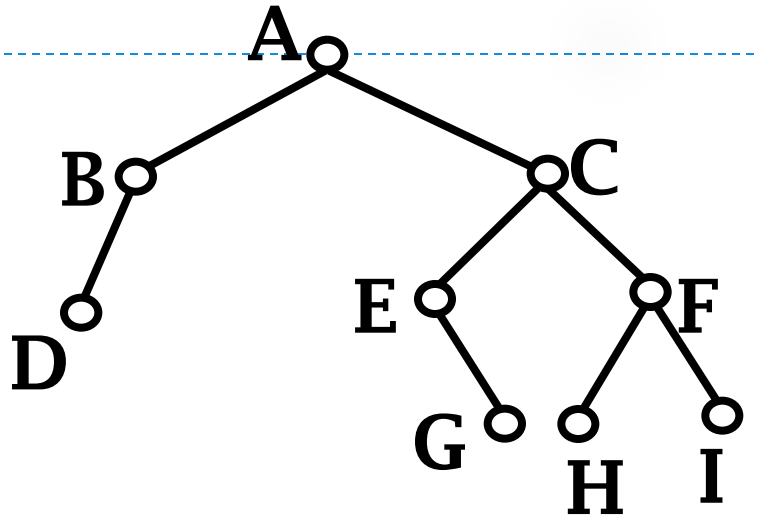
数据结构与算法(五)

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

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。

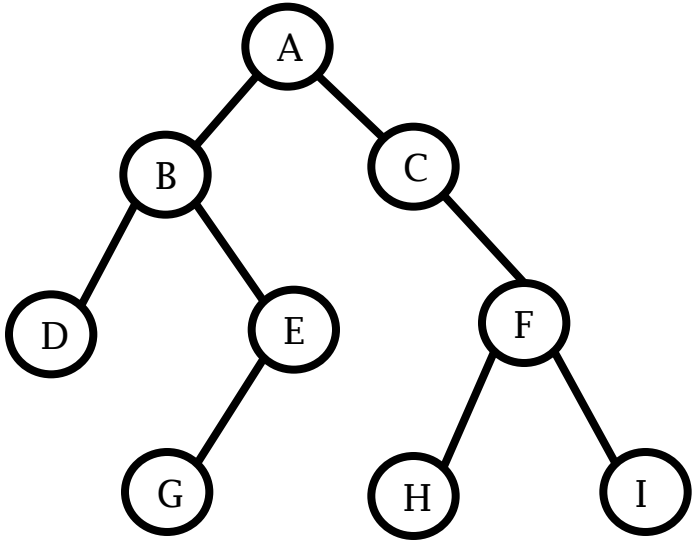
(2) **中序法 (LtR次序，inorder traversal)。**

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。

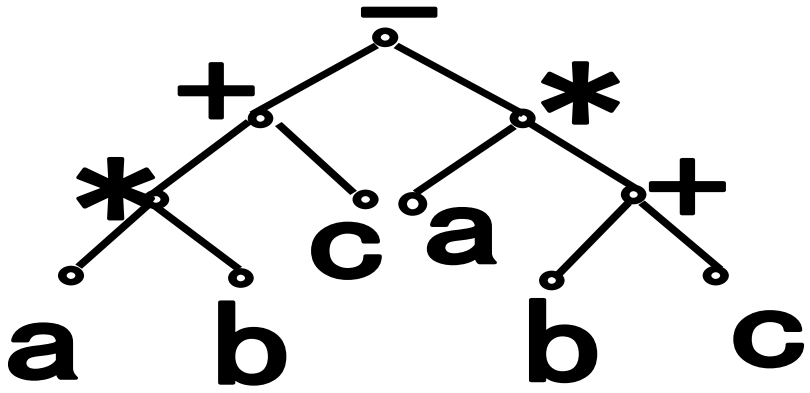
深度优先遍历二叉树



- 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
            epthOrder(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 _____。

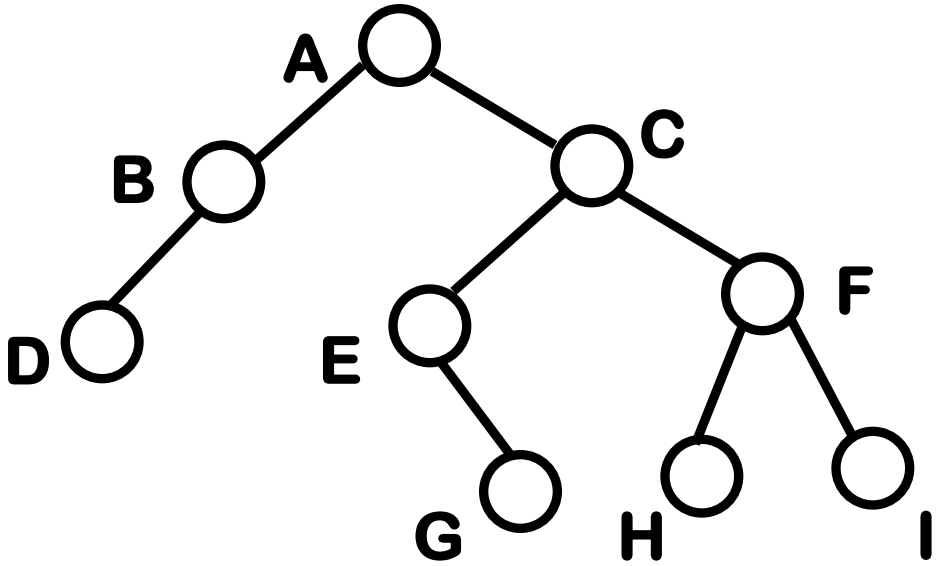
DFS遍历二叉树的非递归算法

- 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

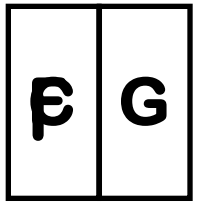
前序序列 A B D E H

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。

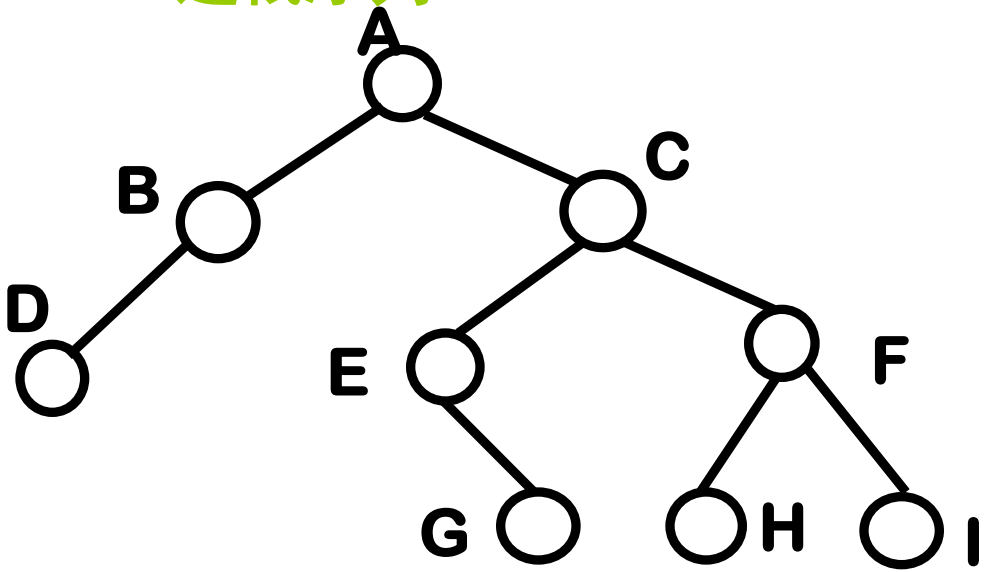
```
template<class T>  void
BinaryTree<T>::PreOrderWithoutRecusion
(BinaryTreeNode<T>* root) {
```

```
    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 }
} }
```

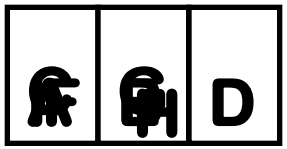
5.2 二叉树的抽象数据类型

中序序列

进栈序列 A B D C E G F H I



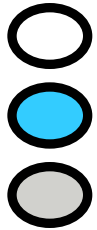
栈



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

```
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
}
```

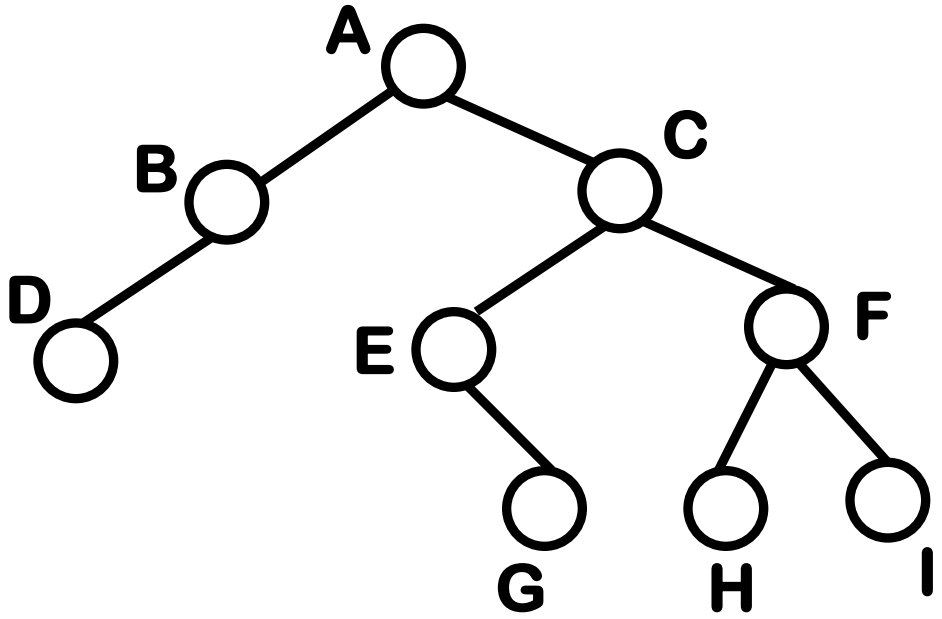
非递归后序遍历二叉树

- 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

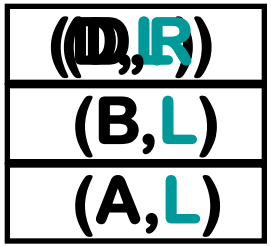
非递归后序遍历二叉树

后序序列　D

出栈序列



栈



未访问结点

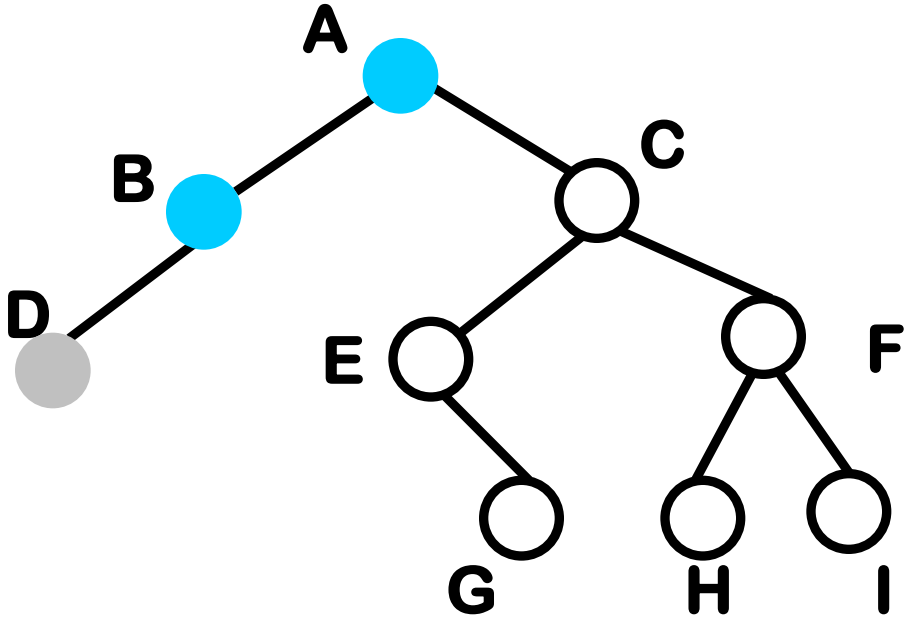
栈中结点

出栈结点

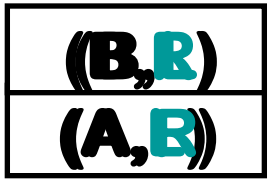
5.2 二叉树的抽象数据类型

后序序列 D B

出栈序列 (D,L) (D,R)



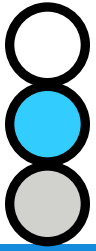
栈



未访问结点

栈中结点

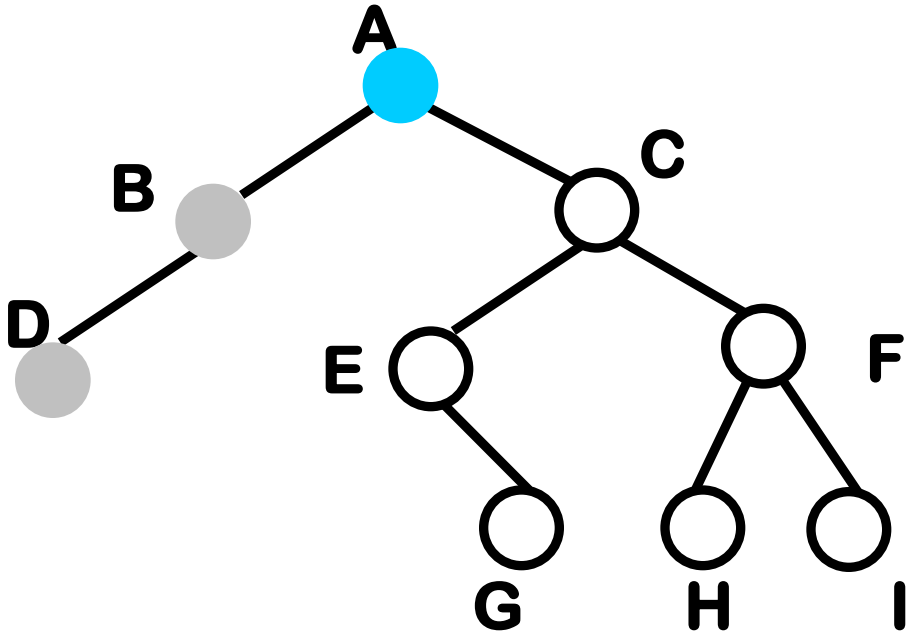
出栈结点



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

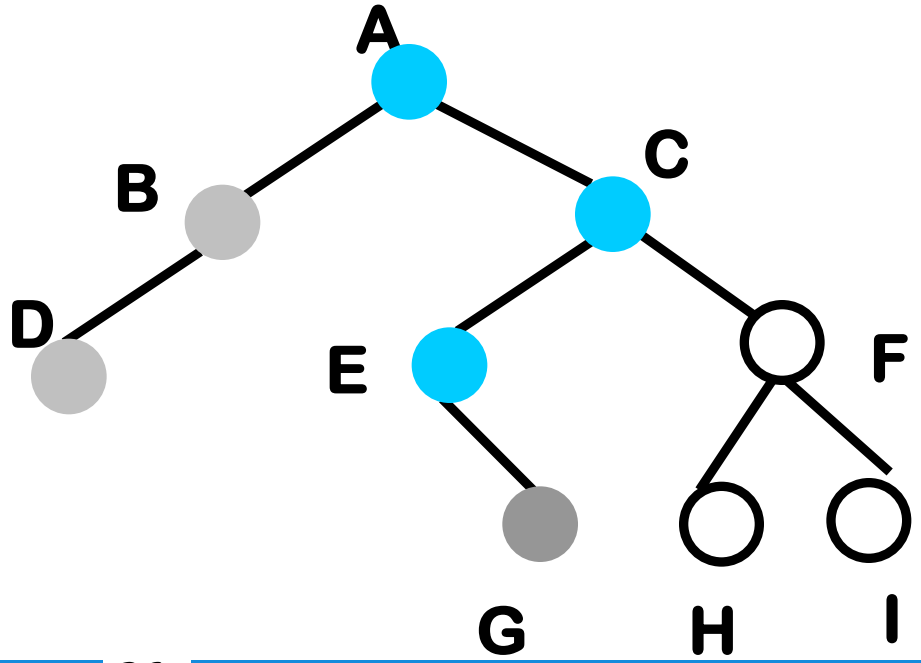
栈中结点

出栈结点

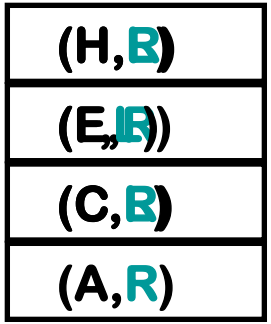
5.2 二叉树的抽象数据类型

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

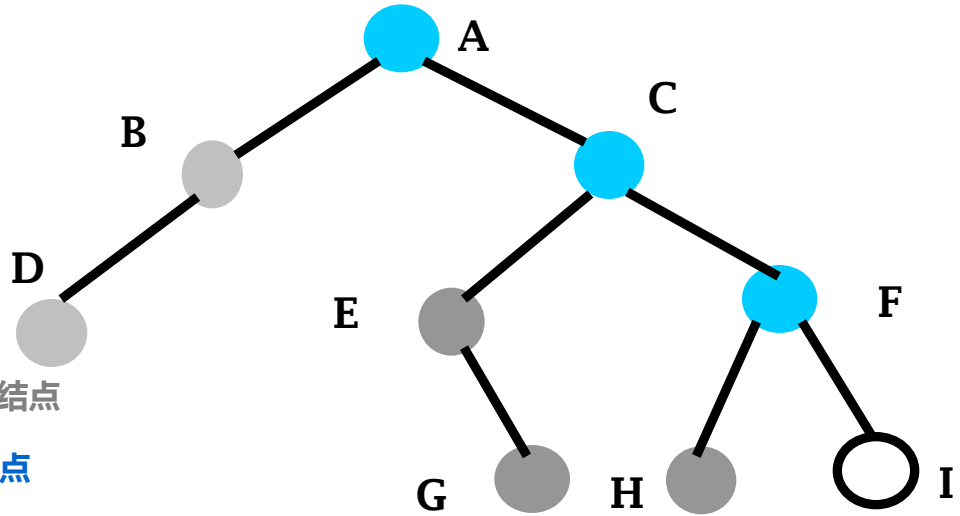
栈中结点

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？

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