Überblick

17. Datenstrukturen
17.1 Einleitung
17.2 Listen
17.3 Assoziative Speicher
17.4 Bäume
17.5 Mengen
17.6 Das Collections-Framework in Java
17.7 Zusammenfassung

Was ist ein assoziativer Speicher?

- Ein assoziativer Speicher ist eine materialisierte Abbildung, die einen Schlüssel auf einen Wert abbildet.
- Als einfachen assoziativen Speicher haben wir die Arrays kennengelernt: Ein Schlüssel (der Index) wird auf einen Wert (der an der Stelle `index` im Array gespeichert ist) abgebildet:

Beispiel

Eine **`char`**-Reihung `gruss` der Länge 13:

`gruss:`	'H'	'e'	'l'	'l'	'o'	','	' '	'W'	'o'	'r'	'l'	'd'	'!'
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12

$$\texttt{gruss} : \{0, 1, \ldots, 12\} \rightarrow \texttt{char}$$

$$i \mapsto \begin{cases} \text{'H'} & \text{falls} \quad i = 0 \\ \text{'e'} & \text{falls} \quad i = 1 \\ & \vdots \\ \text{'!'} & \text{falls} \quad i = 12 \end{cases}$$

Verallgemeinerung

- Allgemein ist ein assoziativer Speicher denkbar als Abbildung aus einer beliebigen Domäne in eine andere (oder auch die gleiche).
- Statt einer Indexmenge ist der Definitionsbereich der Abbildung also irgendeine Domäne, aus der die Schlüssel stammen.
- Beispiel:
  Wörterbuch deutsch – englisch
  - `String` $\rightarrow$ `String`
  - "hallo" $\mapsto$ "hello"

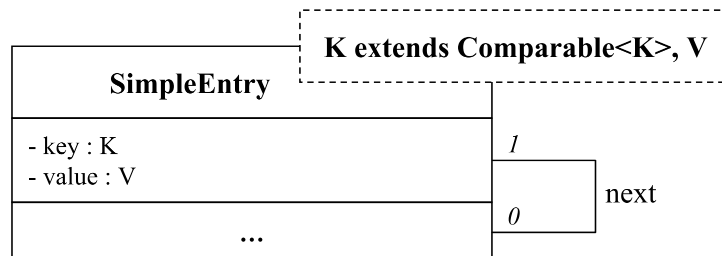
  Platznummer für Passagier auf einem bestimmten Flug
  - `String` $\rightarrow$ **`int`**
  - "Hans-Peter Kriegel" $\mapsto$ 17
- Einen assoziativen Speicher nennt man auch *Map*, *Dictionary* oder *Symboltabelle*.

Computerbasierte assoziative Speicher

- Wörterbücher, Telefonbücher, Lexika und ähnliche gedruckte Nachschlagewerke (also statische assoziative Speicher) unterstützen durch alphabetische Sortierung der Schlüssel (Stichwörter, Namen, ...) effiziente Suche nach einem Eintrag zu einem bestimmten Schlüssel.
- Computerbasierte assoziative Speicher sind dagegen auch dynamisch. Sie können wachsen und schrumpfen (ähnlich wie Listen gegenüber Arrays).
- Im Folgenden betrachten wir grundlegende Ideen, um dieses dynamische Verhalten für das Einfügen neuer Schlüssel und Werte und ihr Löschen effizient zu ermöglichen.

1. Versuch: sortierte lineare Liste

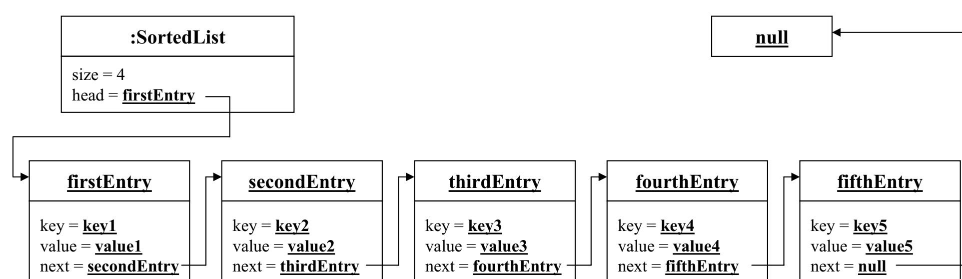
- Als ersten Ansatz versuchen wir einen assoziativen Speicher als *sortierte* lineare Liste zu realisieren.
- Die Klasse `SimpleEntry` muss daher angepasst werden, um nun Schlüssel *und* Eintrag (Wert) zu verwalten.



- Frage: Warum muss der Datentyp `K` für den Schlüssel `key` das Interface `Comparable` implementieren?

1. Versuch: Einfügen/Löschen/Zugriff

- Beim Einfügen wird zunächst die Stelle gesucht, an die der neue Eintrag eingefügt werden muss, um die Sortierung weiterhin zu gewährleisten.
- Dazu muss die Liste vom Beginn an solange durchlaufen werden, bis der aktuelle Eintrag einen Schlüssel besitzt, dessen Wert größer als der Schlüsselwert des neuen Eintrags ist.
- Analog muss beim Löschen der Eintrag gesucht werden, der zum angegebenen Schlüssel passt (durch Suchen vom Beginn der Liste).
- Frage: Wie lässt sich der Zugriff auf den Wert eines bestimmten Schlüssels realisieren?



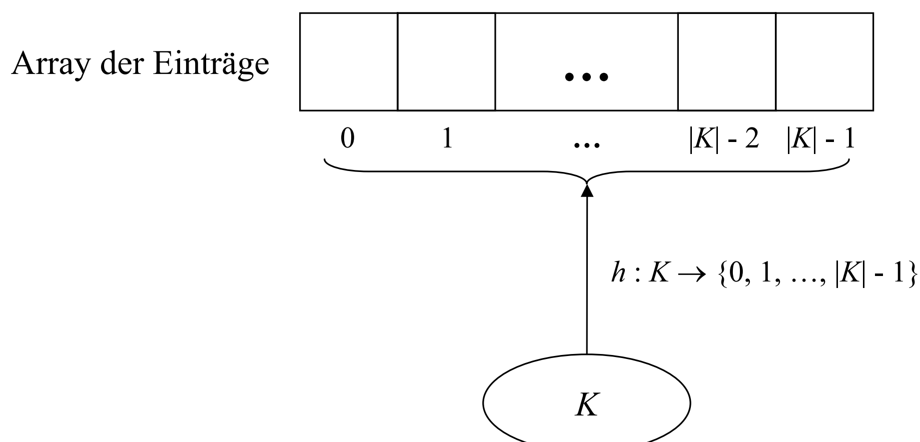
1. Versuch: Analyse

- Effizienzanalyse (worst case):
  - **Einfügen:** Sequentielle Suche nach der richtigen Einfügestelle $\Rightarrow O(n)$.
  - **Löschen:** Sequentielle Suche nach dem zu löschenden Schlüssel $\Rightarrow O(n)$.
  - **Zugriff auf Eintrag mit bestimmten Schlüssel:** Sequentielle Suche nach dem angefragten Schlüssel $\Rightarrow O(n)$.
- Dynamik:
  Die Implementierung mittels sortierter linearer Liste ist dynamisch.
- Fazit:
  Implementierung ist dynamisch, aber nicht wirklich effizient.

2. Versuch: Array

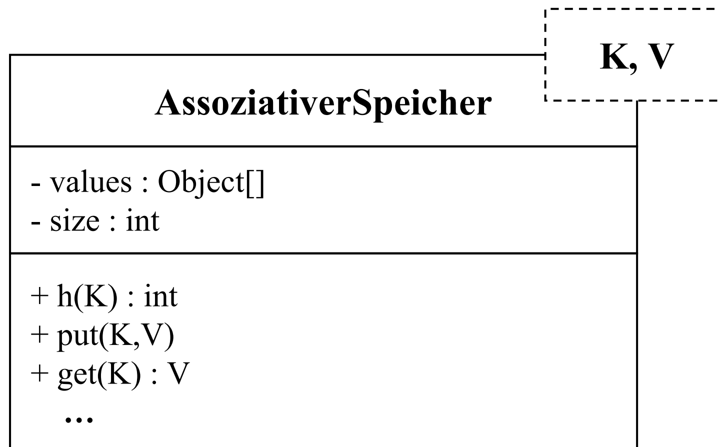
- Idee: Verwende ein Array um die Einträge zu verwalten.
- Wenn wir eine eindeutige Abbildung aller Schlüsselwerte auf einen Array Index definieren können, hätten wir konstanten Zugriff auf die Einträge (im best case)!
- Für die Menge der Schlüssel K benötigen wir also eine Abbildung

  $h : K \rightarrow \{0, \ldots |K| - 1\}$ ($|K|$ bezeichnet die Anzahl der Schlüssel in$|K|$).



2. Versuch: Realisierung

- Die Funktion h bezeichnet man als *Hashfunktion* oder auch als *Adressfunktion*.
- Assoziative Speicher werden meist mittels einer Hashfunktion realisiert (*Hashverfahren*).
- Verwendete Datenstruktur: Array von Objekten des Typs der Einträge.



2. Versuch: Realisierung (Einfügen/Löschen/Suchen)

Theoretisch:

- Einfügen eines Eintrags v mit Schlüssel k:
  - Berechne den Hashwert des Schlüssels k: Auswertung von $h(k)$.
  - Füge Eintrag v in `values[h(k)]` ein (falls bisher leer).
  - Aufwand: $O(1)$ bzw. abhängig von der Auswertung von $h(k)$.
- Löschen eines Eintrags mit Schlüssel k:
  - Berechne den Hashwert des Schlüssels k: Auswertung von $h(k)$.
  - Lösche Eintrag in `values[h(k)]` (setze `values[h(k)] = null;`).
  - Aufwand: $O(1)$ bzw. abhängig von der Auswertung von $h(k)$.
- Suchen eines Eintrags mit Schlüssel k:
  - Berechne den Hashwert des Schlüssels k: Auswertung von $h(k)$.
  - Gib Eintrag in `values[h(k)]` zurück (`return values[h(k)];`).
  - Aufwand: $O(1)$ bzw. abhängig von der Auswertung von $h(k)$.

2. Versuch: Hashfunktionen

- Damit das Einfügen, Löschen und Suchen so funktioniert, wie auf der vorherigen Folie skizziert, muss $h(K)$ für jeden Schlüssel aus K einen (eindeutigen) Array-Index zwischen 0 und $(size - 1)$ zurückgeben (man sagt dann, h ist eine *perfekte Hashfunktion*).
- Beispiele (Annahme: $|K| = n$):
  - Für ganzzahlige Schlüssel:

    $h : \texttt{int} \rightarrow \{0, \ldots, n-1\}$ mit

    $$h(k) = k \,\texttt{\%}\, n \qquad \text{(Modulo-Operator).}$$

  - Für String-Schlüssel:

    Idee: wandle den `String` in ein **`int`** um, indem die **`int`**-Werte aller Zeichen des `Strings` aufsummiert werden.

    $h : \texttt{String} \rightarrow \{0, \ldots, n-1\}$ mit

    $$h(k) = \Big(\sum_{i=0}^{k.\texttt{length()}-1} k.\texttt{charAt}(i) \Big) \,\texttt{\%}\, n.$$

2. Versuch: Hashfunktionen

- Grundsätzliches Vorgehen für $|K| = n$:
  - Wandle den Schlüssel in eine ganze Zahl um.
  - Berechne Hash-Adresse durch Modulo-Operator (Modulo n).
- In Java gibt es die Methode `hashCode()`, die von der Klasse `Object` an alle Java-Objekte vererbt wird.
- `hashCode()` erzeugt für das aufrufende Objekt einen ganzzahligen Wert.
- Beispiel: Die Methode `hashCode()` wurde in der Klasse `String` überschrieben mit:

$$s.\texttt{hashCode()} = \sum_{i=0}^{s.\texttt{length()}-1} s.\texttt{charAt}(i) \cdot 31^{(s.\texttt{length()}-1-i)}.$$

- Eine Hashfunktion für allgemeine Objekt-Schlüssel ist also:

  $h : \texttt{Object} \rightarrow \{0, \ldots, n-1\}$ mit

  $$h(k) = (k.\texttt{hashCode()}) \,\texttt{\%}\, n.$$

2. Versuch: Problem mit Hashfunktionen

- Frage: Macht eine perfekte Hashfunktion bei sehr großer Schlüsselmenge K Sinn?
- Antwort: in der Regel nicht, daher verwendet man in der Praxis meistens keine perfekten Hashverfahren, sondern arbeitet mit mehreren Einträgen pro Bucket (realisierbar z.B. durch ein weiteres Array pro Bucket). Daraus ergeben sich allerdings weitere Probleme, die wir hier nicht näher betrachten wollen.
- Frage: Was passiert, wenn sich die Schlüsselmenge K ändert (z.B. neue Schlüssel hinzukommen)?
- Antwort: im Falle einer perfekten Hashfunktion benötigen wir plötzlich mehr Buckets als vorhanden, d.h. das Array hat weniger Einträge zur Verfügung als benötigt!
- In diesem Fall muss das zugrundeliegende Array erweitert und alle bisherigen Einträge entsprechend kopiert werden. Meist muss auch die verwendete Hashfunktion h an die neue Kardinalität der Schlüsselmenge angepasst werden.

2. Versuch: Analyse

- Effizienzanalyse (best case):
  - **Einfügen:** Auswerten der Hashfunktion $\Rightarrow O(1)$.
  - **Löschen:** Auswerten der Hashfunktion $\Rightarrow O(1)$.
  - **Zugriff auf Eintrag mit bestimmten Schlüssel:** Auswerten der Hashfunktion $\Rightarrow O(1)$.
- Effizienz im worst case?
- Dynamik:
  Die Implementierung mittels Hashfunktion und Array ist nicht dynamisch. Falls die Schlüsselmenge erweitert wird, muss das Array mit den Einträgen entsprechend erweitert und kopiert werden und die Hashfunktion ggf. angepasst werden.
- Fazit:
  Implementierung ist sehr effizient, aber nicht dynamisch (erfordert erheblichen Mehraufwand bei Erweiterung der Schlüsselmenge).
- Standard-Implementierungen eines assoziativen Speichers in Java basierend auf Hashfunktionen sind `java.util.Hashtable` und `java.util.HashMap`.

Überblick

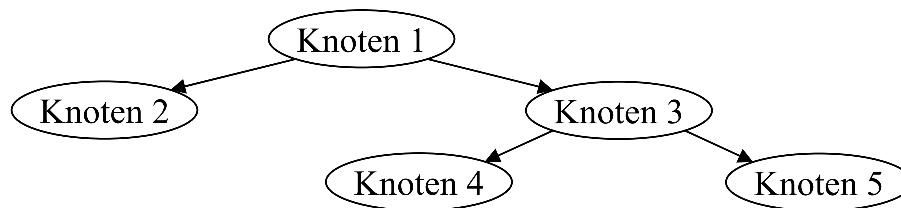
17. Datenstrukturen
17.1 Einleitung
17.2 Listen
17.3 Assoziative Speicher
17.4 Bäume
17.5 Mengen
17.6 Das Collections-Framework in Java
17.7 Zusammenfassung

Motivation

- Bisher hatten wir zwei konträre Ansätze zur Verwaltung von Mengen von Objekten kennen gelernt:
  - Die linearen Listen waren im worst case ineffizient beim Einfügen, Löschen und Suchen, dafür aber dynamisch.
  - Die assoziativen Speicherverfahren (Hashverfahren) waren im best case sehr effizient beim Einfügen, Löschen und Suchen, dafür aber nicht dynamisch.
- Bäume als Datenstruktur stellen nun einen Kompromiss aus den Vor- und Nachteilen beider Konzepte dar: sie sind dynamisch und bieten effizienteres Einfügen, Löschen und Suchen als Listen, allerdings weniger effizienteres als Hashverfahren im best case, aber besser als Hashverfahren im worst case.

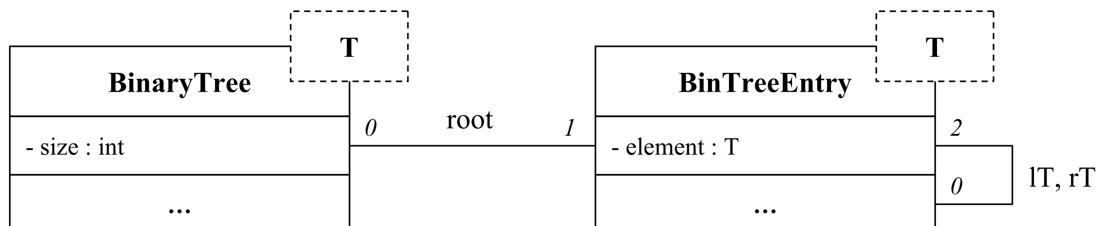
Aufbau

- Bäume organisieren Einträge (im folgenden *Knoten*) nicht mehr linear, sondern *hierarchisch*.
- Induktive Definition eines *binären Baums*:
  - der leere Baum $\langle\rangle$ ist ein binärer Baum
  - sind lT und rT binäre Bäume und K ein Knoten (Eintrag), so ist $\langle K, lT, rT\rangle$ ebenfalls ein binärer Baum.
- Verallgemeinerung: m-äre Bäume: statt 2 Teilbäumen hat jeder Knoten m Teilbäume.
- Beispiel für einen Binärbaum:



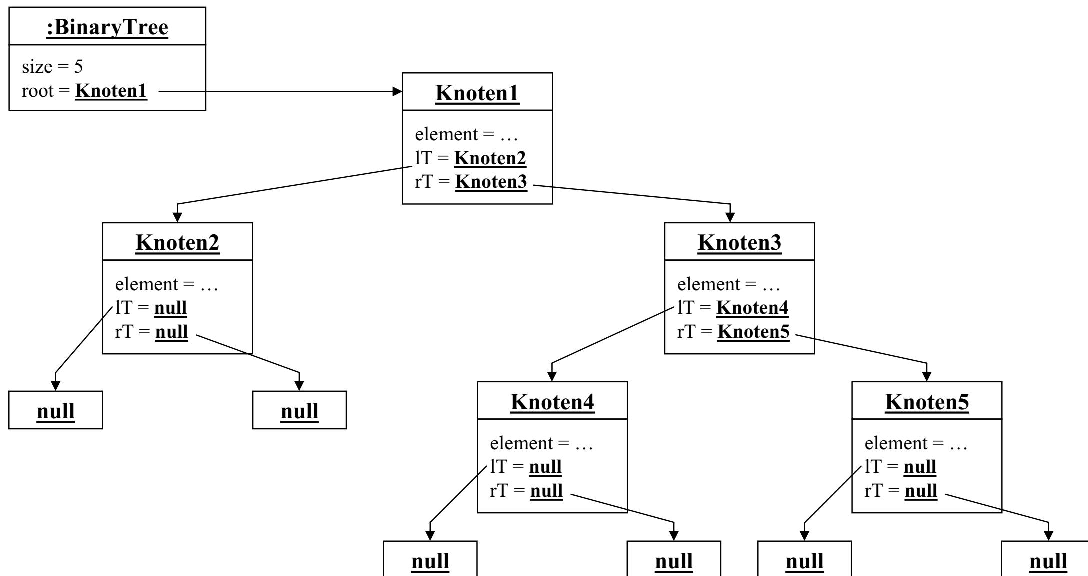
Realisierung

- lT und rT werden auch linker bzw. rechter *Teilbaum* genannt.
- Jeder Koten ist der *Vaterknoten* (*Wurzel*) seiner Teilbäume.
- Knoten, deren linker und rechter Teilbaum leer sind, heißen *Blätter*.
- Der Vaterknoten des gesamten Baumes ist die Wurzel des Baumes.
- Bäume werden ähnlich wie Listen verkettet gespeichert.

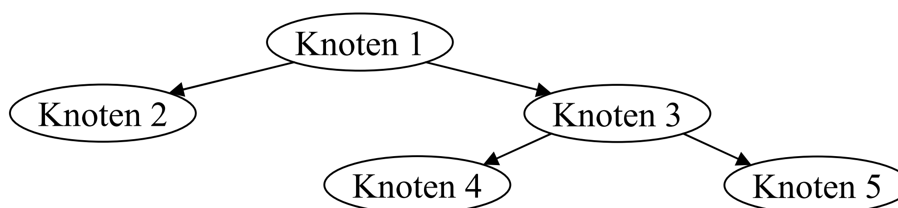


Beispiel

- Der Beispiel-Baum von der vorvorhergehenden Folie:



Baumdurchlauf



- Ein Baum kann in verschiedenen Reihenfolgen durchlaufen werden (z.B. um einen bestimmten Eintrag zu suchen).
- *Präorder*-Reihenfolge: Wurzel – linker Teilbaum – rechter Teilbaum
  Im Beispiel: Knoten 1, Knoten 2, Knoten 3, Knoten 4, Knoten 5
- *Inorder*-Reihenfolge: linker Teilbaum – Wurzel – rechter Teilbaum
  Im Beispiel: Knoten 2, Knoten 1, Knoten 4, Knoten 3, Knoten 5
- *Postorder*-Reihenfolge: linker Teilbaum – rechter Teilbaum – Wurzel
  Im Beispiel: Knoten 2, Knoten 4, Knoten 5, Knoten 3, Knoten 1

Beispiel Präorder-Durchlauf

```
private static void preOrder(BinTreeEntry root)
{
  if(root != null)
  {
    System.out.println(root.getElement());
    preOrder(root.getlT());
    preOrder(root.getrT());
}

public void printTreeInPrOrder()
{
  preOrder(this.root);
}
```


Einfügen, Löschen, Suchen

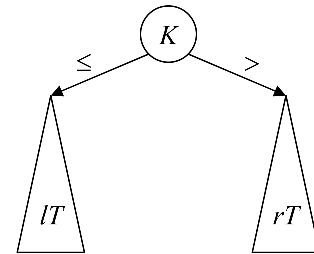
- Bisher unterscheidet sich der Aufbau eines Baumes nicht wirklich von einer Liste:
- Zum Einfügen, Löschen oder Suchen muss im schlechtesten Fall wiederum der gesamte Baum in einer bestimmten Reihenfolge durchlaufen werden $\Rightarrow O(n)$.
- Wie kann man die hierarchische Struktur des Baumes nutzen, um die Effizienz dieser Operationen zu verbessern?
- Idee: Sortiere die Einträge im Baum so, dass eine Art binäre Suche ermöglicht wird, d.h. in jedem Knoten muss eindeutig entscheidbar sein, in welchem Teilbaum die Suche fortgesetzt werden muss.

  $\Rightarrow$ **Suchbäume**

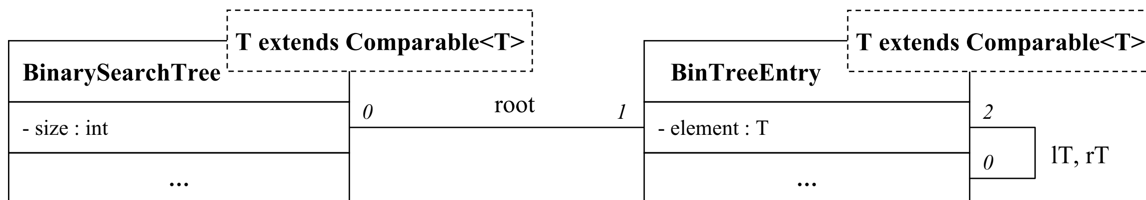
Suchbäume

- Ein (binärer) *Suchbaum* ist ein Binärbaum für dessen Knoten gilt:

  Für jeden Knoten K gilt: alle Schlüssel in lT sind kleiner oder gleich dem Schlüssel in K und alle Schlüssel in rT sind größer als der Schlüssel in K.

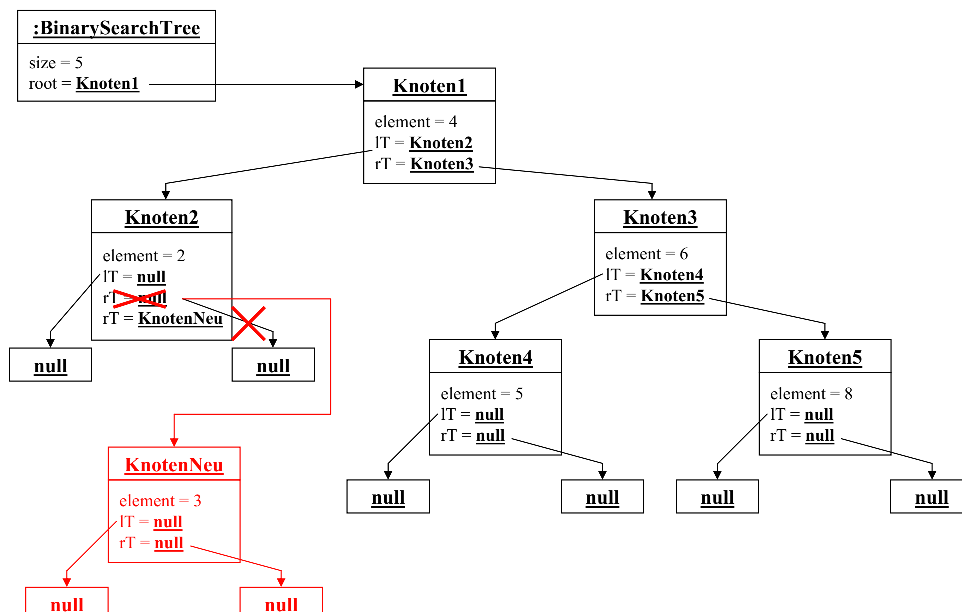


- An der Datenstruktur ändert sich nichts, außer dass der Typ `T` nun natürlich das Interface `Comparable<T>` implementieren muss.



Suchbäume: Einfügen

- Einfügen eines Schlüssels k in einen binären Suchbaum (Annahme: Schlüssel k ist noch nicht im Baum vorhanden):
  - Suche die richtige Einfügestelle (Blatt).
  - Füge neuen Knoten k mit Schlüssel k ein.



Suchbäume: Einfügen (Teil 1)

```
public void insert(T k)

  BinTreeEntry newNode = new BinTreeEntry(k);
  if(this.root == null)
  {
    this.root = newNode;
  }

  // Suche Einfuegestelle
  BinTreeEntry currNode = this.root;
  BinTreeEntry father = null;
  while(currNode != null)
  {
    father = currNode;
    if(newNode.getElement().compareTo(currNode.getElement()) < 0)
    {
      currNode = currNode.getlT();
    }
    if(newNode.getElement().compareTo(currNode.getElement()) > 0)
    {
      currNode = currNode.getrT();
    }
    else
    {
      throw new Exception("Schluessel "+k+" ist bereits vorhanden.");
  }
```

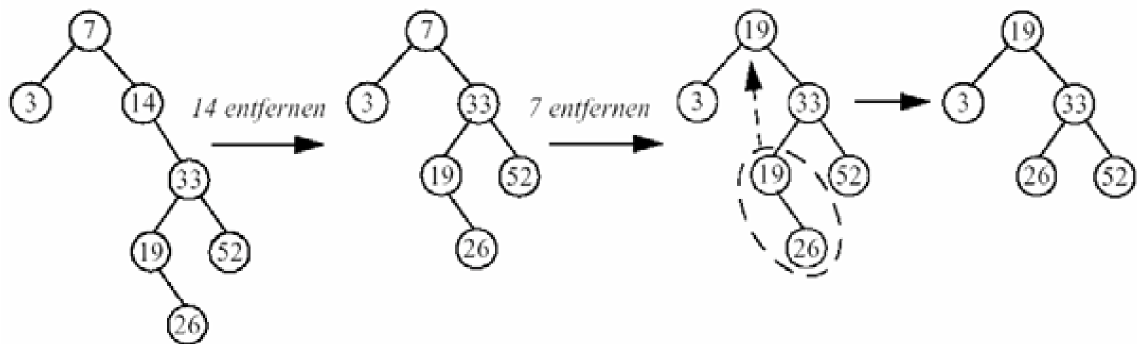

Suchbäume: Einfügen (Teil 2)

```
  // Fuege ein
  if(newNode.getElement().compareTo(father.getElement()) < 0)
  {
    father.setlT(newNode);
  }
  else
  {
    father.setrT(newNode);
  }
  // lT und rT von newNode sind null

}
```

Suchbäume: Löschen

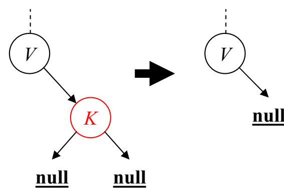
- Löschen eines Knoten *K* mit Schlüssel *k* in einen binären Suchbaum:
  - Suche Knoten *K*.
  - Lösche *K*.
- Löschen ist etwas komplizierter, da nun auch *innere Knoten* (Knoten, die nicht Blätter sind) entfernt werden können:



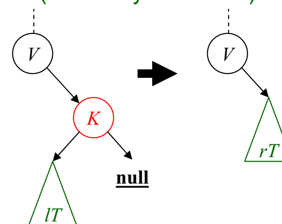
Suchbäume: Löschen

- Fall 1: Knoten *K* besitzt höchstens einen Sohn (Blatt oder Halbblatt)

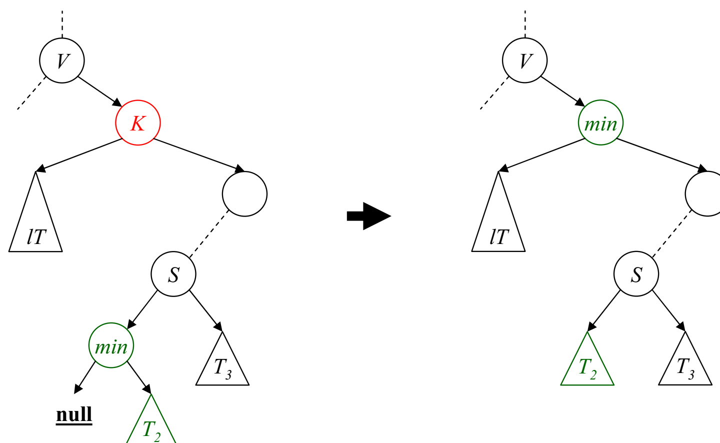
Fall 1.1 *K* ist Blatt



Fall 1.2 *K* hat linken Teilbaum (rechts: symmetrisch)



- Fall 2: Knoten *K* besitzt zwei Söhne (innerer Knoten)



Suchbäume: Suchen

- Suchen eines Schlüssels k in einen binären Suchbaum:
  - Wenn die Wurzel R leer ist: Suche erfolglos beendet.
  - Vergleiche k mit dem Schlüssel der Wurzel R.
  - Bei Gleichheit: Suche erfolgreich beendet.
  - Wenn $k < R$.Schlüssel: suche rekursiv in lT nach k.
  - Wenn $k > R$.Schlüssel: suche rekursiv in rT nach k.
- Laufzeit: $O(h)$, wobei h die Höhe des Baumes ist.
- Höhe eines binären Suchbaums:
  - Worst-Case: Baum entartet zur linearen Liste $\Rightarrow O(n)$.
  - Im Durchschnitt (alle Permutationen der Einfüge-Reihenfolge sind gleichwahrscheinlich): $O(\log n)$.

Suchbäume: Suchen

```
public BinTreeEntry suche(T k)
{
  return suche(this.root, k);
}

private BinTreeEntry suche(BinarySearchTree tree, T key)
{
  if(tree.root == null)
  {
    return null;
  }
  if(key.compareTo(tree.root.getElement()) == 0)
  {
    return tree.root;
  }
  if(key.compareTo(tree.root.getElement()) < 0)
  {
    return suche(tree.root.getlT(), key);
  }
  else
  {
    return suche(tree.root.getrT(), key);
  }
}
```

Suchbäume: Analyse

- Effizienzanalyse:
  - **Einfügen:** Suche nach Einfügestelle $\Rightarrow$ im Durchschnitt $O(\log n)$, im worst-case $O(n)$.
  - **Löschen:** Suche nach zu löschendem Eintrag $\Rightarrow$ im Durchschnitt $O(\log n)$, im worst-case $O(n)$.
  - **Zugriff auf Eintrag mit bestimmten Schlüssel:** Suche im Durchschnitt $O(\log n)$, im worst-case $O(n)$.
- Dynamik:
  Die Implementierung ist dynamisch wie verkettete Listen.
- Fazit:
  Kompromiss zwischen (zumindest theoretisch) effizientem Hashing und dynamisch verketteter Liste.
- Eine Standard-Implementierung eines Suchbaumes als assoziativer Speicher ist `java.util.TreeMap`.