Binäre Suchbäume

Sind die Einträge einer Menge (bzw. die Schlüssel einer *map*) angeordnet, so können alternativ zu Hashtabellen *binäre Suchbäume* eingesetzt werden.

Der Einfachheit halber arbeiten wir mit Einträgen vom Typ `String`. Im allgemeinen nimmt man `<E extends Comparable<E>>` o.ä. Ein binärer Baum besteht aus Objekten (“Knoten”) der folgenden Klasse:

```
class Node {
    String data;
    Node left;
    Node right;

    Node(String data, Node left, Node right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }
}
```

Grafische Darstellung

Das Objektdiagramm zu einem binären Baum sieht tatsächlich wie ein Baum aus.

Man zeichne das Objektdiagramm zu

```
fr = new Node("Friedrich",null,null);
ma = new Node("Margarete",null,null);
to = new Node("Torsten",fr,ma);
sa = new Node("Sabine",null,null);
yannick = new Node("Yannick",to,sa);
```

Strukturbedingung

Genauer gesagt liegt ein binärer Baum nur dann vor, wenn das Objektdiagramm tatsächlich wie ein Baum aussieht.

```
kl = new Node("Klon",yannick,yannick);
```

ist kein binärer Baum.

Erst recht ist

```
om = new Node("Om",null,null);
om.left = om; om.right = om;
```

kein binärer Baum.

Wir verzichten auf die formale Definition.

Terminologie

Ist `t` ein Baum, so bezeichnet man den von `t` bezeichneten Knoten selbst als *Wurzel* des Baumes (der Baum selbst besteht ja aus der Gesamtheit aller Knoten, nicht nur dem Wurzelknoten, aber auf den zeigt `t`).

Die Bäume `t.left` und `t.right` heißen *rechter* und *linker Teilbaum*.

Ein Knoten `n` mit `n.left==null` und `n.right==null` heißt *Blatt*.

Von der Wurzel gibt es zu jedem Blatt einen eindeutigen *Pfad* durch Verfolgen der `left` und `right` Felder. Die Länge des längsten Pfades heißt *Höhe* des Baumes.

Ein Knoten, der kein Blatt ist, heißt *innerer Knoten*.

Die Bäume, die von den Knoten von `t` ausgehen, heißen Teilbäume von `t`.

Einträge

Zu einem binären Baum definiert man die Menge seiner Einträge durch folgende Methode:

```
public static  Set<String> entries(Node t) {
    Set<String> result = new HashSet<String>();
    if (t == null) return result;
    else {
        result.add(t.data);
        result.addAll(entries(t.left));
        result.addAll(entries(t.right));
        return result;
```

So gilt etwa

```
entries(yannick) =
     {"Yannick", "Thorsten", "Sabine", "Friedrich", "Margarete"}
```

Binärer Suchbaum

Ein binärer Baum `t` ist ein *binärer Suchbaum* (binary search tree, BST), wenn folgendes gilt:

Entweder ist `t` gleich `null`, also leer,

Oder alle Elemente von `entries(t.left)` sind kleiner als `t.data` und `t.data` ist kleiner als alle Elemente von `entries(t.right)`

Und `t.left` und `t.right` sind selbst wieder binäre Suchbäume.

[Beispiele an der Tafel]

Suche in BST

Um festzustellen, ob ein gegebenes Element in einem binären Suchbaum vorhanden ist, verwendet man folgende Methode:

```
public static  boolean member(String x, Node t) {
    if (t==null) return false;
    else if (x.compareTo(t.data) == 0)
            return true;
    else if (x.compareTo(t.data) > 0)
            return member(x, t.left);
    else /* x.compareTo(t.data) < 0 */
            return member(x, t.right);
}
```

Einfügen in BST

Um ein neues Element in einen binären Suchbaum einzufügen, verwendet man folgende Methode:

```
public static  Node insert(String x, Node t) {
    if (t==null) return new Node(x,null,null);
    else if (x.compareTo(t.data) == 0)
        return t;
    else if (x.compareTo(t.data) > 0) {
        t.left = insert(x,t.left);return t;
    }
    else {
        t.right = insert(x,t.right);return t;
    }
}
```

Erläuterung

- Durch Vergleich mit dem Wurzeleintrag stellt man fest, ob das neue Element im linken oder im rechten Teilbaum einzufügen ist.
- Man könnte versuchen, `insert` mit Rückgabetyp `void` zu implementieren. Dann gibt es aber Probleme mit `insert(x,null);`.

  Anders gesagt, gibt `insert(x,t)` meistens wieder das Objekt `t` selbst zurück, nachdem allerdings eines seiner "Kinder" modifiziert wurde.

  Im Falle `insert(x,null)` wird aber natürlich nicht `null` zurückgegeben, sondern ein frischer Baum mit einem Eintrag.
- Man könnte das Einfügen auch iterativ mit einer `while` Schleife realisieren, die den entsprechenden Pfad des Baumes abfährt.

Entfernen aus BST

Will man einen Eintrag `x` aus `t` entfernen, so gibt es vier Fälle:

- `x.compareTo(t.data) < 0`: Man entferne `x` rekursiv aus `t.left`;
- `x.compareTo(t.data) > 0`: Man entferne `x` rekursiv aus `t.right`;
- `x.compareTo(t.data) == 0`: und `t.right==null`. Man gebe `t.left` zurück.
- `x.compareTo(t.data) == 0`: und `t.right!=null`. Man überschreibe `t.data` mit dem nächstgrößeren Eintrag. Der befindet sich ganz links in `t.right`.

Entfernen aus BST

```
public static  Node remove(String x,Node t) {
    if (t==null) return t;
    else if (x.compareTo(t.data) == 0) {
        if (t.right == null) return t.left;
        else {
                Node s = t.right;
            if (s.left == null) {
                t.data = s.data;
                t.right = s.right;
            } else {
                while(s.left.left != null)
                        { s = s.left; }
                t.data = s.left.data;
                s.left = s.left.right;
            }
            return t;
        }
```

Entfernen aus BST

```
        }
        else if (x.compareTo(t.data) > 0) {
            t.left = remove(x,t.left);return t;
        }
        else {
            t.right = remove(x,t.right);return t;
        }
    }
```

Komplexität

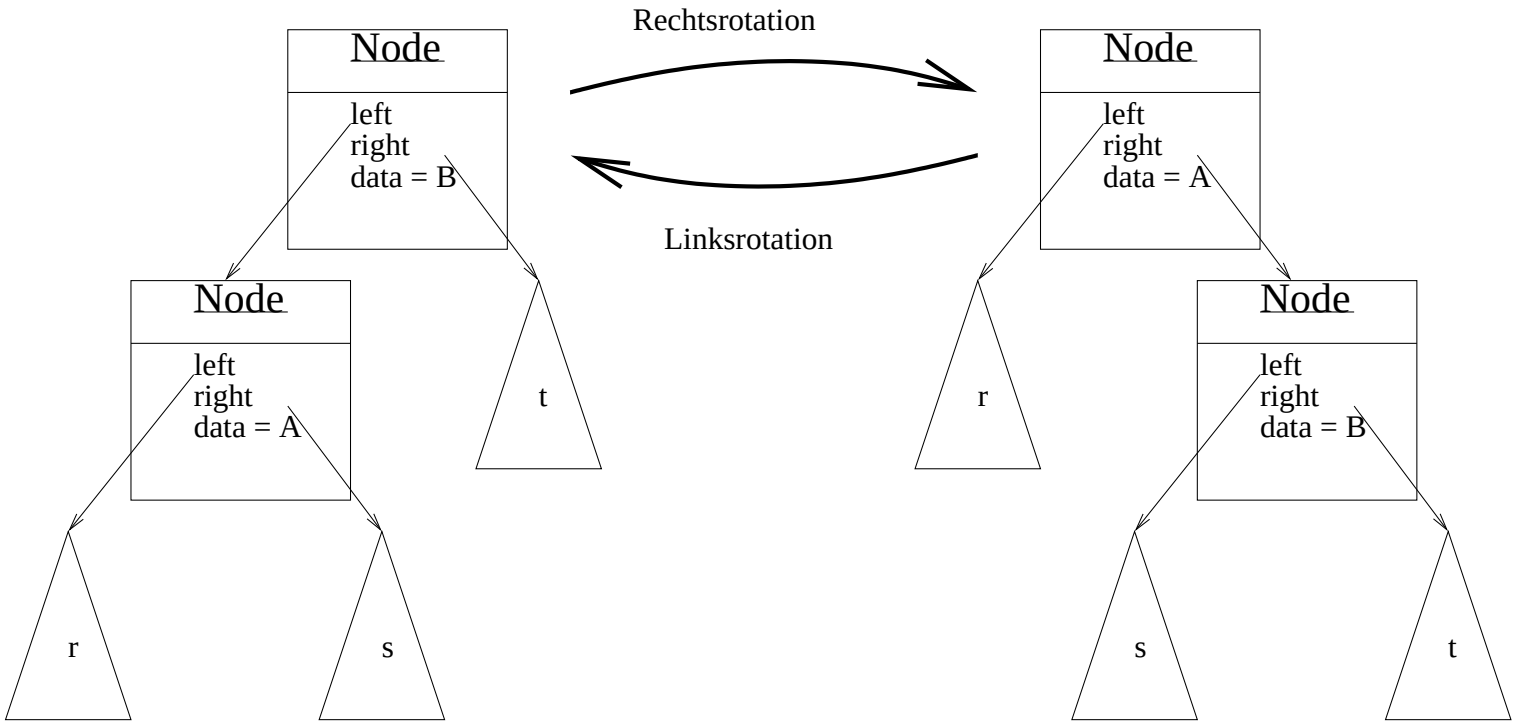
Die Laufzeit der beschriebenen Operationen auf einem BST `t` ist proportional zur *Höhe* von `t`. (*Höhe* = Länge des längsten Pfades von der Wurzel zu einem Blatt.)

Unterscheiden sich die Höhe des linken und rechten Teilbaums um höchstens Eins und gilt diese Eigenschaft auch für alle Teilbäume, so ist der Baum *balanciert*. In diesem Fall ist die Höhe proportional zum Logarithmus der Zahl der Einträge.

Geht die Balancierung durch eine Einfüge- oder Löschoperation verloren, so kann man sie durch geeignete *Rotationen* wiederherstellen.

In der Praxis führt man im BST zusätzliche Verwaltungsinformation mit, die es erlaubt, eine drohende Verletzung der Balancierung schnell zu erkennen. (AVL-Bäume, Rot-Schwarz-Bäume).

Links- und Rechtsrotation



Man beachte, dass die Rotationen die BST-Eigenschaft erhalten.

Verwendung von BST

Mit BST lassen sich die Schnittstellen `Set<E>` und `Map<E>` implementieren (Java's `TreeSet<E>` und `TreeMap<E>`).

Bei geeigneter Balancierung garantieren BST eine schnelle Zugriffszeit (Logarithmus der Größe). Bei Hashtabellen hängt die Zugriffszeit von der Hashfunktion ab und der Verteilung der Zugriffe ab.

Verwendet man konsequent die Schnittstellen `Set` und `Map`, so kann man sehr leicht zwischen den beiden Implementierungen wechseln.

Zusammenfassung Hashtabellen und Suchbäume

- Mengen und Abbildungen sind als Schnittstellen `Set` und `Map` repräsentiert und erlauben den Zugriff auf Daten ohne Rücksicht auf die Reihenfolge.
- Hashtabellen und binäre Suchbäume implementieren diese Schnittstellen.
- Binäre Suchbäume (BST) können verwendet werden, wenn die Daten angeordnet sind.
- In einem BST befinden sich links von einem Knoten jeweils kleinere Einträge und rechts von einem Knoten jeweils größere Einträge. Dadurch kann man sich bei der Suche jeweils auf einen einzigen Pfad beschränken.

Ende der Vorlesung.