# 绪论

C++常用数据结构STL汇总：

<https://blog.csdn.net/Leader_wang/article/details/82959362>

<https://blog.csdn.net/Richard__Ting/article/details/79490665>

《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))

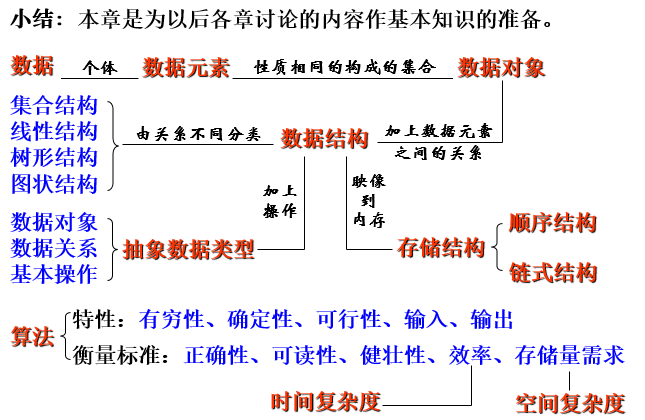
设a{} b{} c{}三个语句块的时间复杂度分别为O(1) O(n) O(n2)，则

（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{

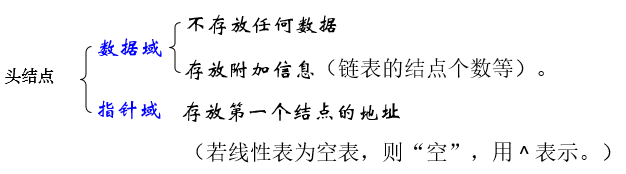
//声明结点的类型和指向结点的指针类型

ElemType data; //数据元素的类型

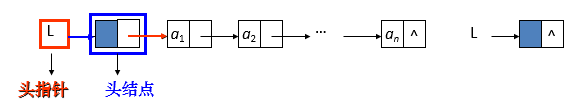
struct LNode \*next; //指示结点地址的指针

}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

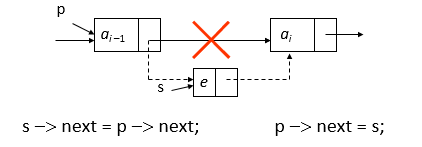
}

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;

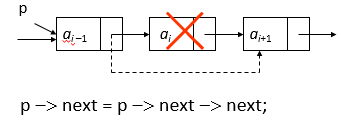
}

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)

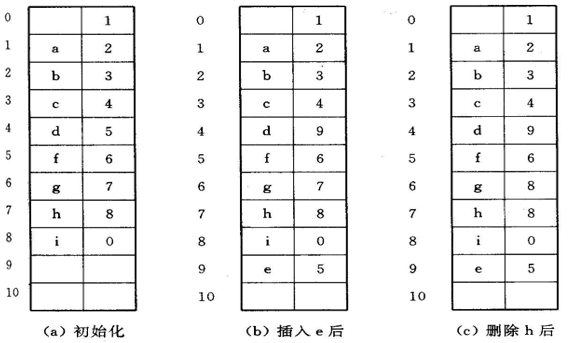
{

// 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

}



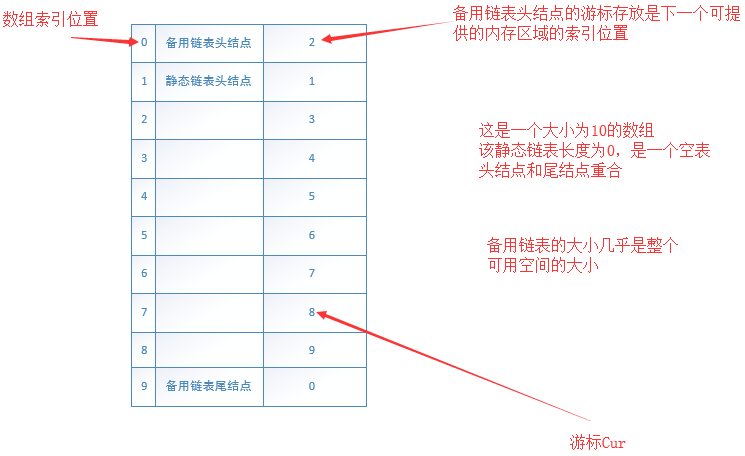
静态链表的插入操作是从备用链表上取出第一个结点的指针域（即第一个可用的空间）作为待插入的新结点，删除将从链表中删除下来的结点链接到备用链表上

**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。

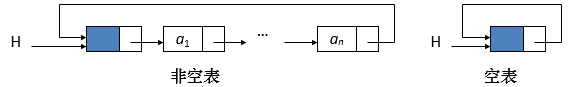
备用链表就是在某个指定内存内，指定内存内的所有的空闲内存链接成的一个链表，是空闲的链表。而含数据的链表就是我们的静态链表。当我们的静态连续需要空闲位置来插入带数据的新结点时，就会向备用结点请求分配一个空闲的结点来使用。通俗的说，备用链表就是用来帮助静态链表来管理限定区域的空闲内存的帮手。如果没有备用链表。当限定的内存区域充满了无用的废弃结点(被静态链表删除的结点)时，静态链表就无能为力了，明明还有空间，但是却无法使用。

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。

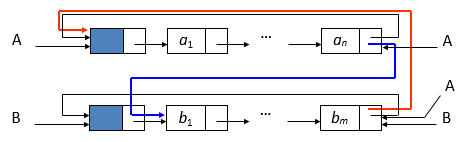
**头指针表示的单循环链表：**

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）



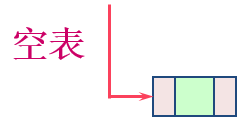
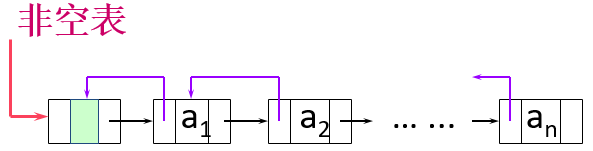
主要操作：C = A->next; A->next = B->next->next; B->next = C; A=B;

此操作仅需改变两个指针即可。时间复杂度是 *O*(1)。

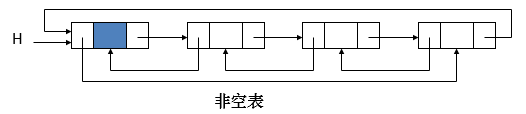
## 双向链表

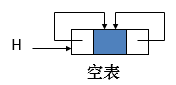
由于在单链表中，NextElem的执行时间为O(1)，PriorElem的执行时间为O(n)

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。

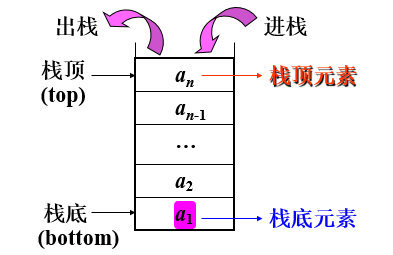
**顺序表、链表各自的使用场合？**

* 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

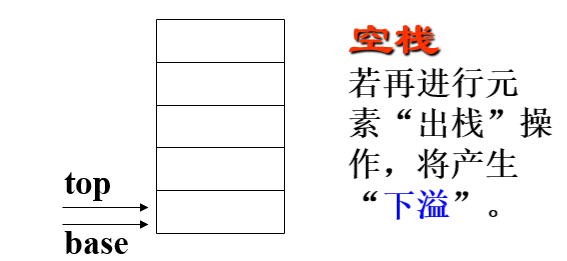
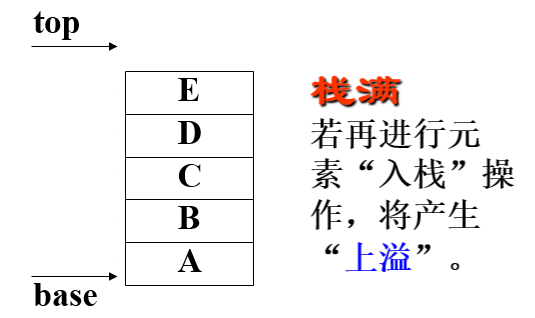
**Pop(&S, &*e*)：**删除 S 的栈顶元素，并用 *e* 返回其值

} 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)

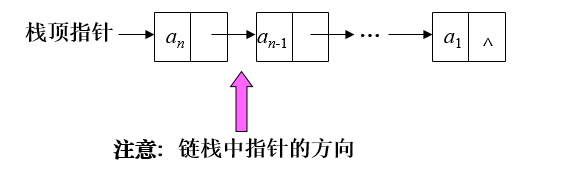
return false;

\*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

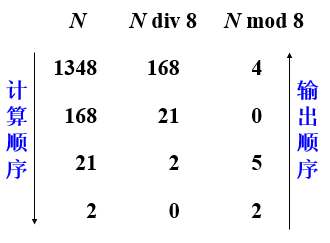
1. 数制转换：

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**；

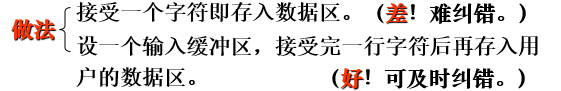
否则和栈顶元素比较，**若相匹配**，则“**左括号出栈**”，**否则表明不匹配**。

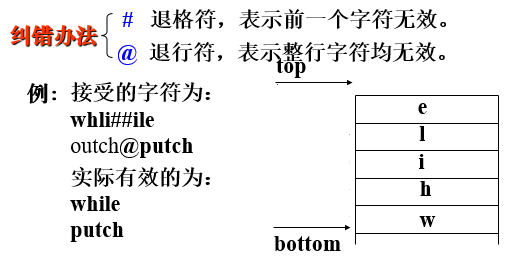
3）表达式检验结束时，**若栈空，则表明表达式中匹配正确**，否则表明“左括号”有多余的。

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。

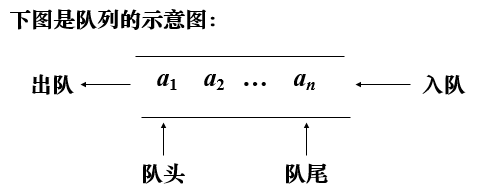
**多个函数嵌套调用的规则是：后调用先返回。**

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**

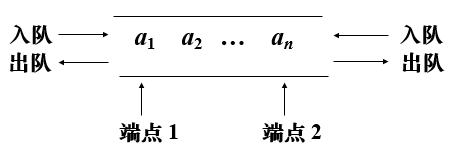
双端队列

**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。



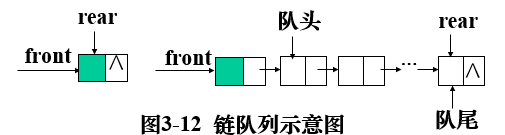
## 队列的表示和实现（链队列）

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;

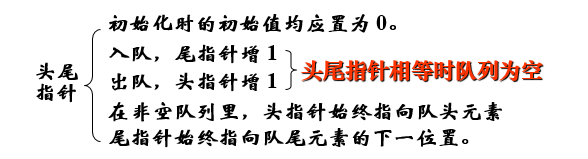
}

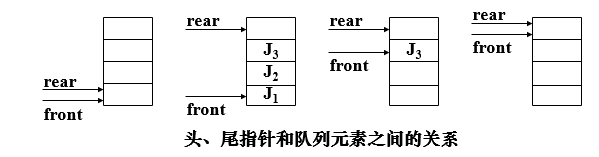
## 循环队列--队列的顺序表示和实现

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。





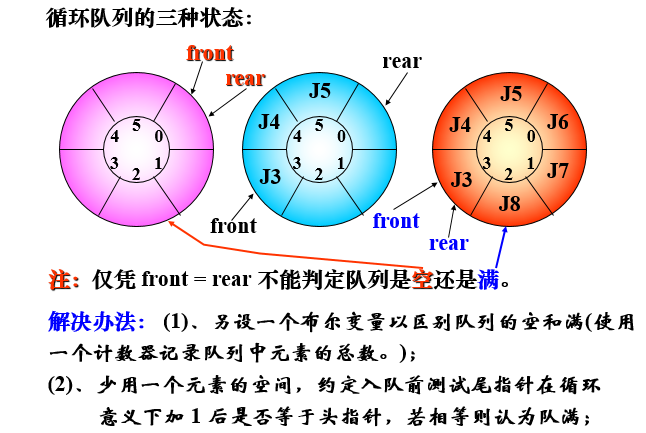
在顺序队列中，当尾指针已经指向了队列的最后一个位置的下一位置时，若再有元素入队，就会发生“溢出”

**“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--;

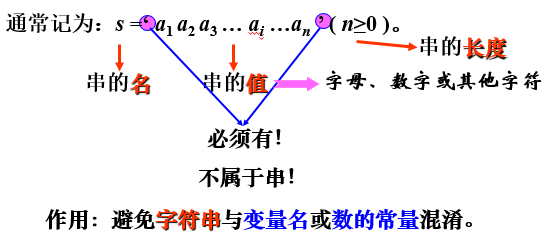
return true;

}

# 串

## 串的概念及其ADT定义

串（字符串）：是由 0 个或多个字符组成的有限序列。



* 例如：test = ’123’就是一个串，其中test是串的名，串的值为123，串的长度是3

空串：不含任何字符的串，长度 = 0，用符号 *φ* 表示。

空格串：仅由一个或多个空格组成的串。

子串：由串中任意个连续的字符组成的子序列。（子串可以是空串和它本身）

主串：包含子串的串。

位置：字符在序列中的序号。（位置的序号是从1开始数的）

子串在主串中的位置：子串的首字符在主串中的位置。



串相等的条件：当两个串的长度相等且各个对应位置的字符都相等时才相等。

例： S=‘JINAN’ S1=‘JI NAN’ S≠S1

串的抽象数据类型的定义：

**ADT String {**

**数据对象：**D＝{ *ai* |*ai*∈CharacterSet, *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：**

**StrAssign (&T, chars)**：将字符串常量chars赋为T的值

**StrCopy (&T, S)**

**DestroyString (&S)**

**StrEmpty (S)**：若 S 为空串，则返回 TRUE，否则返回 FALSE。

**StrCompare (S, T)**：若 S > T，则返回值 > 0；若 S = T，则返回值 = 0；若 S < T，则返回值 < 0。

说明：“串值大小” 是按 “词典次序” 进行比较的，如：

1. StrCompare(“data”, “stru”)<0 （由于对于第一个字符来说d<s）
2. StrCompare(“cat”, “case”)>0 （对于第三个字符来说s>t）

**StrLength (S)**

**Concat (&T, S1, S2)**：用 T 返回由 S1 和 S2 **联接**而成的新串。

**SubString (&Sub, S, pos, len)**：用 Sub 返回串 S 的第 pos 个字符起长度为 len 的子串

**Index (S, T, pos)**：若主串 S 中存在和串 T 值相同的子串，则**返回它在主串 S 中第 pos 个字符之后第一次出现的位置**；否则函数值为 0。

**Replace (&S, T, V)**：用 V 替换主串 S 中出现的所有与 T 相等的**不重叠**的子串。

假设：S=“abcacabcaca”， T=“abca” V=“ab”， 则置换之后的 S=“abcabca”， 而不是 “abbcaca”。（不重叠的子串）

**StrInsert (&S, pos, T)**

**StrDelete (&S, pos, len)**

**ClearString (&S)**

**} ADT String**

## 串的定长表示、基本操作实现

因为串是特殊的线性表，故其存储结构与线性表的存储结构类似，只不过组成串的结点是**单个字符**。

定长顺序存储表示，也称为**静态存储分配的顺序串**。即用一组地址连续的存储单元依次存放串中的字符序列。

**“定长”、“静态”**的意思可简单地理解为一个**确定的存储空间**，它的**长度是不变。**

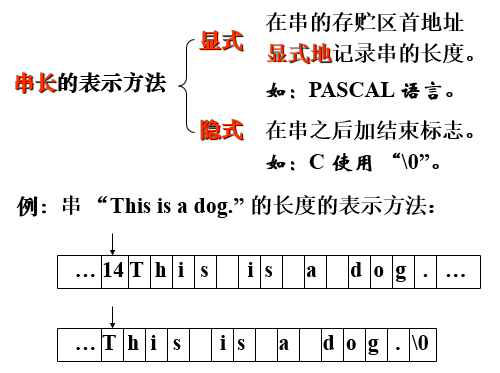
可直接使用定长的字符数组来定义一个串，数组的上界预先给出：

#define maxstrlen 255 // 可在 255 以内定义最大串长。

typedef unsigned char sstring[maxstrlen+1]; // 0 号单元存放串的长度。

sstring a; // sstring代表的是一个一维数组类型，里面存放的类型是unsigned char

串的实际长度可在这个预定义长度的范围内随意设定，超过预定义长度的串值则被舍去，称之为“截断”。



**定长顺序存储表示时串操作的缺点：**

1、需事先预定义串的最大长度，这在程序运行前是很难估计的。

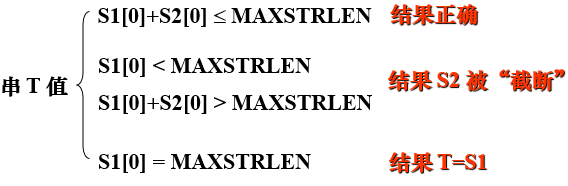
2、由于定义了串的最大长度，使得串的某些操作受限（截尾），如串的联接、插入、置换等运算。

克服办法：不限定最大长度——动态分配串值的存储空间。

定长顺序存储表示时串的操作的实现：

**1、串联接 Concat(&T, S1, S2)**

假设串 T 是由串 S1 联结串 S2 得到的，则只要进行相应的“**串值复制**”操作即可，需要时进行“截断”。



Status Concat(SString &T, SString S1, SString S2) {

if (S1[0]+S2[0] <= MAXSTRLEN) // 未截断

{ T[1...S1[0]] = S1[1...S1[0]];

T[S1[0]+1...S1[0]+S2[0]] = S2[1...S2[0]];

T[0] = S1[0]+S2[0]; uncut = TRUE; }

else

if (S1[0] < MAXSTRSIZE) // 截断

{ T[1...S1[0]] = S1[1...S1[0]];

T[S1[0]+1...MAXSTRLEN] = S2[1...MAXSTRLEN－S1[0]];

T[0] = MAXSTRLEN; uncut = FALSE; }

else // 截断(仅取S1)

{ T[0...MAXSTRLEN] = S1[0...MAXSTRLEN]; T[0] = MAXSTRLEN; uncut = FALSE; }

return uncut;

} // Concat

**2、求子串 SubString(&Sub, S, pos, len)**

求子串的过程即为复制字符序列的过程，将串 S 中的第 pos 个字符开始的长度为 len 的字符串复制到串 Sub 中。

注：1)不会出现“截断”的情况。 2)可能出现“参数非法”的情况，应返回 ERROR。

Status SubString(SString &Sub, SString S, int pos, int len)

{ if (pos < 1 || pos > S[0] || len < 0 || len > S[0]-pos+1)

return ERROR;

Sub[1…len] = S[pos…pos+len-1];

Sub[0]=len;

return OK;

} // SubString

## 串的堆分配表示、基本操作实现

**堆存储结构的特点：**仍以一组空间足够大的、地址连续的存储单元依次存放串值字符序列，但它们的存储空间是在程序执行过程中**动态分配**的。

通常，C 语言中提供的串类型就是以这种存储方式实现的。由动态分配函数 malloc() 分配一块实际串长所需要的存储空间（“堆”），如果分配成功，则返回此空间的起始地址，作为串的基址。由 free( ) 释放串不再需要的空间。

**用堆存放字符串时，其结构用 C 语言定义如下：**

typedef struct {

char \*ch; // 若非空则按串长分配存储区，否则 ch 为 NULL

int length; //串长度

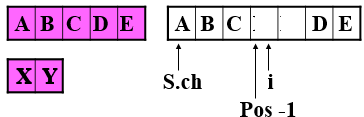
} HString;

串的堆分配的基本操作的实现：

**1. 串插入操作 StrInsert(&S, pos, T) 的实现算法为：**

为串 S 重新分配大小等于串 S 和串 T 长度之和的存储空间，进行串值的复制

**例：S=‘ABCDE’ T=‘XY’ pos=4**



**2. 串联接 Concat 算法描述**

将s1和s2进行串联接，然后将结果赋值给T。注意在这个过程中，如果T原来有内容需要先free(T)释放掉原来的空间

**3. 求子串 SubString 算法描述**

SubString(Hstring &Sub,Hstring S,int pos,int len);

在这个过程中需要注意两点：

（1）如果Sub.ch原来有内容，需要释放掉原来的内容

（2）如果len为NULL证明子串为空子串，空子串为任意串的子串，因此需要单独判断

**4. 求串复制 Strcopy 算法描述**

StrCopy(Hstring &T,Hstring S)：将S的值复制给T，注意free(T.ch)

**5. 串赋值 StrAssign 算法描述**

StrAssign(HString &T,char \*chars)：就是将常量chars复制给串T

在这个过程中要计算chars的长度

for(char c = chars;\*c;c++) i++;

**6. 串比较 StrCompare 算法描述**

int StrCompare(Hstring S,Hstring T)：若S>T 则返回>0 S=T 返回0 S<T 返回<0

**7. 清空串 ClearString 算法描述**

将S.ch置为NULL，并且长度置为0

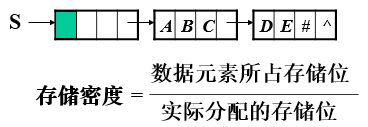
## 串的块链存储

串值也可用单链表存储，简称为链串。 链串与单链表的差异只是它的结点数据域为单个字符。



对于字符来说，char是1个字节，指针是4个字节，因此空间利用率极低。

为了提高**空间利用率**，可使每个结点存放**多个字符**（这是顺序串和链串的综合 (折衷) ），称为**块链结构**。



实际应用时，可以根据问题所需来设置结点的大小。例如：在编辑系统中，整个文本编辑区可以看成是一个串，每一行是一个子串，构成一个结点。即：同一行的串用定长结构（80个字符），行和行之间用指针相联接。

**结点结构用 C 语言定义如下：**

#define CHUNKSIZE 80 // 可由用户定义的块大小

typedef struct Chunk // 结点结构

{

char ch[CHUNKSIZE];

struct Chunk \*next;

} Chunk;

/\*为了便于进行串的操作（联接），当以块链存储串值时，除头指针外还可附设一个尾指针指示链表中的最后一个结点，并给出当前串的长度。其结构用C语言定义如下： \*/

typedef struct // 串的链表结构\_

{

Chunk \*head, \*tail; // 串的头和尾指针

int curlen; // 串的当前长度

} LString;

## 串的模式匹配算法

**模式匹配 ：**子串定位运算。(串匹配)就是在主串中找出子串出现的位置。

* 使用函数Index(S,T,pos) 在串匹配中，将主串S称为目标串，子串T称为模式串
* 如果在主串 S 中能够找到子串 T， 则称匹配成功，返回第一个和子串 T 中第一个字符相等的字符在主串S 中的序号；否则，称匹配失败，返回 0

例如：

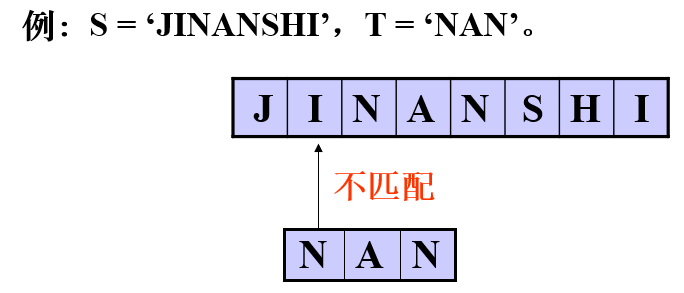
主串：ss=‘abcabcabdabcdef’; 子串：tt1=‘abc’;子串：tt2=‘abd’;

* Index(ss, tt1, 2) ; // 4
* Index(ss, tt1, 5) ; // 10
* Index(ss, tt2, 8) ; // 0

**模式匹配**是各种处理系统中最重要的操作之一，也是一个比较复杂的串操作。模式匹配的算法不同，效率将有很大差别。同一算法应用不同，效率亦有很大差别。

1. **朴素的模式匹配算法**

**算法思想：**从主串 S 的第 pos 个字符起和模式 T 的第一个字符比较之，若相同，则继续比较后续字符；否则从主串 S 的下一个字符起再重新和模式 T 的字符比较之。



当采用定长顺序存储结构时，实现此操作的算法如下：

int Index(SString S, SString T, int pos)

{ i = pos; j = 1;

while (i <= S[0] && j <= T[0])

{ if (S[i] == T[j])

{ ++ i; ++ j; } // 继续比较后继字符

else

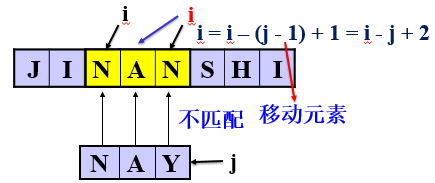
{ i = i – j + 2; j = 1; } // 指针后退重新开始匹配

}

if ( j >T[0]) return i -T[0];

else return 0;

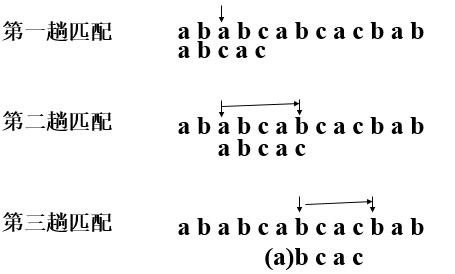
} // Index



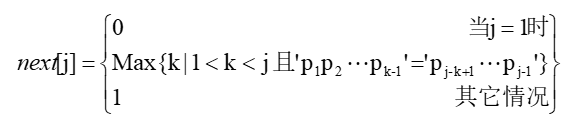
朴素的模式匹配算法评价

* 设计思想简单、易于理解。
* 通常情况下，效率比较高。经常被程序员选用。此时算法的时间复杂度为：O(n+m)。m、n分别为主串和子串的长度。
* 某些特殊的情况下，效率比较低。此时算法的时间复杂度为：O(n\*m)m、n分别为主串和子串的长度。（**比如计算机中00001 和00000000000000001模式匹配就是效率相当低的一种例子。**）
* 因此我们学习另一种高效的匹配算法----KMP算法。

2. **KMP算法**



若令next[j] = k, 则next[j]表明当模式中第j个字符与主串中相应字符“失配”时，在模式串中需重新和主串中该字符进行比较的字符的位置(k)。由此可以得出next函数的定义：



由此定义可以推出下列模式串的next函数值：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| j | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 模式串 | a | b | a | a | b | c | a | c |
| next[j] | 0 | 1 | 1 | 2 | 2 | 3 | 1 | 2 |

# 数组和广义表

## 数组的定义

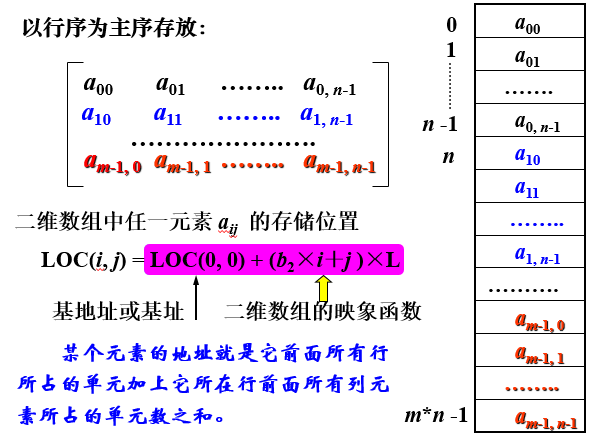
数组：按一定格式排列起来的具有相同类型的数据元素的集合

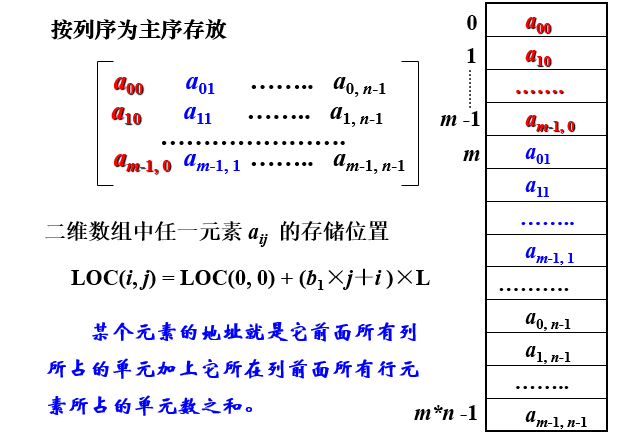
（数组结构又是线性表结构的扩展）

* 一维数组：若线性表中的数据元素为非结构的简单元素，则称为一维数组。
* 二维数组：若一维数组中的数据元素又是一维数组结构，则称为二维数组。
* 三维数组：若二维数组中的元素又是一个一维数组结构，则称作三维数组。
* *n* 维数组：若 *n* -1 维数组中的元素又是一个一维数组结构，则称作 *n* 维数组。

## 数组的顺序表示和实现

数组的存储方式包括两种：以行为主序进行存储和以列为主序进行存储





例：设数组 A[0…59, 0…69] 的基地址为 2048，每个元 素占 2 个存储单元，若以列序为主序顺序存储，则元素 A[31, 57] 的存储地址为

## 矩阵的压缩存储

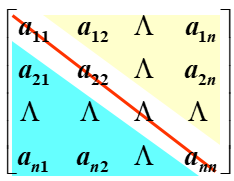
矩阵的常规存储是将矩阵描述为一个二维数组

* **不适宜**常规存储的矩阵：值相同的元素很多且呈某种规律分布；零元素多。
* 矩阵的压缩存储：为多个相同的非零元素只分配一个存储空间；对零元素不分配空间。

1. 特殊矩阵

（1）对称矩阵

对称矩阵上下三角中的元素数均为： ***n*(*n* + 1)/2**



（2）上（下）三角矩阵

三角矩阵的存储：除了存储主对角线及上（下）三角中的元素外，再加一个存储常数 *c* （大多数情况下这个常数是0）的空间

（3）对角线矩阵

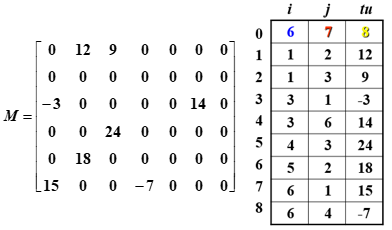
对角矩阵可按行优先顺序或对角线的顺序，将其压缩存储到一维数组中，且也能找到每个非零元素和向量下标的对应关系

2. 稀疏矩阵

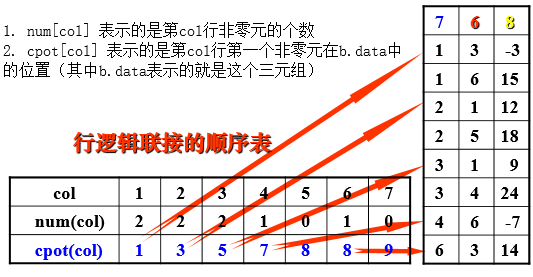
设在 *m*×*n* 的矩阵中有 *t* 个非零元素。 令*δ* =*t* /(*m*×*n*) 当 *δ* ≤0.05 时称为稀疏矩阵

**压缩存储原则：**存各非零元的值、行列位置和矩阵的行列数

（1）三元顺序表法

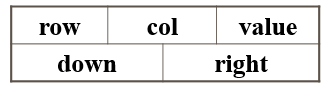


（2）行逻辑联接的顺序表（带行表的三元组）

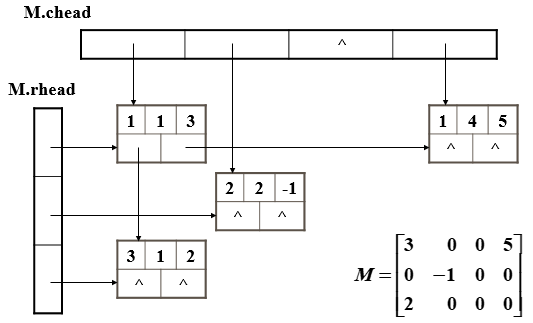


（3）稀疏矩阵的链式存储结构：十字链表

十字链表中结点的结构示意图：



* right：用于链接同一行中的下一个非零元素
* down：用以链接同一列中的下一个非零元素



## 广义表的概念、性质

广义表（又称列表 **Lists**）是***n*≥0**个元素 *a*1, *a*2, …, *an*的有限序列，其中每一个*ai* 或者是**原子**（单个元素），或者是一个**子表**

**广义表通常记作： *LS* = (*a*1，*a*2，…，*an*)**

其中： *LS* 为**表名**， *n* 为表的**长度**， 每一个 ***ai*** 为表的元素

习惯上，一般用大写字母表示广义表，小写字母表示原子

关于广义表要掌握以下几个操作：

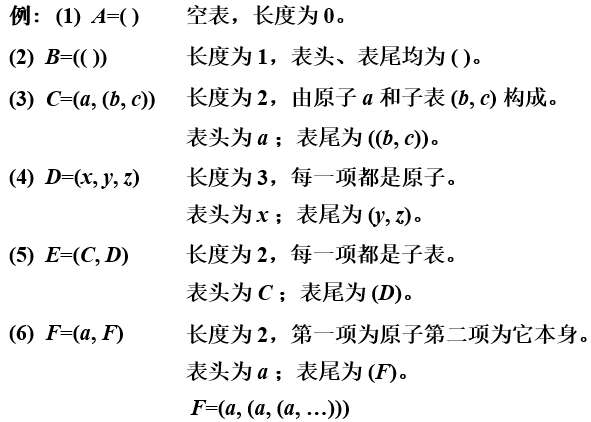
1. 求表头：*LS* 非空 (*n*≥1 )，则其第一个元素 ***a*1** 就是表头，表头可是原子，也可是子表

2. 求表尾：除表头之外的其它元素组成的表，表尾不是最后一个元素，而是一个子表

3. 求长度：最外层所包含元素的个数，例如：C=(a, (b, c)) 是长度为 2 的广义表。

4. 求深度：为该广义表展开后所含括号的重数

*A* = (*b*, *c*) 的深度为 1，*B* = (*A*, *d*) 的深度为 2，*C* = (*f*, *B*, *h*) 的深度为 3



## 广义表的存储结构

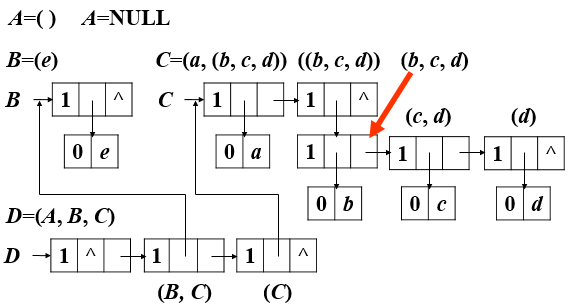
1. 首尾链表



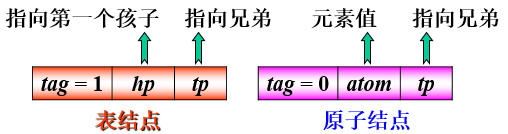
* 标志域 *tag* = 1，指示表头的指针域 *hp*，指示表尾的指针域 *tp*

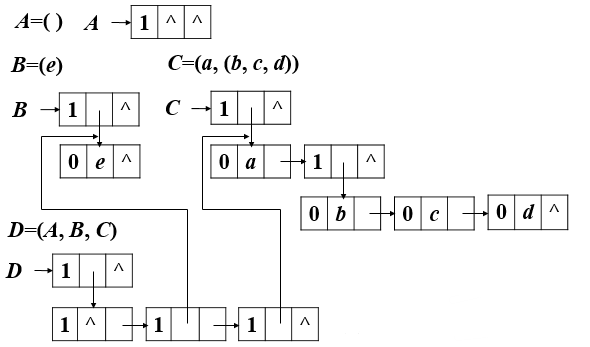


* 标志域 *tag* = 0，值域 *atom*



2．扩展线性链表（孩子兄弟链表）





# 树和二叉树

## 树和二叉树的相关概念、术语

1. 树的定义

树(Tree)是 *n* (*n*≥0)个结点的有限集。若 *n*=0，称为空树；若 *n* > 0，则它满足如下两个条件：

(1) 有且仅有一个特定的称为根 (Root) 的结点；

(2) 其余结点可分为 *m* (*m*≥0) 个互不相交的有限集 *T*1, *T*2, *T*3, …, *Tm*， 其中每一个集合本身又是一棵树，并称为根的子树 (SubTree)

2. 树的基本术语

根结点、叶子结点、孩子、双亲、兄弟、祖先、子孙、度（树的度是度的最大值）、深度

森林（树一定是森林，森林不一定是树）

3. 二叉树

二叉树是 *n* (*n*≥0) 个结点的有限集，它或者是**空集 (*n* = 0)**，或者由一个根结点及两棵**互不相交**的分别称作这个根的**左子树**和**右子树**的**二叉树**组成

**特点：**

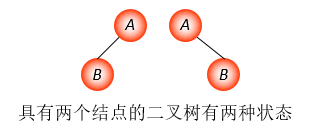
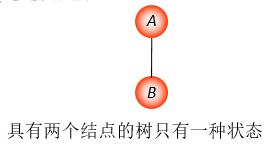
（1）每个结点最多有俩孩子 (二叉树中不存在度大于 2 的结点)

（2）子树有左右之分，其次序不能颠倒

（3）二叉树可以是空集合，根可以有空的左子树或空的右子树

**二叉树不是树的特殊情况，它们是两个概念。**

* 二叉树结点的子树要区分左子树和右子树，即使只有一棵子树也要进行区分，说明它是左子树，还是右子树。树当结点只有一个孩子时，就无须区分它是左还是右。因此二者是不同的。**这是二叉树与树的最主要的差别。**

## 二叉树的五个性质

**性质 1：** 在二叉树的第 *i* 层上至多有 2*i* - 1 个结点 (*i* ≥1)

**性质 2：**深度为 *k* 的二叉树至多有 2*k*－1 个结点（*k* ≥1)

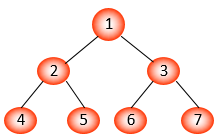
**性质 3：**对任何一棵二叉树 *T*，如果其叶子数为 *n*0，度为 2 的结点数为 *n*2，则 *n*0 = *n*2 + 1

**满二叉树 (Full binary tree)**

一棵深度为 *k* 且有 2*k*- 1 个结点的二叉树称为满二叉树。

**特点：（1）**每一层上的结点数都达到最大。**（2）叶子全部在最底层**。

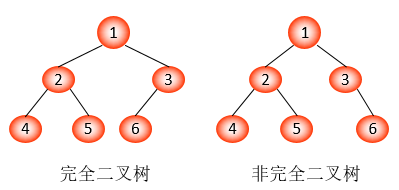
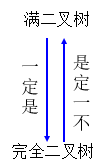
**编号规则：**从根结点开始，自上而下，自左而右。



**完全二叉树 (Complete binary tree)**

深度为 *k* 的具有 *n* 个结点的二叉树，当且仅当其每一个结点都与深度为 *k* 的满二叉树中编号为 1~ *n* 的结点一一对应时，称之为**完全二叉树**。

特点：（1）**叶子只可能分布在层次最大的两层上。** （2）**对任一结点，如果其右子树的最大层次为 L，则其左子树的最大层次必为 L 或 L+ 1**

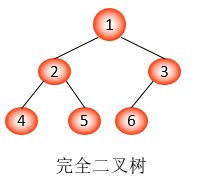
**性质 4：**具有 *n* 个结点的完全二叉树的深度为 ⎣log2*n*⎦ + 1

**性质 5：** 如果对一棵有 *n* 个结点的完全二叉树 (深度为⎣log2*n*⎦+1) 的结点按层序编号 (从第 1 层到第⎣log2*n*⎦+1 层，每层从左到右)，则对任一结点 *i* (1≤*i*≤*n*)，有：

(1) 如果 *i*=1，则结点*i*是二叉树的根，无双亲；如果 *i*>1，则其双亲是结点 ⎣*i*/2⎦

(2) 如果 2*i* > *n*，则结点 *i* 为叶子结点，无左孩子；否则，其左孩子是结点 2*i*

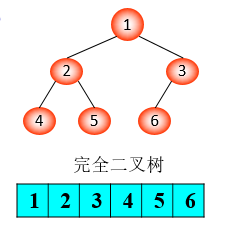
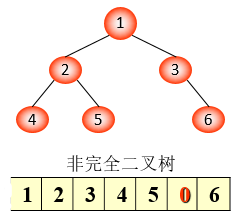
(3) 如果 2*i* + 1 > *n*，则结点 *i* 无右孩子；否则，其右孩子是结点 2*i* + 1



## 二叉树存储、遍历、线索

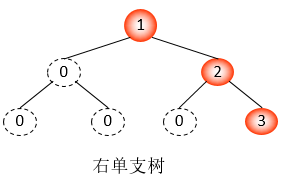
1. 二叉树的存储

（1）顺序存储结构

**此顺序存储结构仅适用于完全二叉树**

**最坏情况：**深度为 *k* 的且只有 *k* 个结点的右单支树需要长度为2*k*-1 的一维数组

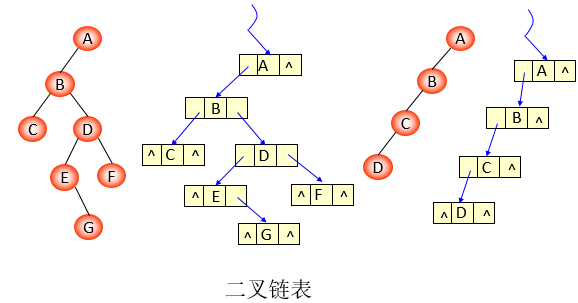


#define MAX\_TREE\_SIZE 100 // 二叉树的最大结点数

typedef TElemType SqBiTree[MAX\_TREE\_SIZE]; // 0 号单元存储根结点

SqBiTree bt;

（2）链式存储结构



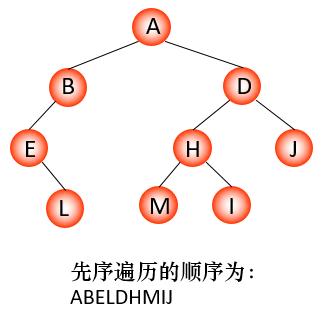
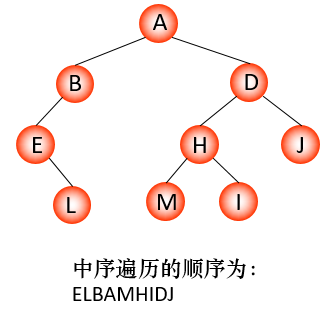
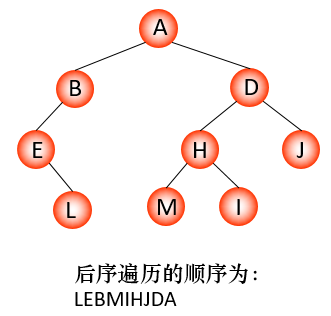
typedef struct BiTNode { // 结点结构

TElemType data;

struct BiTNode \*lchild, \*rchild; // 左右孩子指针

} BiTNode, \*BiTree;

2. 二叉树的遍历

3. 二叉树的相关操作

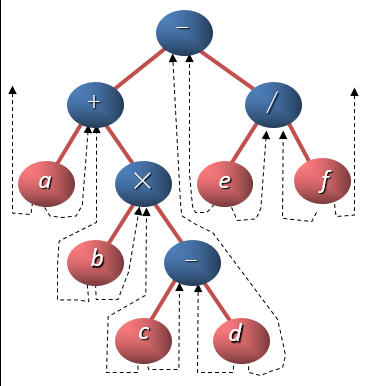
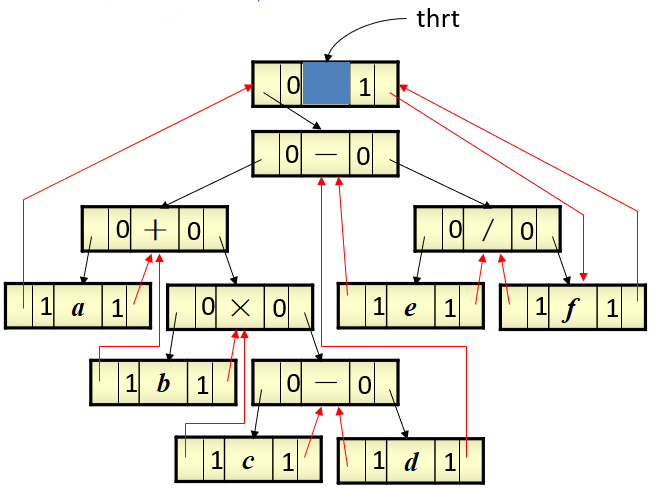
（1）遍历

（2）统计相关结点个数（叶子结点、所有结点）

（3）求深度

（4）建立二叉树

4. 二叉树的线索



在每一个节点中设置左右标记位，若标记为0，则代表是指针，指向的是结点，如果标记为1，则代表是线索，指示的是前驱元素或后继元素

其中为方便起见，在二叉树的线索链表上也添加了一个**头结点**，并另其lchild域的指针指向二叉树的根结点，其rchild的域的指针指向中序遍历是访问的最后一个结点；反之，令二叉树中序序列中的第一个结点的lchild域指针和最后一个rchild域指针均指向头结点

* **线索化：**对二叉树按照某种次序遍历使其变为线索二叉树的**过程**
* **二叉树线索化的目的：**利用线索化后的二叉树中的线索就可以直接找到某些结点在某种遍历序列中的前趋和后继结点

**在线索树（中序）中找结点前驱的方法：**

（1）若左链是线索，则直接指示前驱；

（2）若左链是指针，则“左孩中找最右”。 即：中序前驱左孩找右

**在线索树（中序）中找结点后继的方法：**

（1）若右链是线索，则直接指示后继；

（2）若右链是指针，则“右孩中找最左”。 即：中序后继右孩找左

**在线索树上进行遍历的方法：**

（1）从序列中的第一个结点起，依次找后继，直至后继为空

（2）从序列中的最后一个结点起，依次找前驱，直至前驱为空

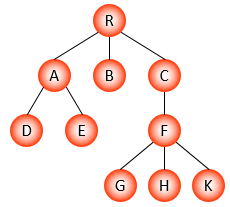
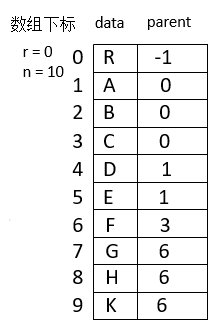
## 树和森林

1. 树的存储结构

（1）双亲表示法

**数据域：**存放结点本身信息

**双亲域：**指示本结点的双亲结点在数组中的位置

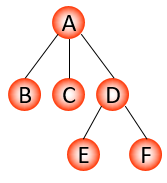
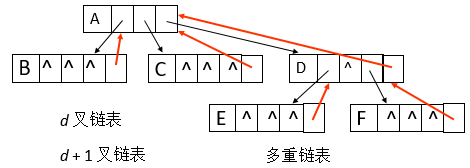
（2）孩子表示法

多重链表：即每一个节点有多个指针域，其中每个指针指向一棵字数的根节点

1）结点同构的多重链表

结点同构： 结点的指针个数相等，为树的度 *d*

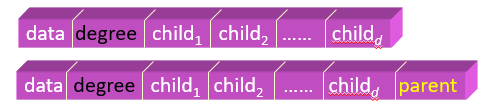


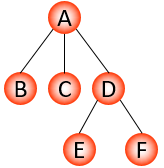
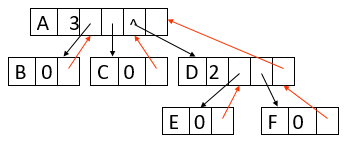
 

在有 *n* 个结点、度为*d* 的树的 *d* 叉链表中，有***n*×(*d*－1)＋1** 个空链域

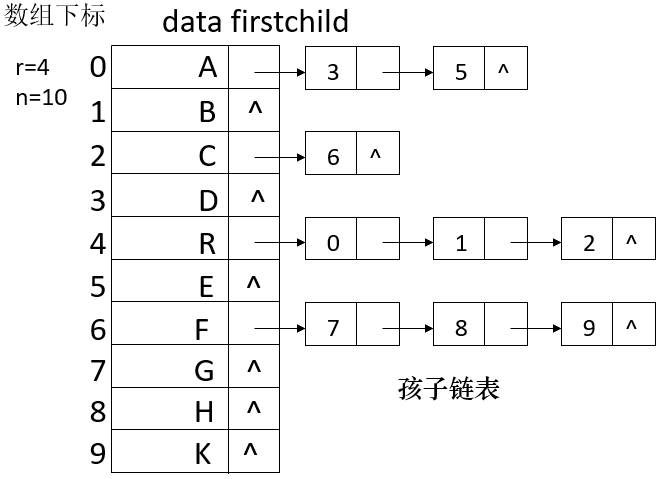
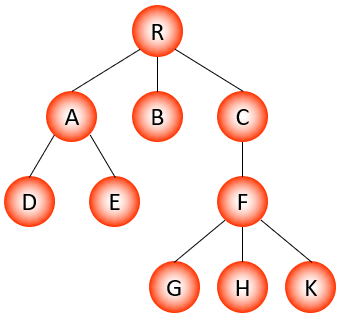
2）结点不同构的多重链表

结点不同构：结点的指针个数不相等，为该结点的度 **degree**。

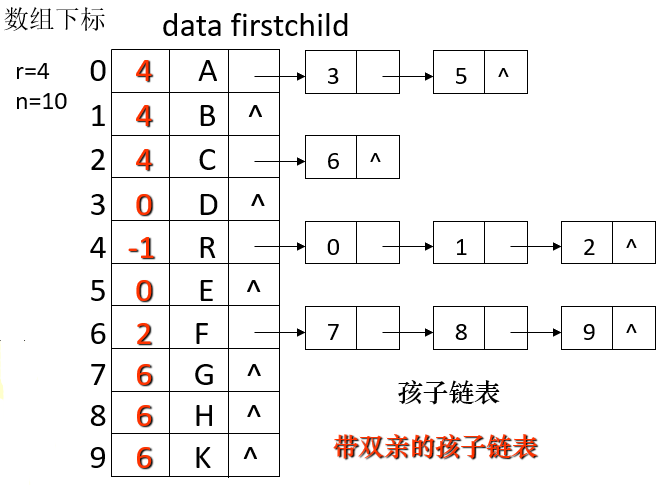


孩子链表：把每个结点的孩子结点排列起来，看成是一个线性表，用单链表存储，则 *n* 个结点有 ***n*** 个孩子链表**（叶子的孩子链表为空表）**。而 *n* 个头指针又组成一个线性表，用顺序表（含 *n* 个元素的结构数组）存储

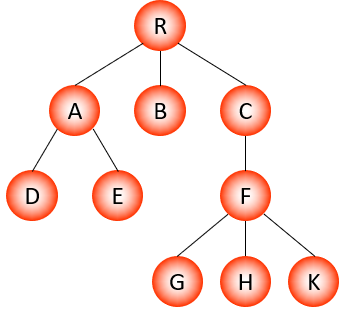
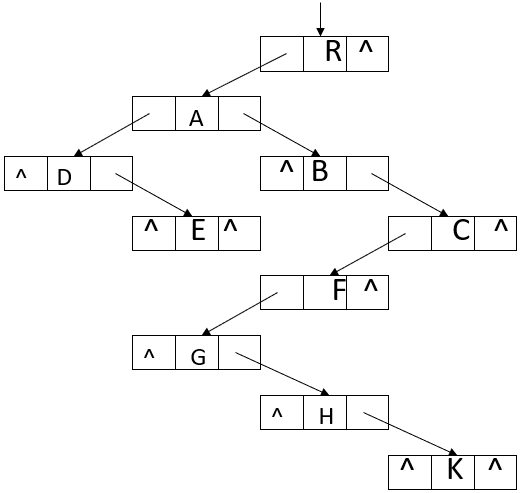


双亲表示法：通过上面两种方法，我们可以将双亲表示法和孩子表示法结合起来，即将双亲表示和孩子链表合在一起：



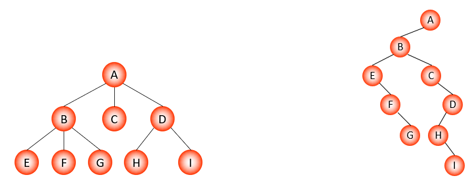
（3）孩子兄弟表示法（二叉树表示法，二叉链表表示法）

实现：用二叉链表作树的存储结构，链表中每个结点的两个指针域分别指向其**第一个孩子结点和下一个兄弟结点**

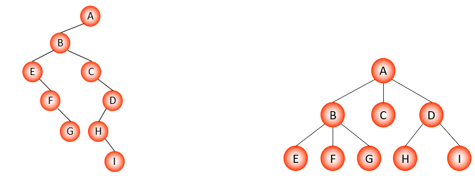
 

2. 森林和二叉树的转换

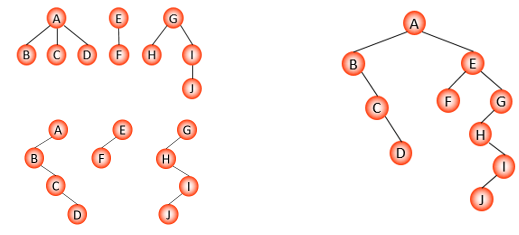
（1）树转换为二叉树（兄弟之间加上线、除长子之外抹去线）



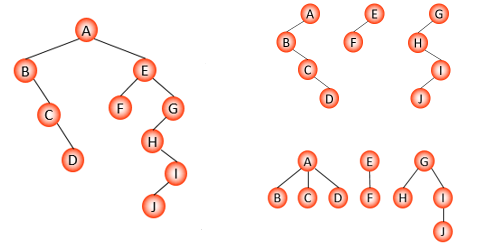
（2）二叉树转换为树（B与FG加线、EFG抹去线）



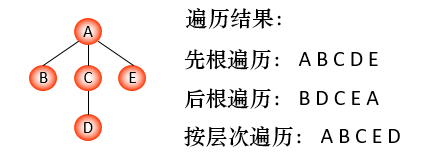
（3）森林转化为二叉树



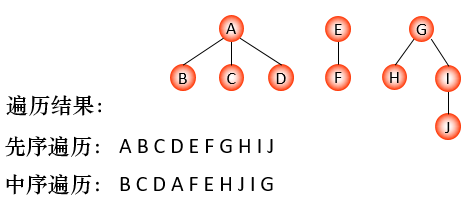
（4）二叉树转换为森林



3. 树的遍历



4. 森林的遍历

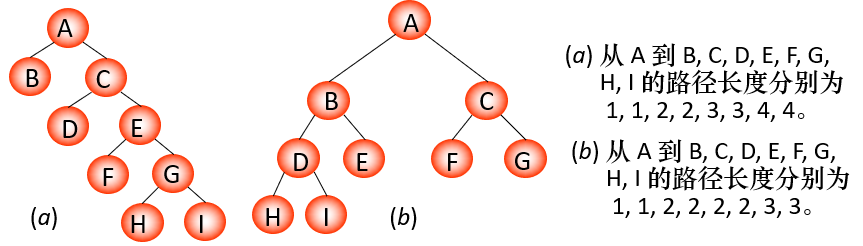


## 哈夫曼树

1. 基本概念

**路径**：从树中一个结点到另一个结点之间的**分支**构成这两个结点间的路径

**结点的路径长度**：两结点间路径上的分支数（主要考虑分支的数量）



**树的路径长度**：从树根到每一个结点的路径长度之和。记作：TL

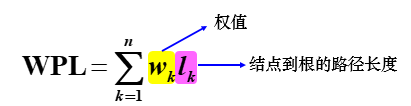
TL（*a*）＝0＋1＋1＋2＋2＋3＋3＋4＋4＝20

TL（*b*）＝0＋1＋1＋2＋2＋2＋2＋3＋3＝16

**权**：将**树中结点赋给一个有着某种含义的数值**，则这个数值称为该结点的权

**结点的带权路径长度**：从根结点到该结点之间的路径长度与该结点的权的乘积

**树的带权路径长度**：树中所有叶子结点的带权路径长度之和。记作：

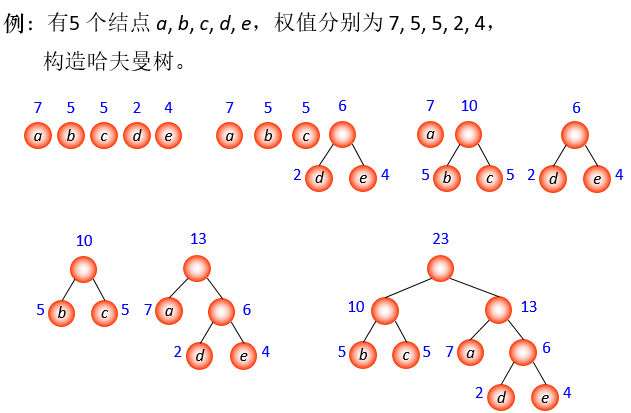


2. 哈夫曼算法（构造哈夫曼树的方法）

注意：

（1）哈夫曼树的结点的度数为 0 或 2， 没有度为 1 的结点。

（2）包含 *n* 个叶子结点的哈夫曼树中共有 2*n* – 1 个结点。

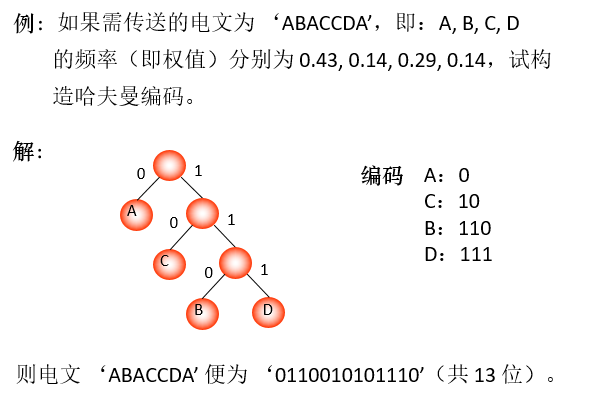


3. 哈夫曼编码

用二叉树设计二进制前缀编码

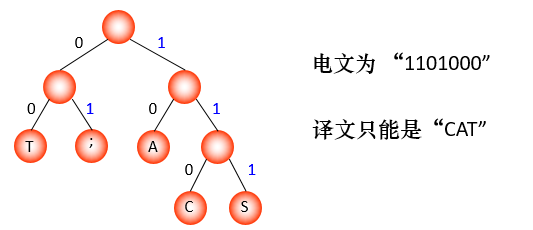
以电文中的字符作为叶子结点构造二叉树。然后将二叉树中结点引向其左孩子的分支标 ‘0’，引向其右孩子的分支标 ‘1’； 每个字符的编码即为从根到每个叶子的路径上得到的 0, 1 序列。如此得到的即为**二进制前缀编码**。

用哈夫曼树设计总长最短的二进制前缀编码



4. 译码

从哈夫曼树根开始，对待译码电文逐位取码。若编码是“0”，则向左走；若编码是“1”，则向右走，一旦到达叶子结点，则译出一个字符；再重新从根出发，直到电文结束



# 图

## 图的基本概念

1. 图的定义：图（Graph）是一种非线性结构，图中任意两个顶点之间都有可能有关联，顶点的前驱和后继结点无限制

2. 图的基本术语

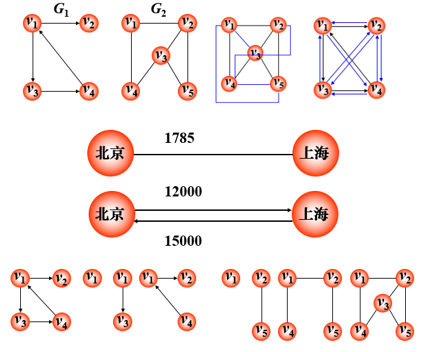
（1）顶点、弧（弧头、弧尾）、边

（2）有向图（含有弧）、无向图（边）、完全图、有向完全图

* 无向图中边的取值范围：0≤e≤n(n-1)/2
* 完全图：有n(n-1)/2条边的无向图（每两个顶点之间都存在着一条边)称为完全图
* 有向图中弧的取值范围：0≤*e*≤*n*(*n*-1)
* 有向完全图：有n(n-1) 条弧的有向图（每两个顶点之间都存在着方向相反的两条弧）称为有向完全图

（3）权、网（带权的图）

（4）子图，邻接点（有向图构成弧，无向图构成边）



（5）度（有向图的度=入度+出度、无向图的度就是所连接的结点的个数）、入度、出度

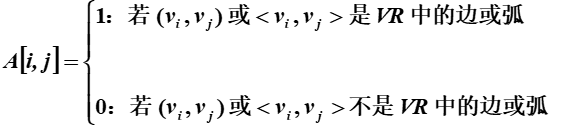
（6）路径、路径长度、连通（两个顶点有路径可到达）、连通图（图中任意两个顶点都是连通的）

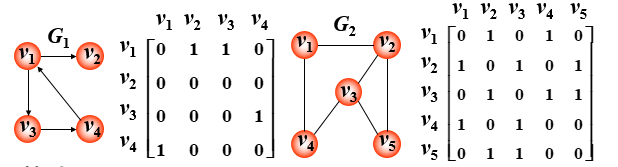
（7）生成树：所有顶点均由边连接在一起但不存在回路的图

## 图的存储结构

1. 邻接矩阵表示法

一个有 *n* 个顶点的图，可用两个数组存储。其中一个一维数组存储数据元素（顶点）的信息，另一个二维数组（邻接矩阵）存储数据元素之间的关系（边或弧）的信息。

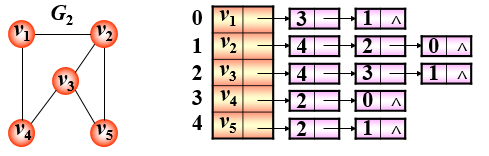




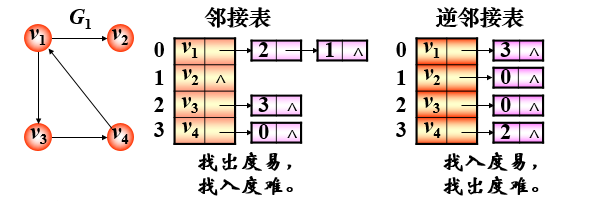
（1）无向图中顶点 *vi* 的度 *TD*(*vi*) 是邻接矩阵中第 *i* 行 1 的个数

（2）有向图中，顶点 *vi* 的出度是邻接矩阵中第 *i* 行 1 的个数，顶点 *vi* 的入度是邻接矩阵中第 *i* 列 1 的个数

2. 邻接表

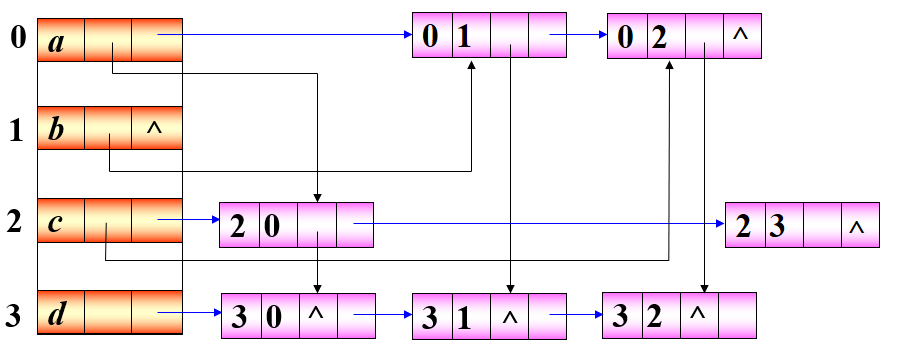




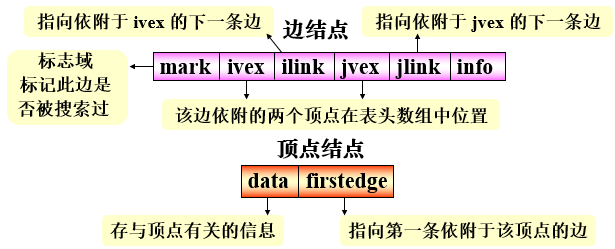


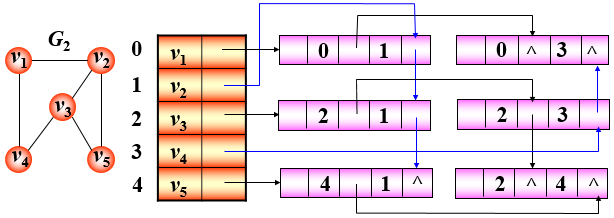
3. 十字链表（只能针对有向图而言）





4. 邻接多重表（只能针对无向图而言）





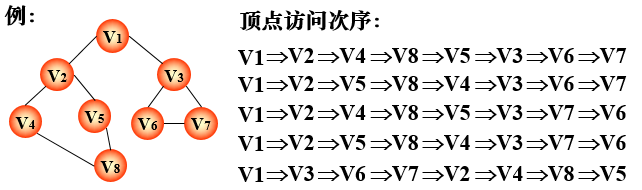
## 图的遍历

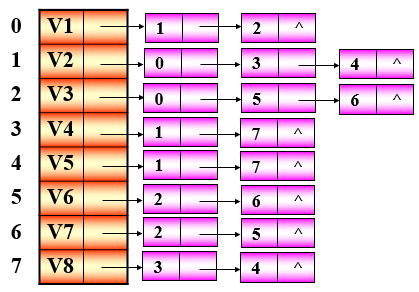
1．深度优先遍历（DFS）

（1）访问指定的起始顶点

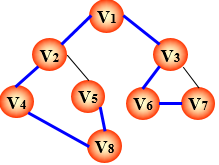
（2）若当前访问的顶点的邻接顶点**有未被访问的，则任选一个访问之**；**反之，退回到最近访问过的顶点**；直到与起始顶点相通的全部顶点都访问完毕

（3）若此时图中尚有顶点未被访问，则再选其中一个顶点作为起始顶点并访问之，转2；反之，遍历结束



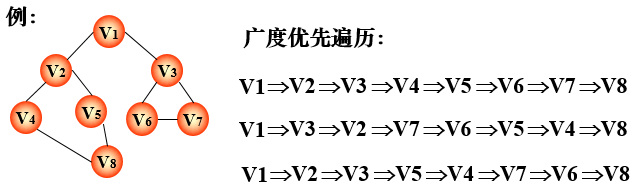


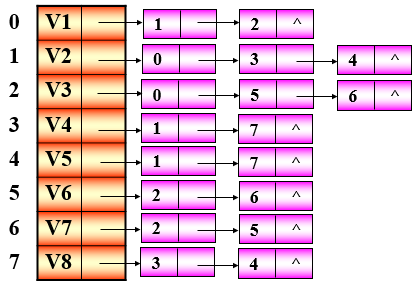
根据上述的存储结构示意图，画出图的结构并且能够写出深度优先遍历序列



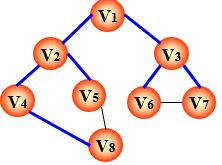
2. 广度优先遍历（BFS）

从图的某一结点出发，首先依次访问该结点的所有邻接顶点 V*i*1, V*i*2, …, V*in* **再按这些顶点被访问的先后次序依次访问与它们相邻接的所有未被访问的顶点**，重复此过程，直至所有顶点均被访问为止





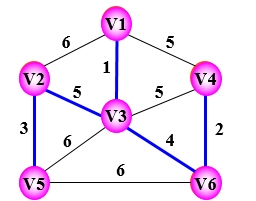
根据上述的存储结构示意图，画出图的结构并且能够写出广度优先遍历序列



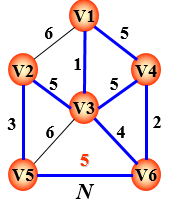
## 最小生成树

**最小生成树**：给定一个无向网络，在该网的所有生成树中，使得各边**权数之和最小**的那棵生成树称为该网的最小生成树，也叫最小代价生成树。

**方法一：普里姆 (Prim) 算法（从点出发）**



**方法二：克鲁斯卡尔 (Kruskal) 算法（从权值最小的除法）**



## 有向无环图及其应用

1. 拓扑排序

**AOV 网:**用一个有向图表示一个工程的各子工程及其相互制约的关系，其中**以顶点表示活动，弧表示活动之间的优先制约关系，称这种有向图为顶点表示活动的网**，简称AOV网

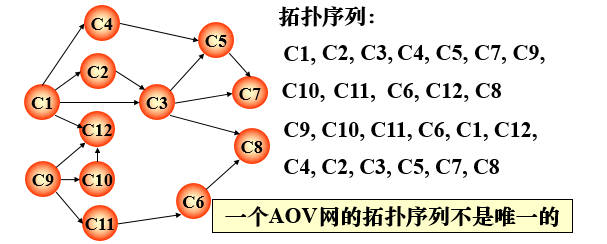
**AOV 网中不允许有回路**，因为如果有回路存在，则表明某项活动以自己为先决条件，显然这是荒谬的

**问题：如何判别 AOV 网中是否存在回路？**

**检测 AOV 网中是否存在环方法**：对有向图构造其顶点的拓扑有序序列，若网中所有顶点都在它的拓扑有序序列中，则该 AOV 网必定不存在环

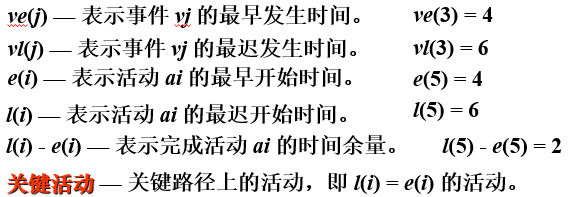
**拓扑排序的方法：**

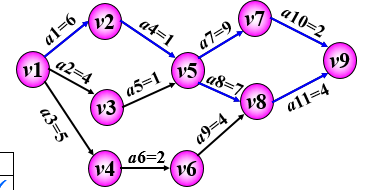
* 在有向图中选一个没有前驱的顶点且输出之
* 从图中删除该顶点和所有以它为尾的弧
* 重复上述两步，直至全部顶点均已输出；或者当图中不存在无前驱的顶点为止

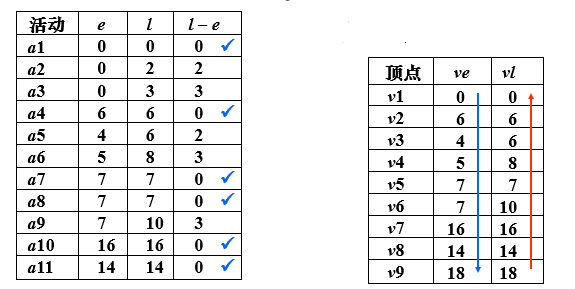


2. 关键路径

**求 *ve*(*i*)、*vl*(*j*) 求 *e*(*i*)、*l*(*i*) 计算 *l*(*i*) - *e*(*i*)**







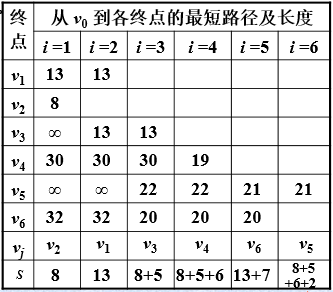
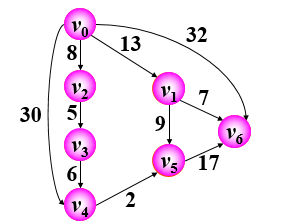
## 最短路径

1. 单源点的最短路径

迪杰斯特拉（Dijkstra）算法：按路径长度递增次序产生各顶点的最短路径。

最短路径的特点：

* 直接从源点到 *vi*<*v*0, *vi*>（只含一条弧）；（最开始的第一条弧即是在所有从源点出发的弧中查找权值最小者）
* 从源点经过已求得的最短路径上的顶点，再到达 *vi*（含有多条弧）

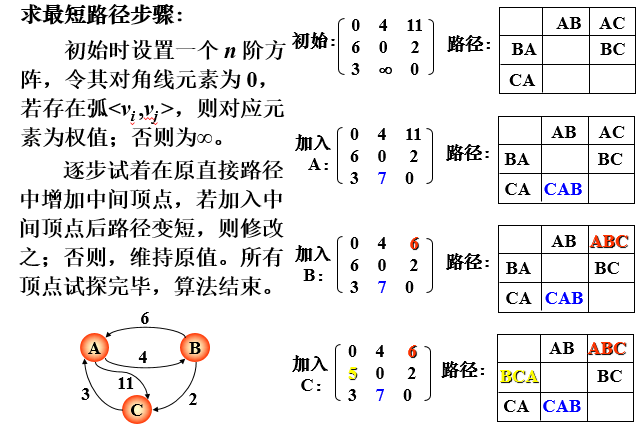


2. 每一对顶点之间的最短路径

方法一：每次以一个顶点为源点，重复执行 Dijkstra 算法 *n* 次。对于方法一来说总的执行时间是***o（n3）***

方法二：弗洛伊德 (Floyd) 算法

算法思想：逐个顶点试探，从 *vi* 到 *vj* 的所有可能存在的路径中，选出一条长度最短的路径。这个算法的时间复杂度也是***o（n3）***



# 查找

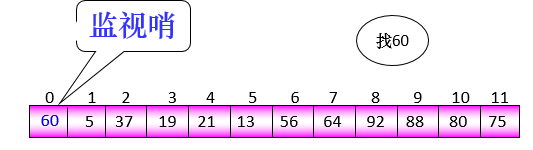
## 查找表的概念

* 查找表：由同一类型的数据元素(或记录)构成的集合。对查找表进行的经常操作为：查找、检索、增加、删除
* 静态查找表: 对查找表只进行前两种操作
* 动态查找表：不仅限于前两种操作

## 静态查找表

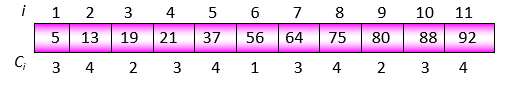
1. 顺序查找

设置监视哨的目的就是为了能够让每一次查找都能找到，若返回的是0，则代表查找不成功，若查找返回的结果不是0，则代表查找成功



平均查找长度（成功与不成功的平均查找长度之和）为 ASL =

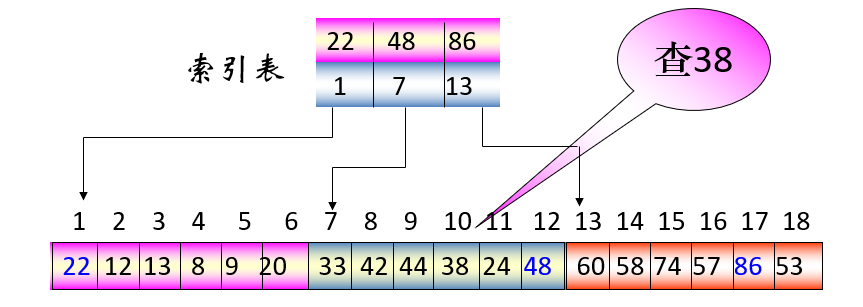
2. 折半查找



折半查找优点：效率比顺序查找高

折半查找缺点：只适用于有序表，且限于顺序存储结构

3. 索引查找（分块查找）



平均查找长度：比顺序查找好，比折半查找差

## 动态查找表

1. 二叉排序树的概念

二叉排序树或者是一棵空树；或者是具有下列性质的二叉树：

（1）若它的左子树不空，则左子树上所有结点的值均小于根节点的值

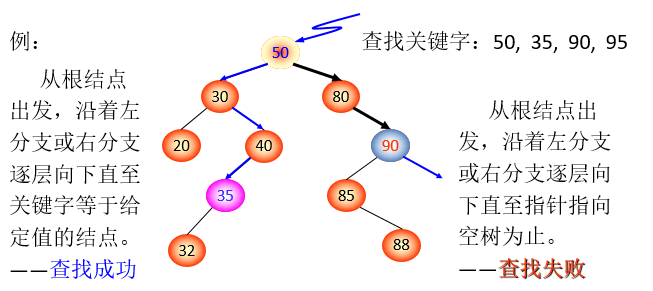
（2）若它的右子树不空，则右子树上所有结点的值均大于根节点的值

（3）它的左右子树也分别是二叉排序树

2. 二叉排序树的查找过程

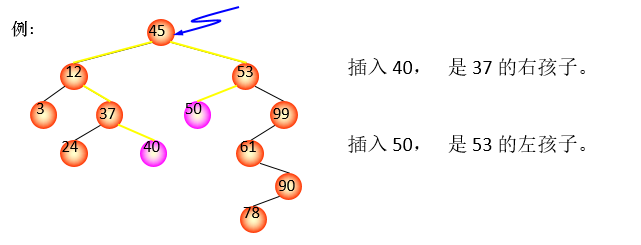
若二叉排序树为空，则查找不成功；否则

* 若给定值等于根结点的关键字，则查找成功
* 若给定值小于根结点的关键字，则继续在左子树上进行查找
* 若给定值大于根结点的关键字，则继续在右子树上进行查找



3. 二叉排序树的插入

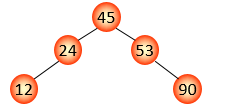
* 若二叉排序树为空树，则新插入的结点为**根结点**
* 若二叉排序树非空，则新插入的结点必为一个新的叶子结点，并且是**查找不成功时查找路径上访问的最后一个结点的左孩子或右孩子结点**



4. 二叉排序树的生成

从空树出发，经过一系列的查找、插入操作 之后，可生成一棵二叉排序树

例：设查找的关键字序列为 {45, 24, 53, 45, 12, 24, 90}，可生成二叉排序树如下：



5. 二叉排序树的删除

删除二叉排序树中的 \*p 结点，分三种情况讨论：

（1）\*p 为叶子结点，因删除叶子结点不破坏树的结构，只需要直接删除

（2）\*p 只有左子树或右子树：\*p 只有左子树，用 \*p 的左孩子代替 \*p；\*p 只有右子树，用 \*p 的右孩子代替 \*p

（3）\*p 左、右子树均非空：有三种方式可以操作（具体动画见ppt）

* 用 \*p 的直接前驱取代 \*p
* 用 \*p 的直接后继取代 \*p
* 用 \*p 的左子树取代 \*p

5. 平衡二叉树

如何提高形态不均衡的二叉排序树的查找效率？

* 做“平衡化”处理，即尽量让二叉树的形状均衡！

平衡二叉树又称 AVL 树，它是具有如下性质的二叉树：

* 左、右子树是平衡二叉树；
* **所有结点的左、右子树深度之差的绝对值≤ 1。**

为了方便起见，给每个结点附加一个数字 = **该结点左子树与右子树的深度差。**这个数字称为结点的**平衡因子**。这样，可以得到 AVL 树的其它性质（可以证明）：

任一结点的平衡因子只能取：**-1、0 或 1**；如果树中任意一个结点的平衡因子的绝对值大于 1，则这棵二叉树就**失去平衡**。

如果在一棵 AVL 树中插入一个新结点后造成失衡，则必须重新调整树的结构，使之恢复平衡。我们称此调整平衡的过程为**平衡旋转**。

（1）LL 平衡旋转



（2）RR 平衡旋转：



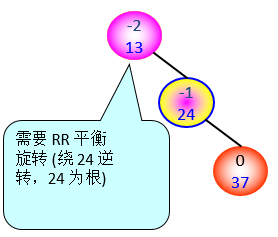
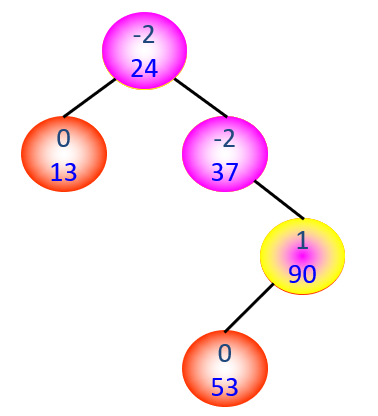
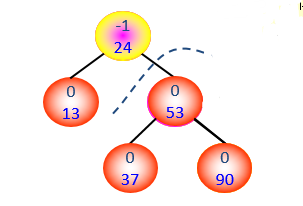
（3）LR 平衡旋转：



（4）RL 平衡旋转：



例：请将下面序列构成一棵平衡二叉排序树： (13, 24, 37, 90, 53)

6. B-树

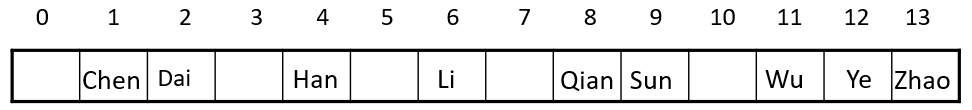
## 哈希表

记录的关键字与记录在表中的存储位置之间存在一种对应（函数）关系。若记录的关键字为 key，记录在表中的位置（称为哈希地址）为 *f* (key)，则称此函数*f* (x) 为哈希函数（散列函数）

例如：对于如下 9 个关键字：

{Zhao, Qian, Sun, Li, Wu, Chen, Han, Ye, Dai}

设哈希函数 *f* (key) = ⎣(Ord(关键字首字母) - Ord(‘A’) + 1) / 2⎦



1) 哈希函数是一个映像，即：将关键字的集合映射到某个地址集合上

2) 若添加关键字 Zhou，很容易产生“冲突”现象，即：key1 ≠ key2，而 *f* (key1) = *f* (key2)

因此：在构造这种特殊的“查找表”时，除了需要选择一个“好”的哈希函数之外；还需要找到一种“处理冲突”的方法（冲突不可能没有，只能减少冲突）

1. 哈希表的定义

根据设定的哈希函数H(key)和所选中的处理冲突的方法建立的查找表。以记录的关键字为自变量，根据哈希函数，计算出对应的哈希地址，并在此存储该记录的内容

2. 哈希函数的构造方法

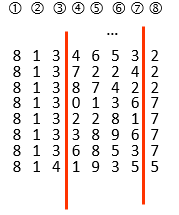
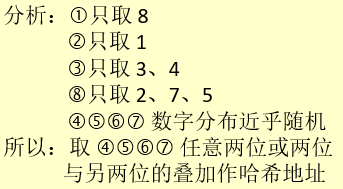
（1）直接定址法

哈希函数为关键字的线性函数H(key) = key 或者 H(key) = *a* × key + *b*

（2） 数字分析法（数字选择法）

构造：取关键字的若干位或其组合作哈希地址

例：有80个记录，关键字为8位十进制数，哈希表长100。则哈希地址可取2位十进制数

（3）平方取中法（较常用）

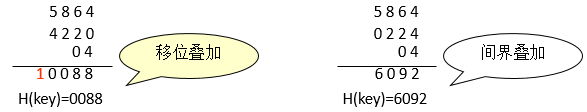
构造：以关键字的平方值的中间几位作为哈希地址

（4）折叠法

构造：将关键字分割成位数相同的几部分，然后取这几部分的叠加和（舍去进位）做哈希地址

* 移位叠加：将分割后的几部分低位对齐相加
* 间界叠加：从一端沿分割界来回折叠，然后对齐相加

例：关键字为：0442205864，哈希地址位数为 4



（5）除留余数法（最常用）

构造：取关键字被某个不大于哈希表表长 *m* 的数 *p* 除后所得余数作哈希地址，即 H(key) = key MOD *p* (*p* ≤ *m)*

（6） 随机数法

构造：取关键字的随机函数值作哈希地址，即：H(key) = Random(key)其中，Random 为伪随机函数

3. 处理冲突的方法

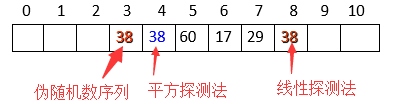
（1）开放定址法

当发生冲突时，在冲突位置的前后附近寻找可以存放记录的空闲单元。用此法解决冲突，要产生一个探测序列，沿着此序列去寻找可以存放记录的空闲单元。

对增量 *di*有三种取法：

* 线性探测再散列：*di* = 1, 2, 3, …, *m*-1
* 二次探测再散列（平方探测再散列）：*di* = 1², -1², 2², -2², 3², …, ±*k*² (*k* ≤ *m*/2)
* 伪随机探测再散列（双散列函数探测再散列）： *di* = 伪随机数序列

例：表长为 11 的哈希表中已填有关键字为 17,60,29 的记录，H(key)=key MOD 11，现有第 4 个记录，其关键字为 38，按三种处理冲突的方法，将它填入表中



（2）再哈希法

方法：构造若干个哈希函数，当发生冲突时，计算另一个哈希地址，即：H*i* = RH*i*(key) *i* =1, 2, …, *k* 其中：RH*i* —— 不同的哈希函数。

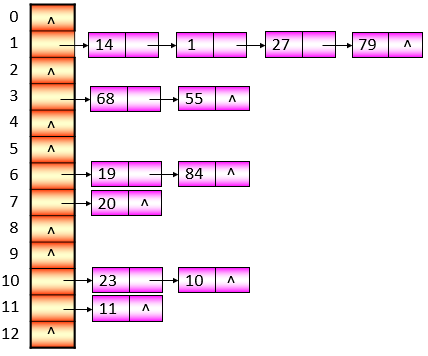
（3）溢出区法

除基本的存储区外（称为基本表），另外建立一个公共溢出区（称为溢出表），当发生冲突时，记录可以存入这个公共溢出区

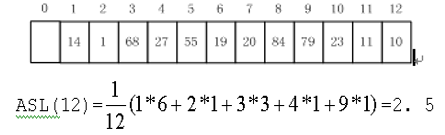
（4）链地址法

方法：将所有关键字为同义词的记录存储在一个单链表（同义词子表）中，并用一维数组存放头指针

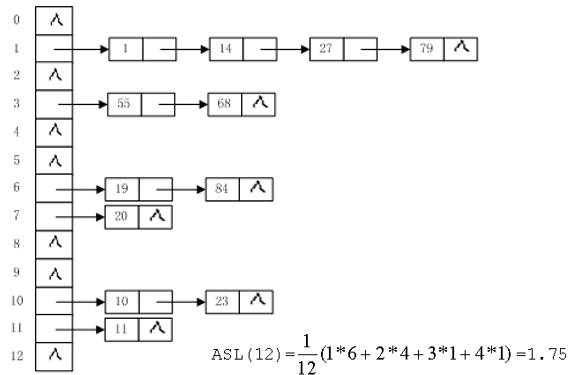
例：已知一组关键字 (19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79) 哈希函数为：H(key)=key MOD 13，用链地址法处理冲突



例如：一组关键字{19,14,23,1,68,20,84,27,55,11,10,79}按照哈希函数Hash(key)=key%13和线性探测再散列处理冲突得到的哈希表



例如：一组关键字{19,14,23,1,68,20,84,27,55,11,10,79}按照哈希函数Hash(key)=key%13和链地址法处理冲突得到的哈希表

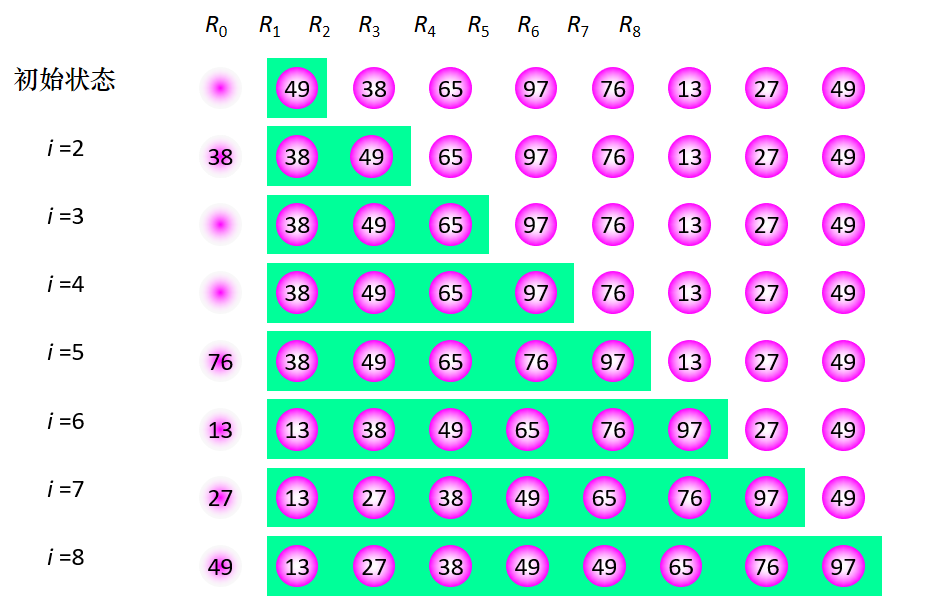


# 内部排序

* **排序：**是计算机程序设计中的一种重要操作，它的功能是将一个数据元素（或记录）的任意序列，重新排列成一个**按关键字有序**的序列。
* **内部排序:** 指的是**待排序记录存放在计算机随机存储器中进行的排序过程。**而外部排序指的是当排序记录的数量很大，以致内存一次不能容纳全部记录，在排序过程中尚需对**外存**进行访问的排序过程
* **排序的稳定性：**如果两记录Ri与Rj的关键字相同，在排序前Ri在Rj的前面，排序之后， Ri依然在Rj之前，我们称这种排序方法是**稳定的**，反之，是不稳定的。

## 插入排序

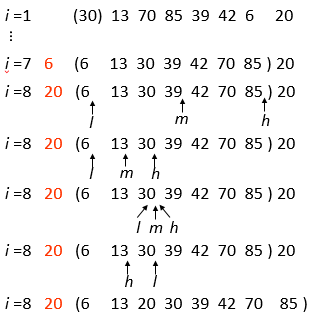
1. 直接插入排序



注意细节：在65和97进行直接插入排序的时候，**没有和r[0]进行交换**

算法分析：时间复杂度 O(n2) 空间复杂度 O(1) 稳定排序

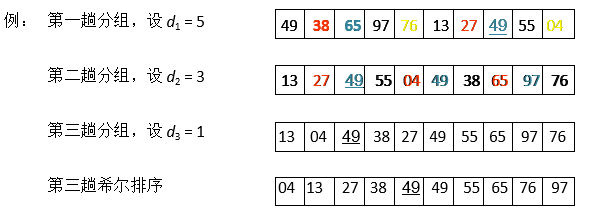
2. 折半插入排序



算法分析：时间复杂度 O(n2) 空间复杂度 O(1) 稳定排序

3. 希尔排序

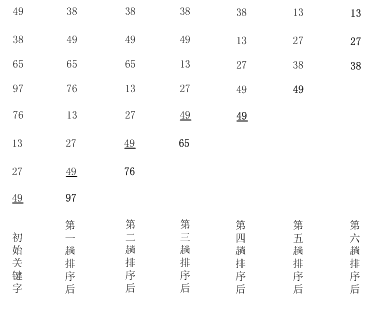
排序过程：先取一个正整数 *d*1 < *n*，把所有相隔 *d*1 的记录放在一组内，组内进行直接插入排序；然后取 *d*2 < *d*1，重复上述分组和排序操作；**直至 *di*=1**，即所有记录放进一个组中排序为止。其中 ***di* 称为增量**（一般取值5 3 1）



算法分析：时间复杂度 O(n1.3) 空间复杂度O(1) 非稳定性排序

## 交换排序

1. 起泡排序

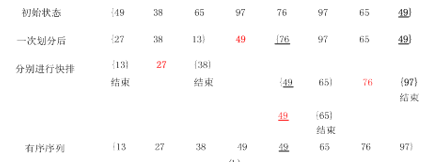


算法分析：时间复杂度 O(n2) 空间复杂度 O(1) 稳定排序

2. 快速排序

**基本思想：任选一个记录**，**（一般取第一个记录）**以它的关键字作为“枢轴”，凡关键字小于枢轴的记录均移至枢轴之前，凡关键字大于枢轴的记录均移至枢轴之后(枢轴要放在合适的位置上)

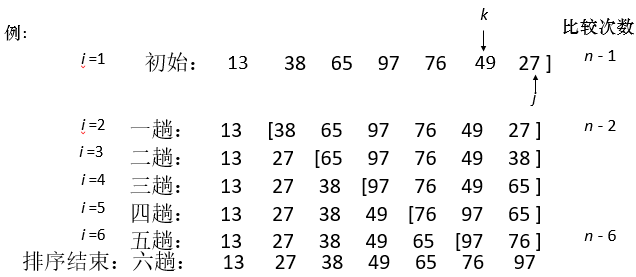
附设两个指针 low 和 high，从 high 所指位置起**向前搜索找到第一个关键字小于枢轴的关键字的记录与枢轴记录交换**，然后**从 low所指位置起向后搜索找到第一个关键字大于枢轴的关键字的记录与枢轴记录交换**，重复这两步直至 low = high 为止



算法分析：时间复杂度 O(nlogn)~O(n2) 空间复杂度 O(n)~O(logn) 非稳定性排序

## 选择排序

1. 简单选择排序



算法分析：时间复杂度 O(n2) 空间复杂度 O(1) 非稳定性排序

2. 堆排序

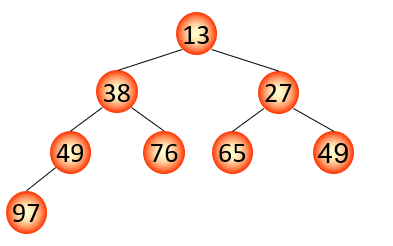
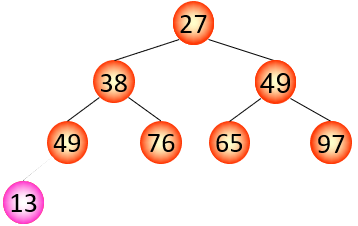
**堆排序：将无序序列建成一个堆**，得到关键字最小（大）的记录；输出堆顶的最小（大）值后，将剩余的 *n*-1 个元素重又建成一个堆，则可得到 *n* 个元素的次小值；如此重复执行，直到堆中只有一个记录为止，每个记录出堆的顺序就是一个有序序列，这个过程叫堆排序

**堆排序需解决的两个问题：**

1、如何由一个无序序列建成一个堆？（这个过程就是筛选的过程）

2、在输出堆顶元素后，如何将剩余元素调整为一个新的堆？

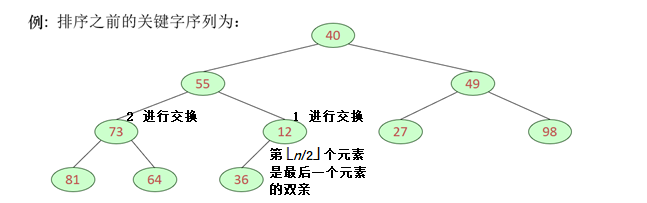
**输出堆顶元素之后，以堆中最后一个元素替代之**；**然后将根结点值与左、右子树的根结点值进行比较，并与其中小者进行交换；**重复上述操作，直至叶子结点，将得到新的堆，称这个从堆顶至叶子的调整过程为“筛选”。

对深度为 ***k*** 的堆，“筛选”所需进行的关键字比较的次数至多为 **2(*k*-1)**。

**堆排序—建堆**

从无序序列的第 **⎣*n*/2⎦** 个元素（即无序序列对应的完全二叉树的最后一个内部结点）起，至第一个元素止，进行反复筛选。

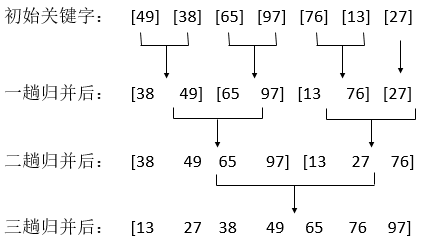


算法分析：时间复杂度O(nlogn) 空间复杂度O(1) 非稳定性排序

## 归并排序

归并：将两个或两个以上的有序表组合成一个新的有序表

在内部排序中，通常采用的是 **2-路归并排序**。即：将两个位置相邻的记录有序子序列归并为一个记录有序的序列。



算法分析：时间复杂度O(nlogn) 空间复杂度O(n) 稳定排序

## 基数排序

以静态链表存储待排记录，并令表头指针指向第一个记录

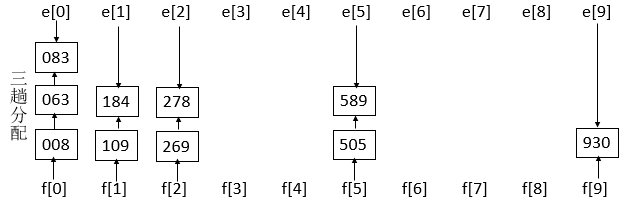


一趟收集：按当前关键字位取值从小到大将各队列首尾相链成一个链表



二趟收集：





三趟收集：



## 各种排序方法比较

