# 散列：

散列是一种常用的数据存储技术，散列后的数据可以快速地插入或取用。散列使用的数据

结构叫做散列表。在散列表上插入、删除和取用数据都非常快，但是对于查找操作来说却

效率低下，比如查找一组数据中的最大值和最小值。这些操作得求助于其他数据结构，二

叉查找树就是一个很好的选择。

要确定的最后一个问题是：散列表中的数组究竟应该有多大？这是编写散列函数时必须要

考虑的。对数组大小常见的限制是：数组长度应该是一个质数。

# 集合 set

集合的两个最重

要特性是：首先，集合中的成员是无序的；其次，集合中不允许相同成员存在

# 图

图由边的集合及顶点的集合组成。

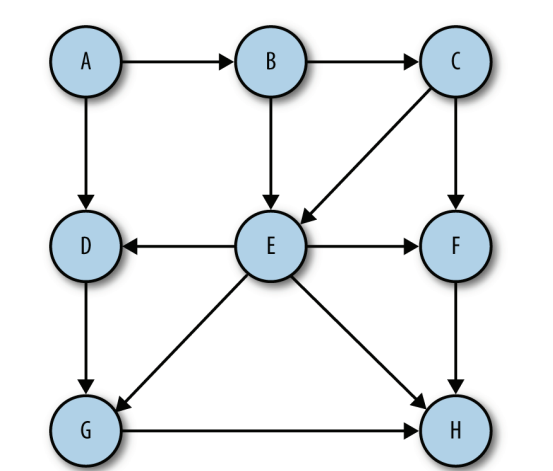
边由顶点

对 (v1,v2) 定义，v1 和 v2 分别是图中的两个顶点。顶点也有权重，也称为成本。如果一个

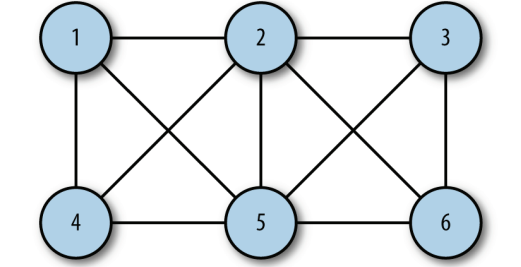
图的顶点对是有序的，则可以称之为有向图。在对有向图中的顶点对排序后，便可以在两

个顶点之间绘制一个箭头。有向图表明了顶点的流向。计算机程序中用来表明计算方向的

流程图就是一个有向图的例子。



如果图是无序的，则称之为无序图，或无向图。图 11-2 展示了一个无序图。



## 1.2 用图对现实中的系统建模

可以用图对现实中的很多系统建模。比如对交通流量建模，顶点可以表示街道的十字路

口，边可以表示街道。加权的边可以表示限速或者车道的数量。建模人员可以用这个系统

来判最佳路线及最有可能堵车的街道。

任何运输系统都可以用图来建模。比如，航空公司可以用图来为其飞行系统建模。将每个

机场看成顶点，将经过两个顶点的每条航线看作一条边。加权的边可以表示从一个机场到

另一个机场的航班成本，或两个机场间的距离，这取决于建模的对象是什么。

包含局域网和广域网（如互联网）在内的计算机网络，同样经常用图来建模。另一个可以

用图来建模的现实系统是消费市场，顶点可以用来表示供应商和消费者。

## 1.3建模

Vertex 类

## 1.4搜索图

确定从一个指定的顶点可以到达其他哪些顶点，这是经常对图执行的操作。我们可能想通

过地图了解到从一个城镇到另一个城镇有哪些路，或者从一个机场到其他机场有哪些航班。

图上的这些操作是用搜索算法执行的。在图上可以执行两种基础搜索：深度优先搜索和广

度优先搜索。本节将仔细研究这两种算法。

### 1.4.1 深度优先搜索

深度优先搜索包括从一条路径的起始顶点开始追溯，直到到达最后一个顶点，然后回溯，

继续追溯下一条路径，直到到达最后的顶点，如此往复，直到没有路径为止。这不是在搜

索特定的路径，而是通过搜索来查看在图中有哪些路径可以选择。

### 1.4.2 广度优先搜索

广度优先搜索从第一个顶点开始，尝试访问尽可能靠近它的顶点。本质上，这种搜索在图

上是逐层移动的，首先检查最靠近第一个顶点的层，再逐渐向下移动到离起始顶点最远的

层。

## 1.5 查找最短路径

广度优先搜索对应的最短路径，代码有很大的问题

## 1.6拓扑排序 图论之后再看

# 排序算法

## 冒泡排序

下冒泡排序算法，它是最慢的排序算法之一，但也是一种最容易实现的排

序算法。

之所以叫冒泡排序是因为使用这种排序算法排序时，数据值会像气泡一样从数组的一端漂

浮到另一端。假设正在将一组数字按照升序排列，较大的值会浮动到数组的右侧，而较小

的值则会浮动到数组的左侧。

伪代码描述：

1. 获取数组的长度
2. 每次遍历数组的长度减一（因为每遍历一次，最大的都会跑到最后面，下次不需要再遍历）
3. 内部遍历数组开始到长度减一的元素，进行比较，然后排序。

## 选择排序

选择排序从数组的开头开始，将第一个元素和其他元

素进行比较。检查完所有元素后，最小的元素会被放到数组的第一个位置，然后算法会从

第二个位置继续。这个过程一直进行，当进行到数组的倒数第二个位置时，所有的数据便

完成了排序。

选择排序会用到嵌套循环。外循环从数组的第一个元素移动到倒数第二个元素；内循环从第

二个数组元素移动到最后一个元素，查找比当前外循环所指向的元素小的元素。每次内循环

迭代后，数组中最小的值都会被赋值到合适的位置。

## 插入排序

插入排序有两个循环。外循环将数组元素挨个移动，而内循环则对外循环中选中的元素及

它后面的那个元素进行比较。如果外循环中选中的元素比内循环中选中的元素小，那么数

组元素会向右移动，为内循环中的这个元素腾出位置，就像之前介绍的姓氏卡片一样

function insertionSort() {

var temp, inner;

for (var outer = 1; outer <= this.dataStore.length - 1; ++outer) {

temp = this.dataStore[outer];

inner = outer;

while (inner > 0 && (this.dataStore[inner - 1] >= temp)) {

this.dataStore[inner] = this.dataStore[inner - 1];

--inner;

}

this.dataStore[inner] = temp;

}

}

## 基本算法比较

大致上可以得出这么个结论：

选择排序和插入排序要比冒泡排序快，插入排序是这三种算法中最快的。

# 高级排序算法

法。它们通常被认为是处理大型数据集的最高效排序算

法，它们处理的数据集可以达到上百万个元素，而不仅仅是几百个或者几千个。这一节将

介绍的算法包括快速排序、希尔排序、归并排序和堆排序。我们会讨论每个算法的实现，

并通过运行计时测试来比较它们的效率。

## 希尔排序

# 检索算法

## 顺序查找

对于查找数据来说，最简单的方法就是从列表的第一个元素开始对列表元素逐个进行判

断，直到找到了想要的结果，或者直到列表结尾也没有找到。这种方法称为顺序查找，有

时也被称为线性查找。它属于暴力查找技巧的一种，在执行查找时可能会访问到数据结构

里的所有元素。

## 二分查找算法

如果你要查找的数据是有序的，二分查找算法比顺序查找算法更高效。

# 高级算法

## 动态规划

使用递归去解决问题虽然简洁，但效率不高。包括 JavaScript 在内的众多语言，不能高效

地将递归代码解释为机器代码，尽管写出来的程序简洁，但是执行效率低下。但这并不是

说使用递归是件坏事，本质上说，只是那些指令式编程语言和面向对象的编程语言对递归

的实现不够完善，因为它们没有将递归作为高级编程的特性。

动态规划方案通常会使用一个数组来建立一张表，用于存放被分解成众多子问题的解。

和递归算法相对立。

## 贪心算法

它可以用于优化通过次优算法找到的解决方案——这些

方案通常是基于递归方案实现的。对许多问题来说，采用动态规划的方式去处理有点大材

小用，往往一个简单的算法就够了。

贪心算法就是一种比较简单的算法。贪心算法总是会选择当下的最优解，而不去考虑这一

次的选择会不会对未来的选择造成影响。使用贪心算法通常表明，实现者希望做出的这一

系列局部“最优”选择能够带来最终的整体“最优”选择。如果是这样的话，该算法将会

产生一个最优解，否则，则会得到一个次优解。然而，对很多问题来说，寻找最优解很麻

烦，这么做不值得，所以使用贪心算法就足够了。

# 总结

前面的数据结构还能理解。后面的高级排序和高级算法，能理解，但是不深入，需要通过刷题来解决问题。现在可以开始刷LeetCode了吧。同时也需要继续用js实现c++那个版本的数据结构与算法，还有uestc的算法课。

遗留：

二叉树算法 二叉排序树，AVL，红黑树，B树，B+树，B\*树。 dfs bfs

动态规划和贪心算法。