数据结构与算法

第3章 栈与队列

主讲：赵海燕
北京大学信息科学技术学院
“数据结构与算法”教学组

国家精品课“数据结构与算法”
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

张铭，王腾蛟，赵海燕
高等教育出版社，2008. 6, “十一五”国家级规划教材

操作受限的线性表

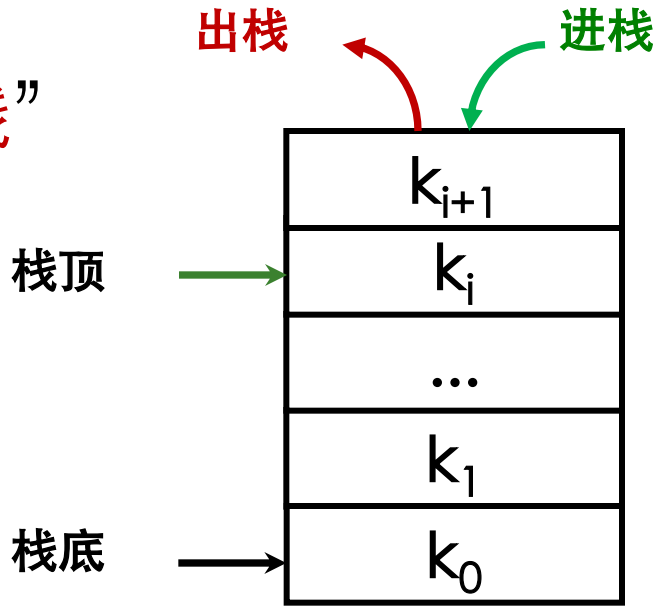
- 栈（Stack）
  - 运算只在表的一端进行
- 队列（Queue）
  - 运算只在表的两端进行

栈

- 后进先出（LastInFirstOut）
  - 一种限制访问端口的线性表
  - 栈存储和删除元素的顺序与元素插入顺序相反
  - 也称为“下推表”
- 栈的主要元素
  - **栈顶**（top）元素：栈的唯一可访问元素
    - 元素插入栈称为“入栈”或“压栈”（push）
    - 删除元素称为“出栈”或“弹出”（pop）
  - 栈底：另一端

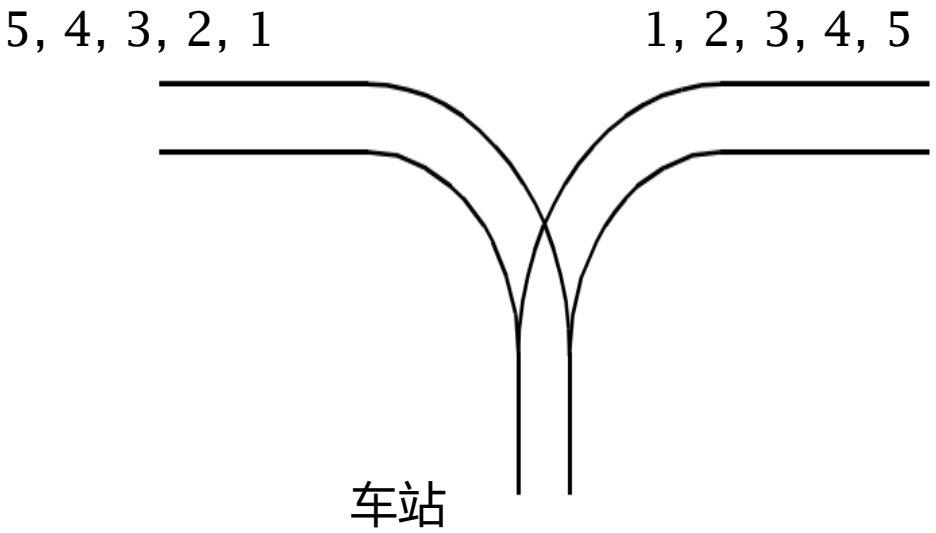
栈的图示

- 每次取出（并被删除）的总是刚压进的元素，而最先压入的元素则被放在栈的底部
- 当栈中没有元素时称为“空栈”
- 应用
  - 表达式求值
  - 消除递归
  - 深度优先搜索



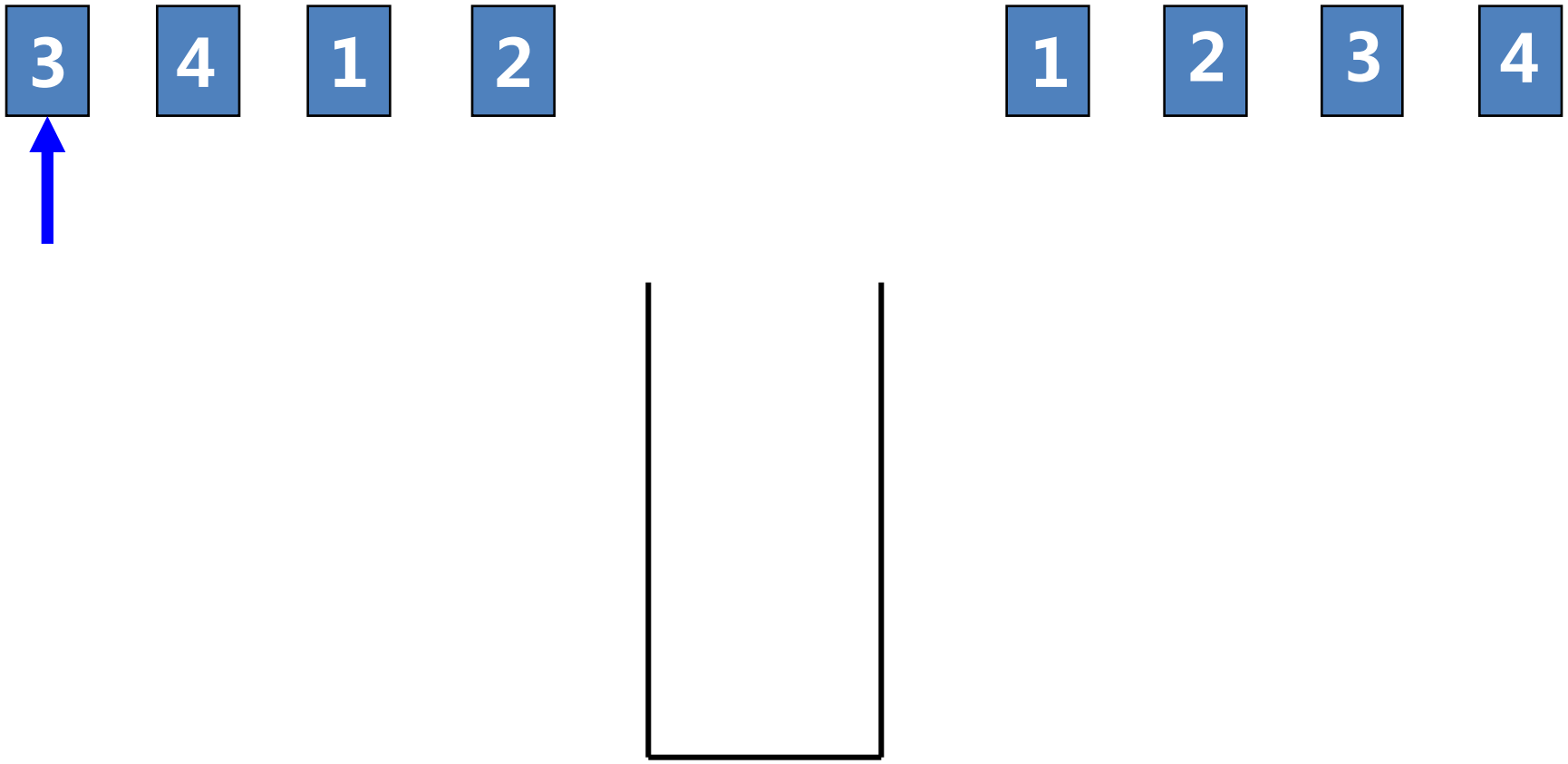
示例：火车进出栈问题

- 判断火车的出栈顺序是否合法
  - http://poj.org/problem?id=1363
- 编号为1, 2, …, n的 *n* 辆火车依次进站，给定一个 *n* 的排列，判断是否为合法的出站顺序?



示例：火车进出栈问题

- 利用 合法的重构 发现 冲突



思考

- 若入栈的顺序为1, 2, 3, 4 的话，则出栈的顺序可以有哪些？
- 从初始输入序列1, 2, …, n，希望利用一个栈得到输出序列$p_1, p_2, \ldots, p_n$ (1， 2，…，n的一种排列)。若存在下标i，j，k，满足i < j < k 同时 $p_j<p_k<p_i$，则输出序列是否合法?

栈的抽象数据类型

```
template <class T>
class Stack {
public:                          // 栈的运算集
    void clear();                // 变为空栈
    bool push(const T item);
                                 // item入栈，成功返回真，否则假
    bool pop(T& item);           // 返回栈顶内容并弹出，成功返回真

    bool top(T& item);           // 返回栈顶但不弹出，成功返回真，否则假
    bool isEmpty();              // 若栈已空返回真
    bool isFull();               // 若栈已满返回真
};
```

栈的实现方式

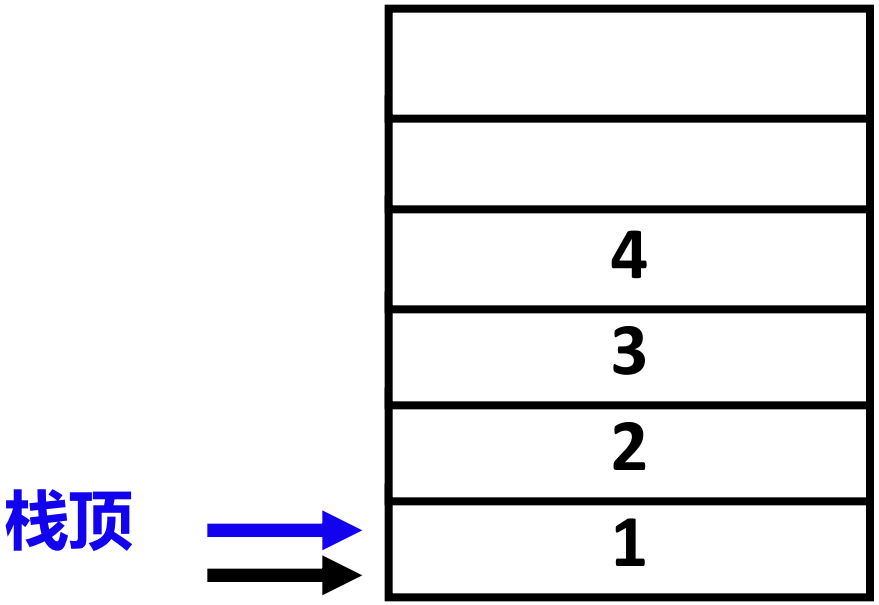
- 顺序栈 (Array-based Stack)
  - 采用向量实现，本质上顺序表的简化版
    - 栈的大小
  - 关键：确定哪一端作为栈顶
  - 上溢/下溢问题
- 链式栈（Linked Stack）
  - 采用单链表方式存储，指针的方向是从栈顶向下链接

顺序栈的类定义

```
template <class T> class arrStack : public Stack <T> {
private:                          // 栈的顺序存储
    int mSize;                    // 栈中最多可存放的元素个数
    int top;                      // 栈顶位置，应小于mSize
    T *st;                        // 存放栈元素的数组
public:                           // 栈的运算的顺序实现
    arrStack(int size) {          // 创建一个给定长度的顺序栈实例
        mSize = size; top = -1; st = new T[mSize];
    }
    arrStack() {                  // 创建一个顺序栈的实例
        top = -1;
    }
    ~arrStack() { delete [] st; }
    void clear() { top = -1; }    // 清空栈
}
```

顺序栈示例

- 按压入先后次序，最后压入的元素编号为4，然后依次为3，2，1



顺序栈的溢出

- 上溢（Overflow）
  - 栈中已有maxsize个元素时，再做进栈运算时产生的现象
- 下溢（Underflow）
  - 对空栈进行出栈运算时所产生的现象

压栈操作

```
bool arrStack<T>::push(const T item) {
    if (top == mSize-1) {  // 栈已满
        cout << "栈满溢出" << endl;
        return false;
    } else {                    // 新元素入栈并修改栈顶指针
        st[++top] = item;
        return true;
    }
}
```

出栈操作

```
bool arrStack<T>::pop(T & item) {          // 出栈
    if (top == -1) {                       // 栈为空
        cout << "栈为空，不能执行出栈操作"<< endl;
        return false;
    } else {
        item = st[top--];                  // 返回栈顶，并缩减1
        return true;
    }
}
```

读栈操作

```
bool arrStack<T>:: top(T & item) {
                                    // 返回栈顶内容，但不弹出
    if (top == -1) {                // 栈空
        cout << " 栈为空，不能读取栈顶元素"<< endl;
        return false;
    }
    else {
        item = st[top];
        return true;
    }
}
```

其他操作

- 清空栈

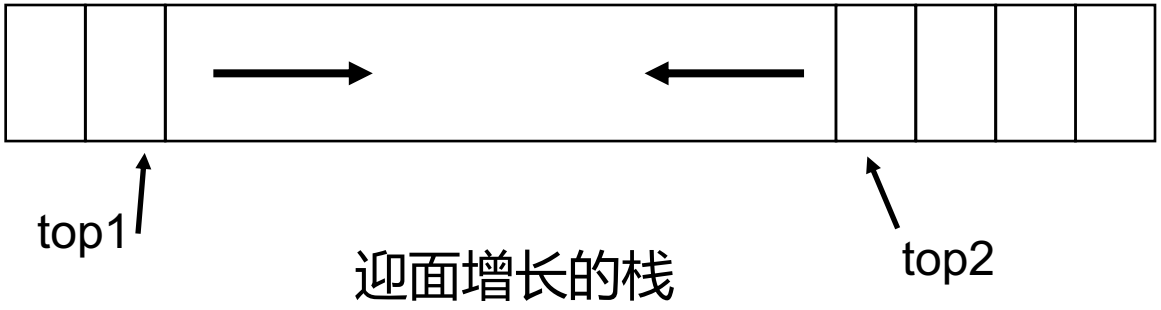
```
void  arrStack<T>::clear() {
        top = -1;
}
```

- 判断栈满与否

```
bool arrStack<T>:: isFull() {
        return  (top == maxsize-1) ;  // 栈满时返回非零值（真值true）
}
```

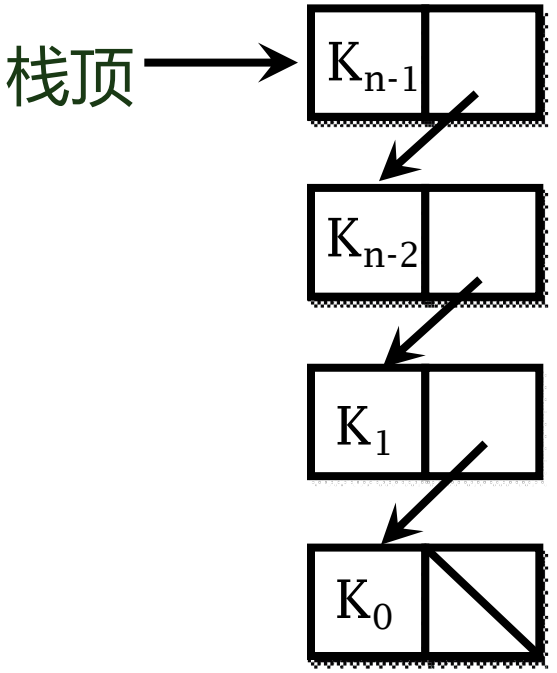
栈的变种

- 两个独立的栈
  - 底部相连：双栈
  - 迎面增长
  - 哪一种更好些?



链式栈

- 采用单链表
- 指针的方向从栈顶向下链接



链式栈的创建

```
template <class T> class lnkStack : public Stack <T> {
private:                                    // 栈的链式存储
    Link<T>* top;                           // 指向栈顶的指针
    int size;                               // 存放元素的个数
public:                                     // 栈运算的链式实现
    lnkStack(int defSize) {                 // 构造函数
        top = NULL; size = 0;
    }
    ~lnkStack() {                           // 析构函数
        clear();
    }
}
```

压栈操作

```
// 入栈操作的链式实现
bool lnksStack<T>:: push(const T item) {
    Link<T>* tmp = new Link<T>(item, top);
    top = tmp;
    size++;
    return true;
}

Link(const T info, Link* nextValue)  {
                    // 具有两个参数的Link构造函数
        data = info;
        next = nextValue;
}
```

出栈操作

```
// 出栈操作的链式实现
bool lnkStack<T>:: pop(T& item) {
    Link <T> *tmp;
    if (size == 0) {
        cout << "栈为空，不能执行出栈操作"<< endl;
        return false;
    }
    item = top->data;
    tmp = top->next;
    delete top;
    top = tmp;
    size--;
    return true;
}
```

顺序栈 PK 链式栈

- 时间效率
  - 入栈/出栈操作均只需 常数时间
  - 顺序栈和链式栈在时间效率上难分伯仲
- 空间效率
  - 顺序栈须事先确定一个固定长度
  - 链式栈的长度可变，但增加结构性开销

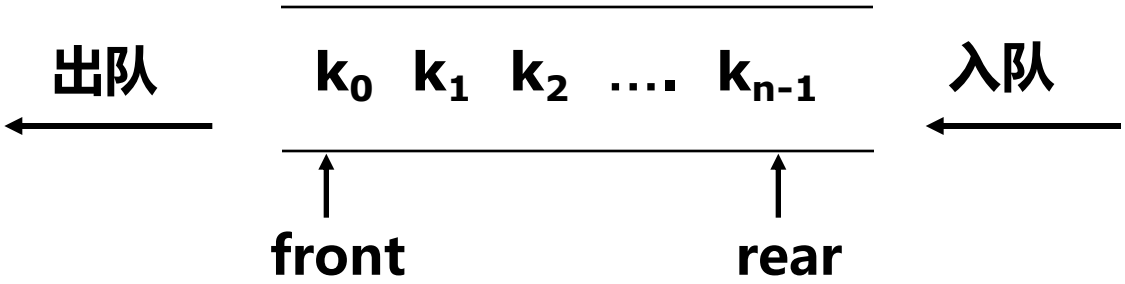
顺序栈 PK 链式栈

- **实际应用中，顺序栈 更广泛些**
  - 存储开销低
  - 易于根据栈顶位置进行相对位移，快速定位并读取栈的内部元素
    - 顺序栈 读取内部元素 的时间为O(1)，而链式栈则需要沿指针链游走，读取第 k个元素需要时间为O(k)
    - 一般来说，栈不允许“读取内部元素”，只能在栈顶操作

栈的应用

- 栈的特点：后进先出
  - 体现了元素间的透明性
- 常用来处理具有递归结构的数据
  - 深度优先搜索
  - 数制转换
  - 表达式求值
  - 行编辑处理
  - 子程序／函数调用的管理
  - 消除递归

队列



- **先进先出（FirstInFirstOut）**
  - 按照到达的顺序来释放元素
  - 限制访问点的线性表
    - 所有的插入在表的一端进行，所有的删除都在表的另一端进行
    - 特例： 空队列
- **主要元素**
  - 队头（front）： 允许删除的一端
  - 队尾（rear）： 允许插入的一端

队列的主要操作

- 入队（enQueue）（插入）
- 出队（deQueue）（删除）
- 取队首（getFront）
- 判断队列是否为空（isEmpty）

队列的抽象数据类型

```
template <class T> class Queue {
public:                                  // 队列的运算集
   void clear();                         // 变为空队列
   bool enQueue(const T item);
                 // 将item插入队尾，成功则返回真，否则返回假
   bool deQueue(T & item) ;
                 // 返回队头元素并将其从队列中删除，成功则返回真
   bool getFront(T & item)
                 // 返回队头元素，但不删除，成功则返回真
   bool isEmpty();     // 返回真，若队列已空
   bool isFull();      // 返回真，若队列已满
};
```

队列的实现方式

- **顺序队列**
  - 关键： 如何防止 **假溢出**
- **链式队列**
  - 用单链表方式存储，队列中每个元素对于链表中的一个结点

队列的溢出

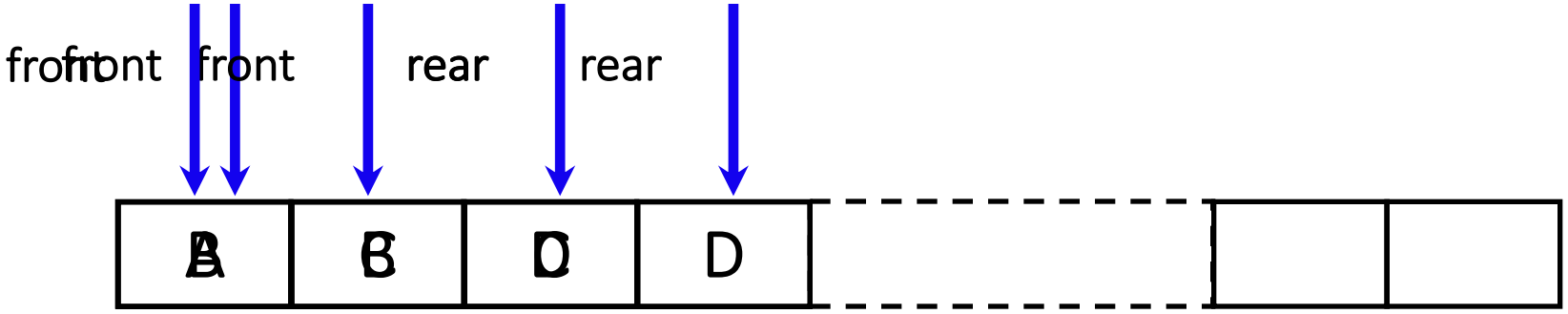
- **上溢**
  - 当队列满时，入队操作所出现的现象
- **下溢**
  - 当队列空时，出队操作所出现的现象
- **假溢出**
  - 顺序队列可能出现的一种现象：当队尾指针达到最大值（rear = MAXNUM）时，再作入队运算就会产生溢出，但可能此时队列前端尚有可用的位置

顺序队列

- 使用顺序表来实现队列
- 用数组存储队列元素，并用两个变量分别指向队列的队头（前端）和队尾（尾端）
  - front
  - rear

顺序队列的维护

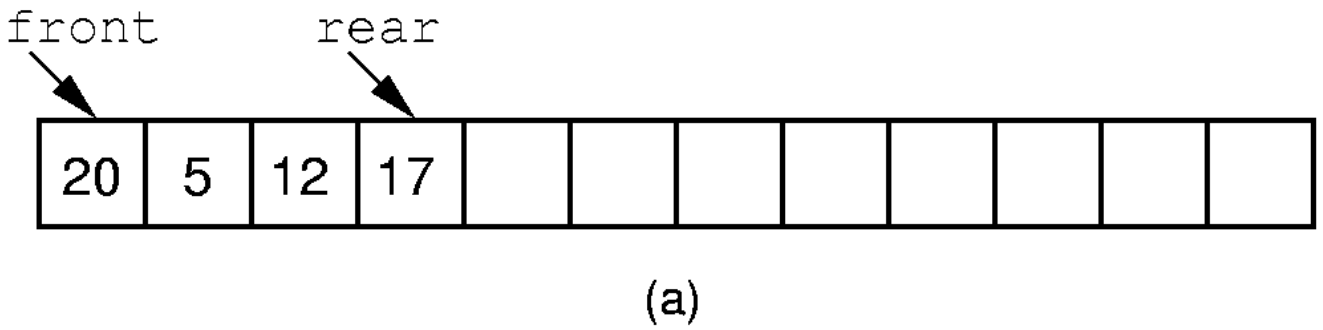
- rear实指



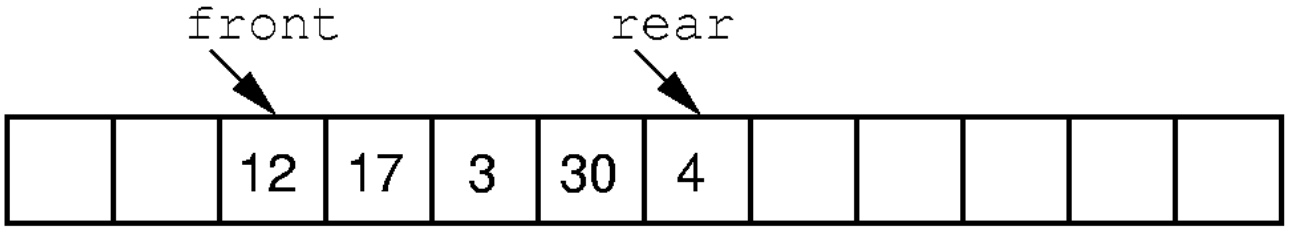
删除

顺序队列的维护

- front和rear都实指

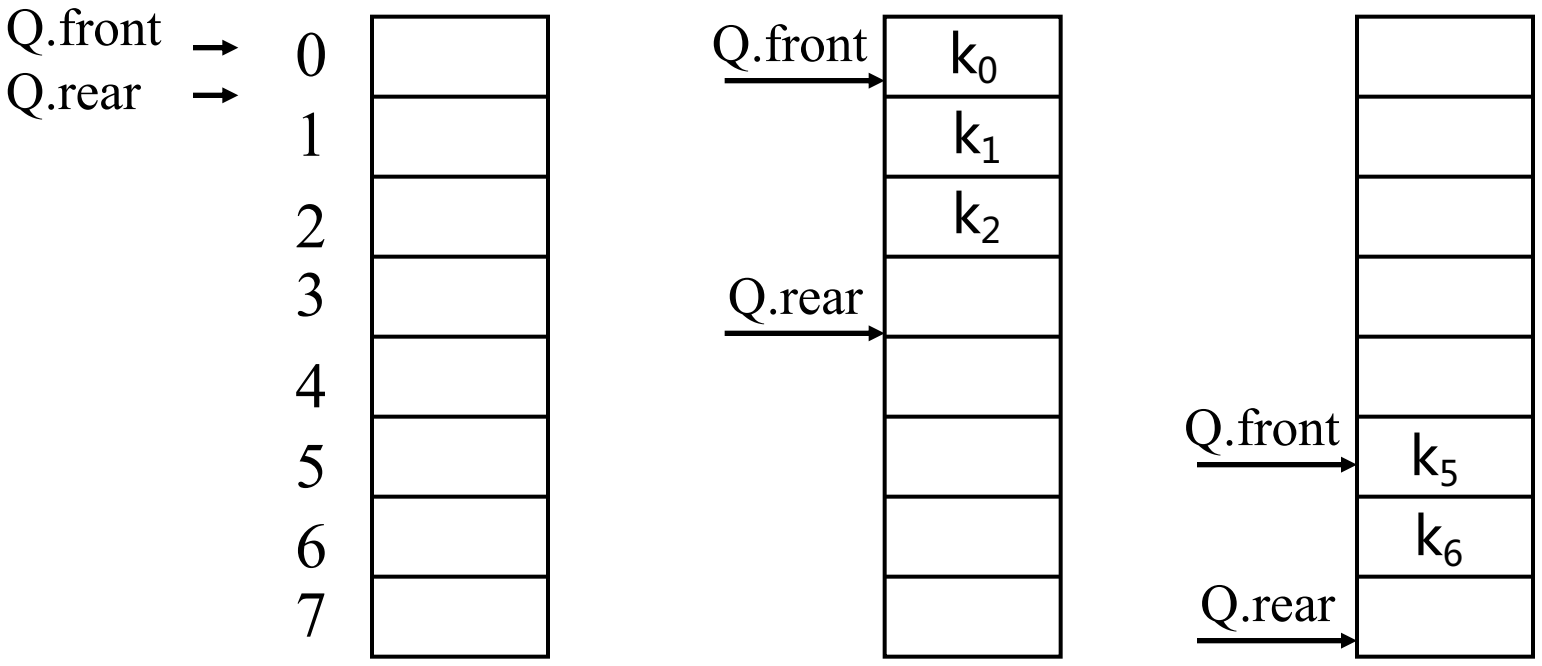


(a)



(b)

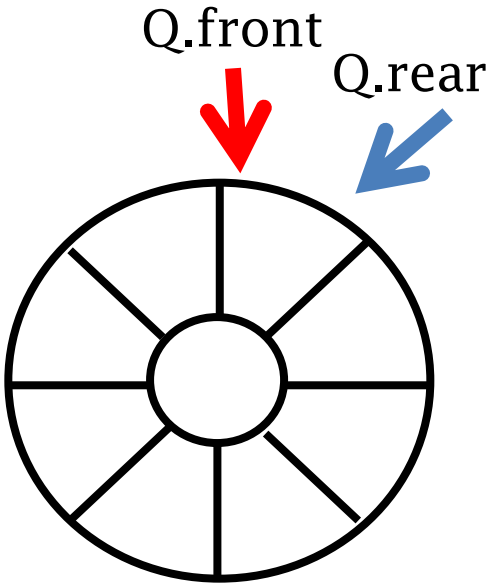
队列示意：普通



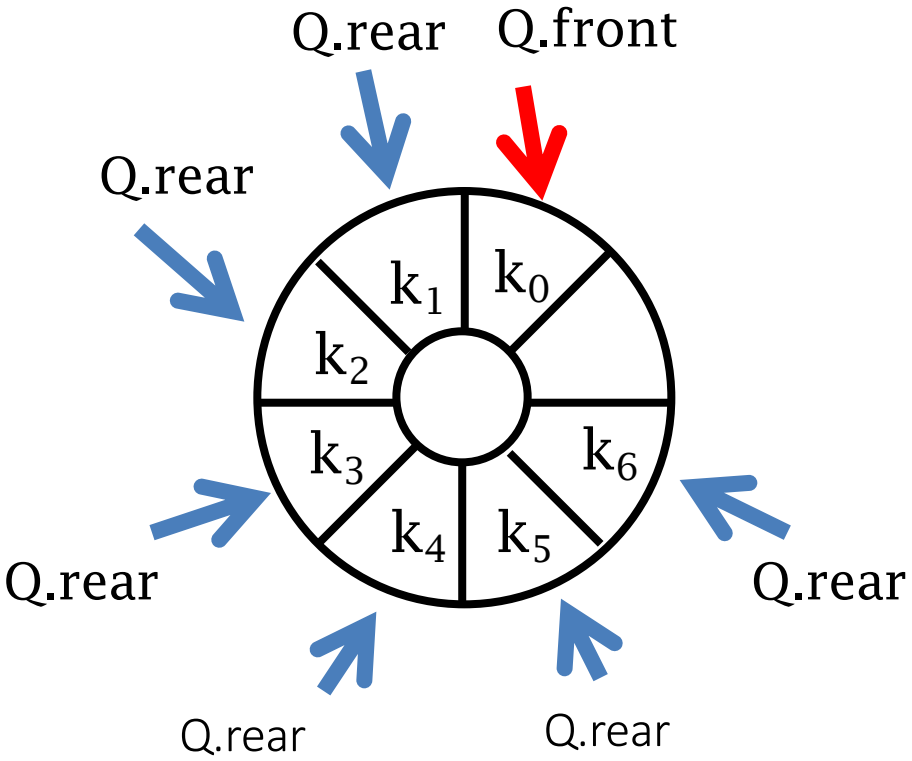
队列空

再进队一个元素如何?

队列示意：环形



空队列



队列空：

Q.rear == Q.front

队列满：

(Q.rear+1) mod M == Q.front

思考

1. 只是用 front, rear 两个变量，长度为mSize = n 的队列，可以容纳的最大元素个数为多少？ 请给出详细的推导过程。
2. 如果不愿意浪费队列的存储单元，还可以采用什么方法?
3. 采用实指和虚指方法实现队尾指针(rear指向队尾元素后一个元素，和实指相比后移一位)，在具体实现上有何异同？哪一种更好?

顺序队列的类定义

```
class arrQueue: public Queue<T> {
private:
    int mSize;              // 存放队列的数组的大小
    int front;              // 表示队头所在位置的下标
    int rear;               // 表示队尾所在位置的下标
    T * qu;                 // 存放类型为T的队列元素的数组
public:                     // 队列的运算集
    arrQueue(int size);     // 创建队列的实例
    ~arrQueue();            // 消除该实例，并释放其空间
}
```

顺序队列的实现

```
template <class Elem> class Aqueue : public Queue<Elem> {
private:
    int size;                          // 队列的最大容量
    int front;                         // 队首元素指针
    int rear;                          // 队尾元素指针
    Elem *listArray;                   // 存储元素的数组
public:
    AQueue(int sz = DefaultListSize) {// 存储数组留一个空位
        size = sz+1;                   // size数组长，sz队列最大长度
        rear = 0; front = 1; // 也可以rear=-1; front=0
        listArray = new Elem[size];
    }
    ~AQueue() { delete [] listArray; }
    void clear() { front = rear+1; }
```

顺序队列的实现

```
bool enqueue(const Elem& it) {
     if (((rear+2) % size) == front) return false;
                    // 还剩一个空位时报告溢出
     rear = (rear+1) % size;          // 实指，需先移动到下一空位
     listArray[rear] = it;
     return true;
}

bool dequeue(Elem& it) {
     if (front == rear) return false;          // 队列为空
     it = listArray[front];           // 先出队，再移动front下标
     front = (front+1) % size;        // 环形增加
     return true;
}
```

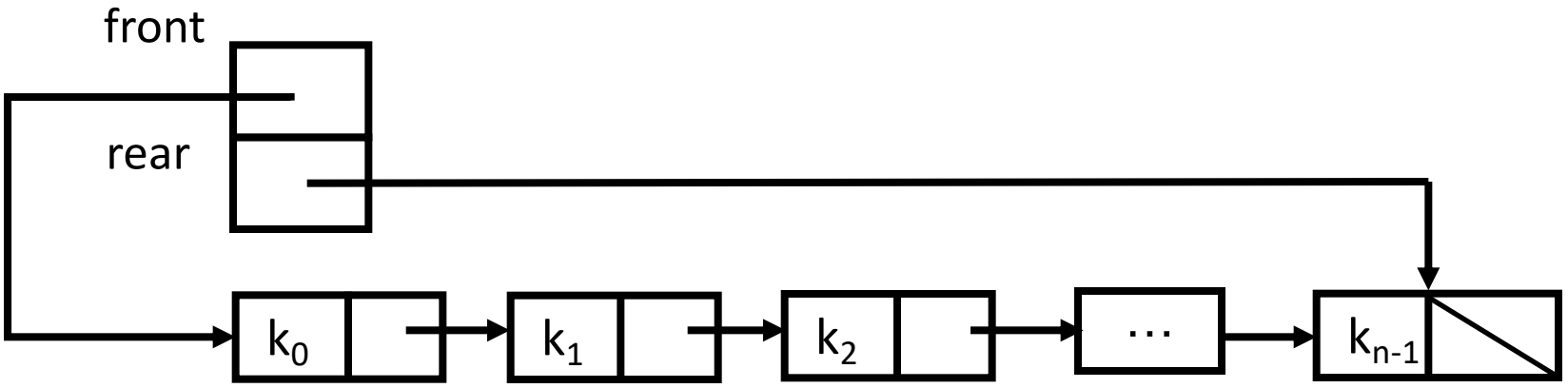
顺序队列的实现

```
bool frontValue(Elem& it) const {
     if (length() == 0)
       return false;       // 队列为空
     it = listArray[front]; return true;
}

int length() const {
    return (size +(rear - front + 1)) % size;
}
```

链式队列

- 单链表队列
- 链接指针的方向是从队队头到队尾



链式队列的类定义

```
template <class T>
class lnkQueue: public Queue<T> {
private:
    int size;                       // 队列中当前元素的个数
    Link<T>* front;                 // 表示队头的指针
    Link<T>* rear;                  // 表示队尾的指针
public:                             // 队列的运算集
    lnkQueue(int size);             // 创建队列的实例
    ~lnkQueue();                    // 消除该实例，并释放其空间
}
```

链式队列的入队

```
bool enQueue(const T item) {      //  item入队，插入队尾
    if (rear == NULL)      {       // 空队列
       front = rear = new Link<T> (item, NULL);
    }
    else  {                        // 添加新的元素
       rear-> next = new Link<T> (item, NULL);
       rear = rear ->next;
    }
    size++;
    return true;
}
```

链式队列的出队

```
bool deQueue(T* item)  {      // 返回队头元素并从队列中删除
    Link<T> *tmp;
    if (size == 0)  {         // 队列为空，没有元素可出队
        cout << "队列为空" << endl;
        return false;
    }
    *item = front->data;
    tmp = front;
    front = front -> next;
    delete tmp;
    if (front == NULL)
        rear = NULL;
    size--;
    return true;
}
```

顺序队列 vs 链式队列

- 顺序队列
  - 固定的存储空间
- 链式队列
  - 可以满足浪涌大小无法估计的情况
- 都不允许访问队列内部元素

队列的应用

- 只要满足先来先服务特性的应用均可采用队列作为其数据组织方式或中间数据结构

- **调度或缓冲**
  - 消息缓冲器
  - 邮件缓冲器
  - 计算机的硬设备之间的通信也需要队列作为数据缓冲
  - 操作系统的资源管理
- **宽度优先搜索**

变种的栈或队列结构

- 双端队列
  - 限制插入和删除在线性表的两端进行
- 超队列
  - 一种删除受限的双端队列：删除只允许在一端进行，而插入可在两端进行
- 超栈
  - 一种插入受限的双端队列，插入只限制在一端而删除允许在两端进行

思考

- 是否可以用栈来模拟队列？如果可以的话，需要几个栈，如何来模拟?

- 试利用非数组变量，按下述条件各设计一个相应的算法以使队列中的元素有序：

  a) 使用两个辅助的队列；

  b) 使用一个辅助的队列。