第3章 栈和队列

- **主要内容：**
- 栈
  - 栈的基本定义
  - 栈的顺序存储和链式存储
  - 栈的应用
  - 栈与递归实现
- 队列
  - 队列的基本定义
  - 队列的链式存储
  - 循环队列—队列的顺序存储

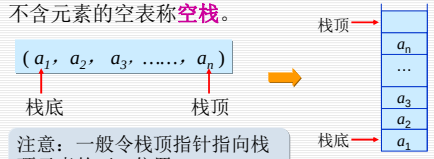
栈和队列概述

- 栈(stack)和队列(queue)是两种重要的线性结构
  - 从数据结构角度看，栈和队列也是线性表，其特殊性在于它们的基本操作是线性表的子集，是操作受限的线性表，可称为限定性的数据结构
  - 从数据类型角度看，其操作规则与线性表大不相同，是完全不同于线性表的抽象数据类型。
    - 栈按“后进先出”的规则进行操作
    - 队列按“先进先出”的规则进行操作

栈和队列是操作受限的特殊线性表

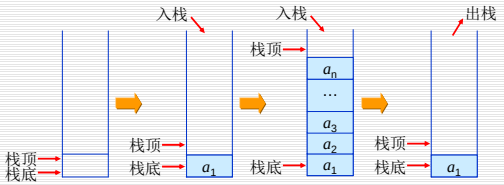
§ 3.1 栈

- 1. 栈的基本定义
- **栈**：限定在表尾进行插入和删除操作的线性表
  - **栈顶**(表尾)：栈中允许插入、删除的一端
  - **栈底**(表头)：栈中不允许插入、删除的一端
  - 插入也称为**进栈**(压栈)；删除也称为**出栈**(退栈)
- 不含元素的空表称**空栈**。



注意：一般令栈顶指针指向栈顶元素的下一位置

栈的操作特性



- 特点：先进后出(FILO)或后进先出(LIFO)

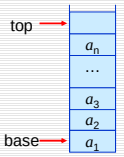
三个元素按a、b、c的次序依次进栈，且每个元素只允许进栈一次，则可能的出栈序列有多少种？

抽象数据类型定义

```
ADT  Stack{
   数据对象：D
   数据关系：R
   基本操作：Initstack(&s);      Destroystack(&s);
             Clearstack(&s);     Stackempty(s);
             Stacklength(S);     Gettop(S,&e);
             Push(&S,e);         Pop(&S,&e);
             Stacktraverse(S);
} ADT  Stack
```


2. 栈的顺序存储

- **顺序栈**：栈的顺序存储结构简称为顺序栈，即利用一组地址连续的存储单元依次存放从栈底到栈顶的数据元素。



如何改造数组实现栈的顺序存储？

- 确定用数组的低端表示栈底，并利用空间分配时的初始指针base来指示栈底位置
- 附设指针top指示栈顶元素在数组中的位置

栈不存在如何表示？

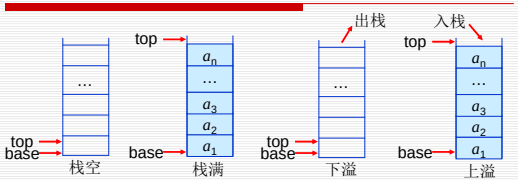
- 若base＝NULL，表示栈结构不存在。

存储结构类型定义

```
#define Stack_init_size  100;
#define Stackincrement  10;

typedef   struct{
   SElemType   *base; /*栈基址*/
   SElemType   *top; /*栈顶指针*/
   int  stacksize;  /*堆栈的大小*/
}SqStack;
```


栈的特殊状态



- 栈空：top=base或top=0
- 栈满：top-base>=stacksize或top=n
- 上溢和下溢
  - 上溢：当栈满时还继续入栈则产生空间溢出(出错)
  - 下溢：当栈空时再做退栈运算则产生溢出(正常，常作为程序控制转移条件)

3.顺序栈的基本操作

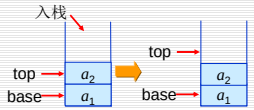
- (1) 初始化栈

```
status Initstack(Sqstack &s)
{  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;  }
```


(2) 入栈和出栈

入栈和出栈需要栈顶指针指示位置

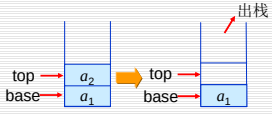
- 栈顶指针指向栈顶元素的下一位置



先入栈元素再修改指针

```
入栈：*s.top++=e;
  或：*s.top=e
      s.top++;
出栈：  e=*--s.top
  或：s.top--;
      e=*s.top ;
```

- 栈顶指针指向栈顶元素



先修改指针再出栈元素

```
入栈：*++s.top=e;
  或：  s.top++;
      *s.top=e;
出栈：e=*s.top--;
```



```
status Push(Sqstack &s, Selemtype e)
{  if (s.top-s.base>=s.stacksize) //是否栈满？避免上溢
   {   s.base=(SElemType *)realloc(s.base,
         (s.tacksize+Stackincrement)*sizeof(SElemType));
       if(!s.base) exit(OVERFLOW);
       s.top=s.base+s.stacksize;
       s.stacksize+=Stackincrement;
    }
   *s.top++=e;
   return OK;  }
```

注意边界条件：上溢和下溢

```
status Pop(Sqstack &s,Selemtype &e)
{  if (s.top==s.base) return ERROR; //是否栈空？避免下溢
   e=*--s.top;
   return Ok;  }
```


(3) 其它操作

```
取栈顶元素
status Gettop(Sqstack s,Selemtype &e)
{  if (s.top==s.base ) return ERROR;
   e=*(s.top-1);    return OK;  }
```

```
撤销栈
status Destroystack(SqStack &s)
{  if(s.base){ free(s.base); s.base=NULL;return OK; }
   return ERROR;     }
```

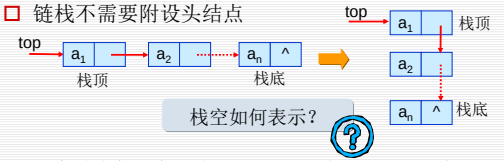
```
置栈空
status Clearstack(SqStack  &s)
{  if(s.base) { s.top=s.base;return OK;}
   return ERROR;        }
```

4. 栈的链式存储

- 链栈：栈的链式存储结构称为链栈。

如何改造链表实现栈的链式存储？

- 用链表表头作为栈顶，方便操作
- 链栈不需要附设头结点



栈空如何表示？

- 一个链表栈由栈顶指针top唯一确定。当top为NULL时，则表示栈空

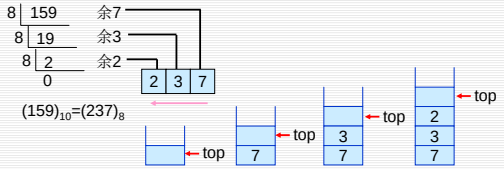
顺序栈和链栈总结

- 时间性能
  - 由于栈限定在栈顶进行插入和删除操作，因此顺序栈的时间复杂度和链栈一样为O(1)
- 空间性能
  - 顺序栈不浪费存储空间，但需要预分配空间。链栈有指针的开销，但是可动态分配空间
- 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。

§ 3.2 栈的应用举例

1. 数制转换

- 将十进制数N转换为r进制的数，其转换方法为辗转相除法
- 例如：把十进制数159转换成八进制数



- 实现方法：用一个栈来存放产生的余数，然后从栈顶到栈底依次输出余数就是所求的r进制数。

算法程序

```
void conversion(int N，int r)
{  SqStack  s;              /*定义一个顺序栈*/
   int   x;
   initStack(&s);           /*初始化栈*/
   while ( N )
   {  Push (&s, N % r );    /*余数入栈*/
      N = N / r; }          /*商作为被除数继续*/
   while ( !stackEmpty(& s ) ) /*输出结果*/
   {  Pop (&s, &x );
      printf ( " %d ", x ); }
}
```

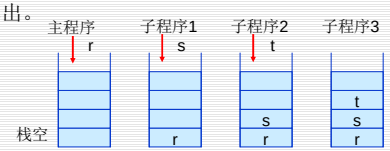
算法时间复杂度为$O(\log_r N)$

2. 括号匹配的检验

- 检验括号是否匹配的方法可表达为“期待的急迫程度”。例：((( ) ( )))
  - 设一个堆栈，把左括号依次压栈，当有匹配的右括号出现时，自动退栈。在算法的开始，栈应为空。

3. 过程/函数调用

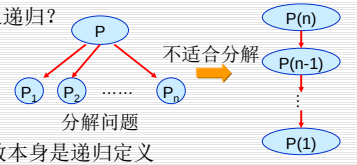
- 利用堆栈保存调用前后程序的上下文，恢复时一层层弹出。



§ 3.3 栈与递归实现

1. 递归

- 递归：直接调用自己或通过一系列的过程调用语句间接地调用自己的过程。
- 为什么引入递归？



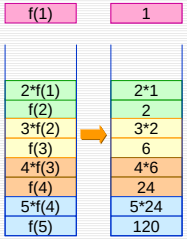
- 数学函数本身是递归定义
- 数据结构本身固有递归结构
- 使用递归求解比迭代更简单

例1:阶乘函数

$$f(n)=\begin{cases}1 & n=1\\ n\bullet f(n-1) & n>1\end{cases}$$

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

令n=5，过程？



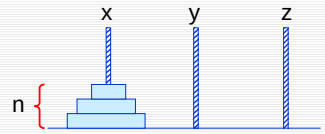
注意：递归的实现需要建立递归工作栈，系统每调用一次函数，就在系统堆栈中分配空间保存断点所需空间

递归的要素

- 递归的边界条件：确定递归到何时终止，也称为**递归出口**
  - 出口一般是一个或多个无须分解、可直接求解的最小子问题
- 递归模式：大问题是如何分解为小问题的，也称为**递归体**
  - 子问题应具有类似于原问题的特性。

注意：在递归步骤里分解问题时，应使子问题相对于原问题更接近于递归终止条件，从而保证经过有限次递归步骤后达到递归终止条件而结束递归。

例2：Tower of Hanoi问题



- 要求：x塔上的圆盘移到z上，并保持叠放顺序
- 游戏规则：
  - 将n个盘从x移至z；每次只能移动一个盘
  - 圆盘可以在x，y，z中的任意塔座上移动
  - 任何时刻都不能将一个较大的圆盘压在较小的圆盘之上

问题分析

- 设问题描述为 hanoi(n, x, y, z)
  - 当n=1问题很简单，可用move(x, 1, z)表示将一个盘从x移至z；
  - 当n>1，可将规模为n的问题转化为规模为n-1的问题，依次类推，直至转换为只有一个圆盘的Hanoi问题
    - 将塔x上的n-1个圆盘借助塔z先移到塔y上
    - 把塔x上剩下的一个圆盘移到塔z上
    - 将n-1个圆盘从塔y借助于塔x移到塔z上

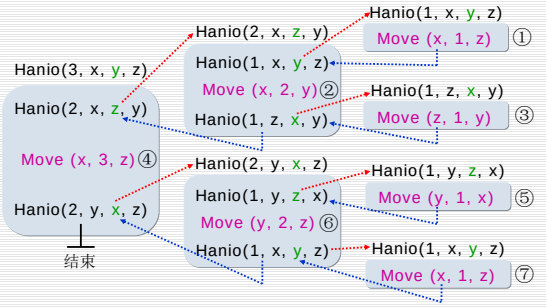
$$hanoi(n,x,y,z)=\begin{cases}hanoi(n-1,x,z,y)\\ move(x,n,z)\\ hanoi(n-1,y,x,z)\end{cases}$$

算法程序

```
int  count=0;
void hanoi(int n,char x,char y, char z)
{  if(n==1)  move(x,1,z);
   else{
     hanoi(n-1,x,z,y);
     move(x,n,z);      /*将x上编号为n的圆盘放到z上*/
     hanoi(n-1,y,x,z);  }
}
```

```
void move(char x, int n, char z)
{  printf("%dmove disk%dfrom%cto%c\n",++count, n,x,z);
   return;  }
```

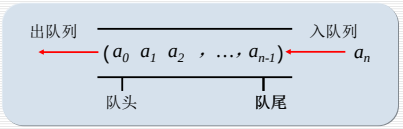

调用过程



§3.4 队列

1. 队列的基本定义

- 队列(Queue)是一种运算受限的线性表。它只允许在表的一端进行插入，而在另一端进行删除。
  - 允许删除的一端称为**队头**(front)，删除也称**出队列**
  - 允许插入的一端称为**队尾**(rear)，插入也称**入队列**
- 特点：先进先出



抽象数据类型定义

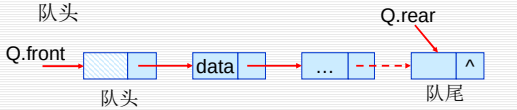
```
ADT  Queue{
   数据对象：D
   数据关系：R
   基本操作：Initqueue(&q);        Destroyqueue(&q);
             Clearqueue(&q);       Queueempty(q);
             Queuelength(q);       Gethead(q,&e);
             Enqueue(&q,e);        Dequeue(&q,&e);
             Queuetraverse(q);
} ADT  Queue
```


2. 队列的链表表示和实现

- 用线性链表表示队列的存储结构为**链队列**。
  - 特点：是限制仅在表头删除和表尾插入的单链表。

如何改造链表实现队列的链式存储？

- 定义链表表头为队头，并利用链表头指针指示队头



- 定义链表表尾为队尾，并增加一个尾指针，指向链表的最后一个结点

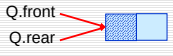
存储结构类型定义

```
typedef struct QNode{
   QElemType data;   /*队列元素类型*/
   struct QNode *next; /*指向下一个队列元素指针*/
}QNode, *QueuePtr;

typedef struct{
   QueuePtr front;    /*队列队头指针*/
   QueuePtr rear;     /*队列队尾指针*/
}LinkQueue;
```


3. 链队列的操作

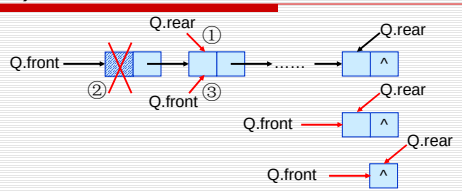
(1) 初始化



空队列

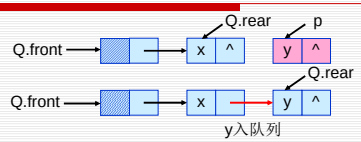
```
status Initqueue(linkqueue &Q)
{
   Q.front = Q.rear =(QueuePtr)malloc(sizeof(Qnode));
   if(!Q.front) exit(OVERFLOW);
   Q.front->next=NULL;
   return OK;
}
```


(2) 销毁队列



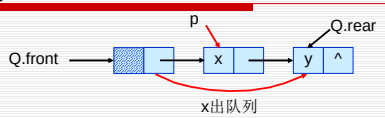

```
status Destroyqueue(linkqueue &Q)
{  while (Q.front)
   {  Q.rear = Q.front->next; //从队列头开始删除
      free(Q.front);
      Q.front= Q.rear; }
   return OK;  }
```

(3) 插入




```
status Enqueue(linkqueue &Q, Qelemtype e)
{   p=(QueuePtr)malloc(sizeof(QNode));
    if(!p) exit(OVERFLOW);
    p->data=e;
    p->next=NULL;
    Q.rear->next=p;
    Q.rear=p;
    return OK;   }
```


(4) 删除

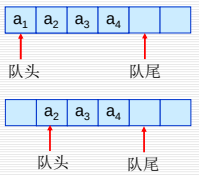



```
status Dequeue(linkqueue &Q,Qelemtype &e)
{   if (Q.front==Q.rear) return ERROR; //队空
    p=Q.front->next;
    e=p->data;
    Q.front->next =p->next;
    if (Q.rear==p) Q.rear=Q.front; //删除的是最后一个结点
    free(p);
    return OK;   }
```


4. 队列的顺序存储结构

- 队列的顺序存储结构称为顺序队列
  - 特点：限制在表头插入表尾删除的顺序表。

如何改造顺序表实现队列的顺序存储？



- 定义顺序表表头为队头，设立队尾指针指示变化的尾元素的下一位置

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，只要求存储在数组中连续的位置

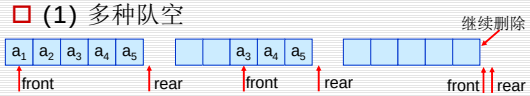
- 设立队头指针指示变化的头元素位置

存储结构类型定义

```
#define MAXQSIZE 100
typedef struct{
   QElemType *base;   /*队列基地址*/
   int   front;       /*头指针，若队列不空，指向头元素*/
   int   rear;        /*尾指针*/
 } SqQueue;
```


5.顺序队列的特殊状态

- (1) 多种队空

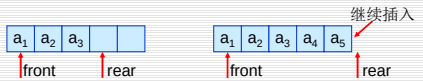


- 队满：Q.rear=MaxQSize
- 出队：e=Q.base[Q.front]；Q.front++;
- 队空：Q.front=Q.rear
- 下溢：队空继续删除

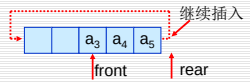
注意：队空可能在队列的任何位置发生。

- 初始队空：Q.front=Q.rear=0

(2) 假上溢



- 进队：Q.base[Q.rear]=x; Q.rear++;
- 上溢：队满继续插入



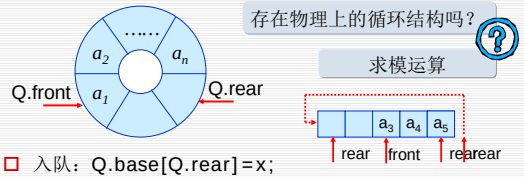
如何解决假上溢？

循环队列

- 假上溢：队尾指针已到最大位置，但队列低端空闲

6.循环队列(Circular Queue)

- 循环队列：将向量空间想象为一个首尾相接的圆环，称为循环向量，存储在其中的队列称为循环队列

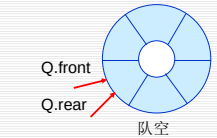


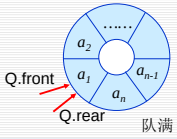
存在物理上的循环结构吗？

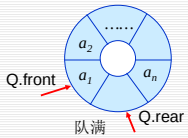
求模运算

- 入队：Q.base[Q.rear]=x;
  Q.rear=(Q.rear+1)% MaxQSize;
- 出队：x=Q.base[Q.front];
  Q.front=(Q.front+1)% MaxQSize;

队满、队空判定条件







Q.front=Q.rear；队空还是队满？

- 少用一个元素的空间
  - 队空：Q.front=Q.rear
  - 队满：(Q.rear+1) % MaxQSize=Q.front

队列能容纳的元素上界：MaxQSize-1

队满、队空判定条件(.续)

- 设置一个计数器num记录队列中元素的个数
  - 队空：num==0
  - 队满：num==MaxSize
- 设置一个标志字段Q.tag=0。若进队操作后有Q.front=Q.rear，则置tag=1；若出队操作后有Q.front=Q.rear，则置tag=0
  - 队空：Q.tag=0， Q.front=Q.rear
  - 队满：Q.tag=1， Q.front=Q.rear

7. 循环队列的操作

- (1) 初始化

```
status Initqueue(SqQueue &Q)
{  Q.base=(QElemType*)
          malloc(MaxQSize*sizeof(QElemType));
   if(!Q.base) exit(OVERFLOW);
   Q.front =Q.rear=0;
   return OK;  }
```

- (2) 取队列长度

```
int Queuelength(SqQueue Q)
{  return (Q.rear-Q.front+MaxQSize)%MaxQsize;  }
```


(3) 插入和删除

- 插入

```
status Enqueue(sqqueue &Q,Qelemtype e)
{   if ((Q.rear+1)%Maxsize==Q.front)  return ERROR;
    Q.base[Q.rear]=e;
    Q.rear=(Q.rear+1)%Maxsize;
    return OK;   }
```

- 删除

```
status Dequeue(sqqueue &Q,Qelemtype &e)
{   if (Q.front==Q.rear )   return ERROR;
     e=Q.base[Q.front];
     Q.front=(Q.front+1)%Maxsize;
     return OK;   }
```


队列总结

- 时间性能：
  - 循环队列和链队列其基本操作算法时间复杂度都为O(1)
- 空间性能
  - 循环队列需要在使用前确定最大队列长度，所以有存储元素个数的限制和空间浪费的问题。
  - 链式队列空间分配灵活，便于实现存储空间的共享，但是每个元素都需要一个指针域，从而产生了结构性开销。