数据结构与算法

树

张晓平

武汉大学数学与统计学院

2016 年 10 月 25 日

1. 遍历二叉树及其应用

树

- 树型结构是一类非常重要的非线性结构。直观地，树型结构是以分支关系定义的层次结构。
- 树在计算机领域中也有着广泛的应用，例如在编译程序中，用树来表示源程序的语法结构；在数据库系统中，可用树来组织信息；在文件系统中，可用树来组织文件。

1. 遍历二叉树及其应用

遍历二叉树及其应用

定义 (遍历二叉树 - Traversing Binary Tree) 按某种次序依次访问二叉树中的所有结点，使得每个结点被访问一次且只被访问一次。

注

- 关键词： 访问和次序。
- 访问指的是要根据实际的需要来确定具体做什么，比如对每个结点做相关计算、输出打印等。

遍历二叉树及其应用

若以 L、D、R 分别表示遍历左子树、遍历根结点和遍历右子树，则有六种遍历方案：

DLR、LDR、LRD、DRL、RDL、RLD。

若规定先左后右，则只有前三种情况，分别是：

- DLR - 前序遍历
- LDR - 中序遍历
- LRD - 后序遍历

对于二叉树的遍历，分别讨论递归遍历和非递归遍历。

- 递归遍历结构清晰，但初学者较难理解。递归算法通过使用栈来实现。
- 非递归遍历由设计者自行定义。

前序遍历

规则：

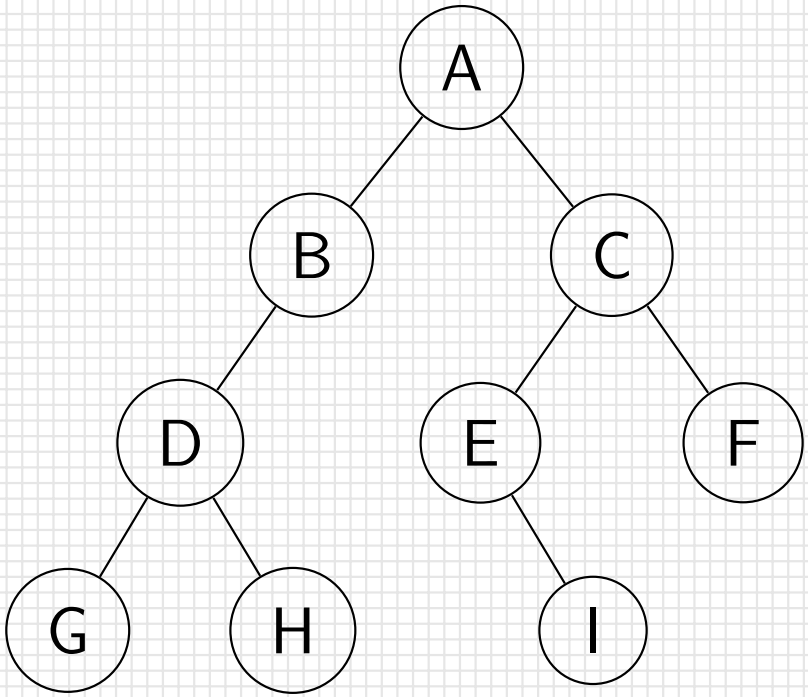
若二叉树为空，则遍历结束；否则

(1) 访问根结点；

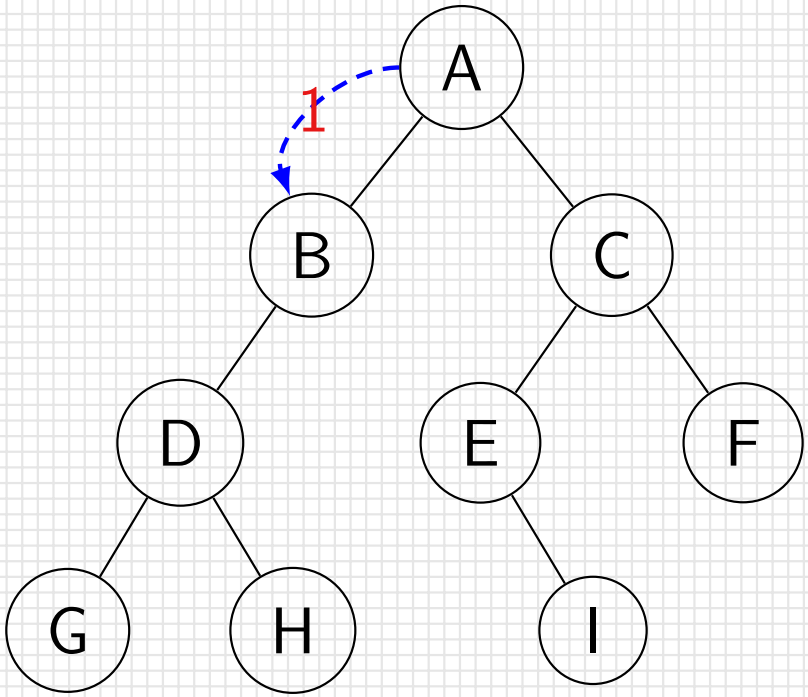
(2) 前序遍历左子树；

(3) 前序遍历右子树。

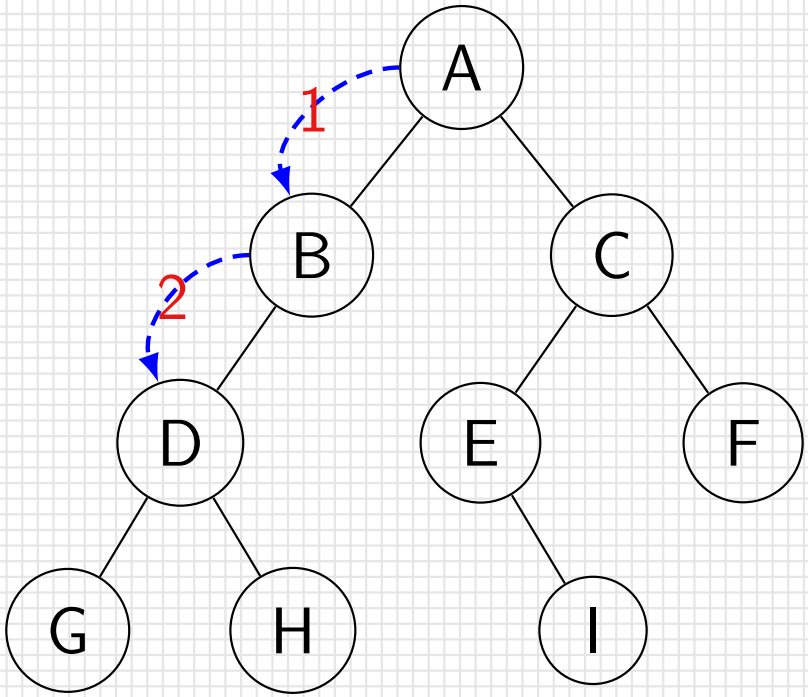
前序遍历



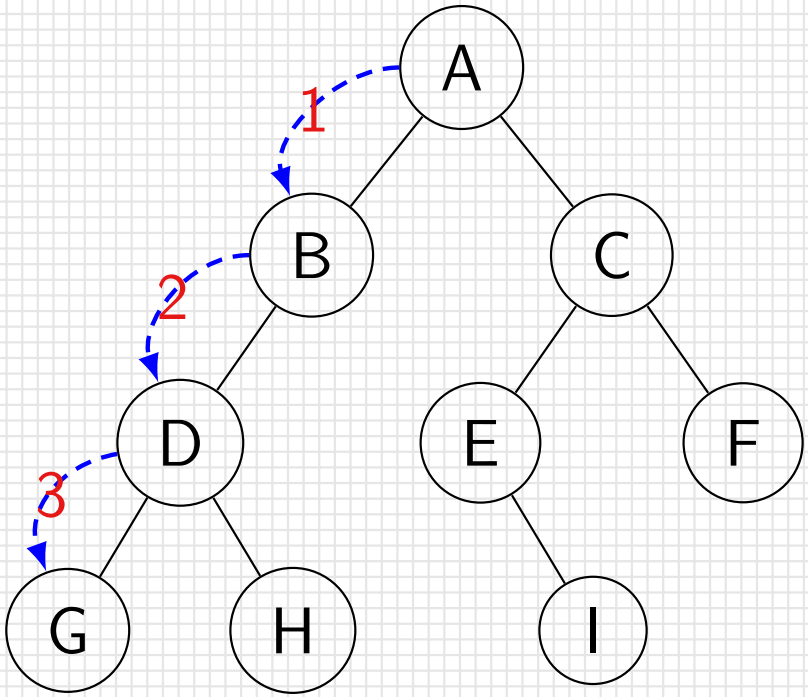
前序遍历



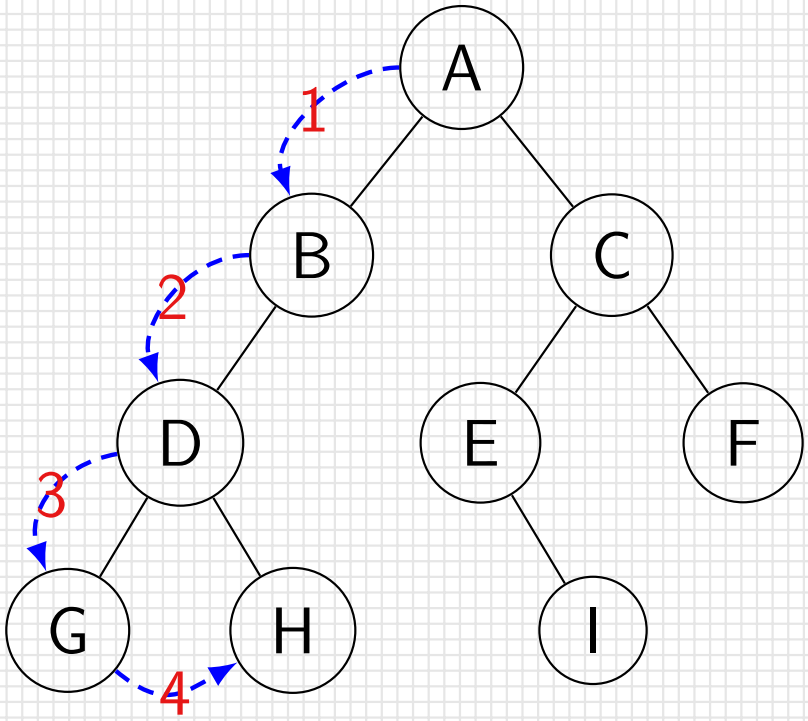
前序遍历



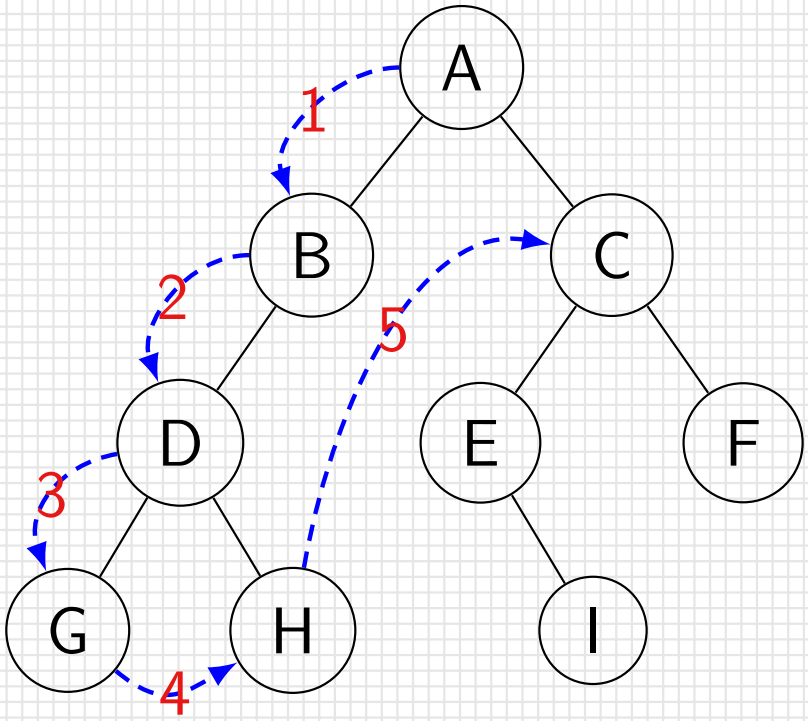
前序遍历



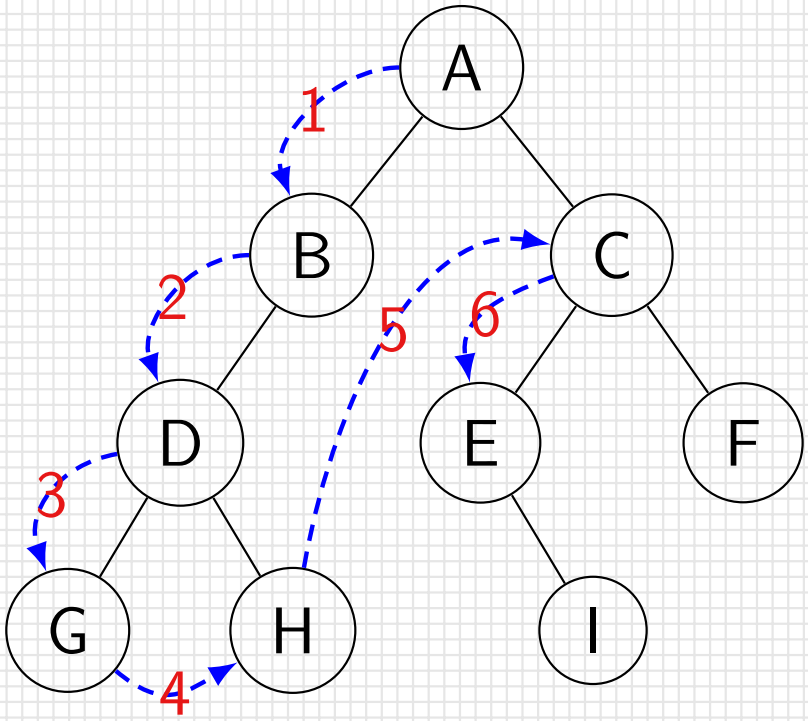
前序遍历



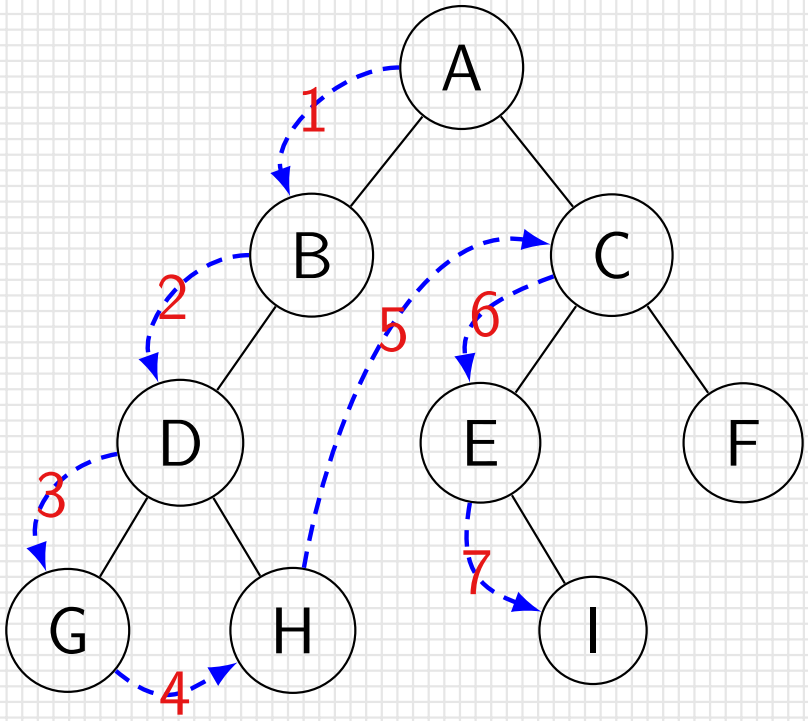
前序遍历



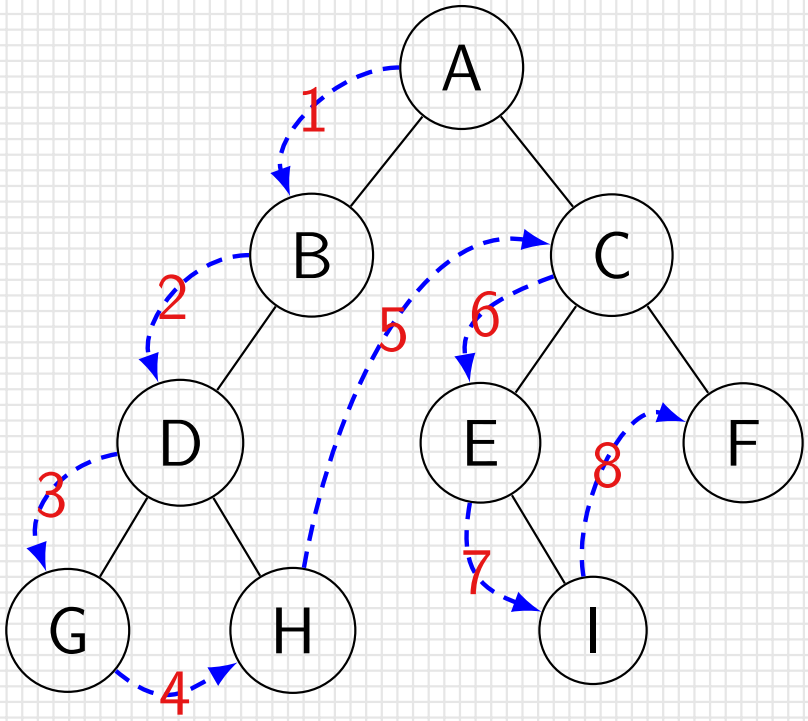
前序遍历



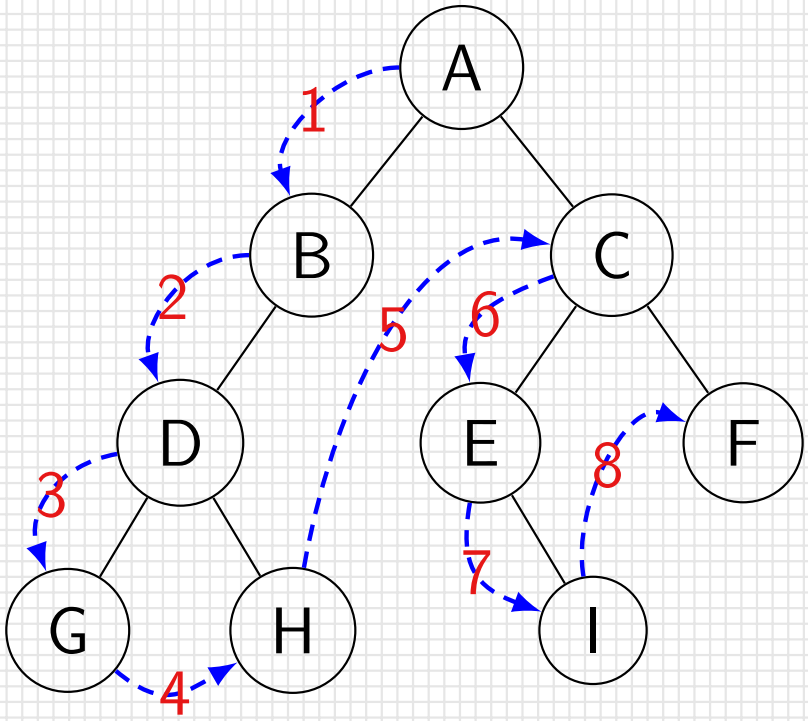
前序遍历



前序遍历



前序遍历



前序遍历结果： A B D G H C E I F

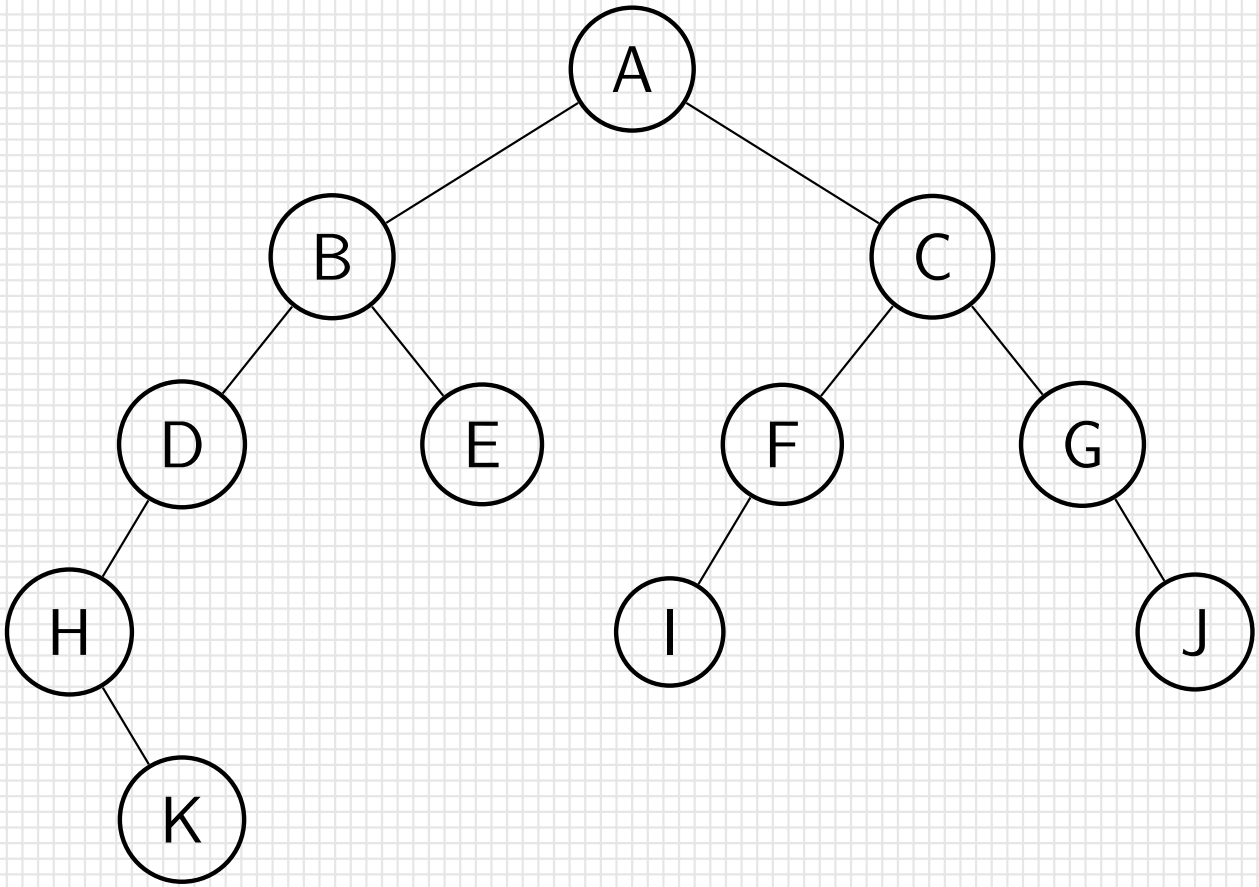
前序遍历

```
#include"BiTree.h"

/* PreOrder Traverse a BiTree */
void PreOrderTraverse(BiTree tree, void(* visit)())
{
  BTNode * pnode = tree;
  if (pnode)
  {
    visit(pnode->data);
    PreOrderTraverse(pnode->lchild, visit);
    PreOrderTraverse(pnode->rchild, visit);
  }
}
```

前序遍历

前序遍历以下二叉树，并打印各结点的值。



前序遍历

假设 visit 函数实现打印功能，即

```
#include"BiTree.h"

void Print(ElemType item)
{
  printf("%d␣", item);
}
```

中序遍历

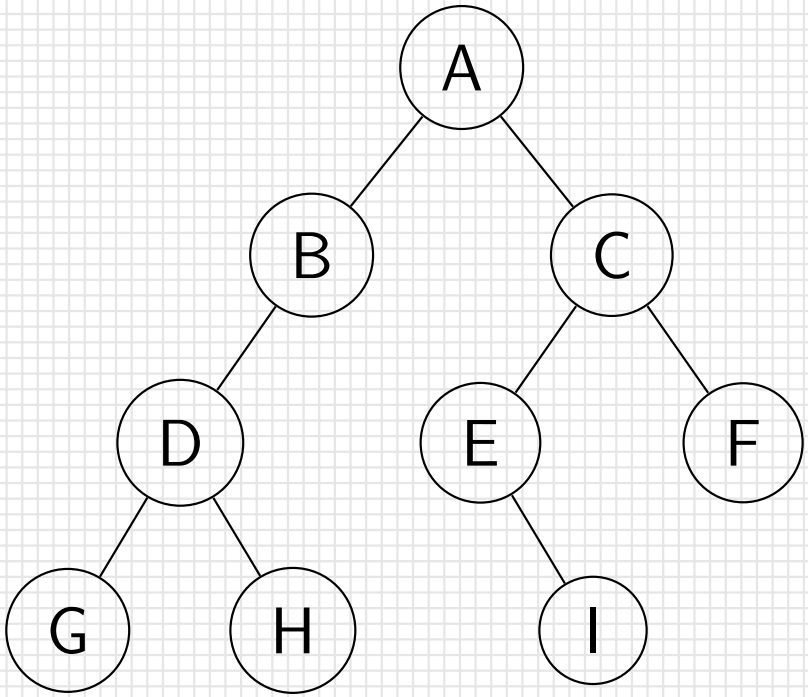
规则：若二叉树为空，则遍历结束；否则

(1) 中序遍历左子树；

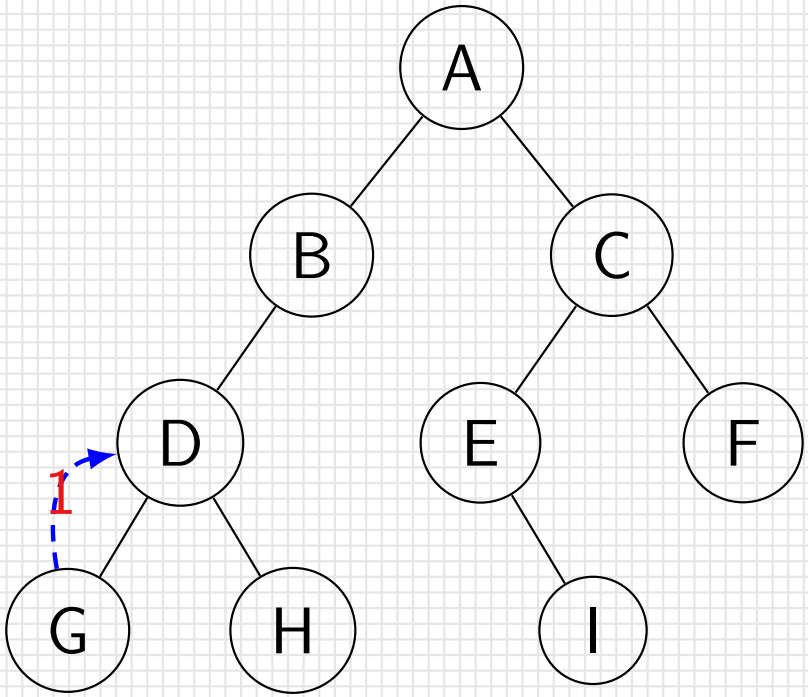
(2) 访问根结点；

(3) 中序遍历右子树。

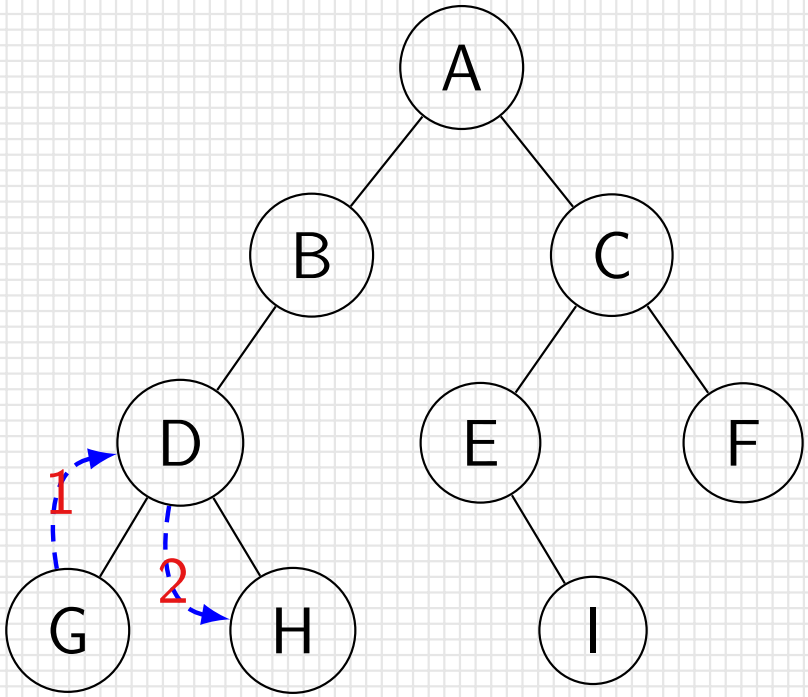
中序遍历



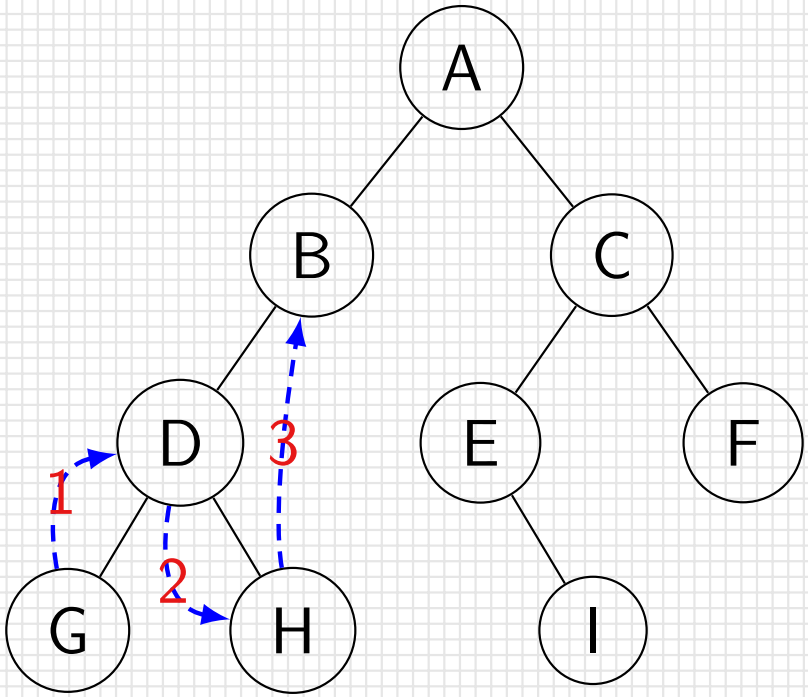
中序遍历



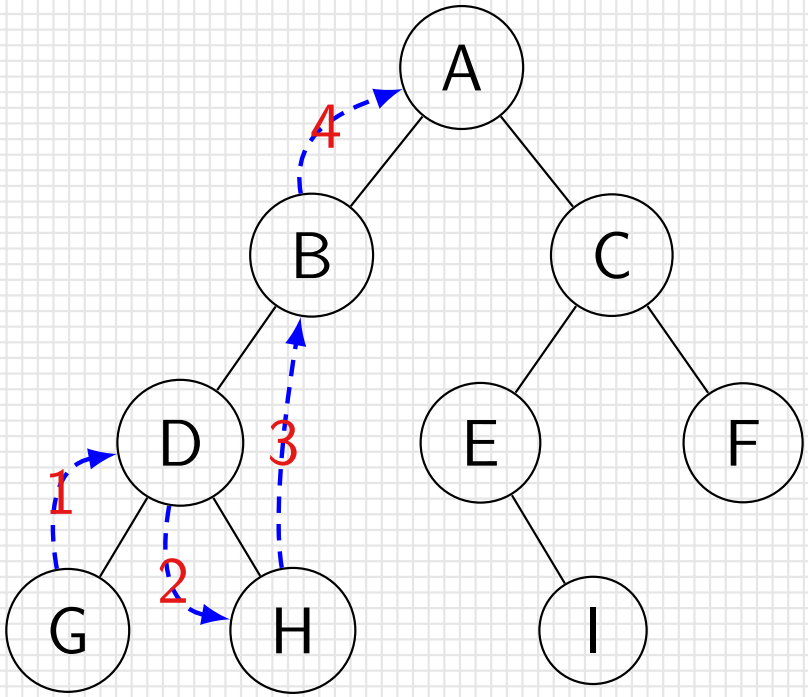
中序遍历



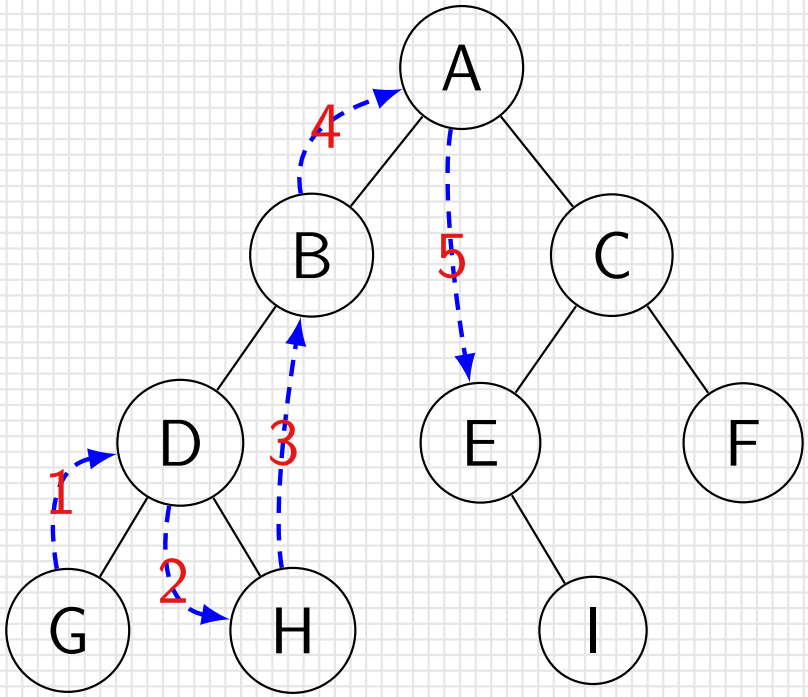
中序遍历



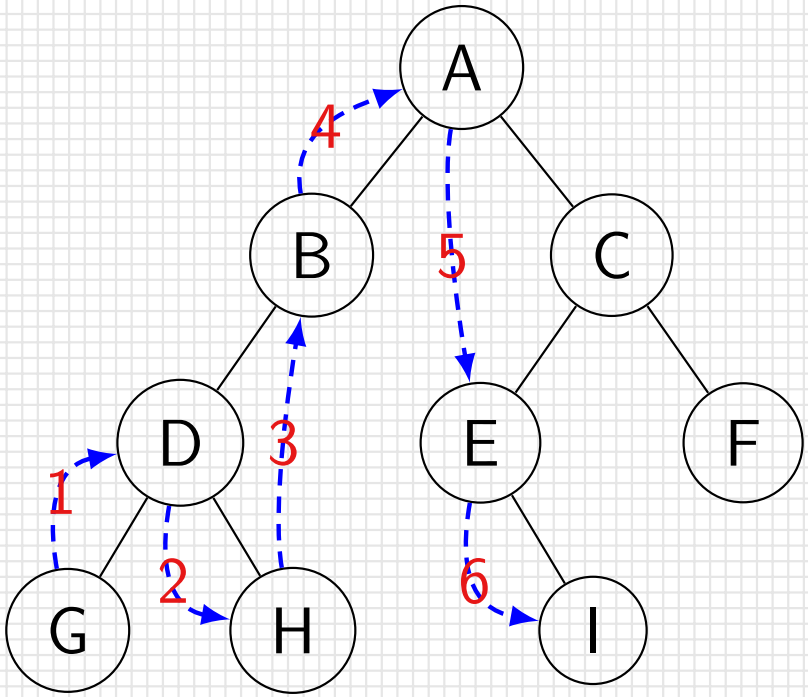
中序遍历



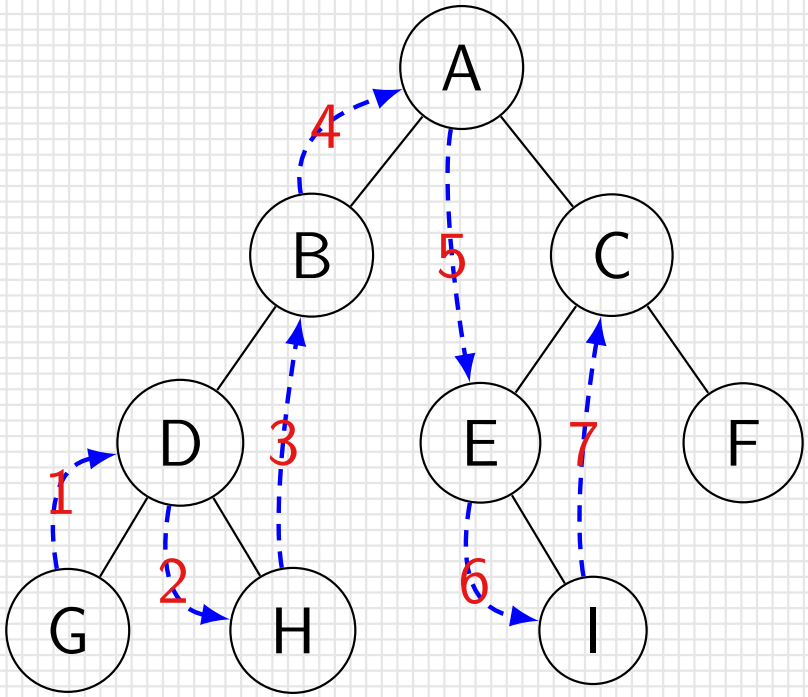
中序遍历



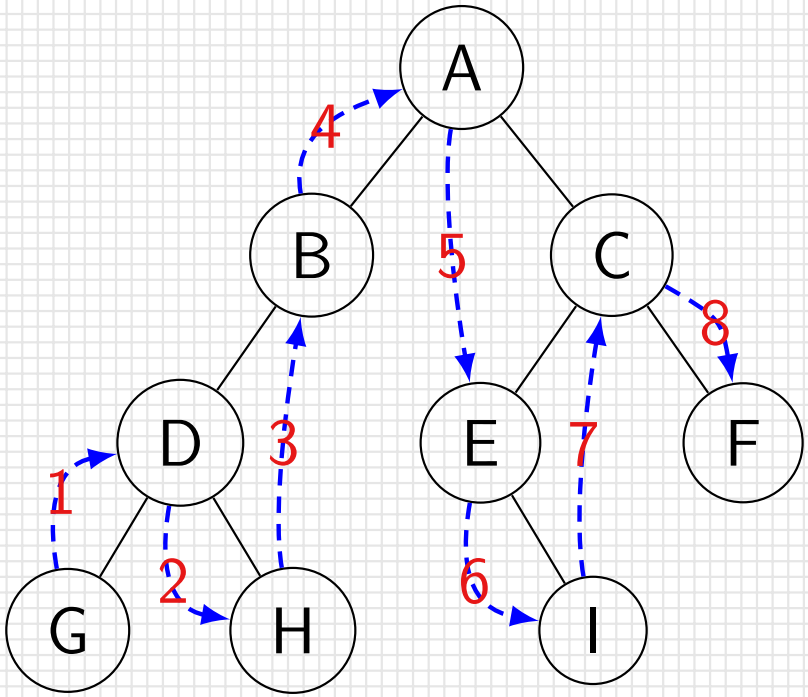
中序遍历



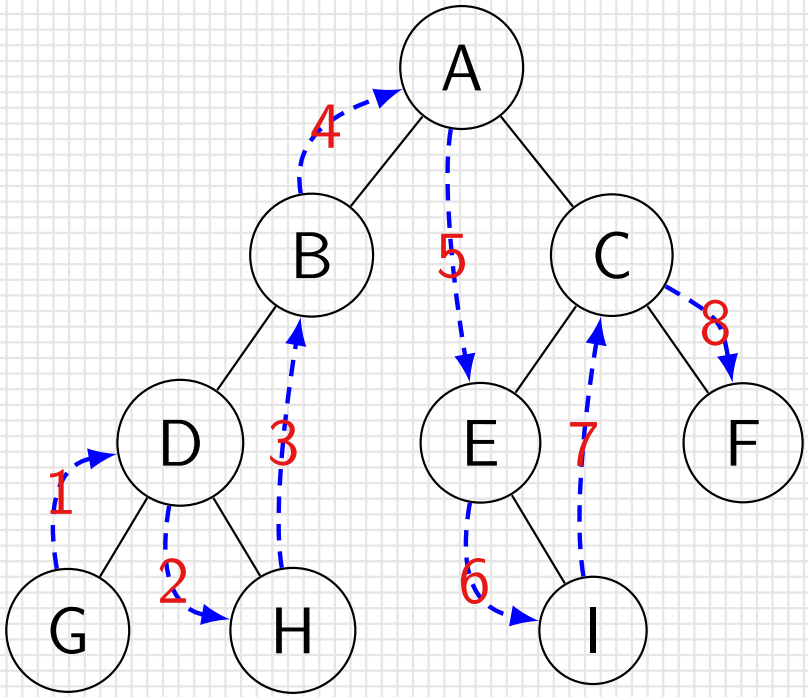
中序遍历



中序遍历



中序遍历



中序遍历结果： G D H B A E I C F

中序遍历

```
#include"BiTree.h"

/* InOrder Traverse a BiTree */
void InOrderTraverse(BiTree tree, void(* visit)())
{
  BTNode * pnode = tree;
  if (pnode)
  {
    InOrderTraverse(pnode->lchild, visit);
    visit(pnode->data);
    InOrderTraverse(pnode->rchild, visit);
  }
}
```

中序遍历

注　中序遍历，相对于前序遍历而言，只是把调用左孩子的递归函数提前了。

后序遍历

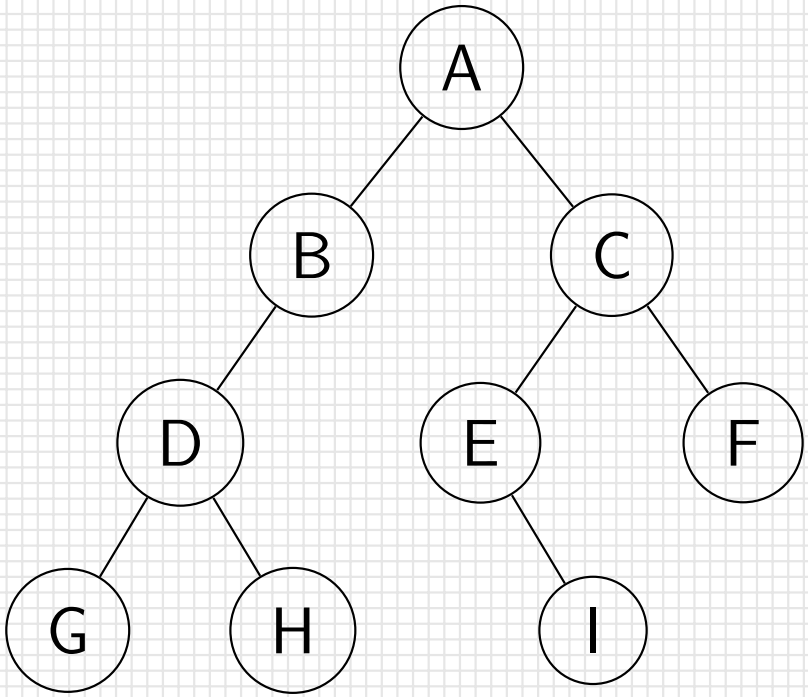
规则：若二叉树为空，则遍历结束；否则

(1) 后序遍历左子树；

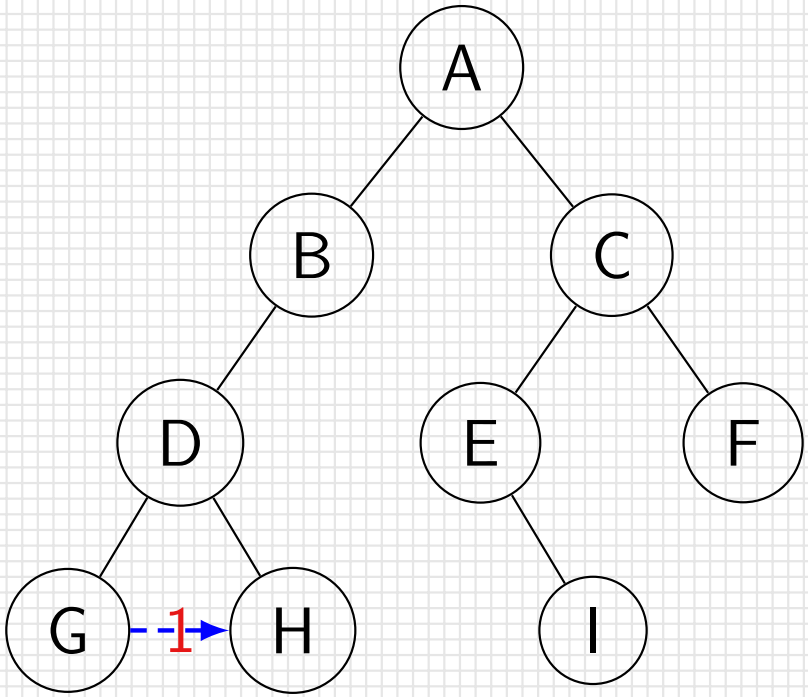
(2) 后序遍历右子树；

(3) 访问根结点。

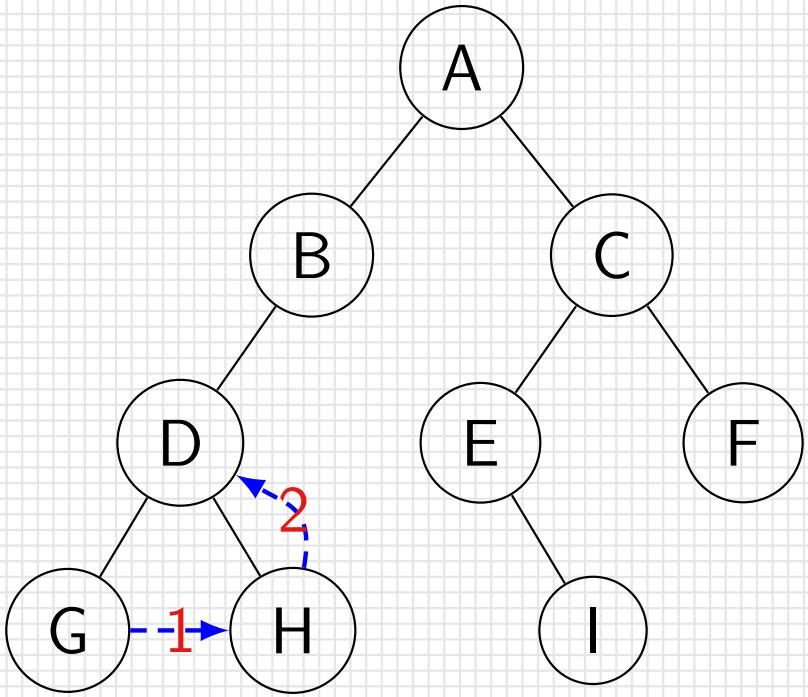
后序遍历



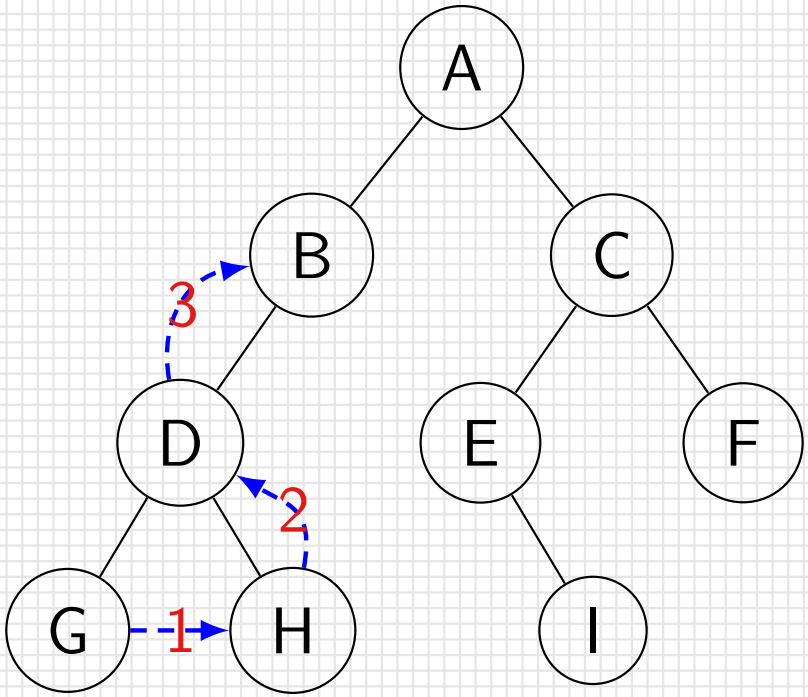
后序遍历



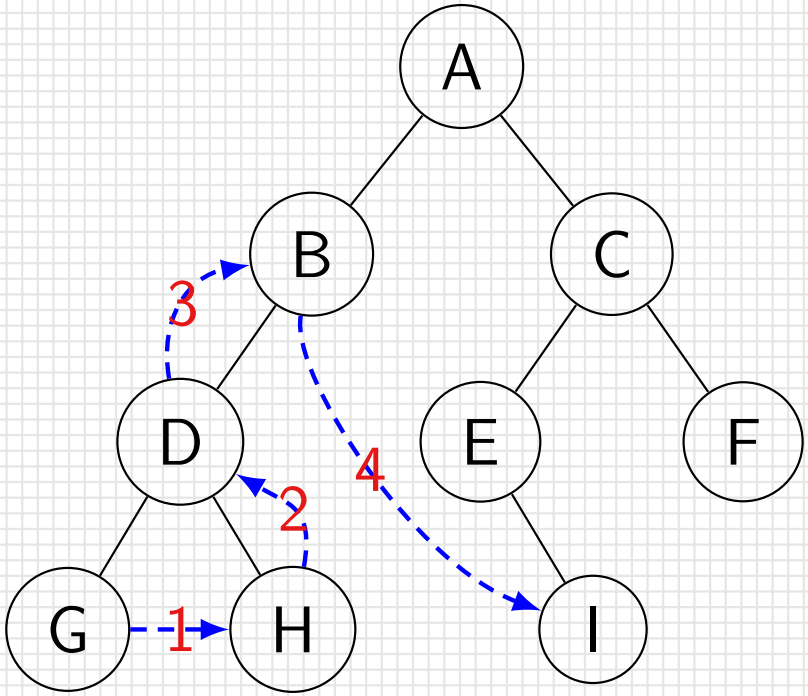
后序遍历



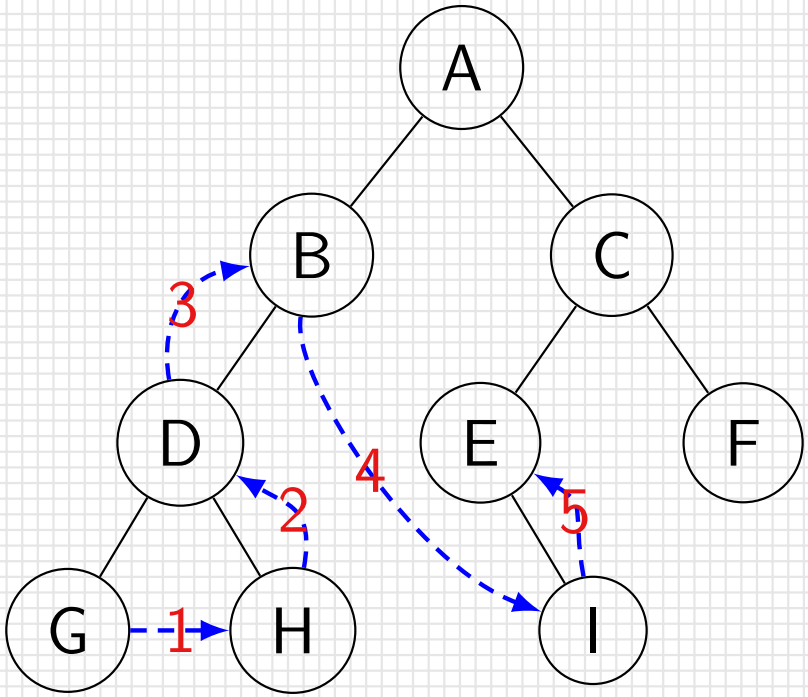
后序遍历



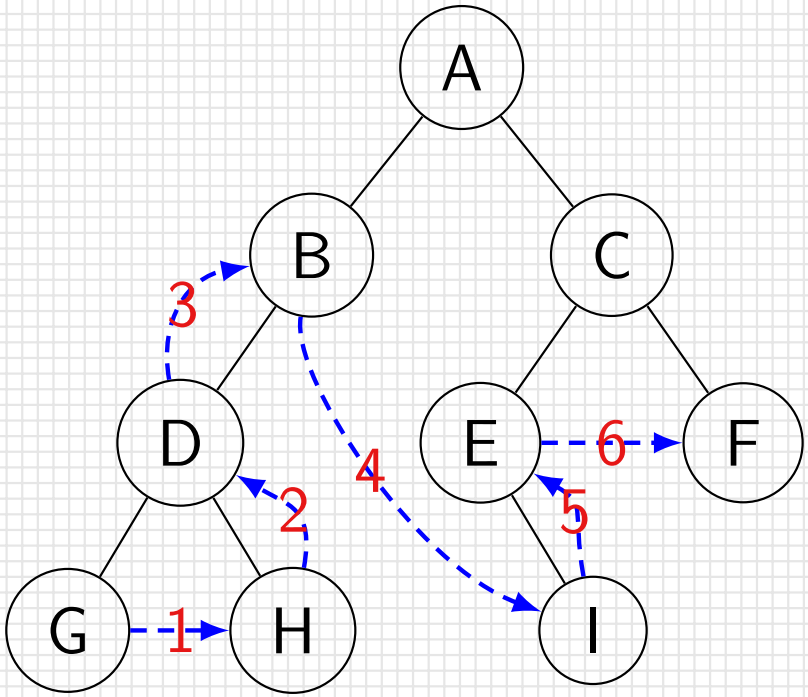
后序遍历



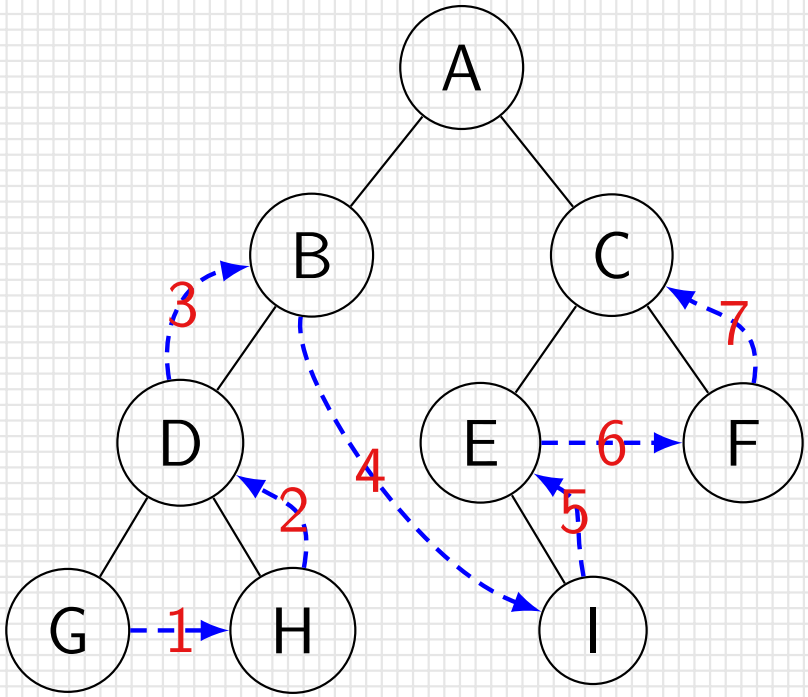
后序遍历



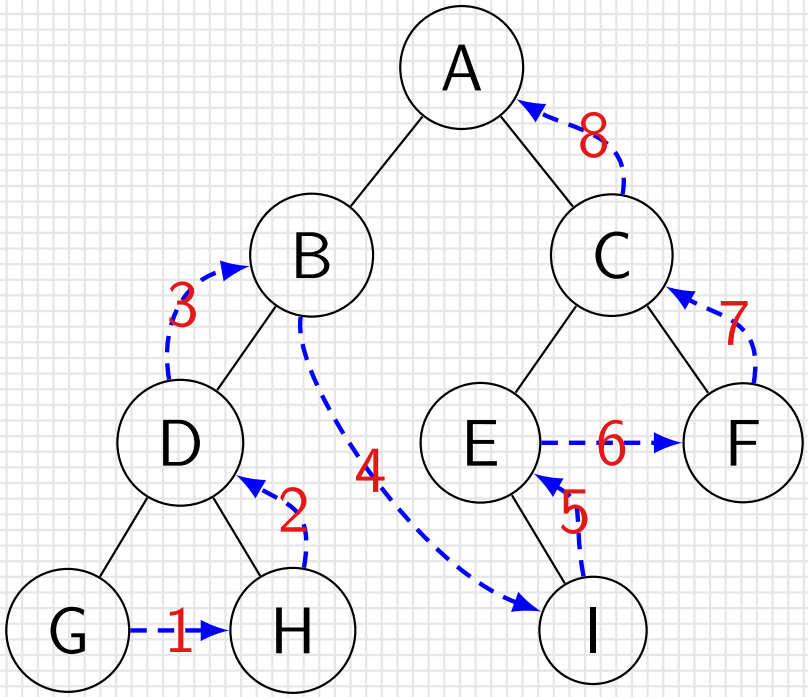
后序遍历



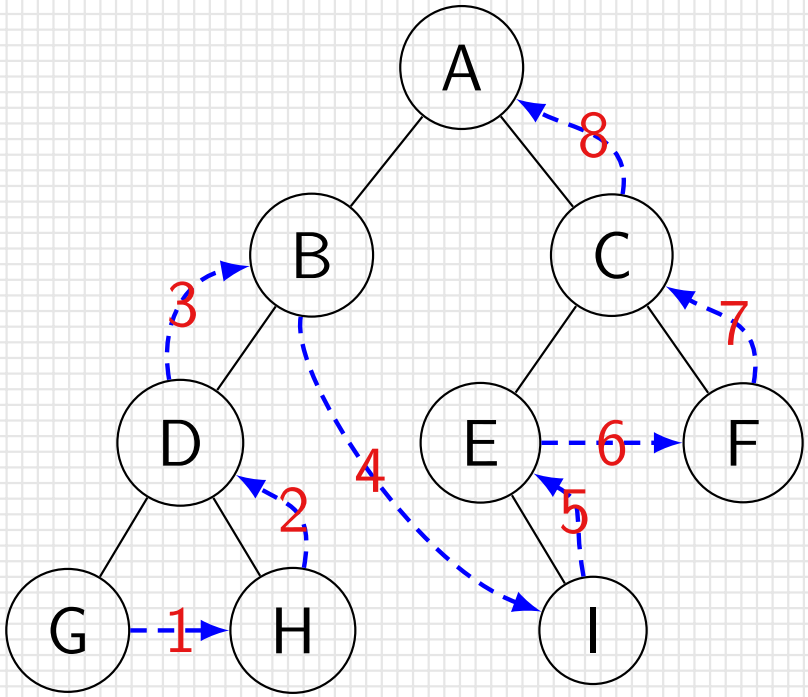
后序遍历



后序遍历



后序遍历



后序遍历结果： G H D B I E F C A

后序遍历

```
#include"BiTree.h"

/* PostOrder Traverse a BiTree */
void PostOrderTraverse(BiTree tree, void(* visit)())
{
  BTNode * pnode = tree;
  if (pnode)
  {
    PostOrderTraverse(pnode->lchild, visit);
    PostOrderTraverse(pnode->rchild, visit);
    visit(pnode->data);
  }
}
```

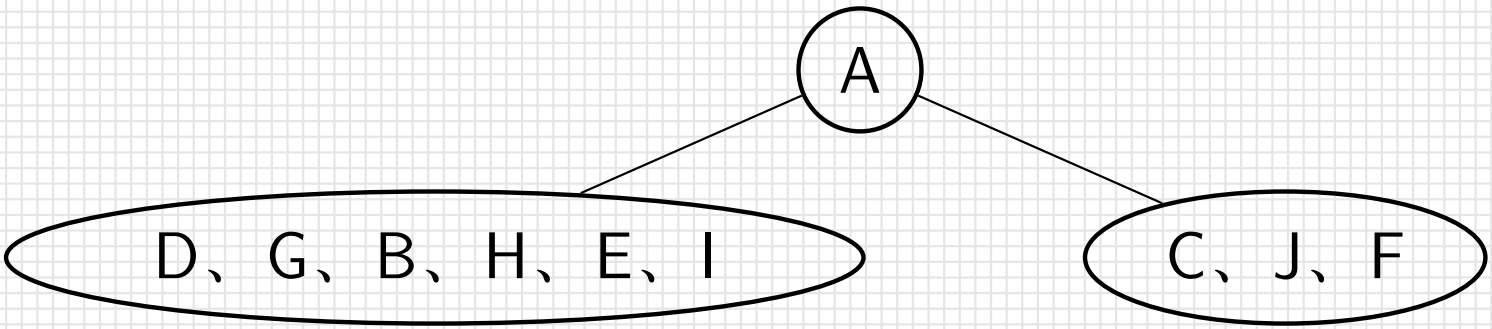
遍历二叉树及其应用

问题　已知一棵二叉树的中序遍历序列为 D、G、B、H、E、I、A、C、J、F，后序遍历序列为 G、D、H、I、E、B、J、F、C、A，请问这棵二叉树的前序遍历结果是多少？

遍历二叉树及其应用

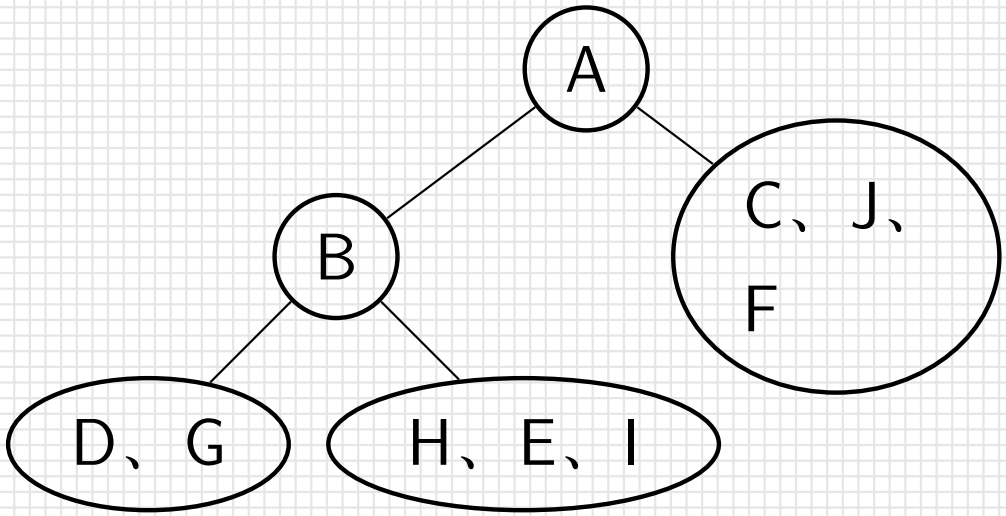
问题　已知一棵二叉树的中序遍历序列为 D、G、B、H、E、I、A、C、J、F，后序遍历序列为 G、D、H、I、E、B、J、F、C、A，请问这棵二叉树的前序遍历结果是多少？

解：1. 由后序遍历序列可知，A 为根结点。由中序遍历序列知，D、G、B、H、E、I 为 A 的左子树结点，C、J、F 为 A 的右子树结点。



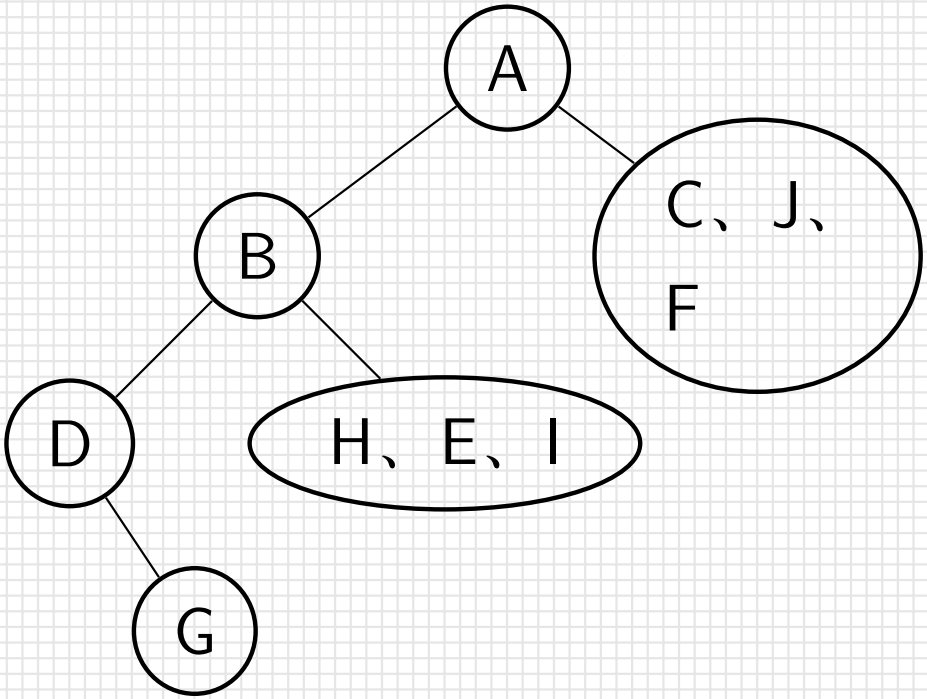
遍历二叉树及其应用

2. 看 A 的左子树，由中序序列 D、G、B、H、E、I 和后序序列 G、D、H、I、E、B，知 B 是该子树的根结点，且 D、G 为 B 的左子树结点，H、E、I 为 B 的右子树结点。



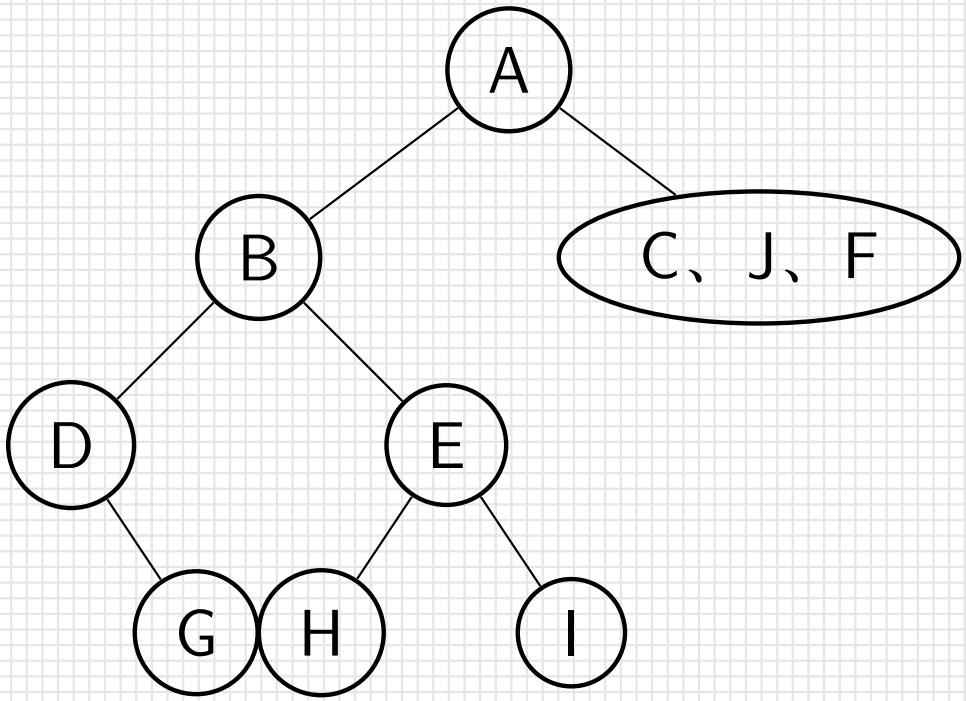
遍历二叉树及其应用

3. 看 B 的左子树，由中序序列 D、G 和后序序列 G、D 知，D 为该子树的根结点，且 G 为 D 的右孩子。



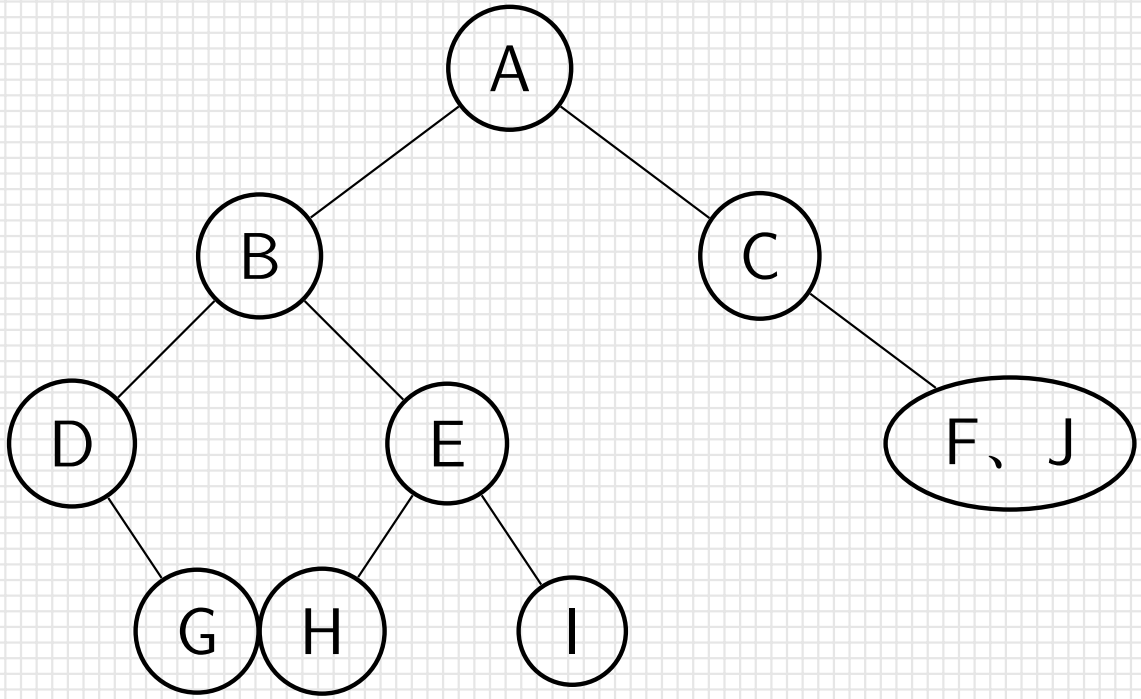
遍历二叉树及其应用

4. 看 B 的右子树，由中序序列 H、E、I 和后序序列 H、I、E 知，E 为该子树的根结点，且 H 为 E 的左孩子，I 为 E 的右孩子。



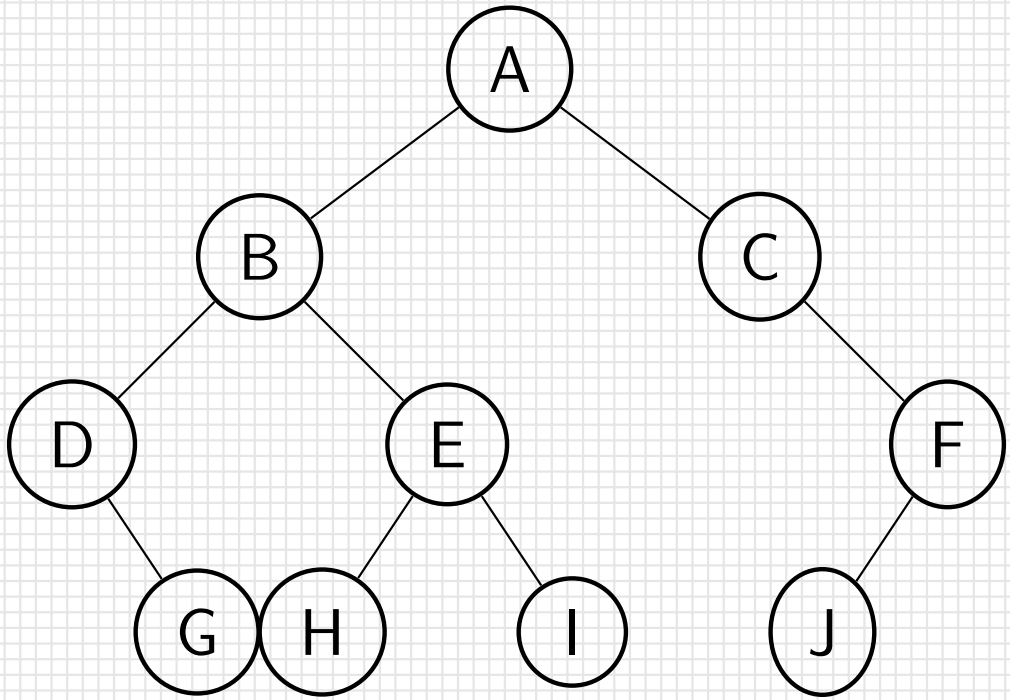
遍历二叉树及其应用

5. 看 A 的右子树，由中序序列 C、J、F 和后序序列 J、F、C 知，C 为该子树的根结点，J、F 为 C 的右子树结点。



遍历二叉树及其应用

6. 看 C 的右子树，由中序序列 J、F 和后序序列 J、F 知，F 为该子树的根结点，且 J 为 F 的左孩子。



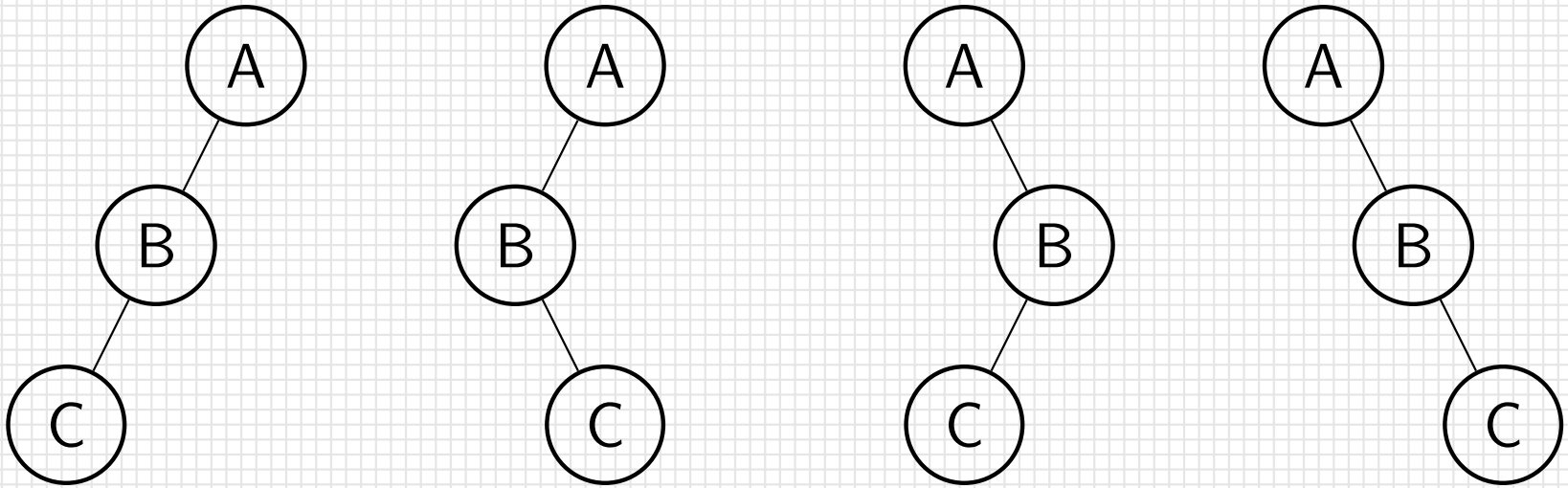
遍历二叉树及其应用

问题　已知一棵二叉树的前序序列为 A、B、C，后序序列为 C、B、A，请问这棵二叉树的中序遍历结果是多少？

遍历二叉树及其应用

问题　已知一棵二叉树的前序序列为 A、B、C，后序序列为 C、B、A，请问这棵二叉树的中序遍历结果是多少？

解：由前序序列和后序序列可以确定根结点为 A，但接下来无法确定哪个结点是左子树，哪个是右子树。这棵树有以下四种可能：



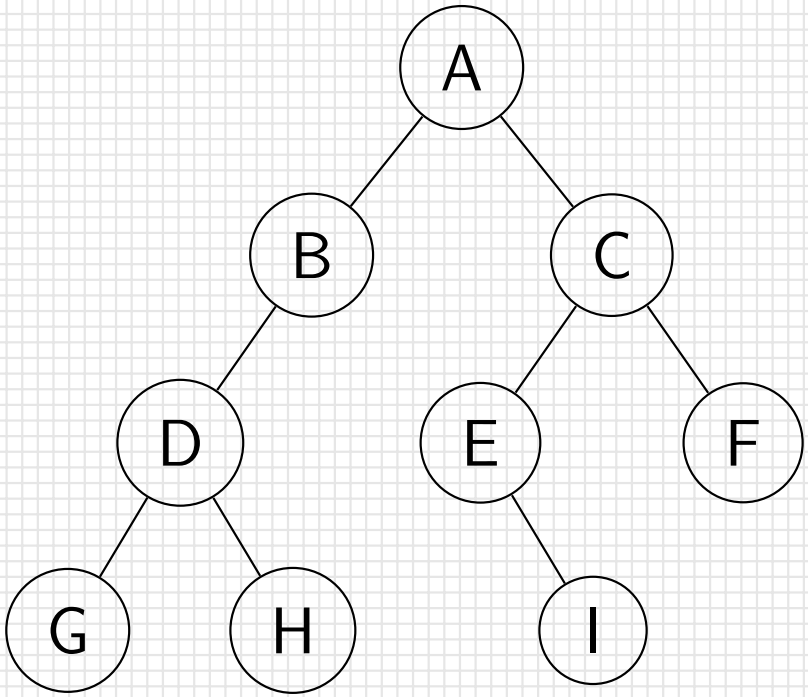
二叉树遍历的性质：

- 已知前序遍历序列和中序遍历序列，可以唯一确定一棵二叉树；
- 已知后序遍历序列和中序遍历序列，可以唯一确定一棵二叉树；
- 已知前序遍历序列和后序遍历序列，不能确定一棵二叉树。

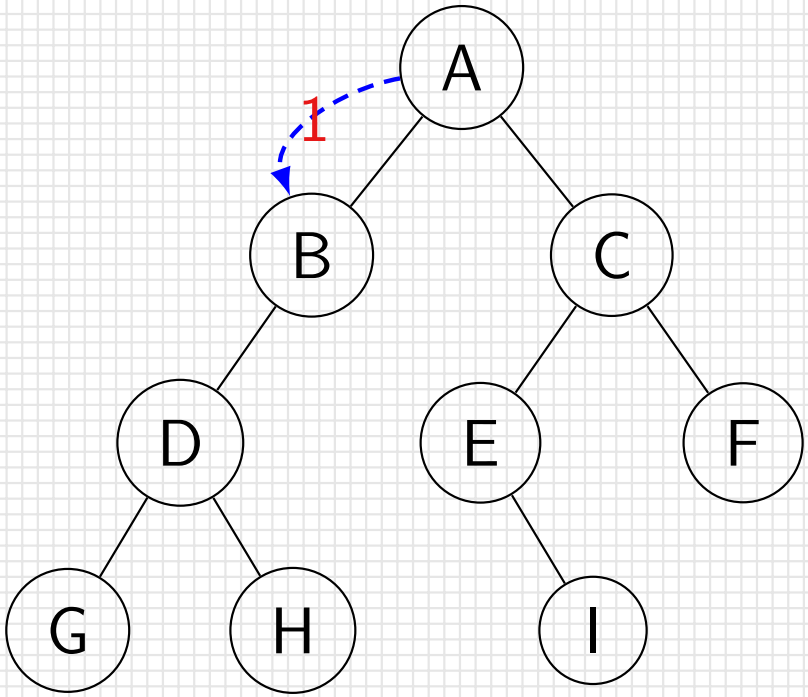
层次遍历

规则：若二叉树为空，则遍历结束；否则从树的第一层，也就是根节点开始访问，从上而下逐层遍历，在同一层中，按从左到右的顺序对结点逐个访问。

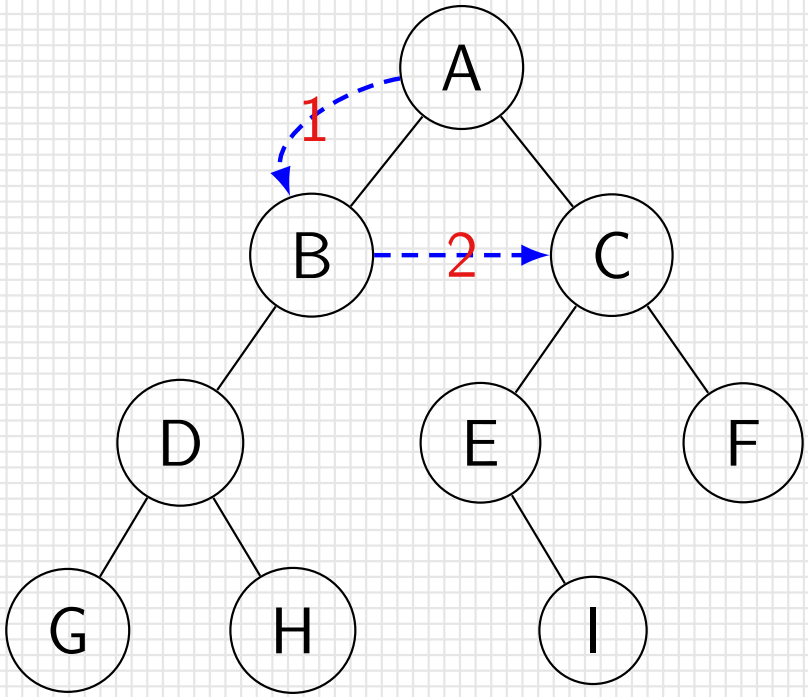
层次遍历



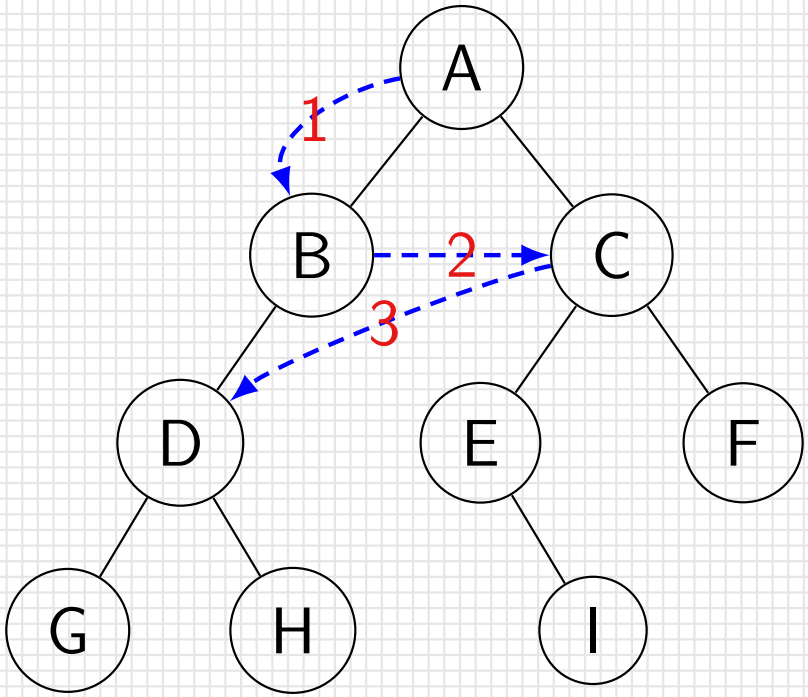
层次遍历



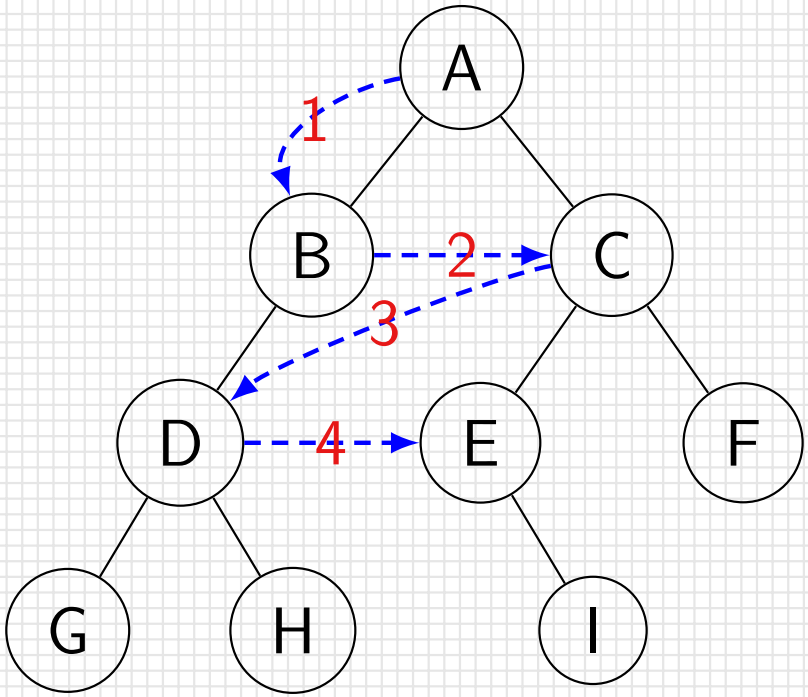
层次遍历



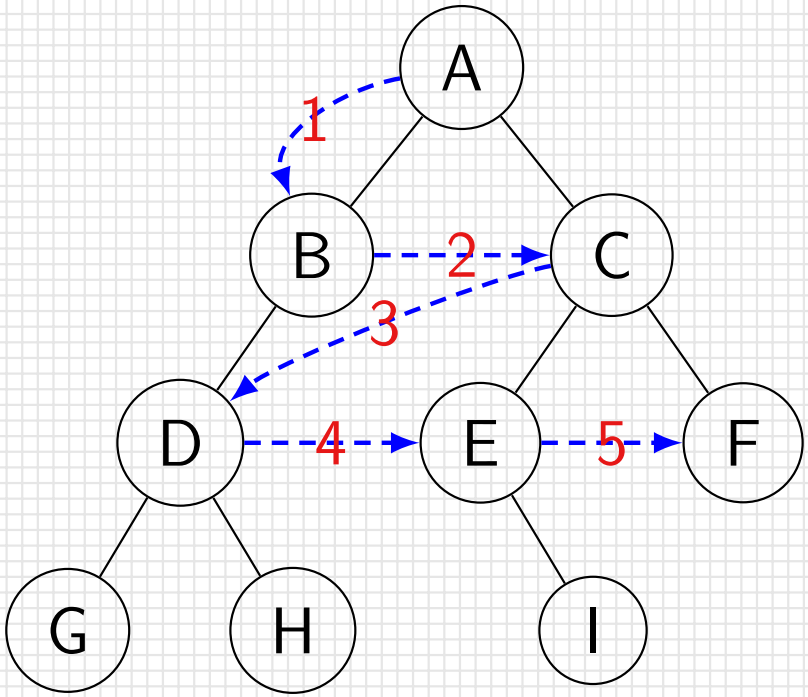
层次遍历



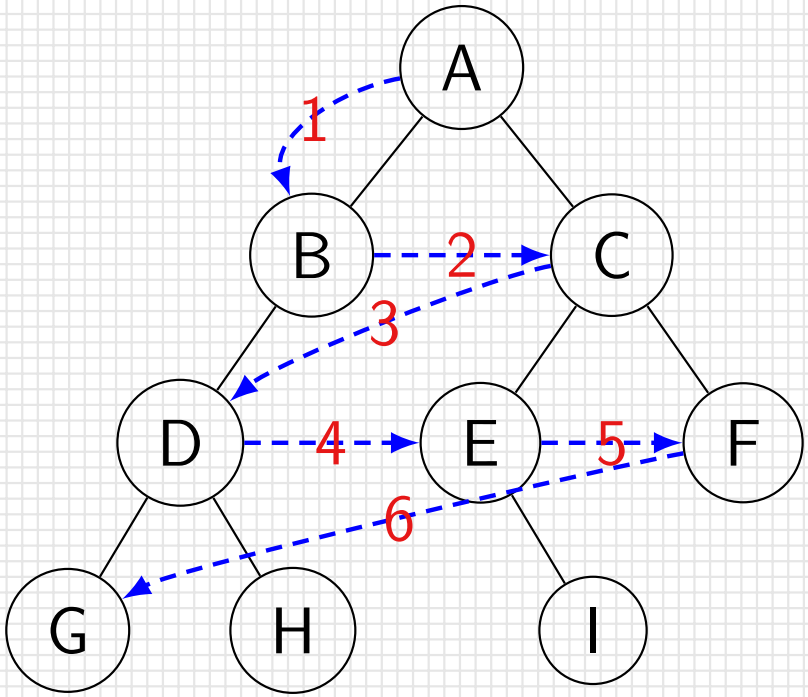
层次遍历



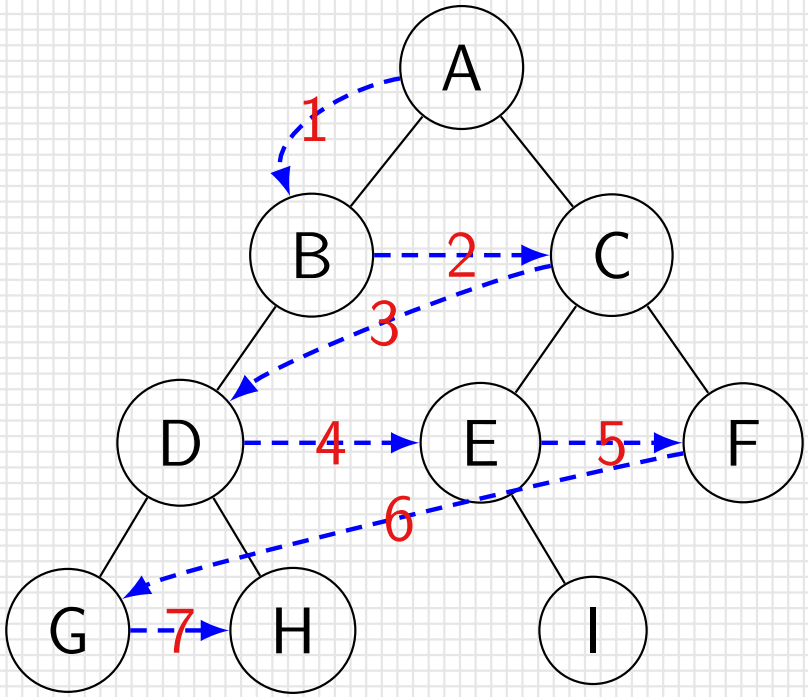
层次遍历



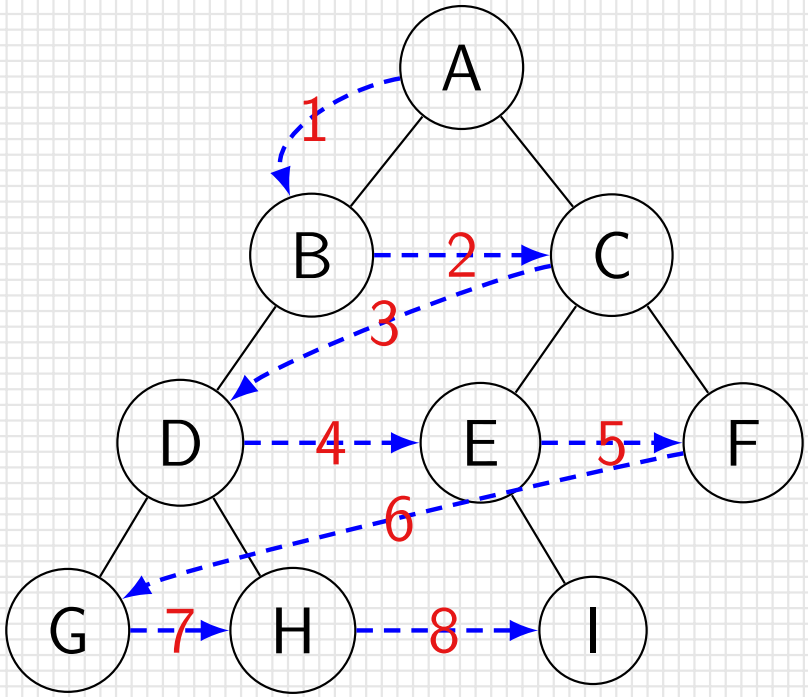
层次遍历



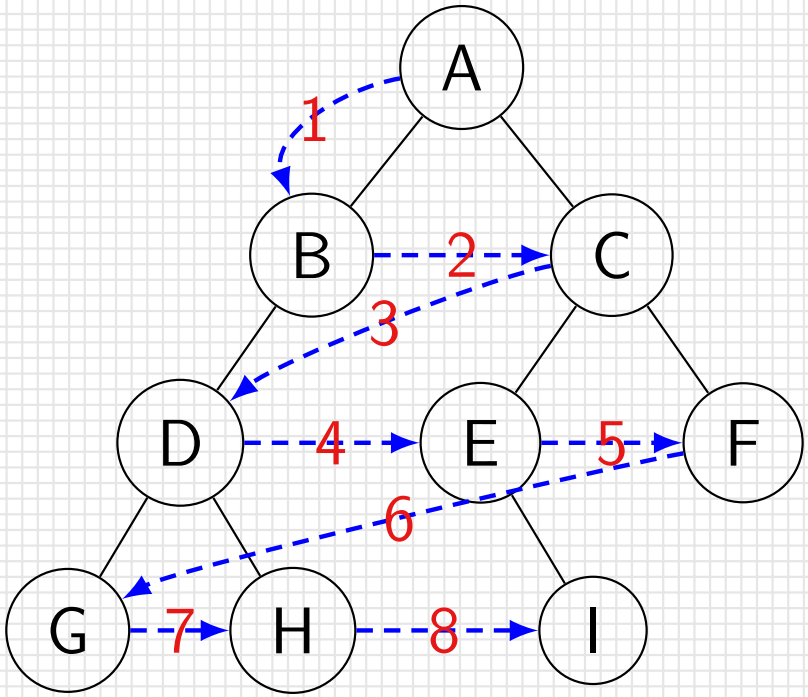
层次遍历



层次遍历



层次遍历



前序遍历结果： A B D G H C E I F

层次遍历 I

```
#include"BiTree.h"

/* LevelOrder Traverse a BiTree */
void LevelOrderTraverse(BiTree tree, void(* visit)())
{
  BTNode * Queue[MAX_NODE], * pnode = tree;
  int front = 0, rear = 0;
  if (pnode != NULL)
  {
    Queue[++rear] = pnode;
    while (front < rear)
    {
      pnode = Queue[++front];
```

层次遍历 II

```
        visit(pnode->data);
        if (pnode->lchild)
          Queue[++rear] = pnode->lchild;
        if (pnode->rchild)
          Queue[++rear] = pnode->rchild;
      }
    }
}
```