树及二叉树

关振扬

September 29, 2025

本章主要内容

- 树的逻辑结构
- 树的存储结构
- 二叉树的逻辑结构
- 二叉树的存储结构及实现
- 二叉树遍历的非递归算法
- 树、森林与二叉树的转换
- 二叉树的应用：哈夫曼树、编码

树的定义

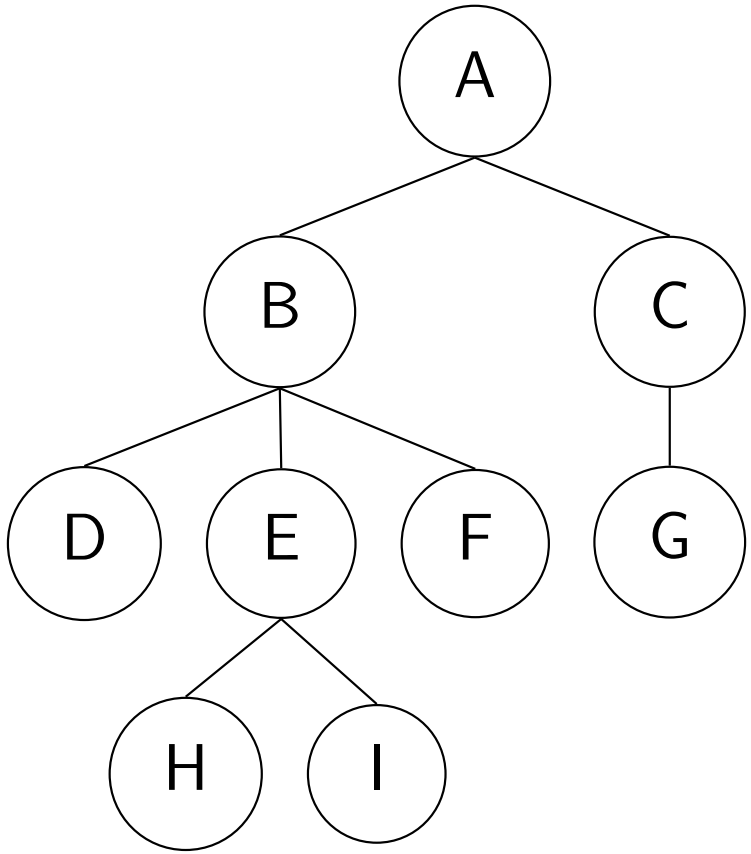
Definition

树是一个 n 个结点的有限集合。当 $n = 0$ 时，称为空树。如果非空，则满足以下条件：

1. 有且仅有一个特定的称为根的结点。
2. 当 $n > 1$ 时，除根结点以外可以分成 m 个互不相交的有限集合 $T_1, \ldots, T_m$，其中每个集合又是一颗树，并称为这个根结点的子树。

有趣的，树的定义是采用递归的方法定义；因此有不少关于树的问题，自然的解决方法都是采用递归。

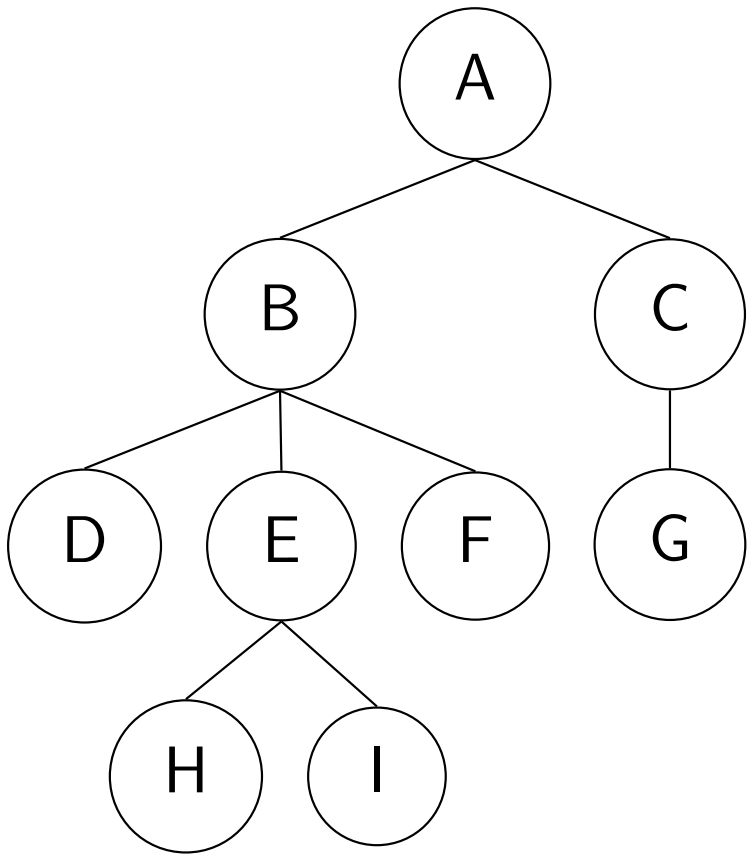
树的例子



实际一点的例子：文件索引

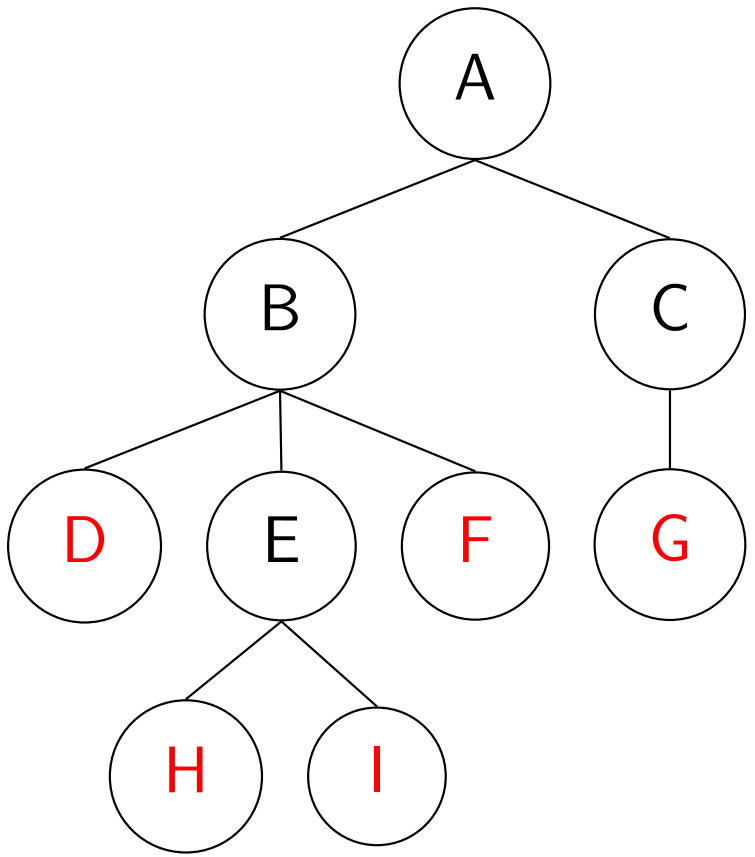
树的基本术语

- 结点的度：结点所拥有的子树的个数
- 树的度：树中各结点度的最大值
- 叶子结点：度为 0 的结点，也称为终端结点
- 分支结点：度不为 0 的结点，也称为非终端结点



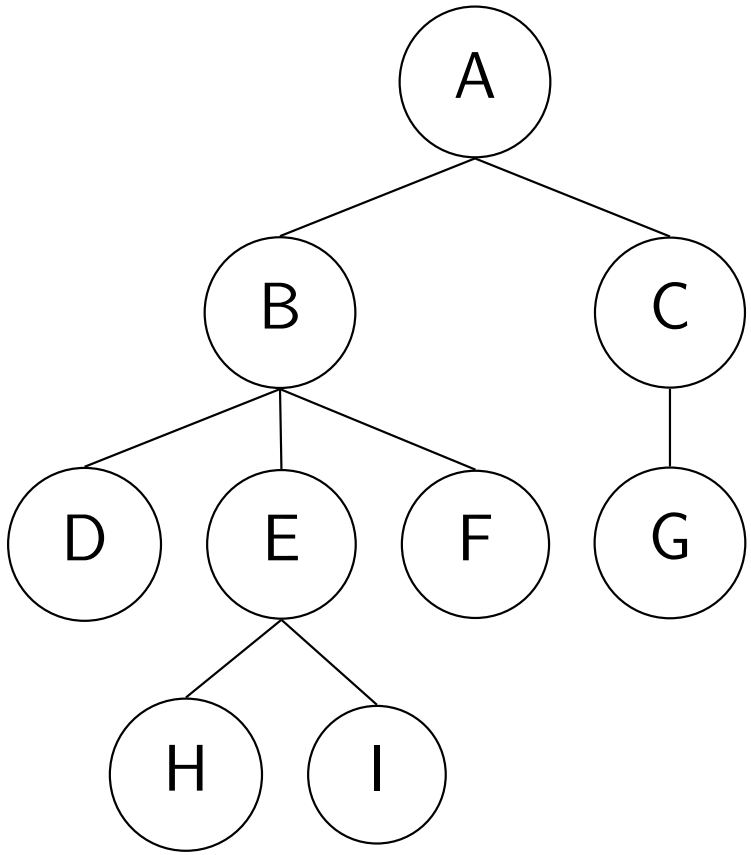
树的基本术语

- 结点的度：结点所拥有的子树的个数
- 树的度：树中各结点度的最大值
- 叶子结点：度为 0 的结点，也称为终端结点
- 分支结点：度不为 0 的结点，也称为非终端结点



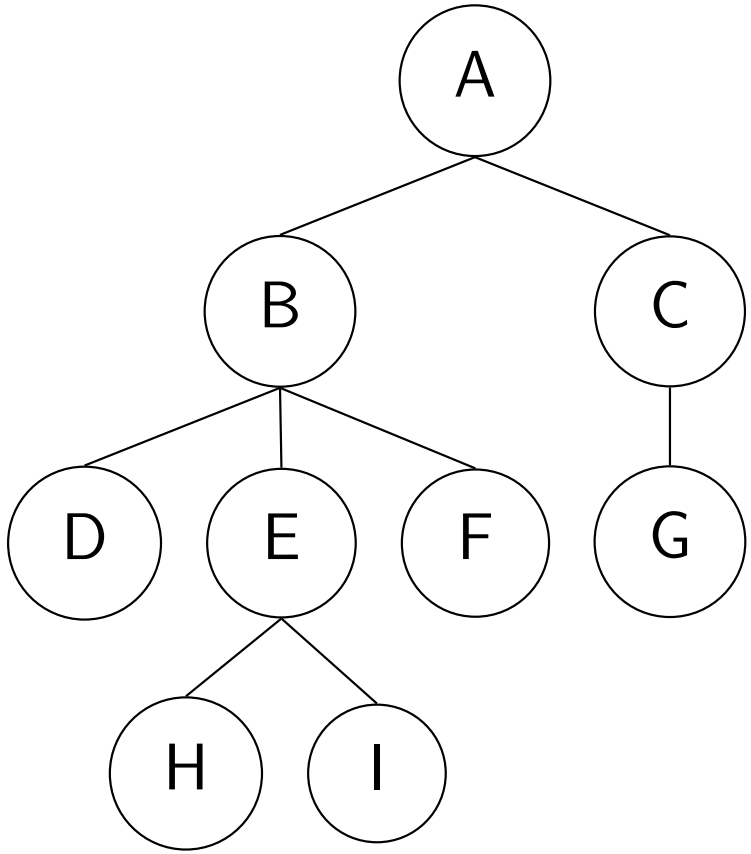
树的基本术语（续）

- 孩子、双亲：一个结点拥有的子树，其根结点称为这个结点的孩子；而同时该结点亦为其孩子的双亲（比如 B 是 A 的孩子；A 是 B 的双亲）
- 兄弟：具有同一个双亲的结点称为兄弟
- 祖先、子孙：双亲的双亲的……双亲、孩子的孩子的……孩子



树的基本术语（续）

- 结点所在层数：根结点的层数为 1；对其他结点，若某结点在第 k 层，那么其孩子结点在第 $k+1$ 层。
- 树的深度：树中所有结点的最大层数，也称为高度。
- 层序编号：将树中结点按照从上层到下层、同层从左到右的次序给编号（从 1 号起）。（这个例子刚好顺字母顺序）



树的基本术语（续）

- 一棵树中的各个子树从左到右是有顺序的，称这棵树为有序树；反之，称为无序树。
- 数据结构一般只讨论有序树
- 几棵树的集合称为森林。

树结构与线性结构的对比

线性结构：

- 第一个数据元素（无前驱）

树结构：

- 根结点（无双亲）

树结构与线性结构的对比

线性结构：

- 第一个数据元素（无前驱）
- 最后一个数据元素（无后继）

树结构：

- 根结点（无双亲）
- 叶子结点（无孩子，可存在复数个）

树结构与线性结构的对比

线性结构：

- 第一个数据元素（无前驱）
- 最后一个数据元素（无后继）
- 一对一

树结构：

- 根结点（无双亲）
- 叶子结点（无孩子，可存在复数个）
- 一对多

树的 ADT

- 由于树的应用很广，因此比较难给出一个统一的 ADT。以下只给出数据的关系以及几个比较普遍的操作。
- 数据之间的关系为双亲、孩子。
- 操作方面：构造函数、析构函数
- 遍历树的方法：前序、后序、层序
- 比较难统一的：增减结点、子树等方法

树的遍历

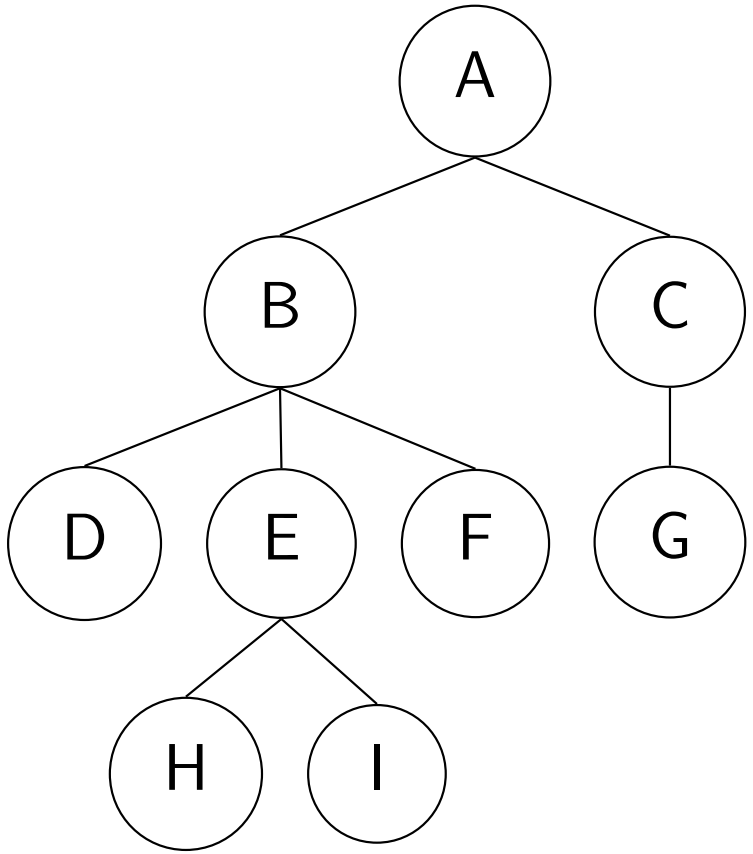
- 从根结点出发，按照某种顺序访问（并处理）树中的所有结点，并对每个结点只访问一次。
- 为了方便解说，我们以下就是假定读出并打印每个位置的数据
- 本质来说：遍历的过程可以看为把树的结构（临时）变为一个线性结构
- 常见的顺序：前序、后序、层序。

前序遍历

前序遍历定义为：

- 若空树，则遍历完成
- 访问根结点
- 从左到右访问每一颗子树

前序结果为：

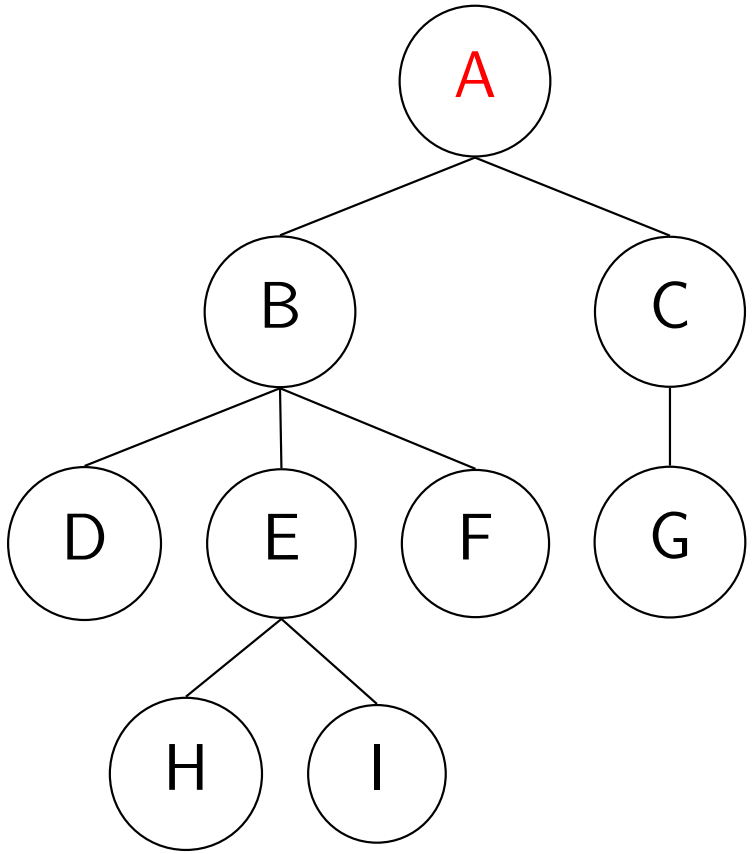


前序遍历

前序遍历定义为：

- 若空树，则遍历完成
- 访问根结点
- 从左到右访问每一颗子树

前序结果为：A

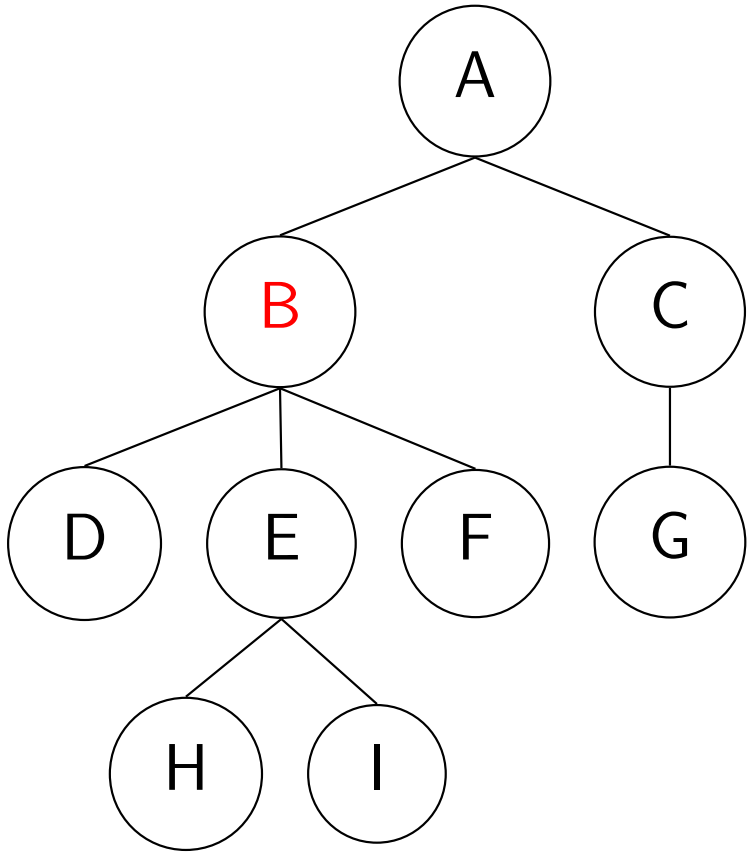


前序遍历

前序遍历定义为：

- 若空树，则遍历完成
- 访问根结点
- 从左到右访问每一颗子树

前序结果为：A B

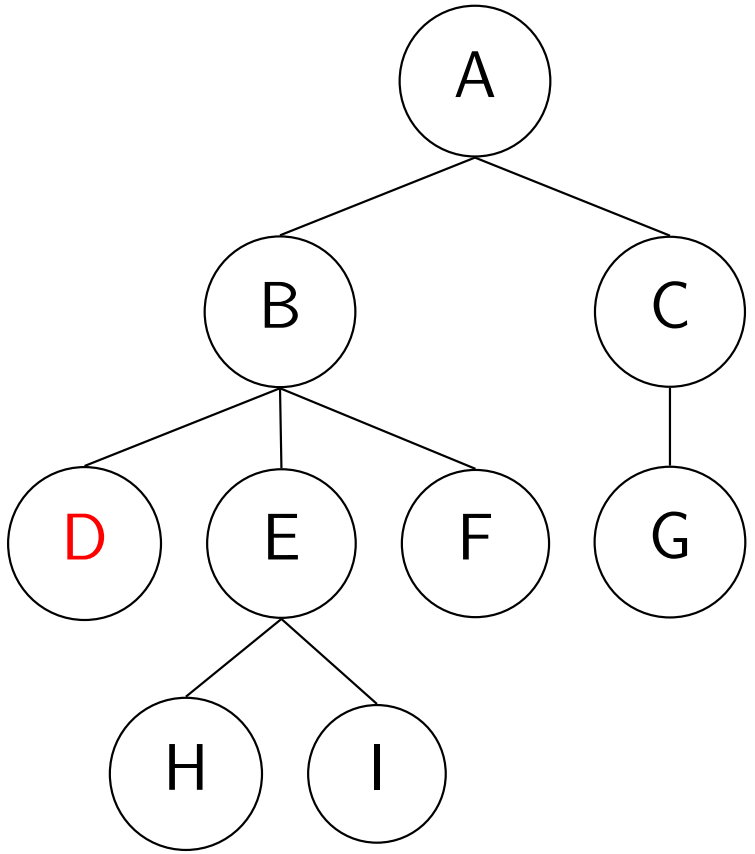


前序遍历

前序遍历定义为：

- 若空树，则遍历完成
- 访问根结点
- 从左到右访问每一颗子树

前序结果为：A B D

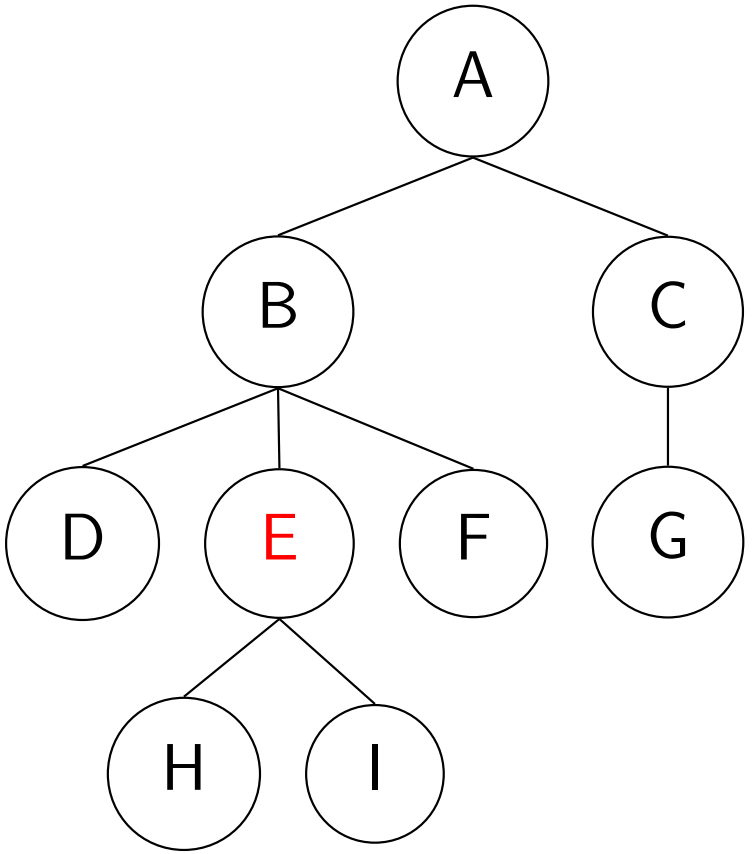


前序遍历

前序遍历定义为：

- 若空树，则遍历完成
- 访问根结点
- 从左到右访问每一颗子树

前序结果为：A B D E

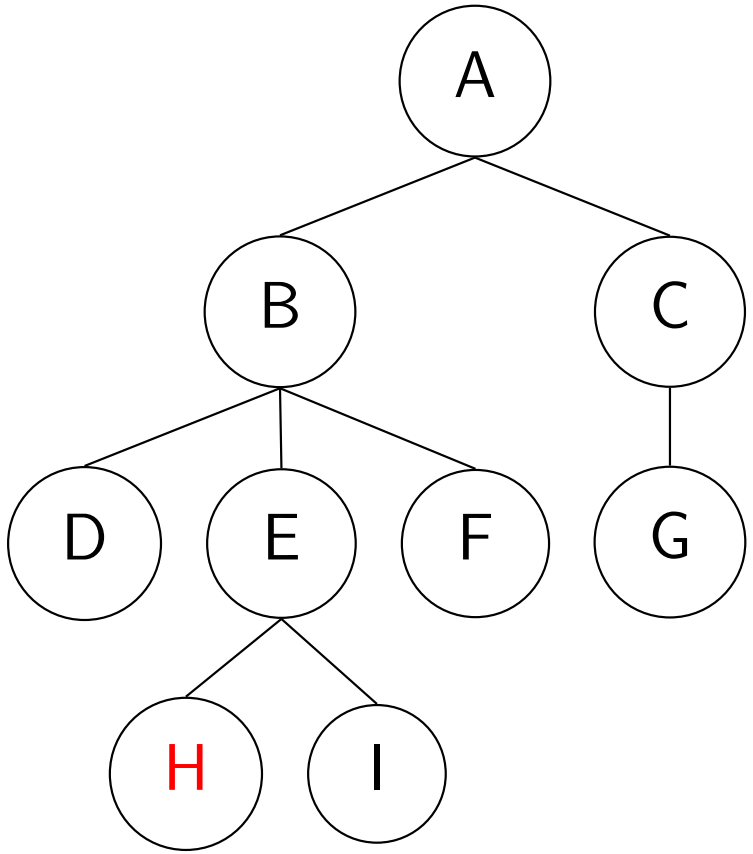


前序遍历

前序遍历定义为：

- 若空树，则遍历完成
- 访问根结点
- 从左到右访问每一颗子树

前序结果为：A B D E H

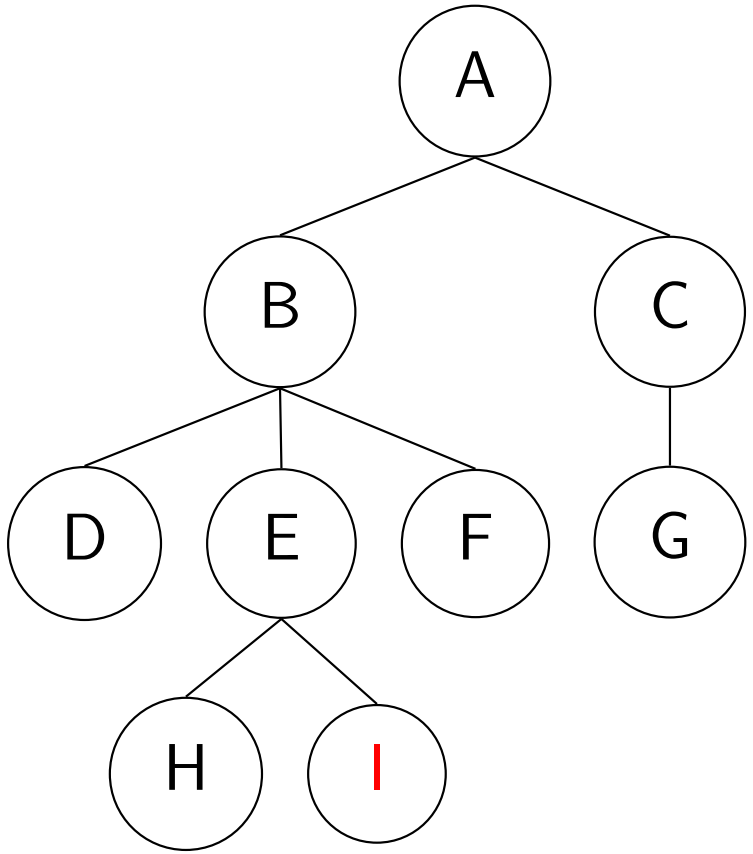


前序遍历

前序遍历定义为：

- 若空树，则遍历完成
- 访问根结点
- 从左到右访问每一颗子树

前序结果为：A B D E H I

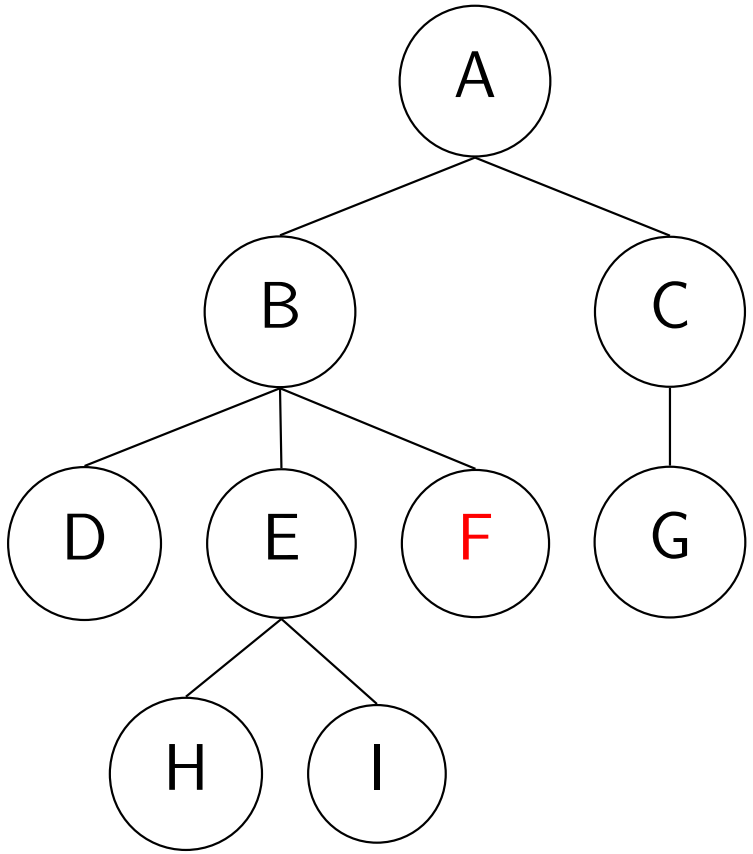


前序遍历

前序遍历定义为：

- 若空树，则遍历完成
- 访问根结点
- 从左到右访问每一颗子树

前序结果为：A B D E H I F

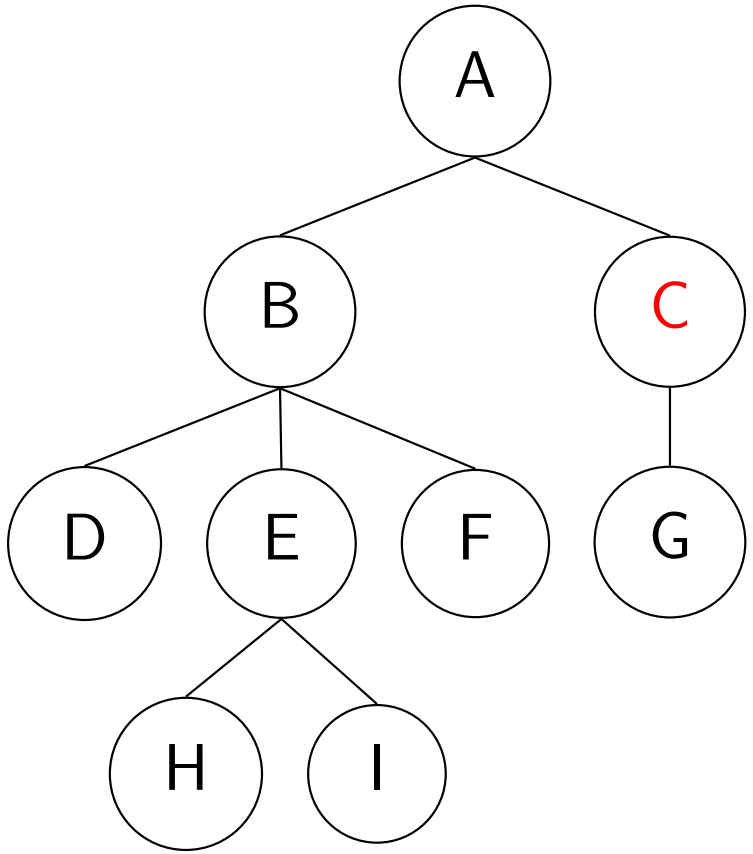


前序遍历

前序遍历定义为：

- 若空树，则遍历完成
- 访问根结点
- 从左到右访问每一颗子树

前序结果为：A B D E H I F C

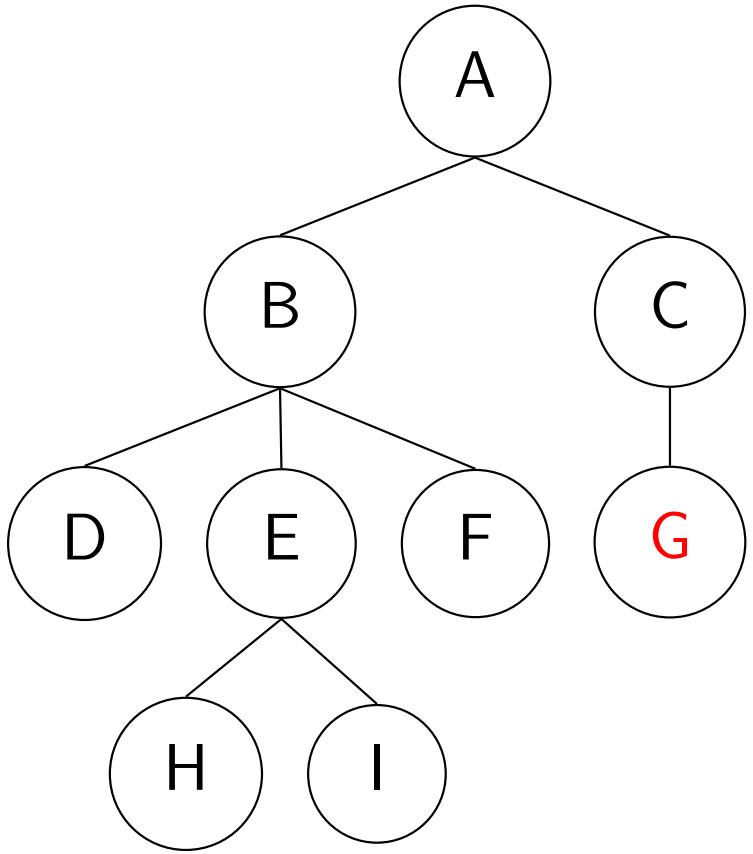


前序遍历

前序遍历定义为：

- 若空树，则遍历完成
- 访问根结点
- 从左到右访问每一颗子树

前序结果为：A B D E H I F C G

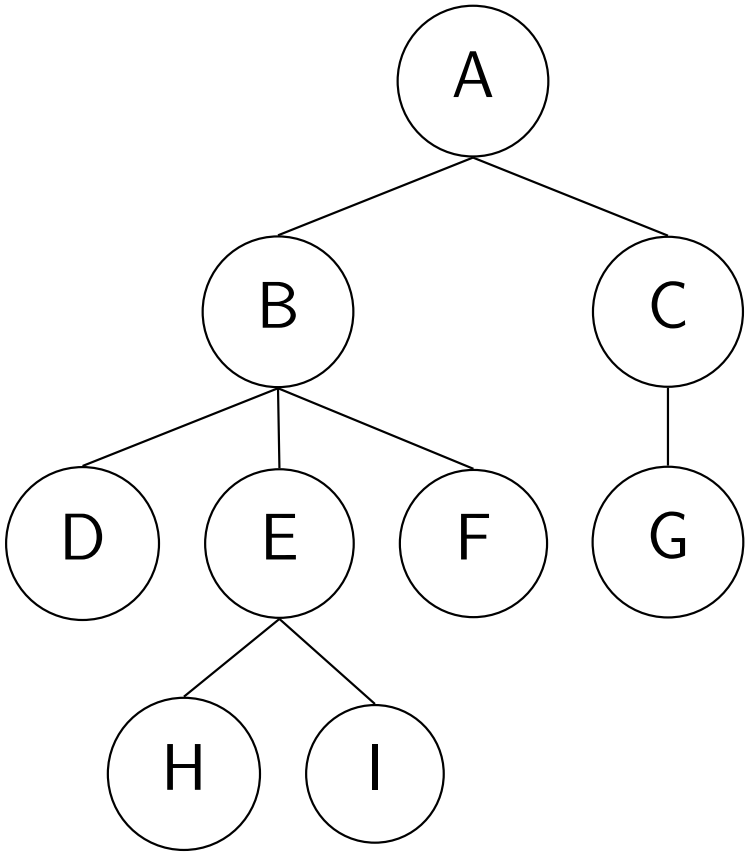


后序遍历

后序遍历定义为：

- 若空树，则遍历完成
- 从左到右访问每一颗子树
- 访问根结点

后序结果为：

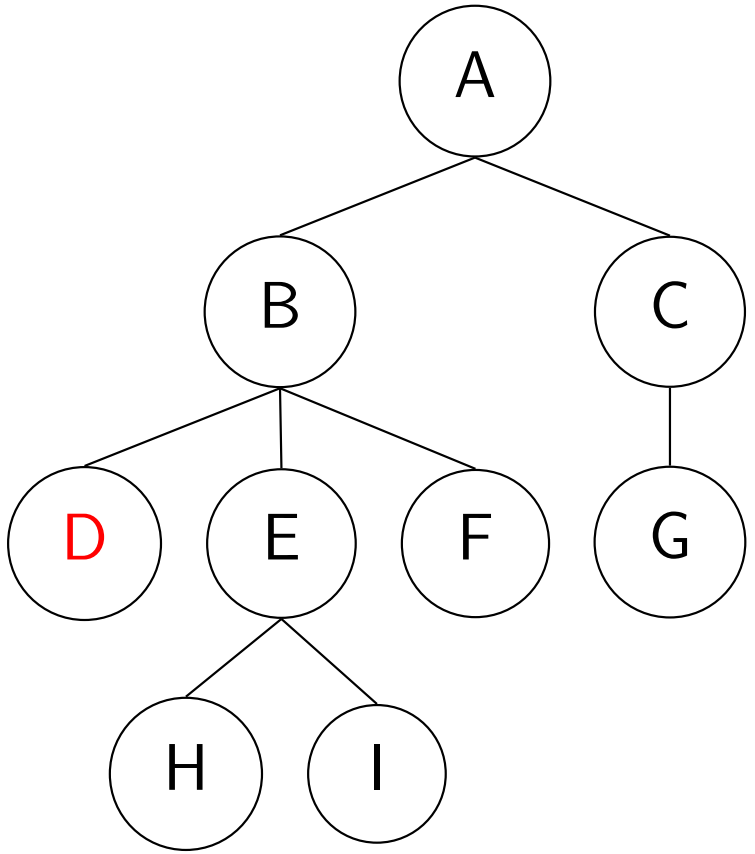


后序遍历

后序遍历定义为：

- 若空树，则遍历完成
- 从左到右访问每一颗子树
- 访问根结点

后序结果为：D

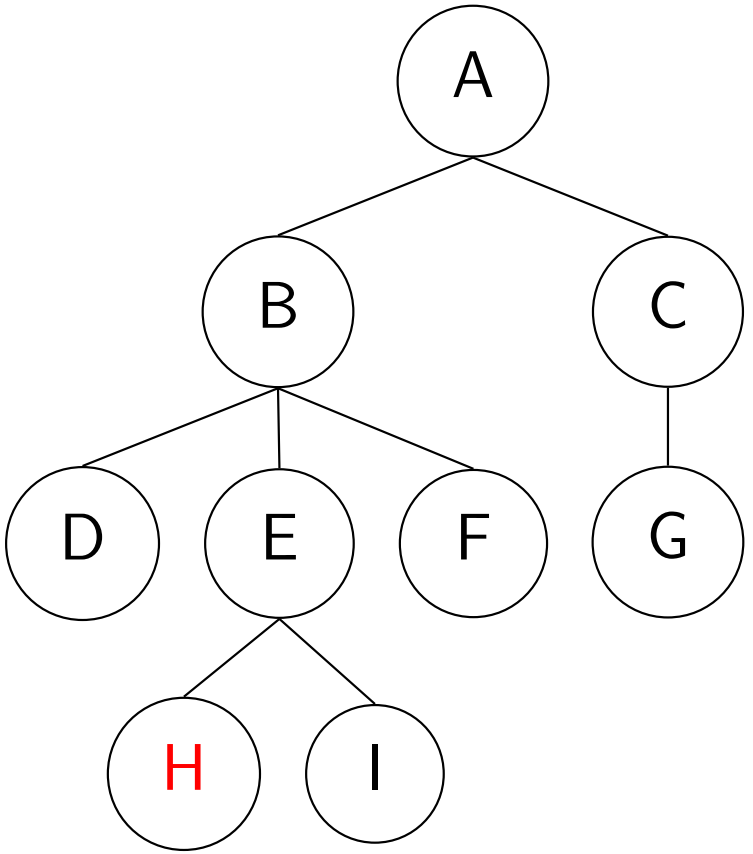


后序遍历

后序遍历定义为：

- 若空树，则遍历完成
- 从左到右访问每一颗子树
- 访问根结点

后序结果为：D H

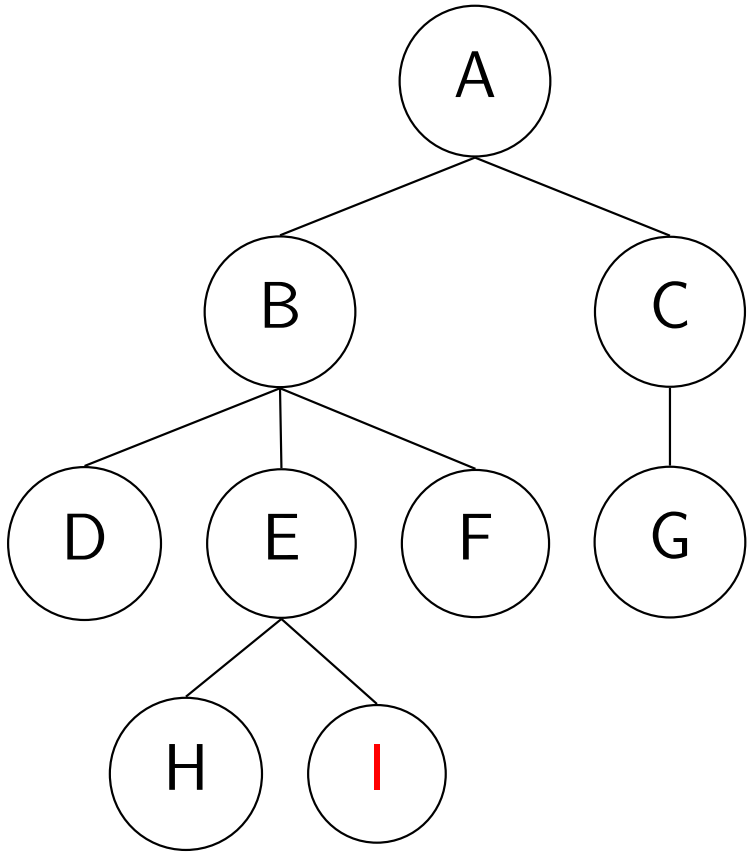


后序遍历

后序遍历定义为：

- 若空树，则遍历完成
- 从左到右访问每一颗子树
- 访问根结点

后序结果为：D H I

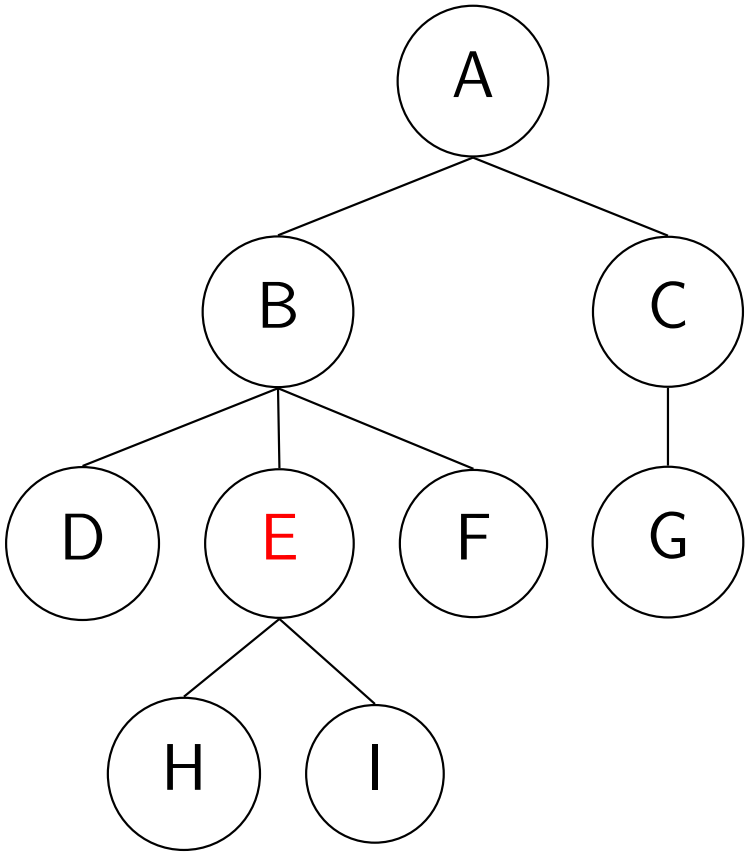


后序遍历

后序遍历定义为：

- 若空树，则遍历完成
- 从左到右访问每一颗子树
- 访问根结点

后序结果为：D H I E

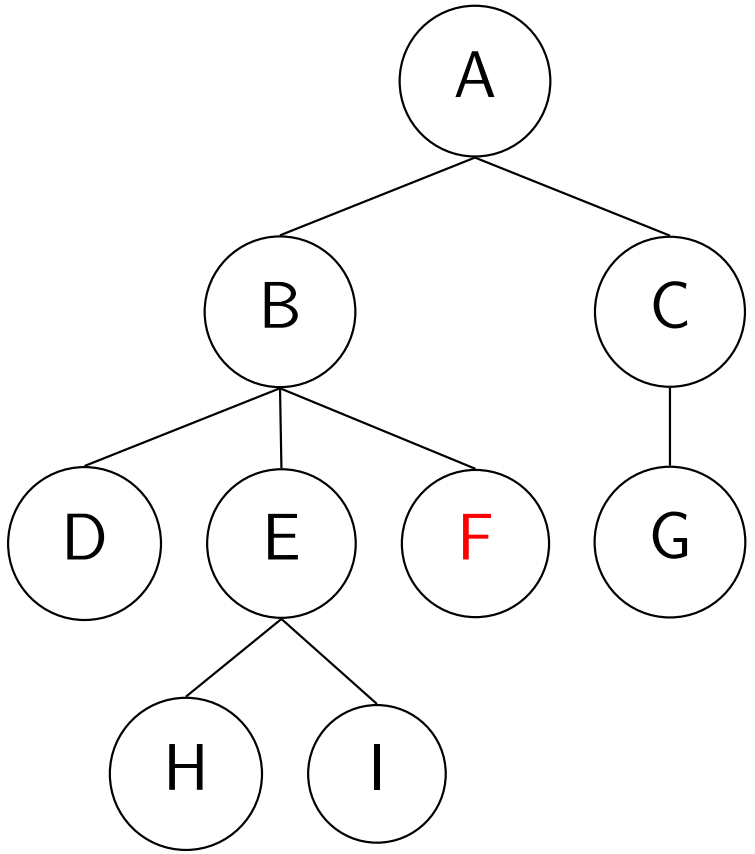


后序遍历

后序遍历定义为：

- 若空树，则遍历完成
- 从左到右访问每一颗子树
- 访问根结点

后序结果为：D H I E F

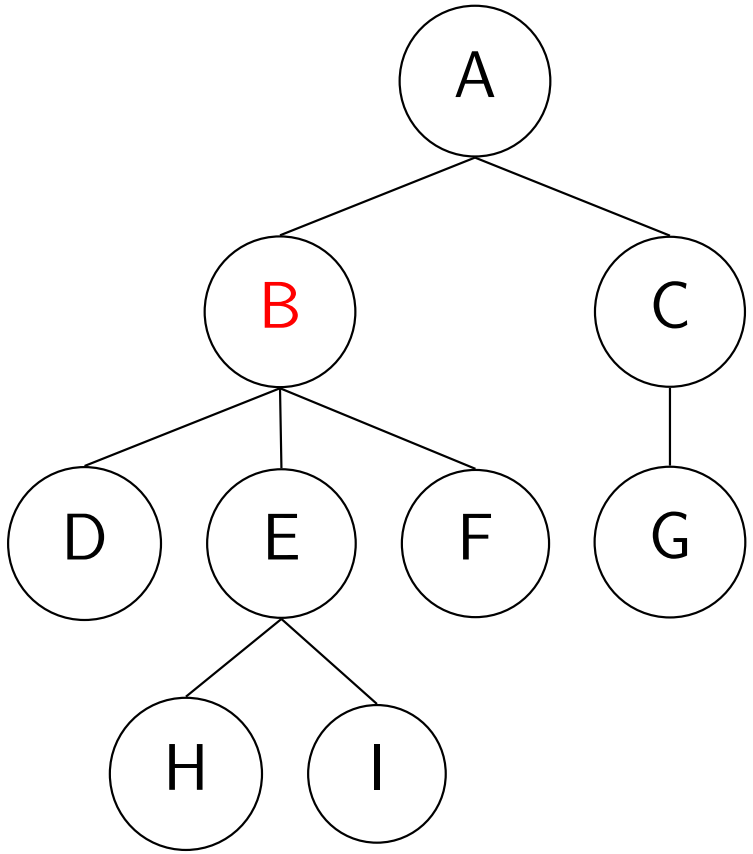


后序遍历

后序遍历定义为：

- 若空树，则遍历完成
- 从左到右访问每一颗子树
- 访问根结点

后序结果为：D H I E F B

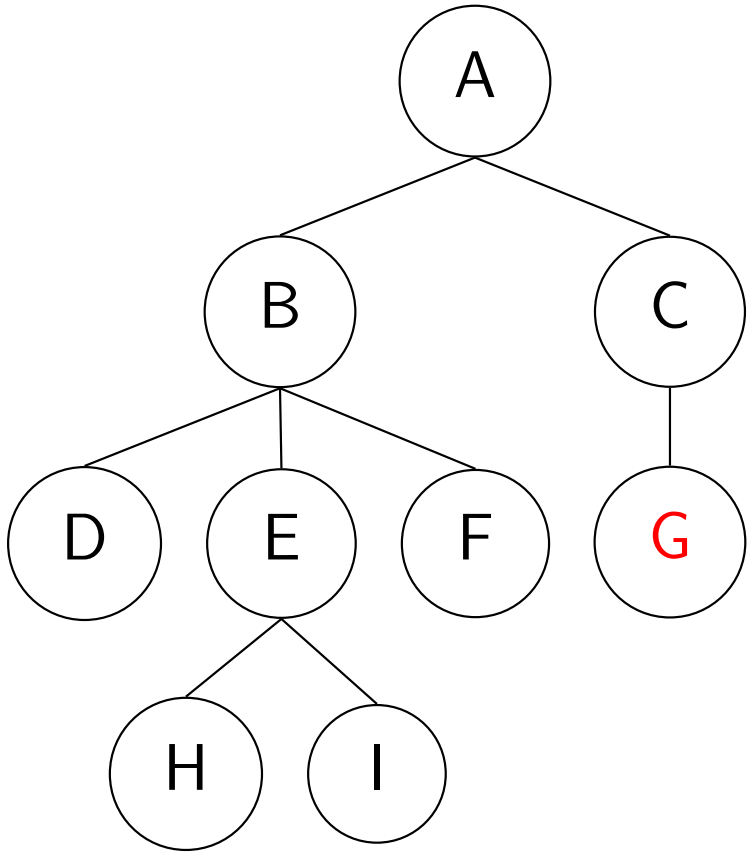


后序遍历

后序遍历定义为：

- 若空树，则遍历完成
- 从左到右访问每一颗子树
- 访问根结点

后序结果为：D H I E F B G

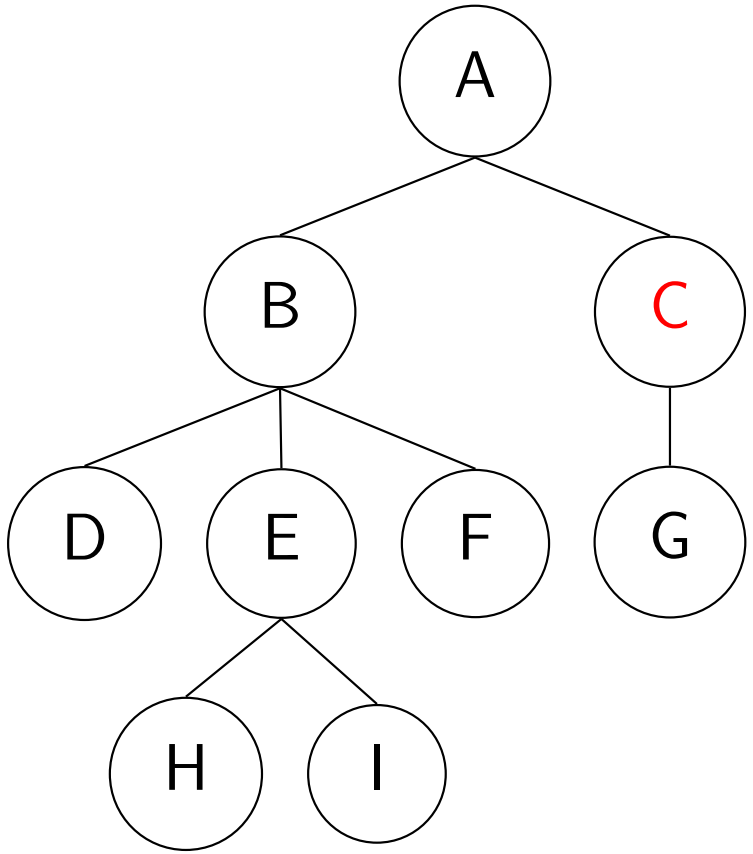


后序遍历

后序遍历定义为：

- 若空树，则遍历完成
- 从左到右访问每一颗子树
- 访问根结点

后序结果为：D H I E F B G C

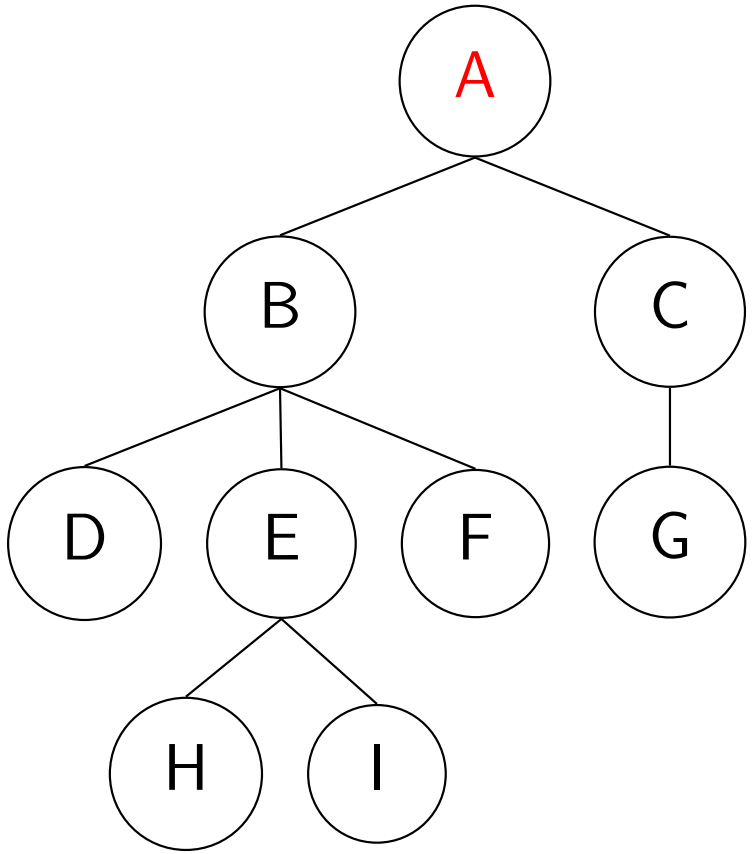


后序遍历

后序遍历定义为：

- 若空树，则遍历完成
- 从左到右访问每一颗子树
- 访问根结点

后序结果为：D H I E F B G C A

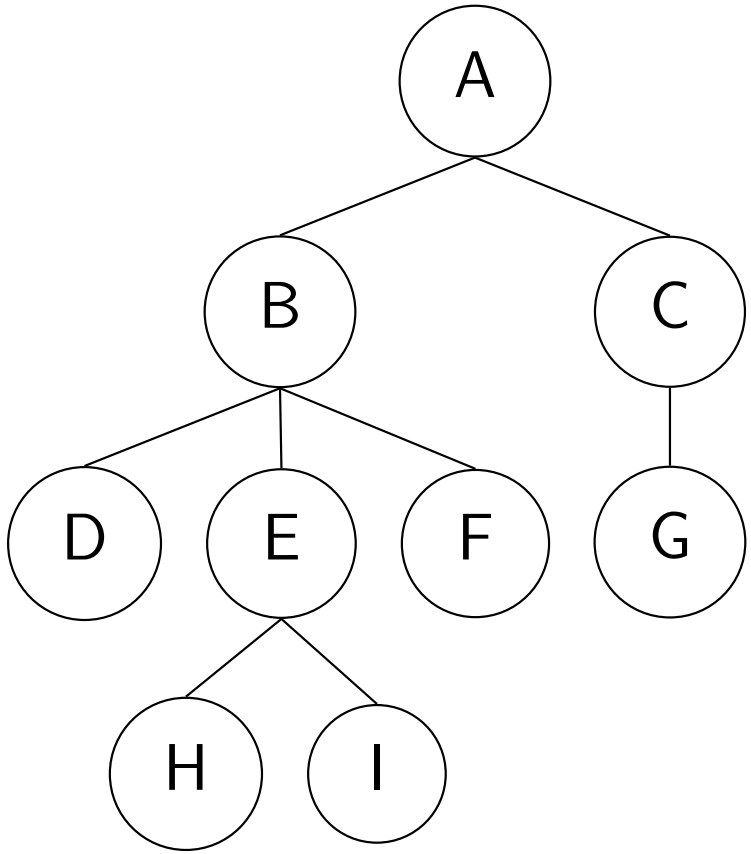


层序遍历

层序遍历定义为：

- 若空树，则遍历完成
- 从左到右访问每一层

层序结果为：

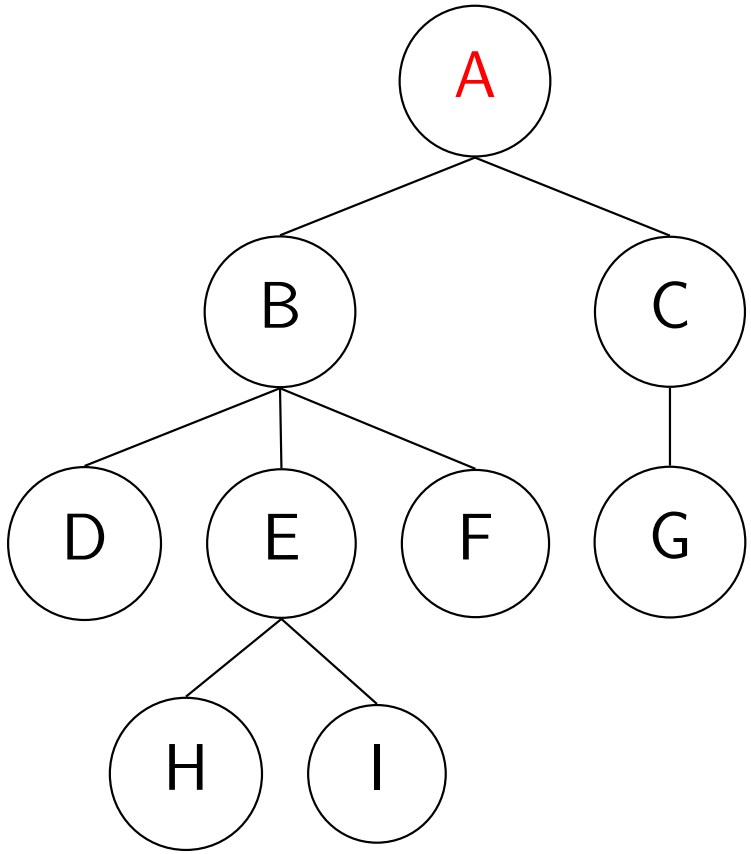


层序遍历

层序遍历定义为：

- 若空树，则遍历完成
- 从左到右访问每一层

层序结果为：A

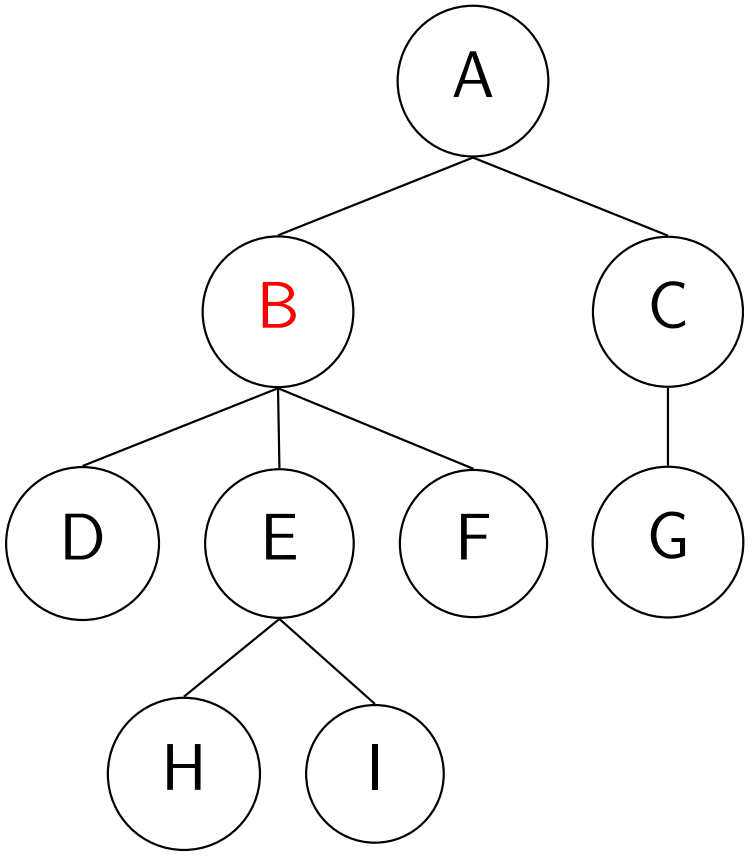


层序遍历

层序遍历定义为：

- 若空树，则遍历完成
- 从左到右访问每一层

层序结果为：A B

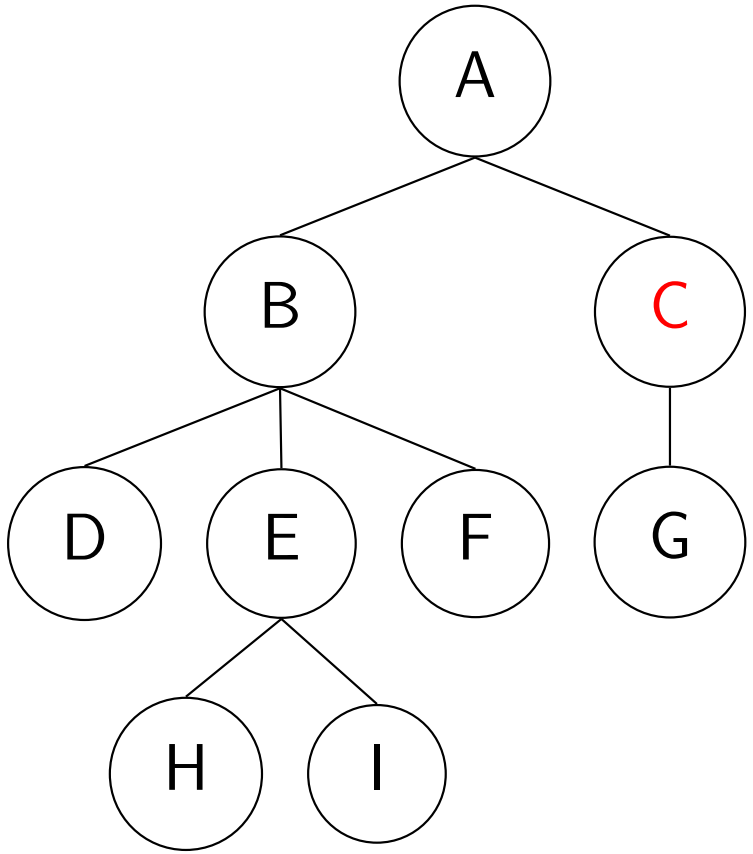


层序遍历

层序遍历定义为：

- 若空树，则遍历完成
- 从左到右访问每一层

层序结果为：A B C

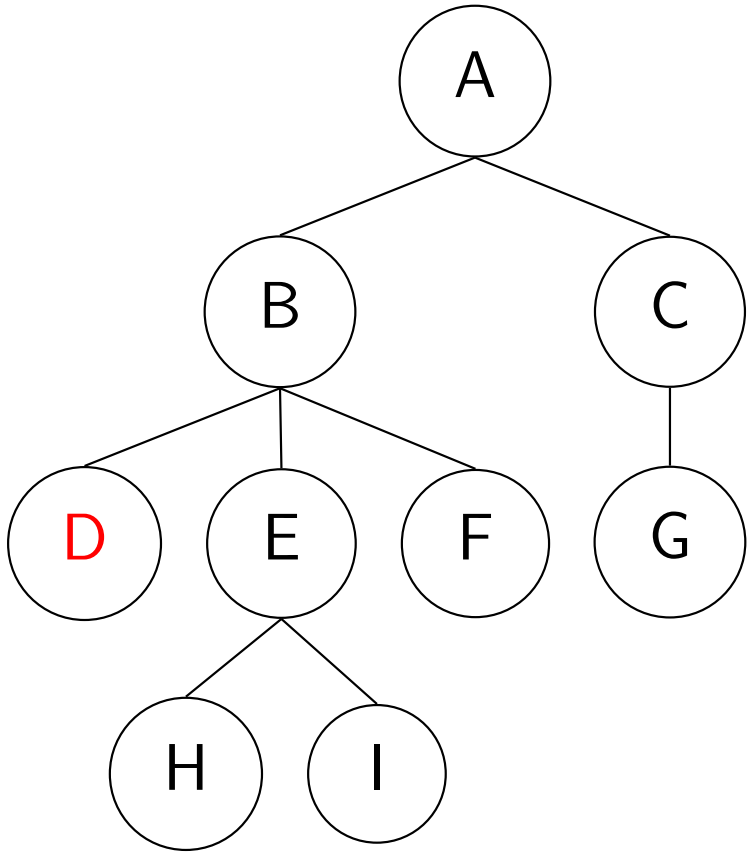


层序遍历

层序遍历定义为：

- 若空树，则遍历完成
- 从左到右访问每一层

层序结果为：A B C D

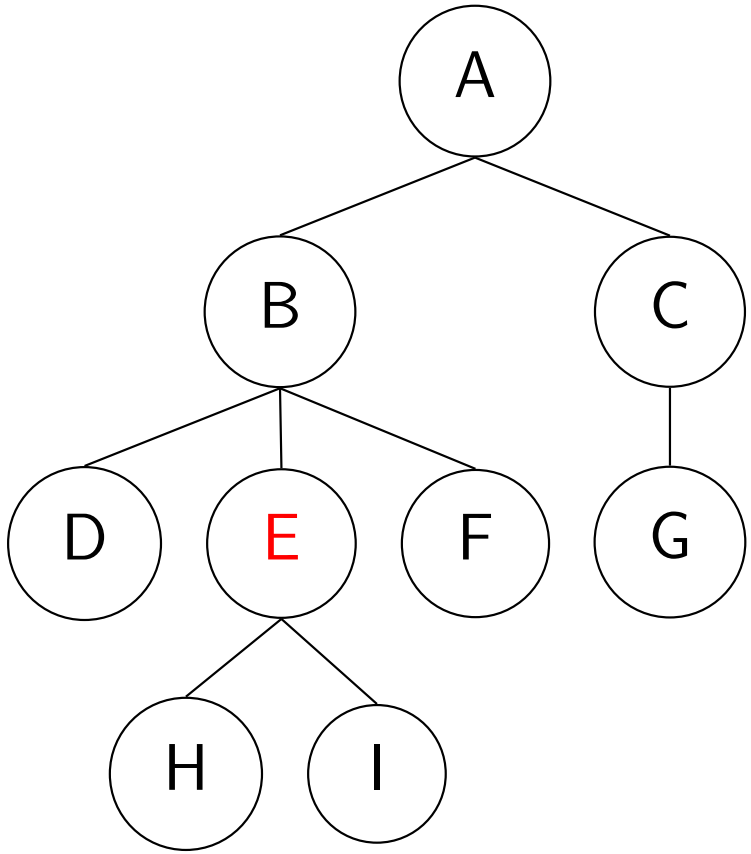


层序遍历

层序遍历定义为：

- 若空树，则遍历完成
- 从左到右访问每一层

层序结果为：A B C D E

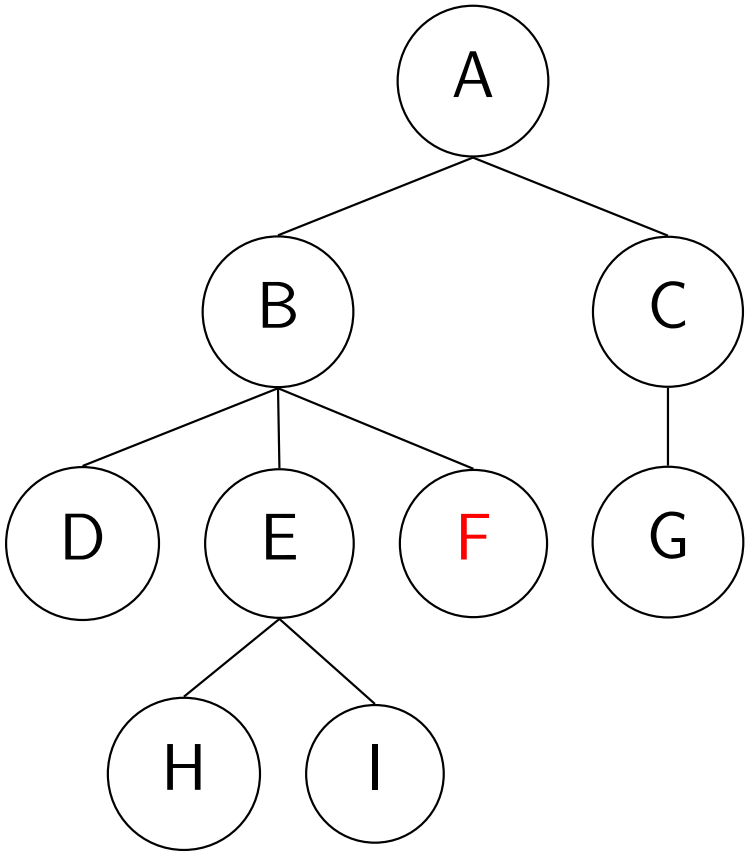


层序遍历

层序遍历定义为：

- 若空树，则遍历完成
- 从左到右访问每一层

层序结果为：A B C D E F

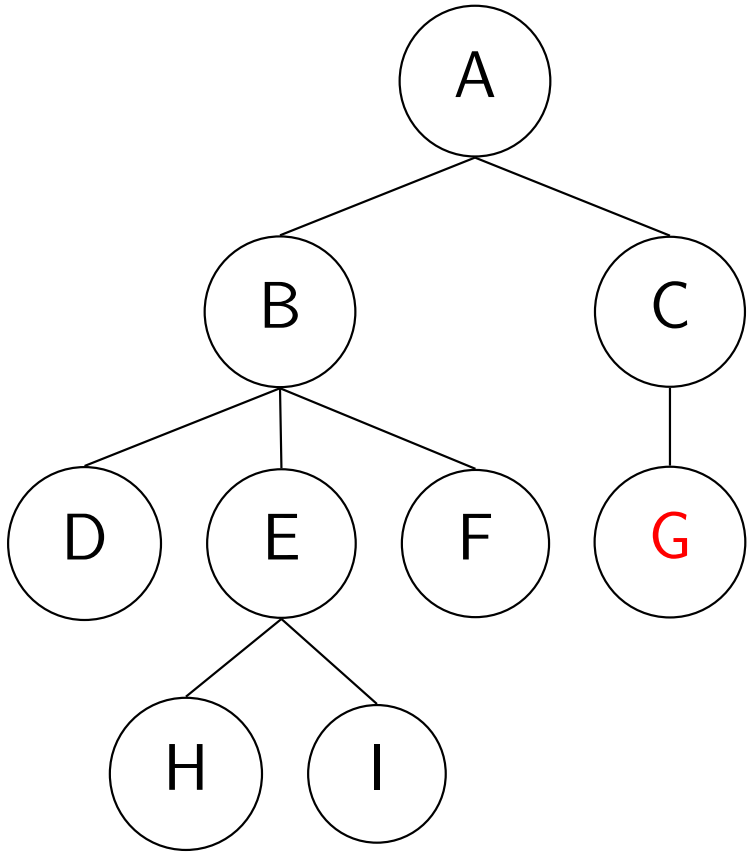


层序遍历

层序遍历定义为：

- 若空树，则遍历完成
- 从左到右访问每一层

层序结果为：A B C D E F G

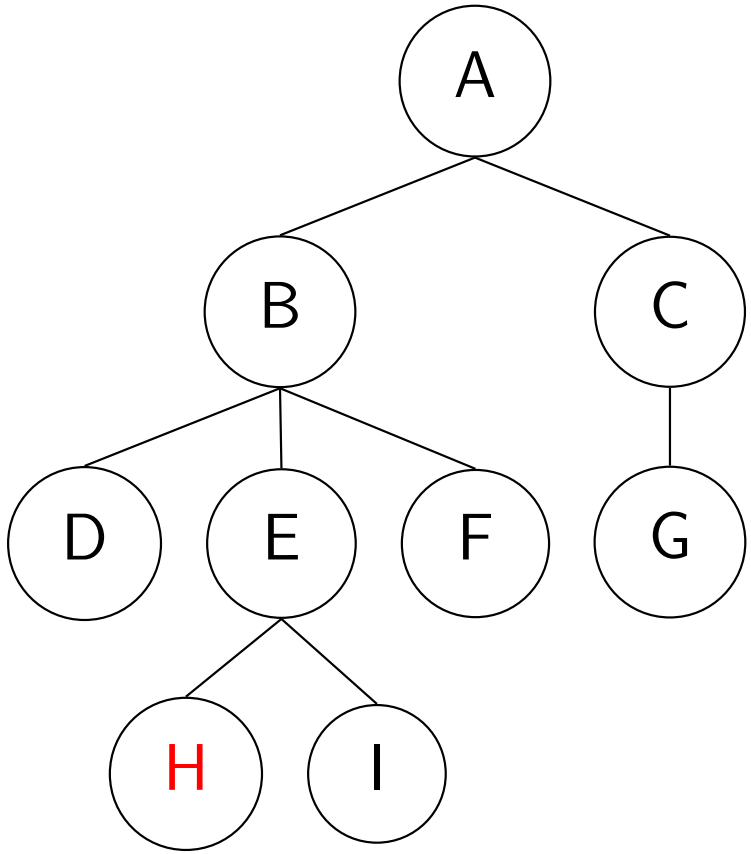


层序遍历

层序遍历定义为：

- 若空树，则遍历完成
- 从左到右访问每一层

层序结果为：A B C D E F G H

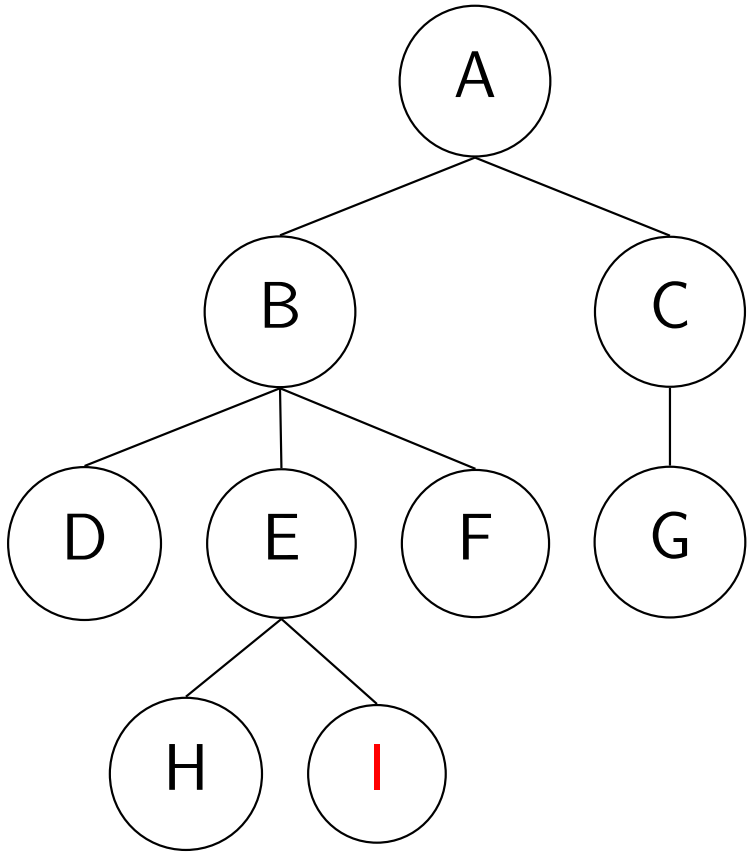


层序遍历

层序遍历定义为：

- 若空树，则遍历完成
- 从左到右访问每一层

层序结果为：A B C D E F G H I

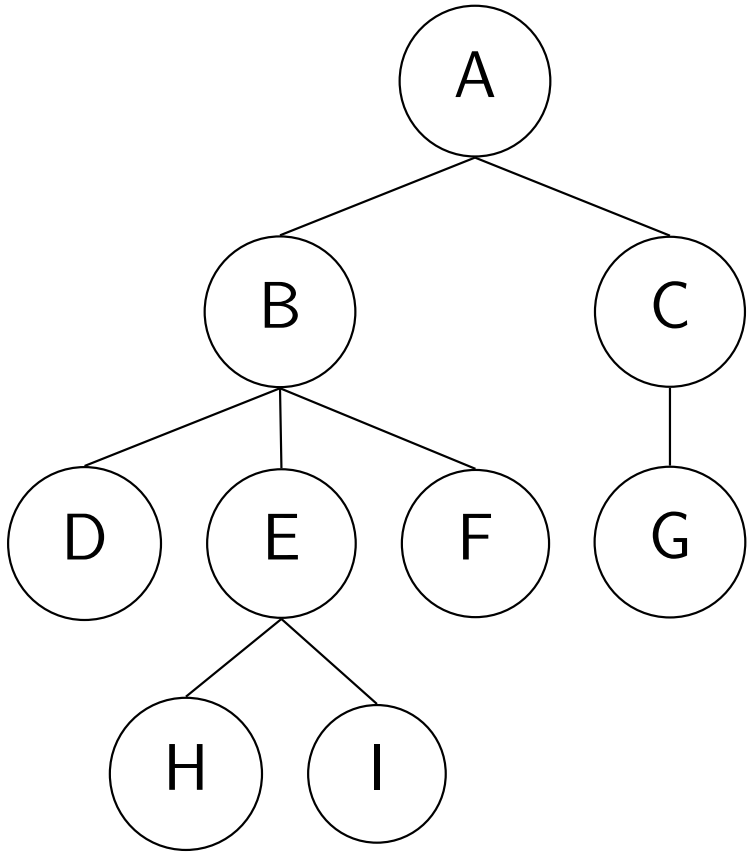


双亲表示法

- 关键问题：如何表达出结点之间的逻辑关系（双亲、孩子）
- 基本思路：使用一个数组存储所有结点
- 每个结点带有数据和双亲结点号码

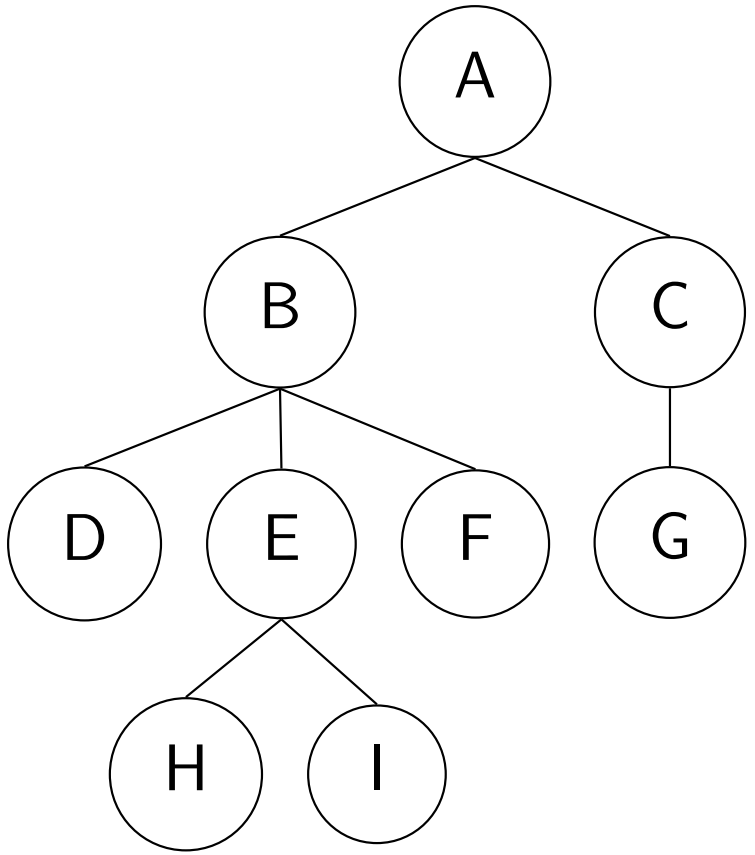
双亲表示法（续）

下标	数据	双亲
0	A	-1
1	B	0
2	C	0
3	D	1
4	E	1
5	F	1
6	G	2
7	H	4
8	I	4



双亲表示法（续）

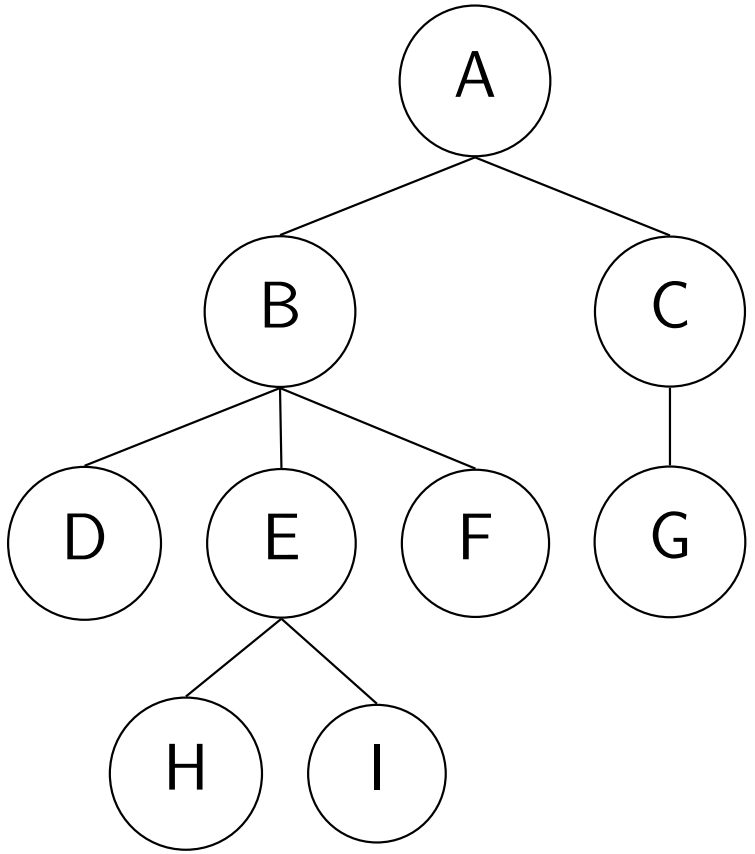
下标	数据	双亲
0	A	-1
1	B	0
2	C	0
3	D	1
4	E	1
5	F	1
6	G	2
7	H	4
8	I	4



如何查找孩子？

双亲表示法（续）

下标	数据	双亲
0	A	-1
1	B	0
2	C	0
3	D	1
4	E	1
5	F	1
6	G	2
7	H	4
8	I	4



如何查找兄弟？

孩子表示法

- 每个结点包括数据与足够多的指针域指向该结点的所有孩子结点。

孩子表示法

- 每个结点包括数据与足够多的指针域指向该结点的所有孩子结点。
- 方案一：每个结点的指针数等于其度数

孩子表示法

- 每个结点包括数据与足够多的指针域指向该结点的所有孩子结点。
- 方案一：每个结点的指针数等于其度数
- 方案二：每个结点都有树的度数的指针空格

孩子表示法

- 每个结点包括数据与足够多的指针域指向该结点的所有孩子结点。
- 方案一：每个结点的指针数等于其度数
- 方案二：每个结点都有树的度数的指针空格
- 前者的问题：结点结构不统一

孩子表示法

- 每个结点包括数据与足够多的指针域指向该结点的所有孩子结点。
- 方案一：每个结点的指针数等于其度数
- 方案二：每个结点都有树的度数的指针空格
- 前者的问题：结点结构不统一
- 后者的问题：空间浪费

孩子表示法

- 每个结点包括数据与足够多的指针域指向该结点的所有孩子结点。
- 方案一：每个结点的指针数等于其度数
- 方案二：每个结点都有树的度数的指针空格
- 前者的问题：结点结构不统一
- 后者的问题：空间浪费
- 方案三：用向量表或链表存储所有指向孩子结点的指针

双亲孩子表示法

- 孩子表示法的问题为查找双亲速度慢。

双亲孩子表示法

- 孩子表示法的问题为查找双亲速度慢。
- 那如果想要查找孩子或双亲的速度都有保证，要怎样？

双亲孩子表示法

- 孩子表示法的问题为查找双亲速度慢。
- 那如果想要查找孩子或双亲的速度都有保证，要怎样？
- 结合前两个表示方法就行了。

孩子兄弟表示法

- 另一个有效的表示方法为孩子兄弟。
- 每个结点指向其第一个孩子，及首个在右边的兄弟。
- 如果还希望查找双亲的速度快，则再配上一个指针指向其双亲。

二叉树的定义

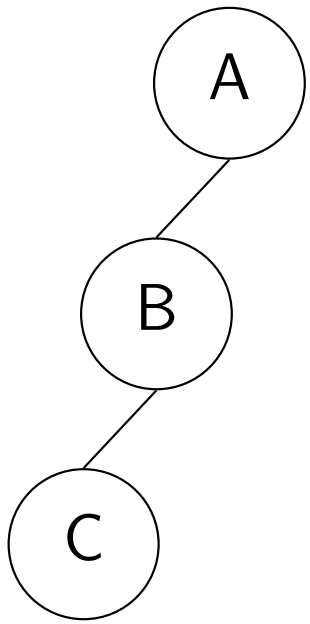
Definition

二叉树是 n 个结点的有限集合。当集合为空时，称为空二叉树。非空时，则有一个根结点以及两颗互不相交的二叉树构成（分别为左子树、右子树）。

二叉树与一般的树有两个很大的分别：一个是度数的上界；另一个则是孩子不一定从左到右，可以有右子树而没有左子树。

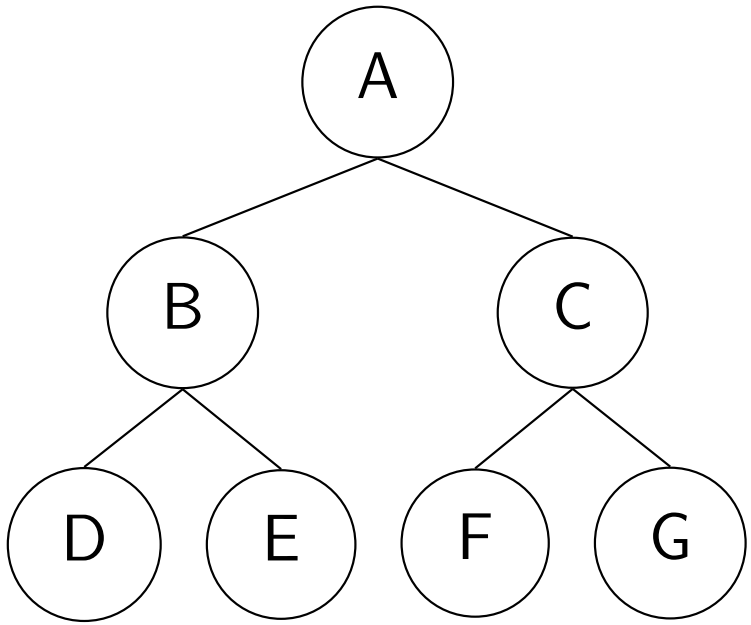
特殊的二叉树：斜树

- 所有结点只有左子树的二叉树称为左斜树。
- 同理，所有结点只有右子树的称为右斜树。
- 两类统称为斜树。
- 特点：结点数等于层数，每层只有一个结点。



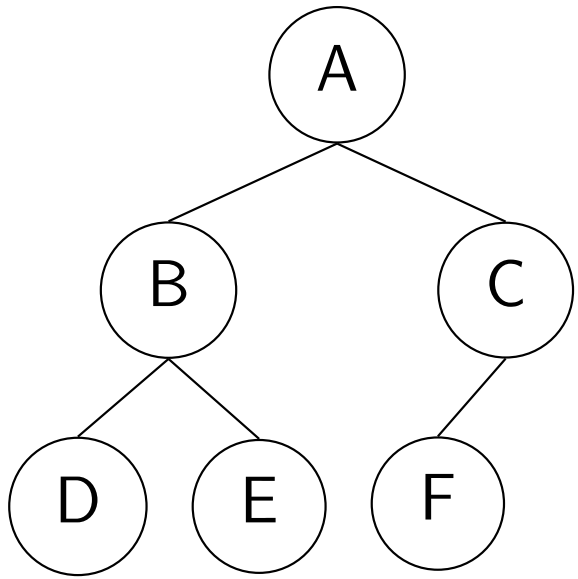
特殊的二叉树：满二叉树

- 所有分支结点均有两个孩子
- 所有孩子在同一层
- k 层的满二叉树的结点数为？



特殊的二叉树：完全二叉树

- 这类树的层序编号与同深度的满二叉树的层序编号完全一致
- 可以看成是由满二叉树不断把最大层序号码的结点删除所得
- 最多有一个度为 1 的结点，而且该结点只有左孩子
- 叶子在最底两层，而且在最底层是集中在左边
- 同样节点个数的二叉树中，完全二叉树的深度最小



二叉树的特性

Theorem

二叉树的第 k 层上最多有 2^{k-1} 个结点。

Theorem

所有深度为 k 的二叉树中，最多可有 $2^k - 1$ 个结点。最少有 k 个结点。

注：最少结点的不一定为斜树。

二叉树的特性（续）

Theorem

二叉树中叶子节点的个数比度 2 的结点个数多一个。

- 设 n_i 为树中度 i 的结点个数，n 为树的结点个数。
- $n = n_0 + n_1 + n_2$
- 除根节点外，每个结点均有一个双亲结点，与树中的分支一一对应。
- $n = n_1 + 2n_2 + 1$
- 从两式可推出结论：$n_0 = n_2 + 1$。

完全二叉树的特性

Theorem

具有 n 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$。

- 设层数为 k。
- 明显这棵树包括了 $k-1$ 层的满二叉树，因此有 $2^{k-1} \leq n$。
- 结点数有上界：$n \leq 2^k - 1$。
- 取对数即有结论。

完全二叉树的特性（续）

Theorem

对 n 个结点的完全二叉树的每个结点赋层序编号。那么对序号为 m 的结点，有：

1. 若 $m = 1$，无双亲结点；否则其双亲结点序号为 $\lfloor m/2 \rfloor$。
2. 若 $2m > n$，则该结点为叶子结点；否则该结点有左孩子，序号为 $2m$。
3. 若 $2m < n$，那么该结点有右孩子，序号为 $2m + 1$。

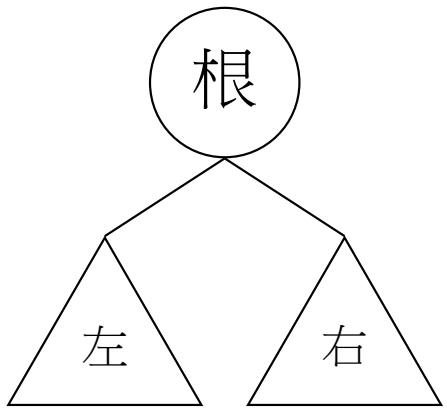
这代表完全二叉树其实可以用顺序结构实现。

二叉树的 ADT

- 与树一样，二叉树由于应用广泛，也是没有完全统一的 ADT。
- 数据之间的关系为双亲、左孩子、右孩子。
- 操作方面：构造函数、析构函数
- 遍历树的方法：前序、中序、后序、层序
- 比较难统一的：增减结点、子树等方法

二叉树的遍历

- 层序遍历还是每层从左到右。
- 前序是访问根、左子树、右子树
- 后序是访问左子树、右子树、根
- 中序是访问左子树、根、右子树

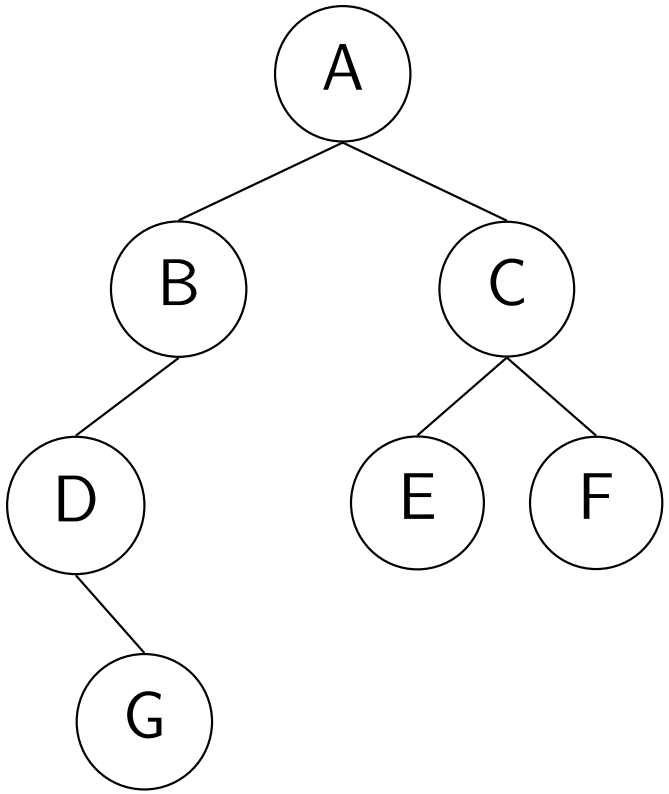


二叉树的遍历（例子）

- 层序遍历就是：
  A,B,C,D,E,F,G。

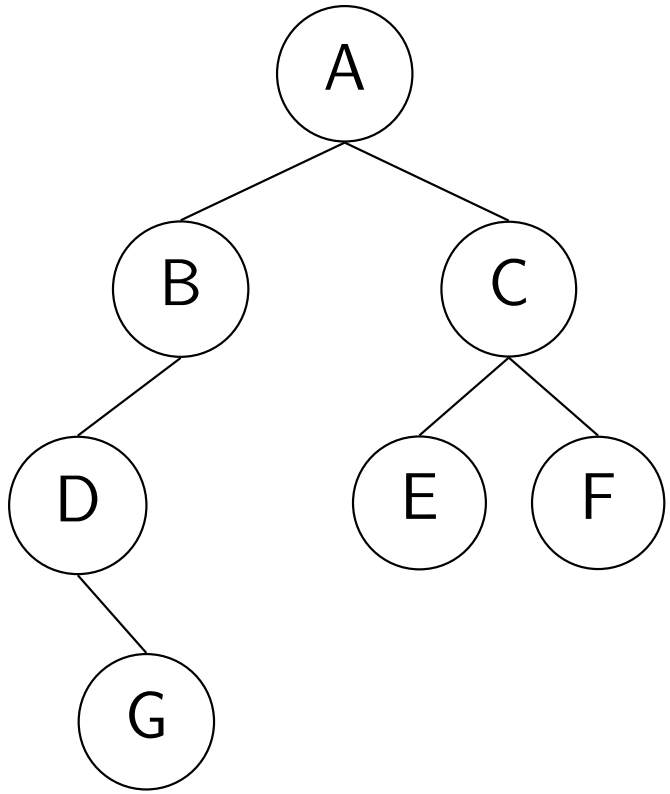
二叉树的遍历（例子）

- 层序遍历就是：A,B,C,D,E,F,G。



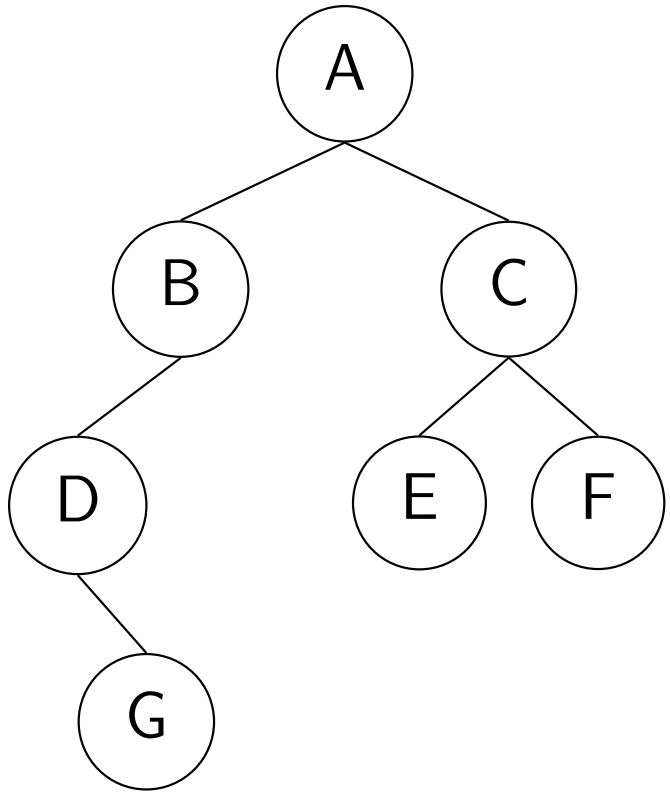
二叉树的遍历（例子）

- 层序遍历就是：
  A,B,C,D,E,F,G。
- 前序：A,B,D,G,C,E,F



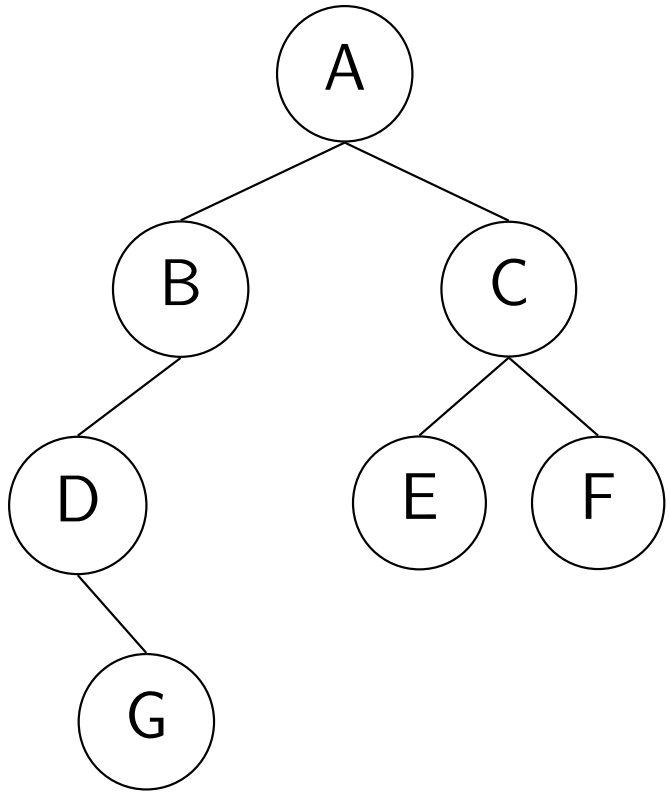
二叉树的遍历（例子）

- 层序遍历就是：
  A,B,C,D,E,F,G。
- 前序：A,B,D,G,C,E,F
- 后序：G,D,B,E,F,C,A



二叉树的遍历（例子）

- 层序遍历就是：
  A,B,C,D,E,F,G。
- 前序：A,B,D,G,C,E,F
- 后序：G,D,B,E,F,C,A
- 中序：D,G,B,A,E,C,F



从遍历结果还原二叉树

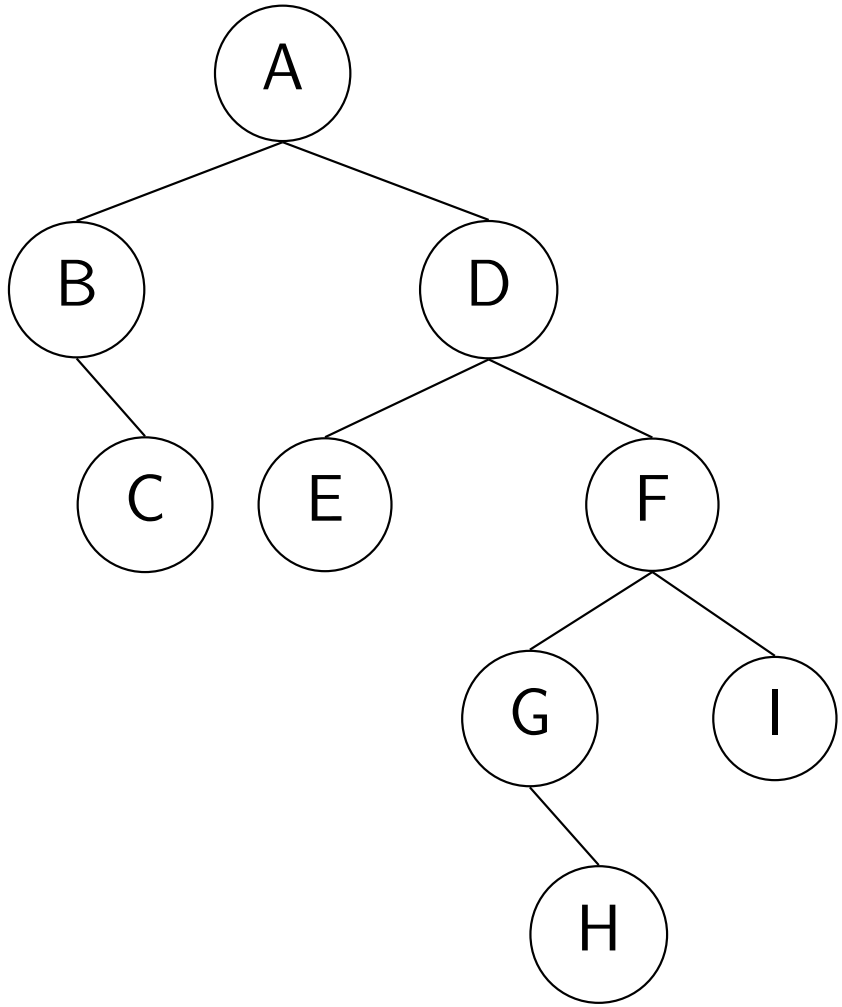
- 当然只知道四种遍历其中一种结果是肯定没可能的。
- 那前序加后序呢？
- 前序加中序呢？

前序加中序

- 假设前序为：
  A,B,C,D,E,F,G,H,I。
- 中序为：
  B,C,A,E,D,G,H,F,I。

前序加中序

- 假设前序为：A,B,C,D,E,F,G,H,I。
- 中序为：B,C,A,E,D,G,H,F,I。



前序加中序：思路

- 前序是根、左、右
- 中序是左、根、右
- 那一开始可以通过前序知道根结点
- 之后从中序的根结点位置，我们可以把左右子树分开来。
- 这样得到的左子树的前序与中序，右子树类同。

顺序结构

- 容易用连续结构的：满二叉树
- 根据我们之前知道的完全二叉树的特性，也可以用连续结构实现：完全二叉树
- 其他二叉树也可以用顺序结构，但会有不少空间浪费

顺序结构

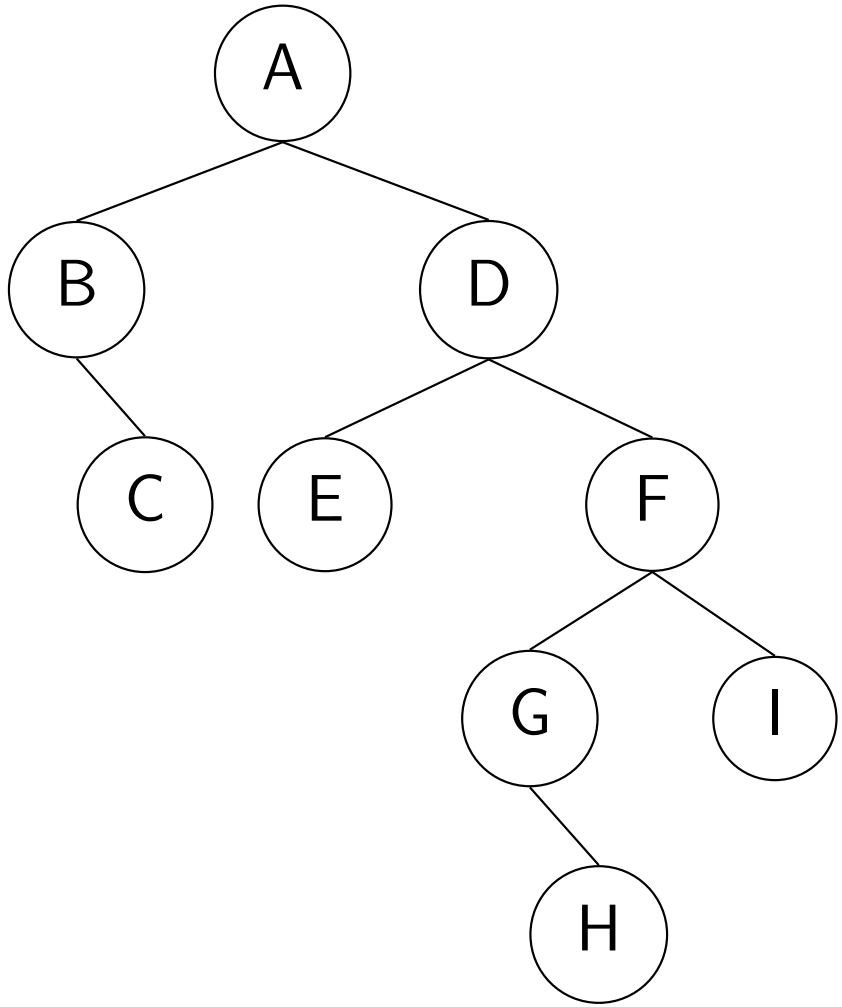
- 容易用连续结构的：满二叉树
- 根据我们之前知道的完全二叉树的特性，也可以用连续结构实现：完全二叉树
- 其他二叉树也可以用顺序结构，但会有不少空间浪费
- 一般情况我们只用顺序结构来存储完全二叉树

链接结构

- 链接结构是很自然的
- 每个结点配一个左指针、一个右指针
- 当然怎么说还会有空指针，但是比顺序结构要来的强。
- 严谨来说，那些空指针都说明了没有子树。

链接结构

- 链接结构是很自然的
- 每个结点配一个左指针、一个右指针
- 当然怎么说还会有空指针，但是比顺序结构要来的强。
- 严谨来说，那些空指针都说明了没有子树。

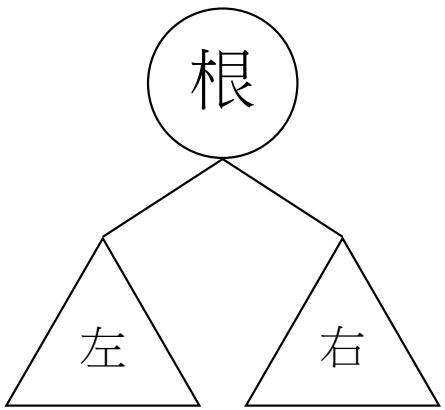


链接结构：前序

- 在假设链接结构下，前序的伪代码：

1. 访问根结点。
2. 若左子树非空，调用前序函数访问左子树。
3. 若右子树非空，调用前序函数访问右子树。

- 中序、后序同理。



链接结构：层序

- 在假设链接结构下，层序的难点在与如何找到下一个要访问的结点。
- 伪代码：

链接结构：层序

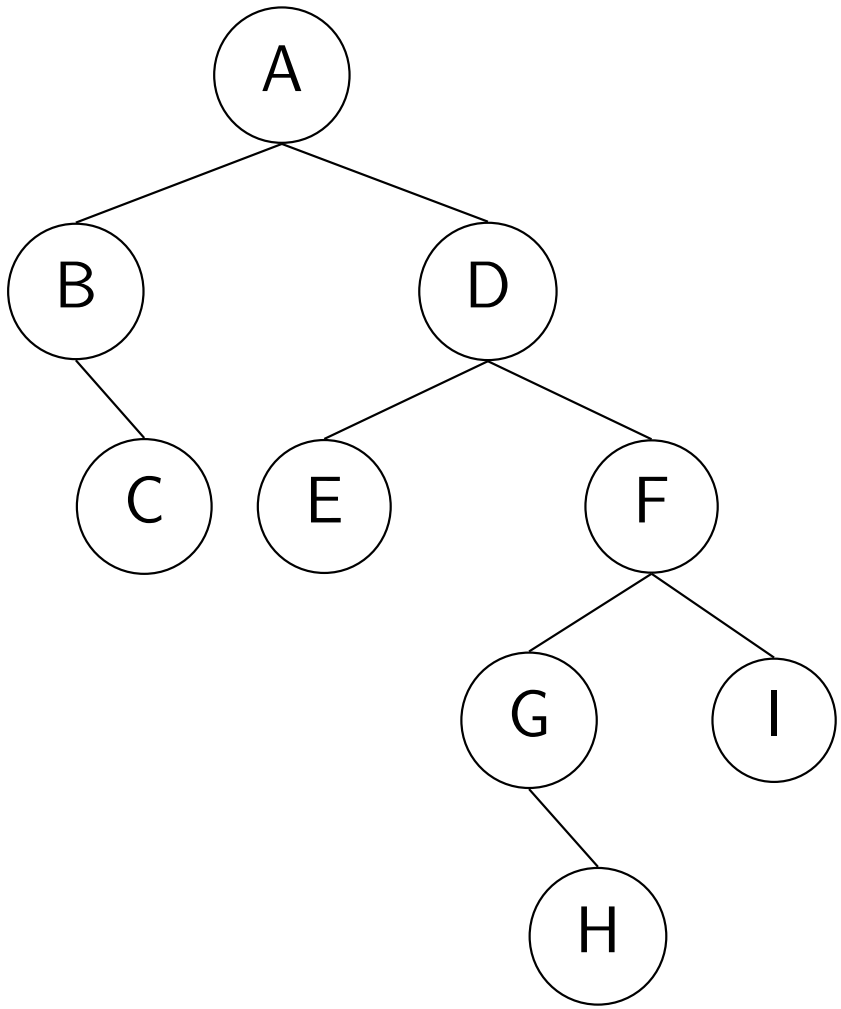
- 在假设链接结构下，层序的难点在与如何找到下一个要访问的结点。
- 伪代码：

1 设置一个空队 Q，并把根结点入队

2 当 Q 非空时：

2A 出队首个结点，访问该结点

2B 把该结点的左、右孩子（若存在）入队



队伍：

链接结构：层序

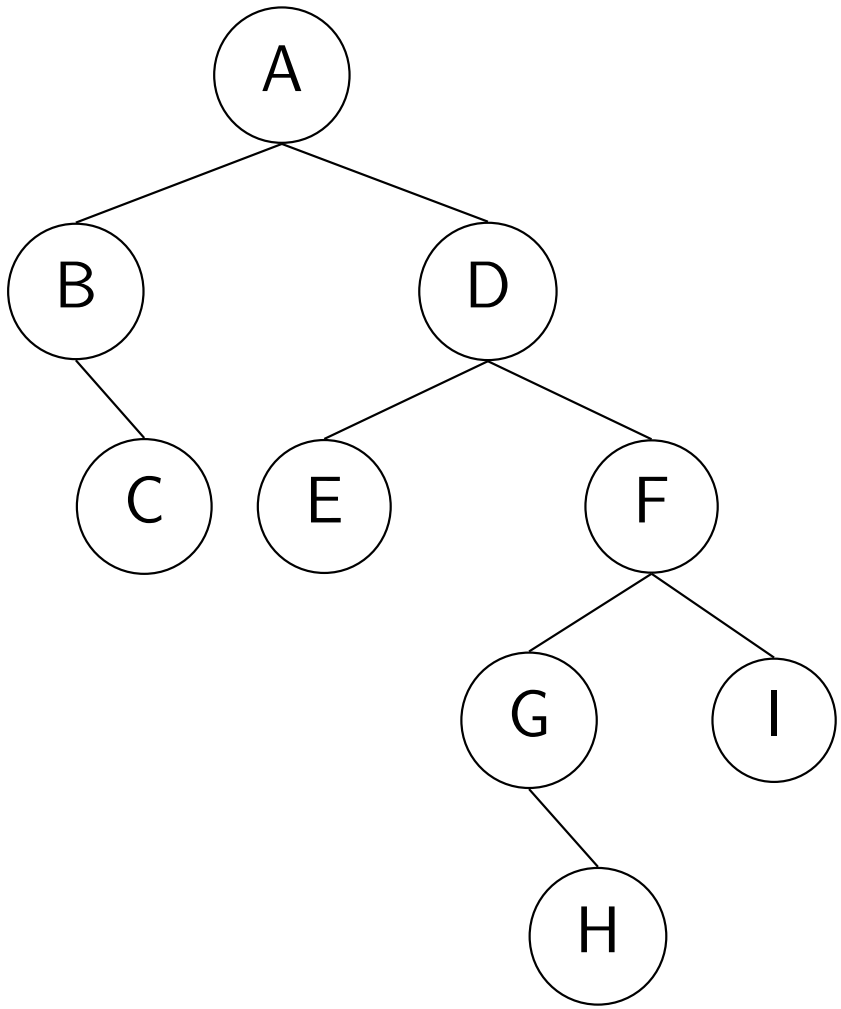
- 在假设链接结构下，层序的难点在与如何找到下一个要访问的结点。
- 伪代码：

1 设置一个空队 Q，并把根结点入队

2 当 Q 非空时：

2A 出队首个结点，访问该结点

2B 把该结点的左、右孩子（若存在）入队



队伍：A

链接结构：层序

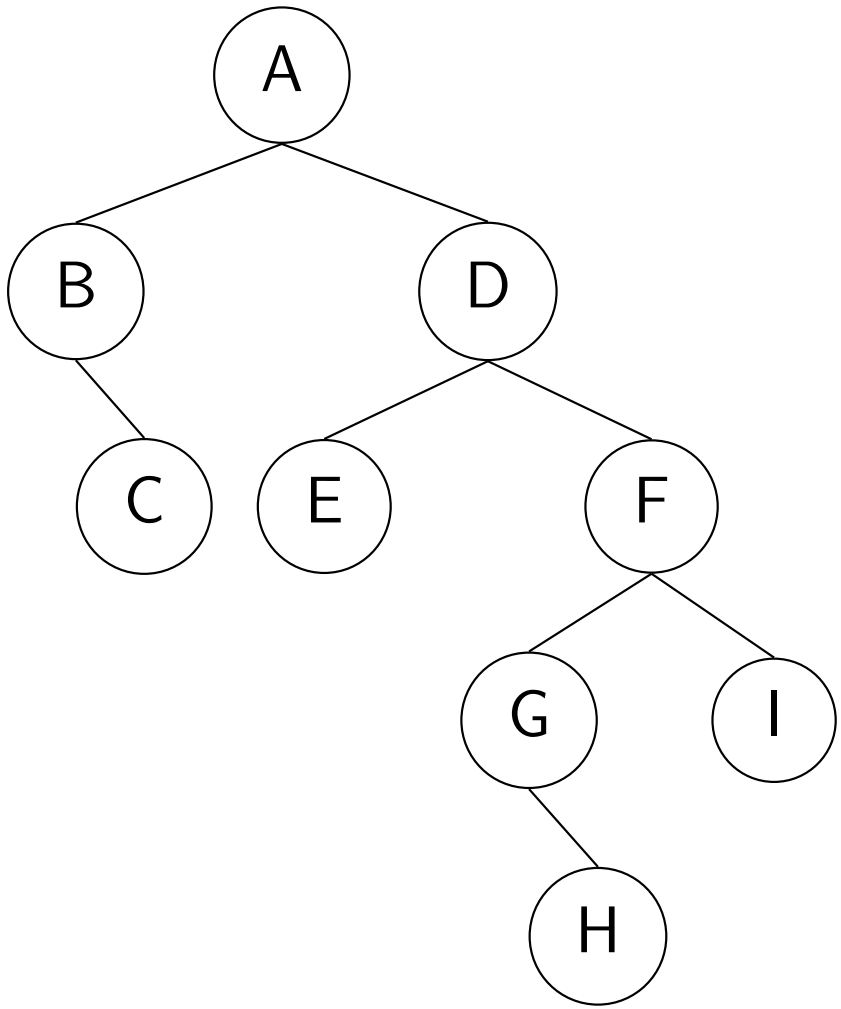
- 在假设链接结构下，层序的难点在与如何找到下一个要访问的结点。
- 伪代码：

1 设置一个空队 Q，并把根结点入队

2 当 Q 非空时：

2A 出队首个结点，访问该结点

2B 把该结点的左、右孩子（若存在）入队



队伍： B D

链接结构：层序

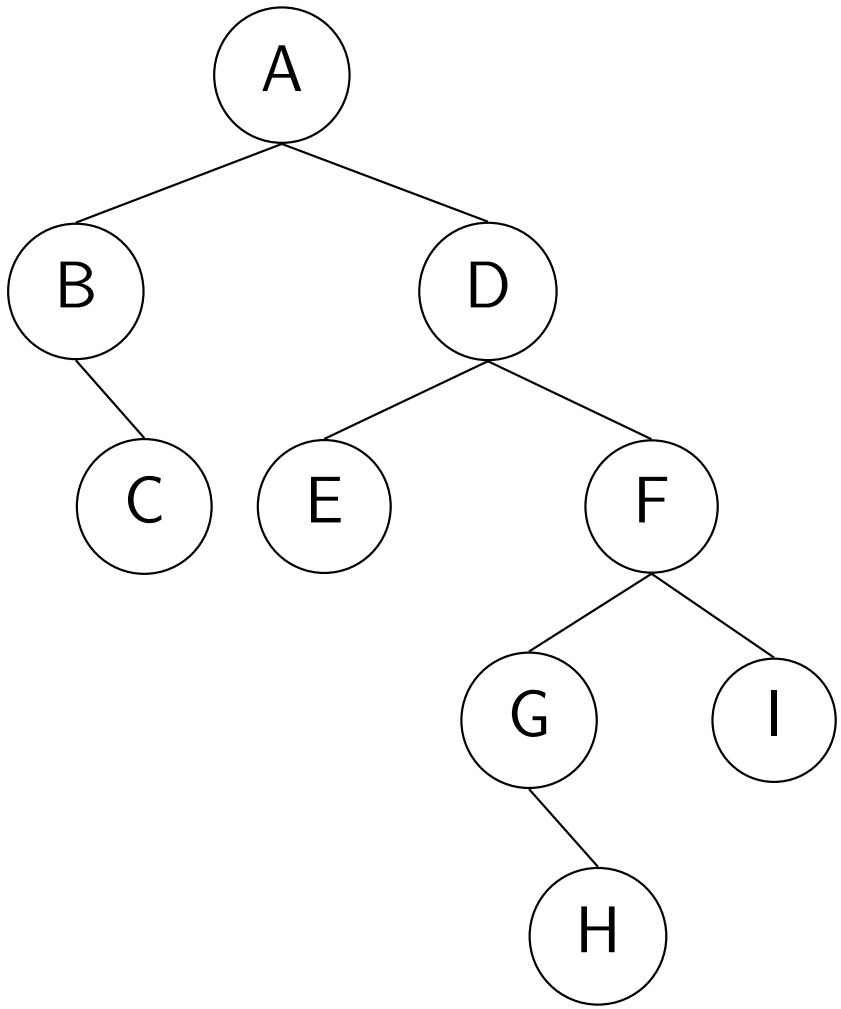
- 在假设链接结构下，层序的难点在与如何找到下一个要访问的结点。
- 伪代码：

1 设置一个空队 Q，并把根结点入队

2 当 Q 非空时：

2A 出队首个结点，访问该结点

2B 把该结点的左、右孩子（若存在）入队



队伍： D C

链接结构：层序

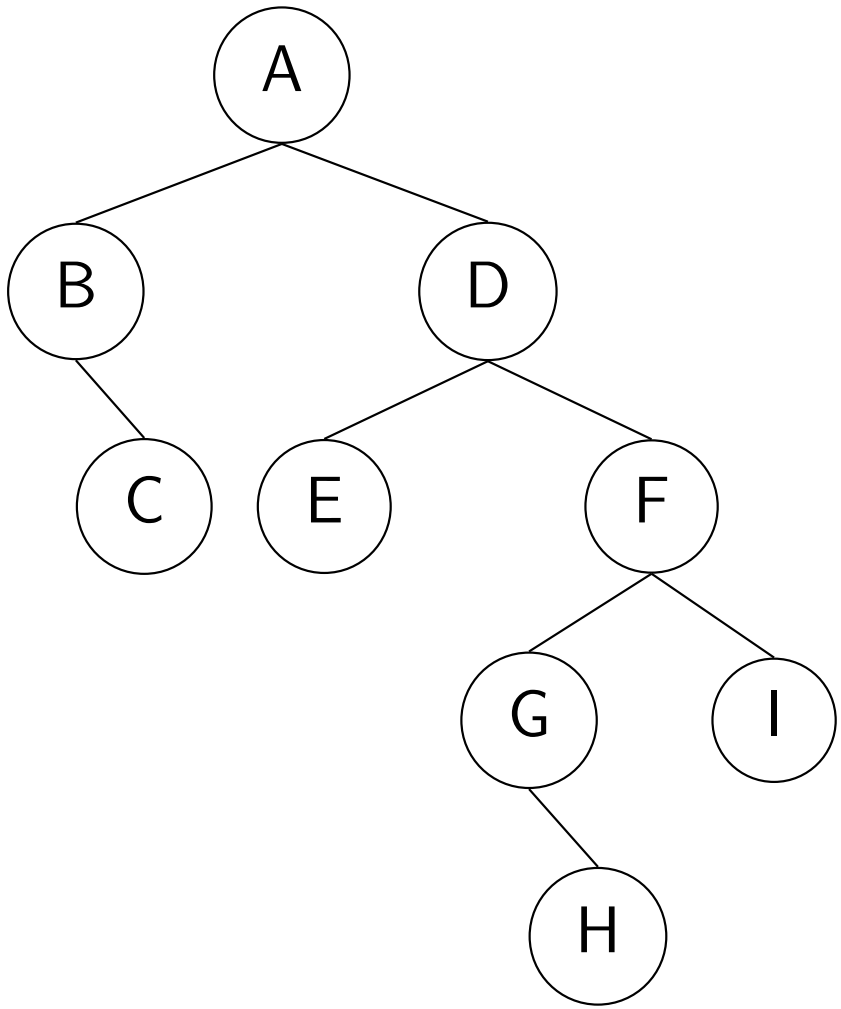
- 在假设链接结构下，层序的难点在与如何找到下一个要访问的结点。
- 伪代码：

1 设置一个空队 Q，并把根结点入队

2 当 Q 非空时：

2A 出队首个结点，访问该结点

2B 把该结点的左、右孩子（若存在）入队



队伍： C E F

链接结构：层序

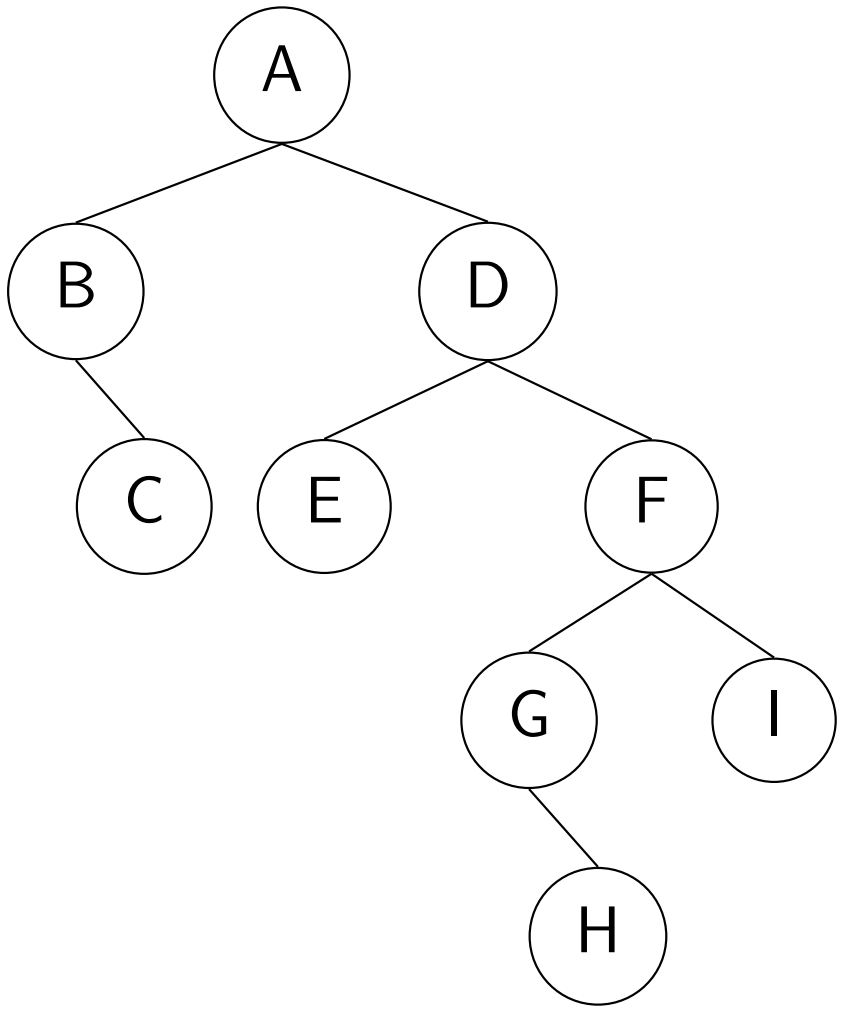
- 在假设链接结构下，层序的难点在与如何找到下一个要访问的结点。
- 伪代码：

1 设置一个空队 Q，并把根结点入队

2 当 Q 非空时：

2A 出队首个结点，访问该结点

2B 把该结点的左、右孩子（若存在）入队



队伍： E F

链接结构：层序

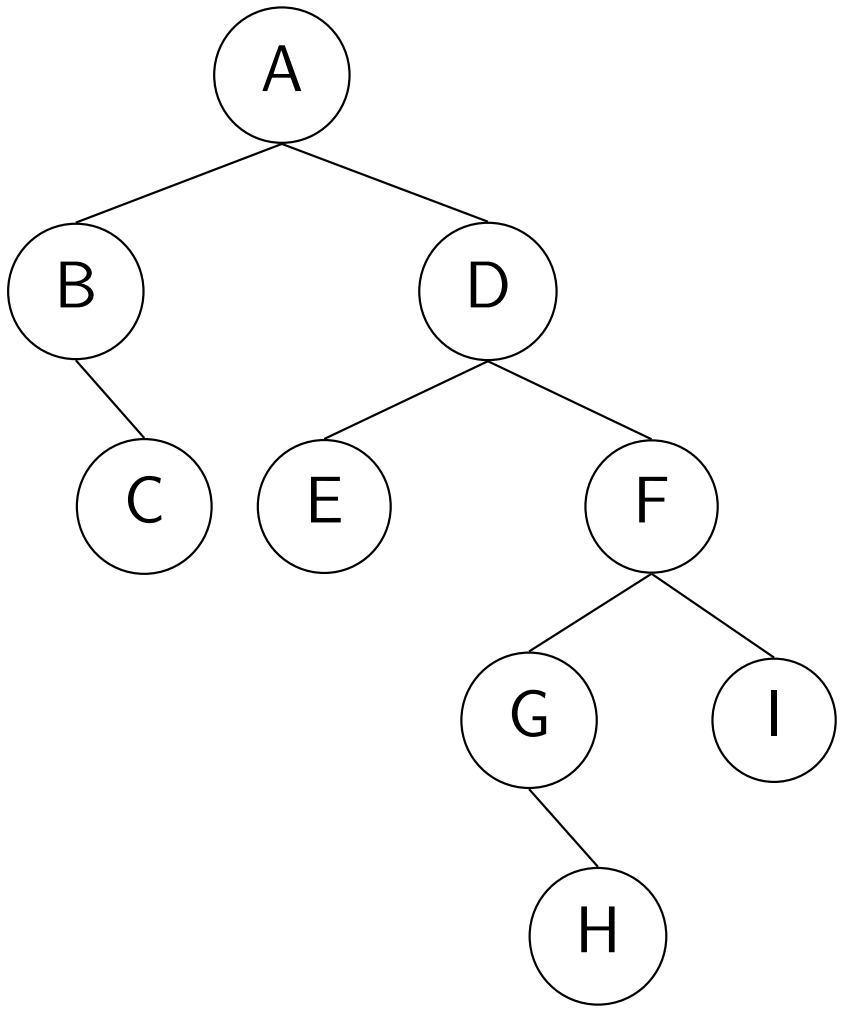
- 在假设链接结构下，层序的难点在与如何找到下一个要访问的结点。
- 伪代码：

1 设置一个空队 Q，并把根结点入队

2 当 Q 非空时：

2A 出队首个结点，访问该结点

2B 把该结点的左、右孩子（若存在）入队



队伍： F

链接结构：层序

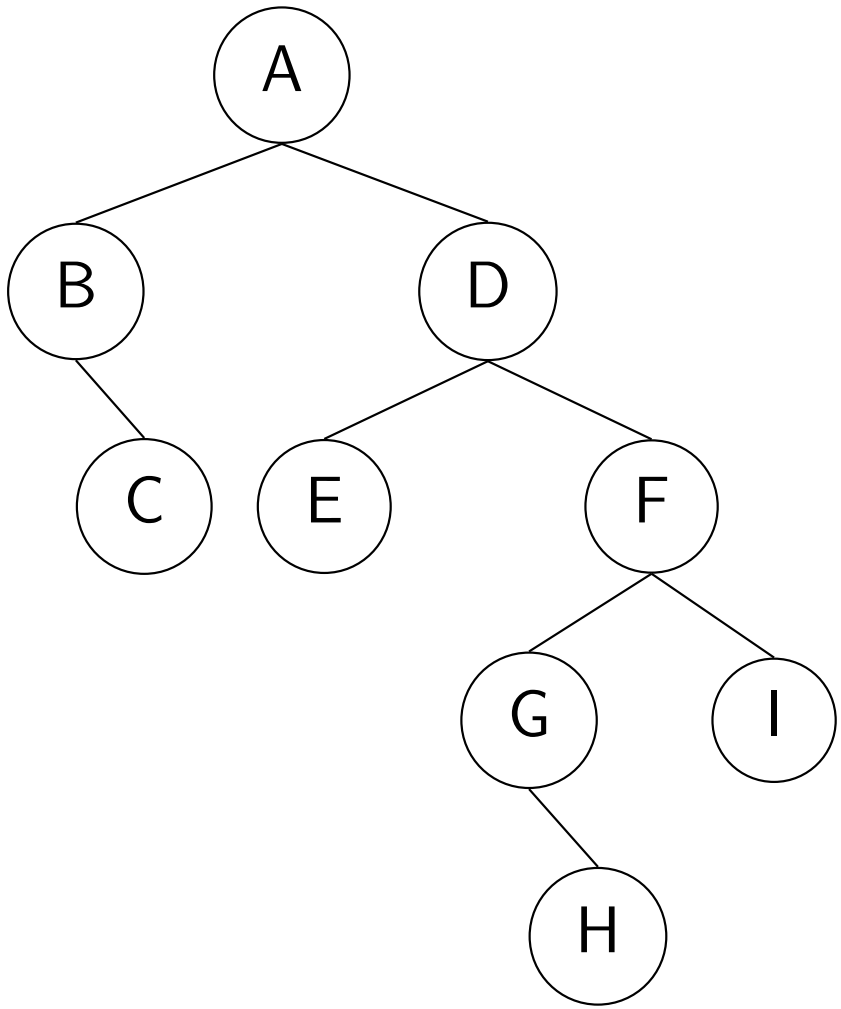
- 在假设链接结构下，层序的难点在与如何找到下一个要访问的结点。
- 伪代码：

1 设置一个空队 Q，并把根结点入队

2 当 Q 非空时：

2A 出队首个结点，访问该结点

2B 把该结点的左、右孩子（若存在）入队



队伍： G I

链接结构：层序

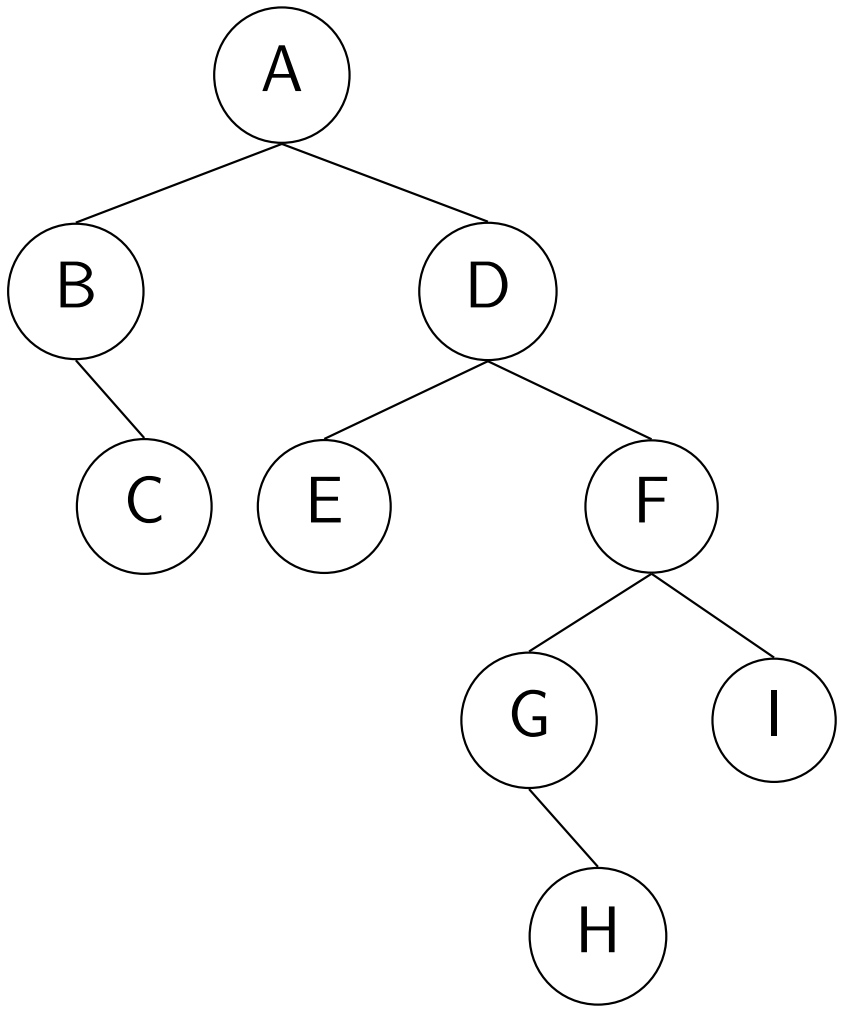
- 在假设链接结构下，层序的难点在与如何找到下一个要访问的结点。
- 伪代码：

1 设置一个空队 Q，并把根结点入队

2 当 Q 非空时：

2A 出队首个结点，访问该结点

2B 把该结点的左、右孩子（若存在）入队



队伍： I H

链接结构：层序

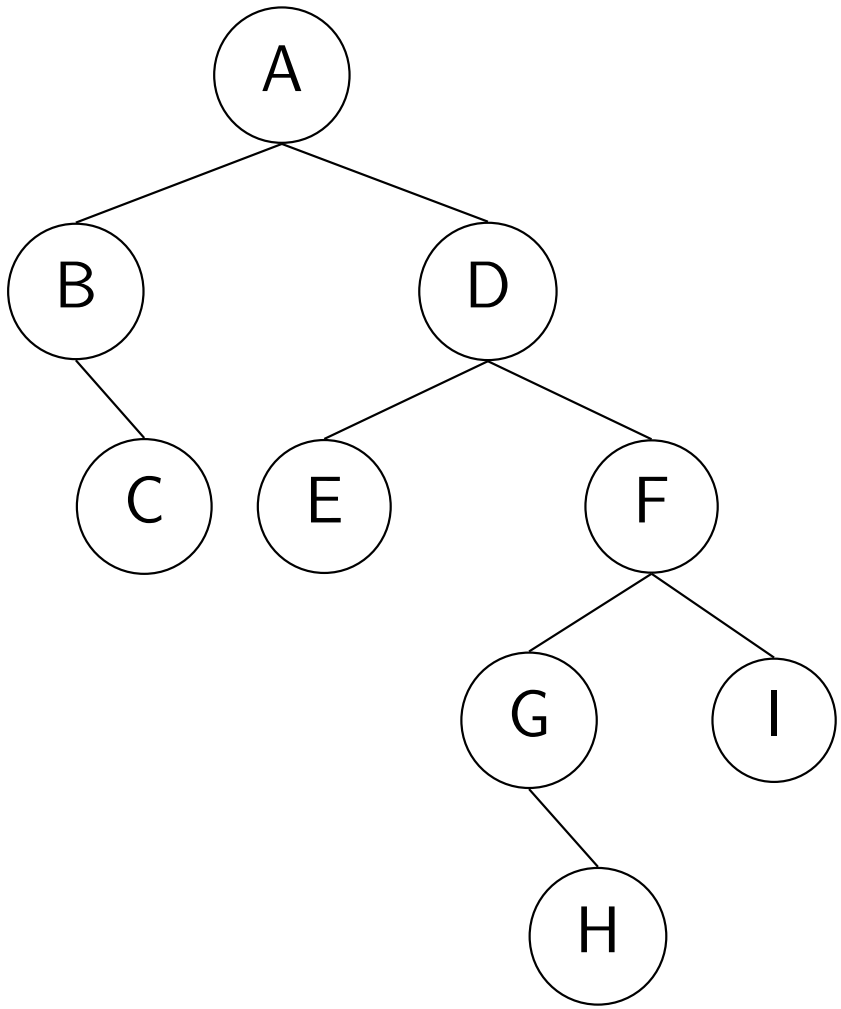
- 在假设链接结构下，层序的难点在与如何找到下一个要访问的结点。
- 伪代码：

1 设置一个空队 Q，并把根结点入队

2 当 Q 非空时：

2A 出队首个结点，访问该结点

2B 把该结点的左、右孩子（若存在）入队



队伍： H

例子：树的结点个数

- 写一个函数算一下树的结点总数：

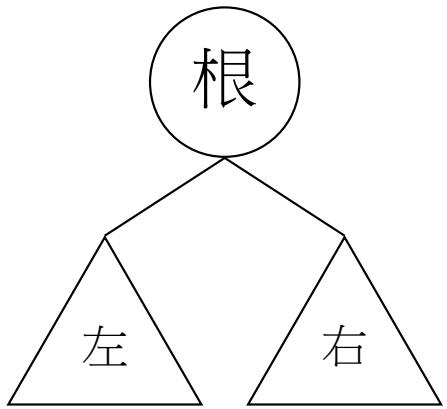
例子：树的结点个数

- 写一个函数算一下树的结点总数：
- 伪代码：

1 设 $L = 0$。若左子树非空，调用同样函数计算左子树结点数。

2 设 $R = 0$。若右子树非空，调用同样函数计算右子树结点数。

3 回传 $L + R + 1$。



空指针可能的用途

- 线索链表：把空指针用与加速前序、后序或中序的遍历速度（下以前序为例）
- 概念：让空的左指针指向前序的前驱，空的右指针指向前序的后继
- 那当然还要有另外的变量声明这个到底是指向孩子还是被物尽其用的空指针。

前序的非递归算法

- 前序是根、左、右的顺序。
- 由于根结点后马上就是左子树，所以难题是怎么在跑完整个左子树后找到右子树的根结点？

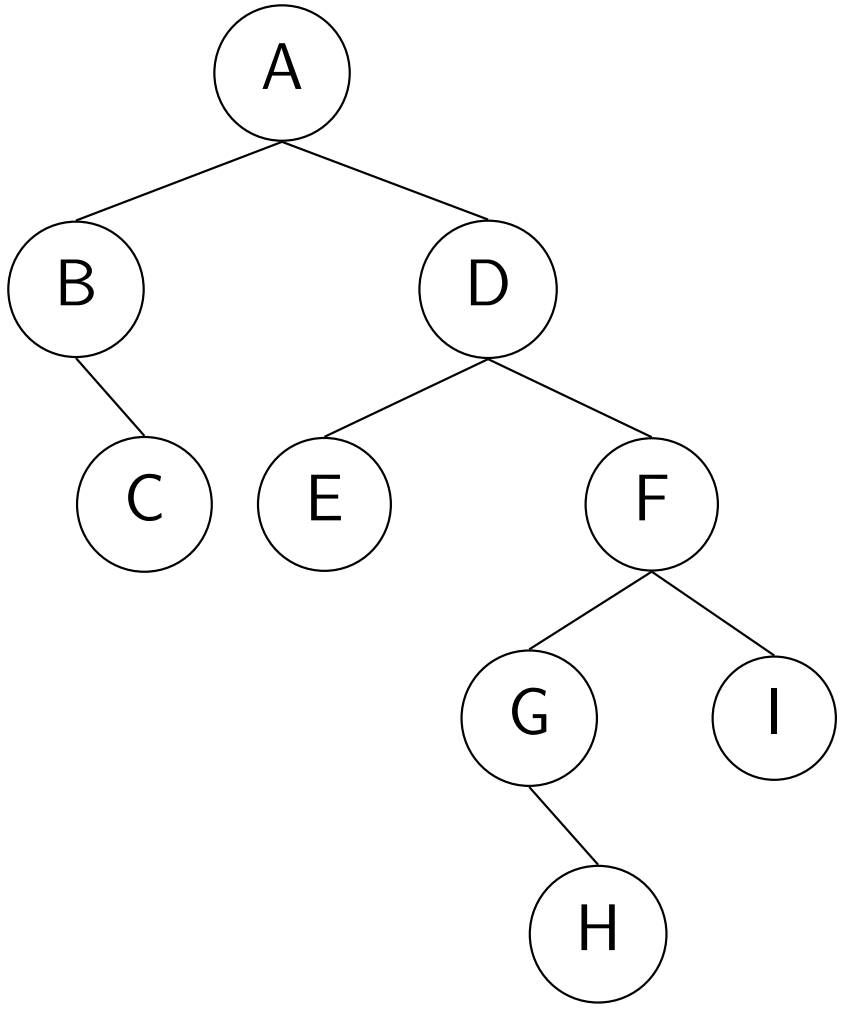
前序的非递归算法

- 前序是根、左、右的顺序。
- 由于根结点后马上就是左子树，所以难题是怎么在跑完整个左子树后找到右子树的根结点？
- 栈！
- 其实某程度每次递归时我们都有用到栈的，只是不用我们特别处理而已。
- 在这个例子里我们就是把所有遇到过的右子树都存到栈里。

前序的非递归算法（例子）

栈：

-
-
-

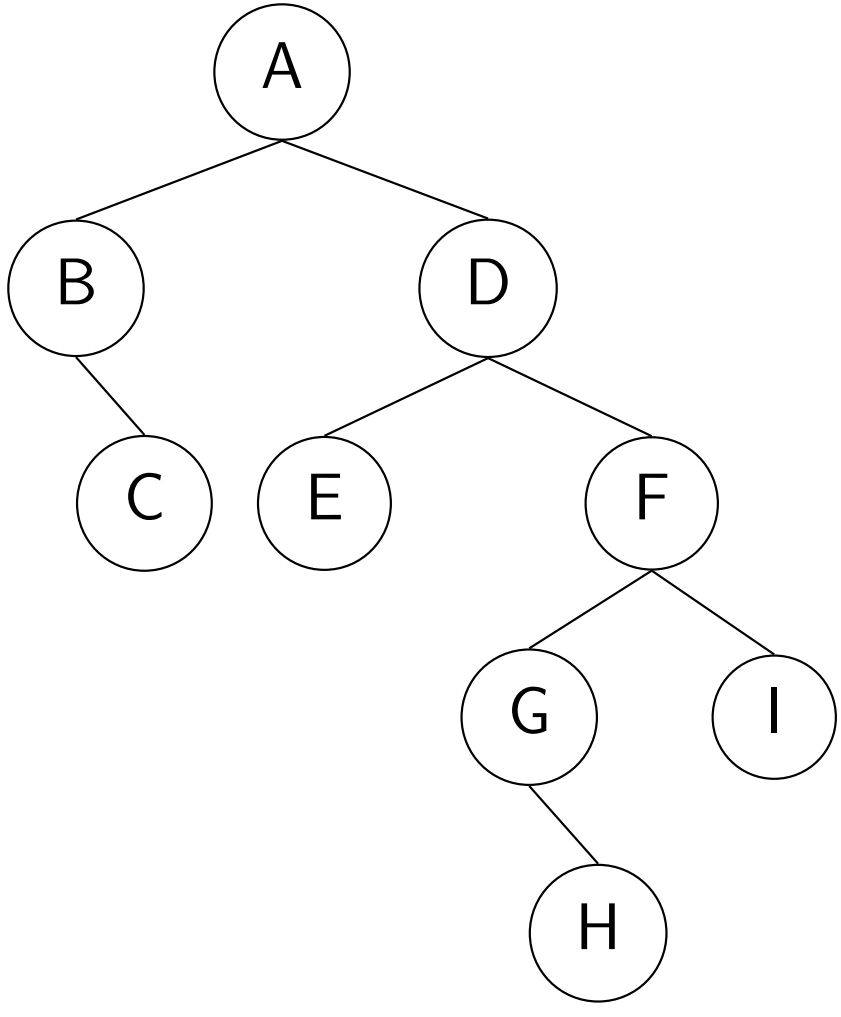


输出：A

前序的非递归算法（例子）

栈：

-
-
- D

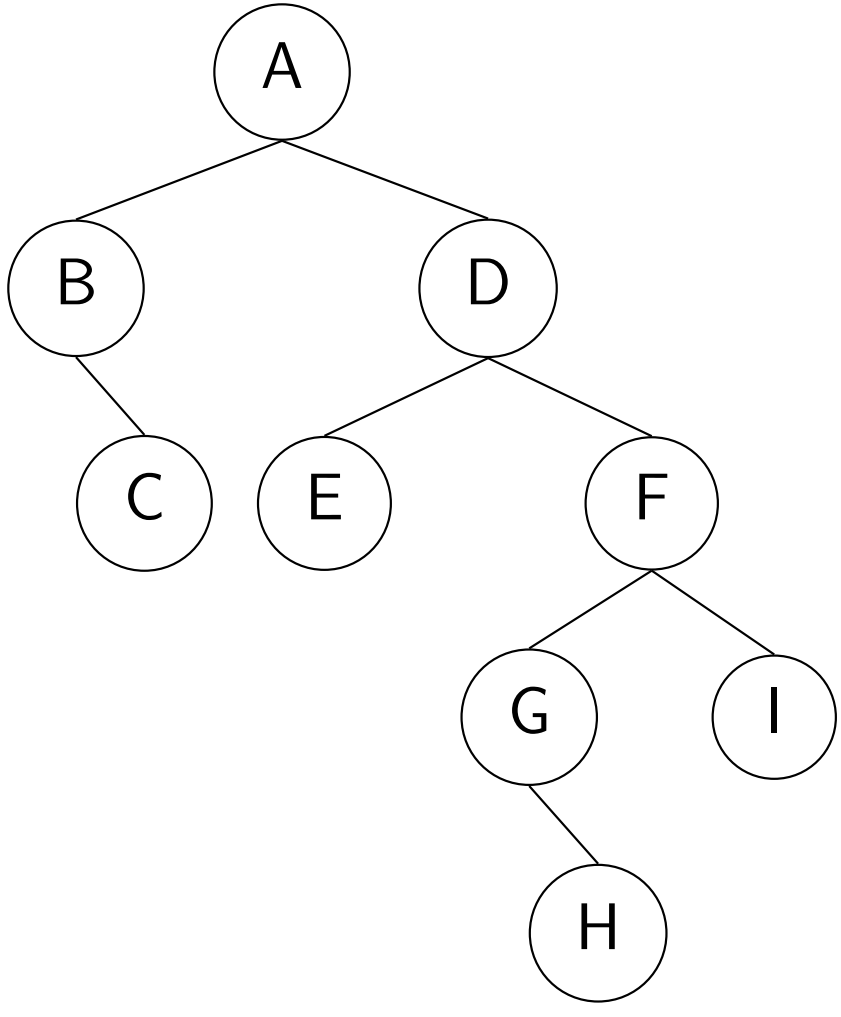


输出：A

前序的非递归算法（例子）

栈：

-
-
- D

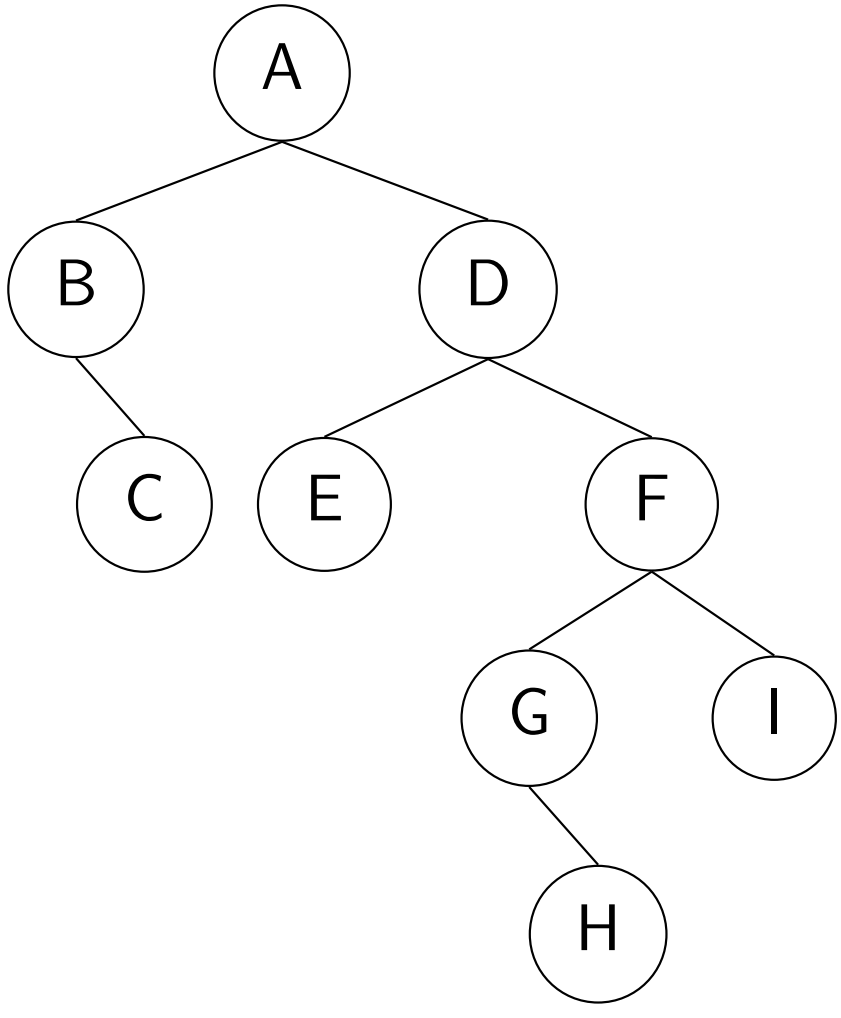


输出：A B

前序的非递归算法（例子）

栈：

-
- C
- D

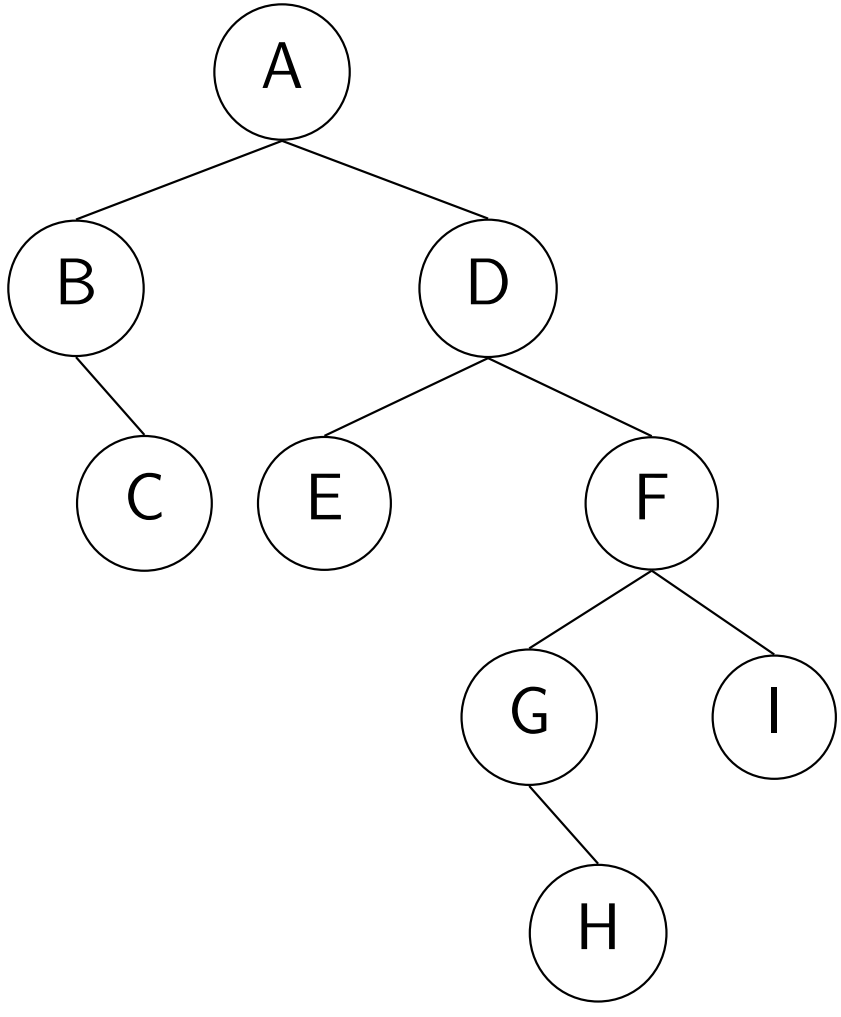


输出：A B

前序的非递归算法（例子）

栈：

-
-
- D

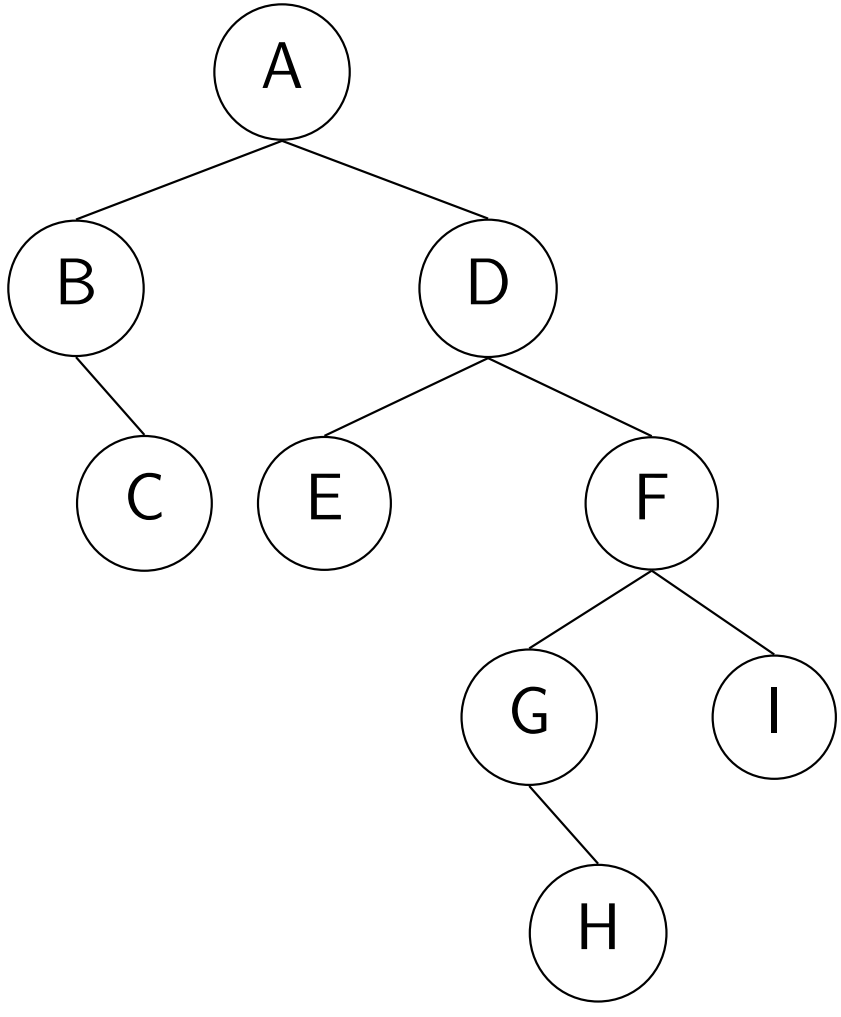


输出：A B C

前序的非递归算法（例子）

栈：

-
-
-

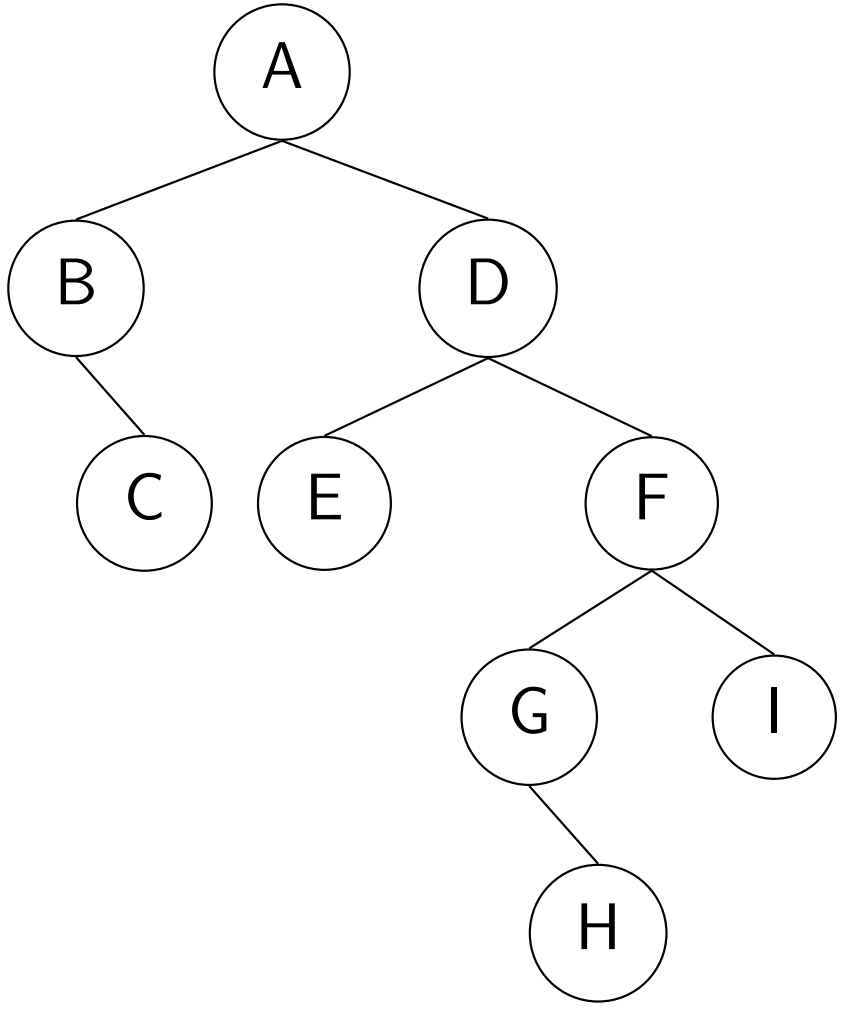


输出：A B C D

前序的非递归算法（例子）

栈：

-
-
- F

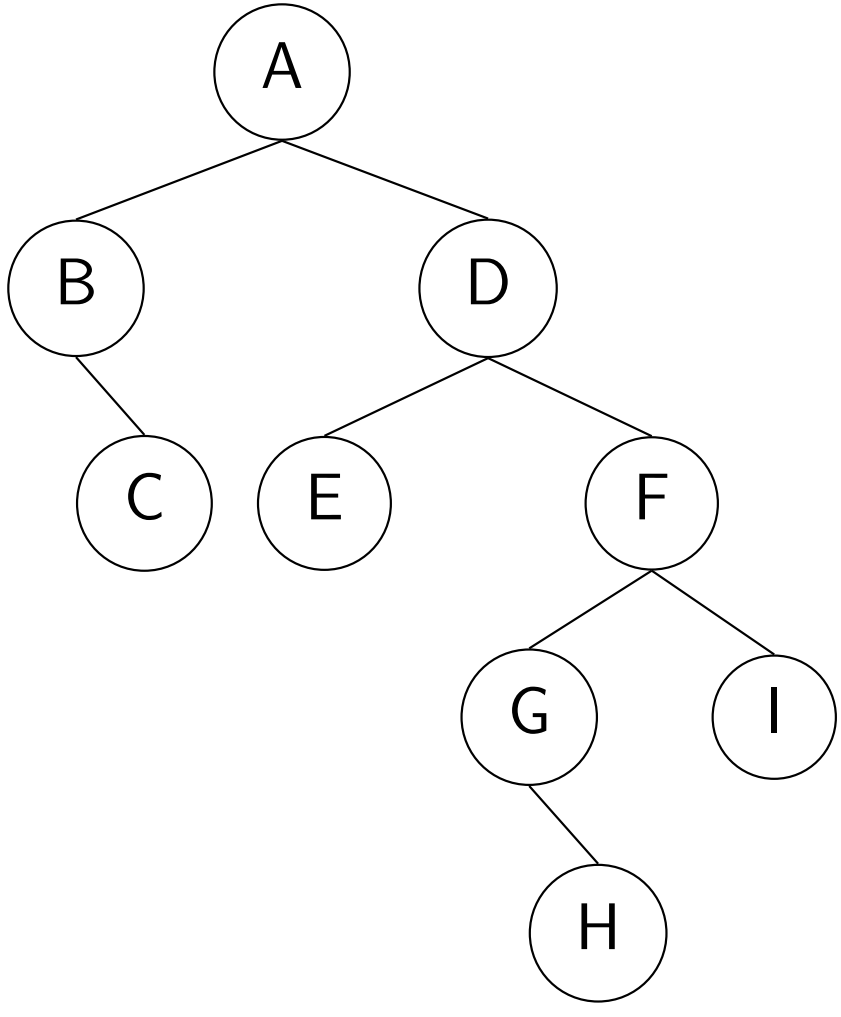


输出：A B C D

前序的非递归算法（例子）

栈：

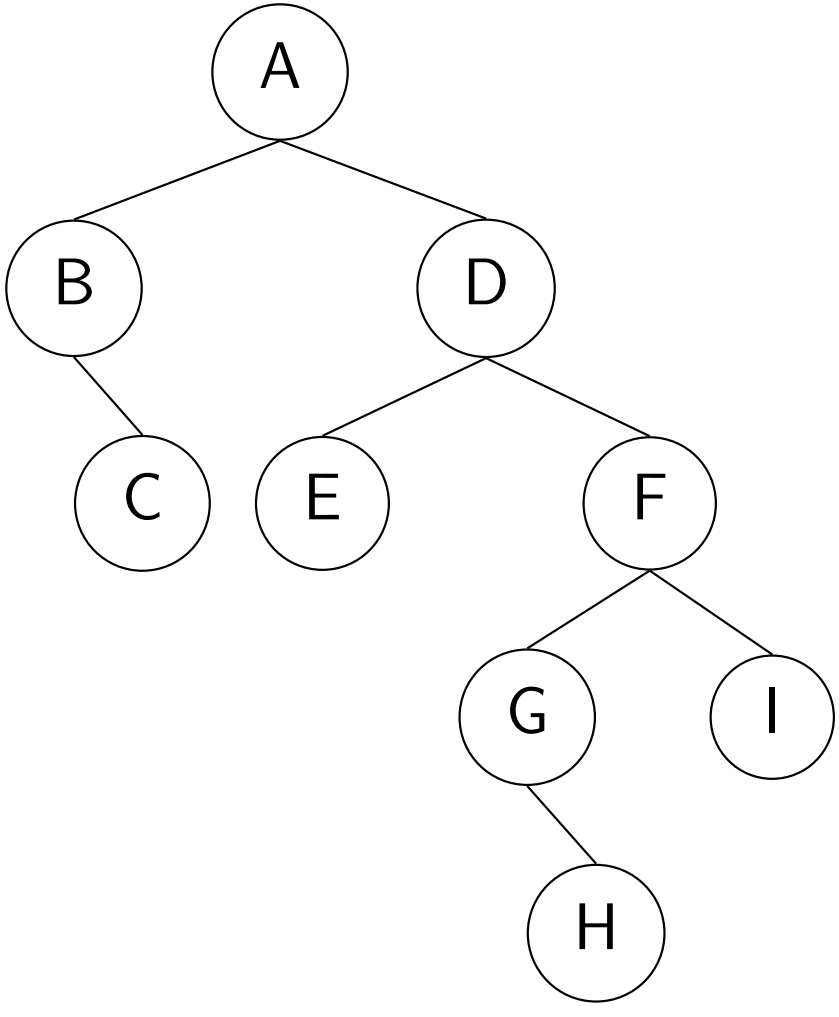
-
-
- F



输出：A B C D E

前序的非递归算法（例子）

栈：

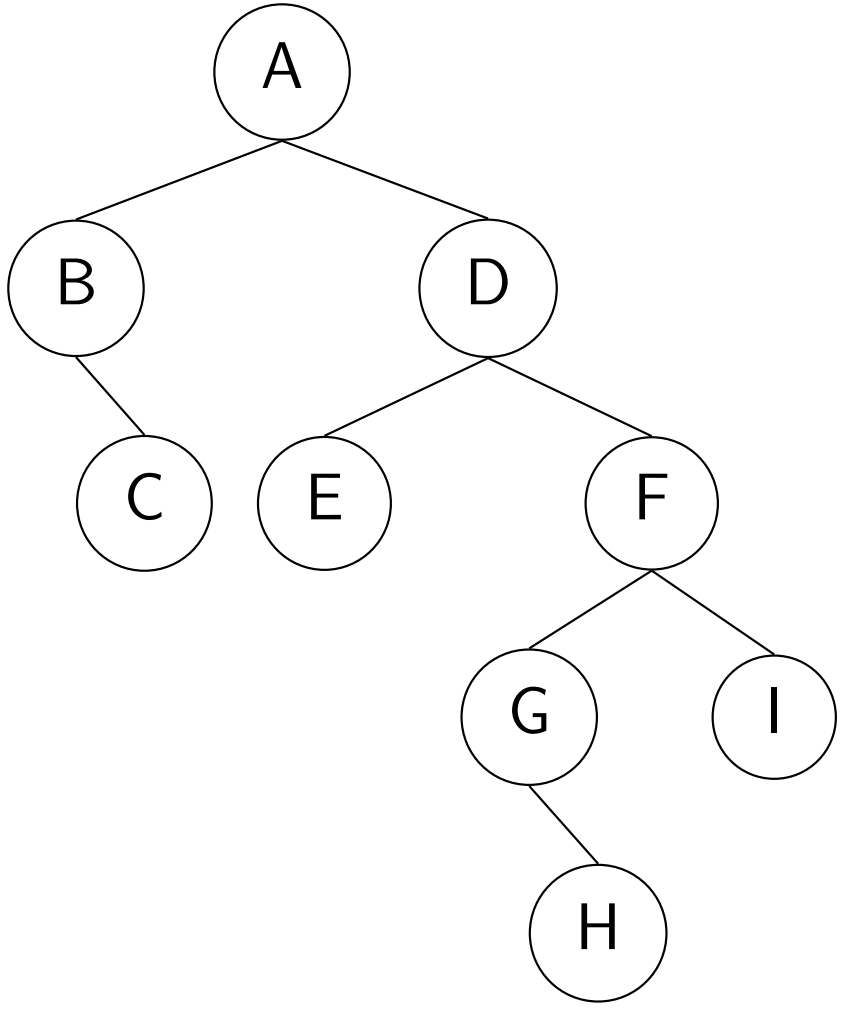


输出：A B C D E F

前序的非递归算法（例子）

栈：

-
-
- I

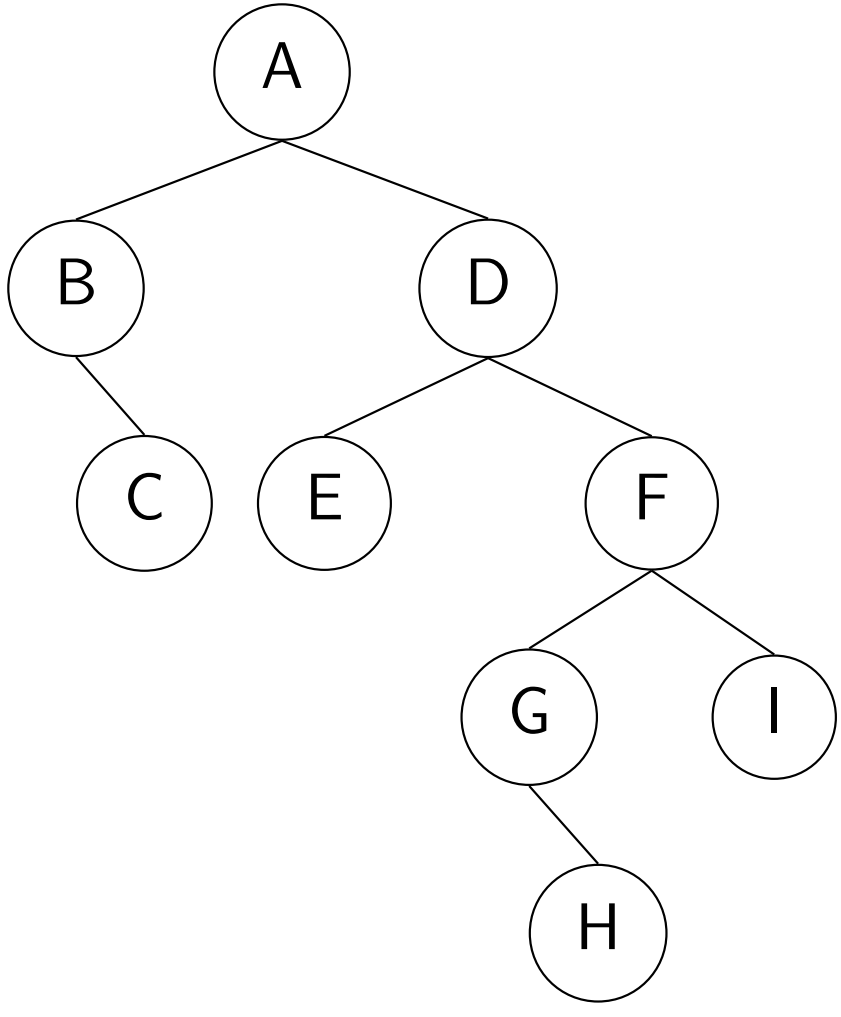


输出：A B C D E F

前序的非递归算法（例子）

栈：

-
-
- I

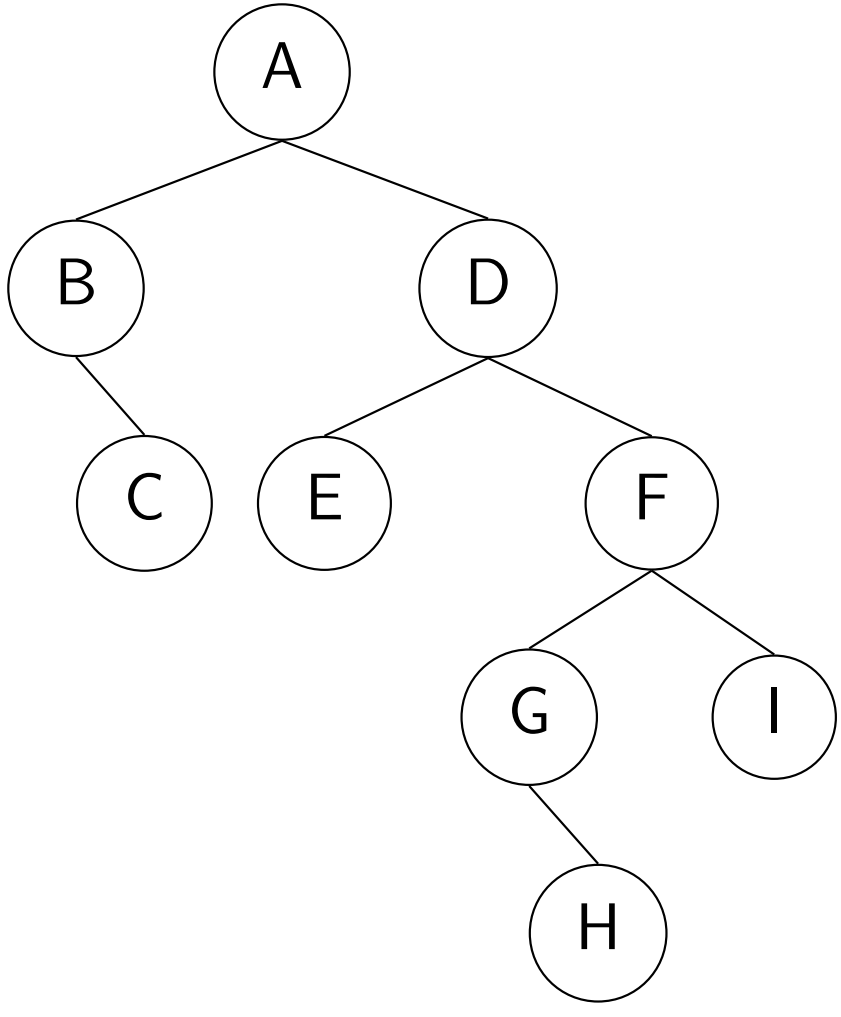


输出：A B C D E F G

前序的非递归算法（例子）

栈：

-
- H
- I

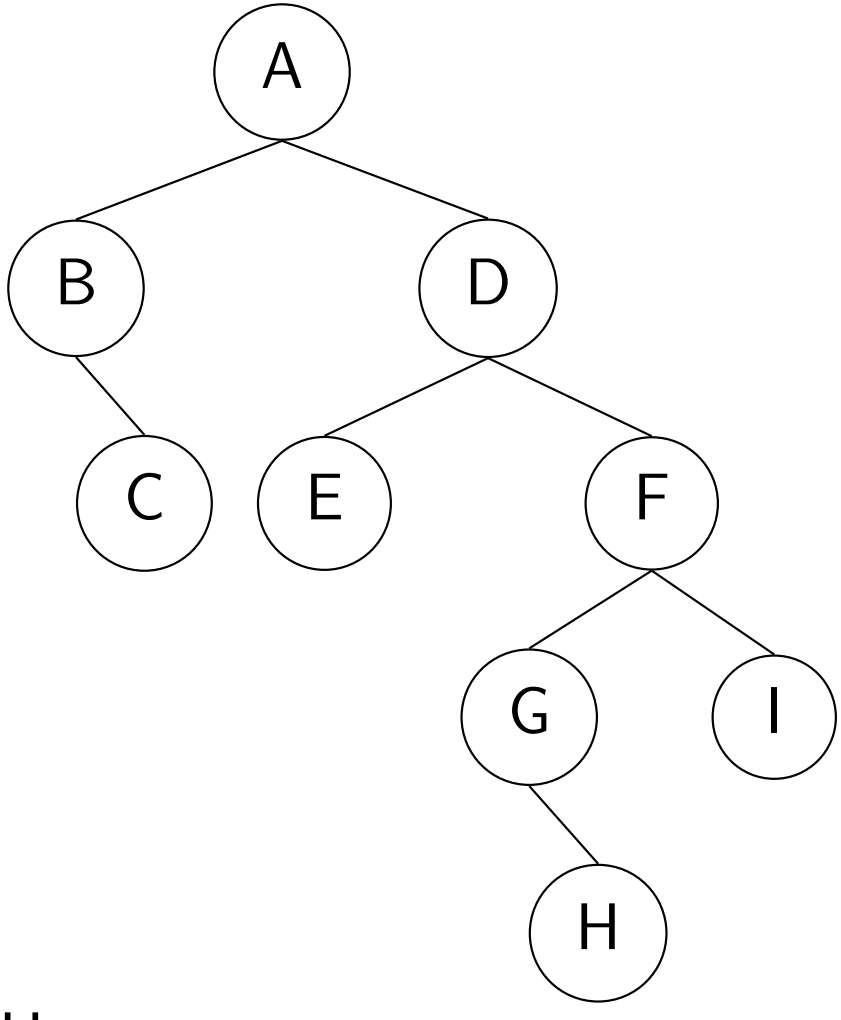


输出：A B C D E F G

前序的非递归算法（例子）

栈：

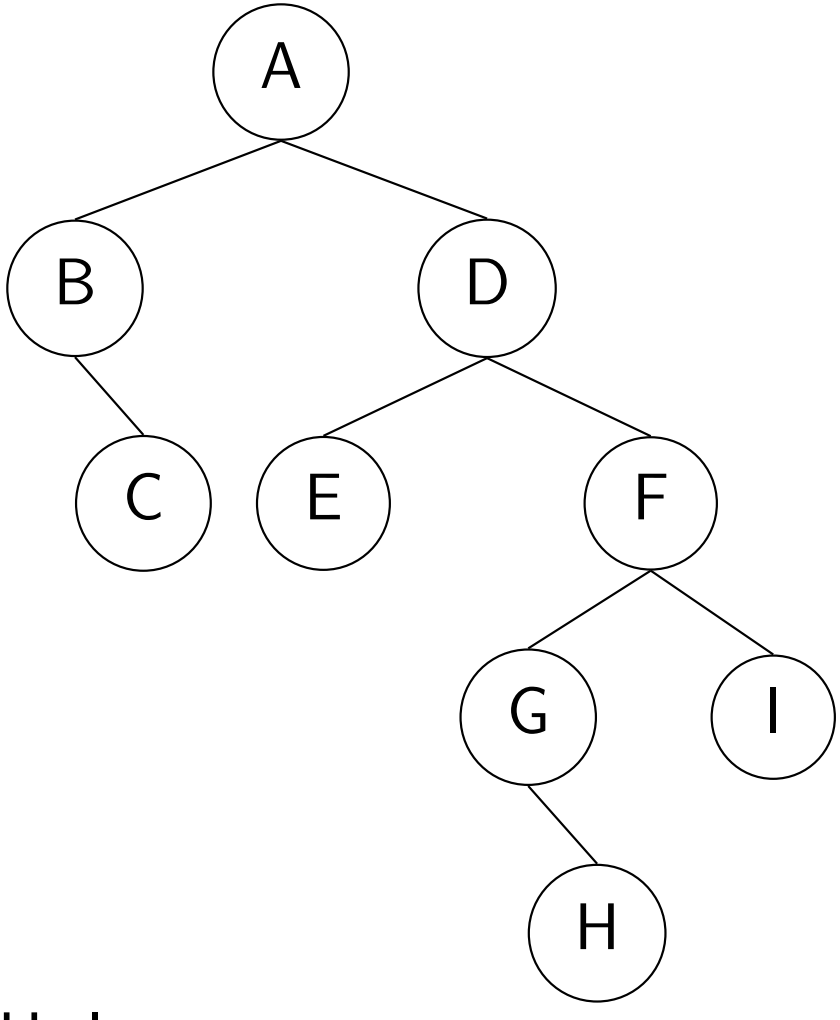
-
-
- I



输出：A B C D E F G H

前序的非递归算法（例子）

栈：



输出：A B C D E F G H I

中序的非递归算法

- 中序是左、根、右的顺序。
- 由于根结点后马上就是右子树，所以难题是怎么在跑完整个左子树后找到根结点？

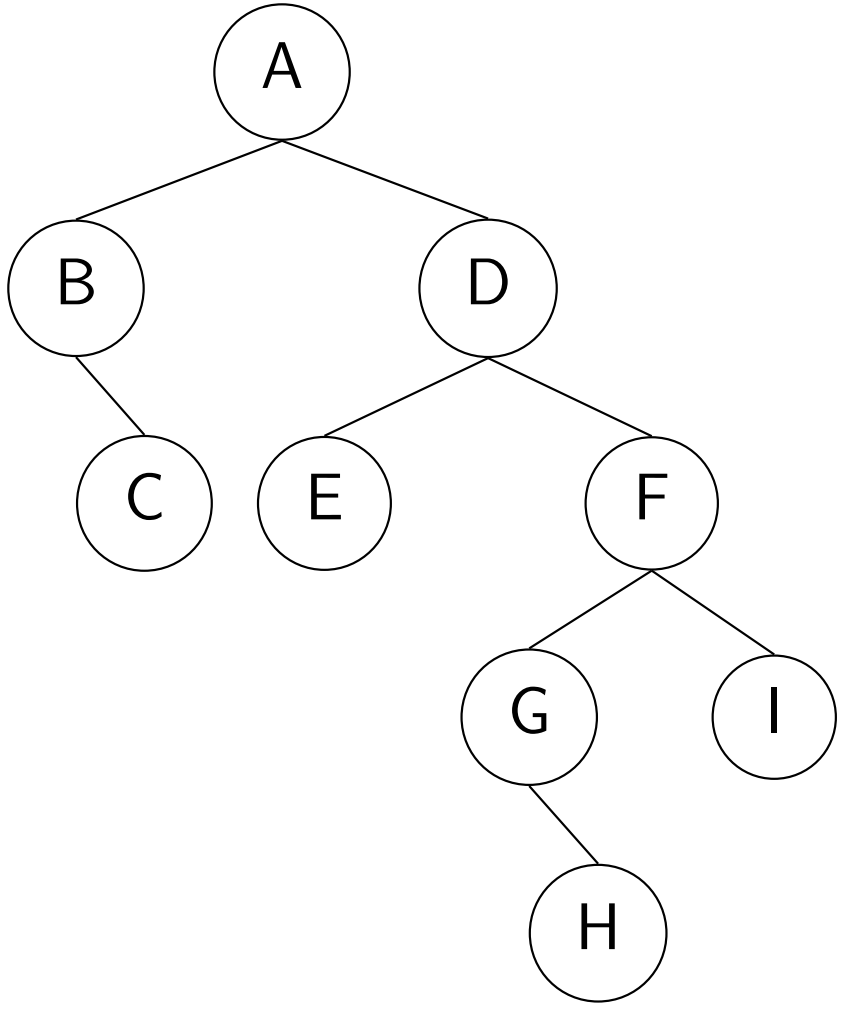
中序的非递归算法

- 中序是左、根、右的顺序。
- 由于根结点后马上就是右子树，所以难题是怎么在跑完整个左子树后找到根结点？
- 栈！
- 在这个例子里我们就是把所有遇到的根结点都存到栈里。

中序的非递归算法（例子）

栈：

-
-
- A

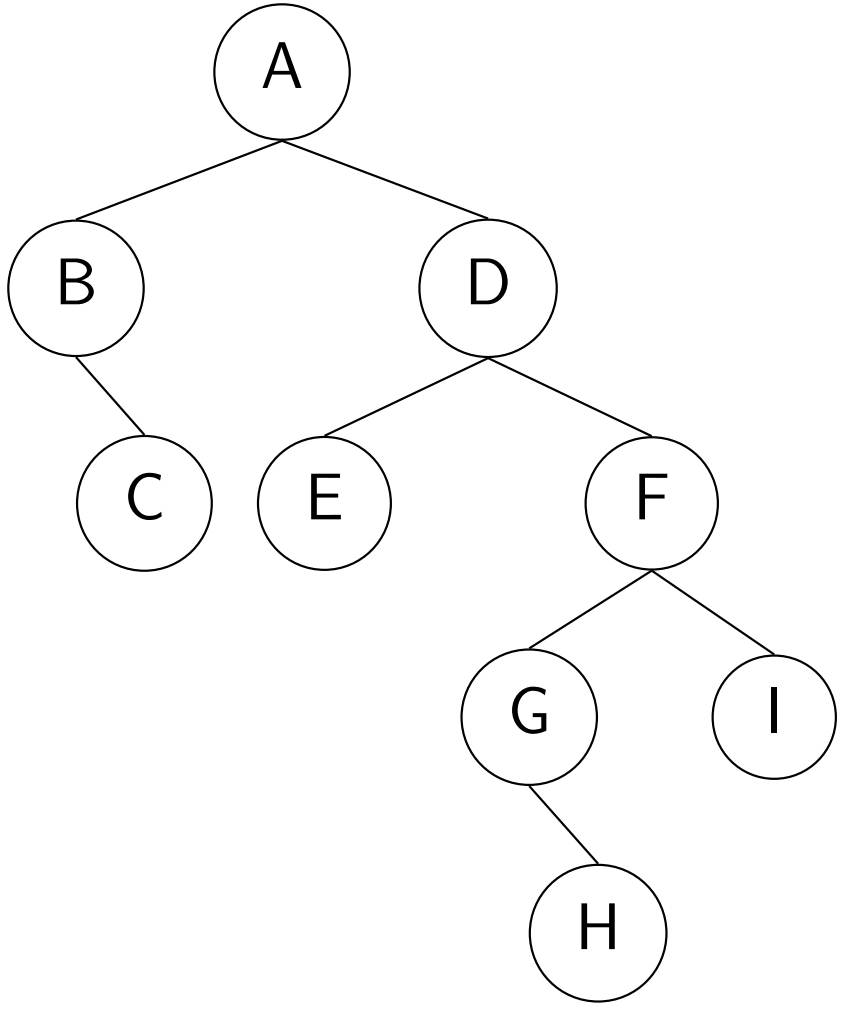


输出：

中序的非递归算法（例子）

栈：

-
- B
- A

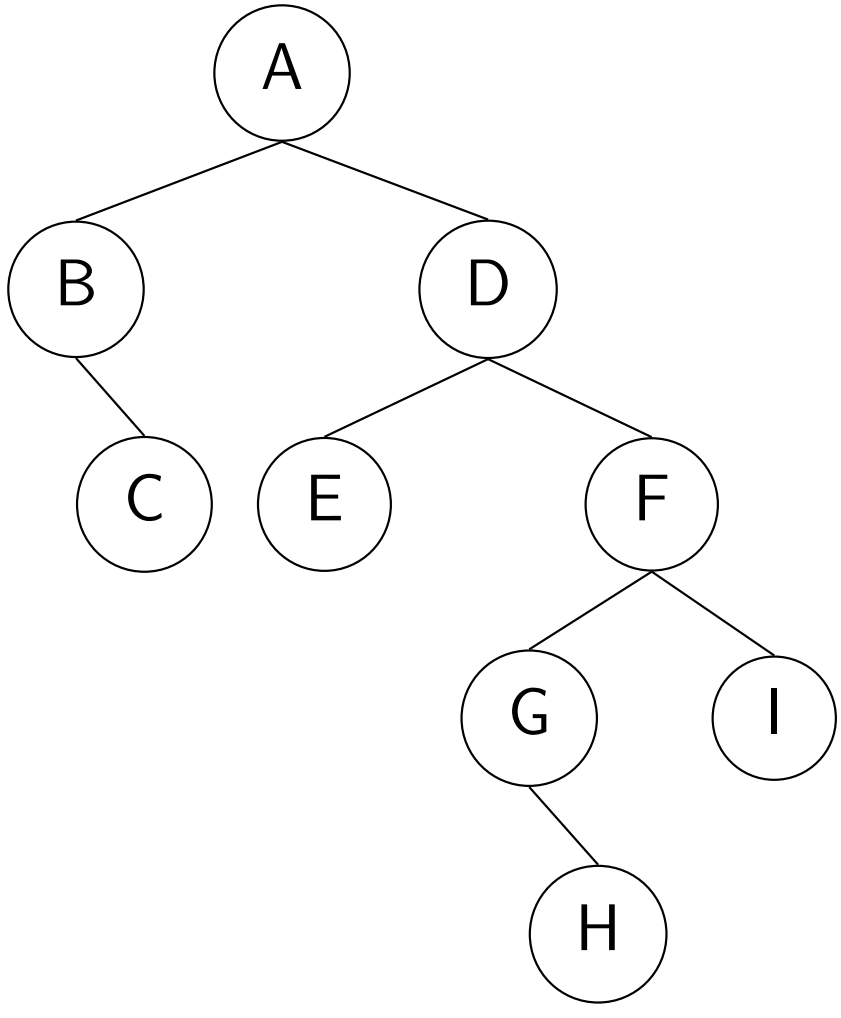


输出：

中序的非递归算法（例子）

栈：

-
-
- A

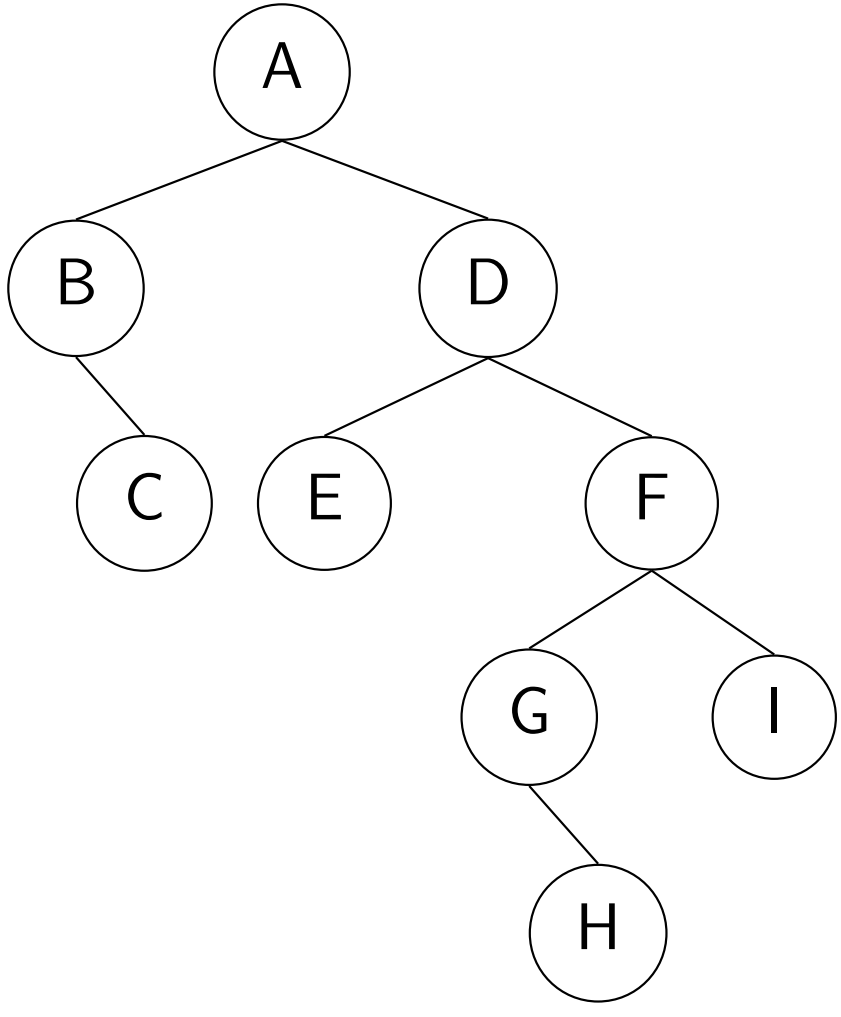


输出：B

中序的非递归算法（例子）

栈：

-
- C
- A

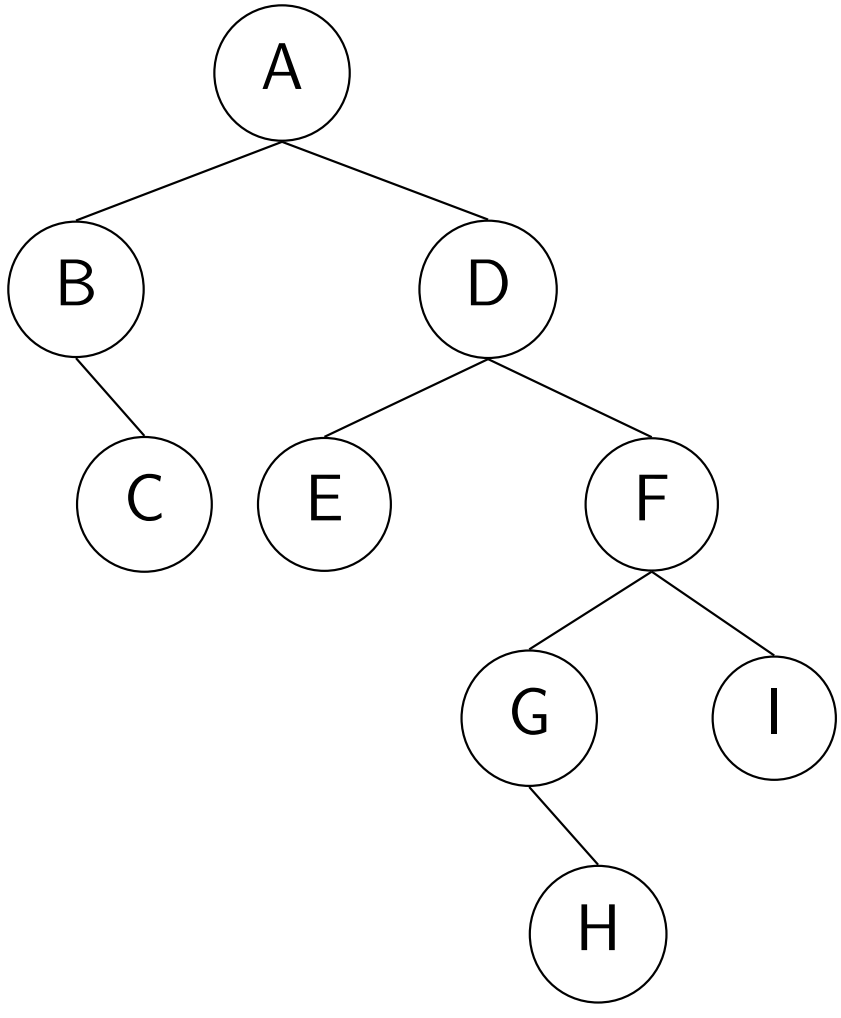


输出：B

中序的非递归算法（例子）

栈：

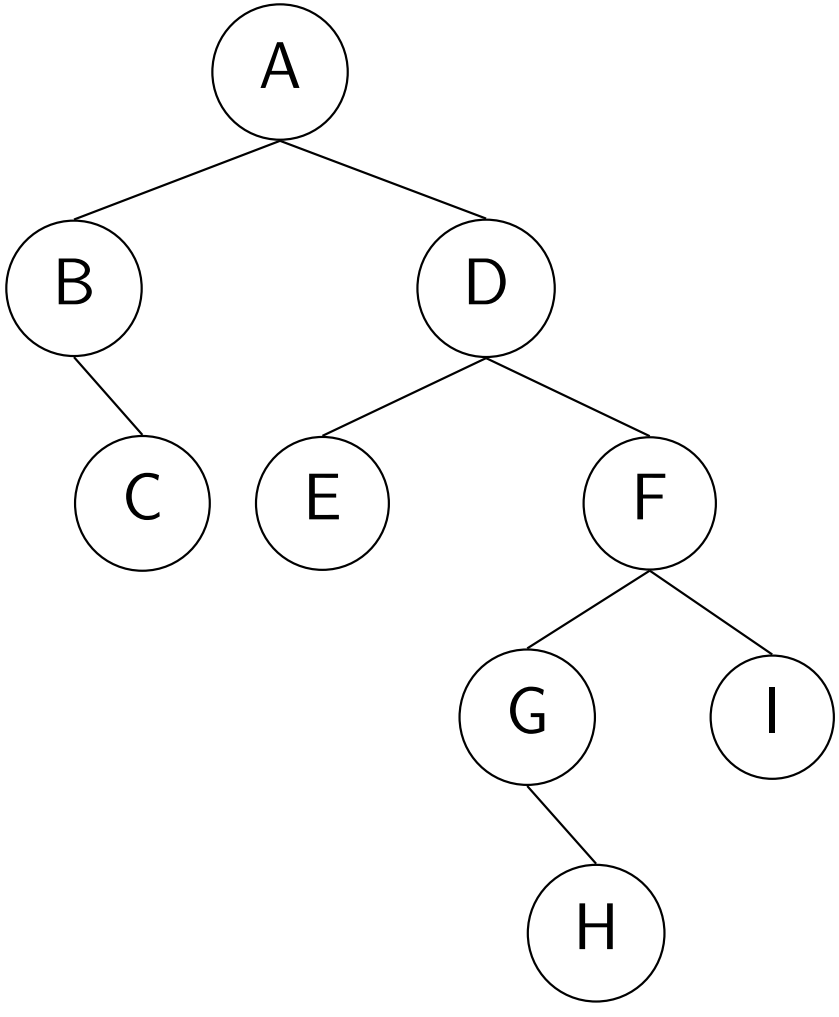
-
-
- A



输出：B C

中序的非递归算法（例子）

栈：

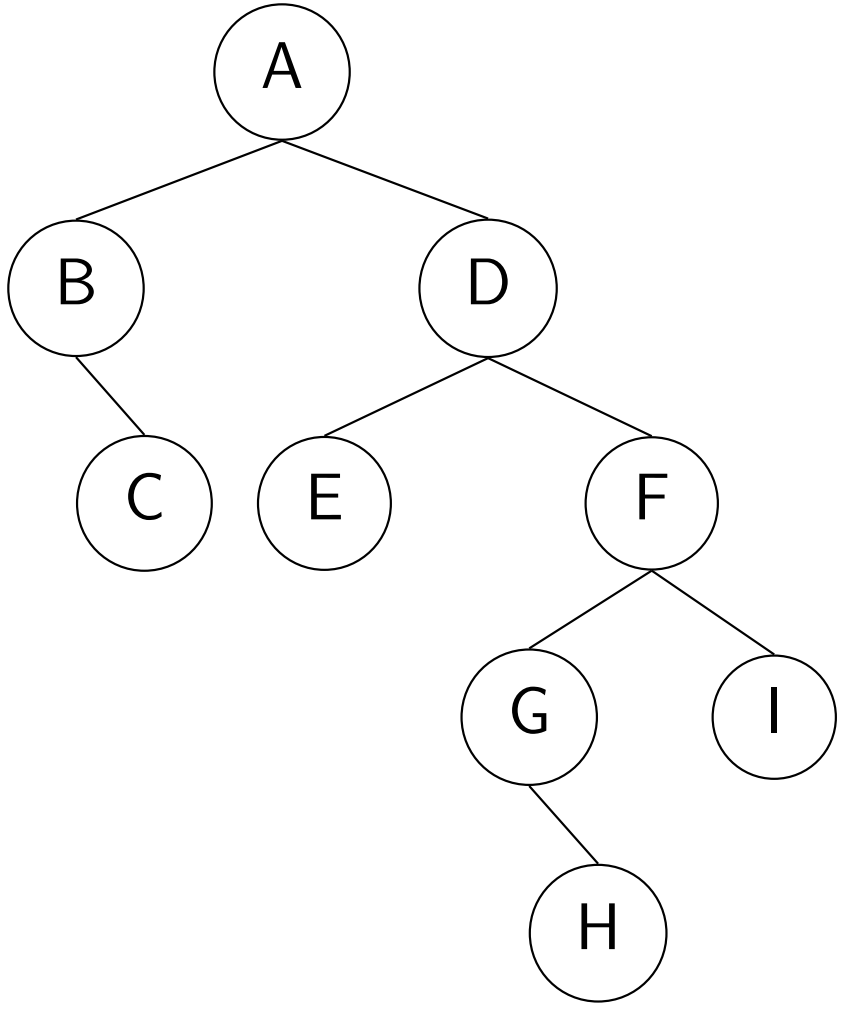


输出：B C A

中序的非递归算法（例子）

栈：

-
-
- D

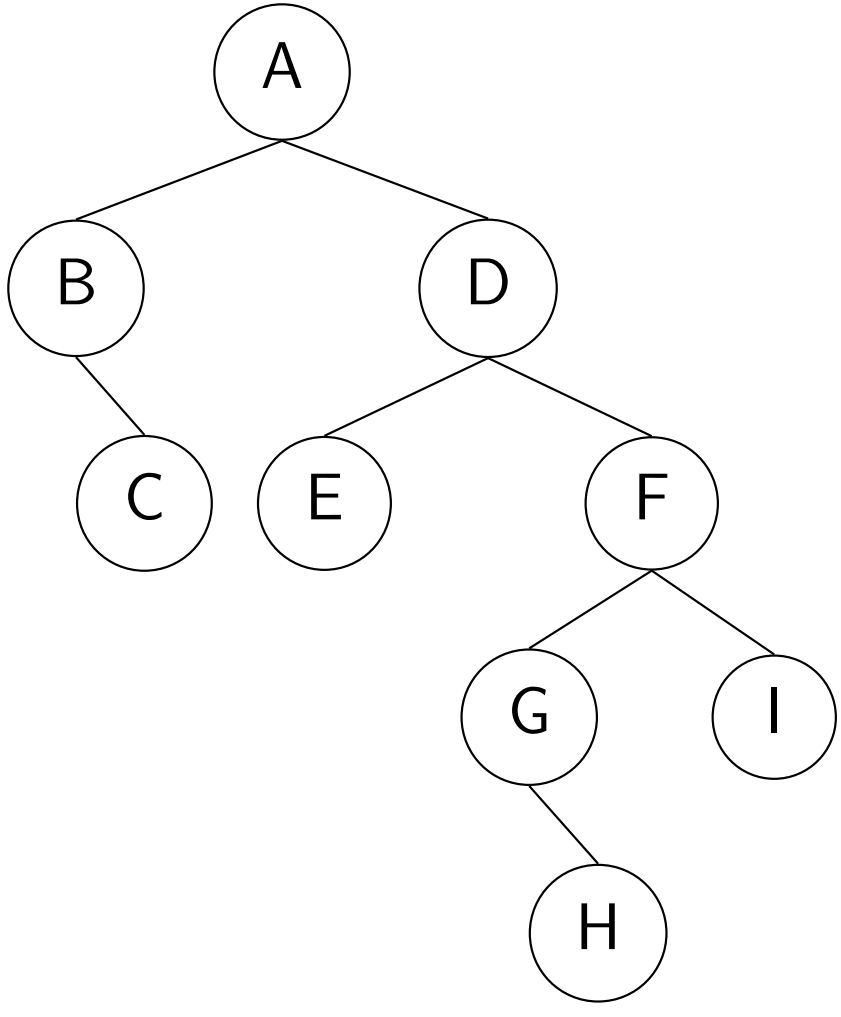


输出：B C A

中序的非递归算法（例子）

栈：

-
- E
- D

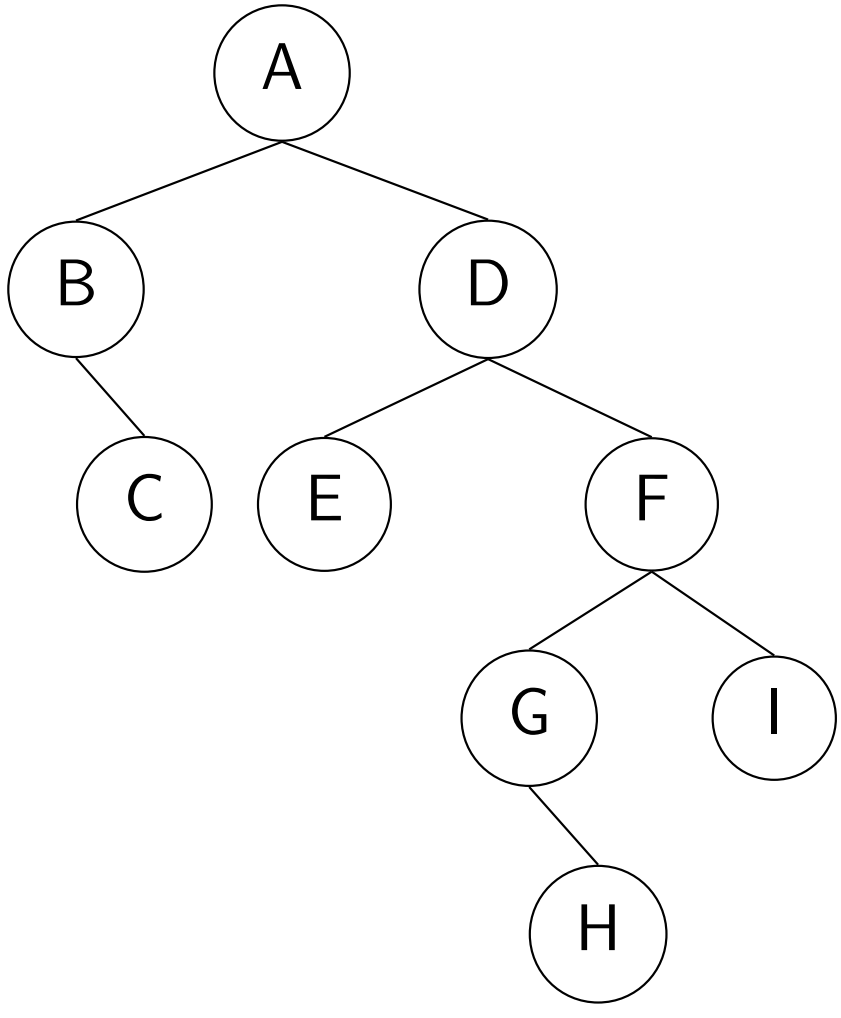


输出：B C A

中序的非递归算法（例子）

栈：

-
-
- D

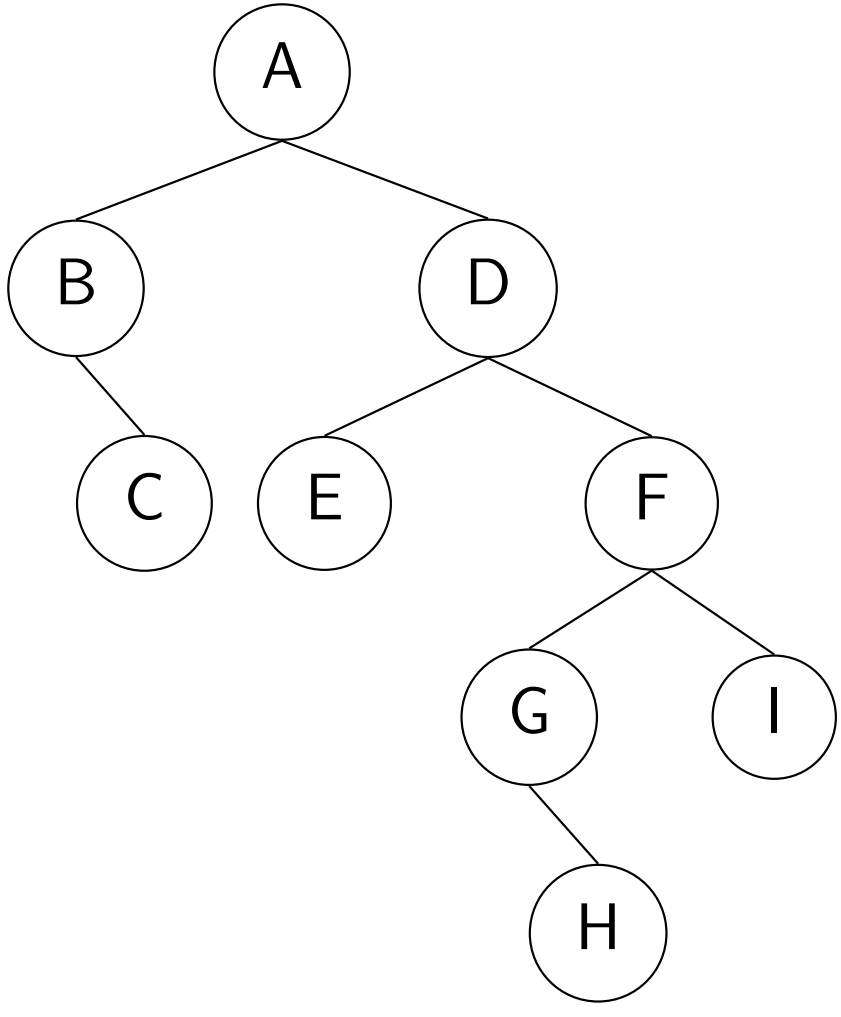


输出：B C A E

中序的非递归算法（例子）

栈：

-
-
-

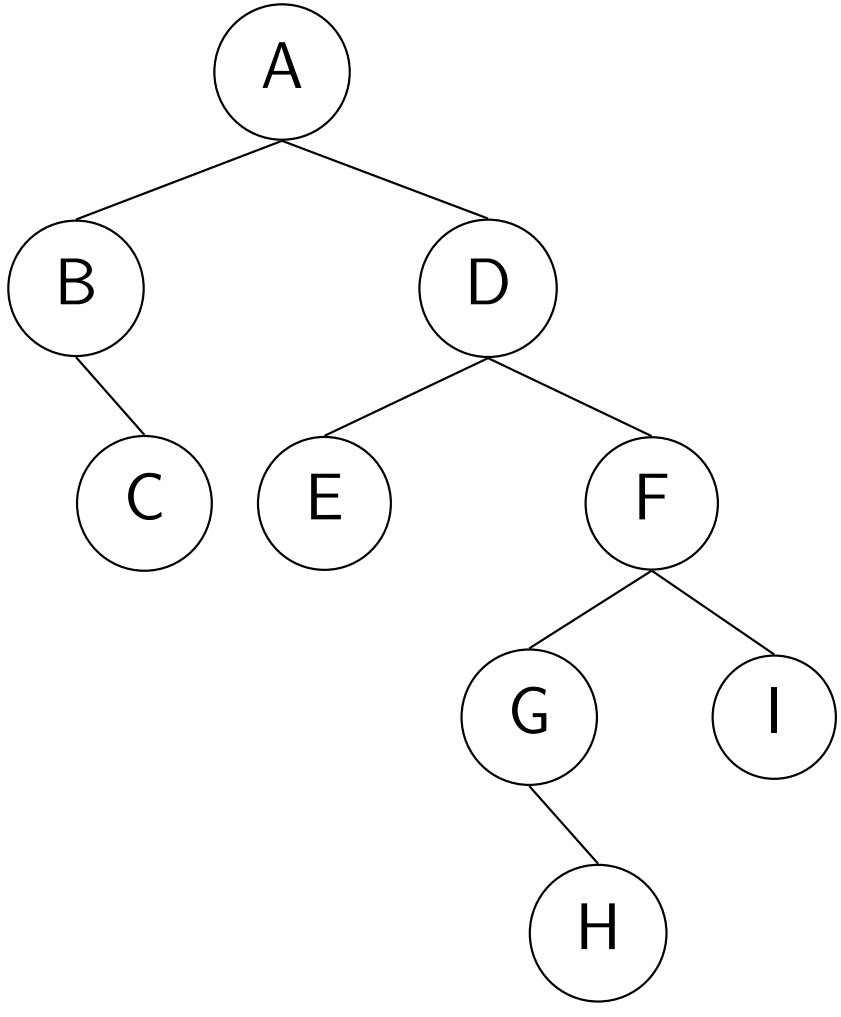


输出：B C A E D

中序的非递归算法（例子）

栈：

-
-
- F

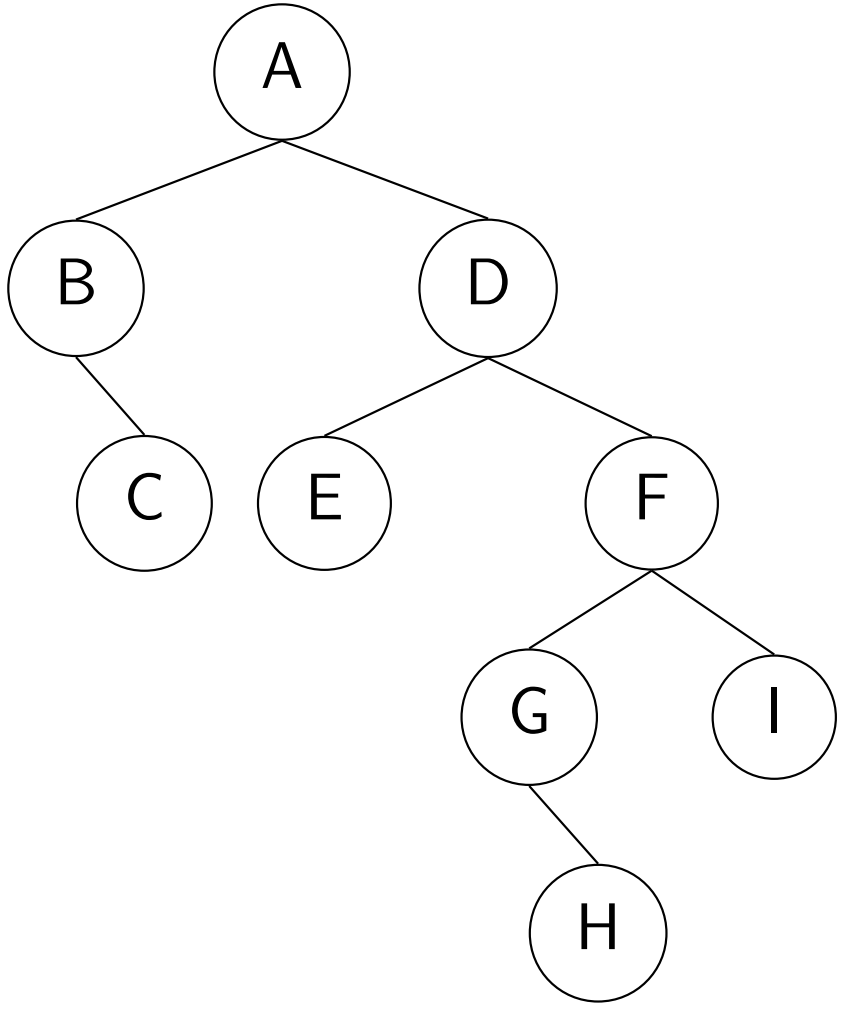


输出：B C A E D

中序的非递归算法（例子）

栈：

-
- G
- F

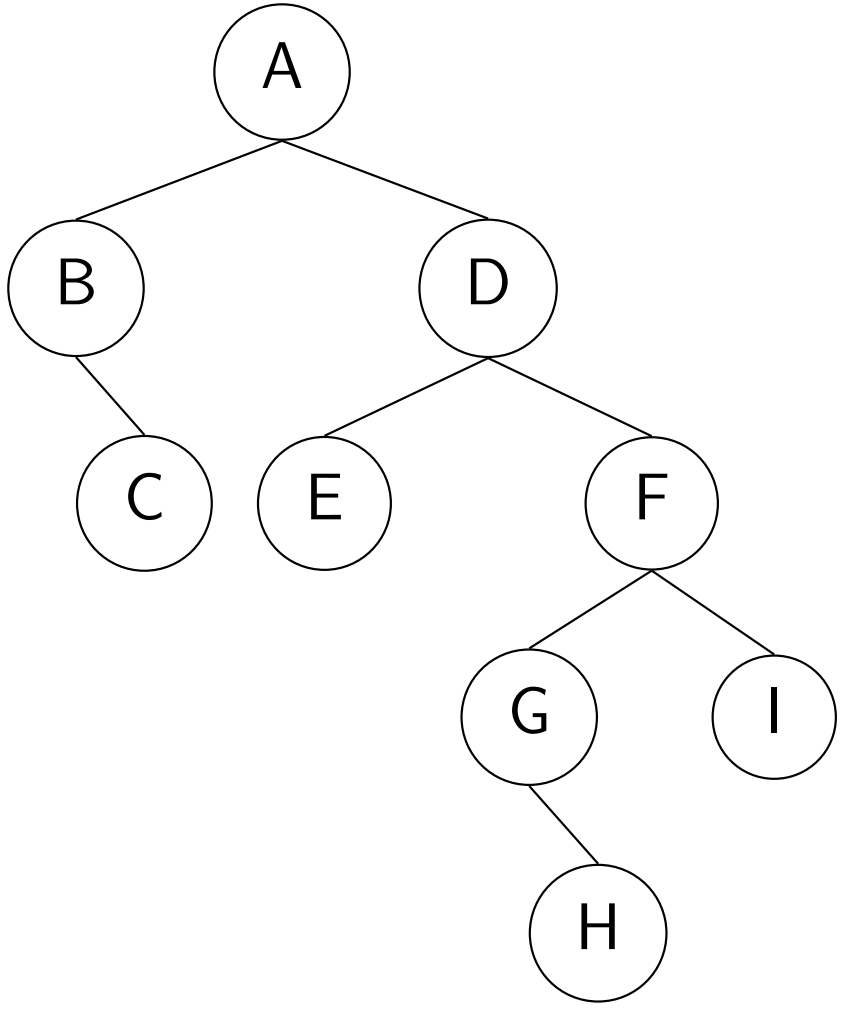


输出：B C A E D

中序的非递归算法（例子）

栈：

-
-
- F

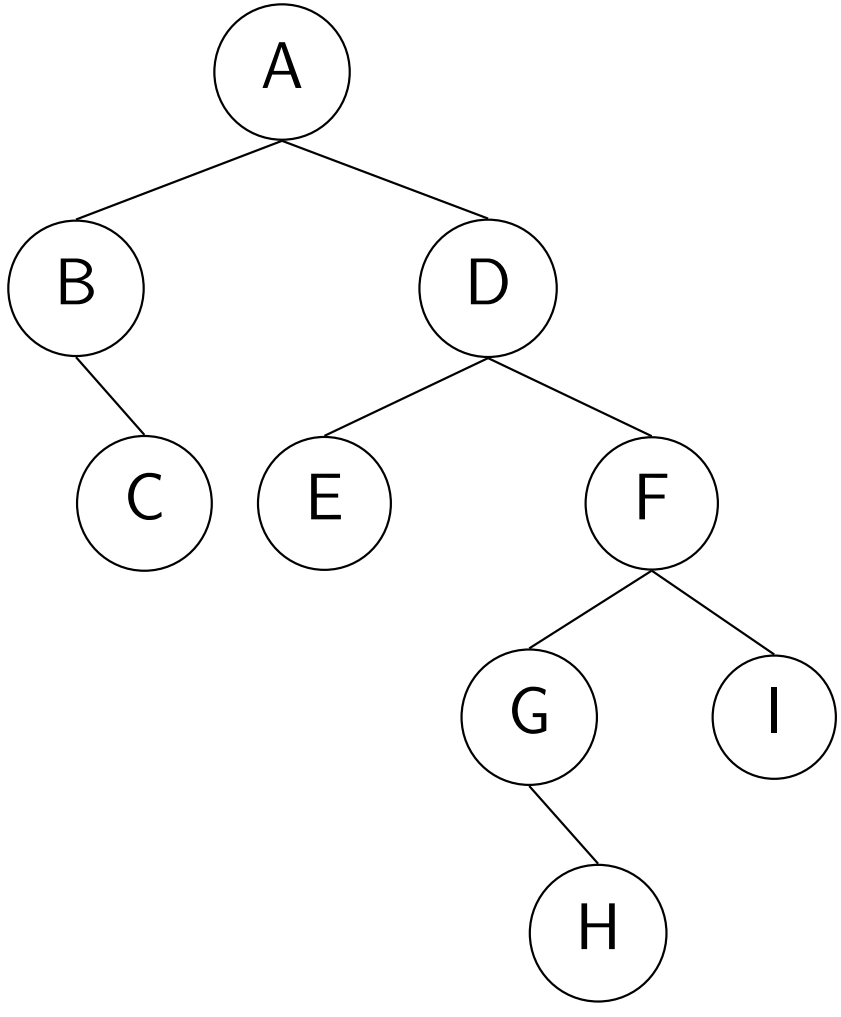


输出：B C A E D G

中序的非递归算法（例子）

栈：

-
- H
- F

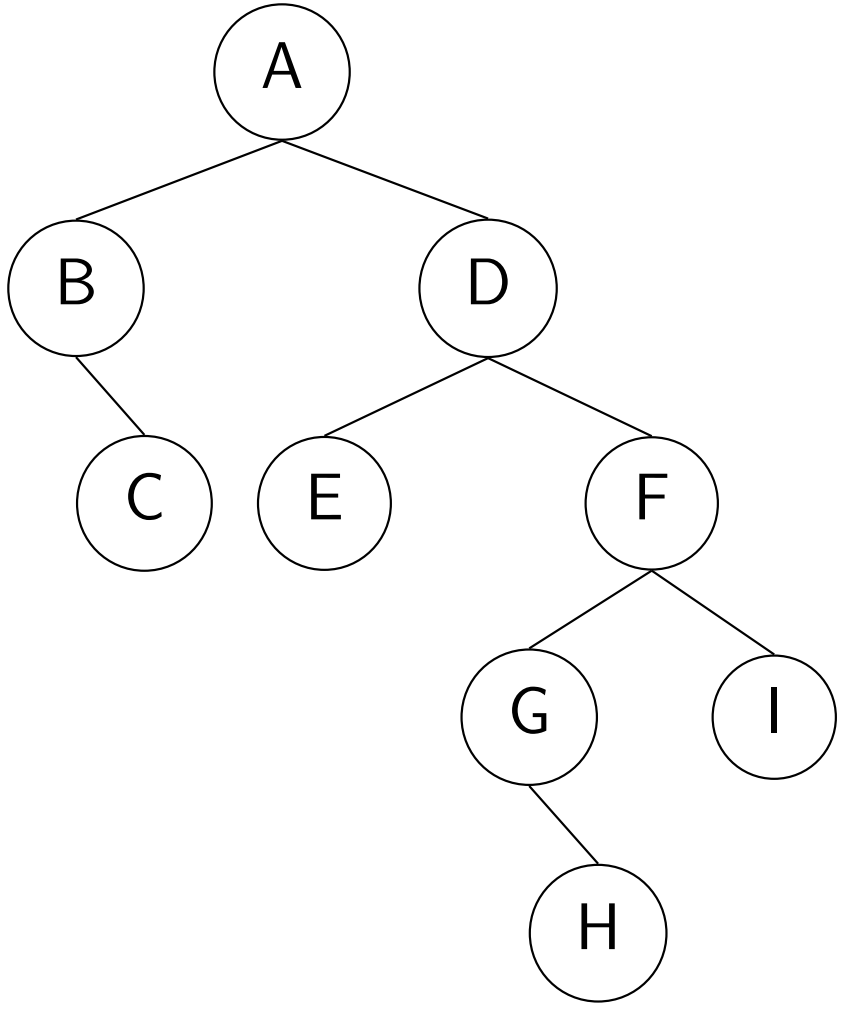


输出：B C A E D G

中序的非递归算法（例子）

栈：

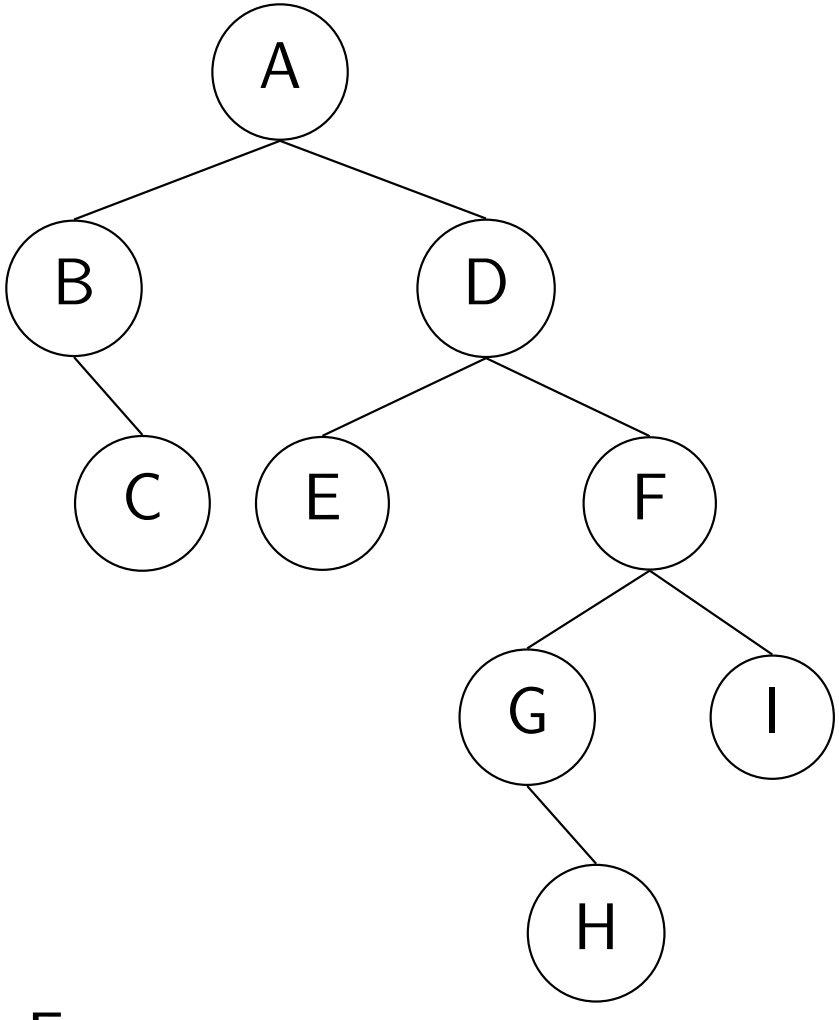
-
-
- F



输出：B C A E D G H

中序的非递归算法（例子）

栈：

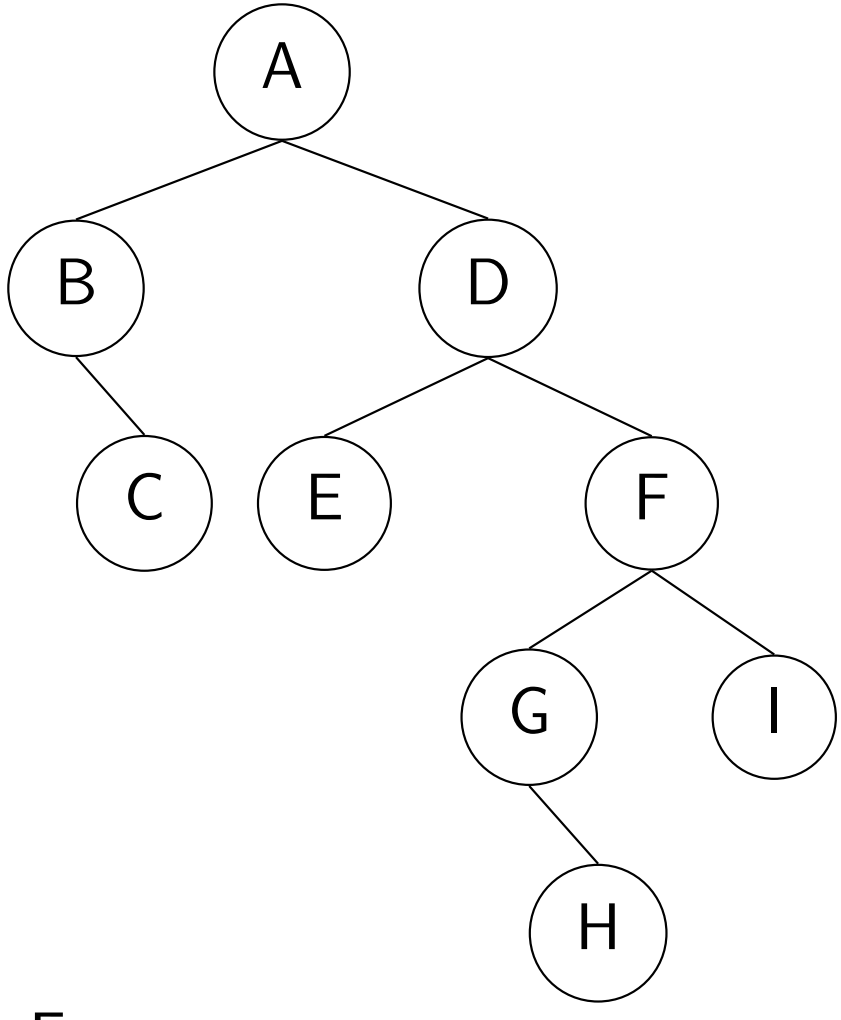


输出：B C A E D G H F

中序的非递归算法（例子）

栈：

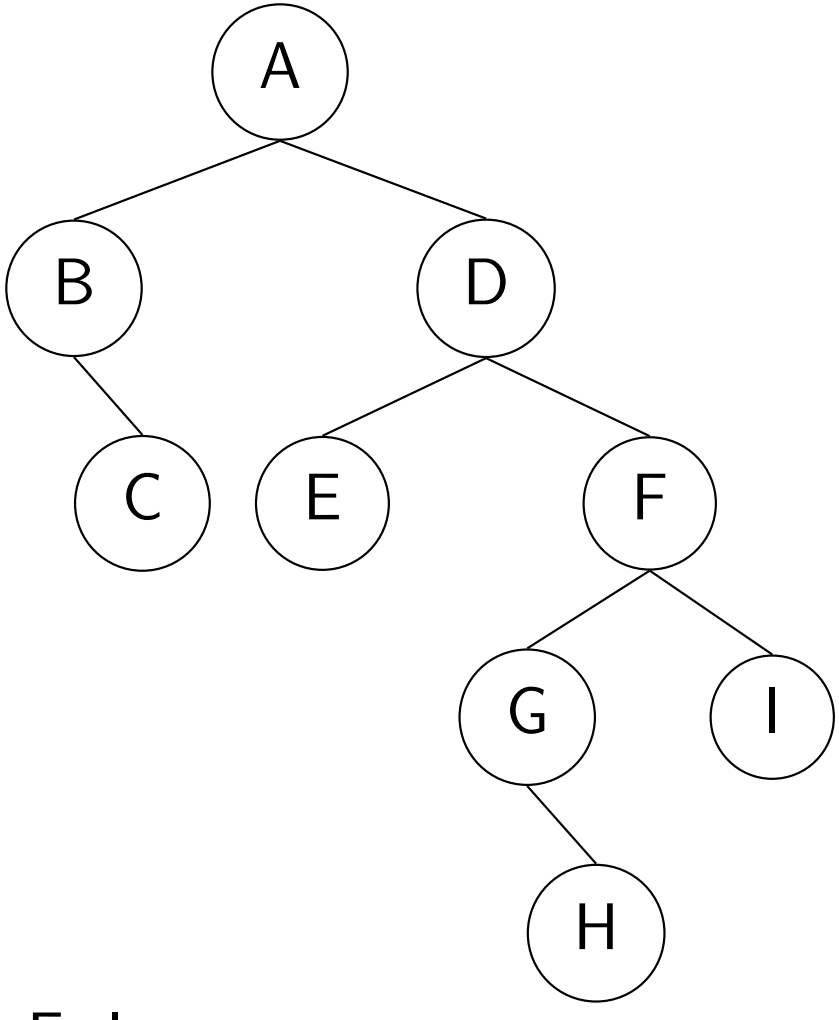
-
-
- I



输出：B C A E D G H F

中序的非递归算法（例子）

栈：



输出：B C A E D G H F I

后序的非递归算法

- 后序是左、右、根的顺序。
- 关键问题：如何跑完左子树后知道右子树的结点，又如何从右子树跑到根结点？

后序的非递归算法

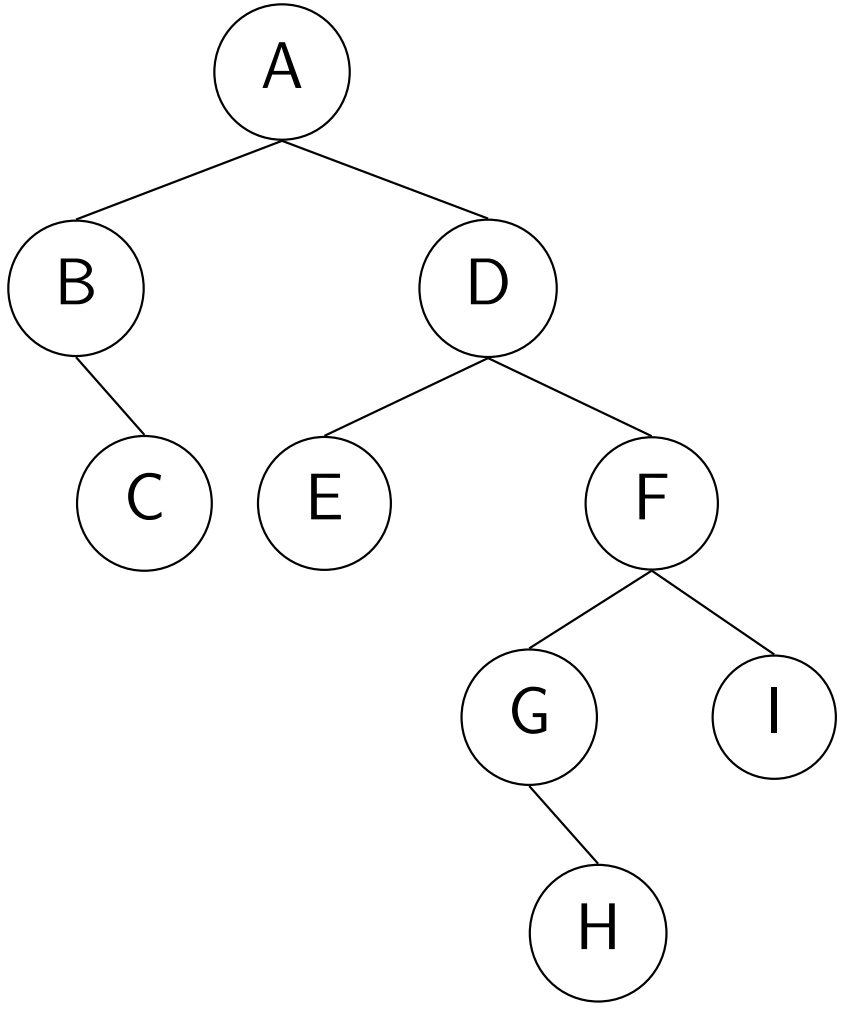
- 后序是左、右、根的顺序。
- 关键问题：如何跑完左子树后知道右子树的结点，又如何从右子树跑到根结点？
- 栈！加指标
- 在这个例子里我们就是把所有遇到的根结点都存到栈里。不过第一次存的时候加个一的标记，第二次入栈时加一个二的标记。

后序的非递归算法（例子）

栈：

-
-
-
-
- A（一）

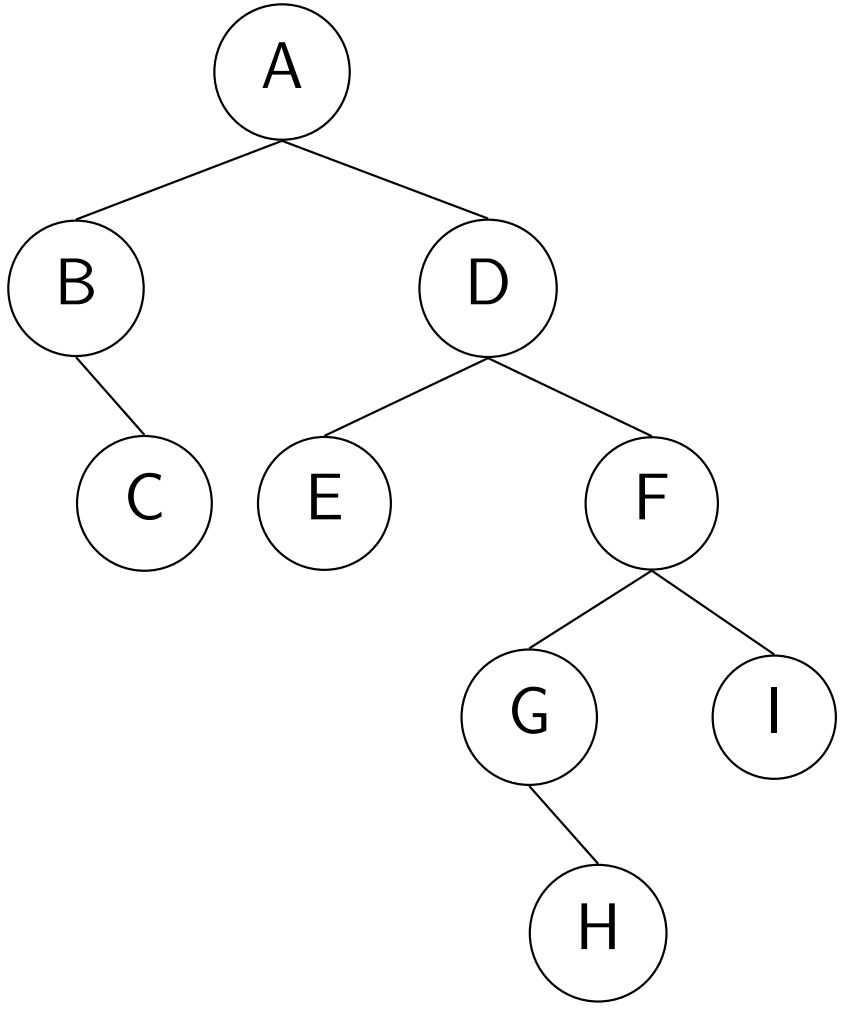
输出：



后序的非递归算法（例子）

栈：

-
-
-
- B（一）
- A（一）



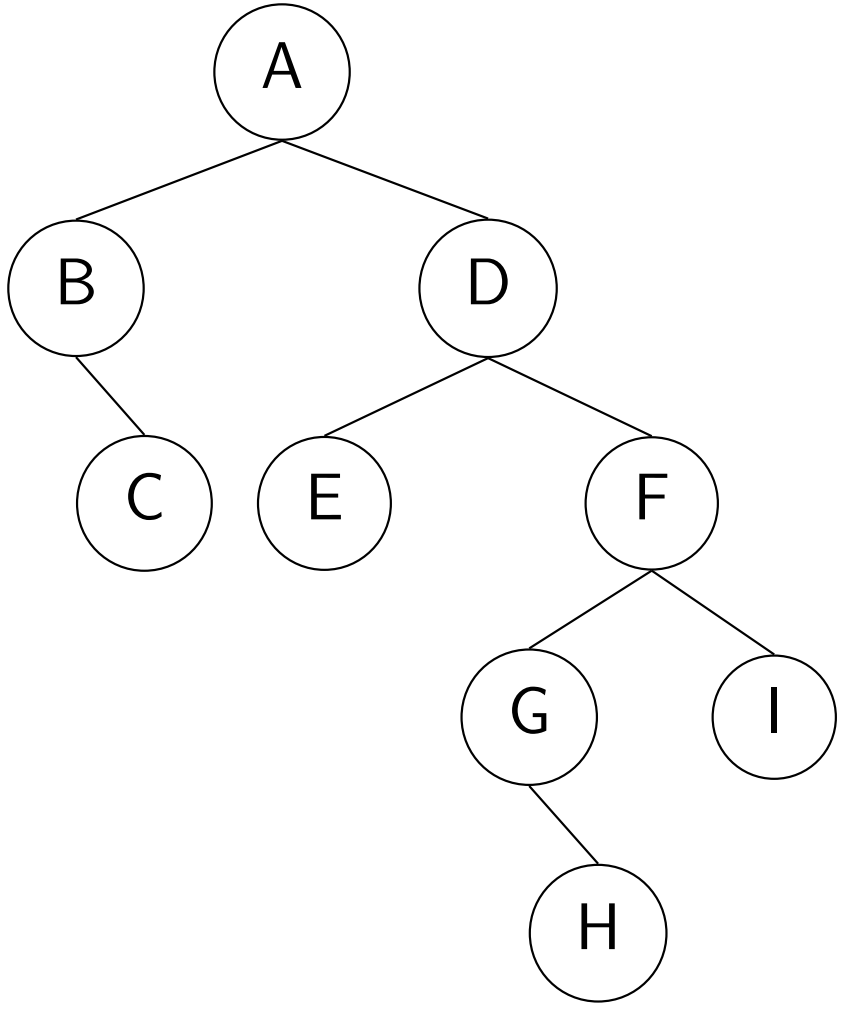
输出：

后序的非递归算法（例子）

栈：

-
-
-
- B（二）
- A（一）

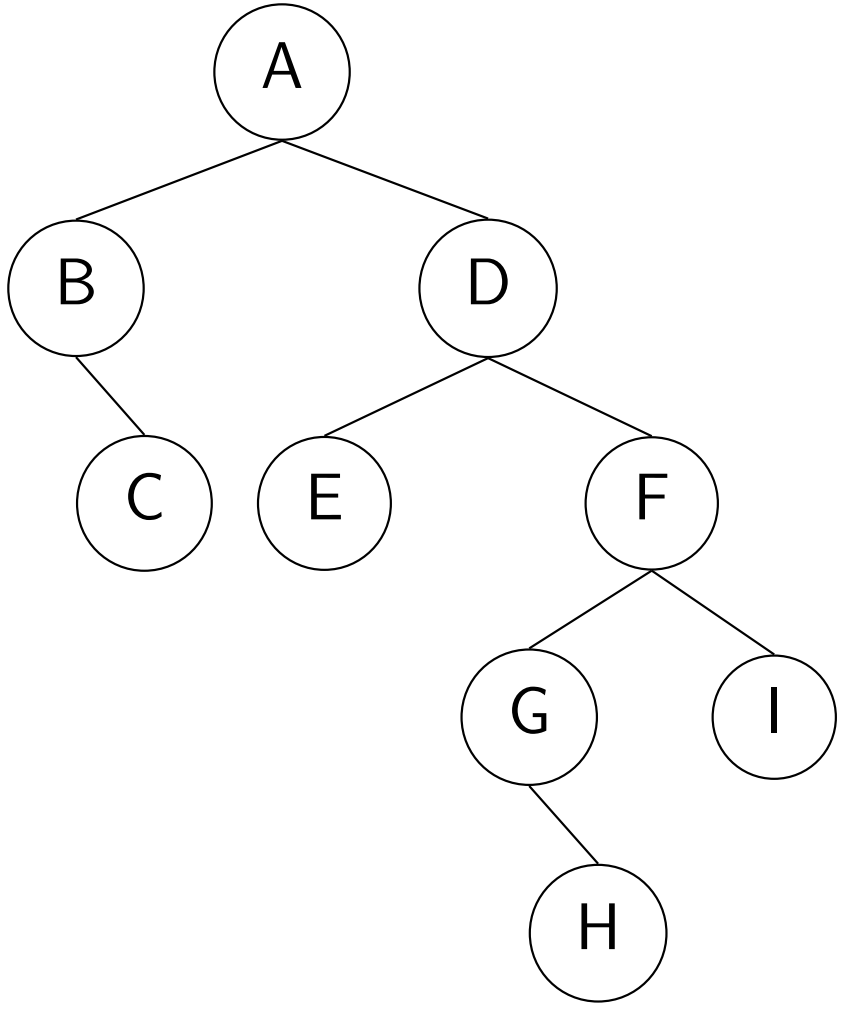
输出：



后序的非递归算法（例子）

栈：

-
-
- C（一）
- B（二）
- A（一）

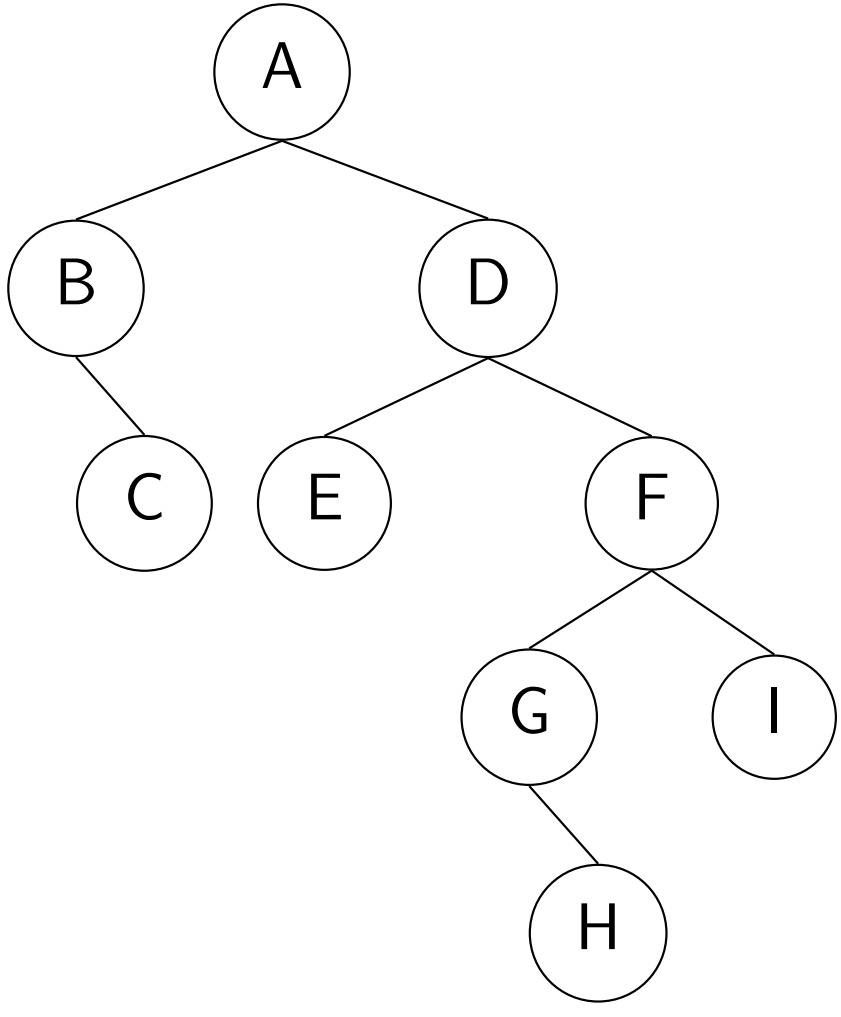


输出：

后序的非递归算法（例子）

栈：

-
-
- C（二）
- B（二）
- A（一）

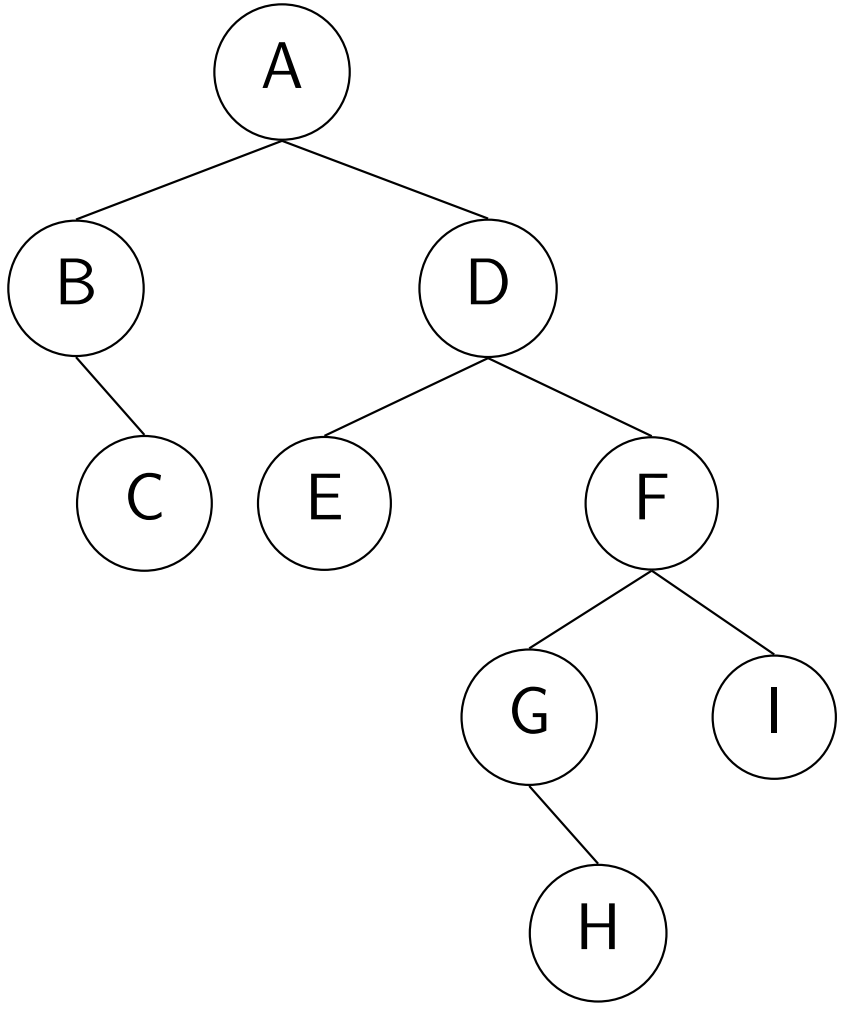


输出：

后序的非递归算法（例子）

栈：

-
-
-
- B（二）
- A（一）



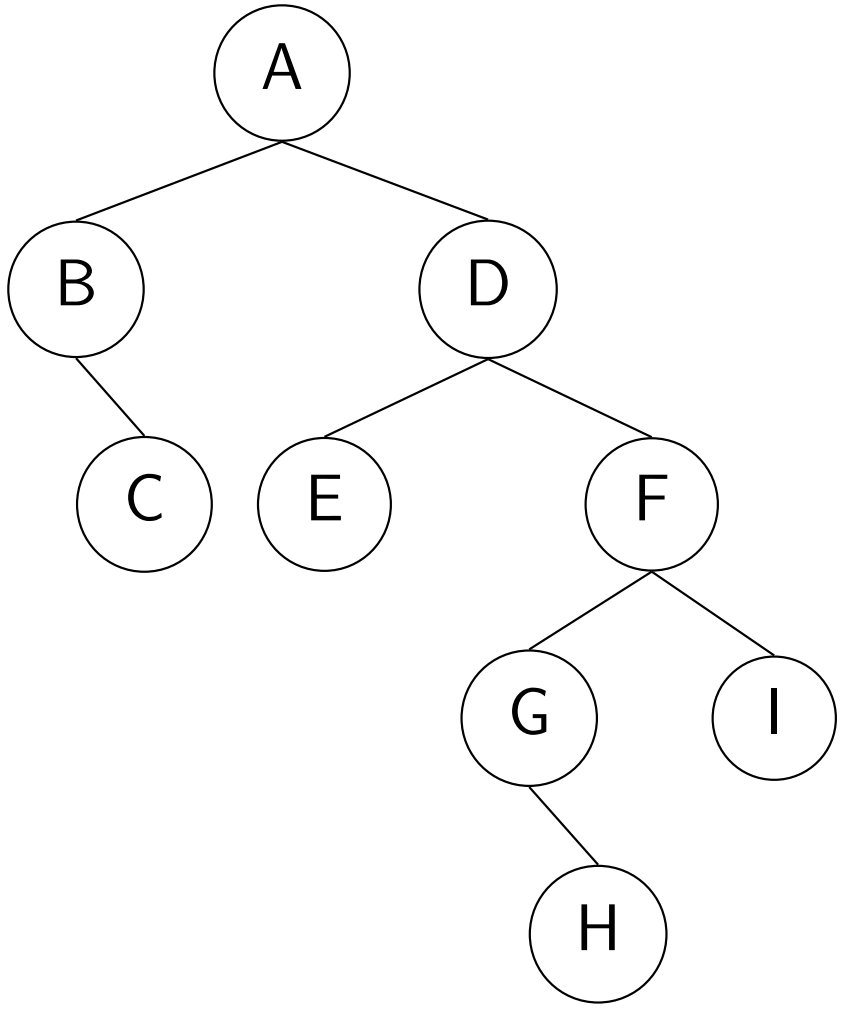
输出：C

后序的非递归算法（例子）

栈：

-
-
-
-
- A（一）

输出：C B

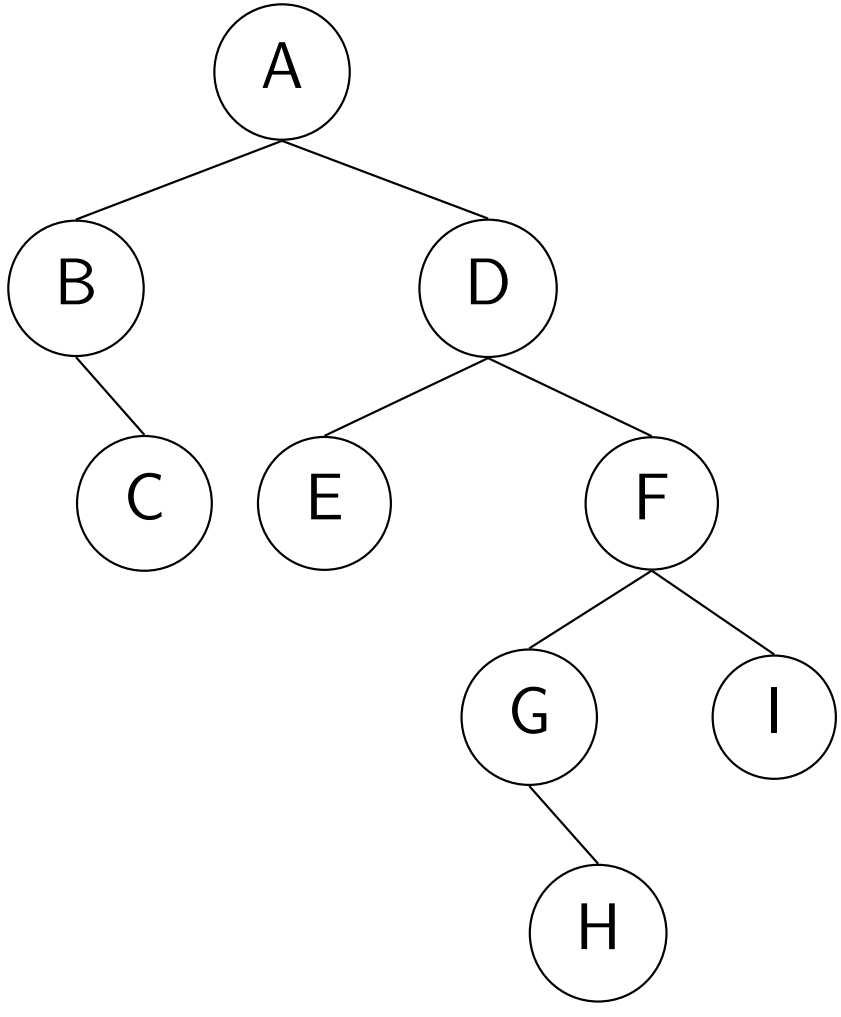


后序的非递归算法（例子）

栈：

-
-
-
-
- A（二）

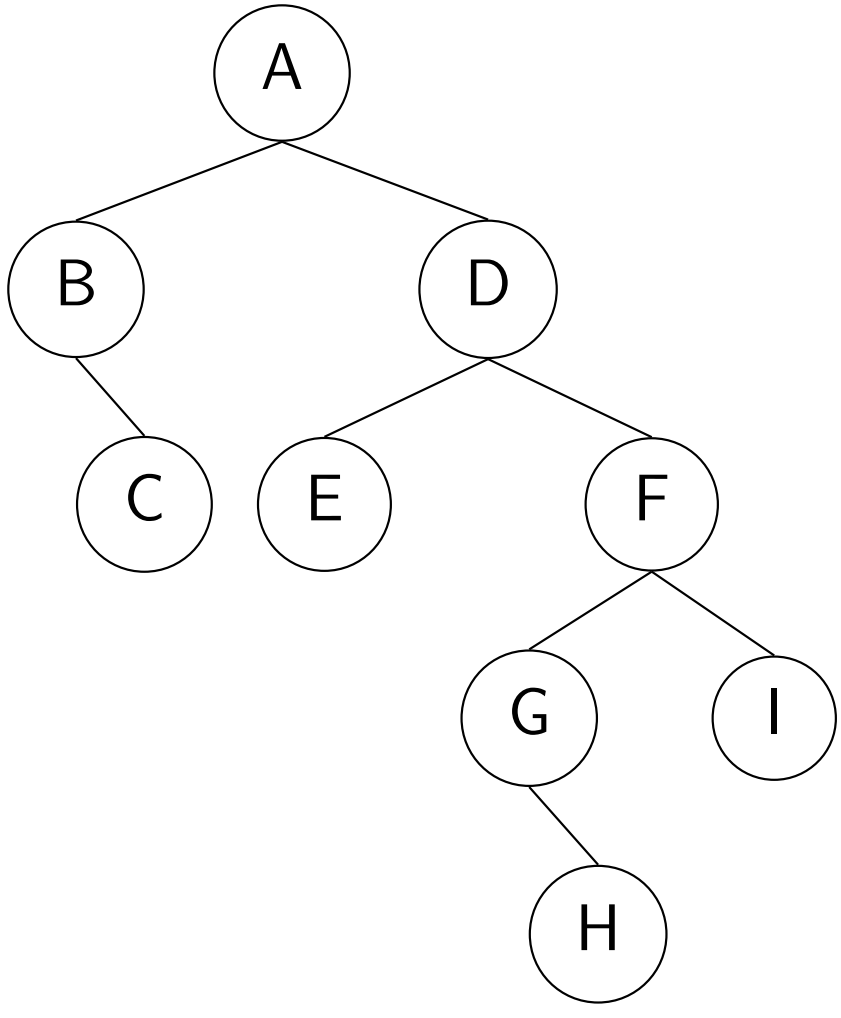
输出：C B



后序的非递归算法（例子）

栈：

-
-
-
- D（一）
- A（二）

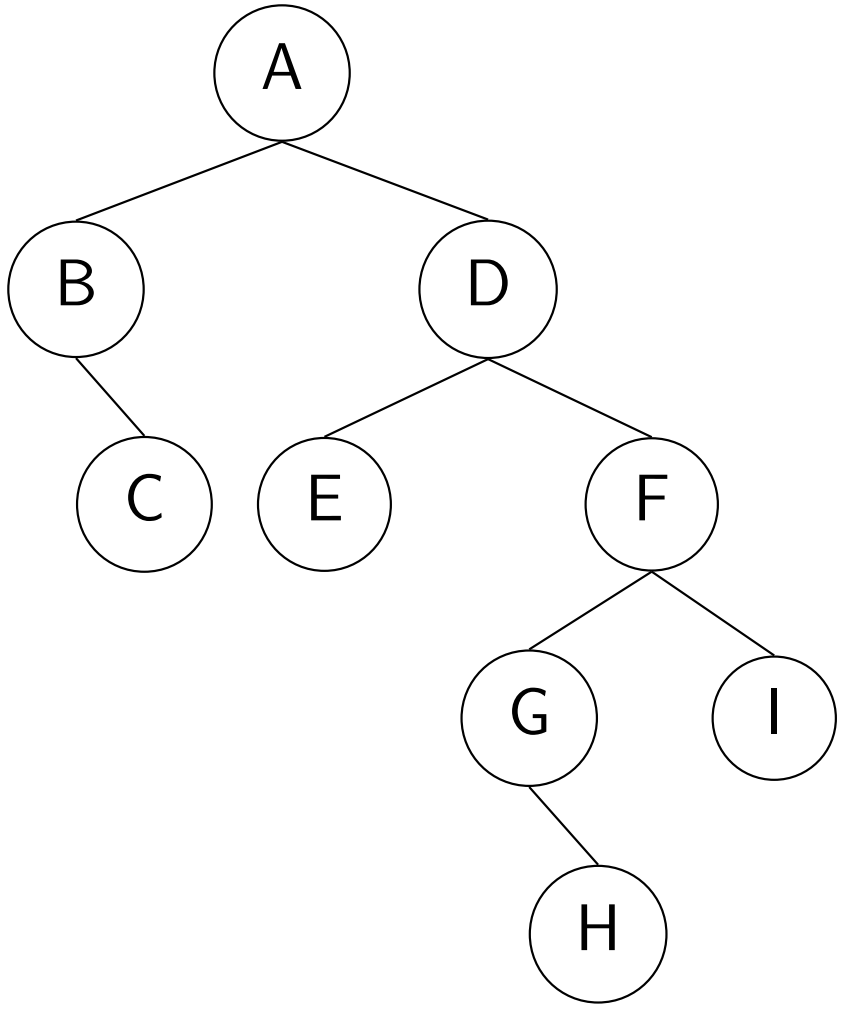


输出：C B

后序的非递归算法（例子）

栈：

-
-
- E（一）
- D（一）
- A（二）

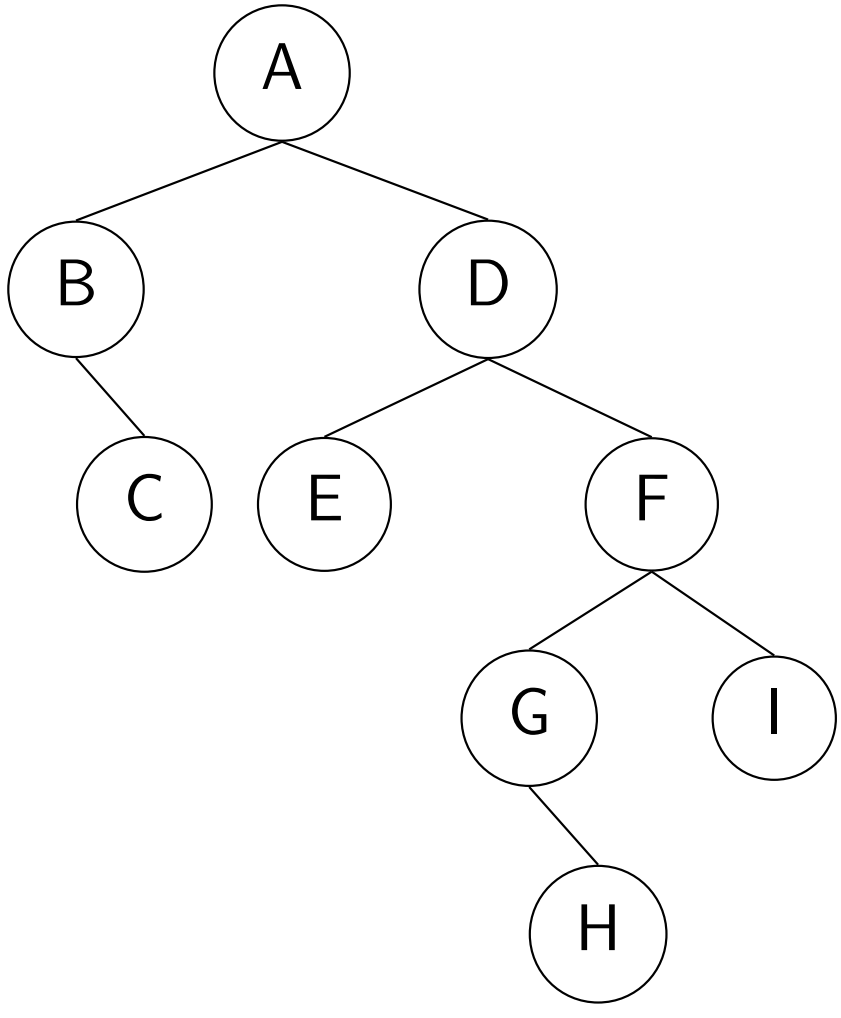


输出：C B

后序的非递归算法（例子）

栈：

-
-
- E（二）
- D（一）
- A（二）

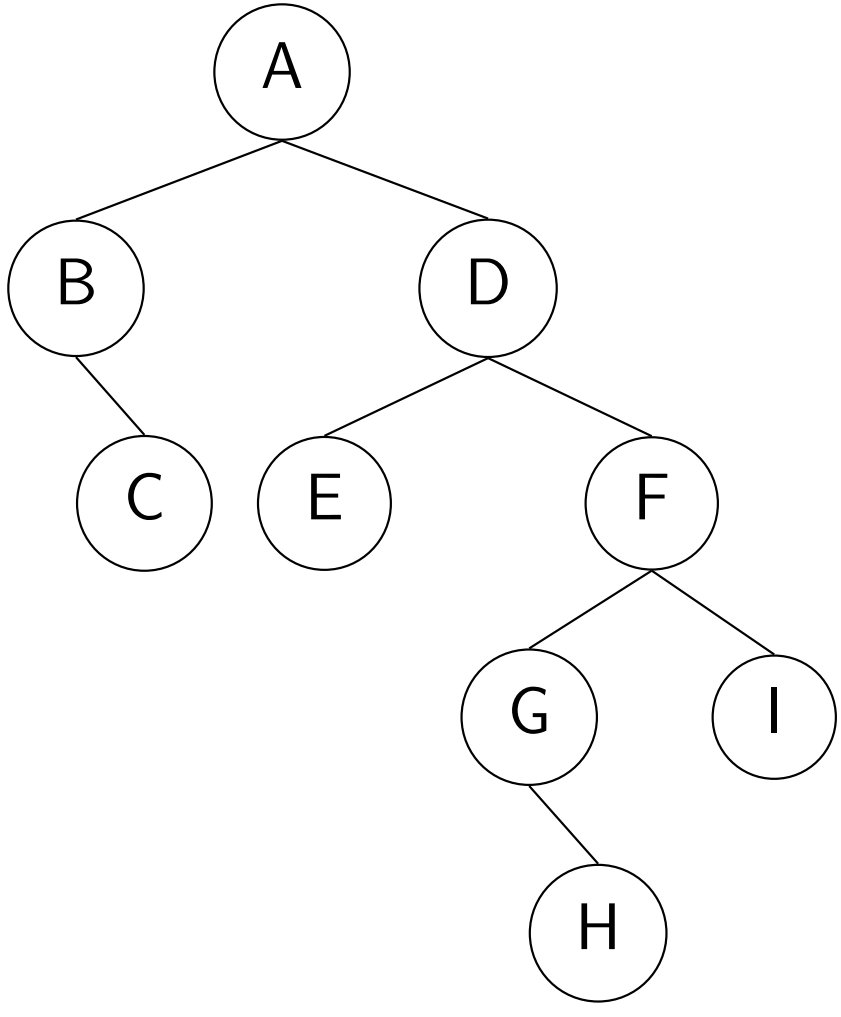


输出：C B

后序的非递归算法（例子）

栈：

-
-
-
- D（一）
- A（二）

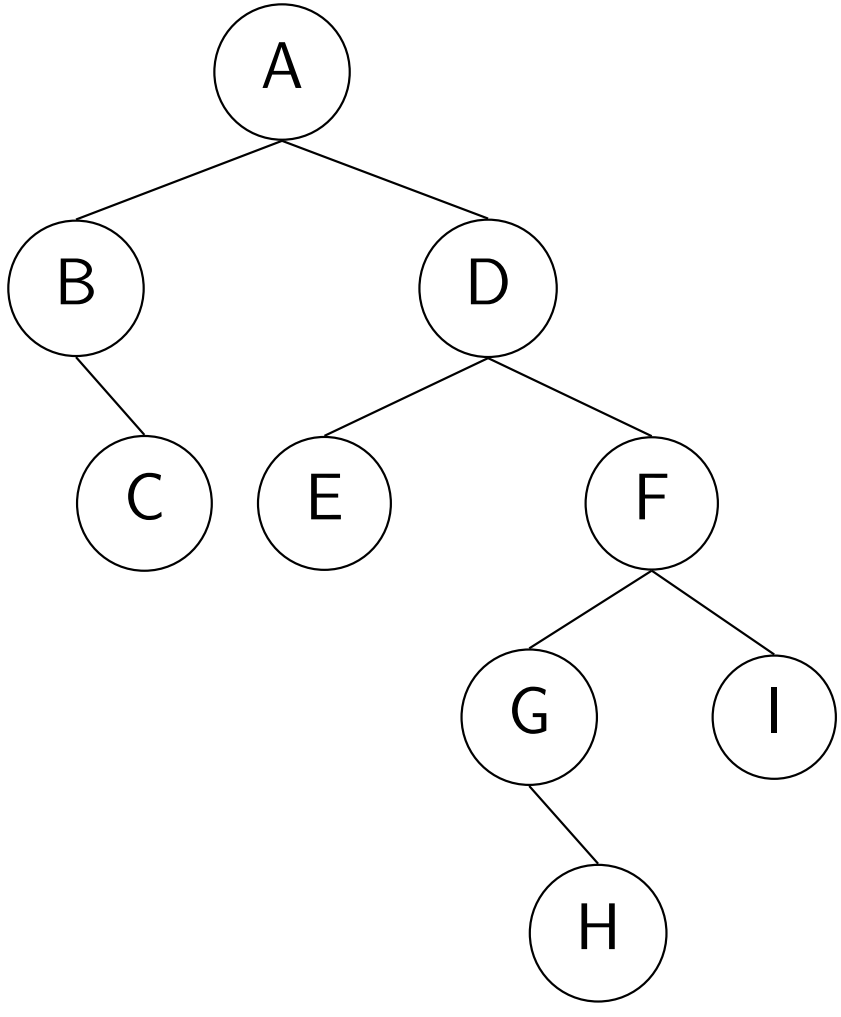


输出：C B E

后序的非递归算法（例子）

栈：

-
-
-
- D（二）
- A（二）

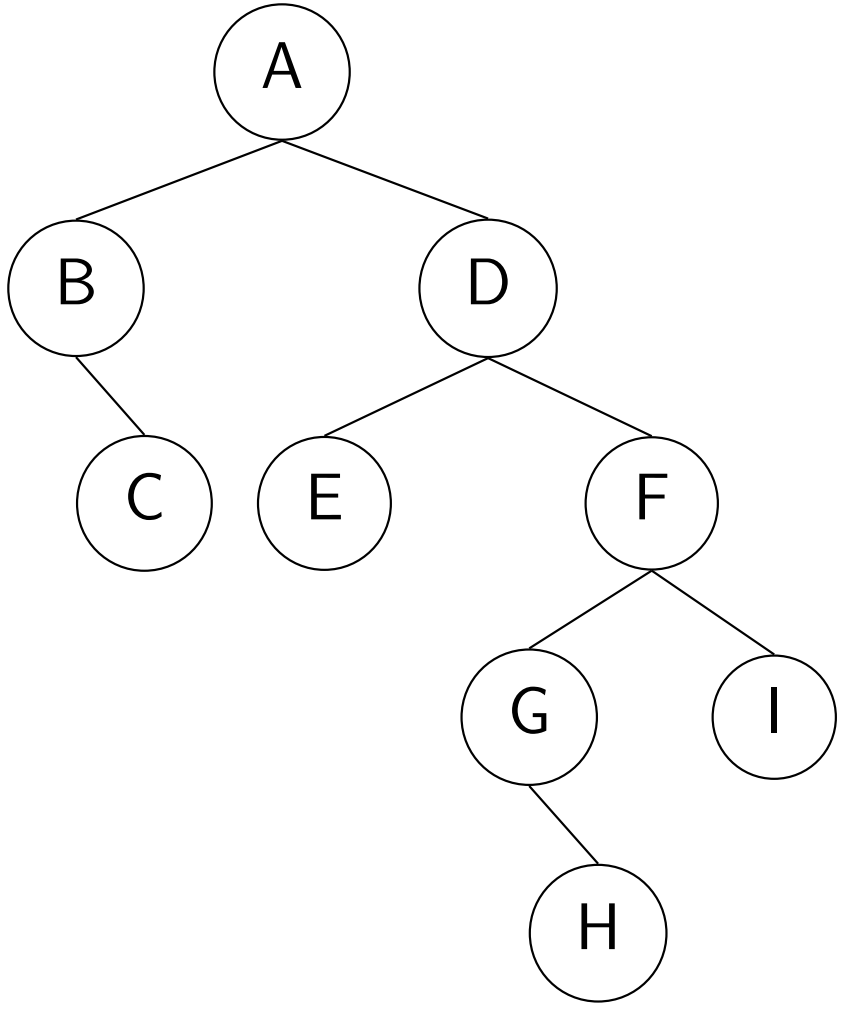


输出：C B E

后序的非递归算法（例子）

栈：

-
-
- F（一）
- D（二）
- A（二）

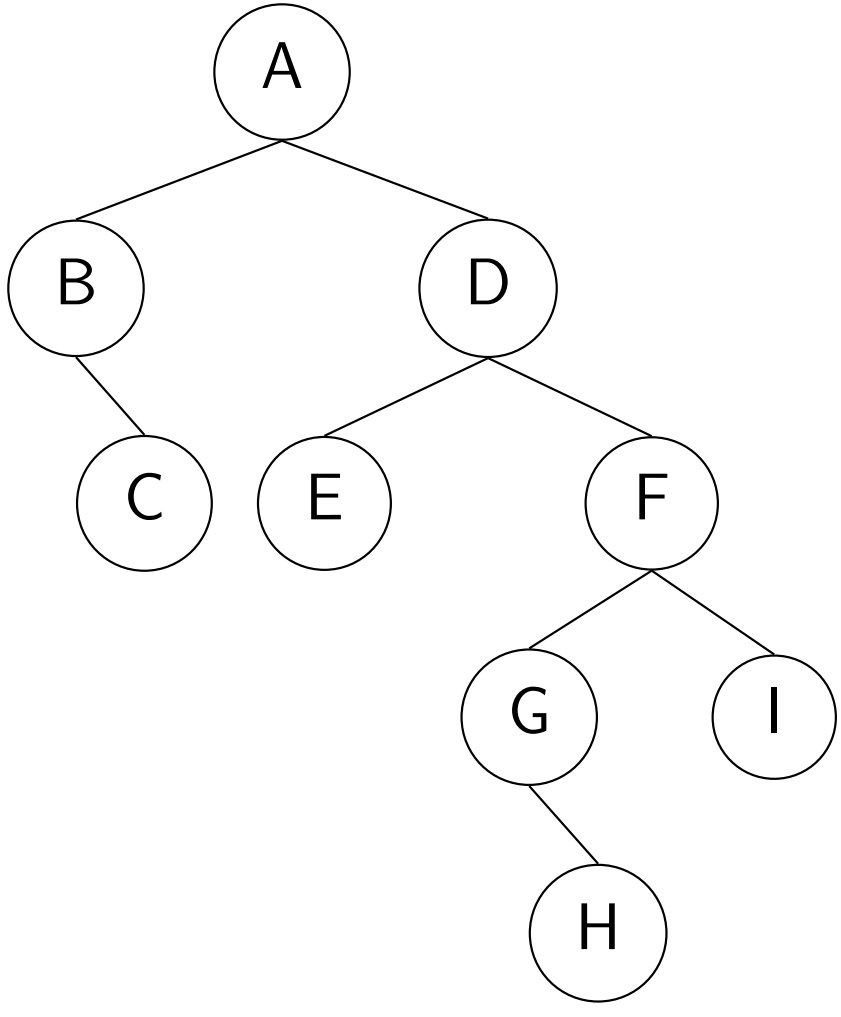


输出：C B E

后序的非递归算法（例子）

栈：

-
- G（一）
- F（一）
- D（二）
- A（二）

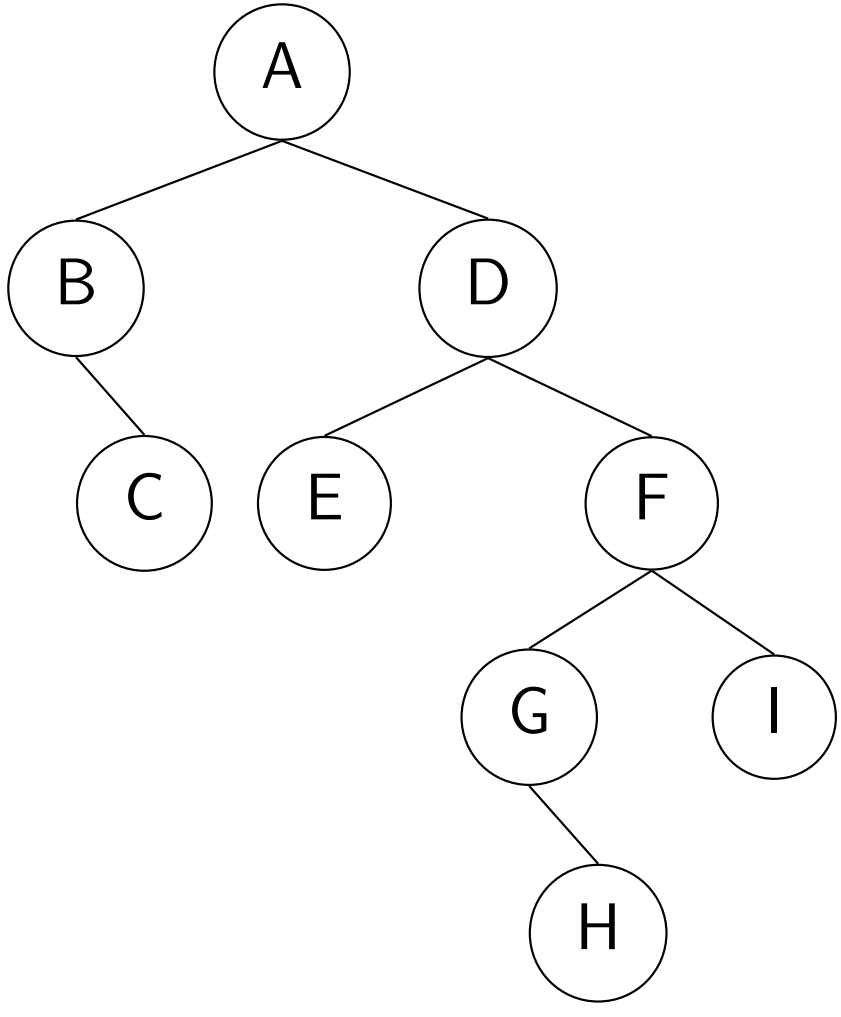


输出：C B E

后序的非递归算法（例子）

栈：

-
- G（二）
- F（一）
- D（二）
- A（二）

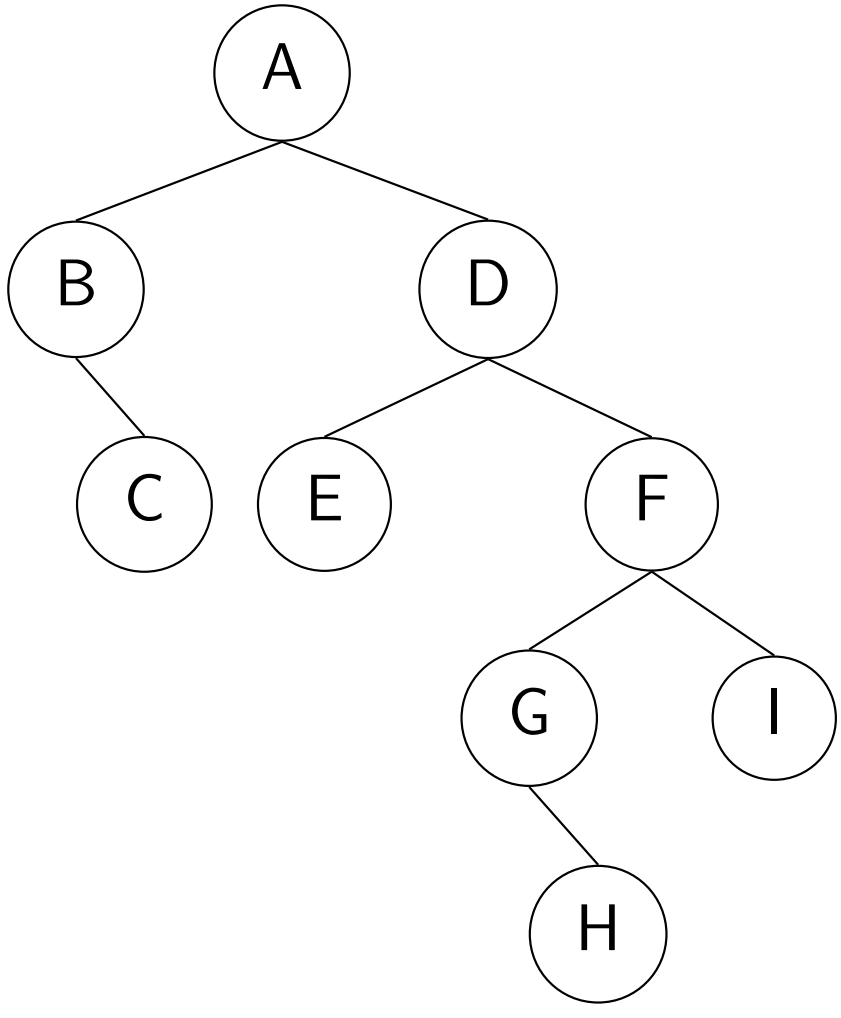


输出：C B E

后序的非递归算法（例子）

栈：

- H（一）
- G（二）
- F（一）
- D（二）
- A（二）

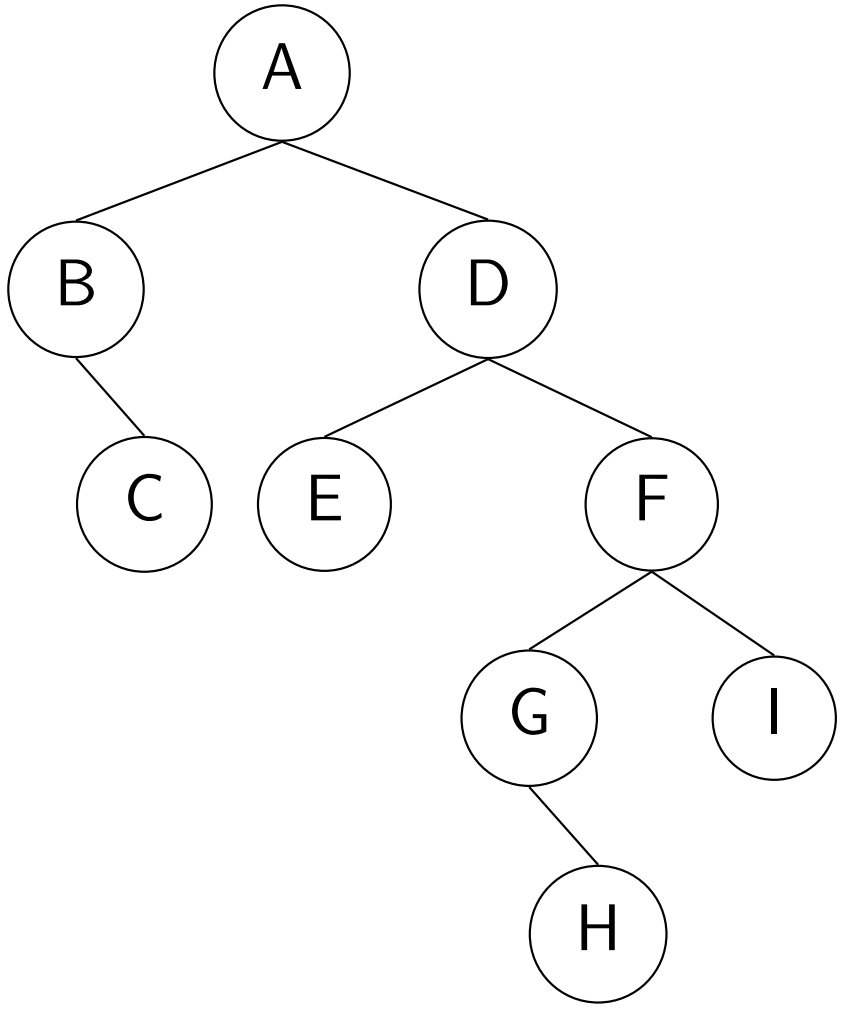


输出：C B E

后序的非递归算法（例子）

栈：

- H（二）
- G（二）
- F（一）
- D（二）
- A（二）

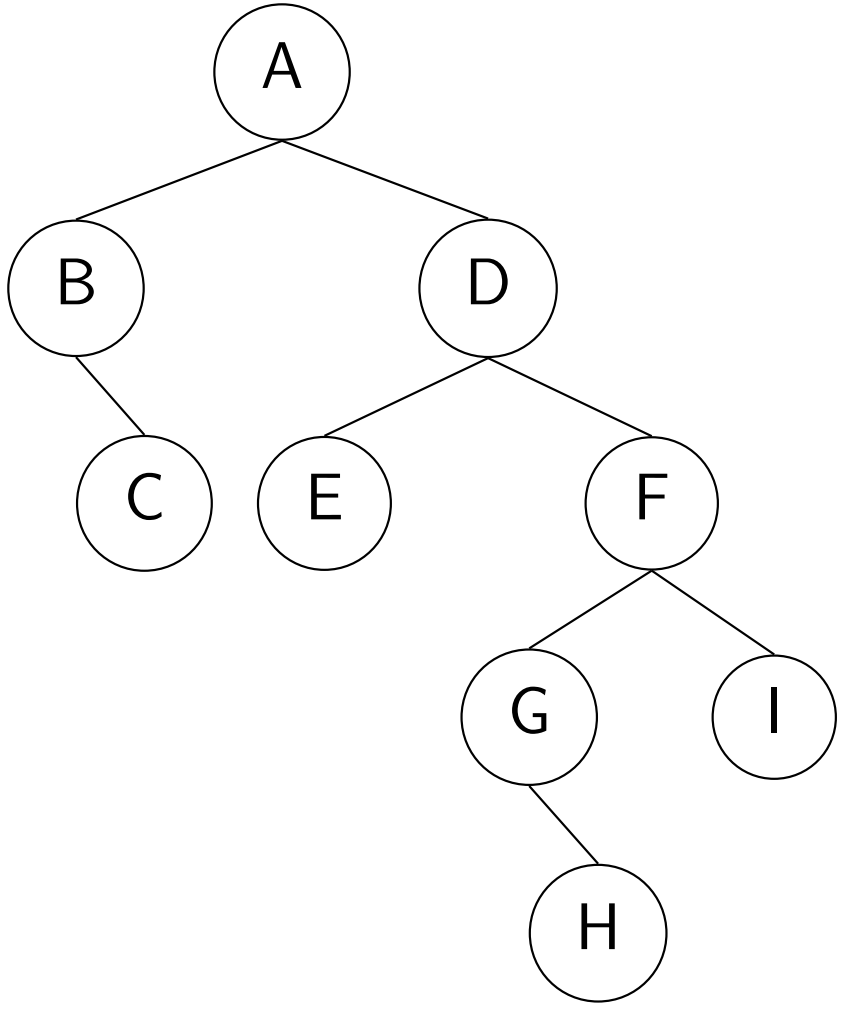


输出：C B E

后序的非递归算法（例子）

栈：

-
- G（二）
- F（一）
- D（二）
- A（二）

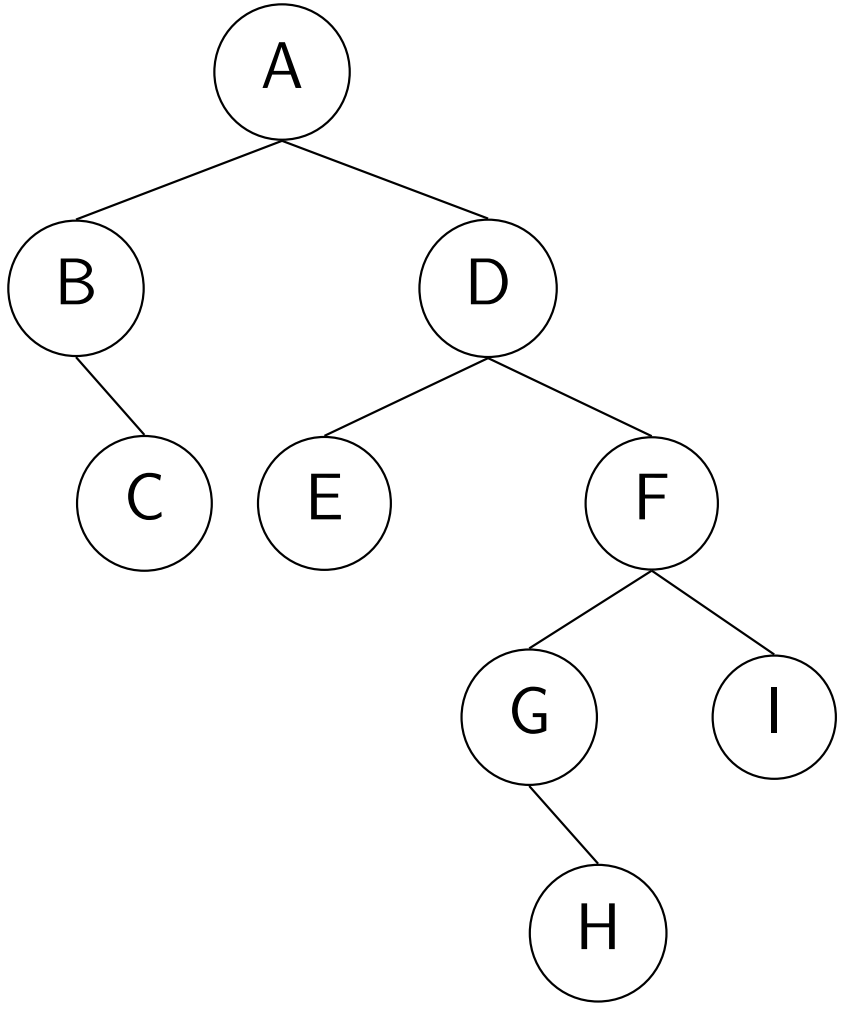


输出：C B E H

后序的非递归算法（例子）

栈：

-
-
- F（一）
- D（二）
- A（二）

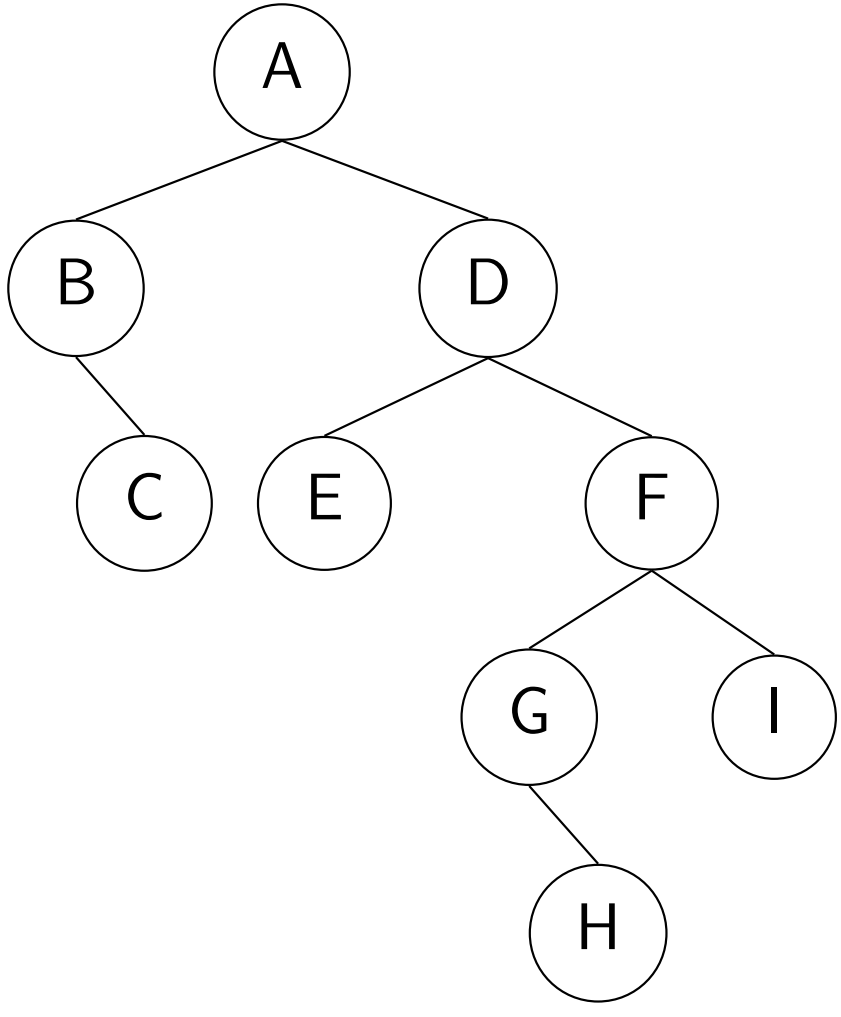


输出：C B E H G

后序的非递归算法（例子）

栈：

-
-
- F（二）
- D（二）
- A（二）

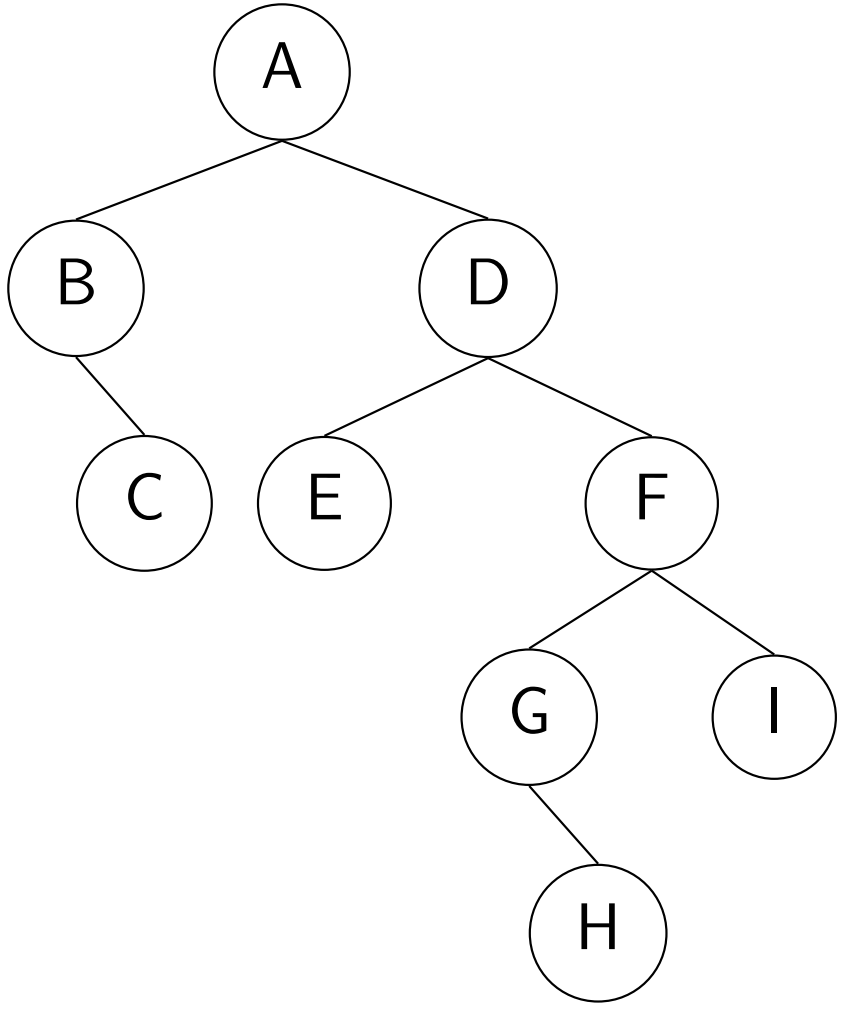


输出：C B E H G

后序的非递归算法（例子）

栈：

-
- I（一）
- F（二）
- D（二）
- A（二）

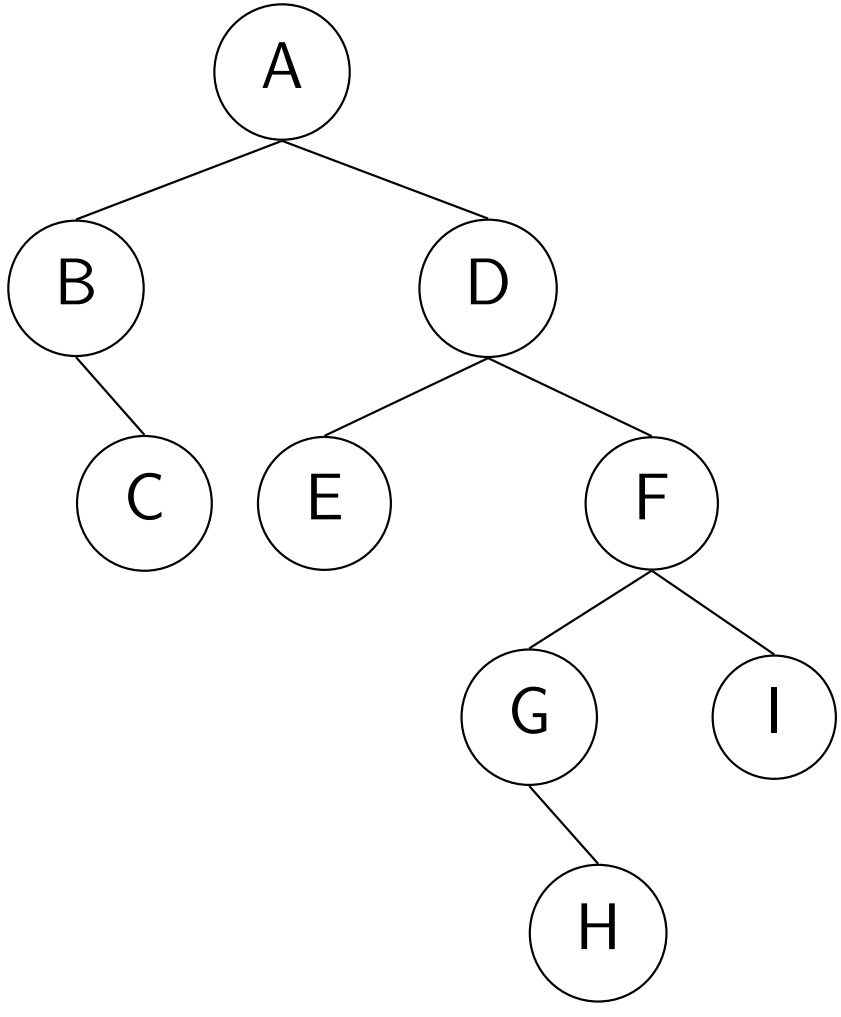


输出：C B E H G

后序的非递归算法（例子）

栈：

-
- I（二）
- F（二）
- D（二）
- A（二）

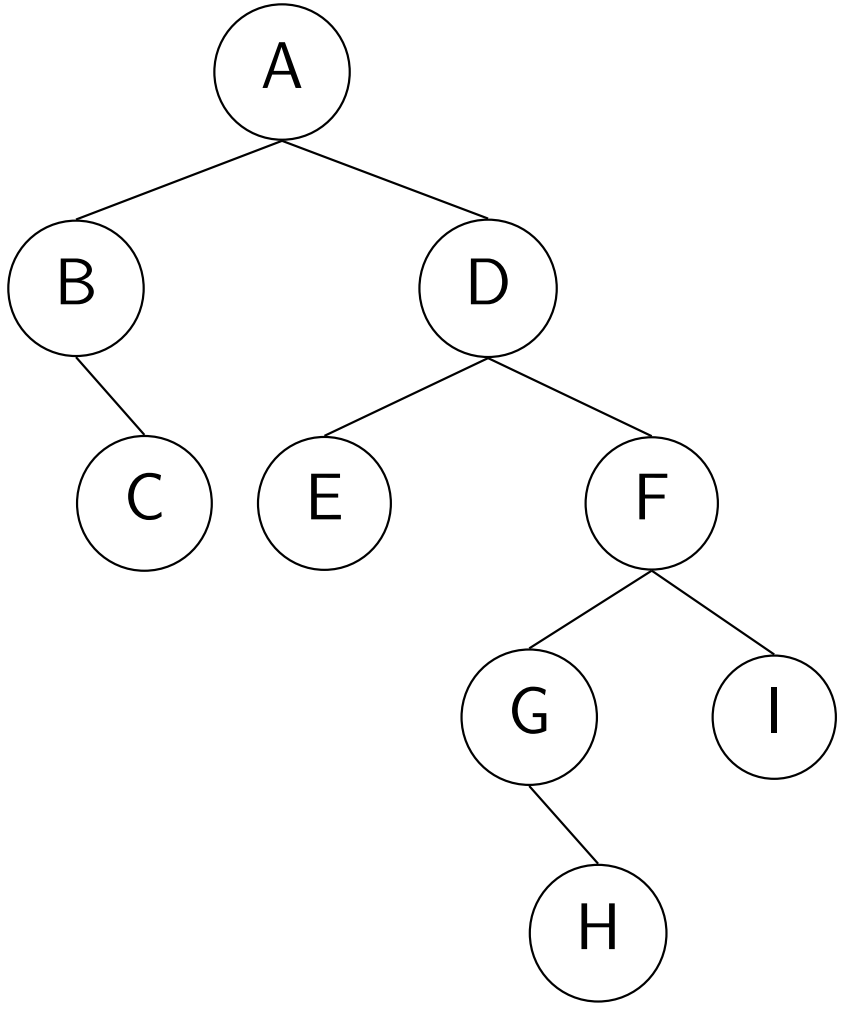


输出：C B E H G

后序的非递归算法（例子）

栈：

-
-
- F（二）
- D（二）
- A（二）

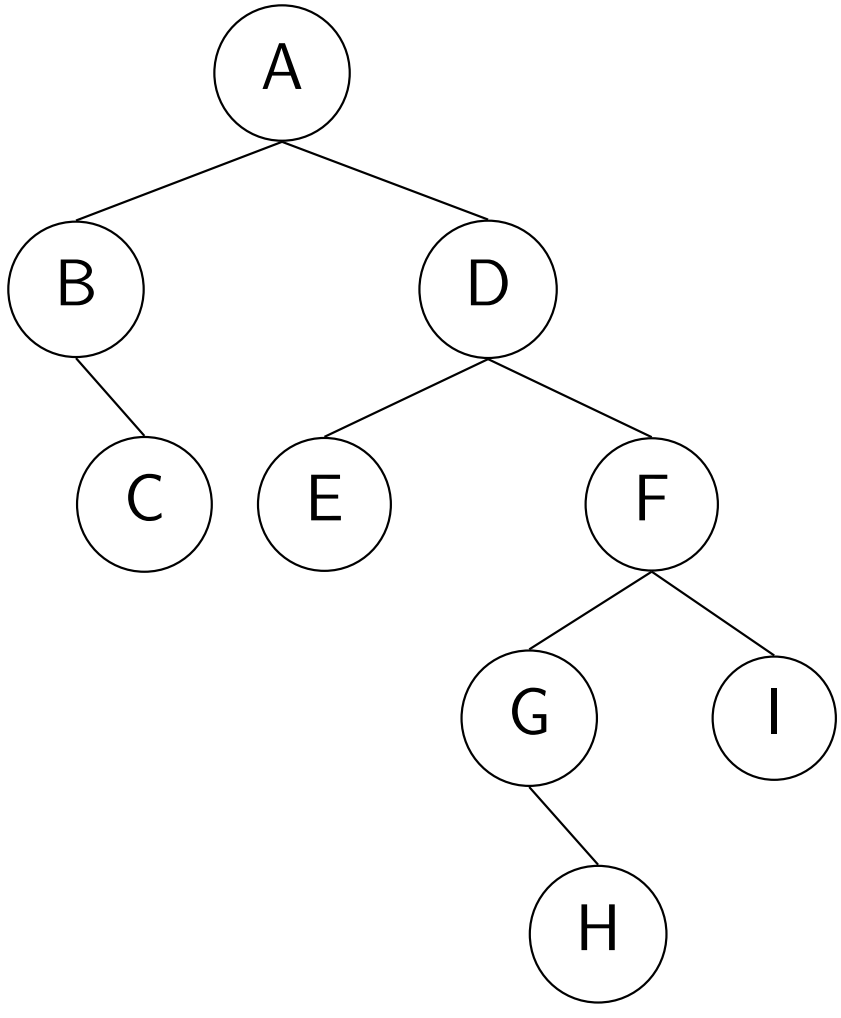


输出：C B E H G I

后序的非递归算法（例子）

栈：

-
-
-
- D（二）
- A（二）

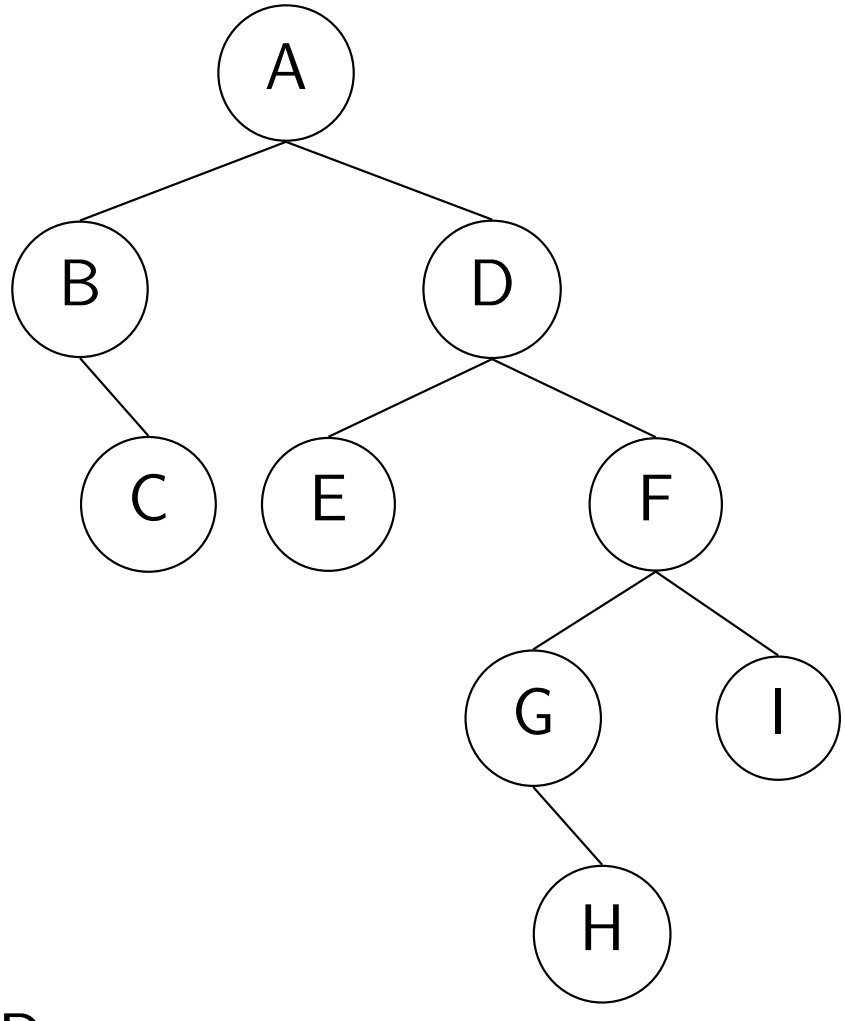


输出：C B E H G I F

后序的非递归算法（例子）

栈：

-
-
-
-
- A（二）

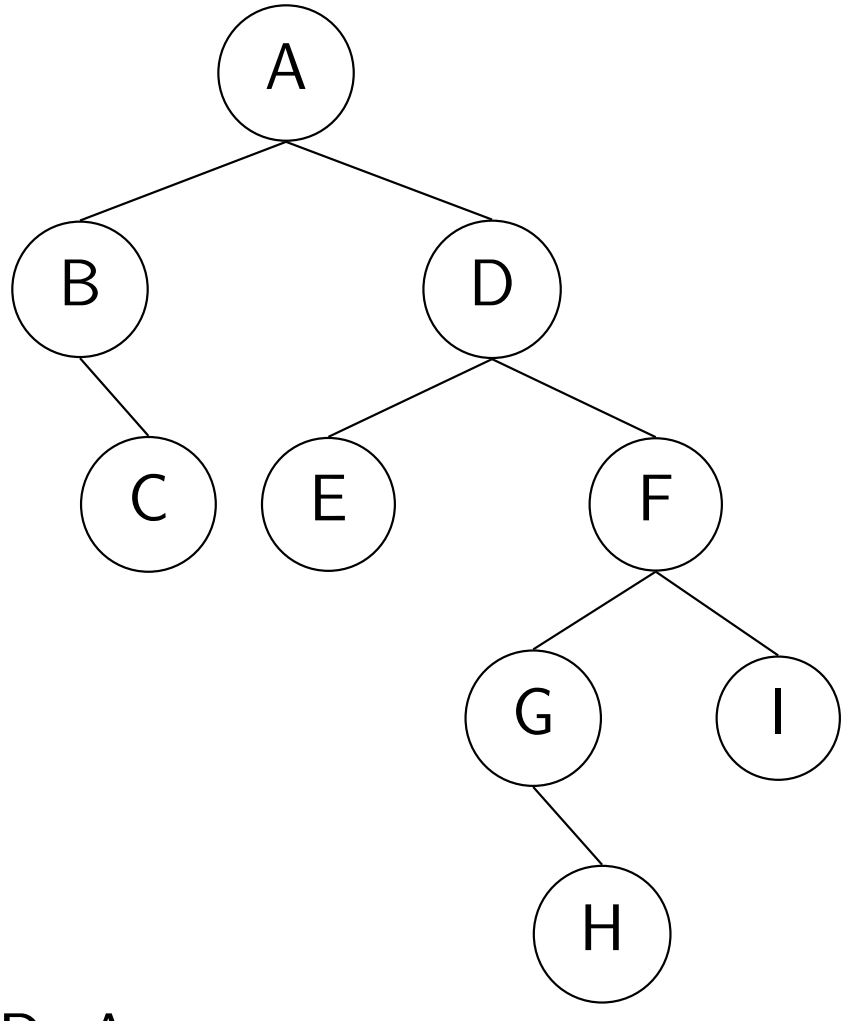


输出：C B E H G I F D

后序的非递归算法（例子）

栈：

-
-
-
-
-



输出：C B E H G I F D A

树的孩子兄弟表示法

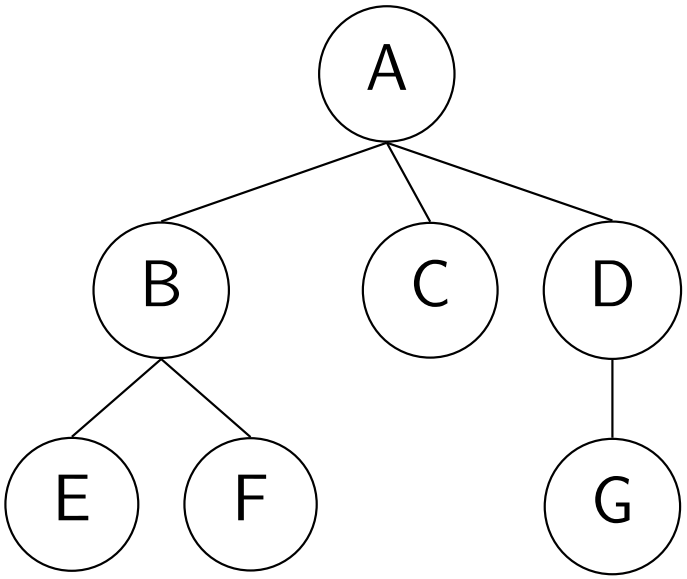
- 其实树和二叉树是有一个一一对应的
- 这个对应是通过树的孩子兄弟表示法
- 树的孩子表示法具体是每个结点记录它的第一个孩子与第一个右兄弟
- 关键：每个结点的指针就两个：第一孩子、第一右兄弟
- 二叉树每个结点也是两个指针！

树的孩子兄弟表示法

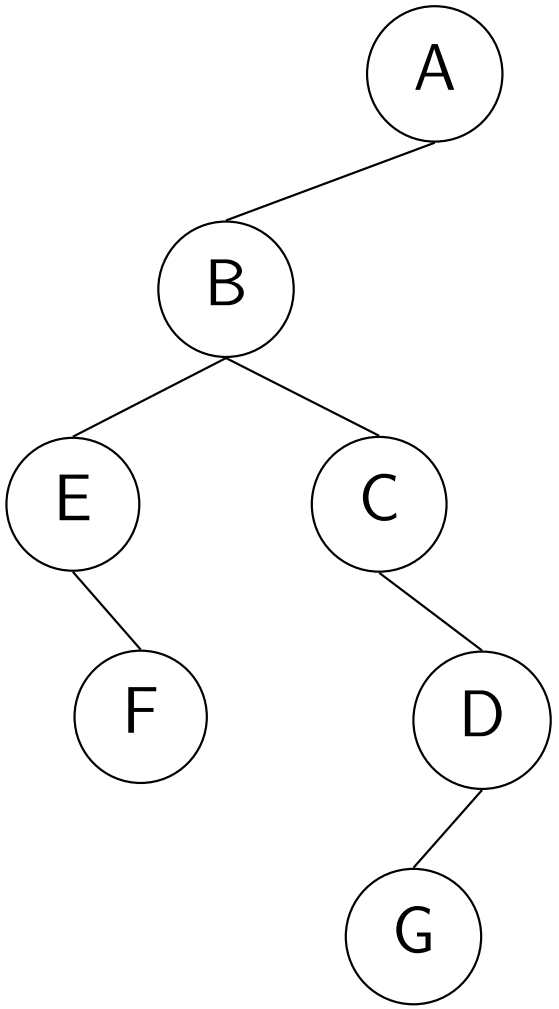
- 其实树和二叉树是有一个一一对应的
- 这个对应是通过树的孩子兄弟表示法
- 树的孩子表示法具体是每个结点记录它的第一个孩子与第一个右兄弟
- 关键：每个结点的指针就两个：第一孩子、第一右兄弟
- 二叉树每个结点也是两个指针！我们把第一孩子对应二叉树的左孩子，第一右兄弟对应二叉树的右孩子

对应例子

树：



二叉树：



树与二叉树对应下的前序

- 一般树的前序要如何在对应二叉树中体现？

树与二叉树对应下的前序

- 一般树的前序要如何在对应二叉树中体现？
- 二叉树的左孩子对应一般树的第一孩子，右孩子对应一般树的第一右兄弟。
- 按根、左、右的顺序，那么对应一般树时：根、左就是这个结点的子树，右就是第一个右兄弟，顺序与一般树的前序一致！
- 对应关系下，一般树的前序与对应二叉树的前序一致。

树与二叉树对应下的后序

- 一般树的后序要如何在对应二叉树中体现？

树与二叉树对应下的后序

- 一般树的后序要如何在对应二叉树中体现？
- 要把孩子子树部分先访问完，访问本结点，最后到兄弟。
- 因此从上述说明：顺序为左、根、右。
- 对应关系下，一般树的后序与对应二叉树的中序一致。

二叉树的应用：哈夫曼树、编码

- 假如你有一个字串，里头只用了字母 A,B,C
- 一般情况下：使用 ASCII 码，每个字符我们用了 8 个 bit 的空间。
- 问题：有没有办法少用一点空间？

二叉树的应用：哈夫曼树、编码

- 假如你有一个字串，里头只用了字母 A,B,C
- 一般情况下：使用 ASCII 码，每个字符我们用了 8 个 bit 的空间。
- 问题：有没有办法少用一点空间？
- 其实这里就是三个不同的字符而已：所以每个用 2 个 bit 就行了吧。（比如：A 对应 00，B 对应 01,C 对应 10）。
- 如果知道出现的大概频率我们可以做的更好。

前缀编码

- 其实编码这个概念你们都有接触到：比如 ASCII 编码中，每个字母都有一个编码，数字也是，标点符号等等也有对应编码……
- 数学一点：我们有一个编码的码字集合 C，原本的字符集 M，那么编码就是一个从 M 到 C 的单射。
- 偏向计算机方向：码字集合一般为 0,1 的数组串，就像我们之前的例子。
- 前缀编码：所有码字都不是其他码字的前缀。
- 前缀编码的最大好处：对给出的已编码串，可以唯一的解出原文

哈夫曼编码

- 问题：给定一堆字符与每个字符在文章的使用次数，设计一个前缀编码让编码结果最短。
- 例：$\{A:1000, B:5, C:3\}$。
- 好的答案：$A:0, B:10, C:11$；长度共用了 1016。
- 坏的答案：$A:110, B:0, C:10$；长度共用了：3011。
- 符合问题要求的编码都称为哈夫曼编码。

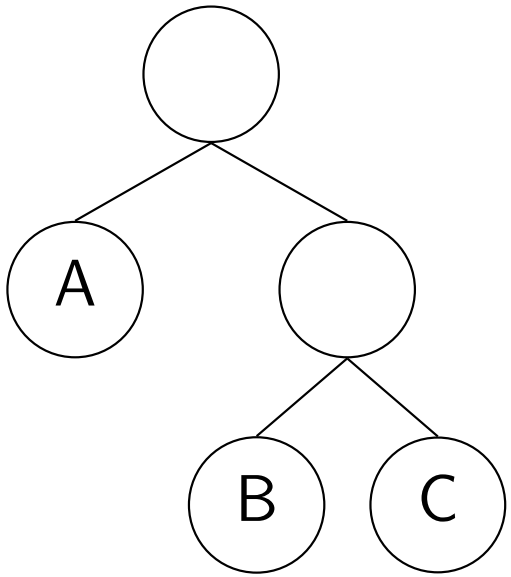
哈夫曼编码

- 问题：给定一堆字符与每个字符在文章的使用次数，设计一个前缀编码让编码结果最短。
- 例：$\{A:1000, B:5, C:3\}$。
- 好的答案：$A:0, B:10, C:11$；长度共用了 1016。
- 坏的答案：$A:110, B:0, C:10$；长度共用了：3011。
- 符合问题要求的编码都称为哈夫曼编码。
- 好了，说了那么久，树呢？

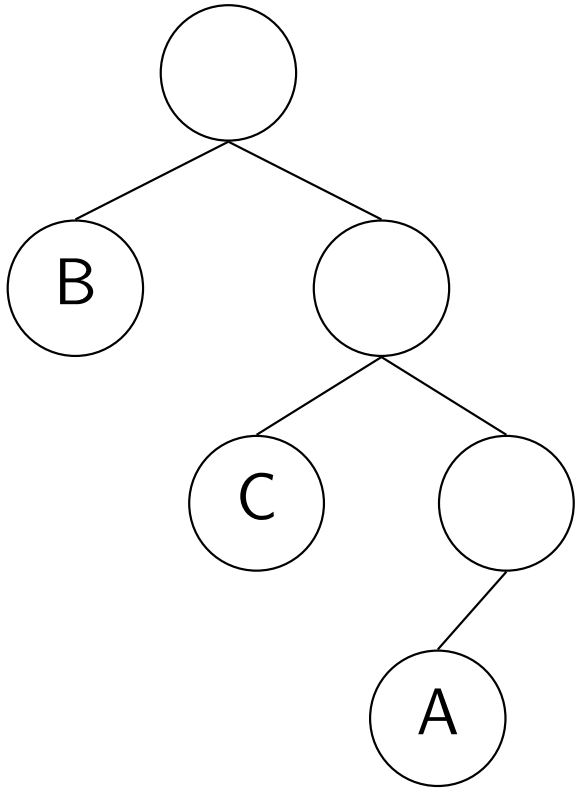
哈夫曼编码与二叉树的关系（例）

其实每个前述的编码都可以建成一棵相关的二叉树。具体是把 0 看成左孩子，1 看成右孩子。例：

$\{A:0, B:10, C:11\}$。



$\{A:110, B:0, C:10\}$。



哈夫曼编码与二叉树的关系

- 在前面的基础下，我们发现每个使用 0、1 有限序列的编码都可以看成一棵二叉树
- 叶子就是我们的原文字符
- 从根到叶子的路径为其编码（左为 0，右为 1）
- 那么前述的问题就等价于找出符合上述条件的最小带权长度的二叉树。

哈夫曼编码与二叉树的关系

- 在前面的基础下，我们发现每个使用 0、1 有限序列的编码都可以看成一棵二叉树
- 叶子就是我们的原文字符
- 从根到叶子的路径为其编码（左为 0，右为 1）
- 那么前述的问题就等价于找出符合上述条件的最小带权长度的二叉树。
- 明显地，度为 1 的点是不必要的。
- 使用越频繁的字符应该在比较靠近根的地方。

哈夫曼算法

1. 对每个字符做一个结点，权重为使用频率。(得到一个森林)
2. 当森林里树的数目多于一棵时：

2a. 选出权（如为单结点）或叶子权值总和（若为已合并过的树）最小的两棵树。

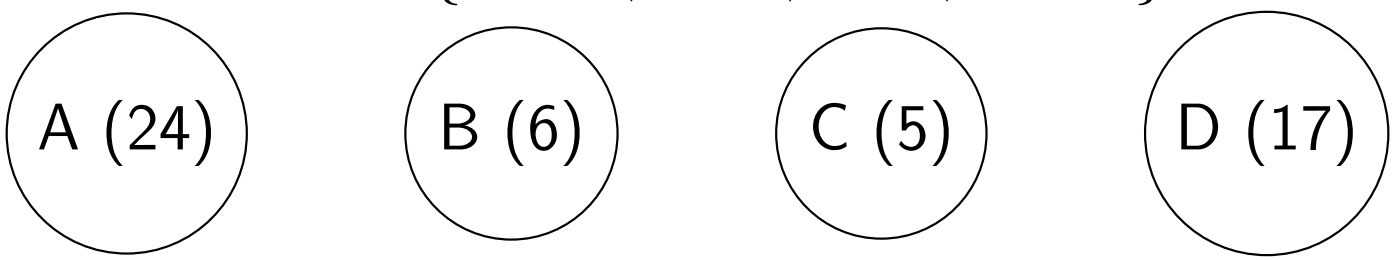
2b. 新造一棵树，其左、右孩子为上步选择的两棵树。

2c. 删去那两棵树，并把新的加到森林去。

3. 剩下的这棵树即为符合要求的编码的对应哈夫曼树。

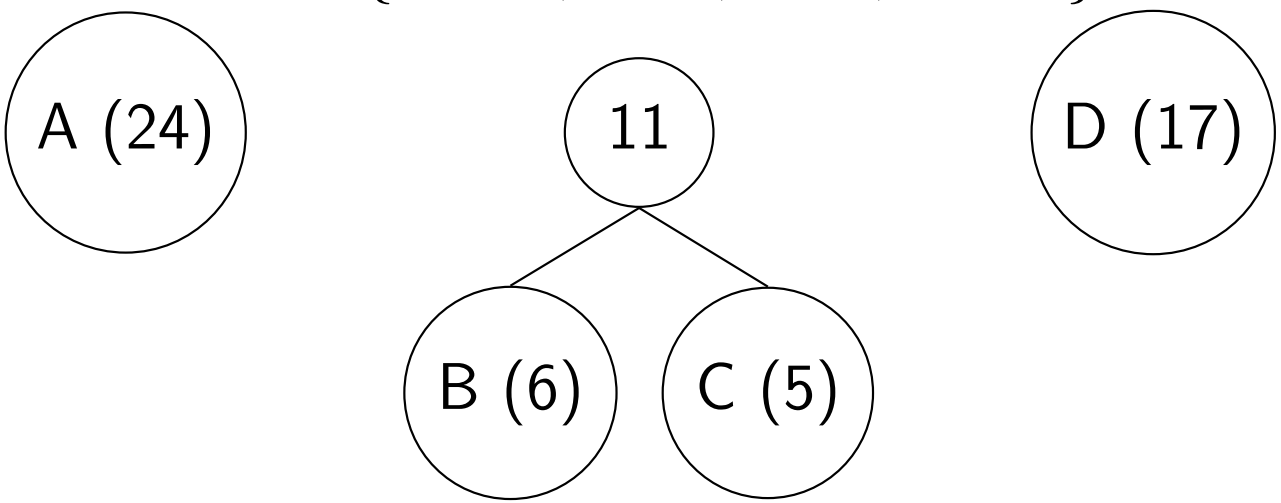
例子

给定 $\{A:24, B:6, C:5, D:17\}$。



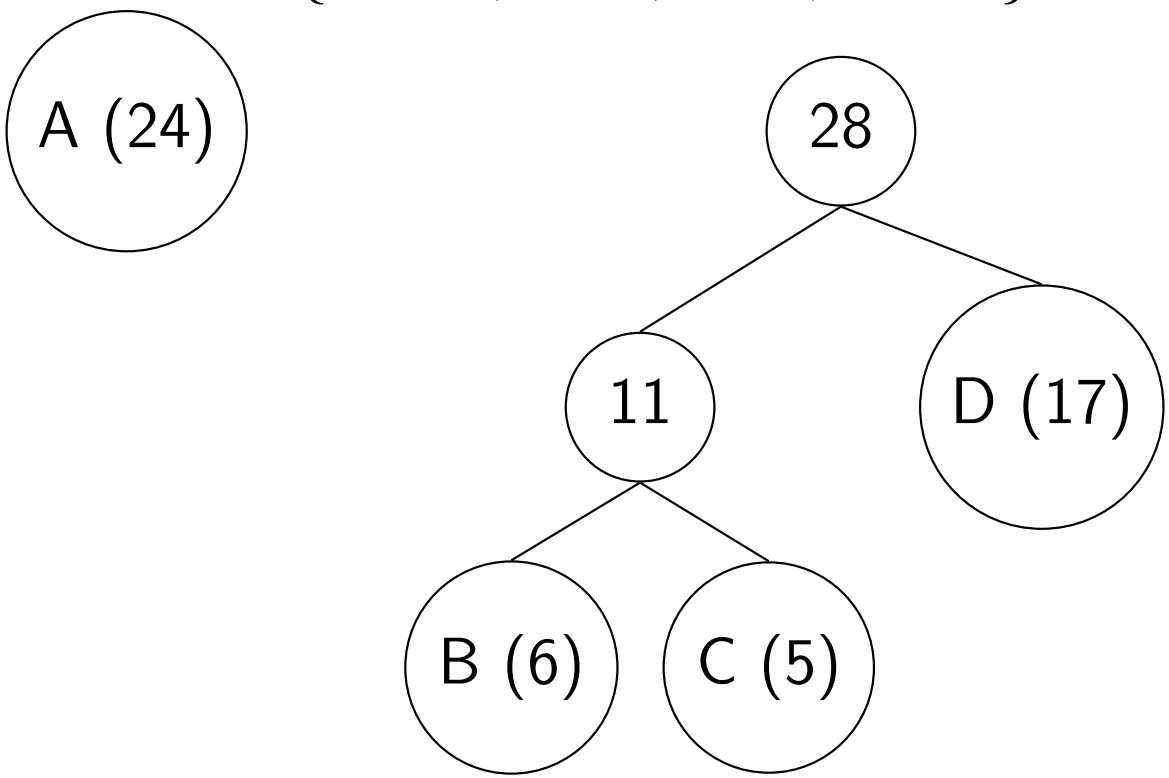
例子

给定 $\{A:24, B:6, C:5, D:17\}$。



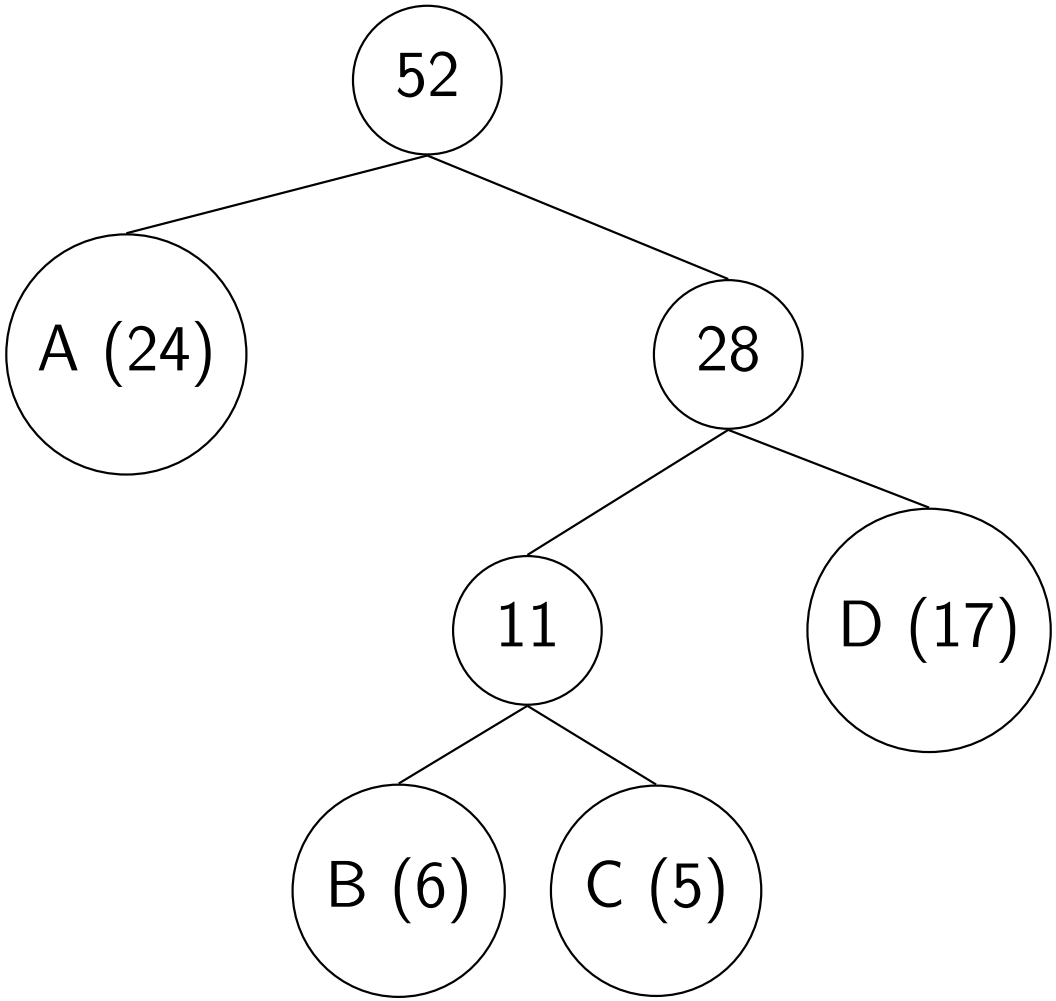
例子

给定 $\{A:24, B:6, C:5, D:17\}$。



例子

给定 $\{A:24, B:6, C:5, D:17\}$。



$A:0, B:100, C:101, D:11$。

试一下

给出 $\{A:9, B:11, C:5, D:7, E:8, F:2, G:3\}$

试一下

给出 $\{A:9, B:11, C:5, D:7, E:8, F:2, G:3\}$
可行答案：

$$A:00$$
$$B:10$$
$$C:010$$
$$D:110$$
$$E:111$$
$$F:0110$$
$$G:0111$$