title: "Set (Binärbaum)" permalink: /03-tree-set/ mathjax: true

Set: Menge (ohne Duplikate)

Letzte Woche hatten wir die Liste als sequenzielle Datenstruktur behandelt. Diese speichert Daten in der Reihenfolge ab, in der sie hinzugefügt werden:

```
IntList xs = new IntListImpl();

xs.add(1);
xs.add(3);
xs.add(3);

System.out.println(xs); // Ausgabe: "[1, 3, 3]"
```

Das Beispiel zeigt aber auch, dass Duplikate enthalten sein können -- was je nach Anwendung allerdings unerwünscht sein kann.

Angenommen, wir möchten nun z.B. herausfinden, welche Buchstaben (`char`s) in einem langen `String` vorkommen. Hätten wir eine Datenstruktur, welche Duplikate einfach unterbindet, so könnten wir folgendes machen:

```
CharSet cs = new CharSetImpl();

String str = "In Ulm und um Ulm und um Ulm herum";

for (char c : str.toLowerCase().toCharArray()) {
        if (c == ' ') continue;  // überspringe Leerzeichen
        cs.add(c)
}

System.out.println(cs);
// gewünschte Ausgabe:
// [i, n, u, l, m, d, h, e, r]
```

Ein *Set* soll also jedes Element nur genau einmal enthalten, auch wenn man versucht es wieder einzufügen.

Das Interface dafür könnte so aussehen:

```
interface CharSet {
        void add(char c);          // Element hinzufügen
        boolean contains(char c);  // prüfen ob bereits enthalten
        char remove(char c);       // Element entfernen
        int size();                // nicht "length", da keine Sequenz!
}
```

> Hinweis: Da ein Set keine Ordnung hat, also alle Elemente "einfach so" darin liegen ohne Index-Nummern, gibt es statt der `length` eine `size` Methode. Daher übergibt man der `remove` Methode auch keinen Index, sondern einen Wert!

Innere Klassen

Wenn wir ein Set nun analog zur Liste implementieren, so brauchen wir wieder eine Hilfsklasse, welche die eigentlichen Daten speichert und die Datenstruktur aufspannt. Da die Klasse spezifisch für diese Struktur und Implementierung ist, kann sie als *innere* Klasse angelegt werden:

```
class CharSetImpl1 implements CharSet {
        class Element {
                char value;
                Element next;
                Element(char c, Element n) {
                        value = c;
                        next = n;
                }
        }

        private Element head;
        // ...
}
```

Innere Klassen sind in Java...

- sinnvoll, wenn sie ausschließlich innerhalb einer Klasse, also lokal verwendet werden.
- normal oder `static` definiert; normale innere Klassen können auf die Variablen der äußeren Instanz zugreifen, *statische* innere Klassen können nur auf statische Variablen und Methoden der äußeren Klasse zugreifen.
- mit Sichtbarkeiten versehen - so wie Variablen und Methoden: *package* (kein Schlüsselwort), `private`, `public`.

Duplikate Vermeiden

Wir können nun Duplikate vermeiden, indem wir vor dem Einfügen prüfen, ob ein Element schon enthalten ist. Dazu beginnen wir vorne und hangeln uns bis hinten durch, wobei wir jedes Element auf (Wert-)Gleichheit prüfen.

```
class CharSetImpl1 implements CharSet {
        // ...
        public boolean contains(char c) {
                if (head == null)
                        return false;

                Element it = head;
                while (it != null) {
```

```
                        if (it.value == c)
                                return true;
                        it = it.next;
                }

                return false;
        }
        // ...
}
```

Das Einfügen (`add`) kann dann analog zur Liste realisiert werden, ebenso die Methoden `size`, `remove` und `toString` (siehe ch02.ListImpl3 bzw. ch03.CharListImpl1).

```
class CharSetImpl1 implements CharSet {
        // ...
        public void add(char c) {
                // nicht einfügen, wenn bereits enthalten!
                if (contains(c)) {
                        return;
                }

                // weiter wie bei Liste...
        }
        // ...
}
```

Wir wollen an dieser Stelle aber garnicht zu ausführlich werden, da die Implementierung des Sets als Liste hochgrading ineffizient ist: bei jedem `add` bzw. `contains` muss zuerst die gesamte Liste durchlaufen werden, um zu prüfen ob das Element nicht bereits enthalten ist. Betrachtet man also den *Aufwand* dieser Methoden in O-Notation, so ist dieser linear in der Anzahl der enthaltenen Elemente: $O(n)$. Das heißt praktisch gesehen: haben wir *doppelt so viele* Elemente im Set, so dauert jeder Aufruf auch *doppelt so lang*.

Das geht natürlich besser:

Binärbaum

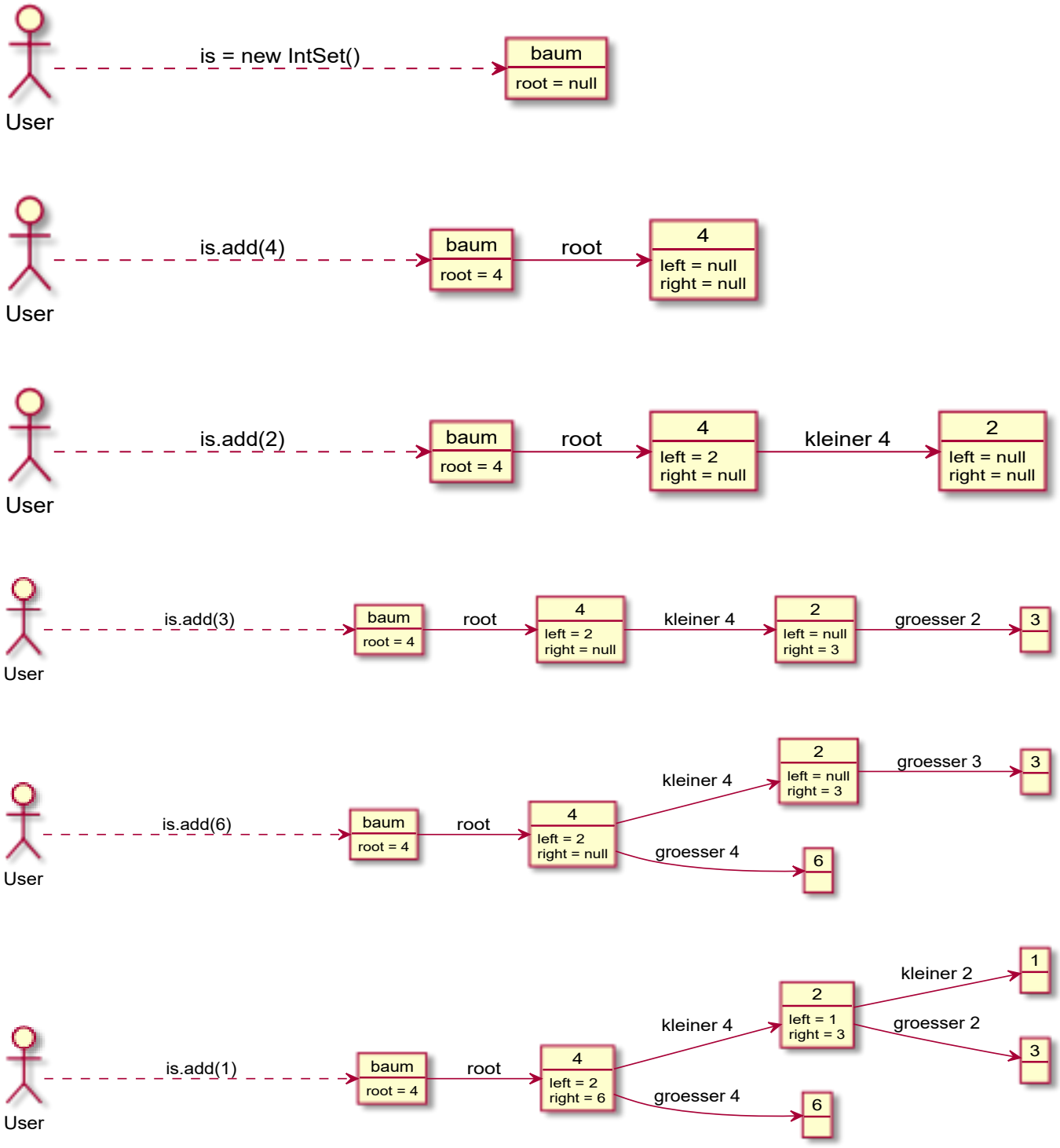
Eine fundamentale Datenstruktur in der Informatik ist der *Binärbaum*. Er unterscheidet sich von der Liste dahingehend, dass jedes Element nicht einen, sondern *zwei* Nachfolger hat -- daher der Name: *binär*.

Ein Element zeigt also nicht *sequenziell* auf das *nächste* Element, sondern unterhält Referenzen auf einen *linken* und *rechten Teilbaum*, welche dann jeweils nur *kleinere* bzw. *größere* Werte als das Element selbst enthalten.

Fügt man nun ein Element ein, so steigt man vom Wurzelknoten (engl. *root*) so weit nach links oder rechts ab, bis man entweder den Wert gefunden hat, oder an einer Stelle angekommen ist, wo man den neuen Wert einzufügen kann.

Aber ein Bild sagt mehr als 1000 Worte; hier ein Binärbaum für Zahlen (`IntSet`), in den der Reihe nach die Zahlen 4, 2, 3, 6, 1 eingefügt werden:

```
IntSet is = new IntSet();  // analog zu CharSet
is.add(4);
is.add(2);
is.add(3);
is.add(6);
is.add(1);
```



> Die Metapher *Baum* kommt von der Verästelung bzw. Verzweigung, auch wenn ein Baum ja eigentlich von der Wurzel nach Oben wächst. Aber Informatiker zählen ja auch von 0 und nicht von 1...

Möchte man nun ein Element suchen (`contains`), so beginnt man am Wurzelknoten (`root`), prüft ob der Wert bereits dort vorhanden ist, und steigt ansonsten nach links bzw. rechts ab, je nachdem ob der gesuchte Wert kleiner oder größer als das angesehene Element ist.

Wie wird so ein Baum nun implementiert, und wie steigt man nach links oder rechts ab? Beginnen wir mit der Struktur; ein Element hat nun also nicht einen Nachfolger `next`, sondern zwei: `left` und `right`.

```
class CharSetImpl2 implements CharSet {
        class Element {
                char value;
                Element left, right;
                Element(char c, Element le, Element re) {
                        value = c;
                        left = le;
                        right = re;
                }
        }
        // ...
}
```

Bei der `contains` Methode wird zwar ähnlich zur Liste iteriert, aber statt `it = it.next` muss unterschieden werden, ob man links oder rechts absteigen möchte:

```
class CharSetImpl2 implements CharSet {
        // ...
        public boolean contains(char t) {
                if (root == null)
                        return false;

                Element it = root;
                while (it != null) {
                        if (t == it.value)
                                return true;
                        else if (t < it.value) {
                                it = it.left;
                        } else {
                                it = it.right;
                        }
                }

                // nicht gefunden!
                return false;
        }
        // ...
}
```

Beim Einfügen wird nun ähnlich vorgegangen, nur dass man hier nach links oder nach rechts schauen muss, statt einfach immer ein Element voraus:

```
class CharSetImpl2 implements CharSet {
        // ...
        public void add(char c) {
                Element e = new Element(c, null, null);

                if (root == null) {
                        root = e;
                        return;
                }

                Element it = root;  // beginne an der Wurzel
                while (it != null) {
                        // schon vorhanden -> fertig!
                        if (it.value == e.value)
                                return;
                        // links absteigen?
                        else if (e.value < it.value) {
                                // wir sind unten angekommen -> einfügen!
                                if (it.left == null) {
                                        it.left = e;
                                        return;
                                } else
                                        it = it.left;
                        } else {
                                // analog
                                if (it.right == null) {
                                        it.right = e;
                                        return;
                                } else
                                        it = it.right;
                        }
                }
        }
        // ...
}
```

Den aufmerksamen Lesern ist bestimmt nicht entgangen, dass zur Vervollständigung von `CharSetImpl2` noch die Methoden `size` und `toString` fehlen -- doch wie kann man diese realisieren?

Zur Erinnerung: Bei der sequenziellen Liste konnte man im Prinzip folgendes machen:

```
String s = "" + head.value;
Element it = head.next;
while (it != null) {
        s += ", " + it.value;
        it = it.next;
}
```

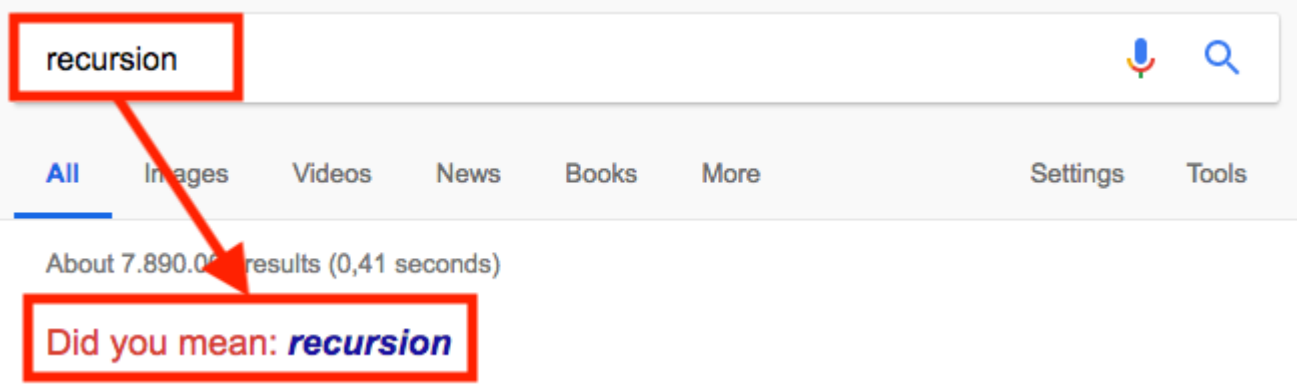
Das funktioniert, da es ja immer genau einen Nachfolger gibt. Doch was nun bei mehr als einem Nachfolger, oder eben zwei wie beim Binärbaum? Folgendes etwa deckt nur einen Teil ab:

```
String s = "" + head.value;
Element it1 = head.left;
while (it1 != null) {
        s += ", " + it1.value;
        it1 = it1.left;
}
Element it2 = head.right;
while (it2 != null) {
        s += ", " + it2.value;
}
```

Würde man das auf dem obigen Beispielbaum (zur Erinnerung: 4, 2, 3, 6, 1) anwenden, so wäre das Ergebnis `[4, 2, 1, 6]` -- die 3 wäre verloren gegangen, da diese zwar im linken Teilbaum vom Wurzelknoten ist, dort aber in einem Rechten (`root.left.right.value`).

Exkurs: Rekursion

Eine elegante Lösung für dieses Problem ist die *Rekursion*, also der wiederholte Aufruf einer Funktion durch sich selbst.



Wir beginnen mit der `size` Methode, welche zurück geben soll, wie viele Elemente im Baum gespeichert sind.

Die Idee ist die Folgende: Vielleicht weiß ich als Baum nicht, wie man so absteigt, dass man alle Knoten besucht (und entsprechend mitzählt); aber wenn ich ein Baumelement habe, so weiss ich doch, dass ich mindestens dieses Element habe, sowie dazu noch alle Elemente seines linken und rechten Teilbaums. Das kann man *rekursiv* hinschreiben:

```
class CharSetImpl2 implements CharSet {
        class Element {
                char value;
                Element left, right;
                int size() {
                        int s = 1;  // schon mal mindestens ein Element.

                        // gibts einen linken Teilbaum? Dann dessen Größe
```

```
dazuaddieren.
                        if (left != null) s += left.size();

                        // analog.
                        if (right != null) s += right.size();

                        return s;
                }
                // ...
        }

        Element root;

        public int size() {
                if (root == null) return 0;
                else return root.size();
        }
        // ...
}
```

Die Methode `size` der Klasse `Element` ruft sich hier selbst auf (`left.size()` bzw. `right.size()`), sie ist daher *rekursiv*. In der Klasse `CharSetImpl2.size()` kann jene Methode jetzt verwendet werden, um die Größe des Baums rekursiv zu bestimmen.

> Hinweis: Es ist zur Rekursion *genau* die selbe Methode aufzurufen, d.h. die Methode in der selben Klasse. Am obigen Beispiel ist die Methode `CharSet.size` *nicht* rekursiv, die Methode `Element.size` hingegen schon.

Dies geht analog für die Methode `toString`:

```
class CharSetImpl2 implements CharSet {
        class Element {
                char value;
                Element left, right;
                public String toString() {
                        String s = "" + value;  // schon mal dieses Element

                        // gibts einen linken Teilbaum? Dann dessen String dazu
                        if (left != null) s += ", " + left.toString();

                        // analog.
                        if (right != null) s += ", " + right.toString();
                }
        }

        Element root;

        public String toString() {
                if (root == null) return "[]";
                else return "[" + root.toString() + "]";
        }
```

```
        // ...
}
```

> Hinweis: Rekursion wird hier nur als Exkurs behandelt, wir kommen gegen Ende des Semesters nochmal im Detail dazu.

Löschen von Elementen

Bei der Liste war das Löschen eines Elementes einfach: Zunächst schaut man immer ein Element voraus, um das zu löschende Element zu finden; man bleibt also ein Element davor stehen. Das eigentliche Löschen wird dadurch erreicht, in dem man nun die Verlinkung so ändert, dass das Element *vor* dem zu löschenden auf das Element *hinter* dem zu löschenden zeigt.

Möchte man nun ein Element aus dem Set bzw. Baum löschen, ist das etwas komplizierter, da man beim Suchen ja rechts und links betrachten muss, analog zu `contains`.

Wir brechen dieses schwierige Problem in einfachere Teilprobleme auf. Zunächst gibt es drei Fälle zu betrachten:

1. Der Baum ist bereits leer; hier sollte eine `NoSuchElementException` geworfen werden.
2. Das zu löschende Element ist das Wurzelelement `root`
3. Das zu löschende Element ist ein innerer Knoten oder Blatt.

Ist 1. trivial, so sehen wir bei 2. und 3. zumindest eine Gemeinsamkeit: Löschen wir ein Element, so müssen wir sicherstellen, dass etwaige Teilbäume (`left` und `right`) wieder in den Baum eingefügt werden. Hierzu erstellt man eine Hilfsmethode `addElement`, welche analog zur bestehenden `add` Methode arbeitet, aber eben gleich statt einem *Wert* ein Element (mit etwaigen Teilbäumen) einfügt.

Es verbleibt noch die Unterscheidung, ob man das Wurzelelement löschen möchte (2.) oder ein Inneres (3.). Eine mögliche `remove` Implementierung könnte also wie folgt vorgehen:

```
public char remove(char c) {
        // Trivialfall prüfen
        if (root == null)
                throw new NoSuchElementException();

        // Spezialfall: Wurzelelement loeschen; delegieren
        if (root.value == c)
                return removeRoot();

        // Am Wurzelknoten beginnen
        Element it = root;
        while (it != null) {
                if (c < it.value) {
                        // ist der gesuchte Wert kleiner, so müssen
                        // wir nach links schauen, und ggf. das linke
                        // Nachfolgeelement löschen; delegieren
                        if (it.left != null && it.left.value == c)
                                return removeElement(it, it.left);
                        it = it.left;
```

```
                } else {
                        // rechts analog.
                        if (it.right != null && it.right.value == c)
                                return removeElement(it, it.right);
                        it = it.right;
                }
        }

        throw new NoSuchElementException();
}
```

Wir haben also zunächst die Grundstruktur des Algorithmus erarbeitet, welche das zu löschende Element zwar bestimmt, aber nicht (selbst) löscht (delegiert auf `removeRoot` bzw. `removeElement`) Möchte man nun das Wurzelelement `root` löschen (2.), so prüft man, ob und welche Teilbäume `left` bzw. `right` existieren:

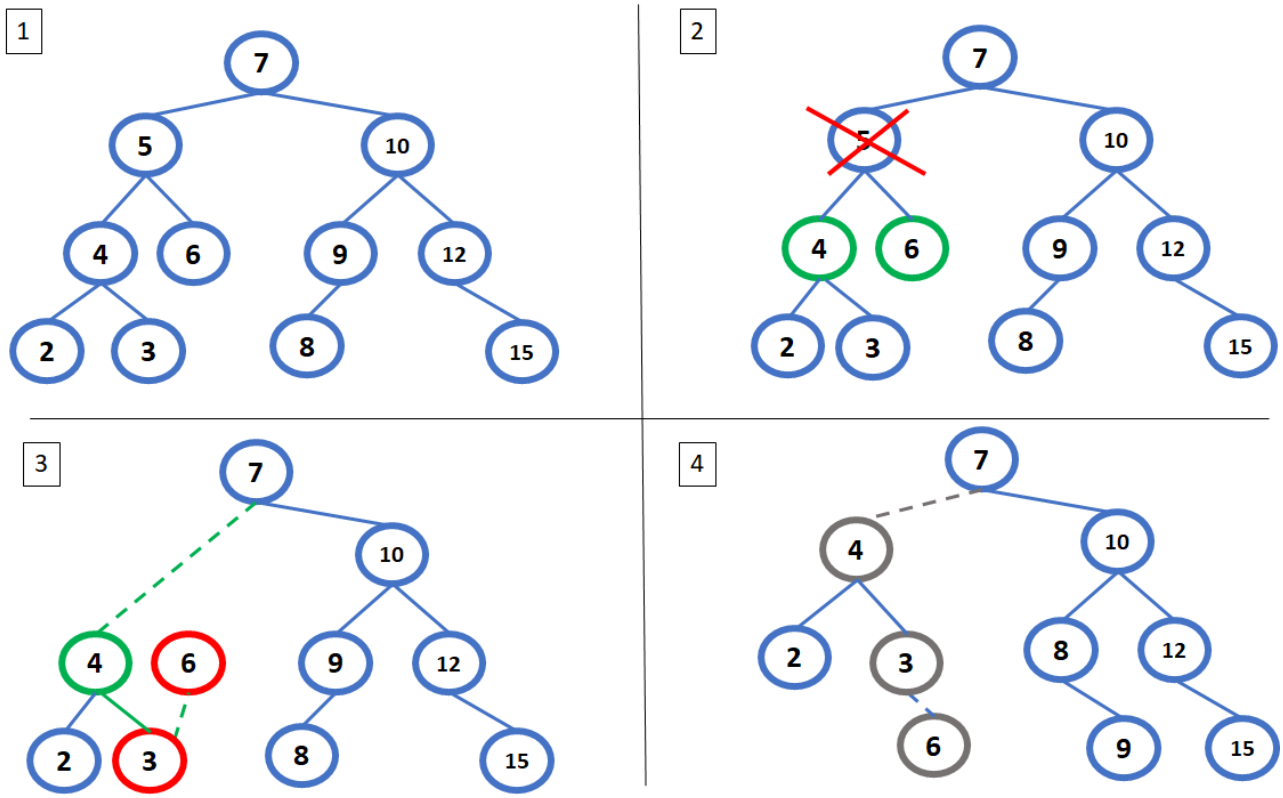
```
private char removeRoot() {
        Element e = root;
        if (e.left == null && root.right == null) {
                // keine Teilbäume, also ist der Baum nun leer
                root = null;
        } else if (e.left == null) {
                // nur rechter Teilbaum, also diesen als Wurzel setzen
                root = e.right;
        } else if (e.right == null) {
                // dito, fuer links
                root = e.left;
        } else {
                // es gibt beide Teilbäume; links wird neue Wurzel,
                // rechts wird als Element eingefügt.
                root = e.left;
                addElement(e.right);
        }

        // Wert des gelöschten Elements zurueck geben
        return e.value;
}
```

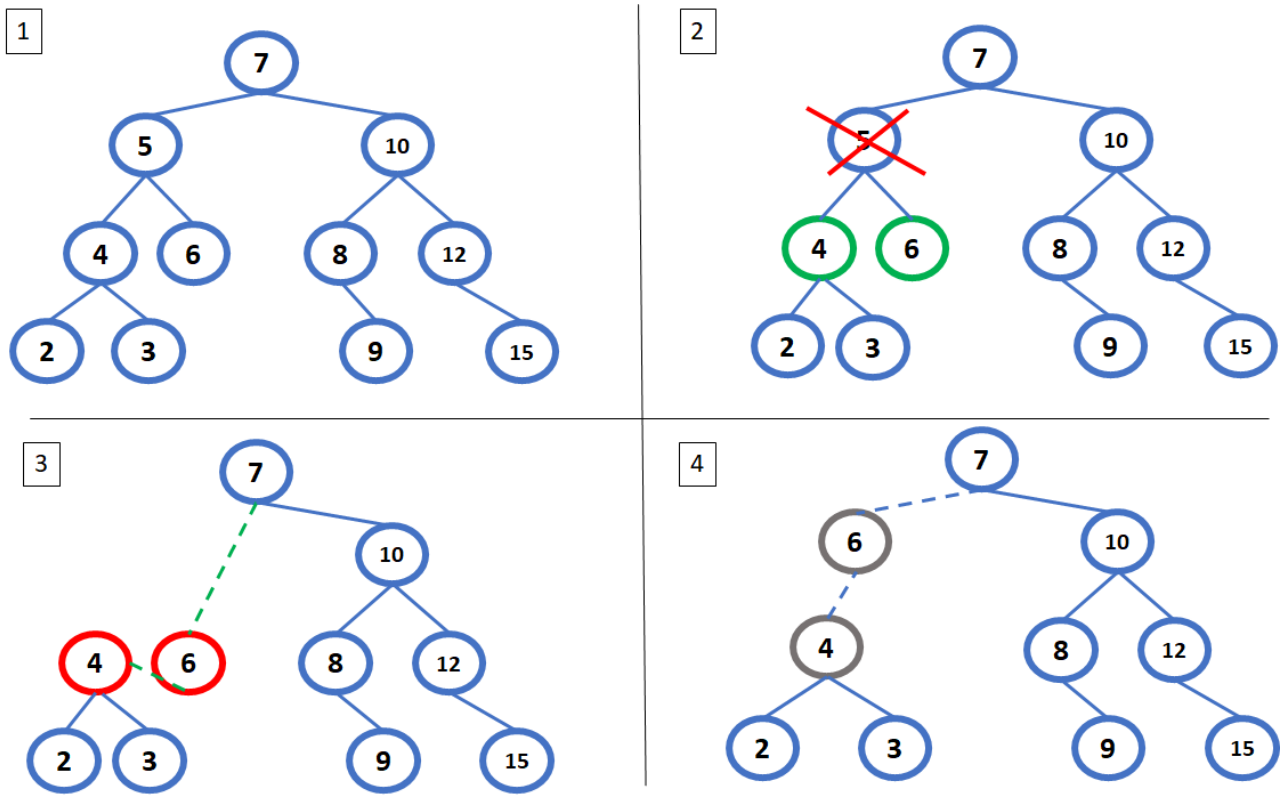
Ähnlich geht man bei inneren Elementen vor.

Schauen wir uns das Ganze mal graphisch an:

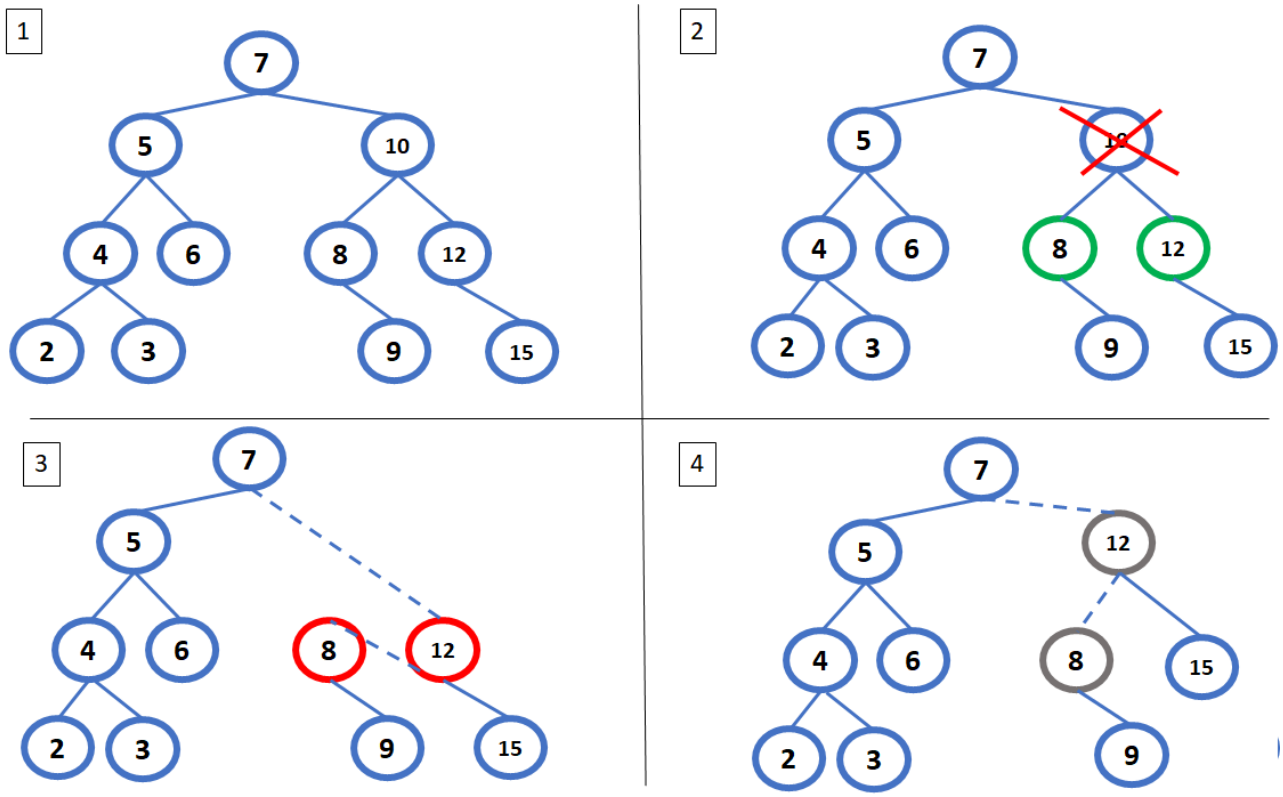
1. Möglichkeit



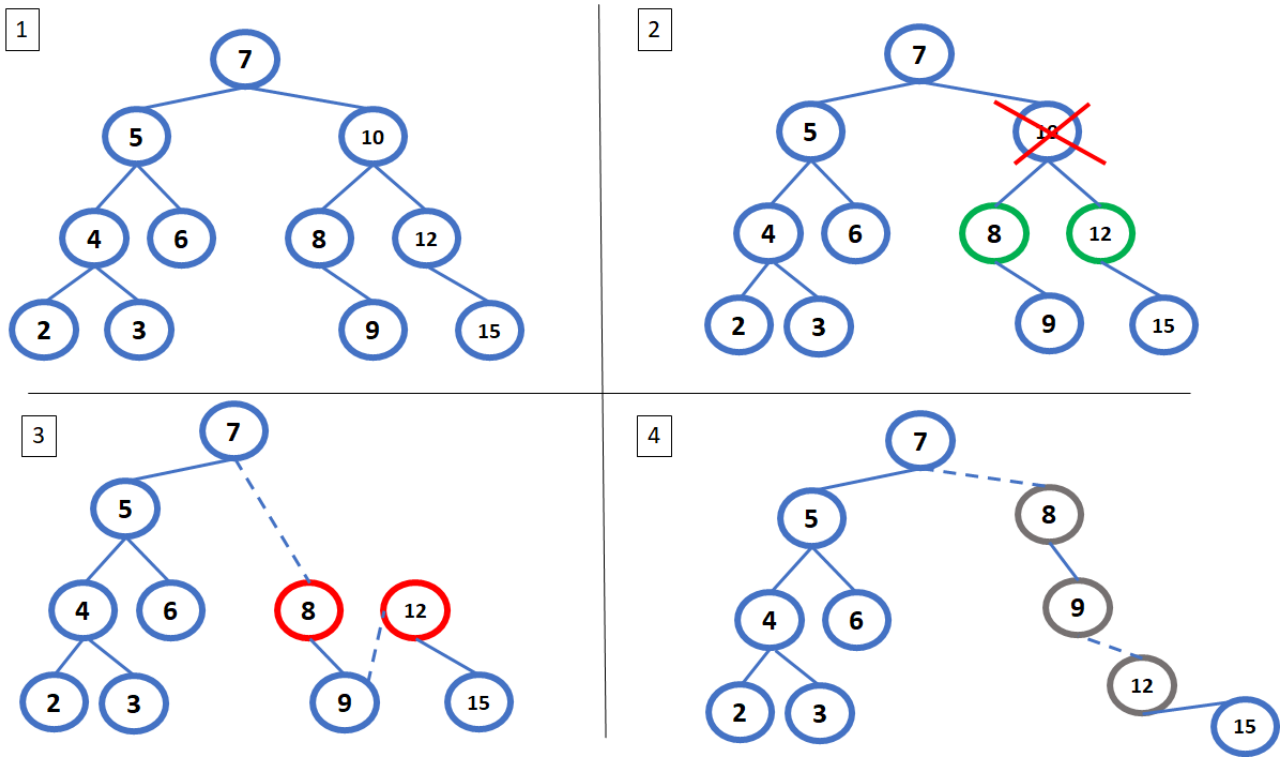
2. Möglichkeit



3. Möglichkeit



4. Möglichkeit



Da der Baum hier aber nur in eine Richtung verlinkt ist, benötigen wir zum Löschen Referenzen auf das Elternelement und das zu löschende Element. Man setzt dann das zu löschende Element auf `null`, und fügt etwaige Teilbäume wieder neu ein.

```
private char removeElement(Element parent, Element element) {
        // wollen wir das linke Element löschen?
        if (element == parent.left) {
```

```
                parent.left = null;
        } else {
                parent.right = null;
        }

        // eventuelle Teilbäume neu in den Baum einfügen
        addElement(element.left);
        addElement(element.right);

        return e.value;
}
```

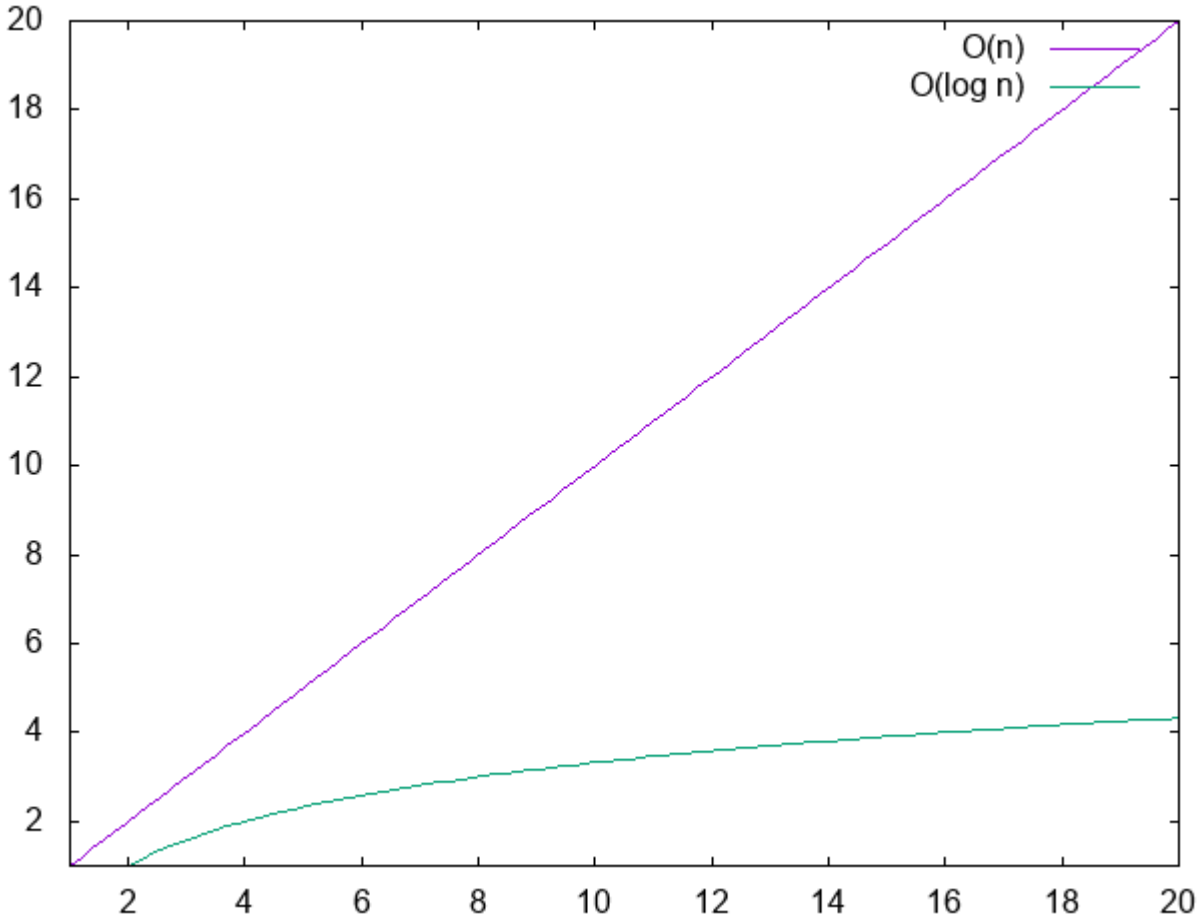
Die `addElement` Methode wird analog zur `add` Methode implementiert, und kann im Beispielcode nachvollzogen werden.

> Hinweis: Diese Implementierung der `remove` Funktion fuehrt dazu, dass der Baum schlecht konditioniert ist bzw. entartet; sie ist aber recht anschaulich zu erklären.

Komplexität

Warum nun das ganze Spektakel mit dem Baum, wenn doch eine Liste so viel einfacher wäre? Man kann mathematisch-kombinatorisch beweisen, dass der mittlere Aufwand (Komplexität) zum Suchen bzw. Einfügen in einen Binärbaum $O(\log n)$ ist; ist der Baum *balanciert* (also die Verzweigung ausgewogen), so ist der Aufwand sogar im *worst case* $O(\log n)$.

Wir erinnern uns an den Eingang dieses Kapitels: Die naive Implementierung eines Sets als Liste hatte die Komplexität $O(n)$. Das scheint beim Lesen nicht arg unterschiedlich, doch auch hier sagt ein Bild mehr als 1000 Worte:



Man sieht: bei linear steigenden n (x-Achse) bleibt die logarithmische Steigung sehr schnell sehr weit unter der Linearen. Die Baumstruktur ist also wesentlich effizienter!

Zusammenfassung

- Ein **Set** ist im Gegensatz zu einer Liste **frei von Duplikaten**; in der Regel werden `add`, `remove`, `contains` und `size` unterstützt.
- Da ein **Set** eine **Menge ohne Ordnung** (Reihenfolge) ist, gibt es keinen Zugriff über einen Index.
- Ein **Binärbaum** ist eine Datenstruktur, bei der Elemente **zwei** Nachfolger haben; in diesen **Teilbäumen** sind dann die kleineren (links) und größeren Elemente (rechts) gespeichert.
- Ein Set kann zwar mit einer Liste implementiert werden, ein Binärbaum ist aber **deutlich effizienter**.
- Bei komplizierter Struktur kann **Rekursion** oft eine elegante Lösung sein; hierbei ruft eine Methode sich selbst wieder auf.

■