第8-9章　栈和队列

----后进先出/先进先出：操作受限的线性表

计算机学院

主要内容

- **堆栈的定义**
- 堆栈的描述
  - 公式化描述
  - 链表描述
- 堆栈的应用
  - 括号匹配、火车车厢重排

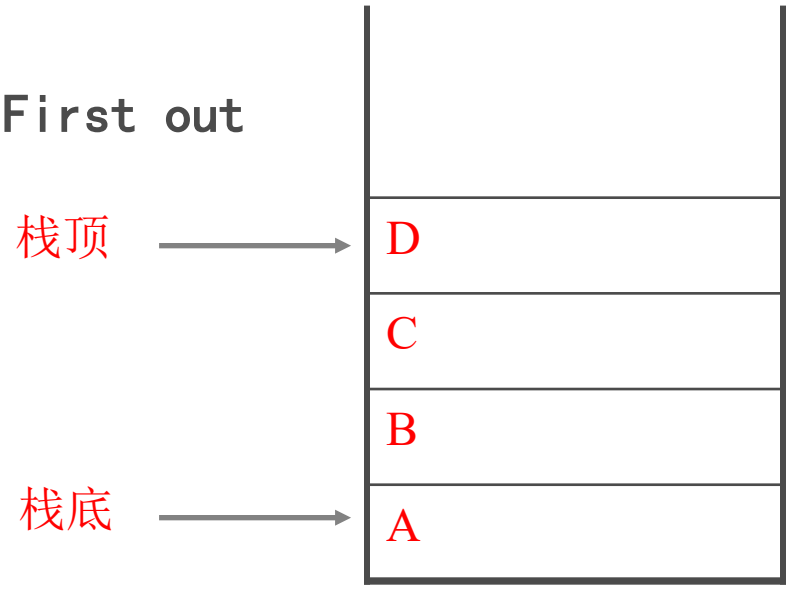
- 队列的定义
- 队列的描述
  - 公式化描述
  - 链表描述
- 队列的应用
  - 火车车厢重排

堆栈定义

- 堆栈（stack）是一个线性表，
  其插入和删除操作都在表的同一端进行，这端被称为栈顶（top），
  另一端被称为栈底（bottom）

- LIFO

Last in, First out




枪管
枪弹
套筒
击锤
复进簧
2
upload in www.cjdby.net


纸牌
游戏(G)
帮助(H)
时间: 38
得分: 26

抽象数据类型

抽象数据类型*Stack*{

实例

元素线性表，栈底，栈顶

操作

Create()：创建一个空的堆栈

IsEmpty()：如果堆栈为空，则返回true，否则返回false

IsFull()：如果堆栈满，则返回true，否则返回false

Top()：返回栈顶元素

Push(*x*)：向堆栈中添加元素*x*

Pop(*x*)：删除栈顶元素，并将它传递给*x*

}

主要内容

- 堆栈的定义
- **堆栈的描述**
  - **公式化描述**
  - 链表描述
- 堆栈的应用
  - 括号匹配、火车车厢重排
  - 汉诺塔、迷宫、开关盒布线、离线等价类

公式化描述：继承线性表

线性表尾部作为栈顶

```
template<class T>
class Stack : private LinearList <T> {
// LIFO objects
   public:
      Stack(int MaxStackSize = 10)
        : LinearList<T> (MaxStackSize) {}
      bool IsEmpty() const
          {return LinearList<T>::IsEmpty();}
      bool IsFull() const
        {return (Length() == GetMaxSize());}
```

公式化描述（续）

```
    T Top() const
       {if (IsEmpty()) throw OutOfBounds();
        T x; Find(Length(), x); return x;}
     Stack<T>& Push(const T& x)
       {Insert(Length(), x); return *this;}
     Stack<T>& Pop(T& x)
       {Delete(Length(), x);
        return *this;}
};
```

取栈顶——提取最后一个元素

压栈——添加到表尾

出栈——提取最后一个元素

实现方法分析

- IsFull需要获取数组大小
  - 方法一
    将类LinearList的成员MaxSize变为protected类型
  - 方法二：LinearList类增加函数
    protected:
    int GetMaxSize() const {return MaxSize;}
    LinearList类的变化不会影响Stack类，更好！

实现方法分析

- 继承方式为什么是private？
  - private继承会把基类的所有成员变为派生类的私有成员
  - 栈虽可看作线性表的特例，但毕竟不是
  - 用户使用Stack类，我们希望他们使用Push、Pop…，而不是Insert、Delete
  - 而private继承恰好可使Insert、Delete成为Stack的私有成员，用户无法看到

Stack的效率

- 构造函数、析构函数与LinearList相同
  - T：基本类型，Θ(1)
  - T：用户自定义类，Θ(MaxStackSize)
- 其他函数：Θ(1)

H1. 自定义的Stack类

```
template<class T>
class Stack {
public:
    Stack(int MaxStackSize = 10);
    ~Stack() {delete [] stack;}
    bool IsEmpty() const {return top == -1;}
    bool IsFull() const {return top == MaxTop;}
    T Top() const;
    Stack<T>& Push(const T& x);
    Stack<T>& Pop(T& x);
private:
    int top;    // current top of stack
    int MaxTop; // max value for top
    T *stack;   // element array
};
```

构造函数

```
template<class T>
Stack<T>::Stack(int MaxStackSize)
{// Stack constructor.
   MaxTop = MaxStackSize - 1;
   stack = new T[MaxStackSize];
   top = -1;
}
```

空栈

Top函数

```
template<class T>
T Stack<T>::Top() const
{// Return top element.
   if (IsEmpty()) throw OutOfBounds();
   return stack[top];
}
```

Push函数

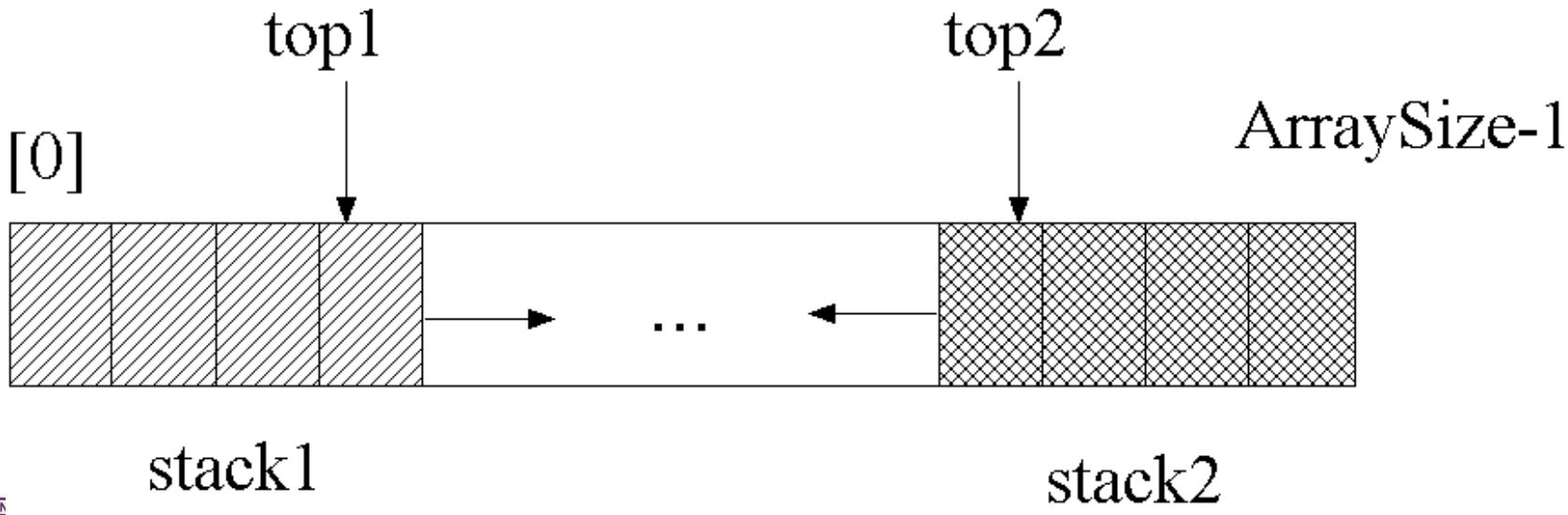
```
template<class T>
Stack<T>& Stack<T>::Push(const T& x)
{// Push x to stack.
  if (IsFull()) throw NoMem(); // Push fails
  stack[++top] = x;
  return *this;
}
```

Pop函数

```
template<class T>
Stack<T>& Stack<T>::Pop(T& x)
{// Delete top element and put in x.
   if (IsEmpty()) throw OutOfBounds();
   x = stack[top--];
   return *this;
}
```

数组描述缺陷

- 与线性表数组描述类似，空间利用率低
- 两个堆栈特例，空间利用率较高
  - Push最坏情况（数组满）仍为O(ArraySize)
  - Pop Θ(1)



主要内容

- 堆栈的定义
- **堆栈的描述**
  - 公式化描述
  - **链表描述**
- 堆栈的应用
  - 括号匹配、火车车厢重排
  - 汉诺塔、迷宫、开关盒布线、离线等价类

链表描述

- 栈顶在链表哪一端？
  - 尾节点
    - Push(x)——Insert(n, x)： $\Theta(n)$
    - Pop(x)——Delete(n, x)： $\Theta(n)$
  - 首节点
    - Push(x)——Insert(0, x)： $\Theta(1)$
    - Pop(x)——Delete(1, x)： $\Theta(1)$

LinkedStack类

```
template<class T>
class LinkedStack : private Chain<T> {
public:
   bool IsEmpty() const
      {return Chain<T>::IsEmpty();}
   bool IsFull() const;
   T Top() const
      {if (IsEmpty()) throw OutOfBounds();
       T x; Find(1, x); return x;}
```

LinkedStack类

```
  LinkedStack<T>& Push(const T& x)
    {Insert(0, x); return *this;}
  LinkedStack<T>& Pop(T& x)
     {Delete(1, x); return *this;}
};
```

IsFull函数

```
template<class T>
bool LinkedStack<T>::IsFull() const
{//Is stack full?
  try {ChainNode<T> *p = new ChainNode<T>;
      delete p;
      return false;}
  catch (NoMem) {return true;}
}
```

- 笨拙！

计算机学院

H2. 自定义的链表实现

```
template <class T>
class Node {
   friend LinkedStack<T>;
   private:
      T data;
      Node<T> *link;
};
```

自定义的链表实现（续）

```
template<class T>
class LinkedStack {
  public:
    LinkedStack() {top = 0;}
    ~LinkedStack();
    bool IsEmpty() const {return top == 0;}
    bool IsFull() const;
    T Top() const;
    LinkedStack<T>& Push(const T& x);
    LinkedStack<T>& Pop(T& x);
  private:
    Node<T> *top; // pointer to top node
};
```

析构函数

```
template<class T>
LinkedStack<T>::~LinkedStack()
{// Stack destructor..
  Node<T> *next;
  while (top) {
    next = top->link;
    delete top;
    top = next;
  }
}
```

IsFull函数

```
template<class T>
bool LinkedStack<T>::IsFull() const
{// Is the stack full?
  try {Node<T> *p = new Node<T>;
       delete p;
       return false;}
  catch (NoMem) {return true;}
}
```

Top函数

```
template<class T>
T LinkedStack<T>::Top() const
{// Return top element.
   if (IsEmpty()) throw OutOfBounds();
   return top->data;
}
```

Push

```
template<class T>
LinkedStack<T>&
  LinkedStack<T>::Push(const T& x)
{// Push x to stack.
   Node<T> *p = new Node<T>;
   p->data = x;
   p->link = top;
   top = p;
   return *this;
}
```

Pop

```
template<class T>
LinkedStack<T>& LinkedStack<T>::Pop(T& x)
{// Delete top element and put it in x.
  if (IsEmpty()) throw OutOfBounds();
  x = top->data;
  Node<T> *p = top;
  top = top->link;
  delete p;
  return *this;
}
```

H1-2小结

- 堆栈的两种实现方式

	公式化	链表
Create()	Θ(1)/Θ(MaxSize)	Θ(1)
Destroy()	Θ(1)/Θ(MaxSize)	Θ(n)
IsEmpty()	Θ(1)	Θ(1)
IsFull()	Θ(1)	Θ(1)
Top()	Θ(1)	Θ(1)
Push()	Θ(1)	Θ(1)
Pop()	Θ(1)	Θ(1)

主要内容

- 堆栈的定义
- 堆栈的描述
  - 公式化描述
  - 链表描述
- **堆栈的应用**
  - **括号匹配、火车车厢重排**
  - 汉诺塔、迷宫、开关盒布线、离线等价类

括号匹配

- (a*(b+c)+d)+(e-b)——符合语法
- (a+b))(——不符合语法
- 寻找匹配括号对——正确处理
  和未匹配括号——错误报告

- 括号匹配是一个基础问题，可以引申到
  - C++编译器
  - 数学公式自动求解

算法设计思路

- （ a ＊（ b ＋ c ）＋ d ）＋（e － b），匹配括号的规律？

  嵌套或者并列

- 右括号与谁匹配（如果有的话）？

  最近（右）未匹配左括号

- 由左至右处理符号的话，“右”←→“后”，靠后的先匹配——LIFO
  - 用一个栈保存未匹配的左括号
  - 由左至右扫描表达式串，遇左括号，push
  - 遇右括号，与栈顶左括号匹配，pop

匹配失败的情况

- （ a + b ））（——失败情况的规律

  右括号之前无与之匹配的左括号
  左括号之后无与之匹配的右括号

- 两种情况对应栈中情况

  遇到一个右括号时，无未匹配的左括号——栈空

  右括号都处理完时，还有未匹配的左括号——表达式串处理完时，栈不空

括号匹配程序

```
#include <iostream.h>
#include <string.h>
#include <stdio.h>
#include "stack.h"

const int MaxLength = 100; // max expression
  length
```

括号匹配程序（续）

```
void PrintMatchedPairs(char *expr)
{// Parenthesis matching.
   Stack<int> s(MaxLength);
   int j, length = strlen(expr);
   // scan expression expr for ( and )
   for (int i = 1; i <= length; i++) {
      if (expr[i - 1] == '(')   s.Push(i);
```

括号匹配程序（续）

```
else if (expr[i - 1] == ')')
        try {s.Pop(j);  // unstack match
          cout << j << ' ' << i << endl;}
        catch (OutOfBounds)
          {cout << "No match for right
parenthesis”<< " at " << i << endl;}
    }
```

括号匹配程序（续）

```
  // remaining ( in stack are unmatched
  while (!s.IsEmpty()) {
    s.Pop(j);
    cout << "No match for left parenthesis at "
        << j << endl;}
}
```

括号匹配程序（续）

```
void main(void)
{
  char expr[MaxLength];
  cout << "Type an expression of length at most "
       << MaxLength << endl;
  cin.getline(expr, MaxLength);
  cout <<"The pairs of matching parentheses in”<< endl;
  puts(expr);
  cout <<"are" << endl;
  PrintMatchedPairs(expr);
}
```

运行实例

Type an expression of length at most 100

(d+(a+b)*c*(d+e)-f))(()

The pairs of matching parentheses in

(d+(a+b)*c*(d+e)-f))(()

are

4 8

12 16

1 19

No match for right parenthesis at 20

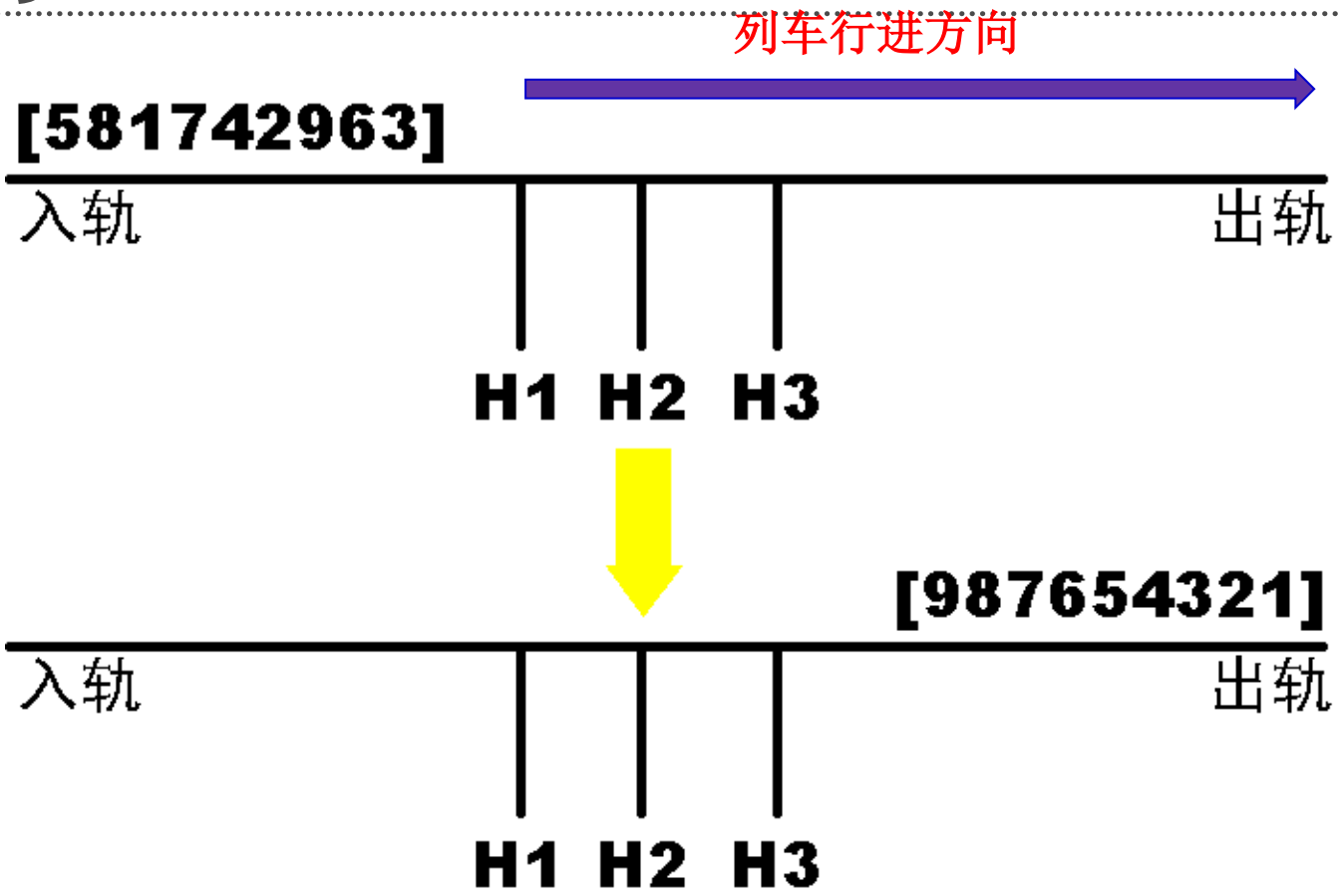
22 23

No match for left parenthesis at 21

火车车厢重排问题

- 货运列车，*n*节车厢，编号1~*n*
- 经过车站*n*～车站1，每站卸掉同号车厢
- 在始发站重新排列车厢，使得车厢按编号排列——每站卸掉最后一节车厢即可
- 转轨站——一个入轨、一个出轨、k个缓冲铁轨→完成重排
  - 允许三种操作
    - 入轨→缓冲轨
    - 缓冲轨→出轨
    - 入轨→出轨

图示



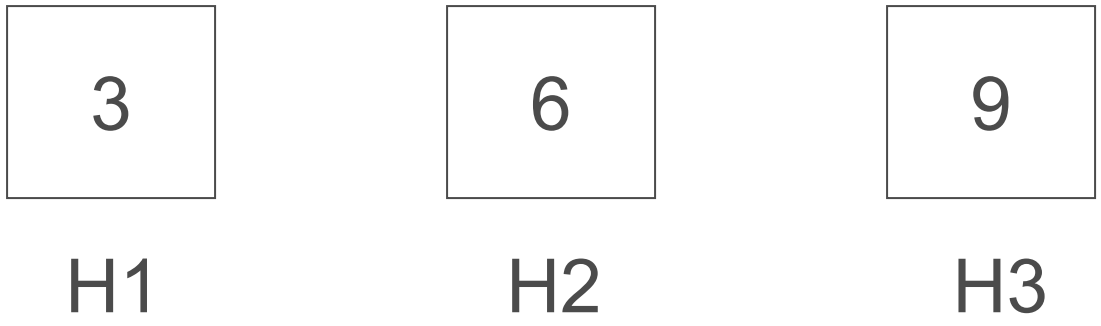
我们试着自己总结出算法

- 初始：581742963

  3：不是排列次序下一个，→H1

  6：次序不对，>3，→H2

  9：次序不对，>3，>6，→H3



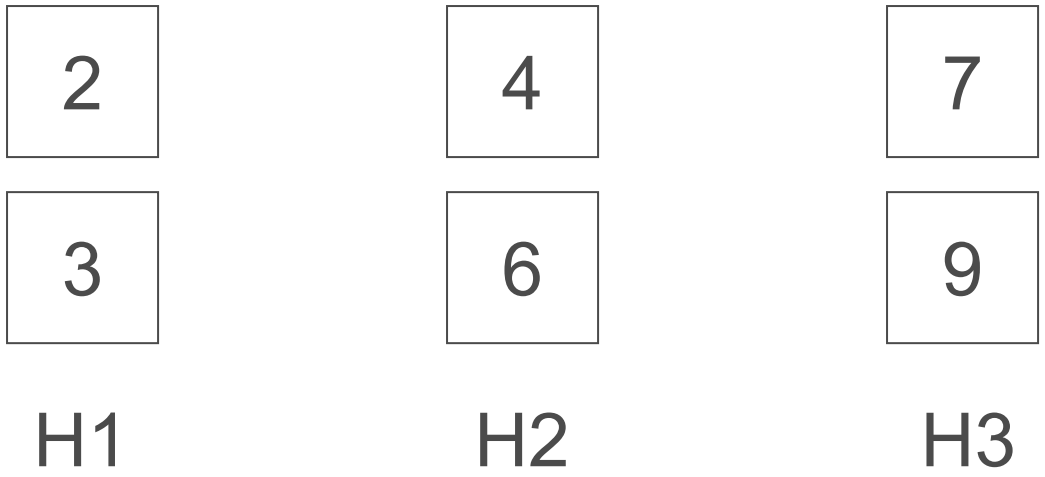
继续

- 入：581742　　出：

2：次序不对，2<3，2→H1

4：次序不对，4>2，4<6，4→H2

7：次序不对，7>2，7>4，7<9→H3



继续

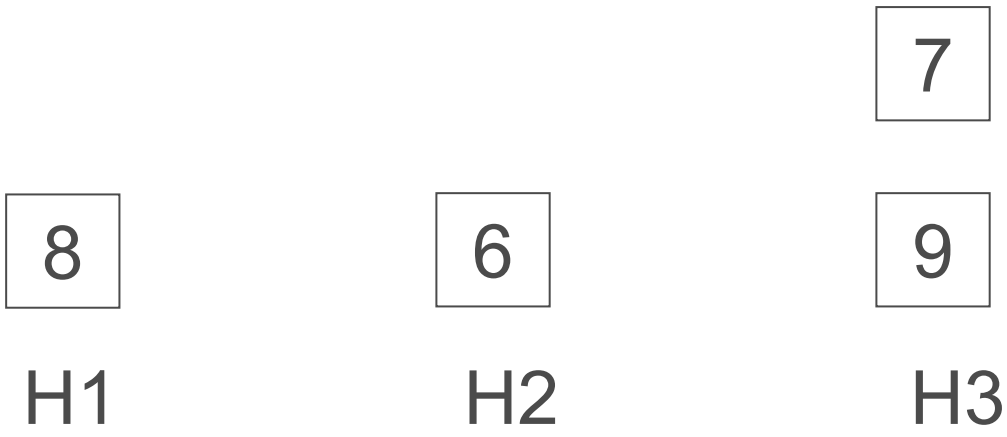
- 入：581　　出：1 2 3 4

1：次序对→出轨

H1：2→出轨，3→出轨

H2：4→出轨

8：次序不对，→H1



继续

- 入：5　　出：1 2 3 4 5 6 7 8 9

5：次序对→出轨

H2：6→出轨

H3：7→出轨

H1：8→出轨

H3：9→出轨，OK!

H1　　H2　　H3

重排算法

- 缓冲轨后进先出，用堆栈保存车厢号
  - 考虑到出轨顺序，**必须栈底大，栈顶小**
- 依次检查入轨车厢编号
  - 如果≠出轨所需要的下一车厢，→缓冲轨
    - 依次检查缓冲轨，若新来的<栈顶，入栈
  - 如果=出轨所需要的下一车厢，→出轨
    - 缓冲轨中车厢可能满足出轨需要，检查缓冲轨栈顶车厢，如有可能，出栈，→出轨，**不是一次**，要反复做，直至栈中无满足出轨需要的车厢

重排程序

k=?

```
bool Railroad(int p[], int n, int k)
{// k track rearrangement of car order p[1:n].
 // Return true if successful, false if impossible.
 // Throw NoMem exception if inadequate space.
```

为什么不是Stack？

```
  // create stacks for holding tracks
  LinkedStack<int> *H;
  H = new LinkedStack<int> [k + 1];

  int NowOut = 1;  // next car to output
  int minH = n+1;  // smallest car in a track
  int minS;        // track with car minH
```

重排程序（续）

```
// rearrange cars
  for (int i = 1; i <= n; i++)
    if (p[i] == NowOut) // send straight out
    {
      cout << "Move car " << p[i] <<" from input to output";
      cout<< endl;
      NowOut++;
      while (minH == NowOut)
      {
        Output(minH, minS, H, k, n);
        NowOut++;
      }
    }
```

重排程序（续）

```
    else {// put car p[i] in a holding track
      if (!Hold(p[i], minH, minS, H, k, n))
      return false;}

  return true;
}
```

Output：缓冲铁轨→出轨

```
void Output(int& minH, int& minS,
        LinkedStack<int> H[], int k, int n)
{// Move from hold to output and update minH and
   minS.
   int c;  // car index


   // delete smallest car minH from stack minS
   H[minS].Pop(c);
   cout << "Move car " << minH << " from holding
   track "<< minS << " to output" << endl;
```

Output：缓冲铁轨→出轨

```
  // find new minH and minS
  // by checking top of all stacks
  minH = n + 2;
  for (int i = 1; i <= k; i++)
    if (!H[i].IsEmpty() &&(c = H[i].Top()) < minH) {
      minH = c;
      minS = i;}
}
```

Hold：入轨→缓冲铁轨

```
bool Hold(int c, int& minH, int &minS,
          LinkedStack<int> H[], int k, int n)
{// Add car c to a holding track.
// find best holding track for car c
  // initialize
  int BestTrack = 0,    // best track so far
      BestTop = n + 1,  // top car in BestTrack
      x;                // a car index
```

Hold：入轨→缓冲铁轨（续）

```
for (int i = 1; i <= k; i++)
  if (!H[i].IsEmpty()) {// track i not empty
    x = H[i].Top();
    if (c < x && x < BestTop) {
      BestTop = x;
      BestTrack = i;}
    }
  else // track i empty
    if (!BestTrack) BestTrack = i;
if (!BestTrack) return false; // no feasible track
```

Hold：入轨→缓冲铁轨（续）

```
// add c to best track
H[BestTrack].Push(c);
cout << "Move car " << c << " from input "
     << "to holding track " << BestTrack << endl;

// update minH and minS if needed
if (c < minH) {minH = c;
               minS = BestTrack;}

return true;
}
```

复杂性

- Output: $\Theta(k)$
- Hold: $\Theta(k)$
- Railroad: $O(kn)$

课堂练习

- 元素a，b，c，d，e依次进入初始为空的栈中，所有元素进栈且只进入一次，允许进栈、退栈操作交替进行。栈空时，在所有可能的出栈序列中，以元素d开头的序列个数是_____。

  A．3

  B．4

  C．5

  D．6

课堂练习

- 若元素a, b, c, d, e, f依次进栈，允许进栈、退栈操作交替进行。但不允许连续三次进行退栈工作，则不可能得到的出栈序列是_____。【2010考研真题】

  A. dcebfa

  B. cbdaef

  C. abcdef

  D. afedcb

课堂练习

- 用S表示进栈操作，用X表示出栈操作，若元素的进栈顺序是1234，为了得到出栈顺序1342，相应的S和X的操作序列为_____。

A. SXSXSSXX

B. SSSXXSXX

C. SXSSXXSX

D. SXSSXSXX

课堂练习

- 已知一个栈的进栈序列为$p_1, p_2, p_3, \cdots, p_n$，其输出序列是1, 2, 3, …, n。若$p_3$=1，则$p_1$的值_____。

A. 一定是2

B. 可能是2

C. 不可能是2

D. 一定是3

课堂练习

- 若一个栈的输入序列为1, 2，……，n，输出序列的第一个元素是n，则第i个输出元素是什么？

我们学习了：

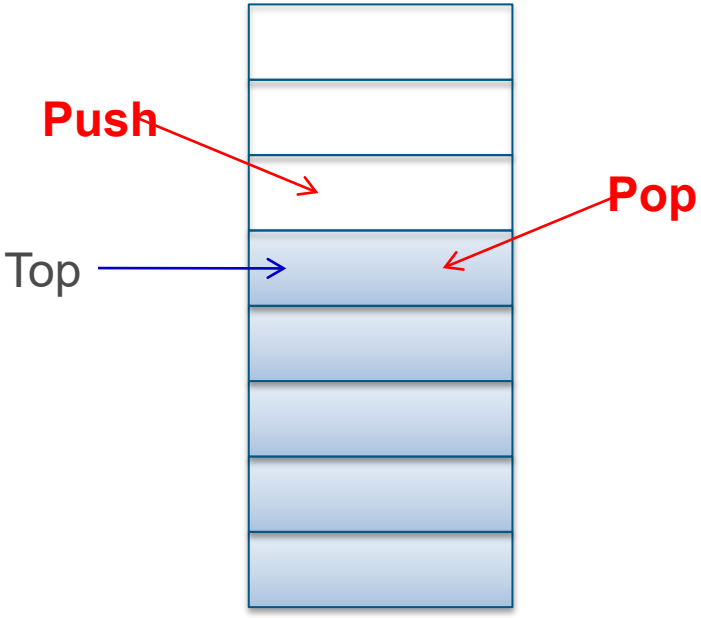
- 堆栈的定义和操作方式
- 堆栈的两种存储形式
  - 顺序、链表
- 堆栈的典型应用
  - 括号匹配、车厢重排

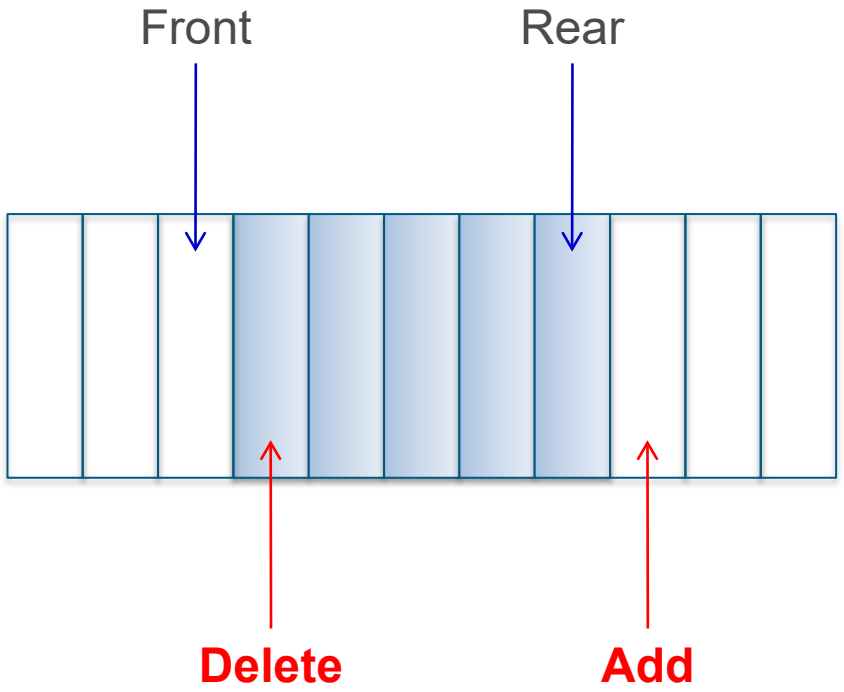
队列定义

- 队列（queue）是一个线性表，
  插入和删除操作分别在表的不同端进行。
  添加新元素的那一端被称为队尾（rear），
  而删除元素的那一端被称为队首（front）

- FIFO First in, First out

堆栈 VS 队列





课堂练习

- 设栈S和队列Q的初始状态均为空，元素a，b，c，d，e，f依次进入栈S。若每个元素出栈后立即进入队列Q，且6个元素出队的顺序是b，d，c，f，e，a，则栈S的容量至少是_____。

A．1

B．2

C．3

D．4

抽象数据类型描述

抽象数据类型*Queue*{

实例

有序线性表，一端称为front，另一端称为rear；

操作

Create()：创建一个空的队列；

IsEmpty()：如果队列为空，返回true，否则返回false；

IsFull()：如果队列满，则返回true；否则返回false；

First()：返回队列的第一个元素；

Last()：返回队列的最后一个元素；

Add(*x*)：向队列中添加元素*x*；

Delete(*x*)：删除队首元素，并送入*x*；

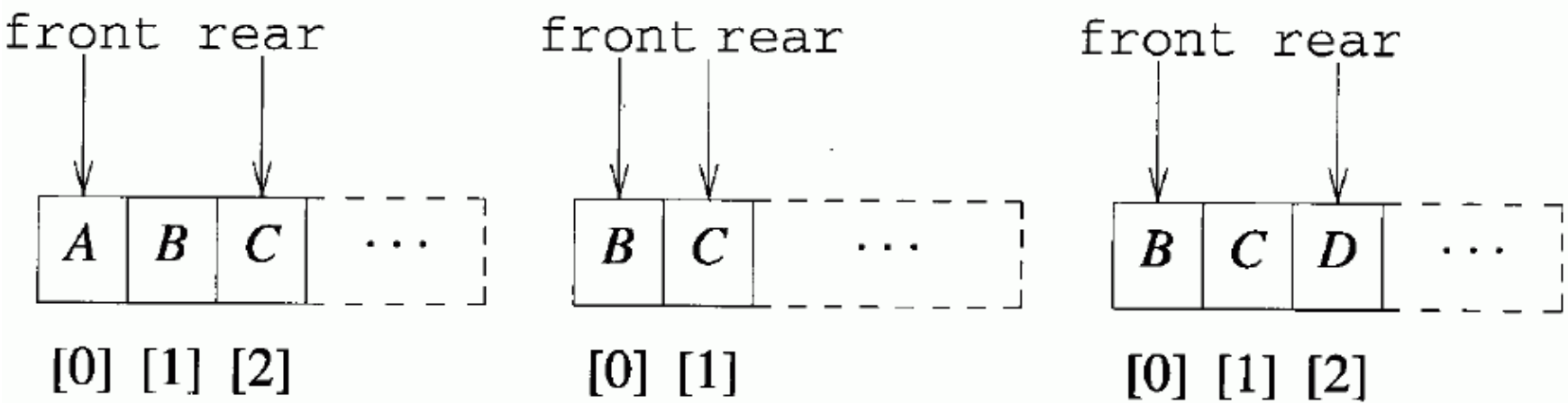
}

主要内容

- 队列的定义
- **队列的描述**
  - **公式化描述**
  - 链表描述
- 队列的应用
  - 火车车厢重排

公式化描述

- **思路1：在数组中从左至右依次存储**

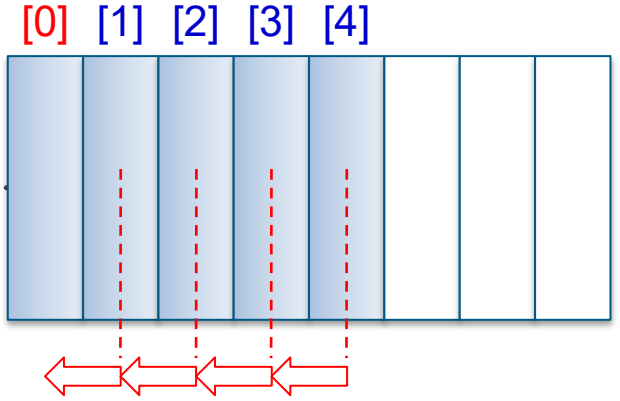


- 第一个元素保存在queue[0]
- 第二个在queue[1]，...，
- 第rear＋1个在queue[rear]

思路1（续）

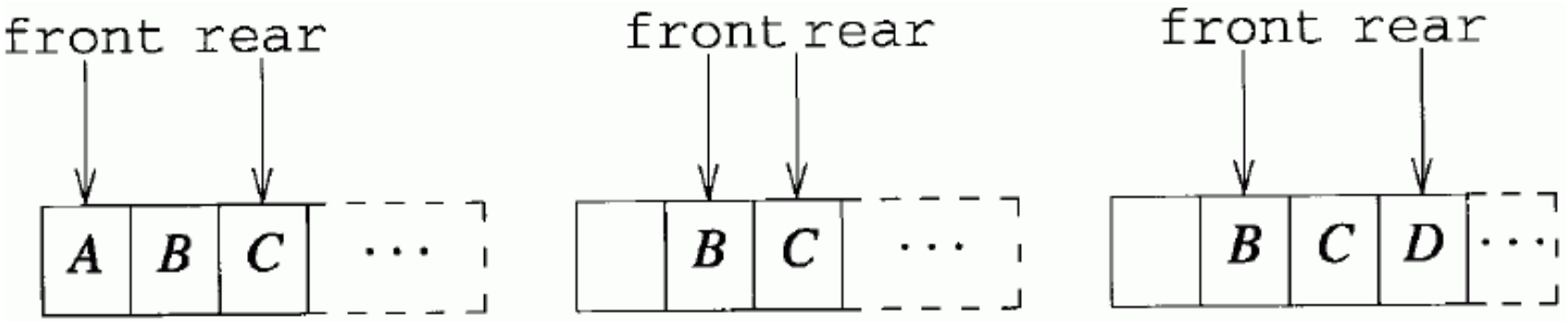
- location[i] = i - 1
- front≡0：队首始终在数组第一个位置
- 队列长度：rear + 1
- 队列空：rear == -1

操作实现



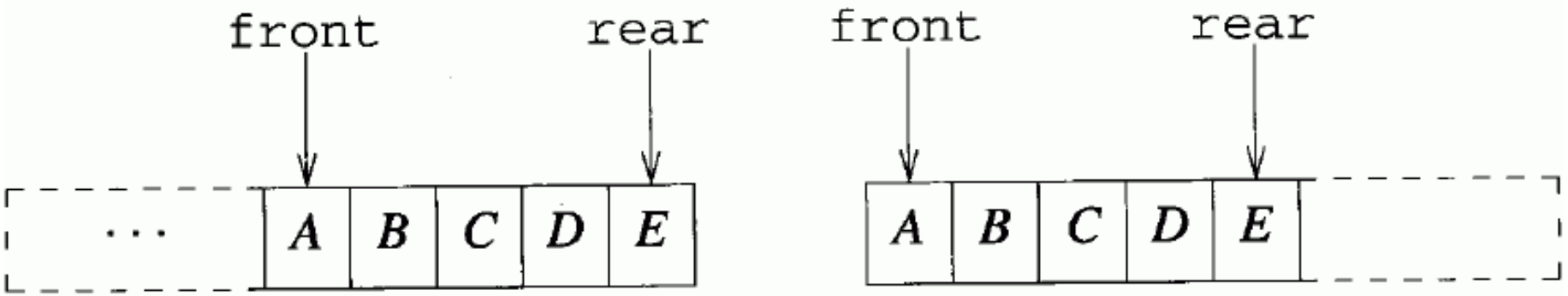
- 添加元素
  - rear++，新元素→queue[rear]
  - O(1)
- 删除元素
  - queue[1], ..., queue[rear]→queue[0], ..., queue[rear - 1]
  - Θ(n)
  - 对比堆栈——Θ(1)

思路2：队列头也移动



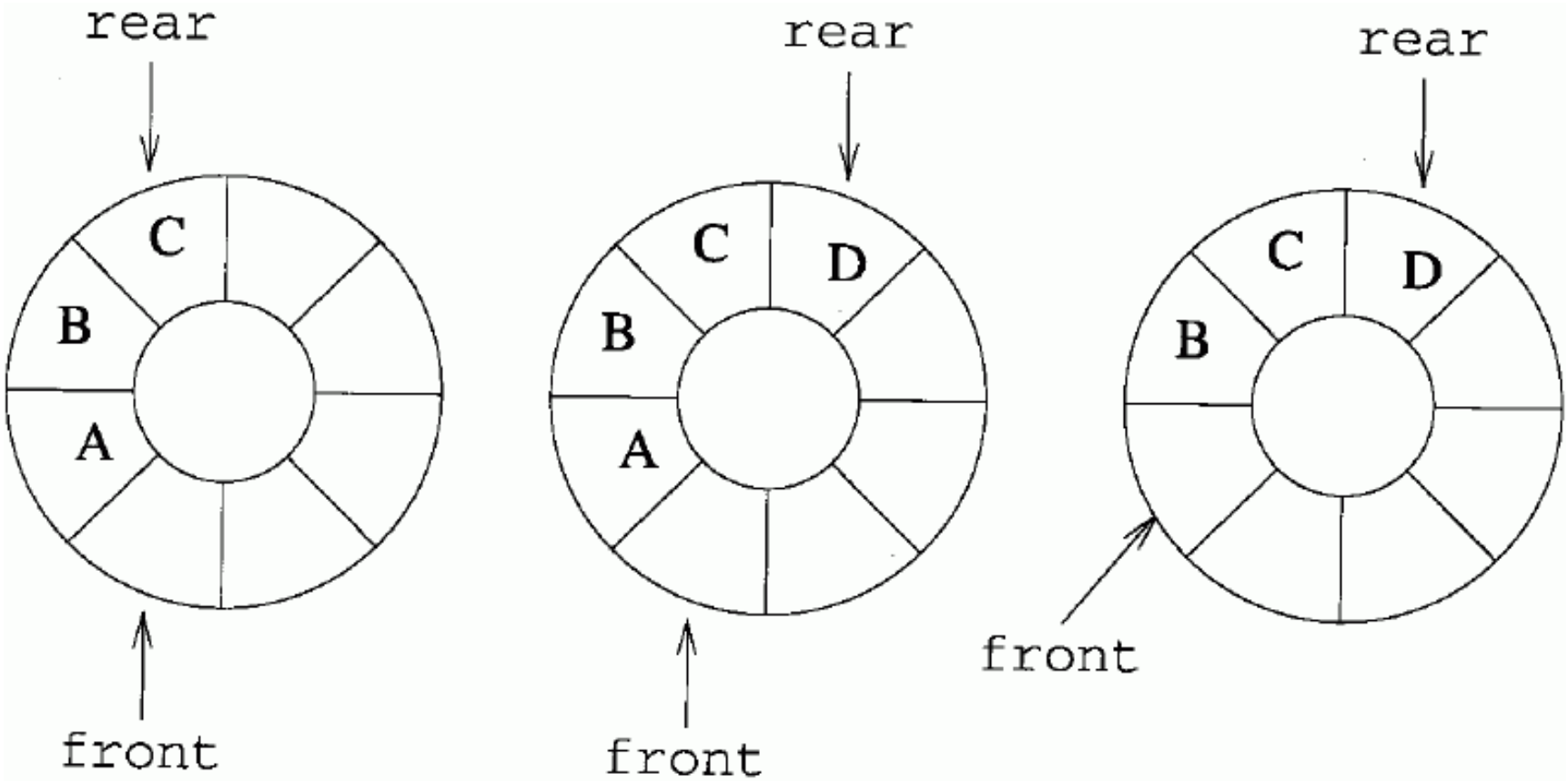
- location[i] = front + i - 1
- 删除：front++, Θ(1)
- 队列逐渐向数组尾部“移动”
- 队列空——front > rear

但是：队尾到达数组尾怎么办



- rear==MaxSize - 1时进行Add操作
  - front > 0
    - 队列移动到数组头部（front←0），再Add，Θ(n)
  - front==0，失败
- **改进了Delete但Add变差，最坏情况仍为Θ(n)**

思路3：循环数组描述方法

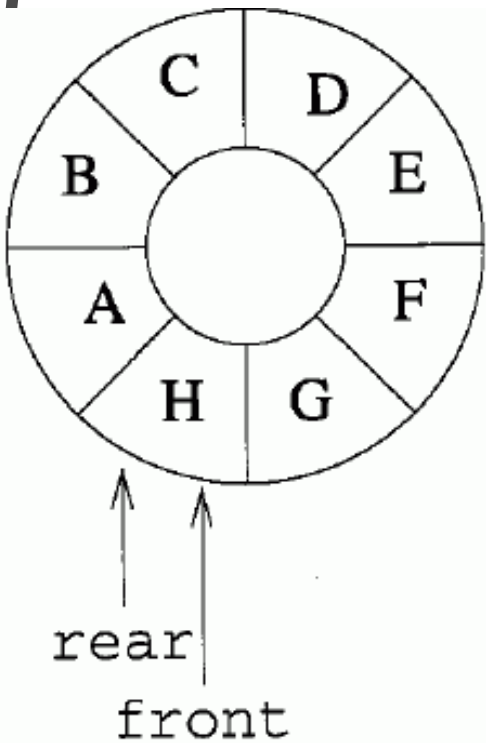


循环数组描述方法

- 把数组扭成环——**首尾相连是关键**
- 最后一个元素之后是第一个元素
- location(i)=(location(1)+i－1) % MaxSize
- front——location(1) － 1
- rear——最后一个元素在数组中的位置
- 插入、删除均为Θ(1)

小问题：队列空和满的区分

- 队列空——front == rear
- 队列满——front == rear
- 混淆！
- 简单解决方法【最终方案】
  - 只允许最多MaxSize-1个元素
  - 队列空：front == rear
  - 队列满：front == (rear + 1) % MaxSize



队列的公式化描述

```
template<class T>
class Queue {// FIFO objects
   public:
      Queue(int MaxQueueSize = 10);
      ~Queue() {delete [] queue;}
      bool IsEmpty() const {return front == rear;}
      bool IsFull() const {
         return (((rear + 1) % MaxSize == front) ? 1 : 0);}
      T First() const; // return front element
      T Last() const; // return last element
      Queue<T>& Add(const T& x);
      Queue<T>& Delete(T& x);
```

循环数组描述

```
  private:
    int front;   //队首
    int rear;    // 队尾
    int MaxSize; // 队列容量
    T *queue;    // element array
};
```

构造函数

```
template<class T>
Queue<T>::Queue(int MaxQueueSize)
{// Create an empty queue whose capacity
   MaxSize = MaxQueueSize + 1;
   queue = new T[MaxSize];
   front = rear = 0;
}
```

留出1个额外空间

- T：基本类型，　Θ(1)
  T：用户自定义类，　Θ(MaxSize)

First

```
template<class T>
T Queue<T>::First() const
{// Return first element of queue.  Throw
 // OutOfBounds exception if the queue is
   empty.
   if (IsEmpty()) throw OutOfBounds();
   return queue[(front + 1) % MaxSize];
}
```

Last

```
template<class T>
T Queue<T>::Last() const
{// Return last element of queue.  Throw
 // OutOfBounds exception if the queue is
   empty.
   if (IsEmpty()) throw OutOfBounds();
   return queue[rear];
}
```

插入操作

```
template<class T>
Queue<T>& Queue<T>::Add(const T& x)
{// Add x to the rear of the queue.  Throw
 // NoMem exception if the queue is full.
   if (IsFull()) throw NoMem();
   rear = (rear + 1) % MaxSize;
   queue[rear] = x;
   return *this;
}
```

请思考：当队列为空时，
插入的第一个元素在什么位置？

删除

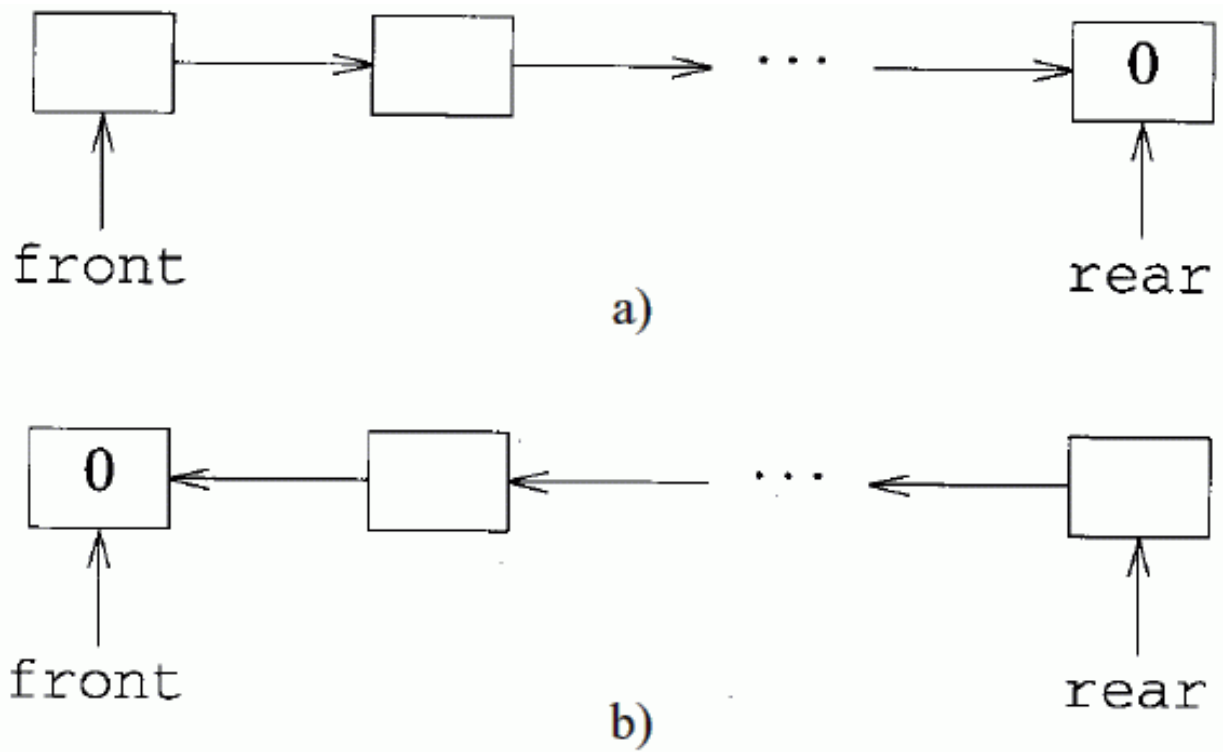
```
template<class T>
Queue<T>& Queue<T>::Delete(T& x)
{// Delete first element and put in x.  Throw
 // OutOfBounds exception if the queue is
   empty.
   if (IsEmpty()) throw OutOfBounds();
   front = (front + 1) % MaxSize;
   x = queue[front];
   return *this;
}
```

主要内容

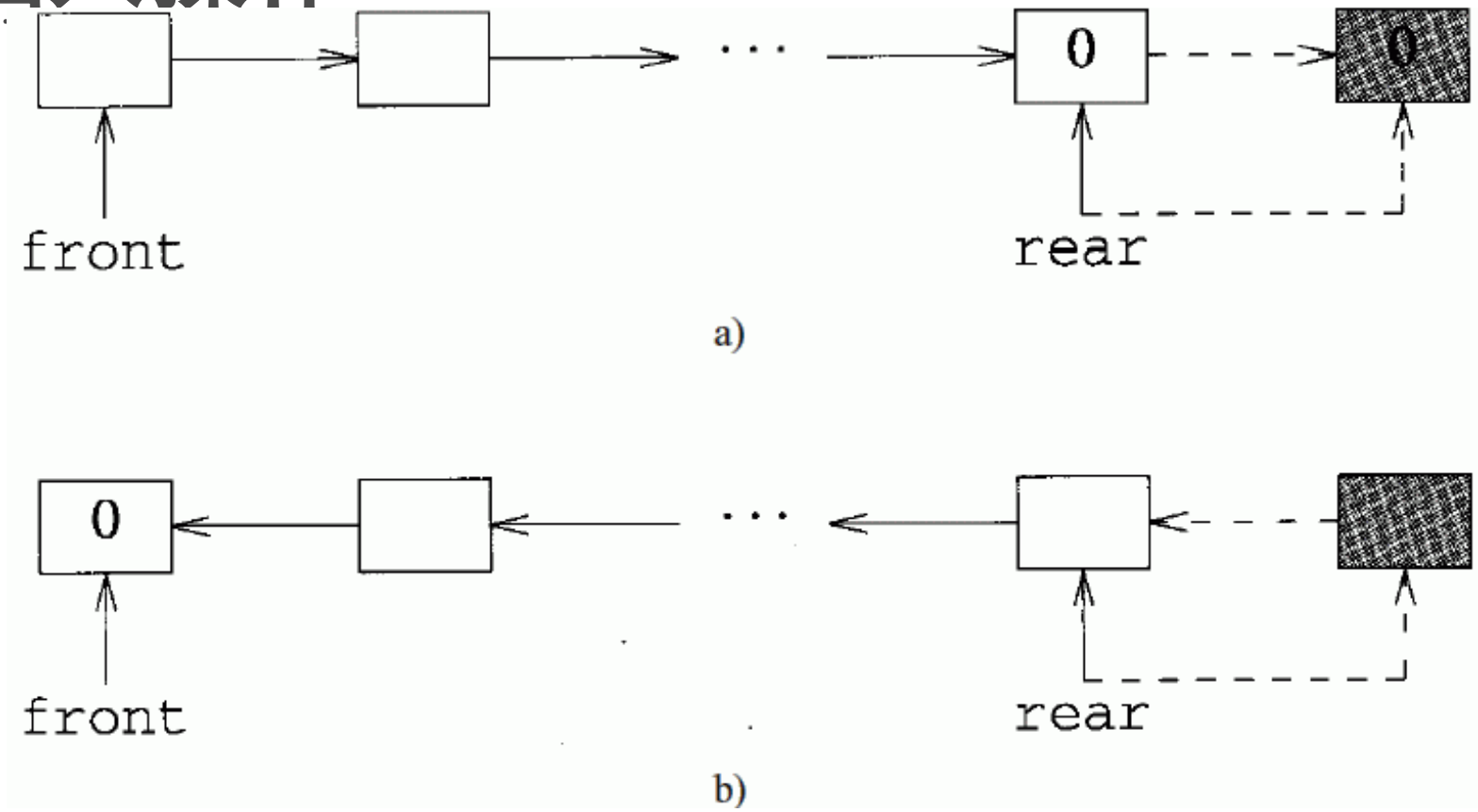
- 队列的定义
- **队列的描述**
  - 公式化描述
  - **链表描述**
- 队列的应用
  - 火车车厢重排
  - 最短路径、识别图元

链表描述

- 两种指针方向
  - front→rear
  - rear→front



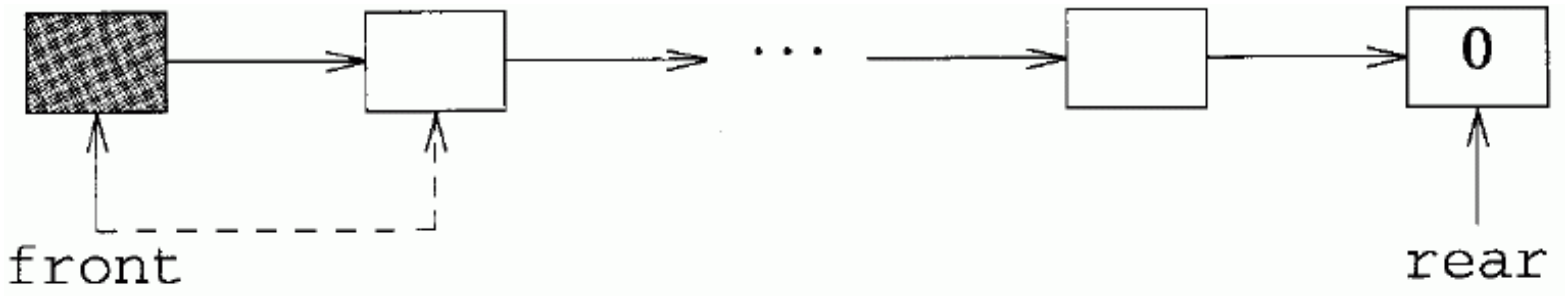
插入操作



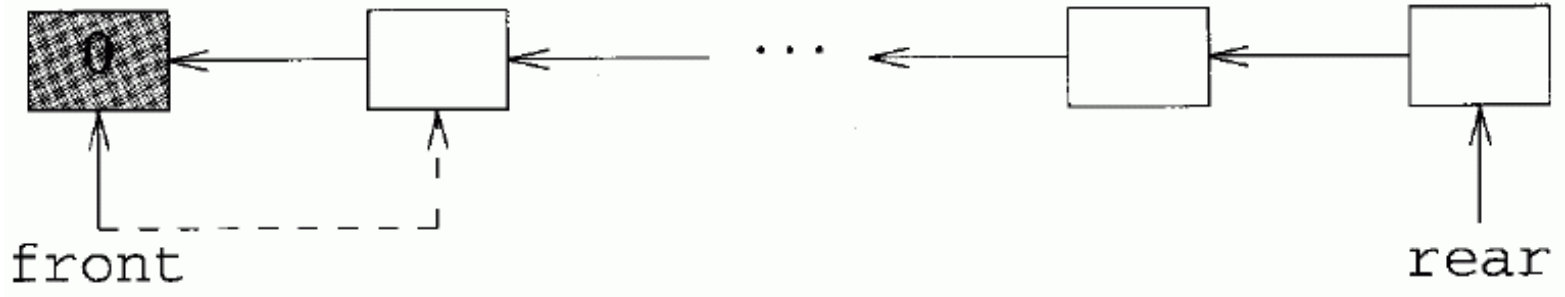
- 同样高效， Θ(1)

删除操作

因Delete操作的复杂性差异，
故选择front→rear的链接方式



a)



b)

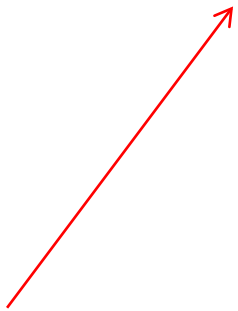
- front→rear Θ(1)，rear→front Θ(n)

LinkedQueue类

```
template<class T>
class LinkedQueue {
// FIFO objects
   public:
      LinkedQueue() {front = rear = 0;}
      ~LinkedQueue(); // destructor
      bool IsEmpty() const
          {return ((front) ? false : true);}
      bool IsFull() const;
      T First() const; // return first element
      T Last() const; // return last element
```

LinkedQueue类

```
    LinkedQueue<T>& Add(const T& x);
    LinkedQueue<T>& Delete(T& x);
  private:
    Node<T> *front;  // pointer to first node
    Node<T> *rear;   // pointer to last node
};
```



与公式化队列相比，减少了两个数据成员：**MaxSize**和***queue**

析构函数

```
template<class T>
LinkedQueue<T>::~LinkedQueue()
{// Queue destructor.  Delete all nodes.
  Node<T> *next;
  while (front) {
    next = front->link;
    delete front;
    front = next;
    }
}
```

析构函数与许多基于链表的数据结构都相似

IsFull

```
template<class T>
bool LinkedQueue<T>::IsFull() const
{// Is the queue full?
   Node<T> *p;
   try {p = new Node<T>;
        delete p;
        return false;}
   catch (NoMem) {return true;}
}
```

First

```
template<class T>
T LinkedQueue<T>::First() const
{// Return first element of queue.  Throw
 // OutOfBounds exception if the queue is
   empty.
   if (IsEmpty()) throw OutOfBounds();
   return front->data;
}
```

Last

```
<class T>
T LinkedQueue<T>::Last() const
{// Return last element of queue.  Throw
 // OutOfBounds exception if the queue is
   empty.
   if (IsEmpty()) throw OutOfBounds();
   return rear->data;
}
```

插入操作

```
template<class T>
LinkedQueue<T>&
  LinkedQueue<T>::Add(const T& x)
{
  Node<T> *p = new Node<T>;
  p->data = x;
  p->link = 0;//新建1节点并赋值
  if (front) rear->link = p;
  else front = p;
  rear = p;
  return *this;
}
```

两种情况：
- 队列不为空
- 队列为空

删除

```
template<class T>
LinkedQueue<T>&
  LinkedQueue<T>::Delete(T& x)
{
  if (IsEmpty()) throw OutOfBounds();
  x = front->data; //备份当前队首节点值
  Node<T> *p = front;
  front = front->link; //确定新的队首
  delete p; //释放原队首的空间
  return *this;
}
```

小结

- 队列的两种实现方式

	公式化	链表
Create()	Θ(1)/Θ(MaxSize)	Θ(1)
Destroy()	Θ(1)/Θ(MaxSize)	Θ(n)
IsEmpty()	Θ(1)	Θ(1)
IsFull()	Θ(1)	Θ(1)
First()	Θ(1)	Θ(1)
Last()	Θ(1)	Θ(1)
Add()	Θ(1)	Θ(1)
Delete()	Θ(1)	Θ(1)

课堂练习

- 某队列允许在其两端进行入队操作，但仅允许在一端进行出队操作。设入队顺序是abcde，则不可能得到的出队顺序是_____。

  A. bacde

  B. dbace

  C. dbcae

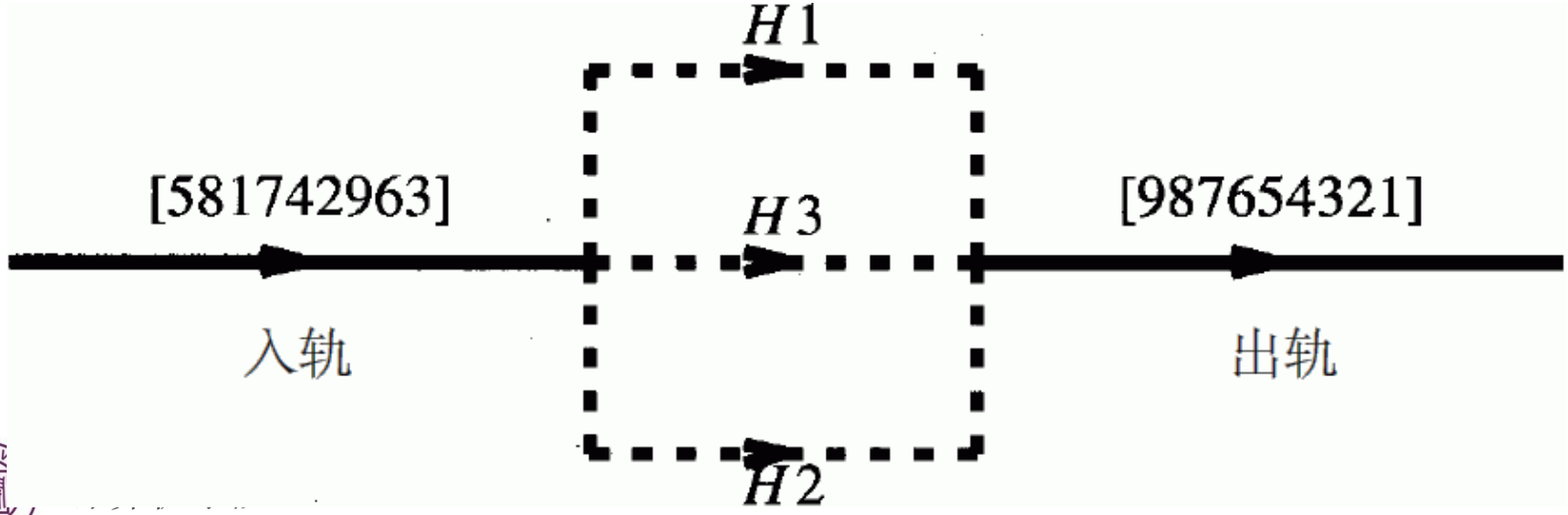
  D. ecbad

主要内容

- 队列的定义
- 队列的描述
  - 公式化描述
  - 链表描述
- **队列的应用**
  - **火车车厢重排**

火车车厢重排问题

- 缓冲铁轨按FIFO方式工作
- 也只允许：入轨→缓冲，缓冲→出轨
- H_k为入轨→出轨的直通轨，可用来缓冲的为H_1~H_{k-1}



例子

3：次序不对，→H1

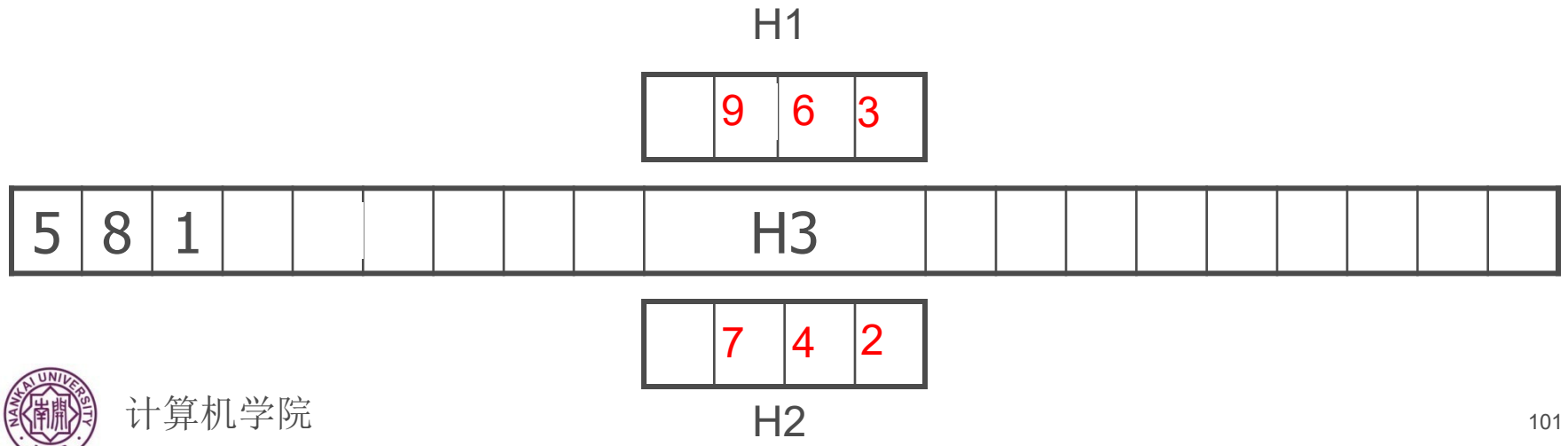
6：次序不对，>3，→H1

9：次序不对，>6，→H1

2：次序不对，<9，→H2

4：次序不对，>2，→H2

7：次序不对，>4，→H2



例子

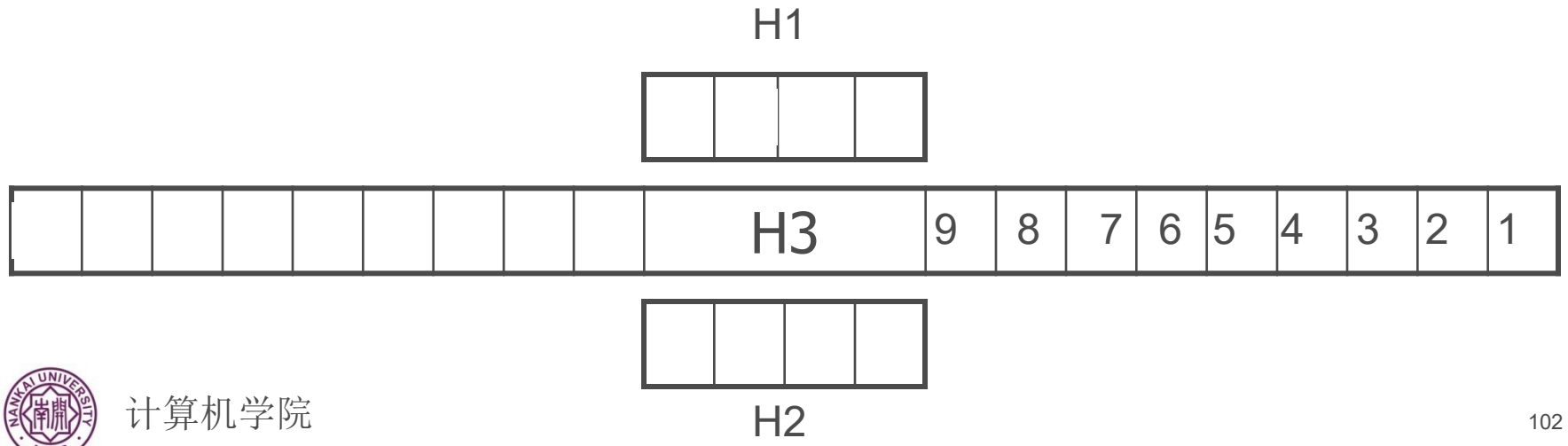
1：次序对，→出轨

2、3、4：→出轨

8：次序不对，<9，>7，→H2

5：次序对，→出轨

6、7、8、9：→出轨



基于链表队列的重排函数

```
bool Railroad(int p[], int n, int k)
{
```

待排车厢　　缓冲轨数

```
    LinkedQueue<int> *H; //链表队列数组，每个缓冲轨是
                         一个基于链表的队列
    H = new LinkedQueue<int> [k];
    k--;                 //第k个缓冲轨不用

    int NowOut = 1;      //当前应该出轨的车厢号
    int minH = n+1;  //所有缓冲轨上最小的车厢号
    int minQ;        //所有缓冲轨上最小的车厢号在哪个轨上?
```

基于链表队列的重排函数

```
for (int i = 1; i <= n; i++) //依次考查入轨上的每个车厢
{
    if (p[i] == NowOut) // 如果正好是要出轨的那节车厢
    {//首先令其出轨，然后从缓冲轨上出轨直至缓冲轨稳定
        cout << "Move car"<<p[i]<<" from input to output" << endl;
        NowOut++;
        while (minH == NowOut) {//该条件用于判断缓冲轨是否稳定
            Output(minH, minQ, H, k, n);
            NowOut++;
        }
    } //if结束时的状态：缓冲轨上的车厢均无法出轨
    else {//如果入轨上的当前车厢不能直接出轨，则将其放入缓冲轨
        if (!Hold(p[i], minH, minQ, H, k))
        return false;
    }
  }
  return true;
}
```

Output函数：一次缓冲轨→出轨

```
void Output(int& minH, int& minQ,
      LinkedQueue<int> H[], int k, int n)
{
    int c;
    H[minQ].Delete(c); //从缓冲轨上出列最小车厢
    cout <<"Move car"<<minH<<"from holding track"<<minQ <<"to output"<<endl;
    minH = n + 2;
    for (int i = 1; i <= k; i++)
    {//一次遍历，求取MIN( First() )，从而更新minH和minQ
       if (!H[i].IsEmpty()&&(c = H[i].First()) < minH) {
           minH = c;
           minQ = i;}
     }
}
```

Hold函数

要缓冲的车厢号

```
bool Hold(int c,int& minH, int &minQ,LinkedQueue<int> H[], int k)
{      ……
    int BestTrack = 0,BestLast = 0, x;
    for (int i = 1; i <= k; i++) //通过循环确定最优缓冲轨
        if (!H[i].IsEmpty()) {
            x = H[i].Last();
            if (c > x && x > BestLast) {
                BestLast = x;    BestTrack = i;}
            }
        else if (!BestTrack) BestTrack = i;
    if (!BestTrack) return false; //如果所有缓冲轨都不能添加车厢c
    H[BestTrack].Add(c);
    cout<<"Move car " << c << " from input "<< "to holding track " << BestTrack << endl;
    if (c < minH) //如果c添加到了一条空缓冲轨且c<minH
        {minH = c;  minQ = BestTrack;}
    return true;
}
```

如果该缓冲轨上已有车厢：
观察其①可否添加？c>x
②是否最大？x>BestLast

只有非空缓冲轨都不行时，
才考虑当前为空的缓冲轨

我们学习了：

- 队列的定义和操作方式
- 队列的两种存储形式
  - 顺序、链表
- 队列的典型应用
  - 车厢重排

本章结束