单调栈

单调栈，通过自己定义一些规则，让栈内的元素满足一定的条件，来进行模拟

单调递减栈：①在一个数组中针对每一个元素从它右边寻找第一个比它大的元素

②在一个数组中针对每一个元素从它左边寻找第一个比它大的元素（从后往前遍历

739. 每日温度

```
vector<int> dailyTemperatures(vector<int>& temperatures) {
    //利用单调递减栈，来寻找下一个更大的值
    int n = temperatures.size();
    vector<int> res(n, 0);
    stack<int> st;  //需要用到下标
    for (int i = 0; i < n; i++){
        while(!st.empty() && temperatures[i] > temperatures[st.top()]){
            //栈顶元素之后更大的值是temperatures[i]
            int cur = st.top();
            st.pop();
            res[cur] = i - cur;
        }
        st.push(i);
    }
    return res;
}
```

1019. 链表中的下一个更大节点

```
vector<int> nextLargerNodes(ListNode* head) {
    vector<int> nums;//从链表中获取数据
    while(head){
        nums.push_back(head->val);
        head = head->next;
    }
    int len = nums.size();
    vector<int> res(len, 0);//首先默认所有节点都没有比他更大的结点，方便处理
    stack<int> st;          //维护一个单调递减的栈，来寻找下一个更大的值,存放的是节点的下标
    for (int i = 0; i < nums.size(); i++){
        while(!st.empty() && nums[i] > nums[st.top()]){
            int cur = st.top();//说明nums[i]比当前栈顶元素更大，也就是当前栈顶元素更大的值就是nums[i]
            st.pop();
            res[cur] = nums[i];
        }
        st.push(i);
    }
    return res;
}
```

单调递增栈

利用单调递减栈，可以很快找到>栈顶元素的元素
当前元素 > 栈顶元素
说明已经找到了右边第一个 >栈顶元素
根据单调栈的性质>栈顶元素的距离左边最近的元素位于栈顶的下一个位置
在这里就需要判断，出栈后栈是否为空，如果出栈后栈空，说明左边没有比栈顶元素更大的值

84. 柱状图中最大的矩形

```
int largestRectangleArea(vector<int>& heights) {
    heights.push_back(0);
    int res = 0;
    stack<int> st;
    for (int i = 0; i < heights.size(); i++){
        while(!st.empty() && heights[i] < heights[st.top()]){
            //当前元素比栈顶元素要低，我们就可以确定一栈顶元素为高的矩形面积
            int cur = st.top();
            st.pop();                    //cur指向栈顶元素并出栈
            int left;
            if (st.empty()) {
                left = 0;   //栈顶元素出栈后，栈为空说明左边没有比他要小的元素，left = 0
                res = max(res, i * heights[cur]);
            }else    {
                left = st.top();    //当栈不空，left指向比cur低的元素
                res = max(res, (i - left - 1) * heights[cur]);
            }

        }
        st.push(i);
    }
    return res;
}
```

单调队列

双端队列*deque*

```
#include <deque>
using namespace std;
```

双端队列可以在对头队尾两端进行插入和删除

1、push_back()、push_front()

push_back()、push_front()分别在队尾、队头进行插入

2、pop_back()、pop_front()

pop_back()、pop_front()分别将队尾元素和队头元素弹出

Acwing 窗口内的最小值和最大值

1 3 -1 -3 5 3 6 7

(1)使用普通的队列

让队列的大小为k，每扫描到一个数就将这个数加入到队列中
再在队列中扫描一遍，获得最大值和最小值，算法的时间复杂度O(n * k)

(2)使用单调队列

首先解决最小值

对1 3 -1而言，1、3都要比-1小并且在-1前面，也就是说窗口内的1、3永远不会成为最小值，所以可以将1、3删除只剩下-1。对-3而言，它比窗口内的-1要小，所以-1不会成为最小值，-1可以删除....

最后删除前面这些>nums[i]的数之后，整个窗口内的数成一个递增有序的状态

始终维持队列中的元素递增(可以存在重复元素)

在这里需要注意的是，当nums[i] < nums[q.back()]时，我们**应该将队尾元素出队**，所以使用的是**双端队列**

当遍历到某个元素时，我们就可以用过单调队列来寻找它的最小值(单调队列的队头元素)

```
#include <iostream>
#include <deque>
using namespace std;

deque<int> dq;
const int N = 1000010;
int nums[N];

int main(){
    int n, k;
    cin>>n>>k;
    for (int i = 0; i < n; i++){
        cin>>nums[i];
    }
    for (int i = 0; i < n; i++){
        if (dq.size() && i-dq.front()+1 > k)  dq.pop_front(); //只保留i和i之前的k个元素
        while(dq.size() && nums[dq.back()] > nums[i]){//维护一个单调递增的双向队列
            dq.pop_back();
        }
        dq.push_back(i);
        if (i >= k-1)   cout<<nums[dq.front()]<<" ";
    }
    cout<<endl;

    dq.clear();
    for (int i = 0; i < n; i++){
        if (dq.size() && i-dq.front()+1 > k)  dq.pop_front(); //只保留i和i之前的k个元素
        while(dq.size() && nums[dq.back()] < nums[i]){//维护一个单调递减的双向队列
            dq.pop_back();
        }
        dq.push_back(i);
        if (i >= k-1)   cout<<nums[dq.front()]<<" ";
    }
    return 0;
}
```