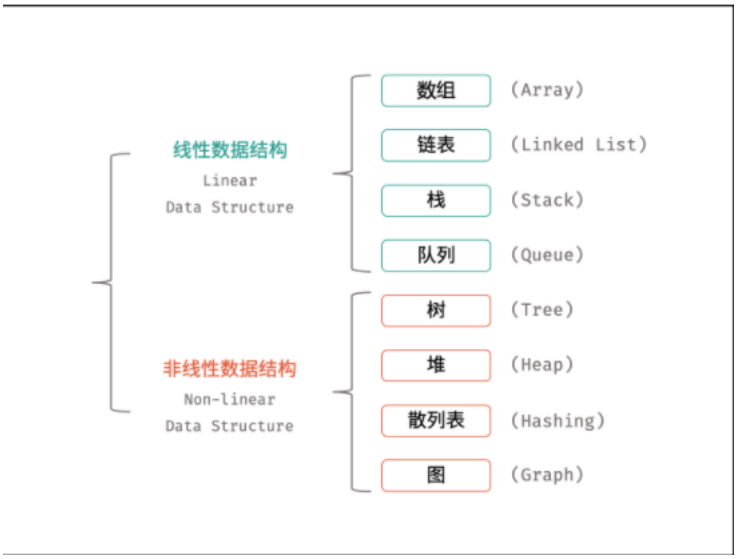
第2章 数据结构基础



2.1 数组

1. **基础**
   数组是将相同类型的元素存储于连续内存空间的数据结构，其长度不可变如下图所示，构建此数组需要在初始化时给定长度，并对数组每个索引元素赋值，代码如下:

```
// 初始化一个长度为 5 的数组 array
int array[5];
int array[] = {2, 3, 1, 0, 2};
// 初始化可变数组
vector<int> array;
// 向尾部添加元素
array.push_back(2);
array.push_back(3);
```

2. **二分查找的实现**

使用二分查找应该注意是否有序以及无重复元素

理解区间定义，处理好边界问题；

写二分法，区间的定义一般为两种， 左闭右闭即[left, right]，或者左闭右开即[left, right)

左闭右闭即[left, right]

while (left <= right) 要使用 <= ，因为left == right是有意义的，所以使用 <=

if (nums[middle] > target) right 要赋值为 middle - 1，因为当前这个nums[middle]一定不是target，那么接下来要查找的左区间结束下标位置就是 middle - 1

```
int search(vector<int>& nums, int target) {
    int left = 0;
    int right = nums.size() - 1; // 定义target在左闭右闭的区间里，[left, right]
    while (left <= right) { // 当left==right，区间[left, right]依然有效，所以用 <=
        int middle = left + ((right - left) / 2);// 防止溢出 等同于(left + right)/2
        if (nums[middle] > target) {
            right = middle - 1; // target 在左区间，所以[left, middle - 1]
        } else if (nums[middle] < target) {
            left = middle + 1; // target 在右区间，所以[middle + 1, right]
        } else { // nums[middle] == target
            return middle; // 数组中找到目标值，直接返回下标
        }
    }
    // 未找到目标值
    return -1;
}
```

如果定义 target 是在一个在左闭右开的区间里，也就是[left, right) ，那么二分法的边界处理方式则截然不同:

while (left < right)，这里使用 < ,因为left == right在区间[left, right)是没有意义的

if (nums[middle] > target) right 更新为 middle，因为当前nums[middle]不等于target，去左区间继续寻找，而寻找区间是左闭右开区间，所以right更新为middle，即

```
int search(vector<int>& nums, int target) {
    int left = 0;
    int right = nums.size(); // 定义target在左闭右开的区间里，即：[left, right)
    while (left < right) { // 因为left == right的时候，在[left, right)是无效的空间，所以使用 <
        int middle = left + ((right - left) >> 1);
        if (nums[middle] > target) {
            right = middle; // target 在左区间，在[left, middle)中
        } else if (nums[middle] < target) {
            left = middle + 1; // target 在右区间，在[middle + 1, right)中
        } else { // nums[middle] == target
            return middle; // 数组中找到目标值，直接返回下标
        }
    }
    // 未找到目标值
    return -1;
}
```

leetcode35 查找插入位置

难点在与循环之后插入位置处理，应该从target所在区间角度考虑理解返回值

版本1

```
int searchInsert(vector<int>& nums, int target) {
    int n = nums.size();
    int left = 0;
    int right = n - 1; // 定义target在左闭右闭的区间里，[left, right]
    while (left <= right) { // 当left==right，区间[left, right]依然有效
        int middle = left + ((right - left) / 2);// 防止溢出 等同于(left + right)/2
        if (nums[middle] > target) {
            right = middle - 1; // target 在左区间，所以[left, middle - 1]
        } else if (nums[middle] < target) {
            left = middle + 1; // target 在右区间，所以[middle + 1, right]
        } else { // nums[middle] == target
            return middle;
        }
    }
    // 分别处理如下四种情况
    // 目标值在数组所有元素之前  [0, -1]
    // 目标值等于数组中某一个元素  return middle;
    // 目标值插入数组中的位置 [left, right]，return  right + 1
    // 目标值在数组所有元素之后的情况 [left, right]，  return right + 1
    return right + 1;
}
```

若最后一次查找执行为

`if (nums[middle] > target) middle=right+1为插入位置 else if (nums[middle] < target) left|=middle+1=right+1 (while条件为<=)为插入位置 故 return rig`

版本2

```
int searchInsert(vector<int>& nums, int target) {
    int n = nums.size();
    int left = 0;
    int right = n; // 定义target在左闭右开的区间里，[left, right)  target
    while (left < right) { // 因为left == right的时候，在[left, right)是无效的空间
        int middle = left + ((right - left) >> 1);
        if (nums[middle] > target) {
            right = middle; // target 在左区间，在[left, middle)中
        } else if (nums[middle] < target) {
            left = middle + 1; // target 在右区间，在 [middle+1, right)中
        } else { // nums[middle] == target
            return middle; // 数组中找到目标值的情况，直接返回下标
        }
    }
    // 分别处理如下四种情况
    // 目标值在数组所有元素之前 [0,0)
    // 目标值等于数组中某一个元素 return middle
    // 目标值插入数组中的位置 [left, right) ，return right 即可
    // 目标值在数组所有元素之后的情况 [left, right)，return right 即可
    return right;
}
```

二分法查找边界问题

注意重复元素

那么这里我采用while (left <= right)的写法，区间定义为[left, right]，即左闭右闭的区间（如果这里有点看不懂了，强烈建议把704.二分查找 这篇文章先看了，704题目做了之后再做这道题目就好很多了）

确定好：计算出来的右边界是不包含target的右边界，左边界同理。

可以写出如下代码

```
// 二分查找，寻找target的右边界（不包括target）
// 如果rightBorder为没有被赋值（即target在数组范围的左边，例如数组[3,3]，target为2），为了处理
int getRightBorder(vector<int>& nums, int target) {
    int left = 0;
    int right = nums.size() - 1; // 定义target在左闭右闭的区间里，[left, right]
    int rightBorder = -2; // 记录一下rightBorder没有被赋值的情况
    while (left <= right) { // 当left==right，区间[left, right]依然有效
        int middle = left + ((right - left) / 2);// 防止溢出 等同于(left + right)/2
        if (nums[middle] > target) {
            right = middle - 1; // target 在左区间，所以[left, middle - 1]
        } else { // 当nums[middle] == target的时候，更新left，这样才能得到target的右边界
            left = middle + 1;
            rightBorder = left;
        }
    }
    return rightBorder;
}
```

寻找左边界

```
查找，寻找target的左边界leftBorder（不包括target）
leftBorder没有被赋值（即target在数组范围的右边，例如数组[3,3],target为4），为了处理情况一
tLeftBorder(vector<int>& nums, int target) {
t left = 0;
t right = nums.size() - 1; // 定义target在左闭右闭的区间里，[left, right]
t leftBorder = -2; // 记录一下leftBorder没有被赋值的情况
ile (left <= right) {
  int middle = left + ((right - left) / 2);
  if (nums[middle] >= target) { // 寻找左边界，就要在nums[middle] == target的时候更新right
      right = middle - 1;
      leftBorder = right;
  } else {
      left = middle + 1;
  }

turn leftBorder;
```

3. **数组元素的移除** *（双指针）*

leetcode209 最小长度子数组

给定一个含有 n 个正整数的数组和一个正整数 s ，找出该数组中满足其和 ≥ s 的长度最小的 连续 子数组，并返回其长度。如果不存在符合条件的子数组，返回 0。

示例：

输入：s = 7, nums = [2,3,1,2,4,3] 输出：2 解释：子数组 [4,3] 是该条件下的长度最小的子数组。

```
class Solution
public:
    int minSubArrayLen(int s, vector<int>& nums) {
        int result = INT32_MAX;
        int sum = 0; // 滑动窗口数值之和
        int i = 0; // 滑动窗口起始位置
        int subLength = 0; // 滑动窗口的长度
        for (int j = 0; j < nums.size(); j++) {
            sum += nums[j];
            // 注意这里使用while，每次更新 i（起始位置），并不断比较子序列是否符合条件
            while (sum >= s) {
                subLength = (j - i + 1); // 取子序列的长度
                result = result < subLength ? result : subLength;
                sum -= nums[i++]; // 这里体现出滑动窗口的精髓之处，不断变更i（子序列的起始位置）
            }
        }
        // 如果result没有被赋值的话，就返回0，说明没有符合条件的子序列
        return result == INT32_MAX ? 0 : result;
    }
};
```

2.2 链表

理论基础

链表以节点为单位，每个元素都是一个独立对象，在内存空间的存储是非连续的。链表的节点对象具有两个成员变量

```
struct ListNode {
    int val;        // 节点值
    ListNode *next; // 后继节点引用
    ListNode(int x) : val(x), next(NULL) {}
};
ListNode* creatList(vector<int> v){
    ListNode* pHead= new ListNode(-1),*r;
    r=pHead;
    for(auto data:v){
        ListNode* s=new ListNode(data);
        r->next=s;
        r=s;
    }
    return pHead->next;
}
```

移除链表元素

设置虚拟头节点处理，统一操作

```
class Solution {
public:
    ListNode* removeElements(ListNode* head, int val) {
        ListNode* dummyHead = new ListNode(0); // 设置一个虚拟头结点
        dummyHead->next = head; // 将虚拟头结点指向head，这样方面后面做删除操作
        ListNode* cur = dummyHead;
        while (cur->next != NULL) {
            if(cur->next->val == val) {
                ListNode* tmp = cur->next;
                cur->next = cur->next->next;
                delete tmp;
            } else {
                cur = cur->next;
            }
        }
        head = dummyHead->next;
        delete dummyHead;
        return head;
    }
};
```

翻转链表

两种思路：从头翻转 或者从尾部开始翻转

```
#方案1 双指针
ListNode* pre=NULL,*cur=pHead;
while(cur){
    ListNode* temp=cur->next;
    cur->next=pre;
    pre=cur;
    cur=temp;
}
return pre;
```

递归实现

```
class Solution {
public:
    ListNode* reverse(ListNode* pre,ListNode* cur){
        if(cur == NULL) return pre;
        ListNode* temp = cur->next;
        cur->next = pre;
        // 可以和双指针法的代码进行对比，如下递归的写法，其实就是做了这两步
        // pre = cur;
        // cur = temp;
        return reverse(cur,temp);
    }
    ListNode* reverseList(ListNode* head) {
        // 和双指针法初始化是一样的逻辑
        // ListNode* cur = head;
        // ListNode* pre = NULL;
        return reverse(NULL, head);
    }

};
```

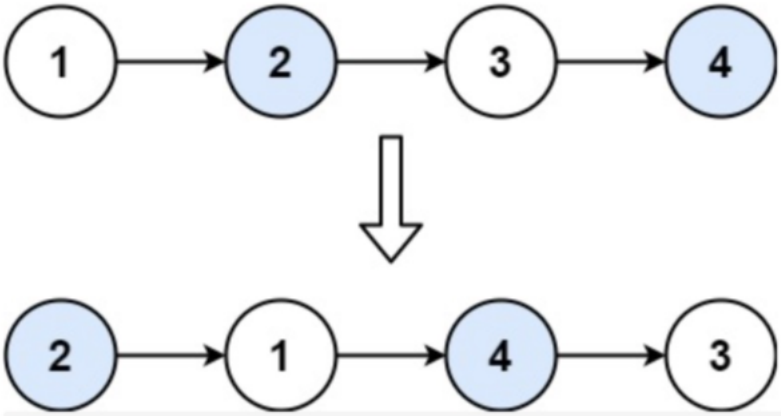
```
#方案2 从尾部开始翻转
ListNode* reverseList(ListNode* head){
    if(!head) return NULL:
    if(!head->next) return head;//递归出口的判断

    head=reverseList(head->next);
    head->next->next=head;
    head->next=NULL;
}
```

leetcode24 两两交换链表中节点

给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

示例 1：



输入： head = [1,2,3,4]
输出： [2,1,4,3]

思路

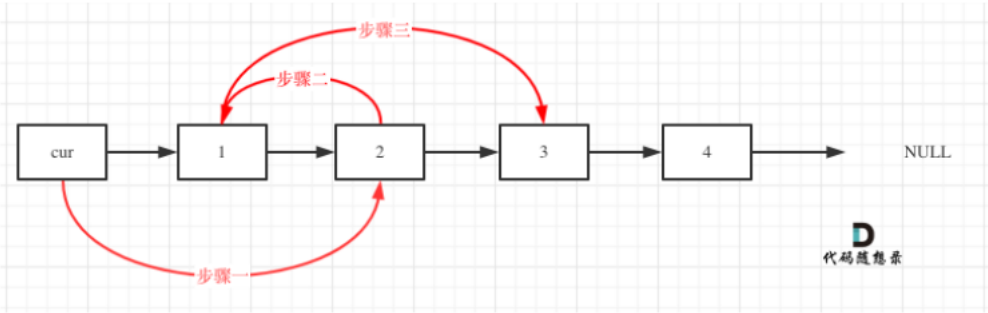
这道题目正常模拟就可以了。

建议使用虚拟头结点，这样会方便很多，要不然每次针对头结点（没有前一个指针指向头结点），还要单独处理。

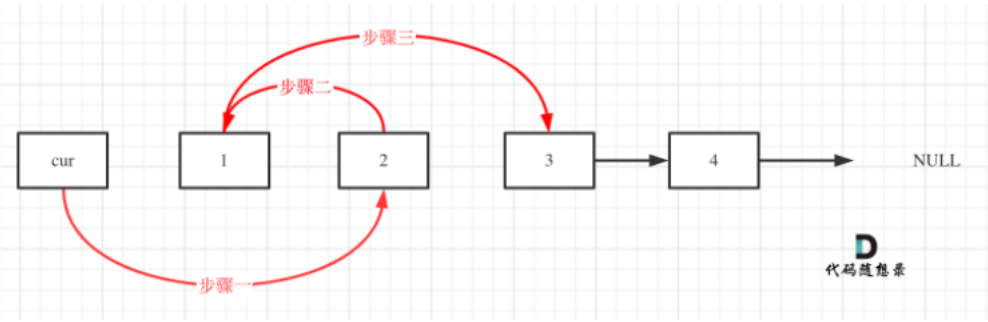
对虚拟头结点的操作，还不熟悉的话，可以看这篇链表：听说用虚拟头节点会方便很多？ 。

接下来就是交换相邻两个元素了，**此时一定要画图，不画图，操作多个指针很容易乱，而且要操作的先后顺序**

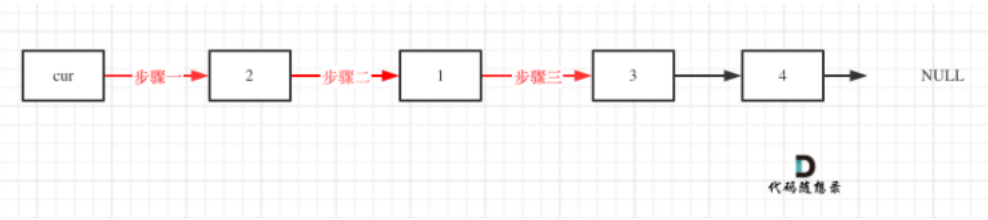
初始时，cur指向虚拟头结点，然后进行如下三步：



操作之后，链表如下：



看这个可能就更直观一些了：



```
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        ListNode* dummyHead = new ListNode(0); // 设置一个虚拟头结点
        dummyHead->next = head; // 将虚拟头结点指向head，这样方面后面做删除操作
        ListNode* cur = dummyHead;
        while(cur->next != nullptr && cur->next->next != nullptr) {
            ListNode* tmp = cur->next; // 记录临时节点
            ListNode* tmp1 = cur->next->next->next; // 记录临时节点

            cur->next = cur->next->next;    // 步骤一
            cur->next->next = tmp;          // 步骤二
            cur->next->next->next = tmp1;   // 步骤三

            cur = cur->next->next; // cur移动两位，准备下一轮交换
        }
        return dummyHead->next;
    }
};
```

链表相交节点问题 链表相交

两链表相加遍历思想

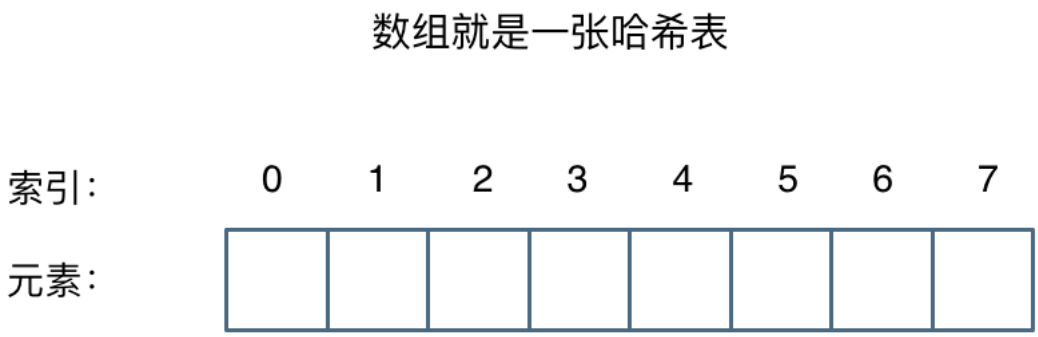
链表环入口问题 环形链表

*step2=2*step1 第二次相遇在入口处*

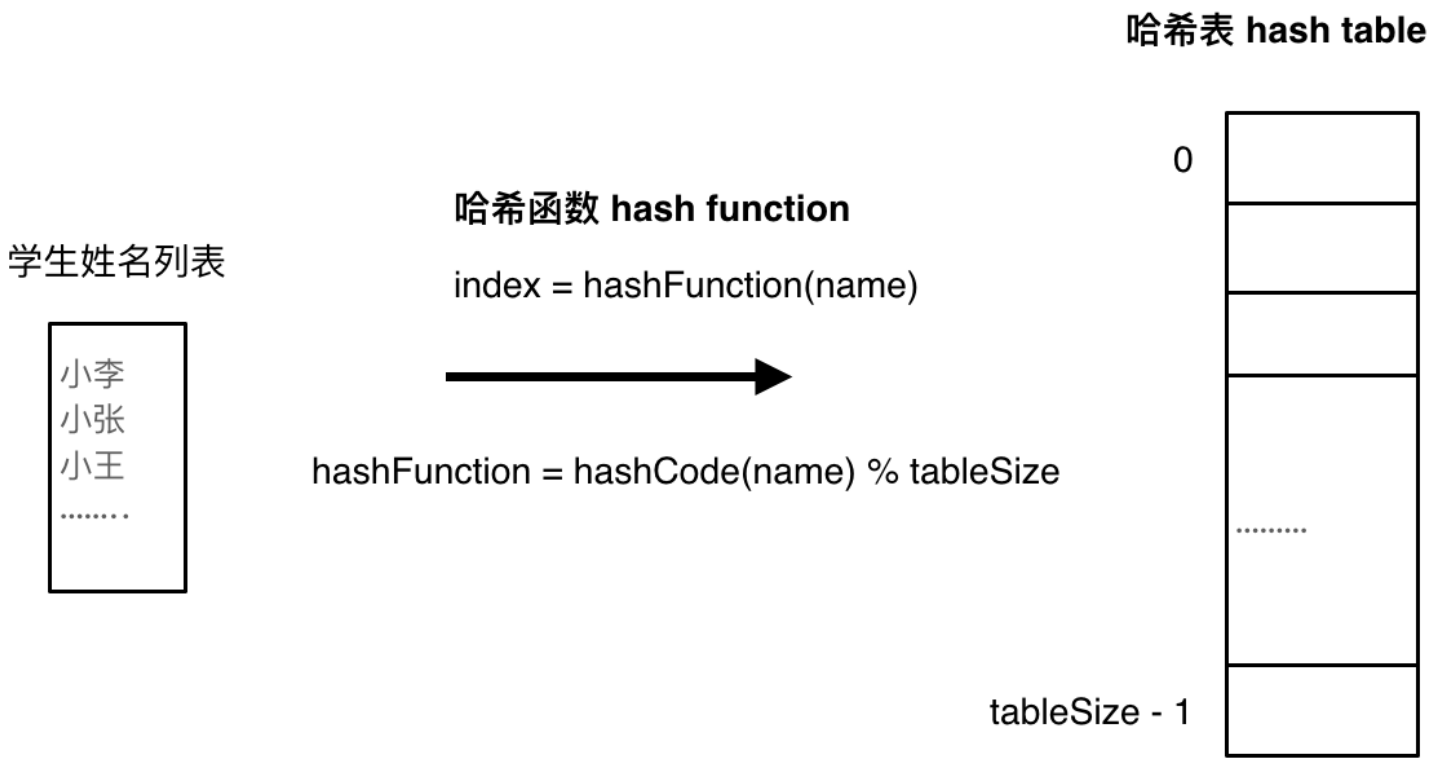
2.3 哈希表

理论基础

哈希表是根据关键码的值而直接进行访问的数据结构
一般哈希表都是用来快速判断一个元素是否出现集合里



例如要查询一个名字是否在这所学校里。
要枚举的话时间复杂度是$O(n)$，但如果使用哈希表的话， 只需要$O(1)$就可以做到。
我们只需要初始化把这所学校里学生的名字都存在哈希表里，在查询的时候通过索引直接就可以知道这位同学在不在这所学校里了。
*将学生姓名映射到哈希表上就涉及到了hash function ， 也就是**哈希函数***
哈希函数



一般哈希碰撞有两种解决方法， 拉链法和线性探测法

常见的三种哈希结构

- 数组
- set （集合）
- map(映射)

集合	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
std::set	红黑树	有序	否	否	$O(\log n)$	$O(\log n)$
std::multiset	红黑树	有序	是	否	$O(\log n)$	$O(\log n)$
std::unordered_set	哈希表	无序	否	否	$O(1)$	$O(1)$

std::unordered_set底层实现为哈希表，std::set 和std::multiset 的底层实现是红黑树，红黑树是一种平衡二叉搜索树，所以key值是有序的，但key不可以修改，改动key值会导致整棵树的错乱，所以只能删除和增加

映射	底层实现	是否有序	数值是否可以重复	能否更改数值	查询效率	增删效率
std::map	红黑树	key有序	key不可重复	key不可修改	$O(\log n)$	$O(\log n)$
std::multimap	红黑树	key有序	key可重复	key不可修改	$O(\log n)$	$O(\log n)$
std::unordered_map	哈希表	key无序	key不可重复	key不可修改	$O(1)$	$O(1)$

std::unordered_map 底层实现为哈希表，std::map 和std::multimap 的底层实现是红黑树。同理，std::map 和std::multimap 的key也是有序的（这个问题也经常作为面试题，考察对语言容器底层的理解）

1. 当我们要使用集合来解决哈希问题的时候，优先使用unordered_set，因为它的查询和增删效率是最优的，如果需要集合是有序的，那么就用set，如果要求不仅有序还要有重复数据的话，那么就用multiset

1. 那么再来看一下map ，在map 是一个key value 的数据结构，map中，对key是有限制，对value没有限制的，因为key的存储方式使用红黑树实现的。**

*虽然std::set、std::multiset 的底层实现是红黑树，不是哈希表，但是std::set、std::multiset 依然使用哈希函数来做映射，只不过底层的符号表使用了红黑树来存储数据，所以使用这些数据结构来解决映射问题的方法，我们依然称之为哈希法。 map也是一样的道理

这里在说一下，一些C++的经典书籍上 例如STL源码剖析，说到了hash_set hash_map，这个与unordered_set，unordered_map又有什么关系呢？*

实际上功能都是一样一样的， 但是unordered_set在C++11的时候被引入标准库了，而hash_set并没有，所以建议还是使用unordered_set比较好，这就好比一个是官方认证的，hash_set，hash_map 是C++11标准之前民间高手自发造的轮子*

总结：当我们遇到了要快速判断一个元素是否出现集合里的时候，就要考虑哈希法

但是哈希法也是牺牲了空间换取了时间，因为我们要使用额外的数组，set或者是map来存放数据，才能实现快速的查找

常见案例

leetcode242 有效的字母异位词 **map的简单使用**

```
/*
给定两个字符串 s 和 t ，编写一个函数来判断 t 是否是 s 的字母异位词。
示例 1: 输入: s = "anagram", t = "nagaram" 输出: true
示例 2: 输入: s = "rat", t = "car" 输出: false
说明: 你可以假设字符串只包含小写字母
*/
bool isAnagram(string s, string t) {
    unordered_map<char,int> map;
    for(auto c:s) ++map[c];
    //for(auto c:map) cout<<c.first<<" "<<c.second<<endl;
    for(auto c:t){
        if(map[c]>0) --map[c];
        else return false;
    }
    for(auto it:map){
        if(it.second!=0) return false;
    }
    return true;
}
```

也可以用数组实现，直接使用set，map不仅占用空间比数组大，
而且速度要比数组慢，把数值映射到key上都要做hash计算的

leetcode349 两个数组的交集 **unordeered_set以及set迭代器的使用**

```
/*
给定两个数组 nums1 和 nums2 ，返回 它们的交集
输出结果中的每个元素一定是 唯一 的。我们可以 不考虑输出结果的顺序
输入：nums1 = [1,2,2,1], nums2 = [2,2]
输出：[2]
*/
class Solution {
public:
    vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
        unordered_set<int> result_set; // 存放结果
        unordered_set<int> nums_set(nums1.begin(), nums1.end());
        for (int num : nums2) {
            // 发现nums2的元素 在nums_set里又出现过
            if (nums_set.find(num) != nums_set.end()) {
                result_set.insert(num);
            }
        }
        return vector<int>(result_set.begin(), result_set.end());
    }
};
```

leetcode1 两个数组之和

给定一个整数数组 nums 和一个目标值 target，请你在该数组中找出和为目标值的那 两个 整数，并返回他们的数组下标
你可以假设每种输入只会对应一个答案。但是，数组中同一个元素不能使用两遍
示例:
给定 nums = [2, 7, 11, 15], target = 9
因为 nums[0] + nums[1] = 2 + 7 = 9
所以返回 [0, 1]

思路

本题呢，则要使用map，那么来看一下使用数组和set来做哈希法的局限。
数组的大小是受限制的，而且如果元素很少，而哈希值太大会造成内存空间的浪费。
set是一个集合，里面放的元素只能是一个key，而两数之和这道题目，不仅要判断y是否存在而且还要记录y的下标位置，
因为要返回x 和 y的下标。所以set 也不能用。
此时就要选择另一种数据结构：map ，map是一种key value的存储结构，
可以用key保存数值，用value在保存数值所在的下标。

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int,int> map_num ;
        vector<int> v;
        for(int i=0;i<nums.size();++i){
            if(map_num.find(target-nums[i])!=map_num.end()){
                v.push_back(map_num[target-nums[i]]);
                v.push_back(i);
            }
            map_num.emplace(nums[i],i);
        }

        return v;
    }
};
```

leetcode383 赎金信

给定一个赎金信 (ransom) 字符串和一个杂志(magazine)字符串，判断第一个字符串 ransom 能不能由第二个字符串 magazines 里面的字符构成。如果可以构成，返回 true ；否则返回 false。
(题目说明：为了不暴露赎金信字迹，要从杂志上搜索各个需要的字母，组成单词来表达意思。杂志字符串中的每个字符只能在赎金信字符串中使用一次。)
注意:
你可以假设两个字符串均只含有小写字母。
canConstruct("a", "b") -> false
canConstruct("aa", "ab") -> false
canConstruct("aa", "aab") -> true

思路

在本题的情况下，使用map的空间消耗要比数组大一些的，因为map要维护红黑树或者哈希表，而且还要做哈希函数，是费时的！数据量大的话就能体现出来差别了。 所以数组更加简单直接有效

其他题目
leetcode15 三数之和 **双指针**
leetcode18 四数之和 **双指针**

2.4 字符串

理论基础

- 双指针法翻转字符串
  leetcode344 反转字符串
- 局部反转+整体反转
  剑指Offer58-II.左旋转字符
- KMP算法 实现strStr()
  leetcode28 实现strStr()
  实现 strStr() 函数。
  给定一个 haystack 字符串和一个 needle 字符串，在 haystack 字符串中找出 needle 字符串出现的第一个位置 (从0开始)。如果不存在，则返回 -1。
  示例 1: 输入: haystack = "hello", needle = "ll" 输出: 2
  示例 2: 输入: haystack = "aaaaa", needle = "bba" 输出: -1

```
class Solution {
public:
    string reverseLeftWords(string s, int n) {
        reverse(s.begin(), s.begin() + n);
        reverse(s.begin() + n, s.end());
        reverse(s.begin(), s.end());
        return s;
    }
};
```

字符串与数值的转化

熟悉to_string(),stoi(),stod()

```
string s="3.1415";
int a= 12345;
doulble d= stod(s);
string str=to_string(a);
```

to_string(val)	一组重载函数，返回数值 val 的 string 表示。val 可以是任何算术类型（参见 2.1.1 节，第 30 页）。对每个浮点类型和 int 或更大的整型，都有相应版本的 to_string。与往常一样，小整型会被提升（参见 4.11.1 节，第 142 页）
stoi(s, p, b) stol(s, p, b) stoul(s, p, b) stoll(s, p, b) stoull(s, p, b)	返回 s 的起始子串(表示整数内容)的数值，返回值类型分别是 int、long、unsigned long、long long、unsigned long long。b 表示转换所用的基数，默认值为 10。p 是 size_t 指针，用来保存 s 中第一个非数值字符的下标，p 默认为 0，即，函数不保存下标
stof(s, p) stod(s, p) stold(s, p)	返回 s 的起始子串（表示浮点数内容）的数值，返回值类型分别是 float、double 或 long double。参数 p 的作用与整数转换函数中一样

参考 c++primer 9.5.5

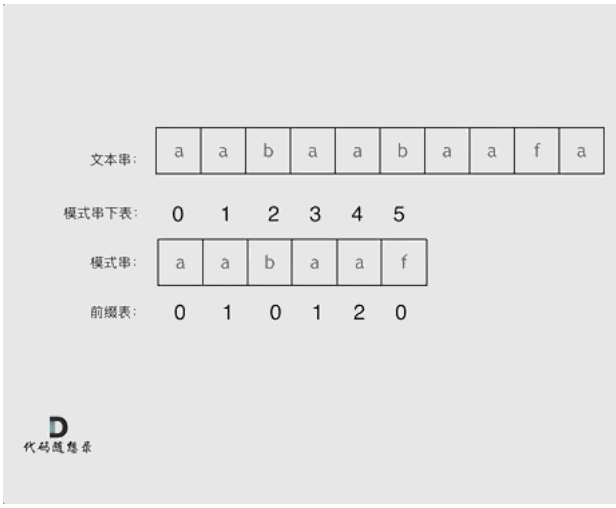
KMP算法

KMP的主要思想是当出现字符串不匹配时，可以知道一部分之前已经匹配的文本内容，可以利用这些信息避免从头再去做匹配了

1. 前缀表

作用:前缀表是用来回退的，它记录了模式串与主串(文本串)不匹配的时候，模式串应该从哪里开始重新匹配。
记录下标i之前（包括i）的字符串中，有多大长度的相同前缀后缀

前缀:是指不包含最后一个字符的所有以第一个字符开头的连续子串；
后缀:是指不包含第一个字符的所有以最后一个字符结尾的连续子串
最长相等前后缀



2. 前缀表与next数组

next数组就可以是前缀表，常用的两种实现:
1.把前缀表统一减一
2.右移一位，初始位置为-1之后作为next数组

构造next[]

构造next数组其实就是计算模式串s，前缀表的过程。 主要有如下三步(以前缀表统一减一为例):

1. 初始化
2. 处理前后缀不相同的情况
3. 处理前后缀相同的情况

- 初始化
  定义两个指针i和j，**j指向前缀起始位置，i指向后缀起始位置**
  然后还要对next数组进行初始化赋值，如下:

```
int j = -1;
next[0] = j;
```

  j 为什么要初始化为 -1呢，因为之前说过 前缀表要统一减一的操作仅仅是其中的一种实现，
  我们这里选择初始化为-1，下文我还会给出j不初始化为-1的实现代码。
  next[i] 表示 i（包括）之前最长相等的前后缀长度（其实就是j）
  所以初始化next[0] = j 。
- 处理前后缀不相同的情况
  因为j初始化为-1，那么i就从1开始，进行s[i] 与 s[j+1]的比较。
  所以遍历模式串s的循环下标i 要从 1开始，代码如下:

```
for (int i = 1; i < s.size(); i++) {
```

  如果 s[i] 与 s[j+1]不相同，也就是遇到 前后缀末尾不相同的情况，就要向前回退
  next[j]就是记录着j（包括j）之前的子串的相同前后缀的长度
  那么 s[i] 与 s[j+1] 不相同，就要找 j+1前一个元素在next数组里的值（就是next[j]）

```
while (j >= 0 && s[i] != s[j + 1]) { // 前后缀不相同了
    j = next[j]; // 向前回退
}
```

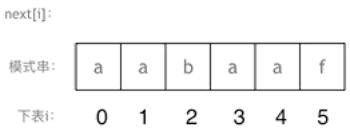
- 处理前后缀相同的情况
  如果 s[i] 与 s[j + 1] 相同，那么就同时向后移动i 和j 说明找到了相同的前后缀，同时还要将j （前缀的长度）赋给next[i], 因为next[i]要记录相同前后缀的长度

```
if (s[i] == s[j + 1]) { // 找到相同的前后缀
    j++;
}
next[i] = j;
```

整体代码如下

```
void getNext(int* next, const string& s){
    int j = -1;
    next[0] = j;
    for(int i = 1; i < s.size(); i++) { // 注意i从1开始
        while (j >= 0 && s[i] != s[j + 1]) { // 前后缀不相同了
            j = next[j]; // 向前回退
        }
        if (s[i] == s[j + 1]) { // 找到相同的前后缀
            j++;
        }
        next[i] = j; // 将j（前缀的长度）赋给next[i]
    }
}
```

next[]构造过程



3. 利用next数组来做匹配

```
int j = -1; // 因为next数组里记录的起始位置为-1
for (int i = 0; i < s.size(); i++) { // 注意i就从0开始
    while(j >= 0 && s[i] != t[j + 1]) { // 不匹配
        j = next[j]; // j 寻找之前匹配的位置
    }
    if (s[i] == t[j + 1]) { // 匹配，j和i同时向后移动
        j++; // i的增加在for循环里
    }
    if (j == (t.size() - 1) ) { // 文本串s里出现了模式串t
        return (i - t.size() + 1);
    }
}
```

前缀表统一减一 C++代码实现

```
class Solution {
public:
    void getNext(int* next, const string& s) {
        int j = -1;
        next[0] = j;
        for(int i = 1; i < s.size(); i++) { // 注意i从1开始
            while (j >= 0 && s[i] != s[j + 1]) { // 前后缀不相同了
                j = next[j]; // 向前回退
            }
            if (s[i] == s[j + 1]) { // 找到相同的前后缀
                j++;
            }
            next[i] = j; // 将j（前缀的长度）赋给next[i]
        }
    }
    int strStr(string haystack, string needle) {
        if (needle.size() == 0) {
            return 0;
        }
        int next[needle.size()];
        getNext(next, needle);
        int j = -1; // // 因为next数组里记录的起始位置为-1
        for (int i = 0; i < haystack.size(); i++) { // 注意i就从0开始
            while(j >= 0 && haystack[i] != needle[j + 1]) { // 不匹配
                j = next[j]; // j 寻找之前匹配的位置
            }
            if (haystack[i] == needle[j + 1]) { // 匹配，j和i同时向后移动
                j++; // i的增加在for循环里
            }
            if (j == (needle.size() - 1) ) { // 文本串s里出现了模式串t
                return (i - needle.size() + 1);
            }
        }
        return -1;
    }
};
```

前缀表（不减一不右移）C++实现

```
class Solution {
public:
    void getNext(int* next, const string& s) {
        int j = 0;
        next[0] = 0;
        for(int i = 1; i < s.size(); i++) {
            while (j > 0 && s[i] != s[j]) {
                j = next[j - 1];
            }
            if (s[i] == s[j]) {
                j++;
            }
            next[i] = j;
        }
    }
    int strStr(string haystack, string needle) {
        if (needle.size() == 0) {
            return 0;
        }
        int next[needle.size()];
        getNext(next, needle);
        int j = 0;
        for (int i = 0; i < haystack.size(); i++) {
            while(j > 0 && haystack[i] != needle[j]) {
                j = next[j - 1];
            }
            if (haystack[i] == needle[j]) {
                j++;
            }
            if (j == needle.size() ) {
                return (i - needle.size() + 1);
            }
        }
        return -1;
    }
};
```

leetcode459 重复的子字符串

给定一个非空的字符串，判断它是否可以由它的一个子串重复多次构成。给定的字符串只含有小写英文字母，并且长度不超过10000。
示例 1:
输入: "abab"
输出: True
解释: 可由子字符串 "ab" 重复两次构成。
思路
next[]为相等前后缀表统一减一;
next[len-1]!=-1 最长串存在相等前后缀
abcabcabc
len_s-(next[len_s-1]+1)为一个周期
若len为周期倍数，返回true

```
class Solution {
public:
    void getNext (int* next, const string& s){
        next[0] = -1;
        int j = -1;
        for(int i = 1;i < s.size(); i++){
            while(j >= 0 && s[i] != s[j+1]) {
                j = next[j];
            }
            if(s[i] == s[j+1]) {
                j++;
            }
            next[i] = j;
        }
    }
    bool repeatedSubstringPattern (string s) {
        if (s.size() == 0) {
            return false;
        }
        int next[s.size()];
        getNext(next, s);
        int len = s.size();
        if (next[len - 1] != -1 && len % (len - (next[len - 1] + 1)) == 0) {
            return true;
        }
        return false;
    }
};
```

2.5 栈与队列

容器以及容器适配器

熟悉 vector,deque的简单使用

表 9.1：顺序容器类型	
vector	可变大小数组。支持快速随机访问。在尾部之外的位置插入或删除元素可能很慢
deque	双端队列。支持快速随机访问。在头尾位置插入/删除速度很快
list	双向链表。只支持双向顺序访问。在 list 中任何位置进行插入/删除操作速度都很快
forward_list	单向链表。只支持单向顺序访问。在链表任何位置进行插入/删除操作速度都很快
array	固定大小数组。支持快速随机访问。不能添加或删除元素
string	与 vector 相似的容器，但专门用于保存字符。随机访问快。在尾部插入/删除速度快

顺序容器的相关操作

- 向容器中添加元素

表 9.5：向顺序容器添加元素的操作

这些操作会改变容器的大小；array 不支持这些操作。
forward_list 有自己专有版本的 insert 和 emplace；参见 9.3.4 节（第 312 页）。
forward_list 不支持 push_back 和 emplace_back。
vector 和 string 不支持 push_front 和 emplace_front。

c.push_back(t) c.emplace_back(*args*)	在 c 的尾部创建一个值为 t 或由 *args* 创建的元素。返回 void
c.push_front(t) c.emplace_front(*args*)	在 c 的头部创建一个值为 t 或由 *args* 创建的元素。返回 void
c.insert(p,t) c.emplace(p,*args*)	在迭代器 p 指向的元素之前创建一个值为 t 或由 *args* 创建的元素。返回指向新添加的元素的迭代器
c.insert(p,n,t)	在迭代器 p 指向的元素之前插入 n 个值为 t 的元素。返回指向新添加的第一个元素的迭代器；若 n 为 0，则返回 p
c.insert(p,b,e)	将迭代器 b 和 e 指定的范围内的元素插入到迭代器 p 指向的元素之前。b 和 e 不能指向 c 中的元素。返回指向新添加的第一个元素的迭代器；若范围为空，则返回 p
c.insert(p,il)	il 是一个花括号包围的元素值列表。将这些给定值插入到迭代器 p 指向的元素之前。返回指向新添加的第一个元素的迭代器；若列表为空，则返回 p

向一个 vector、string 或 deque 插入元素会使所有指向容器的迭代器、引用和指针失效。

- 删除容器中元素

表 9.7：顺序容器的删除操作

这些操作会改变容器的大小，所以不适用于 array。
forward_list 有特殊版本的 erase，参见 9.3.4 节（第 312 页）。
forward_list 不支持 pop_back；vector 和 string 不支持 pop_front。

c.pop_back()	删除 c 中尾元素。若 c 为空，则函数行为未定义。函数返回 void
c.pop_front()	删除 c 中首元素。若 c 为空，则函数行为未定义。函数返回 void
c.erase(p)	删除迭代器 p 所指定的元素，返回一个指向被删元素之后元素的迭代器，若 p 指向尾元素，则返回尾后（off-the-end）迭代器。若 p 是尾后迭代器，则函数行为未定义
c.erase(b,e)	删除迭代器 b 和 e 所指定范围内的元素。返回一个指向最后一个被删元素之后元素的迭代器，若 e 本身就是尾后迭代器，则函数也返回尾后迭代器
c.clear()	删除 c 中的所有元素。返回 void

删除 deque 中除首尾位置之外的任何元素都会使所有迭代器、引用和指针失效。指向 vector 或 string 中删除点之后位置的迭代器、引用和指针都会失效。

- 访问容器中元素

表 9.6：在顺序容器中访问元素的操作	
at 和下标操作只适用于 string、vector、deque 和 array。 back 不适用于 forward_list。	
c.back()	返回 c 中尾元素的引用。若 c 为空，函数行为未定义
c.front()	返回 c 中首元素的引用。若 c 为空，函数行为未定义
c[n]	返回 c 中下标为 n 的元素的引用，n 是一个无符号整数。若 n>=c.size()，则函数行为未定义
c.at(n)	返回下标为 n 的元素的引用。如果下标越界，则抛出一 out_of_range 异常

对一个空容器调用 front 和 back，就像使用一个越界的下标一样，是一种严重的程序设计错误。

访问成员函数返回的是引用

在容器中访问元素的成员函数（即，front、back、下标和 at）返回的都是引用。如果容器是一个 const 对象，则返回值是 const 的引用。如果容器不是 const 的，则返回值是普通引用，我们可以用来改变元素的值：

```
if (!c.empty()) {
    c.front() = 42;          // 将 42 赋予 c 中的第一个元素
    auto &v = c.back();      // 获得指向最后一个元素的引用
    v = 1024;                // 改变 c 中的元素
    auto v2 = c.back();      // v2 不是一个引用，它是 c.back()的一个拷贝
    v2 = 0;                  // 未改变 c 中的元素
}
```

与往常一样，如果我们使用 auto 变量来保存这些函数的返回值，并且希望使用此变量来改变元素的值，必须记得将变量定义为引用类型。

- 迭代器

```
//获取容器的迭代器
c.begin()
c.end()
```

容器适配器：

- **stack**
- **queue**
- **priority_queue**

表 9.17：所有容器适配器都支持的操作和类型	
size_type	一种类型，足以保存当前类型的最大对象的大小
value_type	元素类型
container_type	实现适配器的底层容器类型
A a;	创建一个名为 a 的空适配器
A a(c);	创建一个名为 a 的适配器，带有容器 c 的一个拷贝
关系运算符	每个适配器都支持所有关系运算符：==、!=、<、<=、>和>= 这些运算符返回底层容器的比较结果
a.empty()	若 a 包含任何元素，返回 false，否则返回 true
a.size()	返回 a 中的元素数目
swap(a,b)	交换 a 和 b 的内容，a 和 b 必须有相同类型，包括底层容器类型也必须相同
a.swap(b)	

vector

优点：
A、支持随机访问，访问效率高和方便，它像数组一样被访问，即支持[ ] 操作符和vector.at()。
B、节省空间，因为它是连续存储，在存储数据的区域都是没有被浪费的，但是要明确一点vector 大多情况下并不是满存的，在未存储的区域实际是浪费的。
缺点：
A、在内部进行插入、删除操作效率非常低。
B、只能在vector 的最后进行push 和pop ，不能在vector 的头进行push 和pop 。
C、 当动态添加的数据超过vector 默认分配的大小时要进行内存的重新分配、拷贝与释放，这个操作非常消耗能。
定义：vector<数据类型>名称
v.push_back():压入到最后一个
v.pop_back()：弹出最后一个
v.size():向量长度

stack

定义：stack<数据类型>名称
s.pop()：栈顶元素出栈
s.push(value)：元素入栈
s.top()：返回栈顶元素，元素不弹出

deque

deque双端队列
随机访问每个元素，所需要的时间为常量
弹出队首元素：q.pop_back()无返回值
弹出队尾元素：q.pop_front()无返回值
压入队尾：q.push_back(x)
压入队首：q.push_front(x)
取队首：q.front()
取队尾：q.back()

queue队列

定义：queue<数据类型> 队列名称
queue入队，如例：q.push(x)；将x 接到队列的末端。
queue出队，如例：q.pop()；弹出队列的第一个元素，注意，并不会返回被弹出元素的值。
访问queue队首元素，如例：q.front()，即最早被压入队列的元素。
访问queue队尾元素，如例：q.back()，即最后被压入队列的元素。
判断queue队列空，如例：q.empty()，当队列空时，返回true。
访问队列中的元素个数，如例：q.size()。

priority_queue优先队列

操作基本和队列相同

top 访问队头元素
empty 队列是否为空
size 返回队列内元素个数
push 插入元素到队尾 (并排序)
emplace 原地构造一个元素并插入队列
pop 弹出队头元素
swap 交换内容

定义:

```
priority_queue<Type, Container, Functional>
```

Type 就是数据类型，
Container 就是容器类型（Container必须是用数组实现的容器，比如vector,deque等等，但不能用 list。STL里面默认用的是vector），
Functional 就是比较的方式，当需要用自定义的数据类型时才需要传入这三个参数，使用基本数据类型时，只需要传入数据类型，默认是大顶堆

```
//升序队列
priority_queue <int,vector<int>,greater<int> > q;
//降序队列
priority_queue <int,vector<int>,less<int> >q;

// greater和less是std实现的两个仿函数
// 就是使一个类的使用看上去像一个函数。其实现就是类中实现一个operator(),
// 这个类就有了类似函数的行为，就是一个仿函数类了
```

实例

1. 基本类型例子

```
#include<iostream>
#include <queue>
using namespace std;
int main()
{
    //对于基础类型 默认是大顶堆
    priority_queue<int> a;
    //等同于 priority_queue<int, vector<int>, less<int> > a;

    priority_queue<int, vector<int>, greater<int> > c;  //这样就是小顶堆
    priority_queue<string> b;

    for (int i = 0; i < 5; i++)
    {
        a.push(i);
        c.push(i);
    }
    while (!a.empty())
    {
        cout << a.top() << ' ';
        a.pop();
    }
    cout << endl;

    while (!c.empty())
    {
        cout << c.top() << ' ';
        c.pop();
    }
    cout << endl;

    b.push("abc");
    b.push("abcd");
    b.push("cbd");
    while (!b.empty())
    {
        cout << b.top() << ' ';
        b.pop();
    }
    cout << endl;
    return 0;
}
```

2. pari的比较，先比较第一个元素，第一个相等比较第二个(默认构建大顶堆)

```cpp
#include <iostream>
#include <queue>
#include <vector>
using namespace std;
int main()
{
    priority_queue<pair<int, int> > a;
    pair<int, int> b(1, 2);
    pair<int, int> c(1, 3);
    pair<int, int> d(2, 5);
    a.push(d);
    a.push(c);
    a.push(b);
    while (!a.empty())
    {
        cout << a.top().first << ' ' << a.top().second << '\n';
        a.pop();
    }
}
/* 输出
2 5
1 3
1 2
*/
```

3. 对于自定义类型

运算符重载

重写仿函数

```cpp
#include <iostream>
#include <queue>
using namespace std;

//方法1
struct tmp1 //运算符重载<
{
    int x;
    tmp1(int a) {x = a;}
    bool operator<(const tmp1& a) const
    {
        return x < a.x; //大顶堆
    }
};

//方法2
struct tmp2 //重写仿函数
{
    // 默认是less函数
    // 返回true时，a的优先级低于b的优先级（a排在b的后面）
    bool operator() (tmp1 a, tmp1 b)
    {
        return a.x < b.x; //大顶堆
    }
};


//方法3 小顶堆
struct tmp3 //重写仿函数
{
    // 默认是less函数
    // 返回true时，a的优先级低于b的优先级（a排在b的后面）
    bool operator() (tmp1 a, tmp1 b)
    {
        return a.x > b.x; //小顶堆
    }
};

int main()
{
    tmp1 a(1);
    tmp1 b(2);
    tmp1 c(3);
    priority_queue<tmp1> d;
    d.push(b);
    d.push(c);
    d.push(a);
    while (!d.empty())
    {
        cout << d.top().x << '\n';
        d.pop();
    }
    cout << endl;

    priority_queue<tmp1, vector<tmp1>, tmp2> f;
    f.push(c);
    f.push(b);
    f.push(a);
    while (!f.empty())
    {
        cout << f.top().x << '\n';
        f.pop();
    }


}
```

堆的实现

尝试用泛型编程实现堆

```cpp
#include<iostream>
#include<vector>

using namespace std;
class Heap {
private:
    int N = 0;
    vector<int> heap;// 这个没有意义的是
public:

    Heap() {
        N = 0;
    }
    // 获取最大值
    int top() {
        return heap[0];
    }

    // 数据首先插入末尾 ,然后上浮
    void push(int k) {
        heap.push_back(k); // 直接放到后面，然后上浮
        this->N++;
        swim(heap.size() - 1);
        // print();
    }

    // 删除最大值，最后一位放到开头，然后在下沉
    void pop() {
        heap[0] = heap.back(); // 最后一位上位，然后在下沉
        heap.pop_back();
        this->N--;
        sink(0);
        // print();
    }

    // 下面是小根堆
    // 上浮，在每次插入数据的时候
    void swim(int pos) {
        while (pos > 0 && heap[(pos - 1) / 2] < heap[pos]) {  // 子节点大于父节点
            swap(heap[(pos - 1) / 2], heap[pos]); // 交换两个位置
            pos = (pos - 1) / 2; // 节点切换到上一层
        }
    }

    // 下沉，在每次删除数据的时候
    void sink(int pos) {
        int i;
        while (2 * pos + 2 < N) { //2pos是子节点
            i = 2 * pos + 1;
            if (i < N && heap[i] < heap[i + 1]) i++; // i指向子节点中最大的
            if (heap[pos] >= heap[i]) break; // 已经找到pos应该的位置了
            swap(heap[pos], heap[i]);
            pos = i;
        }
    }

    void print() {
        cout << "打印heap所有值" << endl;
        for (const auto& num : heap) {
            cout << num << " ";
        }cout << endl;
    }
};

void test() {
    Heap * heap=new Heap();
    int a[] = { 1,2,3,4 };
    int b = 3;
    heap->push(a[0]); heap->print();
    heap->push(a[1]); heap->print();
    heap->push(a[2]); heap->print();
    heap->push(a[3]); heap->print();


    heap->pop(); heap->print();
    heap->pop(); heap->print();
```

```
    heap->pop(); heap->print();
    cout << heap->top() << endl;
}

int main() {
    test();


    return 0;
}
```

典型题目

利用deque构造单调队列

leetcode239 滑动窗口最大值

给定一个数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。
返回滑动窗口中的最大值。

```
输入: nums = [1,3,-1,-3,5,3,6,7], k = 3
输出: [3,3,5,5,6,7]
解释:
滑动窗口的位置                最大值
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

队列没有必要维护窗口里的所有元素，只需要维护有可能成为窗口里最大值的元素就可以了，同时保证队里里的元素数值是由大到小的
维护元素单调递减的队列就叫做单调队列，即单调递减或单调递增的队列

```
class MyQueue { //单调队列（从大到小）
public:
    deque<int> que; // 使用deque来实现单调队列
    // 每次弹出的时候，比较当前要弹出的数值是否等于队列出口元素的数值，如果相等则弹出。
    // 同时pop之前判断队列当前是否为空。
    void pop(int value) {
        if (!que.empty() && value == que.front()) {
            que.pop_front();
        }
    }
    // 如果push的数值大于入口元素的数值，那么就将队列后端的数值弹出，直到push的数值小于等于队列入口元素的数值为止。
    // 这样就保持了队列里的数值是单调从大到小的了。
    void push(int value) {
        while (!que.empty() && value > que.back()) {
            que.pop_back();
        }
        que.push_back(value);

    }
    // 查询当前队列里的最大值 直接返回队列前端也就是front就可以了。
    int front() {
        return que.front();
    }
};
```

构造堆或者利用priority_queue

leetcode347 前 K 个高频元素
给你一个整数数组 nums 和一个整数 k ，请你返回其中出现频率前 k 高的元素。你可以按 任意顺序 返回答案

输入: nums = [1,1,1,2,2,3], k = 2
输出: [1,2]

```cpp
// 时间复杂度：O(nlogk)
// 空间复杂度：O(n)
class Solution {
public:
    // 小顶堆
    class comparision{
    public:
        bool operator()(const pair<int,int>& a,const pair<int,int>& b){// 默认重载< 操作符，数据大反而优先级小
            return a.second>b.second;
        }
    };
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int,int> map_nums;
        // 统计频率
        for(int c:nums){
            map_nums[c]++;
        }

        priority_queue<pair<int,int>,vector<pair<int,int>>,comparision> que;
        for(auto iter=map_nums.begin();iter!=map_nums.end();++iter){
            que.push(*iter);
            if(que.size()>k){
                que.pop();
            }
        }
        vector<int> result;
        while(que.size()>0){
            result.push_back(que.top().first);
            que.pop();
        }
        return result;
    }
};


// 时间复杂度：O(nlogk)
// 空间复杂度：O(n)
class Solution {
public:
    // 小顶堆
    class mycomparison {
    public:
        bool operator()(const pair<int, int>& lhs, const pair<int, int>& rhs) {
            return lhs.second > rhs.second;
        }
    };
    vector<int> topKFrequent(vector<int>& nums, int k) {
        // 要统计元素出现频率
        unordered_map<int, int> map; // map<nums[i],对应出现的次数>
        for (int i = 0; i < nums.size(); i++) {
            map[nums[i]]++;
        }

        // 对频率排序
        // 定义一个小顶堆，大小为k
        priority_queue<pair<int, int>, vector<pair<int, int>>, mycomparison> pri_que;

        // 用固定大小为k的小顶堆，扫面所有频率的数值
        for (unordered_map<int, int>::iterator it = map.begin(); it != map.end(); it++) {
            pri_que.push(*it);
            if (pri_que.size() > k) { // 如果堆的大小大于了K，则队列弹出，保证堆的大小一直为k
                pri_que.pop();
            }
        }

        // 找出前K个高频元素，因为小顶堆先弹出的是最小的，所以倒序来输出到数组
        vector<int> result(k);
        for (int i = k - 1; i >= 0; i--) {
            result[i] = pri_que.top().first;
            pri_que.pop();
        }
        return result;

    }
};
```

2.6 树

树的生成

定义

```
struct TreeNode{
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x):val(x),left(NULL),right(NULL){}
} ;
```

构建二叉树:生成TreeNode数组思想

```
// 根据数组构造二叉树
TreeNode* construct_binary_tree(const vector<int>& vec) {
    vector<TreeNode*> vecTree (vec.size(), NULL);
    TreeNode* root = NULL;
    // 把输入数值数组，先转化为二叉树节点数组
    for (int i = 0; i < vec.size(); i++) {
        TreeNode* node = NULL;
        if (vec[i] != -1) node = new TreeNode(vec[i]); // 用 -1 表示null
        vecTree[i] = node;
        if (i == 0) root = node;
    }
    // 遍历一遍，根据规则左右孩子赋值就可以了
    // 注意这里 结束规则是 i * 2 + 2 < vec.size()，避免空指针
    for (int i = 0; i * 2 + 2 < vec.size(); i++) {
        if (vecTree[i] != NULL) {
            // 线性存储转连式存储关键逻辑
            vecTree[i]->left = vecTree[i * 2 + 1];
            vecTree[i]->right = vecTree[i * 2 + 2];
        }
    }

    return root;
}
```

二叉树的遍历

- 递归实现
  - 确定确定递归函数的参数和返回值
  - 确定终止条件
  - 确定单层递归的逻辑

先序遍历

```
class Solution {
public:
    void traversal(TreeNode* cur, vector<int>& vec) {
        if (cur == NULL) return;
        vec.push_back(cur->val);    // 中
        traversal(cur->left, vec);  // 左
        traversal(cur->right, vec); // 右
    }
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> result;
        traversal(root, result);
        return result;
    }
};
```

迭代遍历