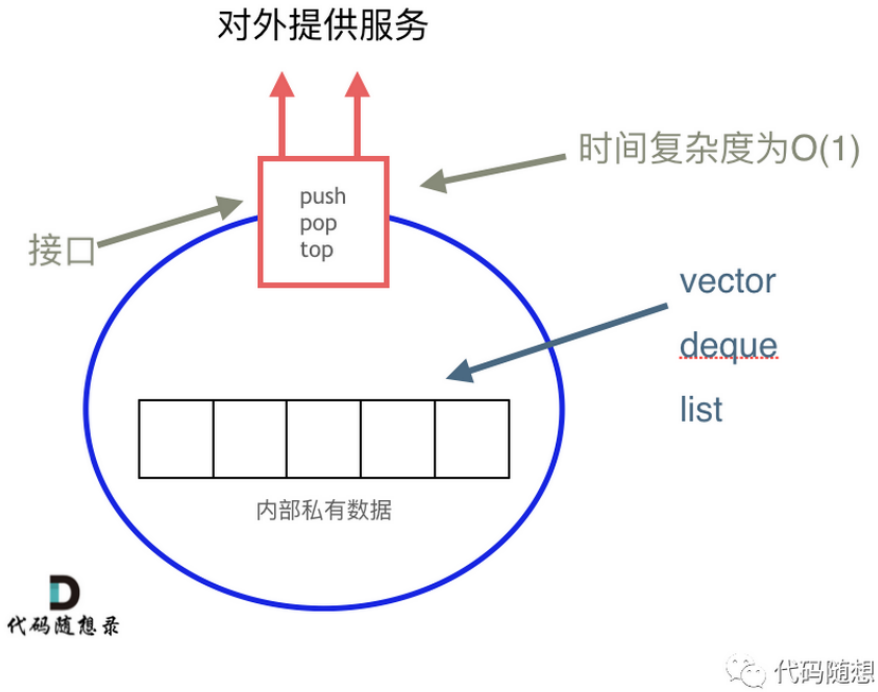
对于栈和队列除了先进后出和先进先出，还有一些内容

> 问问自己：
> 1:C++中stack是容器么?
> 2:我们使用的stack是属于哪个版本的STL?
> 3:我们使用的STL中stack是如何实现的?
> 4:stack提供迭代器来遍历stack空间么?

stack和队列是**STL(C++标准库)**两个数据结构,**我们一般使用的STL也是SGI STL**

栈和队列是以底层容器完成其所有的工作,对外提供统一的接口,底层容器是可插拔的就是说我们可以控制使用哪种容器来实现栈的功能,因此栈和队列不是容器,被归类为container adapter(容器适配器)



> deque和stack都一样不允许有遍历行为,不提供迭代器，**两者底层容器在默认情况下都是deque**

232. 用栈实现队列

使用栈实现队列的下列操作：

push(x) -- 将一个元素放入队列的尾部。
pop() -- 从队列首部移除元素。
peek() -- 返回队列首部的元素。
empty() -- 返回队列是否为空。

示例:

```
MyQueue queue = new MyQueue();
queue.push(1);
queue.push(2);
queue.peek();  // 返回 1
queue.pop();   // 返回 1
queue.empty(); // 返回 false
```

说明:

- 你只能使用标准的栈操作 -- 也就是只有 push to top, peek/pop from top, size, 和 is empty 操作是合法的。
- 你所使用的语言也许不支持栈。你可以使用 list 或者 deque（双端队列）来模拟一个栈，只要是标准的栈操作即可。
- 假设所有操作都是有效的 （例如，一个空的队列不会调用 pop 或者 peek 操作）。

> 在这个问题下，一个栈可能不能解决问题，需要两个栈:一个输入栈和一个输出栈
> 由于一个输入栈的元素加入到一个输出栈中，此时元素从先进后出转换为先进先出

pop()

> **假设队列连续进行了i个push操作，则此时输入栈内有i个元素，第i+1个操作为pop操作，此时由于输出栈为空,把输入栈内的所有元素都放入输出栈然后pop一个元素即可，如果输出栈不为空则无需把输入栈内所有元素放入输出栈直接将输出栈pop一个元素。**

empty()

> 只有当**输入栈和输出栈**都为空时，才确定队列为空

```
1  class MyQueue {
2  public:
3   stack<int> stIn;
4   stack<int> stOut;
5      /** Initialize your data structure here. */
6      MyQueue() {
7
8      }
```

```
9
10      /** Push element x to the back of queue. */
11      void push(int x) {
12  stIn.push(x);
13      }
14
15      /** Removes the element from in front of queue and returns that elem
ent. */
16      int pop() {
17  if(stOut.empty()){
18  //此时输出栈为空必须把输入栈中的所有元素放入输出栈中
19  while(!stIn.empty()){
20  stOut.push(stIn.top());
21  stIn.pop();
22  }
23  }
24  int res=stOut.top();
25  stOut.pop();
26  return res;
27      }
28
29      /** Get the front element. */
30      int peek() {
31  int res=this->pop();
32  stOut.push(res);
33  return res;
34      }
35
36      /** Returns whether the queue is empty. */
37      bool empty() {
38  return stIn.empty()&&stOut.empty();
39      }
40  };
41
42  /**
43   * Your MyQueue object will be instantiated and called as such:
44   * MyQueue* obj = new MyQueue();
45   * obj->push(x);
46   * int param_2 = obj->pop();
47   * int param_3 = obj->peek();
```

```
 * bool param_4 = obj->empty();
 */
```

拓展：

1、在工业级别代码开发中，最忌讳的就是实现一个类似的函数，直接把代码粘过来改改就完事了。

这样的项目代码会越来越乱，**「一定要懂得复用，功能相近的函数要抽象出来，不要大量的复制粘贴，很容易出问题！（踩过坑的人自然懂）」**

用队列实现栈还有点不一样

使用队列实现栈的下列操作：

- push(x) -- 元素 x 入栈
- pop() -- 移除栈顶元素
- top() -- 获取栈顶元素
- empty() -- 返回栈是否为空

注意：

- 你只能使用队列的基本操作-- 也就是 push to back, peek/pop from front, size, 和 is empty 这些操作是合法的。
- 你所使用的语言也许不支持队列。 你可以使用 list 或者 deque（双端队列）来模拟一个队列 , 只要是标准的队列操作即可。
- 你可以假设所有操作都是有效的（例如, 对一个空的栈不会调用 pop 或者 top 操作）。

> 对于用队列实现栈来说，可以用两个队列来模拟栈，只不过没有输入和输出的关系，而另一个队列完全时用来备份的。

pop()

> **对于pop元素时则把一个队列A中的元素都放入另一个队列B中只剩一个元素pop出,然后把队列B中的元素放回队列A中**

```
class MyStack {
public:
 queue<int> que1;
 queue<int> que2;
    /** Initialize your data structure here. */
    MyStack() {

    }
```

```
9
10      /** Push element x onto stack. */
11      void push(int x) {
12   que1.push(x);
13      }
14
15      /** Removes the element on top of the stack and returns that elemen
t. */
16      int pop() {
17   int ans=que1.back();
18   //把A队列中的元素放入B队列中
19   while(que1.size()>1){
20   que2.push(que1.front());
21   que1.pop();
22   }
23   que1.pop();
24   //把队列B中的元素放入队列A中
25   while(!que2.empty()){
26   que1.push(que2.front());
27   que2.pop();
28   }
29   return ans;
30      }
31
32      /** Get the top element. */
33      int top() {
34   que1.back();
35      }
36
37      /** Returns whether the stack is empty. */
38      bool empty() {
39   return que1.empty();
40      }
41  };
42
43  /**
44   * Your MyStack object will be instantiated and called as such:
45   * MyStack* obj = new MyStack();
46   * obj->push(x);
47   * int param_2 = obj->pop();
```

```
 * int param_3 = obj->top();
 * bool param_4 = obj->empty();
 */
```

系统中处处都是栈的应用(数据结构和算法往往隐藏在我们看不到的地方)

匹配问题都是栈的强项

1047. 删除字符串中的所有相邻重复项

给出由小写字母组成的字符串 S，重复项删除操作会选择两个相邻且相同的字母，并删除它们。

在 S 上反复执行重复项删除操作，直到无法继续删除。

在完成所有重复项删除操作后返回最终的字符串。答案保证唯一。

示例：
输入："abbaca"
输出："ca"
解释：
例如，在 "abbaca" 中，我们可以删除 "bb" 由于两字母相邻且相同，这是此时唯一可以执行删除操作的重复项。之后我们得到字符串 "aaca"，其中又只有 "aa" 可以执行重复项删除操作，所以最后的字符串为 "ca"。

提示：
1 <= S.length <= 20000
S 仅由小写英文字母组成。

> **使用一个栈,**
> 对于每个字符元素都放入栈中时候，
> 1:如果栈顶元素与将要加入的元素相同时就弹栈
> 2:如果栈顶元素与将要加入的元素不相同时就入栈
> 重复1和2的操作直到遍历字符串结束，此时将栈中的元素倒序输出即为最终的字符串.

调试小知识点：

> 如果系统输出的异常为**Segmentation fault(当然不是所有的Segmentation fault都是栈溢出导致的)**此时要考虑**是不是无限递归**了，**在企业项目开发中，尽量不要使用递归！！！**

```
class Solution {
public:
    string removeDuplicates(string S) {
 stack<char> sta;
 int i=S.size()-1;
```

```
while(i>=0){
char c=S[i];
if(!sta.empty()&&sta.top()==c){
sta.pop();
 }else{
 sta.push(c);
 }
 i--;
 }
 string ans="";
 while(!sta.empty()){
 ans+=sta.top();
 sta.pop();
 }
 //reverse(ans.begin(),ans.end());当i从0开始时需要reverse
 return ans;
    }
};
```

这不仅仅是一道好题，也展现出计算机的思考方式

150. 逆波兰表达式求值

根据 逆波兰表示法，求表达式的值。

有效的运算符包括 + , - , * , / 。每个运算对象可以是整数，也可以是另一个逆波兰表达式。

说明：

整数除法只保留整数部分。给定逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 0 的情况。

示例 1：
输入：["2", "1", "+", "3", " * "]
输出：9
解释：该算式转化为常见的中缀算术表达式为：((2 + 1) * 3) = 9

示例 2：
输入：["4", "13", "5", "/", "+"]
输出：6
解释：该算式转化为常见的中缀算术表达式为：(4 + (13 / 5)) = 6

逆波兰表达式：是一种后缀表达式，所谓后缀就是指算符写在后面。

平常使用的算式则是一种中缀表达式，如 (1 + 2) * (3 + 4) 。

该算式的逆波兰表达式写法为 ((1 2 +) (3 4 +) *) 。

逆波兰表达式主要有以下两个优点：

- 去掉括号后表达式无歧义，上式即便写成 1 2 + 3 4 + * 也可以依据次序计算出正确结果。
- 适合用栈操作运算：遇到数字则入栈；遇到算符则取出栈顶两个数字进行计算，并将结果压入栈中。

> 逆波兰式表达式值：
> 遍历字符串s
> 如果是数值字符串把其转换为数值加入到栈中，
> 如果为运算符字符串则进行弹栈（由于题目中都是双目表达式所以弹出两个元素）进行元素并把结果放入栈中

API:stoi(string)把字符串转换为int

```
class Solution {
public:
    int evalRPN(vector<string>& tokens) {
 stack<int> sta;
 for(int i=0;i<tokens.size();i++){
 string t=tokens[i];
 if(t=="+"||t=="-"||t=="*"||t=="/"){
 int ans=0;
 int nums1=sta.top();sta.pop();
  int nums2=sta.top();sta.pop();
  if(t=="+") ans=nums1+nums2;
  if(t=="-") ans=nums2-nums1;
  if(t=="*") ans=nums2*nums1;
  if(t=="/") ans=nums2/nums1;
  sta.push(ans);
  }else{
  sta.push(stoi(t));
  }
  }
  return sta.top();
    }
};
```

滑动窗口里求最大值引出一个重要数据结构

239. 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。

这是采用**单调队列**的题目，
每一次移动滑动窗口都是增加一个元素和减去一个元素，
每次移动之后，队列应该告诉我们里面的最大值是什么

这个队列应该长这个样子：

```
class MyQueue {
public:
    void pop(int value) {
    }
    void push(int value) {
    }
    int front() {

        return que.front();
    }
};
```

> 每次移进一个元素用push(),移除一个元素用pop()，返回当前最大值用front，
> 我们采用从队尾到队头为单调递增的单调队列实现功能
> 在该单调队列中我们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，
> 优先级队列就是**对队列中的元素进行排序**

此时需要重新设计队列

> 由于我们是已知每次移动时要移除的元素大小和移进的元素大小
> **所以我们需要设计特定的pop和push操作**
> pop(value)：如果窗口移除的元素value等于单调队列的出口元素，那么队列弹出元素，否则不用任何操作。
> push(value)：如果从队尾push的元素大于back()元素则把back()元素移除，直到push的元素小于等于队列入口元素的数值

```
1  class Solution {
2  public:
3   deque<int> que;
4   void pop(int value){
5   if(!que.empty()&&value==que.front()) que.pop_front();
6   }
7   void push(int value){
8   while(!que.empty()&&value>que.back()){
9   que.pop_back();
10   }
11   que.push(value);
12   }
13      vector<int> maxSlidingWindow(vector<int>& nums, int k) {
14   vector<int> result;
15   //先将k个元素放入队列中
16   for(int i=0;i<k;i++) que.push(nums[i]);
```

```
17    result.push_back(que.front());
18    for(int i=k;i<nums.size();i++){
19    //先移动后计算该滑动窗口里面的最大值
20    que.pop(nums[i-k]);
21    que.push_back(nums[i]);
22    result.push_back(que.front());
23    }
24    return result;
25        }
26  };
```

以上方法的**时间复杂度**为O(n)
因为每一个数组的元素都只能被队列pop和push一次

有一种队列是穿着队列外衣的堆

347.前 K 个高频元素

题目链接：https://leetcode-cn.com/problems/top-k-frequent-elements/

给定一个非空的整数数组，返回其中出现频率前 k 高的元素。

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

示例 2:
输入: nums = [1], k = 1
输出: [1]

提示：
你可以假设给定的 k 总是合理的，且 1 ≤ k ≤ 数组中不相同的元素的个数。
你的算法的时间复杂度必须优于 O(n log n) , n 是数组的大小。
题目数据保证答案唯一，换句话说，数组中前 k 个高频元素的集合是唯一的。
你可以按任意顺序返回答案。

由于O(nlogn)则可以用哈希表记录两个信息第一个信息为数值，第二信息是出现的频率，然后进行排序(快排),然后依次输出k个值解决问题，此时的时间复杂度为O(nlogn)

1、由于需要排列，则可以用采用优先级队列
2、**如果采用大项堆**那么每次移动大项堆弹出去的都是最大的元素，怎么保存最大的元素，因此采用小项堆，每次将**最小的元素弹出**，最后的**小项堆里积累的才是前k个最大元素.**

priority_queue和deque的区别与联系

1、priority_queue是一个披着队列外衣的堆可以实现队列元素的排序(#include<queue>)
则是一个标准的队列，从队头取元素，从队尾移入元素
priority_queue则是利用**max-heap(大项堆)**完成对**元素的排序**，这个大项堆是**以vector为表现形式**的**complete binary tree(完全二叉树)**
2、deque则是一个底层容器(#include<deque>),可以获取deque的队尾和队头元素，且可以从队头和队尾都进行push和pop操作
3、堆是一棵**完全二叉树**,树中的**每个结点的值**都**不大于(不小于)**其左右孩子的值，如果懒得实现则用priority_queue即可

```
class Solution {
public:
 struct cmp1{
 bool operator()(const pair<int,int>& p1,const pair<int,int>&p2){
 return p1.second>p2.second;//升序排序
 }
 };
    vector<int> topKFrequent(vector<int>& nums, int k) {
 unordered_map<int,int> map;//第一个信息表示数组值,第二个信息表示出现的频率
  //首先要统计元素出现的频率
  for(int i=0;i<nums.size();i++){
  map[nums[i]]++;//如果map中没有key为nums[i]的对象则自动创建
  }
  //定义一个小项堆
  priority_queue<pair<int,int>,vector<pair<int,int>>,cmp1> que;
  //把map中的元素放入小项堆中
  for(unordered_map<int,int>::iterator it=map.begin();it!=map.end();it++)
{
  que.push(*it);//map中存放的是pair对象
  if(que.size()>k){
  que.pop();//弹出最小项
  }
  }
  //找出前k个元素
  vector<int> result(k);
  //倒叙将que插入result数组中
  for(int i=k-1;i>=0;i--){
  result[i]=que.top().first;
  que.pop();
```

```
29   }
30   return result;
31      }
32 };
```