广义表（考5-6次）

广义表，又称列表，也是一种线性存储结构。

同数组类似，广义表中既可以存储不可再分的元素，也可以存储广义表，记作：

LS = (a1,a2,...,an)

其中，LS 代表广义表的名称，an 表示广义表存储的数据。广义表中每个 ai 既可以代表单个元素，也可以代表另一个广义表。

原子和子表

通常，**广义表中存储的单个元素称为 "原子"，而存储的广义表称为 "子表"。**

例如创建一个广义表 LS = {1,{1,2,3}}，我们可以这样解释此广义表的构成：广义表 LS 存储了一个原子 1 和子表 {1,2,3}。

以下是广义表存储数据的一些常用形式：

- A = ()：A 表示一个广义表，只不过表是空的。
- B = (e)：广义表 B 中只有一个原子 e。
- C = (a,(b,c,d)) ：广义表 C 中有两个元素，原子 a 和子表 (b,c,d)。
- D = (A,B,C)：广义表 D 中存有 3 个子表，分别是A、B和C。这种表示方式等同于 D = ((),(e),(b,c,d))。
- E = (a,E)：**广义表 E 中有两个元素，原子 a 和它本身。**这是一个递归广义表，等同于：E = (a,(a,(a,...)))。

注意，A = () 和 A = (()) 是不一样的。前者是空表，而后者是包含一个子表的广义表，只不过这个子表是空表。

广义表的表头和表尾

当广义表不是空表时，称第一个数据（原子或子表）为"表头"，剩下的数据构成的新广义表为"表尾"。

强调一下，**除非广义表为空表，否则广义表一定具有表头和表尾，且广义表的表尾一定是一个广义表。**

例如在广义表中 LS={1,{1,2,3},5} 中，表头为原子 1，表尾为子表 {1,2,3} 和原子 5 构成的广义表，即 {{1,2,3},5}。

再比如，在广义表 LS = {1} 中，表头为原子 1 ，但由于广义表中无表尾元素，因此该表的表尾是一个空表，用 {} 表示。

例子：LS={LS,{}}，的深度是无穷，表头元素是LS，表尾元素是{{}}

广义表的长度和深度计算

长度：最外层括号中有几个元素(原子或子表)

深度：括号的重数，递归广义表的深度可以是无穷

排序考点（年年考，理解下面所有即可）

基本思想

1. 直接插入排序

第n次插入时，已经有n-1个数据是有序存放的了，下一步要做的就是在n-1个数据中找一个合适的位置插入第n个数据使其保持有序。

2. 希尔排序

基于直接插入排序的改进。规定步长从大到小（最终步长是1），每个步长下对分组内的数据进行直接插入排序，等到最后一轮时整个序列相较于初始时已经更有序了，然后最后再进行一轮直接插入排序即可。开始时用大步长，目的是把后面较小的元素能够交换到跨度很大的前方。

3. 冒泡排序

第n轮冒泡排序可以得到n个有序的序列，每轮两两交换把最大的元素换到右边界

4. 快速排序

基于冒泡排序的改进。每次选择一个pivot分割序列为两部分，一部分小于pivot一部分大于pivot，每轮确定pivot在最终有序序列中的位置。

5. 直接选择排序

遍历一遍整个序列，找出最大值，将其与最后一个元素交换，下次遍历的有边界变为倒数第二个元素，继续进行前面的操作。每轮都有一个最值被放到一端，和冒泡很像。

6. 堆排序

基于直接选择排序改进。建堆、选择堆顶元素、调整堆结构。

归并排序

基于分治法。合并树。

7. 基数排序

对于整数的排序，先按照个位数从小到大的顺序排放元素，得到新顺序的序列，再按照十位数从小到大的顺序在上一步的新顺序中再次排放元素，再按照百位数从小到大的顺序在上一步的新序列中再次排放元素。

8. 计数排序

每个数都是数组索引，把每个数的个数记录在对于数组位置上，然后遍历数组一个个取出非零单元的索引即可实现排序。

复杂度和稳定性(考过)

名称	时间复杂度			额外空间复杂度	In-place	稳定性
	最好	最坏	平均			
冒泡排序（Bubble Sort）	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	✓	✓
选择排序（Selection Sort）	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	✓	✗
插入排序（Insertion Sort）	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	✓	✓
归并排序（Merge Sort）	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$	✗	✓
快速排序（Quick Sort）	$O(nlogn)$	$O(n^2)$	$O(nlogn)$	$O(logn)$	✓	✗
希尔排序（Shell Sort）	$O(n)$	$O(n^{4/3}) \sim O(n^2)$	取决于步长序列	$O(1)$	✓	✗
堆排序（Heap Sort）	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(1)$	✓	✗
计数排序（Counting Sort）	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	✗	✓
基数排序（Radix Sort）	$O(d*(n+k))$	$O(d*(n+k))$	$O(d*(n+k))$	$O(n+k)$	✗	✓
桶排序（Bucket Sort）	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+m)$	✗	✓

■ 以上表格是基于数组进行排序的一般性结论

■ 冒泡、选择、插入、归并、快速、希尔、堆排序，属于比较排序（Comparison Sorting）

对于nlogn的排序：

1. **快排速度最快**，但是不稳定，有最差O(n^2)的情况，且递归调用占用内存O(logn)
2. **堆排序额外空间占用最少**，但是不稳定
3. **归并排序永远稳定排序**，但是额外空间占用最大O(n)

稳定的排序：

1. 冒泡
2. 归并
3. 插入
4. 基数排序

插入排序：

直接插入排序（顺序搜索） O(n^2)

二分插入排序（二分搜索减少比较） O(n^2)

基于循环数组的插入排序（减少移动） O(n^2)

希尔插入排序（多次小直接插入+整体一个插入排序，利用直接插入排序适用于有序小序列的特性）O(n^1.3~n^2)

交换排序：

冒泡排序 O(n^2)

快速排序O(nlogn)

选择排序：

普通选择排序（顺序搜索选择最值）**最坏最好都是n^2**

```
    //每一轮选择最小的放在左边
      int temp;
      for(int i = 0;i<8;i++)
      {
          temp = i;
          for(int j = i+1;j<8;j++) // 记录最小值的位置
          {
             if(a[j]<a[temp])
                 temp = j;
          }
          swap(a[temp],a[i]); // 把最小值放在前面
      }
```

堆排序（堆顶为最值）O(nlogn)

归并排序（基于分治法）：归并排序的速度仅次于快速排序，用于总体无序，**但各子项相对有序的序列比较有优势。归并排序比较占用内存**，如果不用考虑内存问题，可以考虑使用。**归并排序在任何时候都是稳定的。** 可以使用插入排序结合归并排序的方式，对分治的小数组使用插入排序，合并时使用空间换时间的归并排序。

非比较排序：

基数排序（针对位数小数量多的整数）O(n+k)

最坏最好情况分析(考过)

1. 冒泡排序
    - 最好情况： 初始完全有序O(n)，前提是使用改进的bobble sort，即第一轮循环如果检测出全部有序就直接return

    ```
    public void bubbleSort(int arr[]) {
        boolean didSwap;
        for(int i = 0, len = arr.length; i < len - 1; i++) {
            didSwap = false;
            for(int j = 0; j < len - i - 1; j++) {
                if(arr[j + 1] < arr[j]) {
                    swap(arr, j, j + 1);
                    didSwap = true;
                }
            }
            if(didSwap == false)
                return;
        }
    }
    ```

    - 完全倒序O(n^2)
2. 选择排序

- 最坏最好都是n^2
- 因为冒泡一轮下来时相邻的两个两两比较，因此可以看出初始序列是否有序
- 一轮下来只知道最小的是什么，但是不知道整体是否有序。

3. 直接插入排序
   - 最好情况完全有序O(n)
   - 最坏情况完全倒序O(n^2)

```
for(int index = 1; index < len; index ++){
  int pre = index-1;
  int curr = target[index];

   while (pre >= 0 && target[pre] > curr){ // 如果初始有序，这个循环的条件就不会满足
    //pre之后的元素后移
    target[pre+1] = target[pre];
    pre --;
    }
    target[pre+1] = curr;
}
```

4. 快速排序
   - 最好情况：pivlot平均分割数据，左右两边同样多，递归树是均匀的O(nlogn)
   - 最坏情况：pivlot是最大值或最小值，此时递归树退化为链表，O(n^2)
5. 归并排序
   - 都是nlogn，很稳定

插入排序的算法

```
for(int index = 1; index < len; index ++){
  int pre = index-1; // 已排好序的最后一个元素的下标
  int curr = target[index]; // 待排序的第一个元素下标

   while (pre >= 0 && target[pre] > curr){ // 如果初始有序，这个循环的条件就不会满足
    //pre之后的元素后移
    target[pre+1] = target[pre];
    pre --;
    }
    target[pre+1] = curr;
}
```

冒泡排序的算法(考过)

```java
    boolean didSwap;
    for(int i = 0, len = arr.length; i < len - 1; i++) {
        didSwap = false;
        for(int j = 0; j < len - i - 1; j++) {
            if(arr[j + 1] < arr[j]) {
                swap(arr, j, j + 1);
                didSwap = true;
            }
        }
        if(didSwap == false)
            return;
    }
```

希尔排序的过程推导

p271(考过)

快速排序的过程推导

p275(考过)

堆排序的建堆过程、调整过程推导

p281(考过)

归并排序的合并过程推导

p283(考过)

基数排序的过程推导

p287

二维数组的计算（考4次）

二维数组也是存储在一维顺序空间的，有行序为主序和列序为主序两种方式

1. 写出行数和列数
2. 判断主序是行序还是列序
3. 判断数组行列的起始下标fx和fy是0还是1
4. 行序：（i-fx）* 列数 + （j-fy）= 单元数

   列序：（j-fy）* 行数 + （i-fx）=单元数
5. 地址 = 单元数*每个单元的字节+基地址

注意：fx和fy是数组起始下标，一般都是0或1；

M[10][20]

A[1...8，1...10]

设有二为数组int M[10][20](注：m为0...10,n为0...20),每个元素（整数）栈两个存储单元，数组的起始地址为2000，元素M[5][10]的存储位置为___，M[8][19]的存储值为___

行数：10； 列数： 20

（5-0） * 20 + (10-0) = 110

110*4 = 440

2000+440 = 2440

(8*20+19)*4 + 2000

数组M中每个元素的长度是3个字节，行下标i从1到8，列下标j从1到10，从首的址EA开始连续存放在存储其中。若按行方式存放，元素M[8][5]的起始地址为___；若按列优先方式存放，元素M[8][5]的地址为___。

行数：8； 列数： 10

（8-1） * 10 + (5-1) = 74

74*3 = 222

EA + 222

(5-1)*8 + (8-1) = 39

39*3=117

EA + 117

设8行10列的二维数组A[1...8，1...10]分别以行序为主序和以列序为主序顺序存储时，其首地址相同，那么以以行序为主序时元素a[3，5]的地址与以以列序为主序时( )元素的地址相同（A无第0行第0列，a[i,j]表示第i行第j列的元素）。

A、a[7,3] B、a[8,3] **C、a[1,4]** D、a[6,5]

行数： 10； 列数： 8

（3-1） * 10 + （5-1） = 24

(y-1)*8 + (x-1) = 24

x = 1, y = 4

哈夫曼树的考法（必考）

1.定义：**带权路径长度最短的树**，又叫最优树

n个值构建哈夫曼树需要2n-1个节点

2.画出哈夫曼树，写出编码

3.计算字符串编码长度：**频率*单个字符编码长度求和**

带权路径长度对应的就是平均编码长度

树的基础考法（考过4次以上）

1. 树高和节点数的关系：

   $n\text{-}max = 2^h - 1$; 最多的节点数

   $h = \log_2(n+1)$ 满二叉树

   $h = \text{floor}(\log_2 n) + 1$ 完全二叉树，注意+1

   （应用：n个元素二分查找最坏的比较次数就是树高）

2.每一层的节点数：**$2^{(i-1)}$，i是层数**

3.**$n_0 = n_2 + 1$**

4.非空指针个数：n-1

空指针个数： n+1

5.下面是对于Complete Binary Tree而言的：

1. 第一个非叶子结点序号： **size/2**
2. 叶子结点个数： size - size/2
3. 2度节点个数： size - size/2 - 1
4. 1度节点个数： 1，最多只能有一个，且只有左子树没有右子树
5. 最少节点数：**$2^{(h-1)}$**
6. 最多节点数：**$2^h - 1$**
7. 父亲i和左右孩子的对应关系： **2i，2i+1**
8. 孩子i和父亲的对应关系：**floor(i/2)**
9. 没有左孩子：$n < 2i$
10. 没有右孩子： $n < 2i+1$

6.对于M叉树来说：**(考过两次)**

1. **$N_0 = 1 + N_2 + 2N_3 + ... + (m-1)N_m$**
2. **第n个节点的第一个孩子节点：(n-1)*M + 2; 第i个孩子：(n-1)*M + 1 + i**
3. **第i个孩子节点的父节点编号：floor((i-2)/M)+1**
4. 完全m叉树的树高：$\log_m(n(m-1)+1)$

树的四个遍历算法（押题）

```
// 递归-前序，中序，后序
void solv1e(Node* root)
{
    if(!root) return;
    OPERATE(root); // 遍历树根
    solve(root->left);
```

```
    solve(root->right);
}
void solve(Node* root)
{
    if(!root) return;

    solve(root->left);
    OPERATE(root); // 遍历树根
    solve(root->right);
}
void solve(Node* root)
{
    if(!root) return;

    solve(root->left);
    solve(root->right);
    OPERATE(root); // 遍历树根
}
// 非递归-层序，前序，中序
void solve(Node* root)
{
    if(!root) return;
    queue<Node*> q;
    q.push(root);
    while(q.empty() == false)
    {
        Node* node = q.front();
        q.pop();
        OPERATE(node); // 对node操作

        if(node->left) q.push(node->left); // 加入下一层
        if(node->right) q.push(node->right);
    }
}

    void preorder(Node<T>* node)
    {
        stack<Node<T>*> nodeStack;
        nodeStack.push(node);
        while (!nodeStack.empty())
        {
            Node<T>* tnode = nodeStack.top();
            nodeStack.pop();
            if(this->m_op(tnode->Element))  return;

            if (tnode->right != nullptr) {
                nodeStack.push(tnode->right);
            }

            if (tnode->left != nullptr) {
                nodeStack.push(tnode->left);
            }
        }
    }
```

翻转树算法（押题）

```
//递归
void solve(Node* root)
{
    if(root == nullptr)
    {
        return;
    }
    Node* temp = root->right;
    root->right = root->left;
    root->left = root->right;

    solve(root->left);
    solve(root->right);
}
// 层序遍历
void solve(Node* root)
{
    if(node == NULL) return;

    queue<Node*> q;
    q.push(root); // 初始根节点入队
    while(!q.empty())
    {
        Node* node = q.front();
        q.pop(); // 出队

        Node* temp = q->right; // 交换
        q->right = q->left;
        q->left = q->right;

        if(node->left) q.push(node->left); // 加入下一层
        if(node->right) q.push(node->right);
    }
}
```

求树高（押题）

```
// 递归
int Hight(Node* root)
{
    if(root == nullptr) return 0;
    return max(Hight(root->left),Hight(root->right))+1;
}

// 层序遍历
int Hight(Node* root)
{
    if(root == nullptr) return 0;
```

```
    queue<Node*> q;
    q.push(root); // 初始根节点入队
    int hight = 0;
    int layerNums = 1; // 记录每一层的节点个数
    while(!q.empty())
    {
        Node* node = q.front();
        q.pop(); // 出队
        layerNums--;

        if(node->left) q.push(node->left); // 加入下一层
        if(node->right) q.push(node->right);

        if(layerNums == 0)
        {
            hight++;
            layerNums = q.size(); // 更新为新一层的节点数量
        }
    }
}
```

遍历的应用(押题)

前序遍历

- 显示文件层次结构，因为文件一般存在于磁盘里，是用树存储的，所以使用先序遍历可以打印出文件的层级结构，即**根目录在上面，子目录在后面。**

中序遍历

- BST+中序遍历可以实现**升序或降序**排列

后序遍历

- **删除节点时可以使用**，因为遍历到某个节点时此节点的子节点一定一定遍历(删除)过了。

层序遍历

- **求树高**
- **判断是否为完全二叉树**
- **DFS**

普通链表条件

遍历

while(node != nullptr)

删除

while(node->next != nullptr)

插删链表节点

转置链表（考两次）

```
class Solution {
    public ListNode reverseList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }
        ListNode newHead = reverseList(head.next);
        head.next.next = head;
        head.next = null;
        return newHead;
    }
}

class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* temp; // 保存cur的下一个节点
        ListNode* cur = head;
        ListNode* pre = NULL;
        while(cur) {
            temp = cur->next;  // 保存一下 cur的下一个节点，因为接下来要改变cur->next
            cur->next = pre; // 翻转操作
            // 更新pre 和 cur指针
            pre = cur;
            cur = temp;
        }
        return pre;
    }
};

```

判断环

双指针，快慢指针

循环链表条件（考3-4次）

单向循环链表为空：

L->next = L

双向循环链表为空

L->next = L->prev = L

遍历链表(有header)

while(node != header) do:

```
node=node->next
```

插入、删除的过程

循环队列计算

方式1：多出一个额外的空间，避免歧义（考四次）

- front指向队头的前一个
- rear指向队尾元素

入队：

rear = (rear+1)%M

q[rear] = val

出队：

front = (front+1)%M

x = q[front];

判断空：

front == rear

判断满（**空位法**）：

(rear + 1) % M == front

- front指向队头
- rear指向队尾元素的下一个

入队：

q[rear] = val

rear = (rear+1)%M

出队：

x = q[front];

front = (front+1)%M

判断空：

front == rear

判断满：

(rear + 1) % M == front

求size大小：

(rear - front + M)%M

方式2：标记tag

tag = 0 出队操作，导致队列空

tag = 1 入队操作，导致队列满

方式3: 增加size变量（考两次）

size == 0 空

size == M 满

有front获取rear： rear = (front + **size-1**) % M

有rear获取front： front = （rear-size+M+1）% M（**rear - (size-1) + M**）

二分查找的考点

计算查找长度：

1.查找成功时最坏：**比较次数=floor(log2n)+1**，即n个节点的完全二叉树树高

2.查找成功时平均：ASL=(n+1/n)log2(n+1) - 1，n>50有log2(n+1) - 1

利用二叉判定树计算：

利用二叉判定树计算

$$ASL_{成功} = \frac{每个结点的比较次数之和}{结点数}$$

$$ASL_{失败} = \frac{空指针处比较次数之和}{空指针数}$$

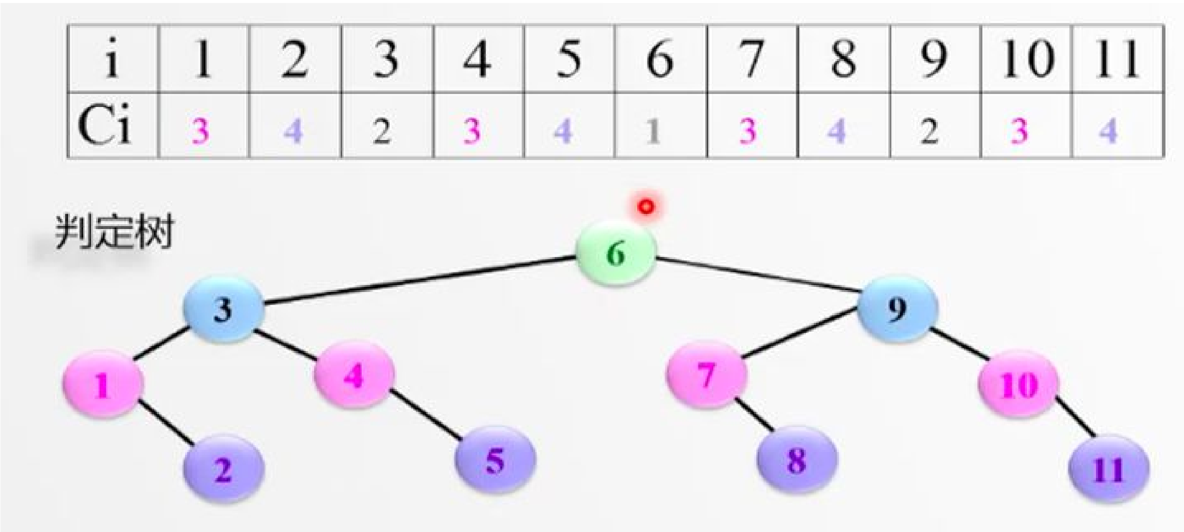
二分查找的算法代码（10年考5-6次）

```
int Search(A,k)
{
```

```
    int i=0,j=len(A)-1,mid;
    while(i <= j)
    {
        mid = (i+j)/2;
        if(k > A[mid])
        {
            i = mid + 1;
        }else if(k< A[mid])
        {
            j = mid - 1;
        }else
        {
            return mid;
        }
    }
    return -1;
}
```

二分查找的判定树画法（1次）

根据判定树得到ASL



i	1	2	3	4	5	6	7	8	9	10	11
Ci	3	4	2	3	4	1	3	4	2	3	4

哈希表考点

哈希冲突的解决方法：

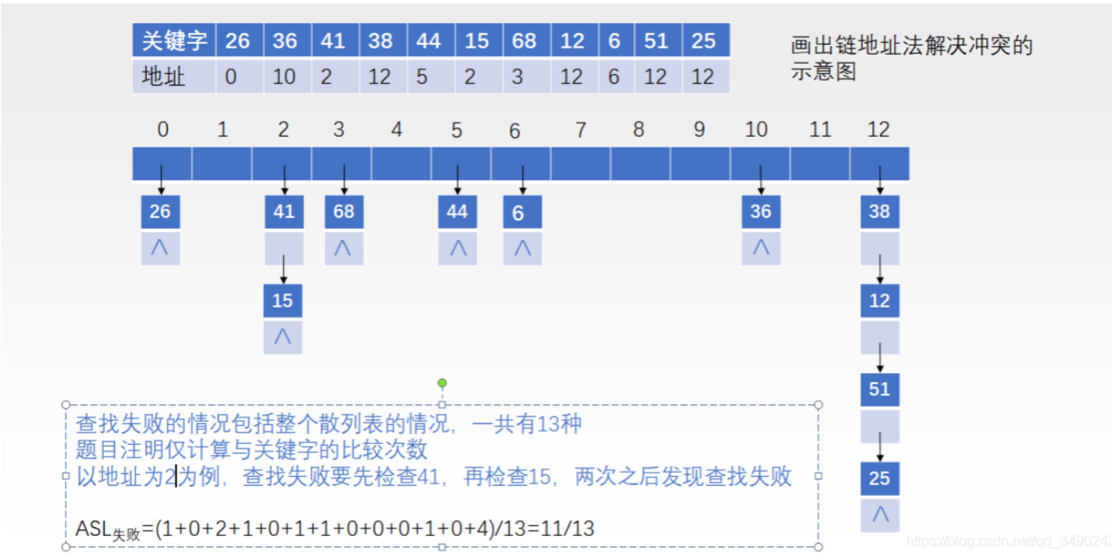
1. **开放定址法**

   H（key）=（H（key）**+ d**）MOD m（其中 m 为哈希表的表长，d 为一个增量）

   - 线性探测法：d=1，2，3，…，m-1（**考过无数次**）
   - 二次探测法：d=1^2，-1^2，2^2，-2^2，3^2，…（考过几次）
   - 伪随机数探测法：d=伪随机数
   - **计算平均查找长度**

- 在线性探测法中，当遇到冲突时，从发生冲突位置起，每次 +1，向右探测，直到有空闲的位置为止；二次探测法中，从发生冲突的位置起，按照 +12，-12，+22，...如此探测，直到有空闲的位置；伪随机探测，每次加上一个随机数，直到探测到空闲位置结束。

2. 再哈希法(好像就考过一次)
3. **链地址法**
    - **计算平均查找长度**
    - 计算查找成功时的ASL：查找一次就找到的数量*1+查找两次就找到的数量*2+....../元素的数量
    - 计算查找失败时的ASL：每个链表上查询到最后一个元素的比较次数累加/bucket长度



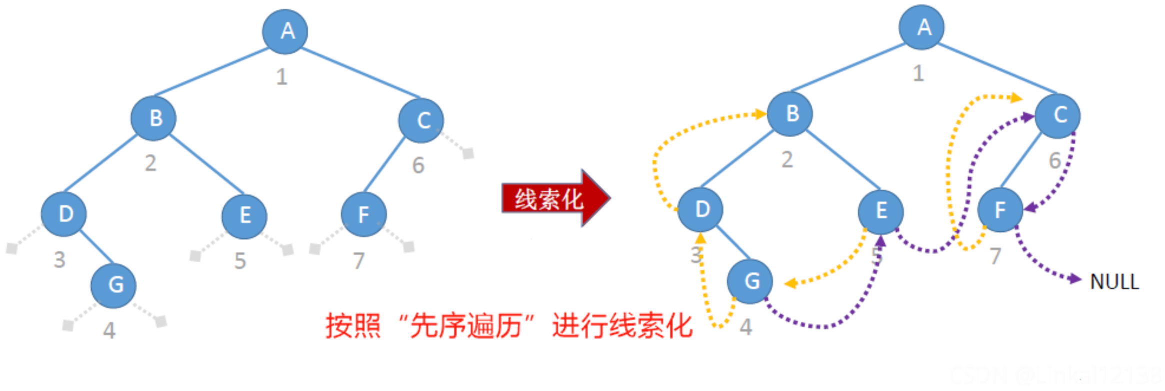
线索二叉树的遍历（押题，考过一次）

先序遍历：根-左-右，最后一个节点是右，是可以有空指针留出来做后继线索的。

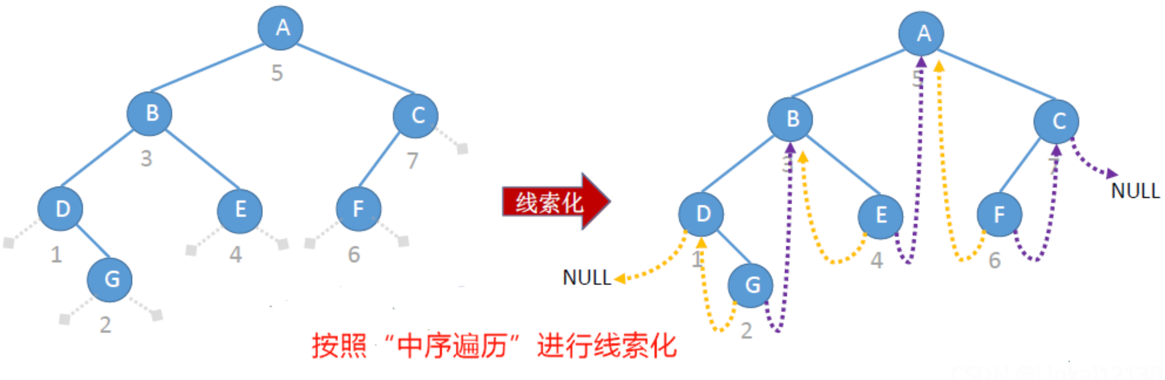
中序遍历：左-根-右，最后一个节点是右，是可以有空指针留出来做后继线索的。

后序遍历：左右-根，最后是根节点，不能有空指针做后继线索

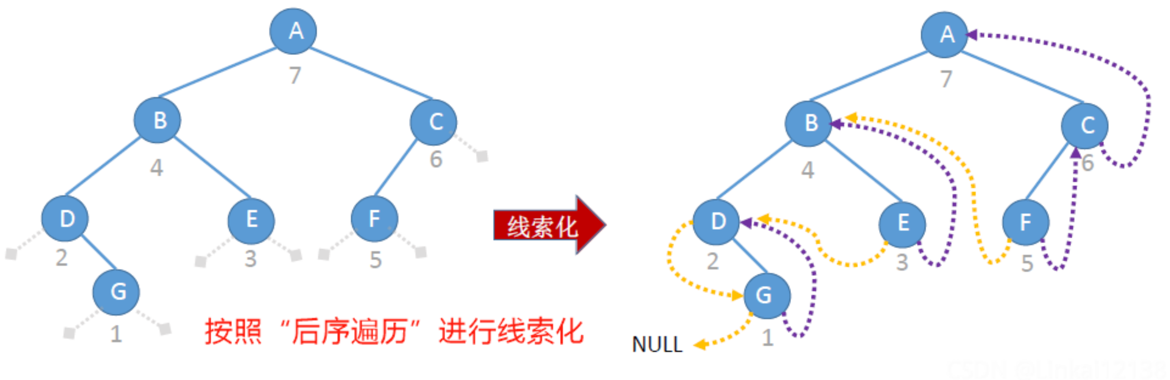
A B D G E C F



D G B E A F C



G D E B F C A



矩阵压缩、稀疏矩阵

对称矩阵、上下三角矩阵

本质：矩阵中的一个二元点通过函数映射为一维点

矩阵中的一个元素：(i,j) 第i行第j列，从(1,1)开始

顺序表中的元素：从0号单元开始存储

映射公式(对于下三角的元素)：index = (i-1)*i/2 + j-1，对于上三角的元素压缩只需要i、j互换即可

稀疏矩阵

按照（行标，列标，元素值）的顺序存放非零元素的三元组。

图的

概念：

1. 连通分量：极大连通子图
2. 生成树：极小连通子图，包含所有顶点和n-1条边
3. 强连通图：任何两个顶点之间都有路径
4. 强**连通分量**：极大强**连通子图**

计算：

1. 入度=出度=边数=总度数/2
2. 完全图的边数：Cn2(无向)，2*Cn2(有向)

邻接表和遍历（押题）

- 实现连通性判断
- 实现连通分支判断

```
class Vertex
{
    public:
    int val;
    int color;
    AdjVertex* first;
};
class Adjvertex
{
    public:
    int index;
    Adjvertex* next;
};
Vertex* vertices = new Vertex[100];
// 注： 下面的代码都是类成员函数，类内已声明成员变量size、vertices等，可以直接使用，且无需函数传参。
bool Func()
{
    for(int i = 0;i <= size;i++)
    {
        vertices[i].color = 0;
    }
    DFS(&vertices[0]);
    for(int i = 0;i <= size;i++)
    {
        if(vertices[i].color == 0)
```

```
        {
            return false;
        }
    }
    return true;
}

void DFS(Vertex* v)
{
    v->color = 1;
    AdjVertex* adv = v->first;
    while(adv != nullptr)
    {
        if(vertices[adv->index].color == 0)
        {
            DFS(&vertices[adv->index]);
        }
        adv = adv->next;

    }
    v->color = -1;
}
```

小东西

1. 邻接矩阵和出入度的关系
   - 行加1：出度
   - 列加1：入度
2. 特殊的字符串：**空串和空格串**
3. 存储方式

> 1、**顺序**存储方式：顺序存储方式就是在一块连续的存储区域一个接着一个的存放数据，把逻辑上相连的结点存储在物理位置上相邻的存储单元里，结点间的逻辑关系由存储单元的邻接挂安息来体现。顺序存储方式也称为顺序存储结构，一般采用数组或者结构数组来描述。
>
> 2、**链接**存储方法：它比较灵活，其不要求逻辑上相邻的结点在物理位置上相邻，结点间的逻辑关系由附加的引用字段表示。一个结点的引用字段往往指导下一个结点的存放位置。链接存储方式也称为链接式存储结构，一般在原数据项中增加应用类型来表示结点之间的位置关系。
>
> 3、**索引**存储方法：除建立存储结点信息外，还建立附加的索引表来标识结点的地址。它细分为两类：稠密索引：每个结点在索引表中都有一个索引项，索引项的地址指示结点所在的的存储位置；稀疏索引：一组结点在索引表中只对应一个索引项，索引项的地址指示一组结点的起始存储位置。
>
> 4、**散列**存储方法：就是根据结点的关键字直接计算出该结点的存储地址。

4. 树的节点数量关系：m叉树满足 N0 = 1 + N2 + 2N3 + 3N4 + ... + (m-1)Nm
5. 二分查找的最大查找长度=完全二叉树的树高=floor(logn）+1