# 绪论

《数据结构》是一门研究非数值计算的程序设计问题中计算机的**操作对象**以及它们之间的**关系**和**操作**的一门学科。

## 基本概念和术语

* 数据：是对客观事物的符号表示
* 数据对象：性质相同的数据元素的集合
* 数据元素：数据的基本单位，一个数据元素由若干个数据项构成（例如，一个书目信息，该书目为一个数据元素，作者，发布时间等信息为数据项），
* 数据项：不可分割的最小单元
* 数据结构：数据元素和数据元素之间存在的特定的关系
* 常见的数据结构有：集合、线性结构、树形结构、图状结构
* 一般用序偶<a1,a2>来表示对应的关系
* 逻辑结构：研究对象的特性及其相互之间的关系（是什么数据结构）
* 物理结构：数据结构在计算机中的表示
* 物理结构根据存储方式可以分为：顺序存储结构（数组）和链式存储结构（指针）
* 数据类型：值的集合和定义在这个值集合上的一组操作的总称（整型+加减乘除）

## 抽象数据类型(ADT)

一种数据类型，其**数据对象和对象操作的规格说明**独立于对象的存储表示和对象上操作的实现。

一个抽象数据类型应该包含定义、表示和实现三个部分

1. 抽象数据类型的定义：

抽象数据类型的定义仅取决于它的一组逻辑特性，而与其在计算机内部如何**表现和实现**无关，即不论其内部结构如何变化，只要它的数学特性不变，都不影响其外部使用。

抽象数据类型可以用三元组来刻画:(D,S,P)。其中D是数据对象，S是D上的关系集，P是对D的基本操作。

ADT 抽象数据类型名{

数据对象：<数据对象的定义>

数据关系：<数据关系的定义>

基本操作：<基本操作的定义>

} ADT 抽象数据类型名

2. 抽象数据类型的表示：

抽象数据类型的**表示**就是要将该类型映射到计算机中，也就是确定抽象数据类型的存储结构以及给出基于该结构之上的基本操作的函数原型。

3. 抽象数据类型的实现：

抽象数据类型的实现就是基于特定存储结构之上的基本操作的实现（函数写一下）

## 算法和算法分析

时间复杂度T(n)是指算法中所有语句的执行次数之和，人们关心的是当n趋于无穷时T(n)的数量级。将算法中基本运算执行次数的数量级作为时间复杂度

**常见的算法的时间复杂度之间的关系为：**

***O*(1)<*O*(log *n*)<*O*(*n*)<*O*(*n*log *n*)<*O*(*n*2)<*O*(2*n*)<*O*(*n*!)<*O*(*nn*)**

时间复杂度的计算遵循两种规则：

加法法则：T(n)=T1(n)+T2(n)=O(f(n))+O(g(n))=O(max(f(n),g(n)))

乘法法则：T(n)=T1(n)\*T2(n)= O(f(n))\*O(g(n))=O(f(n)\*g(n))

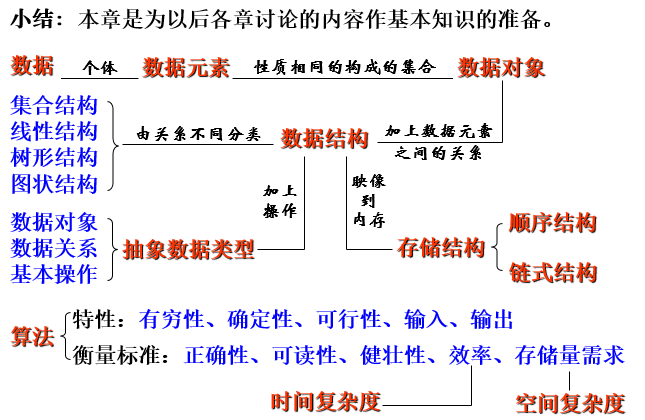
设a{} b{} c{}三个语句块的时间复杂度分别为O(1) O(n) O(n2)，则

（1）a{b{}c{}}的时间复杂度为O(n2)，满足加法法则

（2）a{b{c{}}}的时间复杂度为O(n3)，满足乘法法则

空间复杂度S(n)指算法运行过程中所使用的辅助空间的大小。（**算法在运行过程中临时占用的存储空间**）一般递归算法的空间复杂度为O(n)

算法原地工作时指算法所需所需辅助空间为常量，即O(1)



# 线性表

## 线性表的概念

文字定义：一个线性表是n个数据元素的有限序列。

一个数据元素可以由**若干个数据项**组成，这时，也可以称数据元素为**记录**。含有大量记录的线性表又称“**文件**”。

例1：26 个英文字母组成的字母表：（*A*, *B*, *C*, …, *Z*） （其中数据元素是字符）

例2：学生成绩表（90, 97, 60, 75,…,84）（其中数据元素是整数）

例3：学生健康情况登记表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **姓 名** | **学 号** | **性 别** | **年龄** | **健康情况** |
| **王小林** | **790631** | **男** | **18** | **健康** |
| **陈 红** | **790632** | **女** | **20** | **一般** |
| **刘建平** | **790633** | **男** | **21** | **健康** |
| **张立立** | **790634** | **男** | **17** | **神经衰弱** |
| **……..** | **……..** | **…….** | **…….** | **…….** |

其中整个登记表就是一个文件，其中的记录（数据元素）就是每个学生的情况。

因此：

* **线性表中的数据元素可以是各种各样的，但同一线性表中的元素必定具有相同特性（属于同一数据对象）。**
* **线性表中的数据元素之间存在着序偶关系** <*ai*–1, *ai*>。

线性结构的特点：

最常用且最简单的一种数据结构。



特点：4个“惟一”

在数据元素的非空有限集中，

* 存在唯一的一个被称作“第一个”的数据元素
* 存在唯一的一个被称作“最后一个”的数据元素
* 除第一个之外的数据元素均只有一个前驱
* 除最后一个之外的数据元素均只有一个后继

线性表的ADT定义：

ADT List {

数据对象：D＝{ ai | ai ∈ElemSet, i=1,2,...,n, n≥0 }

数据关系：R1＝{ <ai-1 ,ai >|ai-1 ,ai∈D, i=2,...,n }

基本操作：

结构初始化操作

结构销毁操作

引用型操作（**操作的结果不改变线性表中的数据元素，也不改变数据元素之间的关系）**

加工型操作 （**操作的结果或修改表中的数据元素，或修改元素之间的关系）**

} ADT List

## 线性表的顺序表示和实现

在计算机中**用一组地址连续的存储单元**依次存储线性表的各个数据元素，称作线性表的顺序存储结构或顺序映象。用这种方法存储的线性表称作顺序表。

假设线性表的每个元素需占 *l* 个存储单元，由此，所有数据元素的存储位置均可通过基地址得到：

***LOC*(*ai* ) *= LOC*(*a*1) + (*i* -1) × *l***

**特点：**以物理位置相邻表示逻辑关系；任一元素均可随机存取。因此在查找的时候时间复杂度与n没有关系，因此是o（1）。

**结论：已知位置、获取该位置上的元素非常方便，与该线性表的长度无关**

**动态分配的顺序存储结构：**

考虑到线性表因插入元素而使存储空间不足的问题，应允许数组容量进行动态扩充。（静态顺序存储->动态顺序存储）

#define LIST\_INIT\_SIZE 100 // 线性表存储空间的初识分配量

#define LISTINCREMENT 10 // 线性表存储空间的分配增量

typedef int ElemType;

typedef struct

{

ElemType \*elem; // 数组指针 指示线性表的基地址

int length; // 当前长度

int listsize; // 当前分配的存储容量（以sizeof(ElemType)为单位）

}SqList;

线性表的实现ADT：

1. 初始化

// 初始化顺序表

bool InitList(SqList \*L)

{

L->elem = (ElemType \*)malloc(sizeof(ElemType)\*LIST\_INIT\_SIZE);

if(L->elem == NULL)

return false;

L->length = 0;

L->listsize = LIST\_INIT\_SIZE;

return true;

}

2. 插入

线性表的插入运算是指在表的第 *i* (1 ≤ *i* ≤ *n* +1) 个位置上， 插入一个新结点 *b*，使长度为 *n* 的线性表 (*a*1, …, *ai* –1, *ai*, …, *an*) 变成长度为*n* + 1的线性表 (*a*1, …, *ai* –1, *b*, *ai*, …, *an*)

算法思想：

1）检查 *i* 值是否超出所允许的范围 (1 ≤ *i* ≤ *n* + 1) ，若超出，则进行“超出范围”错误处理；

2）将线性表的第 *i* 个元素和它后面所有元素均后移一个位置；

3）将新元素写入到空出的第 *i* 个位置上；

4）使线性表的长度增 1。

// 插入顺序表操作

bool ListInsert(SqList \*L, int i, ElemType e)

{

ElemType \*newspace;

ElemType \*p,\*q;

if(i > L->length+1 || i<1)// 注意这个加1

return false;

if(L->length >= L->listsize)

{

newspace = (ElemType \*)realloc(L->elem,sizeof(ElemType)\*(LISTINCREMENT+L->listsize));

if(NULL == newspace)

return false;

L->elem = newspace;

L->listsize+=LISTINCREMENT;

}

p = &(L->elem[i-1]);

for(q = &(L->elem[L->length - 1]);q>=p;--q)

\*(q+1) = \*q;

\*p = e;

++L->length;

return true;

}

说明：

（1）realloc的语法：

**格式**：[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88)名=（数据类型\*）realloc（要改变内存大小的指针名，新的大小）。

**返回值**：

如果重新分配成功则返回指向被分配内存的[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88)，否则返回空指针NULL。

Realloc产生两种结果，一种是直接在原有数组的后面直接添加新的内存空间，另一种就是要重新开辟原来的数据重新进行分配。（因此为避免出错，将重新分配的空间的地址赋值给一个新的变量）

（2）在循环过程中，除了上面的代码可以实现，也可以这样操作：

for(int j = L.length-1;j >= i-1;j--)

L.elem[j+1] = L.elem[j];

（3）时间复杂度：

问题规模是表的长度，设它的值为 *n*。

算法的时间主要花费在向后移动元素的 for 循环语句上。该语句的循环次数为 **(*n*– *i* +1)**。由此可看出，所需移动结点的次数不仅依赖于表的长度 *n*，而且还与插入位置 *i* 有关。

当插入位置在表尾 (*i*=*n* +1) 时，不需要移动任何元素；这是最好情况，**其时间复杂度 *O*(1)**。当插入位置在表头 (*i* = 1) 时，所有元素都要向后移动，循环 语句执行 *n* 次，这是最坏情况，**其时间复杂度 *O*(*n*)**。

由此可见，在顺序表上做插入或删除运算，平均要移动表上 **一半**元素。当表长 *n* 较大时，算法的效率相当低。算法的 平均时间复杂度为 *O*(*n*)。

3. 删除操作

线性表的删除运算是指将线性表的第 *i* (1 ≤ *i* ≤ *n*) 个结点 删除，使长度为 *n* 的线性表(*a*1, …, *ai* –1, *ai*, *ai* +1, …, *an*) 变成长度为 *n* -1 的线性表 (*a*1, …, *ai* –1, *ai* +1, …, *an*)

算法思想：

1) 检查 *i* 值是否超出所允许的范围 (1≤*i*≤*n*)，若超出，则进行“超出范围”错误处理；

2) 将线性表的第 *i* 个元素后面的所有元素均前移一个位置；

3) 使线性表的长度减 1。

bool ListDelete(SqList \*L,int i,ElemType \*e)

{

ElemType \*p,\*q;

if(i>L->length || i<i)

return false;

p = &(L->elem[i-1]);

\*e = \*p;

q = &(L->elem[L->length -1]);

for(;p<q;p++)

\*p = \*(p+1);

--L->length;

return true;

}

**删除算法的复杂度分析**

问题规模是表的长度，设它的值为 *n*。

算法的时间主要花费在向前移动元素的 for 循环语句上。 该语句的循环次数为 (*n* – *i*)。由此可看出，所需移动结点的次数不仅依赖于表的长度 *n*，而且还与删除位置 *i* 有关。

当删除位置在表尾 (*i* = *n*) 时，不需要移动任何元素；这是最好情况，其时间复杂度 *O*(1)。当删除位置在表头 (*i* = 1) 时，有 *n* -1 个元素要向前移动，循环语句执行 *n* -1 次，这是最坏情况其时间复杂度 *O*(*n*)。

由此可见，在顺序表上做删除运算，平均约要移动表上**一半元素**。当表长 *n* 较大时，算法的效率相当低。算法的平均时间复杂度为 *O*(*n*)。

## 线性表的链式表示和实现

用一组**物理位置任意**的存储单元来存放线性表的数据元素。 这组存储单元既可以是连续的，**也可以是不连续的**，甚至是零散分布在内存中的任意位置上的。因此，链表中元素的逻辑次序和物理次序不一定相同。

单链表是由头指针唯一确定，因此单链表可以用头指针的名字来命名。

typedef struct LNode{

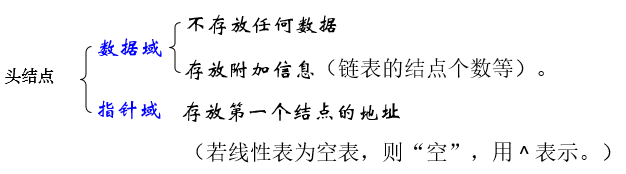
//声明结点的类型和指向结点的指针类型

ElemType data; //数据元素的类型

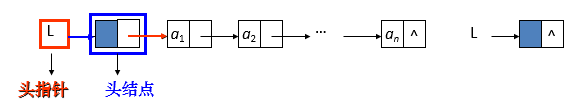
struct LNode \*next; //指示结点地址的指针

}LNode, \*LinkList;

头结点：在单链表的第一个结点之前人为地附设的一个结点，单链表设置头结点的作用是进行操作时，可以对空表、非空表情况以及对首元结点进行统一处理，编程更方便。



头指针存放头结点的地址。



**单链表的基本操作：**

1. 查找运算：

（1）按照序号进行查找

在单链表中，即使知道被访问结点的序号 *i* ，也不能像顺序表中那样直接按序号 *i* 访问结点，而只能从头指针出发，顺链域next逐个结点往下搜索，直到搜索到第 *i* 个结点为止。因此，单链表是非随机存取的存储结构。

bool GetElem(LinkList L,int i,ElemType \*e)// 按照序号进行查找，例如查找第2个元素

{// 算法的时间复杂度为：*O*(*n*)

LinkList p = L->next;

int j = 1;// j从1开始变化

if(p != NULL && j<i)

{

j++;

p = p->next;

}

if(!p || j>i) return false;

\*e = p->data;

return true;

}

（2）按值查找（LocateElem(L,e) 在链表中的实现）

按值查找是在单链表中查找结点值等于给定值 key的结点，若有的话，则返回首次找到的其值为 key 的结点的存储位置；否则返回 NULL。其算法如下：

该算法的执行时间与 key 有关，时间复杂度为：*O*(*n*)

LinkList GetElem(LinkList L,ElemType e)

{

LinkList p = L->next;

while(p != NULL && p->data != e)

p = p->next;

return p;// p此时由两种状态 一种是NULL，另一种就是结点等于e的位置

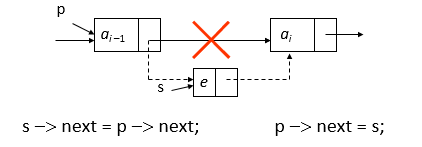
}

2. 插入运算（ListInsert(&L, *i*, *e*)在链表中的实现）

（1）首先找到 *ai* -1 的存储位置 p。

（2）生成一个数据域为 *e* 的新结点。

（3）插入新结点：① 新结点的指针域指向结点 *ai*  ② 结点 *ai* –1 的指针域指向新结点



bool ListInsert(LinkList \*L,int i,ElemType e)

{

LinkList p = \*L;

int j = 0;

while(p != NULL && j<i-1)

{

p = p->next;

j++;

}

if(!p || j>i-1) return false;

LinkList s = (LinkList)malloc(sizeof(LNode));

s->data = e;

s->next = p->next;

p->next = s;

return true;

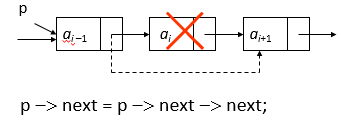
}

3. 删除运算（ListDelete(&L, *i*, &*e*)在链表中的实现）

（1）首先找到 *ai* –1 的存储位置 p

（2）令 p −> next 指向 *ai*+1

（3）释放结点 *ai*的空间



bool ListDelete(LinkList \*L,int i,ElemType \*e)

{

LinkList p = \*L;

int j = 0;

while(p != NULL && j<i-1)

{

j++;

p = p->next;

}

if(!p || j>i-1) return false;

LinkList q = p->next;

\*e = q->data;

p->next = q->next;

free(q);

return true;

}

4. 建立单链表（头插法建表 逆序建表）

// 头插法建立单链表

void CreateList(LinkList \*L,int n)

{

// 初始化一个头结点

(\*L) = (LinkList)malloc(sizeof(LNode));

(\*L)->next = NULL;

LinkList p;

int i = 0;

while(i<n)

{

p = (LinkList)malloc(sizeof(LNode));

p->data = i+1;

p->next = (\*L)->next;

(\*L)->next = p;

++i;

}

}

// 尾插法建立单链表

void CreateList(LinkList \*L,int n)

{

// 初始化一个头结点

(\*L) = (LinkList)malloc(sizeof(LNode));

(\*L)->next = NULL;

LinkList p,r = (\*L);

int i = 0;

while(i<n)

{

p = (LinkList)malloc(sizeof(LNode));

p->data = i+1;

p->next = NULL;

r->next = p;

r = p;

++i;

}

}

## 静态链表

数组的一个分量表示一个结点，同时用游标（指示器cur）代替指针指示结点在数组中的相对位置。数组的第零分量可看成头结点，其指针域指示链表的第一个结点。（该存储结构，当指针域为0时，代表已经结束，这个结构巧妙的利用了数组的下标进行操作）

#define MAXSIZE 1000 //链表的最大长度

typedef struct

{

ElemType data;

int cur;

}component,SLinkList[MAXSIZE];// 定义了一个结构体数组

静态链表相关操作：

1. 查找第一个值为e的元素，若找到返回它在L中的位序，否则返回0

int LocateElem(SLinkList S,ElemType e)

{

int i = S[0].cur;

while(i!=0 && S[i].data != e)

i = S[i].cur;

return i;// i可能为0 也可能为找到e的对应的位序

}

2. 将整个数组初始化成一个链表

void InitSpace(SLinkList \*S)

{

// 将数组初始化为一个链表

for(int i = 0;i<MAXSIZE-1;i++) S[i]->cur = i+1;

S[MAXSIZE-1]->cur = 0;// 最后一个是空的

}

3. 静态链表的malloc函数

int Malloc(SLinkList \*S)

{

// 如果初始化的链表是空的，那么返回分配的结点的下标，否则返回0

int i = S[0]->cur;

// S[0]->cur不为0 说明由可用的空间

if(S[0]->cur) S[0]->cur = S[i]->cur;// 头结点指针域存放的是可用空间的指针域

return i;

}

4. 静态链表的free函数

void Free(SLinkList \*S,int k)

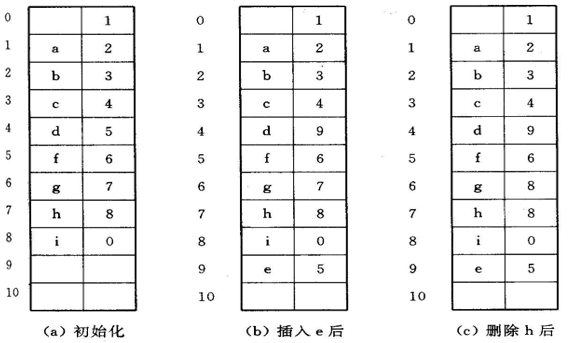
{

// 将下标为k的空闲结点回收到备用链表中

S[k]->cur = S[0]->cur;// 将下一个可用的先存放起来

S[0]->cur = k;// 先将它本身作用当前下一个可用的

}



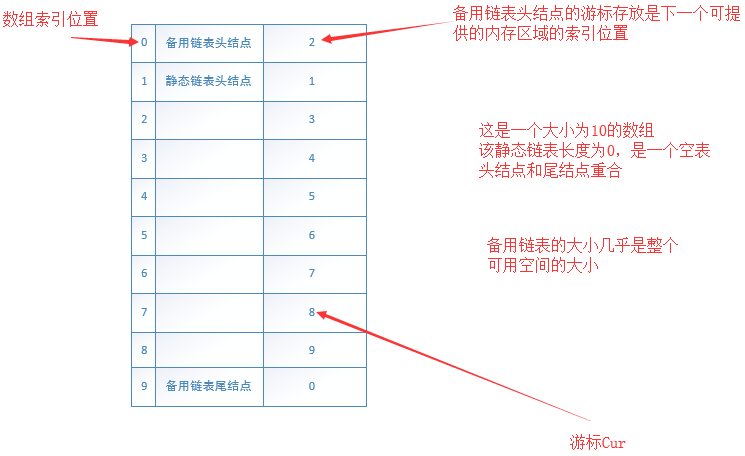
静态链表的插入操作是从备用链表上取出第一个结点的指针域（即第一个可用的空间）作为待插入的新结点，删除将从链表中删除下来的结点链接到备用链表上

**说明：静态链表的备用链表**

由于静态链表提前申请了有限的内存空间，在使用的过程中，极有可能会出现申请的内存空间不足，需要使用之前被遗弃的内存空间。所以这里在静态链表里就出现了个备用链表的概念。

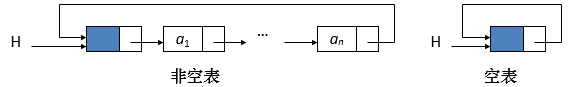
备用链表就是在某个指定内存内，指定内存内的所有的空闲内存链接成的一个链表，是空闲的链表。而含数据的链表就是我们的静态链表。当我们的静态连续需要空闲位置来插入带数据的新结点时，就会向备用结点请求分配一个空闲的结点来使用。通俗的说，备用链表就是用来帮助静态链表来管理限定区域的空闲内存的帮手。如果没有备用链表。当限定的内存区域充满了无用的废弃结点(被静态链表删除的结点)时，静态链表就无能为力了，明明还有空间，但是却无法使用。

备用链表和含数据的静态链表共有一片内存区域，如同一个数组内。备用链表是全部空闲的内存区域集合，静态链表是所有有用的内存区域集合，当静态链表向备用链表申请不到空闲内存区域时，则代表该内存空间已被耗尽。



## 循环链表（单向的）

循环链表：是一种头尾相接的链表（即：表中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. 循环链表为空：H->next == H
2. 循环链表为非空时，遍历的终止条件为p->next == L

优点：从表中任一结点出发均可找到表中其他结点。

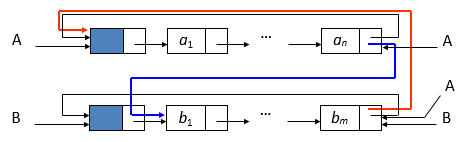
**头指针表示的单循环链表：**

1. 找*a1*的时间复杂度为：O(1)
2. 找*an*的时间复杂度为：O(n)——不方便

**尾指针表示的单循环链表：**

1. *a*1 的存储位置是：R−>next−>next，时间复杂度为O(1)
2. *an*的存储位置是：R，时间复杂度为O(1)

实例：将两个单向循环线性表合并成一个单向循环线性表。（AB分别是尾指针）



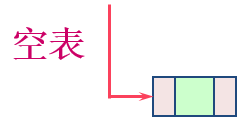
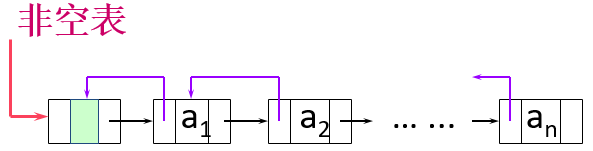
主要操作：C = A->next; A->next = B->next->next; B->next = C; A=B;

此操作仅需改变两个指针即可。时间复杂度是 *O*(1)。

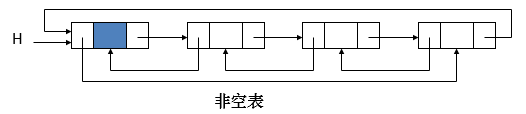
## 双向链表

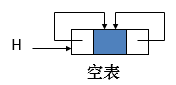
由于在单链表中，NextElem的执行时间为O(1)，PriorElem的执行时间为O(n)

双向链表就是有两个指针域，一个指向直接后继，另一个指向直接前驱

双向循环链表：让头结点的前驱指针指向链表的最后一个结点，让最后一个结点的后继指针指向头结点。





双向链表的结构可以如下定义：

**typedef struct DuLNode{**

**Elemtype data;**

**struct DuLNode \*prior, \*next;**

**} DuLNode, \*DuLinkList;**

双向链表结构的对称性（设指针 p 指向某一结点）：p->prior->next = p->next->prior

双向链表的操作：

1. 插入操作



s->prior = p->prior; s->next = p; p->prior->next = s; p->prior = s;

2. 删除操作



p->next->prior = p->prior; p->prior->next = p->next

## 各种存储类型之比较

1.顺序存储的固有特点：逻辑顺序与物理顺序一致，本质上是用数组存储线性表的各个元素（即随机存取）；存储密度大，存储空间利用率高。

2.链式存储的固有特点：元素之间的关系采用这些元素所在的节点的”指针”信息表示(插、删不需要移动节点)。

3.静态存储的固有特点：在程序运行的过程中不**用考虑追加内存的分配问题**。

4.动态存储的固有特点：可动态分配内存；有效的利用内存资源，使程序具有可扩展性。

**动态顺序表和动态链式表各有哪些优缺点？**

* 动态顺序存储：
  + 优点：存储密度大，存储空间利用率高，可随机存取。节点空间可动态申请追加。
  + 缺点：插入或删除元素时不方便。
* 动态链式存储：
  + 优点：插入或删除元素时很方便，使用灵活。结点空间可以动态申请和释放；
  + 缺点：存储密度小，存储空间利用率低，非随机存取。

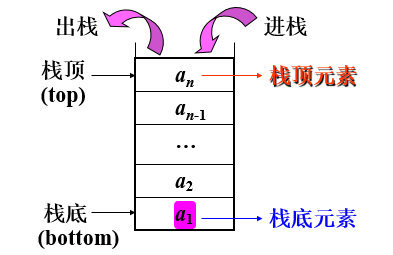
**顺序表、链表各自的使用场合？**

* 顺序表适宜于做查找这样的静态操作；
* 链表宜于做插入、删除这样的动态操作。
* 若线性表的长度变化不大，且其主要操作是查找，则采用顺序表；
* 若线性表的长度变化较大，且其主要操作是插入、删除操作，则采用链表。

# 栈和队列

## 栈的定义及其ADT定义

限定仅在表尾进行插入或删除操作。 后进先出 (LIFO结构)。



对于栈来说，表尾端称为栈顶，相应的，表头端称为栈底。不含元素的空表称为空栈。

栈的抽象数据类型的定义：

**ADT Stack {**

数据对象：D＝{ ai | ai ∈ElemSet, i = 1, 2, ..., n, n≥0 }

数据关系：R1＝{ **<ai-1, ai >**| ai-1, ai∈D, i = 2, ..., n } 约定an 端为栈顶，a1 端为栈底。

基本操作：

**InitStack(&S) ：**构造一个空栈 S

**DestroyStack(&S) ：**销毁一个栈S

**GetTop(S, &*e*)：**用 *e* 返回 S 的栈顶元素

**StackEmpty(S)**：若栈 S 为空栈，则返回 TRUE，否则 FALSE

**StackLength(S)**：返回 S 的元素个数，即栈的长度

**ClearStack(&S)**：将 S 清为空栈

**Push(&S, *e*)：**插入元素 *e* 为新的栈顶元素

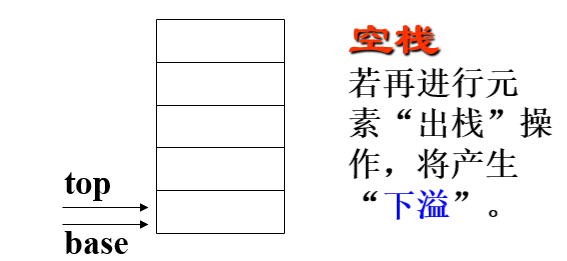
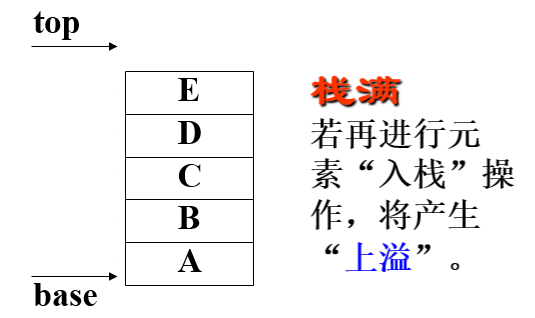
**Pop(&S, &*e*)：**删除 S 的栈顶元素，并用 *e* 返回其值

} ADT Stack

## 栈的表示和实现（顺序栈）

顺序栈：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，同时附设指针 top 指示栈顶元素在顺序栈中的位置

栈的两种状态：栈空、栈满

其中**base为栈底指针**，在顺序栈中，**它始终指向栈底的位置**，若base的值为NULL，则表明**栈结构不存在**。称top为栈顶指针，其初值指向栈底，**即top = base为栈空的标记**，每当插入新的栈顶元素的时候，指针top增加1，删除栈顶元素时，指针top减1.因此：top指示栈顶的位置，在非空顺序栈中栈顶指针指向栈顶元素的下一个位置。

#define STACK\_INIT\_SIZE 100//栈存储空间的初识分配量

#define STACKINCREMENT 10//栈存储空间的分配增量

typedef struct

{

SElemType \*base;//栈底指针，它始终指向栈底位置

SElemType \*top;//栈顶指针

int stacksize;//当前分配的栈可使用的最大存储容量

}SqStack;

栈的基本操作：

bool InitStack(SqStack \*S)

{

// 初始化一个空栈S

S->base = (SElemType \*)malloc(sizeof(SElemType) \* STACK\_INIT\_SIZE);

if(!S->base)

return false;

S->top = S->base;

S->stacksize = STACK\_INIT\_SIZE;

return true;

}

bool GetTop(SqStack S,SElemType \*e)

{

// 若栈不为空，则用e返回S的栈顶元素，并返回true,否则返回false

if(S.top == S.base) return false;

\*e = \*(S.top-1);

return true;

}

bool Push(SqStack \*S,SElemType e)

{

// 插入元素 e 为新的栈顶元素

if(S->top - S->base >= S->stacksize)

{

S->base = (SElemType \*)realloc(S->base,sizeof(SElemType) \* (S->stacksize + STACKINCREMENT));

if(!S->base)

return false;

S->top = S->base+ S->stacksize;

S->stacksize+=STACKINCREMENT;

}

\*(S->top) = e;// 先赋值，之后再让指针往后指，实质就是\*S->top++ = e;

S->top++;

return true;

}

bool Pop(SqStack \*S,SElemType \*e)

{

// 若栈不空，则删除S的栈顶元素，用e返回其值，并返回OK，否则返回error

if(S->top == S->base)

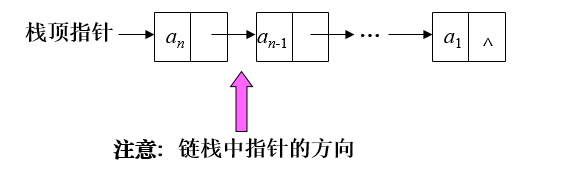
return false;

\*e = \* --S->top;// 先让指针往前指，之后再赋值，实质就是S.top--;e = \*(S.top);

return true;

}

## 栈的表示和实现（链栈）



typedef struct SNode

{

SElemType data;

struct SNode \*next;

}SNode,\*LinkStack;

链栈的主要操作：

bool InitStack(LinkStack \*S)

{

// 初始化栈

(\*S) = NULL;

return true;

}

bool GetTop(LinkStack S,SElemType \*e)

{

// 得到栈顶元素

if(S == NULL)

return false;

\*e = S->data;

return true;

}

bool Push(LinkStack \*S,SElemType e)

{

// 入栈

LinkStack p = (LinkStack)malloc(sizeof(SNode));

if(!p)

return false;

p->next = \*S;

p->data = e;

\*S = p;

return true;

}

bool Pop(LinkStack \*S,SElemType \*e)

{

// 出栈

LinkStack p = \*S;

if(\*S == NULL)

return false;

\*e = p->data;

(\*S) = p->next;

free(p);

return true;

}

## 栈的应用、与递归的关系

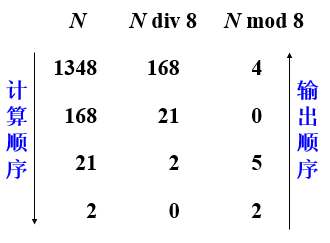
1. 数制转换：

十进制数 *N* 和其他 *d*进制数 *M* 的转换是计算机实现 计算的基本问题，其解决方法很多，其中一个简单算法是 逐次除以基数 *d* 取余法，它基于下列原理：

***N* = (*N* div *d* )\**d* + *N* mod *d***

具体作法为：首先用*N*除以*d*，得到的余数是*d*进制数*M*的最低位*M*0，接着以前一步得到的商作为被除数，再除以*d*，得到的余数是*d*进制数*M*的次最低位*M*1，依次类推，直到商为0时得到的余数是*M*的最高位*Ms*（假定*M* 共有*s* +1 位）。

**例： (1348)10=(2504)8，其运算过程如下：**



实现过程：

void conversion ()

{

int stack[4];

int top=0;// 初始化栈

int N;

scanf(“%d”, N);

while (N) {

stack[top]=N%8;

top++;

N=N/8;

}// push到栈里面

for(top=top-1; top>=0; top--)

printf(“%d”,stack[top]);// 出栈

}

2. 括号匹配的检验：

假设表达式中允许括号嵌套，则检验括号是否匹配的方法可用“期待的急迫程度”这个概念来描述。



过程如下：

1进栈，2进栈，3进栈，4是3所期待的那个人，因此3出栈，5进栈，6是5期待的那个人，因此5出栈，7是2期待的那个人，因此2出栈，8是1期待的那个人，因此1出栈

算法的设计思想：

1）**凡出现左括号，则进栈**；

2）凡出现右括号，首先**检查栈是否空**。**若栈空，则表明该“右括号”多余**；

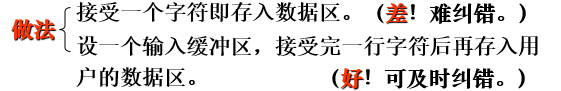
否则和栈顶元素比较，**若相匹配**，则“**左括号出栈**”，**否则表明不匹配**。

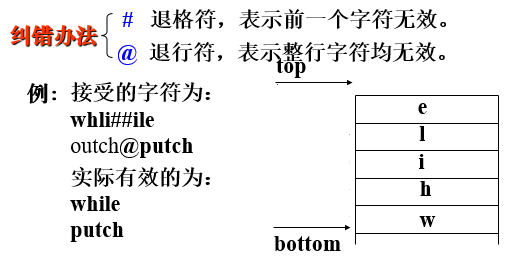
3）表达式检验结束时，**若栈空，则表明表达式中匹配正确**，否则表明“左括号”有多余的。

注意：**在算法开始和结束时，栈应该都是空的**。

3. 行编辑程序：

功能：接受用户从终端输入的数据并存入用户的数据区。





4. 迷宫求解问题：

求迷宫路径算法的基本思想：

（1）若当前位置“可通”，则纳入路径，继续前进；

（2）若当前位置“不可通”，则后退，换方向（按东南西北 的顺序）继续探索；

（3）若四周“均无通路”，则将当前位置从路径中删除出去

5. 表达式求值问题

**先乘除后加减**

**从左算到右；**

**运算规则**

**先括号内，后括号外；**

例：求表达式 **4+2×3-10/5** 的值。

计算顺序为：4 + 2 \*3 = 4 + 6 – 10/5 = 10 – 10/5 = 10 – 2 = 8

为实现算符优先运算，可以使用两个工作栈，一个称作OPTR，用以寄存运算符；另一个称作OPND，用以寄存操作数或运算结果，算法的基本思想是：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 步骤 | OPTR栈 | OPND | 输入字符 | 主要操作 |
| 1 | **#** |  | **4+2×3-10/5#** |  |
| 2 | **#** | **4** | **+2×3-10/5#** | **Push 4** |
| 3 | **# +** | **4** | **2×3-10/5#** | **Push +** |
| 4 | **# +** | **4 2** | **×3-10/5#** | **Push 2** |
| 5 | **# + \*** | **4 2** | **3-10/5#** | **Push \*** |
| 6 | **# + \*** | **4 2 3** | **-10/5#** | **Push 3** |
| 7 | **# +** | **4 6** | **-10/5#** | **2\*3** |
| 8 | **#** | **10** | **-10/5#** | **4+6** |
| 9 | **# -** | **10** | **10/5#** | **Push -** |
| 10 | **# -** | **10 10** | **5#** | **Push 10** |
| 11 | **# - /** | **10 10** | **5#** | **Push /** |
| 12 | **# - /** | **10 10 5** | **#** | **Push 5** |
| 13 | **# -** | **10 2** | **#** | **10/5** |
| 14 | **#** | **8** | **#** | **10-2** |

6. 地图染色问题（同样利用了栈）

7. 栈与递归的实现

**递归：**一个直接调用自己或通过一系列的调用语句间接地调用自己的函数，称做递归函数。



当在一个函数的运行期间调用另一个函数时，在运行该被调用函数之前，需先完成三件事：

1. 将实参等传递给被调用函数，保存返回地址（**入栈**）；
2. **为被调用函数的局部变量分配存储区**；
3. 将控制转移到被调用函数的入口。

从被调用函数返回调用函数之前，应该完成：

1. 保存被调函数的计算结果；
2. 释放被调函数的数据区；
3. 按被调函数保存的返回地址（**出栈**）将控制转移到调用函数。

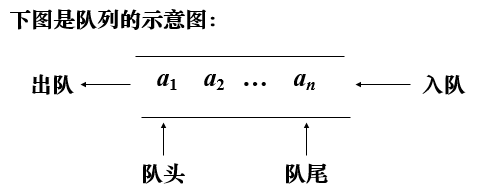
**多个函数嵌套调用的规则是：后调用先返回。**

此时的内存管理实行“栈式管理”（可以看ppt的动画说明）

## 队列的定义及其ADT定义

限定在表的一端插入（队尾）、另一端删除（对头）。

先进先出 (FIFO结构)。



当队列中没有元素时称为**空队列。**

队列的抽象数据类型的定义：

**ADT Queue {**

**数据对象：**D＝{*ai*| *ai*∈ElemSet, *i* =1, 2, ..., *n*, *n*≥0}

**数据关系：**R1＝{ <*ai* -1,*ai* > | *ai* -1, *ai* ∈D, *i* =2, ..., *n*}

约定其中 *a*1 端为队列头，*an* 端为队列尾。

**基本操作：**

**InitQueue(&Q) ：**操作结果：构造一个空队列 Q。

**DestroyQueue(&Q) ：**队列 Q 被销毁，不再存在。

**QueueEmpty(Q) ：**若 Q 为空队列，则返回 TRUE，否则返回 FALSE。

**QueueLength(Q) ：**返回 Q 的元素个数，即队列的长度。

**GetHead(Q, &*e*) ：**用 *e* 返回 Q 的队头元素。

**ClearQueue(&Q) ：**将 Q 清为空队列。

**EnQueue(&Q, *e*)** ：插入元素 *e* 为 Q 的新的队尾元素。

**DeQueue(&Q, &*e*)** ：删除 Q 的队头元素，并用 *e* 返回其值。

**} ADT Queue**

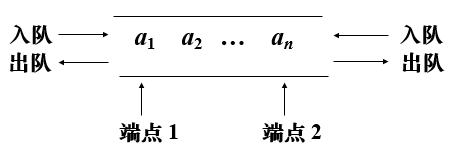
双端队列

**双端队列(double-ended queue**) **：**限定插入和删除在表的两端进行。（端点1、端点2）

先进先出 (FIFO结构)。

**输出受限的双端队列：**一个端点可插入和删除， 另一个端点仅可插入。

**输入受限的双端队列：**一个端点可插入和删除，另一个端点仅可删除。



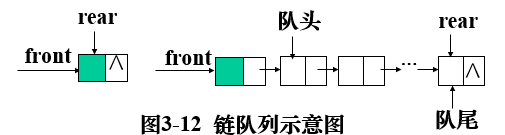
## 队列的表示和实现（链队列）

链队列：用链表表示的队列。是限制仅在队**头删除和队尾插入**的单链表。

一个链队列由一个头指针和一个尾指针唯一确定。

（因为仅有头指针不便于在表尾做插入操作）。

为了操作的方便，也给链队列添加一个头结点，因此，空队列的判定条件是：**头指针和尾指针都指向头结点。**



typedef struct QNode

{

QElemType data;

struct QNode \*next;

}QNode , \*QueuePtr;// 定义队列的结点

typedef struct

{

QueuePtr front;// 定义队头指针

QueuePtr rear;// 定义队尾指针

}LinkQueue;

**队列的基本操作在链队列中的实现：**

bool InitQueue(LinkQueue\* Q)

{

// 初始化一个队列

Q->front = Q->rear = (QueuePtr)malloc(sizeof(QNode));

if(!Q->front)

return false;

Q->front->next = NULL;

return true;

}

bool DestoryQueue(LinkQueue\* Q)

{

while(Q->front)

{

Q->rear = Q->front->next;// 在这里rear起到一个临时指针的作用，指向后一个结点，销毁前一个结点

free(Q->front);

Q->front = Q->rear;

}

return true;

}

bool EnQueue(LinkQueue \*Q,QElemType e)

{

// 插入元素 只能从队尾进行插入

QueuePtr p = (QueuePtr)malloc(sizeof(QNode));

if(!p)

return false;

p->data = e;

p->next = NULL;

Q->rear->next = p;

Q->rear = p;

return true;

}

bool DeQueue(LinkQueue\* Q,QElemType \*e)

{

// 删除元素 只能从对头进行删除

if(Q->front == Q->rear)

return false;// 不能进行删除

QueuePtr p = Q->front->next;

\*e = p->data;

if(p == Q->rear)

{

// 如果此时队列里面就一个元素，那么删除第一个元素的时候，要改变rear指针

Q->rear = Q->front;

}

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

free(p);

return true;

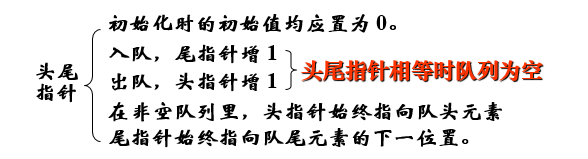
}

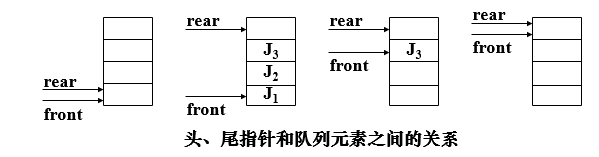
## 循环队列--队列的顺序表示和实现

是限制仅在表头删除和表尾插入的顺序表。

利用一组地址连续的存储单元依次存放队列中的数据元素。

因为：队头和队尾的位置是变化的，所以：设头、尾指针。





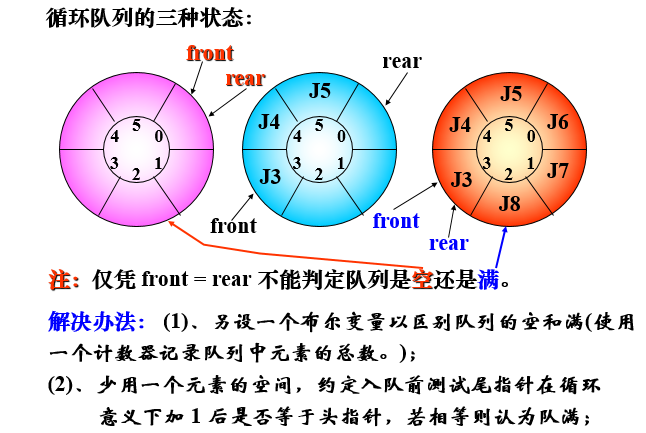
在顺序队列中，当尾指针已经指向了队列的最后一个位置的下一位置时，若再有元素入队，就会发生“溢出”

**“假溢出”——队列的存储空间未满，却发生了溢出。**

解决“假溢出”的问题有两种可行的方法：

(1)平移元素：把元素平移到队列的首部。效率低。

(2)将新元素插入到第一个位置上，构成**循环队列**， 入队和出队仍按“先进先出”的原则进行。 操作效率、空间利用率高。



循环队列的顺序存储结构：

#define MAXQSIZE 100 // 最大队列长度

typedef struct{

QElemType \*base; // 预分配存储空间基址

int front; // 头指针，若队列不空，指向队列头元素

int rear; // 尾指针，若队列不空，指向队列尾元素的下一个位置

}SqQueue;

**循环队列的基本操作：**

循环意义下的加 1 操作可以描述为：rear = (rear + 1)% MAXQSIZE

bool InitQueue(SqQueue \*Q)

{

// 构造一个空的队列

Q->base = (QElemType \*)malloc(sizeof(QElemType) \* MAXQSIZE);

if(!Q->base)

return false;

Q->rear = Q->front = 0;

return true;

}

int QueueLength(SqQueue Q)

{

// 返回Q的元素个数，即队列的长度

return (Q.rear - Q.front + MAXQSIZE)%MAXQSIZE;

}

bool EnQueue(SqQueue\* Q,QElemType e)

{

if((Q->rear + 1)%MAXQSIZE == Q->front)

return false;// 队此时已经满了不能再添加了

Q->base[Q->rear] = e;

Q->rear = (Q->rear + 1)%MAXQSIZE;

return true;

}

bool DeQueue(SqQueue \*Q,QElemType \*e)

{

if(Q->rear == Q->front)

return false;// 表示队列此时为空 不能删除

\*e = Q->base[Q->front];

Q->front = (Q->front + 1)%MAXQSIZE;

return true;

}

## 实验拓展

1． 设有两个栈S1,S2都采用顺序栈方式，并且共享一个存储区[O..maxsize-1],为了尽量利用空间，减少溢出的可能，可采用栈顶相向，迎面增长的存储方式。试设计S1,S2有关入栈和出栈的操作算法

#include <stdio.h>

#include <stdlib.h>

#define MAXSIZE 100

typedef int SElemType;

typedef struct{

SElemType \*base;

int top[2];

}SqStack;

bool InitStack(SqStack \*S)

{

S->base = (SElemType\*)malloc(sizeof(SElemType)\*MAXSIZE);

if(!S->base)

return false;

S->top[0] = 0;

S->top[1] = MAXSIZE -1;

return true;

}

bool Push(SqStack \*S,int i,SElemType e)

{

// 其中i表示的是第几个栈

if(S->top[0] == S->top[1] -1)

return false;// 栈已经满了 不能入栈了

if(i == 0)

{

S->base[S->top[0]] = e;

S->top[0]++;

}

if(i == 1)

{

S->base[S->top[1]] = e;

S->top[1]--;

}

return true;

}

bool Pop(SqStack \*S,int i,SElemType \*e)

{

// 其中i表示的是第几个栈

if(S->top[0] == 0 && i == 0)

return false;

if(S->top[1] == MAXSIZE-1 && i == 1)

return false;

if(i == 0)

{

S->top[0]--;

\*e = S->base[S->top[0]];

}

if(i == 1)

{

S->top[1]++;

\*e = S->base[S->top[1]];

}

return true;

}

2． 编程：假设以数组Q[m]存放循环队列中的元素，同时以rear和length分别指示队列中的队尾位置和队列中所含元素的个数，试给出该循环队列的队空条件和队满条件，并写出相应的初始化、插入、删除元素的操作

队空的条件：length == 0

队满的条件：length == m

#include <stdio.h>

#include <stdlib.h>

#define MAXSIZE 100

typedef int QElemType;

typedef struct{

QElemType \*base;

int rear;

int length;

}SqQueue;

bool InitQueue(SqQueue \*Q)

{

Q->base = (QElemType\*)malloc(sizeof(QElemType)\*MAXSIZE);

if(!Q->base)

return false;

Q->rear = -1;

Q->length = 0;

return true;

}

bool EnQueue(SqQueue \*Q,QElemType e)

{

if(Q->length == MAXSIZE)

return false;// 队列是满的，因此不能进行插入操作

Q->rear = (Q->rear+1)%MAXSIZE;

Q->base[Q->rear] = e;

Q->length++;

return true;

}

bool DeQueue(SqQueue \*Q,QElemType \*e)

{

if(Q->length == 0)

return false;

\*e = Q->base[(Q->rear - Q->length + 1 + MAXSIZE)%MAXSIZE];

Q->length--;

return true;

}