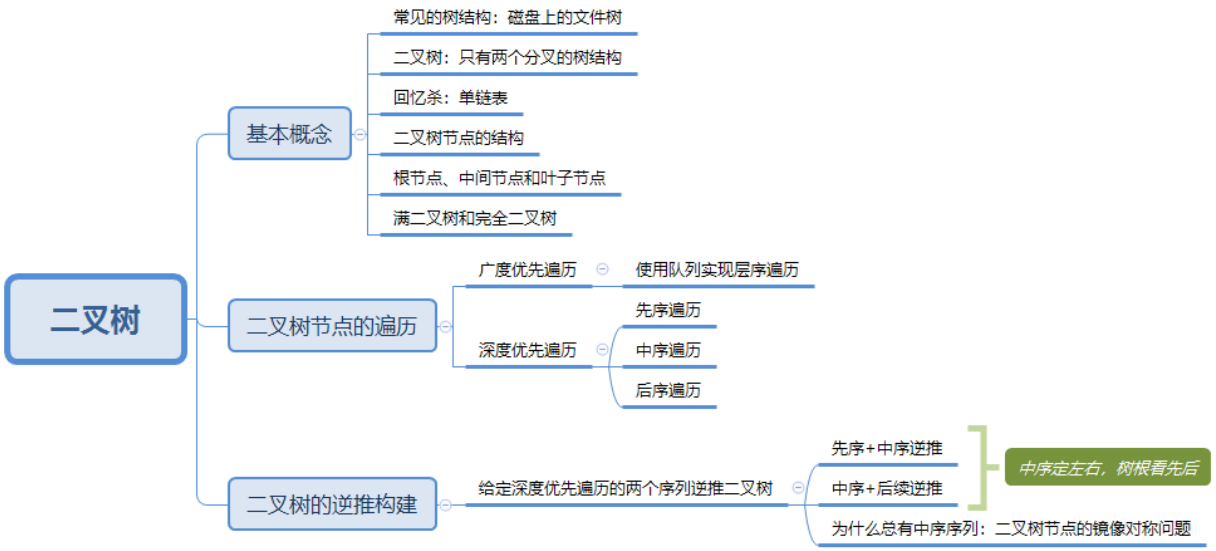


0.写在前面

二叉树结构一直以来都是数据结构课程中的重点和难点。不论是找工作的笔试面试，还是考研的专业课，二叉树所占的比例都是很大的

而在原始的二叉树的基础上，有不断演化出了很多其他基于二叉树的结构，例如本教程涉及的哈夫曼树、红黑树，还有线索二叉树、B+树等等

但是不管从二叉树衍生出来的结构多么复杂多变，但是底层对于二叉树结构的理论和操作都是相通的

所以我们在这一章节中，从最基本的原生二叉树开始，不断进行总结和实践，最终达到理解和掌握原生二叉树和一些比较有代表性的二叉树变种的目的。

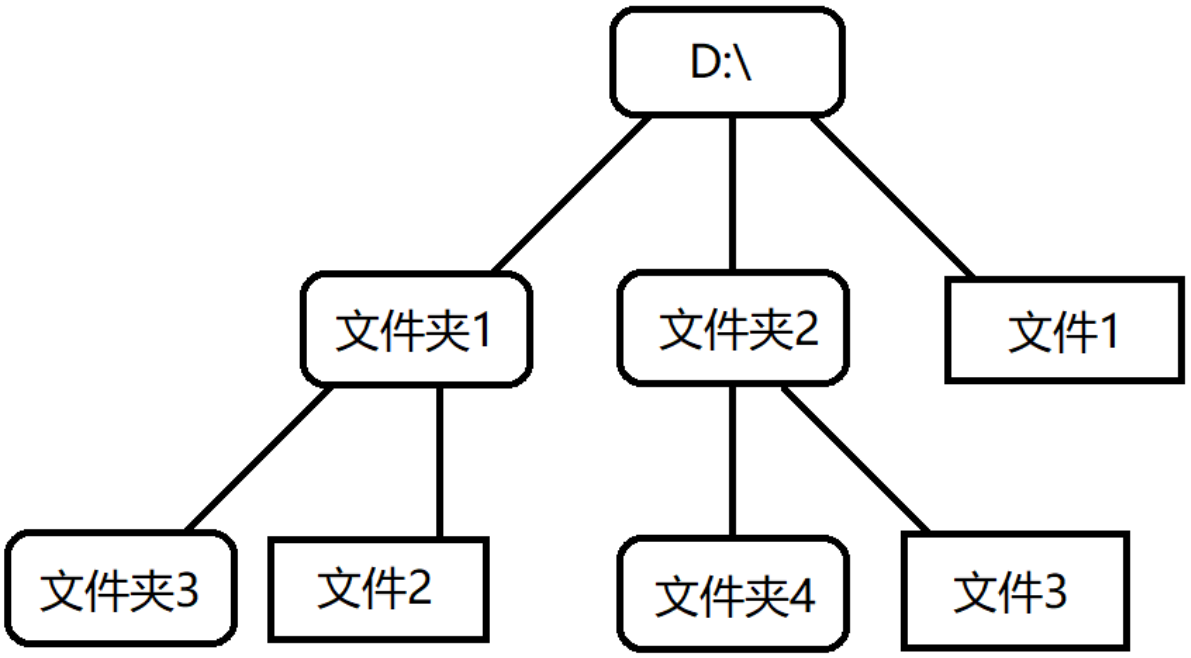
1.二叉树的基本概念

①常见的树结构：磁盘上的文件树

实际上在现实生活中，树状结构是十分普遍的。例如在我们的计算机磁盘上存储的文件夹和文件

就能够构成一个文件树结构。其中盘符下存储有文件和文件夹，文件夹下又有子文件和子文件夹，但是文件之下一定不会有其他子文件和子文件夹，

那么将这些结构画成一张图，我们就能够看出他的结构



通过上面这张图我们可以看出来：

1.通过盘符或者文件夹，我们只能够向下访问其中的子文件和子文件夹，但是我们不能够从文件夹1直接跳转到文件夹2中去

正如我们不会在D盘中直接访问到C盘是一样的道理

2.只有盘符或者文件夹下面才能够下挂子文件和子文件夹，但是文件之下是不能够下挂其他任何结构的

3.每一个盘符和文件夹之下都有多个分支，这些分支“开枝散叶”，形成了复杂的分支结构

而这种分支结构在向上看的时候，最终都会回归到盘符这个层面，所以上面这张图，正如同一棵倒置的大树，其树根在上

我们将这种由节点和分支构成，并且节点之间只能够上线联系，不能够左右沟通的结构，称之为树结构

在上图中： 我们称所有的盘符、文件夹和文件为树结构的节点

而连接节点与节点之间的通路，称之为路径

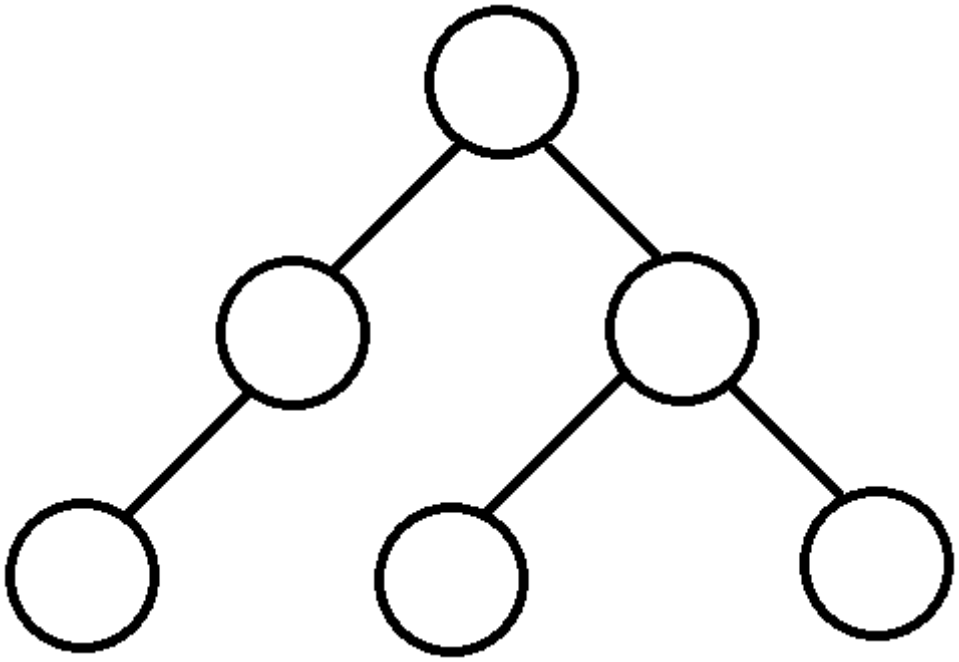
如果一个节点指向另一个节点（例如文件夹1和文件2的关系），那么我们称上面指出的节点为父节点或者双亲节点（文件夹1），

下面被指向的节点为父节点的孩子节点（文件2）。其中，孩子节点简称子节点

②二叉树：只有两个分叉的树结构

上面我们已经了解了什么是树结构。那么，如果在一棵树中，所有的节点都最多只有两个分支，那么这种特殊的树结构，我们称之为二叉树结构

二叉树形如下图所示：



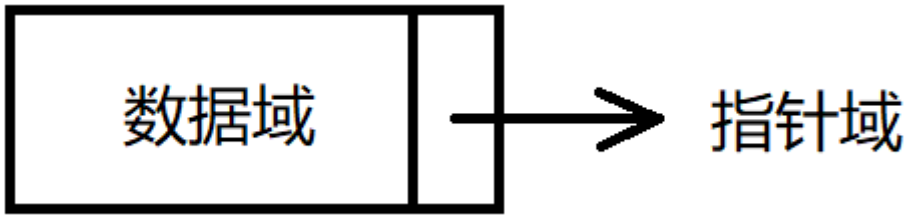
也就是说，在二叉树中，一个节点要么不具有子节点，如果具有子节点的话，最多只能有两个子节点

下面我们开始讨论二叉树中节点在代码中的定义方式

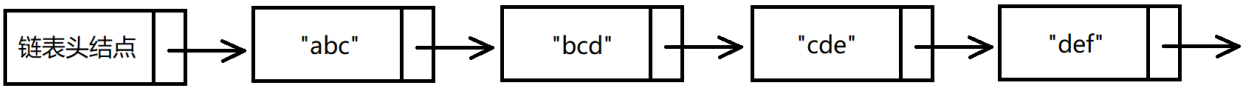
③回忆杀：单链表

首先我们先来回忆一下单链表结构。

在单链表结构中，同样存在节点的概念，我们使用节点存储数据和下一个节点的内存地址，单链表节点如下图所示：



然后将这样的节点串联起来，我们就得到了单链表结构：

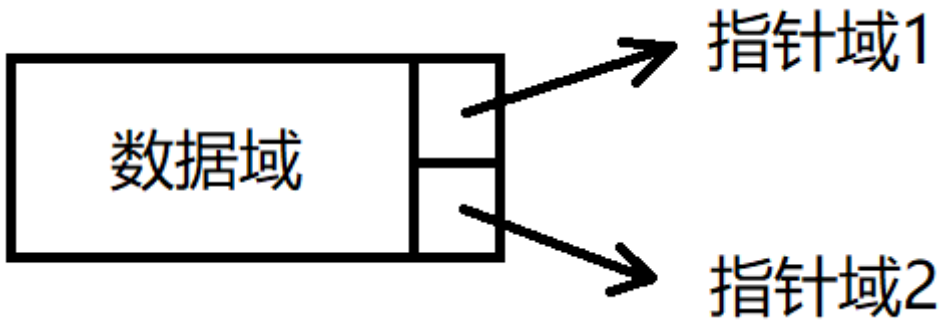


单链表节点的代码定义如下：

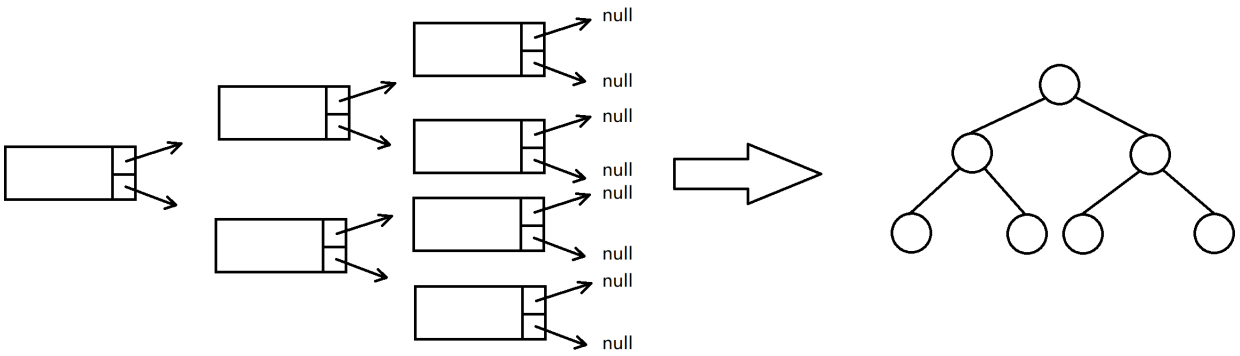
```
public class Node {
    Object data;  //数据域
    Node next;  //指针域
}
```

④二叉树节点的结构

如果我们将上述节点的结构稍加变化，让每一个节点都具有两个指针域，会得到一个什么样的结构呢?



然后我们让其中的每一个指针都指向另一个节点，于是乎我们得到如下图所示的一种结构：



不难看出，上面的两张图之间是等价的。所以同样的，只要我们在单链表节点结构的基础上稍加改造，就能够得到二叉树节点的代码：

习惯性的，我们将一个父节点左边的子节点称之为左孩子节点，右边的子节点称之为右孩子节点

```
class Node {

    Object date;  //数据域

    Node leftChild;  //左孩子节点
    Node rightChild;  //右孩子节点

}
```

⑤根节点、中间节点和叶子节点

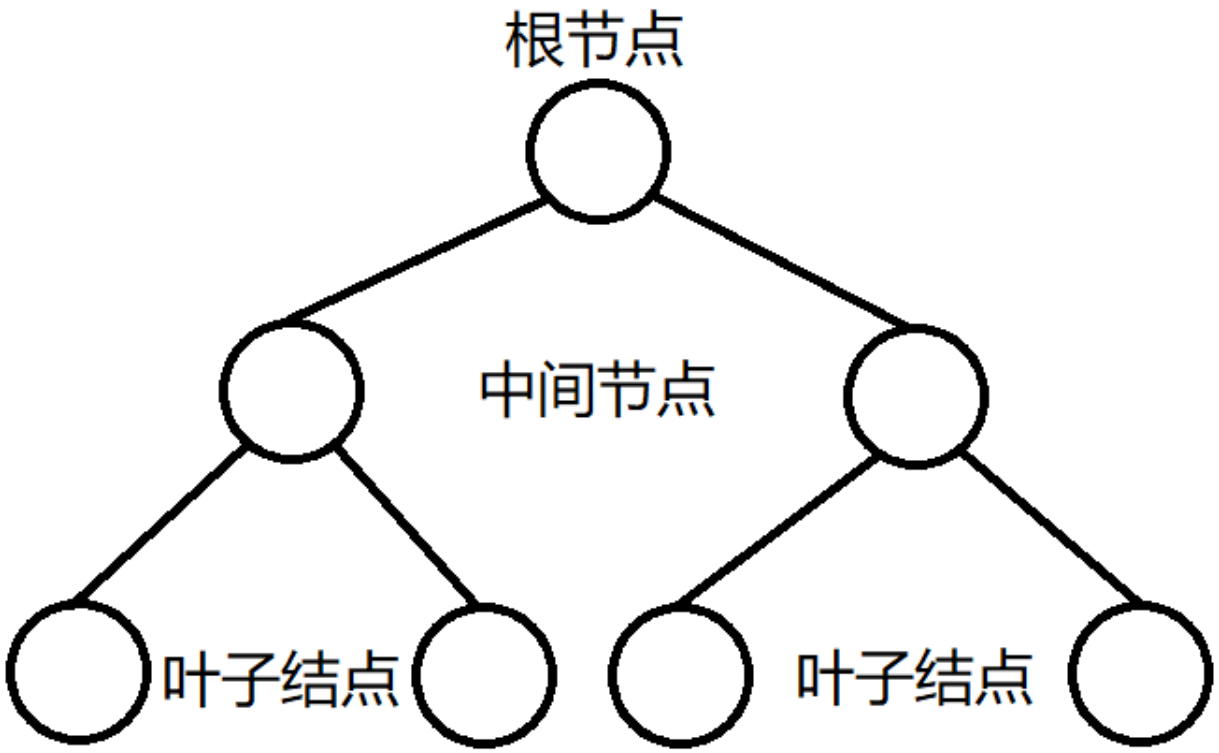
在学习二叉树结构的时候，为了方便讨论，我们将二叉树结构中的节点类型进行如下分类：

根节点：二叉树最上面的节点，特征是只有子节点，没有父节点。如果我们能够得到一个二叉树结构的根节点，那么相当于得到了整个二叉树结构

中间节点：如果一个节点既有父节点，又有子节点，那么我们称这种节点为中间节点

叶子节点：如果一个节点只有父节点，没有任何子节点，那么我们称这种节点为叶子结点

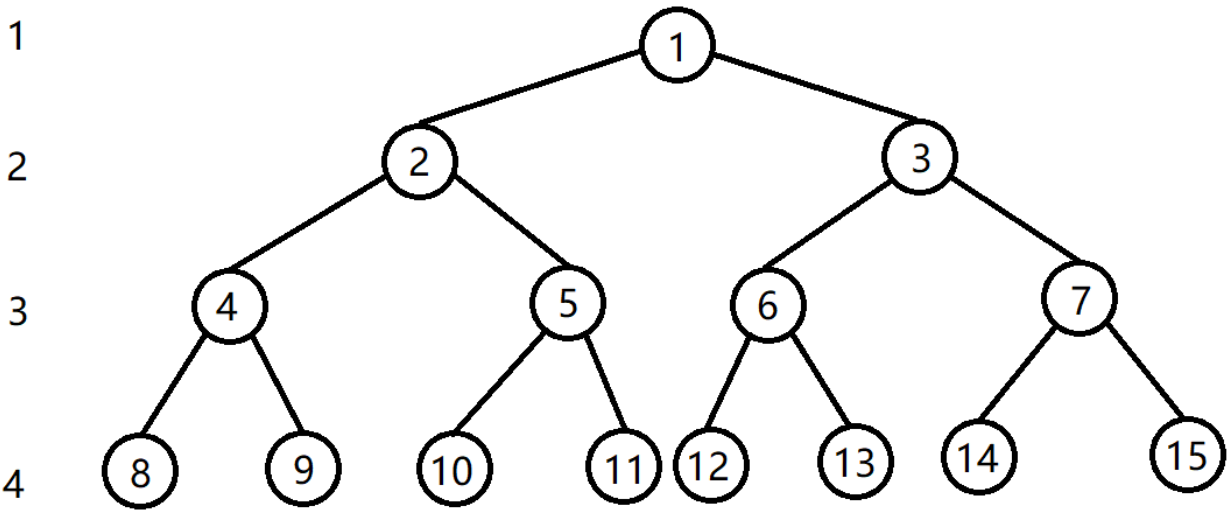
上述三种节点类型，如图所示：



⑥满二叉树和完全二叉树

1.满二叉树的定义和性质

如果在一个二叉树结构中，除了最后一层的叶子结点之外，其余每一个节点都必须有两个子节点，那么这种二叉树称之为满二叉树



在一个满二叉树中存在如下特性：

1.满二叉树的节点数量一定是一个奇数，因为从第2层开始，每一层的节点数量都是2的整数倍，所以最后加上根节点，那么节点的数量一定是奇数

2.一个具有k层的满二叉树，其节点总数为2^k-1个。这个结论很好推理，实际上就是一个公比为2的等比数列和

3.第i层上面的节点数量为2^{i-1}个

4.一个k层的满二叉树，其叶子结点数量（也就是最后一层的节点数量）为2^{k-1}个

5.如果按照从上到下、从左到右的方式为满二叉树的每一个节点从1开始进行编号，那么满二叉树第k层中的最大编号取值为2^k-1

6.满二叉树中，编号为m的节点和其左右孩子节点的编号关系是：

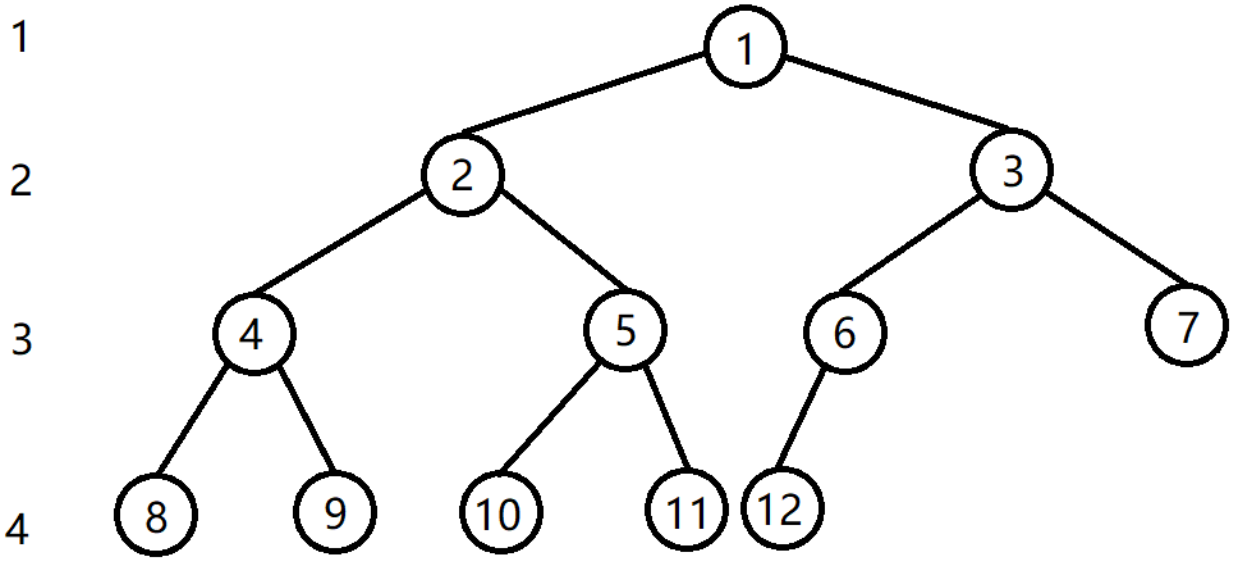
左孩子编号 = 2m

右孩子编号 = 2m+1

实际上我们在讲解堆排序的时候就推导过类似的关系，只不过在堆排序中，节点的编号是从0开始的

2.完全二叉树的定义和性质

在一个二叉树结构中，如果只有最后一层和倒数第2层存在叶子结点，那么这个二叉树就是一个完全二叉树



完全二叉树的许多特性和满二叉树是相似的，其余的一些特殊特性，我们会在下一节内容中进行讨论

3.其他概念和公式的补充

节点的度：一个节点的度可以简单的理解为一个节点孩子节点的数量

在许多面试题中，都使用n_0、n_1、n_2分别表示度为0、度为1、度为2的节点的数量，即有0个孩子节点、1个孩子节点、2个孩子节点的节点数量

在任意一棵二叉树中，这些节点的数量，之间存在如下关系

1.$n_2 = n_0 - 1$：即度为2的节点总比度为0的节点少1个，也就是说，在任意二叉树中，同时具有左右孩子的节点，总比叶子结点的数量少一个

2.从上面的公式我们能够推导出来：$n_{总} = n_0 + n_1 + n_2 = 2n_0 + n_1 - 1 = n_1 + 2n_2 + 1$，且$n_1 = n_{总} - n_0 - n_2 = n_{总} - 2n0 + 1 = n_{总} - 2n_2 - 1$

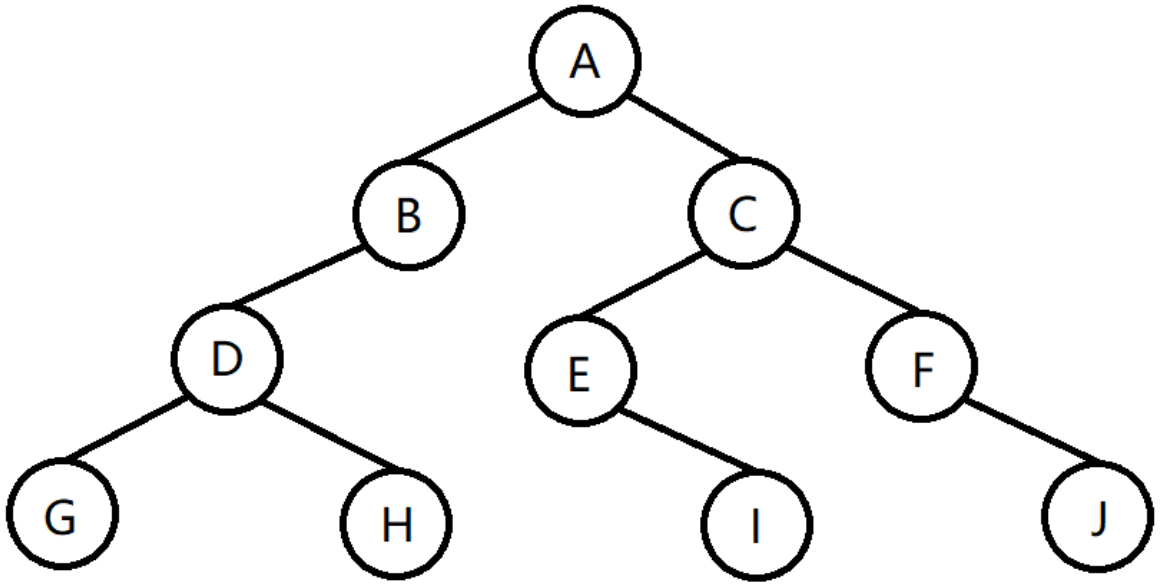
所以，只要我们知道一个二叉树中总的节点个数以及度为0或者度为2的节点数量，就能够推导出其他度的节点的数量

注意：上面的公式常常出现在各种笔试题当中，所以请各位熟练记忆和掌握上述公式以及各种变形公式及其推导过程

2.二叉树节点的遍历

为了方便我们的讨论，在学习二叉树节点的遍历方式之前，我们首先约定一个二叉树的结构，并以此为蓝本，讨论二叉树的各种遍历方式

二叉树结构如下：



并且在代码中给出这个二叉树结构的定义如下：

```
//[1]先把所有的节点定义出来
Node A = new Node("A");
Node B = new Node("B");
Node C = new Node("C");
Node D = new Node("D");
Node E = new Node("E");
Node F = new Node("F");
Node G = new Node("G");
Node H = new Node("H");
Node I = new Node("I");
Node J = new Node("J");

//[2]根据图示挂载每一个节点的左右孩子节点
A.leftChild = B;
A.rightChild = C;
B.leftChild = D;
D.leftChild = G;
D.rightChild = H;
C.leftChild = E;
C.rightChild = F;
E.rightChild = I;
F.rightChild = J;
```

注意：上面的二叉树结构代码属于硬编码，但是在真正开发过程中，我们应该学会灵活创建和使用二叉树结构

二叉树的各种方式的遍历，是一个既简单又麻烦的问题。简单在于，找到规律后我们会发现：如何遍历整个二叉树结构，就如何遍历二叉树的子树结构、

这个过程直接使用递归结构就能够完成；麻烦的是，我们需要找到规律……

下面就然我们一起来尝试使用各种方式遍历二叉树结构

①广度优先遍历

对一棵二叉树进行广度优先遍历，就是对一棵二叉树中所有的节点，按照层次从上到下、每一层中节点从左到右的顺序，将二叉树中的所有节点进行遍历的操作

广度优先遍历方式只有一种，那就是层序遍历，而二叉树的层序遍历操作，需要依赖于队列结构

1.使用队列实现层序遍历

使用队列对二叉树进行层序遍历的步骤如下：

步骤1：将二叉树的根节点加入队列

步骤2：将队列头节点出队列

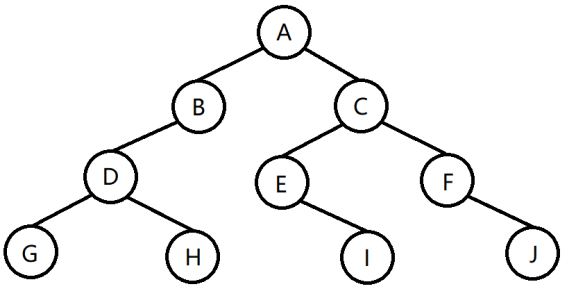
步骤3.1：如果根节点具有左孩子，则根节点的左孩子入队列

步骤3.2：如果根节点具有右孩子，则根节点的右孩子入队列

然后重复上述的步骤2至步骤3.2，直至整个队列中不再具有元素为止

注意：在步骤3.1至步骤3.2中，一定是左孩子先进入队列，然后才是右孩子进入队列

这个流程的图示如下：



(1) A

(2) B C

A

(3) C D

A B

(4) D E F

A B C

(5) E F G H

A B C D

(6) F G H I

A B C D E

(7) G H I J

A B C D E F

(8) H I J

A B C D E F G

(9) I J

A B C D E F G H

(10) J

A B C D E F G H I

(11)

A B C D E F G H I J

实际上广度优先遍历是一种比较简单的操作方式，主要是利用了队列先进先出的数据结构特点

在从二叉树顶端的树根开始加入队列的时候开始，逐层向下，利用二叉树只能就行二分的规则，不断的将子节点按照从左到右的顺序进行加入

最终在队列清空的情况下，得到完成的二叉树自上到下、从左到右的节点遍历顺序

代码实现如下：

```
/**
 * 层序遍历
 * @param root 整个二叉树的根节点
 */
public void levelOrderTraversal(Node root) {

    //[1]创建一个队列结构，用来保存二叉树中的节点，
    //同时也使用这个队列结构来确定层序遍历的节点出入队列顺序
    LinkedList<Node> nodeList = new LinkedList<>();

    //[2]首先将整个二叉树的根节点入队列
    nodeList.offer(root);

    //[3]创建一个循环，开始对二叉树进行层序遍历
    //如果队列中不在具有节点，说明整个二叉树遍历完成
```

```
    while(!nodeList.isEmpty()) {

        //[4]首先将队列头节点出队列
        Node node = nodeList.poll();
        System.out.print(node.data);

        //[5.1]如果队列头结点左孩子不是空，则将左孩子入队列
        if(node.leftChild != null) {
            nodeList.offer(node.leftChild);
        }

        //[5.2]如果队列头节点右孩子不是空，则将右孩子入队列
        if(node.rightChild != null) {
            nodeList.offer(node.rightChild);
        }

    }

}
```

②深度优先遍历

深度优先遍历是二叉树遍历方式中比较重点的一种遍历方式，可以分为先序遍历、中序遍历和后序遍历3种遍历序列

和广度优先遍历方式不同的是，深度优先遍历是从根节点开始，一条路走到黑，一个分支一个分支的进行节点遍历

其中一个分支全部遍历结束之后，再去进行下一个分支的遍历

在深度优先遍历中，我们必须记住这样一个规则：如何遍历整个二叉树，就如何遍历左右子树

带着这样的思想，我们开始讨论如何对二叉树结构进行深度优先遍历

1.先序遍历

先序遍历处理节点的顺序是：根左右，即先访问根节点，然后访问左子树，最后访问右子树

先序遍历二叉树节点的步骤是：

步骤1：首先得到整个二叉树的根节点，并访问这个根节点

步骤2：如果根节点具有左孩子，则将左孩子及其子节点作为一个单独的二叉树进行先序遍历

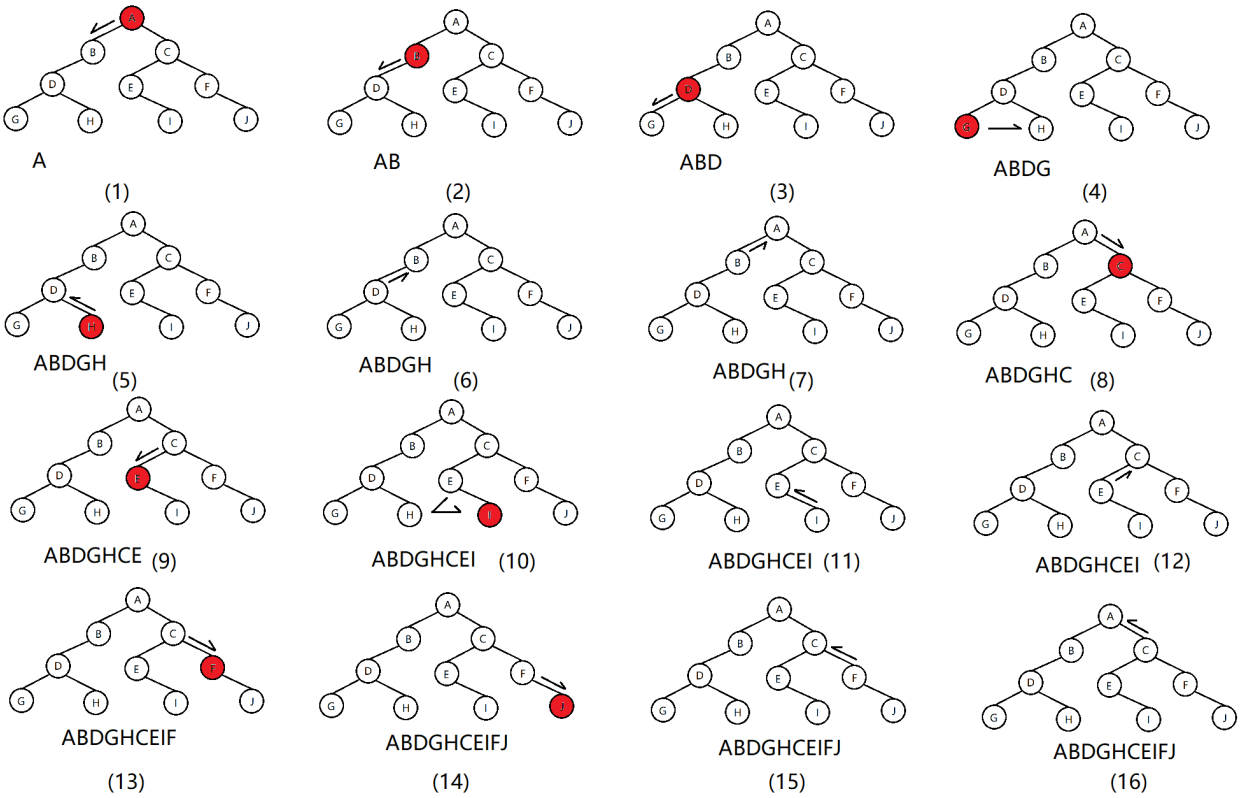
步骤3：在遍历过程中，如果一个节点没有左孩子或者左孩子已经遍历完成，那么判断这个节点有没有右孩子

步骤4：如果这个节点具有右孩子，则将右孩子及其子节点作为一个单独的二叉树进行先序遍历

步骤5：如果这个节点的没有右孩子或者右孩子已经遍历完成，那么向上回溯到父节点，重复判断父节点是否具有右孩子

步骤6：重复上述步2-5，直到最终回溯到根节点为止，表示根节点及其左右子树已经全部遍历完成，得到完整的先序遍历序列

步骤如下图所示：



之前我们说过：如何对一个完整的二叉树进行遍历，那么就如何对其中的子树进行遍历

所以，在代码中我们可以通过递归的方式来实现上面过程的代码：

```
/**
 * 先序遍历
 * @param root 整个二叉树或者子树的根节点
 */
public void preOrderTraversal(Node root) {

    //[1]访问根节点
    System.out.print(root.data);

    //[2]如果左孩子不为空，则先序遍历左子树
```

```
    if(root.leftChild != null) {
        preOrderTraversal(root.leftChild);
    }

    //[3]如果右孩子不为空，则先序遍历右子树
    if(root.rightChild != null) {
        preOrderTraversal(root.rightChild);
    }

}
```

2.中序遍历

中序遍历处理节点的顺序是：左根右，即先访问左子树，然后访问根节点，最后访问右子树

中序遍历二叉树节点的步骤是：

步骤1：在得到根节点的时候，首先不要对根节点进行访问，而是直接判断这个根节点有没有左孩子

步骤2：如果根节点存在左孩子，则对以左孩子为根的左子树进行中序遍历

步骤3：如果一个节点不存在左孩子，或者左孩子已经遍历完成，那么项父节点进行回溯，在回溯到父节点的时候，访问父节点

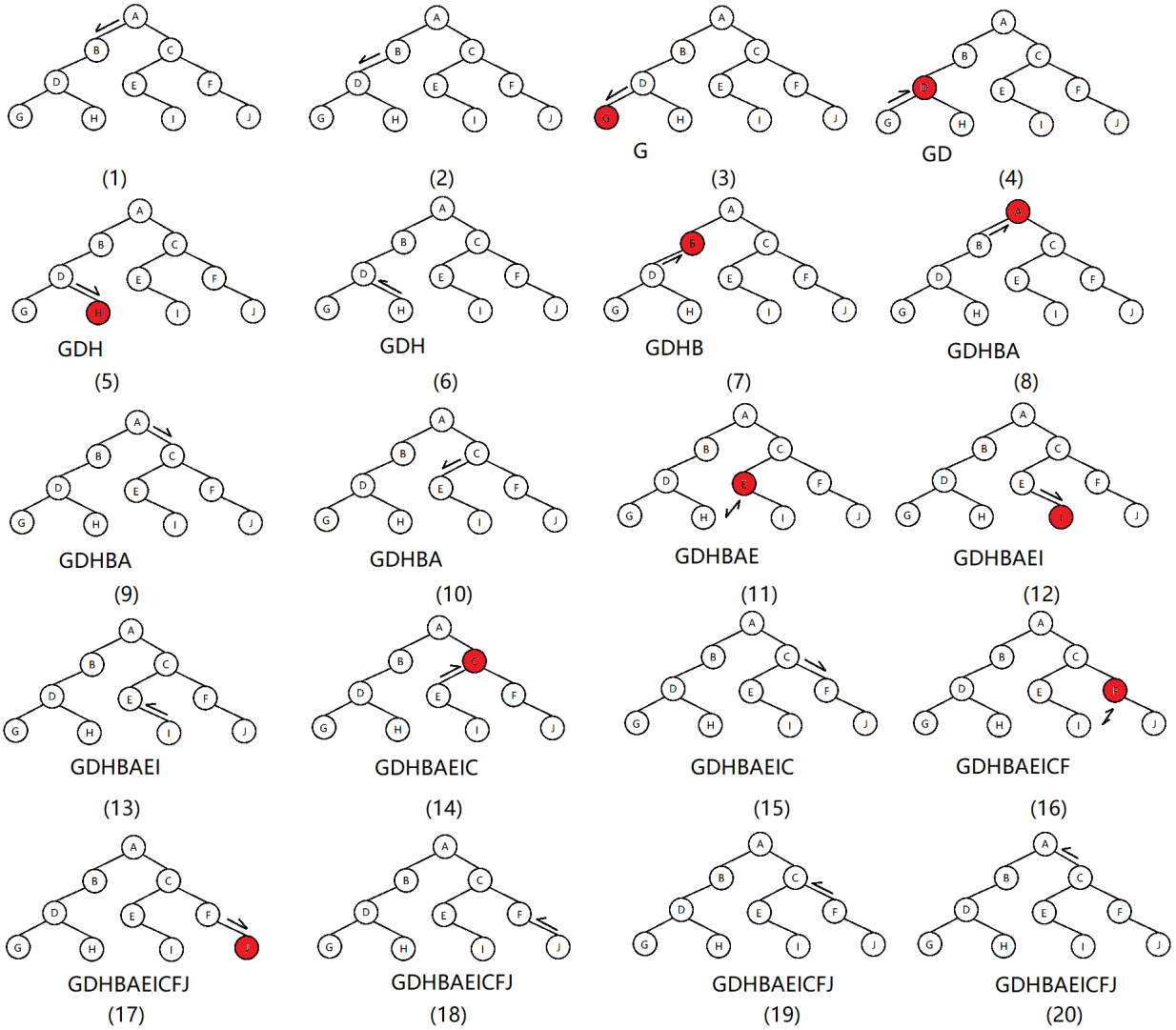
步骤4：在父节点访问完成后，判断父节点是否具有右孩子

步骤5：如果父节点具有右孩子，则对右孩子为根的右子树进行中序遍历

步骤6：如果父节点不具有右孩子或者右子树已经遍历完成，那么向上回溯到父节点

步骤7：重复上述步骤1-6，直到回溯到整个二叉树的根节点为止，遍历完成，得到完整的中序遍历序列

步骤如下图所示：



代码实现：

```
/**
 * 中序遍历
 * @param root 整个二叉树或者子树的根节点
 */
public void inOrderTraversal(Node root) {

    //[1]如果左孩子不为空，则中序遍历左子树
    if(root.leftChild != null) {
        inOrderTraversal(root.leftChild);
    }

    //[2]访问根节点
    System.out.print(root.data);

    //[3]如果右孩子不为空，则中序遍历右子树
    if(root.rightChild != null) {
        inOrderTraversal(root.rightChild);
```

```
18      }
19
20  }
```

3.后序遍历

后序遍历处理节点的顺序是：左右根，即先访问左子树，然后访问右子树，最后访问根节点

后序遍历二叉树节点的步骤是：

步骤1：得到整个二叉树的根节点，但是并不访问根节点，判断根节点是否具有左孩子

步骤2：如果根节点具有左孩子，则后序遍历以左孩子为根的左子树

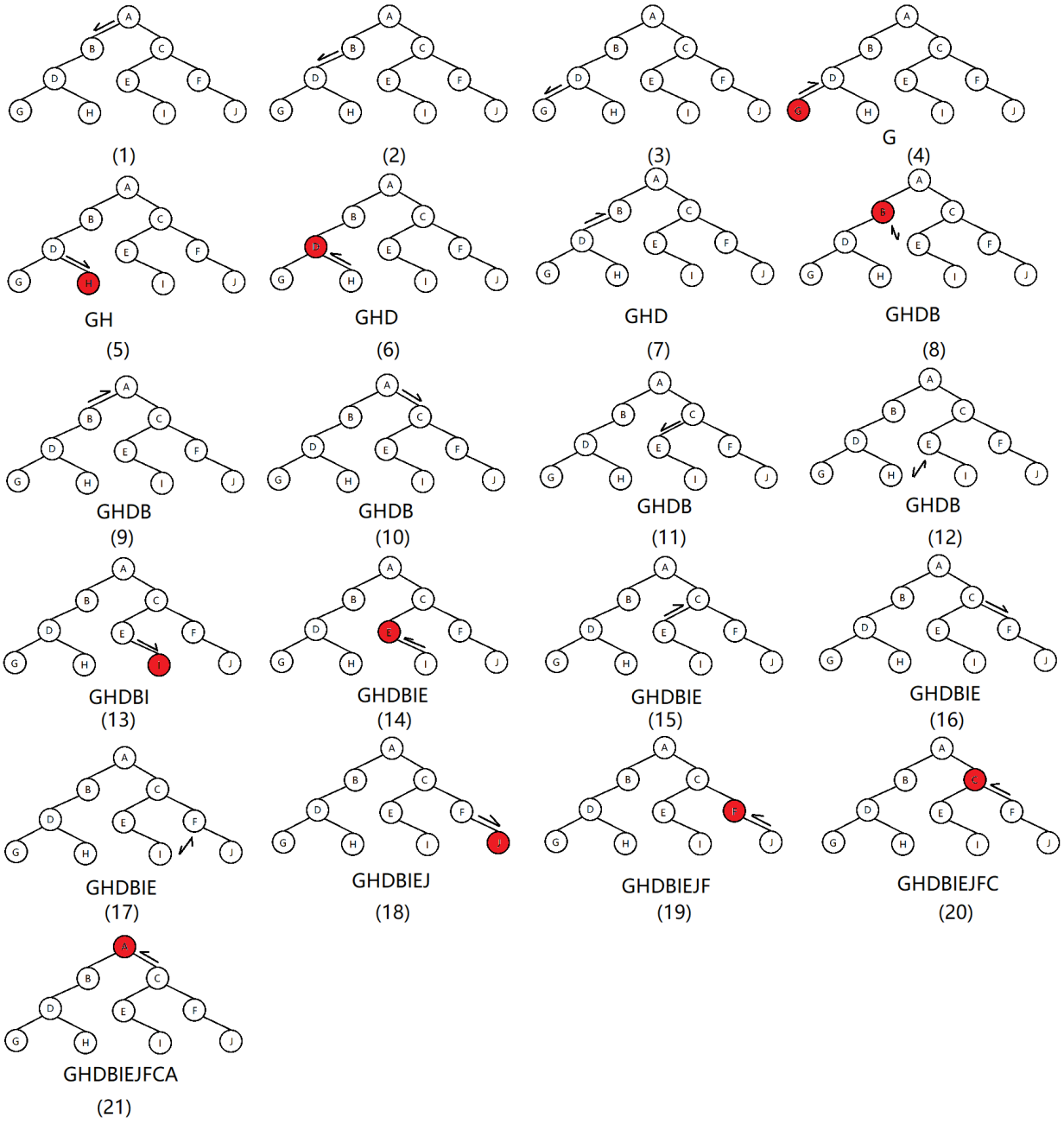
步骤3：如果根节点不具有左孩子，或者左子树遍历完成，则回溯到根节点，但是依然不访问根节点，而是判断根节点是否具有右孩子

步骤4：如果根节点具有右孩子，则后序遍历以右孩子为根的右子树

步骤5：如果根节点不具有右孩子，或者右子树遍历完成，则回溯到根节点，此时访问根节点

步骤6：重复上述步骤1-5，直到回溯到整个二叉树的根节点并且根节点访问完毕为止

步骤如下图所示：



代码实现：

```
/**
 * 后序遍历
 * @param root 整个二叉树或者子树的根节点
 */
public void postOrderTraversal(Node root) {

    //[1]如果左孩子不为空，则后序遍历左子树
    if(root.leftChild != null) {
        postOrderTraversal(root.leftChild);
    }

    //[2]如果右孩子不为空，则后序遍历右子树
    if(root.rightChild != null) {
```

```
        postOrderTraversal(root.rightChild);
    }

    //[3]访问根节点
    System.out.print(root.data);

}
```

4.规律总结

从上面进行的先序、中序、后序三种遍历操作中我们不难得出如下一些规律，这些规律在后序的通过遍历序列反推二叉树结构的过程中会使用到

规律1：在先序序列中，最先出现的节点一定是整棵树的根节点

规律2：在后序序列中，最后出现的节点一定是整棵树的根节点

规律3：在中序序列中，整棵树的根节点出现在序列的中间

规律4：在中序序列中，根节点左边的节点都是左子树的构成节点；根节点右边的节点都是右子树的构成节点

3.二叉树的逆推构建

在经典二叉树问题中，还有一类题目，那就是给定深度优先遍历序列三种顺序中的两种，反推一棵二叉树的结构图

实际上这种问题具有很强的技巧性，只要我们记住这一技巧，就能够很容易的推断出整个二叉树的结构

请记住如下规律：中序定左右，树根看先后

至于上述规律的含义，我们会在下面的具体问题中进行分析

但是需要注意的是：在给定的深度优先遍历序列的两种序列中，只有包含中序遍历序列的情况下，才能够唯一确定一个二叉树的结构

那么如果我们仅给定先序序列和后序序列，为什么不能够唯一确定一个二叉树的结构呢？我们将在后面进行说明

①先序序列+中序序列的逆推过程

通过先序序列和中序序列逆推二叉树的步骤如下：

步骤1：在先序序列中最先出现的节点就是整个二叉树的根节点

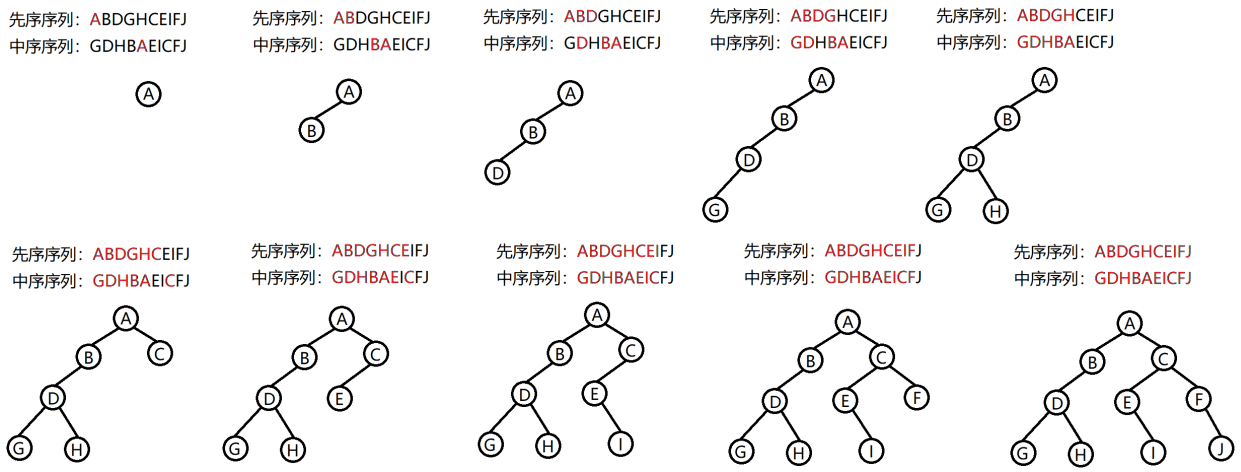
步骤2：在中序序列中找到根节点的位置，根节点左边的所有节点就是树根左子树的构成，根节点右边的所有节点就是树根右子树的构成

步骤3：使用中序序列的左子树部分，在先序序列中进行查找，在先序序列中最先出现的节点就是左子树的树根

步骤4：使用中序序列的右子树部分，在先序序列中进行查找，在先序序列中最先出现的节点就是右子树的树根

步骤5：在确定左右子树的树根节点之后，重复上述步骤1-5，直到先序序列和中序序列中的所有元素全部使用过为止，二叉树构建完成

通过先序序列和中序序列构建二叉树过程如图所示：



②后序序列+中序序列的逆推过程

通过后序序列和中序序列逆推二叉树结构的方式和前面的通过先序序列和中序序列逆推二叉树结构的过程基本相似

只是现在确定子树树根节点的方式变成了：在后序序列中，最后出现的节点才是整个二叉树或者左右子树的树根节点

通过后序序列和中序序列逆推二叉树的步骤如下：

步骤1：在后序序列中最后出现的节点是整个二叉树结构的根节点

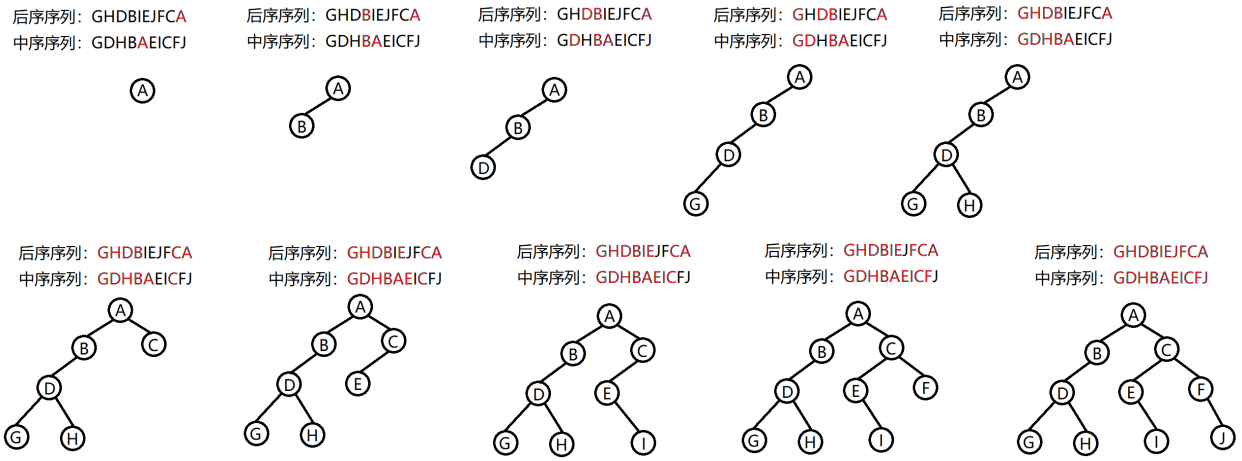
步骤2：在中序序列中找到根节点的位置，根节点左边的所有节点就是树根左子树的构成，根节点右边的所有节点就是树根右子树的构成

步骤3：使用中序序列的左子树部分，在后序序列中进行查找，在后序序列中最后出现的节点就是左子树的树根

步骤4：使用中序序列的右子树部分，在后序序列中进行查找，在后序序列中最后出现的节点就是右子树的树根

在确定左右子树的树根节点之后，重复上述步骤1-5，直到后序序列和中序序列中的所有元素全部使用过为止，二叉树构建完成

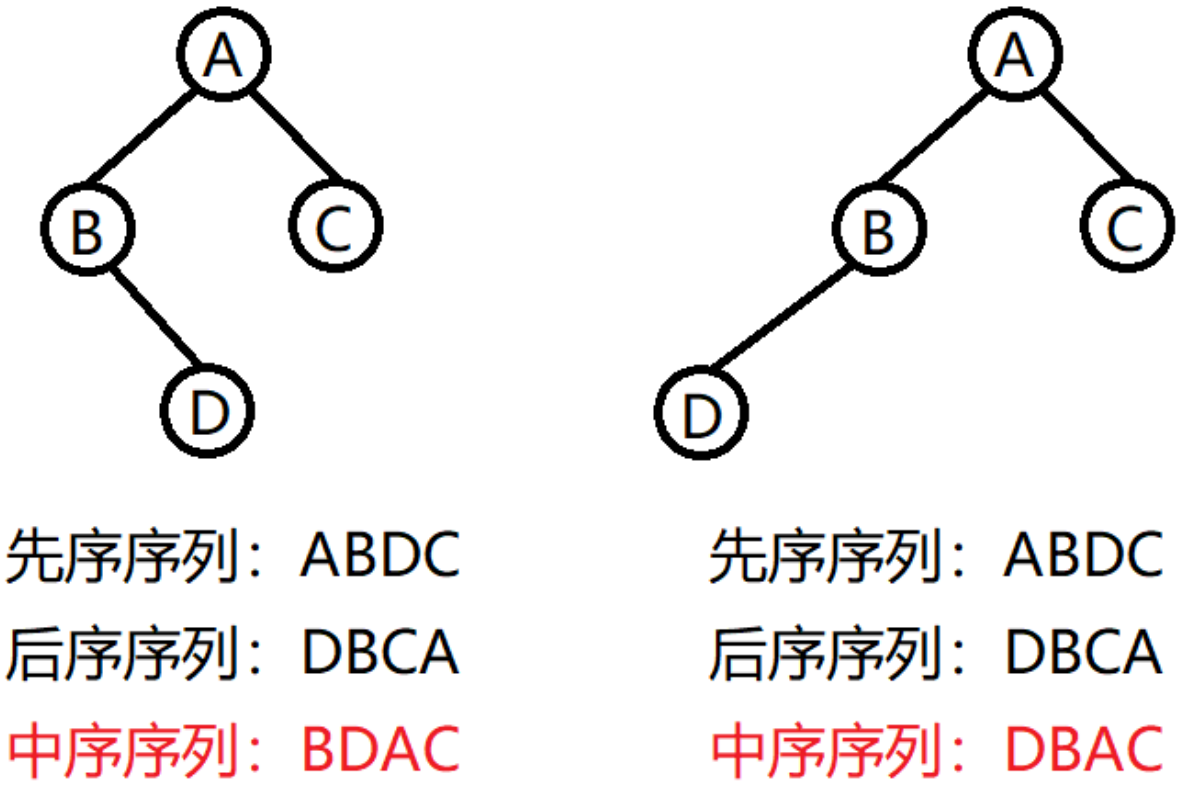
通过后序序列和中序序列构建二叉树过程如图所示：



③二叉树的镜像对称问题

为什么说在之给定先序序列和后序序列的情况下不能够唯一确定一个二叉树的结构呢？

下面我们先来观察如下的两个二叉树结构，并对这两个二叉树结构分别推断先序、中序、后序遍历序列：



从上图中我们不难看出，两个二叉树的结构是不同的，但是两个二叉树的先序序列和后序序列是完全相同的，只有中序序列不同

我们称图中两个二叉树结构下BD两个节点的结构为镜像对称

所以，如果在两个二叉树结构中存在这种进行对称的结构，是不能够通过先序序列和后序序列进行区分的，只能够再通过一个中序序列加以区分