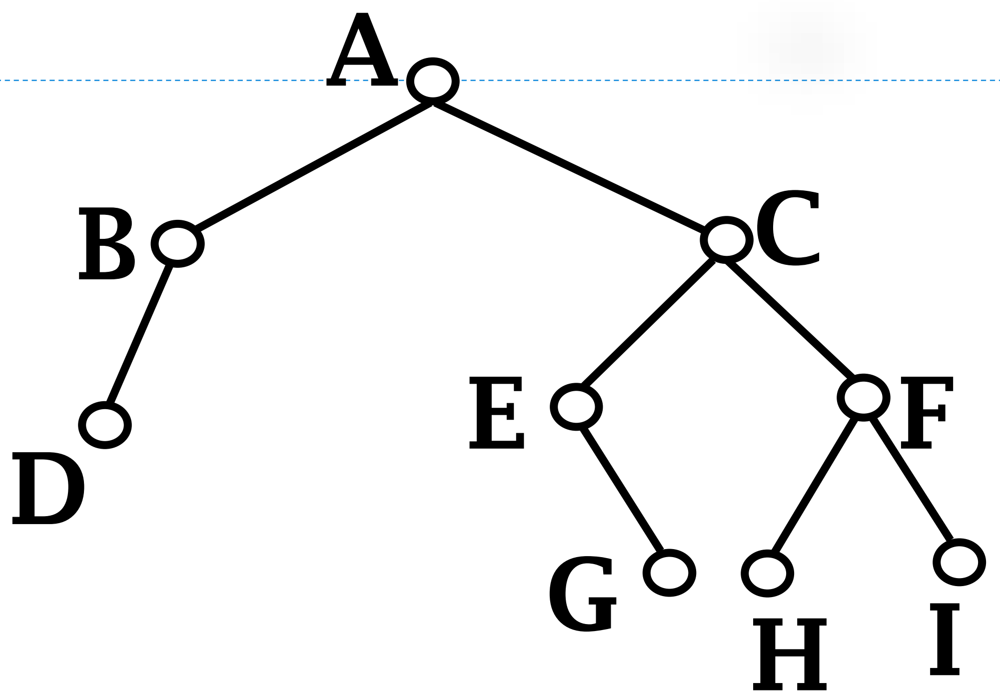
数据结构与算法(五)

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 （ “十一五”国家级规划教材 ）

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

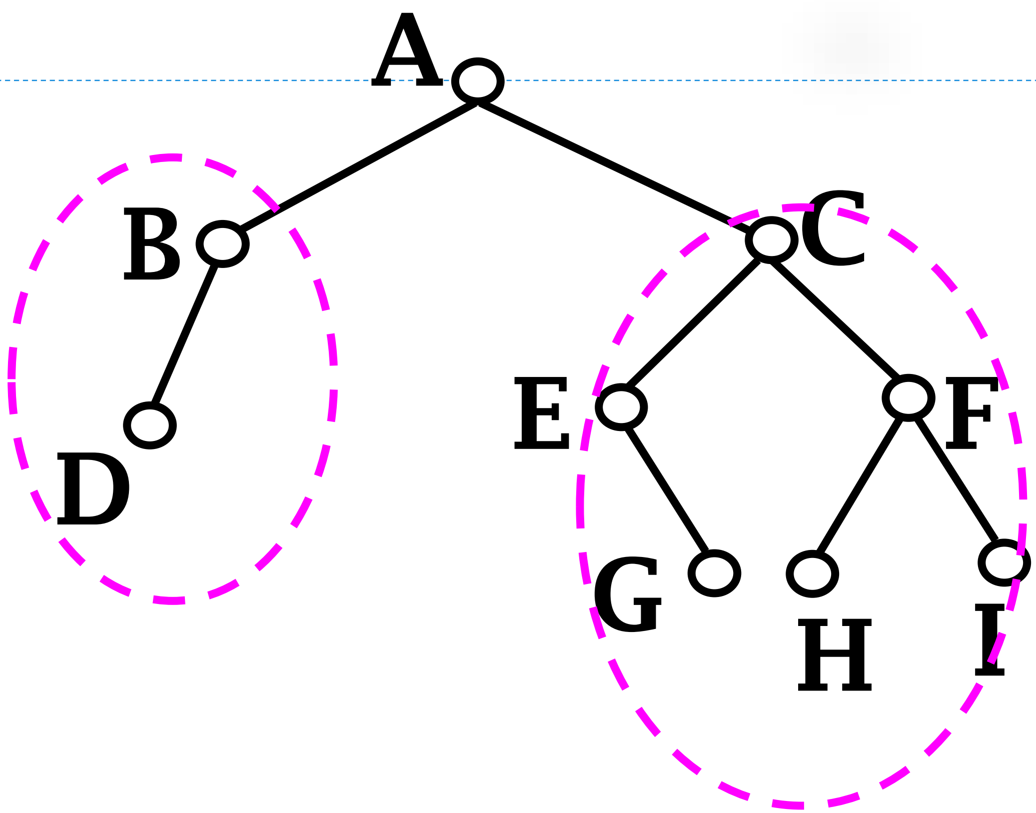
第五章 二叉树



- 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

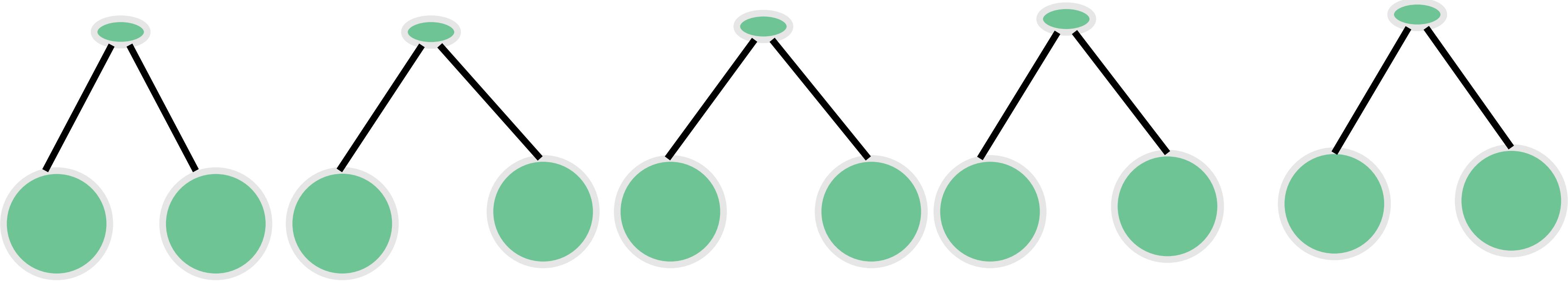
二叉树的概念

- **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



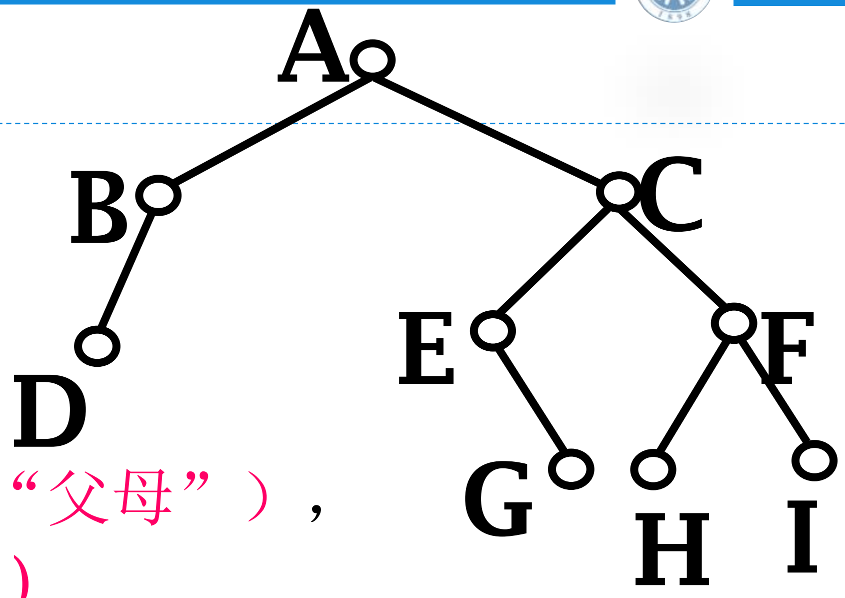
二叉树的五种基本形态

- 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



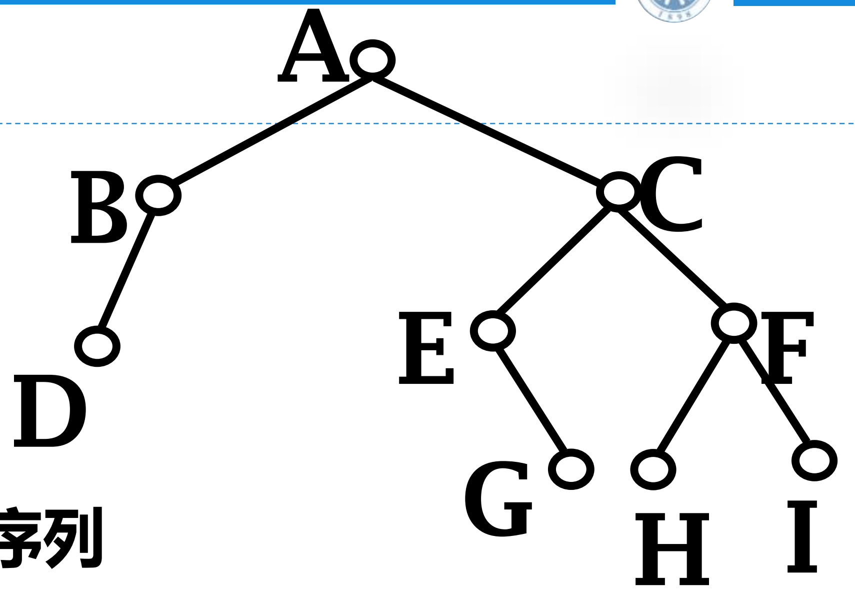
(a)空　(b)独根　(c)空右　(d)空左　(e)左右都不空

二叉树相关术语



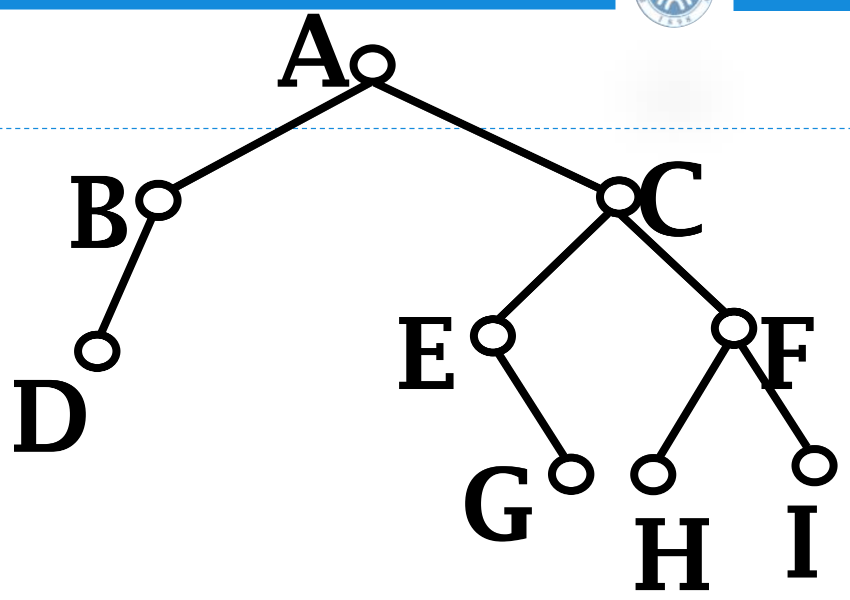
- **结点**
  - **子结点、父结点、最左子结点**
    - 若 <k，k′ >∈ r，则称 k 是 k′ 的父结点（或“父母”），而 k′ 则是 k 的 子结点 (或“儿子”、“子女”)
  - **兄弟结点、左兄弟、右兄弟**
    - 若有序对 <k，k′ >及 <k，k″ >∈ r，则称 k′ 和 k″ 互为兄弟
  - **分支结点、叶结点**
    - 没有子树的结点称作 叶结点（或树叶、终端结点）
    - 非终端结点称为分支结点

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**

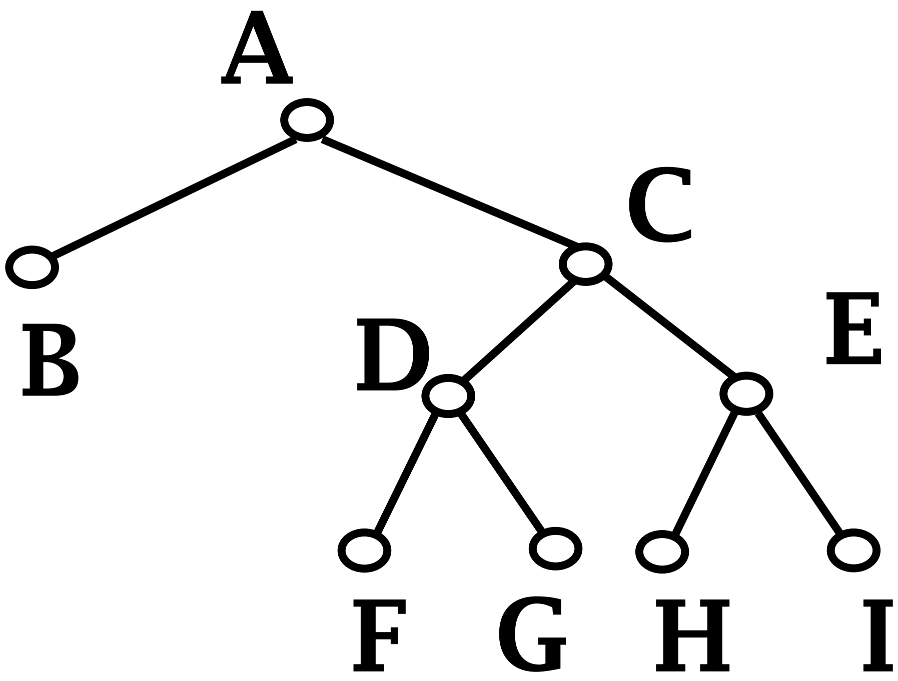
二叉树相关术语



- **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

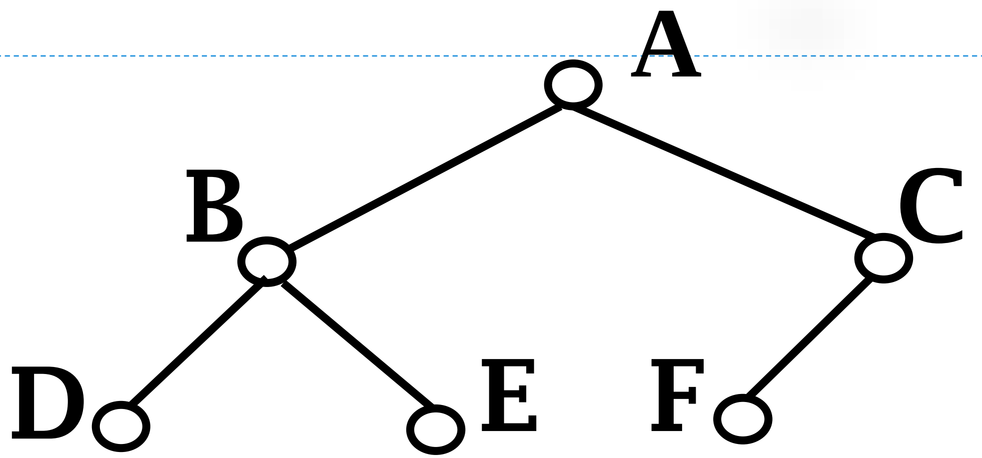
满二叉树

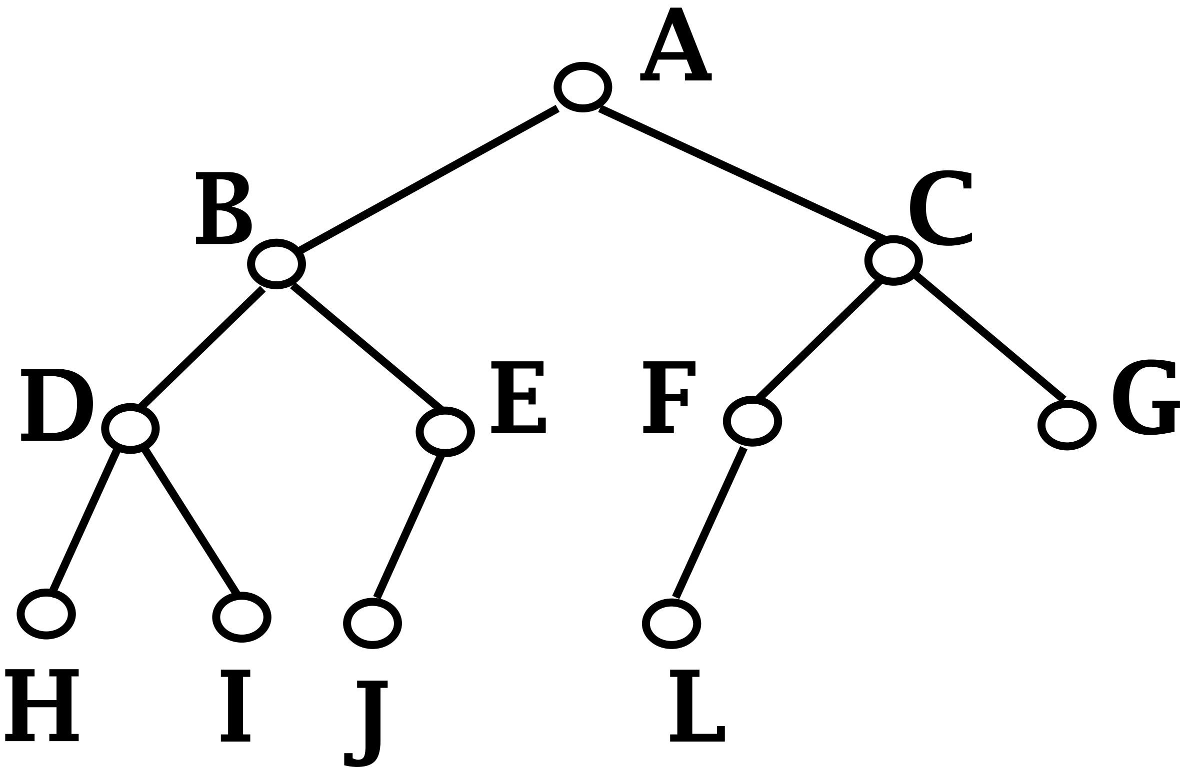
- 如果一棵二叉树的 **任何** 结点，或者是树叶，或者恰有两棵非空子树，则此二叉树称作 **满二叉树**



完全二叉树

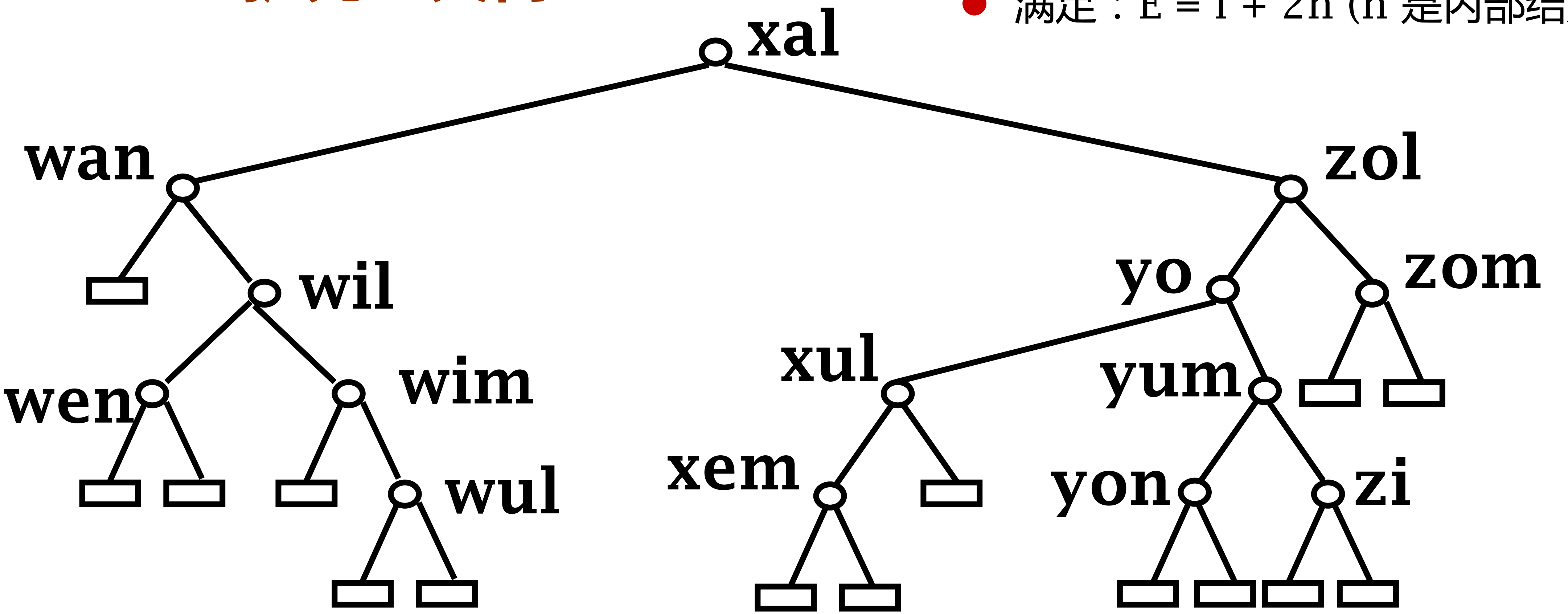
- 最多只有最下面的两层结点度数可以小于2
- 最下一层的结点都集中最左边





- **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

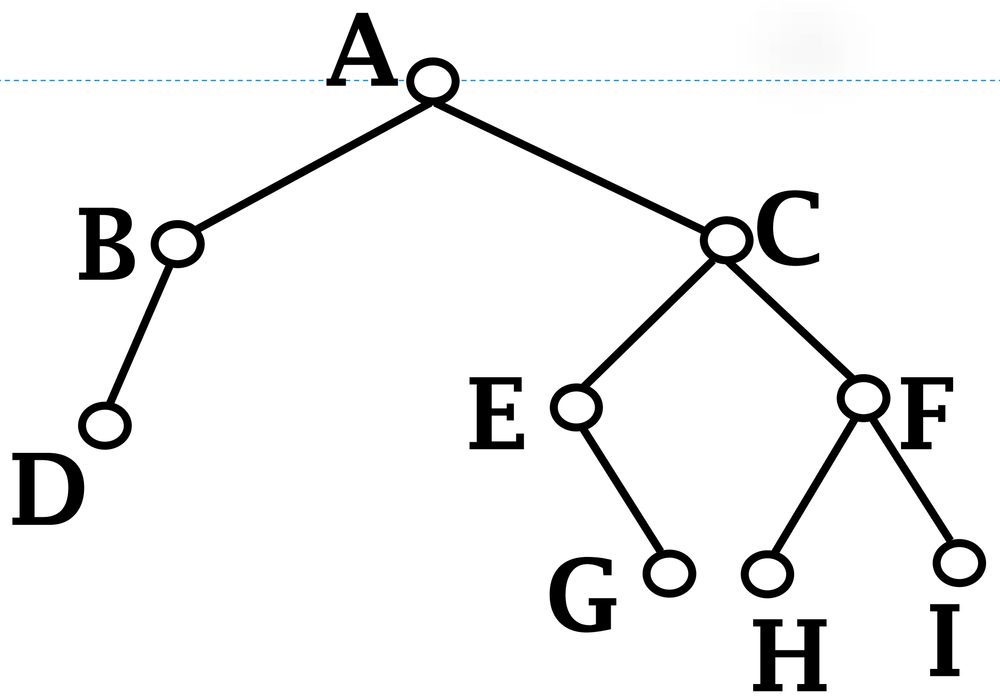
- 性质1. 在二叉树中，第i层上最多有 2^i 个结点（i≥0）
- 性质2. 深度为 k 的二叉树至多有 $2^{k+1}-1$个结点（k≥0）
  其中深度(depth)定义为二叉树中层数最大的叶结点的层数
- 性质3. 一棵二叉树，若其终端结点数为 n_0，度为2的结点数为 n_2，
  则 $n_0=n_2+1$
- 性质4. **满二叉树定理**：非空满二叉树树叶数目等于其分支结点数加1
- 性质5. 满二叉树定理推论：一个非空二叉树的空子树数目等于其结点数加1
- 性质6. 有n个结点（n>0）的完全二叉树的高度为 $\lceil \log_2 (n+1) \rceil$
  （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

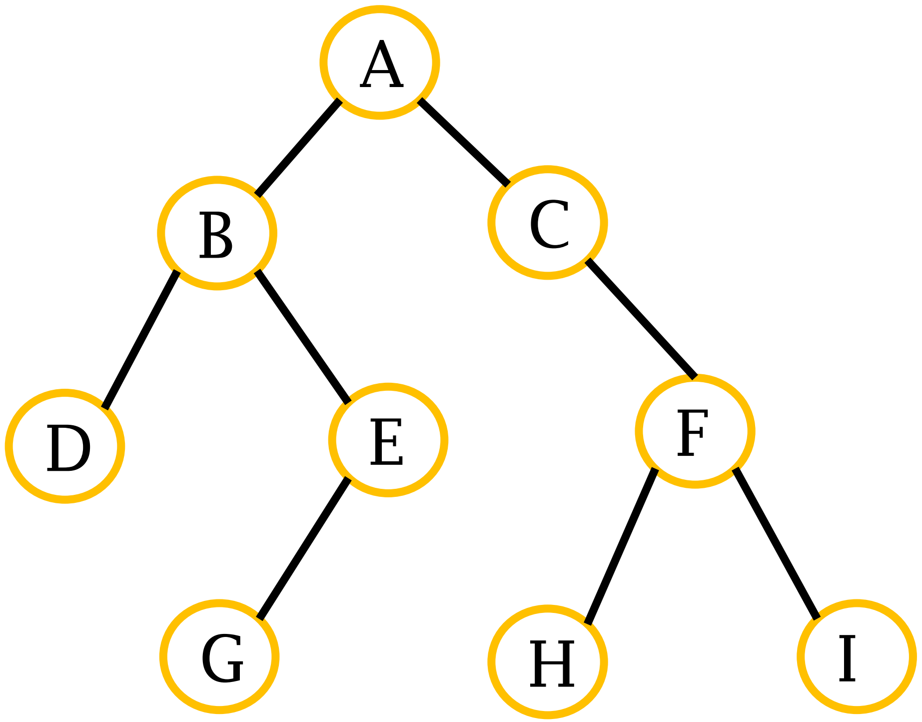
三种深度优先遍历的递归定义：

(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。

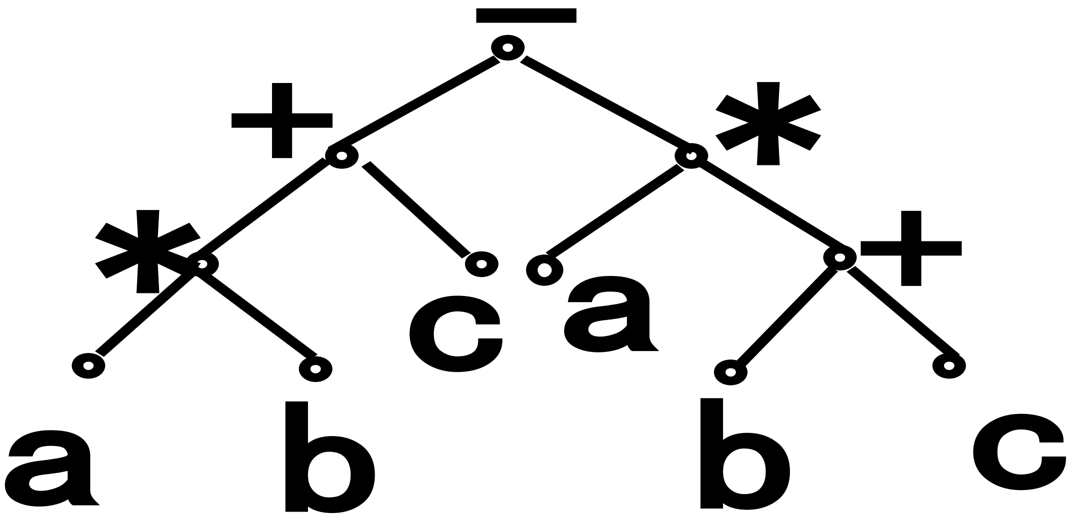
深度优先遍历二叉树



- 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
       }
 }
```

5.2 二叉树的抽象数据类型

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

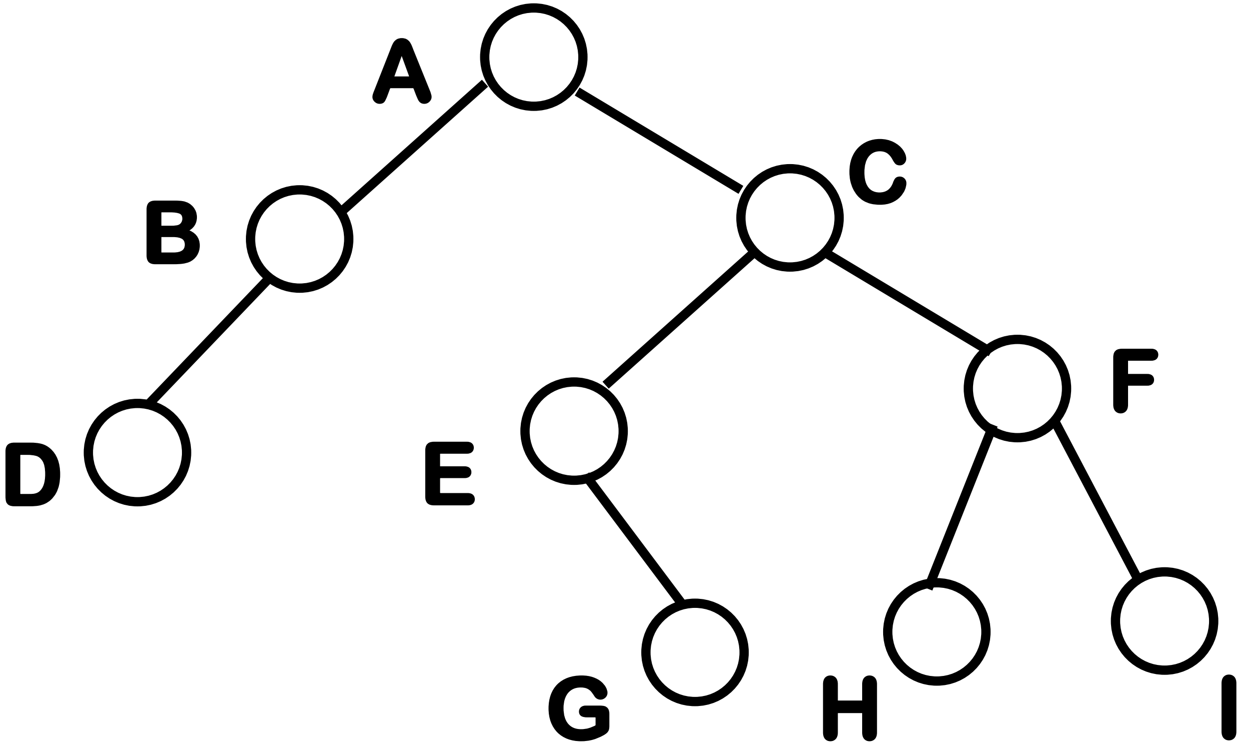
- 递归算法非常简洁——推荐使用
  - 当前的编译系统优化效率很不错了
- 特殊情况用栈模拟递归
  - 理解编译栈的工作原理
  - 理解深度优先遍历的回溯特点
  - 有些应用环境资源限制不适合递归

5.2 二叉树的抽象数据类型

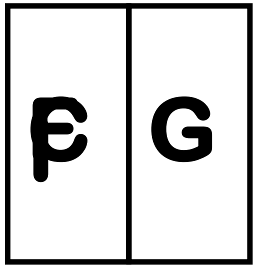
前序序列 A B D E H

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) {
```

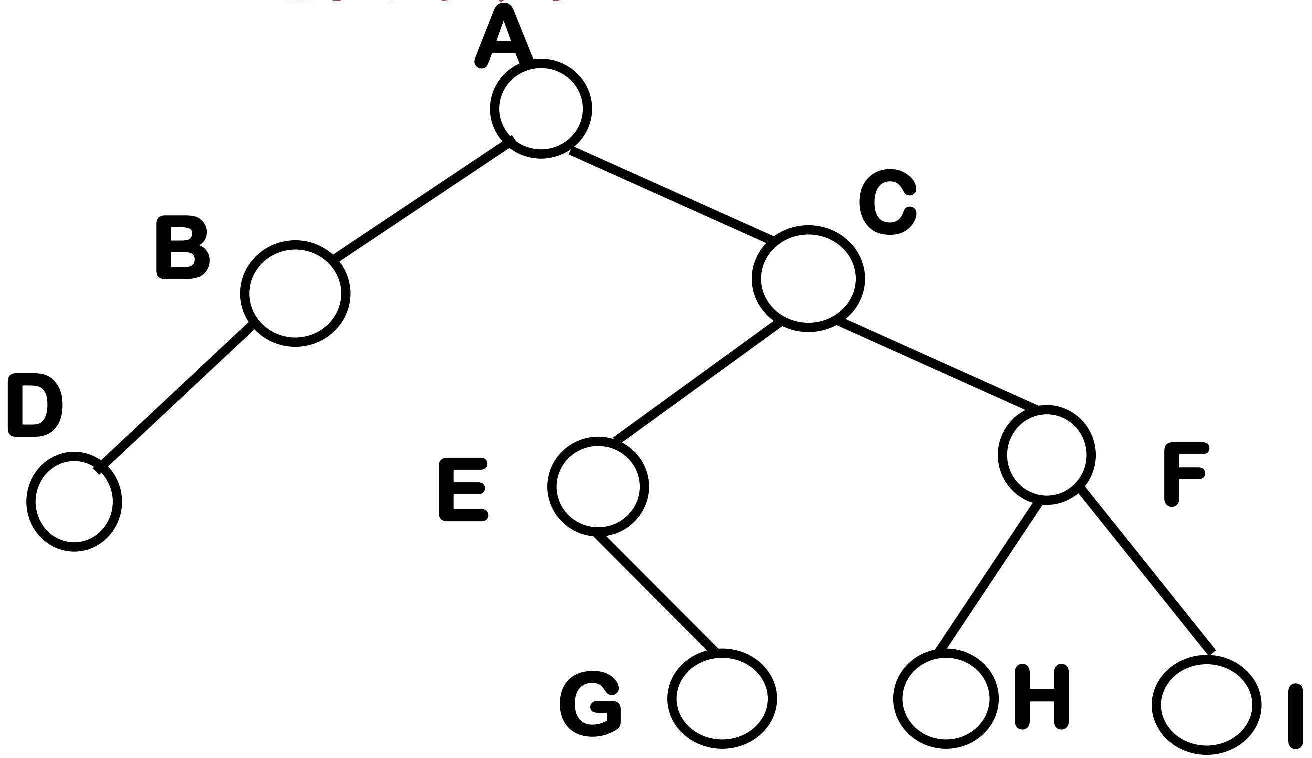
5.2 二叉树的抽象数据类型

```
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 }
} }
```

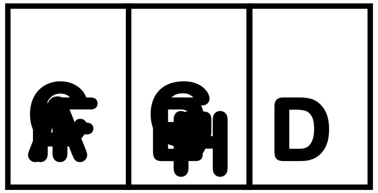
5.2 二叉树的抽象数据类型

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();
```

5.2 二叉树的抽象数据类型

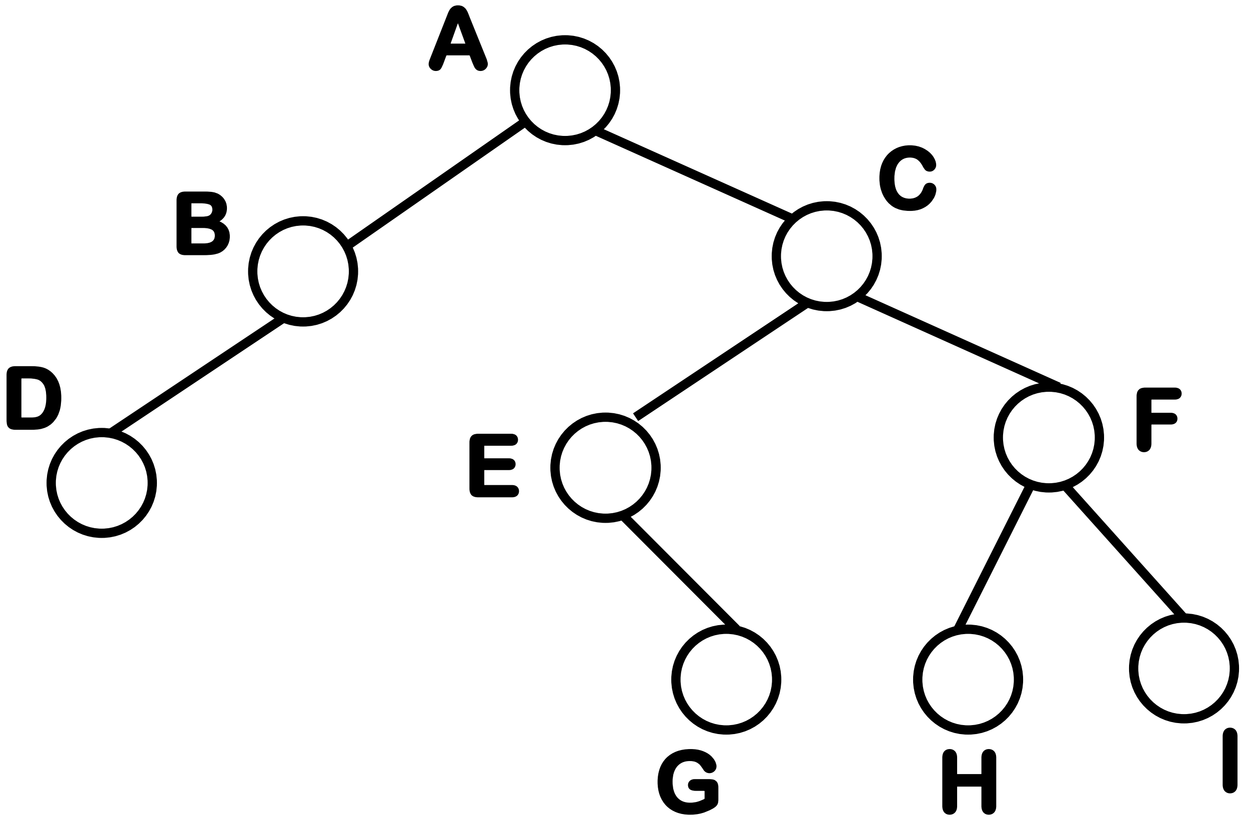
```
    } //end if
   else {      //左子树访问完毕，转向访问右子树
       pointer = aStack.top();
       aStack.pop();                    //栈顶元素退栈
       Visit(pointer->value());              //访问当前结点
       //当前链接结构指向右孩子
       pointer=pointer->rightchild();
   } //end else
 } //end while
}
```

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

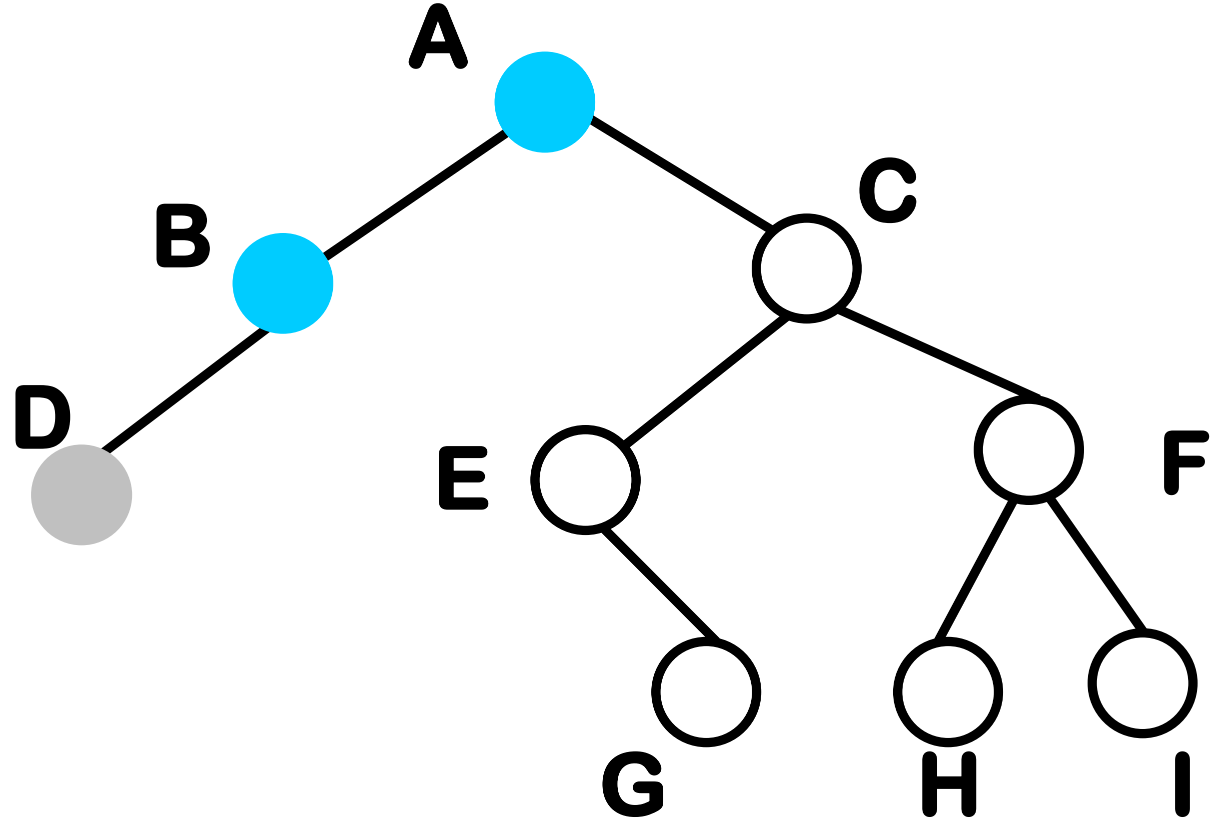
(D,R)
(B,L)
(A,L)

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

(B,R)
(A,R)

未访问结点

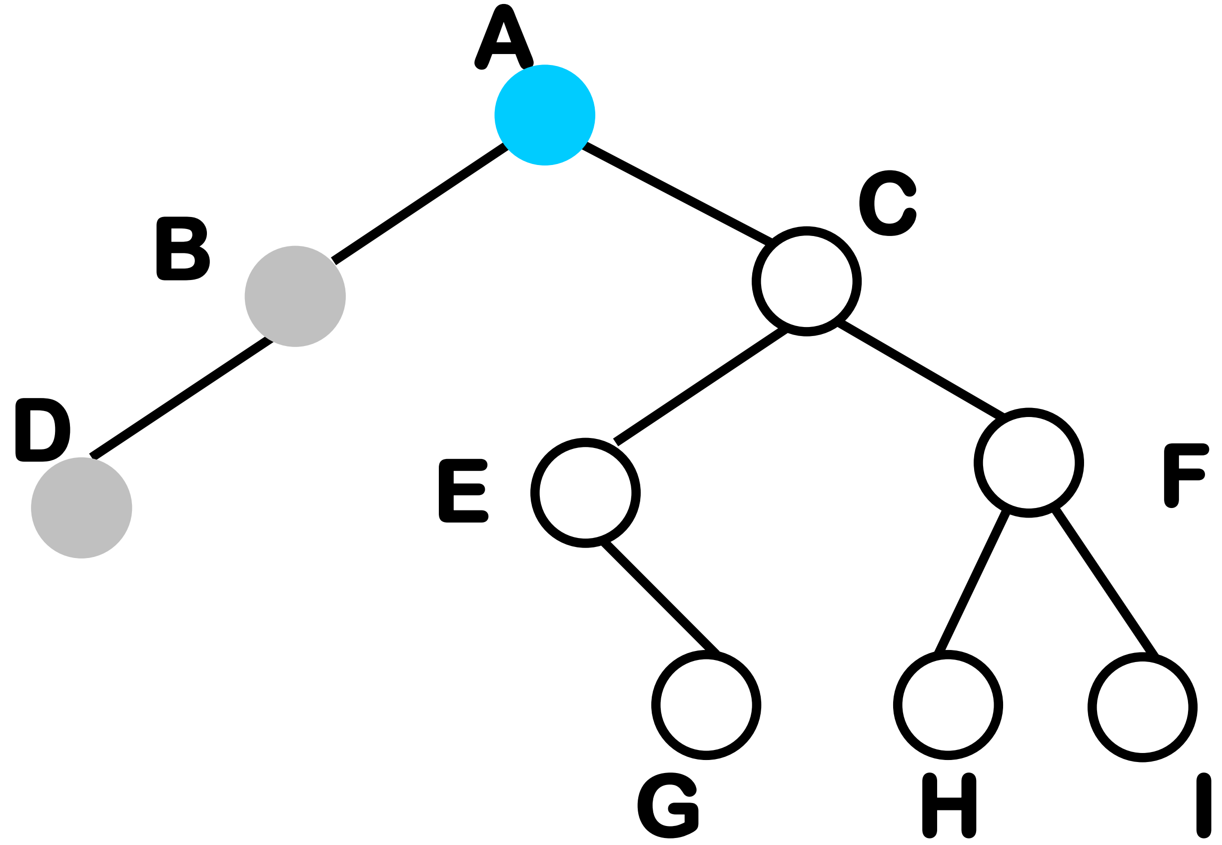
栈中结点

出栈结点

5.2 二叉树的抽象数据类型

后序序列　D　B　G

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

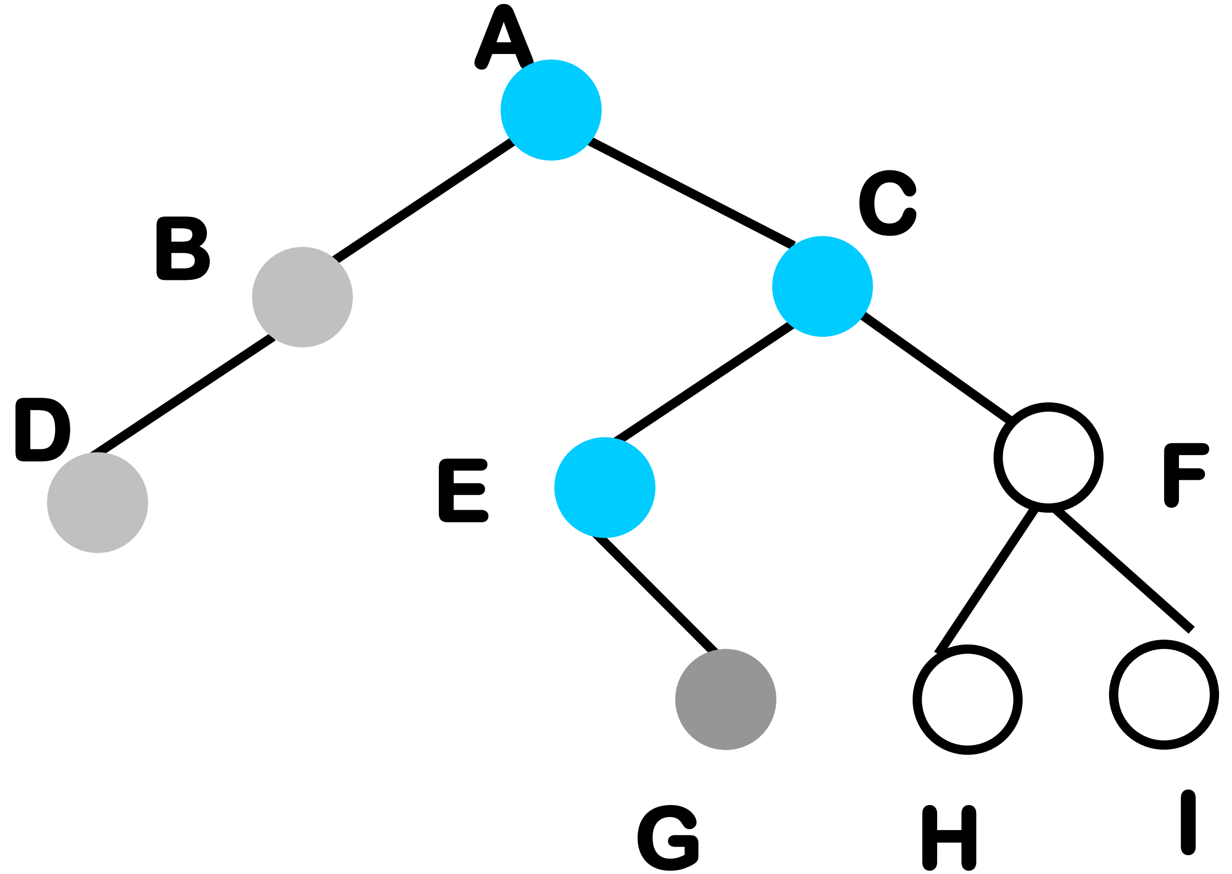
栈中结点

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

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

未访问结点

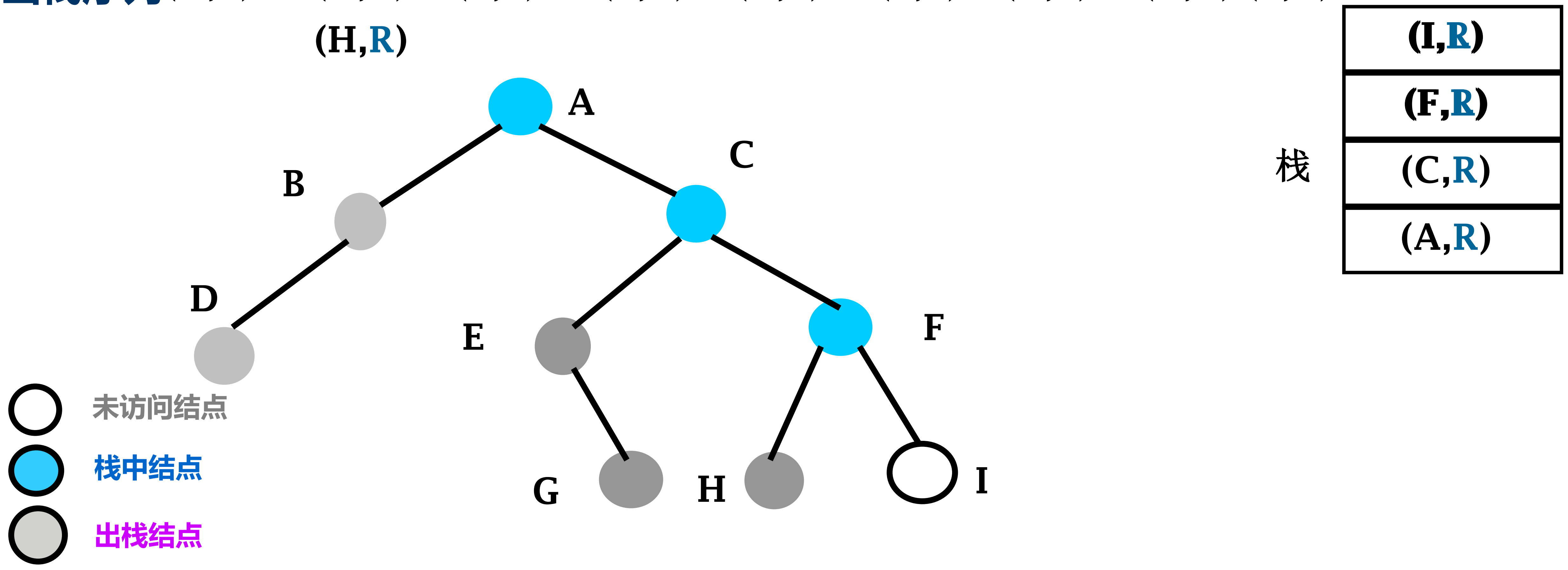
栈中结点

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 I F C A

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



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)

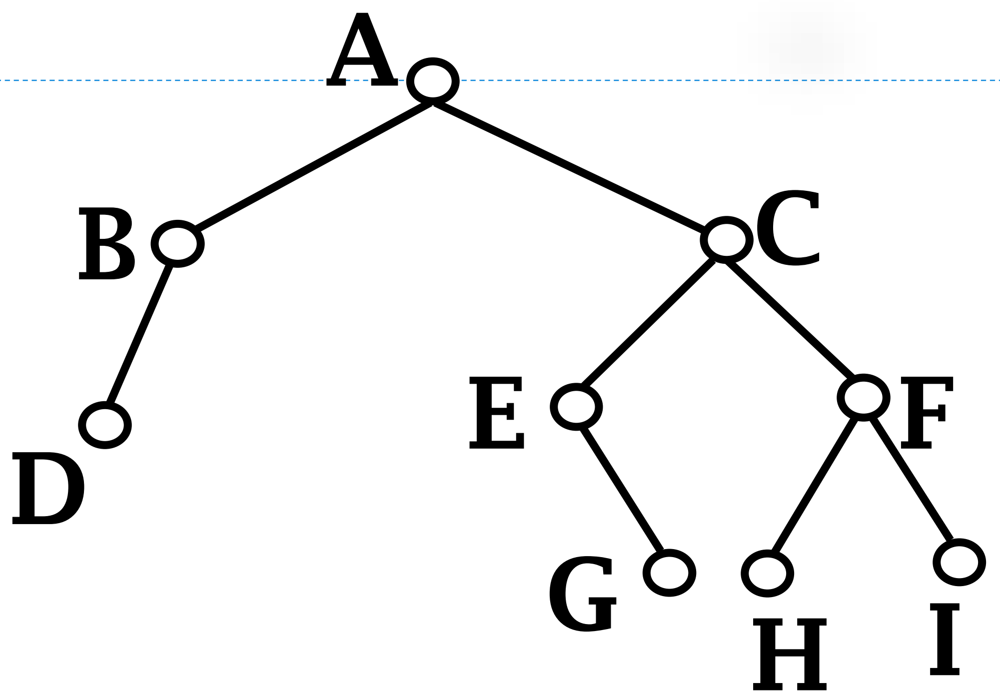
二叉树遍历算法的空间代价分析

- 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？

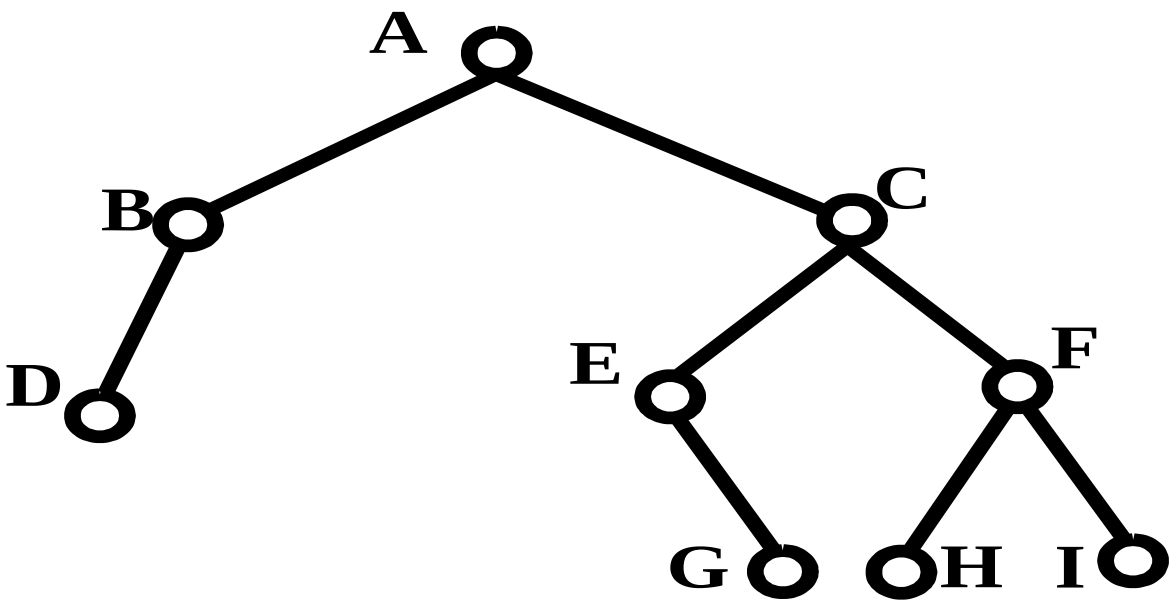
第五章 二叉树



- 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

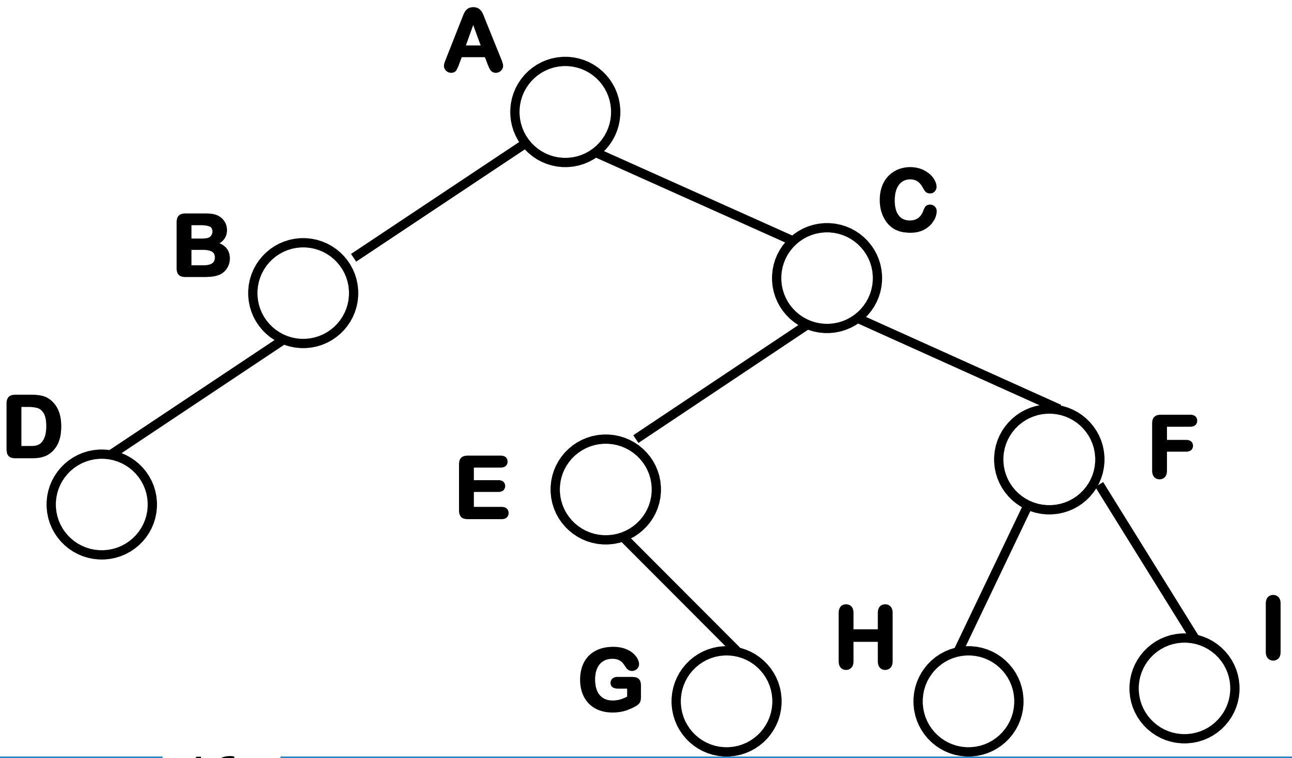


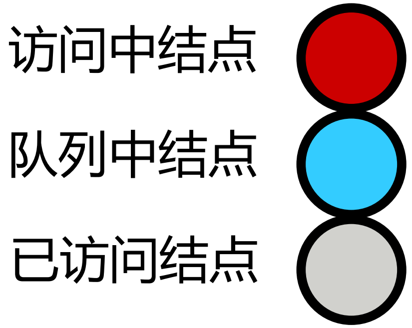
宽度优先遍历二叉树

BFS序列

队列

A	B	C	D	E	F	G	H	I





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)

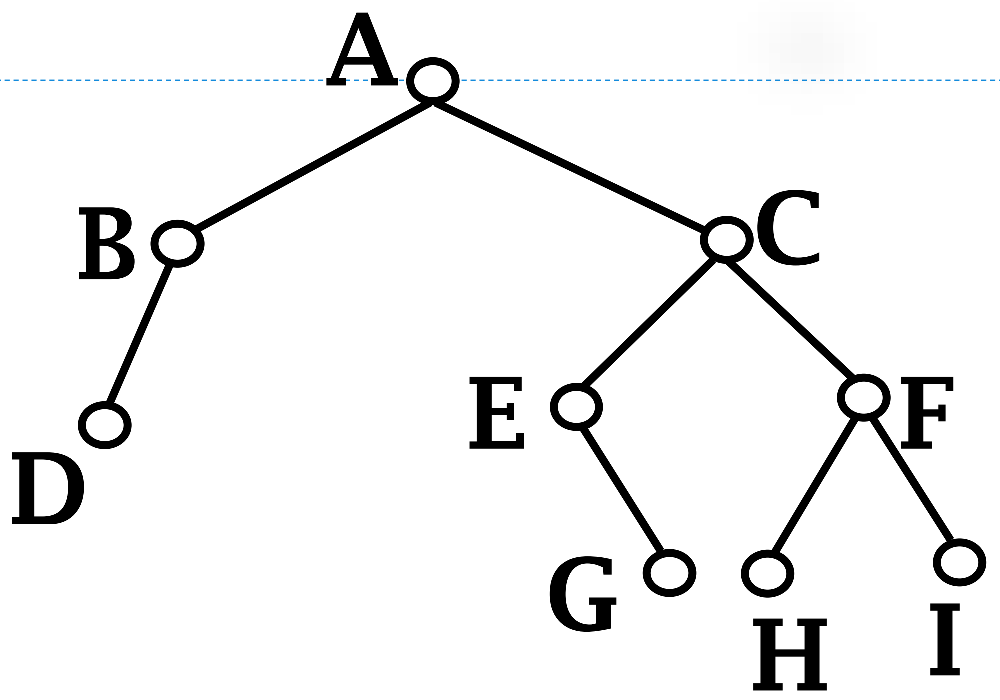
二叉树遍历算法的空间代价分析

- 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

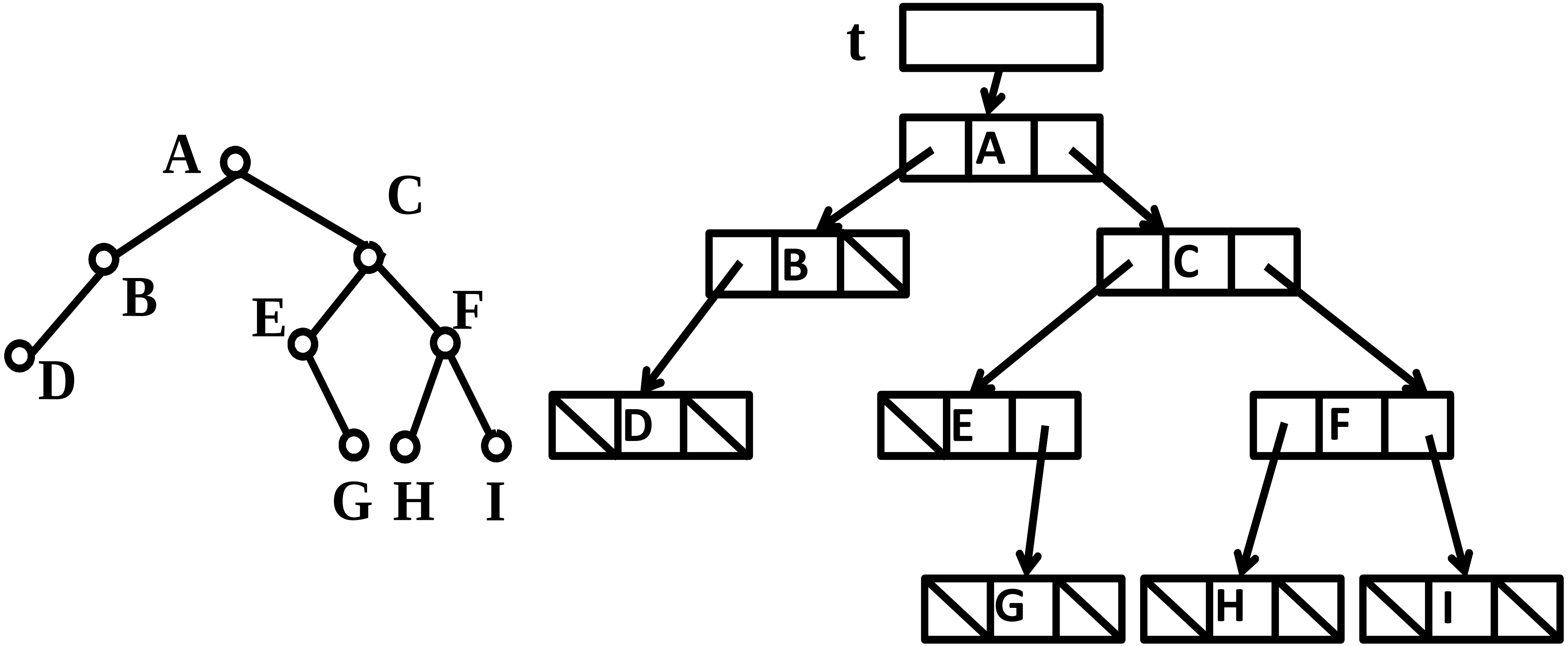
left	info	right

- 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

left	info	parent	right

5.3 二叉树的存储结构

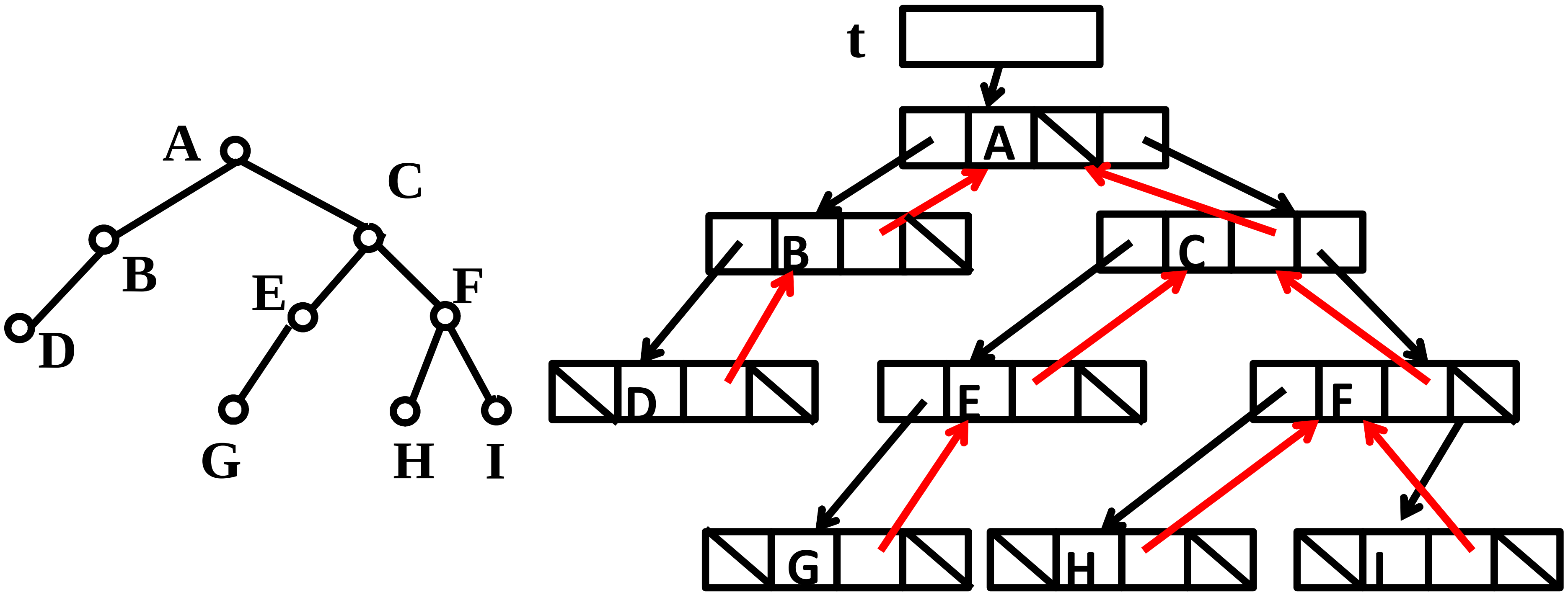
二叉链表



5.3 二叉树的存储结构

"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

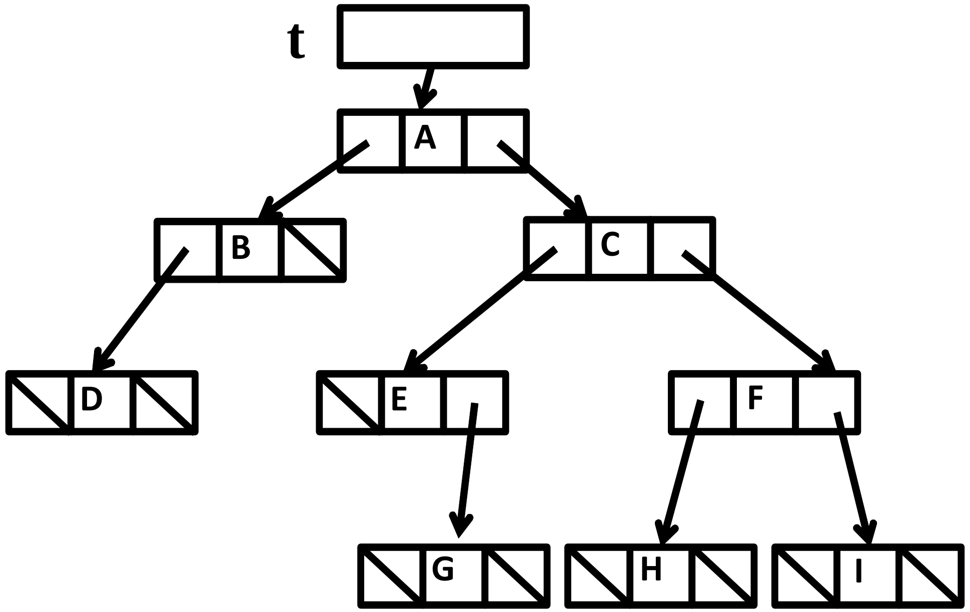
- 存储密度 α (≤1) 表示数据结构存储的效率

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

- 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

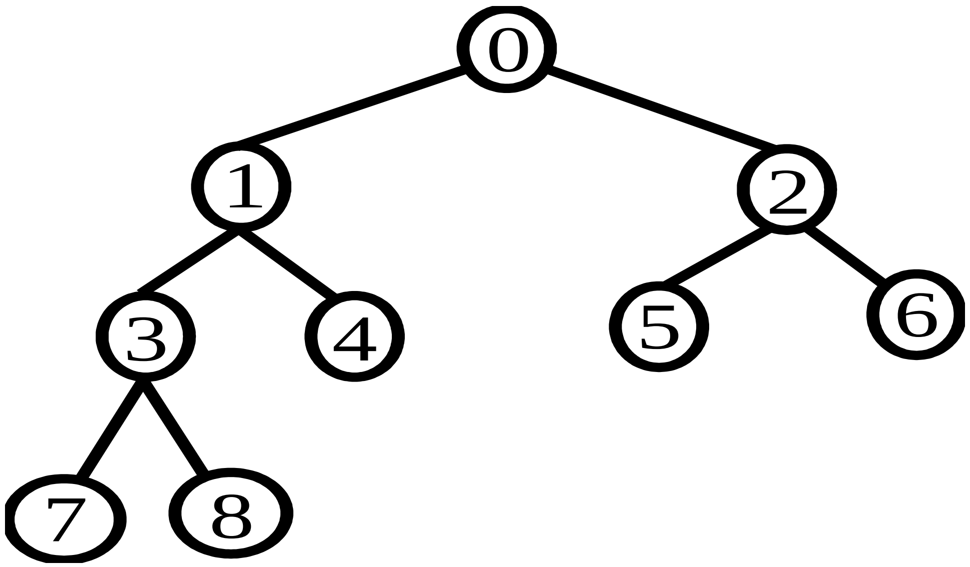
- C++ 可以用两种方法来实现不同的分支与叶结点：
  - 用union联合类型定义
  - 使用C++的子类来分别实现分支结点与叶结点，并采用虚函数isLeaf来区别分支结点与叶结点
- 早期节省内存资源
  - 利用结点指针的一个空闲位（一个bit）来标记结点所属的类型
  - 利用指向叶的指针或者叶中的指针域来存储该叶结点的值

完全二叉树的顺序存储结构

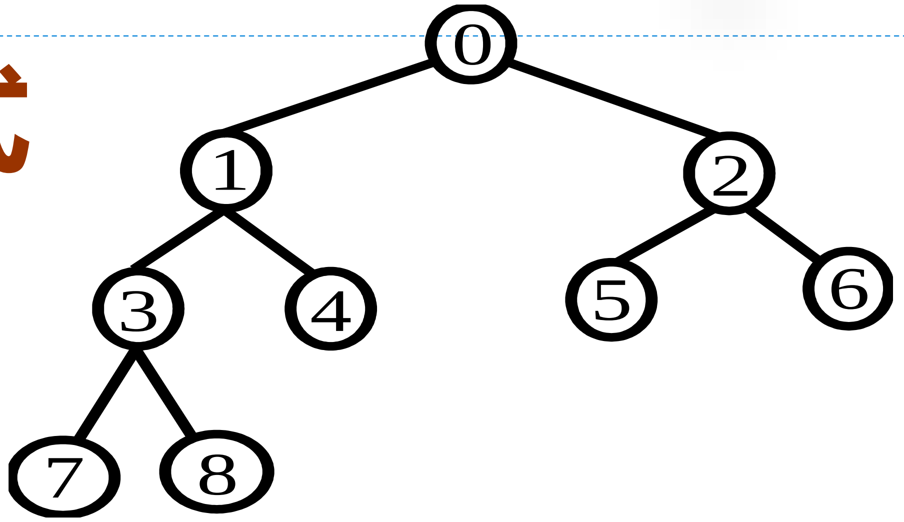
- 顺序方法存储二叉树
  - 把结点按一定的顺序存储到一片连续的存储单元
  - 使结点在序列中的位置反映出相应的结构信息
- 存储结构上是线性的

0	1	2	3	4	5	6	7	8

- 逻辑结构上它仍然是二叉树形结构



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

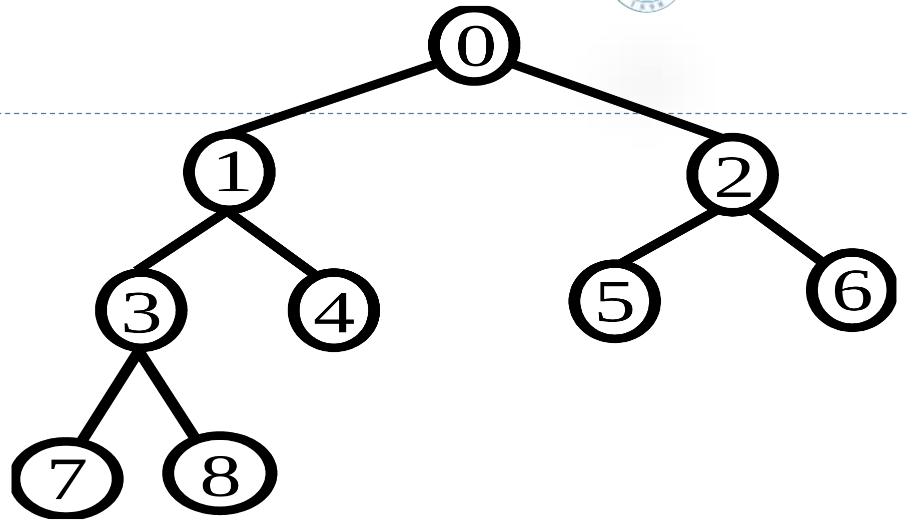
  - 当 2i+1<n 时，结点 i 的左孩子是结点 2i+1，

    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 $\lfloor (i-1)/2 \rfloor$
- 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？

数据结构与算法

谢谢聆听

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