Abstrakter Datentyp Menge

Es sei T ein beliebiger Datentyp. Eine abstrakter Datentyp *Menge* (von Elementen aus T) unterstützt die folgenden Operationen:

- `void insert(T e)`
  Fügt ein Element e vom Typ T in eine Menge M ein.
  $M := M \cup \{e\}$
- `void remove(T e)`
  Löscht das Element e aus einer Menge M.
  $M := M \setminus \{e\}$
- `boolean contains(T e)`
  Prüft, on e sich in einer Menge M befindet.
  Ist $e \in M$ wahr?

Abstrakter Datentyp Wörterbuch

Definition 6.4. Es seien S und T beliebige Datentypen. Ein *Wörterbuch (Dictionary, Map)* ist eine Datenstruktur, die die folgenden Operationen unterstützt:

- `void put(S k, T v)`
  Hinterlegt im Wörterbuch zum Schlüssel k den assoziierten Wert v.
- `T get(S k)`
  Liefert den im Wörterbuch hinterlegten assoziierten Wert für Schlüssel k.
- `void remove(S k)`
  Löscht aus dem Wörterbuch den Schlüssel k und seinen assoziierten Wert.

☞ Ein Wörterbuch entspricht einer partiellen Funktion $f : S \longrightarrow T$.

Effizienz bekannter Implementierungsansätze

Menge bzw. Wörterbuch als verkettete Liste:

- `insert()` bzw. `put()` hat Zeitaufwand $O(1)$
- `remove()`, sowie `contains()` bzw. `get()` haben den Zeitaufwand $O(n)$ (für n Elemente), da die Liste sequentiell durchsucht werden muss.

Menge bzw. Wörterbuch als sortiertes Feld:

- `contains()` bzw. `get()` haben Zeitaufwand $O(\log n)$ bei Implementierung mittels binärer Suche (siehe "Einführung in die Programmierung", Folien 417 bis 421).
- Alle anderen Operationen haben i.A. den Zeitaufwand $O(n)$.

☞ Implementierung möglich mit allen Operationen besser als $O(n)$?

Suchbäume

Definition 6.5. Ein binärer Suchbaum T ist ein binärer Baum mit den folgenden Eigenschaften:

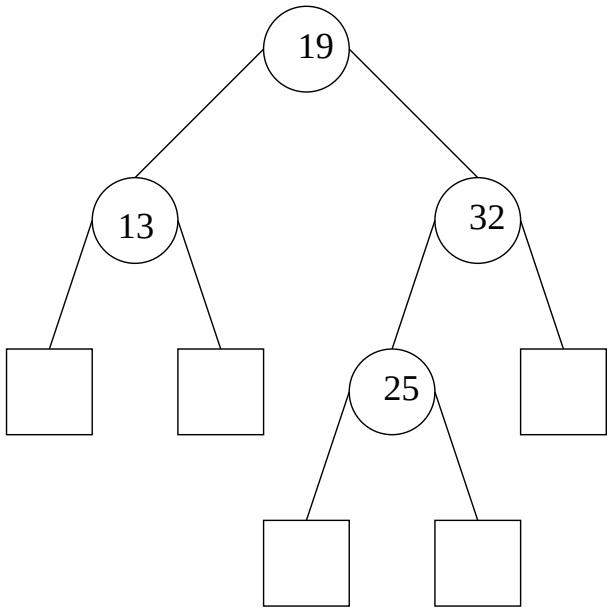
- Jeder innere Knoten w enthält bzw. speichert einen Wert $val(w)$.
- Für alle innere Knoten w und für alle Werte v im linken Teilbaum zu einem Knoten w gilt:
$$v < val(w)$$
- Für alle innere Knoten w und für alle Werte v im rechten Teilbaum zu einem Knoten w gilt:
$$v > val(w)$$

Im Folgenden sagen wir kurz Suchbaum statt binärer Suchbaum.

Beispiel Suchbaum

- Der Baum rechts hat die Eigenschaften eines Suchbaums.
- Die in den inneren Knoten enthaltenen Werte sind bspw. die Elemente einer Menge oder die Schlüssel eines Wörterbuchs.
- Die externen Knoten entsprechen Intervallen ohne Wert bzw. Schlüssel.
- Durch die Verzweigung muss nicht die ganze Menge nach einem Schlüssel durchsucht werden.
- Im Idealfall Suche (`get()` bzw. `contains()` in $O(\log n)$) möglich.

Im Folgenden verzichten wir auf die Darstellung der externen Knoten.



Suchbaum in Java

```
public class SearchTree<T extends Comparable<T>> {

    private class Node {
        T    value;      // Wert am Knoten
        Node left;       // linker Unterbaum
        Node right;      // rechter Unterbaum
    }
    ...
    private Node root;   // Wurzelknoten
}
```

Suchen

Am Beispiel der Methode `boolean contains(T elem)` für Mengen:

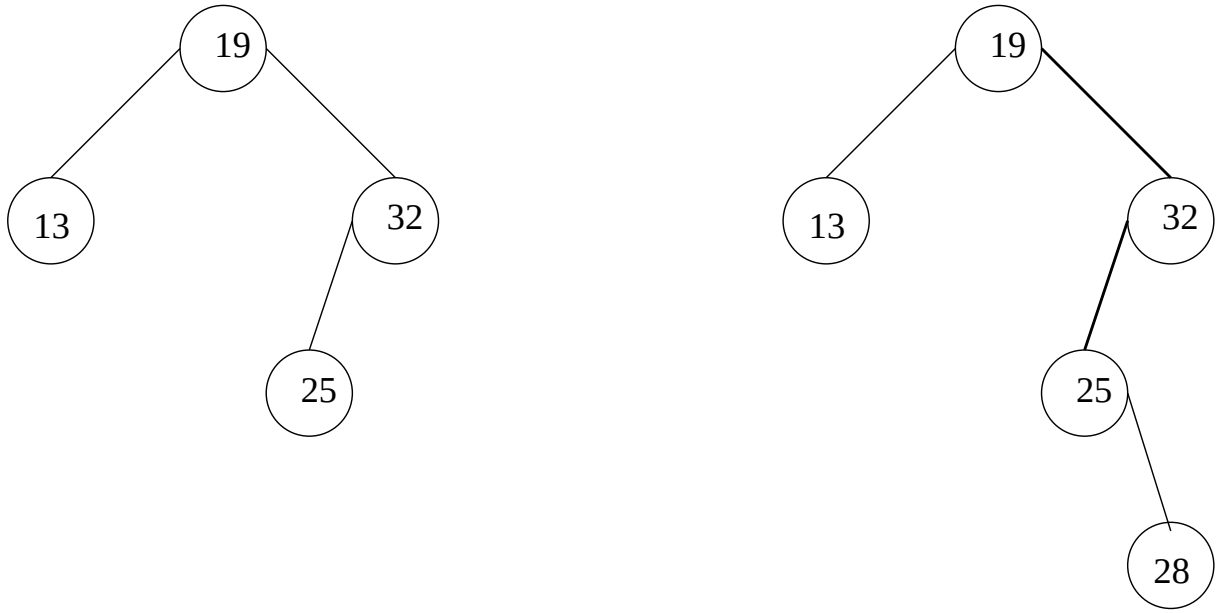
```
Node node = root;
int  diff;

while (node != null) {
    diff = node.value.compareTo(elem);
    if (diff == 0) {              // Element gefunden
        return true;
    }
    else if (diff>0) {            // node.value > elem,
        node = node.left;         // dann liegt elem im linken Unterbaum
    }
    else {                        // node.value < elem,
        node = node.right;        // dann liegt elem im rechten Unterbaum
    }
}
return false;
```

Einfügen: Logik

- Wir gehen davon aus, dass wir einen neuen Wert x einfügen möchten, der noch nicht im Suchbaum enthalten ist.
- Wie beim Suchen muss man den Baum durchlaufen, bis man an einem Knoten w nicht mehr weiterkommt (leerer Unterbaum).
- Wir legen einen neuen Knoten r mit $val(r) = x$ an und setzen r als Unterbaum von w ein.

Einfügen: Veranschaulichung

Einfügen von 28:



Einfügen: Java

Methode `void insert(T elem)`:

```
Node node = root;    // Hilfsknoten fuer Suche und Einfuegen
Node father = null;  // Vater von node bei Suche
int diff = 0;

while (node!=null) {
    father = node;
    diff = node.value.compareTo(elem);
    if (diff==0) {   // elem schon in Suchbaum vorhanden
        return;
    }
    else if (diff>0) {
        node = node.left;
    }
    else {
        node = node.right;
    }
```

```
}

// neuen Knoten anlegen und initialisieren
node = new Node();
node.left = node.right = null;
node.value = elem;

if (father==null) {     // Baum war bisher leer
    root = node;        // dann ist node jetzt die Wurzel
}
else {                  // ansonsten wird node Unterbaum von father
    if (diff>0) {       // father.value > elem => neuer linker Unterbaum
        father.left = node;
    }
    else {              // father.value < elem => neuer rechter Unterbaum
        father.right = node;
    }
}
```

Löschen: Logik

- Wir möchten den Wert x löschen.
- Suche Knoten w mit zu löschendem Wert x. Es sei T_l der linke und T_r der rechte Unterbaum von w.
- Möglichkeiten:

(a) w ist ein Blatt (d.h. T_l und T_r sind leer):
Dann lösche w.

(b 1) T_l ist nicht leer, T_r ist leer:
Dann ersetze w durch die Wurzel von T_l.

(b 2) T_l ist leer, T_r ist nicht leer:
Dann ersetze w durch die Wurzel von T_r.

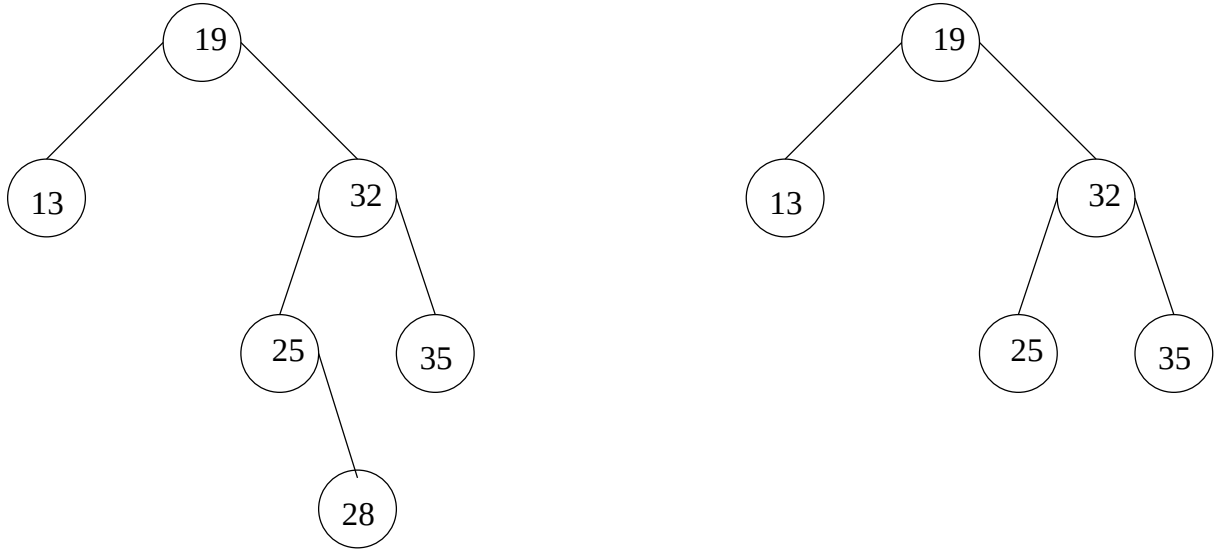
(c) Weder T_l noch T_r ist leer:
Dann bestimme den Knoten w' mit dem größten Wert x' in T_l. Setze den Wert x' in w ein und Lösche w'.

Löschen: Bemerkungen

- Die Fälle (b 1) und (b 2) sind symmetrisch.
- In Fall (c) kann man statt des größten Wertes in T_l auch den kleinsten Wert in T_r bestimmen.
- In Fall (c) kann für den zu löschenden Knoten w' nicht wieder der Fall (c) vorliegen.

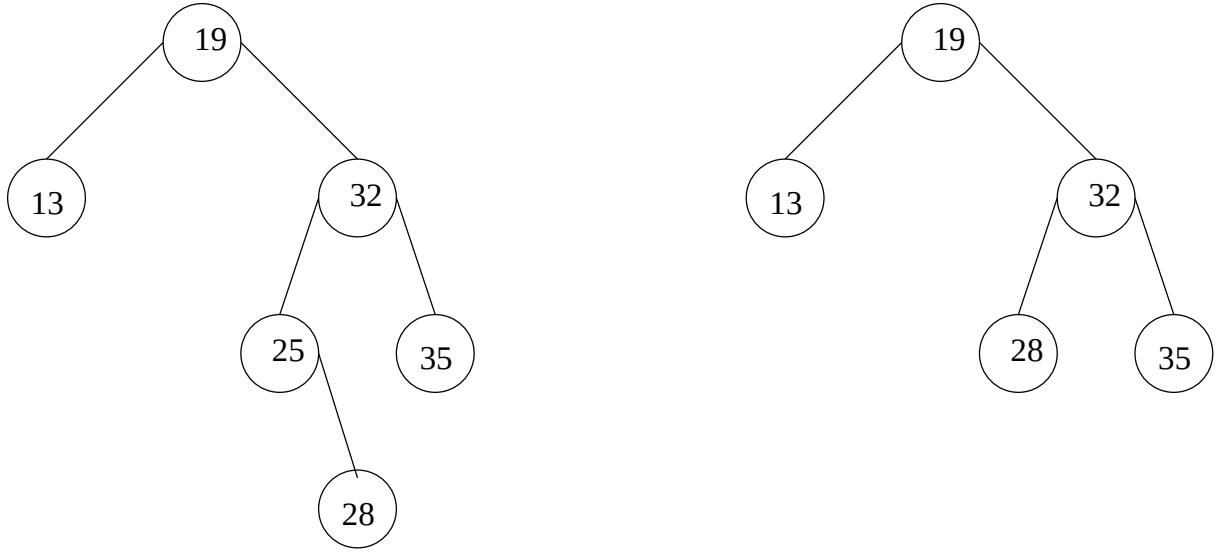
Löschen: Veranschaulichung (1)

Löschen von 28 (Fall (a)):



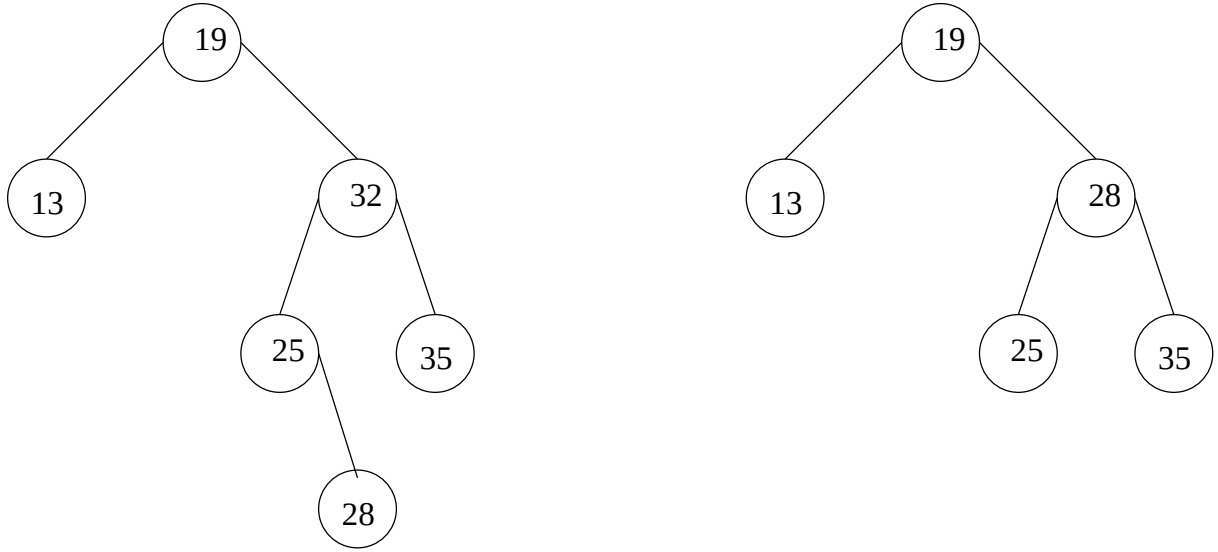
Löschen: Veranschaulichung (2)

Löschen von 25 (Fall (b 2)):



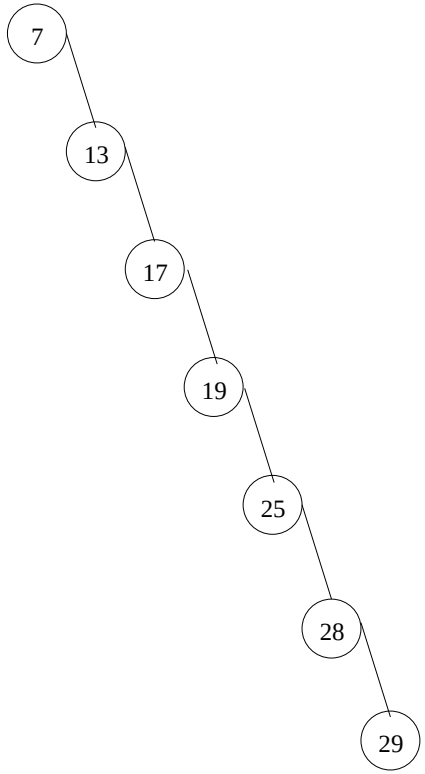
Löschen: Veranschaulichung (3)

Löschen von 32 (Fall (c)):

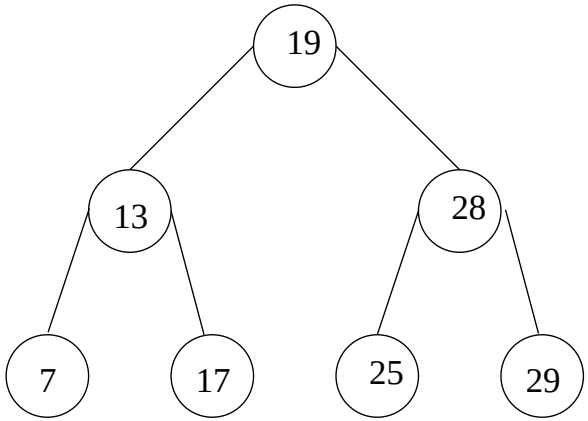


Ausgeglichener und unausgeglichener Baum

Unausgeglichener Baum:



Ausgeglichener Baum:



Effizienz

- Der Aufwand jeder Operation ist proportional zur Höhe des Baumes.
- Beim Einfügen und Löschen muss stets ein Pfad von der Wurzel zu einem Blatt durchlaufen werden.
- Beim Suchen muss im Worst-Case ein Pfad von der Wurzel zu einem Blatt durchlaufen werden.
- Wenn für jeden Knoten eines Baums gilt, dass sich die Höhe des linken und des rechten Teilbaums nicht oder nur gering unterscheiden, dann nennen wir den Baum *ausgeglichen*. Ein Baum, der nicht ausgeglichen ist, wird auch als *entartet* bezeichnet.
- Die Höhe von ausgeglichenen Bäumen ist $O(\log n)$.
- Konsequenz: Für ausgeglichene Bäume können alle Operationen in Zeit $O(\log n)$ ausgeführt werden.