一、基础概念

1. 基本介绍

在TreeMap中，键值对之间按键有序，TreeMap的实现基础是排序二叉树。

2. 构造方法

★ 接口Comparable和Comparator区别

① Comparable位于java.langl包里，绝大多数的类实现了这个接口，比较方法是compareTo。

称为内比较器，可以和自身比较。

② Comparator位于java.util包里，实现Compare接口一般是没有实现Comparable接口或者对compareTo方法不满意，只需要重写compare方法即可。

(1) public TreeMap() {

comparator = null;

} //利用的是键对象自己的排序方法

如果使用默认构造方法，**要求Map中的键实现Comparabe接口**，TreeMap内部进行各种比较时会调用键的Comparable接口中的compareTo方法。

(2) public TreeMap(Comparator<? super K> comparator) {

this.comparator = comparator;

} //利用的是传入的比较器对象

**接受一个比较器对象comparator**，如果comparator不为null，在TreeMap内部进行比较时会调用这个comparator的compare方法，而不再调用键的compareTo方法，也不再要求键实现Comparable接口。

(3) public TreeMap(Map<? extends K, ? extends V> m) {

comparator = null;

putAll(m);

}

接受已有的Map，将其键值对添加到TreeMap中来。依赖的是本身key对象的compareTo方法，将comparator对象设为null。

(4) public TreeMap(SortedMap<K, ? extends V> m) {

comparator = m.comparator();

try {

buildFromSorted(m.size(), m.entrySet().iterator(), null, null);

} catch (java.io.IOException cannotHappen) {

} catch (ClassNotFoundException cannotHappen) {

}

} //接受一个已有的Map，将其键值对添加到TreeMap中来

关于SortedMap接口，扩展了Map接口，表示有序的Map，有一个comparator()方法返回一个比较器对象，TreeMap直接使用该比较器对象的比较方法就可以。

总结： 当传入的值存在comparator对象时，就使用比较器对象。当传入的值没有时，就

需要key对象本身实现了comparable接口，实现了compareTo方法。

3. 比较器对象例子

(1) 默认排序

Map<String, String> map = new TreeMap<>();

map.put("a", "abstract");

map.put("c", "call");

map.put("b", "basic");

map.put("T", "tree");

for(Entry<String,String> kv : map.entrySet()){

System.out.print(kv.getKey()+"="+kv.getValue()+" ");

}

① 输出时为按键排序：

T=tree a=abstract b=basic c=call

② 原因：

String对象中的compareTo方法是大写<小写的。

(2) 传入比较器对象

Map<String, String> map = new TreeMap<>(String.CASE\_INSENSITIVE\_ORDER);

① String类有一个静态成员CASE\_INSENSITIVE\_ORDER，它就是一个忽略大小写的Comparator对象

② 输出为：（按照从小到大自然排序）

a=abstract b=basic c=call T=tree

(3) 传入一个自定义的比较器对象

Map<String, String> map = new TreeMap<>(new Comparator<String>(){

@Override

public int compare(String o1, String o2) {

return o2.compareTo(o1); //自定义了比较方法

}

});

① 正常排序中，compare方法内，是o1.compareTo(o2)，两个对象翻过来，自然就是逆序了，Collections类也有一个静态方法reverseOrder()可以返回一个逆序比较器，如下：

Map<String, String> map = new TreeMap<>(Collections.reverseOrder());

② 输出为：

c=call b=basic a=abstract T=tree

③ 如果是逆序且忽略大小写，可以传入另外一个比较器

Map<String, String> map = new TreeMap<>(

Collections.reverseOrder(String.CASE\_INSENSITIVE\_ORDER));

说明： TreeMap使用键的比较结果对键进行排重，**即使键实际上不同，但只要比较结果相同，它们就会被认为相同，键只会保存一份**。比如传入大小写不一样，但是如果利用了忽略大小写的比较器，就会被认为是一样的，所以只会保存一份。

(4) TreeMap和HashMap构造器用法的区别

① 相同的是，它们都实现了Map接口，都可以按Map进行操作。

② 不同的是，迭代时，TreeMap按键有序，为了实现有序，它要求：要么键实现Comparable

接口，要么创建TreeMap时传递一个Comparator对象。

二、TreeMap底层实现原理（红黑树实现）

1. 基本属性

private final Comparator<? super K> comparator; //比较器对象，如果没传就是null

private transient Entry<K,V> root = null; //指向树的根结点，

private transient int size = 0; //当前键值对的数目

private transient int modCount = 0; //修改次数

1.1 Entry内部类

static final class Entry<K,V> implements Map.Entry<K,V> {

K key; //当前key值

V value; //当前value值

Entry<K,V> left; //指向左子节点

Entry<K,V> right; //指向右子节点

Entry<K,V> parent; //指向父节点

boolean color = BLACK; //color表示颜色，用于红黑树

Entry(K key, V value, Entry<K,V> parent) {

this.key = key;

this.value = value;

this.parent = parent;

}

}

2. 保存键值对方法put()

public V put(K key, V value) {

Entry<K,V> t = root;

//添加第一个节点时，新建一个节点然后指向它

if (t == null) {

//为了检查key的类型，底层是将key强制转换为key，如果key类型不匹配或者为null，或抛出一个null。

compare(key, key);

root = new Entry<>(key, value, null); //此时t=null

size = 1;

modCount++;

return null;

}

int cmp;

Entry<K,V> parent;

//如果不是第一次添加，会执行后面的代码，添加的关键步骤是寻找父节点，找父节点根据是否设置了comparator分为两种情况。

Comparator<? super K> cpr = comparator;

//如果设置了comparator

if (cpr != null) {

do {

parent = t; //parent表示父节点

cmp = cpr.compare(key, t.key); //将插入的值和当前结点比较

if (cmp < 0) //小于0表示小于当前key，就设为左子节点

t = t.left;

else if (cmp > 0)//大于0表示大于当前key，就设为右子节点

t = t.right;

else //等于0表示等于当前key，用新value值覆盖旧value值

return t.setValue(value);

} while (t != null); //等于null就表示没有这个值，新建一个节点

}

else { //如果没有传入comparator对象

if (key == null) //因为compareTo方法要求对象必须不为空

throw new NullPointerException();

@SuppressWarnings("unchecked")

Comparable<? super K> k = (Comparable<? super K>) key;

do {

parent = t;

cmp = k.compareTo(t.key);

if (cmp < 0)

t = t.left;

else if (cmp > 0)

t = t.right;

else

return t.setValue(value);

} while (t != null);

}

//当找不到对应节点时，需要新增一个节点

Entry<K,V> e = new Entry<>(key, value, parent);

//根据比较结果将新建节点设为左孩子和右孩子

if (cmp < 0)

parent.left = e;

else

parent.right = e;

//有一行重要调用fixAfterInsertion(e);，它就是在调整树的结构，使之符合红黑树的约束，保持大致平衡

fixAfterInsertion(e);

size++;

modCount++;

//如果是新建节点，则返回一个null

return null;

}

3. 根据键获取值get()

① public V get(Object key) {

//根据key值获得节点对象，如果没有返回null，如果有返回响应的value值

Entry<K,V> p = getEntry(key);

return (p==null ? null : p.value);

}

② final Entry<K,V> getEntry(Object key) {

//如果存在比较器对象，调用另一个方法

if (comparator != null)

return getEntryUsingComparator(key);

//调用compareTo方法，如果key为null则抛出异常

if (key == null)

throw new NullPointerException();

@SuppressWarnings("unchecked")

Comparable<? super K> k = (Comparable<? super K>) key;

Entry<K,V> p = root; //取得根结点root

while (p != null) {

//根据比较器来获得符合比较器的对象，如果没有则返回null

int cmp = k.compareTo(p.key);

if (cmp < 0)

p = p.left;

else if (cmp > 0)

p = p.right;

else

return p;

}

return null;

}

③ final Entry<K,V> getEntryUsingComparator(Object key) {

@SuppressWarnings("unchecked")

K k = (K) key; //当key值类型出现错误时，会抛出异常

Comparator<? super K> cpr = comparator;

if (cpr != null) {

Entry<K,V> p = root;

while (p != null) {

int cmp = cpr.compare(k, p.key);

if (cmp < 0)

p = p.left;

else if (cmp > 0)

p = p.right;

else

return p;

}

}

return null;

}

3. 包含方法contain()

3.1 判断是否包含key值，containKey()

//根据key值获取结点

public boolean containsKey(Object key) {

return getEntry(key) != null;

}

3.2 判断是否包含value值（遍历二叉树）

public boolean containsValue(Object value) {

//getFirstEntry表示第一个节点，也就是最左边的那个结点

//successor表示后继结点，就是后继算法，再去看一遍。

for (Entry<K,V> e = getFirstEntry(); e != null; e = successor(e))

if (valEquals(value, e.value))

return true;

return false;

}

(1) 后继算法：

① 如果有右孩子(t.right!=null)，则后继为右子树中最小的节点。

② 如果没有右孩子，后继为某祖先节点，从当前节点往上找，如果它是父节点的右孩子，则继续找父节点，直到它不是右孩子或父节点为空，第一个非右孩子节点的父亲节点就是后继节点，如果父节点为空，则后继为null。

4. 根据键删除键值对

public V remove(Object key) {

//先找到对应的节点，如果节点为空则返回空

Entry<K,V> p = getEntry(key);

if (p == null)

return null;

V oldValue = p.value;

//删除对应的节点

deleteEntry(p);

return oldValue;

}

(1) 删除算法：

① 叶子节点：这个容易处理，直接修改父节点对应引用置null即可。

② 只有一个孩子：就是在父亲节点和孩子节点直接建立链接。

③ 有两个孩子：先找到后继，找到后，替换当前节点的内容为后继节点，然后再删除后继节点，因为这个后继节点一定没有左孩子，所以就将两个孩子的情况转换为了前面两种情况。

三、总结

与HashMap相比，TreeMap同样实现了Map接口，但内部使用红黑树实现，红黑树是统计效率比较高的大致平衡的排序二叉树

1. **按键有序**，TreeMap同样实现了SortedMap和NavigableMap接口，可以方便的根据键的顺序进行查找，如第一个、最后一个、某一范围的键、邻近键等。

2. 为了按键有序，TreeMap要求键实现Comparable接口或通过构造方法提供一个Comparator对象。

3. 根据键保存、查找、删除的效率比较高，为O(h)，h为树的高度，在树平衡的情况下，h为log2(N)，N为节点数。

4. 不要求排序，优先考虑HashMap；要求排序，再考虑TreeMap。