# 第07天【算法和数据结构】

## 主要内容

1. 冒泡排序
2. 选择排序
3. 递归算法
4. 折半查找
5. 数据结构概述
6. 线性表
7. 栈和队列
8. 树和二叉树
9. 图

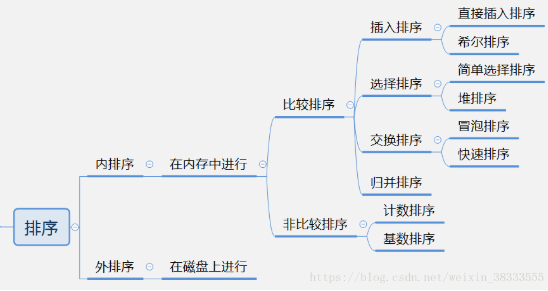
## 学习目标

|  |  |
| --- | --- |
| 知识点 | 要求 |
| 排序概述 | 了解 |
| 冒泡排序 | 掌握 |
| 选择排序 | 掌握 |
| 递归 | 掌握 |
| 折半查找 | 掌握 |
| 数据结构概论 | 理解 |
| 线性表 | 理解 |
| 栈和队列 | 了解 |
| 树和二叉树 | 理解 |
| 图 | 了解 |

## 一、排序

### 1.1排序概述

排序（sorting）的功能是将一个数据元素的任意序列，重新排列成一个按关键字有序的序列。



**内部排序和外部排序**

一类是整个排序过程在内存储器中进行，称为内部排序；

另一类是由于待排序元素数量太大，以至于内存储器无法容纳全部数据，排序需要借助外部存储设备才能完成，这类排序称为外部排序。

本章介绍的排序方法都属于内部排序

**比较排序和非比较排序**

大部分排序都是需要通过比较首先来判断大小，作为排序的依据的。

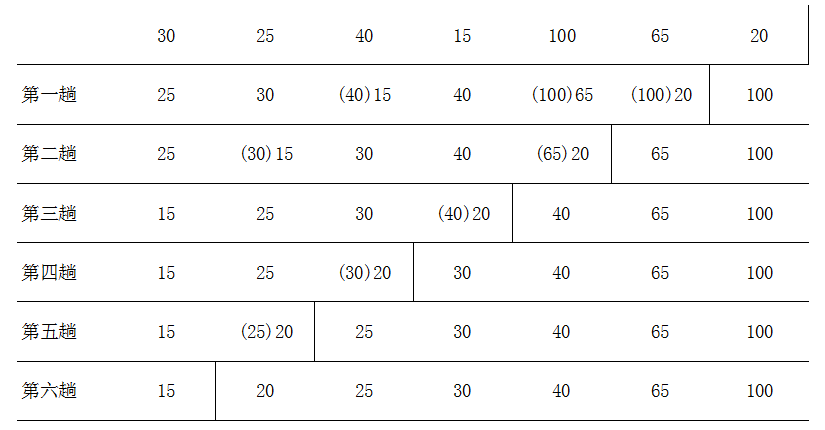
但是也有例外的，比如计数排序、基数排序，不需要进行比较。效率可以做到更高，但是会有一些限制条件，也可能需要更多的空间。

冒泡排序、选择排序、直接插入排序是最基本的三种排序，效率最低，但是算法简单。排序的学习一般从这三种排序算法开始。

### 1.2冒泡排序

冒泡排序的算法

1. 整个数列分成两部分：前面是无序数列，后面是有序数列
2. 初始状态下，整个数列都是无序的，有序数列是空
3. 如果一个数列有n个元素，则至多需要n-1趟循环才能保证数列有序
4. 每一趟循环可以让无序数列中最大数排到最后，（也就是说有序数列的元素个数增加1）
5. 每一趟循环都从数列的第一个元素开始进行比较，依次比较相邻的两个元素，比较到无序数列的末尾即可（而不是数列的末尾）
6. 如果前一个大于后一个，交换



#### 【示例1】冒泡排序算法

|  |
| --- |
| **public class** TestBubbleSort1{  **public static void** main(String [] args){  //定义一个无序数组  **int** [] arr = {75,87,56,45,89,100,76,34,89,97};  //排序前输出  System.**out**.println(**"排序前"**);  **for**(**int** score :arr){  System.**out**.print(score+**"\t"**);  }  //排序  //大循环：n个元素排序，则至多需要n-1趟循环  **for**(**int** i=0;i<arr.**length**-1;i++){  //小循环：每一趟循环都从数列的前两个元素开始进行比较，  //比较到数组的最后  **for**(**int** j=0;j<arr.**length**-1;j++){  //如果前一个大于后一个  **if**(arr[j] > arr[j+1]){  //交换  **int** temp;  temp = arr[j];  arr[j] = arr[j+1];  arr[j+1] = temp;  }  }  }  //排序后输出  System.**out**.println(**"\n排序后"**);  **for**(**int** score :arr){  System.**out**.print(score+**"\t"**);  }  } } |

缺点1：每一趟比较都要比较到数组的最后，没有必要，只要比较到无序数列最后即可

for(int j=0;j<scoreArr.length-1;j++){ }

i j<?

0 <6

1 <5

2 <4

3 <3

i j<6-i scoreArr.length-1-i

解决：j<scoreArr.length-1 修改为 j<scoreArr.length-1-i

缺点2：不管是否有序，都要进行n-1趟循环；

如何判断有序：比较了一趟，没有发生交换

解决：定义一个符号量flag，默认有序true；发生交换，置为false，

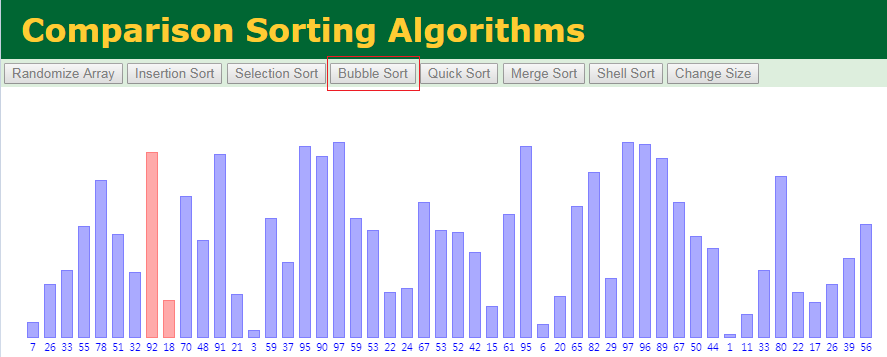
一趟循环结束后，根据flag的值判断是否有序；有序，退出即可；

#### 【示例2】完善冒泡排序算法

|  |
| --- |
| **public class** TestBubbleSort2{  **public static void** main(String [] args){  //定义一个无序数组  **int** [] arr = {75,87,56,45,89,100,76,34,89,97};   //排序前输出  System.**out**.println(**"排序前"**);  **for**(**int** score :arr){  System.**out**.print(score+**"\t"**);  }  //排序  bubbleSort(arr);   //排序后输出  System.**out**.println(**"排序后"**);  **for**(**int** score :arr){  System.**out**.print(score+**"\t"**);  }  }   **public static void** bubbleSort(**int** arr[]){  //大循环：n个元素排序，则至多需要n-1趟循环  **int** temp;  **int** i;  **for**(i=0;i<arr.**length**-1;i++){  //1. 假设有序  **boolean** flag = **true**;  //2.小循环：每一趟循环都从数列的前两个元素开始进行比较，比较到数组的最后  **for**(**int** j=0;j<arr.**length**-1-i;j++){  //如果前一个大于后一个  **if**(arr[j] > arr[j+1]){  //交换   temp = arr[j];  arr[j] = arr[j+1];  arr[j+1] = temp;  //发生了交换，数组无序  flag = **false**;  }  }  //3.判断是否有序，有序，退出  **if**(flag){  **break**;  }  }  } } |

算法动态展示网站

<https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>



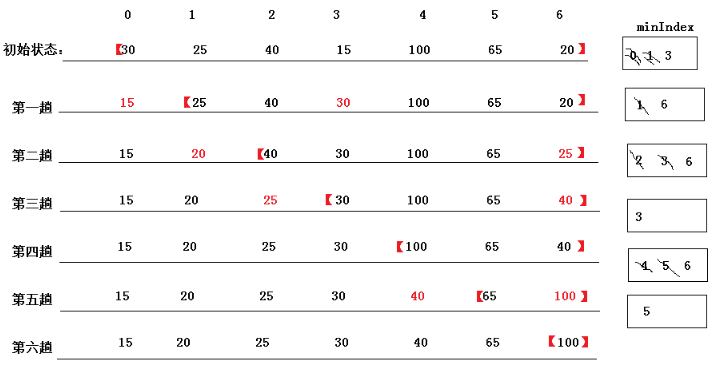
算法动画图解app（破解版支持中文）



### 1.3 选择排序

选择排序的算法

1. 整个数列分成两部分：前面是有序数列，后面是无序数列
2. 初始状态下，整个数列都是无序的，有序数列是空
3. 一共n个数，需要n-1趟循环（一趟都不能少）
4. 每比较完一趟，有序数列数量+1，无序数列数量-1
5. 每趟先假设无序数列的第1个元素（整个数列的第i个元素）是最小的，让当前的最小数，从第i+1个元素开始比较，一直比较到最后一个元素。如果发现更小的数，就假设当前数是最小数。
6. 一趟比较完后，将发现最小数和无序数列的第一个数交换（如果最小数不是无序数列的第一个数）



#### 【示例3】选择排序

|  |
| --- |
| **public class** TestSelectSort {  **public static void** main(String[] args) {  //给定无序的数组  **int** [] scoreArr = {75,87,56,45,89,100,76,34,89,97};  System.**out**.println(Arrays.toString(scoreArr));  //选择排序  selectSort(scoreArr);  System.**out**.println(Arrays.toString(scoreArr));  }  **public static void** selectSort(**int**[] scoreArr) {  //大循环：n个元素排序，则需要n-1趟循环  **for**(**int** i=0;i<scoreArr.**length**-1;i++){  //第i趟先假设第i个最小  **int** minIndex = i;  //从第i+1个元素开始，依次使用最小元素和每元素比较，一直比较到最后  **for** (**int** j = i+1; j <scoreArr.**length** ; j++) {  **if**(scoreArr[minIndex] > scoreArr[j]){  minIndex = j;  }  }  //一趟比较完后，或者最小值的索引，如果不是第i个，就交换  **if**(minIndex !=i){  **int** temp; temp = scoreArr[i];   scoreArr[i] = scoreArr[minIndex]; scoreArr[minIndex] = temp;  }  }  } } |

|  |
| --- |
| **注意**   * 冒泡排序最多需要n-1趟循环，最少1趟循环；选择排序必须进行n-1趟循环，一趟也不能少 * 冒泡排序中最多的操作就是比较和交换，一趟循环中可能发生多次交换；而选择排序中最多的操作是比较，一趟比较结束后如果发现更小的值才交换一次。 * 如果数组元素不是基本数据类型，而是对象，应该如何判断大小呢？可以让相应类实现Comparable接口并调用comparTo()方法进行比较。 * 快速排序是冒泡排序的完善版，都是基于交换的排序，是各种基于比较的排序算法中效率最高的，用到了分治和递归的思想。需要认真准备。 |

### 本节作业

1. 使用冒泡排序实现对分数排序
2. 使用选择排序实现对分数排序
3. 可选：使用选择排序实现对课程名称（比如Java、MySQL等）按照字母排序
4. 可选：查询资料掌握直接插入排序算法并实现对分数排序

## 二、递归和折半查找

### 2.1 递归

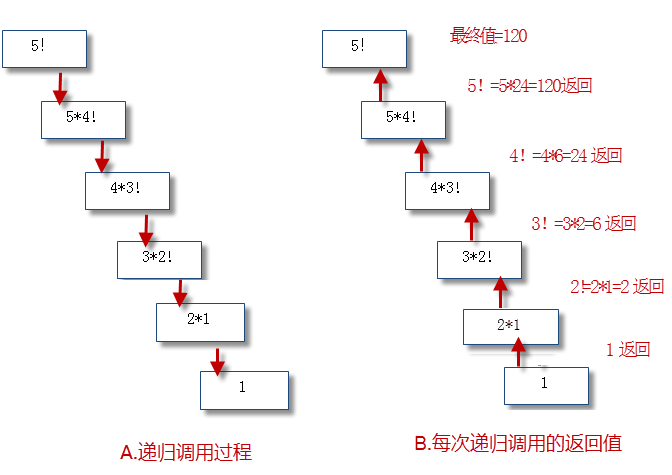
递归（recursion）是一种常见的解决问题的方法，即把问题逐渐简单化。递归的基本思想就是“自己调用自己”，一个使用递归技术的方法将会直接或者间接的调用自己。利用递归可以用简单的程序来解决一些复杂的问题。比如：斐波那契数列的计算、汉诺塔、快速排序等问题。



#### 【示例4】使用递归实现n!

|  |
| --- |
| **public class** TestRecursion1 {  **public static void** main(String[] args) {  //使用循环求n!  **int** n = 6;  **int** fac = 1;  **for**(**int** i=1;i<=n;i++){  fac = fac \* i;  }  System.**out**.println(fac);   //使用递归求n!  **int** result = fac(n);  System.**out**.println(result);  }   **public static int** fac(**int** n){  **int** result;  **if**(n==1){  result = 1;  }**else**{  result = n \* fac(n-1);  }  **return** result;   } } |

递归的调用过程。



#### 【示例5】使用递归实现斐波那契数列

斐波那契数列（Fibonacci sequence），又称黄金分割数列、因数学家列昂纳多·斐波那契（Leonardoda Fibonacci）以兔子繁殖为例子而引入，故又称为“兔子数列”，指的是这样一个数列：1、1、2、3、5、8、13、21、34、……在数学上，斐波那契数列以如下被以递推的方法定义：F(1)=1，F(2)=1, F(n)=F(n-1)+F(n-2)（n>=3，n∈N\*）。

|  |
| --- |
| **public class** TestRecursion2 {  **public static void** main(String[] args) {  //使用循环实现  **int** num1 = 1;  **int** num2 = 1;  **int** numn=0 ;  **int** n= 40;  **long** startTime = System.currentTimeMillis();  **for**(**int** i=3;i<= n;i++ ){  //得到i项的值  numn = num1+ num2;  //改变num2和num1的值  num1 = num2;  num2 = numn;   }  System.**out**.println(n+**" "**+numn);  **long** endTime = System.currentTimeMillis();  System.**out**.println(**"循环花费的时间："**+(endTime - startTime));  //使用递归实现  startTime = System.currentTimeMillis();  System.**out**.println(n+**" "**+fibo(n));  endTime = System.currentTimeMillis();  System.**out**.println(**"递归 花费的时间："**+(endTime - startTime));  }   **public static int** fibo(**int** n){//1,2,3.....n  //给结果指定初始值  **int** result = 0;   //使用递归求结果  **if**(n==1 || n==2){  result = 1;  }**else**{  result = fibo(n-2) + fibo(n-1);  }   //返回结果  **return** result;  } } |

**递归问题的特点**

一个问题可被分解为若干层简单的子问题

子问题和其上层问题的解决方案一致

外层问题的解决依赖于子问题的解决

**递归结构包括两个部分：**

递归结束条件：什么时候不调用自身方法。如果没有条件，将陷入死循环。

递归体。解答：什么时候需要调用自身方法。

**递归的优点**

自然的思路，简单的程序

**递归的缺点**

但是递归调用会占用大量的系统堆栈，内存耗用多，

在递归调用层次多时速度要比循环慢的多

|  |
| --- |
| **注意事项**   * **任何能用递归解决的问题也能使用迭代解决。**当递归方法可以更加自然地反映问题，并且易于理解和调试，并且不强调效率问题时，可以采用递归。 * 在要求高性能的情况下尽量避免使用递归，递归调用既花时间又耗内存。 |

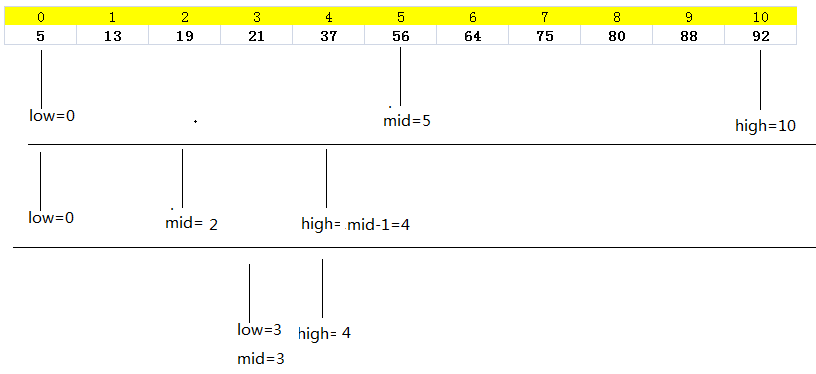
### 2.2 折半查找

折半查找又称为二分查找，这种查找方法需要待查的查找表满足两个条件：

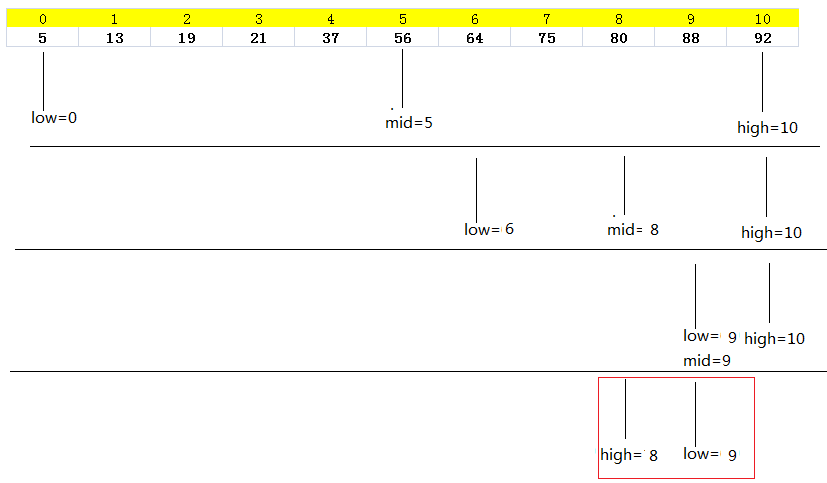
首先，查找表必须使用顺序存储结构；

其次，查找表必须按关键字大小有序排列。

key=21的查找过程

****

key=85的查找过程



#### 【示例6】非递归的折半查找

|  |
| --- |
| **public class** BinarySearch {  **public static void** main(String[] args) {  **int**[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  System.**out**.println(searchLoop(array, 101));  }  **public static int** searchLoop(**int**[] array, **int** findValue) {  // 如果数组为空，直接返回-1，即查找失败  **if** (array == **null**) { **return** -1; }  // 起始位置  **int** start = 0;  // 结束位置  **int** end = array.**length** - 1;  **while** (start <= end) {  // 中间位置  **int** middle = (start + end) / 2;  // 中值  **int** middleValue = array[middle];  **if** (findValue == middleValue) {  // 等于中值直接返回  **return** middle;  } **else if** (findValue < middleValue) {  // 小于中值时在中值前面找  end = middle - 1;  } **else** {  // 大于中值在中值后面找  start = middle + 1;  }  }  // 返回-1，即查找失败  **return** -1;  } } |

#### 【示例7】递归的折半查找

|  |
| --- |
| **public class** BinarySearch {  **public static void** main(String[] args) {  **int**[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  System.***out***.println(*binSearch*(array, 10));  }  **public static int** binSearch(**int** array[], **int** key) {  **int** start = 0;  **int** end = array.**length** - 1;  **return** *binSearch*(array, start, end, key);  }  **public static int** binSearch(**int** array[], **int** start, **int** end, **int** key) {  **int** mid = (end + start) / 2;  **if** (start > end) {  **return** -1;  }  **if** (array[mid] == key) {  **return** mid;  } **else if** (key > array[mid]) {  **return** *binSearch*(array, mid + 1, end, key);  } **else** {  **return** *binSearch*(array, start, mid - 1, key);  }  } } |

### 本节作业

1. 使用递归实现n！
2. 使用递归实现斐波那契数列
3. 使用非递归算法实现折半查找
4. 使用递归算法实现折半查找

## 三、数据结构基础

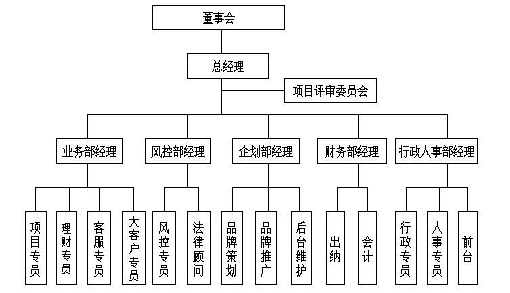
### 3.1数据结构概述

#### 3.1.1什么是数据结构

**数据结构 （不是建筑结构、人体结构）**

数据结构（data structure）是指相互间存在一种或多种特定关系的数据元素的集合。是组织并存储数据以便能够有效使用的一种专门格式，它用来反映一个数据的内部构成，即一个数据由那些成分数据构成，以什么方式构成，呈什么结构。





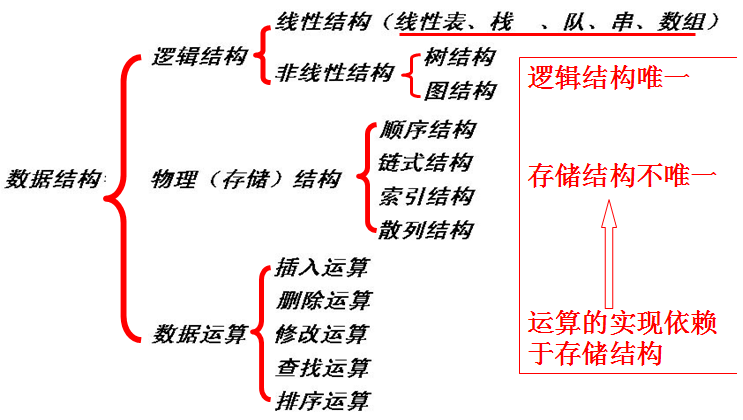
由于信息可以存在于逻辑思维领域，也可以存在于计算机世界，因此作为信息载体的数据同样存在于两个世界中。

表示一组数据元素及其相互关系的数据结构同样也有两种不同的表现形式，

一种是数据结构的逻辑层面，即数据的逻辑结构；

一种是存在于计算机世界的物理层面，即数据的存储结构。

**数据结构=逻辑结构+存储结构+（在存储结构上的）运算/操作**



#### 3.1.2数据的逻辑结构

数据的逻辑结构指数据元素之间的逻辑关系（和实现无关）。

线性结构：有且只有一个开始结点和一个终端结点，并且所有结点都最多只有一个直接前驱和一个直接后继。

线性表就是一个典型的线性结构，它有四个基本特征：

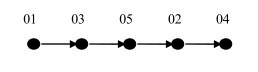
1．集合中必存在唯一的一个"第一个元素"；

2．集合中必存在唯一的一个"最后的元素"；

3．除最后元素之外，其它数据元素均有唯一的"直接后继"；

4．除第一元素之外，其它数据元素均有唯一的"直接前驱"。

线性结构：数据元素之间存在着"一对一"的线性关系的数据结构。

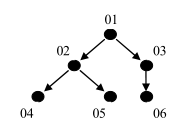


生活案例：冰糖葫芦、 排队上地铁



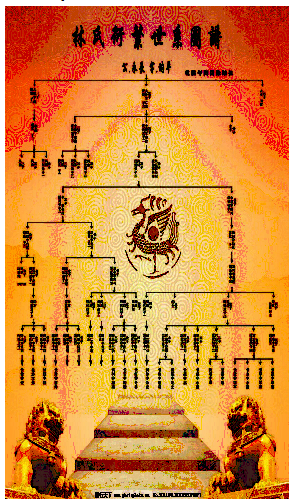
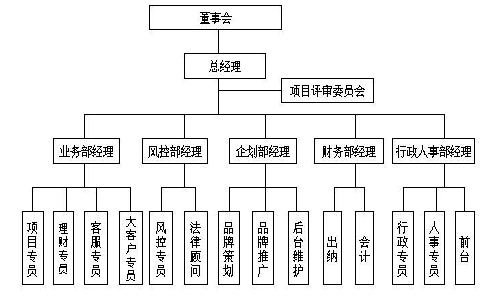
**树状结构**：除了一个数据元素（元素 01）以外每个数据元素有且仅有一个直接前驱元素，但是可以有多个直接后续元素。

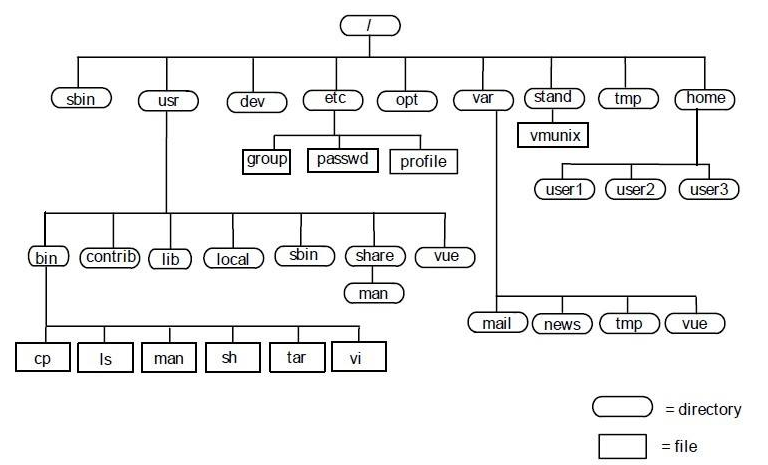
特点是数据元素之间是 1 对 多的联系



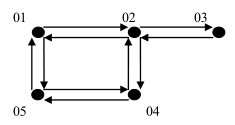
生活案例：单位组织架构、族谱

技术案例：文件系统。

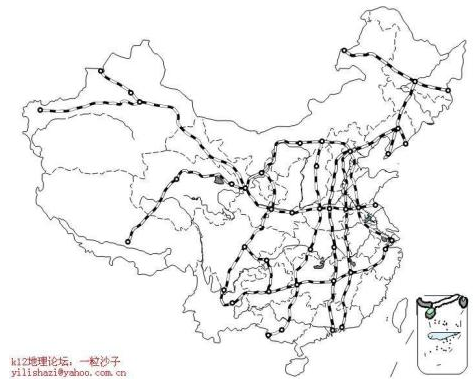
  



**网络结构**：每个数据元素可以有多个直接前驱元素，也可以有多个直接后续元素。特点是数据元素之间是多对 多 的联系

****

**生活案例：交通线路图，地铁图**

  ****

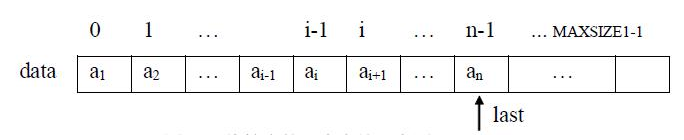
#### 3.1.3数据的存储结构

数据的存储结构主要包括数据元素本身的存储以及数据元素之间关系表示，是数据的逻辑结构在计算机中的表示。包括 顺序存储、链式存储、索引存储，以及散列存储四种。

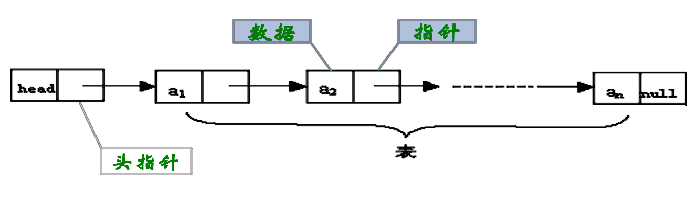
**顺序存储结构**：把逻辑上相邻的节点存储在物理位置上相邻的存储单元中，结点之间的逻辑关系由存储单元的邻接关系来体现。

由此得到的存储结构为顺序存储结构，通常顺序存储结构是借助于计算机程序设计语言（例如C/C++）的数组来描述的。

（数据元素的存储对应于一块连续的存储空间，数据元素之间的前驱和后续关系通过数据元素，在存储器中的相对位置来反映）

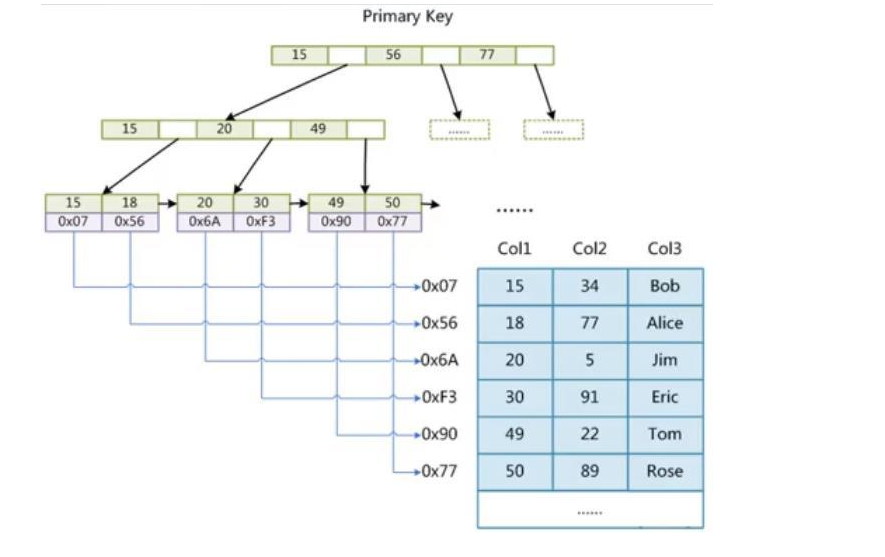


**链式存储结构**：数据元素的存储对应的是不连续的存储空间，每个存储节点对应一个需要存储的数据元素。每个结点是由数据域和指针域组成。元素之间的逻辑关系通过存储节点之间的链接关系反映。逻辑上相邻节点物理上不必相邻。



**索引存储结构**：除建立存储结点信息外，还建立附加的索引表来标识结点的地址。

比如图书、字典的目录



**散列存储结构**：根据结点的关键字直接计算出该结点的存储地址，比如Java中的HashSet、HashMap底层就是散列存储结构。这是一种神奇的结构，添加、查询速度快。



**注意：**

1. **同一逻辑结构可以对应多种存储结构。**
2. **同样的运算，在不同的存储结构中，其实现过程是不同的**

### 3.2线性表

#### 3.2.1线性表定义

**线性表是n个类型相同数据元素的有限序列，通常记作（a0 , a 1 , …a i-1 , a i , a i+1 …,a n-1 ）。**

**1.相同数据类型**

从a 0到a n-1 的n个数据元素是具有相同属性的元素。

比如说可都是数字，例如(23, 14, 66, 5, 99)；也可以是字符，例如(A, B, C, … Z)；

当然也可以是具有更复杂结构的数据元素，例如学生、商品、装备。

**2.序列(顺序性)**

相同数据类型意味着在内存中存储时，每个元素会占用相同的内存空间，便于后续的查询定位。

在线性表的相邻数据元素之间存在着序偶关系，

即a i-1 是a i 的直接前驱，则a i 是a i-1 的直接后续，同时a i 又是a i+1 的直接前驱，a i+1 是a i 的直接后续。

唯一没有直接前驱的元素a 0称为表头，唯一没有后续的元素a n-1称为表尾。

除了表头和表尾元素外，任何一个元素都有且仅有一个直接前驱和直接后继。

**3.有限**

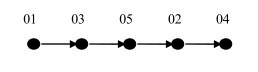
线性表中数据元素的个数n定义为线性表的长度，n是一个有限值。

当n=0 时线性表为空表。

在非空的线性表中每个数据元素在线性表中都有唯一确定的 序号，例如a0 的序号是0，ai 的序号是i。

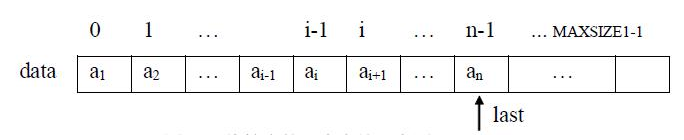
在一个具有n > 0 个数据元素的线性表中，数据元素序号的范围是[0, n-1]。

线性表的逻辑结构如图所示：

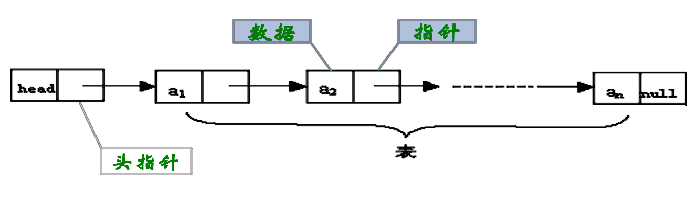


线性表逻辑结构对应的顺序存储结构为顺序表，对应的链式存储结构为链表。

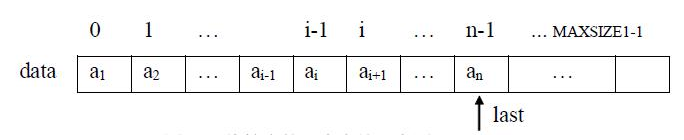
顺序表



链表



#### 3.2.2顺序表----顺序存储结构



**特点**：在内存中分配连续的空间，只存储数据，不需要存储地址信息。位置就隐含着地址。

**优点**：

1.节省存储空间，因为分配给数据的存储单元全用存放结点的数据（不考虑c/c++语言中数组需指定大小的情况），结点之间的逻辑关系没有占用额外的存储空间。

2.索引查找效率高，即每一个结点对应一个序号，由该序号可以直接计算出来结点的存储地址。

假设线性表的每个数据元素需占用K个存储单元，并以元素所占的第一个存储单元的地址作为数据元素的存储地址。则线性表中序号为i的数据元素的存储地址LOC(ai)与序号为i+1 的数据元素的存储地址LOC(a i+1)之间的关系为

        LOC(a i+1 ) = LOC(ai ) + K

通常来说，线性表的i号元素a i 的存储地址为

        LOC(a i ) = LOC(a 0 ) + i×K

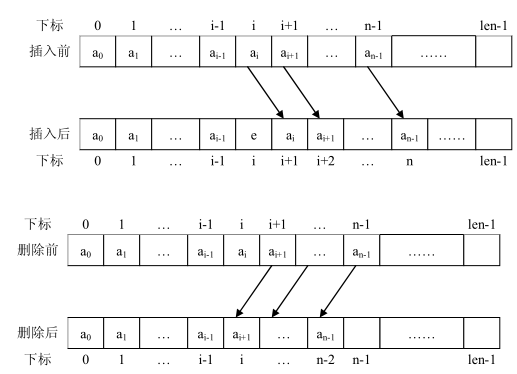
其中LOC(a 0 )为 0 号元素a 0 的存储地址，通常称为线性表的起始地址。

**缺点：**

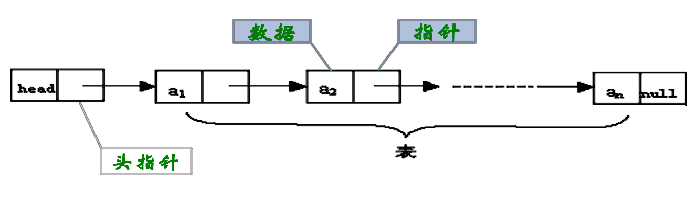
1.插入和删除操作需要移动元素，效率较低。

2.必须提前分配固定数量的空间，如果存储元素少，可能导致空闲浪费。

3.按照内容查询效率低，因为需要逐个比较判断



#### 3.2.3链表----链式存储结构

****

**特点**：数据元素的存储对应的是不连续的存储空间，每个存储结点对应一个需要存储的数据元素。每个结点是由数据域和指针域组成。 元素之间的逻辑关系通过存储节点之间的链接关系反映出来。逻辑上相邻的节点物理上不必相邻。

**缺点：**

1、比顺序存储结构的存储密度小 (每个节点都由数据域和指针域组成，所以相同空间内假设全存满的话顺序比链式存储更多)。

2、查找结点时链式存储要比顺序存储慢（每个节点地址不连续、无规律，导致按照索引查询效率低下）。

**优点：**

1、插入、删除灵活 (不必移动节点，只要改变节点中的指针，但是需要先定位到元素上)。

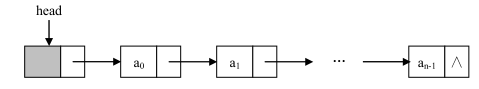
2、有元素才会分配结点空间，不会有闲置的结点。

在使用单链表实现线性表的时候，**为了使程序更加简洁，我们通常在单链表的最前面添加一个哑元结点，也称为头结点。**

在头结点中不存储任何实质的数据对象，其 next 域指向线性表中 0 号元素所在的结点，

**可以对空表、非空表的情况以及对首元结点进行统一处理，编程更方便,常用头结点**。

 一个带头结点的单链表实现线性表的结构图如图 所示。



#### 3.2.4其他链表

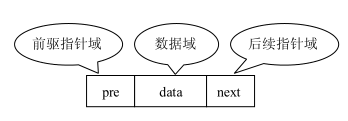
**双向链表**

单链表一个优点是结构简单，但是它也有一个缺点，即在单链表中只能通过一个结点的引用访问其后续结点，而无法直接访问其前驱结点，

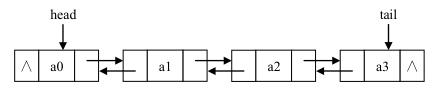
要在单链表中找到某个结点的前驱结点，必须从链表的首结点出发依次向后寻找，但是需要Ο(n)时间。

为此我们可以扩展单链表的结点结构，使得通过一个结点的引用，不但能够访问其后续结点，也可以方便的访问其前驱结点。

扩展单链表结点结构的方法是，在单链表结点结构中新增加一个域，该域用于指向结点的直接前驱结点。扩展后的结点结构是构成双向链表的结点结构，如图 所示。



双向链表是通过上述定义的结点使用 pre 以及 next 域依次串联在一起而形成的。一个双向链表的结构如图所示。



在双向链表中同样需要完成数据元素的查找、插入、删除等操作。在双向链表中进行查找与在单链表中类似，只不过在双向链表中查找操作可以从链表的首结点开始，也可以从尾结点开始，但是需要的时间和在单链表中一样。**Java中的LinkedList底层使用的就是双向链表。**

**循环链表**

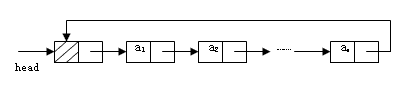
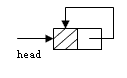
在一个循环链表中, 首节点和末节点被连接在一起。这种方式在单向和双向链表中皆可实现。要遍历一个循环链表，你开始于任意一个节点然后沿着列表的任一方向直到返回开始的节点。循环链表可以被视为"无头无尾"。

循环链表中第一个节点之前就是最后一个节点，反之亦然。

循环链表的无边界使得在这样的链表上设计算法会比普通链表更加容易。

对于新加入的节点应该是在第一个节点之前还是最后一个节点之后可以根据实际要求灵活处理，区别不大。

单向链表的循环带头结点的非空链表和空链表如图所示

|  |
| --- |
| 双向链表的循环带头结点的非空链表    双向链表的循环带头结点的空链表 |

### 本节作业

1. 数据的逻辑结构有哪些，各有什么特点
2. 数据的存储结构有哪些，各有什么特点
3. **顺序存储结构和链式存储结构的特点、优缺点对比**
4. 辨析概念：单链表、双向链表、循环链表

## 四、数据结构基础

### 4.1栈和队列

#### 4.1.1栈

**栈的定义**

栈（stack ）又称堆栈，它是**运算受限**的线性表。其限制是仅允许在表的一端进行插入和删除操作，不允许在其他任何位置进行插入、查找、删除等操作。

表中进行插入、删除操作的一端称为 栈顶（top），栈顶保存的元素称为栈顶元素。相对的，表的另一端称为栈底（bottom）

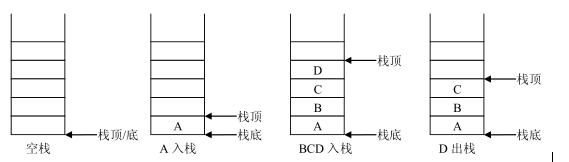
当栈中没有数据元素时称为空栈；

向一个栈插入元素又称为 进栈或 入栈 push；

从一个栈中删除元素又称为 出栈或 退栈  pop。

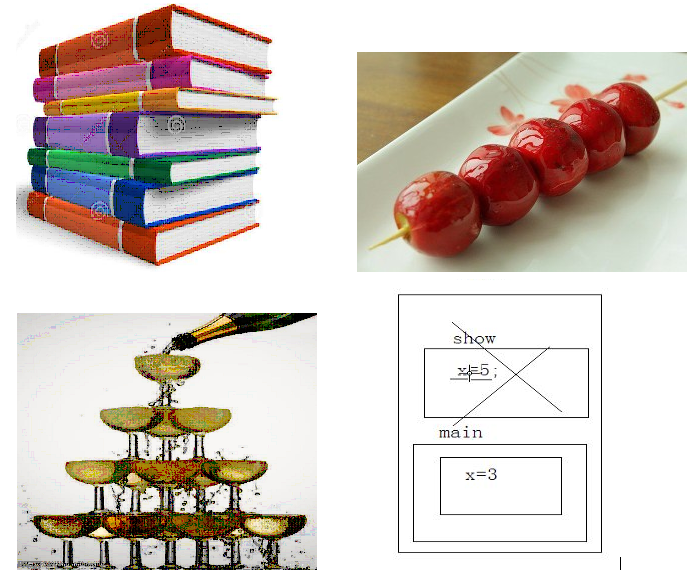
由于栈的插入和删除操作仅在栈顶进行，后进栈的元素必定先出栈，

所以又把堆栈称为 **先进后出表**（Last In First Out，简称 LIFO）



生活案例：摞盘子和取盘子、一摞书、酒杯塔（各层间可以简单理解为栈，每层内部不是栈） 。

技术案例：Java的栈内存



#### 4.1.2队列

**队列定义**

队列（queue）简称队，它同堆栈一样，也是一种运算受限的线性表，其限制是仅允许在表的一端进行插入，而在表的另一端进行删除。

在队列中把插入数据元素的一端称为 队尾（rear），删除数据元素的一端称为队首（front）。

向队尾插入元素称为 进队或入队，新元素入队后成为新的队尾元素；从队列中删除元素称为离队或出队，元素出队后，其后续元素成为新的队首元素。

由于队列的插入和删除操作分别在队尾和队首进行，每个元素必然按照进入的次序离队，也就是说先进队的元素必然先离队，所以称队列为 先进先出表（First In First Out,简称FIFO）。

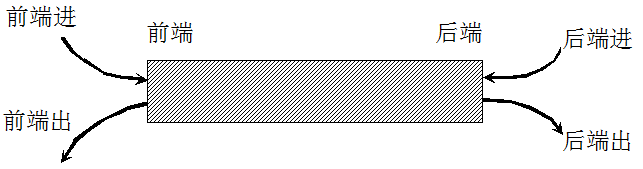




#### 4.1.3双端队列deque

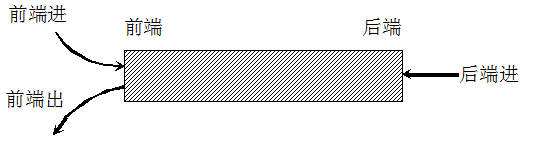
double ended queue  通常读为"deck"

所谓双端队列是指两端都可以进行进队和出队操作的队列，如下图所示，将队列的两端分别称为前端和后端，两端都可以入队和出队。其元素的逻辑结构仍是线性结构

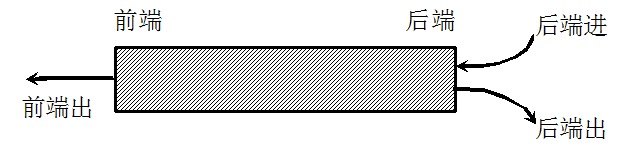


在双端队列进队时：前端进的元素排列在队列中后端进的元素的前面，后端进的元素排列在队列中前端进的元素的后面。在双端队列出队时，无论前端出还是后端出，先出的元素排列在后出的元素的前面。

**输出受限的双端队列**，即一个端点允许插入和删除，另一个端点只允许插入的双端队列。



**输入受限的双端队列，**即一个端点允许插入和删除，另一个端点只允许删除的双端队列。



**双端队列既可以用来队列操作，也可以用来实现栈操作（只操作一端就是栈了）**

### 4.2树和二叉树

#### 4.2.1树

树是由一个集合以及在该集合上定义的一种关系构成的。集合中的元素称为树的结点，所定义的关系称为父子关系。

父子关系在树的结点之间建立了一个层次结构。

树的结点包含一个数据元素及若干指向其子树的若干分支。

在这种层次结构中有一个结点具有特殊的地位，这个结点称为该树的根结点，或简称为树根。

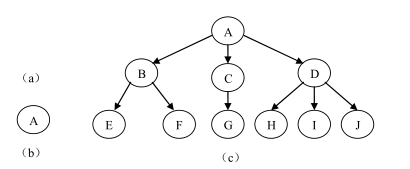
我们可以形式地给出树的递归定义如下：

**树（tree ）是 n（n ≥ 0）个结点的有限集**。它

1) 或是一棵空树（n = 0），空树中不包含任何结点。

2) 或是一棵非空树（n > 0），此时有且仅有一个特定称为根（root ）的结点；

当n > 1 时，其余结点可分为m（m > 0）个互不相交的有限集T 1 ，T 2 ，…，T m，其中每一个本身又是一棵树，并且称为根的 子树（sub tree）。

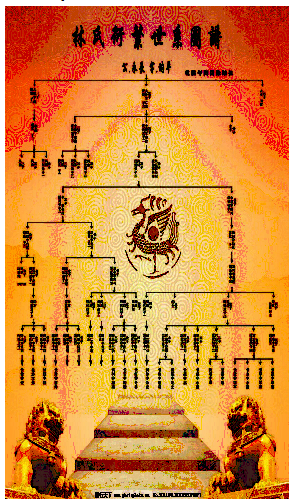
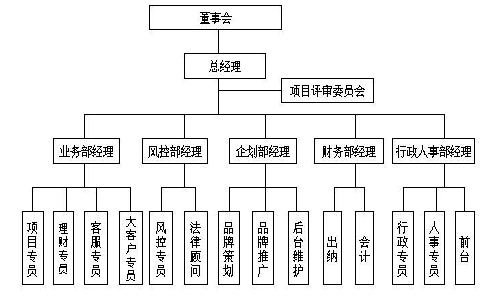


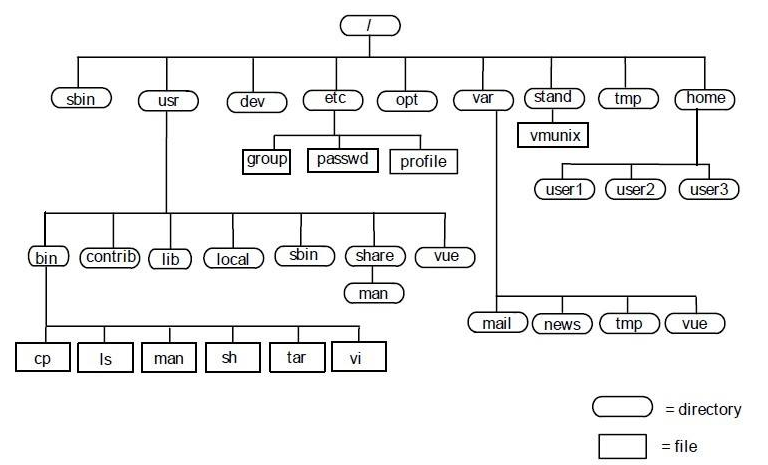
例如图 （a）是一棵空树、（b）是只有一个根节点的树、（c）是一棵有 10个结点的树，其中A是根，

其余的结点分成 3 个不相交的集合：T1 ={B,E,F}、T2 ={C,G}、T3 ={D,H,I,J}，每个集合都构成一棵树，且都是根A的子树。

生活案例：树：单位组织架构、族谱

技术案例：文件系统。

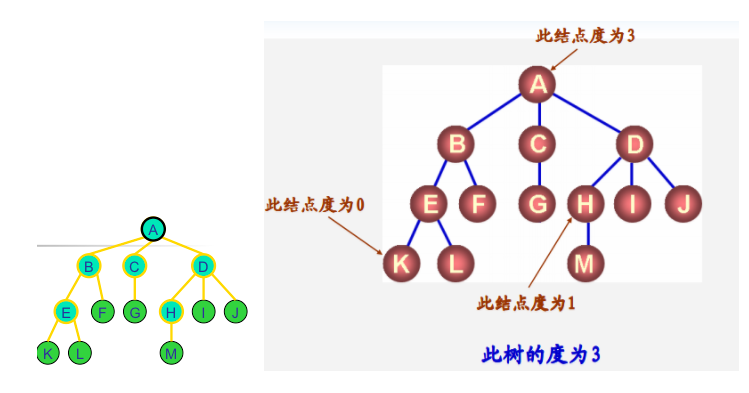


**结点的度与树的度**

结点拥有的子树的数目称为结点的 度（Degree）。

度为0的结点称为叶子（leaf ）或终端结点。度不为 0 的结点称为 非终端结点或 分支结点。除根之外的分支结点也称为内部结点。

树内各结点的度的最大值称为树的度。



**父亲、儿子、兄弟**

父亲（parent）：一个结点的直接前驱结点

儿子（child）：一个结点的直接后继结点

兄弟（sibling） ：同一个父亲结点的其他结点

结点 A 是结点 B、C、D 的父亲，结点 B、C、D 是结点 A 的孩子。

由于结点 H、I、J 有同一个父结点 D，因此它们互为兄弟。

**祖先、子孙、堂兄弟**

将父子关系进行扩展，就可以得到祖先、子孙、堂兄弟等关系。

结点的 祖先是从根到该结点路径上的所有结点。

以某结点为根的树中的任一结点都称为该结点的 子孙。

父亲在同一层次的结点互为 堂兄弟

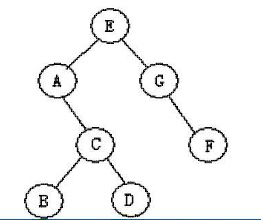
#### 4.2.2二叉树

**二叉树**：

每个结点的度均不超过 2 的**有序**树，称为 二叉树（binary tree） 。

与树的递归定义类似，二叉树的递归定义如下：

二叉树或者是一棵空树，或者是一棵由一个根结点和两棵互不相交的分别称为根的左子树和右子树的子树所组成的非空树。

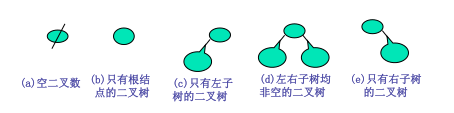


由以上定义可以看出，

二叉树中每个结点的孩子数只能是 0、1 或 2 个，并且每个孩子都有左右之分。

位于左边的孩子称为左孩子，位于右边的孩子称为右孩子；

以左孩子为根的子树称为左子树，以右孩子为根的子树称为右子树。



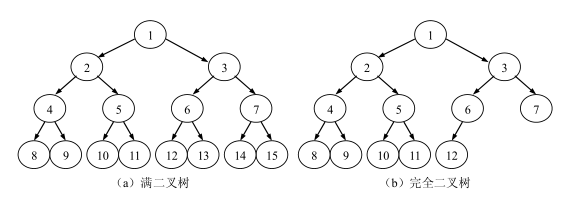
**满二叉树：**

高度为k并且有 2k+1 -1 个结点的二叉树。

满二叉树中，每层结点都达到最大数，即每层结点都是满的，因此称为满二叉树。

**完全二叉树：**

若在一棵满二叉树中，在最下层从最右侧起去掉相邻的若干叶子结点，得到的二叉树即为完全二叉树。



**满二叉树必为完全二叉树，而完全二叉树不一定是满二叉树**

**二叉树的存储结构**

二叉树存储结构有两种：顺序存储结构和链式存储结构。更多使用链式存储结构

**链式存储结构**

设计不同的结点结构可构成不同的链式存储结构。

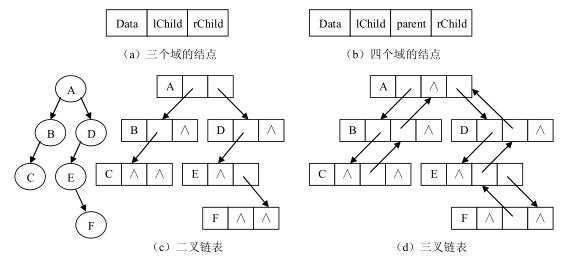
在二叉树中每个结点都有两个孩子，则可以设计每个结点至少包括 3 个域：数据域、左孩子域和右孩子域。

数据域存放数据元素，左孩子域存放指向左孩子结点的指针，右孩子域存放指向右孩子结点的指针。如图 （a）所示。

利用此结点结构得到的二叉树存储结构称为二叉链表。

为了方便找到父结点，可以在上述结点结构中增加一个指针域，指向结点的父结点。如图 （b）所示。

采用此结点结构得到的二叉树存储结构称为三叉链表。



#### 4.2.3查找树

**二叉查找/搜索/排序树  BST  (binary search/sort tree)**

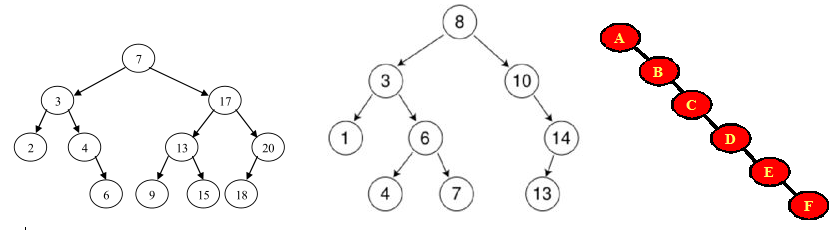
或者是一棵空树；

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它的根节点的值；

（2）若它的右子树上所有结点的值均大于它的根节点的值；

（3）它的左、右子树也分别为二叉排序树。



**注意：对二叉查找树进行中序遍历，得到有序集合。**

**平衡二叉树**（Self-balancing binary search tree）  自平衡二叉查找树  又被称为AVL树（有别于AVL算法）

它是一 棵空树

或它的左右两个子树的高度差(平衡因子)的绝对值不超过1，

并且左右两个子树都是一棵平衡二叉树，

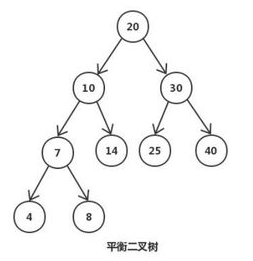
同时，平衡二叉树必定是二叉搜索树，反之则不一定

平衡因子（平衡度）：结点的平衡因子是结点的左子树的高度减去右子树的高度。（或反之定义）

平衡二叉树：每个结点的平衡因子都为 1、－1、0 的二叉排序树。或者说每个结点的左右子树的高度最多差1的二叉排序树。

平衡二叉树的目的是为了减少二叉查找树层次，提高查找速度

平衡二叉树的常用实现方法有AVL、红黑树、替罪羊树、Treap、伸展树等



**红黑树**

R-B Tree，全称是Red-Black Tree，又称为"红黑树"，它一种平衡二叉树。红黑树的每个节点上都有存储位表示节点的颜色，可以是红(Red)或黑(Black)。

红黑树的特性:

（1）每个节点或者是黑色，或者是红色。

（2）根节点是黑色。

（3）每个叶子节点（NIL）是黑色。 [注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！]

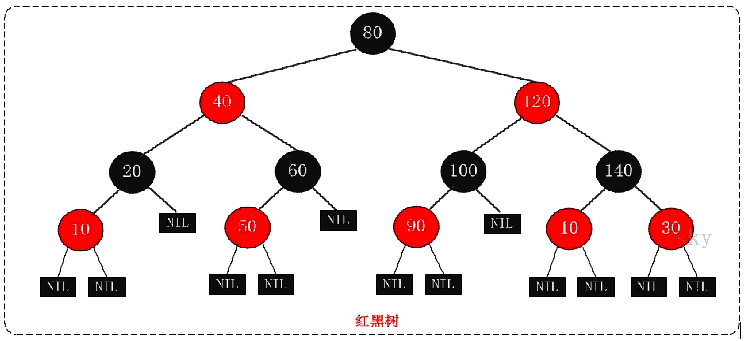
（4）如果一个节点是红色的，则它的子节点必须是黑色的。

（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

注意：

(01) 特性(3)中的叶子节点，是只为空(NIL或null)的节点。

(02) 特性(5)，确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树



红黑树的应用比较广泛，主要是用它来存储有序的数据，它的时间复杂度是O(logN)，效率非常之高。

它虽然是复杂的，但最坏情况运行时间也是非常良好的，并且在实践中是高效的： 它可以在O(log n)时间内做查找，插入和删除，这里的n 是树中元素的数目。

例如，Java集合中的TreeSet和TreeMap，C++ STL中的set、map，以及Linux虚拟内存的管理，都是通过红黑树去实现的。

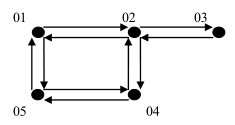
**区别**

1、红黑树放弃了追求完全平衡，追求大致平衡，在与平衡二叉树的时间复杂度相差不大的情况下，保证每次插入最多只需要三次旋转就能达到平衡，实现起来也更为简单。

2、平衡二叉树追求绝对平衡，条件比较苛刻，实现起来比较麻烦，每次插入新节点之后需要旋转的次数不能预知。

### 4.3图

1.图的基本概念：多对多关系





图（graph）是一种网状数据结构，图是由非空顶点集合和一个描述顶点间关系的集合组成。

其形式化的定义如下：

Graph = ( V , E )

V = {x| x∈某个数据对象}

E = {<u , v>| P(u , v)∧(u,v∈V)}

V 是具有相同特性的数据元素的集合，V 中的数据元素通常称为顶点（Vertex） ），

E 是两个顶点之间关系的集合。P(u , v)表示 u 和 v 之间有特定的关联属性。

若<u , v>∈E，则<u , v>表示从顶点 u 到顶点 v 的一条弧，并称 u 为弧尾或起始点，称v 为弧头或终止点，

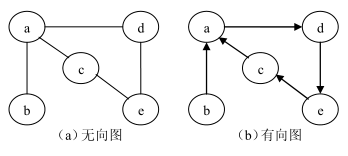
此时图中顶点之间的连线是有方向的，这样的图称为**有向图（directedgraph）**。

若<u , v>∈E 则必有<v , u>∈E，即关系 E 是对称的，此时可以使用一个无序对(u , v)来代替两个有序对，

它表示顶点 u 和顶点 v 之间的一条边，此时图中顶点之间的连线是没有方向的，这种图称为 **无向图（undirected graph）**。

在无向图和有向图中 V 中的元素都称为顶点，而顶点之间的关系却有不同的称谓，即弧或边，为避免麻烦，在不影响理解的前提下，我们统一的将它们称为 边（edge） 。

并且我们还约定顶点集与边集都是有限的，并记顶点与边的数量为|V|和|E|。



无向图实际上也是有向图，是双向图

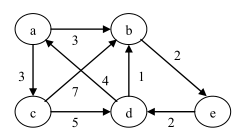
**加权图：**

在实际应用中，图不但需要表示元素之间是否存在某种关系，

而且图的边往往与具有一定实际意义的数有关，即每条边都有与它相关的实数，称为权。

这些权值可以表示从一个顶点到另一个顶点的距离或消耗等信息，在本章中假设边的权均为正数。

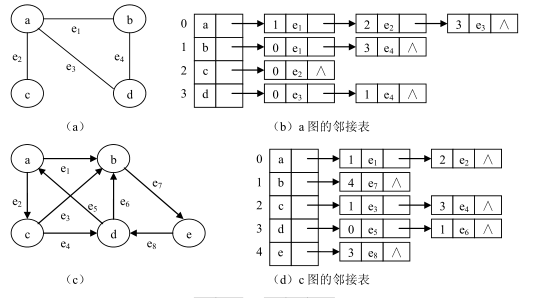
这种边上具有权值的图称为 带权图（weighted graph）



**图的存储结构**

可以采用顺序存储结构和链式存储结构，更多采用链式存储结构

**邻接表：**链表  链式存储结构



### 本节作业

1. 线性表、栈、队列和联系和区别。
2. 双端队列的特点和应用
3. 满二叉树和完全二叉树的关系
4. 二叉树、二叉查找树、平衡二叉树、红黑树的关系
5. 无向图、有向图、加权图的定义