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 18 日

1. 树的基本概念

- 定义和基本术语
- 树的表示形式
- 树的抽象数据类型

2. 树的存储结构

3. 二叉树

4. 遍历二叉树及其应用

5. 二叉树相关代码

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. 树的基本概念
   - 定义和基本术语
   - 树的表示形式
   - 树的抽象数据类型
2. 树的存储结构
3. 二叉树
4. 遍历二叉树及其应用
5. 二叉树相关代码

1. 树的基本概念
    - 定义和基本术语
    - 树的表示形式
    - 树的抽象数据类型
2. 树的存储结构
3. 二叉树
4. 遍历二叉树及其应用
5. 二叉树相关代码

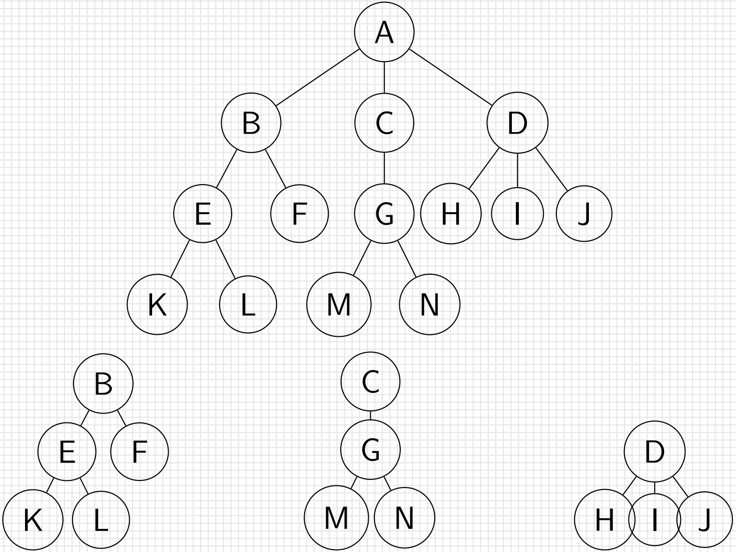
定义和基本术语

定义 (树-Tree)　树是 $n(n \geq 0)$ 个结点的有限集合 T。$n = 0$ 时称为空树。在任意一棵非空树中：

(1) 有且只有一个特定的称为根 (Root) 的结点；

(2) 若 $n > 1$ 时，其余结点被分为 $m(m > 0)$ 个互不相交的子集 $T_1, T_2, \cdots, T_m$，其中每个子集本身又是一棵树，称其为根的子树 (Subtree)。

定义和基本术语



定义和基本术语

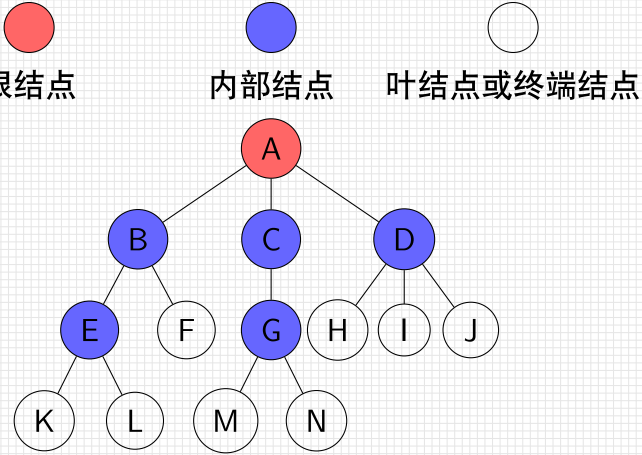
- $n>0$ 时，根节点是唯一的，不可能存在多个根结点。
- $m>0$ 时，子树的个数没有限制，但它们一定互不相交。

定义和基本术语

定义 (结点-node)　树的结点包含一个数据元素及其若干指向其子树的分支。

定义和基本术语

定义 (结点-node)　树的结点包含一个数据元素及其若干指向其子树的分支。

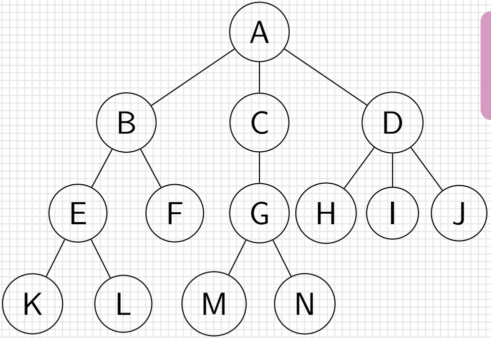


定义和基本术语

定义 (结点的度、树的度)　结点所拥有的子树的棵数称为结点的度。树中结点度的最大值称为树的度。

定义和基本术语

定义 (结点的度、树的度)　结点所拥有的子树的棵数称为结点的度。树中结点度的最大值称为树的度。



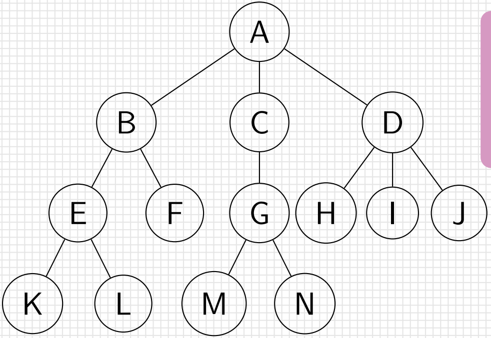
A 的度为 3，B 的度为 2，M 的度为 0，树的度为 3.

定义和基本术语

定义 (叶子结点、分支结点)　度为 0 的结点称为叶子结点；度不为 0 的结点称为分支结点。除根结点外，分支结点又称为内部结点。

定义和基本术语

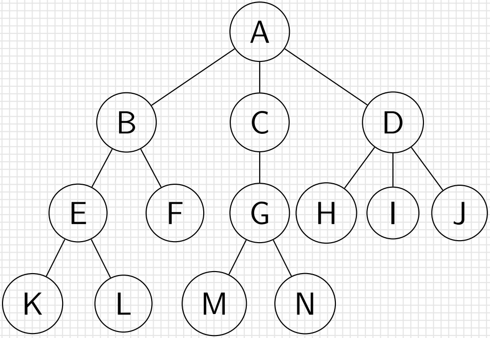
定义 (叶子结点、分支结点)　度为 0 的结点称为叶子结点；度不为 0 的结点称为分支结点。除根结点外，分支结点又称为内部结点。



H、I、J、K、L、M、N 是叶子结点，而所有其它结点都是分支结点。

定义和基本术语

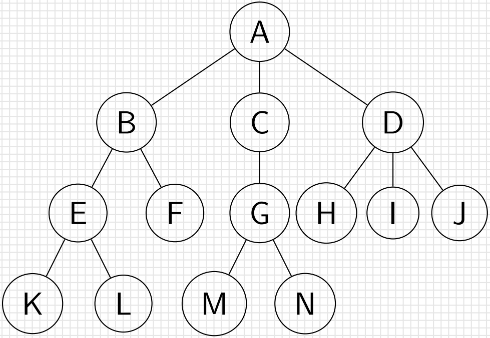
定义 (孩子结点、双亲结点、兄弟结点)　结点的子树的根称为该结点的孩子 (child)，相应地，该结点是其孩子结点的双亲 (parent)。同一双亲的所有子结点互称兄弟 (sibling)。



B、C、D 是 A 的孩子，而 A 是 B、C、D 的双亲；类似地，E、F 是 B 的孩子，B 是 E、F 的双亲。B、C、D 互为兄弟；E、F 互为兄弟。

定义和基本术语

定义 (层次、堂兄弟结点) 结点的层次 (level)从根开始定义，根为第 1 层，根的孩子为第 2 层。若某结点在第 l 层，则其子树的根就在第 $l+1$ 层。双亲在同一层上的结点互为堂兄弟。

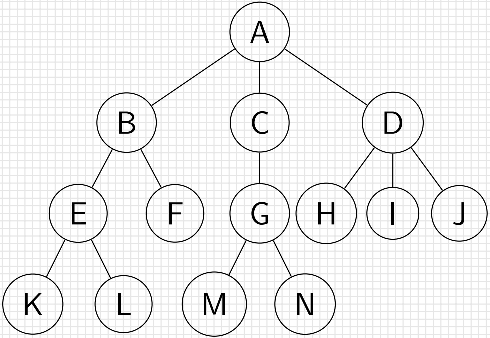


E、F、G、H、I、J 互为为堂兄弟。

定义和基本术语

定义 (结点的层次路径、祖先、子孙)　从根结点开始，到达某结点 p 所经过的所有结点构成结点 p 的层次路径(有且只有一条)。结点 p 的层次路径上的所有结点 (p 除外) 称为 p 的祖先 (ancester) 。以某一结点为根的子树中的任意结点称为该结点的子孙结点 (descent)。

定义和基本术语

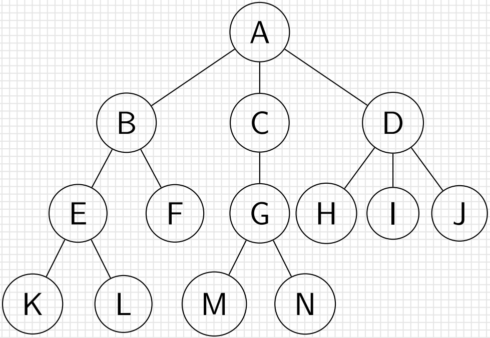


K 的层次路径为：A、B、E、K。

A、B、E 为 K 的祖先。

定义和基本术语

定义 (树的深度-depth)　树中结点的最大层次值，又称为树的高度。



该树的高度为 4.

定义和基本术语

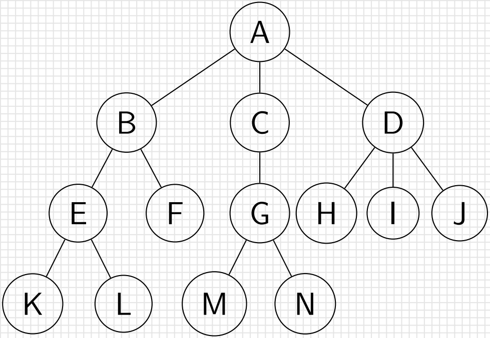
定义 (有序树和无序树) 对于一棵树，若其中每一个结点的子树具有一定的次序，则该树称为有序树，否则称为无序树。

定义 (森林-forest) 森林是 $m(m \geq 0)$ 棵互不相交的树的集合。显然，若将一棵树的根结点删除，剩余的子树就构成了森林。

1. 树的基本概念
    - 定义和基本术语
    - 树的表示形式
    - 树的抽象数据类型
2. 树的存储结构
3. 二叉树
4. 遍历二叉树及其应用
5. 二叉树相关代码

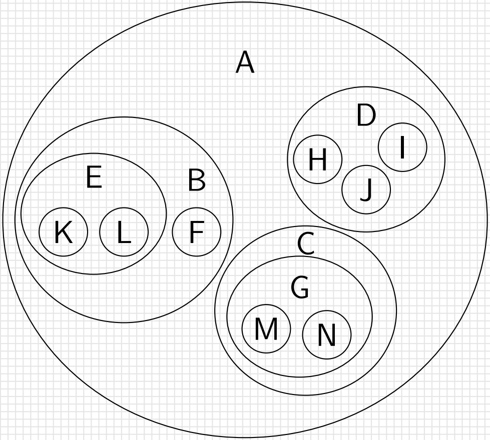
树的表示形式

1、倒悬树



树的表示形式

2、嵌套集合



树的表示形式

3、**广义表形式**

(A(B(E(K,L),F), C(G(M,N)), D(H,I,J)))

线性结构与树型结构的区别

线性结构

- 第一个元素：无前驱
- 最后一个元素：无后继
- 中间元素：一个直接前驱和一个直接后继

树结构

- 根结点：无双亲，且唯一
- 叶结点：无孩子，不唯一
- 中间结点：一个双亲多个孩子

1. 树的基本概念
    - 定义和基本术语
    - 树的表示形式
    - 树的抽象数据类型
2. 树的存储结构
3. 二叉树
4. 遍历二叉树及其应用
5. 二叉树相关代码

树的抽象数据类型

```
ADT Tree{
Data:
树由一个根结点和若干棵子树构成。结点有相同数据类型及层次关系。
Operation:
  Init(*T):
  Clear(*T):
  Creat(*T):
  IsEmpty(T):
  Depth(T):
  Root(T):
  Value(T, e):
  Assign(T, e, value):
  ...
} ADT Tree
```

1. 树的基本概念
    - 定义和基本术语
    - 树的表示形式
    - 树的抽象数据类型
2. 树的存储结构
3. 二叉树
4. 遍历二叉树及其应用
5. 二叉树相关代码

树的存储结构

存储结构有两种方式，即顺序存储和链式存储。

- 顺序存储使用一段连续的存储单元依次存储各个数据元素。这对于线性表来说非常自然，但对树这样一对多的结构呢？
- 树中某个结点的孩子可以有多个，这意味着，无论按何种顺序将树中所有结点存储到数组中，结点的存储位置都无法直接反映其逻辑关系。也就是说简单的顺序存储结构不能满足树的实现要求。

树的存储结构

我们可以充分利用顺序存储和链式存储结构的特点，来实现对树的存储结构的表示。

树的存储结构

- 双亲表示法
- 孩子表示法
- 孩子兄弟表示法

双亲表示法

假设用一组连续空间来存储树的结点，同时在每个结点中附设一个指示器，以指示其双亲结点在数组中的位置。也就是说，每个结点除了知道自己是谁以外，还需知道其双亲在哪儿。

data	parent

data	数据域	存储结点的数据信息
parent	指针域	存储该结点的双亲在数组中的下标

图: 双亲表示法的结点结构

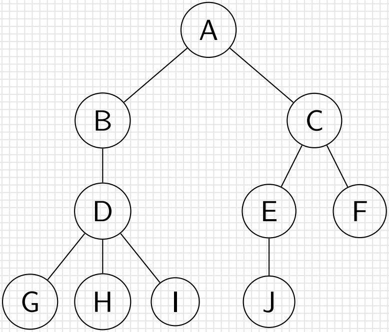
双亲表示法

```
#define MAX_TREE_SIZE 100
typedef int ElemType;
typedef struct {
  ElemType data;          /* 结点数据 */
  int parent;             /* 双亲位置 */
} PTNode;
typedef struct {
  PTNode nodes[MAX_TREE_SIZE];  /* 结点数组 */
  int r, n;           /* 根的位置和结点数 */
} Tree;
```

双亲表示法

由于根结点没有双亲，约定其指针域为 -1，也就是说，所有结点都存有其双亲的位置。

双亲表示法



index	data	parent
0	A	-1
1	B	0
2	C	0
3	D	1
4	E	2
5	F	2
6	G	3
7	H	3
8	I	3
9	J	4

双亲表示法

这样的存储结构，根据结点的 parent 指针可以很容易找到其双亲结点，时间复杂度为 $O(1)$。当 parent 为 -1 时，表示找到了根结点。

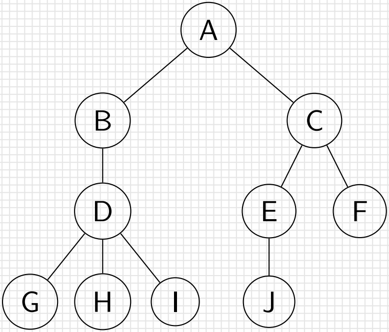
但是，若想知道结点的孩子是什么，则必须遍历整个结构。

双亲表示法 + 长子域

改进:

增加一个最左边孩子的域，不妨称之为长子域，这样就可以很容易找到结点的孩子。若该结点没有孩子，其长子域就设为 -1。

双亲表示法 + 长子域

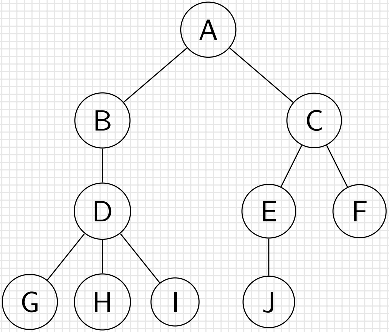


index	data	parent	firstchild
0	A	-1	1
1	B	0	3
2	C	0	4
3	D	1	6
4	E	2	9
5	F	2	-1
6	G	3	-1
7	H	3	-1
8	I	3	-1
9	J	4	-1

双亲表示法 + 右兄弟域

如若关注各兄弟之间的关系，可增加一个右兄弟域来体现兄弟关系，亦即，某个结点若存在右兄弟，则记录下其右兄弟的下标；若不存在，则赋值为 -1。

双亲表示法 + 右兄弟域



index	data	parent	rightsib
0	A	-1	-1
1	B	0	2
2	C	0	-1
3	D	1	-1
4	E	2	5
5	F	2	-1
6	G	3	7
7	H	3	8
8	I	3	-1
9	J	4	-1

双亲表示法 + 右兄弟域

若结点的孩子很多，超过了 2 个。但我们又关注结点的双亲、结点的孩子以及结点的兄弟，并且对时间遍历要求还比较高，那我们可以把此结构扩展为有双亲域、长子域以及右兄弟域。

树的存储结构

存储结构的设计是一个非常灵活的过程。一个存储结构设计得是否合理，取决于基于该存储结构的运算是否适合、是否方便，时间复杂度好不好等。但也不是越多越好。

孩子表示法

由于树中每个结点可能有多棵子树，可以考虑用多重链表，即每个结点有多个指针域，其中每个指针指向一颗子树的根结点，该方法称为多重链表表示法。

不过，树的每个结点的度，即其孩子是不同的，可以考虑如下两种方案。

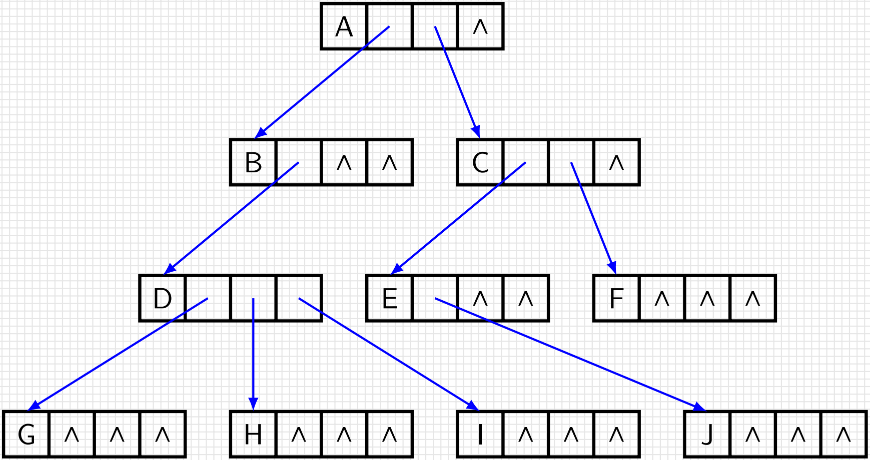
孩子表示法

方案一： 指针域的个数等于树的度。

data	child1	child2	child3	…	childd

data	数据域	存储结点的数据信息
child1-childd	指针域	指向该结点的各个孩子的结点

孩子表示法



孩子表示法

当各结点的度相差很大时，该方法显然是浪费空间的。不过，当各结点的度相差很小时，开辟的空间得以充分利用，此存储结构的缺点反而成了优点。

孩子表示法

当各结点的度相差很大时，该方法显然是浪费空间的。不过，当各结点的度相差很小时，开辟的空间得以充分利用，此存储结构的缺点反而成了优点。

问题　如果很多指针域为空，为什么不能按需分配呢？

孩子表示法

方案二： 每个结点的指针域的个数等于该结点的度， 专门取一个位置来存储结点指针域的个数。

data	degree	child1	child2	child3	…	childd

data	数据域	存储结点的数据信息
degree	度域	存储该结点的孩子的个数
child1-childd	指针域	指向该结点的各个孩子的结点

孩子表示法

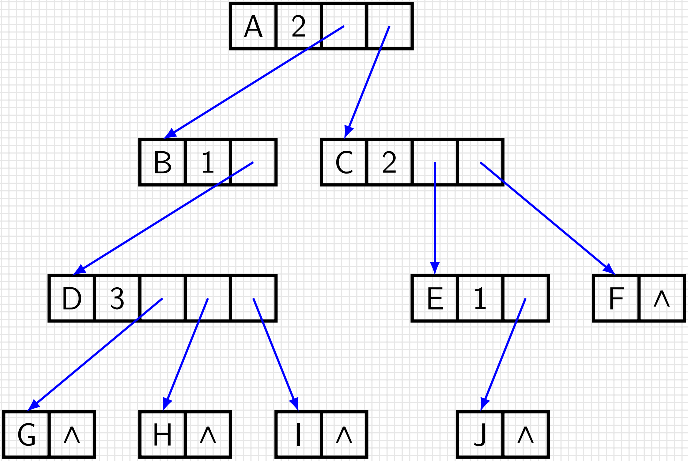


图: 多重链表表示法 2：按需分配指针域

孩子表示法

该方法克服了空间浪费的缺点，但由于各个结点的链表结构不同，需要维护结点度的值，在运算上会带来时间上的损耗。

孩子表示法

该方法克服了空间浪费的缺点，但由于各个结点的链表结构不同，需要维护结点度的值，在运算上会带来时间上的损耗。

问题　能否有更好的办法，既可以减少空指针的浪费又能使结点结构相同？

孩子表示法

该方法克服了空间浪费的缺点，但由于各个结点的链表结构不同，需要维护结点度的值，在运算上会带来时间上的损耗。

问题　能否有更好的办法，既可以减少空指针的浪费又能使结点结构相同？

为了遍历整棵树，可以把结点放在一个顺序存储结构的数组中，但每个结点的孩子有多少不确定，可以再对每个结点的孩子建立一个单链表来体现它们的关系。

孩子表示法

把每个结点的孩子排列起来，以单链表做存储结构，n 个结点有 n 个孩子链表，如果是叶子结点则此单链表为空。

而 n 个结点组成一个线性表，采用顺序存储结构，存放在一个一维数组中。

孩子表示法

为此，设计两种结点结构：

1、孩子链表的孩子结点：

child	next

child	数据域	存储某个结点在表头数组中的下标
next	指针域	存储指向某结点的下一个孩子的指针

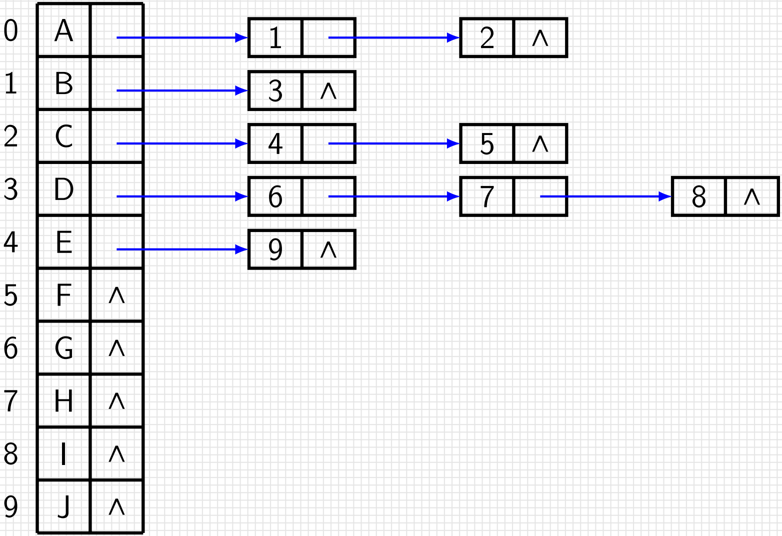
树的抽象数据类型

2、表头数组的表头结点：

data	firstchild

data	数据域	存储结点的数据信息
firstchild	头指针域	存储该结点的孩子链表的头指针

孩子表示法



孩子表示法

```
#define MAX_TREE_SIZE 100
typedef struct CTNode {        /* 孩子结点 */
  int child;
  struct CTNode *next;
} * ChildPtr;
typedef struct {               /* 表头结构 */
  ElemType data;
  ChildPtr firstchild;
} CTBox;
typedef struct {               /* 树结构 */
  CTBox nodes[MAX_TREE_SIZE];    /* 结点数组 */
  int r, n;            /* 根的位置和结点数 */
} Tree;
```

孩子表示法

该结构对于查找某个结点的某个孩子，或者找某个结点的兄弟，只需查找该结点的孩子单链表即可。

若想遍历整棵树，只需对头结点的数组循环即可。

孩子表示法

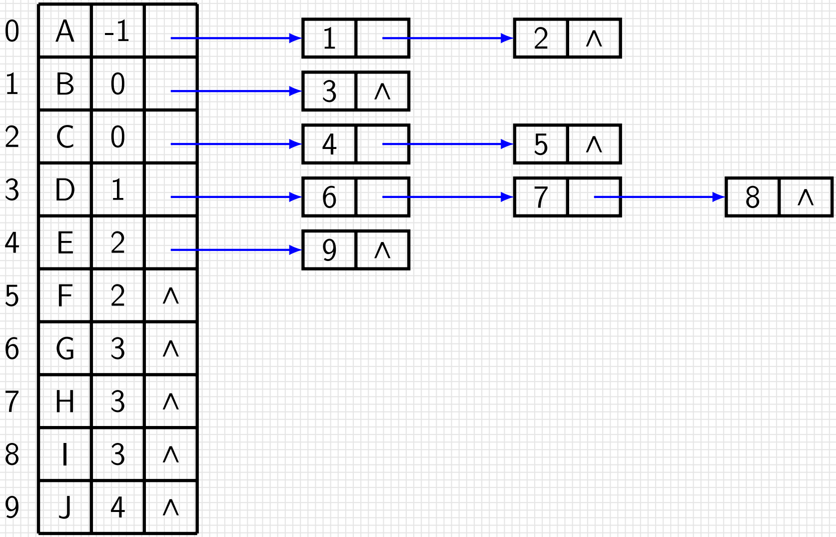
问题　如何知道某个结点的双亲呢？

孩子表示法

问题　如何知道某个结点的双亲呢？

需遍历整棵树，相对比较麻烦。可以考虑把双亲表示法和孩子表示法结合起来，设计一种所谓的“双亲孩子表示法”。

孩子双亲表示法



孩子兄弟表示法

问题　以上分别从双亲的角度和孩子的角度研究了树的存储结构，那么从树结点的兄弟的角度又会如何？

孩子兄弟表示法

问题　以上分别从双亲的角度和孩子的角度研究了树的存储结构，那么从树结点的兄弟的角度又会如何？

对于树这样的层级结构来说，只研究结点的兄弟是不行的。

孩子兄弟表示法

问题　以上分别从双亲的角度和孩子的角度研究了树的存储结构，那么从树结点的兄弟的角度又会如何？

对于树这样的层级结构来说，只研究结点的兄弟是不行的。

任何一棵树，其结点的长子如果存在就是唯一的，它的大弟如果存在也是唯一的。因此，可设置两个指针，分别指向该结点的长子和右兄弟。

孩子兄弟表示法

data	firstchild	rightsib

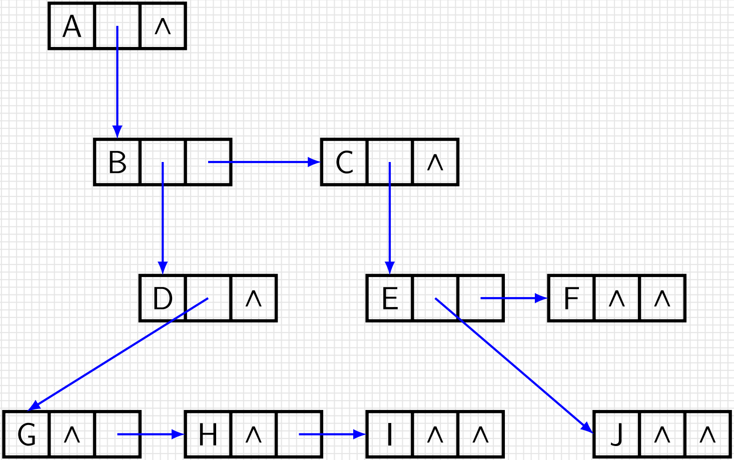
data	数据域	结点的数据信息
firstchild	指针域	长子的存储地址
rightsib	指针域	大弟结点的存储地址

图: 孩子兄弟表示法结构定义

孩子兄弟表示法

```
typedef struct CSNode {
  ElemType data;
  struct CSNode * firstchild, * rightsib;
} CSNode, * CSTree;
```

孩子兄弟表示法



孩子兄弟表示法

该表示法便于查找某个结点的某个孩子。对于某个结点，可通过 firstchild 找到其长子，再通过长子的 rightsib 找到其二弟，接着一直找下去，直到找到具体的孩子。

孩子兄弟表示法

该表示法便于查找某个结点的某个孩子。对于某个结点，可通过 firstchild 找到其长子，再通过长子的 rightsib 找到其二弟，接着一直找下去，直到找到具体的孩子。

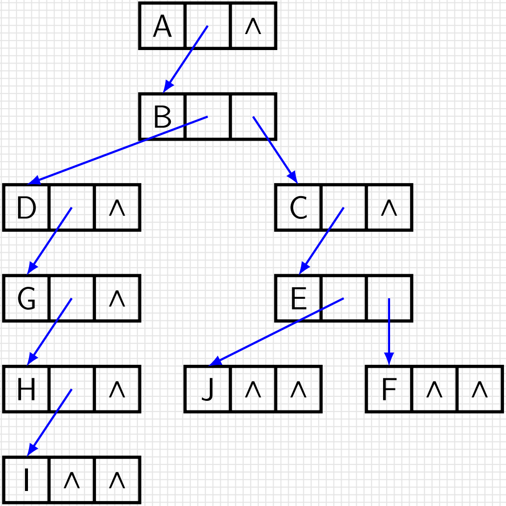
若想找某个结点的双亲，该表示法也有缺陷。

可考虑再增加一个 parent 指针域来解决快速查找双亲的问题。

孩子兄弟表示法

孩子兄弟表示法的最大好处是它把一颗复杂的树变成了一棵二叉树。

孩子兄弟表示法



1. 树的基本概念
   - 定义和基本术语
   - 树的表示形式
   - 树的抽象数据类型
2. 树的存储结构
3. 二叉树
4. 遍历二叉树及其应用
5. 二叉树相关代码

二叉树

定义 二叉树 (Binary tree) 是 $n(n \geq 0)$ 个结点的有限集合。当 $n = 0$ 时称为空树，否则

(1) 有且只有一个特殊的称为树的根 (Root) 结点；

(2) 当 $n > 1$ 时，其余结点被分成两个互不相交的子集 T_1, T_2，分别称之为左、右子树，并且左、右子树又都是二叉树。

由此可知，二叉树的定义是递归的。

二叉树

二叉树在树结构中起着非常重要的作用。因为二叉树结构简单，存储效率高，树的操作算法相对简单，且任何树都很容易转化成二叉树结构。上节中引入的有关树的术语也都适用于二叉树。

二叉树

二叉树的特点：

- 每个结点最多有两棵子树，所以二叉树不存在度大于 2 的结点。
- 左子树和右子树是有顺序的，次序不能任意颠倒。
- 对于树中的某结点，即使只有一棵子树，也要区分它是左子树还是右子树。

二叉树

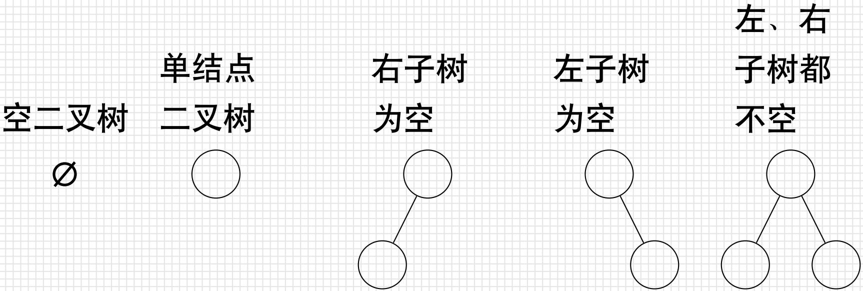


图: 二叉树的 5 种基本形态

特殊二叉树

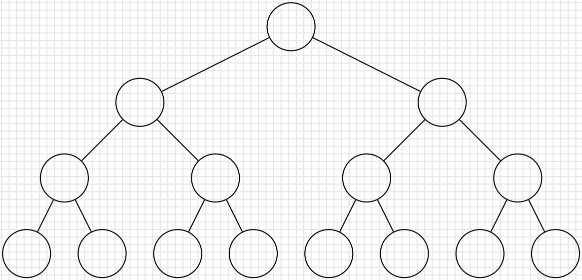
1、斜树

- 所有结点都只有左子树的二叉树叫左斜树；所有结点都只有右子树的二叉树叫右斜树。
- 特点： 每一层都只有一个结点，结点的个数与二叉树的深度相同。
- 这与线性表结构相同。其实线性表结构可以理解为树的一种极其特殊的表现形式。

特殊二叉树

2、满二叉树

在一棵二叉树中，如果所有分支结点都存在左子树和右子树，并且所有叶子都在同一层，这样的二叉树称为满二叉树。



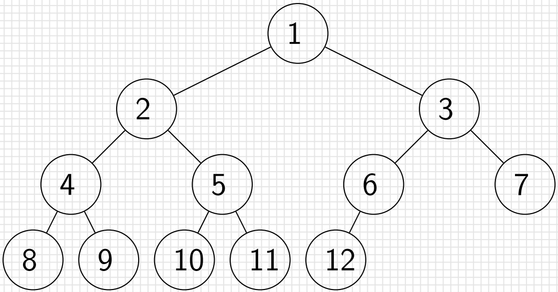
特殊二叉树

满二叉树的特点：

- 叶子只能出现在最下一层。
- 非叶子结点的度一定是 2。
- 若深度为 k，则结点数为 $2^k - 1$。在同样深度的二叉树中，满二叉树的结点个数最多，叶子数最多。
- 可对满二叉树的结点进行连续编号，若规定从根结点开始，按"自上而下、自左至右"的原则进行。

3、完全二叉树 (Complete Binary Tree)

对一棵深度为 k、结点数为 n 的二叉树按层序编号，若编号 $i(1 \le i \le n)$ 的结点与同样深度的满二叉树中编号为 i 的结点在二叉树中位置完全相同，则该二叉树称为完全二叉树。



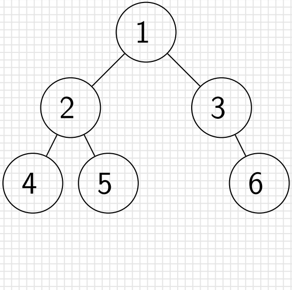


图: 非完全二叉树

完全二叉树是满二叉树的一部分，而满二叉树是完全二叉树的特例。

满二叉树的特点

- 叶子结点只能出现在最下两层。
- 最下层的叶子一定集中在左部连续位置。
- 倒数第二层，若有叶子结点，一定都在右部连续位置。
- 如果结点度为 1，则该结点只有左孩子，即不存在只有右子树的情况。
- 同样结点数的二叉树，完全二叉树的深度最小。

二叉树的性质

性质 (1)　在非空二叉树中，第 i 层上至多有 2^{i-1} 个结点 $(i \geq 1)$。

- 第一层是根节点，只有一个，$2^{1-1} = 2^0 = 1$。
- 第二层有两个，$2^{2-1} = 2^1 = 2$。
- 第三层有两个，$2^{3-1} = 2^2 = 4$。
- 第四层有两个，$2^{4-1} = 2^3 = 8$。

利用数学归纳法，容易得出在二叉树的第 i 层上至多有 2^{i-1} 个结点 $(i \geq 1)$。

二叉树的性质

性质 (2)　深度为 k 的二叉树至多有 2^k-1 个结点 $(k\geq 1)$.

- 如果有一层，至多 $1=2^1-1$ 个结点。
- 如果有二层，至多 $1+2=3=2^2-1$ 个结点。
- 如果有三层，至多 $1+2+4=2^3-1$ 个结点。
- 如果有四层，至多 $1+2+4+8=2^4-1$ 个结点。

利用数学归纳法，容易得出，如果有 k 层，此二叉树至多有 2^k-1 个结点。

二叉树的性质

性质 (3)　对任何一棵二叉树，若其叶子结点数为 n_0，度为 2 的结点数为 n_2，则 $n_0 = n_2 + 1$.

二叉树的性质

性质 n 个结点的完全二叉树深度为 $\lfloor \log_2 n \rfloor + 1$.

二叉树的性质

性质 (5)　若对一棵有 n 个结点的完全二叉树的结点按层序自左至右进行编号，则对于编号为 $i(1 \le i \le n)$ 的结点：

- 若 $i = 1$，i 号结点是根，无双亲；若 $1 < i \le n$，其双亲结点编号是 $\lfloor i/2 \rfloor$。
- 若 $2i \le n$，i 号结点的左孩子编号为 $2i$；若 $2i > n$，无左孩子。
- 若 $2i+1 \le n$，i 号结点的右孩子编号为 $2i+1$；若 $2i+1 > n$，无右孩子。

二叉树的顺序存储

二叉树存储结构的类型定义：

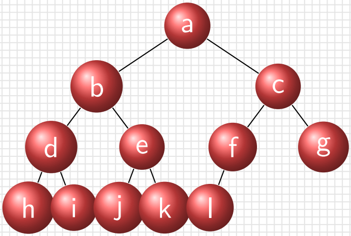
```
#define MAX_SIZE  100
typedef TElemType SqBiTree[MAX_SIZE];
```

用一组地址连续的存储单元依次“自上而下、自左至右”存储完全二叉树的数据元素。

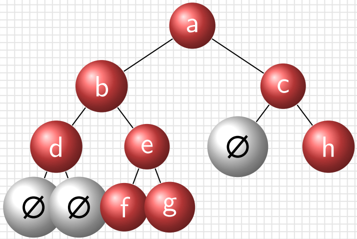
- 对于完全二叉树上编号为 i 的结点元素存储在一维数组的下标值为 $i-1$ 的分量中；
- 对于一般的二叉树，将其每个结点与完全二叉树上的结点相对照，存储在一维数组中。

二叉树的顺序存储

完全二叉树



非完全二叉树



1	2	3	4	5	6	7	8	9	10	11	12
a	b	c	d	e	f	g	h	i	j	k	l

1	2	3	4	5	6	7	8	9	10	11
a	b	c	d	e	∅	h	∅	∅	f	g

二叉树的顺序存储

最坏的情况下，一个深度为 k 且只有 k 个结点的单支树需要长度为 $2k-1$ 的一维数组。

二叉树的链式存储

设计不同的结点结构，可构成不同的链式存储结构。

结点类型及其定义

(1) 二叉链表结点。有三个域：一个数据域，两个分别指向左右子结点的指针域。

lchild	data	rchild

```
typedef struct BTNode{
  ElemType  data ;
  struct BTNode * lchild, * rchild ;
} BTNode;
```

图: 二叉链表结点

结点类型及其定义

(2) 三叉链表结点。除二叉链表的三个域外，再增加一个指针域，用来指向结点的父结点。

lchild	data	parent	rchild

```
typedef struct BTNode3{
  ElemType  data ;
  struct BTNode3  * lchild, * rchild, * parent;
}BTNode3;
```

图: 三叉链表结点

二叉树的链式存储形式

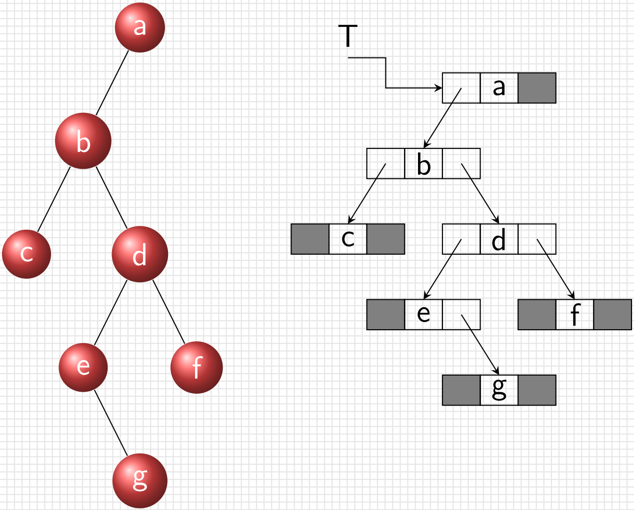


图: 二叉树及其二叉链表

二叉树的链式存储形式

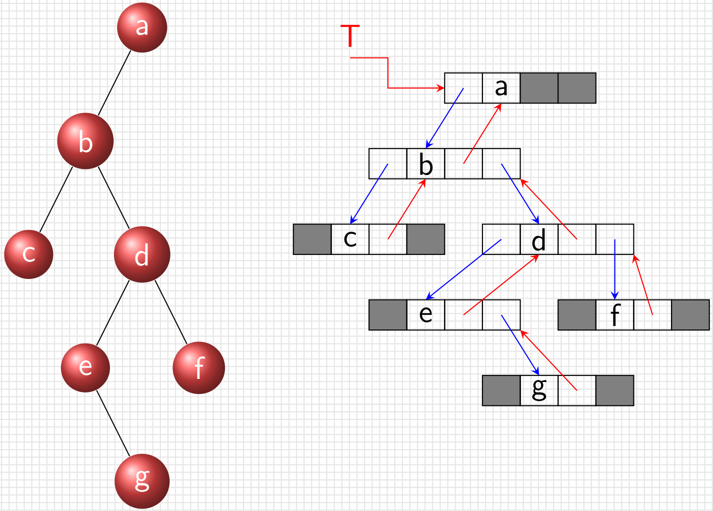


图: 二叉树及其三叉链表

1. 树的基本概念
    - 定义和基本术语
    - 树的表示形式
    - 树的抽象数据类型
2. 树的存储结构
3. 二叉树
4. 遍历二叉树及其应用
5. 二叉树相关代码

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

树的抽象数据类型

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：

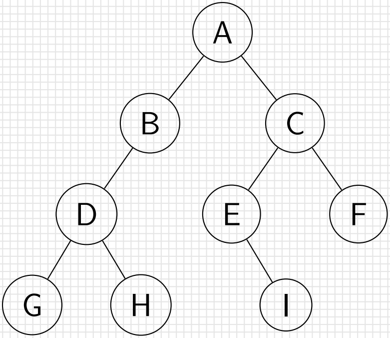
若二叉树为空，则遍历结束；否则

(1) 访问根结点；

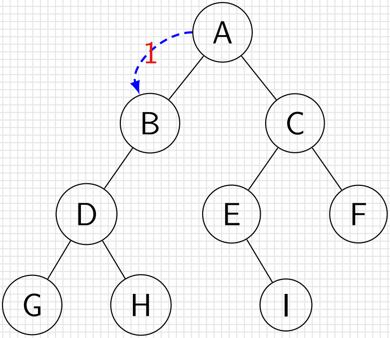
(2) 前序遍历左子树；

(3) 前序遍历右子树。

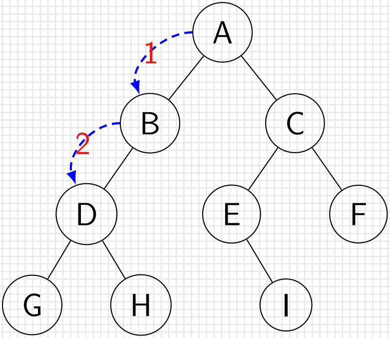
前序遍历



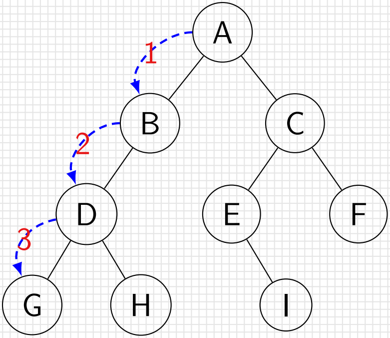
前序遍历



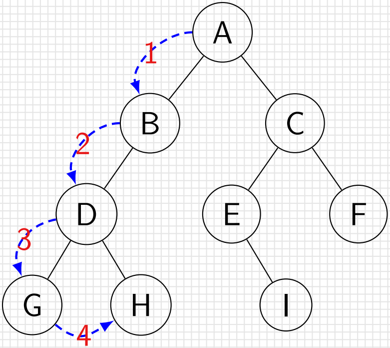
前序遍历



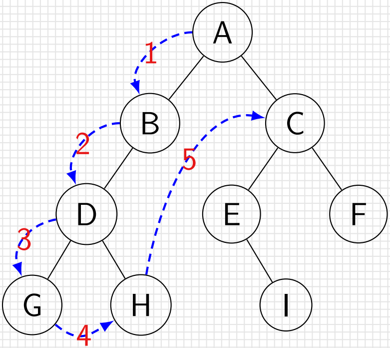
前序遍历



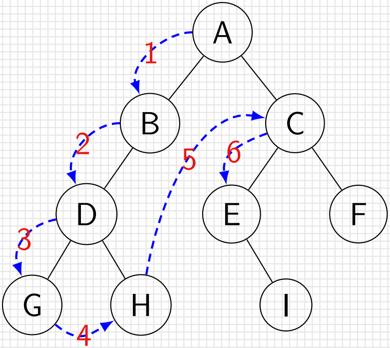
前序遍历



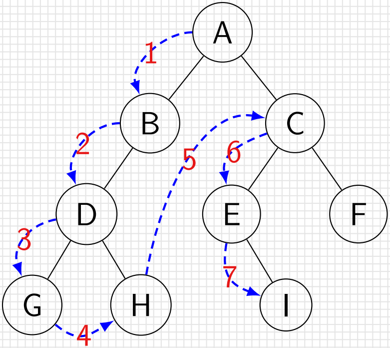
前序遍历



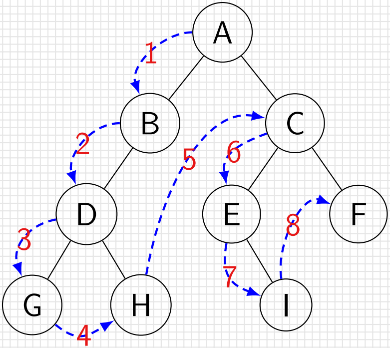
前序遍历



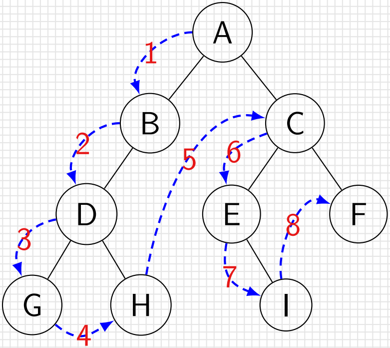
前序遍历



前序遍历



前序遍历



前序遍历结果： A B D G H C E I F

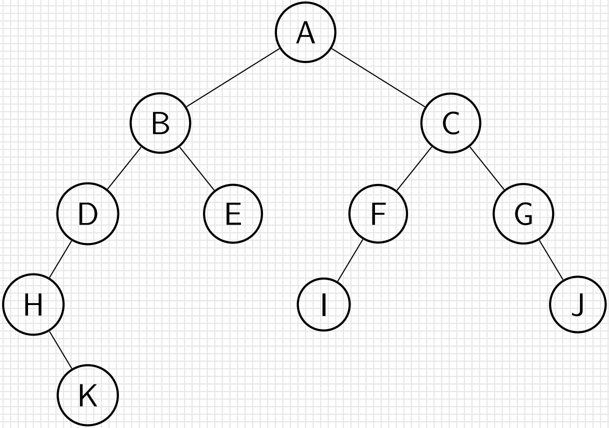
前序遍历

```
#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

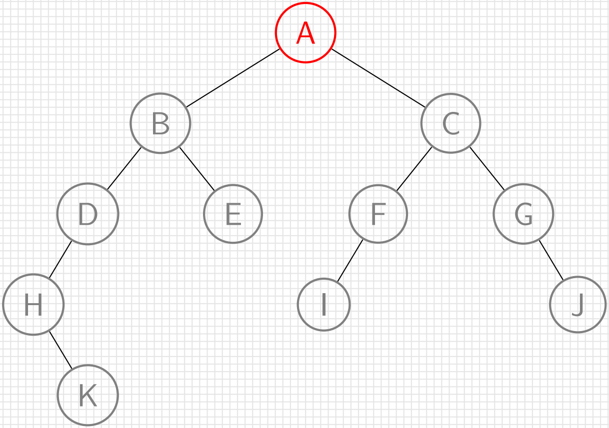
假设 visit 函数实现打印功能，即

```
#include"BiTree.h"

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

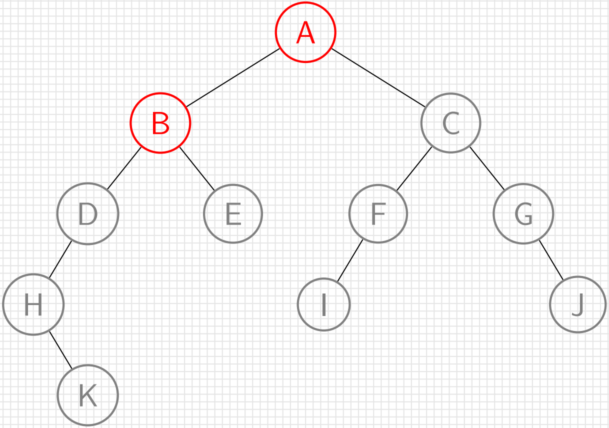
前序遍历

1、调用 `PreOrderTraverse(tree, Print)`，访问根结点 A，打印字母 A.



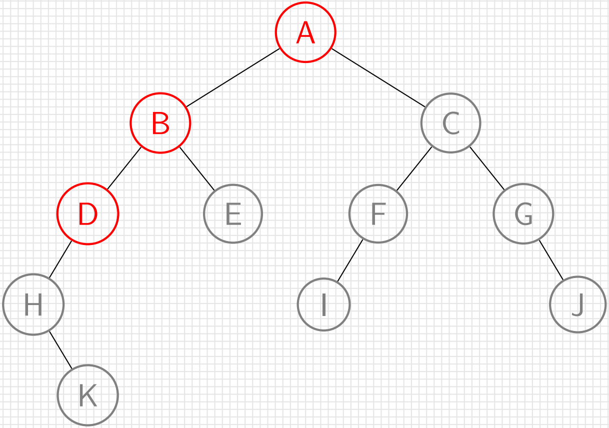
前序遍历

2、调用 `PreOrderTraverse(pnode->lchild, Print)`，访问结点 A 的左孩子 B，打印字母 B.



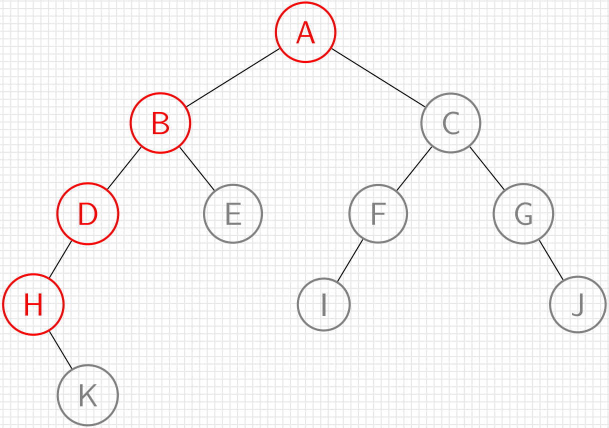
前序遍历

3、调用 `PreOrderTraverse(pnode->lchild, Print)`, 访问结点 B 的左孩子 D，打印字母 D.



前序遍历

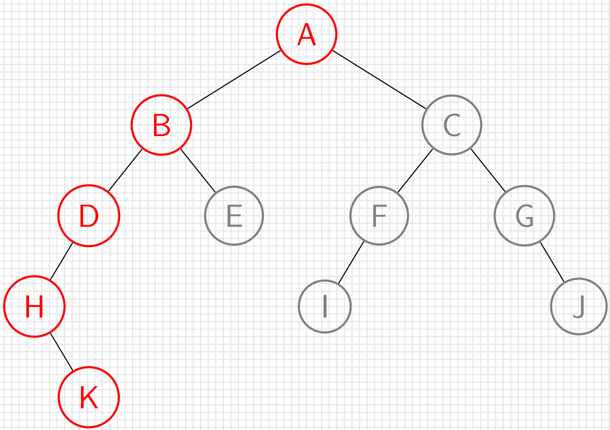
4、调用 `PreOrderTraverse(pnode->lchild, Print)`，访问结点 D 的左孩子 H，打印字母 H.



前序遍历

5、调用 `PreOrderTraverse(pnode->lchild, Print)`，访问结点 H 的左孩子，不存在，返回至结点 H；调用 `PreOrderTraverse(pnode->rchild, Print)`，访问结点 H 的右孩子 K，打印字母 K.

前序遍历



前序遍历

6、调用 `PreOrderTraverse(pnode->lchild, Print)`，访问结点 K 的左孩子，不存在，返回至结点 K；

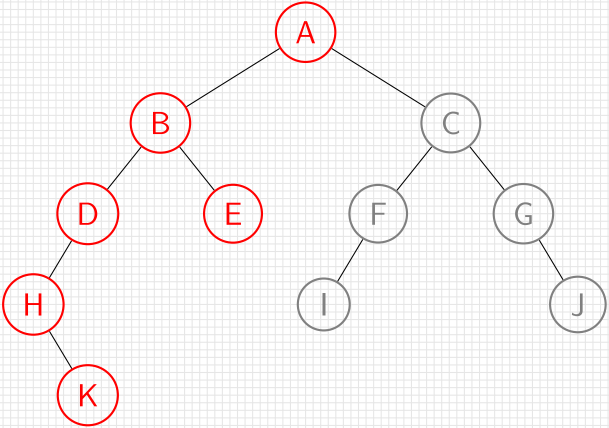
调用 `PreOrderTraverse(pnode->rchild, Print)`，访问结点 K 的右孩子，不存在，返回至结点 K；

至此，结点 K 访问完毕，返回至结点 H；结点 H 也访问完毕，返回到结点 D；

调用 `PreOrderTraverse(pnode->rchild, Print)`，访问结点 D 的右孩子，不存在，返回到 B 结点；

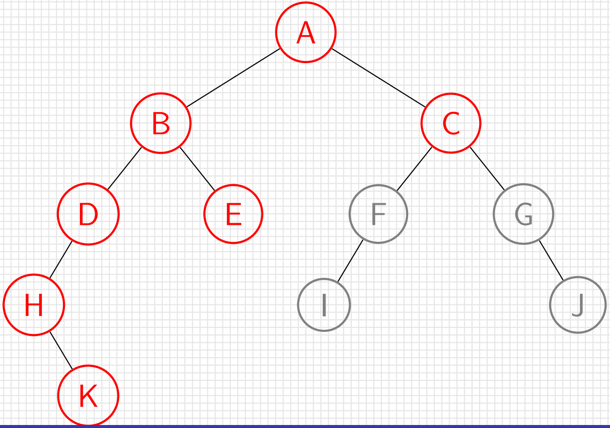
调用 `PreOrderTraverse(pnode->rchild, Print)`，访问结点 B 的右孩子 E，打印字母 E。

前序遍历



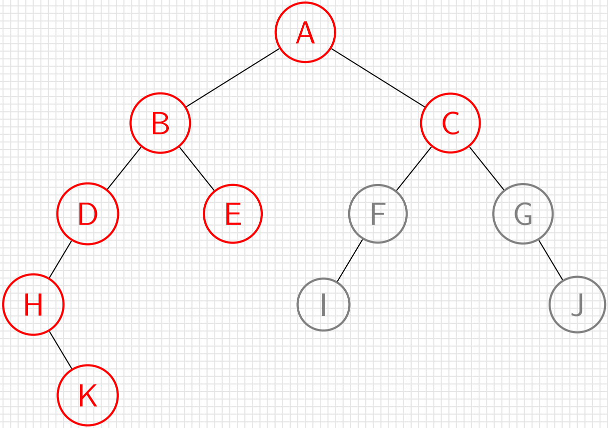
前序遍历

7、结点 E 没有孩子，返回到结点 B，结点 B 访问完毕，返回到结点 A；调用 `PreOrderTraverse(pnode->rchild, Print)`，访问 A 结点的右孩子 C，打印 C。



前序遍历

8、之后类似前面的递归调用，依次打印 F、I、G、J。



中序遍历

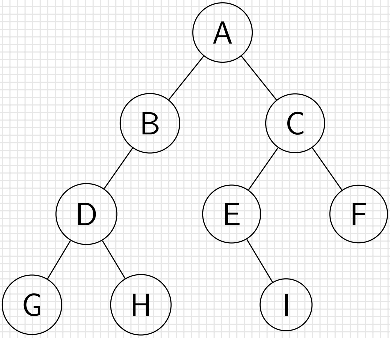
规则：若二叉树为空，则遍历结束；否则

(1) 中序遍历左子树；

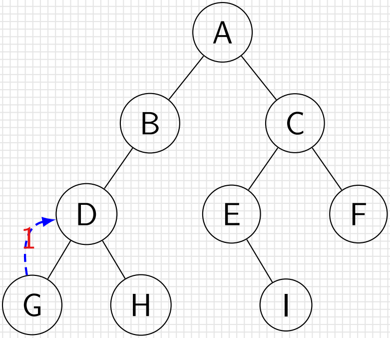
(2) 访问根结点；

(3) 中序遍历右子树。

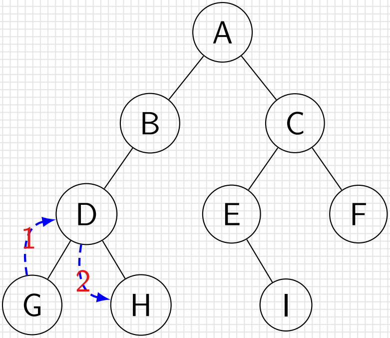
中序遍历



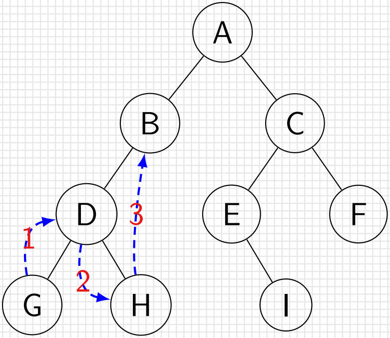
中序遍历



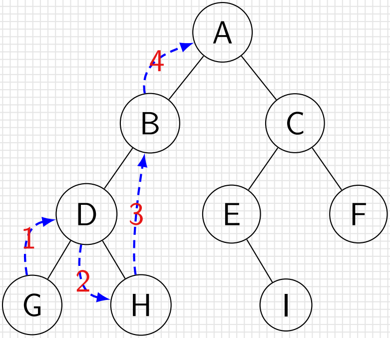
中序遍历



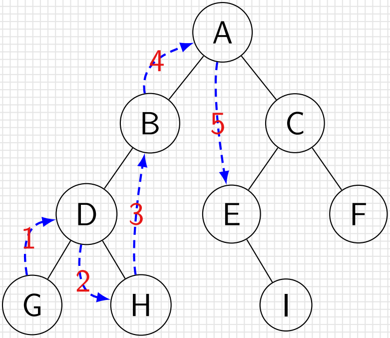
中序遍历



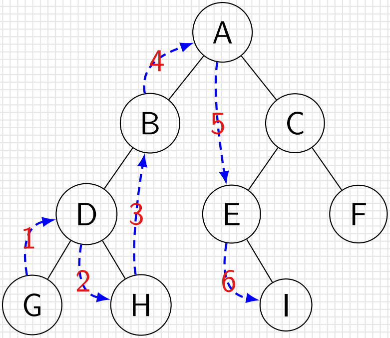
中序遍历



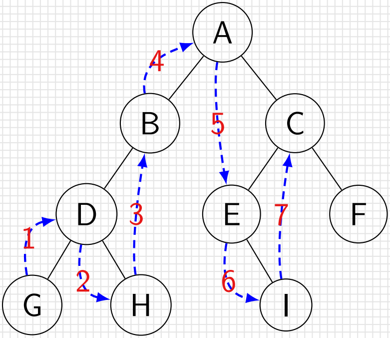
中序遍历



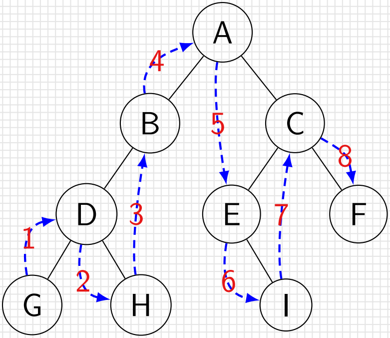
中序遍历



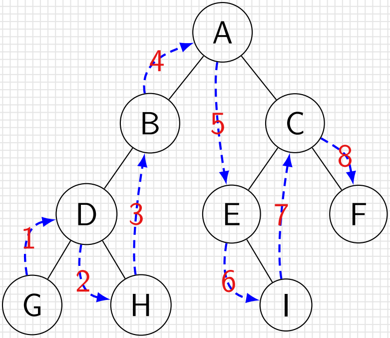
中序遍历



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

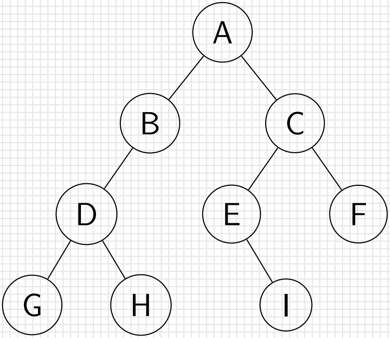
规则： 若二叉树为空，则遍历结束；否则

(1) 后序遍历左子树；

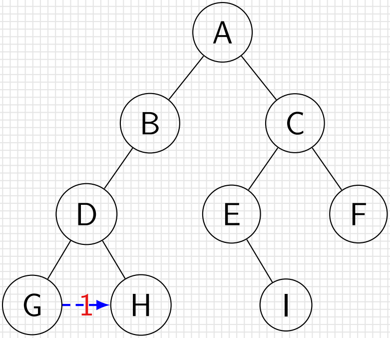
(2) 后序遍历右子树；

(3) 访问根结点。

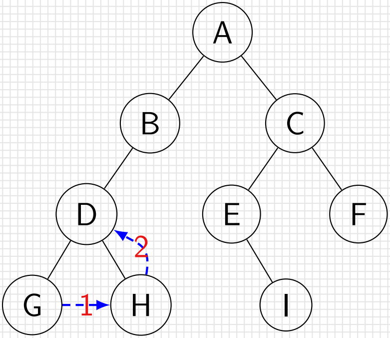
后序遍历



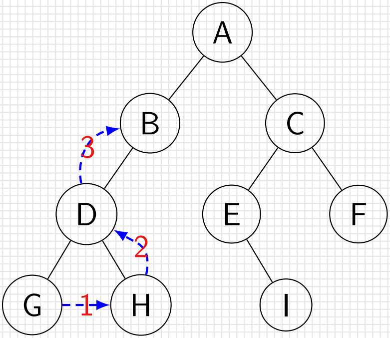
后序遍历



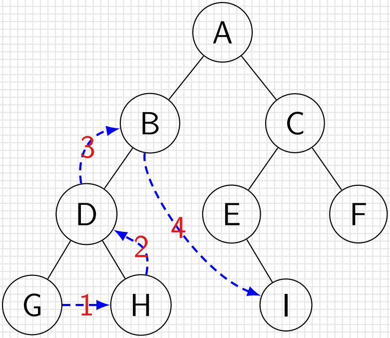
后序遍历



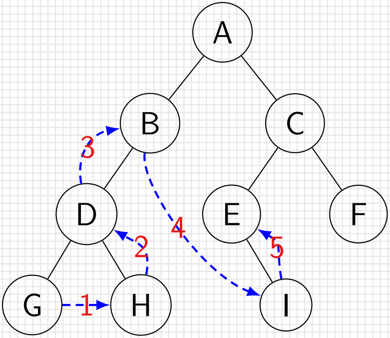
后序遍历



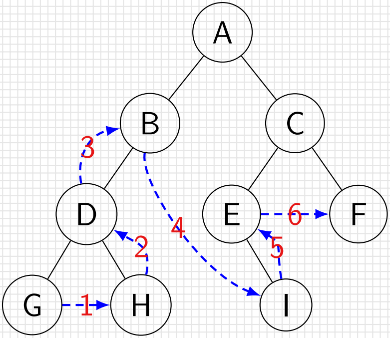
后序遍历



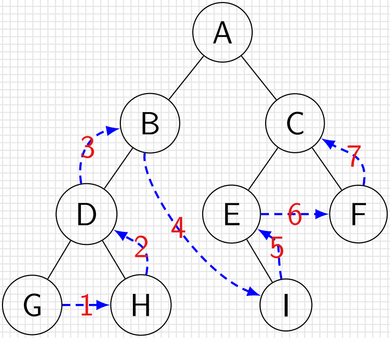
后序遍历



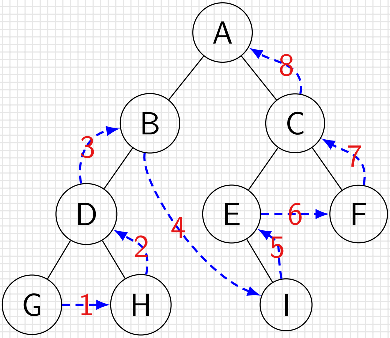
后序遍历



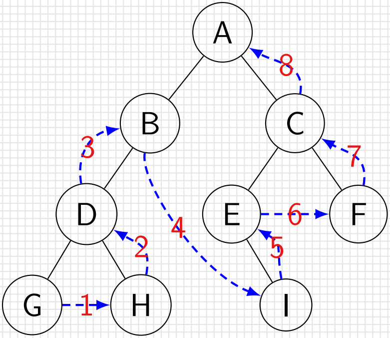
后序遍历



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);
  }
}
```

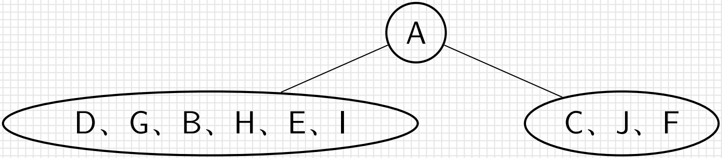
遍历二叉树及其应用

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

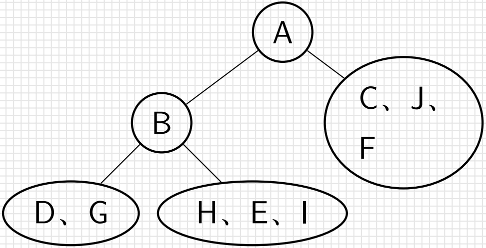
问题　已知一棵二叉树的中序遍历序列为 D、G、B、H、E、I、A、C、J、F，后序遍历序列为 G、D、H、I、E、B、J、F、C、A，请问这棵二叉树的前序遍历结果是多少？

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。



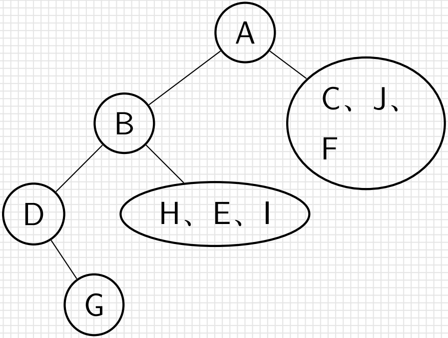
遍历二叉树及其应用

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。



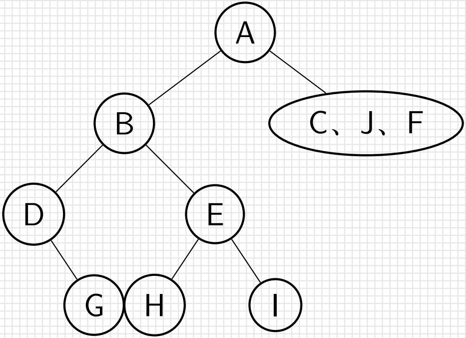
遍历二叉树及其应用

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。



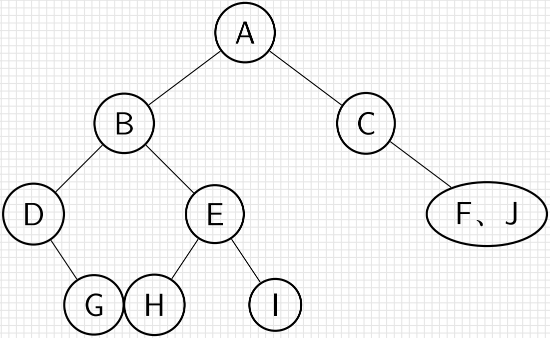
遍历二叉树及其应用

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。



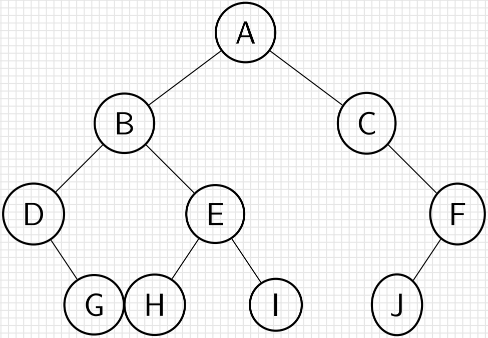
遍历二叉树及其应用

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。



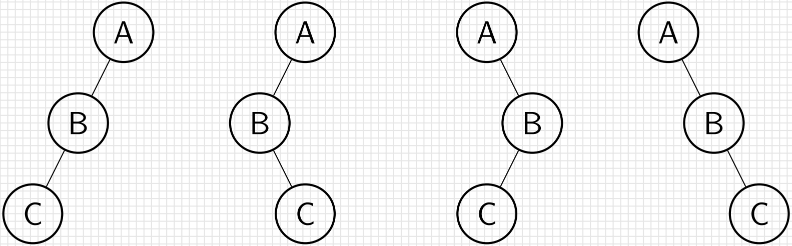
遍历二叉树及其应用

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：



树的抽象数据类型

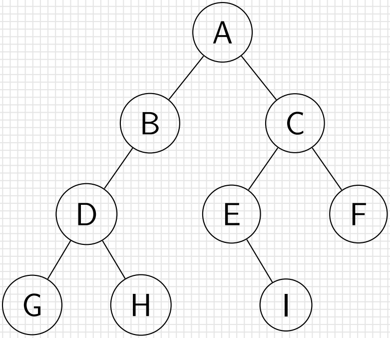
二叉树遍历的性质：

- 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。

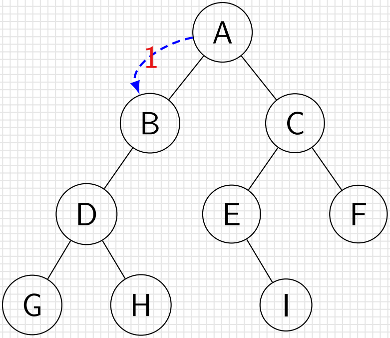
层次遍历

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。

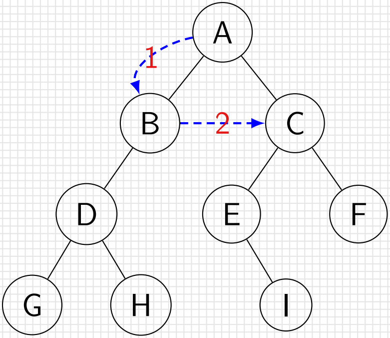
层次遍历



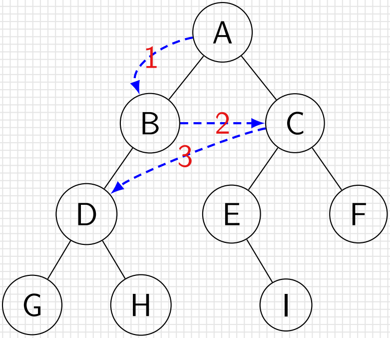
层次遍历



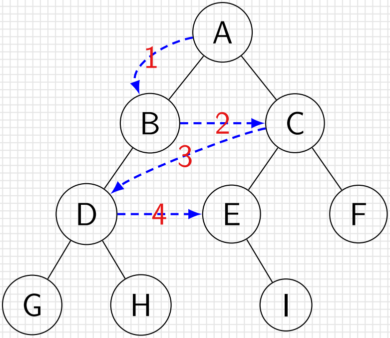
层次遍历



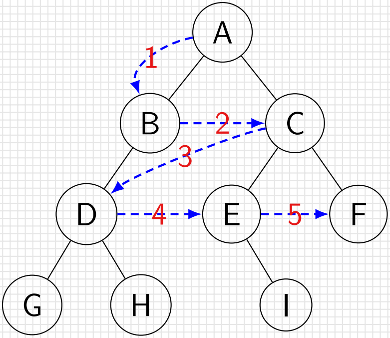
层次遍历



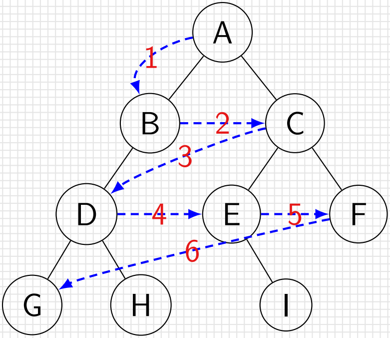
层次遍历



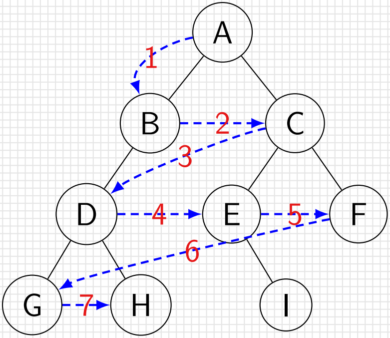
层次遍历



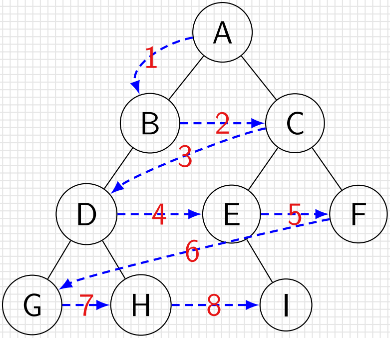
层次遍历



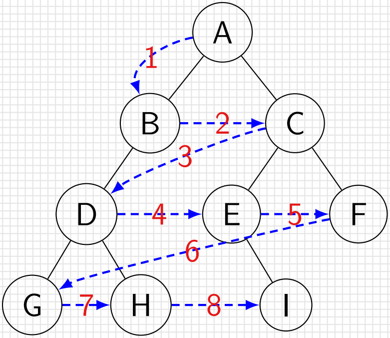
层次遍历



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;
    }
  }
}
```

1. 树的基本概念
    - 定义和基本术语
    - 树的表示形式
    - 树的抽象数据类型
2. 树的存储结构
3. 二叉树
4. 遍历二叉树及其应用
5. 二叉树相关代码

BiTree.h I

```
#include<stdio.h>
#include<stdlib.h>
#define MAX_NODE  50

typedef int ElemType;
typedef struct node {
  struct node * lchild;
  struct node * rchild;
  ElemType data;
} BTNode, * BTree;

BiTree InitBiTree(BTNode * root);
```

BiTree.h II

```
BTNode * MakeNode(ElemType data, BTNode * lchild,
BTNode * rchild);
void FreeNode(BTNode * pnode);
void ClearBiTree(BiTree tree);
void DestroyBiTree(BiTree tree);
int IsEmpty(BiTree tree);
int GetDepth(BiTree tree);
ElemType GetItem(BTNode * pnode);
void SetItem(BTNode * pnode, ElemType item);
BiTree SetLChild(BiTree parent, BiTree lchild);
BiTree SetRChild(BiTree parent, BiTree rchild);
BiTree GetLChild(BiTree tree);
BiTree GetRChild(BiTree tree);
```

BiTree.h III

```
BiTree InsertChild(BiTree parent, int lr, BiTree
child);
void DeleteChild(BiTree parent, int lr);
void PreOrderTraverse(BiTree tree, void(* visit)());
void InOrderTraverse(BiTree tree, void(* visit)());
void PostOrderTraverse(BiTree tree, void(* visit)());
void LevelOrderTraverse(BiTree tree, void(* visit)())
;
void Print(ElemType item);
```

InitBiTree.c

```
#include"BiTree.h"

/* Creat a new bitree */
BiTree InitBiTree(BTNode * root)
{
  BiTree tree = root;
  return tree;
}
```

MakeNode.c I

```c
#include"BiTree.h"

/* Generate a node */
BTNode * MakeNode(ElemType item, BTNode * lchild,
BTNode * rchild)
{
  BTNode * pnode = (BTNode *) malloc(sizeof(BTNode));
  if (pnode)
  {
    pnode->data = item;
    pnode->lchild = lchild;
    pnode->rchild = rchild;
  }
```

MakeNode.c II

```
  return pnode;
}
```

FreeNode.c

```c
#include"BiTree.h"

/* Free a node */
void FreeNode(BTNode * pnode)
{
  if(pnode != NULL)
    free(pnode);
}
```

ClearBiTree.c

```c
#include"BiTree.h"

/* Clear a bitree */
void ClearBiTree(BiTree tree)
{
  BTNode * pnode = tree;
  if (pnode->lchild != NULL)
    ClearBiTree(pnode->lchild);

  if (pnode->rchild != NULL)
    ClearBiTree(pnode->rchild);

  FreeNode(pnode);
}
```

DestroyBiTree.c

```c
#include"BiTree.h"

/* Destroy a BiTree */
void DestroyBiTree(BiTree tree)
{
  if(tree)
    ClearBiTree(tree);
}
```

IsEmpty.c

```c
#include"BiTree.h"

/* Is a BiTree Empty? */
int IsEmpty(BiTree tree)
{
  if (tree == NULL)
    return 0;
  else
    return 1;
}
```

GetDepth.c I

```
#include"BiTree.h"

/* Return a BiTree's depth */
int GetDepth(BiTree tree)
{
  int cd, ld, rd;
  cd = ld = rd = 0;
  if (tree)
  {
    ld = GetDepth(tree->lchild);
    rd = GetDepth(tree->rchild);
    cd = (ld > rd ? ld : rd);
    return cd + 1;
```

GetDepth.c II

```
    }
    return 0;
}
```

GetRoot.c

```
#include"BiTree.h"

/* Return a BiTree's root */
BiTree GetRoot(BiTree tree)
{
  return tree;
}
```

GetItem.c

```
#include"BiTree.h"

/* Return a node's value */
ElemType GetItem(BTNode * pnode)
{
  return pnode->data;
}
```

SetItem.c

```
#include"BiTree.h"

/* Set a node's value */
void SetItem(BTNode * pnode, ElemType item)
{
  pnode->data = item;
}
```

SetLChild.c

```c
#include"BiTree.h"

/* Set Left Child */
BiTree SetLChild(BiTree parent, BiTree lchild)
{
  parent->lchild = lchild;
  return lchild;
}
```

SetRChild.c

```c
#include"BiTree.h"

/* Set right Child */
BiTree SetRChild(BiTree parent, BiTree rchild)
{
  parent->rchild = rchild;
  return rchild;
}
```

GetLChild.c

```
#include"BiTree.h"

/* Return left child */
BiTree GetLChild(BiTree tree)
{
  if (tree)
    return tree->lchild;
  return NULL;
}
```

GetLChild.c

```c
#include"BiTree.h"

/* Return left child */
BiTree GetLChild(BiTree tree)
{
  if (tree)
    return tree->lchild;
  return NULL;
}
```

InsertChild.c I

```c
#include"BiTree.h"

/* Insert a new SubBiTree */
BiTree InsertChild(BiTree parent, int lr, BiTree
child)
{
  if (parent)
  {
    if (lr == 0 && parent->lchild == NULL)
    {
      parent->lchild = child;
      return child;
    }
```

InsertChild.c II

```
    if (lr == 1 && parent->rchild == NULL)
    {
      parent->rchild = child;
      return child;
    }
  }
  return NULL;
}
```

DeleteChild.c I

```
#include"BiTree.h"

/* Delete SubBiTree */
void DeleteChild(BiTree parent, int lr)
{
  if (parent)
  {
    if(lr == 0 && parent->lchild != NULL){
      parent->lchild = NULL;
      /* FreeNode(parent->lchild); */
    }

    if(lr == 1 && parent->rchild != NULL){
```

DeleteChild.c II

```
        parent->rchild = NULL;
        /* FreeNode(parent->rchild); */
      }
   }
}
```

PreOrderTraverse.c

```
#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);
  }
}
```

InOrderTraverse.c

```c
#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);
  }
}
```

PostOrderTraverse.c

```
#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);
  }
}
```

LevelOrderTraverse.c 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];
```

LevelOrderTraverse.c II

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

BiTreeTest.c I

```c
#include"BiTree.h"

int main(void){
  BTNode * n1 = MakeNode(10, NULL, NULL);
  BTNode * n2 = MakeNode(20, NULL, NULL);
  BTNode * n3 = MakeNode(30, n1, n2);
  BTNode * n4 = MakeNode(40, NULL, NULL);
  BTNode * n5 = MakeNode(50, NULL, NULL);
  BTNode * n6 = MakeNode(60, n4, n5);
  BTNode * n7 = MakeNode(70, NULL, NULL);

  BiTree tree = InitBiTree(n7);
  SetLChild(tree, n3);
```

BiTreeTest.c II

```
SetRChild(tree, n6);

printf("Depth of BiTree is %d\n", GetDepth(tree));

printf("PreOrder Traverse:\n");
PreOrderTraverse(tree, Print); printf("\n");

printf("InOrder Traverse:\n");
InOrderTraverse(tree, Print); printf("\n");

printf("PostOrder Traverse:\n");
PostOrderTraverse(tree, Print); printf("\n");

printf("LevelOrder Traverse:\n");
```

BiTreeTest.c III

```
  LevelOrderTraverse(tree, Print); printf("\n");

  SetItem(tree, 100);
  printf("Root:␣%d\n", GetItem(tree));

  DeleteChild(tree, 0);
  printf("PreOrder␣Traverse:\n");
  PreOrderTraverse(tree, Print); printf("\n");

  DestroyBiTree(tree);
  if(IsEmpty(tree))
    printf("BiTree␣is␣empty,␣succeed␣to␣destroy!\n");
}
```

运行结果

```
Depth of BiTree is 3
PreOrder Traverse:
70 30 10 20 60 40 50
InOrder Traverse:
10 30 20 70 40 60 50
PostOrder Traverse:
10 20 30
Root: 100
PreOrder Traverse:
100 60 40 50
BiTree is empty, succeed to destroy!
```