# 第一章 概论

## 1.1 引言

程序=算法+数据结构。这里的数据结构指的是数据的逻辑结构和存储结构。算法是对数据运算的描述。好的算法建立在研究数据的特性及数据之间存在的关系的基础上，这正是数据结构这要研究的内容。，

数据结构

Data structure是带有结构的数据元素的集合。结构指数据之间的相互关系，包含内容：

* 数据的逻辑关系：如线性结构的顺序表，非线性结构的树，图，网状
* 数据元素及关系的存储方式：如顺序存储结构，链式存储结构。索引存储等
* 数据的运算：即对数据元素施加的操作。如CRUD，排序等。

### 算法分析

算法的正确性首先要考虑。此外要考虑如下几点：

* 执行算法所耗费的时间，即时间复杂度
* 执行算法所耗费的存储空间，主要是辅助空间，即空间复杂度。
* 算法应易于理解，易于调试，即可读性和可操作性。

时间复杂度

Time complexity。描述该算法的运行时间。算法中基本操作重复执行的次数是问题规模n的某个函数f（n），算法的时间量度记为T（n）= O（f（n））

渐进时间复杂度

若存在函数 f（n），使得当n趋近于无穷大时，T(n）/f（n）的极限值为不等于零的常数，则称 f（n）是T（n）的同数量级函数。记作 T（n）= O（f（n）），称O（f（n）） 为算法的渐进时间复杂度，简称时间复杂度。

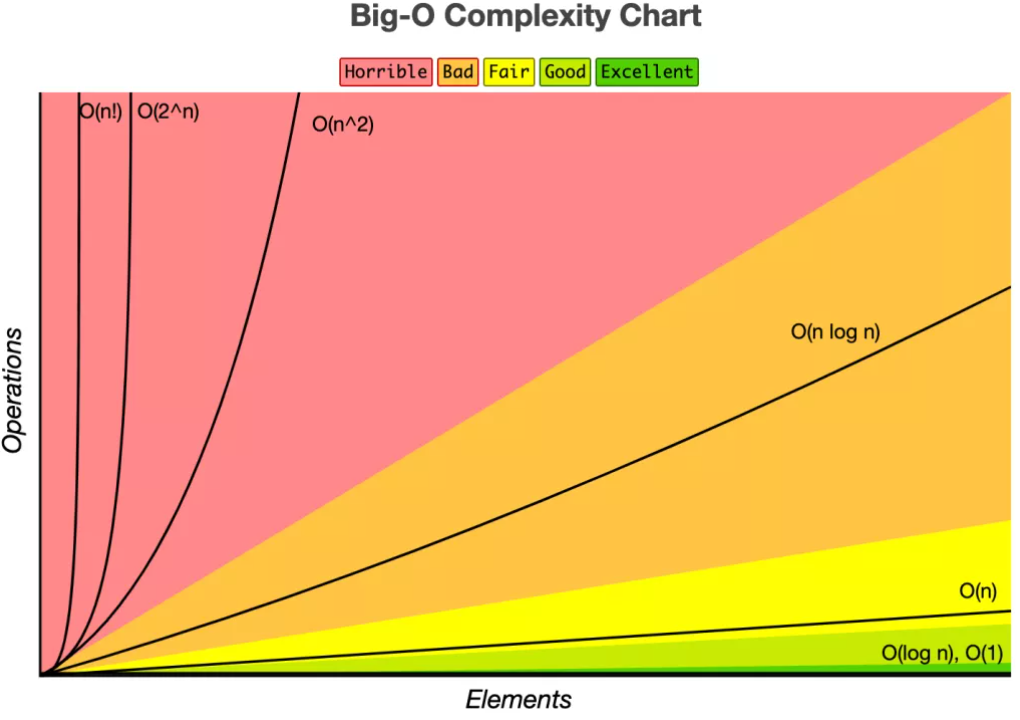
空间复杂度

数组排序



大-O 复杂度曲线

复杂度 O（1）< O（logn）< O（n）< O（n^2）



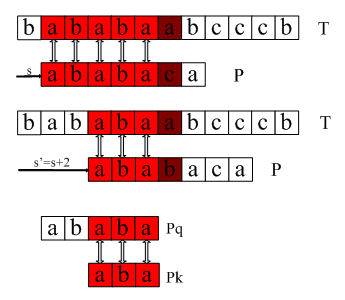
[算法复杂度速查表](https://mp.weixin.qq.com/s?__biz=MzU0MzQ5MDA0Mw==&mid=2247490260&idx=2&sn=b7d07f8153e2998fa37122325a4bf92a&chksm=fb0bf040cc7c7956b270945171100848173c670da68e9257396488f0e6445f3793e54f7b31c2&mpshare=1&scene=1&srcid=&sharer_sharetime=1590139195683&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=fa39e04f3e1bcf44b631646d69ae5652473a589393119bed6d988f051dbfc2f96133b99178872ec0505f8d9b04ee713c631ac1adee5ddd1547a197355d811779ce94649ba1182f637e6602654c98d95c&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10+x64&version=62090070&lang=zh_CN&exportkey=AYvkq%2BojmbRNAeQst2FRuHE%3D&pass_ticket=xoZWNA6nlLOr%2BtBYnLlVyAx58odUqNERXEtfBc1u0egP9IOs%2BzxcGBxteF5NX6bC)

## 1.2 数据结构

### 串

子串定位又模式匹配Pattern Matching。子串称模式串，主串称目标串；

朴素的串匹配：



### 顺序表

Sequential list

1. #define ListSize 100
2. Typedef **int** DataType;
3. Typedef **struct**{
4. DataType data[ListSize];
5. Int length;
6. }Seqlist

插入：

指在表第i（1<=i<=n+1）位置，插入x，使n长度变成n+1

void insertList(Seqlist \* L,DataType x,int i)

{

if(i<1||i>L->length+1) //可以在length+1上插

Error(“position error”);

If(L->length>=ListSize)

Error(“overflow”);

For(j=L->length-1;j>=i-1;j--)

{

L->data[j+1]=L->data[j];

}

L->data[i-1]=x;

L->length++;

表中位置i比向量下标大1，

### Heap

是计算机科学中的一种特别的树状数据结构。满足下面特性

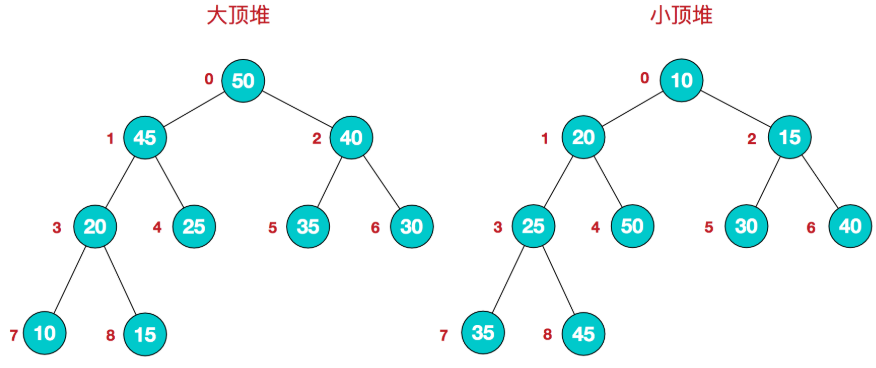
大顶堆：每个结点的值都大于或等于其左右孩子结点的值。

小顶堆：每个结点的值都小于或等于其左右孩子结点的值。

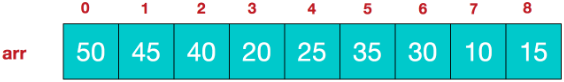
二叉堆

堆并不一定是完全二叉树，平时使用完全二叉树的原因是易于存储，并且便于索引。

* 二叉堆是一颗完全二叉树。
* 每一次堆调整的复杂度为O(logn)，数的高度?
* 平均时间复杂度均为O(nlogn)，它也是不稳定排序。



同时，我们对堆中的结点按层进行编号，将这种逻辑结构映射到数组中就是下面这个样子



该数组从逻辑上讲就是一个堆结构，我们用简单的公式来描述一下堆的定义就是：

大顶堆：arr[i] >= arr[2i+1] && arr[i] >= arr[2i+2]

小顶堆：arr[i] <= arr[2i+1] && arr[i] <= arr[2i+2]

### Stack

**顺序栈：**

#define StackSize 100

typedef struct{

DataType data[StackSize];

int top;

} SeqStack;

Int StackEmpty(SeqStack \* s)

{

Return s->top==-1;

}

void push(SeqStack \* s,DataType x)

{

s->data[++s->top]=x;

}

**链栈**

struct StackNode{

DataType data;

struct StackNode \* next;

}

Typedef struct{

StackNode \* top;

}LinkStack;

指向指针的指针。LinkStack \*s;

initStack :s->top=NULL;

int StackEmpty

{

return s->top==NULL;

};

入栈

void push(LinkStack \* s,x)

{

StackNode \* p=(StackNode \*)malloc(sizeof(StackNode));

p->data=x;

p->next=s->top;

s->top=p;

}

### Queue

区别于线性表list

双链表

Typedef struct dlistnode{

DataType data;

Struct dlistnode \* prior,\* next;

}DlistNode;

#### 队列

struct Queue

{

char data[QueueSize];

int front,rear;

};

#### 循环队列

顺时针旋转，rear逻辑上在front之前,m[0]上有元素，rear提前有个空位置时front=(rear-len+m)%m。

rear没有时（题目中会说**rear指向尾元素的位置**），（rear-len+m+1）%m

Rear，front的长度都<QuenueSize，当转一圈后，从num为1；

typedef struct

{

char data[QueueSize];

int count;

int rear,front;

}CirQueue;

void initCirQueue(CirQueue \*cirQueue)

{

cirQueue->count=0;

cirQueue->front=0;

cirQueue->rear=0;

}

int isCirQueueEmputy(CirQueue \*cirQueue)

{

return cirQueue->count==0;

}

int isCirQueueFull(CirQueue \*cirQueue)

{

return cirQueue->count==QueueSize;

}

void EnQueue(CirQueue \*cirQueue,char x)

{

if(isCirQueueFull(cirQueue))

{

printf("cirQueue is Full");

exit(1);

}

cirQueue->data[cirQueue->rear]=x; //队尾进入

int num= (++cirQueue->rear)%QueueSize;//如果队列长10，

cirQueue->rear=num;

cirQueue->count++;

}

char DeQueue(CirQueue \*cirQueue)

{

if(isCirQueueEmputy(cirQueue))

{

printf("cirQueue is Emputy");

exit(1);

}

char str=cirQueue->data[cirQueue->front];

cirQueue->front=(++cirQueue->front)%QueueSize;都是++，顺序循环

cirQueue->count--;

return str;

}

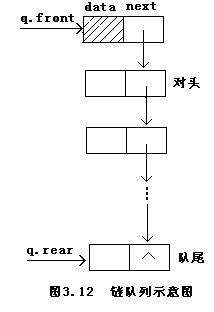
**循环队列**

Typedef struct quenuenode{

DataType data;

Struct quenuenode \* next;

}QuenuNode;



Typedef struct{

QueueNode \* front;

QueueNode \* rear;

}LinkQueue;

循环单链表：

rear->next->next来表示front

双链表：

Typedef struct dlistnode{

DataType data;

Struct dlistnode \* prior,\*next;

}DlistNode;

#### 优先队列

优先队列是计算机科学中的一类抽象数据类型。优先队列中的每个元素都有各自的优先级，优先级最高的元素最先得到服务；优先级相同的元素按照其在优先队列中的顺序得到服务。优先队列往往用堆来实现。

初级实现

有许多简单低效的实现。如用一个有序的数组；或使用无序数组，在每次取出时搜索全集合，这种方法插入的效率为O(1)，但取出时效率为​O(n)。

典型实现

* 二叉堆：插入与提取操作的时间复杂度为O(log n)
* 自平衡二叉查找树

# 第二章 常见算法

## 动态规划

动态规划算法的核心是记住已经求过的解，记住求解的方式有两种：

* 自顶向下的备忘录法
* 自底向上

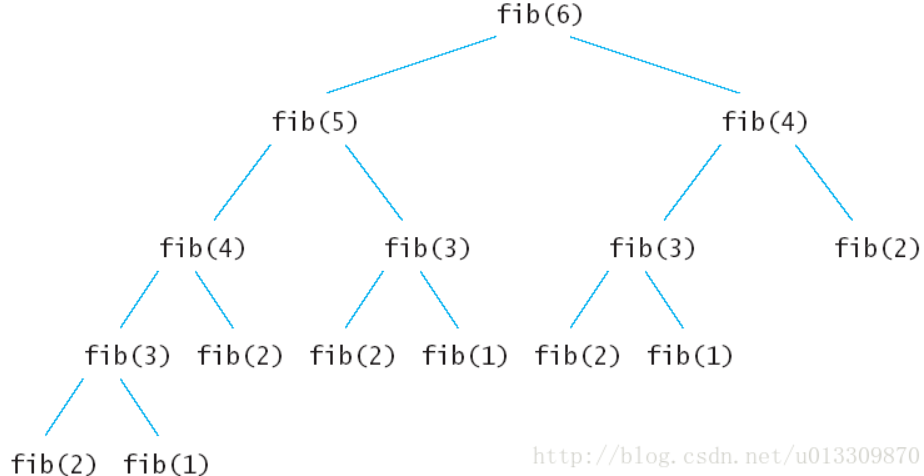
举一个最简单的例子：求斐波拉契数列Fibonacci

1. F(0) = 0，F(1) = 1
2. F(n) = F(n - 1) + F(n - 2)，其中 n > 1

递归算法

1. **public** **int** fib(**int** n){
2. if(n==0||n==1) return n;
3. return fib( n-1)+fib(n-2);
4. }

假如输入6，那么执行的递归树如下：



递归很多重复的节点被执行，若把执行过的子节点存起来，后面要用到的时候直接查表调用的话可以节约大量的时间。下面看看动态规划的两种方法。

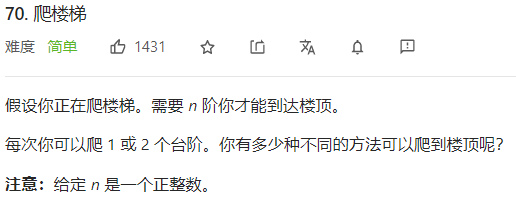
1.自顶向下的备忘录法

1. **public** **int** fibonacci(**int** n) {
2. **if** (n <= 0) **return** n;
3. **int**[] memo = **new** **int**[n + 1];
4. **for** (**int** i = 0; i <= n; i++) memo[i] = -1;
5. **return** upDown(n, memo);
6. }
8. **public** **static** **int** upDown(**int** n, **int**[] memo) {
9. **if** (memo[n] != -1) **return** memo[n];
10. **if** (n <= 2) {
11. memo[n] = 1;
12. } **else** {
13. memo[n] = upDown(n - 1, memo) + upDown(n - 2, memo);
14. }
15. **return** memo[n];
16. }

2. 自底向上

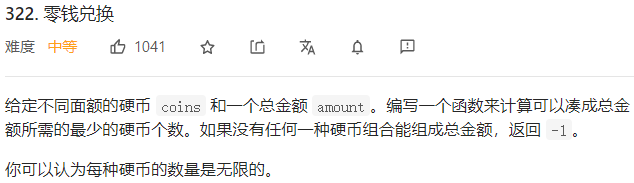
1. **public** **int** fib(**int** n) {
2. **if**(n==0||n==1) **return** n;
3. **int** [] nums=**new** **int**[n+1];
4. nums[1]=1;
5. **for**(**int** i=2;i<=N;i++){
6. nums[i]=nums[i-1]+nums[i-2];
7. }
8. **return** nums[N];
9. }

### [70. 爬楼梯](https://leetcode-cn.com/problems/climbing-stairs/)



1. **public** **int** climbStairs(**int** n) {
2. **if**(n<3) **return** n;
3. **int**[] dp=**new** **int**[n];
4. dp[0]=1;
5. dp[1]=2;
6. **for**(**int** i=2;i<n;i++){
7. dp[i]=dp[i-1]+dp[i-2];
8. }
9. **return** dp[n-1];
10. }

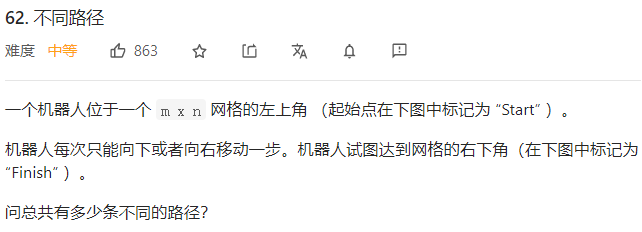
### [322. 零钱兑换](https://leetcode-cn.com/problems/coin-change/)



动态规划：自下而上

1. **public** **int** coinChange(**int**[] coins, **int** amount) {
2. **int** max = amount + 1;
3. **int**[] dp = **new** **int**[amount + 1];
4. Arrays.fill(dp, max);
5. dp[0] = 0;
6. **for** (**int** i = 1; i <= amount; i++) {
7. **for** (**int** j = 0; j < coins.length; j++) {
8. **if** (coins[j] <= i) {
9. dp[i] = Math.min(dp[i], dp[i - coins[j]] + 1);
10. }
11. }
12. }
13. **return** dp[amount] > amount ? -1 : dp[amount];
14. }

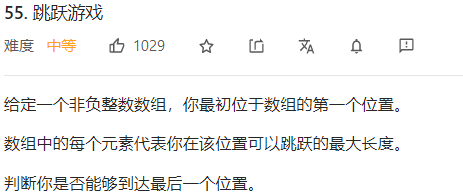
### [62. 不同路径](https://leetcode-cn.com/problems/unique-paths/)



1. **public** **int** uniquePaths(**int** n,**int** m) {
2. **int**[][] dp=**new** **int**[n][m];
3. **for**(**int** i=0;i<m;i++) dp[0][i]=1;
4. **for**(**int** i=0;i<n;i++) dp[i][0]=1;
5. **for**(**int** i=1;i<n;i++){
6. **for**(**int** j=1;j<m;j++){
7. dp[i][j]=dp[i-1][j]+dp[i][j-1];
8. }
9. }
10. **return** dp[n-1][m-1];
11. }

## 贪心

### [55. 跳跃游戏](https://leetcode-cn.com/problems/jump-game/)



1. **public** **boolean** canJump(**int**[] nums) {
2. **int** n = nums.length;
3. **int** rightmost = 0;
4. for (**int** i = 0; i < n; ++i) {
5. **if** (i <= rightmost) {
6. rightmost = Math.max(rightmost, i + nums[i]);
7. if (rightmost >= n - 1) {
8. **return** **true**;
9. }
10. }
11. }
12. **return** **false**;
13. }

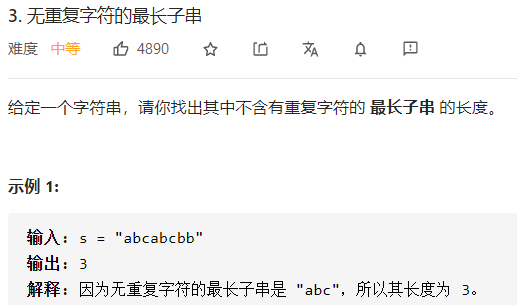
复杂度分析

时间复杂度：O(n)

空间复杂度：O(1)，不需要额外的空间开销。

## 滑动窗口

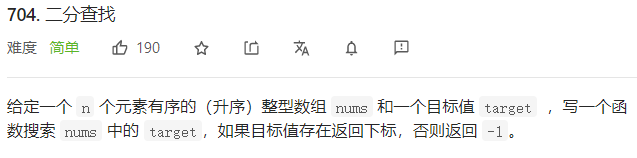
### [3. 无重复字符的最长子串](https://leetcode-cn.com/problems/longest-substring-without-repeating-characters/)



1. **public** **int** lengthOfLongestSubstring(String s) {
2. Set<Character> occ = **new** HashSet<>();  //记录每个字符是否出现过
3. **int** n = s.length();
4. // 右指针，初始值为 -1，相当于我们在字符串的左边界的左侧，还没有开始移动
5. **int** rk = -1, ans = 0;
6. **for** (**int** i = 0; i < n; ++i) {
7. **if** (i != 0) {
8. // 左指针向右移动一格，移除一个字符
9. occ.remove(s.charAt(i - 1));
10. }
11. **while** (rk + 1 < n && !occ.contains(s.charAt(rk + 1))) {
12. // 不断地移动右指针
13. occ.add(s.charAt(rk + 1));
14. ++rk;
15. }
16. // 第 i 到 rk 个字符是一个极长的无重复字符子串
17. ans = Math.max(ans, rk - i + 1);
18. }
19. **return** ans;
20. }

## 查找

### [704. 二分查找](https://leetcode-cn.com/problems/binary-search/)



1. **public** **int** search(**int**[] nums, **int** target) {
2. **int** left=0,right=nums.length-1;
3. **while**(left<=right){
4. **int** mid =(left +right)/2;
5. **if**(target<nums[mid]){
6. right= mid-1;
7. }**else** **if**(target>nums[mid]){
8. left=mid+1;
9. }**else**{
10. **return** mid;
11. }
12. }
13. **return** -1;
14. }

## 排序

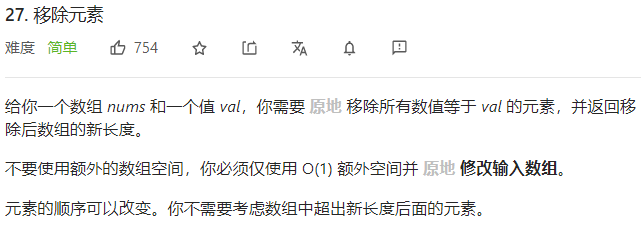
### [912. 排序数组](https://leetcode-cn.com/problems/sort-an-array/)



1. **public** **int**[] sortArray(**int**[] nums) {
2. Arrays.sort(nums);
3. //sort(nums, 0, nums.length - 1);
4. **return** nums;
5. }
7. **private** **void** sort(**int**[] arr, **int** left, **int** right) {
8. **if** (left >= right) {
9. **return**;
10. }
11. **int** mid = (left + right) / 2;
12. sort(arr, left, mid);//左边归并排序，使得左子序列有序
13. sort(arr, mid + 1, right);//右边归并排序，使得右子序列有序
14. merge(arr, left, mid, right);//将两个有序子数组合并操作
15. }
17. **private** **void** merge(**int**[] arr, **int** left, **int** mid, **int** right) {
18. **int**[] temp = **new** **int**[right - left + 1];
19. **int** i = left;//左序列指针
20. **int** j = mid + 1;//右序列指针
21. **int** t = 0;//临时数组指针
22. **while** (i <= mid && j <= right) {
23. temp[t++] = arr[i] <= arr[j] ? arr[i++] : arr[j++];
24. }
25. **while** (i <= mid) {
26. temp[t++] = arr[i++];
27. }
28. **while** (j <= right) {
29. temp[t++] = arr[j++];
30. }
31. **while** (left <= right) {
32. arr[right--] = temp[--t];
33. }
34. }

## 双指针

### [27. 移除元素](https://leetcode-cn.com/problems/remove-element/)

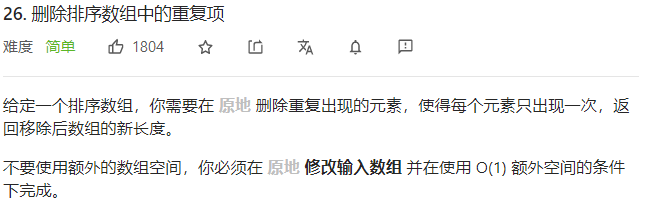


双指针

保留两个指针 i 和 j，其中 j 是慢指针，i 是快指针。

1. **public** **int** removeElement(**int**[] nums, **int** val) {
2. **int** j = 0;
3. **for** (**int** i = 0; i < nums.length; i++) {
4. **if** (nums[i] != val) {
5. nums[j] = nums[i];
6. j++;
7. }
8. }
9. **return** j;
10. }

该解法与 [26. 删除排序数组中的重复项](https://leetcode-cn.com/problems/remove-duplicates-from-sorted-array/) 的解法十分相似。

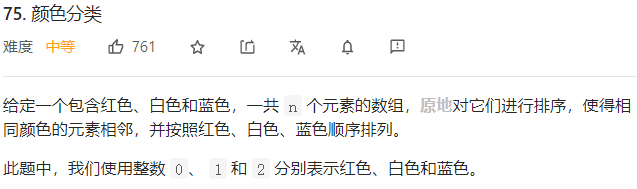


1. **public** **int** removeDuplicates(**int**[] nums) {
2. **if** (nums.length == 0) **return** 0;
3. **int** i = 0;
4. **for** (**int** j = 1; j < nums.length; j++) {
5. **if** (nums[j] != nums[i]) {
6. i++;
7. nums[i] = nums[j];
8. }
9. }
10. **return** i + 1;
11. }

My

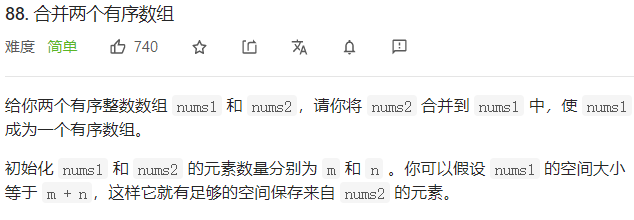
1. **public** **int** removeDuplicates(**int**[] nums) {
2. **if**(nums.length==0) **return** 0;
3. **int** j=0;
4. **for**(**int** i=1;i<nums.length;i++){
5. **if**(nums[i]==nums[j]){
6. **continue**;
7. }
8. nums[++j]=nums[i];
9. }
10. **return** j+1;
11. }

### 75. 颜色分类



1. **public** **void** sortColors(**int**[] nums) {
2. **int** n = nums.length;
3. **int** p0 = 0, p1 = 0;
4. **for** (**int** i = 0; i < n; ++i) {
5. **if** (nums[i] == 1) {
6. **int** temp = nums[i];
7. nums[i] = nums[p1];
8. nums[p1] = temp;
9. ++p1;
10. } **else** **if** (nums[i] == 0) {
11. int temp = nums[i];
12. nums[i] = nums[p0];
13. nums[p0] = temp;
14. **if** (p0 < p1) {
15. temp = nums[i];
16. nums[i] = nums[p1];
17. nums[p1] = temp;
18. }
19. ++p0;
20. ++p1;
21. }
22. }
23. }

### [88. 合并两个有序数组](https://leetcode-cn.com/problems/merge-sorted-array/)



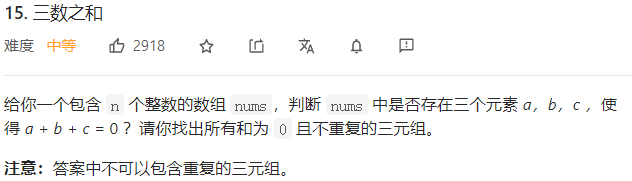
双指针 / 从前往后

1. **public** **void** merge(**int**[] nums1, **int** m, **int**[] nums2, **int** n) {
2. **int** [] nums1\_copy = **new** **int**[m];
3. System.arraycopy(nums1, 0, nums1\_copy, 0, m);
4. **int** p1 = 0,p2 = 0,**int** p = 0;
5. **while** ((p1 < m) && (p2 < n))
6. nums1[p++] =(nums1\_copy[p1]<nums2[p2]) ?nums1\_copy[p1++] :nums2[p2++];     // if there are still elements to add
7. **if** (p1 < m)
8. System.arraycopy(nums1\_copy, p1, nums1, p1 + p2, m + n - p1 - p2);
9. **if** (p2 < n)
10. System.arraycopy(nums2, p2, nums1, p1 + p2, m + n - p1 - p2);
11. }

合并后排序

1. **public** **void** merge(**int**[] nums1, **int** m, **int**[] nums2, **int** n) {
2. System.arraycopy(nums2, 0, nums1, m, n);
3. Arrays.sort(nums1);
4. }

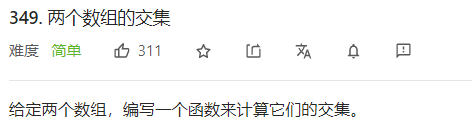
### [15. 三数之和](https://leetcode-cn.com/problems/3sum/)



排序 + 双指针

1. **public** List<List<Integer>> threeSum(**int**[] nums) {
2. int n = nums.length;
3. Arrays.sort(nums);
4. List<List<Integer>> ans = new ArrayList<>();
5. **for** (**int** first = 0; first < n; ++first) {  //枚举a
6. // 需要和上一次枚举的数不相同
7. **if** (first > 0 && nums[first] == nums[first - 1]) **continue**;
8. // c 对应的指针初始指向数组的最右端
9. **int** third = n - 1;
10. **int** target = -nums[first];
11. // 枚举 b
12. **for** (**int** second = first + 1; second < n; ++second) {
13. // 需要和上一次枚举的数不相同
14. **if** (second > first + 1 && nums[second] == nums[second - 1]) {
15. **continue**;
16. }
17. // 需要保证 b 的指针在 c 的指针的左侧
18. **while** (second < third && nums[second] + nums[third] > target) {                      --third;
19. }
20. //指针重合，随着b后续增加，就不满足a+b+c=0且b<c的c了，可退出循环
21. **if** (second == third) **break**;
22. **if** (nums[second] + nums[third] == target) {
23. List<Integer> list = **new** ArrayList<Integer>();
24. list.add(nums[first]);
25. list.add(nums[second]);
26. list.add(nums[third]);
27. ans.add(list);
28. }
29. }
30. }
31. **return** ans;
32. }

### [349. 两个数组的交集](https://leetcode-cn.com/problems/intersection-of-two-arrays/)



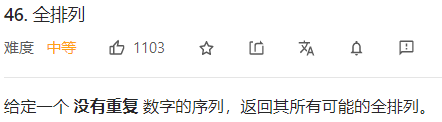
排序 + 双指针

1. **public** **int**[] intersection(**int**[] nums1, **int**[] nums2) {
2. Arrays.sort(nums1);
3. Arrays.sort(nums2);
4. **int** length1 = nums1.length, length2 = nums2.length;
5. **int**[] intersection = **new** **int**[length1 + length2];
6. **int** index = 0, index1 = 0, index2 = 0;
7. **while** (index1 < length1 && index2 < length2) {
8. **int** num1 = nums1[index1], num2 = nums2[index2];
9. **if** (num1 == num2) {
10. // 保证加入元素的唯一性
11. **if** (index == 0 || num1 != intersection[index - 1]) {
12. intersection[index++] = num1;
13. }
14. index1++;
15. index2++;
16. } **else** **if** (num1 < num2) {
17. index1++;
18. } **else** {
19. index2++;
20. }
21. }
22. **return** Arrays.copyOfRange(intersection, 0, index);
23. }

## 搜索回溯

回溯法：一种通过探索所有可能的候选解来找出所有的解的算法。如果候选解被确认不是一个解的话（或者至少不是最后一个解），回溯算法会通过在上一步进行一些变化抛弃该解，即回溯并且再次尝试。

### [46. 全排列](https://leetcode-cn.com/problems/permutations/)



搜索回溯

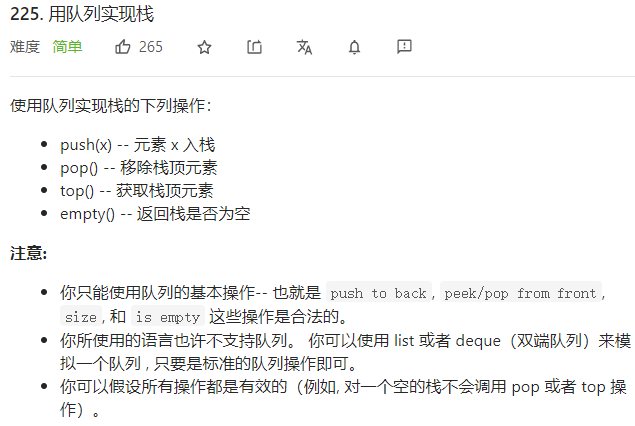
1. **public** List<List<Integer>> permute(**int**[] nums) {
2. List<List<Integer>> res = **new** ArrayList<>();
3. List<Integer> output = **new** ArrayList<Integer>();
4. **for** (**int** num : nums) output.add(num);
5. **int** n = nums.length;
6. backtrack(n, output, res, 0);
7. **return** res;
8. }
10. **void** backtrack(**int** n, List output, List<List<Integer>> res, **int** first) {
11. **if** (first == n) {
12. res.add(**new** ArrayList<Integer>(output));
13. }
14. **for** (**int** i = first; i < n; i++) {
15. // 动态维护数组
16. Collections.swap(output, first, i);
17. // 继续递归填下一个数
18. backtrack(n, output, res, first + 1);
19. // 撤销操作
20. Collections.swap(output, first, i);
21. }
22. }

## 深度/广度优先

# 第三章 数据结构

## 3.1 栈

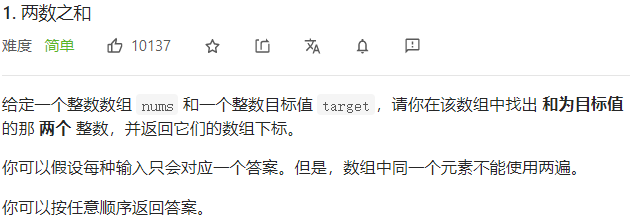
### [225. 用队列实现栈](https://leetcode-cn.com/problems/implement-stack-using-queues/)



1. **class** MyStack {
3. **private** Deque<Integer> queue;
5. **public** MyStack() {
6. queue=**new** LinkedList();
7. }
9. **public** **void** push(**int** x) {
10. queue.addFirst(x);
11. }
13. **public** **int** pop() {
14. **return** queue.remove();
15. }
17. **public** **int** top() {
18. **return** queue.element();
19. }
21. **public** **boolean** empty() {
22. **return** queue.size()==0;
23. }
24. }

## 3.2 哈希表

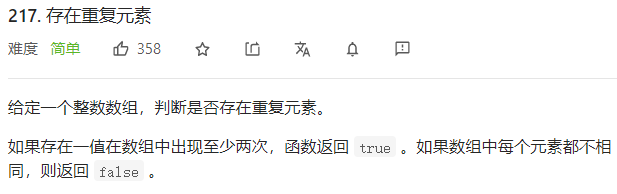
### 1.两数之和（Two Sum）



1. **public** **int**[] twoSum(**int**[] nums, **int** target) {
2. Map<Integer, Integer> map = **new** HashMap<Integer, Integer>();
3. **for** (**int** i = 0; i < nums.length; ++i) {
4. **if** (map.containsKey(target - nums[i])) {
5. **return** **new** **int**[]{map.get(target - nums[i]), i};
6. }
7. map.put(nums[i], i);
8. }
9. **return** **new** **int**[0];
10. }

* 时间复杂度：O(n)，n是数组的元素数量。map可以 O(1)寻找 target - x。
* 空间复杂度：O(n)，n是数组中的元素数量。主要为哈希表的开销。

### [217. 存在重复元素](https://leetcode-cn.com/problems/contains-duplicate/)

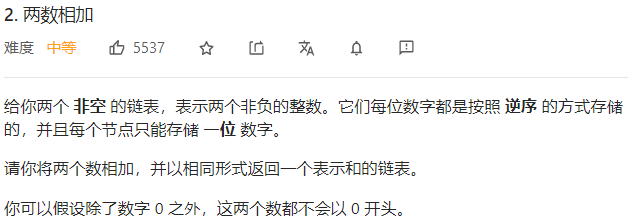


哈希表

1. **public** **boolean** containsDuplicate(**int**[] nums) {
2. Set<Integer> set = new HashSet<Integer>();
3. **for** (**int** x : nums) {
4. **if** (!set.add(x)) {
5. **return** **true**;
6. }
7. }
8. **return** **false**;
9. }

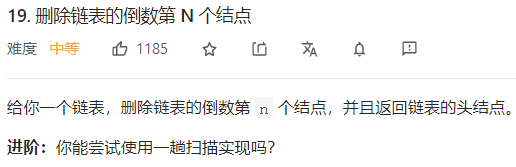
## 3.3 链表

### 2.两数相加（Add Two Numbers）



1. **public** ListNode addTwoNumbers(ListNode l1, ListNode l2) {
2. ListNode head=**null**,tail=**null**;
3. **int** carry=0;
4. **while** (l1 != **null**||l2 != **null**) {
5. **int** val1=l1==**null**?0:l1.val;
6. **int** val2=l2==**null**?0:l2.val;
7. **int** sum=val1+val2+carry;
8. **if**(head==**null**){
9. tail= head=**new** ListNode(sum%10);
10. }**else** {
11. tail.next=**new** ListNode(sum%10);
12. tail=tail.next;
13. }
14. carry=sum/10;
15. **if**(l1 != **null**){l1=l1.next;}
16. **if**(l2 != **null**){ l2=l2.next;}
17. }
18. **if**(carry>0){
19. tail.next = **new** ListNode(carry);
20. }
21. **return** head;
22. }

### 19. 删除链表的倒数第 N 个结点（Remove Nth Node From End of List）



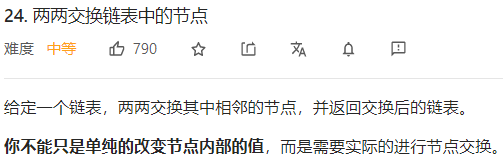
1. **public** ListNode removeNthFromEnd(ListNode head, **int** n) {
2. ListNode dummy = **new** ListNode(0, head);
3. ListNode second = dummy;
4. ListNode first = head;
5. **for** (**int** i = 0; i < n; ++i) {
6. first = first.next;
7. }
8. **while** (first != **null**) {
9. first = first.next;
10. second = second.next;
11. }
12. second.next = second.next.next;
13. **return** dummy.next;
14. }

### 21. 合并两个有序链表



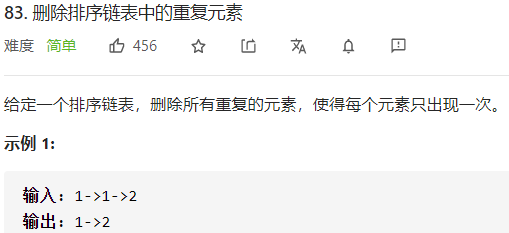
1. **public** ListNode mergeTwoLists(ListNode l1, ListNode l2) {
2. ListNode head = **new** ListNode(-1);
3. ListNode tail=head;
4. **while**(l1!=**null**&&l2!=**null**){
5. **if**(l1.val<l2.val){
6. tail.next=l1;
7. l1=l1.next;
8. }**else**{
9. tail.next=l2;
10. l2=l2.next;
11. }
12. tail=tail.next;
13. }
14. tail.next=l1!=**null**?l1:l2;
15. **return** head.next;
16. }

### 24. 两两交换链表中的节点



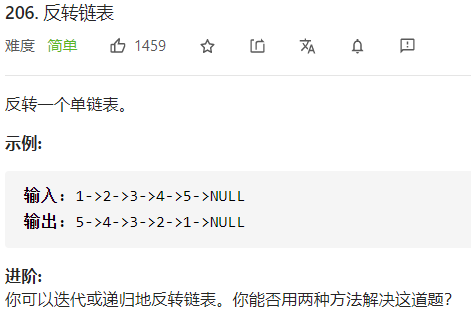
1. **public** ListNode swapPairs(ListNode head) {
2. **if**(head==**null**||head.next==**null**){
3. **return** head;
4. }
5. ListNode curr = head,pre=**null**;
6. head = head.next;
7. **while** (curr != **null**) {
8. ListNode next = curr.next;
9. **if** (next == **null**) {
10. **break**;
11. }
12. curr.next = next.next;
13. next.next = curr;
14. **if**(pre!=**null**)
15. pre.next=next;
16. pre=curr;
17. curr = curr.next;
18. }
19. **return** head;
20. }

### 83. 删除排序链表中的重复元素



1. **public** ListNode deleteDuplicates(ListNode head) {
2. ListNode cur=head;
3. **while**(cur!=**null**&&cur.next!=**null**){
4. **if**(cur.val==cur.next.val){
5. cur.next=cur.next.next;
6. **continue**;
7. }
8. cur=cur.next;
9. }
10. **return** head;
11. }

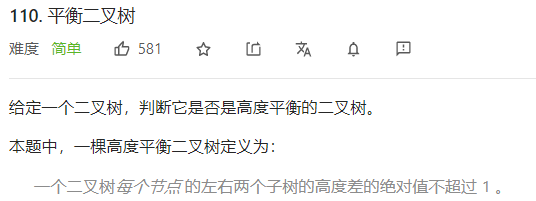
### 206. 反转链表



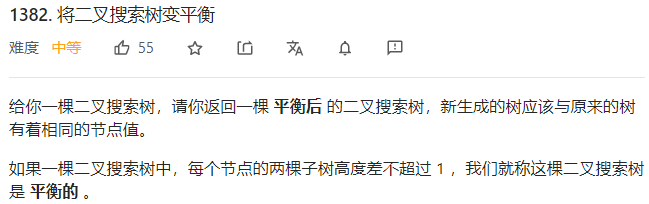
1. **public** ListNode reverseList(ListNode head) {
2. ListNode prev = **null**;
3. ListNode curr = head;
4. **while** (curr != **null**) {
5. ListNode next = curr.next;
6. curr.next = prev;
7. prev = curr;
8. curr = next;
9. }
10. **return** prev;
11. }

## 3.4 树

### [110. 平衡二叉树](https://leetcode-cn.com/problems/balanced-binary-tree/)



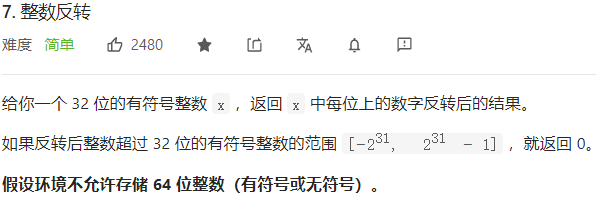
### [1382. 将二叉搜索树变平衡](https://leetcode-cn.com/problems/balance-a-binary-search-tree/)



# 第四章 Leetcode

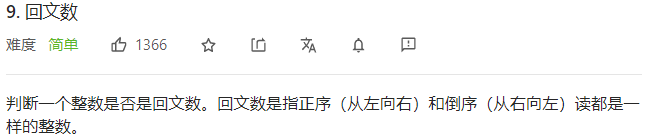
## 整数

### 7.整数反转



1. **public** **int** reverse(**int** x) {
2. **int** reverseNum=0;
3. **while**(x!=0){
4. **if**((reverseNum>Integer.MAX\_VALUE/10)
5. ||(reverseNum<Integer.MIN\_VALUE/10)){
6. **return** 0;
7. }
8. reverseNum=reverseNum\*10+x%10;
9. x=x/10;
10. }
11. **return** reverseNum;
12. }

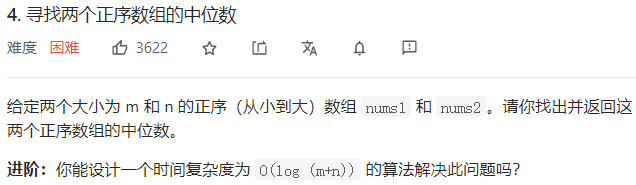
### 9. 回文数



1. **public** **boolean** isPalindrome(**int** x) {
2. **int** n=0,m=x;
3. **while**(m>0){
4. n=10\*n+m%10;
5. m=m/10;
6. }
7. **return** n==x;
8. }

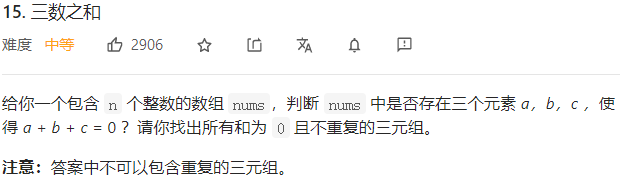
## 数组

### 4. 寻找两个正序数组的中位数



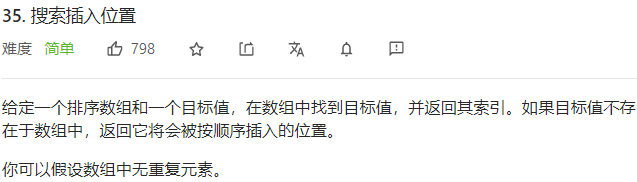
1. **public** **double** findMedianSortedArrays(**int**[] nums1, **int**[] nums2) {
2. **int** [] nums3=**new** **int**[nums1.length+nums2.length];
3. **int** i= nums1.length-1,j=nums2.length-1;
4. **int** t=nums3.length-1;
5. **while**(i>=0&&j>=0){
6. nums3[t--]=nums1[i]>nums2[j]?nums1[i--]:nums2[j--];
7. }
8. **while**(i>=0){
9. nums3[t--]=nums1[i--];
10. }
11. **while**(j>=0){
12. nums3[t--]=nums2[j--];
13. }
14. **if**(nums3.length%2==1) {
15. **return** nums3[(nums3.length-1)/2];
16. }**else**{
17. **int** sum= nums3[(nums3.length-1)/2]+nums3[(nums3.length-1)/2+1];
18. **return** (**double**) sum/2;
19. }
20. }

### 15. 三数之和



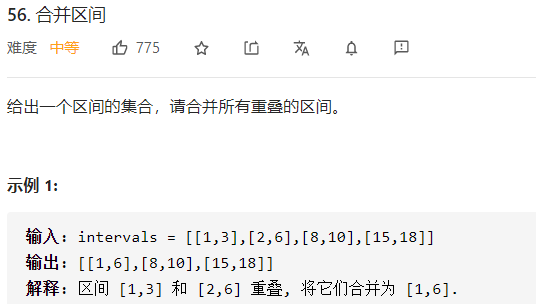
TODO

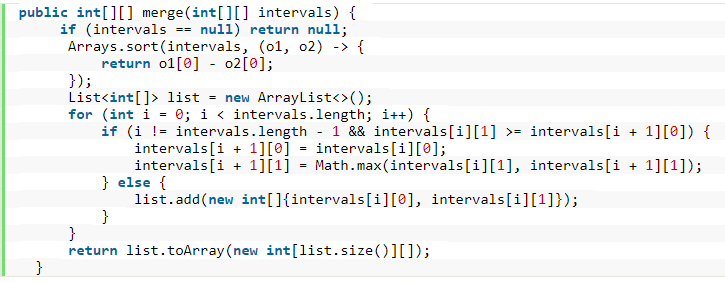
### 35. 搜索插入位置



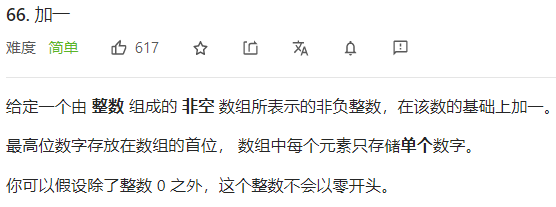
1. **public** **int** searchInsert(**int**[] nums, **int** target) {
2. **int** left=0,right=nums.length-1;
3. **while**(left<=right){
4. **int** mid =(left +right)/2;
5. **if**(target<nums[mid]){
6. right= mid-1;
7. }**else** **if**(target>nums[mid]){
8. left=mid+1;
9. }**else**{
10. **return** mid;
11. }
12. }
13. **return** left;
14. }

### 56. 合并区间



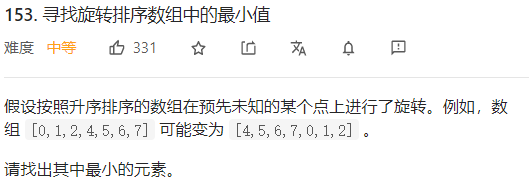


### 66. 加一



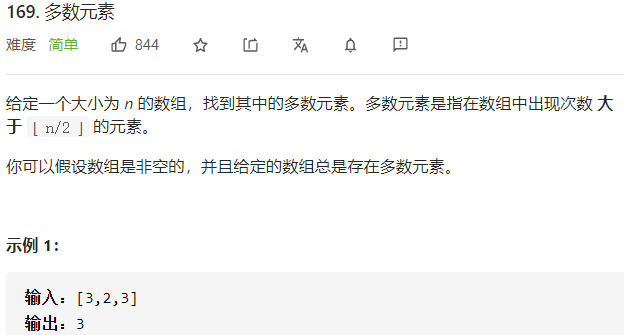
1. **public** **int**[] plusOne(**int**[] digits) {
2. **for**(**int** i=digits.length-1;i>=0;i--){
3. digits[i]++;
4. digits[i]=digits[i]%10;
5. **if**(digits[i]!=0){
6. **return** digits;
7. }
8. }
9. //出现999..
10. digits= **new** **int**[digits.length + 1];
11. digits[0] = 1;
12. **return** digits;
13. }

### 153. 寻找旋转排序数组中的最小值



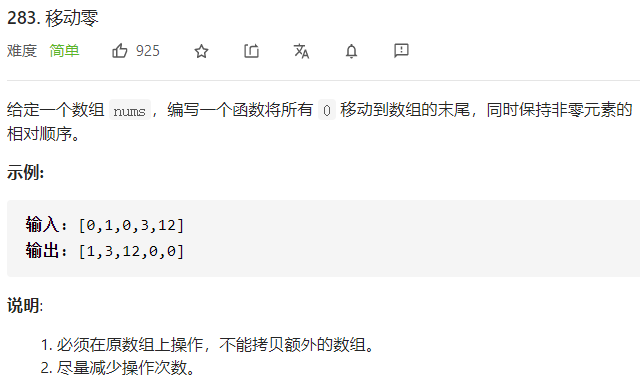
1. **public** **int** findMin(**int**[] nums) {
2. **if**(nums.length==1) **return** nums[0];
3. **for**(**int** i=1;i<nums.length;i++){
4. **if**(nums[i]<nums[i-1]){
5. **return** nums[i];
6. }
7. }
8. **return** nums[0];
9. }

### 169. 多数元素



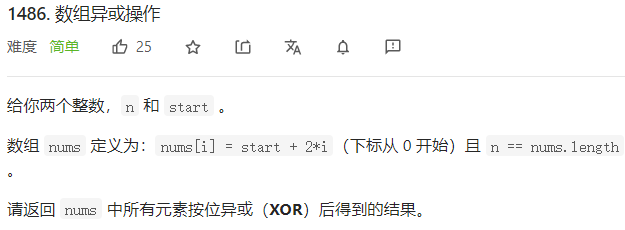
1. Map<Integer,Integer> map=**new** HashMap<>();
2. **public** **int** majorityElement(**int**[] nums) {
3. **for**(**int** i=0;i<nums.length;i++){
4. **int** val= map.get(nums[i])==**null**?0:map.get(nums[i]);
5. map.put(nums[i],++val);
6. **if**(map.get(nums[i])>nums.length/2){
7. **return** nums[i];
8. }
9. }
10. **return** 0;
11. }

### 283. 移动零



1. **public**  **void** moveZeroes(**int**[] nums) {
2. **int** n = nums.length, left = 0, right = 0;
3. **while** (right < n) {
4. **if** (nums[right] != 0) {
5. swap(nums, left, right);
6. left++;
7. }
8. right++;
10. }
11. }
13. **public**  **void** swap(**int**[] nums, **int** left, **int** right) {
14. **if**(left==right) **return**;
15. **int** temp = nums[left];
16. nums[left] = nums[right];
17. nums[right] = temp;
18. }

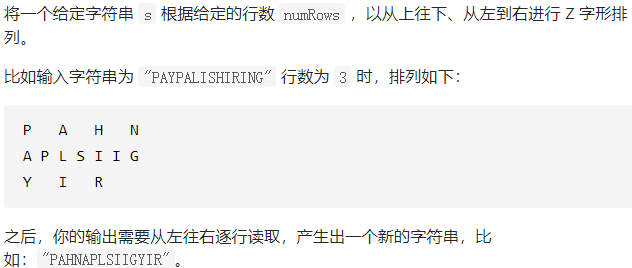
### [1486. 数组异或操作](https://leetcode-cn.com/problems/xor-operation-in-an-array/)



1. **public** **int** xorOperation(**int** n, **int** start) {
2. **int** [] nums=**new** **int**[n];
3. **int** result=0;
4. **for**(**int** i=0;i<n;i++){
5. nums[i]=start + 2\*i;
6. **if**(i>0){
7. nums[i]=nums[i]^nums[i-1];
8. }
9. }
10. **return** nums[n-1];
11. }

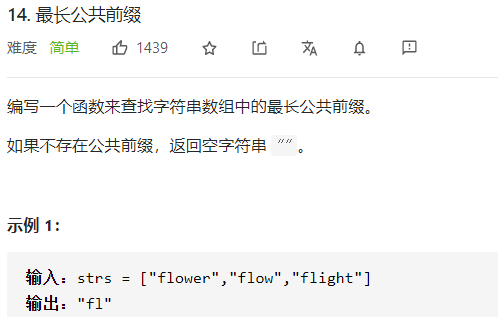
## 字符串

### Z 字形变换



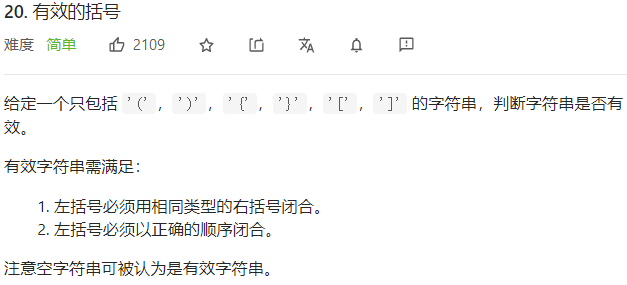
1. **public** String convert(String s, **int** numRows) {
2. **if**(s==**null**||s.length()==0) **return** **null**;
3. StringBuilder [] array=**new** StringBuilder[numRows];
4. **for**(**int** i=0;i<array.length;i++){
5. array[i]=**new** StringBuilder();
6. }
7. **int** d=numRows==1?0:1;
8. **int** index=0;
9. **for**(**char** c : s.toCharArray()){
10. array[index].append(c);
11. index+=d;
12. **if**(index==0||index==numRows-1)  d=-d;
13. }
14. StringBuilder sb=**new** StringBuilder();
15. **for**(**int** i=0;i<array.length;i++){
16. sb.append(array[i].toString());
17. }
18. **return** sb.toString();
19. }

### [14. 最长公共前缀](https://leetcode-cn.com/problems/longest-common-prefix/)



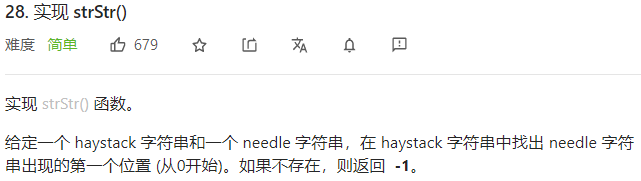
1. **public** String longestCommonPrefix(String[] strs) {
2. **if**(strs.length==0) **return** "";
3. String s = strs[0];
4. **int** i=0;
5. **for** (; i < s.length(); i++) {
6. String sub = s.substring(i, i+1);
7. **for** (**int** j = 0; j < strs.length; j++) {
8. String str = strs[j];
9. **if** (i>=str.length()||!sub.equals(str.substring(i, i+1))) {
10. **return** s.substring(0,i);
11. }
12. }
13. }
14. **return** s.substring(0,i);

### 20. 有效的括号



1. **public**  **boolean** isValid(String s) {
2. Deque<Character> q=**new** LinkedList<>();
3. **for**(**int** i=0;i<s.length();i++){
4. **if**(q.peek()!=**null**&& getRes(q.peek())==**null**){
5. **return** **false**;
6. }
7. **if**(q.peek()!=**null**&& getRes(q.peek())==s.charAt(i)){
8. q.pop();
9. }**else**{
10. q.push(s.charAt(i));
11. }
12. }
13. **return** q.size()==0;
14. }
16. **public**  Character getRes(Character c){
17. **if**(c=='(')**return** ')';
18. **else** **if** (c=='{')**return** '}';
19. **else** **if** (c=='[')**return** ']';
20. **return** **null**;
21. }

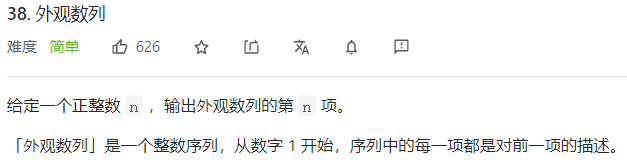
### 28. Implement strStr()



1. **public** **int** strStr(String haystack, String needle) {
2. **int** L = needle.length(), n = haystack.length();
3. **for** (**int** i = 0; i <= n - L; ++i) {
4. **if** (haystack.substring(i, i + L).equals(needle)) {
5. **return** i;
6. }
7. }
8. **return** -1;
9. }

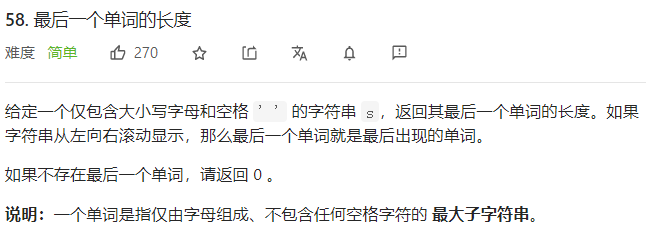
* 时间复杂度：O((N - L)L)
* 空间复杂度：O(1)O(1)

### 38. 外观数列



1. **public** String countAndSay(**int** n) {
2. String[] r=**new** String[n];
3. r[0]="1";
4. **for**(**int** i=1;i<n;i++){
5. StringBuilder sb=**new** StringBuilder();
6. String s= r[i-1];
7. **for**(**int** j=0; j<s.length();){
8. **int** count=1;
9. **char** c=s.charAt(j);
10. **while**(++j<s.length()&&s.charAt(j)==c){
11. count++;
12. }
13. sb.append(count).append(c);
14. }
15. r[i]=sb.toString();
16. }
17. **return** r[n-1];
18. }

### 58. 最后一个单词的长度



1. **public** **int** lengthOfLastWord(String s) {
2. **if**(s==**null**||s.length()==0){
3. **return** 0;
4. }
5. **int** len=0;
6. **int** index=s.length()-1;
7. **while**(index>=0){
8. **if**(s.charAt(index)==' '){
9. index--;
10. **continue**;
11. }
12. **while**(index>=0&&s.charAt(index--)!=' '){
13. len++;
14. }
15. **return** len;
16. }
17. **return** len;
18. }

## 树

### 104. Maximum Depth of Binary Tree

二叉树最大深度

1. **public** **int** maxDepth(TreeNode root) {
2. **if**(root==**null**) **return** 0;
3. **int** leftHeight = maxDepth(root.left);
4. **int** rightHeight = maxDepth(root.right);
5. **return** Math.max(leftHeight, rightHeight) + 1;
6. }

### 110. Balanced Binary Tree

平衡二叉树

1. **public** **boolean** isBalanced(TreeNode root) {
2. **if**(root==**null**) **return** **true**;
3. **int** r= Math.abs(maxDepth(root.left)-maxDepth(root.right));
4. **return** r<=1&&isBalanced(root.left)&&isBalanced(root.right);
5. }

## 其他

### 13. 罗马数字转整数

# 第五章 查找

查找算法

顺序查找：复杂度O(n)，缺点：的算法在数据量很大时显然是糟糕的

二分查找（binary search）：要求被检索数据有序。

二叉树查找（binary tree search）等。，只能应用于二叉查找树上

## 4.1 顺序表的查找

顺序表指线性表的顺序存储结构。

### 顺序查找

Sequential Search。又称线性查找。从表的一端开始，顺序扫描线性表，依次比较关键字和目标值。

1. **public** **static** **int** search(**int**[] arr, **int** keyword) {
2. **for** (**int** i = 0; i < arr.length; i++) {
3. **if** (arr[i] == keyword) {
4. **return** i;
5. }
6. }
7. **return** -1;
8. }

### 二分查找

Binary Search。折半查找

1. **static** **int** search(**int** arr[], **int** keyword) {
2. **int** low = 0;
3. **int** high = arr.length - 1;
4. **while** (low <= high) {
5. **int** mid = low + (high - low) / 2;
6. **if** (arr[mid] > keyword) {
7. high = mid - 1;
8. } **else** **if** (arr[mid] < keyword) {
9. low = mid + 1;
10. } **else** {
11. **return** mid;
12. }
13. }
14. **return** -1;
15. }

递归

TODO

### 索引顺序查找

### 三种查找方法的比较

分块查找

每块不一定有序，但分块有序。将[1..n]分为n块，前b-1快都有[n/b];抽取每块最大关键字组成ID[1..b];由于块内无序，只能用顺序查找。

## 4.2 散列表1的查找

### 5.3.1 散列表的概念

# 第六章 排序

基本概念

内部排序：排序过程都放在内存不涉及内外存交换。大多数算法的开销是记录的比较和记录的移动，算法所需辅助空间不依赖于问题的规模n，辅助空间是O（1）。又称就地排序。

外部排序：指记录数量很大，内存一次不能容纳全部记录，需辅助空间。

算法评价标准

时间开销：关键字比较次数和记录移动次数。

时间开销及需要的附加空间。另外算法本身的复杂度也是考虑的重要因素。

本章讨论中，都假定顺序表作为存储结构

1. #define MAXSIZE 100
2. **typedef** **int** KeyType;
3. **typedef** **struct**{
4. KeyType key;
5. InfoType otherinfo;
6. } RecType;
7. **typedef** RecType SeqList[MAXSIZE+1];//0元素空着或用作哨兵单元
8. SeqList R;  //R为待排序的记录文件

* 内部排序

内部排序方法的分析比较

* 外部排序

外存信息的存取

外部排序的方法

## 插入排序

基本思想：每次将待排序的记录插入前面已排序文件的适当位置。

### 直接插入排序

直接插入排序（Straight Insertion Sort）是一种最简单的排序方法，其基本操作是待排序的记录插入到已排好的有序表中

1. public **void** insertSort(**int**[] arr) {
2. **int** j; // 已排序列表下标
3. **for** (**int** i = 1; i < arr.length; i++) {
4. **if** (arr[i] < arr[i - 1]) {
5. **int** temp = arr[i];
6. **for** (j = i - 1; j >= 0 && temp < arr[j]; j--) {
7. arr[j + 1] = arr[j]; //后移
8. }
9. arr[j + 1] = temp; // 插入待排序元素
10. }
11. }
12. }

时间复杂度：O(n2)

[动画演示](https://www.runoob.com/wp-content/uploads/2019/03/insertionSort.gif)

### 5.1.2 希尔排序

分组的的插入排序

void shellSort(Seqlist R)

{

int increment=1;

do{

increment=increment/3+1;

shellPass(R,increment);

}while(increment>1)

}

void shellPass(Seqlist R,int d)

{

int i,j;

for(i=d+1;i<=n;i++)

{

a[0]=a[i]; //搬走后留下空位置

j=i-d;

while(a[0]<a[j]&&j>0)

{

a[j+d]=a[j]; //搬走后留下空位置

j=j-d;

}

a[j+d]=a[0];

}

}

## 5.2 交换排序

两两比较待排序记录的关键字，若次序相反即进行交换，直到所有记录都没有反序为止。

[动画演示](https://www.runoob.com/wp-content/uploads/2019/03/bubbleSort.gif)

### 冒泡排序

1. **public** **void** bubbleSort(**int**[] arr) {
2. **for** (**int** i = 1; i < arr.length; i++) {
3. **boolean** flag = **true**;//若为true，则表示此次循环没有进行交换
4. **for** (**int** j = 0; j < arr.length - i; j++) {
5. **if** (arr[j] > arr[j + 1]) {
6. **int** tmp = arr[j];
7. arr[j] = arr[j + 1];
8. arr[j + 1] = tmp;
9. flag = **false**;
10. }
11. }
12. **if** (flag) **break**;
13. }
14. }

时间复杂度

最好情况：数组是正序。比较次数：n-1； 移动：0。比较的时间复杂度为O(n)

最坏情况：O(n2)，数组是反序。

### 快速排序

[Reference](https://www.runoob.com/w3cnote/quick-sort-2.html)

Quick Sort，对冒泡排序一种改进。通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

1. **public** **void** sort(**int**[] arr) {
2. quickSort(arr, 0, arr.length - 1);
3. }
5. **private** **int**[] quickSort(**int**[] arr, **int** left, **int** right) {
6. **if** (left < right) {
7. **int** partitionIndex = partition(arr, left, right);
8. quickSort(arr, left, partitionIndex - 1);
9. quickSort(arr, partitionIndex + 1, right);
10. }
11. **return** arr;
12. }
13. // 以left为基准将数据分为两部分
14. **private** **int** partition(**int**[] arr, **int** left, **int** right) {
15. **int** pivot = left;// 设定基准值（pivot）
16. **int** index = pivot + 1;
17. **for** (**int** i = index; i <= right; i++) {
18. **if** (arr[i] < arr[pivot]) {
19. swap(arr, i, index);
20. index++;
21. }
22. }
23. swap(arr, pivot, index - 1);
24. **return** index - 1;
25. }
27. **private** **void** swap(**int**[] arr, **int** i, **int** j) {
28. **int** temp = arr[i];
29. arr[i] = arr[j];
30. arr[j] = temp;
31. }

时间复杂度

平均的时间复杂度是O(nlog2n)，最坏情况是O(n2)

## 5.3 选择排序

### 4.3.1 直接选择排序

每次查找从所有无序区中找到最小的放到已排序序列。，

1. **public** **void** sort(**int**[] arr) {
2. **for** (**int** i = 0; i < arr.length - 1; i++) {
3. **int** min = i;
4. **for** (**int** j = i + 1; j < arr.length; j++) {
5. **if** (arr[j] < arr[min]) {
6. min = j;
7. }
8. }
9. **if** (i != min) {
10. **int** tmp = arr[i];
11. arr[i] = arr[min]; //arr[i]是已排序队列
12. arr[min] = tmp;
13. }
14. }
15. }

比较的时间复杂度：n(n-1)/2= O(n2)

### 4.3.2 堆排序

改进直接选择排序。直接选择排序在查找关键字所进行的比较中，很多很可能在前面的已做过，只是当时没将这些结果保存下来，在后一趟排序时又重复了这些比较操作，树形排序可以克服这点。

基本思想：记录R[1...n]是一棵完全二叉树的顺序存储结构，利用节点的内在关系，在当前无序区中选择最大（小）记录。

小根堆：ki<=K2i且Ki<=K2i+1

大根堆：ki>=K2i且Ki>=K2i+1

[图解排序算法(三)之堆排序](https://www.cnblogs.com/chengxiao/p/6129630.html)

Sort

1. **public** **static** **void** sort(**int**[] arr) {
2. //1.构建大顶堆，使其每个元素满足顶堆条件
3. **for** (**int** i = arr.length / 2 - 1; i >= 0; i--) {
4. adjustHeap(arr, i, arr.length);
5. }
6. //2.调整+调整
7. **for** (**int** i = arr.length - 1; i > 0; i--) {
8. swap(arr, 0, i);//将堆顶元素与末尾元素进行交换
9. adjustHeap(arr, 0, i);//重新对堆进行调整
10. }
11. }
12. **void** adjustHeap(**int**[] arr, **int** parent, **int** length) {
13. **int** temp = arr[parent];
14. **int** child = parent \* 2 + 1;  //从parent的左子结点开始，从上往下
15. **for** (; child < length; child = child \* 2 + 1) {
16. **if** (child + 1 < length && arr[child] < arr[child + 1]) {
17. child++;
18. }
19. **if** (arr[child] > temp) {
20. arr[parent] = arr[child];
21. parent = child;
22. } **else** {
23. **break**;
24. }
25. }
26. arr[parent] = temp;
27. }

时间复杂度

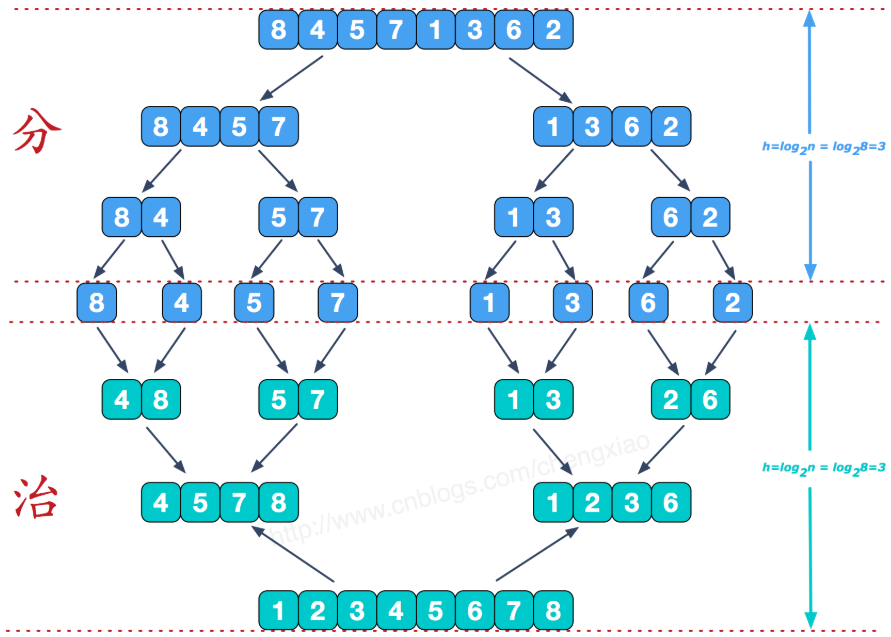
在堆排序中，一次堆构建O(log2n)。要n-1趟选择，所以整个堆排序的时间复杂度为O(nlog2n)

### 4.3.3 树形选择排序

同简单选择排序相比，该算法减少了不同记录之间的比较次数，但是程序运行所需要的空间较多。

## 5.4 归并排序

[图解排序算法(四)之归并排序](https://www.cnblogs.com/chengxiao/p/6194356.html)



归并排序的两种实现：自底向上和自顶向下。即为**二路归并排序**。

分

1. **void** sort(**int**[] arr, **int** left, **int** right) {
2. **if** (left >= right)  **return**;
3. **int** mid = (left + right) / 2;
4. sort(arr, left, mid);
5. sort(arr, mid + 1, right);
6. merge(arr, left, mid, right);
7. }

合并

1. **void** merge(**int**[] arr, **int** left, **int** mid, **int** right) {
2. **int**[] temp = **new** **int**[right - left + 1];
3. **int** i = left;
4. **int** j = mid + 1;
5. **int** t = 0;
6. **while** (i <= mid && j <= right) {
7. temp[t++] = arr[i] <= arr[j] ? arr[i++] : arr[j++];
8. }
9. **while** (i <= mid) {
10. temp[t++] = arr[i++];
11. }
12. **while** (j <= right) {
13. temp[t++] = arr[j++];
14. }
15. **while** (left <= right) {
16. arr[right--] = temp[--t];
17. }
18. }

* 时间复杂度：O(nlog2n)。首先是折半拆分，由二分查找算法可知，长度为n需log2n次拆分。然后是归并，长度n需n次归并，则总的时间复杂度为O(nlog2n)。。
* 空间复杂度：O(n)。我们需要额外O(n)空间的tmp数组，且归并排序递归调用的层数最深为log2n，所以还需要额外的O(log2n)的栈空间，所需空间复杂度即O(n+ log2n) =O(n)。

## 5.5 分配排序

### 桶排序

Bucket Sort，也称箱排序（Bin Sort）

箱个数取决关键字取值范围，R[0…n-1],分配时间O（n），收集时间O（m）或O（m+n）

1箱子容量不可预知，2保持稳定故箱子采用**链队列**

注意：适用于关键字取值范围较小。

箱排序的变种：桶排序，假设输入n个关键字是分布在[0,1）上，找某个合适的数映射到此区间。区间是一个桶。指针B[0…n-1]表示n个桶，记录0<=R[i]<=n-1通过n\*(R[i].key)

### 基数排序

## 5.6 外部排序

外部排序指大文件的排序，即待排序的记录存储在外存储器上，在排序过程中需进行多次的内外存的交换。

### 外存信息的存取

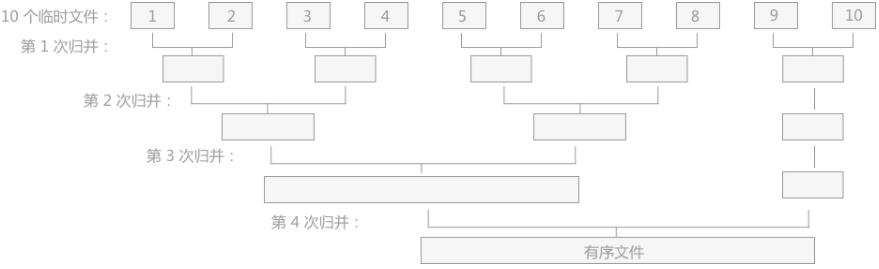
### 外部排序的方法

外部排序两个阶段

* 按可用内存大小将外存上文件分为若干的子文件或段(segment)，依次读入内存并用内部排序方法排序，并将排序后得到的有序子文件重新写入外存，有序子文件为归并段；
* 对这些归并段进行逐趟归并，直至得到一个完整的有序文件。

例如，有一个含有 10000个记录的文件，但是内存的可使用容量为1000，需用外部排序算法，具体分为两步：

* 将文件其等分为 10个临时文件，然后将这 10 个文件依次进入内存，采取适当的内存排序算法对其中的记录进行排序，将得到的有序文件（初始归并段）移至外存。
* 对归并段两两归并。



如图进行了4次归并，每次由m个归并段得到⌈m/2⌉个归并段，这种归并方式称为 2-路平衡归并

m个初始归并段进行k-路平衡归并，归并的次数为：s=logkm。

归并操作的次数越多，访问外存的次数就越多，减少归并次数从而提高算法效率的目的，从两个角度实现：

* 增加 k-路平衡归并中的k值；
* 尽量减少初始归并段的数量 m，即增加每个归并段的容量

败者树实现内部归并

败者树是树形选择排序的一种变形，本身是一棵完全二叉树。

# 第七章 树和二叉树

常见的数组、链表、栈和队列都是线性结构，在存储大量数据时访问速度比较慢，而树(Tree)则是一种非线性结构，使得访问时间复杂度降低到O(log2n)。

术语

* 叶子节点：没有子节点（即度为0），简称叶子。
* 度：节点的度指节点孩子数量。一棵树的度指最大孩子树。
* 阶：m阶B-tree指每个节点最多有m个子节点。
* 深度(k)=高度(h)
* 森林(Forest)：由多棵不相交的树的集合。

哈希适合等于性的查找，树结构适合”范围查找“，lucene适合字符串的查找，，数据组织要选择合适的的算法。

## 3.1 二叉树

Binary Tree。树中节点的度不大于2的有序树。常见的性质：

* 第i层上至多有2i-1个节点
* 深度为h的二叉树中至多含有2h-1个节点
* 叶节点个数为n0，度数为2的节点个数为n2，则有: n0 = n2 + 1

证明：(1)n= n0+n1+ n2 (n指数量)(2)n= n1+2\* n2+1 算孩子的总数

### 3.1.1 定义和性质

二叉树，森林，树的转换：

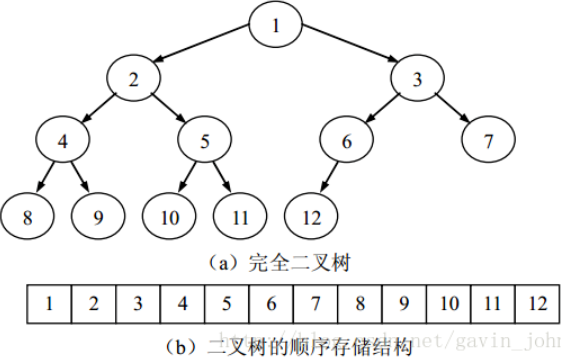
* 都去掉双亲到右孩子的线
* 二叉树—>右孩子到双亲
* 森林-->连接兄弟。双亲到右孩子
* 顺时针旋转45·

### 3.1.2 存储结构

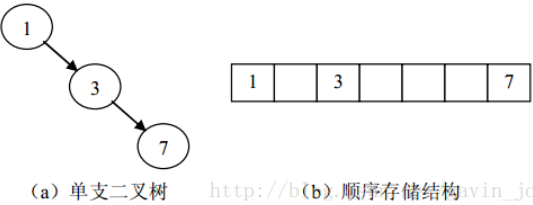
有两种，分别为顺序存储和链式存储

顺序存储

顺序存储一个具有n节点的完全二叉树，只要从树根从上到下，每层从左至右给每个节点进行编号，就能得到一个反应整个二叉树结构的线性序列。

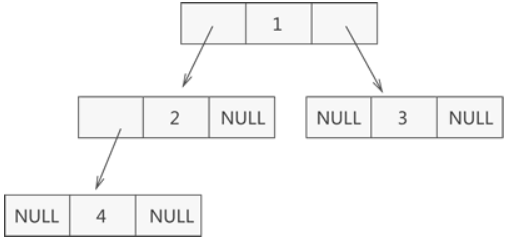


对于一般的二叉树而言，必须用“虚结点”将二叉树补成完全二叉树，否则无法确定结点之间关系，这样会造成存储空间的浪费。极端情况，对于单支二叉树，为了存储k个结点，需要2k-1个存储单元。



链式存储结构

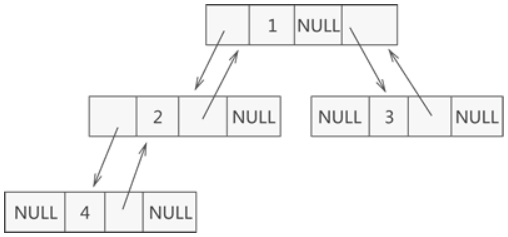
采用链式存储，只需从树的根节点开始，将各个节点及其左右孩子使用链表存储即可。



其节点结构

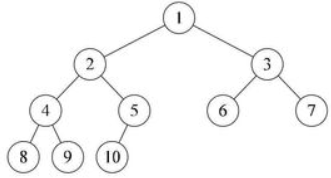
1. **public** **class** TreeNode<T> {
2. **private** T data;// 数据域
3. **private** TreeNode<T> left;
4. **private** TreeNode<T> right;
5. }

某些场景需查找某节点的父节点"，可在节点结构中再添加一个指针域，指向其父亲节点。这样的链表结构，通常称为三叉链表。可轻松找到各节点的父节点。



### 3.1.1 完全二叉树

* 只有最下面的两层结点度数小于2，其余各层的结点度数都等于2，
* 所有叶子节点都在最下一层且集中在最左边。

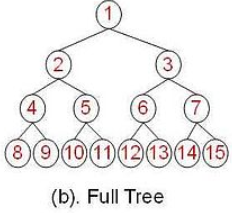


性质：

* 2k-2 < 节点数 <= 2k-1
* n个结点的完全二叉树的深度k=└log2n┘+1
* 非根节点的父节点的下标为(i-1)/2;
* 第一个非叶子结点 n/2
* 2i+1 <= n-1,则节点i的左子结点的下标为2i+1;否则没有左子结点.
* 2i+2 <= n-1,则节点i的右子节点的下标为2i+2;否则没有右子节点

### 3.1.2 满二叉树

Full Binary Tree。完全二叉树的特例。



特性

对树中的结点按从上至下、从左到右的顺序进行编号，编号为i（1≤i≤n）

* 节点数n=2k-1
* 节点i，左子节点编号是2i，而右子节点是2i +1。父节点下标i/2。
* 第h层节点数2h-1

## 3.2 二叉树的运算

二叉树常被用于实现二叉查找树和二叉堆。

### 3.2.1 二叉树的生成

生成指建立二叉树的存储结构。

建立二叉树的顺序存储结构比较简单，下面着重讨论如何建立二叉树的链式存储结构，即二叉链表。

按广义表

按完全二叉树的层次

### 3.2.2 二叉树的遍历

先序遍历

先访问根节点，然后访问左节点，最后访问右节点(根->左->右)

1. **public** **void** preOrder(TreeNode node) {
2. **if** (node != **null**) {
3. System.out.print(node.val);;
4. preOrder(node.left);
5. preOrder(node.right);
6. }
7. }

中序遍历

先访问左节点，然后访问根节点，最后访问右节点(左->根->右)

后序遍历

先访问左节点，然后访问右节点，最后访问根节点(左->右->根) }

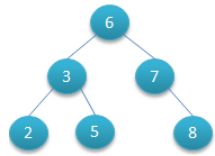
## 3.3 树的查找

### 3.3.1 二叉查找树(BST)

Binary Search Tree,又叫二叉排序树（Binary Sort Tree）、二叉搜索树。

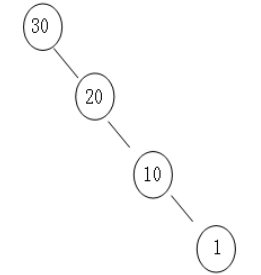
左子树<节点<右子树，中序遍历得到有序递增序列。通常采用**二叉链表**作为存储结构。

注意：不一定是完全二叉树

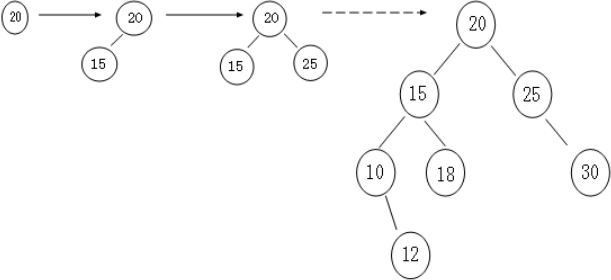


复杂度：O(log2n)~O(n)

二叉查找树在最坏的情况下会出现“链表”的形式，复杂度退化到O(N)，比如下图。



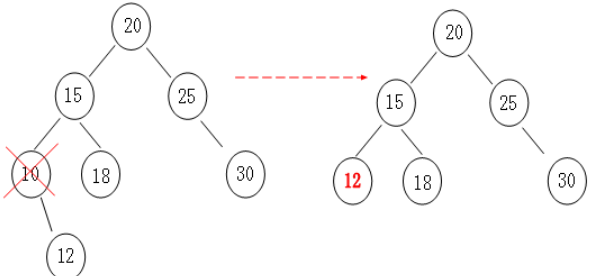
#### 添加节点



#### 删除

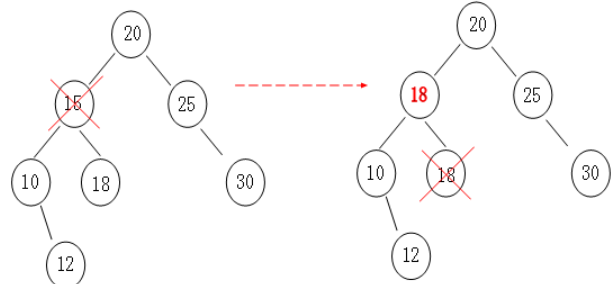
单孩子

如果删除的节点有左孩子那就把左孩子顶上去，如果有右孩子就把右孩子顶上去



左右都有孩子

根据”中序遍历“找到要删除结点的后一个结点，然后顶上去即可



### 3.3.2 平衡二叉树（AVL）

Balanced Binary Tree，又称AVL树。

* 本质上也是BST。
* 任一节点的左右子树最大高度差为1。

算法思路

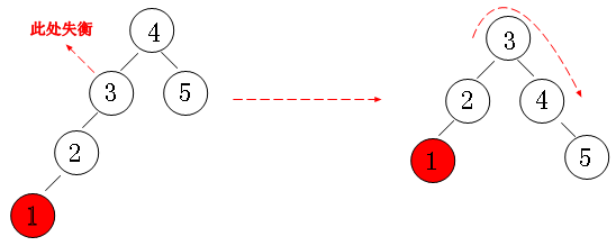
查找与二叉树一样，每次查询时对半分，最大的不同点：每次插入或者删除后计算节点高度，然后按需进行调整。树节点旋转成完全二叉树，减小树的深度，做到查找是严格的O(logN)

#### 插入场景

节点再怎么失衡都逃不过4种情况

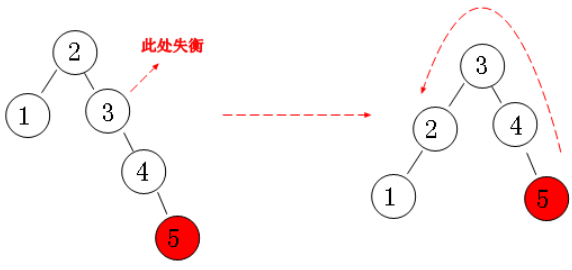
1）左左型(左子树的左边)

左子树的左边节点处添加节点。如图追加节点1时，导致了“节点3"失衡。把这棵树比作齿轮，我们在“节点5”处把齿轮往下拉一个位置，也就变成了后面这样“平衡”的形式。

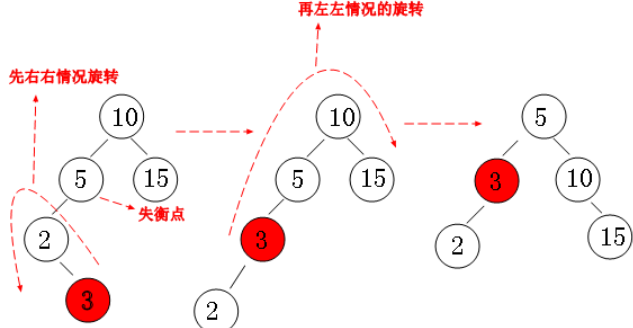


2）右右型(右子树的右边节点)

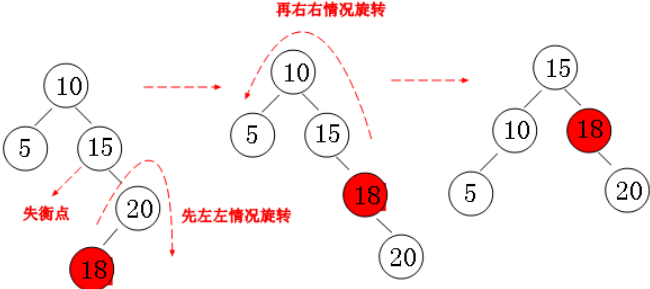
和左左型类似，节点1的地方将树往下拉一位，形成平衡效果。



3）左右型(左子树的右边节点)



4）右左型(右子树的左边节点)



[6天通吃树结构—— 第二天 平衡二叉树](https://www.cnblogs.com/huangxincheng/archive/2012/07/22/2603956.html)

#### 删除场景

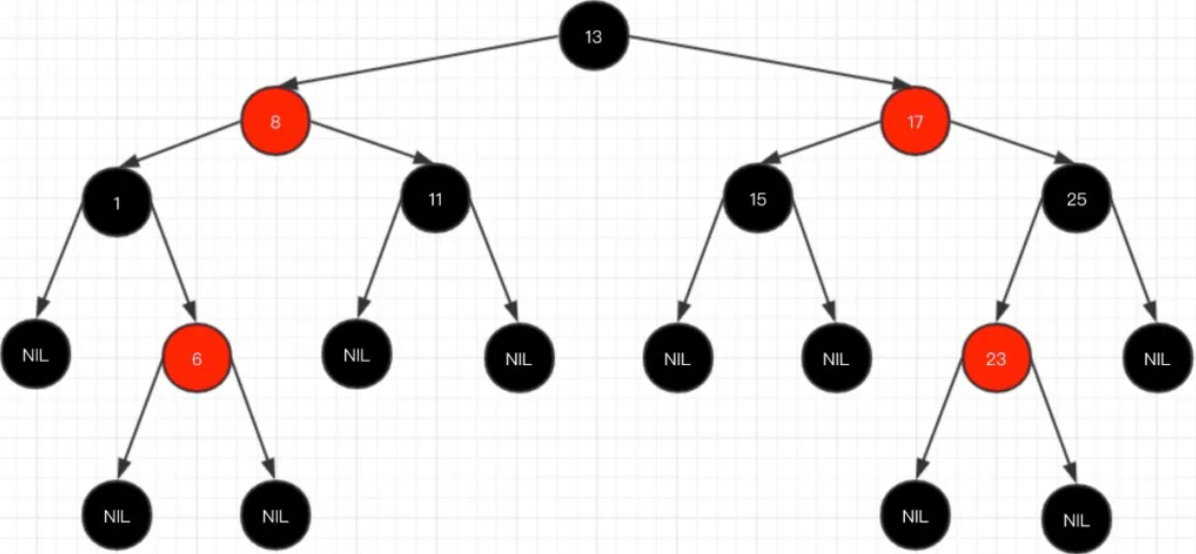
删除方法跟添加方法也类似，当删除一个结点的时候，可能会引起祖先结点的失衡，所以在每次”结点“回退的时候计算结点高度。

### 3.3.3 红黑树

红黑树是一种自平衡的二叉查找树。除了符合BST特性，还有下列特性：

* 节点是红色或黑色。
* 根节点是黑色。
* 叶节点（NIL）是黑色。(这里叶节点指为空的叶子节点）。
* 每个红节点的两个子节点是黑色。(路径上不能有两个连续红色节点)
* 性质5：任一节点到叶节点的路径包含相同数目的黑节点。

[漫画：什么是红黑树？](https://zhuanlan.zhihu.com/p/31805309)



AVL vs 红黑树

AVL的高度差不能大于1，删除时，可能需要多次调整，在树的深度很大的情况下，删除效率会非常低，（调整频率高？） 如何提高这种效率？红黑树由此诞生了。

红黑树的任一路径不会比其他路径长出俩倍，因而，红黑树是相对的接近平衡的二叉树。

调整频率高，AVL比红黑树查找效率高？

算法思路

红黑树保证树的高度平衡通过旋转(rotation)+变色(recolor)。

[红黑树是怎么实现的](https://mp.weixin.qq.com/s?__biz=MzI4NDY5Mjc1Mg==&mid=2247490951&idx=2&sn=dee13fa373f2b7f6f138cdb3494de09a&chksm=ebf6c5f8dc814ceeee42c3986cf5585e931150c7059eab6d5bbc969d254541f1f2cdc99215a0&mpshare=1&scene=1&srcid=&sharer_sharetime=1584771050673&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=6ac3b46b266ce08c6c71cb5e290c31b9b613fda10ae680210c0ec525dcaff9cb1c4a15a73141ebbb60ded8b6f3c86e49c15f28af709f09e69dcd9d8916a607ec4d0098e81562534338673931e80189c5&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62080079&lang=zh_CN&exportkey=AXwPuw8OLjDc92U8pbLPY5U%3D&pass_ticket=bkroCR%2BRS8nQDbqetUMwAbPICZlPiN5gxNslIvRA2L78iC8ilF%2FBxf40U0ylFSRI)

#### 插入场景

插入步骤

1）将红黑树当作一颗二叉查找树，将节点插入树的底部；

2）插入节点默认着色为红色，如果是根节点，颜色着为黑色；

3）通过一系列的旋转或着色等操作，使之重新成为一颗红黑树

新插入的节点为红色，新插入节点的场景，例如：

1.插入的节点，父亲为黑色；

2.插入的节点，父亲为红色；

当新插入的节点的父亲为黑色，因为新插入的节点为红色，因此不会违反任何特性

当新插入的节点的父亲为红色时！因为新插入的节点为红色，违反不能出现两个连续的红节点，因此需要进行调整！

这种场景有3种调整情况，为了便于下面分析，假设将新插入节点用z代替，z的父节点用a代替，a的父节点用c代替，z的叔节点用y代替，如下

情况1：z的叔节点y是红色的

TreeMap

* 对key进行自然排序，
* 添加的对象需要实现Comparable接口（jdk自带的类型已实现）
* TreeMap内部基于红黑树排序

### 3.3.4 B树

即Balanced Tree，是一种平衡的多路查找树，适用于读写相对大的数据块的存储系统，例如磁盘。B树减少定位记录时所经历的中间过程，从而加快存取速度。

定义：

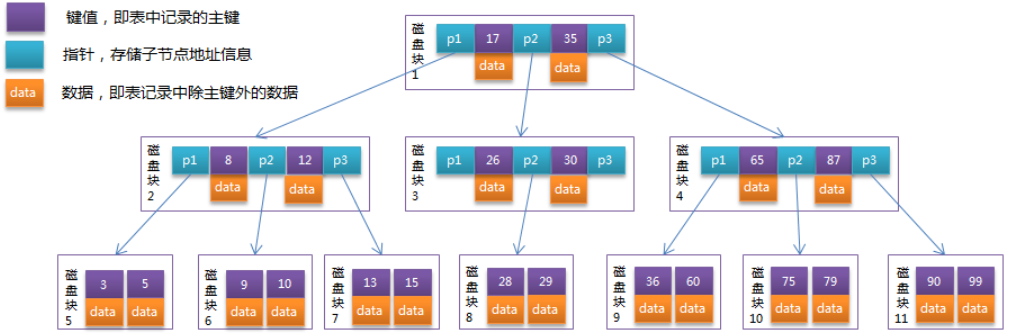
每个节点包含信息：（n,p0,k1,p1,k2…,kn,pn）。

* n为关键字个数。
* ki (i=1,…n)为关键字，ki<ki+1。 关键字数量： (m/2)-1 ≤ n ≤ m-1 (非根节点)
* pi (i=0,…n)为指向子树节点的指针。p(i)指向的子树的节点关键字均小于ki+1，大于k(i)。

子树数量：(m/2) ≤ n ≤ m（除根节点的非叶子节点）

* 所有叶子节点都在同一层，且不包含其它关键字信息（指针为空）

如图一个3阶的B-Tree：



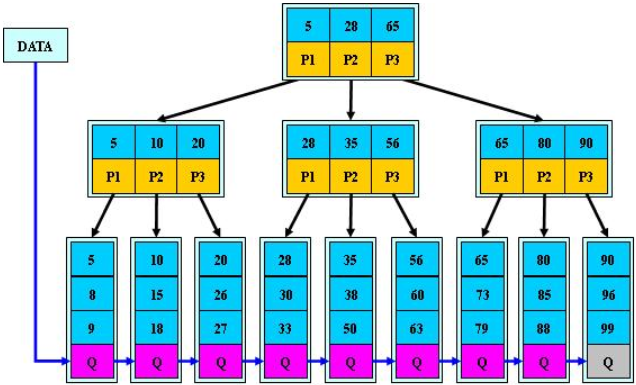
每个节点占用一个盘块的磁盘空间，一个节点上有两个升序排序的关键字和三个指向子树根节点的指针，指针存储的是子节点所在磁盘块的地址。两个关键词划分成的三个范围域对应三个指针指向的子树的数据的范围域。

模拟查找关键字29的过程：需要3次磁盘I/O操作，和3次内存查找操作。节点中的关键字是一个有序表结构，可以利用二分法查找提高效率。而3次磁盘I/O是影响查找效率的决定因素。B树相对于AVL缩减了节点个数，使每次磁盘I/O取到内存的数据都发挥了作用，从而提高了查询效率。

### 3.3.5 B+树

B+Tree是B树基础上为了更充分的利用结点空间，让遍历查询速度更稳定而扩展的结构，它规定只在叶子节点存数据，非叶子结点只存索引，使内部节点的度数尽可能的大。且叶子结点用一个链表连接起来。特性：

* B+树的非叶子节点只存数据索引，使得B+树每个节点所能保存的关键字大大增加；
* B+叶子节点保存了父节点的所有关键字和关键字记录的指针，每个叶子节点的关键字从小到大链接；
* B+树的根节点关键字数量和其子节点个数相等;
* B+树的数据必须要到叶子节点才能获取到，每次查找的次数都相同所以查询速度更稳定；



### 3.3.6 小结

#### 使用场景

B树的应用

B树大量应用在数据库和文件系统当中。比如MongoDB，innodb。

设计思想是，将相关数据尽量集中在一起，以便一次读取多个数据，减少硬盘操作次数。B树算法减少定位记录时所经历的中间过程，从而加快存取速度。

假定一个节点可以容纳100个值，那么3层的B树可以容纳100万个数据，如果换成二叉查找树，则需要20层！假定操作系统一次读取一个节点，并且根节点保留在内存中，那么B树在100万个数据中查找目标值，只需要读取两次硬盘。

如mongoDB数据库使用，单次查询平均快于Mysql（但侧面来看Mysql至少平均查询耗时差不多）

B+树的应用

mysql使用B+树作为索引：

B+树相对B树的优点：

B+树的所有Data域在叶子节点，一般来说都会进行一个优化，就是将所有的叶子节点用指针串联起来，遍历叶子节点就能获取全部数据，这样就能进行区间访问了。

IO一次读数据是从磁盘上读的，磁盘容量是固定的，取数据量大小是固定的，非叶子节点不存储数据，节点小，磁盘IO次数就少。

红黑树

红黑树往往出现由于树的深度过大而造成磁盘IO读写过于频繁，进而导致效率低下的情况在数据较小，可以完全放到内存中时，红黑树的时间复杂度比B树低。

如linux中进程的调度用的是红黑树。

反之，数据量较大，外存中占主要部分时，B树因其读磁盘次数少，而具有更快的速度。

为什么说B+树比B树更适合实际应用中操作系统的文件索引和数据库索引？

* B+Tree的磁盘读写代价更低 ，B+Tree的内部结点没有具体信息，一个盘块所能容纳的更多关键字。一次性读入内存中的要查找的关键字也就越多。IO读写次数也就降低了。

举个例子，假设磁盘中的一个盘块容纳16bytes，而一个关键字2bytes，一个关键字具体信息指针2bytes。一棵9阶B-tree(一个结点最多8个关键字)的内部结点需要2个盘快。而B+ 树内部结点只需要1个盘快。当需要把内部结点读入内存中的时候，B 树就比B+ 树多一次盘块查找时间(在磁盘中就是盘片旋转的时间)。

* B+tree的查询效率更加稳定，由于非叶子结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。
* B树在提高了磁盘IO性能的同时并没有解决元素遍历的效率低下的问题。B+树只要遍历叶子节点就可以实现整棵树的遍历。而且在数据库中基于范围的查询是非常频繁的，而B树不支持这样的操作。

## 3.4 其他树

### 3.4.1 线索二叉树

#### 二叉树的线索化

#### 二叉线索链表上的运算

### 3.4.2 树和森林

#### 树的存储结构

#### 树、森林与二叉树的转换

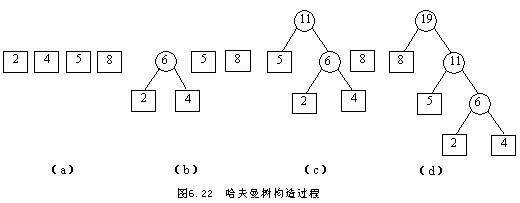
#### 树和森林的遍历

### 3.4.3 哈夫曼树及其应用

#### 3.5.1 最优二叉树(哈夫曼树)

在权为w1，w2…..的n个**叶子**所构成的二叉树带权路径长度最小的二叉树

1. n个节点，每个都是孤立的节点。经过n-1次合并产生n-1个节点共2n-1个



遍历：

#### 3.5.2 哈夫曼编码

# 第八章 图

G=(V,E),v是Vertex，E是Edge。

<Vj,Vi>。有向边也称弧（Arc），始点是弧尾（Tail），终点弧头（Head）；

**有向图（Digraph）顶点集和边集的表示：**

V（G1）={v1,v2,v3}

E(G1)={<v1,v2>,<v2,v1>,<v2,v3>}; //无向图不用写重复的

无向图（unDigraph）的度（Degree）：

**D(v)关联该顶点边的数目**

有向图是D(v)=ID(v)+OD(v);

**路径**：只要能从一个顶点到另个。如果除了开始和结尾有重复，则称是**简单路径，**若开始结尾相同就称**简单回路或简单环**

**无向图：**

Vj到Vi有路径称Vj和Vi连通，若V（G）任意连通即有路径称G为**连通图**（Connecte Graph）

极大连通子图是**连通分量**（Connected Component）。极大指走过节点多。即全部走未走过的节点

**有向图：**

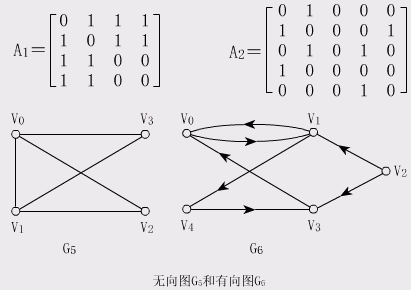
有根图：存在顶点v能到所有顶点，称图为有根图，v称为图的**根**

连通图有向时是**强连通图**

储存结构

1. **邻接矩阵表示法**

 　设G=(V，E)是具有n个顶点的图，则G的邻接矩阵是具有如下性质的n阶方阵：

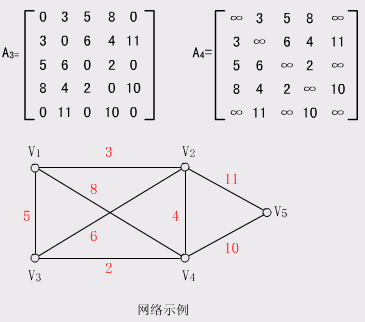


**总结：**

1、有几个顶点就有几行几列。做深度遍历时同邻接表在顶点(Vertex)的跳动，它是在不同行之间的跳动。

3．网络的邻接矩阵

若G是网络，则邻接矩阵可定义为：  
        
    
  
  其中：  
      wij表示边上的权值；  
      ∞表示一个计算机允许的、大于所有边上权值的数。  
【例】下面带权图的两种邻接矩阵分别为A3和A4。

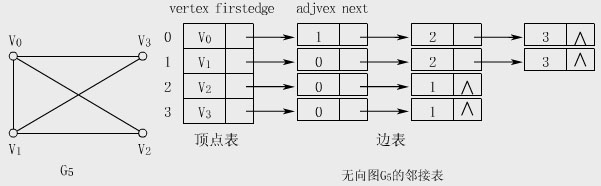


1. **邻接表表示**

把所有邻接于vi的顶点链接称链，称vi的**邻接表**。

无向图的邻接表为**边表**。

有向图是**出边表**。有向图只链接从某顶点的出边，逆邻接表是**入边表**



**邻接矩阵（Adjacency Matrix）的逻辑结构**

#define MaxVertexNum 100;

typedef char VertexType; //顶点类型

typedef int EdgeType; //权值类型

typedef struct{

VertexType vexs[MaxVertexNum];

EdeType edges[MaxVertexNum][ MaxVertexNum];

int n,e;

}

**邻接表的逻辑结构：**

typedef struct node{

int adijvex;

struct node \* next;

}EdgeNode; //边表节点

typedef struct vnode{

VertexType vertex;

EdgeNode \* firstedge;

}VertexNode; //顶点节点

typedef VertexNode AdjList[MaxVerexNum];

typedef struct{

AdjList adjlist;

int n,e; //n当前顶点数，e 边数

}ALGraph;

**图的遍历**

Typedef enum{TRUE,FALSE} Boolean;

Boolean visited[MaxVertexNum];

void DFSTtraverse(ALGraph \* G)

{

int i;

for(i=0;i<G->n;i++)

Visited[i]=FALSE;

for(i=0;i<G->n;i++)

If(!visited[i])

DFS(G,i);

}

void DFS(ALGraph \* G,int i)

{

EdgeNode \* p;

print(“vist vertex:%c”,G->adjlist[i].vertex);

visted[i]=TRUE;

p=G->adjlist[i].firstedge;

while(p)

{

if(!visted[p->adjvex])

DFS(G,p->adjvex);

p=p->next;

}

}

void DFSM(Mgraph \* G, int i)

{

int j;

printf(“visit vertex:%c”,G->vexs[i]);

visited[i]=TRUE;

for(j=0;j<G->n;j++)

if(G->edges[i][j]==1&&!visited[j])

DFSM(G,j)

}

**广度遍历(Breadth-First Traversal)**

void BFS(ALGraph \* G,int k)

{

int i;

CirQuenu Q;

EdgeNode \* p;

initQueue($=&Q);

printf(“visit vertex:%c”,G->adjlist[k].vertex);

visted[k]=TRUE;

EnQueue(&Q,k);

while(!Queue(&Q)){

i=DeQueue(&Q);

p=G->adjlist[i].firstedge;

while(p)

{

if(!visited[p->adjvex]){

printf(“visit vertex:%c”,G->adjlist[k].vertex);

visited[p->adjvex]=true;

EnQueue(&Q,p->adjvex);

}

p=p->next;

}

}

}

**生成树和最小生成树**

无回路的连通图，没确定根，也称**自由树**。

连通图的子图如果是包含G的所有顶点的树，称**生成树**(Spanning Tree)

实际生成树是有N个顶点和有n-1条边的极小连通子图。极小指边数最小。

深度遍历（DFS）：从v开始，搜索**第一个**邻接点w，从w开始重新DFS。若有未访问过的，**随便**选一个开始DFS。

深度优先生成树：

广度优先生成树：

**最小生成树**

Prim算法：

1. 从0开始，用虚线相连确定权最小用实线相连，依次到实线那头继续。。

Kruska算法：

### 多维数组和广义表

计算机内存是一维的。一般不对数组插入，采用顺序存储。

1. 行优先顺序：a11,a12…,a

LOC(aij)=LOC(a11)+[(i-1)\*n+j-1]\*d

1. 列优先:a11,a21..an1,a12,a22…an2….

**矩阵的压缩存储**

**对称矩阵（**对角线上有数据的**）**

N阶方阵 Aij=Aji;一般只存一半；压缩存储时

对角线上是aii.

**三角矩阵（**对角线上有数据的**）**

上三角矩阵：下三角（不包括主角线）均为常数；

**对角矩阵：**

**稀疏矩阵：**

1. 三元组表
2. 带行表的三元组表

**广义表（Lists）**

对线性和树的推广。

LS=（a1,a2,….an）; a1是表头，其余元素组成的表LS称表尾

A=(x,L)=(x,(a,b)) 或(x,L(a,b))

**空表 E=（）。不能运算。tail（（））=( );**

**tail( (( )) )=head( (( )) )=( )。(( ),( ))不对？**

1. 子表大写
2. 深度指单方向括号的层数。A是2
3. 线性表都是原子项。与树对应纯表。
4. head(LS) 原子和字表都行

广义表的图形表示：

**A( ) B(e) C(a,(b,c,d)) D(A(),B(e),C(a,(b,c,d)))**

**E((a,(a,b),((a,b),c)))**



纯表: 与树对应

再入表：允许节点再入

递归表：允许节点递归

广义表是线性表和树的推广

## 图的存储结构

## 图的遍历

## 图的生成树和最小生成树

### 图的生成树

### 最小生成树

## 最短路径

## 拓扑排序

菜单递归查找

Find all root menu

1. **public** **void** buildMenu() {
2. List<Menu> allMenus = **new** ArrayList<Menu>();//dao.getAllMenus();
3. List<Menu> parents = **new** ArrayList<Menu>();
4. //find all 1 level menu
5. **for** (Menu menu : allMenus) {
6. **if** (menu.getParentId() == **null**) {
7. parents.add(menu);
8. }
9. }
10. **for** (Menu menu : parents) {
11. menu.setChildren(getChild(menu.getMenuId(),allMenus));
12. }
13. }

getChild

1. **public** List<Menu> getChild(**long** menuId,List<Menu> menuList) {
2. List<Menu> childList = **new** ArrayList<Menu>();
3. **for** (Menu menu : menuList) {
4. **if** (menu.getParentId().equals(menuId)) {
5. childList.add(menu);
6. }
7. }
8. //递归终止条件
9. **if** (childList.size()==0) {
10. **return** **null**;
11. }
12. //开始递归
13. **for** (Menu menu : childList) {
14. menu.setChildren(menu.getChild(menu.getMenuId(),menuList));
15. }
16. **return** childList;
17. }

文件

逻辑结构和储存结构：

数据结构的存储结构是和相应的数据在内存中的物理地址之间的关系有关。而逻辑结构只是描述数据之间的关系（三大逻辑结构的一种）。

举例说，线性表（元素之间的逻辑关系是线性的）可以是顺序存储的方式，即所有元素相邻存放，在物理地址上是连续的（存储结构）；而对于链式存储的线性表，他的所有元素之间不一定是线性相连的，可能是第一个结点（元素）的地址为0x123,而第二个元素又出现在物理地址0x100上。也就是说逻辑结构是线性的但是存储结构不一定就是线性的了。

**文件的组织方式（存储结构）：**

在外存的组织结构：顺序组织，索引组织。。。

**顺序文件：**

逻辑结构和物理结构顺序一致

**索引文件：**

**索引顺序文件：**

ISAM（Indexed Sequential Access Method）：索引顺序存取

静态索引结构，

VSAM(Virtual Storage Access Method):虚拟储存存取方法

也是顺序索引结构，采用B+树的动态索引结构

**散列文件：**

**多关键字文件：**