Algorithm

数组

数组的基本概念

数组（Array）：数组是一种**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**相同数据类型**的数据。

- 线性表与非线性表
  - 线性表（Linear List）：数据排成像一条线一样的结构。每个线性表上的数据最多只有前和后2个方向。如：数组、链表、队列、栈等。
  - 非线性表：数据之间不是简单的前后关系。如：二叉树、堆、图等。
- 连续的内存空间和相同类型的数据
  - 正是因为这2个限制，数组具有一个杀手级的特性：**随机访问。**即其根据下标随机访问的时间复杂度为O(1)。但是要**注意**数组中查找一个元素的时间复杂度不是O(1)。
  - 但是也正是因为数组存储需要连续的内存空间，因此其插入和删除操作非常低效，因为为了保证数组数据的连续性，需要做大量的数据搬移工作。

数组的基本操作

- 插入：
  - 实现：将某个元素插入到数组第k个位置，需要将k~n部分的元素向后搬移一位，然后插入元素。若插入到数组的末尾，时间复杂度O(1)；插入到数组的开头，时间复杂度O(n)；插入的平均时间复杂度为（1+2+...+n）/n = O(n)。注意这个实现针对的数组是有序的。
  - **优化**：若数组只是一个存储数据的集合，其元素是无序的，则插入时不需要搬移数据。此时若想将某个元素插入到数组第k个位置，首先将该位置的元素移动到数组末尾，然后将待插入元素插入到第k个位置，时间复杂度降为O(1)。但是要注意这个优化是针对数组是无序的,这个优化其实是放弃了数组逻辑上的连续。
- 删除
  - 实现：若要删除第k个位置的元素，则需要将k+1~n个元素向前搬移一位。若删除数组的末尾，时间复杂度O(1)；删除数组的开头，时间复杂度O(n)；删除平均时间复杂度为 O(n)。
  - **优化**：将多次删除一起执行。当我们要删除多个元素的时候，并不立即进行搬移操作，而是将多个元素标记为已删除，然后当数组满了的时候，将标记的元素一起删除，这样就减少了数组搬移的次数，提高了效率。这就是**JVM中标记-清除**算法的核心思想。
- 随机访问
  - 实现：数组的随机访问，只需要根据首地址和下标通过寻址公式计算出对应内存地址即可。

数组的内存模型

- 访问数组的本质：计算机为每一个数组分配了一段连续的内存，计算机通过访问内存的地址访问内存，因此访问数组的本质就是访问一段连续内存。对内存的要求较高。
- 既然访问数组其实是访问一段连续内存，那么计算机就可以根据数组的首地址和下标，通过公式计算出每一个元素所在的位置。数组访问寻址公式：
  - 给定一维数组 `a[n]`，则 `a[k]` 的内存地址为：`a[k]_address = base_address + k * type_size`。
  - 给定二维数组 `a[m][n]` 则 `a[i][j]` 的内存地址为：`a[i][j]_address = base_address + (i*n + j) * type_size`。

链表

链表的基本概念

链表（Linked List）：链表是一种**线性表**数据结构。它用一组**不连续的内存空间**，来存储一组具有**相同数据类型**的数据。

链表的基本操作

- 插入：链表的插入只需要调整前后节点的指针即可,时间复杂度为O(1)
- 删除：链表的删除只需要调整前后节点的指针即可,时间复杂度为O(1)
- 随机访问：链表的随机访问，无法像数组一样通过寻址公式计算出下标的内存位置，只能通过遍历找到相应的节点，时间复杂度为O(n)。

链表的分类

1. 单链表

结构：head -> data -> data -> null

特点：结构简单，但是随机访问某个节点时，只能从头到尾遍历，找到这个节点。

2. 循环链表

结构：head -> data -> data -> head

特点：与单链表相比，其优势在于可以从尾到头遍历。

3. 双向链表

结构：head <-> data <-> data -> null

特点：双向链表最大的优势在于找到前驱结点的时间复杂度为O(1)，正因为这个特点，使得双向链表在某些情况下，插入、删除等操作要比单链表简单高效，尽管时间复杂度都是O(1)。这也是为什么双向链表占用的内存要高于单链表，但是日常使用中，双向链表更加常用的原因。**Java中LinkedHashMap就用双向链表来记录插入的键值对的顺序。**

4. 双向循环链表

双向循环链表是双向链表与循环链表的组合。

结构：head <-> data <-> data -> head

问题：为什么双向链表比单链表高效?

插入和删除

（1）删除操作

在实际开发中，从链表删除一个数据主要是以下2种情况：

- 删除某个“值等于给定值”的节点
- 删除给定指针指向的节点

事实上，删除操作本身的时间复杂度为O(1),单链表和双向链表的性能差别，主要在于查找的过程。

对于第一种情况，无论单链表还是双向链表，无论是插入还是删除，都需要从头到尾遍历找到“值等于给定值”的节点，因此二者的时间复杂度均为O(n)。

对于第二种情况，已经找到了要删除的节点（记为q），但是要删除这个节点需要知道其前驱节点(记为p)。单链表中，无法立即知道q的前驱节点p,因此只能从头遍历，直到`p->next = q`，这个过程的时间复杂度是O(n); 而双向链表要删除的节点q已经保存了前驱节点p的指针。因此可以O(1)的时间找到p。因此单链表删除操作时间复杂度为O(n)，而双向链表为O(1)。

（2）插入操作

分析同上。

查找

对于一个有序链表，双向链表按值查找的效率要高于单链表。因为，我们可以记录上一次查找的位置p，每次查找时，比较要查找的值与p的关系，若小于p，则向前，反之则向后，平均只需要查找一半的数据。

因此，双向链表在删除和插入给定节点以及在有序链表中查找值这几种情形下，效率要高于单链表。

事实上，这里的双向链表的性能来自于其对空间的牺牲，它的空间占用要高于单链表，体现了 **“空间换时间”** 的设计思想。

####

数组 VS 链表

1. 内存

从内存来讲，数组要求内存是一块连续的内存空间，而链表则不需要内存空间连续，例如，如果申请一个100MB大小的数组，当内存中没有连续的大于等于100MB的内存空间时，即使内存的总空间大于100MB，内存也会申请失败；而链表则不需要连续的内存空间，链表通过指针将不同的内存块串联起来，如果内存的总空间大于100MB，那么申请100MB的链表则没有任何问题。

尽管数组需要连续的内存空间，对内存的要求较高，但是正因为这个特性，数组可以借助CPU缓存机制，预读数组中的数据，因此访问效率更高。而链表由于内存不连续，因此对CPU缓存不友好，无法预读。

若代码对内存使用比较苛刻，那么使用数组会比较好。因为链表的每个节点都需要使用额外的内存存储指向下一个节点的指针，额外内存高于数组。

链表在进行插入和删除操作的时候，导致内存频繁的申请和释放，容易造成内存碎片。在Java中，有可能到导致频繁的GC。

2. 基本操作

时间复杂度	数组	链表
插入、删除	O(n)	O(n)
随机访问	O(1)	O(n)

根据二者的时间复杂度，在随机访问更加频繁的场景下（插入、删除操作非常少），数组更加适用；在插入、删除操作频繁的场景下，链表更加适用。

栈

栈的基本概念

栈（stack）：栈是一种“操作受限”的**线性表**数据结构。栈元素具有后进先出，先进后出的特点。

栈的存储

顺序栈：用数组实现的栈。
链式栈：用链表实现的栈。

栈的基本操作

栈具有2个基本操作：栈的插入和删除，而且只能在一端进行插入和删除操作。

栈的插入和删除操作的时间复杂度为O(1)。

示例：一摞叠在一起的盘子能很好地说明栈。放盘子的时候，都是从下往下一个个放；取盘子的时候，则是从上往下一个个取，不能从中间任意抽出。只涉及到了线性表一端的插入和删除。

栈的应用

当某个数据集合只涉及在一端插入和删除数据，并且满足后进先出、先进后出的特性，那么就应该首选栈作为其数据结构。 例如，函数调用就是依托栈实现的。

队列

队列的基本概念

队列（Queue）:队列是一种“操作受限”的线性表数据结构。队列元素具有先进先出，后进后出的特点。

队列的存储

顺序队列：用数组实现的队列。
链式队列：用链表实现的队列。

队列的基本操作

队列具有2个基本操作：入队和出队。

队列的入队和出队的时间复杂度均为O(1)。

示例：排队买票，先来的先买，后来的只能排到队尾，不能插队。

队列的应用

阻塞队列是在队列的基础上增加了阻塞操作。当队列为空的时候，从队头取数据会被阻塞，直到队列中存在数据的时候，才返回数据；当队列满了的时候，插入数据操作会被阻塞，直到队列中有空闲位置时再插入数据然后再返回。我们可以使用阻塞队列实现**生产者-消费者模型**。

在多线程的情形下，就会存在安全问题，这就需要并发队列了。

并发队列是指线程安全的队列。并发队列最简单的实现就是在 `enqueue()` 和 `dequeue()` 方法上加锁，但是锁的粒度比较大，因此并发度不是很好，同一时刻仅仅允许一个操作。但是，基于数组的循环队列，利用CAS原子操作，可以实现非常高效的并发队列。这也是循环队列比链式队列应用更加广泛的原因。

如何实现无锁并发队列?

使用数组+CAS的方式可以实现。在入队前，获取tail的位置，入队时，比较tail是否发生变化，若没有发生变化，则允许入队，反之入队失败。出队操作分析同上。

对于大部分资源有限的场景，当没有空闲资源时，基本上都可以通过队列实现请求排队。 例如线程池，分布式应用中的消息队列（如kafka）等。

问题：当线程池没有空闲线程时，新的任务请求线程资源时，线程池该如何处理?

一般有2种策略：

- 非阻塞的处理方式，直接拒绝任务请求
- 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。

那么如何存储排队的请求呢？一般希望公平的处理每个请求，因此按照先来的请求先处理，那么就适合用队列来存储请求。那么队列该如何实现呢？是基于数组还是基于链表呢?

- 基于链表的实现方式，可以实现一个可以支持无限排队的**无界队列**（unbounded queue），但是可能导致多多的请求等待，请求的响应时间过长。因此，针对响应时间比较敏感的系统，基于链表实现的无界队列的线程池是不合适的。
- 基于数组的实现方式，可以实现一个**有界队列**（bounded queue），队列大小有限，因此当请求数量超过队列大小时，接下来的请求就只能拒绝处理。这种方式适用于响应时间比较敏感的系统。队列的大小的设计就至关重要了。太大会导致等待请求太多，太小则会造成无法充分利用系统资源，发挥最大性能。

二分查找

二分查找

- 二分查找：高效的有序数据集合查找算法
  - 时间复杂度：O(logn)
  - O(logn)是一个快到很恐怖复杂度。例如：当$n=2^{32}$（约42亿）时，logn=32。即从42亿有序数据中二分查找一个数据，最多32次。相反，$O(2^n)$是一个慢的恐怖的时间复杂度。
- 应用场景：
  - 针对数组，原因是：
    - 数组：根据下标随机访问的时间复杂度O(1)
    - 链表：根据下标随机访问的时间复杂度O(n)
  - 针对有序数据

        - 二分查找只能用在插入、删除操作不频繁的静态数据中，一次排序多次查找的场景中，因为排序最快O(nlogn)
        - 那么动态数据集合如何快速查找某个数据呢？答案是二叉树。
    - 数据量太小不适合二分查找
        - 数据量太小，直接顺序遍历即可，二分查找的优势并不明显
        - 但是如果数据之间的比较操作比较耗时间，那么减少比较操作会大大提高性能，因此即使数据量小，但是二分查找可以减小比较次数，例如数组中存储的是长度超过300的字符串，这样长度的字符串的比较则会非常耗时间。
    - 数据量太大不适合二分查找
        - 二分查找需要依赖数组，而数组为了支持随机访问的特性，要求内存空间是连续的，因此对内存的要求比较苛刻。比如，要存储1GB的数据，则需要1GB的连续内存空间。
        - 这里的**连续**是指：即使有2GB内存空间剩余，但是这些空间是零散的，没有连续的1GB的空间，那照样无法申请一个1GB的数组。

问题

假设我们有 1000 万个整数数据，每个数据占 8 个字节，如何设计数据结构和算法，快速判断某个整数是否出现在这 1000 万数据中？ 我们希望这个功能不要占用太多的内存空间，最多不要超过 100MB，你会怎么做呢？

- 1000 * 10^4 * 8 / 10^6 = 80MB，可以使用数组存储80MB的整数，然后排序，使用二分查找取得某个数据。
- 大部分情况下，使用二分查找可以解决的问题，散列表和二叉树也可以解决，那么是否可以用散列表或者二叉树呢？答案是否定的。因为散列表和二叉树都需要额外的存储，100MB必然不够。

排序

排序算法	最好	最坏	平均	额外空间复杂度	稳定性	原地排序
冒泡排序	O(n)	O(n^2)	O(n^2)	O(1)	是	是
插入排序	O(n)	O(n^2)	O(n^2)	O(1)	是	是
选择排序	O(n^2)	O(n^2)	O(n^2)	O(1)	否	是
归并排序	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	是	否
快速排序	O(nlogn)	O(n^2)	O(nlogn)	O(1)	否	是
堆排序	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	否	是
桶排序	O(n)	-	-	O(n)	是	否
计数排序	O(n+k)(k是数据范围)	-	-	O(k)	是	否
基数排序	O(n*d)(d是维度)	-	-	O(n*d)	是	否

递归

什么是递归?

如何实现递归?

递归存在的问题是什么?

递归的应用

设计思想

空间换时间

当内存空间较为充足的时候，如果需要追求代码的执行速度，那么我们就可以选择空间复杂度相对较高，时间复杂度相对较低的数据结构或者算法。

时间换空间

当内存比较紧缺的时候，例如代码执行在手机或单片机上的时候，那么我们就需要选择空间复杂度相对较低，空间复杂度相对较低的数据结构或者算法。

事实上，我们不可能既保证时间的效率又保证空间的效率，我们只能尽力实现二者的平衡，根据实际权衡时间和空间。

缓存

缓存是一种提高数据读取性能的技术，在软件开发与硬件设计中应用非常广泛。

缓存的设计思想

缓存利用了**空间换时间**的设计思想。

如果我们把数据存储在硬盘上，会比较节省内存，但是查找数据的时候，则需要从硬盘把数据读入，速度非常慢。通过缓存技术，事先将数据从硬盘加载到了内存中，这样查询数据的时候，就可以从内存中直接读取，因为内存中读取数据的速度远远快于硬盘中读取数据的速度，因此速度非常快，但是此时内存空间就变少了，相当于利用空间换时间。

缓存的特征

命中率：当某个请求通过访问缓存而得到响应时，缓存命中。缓存的命中率越高，则缓存的利用率也就越高。

最大空间：缓存通常位于内存中，内存速度虽然要快于硬盘，但是内存的空间要远远小于硬盘，因此缓存的最大空间不可能非常大。

淘汰策略：缓存的大小有限，当缓存存放的数据量大于最大空间时，就涉及到数据的淘汰问题。常见的缓存淘汰策略有：

- 最少使用策略LFU(Least Frequently Used)：优先淘汰最少使用的数据。
- 先进先出策略FIFO(First In, First Out)：在实时性要求的场景下，需要经常访问最新的数据，就可以使用FIFO，使得最先进入的数据被淘汰。
- 最近最少使用策略LRU(Least Recently Used)：优先淘汰最久未使用的数据，也就是上次被访问时间距离现在最久的数据。LRU可以保证内存中的数据都是热点数据，即经常被访问的数据，从而保证缓存的命中率。