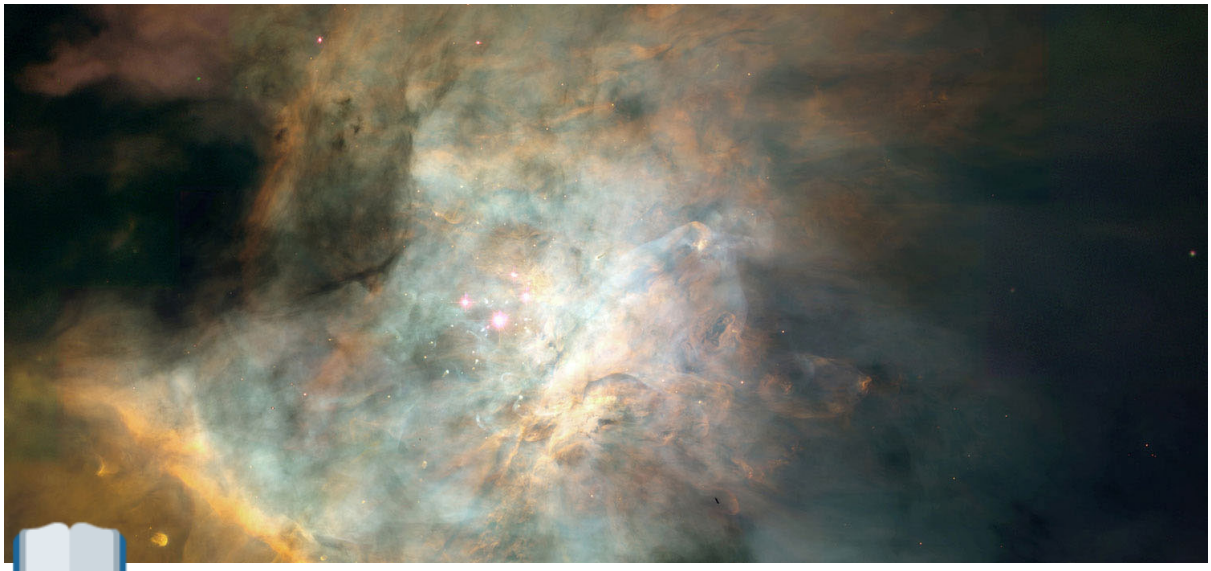


数据结构(I)

作者：Zhuoran Wang Email: wangzhuoran200105@126.com

本节内容包括了链表、队列、栈、KMP、Trie树、并查集、堆、哈希等常用数据结构

链表LinkedList

1. 单链表

单链表在算法竞赛或者在笔试里最常用的作用就是写**邻接表**，邻接表的作用是**存储树和图**

这里单链表我们用数组模拟，也可以采用结构体模拟；后者更好理解但是在算法竞赛中出于对效率的考虑，我们用数组模拟单链表（以及后续的所有数据结构）

当然也可以使用C++或是Java自带的单链表，但是这些效率都不如数组高

> 这里实现的单链表又叫静态链表，写工程的时候常用动态链表

数组模拟单链表：

head存储链表头，e[]存储节点的值，ne[]存储节点的next指针，idx表示当前用到了哪个节点

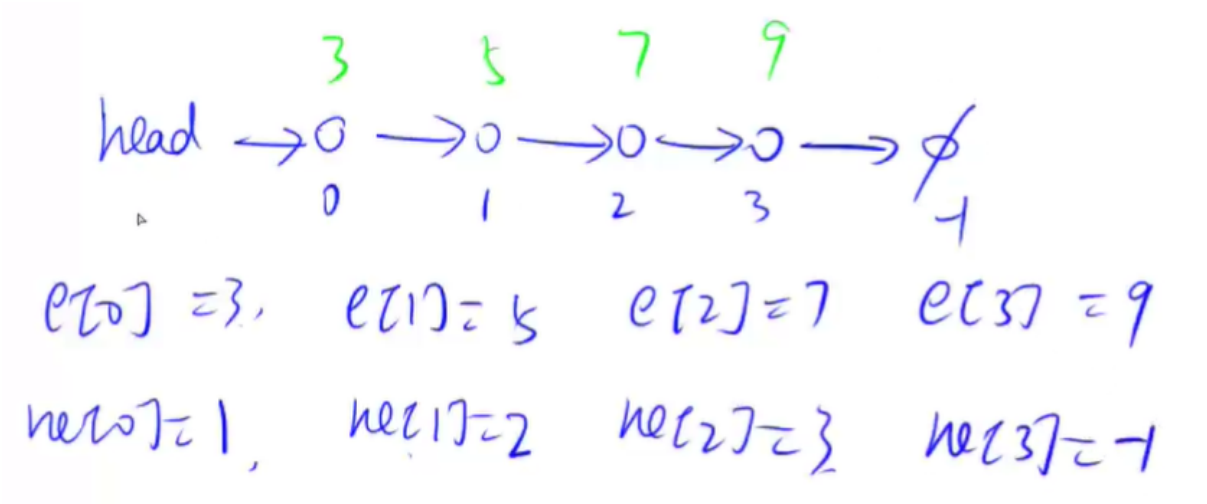
e[]和ne[]通过下标关联起来

ne[]里存的是当前点下一个点的下标是多少，如图，0号点的下一个点是1号点，因此ne[0]=1

空节点用-1表示

idx实际上维护的同时也是链表的长度，代表着数组e和ne用到了哪里

head储存的是表头元素的下标




```
// head存储链表头，e[]存储节点的值，ne[]存储节点的next指针，idx表示当前用到了哪个节点
int head, e[N], ne[N], idx;

// 初始化
```

```
void init()
{
    head = -1;
    idx = 0;
}

// 在链表头插入一个数a
void insert(int a)
{
    e[idx] = a, ne[idx] = head, head = idx ++ ;
    //ne[idx] = head是说，这个点的下一个点是当前head指向那个点的下标
    //然后我们要更新head的值，也就是更新头节点的下标值为我们刚刚插入的那个点的下标值
    //最后更新idx的值为数组里没有用过的下标
}

// 将头结点删除，需要保证头结点存在
void remove()
{
    head = ne[head];
}
```

```
#include <iostream>
#include <algorithm>

using namespace std;

const int N = 100010;
int head, e[N], ne[N], idx;

void init(){
    head = -1;
    idx = 0;
}

void add_to_head(int x){
    e[idx] = x, ne[idx] = head, head = idx ++;
}
//在第k个点后面加入x
void insert(int k, int x){
    e[idx] = x, ne[idx] = ne[k], ne[k] = idx ++;
}
//删除第k个点，这里的第k个指插入的第k个，也就是数组里的第k-1的点
//第k个插入的点的下标是k-1
//下标是从0开始的，但是插入的第几个点的计数是从1开始的
void remove(int k){
    ne[k] = ne[ne[k]];
}

int main(){

    init();
    return 0;
}
```

2. 双链表

最常用的场景是优化某些问题

Code

```
int e[N], l[N], r[N], idx;

//这里偷个懒：0号位置表示head，1号位置表示tail
void init(){
    r[0] = 1, l[1] = 0;
    idx = 2;
}


// 在节点k的右边插入一个数x
void add(int k, int x){
    e[idx] = x;
    r[idx] = r[k];
    l[idx] = k;
    l[r[k]] = idx;
    r[k] = idx;
    idx++;
}
//要在k的左边插入，直接add(l[k], x)就可以了
//注意不是k-1


void remove(int k){
    r[l[k]] = r[k];
    l[r[k]] = l[k];
}
```

栈 Stack

栈是一种先进后出的数据结构

栈的代码模板

```
// tt表示栈顶
int stk[N], tt = 0;

//插入
void push(int x){
    stk[++tt] = x;
```

```
}

//弹出
void pop(){
    tt--;
}

//判断栈是否为空
bool isEmpty(){
    if(tt>0) return false;
    else return true;
}

//查看栈顶元素
int peak(){
    return stk[tt];
}
```


写代码一定要练习熟练度！就像背单词和古诗一样，是从记忆中提取出来的。做题的时候不能从头再推一边，而是要理解着记忆模板！

单调栈

常用场景：给定一个序列，求一下每一个数左边，离他最近并且比他小的数；或者对称一下，在他右边并且比他大的数

给定一个长度为N的整数数列，输出每个数左边第一个比它小的数，如果不存在则输出-1。

输入格式

第一行包含整数N，表示数列长度。

第二行包含N个整数，表示整数数列。

输出格式

共一行，包含N个整数，其中第i个数表示第i个数的左边第一个比它小的数，如果不存在则输出-1。

数据范围

1≤N≤10^5, 1≤数列中元素≤10^9

输入样例：

```
5
3 4 2 7 5
```

输出样例：

```
-1 3 -1 2 2
```

题解

首先想一下暴力做法，利用双指针二重循环可以搞定

```
for(int i = 0; i<n; i++){
  for(int j = i-1; j>=0; j--){
    if(a[i]>a[j]){
      cout<<a[j]<<endl;
      break;
    }
  }
}
```

然后考虑是否有性质可以优化这种解法，发现当$a_x \geq a_y (x < y)$的时候，a_x以后永远用不到了，（a_y比a_x小，而且还在a_x后面，因此后面的元素会优先选择a_y作为答案输出出来）。并且注意到，如果上面这种状况发生，可以用栈模拟。

最后，所有逆序的$a_x \geq a_y (x < y)$中的a_x都被弹出栈了，栈最后的样子是一个单调的增函数。

```
#include <iostream>

using namespace std;

const int N = 100010;

int n;
int stk[N], tt = 0;

int main(void){
    cin>>n;

    for(int i = 0; i<n; i++) {
        int x;
        cin>>x;
        while(tt && stk[tt]>=x){
            tt--;
```

```
        }

        if(tt) cout<<stk[tt]<<' ';
        else cout << -1 <<' ';

        stk[++ tt] = x;
    }

    return 0;
}
```

队列Queue

队列是一种先进先出的数据结构

队列代码模板

```
// hh 表示队头，tt表示队尾
int q[N], hh = 0, tt = -1;

// 向队尾插入一个数
void push(int x){
    q[ ++ tt] = x;
}

// 从队头弹出一个数
void pop(){
    hh++;
}

int peak(){
    return q[hh];
}

bool isEmpty(){
    if(hh>tt){
        return true;
    }else return false;
}
```

单调队列

应用场景：滑动窗口

给定一个大小为n≤10^6的数组。

有一个大小为k的滑动窗口，它从数组的最左边移动到最右边。

您只能在窗口中看到k个数字。

每次滑动窗口向右移动一个位置。

以下是一个例子：

该数组为[1 3 -1 -3 5 3 6 7]，k为3。

例子

窗口位置	最小值	最大值
[1 3 -1] -3 5 3 6 7	-1	3
1 [3 -1 -3] 5 3 6 7	-3	3
1 3 [-1 -3 5] 3 6 7	-3	5
1 3 -1 [-3 5 3] 6 7	-3	5
1 3 -1 -3 [5 3 6] 7	3	6
1 3 -1 -3 5 [3 6 7]	3	7

您的任务是确定滑动窗口位于每个位置时，窗口中的最大值和最小值。

输入格式

输入包含两行。

第一行包含两个整数n和k，分别代表数组长度和滑动窗口的长度。

第二行有n个整数，代表数组的具体数值。

同行数据之间用空格隔开。

输出格式

输出包含两个。

第一行输出，从左至右，每个位置滑动窗口中的最小值。

第二行输出，从左至右，每个位置滑动窗口中的最大值。

输入样例：

```
8 3
1 3 -1 -3 5 3 6 7
```

输出样例：

```
-1 -3 -3 -3 3 3
3 3 5 5 6 7
```

题解：

首先还是考虑暴力做法，可以用一个队列来维护当前的滑动窗口，经过分析，时间复杂度是$O(kn)$, 这样大概率会超时，因此我们也要考虑如何优化。

偏个题：STL最大的缺点就是慢.....和编译器开不开$O2$优化有关

优化思路与单调栈类似，用队列来维持滑动窗口，并构建单调递增序列

```
#include<iostream>
using namespace std;
const int N = 1e6 + 10;
int n, k, q[N], a[N];//q[N]存的是数组下标
int main()
{
    int tt = -1, hh=0;//hh队列头 tt队列尾
    cin.tie(0);
    ios::sync_with_stdio(false);
    cin>>n>>k;
    for(int i = 0; i <n; i ++) cin>>a[i];
    for(int i = 0; i < n; i ++)
    {
        //维持滑动窗口的大小
        //当队列不为空(hh <= tt) 且 当当前滑动窗口的大小(i - q[hh] + 1)>我们设定的
        //滑动窗口的大小(k),队列弹出队列头元素以维持滑动窗口的大小
        if(hh <= tt && k < i - q[hh] + 1) hh ++;
        //构造单调递增队列
        //当队列不为空(hh <= tt) 且 当队列队尾元素>=当前元素(a[i])时,那么队尾元素
        //就一定不是当前窗口最小值,删去队尾元素,加入当前元素(q[ ++ tt] = i)
        while(hh <= tt && a[q[tt]] >= a[i]) tt --;
        q[ ++ tt] = i;
        if(i + 1 >= k) printf("%d ", a[q[hh]]);
    }
    puts("");
    hh = 0,tt = -1;
    for(int i = 0; i < n; i ++)
    {
        if(hh <= tt && k < i - q[hh] + 1) hh ++;
        while(hh <= tt && a[q[tt]] <= a[i]) tt --;
        q[ ++ tt] = i;
        if(i + 1 >= k ) printf("%d ", a[q[hh]]);
    }
    return 0;
}
```