

## Lesson5--二叉树

### 【本节目标】

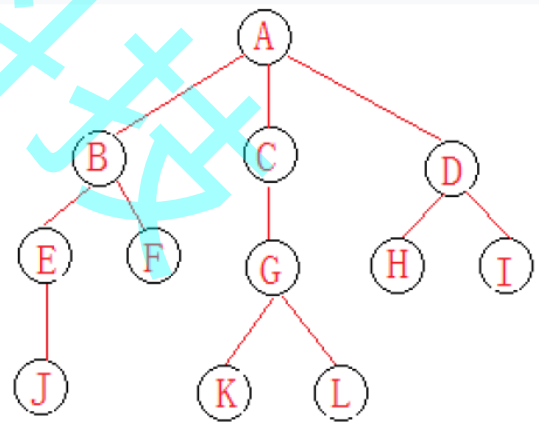
- 1.树概念及结构
- 2.二叉树概念及结构
- 3.二叉树顺序结构及实现
- 4.二叉树链式结构及实现

### 1.树概念及结构

#### 1.1树的概念

树是一种**非线性**的数据结构，它是由 $n$  ( $n \geq 0$ ) 个有限结点组成一个具有层次关系的集合。把它叫做树是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。

- 有一个特殊的结点，称为根结点，根节点没有前驱结点
- 除根节点外，其余结点被分成 $M$  ( $M > 0$ ) 个互不相交的集合 $T_1$ 、 $T_2$ 、.....、 $T_m$ ，其中每一个集合 $T_i$  ( $1 \leq i \leq m$ ) 又是一棵结构与树类似的子树。每棵子树的根结点有且只有一个前驱，可以有0个或多个后继
- 因此，树是递归定义的。

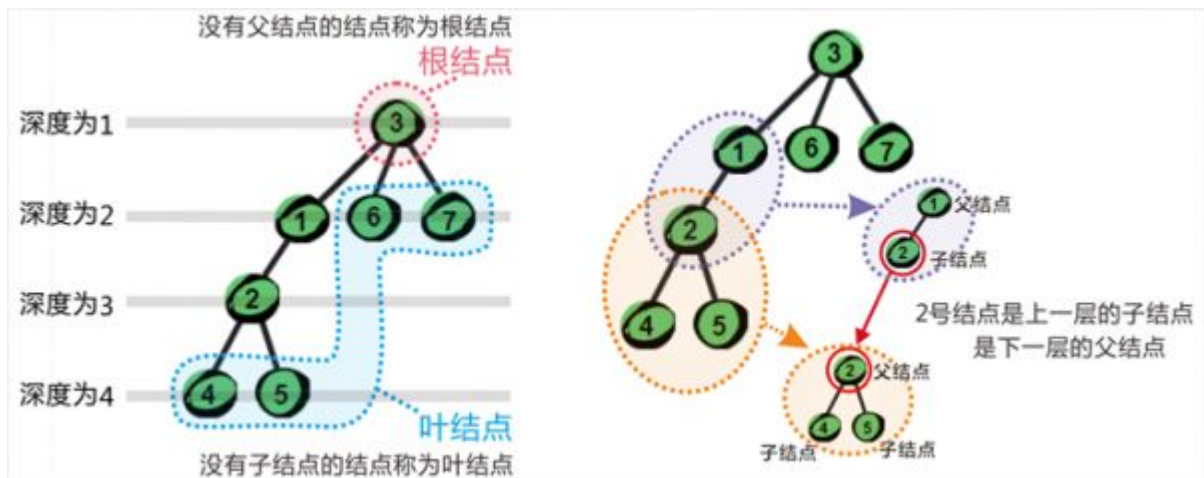


空树

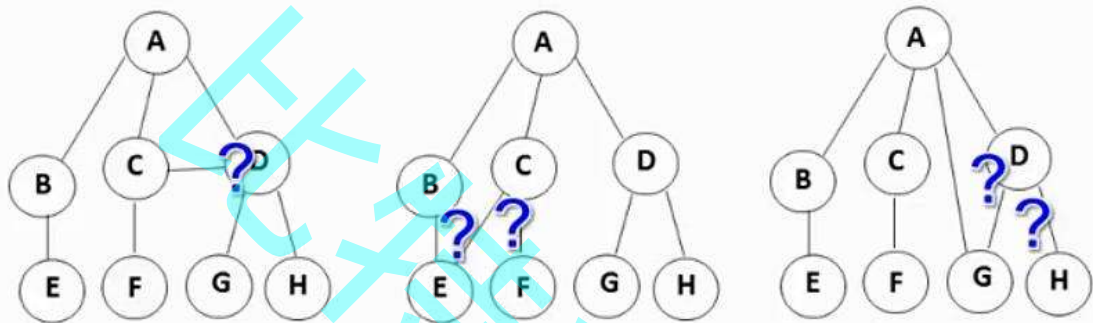


单个结点

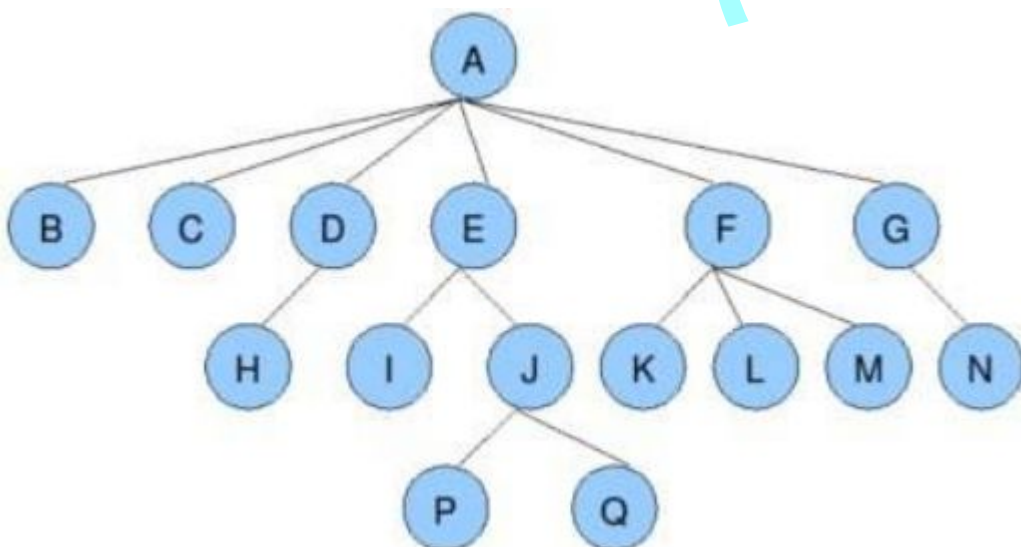
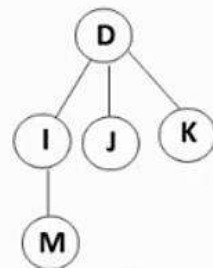
一棵普通的树



## ❖ 树与非树？



- 子树是**不相交**的；
- 除了根结点外，每个结点有且仅有一个父结点；
- 一棵**N**个结点的树有**N-1**条边。



节点的度：一个节点含有的子树的个数称为该节点的度；如上图：A的为6

叶节点或终端节点：度为0的节点称为叶节点；如上图：B、C、H、I...等节点为叶节点

非终端节点或分支节点：度不为0的节点；如上图：D、E、F、G...等节点为分支节点

双亲节点或父节点：若一个节点含有子节点，则这个节点称为其子节点的父节点；如上图：A是B的父节点

孩子节点或子节点：一个节点含有的子树的根节点称为该节点的子节点；如上图：B是A的孩子节点

兄弟节点：具有相同父节点的节点互称为兄弟节点；如上图：B、C是兄弟节点

树的度：一棵树中，最大的节点的度称为树的度；如上图：树的度为6

节点的层次：从根开始定义起，根为第1层，根的子节点为第2层，以此类推；

树的高度或深度：树中节点的最大层次；如上图：树的高度为4

堂兄弟节点：双亲在同一层的节点互为堂兄弟；如上图：H、I互为兄弟节点

节点的祖先：从根到该节点所经分支上的所有节点；如上图：A是所有节点的祖先

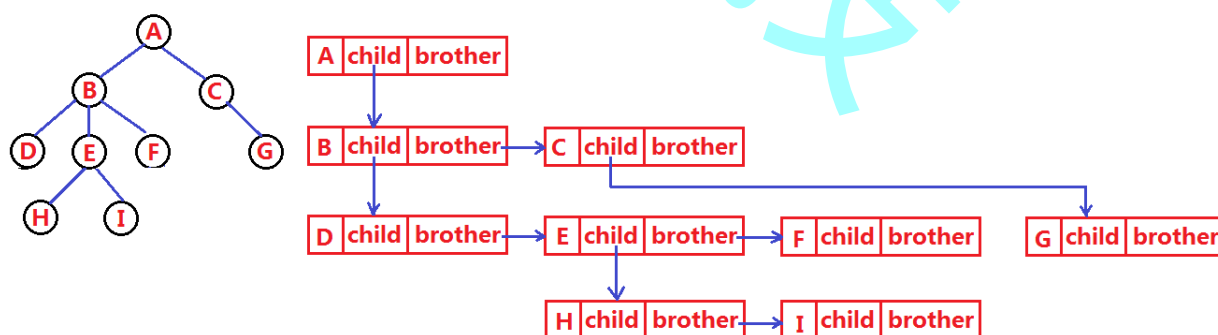
子孙：以某节点为根的子树中任一节点都称为该节点的子孙。如上图：所有节点都是A的子孙

森林：由m (m>0) 棵互不相交的树的集合称为森林；

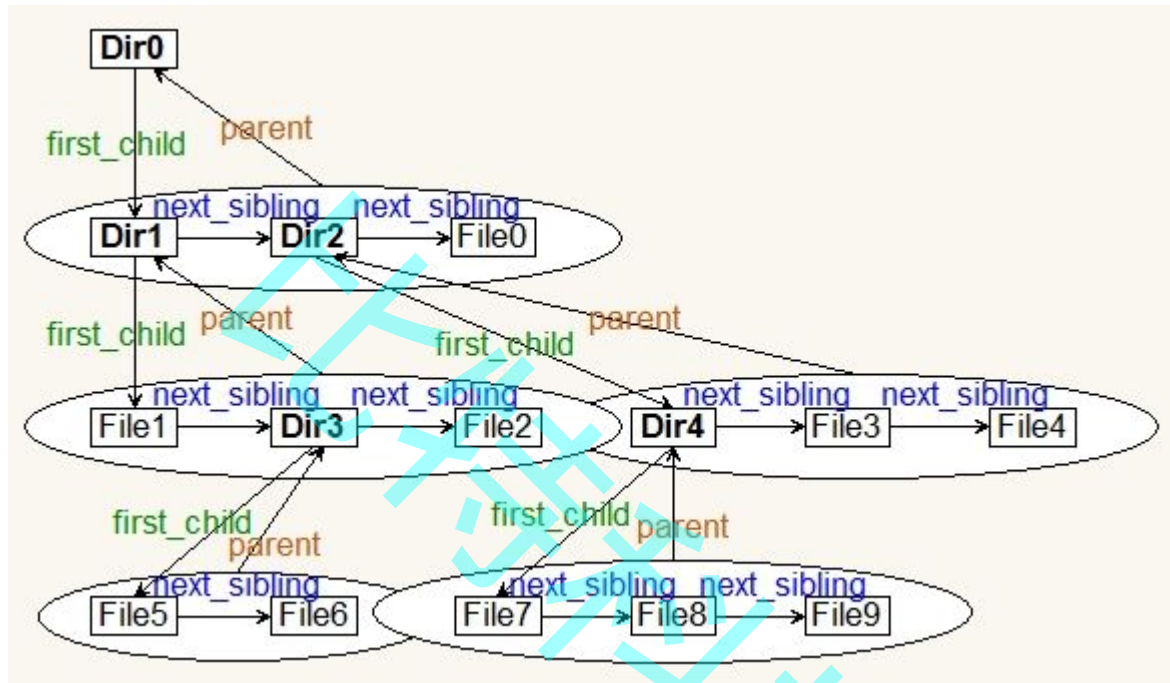
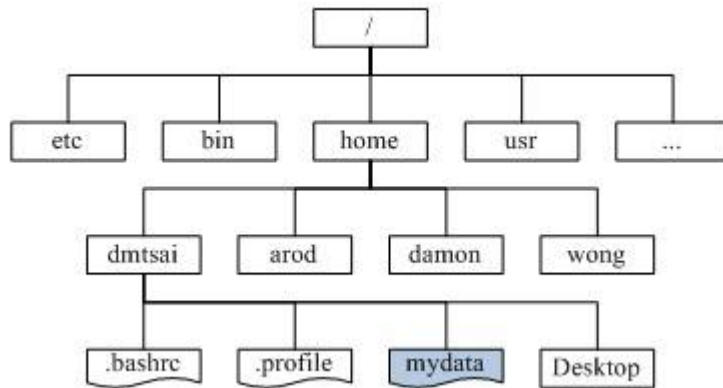
## 1.2树的表示

树结构相对线性表就比较复杂了，要存储表示起来就比较麻烦了，实际中树有很多种表示方式，如：双亲表示法，孩子表示法、孩子兄弟表示法等等。我们这里就简单的了解其中最常用的**孩子兄弟表示法**。

```
typedef int DataType;
struct Node
{
    struct Node* _firstChild1;    // 第一个孩子结点
    struct Node* _pNextBrother;   // 指向其下一个兄弟结点
    DataType _data;               // 结点中的数据域
};
```



## 1.3树在实际中的运用（表示文件系统的目录树结构）



## 2.二叉树概念及结构

### 2.1概念

一棵二叉树是结点的一个有限集合，该集合或者为空，或者是由一个根节点加上两棵别称为左子树和右子树的二叉树组成。

二叉树的特点：

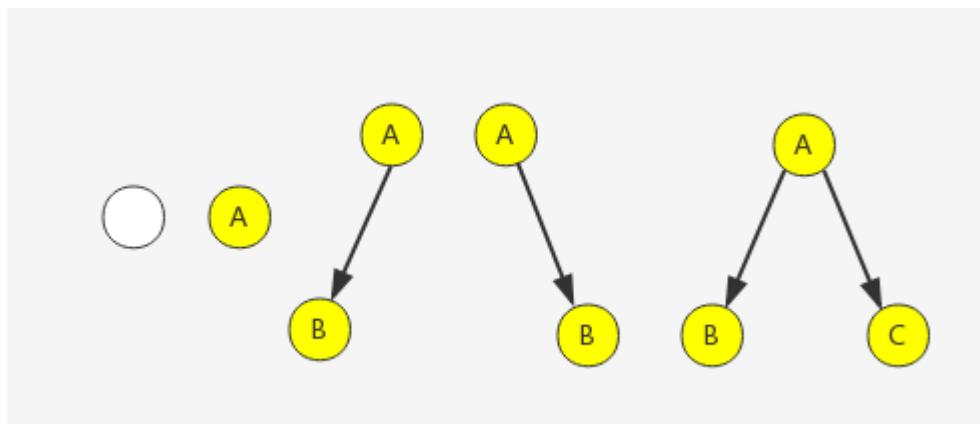
1. 每个结点最多有两棵子树，即二叉树不存在度大于2的结点。
2. 二叉树的子树有左右之分，其子树的次序不能颠倒。

### 2.2现实中的二叉树：



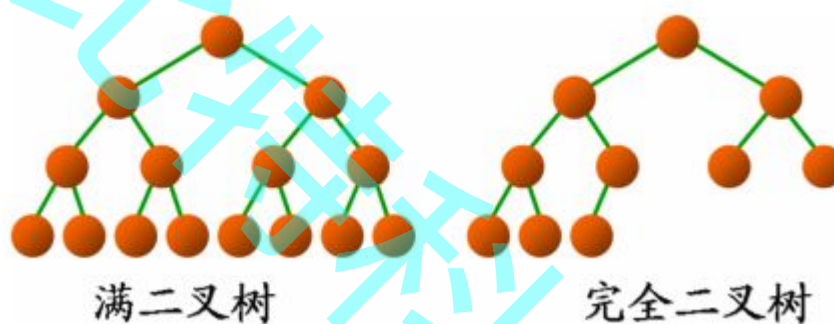


2.3数据结构中的二叉树:



## 2.4特殊的二叉树:

1. 满二叉树: 一个二叉树, 如果每一个层的结点数都达到最大值, 则这个二叉树就是满二叉树。也就是说, 如果一个二叉树的层数为 $K$ , 且结点总数是 $(2^k) - 1$ , 则它就是满二叉树。
2. 完全二叉树: 完全二叉树是效率很高的数据结构, 完全二叉树是由满二叉树而引出来的。对于深度为 $K$ 的, 有 $n$ 个结点的二叉树, 当且仅当其每一个结点都与深度为 $K$ 的满二叉树中编号从1至 $n$ 的结点一一对应时称之为完全二叉树。要注意的是满二叉树是一种特殊的完全二叉树。



## 2.5 二叉树的存储结构

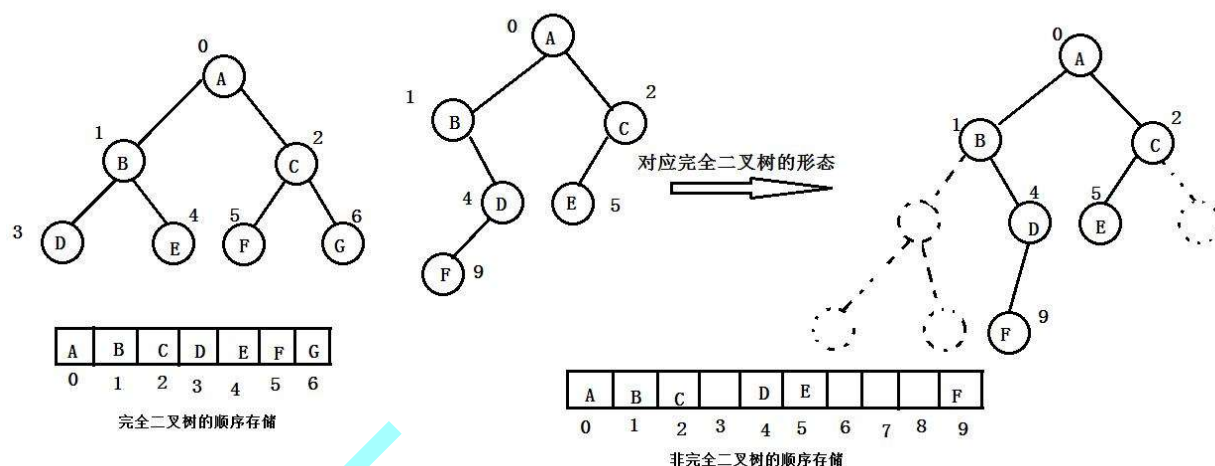
二叉树一般可以使用两种结构存储, 一种顺序结构, 一种链式结构。

### 二叉树的性质

1. 若规定根节点的层数为1, 则一棵非空二叉树的第 $i$ 层上最多有 $2^{(i-1)}$ 个结点。
2. 若规定根节点的层数为1, 则深度为 $h$ 的二叉树的最大结点数是 $2^h - 1$ 。
3. 对任何一棵二叉树, 如果度为0其叶结点个数为  $n_0$ , 度为2的分支结点个数为  $n_2$ , 则有  $n_0 = n_2 + 1$
4. 若规定根节点的层数为1, 具有 $n$ 个结点的满二叉树的深度,  $h = \log_2(n+1)$ . (ps:  $\log_2(n+1)$ 是log以2为底,  $n+1$ 为对数)
5. 对于具有 $n$ 个结点的完全二叉树, 如果按照从上至下从左至右的数组顺序对所有节点从0开始编号, 则对于序号为 $i$ 的结点有:
  1. 若 $i > 0$ ,  $i$ 位置节点的双亲序号:  $(i-1)/2$ ;  $i=0$ ,  $i$ 为根节点编号, 无双亲节点
  2. 若 $2i+1 < n$ , 左孩子序号:  $2i+1$ ,  $2i+1 \geq n$ 否则无左孩子
  3. 若 $2i+2 < n$ , 右孩子序号:  $2i+2$ ,  $2i+2 \geq n$ 否则无右孩子

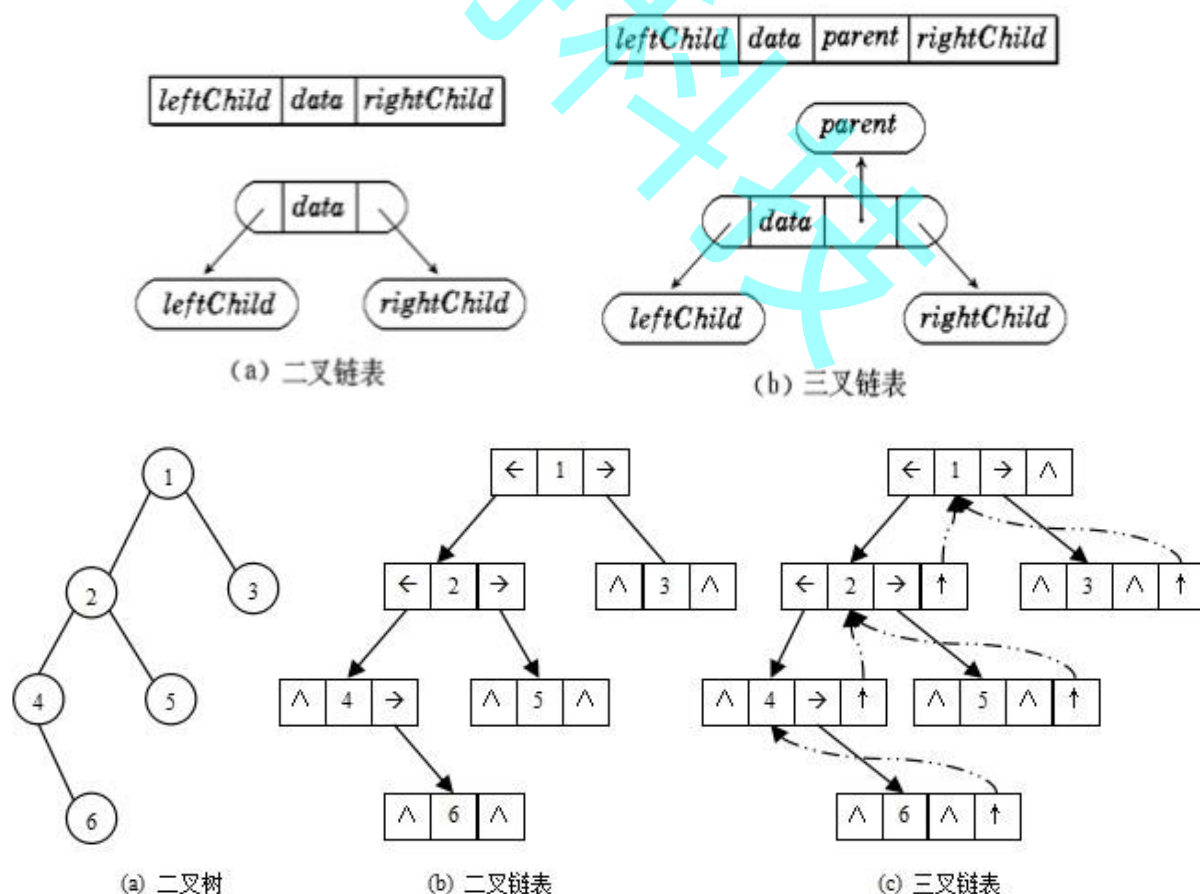
### 2.5.1 顺序存储:

顺序结构存储就是使用**数组来存储**，一般使用数组**只适合表示完全二叉树**，因为不是完全二叉树会有空间的浪费。而现实中使用中只有堆才会使用数组来存储，关于堆我们后面的章节会专门讲解。**二叉树顺序存储在物理上是一个数组，在逻辑上是一颗二叉树。**



## 2.5.2 链式存储:

二叉树的链式存储结构是指，用链表来表示一棵二叉树，即用链来指示元素的逻辑关系。通常的方法是链表中每个结点由三个域组成，数据域和左右指针域，左右指针分别用来给出该结点左孩子和右孩子所在的链结点的存储地址。链式结构又分为二叉链和三叉链，当前我们学习中一般都是二叉链，后面课程学到高阶数据结构如红黑树等会用到三叉链。





```
// 二叉链
struct BinaryTreeNode
{
    struct BinTreeNode* _pLeft;    // 指向当前节点左孩子
    struct BinTreeNode* _pRight;   // 指向当前节点右孩子
    BTDataType _data; // 当前节点值域
}

// 三叉链
struct BinaryTreeNode
{
    struct BinTreeNode* _pParent; // 指向当前节点的双亲
    struct BinTreeNode* _pLeft;   // 指向当前节点左孩子
    struct BinTreeNode* _pRight;  // 指向当前节点右孩子
    BTDataType _data; // 当前节点值域
};
```

## 选择题

- 某二叉树共有 399 个结点，其中有 199 个度为 2 的结点，则该二叉树中的叶子结点数为 ( )  
 A 不存在这样的二叉树  
 B 200  
 C 198  
 D 199
- 下列数据结构中，不适合采用顺序存储结构的是 ( )  
 A 非完全二叉树  
 B 堆  
 C 队列  
 D 栈
- 在具有  $2n$  个结点的完全二叉树中，叶子结点个数为 ( )  
 A  $n$   
 B  $n+1$   
 C  $n-1$   
 D  $n/2$
- 一棵完全二叉树的节点数位为 531 个，那么这棵树的高度为 ( )  
 A 11  
 B 10  
 C 8  
 D 12
- 一个具有 767 个节点的完全二叉树，其叶子节点个数为 ( )  
 A 383  
 B 384  
 C 385  
 D 386

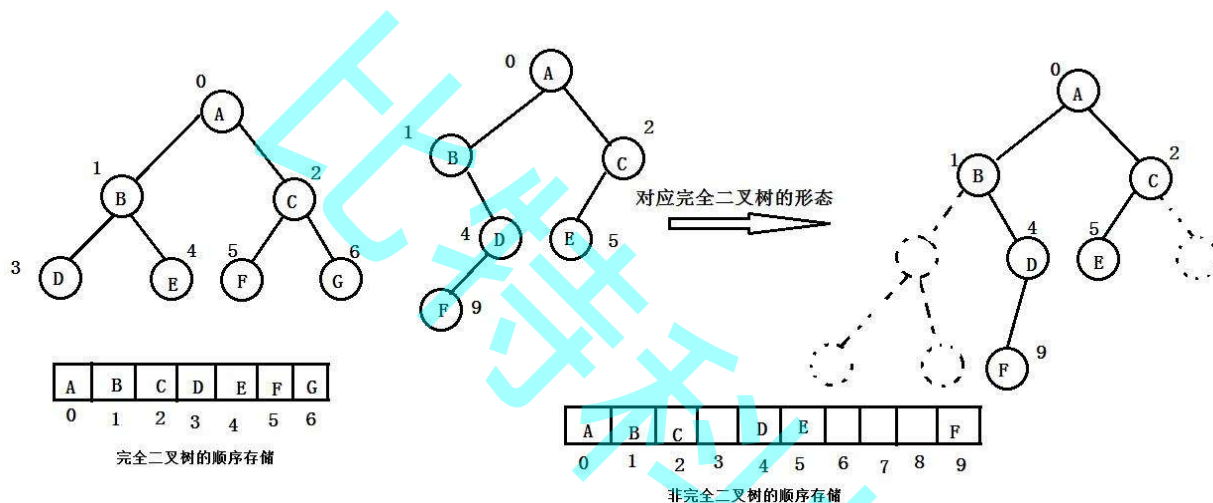
## 选择题答案

1. B
2. A
3. A
4. B
5. B

## 3. 二叉树的顺序结构及实现

### 3.1 二叉树的顺序结构

普通的二叉树是不适合用数组来存储的，因为可能会存在大量的空间浪费。而完全二叉树更适合使用顺序结构存储。现实中我们通常把堆(一种二叉树)使用顺序结构的数组来存储，需要注意的是这里的堆和操作系统虚拟进程地址空间中的堆是两回事，一个是数据结构，一个是操作系统中管理内存的一块区域分段。

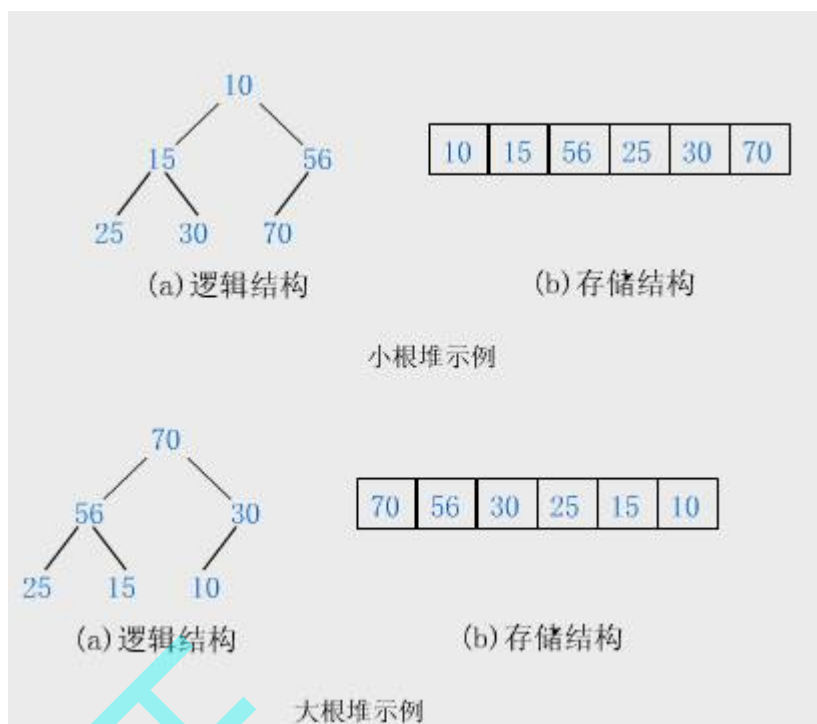


### 3.2 堆的概念及结构

如果有一个关键码的集合 $K = \{k_0, k_1, k_2, \dots, k_{n-1}\}$ ，把它的所有元素按完全二叉树的顺序存储方式存储在一个一维数组中，并满足： $k_i \leq k_{2i+1}$  且  $k_i \leq k_{2i+2}$  ( $k_i \geq k_{2i+1}$  且  $k_i \geq k_{2i+2}$ )  $i = 0, 1, 2, \dots$ ，则称为小堆(或大堆)。将根节点最大的堆叫做最大堆或大根堆，根节点最小的堆叫做最小堆或小根堆。

堆的性质：

- 堆中某个节点的值总是不大于或不小于其父节点的值；
- 堆总是一棵完全二叉树。



### 选择题

- 下列关键字序列为堆的是: ()  
 A 100, 60, 70, 50, 32, 65  
 B 60, 70, 65, 50, 32, 100  
 C 65, 100, 70, 32, 50, 60  
 D 70, 65, 100, 32, 50, 60  
 E 32, 50, 100, 70, 65, 60  
 F 50, 100, 70, 65, 60, 32
- 已知小根堆为8, 15, 10, 21, 34, 16, 12, 删除关键字 8 之后需重建堆, 在此过程中, 关键字之间的比较次数是 ()。  
 A 1  
 B 2  
 C 3  
 D 4
- 一组记录排序码为(5 11 7 2 3 17), 则利用堆排序方法建立的初始堆为  
 A(11 5 7 2 3 17)  
 B(11 5 7 2 17 3)  
 C(17 11 7 2 3 5)  
 D(17 11 7 5 3 2)  
 E(17 7 11 3 5 2)  
 F(17 7 11 3 2 5)
- 最小堆[0, 3, 2, 5, 7, 4, 6, 8], 在删除堆顶元素0之后, 其结果是 ()  
 A[3, 2, 5, 7, 4, 6, 8]  
 B[2, 3, 5, 7, 4, 6, 8]  
 C[2, 3, 4, 5, 7, 8, 6]  
 D[2, 3, 4, 5, 6, 7, 8]

## 选择题答案

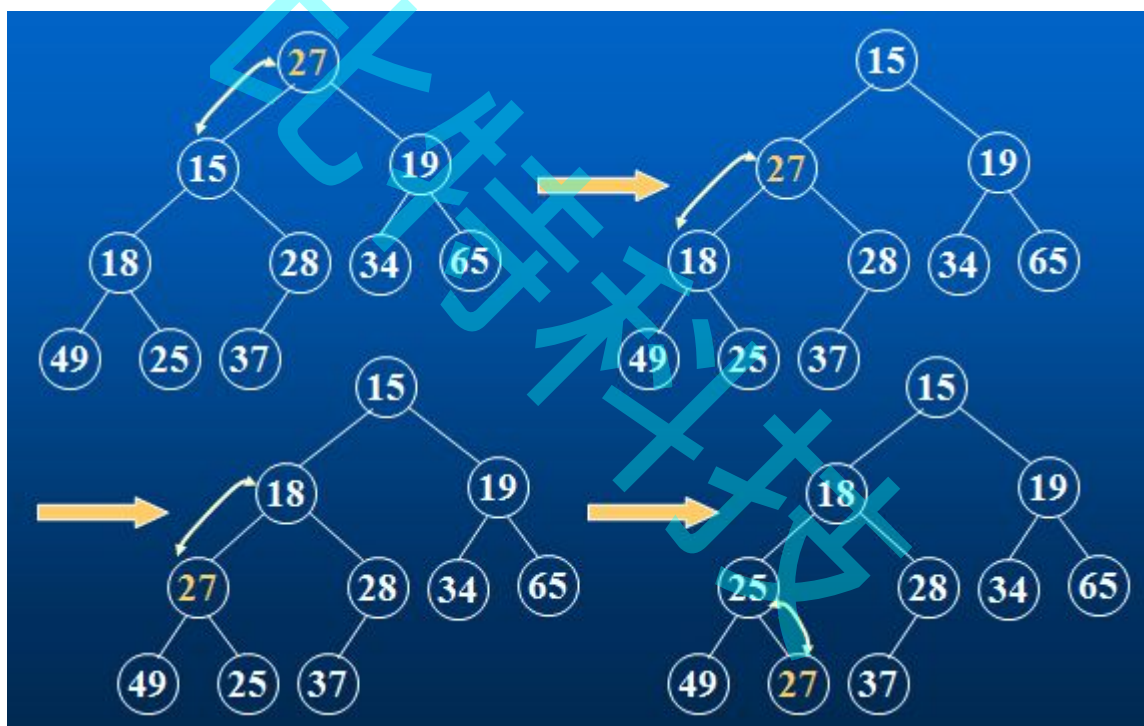
1. A
2. C
3. C
4. C

## 3.2 堆的实现

### 3.2.1 堆向下调整算法

现在我们给出一个数组，逻辑上看做一颗完全二叉树。我们通过从根节点开始的向下调整算法可以把它调整成一个小堆。向下调整算法有一个前提：左右子树必须是一个堆，才能调整。

```
int a[] = {27,15,19,18,28,34,65,49,25,37};
```

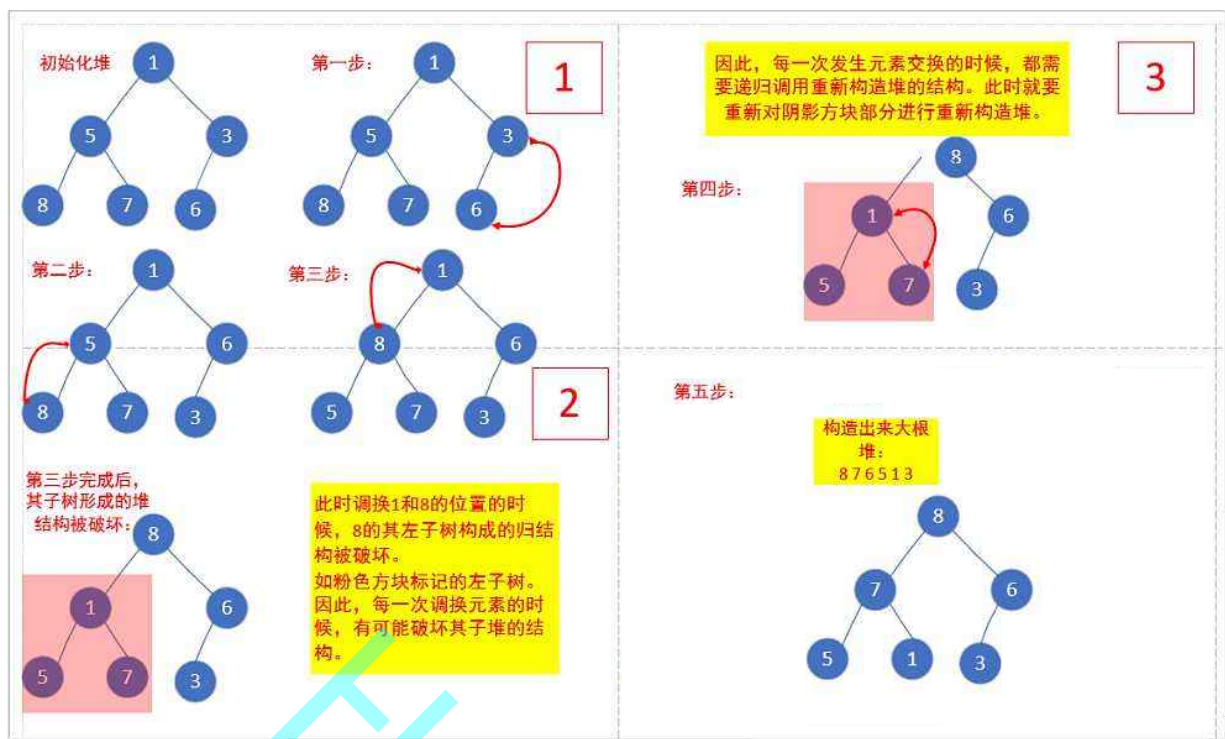


### 3.2.2 堆的创建

下面我们给出一个数组，这个数组逻辑上可以看做一颗完全二叉树，但是还不是一个堆，现在我们通过算法，把它构建成一个堆。根节点左右子树不是堆，我们怎么调整呢？这里我们从倒数的第一个非叶子节点的子树开始调整，一直调整到根节点的树，就可以调整成堆。

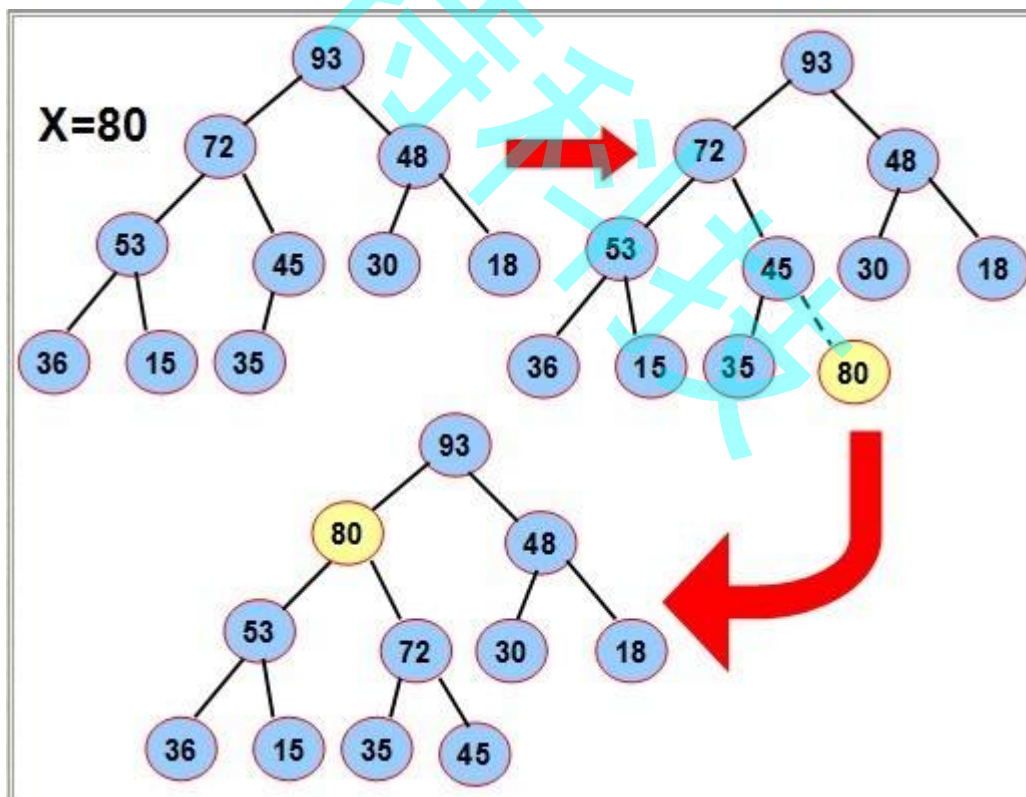
```
int a[] = {1,5,3,8,7,6};
```





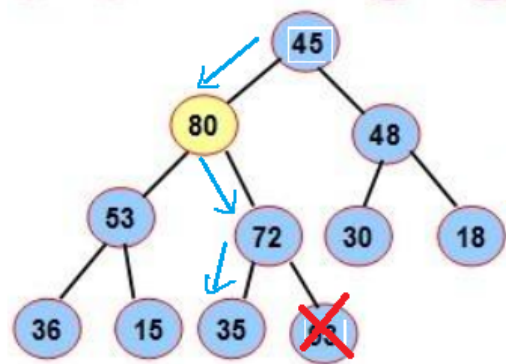
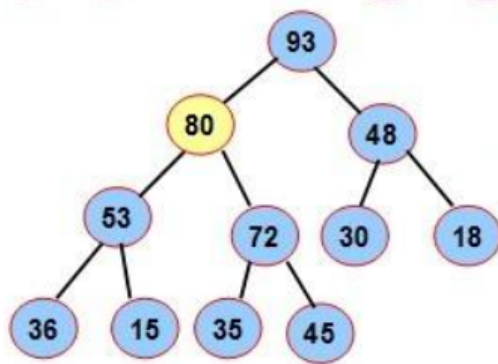
### 3.2.3 堆的插入

先插入一个80到数组的尾上，再进行向上调整算法，直到满足堆。

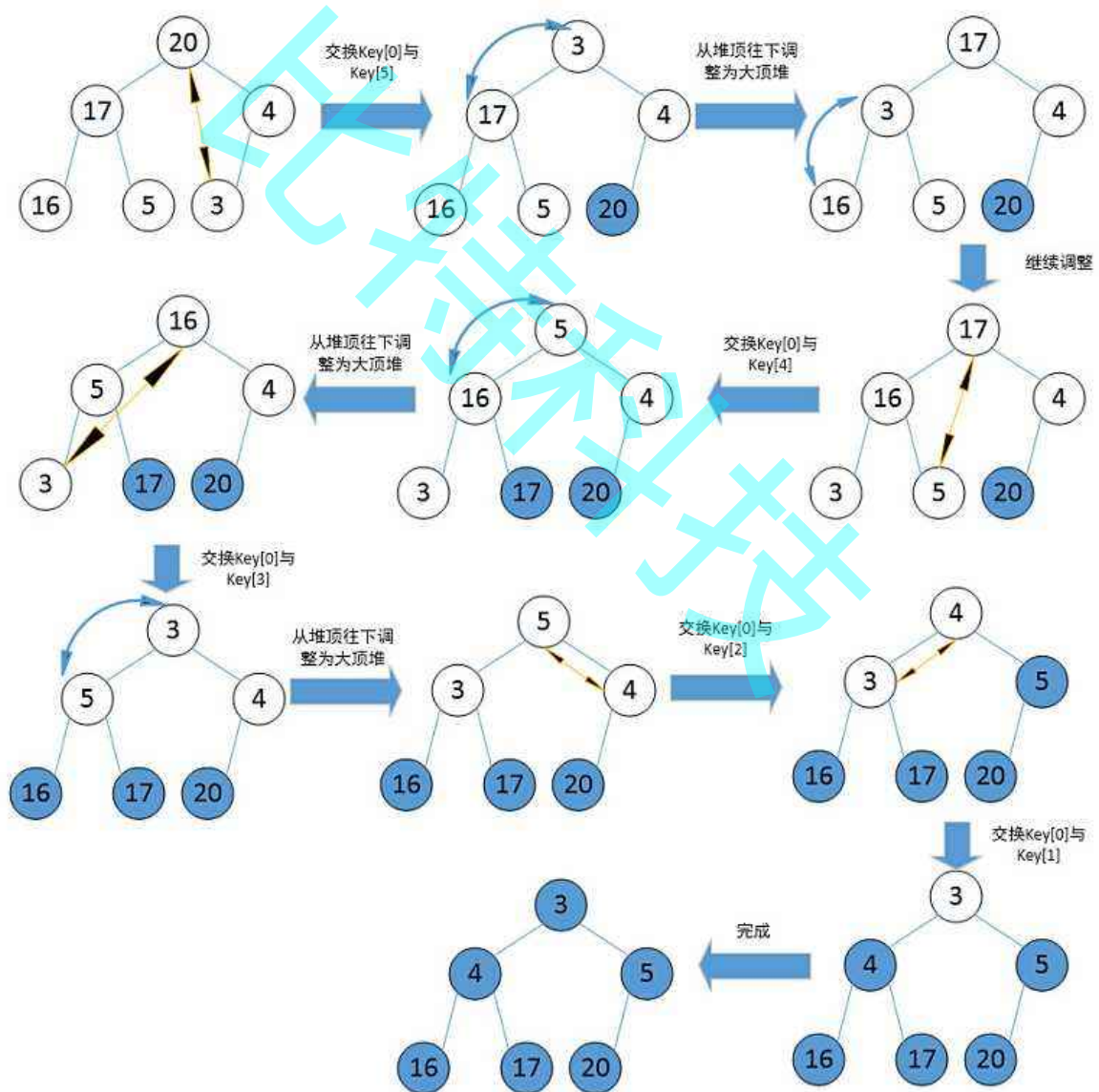


### 3.2.4 堆的删除

删除堆是删除堆顶的数据，将堆顶的数据跟最后一个数据一换，然后删除数组最后一个数据，再进行向下调整算法。



### 3.2.5 堆排序



### 3.2.6 堆的代码实现

```
typedef int HPDataType;
```

```

typedef struct Heap
{
    HPDataType* _a;
    int _size;
    int _capacity;
}Heap;

// 堆的构建
void HeapCreate(Heap* hp, HPDataType* a, int n);
// 堆的销毁
void HeapDestory(Heap* hp);
// 堆的插入
void HeapPush(Heap* hp, HPDataType x);
// 堆的删除
void HeapPop(Heap* hp);
// 取堆顶的数据
HPDataType HeapTop(Heap* hp);
// 堆的数据个数
int HeapSize(Heap* hp);
// 堆的判空
int HeapEmpty(Heap* hp);

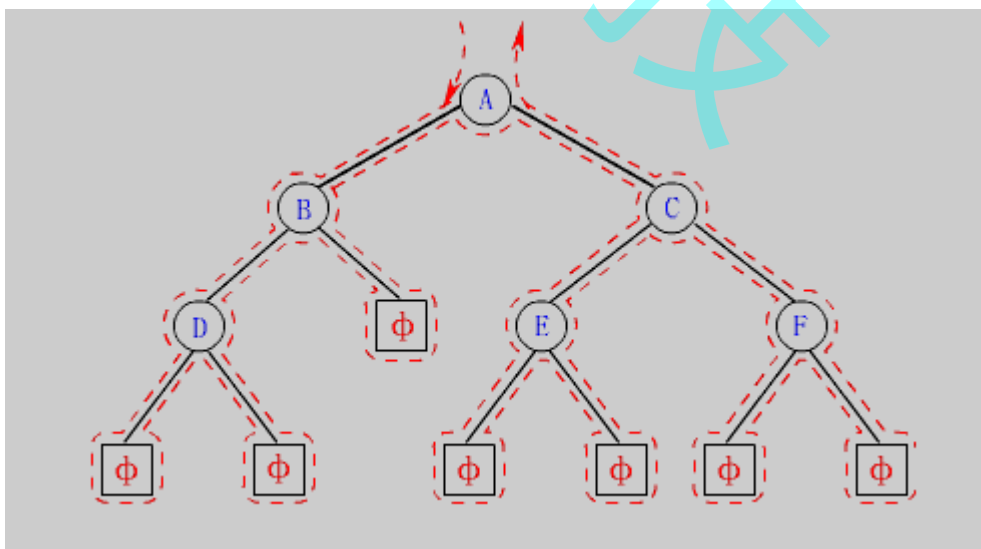
// 对数组进行堆排序
void HeapSort(int* a, int n);

```

## 4.二叉树链式结构的实现

### 4.1二叉树链式结构的遍历

所谓遍历(Traversal)是指沿着某条搜索路线,依次对树中每个结点均做一次且仅做一次访问。访问结点所做的操作依赖于具体的应用问题。遍历是二叉树上最重要的运算之一,是二叉树上进行其它运算之基础。

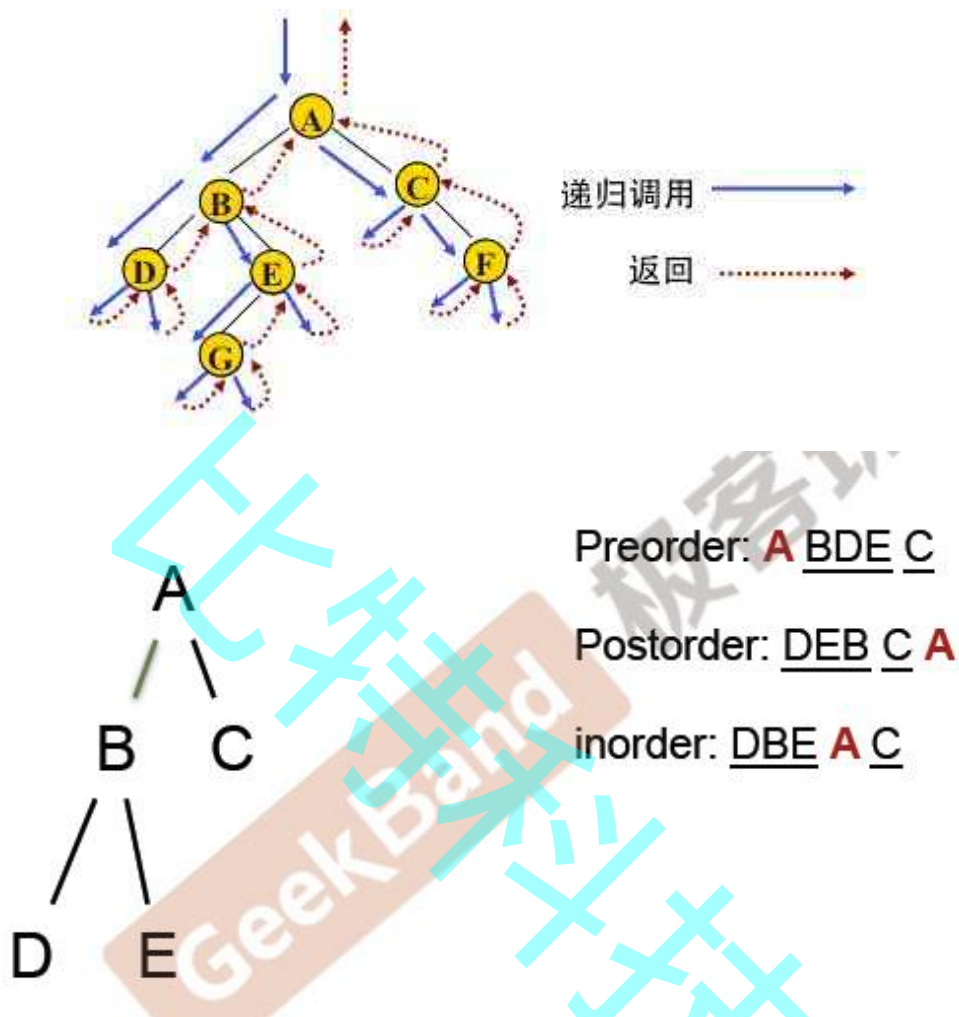


**前序/中序/后序的递归结构遍历：**是根据访问结点操作发生位置命名

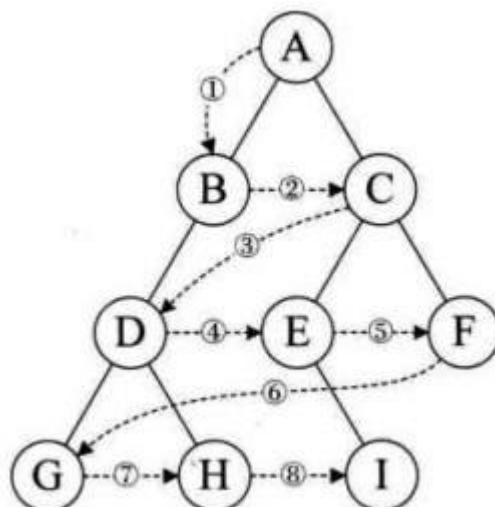
1. NLR：前序遍历(Preorder Traversal 亦称先序遍历)——访问根结点的操作发生在遍历其左右子树之前。
2. LNR：中序遍历(Inorder Traversal)——访问根结点的操作发生在遍历其左右子树之中（间）。

3. LRN: 后序遍历(Postorder Traversal)——访问根结点的操作发生在遍历其左右子树之后。

由于被访问的结点必是某子树的根，所以N(Node) 、L(Left subtree) 和R(Right subtree) 又可解释为根、根的左子树和根的右子树。NLR、LNR和LRN分别又称为先根遍历、中根遍历和后根遍历。

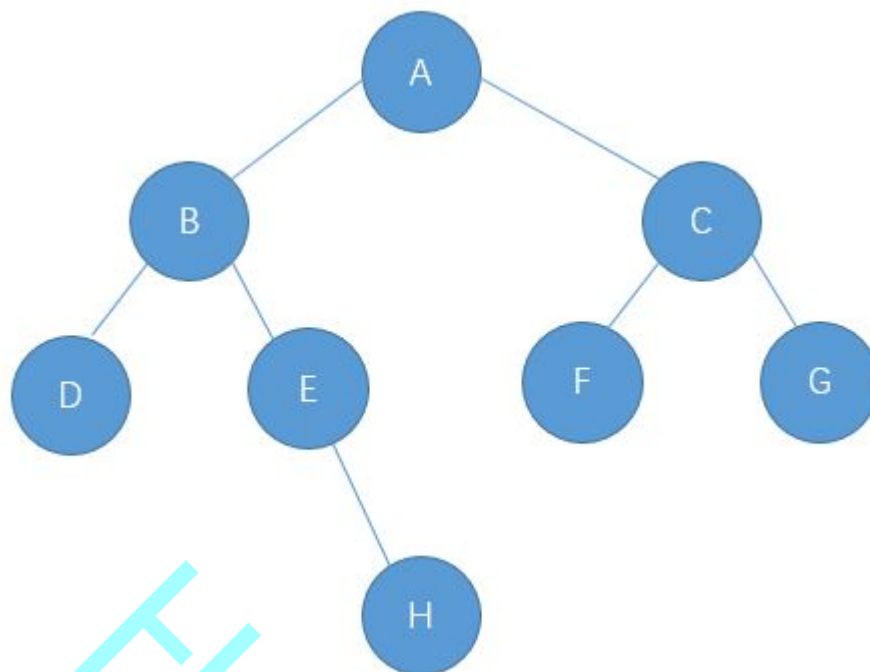


**层序遍历:** 除了先序遍历、中序遍历、后序遍历外，还可以对二叉树进行层序遍历。设二叉树的根节点所在层数为1，层序遍历就是从所在二叉树的根节点出发，首先访问第一层的树根节点，然后从左到右访问第2层上的节点，接着是第三层的节点，以此类推，自上而下，自左至右逐层访问树的结点的过程就是层序遍历。





练习：请写出下面的前序/中序/后序/层序遍历



### 选择题

1. 某完全二叉树按层次输出（同一层从左到右）的序列为 ABCDEFGH。该完全二叉树的前序序列为（ ）  
A ABDHECFG  
B ABCDEFGH  
C HDBEAFCG  
D HDEBFGCA
2. 二叉树的先序遍历和中序遍历如下：先序遍历：EFHIGJK；中序遍历：HFIEJKG。则二叉树根结点为（ ）  
A E  
B F  
C G  
D H
3. 设一棵二叉树的中序遍历序列：badce，后序遍历序列：bdeca，则二叉树先序遍历序列为\_\_\_\_。  
A adbce  
B decab  
C debac  
D abcde
4. 某二叉树的后序遍历序列与中序遍历序列相同，均为 ABCDEF，则按层次输出（同一层从左到右）的序列为  
A FEDCBA  
B CBAFED  
C DEFCBA  
D ABCDEF

### 选择题答案

1. A
2. A
3. D
4. A

## 4.2 二叉树的基础面试题

1. 单值二叉树。 [OJ链接](#)
2. 二叉树最大深度。 [OJ链接](#)
3. 翻转二叉树。 [OJ链接](#)
4. 检查两颗树是否相同。 [OJ链接](#)
5. 对称二叉树。 [OJ链接](#)
6. 二叉树的前序遍历。 [OJ链接](#)
7. 二叉树中序遍历。 [OJ链接](#)
8. 二叉树的后序遍历。 [OJ链接](#)
9. 另一颗树的子树。 [OJ链接](#)
10. 判断一颗二叉树是否是平衡二叉树。 [OJ链接](#)
11. 二叉树的构建及遍历。 [OJ链接](#)

## 4.3 二叉树的基础面试题补充（没有合适的OJ）

```
typedef char BTDataType;

typedef struct BinaryTreeNode
{
    BTDataType _data;
    struct BinaryTreeNode* _left;
    struct BinaryTreeNode* _right;
}BTNode;

// 通过前序遍历的数组"ABD##E#H##CF##G##"构建二叉树
BTNode* BinaryTreeCreate(BTDataType* a, int n, int* pi);
// 二叉树销毁
void BinaryTreeDestory(BTNode** root);
// 二叉树节点个数
int BinaryTreeSize(BTNode* root);
// 二叉树叶子节点个数
int BinaryTreeLeafSize(BTNode* root);
// 二叉树第k层节点个数
int BinaryTreeLevelKSize(BTNode* root, int k);
// 二叉树查找值为x的节点
BTNode* BinaryTreeFind(BTNode* root, BTDataType x);
// 二叉树前序遍历
void BinaryTreePrevOrder(BTNode* root);
// 二叉树中序遍历
void BinaryTreeInOrder(BTNode* root);
// 二叉树后序遍历
void BinaryTreePostOrder(BTNode* root);
// 层序遍历
void BinaryTreeLevelOrder(BTNode* root);
// 判断二叉树是否是完全二叉树
int BinaryTreeComplete(BTNode* root);
```