Lesson5--二叉树

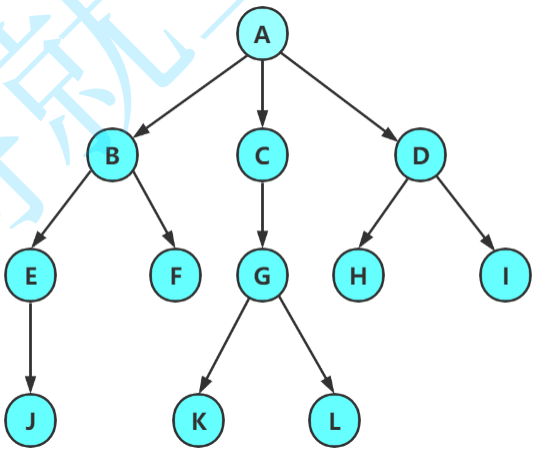
【本节目标】

- **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**

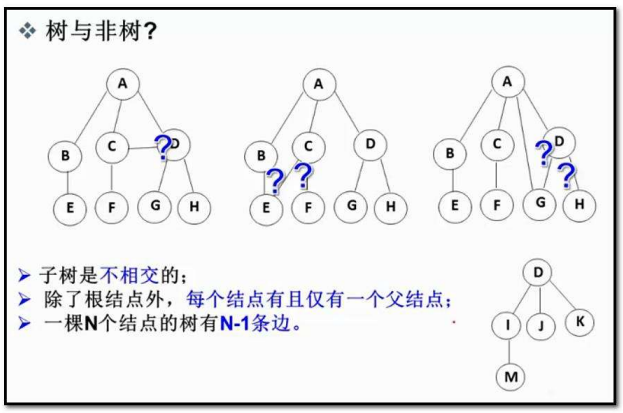
1.树概念及结构

1.1树的概念

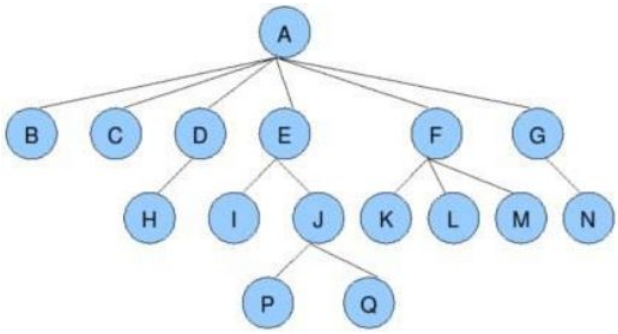
> 树是一种**非线性**的数据结构，它是由n（n>=0）个有限结点组成一个具有层次关系的集合。**把它叫做树是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。**
>
> - 有一个**特殊的结点，称为根结点**，根结点没有前驱结点
> - 除根结点外，**其余结点被分成M(M>0)个互不相交的集合T1、T2、.....、Tm**，其中每一个集合Ti(1<= i <= m)又是一棵结构与树类似的子树。每棵子树的根结点有且只有一个前驱，可以有0个或多个后继
> - 因此，**树是递归定义**的。



注意：树形结构中，子树之间不能有交集，否则就不是树形结构



1.2 树的相关概念



结点的度：一个结点含有的子树的个数称为该结点的度； 如上图：A的为6

叶结点或终端结点：度为0的结点称为叶结点； 如上图：B、C、H、I...等结点为叶结点

非终端结点或分支结点：度不为0的结点； 如上图：D、E、F、G...等结点为分支结点

双亲结点或父结点：若一个结点含有子结点，则这个结点称为其子结点的父结点； 如上图：A是B的父结点

孩子结点或子结点：一个结点含有的子树的根结点称为该结点的子结点； 如上图：B是A的孩子结点

兄弟结点：具有相同父结点的结点互称为兄弟结点； 如上图：B、C是兄弟结点

树的度：一棵树中，最大的结点的度称为树的度； 如上图：树的度为6

结点的层次：从根开始定义起，根为第1层，根的子结点为第2层，以此类推；

树的高度或深度：树中结点的最大层次； 如上图：树的高度为4

堂兄弟结点：双亲在同一层的结点互为堂兄弟；如上图：H、I互为兄弟结点

结点的祖先：从根到该结点所经分支上的所有结点；如上图：A是所有结点的祖先

子孙：以某结点为根的子树中任一结点都称为该结点的子孙。如上图：所有结点都是A的子孙

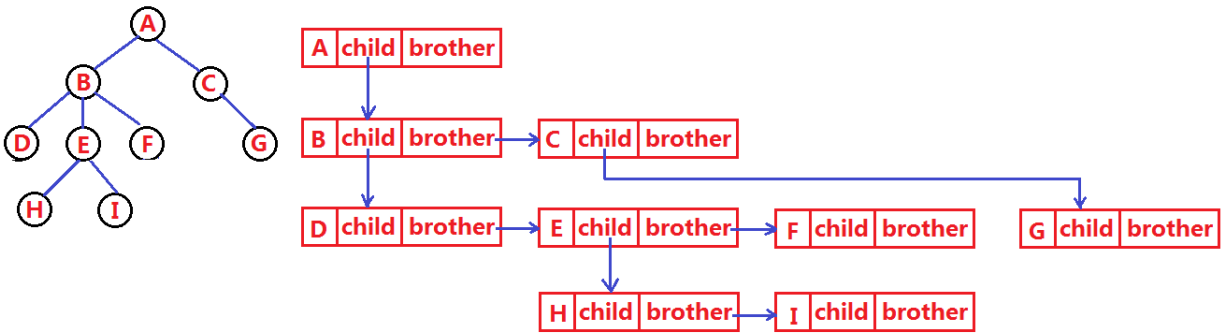
森林：由m（m>0）棵互不相交的树的集合称为森林；

1.3 树的表示

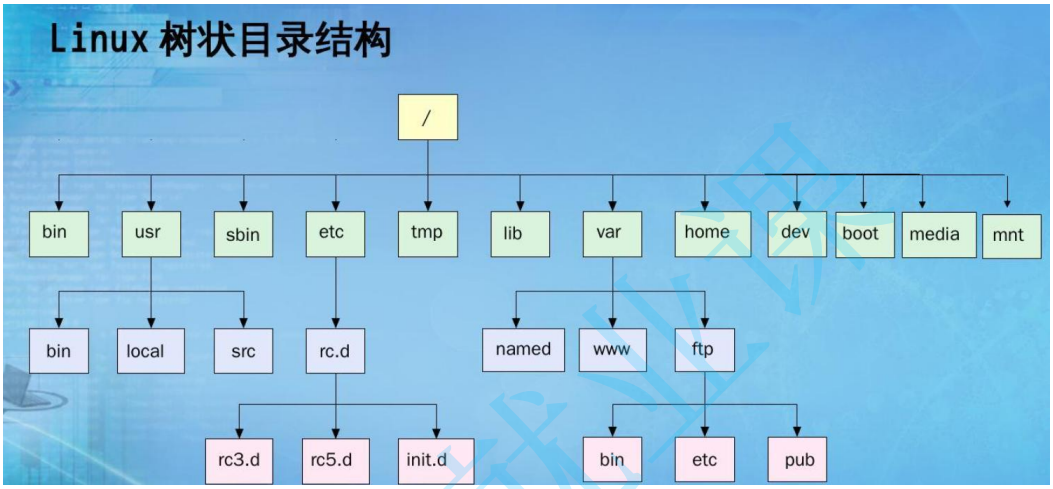
树结构相对线性表就比较复杂了，要存储表示起来就比较麻烦了，**既然保存值域，也要保存结点和结点之间的关系**，实际中树有很多种表示方式如：双亲表示法，孩子表示法、孩子双亲表示法以及孩子兄弟表示法等。我们这里就简单的了解其中最常用的**孩子兄弟表示法**。

```
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
};
```

比特就业课



1.4 树在实际中的运用（表示文件系统的目录树结构）

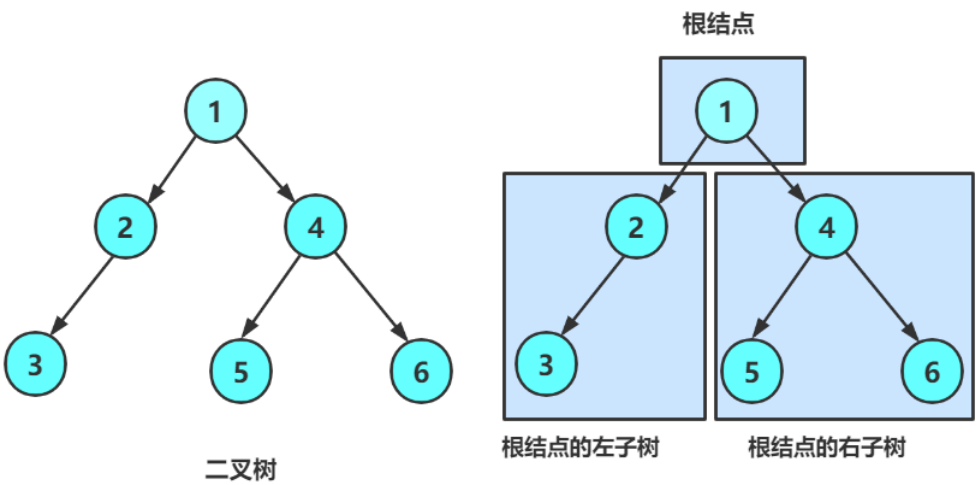


2.二叉树概念及结构

2.1概念

一棵二叉树是结点的一个有限集合，该集合:

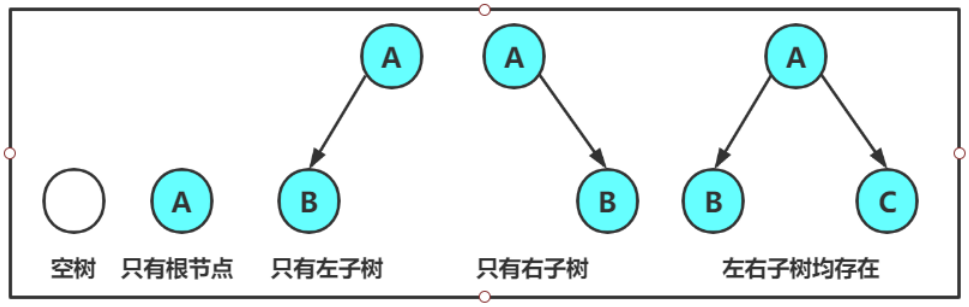
1. 或者为空
2. 由一个根结点加上两棵别称为左子树和右子树的二叉树组成



从上图可以看出:

1. 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，次序不能颠倒，因此二叉树是有序树

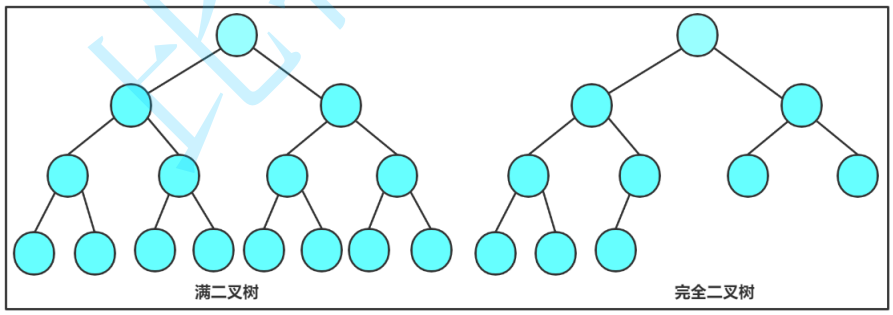
注意：对于任意的二叉树都是由以下几种情况复合而成的：



2.2现实中的二叉树：

2.3 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。



2.4 二叉树的性质

1. 若规定根结点的层数为1，则一棵非空二叉树的**第i层上最多有**$2^{(i-1)}$ 个结点.
2. 若规定根结点的层数为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**$n_0 = n_2 + 1$

```
/*
* 假设二叉树有N个结点
* 从总结点数角度考虑：N = n0 + n1 + n2 ①
*
* 从边的角度考虑，N个结点的任意二叉树，总共有N-1条边
* 因为二叉树中每个结点都有双亲，根结点没有双亲，每个节点向上与其双亲之间存在一条边
* 因此N个结点的二叉树总共有N-1条边
*
* 因为度为0的结点没有孩子，故度为0的结点不产生边；度为1的结点只有一个孩子，故每个度为1的结点* * 产生一条边；度为2的结点有2个孩子，故每个度为2的结点产生两条边，所以总边数为：n1+2*n2
* 故从边的角度考虑：N-1 = n1 + 2*n2 ②
* 结合① 和 ②得：n0 + n1 + n2 = n1 + 2*n2 - 1
* 即：n0 = n2 + 1
*/
```

4. 若规定根结点的层数为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结点从0开始编号，则对于序号为i的结点有：
    1. **若i>0，i位置结点的双亲序号：(i-1)/2**；i=0，i为根结点编号，无双亲结点
    2. **若2i+1<n，左孩子序号：2i+1，2i+1>=n否则无左孩子**
    3. **若2i+2<n，右孩子序号：2i+2，2i+2>=n否则无右孩子**

```
1. 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

2.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

3.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

4.一棵完全二叉树的结点数位为531个，那么这棵树的高度为（ ）
A 11
B 10
C 8
D 12


5.一个具有767个结点的完全二叉树，其叶子结点个数为（）
```

比特就业课

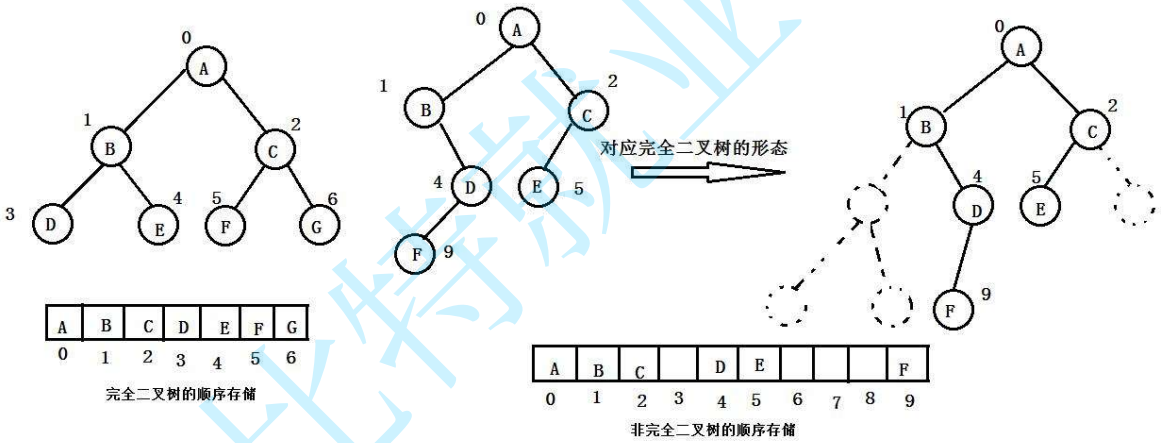
```
A 383
B 384
C 385
D 386

答案:
1.B
2.A
3.A
4.B
5.B
```

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。

1. **顺序存储**

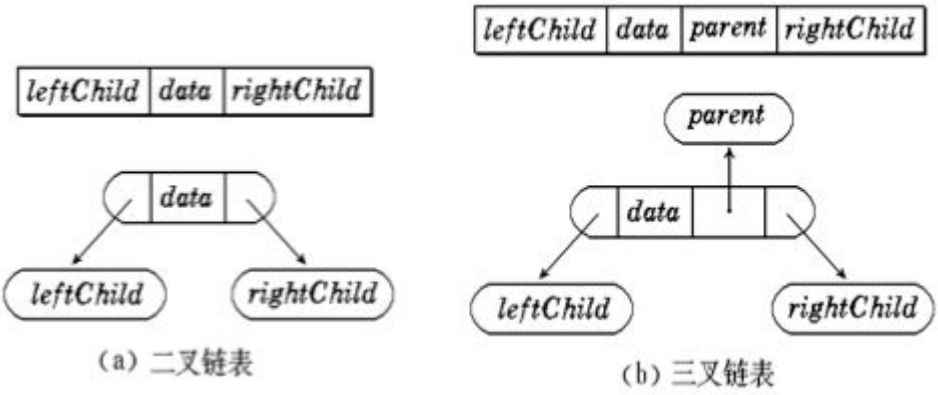
   顺序结构存储就是使用**数组来存储**，一般使用数组**只适合表示完全二叉树**，因为不是完全二叉树会有空间的浪费。而现实中使用中只有堆才会使用数组来存储，关于堆我们后面的章节会专门讲解。**二叉树顺序存储在物理上是一个数组，在逻辑上是一颗二叉树。**

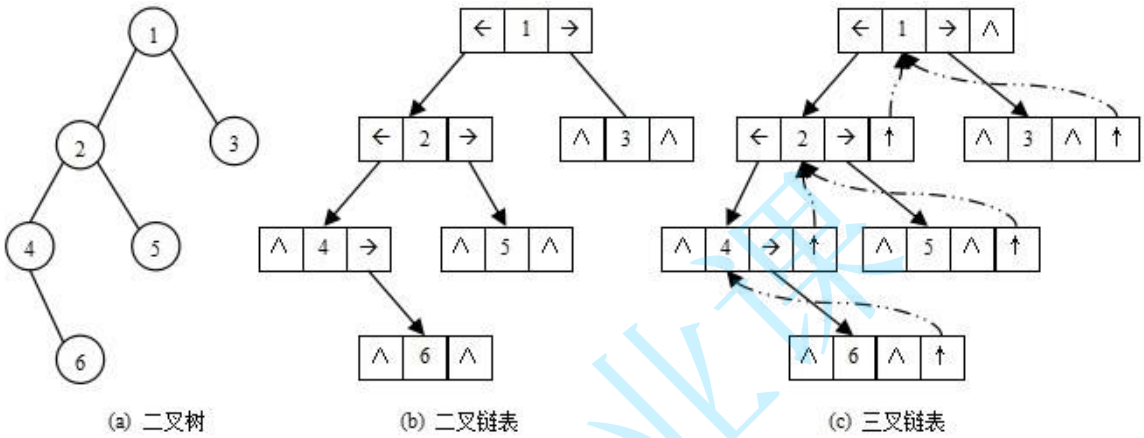


2. **链式存储**

   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。

比特就业课



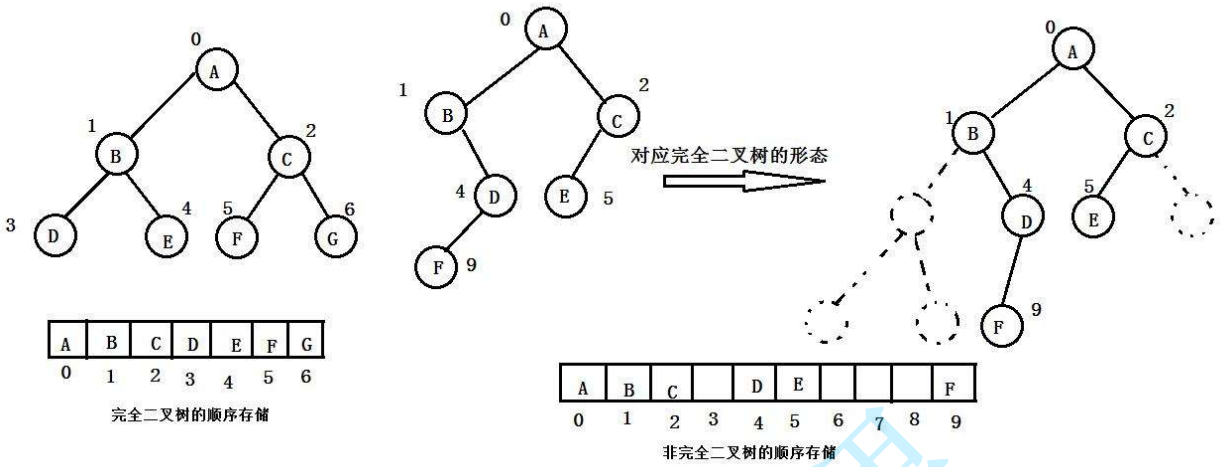



```c
typedef int BTDataType;
// 二叉链
struct BinaryTreeNode
{
    struct BinTreeNode* left;   // 指向当前结点左孩子
    struct BinTreeNode* right;  // 指向当前结点右孩子
    BTDataType data;            // 当前结点值域
}

// 三叉链
struct BinaryTreeNode
{
    struct BinTreeNode* parent; // 指向当前结点的双亲
    struct BinTreeNode* left;   // 指向当前结点左孩子
    struct BinTreeNode* right;  // 指向当前结点右孩子
    BTDataType data;            // 当前结点值域
};
```

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。**

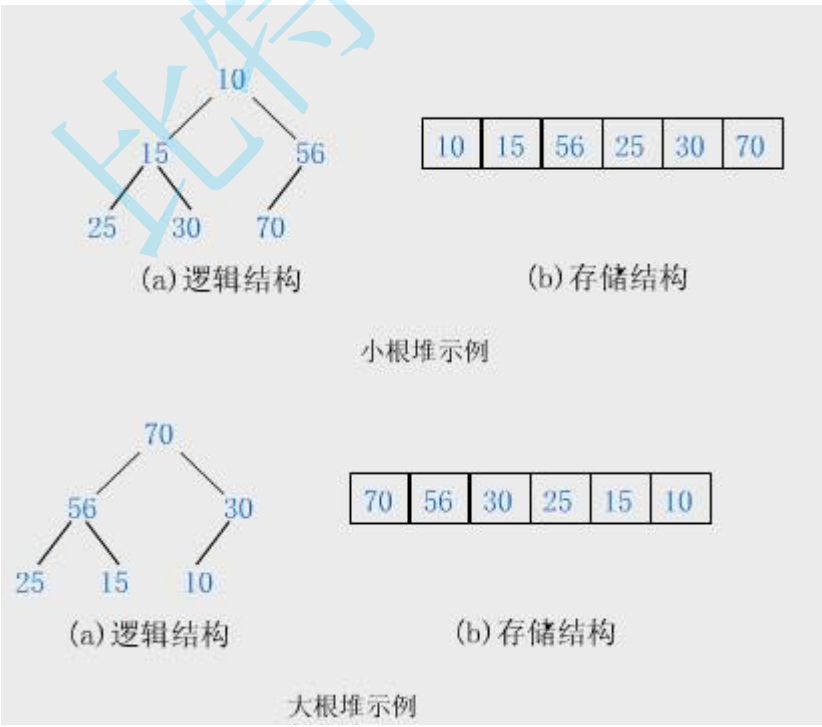


3.2 堆的概念及结构

如果有一个关键码的集合K = {k_0，k_1，k_2，...，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 <= K_{2*i+1} 且 K_i<= K_{2*i+2} (K_i >= K_{2*i+1} 且 K_i >= K_{2*i+2}) i = 0，1，2...，则称为小堆(或大堆)。将根结点最大的堆叫做最大堆或大根堆，根结点最小的堆叫做最小堆或小根堆。

堆的性质：

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



选择题

```
1.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

2.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

3.一组记录排序码为(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)

4.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
```

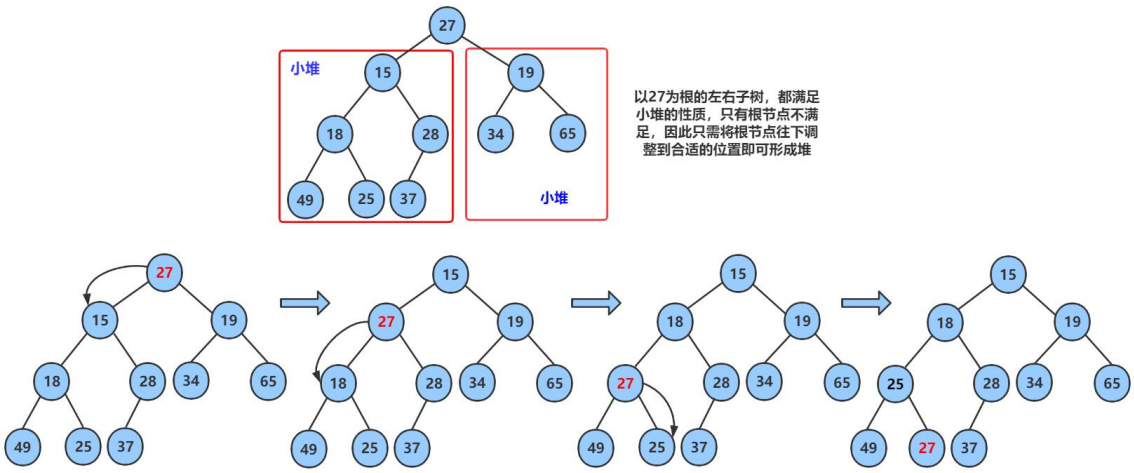
3.3 堆的实现

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。我们通过从根结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 array[] = {27,15,19,18,28,34,65,49,25,37};
```

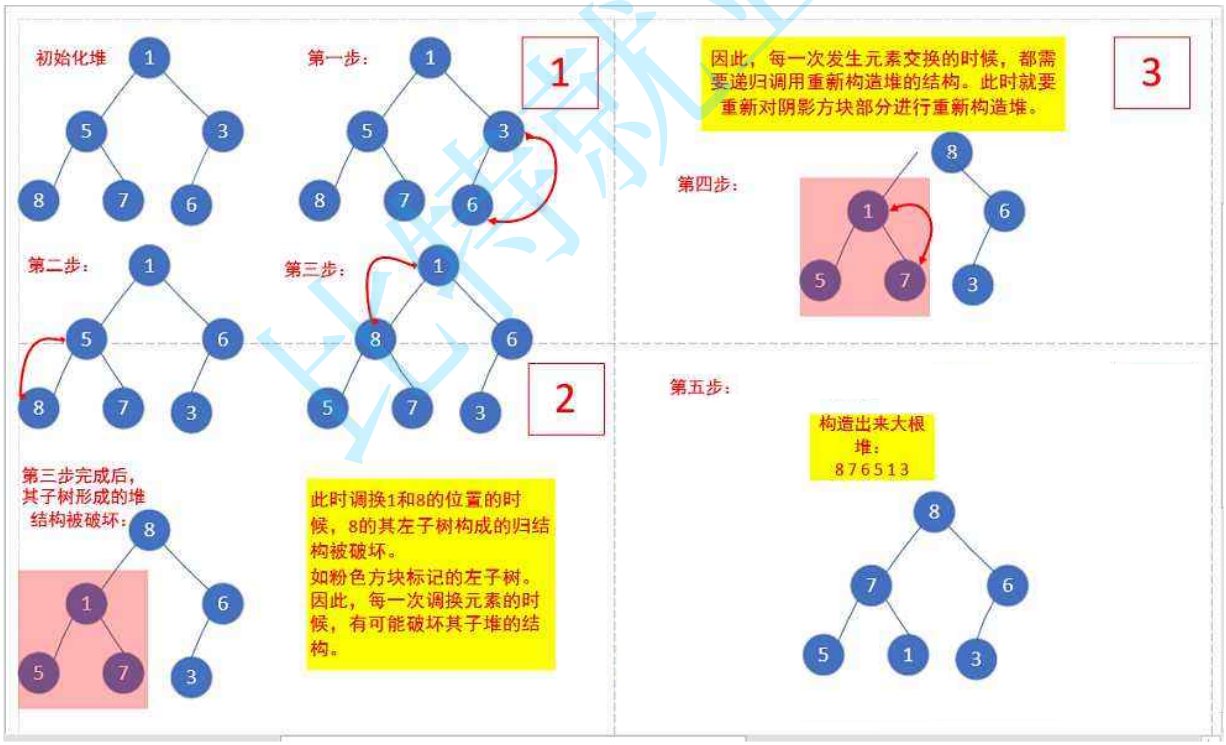
比特就业课



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。根结点左右子树不是堆，我们怎么调整呢？这里我们从倒数的第一个非叶子结点的子树开始调整，一直调整到根结点的树，就可以调整成堆。

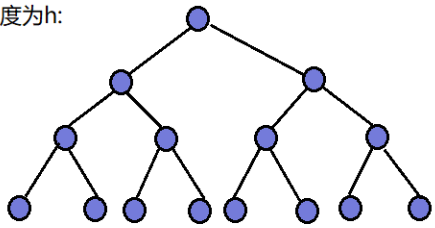
```
1  int a[] = {1,5,3,8,7,6};
```



3.2.3 建堆时间复杂度

因为堆是完全二叉树，而满二叉树也是完全二叉树，此处为了简化使用满二叉树来证明(时间复杂度本来看的就是近似值，多几个结点不影响最终结果)：

假设树的高度为h:



第1层，2^0个节点，需要向下移动h-1层
第2层，2^1个节点，需要向下移动h-2层
第3层，2^2个节点，需要向下移动h-3层
第4层，2^3个节点，需要向下移动h-4层
.....
第h-1层，2^{h-2}个节点，需要向下移动1层

则需要移动节点总的移动步数为：

$$T(n) = 2^0 * (h-1) + 2^1 * (h-2) + 2^2 * (h-3) + 2^3 * (h-4) + ... + 2^{h-3} * 2 + 2^{h-2} * 1 \quad ①$$

$$2 * T(n) = 2^1 * (h-1) + 2^2 * (h-2) + 2^3 * (h-3) + 2^4 * (h-4) + ... + 2^{h-2} * 2 + 2^{h-1} * 1 \quad ②$$

②－① 错位相减：

$$T(n) = 1 - h + 2^1 + 2^2 + 2^3 + 2^4 + ... + 2^{h-2} + 2^{h-1}$$

$$T(n) = 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + ... + 2^{h-2} + 2^{h-1} - h$$

$$T(n) = 2^h - 1 - h$$

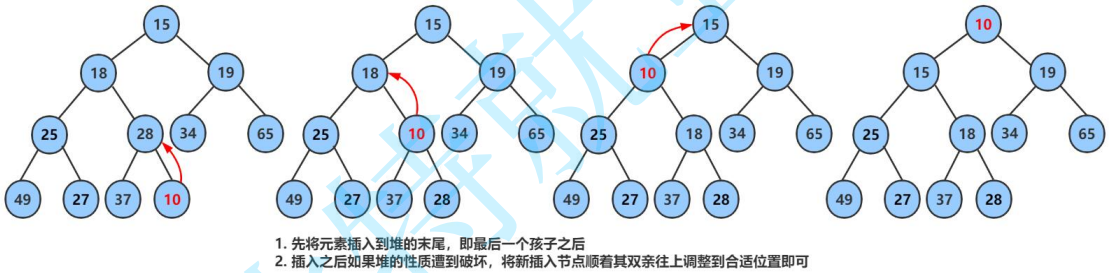
$$n = 2^h - 1 \qquad h = \log_2(n+1)$$

$$T(n) = n - \log_2(n+1) \approx n$$

因此：**建堆的时间复杂度为O(N)。**

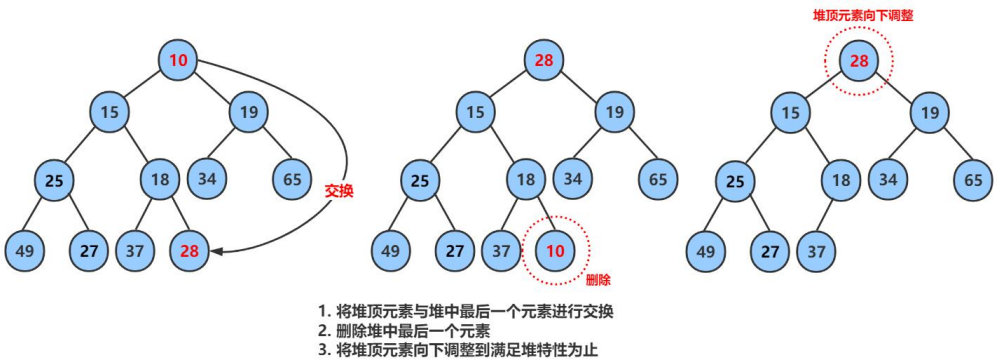
3.2.4 堆的插入

先插入一个10到数组的尾上，再进行向上调整算法，直到满足堆。



3.2.5 堆的删除

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



3.2.6 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);
```

3.4 堆的应用

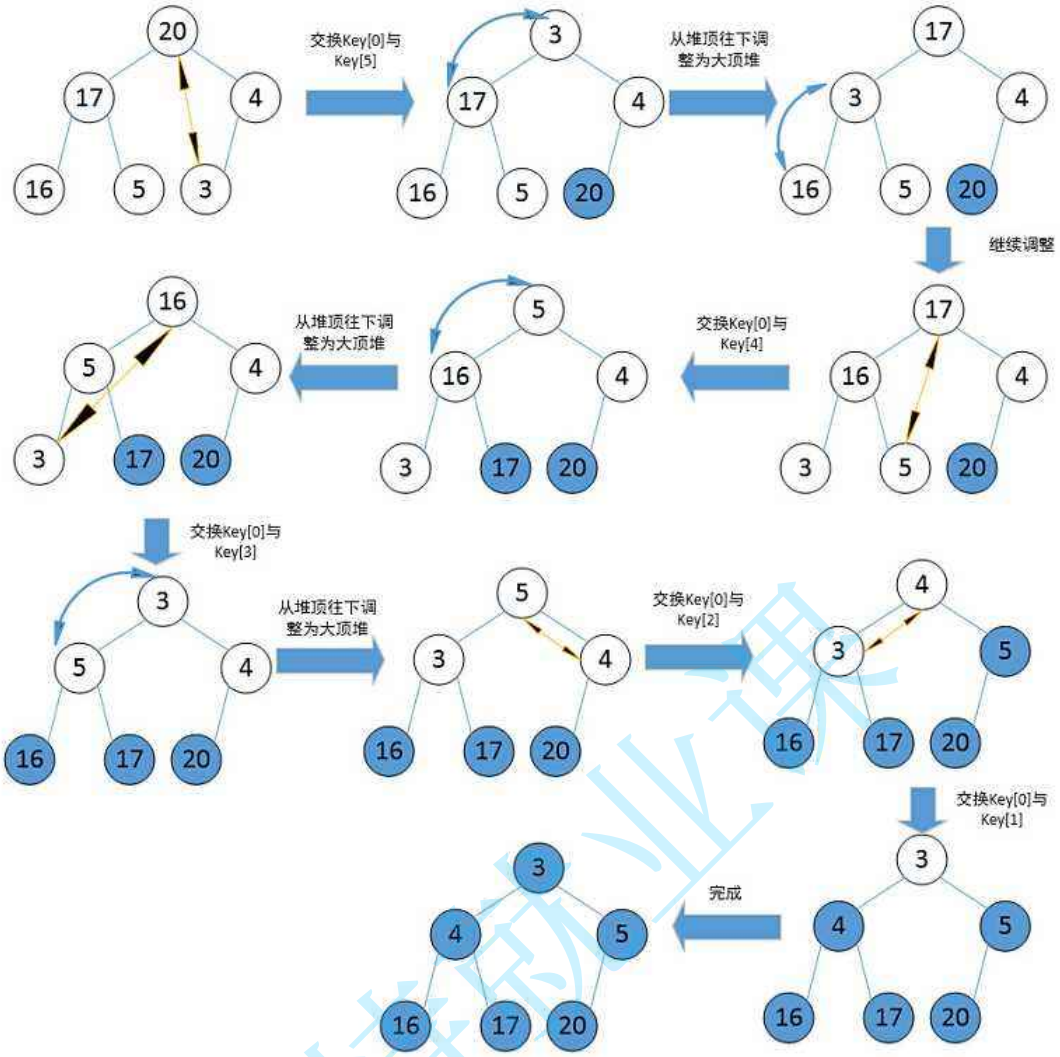
3.4.1 堆排序

堆排序即利用堆的思想来进行排序，总共分为两个步骤：

1. **建堆**
    - 升序：建大堆
    - 降序：建小堆
2. **利用堆删除思想来进行排序**

建堆和堆删除中都用到了向下调整，因此掌握了向下调整，就可以完成堆排序。

比特就业课



3.24.2 TOP-K问题

TOP-K问题：即求数据结合中前K个最大的元素或者最小的元素，一般情况下数据量都比较大。

比如：专业前10名、世界500强、富豪榜、游戏中前100的活跃玩家等。

对于Top-K问题，能想到的最简单直接的方式就是排序，但是：如果数据量非常大，排序就不太可取了(可能数据都不能一下子全部加载到内存中)。最佳的方式就是用堆来解决，基本思路如下：

1. **用数据集合中前K个元素来建堆**
   - 前k个最大的元素，则建小堆
   - 前k个最小的元素，则建大堆
2. **用剩余的N-K个元素依次与堆顶元素来比较，不满足则替换堆顶元素**

将剩余N-K个元素依次与堆顶元素比完之后，堆中剩余的K个元素就是所求的前K个最小或者最大的元素。

```
void PrintTopK(int* a, int n, int k)
{
    // 1. 建堆--用a中前k个元素建堆

    // 2. 将剩余n-k个元素依次与堆顶元素交换，不满则则替换
}
```

```
7
8  void TestTopk()
9  {
10     int n = 10000;
11     int* a = (int*)malloc(sizeof(int)*n);
12     srand(time(0));
13     for (size_t i = 0; i < n; ++i)
14     {
15         a[i] = rand() % 1000000;
16     }
17     a[5] = 1000000 + 1;
18     a[1231] = 1000000 + 2;
19     a[531] = 1000000 + 3;
20     a[5121] = 1000000 + 4;
21     a[115] = 1000000 + 5;
22     a[2335] = 1000000 + 6;
23     a[9999] = 1000000 + 7;
24     a[76] = 1000000 + 8;
25     a[423] = 1000000 + 9;
26     a[3144] = 1000000 + 10;
27      PrintTopK(a, n, 10);
28 }
```

4.二叉树链式结构的实现

4.1 前置说明

在学习二叉树的基本操作前，需先要创建一棵二叉树，然后才能学习其相关的基本操作。由于现在大家对二叉树结构掌握还不够深入，为了降低大家学习成本，此处手动快速创建一棵简单的二叉树，快速进入二叉树操作学习，等二叉树结构了解的差不多时，我们反过头再来研究二叉树真正的创建方式。

```
1  typedef int BTDataType;
2  typedef struct BinaryTreeNode
3  {
4      BTDataType _data;
5      struct BinaryTreeNode* _left;
6      struct BinaryTreeNode* _right;
7  }BTNode;
8
9  BTNode* CreatBinaryTree()
10 {
11     BTNode* node1 = BuyNode(1);
12     BTNode* node2 = BuyNode(2);
13     BTNode* node3 = BuyNode(3);
14     BTNode* node4 = BuyNode(4);
15     BTNode* node5 = BuyNode(5);
16     BTNode* node6 = BuyNode(6);
17
18     node1->_left = node2;
19     node1->_right = node4;
20     node2->_left = node3;
21     node4->_left = node5;
22     node4->_right = node6;
```

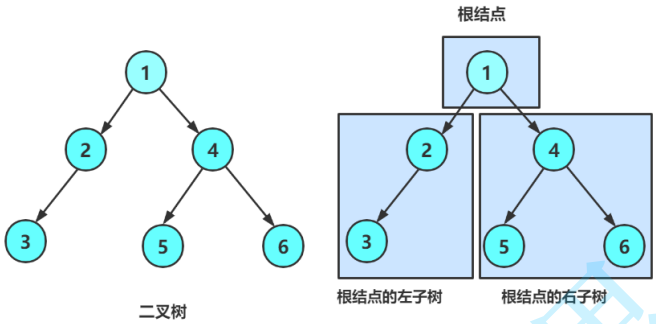
比特就业课

```
23      return node1;
24  }
```

注意：上述代码并不是创建二叉树的方式，真正创建二叉树方式后序详解重点讲解。

再看二叉树基本操作前，再回顾下二叉树的概念，**二叉树是：**

1. **空树**
2. **非空：根结点，根结点的左子树、根结点的右子树组成的。**

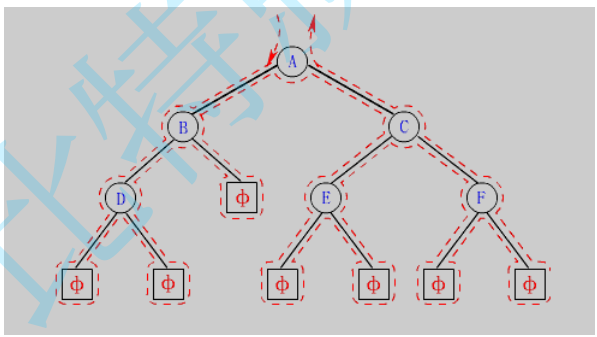


从概念中可以看出，二叉树定义是递归式的，因此后序基本操作中基本都是按照该概念实现的。

4.2二叉树的遍历

4.2.1 前序、中序以及后序遍历

学习二叉树结构，最简单的方式就是遍历。所谓**二叉树遍历(Traversal)是按照某种特定的规则，依次对二叉树中的结点进行相应的操作，并且每个结点只操作一次。**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，也是二叉树上进行其它运算的基础。



按照规则，二叉树的遍历有：**前序/中序/后序的递归结构遍历**：

1. 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. 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. 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。

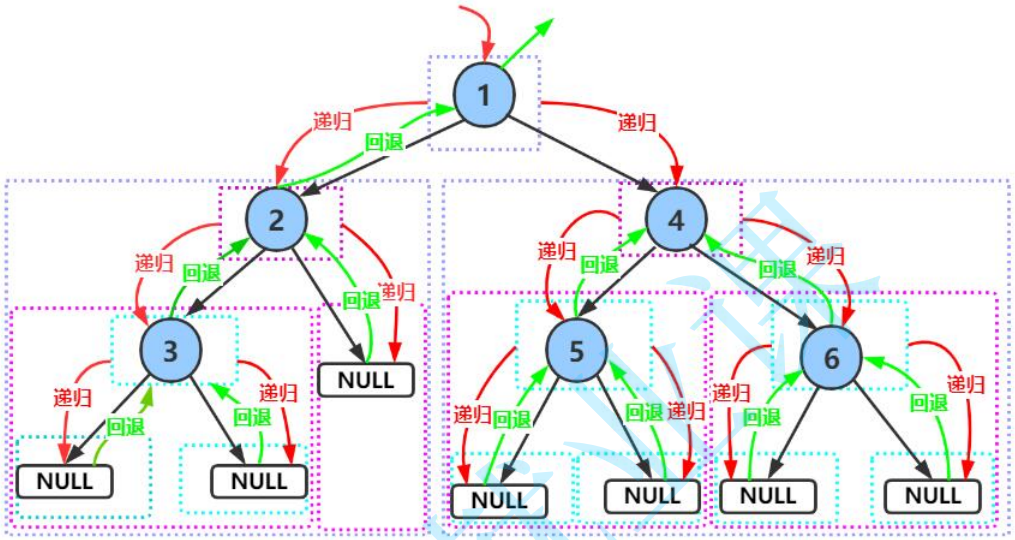
由于被访问的结点必是某子树的根，**所以N(Node)、L(Left subtree)和R(Right subtree)又可解释为根、根的左子树和根的右子树。**NLR、LNR和LRN分别又称为先根遍历、中根遍历和后根遍历。

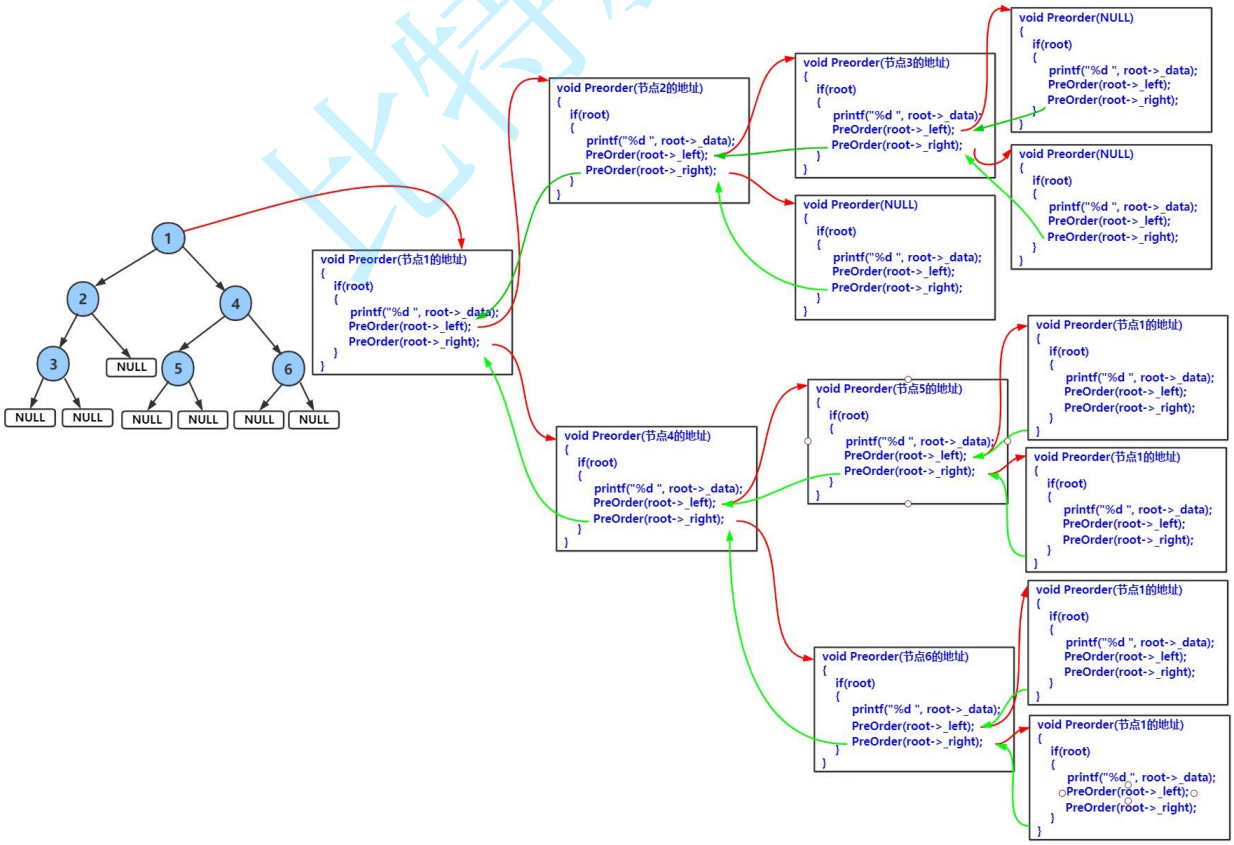
比特就业课

```c
// 二叉树前序遍历
void PreOrder(BTNode* root);
// 二叉树中序遍历
void InOrder(BTNode* root);
// 二叉树后序遍历
void PostOrder(BTNode* root);
```

下面主要分析前序递归遍历，中序与后序图解类似，同学们可自己动手绘制。

前序遍历递归图解：

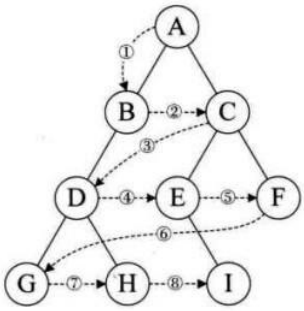




> **前序遍历结果：1 2 3 4 5 6**
>
> **中序遍历结果：3 2 1 5 4 6**
>
> **后序遍历结果：3 2 5 6 4 1**

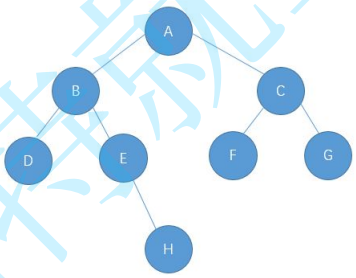
4.2.2 层序遍历

> **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。设二叉树的根结点所在层数为1，层序遍历就是从所在二叉树的根结点出发，首先访问第一层的树根结点，然后从左到右访问第2层上的结点，接着是第三层的结点，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。




```
// 层序遍历
void LevelOrder(BTNode* root);
```

> **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.设一课二叉树的中序遍历序列：badce，后序遍历序列：bdeca，则二叉树前序遍历序列为____。
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
```

比特就业课

```
为
A FEDCBA
B CBAFED
C DEFCBA
D ABCDEF
```

选择题答案

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

4.3 结点个数以及高度等

```
// 二叉树结点个数
int BinaryTreeSize(BTNode* root);
// 二叉树叶子结点个数
int BinaryTreeLeafSize(BTNode* root);
// 二叉树第k层结点个数
int BinaryTreeLevelKSize(BTNode* root, int k);
// 二叉树查找值为x的结点
BTNode* BinaryTreeFind(BTNode* root, BTDataType x);
```

4.4 二叉树基础oj练习

1. 单值二叉树。OJ链接
2. 检查两颗树是否相同。OJ链接
3. 对称二叉树。OJ链接
4. 二叉树的前序遍历。 OJ链接
5. 二叉树中序遍历 。OJ链接
6. 二叉树的后序遍历 。OJ链接
7. 另一颗树的子树。OJ链接

4.5 二叉树的创建和销毁

二叉树的构建及遍历。OJ链接

```
// 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);
// 判断二叉树是否是完全二叉树
int BinaryTreeComplete(BTNode* root);
```

比特就业课