Abgabetermin: SA in der KW 22

Gr. 1, E. Pitzer **Name** _____ **Aufwand in h** _____
Gr. 2, F. Gruber-Leitner

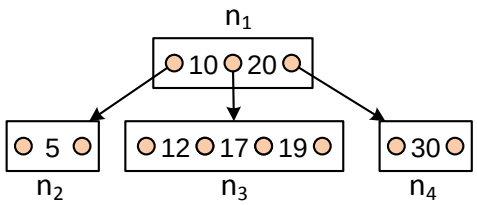
Punkte _____ **Kurzzeichen Tutor / Übungsleiter** _____ / _____

2-3-4-Bäume (6 + 18 Punkte)

Mengen *(sets)* und Wörterbücher *(dictionaries oder maps)* sind in der Praxis häufig benötigte Behälterklassen. Sie sind daher auch in jedem ernst zu nehmenden Behälter-Framework enthalten (so auch im JDK). Sollen die Elemente in sortierter Reihenfolge gehalten werden, werden zur Realisierung dieser Behältertypen meistens binäre Suchbäume eingesetzt. Die in der Übung behandelte Implementierung eines binären Suchbaums hat leider den Nachteil, dass der Baum zu einer linearen Liste entarten kann. Das hat zur Konsequenz, dass alle Operationen auf dem Suchbaum nicht mehr logarithmische, sondern lineare Laufzeitkomplexität aufweisen.

Diesem Problem kann man beikommen, indem man den Suchbaum bei jeder Einfüge- und Löschoperation ausbalanziert. Ein Baum ist balanziert, wenn der linke und der rechte Unterbaum im Wesentlichen dieselbe Höhe aufweisen und diese Eigenschaft auch für die Unterbäume der Unterbäume gilt.

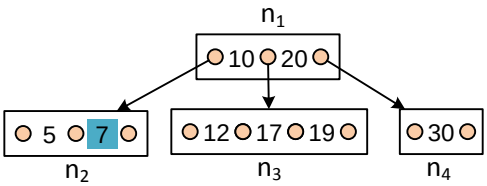
Mit so genannten 2-3-4-Bäumen lassen sich alle Baumoperationen so realisieren, dass der Baum immer ausbalanziert bleibt. Im Gegensatz zu Binärbäumen, bei denen jeder Knoten zwei Zeiger auf die Nachfolgerknoten aufweisen kann, können 2-3-4-Bäume Knoten mit zwei, drei oder vier Zeigern auf Nachfolgerknoten besitzen (siehe nachfolgende Abbildung).



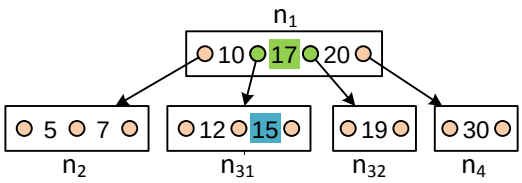
In jedem Knoten des Baums werden bis zu drei aufsteigend sortierte Schlüssel gespeichert. Damit in derartigen Bäumen effizient gesucht werden kann, sind die Elemente in Unterbäumen eines Knotens folgendermaßen angeordnet: Alle Schlüssel im ersten Unterbaum (jener, welcher am weitesten links liegt) sind kleiner als er erste Schlüsselwert, alle Schlüssel im zweiten Unterbaum sind größer oder gleich wie der erste, aber kleiner als der zweite Schlüssel, usw.

Die Suche nach einem Element in einem 2-3-4-Baum kann daher folgendermaßen implementiert werden: Zunächst wird in den Schlüsseln des Wurzelknotens nach dem Element gesucht. Wird dieses hier nicht gefunden, wird ermittelt, zwischen welchen Schlüsselwerten sich das Element befindet und die Suche beim entsprechenden Nachfolgerknoten fortgesetzt. Dies wird so lange wiederholt bis man das Element gefunden hat oder die Suche erfolglos bei einem Blatt des Baumes abgebrochen werden muss.

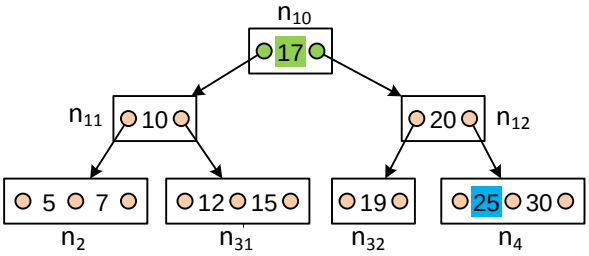
Das Einfügen eines neuen Elements gestaltet sich hingegen etwas komplizierter. Da neue Elemente nur in Blättern eingefügt werden, muss zunächst mit der oben beschriebenen Suchstrategie jenes Blatt bestimmt werden, in welches das Element gehört. In dieses Blatt wird das Element sortiert eingefügt, sofern es sich beim Blatt um einen 2- oder 3-Knoten handelt. Will man beispielsweise in den obigen Baum das Element 7 einfügen, so kann dieses problemlos in den Knoten n_2 aufgenommen werden:



Handelt es sich hingegen beim betroffenen Blatt um einen 4-Knoten, muss dieser Knoten vor dem Einfügen in zwei 2-Knoten aufgespalten werden. Dazu wird der mittlere der drei Schlüssel im Vorgängerknoten eingefügt und aus dem linken und rechten Schlüssel zwei 2-Knoten gebildet, die an den passenden Stellen in den Vorgängerknoten gehängt werden. Ein Beispiel soll dieses Vorgehen verdeutlichen. Will man in obigen Baum das Element 15 einfügen, so muss dies im Knoten n_3 erfolgen. Da dieser bereits vollständig aufgefüllt ist, wird der mittlere Schlüssel 17 in den Vorgängerknoten n_1 verschoben und n_3 in n_{31} und n_{32} zerlegt. Anschließend wird 15 in n_{31} eingefügt.



Durch das Aufteilen eines Knotens und dem damit verbundenen Einfügen eines neuen Wertes in den Vorgängerknoten könnte auch dieser überlaufen. Um dies von Vornherein zu verhindern, werden beim Durchwandern des Baums von der Wurzel bis zum Blatt, in das eingefügt werden soll, alle angetroffenen 4-Knoten aufgeteilt. Soll beispielsweise im obigen Baum 25 eingefügt werden, muss zunächst der Wurzelknoten n_1 in die 2-Knoten n_{11} und n_{12} geteilt werden. Da der Wurzelknoten keine Vorgänger hat, muss für den mittleren Schlüssel ein neuer Wurzelknoten n_{10} geschaffen werden:



Ihre Aufgabe ist es nun, zwei Implementierungen für das Interface `SortedTreeSet<T>` sowie deren Basisinterfaces `SortedSet<T>` und `Iterable<T>` zu erstellen:

```
package swe4.collections;

public interface SortedSet<T> extends Iterable<T> {
  boolean        add(T elem);      // Fügt elem in den Set ein, falls elem noch nicht
                                   // im Set enthalten war. In diesem Fall wird
                                   // true zurückgegeben. Sonst false.
  T              get(T elem);      // Gibt eine Referenz auf das Element im Set
                                   // zurück das gleich zu elem ist und null, wenn
                                   // ein derartiges Element nicht existiert.
  boolean        contains(T elem); // Gibt zurück, ob ein zu elem gleiches Element
                                   // im Set existiert.
  int            size();           // Gibt die Anzahl der Element im Set zurück.
  T              first();          // Gibt das kleinstes Element im Set zurück.
  T              last();           // Gibt das größtes Element im Set zurück.
  Comparator<T>  comparator();     // Liefert den Comparator oder null, wenn
                                   // „natürliche Sortierung“ verwendet wird.
  Iterator<T>    iterator();
}

public interface SortedTreeSet<T> extends SortedSet<T> {
  int height();                    // Gibt die Höhe des Baums zurück.
}
```

Beachten Sie, dass Implementierungen von `SortedSet<T>` Mengen im mathematischen Sinne realisieren, d. h., dass gleiche Elemente nur einmal in der Menge enthalten sein dürfen. Die Methode `add()` gibt daher auch zurück, ob ein Element eingefügt worden ist (`true`) oder ob es sich bereits in der Menge befunden hat (`false`).

a) Implementieren Sie zunächst die Klasse `BSTSet<T>`, welche die gegebenen Interfaces in Form eines binären Suchbaums realisiert. Passen Sie dazu den in der Übung erstellten binären Suchbaums so an, dass die angeführten Anforderungen erfüllt sind und ergänzen Sie die noch fehlenden Operationen.

b) Implementieren Sie die Klasse `TwoThreeFourTreeSet<T>` unter Verwendung eines 2-3-4-Baums als interne Datenstruktur.

Die beiden Klassen müssen einen Standard-Konstruktor und einen Konstruktor, an den ein Vergleichsobjekt übergeben werden kann, das `java.util.Comparator<T>` implementiert, zur Verfügung stellen. Wird ein Vergleichsobjekt übergeben, wird dieses zum Vergleichen von Elementen herangezogen. Ist kein Vergleichsobjekt vorhanden, wird angenommen, dass die eingefügten Elemente das Interface `Comparable<T>` unterstützen und der Vergleich auf dieser Basis durchgeführt („natürliche Sortierung“).

Testen Sie Ihre Implementierung ausführlich. Auf der Lernplattform stehen Ihnen die Klassen `TwoThreeFourTreeSetTest`, `TwoThreeFourTreeSetTest` und ihre Basisklasse `SortedTreeSetTestBase` zur Verfügung, die Unittests enthalten, welche die Korrektheit Ihrer Implementierung überprüfen. Ihre Implementierung muss diese Tests bestehen. Erweitern Sie die Testsuite um zumindest 10 weitere sinnvolle Testfälle, die sich signifikant von den bestehenden Tests unterscheiden.