自己动手实现跳表

作者：zhengliwei
原文链接：https://ld246.com/article/1603534671622
来源网站：链滴

hello，大家好，欢迎来到银之庭。我是Z，一个普通的程序员。今天我们来讲一下跳表这种数据结构并尝试自己动手实现一个跳表。

1. 简介

先来简单了解下跳表。跳表，全称跳跃表，英文名skip list。它的使用场景是对有序数据的增加，删和修改。这种场景我们一般使用的数据结构有红黑树，AVL树等，这些数据结构能让增，删，查的时复杂度都稳定在O(logn)上，但缺点是实现复杂，理解困难。而跳表可以说是他们的对立面，跳表通使用随机的方式，让增，删，查的平均时间复杂度维持在O(logn)，同时实现非常简单，容易理解，然，代价就是在最差的情况下操作的时间复杂度会变成稍高于O(n)，而且有轻微的空间占用增加，毕，天下没有免费的午餐啊。

说实话，我们在业务开发中用跳表的机会不多，我个人理解，跳表是针对一个需求特化出来的一种数结构，它本身的数据组织形式是没有意义的（反例就是树型结构，在业务中通常用来表示部门架构等务数据，它的数据组织形式就是有意义的），如果我们直接在业务代码中用跳表来维护一组对象，后的维护者很可能不知道为什么要把数据组织成这种形式。

跳表一般用在框架或组件中，这些地方对业务开发者比较透明，可以使用一些复杂，或抽象的数据结来达到提高性能的目的。跳表的典型使用场景就是redis的源码，redis用跳表作为sorted set在数据较大时的底层存储数据结构。

2. 原理

下面我们来看下跳表的实现原理。

跳表可以认为是从单链表演化来的。单链表的整体结构如下：

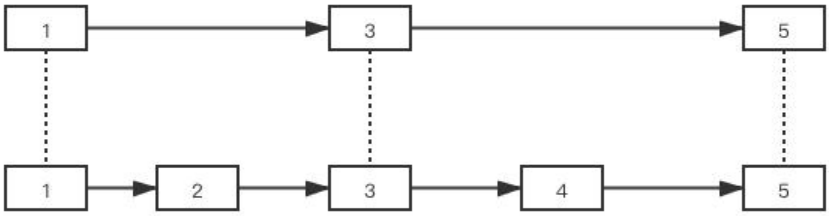
节点一般只有当前值和下一节点的指针两个属性，如下：

```
@Data
public class Node<T> {
    private T data;
    private Node<T> next;
}
```

单链表在查询时需要从前往后，一个一个地遍历节点，效率较低，有没有办法让链表像有序数组一样以二分查找呢？当然有啦～比如，我们在原链表的基础上增加一级索引，这级索引只包含部分元素，查找时，可以像在B树上查找一样，快速定位需要查找的元素的所在区间，然后去下一级列表中去遍这一区间的数据。

PS：网上有人把在跳表上的查询类比为在有序数组上的二分查找，我个人觉得更像是在B树上的查找B树可以认为也是给底层节点（叶子节点）构造了多级索引（叶子节点之上的各层节点），在查找时是通过各级索引快速定位所查节点在下一级索引中的区间，然后到下一级索引再定位下下一级索引的间，以此类推，最后到达底层，也就是顺序存储的节点列表，再一个一个遍历即可。跳表和B树的区在于多级索引的组织形式不同。

给上面的单链表增加一级索引后结构类似这样：



上面一层我们称之为第一级索引，下面一层为第零级索引，在这种结构上查找指定元素时，可以从第季索引开始遍历，找到小于指定元素的最大的一个（即next >= value的元素），然后向下一级索引续遍历，假如我们要找的是4的话，我们的遍历顺序会是1 -> 3 -> 4，省略了2的遍历，在数据量更时这种优势会更明显。把这种索引多加几级，就能实现平均时间复杂度为O(logn)的查找了。

另外，为了索引层数不爆炸，通常我们会限制一个最高索引层数，一般16或32就足够了。为了让查找数据量更少，我们在插入元素决定这个元素要位于第几层时（底层为第零层，往上依次增加层数，如面图里的元素3，就位于第一层）可以从第零层开始，50%概率往上走一层，如果不走，则直接返回前层数，如果走了，继续算概率，确定要不要继续往上走，直到概率不命中，或到最大层数。这样一，从下往上层中元素个数会以logn的程度递减，这时从上往下查找起来的时间复杂度才是O(logn)。

插入节点时，我们需要先确定这个节点所处的层数，这是个随机的过程，上面也说了，假如这次的节确定各位第二层，我们就需要维护第零，一，二层的指针关系，把新节点加到这几层。同样，在删除点时，我们也需要维护要删除的节点所在的层及以下所有层的指针关系，把指向这个节点的指针调整指向下一个节点，类似链表的删除过程。

3. 代码实现

到这是不是觉得跳表很简单？太天真了，你可以先试试不看任何网上的代码，自己试着实现一个，你会体会到什么叫脑子：好，我会了。手：不，你没有。了。我也是自信满满地打开编辑器，然后憋了天也没写出啥代码来，最后还是参考着网上的代码才写出来。。。如下：

```
class Node<T extends Comparable<T>> {
    public T data;
    public int level; // 节点所在层数，从0开始计算
    public Node<T>[] nextList; // 当前节点在不同层的下一节点指针
}

public class SkipList<T extends Comparable<T>> {

    private static final int MAX_LEVEL = 16; // 最大层数

    private Node<T> head; // 头节点

    public SkipList() {
        this.head = new Node<>();
        head.level = MAX_LEVEL - 1; // 头节点直接置为顶层，因为后续遍历都是从头节点开始的
        head.nextList = new Node[MAX_LEVEL];
    }

    /**
     * 添加元素
     *
     * @param value
     */
```

```
    public void add(T value) {
        Node<T> node = new Node<>();
        node.data = value;
        int level = getLevel();
        node.level = level;
        node.nextList = new Node[level];

        Node<T> cur = head;
        Node<T>[] needUpdateNodes = new Node[level]; // 保存各层需要调整的节点
        // 从新节点所在层开始（因为所在层以上的层是肯定不需要调整的），找到新节点左侧的节点就是需要调整的节点
        for (int i = level - 1; i >= 0; i--) {
            while (cur.nextList[i] != null && cur.nextList[i].data.compareTo(value) < 0) {
                cur = cur.nextList[i];
            }
            needUpdateNodes[i] = cur;
        }
        // 调整各层指定节点，实际上就是插入新节点，类似链表插入新节点
        for (int i = level - 1; i >= 0; i--) {
            node.nextList[i] = needUpdateNodes[i].nextList[i];
            needUpdateNodes[i].nextList[i] = node;
        }

    }

    /**
     * 删除指定元素
     *
     * @param value
     */
    public void remove(T value) {
        Node<T> p = head;
        int level = MAX_LEVEL - 1;
        while (level >= 0) {
            if (p.nextList[level] != null && p.nextList[level].data.compareTo(value) < 0) {
                // 该层的下一个节点值比要删除的值小，可以直接跳到下一个节点。
                p = p.nextList[level];
            } else if (p.nextList[level] != null && p.nextList[level].data.compareTo(value)>0) {
                // 该层的下一个节点值比要删除的大，应该跳到下一层继续查找。
                level -= 1;
            } else if (p.nextList[level] != null) {
                // 该层下一个节点值等于要删除的值，在该层把下一个节点删掉，类似链表删除，然后继跳到下一层处理。
                p.nextList[level] = p.nextList[level].nextList[level];
                level -= 1;
            } else {
                // 该节点是最后一个节点，或是个空条表，直接向下，最后跳出循环。
                level -= 1;
            }
        }
    }

    /**
     * 按值查找元素
```

```
     *
     * @param value
     * @return
     */
    public Node<T> find(T value) {
        Node<T> p = head;
        int level = MAX_LEVEL - 1;
        while (level >= 0) {
            if (p.nextList[level] != null && p.nextList[level].data.compareTo(value) < 0) {
                p = p.nextList[level];
            } else if (p.nextList[level] != null && p.nextList[level].data.compareTo(value)>0) {
                level -= 1;
            } else if (p.nextList[level] != null) {
                return p.nextList[level];
            } else {
                level -= 1;
            }
        }

        return null;
    }

    /**
     * 获取层数，从1层开始，每次有1/2的概率向上一层，直到没有命中概率或到达顶层
     *
     * @return
     */
    private int getLevel() {
        int level = 1;
        while (true) {
            if (Math.random() < 0.5 && level < MAX_LEVEL) {
                level += 1;
            } else {
                break;
            }
        }

        return level;
    }

}
```

注释比较详细，这里就不多赘述了，需要注意理解的一点是：不同节点的nextList数组在同一下标处组成了一个类似链表的结构的，比如在第二层，就有node1.nextList[2] = node3, node3.nextList[2] = node5，如果有个节点node2是第二层或以上的节点，值在node1和node3之间，插入它时就需要它的node2.nextList[2]进行处理，类似链表插入，代码形如node2.nextList[2] = node3; node1.nextlist[2] = node2。理解了这一点就比较容易写代码了。

PS：上面代码里的remove()和find()方法是我自己写的，网上大多数实现都是类似add()方法，双重历，对于我的代码我实验了几种情况，没什么问题，如果某位同学发现了我代码里的错误，欢迎留言出～

PPS：代码纯手写，只实现了最基础的功能，没有进行任何优化，肯定比不上各种开源组件里的实现

以后有机会再给大家介绍几个更优秀的实现吧。