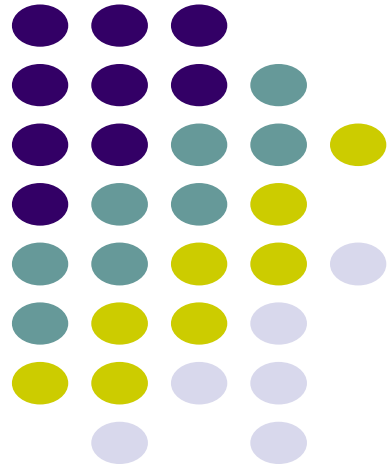
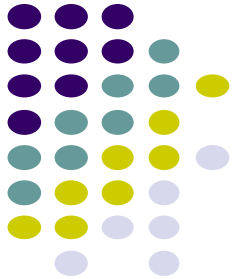
L4：队列

吉林大学计算机学院
谷方明
fmgu2002@sina.com



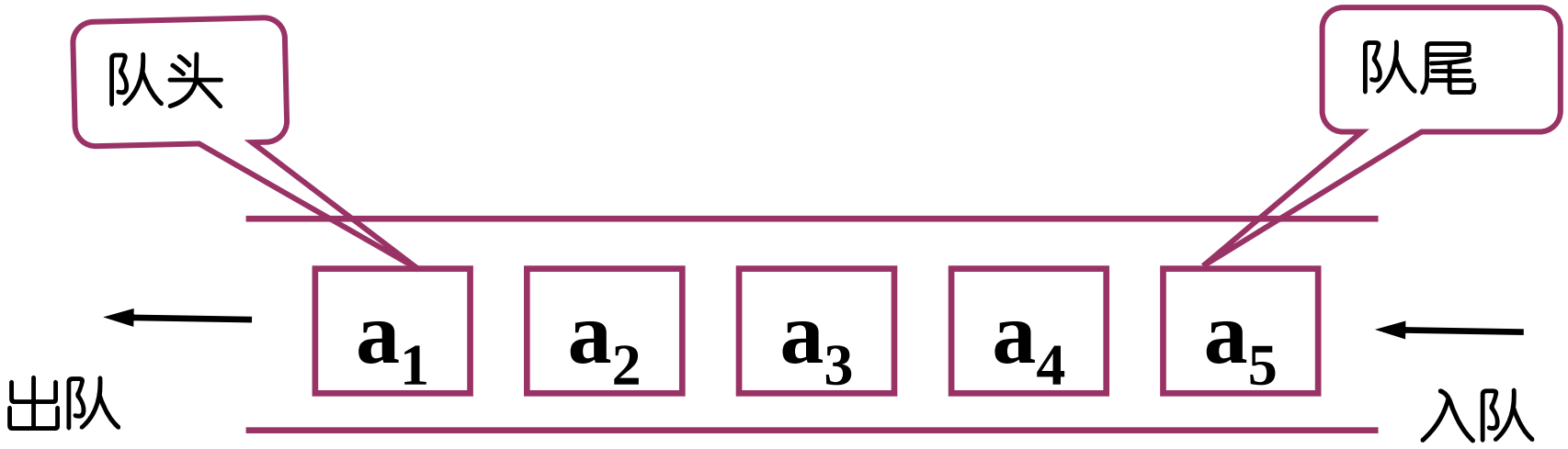
学习目标



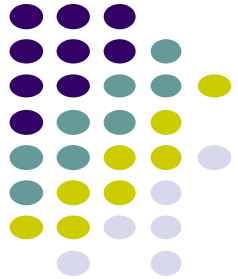
- 掌握队列的定义、特性和基本操作；
- 掌握队列的顺序存储方式及实现；能够解决顺序存储引起的假溢出问题
- 掌握队列的链接存储方式及实现；
- 初步应用队列解题
- 掌握单调队列、单调栈，了解栈和队列的其一些它拓展

引入

- 示例：排队



队列的定义



- 队列（**Queue**）是一种操作受限的线性表，所有插入都在表的一端进行，所有删除都在表的另一端进行。

术语

- 队头（front）：进行删除的一端；
- 队尾（rear）：进行插入的一端；
- 空队列：没有元素的队列。
- 入队：插入
- 出队：删除

队列的特性

- 先进先出 **(First In First Out，FIFO)**

- 队列也称先进先出表、**FIFO**表；

队列的基本操作

1. 入队
2. 出队
3. 取队首元素

4. 队列初始化
5. 判队列空
6. 判队列满
7. 清空队列

队列的顺序存储

- 按顺序存储方式存放队列元素，称为顺序队。

- 以整数队列为例

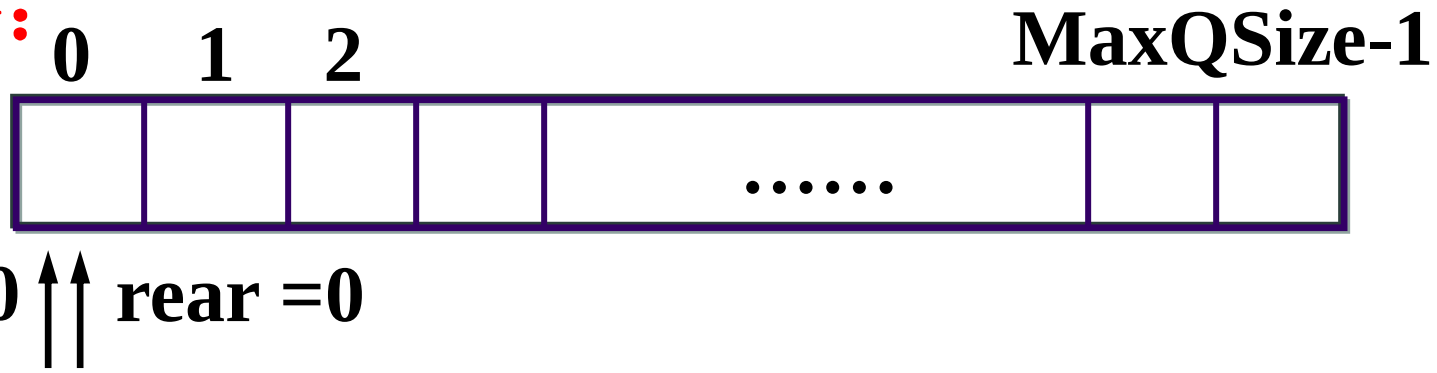
  数　　组：int que[MaxQSize]

  队首指针：front = 0

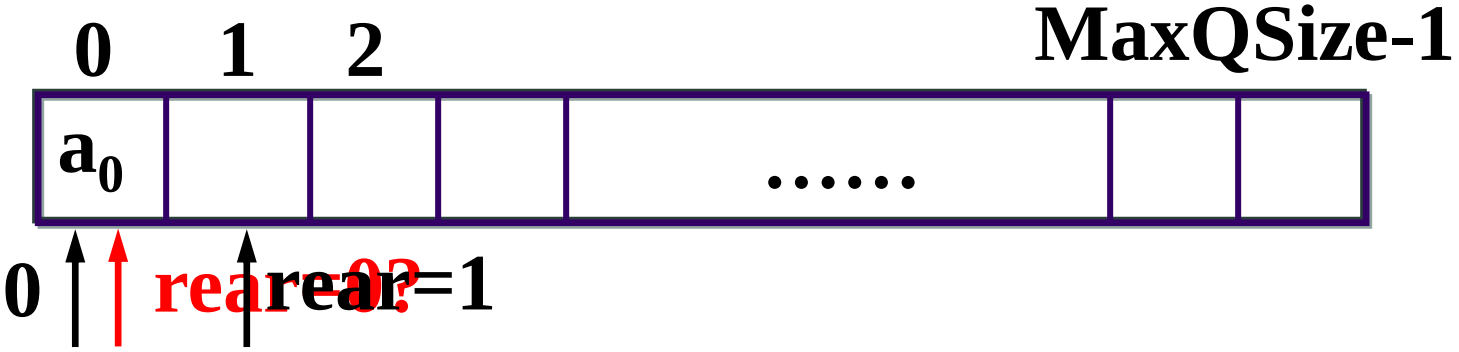
  队尾指针：rear = 0

插入操作模拟

初始状态：



a_0入队：



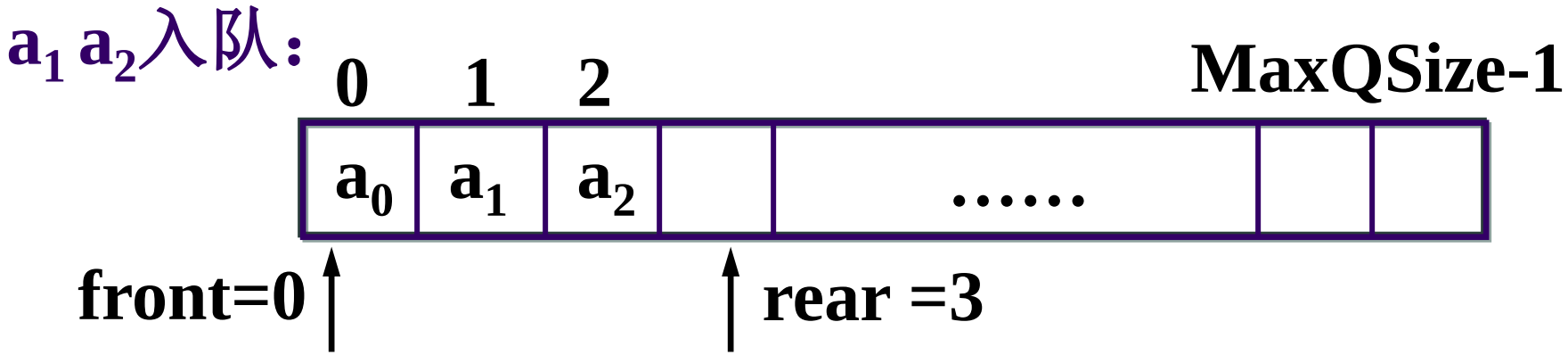
队列状态设置

- 队首指针： front 队首元素的下标
  队尾指针： rear 队尾元素的下标 加 1

- 队列空： **front == rear**
  队列满： **rear == MaxQSize**

插入操作

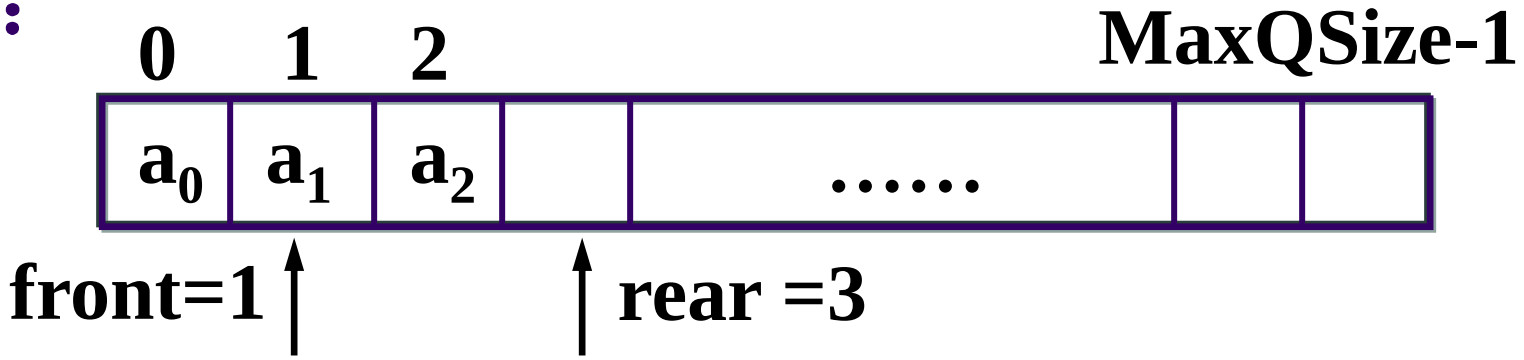
- 队尾插入元素：**rear=rear+1**



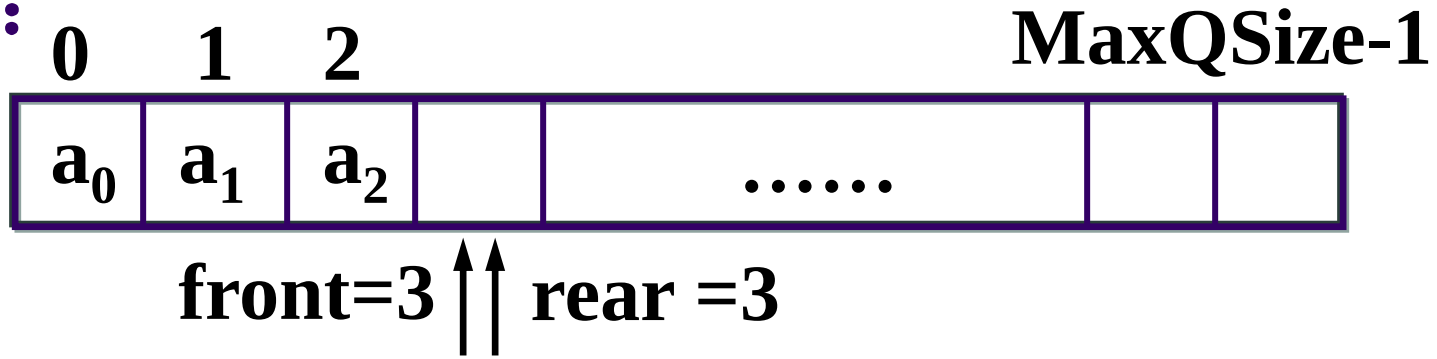
删除操作

- 保存队首元素；front=front+1

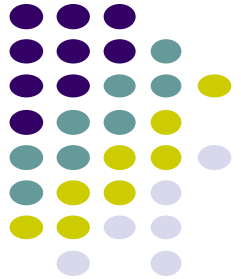
a_0出队：



a_1 a_2出队：



简单实现



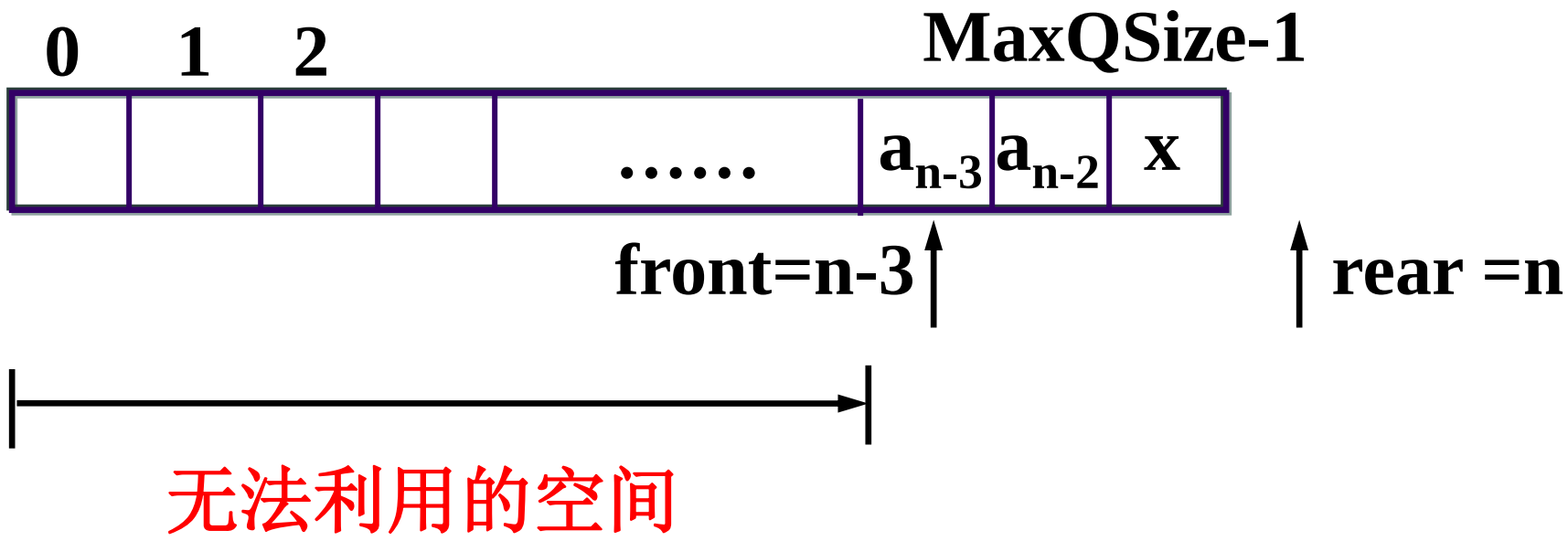
```
void qins(int x){
    que[rear++]=x;
}
int qdel(){
    return que[front++];
}
int getfront(){
    return que[front];
}
```

测试

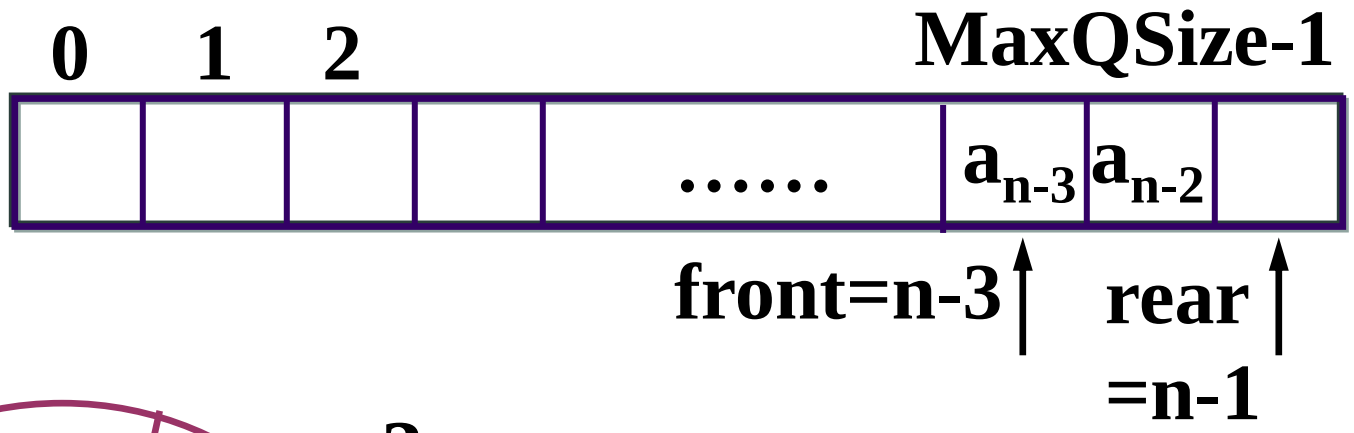
```
int main(){
    int n,i;
    scanf("%d",&n);

    for(i=1;i<=n;i++) qins(i);

    for(i=1;i<=n;i++) printf("%d ",qdel());
    printf("\n");
}
```

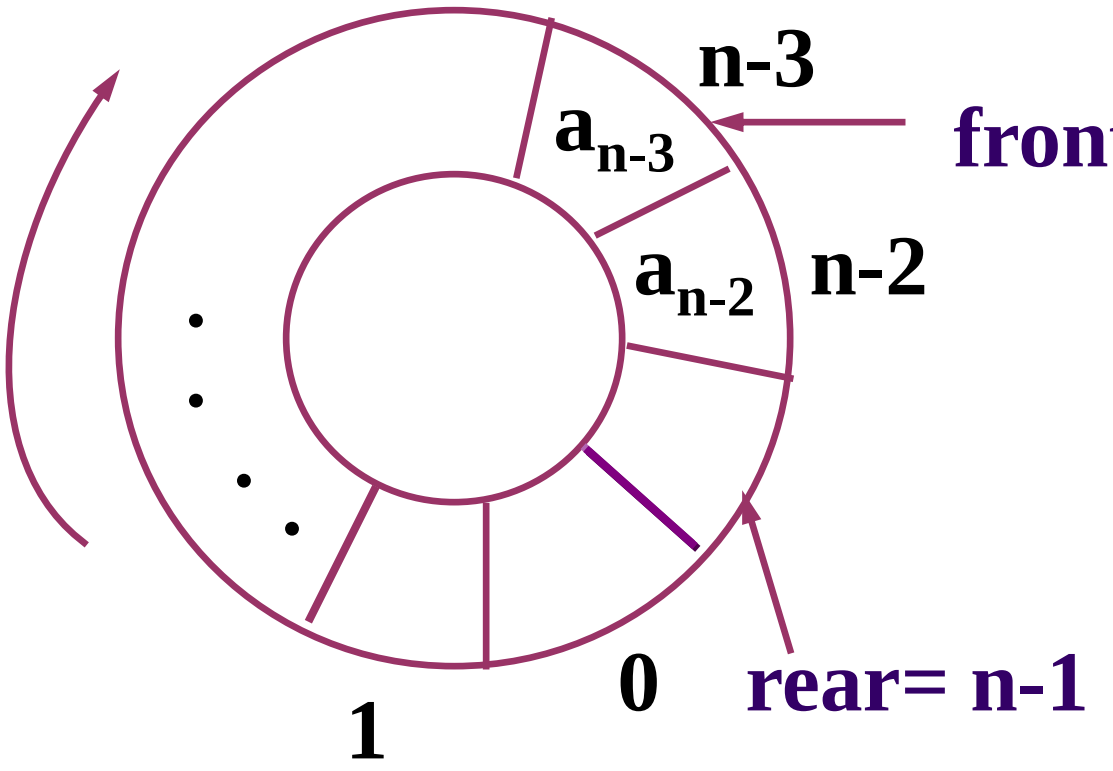
假溢出

- 溢出：空间不够用

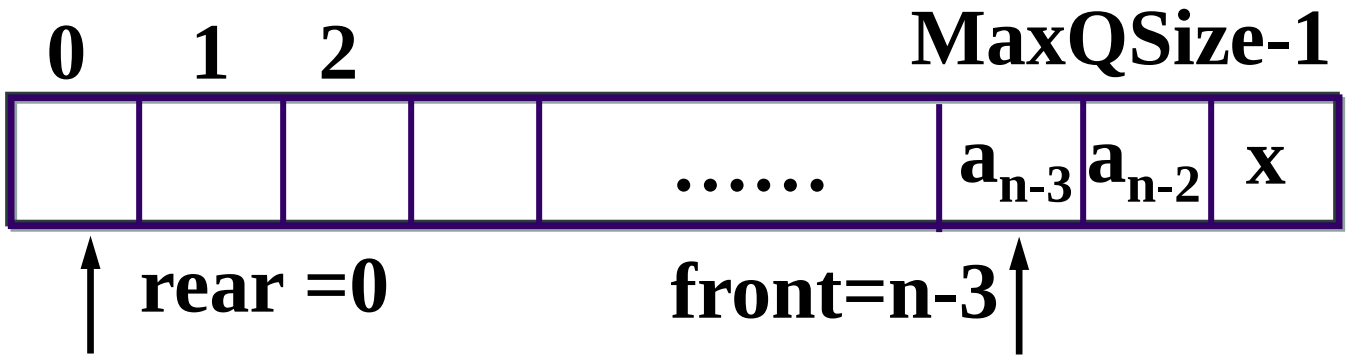


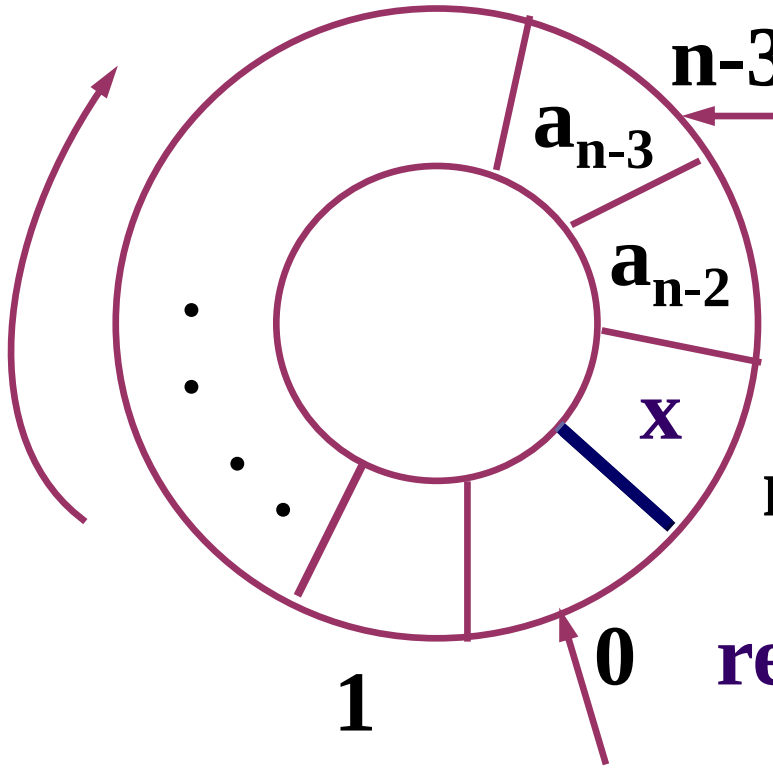
解决方案1


0
1
2
MaxQSize-1
……
a_{n-3}
a_{n-2}
front=n-3
rear
=n-1


n-3
a_{n-3}
front=n-3
a_{n-2}
n-2
0
rear= n-1
1

插入元素： rear顺时针移动一位



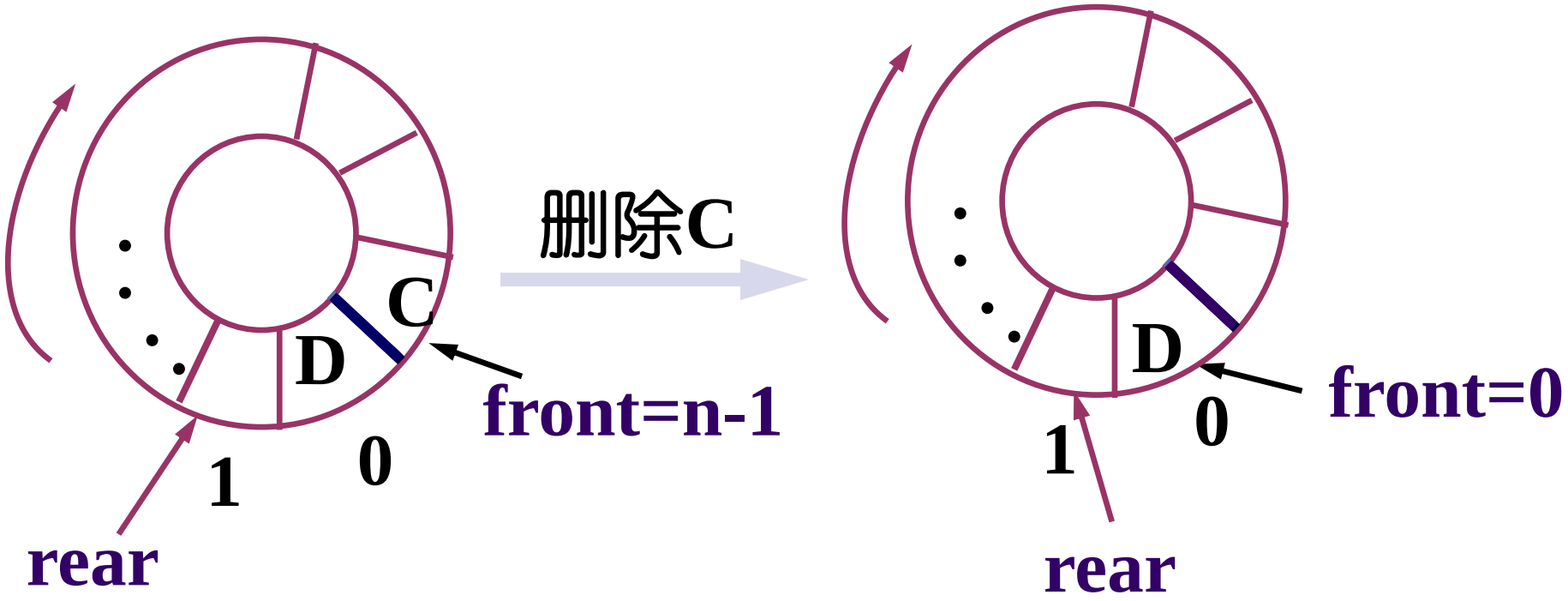


front=n-3

rear = (rear+1) % MaxQSize

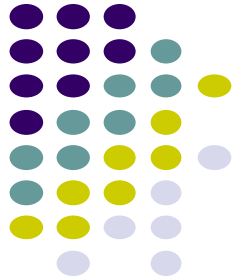
rear=0

删除元素：front顺时针移动一位



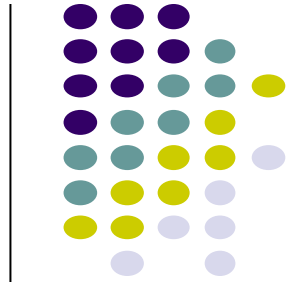
front = (front+1) % MaxQSize;

循环队列



- **front指向队首位置，删除一个元素就将front顺 时针移动一位；**
- **rear指向元素要插入的位置，插入一个元素就将rear顺时针移动一位；**
- **队列状态判断**
  - ✓ count方案：存放队列中元素的个数，当count等于MaxQSize时，不可再向队列中插入元素。

    队空：count = 0，队满：count = MaxQSize
  - ✓ 其它方案：课后思考，可以只用front和rear

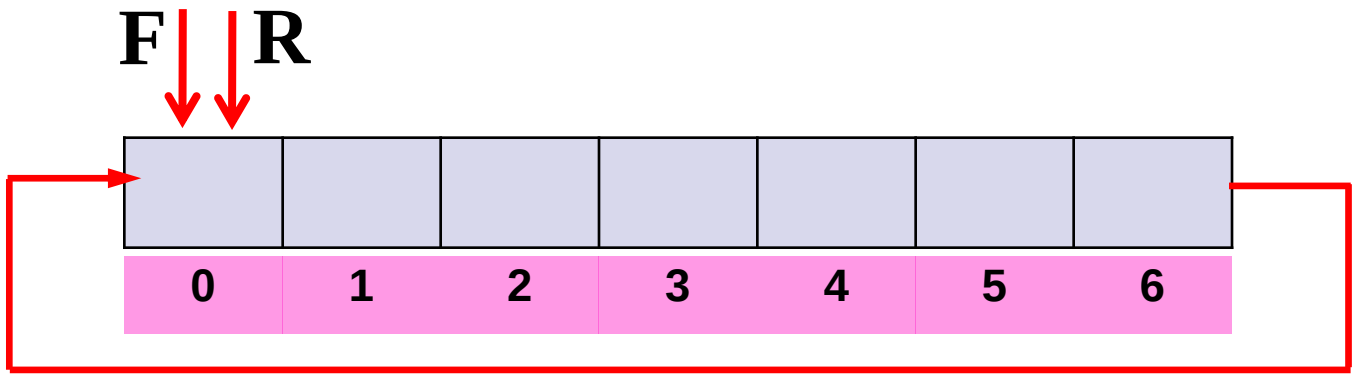


```
template<class T>
struct AQueue
{
    T QArray[MaxQSize];
    int front,rear,count;
    AQueue(){front=rear=count=0;}
    void qinsert(int x){
        QArray[rear]=x;
        rear=(rear+1)%MaxQSize;
    }
    T qdelete(){
        int x=QArray[front];
        front=(front+1)%MaxQSize;
        return x;
    }
};
```

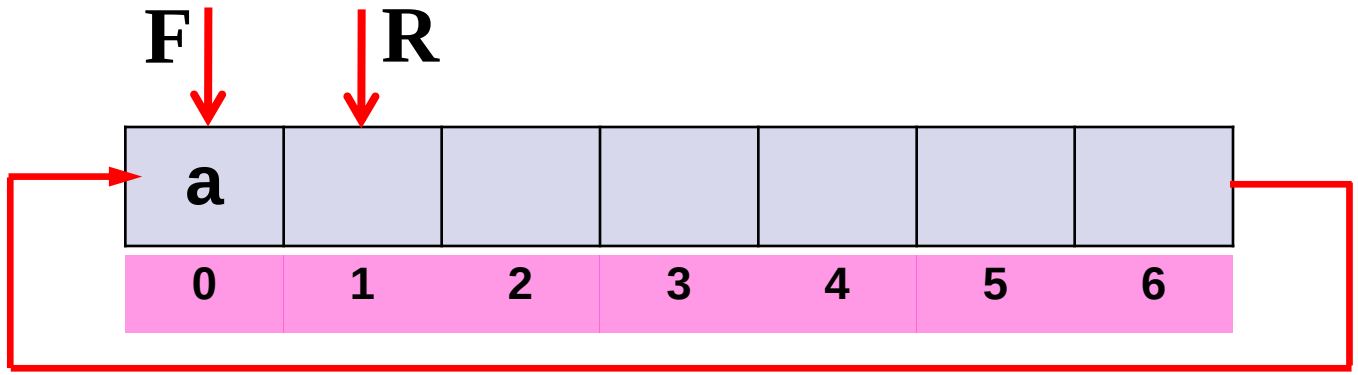
测试

```
int main(){
    int n,i;
    AQueue<char> q;
    scanf("%d",&n);

    for(i=1;i<=n;i++) q.qinsert(i+'a'-1);

    for(i=1;i<=n;i++) printf("%c ",q.qdelete());
    printf("\n");
}
```

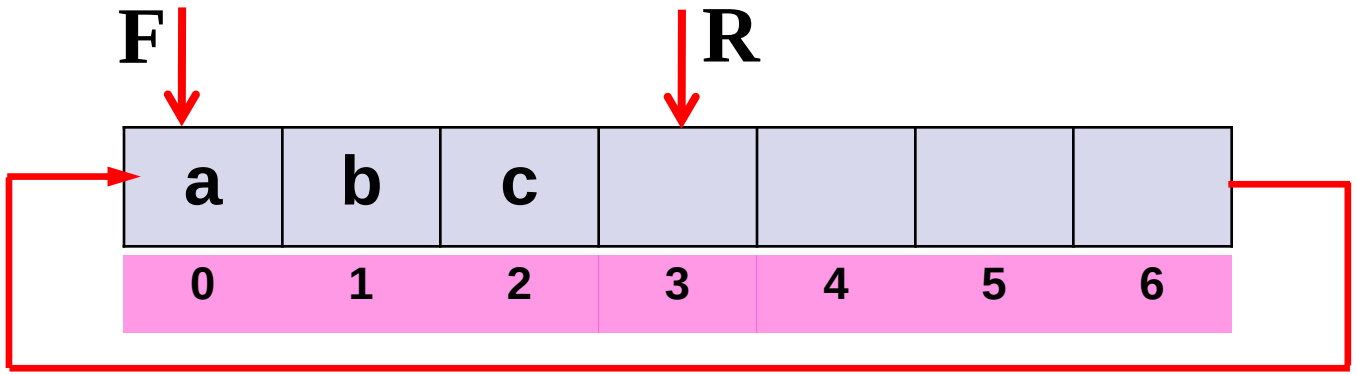
循环队列运行示意图



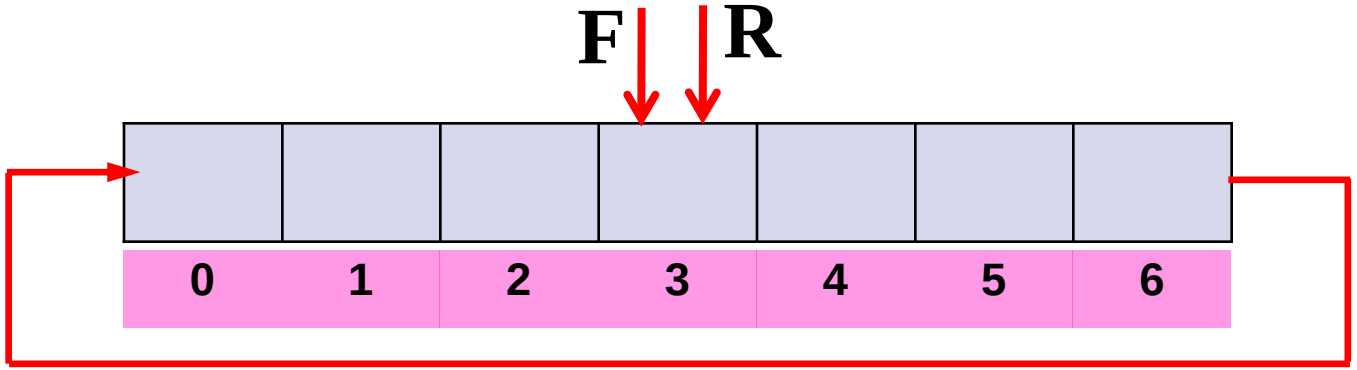
(a) 创建一个队列



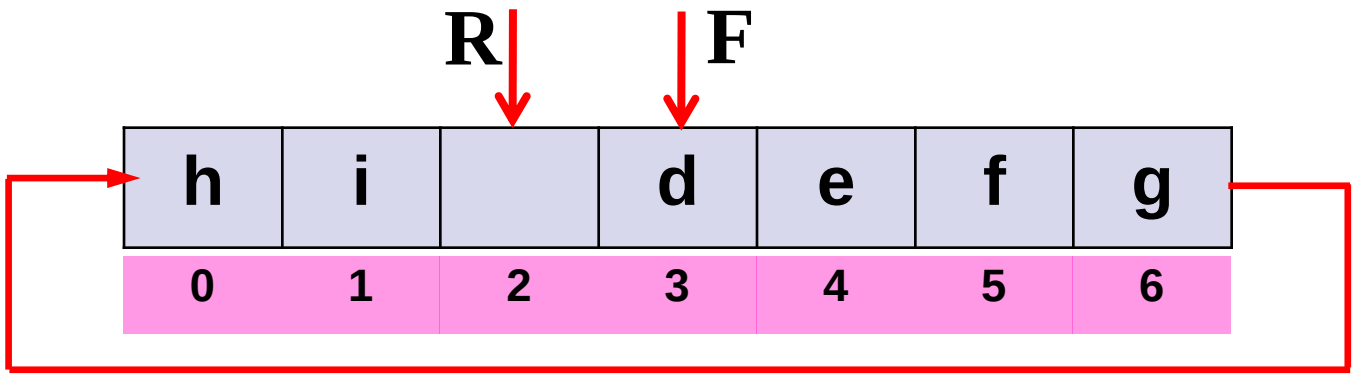
(b) 插入元素 a



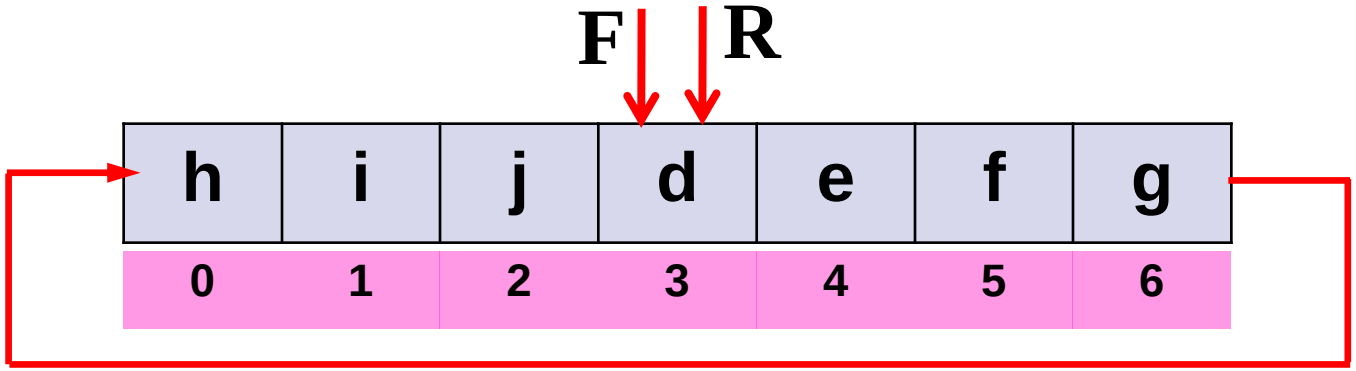
(c) 插入元素b、c



(d) 取出元素 a、b、c

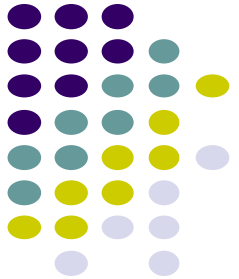


(e) 插入元素d、e、f、g、h、i



(f) 插入元素 j

假溢出小结

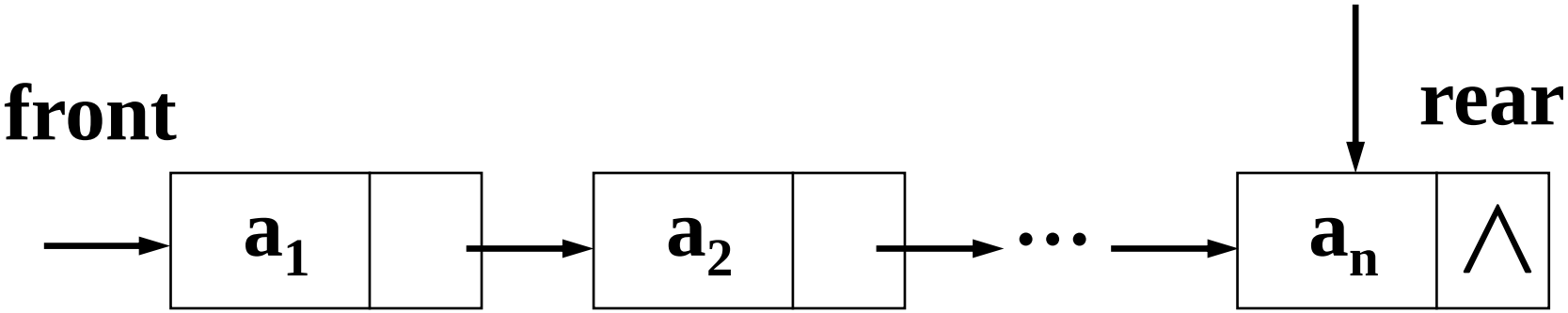


- 假溢出现象是队列的实现造成的。不是队列固有问题。

- 解决方案
  - ✓ 循环队列（推荐）
  - ✓ front固定为0 （教材提到，低效）
  - ✓ ……（课后思考）

队列的链接存储

- 用链接存储实现队列，要为每个元素分配一个额外的指针空间，指向后继结点。也称为链队。



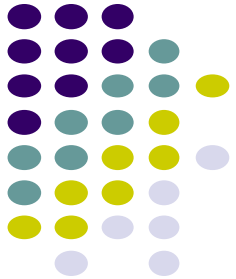
- 结点

```
template <class T>
struct Node{
   T data;
   Node* next;
   Node(){next=NULL;}
};
```

- 链队

```
template<class T>
struct LQueue{
   Node<T> * front,*rear;
   LQueue(){front=rear=NULL;}
   void qinsert(const T& item);
   void qdelete(T& item);
};
```

插入算法ADL描述



算法QInsert (item)

// 将元素item插入队尾

QI1. [创建新结点]

s⇐ AVAIL. data(s)←item. next(s)←NULL.

QI2. [队空？]

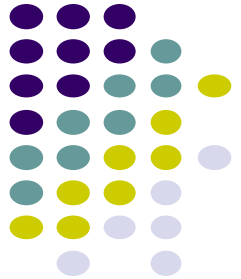
IF front=NULL THEN front←s.

ELSE next(rear)←s.

QI3. [更新队尾指针]

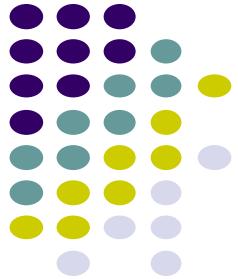
rear←s.

插入算法实现(模板，声明实现分离且放在同一文件中)



```
template <class T>
void LQueue<T>::qinsert(const T & item){
   Node<T> s=new Node<T> ; //malloc
   s->data=item;
   if(front) rear->next=s;else front=s;
   rear=s;
}
```

删除算法ADL描述



算法**QDelete (.item)**

// 删除队首结点并将其字段值存于item

QD1. [队列空？]

IF front=NULL

THEN (PRINT “队列为空”. RETURN.)

QD2. [出队]

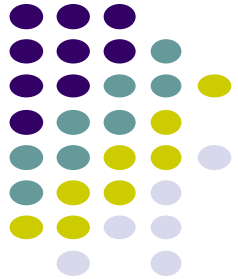
q←front. item←data(q). front←next(front).

AVAIL⇐q.

QD3. [出队后队列空？]

IF front=NULL THEN rear←NULL.

删除算法实现(模板)

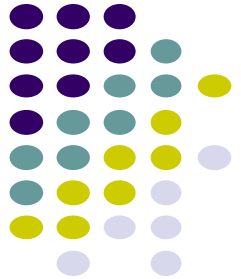


```
template <class T>
void LQueue<T>::qdelete(T& item){
   if(front==NULL) return;
   Node<T>* q=front;
   item=q->data;
   front=front->next;
   delete q; //free
   if(front==NULL)rear=NULL;
}
```

测试

```
int main(){
    int n,i,item;LQueue<int> q;
    scanf("%d",&n);

    for(i=1;i<=n;i++) q.qinsert(i);

    for(i=1;i<=n;i++){
        q.qdelete(item);
        printf("%d ",item);
    }
    printf("\n");
}
```

顺序队列与链式队列的比较



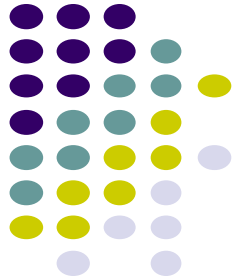
- 空间复杂度：顺序队列必须初始就申请固定的空间，当队列不满时，必然造成空间的浪费；链式队列所需空间是根据需要随时申请的，其代价是为每个元素提供空间以存储其next指针域。
- 时间复杂度，对于队列的基本操作（入队、出队和取队首），顺序队列和链式队列的时间复杂性均为$O(1)$.

队列的应用

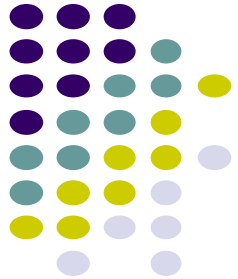
- 凡数据符合先进先出性的问题，可考虑用队列

- 经典应用
  - ✓ 任务调度；
  - ✓ 广度优先搜索；

例题：Blah数集



- **大数学家高斯小时候偶然发现一种有趣的自然数集合Blah。以a为基的集合Ba定义如下：**
  1. a是集合Ba的基，且a是Ba的第一个元素；
  2. 若x在集合Ba中，则2x+1和3x+1也都在Ba中；
  3. 没有其它元素在集合Ba中。
- **现在小高斯想知道如果将集合Ba中元素按照升序排列，第n个元素会是多少？**
- 时间限制:3000ms　　内存限制:65536kB



- 输入：多行，每行包括两个数，集合的基a(1<=a<=50))以及所求元素序号n(1<=n<=1000000)
- 输出：对于每个输入，输出集合Ba的第n个元素值
- 样例输入

  1 5

  28 5437
- 样例输出

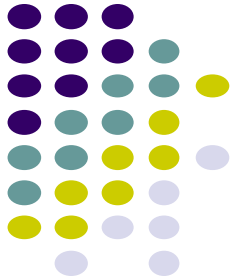
  9

  900585

分析

- **暴力模拟**
  - ✓ $O(n^2)$
- **标识数组(BFS)**
  - ✓ 空间越界
- **队列模拟**
  - ✓ 2x+1 和3x+1按生成次序分别构成1个递增数列；
  - ✓ 将两个队列放到两个容器中；
  - ✓ 每次从两个容器中分别取最小元素（**最先放入：队列**），作为当前元素。

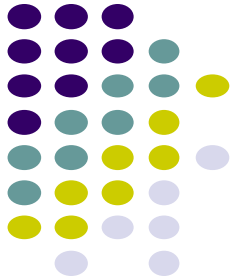
算法描述（自然语言）



1. 设x为基，初始化两个容器q2和q3。
2. q2放入数2x+1，q3放入数3x+1;
3. 设x2、x3分别是q2和q3的首元素，令x=min(x2,x3)；（具体有三种情况： x2 > x3 、x2 = x3 和 x2 < x3 ）
4. 重复2、3两步，直至取出第n项为止。

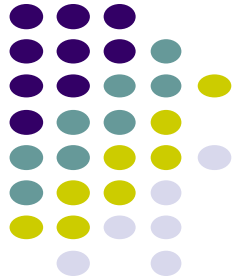
- 效率分析： 时间O(n)，空间O(n)

STL中的队列queue



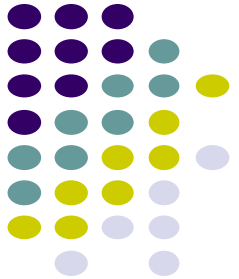
- #include <queue>
- queue<int > q; //不用声明大小
- q.push(x); //队尾压入
- q.pop();　　//弹出队首，与front合用
- q.front();　//取队首
- q.back();　//取队尾
- q.empty(); //判空
- q.size();　//队列长度

STL中的双端队列deque



- #include <deque>
- deque<int > dq; //不用声明大小
- dq.push_back(x); //队尾插入
- dq.pop_back();   //队尾删除
- dq.push_front(x); //队头插入
- dq.pop_front(); //队头删除
- dq.front(); //取队首
- dq.back(); //取队尾

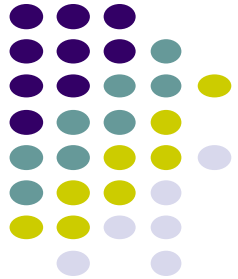
双端队列



- 双端队列（**Double-ended queue**，简称为**Deque**）也是一种操作受限的线性表，插入和删除只能发生在表的两端。

- 可用双端队列去模拟栈和队列，但效率略低。
- **STL**中**stack**和**queue**的底层容器默认是**deque.**

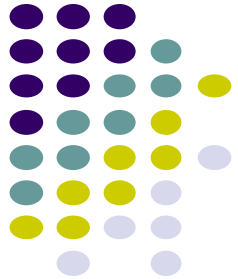
单调队列



- **单调队列是一种特殊的队列，队列中的元素保持单调递增 或单调递减。**

- **利用单调性求极值。**
  - ✓ 单调队列的队首元素是连续一段数的最值（极值）。

例



- n个数的数列，从左至右输出每个长度为m的区间内的最小值。
- 样例输入

  6 3

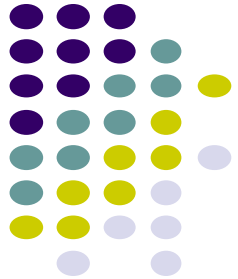
  1 2 5 3 4 6
- 样例输出

  1 2 3 3

解法1：暴力模拟

- 每次对以i结尾长度为m的区间，扫描求最小值
- O(n*m)

解法2：单调队列方法



- 维护一个区间正确且单调递增的队列，每次队首就是当前区间的最值
  - ✓ 单调性维护：当前元素和队尾元素比较，若队尾元素大于当前元素，队尾元素出队；重复处理，直到队空或队尾元素小于当前元素；当前元素入队；
  - ✓ 区间维护：若队首元素的位置不在区间内，则队首元素出队；重复处理，直到队首元素在区间内（队列里保存下标方便处理）

参考代码

- 单调性维护

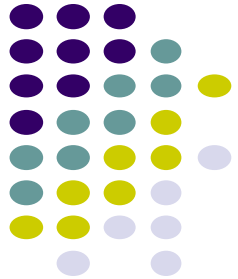
```
// f队首，r队尾后空位
while(f<r && a[que[r-1]] > a[i] ) r--;
que[r++] = i;
```

- 区间维护

```
while( f<r && que[f] < i-m+1 ) f++;
```

- 当前区间最值 ： a[que[f]]

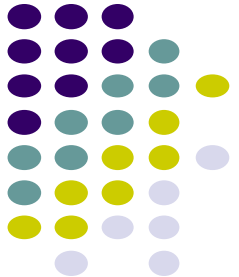
效率分析和小结



- **n-m+1个取区间最值的总代价为O(n)**
  - ✓ 维护代价：每个数据最多入队1次、出队1次，n-m+1次的维护的总代价最多2n次队列操作，O(n)。
  - ✓ 取最值：一次区间最值代价：O(1)，n-m+1次取最值得总代价为O(n)

- **相比其它方法，单调队列处理此类问题较优。**
- **单调队列还有一些其它应用，如优化动态规划**
- **单调队列也可用deque实现**

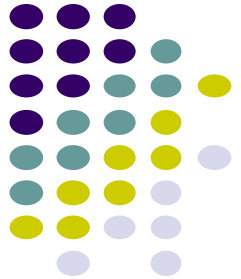
单调栈



- 单调栈是一种特殊的栈，栈中的元素保持单调递增 或 单调递减。

- 利用单调性，取当前元素左（右）边的第一个比它小（大）的元素比较方便。

例



- **n个正数的数列**，询问每个数左边的第一个比它小的数。如果不存在，输出-1。
- **样例输入**

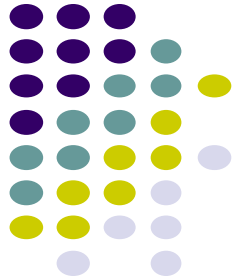
  6

  1 2 5 3 4 6
- **样例输出**

  -1 1 2 2 3 4

解法1：暴力模拟

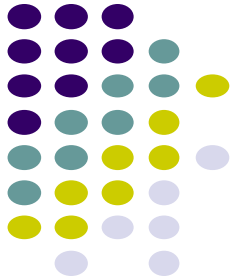
- $O(n^2)$



解法2：单调栈

- 维护一个单调递增的栈，使用当前元素维护单调性后，栈顶即为所求
  - ✓ 单调性维护：当前元素和栈顶元素比较，若栈顶元素大于当前元素，栈顶元素出栈；重复处理，直到栈空或栈顶元素小于等于当前元素；
  - ✓ 若栈不空，栈顶元素即为所求；否则，不存在
  - ✓ 当前元素入栈；

实现和分析



- 单调性维护

  while(top!=-1 && s[top] > a[i]) top--;

- 效率分析
  - ✓ 每个数据最多入栈1次、出栈1次，n次维护最多2n次。n次求最值操作的总代价为O(n)，每次求值操作的平均(摊还)代价为O(1)

- 单调栈有很多神奇的应用，需要在应用中发现

其它栈和队列

- 双栈：两个底（顶）部相连的栈

- 根据实际需要，设计新的栈和队列。

总结

- 队列的定义、特性和基本操作；
- 队列的顺序存储方式及实现（顺序队）；
  - ✓ 假溢出问题：一种解决方案是循环队列
- 队列的链接存储方式及实现（链队）；
- 队列的应用
  - ✓ 容器、FIFO
- 队列和栈的拓展
  - ✓ 双端队列、单调队列等
  - ✓ 单调栈、双栈等