**Java中AVL平衡二叉树实现Map (仿照TreeMap和TreeSet)**

**1、下面是AVLTreeMap的实现**

**package** com;

**import** java.io.IOException;

**import** java.util.\*;

**public** **class** AVLTreeMap<K, V> **extends** AbstractMap<K, V> **implements** NavigableMap<K, V>, java.io.Serializable {

**private** **static** **final** **long** *serialVersionUID* = 1731396135957583906L;

**private** **final** Comparator<? **super** K> comparator;

**private** **transient** Entry<K, V> root = **null**;

**private** **transient** **int** size = 0;

**private** **transient** **int** modCount = 0;

**public** AVLTreeMap() {

comparator = **null**;

}

**public** AVLTreeMap(Comparator<? **super** K> comparator) {

**this**.comparator = comparator;

}

**public** AVLTreeMap(Map<? **extends** K, ? **extends** V> m) {

comparator = **null**;

putAll(m);

}

**public** AVLTreeMap(SortedMap<K, ? **extends** V> m) {

comparator = m.comparator();

**try** {

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

} **catch** (IOException e) {

} **catch** (ClassNotFoundException e) {

}

}

**public** **int** size() {

**return** size;

}

**public** **boolean** containsKey(Object key) {

**return** getEntry(key) != **null**;

}

**public** **boolean** containsValue(Object value) {

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

**if** (*valEquals*(value, e.value))

**return** **true**;

}

**return** **false**;

}

**public** V get(Object key) {

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

**return** p == **null** ? **null** : p.value;

}

**public** Comparator<? **super** K> comparator() {

**return** comparator;

}

**public** K firstKey() {

**return** *key*(getFirstEntry());

}

**public** K lastKey() {

**return** *key*(getLastEntry());

}

@SuppressWarnings({ "rawtypes", "unchecked" })

**public** **void** putAll(Map<? **extends** K, ? **extends** V> map) {

**int** mapSize = map.size();

**if** (size == 0 && mapSize != 0 && map **instanceof** SortedMap) {

Comparable<? **super** K> cmp = (Comparable<? **super** K>) ((SortedMap) map).comparator();

**if** (cmp == comparator || (cmp != **null** && cmp.equals(comparator))) {

++modCount;

**try** {

buildFromSorted(mapSize, map.entrySet().iterator(), **null**, **null**);

} **catch** (IOException e) {

} **catch** (ClassNotFoundException e) {

}

**return**;

}

}

**super**.putAll(map);

}

@SuppressWarnings("unchecked")

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

**if** (comparator != **null**) {

**return** getEntryUsingComparator(key);

}

**if** (key == **null**)

**throw** **new** NullPointerException();

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

Entry<K, V> p = root;

**while** (p != **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**;

}

@SuppressWarnings("unchecked")

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

K k = (K) key;

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**;

}

**final** Entry<K, V> getCeilingEntry(Object key) {

Entry<K, V> p = root;

**while** (p != **null**) {

**int** cmp = compare(key, p.key);

**if** (cmp < 0) {

**if** (p.left != **null**)

p = p.left;

**else**

**return** p;

} **else** **if** (cmp < 0) {

**if** (p.right != **null**)

p = p.right;

**else** {

Entry<K, V> parent = p.parent;

Entry<K, V> ch = p;

**while** (parent != **null** && ch == parent.right) {

ch = parent;

parent = parent.parent;

}

**return** parent;

}

} **else** {

**return** p;

}

}

**return** **null**;

}

**final** Entry<K, V> getFloorEntry(Object key) {

Entry<K, V> p = root;

**while** (p != **null**) {

**int** cmp = compare(key, p.key);

**if** (cmp > 0) {

**if** (p.right != **null**)

p = p.right;

**else**

**return** p;

} **else** **if** (cmp < 0) {

**if** (p.left != **null**)

p = p.left;

**else** {

Entry<K, V> parent = p.parent;

Entry<K, V> ch = p;

**while** (parent != **null** && ch == parent.left) {

ch = parent;

parent = parent.parent;

}

**return** parent;

}

} **else** {

**return** p;

}

}

**return** **null**;

}

**final** Entry<K, V> getHigherEntry(Object key) {

Entry<K, V> p = root;

**while** (p != **null**) {

**int** cmp = compare(key, p.key);

**if** (cmp < 0) {

**if** (p.left != **null**)

p = p.left;

**else**

**return** p;

} **else** {

**if** (p.right != **null**)

p = p.right;

**else** {

Entry<K, V> parent = p.parent;

Entry<K, V> ch = p;

**while** (parent != **null** && ch == parent.right) {

ch = parent;

parent = parent.parent;

}

**return** parent;

}

}

}

**return** **null**;

}

**final** Entry<K, V> getLowerEntry(Object key) {

Entry<K, V> p = root;

**while** (p != **null**) {

**int** cmp = compare(key, p.key);

**if** (cmp > 0) {

**if** (p.right != **null**)

p = p.right;

**else**

**return** p;

} **else** {

**if** (p.left != **null**)

p = p.left;

**else** {

Entry<K, V> parent = p.parent;

Entry<K, V> ch = p;

**while** (parent != **null** && ch == parent.left) {

ch = parent;

parent = parent.parent;

}

**return** parent;

}

}

}

**return** **null**;

}

@SuppressWarnings("unchecked")

**public** V put(K key, V value) {

Entry<K, V> t = root;

**if** (t == **null**) {

root = **new** Entry<K, V>(key, value, **null**);

root.height = 1;

size++;

modCount++;

**return** **null**;

}

**int** cmp;

Entry<K, V> parent;

Comparator<? **super** K> cpr = comparator;

**if** (cpr != **null**) {

**do** {

parent = t;

cmp = cpr.compare(key, t.key);

**if** (cmp < 0)

t = t.left;

**else** **if** (cmp > 0)

t = t.right;

**else**

**return** t.setValue(value);

} **while** (t != **null**);

} **else** {

**do** {

parent = t;

cmp = ((Comparable<? **super** K>) key).compareTo(t.key);

**if** (cmp < 0)

t = t.left;

**else** **if** (cmp > 0)

t = t.right;

**else**

**return** t.setValue(value);

} **while** (t != **null**);

}

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

**if** (cmp < 0)

parent.left = e;

**else**

parent.right = e;

fixAfterInsertion(e);

size++;

modCount++;

**return** **null**;

}

**public** V remove(Object key) {

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

**if** (p == **null**) {

**return** **null**;

}

V oldVal = p.value;

deleteEntry(p);

**return** oldVal;

}

**public** **void** clear() {

size = 0;

modCount++;

root = **null**;

}

@SuppressWarnings("unchecked")

**public** Object clone() {

AVLTreeMap<K, V> clone = **null**;

**try** {

clone = (AVLTreeMap<K, V>) **super**.clone();

} **catch** (CloneNotSupportedException e) {

**throw** **new** InternalError();

}

clone.root = **null**;

clone.size = 0;

clone.modCount = 0;

clone.entrySet = **null**;

clone.keySet = **null**;

clone.descendingMap = **null**;

**try** {

clone.buildFromSorted(size, entrySet().iterator(), **null**, **null**);

} **catch** (IOException e) {

} **catch** (ClassNotFoundException e) {

}

**return** clone;

}

**public** Map.Entry<K, V> firstEntry() {

**return** *exportEntry*(getFirstEntry());

}

**public** Map.Entry<K, V> lastEntry() {

**return** *exportEntry*(getLastEntry());

}

**public** Map.Entry<K, V> pollFirstEntry() {

AVLTreeMap.Entry<K, V> p = getFirstEntry();

Map.Entry<K, V> result = *exportEntry*(p);

**if** (p != **null**)

deleteEntry(p);

**return** result;

}

**public** Map.Entry<K, V> pollLastEntry() {

AVLTreeMap.Entry<K, V> e = getLastEntry();

Map.Entry<K, V> result = *exportEntry*(e);

**if** (e != **null**)

deleteEntry(e);

**return** result;

}

**public** Map.Entry<K, V> lowerEntry(K key) {

**return** *exportEntry*(getLowerEntry(key));

}

**public** K lowerKey(K key) {

**return** *keyOrNull*(getLowerEntry(key));

}

**public** Map.Entry<K, V> floorEntry(K key) {

**return** *exportEntry*(getFloorEntry(key));

}

**public** K floorKey(K key) {

**return** *keyOrNull*(getFloorEntry(key));

}

**public** Map.Entry<K, V> ceilingEntry(K key) {

**return** *exportEntry*(getCeilingEntry(key));

}

**public** K ceilingKey(K key) {

**return** *keyOrNull*(getCeilingEntry(key));

}

**public** Map.Entry<K, V> higherEntry(K key) {

**return** *exportEntry*(getHigherEntry(key));

}

**public** K higherKey(K key) {

**return** *keyOrNull*(getHigherEntry(key));

}

**private** **transient** EntrySet entrySet = **null**;

**private** **transient** KeySet<K> navigableKeySet = **null**;

**private** **transient** NavigableMap<K, V> descendingMap = **null**;

@SuppressWarnings("unused")

**private** **transient** Set<K> keySet = **null**;

**private** **transient** Collection<V> values = **null**;

**public** Set<K> keySet() {

**return** navigableKeySet();

}

@SuppressWarnings({ "unchecked", "rawtypes" })

**public** NavigableSet<K> navigableKeySet() {

KeySet<K> ks = navigableKeySet;

**return** (ks != **null**) ? ks : (navigableKeySet = **new** KeySet(**this**));

}

**public** NavigableSet<K> descendingKeySet() {

**return** descendingMap().navigableKeySet();

}

**public** Set<Map.Entry<K, V>> entrySet() {

EntrySet es = entrySet;

**return** (es != **null**) ? es : (entrySet = **new** EntrySet());

}

**public** Collection<V> values() {

Collection<V> vs = values;

**return** (vs != **null**) ? vs : (values = **new** Values());

}

@SuppressWarnings({ "unchecked", "rawtypes" })

**public** NavigableMap<K, V> descendingMap() {

NavigableMap<K, V> km = descendingMap;

**return** (km != **null**) ? km : (descendingMap = **new** DescendingSubMap(**this**, **true**, **null**, **true**, **true**, **null**, **true**));

}

@SuppressWarnings({ "unchecked", "rawtypes" })

**public** NavigableMap<K, V> subMap(K fromKey, **boolean** fromInclusive, K toKey, **boolean** toInclusive) {

**return** **new** AscendingSubMap(**this**, **false**, fromKey, fromInclusive, **false**, toKey, toInclusive);

}

@SuppressWarnings({ "unchecked", "rawtypes" })

**public** NavigableMap<K, V> headMap(K toKey, **boolean** inclusive) {

**return** **new** AscendingSubMap(**this**, **true**, **null**, **true**, **false**, toKey, inclusive);

}

@SuppressWarnings({ "unchecked", "rawtypes" })

**public** NavigableMap<K, V> tailMap(K fromKey, **boolean** inclusive) {

**return** **new** AscendingSubMap(**this**, **false**, fromKey, inclusive, **true**, **null**, **true**);

}

**public** SortedMap<K, V> subMap(K fromKey, K toKey) {

**return** subMap(fromKey, **true**, toKey, **false**);

}

**public** SortedMap<K, V> headMap(K toKey) {

**return** headMap(toKey, **false**);

}

**public** SortedMap<K, V> tailMap(K fromKey) {

**return** tailMap(fromKey, **true**);

}

**class** Values **extends** AbstractCollection<V> {

**public** Iterator<V> iterator() {

**return** **new** ValueIterator(getFirstEntry());

}

**public** **int** size() {

**return** AVLTreeMap.**this**.size();

}

**public** **boolean** contains(Object o) {

**return** AVLTreeMap.**this**.containsValue(o);

}

**public** **boolean** remove(Object o) {

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

**if** (*valEquals*(o, e.value)) {

deleteEntry(e);

**return** **true**;

}

}

**return** **false**;

}

**public** **void** clear() {

AVLTreeMap.**this**.clear();

}

}

**class** EntrySet **extends** AbstractSet<Map.Entry<K, V>> {

**public** Iterator<Map.Entry<K, V>> iterator() {

**return** **new** EntryIterator(getFirstEntry());

}

**public** **int** size() {

**return** AVLTreeMap.**this**.size();

}

@SuppressWarnings("unchecked")

**public** **boolean** contains(Object o) {

**if** (!(o **instanceof** Map.Entry)) {

**return** **false**;

}

Map.Entry<K, V> entry = (Map.Entry<K, V>) o;

V value = entry.getValue();

Entry<K, V> p = getEntry(entry.getKey());

**return** p != **null** && *valEquals*(p.getValue(), value);

}

@SuppressWarnings("unchecked")

**public** **boolean** remove(Object o) {

**if** (!(o **instanceof** Map.Entry)) {

**return** **false**;

}

Map.Entry<K, V> entry = (Map.Entry<K, V>) o;

V value = entry.getValue();

Entry<K, V> p = getEntry(entry.getKey());

**if** (p != **null** && *valEquals*(p.getValue(), value)) {

deleteEntry(p);

**return** **true**;

}

**return** **false**;

}

**public** **void** clear() {

AVLTreeMap.**this**.clear();

}

}

Iterator<K> keyIterator() {

**return** **new** KeyIterator(getFirstEntry());

}

Iterator<K> descendingKeyIterator() {

**return** **new** DescendingKeyIterator(getLastEntry());

}

**static** **final** **class** KeySet<E> **extends** AbstractSet<E> **implements** NavigableSet<E> {

**private** **final** NavigableMap<E, Object> m;

KeySet(NavigableMap<E, Object> m) {

**this**.m = m;

}

@SuppressWarnings({ "unchecked", "rawtypes" })

**public** Iterator<E> iterator() {

**if** (m **instanceof** AVLTreeMap)

**return** ((AVLTreeMap) m).keyIterator();

**else**

**return** (Iterator<E>) (((AVLTreeMap.NavigableSubMap) m).keyIterator());

}

@SuppressWarnings({ "unchecked", "rawtypes" })

**public** Iterator<E> descendingIterator() {

**if** (m **instanceof** AVLTreeMap)

**return** ((AVLTreeMap) m).descendingKeyIterator();

**else**

**return** (Iterator<E>) (((AVLTreeMap.NavigableSubMap) m).descendingKeyIterator());

}

**public** **int** size() {

**return** m.size();

}

**public** **boolean** isEmpty() {

**return** m.isEmpty();

}

**public** **boolean** contains(Object o) {

**return** m.containsKey(o);

}

**public** **void** clear() {

m.clear();

}

**public** E lower(E e) {

**return** m.lowerKey(e);

}

**public** E floor(E e) {

**return** m.floorKey(e);

}

**public** E ceiling(E e) {

**return** m.ceilingKey(e);

}

**public** E higher(E e) {

**return** m.higherKey(e);

}

**public** E first() {

**return** m.firstKey();

}

**public** E last() {

**return** m.lastKey();

}

**public** Comparator<? **super** E> comparator() {

**return** m.comparator();

}

**public** E pollFirst() {

Map.Entry<E, Object> e = m.pollFirstEntry();

**return** e == **null** ? **null** : e.getKey();

}

**public** E pollLast() {

Map.Entry<E, Object> e = m.pollLastEntry();

**return** e == **null** ? **null** : e.getKey();

}

**public** **boolean** remove(Object o) {

**int** oldSize = m.size();

m.remove(o);

**return** oldSize != m.size();

}

**public** NavigableSet<E> descendingSet() {

**return** **new** AVLTreeSet<E>(m.descendingMap());

}

**public** NavigableSet<E> subSet(E fromElement, **boolean** fromInclusive, E toElement, **boolean** toInclusive) {

**return** **new** AVLTreeSet<E>(m.subMap(fromElement, fromInclusive, toElement, toInclusive));

}

**public** NavigableSet<E> headSet(E toElement, **boolean** inclusive) {

**return** **new** AVLTreeSet<E>(m.headMap(toElement, inclusive));

}

**public** NavigableSet<E> tailSet(E fromElement, **boolean** inclusive) {

**return** **new** AVLTreeSet<E>(m.tailMap(fromElement, inclusive));

}

**public** SortedSet<E> subSet(E fromElement, E toElement) {

**return** subSet(fromElement, **true**, toElement, **false**);

}

**public** SortedSet<E> headSet(E toElement) {

**return** headSet(toElement, **false**);

}

**public** SortedSet<E> tailSet(E fromElement) {

**return** tailSet(fromElement, **true**);

}

}

**abstract** **class** PrivateEntryIterator<T> **implements** Iterator<T> {

Entry<K, V> next;

Entry<K, V> lastReturned;

**int** expectedModCount;

PrivateEntryIterator(Entry<K, V> first) {

expectedModCount = modCount;

lastReturned = **null**;

next = first;

}

**public** **final** **boolean** hasNext() {

**return** next != **null**;

}

**final** Entry<K, V> nextEntry() {

Entry<K, V> e = next;

**if** (next == **null**)

**throw** **new** NoSuchElementException();

**if** (modCount != expectedModCount)

**throw** **new** ConcurrentModificationException();

next = *successor*(e);

lastReturned = e;

**return** e;

}

**final** Entry<K, V> prevEntry() {

Entry<K, V> e = next;

**if** (next == **null**)

**throw** **new** NoSuchElementException();

**if** (modCount != expectedModCount)

**throw** **new** ConcurrentModificationException();

next = *predecessor*(e);

lastReturned = e;

**return** e;

}

**public** **void** remove() {

**if** (lastReturned == **null**)

**throw** **new** IllegalStateException();

**if** (modCount != expectedModCount)

**throw** **new** ConcurrentModificationException();

**if** (*leftOf*(lastReturned) != **null** && *rightOf*(lastReturned) != **null**)

next = lastReturned;

deleteEntry(lastReturned);

expectedModCount = modCount;

lastReturned = **null**;

}

}

**final** **class** EntryIterator **extends** PrivateEntryIterator<Map.Entry<K, V>> {

EntryIterator(Entry<K, V> first) {

**super**(first);

}

**public** Map.Entry<K, V> next() {

**return** nextEntry();

}

}

**final** **class** ValueIterator **extends** PrivateEntryIterator<V> {

ValueIterator(Entry<K, V> first) {

**super**(first);

}

**public** V next() {

**return** nextEntry().value;

}

}

**final** **class** KeyIterator **extends** PrivateEntryIterator<K> {

KeyIterator(Entry<K, V> first) {

**super**(first);

}

**public** K next() {

**return** nextEntry().key;

}

}

**final** **class** DescendingKeyIterator **extends** PrivateEntryIterator<K> {

DescendingKeyIterator(Entry<K, V> first) {

**super**(first);

}

**public** K next() {

**return** prevEntry().key;

}

}

@SuppressWarnings("unchecked")

**final** **int** compare(Object k1, Object k2) {

**return** comparator == **null** ? ((Comparable<? **super** K>) k1).compareTo((K) k2) : comparator.compare((K) k1, (K) k2);

}

**final** **static** **boolean** valEquals(Object o1, Object o2) {

**return** o1 == **null** ? o2 == **null** : o1.equals(o2);

}

@SuppressWarnings({ "unchecked", "rawtypes" })

**static** <K, V> Map.Entry<K, V> exportEntry(Entry<K, V> e) {

**return** e == **null** ? **null** : **new** AbstractMap.SimpleImmutableEntry(e);

}

**static** <K, V> K keyOrNull(Entry<K, V> e) {

**return** e == **null** ? **null** : e.key;

}

**static** <K, V> K key(Entry<K, V> e) {

**if** (e == **null**)

**throw** **new** NoSuchElementException();

**return** e.key;

}

**final** **int** max(**int** height1, **int** height2) {

**return** (height1 > height2) ? height1 : height2;

}

**static** **abstract** **class** NavigableSubMap<K, V> **extends** AbstractMap<K, V> **implements** NavigableMap<K, V>, java.io.Serializable {

**private** **static** **final** **long** *serialVersionUID* = 3330238317193227055L;

**final** AVLTreeMap<K, V> m;

**final** K lo, hi;

**final** **boolean** fromStart, toEnd;

**final** **boolean** loInclusive, hiInclusive;

NavigableSubMap(AVLTreeMap<K, V> m, **boolean** fromStart, K lo, **boolean** loInclusive, **boolean** toEnd, K hi, **boolean** hiInclusive) {

**if** (!fromStart && !toEnd) {

**if** (m.compare(lo, hi) > 0)

**throw** **new** IllegalArgumentException("fromKey > toKey");

} **else** {

**if** (!fromStart)

m.compare(lo, lo);

**else**

m.compare(hi, hi);

}

**this**.m = m;

**this**.lo = lo;

**this**.hi = hi;

**this**.fromStart = fromStart;

**this**.toEnd = toEnd;

**this**.loInclusive = loInclusive;

**this**.hiInclusive = hiInclusive;

}

**final** **boolean** tooLow(Object key) {

**if** (!fromStart) {

**int** c = m.compare(key, lo);

**if** (c < 0 || (c == 0 && !loInclusive))

**return** **true**;

}

**return** **false**;

}

**final** **boolean** tooHigh(Object key) {

**if** (!toEnd) {

**int** c = m.compare(key, hi);

**if** (c > 0 || (c == 0 && !hiInclusive))

**return** **true**;

}

**return** **false**;

}

**final** **boolean** inRange(Object key) {

**return** !tooLow(key) && !tooHigh(key);

}

**final** **boolean** inClosedRange(Object key) {

**return** (fromStart || m.compare(key, lo) >= 0) && (toEnd || m.compare(key, hi) <= 0);

}

**final** **boolean** inRange(Object key, **boolean** inclusive) {

**return** inclusive ? inRange(key) : inClosedRange(key);

}

**final** AVLTreeMap.Entry<K, V> absLowest() {

AVLTreeMap.Entry<K, V> e = fromStart ? m.getFirstEntry() : (loInclusive ? m.getCeilingEntry(lo) : m.getHigherEntry(lo));

**return** (e == **null** || tooHigh(e.key)) ? **null** : e;

}

**final** AVLTreeMap.Entry<K, V> absHighest() {

AVLTreeMap.Entry<K, V> e = toEnd ? m.getLastEntry() : (hiInclusive ? m.getFloorEntry(hi) : m.getLowerEntry(hi));

**return** (e == **null** || tooLow(e.key) ? **null** : e);

}

**final** AVLTreeMap.Entry<K, V> absCeiling(K key) {

**if** (tooLow(key))

**return** absLowest();

AVLTreeMap.Entry<K, V> e = m.getCeilingEntry(key);

**return** (e == **null** || tooHigh(e.key)) ? **null** : e;

}

**final** AVLTreeMap.Entry<K, V> absHigher(K key) {

**if** (tooLow(key))

**return** absLowest();

AVLTreeMap.Entry<K, V> e = m.getHigherEntry(key);

**return** (e == **null** || tooHigh(e.key)) ? **null** : e;

}

**final** AVLTreeMap.Entry<K, V> absFloor(K key) {

**if** (tooHigh(key))

**return** absHighest();

AVLTreeMap.Entry<K, V> e = m.getFloorEntry(key);

**return** (e == **null** || tooLow(e.key)) ? **null** : e;

}

**final** AVLTreeMap.Entry<K, V> absLower(K key) {

**if** (tooHigh(key))

**return** absHighest();

AVLTreeMap.Entry<K, V> e = m.getLowerEntry(key);

**return** (e == **null** || tooLow(e.key)) ? **null** : e;

}

**final** AVLTreeMap.Entry<K, V> absHighFence() {

**return** toEnd ? **null** : (hiInclusive ? m.getHigherEntry(hi) : m.getCeilingEntry(hi));

}

**final** AVLTreeMap.Entry<K, V> absLowFence() {

**return** fromStart ? **null** : (loInclusive ? m.getLowerEntry(lo) : m.getCeilingEntry(lo));

}

**abstract** AVLTreeMap.Entry<K, V> subLowest();

**abstract** AVLTreeMap.Entry<K, V> subHighest();

**abstract** AVLTreeMap.Entry<K, V> subCeiling(K key);

**abstract** AVLTreeMap.Entry<K, V> subHigher(K key);

**abstract** AVLTreeMap.Entry<K, V> subFloor(K key);

**abstract** AVLTreeMap.Entry<K, V> subLower(K key);

**abstract** Iterator<K> keyIterator();

**abstract** Iterator<K> descendingKeyIterator();

**public** **boolean** isEmpty() {

**return** (fromStart && toEnd) ? m.isEmpty() : entrySet().isEmpty();

}

**public** **int** size() {

**return** (fromStart && toEnd) ? m.size() : entrySet().size();

}

**public** **final** **boolean** containsKey(Object key) {

**return** inRange(key) && m.containsKey(key);

}

**public** **final** V put(K key, V value) {

**if** (!inRange(key))

**throw** **new** IllegalArgumentException("key out of range");

**return** m.put(key, value);

}

**public** **final** V get(Object key) {

**return** !inRange(key) ? **null** : m.get(key);

}

**public** **final** V remove(Object key) {

**return** !inRange(key) ? **null** : m.remove(key);

}

**public** Map.Entry<K, V> ceilingEntry(K key) {

**return** *exportEntry*(subCeiling(key));

}

**public** K ceilingKey(K key) {

**return** *keyOrNull*(subCeiling(key));

}

**public** Map.Entry<K, V> higherEntry(K key) {

**return** *exportEntry*(subHigher(key));

}

**public** K higherKey(K key) {

**return** *keyOrNull*(subHigher(key));

}

**public** Map.Entry<K, V> floorEntry(K key) {

**return** *exportEntry*(subFloor(key));

}

**public** K floorKey(K key) {

**return** *keyOrNull*(subFloor(key));

}

**public** Map.Entry<K, V> lowerEntry(K key) {

**return** *exportEntry*(subLower(key));

}

**public** K lowerKey(K key) {

**return** *keyOrNull*(subLower(key));

}

**public** K firstKey() {

**return** *key*(subLowest());

}

**public** K lastKey() {

**return** *key*(subHighest());

}

**public** Map.Entry<K, V> firstEntry() {

**return** *exportEntry*(subLowest());

}

**public** Map.Entry<K, V> lastEntry() {

**return** *exportEntry*(subHighest());

}

**public** Map.Entry<K, V> pollFirstEntry() {

AVLTreeMap.Entry<K, V> e = subLowest();

Map.Entry<K, V> result = *exportEntry*(e);

**if** (e != **null**)

m.deleteEntry(e);

**return** result;

}

**public** Map.Entry<K, V> pollLastEntry() {

AVLTreeMap.Entry<K, V> e = subHighest();

Map.Entry<K, V> result = *exportEntry*(e);

**if** (e != **null**)

m.deleteEntry(e);

**return** result;

}

**transient** NavigableMap<K, V> descendingMapView = **null**;

**transient** EntrySetView entrySetView = **null**;

**transient** KeySet<K> navigableKeySetView = **null**;

@SuppressWarnings({ "unchecked", "rawtypes" })

**public** **final** NavigableSet<K> navigableKeySet() {

KeySet<K> nksv = navigableKeySetView;

**return** nksv != **null** ? nksv : (navigableKeySetView = **new** AVLTreeMap.KeySet(**this**));

}

**public** **final** Set<K> keySet() {

**return** navigableKeySet();

}

**public** NavigableSet<K> descendingKeySet() {

**return** descendingMap().navigableKeySet();

}

**public** SortedMap<K, V> subMap(K fromKey, K toKey) {

**return** subMap(fromKey, **true**, toKey, **false**);

}

**public** SortedMap<K, V> headMap(K toKey) {

**return** headMap(toKey, **false**);

}

**public** SortedMap<K, V> tailMap(K fromKey) {

**return** tailMap(fromKey, **true**);

}

**abstract** **class** EntrySetView **extends** AbstractSet<Map.Entry<K, V>> {

**private** **transient** **int** size = -1, sizeModCount;

**public** **int** size() {

**if** (fromStart && toEnd)

**return** m.size();

**if** (size == -1 || sizeModCount != m.modCount) {

size = 0;

Iterator<Map.Entry<K, V>> i = iterator();

**while** (i.hasNext()) {

size++;

i.next();

}

}

**return** size;

}

**public** **boolean** isEmpty() {

AVLTreeMap.Entry<K, V> e = absLowest();

**return** (e == **null** || tooHigh(e));

}

@SuppressWarnings("unchecked")

**public** **boolean** contains(Object o) {

**if** (!(o **instanceof** Map.Entry))

**return** **false**;

Map.Entry<K, V> entry = (Map.Entry<K, V>) o;

K key = entry.getKey();

**if** (!inRange(key))

**return** **false**;

AVLTreeMap.Entry<K, V> node = m.getEntry(key);

**return** node != **null** && *valEquals*(node.getValue(), entry.getValue());

}

@SuppressWarnings("unchecked")

**public** **boolean** remove(Object o) {

**if** (!(o **instanceof** Map.Entry))

**return** **false**;

Map.Entry<K, V> entry = (Map.Entry<K, V>) o;

K key = entry.getKey();

**if** (!inRange(key))

**return** **false**;

AVLTreeMap.Entry<K, V> node = m.getEntry(key);

**if** (node != **null** && *valEquals*(node.getValue(), entry.getValue())) {

m.deleteEntry(node);

**return** **true**;

}

**return** **false**;

}

}

**abstract** **class** SubMapIterator<T> **implements** Iterator<T> {

AVLTreeMap.Entry<K, V> lastReturned;

AVLTreeMap.Entry<K, V> next;

**final** K fenceKey;

**int** expectedModCount;

SubMapIterator(AVLTreeMap.Entry<K, V> first, AVLTreeMap.Entry<K, V> fence) {

expectedModCount = m.modCount;

lastReturned = **null**;

next = first;

fenceKey = (fence == **null** ? **null** : fence.key);

}

**public** **boolean** hasNext() {

**return** (next != **null**) && (next.key != fenceKey);

}

**final** AVLTreeMap.Entry<K, V> nextEntry() {

AVLTreeMap.Entry<K, V> e = next;

**if** (next == **null**)

**throw** **new** NoSuchElementException();

**if** (expectedModCount != m.modCount)

**throw** **new** ConcurrentModificationException();

next = *successor*(e);

lastReturned = e;

**return** e;

}

**final** AVLTreeMap.Entry<K, V> prevEntry() {

AVLTreeMap.Entry<K, V> e = next;

**if** (next == **null**)

**throw** **new** NoSuchElementException();

**if** (expectedModCount != m.modCount)

**throw** **new** ConcurrentModificationException();

next = *predecessor*(e);

lastReturned = e;

**return** e;

}

**final** **void** removeAscending() {

**if** (lastReturned == **null**)

**throw** **new** IllegalStateException();

**if** (expectedModCount != m.modCount)

**throw** **new** ConcurrentModificationException();

**if** (lastReturned.left != **null** && lastReturned.right != **null**)

next = lastReturned;

m.deleteEntry(lastReturned);

lastReturned = **null**;

expectedModCount = m.modCount;

}

**final** **void** removeDescending() {

**if** (lastReturned == **null**)

**throw** **new** IllegalStateException();

**if** (expectedModCount != m.modCount)

**throw** **new** ConcurrentModificationException();

m.deleteEntry(lastReturned);

lastReturned = **null**;

expectedModCount = m.modCount;

}

}

**final** **class** SubMapEntryIterator **extends** SubMapIterator<Map.Entry<K, V>> {

SubMapEntryIterator(AVLTreeMap.Entry<K, V> first, AVLTreeMap.Entry<K, V> fence) {

**super**(first, fence);

}

**public** Map.Entry<K, V> next() {

**return** nextEntry();

}

**public** **void** remove() {

removeAscending();

}

}

**final** **class** SubMapKeyIterator **extends** SubMapIterator<K> {

SubMapKeyIterator(AVLTreeMap.Entry<K, V> first, AVLTreeMap.Entry<K, V> fence) {

**super**(first, fence);

}

**public** K next() {

**return** nextEntry().key;

}

**public** **void** remove() {

removeAscending();

}

}

**final** **class** DescendingSubMapEntryIterator **extends** SubMapIterator<Map.Entry<K, V>> {

DescendingSubMapEntryIterator(AVLTreeMap.Entry<K, V> first, AVLTreeMap.Entry<K, V> fence) {

**super**(first, fence);

}

**public** Map.Entry<K, V> next() {

**return** prevEntry();

}

**public** **void** remove() {

removeDescending();

}

}

**final** **class** DescendingSubMapKeyIterator **extends** SubMapIterator<K> {

DescendingSubMapKeyIterator(AVLTreeMap.Entry<K, V> first, AVLTreeMap.Entry<K, V> fence) {

**super**(first, fence);

}

**public** K next() {

**return** prevEntry().key;

}

**public** **void** remove() {

removeDescending();

}

}

}

**static** **final** **class** AscendingSubMap<K, V> **extends** NavigableSubMap<K, V> {

**private** **static** **final** **long** *serialVersionUID* = 8576741467287943049L;

AscendingSubMap(AVLTreeMap<K, V> m, **boolean** fromStart, K lo, **boolean** loInclusive, **boolean** toEnd, K hi, **boolean** hiInclusive) {

**super**(m, fromStart, lo, loInclusive, toEnd, hi, hiInclusive);

}

**public** Comparator<? **super** K> comparator() {

**return** m.comparator();

}

**public** NavigableMap<K, V> subMap(K fromKey, **boolean** fromInclusive, K toKey, **boolean** toInclusive) {

**if** (!inRange(fromKey, fromInclusive))

**throw** **new** IllegalArgumentException("fromKey out of range");

**if** (!inRange(toKey, toInclusive))

**throw** **new** IllegalArgumentException("toKey out of range");

**return** **new** AscendingSubMap<K, V>(m, **false**, fromKey, fromInclusive, **false**, toKey, toInclusive);

}

**public** NavigableMap<K, V> headMap(K toKey, **boolean** inclusive) {

**if** (!inRange(toKey, inclusive))

**throw** **new** IllegalArgumentException("toKey out of range");

**return** **new** AscendingSubMap<K, V>(m, fromStart, lo, loInclusive, **false**, toKey, inclusive);

}

**public** NavigableMap<K, V> tailMap(K fromKey, **boolean** inclusive) {

**if** (!inRange(fromKey, inclusive))

**throw** **new** IllegalArgumentException("fromKey out of range");

**return** **new** AscendingSubMap<K, V>(m, **false**, fromKey, inclusive, toEnd, hi, hiInclusive);

}

**public** NavigableMap<K, V> descendingMap() {

NavigableMap<K, V> mv = descendingMapView;

**return** (mv != **null**) ? mv : (descendingMapView = **new** DescendingSubMap<K, V>(m, fromStart, lo, loInclusive, toEnd, hi, hiInclusive));

}

Iterator<K> keyIterator() {

**return** **new** SubMapKeyIterator(absLowest(), absHighFence());

}

Iterator<K> descendingKeyIterator() {

**return** **new** DescendingSubMapKeyIterator(absHighest(), absLowFence());

}

**final** **class** AscendingEntrySetView **extends** EntrySetView {

**public** Iterator<Map.Entry<K, V>> iterator() {

**return** **new** SubMapEntryIterator(absLowest(), absHighFence());

}

}

**public** Set<Map.Entry<K, V>> entrySet() {

EntrySetView es = entrySetView;

**return** (es != **null**) ? es : **new** AscendingEntrySetView();

}

AVLTreeMap.Entry<K, V> subLowest() {

**return** absLowest();

}

AVLTreeMap.Entry<K, V> subHighest() {

**return** absHighest();

}

AVLTreeMap.Entry<K, V> subCeiling(K key) {

**return** absCeiling(key);

}

AVLTreeMap.Entry<K, V> subHigher(K key) {

**return** absHigher(key);

}

AVLTreeMap.Entry<K, V> subFloor(K key) {

**return** absFloor(key);

}

AVLTreeMap.Entry<K, V> subLower(K key) {

**return** absLower(key);

}

}

**static** **final** **class** DescendingSubMap<K, V> **extends** NavigableSubMap<K, V> {

**private** **static** **final** **long** *serialVersionUID* = -5614587065744400071L;

DescendingSubMap(AVLTreeMap<K, V> m, **boolean** fromStart, K lo, **boolean** loInclusive, **boolean** toEnd, K hi, **boolean** hiInclusive) {

**super**(m, fromStart, lo, loInclusive, toEnd, hi, hiInclusive);

}

**public** Comparator<? **super** K> comparator() {

**return** m.comparator();

}

**public** NavigableMap<K, V> subMap(K fromKey, **boolean** fromInclusive, K toKey, **boolean** toInclusive) {

**if** (!inRange(fromKey, fromInclusive))

**throw** **new** IllegalArgumentException("fromKey out of range");

**if** (!inRange(toKey, toInclusive))

**throw** **new** IllegalArgumentException("toKey out of range");

**return** **new** DescendingSubMap<K, V>(m, **false**, toKey, toInclusive, **false**, fromKey, fromInclusive);

}

**public** NavigableMap<K, V> headMap(K toKey, **boolean** inclusive) {

**if** (!inRange(toKey, inclusive))

**throw** **new** IllegalArgumentException("toKey out of range");

**return** **new** DescendingSubMap<K, V>(m, **false**, toKey, inclusive, fromStart, lo, loInclusive);

}

**public** NavigableMap<K, V> tailMap(K fromKey, **boolean** inclusive) {

**if** (!inRange(fromKey, inclusive))

**throw** **new** IllegalArgumentException("fromKey out of range");

**return** **new** DescendingSubMap<K, V>(m, fromStart, lo, loInclusive, **false**, fromKey, inclusive);

}

**public** NavigableMap<K, V> descendingMap() {

NavigableMap<K, V> mv = descendingMapView;

**return** (mv != **null**) ? mv : (descendingMapView = **new** AscendingSubMap<K, V>(m, fromStart, lo, loInclusive, toEnd, hi, hiInclusive));

}

Iterator<K> keyIterator() {

**return** **new** DescendingSubMapKeyIterator(absHighest(), absLowFence());

}

Iterator<K> descendingKeyIterator() {

**return** **new** SubMapKeyIterator(absLowest(), absHighFence());

}

**final** **class** DescendingEntrySetView **extends** EntrySetView {

**public** Iterator<Map.Entry<K, V>> iterator() {

**return** **new** DescendingSubMapEntryIterator(absHighest(), absLowFence());

}

}

**public** Set<Map.Entry<K, V>> entrySet() {

EntrySetView es = entrySetView;

**return** es != **null** ? es : **new** DescendingEntrySetView();

}

AVLTreeMap.Entry<K, V> subLowest() {

**return** absHighest();

}

AVLTreeMap.Entry<K, V> subHighest() {

**return** absLowest();

}

AVLTreeMap.Entry<K, V> subCeiling(K key) {

**return** absFloor(key);

}

AVLTreeMap.Entry<K, V> subHigher(K key) {

**return** absLower(key);

}

AVLTreeMap.Entry<K, V> subFloor(K key) {

**return** absCeiling(key);

}

AVLTreeMap.Entry<K, V> subLower(K key) {

**return** absHigher(key);

}

}

@SuppressWarnings("unused")

**private** **class** SubMap **extends** AbstractMap<K, V> **implements** SortedMap<K, V>, java.io.Serializable {

**private** **static** **final** **long** *serialVersionUID* = 3420410162963921747L;

**private** **boolean** fromStart = **false**, toEnd = **false**;

**private** K fromKey, toKey;

**private** Object readResolve() {

**return** **new** AscendingSubMap<K, V>(AVLTreeMap.**this**, fromStart, fromKey, **true**, toEnd, toKey, **true**);

}

**public** Set<Map.Entry<K, V>> entrySet() {

**throw** **new** InternalError();

}

**public** K lastKey() {

**throw** **new** InternalError();

}

**public** K firstKey() {

**throw** **new** InternalError();

}

**public** SortedMap<K, V> subMap(K fromKey, K toKey) {

**throw** **new** InternalError();

}

**public** SortedMap<K, V> headMap(K toKey) {

**throw** **new** InternalError();

}

**public** SortedMap<K, V> tailMap(K fromKey) {

**throw** **new** InternalError();

}

**public** Comparator<? **super** K> comparator() {

**throw** **new** InternalError();

}

}

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

K key;

V value;

Entry<K, V> left;

Entry<K, V> right;

Entry<K, V> parent;

**int** height = 1; // 节点的高度

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

**this**.key = key;

**this**.value = value;

**this**.parent = parent;

}

**public** K getKey() {

**return** key;

}

**public** V getValue() {

**return** value;

}

**public** V setValue(V value) {

V oldVal = **this**.value;

**this**.value = value;

**return** oldVal;

}

@SuppressWarnings("unchecked")

**public** **boolean** equals(Object o) {

**if** (!(o **instanceof** Map.Entry))

**return** **false**;

Map.Entry<K, V> e = (Map.Entry<K, V>) o;

**return** *valEquals*(key, e.getKey()) && *valEquals*(value, e.getValue());

}

**public** **int** hashCode() {

**int** keyHash = (key == **null**) ? 0 : key.hashCode();

**int** valueHash = (value == **null**) ? 0 : value.hashCode();

**return** keyHash ^ valueHash;

}

**public** String toString() {

**return** key + "=" + value;

}

}

**final** Entry<K, V> getFirstEntry() {

Entry<K, V> p = root;

**if** (p != **null**)

**while** (p.left != **null**)

p = p.left;

**return** p;

}

**final** Entry<K, V> getLastEntry() {

Entry<K, V> p = root;

**if** (p != **null**)

**while** (p.right != **null**)

p = p.right;

**return** p;

}

**static** <K, V> Entry<K, V> successor(Entry<K, V> t) {

**if** (t == **null**) {

**return** **null**;

} **else** **if** (t.right != **null**) {

Entry<K, V> p = t.right;

**while** (p.left != **null**)

p = p.left;

**return** p;

} **else** {

Entry<K, V> p = t.parent;

Entry<K, V> ch = t;

**while** (p != **null** && ch == p.right) {

ch = p;

p = p.parent;

}

**return** p;

}

}

**static** <K, V> Entry<K, V> predecessor(Entry<K, V> t) {

**if** (t == **null**) {

**return** **null**;

} **else** **if** (t.left != **null**) {

Entry<K, V> p = t.left;

**while** (p.right != **null**)

p = p.right;

**return** p;

} **else** {

Entry<K, V> p = t.parent;

Entry<K, V> ch = t;

**while** (p != **null** && ch == p.left) {

ch = p;

p = p.parent;

}

**return** p;

}

}

**private** **static** <K, V> **int** heightOf(Entry<K, V> e) {

**return** (e == **null** ? 0 : e.height);

}

**private** **static** <K, V> Entry<K, V> parentOf(Entry<K, V> e) {

**return** (e == **null** ? **null** : e.parent);

}

**private** **void** setHeight(Entry<K, V> e, **int** height) {

**if** (e != **null**)

e.height = height;

}

**private** **static** <K, V> Entry<K, V> leftOf(Entry<K, V> e) {

**return** (e == **null** ? **null** : e.left);

}

**private** **static** <K, V> Entry<K, V> rightOf(Entry<K, V> e) {

**return** (e == **null** ? **null** : e.right);

}

**private** **void** rotateLeft(Entry<K, V> p) {

**if** (p != **null**) {

Entry<K, V> r = p.right;

p.right = r.left;

**if** (r.left != **null**)

r.left.parent = p;

r.parent = p.parent;

**if** (p.parent == **null**)

root = r;

**else** **if** (p.parent.left == p)

p.parent.left = r;

**else**

p.parent.right = r;

r.left = p;

p.parent = r;

// 重新设置节点的高度

setHeight(p, max(*heightOf*(*leftOf*(p)), *heightOf*(*rightOf*(p))) + 1);

setHeight(r, max(*heightOf*(*leftOf*(r)), *heightOf*(*rightOf*(r))) + 1);

}

}

**private** **void** rotateRight(Entry<K, V> p) {

**if** (p != **null**) {

Entry<K, V> l = p.left;

p.left = l.right;

**if** (l.right != **null**)

l.right.parent = p;

l.parent = p.parent;

**if** (p.parent == **null**)

root = l;

**else** **if** (p.parent.right == p)

p.parent.right = l;

**else**

p.parent.left = l;

l.right = p;

p.parent = l;

// 重新设置节点的高度

setHeight(p, max(*heightOf*(*leftOf*(p)), *heightOf*(*rightOf*(p))) + 1);

setHeight(l, max(*heightOf*(*leftOf*(l)), *heightOf*(*rightOf*(l))) + 1);

}

}

// 此方法最多旋转两次

**private** **void** fixAfterInsertion(Entry<K, V> x) {

x.height = 1;

**while** (x.parent != **null**) {

Entry<K, V> parent = *parentOf*(x);

**if** (x == *leftOf*(parent)) {

Entry<K, V> sib = *rightOf*(parent);

**if** (*heightOf*(x) == *heightOf*(sib)) {

**break**;// x=root;

} **else** **if** (*heightOf*(x) - *heightOf*(sib) > 1) {

**if** (*heightOf*(*rightOf*(x)) > *heightOf*(*leftOf*(x))) {

rotateLeft(x);

}

rotateRight(parent);

// x = parentOf(parent);

**break**;// x=root;

} **else** {

setHeight(parent, max(*heightOf*(x), *heightOf*(sib)) + 1);

x = parent;

}

} **else** {

Entry<K, V> sib = *leftOf*(parent);

**if** (*heightOf*(x) == *heightOf*(sib)) {

**break**;

} **else** **if** (*heightOf*(x) - *heightOf*(sib) > 1) {

**if** (*heightOf*(*leftOf*(x)) > *heightOf*(*rightOf*(x))) {

rotateRight(x);

}

rotateLeft(parent);

// x = parentOf(parent);

**break**;// x=root;

} **else** {

setHeight(parent, max(*heightOf*(x), *heightOf*(sib)) + 1);

x = parent;

}

}

}

}

**private** **void** deleteEntry(Entry<K, V> p) {

modCount++;

size--;

**if** (p.left != **null** && p.right != **null**) {

Entry<K, V> s = *successor*(p);

p.key = s.key;

p.value = s.value;

p = s;

}

Entry<K, V> replacement = p.left != **null** ? p.left : p.right;

**if** (replacement != **null**) {

replacement.parent = p.parent;

**if** (p.parent == **null**)

root = replacement;

**else** **if** (p == *leftOf*(*parentOf*(p)))

p.parent.left = replacement;

**else**

p.parent.right = replacement;

p.left = p.right = p.parent = **null**;

**if** (replacement.parent != **null**)

fixAfterDeletion(replacement.parent);

} **else** **if** (p == root) {// 删除叶子节点，但此叶子节点是跟节点

root = **null**;

} **else** { // 删除叶子节点，但此叶子节点不是跟节点

Entry<K, V> parent = p.parent;

**if** (p == *leftOf*(parent)) {

p.parent.left = **null**;

**if** (*heightOf*(*rightOf*(parent)) != 1)

fixAfterDeletion(parent);

} **else** {

p.parent.right = **null**;

**if** (*heightOf*(*leftOf*(parent)) != 1)

fixAfterDeletion(parent);

}

}

}

// 此方法最多旋转N=(h-1)次,h为树的高度

**private** **void** fixAfterDeletion(Entry<K, V> x) {

**while** (x != **null**) {

Entry<K, V> left = x.left;

Entry<K, V> right = x.right;

**if** ((*heightOf*(left) - *heightOf*(right) == 1) || (*heightOf*(right) - *heightOf*(left) == 1)) {

**break**;

} **else** **if** (*heightOf*(left) - *heightOf*(right) > 1) {

**if** (*heightOf*(*rightOf*(left)) > *heightOf*(*leftOf*(left))) {

rotateLeft(left);

}

rotateRight(x);

x = x.parent; // 由于rotateRight(x),还原x节点的指针

} **else** **if** (*heightOf*(right) - *heightOf*(left) > 1) {

**if** (*heightOf*(*leftOf*(right)) > *heightOf*(*rightOf*(right))) {

rotateRight(right);

}

rotateLeft(x);

x = x.parent;// 由于rotateLeft(x),还原x节点的指针

} **else** {

setHeight(x, max(*heightOf*(left), *heightOf*(right)) + 1);

}

x = x.parent;

}

}

**private** **void** writeObject(java.io.ObjectOutputStream s) **throws** IOException {

s.defaultWriteObject();

s.writeInt(size);

**for** (Iterator<Map.Entry<K, V>> i = entrySet().iterator(); i.hasNext();) {

Map.Entry<K, V> e = i.next();

s.writeObject(e.getKey());

s.writeObject(e.getValue());

}

}

**private** **void** readObject(**final** java.io.ObjectInputStream s) **throws** IOException, ClassNotFoundException {

s.defaultReadObject();

**int** size = s.readInt();

buildFromSorted(size, **null**, s, **null**);

}

**void** readTreeSet(**int** size, java.io.ObjectInputStream s, V defaultVal) **throws** IOException, ClassNotFoundException {

buildFromSorted(size, **null**, s, defaultVal);

}

**void** addAllForTreeSet(SortedSet<? **extends** K> set, V defaultVal) {

**try** {

buildFromSorted(size, set.iterator(), **null**, defaultVal);

} **catch** (IOException cannotHappen) {

} **catch** (ClassNotFoundException cannotHappen) {

}

}

@SuppressWarnings({ "rawtypes" })

**private** **void** buildFromSorted(**int** size, Iterator it, java.io.ObjectInputStream str, V defaultVal) **throws** IOException, ClassNotFoundException {

**this**.size = size;

root = buildFromSorted(0, size - 1, *computeRootHeight*(size), it, str, defaultVal);

}

@SuppressWarnings({ "unchecked", "rawtypes" })

**private** **final** Entry<K, V> buildFromSorted(**int** lo, **int** hi, **int** height, Iterator it, java.io.ObjectInputStream str, V defaultVal) **throws** IOException, ClassNotFoundException {

**if** (hi < lo)

**return** **null**;

**int** mid = (lo + hi) / 2;

Entry<K, V> left = **null**;

**if** (lo < mid) {

left = buildFromSorted(lo, mid - 1, height - 1, it, str, defaultVal);

}

K key;

V value;

**if** (it != **null**) {

**if** (defaultVal == **null**) {

Map.Entry<K, V> e = (Map.Entry<K, V>) it.next();

key = e.getKey();

value = e.getValue();

} **else** {

key = (K) it.next();

value = defaultVal;

}

} **else** {

key = (K) str.readObject();

value = (V) (defaultVal != **null** ? defaultVal : str.readObject());

}

Entry<K, V> middle = **new** Entry<K, V>(key, value, **null**);

middle.height = height;

**if** (left != **null**) {

middle.left = left;

left.parent = middle;

}

**if** (mid < hi) {

Entry<K, V> right = buildFromSorted(mid + 1, hi, height - 1, it, str, defaultVal);

middle.right = right;

right.parent = middle;

}

**return** middle;

}

**private** **static** **int** computeRootHeight(**int** size) {

**int** height = 0;

**for** (**int** m = size; m > 0; m = m / 2)

height++;

**return** height;

}

}

**2、下面是AVLTreeSet的实现**

**package** com;

**import** java.io.IOException;

**import** java.util.\*;

**public** **class** AVLTreeSet<E> **extends** AbstractSet<E> **implements** NavigableSet<E>, Cloneable, java.io.Serializable {

**private** **static** **final** **long** *serialVersionUID* = 5711293308239082950L;

**private** **transient** NavigableMap<E, Object> m;

**private** **static** **final** Object *PRESENT* = **new** Object();

AVLTreeSet(NavigableMap<E, Object> m) {

**this**.m = m;

}

**public** AVLTreeSet() {

**this**(**new** AVLTreeMap<E, Object>());

}

**public** AVLTreeSet(Comparator<? **super** E> comparator) {

**this**(**new** AVLTreeMap<E, Object>(comparator));

}

**public** AVLTreeSet(Collection<? **extends** E> c) {

**this**();

addAll(c);

}

**public** AVLTreeSet(SortedSet<E> s) {

**this**(s.comparator());

addAll(s);

}

**public** Iterator<E> iterator() {

**return** m.navigableKeySet().iterator();

}

**public** Iterator<E> descendingIterator() {

**return** m.descendingKeySet().iterator();

}

**public** NavigableSet<E> descendingSet() {

**return** **new** AVLTreeSet<E>(m.descendingMap());

}

**public** **int** size() {

**return** m.size();

}

**public** **boolean** isEmpty() {

**return** m.isEmpty();

}

**public** **boolean** contains(Object o) {

**return** m.containsKey(o);

}

**public** **boolean** add(E e) {

**return** m.put(e, *PRESENT*) == **null**;

}

**public** **boolean** remove(Object o) {

**return** m.remove(o) == *PRESENT*;

}

**public** **void** clear() {

m.clear();

}

@SuppressWarnings("unchecked")

**public** **boolean** addAll(Collection<? **extends** E> c) {

**if** (m.size() == 0 && c.size() != 0 && c **instanceof** SortedSet && m **instanceof** AVLTreeMap) {

SortedSet<? **extends** E> set = (SortedSet<? **extends** E>) c;

AVLTreeMap<E, Object> map = (AVLTreeMap<E, Object>) m;

Comparator<? **super** E> cc = (Comparator<? **super** E>) set.comparator();

Comparator<? **super** E> mc = map.comparator();

**if** (cc == mc || (cc != **null** && cc.equals(mc))) {

map.addAllForTreeSet(set, *PRESENT*);

**return** **true**;

}

}

**return** **super**.addAll(c);

}

**public** NavigableSet<E> subSet(E fromElement, **boolean** fromInclusive, E toElement, **boolean** toInclusive) {

**return** **new** AVLTreeSet<E>(m.subMap(fromElement, fromInclusive, toElement, toInclusive));

}

**public** NavigableSet<E> headSet(E toElement, **boolean** inclusive) {

**return** **new** AVLTreeSet<E>(m.headMap(toElement, inclusive));

}

**public** NavigableSet<E> tailSet(E fromElement, **boolean** inclusive) {

**return** **new** AVLTreeSet<E>(m.tailMap(fromElement, inclusive));

}

**public** SortedSet<E> subSet(E fromElement, E toElement) {

**return** subSet(fromElement, **true**, toElement, **false**);

}

**public** SortedSet<E> headSet(E toElement) {

**return** headSet(toElement, **false**);

}

**public** SortedSet<E> tailSet(E fromElement) {

**return** tailSet(fromElement, **true**);

}

**public** Comparator<? **super** E> comparator() {

**return** m.comparator();

}

**public** E first() {

**return** m.firstKey();

}

**public** E last() {

**return** m.lastKey();

}

**public** E lower(E e) {

**return** m.lowerKey(e);

}

**public** E floor(E e) {

**return** m.floorKey(e);

}

**public** E ceiling(E e) {

**return** m.ceilingKey(e);

}

**public** E higher(E e) {

**return** m.higherKey(e);

}

**public** E pollFirst() {

Map.Entry<E, Object> e = m.pollFirstEntry();

**return** e == **null** ? **null** : e.getKey();

}

**public** E pollLast() {

Map.Entry<E, Object> e = m.pollLastEntry();

**return** e == **null** ? **null** : e.getKey();

}

@SuppressWarnings("unchecked")

**public** Object clone() {

AVLTreeSet<E> clone = **null**;

**try** {

clone = (AVLTreeSet<E>) **super**.clone();

} **catch** (CloneNotSupportedException e) {

**throw** **new** InternalError();

}

clone.m = **new** AVLTreeMap<E, Object>(m);

**return** clone;

}

**private** **void** writeObject(java.io.ObjectOutputStream s) **throws** IOException {

s.defaultWriteObject();

s.writeObject(m.comparator());

s.writeInt(m.size());

**for** (Iterator<E> i = m.keySet().iterator(); i.hasNext();) {

s.writeObject(i.next());

}

}

@SuppressWarnings("unchecked")

**private** **void** readObject(java.io.ObjectInputStream s) **throws** IOException, ClassNotFoundException {

s.defaultReadObject();

Comparator<? **super** E> c = (Comparator<? **super** E>) s.readObject();

AVLTreeMap<E, Object> tm;

**if** (c == **null**) {

tm = **new** AVLTreeMap<E, Object>();

} **else** {

tm = **new** AVLTreeMap<E, Object>(c);

}

**int** size = s.readInt();

tm.readTreeSet(size, s, *PRESENT*);

}

}