数据结构—C++实现

沈 俊

jshen@t.shu.edu.cn

上海大学 计算机工程与科学学院

2020年12月

第4章　栈和队列

- 栈
  - 栈的基本概念
  - 栈的存储结构
  - 栈的简单应用
- 队列
  - 队列的基本概念
  - 队列的存储结构
  - 队列的简单应用
- 递归的实现
  - 递归的概念
  - 递归过程与递归工作栈
  - 消除递归

4.1 栈

栈（stack）是限定仅在表尾一端进行插入或删除操作的特殊线性表。

对于栈来说，允许进行插入或删除操作的一端称为栈顶（top），而另一端称为栈底（bottom）。

不含元素栈称为空栈；

向栈中插入一个新元素称为入栈或压栈；

从栈中删除一个元素称为出栈或退栈。

栈又称为后进先出（Last In First Out，LIFO）的线性表。

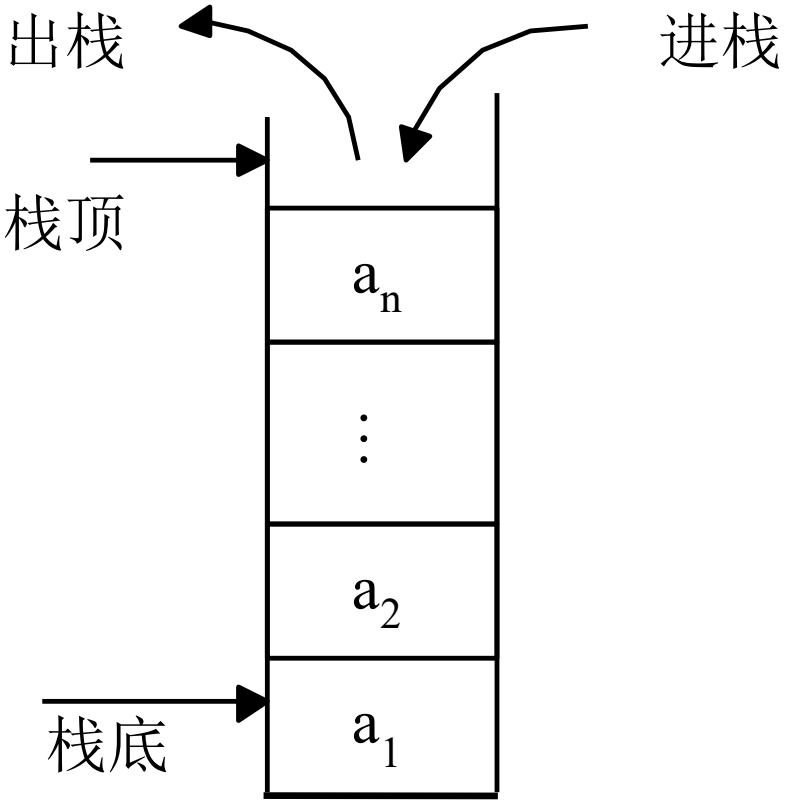


图4.1 栈的结构示意图

栈的实例

- 日常生活中的叠碗

栈的基本操作

1、初始化
2、求长度
3、取栈顶元素
4、入栈
5、出栈
6、判断栈是否为空栈
7、清空栈

栈的存储结构

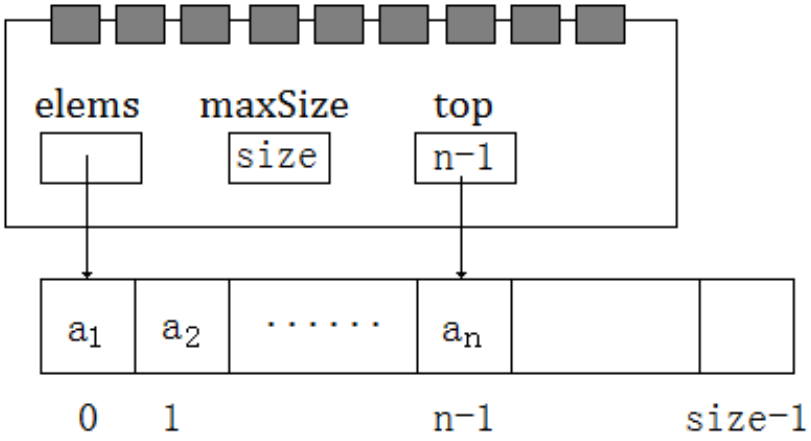
1、栈的顺序存储

2、栈的链式存储

3、栈的两种存储结构的比较

栈的顺序存储结构

```
template<class ElemType>
class SeqStack {
protected:
    int top;            // 栈顶指针
    int maxSize;        // 栈最大容量
    ElemType *elems;// 元素存储空间
```

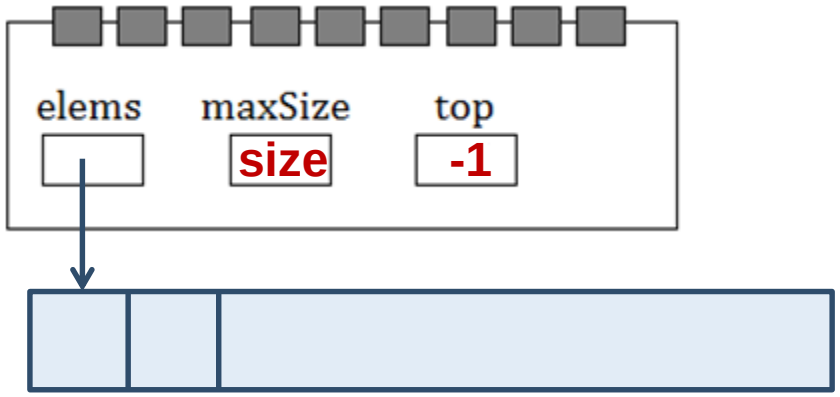


栈的顺序存储结构

```
public:
        SeqStack(int size = DEFAULT_SIZE);
        virtual ~SeqStack();
        int GetLength() const;
        bool IsEmpty() const;
        void Clear();
        Status Push(const ElemType e);
        Status Top(ElemType &e) const;
        Status Pop(ElemType &e);
        ……
};
```

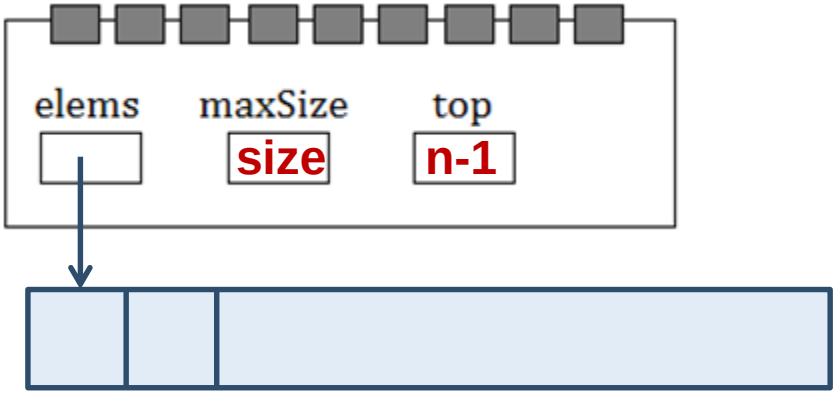
顺序栈的构造函数

```
template<class ElemType>
SeqStack<ElemType>::SeqStack(int size)
// 操作结果：构造一个最大容量为size的空栈
{
        maxSize = size;
        elems = new ElemType[maxSize];
        top = -1;
}
```



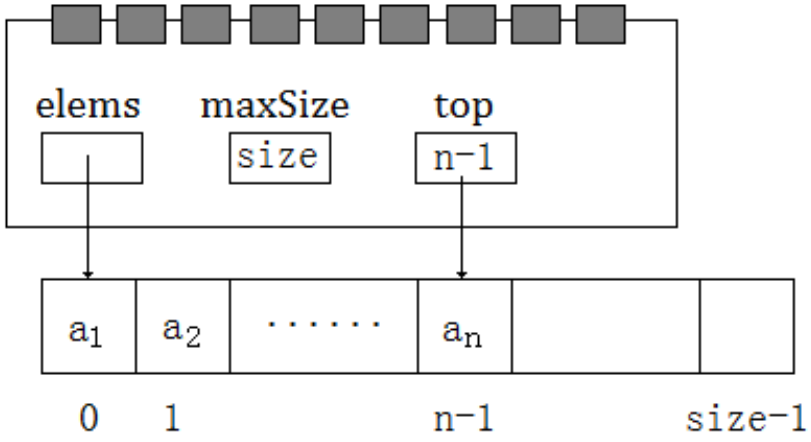
顺序栈的析构函数

```
template<class ElemType>
SeqStack<ElemType>::~SeqStack()
// 操作结果：销毁栈
{
        delete []elems;
}
```



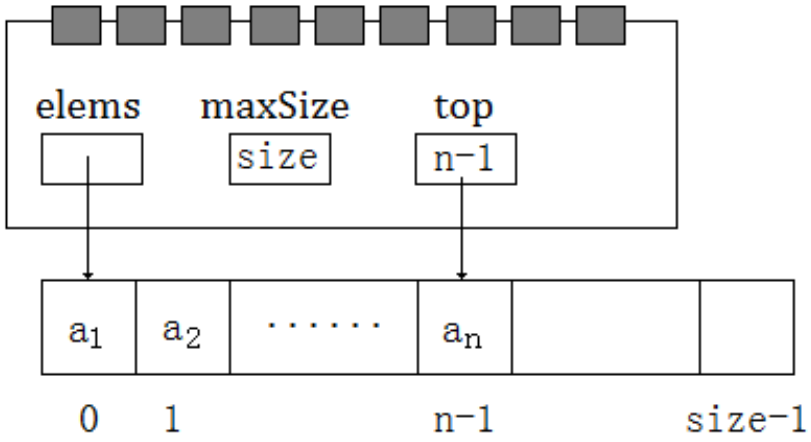
求顺序栈的长度

```
template <class ElemType>
int SeqStack<ElemType>::GetLength() const
// 操作结果：返回栈中元素个数
{
        return top+1;
}
```



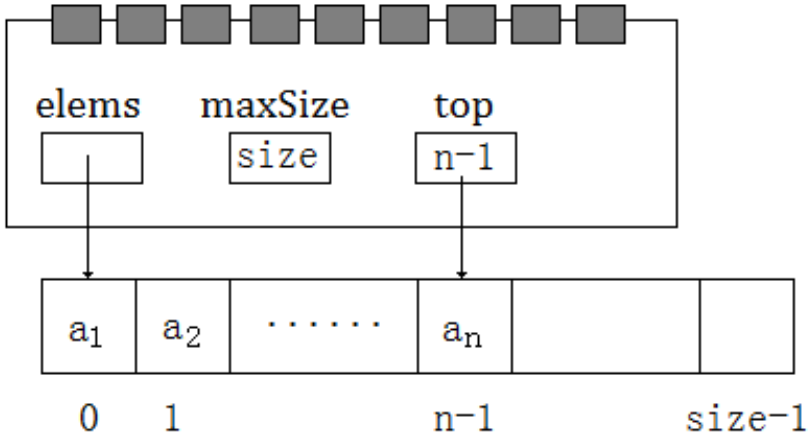
判断顺序栈是否为空

```
template<class ElemType>
bool SeqStack<ElemType>::IsEmpty() const
// 操作结果：如栈为空，则返回true，否则返回false
{
        return top == -1;
}
```



清空顺序栈

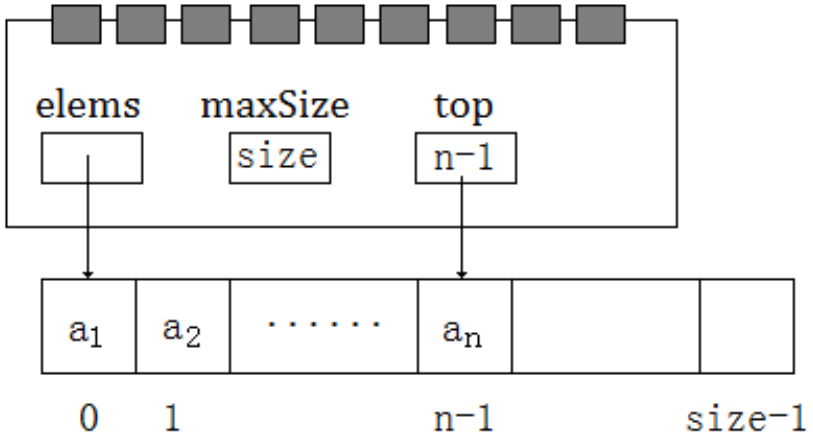
```
template<class ElemType>
void SeqStack<ElemType>::Clear()
{
        top = -1;
}
```



入栈

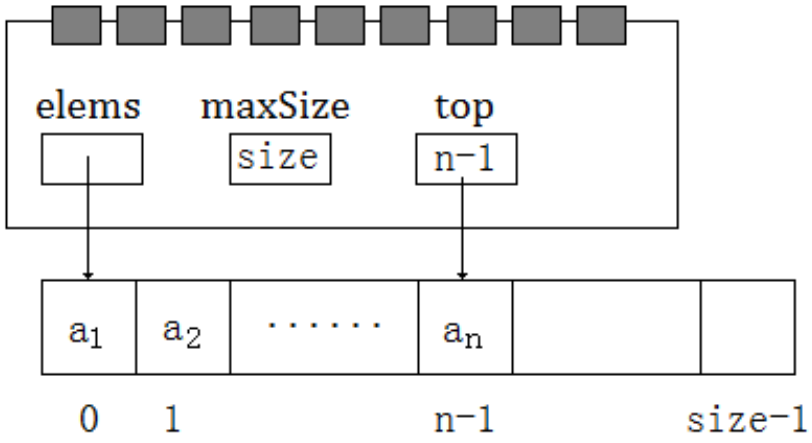
```
template<class ElemType>
Status SeqStack<ElemType>::Push(const ElemType e)
{
   if (top == maxSize - 1)      // 栈已满
         return OVER_FLOW;
   else  {
         elems[++top] = e;      // 将元素e追加到栈顶
         return SUCCESS; // 操作成功

   }
}
```



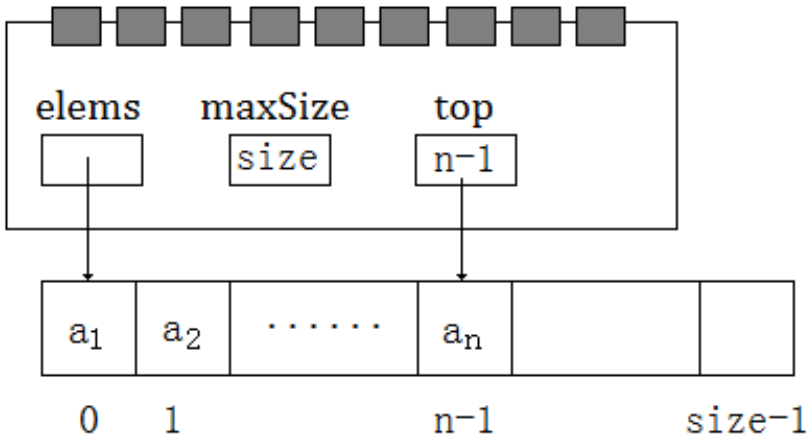
取栈顶元素

```
template<class ElemType>
Status SeqStack<ElemType>::Top(ElemType &e) const
{
   if (IsEmpty())                    // 栈空
         return UNDER_FLOW;
   else{
         e = elems[top];       // 用e返回栈顶元素
         return SUCCESS;  // 栈非空,操作成功
   }
}
```



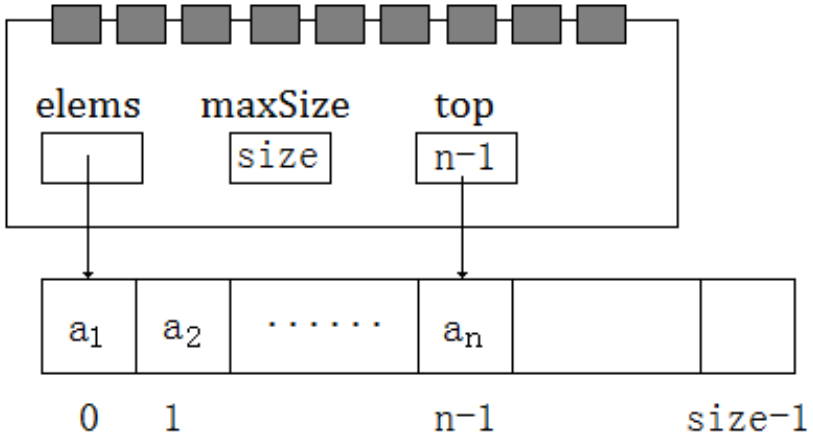
出栈

```
template<class ElemType>
Status SeqStack<ElemType>::Pop(ElemType &e)
{
   if (IsEmpty())                  // 栈空
         return UNDER_FLOW;
   else{
         e = elems[top--];    // 用e返回栈顶元素
         return SUCCESS;  // 操作成功
   }
}
```



遍历顺序栈

```
template <class ElemType>
void SqStack<ElemType>::Traverse(void
(*Visit)(const ElemType &)) const
{
        for (int i = top; i >=0 ; i--)
                (*Visit)(elems[i]);
}
```



两个顺序栈共享一个数组的存储空间

栈的顺序存储结构是一种静态的存储结构，栈中元素的多少受到MAXSIZE的限制，空间太大会造成存储空间的浪费，为解决这个问题，同时建立多个顺序栈，以实现存储空间的共享，这样就可以相互调节余缺，既能高效地节约存储空间，又能降低上溢的发生概率。

为两个栈申请一个共享的一维数组空间，将两个栈的栈底分别放在一维数组的两端，分别是-1和m。 由于两个栈顶动态变化，这样可以形成互补，使得每个栈可用的最大空间与实际使用的需求有关。由此可见，两栈共享比两个栈分别申请M/2的空间利用率高。

两个顺序栈共享一个数组的存储空间

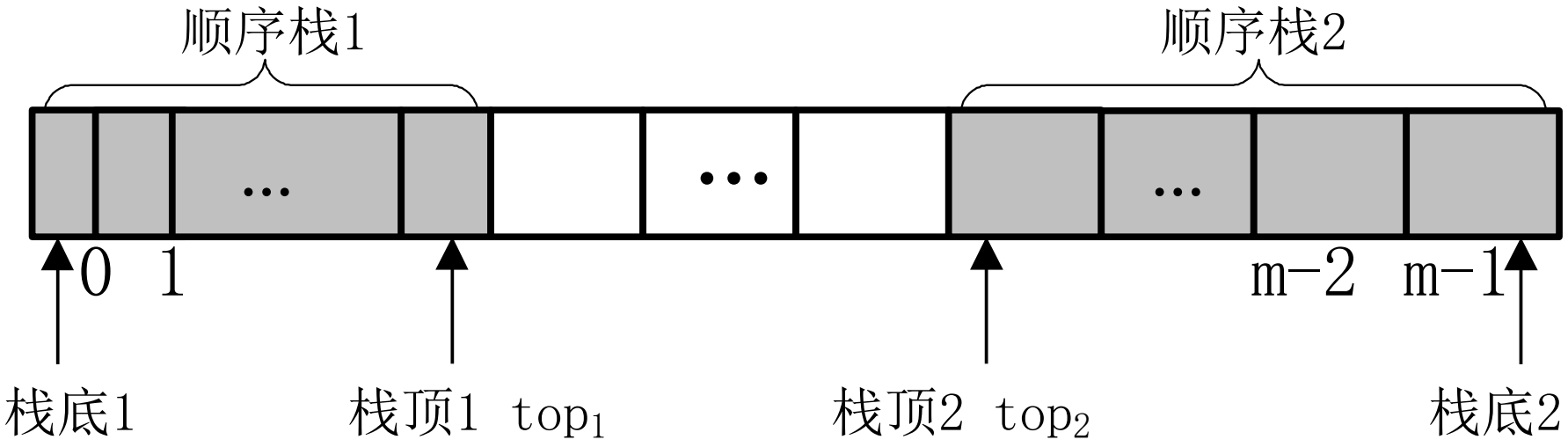
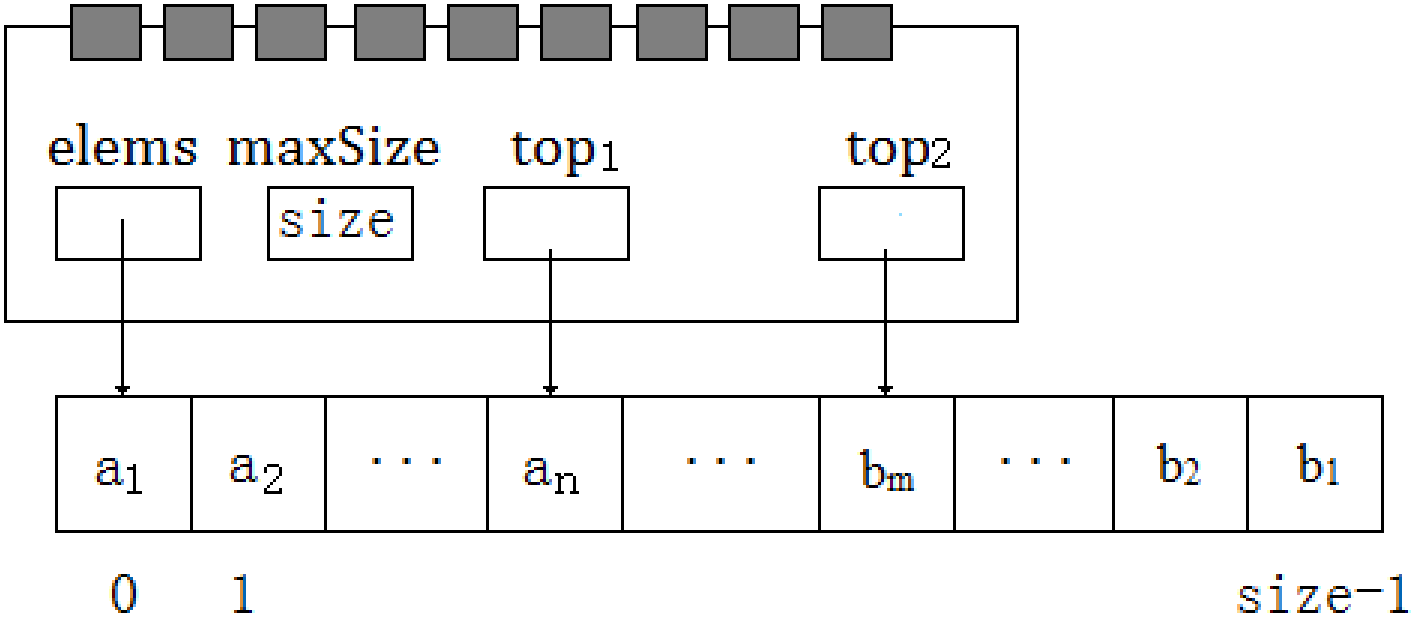


图4.5 共享栈

栈空的条件：$top_1 = -1$，$top_2=m$

栈满的条件：$top_1 + 1 = top_2$

两个顺序栈共享一个数组的存储空间



栈的链式存储结构

- 用不带头结点的单链表表示栈。
- top是栈顶指针，指向链栈的栈顶结点；
- top=NULL 表示栈空；
- 若链栈非空，则top是指向链表的第一个结点(栈顶结点)的指针。
- 栈顶结点是最后一个入栈的元素，而栈底结点是最先入栈的元素。

链栈存储的示例

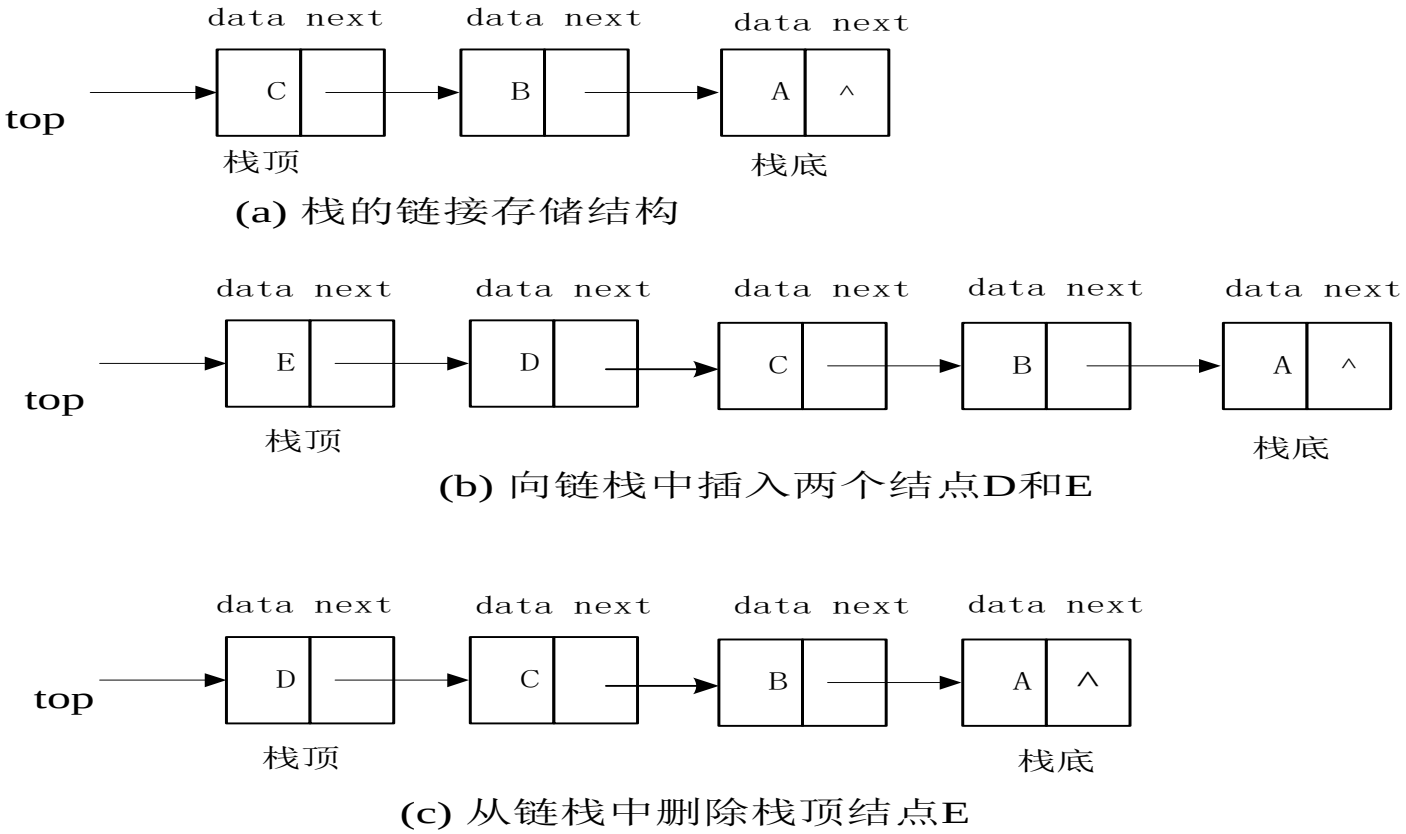
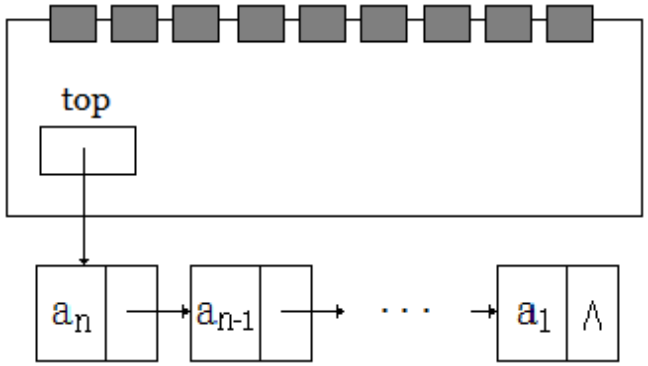


图4.4 栈的链接存储结构及其操作过程

链式栈的类模板定义

```
#include "node.h"                          // 结点类

template<class ElemType>
class LinkStack
{
protected:
        Node<ElemType> *top;               // 栈顶指针
```

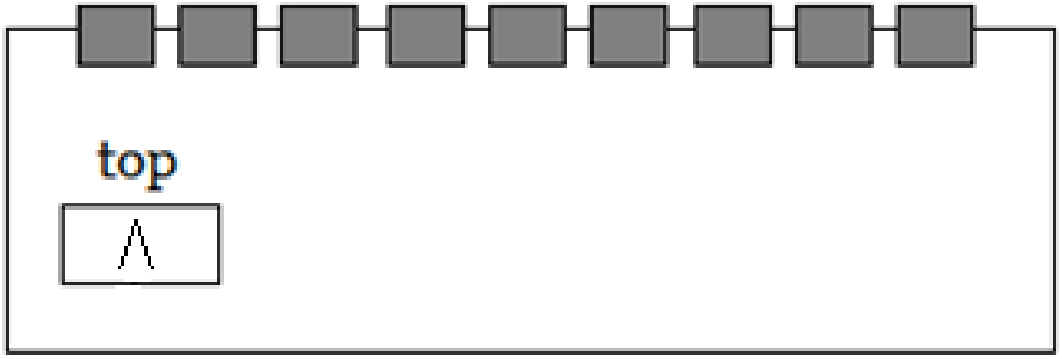


链式栈的类模板定义

```
public:
        LinkStack();
        virtual ~LinkStack();
        int GetLength() const;
        bool IsEmpty() const;
        void Clear();
        Status Push(const ElemType e);
        Status Top(ElemType &e) const;
        Status Pop(ElemType &e);
        ……
};
```

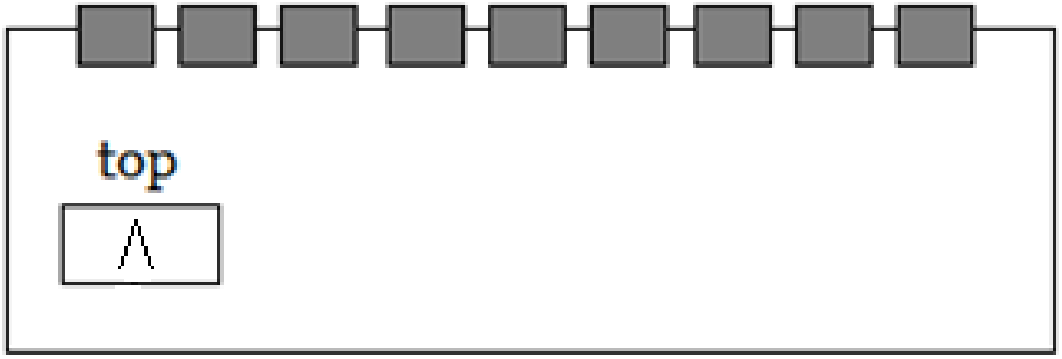
链序栈的构造函数

```
template<class ElemType>
LinkStack<ElemType>::LinkStack()
{
        top = NULL;
}
```



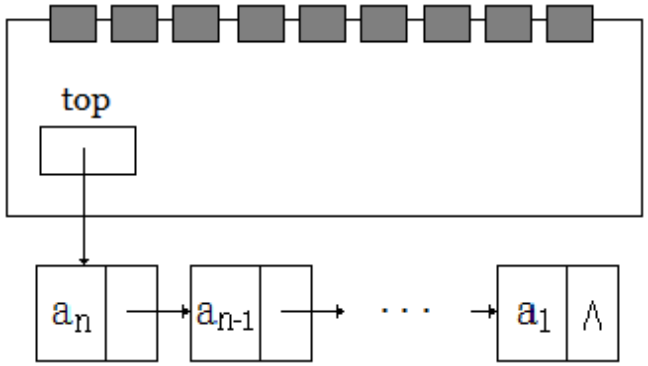
链序栈的析构函数

```
template<class ElemType>
LinkStack<ElemType>::~LinkStack()
{
        Clear();
}
```



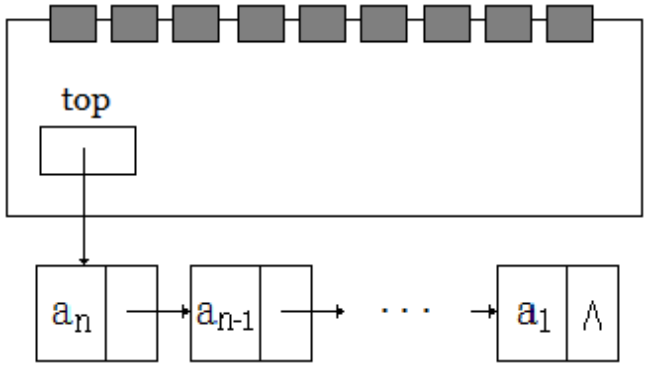
求链式栈的长度

```
template <class ElemType>
int LinkStack<ElemType>::GetLength() const
// 操作结果：返回栈中元素个数
{
   int count = 0;                    // 计数器
   Node<ElemType> *p;
   for (p = top; p != NULL; p = p->next)
         count++;                    // 统计链栈中结点数
   return count;
}
```



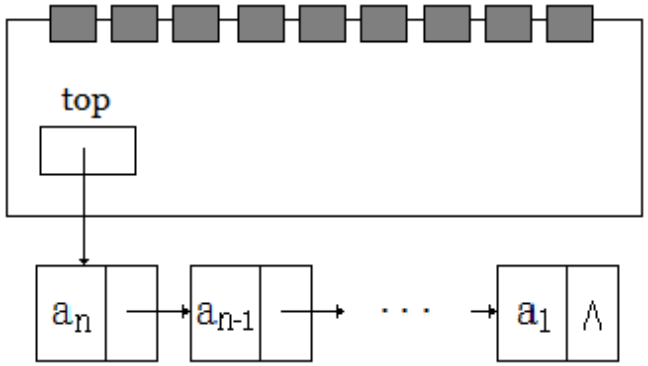
判断链式栈是否为空

```
template<class ElemType>
bool LinkStack<ElemType>::IsEmpty() const
{
        return top == NULL;
}
```



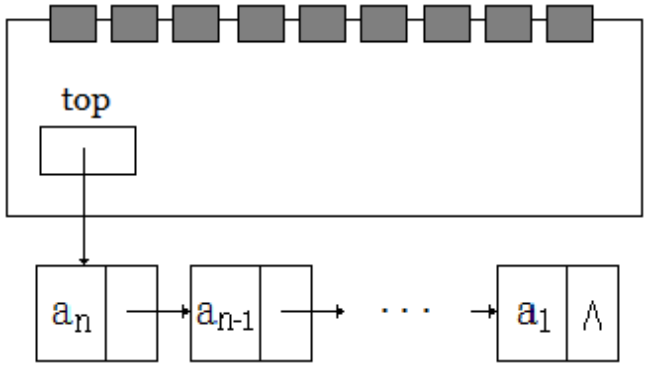
清空链式栈

```
template<class ElemType>
void LinkStack<ElemType>::Clear()
// 操作结果：清空栈
{
   Node<ElemType> *p;
   while (top != NULL) {
         p = top;
         top = top->next;
         delete p;
   }
}
```



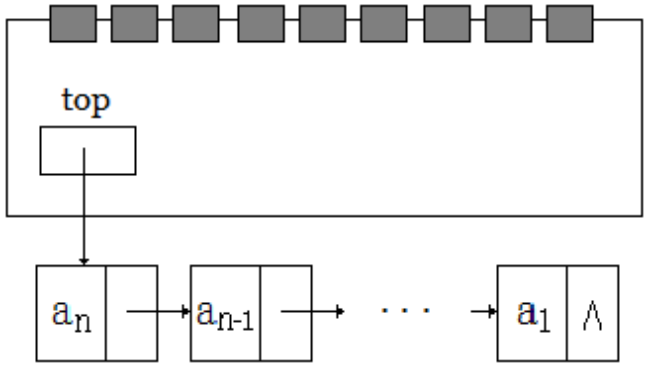
入栈

```
template<class ElemType>
Status LinkStack<ElemType>::Push(const ElemType e)
{
   Node<ElemType> *p = new Node<ElemType>(e, top);
   if (p == NULL)      // 系统内存耗尽
         return OVER_FLOW;
   else {              // 操作成功
         top = p;
         return SUCCESS;
   }
}
```



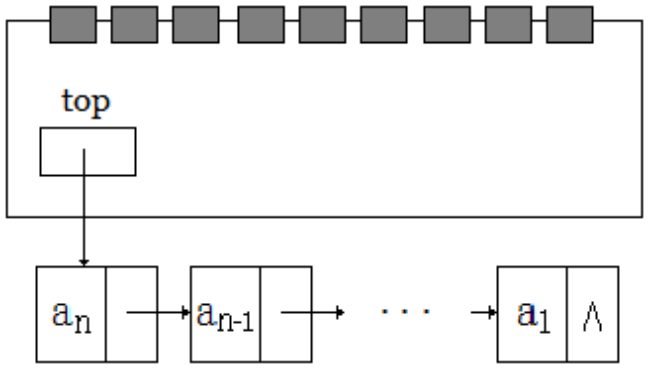
取栈顶元素

```
template<class ElemType>
Status LinkStack<ElemType>::Top(ElemType &e) const
{
   if(IsEmpty())                          // 栈空
         return UNDER_FLOW;
   else {
         e = top->data;                   // 用e返回栈顶元素
         return SUCCESS;
   }
}
```



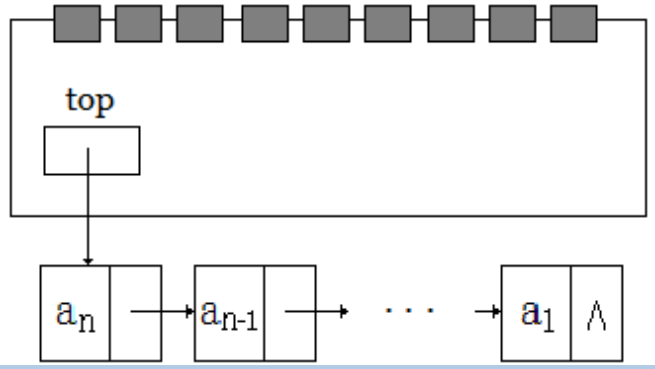
出栈

```
template<class ElemType>
Status LinkStack<ElemType>::Pop(ElemType &e)
{
   if  (IsEmpty())                          // 栈空
         return UNDER_FLOW;
   else {                                   // 操作成功
         Node<ElemType> *p = top;           // 保留原栈顶
         e = top->data;                     // 用e返回栈顶元素
         top = top->next;
         delete p;
         return SUCCESS;
   }
}
```



遍历链式栈

```
template <class ElemType>
void LinkStack<ElemType>::Traverse(void (*Visit)(const ElemType &)) const
// 从栈顶到栈底依次对栈的每个元素调用函数(*visit)访问
{
        Node<ElemType> *p;
        for (p = top; p != NULL; p = p->next)
                (*Visit)(p->data);
}
```



栈的应用——表达式求值

- 中缀表达式

(32 — 26) * 5 + 28 / 4

- 后缀表达式（逆波兰式）

32 26 — 5 * 28 4 / +

- 前缀表达式为：

+ * — 32 26 5 / 28 4

后缀表达式的计算

步	扫描项	操　作	栈中内
1		置空栈	空。
2	32	32 进栈	32。
3	26	26 进栈	32　26。
4	-	26、32 出栈，计算 32-26，将结果进栈	6。
5	5	5 进栈	6　5。
6	*	5、6 出栈，计算 6*5，将结果进栈	30。
7	28	28 进栈	30　28。
8	4	4 进栈	30　28　4。
9	/	4、28 出栈，计算 28/4，将结果进栈	30　7。
10	+	7、30 出栈，计算 30+7，将结果进栈	37。
11	#	37 出栈	空。

后缀表达式的计算

```
bool IsOperator(char ch)
{
   if (ch == '#' || ch == '(' || ch == '^' || ch == '*' || ch == '/'
                || ch == '+' || ch== '-' || ch == ')')
        return true;
    else
        return false;
};
```

后缀表达式的计算

```
void PostfixExpressionCalculation()  {
   LinkStack<double> opnd;
   char ch;
   double operand, first, second;
   cout << "输入后缀表达式, 以'#'号结束:";
   ch=GetChar();
   while (ch != '#') {
      if (isdigit(ch) || ch == '.') {
         cin.putback(ch);  cin >> operand;   opnd.Push(operand);
      }
      else if (!IsOperator(ch) || ch == '(' || ch ==')') {
         throw Error("表达式中有非法符号!");
      }
```

后缀表达式的计算

```
    else {              // ch为操作符
        if (opnd.Pop(second) == UNDER_FLOW)
            throw Error("缺少操作数!");
        if (opnd.Pop(first) == UNDER_FLOW)
            throw Error("缺少操作数!");
        opnd.Push(Operate(first, ch, second));
    }
    ch=GetChar();
  }
  if (opnd.Pop(operand) == UNDER_FLOW)  throw Error("缺少操作数!");
  if (!opnd.IsEmpty())      throw Error("缺少操作符!");
 cout << "表达式结果为：" << operand << endl;
};
```

中缀式转换为后缀式

表4-3 运算符间优先级关系表

op1 \ op2	＋	－	*	/	^	(	)	#
＋	＞(2)	＞(2)	＜(1)	＜(1)	＜(1)	＜(1)	＞(2)	＞(2)
－	＞(2)	＞(2)	＜(1)	＜(1)	＜(1)	＜(1)	＞(2)	＞(2)
*	＞(2)	＞(2)	＞(2)	＞(2)	＜(1)	＜(1)	＞(2)	＞(2)
/	＞(2)	＞(2)	＞(2)	＞(2)	＜(1)	＜(1)	＞(2)	＞(2)
^	＞(2)	＞(2)	＞(2)	＞(2)	＜(1)	＜(1)	＞(2)	＞(2)
（	＜(1)	＜(1)	＜(1)	＜(1)	＜(1)	＜(1)	＝(0)	×(-1)
）	＞(2)	＞(2)	＞(2)	＞(2)	＞(2)	×(-1)	＞(2)	＞(2)
#	＜(1)	＜(1)	＜(1)	＜(1)	＜(1)	＜(1)	×(-1)	＝(3)

判断两个相邻算符的优先级

```
int OperPrior(char op1, char op2)  {
   int prior;
   switch (op1) {
      case '+' :
      case '-' : if (op2 == '+' || op2 == '-' || op2 == ')' || op2 == '#')  prior=2;
                 else    prior=1;
                 break;
      case '*' :
      case '/' :
      case '^' :  if (op2 == '^' || op2 == '(')       prior=1;
                  else         prior=2;
                  break;
```

判断两个相邻算符的优先级

```
    case '(' :  if (op2 == ')')              prior=0;
                else if (op2 == '#')          prior=-1;
                else                          prior=1;
                break;
     case ')' :  if (op2 == '(')              prior=-1;
                 else                         prior=2;
                 break;
     case '#' : if (op2 == ')')               prior=-1;
                 else if (op2 == '#')         prior=3;
                 else                         prior=1;
                 break;
  }
  return prior;
}
```

中缀式转换为后缀式的算法

1、操作符栈初始化，将结束符"#"入栈，读入中缀表达式的首字符ch。

2、分三种情况重复执行以下步骤，直到ch ＝"#"，且栈顶的操作符op也是"#"时，停止循环。

其一：ch是数字或“.”，则说明当前读入的是操作数。

其二：ch不属于其一情况，也不是操作符。

其三：ch不属于上面二种情况，则说明ch是操作符。

若ch是“（”，并且前一项是“）”或操作数。

若op的优先级高于ch的优先级。

若op与ch之间不存在优先关系。

若op的优先级和ch的优先级相等，说明op为"（"，ch为"）"。

中缀式转换为后缀式的算法

```
void InfixInToPostfix()  {
    LinkStack<char>    optr;
    char ch;
    char priorChar;
    char op='#';
    double operand;
    int operandCount=0;
    optr.Push('#');
    priorChar='#';
    cout << "输入中缀表达式:";
    ch=GetChar();
    while (op != '#' || ch != '#') {
        if (isdigit(ch) || ch == '.')   {
            if (priorChar == '0' || priorChar == ')')
                throw Error("两个操作数之间缺少运算符!");
```

中缀式转换为后缀式的算法

```
    cin.putback(ch);                cin >> operand;
    cout << operand << "  ";  operandCount++;
    priorChar='0';                   ch=GetChar();
}
else if (!IsOperator(ch)) {
    throw Error("表达式中有非法符号!");  // 抛出异常
 }
 else {      // ch为操作符
    if (ch == '(' && (priorChar == '0' || priorChar == ')'))
        throw Error("'('前缺少操作符!");
    while (OperPrior(op, ch) == 2) {
        if (operandCount < 2)   throw Error("缺少操作数!");
        operandCount--;   optr.Pop(op);   cout << op << "  ";
        if (optr.Top(op) == UNDER_FLOW)
                throw Error("缺少操作符!");
     }
```

中缀式转换为后缀式的算法

```
            switch (OperPrior(op, ch)) {
                case -1 : throw Error("括号不匹配!");
                case 0 : optr.Pop(op);
                                if (optr.Top(op) == UNDER_FLOW)
                                        throw Error("缺少操作符!");
                              priorChar=ch;     ch=GetChar();
                               break;
                 case 1 : optr.Push(ch);      op=ch;
                              priorChar=ch;       ch=GetChar();
                               break;
                }
            }
    }
    if (operandCount != 1)      throw Error("缺少操作数!");
    cout << endl;
};
```

栈的两种存储结构的比较

栈的顺序存储结构是一种静态的存储结构，必须确定存储空间的大小，太大会造成存储空间的浪费，太小会因栈满而产生溢出。

栈的链接存储结构是一种动态的存储结构，因为结点是动态分配的，所以不会出现溢出问题。

4.2 队列

队列 (Queue)是另一种限定性的线性表，它只允许在表的一端插入元素，而在另一端删除元素，所以队列具有先进先出(Fist In Fist Out， 缩写为FIFO)的特性。

在队列中，允许插入的一端叫做队尾(rear)，允许删除的一端则称为队头(front)。

在队列中插入一个新元素的操作简称为进队或入队，新元素进队后就成为新的队尾元素；

从队列中删除一个元素的操作简称为出队或离队，当元素出队后，其后继元素就成为新的队头元素

队列

图4.6 队列的结构示意图

队列的示例

队列的基本操作

(1) 初始化：初始化一个空队列。

(2) 求长度：统计队列中数据元素的数目。

(3) 取队头元素：当队列不空时，取队头数据元素，并返回成功状态；否则返回不成功状态。

(4) 入队：在队尾插入一个新元素。如果能顺利插入元素，则返回插入成功；否则返回插入失败。

(5) 出队：当队列不为空时，删除队头元素，并返回出队成功；否则返回出队失败。

(6) 判断队列是否为空：如果队列为空，则返回值TRUE，否则返回值FALSE。

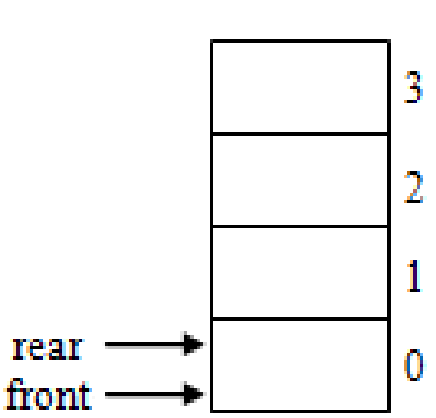
(7) 清空队列：将队列设置成空队列。

队列的存储结构

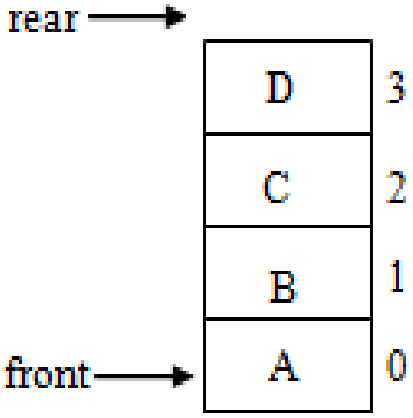
- 循环队列
- 链式队列

循环队列

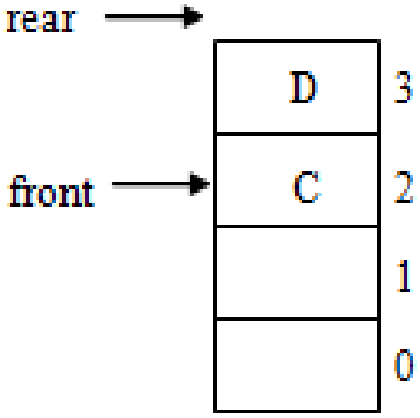
队列的顺序存储结构利用一个一维数组和两个指针来实现。一维数组用来存储当前队列中的所有元素，两个指针**front**和**rear**分别指向当前队列的队首元素和队尾元素，分别称为队首指针和队尾指针。



空队列



A、B、C、D 入队列



A、B 出队列

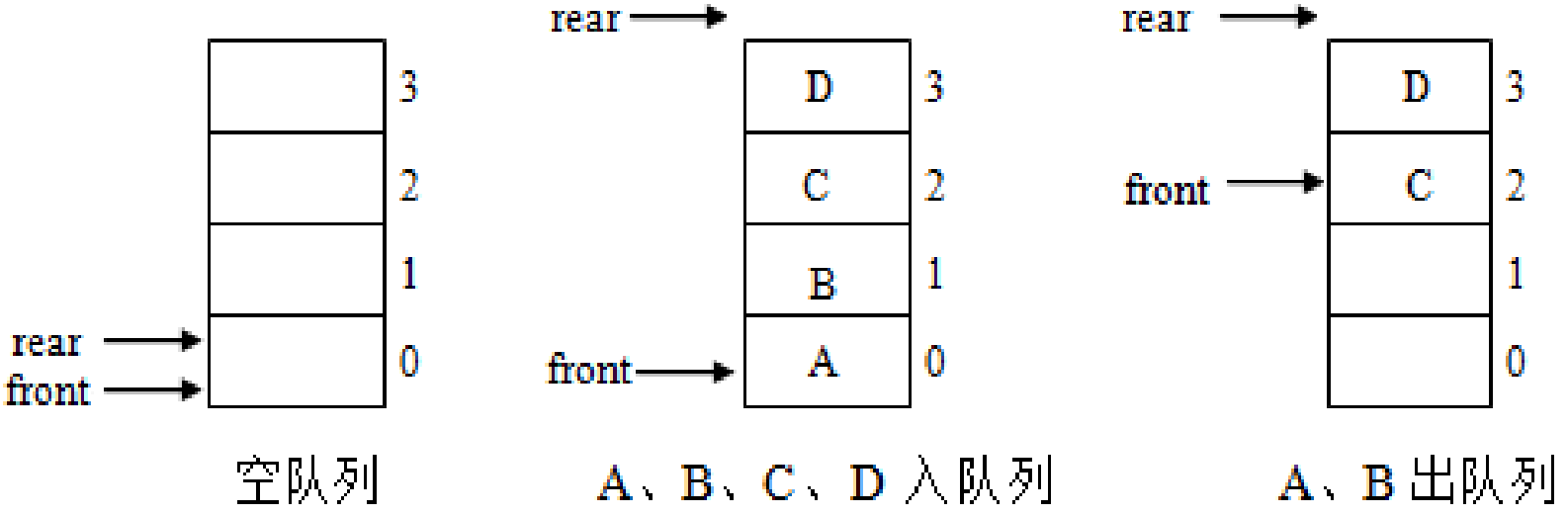
循环队列

队列的三种状态：

① 若顺序队列为空，则front=rear，队列的初始状态可设置为front=rear=0；

② 若顺序队列为满，则rear=MAXSIZE；

③ 若顺序队列非空非满，则MAXSIZE>rear>front



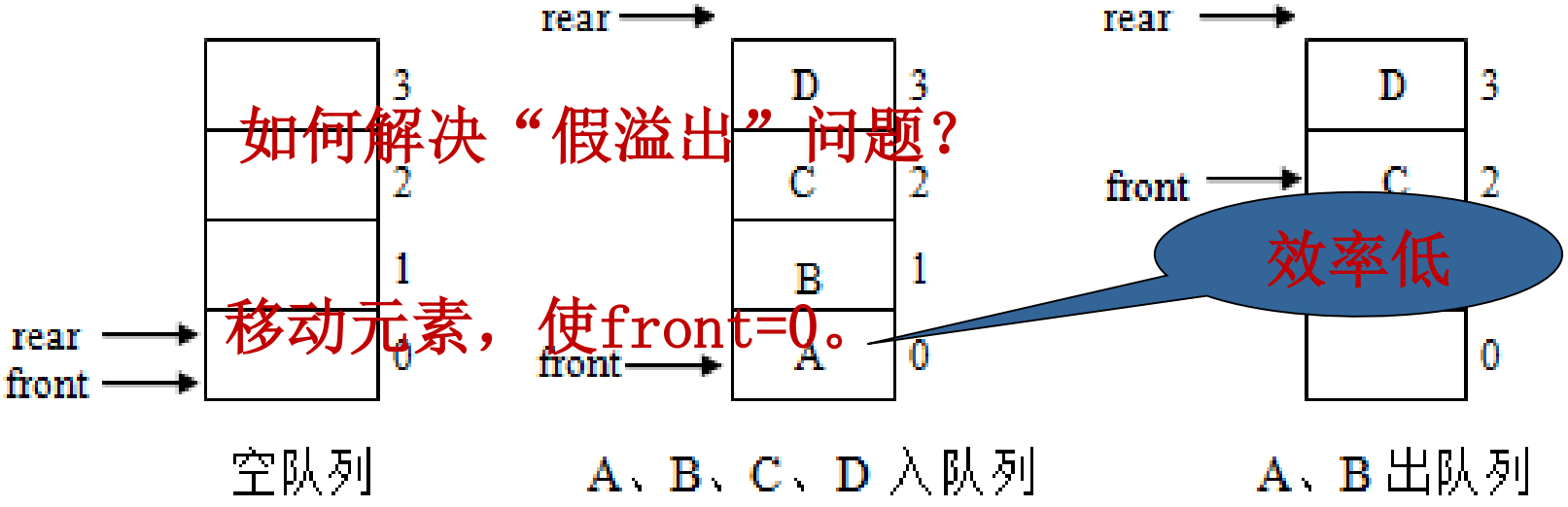
循环队列

同栈类似，顺序队列也有上溢和下溢现象：

① 当队空时，再进行出队就会产生“下溢”；

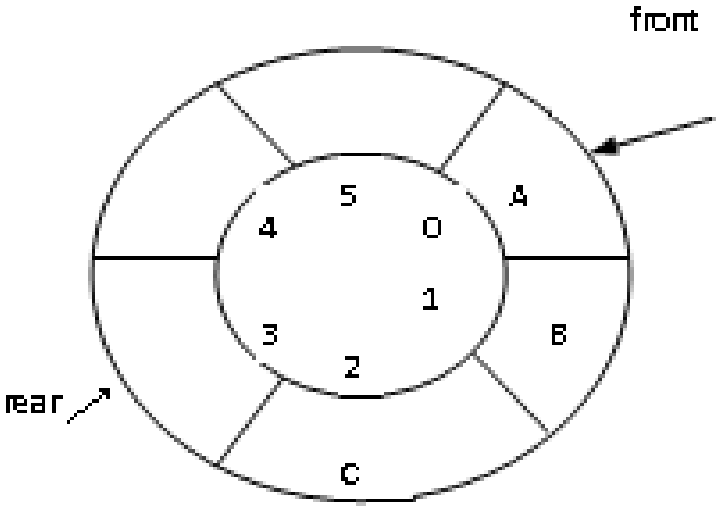
② 当队满时，再进行入队就会产生“上溢”；

③ 此外，顺序队列还存在“假上溢”现象，即rear=MAXSIZE并且front>0时。



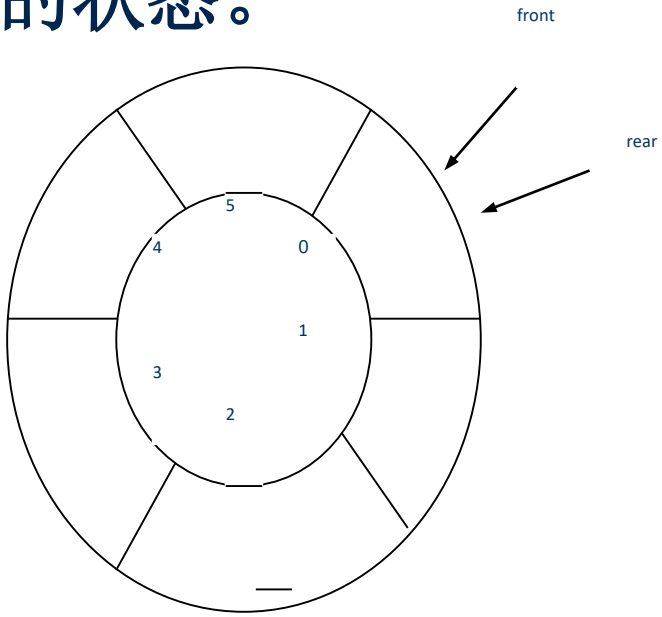
循环队列

将整个数组空间变成一个首尾相接的圆环，即把elems [0]放在elems [MAXSIZE-1]之后，我们称这种数组为循环数组。用循环数组表示的队列称为循环队列。

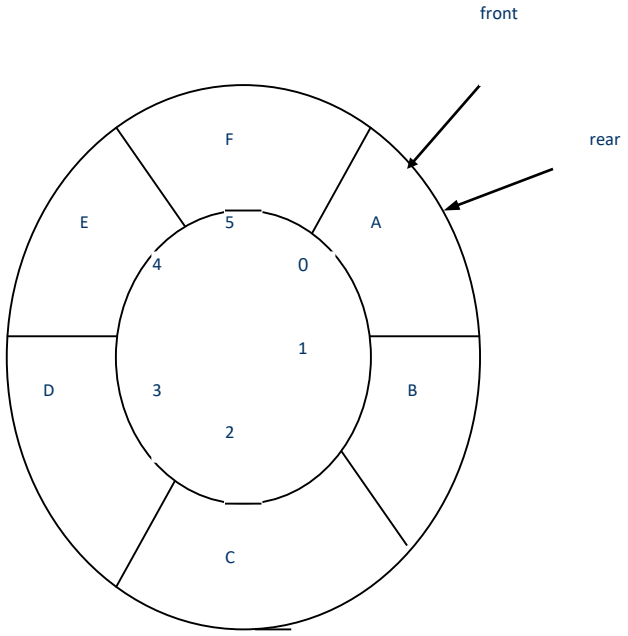


循环队列

这种循环队列出现新的问题：无法区分队列空和队列满的状态。



(a) 队列空



(b) 队列满

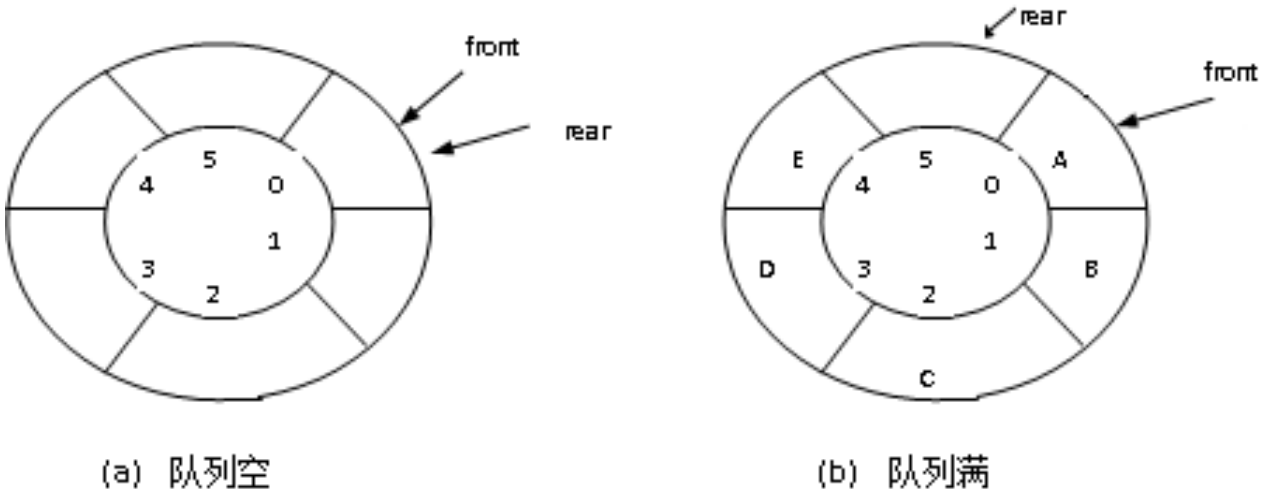
循环队列

队列的三种状态(设置一个空闲单元不用)：

① 队满的条件：(rear+1)%MAXSIZE == front;

② 队空的条件：rear == front;

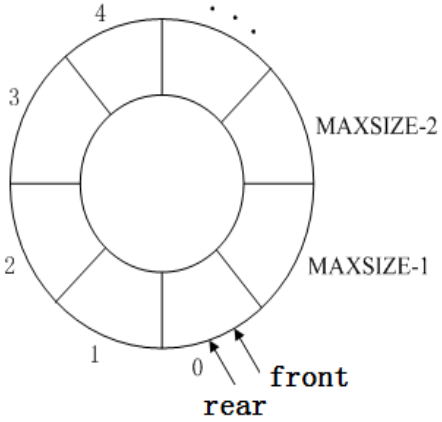
③ 其它为一般情况（非空、非满）



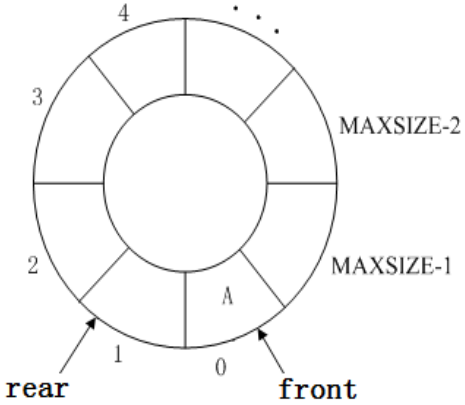
(a) 队列空　　(b) 队列满

图 4-10 循环队列的队列空和队列满

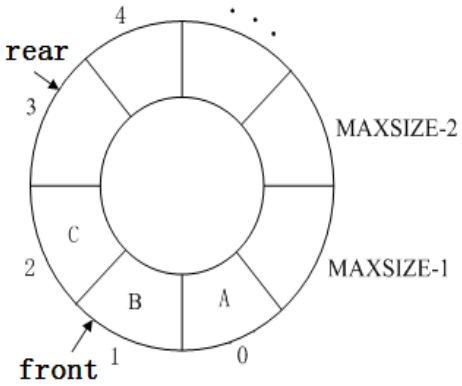
循环队列



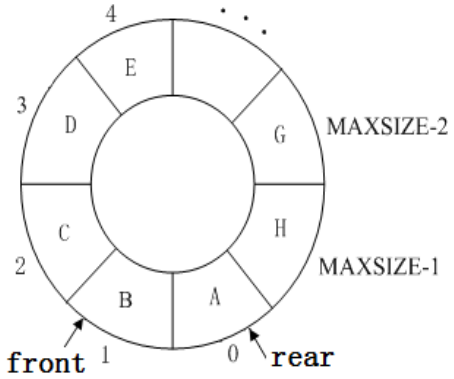
(a) 循环队列为空队列



(b) 元素A入队



(c) 元素A出队及B，C入队

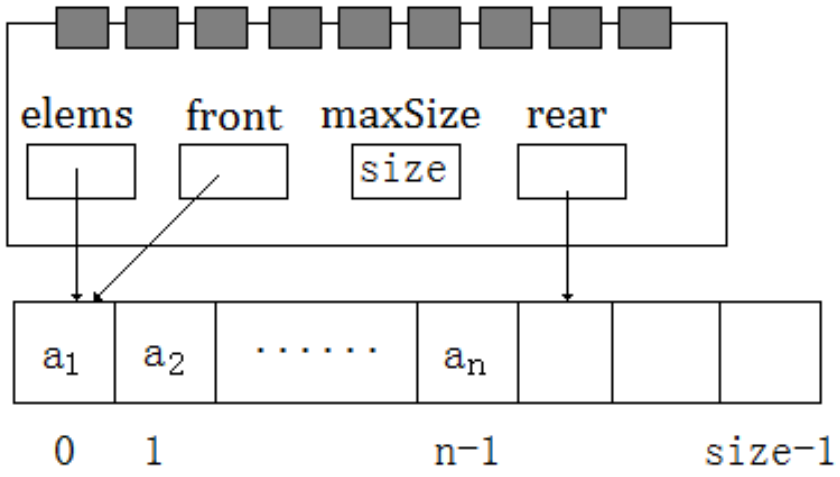


(d) 循环队列为满队

图4.7 设空闲单元的循环队列进行入队和出队操作首尾指针变化情况示意图

循环队列的类模板定义

```
template<class ElemType>
class SeqQueue {
protected:
        int front, rear;// 队头队尾指针
        int maxSize; // 队列容量
        ElemType *elems; // 元素存储空间
```

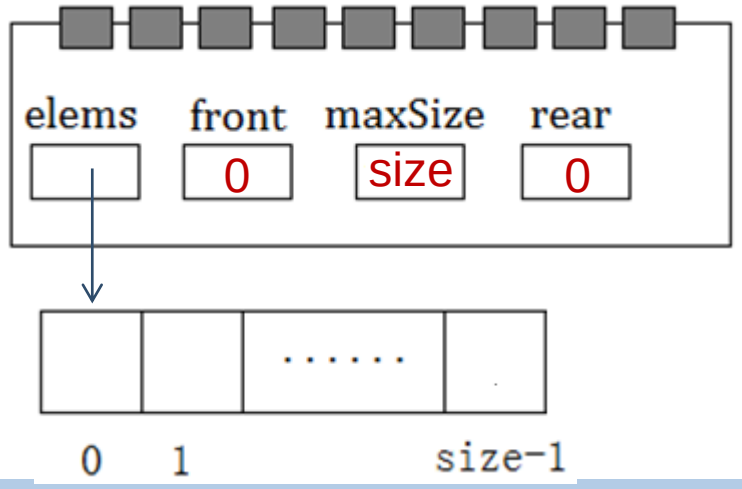


循环队列的类模板定义

```
public:
        SeqQueue(int size = DEFAULT_SIZE);
        virtual ~SeqQueue();
        int GetLength() const;
        bool IsEmpty() const;
        void Clear();
        Status DelQueue(ElemType &e);
        Status GetHead(ElemType &e) const
        Status EnQueue(const ElemType e);
       ……
};
```

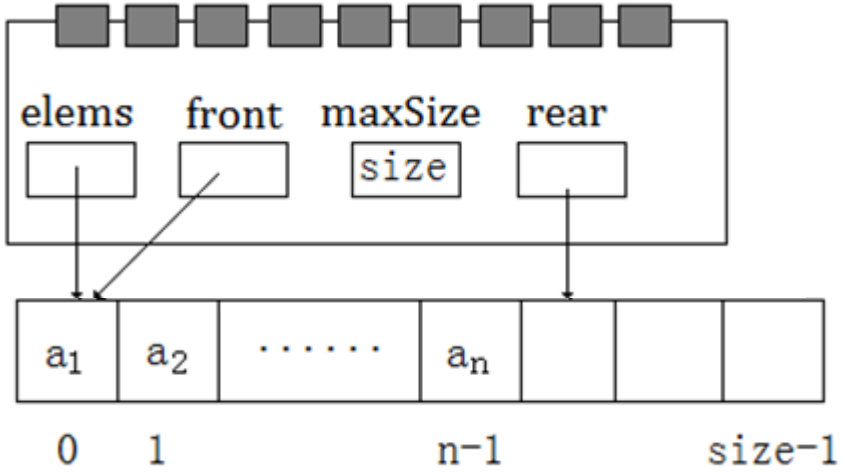
循环队列的构造函数

```
template<class ElemType>
SeqQueue<ElemType>::SeqQueue(int size){
        maxSize = size;
        if (elems != NULL) delete []elems;
        elems = new ElemType[maxSize];
        rear = front = 0;
}
```



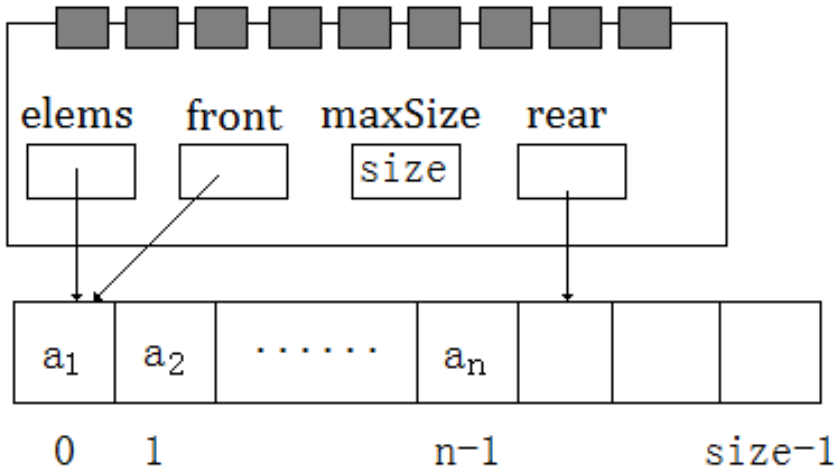
循环队列的析构函数

```
template <class ElemType>
SeqQueue<ElemType>::~SeqQueue()
{
        delete []elems;
}
```



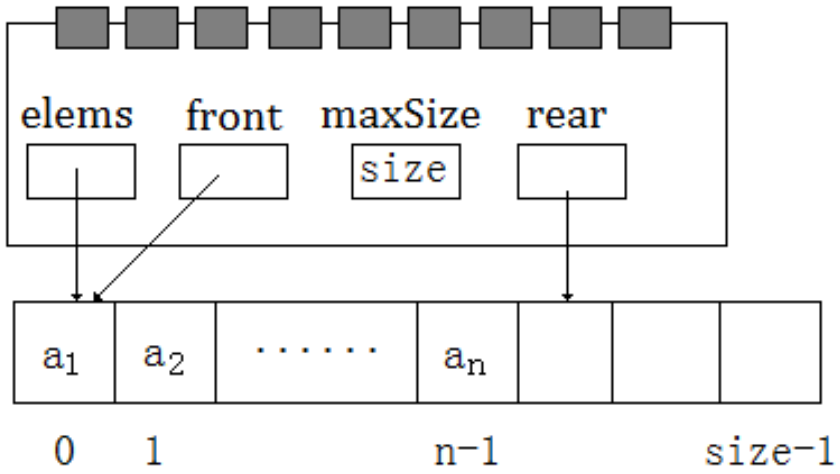
求循环队列的长度

```
template<class ElemType>
int SeqQueue<ElemType>::GetLength() const
{
        return (rear - front + maxSize) % maxSize;
}
```



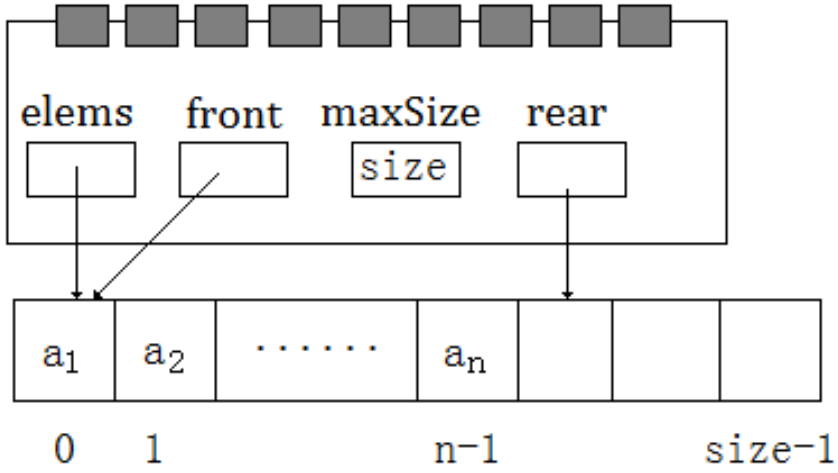
判断循环队列是否为空

```
template<class ElemType>
bool SeqQueue<ElemType>::IsEmpty() const
{
        return     rear == front;
}
```



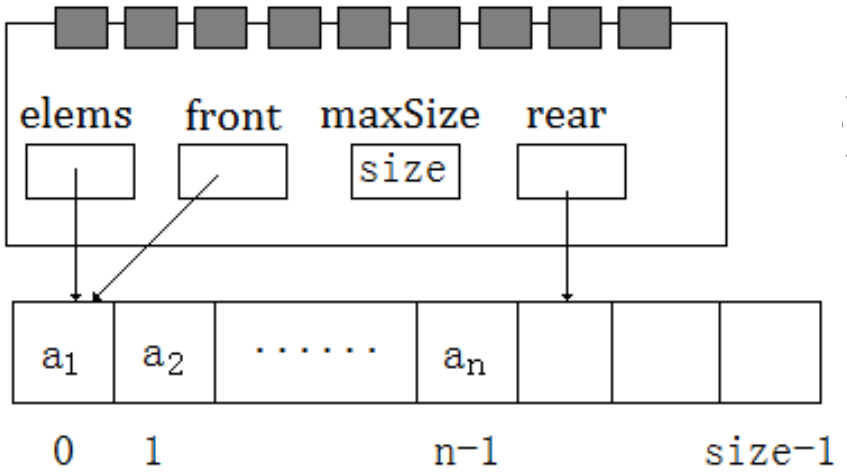
清空循环队列

```
template<class ElemType>
void SeqQueue<ElemType>::Clear()
{
        rear = front = 0;
}
```



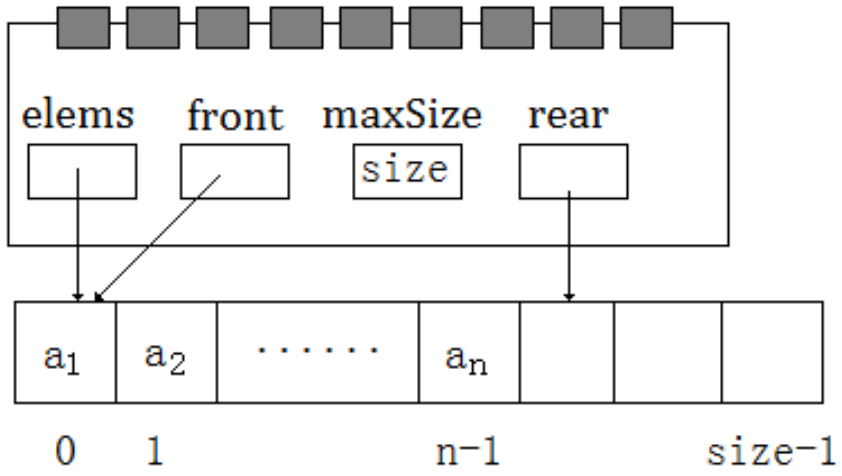
入队

```
template<class ElemType>
Status SeqQueue<ElemType>                                    )
{
        if ((rear + 1) % maxSize
                return OVER_FL
        else    {
                elems[rear] = e;
                rear = (rear + 1) % maxSize;
                return SUCCESS;
        }
}
```



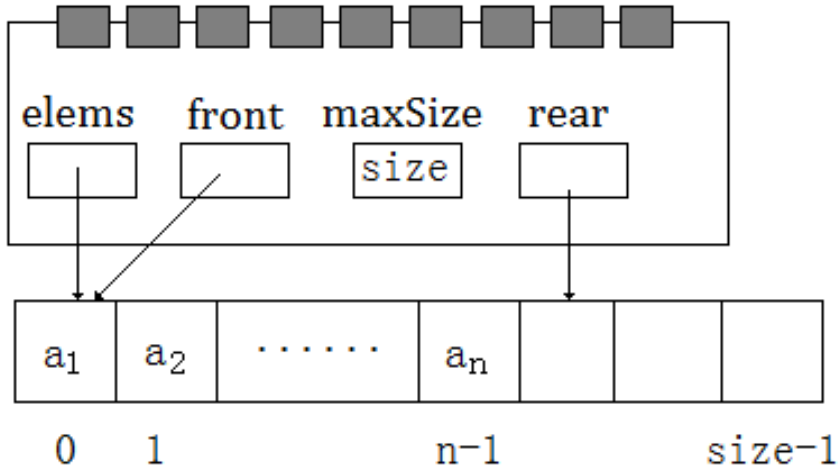
取队头元素

```
template<class ElemType>
Status SeqQueue<ElemType>::G
{
        if (!Empty())   {
                e = elems[front];
                return SUCCESS;
        }
        else
                return UNDER_FLOW;
}
```



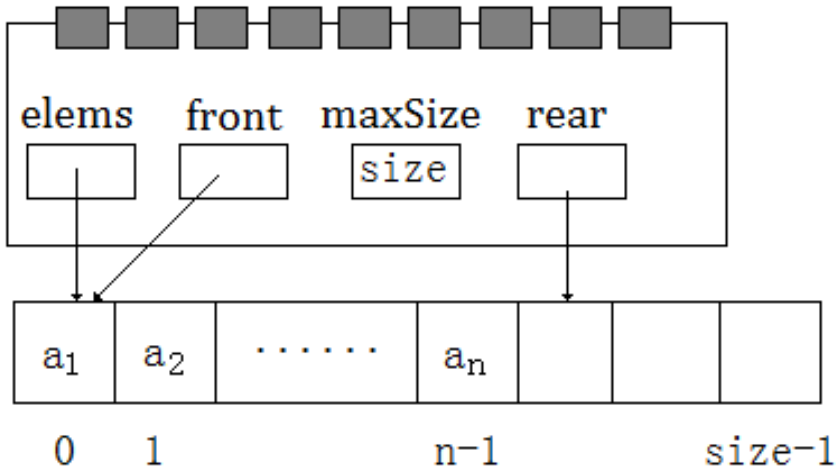
出队

```
template<class ElemType>
Status SeqQueue<ElemType>::
{
        if (!IsEmpty())         {
                e = elems[front];
                front = (front + 1) % maxSize;
                return SUCCESS;
        }
        else
                return UNDER_FLOW;
}
```



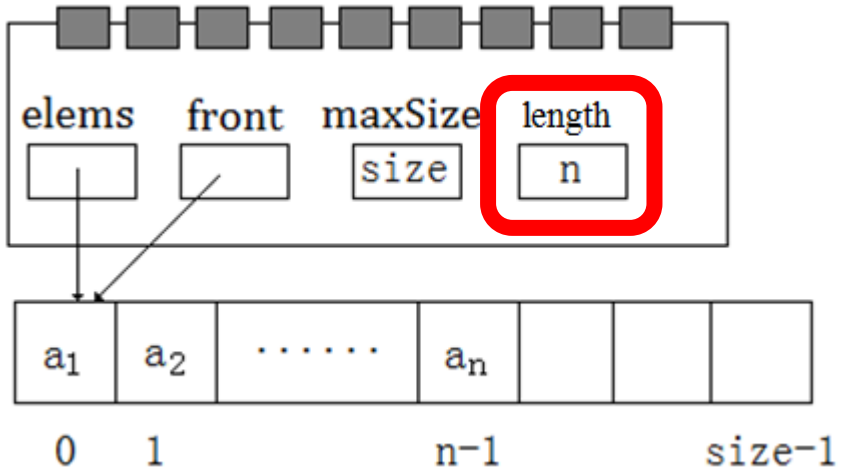
遍历循环队列

```
template <class ElemType>
void SeqQueue<ElemType>::Traverse(void
(*Visit)(const ElemType &)) const
{
        for (int i = front; i != rear; i = (i + 1) % maxSize)
                (*Visit)(elems[i]);
}
```



顺序队列的另一种定义

```
template<class ElemType>
class SeqQueue {
protected:
        int front, length;// 队头队尾指针
        int maxSize; // 队列容量
        ElemType *elems; // 元素存储空间
```



链式队列

队列的链接存储结构是用一个单链表存放队列元素的。队列的链接存储结构称为链式队列。由于队列只允许在表尾进行插入操作、在表头进行删除操作，因此，链队需设置两个指针：队头指针front和队尾指针rear，分别指向单链表的第一个结点(表的头结点)和最后一个结点(队尾结点)。

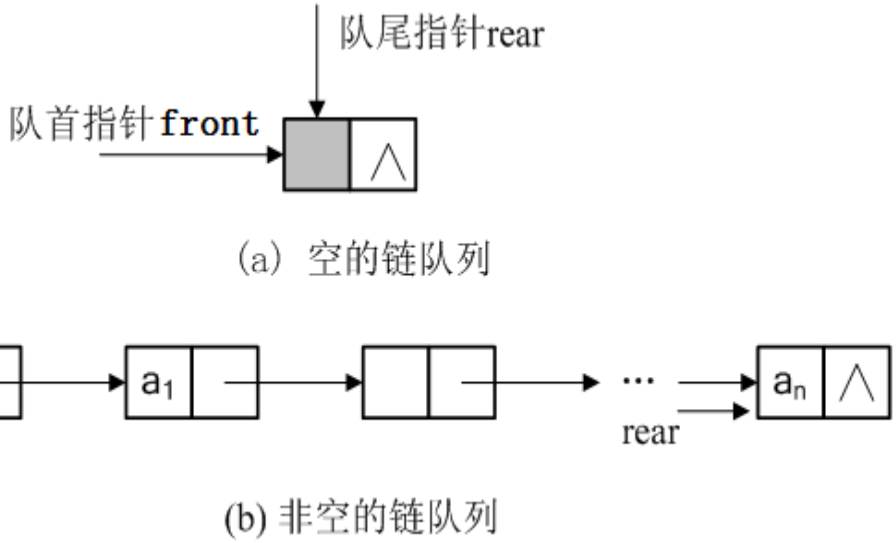
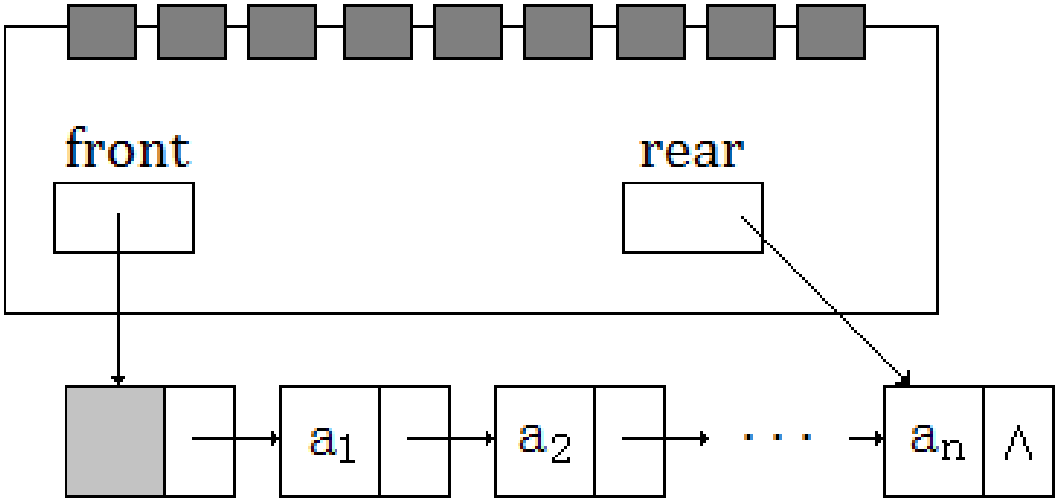


图4.8 带头结点的链接队列的结构示意图

链式队列的类模板定义

```
template<class ElemType>
class LinkQueue {
protected:
        Node<ElemType> *front, *rear; // 队头队尾指指
```

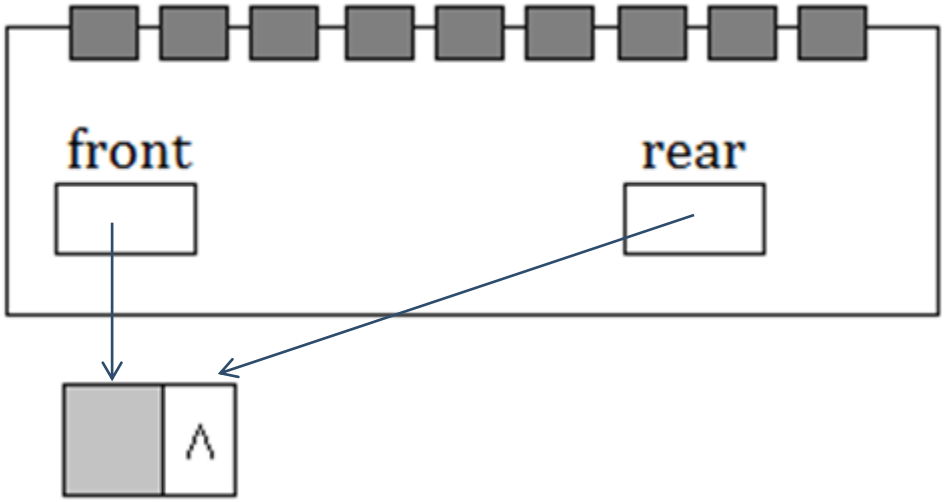


链式队列的类模板定义

```
public:
        LinkQueue();
        virtual ~LinkQueue();
        int GetLength() const;
        bool IsEmpty() const;
        void Clear();
        Status DelQueue(ElemType &e);
        Status GetHead(ElemType &e) const;
        Status EnQueue(const ElemType e);
        ……
};
```

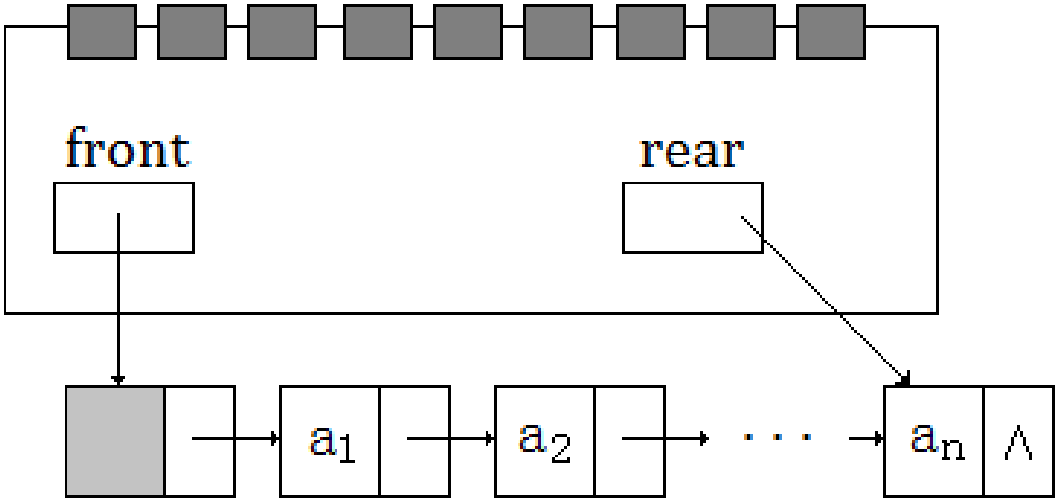
链式队列的构造函数

```
template<class ElemType>
LinkQueue<ElemType>::LinkQueue()
{
        rear = front = new Node<ElemType>;
}
```



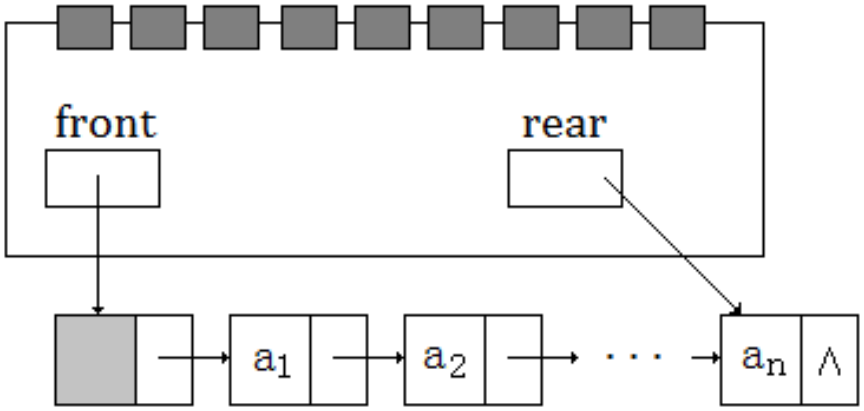
链式队列的析构函数

```
template<class ElemType>
LinkQueue<ElemType>::~LinkQueue()
{
        Clear();
        delete front;
}
```



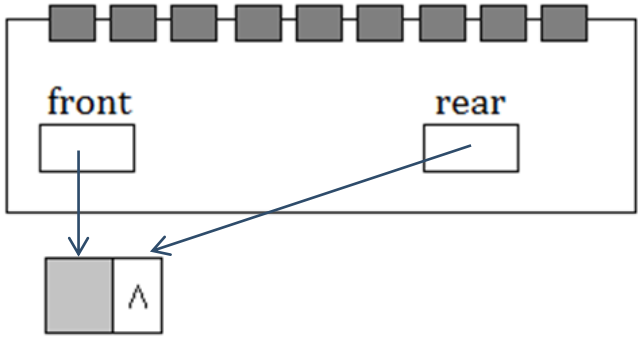
求链式队列的长度

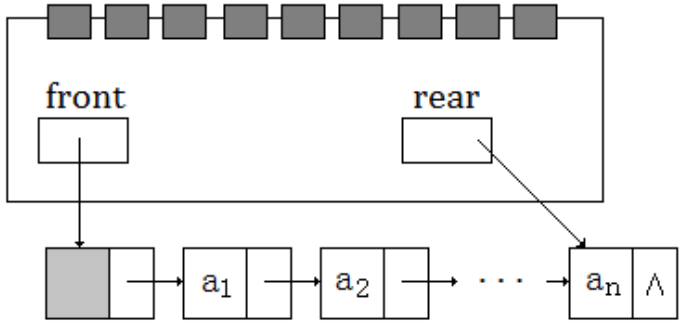
```
template<class ElemType>
int LinkQueue<ElemType>::GetLength() const
{
        int count = 0;
        Node<ElemType> *p;
        for (p = front->next; p != NULL; p = p->next)
                count++;
        return count;
}
```



判断链式队列是否为空

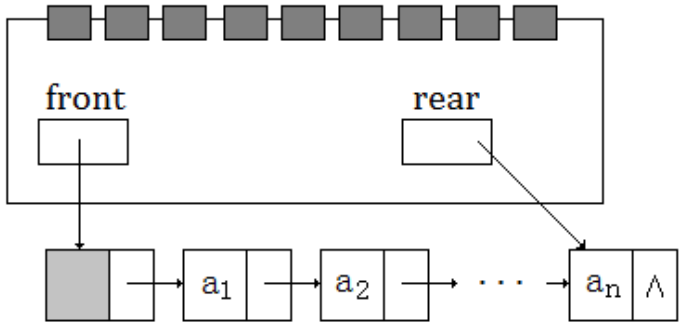
```
template<class ElemType>
bool LinkQueue<ElemType>::IsEmpty() const
{
        return rear == front;
}
```





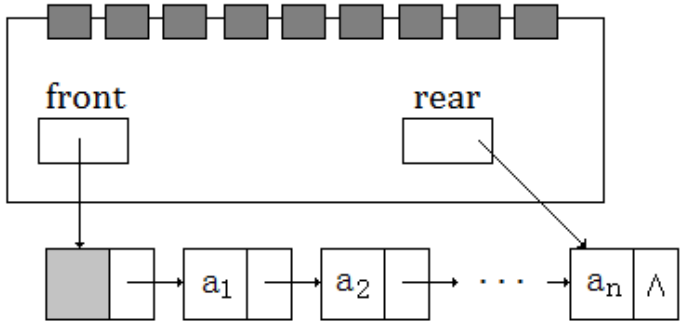
清空链式队列

```
template<class ElemType>
void LinkQueue<ElemType>::Clear()
{
        Node<ElemType> *p = front->next;
        while (p != NULL)    {
                front->next = p->next;
                delete p;
                p = front->next;
        }
        rear = front;
}
```



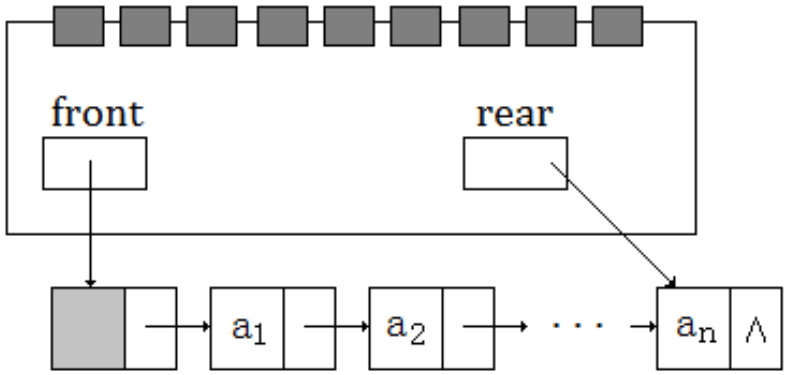
入队

```
template<class ElemType>
Status LinkQueue<ElemType>::EnQueue(const ElemType e)
{
        Node<ElemType> *p;
        p = new Node<ElemType>(e);
        if (p) {
                rear->next = p;
                rear = rear->next;
                return SUCCESS;
        }   else
                return OVER_FLOW;
}
```



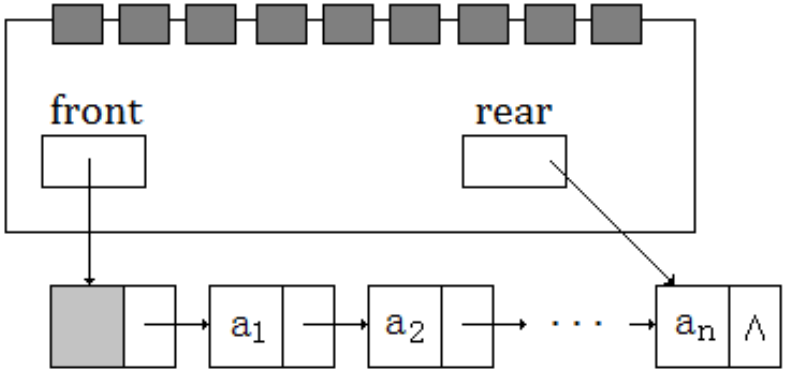
取队头元素

```
template<class ElemType>
Status LinkQueue<ElemType>::GetHead(ElemType &e) const
{
        if (!IsEmpty())         {
                e = front->next->data;
                return SUCCESS;
        }
        else
                return UNDER_FLOW;
}
```



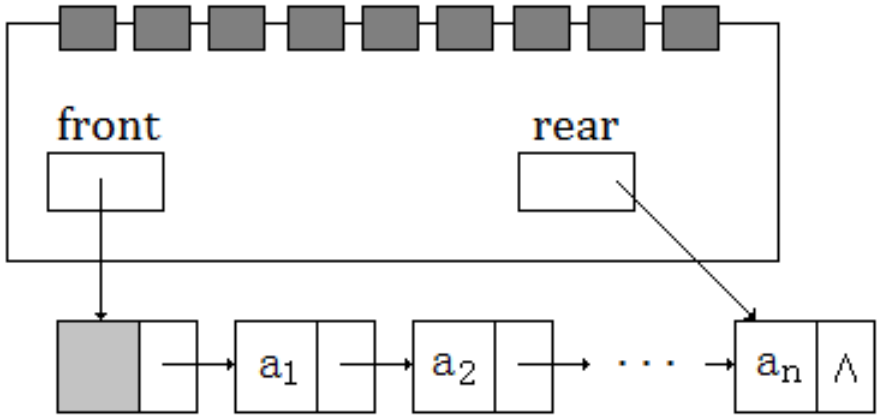
出队

```
template<class ElemType>
Status LinkQueue<ElemType>::DelQueue(ElemType &e){
        if (!IsEmpty())          {
           Node<ElemType> *p = front->next;
           e = p->data;          front->next = p->next;
           if (rear == p)        rear = front;
           delete p;
           return SUCCESS;
        }
        else
           return UNDER_FLOW;
}
```



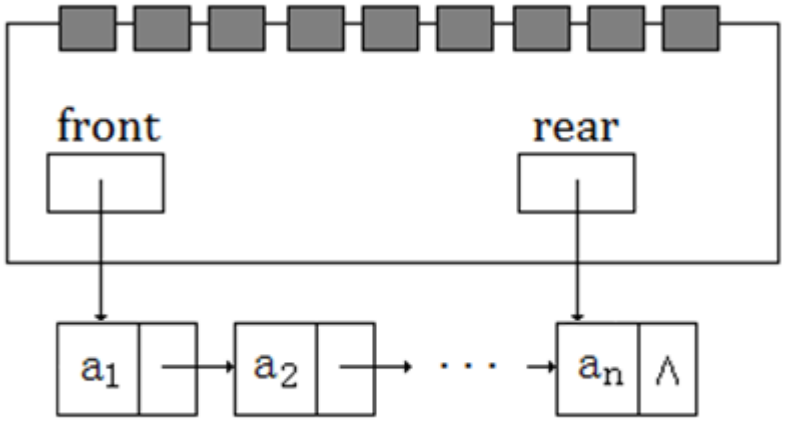
遍历链式队列

```
template <class ElemType>
void LinkQueue<ElemType>::Traverse(void (*Visit)(const ElemType &)) const
{
        Node<ElemType> *p;
        for (p = front->next; p != NULL; p = p->next)
                (*Visit)(p->data);
}
```



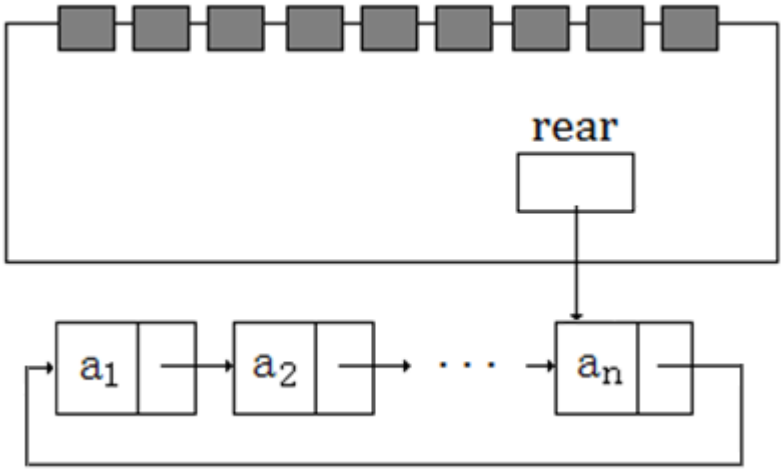
链式队列的变化1

```
template<class ElemType>
class LinkQueue {
protected:
        Node<ElemType> *front, *rear; // 队头队尾指指
```



链式队列的变化2

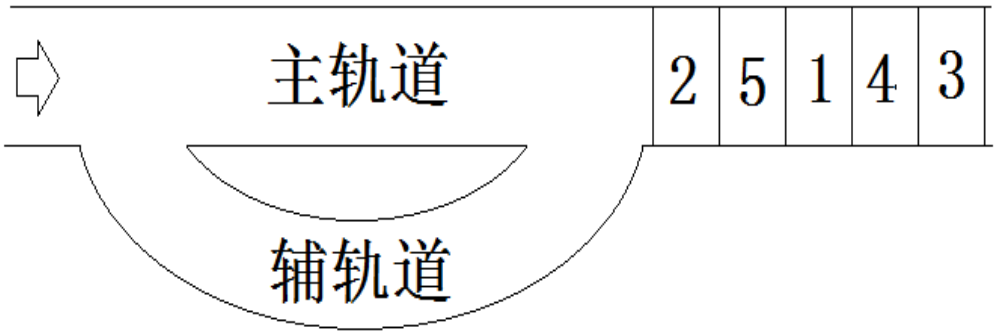
```
template<class ElemType>
class LinkQueue {
protected:
        Node<ElemType> *rear; // 队尾指指
```



队列的应用

车厢调度

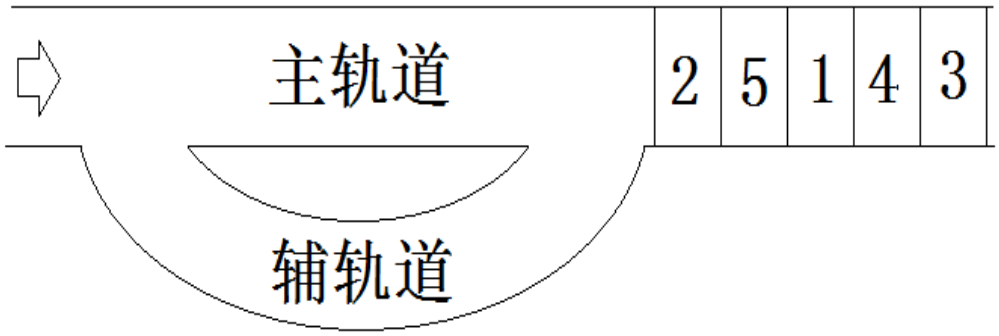
一个由2条平行铁轨组成铁路调度系统下图所示。其中辅助铁轨用于对车厢次序进行调整，它在主铁轨中间，把主铁轨分成左、右两部分。主铁轨左边的车厢可以直接开到主铁轨右边；也可以从主铁轨左边进入辅助铁轨；辅助铁轨上的车厢只可以进入主铁轨右边。



队列的应用

车厢调度

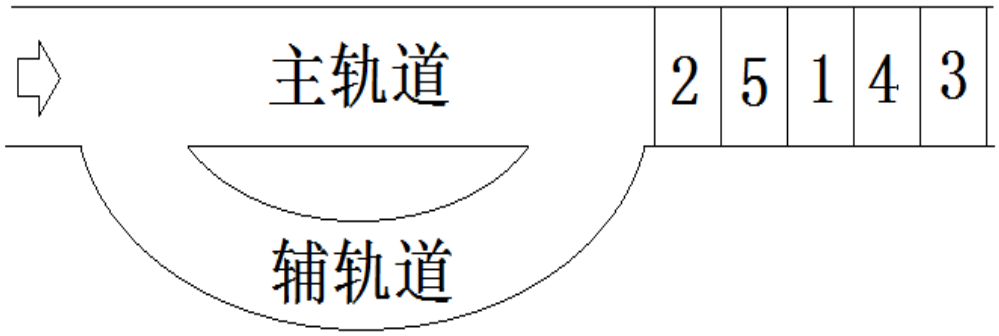
现在有n节火车车厢，编号为1、2、…、n，在主铁轨的左边依次排列，要求通过这个调度系统，在主铁轨的右按规定的编号次序出站（例如：有5节车厢以1、2、3、4、5的顺序依次进入，要求以3、4、1、5、2的顺序出站）。



队列的应用

1．数据结构选择

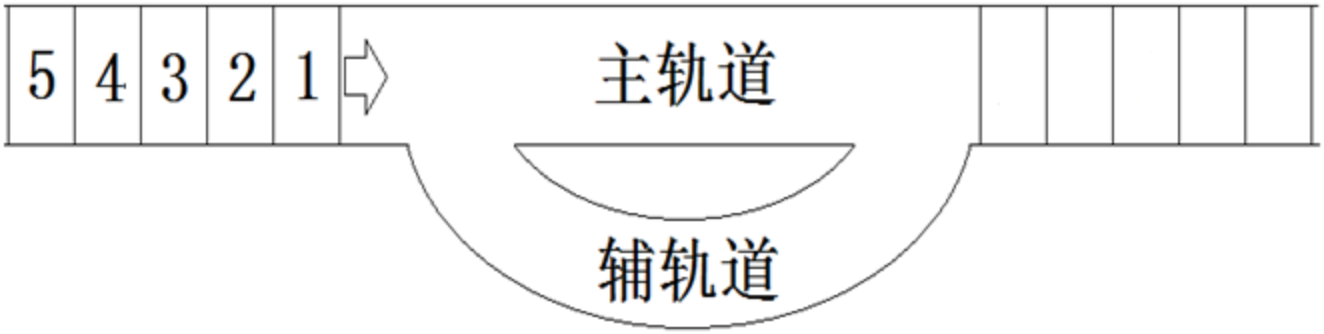
由于n节车厢在主铁轨左边按1、2、…、n的顺序排列，所以可以简单的用一个计数器表示；n节车厢在主铁轨右边的出站顺序可以用一个线性表表示，也可以依次输入，在此通过循环依次从输入流获得。在这个调度系统中主要考虑辅轨道的实现。由于辅轨道具有FIFO特征，所以可以用一个队列进行模拟。



队列的应用

2．算法思想

设置一个主循环按车厢在主轨道右边的出站顺序依次读入当前要出站的车厢号d，对当前出站的车厢号d进行执行如下步骤：



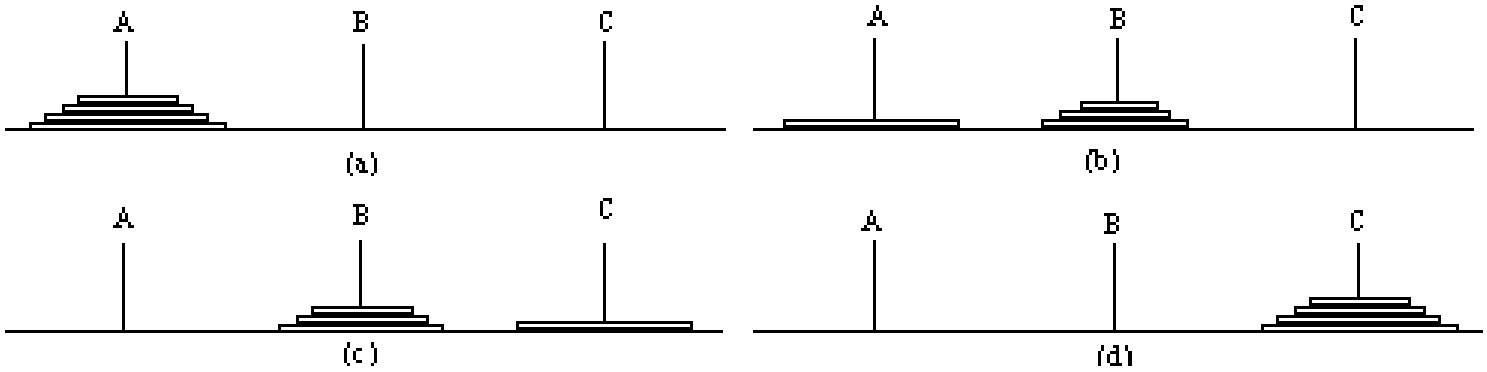
车厢调度

```
int main(void)
{
    LinkQueue<int> qa;
    int n, x, d, No;
    cout << "输入车厢数：";     cin >> n;
    No = 1;
    cout << "输入 " << n << " 节车厢的出站顺序：";
    for (int i = 1; i <= n; i++){
        cin >> d;
        if (qa.GetHead(x) == SUCCESS && x == d) {
            cout << "第 " << x << " 号车厢从辅轨道进入主轨道右边." << endl;
            qa.DelQueue(x);
        }
```

车厢调度

```
else if (No <= d) {
    while (No <= n && No < d) {
        cout << "第" << No << "号从主轨道左边进入辅轨道." << endl;
        qa.EnQueue(No++);
    }
    if (No == d) {
        cout << "第" << No << "号从主轨道左边进入主轨道右边." << endl;
        No++;
    }
  }
  else   break;
}
if (qa.IsEmpty())       cout << "调度完成." << endl;
else       cout << “调度无法完成.” << endl;
system(“PAUSE”);  return 0;
}
```

4.3 递归



4. 3 递归

```
# include <iostream.h>
void Hanoi (int n, char A, char B, char C) {
   //把n 个盘子从A针借助B针移到C针上
      if (n == 1 )              //递归出口，一个盘子直接移动
            cout<<"Move"<<A<<"to"<<C<<endl;
      else  {
            Hanoi (n-1, A, C, B);
            cout<<"Move"<<A<<"to"<<C<<endl;
            Hanoi (n-1, B, A, C);
      }
}
```

递归过程与递归工作栈

一般情况，对一个非递归函数的调用，在函数调用前要保存以下三方面的信息：

（1）返回地址。

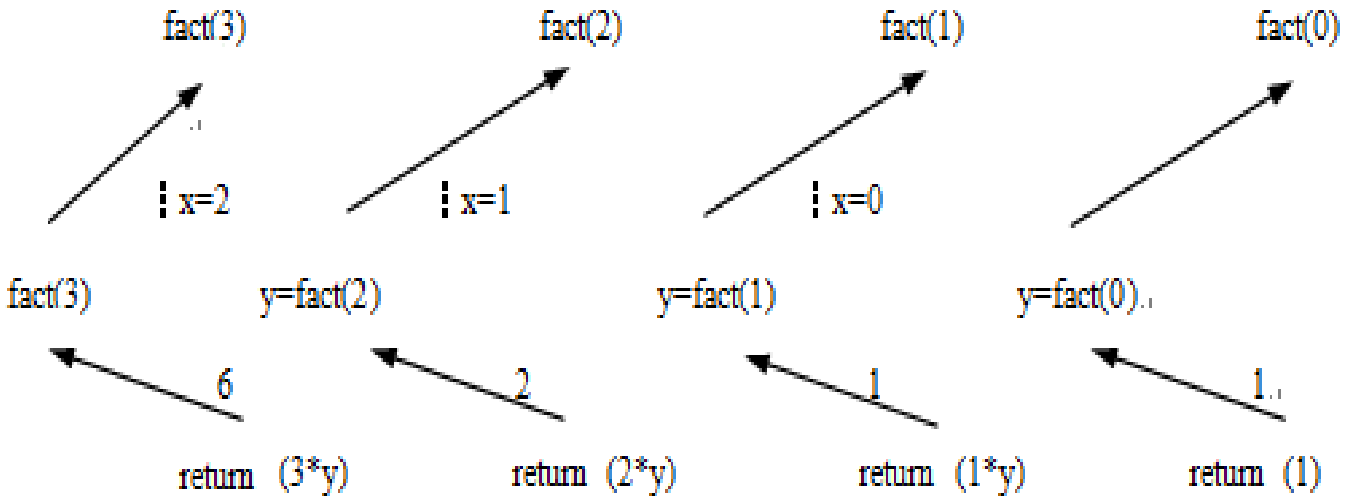
（2）本函数调用时，与形参结合的实参值。包括函数名和函数参数。

（3）本函数的局部变量值。

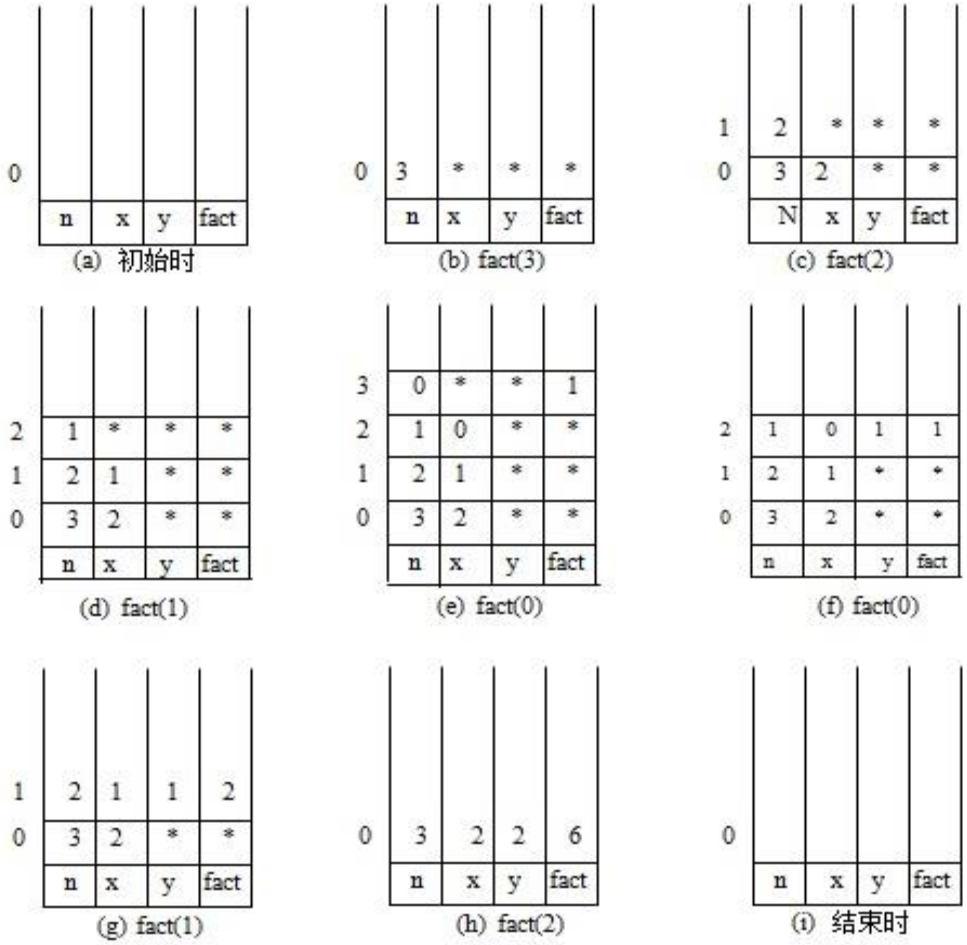
$$n! = \begin{cases} 1 & \text{当}n=0\text{时} \\ n*(n-1)! & \text{当}n>0\text{时} \end{cases}$$

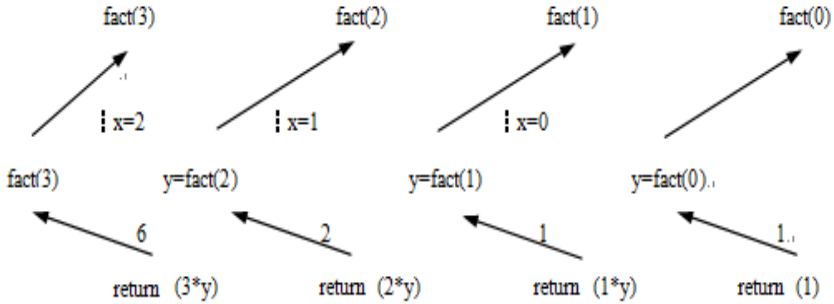
递归过程与递归工作栈

$$n! = \begin{cases} 1 & \text{当}n=0\text{时} \\ n*(n-1)! & \text{当}n>0\text{时} \end{cases}$$



递归过程与递归工作栈


0 n x y fact
(a) 初始时
0 3 * * *
(b) fact(3)
1 2 * * *
0 3 2 * *
N x y fact
(c) fact(2)
2 1 * * *
1 2 1 * *
0 3 2 * *
(d) fact(1)
3 0 * * 1
2 1 0 * *
1 2 1 * *
0 3 2 * *
(e) fact(0)
2 1 0 1 1
1 2 1 * *
0 3 2 * *
(f) fact(0)
1 2 1 1 2
0 3 2 * *
(g) fact(1)
0 3 2 2 6
(h) fact(2)
(i) 结束时


fact(3)
fact(2)
fact(1)
fact(0)
x=2
x=1
x=0
fact(3)
y=fact(2)
y=fact(1)
y=fact(0)
6
2
1
1
return (3*y)
return (2*y)
return (1*y)
return (1)

消除递归

（1）对于尾递归和单向递归的算法，可用循环结构的算法来替代。

（2）自己用栈来模拟系统运行时的栈，保存有关的信息，从而用非递归算法来模拟递归算法。

尾递归的消除

```
long fact (int n) {
    if (n == 0)
        return 1;
    else
        return  n * fact(n-1);
}
```

```
long fact(int n){
    int product=1;
    for (int i=1; i<=n; i++)
        product= product*i
    return product
}
```

单向递归的消除

```
long Fib(int n)  {
    if (n == 0 || n ==1 )       return  n;
    else       return Fib(n-1)+Fib(n-2);
}
```

```
long Fib (int n)    {
    if (n == 0 || n == 1) return  n;
    long twoback =0, oneback =1, current;
    for (int i=2; i<= n; i++) {
        current=twoback+oneback;
        twoback=oneback;
        oneback=current;
    }
    return current;
}
```

用栈来模拟递归算法

```
struct Datatypes {
    short int retAddr;    // 模仿返回地址
    int nDisk;            // 参数n
    char SourcePeg;       // 参数A
    char AuxPeg;          // 参数B
    char DestPeg;         // 参数C
};
```

用栈来模拟递归算法

```
void SimuTowers(int n, char A, char B, char C)  {
    Datatypes  currArea;        // 当前工作区
    LinkStack<Datatypes> s; // 模拟系统运行时的堆栈
    char temp;
    short int i;
    currArea.retAddr =1;        currArea.nDisk =n;
    currArea.SourcePeg =A;  currArea.AuxPeg =B;
    currArea.DestPeg =C;     s.Push(currArea);
start:
    if (currArea.nDisk==1){
      cout << "Move Disk 1 from Peg " <<    currArea.SourcePeg
           << " to Peg " << currArea.DestPeg << endl;
      i=currArea.retAddr;
      s.Pop(currArea);     // 出栈恢复当前工作区
```

用栈来模拟递归算法

```
    switch(i) {
       case 1: goto L1;
       case 2: goto L2;
       case 3: goto L3;
    }
}
s.Push(currArea);                    // 当前工作区入栈
currArea.nDisk--;
temp=currArea.AuxPeg;
currArea. AuxPeg =currArea.DestPeg;
currArea.DestPeg =temp;
currArea.retAddr =2;
goto start;
```

用栈来模拟递归算法

```
L2: cout << "Move Disk " << currArea.nDisk << " from Peg "
          << currArea.SourcePeg << " to Peg "
          << currArea.DestPeg << endl;
    s.Push(currArea); currArea.nDisk--; temp=currArea.SourcePeg;
    currArea.SourcePeg=currArea.AuxPeg;
    currArea.AuxPeg=temp;     currArea.retAddr=3;
    goto start;
L3:  i=currArea.retAddr;     s.Pop(currArea);
    switch(i){
       case 1: goto L1;
       case 2: goto L2;
       case 3: goto L3;
    }
L1:     return;
}
```


祝你成功！