第3章 栈和队列

本章的基本内容是：

3.1 栈

3.2 栈的应用

3.3 队列

3.4 队列的应用

3.1 栈

问题1：在非常拥挤的情况下，汽车司机应该如何组织人们上、下车？（假设通勤车只有一个门）

如果编程模拟通勤车上下车情形，就要使用某种后进先出的数据结构，而具有这种属性的数据结构就是栈。

事实上，所有只有一个开口的容器都可以看做是一个后进先出或先进后出的机构，如井、陷阱、坑、瓶子等。

问题2：考虑一下表达式的求值过程

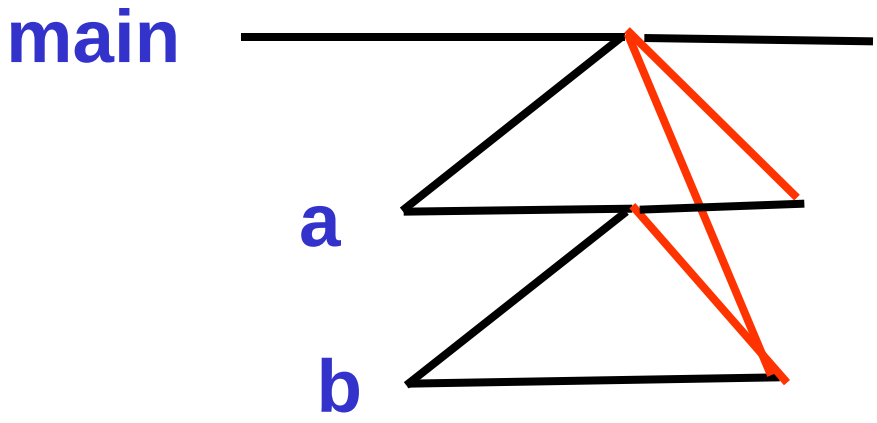
1+2*3 → 123*+ → 7

1*2+3 → 12*3+ → 5

实际上，程序设计里面也有多处可见先进先出或后进先出的现象。例如：函数调用

```
void a();
void b();
 main()
{   ….
    a( );
    cout<< “return from a”;
 }
void a()
 {
    …
    b( );
    cout<<“return from b”;
}
 void b( )
 {
   int num;   cin>>num;   cout<< num;
 }
```

后进先出即为栈



后调用，先返回

栈的定义

栈：限定仅在表尾进行插入和删除操作的线性表。

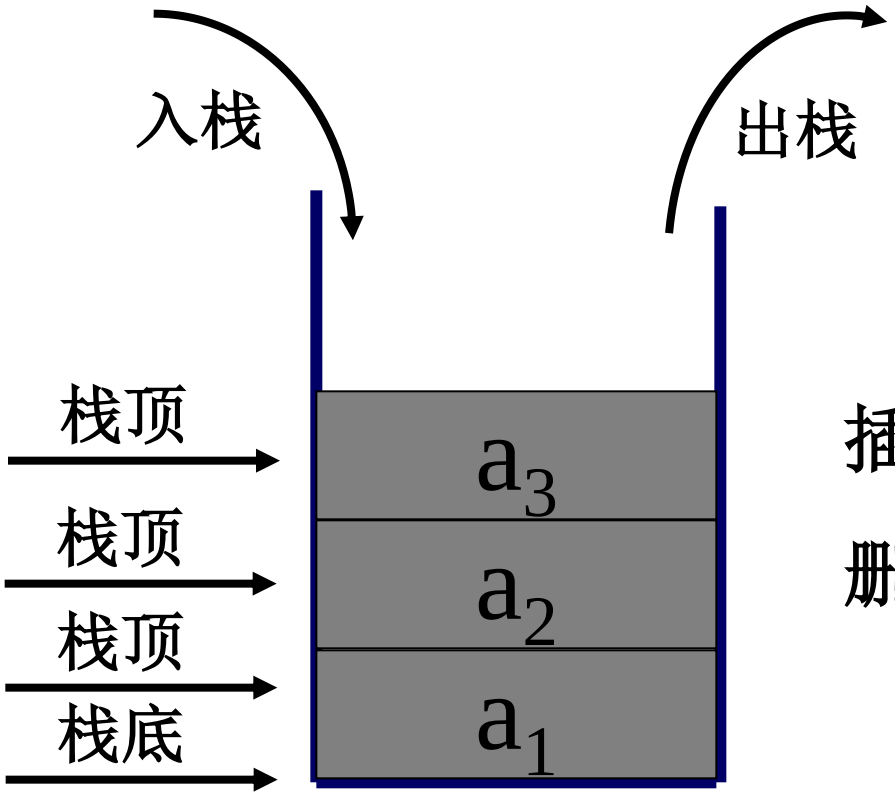
空栈：不含任何数据元素的栈。

允许插入和删除的一端称为栈顶，另一端称为栈底。

$$(a_1, a_2, \cdots\cdots, a_n)$$

栈底　　栈顶

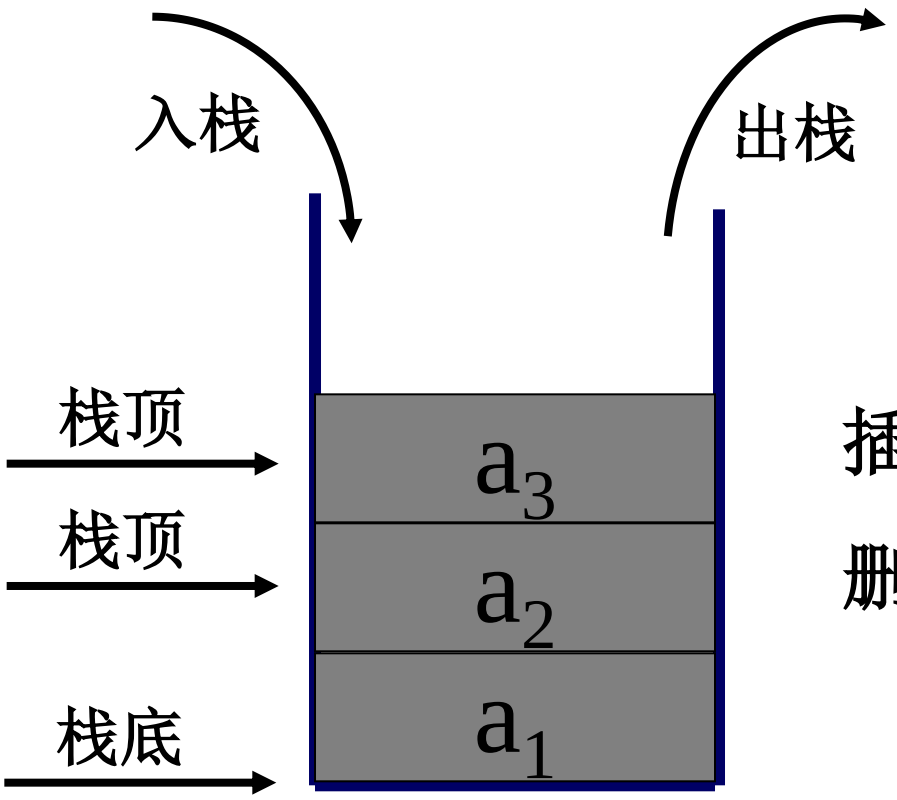
栈的示意图



插入：入栈、进栈、压栈

删除：出栈、弹栈

栈的示意图



插入：入栈、进栈、压栈

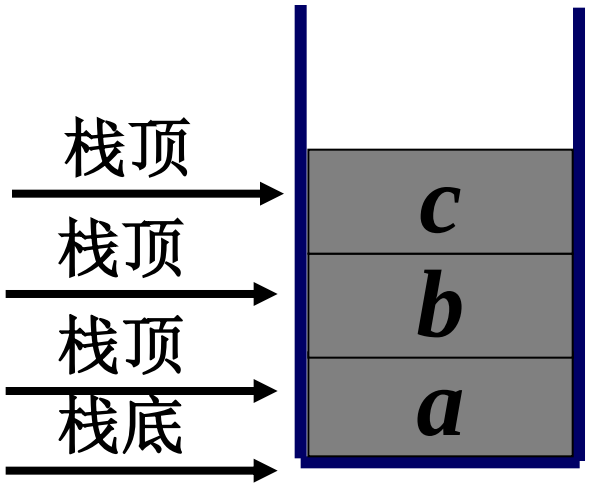
删除：出栈、弹栈

栈的操作特性：后进先出(LIFO)

栈的逻辑结构

例：有三个元素按*a*、*b*、*c*的次序依次进栈，且每个元素只允许进一次栈，则可能的出栈序列有多少种？

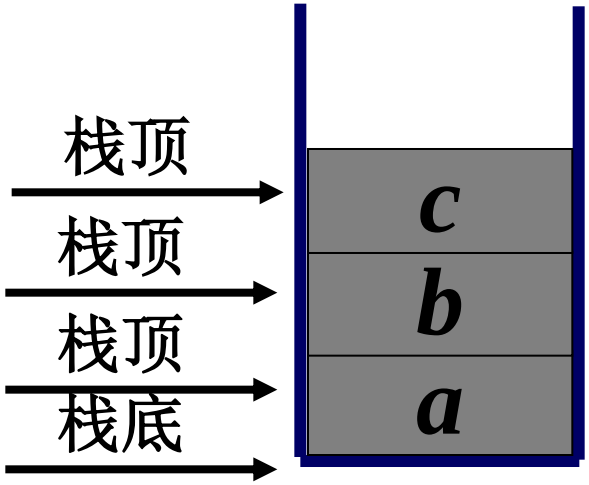
- 情况1：



栈的逻辑结构

例：有三个元素按*a*、*b*、*c*的次序依次进栈，且每个元素只允许进一次栈，则可能的出栈序列有多少种？

- 情况1：



出栈序列：*c*

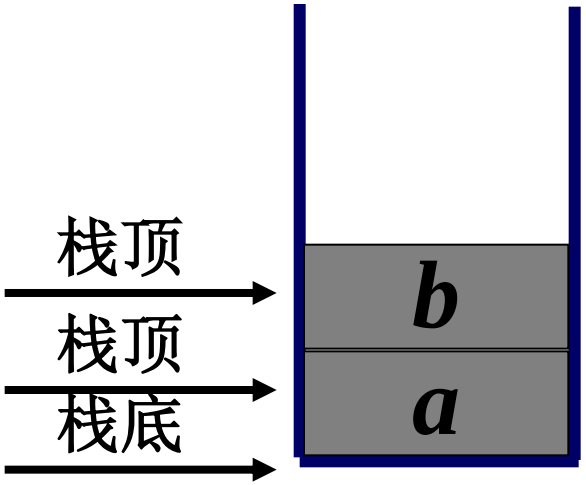
出栈序列：*c*、*b*

出栈序列：*c*、*b*、*a*

栈的逻辑结构

例：有三个元素按*a*、*b*、*c*的次序依次进栈，且每个元素只允许进一次栈，则可能的出栈序列有多少种？

- 情况2：

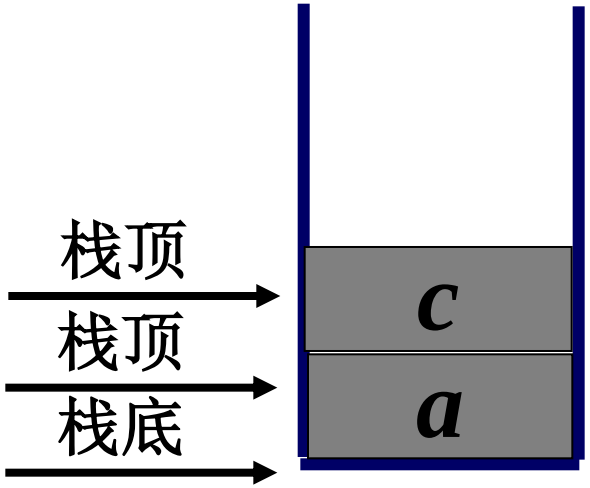


出栈序列：*b*

栈的逻辑结构

例：有三个元素按*a*、*b*、*c*的次序依次进栈，且每个元素只允许进一次栈，则可能的出栈序列有多少种？

- 情况2：



出栈序列：*b*

出栈序列：*b*、*c*

出栈序列：*b*、*c*、*a*

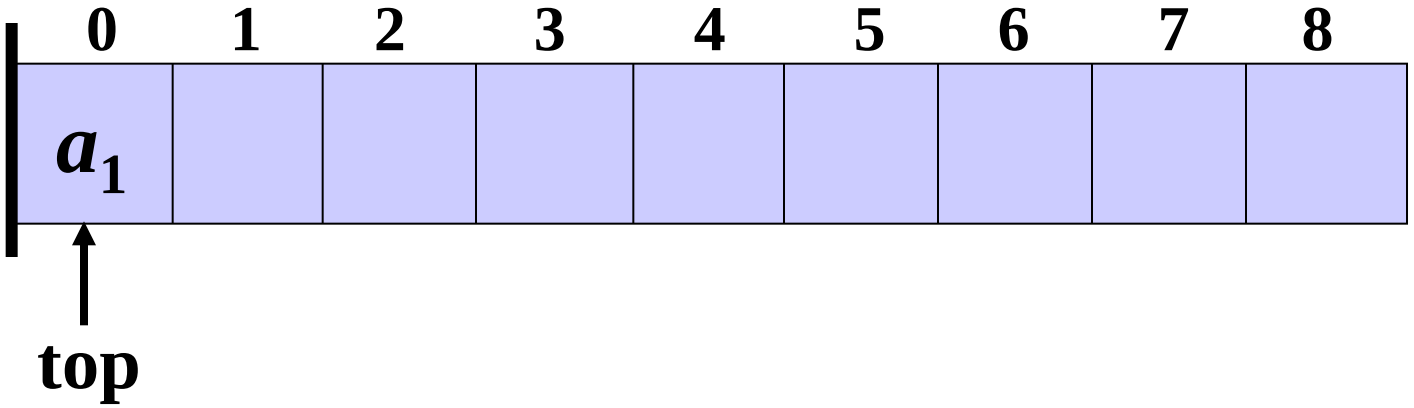
注意：栈只是对表插入和删除操作的位置进行了限制，并没有限定插入和删除操作进行的时间。

可能是出栈次序有abc、acb、bac、bca、cba五种。

栈的顺序存储结构及实现

顺序栈——栈的顺序存储结构

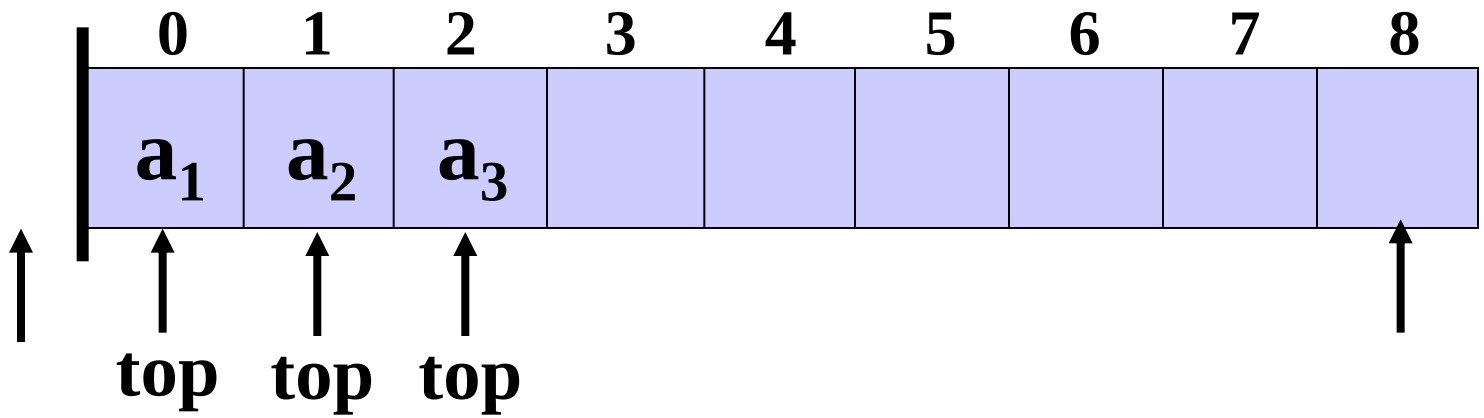
如何改造数组实现栈的顺序存储？



确定用数组的哪一端表示栈底。

附设指针top指示栈顶元素在数组中的位置。

栈的顺序存储结构及实现



进栈：top加1

出栈：top减1

栈空：top= −1

栈满：top= MAX_SIZE

顺序栈类的声明

```
const int maxSize=100;    //表的最大尺寸
typedef int dataType;     // 表元素为整型
class  seqStack{
    public:
        seqStack ( ) ;
        ~seqStack ( );
        void  Push (dataType  x );
        dataType   Pop ( );
        dataType   GetTop ( );
        bool  Empty ( );
    private:
        dataType  data[maxSize];
        int  top;
}
```

顺序栈的实现——入栈

操作接口： **void Push(dataType x);**

```
void seqStack::Push (dataType x){
    if (top==maxSize−1) throw “溢出”;
    top++;
    data[top]=x;
}
```

时间复杂度？

顺序栈的实现——出栈

操作接口：　dataType Pop( );

```
dataType  seqStack:: Pop ( ){
    if (top==−1)  throw “溢出”;
    return data[top−−];
}
```

时间复杂度？

思考题：两栈共享空间问题

在一个程序中需要同时使用具有相同数据类型的两个栈，如何顺序存储这两个栈？

解决方案1：

直接解决：为每个栈开辟一个数组空间。

会出现什么问题？如何解决？

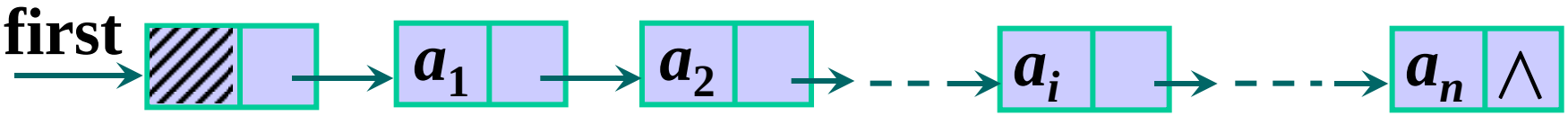
解决方案2：

顺序栈单向延伸——使用一个数组来存储两个栈

栈的链接存储结构及实现

链栈：栈的链接存储结构

如何改造链表实现栈的链接存储？

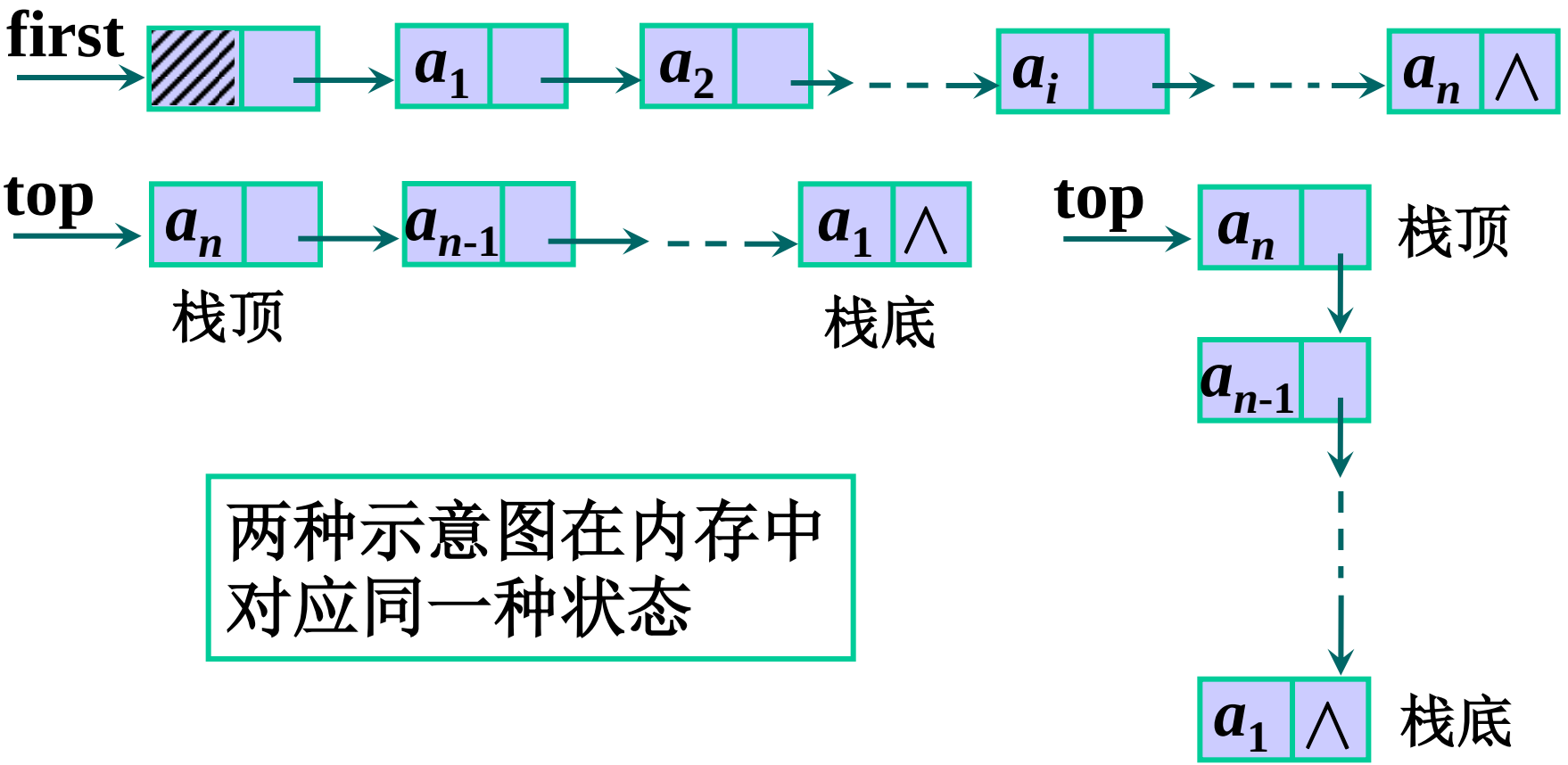


将哪一端作为栈顶？ 将链头作为栈顶，方便操作。

链栈需要加头结点吗？ 链栈不需要附设头结点。

栈的链接存储结构及实现

链栈： 栈的链接存储结构



两种示意图在内存中
对应同一种状态

链栈的类声明

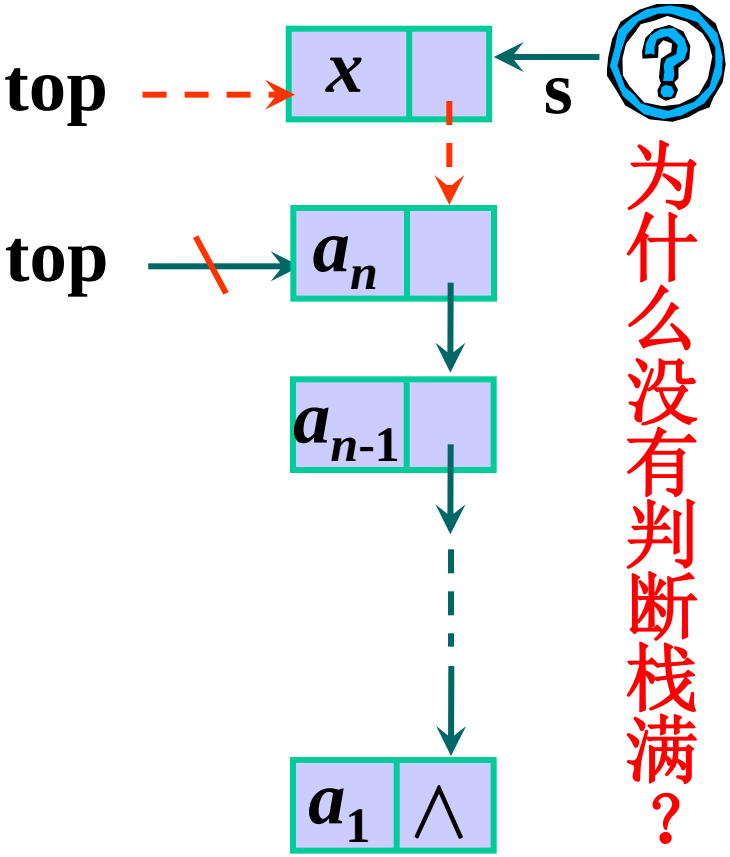
```
typedef int dataType
class LinkStack{
  public:
      LinkStack( );
      ~LinkStack( );
      void Push(dataType x);
      dataType Pop( );
      dataType GetTop( );
      bool Empty( );
  private:
      node *top;
}
```

结点的定义如下：

```
typedef int dataType;
struct node
{
    dataType data;
    node *next;
};
```

链栈的实现——插入

操作接口： void Push(dataType x);



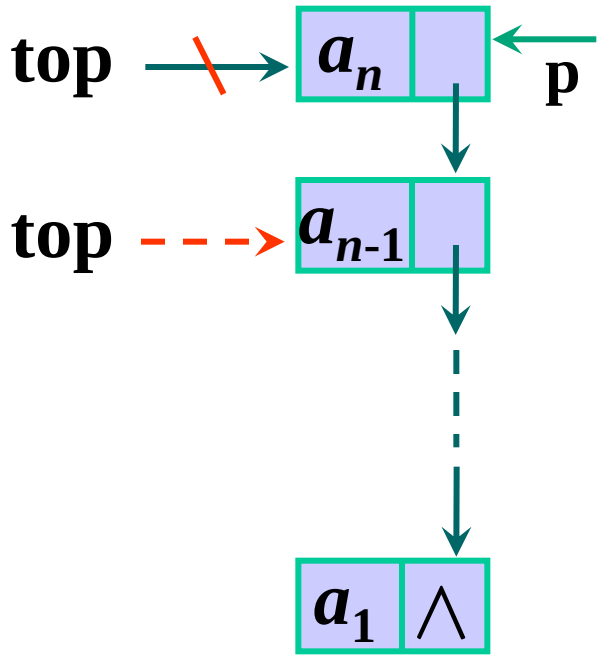
为什么没有判断栈满？

算法描述：

```
voidLinkStack::Push(dataType x)
{
    node *s=new node;
    s->data=x;
    s->next=top;
   top=s;
}
```

链栈的实现——删除

操作接口： dataType Pop( );



top++可以吗？

算法描述：

```
dataType LinkStack::Pop( ){
    if（top==NULL）
        throw "下溢";
    int x=top->data;
    node *p=top;
    top=top->next;
    delete p;
    return x;
}
```

顺序栈和链栈的比较

时间性能：相同，都是常数时间$O(1)$。

空间性能：

- 顺序栈：有元素个数的限制和空间浪费的问题。
- 链栈：没有栈满的问题，只有当内存没有可用空间时才会出现栈满，但是每个元素都需要一个指针域，从而产生了结构性开销。

总之，当栈的使用过程中元素个数变化较大时，用链栈是适宜的，反之，应该采用顺序栈。

3.2 栈的应用

应用1：乘坐校园通勤车

本应用模拟问题1：先上车的人最后下来。每个乘客用一个编号来表示，用户输入的乘客编号顺序就是乘客的等车顺序，程序就是倒着输出这一串编号，即乘客的下车顺序。

下面是乘客上/下车的主程序:

```
int main(){
//输入：用户提供数值n,代表n个乘客和n个乘客的编号
//输出：将乘客的编号倒着输出出来
    int n,item;
    seqStack passengers;
    cout<<“输入乘客人数n”<<endl;
    cin>>n;
    cout<<“按上车顺序输入乘客编号”;
    cout<<endl;
    for (int i=0;i<n;i++){
        cin>>item;
        passengers.push(item);
    }
    cout<<endl;
    cout<<“乘客的下车顺序是：”;
    while(!passengers.empty())
        cout<<passengers.pop()<<“ ”;
    cout<<endl;
}
```

运行结果：

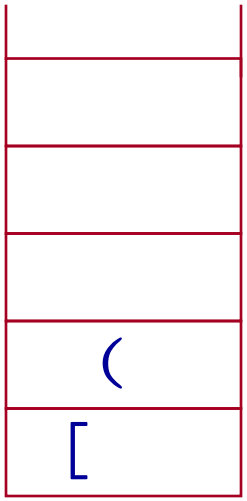
```
输入乘客人数n
10
按上车顺序输入乘客编号
11
24
35
46
57
89
3
5
8
9

乘客的下车顺序是：9 8 5 3 89 57 46 35 24 11
Press any key to continue
```

应用2：括号匹配的检验

[（ [ ] [ ] ）]

[（ （[ ] [ ] ）]



左括号：进栈

右括号：与栈顶括号匹配，
栈顶括号出栈；
否则，与栈顶括
号不匹配 ，就是错误

括号匹配算法的设计思想：

1）凡出现左括弧，则进栈；

2）凡出现右括弧，首先检查栈是否空

若栈空，则表明该“右括弧”多余，

否则和栈顶元素比较，

若相匹配，则“左括弧出栈”，

否则表明不匹配。

3）表达式检验结束时，

若栈空，则表明表达式中匹配正确，

否则表明“左括弧”有余。

算法:

```
bool matching(char exp[ ])
{  int state=1;
   ch=*exp++;
   while(ch!='#'&&state) {
      switch of ch {
         case 左括号:{push(s,ch);break;}
         case ')':{if(!Empty(s)&&GetTop(s)='(' )
                        pop(S,e);
                  else state=0;break; }
```

算法（续）：

```
        case ‘]’:{if(! Empty(s)&&GetTop(s)=‘[‘ )
                        pop(s,e);
                    else state=0;break; }
     }//switch
    ch=*exp++;
  }//while
  if(state &&Empty(s)) return TRUE;
   else  return FALSE;
}//matching
```

```
int main()
{
   seqStack openings;                    //存放尚未匹配的左括号
   char symbol;          //用来记录当前符号的变量
   bool match=true;
   cout<<“请输入一个带括号的字符串"<<endl;
   while(match && (symbol = cin.get())!='\n'){
      if (symbol =='{'|| symbol =='(' || symbol =='[')
            openings.Push(symbol);
      if (symbol =='}'|| symbol ==')' || symbol ==']'){
            if (openings.Empty()){
                  cout<<"检测到不匹配的括号"<<symbol<<endl;
                  match=false;
                  return 0;
            }
            else{
                  char topSymbol;
                  topSymbol=openings.Pop();
                  match=(symbol=='}'&&topSymbol=='{') ||  //判断是否匹配
                        (symbol==')'&&topSymbol=='(') || (symbol==']'&&topSymbol=='[');
                  if (!match) {
                        cout<<"括号类型不匹配"<<match<<symbol<<endl;
                        return 0;
                  }
            }
      }
   }
   if (!openings.Empty())  //若字符串处理完，还剩左括号，报告错误
        cout<<"检测到多余的左括号。"<<endl;
   else
        cout<<"所有括号完美匹配，谢谢！ "<<endl;
   return 0;
}
```

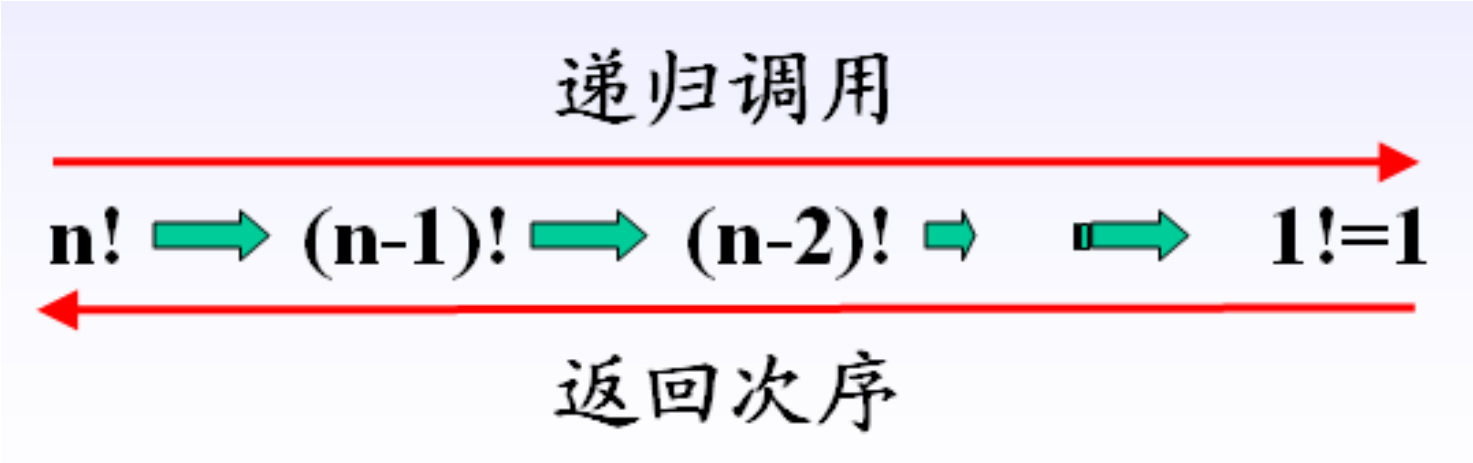
运行结果：

```
请输入一个带括号的字符串
abc(are(hjk)jkl]
括号类型不匹配0]
Press any key to continue
```

```
请输入一个带括号的字符串
(([we] are {nothing}))
所有括号完美匹配，谢谢！
Press any key to continue
```

应用3：阶乘计算

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



递归的基本思想：

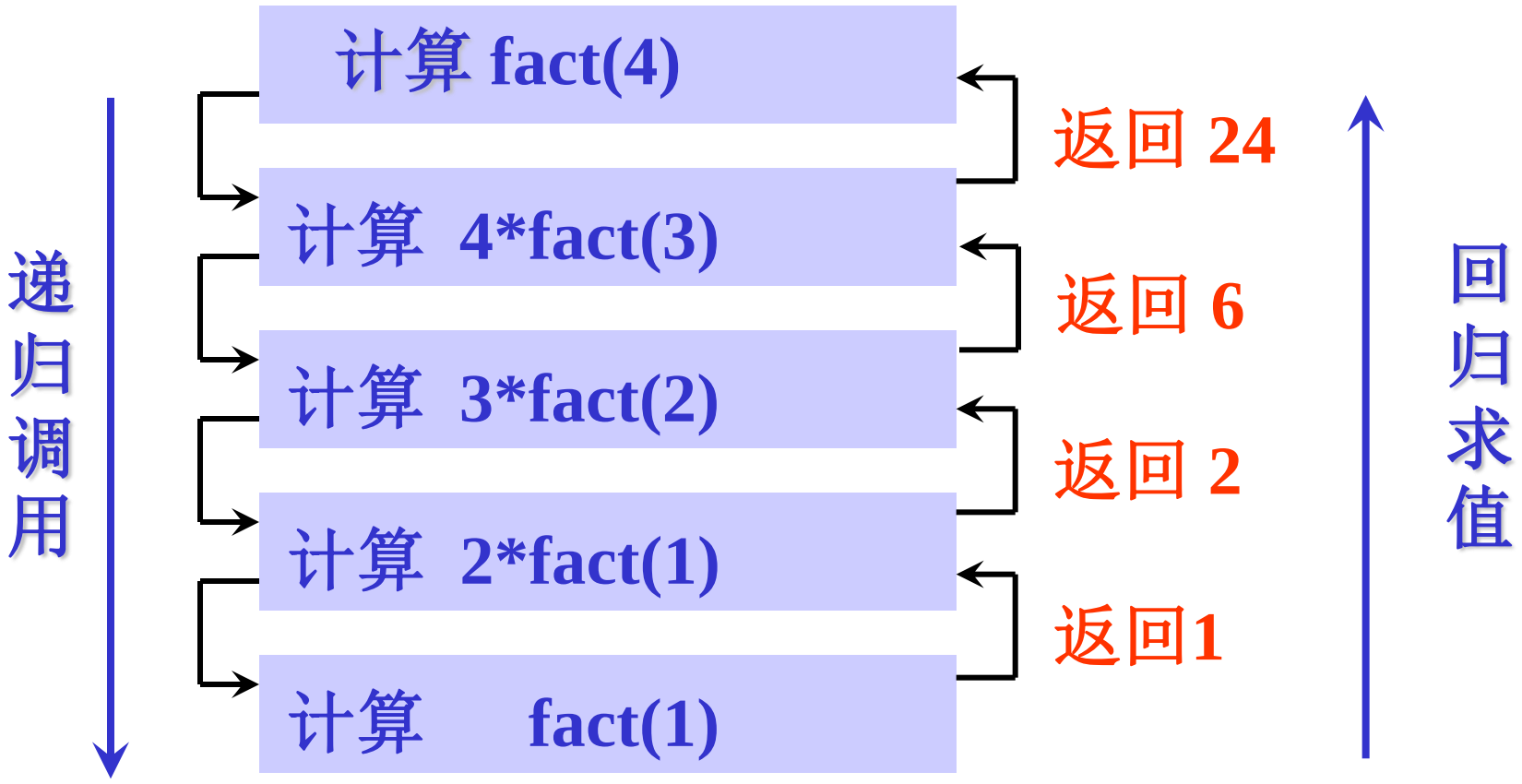
把一个不能或不好解决的大问题转化为一个或几个小问题，再把这些小问题进一步分解成更小的小问题，直至每个小问题都可以直接解决。

递归的要素：

(1) 递归边界条件：确定递归到何时终止，也称为递归出口；

(2) 递归模式：大问题是如何分解为小问题的，也称为递归体。

求解阶乘 n! 的过程：


计算 fact(4)
计算 4*fact(3)
计算 3*fact(2)
计算 2*fact(1)
计算 fact(1)
返回 24
返回 6
返回 2
返回1
递归调用
回归求值

阶乘函数的递归算法：

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

3.3 队列

从排队问题说起…

1.银行排队问题

2.理发店排队问题

3.作业排队问题

那么，如何编程实现排队模拟呢？

队列的定义

队列：只允许在一端进行插入操作，而另一端进行删除操作的线性表。

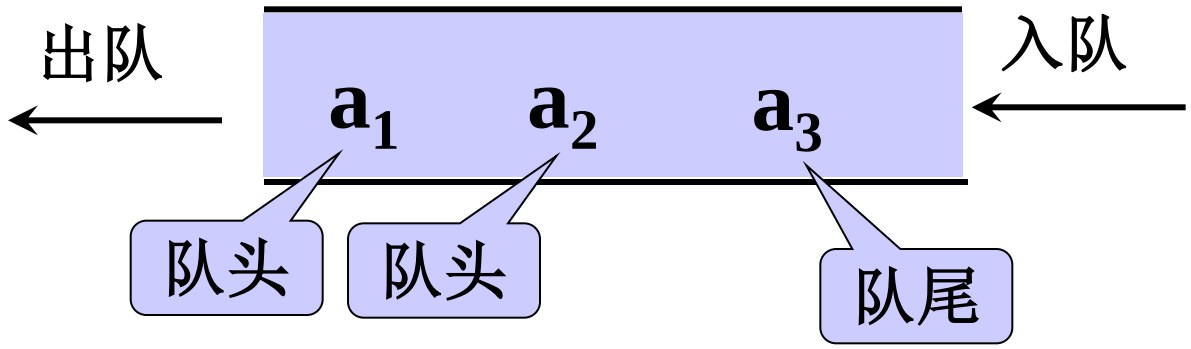
空队列：不含任何数据元素的队列。

允许插入（也称入队、进队）的一端称为队尾，允许删除（也称出队）的一端称为队头。

$$(a_1, a_2, \cdots\cdots, a_n)$$

队头　　队尾

队列的逻辑结构



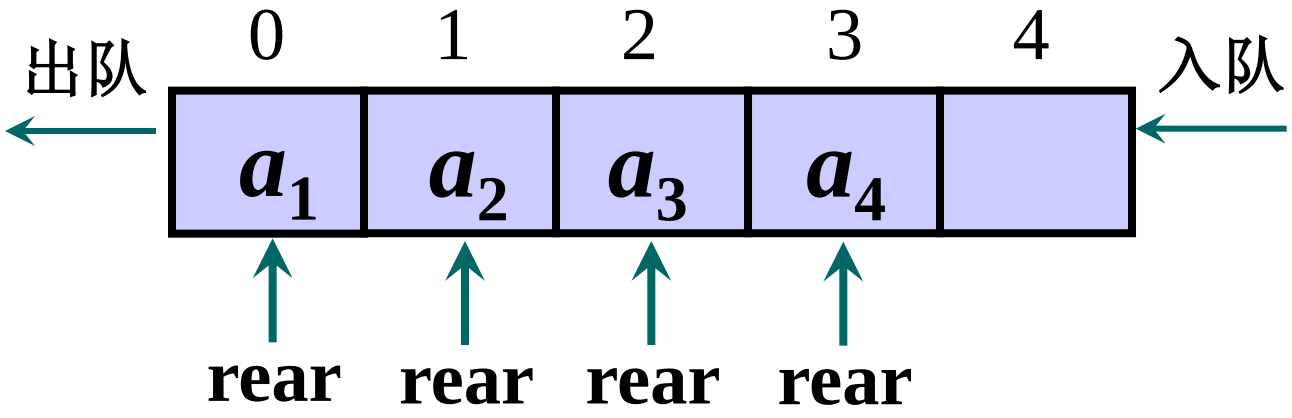
队列的操作特性：先进先出(FIFO)

队列的顺序存储结构及实现

顺序队列： 队列的顺序存储结构

如何改造数组实现队列的顺序存储？

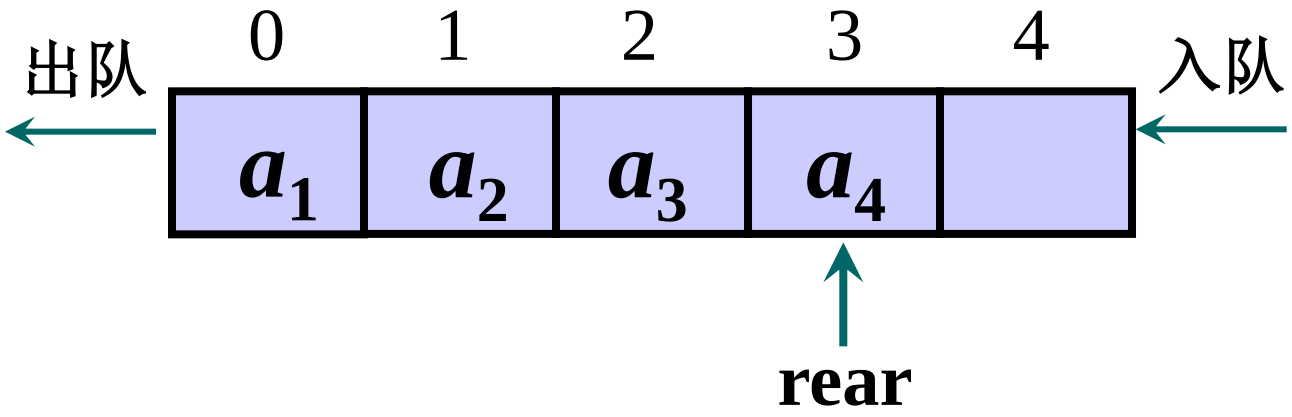
例：$a_1a_2a_3a_4$依次入队



入队操作时间性能为$O(1)$

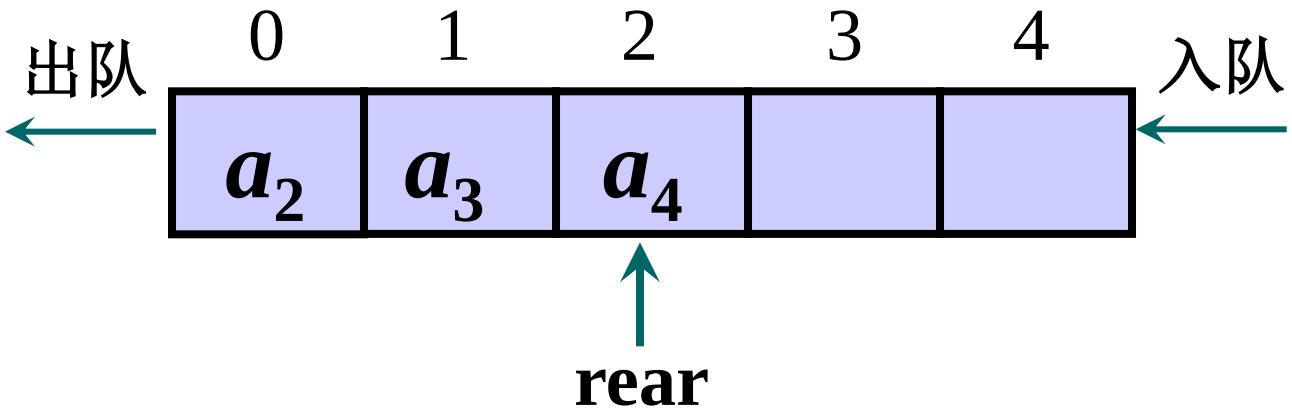
如何改造数组实现队列的顺序存储？

例：a_1a_2依次出队



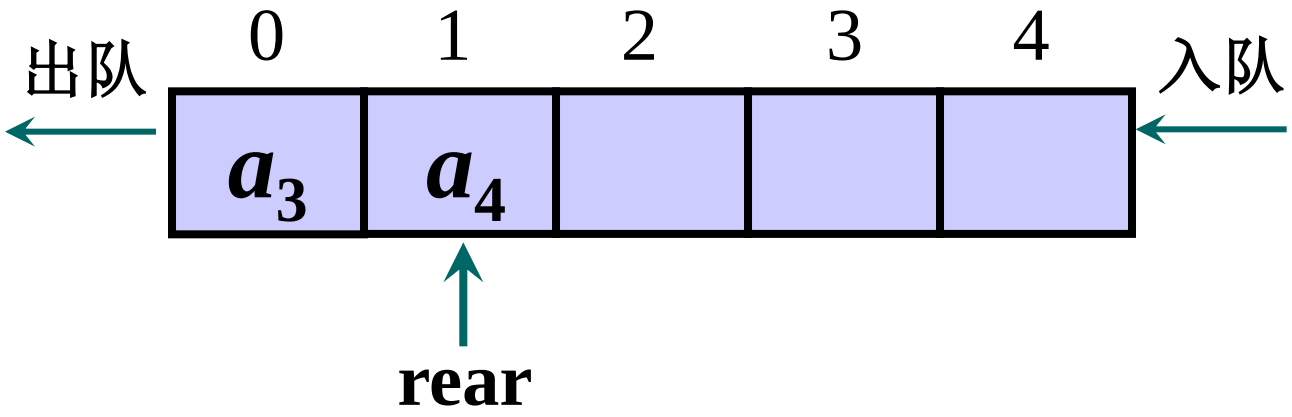
如何改造数组实现队列的顺序存储？

例：a_1a_2依次出队



如何改造数组实现队列的顺序存储？

例：a_1a_2依次出队



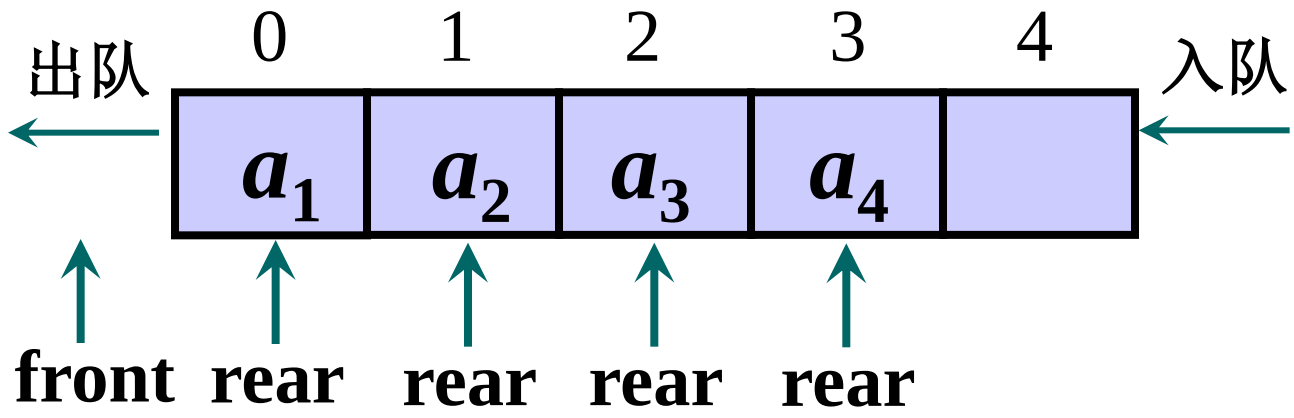
出队操作时间性能为O(*n*)

如何改进出队的时间性能？

放宽队列的所有元素必须存储在数组的前*n*个单元这一条件，只要求队列的元素存储在数组中连续的位置。

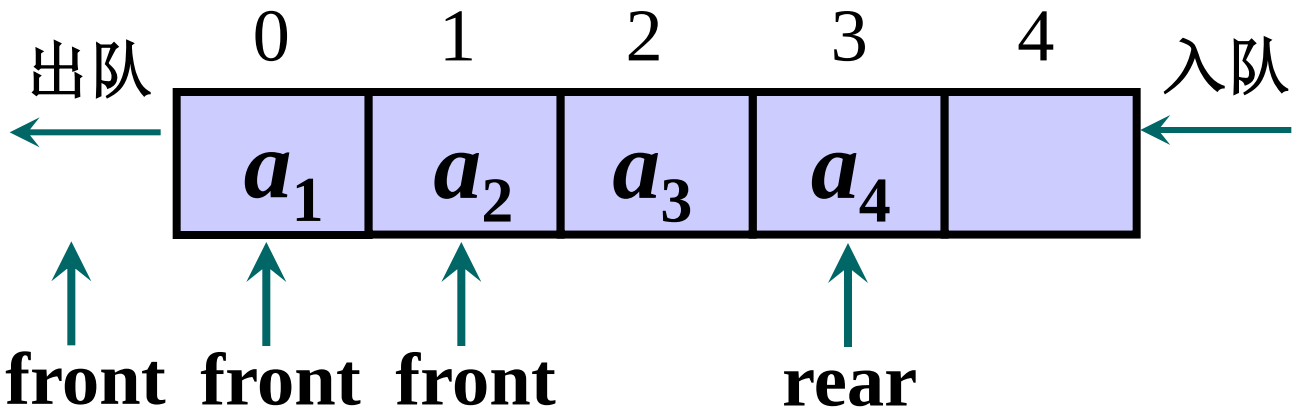
设置队头、队尾两个指针

例：$a_1a_2a_3a_4$依次入队



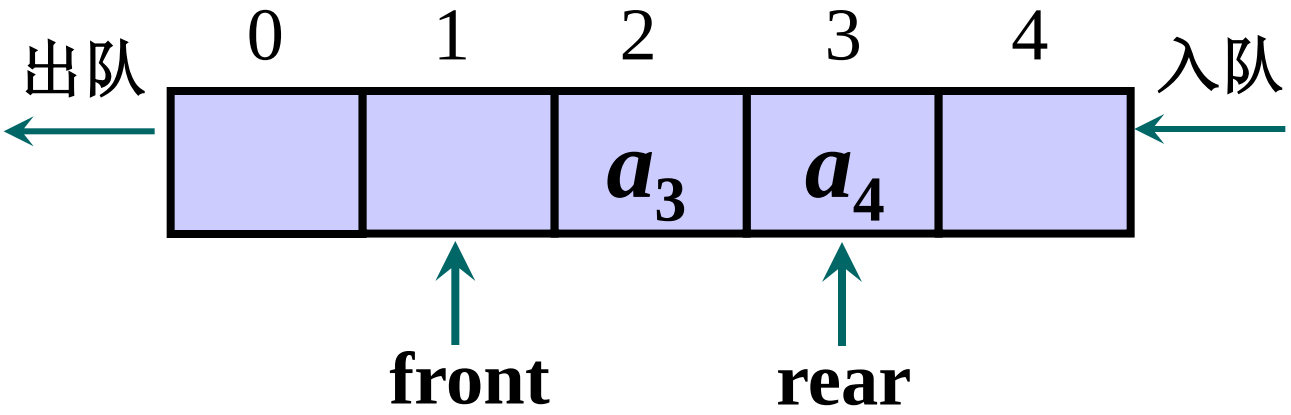
入队操作时间性能仍为$O(1)$

例：a_1a_2依次出队



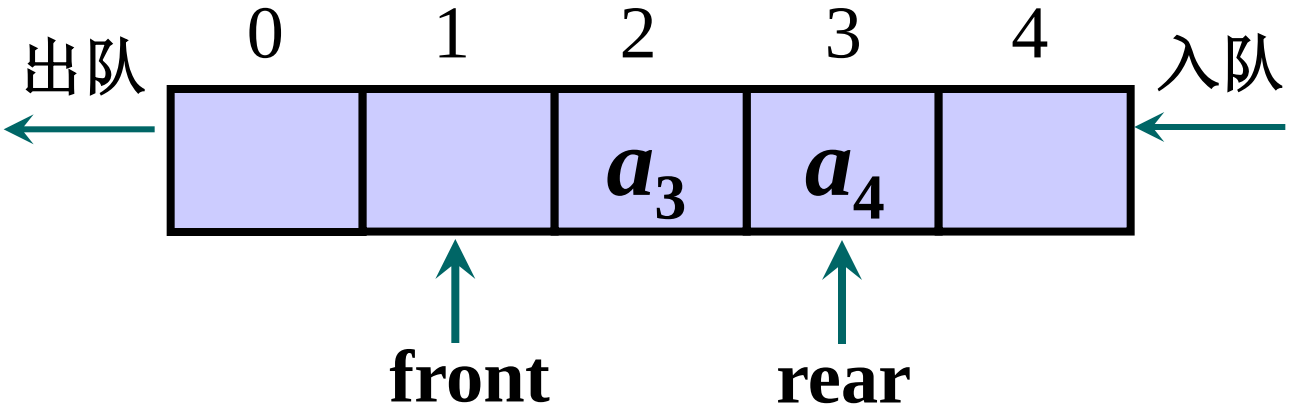
出队操作时间性能提高为$O(1)$

例：a_1a_2依次出队



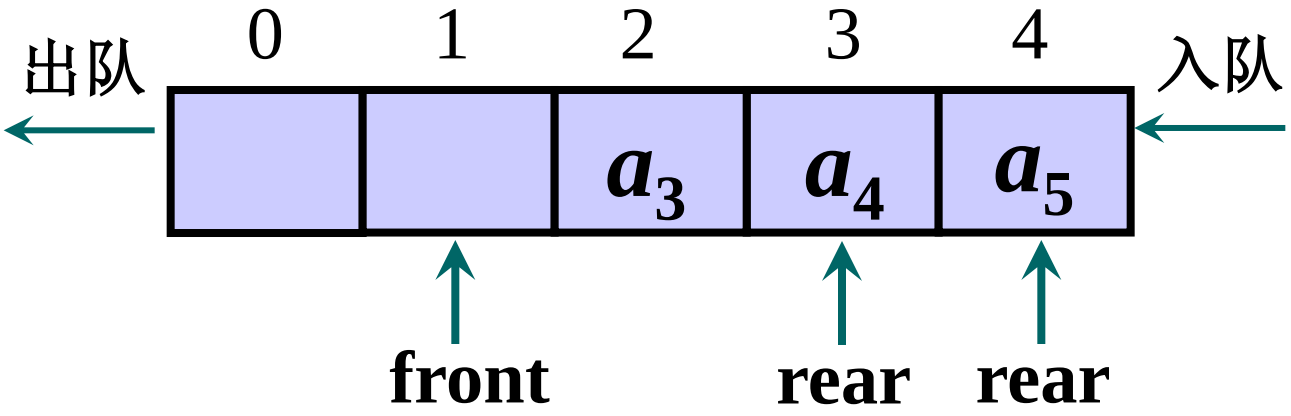
队列的移动有什么特点？

例：a_1a_2依次出队



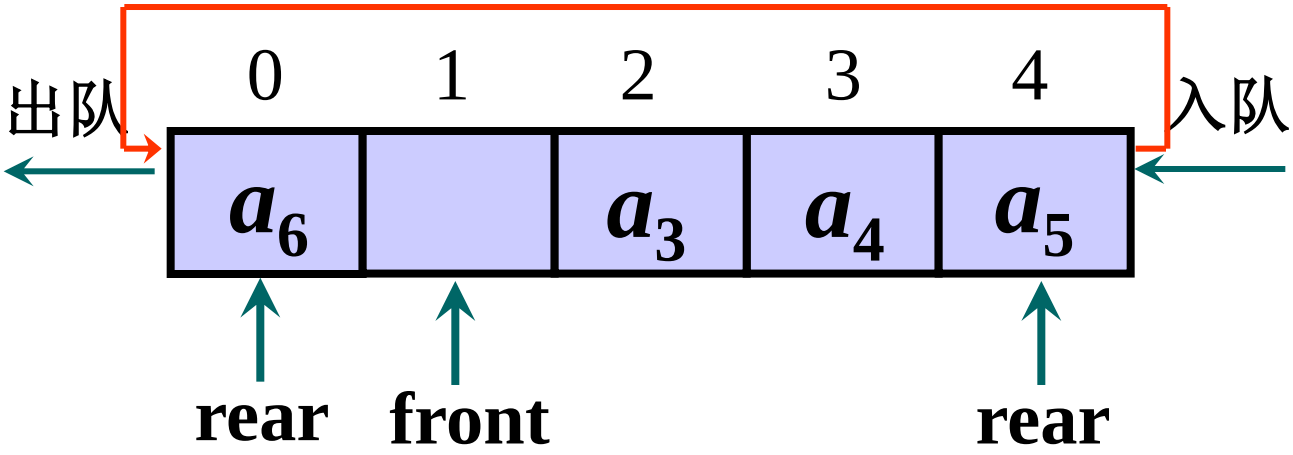
整个队列向数组下标较大方向移动 ➡ 单向移动性

继续入队会出现什么情况？



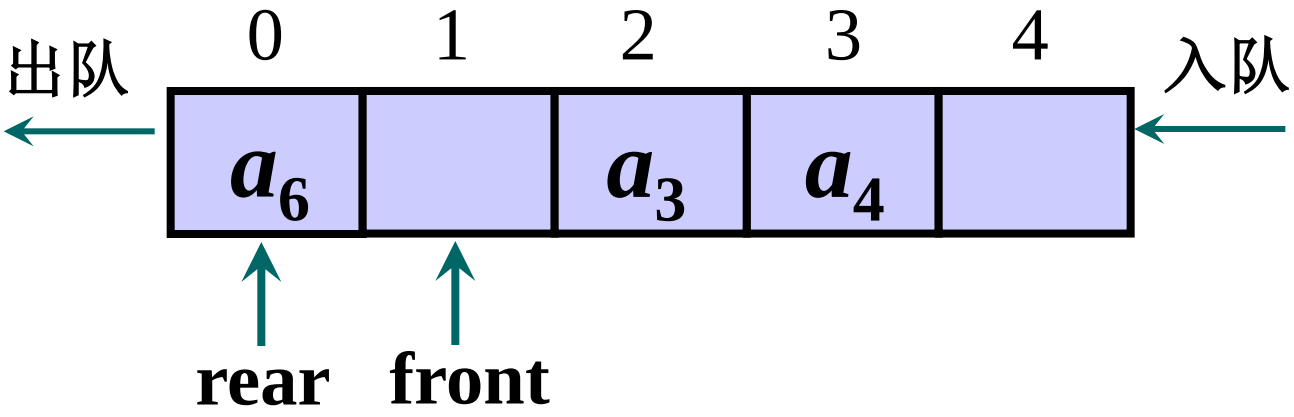
假溢出：当元素被插入到数组中下标最大的位置上之后，队列的空间就用尽了，尽管此时数组的低端还有空闲空间，这种现象叫做假溢出。

如何解决假溢出？



循环队列：将存储队列的数组头尾相接。

如何实现循环队列？

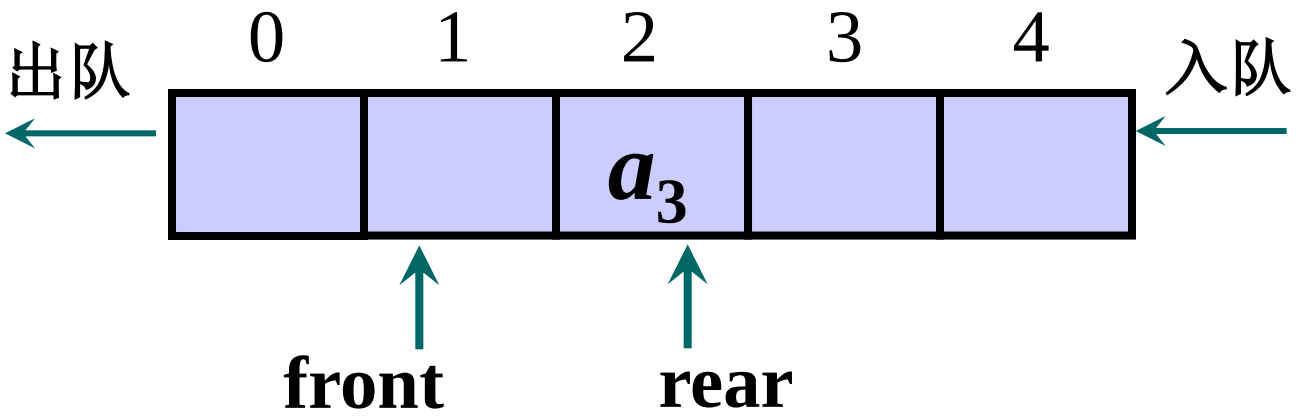


不存在物理的循环结构，用软件方法实现。

求模：（4＋1）mod 5＝0

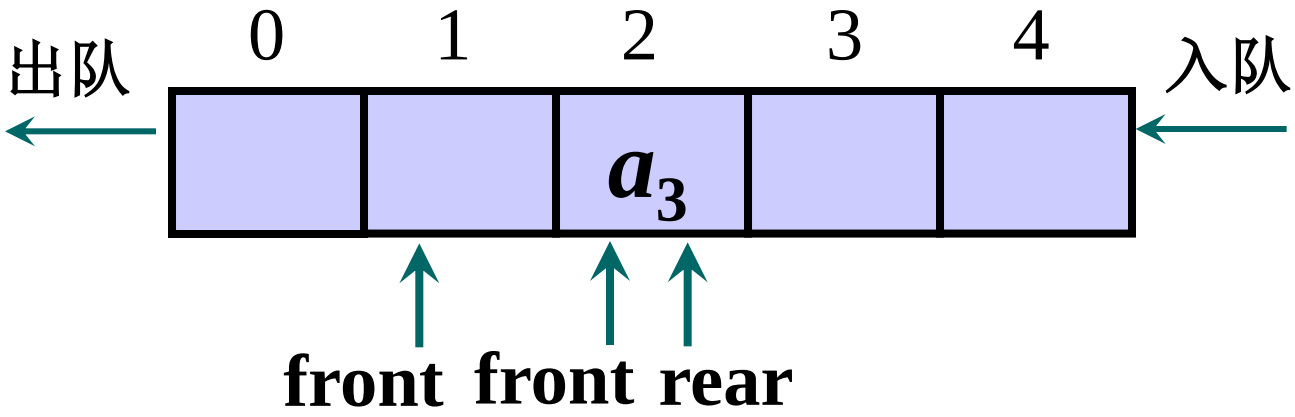
如何判断循环队列队空？

队空的临界状态



如何判断循环队列队空？

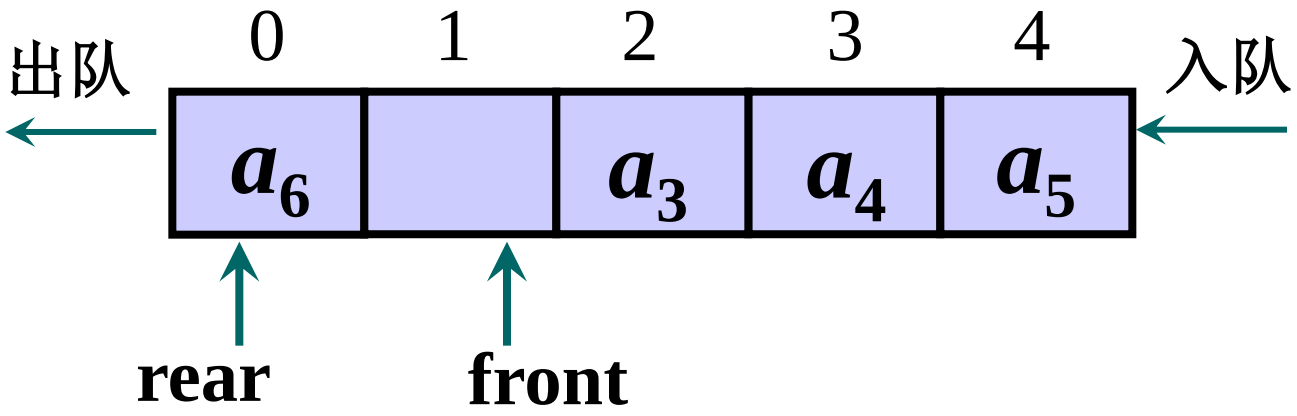
执行出队操作



队空：front=rear

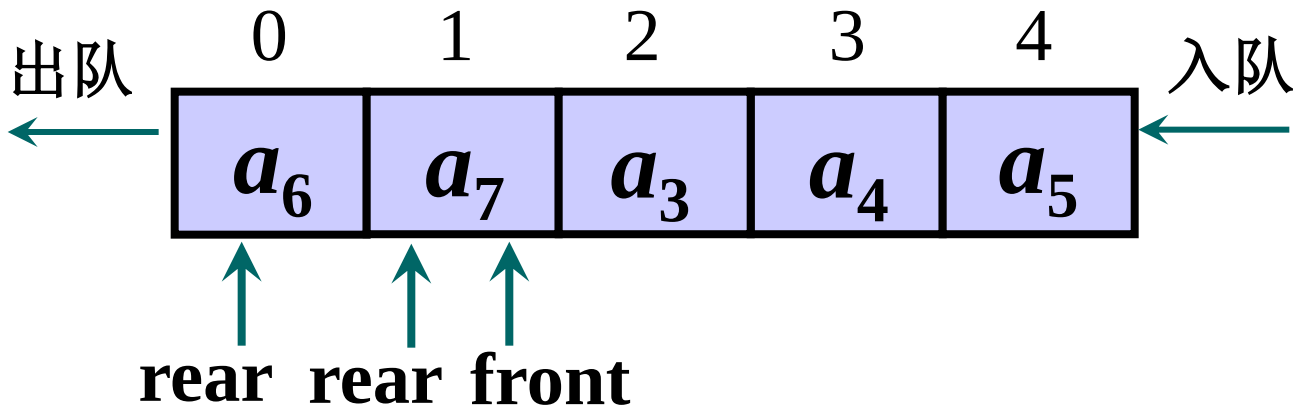
如何判断循环队列队满？

队满的临界状态



如何判断循环队列队满？

执行入队操作



队满：front=rear

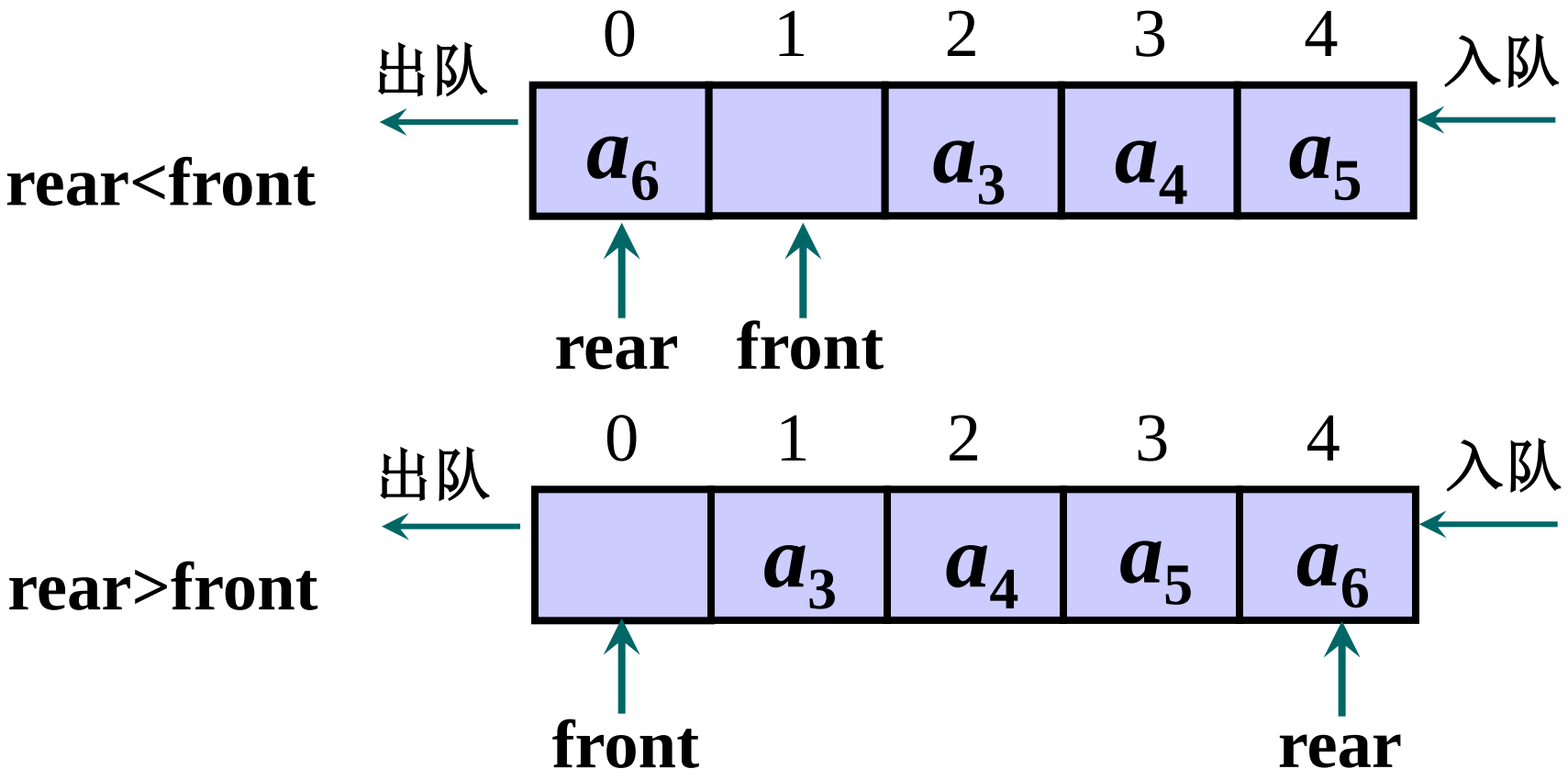
如何确定不同的队空、队满的判定条件？
为什么要将队空和队满的判定条件分开？

方法一：修改队满条件，浪费一个元素空间，队满时数组中只有一个空闲单元。

方法二：设置标志flag，当front=rear且flag=0时为队空，当front=rear且flag=1时为队满。

方法三：附设一个存储队列中元素个数的变量num，当num=0时队空，当num=QueueSize时为队满。

哪种方式最好呢？



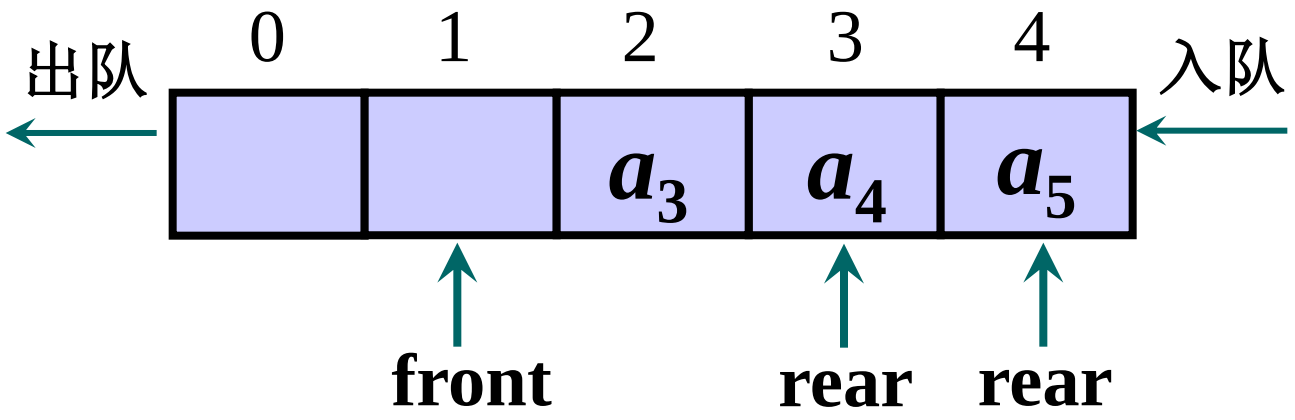
队满的条件：(rear+1) mod QueueSize=front

循环队列类的声明

```
const int QueueSize=100;
typedef int dataType;
class CirQueue
{
  public:
     CirQueue( );
     ~ CirQueue( );
     void EnQueue(dataType x);
    dataType DeQueue( );
    dataType GetQueue( );
    bool Empty( );
  private:
    dataType data[QueueSize];
    int front, rear;
};
```

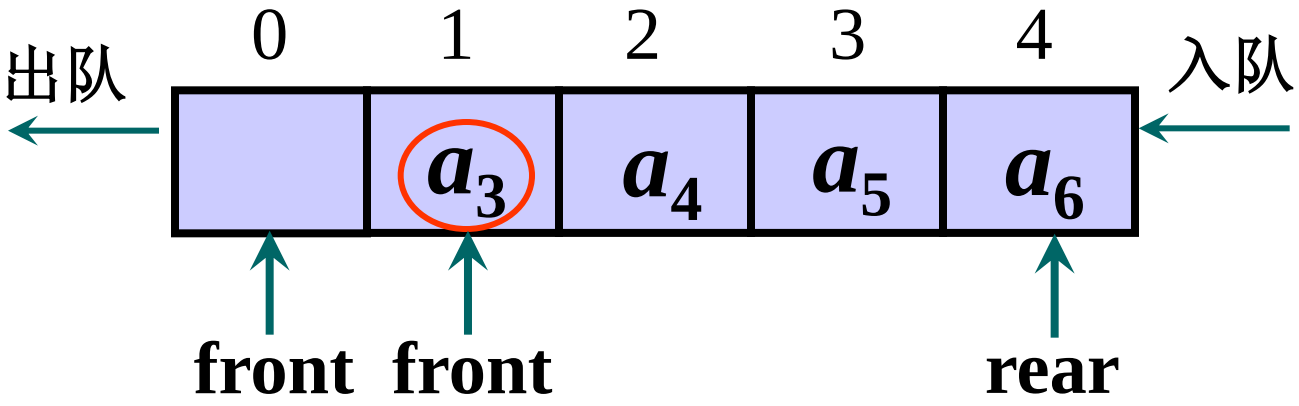
循环队列的实现——入队

```
void CirQueue::EnQueue（dataType x）
 {
   if（（rear+1）% QueueSize ==front）throw "上溢";
   rear=（rear+1）% QueueSize;
   data[rear]=x;
 }
```



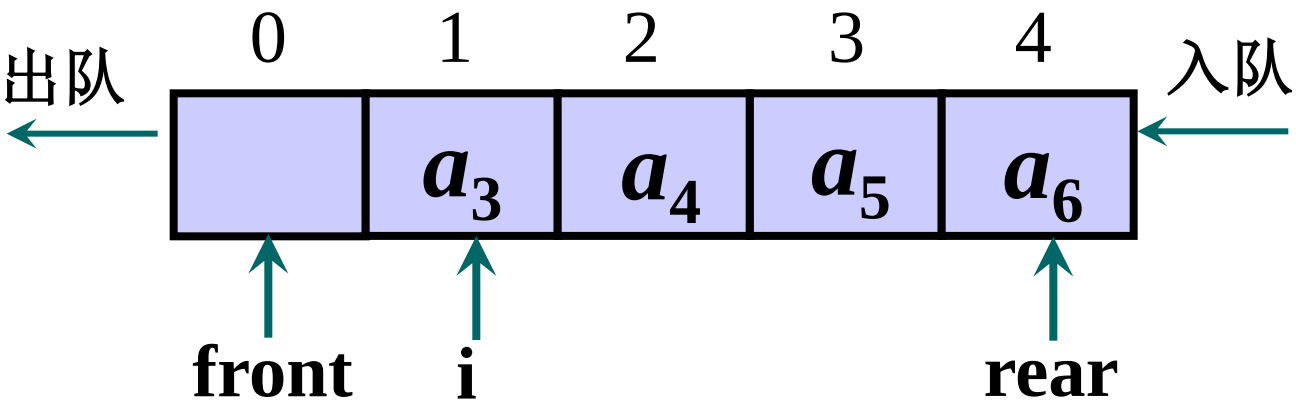
循环队列的实现——出队

```
dataType CirQueue::DeQueue( )
{
    if (rear==front) throw "下溢";
    front=(front+1) % QueueSize;
    return data[front];
}
```



循环队列的实现——读队头元素

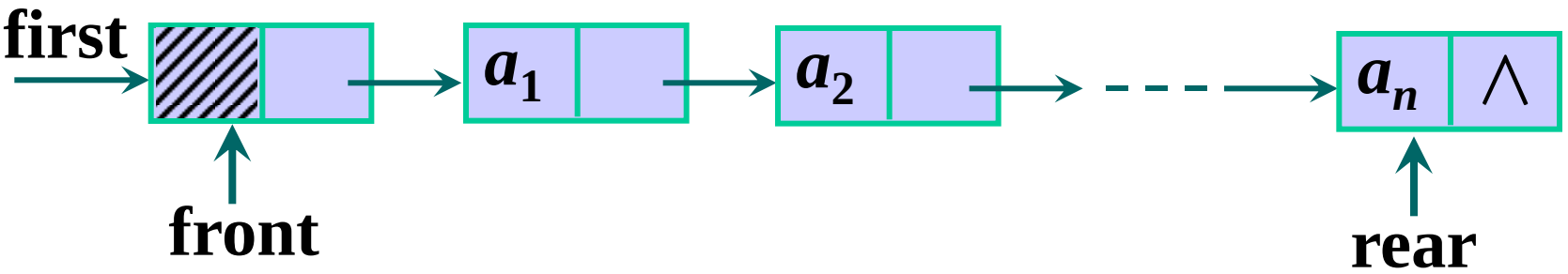
```
dataType CirQueue::GetQueue（ ）
{
   if（rear==front）throw "下溢";
   i=（front+1）% QueueSize;
   return data[i];
}
```



队列的链接存储结构及实现

链队列：队列的链接存储结构

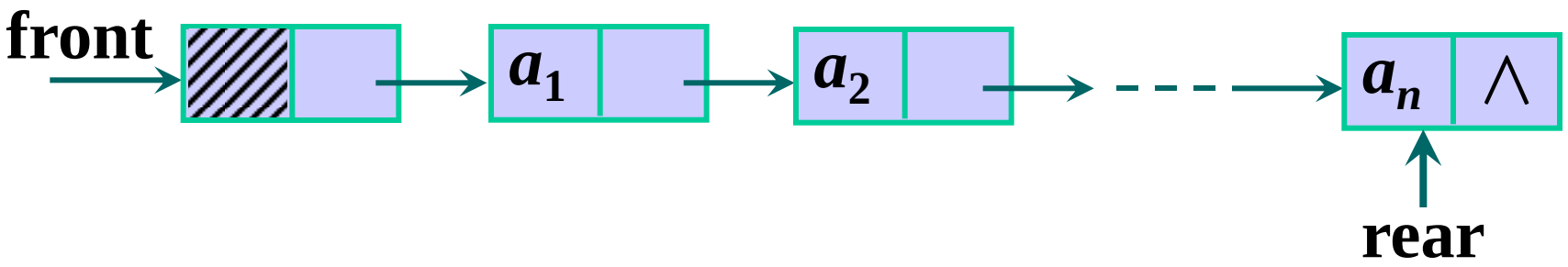
如何改造单链表实现队列的链接存储？



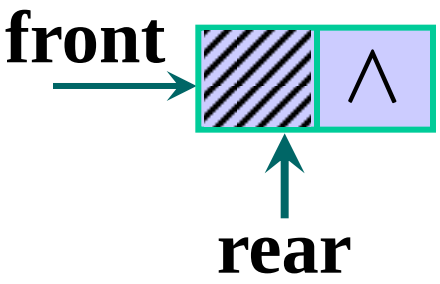
队头指针即为链表的头指针

队列的链接存储结构及实现

非空链队列



空链队列



链队列类的声明

```
typedef int dataType;
class LinkQueue
{
  public:
     LinkQueue( );
     ~LinkQueue( );
     void EnQueue(dataType x);
     dataType DeQueue( );
     dataType GetQueue( );
     bool Empty( );
  private:
     node *front, *rear;
};
```

结点的定义如下：

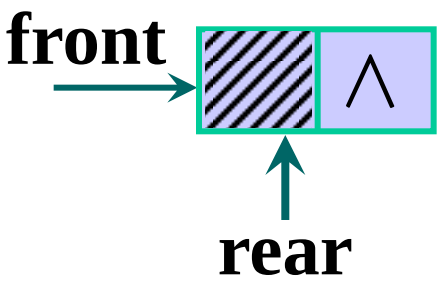
```
typedef int dataType;
struct node
{
     dataType data;
     node *next;
};
```

链队列的实现——构造函数

操作接口： LinkQueue( );

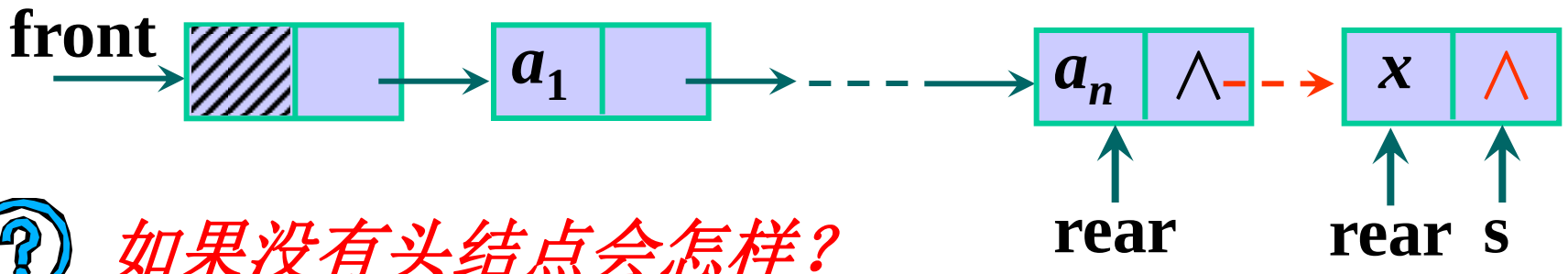
算法描述：

```
LinkQueue::LinkQueue（ ）
{
    front=new node;
    front->next=NULL;
    rear=front;
}
```

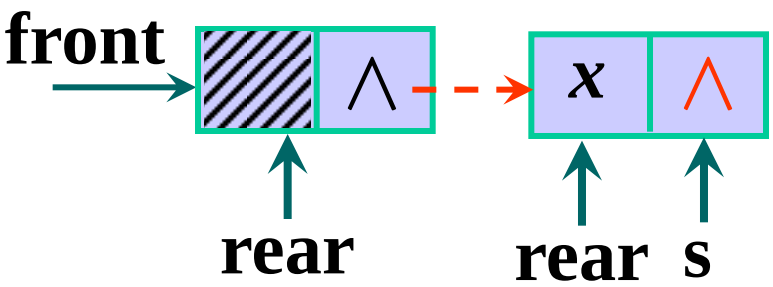


链队列的实现——入队

操作接口： void EnQueue(dataType x);



如果没有头结点会怎样？

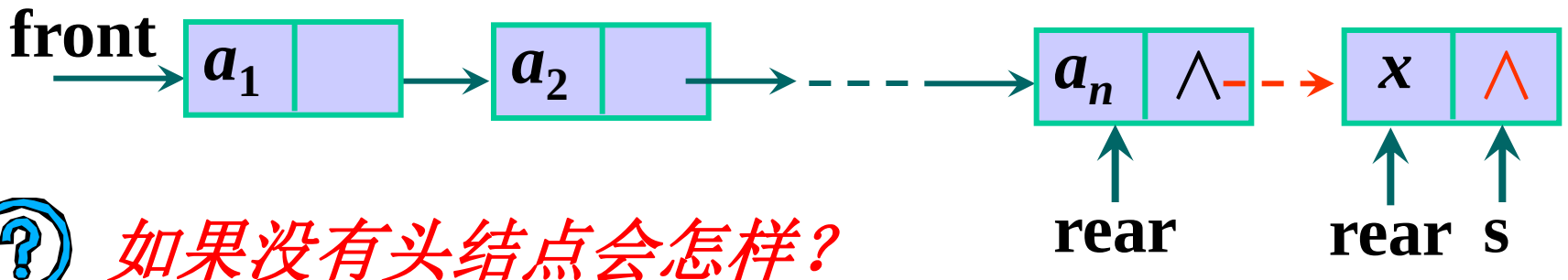


算法描述：

```
s->next=NULL;
rear->next=s;
rear=s;
```

链队列的实现——入队

操作接口： **void EnQueue(dataType x);**



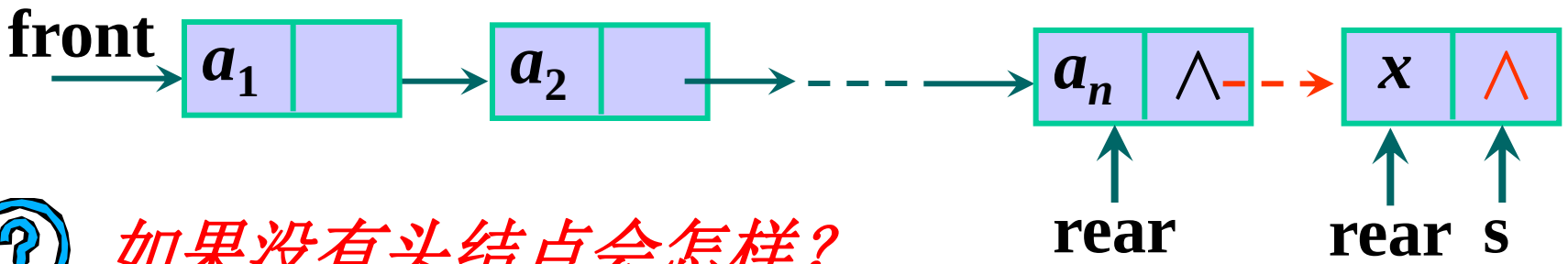
如果没有头结点会怎样？

算法描述：

```
s->next=NULL;
rear->next=s;
rear=s;
```

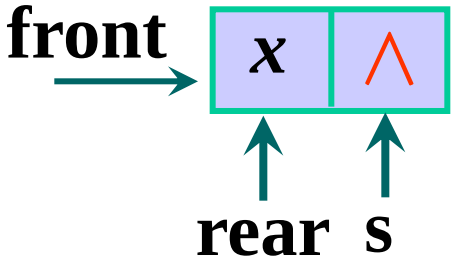
链队列的实现——入队

操作接口： void EnQueue(dataType x);



如果没有头结点会怎样？

front=rear=NULL



算法描述：

s->next=NULL;

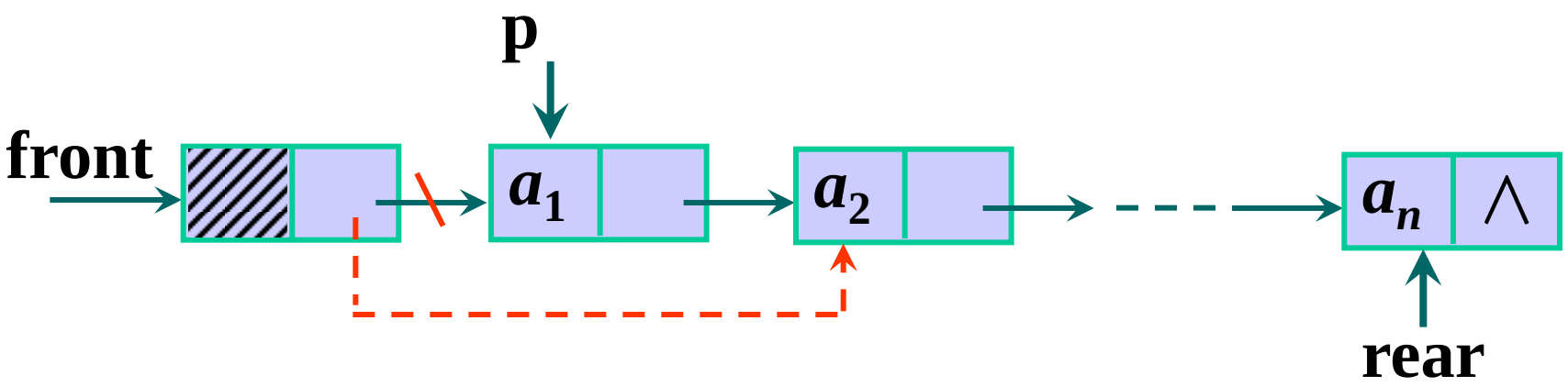
rear=s;

front=s;

链队列的实现——入队

```
void LinkQueue::EnQueue（dataType x）
{
    node *s=new node;
    s->data=x;
    s->next=NULL;
    rear->next=s;
    rear=s;
}
```

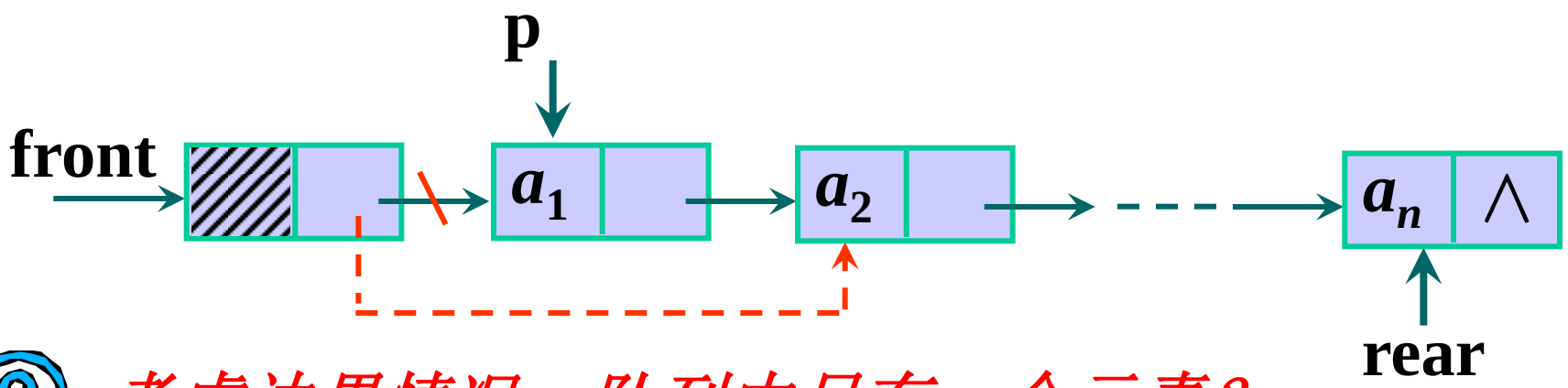
链队列的实现——出队



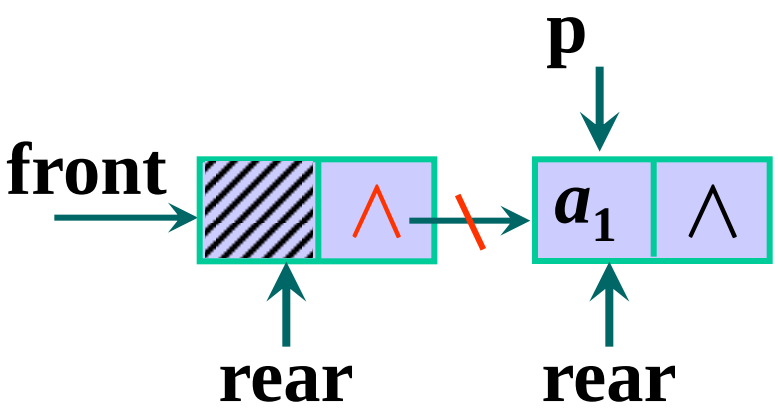
算法描述：

```
p=front->next;
front->next=p->next;
```

链队列的实现——出队



考虑边界情况：队列中只有一个元素？



如何判断边界情况？

算法描述：

```
if (p->next==NULL)
    rear=front;
```

链队列的实现——出队

```
dataType LinkQueue::DeQueue( )
{
    if (rear==front) throw "下溢";
    node *p=front->next;
    int x=p->data;
    front->next=p->next;
    if (p->next==NULL) rear=front;
    delete p;
    return x;
}
```

循环队列和链队列的比较

时间性能：

循环队列和链队列的基本操作都需要常数时间$O(1)$。

空间性能：

循环队列：必须预先确定一个固定的长度，所以有存储元素个数的限制和空间浪费的问题。

链队列：没有队列满的问题，只有当内存没有可用空间时才会出现队列满，但是每个元素都需要一个指针域，从而产生了结构性开销。

3.4 队列的应用

应用1：先来先得礼品赠送

很多购物网站或商场会搞些所谓的先来先得之类的促销。模拟这个先来先得场景：每个客户用一个编号来表示，用户给出的客户编号的顺序就是客户访问网站或商场的次序，设计的程序就是输出这一串编号，即获得礼品的客户顺序。

下面是先来先得礼品赠送主程序：

```
int main(){
//输入：用户提供数值n,代表n个客户和n个客户的编号
//输出：将客户的编号按照输入的顺序输出出来
    int n,item;
    CirQueue customers;
    cout<<“输入客户人数n”<<endl;
    cin>>n;
    cout<<“按访问网站或商场的顺序输入客户编号”<<endl;
    for (int i=0;i<n;i++){
        cin>>item;
        customers.EnQueue(item);
    }
    cout<<endl<<endl;
    cout<<“领取礼品的客户顺序是：”;
    while(!customers.empty())
        cout<<customers.DeQueue()<<“ ”;
    cout<<endl;
}
```

运行结果：

```
输入客户人数n
10
按访问网站或商场的顺序输入客户编号
11
24
35
46
57
89
3
4
8
9


领取礼品的客户顺序是：11 24 35 46 57 89 3 4 8 9
Press any key to continue
```

应用2： 解素数环问题

```
typedef int T;                          //抽象数据类型T定义为int
#include "CirQueue.h"                   //顺序循环队列类
bool isPrime(int k)                     //判断k是否为素数
{
    int j=2;
    if(k==2)
        return true;
    if(k<2 || k>2 && k%2==0)
        return false;
    else
    {
        j=3;
        while(j<k && k%j!=0)
            j=j+2;
        if(j>=k)
            return true;
        else
```

3.4 队列的应用

应用3：处理等待问题时系统设立队列。

队列具有“先进先出”的特性，当需要按一定次序等待时，系统需设立一个队列。如：银行模拟问题、理发馆排队问题等。

应用4：实现广度遍历算法时使用队列。

实现广度遍历算法，如按层次遍历二叉树、以广度优先算法遍历图，都需要使用队列（详细算法将在以后的章节中介绍）。

本章的基本内容是：

栈和队列

- 从数据结构角度看，栈和队列是操作受限的线性表，它们的逻辑结构相同。

线性表——具有相同类型的数据元素的有限序列。

限制插入、删除位置

栈——仅在表尾进行插入和删除操作的线性表。

队列——在一端进行插入操作，而另一端进行删除操作的线性表。

本章作业

习题3（P77）：4,5