数据结构与算法

第3章 栈与队列

操作受限的线性表

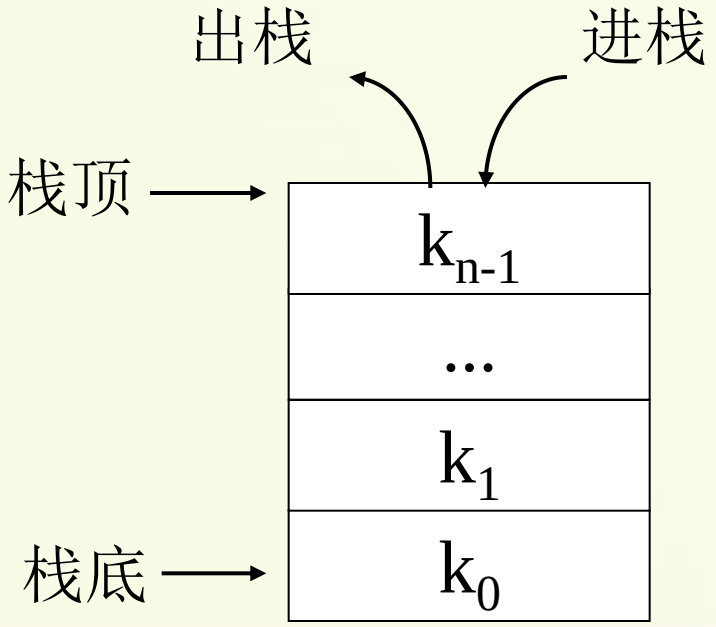
- 栈（Stack）
  - 运算只在表的一端进行
- 队列（Queue）
  - 运算只在表的两端进行

3.1 栈

- 后进先出（LastInFirstOut）
  - 一种限制访问端口的线性表
  - 栈存储和删除元素的顺序与元素到达的顺序相反
  - 也称为“下推表”
- 栈的主要元素
  - 栈顶（top）元素：栈的唯一可访问元素
    - 元素插入栈称为“入栈”或“压栈”（push）
    - 删除元素称为“出栈”或“弹出”（pop）
  - 栈底：另一端

栈的示意图

- 每次取出（并被删除）的总是刚压进的元素，而最先压入的元素则被放在栈的底部
- 当栈中没有元素时称为“空栈”



栈的主要操作

- 入栈（ push ）
- 出栈（pop）
- 取栈顶元素（ top ）
- 判断栈是否为空（ isEmpty ）

3.1.1 栈的抽象数据类型

```
template <class T>                    // 栈的元素类型为 T
class Stack {
public:                               // 栈的运算集
    void clear();                     // 变为空栈
    bool push(const T item);          // item入栈，成功则返回真，否则返回假
    bool pop(T & item);     // 返回栈顶内容并弹出，成功返回真，否则返回假
    bool top(T& item);      // 返回栈顶内容但不弹出，成功返回真，否则返回假
    bool isEmpty(;                    // 若栈已空返回真
    bool isFull();                    // 若栈已满返回真
};
```

栈的实现方式

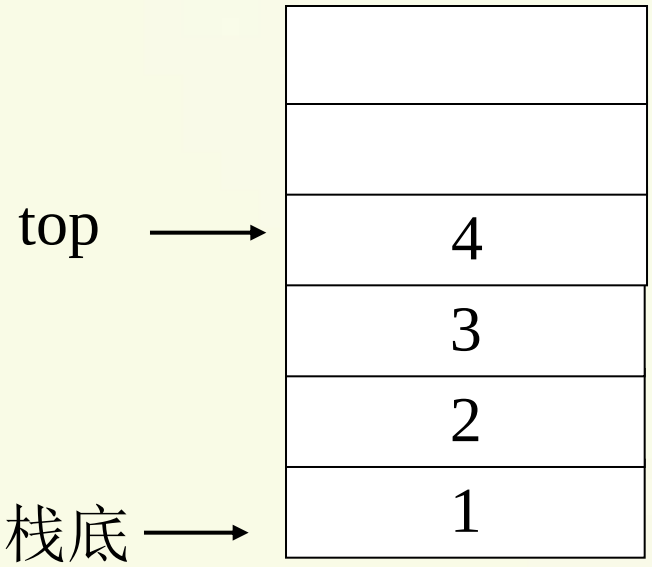
- 顺序栈（Array-based Stack）
  - 使用向量实现，本质上是顺序表的简化版
    - 栈的大小
  - 关键是确定哪一端作为栈顶
- 链式栈（Linked Stack）
  - 用单链表方式存储，其中指针的方向是从栈顶向下链接

3.1.2 顺序栈

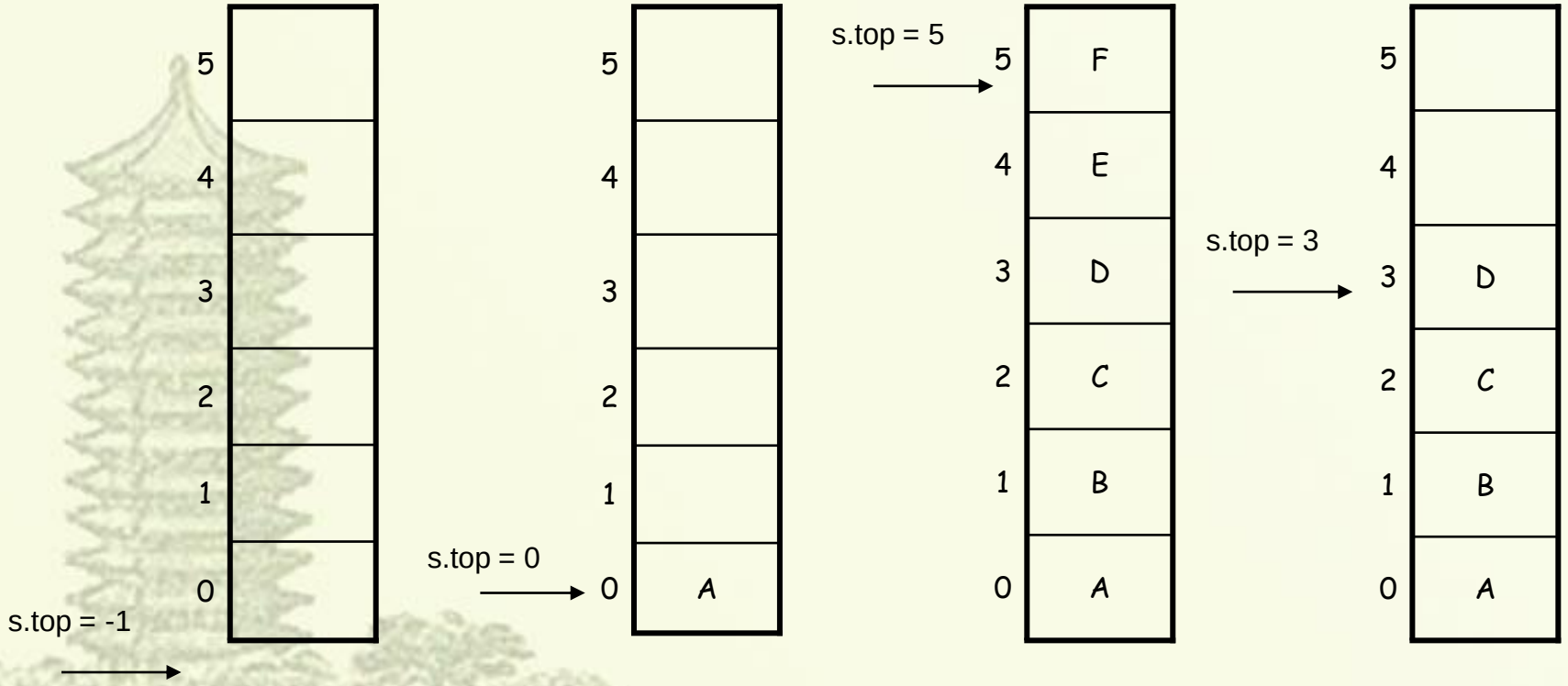
```
template <class T> class arrStack : public Stack <T> {
private:                                  // 栈的顺序存储
    int     mSize;                        // 栈中最多可存放的元素个数
    int     top;                          // 栈顶位置，应小于mSize
    T       *st;                          // 存放栈元素的数组
public:                                   // 栈的运算的顺序实现
    arrStack(int size) {                  // 创建一个给定长度的顺序栈实例
        mSize = size;  top = -1;
        st = new T[mSize];
    }
    arrStack() {                          // 创建一个顺序栈的实例
        top = -1;
    }
    ~arrStack() {                         // 析构函数
        delete [] st;
    }
    void clear() {                        // 清空栈内容
        top = -1;
    }
}
```

顺序栈

- 按压入先后次序，最后压入的元素编号为4，然后依次为3,2,1



顺序栈示意



顺序栈的溢出

- 上溢（Overflow）
  - 当栈中已经有maxsize个元素时，如果再做进栈运算，所产生的现象
- 下溢（Underflow）
  - 对空栈进行出栈运算时所产生的现象

顺序栈

- 若入栈的顺序为1,2,3,4的话，则出栈的顺序可以有哪些?
  - 1234
  - 1243
  - 1324
  - 1342
  - 1423
  - 1432
  - 2134
  - 2143

  ……

压入栈顶

```
bool arrStack<T>::push(const T item) {
    if (top == mSize-1) {                    // 栈已满
        cout << "栈满溢出" << endl;
        return false;
    }
    else {                    // 新元素入栈并修改栈顶指针
        st[++top] = item;
        return true;
    }
}
```

从栈顶弹出

```
bool arrStack<T>::pop(T & item) {        // 出栈的顺序实现
    if (top == -1) {                     // 栈为空
        cout << "栈为空，不能执行出栈操作"<< endl;
        return false;
    }
    else {
        item = st[top--];                // 返回栈顶元素并修改栈顶指针
        return true;
    }
}
```

从栈顶读取，但不弹出

```
bool arrStack<T>:: top(T & item) {
                                    // 返回栈顶内容，但不弹出
    if (top == -1) {                                    // 栈空
        cout << " 栈为空，不能读取栈顶元素"<< endl;
        return false;
    }
    else {
        item = st[top];
        return true;
    }
}
```

其他算法

- 把栈清空

```
void  arrStack<T>::clear() {
    top = -1;
}
```

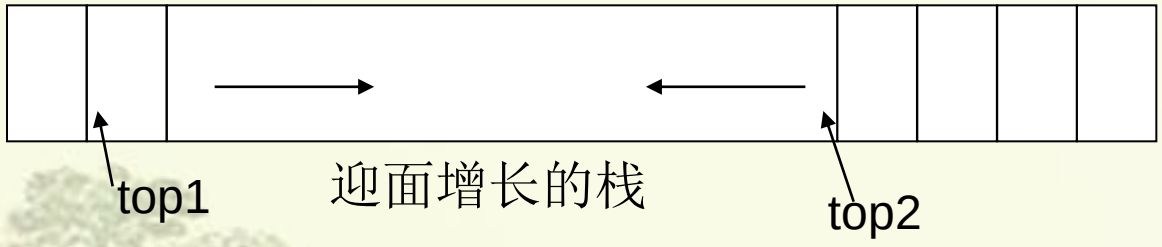
- 栈满时，返回非零值（真值true）

```
bool arrStack<T>:: isFull() {
    return  (top == maxsize-1) ;
}
```

栈的变种

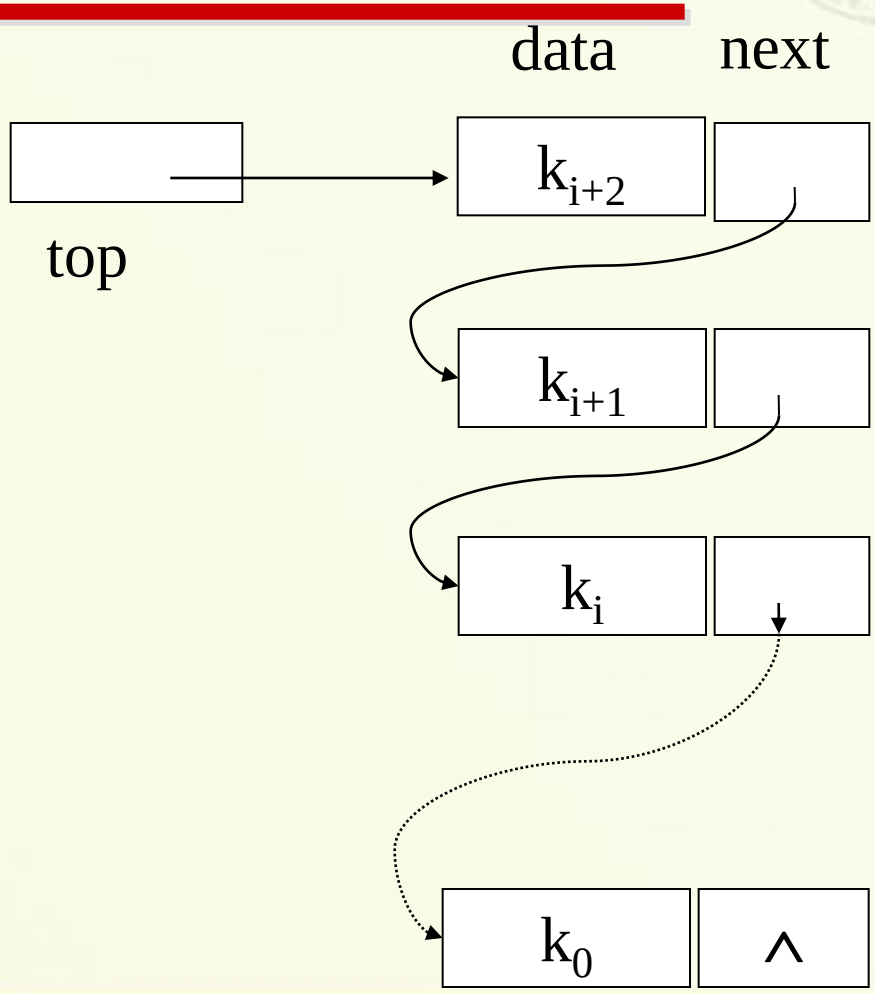
- 两个独立的栈
  - 底部相连：双栈
  - 迎面增长

哪一种更好些？



3.1.3 链式栈

- 用单链表方式存储
- 指针的方向从栈顶向下链接



链式栈的创建

```
template <class T>
class lnkStack : public Stack <T> {
private:              // 栈的链式存储
    Link<T>*          top;      // 指向栈顶的指针
    int               size;     // 存放元素的个数
public:                                  // 栈运算的链式实现
    lnkStack(int defSize) {                      // 构造函数
    top = NULL;
    size = 0;
    }
    ~lnkStack() {                        // 析构函数
          clear();
    }
}
```

压入栈顶

```
// 入栈操作的链式实现
bool lnksStack<T>:: push(const T item) {
   Link<T>* tmp = new Link<T>(item, top);
    top = tmp;
    size++;
    return true;
}
```

自单链栈弹出

```
// 出栈操作的链式实现

bool lnkStack<T>:: pop(T& item) {
    Link <T> *tmp;
    if (size == 0) {
        cout << "栈为空，不能执行出栈操作"<< endl;
        return false;
    }
    item = top->data;
    tmp = top->next;
    delete top;
    top = tmp;
    size--;
    return true;
}
```

顺序栈和链式栈的比较

- 时间效率
  - 所有操作都只需常数时间
  - 顺序栈和链式栈在时间效率上难分伯仲
- 空间效率
  - 顺序栈须说明一个固定的长度
  - 链式栈的长度可变，但增加结构性开销

顺序栈和链式栈的比较

- 实际应用中，顺序栈比链式栈用得更广泛些
  - 顺序栈容易根据栈顶位置，进行相对位移，快速定位并读取栈的内部元素
  - 顺序栈读取内部元素的时间为O(1)，而链式栈则需要沿着指针链游走，显然慢些，读取第k个元素需要时间为O(k)
    - 一般来说，栈不允许“读取内部元素”，只能在栈顶操作

3.1.4 栈的应用

- 栈的特点：后进先出
  - 体现了元素之间的透明性
- 常用来处理具有递归结构的数据
  - 深度优先搜索
  - 表达式求值
  - 子程序 / 函数调用的管理
  - 消除递归

计算表达式的值

- 表达式的递归定义
  - 基本符号集：{0，1，…，9，+，-，*，/，（，）}
  - 语法成分集：{〈表达式〉，〈项〉，〈因子〉，〈常数〉，〈数字〉 }
  - 语法公式集
- 中缀表达式

  23+(34*45)/(5+6+7)
- 后缀表达式

  23 34 45 * 5 6 + 7 + / +
- 后缀表达式求值

中缀表达法的语法公式

〈表达式〉 ::= 〈项〉 + 〈项〉
| 〈项〉 — 〈项〉
| 〈项〉

〈项〉 ::= 〈因子〉 * 〈因子〉
| 〈因子〉 / 〈因子〉
| 〈因子〉

〈因子〉 ::= 〈常数〉
| (〈表达式〉)

〈常数〉 ::= 〈数字〉
| 〈数字〉〈常数〉

〈数字〉 ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

中缀表达的算术表达式的计算次序

1. 先执行括号内的计算，后执行括号外的计算。在具有多层括号时，按层次反复地脱括号，左右括号必须配对
2. 在无括号或同层括号时，先乘(*) 、除(/)，后加(+)、减(-)
3. 在同一个层次，若有多个乘除（*、 / ）或加减(+，-)的运算，那就按自左至右顺序执行

23+(34*45)/(5+6+7) ＝？

计算特点？

后缀表达式

<表达式> ::= <项> <项> +
| <项> <项> −
| <项>

<项> ::= <因子> <因子> *
| <因子> <因子> /
| <因子>

<因子> ::= <常数>

<常数> ::= <数字>
| <数字><常数>

<数字> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

后缀表达式的计算

- 23 34 45 * 5 6 + 7 + / + =?

计算特点？

中缀和后缀表达式的主要异同？

23+34*45/(5+6+7) = ?

23 34 45 * 5 6 + 7 + / + =?

中缀表达式→后缀表达式

- 从左到右扫描中缀表达式。用栈存放表达式中的操作数、开括号以及在此开括号后暂不确定计算次序的其他符号：
  (1) 当输入的是操作数时，直接输出到后缀表达式序列；
  (2) 当遇到开括号时，把它压入栈；
  (3) 当输入遇到闭括号时，先判断栈是否为空，若为空（括号不匹配），应该作为错误异常处理，清栈退出。若非空，则把栈中元素依次弹出，直到遇到第一个开括号为止，将弹出的元素输出到后缀表达式序列中（弹出的开括号不放到序列中），若没有遇到开括号，说明括号不配对，做异常处理，清栈退出。

中缀表达式→后缀表达式

- 从左到右扫描中缀表达式。用栈存放表达式中的操作数、开括号以及在此开括号后暂不确定计算次序的其他符号：

（4）当输入为运算符op（ 四则运算 + - * / 之一）时

（a）循环 当（栈非空 and 栈顶不是开括号 and 栈顶运算符的优先级不低于输入的运算符的优先级）时，反复操作

将栈顶元素弹出，放到后缀表达式序列中；

（b）将输入的运算符压入栈内；

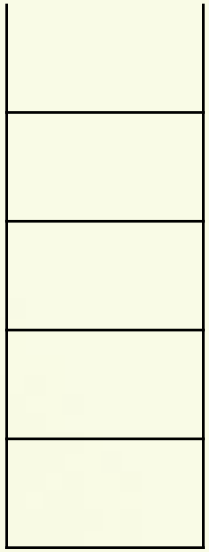
（5）最后，当中缀表达式的符号全部读入时，若栈内仍有元素，把它们全部依次弹出，放在后缀表达式序列的尾部。若弹出的元素遇到开括号，则说明括号不匹配，做异常处理，清栈退出。

中缀表达式→后缀表达式

待处理中缀表达式：

23	+	(	34 *	45	)	/	(	5	+	6	+	7	)

栈的状态



输出后缀表达式：

后缀表达式求值

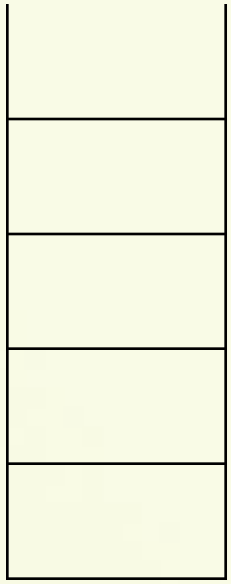
- 循环：依次顺序读入表达式的符号序列（假设以＝作为输入序列的结束），并根据读入的元素符号逐一分析：
  1. 当遇到的是一个操作数，则压入栈顶；
  2. 当遇到的是一个运算符，就从栈中两次取出栈顶，按照运算符对这两个操作数进行计算。然后将计算结果压入栈顶
- 如此继续，直到遇到符号＝，这时栈顶的值就是输入表达式的值

后缀表达式求值

待处理后缀表达式：

23	34	45	*	5	6	+	7	+	/	+

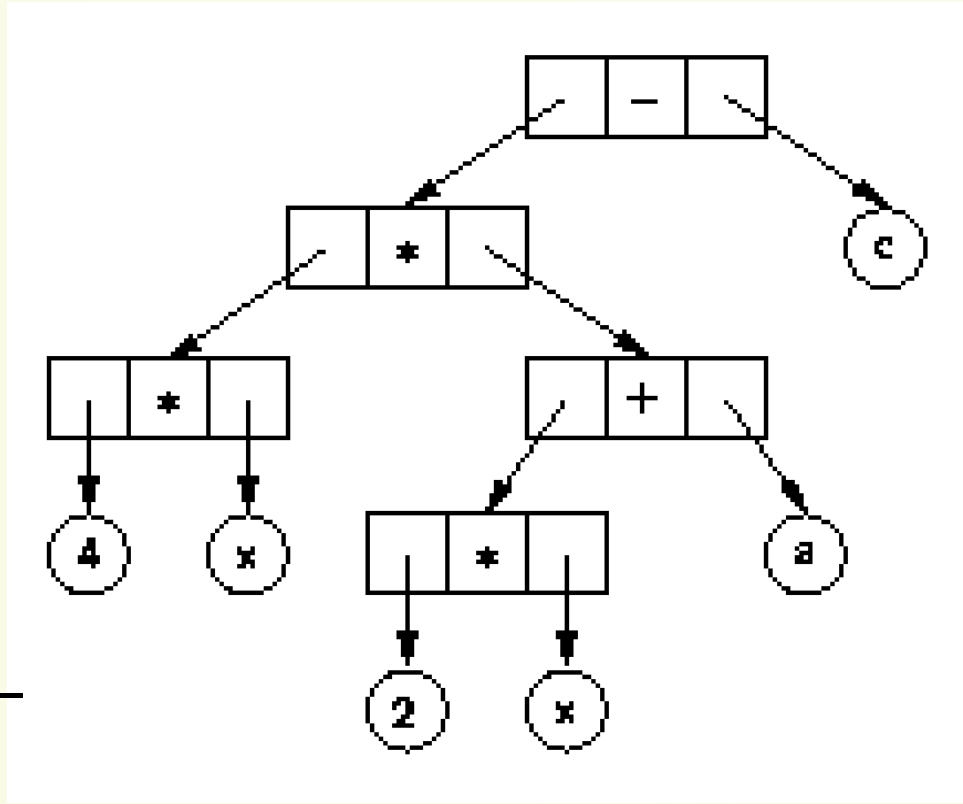
栈状态的变化



1530
11
18
85
108

后缀表达式求值

- 中缀表达式：
  - 运算符在中间
  - 需要括号改变优先级
  - 4* x * (2 * x + a) – c
- 后缀表达式
  - 运算符在后面
  - 完全不需要括号
  - 4 x * 2 x * a + * c –



3.1.5 栈与递归

- 函数的递归定义
- 主程序和子程序的参数传递
- 栈在实现函数递归调用中所发挥的作用

递归

- 作为数学和计算机科学的基本概念，递归是解决复杂问题的一个有力手段
  - 符合人类解决问题的思维方式，递归能够描述和解决很多问题，为此许多程序设计语言都提供了递归的机制
  - 而计算机则只能按照指令集来顺序地执行。
- 计算机内（编译程序）是如何将程序设计中的便于人类理解的递归程序转换为计算机可处理的非递归程序的?

递归定义阶乘n！函数

- 阶乘n！的递归定义如下：

$$\text{factiorial}(n)=\begin{cases}1, \text{if } n\le 0\\ n\times\text{factiorial}(n-1), \text{if } n>0\end{cases}$$

- 递归定义由两部分组成：
  - 递归基础/出口
  - 递归规则

计算阶乘n！的循环迭代程序

```
// 使用循环迭代方法，计算阶乘n!的程序
long factorial(long n) {
    int  m = 1;
    int  i ;
    if (n>0)
        for ( i = 1;  i  <= n;  i++ )
                m  = m * i ;
    return m ;
}
```

计算阶乘n！的递归程序

```
// 递归定义的计算阶乘n!的函数
long factorial(long n) {
    if (n == 0)
        return 1;
    else
        return n * factorial( n-1) ;  // 递归调用
}
```

递归问题求解过程示例：阶乘

回推：

4! = 4*3!

4*3! = 24

3! = 3*2!

2! = 2*1!

1! = 1*0!

递推：

4! = 4*3! = 24

3! = 3*2! = 6

2! = 2*1! = 2

1! = 1*0! = 1

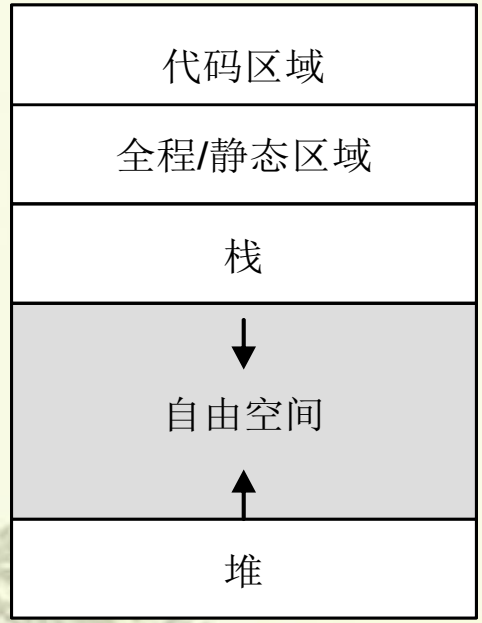
0! = 1 已知

递归函数的实现

- 栈的一个最为广泛的应用也许是用户所看不到，此即大多数程序设计语言运行环境都提供的函数调用机制。
  - 运行时环境指的是目标计算机上用来管理存储器并保存指导执行过程所需的信息的寄存器及存储器的结构。
- 函数调用
  - 在非递归情况下，数据区的分配可以在程序运行前进行，一直到整个程序运行结束才释放，这种分配称为静态分配。采用静态分配时，函数的调用和返回处理比较简单，不需要每次分配和释放被调函数的数据区
  - 在递归调用的情况下，被调函数的局部变量不能静态地分配某些固定单元，而必须每调用一次就分配一份，以存放当前所使用的数据，当返回时随即释放。故其存储分配只能在执行调用时才能进行，即所谓的动态分配：在内存中开辟一个称为运行栈（runtime stack）的足够大的动态区

函数运行时的动态存储分配

- 用作动态数据分配的存储区可按多种方式组织。典型的组织是将这个存储器分为栈（stack）区域和堆（heap）区域
  - 栈区域用于分配发生在后进先出LIFO风格中的数据（诸如函数的调用）
  - 堆区域则用于不符合LIFO（诸如指针的分配）的动态分配



函数调用与返回处理

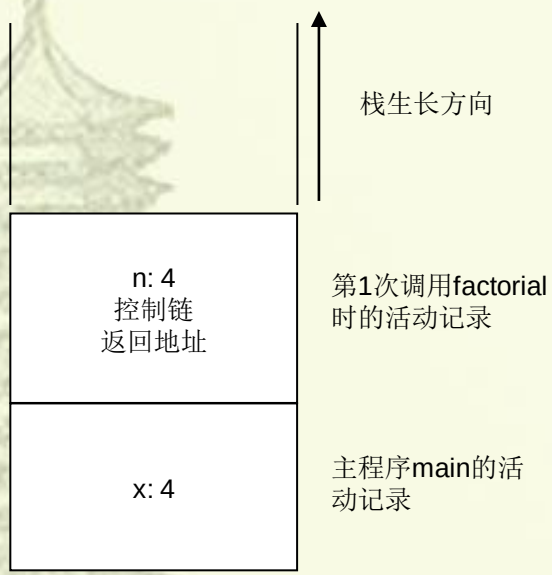
- 调用可以分解成以下三步来实现：
  - 调用函数发送调用信息。调用信息包括调用方要传送给被调方的信息，诸如实参、返回地址等。
  - 分配被调方需要的局部数据区，用来存放被调方定义的局部变量、形参变量（存放实参）、返回地址等，并接受调用方传送来的调用信息。
  - 调用方暂停，把计算控制转到被调方，亦即自动转移到被调用的函数的程入口。
- 当被调方结束运行，返回到调用方时，其返回处理一般也分解为三步进行：
  - 传送返回信息，包括被调方要传回给调用方的信息，诸如计算结果等。
  - 释放分配给被调方的数据区。
  - 按返回地址把控制转回调用方。

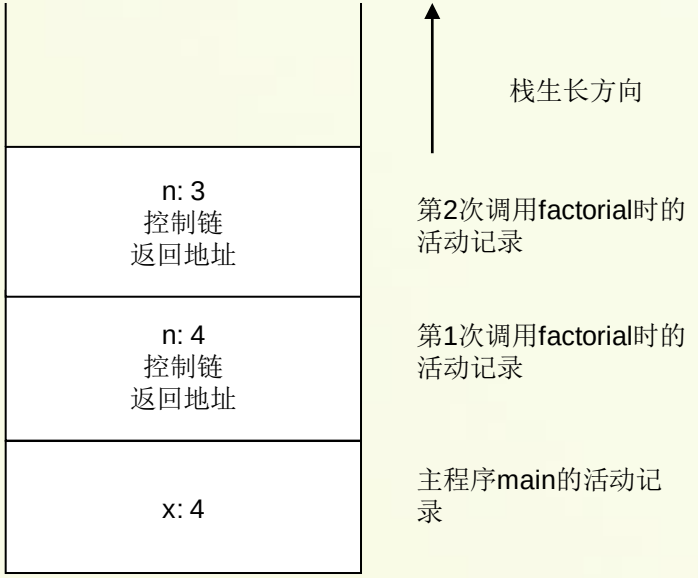
递归时运行栈的变化示例

以阶乘为例：

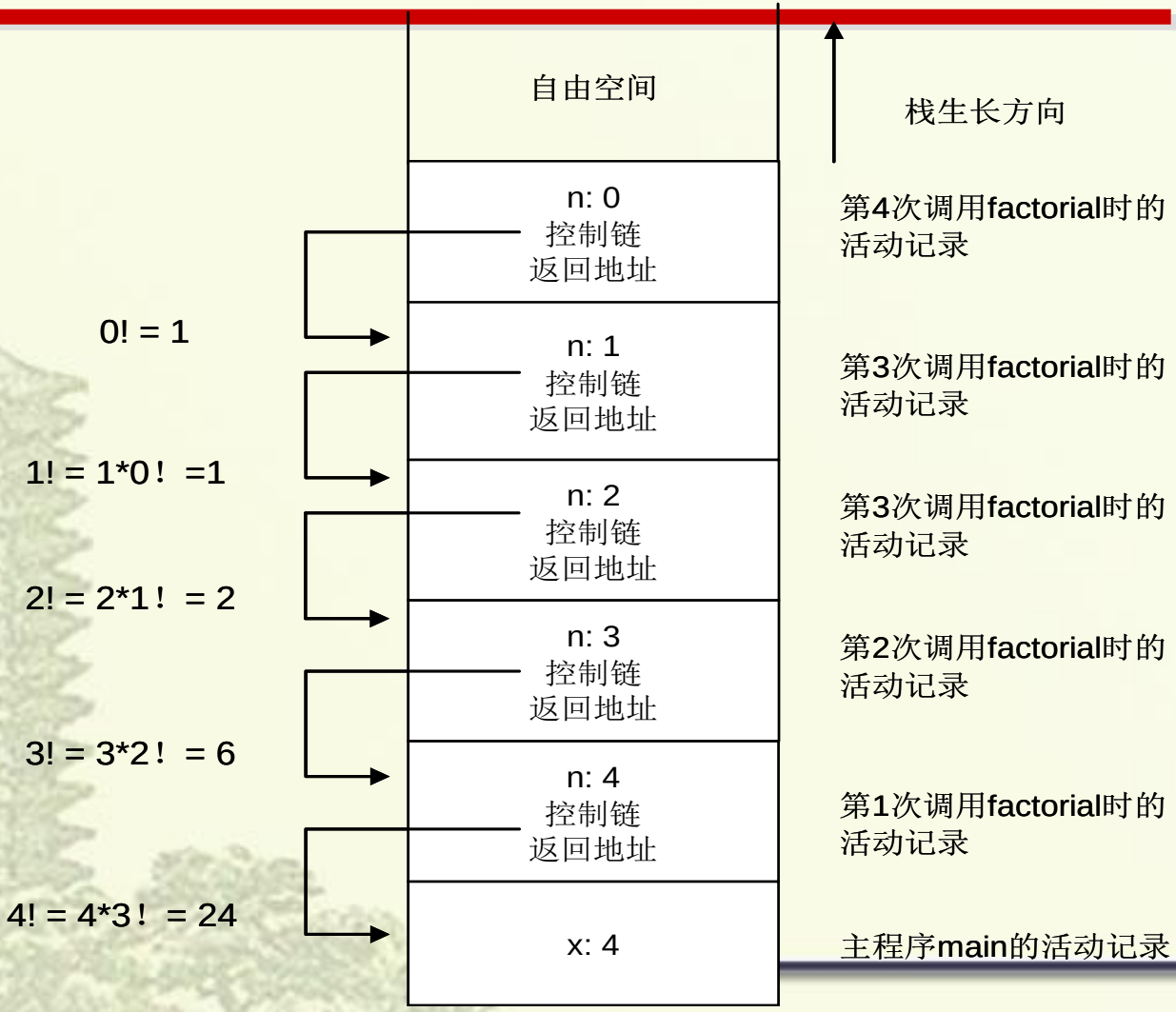
```
#include <stdio.h>
main() {
    int x;
    scanf(“%d”, &x);
    printf(“%d\n”, factorial(4));
}
```

计算阶乘时的运行栈

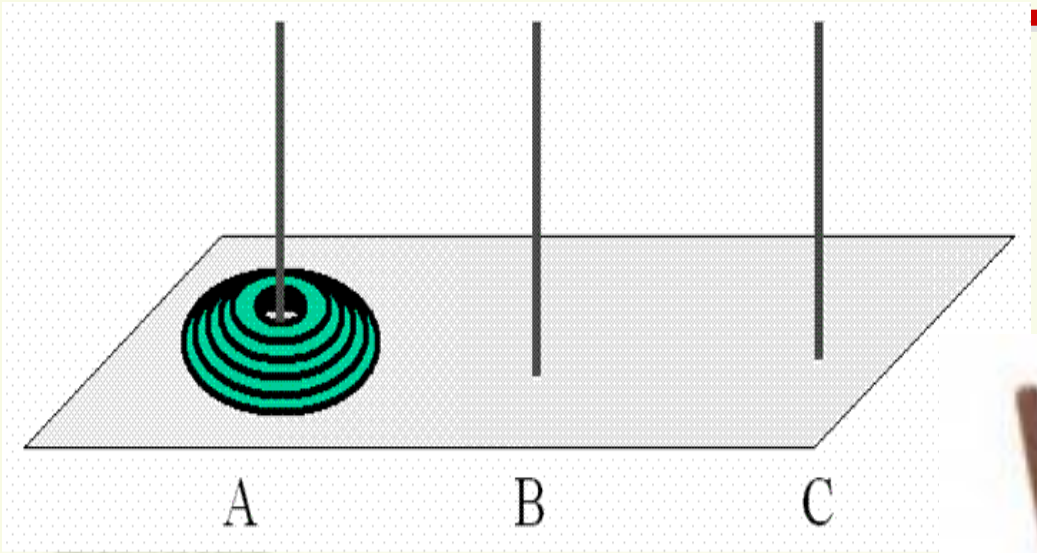

栈生长方向
n: 4
控制链
返回地址
第1次调用factorial
时的活动记录
x: 4
主程序main的活
动记录


栈生长方向
n: 3
控制链
返回地址
第2次调用factorial时的
活动记录
n: 4
控制链
返回地址
第1次调用factorial时的
活动记录
x: 4
主程序main的活动记
录

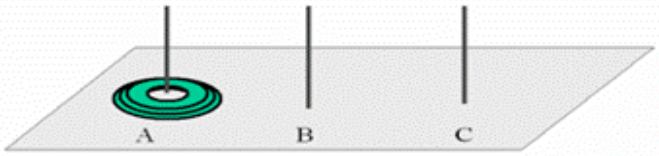
计算阶乘时的运行栈


自由空间
栈生长方向
n: 0
控制链
返回地址
第4次调用factorial时的活动记录
0! = 1
n: 1
控制链
返回地址
第3次调用factorial时的活动记录
1! = 1*0！ =1
n: 2
控制链
返回地址
第3次调用factorial时的活动记录
2! = 2*1！ = 2
n: 3
控制链
返回地址
第2次调用factorial时的活动记录
3! = 3*2！ = 6
n: 4
控制链
返回地址
第1次调用factorial时的活动记录
4! = 4*3！ = 24
x: 4
主程序main的活动记录

hanoi塔问题的递归求解

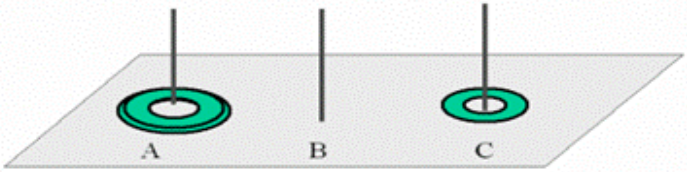


hanoi塔的执行过程（n=3）



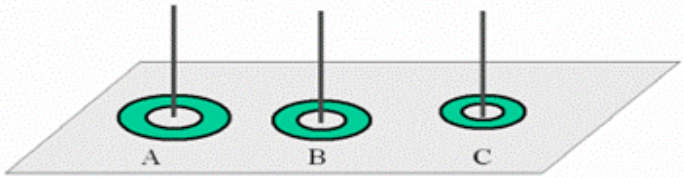
1

move(A,C)



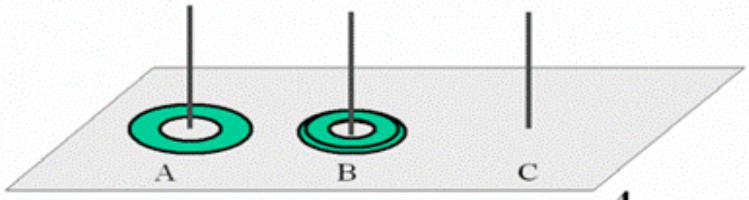
2

move(A,B)



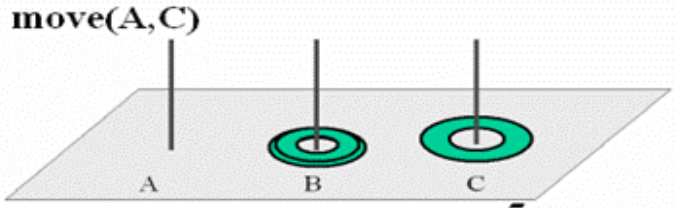
3

move(C,B)



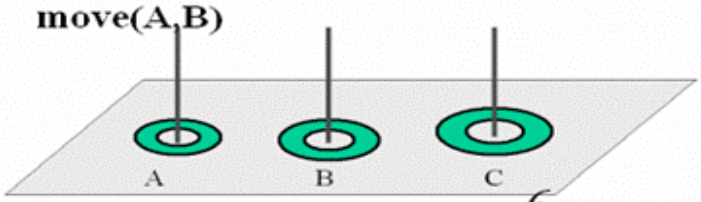
4

move(A,C)



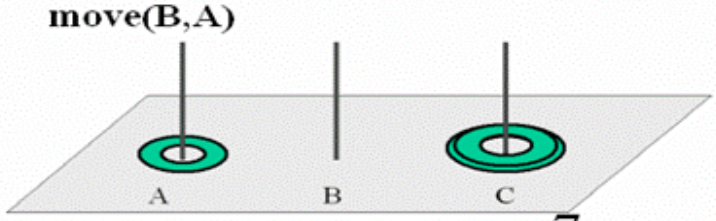
5

move(A,B)



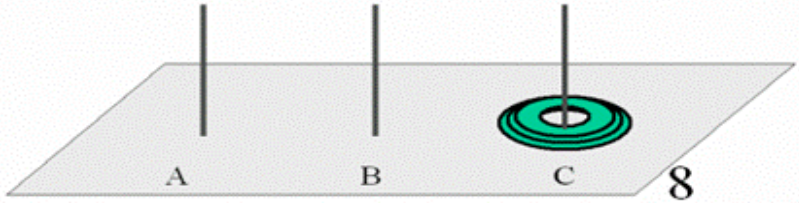
6

move(B,A)



7

move(A,C)



8

hanoi塔问题的递归求解程序

hanoi(n,X,Y,Z)的含义是：

移动 *n* 个槃环，由X柱子（出发柱）将槃环移动到Z 柱（终点柱）

其移动过程中， Y柱和X柱皆可用于暂时存放环槃，不过注意每一步移动都必须严格遵循大盘不能压小盘的原则

例如， hanoi(2, ‘B’, ‘C’ , ‘A’), 其含义是 初始位于B柱上部的2个环槃移动 到A 柱，执行过程中可以使用C，B 柱暂存环槃

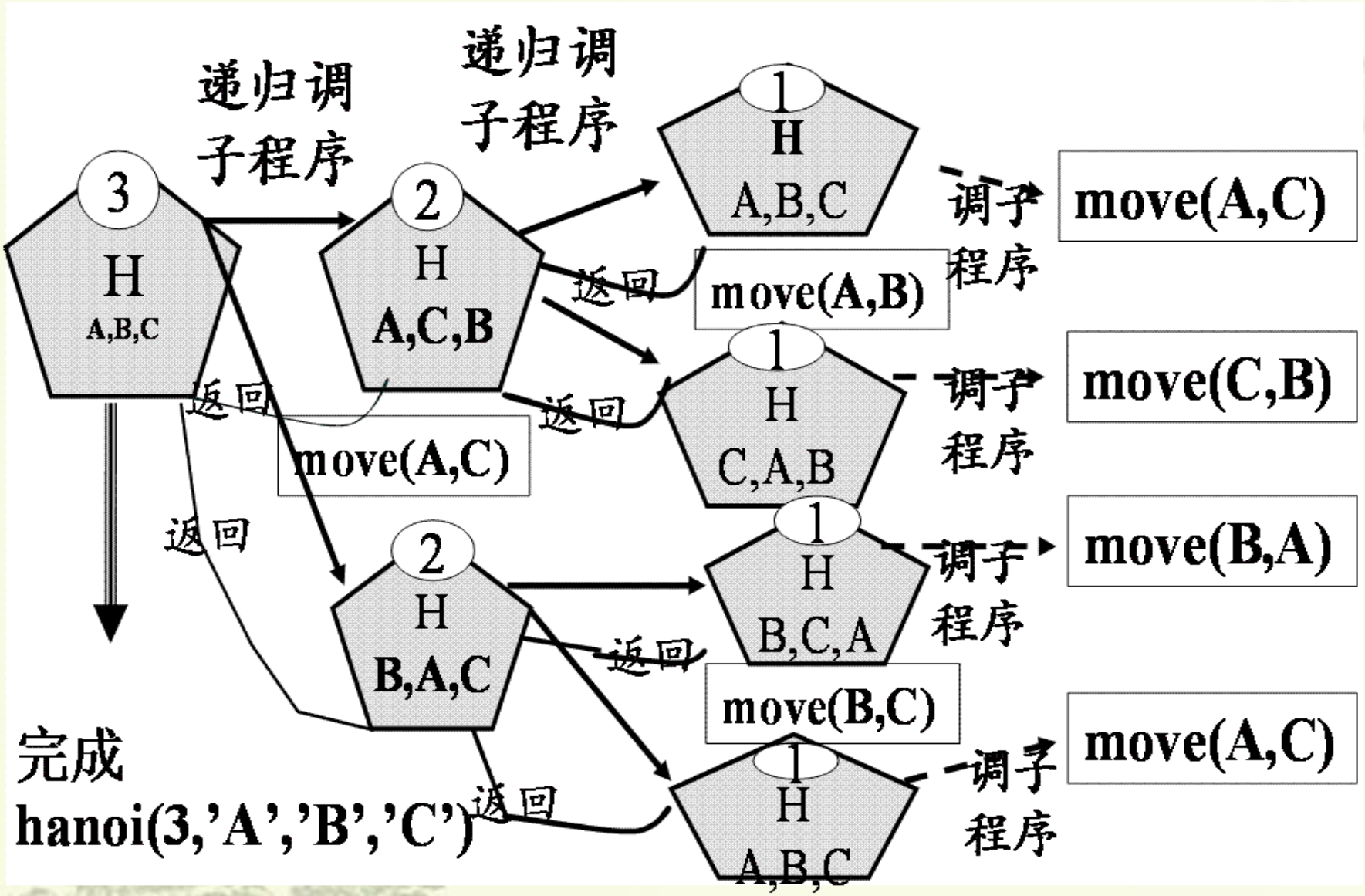
hanoi塔问题的递归求解

```
// move(char X, char Y)子程序，表示移动一步，

// 输出打印，把柱X的顶部环槃移到柱Y
void move(char X, char Y) {
    cout  << " move "  <<  X  << "to" << Y  << endl;
}
```

hanoi塔问题的递归求解

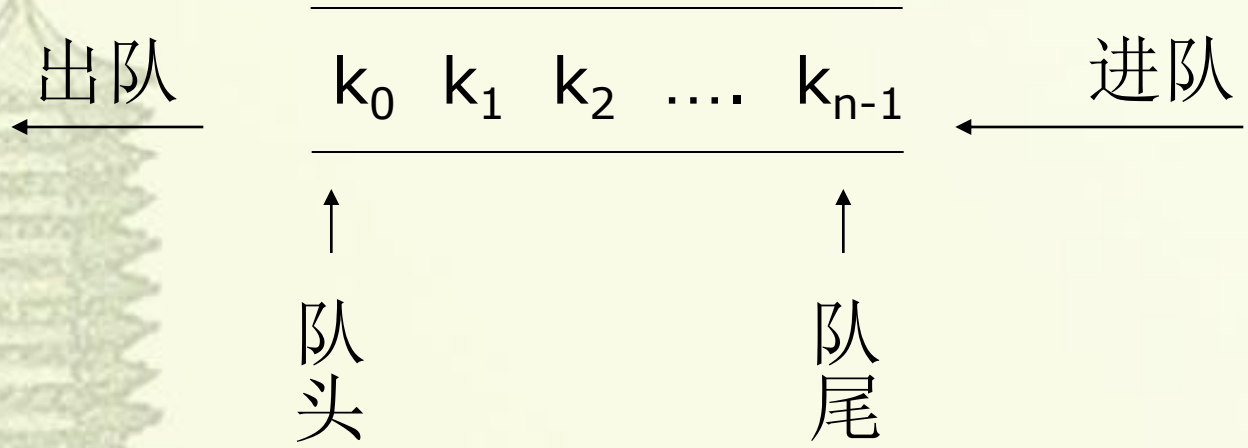
```
void hanoi(int n, char X, char Y, char Z)  {
    if (n <= 1)
        move(X, Z);
    else {
        // 最大的环槃在X上不动，把X上的n-1个环槃移到Y
        hanoi(n-1, X, Z, Y);
        move(X, Z);          // 移动最大环槃到Z，放好
        hanoi(n-1,Y,X,Z);  // 把 Y上的n－1个环槃移到Z
    }
}
```


递归调
子程序
递归调
子程序
3
H
A,B,C
2
H
A,C,B
1
H
A,B,C
调子
程序
move(A,C)
返回
move(A,B)
1
H
C,A,B
调子
程序
move(C,B)
返回
返回
move(A,C)
返回
2
H
B,A,C
1
H
B,C,A
调子
程序
move(B,A)
返回
move(B,C)
1
H
A,B,C
调子
程序
move(A,C)
返回
完成
hanoi(3,'A','B','C')

3.2 队列

- 先进先出（FirstInFirstOut）
  - 限制访问点的线性表
    - 按照到达的顺序来释放元素
    - 所有的插入在表的一端进行，所有的删除都在表的另一端进行
- 主要元素
  - 队头（front）
  - 队尾（rear）

队列示意图



队列的主要操作

- 入队列（enQueue）
- 出队列（deQueue）
- 取队首元素（getFront）
- 判断队列是否为空（isEmpty）

3.2.1 队列的抽象数据类型

```
template <class T>  class Queue  {
public:                                  // 队列的运算集
    void clear();                        // 变为空队列
    bool enQueue(const T item);
                    // 将item插入队尾，成功则返回真，否则返回假
    bool deQueue(T& item);
                    // 返回队头元素并将其从队列中删除，成功则返回真
    bool getFront(T& item);
                    // 返回队头元素，但不删除，成功则返回真
    bool isEmpty();                      // 返回真，若队列已空
    bool isFull();                       // 返回真，若队列已满
};
```

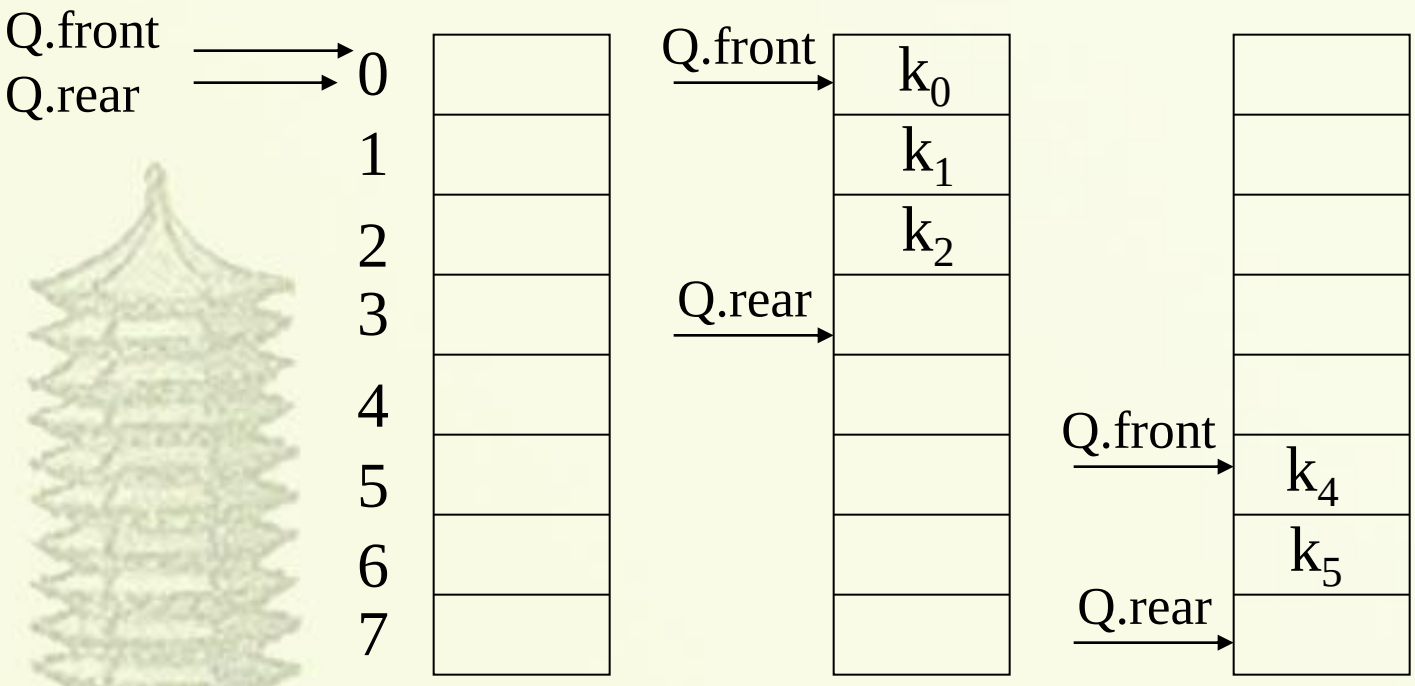
队列的实现方式

- 顺序队列
  - 关键是如何防止假溢出
- 链式队列
  - 用单链表方式存储，队列中每个元素对于链表中的一个结点

3.2.2 顺序队列

- 使用顺序表来实现队列
- 用向量存储队列元素，并用两个变量分别指向队列的前端和尾端

队列示意：普通



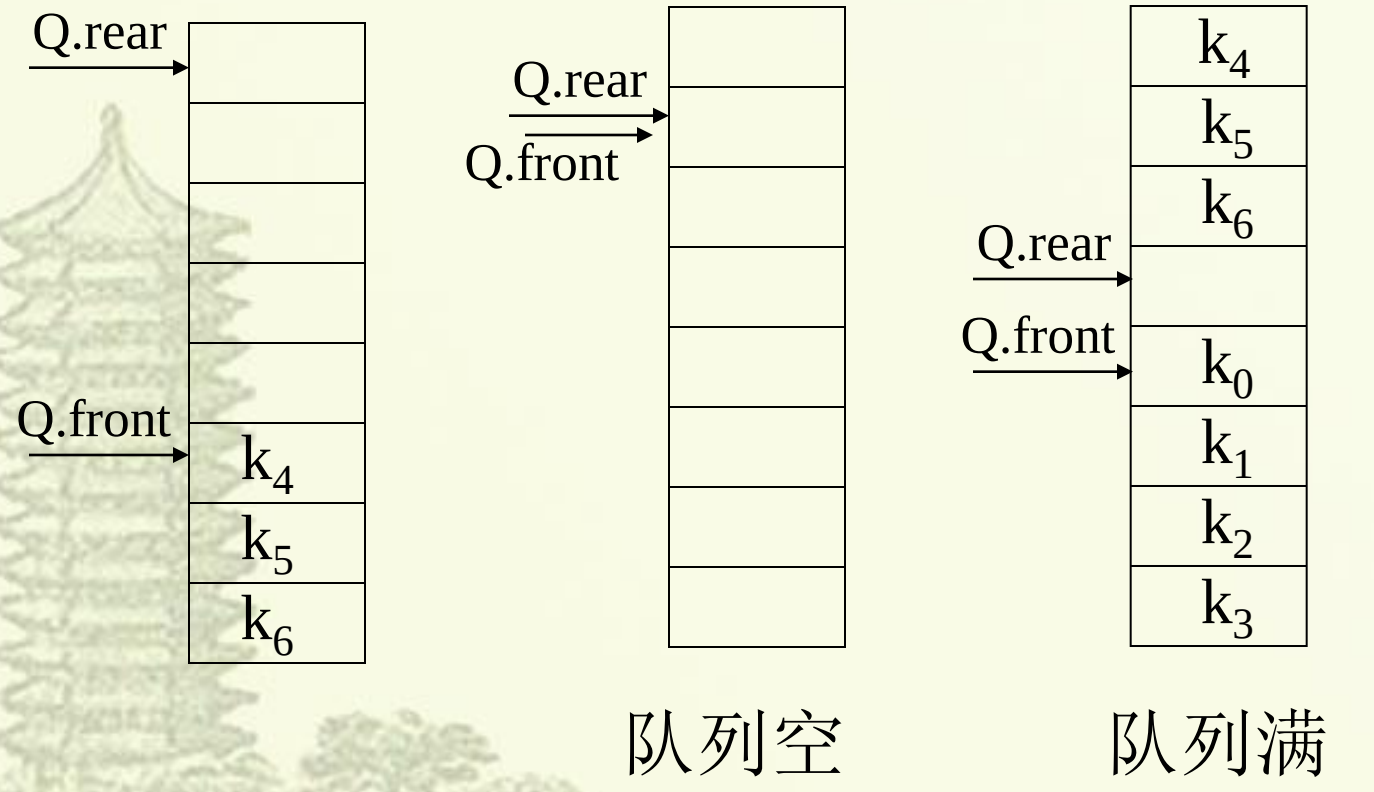
队列空

再进队一个元素如何？

队列的溢出

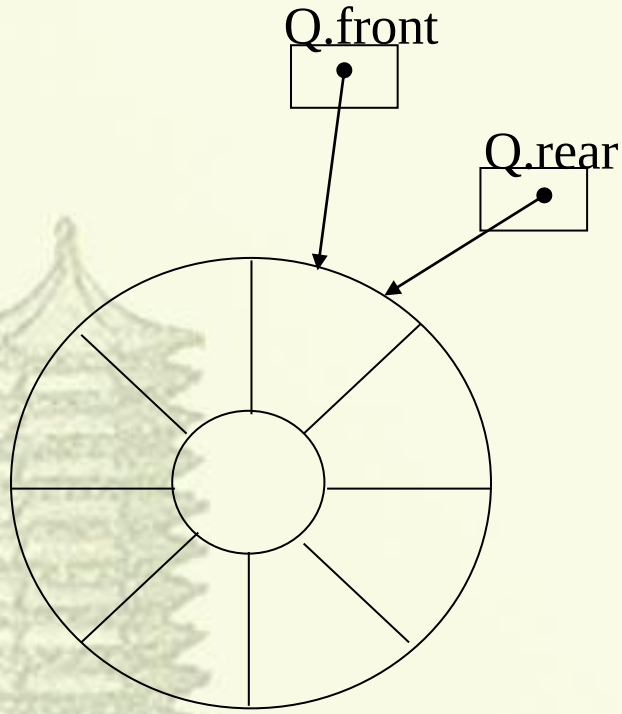
- 上溢
  - 当队列满时，再做进队操作，所出现的现象
- 下溢
  - 当队列空时，再做删除操作，所出现的现象
- 假溢出
  - 当 rear = MAXNUM时，再作插入运算就会产生溢出，如果这时队列的前端还有许多空的（可用的）位置，这种现象称为假溢出

队列示意：环形

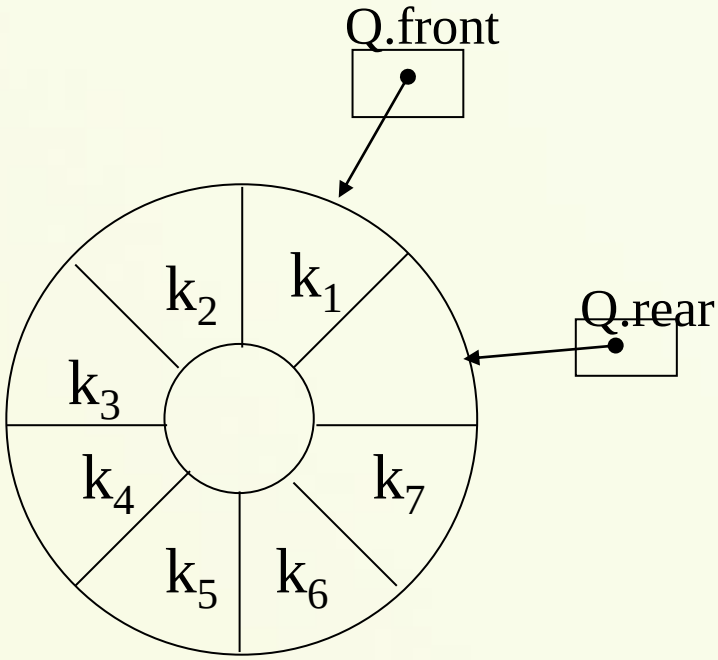


队列空　　队列满

队列示意：环形



空队列

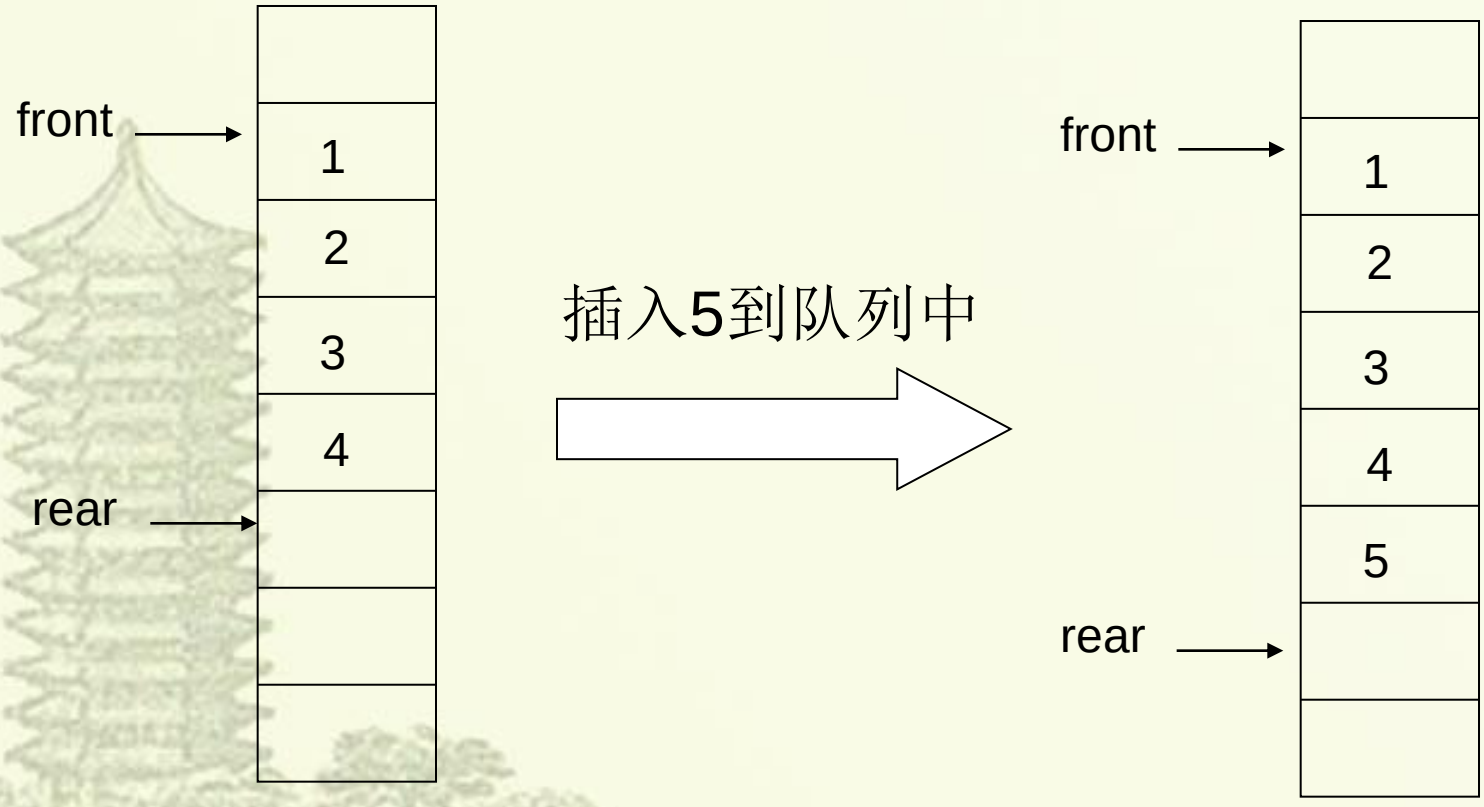


队列满，判断
(Q.rear+1) == Q.front

顺序队列的类定义

```
class arrQueue: public Queue<T> {
private:
    int   mSize;                    // 存放队列的数组的大小
    int   front;                    // 表示队头所在位置的下标
    int   rear;                     // 表示队尾所在位置的下标
    T     *qu;                      // 存放类型为T的队列元素的数组
public:                             // 队列的运算集
    arrQueue(int size) {            // 创建队列的实例
        mSize = size +1;  // 浪费一个存储空间，以区别队列空和队列满
        qu = new T [mSize];
        front = rear = 0;
    }
    ~arrQueue() {                   // 消除该实例，并释放其空间
        delete [] qu;
    }
}
```

新元素插入队列尾端



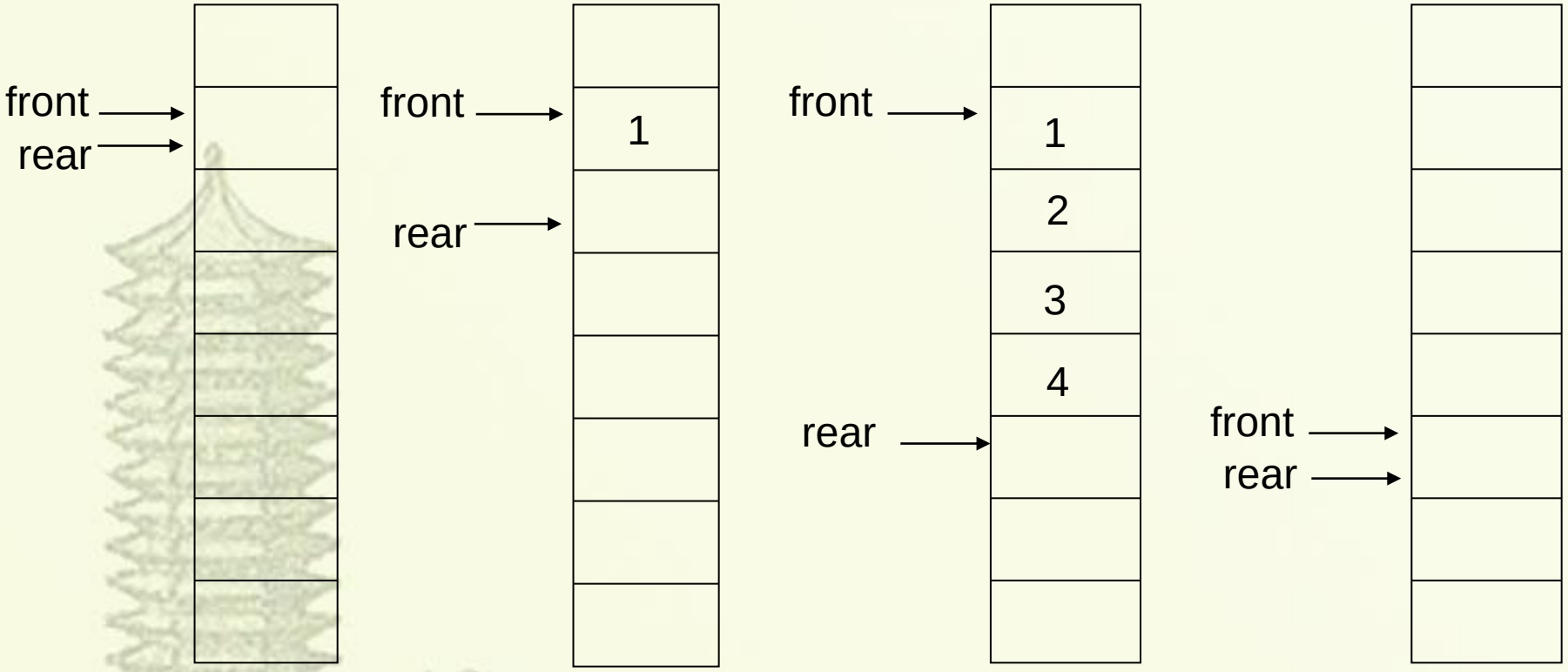
队列的插入

```
bool arrQueue<T> :: enQueue(const T item) {
                                                    // item入队，插入队尾
    if (((rear + 1 ) % mSize) == front) {
        cout << "队列已满，溢出" << endl;
        return false;
    }
    qu[rear] = item;
    rear = (rear +1) % mSize;                       // 循环后继
    return true;
}
```

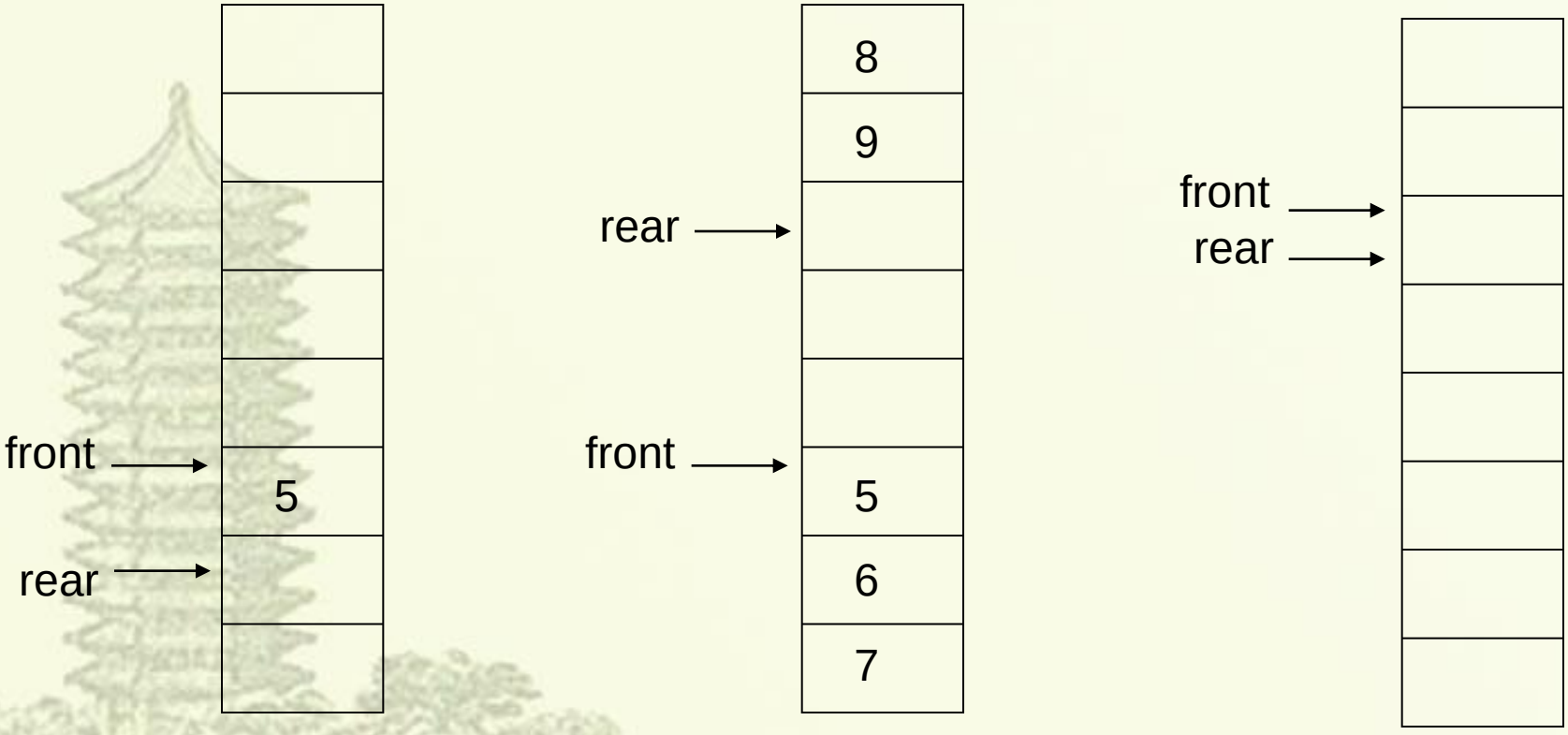
从队列前端取出/删除

```
bool arrQueue<T> :: deQueue(T& item) {
                                        // 返回队头元素并从队列中删除
    if ( front == rear) {
        cout << "队列为空" << endl;
        return false;
    }
    item = qu[front];
    front = (front +1) % mSize;
    return true;
}
```

队列的运行示意图

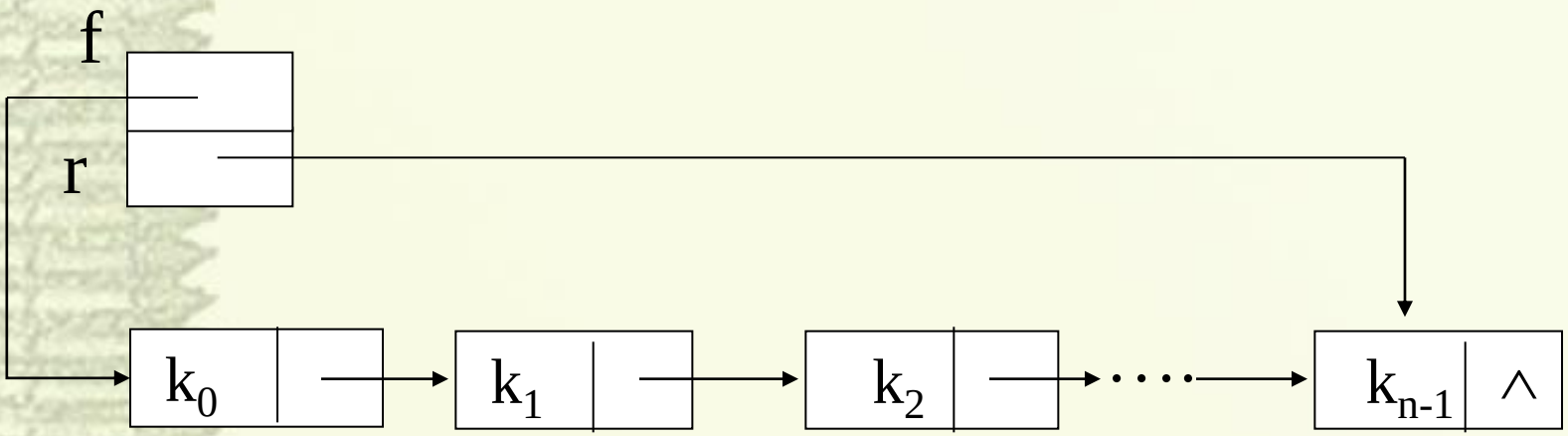


队列的运行示意图



3.2.3 链式队列

- 单链表队列
- 链接指针的方向是从队列的前端向尾端链接



链式队列的类定义

```
template <class T>
class lnkQueue: public Queue <T> {
private:
    int    size;              // 队列中当前元素的个数
    Link<T>* front;           // 表示队头的指针
    Link<T>* rear;            // 表示队尾的指针
public:                                  // 队列的运算集
    lnkQueue(int size) {                        // 创建队列的实例
        size = 0;
        front = rear = NULL;
    }
    ~lnkQueue() {                        // 消除该实例，并释放其空间
        clear();
    }
}
```

链式队列的插入

```
// item入队，插入队尾
bool lnkQueue <T> :: enQueue(const T item) {
        if (rear == NULL)       {       // 空队列
                front = rear = new Link<T> (item, NULL);
        }
        else {                          // 添加新的元素
                rear-> next = new Link<T> (item, NULL);
                rear = rear ->next;
        }
        size++;
        return true;
  }
```

链式队列的删除

```
// 返回队头元素并从队列中删除
bool lnkQueue <T> :: deQueue(T& item) {
        Link<T> *tmp;
        if (size == 0) {            // 队列为空，没有元素可出队
                cout << "队列为空" << endl;
                return false;
        }
        &item = front->data;
        tmp = front;
        front = front -> next;
        delete tmp;
        if (front == NULL)
            rear = NULL;
        size--;
        return true;
}
```

顺序队列与链式队列的比较

- 顺序队列
  - 固定的存储空间
  - 方便访问队列内部元素
- 链式队列
  - 可以满足浪涌大小无法估计的情况
  - 访问队列内部元素不方便

队列的应用

- 只要满足先来先服务特性的应用均可采用队列作为其数据组织方式或中间数据结构。
- 调度或缓冲
  - 消息缓冲器
  - 邮件缓冲器
  - 计算机的硬设备之间的通信也需要队列作为数据缓冲
  - 操作系统的资源管理
- 宽度优先搜索

变种的栈或队列结构

- 双端队列
- 双栈
- 超队列
- 超栈

小结

- 栈
  - 栈的特点
  - 栈的实现
  - 栈的应用
- 队列
  - 队列的特点
  - 队列的实现
  - 队列的应用

The End