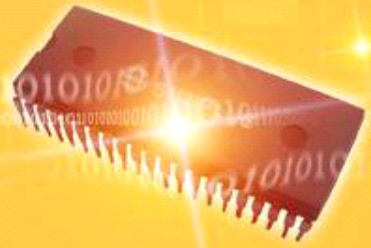
凌阳教育
www.sunplusedu.com

嵌入式电子工程师



值得信赖的教育品牌

大纲

- 概述
- 线性表
- 栈和队列
- 树与图
- 查找与排序

概述

概述

- 著名计算机科学家沃思(Nikiklaus Wirth)提出：程序 = 数据结构 + 算法。
- 数据结构：描述数据的类型和组织形式
- 算法：描述对数据的操作步骤
- 数据结构与算法：大多时候用的是旧知识新思想

概述

- 数据结构
  - 指的是计算机内部数据的组织形式和存储方法，或者说是相互之间存在一种或多种特定关系的数据元素的集合。
  - 线性结构：主要包括顺序表、链表、栈、队列等，最简单的如数组。
  - 树结构：人工智能中的“博弈树”，商业智能中的“决策树”，多媒体技术中的“哈夫曼树”等。
  - 图结构：各元素之间具有复杂对应关系的数据结构，如神经网络系统，贝叶斯网络等。

概述

- 算法
  - 是对特定问题求解步骤的一种描述或求解问题的策略，它是指令的有限序列。
  - 特征:
    - 有穷性: 执行指令的时间和次数是有限的
    - 确定性: 每一指令有确切的含义，无二义
    - 可行性: 每一操作都可以通过已经实现的基本运算，执行有限次来实现
    - 输入: 0个或多个输入
    - 输出: 能产生1个或多个输出

线性表

线性表

- 顺序表
  - 用一组连续地址的内存单元来存储整张数据表信息，这种存储结构下的线性表就叫做顺序表。
  - 特征:
    - 有唯一一个表名标识该顺序表
    - 占据一块连续的内存单元
    - 数据顺序存放，元素之间有先后关系

线性表

- 顺序表
  - 学生信息表，如图：

```
struct stu
{
    char name[10];
    int number;
    char sex;
    int age;
};
struct stu student[MAX_INFO];
```

student

姓名	学号	性别	年龄	
zs	101	S	23	0000H
ls	102	S	24	0018H
ww	103	S	25	0030H
xm	104	S	23	……
mh	105	F	22	

线性表

- 顺序表
  - 静态顺序表：容量固定，方法如定义一个数组的方法类似。
  - 初始化如下：

```
#define MAXSIZE 100
typedef struct stu ElemType ;
ElemType SQlist[MAXSIZE];
int len;
```

线性表

- 顺序表
  - 动态顺序表：容量可以动态追加，定义采用malloc()，追加采用realloc()。

```
void *realloc(void *mem_address,
unsigned int newsize);
```

功能：

先按照newsize指定的大小分配空间，将原有数据从头到尾拷贝到新分配的内存区域，而后释放原来mem_address所指内存区域，同时返回新分配的内存区域的首地址。即重新分配存储器块的地址。

返回值：

如果重新分配成功则返回指向被分配内存的指针，否则返回空指针NULL。

线性表

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int * a,i=0,j=0;
    a=malloc(5*4);//申请20个字节 每个整型元素4个字节
    while (1)
    {
        if(i>=5)
            a=(int *)realloc(a,(i+1)*4);
        printf("please input a int press 0 exit\n");
        scanf("%d",&a[i++]);
        getchar();
        if(a[i-1]==0)
            break;
    }
    for(j=0;j<i;j++)
        printf("%d\t",a[j]);
    printf("\n");
    return 0;
}
```

线性表

- 顺序表
  - 多维数组
    - 可以理解成，不同的维数就是不同的构造类型，占用不同的内存空间，即，内存偏移量不同，如:
    - 一维数组: (a[i])，i个元素，一条线性表,*(a+i);
    - 二维数组: (a[i][j])，i行j列，共i*j个元素，一张行列矩阵二维线性表,*(*a+i);
    - 三维数组: (a[i][j][k])，共i*j*k个元素，一个立方容器，即i张矩阵二维表组成,*(*(*a)+i);
    - 四维数组: (a[i][j][k][m])，共i*j*k*m个元素，可以认为是i个立方容器组成,*(*(*(*a))+i);

线性表

- 顺序表
  - 不管是二维还是三维还是多维数组，都可以定义一个一维指针来指向
  - 因为多维数组是顺序存储结构，所以最终会退化成一维指针指向，并通过*(p+i)来遍历数组
  - 其实即使你定义多维指针来指向时，她还是会退化成一维的
  - 为了体现多维特点也可以定义成int(*p)[][][]...=str ，但元素访问时变得很麻烦，要一级一级取地址直至取到内容为止

线性表

- 顺序表
  - 但也有另一个用处，就是可以像数组名一样通过下标进行数据访问，所以在程序设计中可以根据用户自己的实际需求进行定义

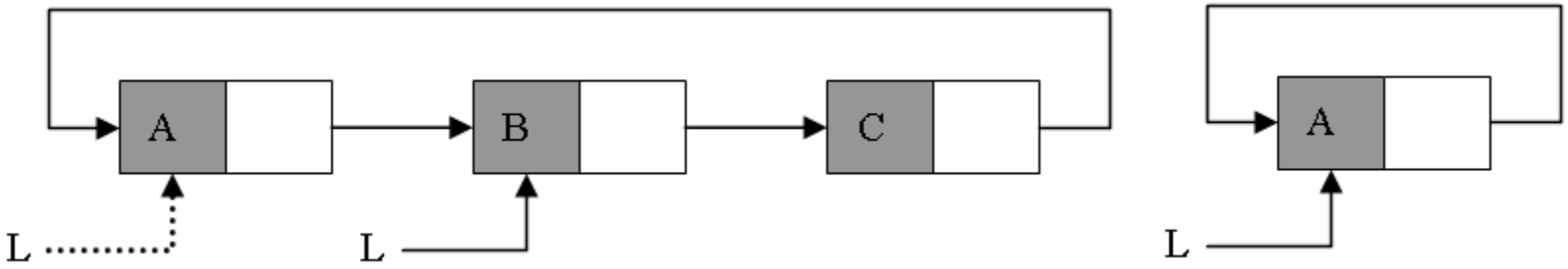
二维数组：			
int str[2][3]	*(p+i)	*(*(p)+i)	p[i][j]
int(*p)[3]=str	×	√	√
int *p=str	√	×	×
三维数组：			
int str[2][3][4]	*(p+i)	*(*(*(p))+i)	p[i][j][k]
int(*p)[3][4]=str	×	√	√
int *p=str	√	×	×

线性表

- 顺序表
  - 由此我们可以看到:
    - 静态顺序表不便于升级和维护。
    - 动态顺序表插入或删除成员时是低效的，且不适合用于复杂的数据结构中。
- 链表
  - 链表作为动态内存管理的一种经典方式，利用这种管理方式可以用在很多复杂的数据结构中，算得上是一种常用基本数据结构。

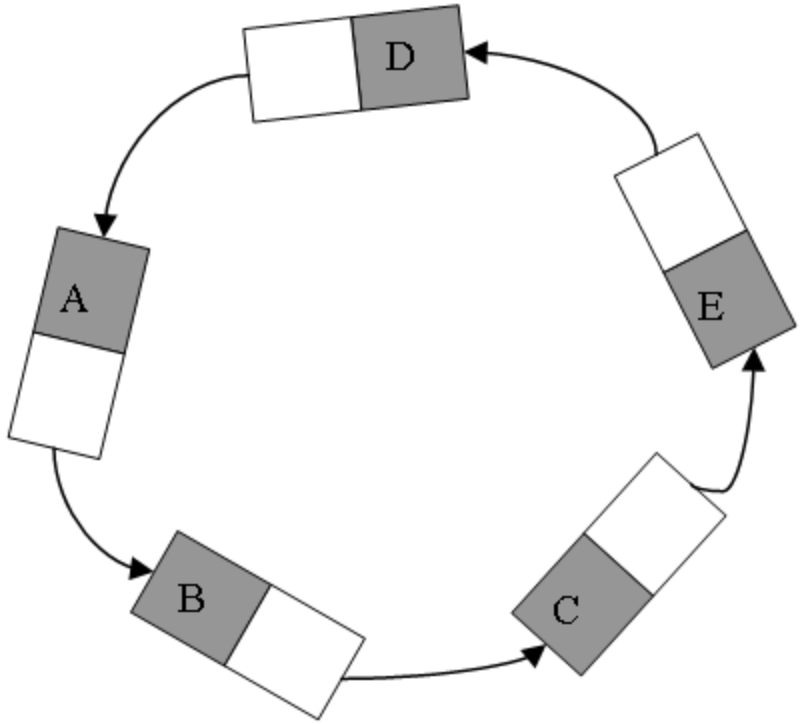
线性表

- 链表
  - 单向循环链表
    - 它的特点是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。
    - 由此，从表中任一结点出发均可找到表中其他结点。



线性表

- 链表
  - 单向循环链表



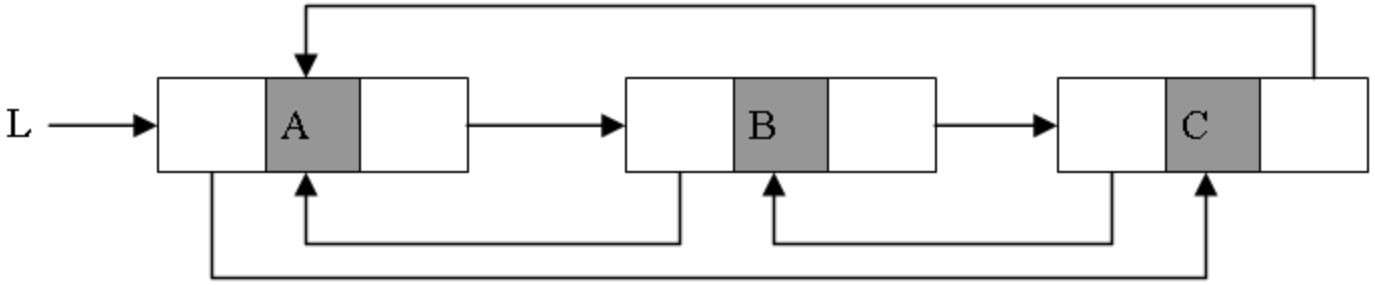
线性表

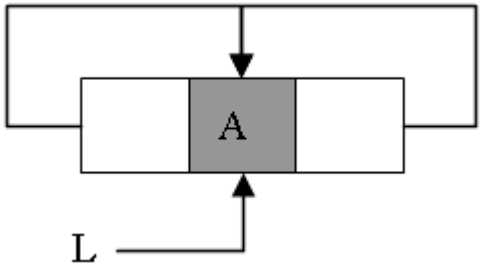
- 链表
  - 双向循环链表
    - 以上讨论的链表都只能从某个结点出发，顺着指针往一个方向寻查其它结点，为了克服这种单向性的缺点，于是又加入了一个指针域，用于索引另一个向方上的结点。
    - 结点数据结构：

```
typedef struct node
{
    int num;
    int age;

    struct node *prior;
    struct node *next;
}TYPE;
```

线性表

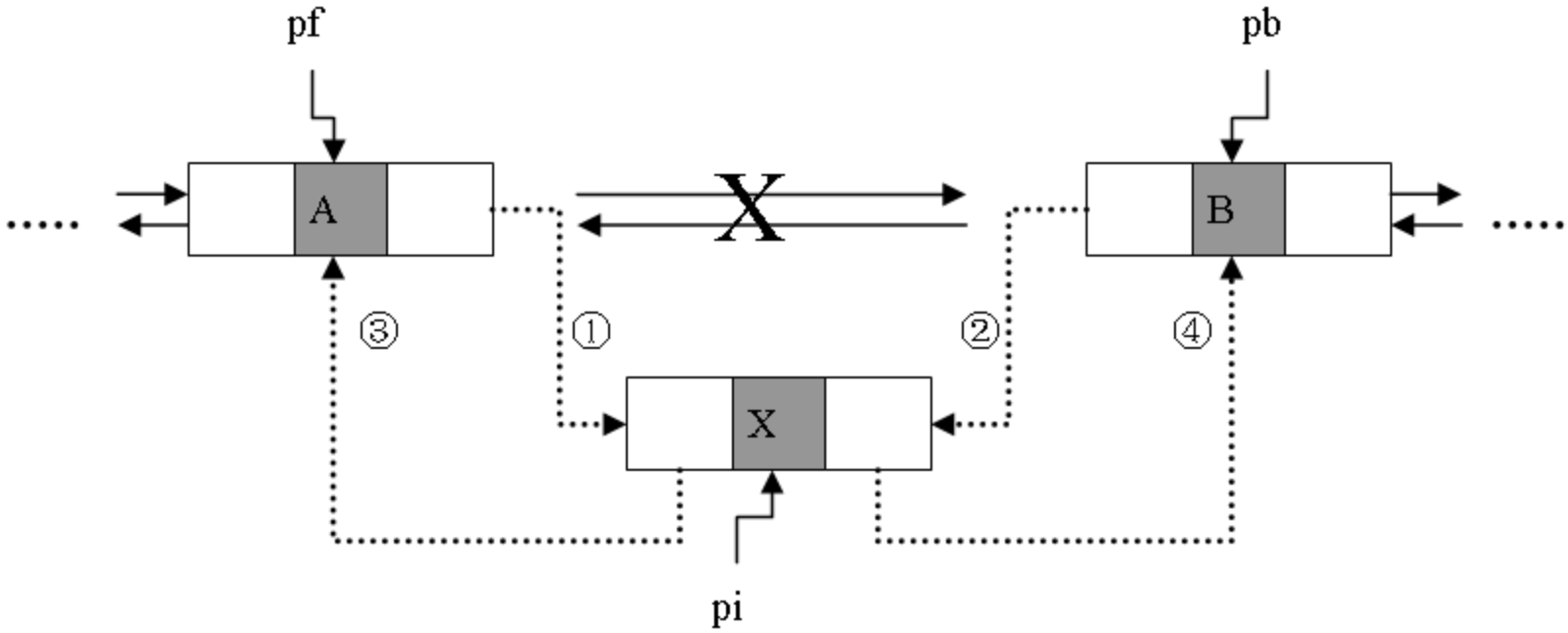
- 链表
  - 双向循环链表





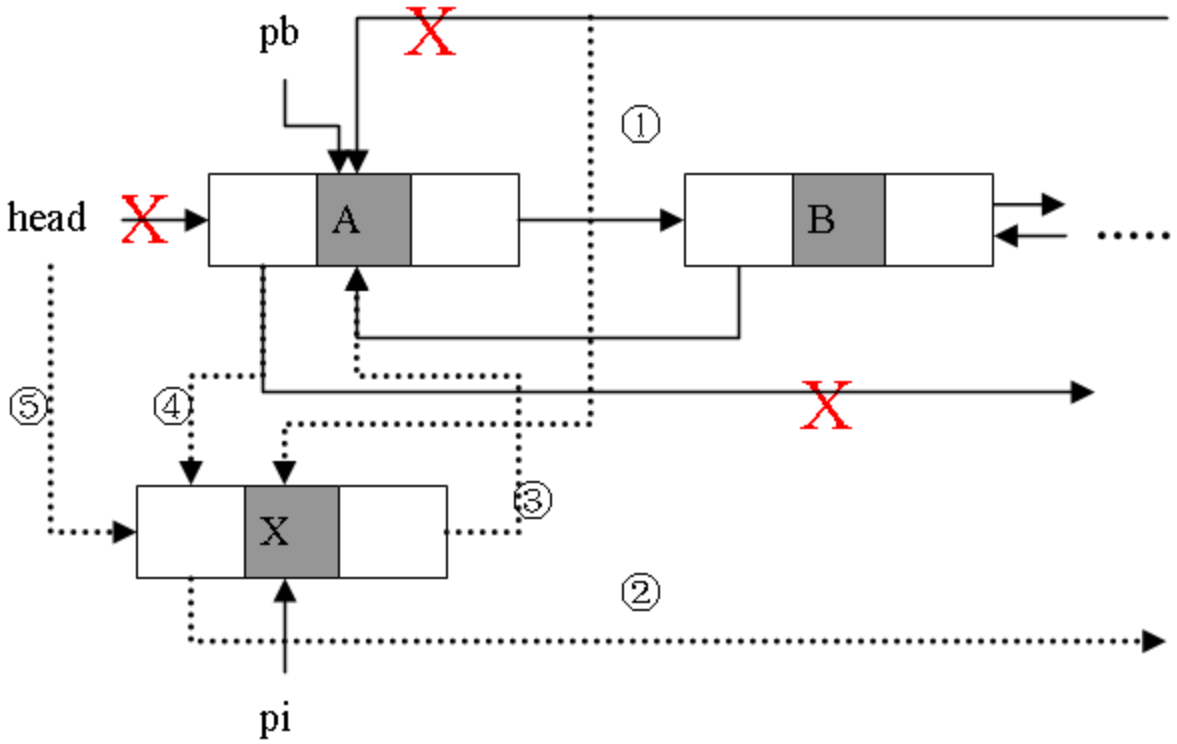
线性表

- 链表
  - 在双向链表中间插入一个结点



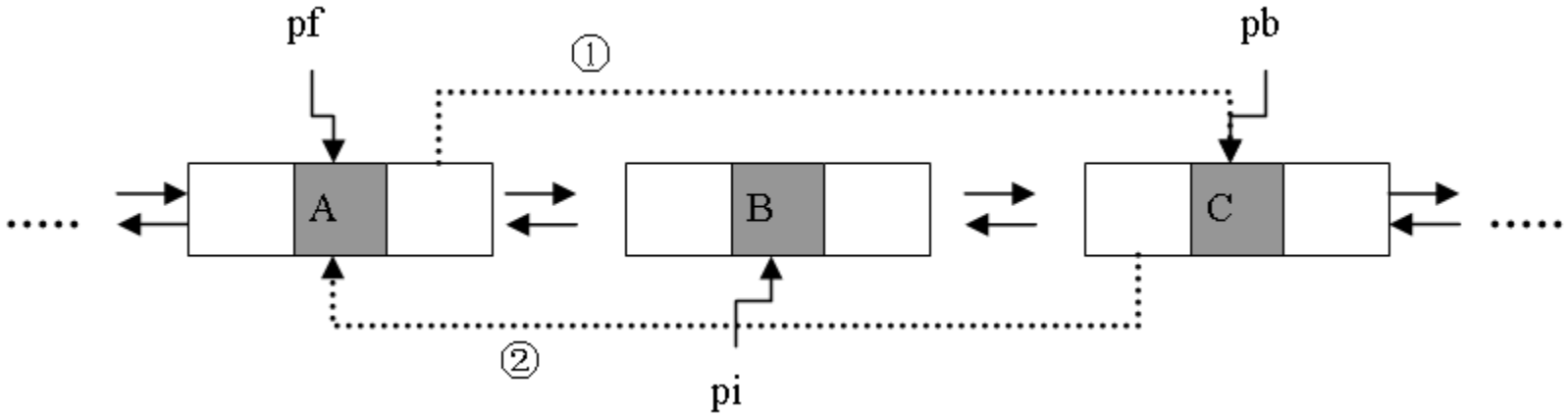
线性表

- 链表
  - 在双向链表表头插入一个结点



线性表

- 链表
  - 在双向链表表尾插入一个结点与在表头插入类似
  - 其实对于循环双向链表来说，可以没有头尾结点概念的。
  - 在双向链表中删除一个结点



线性表

- 链表
  - 共享链表
    - 基于单双向链表，将指针域和数据域进行分开管理，这是linux内核常见的一种链表形式。
    - 链表的操作形式多种多样，共享链表可以将这些操作独立出来实现共享，从而可以提高代码的重用性和健状性。
    - 到此可以看到链表原来可以这样灵活的运用，这也是开源给大家带来的幸福。

线性表

- 链表
  - 共享链表
    - 操作分类:
      - 初始化链表第一个节点
      - 将新节点添加到链表头或尾
      - 删除一个节点
      - 替换一个节点
      - 将一个节点移动到链表头或尾
      - 将一个节点移动到另一个节点的前或后面
      - 判断一个链表是否为空或只有一个节点

线性表

- 链表
  - 共享链表
    - 数据结构的实现:
      - 通过结构体中的成员，求成员地址相对于结构体的偏移量。
      - 通过结构体中的成员地址，求取该成员所在结构体的首地址。

线性表

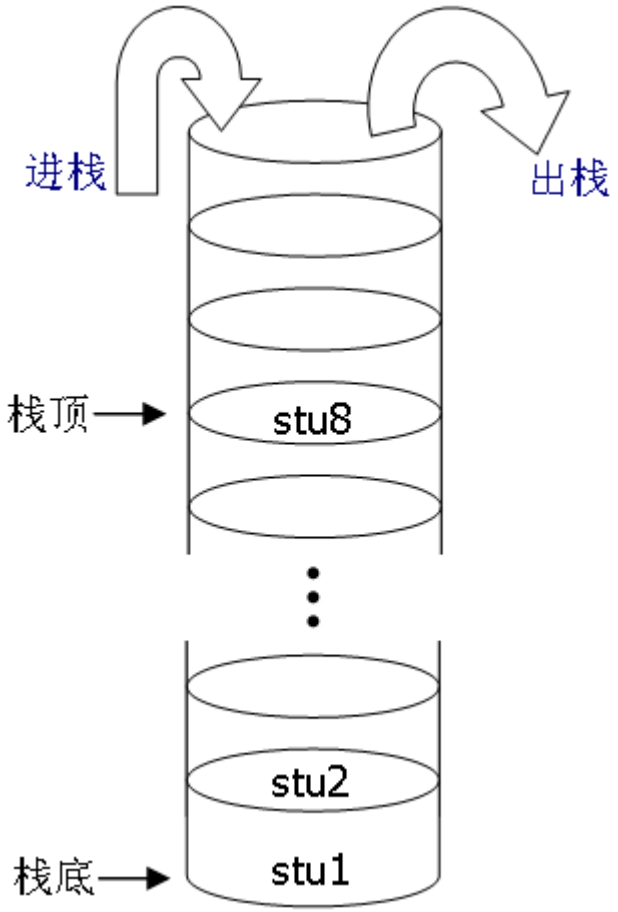
```
#define offsetof(TYPE, MEMBER) ((int)&((TYPE *)0)->MEMBER)
#define container_of(ptr, type, member) \
({const typeof( ((type *)0)->member )*__mptr = (ptr);\
(type *)( (char *)__mptr - offsetof(type,member) );})

typedef struct list_head
{
    struct list_head *prior;
    struct list_head *next;
}NODE;

typedef struct node
{
    NODE list;
    int num;
    int age;
}TYPE;
```

栈和队列

- 栈
  - 限定仅在表尾进行插入或删除操作的线性表。
  - 栈顶：表尾端
  - 栈底：表头端
  - 应用：数制转换，行编辑程序，树的遍历等。
  - 凡是对数据的处理具有“后进先出”的特点，都可以用栈这种数据结构来操作
  - 这种LIFO的数据特征可以用下图形象表示：

栈和队列

➢ 栈

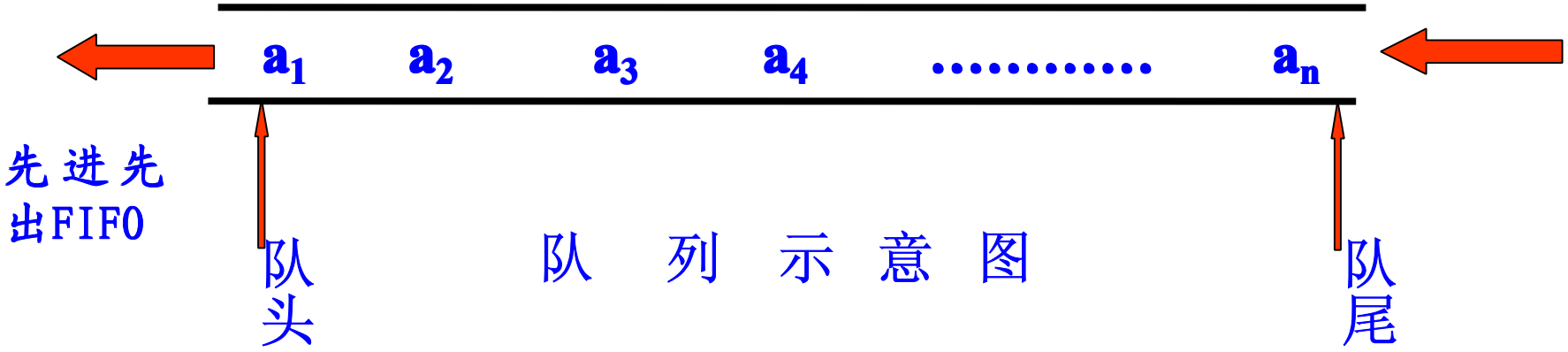


栈和队列

- 队列
  - 限定只允许在表的一端插入，另一端删除，具有先进先出特点的线性表。
  - 队尾：允许插入的一端
  - 队头：允计删除的一端
  - 应用：凡是对数据的处理具有“先进先出”的特点，都可以用队列这种数据结构来操作。
  - 无论栈还是队列，都具有缓存数据的作用，只是跟据实际存取需要，来选择那种线性结构。

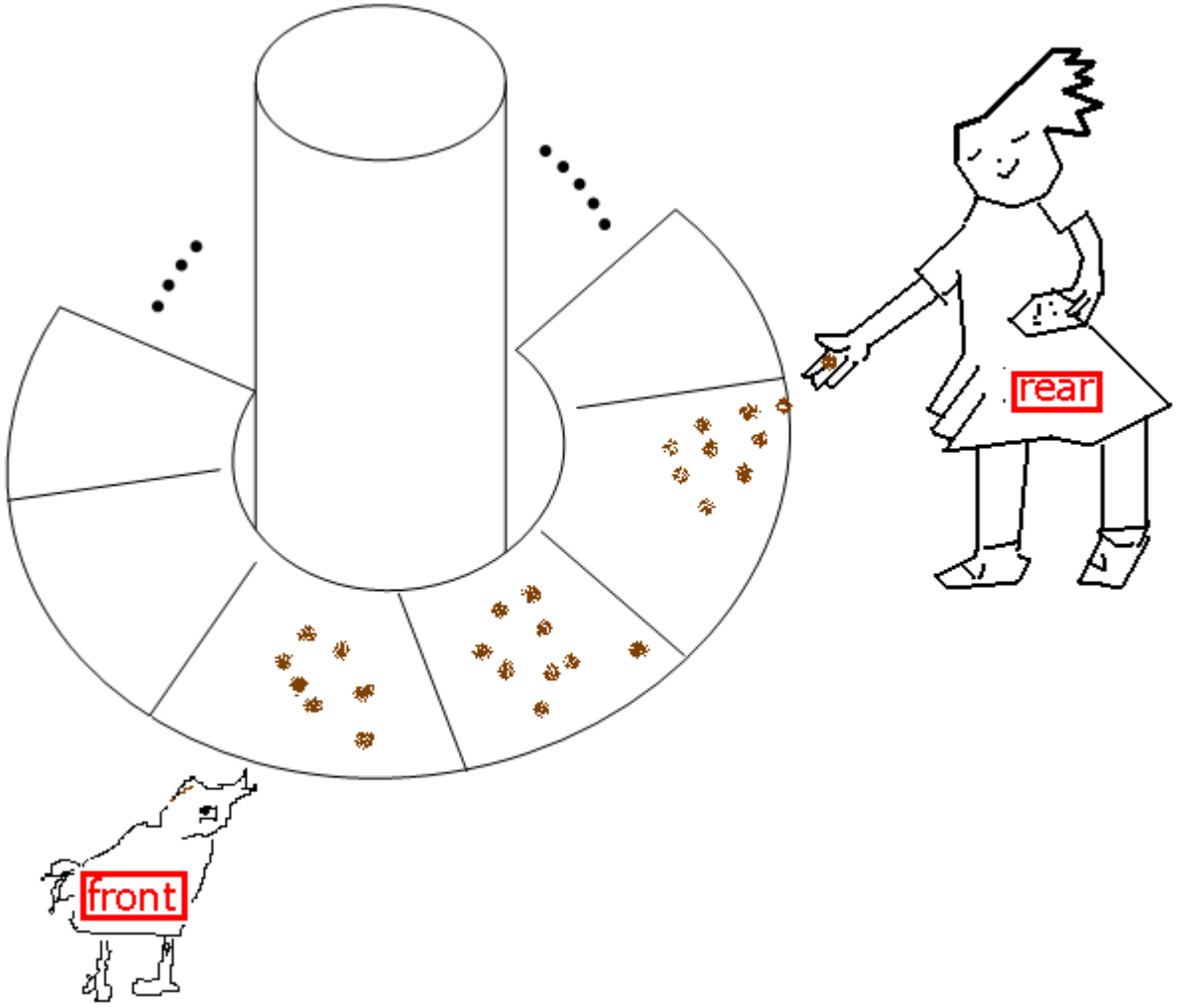
栈和队列

➢ 队列



栈和队列

- 队列
  - 循环队列：



栈和队列

- 队列
  - 循环队列
    - 队尾插入信息并移动指针:
      rear=(rear+1)%MAX
    - 队头取走信息并移动指针:
      front=(front+1)%MAX
    - 绥冲区有信息: rear与front不相等，有可能大于或小于
    - 绥冲区无信息: rear == front
    - 队满条件:
      (rear+1)%MAX=front(rear>=0 && rear<MAX)

树与图

- 递归函数调用
  - 即，主调函数又是被调函数，递归调用层数与内存分配栈的大小有关。
  - 若一个带参函数进行无限递归调用，可能会导致栈溢出，递归调用的效率没有循环语句高。
  - 优缺点：
    - 可以精简一些复杂算法，但消耗内存资源，容易导致栈溢出，程序难于阅读和维护

树与图

- 递归函数调用
  - 递归调用执行过程:
    - 递归函数调用之前代码(函数调入)：每次进入都会执行一遍(调用函数后面的都不再执行)，直到不满足某个条件而退出。
    - 递归函数调用之后代码(函数返回)：每次退出时执行一遍(调用函数前面的都不再执行)。
    - 传入参数：传入的参数即局部变量(调用前对参数的访问顺序如54321)，将全部存在栈中最后返回时就像出队的过程(先入先出如12345)

树与图

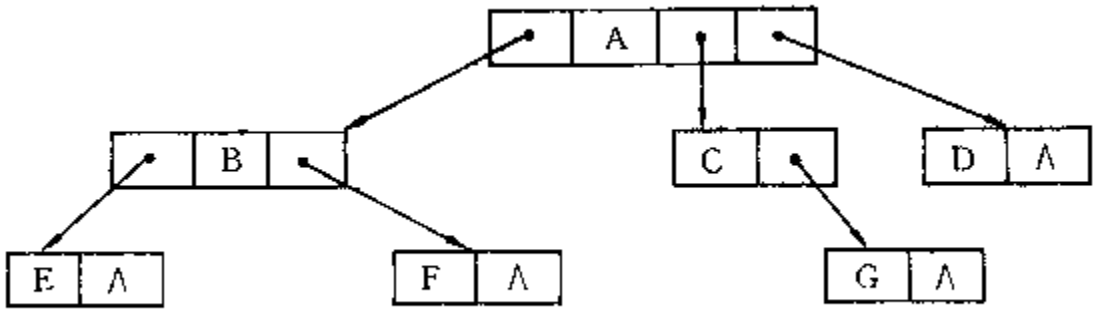
- 树
  - 由n(n≥0)个结点组成的有穷集合
  - 特点:
    - 有且仅有一个称为根(Root)的结点;
    - 当n>1时，其余的结点分为m(m>0)个互不相交的有限集，其中每一个集合本身又是一棵树，并称为根的子树(SubTree)
  - 树结构存储形式:

    一般采用多重链表存储形式，每一个结点由一个数据域和若干个指针域组成，每一个指针域指向该结点的一个孩子结点。

树与图

- 树
  - 多重链表描述如:

```
#define MaxChild    10
typedef struct node
{
    dataType data;
    struct node *child[MaxChild];
}
```



树与图

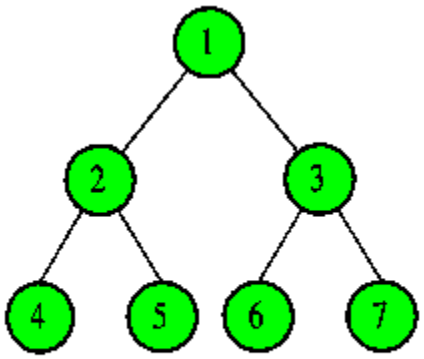
- 二叉树
  - 它或者为空，或者由一个根结点加上两棵分别称为左子树和右子树的互不相交的二叉树组成。
  - 首先从定义形式上就是递归的
  - 通过一定的方法可将一棵多叉树转化为二叉树。
  - 二叉树的使用范围最广，最具代表意义

树与图

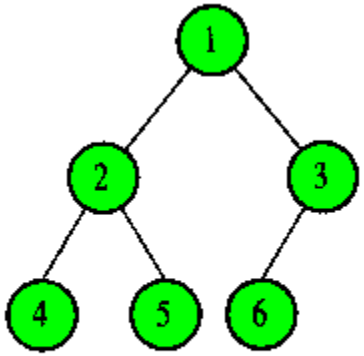
二叉树

二叉树结点描述:

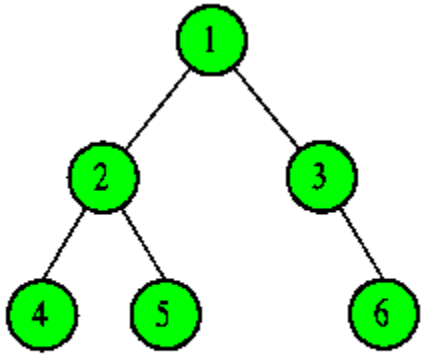
```
typedef struct BiTnode
{
    ElemType data;
    struct BiTnode *lchild,*rchild;
}BiTnode,*BiTree;
```



(a) k=3的满二叉树



(b) 完全二叉树



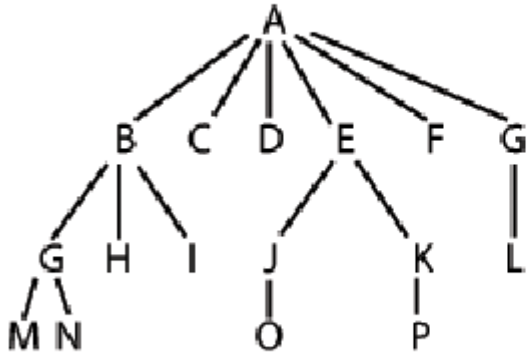
(c) 非完全二叉树

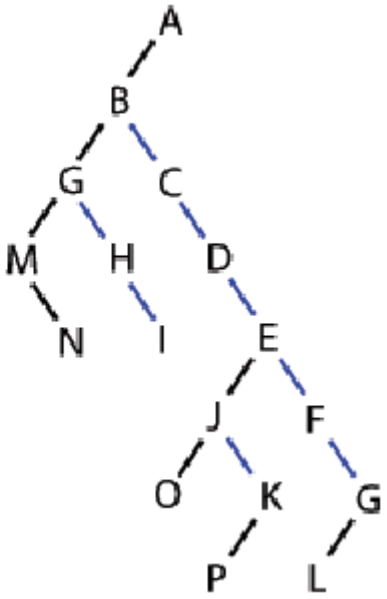
树与图

- 二叉树
  - 二叉树的遍历，由于二叉树的结构本身具有递归特性，每个节点都可看成一棵小树，所以创建多棵二叉树与创建一棵的过程类似，从而常采用递归的方法遍历二叉树。
  - 多叉树转二叉树，多叉树中的左节点变为二叉树中的左节点，多叉树中同层的其它节点，全部依次变成变成二叉树中下层的右节点(除了左节点其它兄弟关系变父子关系)。

树与图

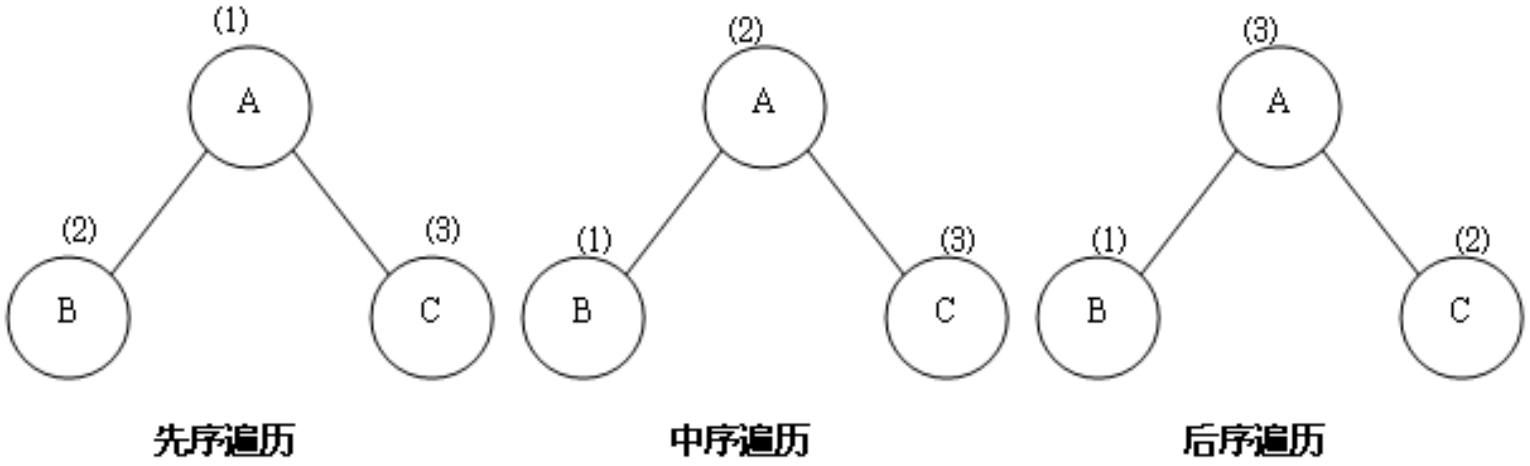
- 二叉树
- 多叉树转二叉树，如图：



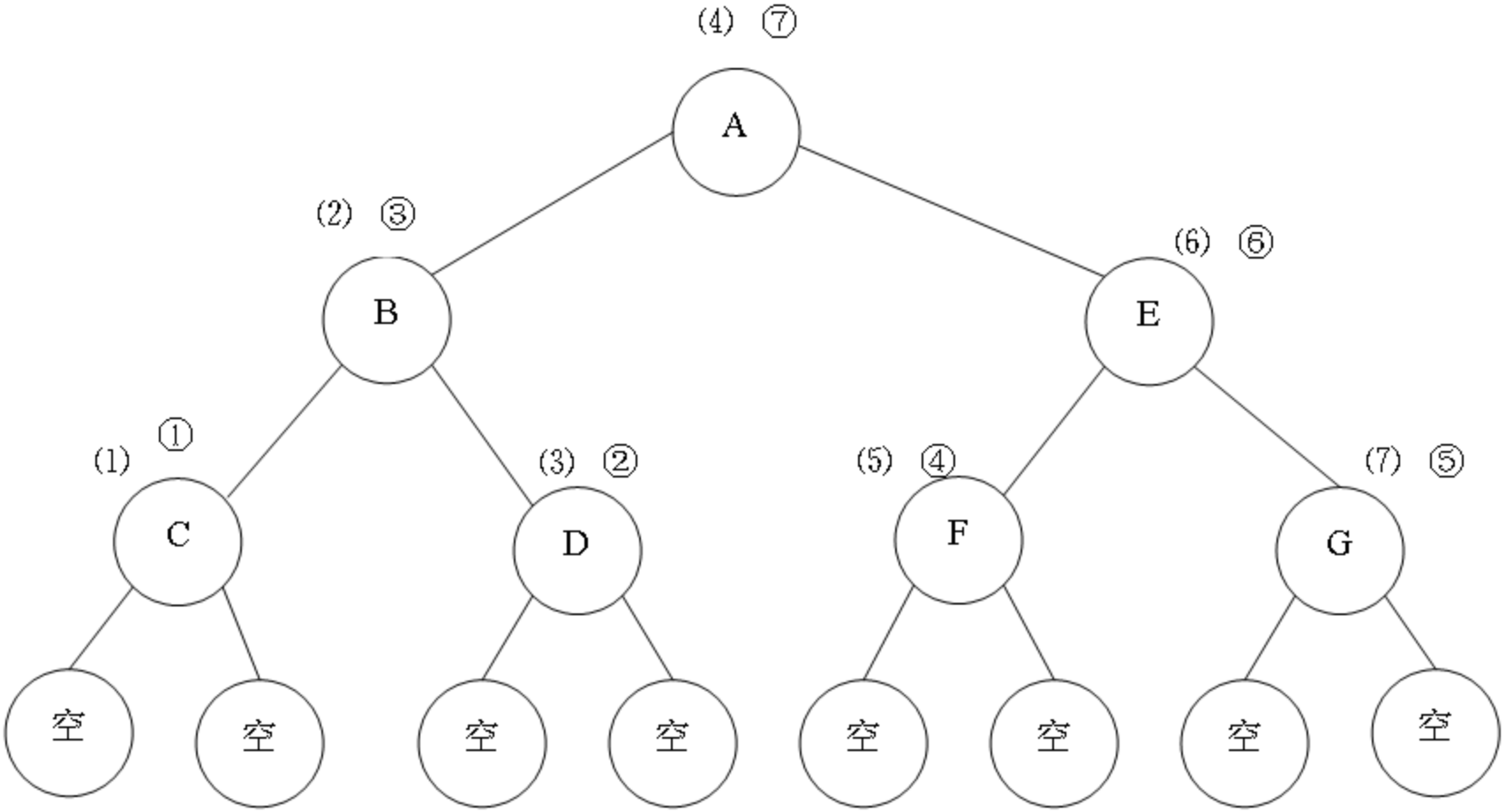


树与图

- 二叉树
  - 先序遍历：也叫先根遍历，根>左子树>右子树
  - 中序遍历：也叫中根遍历，左子树>根>右子树
  - 后序遍历：也叫后根遍历，左子树>右子树>根
  - 遍历过程就是将一颗大树至顶向下分离的过程：



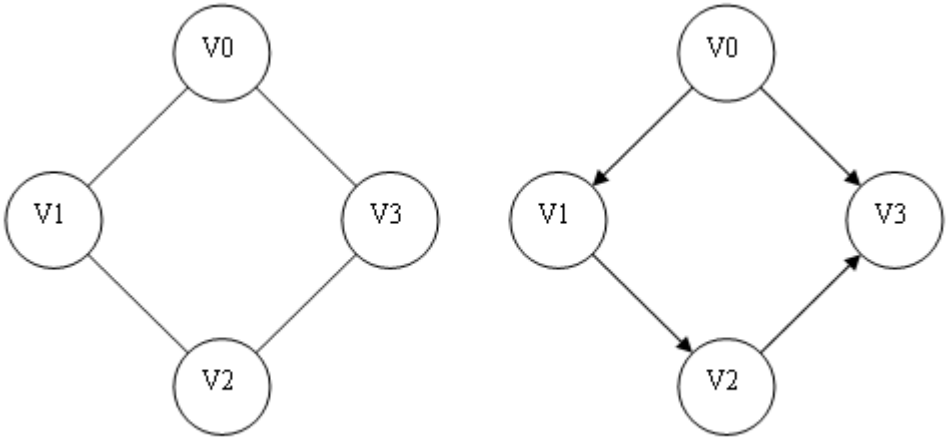
树与图



先序遍历与创建相同，中序遍历，后序遍历

树与图

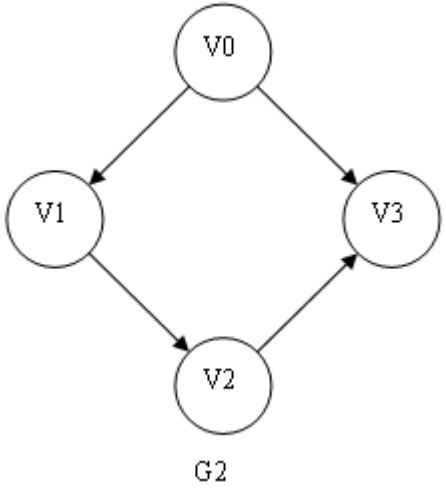
- 图
  - 数据元素之间存在“一对多”或者“多对一”的关系，也就是任意两个数据元素之间都可以存在关系。
  - 如下，G1为无向图，G2为有向图：



树与图

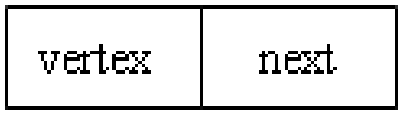
- 图
  - 图的存储形式
    - 邻接距阵：利用两个数组来存储一个图，一个一维数组vertex[]存放顶点数据，一个二维数组A[i][j]用于存放顶点i,j之间的相互关系。
    - A[i][j]为1表示在该方向上有关系，0表示没有关系。

i \ j				
	0	1	0	1
	0	0	1	0
	0	0	0	1
	0	0	0	0



树与图

- 图
  - 图的存储形式
    - 邻接表：是一种将顺序分配与链式分配相结合的存储方法，链表用于存放临边的信息，结构数组用来存放顶点数据信息和第一条临边地址。
    - 一般用结构数组的下标表示顶点在图中的位置。

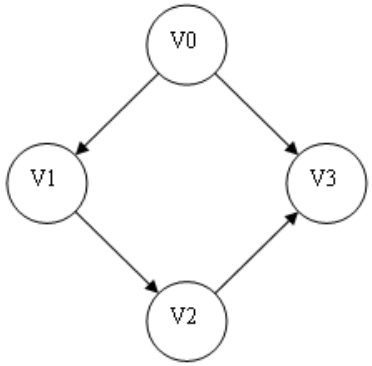


顶点结构

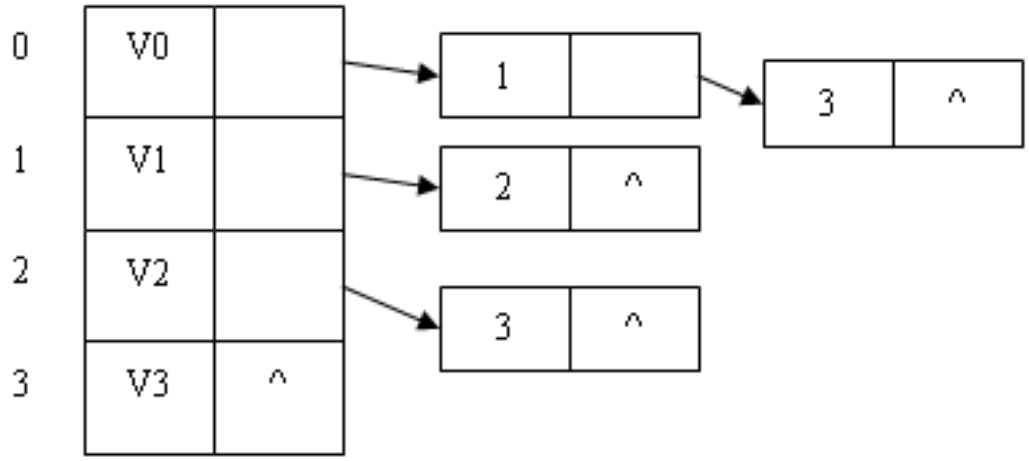
边结点结构

树与图

- 图
  - 图的存储形式
    - 边的权重，weight用于表示边的权重，即顶点间的具体关系，如距离、代价、耗费等。
    - adjvex表示边指向的顶点位置。
    - G2邻接表如图：



G2



树与图

- 图
  - 图的遍历：从图中某一顶点出发，访遍图中其余顶点。
    - 深度优先搜索，从图中的某个顶点v出发，依次从v的未被访问过的邻接点出发，不断深度优先遍历该图，直到图中与顶点v路径相通的所有顶点都被访问到为止。
    - 由于一个图结构未必是连通的，因此一次深度优先搜索不一定可以遍历到图中所有顶点，此时应另选一个未访问的顶点继续深度优先搜索，直到图中所有顶点都被访问到为止。

树与图

- 图
  - 由深度优先搜索的特征可以知道，深度优先搜索的顶点必须是连通的，并且必须要有标志已被访问过顶点的标记，其次是已访问过的顶点不会占用太多重复遍历的时间。
  - 广度优先搜索，从图中顶点v出发，依次访问v的各个未被访问的邻接点，结束后再从这些邻接点出发，按照同样的原则依次访问它们的未被访问的邻接点，如些循环，直到图中所有与v连通邻接点都被访问到。

树与图

- 图
  - 与深度优先一样，如果也存在未连通的结点，另选一个未访问的顶点继续广度优先搜索，直到图中所有顶点都被访问到为止。
  - 树结构是图结构的一种特例，所以广度优先搜索同样适用于对树的遍历，因为树结构是标准的层次结构。
  - 对G2进行深度优先访问结果是：0,1,2,3
  - 对G2进行广度优先访问结果是：0,1,3,2

查找与排序

- 查找
  - 根据给定的某个值，在查找表中确定一个其关键字等于给定值的记录或数据元素，若表中存在这样的一个记录，则称查找是成功的。
  - 在文件系统中，经常要对磁盘上文件的记录进行各种检索操作，如根据歌曲名找到相应的音频文件路径播放歌曲。
  - 常见的有顺序查找、二分查找、哈希表等

查找与排序

- 顺序查找
  - 从文件第一个记录开始，将每个记录的关键字与给定的关键字key进行比较，如果查找到某个记录，就返回该记录的地址，否则返回失败。
  - 时间复杂度最高为O(n)
  - 例如，现有一个学生信息表，如果我们已知学号，通过学号来查到学生的其它信息，可以这样设计：

查找与排序

```
typedef struct student{
    int id;                    /*学生编号*/
    char name[10];             /*学生姓名*/
    float score;               /*成绩*/
}Student;
Student stu[4] = {{1004,"TOM",100} ,
                  {1002,"LILY",95},
                  {1001,"ANN",93},
                  {1003,"LUCY",98}
                  };                              /*初始化结构体数组*/

int search(Student stu[],int n,int key){
    int i;
    for(i=0; i<n; i++)
        if( stu[i].id == key )                    /*查找成功*/
            return i;
    return -1;                                    /*查找失败*/

}
```

查找与排序

- 二分查找
  - 如果从文件中读取的记录关键字是有序排列的，则可以用一种效率更高的查找方法来实现，即二分查找。
  - 基本思路是，每次都将关键字与中间的元素对比，如果不相等再判断关键字范围并移动上下限指针，继续折半比较。
  - 时间复杂度最高为O(n/2)

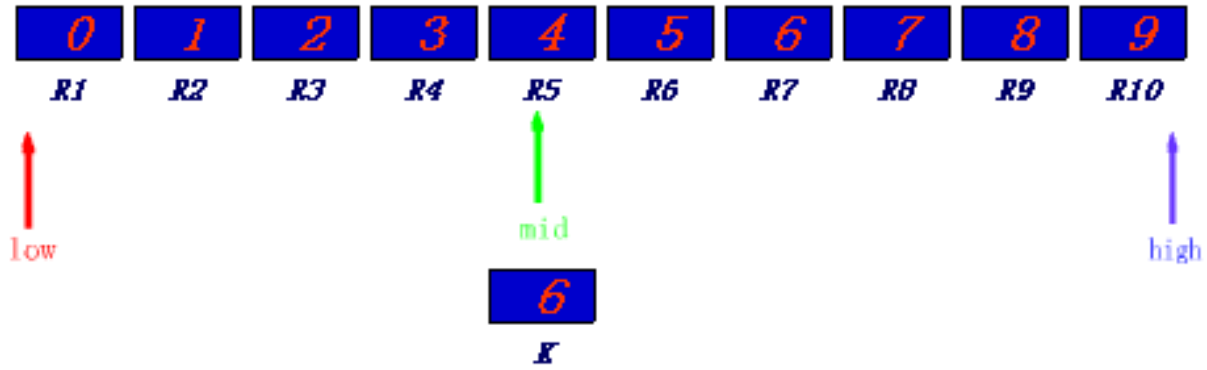
查找与排序

➢ 二分查找

二分查找(*BinarySearch*)

R[low..high]为当前查找区间

设置当前查找区间的下界low和上界high!



查找与排序

- 哈希表
  - 又叫散列表，是根据关键码值直接进行访问的数据结构，也就是说，它通过把关键码值映射到表中某一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数，存放记录的数组叫做散列表。
  - 哈希表是一个以空间换取时间的数据结构，理想情况下的时间复杂度为O(1)
  - 冲突：不同关键字对应同一存储单元，即：$H(key_m)=H(key_n)$

查找与排序

- 哈希表
  - 装填因子，$\alpha = n/m$, 表示有n个元素要装载到m个存储单元中，装填因子越小，出现冲突的可能性就越小。
  - 出现冲突的可能性越小，那么时间复杂度就会越小，从而可知要减小时间复杂度，就必须扩大存储空间。
  - 散列函数，能使数据序列的检索访问过程更加迅速有效，通过散列函数，数据元素将被更快地定位 ,设计一个优秀的散列函数是关键。
  - 散列函数的构造方法有:

查找与排序

- 直接定址法：取关键字或关键字的某个线性函数值为散列地址。即H(key)=key或H(key) = a•key + b，其中a和b为常数（这种散列函数叫做自身函数）
- 数字分析法
- 平方取中法
- 折叠法
- 随机数法
- 除留余数法

查找与排序

- 哈希表
  - 冲突处理
    - 开放寻址法：Hi=(H(key) + di)%m，其中H(key)为散列函数，m为散列表长，di为增量序列，可有下列三种取法：

      1. di=1,2,3,…，m-1，称线性探测再散列；

      2. di=1^2，$(-1)^2$，2^2，$(-2)^2$，$(3)^2$，…，±(k)^2，(k<=m/2)称二次探测再散列；

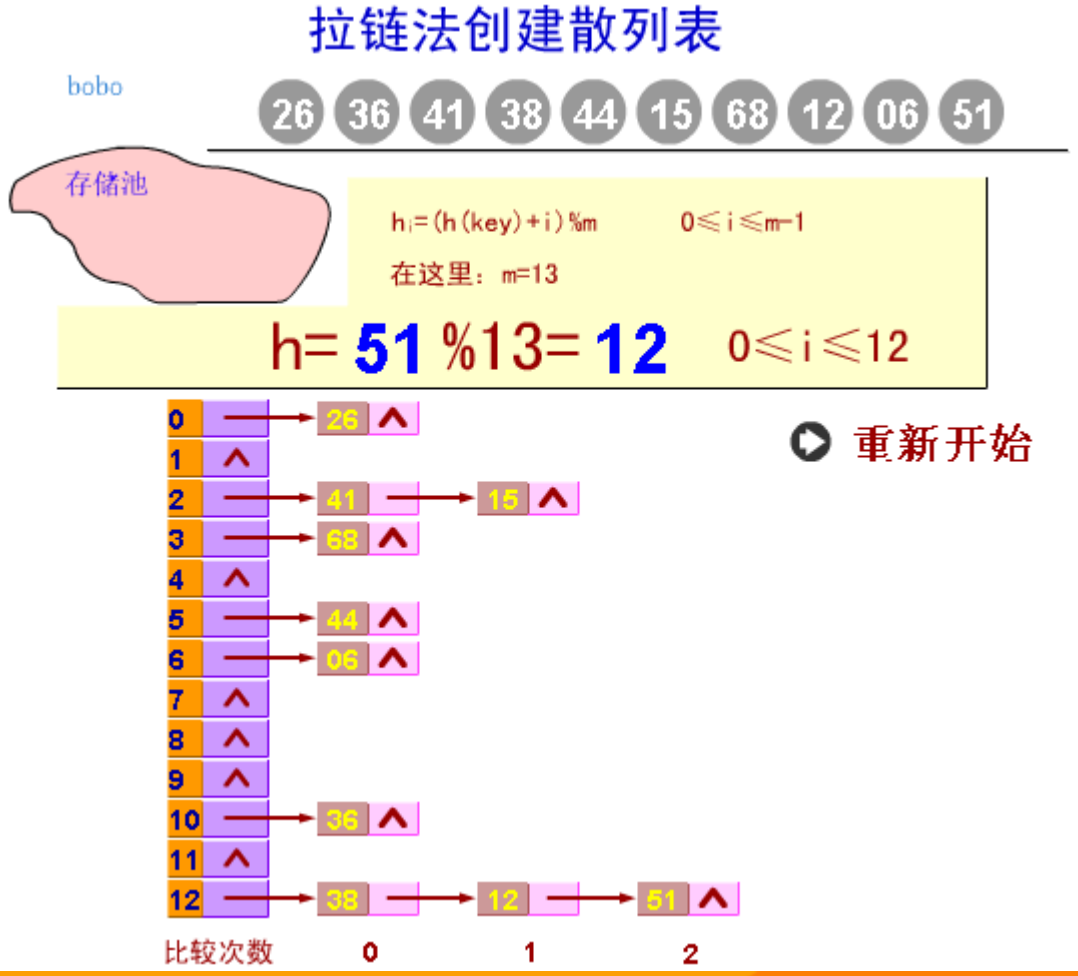
      3. di=伪随机数序列，称伪随机探测再散列。

查找与排序

- 哈希表
  - 冲突处理
    - 再散列法：Hi=RHi(key)， i=1,2,...,k RHi均是不同的散列函数，即在同义词产生地址冲突时计算另一个散列函数地址，直到冲突不再发生，这种方法不易产生“聚集”，但增加了计算时间。
    - 链地址法：又叫拉链法，他将产生冲突的元素链成一条链表，它是一种哈希表与遍历或二分查找等其它检索方法相结合的一种方法。
    - 建立一个公共溢出区

查找与排序

➢ 哈希表


拉链法创建散列表
bobo
26 36 41 38 44 15 68 12 06 51
存储池
$h_i=(h(key)+i)\%m$ $0\leqslant i\leqslant m-1$
在这里：m=13
h= 51 %13= 12 $0\leqslant i\leqslant 12$
重新开始
0 26
1
2 41 15
3 68
4
5 44
6 06
7
8
9
10 36
11
12 38 12 51
比较次数 0 1 2

查找与排序

- 选择法排序
  - 从第一个元素开始作为起始值，与起始值比较找出剩下未比较过的最值与起始值交换，即选出一个最值出来。
  - 步骤：

    ①首先假设第一个元素“i”是最值。

    ②从第二个元素下标“j”开始一个个取出与“i”进行比较是否交换。

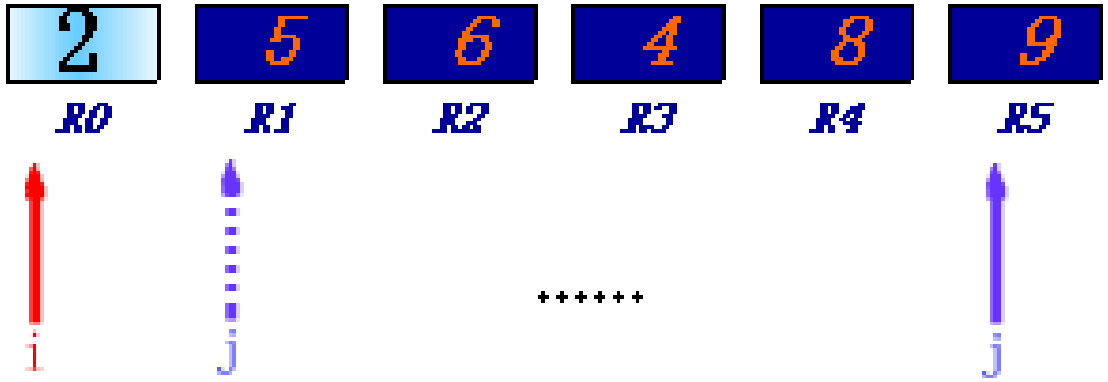
    ③内循环一次则找出一个最值与“i”进行交换，否则不变

    ④外循环“i”所指向的值就是上一轮的最值，“i”加一后进入下一轮比较。

查找与排序

选择法排序

选择排序处理过程:




```
for(i=0;i<n;i++)// " i " 就是起始值
{
    for(j=i+1;j<n;j++)
    {
        if(k[j] < k[i])
            swap(&k[j],&k[i]);
    }
}
```

查找与排序

- 插入法排序
  - 通过数据移动，留出合适位置插入顺序合适的值，而无须数据交换。
  - 步骤：

①从第二个元素“i”开始绥存准备用于比较，并留出一个空位。

②将空位前的元素“j”拿来与绥存值比较不满足则移动，直到向前找到头。

③比较的目的是要让绥存值插入后成为从开头到插入点这个区间中的最值。

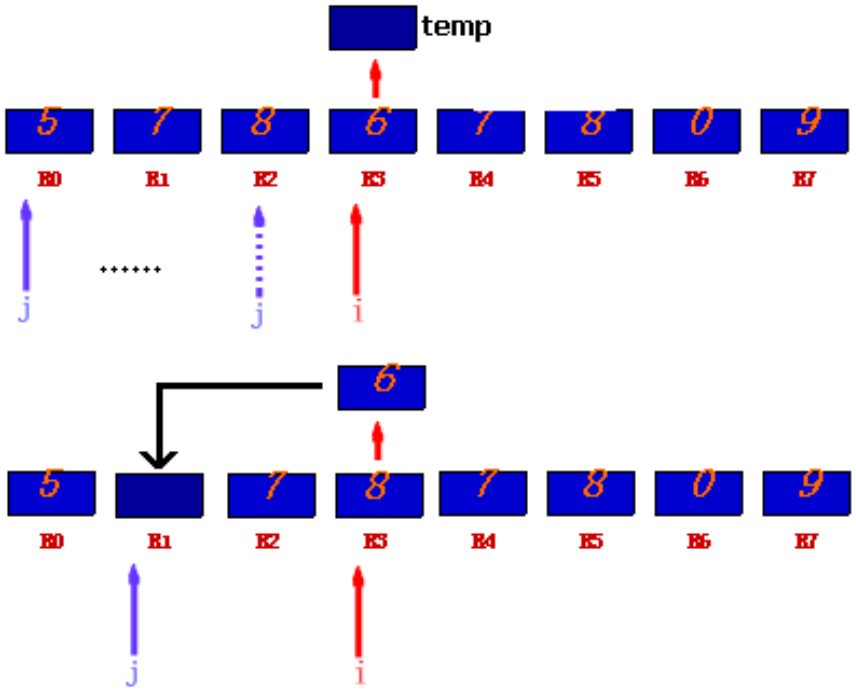
④如果绥存值向前看不是最值，则往后移动，直到可以让绥存值插入后成为最值时停止。

查找与排序

➢ 插入法排序

④将绥存值插入到最值位置(即内循环下一个将要移动的位置)。

➢ 选择排序处理过程:




```
for(i=1;i<n;i++)
{
    temp = k[i];
    j = i - 1;
    while(j>=0 && k[j]<temp)
        k[j+1] = k[j--];
    k[j+1] = temp;
}
```

查找与排序

- 冒泡法排序
  - 每遍历一次都从第一个元素开始找出一个最值，依次相邻比较往后推移，结尾条件向前移动。
  - 步骤：

    ①每次都从第一个元素“j”开始，不断与下一个元素“j+1”比较。

    ②将相邻比较的最值交换到后面。

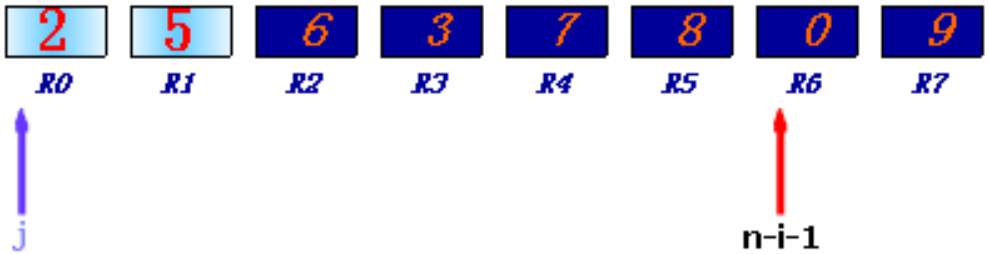
    ③外循环次数为总长度，内循环次数要减一，否则最长长度加一以后就溢出了。

    ④加上flag标志，一旦排好序就退出循环，提高效率。

查找与排序

冒泡法排序

冒泡排序处理过程:




```
for(i=0;i<n;i++)
{
    for(j=0;j<n-i-1;j++)
    {
        if(k[j]>k[j+1])
        {
            swap(&k[j+1],&k[j]);
        }
    }
}
```

查找与排序

- 希尔排序
  - 由希尔在1959年提出，设定一个元素间隔增量gap，将参加排序的序列按这个间隔分成若干个子序列，对子序列用一般排序法排序。
  - 与冒泡排序的思想相似，即，将两数比较交换，冒泡法是相邻两数比较交换，而希尔排序只有当希尔间隔值为1时才与冒泡法完全一样，所以可以跟据希尔间隔值的不同进行分组排序。
  - 最优间隔值是至今未解决的数据难题，一般用折半间隔，直到间隔为1。如果直接将希尔间隔设为1效率则是最低的，需要多次遍历。

查找与排序

- 希尔排序
  - 希尔排序处理过程:
    - 将固定间隔值两两比较交换，若干次后，这个序列将成为一个有序序列。
    - 对分好组的子序列采用两两比较交换，循环比较一遍下来不一定排好了序，所以对于子序列可能会进行多次循环比较，直到排好序。

查找与排序

- 希尔排序
  - 希尔排序处理过程:
    - 直接将希尔间隔设为1:

```
do
{                                   /*子序列应用冒泡排序*/
    flag = 0;
    for(i=0;i<n-1;i++)
    {
        if(k[i]>k[i+1]) //相邻的前后值进行比较
        {
            swap(&k[i],&k[i+1]);
            flag = 1;
        }
    }
}while(flag !=0);
```

查找与排序

- 采用折半法设置希尔间隔值:

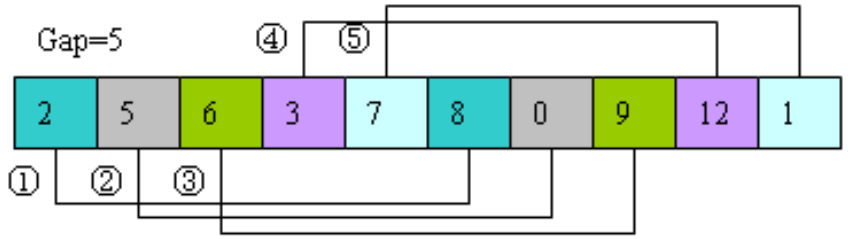
```
while(gap > 1)
{
    gap = gap/2;                    /*增量缩小，每次减半*/
    do
    {                               /*子序列应用冒泡排序*/
        flag = 0;
        for(i=0;i<n-gap;i++) //n-gap 是控制上限不让越界
        {
            j = i + gap;     //相邻间隔的前后值进行比较
            if(k[i]>k[j])
            {
                swap(&k[i],&k[j]);
                flag = 1;
            }
        }
    }while(flag !=0);
}
```

查找与排序

➢希尔排序处理过程:

排序{2, 5, 6, 3, 7, 8, 0, 9, 12, 1}图解过程如下:

红色表示不交换



原序列:

2	5	6	3	7	8	0	9	12	1

Gap=5

2	0	6	3	1	8	5	9	12	7

Gap=2

2	0	1	3	6	8	5	9	12	7
2	0	1	3	5	8	6	9	12	7
2	0	1	3	5	8	6	7	12	9

Gap=2

1	0	2	3	5	8	6	7	12	9
1	0	2	3	5	7	6	8	12	9

Gap=1

0	1	2	3	5	7	6	8	12	9
0	1	2	3	5	6	7	8	12	9
0	1	2	3	5	6	7	8	9	12

查找与排序

- 快速排序
  - 由C.A.R Hoarse提出的一种排序算法，在各种内部排序方法中，快速排序被认为是目前最好的一种排序方法。
  - 找一个基准值，分别将大于和小于基准值的数据放到基准值左右两边，即一次划分。
  - 由于处在两边的数据也是无序的，所以再用同样的划分方法对左右两边的序列进行再次划分，直到划分元素只剩1个时结束，类似微分过程。

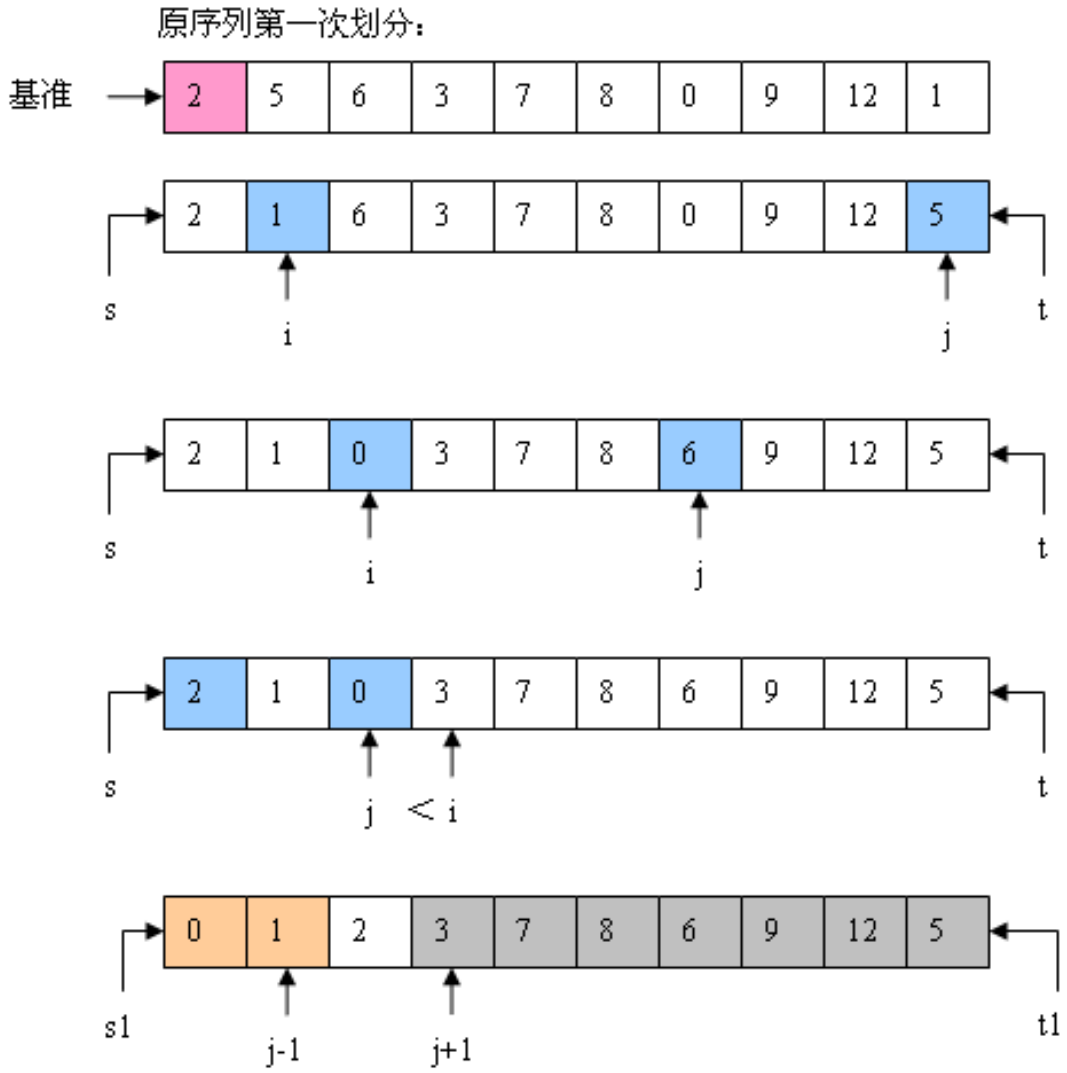
查找与排序

- 快速排序
  - 快速排序处理过程:
    - 由于对序列划分的过程完全相同，所以可以采用递归来实现整个过程。
    - 划分过程如下:

```
s = 0; j = n+1;
while(1){
do i++;
while(!(k[s]<=k[i] || i==n));//从第一个开始求出第一个大于基准值的元素位置i
do j--;
while(!(k[s]>=k[j] || j==s));//从最后开始求出第一个小于基准值的元素位置j
if(i<j)
    swap(&k[i],&k[j]);     /*交换k[i]和k[j]的位置*/
else break;
}
swap(&k[s],&k[j]);//将基准元素与从后往前的第一个大于s的元素进行交换，即放在中间
```

查找与排序

➢ 快速排序


原序列第一次划分：
基准
2 5 6 3 7 8 0 9 12 1
2 1 6 3 7 8 0 9 12 5
s
i
j
t
2 1 0 3 7 8 6 9 12 5
s
i
j
t
2 1 0 3 7 8 6 9 12 5
s
j ＜ i
t
0 1 2 3 7 8 6 9 12 5
s1
j-1
j+1
t1

查找与排序

- 其它排序算法
  - 堆排序、归并排序、基数排序(多关键字、链式)、树形选择排序等等。
  - **内部排序:** 整个排序过程不需要访问外存便能完成的排序。
  - **外部排序:** 若参加排序的记录数量很大，整个序列的排序过程不可能在内存中完成，排序需要借助外部存储设备才能完成的排序问题。

凌阳教育网站: http://www.sunplusedu.com
凌阳教育: E-mail: edu@sunplusedu.com
联系电话: 010—62981113—2934
010—62981113—2923

凌阳教育
值得信赖的教育品牌