```
package projectFinal;

import java.util.*;

public class SkipListSet<T extends Comparable<T>> implements SortedSet<T> {

    // Private internal class, skip list set item
    private class SkipListSetItem<T> {
        T value;
        List<SkipListSetItem<T>> next; // list of next pointers, ie each index of
list differnt level of next

        SkipListSetItem(T value, int height) {
            this.value = value;
            this.next = new ArrayList<>(Collections.nCopies(height, null));
        }

        int height() {
            return next.size();
        }
    }

    // Private internal class, skip list set iterator
    private class SkipListSetIterator implements Iterator<T> {
        private SkipListSetItem<T> current;       // Tracks the current node during
iteration
        private SkipListSetItem<T> lastReturned; // Tracks the last node returned
by `next()`
        private boolean canRemove;               // Tracks if `remove()` is valid
to call

        SkipListSetIterator() {
            this.current = head; // Start iteration at the head of the skip list
            this.lastReturned = null; // No node has been returned yet
            this.canRemove = false; // Initially, `remove()` cannot be called
        }

        @Override
        public boolean hasNext() {
            return current.next.get(0) != null;
        }

        @Override
        public T next() {
            if (!hasNext()) {
                throw new NoSuchElementException("No more elements in the skip
list.");
            }
            lastReturned = current.next.get(0); // Save the node being returned
            current = current.next.get(0); // Advance to the next node
            canRemove = true; // Mark `remove()` as valid after `next()` is called
            return lastReturned.value; // Return the value of the node
        }

        @Override
        public void remove() {
            if (!canRemove) {
                throw new IllegalStateException("Remove can only be called after
next() and only once per next() call.");
```

```
            }
            if (lastReturned == null) {
                throw new IllegalStateException("No element to remove. Call next()
first.");
            }

            SkipListSet.this.remove(lastReturned.value); // Delegate removal to
SkipListSet
            canRemove = false; // Mark `remove()` as invalid until `next()` is
called again
            lastReturned = null; // Clear the last returned node to prevent
multiple removals
        }
    }
```

```
    private static final double PROBABILITY = 0.5;
    private SkipListSetItem<T> head;
    private int maxHeight;
    private int size;

    // constructor no argument
    public SkipListSet() {
        this.head = new SkipListSetItem<>(null, 1);
        this.maxHeight = 1;
        this.size = 0;
    }

    // constructor with collection as argument
    public SkipListSet(Collection<? extends T> collection) {
        this();
        addAll(collection);
    }

    @Override
    public Iterator<T> iterator() {
        return new SkipListSetIterator();
    }

    // Randomly determines the height of a new node based on the set probability
    private int randomHeight() {
        int height = 1;
        while (Math.random() < PROBABILITY && height < maxHeight + 1) {
            height++;
        }
        return height;
    }

    @Override
    public boolean add(T value) {
        List<SkipListSetItem<T>> update = new
ArrayList<>(Collections.nCopies(maxHeight, null));
        SkipListSetItem<T> current = head;
```

```
        // if list empty, just add value with height 1
        if (current == null) {
            SkipListSetItem<T> newNode = new SkipListSetItem<>(value, 1);
            current = newNode;
            return true;
        }

        // Traverse the skip list to find the insertion point
        for (int i = maxHeight - 1; i >= 0; i--) {
            while (current.next.get(i) != null &&
current.next.get(i).value.compareTo(value) < 0) {
                current = current.next.get(i);
            }
            update.set(i, current);
        }

        current = current.next.get(0); // Move to the bottom level

        // Check if the value already exists
        if (current != null && current.value.equals(value)) {
            return false;
        }

        // Increment height if we reach 2^maxHeight items
        if (size + 1 >= (1 << maxHeight)) {  // Check if size + 1 == 2^maxHeight
            maxHeight++;
            head.next.add(null);  // Increase the height of the head node
            update.add(head);
        }

        // Determine the height of the new node
        int nodeHeight = randomHeight();
        if (nodeHeight > maxHeight) {
            nodeHeight = maxHeight; // Cap node height to maxHeight
        }

        // Create the new node and update pointers
        SkipListSetItem<T> newNode = new SkipListSetItem<>(value, nodeHeight);
        for (int i = 0; i < nodeHeight; i++) {
            newNode.next.set(i, update.get(i).next.get(i));
            update.get(i).next.set(i, newNode);
        }

        size++;
        return true;
    }


    @Override
    public boolean remove(Object o) {
        // Validate input
        if (o == null || !(o instanceof Comparable)) {
            return false;
        }

        T value = (T) o;
        List<SkipListSetItem<T>> update = new
ArrayList<>(Collections.nCopies(maxHeight, null));
        SkipListSetItem<T> current = head;
```

```
        // Traverse the skip list to locate the node to be removed
        for (int i = maxHeight - 1; i >= 0; i--) {
            while (current.next.get(i) != null &&
current.next.get(i).value.compareTo(value) < 0) {
                current = current.next.get(i);
            }
            update.set(i, current);
        }

        current = current.next.get(0); // Move to the bottom level

        // Node not found
        if (current == null || !current.value.equals(value)) {
            return false;
        }

        // Update pointers to bypass the node being removed
        for (int i = 0; i < current.height(); i++) {
            if (update.get(i).next.get(i) != current) {
                break;
            }
            update.get(i).next.set(i, current.next.get(i));
        }

        size--; // Decrease the size of the skip list

        // Adjust maxHeight dynamically
        while (maxHeight > 2 && size < (1 << (maxHeight - 1))) { // Check if size <
2^(maxHeight-1)
            head.next.remove(maxHeight - 1); // Remove the topmost level
            maxHeight--; // Reduce the maxHeight
        }

        return true;
    }



    public void printList() {
        if (head == null) {
            return; // Skip list is empty
        }

        for (int i = maxHeight - 1; i >= 0; i--) { // Start from the topmost level
            SkipListSetItem<T> current = head; // Start at the head for this level
            System.out.print((i + 1) + ":   "); // Label for the level

            // Traverse and print all nodes at the current level
            while (current != null) {
                if (current.value != null) { // Skip printing head (if it doesn't
store a value)
                    System.out.print(current.value + " ");
                }
                current = current.next.get(i); // Move to the next node at this
level
            }

            System.out.println(); // New line for the next level
```

```
        }
    }


    public void reBalance() {
        List<T> values = new ArrayList<>();
        SkipListSetItem<T> current = head.next.get(0);

        // Collect all values
        while (current != null) {
            values.add(current.value);
            current = current.next.get(0);
        }

        // Clear the skip list
        clear();

        // Re-insert all values with new randomized heights
        for (T value : values) {
            add(value);
        }
    }


    @Override
    public boolean contains(Object o) {
        if (o == null) {
            throw new NullPointerException("Cannot search for null in the skip
list.");
        }

        if (!(o instanceof Comparable)) {
            throw new ClassCastException("Object must implement Comparable to be
searched in the skip list.");
        }

        T value = (T) o;
        SkipListSetItem<T> current = head;

        // Traverse from the top level to the bottom level
        for (int i = maxHeight - 1; i >= 0; i--) {
            while (current.next.get(i) != null &&
current.next.get(i).value.compareTo(value) < 0) {
                current = current.next.get(i);
            }
        }

        current = current.next.get(0); // Move to the bottom level

        // Return true if the value matches
        return current != null && current.value.equals(value);
    }

    @Override
    public int size() {
        return size;
    }

    @Override
```

```
public void clear() {
    head = new SkipListSetItem<>(null, 1);
    maxHeight = 1;
    size = 0;
}

@Override
public Comparator<? super T> comparator() {
    return null;
}

@Override
public SortedSet<T> subSet(T fromElement, T toElement) {
    throw new UnsupportedOperationException();
}

@Override
public SortedSet<T> headSet(T toElement) {
    throw new UnsupportedOperationException();
}

@Override
public SortedSet<T> tailSet(T fromElement) {
    throw new UnsupportedOperationException();
}

@Override
public T first() {
    if (size == 0) {
        throw new NoSuchElementException();
    }
    return head.next.get(0).value;
}

@Override
public T last() {
    if (size == 0) {
        throw new NoSuchElementException();
    }
    SkipListSetItem<T> current = head;
    for (int i = maxHeight - 1; i >= 0; i--) {
        while (current.next.get(i) != null) {
            current = current.next.get(i);
        }
    }
    return current.value;
}

@Override
public Object[] toArray() {
    Object[] array = new Object[size];
    int index = 0;
    for (T value : this) {
        array[index++] = value;
    }
    return array;
}

@Override
```

```
    public <E> E[] toArray(E[] a) {
        if (a.length < size) {
            a = (E[])
java.lang.reflect.Array.newInstance(a.getClass().getComponentType(), size);
        }
        int index = 0;
        Object[] result = a;
        for (T value : this) {
            result[index++] = value;
        }
        if (a.length > size) {
            a[size] = null;
        }
        return a;
    }

    @Override
    public boolean containsAll(Collection<?> c) {
        for (Object o : c) {
            if (!contains(o)) {
                return false;
            }
        }
        return true;
    }

    @Override
    public boolean addAll(Collection<? extends T> c) {
        boolean modified = false;
        for (T item : c) {
            if (add(item)) {
                modified = true;
            }
        }
        return modified;
    }

    @Override
    public boolean retainAll(Collection<?> c) {
        boolean modified = false;
        Iterator<T> it = iterator();
        while (it.hasNext()) {
            if (!c.contains(it.next())) {
                it.remove();
                modified = true;
            }
        }
        return modified;
    }

    @Override
    public boolean removeAll(Collection<?> c) {
        boolean modified = false;
        for (Object o : c) {
            if (remove(o)) {
                modified = true;
            }
        }
        return modified;
```

```
    }

    @Override
    public boolean isEmpty() {
        return size == 0;
    }

    @Override
    public int hashCode() {
        int hashCode = 0;
        for (T value : this) {
            hashCode += value.hashCode();
        }
        return hashCode;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (!(obj instanceof SkipListSet)) {
            return false;
        }
        SkipListSet<?> other = (SkipListSet<?>) obj;
        if (size != other.size) {
            return false;
        }
        Iterator<T> it1 = iterator();
        Iterator<?> it2 = other.iterator();
        while (it1.hasNext() && it2.hasNext()) {
            T value1 = it1.next();
            Object value2 = it2.next();
            if (!value1.equals(value2)) {
                return false;
            }
        }
        return !it1.hasNext() && !it2.hasNext();
    }
}
```