中国科学技术大学
University of Science and Technology of China

数据结构

栈和队列（5 学时）

September 28, 2020

目录

栈和队列的简介

操作受限的线性表

- 线性表的插入和删除操作可以在任意位置进行
- 若将基本操作中的插入和删除加以“限制”，只能在特殊的位置进行，那么就得到了“栈和队列”

栈/stack

- 是限定仅在表尾进行插入或删除操作的线性表。

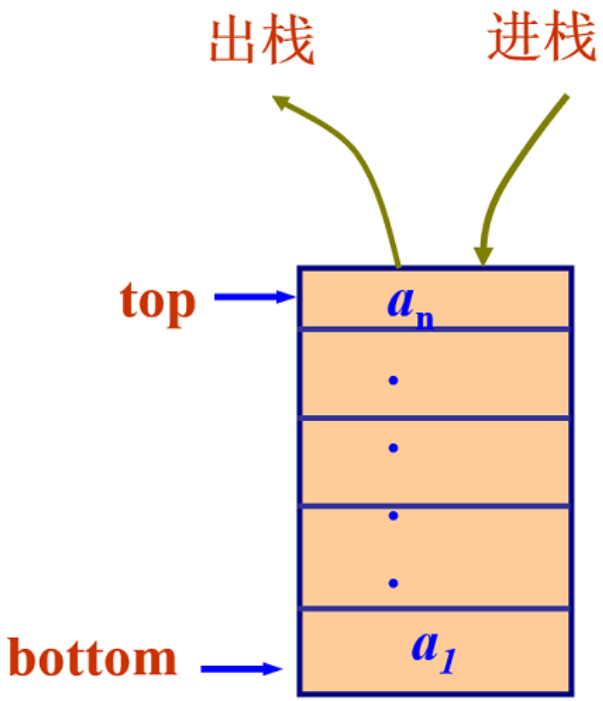
队列/queue

- 只允许在表的一端进行插入，而在另一端删除元素的线性表。

栈

定义与特点

- 允许插入和删除的一端称为栈顶 (top)，另一端称为栈底 (bottom)
- 后进先出 (LIFO)



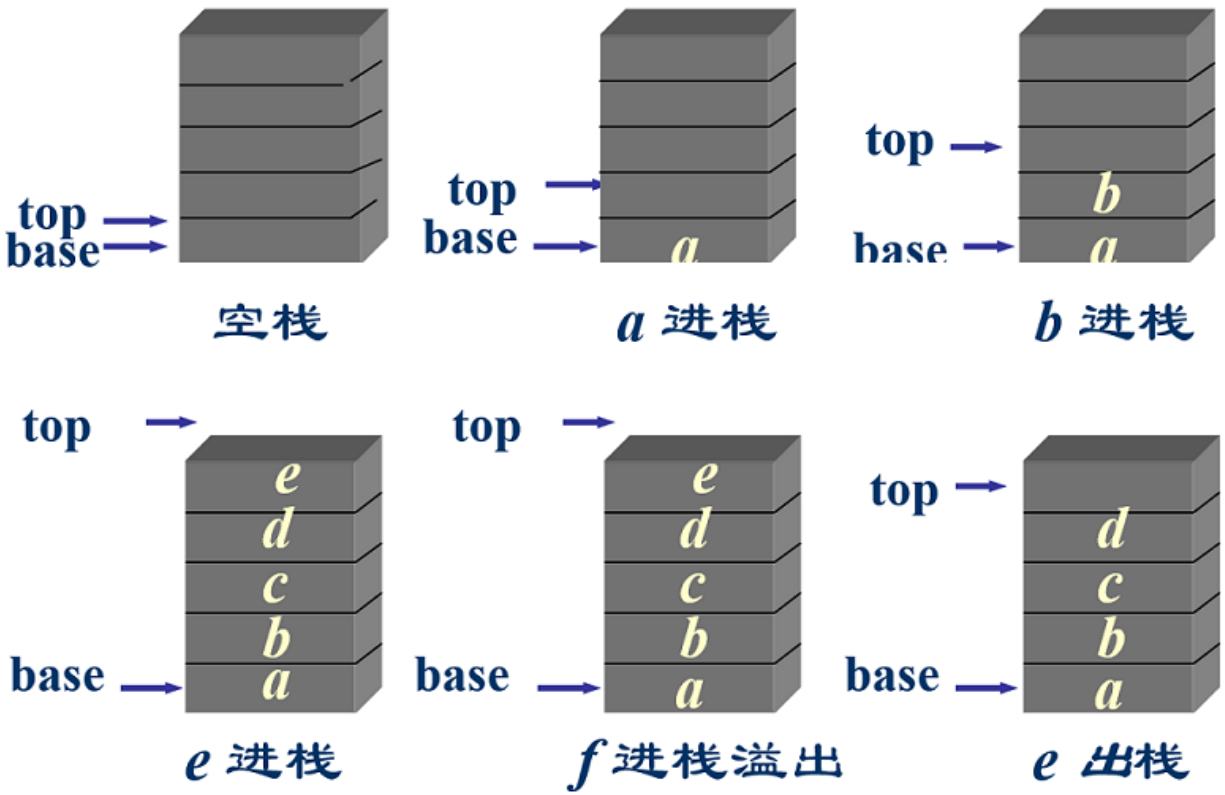
栈的 ADT

```
ADT Stack {
  数据对象: D = {a_i | a_i ∈ ElemSet, i = 1, 2, ..., n, n ≥ 0}
  数据关系: R1 = {< a_{i-1}, a_i > | a_{i-1}, a_i ∈ D, i = 2, ..., n}
            约定a_n, a_1的位置分别为栈顶端和栈底端
  基本操作:
     InitStack(stack &S);          //栈的初始化
     DestroyStack(&S) ;            //栈的销毁
     ClearStack(&S);               //清空栈
     StackEmpty(S);                //判栈空否
     StackLength(S);               //取栈长
     GetTop(S,&e);                 //取栈顶元素
     Push(&S,e);                   //进栈
     Pop(&S,&e);                   //出栈
     StackTraverse(S,visit());
 }ADT Stack
```

顺序栈

存储结构与操作示例

- 栈的顺序存储结构，利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素。
- 指针 top 指向栈顶元素的下一个位置，base 为栈底指针，指向栈底的位置 (栈的起始地址)。



顺序栈的实现

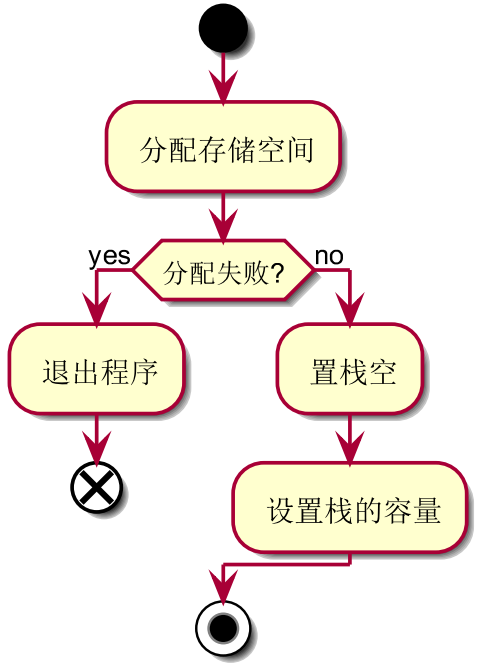
在内存中的表示：静态部分

```
#define STACK_INIT_SIZE  100
#define STACKINCREMENT  10

typedef char SElemType;

typedef struct {//顺序栈定义
    SElemType *base; //栈底指针
    SElemType *top; //栈顶指针
    int stacksize;  //当前已分配的存储空间
} SqStack;
```

顺序栈的实现

基本操作的实现：初始化

```
Status InitStack ( SqStack  &S) {   //置空栈
 S.base = ( SElemType *)
       malloc (STACK_INIT_SIZE * sizeof(SElemType));
 if (!S.base) exit(OVERFLOW);
      S.top = S.base ;
     S.stacksize= STACK_INIT_SIZE ;
     return OK;
}
```



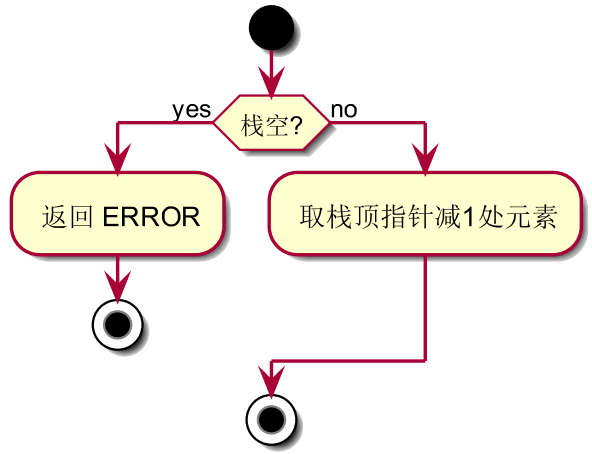
基本操作的实现：判定栈空

```
Status StackEmpty (SqStack  S) {

   if( S.top == S.base )
      return TRUE      //判栈空,空则返回1
   else return FALSE; //否则返回0
}
```

顺序栈的实现

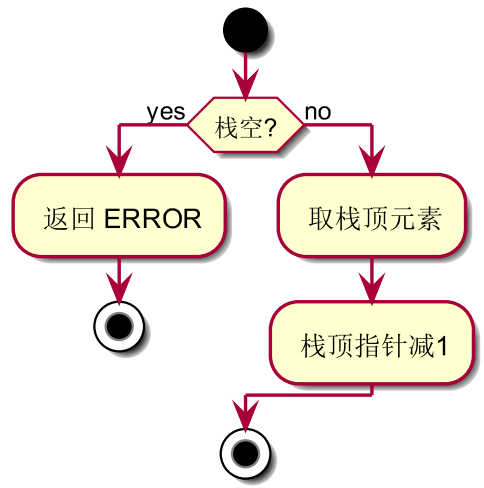
基本操作的实现：取栈顶元素

```
Status GetTop (SqStack S, SElemType &e) {
  //若栈空返回0, 否则栈顶元素读到e并返回1
   if ( S.top==S.base ) return ERROR;

   e=*(S.top-1);

  return OK;
}
```



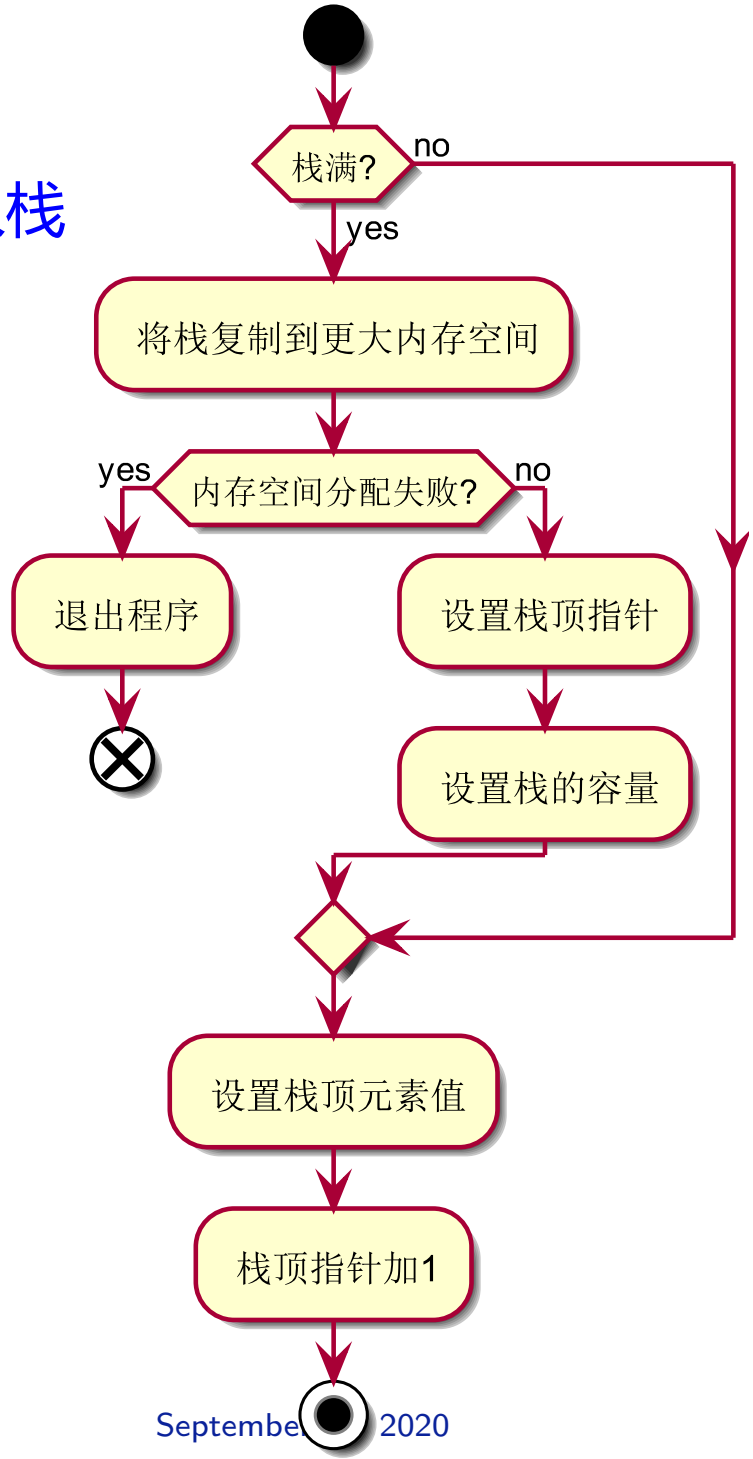
基本操作的实现：出栈

```
Status Pop (SqStack &S, SElemType &e) {
//若栈空返回ERROR, 否则栈顶元素退出到e并返回OK
   if ( S.top == S.base ) return ERROR;
   e = * --S.top;
   return OK;
}
```



顺序栈的实现

基本操作的实现：入栈



顺序栈的实现

基本操作的实现：入栈

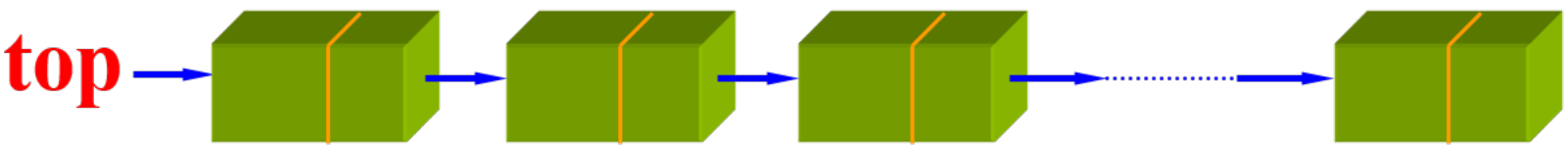
```
Status Push (SqStack *S, SElemType e) {
  //插入元素x为新的栈顶元素
    if (S->top-S->base>=S->stacksize){
      S->base =( StackData *) realloc(S->base ,
              (S->stacksize+ STACKINCREMENT) *
              sizeof(SElemType));
      if(! S->base)exit(OVERFLOW);
      S->top= S->base + S->stacksize;
     S->stacksize+= STACKINCREMENT; //追加存储空间
    }
    *(S->top)=e;
     （S->top）++;
     return OK;
}
```

- 入栈操作中的 realloc() 函数的说明：realloc()
  https://baike.baidu.com/item/realloc

链式栈

存储结构

- 链栈无栈满问题，空间可扩充
- 插入与删除仅在栈顶处执行
- 链栈的栈顶在链头
- 适合于多栈操作



链式栈的实现

链式栈在内存中的表示：静态部分

```
typedef int StackData;
typedef struct node {
    StackData data;              //结点
    struct node *link;           //链指针
} StackNode;
typedef struct {
    StackNode *top;             //栈顶指针
} LinkStack;
```

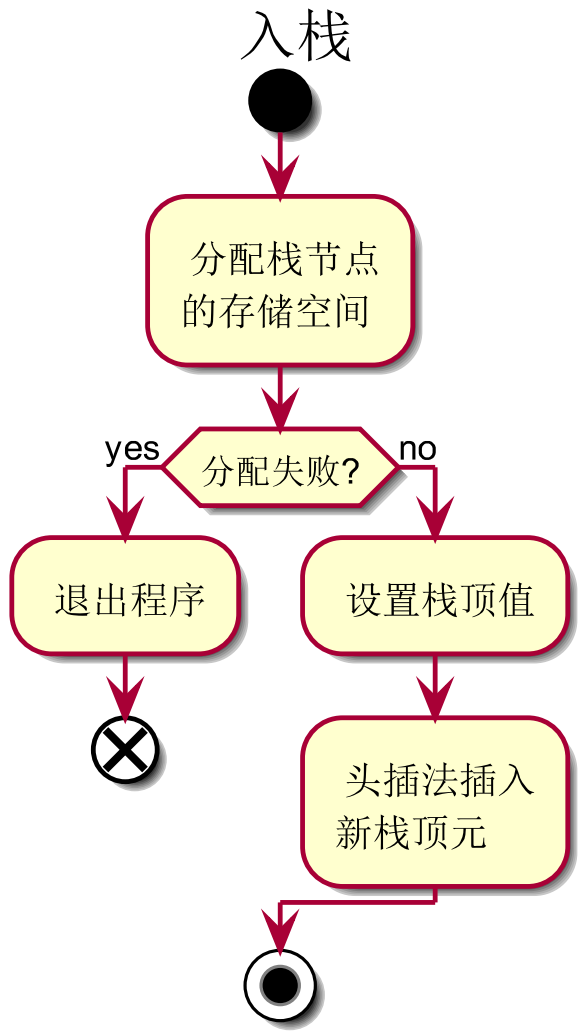
链式栈的实现

链式栈基本操作的实现：初始化

```
Status InitStack ( LinkStack &S ) {
    S.top = NULL;
}
```

链式栈基本操作的实现：入栈

```
Staus Push ( LinkStack &S, StackData e ) {
    StackNode *p = ( StackNode * ) malloc
        ( sizeof ( StackNode ) );
    p->data = e;  p->link = S.top;
    S.top = p;  return OK;
}
```



(代码问题：没有检查空间分配是否成功)

链式栈的实现

链式栈基本操作的实现：判栈空

```
int StackEmpty (LinkStack &S) {
    return S.top == NULL;
}
```

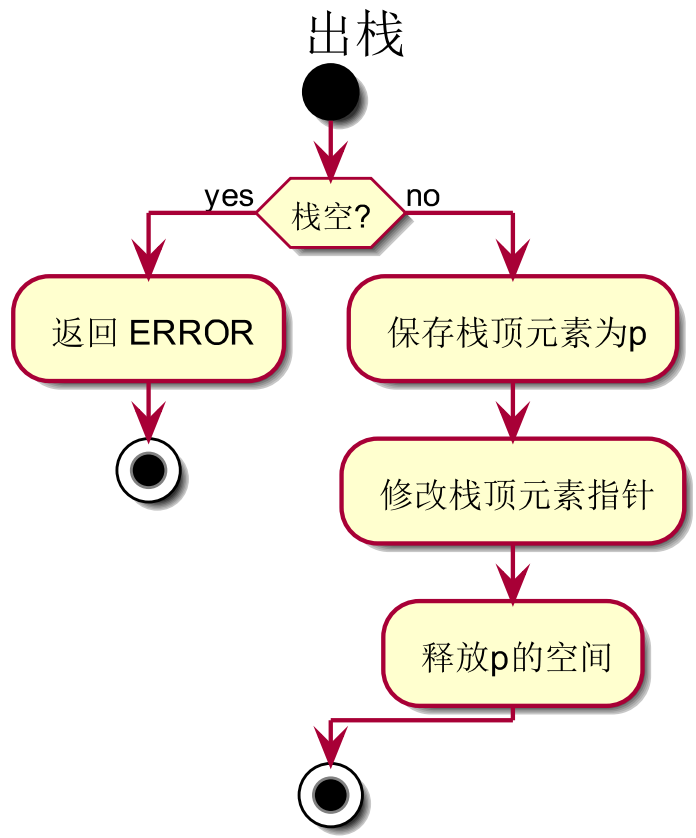
链式栈基本操作的实现：取栈顶元素

```
Status GetTop ( LinkStack S, StackData &e ) {
    if ( StackEmpty (S) ) return ERROR;
    e = S.top->data;
    return OK;
}
```

链式栈的实现

链式栈基本操作的实现：出栈

```
int Pop ( LinkStack &S, StackData &e ) {
    if ( StackEmpty (S) ) return ERROR;
    StackNode * p = S.top;
    S.top = p->link;
    e = p->data;
    free (p);
    return OK;
}
```



栈的应用例子

问题列表

- 数制转换
- 行编辑器
- 检测括号匹配
- 表达式求值
- 函数调用与递归

数制转换

问题描述

- 输入 k 进制数 $(a_n \ldots a_1 a_0)_k$
- 输出 s 进制数 $(b_m \ldots b_1 b_0)_s$
- 十进制数 $(a_n \ldots a_1 a_0)_{10} = a_n \times 10^n + \times \ldots \times a_1 \times 10^1 + a_0 \times 10^0$
- k 进制数 $(a_n \ldots a_1 a_0)_k = a_n \times k^n + \times \ldots \times a_1 \times k^1 + a_0 \times k^0$
- 如果上述式子右侧的加法统一采用十进制加法，那么同一个数的两个不同进制的表示展开计算的结果相同
- 例子 $(1348)_{10} = (2504)_8$

数制转换的运算过程

如 $(1348)_{10} = (2504)_8$，运算过程为：

N	N div 8	N mod 8
1348	168	4
168	21	0
21	2	5
2	0	2

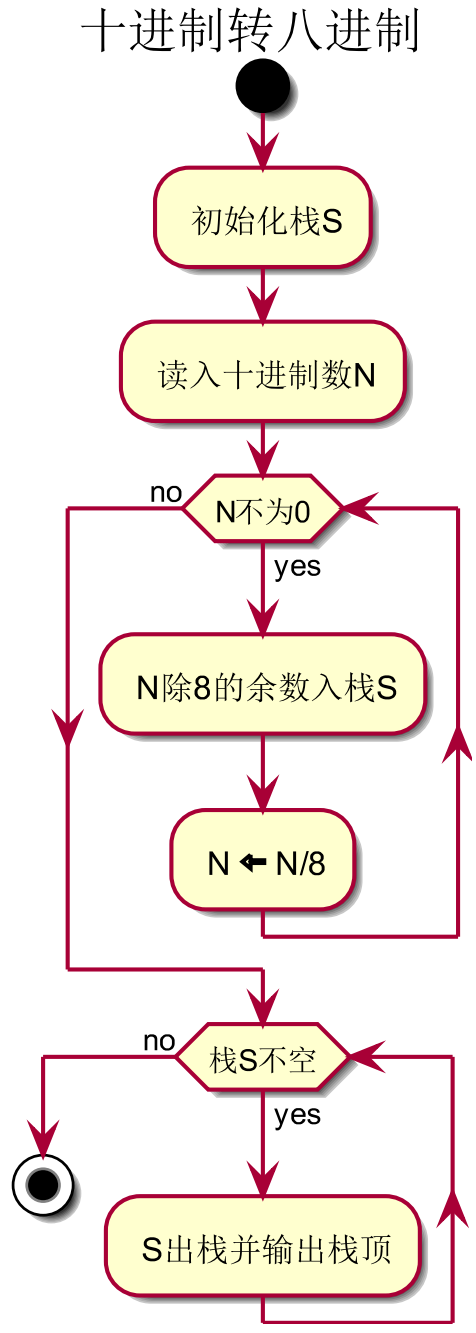
$(1348)_{10} = (2504)_8$

- 按照上述自上而下的除 8 求余的运算过程，先输出个位 4，再输出 0，5，2
  - 在屏幕打印输出的结果（水平方向上，自左向右）依次是 $a_0\, a_1 \ldots a_n$，与我们书写习惯 $a_n \ldots a_1\, a_0$ 不符合
  - 解决办法 1：把 a_i 存储在线性表中，然后逆序打印线性表即可
  - 解决办法 2：先依次计算 a_i 并存入栈中，然后依次出栈输出即可

数制转换

代码实现

```
1       void conversion( ){
2         InitStack(S); //初始化一个栈
3         scanf("%d",N); //转换并保存余数到栈内
4         while (N) {
5           Push(S,N%8);
6           N=N/8;
7         }
8         while (!StackEmpty(S)){
9           Pop(S,e); //出栈，实现余数逆序
10          printf("%d",e);
11        }
12      }
```



利用 stack 的 LIFO 性质来得到逆序序列!

行编辑器

功能解释

- 早期的电脑编辑器，屏幕就如一张纸，打印在屏幕上的字符无法删除，连光标后退也不行。编辑文字的过程中如果出错，怎么办?
- 行编辑器不同与写字板、记事本、ms-word 等全屏编辑器，一次只能编辑一行中的文字
- 输入字符'#' 表示删除上一个字符，输入字符'@' 表示删除整行，输入'\n' 表示一行输入结束
- 例子：foawe32##fe@fw9# $\Longrightarrow$ fw

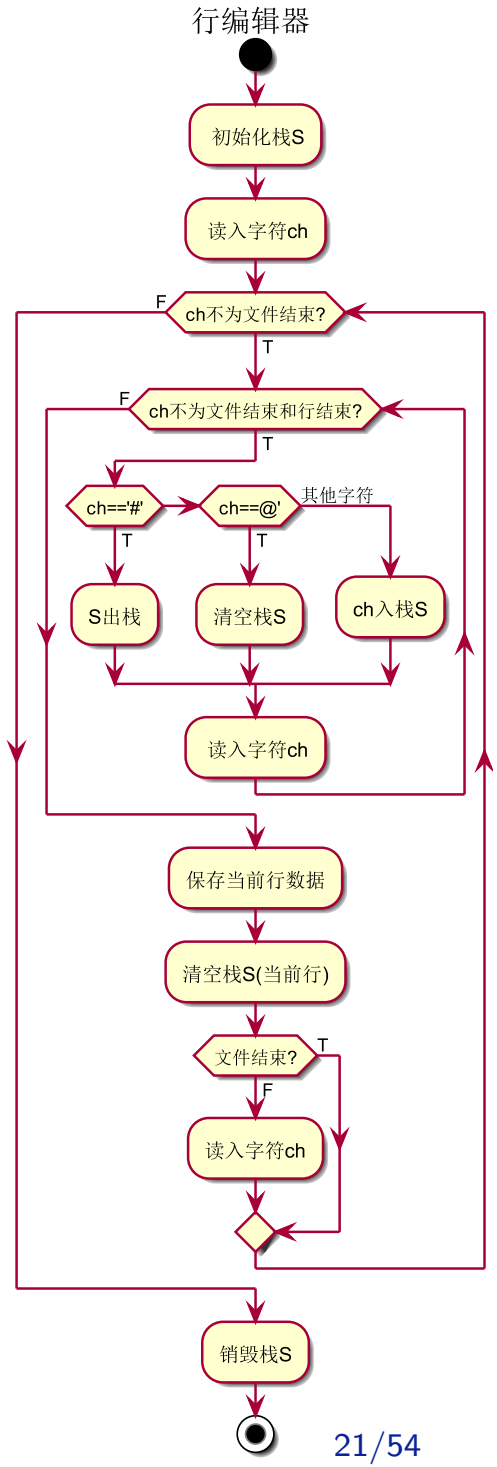
实现要点

- 用一个栈 S 作为输入缓冲区；栈中存储当前行的字符
- 输入字符用 push 依次压入栈中
- 编辑时，删除前一字符用 Pop 实现；删除全行用 ClearStack 实现

行编辑器

代码实现

```
1       Void LineEdit(){
2         InitStack(S);
3         ch = getchar();
4         while (ch!=EOF){
5           while (ch!=EOF&&ch!= '\n' ){
6             switch(ch){
7               case '#': Pop(S,c);break;
8               case '@': ClearStack(S);break;
9               default: Push(S,ch);
10            }
11            ch = getchar();
12          }//行结束或编辑结束
13          //将栈内所有字符传送至用户数据区;
14          ClearStack(S);
15          if (ch!=EOF) ch=getchar();
16        }//编辑结束
17        DestroyStack(S);
18      }
```



检测括号匹配

问题描述

- 编写程序或文档过程中有可能会发生括号、引号等多或少的问题，造成不匹配。
- 利用栈实现括号是否匹配的检测。
- 例子：
  - 匹配的：([])(())([])
  - 未匹配的：[](()])

检测括号匹配

求解思想

- 定义括号栈，初始化为空，算法结束时，栈也应为空，否则输出“未匹配”
- 处理括号字符串中的每个字符
  - 遇到左括号，入栈；
  - 遇到右括号，判定是否和栈顶括号匹配，若不匹配，算法输出“未匹配”并结束程序；否则出栈；（栈空也属于不匹配）
  - 跳到处理下一个字符

思考

- 编写伪代码实现检测括号匹配
- 如果程序中有这样的代码，“printf(“%c” ,'}')”，如何实现检测括号匹配？（用一个标识变量，初值为 F，遇到单引号就翻转（T-F 互换），当其为 T 时，所有非单引号输入字符都忽略；）

表达式求值

问题描述

- 输入算数四则运算表达式，给出计算结果
- 算符优先法: 根据表达式中运算符的优先关系来实现对表达式的编译或解释执行。
- $4 + 2 \times 3 - 10/5 = 4 + 6 - 10/5 = 10 - 10/5 = 10 - 2 = 8$
- 如下：算符优先关系的定义（教材表 3.1)

表 3.1　算符间的优先关系

θ_1 \ θ_2	+	−	*	/	(	)	#
+	>	>	<	<	<	>	>
−	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	≐	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

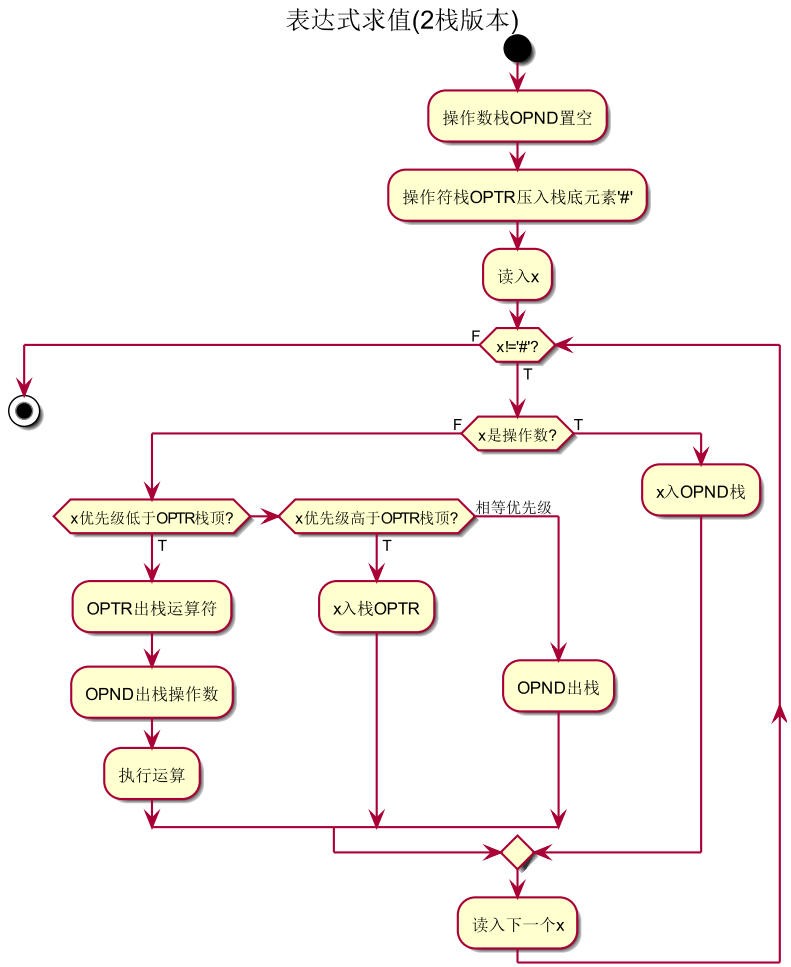
表达式求值

内存描述部分：用两个工作栈保存表达式

- 运算符栈 OPTR，字符栈
- 操作数栈 OPND，数值栈

算法思想

1. 置操作数栈 OPND 为空栈；运算符栈 OPTR 中压入栈底元素：表达式起始符“#”；(结束也用“#”标记)
2. 依次读入表达式中每个字符，若是操作数，则入 OPND(等待判断处理)；若是运算符，则和 OPTR 栈顶元素比较优先级，做相应处理：
   a. 当前运算符比栈顶运算符优先级低，则处理栈顶运算符；
   b. 当前运算符比栈顶运算符优先级高，入 OPTR 栈继续“等待时机”；
   c. 当前运算符比栈顶运算符优先级相等，消配对括号。



表达式求值

进一步理解和思考

- 结合教材例 3-1 理解算法过程
- 练习：尝试实现字符栈和数值栈的 ADT，然后利用栈的基本操作实现表达式求值。参考教材的伪代码算法 3.4。

表达式求值

第二种实现方法：利用表达式的逆波兰式

- 想法：不存储操作符或不要操作符栈，只保留操作数栈，遇到操作符就从操作数栈中出栈两个操作数
- 存在问题：按照通常的表达式的输入次序，不能实现这个功能，比如：4*2+5+6*3，遇到乘号时，操作数栈中操作数只有一个 4，因此需要将表达式改成：4 2 * 5 + 6 3 * + 输入

逆波兰式

- 普通正常形式的表达式叫“中缀表达式”，逆波兰式也叫做“后缀表达式”，简单理解就是运算符放在操作数中间和放在操作数之后的区别
- 给定一个中缀表达式，如何获得逆波兰式?
- 复杂例子的逆波兰式：1+4*2+(5+6)*3 $\Rightarrow$ 1 4 2 * + 5 6 + 3 * + (注意到括号没了)

表达式求值

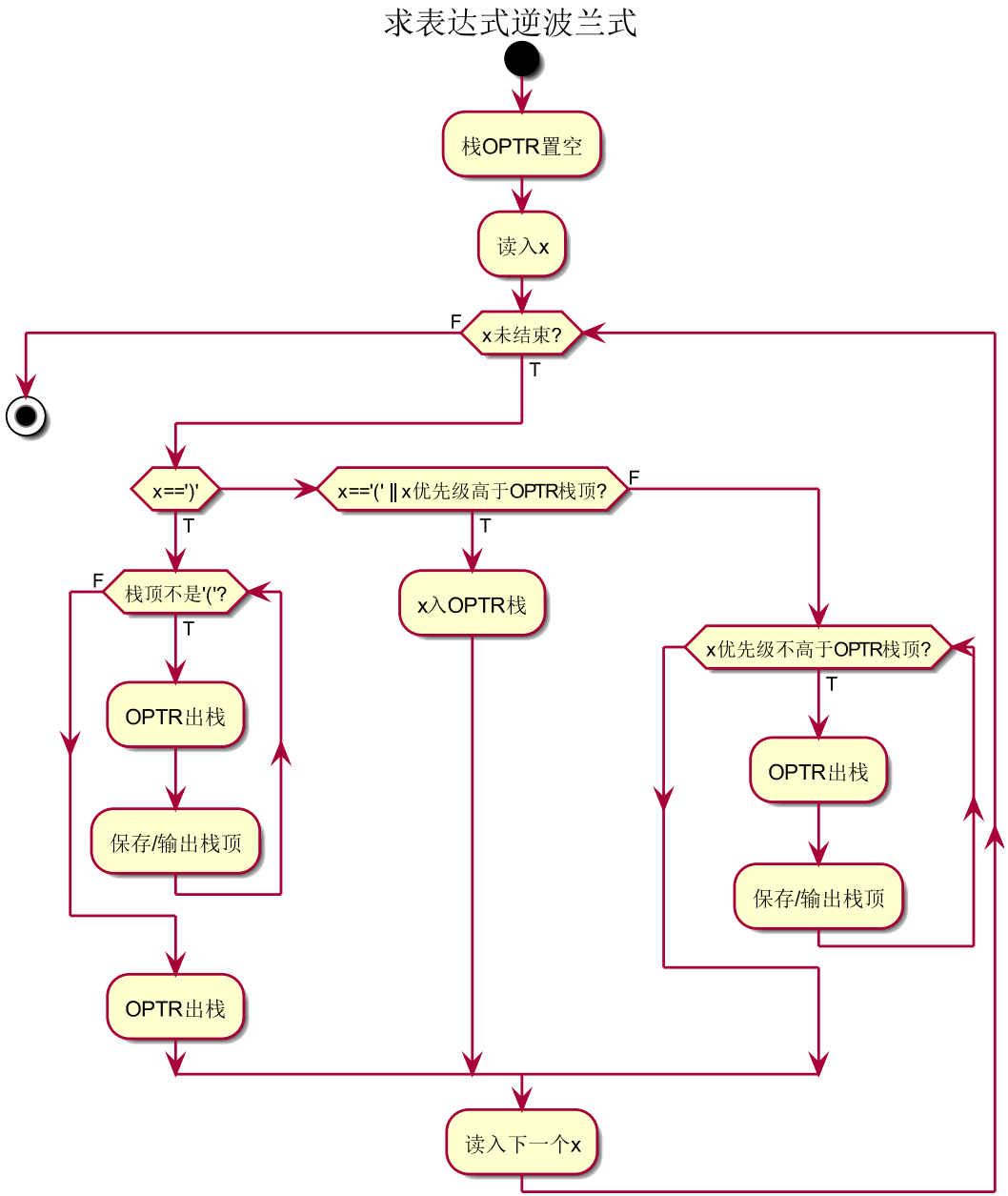
求表达式的逆波兰式：利用栈

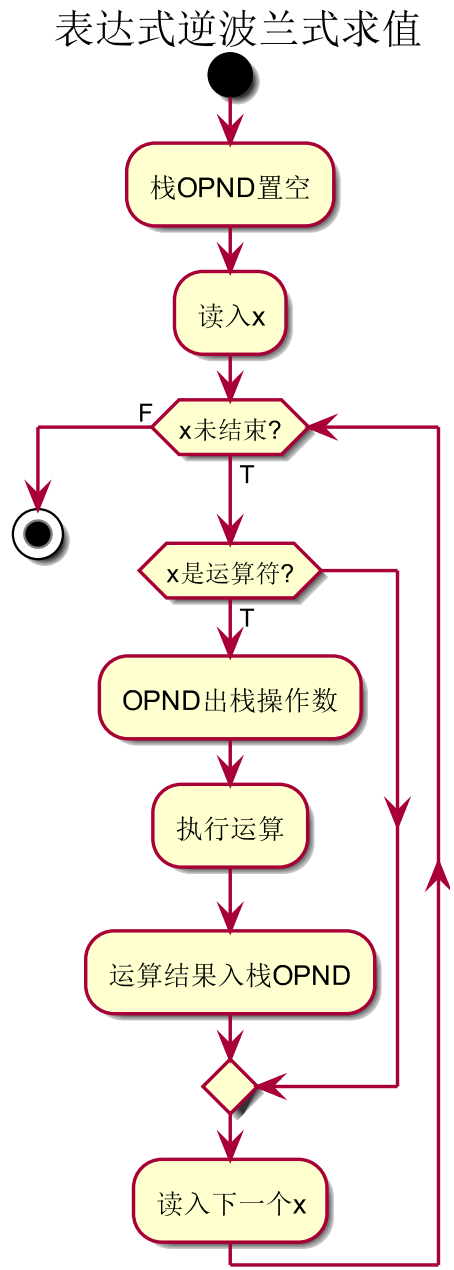
- 用一个栈来实现，初始化空栈；
- 遇到操作数，直接输出操作数；
- 遇到操作符，比较栈顶操作符，栈顶优先级低，操作符入栈；栈顶优先级不低于当前操作符，则栈顶出栈并输出，继续比较当前操作符和新栈顶；类似处理；
- 遇左括号，其优先级最低，入栈；遇右括号，出栈，直至左括号出栈，左右括号不输出。

思考

- 采用逆波兰式对表达式求值和 2 栈表达式求值方法，二者的优劣？
- 同时用两个栈，串行两次用不同的栈

表达式求值：中缀式 ⇒ 后缀式 ⇒ 求值





表达式求值：中缀式 ⇒ 后缀式 ⇒ 求值

中缀式：1+4*2+(5+6)*3 ⇒ 后缀式

- 1 输出，+ 入栈，4 输出,* 入栈，2 输出，* 出栈输出，+ 出栈输出，+ 入栈，(入栈，5 输出，+ 入栈，6 输出，+ 出栈输出，(出栈，* 入栈，3 输出，* 出栈输出，+ 出栈输出，最后得到：1 4 2 * + 5 6 + 3 * +

后缀式 ⇒ 求值

- 1 入栈，4 入栈，2 入栈，4 和 2 出栈，* 运算得 8 入栈，8 和 1 出栈，+ 运算得 9 入栈，5 入栈，6 入栈，6 和 5 出栈，+ 运算的 11 入栈，3 入栈，3 和 11 出栈，* 运算得 33 入栈，33 和 9 出栈，+ 运算的 42

函数调用

基础知识

- 程序的执行过程可看作连续的函数调用。当一个函数执行完毕时，程序要回到调用指令的下一条指令 (紧接函数调用指令) 处继续执行。
- 函数调用过程通常使用栈实现，每个用户态进程对应一个调用栈结构。
- 用堆栈传递函数参数、保存返回地址、临时保存寄存器原有值 (即函数调用的上下文) 以备恢复以及存储本地局部变量；存储的次序不同语言、编译器等略有区别；
- 栈的每个数据元素称为栈帧、工作记录或活动记录

内存片段：栈帧内容

函数调用

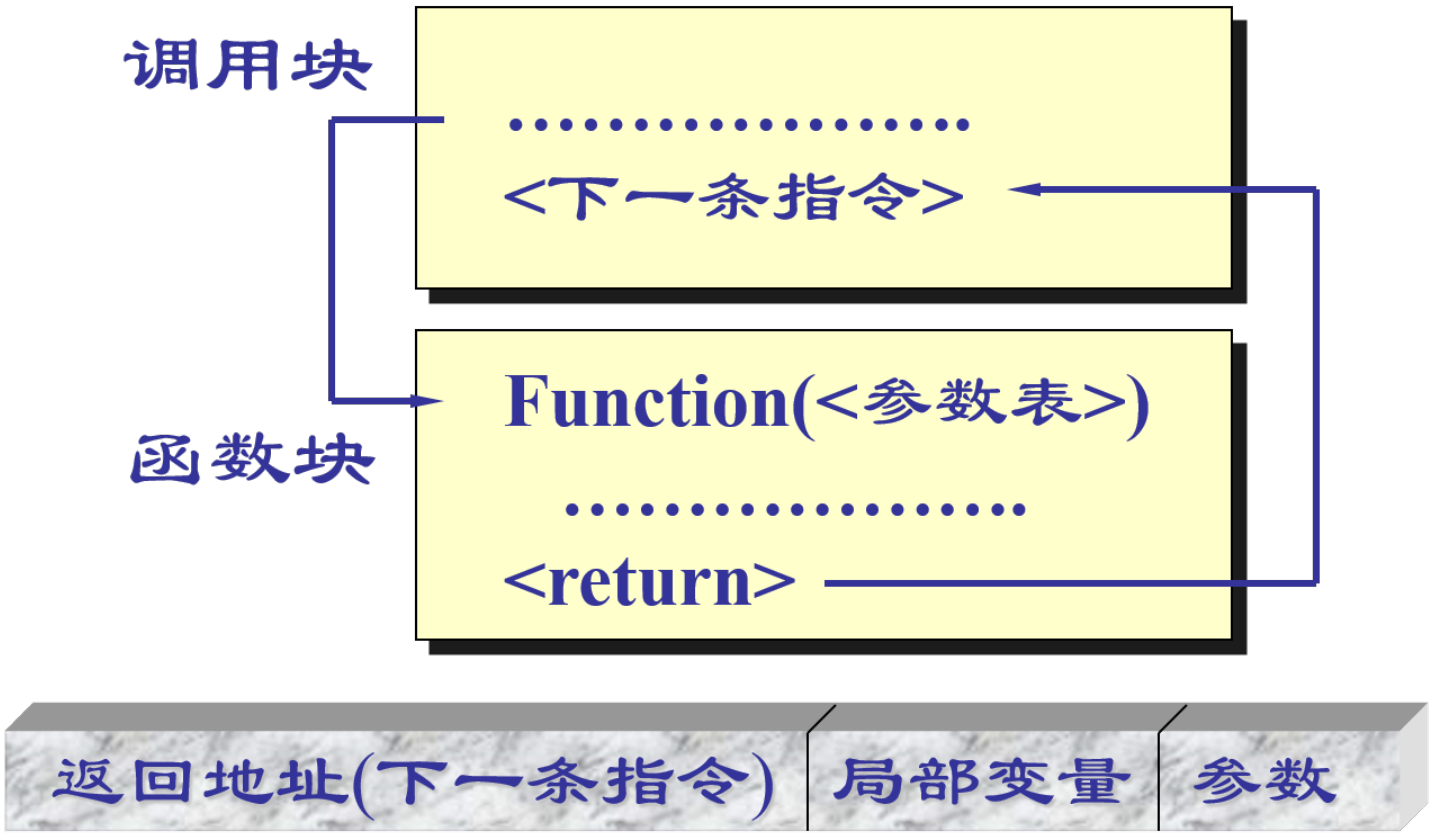
函数调用的主要过程

1. 保存主调用函数的运行状态和返回地址，将其栈帧入栈内；包括将被调函数实参“原件”放在栈帧中入栈，制作副本以便顶替形参来参加被调函数的运行；同时将各寄存器内容也放在栈帧中入栈
2. 执行被调用函数的函数体内的语句
3. 将各寄存器内容弹出栈，恢复主调用函数的程序的运行状态，从而释放被占栈空间
4. 按照返回地址将控制权交还给主调用函数

栈保证调函数的嵌套用过程和恢复过程不会出现混乱

函数调用

图示



函数调用特例：递归

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；
- 一个函数直接或间接调用自身，则称这个函数是递归的

各种不同类型的递归

- 定义是递归的，如阶乘函数 $n! = n * (n-1)!$
- 数据结构是递归的，如单链表的
- 算法是递归的

定义是递归的

阶乘函数

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

递归算法求阶乘：时空性能?

```
1  long Factorial(long n){
2    if (n==0)
3      return 1;
4    else
5      return n*Factorial(n-1);
6  }
```

阶乘函数的求解过程


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
栈底
栈顶
递归调用
参数传递
结果返回
回归求值
依次保存现场，入栈
依次出栈

数据结构是递归的

单链表

```
1  typedef struct Lnode { //链表结点
2    ElemType data; //结点数据域
3    struct Lnode * next;//结点链域
4  }ListNode, *LinkList;
```

递归算法求解单链表的搜索: 时空性能?

```
1  void Search(ListNode *f, ListData x){
2    if (f!=NULL)
3      if (f->data==x)
4        printf ("\%d\n", f->data);
5      else
6        Search(f->next, x);
7  }
```

算法是递归的

汉诺塔问题

- 问题描述略

递归算法求解汉诺塔问题: 时空性能?

```
1  void hanoi(int n, char X, char Y, char Z){
2    if (n==1)
3      printf("move \%s",X," to \%s",Z);
4    else {
5      Hanoi(n-1,X,Z,Y);
6      printf("move \%s",X, " to \%s",Z);
7      Hanoi(n-1, Y, X, Z);
8    }
9  }
```

问题的递归求解算法

问题能用递归求解所要具备的三个条件

1. 能将一个问题转变成一个新问题，而新问题与原问题的解法相同或类同，所不同的仅是所处理的对象，且这些处理对象的变化是有规律的
2. 可以通过上述转化使问题逐步简单化
3. 必须有一个明确的递归出口 (递归的边界)

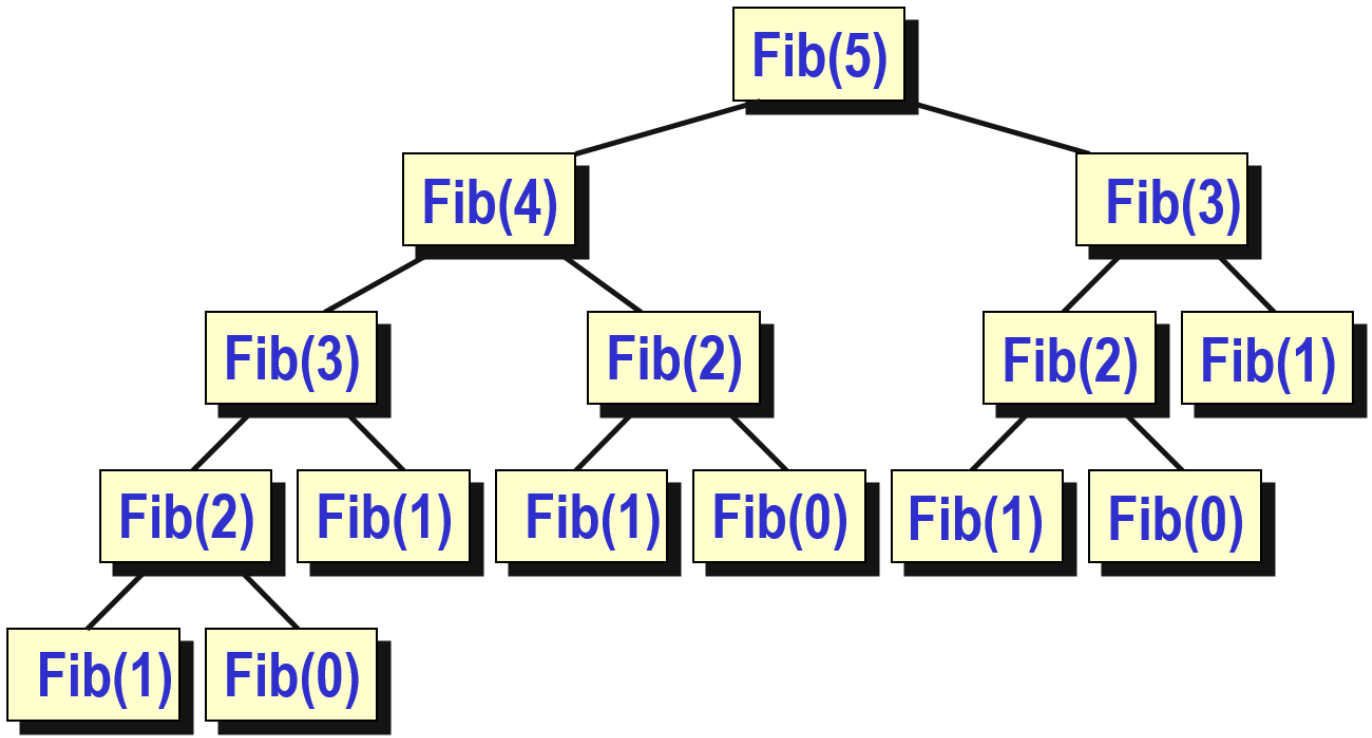
递归的优点和缺点

- 优点：递归方法是程序设计的重要方法，它使得程序的结构清晰，形式简洁，易于阅读，正确性容易证明
- 缺点：通常，性能较非递归算法差

递归的例子

递归算法的性能较差

- 函数调用的开销，包括栈帧的处理等，耗费资源；
- 大量的重复计算，如下图的递归求解斐波那契数过程



递归转换为非递归

目的：提高算法的时空性能

- 尾递归，即算法最后一句是递归调用，可直接用循环实现其非递归过程
- 尾递归的例子：求 $n!$ 和斐波那契数的递归算法
- 非尾递归：借助栈来实现非递归过程，即手动控制额外的存储空间来保存（递归调用的）栈帧中的部分信息。
  - 识别需要在函数间“传递”的信息/数据是将递归改成非递归的关键
  - 这些数据通常是递归算法的参数或返回值，这些参数会保存为栈帧中的局部变量或参数，将这些数据存储在全局栈中

可以不用栈而用线性表吗？

栈的练习题 1

证明题

- 若借助栈由输入序列 1，2，...，n 得到的输出序列为 $p_1, p_2, \ldots, p_n$(它是输入序列的一个排列)，则在输出序列中不可能出现这样的情形：
- 存在着 $i < j < k$ 使 $p_j < p_k < p_i$ 。

证明思路

- 反证法，假设上述情形可以出现，那么 p_i 是三者中最先出栈的 (因为 i 最小)，又因为 p_i 最大，根据入栈次序，在 p_i 出栈的时刻，p_j, p_k 都在栈内，..... 请补充完整

栈的练习题 2

单链表的反转

- 用递归算法实现
- 用非递归算法算法实现
- 附加要求：时间复杂度 O(n) 或空间复杂度 O(1)，能同时达到吗?

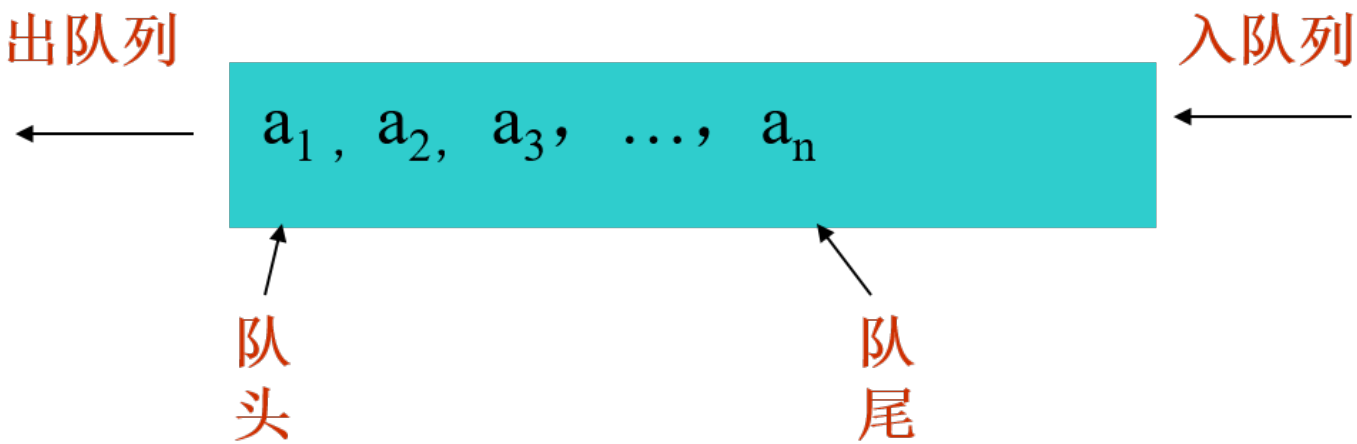
```
*Linknode revlink(H,T){
    if(H!=NULL)
H1=revlink(H->next,T1);
    T1->next=H;
    }
```

写的可能有问题，注意参数传递

队列

定义与特点

- 定义：只允许在表的一端进行插入，而在另一端删除元素的线性表。在队列中，允许插入的一端叫队尾（rear），允许删除的一端称为队头 (front)。
- 特点：先进先出 (FIFO)



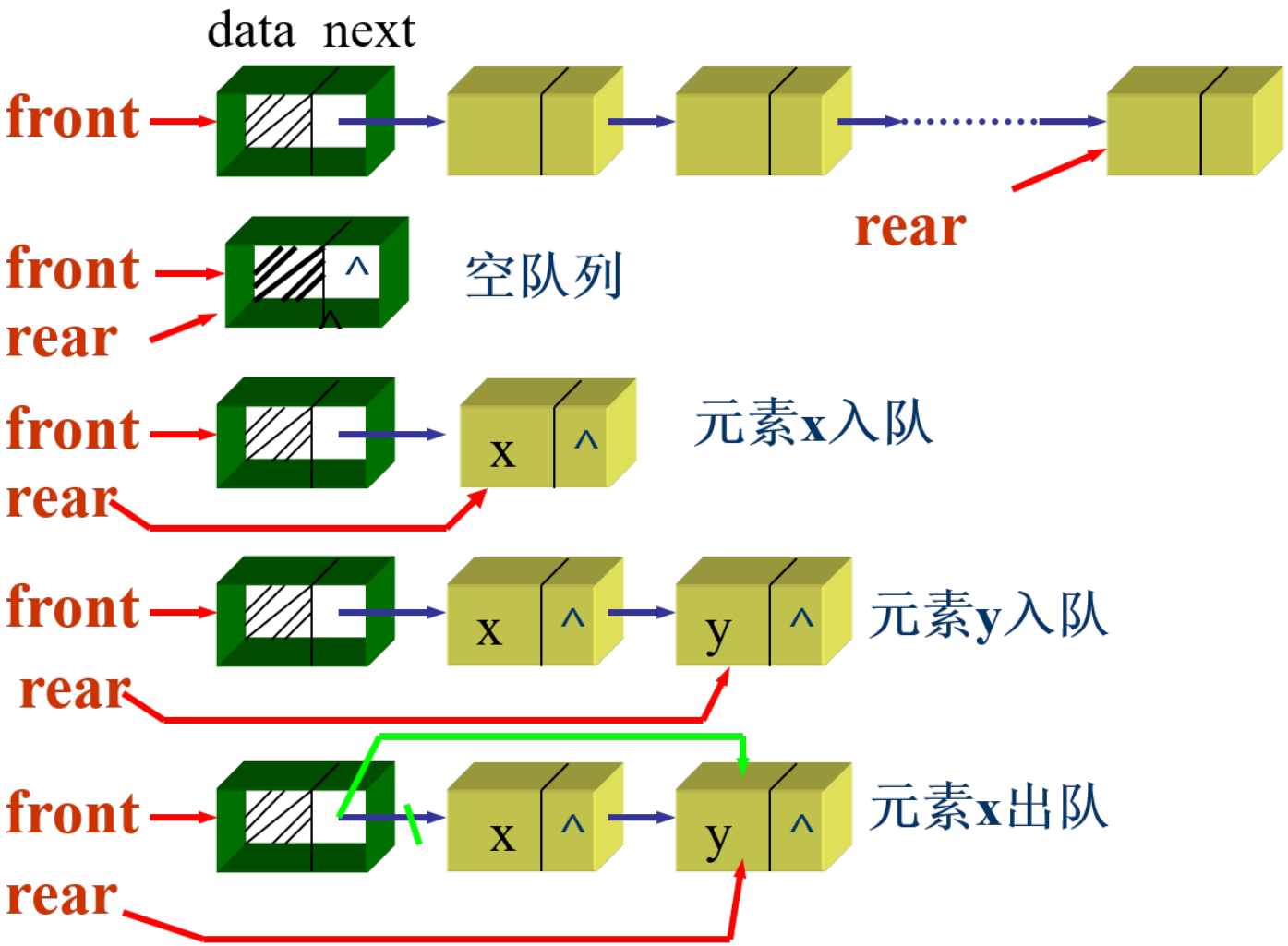
队列的 ADT

```
ADT Queue{
  数据对象: D = {a_i | a_i ∈ ElemSet, i = 1, 2, ..., n, n ≥ 0}
  数据关系: R1 = {< a_{i-1}, a_i > | a_{i-1}, a_i ∈ D, i = 2, ..., n}
            约定a_n, a_1的位置分别为队尾端和对头端
  基本操作:
     InitQueue(Queue &Q);          //队列的初始化
     DestroyQueue(&Q) ;            //队列的销毁
     ClearQueue(&Q);               //清空队列
     QueueEmpty(Q);                //判队列空否
     QueueLength(Q);               //取队列长
     GetHead(Q,&e);                //取队列顶元素
     EnQueue(&Q,e);                //进队列
     DeQueue(&Q,&e);               //出队列
     QueueTraverse(S,visit());
 }ADT Queue
```

链队列

存储结构与操作示例

- 如图，有两个分别指示队头和队尾的指针，链式队列在进队时无队满问题，但有队空问题。



链队列的实现

链队列在内存中的表示：静态部分

```
1  typedef int QElemType;    //整数队列
2
3  typedef struct Qnode{
4     QElemType data;        //队列结点数据
5     struct Qnode *next;    //结点链指针
6  }Qnode,*QueuePtr;
7
8  typedef struct{
9    QueuePtr rear, front;
10 }LinkQueue;
```

链队列的实现

基本操作的实现：入队

```
1  Status EnQueue(LinkQueue &Q, QElemType e){
2      p =(QueuePtr) malloc(sizeof(QNode));
3      ···
4      p->data=e; p->next=NULL;
5      Q.rear->next = p;                    //入队
6      Q.rear =p;
7      return OK;
8  }
```

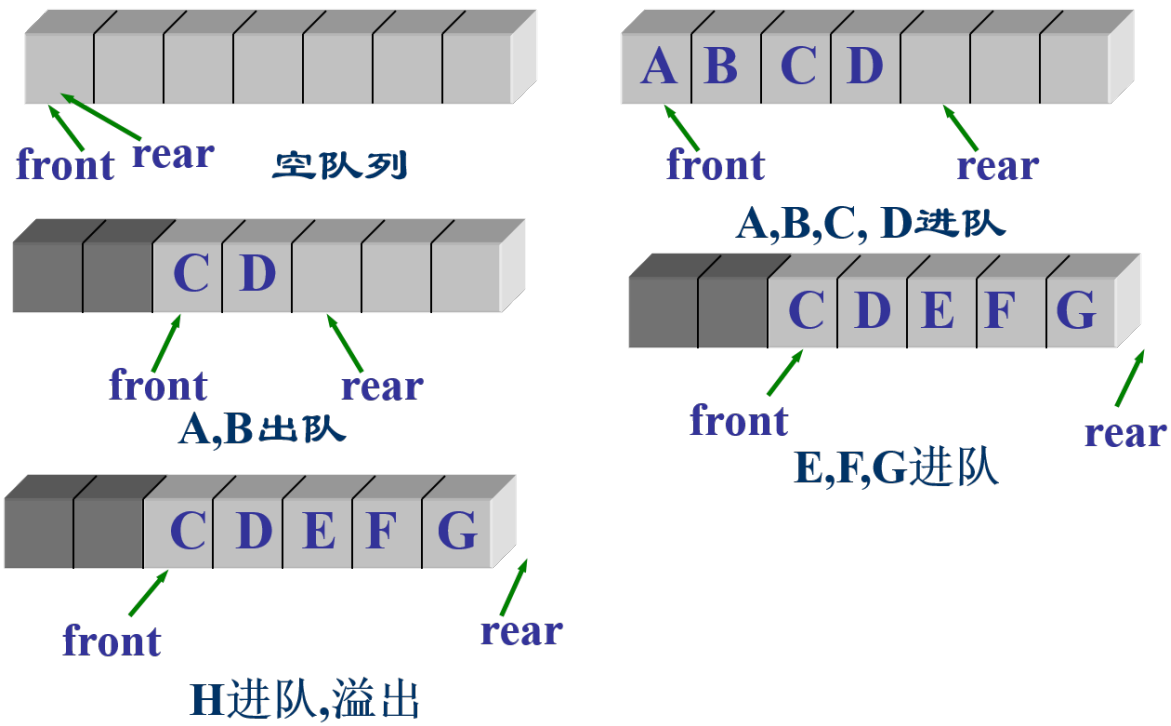
基本操作的实现：出队（删去队头结点，并返回队头元素的值）

```
1  int DeQueue(LinkQueue &Q, QElemType &e){
2      if (Q.front==Q.rear) return ERROR; //判队空
3      p=Q.front ->next;
4      e=p->data;                           //保存队头的值
5      Q.front->next=p->next;               //新队头
6      if (Q.rear==p) Q.rear=Q.front ; //防止只有一个节点会把rear指针删去
7      free(p);
8      return OK;
9  }
```

顺序队列

存储结构

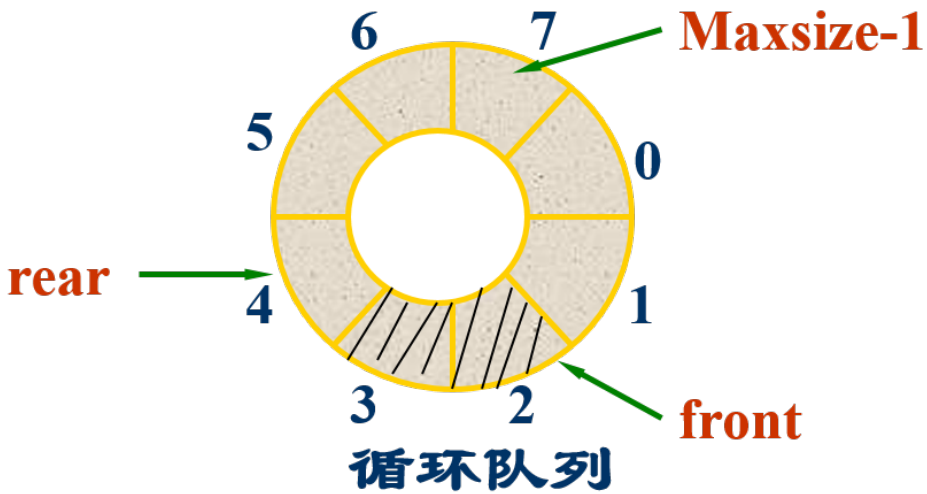
- 队列的顺序存储表示，插入新的队尾元素，尾指针增 1，$\text{rear} = \text{rear} + 1$，删除队头元素，头指针增 1，$\text{front} = \text{front} + 1$
- 在非空队列中，头指针始终指向队列头元素，而尾指针始终指向队列尾元素的下一个位置
- 队满时再进队将溢出，会有“假溢出”问题，如图



循环队列

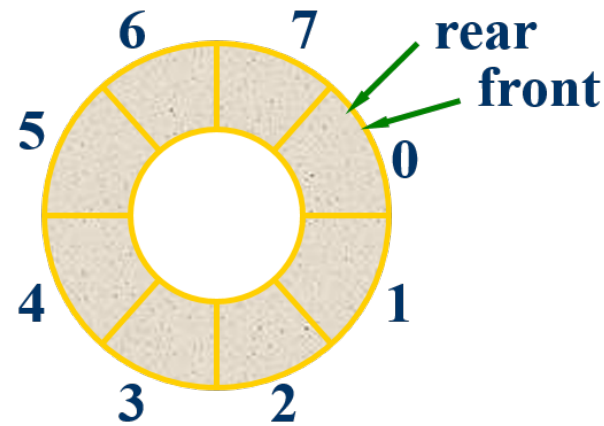
解决“假溢出”问题

- 队头、队尾指针加 1，可用取模 (余数) 运算实现
- 队头指针进 1: front = (front+1) %maxsize;
- 队尾指针进 1: rear = (rear+1) % maxsize
- 队列初始化：front = rear = 0;
- 队空条件：front == rear;
- 队满条件：(rear+1) % maxsize == front;

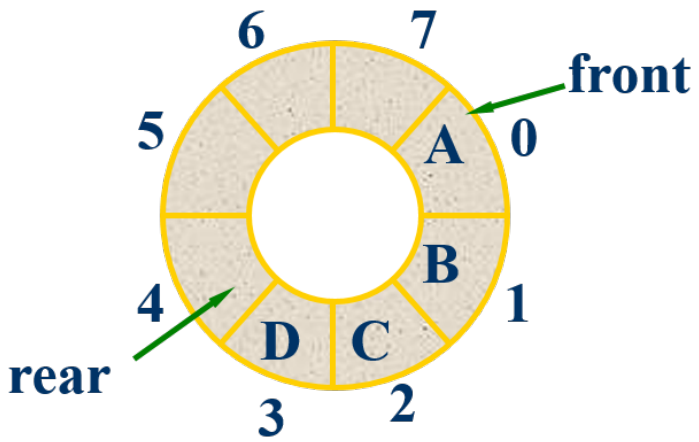


循环队列

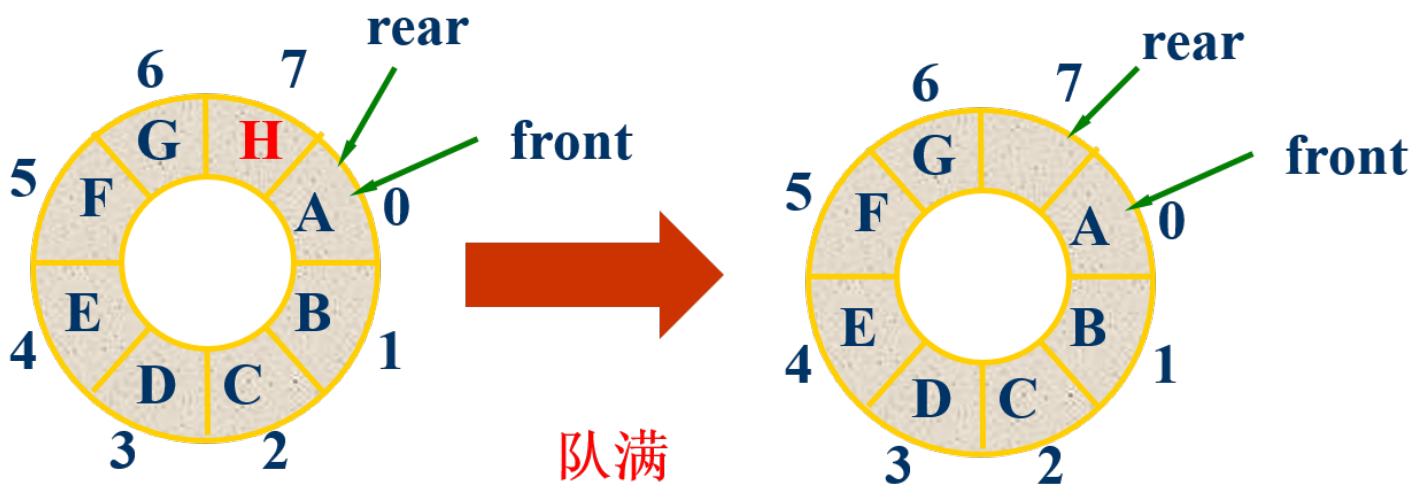
图解队满条件：(rear+1) % maxsize == front



空队列



一般情况



循环队列的实现

循环队列在内存中的表示：静态部分

```
1  #define MAXSIZE 100
2  typedef struct{
3      QElemType *base;
4      int front;
5      int rear;
6  } SqQueue;
```

基本操作的实现：初始化

```
1  Status InitQueue(SqQueue &Q){
2     Q.base=(QElemType *)malloc
3                    (MAXSIZE*sizeof(QElemType));
4     if (! Q.base) exit(OVERFLOW);
5     Q.rear = Q.front = 0;
6     return OK;
7  }
```

循环队列的实现

基本操作的实现：出队

```
1  Status DeQueue(SqQueue &Q, QElemType &e){
2      if ( Q.front == Q.rear ) return ERROR; //队空
3      e = Q.base[Q.front];
4      Q.front = ( Q.front+1) % MAXSIZE;
5      return OK;
6  }
```

基本操作的实现：入队

```
1  Status EnQueue(SqQueue &Q, QElemType e){
2      if ( (Q.rear+1) % MAXSIZE ==Q.front)
3            return ERROR;      //队满
4      Q.base[Q.rear] = e;
5      Q.rear = ( Q.rear+1) % MAXSIZE;
6      return OK;
7  }
```

队列的应用

离散事件模拟

- 超市收银台模拟程序，银行业务模拟程序等
- 模拟“等待-服务”这类业务的活动，并计算每个客户的平均逗留时间（= 等待时间 + 服务时间）
- 事件驱动模拟 (不同于教材的描述)：假设时间从 0 时刻开始，每个时刻以 p_a 的概率生成一个到达事件（客户）；第 i 个服务窗口以 p_i 的概率生成一个离开事件/客户（假设当前时刻窗口 i 正服务一个客户），客户该如何排队，才能降低逗留时间?

数据逻辑结构

- 事件列表：保留客户到达事件（发生时刻），客户离开事件（发生时刻）
- n 个服务窗口（收银台），模拟成 n 个队列。(可不可以 n 个窗口只用一个队列？先来先服务队列？长/短服务优先队列？好处?)
- 专门研究排队原理的排队论还要讨论服务时间，因此需要记录每个客户被服务时间

队列的应用

银行服务模拟：一个队列

```
1  Type struct{
2      int ArrivalTime;
3      int ServeiceStartTime; //上一客户的离开时刻
4      int LeaveTime;
5  } QElemType;              //队列元素类型
```

超市收银台模拟：多个队列

- 思考：存储结构设计?
- 如何将新到达客户分配到某个队列?

排队论的关键参数

- 到达时间分布、服务时间分布和服务窗口数量
- 现实应用中，收集上述三个时间，构成样本，用于统计分析