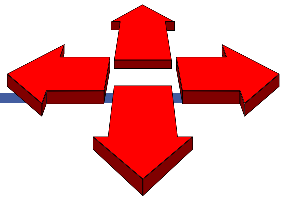
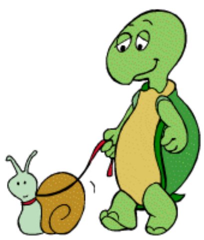
Sammlungen implementieren II: Mengen



- Aufwand für Operationen auf Mengen?
- Bäume als Realisierung für Mengen:
  - binäre Bäume
  - Suchbäume
  - balancierte Bäume
- Hash-Verfahren als Realisierung für Mengen:
  - Hash-Tabelle
  - Hash-Funktion

Einfügen in Mengen: hoher Aufwand?

- Wenn ein Element in eine Menge eingefügt werden soll, muss die Implementierung der Menge prüfen, ob das Element ein Duplikat ist.
- Es ist also bei jedem Einfügen ein Test auf Enthaltensein nötig.
- Würde die Menge intern als einfache Liste implementiert, würde der Aufwand für das Einfügen linear von der Größe der Menge abhängen (***O(n)***).
- Bei großen Mengen ist das nicht akzeptabel.
  - ➔ Wir brauchen etwas Schnelleres.

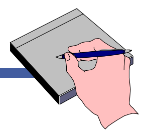


Axel Schmolitzky, Heinz Züllighoven, et al.

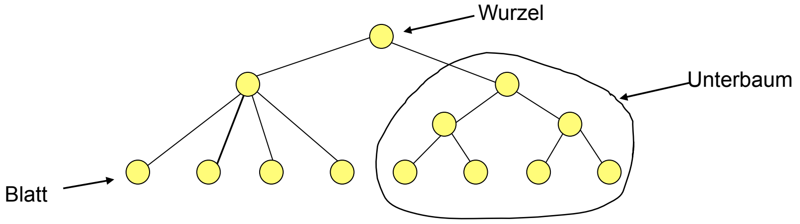
Effiziente Suchverfahren: Anforderungen

- Der **Test auf Enthaltensein** wird häufig auch als **Suche** bezeichnet: Wir suchen ein Element in einer Sammlung; wenn wir es finden, ist das Testergebnis positiv.
- Um nicht jedes Element in der Menge mit dem zu suchenden Element vergleichen zu müssen (linearer Aufwand, ***O(n)***), müssen wir die Elemente geeignet strukturieren.
- Voraussetzung dafür ist, dass die Elemente bestimmte Eigenschaften haben. Zwei typische Anforderungen an Elemente sind, dass sie
  - **sortierbar** oder
  - **kategorisierbar** sind.
- Sortierbare Elemente ermöglichen eine **binäre Suche** oder eine Realisierung mit einem **Suchbaum**.
- Kategorisierbare Elemente ermöglichen **Hash-Verfahren**.

Bäume

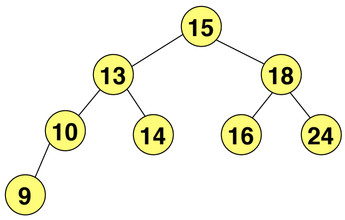


- Ein **Baum** (engl.: tree) ist eine Struktur, in der Knoten miteinander durch (gerichtete) Kanten verbunden sind.
- Jeder Knoten kann beliebig viele **Kindknoten** (Nachfolger) haben, mit denen er über Kanten verbunden ist. Ein Knoten hat aber immer **maximal einen** Vorgängerknoten.
- Ein Knoten ohne Vorgänger heißt **Wurzel** des Baumes.
- Ein Knoten ohne Kindknoten wird als **Blatt** bezeichnet.
- Bäume sind **rekursiv**: Ein Kindknoten kann wieder als ein Wurzelknoten eines kleineren Baumes angesehen werden, der als **Unterbaum** bezeichnet werden kann.



Binäre Suchbäume

- In einem **binären Baum** hat ein Knoten maximal zwei Kindknoten.
- Wenn jeder Knoten ein sortierbares Element enthält und gilt, dass im linken Unterbaum alle „kleineren“ und im rechten Unterbaum alle „größeren“ Element liegen, dann wird der Baum zu einem **binären Suchbaum**.

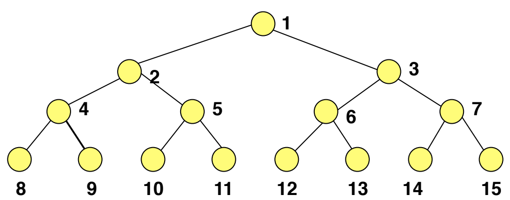


Anordnung der Menge
M = { 9 10 13 14 15 16 18 24 }
als binärer Suchbaum

Merkmale von binären Bäumen

- Viele Eigenschaften von (binären) Bäumen beziehen sich auf die Anzahl der Knoten und Blätter sowie die Höhe eines Baums.
- Beispiele für Eigenschaften:
  - Ein voller binärer Baum mit Höhe h hat 2^h Blätter.
  - Die Zahl der Knoten eines vollen binären Baums mit Höhe h ist $2^{h+1} - 1$

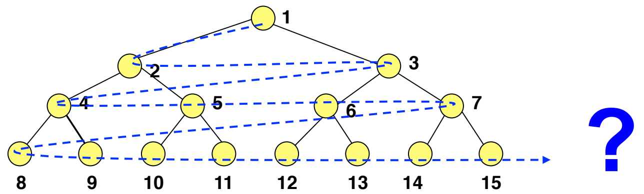
Höhe h= 0 — **2^0 Knoten**
1 — $2^0 + 2^1$
2 — $2^0 + 2^1 + 2^2$
3 — $2^{h+1} - 1$



nach © Neumann

Traversieren von Bäumen

- Für viele Anwendungen müssen alle Knoten eines Baumes der Reihe nach bearbeitet werden. Dazu muss ein Ordnungsprinzip gewählt werden, um die Knoten eines Baumes zu „linearisieren“.
- Die bekanntesten Traversierungsstrategien sind:
  - Breitendurchlauf
  - Tiefendurchlauf

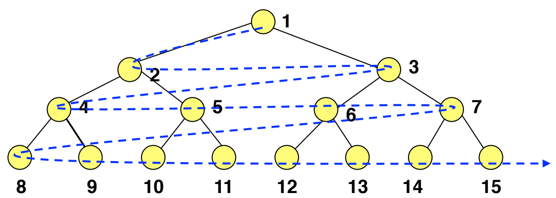


nach © Neumann

Breitendurchlauf

Breitendurchlauf (level-order tree traversal)

- Die Idee:
  Verfahren, um einen Baum "schichtenweise" abzuarbeiten,
  z.B. bei der Suche nach Zielknoten mit minimalem Abstand zur Wurzel.
- Strategie:
  - Breite vor Tiefe,
  - links vor rechts



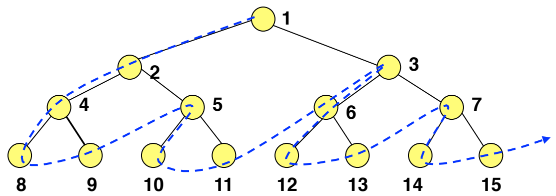
`[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]`

nach © Neumann

Tiefendurchlauf

Tiefendurchlauf

- Die Idee:
  Allgemeines Verfahren, um einen Baum "astweise" in Richtung seiner Blätter abzuarbeiten,
  z.B. bei der Suche nach Planschritten, die zu einem Ziel führen sollen.
- Strategie:
  - Tiefe vor Breite,
  - links vor rechts

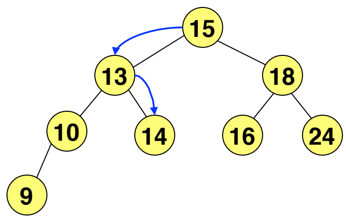


`[1,2,4,8,9,5,10,11,3,6,12,13,7,14,15]`

nach © Neumann

Suchalgorithmus für binäre Suchbäume

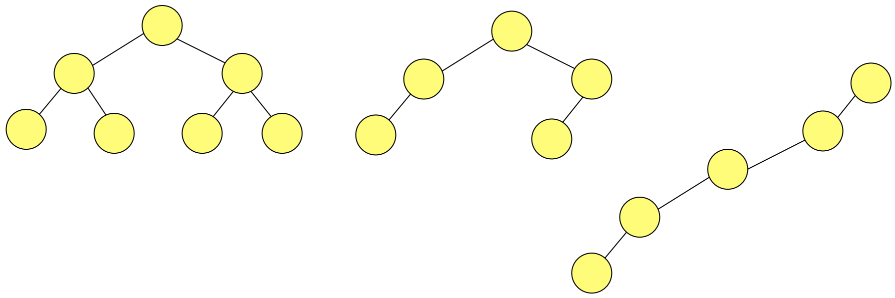
- Die Suche nach einem Element ist (rekursiv) folgendermaßen möglich:
  - Ist der Baum leer?
    - Ja → **Suche erfolglos**
    - Nein: Enthält der Wurzelknoten das gesuchte Element?
      - Ja → **Suche erfolgreich**
      - Nein: Ist das gesuchte Element kleiner als des aktuelle Element?
        - Ja: **Weitersuchen** im linken Teilbaum
        - Nein: **Weitersuchen** im rechten Teilbaum



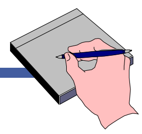
Suche nach k = 14

Bäume können *degenerieren*

- Damit sind wir fast am Ziel: Bei einem „gutmütigen“ Baum ist die Suche recht schnell.
- Ein binärer Baum kann jedoch „degenerieren“: Eine verkettete Liste kann als ein verkümmerter binärer Baum mit jeweils einem Kindknoten gesehen werden.
- Es fehlt noch eine entscheidende Bedingung...



Balancierte binäre Suchbäume

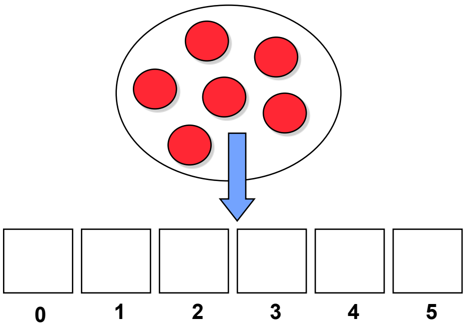


- In einem **balancierten binären Baum** gilt für jeden Knoten, dass die **Höhen** seiner beiden Unterbäume sich **maximal um eins** unterscheiden.
- Die Höhe ***h*** eines solchen Baumes berechnet sich dann logarithmisch aus der Anzahl ***n*** der Knoten im Baum:
  - ***h = ld(n)*** (ld ist hier der Logarithmus Dualis, also der Logarithmus zur Basis 2)
- Unsere **Suche** muss von der Wurzel bis zu jedem Blatt dann höchstens ***ld(n)* Vergleiche** vornehmen; der Aufwand ist damit in seiner Größenordnung ***O(log(n))***.
- Die Baum-Implementationen des JCF (`TreeSet` und `TreeMap`) benutzen binäre, balancierte Bäume. Sie setzen jedoch voraus, dass die **Elemente sortierbar** sind (sie müssen das Interface `Comparable` implementieren).

- Es geht aber noch schneller; und das sogar ohne die Notwendigkeit, dass die Elemente sortierbar sind!

Hash-Verfahren: Die Grundidee

- Die Grundidee von **Hash-Verfahren** (auch: Hashing) ist, Elemente auf eine Indexstruktur abzubilden. Dabei soll sich aus einem Element **unmittelbar** sein Index **berechnen** lassen.

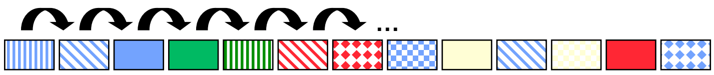


nach © Budd

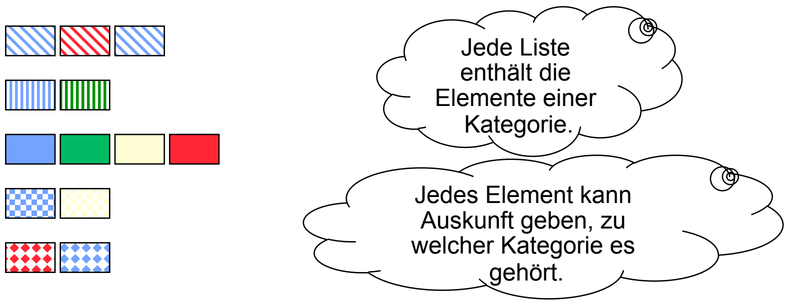
SE1 – Level 4 13

Lösungsansatz für Hashing: mehrere Listen statt einer

- Statt eine Liste komplett zu durchsuchen:



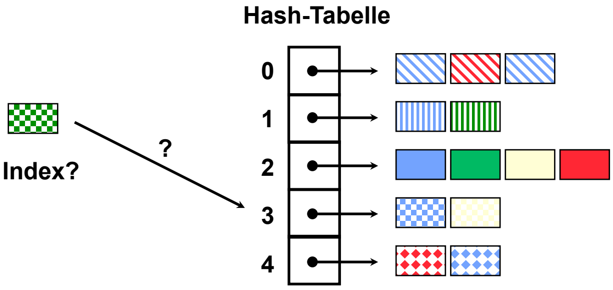
- Mehrere Listen + Wissen, in welcher gesucht werden muss!



SE1 – Level 4 14

Im Kern: die Hash-Tabelle

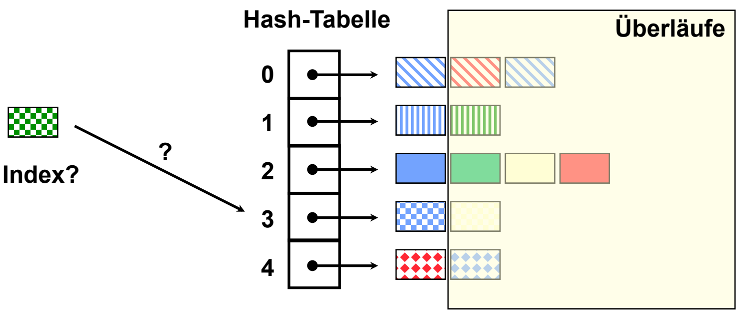
- Im Kern eines Hash-Verfahrens steht die sogenannte **Hash-Tabelle** (meist als **Array** realisiert). Sie kann verstanden werden als eine Tabelle von (möglichst kurzen) Listen.
- Für ein zu suchendes Element wird zuerst der Index der Liste in der Tabelle ermittelt, in der Elemente der gleichen Kategorie liegen.
- Nach einem (schnellen) indexbasierten Zugriff auf die Liste wird diese dann (linear) durchsucht.



SE1 – Level 4 15

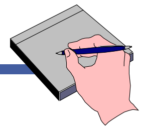
Ziel: möglichst wenige Überläufe

- Ideal für ein Hash-Verfahren ist, wenn die Listen maximal ein Element enthalten; nach der Indexberechnung ist dann für eine Suche **maximal ein Vergleich** notwendig!
- Enthält eine Liste mehr als ein Element, so werden die überschüssigen Elemente als **Überläufe** bezeichnet.
  - In einigen Darstellungen werden die Listen deshalb auch als Überlaufbehälter (engl.: bucket) bezeichnet.

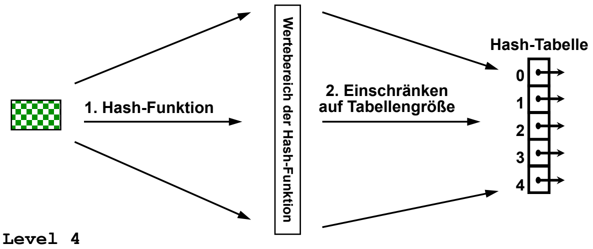


SE1 – Level 4 16

Entscheidend: Die Hash-Funktion



- Die Basis des Verfahrens ist die **Hash-Funktion**:
  - Sie bildet ein **Element** auf einen **ganzzahligen Wert** ab, also auf einen Integer-Wert.
  - Der berechnete Wert bildet eine **künstliche Kategorie**: Alle Elemente mit demselben Wert fallen in dieselbe Kategorie.
  - Wenn die Hash-Funktion zwei verschiedene Elemente auf denselben Wert abbildet, wird dies als **Kollision** bezeichnet.
  - Der berechnete Wert muss in einem zweiten Schritt auf einen **Index** in der Hash-Tabelle abgebildet werden.
- Entscheidend ist die **Güte** der Hash-Funktion: Je gleichmäßiger sie die Elemente in der Tabelle verteilt, desto schneller ist das Verfahren.



Beispiel: Hash-Funktion mit Kollision

Beispiel:
Einfügen von Monatsnamen in eine Hash-Tabelle mit 12 Positionen.

$c_1 \ldots c_k$ Monatsname als Zeichenkette
$N(c_k)$ Binärdarstellung eines Zeichens

Hashfunktion h mit m=12 bildet Zeichenketten in Indizes 0 ... 11 ab:

$$h(c_1 \ldots c_k) = \sum_{i=1}^{k} N(c_i) \bmod m$$

Verteilung der Namen mit Kollisionen:

0	November	6	Mai, September
1	April, Dezember	7	Juni
2	März	8	Januar
3	-	9	Juli
4	August	10	-
5	Oktober	11	Februar

Eine **ideale Hash-Funktion** bildet alle Schlüssel eins-zu-eins auf unterschiedliche Integerwerte ab, die fortlaufend sind und bei Null beginnen.

nach © Neumann

Hash-Verfahren im Java Collections Framework

- Die Implementation `HashSet` für das Interface `Set` im JCF basiert auf einem Hash-Verfahren.
- Als Basis für die Hash-Funktion wird das Ergebnis der Operation `hashCode` verwendet, die in der Klasse `Object` und damit für alle Objekte definiert ist.
- **Vorsicht:** Wenn für eine Klasse die Operation `equals` redefiniert wird, dann muss garantiert sein, dass für zwei Exemplare dieser Klasse, die nach der neuen Definition gleich sind, auch die Operation `hashCode` den gleichen Wert liefert!
  - Es besteht sonst die Möglichkeit, dass in ein `HashSet` Duplikate eingetragen werden können, weil die beiden Elemente in verschiedenen Überlaufbehältern landen.
  - Die Spezifikation von `Set` würde damit nicht eingehalten, weil das Implementationsverfahren zufällig auf Hashing basiert!

Vorsicht, die zweite: hashCode sollte **niemals** basierend auf veränderlichen Exemplarvariablen implementiert werden!

Indexberechnung für die Hash-Tabelle

- **Und nochmal aufgepasst** : Der Wertebereich einer Hash-Funktion erlaubt üblicherweise auch negative Werte; in der Hash-Tabelle (einem Array) kann aber nur mit positiven Werten indiziert werden!
- Beim Abbilden des Hash-Wertes auf einen gültigen Index muss deshalb ein eventuell vorhandenes **negatives Vorzeichen entfernt** werden. Dazu gibt es mehrere Möglichkeiten:
  - Maskieren des Vorzeichens über eine Bitmaske
  - Rechtsschieben der Bits des Hash-Wertes (Eliminieren des LSB)
  - In Java: Anwenden von `Math.abs()`
    - Sonderfall beachten: Im Zweierkomplement ist der Betrag der kleinsten negativen Zahl immer um Eins größer als die größte positive Zahl. Math.abs liefert bei der kleinsten negativen Zahl deshalb das Argument als Ergebnis, also wieder eine negative Zahl!
    - Der String „hochenwiseler“ beispielsweise liefert in Java `Integer.MIN_VALUE` als Hash-Wert.

Anfangskapazität & Befüllungsgrad

- Dynamische Größenanpassung
  - Die Hash-Verfahren im JCF passen die **Größe der Hash-Tabelle dynamisch** an (ähnlich wie bei wachsenden Arrays).
  - Auf diesen Prozess kann mit zwei Konstruktorparametern Einfluss genommen werden:
    – der **Anfangskapazität** (initial capacity) als Größe der Tabelle
    – dem **Befüllungsgrad** (load factor)
  - Die Hash-Tabelle wird mit der Anfangskapazität angelegt. Sobald mehr Elemente eingefügt sind als die aktuelle Kapazität multipliziert mit dem Befüllungsgrad, wird die Kapazität erhöht (Aufruf der privaten Methode **`rehash`**).

`Set`-Implementierungsvarianten im JCF

TreeSet

- **Balancierter Binärer Suchbaum**
- Einfügen und Entfernen durch Baumsuche in ***O(log n)***.
- Reorganisation bei Ungleichgewichten durch Knotenumordnung ist akzeptabel schnell, kommt aber oft vor.

HashSet

- **Hash-Verfahren** mit dynamischer Anpassung der Hash-Tabelle
- Einfügen und Entfernen in **konstanter Zeit**.
- Reorganisation nach Erreichen des Load-Factors ist durch komplettes Rehashen teuer!
- Geschwindigkeit hängt auch von der Güte der Hash-Werte ab.

Zusammenfassung

- Bei **Mengen** ist der Test auf Enthaltensein (**Suche**) typischerweise die wichtigste Operation.
- Mit einer naiven Implementation ist der Such-Aufwand ***O(n)***.
- Bei einem **balancierten binären Suchbaum** reduziert sich der Such-Aufwand auf ***O(log(n)).***
- Die `Set`-Implementation `TreeSet` des JCF erfordert, dass die Elemente das Interface `Comparable` implementieren.
- Mit einem guten **Hash-Verfahren** kann eine Suche in ***O(1)*** durchgeführt werden.
- Die `Set`-Implementation `HashSet` des JCF implementiert ein dynamisches Hash-Verfahren.

