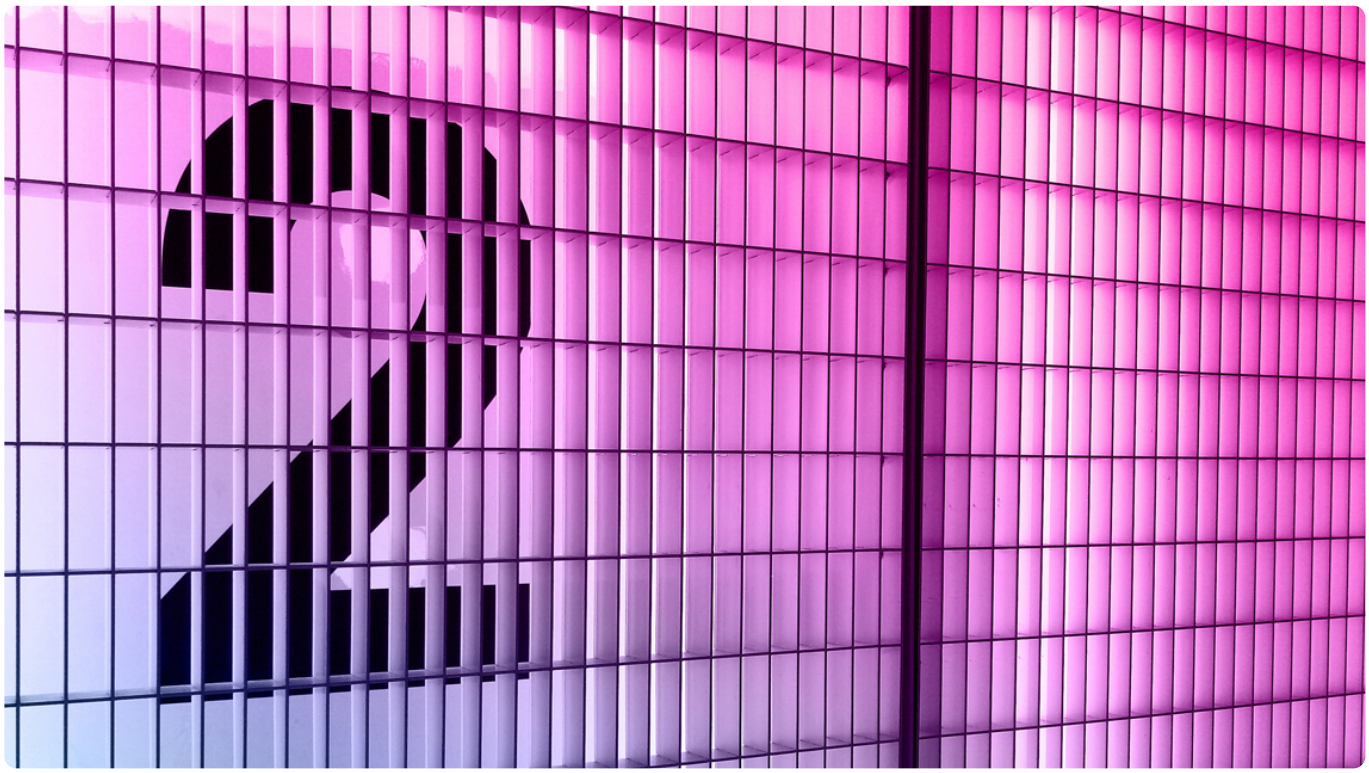
加微信：642945106 发送“ 赠送 ”领取赠送精品课程
发数字“ 2 ”获取众筹列表

02 | 非线性结构检索：数据频繁变化的情况下，如何高效检索？

2020-03-25 陈东

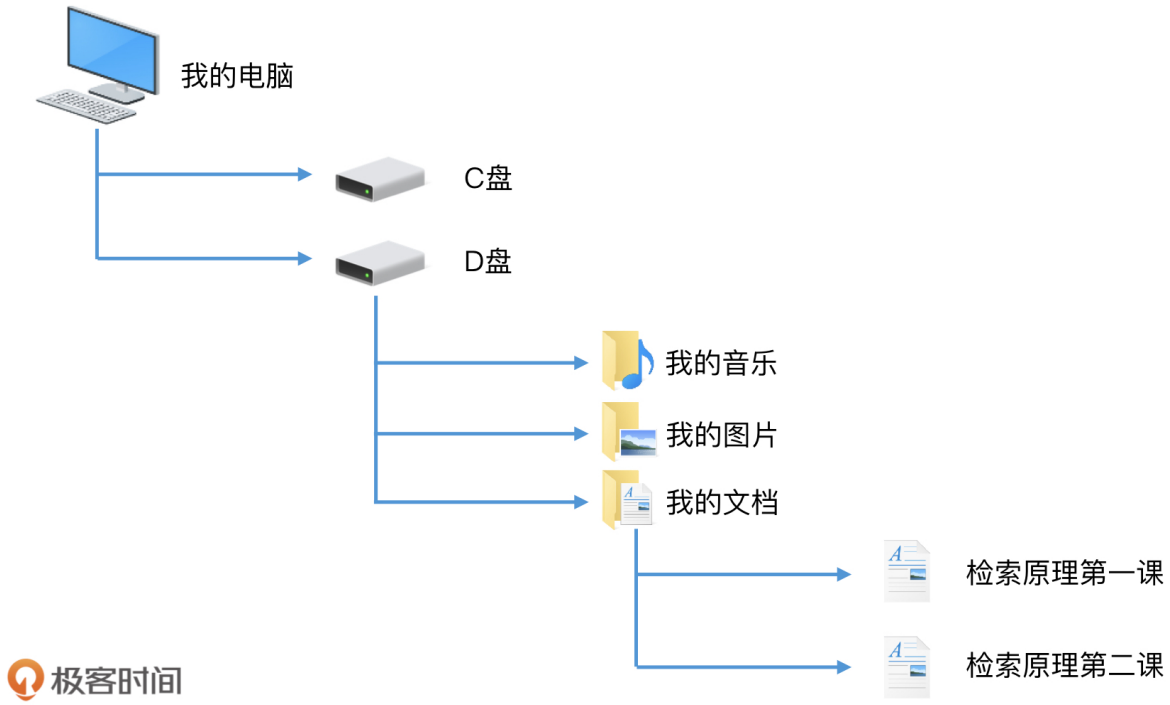
检索技术核心20讲 进入课程 >



讲述：陈东
时长 18:55　大小 17.33M

你好，我是陈东。

当我们在电脑中查找文件的时候，我们一般习惯先打开相应的磁盘，再打开文件夹以及子文件夹，最后找到我们需要的文件。这其实就是一个检索路径。如果把所有的文件展开，这个查找路径其实是一个树状结构，也就是一个非线性结构，而不是一个所有文件平铺排列的线性结构。



树状结构：文件组织例子

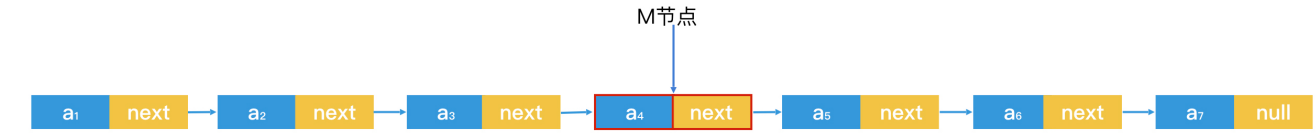
我们都知道，有层次的文件组织肯定比散乱平铺的文件更容易找到。这样熟悉的一个场景，是不是会给你一个启发：对于零散的数据，非线性的树状结构是否可以帮我们提高检索效率呢？

另一方面，我们也知道，在数据频繁更新的场景中，连续存储的有序数组并不是最合适的存储方案。因为数组为了保持有序必须不停地重建和排序，系统检索性能就会急剧下降。但是，非连续存储的有序链表倒是具有高效插入新数据的能力。因此，我们能否结合上面的例子，使用非线性的树状结构来改造有序链表，让链表也具有二分查找的能力呢？今天，我们就来讨论一下这个问题。

树结构是如何进行二分查找的？

上一讲我们讲了，因为链表并不具备“随机访问”的特点，所以二分查找无法生效。当链表想要访问中间的元素时，我们必须从链表头开始，沿着指针一步一步遍历，需要遍历一半的节点才能到达中间节点，时间代价是 O(n/2)。而有序数组由于可以“随机访问”，因此只需要 O(1) 的时间代价就可以访问到中间节点了。

那如果我们能在链表中以 O(1) 的时间代价快速访问到中间节点，是不是就可以和有序数组一样使用二分查找了？你先想想看该怎么做，然后我们一起来试着改造一下。



极客时间

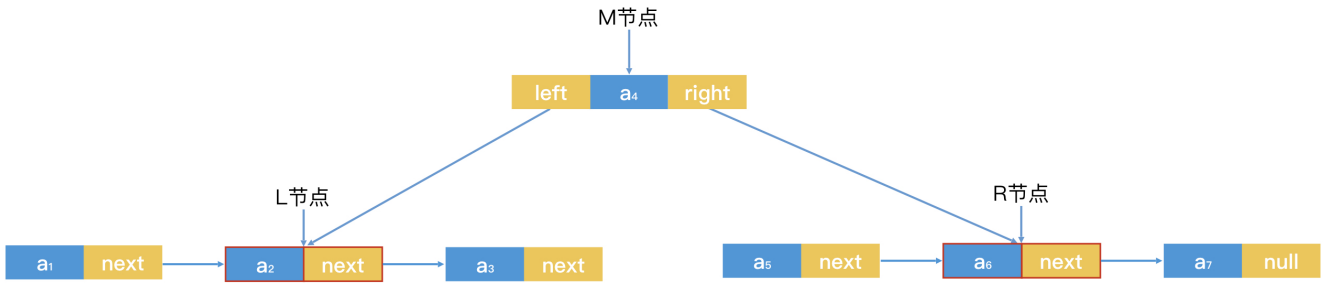
直接记录和访问中间节点

既然我们希望能以 O(1) 的时间代价访问中间节点，那将这个节点直接记录下来是不是就可以了？因此，如果我们把中间节点 M 拎出来单独记录，那我们的第一步操作就是直接访问这个中间节点，然后判断这个节点和要查找的元素是否相等。如果相等，则返回查询结果。如果节点元素大于要查找的元素，那我们就到左边的部分继续查找；反之，则在右边部分继续查找。

对于左边或者右边的部分，我们可以将它们视为两个独立的子链表，依然沿用这个逻辑。如果想用 O(1) 的时间代价就能访问这两个子链表的中间节点，我们就应该把左边的中间节点 L 和右边的中间节点 R，单独拎出来记录。

并且，由于我们是在访问完了 M 节点以后，才决定接下来该去访问左边的 L 还是右边的 R。因此，我们需要将 L 和 M，R 和 M 连接起来。我们可以让 M 带有两个指针，一个左指针指向 L，一个右指针指向 R。这样，在访问 M 以后，一旦发现 M 不是我们要查找的节点，那么，我们接下来就可以通过指针快速访问到 L 或者 R 了。

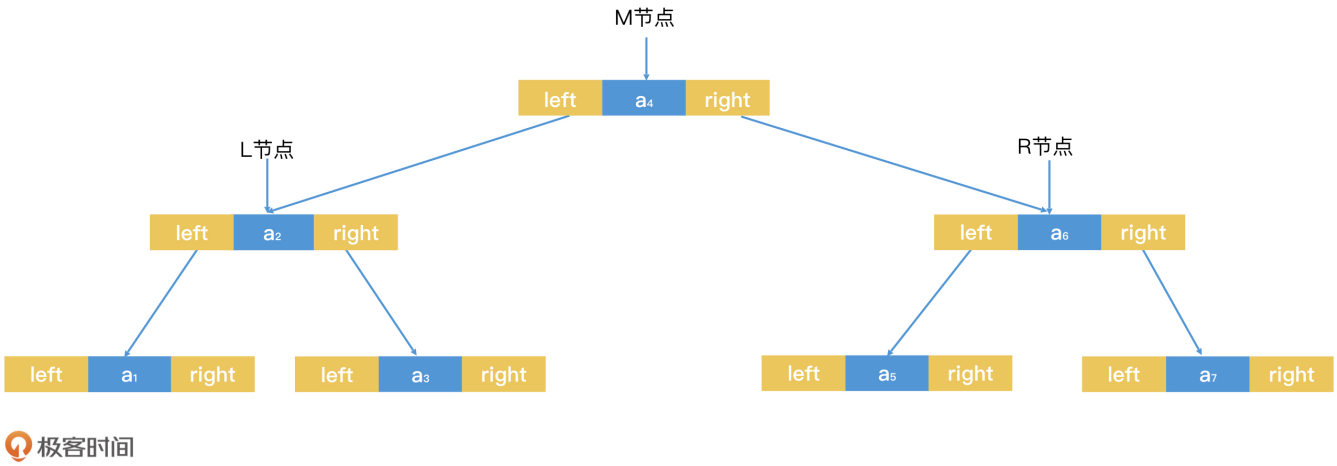
防止断更 请务必加
首发微信：17161436655



极客时间

将 M 节点改为带两个指针，指向 L 节点和 R 节点

对于其余的节点，我们也可以进行同样的处理。下面这个结构，你是不是很熟悉？没错，这就是我们常见的二叉树。你可以再观察一下，这个二叉树和普通的二叉树有什么不一样？



二叉检索树结构

没错，这个二叉树是有序的。它的左子树的所有节点的值都小于根节点，同时右子树所有节点的值都大于等于根节点。这样的有序结构，使得它能使用二分查找算法，快速地过滤掉一半的数据。具备了这样特点的二叉树，就是二叉检索树（Binary Search Tree），或者叫二叉排序树（Binary Sorted Tree）。

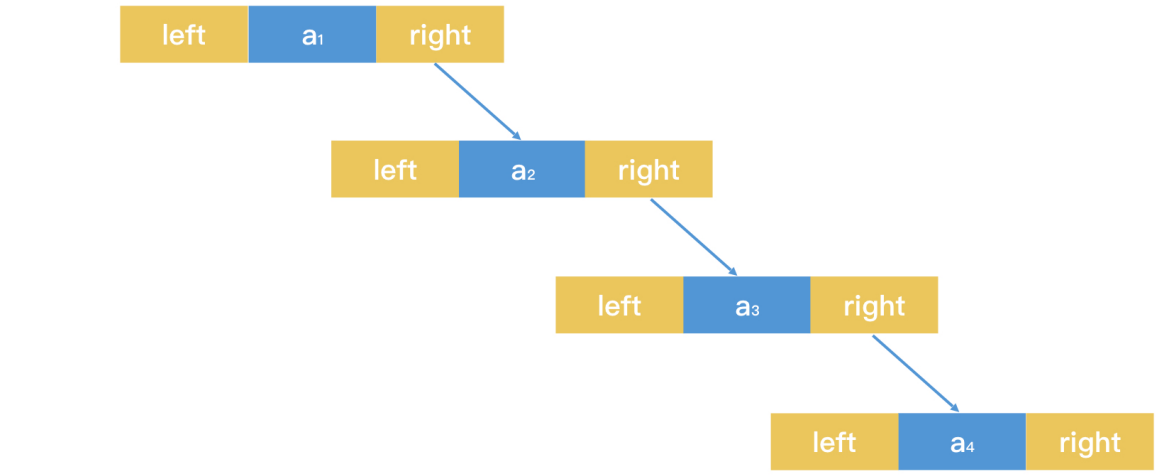
讲到这里，不知道你有没有发现，**尽管有序数组和二叉检索树，在数据结构形态上看起来差异很大，但是在提高检索效率上，它们的核心原理都是一致的。**那么，它们是如何提高检索效率的呢？核心原理又一致在哪里呢？接下来，我们就从两个主要方面来看。

> 将数据有序化，并且根据数据存储的特点进行不同的组织。对于连续存储空间的数组而言，由于它具有“随机访问”的特性，因此直接存储即可；对于非连续存储空间的有序链表而言，由于它不具备“随机访问”的特性，因此，需要将它改造为可以快速访问到中间节点的树状结构。
>
> 在进行检索的时候，它们都是通过二分查找的思想从中间节点开始查起。如果不命中，会快速缩小一半的查询空间。这样不停迭代的查询方式，让检索的时间代价能达到 O(log n) 这个级别。

说到这里，你可能会问，二叉检索树的检索时间代价一定是 O(log n) 吗？其实不一定。

二叉检索树的检索空间平衡方案

我们先来看一个例子。假设，一个二叉树的每一个节点的左指针都是空的，右子树的值都大于根节点。那么它满足二叉检索树的特性，是一颗二叉检索树。但是，如果我们把左边的空指针忽略，你会发现它其实就是一个单链表！单链表的检索效率如何呢？其实是 O(n)，而不是 O(log n)。



极客时间

退化成链表的二叉检索树

为什么会出现这样的情况呢？

最根本的原因是，这样的结构造成了检索空间不平衡。在当前节点不满足查询条件的时候，它无法把“一半的数据”过滤掉，而是只能过滤掉当前检索的这个节点。因此无法达到“快速减小查询范围”的目的。

因此，为了提升检索效率，我们应该尽可能地保证二叉检索树的平衡性，让左右子树尽可能差距不要太大。这样无论我们是继续往左边还是右边检索，都可以过滤掉一半左右的数据。

也正是为了解决这个问题，有更多的数据结构被发明了出来。比如：AVL 树（平衡二叉树）和红黑树，其实它们本质上都是二叉检索树，但它们都在保证左右子树差距不要太大上

做了特殊的处理，保证了检索效率，让二叉检索树可以被广泛地使用。比如，我们常见的 C++ 中的 Set 和 Map 等数据结构，底层就是用红黑树实现的。

这里，我就不再详细介绍 AVL 树和红黑树的具体实现了。为了保证检索效率，我们其实只需要在数据的组织上考虑检索空间的平衡划分就好了，这一点都是一样的。

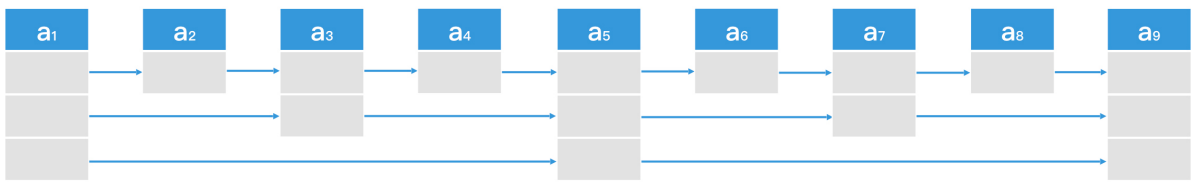
跳表是如何进行二分查找的？

除了二叉检索树，有序链表还有其他快速访问中间节点的改造方案吗？我们知道，链表之所以访问中间节点的效率低，就是因为每个节点只存储了下一个节点的指针，要沿着这个指针遍历每个后续节点才能到达中间节点。那如果我们在节点上增加一个指针，指向更远的节点，比如说跳过后一个节点，直接指向后面第二个节点，那么沿着这个指针遍历，是不是遍历速度就翻倍了呢？

同理，如果我们能增加更多的指针，提供不同步长的遍历能力，比如一次跳过 4 个节点，甚至一半的节点，那我们是不是就可以更快速地访问到中间节点了呢？

这当然是可以实现的。我们可以为链表的某些节点增加更多的指针。这些指针都指向不同距离的后续节点。这样一来，链表就具备了更高效的检索能力。这样的数据结构就是**跳表**（Skip List）。

一个理想的跳表，就是从链表头开始，用多个不同的步长，每隔 2^n 个节点做一次直接链接（n 取值为 0，1，2……）。跳表中的每个节点都拥有多个不同步长的指针，我们可以在每个节点里，用一个数组 next 来记录这些指针。next 数组的大小就是这个节点的层数，next[0]就是第 0 层的步长为 1 的指针，next[1]就是第 1 层的步长为 2 的指针，next[2]就是第 2 层的步长为 4 的指针，依此类推。你会发现，不同步长的指针，在链表中的分布是非常均匀的，这使得整个链表具有非常平衡的检索结构。



举个例子，当我们要检索 k=a_6时，从第一个节点 a_1开始，用最大步长的指针开始遍历，直接就可以访问到中间节点 a_5。但是，如果沿着这个最大步长指针继续访问下去，下一个节点是大于 k 的 a_9，这说明 k 在 a_5和 a_9之间。那么，我们就在 a_5和 a_9之间，用小一个级别的步长继续查询。这时候，a_5的下一个元素是 a_7，a_7依然大于 k 的值，因此，我们会继续在 a_5和 a_7之间，用再小一个级别的步长查找，这样就找到 a_6了。这个过程其实就是二分查找。时间代价是 O(log n)。

跳表的检索空间平衡方案

不知道你有没有注意到，我在前面强调了一个词，那就是“理想的跳表”。为什么要叫它“理想”的跳表呢？难道在实际情况下，跳表不是这样实现的吗？的确不是。当我们要在跳表中插入元素时，节点之间的间隔距离就被改变了。如果要保证理想链表的每隔 2^n 个节点做一次链接的特性，我们就需要重新修改许多节点的后续指针，这会带来很大的开销。

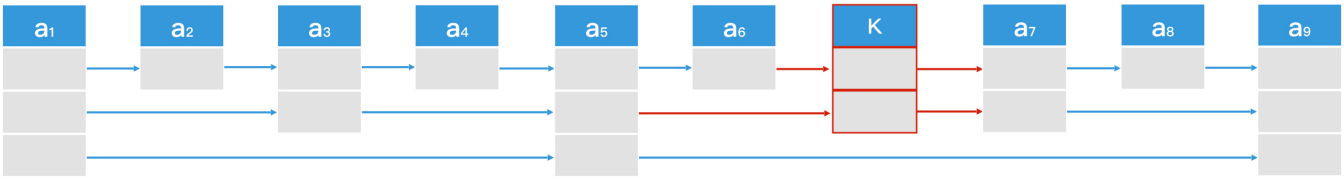
所以，在实际情况下，我们会在检索性能和修改指针代价之间做一个权衡。为了保证检索性能，我们不需要保证跳表是一个“理想”的平衡状态，只需要保证它在大概率上是平衡的就可以了。因此，当新节点插入时，我们不去修改已有的全部指针，而是仅针对新加入的节点为它建立相应的各级别的跳表指针。具体的操作过程，我们一起来看看。

首先，我们需要确认新加入的节点需要具有几层的指针。我们通过随机函数来生成层数，比如说，我们可以写一个函数 RandomLevel()，以 (1/2)^n 的概率决定是否生成第 n 层。这样，通过简单的随机生成指针层数的方式，我们就可以保证指针的分布，在大概率上是平衡的。

在确认了新节点的层数 n 以后，接下来，我们需要将新节点和前后的节点连接起来，也就是为每一层的指针建立前后连接关系。其实每一层的指针链接，你都可以看作是一个独立的单链表的修改，因此我们只需要用单链表插入节点的方式完成指针连接即可。

这么说，可能你理解起来不是很直观，接下来，我通过一个具体的例子进一步给你解释一下。

我们要在一个最高有 3 层指针的跳表中插入一个新元素 k，这个跳表的结构如下图所示。



极客时间

假设我们通过跳表的检索已经确认了，k 应该插入到 a_6和 a_7两个节点之间。那接下来，我们要先为新节点随机生成一个层数。假设生成的层数为 2，那我们就要修改第 0 层和第 1 层的指针关系。对于第 0 层的链表，k 需要插入到 a_6和 a_7之间，我们只需要修改 a_6和 a_7的第 0 层指针；对于第 1 层的链表，k 需要插入到 a_5和 a_7之间，我们只需要修改 a_5和 a_7的第 1 层指针。这样，我们就完成了将 k 插入到跳表中的动作。

通过这样一种方式，我们可以大大减少修改指针的代价。当然，由于新加入节点的层数是随机生成的，因此在节点数目较少的情况下，如果指针分布的不合理，检索性能依然可能不高。但是当节点数较多的时候，指针会趋向均匀分布，查找空间会比较平衡，检索性能会趋向于理想跳表的检索效率，接近 O(log n)。

因此，相比于复杂的平衡二叉检索树，如红黑树，跳表用一种更简单的方式实现了检索空间的平衡。并且，由于跳表保持了链表顺序遍历的能力，在需要遍历功能的场景中，跳表会比红黑树用起来更方便。这也就是为什么，在 Redis 这样的系统中，我们经常会利用跳表来代替红黑树作为底层的数据结构。

重点回顾

好了，关于非线性结构的检索技术，我们就先讲到这里。我们一起回顾一下今天的重点内容。

首先，对于数据频繁变化的应用场景，有序数组并不是最适合的解决方案。我们一般要考虑采用非连续存储的数据结构来灵活调整。同时，为了提高检索效率，我们还要采取合理的组织方式，让这些非连续存储的数据结构能够使用二分查找算法。

数据组织的方式有两种，一种是二叉检索树。一个平衡的二叉检索树使用二分查找的检索效率是 O(log n)，但如果我们不做额外的平衡控制的话，二叉检索树的检索性能最差会退化到 O(n)，也就和单链表一样了。所以，AVL 树和红黑树这样平衡性更强的二叉检索树，在实际工作中应用更多。

除了树结构以外，另一种数据组织方式是跳表。跳表也具备二分查找的能力，理想跳表的检索效率是 O(log n)。为了保证跳表的检索空间平衡，跳表为每个节点随机生成层级，这样的实现方式比 AVL 树和红黑树更简单。

无论是二叉检索树还是跳表，它们都是通过将数据进行合理组织，然后尽可能地平衡划分检索空间，使得我们能采用二分查找的思路快速地缩减查找范围，达到 O(log n) 的检索效率。

除此之外，我们还能发现，当我们从实际问题出发，去思考每个数据结构的特点以及解决方案时，我们就会更好地理解一些高级数据结构和算法的来龙去脉，从而达到更深入地理解和吸收知识的目的。并且，这种思考方式，会在不知不觉中提升你的设计能力以及解决问题的能力。

课堂讨论

今天的内容比较多，你可以结合我留的课堂讨论题，加深理解。

二叉检索树和跳表都能做到 O(log n) 的查询时间代价，还拥有灵活的调整能力，并且调整代价也是 O(log n)（包括了寻找插入位置的时间代价）。而有序数组的查询时间代价也是 O(log n)，调整代价是 O(n)，那这是不是意味着二叉检索树或者跳表可以用来替代有序数组呢？有序数组自己的优势又是什么呢？

欢迎在留言区畅所欲言，说出你的思考过程和最终答案。如果有收获，也欢迎把这篇文章分享给你的朋友。

极客时间

检索技术核心 20 讲

从搜索引擎到推荐引擎，带你吃透检索

陈东

奇虎 360 商业产品事业部
资深总监

新版升级：点击「请朋友读」，20位好友免费读，邀请订阅更有现金奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇　01 | 线性结构检索：从数组和链表的原理初窥检索本质

下一篇　03 | 哈希检索：如何根据用户ID快速查询用户信息?

精选留言 (17)

写留言

峰
2020-03-25

1，虽然时间复杂度一致，但并不代表真实的时间一致，时间复杂度只是量级，所以数据量小的情况下，数组能甩其他结构几条街。2，就是存储空间的节省。

这篇文章让我有很多延伸想法，1，检索操作映射到底层核心就是cpu对存储的寻址方式，而这上层的抽象就是指针和数组，老师向我们展示了添加指针构建二叉树的过程，反过...
展开

作者回复: 你的思考有两点非常好!
一个是从实际实现出发，而不是停留在纸面分析。由于有内存局部性原理，数组的查询效率是高于树和跳表的。甚至在小数据的情况下，都有可能数组的移动代价也不高（可用内存拷贝）。还有，数组还有范围查找能力更强的特点。
而另一点，你很好地去开始思考和抽象事情的核心，这是构建自己知识体系的很好的实践。

10

每天晒白牙
2020-03-25

跳表并不能完全替代有序数组
1.有序数据占用的内存空间小于调表
2.有序数组的读取操作能保持在很稳定的时间复杂度，而调表并不能
3.因为数组存储空间是连续的，可以利用内存的局部性原理加快查询
...
展开

5

一步
2020-03-25

有序数组 使用的是一段连续的内存可以支持随机访问，而且由于使用的是连续的内存的 可以高效使用 CPU 的局部性原理，可以缓存要访问数据之后的数据，进而范围查询更高效

作者回复: 总结得很好

4

老姜
2020-03-25

随机访问，充分利用CPU缓存，节省内存
展开

作者回复: 回答很简明扼要!
除此之外，还有范围查找效率更高（CPU缓存 + 内存拷贝）

4

旭东
2020-03-29

这个算法层层引入挺好，学习方法
展开

作者回复: 这个也是我专栏设计的目标之一。用前面的知识引出后面的知识。后面的知识是前面的知识的灵活应用。
课程用到的知识基本都是自成体系，这样大家在学习的时候就不用再去其他地方补基础知识了。

2

Harry
2020-03-25

为了将 k 插入到跳表中，需要检索跳表以确定其插入位置，同时还要为其生成一个随机的层级。

如何确定插入位置？为何使用一个随机的层级？目前还没有头绪...

展开

作者回复: 1.确认k的插入位置 : 其实就是在跳表中查询“k”，如果不存在，那么最后的查询位置，就是k应该插入的位置；如果k已经存在（假设允许重复元素），那么就在找到的k的后面插入。

2.随机的层级，是用概率的思路来解决跳表指针平衡分布的问题。我可以换一种角度再描述一下。
你可以按我的描述，在纸上将图画出来，看看是否会好理解:
假设我们有m个节点，由于随机函数是（1/2）^n，因此，每个节点有第0层的概率是1，也就是说，所有的节点都有一个指向下一个节点的指针。（试着画出来）
而节点拥有第1层的概率，变成了1/2，也就是只有一半的随机节点会有第1层，那么这一半的节点就会连起来。（试着画出来）
在这一半的节点中，拥有第3层的节点数，又是随机的一半。（试着画出来）
你会看到，通过一个简单的随机函数，就能完成跳表的平衡分布指针的目的。

2

SkillIP
2020-03-25

最主要还是以空间换时间吧!
还有就是要考虑业务场景，如果存储的对象本身没有多大，却存了一大堆地址，是得不偿失的。

展开

作者回复: 是的。合理的空间利用率也是很重要的事情。
此外，由于有内存局部性原理，因此，连续空间上的查询性能实际上会比树和跳表好

2

千里之行
2020-03-25

在小数据量、修改不频繁的场景下，有序数组可以获得稳定的且较短的查询时间，但调表由于新插入元素的指针间隔并不均匀，所以查询时间就得不到很好的保证

展开

作者回复: 是的，小数据量下，数组会有更稳定的性能。
但不仅如此，由于内存局部性原理，大数据量下，数组的查询效率依然很高。
此外，对于范围查找，数组也有更高的效率（内存局部性原理+内存拷贝）

2

每天晒白牙
2020-03-25

跳表并不能完全替代有序数组
1.有序数据占用的内存空间小于调表
2.有序数组的读取操作能保持在很稳定的时间复杂度，而调表并不能
3.因为数组存储空间是连续的，可以利用内存的局部性原理加快查询

展开

作者回复: 在内存空间之外，你说到了很重要的两点:
1.小数据范围下，跳表性能没有数组稳定。
2.考虑内存局部性原理，数组实际查询效率更高。

此外还有一点，考虑到范围查找需求，数据的处理效率会更高（内存拷贝+内存局部性原理）

2

一单成名
2020-03-25

只想到一个节约内存空间

展开

作者回复: 节约空间的确是最大的一个优点。很多时候，内存空间是紧张的，甚至需要我们做数据压缩处理。在这种场景下，如果适合用数组，就不应该用树或者跳表。
此外，尽管理论上时间代价级别相同，但由于有内存局部性原理，连续空间的数据被处理的效率会更高。因此数组的实际查询效率会更高。

并且，如果考虑范围查找，数组能快速处理大段区域（比如使用内存拷贝技术），再叠加局部性原理，范围查找数组效率会远高于跳表和树。

1 2

aoe
2020-03-30

对数组、链表、跳表、树的搜索有了进一步认识。谢谢老师！

作者回复: 继续加油！

1

努力努力再努力Xmn
2020-03-29

老师说RandonLevel的结果是随机的，如果说RandomLevel的结果没有范围的话，那next数组的大小不就没法确定了吗？

展开

作者回复: 其实可以使用可变长数组来处理。一种简单的可变长数组实现方案就是:先给数组一个固定的初始长度，如果数组写满了，就生成一个原数组两倍大小的新数组，然后将原数组的元素都导入新数组中。
关于可变长数组，你可以参考一下c++中vector的实现方案。

1

张珮磊想静静
2020-03-28

在读少写多的场景下，需要频繁变更以使得数据有序，是不是就不会考虑用数组，而是优先考虑二叉树和跳表

作者回复: 如果频繁修改的代价很大，影响到了整体性能，那么的确二叉树和跳表表现会更好。当然，如果数据规模不大的情况下，数组性能有内存局部性原理的加持，即使是频繁插入和删除，也不一定性能很差。因此要合理评估。

1

明翼

2020-03-27

数组有自己的独特优势，比如数据集中存放的，程序在从内存到缓存取数据的时候，一次性搬一块数据，临近的数据都搬进来了，而二叉树和调表，用指针连接，无法利用缓存局部性原理的优势；虽然数组插入和删除性能差，当时有的场景并不需要这些，有的只需要构建好就不变了，这种比较适合用数组。

展开

作者回复: 是的。数组的内存局部性优势是很重要的一点。甚至对于小规模数据集来说，插入删除时大块移动数据的性能也不见得差。

1

努力努力再努力Xmn

2020-03-26

对于图中的插入节点k，听了老师的讲解也明白了，但是在思考具体的实现的时候，不知道如何下手，对于图中的单链表而言，插入节点k需要修改两层的指针连接，那它不就需要两个pre指针，分别指向a5, a6;那如果要插入的节点位于a4和a5之间而且RandomLevel的结果为3的话，岂不是需要3个pre指针；昨晚想了好久也没想出来应该如何实现插入，今天我在总结向老师提问的时候突然想到是不是跳表在设计的时候每一层都有一个指针呢？...

展开

作者回复: 动手实践是非常棒的！尤其是我的专栏重点是在检索原理和分析上，因此如果你能自己去学习细节，这就是非常好的补充。接下来，我来给你补充一下细节:

1.跳表本身已经有了许多指针，因此如果用双链表的方式实现的话，指针数会翻倍。不合适。因此，跳表是单向的，没有pre指针。

2.对于单链表的插入，我们的具体实现是给单链表加上一个头节点，然后寻找插入位置的“pre节点”来完成操作。跳表也是一样，有一个头节点。头节点的level，是所以节点中level的最大值。从头节点开始，每个level的指针链接，可以看做多个独立的单链表。（你可以画出来看看）

3.插入新节点的时候，核心思路是“寻找每个level的pre节点”。以我文章中插入k的例子来说，我们是怎么寻找到它的插入位置的？我们是先用第2层的指针进行遍历，发现k应该在a5到a9之间。于是，k的第2层的pre节点是a5！把a5记下来！（pre_node[2]=a5；）

然后，接下来，我们会用第1层的指针去遍历，这时候，会发现k在a5和a7之间。于是，k的第1层的pre节点是a5！！（pre_node[1]=a5；）

最后，用第0层的指针遍历，发现k在a6和a7之间，于是，k的第0层的pre节点是a6！！（pre_node[0]=a6；）

下一步，就把每一层的pre node和k的对应层连起来就好了。

4.一个有意思的场景，k有小概率生成4层指针，甚至5层指针，如果超出了之前所有节点的level，那么头节点就会扩展自己的level，低层level处理不变。新增的高层level直接连到k上。你会发现，随机level这个方法，完全基于概率，不用去考虑当前跳表中有多少个节点，这也是很巧妙的一点。

第二个问题:二叉检索树如何处理重复节点?如果你注意看我原文，我写了“右子树所有节点的值都大于等于根节点”。因此，二叉检索树是允许插入重复节点的。它的插入逻辑是“放在以该节点为根，右子树的最左节点的位置”。其实所谓“右子树的最左节点的位置”，其实拉直成链表来看，就是紧邻这个元素的后一个位置。

1

pedro
2020-03-25

我在学习红黑树的时候，深知红黑树的复杂，即使后来阅读《算法4》通过2-3树的方式来写红黑树，其实还是很有难度的，跳表在本身实现简单的情况下拥有和红黑树同等级别的性能，虽然应用没有红黑树广泛，但是确实是一个极其精巧的数据结构。

展开

作者回复: 是的。所以跳表的设计很精巧。而且这种随机搭建高速通路的思路，在一些图搜索的场景中也有借鉴。

1

jkhcw
2020-03-25

有序数组优势是:一，快速而稳定的读性能，时间复杂度永远是O（1）.二，不会导致内存碎片化，避免了内存浪费.

展开

作者回复: 数组的确比跳表更加稳定。不过内存碎片化问题，这个会有内存碎片处理来解决。

其实，还有两点是很关键的，一个是考虑实际情况，内存有局部性原理，对连续空间的处理会更高效；另一个是考虑范围查找，数组将查询结果成片复制出来效率会更高（内存拷贝技术+内存局部性原理）

1