**线性表的顺序表示：**

1. 线性表的顺序存储表示：

用一组地址连续的存储单元依次存储线性表的数据元素，这种存储结构通常称为顺序表，特点是逻辑上的相邻元素在其物理次序也是相邻的。

线性表的顺序存储结构是一种 随机存取 的存储结构。

l为线性表每个元素占用的存储单元个数

ai的存储位置可以表示为：LOC(ai)=LOC(a1)+(i-1)\*l;

//顺序表的存储结构

#define MAXSIZE 100

typedef struct{

ElemType \*elem; //存储空间的基地址

Int length; //当前长度，可获取表长，判断表是否为空

}SqList

1. 顺序表的基本操作
2. 初始化：

为顺序表动态分配一个预定义大小的数组空间，使elem指向这段空间的基地址。

将表的当前长度设为0

1. 查找：

根据给定元素序号查找:查找表中第i个数据元素，返回数组中第i-1个数据元素的值

根据给定值查找：查到第i个元素与给定值匹配，返回该元素的位序 i+1

1. 插入：

在第i个元素之前插入一个元素，需从最后一个元素即第n个元素开始，依次向后移动一个位置，直至第i个元素（共n-i+1个元素）

1. 删除：

删除第i个元素，需从第i+1个元素开始直至第n个元素，依次前移一个位置

**线性表的链式表示实现：（线性链表，单链表）**

1. 线性表的链式存储结构：

用一组任意的存储单元存储线性表的数据元素（这里存储单元可以是连续的，也可以是不连续的）

数据元素ai的存储映像，为结点node，其包含两个域：数据域和指针域

//单链表的存储结构

typedef struct LNode{

ElemType data; //结点的数据域

Struct LNode \*next; //结点的指针域

}LNode,\*LinkList; //LinkList为指向结构体LNode的指针类型

1. 单链表的基本操作：
2. 初始化：

//构造一个空的单链表L

Int initlist\_L(LinkList &L)

{

L=new LNode; //生成新结点作为头结点，用头指针指向头结点

L->next=NULL; //头结点的指针域为空，链表为空

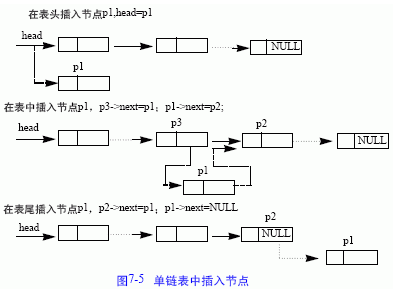
return 0;

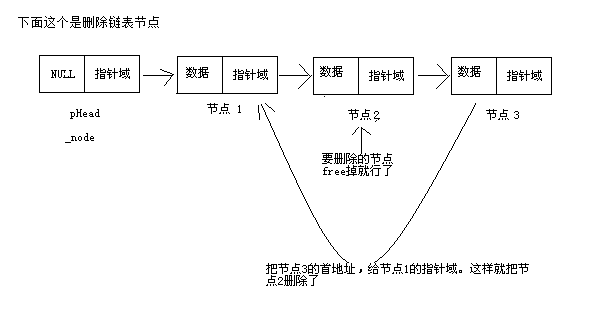
}

1. 查找：

按序查找：

按值查找：

1. 插入：  
    
2. 删除：



1. 前插法创建链表：

通过将新结点插入链表的头部（头结点之后）来创建链表

1. 后插法创建链表：

通过将新结点逐个插入到链表的尾部来创建链表。

1. 循环链表：

表中的最后一个结点的指针域指向头结点，整个链表形成一个环。

1. 双向链表：

双向链表的结点有两个指针域，一个指向直接后继，一个指向直接前继。

//双向链表的存储结构

typedef struct DuLNode{

ElemType data;

struct DuLNode \*prior;

struct DuLNode \*next;

}DuLNode,\*DuLinkList;

双向链表的优势在于：可以直接访问前驱，在某些时候查找速度比较快，但是也占用较多的存储空间。

**线性表的应用（表示集合和多项式）**

**栈：后进先出**

1. 栈是一种特殊的线性表，其特殊点在于限定插入和删除数据元素的操作都只能在线性表的一端操作。对栈来说，表的尾端有其特殊含义，称为栈顶（top）

//顺序栈存储空间的

#define MAXSIZE 100

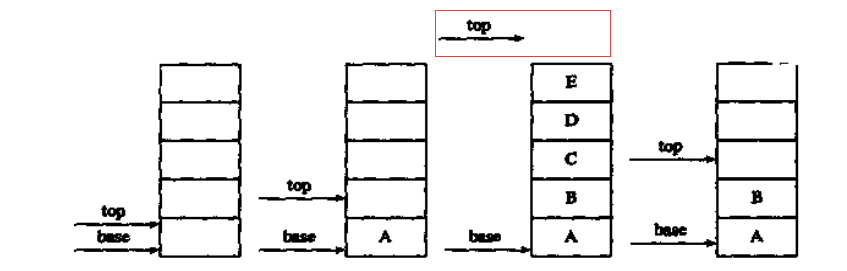
typedef struct{

SElemType \*base;

SElemType \*top;

int stacksize;

}SqStack;



1. 栈的操作：

构造空栈：

判栈空：

判栈满：

进栈：

退栈：

取栈顶元素：

顺序栈操作：

1. 初始化：

int Initstack(SqStack &S)

{

S.base=new SElemType[MAXSIZE];

if(!S.base) exit(-1);

S.top=S.base;

S.stacksize=MAXSIZE;

return 0;

}

1. 入栈：

int push(SqStack &S, SElemType e)

{

if(S.top-S.base==S.stacksize) return -1; //栈满

\*S.top++=e; //元素e压入栈顶，栈顶指针加1

return 0;

}

1. 出栈：

//若栈不空，则删除S的栈顶元素，用e返回其值

int pop(SqStack &S, SElemType &e)

{

if(S.top==S.base) return -1; //空栈

e=\*--S.top;

return 0;

}

1. 取栈顶元素：

SElemType gettop(SqStack S)

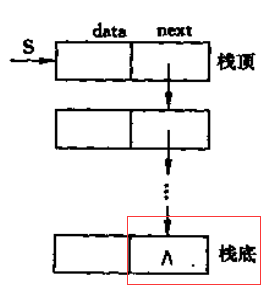
{

if(S.top==S.base) exit(-1); //空栈

return \*(S.top-1);

}

链栈的表示和实现：



typedef struct StackNode{

SElemType data;

struct StackNode \*next;

}

栈的应用：利用后进先出的特点，从高位到地位正好和计算过程相反

括号的匹配

\*通常，当在一个函数运行期间调用另一个函数时，在运行被调用函数之前，系统需先完成3件事：

a，将所有的实参，返回地址等信息传递给被调用函数保存；

b，为被调用函数的局部变量分配存储区；

c，将控制转移到被调函数的入口。

\*从被调函数返回调用函数之前，系统也完成3件工作：

a，保存被调函数的计算结果；

b，释放被调函数的数据区；

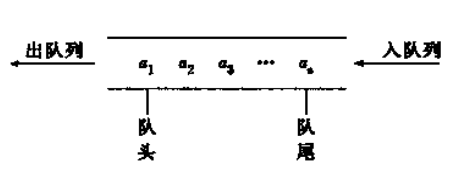
c，依照被调函数保存的返回地址将控制信息转移到调用函数；

递归算法的应用，将递归转换为非递归：对于单向递归，尾递归

**队列：先进先出**

队列是一种先进先出的线性表。它只允许在表的一端插入，在表的另一端删除元素。

允许插入的一端叫做队尾（rear）允许删除的一端叫做对头（front）



循环队列—队列的顺序表示：

#define MAXQSIZE 100

typedef struct{

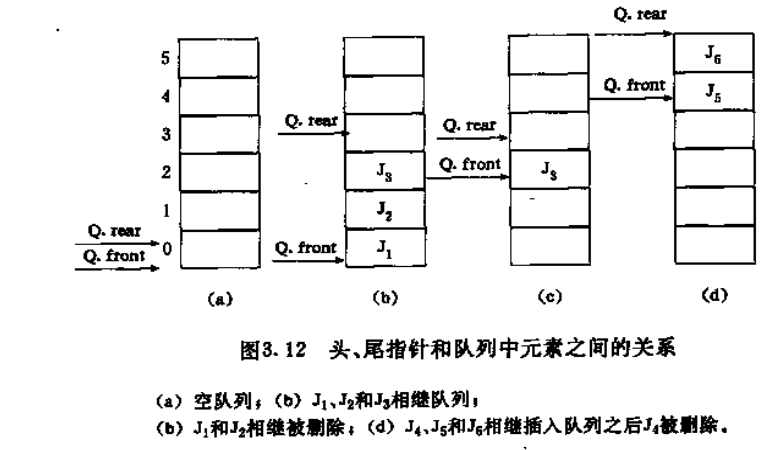
QElemType \*base;

int front; //头指针

int rear; //尾指针

}

操作：



为了在c语言中描述方便起见，在此约定，初始化创建空队列时，令front=rear=0，每当插入新元素是，尾指针rear增1，每当删除元素时头指针front增1。

由于队列“队尾入队，对头出队”的特点，会造成“假溢出”现象，循环队列可以很好解决这个问题（采用取模的方法实现Q.rear=(Q.rear+1)%队列长度）

对于循环队列不能以头，尾指针是否相同来判断队列空间是“满”还是“空”

解决方法：

1. 少一个元素空间，即队列空间大小为m时，有m-1个元素就认为是队满。这样判断对空的条件不变，即当头，尾指针值相同时，认为队空。

队空条件：Q.front=Q.rear

队满条件：(Q.rear+1)%MAXQSIZE=Q.front

循环队列的操作：

求循环队列的长度：(Q.rear-Q.front+MAXQSIZE)%MAXQSIZE

入队：Q.base[Q.rear]=e;

Q.rear=(Q.rear+1)%MAXQSIZE;

出队：e=Q.base[Q.front];

Q.front=(Q.front+1)%MAXQSIZE;

链队—队列的链式表示和实现：

typedef struct QNode{

QElemType data;

strcut QNode \*next;

}QNode,\*QueuePtr;

typedef struct{

QueuePtr front;

QueuePtr rear;

}LinkQueue;

操作：

1. 初始化：

struct LinkQueue Q;

Q.front=Q.rear=new QNode;

Q.front->next=NULL;

b，入队：

p=new QNode;

p->data=e;

p->next=NULL;

Q.rear->next=p;

Q.rear=p;

c，出队：

if(Q.front==Q.rear) return -1; //队空

p=Q.front->next;

e=p->data;

Q.front->next=p->next;

if(Q.rear==p) Q.rear=Q.front;

deletc p;

课本p66，图3.14

队列的应用：

1. 常是解决操作系统用来解决主机和外设之间速度不匹配问题或多个用户引起的资源竞争问题。
2. 杨辉三角，