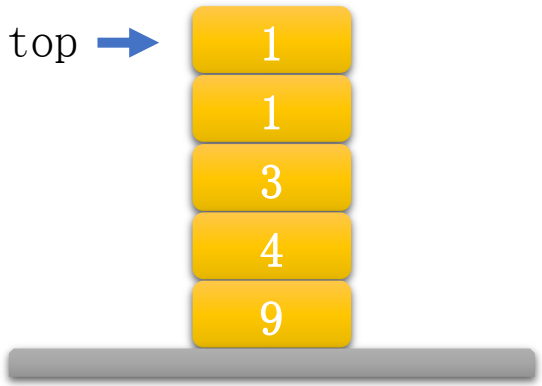
基础数据结构

湖南师大附中 许力

栈的拓展：单调栈

- 栈中元素保持单调不增/不减的，即为单调栈



- 这里的递增/递减，严格来说应从栈顶-栈底顺序，但有时也常常按入栈顺序（即栈底-栈顶）

栈的拓展：单调栈

- 栈中元素保持单调不增/不减的，即为单调栈

- 比如：数列[1,2,3,0,6]，要维护一个单调递增的栈（按入栈顺序）

1. 1待入栈，栈空，栈[1]
2. 2待入栈，2>1，栈[1,2]
3. 3待入栈，3>2，栈[1,2,3]
4. 0待入栈，此时3>0，弹出3，2>0，弹出2，1>0，弹出1，栈[0]
5. 6待入栈，6>0，最后栈[0,6]

单调栈

- 那么单调栈有什么用呢？
- 还是这个数列[1,2,3,0,6]，我们所维护的单调递增栈（按入栈顺序），当前待入栈数入栈前的栈顶数，就是比当前待入栈数小的最近的数

- 比如[1,2,3,0,6]中比第i个数小的第一个数分别是[0,1,2,0,0]

1. 1待入栈，栈空
2. 2待入栈，栈顶数1就是比2小的第一个数
3. 3待入栈，栈顶数2就是比3小的第一个数
4. 0待入栈，需要弹栈顶元素至空
5. 6待入栈，栈顶数0就是比6小的第一个数

最小元素

- 给定一个长为n的整数序列，求出序列中每个元素右边第一个比该元素小的元素。如果没有则输出“0”

$n \leqslant 10^6$

Sample input	Sample output
8 //n 7 2 1 4 5 1 3 2	2 1 0 1 1 0 2 0

分析

- 这题暴力做法$O(n^2)$

- 利用单调栈怎么做？[7,2,1,4,5,1,3,2]

1. 维护一个单调递增的栈（按入栈顺序），从左至右遍历序列
2. 考虑当前处理的元素x，栈顶元素为y

分析

- 利用单调栈怎么做？[7,2,1,4,5,1,3,2]

3. 如果x<y，那么可以知道y右边第一个比y小的元素就是x，为什么？
4. x就是此时对应的答案，输出x
5. 将y从栈中弹出（此时要维持栈中的单调递增性质）

分析

- 利用单调栈怎么做？[7,2,1,4,5,1,3,2]

6. 弹出y之后，要么栈已经空了，要么新的栈顶元素y≤x
7. 将x压入栈中，栈中元素依然保持单调递增

分析

- 利用单调栈怎么做？[7,2,1,4,5,1,3,2]

6. 弹出y之后，要么栈已经空了，要么新的栈顶元素y≤x
7. 将x压入栈中，栈中元素依然保持单调递增

分析

- 利用单调栈怎么做？[7,2,1,4,5,1,3,2]

6. 弹出y之后，要么栈已经空了，要么新的栈顶元素y≤x
7. 将x压入栈中，栈中元素依然保持单调递增

分析

- 利用单调栈怎么做？[7,2,1,4,5,1,3,2]

6. 弹出y之后，要么栈已经空了，要么新的栈顶元素y≤x
7. 将x压入栈中，栈中元素依然保持单调递增

分析

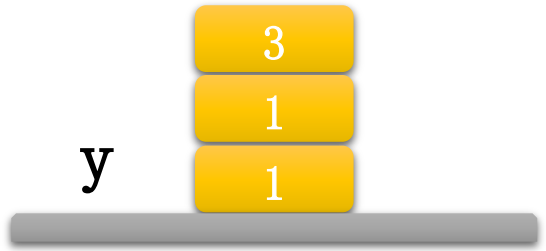
- 利用单调栈怎么做？[7,2,1,4,5,1,3,2]

6. 弹出y之后，要么栈已经空了，要么新的栈顶元素y≤x
7. 将x压入栈中，栈中元素依然保持单调递增

分析

- 利用单调栈怎么做？[7,2,1,4,5,1,3,2]

6. 弹出y之后，要么栈已经空了，要么新的栈顶元素y≤x
7. 将x压入栈中，栈中元素依然保持单调递增



分析

- 利用单调栈怎么做？[7,2,1,4,5,1,3,2]

8. 整个序列遍历完毕后，栈中剩下的元素均属于无解的情况

最后得[2,1,0,1,1,0,2,0]

分析

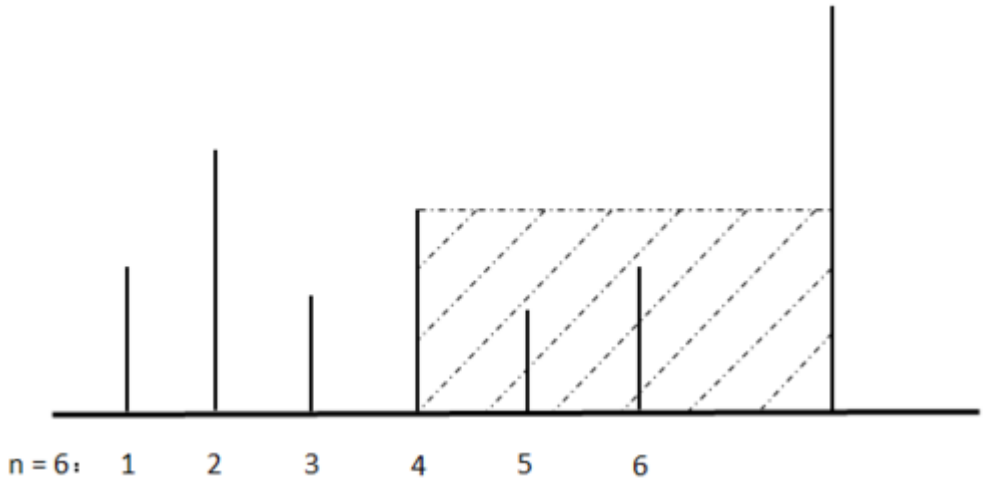
- 时间复杂度O(n)

- 单调栈的本质是，在找第一个比某数小的问题时，当一个数a在另一个数b的前面，而且还比b大，那么数a就完全没有必要考虑了：它被b完全屏蔽了

木板

- 地上从左到右竖立着n块木板，从1到n依次编号。我们知道每块木板的高度，在第n块木板右侧竖立着一块无限高的木板
- 现在对每块木板依次做如下的操作：对于第i块木板，我们从其右侧开始倒水，直到水的高度等于第i块木板的高度，倒入的水会淹没a_i块木板（如果木板左右两侧水的高度≥木板高度即视为木板被淹没）
- 求对所有木板依次完成该操作后，所有a_i的和是多少

Sample input	Sample output
5 //n 10 5 8 12 6 //a_i	3

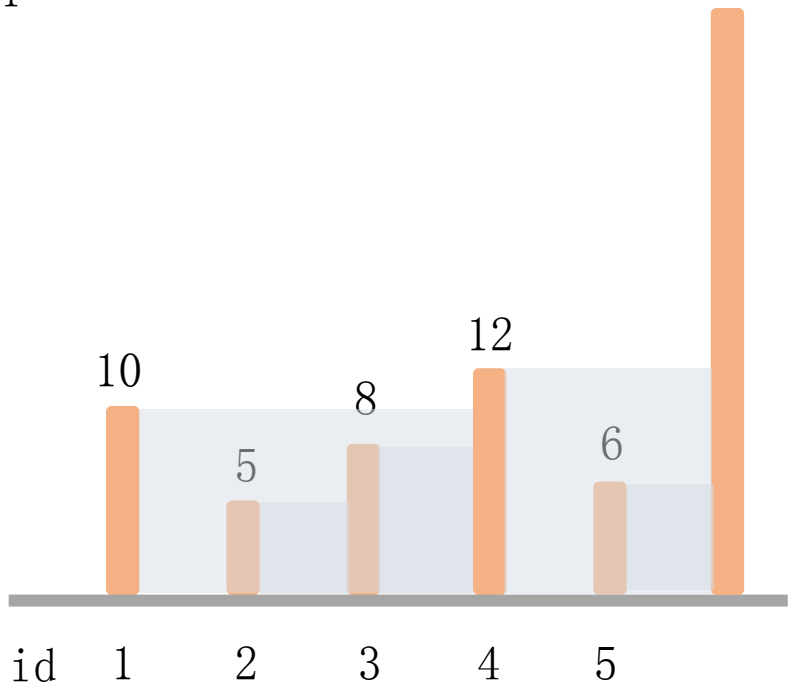


分析

- 我们首先看看样例数据怎么得到：

1. i=1，木板高度10，可以淹没2号和3号木板，a_1=2
2. i=2，木板高度5，无法淹没木板，a_2=0
3. i=3，木板高度8，无法淹没木板，a_3=0
4. i=4，木板高度12，可以淹没5号木板，a_4=1
5. i=5，木板高度6，无法淹没木板，a_5=0

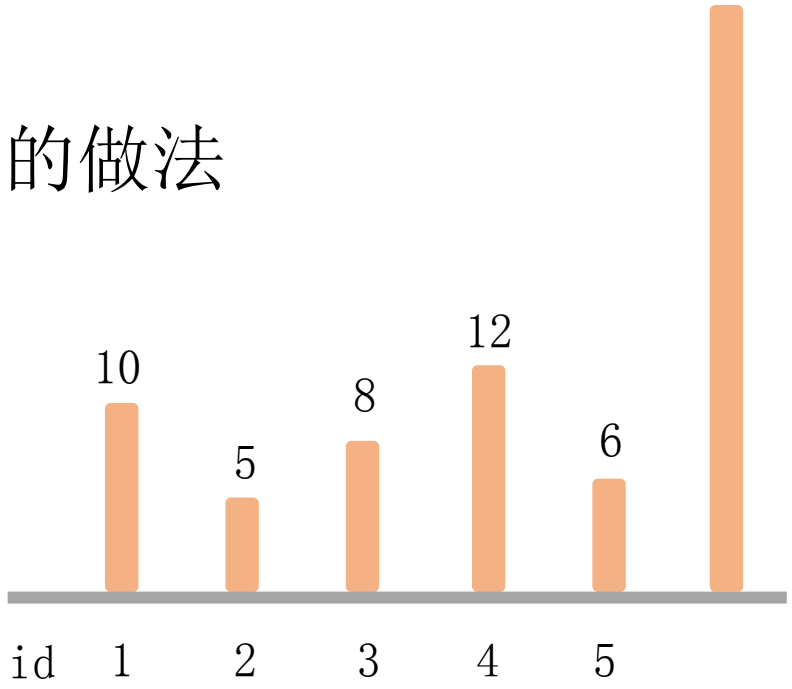
总和：2+0+0+1+0=3



分析

- 问题实际上就是：寻找第i个数右边第一个比它大的数，然后统计两个数之间的数的个数

- 依然可以通过维护一个单调栈来实现O(n)的做法



分析

- 依然可以通过维护一个单调栈来实现O(n)的做法

1. 栈空，10直接入栈

此时栈顶即为10，也就是接下来要寻找比10大的木板

分析

- 依然可以通过维护一个单调栈来实现O(n)的做法

2. 5<10，5直接入栈

此时栈顶为5，也就是接下来要寻找比5大的木板

分析

- 依然可以通过维护一个单调栈来实现O(n)的做法

3. 8>5，说明找到了一块>5的木板，计算a_2=3-2-1
4. 弹出栈顶5
5. 当前栈顶为10，8<10，8入栈

此时栈顶为8，也就是接下来要寻找比8大的木板

分析

- 依然可以通过维护一个单调栈来实现O(n)的做法

6. 12>8，说明找到了一块>8的木板，计算$a_3=4-3-1=0$
7. 弹出栈顶8
8. 当前栈顶为10，12>10，继续弹出栈顶10

此时栈空，说明比10大的（第一块）木板已经找到了，就是12

计算$a_1=4-1-1=2$

分析

- 依然可以通过维护一个单调栈来实现O(n)的做法

9. 栈空，12直接入栈

此时栈顶即为12，也就是接下来要寻找比12大的木板

分析

- 依然可以通过维护一个单调栈来实现O(n)的做法

9. 6<12，6直接入栈

此时栈顶即为6，也就是接下来要寻找比6大的木板

10. 数列结束

此时栈为[12,6]，栈顶6

分析

- 依然可以通过维护一个单调栈来实现O(n)的做法

11. 此时最右端有无限高的木板

计算a_5=6-5-1

计算a_4=6-4-1

sum=$a_1+a_2+a_3+a_4+a_5$

分析

- 从左往右元素依次待入栈
- 遇到比栈顶大的元素，就弹出当前栈顶，并计算答案a_i=i-top_id-1
- 直到元素比当前栈顶小或者栈空，待入栈元素就直接入栈
- 元素全都入栈后，栈内剩余的元素都是右侧没有比它们更大的，此时依次出栈，并计算a_i=(n+1)-top_id-1（n+1是最右边无限高木板的id）

课外加练

- luogu 2947　　向右看齐
- luogu 1823　　音乐会的等待
- luogu 1901　　发射站

队列（queue）

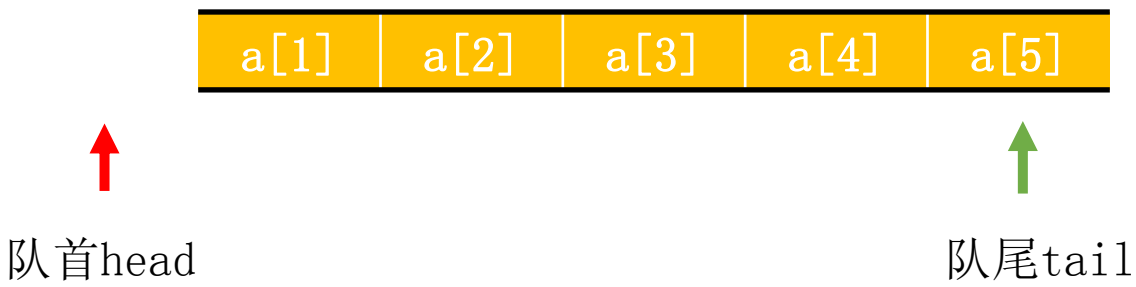
- 先进先出（FIFO）的线性数据结构

- 其模型类似于排队，先来的同学排在前面并先上车；后来的同学只能排在队尾并依序前进

队列

- 队列和栈不同，队列两头都可以操作
- 但是一端为队首，只支持删除操作；一端为队尾，只支持插入操作

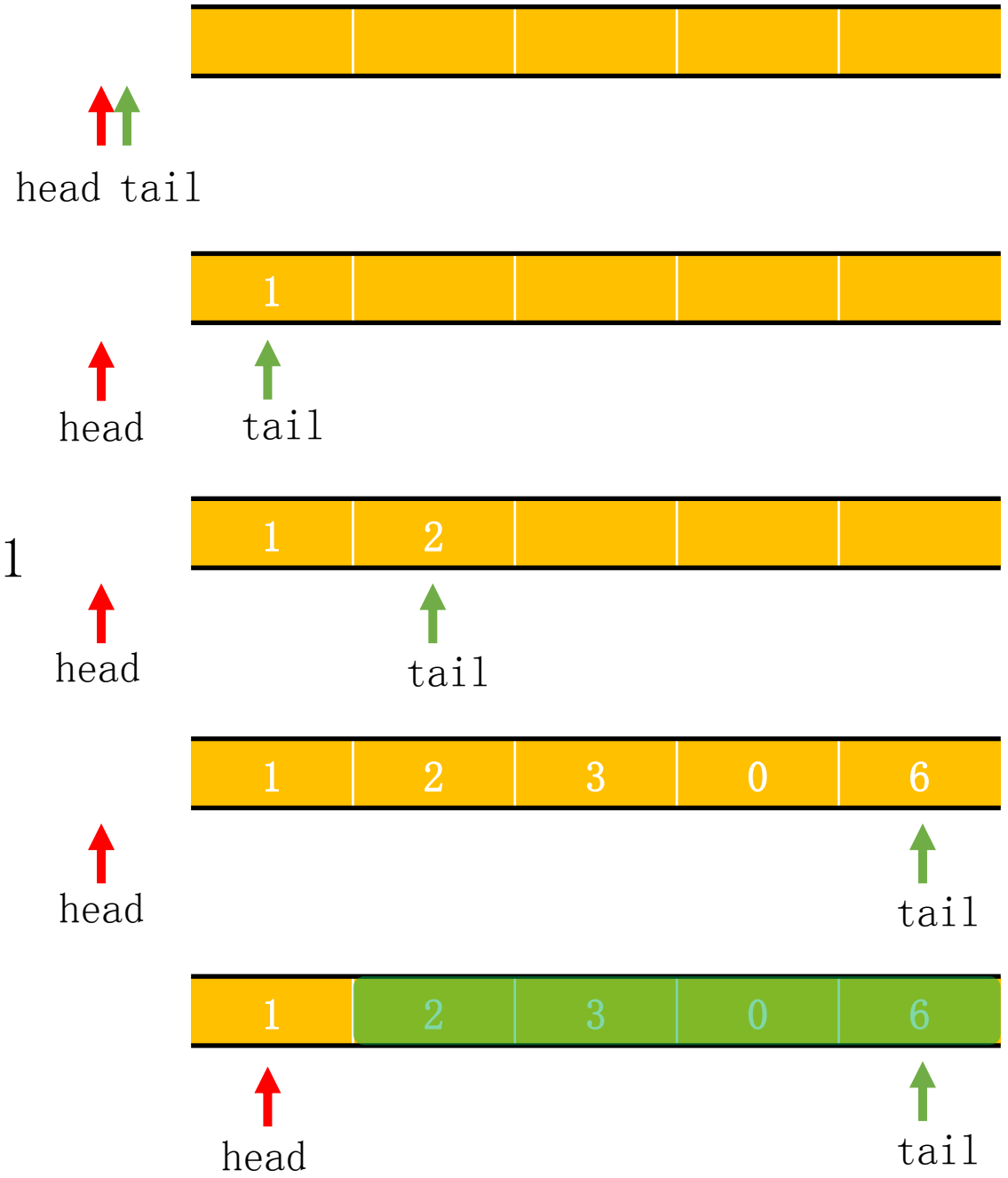
- 当然在实际操作中，不可能像实际生活中的排队那样让整个队列一点点平移，那代价太高了
- 实际移动的是首尾指针变量head、tail
- head指向队首的前一个id，tail指向队尾所在的id



队列

- 以数列[1, 2, 3, 0, 6]为例

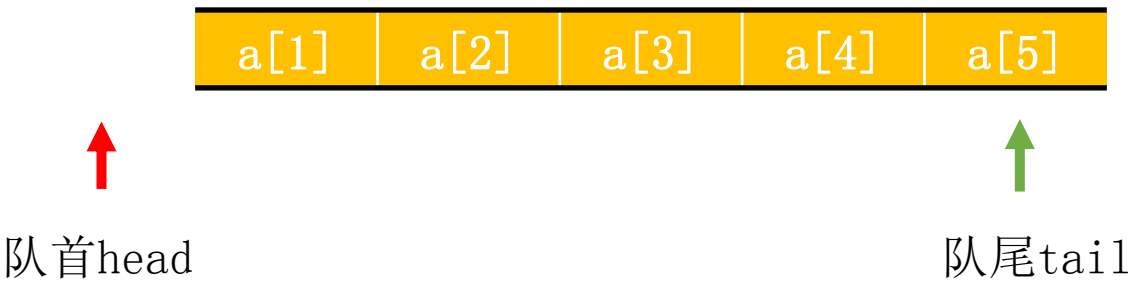
1. 1入队，当前队首1
2. 2入队，当前队首1
3. 3、0、6依次入队，当前队首1
4. 1出队，当期队首2



队列的代码实现

1. 数组模拟实现队列

代码要求：用数组模拟实现队列的日常操作：入队、出队



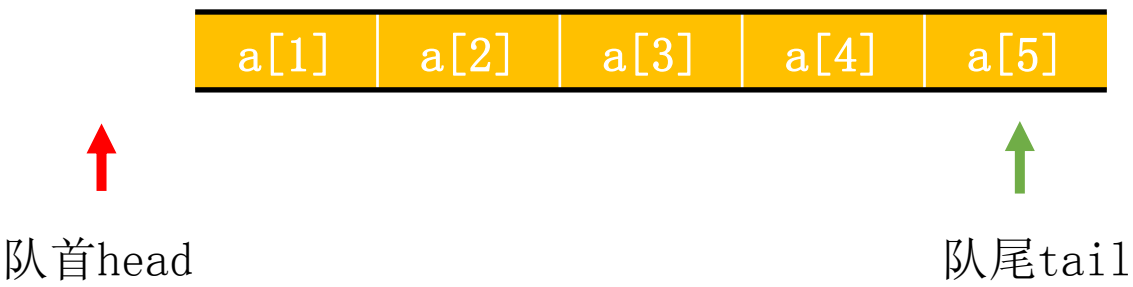
队列的代码实现

1. 数组模拟实现队列

需要分别记录队首head、队尾tail

入队：tail++

出队：head++



参考代码

```
5
1 2 3 0 6
1 2 3 0 6
--------------------------------
Process exited after 6.405 seconds wi
th return value 0
请按任意键继续. . .
```

```cpp
#include<bits/stdc++.h>
using namespace std;
#define N 100010
int que[N], head, tail;
int main()
{
    int n, x;
    scanf("%d", &n);
    while (n --)
        scanf("%d", &x), que[tail++] = x;
    while (head < tail)
        printf("%d ", que[head]), que[head++];
    return 0;
}
```

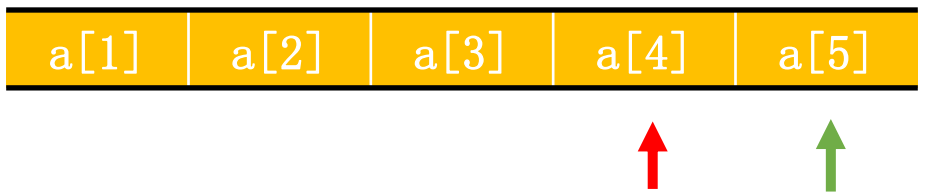
队列的代码实现

1. 数组模拟实现队列

但和栈不一样的是：这样模拟很可能会出事！

- 因为有head、tail两个指针一前一后“++”，所以，会出现长度为n的数组无法模拟长度为n的队列的情况
- 随着大量的入队、出队操作，可能队列本身不超过n，但是已经严重偏离数组的起始位置了

- 这种情况被称为“假溢出”

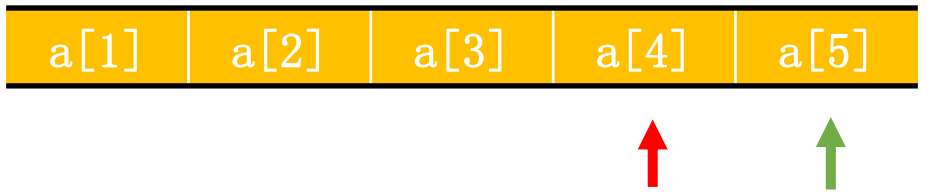


队列的代码实现

1. 数组模拟实现队列

① 解决办法之一，是把数组大小翻倍，但浪费内存不说，依然有溢出可能（某些元素重复出队又入队）

② 解决办法之二，是把数组首尾循环利用，虚拟成环。这样长度为n的数组就可以模拟长度为n的队列



循环队列

- 这种结构的队列被称为“循环队列”
- 元素x入队：

```
que[tail++] = x; if (tail == N) tail = 0;
```

- 元素出队：

```
que[head++]; if (head == N) head = 0;
```

- 但是新的问题产生了：head有可能出现在tail的右边！

循环队列

- 所以队列判空的条件得改
- 原来是：

```
while (head < tail)
```

- 但是head如果出现在tail右边时，队列也可能满，所以要改为：

```
while (head != tail)
```

参考代码

```cpp
#include<bits/stdc++.h>
using namespace std;
#define N 100010
int que[N], head, tail;
int main()
{
    int n, x;
    scanf("%d", &n);
    while (n --)
    {
        scanf("%d", &x);
        que[tail++] = x; if (tail == N) tail = 0;
    }
    while (head != tail)
    {
        printf("%d ", que[head]);
        que[head++]; if (head == N) head = 0;
    }
    return 0;
}
```

队列的代码实现

2. STL实现队列

STL中自带标准队列：queue。具体用法：

1. 定义：queue<int> q
2. 元素x入队：q.push(x)
3. 返回队首元素：q.front()
4. 队首元素出队：q.pop()
5. 返回当前队列大小：q.size()
6. 队列空：q.empty()

参考代码

```
5
1 2 3 0 6
5
1 2 3 0 6
--------------------------------
Process exited after 4.009 seconds wi
th return value 0
请按任意键继续. . .
```

```cpp
#include<bits/stdc++.h>
using namespace std;
int main()
{
    queue<int> q;
    int n, x;
    scanf("%d", &n);
    for (int i = 0; i < n; i ++)
        scanf("%d", &x), q.push(x);
    printf("%d\n", q.size());
    while (!q.empty())
    {
        printf("%d ", q.front());
        q.pop();
    }
    return 0;
}
```

队列的拓展：优先队列

- 优先队列是一种特殊的队列，优先队列中的元素始终保持有序（默认从大到小），即便有元素入队/出队也是如此（注意这和后面的单调队列是有区别的）

- 比如数列[1,2,3,0,6]，逐个入队后，队列里长这样：

优先队列

- 优先队列是一种特殊的队列，优先队列中的元素始终保持有序（默认从大到小，注意这和后面的单调队列是有区别的），即便有元素入队/出队也是如此

- 然后弹出队首6，新元素4入队，队列里长这样：

优先队列的代码实现

- STL中自带优先队列：priority_queue。具体用法：

1. 定义：priority_queue<int> q（默认从大到小）
2. 定义：priority_queue<int, vector<int>, grearer<int> > q（从小到大）
3. 元素x入队：q.push(x)
4. 返回队首元素：q.top()
5. 队首元素出队：q.pop()
6. 返回队列长度：q.size()
7. 队列空：q.empty()

参考代码（从大到小）

```
5
1 2 3 0 6
6 3 2 1 0
--------------------------------
Process exited after 3.259 seconds wi
th return value 0
请按任意键继续. . .
```

```cpp
#include<bits/stdc++.h>
using namespace std;
int main()
{
    priority_queue<int> q;
    int n, x;
    scanf("%d", &n);
    while (n --)
        scanf("%d", &x), q.push(x);
    while (!q.empty())
        printf("%d ", q.top()), q.pop();
    return 0;
}
```

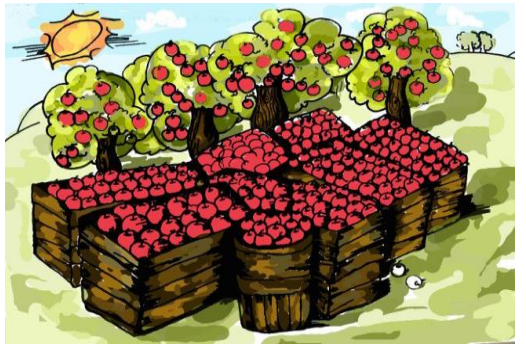
参考代码（从小到大）

```
5
1 2 3 0 6
0 1 2 3 6
--------------------------------

Process exited after 5.087 seconds wi
th return value 0
请按任意键继续. . .
```

```cpp
#include<bits/stdc++.h>
using namespace std;
int main()
{
    priority_queue<int,vector<int>,greater<int> > q;
    int n, x;
    scanf("%d", &n);
    while (n --)
        scanf("%d", &x), q.push(x);
    while (!q.empty())
        printf("%d ", q.top()), q.pop();
    return 0;
}
```

合并果子



- 果园里，陶陶已经将所有的苹果打了下来，而且分成了不同的堆。陶陶决定把所有的苹果合成一堆。每一次合并，陶陶可以把两堆苹果合并到一起，消耗的体力等于两堆苹果的数量之和。可以看出，所有的苹果经过 n-1 次合并之后， 就只剩下一堆了。已知每堆苹果的数目，你的任务是设计出合并的先后次序方案，使耗费的体力之和最少，输出最小的体力耗费值。
- 输入数据为两行：第一行为一个正整数n，表示有n堆果子（n≤10,000）第二行有n个用空格隔开的正整数，表示每堆果子的数量

Sample input	Sample output
3 //n 1 2 9	15

分析

- 这题我们之前模拟/贪心就做过了，现在回顾一下做法：

1. 找出最小的和次小的
2. 将这两个数从数组中删除，并将它们的和重新加入数组，同时累计答案
3. 重复以上两步直到剩下最后一个数，最后一个数不加

分析

1. 找出最小的和次小的

完全可以通过维护一个从小到大的优先队列，连续两次返回队首元素就可做到这一步（还需要使队首元素出队）

2. 将这两个数删除，并将它们的和重新加入

上一步的队首元素已经两次出队，只需将它们的和再入队即可

3. 重复以上两步直到剩下最后一个数，最后一个数不加

以上两步重复n-1次即可做到

课外加练

- luogu 1190　　接水问题
- luogu 1334　　瑞瑞的木板
- luogu 2827　　蚯蚓

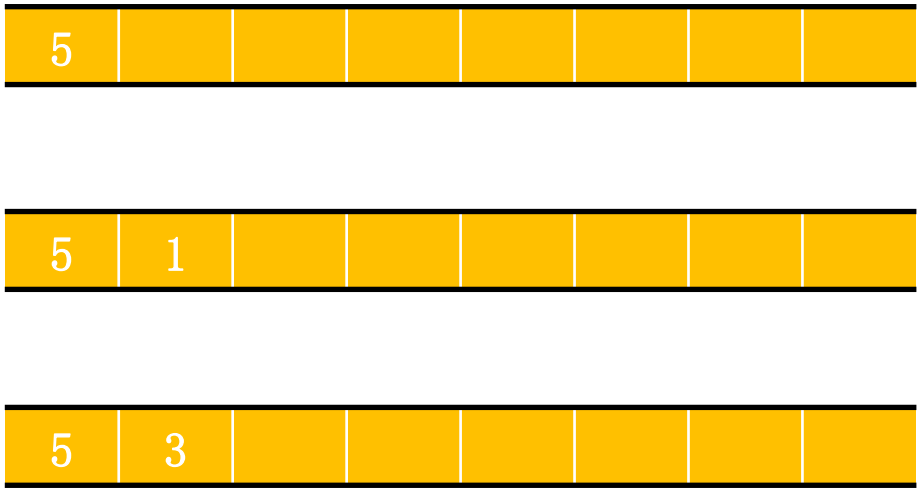
队列的拓展：单调队列

- 类似单调栈，单调队列即是队列元素保持单调递增/递减的队列

单调队列

- 假定我们现在要维护一个长度最大为k的单调递减队列
- 原数列为[5,1,3,6,7,6,2,1]，k=3

1. 5入队尾
2. 1<5，1入队尾
3. 3≥1，1出队尾，3<5，5不出队，3入队尾



- 从这里可以看出：单调队列既可以弹出队首，也可以弹出队尾!
- 单调队列和队列不同，它其实是双端队列deque

单调队列

- 在第3步，蕴含了一个非常重要的单调思想：

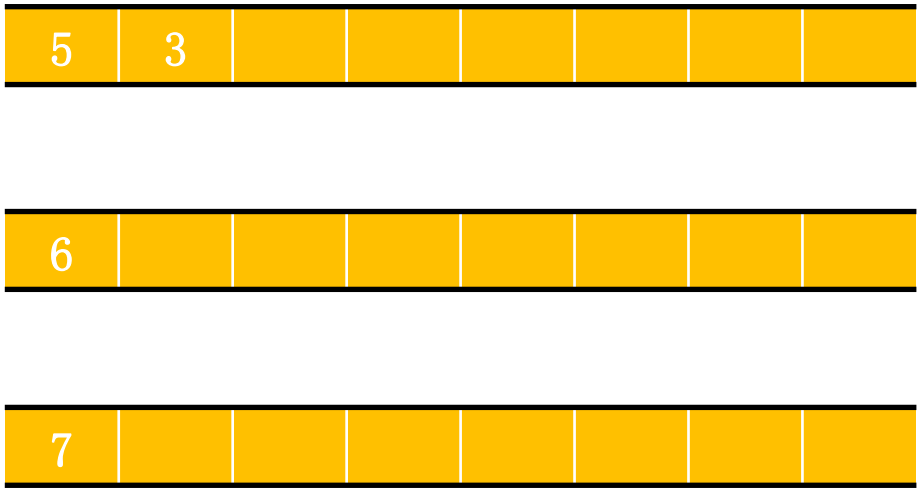
1. 5入队尾
2. 1<5，1入队尾
3. 3≥1，1出队尾，3<5，5不出队，3入队尾

- **如果a[i]<a[j]，且i<j，那么在维护单调递减队列时，a[j]比a[i]更优**
- 意思就是：a比b要小，又还在b的前面，那就舍弃a，保留b
- 这句话的前提是我们在维护一个从左至右的单调递减序列，单调递增则相反

单调队列

- 假定我们现在要维护一个长度最大为k的单调递减队列
- 原数列为[5,1,3,6,7,6,2,1]，k=3

4. 6≥3，3出队尾
5. 6≥1，1出队尾
6. 6≥5，5出队尾
7. 队列空，6直接入队尾
8. 7≥6，6出队尾
9. 队列空，7直接入队尾

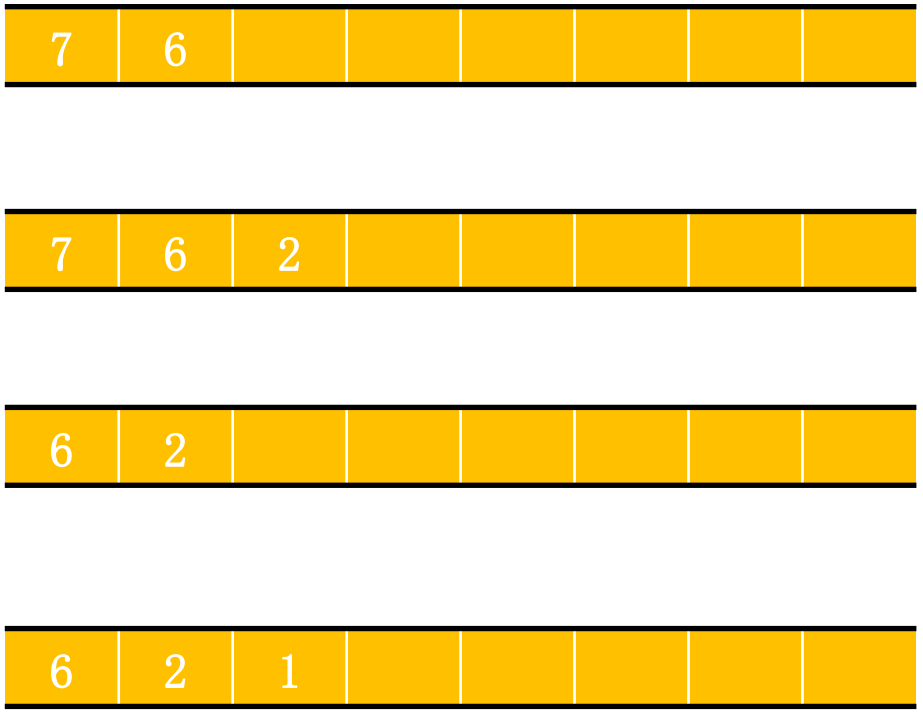


单调队列

- 假定我们现在要维护一个长度最大为k的单调递减队列
- 原数列为[5,1,3,6,7,6,2,1]，k=3

7. 6<7，6入队尾
8. 2<6，2入队尾
9. 但此时k≥3，7出队首
10. 1<2，1入队尾
11. 此时虽然k≥3，但没有新元素待入队

所以6无需出队首



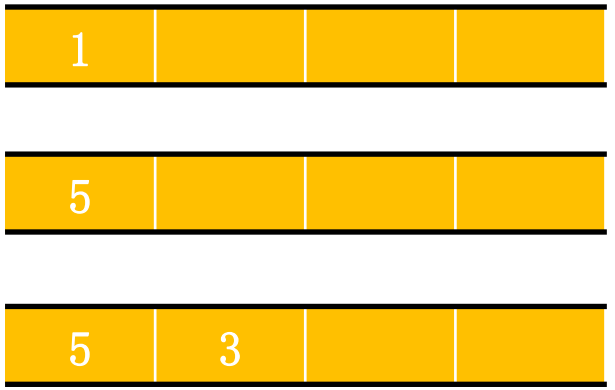
单调队列

- 那么单调队列有什么用呢？
- 它可以用于解决数列中长度固定的区间的最值问题，比如
- 数列[1,5,3,4,8,9,6,0,7,3,6]，k=3，以求最大值为例

单调队列

- 具体做法前面已经演示过，这里再总结一下

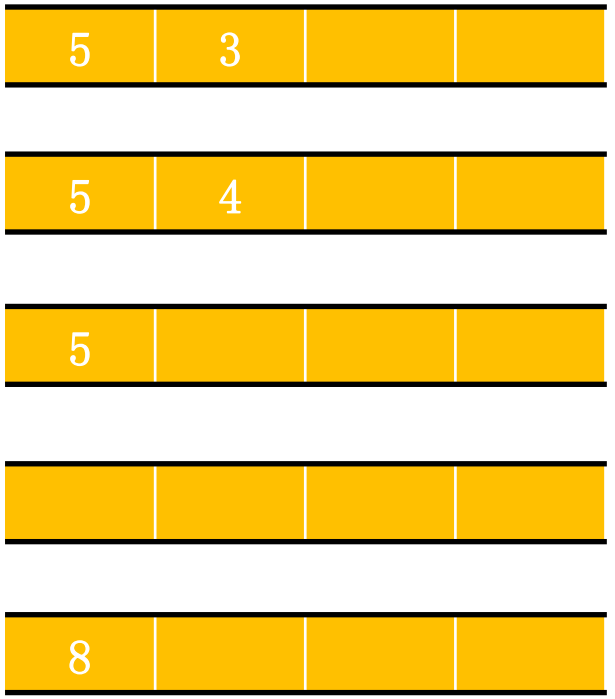
1. 从左至右遍历数列，队列空，1直接入队尾
2. 待入队元素5≥队尾元素1，1从队尾弹出，5入队尾
3. 待入队元素3<队尾元素5，3入队尾
4. **因为此时已经有三个元素先后入队，返回此时的队首5即为第一个ans**



单调队列

- 具体做法前面已经演示过，这里再总结一下

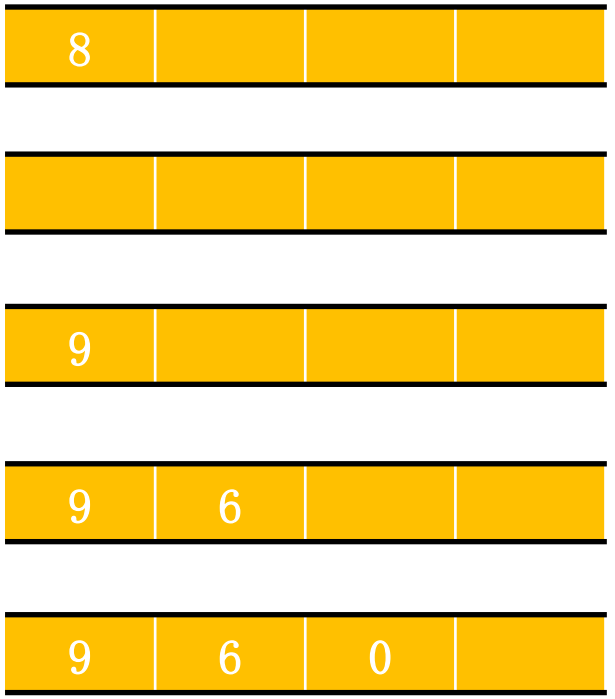
5. 待入队元素4≥队尾元素3，3从队尾弹出
6. 待入队元素4<队尾元素5，4**入队**尾
7. 返回此时的队首5即为第二个ans
8. 待入队元素8≥队尾元素4，4从队尾弹出
9. 待入队元素8≥队尾元素5，5从队尾弹出
10. 队列空，8直接**入队**尾
11. **返回此时的队首8即为第三个ans**



单调队列

- 具体做法前面已经演示过，这里再总结一下

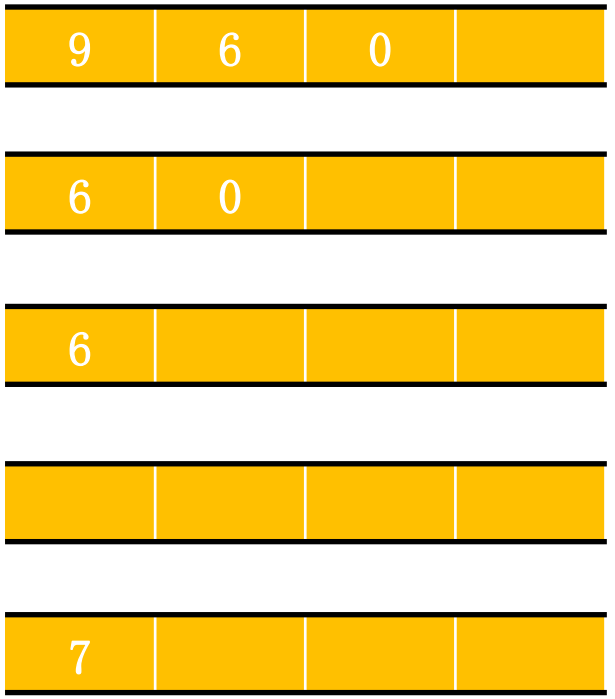
12. 待入队元素9≥队尾元素8，8从队尾弹出
13. 队列空，9直接入队尾
14. **返回此时的队首9即为第四个ans**
15. 待入队元素6<队尾元素9，6入队尾
16. **返回此时的队首9即为第五个ans**
17. 待入队元素0<队尾元素6，0入队尾
18. **返回此时的队首9即为第六个ans**



单调队列

- 具体做法前面已经演示过，这里再总结一下

19. 此时队列中已有3个元素，弹出队首9
20. 待入队元素7≥队尾元素0，0从队尾弹出
21. 待入队元素7≥队尾元素6，6从队尾弹出
22. 队列空，7直接入队尾
23. **返回此时的队首7即为第七个ans**



单调队列

- 具体做法前面已经演示过，这里再总结一下

24. 待入队元素3<队尾元素7，3**入队尾**

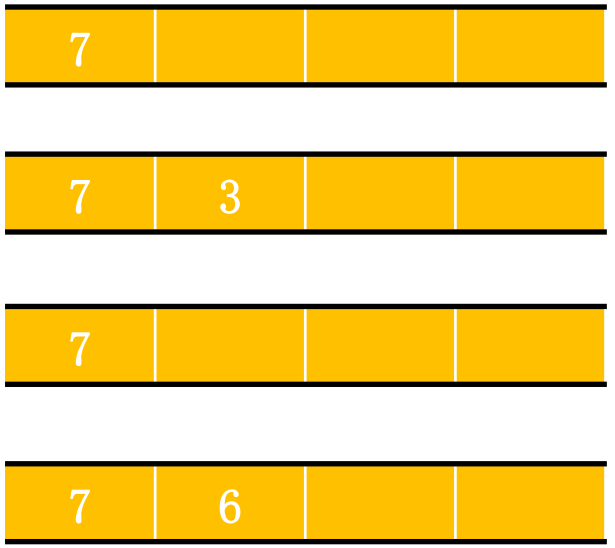
25. 返回此时的队首7即为第八个ans

26. 待入队元素6≥队尾元素3，3从队尾弹出

27. 待入队元素6<队尾元素7，6**入队尾**

28. 返回此时的队首7即为第九个ans

数列访问完毕



- 因为每个元素至多仅入队、出队一次，因此复杂度O(n)

单调队列

时刻	待入队	入队后	max
1	1	1	-
2	5	5	-
3	3	5，3	5
4	4	5，4	5
5	8	8	8
6	9	9	9
7	6	9，6	9
8	0	9，6，0	9
9	7	7	7
10	3	7，3	7
11	6	7，6	7

- 可以形成如图的表格

数列[1,5,3,4,8,9,6,0,7,3,6]
求max

k=3

单调队列

时刻	待入队	入队后	max
1	5	5	-
2	1	5，1	-
3	3	5，3	5
4	6	6	6
5	7	7	7
6	6	7，6	7
7	2	7，6，2	7
8	1	6，2，1	6

模仿写出：

数列[5,1,3,6,7,6,2,1]求max，k=3

单调队列

时刻	待入队	入队后	min
1	1	1	-
2	2	1，2	1
3	2	2	1
4	8	2，8	2
5	6	6	6

模仿写出：

数列[1,2,2,8,6]求min，k=2

单调队列

- 总结：

1. 维护一个单调递减的队列
2. 每当队首离当前位置差 ≥ k 则弹出队首
3. 每当当前元素 ≥ 队尾，则弹出队尾，然后将当前元素加入队尾
4. 此时以当前元素为右端点的长度为 k 的区间的最大值就是队首

单调队列的代码实现

- STL自带的双端队列deque，刚好可以用来实现单调队列

具体用法：

1. 定义：deque<int> q
2. 元素x入队尾 / 入队首：q.push_back(x) / q.push_front(x)
3. 返回队尾元素 / 队首元素：q.back() / q.front()
4. 队尾元素 / 队首元素出队：q.pop_back() / q.pop_front()
5. 队列大小：q.size()
6. 队列空：q.empty()

单调队列的代码实现

- 数组模拟实现单调队列

1. back -- 退队尾
2. front ++ 退队首
3. back ++ 入队尾
4. q[front].x 返回队首元素x
5. front <= back 队列不空

课外加练

- luogu 1886　　滑动窗口
- luogu 1440　　求m区间内的最小值