第四章 栈与队列

刘杰

人工智能学院

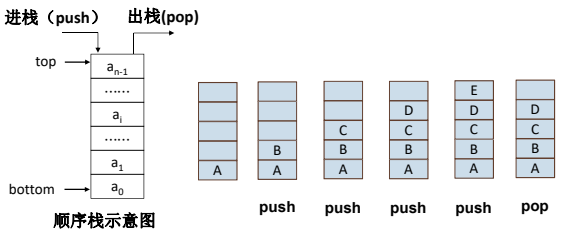
目录

4.1 栈

- 栈： 栈（stack）是限定仅在一端进行插入和删除的线性表,是"操作受限"的线性表。
- 能进行插入和删除的这一端称为栈顶（top），表的另一端称为栈底（bottom）。
- 用线性表的记号来表示栈，可以写为：S = (a_0, a_1, ...,a_i,... a_{n-1}), 0<i<n。当指定a_{n-1}那一端为栈顶的时候，另一端a_0就是栈底。当n = 0时，称为空栈。

栈的操作

- 在栈顶插入一个元素称为压栈（push）或入栈，从栈顶删除一个元素称为出栈（pop）。
- 入栈时按a_0, a_1, ..., a_{n-1}的次序入栈，而出栈时次序刚好相反，先退出a_{n-1}，然后才能退出a_{n-2}，最后退出a_0。所以栈又称为后进先出（LIFO，Last In First Out）结构。
- 实现栈的方法有两种方法：顺序栈和链式栈，它们分别对应于顺序表和单链表。



顺序栈示意图

栈的抽象数据类型描述

ADT stack

{

//实例元素的线性表，一端为栈底，另一端为栈顶

操作:

Stack Create (maxStackSize): 创建一个最大长度为maxStackSize的空栈

isEmpty ( ): 如果栈为空（ stack长度== 0），则返回TRUE，否则返回FALSE

IsFull ( ): 如果栈满（stack长度== maxStackSize ），则返回TRUE 否则返回FALSE

Top ( ): 返回栈顶元素

Push (stack, item): 向栈中添加元素x

Pop (stack) : 删除栈顶元素

}

顺序栈

栈的顺序存储结构简称为顺序栈，和线性表相类似，用一维数组来存储栈。根据数组是否可以根据需要增大，又可分为静态顺序栈和动态顺序栈。

- 静态顺序栈实现简单，但不能根据需要增大栈的存储空间；
- 动态顺序栈可以根据需要增大栈的存储空间，但实现稍为复杂。

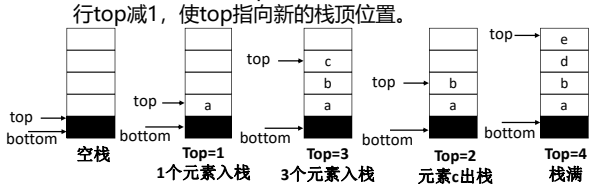
栈的静态顺序存储表示

采用**静态**一维数组来存储栈。

- 栈底固定不变的；栈顶则随着进栈和退栈操作而变化，用一个整型变量top（称为栈顶指针）来指示当前栈顶位置。
- 用top=0表示栈空的初始状态，每次top指向栈顶在数组中的存储位置。

栈的静态顺序存储表示

- 结点入栈：首先执行top加1，使top指向新的栈顶位置，然后将数据元素保存到栈顶（top所指的当前位置）。
- 结点出栈：首先把top指向的栈顶元素取出，然后执行top减1，使top指向新的栈顶位置。



静态堆栈变化示意图

基本操作的实现

1 栈的类型定义

```
# define  MAX_STACK_SIZE  100      /* 栈向量大小 */
typedef  int  ElemType ;
typedef struct  sqstack {
   ElemType   stack_array[MAX_STACK_SIZE] ;
   int  top, bottom;
}SqStack ;
```


基本操作的实现

2 栈的初始化

```
SqStack Init_Stack(void) {
   SqStack  S ;
   S.bottom=S.top=0 ;
   return S ;
}
```


基本操作的实现

3 压栈（元素入栈）

```
Status push(SqStack S , ElemType  e) {
/*  使数据元素e进栈成为新的栈顶  */
   if  (S.top==MAX_STACK_SIZE-1)
      return  ERROR;     /*  栈满，返回错误标志    */
   S.top++ ;     /*  栈顶指针加1  */
   S.stack_array[S.top]=e  ;  /* e成为新的栈顶  */
   return OK;       /*  压栈成功    */
}
```

基本操作的实现

4 弹栈（元素出栈）

```
Status  pop( SqStack   S, ElemType  *e ) {
/*弹出栈顶元素*/
   if ( S.top==0 )
      return ERROR ;      /*  栈空，返回错误标志    */
   *e=S.stack_array[S.top] ;
   S.top-- ;
   return OK ;
}
```


栈的溢出

- 上溢：当栈满时做进栈运算产生的空间溢出。是一种出错状态，应设法避免。
- 下溢：当栈空时做退栈运算产生的溢出。有可能是正常现象，因为栈在使用时，其初态或终态都是空栈，所以下溢常用来作为控制转移的条件。

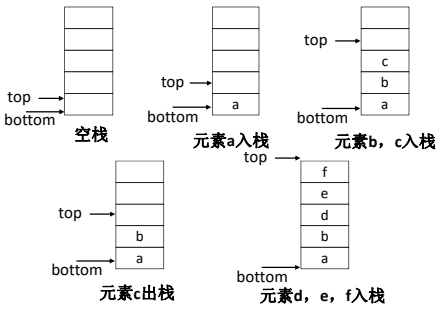
栈的动态顺序存储表示

采用**动态**一维数组来存储栈。所谓动态，指的是栈的大小可以根据需要增加。

- 用bottom表示栈底指针，栈底固定不变的；栈顶则随着进栈和退栈操作而变化。用top（称为栈顶指针）指示当前栈顶位置。
- 用top=bottom作为栈空的标记，每次top指向栈顶数组中的下一个存储位置。

栈的动态顺序存储表示

- 结点入栈：首先将数据元素保存到栈顶（top所指的当前位置），然后执行top加1，使top指向栈顶的下一个存储位置;
- 结点出栈：首先执行top减1，使top指向栈顶元素的存储位置，然后将栈顶元素取出。



基本操作的实现

1 栈的类型定义

```
#define  STACK_SIZE  100    /*  栈初始向量大小  */
#define STACKINCREMENT 10   /*  存储空间分配增量  */
typedef  int  ElemType ;
typedef struct Sqstack {
   ElemType  *bottom;    /*  栈不存在时值为NULL  */
   ElemType  *top;     /*  栈顶指针  */
   int   stacksize ;     /*  当前已分配空间，以元素为单位  */
}SqStack ;
```

基本操作的实现

2 栈的初始化

```
Status Init_Stack(void) {
   SqStack  S ;
   S.bottom=
      (ElemType *)malloc(STACK_SIZE *sizeof(ElemType));
   if (! S.bottom) return  ERROR;
   S.top=S.bottom ;   /* 栈空时栈顶和栈底指针相同 */
   S. stacksize=STACK_SIZE;
   return OK ;
}
```


基本操作的实现

3 压栈（元素入栈）

```
Status push(SqStack S , ElemType  e) {
  if  (S.top-S.bottom>=S. stacksize-1) {
    /* 栈满，追加存储空间 */
    S.bottom=
      (ElemType *)realloc(S.bottom,
      (STACKINCREMENT+ S. stacksize)*sizeof(ElemType));
    if (! S.bottom)  return  ERROR;
    S.top=S.bottom+S. stacksize-1 ;
    S. stacksize+=STACKINCREMENT ;
  }
  *S.top=e;  S.top++ ;   /* 栈顶指针加1，e成为新的栈顶 */
  return OK;}
```


基本操作的实现

4 弹栈（元素出栈）

```
Status pop( SqStack   S, ElemType  *e ) {
/*弹出栈顶元素*/
   if ( S.top== S.bottom )
      return ERROR ;     /* 栈空，返回失败标志 */
   S.top-- ;
   e=S. top ;
   return  OK ;
}
```


栈的链式存储表示

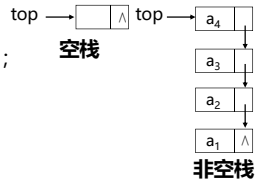
栈的链式表示

- 链栈：栈的链式存储结构，是运算受限的单链表。
- 链栈的插入和删除操作只能在表头位置上进行。因此，链栈没有必要像单链表那样附加头结点，栈顶指针top就是链表的头指针。图是栈的链式存储表示形式。出入栈的时间开销仅O(1)。

栈的链式存储表示

链栈的结点类型声明如下：

```
typedef  struct  Stack_Node
{
   ElemType   data ;
   struct Stack_Node  *next ;
} Stack_Node ;
```



基本操作的实现

1 栈的初始化

```
Stack_Node  *Init_Link_Stack(void) {
   Stack_Node  *top ;
   top=(Stack_Node*)malloc(
      sizeof(Stack_Node )) ;
   top->next=NULL ;
   return top;
}
```

基本操作的实现

2 **压栈（元素入栈）**

```
Status push(Stack_Node *top , ElemType  e) {
   Stack_Node  *p ;
   p=(Stack_Node  *)malloc(sizeof(Stack_Node)) ;
    /*  申请新结点失败，返回错误标志 */
   if (!p)  return  ERROR;
   p->data=e ;
   p->next=top->next  ;
   top->next=p ;
   return OK;
}
```


基本操作的实现

3 **弹栈（元素出栈）**

```
Status pop(Stack_Node  *top , ElemType *e) {
/*  将栈顶元素出栈  */
   Stack_Node  *p ;
   ElemType  e ;
   /*  栈空，返回错误标志    */
   if  (top->next==NULL ) return ERROR ;
   p=top->next ; e=p->data ;   /*  取栈顶元素  */
   top->next=p->next ;    /*  修改栈顶指针  */
   free(p) ;
   return OK ;
}
```


顺序栈与链式栈的比较

	时间效率	空间效率
顺序栈	都只需要常数时间	顺序栈必须申请一个固定长度的数组，当栈中元素相对较少时，空间浪费较大。
链式栈		链式栈的长度虽然可变，但是对于每个元素都需要一个指针域，这又产生了结构性空间开销。

栈的应用

由于栈具有的"后进先出"的固有特性，因此，栈成为程序设计中常用的工具和数据结构。以下是几个栈应用的例子。

数制转换

十进制整数N向其它进制数d(二、八、十六)的转换是计算机实现计算的基本问题。

转换法则：该转换法则对应于一个简单算法原理：

n=(n div d)*d+n mod d

其中：div为整除运算, mod为求余运算

例如 $(1348)_{10}= (2504)_8$，其运算过程如下：

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

数制转换

采用静态顺序栈方式实现

```
void conversion(int  n , int  d) {
  /*将十进制整数N转换为d(2或8)进制数*/
  SqStack  S ;   int k, *e ;
  S=Init_Stack();
  /*  求出所有的余数，入栈  */
  while  (n>0)   {  k=n%d ; push(S , k) ; n=n/d ; }
  while  (S.top!=0) {
     /*  栈不空时出栈，输出  */
     pop(S, e) ;
     printf("%1d" , *e);
  }
}
```

括号匹配问题

在文字处理软件或编译程序设计时，常常需要检查一个字符串或一个表达式中的括号是否相匹配?

- 匹配思想：从左至右扫描一个字符串（或表达式），则每个右括号将与最近遇到的那个左括号相匹配。则可以在从左至右扫描过程中把所遇到的左括号存放到堆栈中。每当遇到一个右括号时，就将它与栈顶的左括号（如果存在）相匹配，同时从栈顶删除该左括号。
- 算法思想：设置一个栈，当读到左括号时，左括号进栈。当读到右括号时，则从栈中弹出一个元素，与读到的左括号进行匹配，若匹配成功，继续读入；否则匹配失败，返回FLASE。

算法实现

```
#define TRUE  0
#define FLASE  -1
SqStack  S ;
S=Init_Stack() ;  /*堆栈初始化*/
int Match_Brackets(){
 char ch , x ;
 scanf("%c" , &ch) ;
 while (asc(ch)!=13) {
  if ((ch=='(')||(ch=='[')) push(S , ch) ;
  else if  (ch==']') {
    x=pop(S) ;
    if (x!='[') {
      printf("'['括号不匹配") ;
      return FLASE ;}}
```

```
else if  (ch==')') {
     x=pop(S) ;
     if (x!='(') {
       printf("'('括号不匹配") ;
       return FLASE ;
     }
   }
 }
 if  (S.top!=0) {
   printf("括号数量不匹配！ ") ;
   return FLASE ;
 }
 else return TRUE ;
}
```


栈与递归调用的实现

栈的另一个重要应用是在程序设计语言中实现递归调用，系统工作栈。

递归调用：一个函数（或过程）直接或间接地调用自己本身，简称递归（Recursive）。

递归是程序设计中的一个强有力的工具。因为递归函数结构清晰，程序易读，正确性很容易得到证明。

为了使递归调用不至于无终止地进行下去，实际上有效的递归调用函数（或过程）应包括两部分：递推规则（方法），终止条件。

例如：求n!

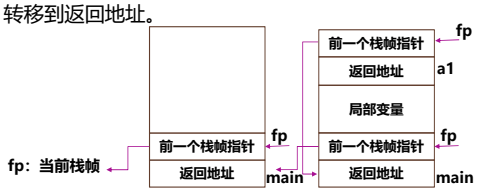
$$Fact(n)=\begin{cases}1 & \text{当n=0时 终止条件}\\ n*fact(n-1) & \text{当n>0时 递推规则}\end{cases}$$

为保证递归调用正确执行，系统设立一个“递归工作栈”，作为整个递归调用过程期间使用的数据存储区。

每一层递归包含的信息如：参数、局部变量、上一层的返回地址构成一个“工作记录”。每进入一层递归，就产生一个新的工作记录压入栈顶；每退出一层递归，就从栈顶弹出一个工作记录。

从被调函数返回调用函数的一般步骤：

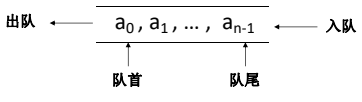
- (1) 若栈为空，则执行正常返回。
- (2) 从栈顶弹出一个工作记录。
- (3) 将“工作记录”中的参数值、局部变量值赋给相应的变量；读取返回地址。
- (4) 将函数值赋给相应的变量。
- (5) 转移到返回地址。



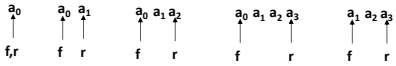
4.2、队列

- 队列（Queue）：只能在表的一端插入，在另一端删除的特殊线性表。
- 能进行插入（put）的一端称为队列的队尾，能进行删除（pop）的一端称为队列的队头/队首。在队列尾部插入时称为入队（enqueue）操作，从队列头部删除时称为出队（dequeue）操作。这种先来先服务的特性称为先进先出（First In First Out），简称FIFO。
- 当我们用原来线性表的记号来表示队列的时候，队列可以写为：Q = (a_0, a_1 , ..., a_{n-1})，其中a_0的一端为队列的头，a_{n-1}的一端为队列的尾。
- 例如：排队购物。操作系统中的作业排队。先进入队列的成员总是先离开队列。

队列中没有元素时称为空队列。在空队列中依次加入元素$a_0, a_1, \ldots, a_{n-1}$之后，$a_0$是队首元素，$a_{n-1}$是队尾元素。显然退出队列的次序也只能是$a_0, a_1, \ldots, a_{n-1}$ ，即队列的修改是依先进先出的原则进行的，如图所示。



队列示意图



队列的抽象数据类型描述

ADT Queue {

数据对象：有限长的有序表，元素取自Element

成员函数：

Create (maxQueueSize): 创建一个新的队列;

Bool IsEmpty (queue): 如果队列为空(queue长度==0)，则返回TRUE，否则返回FALSE；

Bool IsFull (queue): 如果队列满（queue长度==maxQueueSize），则返回TRUE，否则返回FALSE；

First ( ): 返回队列的第一个元素;

Last ( ): 返回队列的最后一个元素;

Bool AddQ (queue,item):向队列中添加元素item；

Element DeleteQ (queue):删除队列的头元素;

}

队列的顺序表示和实现

利用一组连续的存储单元（一维数组）依次存放从队首到队尾的各个元素，称为顺序队列。

对于队列，和顺序栈相类似，也有动态和静态之分。以下介绍的是静态顺序队列，其类型定义如下：

```
#define  MAX_QUEUE_SIZE   100
typedef  struct  queue {
   ElemType   Queue [MAX_QUEUE_SIZE] ;
   int   front ;
   int   rear ;
}SqQueue;
typedef  struct {
   int key;
   //……其他属性
} ElemType
```


顺序队列

- 为了掌握队列的位置，必须设两个指标分别指向队列的头和尾
- 指向队列头的叫做front，指向队列尾的叫做rear。
- 在出队列和入队列时，分别根据这两个值，可以只用O(1)的时间开销完成操作。

从空队列开始，依次将5、12、9、37插入队列，此后，先使5，12依次出队列，再将25，8，16依次插入队列，再让9出队列，将7，4，19，20插入。

经过多次使用后，顺序队列在数组中的情况

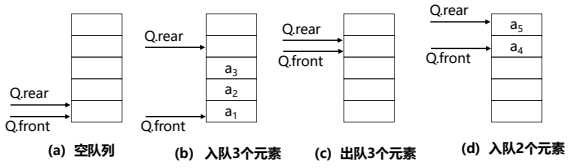
队列的顺序存储结构

设立一个队首指针front，队尾指针rear，分别指向队首和队尾元素。在非空队列里，队首指针始终指向队头元素，而队尾指针始终指向队尾元素的下一位置。

- 初始化：front=rear=0。
- 入队：将新元素插入rear所指的位置，然后rear加1。
  AddQ(queue,element item):: = queue[rear++]=item
- 出队：删去front所指的元素，然后加1并返回被删元素。
  DeleteQ(queue)::= queue[front++]
- 队列为空：Bool IsEmpty (queue):: = front=rear
- 队满：Bool IsFull:: rear = MAX_QUEUE_SIZE-1

在非空队列里，队首指针始终指向队头元素，而队尾指针始终指向队尾元素的下一位置。

假溢出：在入队和出队操作中，头、尾指针只增加不减小，致使被删除元素的空间永远无法重新利用。因此，尽管队列中实际元素个数可能远远小于数组大小，但可能由于尾指针已超出向量空间的上界而不能做入队操作。该现象称为假溢出。如图所示是数组大小为5的顺序队列中队首、队尾指针和队列中元素的变化情况。



(a) 空队列　(b) 入队3个元素　(c) 出队3个元素　(d) 入队2个元素

队列示意图

当rear = MAX_QUEUE_SIZE-1，应该把整个队列中的数据移到左边。

具体做法：

第一个元素放在queue[0]

并令front=0

计算rear并让它指向恰当单元。

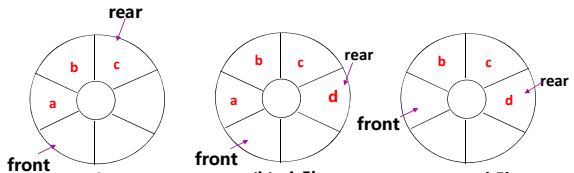
移动数据

弊端：O(MAX_QUEUE_SIZE)复杂度

循环队列

为充分利用向量空间，克服上述"假溢出"现象的方法是：将为队列分配的向量空间看成为一个首尾相接的圆环，并称这种队列为循环队列（Circular Queue）。

Front不在指向队头元素而是指向反时针方向的下一个，rear指向最后一个元素。



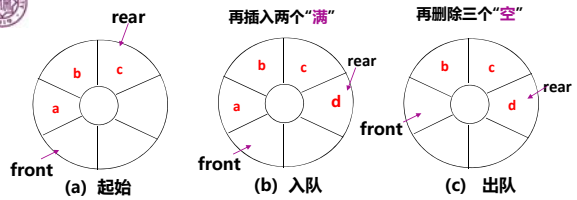
(a) 起始　(b) 入队　(c) 出队

在循环队列中进行出队、入队操作时，队首、队尾指针仍要加1，朝前移动。只不过当队首、队尾指针指向向量上界(MAX_QUEUE_SIZE-1)时，其加1操作的结果是指向向量的下界0。

这种循环意义下的加1操作可以描述为：

```
if (rear==MAX_QUEUE_SIZE)  rear=0;
else    rear++;
```

或者，rear=(rear+1)% MAX_QUEUE_SIZE



(a) 起始　(b) 入队　(c) 出队

入队时尾指针向前追赶头指针，出队时头指针向前追赶尾指针，故队空和队满时头尾指针均相等。因此，无法通过front==rear来判断队列"空"还是"满"。

解决此问题的方法是：约定入队前，测试尾指针在循环意义下加1后是否等于头指针，若相等则认为队"满"（front所指的单元始终为空）：

- **循环队列为空：front==rear。**
- **循环队列满：(rear+1)%MAX_QUEUE_SIZE ==front。**

2 入队操作

```
void addq (element item) {
   rear=(rear+1) % MAX_QUEUE_SIZE ;
   if (front==rear)
      queuefull () ; // 报告错误并退出
   queue[rear]=item ;
}
```

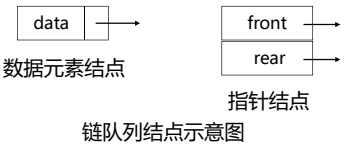
3 出队操作

```
ElemType delete() {
   ElemType item;
   if (front==rear)
      return queueEmpty () ;
   front = (front+1) % MAX_QUEUE_SIZE ;
   return queue[front];
}
```


队列的链式表示和实现

1 队列的链式存储表示

- 队列的链式存储结构简称为链队列，它是限制仅在表头进行删除操作和表尾进行插入操作的单链表。
- 需要两类不同的结点：数据元素结点，队列的队首指针和队尾指针的结点。



数据元素结点　　指针结点

链队列结点示意图

数据元素结点类型定义：

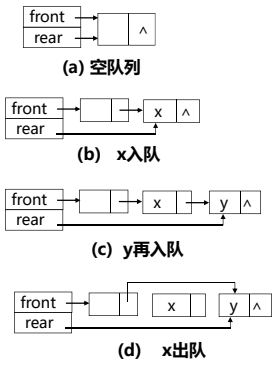
```
typedef struct queue {
   ElemType data ;
   struct queue  *next ;
}queue;
```

指针结点类型定义：

```
typedef struct link_queue {
    queue  *front , *rear ;
}link_queue ;
```


2 链队运算及指针变化

链队的操作实际上是单链表的操作，只不过是删除在表头进行，插入在表尾进行。插入、删除时分别修改不同的指针。链队运算及指针变化如图所示。



队列操作及指针变化

链队列的基本操作

1 链队列的初始化

```
link_queue *Init_LinkQueue (void) {
   link_queue *Q ;  queue  *p ;
   /* 开辟头结点 */
   p=(queue*)malloc(sizeof (queue)) ;
   p->next=NULL ;
   /*  开辟链队的指针结点  */
   Q = (link_queue *)malloc(sizeof (link_queue )) ;
   Q.front = Q.rear = p ;
   return Q;
}
```


链队列的基本操作

2 链队列的入队操作

在已知队列的队尾插入一个元素e，即修改队尾指针(Q.rear)。

```
Status Insert_CirQueue(link_queue *Q , ElemType  e){
   /*  将数据元素e插入到链队列Q的队尾  */
   p=(queue *)malloc(sizeof (queue)) ;
   /*  申请新结点失败，返回错误标志 */
   if (!p)  return  ERROR;
   p->data=e ; p->next=NULL ;      /*  形成新结点 */
   Q.rear->next=p ;  Q.rear=p ;  /*  新结点插入到队尾  */
   return OK;
}
```

链队列的基本操作

3 链队列的出队操作

```
Status  Delete_LinkQueue (link_queue *Q, ElemType *x) {
   queue *p ;
   if (Q.front==Q.rear)  return ERROR ;   /* 队空 */
   p=Q.front->next ;  /* 取队首结点 */
   *x=p->data ;
   Q.front->next=p->next ;    /* 修改队首指针 */
   /* 当队列只有一个结点时应防止丢失队尾指针 */
   if  (p==Q.rear)  Q.rear = Q.front ;
   free(p) ;
   return OK ;
}
```


链队列的基本操作

4 链队列的销毁

```
void  Destroy_LinkQueue (link_queue *Q ) {
   /* 将链队列Q的队首元素出队 */
   while  (Q.front!=NULL) {
      /* 令尾指针指向队列的第一个结点 */
      Q.rear=Q.front->next;
      /* 每次释放一个结点，第一次是头结点，以后是元素结点
*/
      free(Q.front);
      Q.front=Q.rear;
   }
}
```