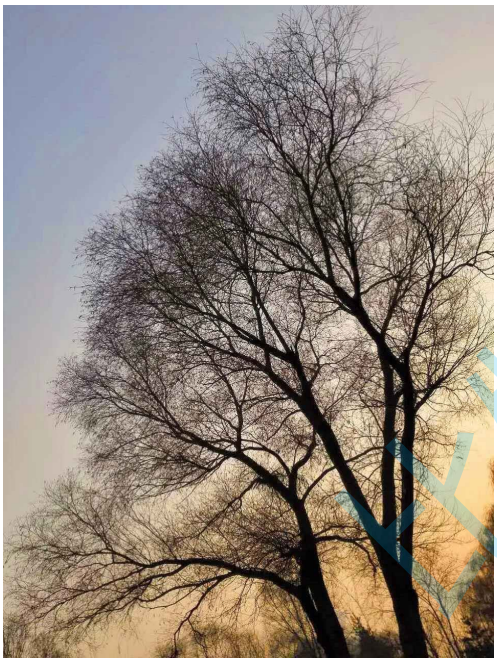
4 二叉树

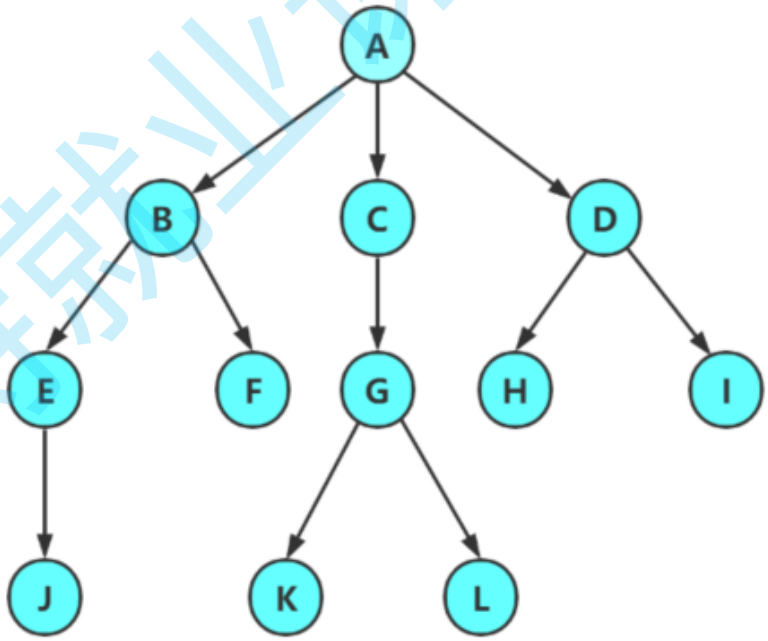
1. 树

1.1 树的概念与结构

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。

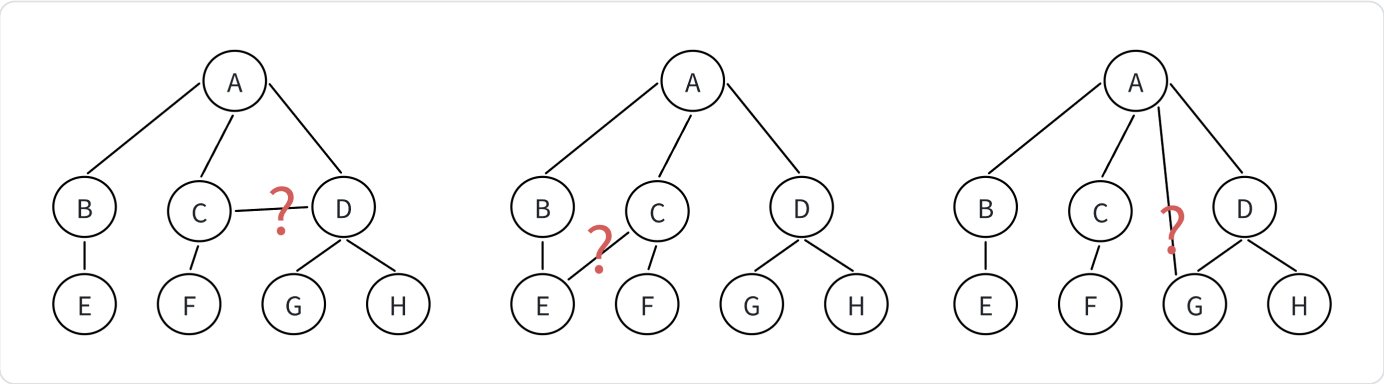
- 有一个特殊的结点，称为根结点，根结点没有前驱结点。
- 除根结点外，其余结点被分成 `M(M>0)` 个互不相交的集合 `T1、T2、……、Tm` ，其中每一个集合 `Ti(1 <= i <= m)` 又是一棵结构与树类似的子树。每棵子树的根结点有且只有一个前驱，可以有 `0` 个或多个后继。因此，树是递归定义的。





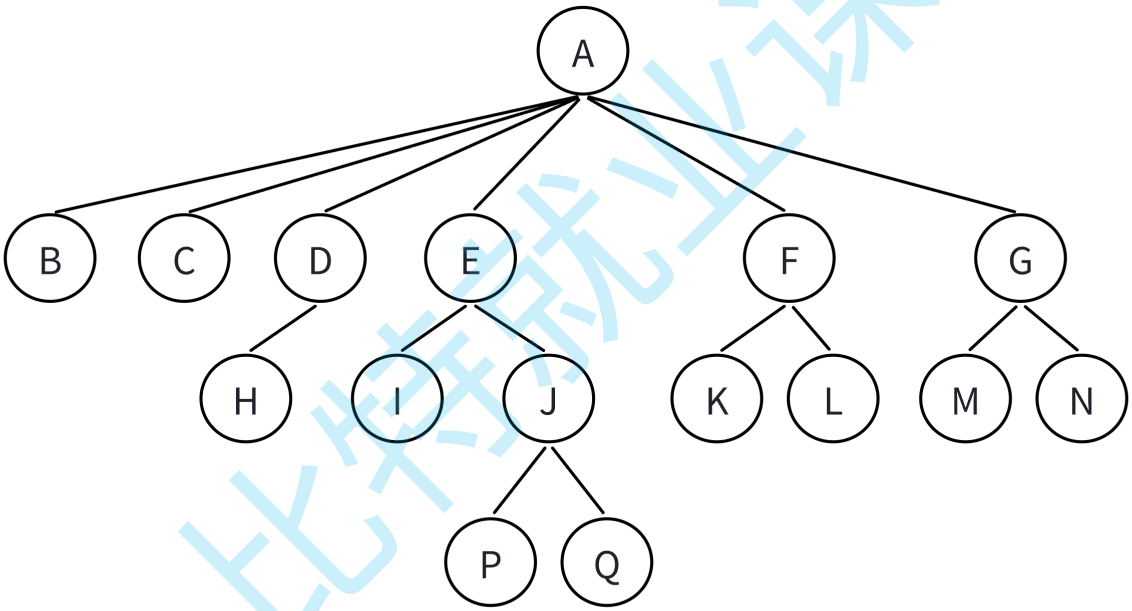
> 树形结构中，子树之间不能有交集，否则就不是树形结构

非树形结构：



- 子树是不相交的（如果存在相交就是图了，图以后得课程会有讲解）
- 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

1.2 树相关术语



父结点/双亲结点：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

子结点/孩子结点：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

结点的度：一个结点有几个孩子，他的度就是多少；比如A的度为6，F的度为2，K的度为0

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`

叶子结点/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

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

兄弟结点：具有相同父结点的结点互称为兄弟结点(亲兄弟)； 如上图： `B、C` 是兄弟结点

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`

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

路径：一条从树中任意节点出发，沿父节点-子节点连接，达到任意节点的序列；比如A到Q的路径为：A-E-J-Q；H到Q的路径H-D-A-E-J-Q

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

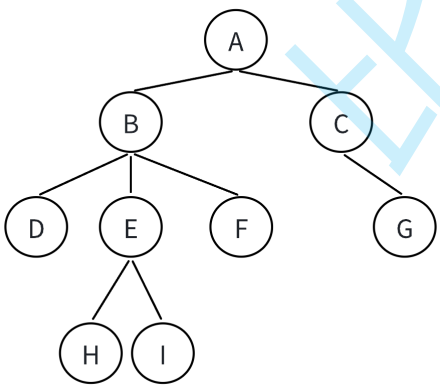
森林：由 `m (m>0)` 棵互不相交的树的集合称为森林；

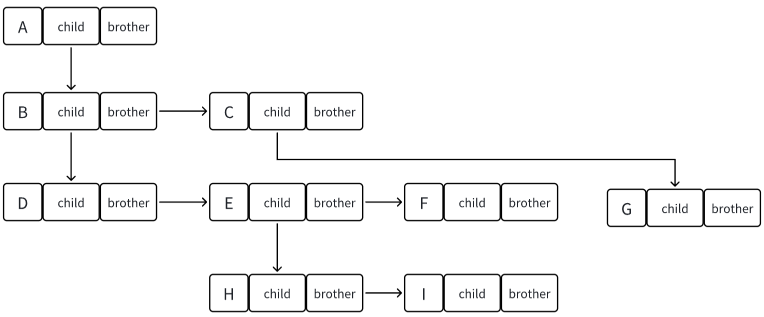
1.3 树的表示

孩子兄弟表示法：

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

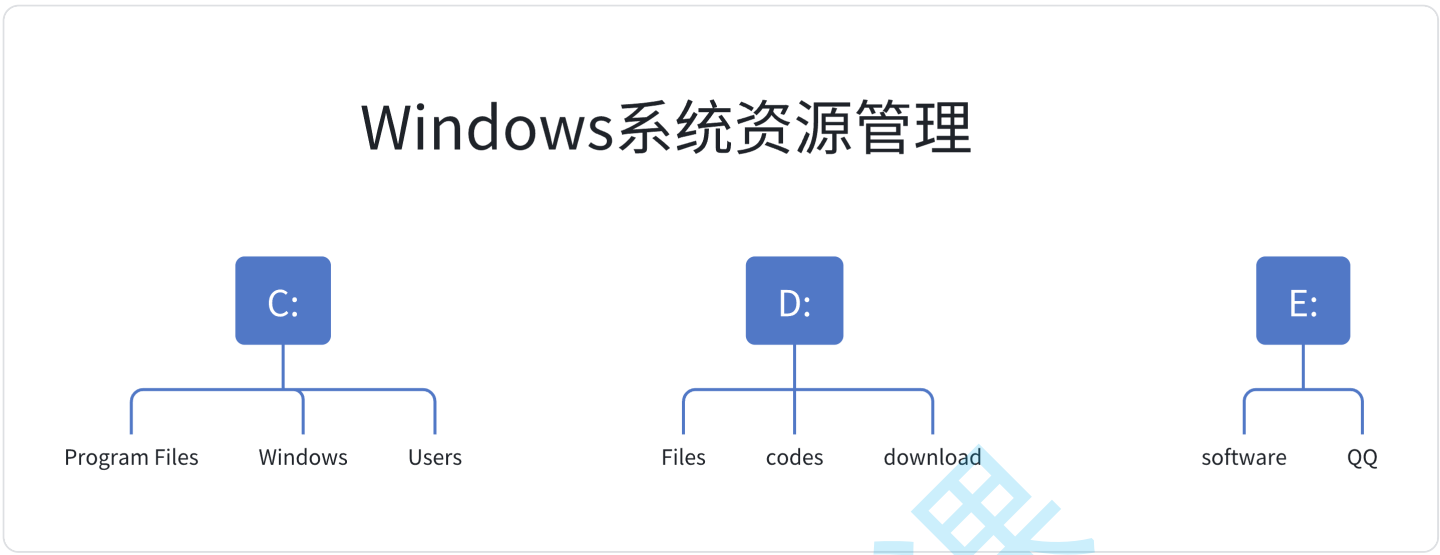
```
struct TreeNode
{
    struct Node* child;   // 左边开始的第一个孩子结点
    struct Node* brother; // 指向其右边的下一个兄弟结点
    int data;             // 结点中的数据域
};
```





1.4 树形结构实际运用场景

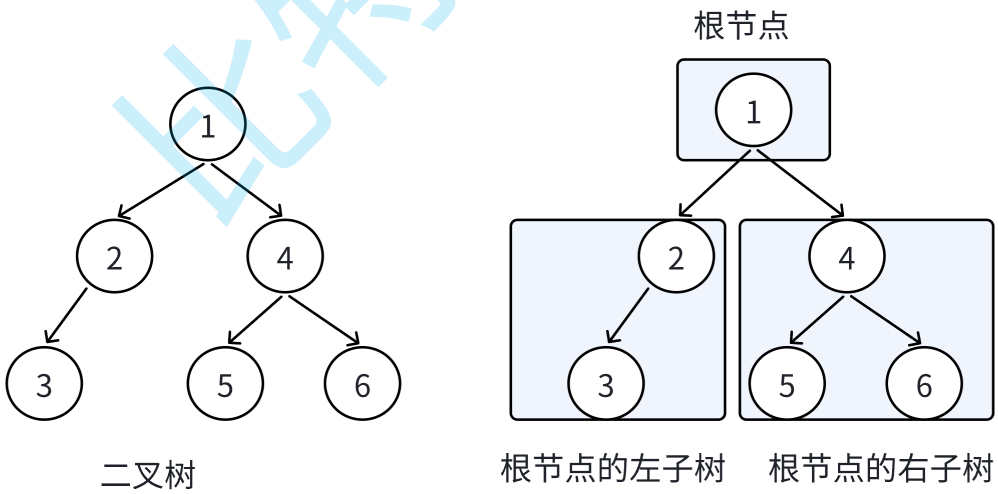
文件系统是计算机存储和管理文件的一种方式，它利用树形结构来组织和管理文件和文件夹。在文件系统中，树结构被广泛应用，它通过父结点和子结点之间的关系来表示不同层级的文件和文件夹之间的关联。



2. 二叉树

2.1 概念与结构

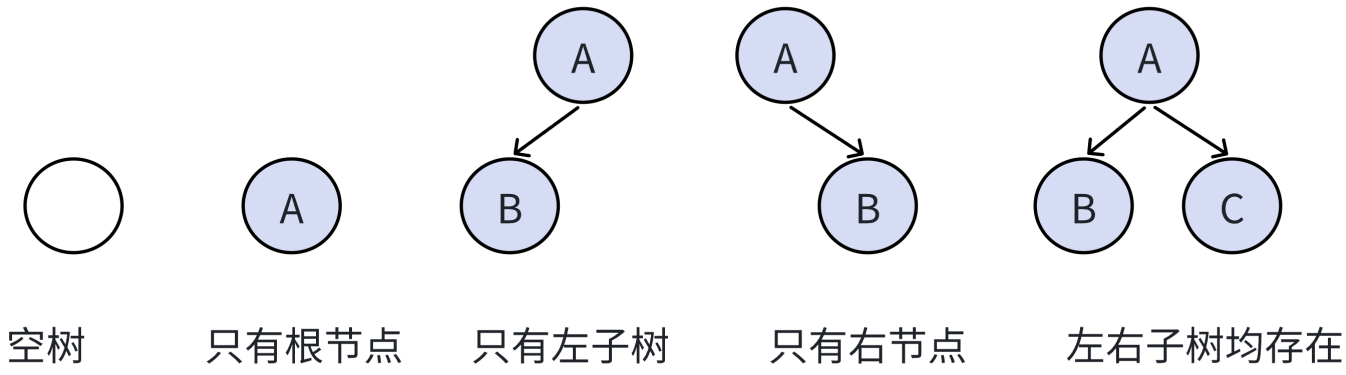
在树形结构中，我们最常用的就是二叉树，一棵二叉树是结点的一个有限集合，该集合由一个根结点加上两棵别称为左子树和右子树的二叉树组成或者为空。



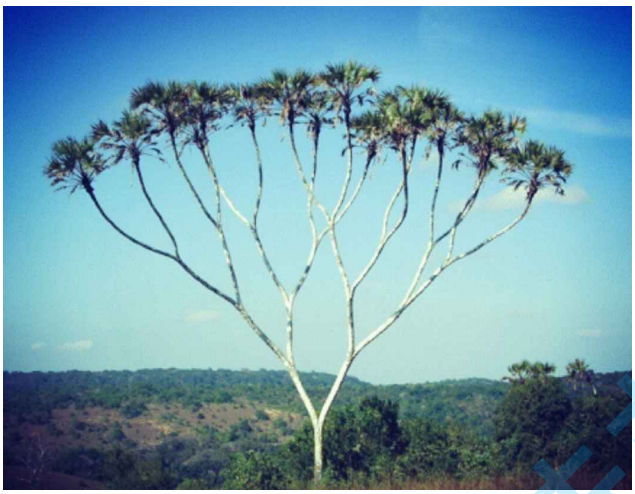
从上图可以看出二叉树具备以下特点：

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



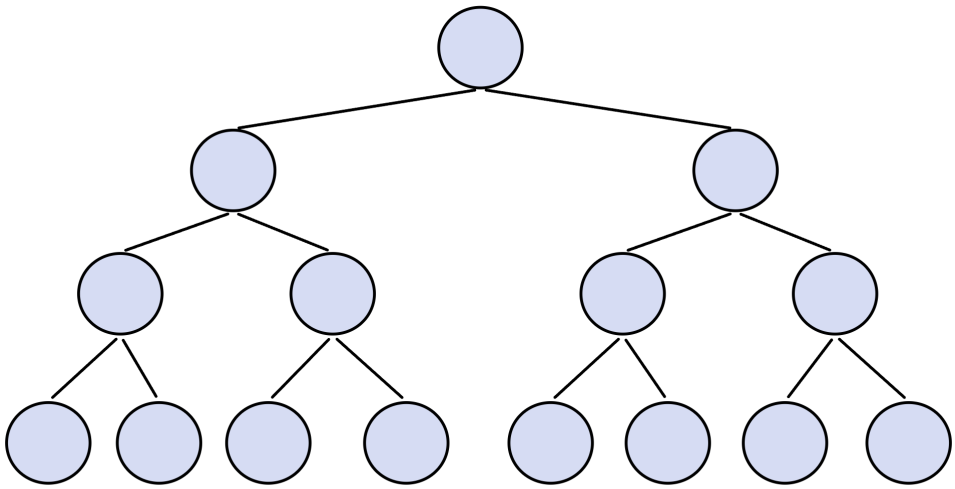
> 现实中的二叉树




2.2 特殊的二叉树

2.2.1 满二叉树

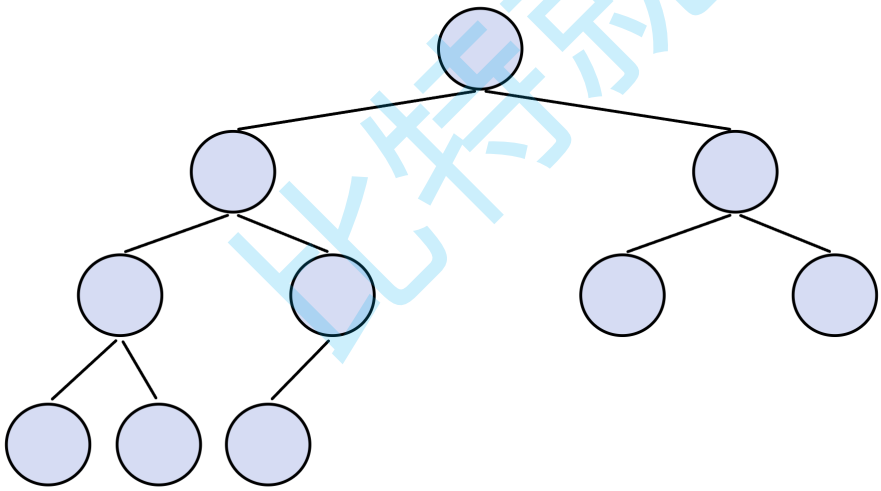
一个二叉树，如果每一个层的结点数都达到最大值，则这个二叉树就是满二叉树。也就是说，如果一个二叉树的层数为 `K` ，且结点总数是 $2^k - 1$ ，则它就是满二叉树。



满二叉树

2.2.2 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。



完全二叉树

💡 二叉树性质

根据满二叉树的特点可知：

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

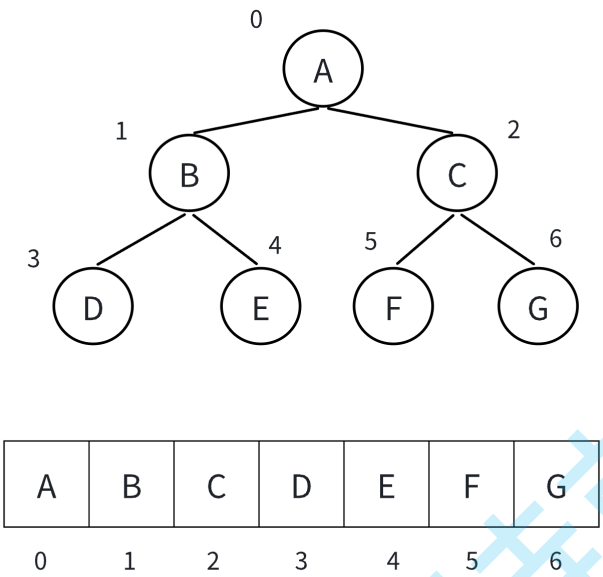
3）若规定根结点的层数为 `1` ，具有 `n` 个结点的满二叉树的深度 $h = log_2(n+1)$ (`log` 以2为底， `n+1` 为对数)

2.3 二叉树存储结构

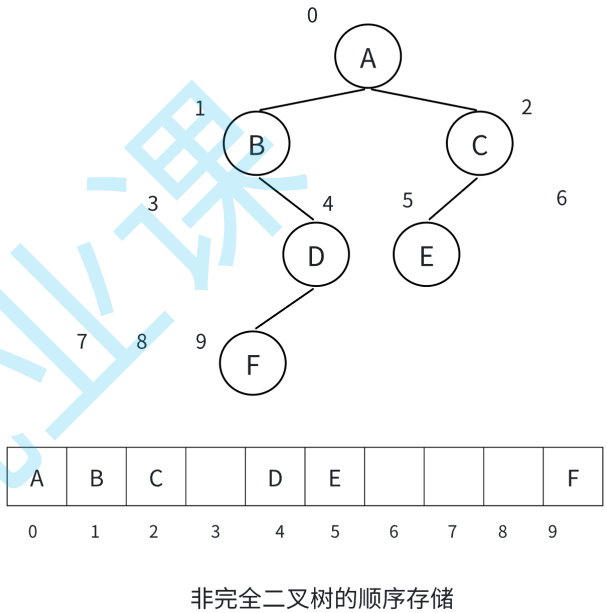
二叉树一般可以使用两种结构存储，一种顺序结构，一种链式结构。

2.3.1 顺序结构

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，完全二叉树更适合使用顺序结构存储。



完全二叉树的顺序存储

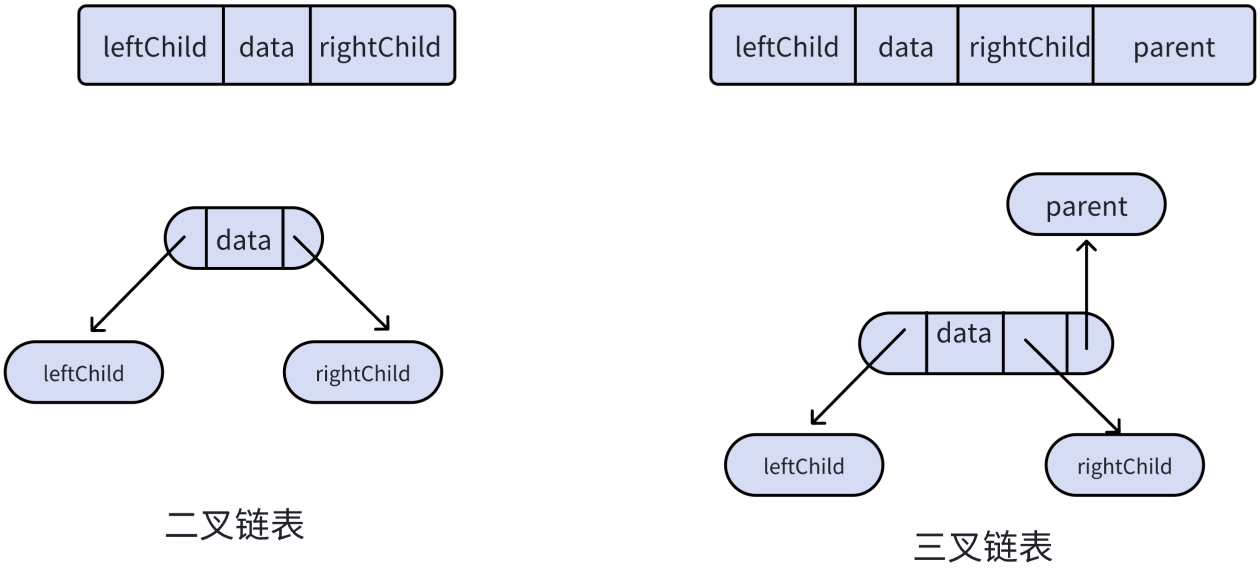


非完全二叉树的顺序存储

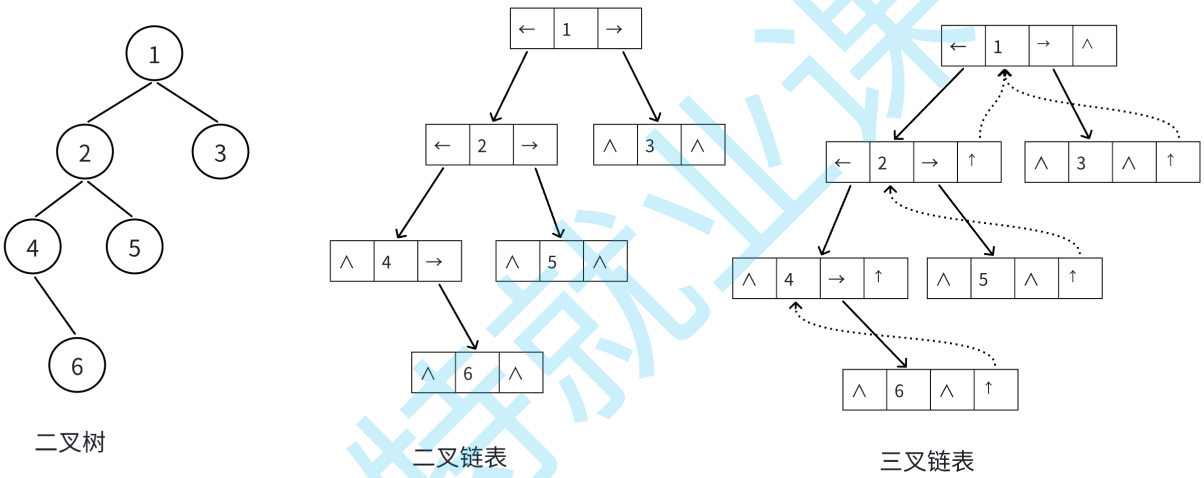
现实中我们通常把堆（一种二叉树）使用顺序结构的数组来存储，需要注意的是这里的堆和操作系统虚拟进程地址空间中的堆是两回事，一个是数据结构，一个是操作系统中管理内存的一块区域分段。

2.3.2 链式结构

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。



二叉链表　　三叉链表

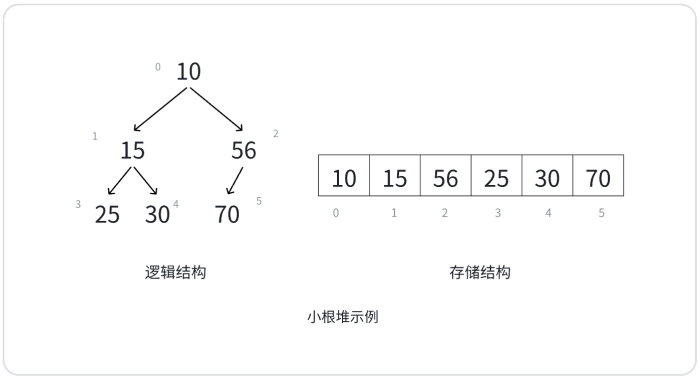


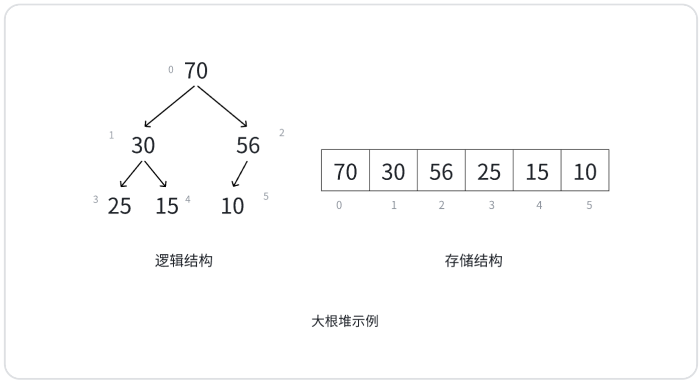
3. 实现顺序结构二叉树

一般堆使用顺序结构的数组来存储数据，堆是一种特殊的二叉树，具有二叉树的特性的同时，还具备其他的特性。

3.1 堆的概念与结构

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：$K_i <= K_{2*i+1}$ （$K_i >= K_{2*i+1}$ 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。





> 堆具有以下性质

- 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。

> 二叉树性质
>
> - 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

3.2 堆的实现

堆底层结构为数组，因此定义堆的结构为：

```c
typedef int HPDataType;

typedef struct Heap
{
    HPDataType* a;
    int size;
    int capacity;
}HP;
//默认初始化堆
void HPInit(HP* php);
//利用给定数组初始化堆
void HPInitArray(HP* php, HPDataType* a, int n);
//堆的销毁
void HPDestroy(HP* php);

//堆的插入
void HPPush(HP* php, HPDataType x);
```

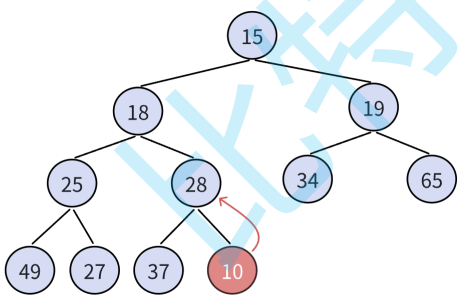
```
//堆的删除
HPDataType HPTop(HP* php);
// 删除堆顶的数据
void HPPop(HP* php);
// 判空
bool HPEmpty(HP* php);
//求size
int HPSize(HP* php);
//向上调整算法
void AdjustUp(HPDataType* a, int child);
//向下调整算法
void AdjustDown(HPDataType* a, int n, int parent);
```

3.2.1 向上调整算法

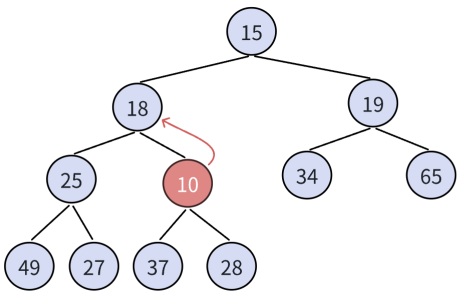
> **堆的插入**

将新数据插入到数组的尾上，再进行向上调整算法，直到满足堆。

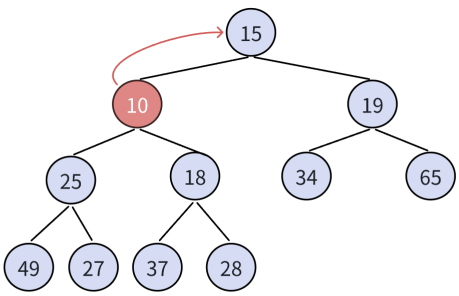
> 向上调整算法
>
> - 先将元素插入到堆的末尾,即最后一个孩子之后
> - 插入之后如果堆的性质遭到破坏，将新插入结点顺着其双双亲往上调整到合适位置即可



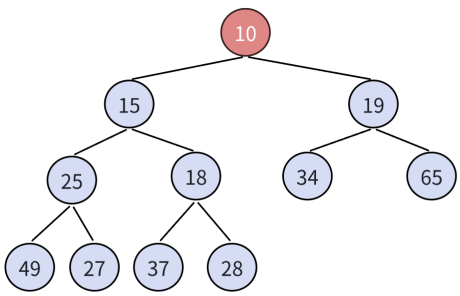
第一步



第二步



第三步



第四步

```
void AdjustUp(HPDataType* a, int child)
```

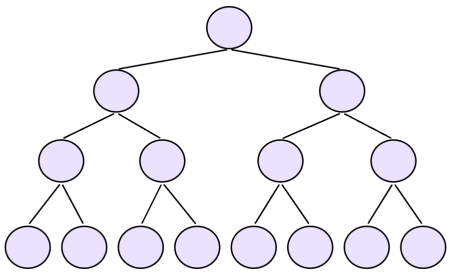
```
{
    int parent = (child - 1) / 2;
    while(child > 0)
    {
        if (a[child] > a[parent])
        {
            Swap(&a[child], &a[parent]);
            child = parent;
            parent = (parent - 1) / 2;
        }
        else
        {
            break;
        }
    }
}
void HPPush(HP* php, HPDataType x)
{
    assert(php);

    if (php->size == php->capacity)
    {
        size_t newCapacity = php->capacity == 0 ? 4 : php->capacity * 2;
        HPDataType* tmp = realloc(php->a, sizeof(HPDataType) * newCapacity);
        if (tmp == NULL)
        {
            perror("realloc fail");
            return;
        }
        php->a = tmp;
        php->capacity = newCapacity;
    }

    php->a[php->size] = x;
    php->size++;

    AdjustUp(php->a, php->size-1);
}
```

> 计算向上调整算法建堆时间复杂度

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)

分析：

第1层， 2^0 个结点，需要向上移动0层



第2层，2^1 个结点，需要向上移动1层

第3层，2^2 个结点，需要向上移动2层

第4层，2^3 个结点，需要向上移动3层

.....

第h层，2^{h-1} 个结点，需要向上移动h-1层

则需要移动结点总的移动步数为：每层结点个数 * 向上调整次数（第一层调整次数为0）

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

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

② — ① 错位相减：

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

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

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

$$T(h) = -(2^h - 1) + 2^h * (h-1) + 2^0$$

$$T(h) = -(2^h - 1) + 2^h * (h-1) + 2^0$$

根据二叉树的性质：$n = 2^h - 1$ 和 $h = log_2(n+1)$

$$T(n) = -N + 2^h * (h-1) + 2^0$$

$$F(h) = 2^h(h-2) + 2$$

$$F(n) = (n+1)(\log_2(n+1) - 2) + 2$$

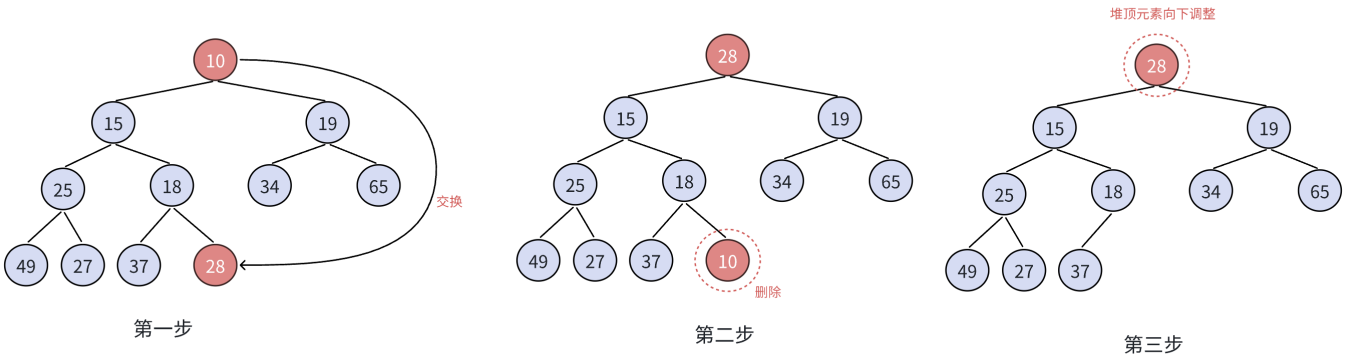
由此可得：

> 💡 向上调整算法建堆时间复杂度为：$O(n * \log_2 n)$

3.2.2 向下调整算法

> **堆的删除**

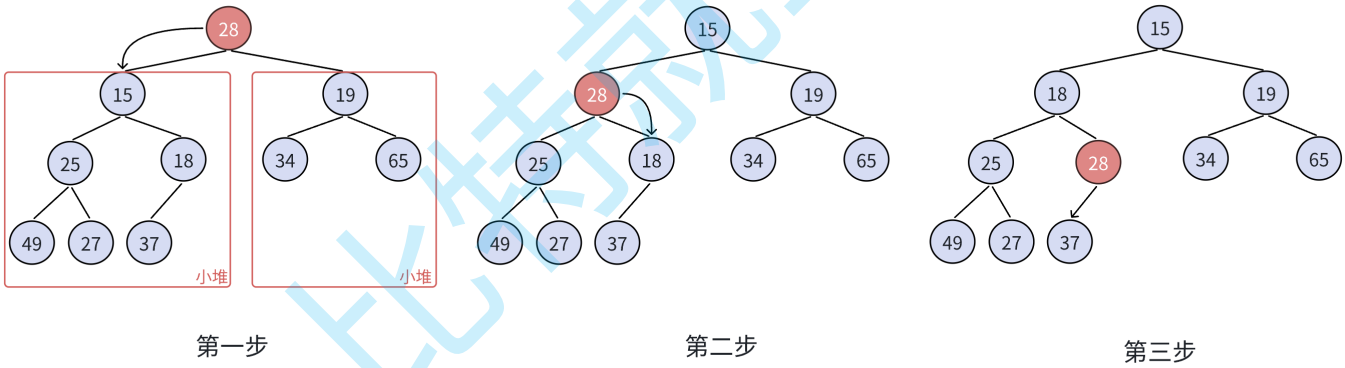
删除堆是删除堆顶的数据，将堆顶的数据根最后一个数据一换，然后删除数组最后一个数据，再进行向下调整算法。



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。

向下调整算法

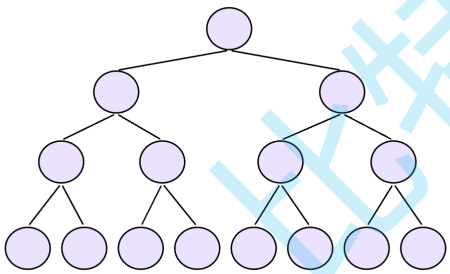
- 将堆顶元素与堆中最后一个元素进行交换
- 删除堆中最后一个元素
- 将堆顶元素向下调整到满足堆特性为止




```
1  void AdjustDown(HPDataType* a, int n, int parent)
2  {
3      int child = parent * 2 + 1;
4      while (child < n)
5      {
6          // 假设法，选出左右孩子中小的那个孩子
7          if (child+1 < n && a[child + 1] > a[child])
8          {
9              ++child;
10         }
11
12         if (a[child] > a[parent])
```

```
        {
            Swap(&a[child], &a[parent]);
            parent = child;
            child = parent * 2 + 1;
        }
        else
        {
            break;
        }
    }
}
void HPPop(HP* php)
{
    assert(php);
    assert(php->size > 0);

    Swap(&php->a[0], &php->a[php->size - 1]);
    php->size--;

    AdjustDown(php->a, php->size, 0);
}
```

> 计算向下调整算法建堆时间复杂度



分析：

第1层，2^0 个结点，需要向下移动h-1层

第2层，2^1 个结点，需要向下移动h-2层

第3层，2^2 个结点，需要向下移动h-3层

第4层，2^3 个结点，需要向下移动h-4层

.....

第h-1层，2^{h-2} 个结点，需要向下移动1层

则需要移动结点总的移动步数为：每层结点个数 * 向下调整次数

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

①

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

②

② — ① 错位相减：

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

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

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

根据二叉树的性质：$n = 2^h - 1$ 和 $h = log_2(n+1)$

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

> 💡 向下调整算法建堆时间复杂度为：$O(n)$

3.3 堆的应用

3.3.1 堆排序

> 版本一：基于已有数组建堆、取堆顶元素完成排序版本

```
// 1、需要堆的数据结构
// 2、空间复杂度 O(N)
void HeapSort(int* a, int n)
{
    HP hp;
    for(int i = 0; i < n; i++)
    {
        HPPush(&hp,a[i]);
    }

    int i = 0;
    while (!HPEmpty(&hp))
    {
        a[i++] = HPTop(&hp);

        HPPop(&hp);
    }

    HPDestroy(&hp);
}
```

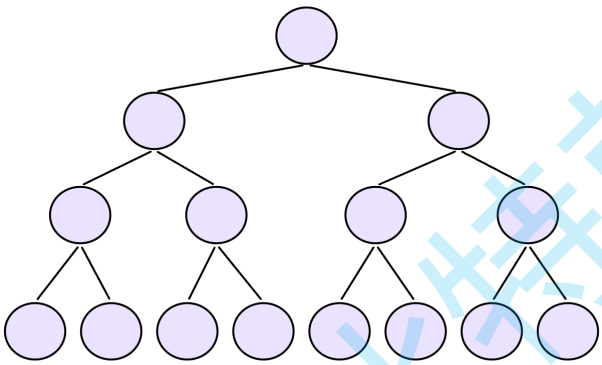
该版本有一个前提，必须提供有现成的数据结构堆

> 版本二：数组建堆，首尾交换，交换后的堆尾数据从堆中删掉，将堆顶数据向下调整选出次大的数据

```
// 升序，建大堆
// 降序，建小堆
// O(N*logN)
void HeapSort(int* a, int n)
```

```
{
    // a数组直接建堆 O(N)
    for (int i = (n-1-1)/2; i >= 0; --i)
    {
        AdjustDown(a, n, i);
    }

    // O(N*logN)
    int end = n - 1;
    while (end > 0)
    {
        Swap(&a[0], &a[end]);
        AdjustDown(a, end, 0);
        --end;
    }
}
```

> 堆排序时间复杂度计算



分析：

第1层，2^0 个结点，交换到根结点后，需要向下移动0层

第2层，2^1 个结点，交换到根结点后，需要向下移动1层

第3层，2^2 个结点，交换到根结点后，需要向下移动2层

第4层，2^3 个结点，交换到根结点后，需要向下移动3层

.....

第h层，2^{h-1} 个结点，交换到根结点后，需要向下移动h-1层

通过分析发现，堆排序第二个循环中的向下调整与建堆中的向上调整算法时间复杂度计算一致，此处不再赘述。因此，堆排序的时间复杂度为 $O(n + n * \log n)$ ，即 $O(n \log n)$

💡 堆排序时间复杂度为： $O(n \log n)$

3.3.2 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 CreateNDate()
{
    // 造数据
    int n = 100000;
    srand(time(0));
    const char* file = "data.txt";
    FILE* fin = fopen(file, "w");
    if (fin == NULL)
    {
        perror("fopen error");
        return;
    }

    for (int i = 0; i < n; ++i)
    {
        int x = (rand()+i) % 1000000;
        fprintf(fin, "%d\n", x);
    }

    fclose(fin);
}

void topk()
{
    printf("请输入k: >");
    int k = 0;
    scanf("%d", &k);

    const char* file = "data.txt";
    FILE* fout = fopen(file, "r");
    if (fout == NULL)
```

```
    {
        perror("fopen error");
        return;
    }

    int val = 0;
    int* minheap = (int*)malloc(sizeof(int) * k);
    if (minheap == NULL)
    {
        perror("malloc error");
        return;
    }

    for (int i = 0; i < k; i++)
    {
        fscanf(fout, "%d", &minheap[i]);
    }

    // 建k个数据的小堆
    for (int i = (k - 1 - 1) / 2; i >= 0; i--)
    {
        AdjustDown(minheap, k, i);
    }

    int x = 0;
    while (fscanf(fout, "%d", &x) != EOF)
    {
            // 读取剩余数据，比堆顶的值大，就替换他进堆
        if (x > minheap[0])
        {
            minheap[0] = x;
            AdjustDown(minheap, k, 0);
        }
    }

    for (int i = 0; i < k; i++)
    {
        printf("%d ", minheap[i]);
    }

    fclose(fout);

}
```

时间复杂度：$O(n) = k + (n-k)\log_2 k$

4. 实现链式结构二叉树

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 ，其结构如下：

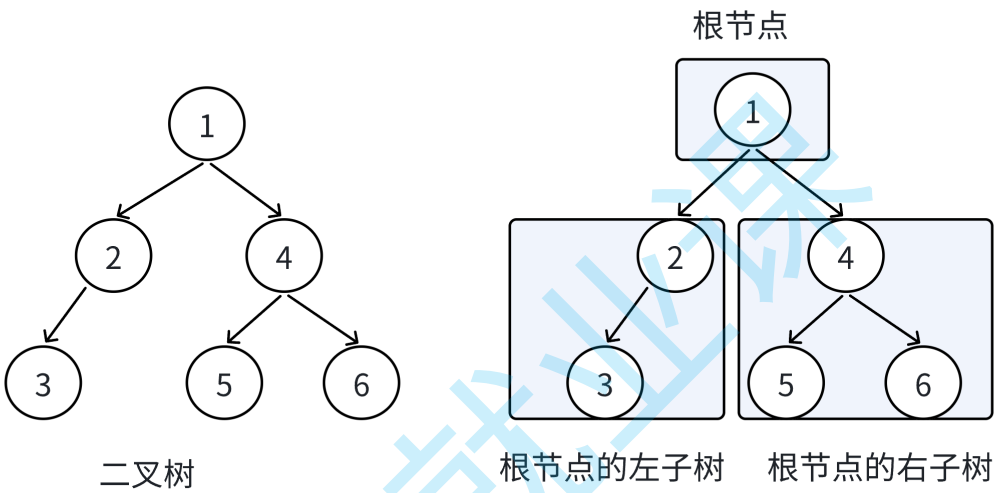
```
typedef int BTDataType;
// 二叉链
typedef struct BinaryTreeNode
{
    struct BinTreeNode* left; // 指向当前结点左孩子
    struct BinTreeNode* right; // 指向当前结点右孩子
    BTDataType val; // 当前结点值域
}BTNode;
```

二叉树的创建方式比较复杂，为了更好的步入到二叉树内容中，我们先手动创建一棵链式二叉树

```
BTNode* BuyBTNode(int val)
{
    BTNode* newnode = (BTNode*)malloc(sizeof(BTNode));
    if (newnode == NULL)
    {
        perror("malloc fail");
        return NULL;
    }
    newnode->val = val;
    newnode->left = NULL;
    newnode->right = NULL;
    return newnode;
}
BTNode* CreateTree()
{
    BTNode* n1 = BuyBTNode(1);
    BTNode* n2 = BuyBTNode(2);
    BTNode* n3 = BuyBTNode(3);
    BTNode* n4 = BuyBTNode(4);
    BTNode* n5 = BuyBTNode(5);
    BTNode* n6 = BuyBTNode(6);
    BTNode* n7 = BuyBTNode(7);


    n1->left = n2;
    n1->right = n4;
```

```
27      n2->left = n3;
28      n4->left = n5;
29      n4->right = n6;
30      n5->left = n7;
31
32      return n1;
33  }
```

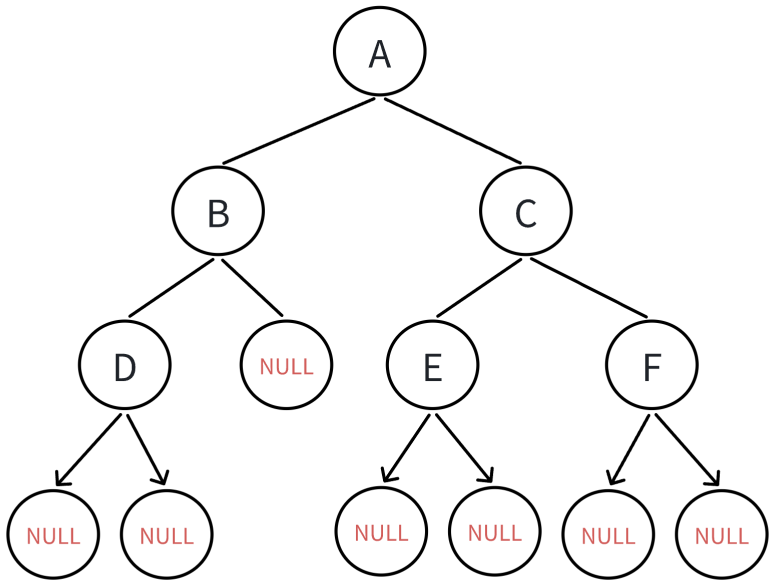
回顾二叉树的概念，二叉树分为空树和非空二叉树，非空二叉树由根结点、根结点的左子树、根结点的右子树组成的



根结点的左子树和右子树分别又是由子树结点、子树结点的左子树、子树结点的右子树组成的，因此二叉树定义是递归式的,后序链式二叉树的操作中基本都是按照该概念实现的。

4.1 前中后序遍历

二叉树的操作离不开树的遍历，我们先来看看二叉树的遍历有哪些方式



二叉树

4.1.1 遍历规则

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

> 访问顺序为：根结点、左子树、右子树

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）

> 访问顺序为：左子树、根结点、右子树

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

> 访问顺序为：左子树、右子树、根结点

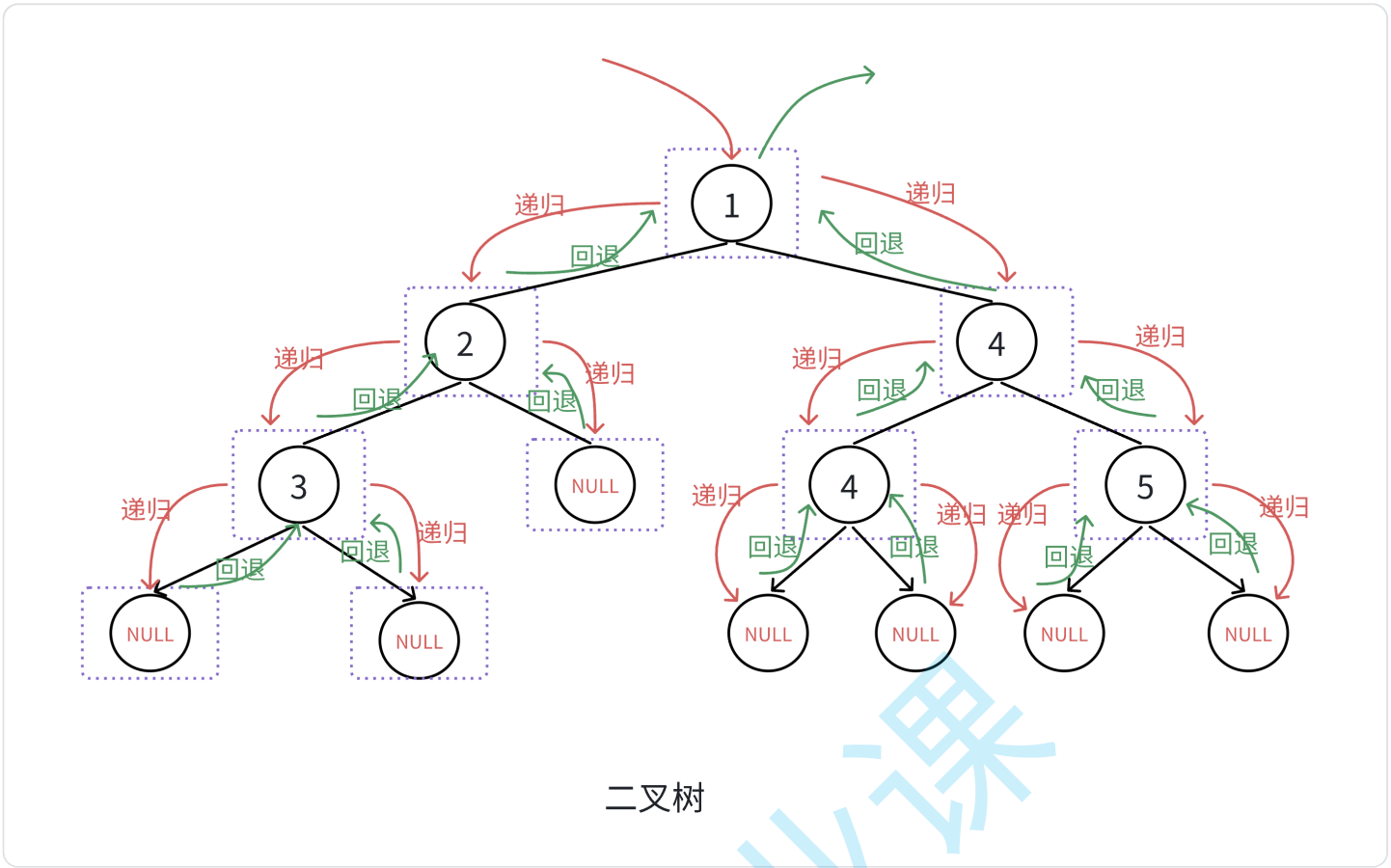
4.1.2 代码实现

```
1  void PreOrder(BTNode* root)
2  {
3      if (root == NULL)
4      {
5          printf("N ");
6          return;
7      }
8
9      printf("%d ", root->val);
10     PreOrder(root->left);
11     PreOrder(root->right);
12 }
13
14 void InOrder(BTNode* root)
```

```
{
    if (root == NULL)
    {
        printf("N ");
        return;
    }

    InOrder(root->left);
    printf("%d ", root->val);
    InOrder(root->right);
}

void PostOrder(BTNode* root)
{
    if (root == NULL)
    {
        printf("N ");
        return;
    }

    InOrder(root->left);
    InOrder(root->right);
    printf("%d ", root->val);
}
```

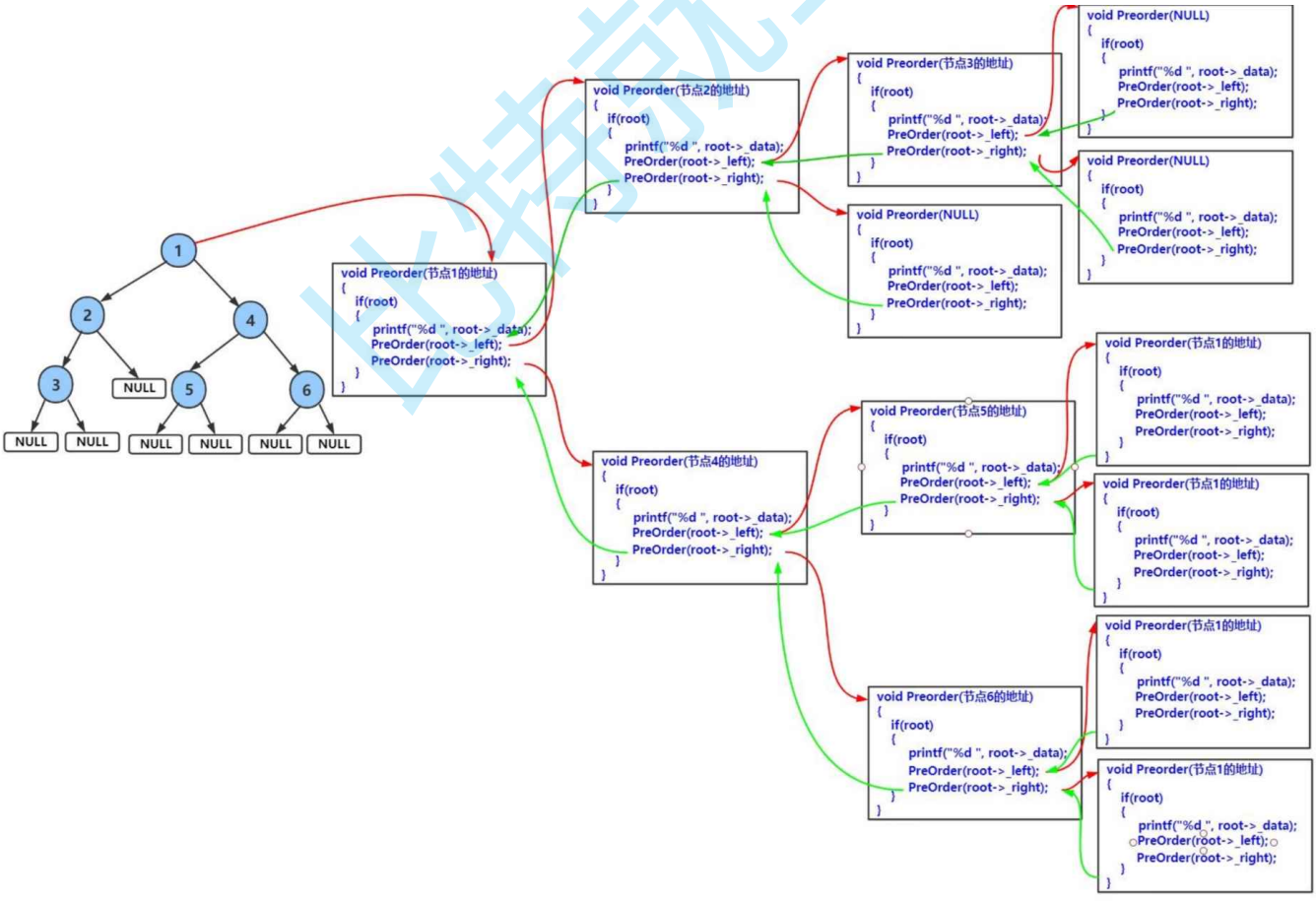
图解遍历：

> 以前序遍历为例：



二叉树

函数递归栈帧图：



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 1 5 6 4 1

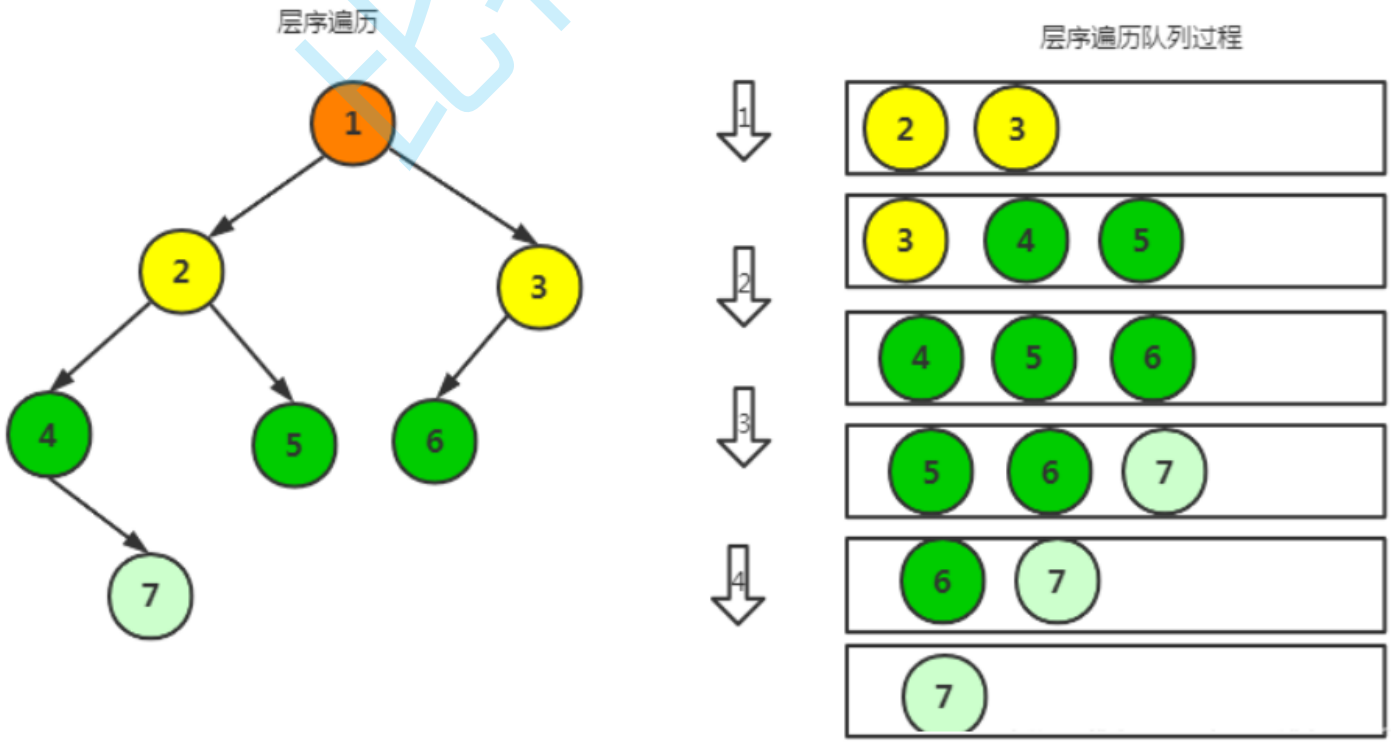
4.2 结点个数以及高度等

```
// 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);
//二叉树的深度/高度
int BinaryTreeDepth(BTNode* root);
// 二叉树查找值为x的结点
BTNode* BinaryTreeFind(BTNode* root, BTDataType x);
// 二叉树销毁
void BinaryTreeDestory(BTNode** root);
```

4.3 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

实现层序遍历需要额外借助数据结构：**队列**



```c
// 层序遍历
void LevelOrder(BTNode* root)
{
    Queue q;
    QueueInit(&q);
    QueuePush(&q, root);
    while (!QueueEmpty(&q))
    {
        BTNode* top = QueueFront(&q);
        printf("%c ", top->data);
        QueuePop(&q);
        if (top->_left)
        {
            QueuePush(&q, top->_left);
        }
        if (top->_right)
        {
            QueuePush(&q, top->_right);
        }
    }
    QueueDesTroy(&q);
}
```

4.4 判断是否为完全二叉树

```c
// 判断二叉树是否是完全二叉树
bool BinaryTreeComplete(BTNode* root)
{
    Queue q;
    QueueInit(&q);
    QueuePush(&q, root);
    while (!QueueEmpty(&q))
    {
        BTNode* top = QueueFront(&q);
        QueuePop(&q);
        if (top == NULL)
        {
            break;
        }
        QueuePush(&q, top->_left);
        QueuePush(&q, top->_right);
    }
    while (!QueueEmpty(&q))
    {
        BTNode* top = QueueFront(&q);
```

```
        QueuePop(&q);
        if (top != NULL)
        {
            QueueDesTroy(&q);
            return false;
        }
    }
    QueueDesTroy(&q);
    return true;
}
```

5. 二叉树算法题

5.1 单值二叉树

https://leetcode.cn/problems/univalued-binary-tree/description/

5.2 相同的树

https://leetcode.cn/problems/same-tree/description/

> 基于上一道OJ题，拓展学习
>
> 对称二叉树：https://leetcode.cn/problems/symmetric-tree/description/

5.3 另一棵树的子树

https://leetcode.cn/problems/subtree-of-another-tree/description/

5.4 二叉树遍历

前序遍历：https://leetcode.cn/problems/binary-tree-preorder-traversal/description/

> 课下完成剩下的遍历方式
>
> 中序遍历：
>
> https://leetcode.cn/problems/binary-tree-inorder-traversal/description/
>
> 后序遍历：
>
> https://leetcode.cn/problems/binary-tree-postorder-traversal/description/

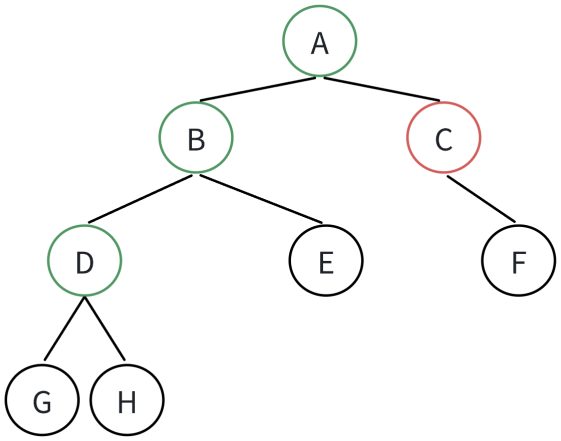
5.5 二叉树的构建及遍历

https://www.nowcoder.com/practice/4b91205483694f449f94c179883c1fef

6. 二叉树选择题

💡 二叉树性质

1）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 $n_0 = n_2 + 1$



在该二叉树中：
-度为2的节点有3个
-度为1的节点有1个
-度为0的节点有4个
因此二叉树的边数:
根据2a+b得出　：边数 = 2 * 3 + 1 = 7
根据a+b+c-1得出：边数 = 3 + 1 + 4 - 1 = 7

> 证明上述性质：
>
> 假设一个二叉树有 `a` 个度为2的节点， `b` 个度为1的节点， `c` 个叶节点，则这个二叉树的边数是 `2a+b`
>
> 另一方面，由于共有 `a+b+c` 个节点，所以边数等于 `a+b+c-1`
>
> 结合上面两个公式：
>
> `2a+b = a+b+c-1` ，即： `a = c-1`

根据二叉树的性质，完成以下选择题：

```
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.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

3.一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
```

```
4.一个具有767个结点的完全二叉树，其叶子结点个数为（）
A 383
B 384
C 385
D 386

答案：
1.B
2.A
3.B
4.B
```

链式二叉树遍历选择题

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

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