Data Structure & Algorithm Analysis

Queue

Zibin Zheng（郑子彬）

School of Data and Computer Science，SYSU

http://www.inpluslab.com

课程主页：http://inpluslab.sysu.edu.cn/dsa2016/

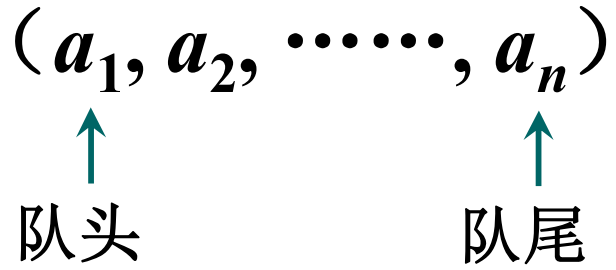
栈和队列

两种特殊的线性表——栈和队列

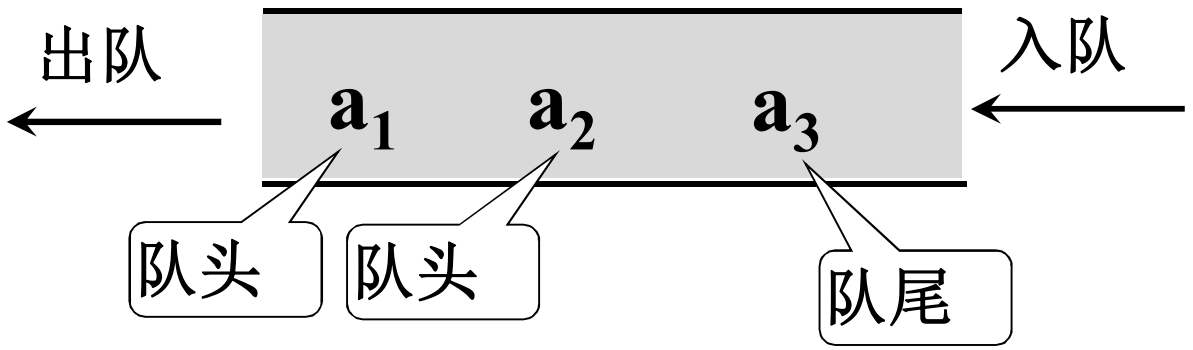
- 从数据结构角度看，栈和队列是操作受限的线性表，他们的逻辑结构相同。
- 从抽象数据类型角度看，栈和队列是两种重要的抽象数据类型。

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。
- 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。
- 空队列：不含任何数据元素的队列。



队列的逻辑结构



队列的操作特性： 先进先出

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

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

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

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

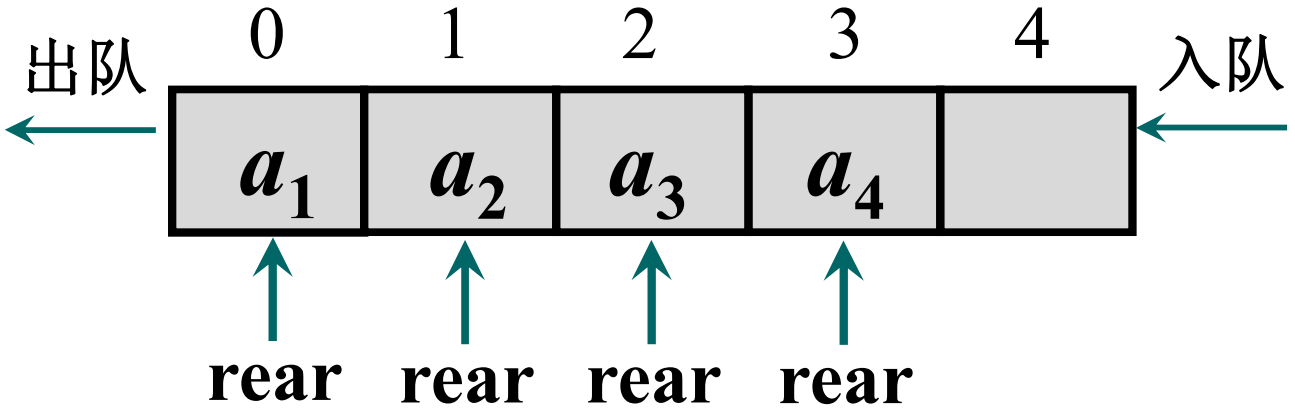
endADT

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

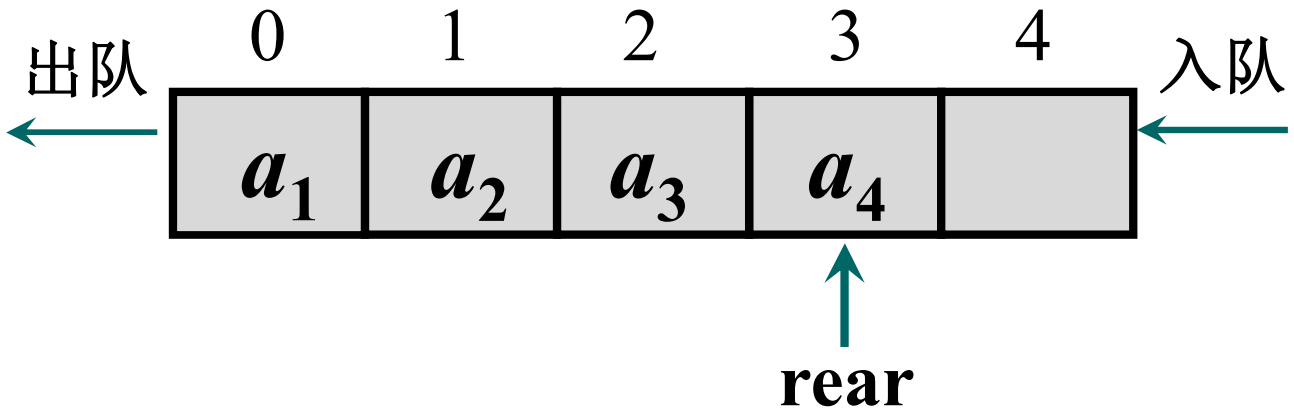


入队操作时间性能为$O(1)$

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？

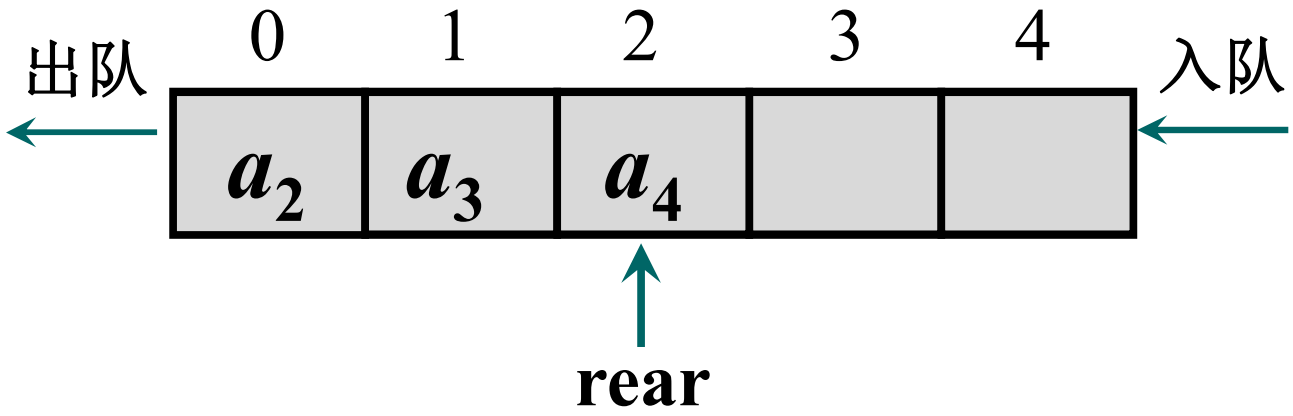
例：a_1a_2依次出队



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？

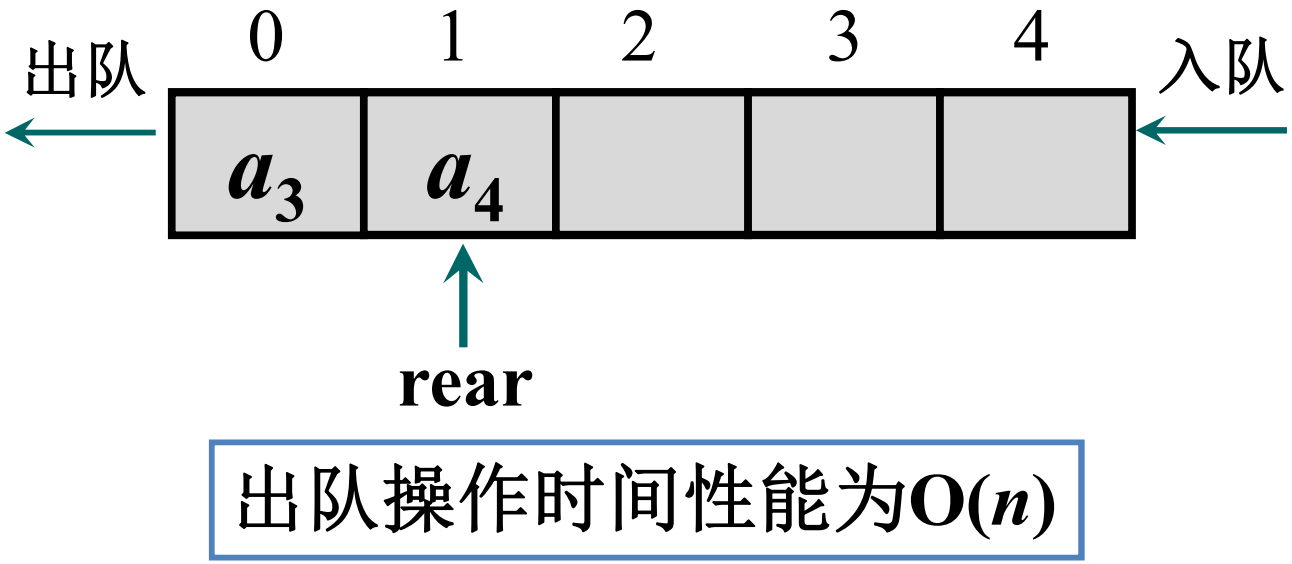
例：a_1a_2依次出队



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)

队列的顺序存储结构及实现

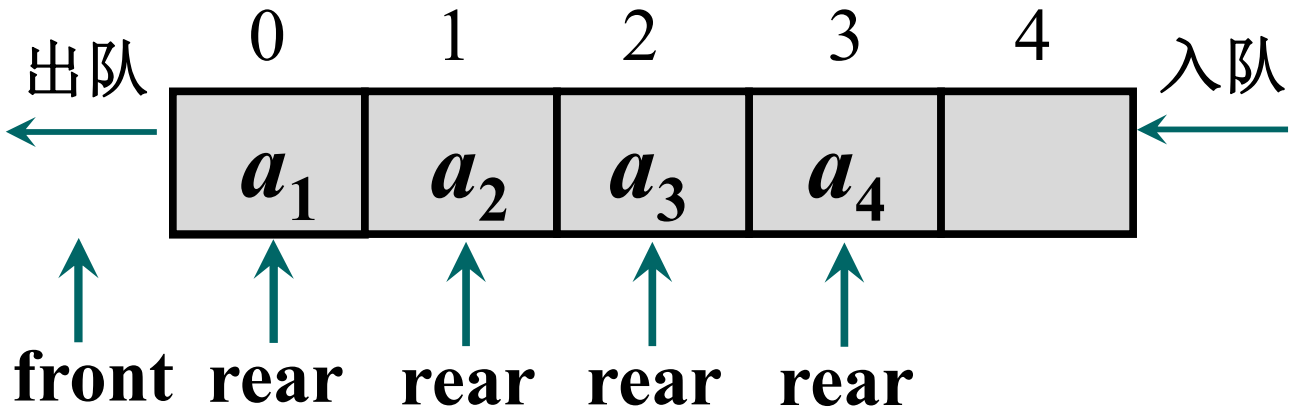
如何改进出队的时间性能？

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

队列的顺序存储结构及实现

例：$a_1a_2a_3a_4$依次入队

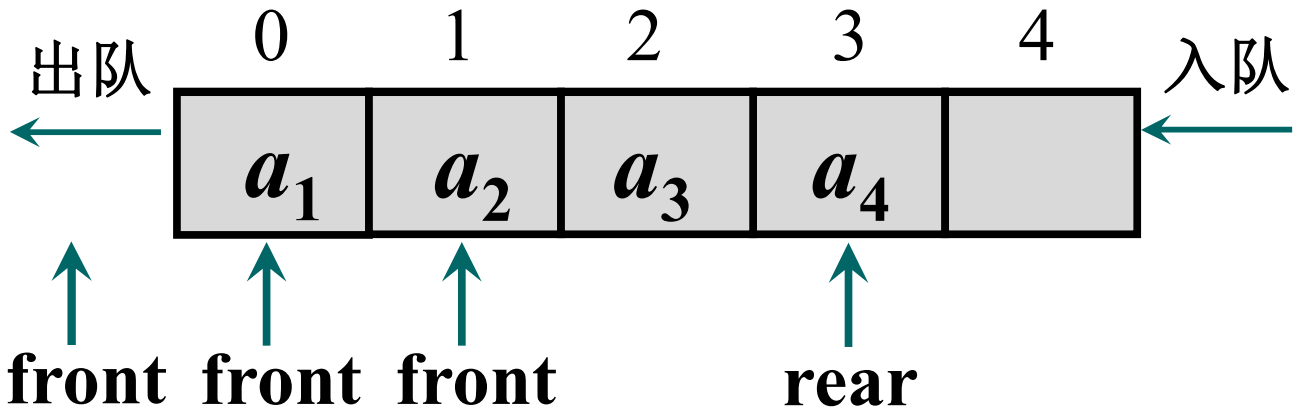


约定：队头指针front指向队头元素的前一个位置，
队尾指针rear指向队尾元素。

入队操作时间性能仍为$O(1)$

队列的顺序存储结构及实现

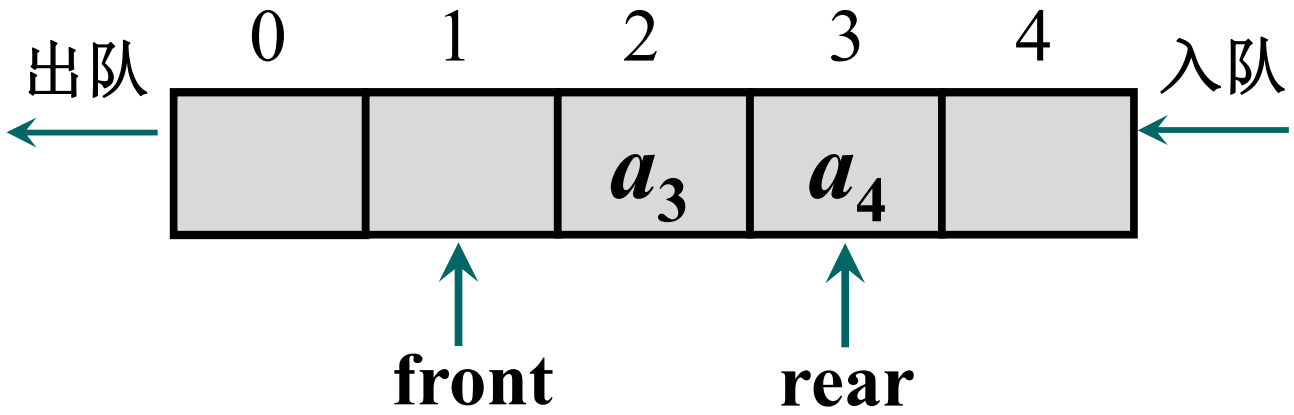
例：a_1a_2依次出队



出队操作时间性能提高为$O(1)$

队列的顺序存储结构及实现

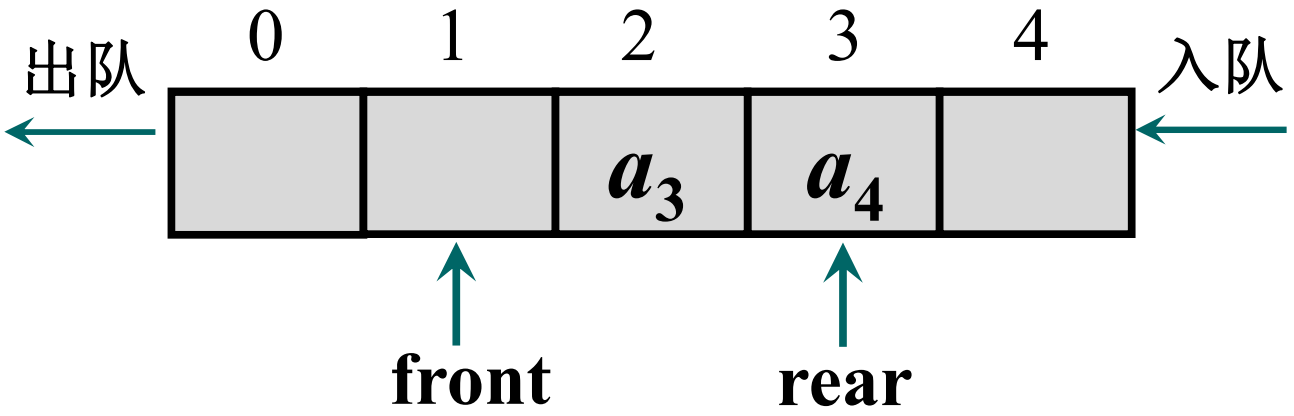
例：a_1a_2依次出队



队列的移动有什么特点？

队列的顺序存储结构及实现

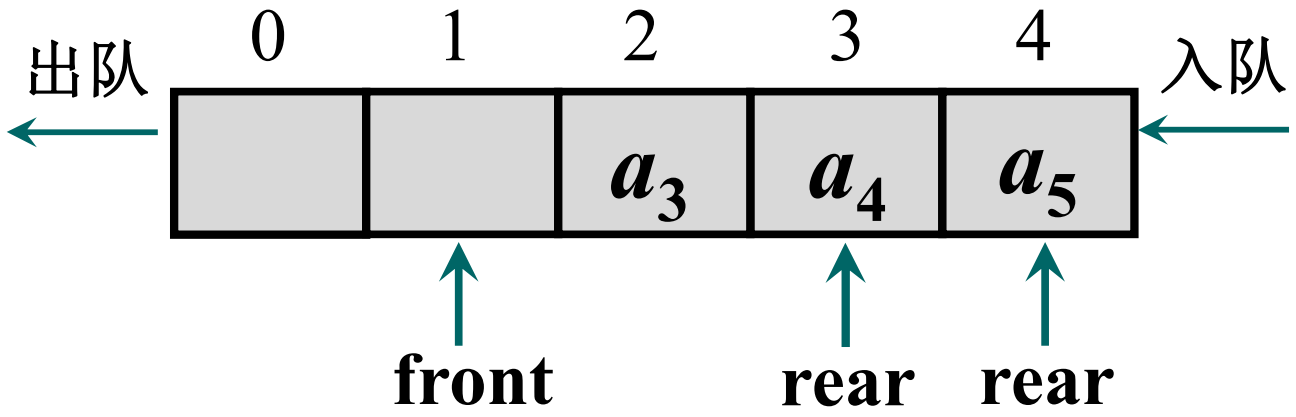
例：a_1a_2依次出队



整个队列向数组下标较大方向移动 ➡ 单向移动性

队列的顺序存储结构及实现

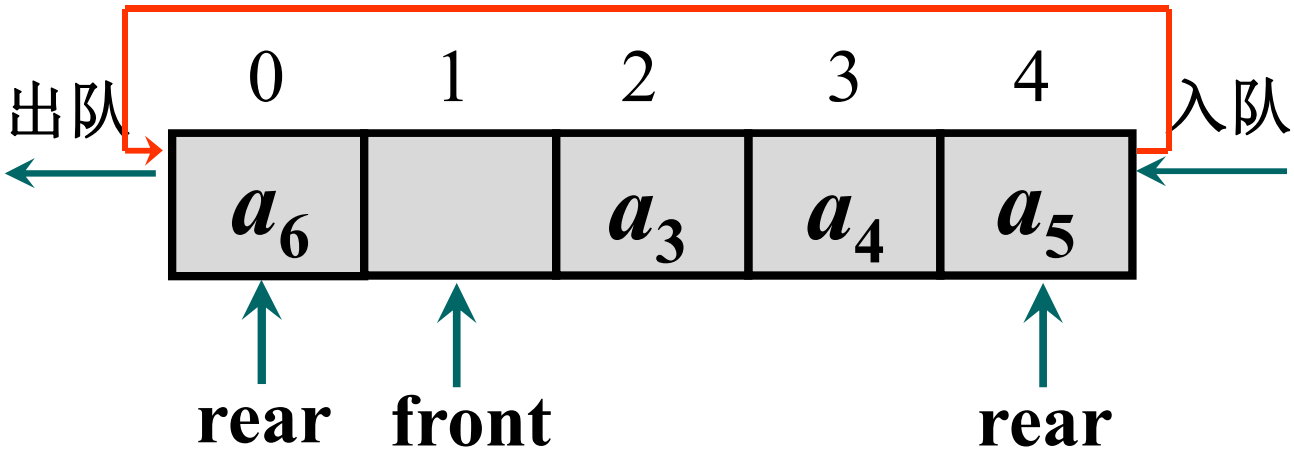
继续入队会出现什么情况？



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。

队列的顺序存储结构及实现

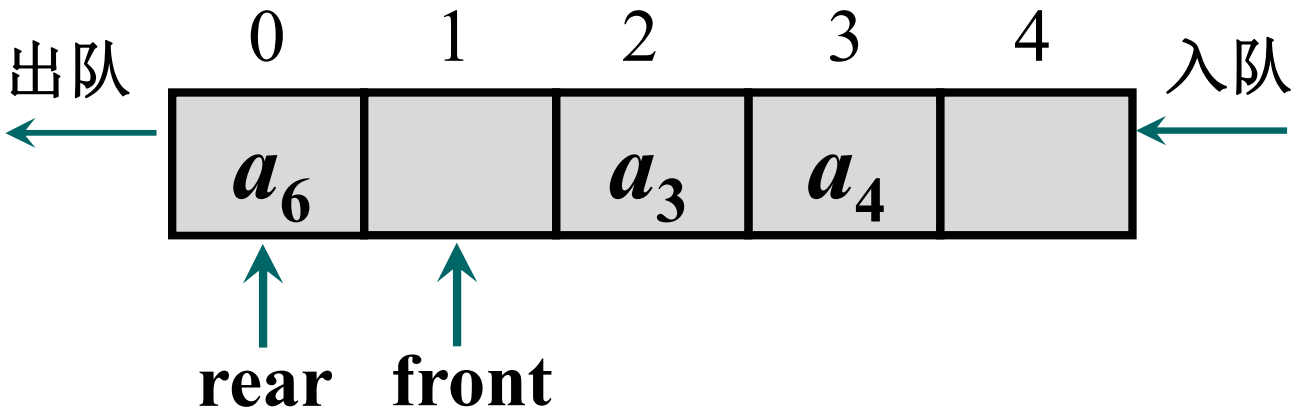
如何解决假溢出？



循环队列：将存储队列的数组头尾相接。

队列的顺序存储结构及实现

如何实现循环队列？



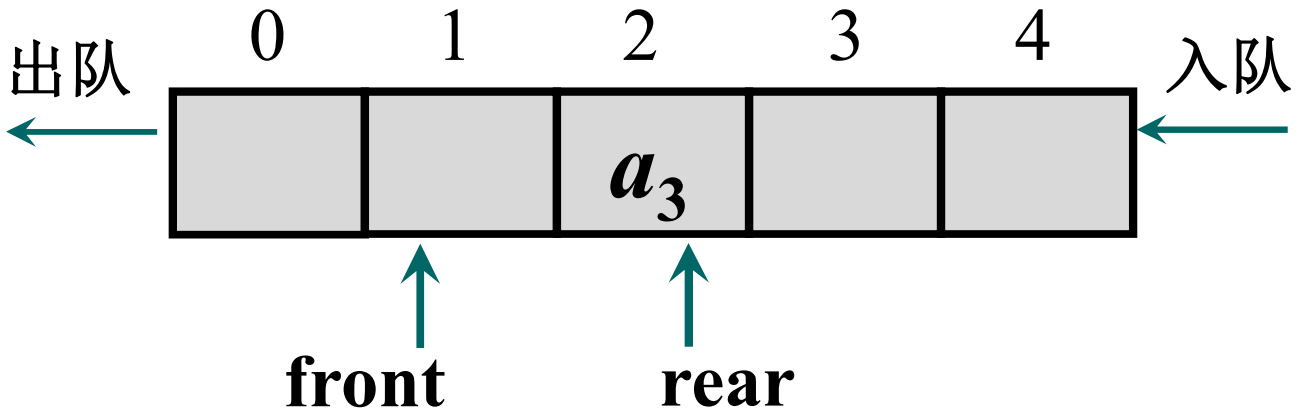
不存在物理的循环结构，用软件方法实现。

求模：（4＋1）mod 5＝0

队列的顺序存储结构及实现

如何判断循环队列队空？

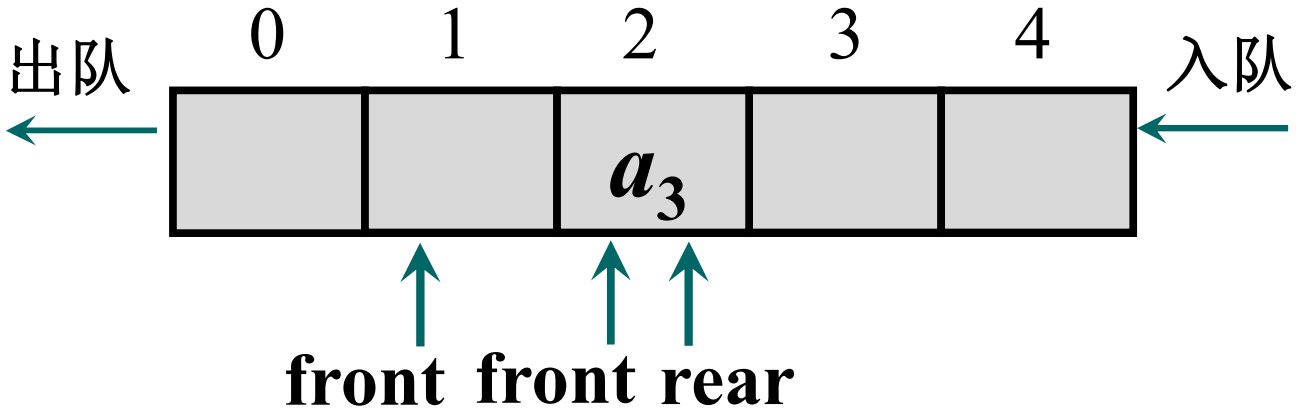
队空的临界状态



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

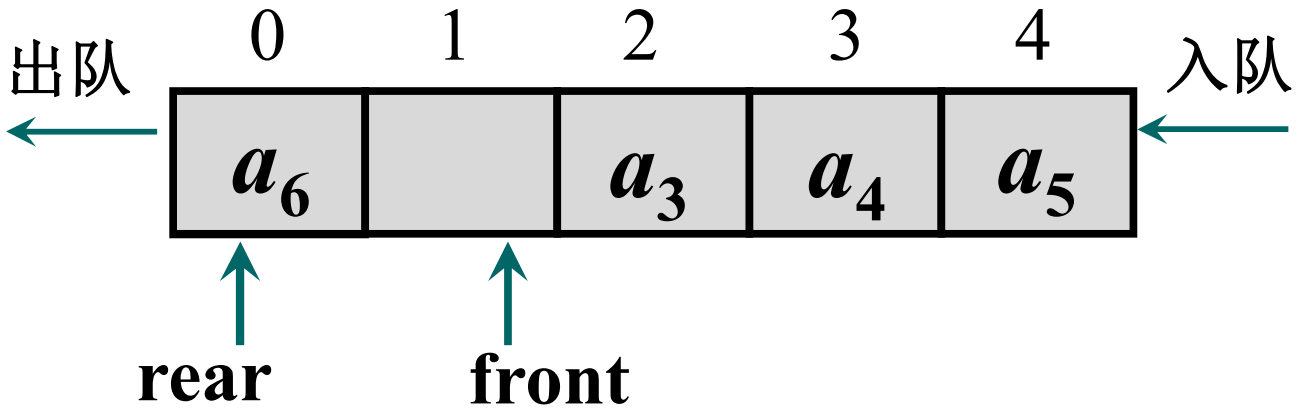


队空：front==rear

队列的顺序存储结构及实现

如何判断循环队列队满？

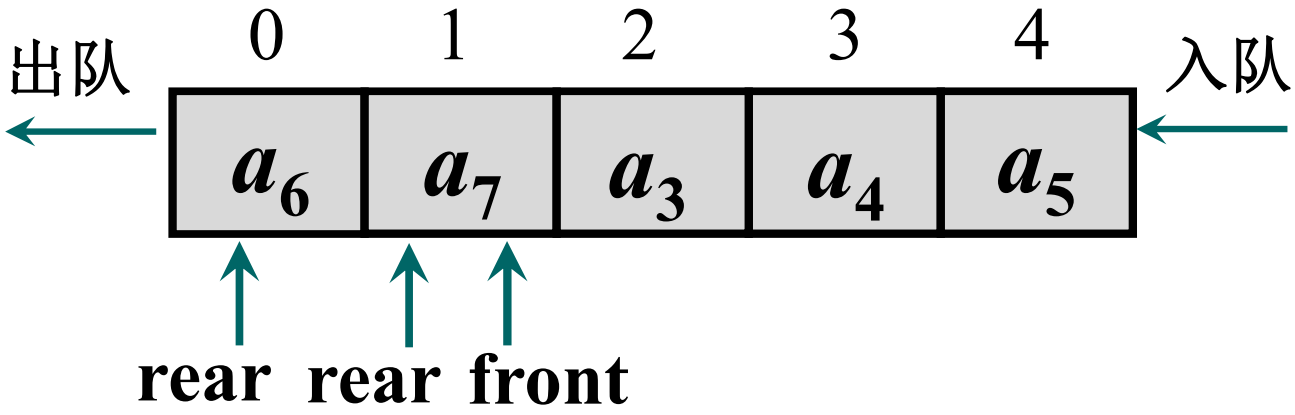
队满的临界状态



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

队列的顺序存储结构及实现

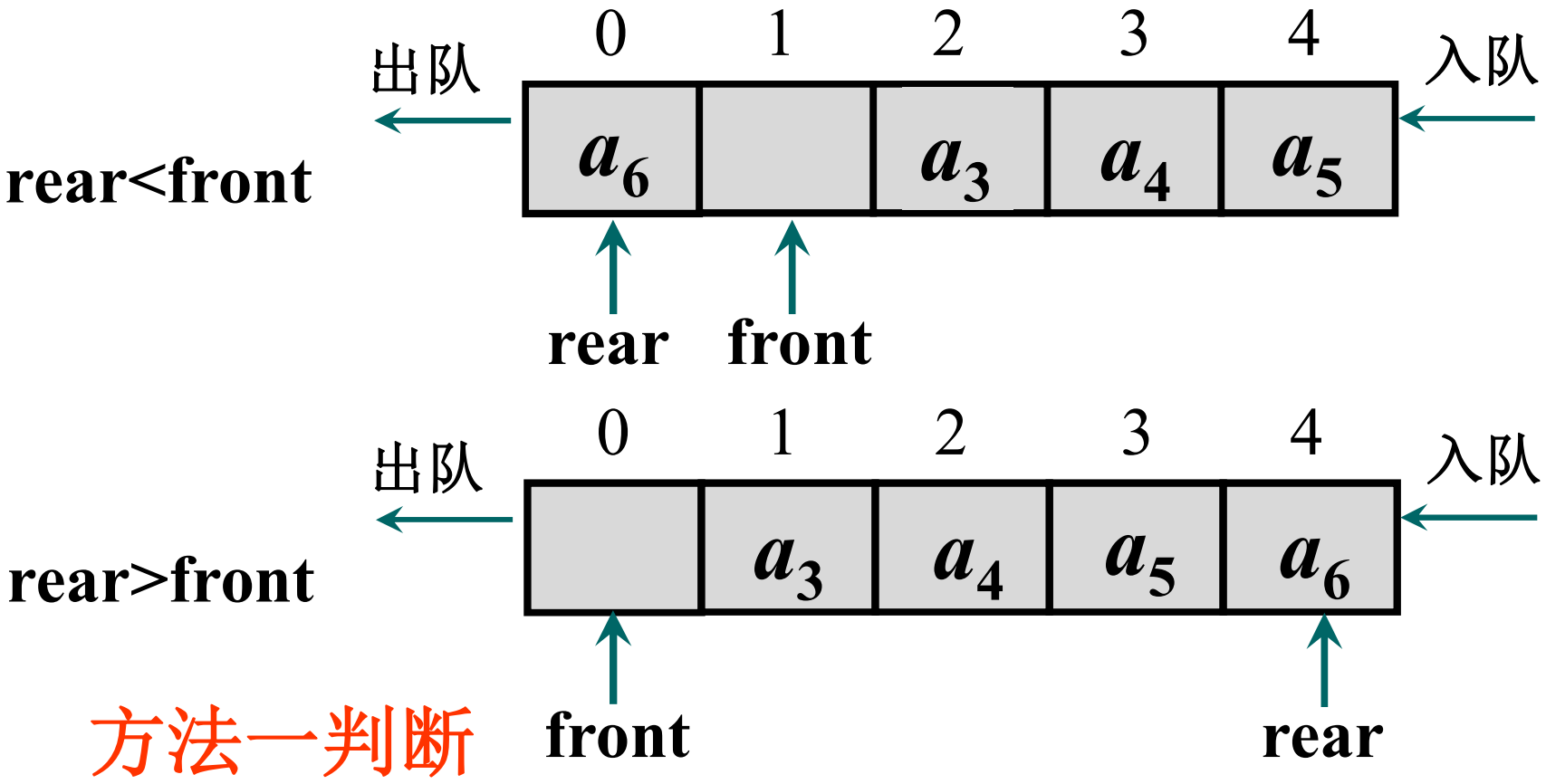
如何确定不同的队空、队满的判定条件？
为什么要将队空和队满的判定条件分开？

方法一：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：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。

队列的顺序存储结构及实现



方法一判断

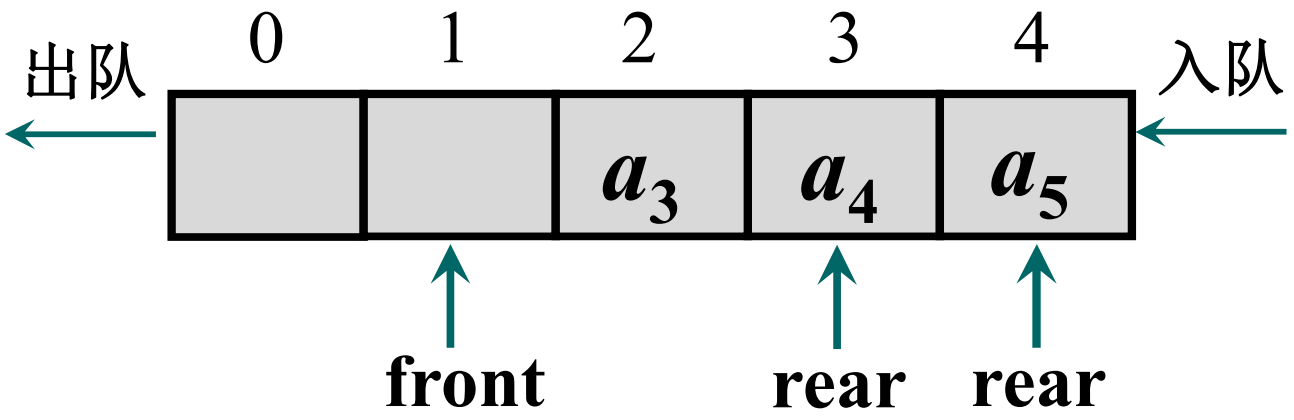
队满的条件：(rear+1) mod QueueSize=front

循环队列类的声明

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

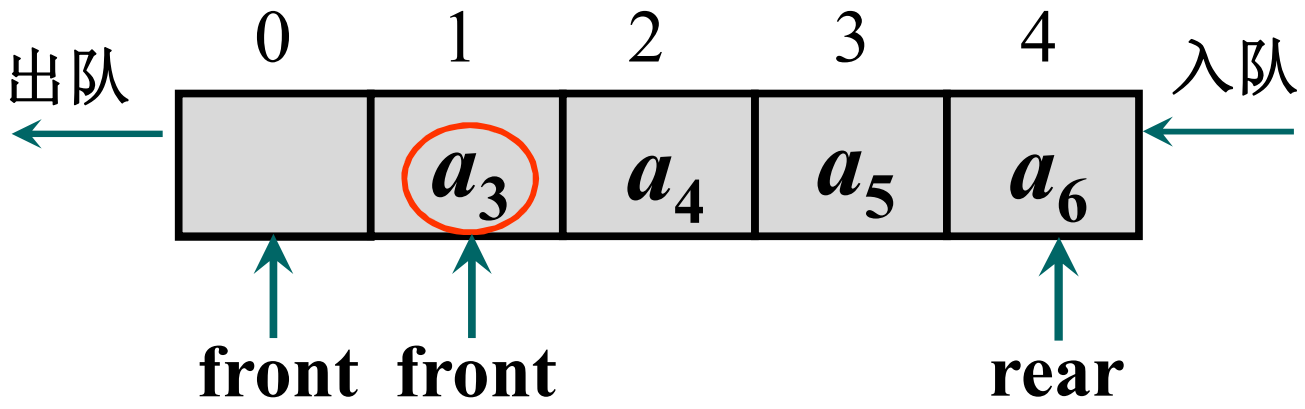
循环队列的实现——入队

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



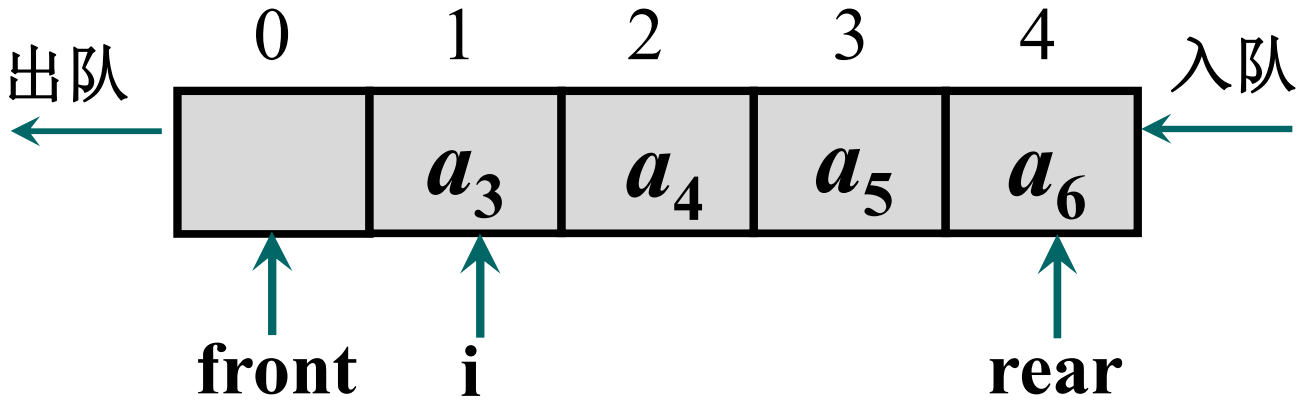
循环队列的实现——出队

```
template <class DataType>
DataType CirQueue<DataType> ::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

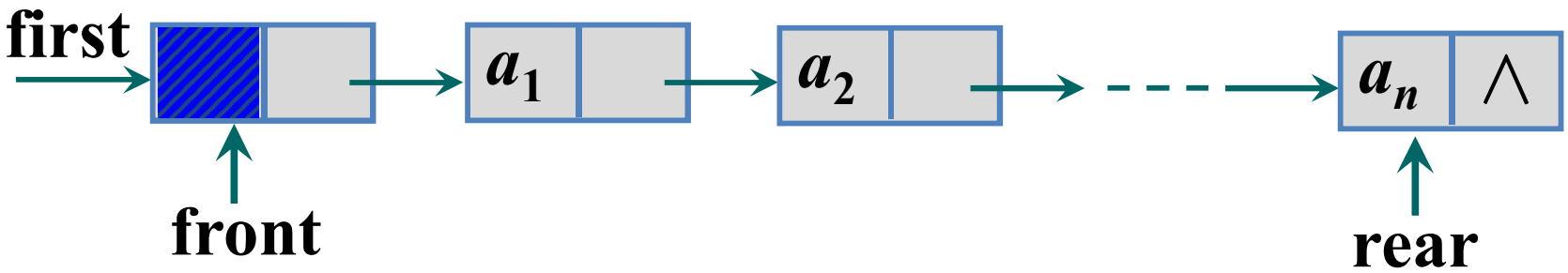
```
template <class DataType>
DataType CirQueue<DataType> ::GetQueue( )
{
    if (rear == front) throw "下溢";
    i = (front + 1) % QueueSize;
    return data[i];
}
```



队列的链接存储结构及实现

链队列： 队列的链接存储结构

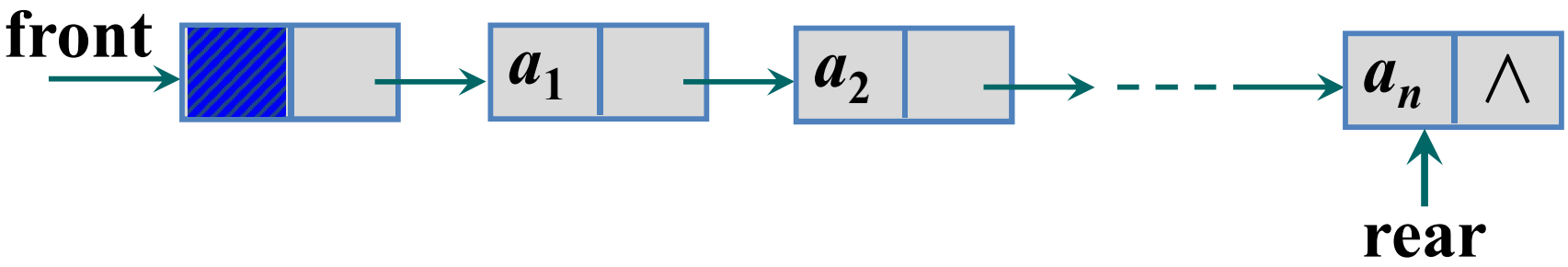
如何改造单链表实现队列的链接存储？



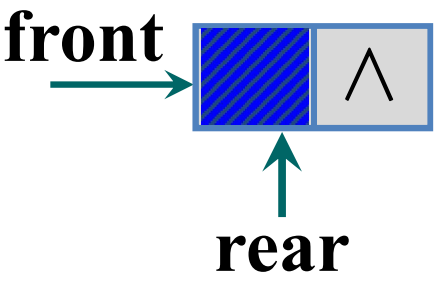
队头指针即为链表的头指针

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

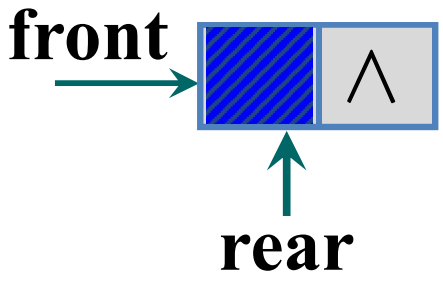
```
template <class DataType>
class LinkQueue
{
 public:
    LinkQueue( );
    ~LinkQueue( );
    void EnQueue(DataType x);
    DataType DeQueue( );
    DataType GetQueue( );
    bool Empty( );
 private:
    Node<DataType> *front, *rear;
};
```

链队列的实现——构造函数

操作接口： **LinkQueue( );**

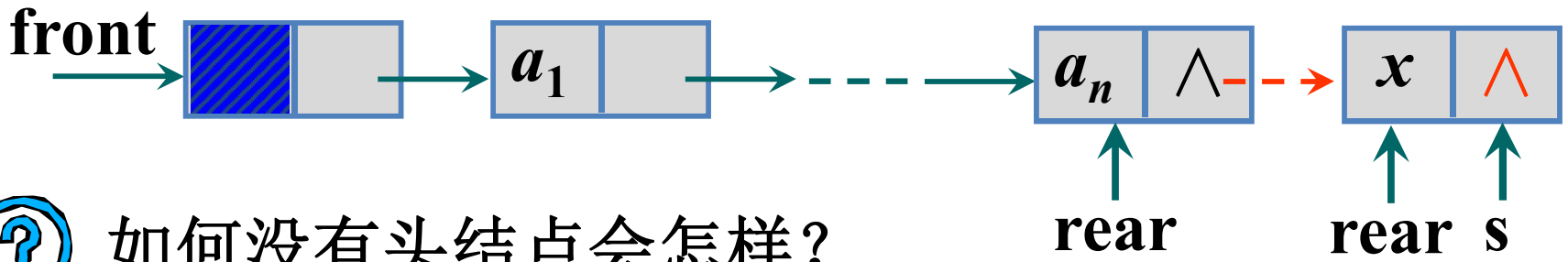
算法描述：

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

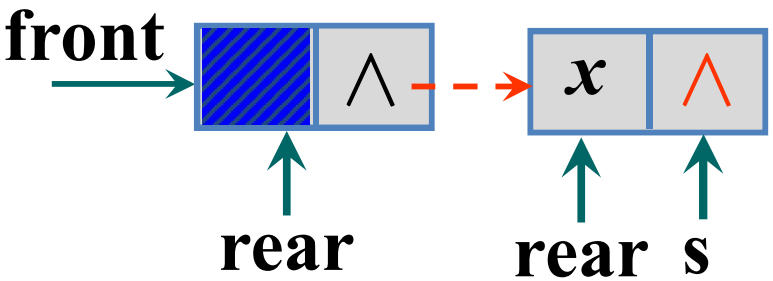


链队列的实现——入队

操作接口： void EnQueue(DataType x);



如何没有头结点会怎样？

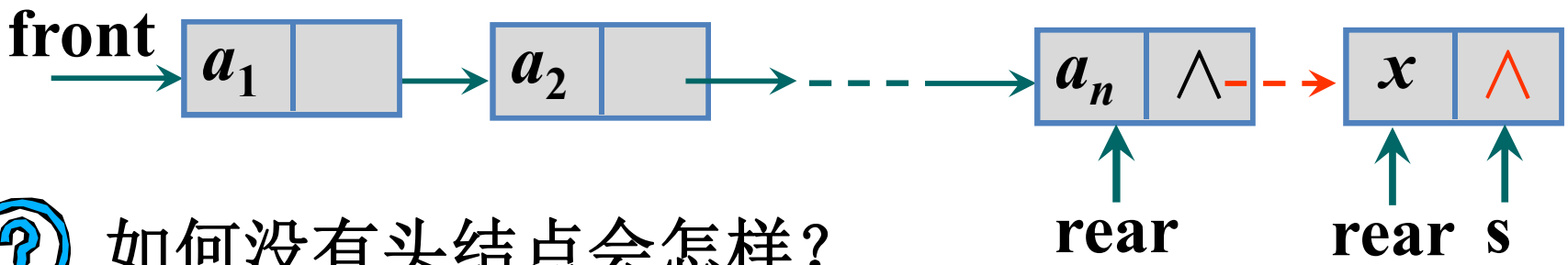


算法描述：

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

链队列的实现——入队

操作接口： void EnQueue(DataType x);



如何没有头结点会怎样？

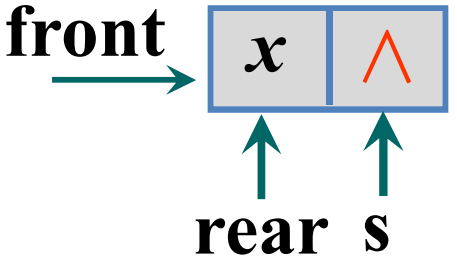
算法描述：

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

链队列的实现——入队

操作接口： **void EnQueue(DataType x);**

front=rear=NULL



如何没有头结点会怎样？

算法描述：

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

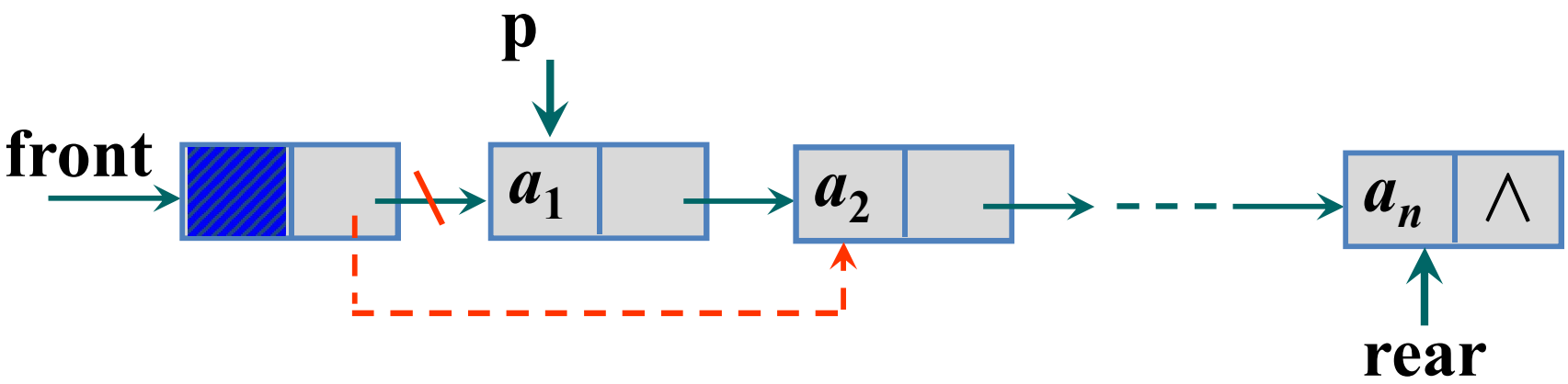
算法描述：

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

链队列的实现——入队

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

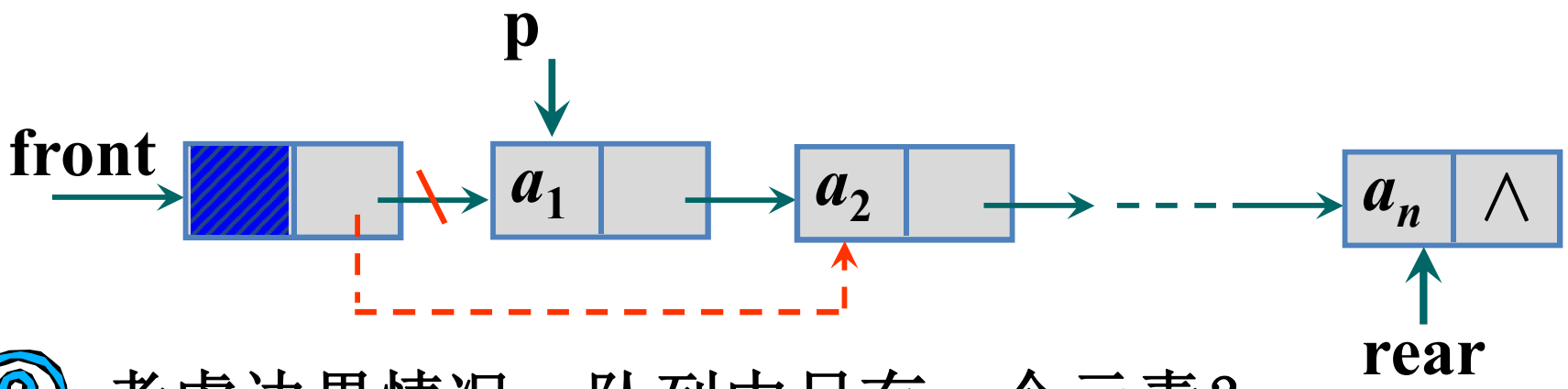
链队列的实现——出队



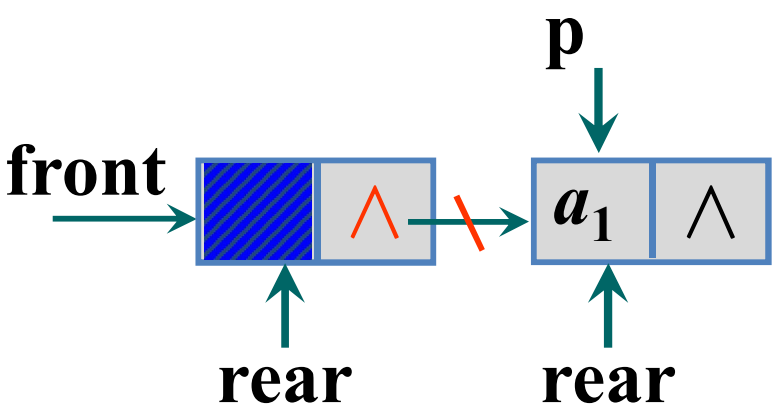
算法描述：

```
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

```
template <class DataType>
DataType LinkQueue<DataType> ::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;
}
```

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：

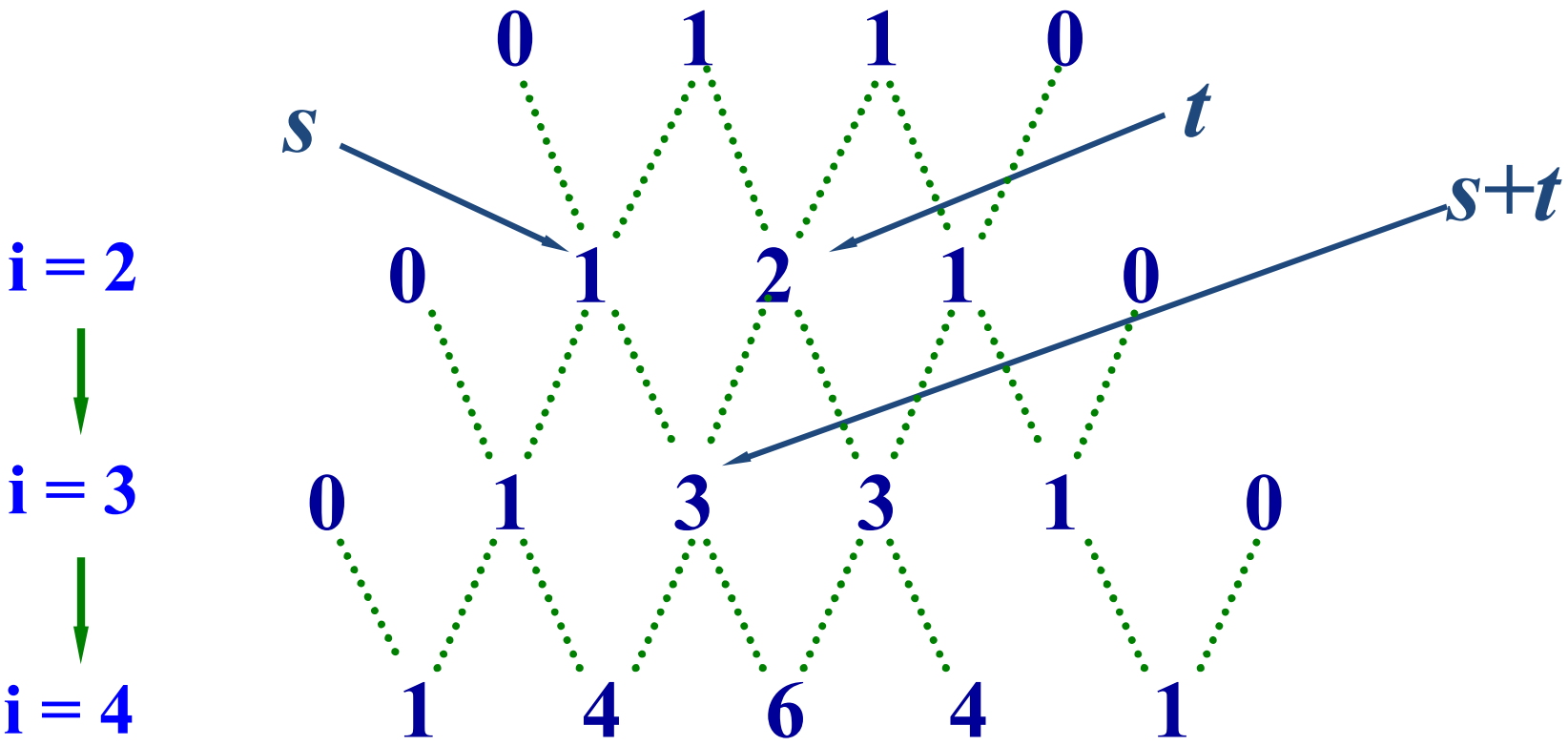
- 循环队列：必须预先确定一个固定的长度，所以有存储元素个数的限制和空间浪费的问题。
- 链队列：没有队列满的问题，只有当内存没有可用空间时才会出现队列满，但是每个元素都需要一个指针域，从而产生了结构性开销。

队列的应用：打印杨辉三角形

- 算法逐行打印二项展开式 $(a+b)^i$ 的系数：杨辉三角形 **(Pascal's triangle)**

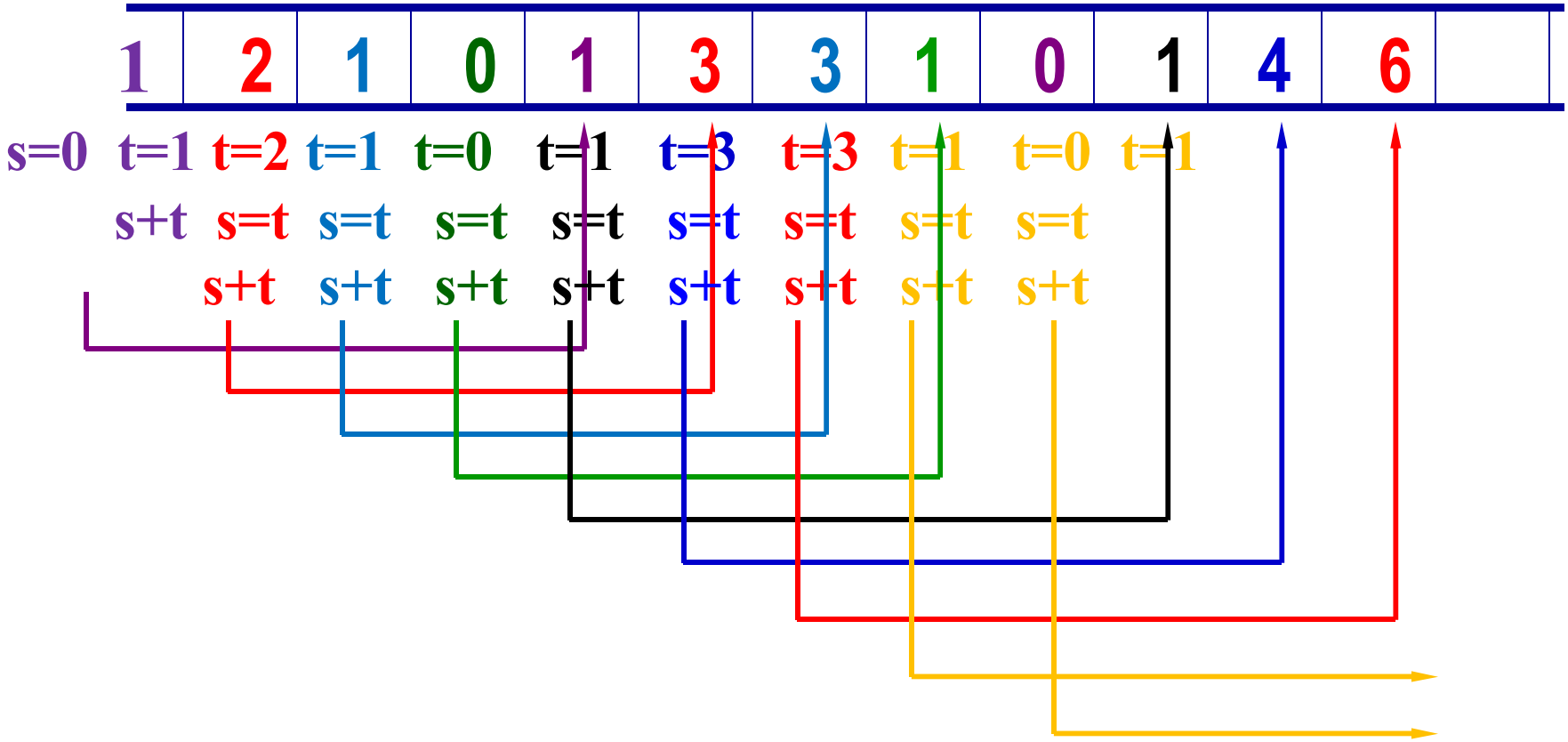
				1		1						$i=1$
			1		2		1					2
		1		3		3		1				3
	1		4		6		4		1			4
1		5		10		10		5		1		5
	1		6		15		20		15		6		1		6

分析第i行元素与第i+1行元素的关系



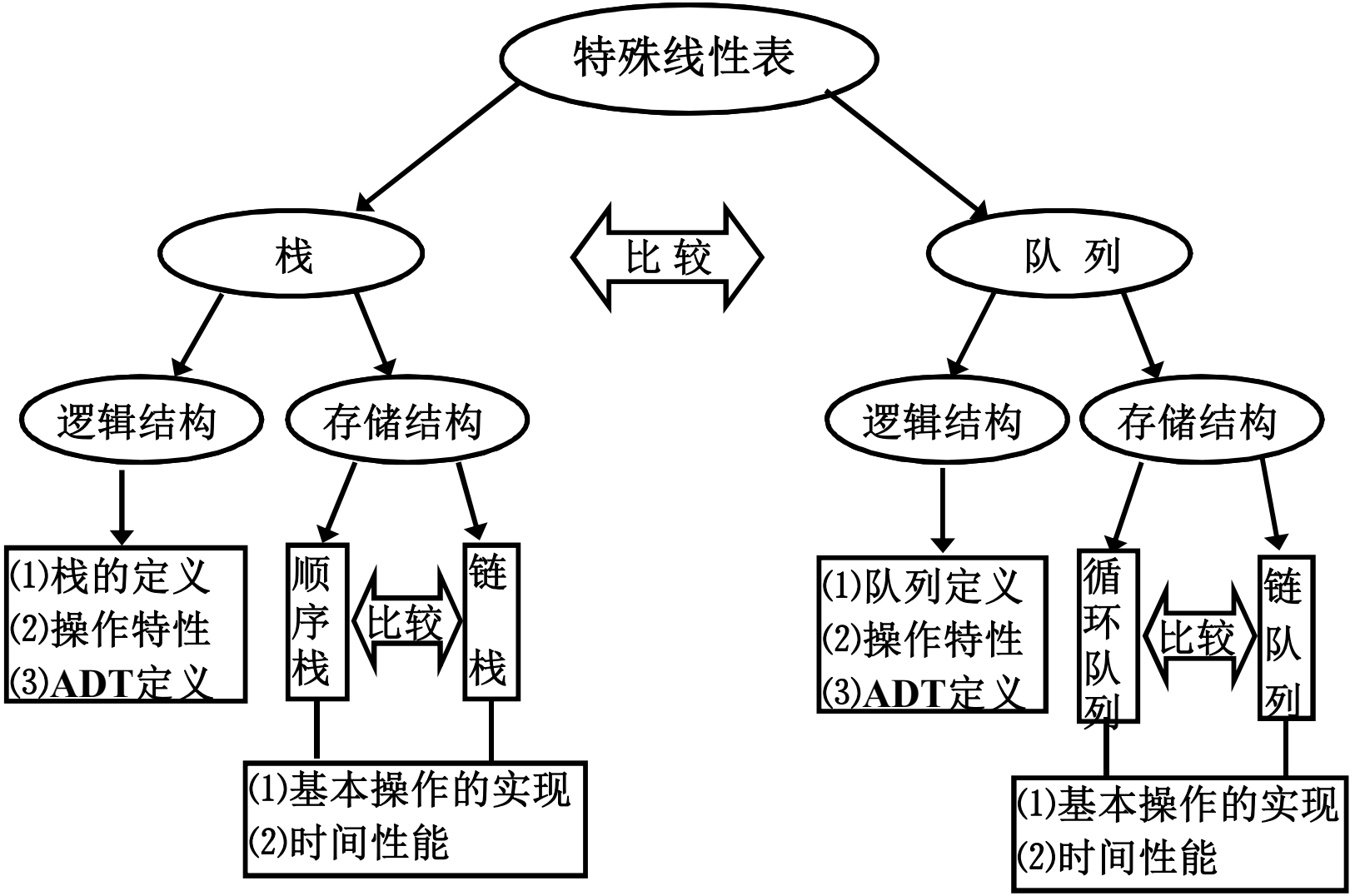
从前一行的数据可以计算下一行的数据

从第i行数据计算并存放第i+1行数据



代码参考：http://blog.csdn.net/cainv89/article/details/51526295

本章总结


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)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
(1)基本操作的实现
(2)时间性能

栈的应用：递归

- **递归的定义**

  若一个对象部分地包含它自己，或用它自己给自己定义，则称这个对象是递归的；若一个过程直接地或间接地调用自己，则称这个过程是递归的过程

- 以下三种情况常常用到递归方法
  - 定义是递归的
  - 数据结构是递归的
  - 问题的解法是递归的

定义是递归的

例如，阶乘函数

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

求解阶乘函数的递归算法

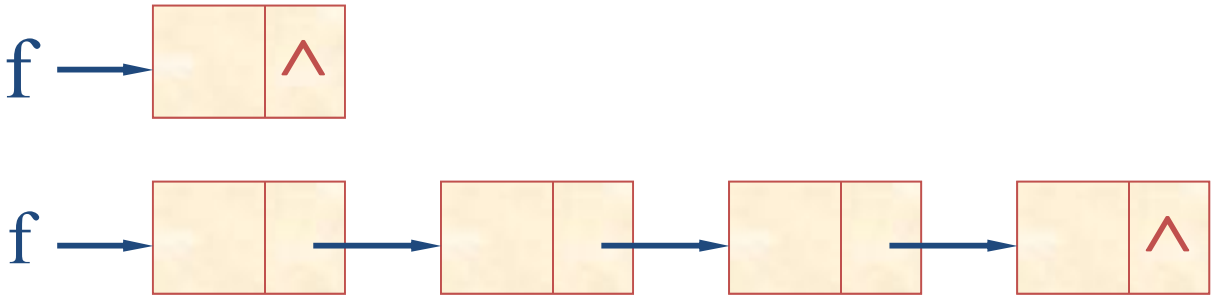
```
long Factorial(long n) {
    if (n == 0) return 1;
else return n*Factorial(n−1);
}
```

求解阶乘n!的过程


主程序 main : fact(4)
参数 4　计算 4*fact(3)　返回 24
参数 3　计算 3*fact(2)　返回 6
参数 2　计算 2*fact(1)　返回 2
参数 1　计算 1*fact(0)　返回 1
参数 0　直接定值 = 1　返回 1
递归调用
参数传递
结果返回
回归求值

数据结构是递归的

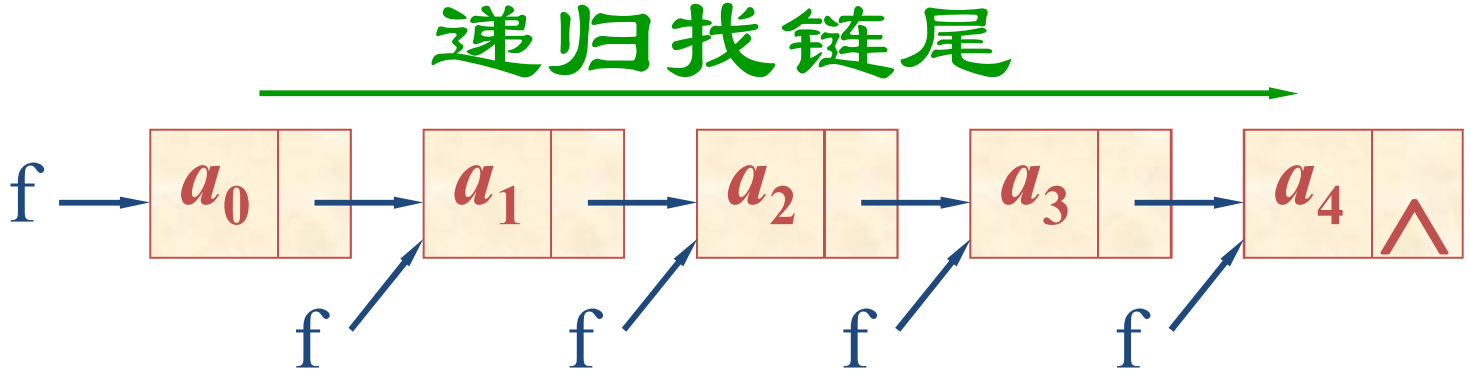
- 例如，单链表结构



- 一个结点，它的指针域为NULL，是一个单链表
- 一个结点，它的指针域指向单链表，仍是一个单链表

搜索链表的最后一个节点并打印其数值

```
template <class E>
void Print(ListNode<E> *f) {
    if (f->next == NULL)
        cout << f->data << endl;
    else Print(f->next);
}
```



问题的解法是递归的

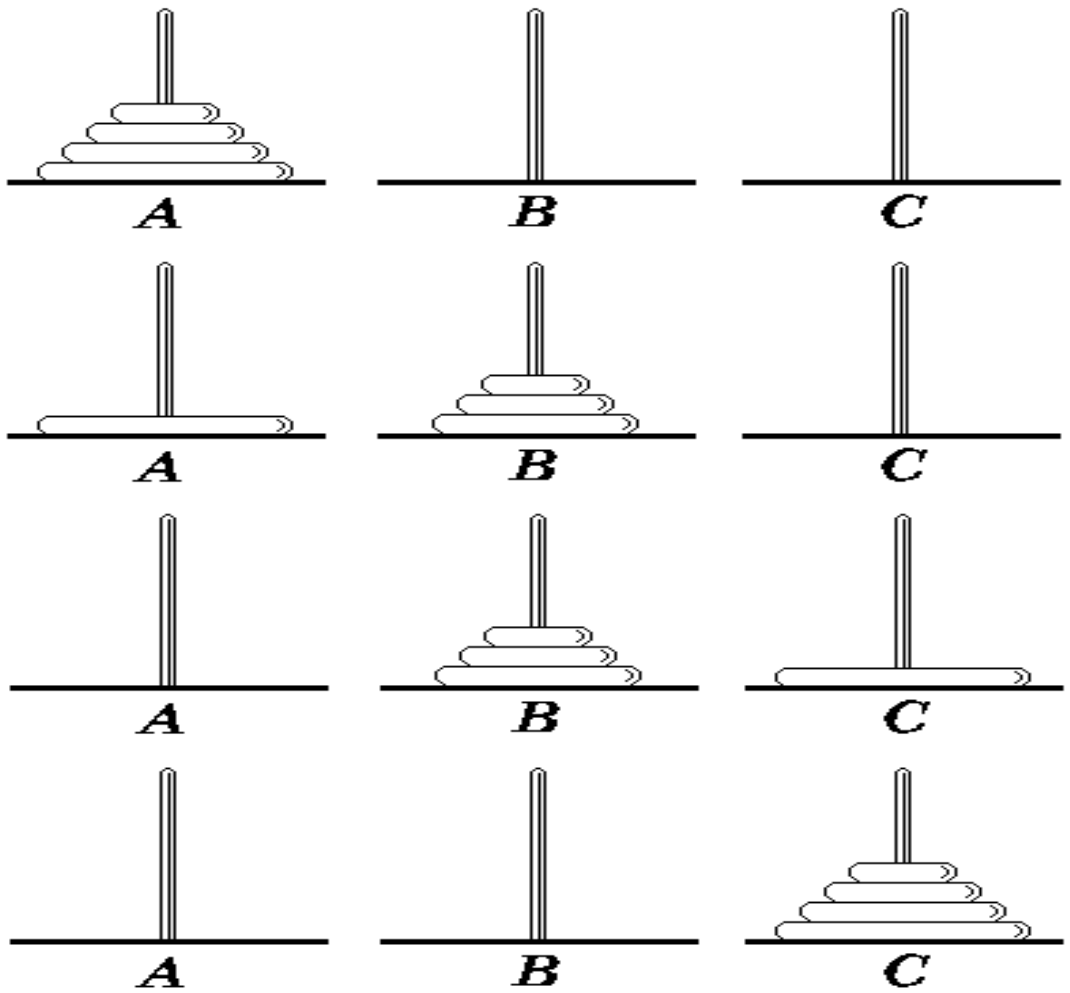
- 例，汉诺塔(Tower of Hanoi)问题的解法：

  如果 n = 1，则将这一个盘子直接从 A 柱移到 C 柱上。否则，执行以下三步：

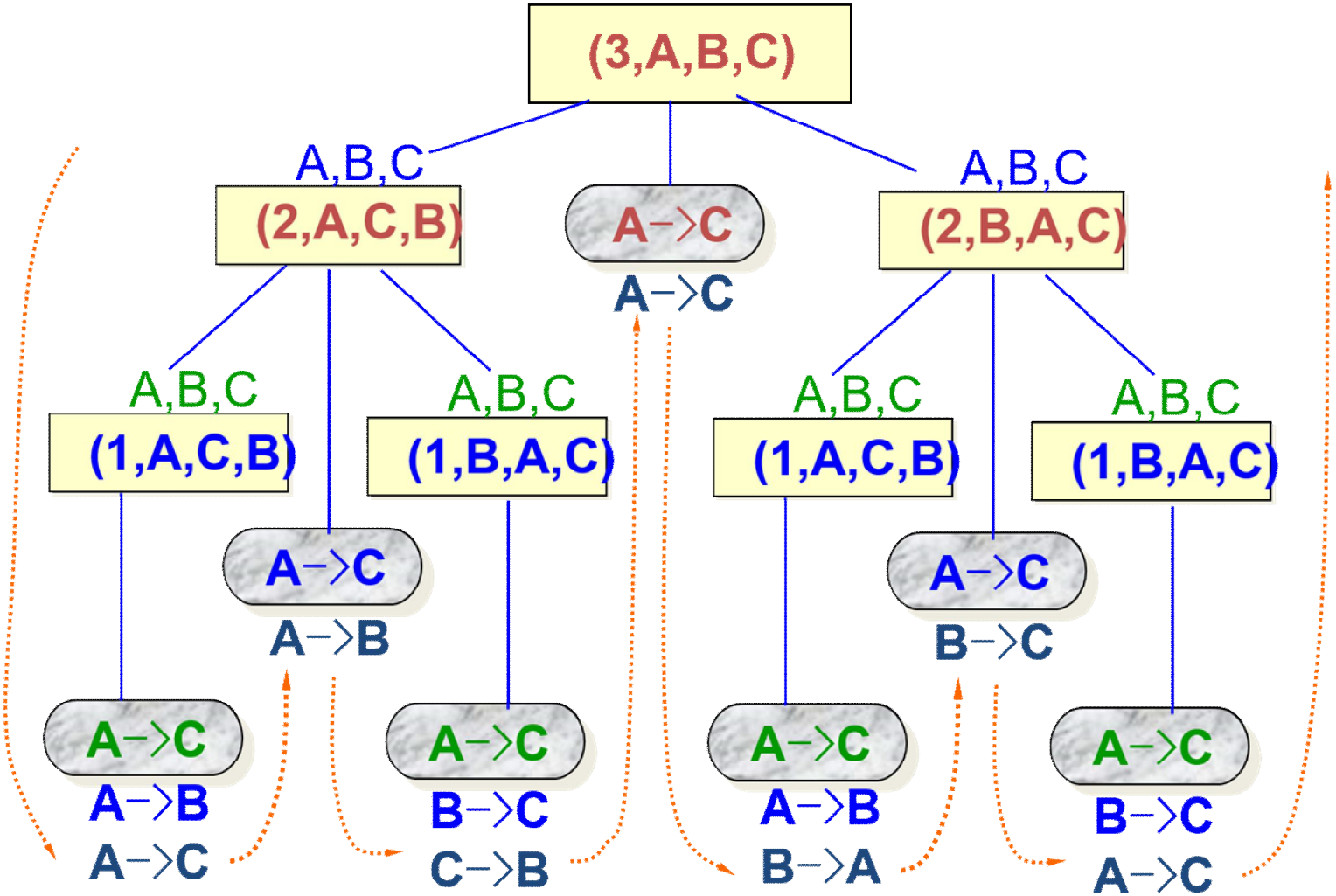
  ① 用 C 柱做过渡，将 A 柱上的 (n-1) 个盘子移到 B 柱上：

  ② 将 A 柱上最后一个盘子直接移到 C 柱上；

  ③ 用 A 柱做过渡，将 B 柱上的 (n-1) 个盘子移到 C 柱上。


A
B
C
A
B
C
A
B
C
A
B
C

```
#include <iostream.h>
void Hanoi (int n, char A, char B, char C) {
//解决汉诺塔问题的算法
    if (n == 1) 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);
          }
}
```


(3,A,B,C)
A,B,C
(2,A,C,B)
A->C
A->C
A,B,C
(2,B,A,C)
A,B,C
(1,A,C,B)
A,B,C
(1,B,A,C)
A->C
A->B
A,B,C
(1,A,C,B)
A,B,C
(1,B,A,C)
A->C
B->C
A->C
A->B
A->C
A->C
B->C
C->B
A->C
A->B
B->A
A->C
B->C
A->C

自顶向下、逐步分解的策略

- 子问题应与原问题做同样的事情，且更为简单
- 解决递归问题的策略是把一个规模比较大的问题分解为一个或若干规模比较小的问题，分别对这些比较小的问题求解，再综合它们的结果，从而得到原问题的解

  — 分而治之策略（分治法）
- 这些比较小的问题的求解方法与原来问题的求解方法一样

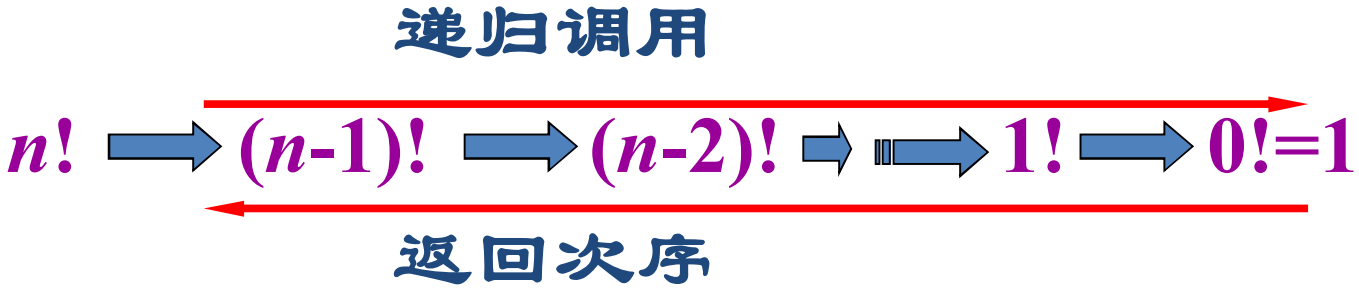
构成递归的条件

- 不能无限制地调用本身，必须有一个出口，化简为非递归状况直接处理

```
Procedure <name> ( <parameter list> ) {
  if ( < initial condition> )    //递归结束条件
    return ( initial value );
  else                           //递归
    return (<name> ( parameter exchange ));
}
```

递归过程及递归工作栈

- 递归过程在实现时，需要自己调用自己
- 层层向下递归，退出时的次序正好相反：

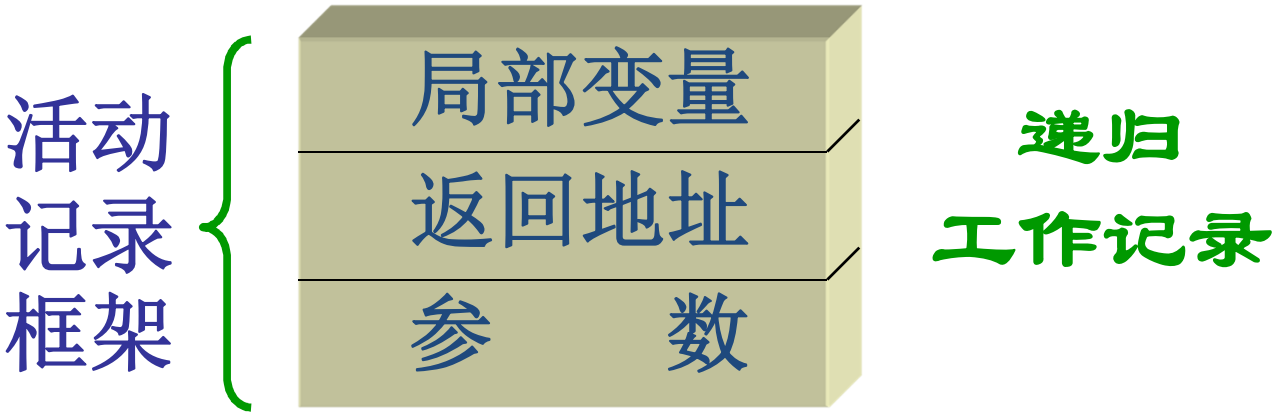


- 主程序第一次调用递归过程为外部调用
- 递归过程每次递归调用自己为内部调用
- 它们返回调用它的过程的地址不同

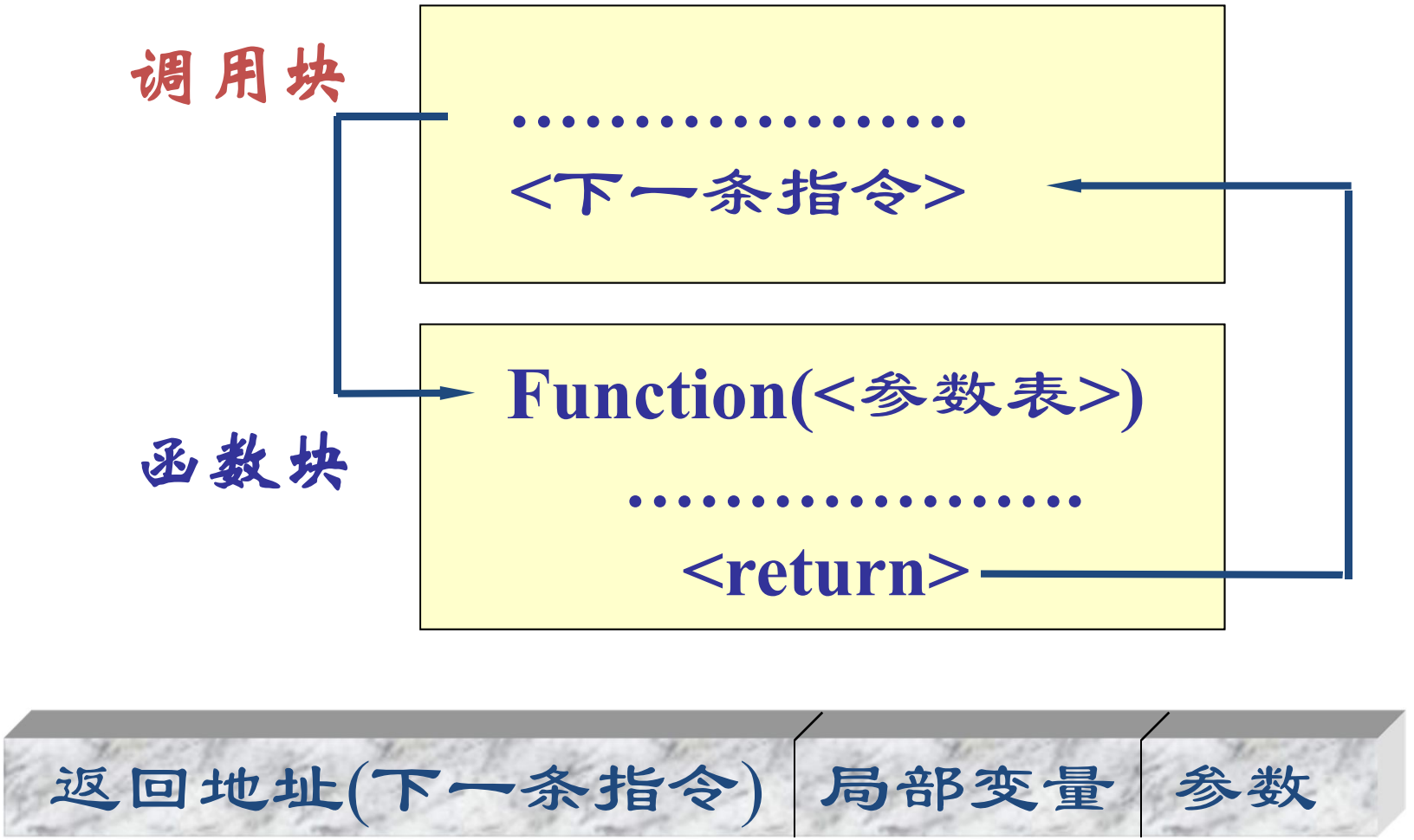
```
long Factorial(long n) {
  int temp;
  if (n == 0) return 1;
  else temp = n * Factorial(n−1);
  return temp;
}

void main() {
  int n;
  n = Factorial(4);
}
```

递归工作栈

- 每一次递归调用时，需要为过程中使用的参数、局部变量等另外分配存储空间
- 每层递归调用需分配的空间形成递归工作记录，按后进先出的栈组织



函数递归时的活动记录



计算Fact时活动记录的内容

递归过程改为非递归过程

- 递归过程简洁、易编、易懂
- 递归过程效率低，重复计算多
- 改为非递归过程的目的是提高效率
- 单向递归和尾递归可直接用迭代实现其非递归过程
  - **单向递归**：如求解斐波那契数列这样问题的递归。
  - **尾递归**：它的递归只有一条语句，且放在过程的最后。
    - 这时不必利用递归栈保存返回地址；
    - 返回后，参数和局部变量都不再需要。
- 其他情形必须借助栈实现非递归过程

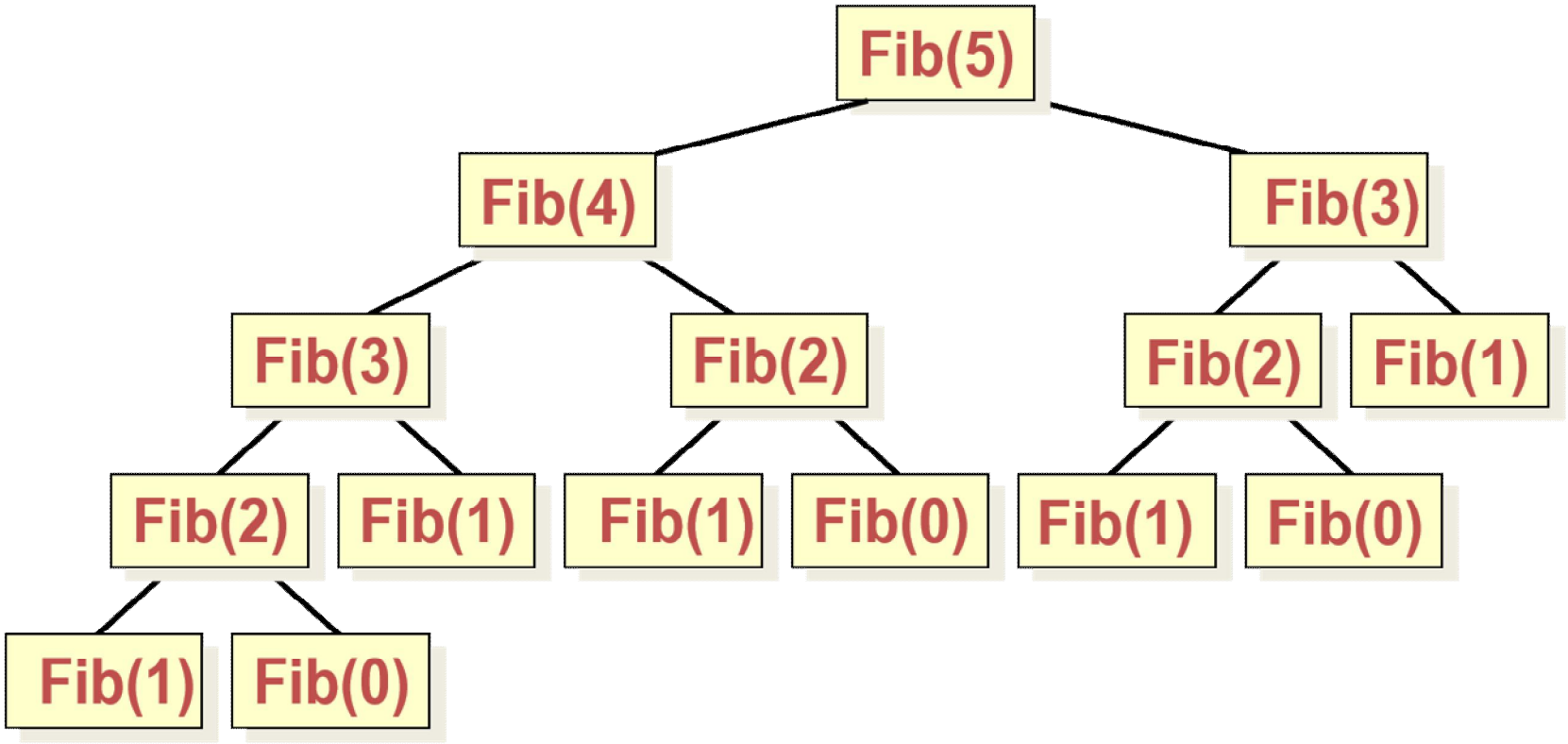
计算斐波那契数列

$$\mathrm{Fib}(n)=\begin{cases} n, & n=0,1 \\ \mathrm{Fib}(n-1)+\mathrm{Fib}(n-2), & n>1 \end{cases}$$

如 $\mathbf{F_0=0, F_1=1, F_2=1, F_3=2, F_4=3, F_5=5}$

求解斐波那契数列的递归算法

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



斐波那契数列的递归调用树

调用次数　NumCall(k) = 2*Fib(k+1)−1

单向递归用迭代法实现

```
long FibIter(long n) {
    if (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;
}
```

尾递归用迭代法实现

25	36	72	18	99	49	54	63

```
void recfunc(int A[ ],  int n) {
    if (n >= 0) {
        cout << A[n] << “  ”;
        n--;
        recfunc(A, n);
    }
}
```

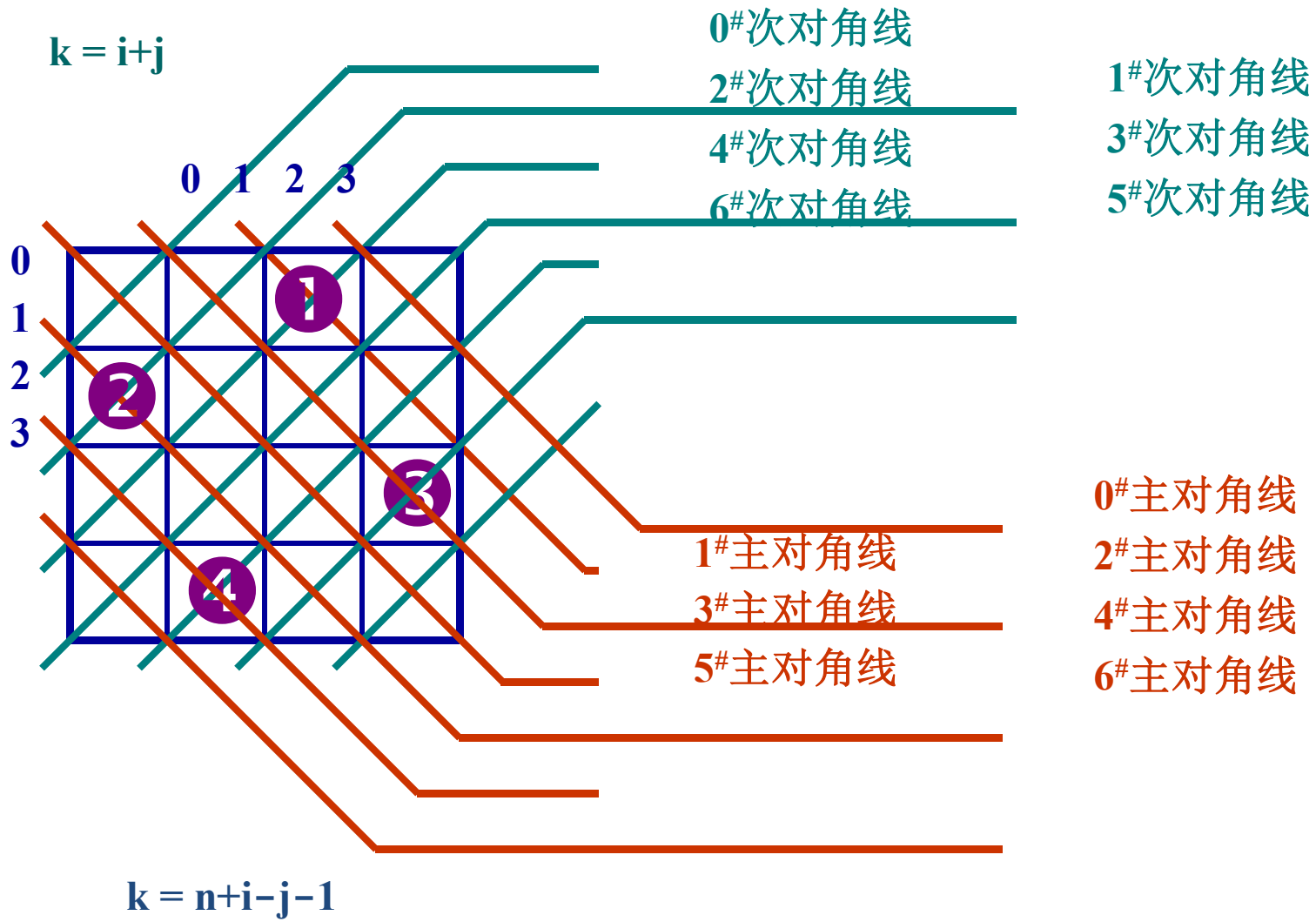
```
void sterfunc(int A[ ], int n) {
//消除了尾递归的非递归函数
    while (n >= 0) {
        cout << "value    " << A[n] << endl;
        n--;
    }
}
```

递归与回溯

- 对一个包含有许多结点，且每个结点有多个分支的问题，可以先选择一个分支进行搜索。当搜索到某一结点，发现无法再继续搜索下去时，可以沿搜索路径回退到前一结点，沿另一分支继续搜索。
- 如果回退之后没有其他选择，再沿搜索路径回退到更前结点，….。依次执行，直到搜索到问题的解，或搜索完全部可搜索的分支没有解存在为止。
- 回溯法与分治法本质相同，可用递归求解。

n皇后问题

- 在 n 行 n 列的国际象棋棋盘上，若两个皇后位于同一行、同一列、同一对角线上，则称为它们为互相攻击。n 皇后问题是指找到这 n 个皇后的互不攻击的布局。


k = i+j
0 1 2 3
0
1
2
3
1
2
3
4
0#次对角线
2#次对角线
4#次对角线
6#次对角线
1#次对角线
3#次对角线
5#次对角线
1#主对角线
3#主对角线
5#主对角线
0#主对角线
2#主对角线
4#主对角线
6#主对角线
k = n+i−j−1

解题思路

- 安放第 i 行皇后时，需要在列的方向从 0 到 n-1 试探 (j = 0, …, n-1)
- 在第 j 列安放一个皇后：
  - 如果在列、主对角线、次对角线方向有其它皇后，则出现攻击，撤消在第 j 列安放的皇后。
  - 如果没有出现攻击，在第 j 列安放的皇后不动，递归安放第 i+1行皇后。

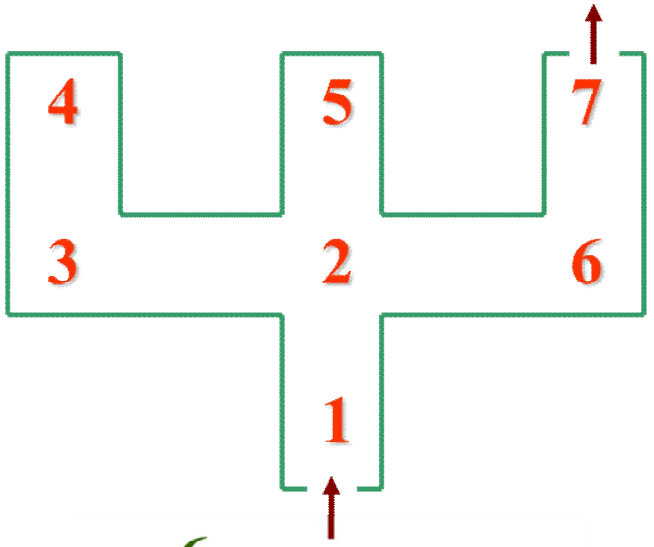
- 设置 4 个数组
  - col [n] ： col[i] 标识第 i 列是否安放了皇后
  - md[2n−1] : md[k] 标识第 k 条主对角线是否安放了皇后
  - sd[2n−1] : sd[k] 标识第 k 条次对角线是否安放了皇后
  - q[n] : q[i] 记录第 i 行皇后在第几列

```
void Queen(int i) {
    for (int j = 0;  j < n;  j++) {
        if (第 i 行第 j 列没有攻击) {
            在第 i 行第 j 列安放皇后;
            if (i == n-1) 输出一个布局;
            else Queen(i+1);
            撤消第 i 行第 j 列的皇后;
        }
    }
}
```

```
void Queen(int i) {
    for (int j = 0;  j < n;  j++) {
      if (!col[j] && !md[n+i-j-1] && !sd[i+j])
      {
          /*第 i 行第 j 列没有攻击 */
          col[j] = md[n+i-j-1] = sd[i+j] = 1;
          q[i] = j; /*在第 i 行第 j 列安放皇后*/
          if (i == n-1) {      /*输出一个布局*/
             for (int k = 0; k < n; k++)
                          cout << k << q[k] << ',';
              cout << endl;
          }
          else  Queen(i+1);
          col[j] = md[n+i-j-1] = sd[i+j] = 0;
          q[i] = 0;  /*撤消第 i 行第 j 列的皇后*/
      }
    }
}
```

迷宫问题

小型迷宫



路口	动作	结果
1 (入口)	正向走	进到 2
2	左拐弯	进到 3
3	右拐弯	进到 4
4 (堵死)	回溯	退到 3
3 (堵死)	回溯	退到 2
2	正向走	进到 5
5 (堵死)	回溯	退到 2
2	右拐弯	进到 6
6	左拐弯	进到 7
		(出口)

小型迷宫的数据

6

左行 0 3 0 0 0 0 7 7

直行 2 5 0 0 0 0 0

右行 0 6 4 0 0 0 0

类定义

- 迷宫

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
class Maze {
private:
    int MazeSize;
    int EXIT;
    Intersection *intsec;
public:
    Maze(char *filename);
    int TraverseMaze(int CurrentPos);
}
```

交通路口结构定义

```
struct Intersection {
    int left;
    int forward;
    int right;
}
```

```
Maze :: Maze(char *filename) {
//构造函数：从文件 filename 中读取各路口
//和出口的数据
    ifstream fin;
    fin.open(filename, ios::in | ios::nocreate);
     //为输入打开文件,文件不存在则打开失败
    if (!fin) {
        cerr <<“迷宫数据文件” << filename
             <<“打不开” << endl;
        exit(1);
    }
    fin >> MazeSize;          //输入迷宫路口数
```

```
        intsec = new Intersection[MazeSize+1];
         //创建迷宫路口数组
        for (int i = 1; i <= MazeSize; i++)
            fin >> intsec[i].left >> intsec[i].forward
                >> intsec[i].right;
        fin >> EXIT;                          //输入迷宫出口
        fin.close();
}
```

迷宫漫游与求解算法

```
int Maze::TraverseMaze(int CurrentPos) {
    if (CurrentPos > 0) {                     //路口从 1 开始
```

```
    if (CurrentPos == EXIT) {          //出口处理
        cout << CurrentPos << "   "; return 1;
    }
    else          //递归向左搜寻可行
    if (TraverseMaze(intsec[CurrentPos].left))
      { cout << CurrentPos << "   "; return 1; }
    else          //递归向前搜寻可行
    if (TraverseMaze(intsec[CurrentPos].forward))
      { cout << CurrentPos << "   "; return 1; }
    else          //递归向右搜寻可行
    if (TraverseMaze(intsec[CurrentPos].right))
      { cout << CurrentPos << "   "; return 1; }
  }
  return 0;
}
```

非递归程序的实现原理

- 与递归函数的原理相同，只不过把由系统负责的保存工作信息变为由程序自己保存
  - 减少保存数据的冗余(主要是节省了局部变量的空间)，提高存储效率
  - 减少函数调用的处理以及冗余的重复计算，提高时间效率
- 程序要完成的工作分成两类：
  - 手头工作　程序正在做的工作，必须有其结束条件，不能永远做下去
  - 保存在栈中的待完成的工作 由于某些工作不能一步完成，必须暂缓完成，把它保存在栈中，保存的待完成工作必须含有完成该项工作的所有必要信息。
- 程序须有序地完成各项工作
  - 手头工作和待完成工作可互相切换
  - 待完成工作必须转换成手头工作才能处理

谢谢！