stackOverFlow

B树Java代码实现以及测试

B树定义

B 树又叫平衡多路查找树。**一棵m阶的B 树 (m叉树)的特性如下**：

- 根节点至少有两个孩子
- 每个非根节点至少有M/2（上取整）个孩子，至多有M个孩子。
- 每个非根节点至少有M/2-1(上取整)个关键字，至多有M-1个关键字。并以升序排列。
- key[i]和key[i+1]之间的孩子节点的值介于key[i]和key[i+1]之间。
- 所有的叶子节点都在同一层。

注意：B-树，即为B树。

B树Java实现

```
/**
 * 一颗B树的简单实现。
 *
 * @param <K> - 键类型
 * @param <V> - 值类型
 */
@SuppressWarnings("all")
public class BTree<K, V> {
    private static Log logger = LogFactory.getLog(BTree.class);

    /**
     * B树节点中的键值对。
     * <p/>
     * B树的节点中存储的是键值对。
     * 通过键访问值。
     *
     * @param <K> - 键类型
     * @param <V> - 值类型
     */
    private static class Entry<K, V> {
        private K key;
        private V value;

        public Entry(K k, V v) {
            this.key = k;
            this.value = v;
        }

        public K getKey() {
            return key;
        }

        public V getValue() {
            return value;
```

昵称： kosamino
园龄： 5年3个月
粉丝： 198
关注： 32
+加关注

< 2021年11月 >

日	一	二	三	四	五	六
31	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	1	2	3	4
5	6	7	8	9	10	11

搜索

找找看
谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

我的标签

设计模式(29)
MySQL(23)
Linux(17)
Linux命令(16)
tomcat(13)
redis(12)
mongodb(10)
hashMap(9)
Thread(9)
Netty(7)
更多

积分与排名

积分 - 415736
排名 - 1329

随笔分类

docker(1)
elasticsearch(3)
go(1)
IO(6)
Java基础(37)
JDK特性(3)

```
        }

        public void setValue(V value) {
            this.value = value;
        }

        @Override
        public String toString() {
            return key + ":" + value;
        }
    }

    /**
     * 在B树节点中搜索给定键值的返回结果。
     * <p/>
     * 该结果有两部分组成。第一部分表示此次查找是否成功，
     * 如果查找成功，第二部分表示给定键值在B树节点中的位置，
     * 如果查找失败，第二部分表示给定键值应该插入的位置。
     */
    private static class SearchResult<V> {
        private boolean exist;
        private int index;
        private V value;

        public SearchResult(boolean exist, int index) {
            this.exist = exist;
            this.index = index;
        }

        public SearchResult(boolean exist, int index, V value) {
            this(exist, index);
            this.value = value;
        }

        public boolean isExist() {
            return exist;
        }

        public int getIndex() {
            return index;
        }

        public V getValue() {
            return value;
        }
    }

    /**
     * B树中的节点。
     * <p>
     * TODO 需要考虑并发情况下的存取。
     */
    private static class BTreeNode<K, V> {
        /**
         * 节点的项，按键非降序存放
         */
        private List<Entry<K, V>> entrys;
        /**
         * 内节点的子节点
         */
        private List<BTreeNode<K, V>> children;
        /**
         * 是否为叶子节点
         */
        private boolean leaf;
```

JVM及JVM调优(14)
Kafka(4)
Linux场景应用(35)
Linux命令详解(31)
MongoDB(11)
mybatis(4)
MySQL(26)
Netty(12)
Nginx(4)
更多

随笔档案

2021年6月(1)
2021年4月(3)
2021年3月(1)
2021年2月(1)
2020年11月(4)
2020年10月(6)
2020年9月(3)
2020年7月(1)
2020年5月(8)
2020年4月(20)
2020年3月(30)
2019年12月(4)
2019年11月(7)
2019年10月(24)
2019年9月(12)
更多

阅读排行榜

1. Centos和Redhat的区别与联系(88392)
2. SpringCloud之Zuul网关原理及其配置(86785)
3. SpringCloud之Eureka注册中心原理及其搭建(78276)
4. java集合继承关系图(48268)
5. SpringBoot定时任务(schedule、quartz)(43612)
6. Map、Set、List集合差别及联系详解(35039)
7. 死锁产生的原因和解锁的方法(30419)
8. Linux软链接创建及删除(27303)
9. SpringCloud之RabbitMQ消息队列原理及配置(25106)
10. 分布式事务解决方案及实现(23746)

评论排行榜

1. SpringCloud之Zuul网关原理及其配置(8)
2. 国密SM2加解密Java工具类(附前端VUE代码)(6)
3. SpringCloud之Hystrix容错保护原理及配置(4)
4. java集合继承关系图(4)
5. Java构建指定大小文件(3)

推荐排行榜

1. SpringCloud之Zuul网关原理及其配置(32)
2. SpringCloud之Eureka注册中心原理及其搭建(10)
3. java集合继承关系图(8)
4. SpringCloud之Hystrix容错保护原理及配置(7)
5. SpringBoot定时任务(schedule、quartz)(6)

最新评论

```
    /**
     * 键的比较函数对象
     */
    private Comparator<K> kComparator;

    private BTreeNode() {
        entrys = new ArrayList<Entry<K, V>>();
        children = new ArrayList<BTreeNode<K, V>>();
        leaf = false;
    }

    public BTreeNode(Comparator<K> kComparator) {
        this();
        this.kComparator = kComparator;
    }

    public boolean isLeaf() {
        return leaf;
    }

    public void setLeaf(boolean leaf) {
        this.leaf = leaf;
    }

    /**
     * 返回项的个数。如果是非叶子节点，根据B树的定义，
     * 该节点的子节点个数为({@link #size()} + 1)。
     *
     * @return 关键字的个数
     */
    public int size() {
        return entrys.size();
    }

    int compare(K key1, K key2) {
        return kComparator == null ? ((Comparable<K>) key1).co
    }

    /**
     * 在节点中查找给定的键。
     * 如果节点中存在给定的键，则返回一个<code>SearchResult</code>，
     * 标识此次查找成功，给定的键在节点中的索引和给定的键关联的值；
     * 如果不存在，则返回<code>SearchResult</code>，
     * 标识此次查找失败，给定的键应该插入的位置，该键的关联值为null。
     * <p/>
     * 如果查找失败，返回结果中的索引域为[0, {@link #size()}];
     * 如果查找成功，返回结果中的索引域为[0, {@link #size()} - 1]
     * <p/>
     * 这是一个二分查找算法，可以保证时间复杂度为O(log(t))。
     *
     * @param key - 给定的键值
     * @return - 查找结果
     */
    public SearchResult<V> searchKey(K key) {
        int low = 0;
        int high = entrys.size() - 1;
        int mid = 0;
        while (low <= high) {
            mid = (low + high) / 2; // 先这么写吧，BTree实现中，l
            Entry<K, V> entry = entrys.get(mid);
            if (compare(entry.getKey(), key) == 0) // entrys.g
                break;
            else if (compare(entry.getKey(), key) > 0) // entr
                high = mid - 1;
            else // entry.get(mid).getKey() < key
```

1. Re:国密SM2加解密Java工具类(附前端VUE代码)

@彼岸之云 暂时还没有，要是你这边测试了1.6.以上版本，可以一起交流呢，我也是因为项目明确要求国密方案时，临时做的...

--kosamino

2. Re:国密SM2加解密Java工具类(附前端VUE代码)

@kosamino 你好，发现问题了，你用的jar包是1.56的，我用的是1.61的，1.6以上的版本里好多方法跟1.6以下的版本的方法都不一样了，而我这边必须得用1.6以上的jar包，网上的代码都是...

--彼岸之云

3. Re:国密SM2加解密Java工具类(附前端VUE代码)

@彼岸之云 是Java自身加解后解密是乱码么，我上面的代码应该没这个问题的，你看看是不是你没有生成对应的公私钥-public_key/private_key...

--kosamino

4. Re:国密SM2加解密Java工具类(附前端VUE代码)

大佬，我用Java代码加密后再去解密解出来的是乱码，而且好像每次解出来的值都在变，这是什么原因呢?

--彼岸之云

5. Re:SpringCloud之RabbitMQ消息队列原理及配置

博主厉害

--我是彭于晏

6. Re:SpringCloud之Eureka注册中心原理及其搭建

优雅停服！是个好东西!

这个 actuator 用的越来越多了，几乎成为了spring boot/cloud项目的标配了

--快乐的凡人721

7. Re:Redis缓存策略设计及常见问题

16384

--小小小尘埃

8. Re:Netty学习之Demo搭建

注释很详尽,针不戳

--蒜蓉鲮鱼

9. Re:国密SM2加解密Java工具类(附前端VUE代码)

@牛成 <dependency>
<groupId>org.bouncycastle</groupId>
<artifactId>bcprov-jdk15on</artifactId> <versio...

--kosamino

10. Re:国密SM2加解密Java工具类(附前端VUE代码)

大佬，你maven引入的bouncycastle是哪个版本的?

--牛成

```
                low = mid + 1;
            }
            boolean result = false;
            int index = 0;
            V value = null;
            if (low <= high) // 说明查找成功
            {
                result = true;
                index = mid; // index表示元素所在的位置
                value = entrys.get(index).getValue();
            } else {
                result = false;
                index = low; // index表示元素应该插入的位置
            }
            return new SearchResult<V>(result, index, value);
        }

        /**
         * 将给定的项追加到节点的末尾，
         * 你需要自己确保调用该方法之后，节点中的项还是
         * 按照关键字以非降序存放。
         *
         * @param entry - 给定的项
         */
        public void addEntry(Entry<K, V> entry) {
            entrys.add(entry);
        }

        /**
         * 删除给定索引的<code>entry</code>。
         * <p/>
         * 你需要自己保证给定的索引是合法的。
         *
         * @param index - 给定的索引
         */
        public Entry<K, V> removeEntry(int index) {
            return entrys.remove(index);
        }

        /**
         * 得到节点中给定索引的项。
         * <p/>
         * 你需要自己保证给定的索引是合法的。
         *
         * @param index - 给定的索引
         * @return 节点中给定索引的项
         */
        public Entry<K, V> entryAt(int index) {
            return entrys.get(index);
        }

        /**
         * 如果节点中存在给定的键，则更新其关联的值。
         * 否则插入。
         *
         * @param entry - 给定的项
         * @return null，如果节点之前不存在给定的键，否则返回给定键之前关联
         */
        public V putEntry(Entry<K, V> entry) {
            SearchResult<V> result = searchKey(entry.getKey());
            if (result.isExist()) {
                V oldValue = entrys.get(result.getIndex()).getValu
                entrys.get(result.getIndex()).setValue(entry.getVa
                return oldValue;
            } else {
```

```
            insertEntry(entry, result.getIndex());
            return null;
        }
    }

    /**
     * 在该节点中插入给定的项，
     * 该方法保证插入之后，其键值还是以非降序存放。
     * <p/>
     * 不过该方法的时间复杂度为O(t)。
     * <p/>
     * <b>注意：</b>B树中不允许键值重复。
     *
     * @param entry - 给定的键值
     * @return true，如果插入成功，false，如果插入失败
     */
    public boolean insertEntry(Entry<K, V> entry) {
        SearchResult<V> result = searchKey(entry.getKey());
        if (result.isExist())
            return false;
        else {
            insertEntry(entry, result.getIndex());
            return true;
        }
    }

    /**
     * 在该节点中给定索引的位置插入给定的项，
     * 你需要自己保证项插入了正确的位置。
     *
     * @param entry - 给定的键值
     * @param index - 给定的索引
     */
    public void insertEntry(Entry<K, V> entry, int index) {
        /*
         * 通过新建一个ArrayList来实现插入真的很恶心，先这样吧
         * 要是有类似C中的reallocate就好了。
         */
        List<Entry<K, V>> newEntrys = new ArrayList<Entry<K, V
        int i = 0;
        // index = 0或者index = keys.size()都没有问题
        for (; i < index; ++i)
            newEntrys.add(entrys.get(i));
        newEntrys.add(entry);
        for (; i < entrys.size(); ++i)
            newEntrys.add(entrys.get(i));
        entrys.clear();
        entrys = newEntrys;
    }

    /**
     * 返回节点中给定索引的子节点。
     * <p/>
     * 你需要自己保证给定的索引是合法的。
     *
     * @param index - 给定的索引
     * @return 给定索引对应的子节点
     */
    public BTreeNode<K, V> childAt(int index) {
        if (isLeaf())
            throw new UnsupportedOperationException("Leaf node
        return children.get(index);
    }

    /**
```

```
         * 将给定的子节点追加到该节点的末尾。
         *
         * @param child - 给定的子节点
         */
        public void addChild(BTreeNode<K, V> child) {
            children.add(child);
        }

        /**
         * 删除该节点中给定索引位置的子节点。
         * </p>
         * 你需要自己保证给定的索引是合法的。
         *
         * @param index - 给定的索引
         */
        public void removeChild(int index) {
            children.remove(index);
        }

        /**
         * 将给定的子节点插入到该节点中给定索引
         * 的位置。
         *
         * @param child - 给定的子节点
         * @param index - 子节点带插入的位置
         */
        public void insertChild(BTreeNode<K, V> child, int index)
            List<BTreeNode<K, V>> newChildren = new ArrayList<BTre
            int i = 0;
            for (; i < index; ++i)
                newChildren.add(children.get(i));
            newChildren.add(child);
            for (; i < children.size(); ++i)
                newChildren.add(children.get(i));
            children = newChildren;
        }
    }

    private static final int DEFAULT_T = 2;

    /**
     * B树的根节点
     */
    private BTreeNode<K, V> root;
    /**
     * 根据B树的定义，B树的每个非根节点的关键字数n满足(t - 1) <= n <= (2t
     */
    private int t = DEFAULT_T;
    /**
     * 非根节点中最小的键值数
     */
    private int minKeySize = t - 1;
    /**
     * 非根节点中最大的键值数
     */
    private int maxKeySize = 2 * t - 1;
    /**
     * 键的比较函数对象
     */
    private Comparator<K> kComparator;

    /**
     * 构造一颗B树，键值采用采用自然排序方式
     */
    public BTree() {
```

```
        root = new BTreeNode<K, V>();
        root.setLeaf(true);
    }

    public BTree(int t) {
        this();
        this.t = t;
        minKeySize = t - 1;
        maxKeySize = 2 * t - 1;
    }

    /**
     * 以给定的键值比较函数对象构造一颗B树。
     *
     * @param kComparator - 键值的比较函数对象
     */
    public BTree(Comparator<K> kComparator) {
        root = new BTreeNode<K, V>(kComparator);
        root.setLeaf(true);
        this.kComparator = kComparator;
    }

    public BTree(Comparator<K> kComparator, int t) {
        this(kComparator);
        this.t = t;
        minKeySize = t - 1;
        maxKeySize = 2 * t - 1;
    }

    @SuppressWarnings("unchecked")
    int compare(K key1, K key2) {
        return kComparator == null ? ((Comparable<K>) key1).compar
    }

    /**
     * 搜索给定的键。
     *
     * @param key - 给定的键值
     * @return 键关联的值，如果存在，否则null
     */
    public V search(K key) {
        return search(root, key);
    }

    /**
     * 在以给定节点为根的子树中，递归搜索
     * 给定的<code>key</code>
     *
     * @param node - 子树的根节点
     * @param key  - 给定的键值
     * @return 键关联的值，如果存在，否则null
     */
    private V search(BTreeNode<K, V> node, K key) {
        SearchResult<V> result = node.searchKey(key);
        if (result.isExist())
            return result.getValue();
        else {
            if (node.isLeaf())
                return null;
            else
                search(node.childAt(result.getIndex()), key);

        }
        return null;
    }
```

```
    /**
     * 分裂一个满子节点<code>childNode</code>。
     * <p/>
     * 你需要自己保证给定的子节点是满节点。
     *
     * @param parentNode - 父节点
     * @param childNode  - 满子节点
     * @param index      - 满子节点在父节点中的索引
     */
    private void splitNode(BTreeNode<K, V> parentNode, BTreeNode<K,
        assert childNode.size() == maxKeySize;

        BTreeNode<K, V> siblingNode = new BTreeNode<K, V>(kCompara
        siblingNode.setLeaf(childNode.isLeaf());
        // 将满子节点中索引为[t, 2t - 2]的(t - 1)个项插入新的节点中
        for (int i = 0; i < minKeySize; ++i)
            siblingNode.addEntry(childNode.entryAt(t + i));
        // 提取满子节点中的中间项，其索引为(t - 1)
        Entry<K, V> entry = childNode.entryAt(t - 1);
        // 删除满子节点中索引为[t - 1, 2t - 2]的t个项
        for (int i = maxKeySize - 1; i >= t - 1; --i)
            childNode.removeEntry(i);
        if (!childNode.isLeaf()) // 如果满子节点不是叶节点，则还需要处理
        {
            // 将满子节点中索引为[t, 2t - 1]的t个子节点插入新的节点中
            for (int i = 0; i < minKeySize + 1; ++i)
                siblingNode.addChild(childNode.childAt(t + i));
            // 删除满子节点中索引为[t, 2t - 1]的t个子节点
            for (int i = maxKeySize; i >= t; --i)
                childNode.removeChild(i);
        }
        // 将entry插入父节点
        parentNode.insertEntry(entry, index);
        // 将新节点插入父节点
        parentNode.insertChild(siblingNode, index + 1);
    }

    /**
     * 在一个非满节点中插入给定的项。
     *
     * @param node  - 非满节点
     * @param entry - 给定的项
     * @return true，如果B树中不存在给定的项，否则false
     */
    private boolean insertNotFull(BTreeNode<K, V> node, Entry<K, V
        assert node.size() < maxKeySize;

        if (node.isLeaf()) // 如果是叶子节点，直接插入
            return node.insertEntry(entry);
        else {
            /* 找到entry在给定节点应该插入的位置，那么entry应该插入
             * 该位置对应的子树中
             */
            SearchResult<V> result = node.searchKey(entry.getKey()
            // 如果存在，则直接返回失败
            if (result.isExist())
                return false;
            BTreeNode<K, V> childNode = node.childAt(result.getInd
            if (childNode.size() == 2 * t - 1) // 如果子节点是满节点
            {
                // 则先分裂
                splitNode(node, childNode, result.getIndex());
                /* 如果给定entry的键大于分裂之后新生成项的键，则需要插入该
                 * 否则左边。
```

```
             */
            if (compare(entry.getKey(), node.entryAt(result.ge
                childNode = node.childAt(result.getIndex() + 1
        }
        return insertNotFull(childNode, entry);
    }
}

/**
 * 在B树中插入给定的键值对。
 *
 * @param key                          - 键
 * @param value                        - 值
 * @return true，如果B树中不存在给定的项，否则false
 */
public boolean insert(K key, V value) {
    if (root.size() == maxKeySize) // 如果根节点满了，则B树长高
    {
        BTreeNode<K, V> newRoot = new BTreeNode<K, V>(kCompara
        newRoot.setLeaf(false);
        newRoot.addChild(root);
        splitNode(newRoot, root, 0);
        root = newRoot;
    }
    return insertNotFull(root, new Entry<K, V>(key, value));
}

/**
 * 如果存在给定的键，则更新键关联的值，
 * 否则插入给定的项。
 *
 * @param node  - 非满节点
 * @param entry - 给定的项
 * @return true，如果B树中不存在给定的项，否则false
 */
private V putNotFull(BTreeNode<K, V> node, Entry<K, V> entry)
    assert node.size() < maxKeySize;

    if (node.isLeaf()) // 如果是叶子节点，直接插入
        return node.putEntry(entry);
    else {
        /* 找到entry在给定节点应该插入的位置，那么entry应该插入
         * 该位置对应的子树中
         */
        SearchResult<V> result = node.searchKey(entry.getKey()
        // 如果存在，则更新
        if (result.isExist())
            return node.putEntry(entry);
        BTreeNode<K, V> childNode = node.childAt(result.getInd
        if (childNode.size() == 2 * t - 1) // 如果子节点是满节点
        {
            // 则先分裂
            splitNode(node, childNode, result.getIndex());
            /* 如果给定entry的键大于分裂之后新生成项的键，则需要插入该
             * 否则左边。
             */
            if (compare(entry.getKey(), node.entryAt(result.ge
                childNode = node.childAt(result.getIndex() + 1
        }
        return putNotFull(childNode, entry);
    }
}

/**
 * 如果B树中存在给定的键，则更新值。
```

```
     * 否则插入。
     *
     * @param key   - 键
     * @param value - 值
     * @return 如果B树中存在给定的键，则返回之前的值，否则null
     */
    public V put(K key, V value) {
        if (root.size() == maxKeySize) // 如果根节点满了，则B树长高
        {
            BTreeNode<K, V> newRoot = new BTreeNode<K, V>(kCompara
            newRoot.setLeaf(false);
            newRoot.addChild(root);
            splitNode(newRoot, root, 0);
            root = newRoot;
        }
        return putNotFull(root, new Entry<K, V>(key, value));
    }

    /**
     * 从B树中删除一个与给定键关联的项。
     *
     * @param key - 给定的键
     * @return 如果B树中存在给定键关联的项，则返回删除的项，否则null
     */
    public Entry<K, V> delete(K key) {
        return delete(root, key);
    }

    /**
     * 从以给定<code>node</code>为根的子树中删除与给定键关联的项。
     * <p/>
     * 删除的实现思想请参考《算法导论》第二版的第18章。
     *
     * @param node - 给定的节点
     * @param key  - 给定的键
     * @return 如果B树中存在给定键关联的项，则返回删除的项，否则null
     */
    private Entry<K, V> delete(BTreeNode<K, V> node, K key) {
        // 该过程需要保证，对非根节点执行删除操作时，其关键字个数至少为t。
        assert node.size() >= t || node == root;

        SearchResult<V> result = node.searchKey(key);
        /*
         * 因为这是查找成功的情况，0 <= result.getIndex() <= (node.siz
         * 因此(result.getIndex() + 1)不会溢出。
         */
        if (result.isExist()) {
            // 1.如果关键字在节点node中，并且是叶节点，则直接删除。
            if (node.isLeaf())
                return node.removeEntry(result.getIndex());
            else {
                // 2.a 如果节点node中前于key的子节点包含至少t个项
                BTreeNode<K, V> leftChildNode = node.childAt(resul
                if (leftChildNode.size() >= t) {
                    // 使用leftChildNode中的最后一个项代替node中需要删除
                    node.removeEntry(result.getIndex());
                    node.insertEntry(leftChildNode.entryAt(leftChi
                    // 递归删除左子节点中的最后一个项
                    return delete(leftChildNode, leftChildNode.ent
                } else {
                    // 2.b 如果节点node中后于key的子节点包含至少t个关键字
                    BTreeNode<K, V> rightChildNode = node.childAt(
                    if (rightChildNode.size() >= t) {
                        // 使用rightChildNode中的第一个项代替node中需要
                        node.removeEntry(result.getIndex());
```

```
                    node.insertEntry(rightChildNode.entryAt(0)
                    // 递归删除右子节点中的第一个项
                    return delete(rightChildNode, rightChildNo
                } else // 2.c 前于key和后于key的子节点都只包含t-1个
                {
                    Entry<K, V> deletedEntry = node.removeEntr
                    node.removeChild(result.getIndex() + 1);
                    // 将node中与key关联的项和rightChildNode中的项
                    leftChildNode.addEntry(deletedEntry);
                    for (int i = 0; i < rightChildNode.size();
                        leftChildNode.addEntry(rightChildNode.
                    // 将rightChildNode中的子节点合并进leftChildN
                    if (!rightChildNode.isLeaf()) {
                        for (int i = 0; i <= rightChildNode.si
                            leftChildNode.addChild(rightChildN
                    }
                    return delete(leftChildNode, key);
                }
            }
        }
    } else {
        /*
         * 因为这是查找失败的情况，0 <= result.getIndex() <= node.
         * 因此(result.getIndex() + 1)会溢出。
         */
        if (node.isLeaf()) // 如果关键字不在节点node中，并且是叶节点
        {
            logger.info("The key: " + key + " isn't in this BT
            return null;
        }
        BTreeNode<K, V> childNode = node.childAt(result.getInd
        if (childNode.size() >= t) // // 如果子节点有不少于t个项，
            return delete(childNode, key);
        else // 3
        {
            // 先查找右边的兄弟节点
            BTreeNode<K, V> siblingNode = null;
            int siblingIndex = -1;
            if (result.getIndex() < node.size()) // 存在右兄弟节
            {
                if (node.childAt(result.getIndex() + 1).size()
                    siblingNode = node.childAt(result.getIndex
                    siblingIndex = result.getIndex() + 1;
                }
            }
            // 如果右边的兄弟节点不符合条件，则试试左边的兄弟节点
            if (siblingNode == null) {
                if (result.getIndex() > 0) // 存在左兄弟节点
                {
                    if (node.childAt(result.getIndex() - 1).si
                        siblingNode = node.childAt(result.getI
                        siblingIndex = result.getIndex() - 1;
                    }
                }
            }
            // 3.a 有一个相邻兄弟节点至少包含t个项
            if (siblingNode != null) {
                if (siblingIndex < result.getIndex()) // 左兄弟
                {
                    childNode.insertEntry(node.entryAt(sibling
                    node.removeEntry(siblingIndex);
                    node.insertEntry(siblingNode.entryAt(sibli
                    siblingNode.removeEntry(siblingNode.size()
                    // 将左兄弟节点的最后一个孩子移到childNode
                    if (!siblingNode.isLeaf()) {
```

```
                        childNode.insertChild(siblingNode.chil
                        siblingNode.removeChild(siblingNode.si
                    }
                } else // 右兄弟节点满足条件
                {
                    childNode.insertEntry(node.entryAt(result.
                    node.removeEntry(result.getIndex());
                    node.insertEntry(siblingNode.entryAt(0), r
                    siblingNode.removeEntry(0);
                    // 将右兄弟节点的第一个孩子移到childNode
                    // childNode.insertChild(siblingNode.child
                    if (!siblingNode.isLeaf()) {
                        childNode.addChild(siblingNode.childAt
                        siblingNode.removeChild(0);
                    }
                }
                return delete(childNode, key);
            } else // 3.b 如果其相邻左右节点都包含t-1个项
            {
                if (result.getIndex() < node.size()) // 存在右兄
                {
                    BTreeNode<K, V> rightSiblingNode = node.ch
                    childNode.addEntry(node.entryAt(result.get
                    node.removeEntry(result.getIndex());
                    node.removeChild(result.getIndex() + 1);
                    for (int i = 0; i < rightSiblingNode.size(
                        childNode.addEntry(rightSiblingNode.en
                    if (!rightSiblingNode.isLeaf()) {
                        for (int i = 0; i <= rightSiblingNode.
                            childNode.addChild(rightSiblingNod
                    }
                } else // 存在左节点，在前面插入
                {
                    BTreeNode<K, V> leftSiblingNode = node.chi
                    childNode.insertEntry(node.entryAt(result.
                    node.removeEntry(result.getIndex() - 1);
                    node.removeChild(result.getIndex() - 1);
                    for (int i = leftSiblingNode.size() - 1; i
                        childNode.insertEntry(leftSiblingNode.
                    if (!leftSiblingNode.isLeaf()) {
                        for (int i = leftSiblingNode.size(); i
                            childNode.insertChild(leftSiblingN
                    }
                }
                // 如果node是root并且node不包含任何项了
                if (node == root && node.size() == 0)
                    root = childNode;
                return delete(childNode, key);
            }
        }
    }
}

/**
 * 一个简单的层次遍历B树实现，用于输出B树。
 */
public void output() {
    Queue<BTreeNode<K, V>> queue = new LinkedList<BTreeNode<K,
    queue.offer(root);
    while (!queue.isEmpty()) {
        BTreeNode<K, V> node = queue.poll();
        for (int i = 0; i < node.size(); ++i)
            System.out.print(node.entryAt(i) + " ");
        System.out.println();
        if (!node.isLeaf()) {
```

```
                for (int i = 0; i <= node.size(); ++i)
                    queue.offer(node.childAt(i));
            }
        }
    }

    public static void main(String[] args) {
        Random random = new Random();
        BTree<Integer, Integer> btree = new BTree<Integer, Integer
        List<Integer> save = new ArrayList<Integer>();
        for (int i = 0; i < 10; ++i) {
            int r = random.nextInt(100);
            save.add(r);
            System.out.print(r + "  ");
            btree.insert(r, r);
        }

        System.out.println("----------------------");
        btree.output();
        System.out.println("----------------------");
        btree.delete(save.get(0));
        btree.output();
    }
}
```

Stay hungry，stay foolish！

分类: 数据结构

标签: B树

kosamino
关注 - 32
粉丝 - 198
+加关注

« 上一篇： 二叉树BinaryTree构建测试(无序)
» 下一篇： TreeMap核心源码实现解析

posted on 2019-10-25 04:07 kosamino 阅读(2274) 评论(1) 编辑 收藏 举报

刷新评论 刷新页面 返回顶部