第3章 栈和队列

本章的基本内容是:

特殊的线性表——栈、队列

- 从数据结构角度看，栈和队列是操作受限的线性表，他们的逻辑结构相同。

特殊线性表——栈

栈的逻辑结构

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。

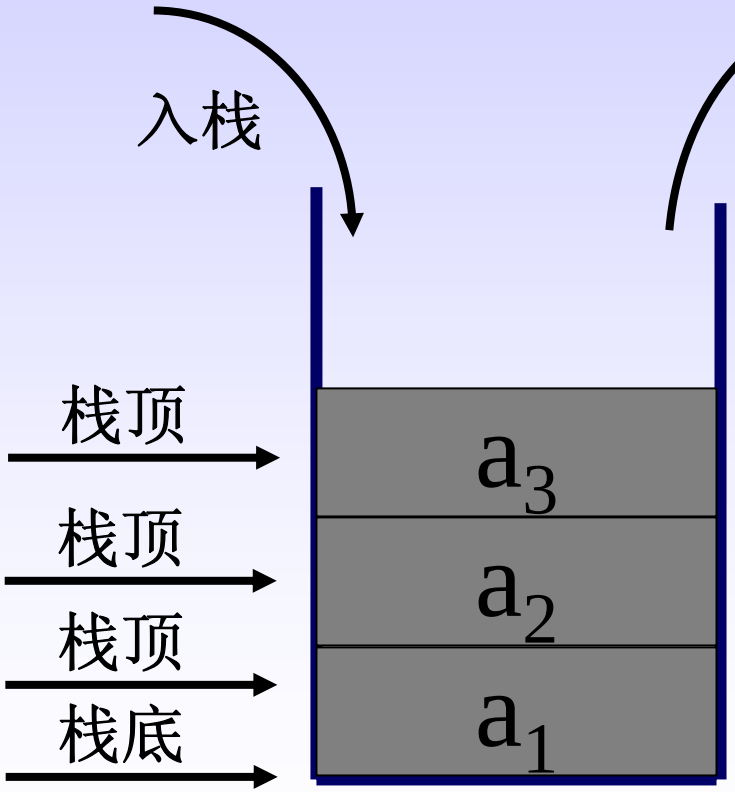
允许插入和删除的一端称为栈顶，另一端称为栈底。

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

栈底　　栈顶

特殊线性表——栈

栈的示意图

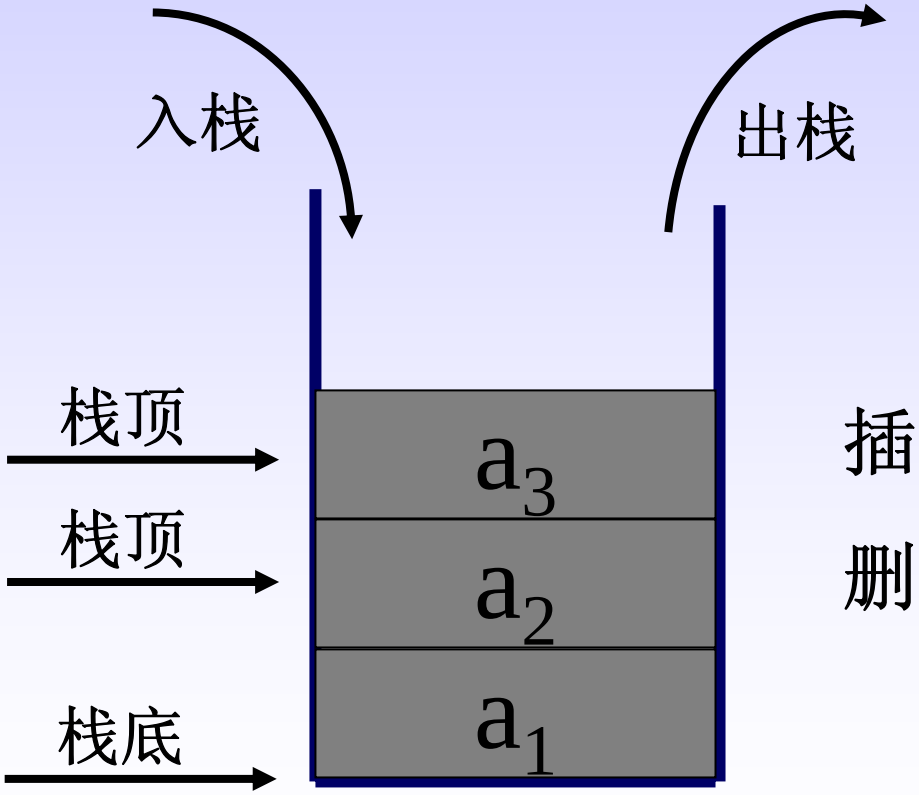


插入：入栈、进栈、压栈

删除：出栈、弹栈

特殊线性表——栈

栈的示意图



插入：入栈、进栈、压栈

删除：出栈、弹栈

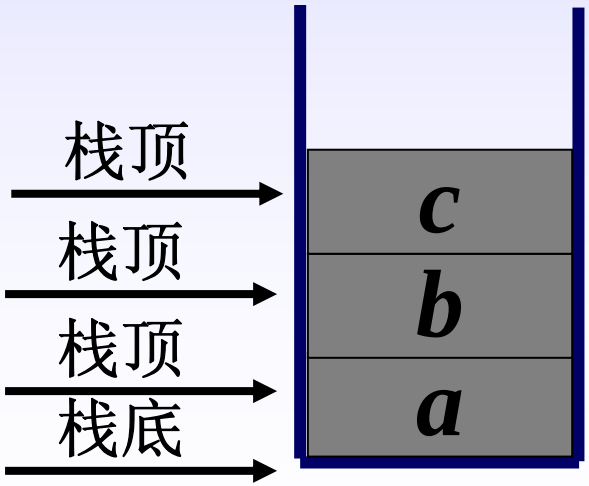
栈的操作特性： 后进先出(LIFO)

特殊线性表——栈

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：

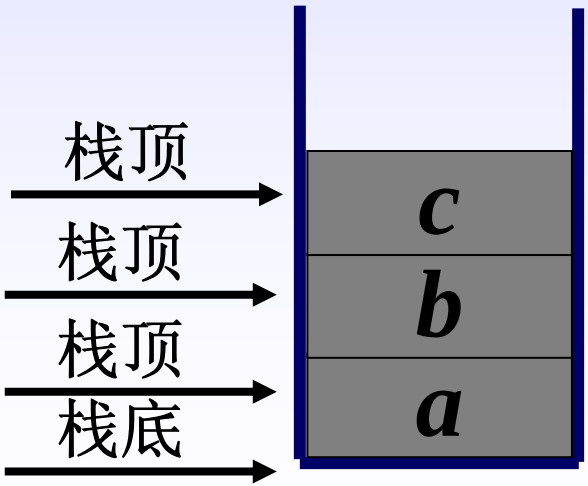


特殊线性表——栈

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*

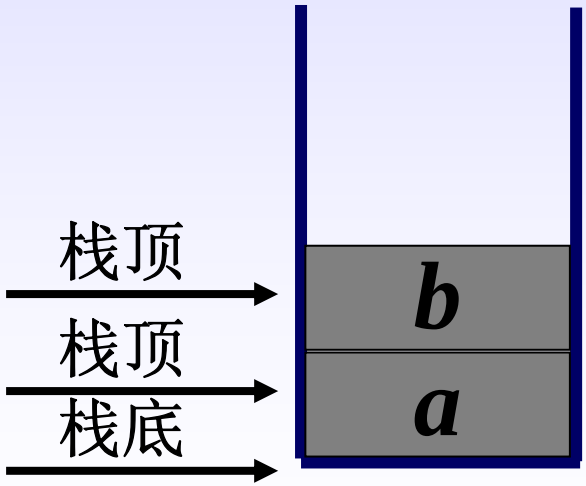
出栈序列：*c*、*b*、*a*

特殊线性表——栈

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：



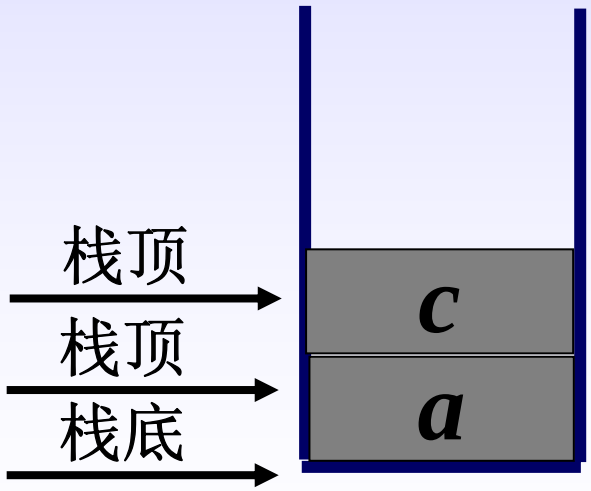
出栈序列：*b*

特殊线性表——栈

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。

特殊线性表——栈

栈的抽象数据类型定义

ADT Stack

Data

栈中元素具有相同类型及后进先出特性，
相邻元素具有前驱和后继关系

Operation

InitStack

前置条件：栈不存在
输入：无
功能：栈的初始化
输出：无
后置条件：构造一个空栈

特殊线性表——栈

栈的抽象数据类型定义

DestroyStack

前置条件：栈已存在
输入：无
功能：销毁栈
输出：无
后置条件：释放栈所占用的存储空间

Push

前置条件：栈已存在
输入：元素值x
功能：在栈顶插入一个元素x
输出：如果插入不成功，抛出异常
后置条件：如果插入成功，栈顶增加了一个元素

特殊线性表——栈

栈的抽象数据类型定义

Pop

前置条件：栈已存在
输入：无
功能：删除栈顶元素
输出：如果删除成功，返回被删元素值，否则，抛出异常
后置条件：如果删除成功，栈减少了一个元素

GetTop

前置条件：栈已存在
输入：无
功能：读取当前的栈顶元素
输出：若栈不空，返回当前的栈顶元素值
后置条件：栈不变

特殊线性表——栈

栈的抽象数据类型定义

Empty

前置条件：栈已存在

输入：无

功能：判断栈是否为空

输出：如果栈为空，返回1，否则，返回0

后置条件：栈不变

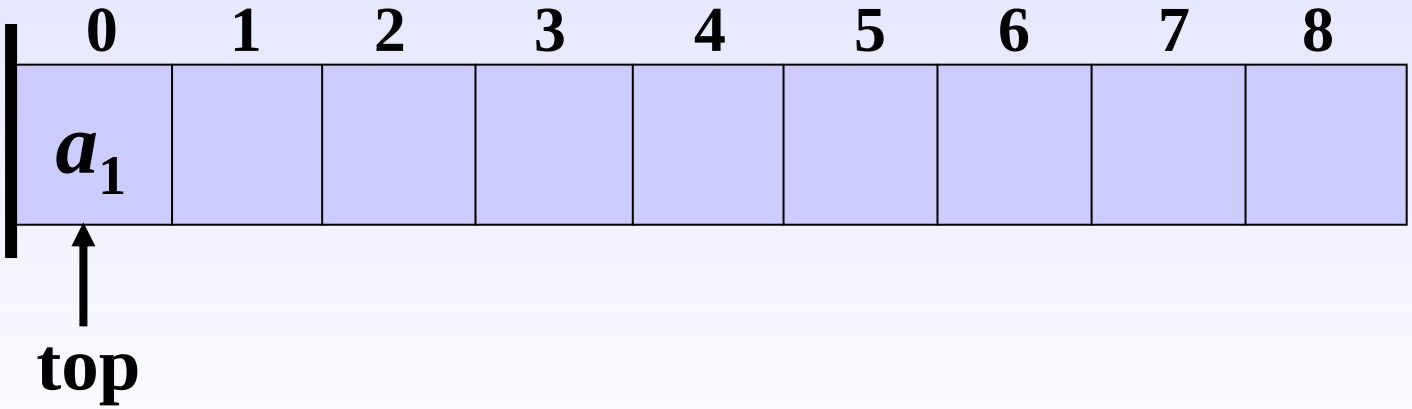
endADT

特殊线性表——栈

栈的顺序存储结构及实现

栈的顺序存储结构——顺序栈

如何改造数组实现栈的顺序存储？

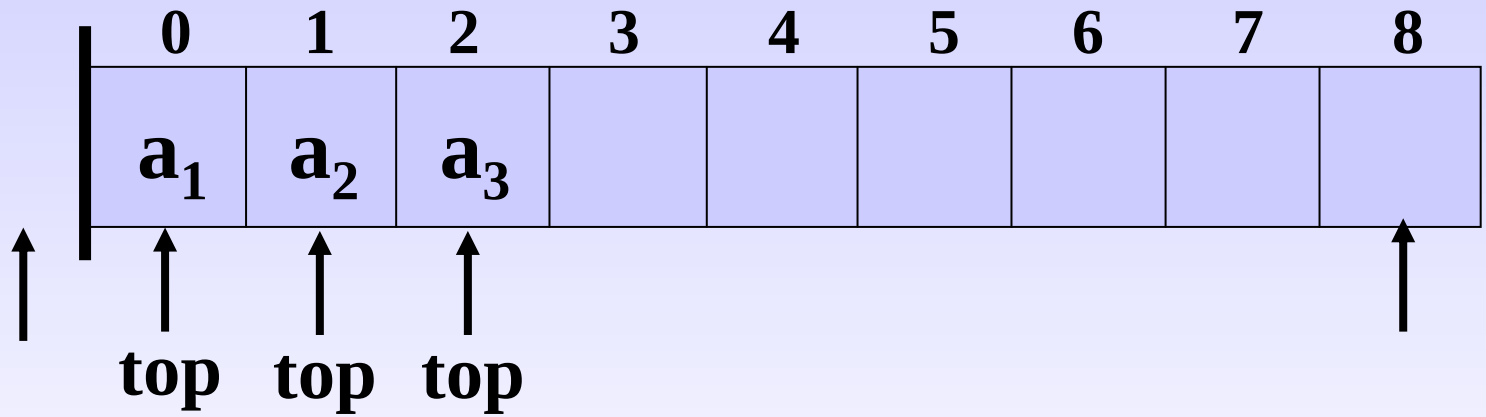


确定用数组的哪一端表示栈底。

附设指针top指示栈顶元素在数组中的位置。

特殊线性表——栈

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= StackSize-1

特殊线性表——栈

顺序栈类的声明

```
const  int  StackSize=100;
template  <class T>
class  SeqStack
{
   public:
      SeqStack ( ){top=-1; } ;
      ~SeqStack ( );
      void  Push ( T  x );
      T   Pop ( );
      T   GetTop ( ) {if (top!=-1)  return data[top];}
      bool  Empty ( ) {
         if(top==-1) return 1; else return 0;}
   private:
      T  data[StackSize];
      int  top;
}
```

特殊线性表——栈

顺序栈的实现——入栈

操作接口： **void Push(T x);**

```
template <class T>
void SeqStack::Push ( T x)
{
    if (top==StackSize−1) throw “溢出”;
    top++;
    data[top]=x;
 }
```

时间复杂度？

特殊线性表——栈

顺序栈的实现——出栈

操作接口： **T Pop( );**

```
template <class T>
T SeqStack:: Pop ( )
{
    if (top==−1) throw “溢出”;
    x=data[top−−];
    return x;
}
```

时间复杂度？

特殊线性表——栈

两栈共享空间

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？

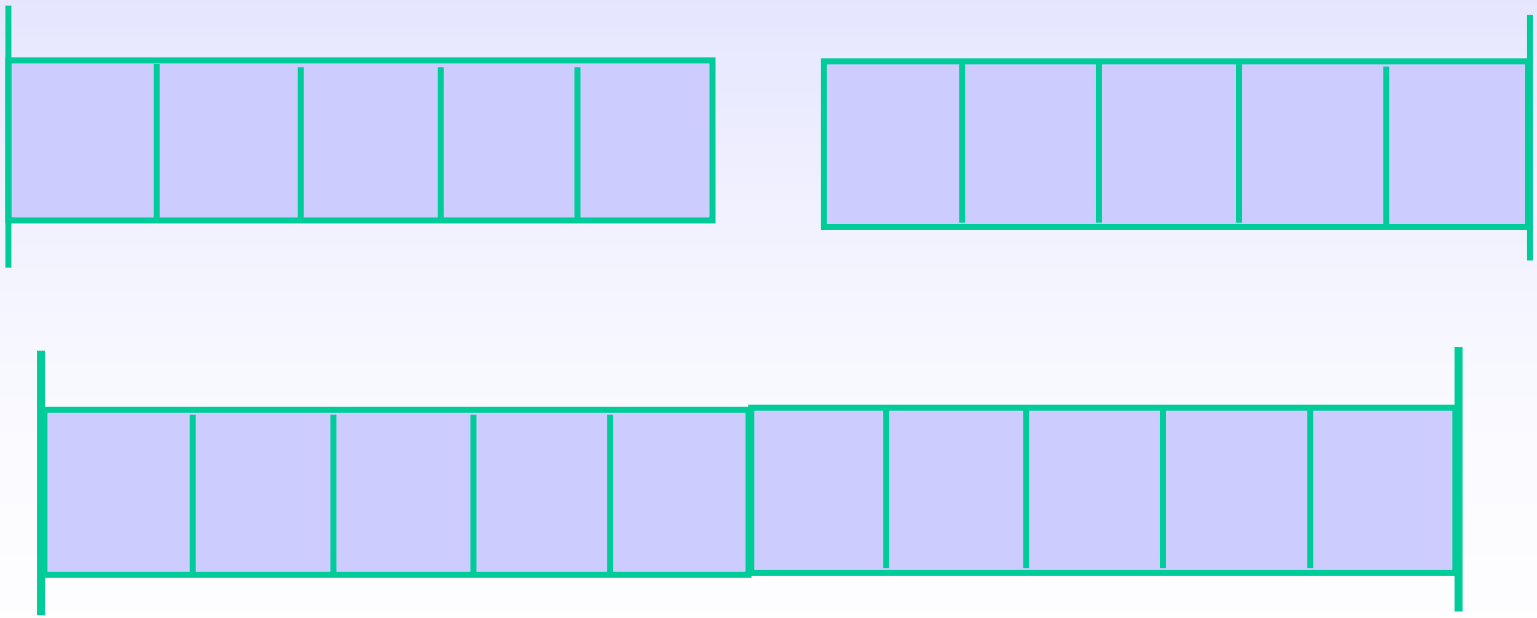
解决方案2：

顺序栈单向延伸——使用一个数组来存储两个栈

特殊线性表——栈

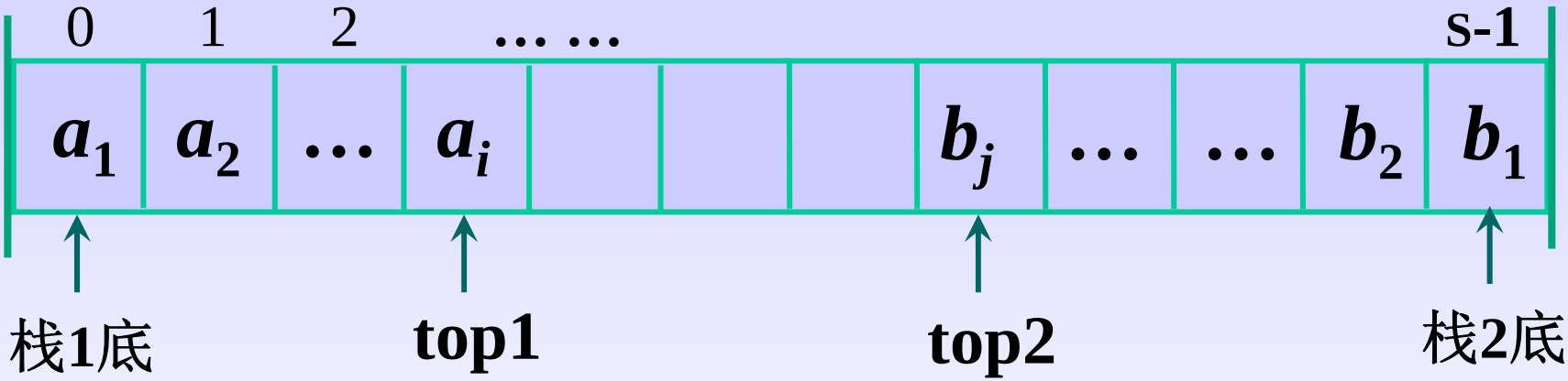
两栈共享空间

两栈共享空间：使用一个数组来存储两个栈，让一个栈的栈底为该数组的始端，另一个栈的栈底为该数组的末端，两个栈从各自的端点向中间延伸。



两栈共享空间

两栈共享空间



栈1的底固定在下标为0的一端；

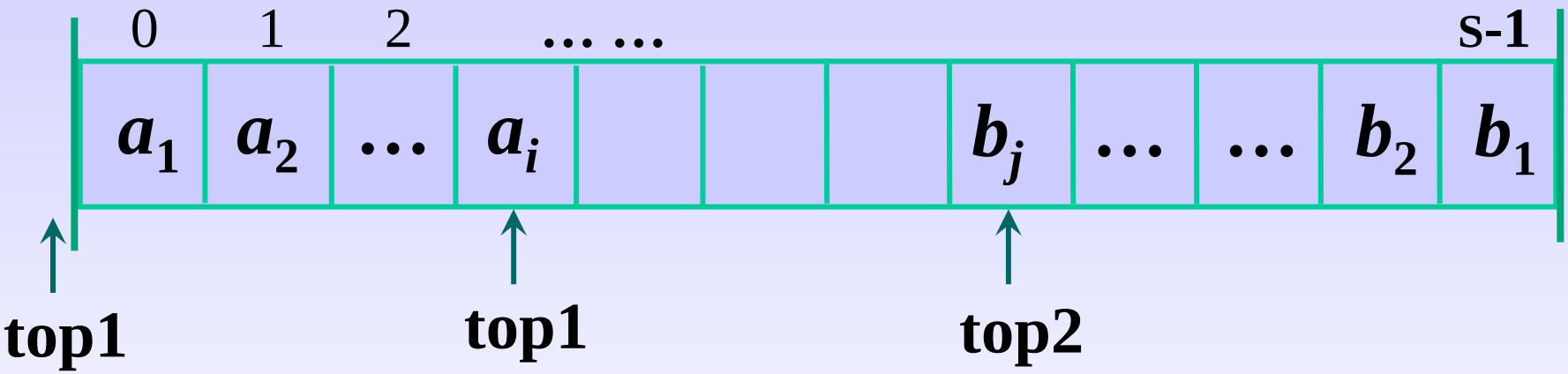
栈2的底固定在下标为StackSize-1的一端。

top1和top2分别为栈1和栈2的栈顶指针；

StackSize为整个数组空间的大小（图中用S表示）；

两栈共享空间

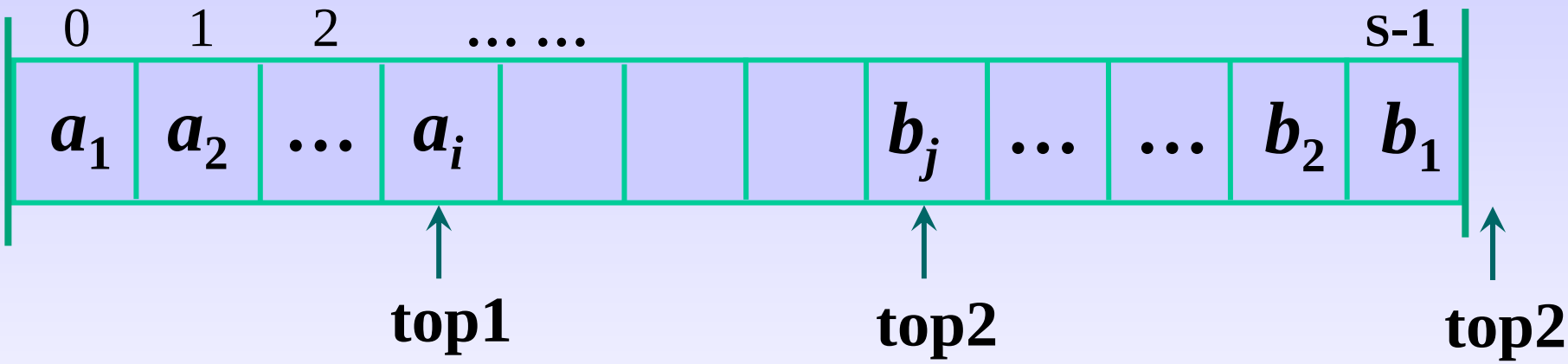
两栈共享空间



什么时候栈1为空？ top1= -1

两栈共享空间

两栈共享空间

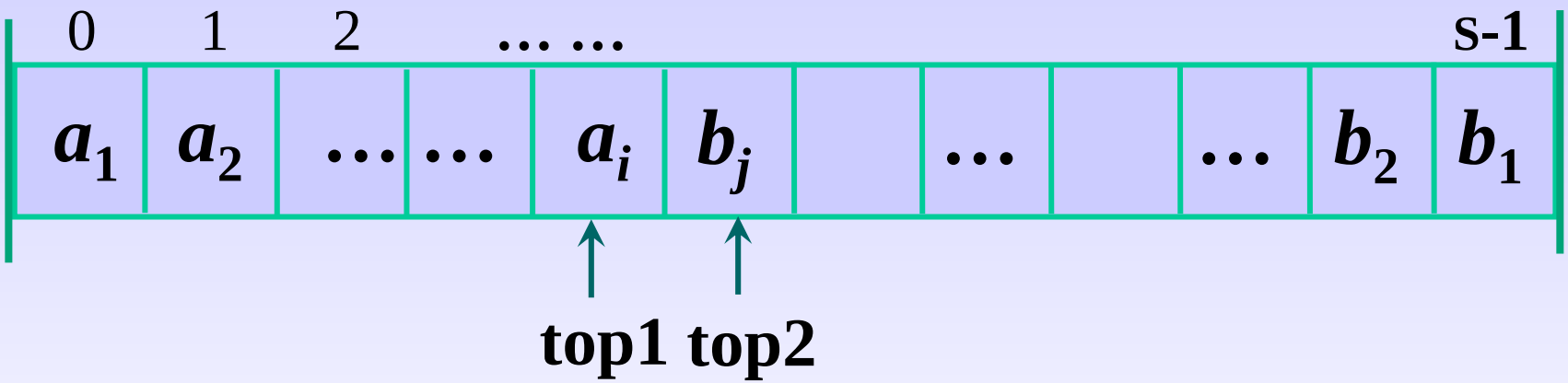


什么时候栈1为空？ top1= -1

什么时候栈2为空？ top2= StackSize

两栈共享空间

两栈共享空间



什么时候栈1为空？ top1= -1

什么时候栈2为空？ top2= StackSize

什么时候栈满？ top2= top1+1

两栈共享空间

两栈共享空间类的声明

```
const int StackSize=100;
template <class T>
class BothStack
{
  public:
      BothStack( ) {top1= -1;  top2=StackSize;}
      ~BothStack( );
      void Push(int i, T x);
      T Pop(int i);
      T GetTop(int i);
      bool Empty(int i);
  private:
      T data[StackSize];
      int top1, top2;
};
```

两栈共享空间

两栈共享空间的实现——插入

操作接口：**void Push(int i, T x)；**

1. 如果栈满，则抛出上溢异常；
2. 判断是插在栈1还是栈2；
2.1 若在栈1插入，则
2.1.1 top1加1；
2.1.2 在top1处填入x；
2.2 若在栈2插入，则
2.2.1 top2减1；
2.2.2 在top2处填入x；

两栈共享空间

两栈共享空间的实现——插入

操作接口：**void Push(int i, T x) ;**

```
template <class T>
void BothStack<T>::Push(int i, T x )
{
   if (top1==top2-1)                throw "上溢";
   if (i==1)            data[++top1]=x;
   if (i==2)            data[--top2]=x;
}
```

两栈共享空间

两栈共享空间的实现——删除

操作接口：**T Pop(int i)；**

1. 若是在栈1删除，则
   1.1 若栈1为空栈，抛出下溢异常；
   1.2 删除并返回栈1的栈顶元素；
2. 若是在栈2删除，则
   2.1 若栈2为空栈，抛出下溢异常；
   2.2 删除并返回栈2的栈顶元素；

两栈共享空间

两栈共享空间的实现——删除

操作接口：**T Pop(int i) ;**

```
T BothStack<T>::Pop(int i)
{
   if (i==1) //将栈1的栈顶元素出栈
    {     if (top1== -1)    throw "下溢";
      return data[top1--];       }
   if (i==2)  //将栈2的栈顶元素出栈
    {  if (top2==StackSize) throw "下溢";
        return data[top2++];    }
}
```

两栈共享空间

两栈共享空间的实现——读取栈i当前的栈顶元素

操作接口：**T GetTop(int i);**

```
template <class T>
T BothStack<T>::GetTop(int i)
{
    if(i==1)
        {     if (top1!=-1) return data[top1]; }
    if(i==2)
        {    if(top2!=StackSize) return data[top2]; }
}
```

两栈共享空间

两栈共享空间的实现——判断栈i是否为空栈

操作接口： **bool Empty(int i) ;**

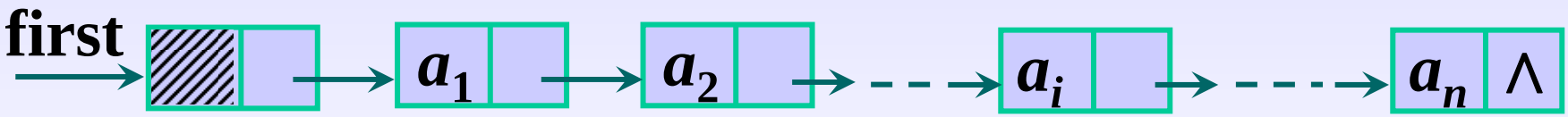
```
template <class T>
bool BothStack<T>::Empty(int i)
{
    if(i==1)
        {  if(top1==-1) return 1;  else   return 0;     }
    if(i==2)
        {  if(top2==StackSize) return 1;
           else return 0;     }
}
```

特殊线性表——栈

栈的链接存储结构及实现

栈的链接存储结构——链栈

如何改造链表实现栈的链接存储？



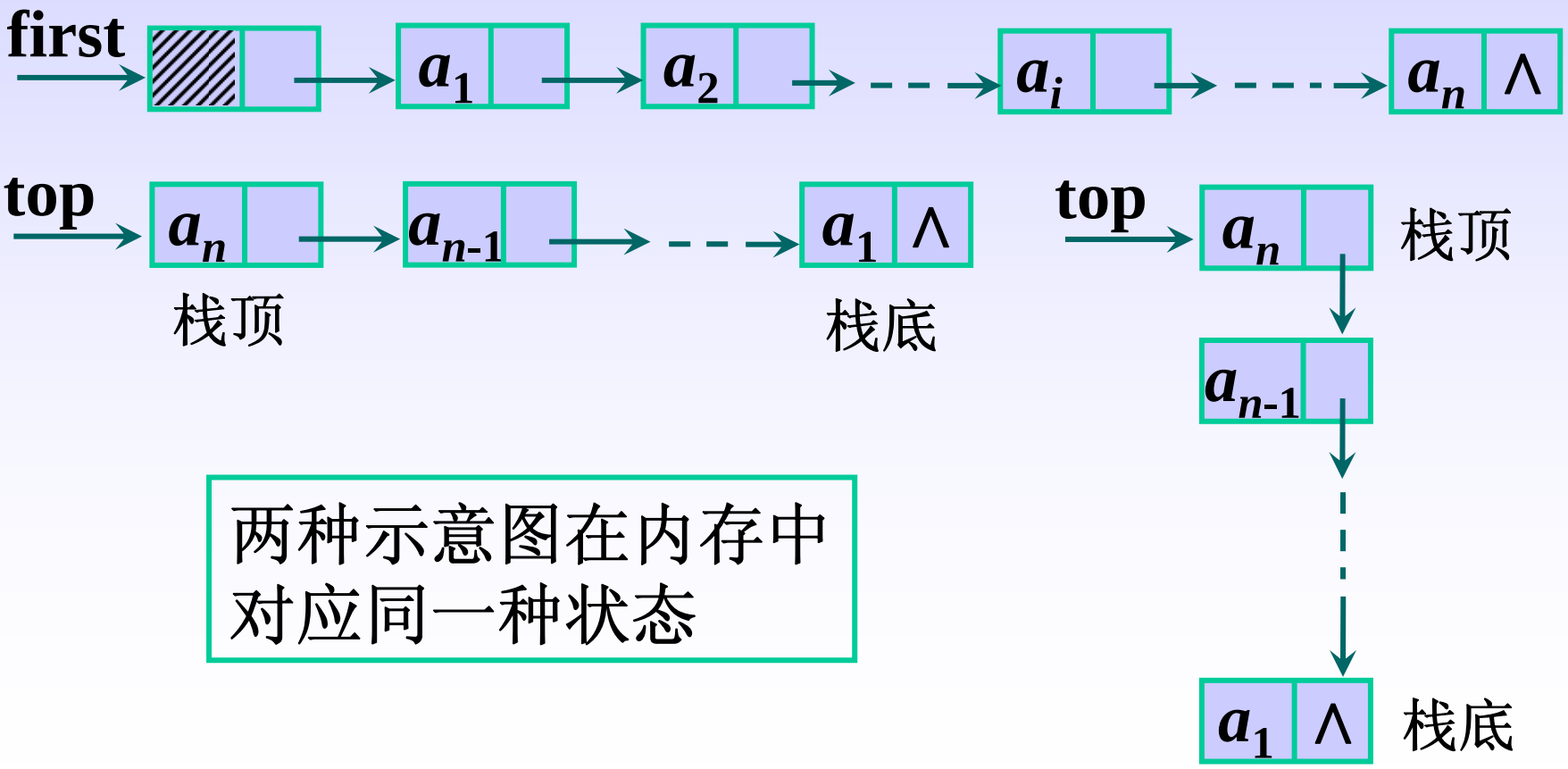
将哪一端作为栈顶？ 将链头作为栈顶，方便操作。

链栈需要加头结点吗？ 链栈不需要附设头结点。

特殊线性表——栈

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

特殊线性表——栈

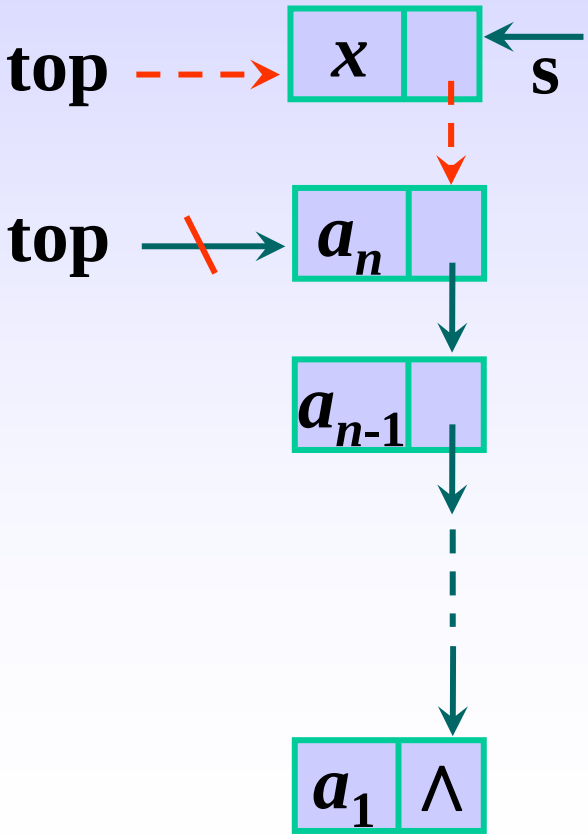
链栈的类声明

```
template <class T>
class LinkStack
{
  public:
      LinkStack(){top=NULL; }
      ~LinkStack( );
      void Push(T x);
      T Pop( );
      T GetTop( ){if(top!=NULL)return top->data; }
      bool Empty ( ){return(top==NULL? 1: 0); }
  private:
      Node<T> *top;
}
```

特殊线性表——栈

链栈的实现——插入

操作接口： void Push(T x);



为什么没有判断栈满？

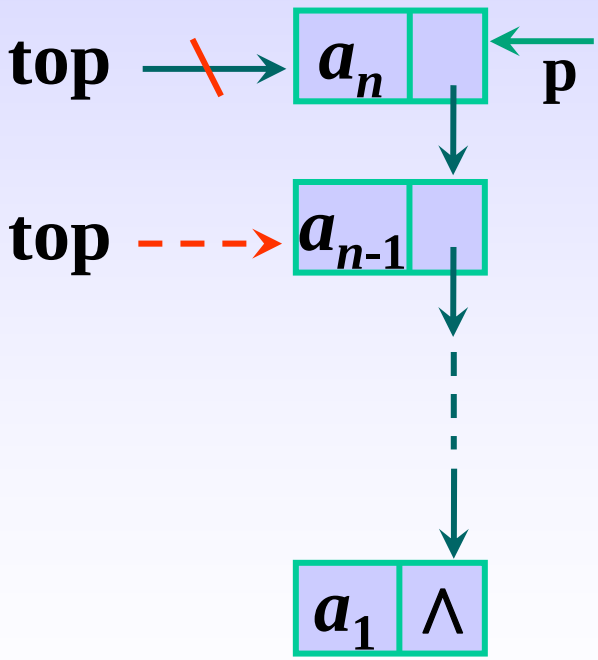
算法描述：

```
template <class T>
void LinkStack::Push(T x)
{
    s=new Node<T>;
    s->data=x;
    s->next=top;
   top=s;
}
```

特殊线性表——栈

链栈的实现——插入

操作接口： T Pop( );



top++可以吗？

算法描述：

```
template <class T>
T LinkStack::Pop( )
{
    if (top==NULL)
        throw "下溢";
    x=top->data;
    p=top;
    top=top->next;
    delete p;
    return x;
}
```

特殊线性表——栈

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。

栈的应用举例—递归

1 递归的定义

子程序（或函数）直接调用自己或通过一系列调用语句间接调用自己，是一种描述问题和解决问题的基本方法。

2 递归的基本思想

问题分解：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。

栈的应用举例—递归

3 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。

栈的应用举例—递归

例1　阶乘函数

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

递归算法

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

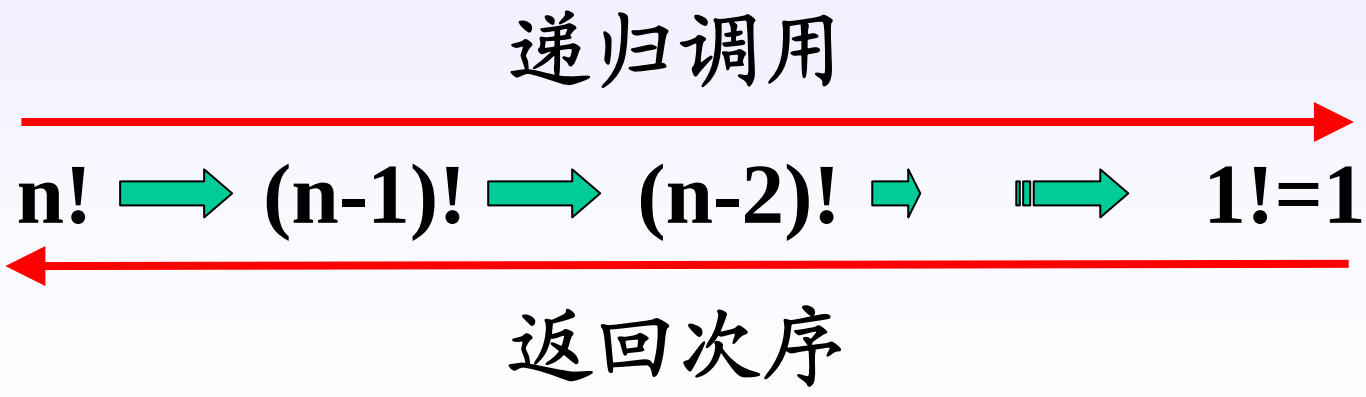
栈的应用举例—递归

求解阶乘 n! 的过程

栈的应用举例—递归

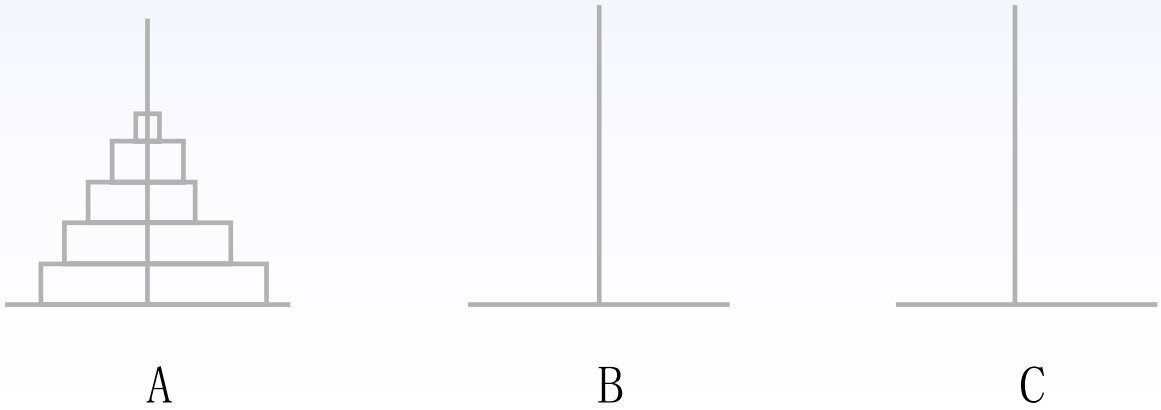
递归过程与递归工作栈

- 递归过程在实现时，需要自己调用自己。
- 层层向下递归，返回次序正好相反:



递归的经典问题——汉诺塔问题

在印度，有一个古老的传说：在世界中心贝拿勒斯（在印度北部）的圣庙里，一块黄铜板上插着三根宝石针。印度教的主神梵天在创造世界的时候，在其中一根针上从下到上穿好了由大到小的64片金片，这就是所谓的汉诺塔。不论白天黑夜，总有一个僧侣在按照下面的法则移动这些金片：一次只移动一片，不管在哪根针上，小片必须在大片上面。

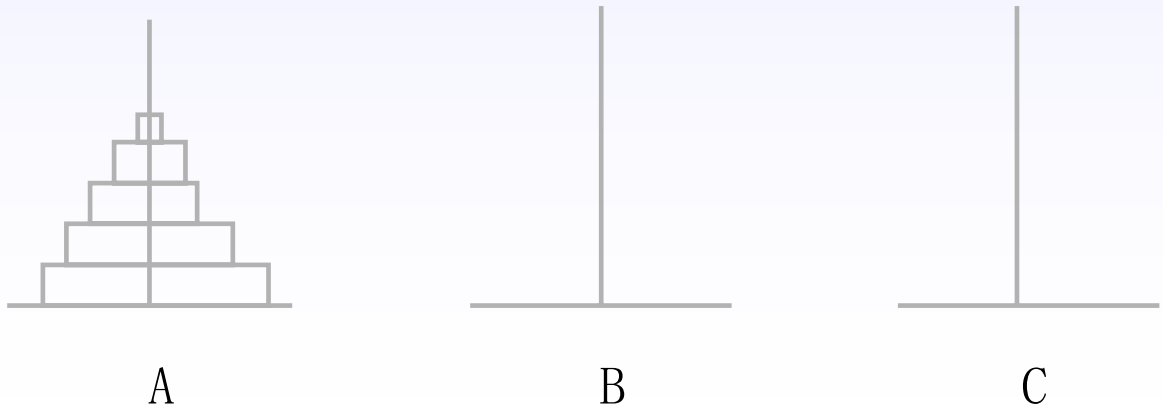


递归的经典问题——汉诺塔问题

僧侣们预言，当所有的金片都从梵天穿好的那根针上移到另外一根针上时，世界就将在一声霹雳中消灭，而梵塔、庙宇和众生也都将同归于尽。

这个传说的可信度有多大?

考虑一下把64片金片，由一根针上移到另一根针上，始终保持上小下大的顺序。需要多少次移动?



递归的经典问题——汉诺塔问题

这里需要递归的方法。

假设有n片，移动次数是f(n).显然

f(1)=1,f(2)=3,f(3)=7,且f(k+1)=2*f(k)+1。

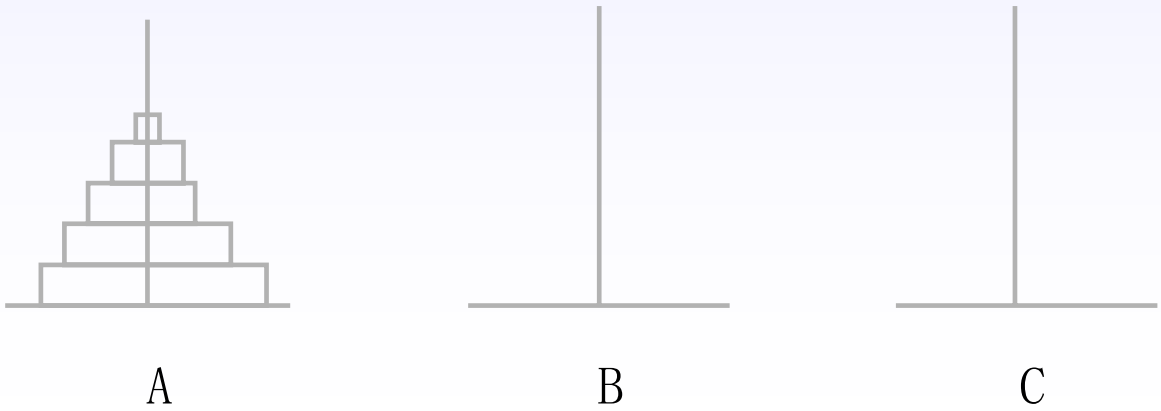
可以证明$f(n)=2^n-1$。

n=64时，$f(64)= 2^{64}-1=18446744073709551615$

假如每秒钟一次，共需多长时间？一年大约有31536926 秒，计算表明移完这些金片需要5800多亿年，比地球寿命还要长，事实上，那时世界、梵塔、庙宇和众生都早已经灰飞烟灭。

例 汉诺塔问题

有三根针A、B、C。A针上有n个盘子，大的在下，小的在上，要求把这n个盘子从A针移到C针，在移动过程中可以借助B针，每次只允许移动一个盘，且在移动过程中在三根针上都保持大盘在下，小盘在上。



栈的应用举例—递归

汉诺塔问题的递归求解分析：

将n 个盘子从A针移到C针步骤：

如果 n = 1，则将这一个盘子直接从 塔A移到塔 C 上。

否则，执行以下三步：

①将A 上n-1个盘子移到 B针上（借助C针）；

②把A针上剩下的一个盘子移到C针上；

③将n-1个盘子从B针移到C针上（借助A针）；

事实上，上面三个步骤包含两种操作：

①将多个盘子从一个针移到另一个针上，这是一个递归的过程。 hanoi函数实现。

②将1个盘子从一个针上移到另一针上。用move函数实现。

```
#include <iostream>
using namespace std;
void move(char getone,char putone)
{ cout<<getone<<"-->"<<putone<<endl; }
void hanoi(int n,char one,char two,char three)
{   void move(char getone,char putone);
    if (n==1) move (one,three);
    else
    {  hanoi(n-1,one,three,two);
       move(one,three);
       hanoi(n-1,two,one,three);
    }
}
```

```
int main()
{
    void hanoi(int n,char one,char two,char three);
    int m;
    cout<<"Enter the number of diskes:";
    cin>>m;
    cout<<"the steps to moving "<<m
      <<" diskes:"<<endl;
    hanoi(m,'A','B','C');
}
```

运行结果：

Enter the number of diskes:3

the steps to moving 3 diskes:

A-->C

A-->B

C-->B

A-->C

B-->A

B-->C

A-->C

估计hanoi执行时间

```
#include <iostream>
#include <ctime>
using namespace std;
void move(char getone,char putone)
{
cout<<getone<<"-->"<<putone<<"   "<<endl; }
void hanoi(int n,char one,char two,char three)
{    void move(char getone,char putone);
     if (n==1) move (one,three);
     else
     {  hanoi(n-1,one,three,two);
        move(one,three);
        hanoi(n-1,two,one,three);
     }
}
```

标准库ctime里时间函数
time_t time (0)
计算从1970年1月1日到当前系统时间(秒)

```
int main()
{
void hanoi(int n,char one,char two,char three);
int m;
unsigned start,finish;
cout<<"Enter the number of diskes:";
cin>>m;
start=(unsigned)time(0);
cout<<"the steps to moving "<<m <<"
diskes:"<<endl;
hanoi(m,'A','B','C');
finish=(unsigned)time(0);
cout<<start<<"  "<<finish<<endl;
cout<< finish-start<<endl;
/*double d=104;
int i;
for(i=20;i<=64;i++)
{d=d*2;}
cout<<d/60/60/24/365;*/
}
```

栈的应用举例—递归

递归函数的内部执行过程

(1) 运行开始时，首先为递归调用建立一个工作栈，其结构包括值参、局部变量和返回地址；

(2) 每次执行递归调用之前，把递归函数的值参和局部变量的当前值以及调用后的返回地址压栈；

(3) 每次递归调用结束后，将栈顶元素出栈，使相应的值参和局部变量恢复为调用前的值，然后转向返回地址指定的位置继续执行。

框架 | 值　参

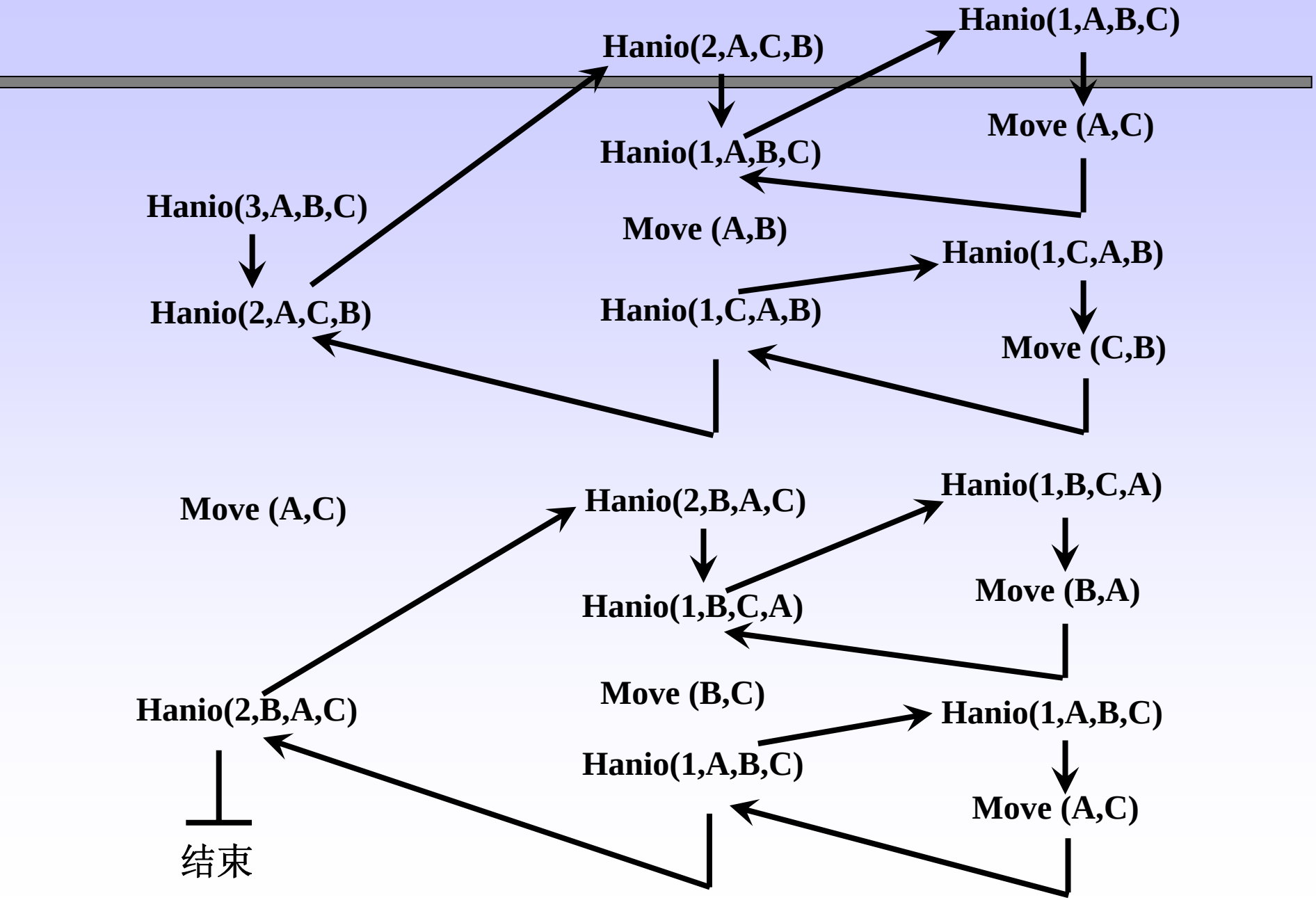
栈的应用举例—递归

递归函数的运行轨迹

(1) 写出函数当前调用层执行的各语句，并用有向弧表示语句的执行次序；

(2) 对函数的每个递归调用，写出对应的函数调用，从调用处画一条有向弧指向被调用函数入口，表示调用路线，从被调用函数末尾处画一条有向弧指向调用语句的下面，表示返回路线；

(3) 在返回路线上标出本层调用所得的函数值。


Hanio(1,A,B,C)
Hanio(2,A,C,B)
Move (A,C)
Hanio(1,A,B,C)
Hanio(3,A,B,C)
Move (A,B)
Hanio(1,C,A,B)
Hanio(2,A,C,B)
Hanio(1,C,A,B)
Move (C,B)
Move (A,C)
Hanio(2,B,A,C)
Hanio(1,B,C,A)
Move (B,A)
Hanio(1,B,C,A)
Move (B,C)
Hanio(2,B,A,C)
Hanio(1,A,B,C)
Hanio(1,A,B,C)
Move (A,C)
结束

特殊线性表——队列

队列的逻辑结构

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。

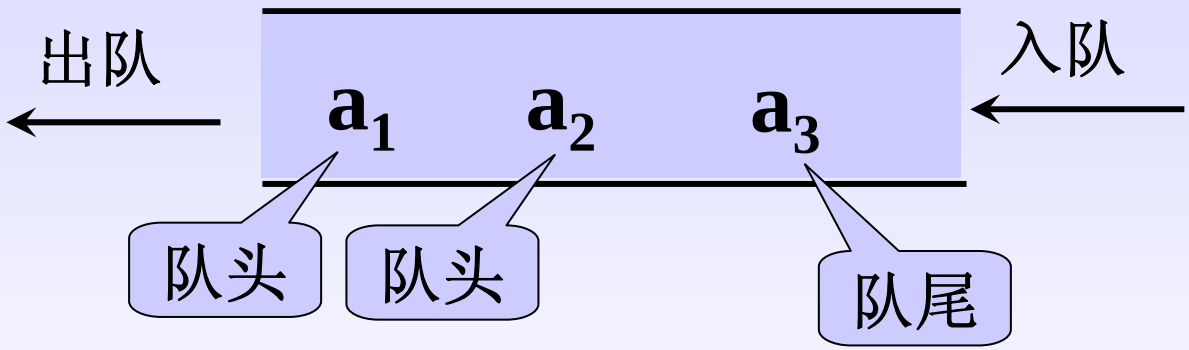
允许插入（也称入队、进队）的一端称为队尾，允许删除（也称出队）的一端称为队头。

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

队头　　队尾

特殊线性表——队列

队列的逻辑结构



队列的操作特性：先进先出(FILO)

特殊线性表——队列

队列的抽象数据类型定义

ADT Queue

Data

队列中元素具有相同类型及先进先出特性，
相邻元素具有前驱和后继关系

Operation

InitQueue

前置条件：队列不存在
输入：无
功能：初始化队列
输出：无
后置条件：创建一个空队列

特殊线性表——队列

队列的抽象数据类型定义

DestroyQueue

前置条件：队列已存在
输入：无
功能：销毁队列
输出：无
后置条件：释放队列所占用的存储空间

EnQueue

前置条件：队列已存在
输入：元素值x
功能：在队尾插入一个元素
输出：如果插入不成功，抛出异常
后置条件：如果插入成功，队尾增加了一个元素

特殊线性表——队列

队列的抽象数据类型定义

DeQueue

前置条件：队列已存在
输入：无
功能：删除队头元素
输出：如果删除成功，返回被删元素值
后置条件：如果删除成功，队头减少了一个元素

GetQueue

前置条件：队列已存在
输入：无
功能：读取队头元素
输出：若队列不空，返回队头元素
后置条件：队列不变

特殊线性表——队列

队列的抽象数据类型定义

Empty

前置条件：队列已存在

输入：无

功能：判断队列是否为空

输出：如果队列为空，返回**1**，否则，返回**0**

后置条件：队列不变

endADT

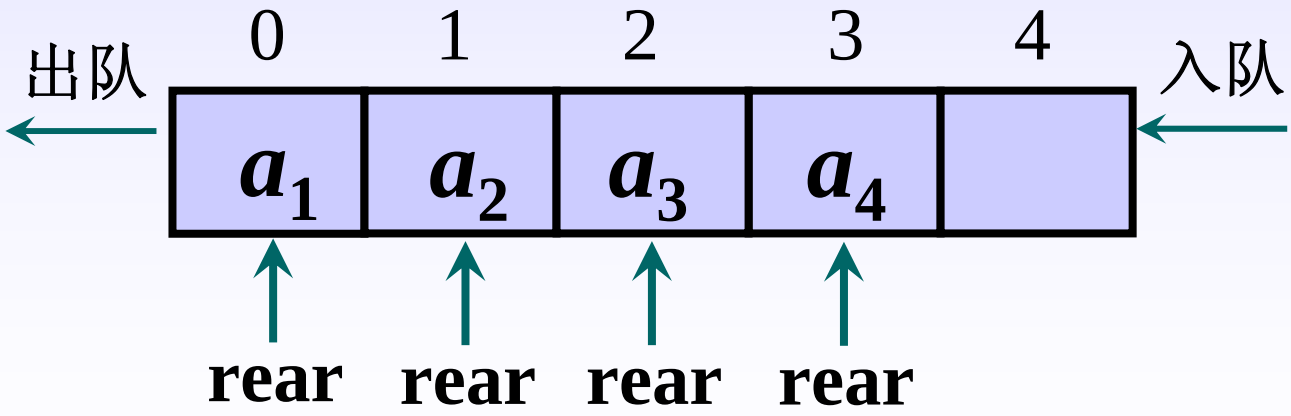
特殊线性表——队列

队列的顺序存储结构及实现

队列的顺序存储结构——循环队列

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



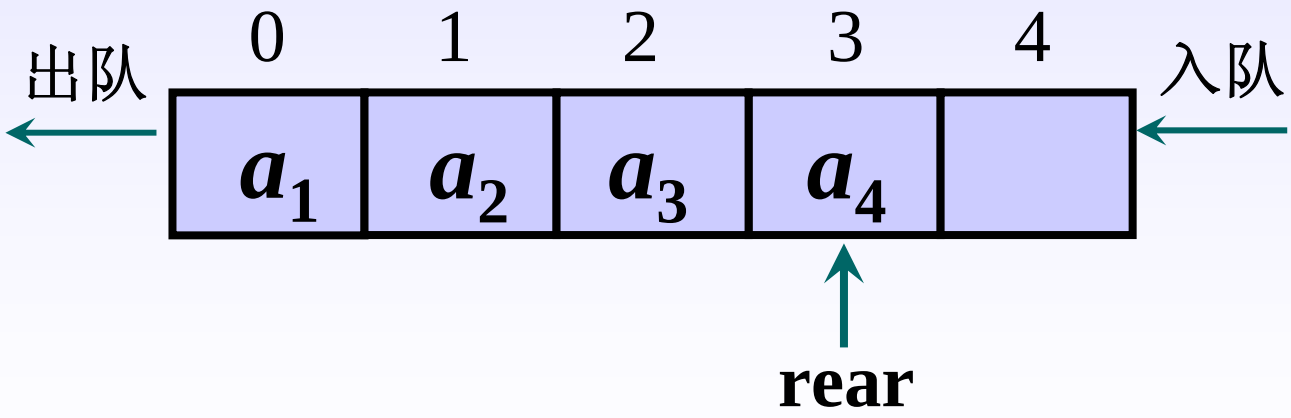
入队操作时间性能为$O(1)$

特殊线性表——队列

队列的顺序存储结构及实现

如何改造数组实现队列的顺序存储？

例：a_1a_2依次出队

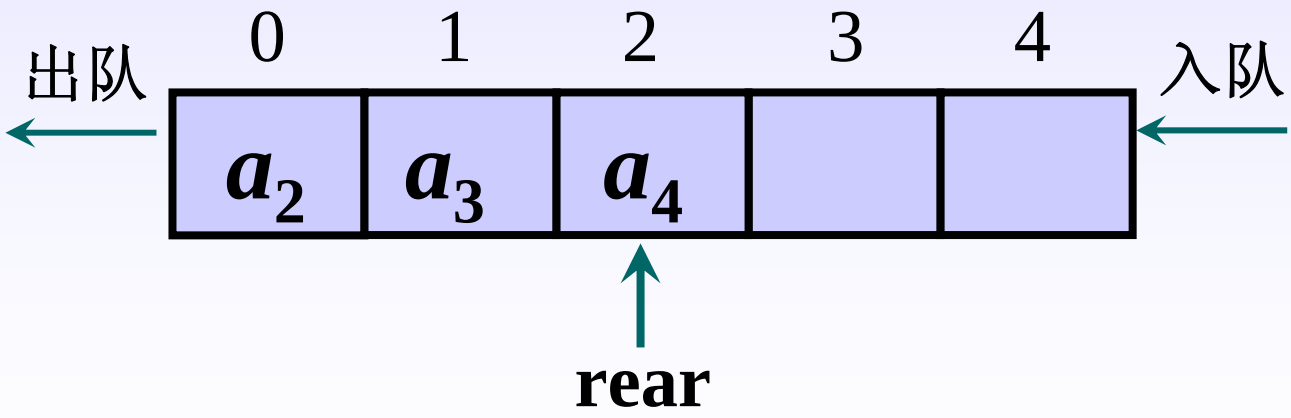


特殊线性表——队列

队列的顺序存储结构及实现

如何改造数组实现队列的顺序存储？

例：a_1a_2依次出队

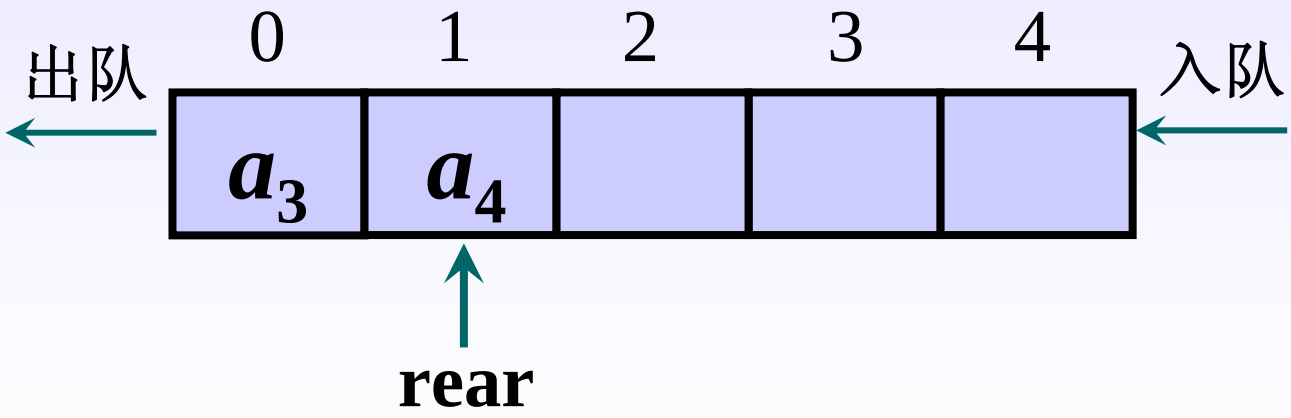


特殊线性表——队列

队列的顺序存储结构及实现

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*)

特殊线性表——队列

队列的顺序存储结构及实现

如何改进出队的时间性能？

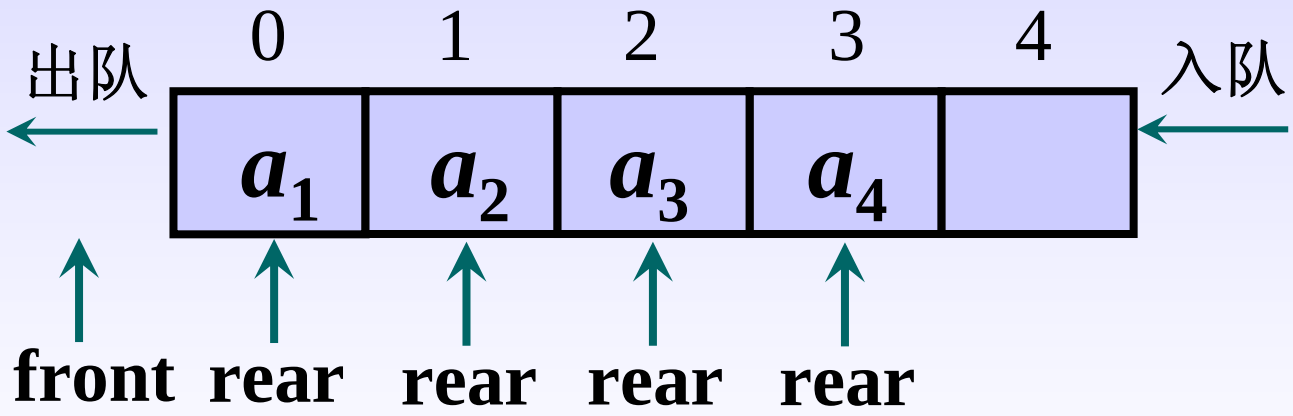
放宽队列的所有元素必须存储在数组的前***n***个单元这一条件，只要求队列的元素存储在数组中连续的位置。

设置队头、队尾两个指针

特殊线性表——队列

队列的顺序存储结构及实现

例：$a_1a_2a_3a_4$依次入队

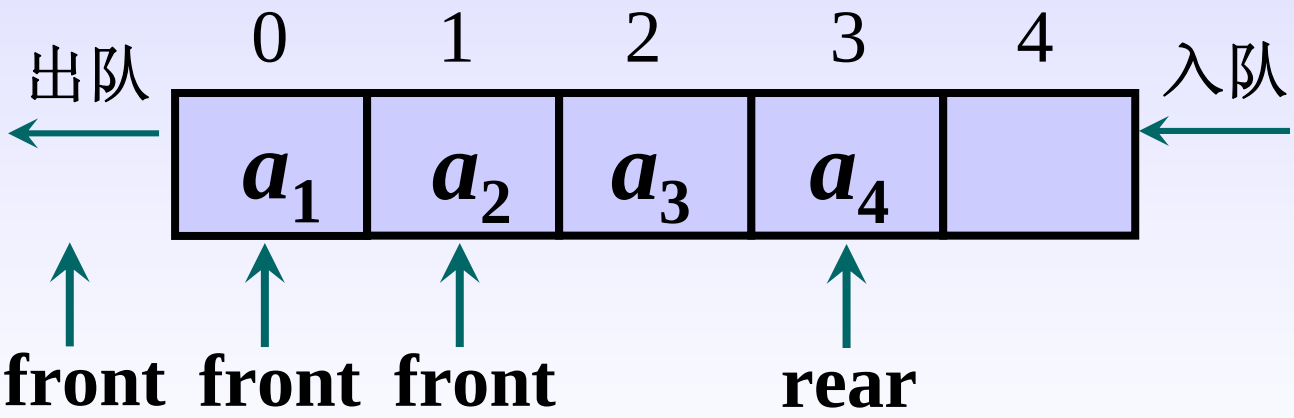


入队操作时间性能仍为$O(1)$

特殊线性表——队列

队列的顺序存储结构及实现

例：a_1a_2依次出队

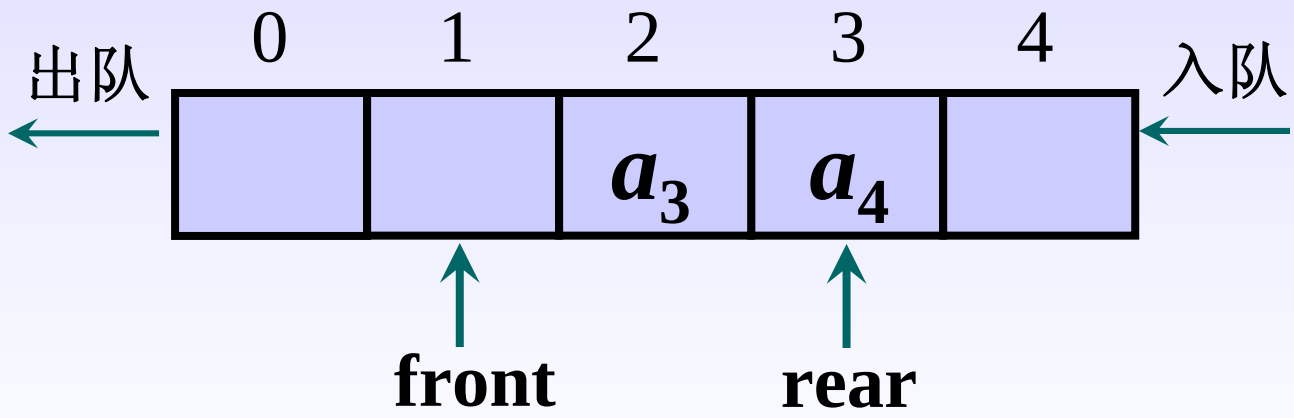


出队操作时间性能提高为$O(1)$

特殊线性表——队列

队列的顺序存储结构及实现

例：a_1a_2依次出队

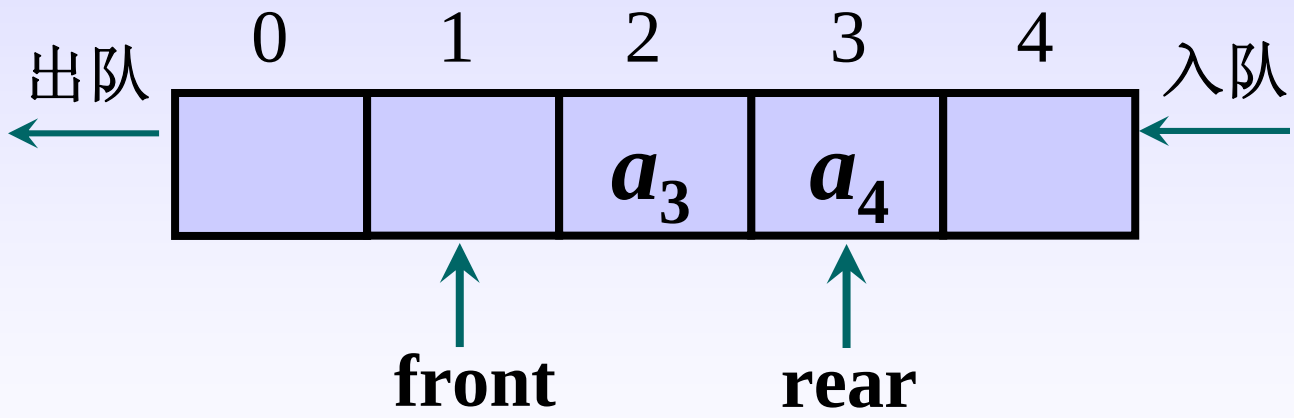


队列的移动有什么特点？

特殊线性表——队列

队列的顺序存储结构及实现

例：a_1a_2依次出队

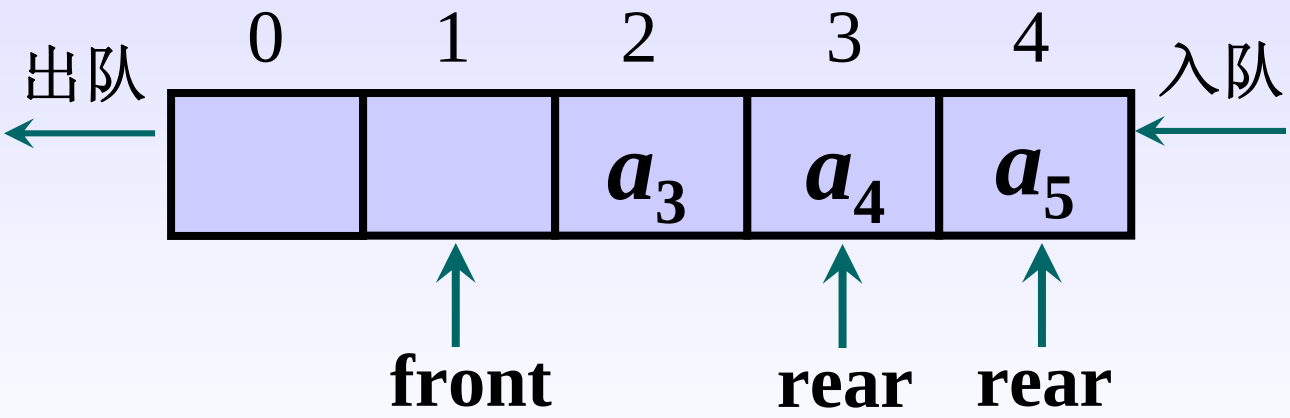


整个队列向数组下标较大方向移动 ➡ 单向移动性

特殊线性表——队列

队列的顺序存储结构及实现

继续入队会出现什么情况？

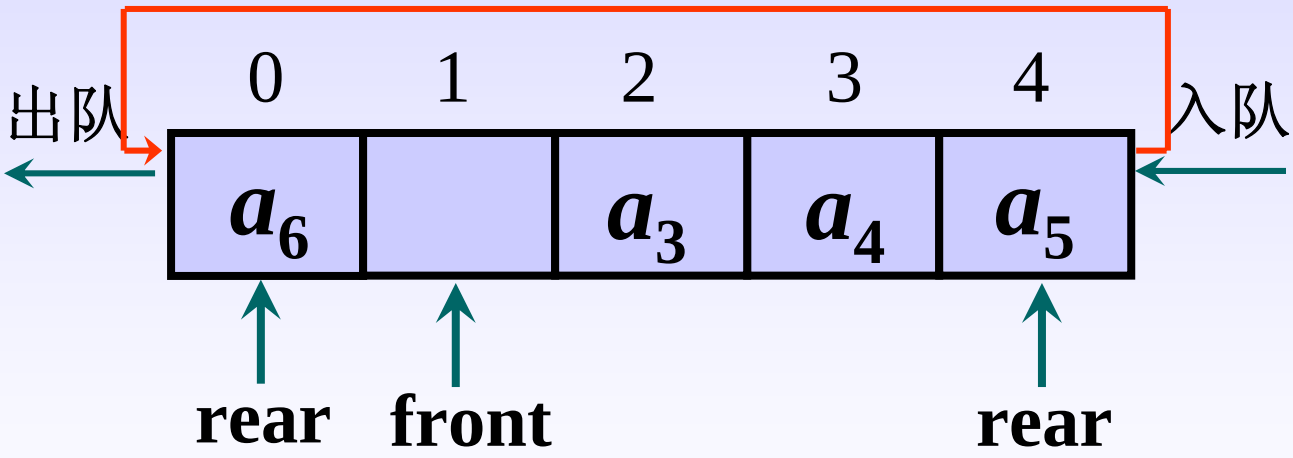


假溢出：当元素被插入到数组中下标最大的位置上之后，队列的空间就用尽了，尽管此时数组的低端还有空闲空间，这种现象叫做假溢出。

特殊线性表——队列

队列的顺序存储结构及实现

如何解决假溢出？

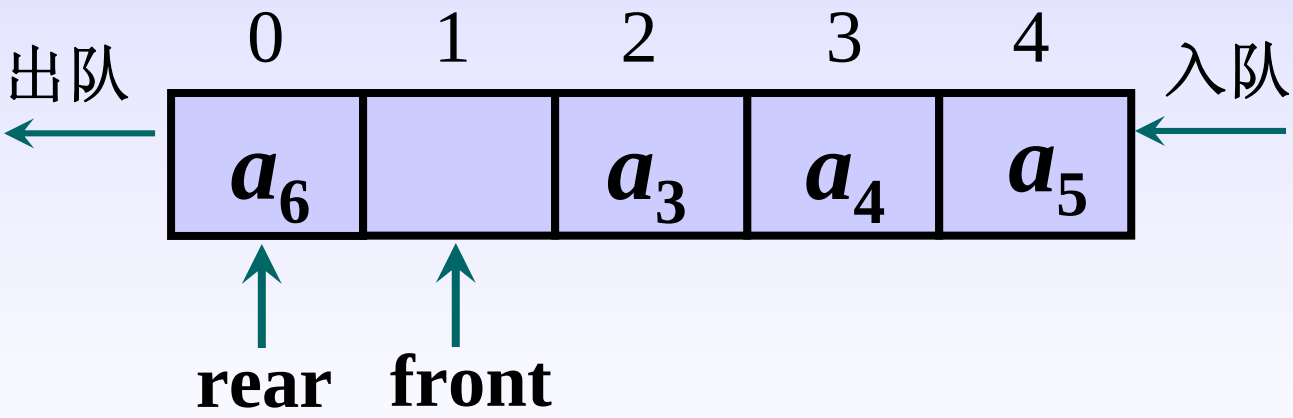


循环队列：将存储队列的数组头尾相接。

特殊线性表——队列

队列的顺序存储结构及实现

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。

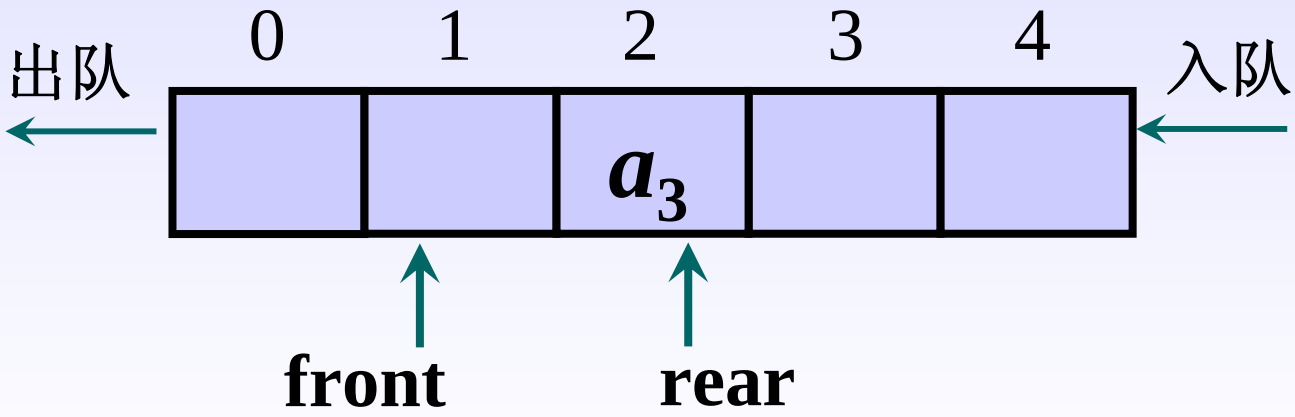
求模：（4＋1）mod 5＝0

特殊线性表——队列

队列的顺序存储结构及实现

如何判断循环队列队空？

队空的临界状态

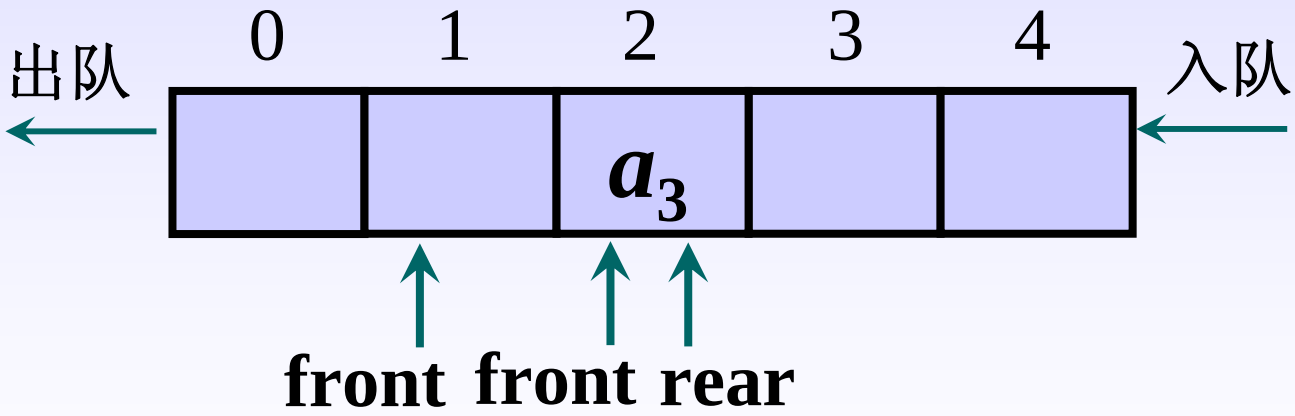


特殊线性表——队列

队列的顺序存储结构及实现

如何判断循环队列队空？

执行出队操作



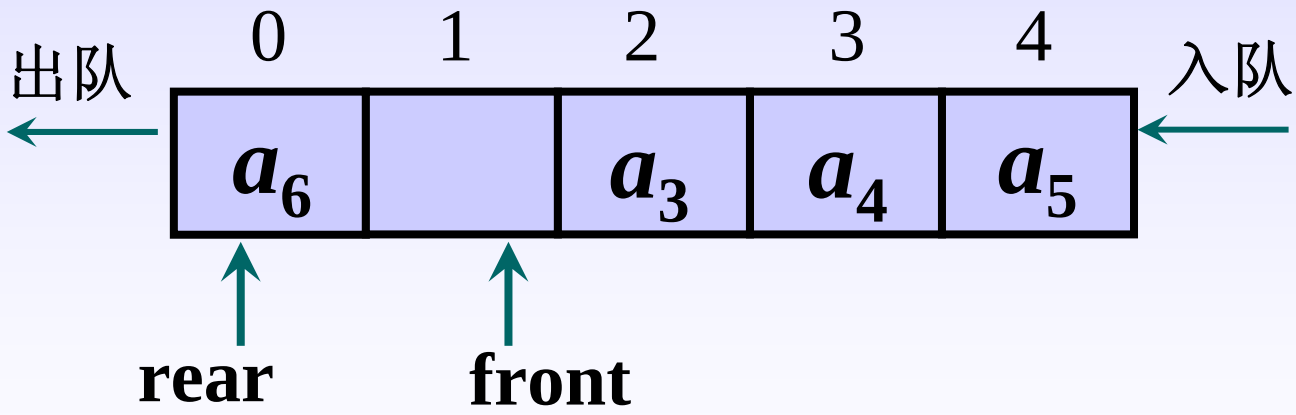
队空：**front=rear**

特殊线性表——队列

队列的顺序存储结构及实现

如何判断循环队列队满？

队满的临界状态

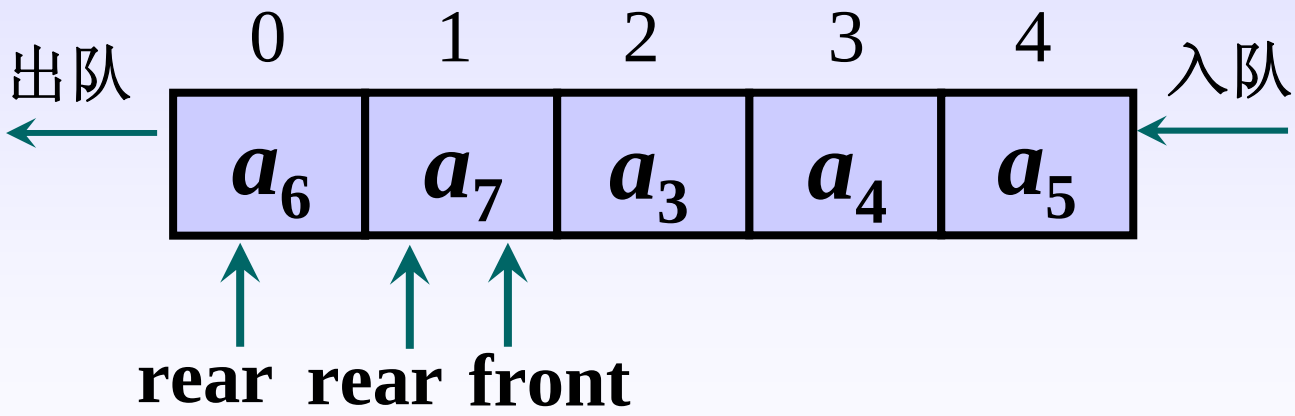


特殊线性表——队列

队列的顺序存储结构及实现

如何判断循环队列队满？

执行入队操作



队满：**front=rear**

特殊线性表——队列

队列的顺序存储结构及实现

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？

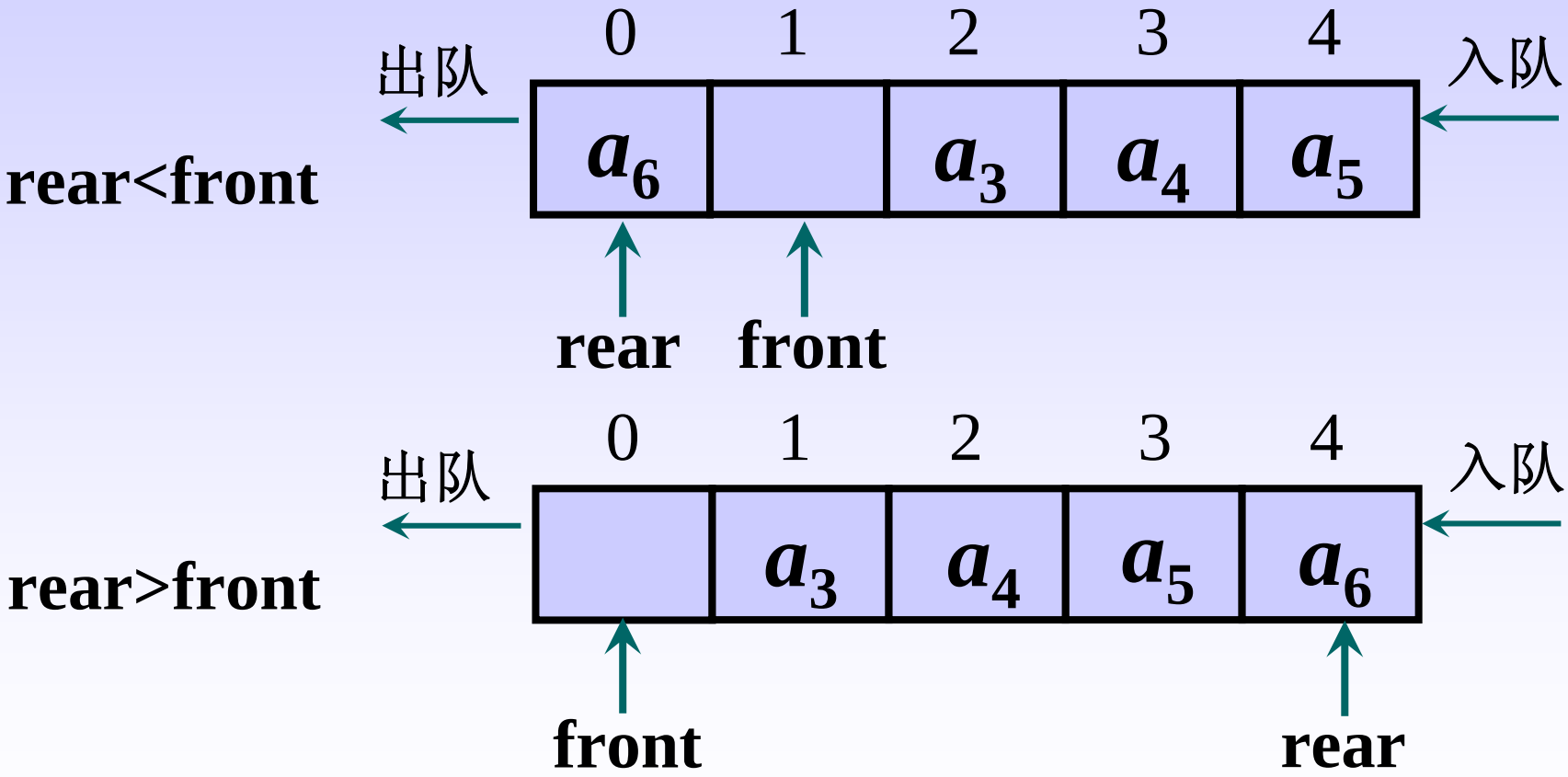
方法一：附设一个存储队列中元素个数的变量**num**，当**num=0**时队空，当**num=QueueSize**时为队满；

方法二：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；

方法三：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。

特殊线性表——队列

队列的顺序存储结构及实现



队满的条件：(rear+1) mod QueueSize=front

特殊线性表——队列

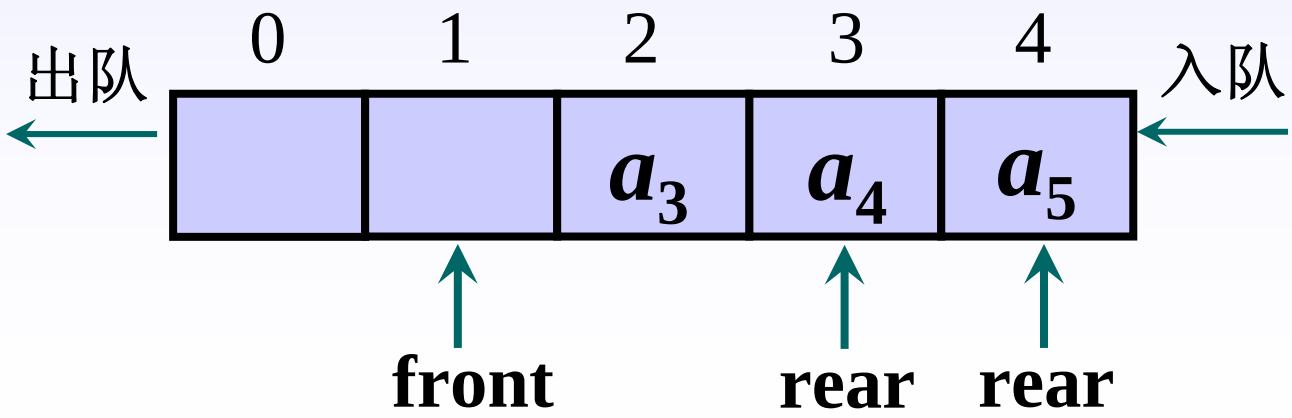
循环队列类的声明

```
const int QueueSize=100;
template <class T>
class CirQueue
{
  public:
     CirQueue( ){front=rear=QueueSize-1;}
     ~ CirQueue( );
     void EnQueue(T x);
    T DeQueue( );
    T GetQueue( );
    bool Empty( ){return(front==rear? 1: 0);}
  private:
    T data[QueueSize];
    int front, rear;
};
```

特殊线性表——队列

循环队列的实现——入队

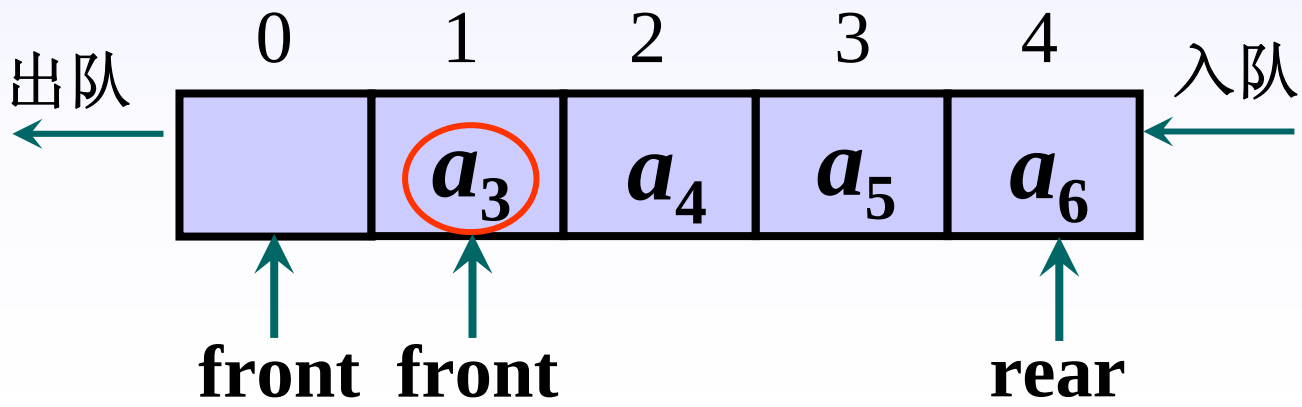
```
template <class T>
 void CirQueue::EnQueue（T x）
{
  if（（rear+1）% QueueSize ==front）throw "上溢";
  rear=（rear+1）% QueueSize;
  data[rear]=x;
}
```



特殊线性表——队列

循环队列的实现——出队

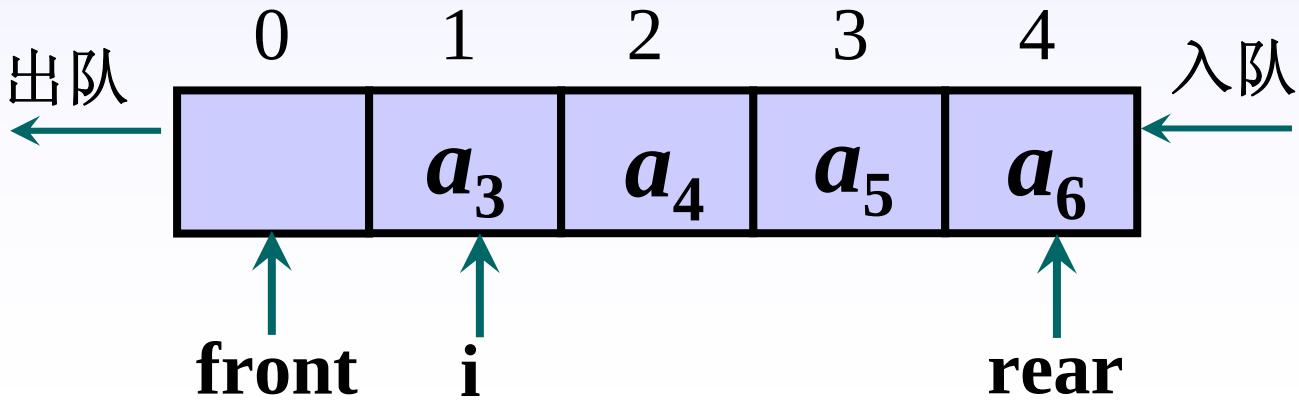
```
template <class T>
T CirQueue::DeQueue( )
{
    if (rear==front) throw "下溢";
    front=(front+1) % QueueSize;
    return data[front];
}
```



特殊线性表——队列

循环队列的实现——读队头元素

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

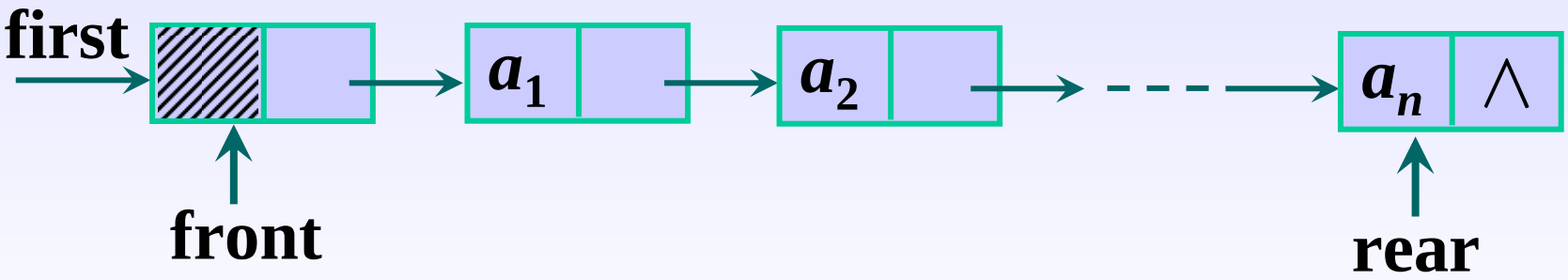


特殊线性表——队列

队列的链接存储结构及实现

队列的链接存储结构——链队列

如何改造单链表实现队列的链接存储？

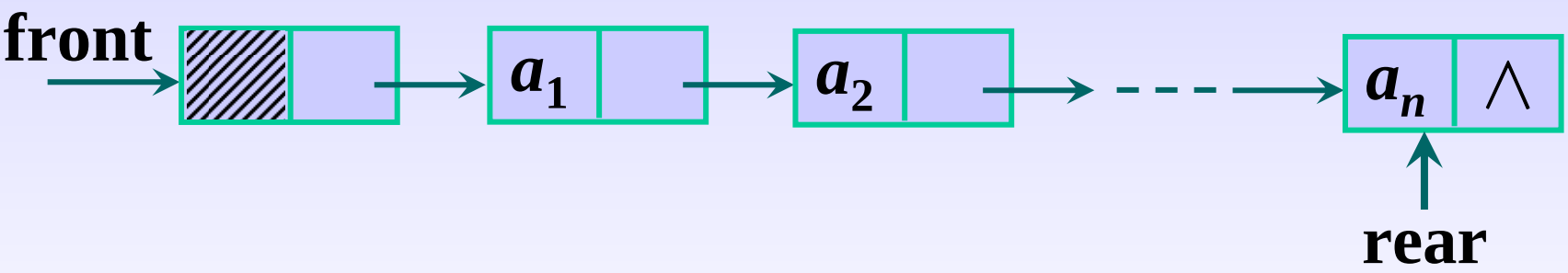


队头指针即为链表的头指针

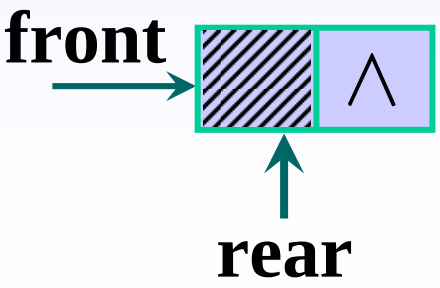
特殊线性表——队列

队列的链接存储结构及实现

非空链队列



空链队列



特殊线性表——队列

链队列类的声明

```
template <class T>
class LinkQueue
{
  public:
     LinkQueue( );
     ~LinkQueue( );
     void EnQueue(T x);
     T DeQueue( );
     T GetQueue( );
     int Empty( ){return(front==rear?1:0);}
  private:
     Node<T> *front, *rear;
};
```

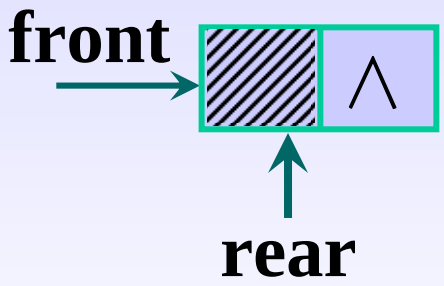
特殊线性表——队列

链队列的实现——构造函数

操作接口： **LinkQueue( );**

算法描述：

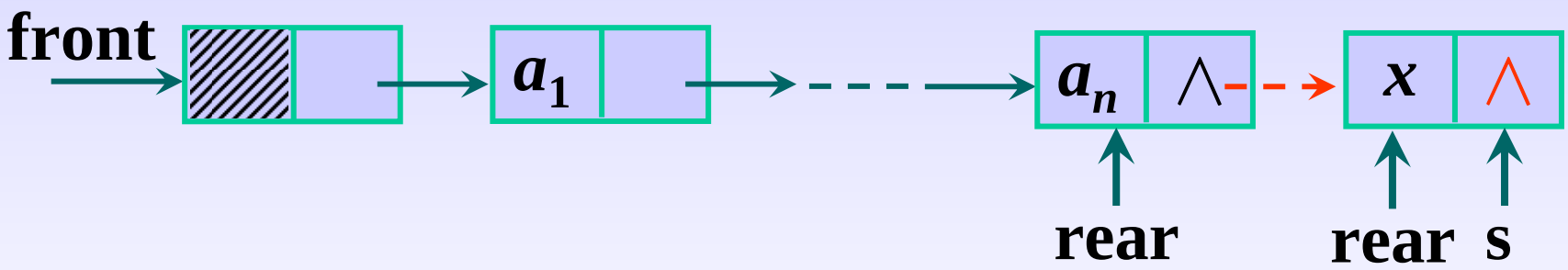
```
template <class T>
LinkQueue::LinkQueue( )
{
    front=new Node<T>;
    front->next=NULL;
    rear=front;
}
```

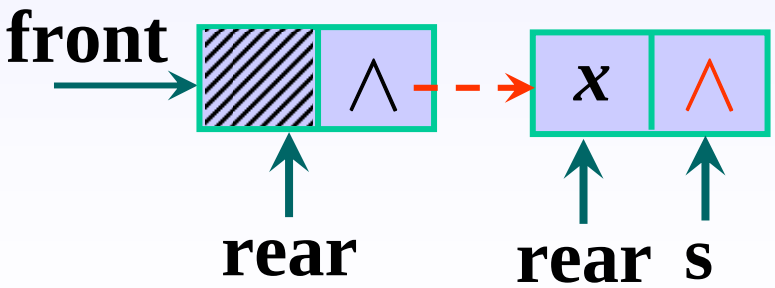


特殊线性表——队列

链队列的实现——入队

操作接口： **void EnQueue(T x);**





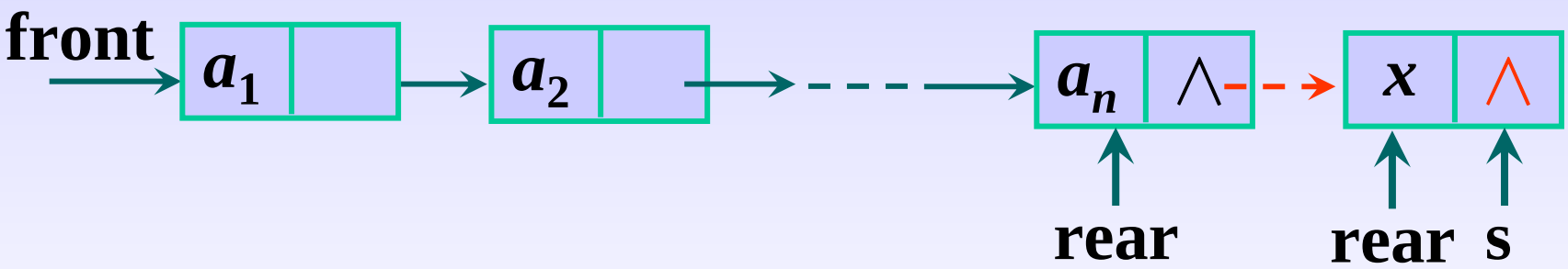
算法描述：

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

特殊线性表——队列

链队列的实现——入队

操作接口： **void EnQueue(T x);**



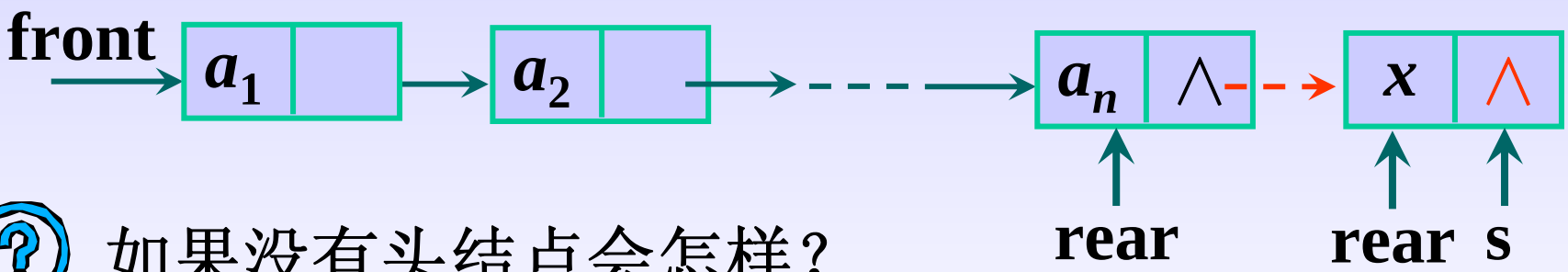
算法描述：

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

特殊线性表——队列

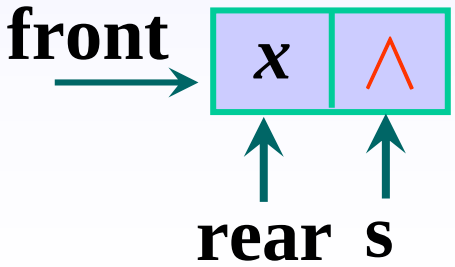
链队列的实现——入队

操作接口： **void EnQueue(T 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;

入队操作不统一

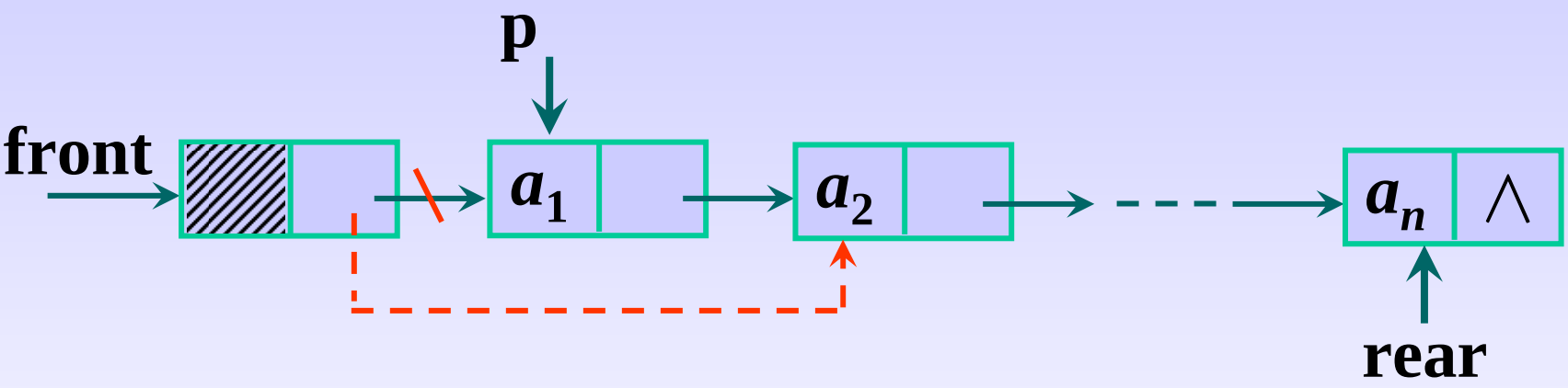
特殊线性表——队列

链队列的实现——入队

```
template <class T>
void LinkQueue::EnQueue(T x)
{
    s=new Node<T>;
    s->data=x;
    s->next=NULL;
    rear->next=s;
    rear=s;
}
```

特殊线性表——队列

链队列的实现——出队

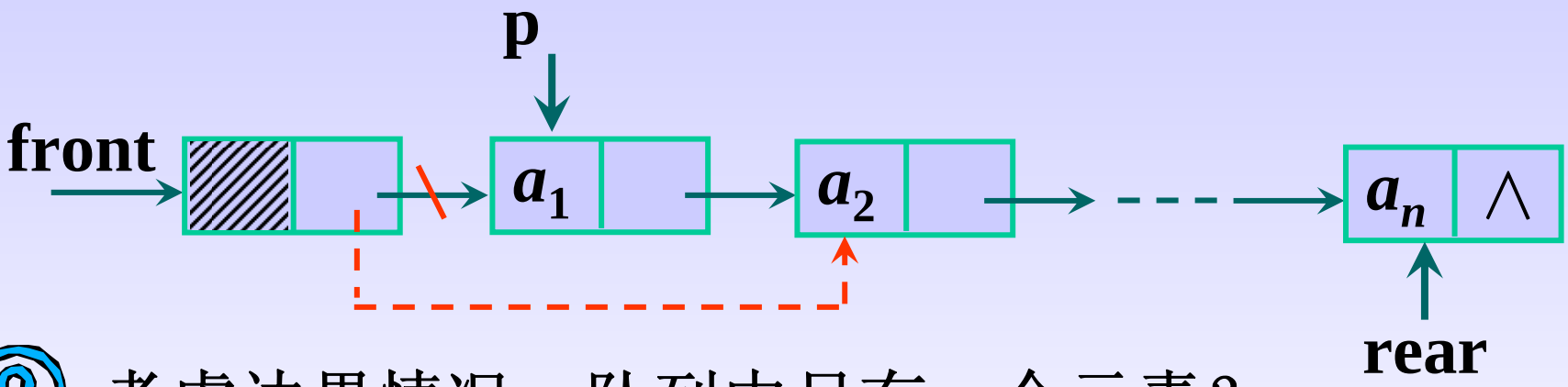


算法描述：

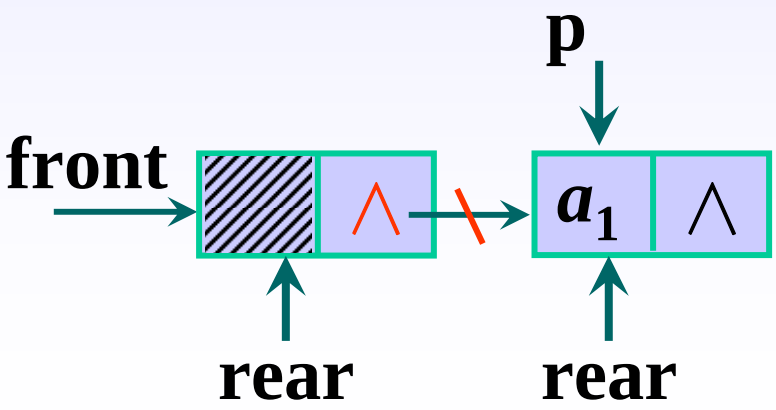
```
p=front->next;
front->next=p->next;
```

特殊线性表——队列

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;
```

特殊线性表——队列

链队列的实现——出队

```
template <class T>
T LinkQueue::DeQueue( )
{
    if (rear==front) throw "下溢";
    p=front->next;
    x=p->data;
    front->next=p->next;
    if (p->next==NULL) rear=front;
    delete p;
    return x;
}
```

特殊线性表——队列

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。

栈的应用举例

数制转换

括号匹配的检验

数制转换

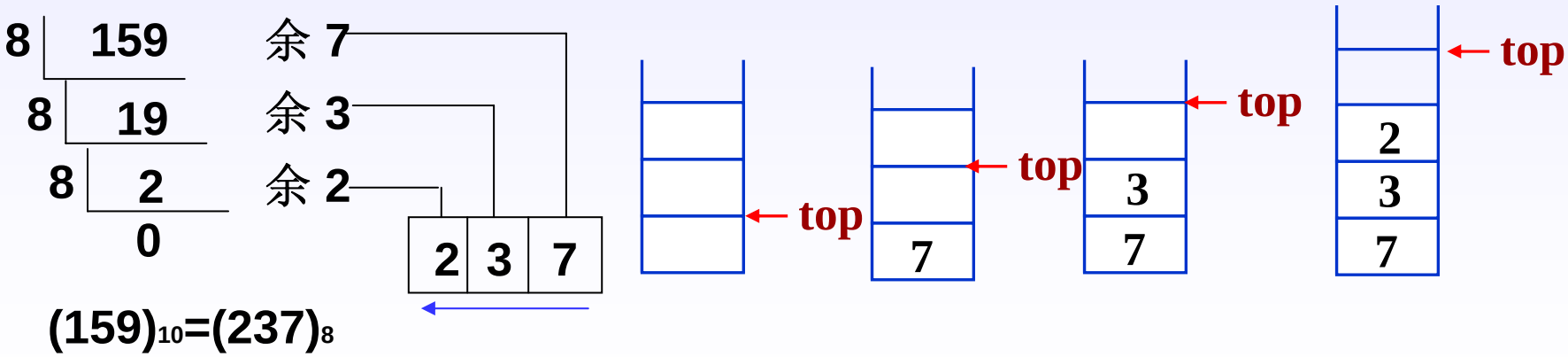
十进制N和其它d进制数的转换是计算机实现计算的基本问题。

基于下列原理:

$$N=(n \text{ div } d)*d+n \bmod d$$

(其中:div为整除运算,mod为求余运算)

例 把十进制数159转换成八进制数



$(159)_{10}=(237)_8$

数制转换

```
void conversion (int N,int d) {
   SeqStack<int> S;  // 构造空栈
         int e;
   while (N) {
     S.Push(N % d); // "余数"入栈
     N = N/d;//非零商继续运算
         }
   while (!S.Empty()) {  // 和"求余"所得相逆的顺序输出d进制的各位数
     e=S.Pop();
     cout<<e;
   }
} // conversion
```

括号匹配的检验

假设在表达式中

（[ ]（ ））或[（[ ][ ]）]

等为正确的格式，

[（]）或（[]（ ）或（（ ））均为不正确的格式

检验括号是否匹配的方法可用“期待的急迫程度”描述：

后出现的“左括号”，它等待与其匹配的“右括号”出现的“急迫”心情要比先出现的左括号高。

- 对“左括号”来说，后出现的比先出现的“优先”等待检验，
- 对"右括号"来说，每个出现的右括号要去找在它之前"最后"出现的那个左括号去匹配。

括号匹配的检验

例如：考虑下列括号序列：

[([ ] [ ])]

1 2 3 4 5 6 7 8

- 这个处理过程恰与栈的特点相吻合。
- 在算法中设置一个栈，每读入一个括号：
  - 若是右括号，则
    - 或者使置于栈顶的最急迫的期待得以消解，
    - 或者是不合法的情况；
  - 若是左括号，则
    - 作为一个新的更急迫的期待压入栈中，使原有的在栈中的所有未消解的期待的急迫性都降了一级。
- 在算法的 开始和结束时，栈都应该是空的

分析可能出现的不匹配的情况:

- 到来的右括号并非是所“期待”的;

  [( ])

- 到来的是“不速之客”;

  [( )]) 或)

- 直到结束，也没有到来所“期待”的括号。

  ((

这三种情况对应到栈的操作即为：

- 和栈顶的左括号不相匹配;
- 栈中并没有左括号等在哪里;
- 栈中还有左括号没有等到和它相匹配的右括号。

算法思想：

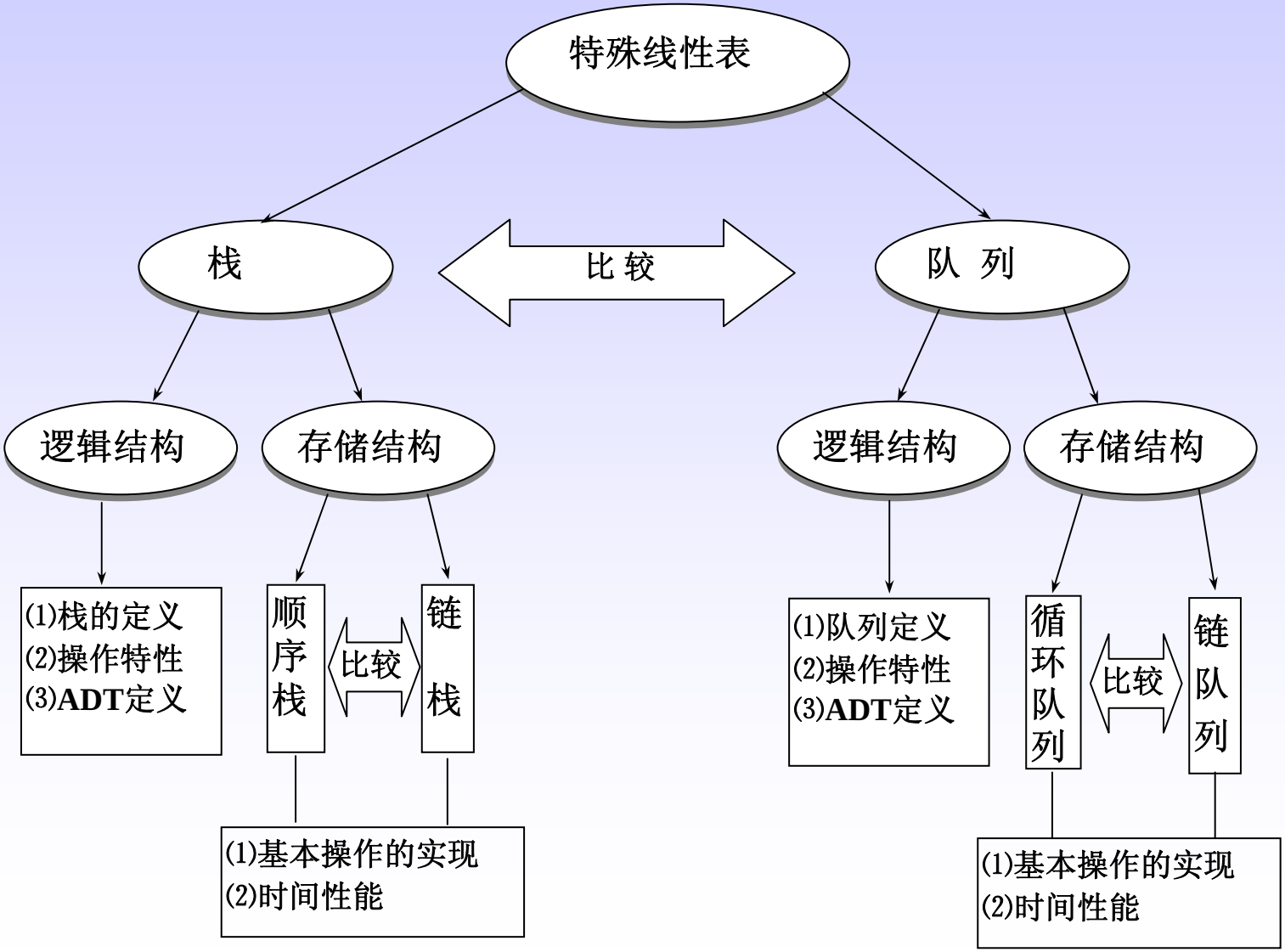
1）凡出现左括号，则进栈；

2）凡出现右括号，首先检查栈是否空
若栈空，则表明该“右括号”多余，
否则和栈顶元素比较，
若相匹配，则“左括号出栈”，
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“左括号”有余。

```
bool matching(char exp[]) {
  // 检验表达式中所含括号是否正确嵌套,
  //若是，则返回TRUE，否则返回FALSE. '#' 为表达式的结束符
  int state = 1,i=0;
  char ch,e;
  ch = exp[i++];
  SeqStack <char> S;  // 构造空栈
  while (ch!='\0' && state) {
     if(ch=='(' || ch=='[') S.Push(ch);  // 凡左括号一律入栈
     else if(ch== ')')
          if (!S.Empty() && S.GetTop()=='(') e=S.Pop(); else  state = 0;
          else if(ch==']')
          if (!S.Empty() && S.GetTop()=='[') e=S.Pop(); else  state = 0;
     ch = exp[i++];
    } // while
  if ( state && S.Empty() )  return 1;
  else  return 0;
}//matching
```

栈和队列


特殊线性表
栈
比较
队 列
逻辑结构
存储结构
逻辑结构
存储结构
(1)栈的定义
(2)操作特性
(3)ADT定义
顺序栈
比较
链栈
(1)基本操作的实现
(2)时间性能
(1)队列定义
(2)操作特性
(3)ADT定义
循环队列
比较
链队列
(1)基本操作的实现
(2)时间性能