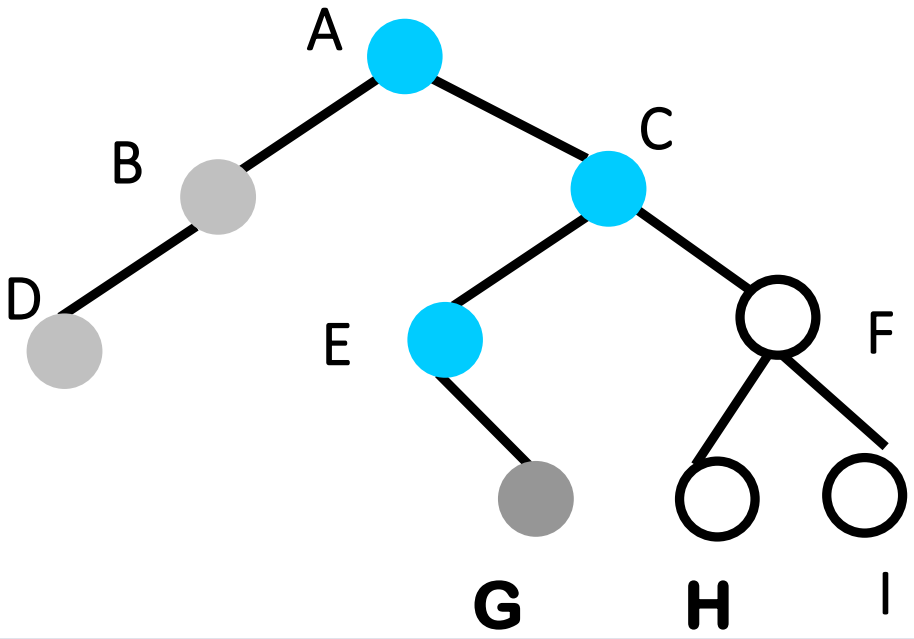
栈



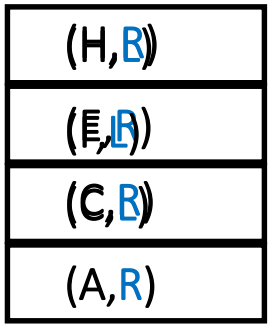
后序非递归示例

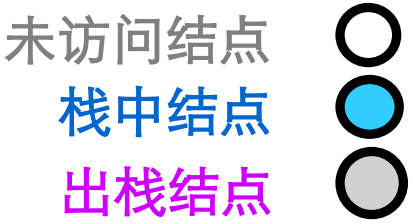
后序序列 D B G E H

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



栈

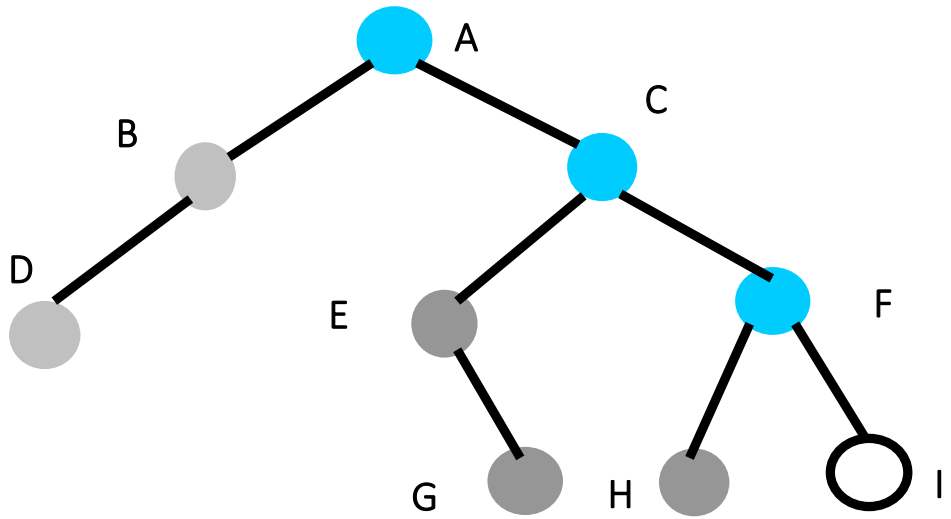




后序非递归示例

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

问题讨论

- **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?
- **非递归周游的意义**
  - 栈中存放的内容？
- **非递归的优缺点?**

深度优先遍历二叉树分析

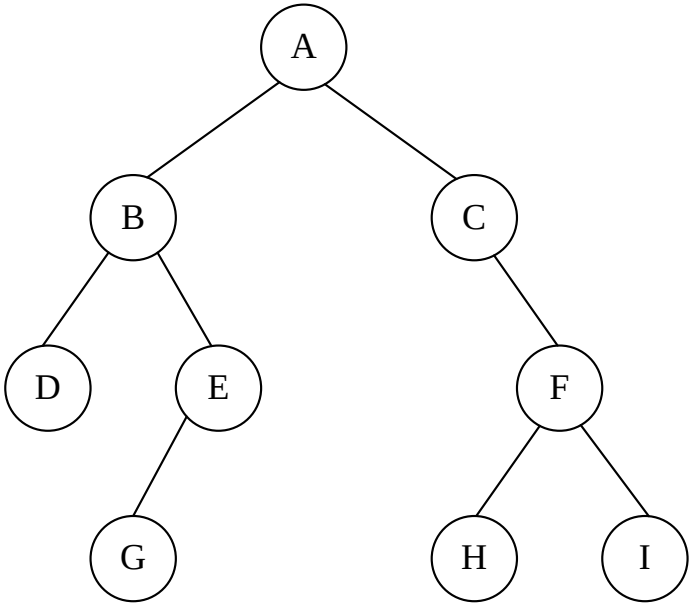
- 时间代价
  - 对于 n 个结点的二叉树，遍历完树的每一结点均需要O(n)时间
    - 前序、中序，某些结点入/出栈一次，不超过O(n)
    - 后序，每个结点分别从左、右边各入/出一次，O(n)
  - 前提是各结点处理（函数Visit的执行）时间为常数
- 空间代价
  - 所需辅助空间为遍历过程中栈的最大容量，即树的高度个结点空间
  - 最坏情况下具有 n 个结点的二叉树高度为 n，则所需要的空间复杂度为O(n)，最好 O(log n)

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

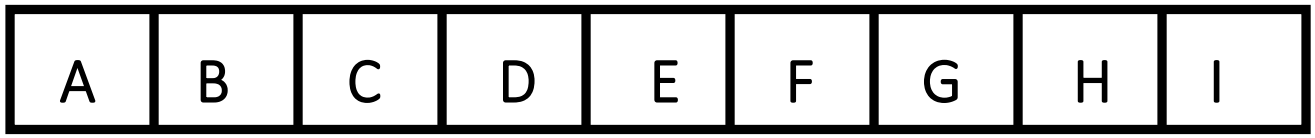
使用中间数据结构?

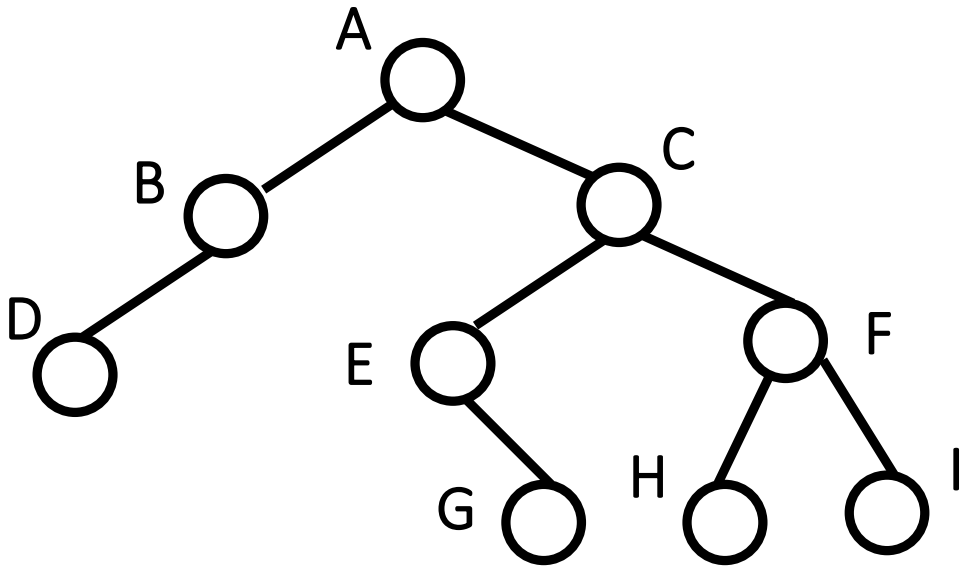


广度优先遍历二叉树示例

BFS序列

队列

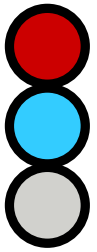




访问中结点

队列中结点

已访问结点



广度优先遍历二叉树

```
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
    }
}
```

广度优先遍历二叉树分析

- 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)
- 非递归入出队列时间
  - 宽搜，正好每个结点入/出队一次，O(n)
  - 代价与树的最大宽度有关
    - 最佳 O(1)
    - 最坏 O(n)

二叉树的存储

二叉树的存储实现

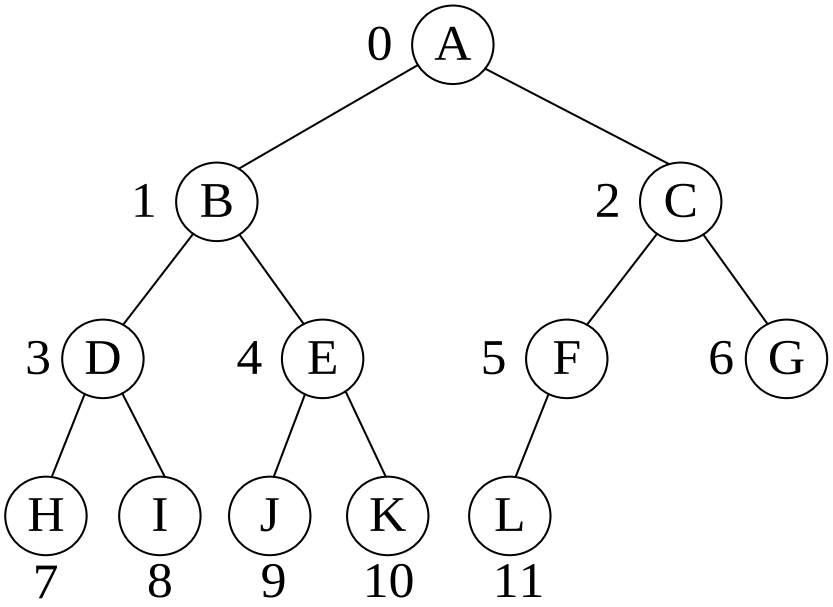
- 根据不同的应用，二叉树可采用不同的存储结构
  - 用数组实现完全二叉树（顺序存储）
  - 用指针实现二叉树 （链式存储）
    - 空间开销分析
  - 穿线二叉树

用数组实现完全二叉树

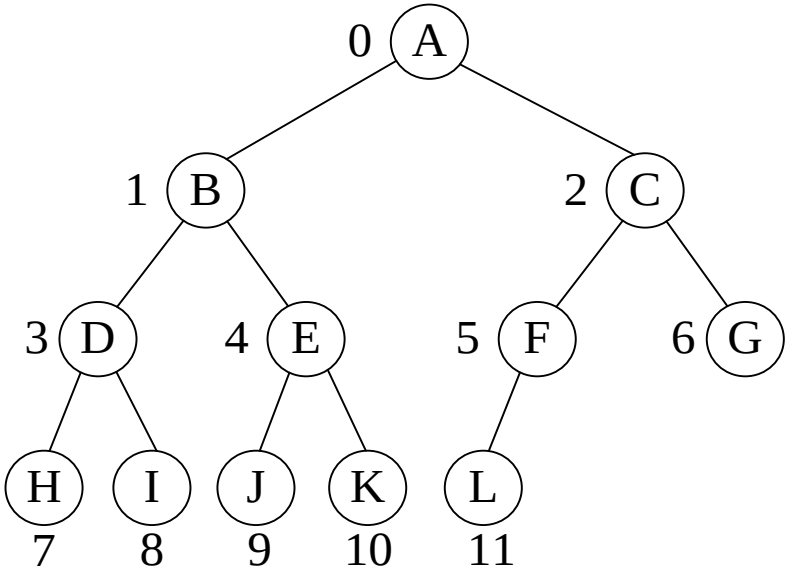
- 当要求一个二叉树按紧凑结构存储，且在处理过程中，该二叉树结构的大小和形状不动态发生剧烈变化时，可采用顺序存储方法
  - 即，把所有结点按照一定的次序顺序存储到一片连续的存储单元中
  - 适当安排结点的线性序列，可使结点在序列中的相互位置反映出二叉树结构的部分信息
  - 一般情况下，这样的信息不足以刻画整个结构，还需在结点中附加一些其他的必要信息，以完全反映整个结构

用数组实现完全二叉树

- 按层次顺序将一棵具有 *n* 个结点的完全二叉树的结点从 *0* 到 *n-1*编号，得到结点的一个线性序列



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	-	-

完全二叉树顺序存储总结

- **完全二叉树的层次序列足以反映其结构**
  - 所有结点按层次顺序依次存储在一片连续的存储单元中，根据结点的存储地址可算出它的左右子结点、父结点的存储地址
  - 最简单、最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**逻辑结构上依然为二叉树**

思考

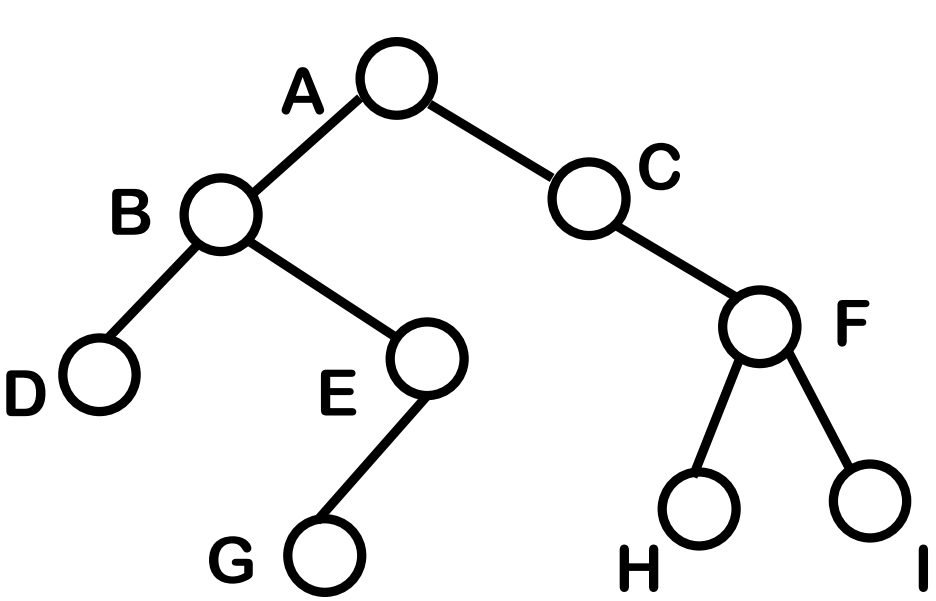
- 完全三叉树的下标公式?
- 非完全的二叉树可以采用顺序存储吗？若可以，如何存储?

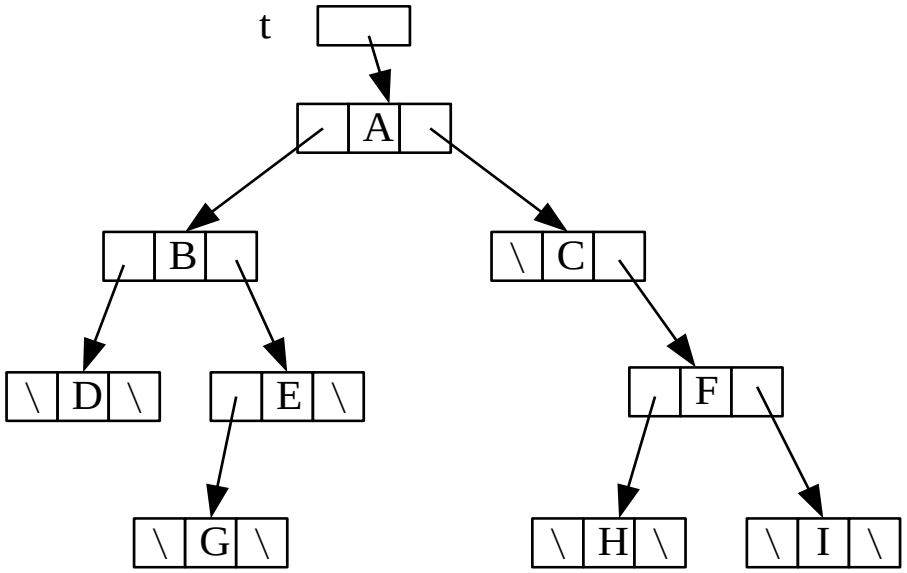
二叉树的链式实现

- 表示树形结构最自然的方法是**链接**方法，各结点随机地存储在内存空间中，结点之间的关系用**指针**表示
  - 每个结点中除存储结点本身的数据外设置两个指针字段 left 和right，分别指向其左子和右子
    - 当结点的某个子女为空时，则相应的指针为空指针
  - 结点的形式为

left	info	right

二叉树的链式实现





- 特点?
  - 在含有 *n*个结点的二叉链表中有*n+1*个空指针
  - 可否利用这些空指针? 存储方便某些操作的有用信息?

二叉树的链式实现

```
// 用二叉链表实现二叉树需要在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

// 判定二叉树是否为空树
template<class T>
bool BinaryTree<T>:: isEmpty() const {
                    return ( root != NULL ? false : true);
}
```

二叉树的链式实现

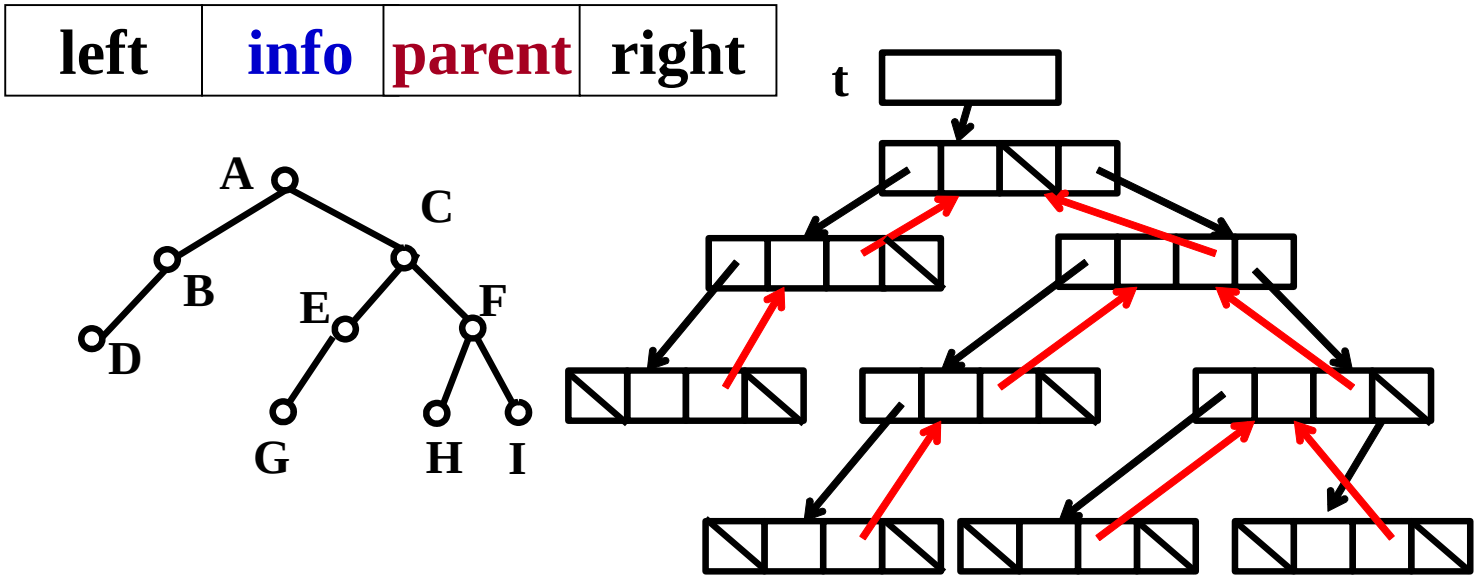
```
// 创建一棵新树，参数info为根结点元素，leftTree和rightTree是不同的两棵树
template <class T>
void BinaryTree<T>:: CreateTree (const T& info, BinaryTree<T>& leftTree,
     BinaryTree<T>& rightTree) {
     root = new BinaryTreeNode<T>(info, leftTree.root, rightTree.root);
                         // 创建新树
     leftTree.root = rightTree.root = NULL;
                         // 原来两棵子树根结点置为空，避免非法访问
}
// 后序周游删除二叉树
template<class T>
void BinaryTree<T>:: DeleteBinaryTree(BinaryTreeNode<T> *root)  {
     if (root != NULL) {
          DeleteBinaryTree(root->left);                    // 递归删除左子树
          DeleteBinaryTree(root->right);                   // 递归删除右子树
          delete root;                                     // 删除根结点
     }
}
```

二叉树的链式实现

```
// 求给定结点的父结点
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;                          // 如果子结点为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;
}
```

二叉树的链式实现

- 链接的其他表示方式：**三叉链表**
  - 结点结构中加入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



链式存储的特点

- **优点**
  - 运算方便，通过指针可直接访问相关结点
- **问题**
  - 空指针太多
  - 存储密度低

空间开销分析

- **结构性开销γ是**指为了实现数据结构而花费的辅助空间比例
  - 这些辅助空间并非用来存储数据，而是为了保存数据结构的逻辑特性或为了方便运算

$$\gamma(\text{结构性开销}) = \frac{\text{辅助结构存储量}}{\text{整个结构占用的存储总量}}$$

空间开销分析

- **存储密度α (≤1)表示数据结构存储的效率**

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

$\alpha = 1 - \gamma$

- 若所有的存储空间均用于数据本身，则称此存储结构**紧凑结构**，否则**非紧凑结构。**
  - 存储密度越大，则存储空间的利用效率越高；紧凑结构的存储密度为1
  - 非紧凑结构的存储密度小于1，二叉链表的存储显然是非紧凑结构

空间开销分析

根据满二叉树定理：一半的指针是空的

- 若只有叶结点存储数据，分支结点为内部结构（如Huffman树），则开销取决于二叉树是否满
  - 越满存储效率越高
- 若按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：$2pn$
  - 若 $p = d$，则 $2p/(2p + d) = 2/3$

空间开销分析

去掉满二叉树叶结点中的指针，

$$\frac{n/2(2p)}{n/2(2p)+nd}=\frac{p}{p+d}$$

则结构性开销为1/2 （假设p = d）

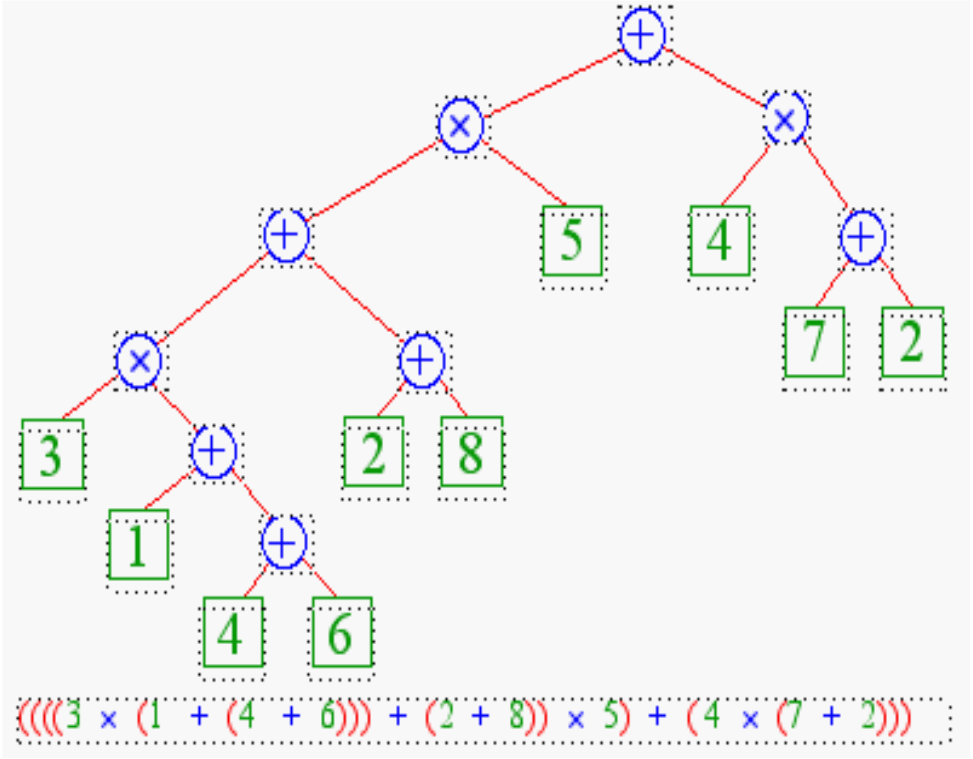
若只在叶结点存储数据，则结构性开销为

$2pn/(2pn + d(n+1)) \Rightarrow 2/3$ （假设 p = d）

注意：区分叶结点和分支结点需要额外的算法时间

区别叶结点和分支结点的存储

- 当叶结点和分支结点差别较大，或出于应用要求而需要区分叶结点和分支结点时



internal	llink	op	rlink

leaf	operand

补充内容：线索化二叉树

- 链表存储形式的二叉树中**有一半以上的指针**为空，如何利用这些空指针的存储空间?

  One way:

  - **利用空指针建立遍历线索**

- **目的**：利用**空指针的存储空间**，建立**遍历线索**，保存遍历过程中**得到的信息以供某些操作使用**

  （1）增加两个标志位

  （2）利用结构中的空指针，并设立标志

穿线二叉树 (ThreadBinaryTree)

- 在二叉链表存储形式的二叉树中，**利用结点中的空指针存储遍历线索**
  - 空的左指针指向结点在某种遍历序列下的前驱
  - 空的右指针指向结点在同一种遍历序列下的后继

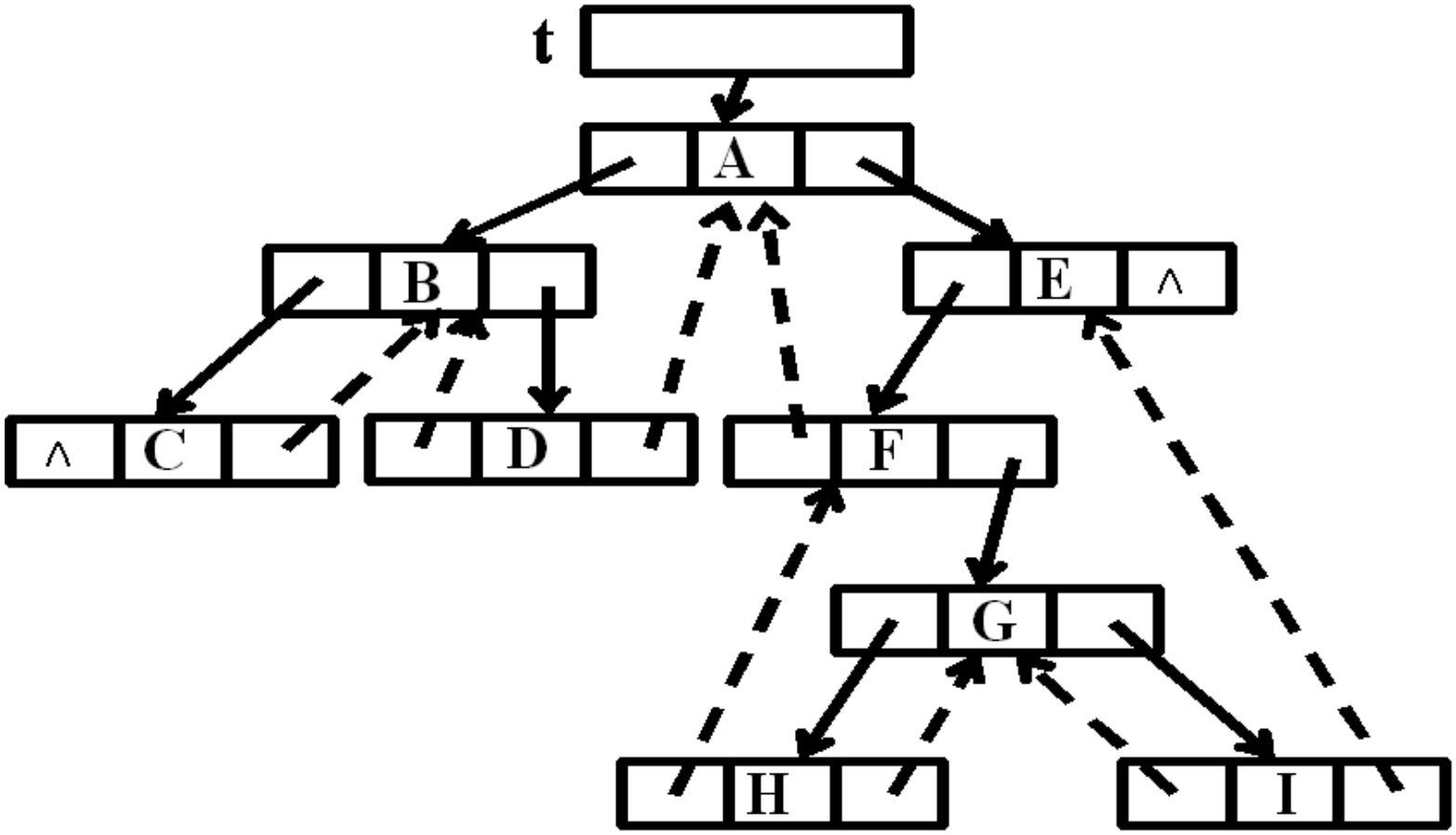
  这样的二叉树称为穿线树，可分为
  - 中序穿线树
  - 前序穿线树
  - 后序穿线树
- **穿线树可以只穿一半（前驱线索或后继线索）**

穿线二叉树

- 为了区分线索和指针，在每个结点中增加两个**标志位**，分别标识左右指针域是**指针**还是**线索**
  - **lTag = 0,　left为左子女指针**
  - **lTag = 1,　left为前驱线索**
  - **rTag= 0,　right为右子女指针**
  - **rTag = 1,　right为后继指针**

left	lTag	info	rTag	right

中序穿线二叉树示例



穿线二叉树结点类

```
template <class T> class ThreadBinaryTreeNode  {
private:
    int lTag,  rTag;                                    // 左右标志位
    ThreadBinaryTreeNode<T> *left, *right;              // 线索或左右子树
    T  element;
public:
    ThreadBinaryTreeNode();                             // 缺省构造函数
    ThreadBinaryTreeNode(const T)                       // 拷贝构造函数
          :element(T), left(NULL), right(NULL), lTag(0), rTag(0)   {};
  T& value() const {return element};
    ThreadBinaryTreeNode<T>* leftchild() const {return left};
    ThreadBinaryTreeNode<T>* rightchild() const {return right};
    void setValue(const T& type) {element = type};
  virtual ~ThreadBinaryTreeNode();                      // 析构函数
};
```

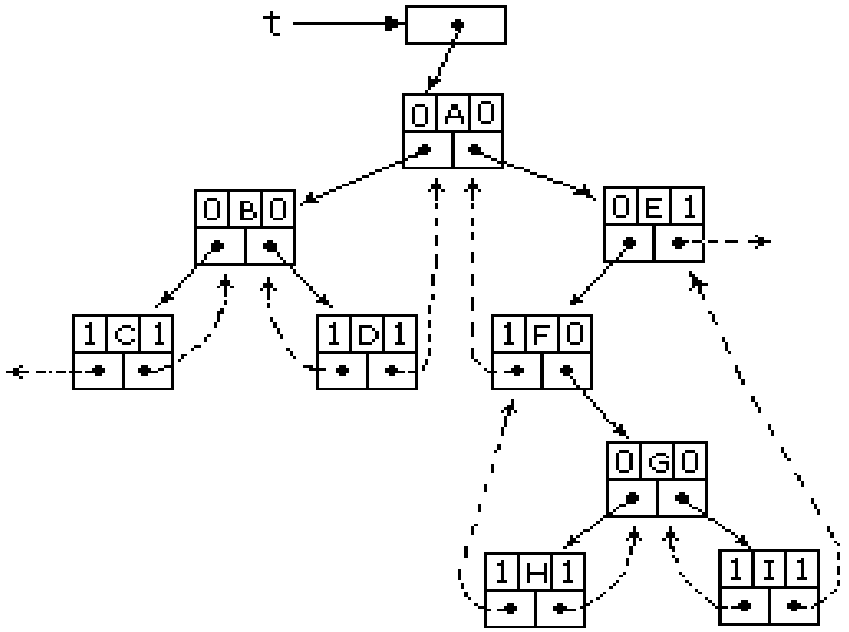
中序穿线二叉树类

```
template <class T> class ThreadBinaryTree {
private:
    ThreadBinaryTreeNode<T>* root;          // 根结点指针
public:
    ThreadBinaryTree() {root=NULL;};        // 构造函数
    virtual ~ThreadBinaryTree()  {DeleteTree(root);};
    // 返回根结点指针
    ThreadBinaryTreeNode<T>* getroot() {return root;};
    //中序线索化二叉树
    void InThread(ThreadBinaryTreeNode<T>* root);
    // 中序周游
    void InOrder(ThreadBinaryTreeNode<T>* root);
};
```

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) {                   // 建立前驱线索
              root->left = pre;
              root->lTag = 1;
          }
          if ((pre) && (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
}
```

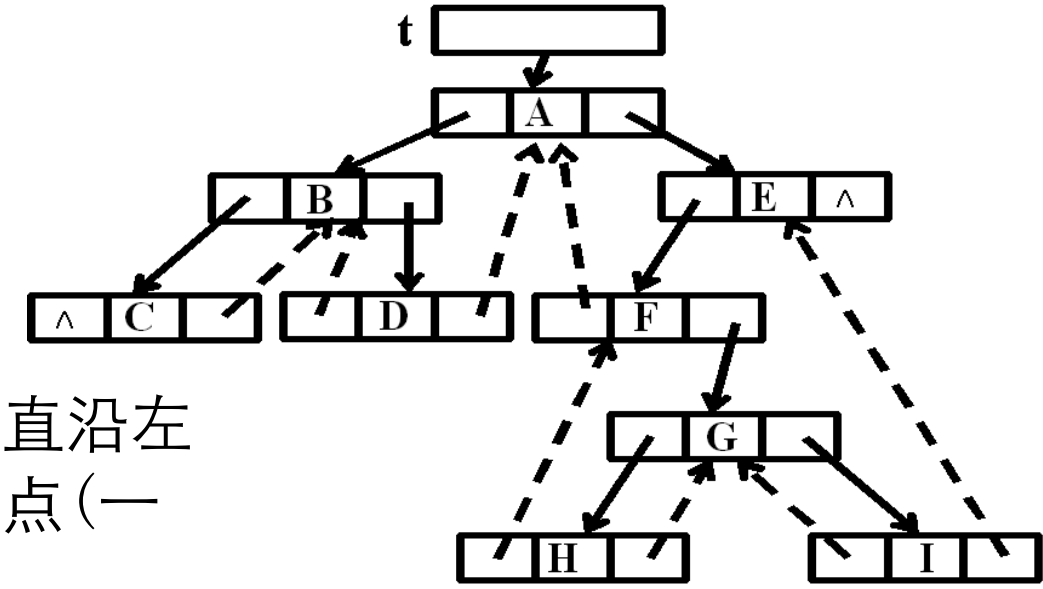
穿线树的性质



- 线索总是指向更上层的结点（向上访问能力）
  - 中序穿线树中，结点 z 的左线索指向其祖先结点y，z是y的右子树的“最左下”的结点（即按中序周游列出的y的后继结点）
  - 中序穿线树中，结点z的右线索指向其祖先结点x，z是x的左子树的“最右下”的结点（即按中序周游列出x的前驱结点）

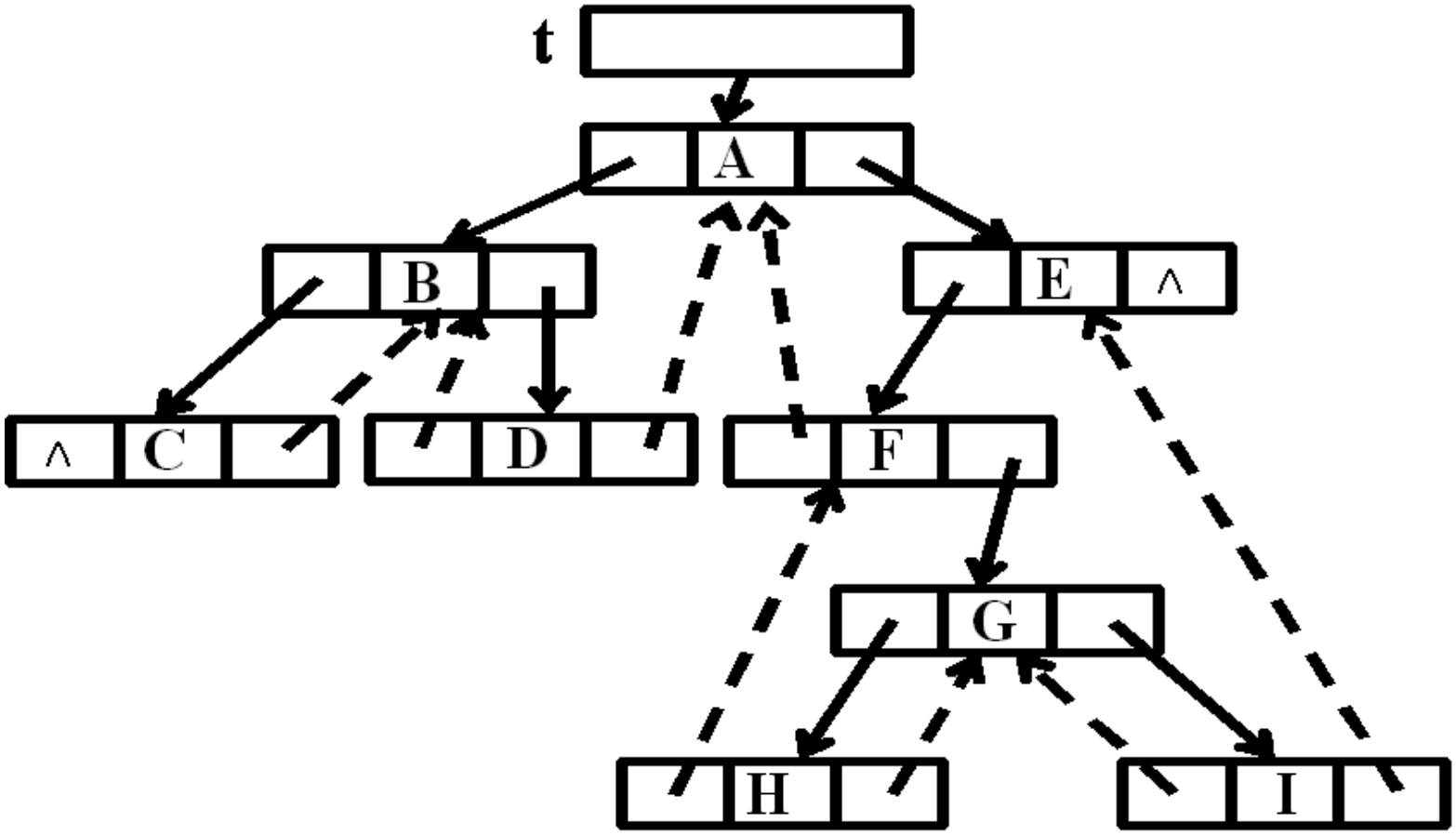
“向上”的线索可用来在中序穿线树里找出指定结点在前序下的后继结点和后序下的前驱结点，以及父结点

遍历穿线树



- 中序遍历中序穿线树
  - 从穿线树的根出发，一直沿左指针，找到“最左”结点(一定是中序的第一个结点，why?)；然后反复地找结点的中序后继
  - 一个结点的右指针若是线索，则右指针就是下一个要遍历的结点，若右指针不是线索，则其中序后继是其右子树的“最左”结点?

中序穿线二叉树示例



中序遍历穿线树的实现

```
Template <class T>void ThreadBinaryTree<T>::InOrder(
            ThreadBinaryTreeNode<T>* root)  {
    ThreadBinaryTreeNode<T>* pointer;
    if (root == NULL)  return;                          // 是否为空二叉树
    pointer = root;
    while (pointer->leftchild() != NULL)                // 找“最左下”结点
            pointer = pointer->leftchild();
    // 访问当前结点并找出当前结点的中序后继
    while (1) {
            visit(pointer->value());                    // 访问当前结点
            if (pointer->rightchild() == NULL)  return;
    if (pointer->rTag == 1)
                pointer = pointer->rightchild();        // 按照线索寻找后继
    else {                                              // 按照指针寻找后继
                pointer = pointer->rightchild();
                while (pointer->lTag  ==  0)
                    pointer = pointer->leftchild();     // 沿左链下降
            }  // end else
    }  // end while
}
```

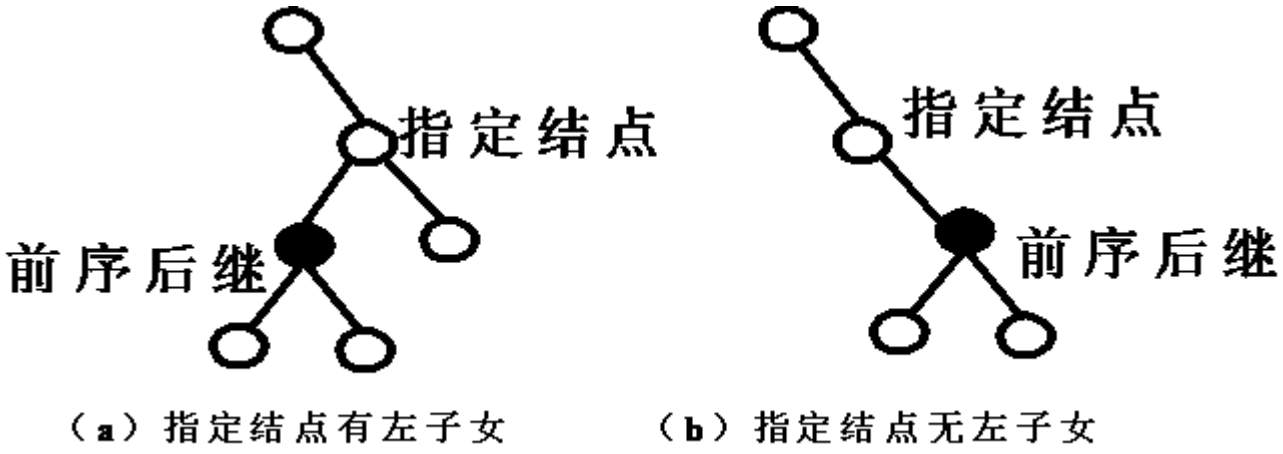
穿线二叉树的应用

1. **求中序穿线树里指定结点在前序下的后继结点**
   - 事实：若一个叶结点是某子树的中序下的最后一个结点，则它必是该子树的前序下最后一个结点

穿线二叉树的应用

- 情况一：指定结点非叶结点
  - 存在左子结点，则左子是它的前序后继
  - 没有左子结点，则右子是它的前序后继

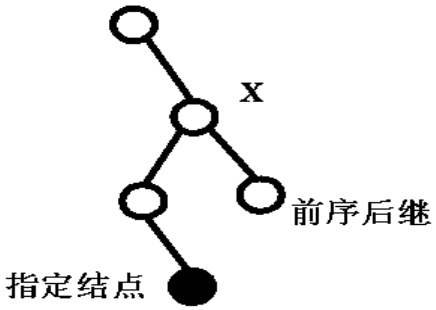
  此时不需要线索的帮助



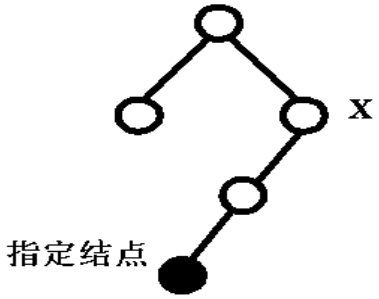
（a）指定结点有左子女　　（b）指定结点无左子女

穿线二叉树的应用

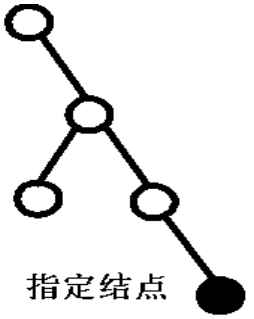
- 情况二：指定结点为叶结点
  - 若其为“某结点x”左子树中前序序列的终结结点，且结点 x 存在右子结点，则指定结点的前序后继就是结点x的右子结点 (图(a))
  - 若其并非任何结点的左子树前序序列的终结结点(图(c))；或虽为某结点x的左子树中前序序列的终结结点，但结点 x 没有右子结点(图(b))，则该结点没有前序后继
  - 关键在于如何找出“某结点x”



（a）结点 x 有右子女



（b）结点 x 无右子女



（c）不能找到结点 x，使指定结点是结点 x 的左子树中按前序的最后结点

穿线二叉树的应用

- **已知事实：若指定结点的右线索**指向一个祖先结点x，则指定结点是结点 x 的左子树的中序最后一个结点
  - 由此可知，指定结点也是结点 x 的左子树的前序最后一个结点。因此，指定结点的右线索所指的结点就是此处要找的某结点x
  - 故可借助于中序穿线树中右线索来找指定结点的前序后继结点

中序穿线二叉树中找指定结点的前序后继

```
template <class T> ThreadBinaryTreeNode<T>*
ThreadBinaryTree<T> :: FindNextinInorderTree
    (ThreadBinaryTreeNode<T>* pointer)  {
    ThreadBinaryTreeNode<T>* temPointer = NULL;
    if (pointer->lTag == 0)                          //  指定结点有左子女
        repointleftchild();
    else
        tempPointer = pointer;
    while (tempPointer->rTag == 1)
            tempPointer = tempPointer->rightchild();
    tempPointer = tempPointer->rightchild();
    return tempPointer;
}
```

穿线树总结

- **任何具有n个结点的二叉树，其二叉链存储表示中 *2n*个指针中的 空指针 数目 为 *n+1***
  - 空指针空间存储指向结点某种遍历次序下的前驱结点和后继结点的指针
- 穿线树的**最大优点**：**线索的存在**使得**遍历二叉树**和**找结点在指定次序下的前驱、后继**的算法变得简洁高效
  - 不再需要中间数据结构

二叉树的存储总结

- 二叉树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
  - 具体应用中采取什么存储结构，除根据二叉树的形态之外，还应考虑
    - 时间复杂度
    - 空间复杂度
    - 算法简洁性