Lösung Immutable TreeSet

Ein TreeSet wird typischerweise als Binärer Suchbaum implementiert. Es lässt sich wie folgt definieren: TreeSet<E> = Nil | Branch(TreeSet<E> left, E elem, TreeSet<E> right)

In Prosa: Ein TreeSet ist entweder ein leerer Baum (Nil) oder es ist eine Verzweigung (Branch) welcher aus drei Teilen besteht: Einem linken Teilbaum, dem Element und dem rechten Teilbaum.

Entsprechend ist die Lösung Implementiert:

Nil

```
private static class Nil<E extends Comparable<E>> extends TreeSet<E> {
    public TreeSet<E> insert(E e) { return new Branch<E>(this, e, this); }
    public boolean contains(E e) { return false; }
    public boolean isEmpty() { return true;}
}
```

Nil gibt bei *contains* Aufrufen immer *false* zurück, da es ja leer ist. Bei *isEmpty* gibt es immer *true* zurück. Wenn in *Nil* ein Element eingefügt werden soll, wird ein neuer *Branch* erzeugt, bei dem der linke und der rechte Teilbaum *Nil* sind und als Element das Argument e übergeben wird.

Branch

```
private static class Branch<E extends Comparable<E>> extends TreeSet<E> {
    private final E e;
    private final TreeSet<E> left, right;

    public Branch(TreeSet<E> left, E e, TreeSet<E> right) {
        this.left = left; this.e = e; this.right = right;
    }
    ...
}
```

Die *Branch* Klasse nimmt im Konstruktor den linken, den rechten Teilbaum und das Element *e*. Alle Felder sind *final* markiert.

isEmpty

```
public boolean isEmpty() { return false; }
```

Die Methode *isEmpty* gibt immer *false* zurück.

contains

```
public boolean contains(final E elem) {
    final int cmp = e.compareTo(elem);
    final boolean result;
    if (cmp == 0) { result = true; }
    else if (cmp < 0) { result = right.contains(elem); }
    else { result = left.contains(elem); }
    return result;
}
```

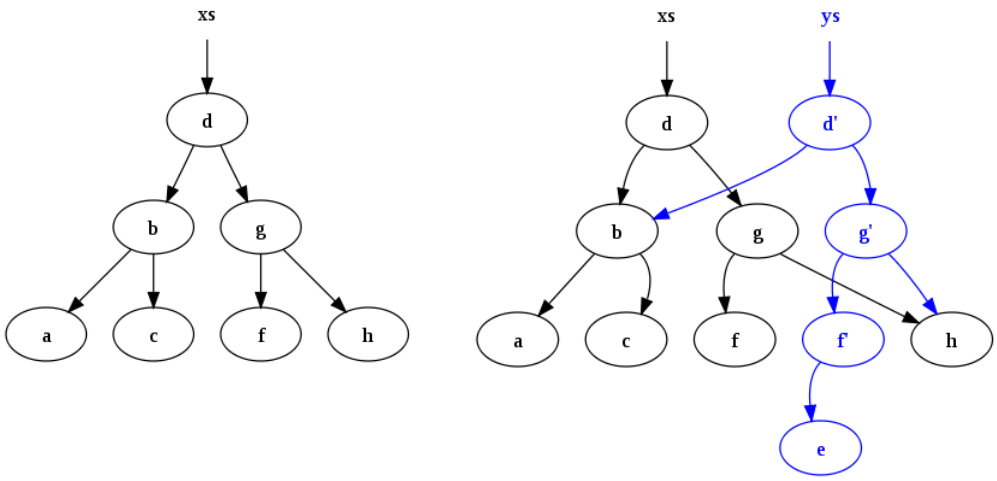
Die *contains* Methode vergleicht das zu prüfende Argument *elem* mit seinem Element *e*. Wenn das Resultat 0 ist, wurde das *elem* gefunden. Wenn das Resultat des Vergleiches negativ ist, bedeutet dies, dass das gesuchte Element *elem* grösser ist als das aktuelle Element e dieses *Branch* Objektes. D

Entsprechend muss im rechten Teilbaum weitergesucht werden. Ist das Vergleichsresultat positiv, muss im linken Teilbaum weitergesucht werden.
Wenn das gesuchte Element nicht im Set ist, wird man schlussendlich auf einem *Nil* Objekt landen, das dann *false* zurück gibt.

`insert`

```
public TreeSet<E> insert(final E elem) {
    final int cmp = e.compareTo(elem);
    final TreeSet<E> result;
    if (cmp == 0) { result = this; }
    else if (cmp < 0) { result = new Branch<E>(left, e, right.insert(elem)); }
    else { result = new Branch<E>(left.insert(elem), e, right); }
    return result;
}
```

Die *insert* Methode hat eine ähnliche Struktur wie die *contains* Methode. Sie vergleicht ebenfalls das Argument mit dem aktuellen Element *e*. Wenn das Resultat 0 ist, befinden wir uns in einem *Branch*, der das einzufügende Element bereits enthält.
Wenn das Resultat des Vergleichs negativ ist, bedeutet dies, dass das einzufügende Element *elem* rekursiv im rechten Teilbaum eingefügt werden muss. Da sich durch das Einfügen der rechte Teilbaum verändern wird, muss das aktuelle Branch Objekt durch eine neue Instanz ersetzt werden. Der linke Teilbaum *left* und das aktuelle Element *e* bleiben dabei gleich. Der rechte Teilbaum jedoch wird ersetzt durch jenen Baum, der durch das Einfügen des Elements *elem* in den rechten Teilbaum entsteht. Analoges gilt für den linken Teilbaum.
Diese Strategie nennt man Path-Copying, da dabei nicht der ganze Baum sondern nur der Pfad zum gesuchten Element kopiert werden muss. Wenn ein Element rechts eingefügt werden muss, kann der ganze linke Teilbaum unverändert bleiben und zwischen dem ursprünglichen Baum und dem neuen Baum geteilt werden (Structural Sharing). Die zwei folgenden Bilder[1] Illustrieren die Situation.



Nachteil: Falls ein Element in den Baum eingefügt ist das bereits enthalten ist, so wird trotzdem eine Kopie erstellt.

[1] Quelle: http://en.wikipedia.org/wiki/Persistent_data_structure