一、数组

1）为什么很多编程语言中，数组的下标都从0开始编号?

2）数组和链表的区别

3）容器 vs 数组：java中的ArrayList与数组相比，到底有哪些优势呢?

4）二维数组寻址

5）Java 中的JVM、聊一下对标记清除垃圾回收算法的理解

leetcode

二、链表

1）三种最常见的链表结构：

2）链表应用场景

3）如何实现LRU缓存淘汰算法?

4）写链表的6个技巧

leetcode

三、栈

1）栈的结构是什么?

2）栈如何完成表达式中括号的匹配?

3）如何实现浏览器的前进和后退功能?

leetcode

四、队列

1）队列的应用有哪些

2）当我们向固定大小的线程池中请求一个线程时，如果线程池中没有空闲资源了，这个时候线程池如何处理这个请求？是拒绝请求还是排队请求？各种处理策略又是怎么实现的呢?

leetcode

五、递归

1）什么是递归?

2）递归的优点与缺点

3）什么样的问题可以用递归来解决呢？递归的3个条件

5）递归常见问题及解决方案

leetcode

六、排序

1）各种排序算法的时间复杂度对比

2）如何分析一个排序算法

3）电商系统中，订单时间和金额的排序问题

4）有序度和逆序度的概念

5）插入排序和冒泡排序的时间复杂度相同，都是 O(n2)，在实际的软件开发里，为什么我们更倾向于使用插入排序算法而不是冒泡排序算法呢?

6）如何用有限的内存对10个文件进行排序

7）如何根据年龄给100万用户数据排序?

leetcode

七、查找

1）二分查找应用的局限性

2）二分查找的经典变体

3）如何快速定位IP对应的省份地址?

leetcode

八、跳表

1）跳表是什么?

2）跳表的时间复杂度与空间

3）为什么要Redis要用跳表来实现有序集合？（跳表vs红黑树）

九、散列表

1)散列表是什么?

2）Word 文档中单词拼写检查功能是如何实现的?

3）假设我们有 10 万条 URL 访问日志，如何按照访问次数给 URL 排序?

4）有两个字符串数组，每个数组大约有 10 万条字符串，如何快速找出两个数组中相同的字符串?

leetcode

十、哈希算法

1）什么是哈希算法?

2）哈希算法要满足的几点要求

3）哈希算法的7个常见应用

4）为什么哈希算法可以用于安全加密：

5）唯一标识：用哈希算法搜索一张图是否在图库中存在

6）数据校验：如何判断文件下载过程中是否被篡改?

7）如何防止密码被拖库?

8）区块链使用的是哪种哈希算法？是为了解决什么问题而使用的呢?

一、数组

1）为什么很多编程语言中，数组的下标都从0开始编号？

数组（Array）是一种线性表数据结构。它用一组连续的内存空间，来存储一组具有相同类型的数据。因为有连续的内存空间和相同类型的数据，数组就可以实现随机访问。实现随机访问的方式是

a[i]_address = base_address + i * data_type_size

其中 data_type_size 表示数组中每个元素的大小。

如果用 a 来表示数组的首地址，a[0]就是偏移为 0 的位置，也就是首地址，a[k]就表示偏移 k 个 type_size 的位置，所以计算 a[k]的内存地址只需要用这个公式

a[k]_address = base_address + k * type_size

但是，如果数组从 1 开始计数，那我们计算数组元素 a[k]的内存地址就会变为：

a[k]_address = base_address + (k-1)*type_size

对比两个公式，我们不难发现，从 1 开始编号，每次随机访问数组元素都多了一次减法运算，对于 CPU 来说，就是多了一次减法指令。

数组作为非常基础的数据结构，通过下标随机访问数组元素又是其非常基础的编程操作，效率的优化就要尽可能做到极致。所以为了减少一次减法操作，数组选择了从 0 开始编号，而不是从 1 开始。

2）数组和链表的区别

- 链表适合插入、删除，时间复杂度 O(1)；数组支持随机访问，根据下标随机访问的时间复杂度为 O(1)。
- 对内存要求方面: 数组对内存的要求更高。因为数组需要一块连续内存空间来存放数据。（可能出现的问题就是:内存总的剩余空间足够，但是申请容量较大的数组时申请失败） 链表对内存的要求较低，是因为链表不需要连续的内存空间，它通过“指针”将一组零散的内存块串联起来使用。 但是要注意:链表虽然方便。但是内存开销比数组大了将近一倍，
- 数组的缺点是大小固定，一经声明就要占用整块连续内存空间。如果声明的数组过大，系统可能没有足够的连续内存空间分配给它，导致“内存不足（out of memory）”。如果声明的数组过小，则可能出现不够用的情况。这时只能再申请一个更大的内存空间，把原数组拷贝进去，非常费时。链表本身没有大小的限制，天然地支持动态扩容，我觉得这也是它与数组最大的区别。

3）容器 vs 数组：java中的ArrayList与数组相比，到底有哪些优势呢？

容器优势：

1.ArrayList 最大的优势就是可以将很多数组操作的细节封装起来。比如数组插入、删除数据时需要搬移其他数据等。
2.支持动态扩容。
数组优势：
1.Java ArrayList 无法存储基本类型，比如 int、long，需要封装为 Integer、Long 类，而 Autoboxing、Unboxing 则有一定的性能消耗，所以如果特别关注性能，或者希望使用基本类型，就可以选用数组。
2. 如果数据大小事先已知，并且对数据的操作非常简单，用不到 ArrayList 提供的大部分方法，也可以直接使用数组。
3. 还有一个是我个人的喜好，当要表示多维数组时，用数组往往会更加直观。比如 Object[][] array；而用容器的话则需要这样定义：ArrayList > array。
总结：对于业务开发，直接使用容器就足够了，省时省力。毕竟损耗一丢丢性能，完全不会影响到系统整体的性能。但如果你是做一些非常底层的开发，比如开发网络框架，性能的优化需要做到极致，这个时候数组就会优于容器，成为首选。

4）二维数组寻址

对于 m * n 的数组，a [i][j] (i < m,j < n)的地址为：address = base_address + (i * n + j) * type_size

5）Java 中的JVM、聊一下对标记清除垃圾回收算法的理解

大多数主流虚拟机采用可达性分析算法来判断对象是否存活，在标记阶段，会遍历所有 GC ROOTS，将所有 GC ROOTS 可达的对象标记为存活。只有当标记工作完成后，清理工作才会开始。
不足：1.效率问题。标记和清理效率都不高，但是当知道只有少量垃圾产生时会很高效。2.空间问题。会产生不连续的内存空间碎片。

leetcode

二分查找：https://leetcode-cn.com/problems/binary-search/
测试用例：
1）[-1,0,3,5,9,12],2 返回-1
2）[-1,0,3,5,9,12],9 返回4
3）[2,5], 5 返回1

二、链表

1）三种最常见的链表结构：

- 单链表：内存块称作“结点”、链上的下一个结点的地址称作“后继指针”。头结点用于记录链表的基地址，尾结点的后继指针指向一个空地址 NULL。单向链表的插入和删除操作，我们只需要考虑相邻结点的指针改变，所以对应的时间复杂度是 O(1)。
- 双向链表：结点不止有一个后继指针 next 指向后面的结点，还有一个前驱指针 prev 指向前面的结点。
- 循环链表：循环链表的尾结点指针是指向链表的头结点。可用于解决约瑟夫问题：人们站在一个等待被处决的圈子里。 计数从圆圈中的指定点开始，并沿指定方向围绕圆圈进行。 在跳过指定数量的人之后，处刑下一个人。 对剩下的人重复该过程，从下一个人开始，朝同一方向跳过相同数量的人，直到只剩下一个人，并被释放。

2）链表应用场景

缓存是一种提高数据读取性能的技术，在硬件设计、软件开发中都有着非常广泛的应用，比如常见的 CPU 缓存、数据库缓存、浏览器缓存等等。缓存的大小有限，当缓存被用满时，哪些数据应该被清理出去，哪些数据应该被保留？这就需要缓存淘汰策略来决定。常见的策略有三种：先进先出策略 FIFO（First In，First Out）、最少使用策略 LFU（Least Frequently Used）、最近最少使用策略 LRU（Least Recently Used）。

3）如何实现LRU缓存淘汰算法?

我们维护一个有序单链表，越靠近链表尾部的结点是越早之前访问的。当有一个新的数据被访问时，我们从链表头开始顺序遍历链表。1. 如果此数据之前已经被缓存在链表中了，我们遍历得到这个数据对应的结点，并将其从原来的位置删除，然后再插入到链表的头部。2. 如果此数据没有在缓存链表中，又可以分为两种情况：如果此时缓存未满，则将此结点直接插入到链表的头部；如果此时缓存已满，则链表尾结点删除，将新的数据结点插入链表的头部。

4）写链表的6个技巧

技巧一：理解指针或引用的含义
技巧二：警惕指针丢失和内存泄漏

```
// 错误示范：x->next指错
p->next = x;  // 将p的next指针指向x结点；
x->next = p->next;  // 将x的结点的next指针指向b结点；
```

技巧三：利用哨兵简化实现难度：单链表的插入，第一个节点不一样，利用哨兵可以简化操作
技巧四：重点留意边界条件处理，以下情况代码是否能够工作?

- 如果链表为空时
- 如果链表只包含一个结点时
- 如果链表只包含两个结点时
- 代码逻辑在处理头结点和尾结点的时候

技巧五：技巧五：举例画图，辅助思考
技巧六：多写多练，没有捷径

leetcode

单链表反转（206）：https://leetcode-cn.com/problems/reverse-linked-list/ 测试用例[1,2,3,4,5]、[1,2]、[1]
链表中环的检测(141)：https://leetcode-cn.com/problems/linked-list-cycle/
两个有序的链表合并(21):https://leetcode-cn.com/problems/merge-two-sorted-lists/
删除链表倒数第 n 个结点(19):https://leetcode-cn.com/problems/remove-nth-node-from-end-of-list/
求链表的中间结点(876):https://leetcode-cn.com/problems/middle-of-the-linked-list/

三、栈

1）栈的结构是什么？

栈是一种操作受限的数据结构，只支持入栈和出栈操作。后进先出是它最大的特点。栈既可以通过数组实现（称为顺序栈），也可以通过链表来实现（链式栈）。不管基于数组还是链表，入栈、出栈的时间复杂度都为 O(1)。

2）栈如何完成表达式中括号的匹配？

背景：假设表达式中只包含三种括号，圆括号 ()、方括号[]和花括号{}，并且它们可以任意嵌套。比如，{[] ()[{}]}或[{()}([])]等都为合法格式，而{[}()]或[({)]为不合法的格式。那我现在给你一个包含三种括号的表达式字符串，如何检查它是否合法呢？
答：当扫描到左括时，则将其压入栈中；当扫描到右括号时，从栈顶取出一个左括号。如果能够匹配，比如“(”跟“)”匹配，“[”跟“]”匹配，“{”跟“}”匹配，则继续扫描剩下的字符串。如果扫描的过程中，遇到不能配对的右括号，或者栈中没有数据，则说明为非法格式。当所有的括号都扫描完成之后，如果栈为空，则说明字符串为合法格式；否则，说明有未匹配的左括号，为非法格式。

3）如何实现浏览器的前进和后退功能？

背景：当你依次访问完一串页面 a-b-c 之后，点击浏览器的后退按钮，就可以查看之前浏览过的页面 b 和 a。当你后退到页面 a，点击前进按钮，就可以重新查看页面 b 和 c。但是，如果你后退到页面 b 后，点击了新的页面 d，那就无法再通过前进、后退功能查看页面 c 了。
答：我们使用两个栈，X 和 Y，我们把首次浏览的页面依次压入栈 X，当点击后退按钮时，再依次从栈 X 中出栈，并将出栈的数据依次放入栈 Y。当我们点击前进按钮时，我们依次从栈 Y 中取出数据，放入栈 X 中。当栈 X 中没有数据时，那就说明没有页面可以继续后退浏览了。当栈 Y 中没有数据，那就说明没有页面可以点击前进按钮浏览了。

leetcode

有效的括号 https://leetcode-cn.com/problems/valid-parentheses/

四、队列

1）队列的应用有哪些

比如高性能队列 Disruptor、Linux 环形缓存，都用到了循环并发队列；Java concurrent 并发包利用 ArrayBlockingQueue 来实现公平锁等。

2）当我们向固定大小的线程池中请求一个线程时，如果线程池中没有空闲资源了，这个时候线程池如何处理这个请求？是拒绝请求还是排队请求？各种处理策略又是怎么实现的呢？

我们一般有两种处理策略。第一种是非阻塞的处理方式，直接拒绝任务请求；另一种是阻塞的处理方式，将请求排队，等到有空闲线程时，取出排队的请求继续处理。那如何存储排队的请求呢？我们前面说过，队列有基于链表和基于数组这两种实现方式。这两种实现方式对于排队请求又有什么区别呢？基于链表的实现方式，可以实现一个支持无限排队的无界队列（unbounded queue），但是可能会导致过多的请求排队等待，请求处理的响应时间过长。所以，针对响应时间比较敏感的系统，基于链表实现的无限排队的线程池是不合适的。而基于数组实现的有界队列（bounded queue），队列的大小有限，所以线程池中排队的请求超过队列大小时，接下来的请求就会被拒绝，这种方式对响应时间敏感的系统来说，就相对更加合理。不过，设置一个合理的队列大小，也是非常有讲究的。队列太大导致等待的请求太多，队列太小会导致无法充分利用系统资源、发挥最大性能。

leetcode

用队列实现栈（不是很适合用于面试） https://leetcode-cn.com/problems/implement-stack-using-queues/
用栈实现队列（不是很适合用于面试）https://leetcode-cn.com/problems/implement-queue-using-stacks/

五、递归

1）什么是递归?

1.递归是一种非常高效、简洁的编码技巧，一种应用非常广泛的算法，比如DFS深度优先搜索、前中后序二叉树遍历等都是使用递归。
2.方法或函数调用自身的方式称为递归调用，调用称为递，返回称为归。
3.基本上，所有的递归问题都可以用递推公式来表示，比如
f(n) = f(n-1) + 1;
f(n) = f(n-1) + f(n-2);
f(n)=n*f(n-1);

2）递归的优点与缺点

优点：递归代码的表达力很强，写起来非常简洁；
缺点：空间复杂度高、有堆栈溢出的风险、存在重复计算、过多的函数调用会耗时较多等问题。

3）什么样的问题可以用递归来解决呢？递归的3个条件

1. 一个问题的解可以分解为几个子问题的解
2. 这个问题与分解之后的子问题，除了数据规模不同，求解思路完全一样
3. 存在递归终止条件

4）如何实现递归

1.递归代码编写
写递归代码的关键就是找到如何将大问题分解为小问题的规律，并且基于此写出递推公式，然后再推敲终止条件，最后将递推公式和终止条件翻译成代码。
2.递归代码理解
对于递归代码，若试图想清楚整个递和归的过程，实际上是进入了一个思维误区。
那该如何理解递归代码呢？如果一个问题A可以分解为若干个子问题B、C、D，你可以假设子问题B、C、D已经解决。而且，你只需要思考问题A与子问题B、C、D两层之间的关系即可，不需要一层层往下思考子问题与子子问题，子子问题与子子子问题之间的关系。屏蔽掉递归细节，这样子理解起来就简单多了。
因此，理解递归代码，就把它抽象成一个递推公式，不用想一层层的调用关系，不要试图用人脑去分解递归的每个步骤。

5）递归常见问题及解决方案

1）堆栈溢出

- 为什么会造成堆栈溢出

函数调用会使用栈来保存临时变量。每调用一个函数，都会将临时变量封装为栈帧压入内存栈，等函数执行完成返回时，才出栈。系统栈或者虚拟机栈空间一般都不大。如果递归求解的数据规模很大，调用层次很深，一直压入栈，就会有堆栈溢出的风险。

- 如何预防堆栈溢出呢?

通过在代码中限制递归调用的最大深度的方式来解决这个问题。递归调用超过一定深度（比如 1000）之后，我们就不继续往下再递归了，直接返回报错。

2）重复计算

通过某种数据结构来保存已经求解过的值，从而避免重复计算。

leetcode

斐波那契数 https://leetcode-cn.com/problems/fibonacci-number/

六、排序

1）各种排序算法的时间复杂度对比

	时间复杂度	是稳定排序?	是原地排序?
冒泡排序	$O(n^2)$	√	√
插入排序	$O(n^2)$	√	√
选择排序	$O(n^2)$	×	√
快速排序	$O(n\log n)$	×	√
归并排序	$O(n\log n)$	√	×
计数排序	$O(n+k)$ k是数据范围	√	×
桶排序	$O(n)$	√	×
基数排序	$O(dn)$ d是维度	√	×

2）如何分析一个排序算法

a.算法的执行效率

- 最好情况、最坏情况、平均情况（以及其需要排序的原始数据是什么样子的）
- 时间复杂度的系数、常数、低阶
- 比较次数和交换（或移动的次数）

b.算法的内存消耗

- 原地排序算法：指空间复杂度是O(1)的排序算法

c.排序算法的稳定性

- 稳定性。这个概念是说，如果待排序的序列中存在值相等的元素，经过排序之后，相等元素之间原有的先后顺序不变。

3）电商系统中，订单时间和金额的排序问题

背景：我们现在要给电商交易系统中的“订单”排序。订单有两个属性，一个是下单时间，另一个是订单金额。如果我们现在有 10 万条订单数据，我们希望按照金额从小到大对订单数据排序。对于金额相同的订单，我们希望按照下单时间从早到晚有序。对于这样一个排序需求，我们怎么来做呢?

答：我们先按照下单时间给订单排序，注意是按照下单时间，不是金额。排序完成之后，我们用稳定排序算法，按照订单金额重新排序。两遍排序之后，我们得到的订单数据就是按照金额从小到大排序，金额相同的订单按照下单时间从早到晚排序的。

4）有序度和逆序度的概念

有序度是数组中具有有序关系的元素对的个数。有序元素对用数学表达式表示：a[i] <= a[j], 如果i < j。

满有序度，完全有序的数组的有序度。

逆序度 = 满有序度 - 有序度

例子1：对于一个倒序排列的数组，比如 6，5，4，3，2，1，有序度是 0；对于一个完全有序的数组，比如 1，2，3，4，5，6，有序度就是 n*(n-1)/2，也就是 15。

例子2：冒泡排序包含两个操作原子，比较和交换。每交换一次，有序度就加 1。不管算法怎么改进，交换次数总是确定的，即为逆序度，也就是n*(n-1)/2–初始有序度。

5）插入排序和冒泡排序的时间复杂度相同，都是 O(n2)，在实际的软件开发里，为什么我们更倾向于使用插入排序算法而不是冒泡排序算法呢?

冒泡排序不管怎么优化，元素交换的次数是一个固定值，是原始数据的逆序度。插入排序是同样的，不管怎么优化，元素移动的次数也等于原始数据的逆序度。但是，从代码实现上来看，冒泡排序的数据交换要比插入排序的数据移动要复杂，冒泡排序需要 3 个赋值操作，而插入排序只需要 1 个。我们来看这段操作：

```
1
```

```
//冒泡排序中数据的交换操作：
if (a[j] > a[j+1]) { // 交换
   int tmp = a[j];
   a[j] = a[j+1];
   a[j+1] = tmp;
   flag = true;
}

插入排序中数据的移动操作：
if (a[j] > value) {
  a[j+1] = a[j];  // 数据移动
} else {
  break;
}
```

我们把执行一个赋值语句的时间粗略地计为单位时间（unit_time），然后分别用冒泡排序和插入排序对同一个逆序度是 K 的数组进行排序。用冒泡排序，需要 K 次交换操作，每次需要 3 个赋值语句，所以交换操作总耗时就是 3*K 单位时间。而插入排序中数据移动操作只需要 K 个单位时间。

6）如何用有限的内存对10个文件进行排序

问题：现在你有 10 个接口访问日志文件，每个日志文件大小约 300MB，每个文件里的日志都是按照时间戳从小到大排序的。你希望将这 10 个较小的日志文件，合并为 1 个日志文件，合并之后的日志仍然按照时间戳从小到大排列。如果处理上述排序任务的机器内存只有 1GB，你有什么好的解决思路，能“快速”地将这 10 个日志文件合并吗？

参考1：先取得十个文件时间戳的最小值数组的最小值a，和最大值数组的最大值b。然后取mid=(a+b)/2，然后把每个文件按照mid分割，取所有前面部分之和，如果小于1g就可以读入内存快排生成中间文件，否则继续取时间戳的中间值分割文件，直到区间内文件之和小于1g。同理对所有区间都做同样处理。最终把生成的中间文件按照分割的时间区间的次序直接连起来即可。

参考2：参考1最大好处是充分利用了内存。

但是我还是会这么做：

1.申请10个40M的数组和一个400M的数组。

2.每个文件都读40M，取各数组中最大时间戳中的最小值。

3.然后利用二分查找，在其他数组中快速定位到小于/等于该时间戳的位置，并做标记。

4.再把各数组中标记位置之前的数据全部放在申请的400M内存中，

5.在原来的40M数组中清除已参加排序的数据。[可优化成不挪动数据，只是用两个索引标记有效数据的起始和截止位置]

6.对400M内存中的有效数据[没装满]做快排。

将排好序的直接写文件。

7.再把每个数组尽量填充满。从第2步开始继续，知道各个文件都读区完毕。

这么做的好处有：

1.每个文件的内容只读区一次，且是批量读区。比每次只取一条快得多。

2.充分利用了读区到内存中的数据。曹源 同学在文件中查找那个中间数是会比较困难的。

3.每个拷贝到400M大数组中参加快排的数据都被写到了文件中，这样每个数只参加了一次快排。

7）如何根据年龄给100万用户数据排序？

根据年龄给 100 万用户排序，就类似按照成绩给 50 万考生排序。我们假设年龄的范围最小 1 岁，最大不超过 120 岁。我们可以遍历这 100 万用户，根据年龄将其划分到这 120 个桶里，然后依次顺序遍历这 120 个桶中的元素。这样就得到了按照年龄排序的 100 万用户数据。

（桶排序：核心思想是将要排序的数据分到几个有序的桶里，每个桶里的数据再单独进行排序。桶内排完序之后，再把每个桶里的数据按照顺序依次取出，组成的序列就是有序的了。）

leetcode

https://leetcode-cn.com/problems/third-maximum-number/ 第三大的元素

https://leetcode-cn.com/problems/merge-sorted-array/ 合并两个有序数组

七、查找

1）二分查找应用的局限性

首先，二分查找依赖的是顺序表结构，简单点说就是数组。（二分查找依赖下标随机访问元素）

其次，二分查找针对的是有序数据。

再次，数据量太小不适合二分查找。

最后，数据量太大也不适合二分查找。

2）二分查找的经典变体

- 查找第一个值等于给定值的元素； 测试用例[1,3,4,5,6,8,8,8,11,18]
- 查找最后一个值等于给定值的元素;
- 查找第一个大于给定值的元素
- 查找最后一个小于等于给定值的元素

3）如何快速定位IP对应的省份地址？

背景：我们维护一个很大的 IP 地址库来实现的。地址库中包括 IP 地址范围和归属地的对应关系。

例子：当我们想要查询 202.102.133.13 这个 IP 地址的归属地时，我们就在地址库中搜索，发现这个 IP 地址落在[202.102.133.0, 202.102.133.255]这个地址范围内，那我们就可以将这个 IP 地址范围对应的归属地“山东东营市”显示给用户了。

```
[202.102.133.0, 202.102.133.255] 山东东营市
[202.102.135.0, 202.102.136.255] 山东烟台
[202.102.156.34, 202.102.157.255] 山东青岛
[202.102.48.0, 202.102.48.255] 江苏宿迁
[202.102.49.15, 202.102.51.251] 江苏泰州
[202.102.56.0, 202.102.56.255] 江苏连云港
```

问题：假设我们有 12 万条这样的 IP 区间与归属地的对应关系，如何快速定位出一个 IP 地址的归属地呢?

（在庞大的地址库中逐一比对 IP 地址所在的区间，是非常耗时的。）

答：

如果 IP 区间与归属地的对应关系不经常更新，我们可以先预处理这 12 万条数据，让其按照起始 IP 从小到大排序。如何来排序呢？我们知道，IP 地址可以转化为 32 位的整型数。所以，我们可以将起始地址，按照对应的整型值的大小关系，从小到大进行排序。

然后，这个问题就可以转化为我刚讲的第四种变形问题“在有序数组中，查找最后一个小于等于某个给定值的元素”了。当我们要查询某个 IP 归属地时，我们可以先通过二分查找，找到最后一个起始 IP 小于等于这个 IP 的 IP 区间，然后，检查这个 IP 是否在这个 IP 区间内，如果在，我们就取出对应的归属地显示；如果不在，就返回未查找到。

leetcode

https://leetcode-cn.com/problems/sqrtx/ x的平方根

进阶：求x的平方根且精确到小数点后6位

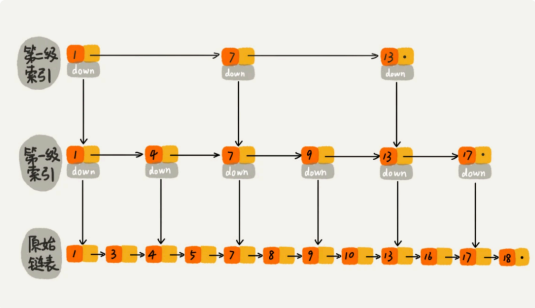
```
def sqrt(x):
    '''
    求平方根，精确到小数点后6位
    '''
    low = 0
    mid = x / 2
    high = x
    while abs(mid ** 2 - x) > 0.000001:
        if mid ** 2 < x:
            low = mid
        else:
            high = mid
        mid = (low + high) / 2
    return mid
```

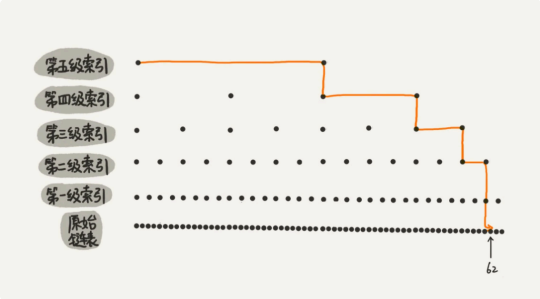
八、跳表

1）跳表是什么？

链表加多级索引的结构，就是跳表。

例子：从图中我们可以看出，原来没有索引的时候，查找 62 需要遍历 62 个结点，现在只需要遍历 11 个结点，速度提高了很多





2）跳表的时间复杂度与空间

时间复杂度为O(logn)、空间复杂度为O(n)。推理过程见原文。

3）为什么要Redis要用跳表来实现有序集合？（跳表vs红黑树）

Redis中的有序集合支持的核心操作主要有：插入一个数据；删除一个数据；查找一个数据；按照区间查找数据（比如查找值在[100, 356]之间的数据）；迭代输出有序序列。

- 按照区间查找更高效：插入、删除、查找以及迭代输出有序序列这几个操作，红黑树也可以完成，时间复杂度跟跳表是一样的。但是，按照区间来查找数据这个操作，红黑树的效率没有跳表高。对于按照区间查找数据这个操作，跳表可以做到 O(logn) 的时间复杂度定位区间的起点，然后在原始链表中顺序往后遍历就可以了。这样做非常高效。
- 跳表更容易代码实现
- 跳表更加灵活，它可以通过改变索引构建策略，有效平衡执行效率和内存消耗。

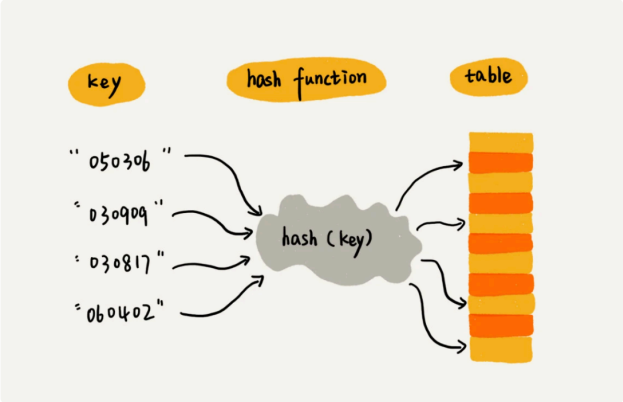
九、散列表

1)散列表是什么?

散列表用的就是数组支持按照下标随机访问的时候，时间复杂度是 O(1) 的特性。我们通过散列函数把元素的键值映射为下标，然后将数据存储在数组中对应下标的位置。当我们按照键值查询元素时，我们用同样的散列函数，将键值转化数组下标，从对应的数组下标的位置取数据。

例子：假如我们有 89 名选手参加学校运动会。为了方便记录成绩，每个选手胸前都会贴上自己的参赛号码。这 89 名选手的编号依次是 1 到 89。现在我们希望编程实现这样一个功能，通过编号快速找到对应的选手信息。你会怎么做呢？我们可以把这 89 名选手的信息放在数组里。编号为 1 的选手，我们放到数组中下标为 1 的位置；编号为 2 的选手，我们放到数组中下标为 2 的位置。以此类推，编号为 k 的选手放到数组中下标为 k 的位置。

参赛选手的编号我们叫做**键（key）**或者关键字。我们用它来标识一个选手。我们把参赛编号转化为数组下标的映射方法就叫作**散列函数（或“Hash 函数”“哈希函数”）**，而散列函数计算得到的值就叫作**散列值（或“Hash 值”“哈希值”）**。



2）Word 文档中单词拼写检查功能是如何实现的?

常用的英文单词有 20 万个左右，假设单词的平均长度是 10 个字母，平均一个单词占用 10 个字节的内存空间，那 20 万英文单词大约占 2MB 的存储空间，就算放大 10 倍也就是 20MB。对于现在的计算机来说，这个大小完全可以放在内存里面。所以我们可以用散列表来存储整个英文单词词典。当用户输入某个英文单词时，我们拿用户输入的单词去散列表中查找。如果查到，则说明拼写正确；如果没有查到，则说明拼写可能有误，给予提示。借助散列表这种数据结构，我们就可以轻松实现快速判断是否存在拼写错误。

3）假设我们有 10 万条 URL 访问日志，如何按照访问次数给 URL 排序?

遍历 10 万条数据，以 URL 为 key，访问次数为 value，存入散列表，同时记录下访问次数的最大值 K，时间复杂度 O(N)。

如果 K 不是很大，可以使用桶排序，时间复杂度 O(N)。如果 K 非常大（比如大于 10 万），就使用快速排序，复杂度 O(NlogN)。

4）有两个字符串数组，每个数组大约有 10 万条字符串，如何快速找出两个数组中相同的字符串?

以第一个字符串数组构建散列表，key 为字符串，value 为出现次数。再遍历第二个字符串数组，以字符串为 key 在散列表中查找，如果 value 大于零，说明存在相同字符串。时间复杂度 O(N)。

leetcode

https://leetcode-cn.com/problems/design-hashset/submissions/705. 设计哈希集合

十、哈希算法

1）什么是哈希算法?

将任意长度的二进制值串映射为固定长度的二进制值串，这个映射的规则就是哈希算法。而通过原始数据映射之后得到的二进制值串就是哈希值。（常用哈希算法MD5，SHA）

2）哈希算法要满足的几点要求

从哈希值不能反向推导出原始数据（所以哈希算法也叫单向哈希算法）；

对输入数据非常敏感，哪怕原始数据只修改了一个 Bit，最后得到的哈希值也大不相同；

散列冲突的概率要很小，对于不同的原始数据，哈希值相同的概率非常小；

哈希算法的执行效率要尽量高效，针对较长的文本，也能快速地计算出哈希值。

3）哈希算法的7个常见应用

安全加密、唯一标识、数据校验、散列函数、负载均衡、数据分片、分布式存储。

4）为什么哈希算法可以用于安全加密：

第一点是很难根据哈希值反向推导出原始数据，第二点是散列冲突的概率要很小。

5）唯一标识：用哈希算法搜索一张图是否在图库中存在

如果要在海量的图库中，搜索一张图是否存在，我们不能单纯地用图片的元信息（比如图片名称）来比对，因为有可能存在名称相同但图片内容不同，或者名称不同图片内容相同的情况。那我们该如何搜索呢?

我们可以给每一个图片取一个唯一标识，或者说信息摘要。比如，我们可以从图片的二进制码串开头取 100 个字节，从中间取 100 个字节，从最后再取 100 个字节，然后将这 300 个字节放到一块，通过哈希算法（比如 MD5），得到一个哈希字符串，用它作为图片的唯一标识。通过这个唯一标识来判定图片是否在图库中，这样就可以减少很多工作量。

6）数据校验：如何判断文件下载过程中是否被篡改?

背景：我们知道，BT 下载的原理是基于 P2P 协议的。我们从多个机器上并行下载一个 2GB 的电影，这个电影文件可能会被分割成很多文件块（比如可以分成 100 块，每块大约 20MB）。等所有的文件块都下载完成之后，再组装成一个完整的电影文件就行了。如何校验文件下载中是否被宿主机器恶意修改过，又或者下载过程中出现了错误?

哈希算法有一个特点，对数据很敏感。只要文件块的内容有一丁点儿的改变，最后计算出的哈希值就会完全不同。所以，当文件下载

完成之后，我们可以通过相同的哈希算法，对下载好的文件求哈希值，然后跟种子文件中保存的哈希值比对。如果不同，说明这个文件块不完整或者被篡改了，需要再重新从其他宿主机器上下载这个文件块。

7）如何防止密码被拖库?

可以通过哈希算法，对用户密码进行加密之后再存储，不过最好选择相对安全的加密算法，比如 SHA 等（因为 MD5 已经号称被破解了）。我们可以引入一个盐（salt），跟用户的密码组合在一起，增加密码的复杂度。我们拿组合之后的字符串来做哈希算法加密，将它存储到数据库中，进一步增加破解的难度。

8）区块链使用的是哪种哈希算法？是为了解决什么问题而使用的呢?

区块链是一块块区块组成的，每个区块分为两部分：区块头和区块体。

区块头保存着 自己区块体 和 上一个区块头 的哈希值。

因为这种链式关系和哈希值的唯一性，只要区块链上任意一个区块被修改过，后面所有区块保存的哈希值就不对了。

区块链使用的是 SHA256 哈希算法，计算哈希值非常耗时，如果要篡改一个区块，就必须重新计算该区块后面所有的区块的哈希值，短时间内几乎不可能做到。

9）如何才能实现一个会话粘滞（session sticky）的负载均衡算法呢？也就是说，我们需要在同一个客户端上，在一次会话中的所有请求都路由到同一个服务器上。

可以通过哈希算法，对客户端 IP 地址或者会话 ID 计算哈希值，将取得的哈希值与服务器列表的大小进行取模运算，最终得到的值就是应该被路由到的服务器编号。

10）数据分片：如何统计“搜索关键词”出现的次数?

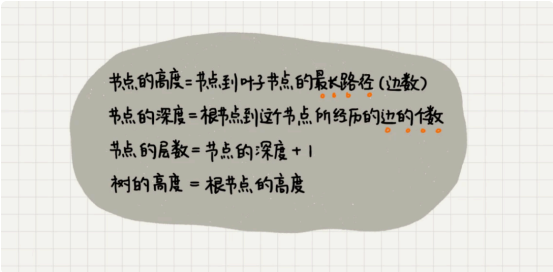
背景：假如我们有 1T 的日志文件，这里面记录了用户的搜索关键词，我们想要快速统计出每个关键词被搜索的次数，该怎么做呢?

我们来分析一下。这个问题有两个难点，第一个是搜索日志很大，没办法放到一台机器的内存中。第二个难点是，如果只用一台机器来处理这么巨大的数据，处理时间会很长。针对这两个难点，我们可以先对数据进行分片，然后采用多台机器处理的方法，来提高处理速度。具体的思路是这样的：为了提高处理的速度，我们用 n 台机器并行处理。我们从搜索记录的日志文件中，依次读出每个搜索关键词，并且通过哈希函数计算哈希值，然后再跟 n 取模，最终得到的值，就是应该被分配到的机器编号。这样，哈希值相同的搜索关键词就被分配到了同一个机器上。也就是说，同一个搜索关键词会被分配到同一个机器上。每个机器会分别计算关键词出现的次数，最后合并起来就是最终的结果。

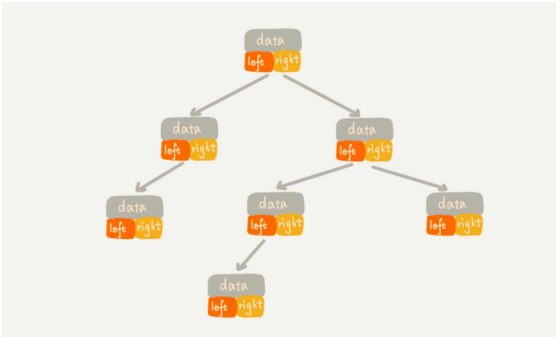
十一：二叉树

1）二叉树基础知识

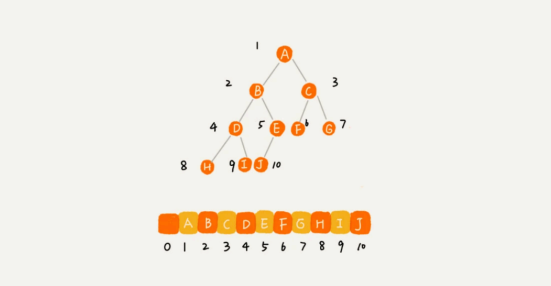
- 二叉树的高度（Height）、深度（Depth）、层（Level）。



- 满二叉树：叶子节点全都在最底层，除了叶子节点之外，每个节点都有左右两个子节点
- 完全二叉树：叶子节点都在最底下两层，最后一层的叶子节点都靠左排列，并且除了最后一层，其他层的节点个数都要达到最大
- 链式存储法：链式存储法。从图中你应该可以很清楚地看到，每个节点有三个字段，其中一个存储数据，另外两个是指向左右子节点的指针

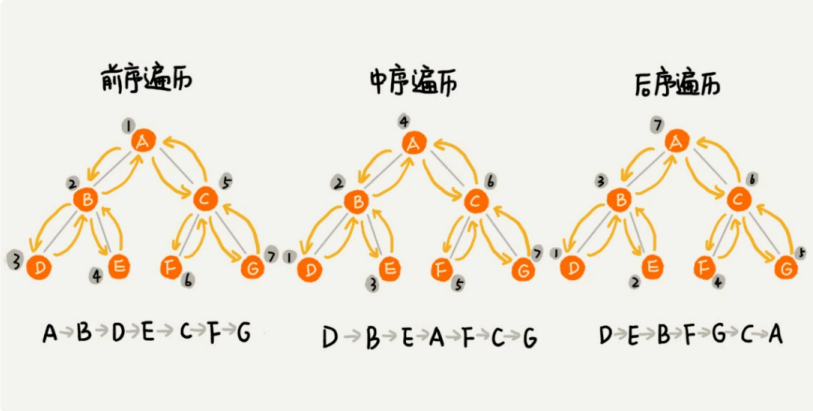


- 顺序存储法：我们把根节点存储在下标 i = 1 的位置，那左子节点存储在下标 2 * i = 2 的位置，右子节点存储在 2 * i + 1 = 3 的位置。**(适合存储完全二叉树。**



2）二叉树的遍历：前序中序后序

- 前序遍历，对于树中的任意节点来说，先打印这个节点，然后再打印它的左子树，最后打印它的右子树。
- 中序遍历，对于树中的任意节点来说，先打印它的左子树，然后再打印它本身，最后打印它的右子树。
- 后序遍历，对于树中的任意节点来说，先打印它的左子树，然后再打印它的右子树，最后打印这个节点本身。



3）二叉查找树

定义:二叉查找树中，每个节点的值都大于左子树节点的值，小于右子树节点的值。

特性:只需要中序遍历，就可以在 O(n) 的时间复杂度内，输出有序的数据序列。

4）二叉树的查找、插入、删除操作

代码略

5）二叉查找树和散列表的比较

散列表的优势：时间负责度低。散列表的插入、删除、查找操作的时间复杂度可以做到常量级的 O(1)，非常高效。而二叉查找树在比较平衡的情况下，插入、删除、查找操作时间复杂度才是 O(logn)。

二叉树的优势：

1. 散列表中的数据是无序存储的，如果要输出有序的数据，需要先进行排序。而对于二叉查找树来说，我们只需要中序遍历，就可以在 O(n) 的时间复杂度内，输出有序的数据序列。
2. 散列表扩容耗时很多，而且当遇到散列冲突时，性能不稳定，尽管二叉查找树的性能不稳定，但是在工程中，我们最常用的平衡二叉查找树的性能非常稳定，时间复杂度稳定在 O(logn)。
3. 尽管散列表的查找等操作的时间复杂度是常量级的，但因为哈希冲突的存在，这个常量不一定比 logn 小，所以实际的查找速度可能不一定比 O(logn) 快。加上哈希函数的耗时，也不一定就比平衡二叉查找树的效率高。
4. 散列表的构造比二叉查找树要复杂，需要考虑的东西很多。比如散列函数的设计、冲突解决办法、扩容、缩容等。平衡二叉查找树只需要考虑平衡性这一个问题，而且这个问题的解决方案比较成熟、固定。
5. 为了避免过多的散列冲突，散列表装载因子不能太大，特别是基于开放寻址法解决冲突的散列表，不然会浪费一定的存储空间。

leetcode

树问题集合https://leetcode-cn.com/leetbook/read/data-structure-binary-tree/xefb4e/

（二叉树的前序、中序、后序遍历（递归 vs非递归方法）；二叉树的最大深度；对称二叉树；二叉树的路径总和）

十二、堆

十三、图

1）图基础

图中的元素我们就叫做顶点（vertex）；

顶点与顶点之间的关系就叫做边（edge）；

无向图中，顶点的度（degree）就是跟顶点相连接的边的条数。

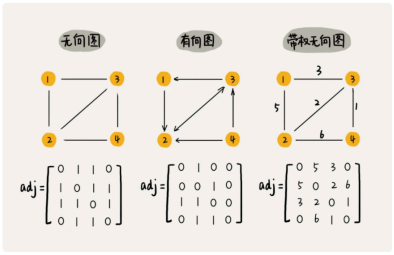
有向图中，我们把度分为入度（In-degree）和出度（Out-degree）（微博的例子，入度就表示有多少粉丝，出度就表示关注了多少人）

带权图（weighted graph）。在带权图中，每条边都有一个权重（weight），我们可以通过这个权重来表示 QQ 好友间的亲密度。

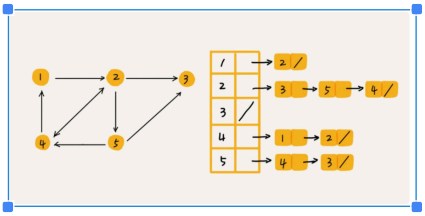
2）图的存储方式（邻接矩阵、邻接表）

- 邻接矩阵（Adjacency Matrix）。即一个二维数组，行与列代表节点，值代表边。

缺点：浪费空间；优点：能高效获取两个节点间的关系；方便计算，能将图的计算转化为矩阵运算。



- 邻接表（Adjacency List）每个顶点对应一条链表，链表中存储的是与这个顶点相连接的其他顶点
- 优点：节省空间，缺点：使用起来很耗时间。



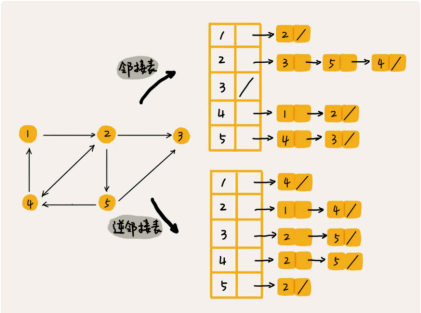
3）如何存储微博、微信等这些社交网络的好友关系

问题：微博、微信、LinkedIn 这些社交软件我想你肯定都玩过吧。在微博中，两个人可以互相关注；在微信中，两个人可以互加好友。那你知道，如何存储微博、微信等这些社交网络的好友关系吗？

需求：判断用户 A 是否关注了用户 B；判断用户 A 是否是用户 B 的粉丝；用户 A 关注用户 B；用户 A 取消关注用户 B；根据用户名称的首字母排序，分页获取用户的粉丝列表；根据用户名称的首字母排序，分页获取用户的关注列表。

回答：关于如何存储一个图，前面我们讲到两种主要的存储方法，邻接矩阵和邻接表。因为社交网络是一张稀疏图，使用邻接矩阵存储比较浪费存储空间。所以，这里我们采用邻接表来存储。邻接表中存储了用户的关注关系，逆邻接表中存储的是用户的被关注关系。

在数据库中的存储方式：一列userid，一列followid，为了提高效率，两列都建立了索引。



user_id	follower_id
1	4
2	1
2	4
3	2
3	5
4	2
4	5
5	2