Software-Entwicklung 1

V12: Listen- und Mengenimplementierung

Prof. Maalej & Team @maalejw

Überblick

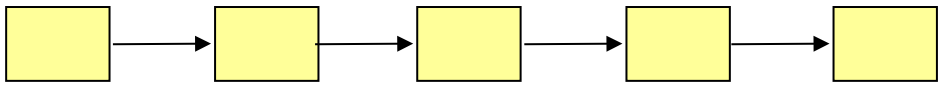
1. **Listenimplementierung**
   - **Einfach verkettet**
   - **Doppelt verkettet**
2. **Mengenimplementierung**
   - **Bäume**
   - **Hashing**

Von Sammlungen zu dynamischen Datenstrukturen

- Sammlungen sind bisher wie Mengen und Listen:
  - Elemente können hinzugefügt und entnommen werden
  - Unterschiedliche Organisationsprinzipien

- Wir klären im Weiteren die Begriffe
  - „dynamisch“
  - „Datenstruktur“

- Einstieg in die wichtigsten dynamischen Datenstrukturen mit spezifischen Stärken und Schwächen



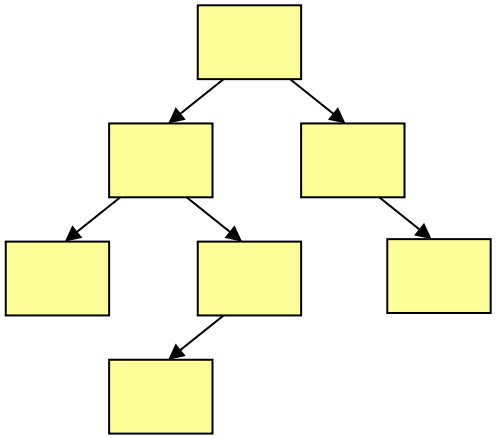
Implementationen für Sammlungen

- Für die Collection Interfaces (List, Set) stehen einige Implementationen zur Verfügung
- Implementationen basieren auf zwei grundlegenden Programmierkonstrukten:
  - Arrays
  - verkettete Strukturen
- Insgesamt können im JCF vier Implementierungskonzepte unterschieden werden:
  - verkettete Listen
  - wachsende Arrays
  - balancierte Bäume
  - Hash-Verfahren

Dies sind Beispiele für dynamische Datenstrukturen!

Dynamische Datenstrukturen

- Dynamische Datenstrukturen bezeichnen die **Organisationsform** von **veränderbaren** Sammlungen von Objekten



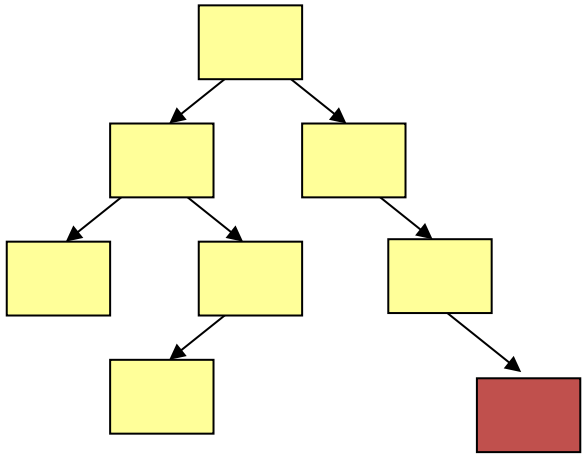
- Eine Struktur ist ein
  - Gebilde aus Elementen (Objekten)
  - mit Beziehungen (Relationen)

- Dynamische Datenstrukturen sind meist gleichartig rekursiv aufgebaut

Strukturen verändern

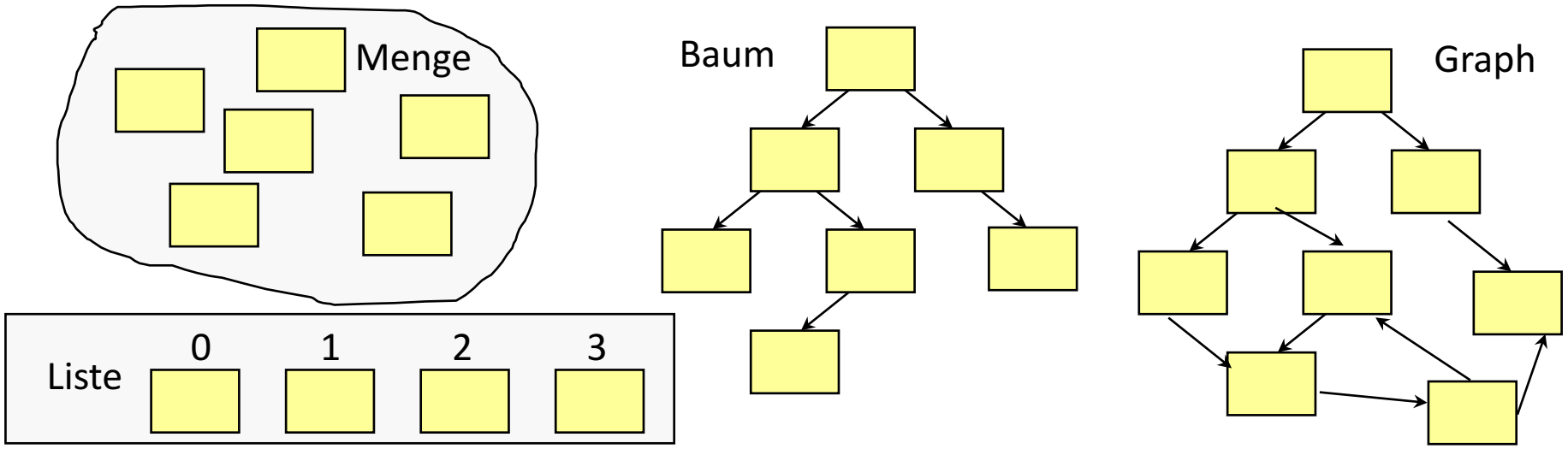
- Ändern einer Struktur bedeutet
  - Hinzufügen, Modifizieren und Löschen von Elementen
  - Ändern von Beziehungen

- Datenstrukturen sind dynamisch, wenn sie durch das Einfügen und Entfernen ihrer Elemente wachsen und schrumpfen



Einteilung dynamischer Datenstrukturen

- Einteilung nach grundlegender Struktur
- Wir unterscheiden:
  - Strukturen von Elementen **ohne Relation** zueinander (z.B. für Mengen)
  - **Lineare** oder sequenzielle Strukturen (z.B. für Listen)
  - Bäume, in denen ein Element **ein Vorgängerelement** aber **mehr** als ein **Nachfolgerelement** haben kann
  - Graphen, in denen ein Element **beliebig** mit anderen Elementen **verbunden** sein kann

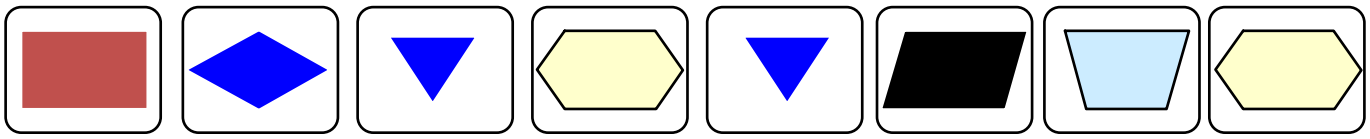


Sammlungen als Listen implementieren

- Listen sind die grundlegendste (elementare) sequenzielle Struktur

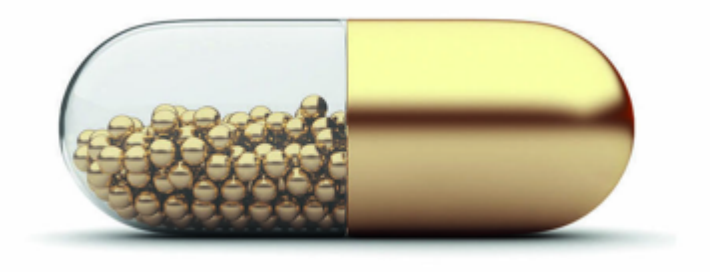
- Weitere lineare Sammlungsarten wie Stacks und Queues bauen darauf auf

- Zwei Implementationsformen:
  - eine basierend auf Verkettung
  - eine basierend auf Arrays



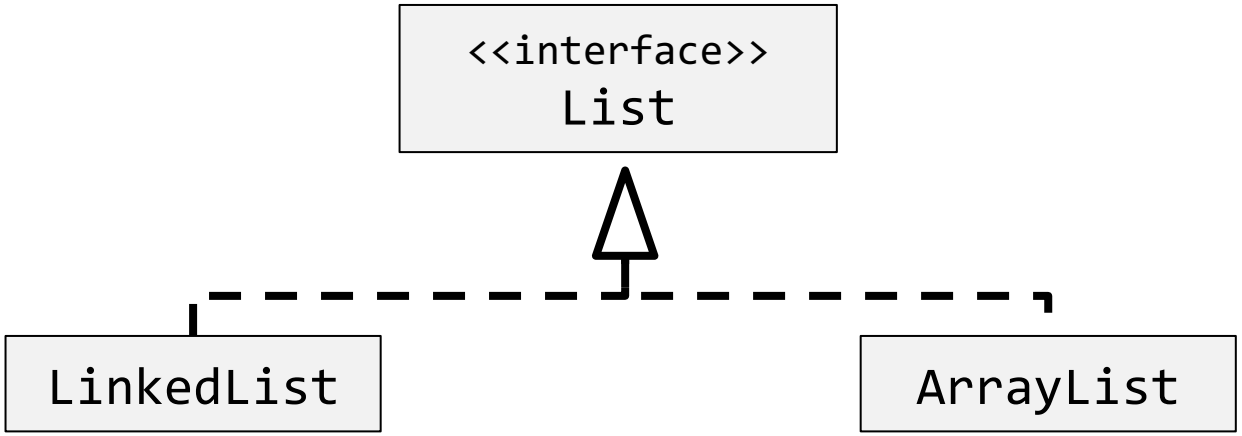
Klientensicht auf Listen

- Relevant für den Klienten:
  - Beinhaltet beliebig viele Elemente
  - Zugriff über den Index auf ein beliebiges Element
  - Einfügen eines Elements erhöht den Index der nachfolgenden Elemente
  - Entfernen eines Elements verringert den Index der nachfolgenden Elemente
  - Testen ob ein gegebenes Element bereits in der Liste enthalten ist



Listenimplementationen

- Liste ist im JCF mit dem Interface List modelliert
- Das JCF bietet zwei Implementationen für dieses Interface:
  - LinkedList
  - ArrayList
- LinkedList basiert auf dem Konzept verkettete Liste
- ArrayList basiert auf dem Konzept wachsender Arrays



Lineare Datenstrukturen für Listen: Verkettung

- Wir haben die Liste als Sammlung kennen gelernt, deren Wertemenge Elemente als endliche Folgen eines Grundtyps umfasst:
  - Listenelemente besitzen eine Reihenfolge
  - Elemente des Grundtyps können mehrfach enthalten sein (Duplikate)

- Eine verkettete Liste als Struktur betrachtet ist eine Sequenz ihrer Elemente:
  - Jedes Listenelement ist mit dem nächsten verbunden
  - Vom Anfang zum Ende der Liste, muss jedes Element traversiert werden

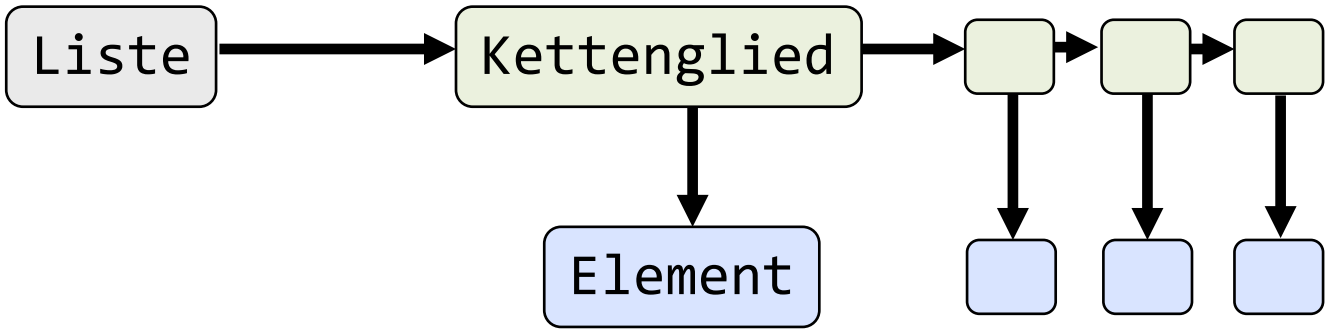
Einfach und doppelt verkettete Listen

- **Einfach verkettete Liste:**
  - Jedes Listenelement hat nur eine Referenz auf sein Nachfolgerelement
  - Die Liste kann nur elementweise vom Anfang zum Ende traversiert werden

- **Doppelt verkettete Liste:**
  - Jedes Listenelement hat eine Referenz auf sein Nachfolger- und sein Vorgängerelement
  - Die Liste kann elementweise in beide Richtungen traversiert werden

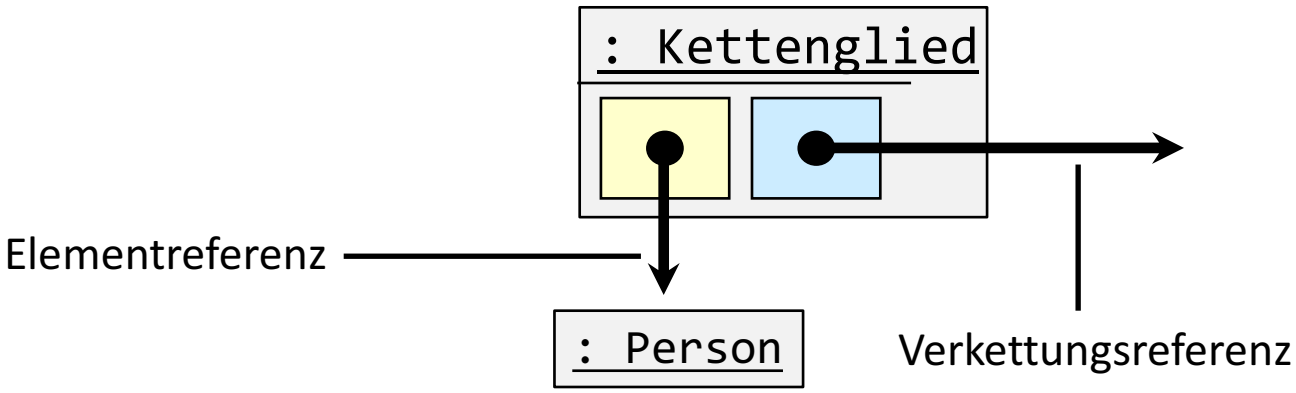
Objektorientierter Entwurf einer Liste

- Ein Listenobjekt, das die Liste für den Klienten repräsentiert
- Objekte als Kettenglieder, die die Verkettung der Liste realisieren (unsichtbar für den Klienten)
- Objekte, die als Elemente in der Liste gespeichert sind (verwaltet über die Umgangsformen der Liste)



Komposition eines Kettenglied Objektes

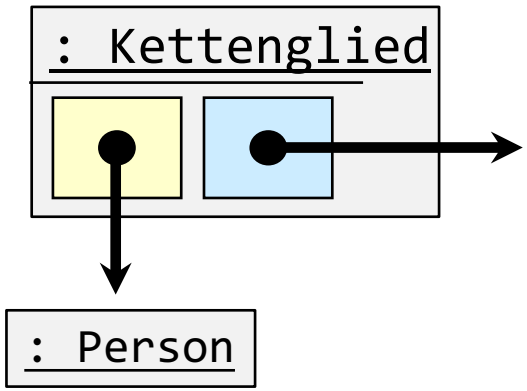
- Jedes Kettenglied ist ein eigenes Objekt (Klasse `Kettenglied`)
- Jedes Kettenglied hält eine Referenz auf das eigentliche Element der Sammlung
- Ein Kettenglied hält außerdem mindestens eine Referenz auf ein weiteres Kettenglied (das Nachfolgerelement) als Verkettungsreferenz



Klassendefinition für Kettenglieder

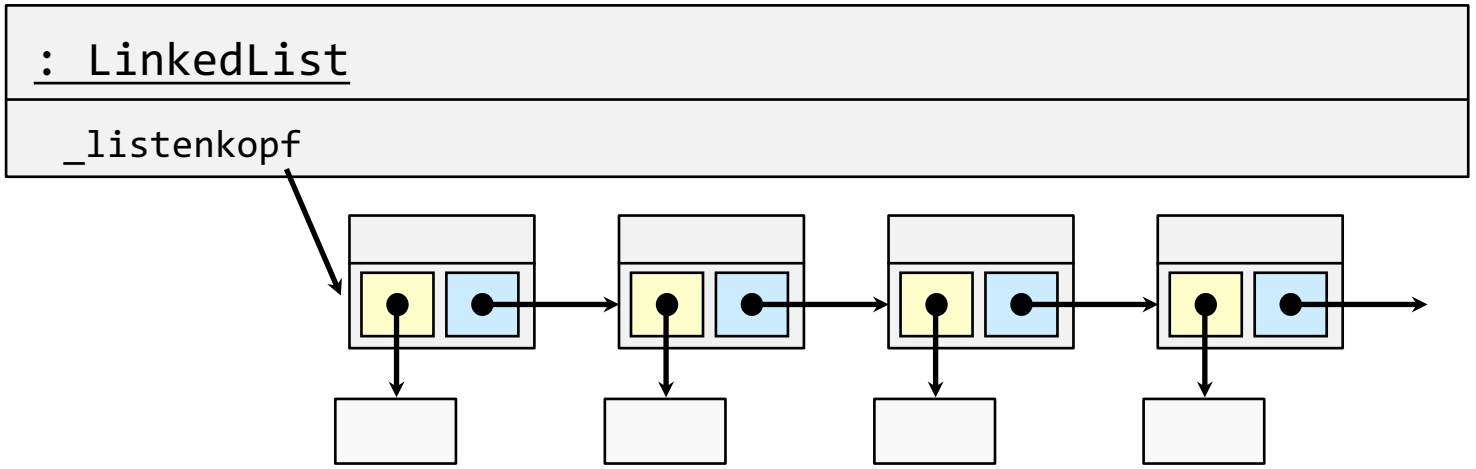
- Die Exemplarvariable hat den gleichen Typ wie die definierende Klasse
- Wird auch strukturelle Rekursion genannt

```
class Kettenglied {
    private Kettenglied _nachfolger;
    private Person _element;
    ...
}
```



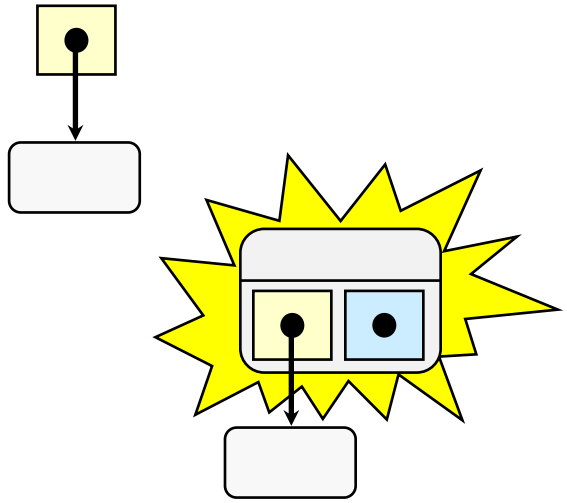
Konstruktion einer einfach verketteten Liste

- Die Liste selbst ist ein Exemplar einer eigenen Klasse z.B. `LinkedList`
- Die Kettenglieder (Exemplare von `Kettenglied`) bilden die innere Struktur der Liste
- Die enthaltenen referenzierten Elemente werden üblicherweise nicht als Teil der Liste angesehen
- In der Klasse LinkedList wird üblicherweise die Referenz auf das erste Kettenglied gehalten („Listenkopf“)

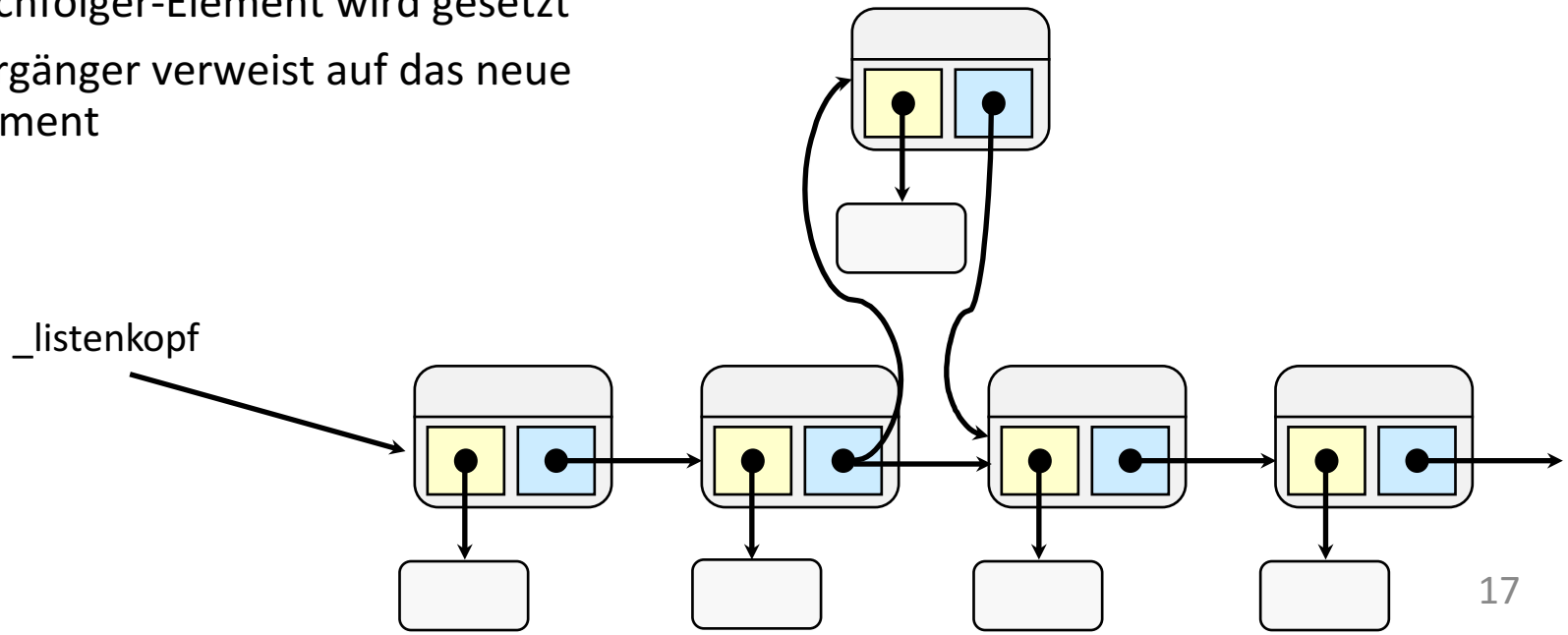


Einfügen in eine verkettete Liste

- Gegeben: (eine Referenz auf) ein neues Element

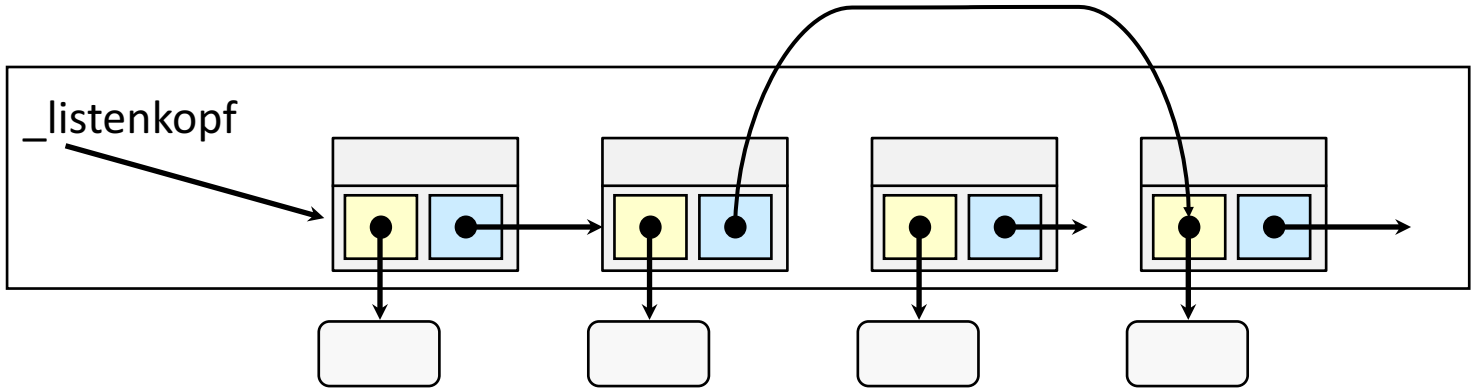


- Erzeugung eines neuen Kettenglieds
- Das neue Kettenglied wird an die gewünschte Stelle „eingekettet“:
  - Nachfolger-Element wird gesetzt
  - Vorgänger verweist auf das neue Element



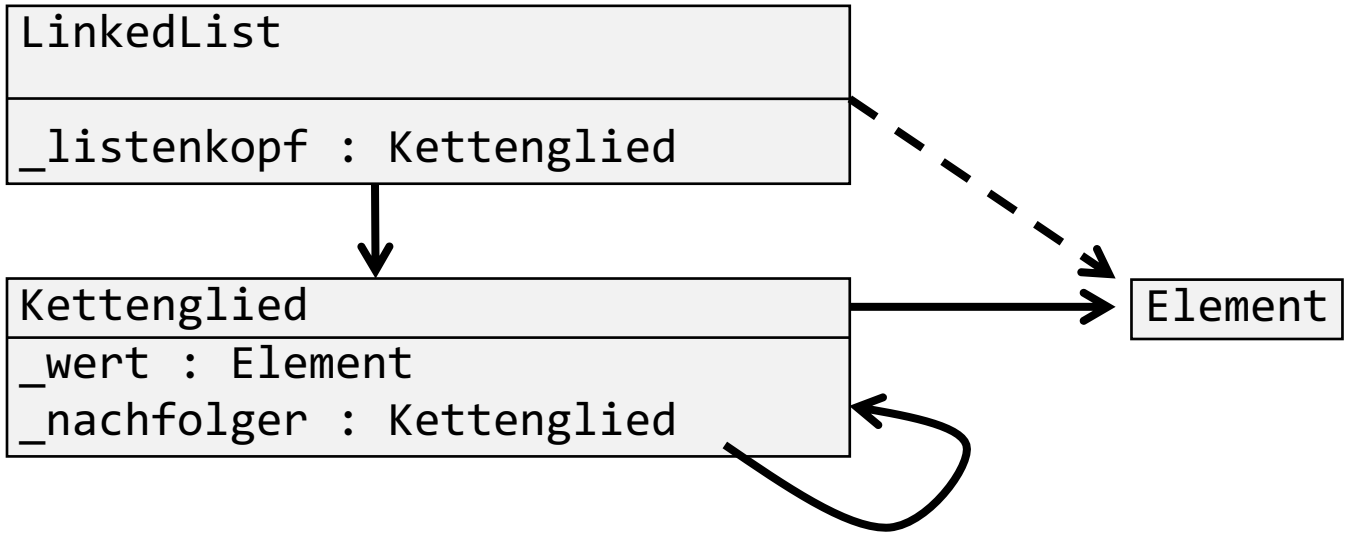
Entfernen aus einer verketteten Liste

- Die Referenz des Vorgängers wird auf den Nachfolger umgebogen
- Das Kettenglied wird dann vom Garbage-Collector entfernt, sobald keine Referenz mehr darauf existiert



Klassendiagramm einer einfach verketteten Liste

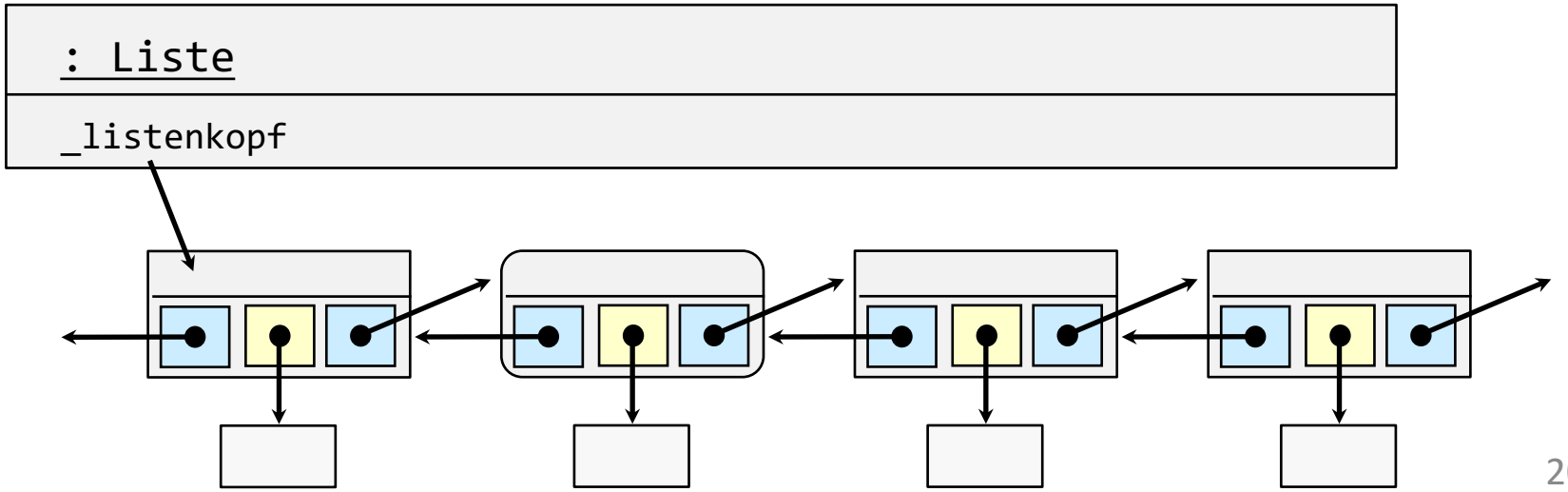
- **`LinkedList`**
  - Hält einen Verweis auf die Klasse Kettenglied im Attribut _listenkopf
  - Benutzt die Klasse Element in den Parametern ihrer Methoden



- **`Kettenglied`**
  - Speichert jeweils ein Exemplar der Klasse Element im Attribut _wert
  - Kettenglied verweist auf sich selbst, um im Attribut _nachfolger das nächste Kettenglied referenzieren zu können

Doppelt verkettete Liste

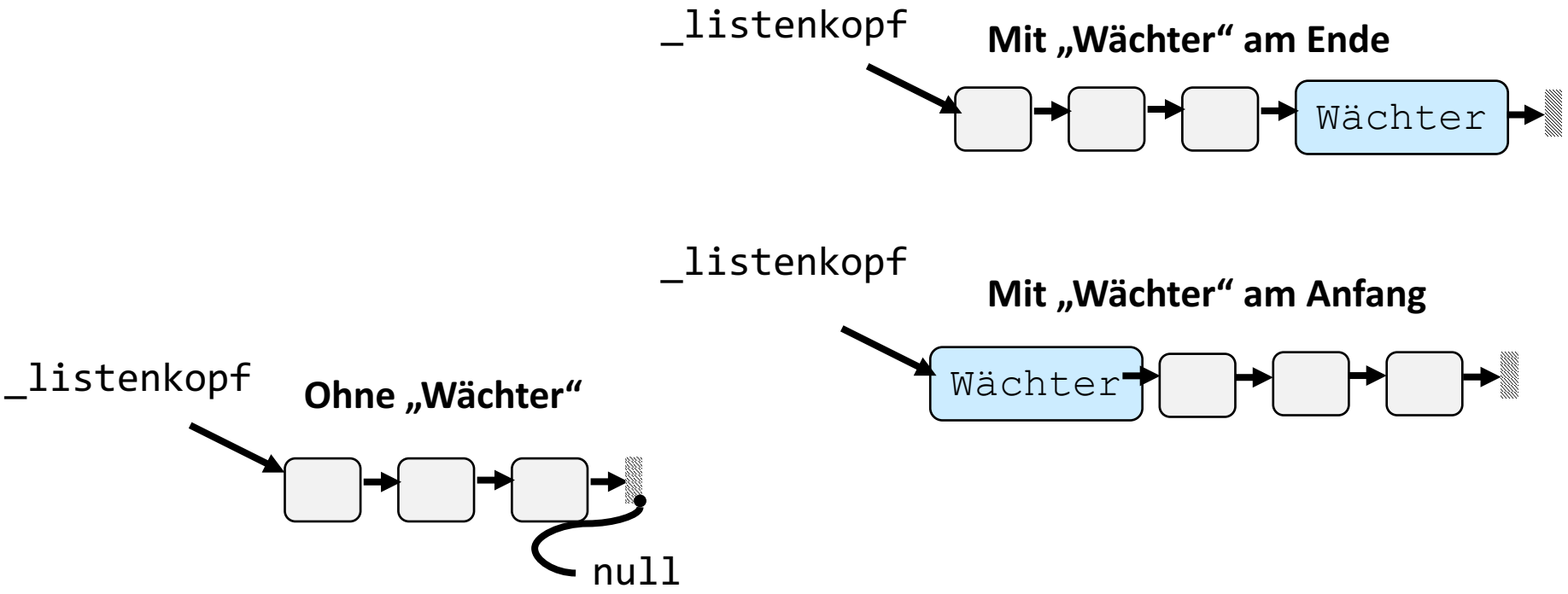
- Ein Kettenglied hat zusätzlich eine Referenz auf das vorige Kettenglied
- Ermöglicht ein effizientes Durchlaufen der Liste in beide Richtungen
- Einfügen und Entfernen werden vereinfacht
- Die JCF-Implementation LinkedList basiert auf diesem Konzept



Designalternative für Listenenden

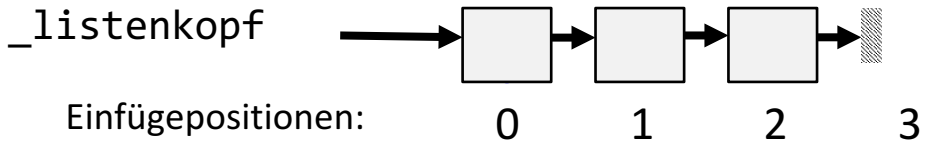
- Am Listenenden (Anfang und/oder Ende) können explizit leere Kettenelemente stehen, sog. Wächter (engl. sentinel)

- Vorteil eines Wächter-Objekts:
  - Weniger Sonderfälle zu programmieren (Beim Einfügen, Entfernen etc.)



Sonderfälle: Beispiel Listenanfang

- Angenommen, es soll ein Element an Position x eingefügt werden.



- **Ohne Wächter** kann nicht in jedem Fall auf das Element vor der Einfügeposition positioniert werden (explizite Behandlung mit einer if-Abfrage)

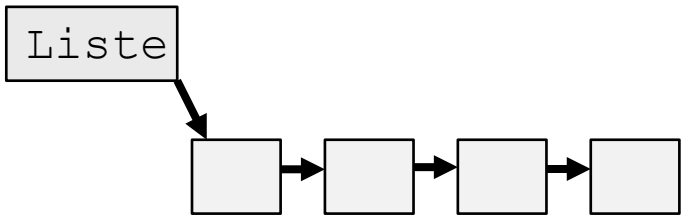
- **Mit Wächter** entfällt diese Behandlung, da ein Wächterelement am Anfang steht

Designalternative Verweise

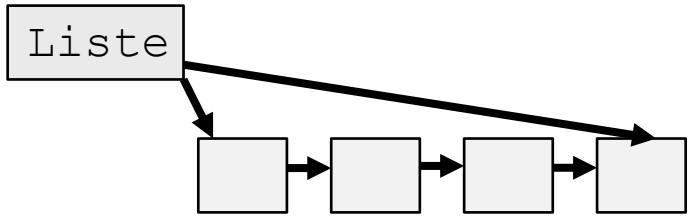
- Neben einem Verweis auf den Anfang kann auch ein Verweis auf das Ende einer Liste gehalten werden

- Vorteil:
  - Der Zugriff auf das Listenende ist genau so schnell wie auf den Listenanfang (gut für Operationen wie Anfügen)

Verweis auf Listenanfang



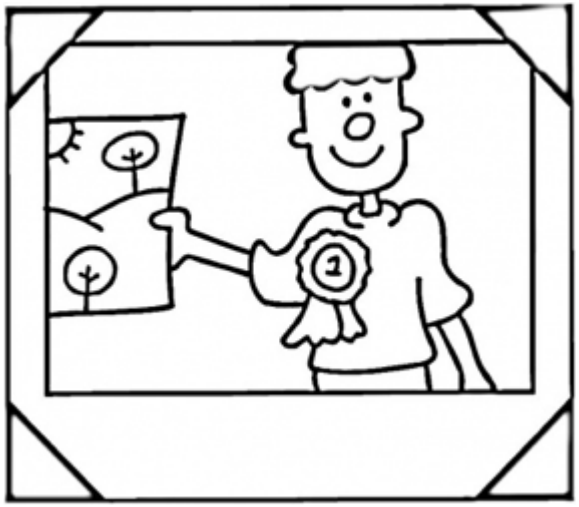
Verweis auf Listenanfang und -ende



Zitat: Good Programmers / Bad Programmers

- *"Good programmers plan before they write code, especially when there are pointers involved. For example, if you ask them to reverse a linked list, good candidates will always make a little drawing on the side and draw all the pointers and where they go. They have to.* ***It is humanly impossible to write code to reverse a linked list without drawing little boxes with arrows between them.*** *Bad programmers will start writing code right away."*

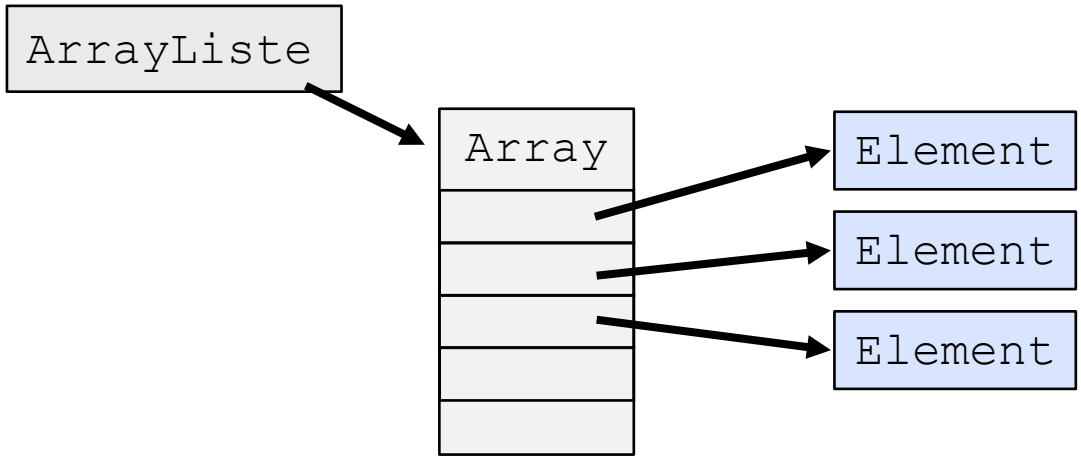
  (Blog: Joel on Software, March 2000)



Array-Implementationen für Listen

- Grundidee: Eine Klasse ArrayListe, repräsentiert dem Klienten gegenüber die gesamte Liste und bietet alle Operationen einer Liste an ihrer Schnittstelle

- Intern wird ein Array verwendet, in dem alle bisher eingefügten Elemente gehalten werden (Innensicht, Implementation)



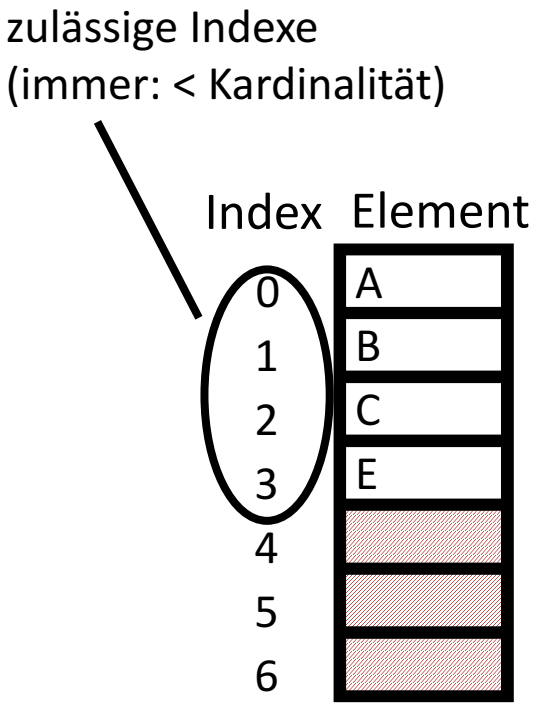
„Wachsende“ Arrays für Listen

- Arrays sind als direkte Implementation wegen ihrer festen Größe ungeeignet

- Konzept von wachsenden Arrays:
  - Erzeugung eines Arrays mit einer Anfangsgröße
  - Füllung mit den einzufügenden Elementen
  - Wenn das Array voll ist; Erzeugung eines größeren und einer Kopie aller enthaltenden Elemente in das neue Array

- Unterscheidung zwischen
  - Logischer Größe der Liste (Anzahl der Elemente, **Kardinalität**)
  - Physikalischer Größe (**Kapazität**) des implementierenden Arrays

- Es muss immer gelten: Kapazität ≥ Kardinalität

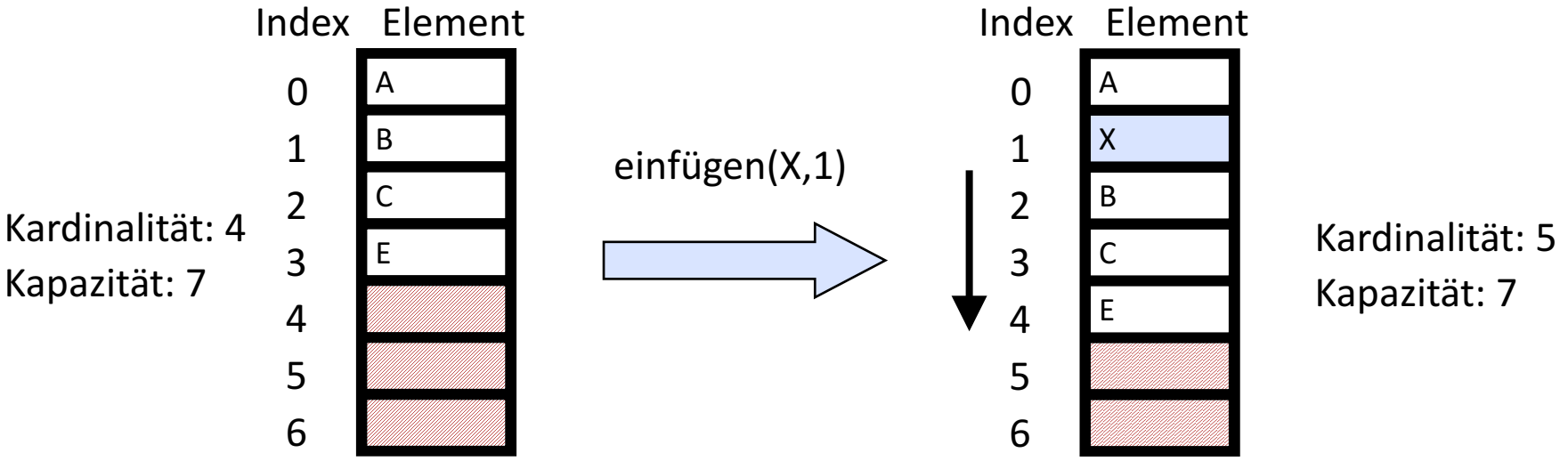


Operationen auf Array-Listenimplemenation

- Der Zugriff auf eine beliebige Indexposition ist schnell: Indexbasierter Arrayzugriff

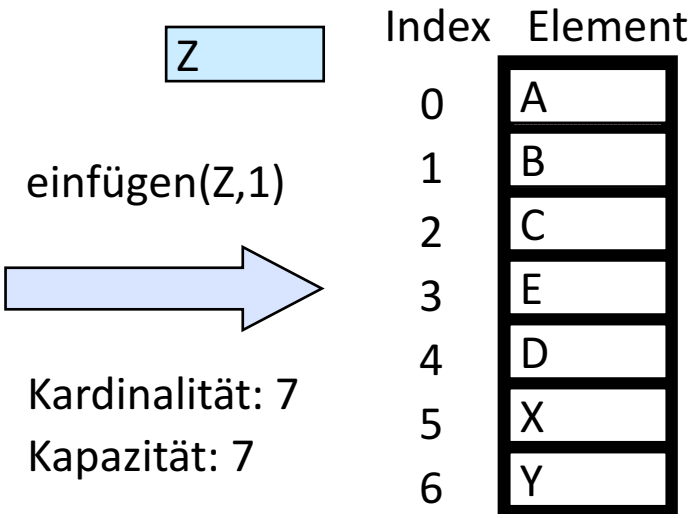
- Bei jedem Einfügen oder Löschen müssen die nachfolgenden Elemente innerhalb des Arrays verschoben werden

- Extremfall (einfügen am Listenanfang): Alle Elemente müssen um eine Position verschoben werden



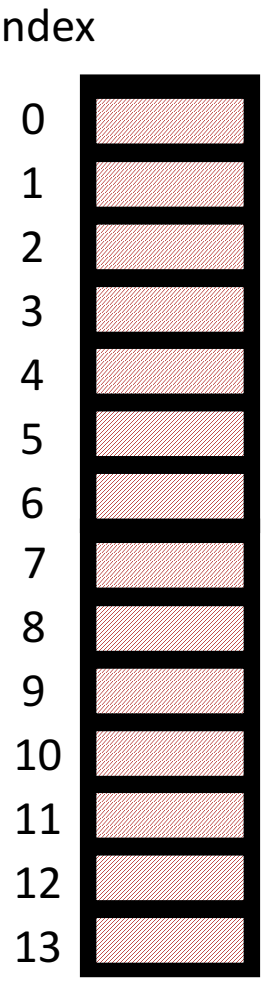
Wachsende Arrays

- Wenn die Kapazität des Arrays für ein neu einzufügendes Element nicht ausreicht, muss zuerst ein größeres Array (beispielsweise mit doppelter Kapazität) angelegt werden



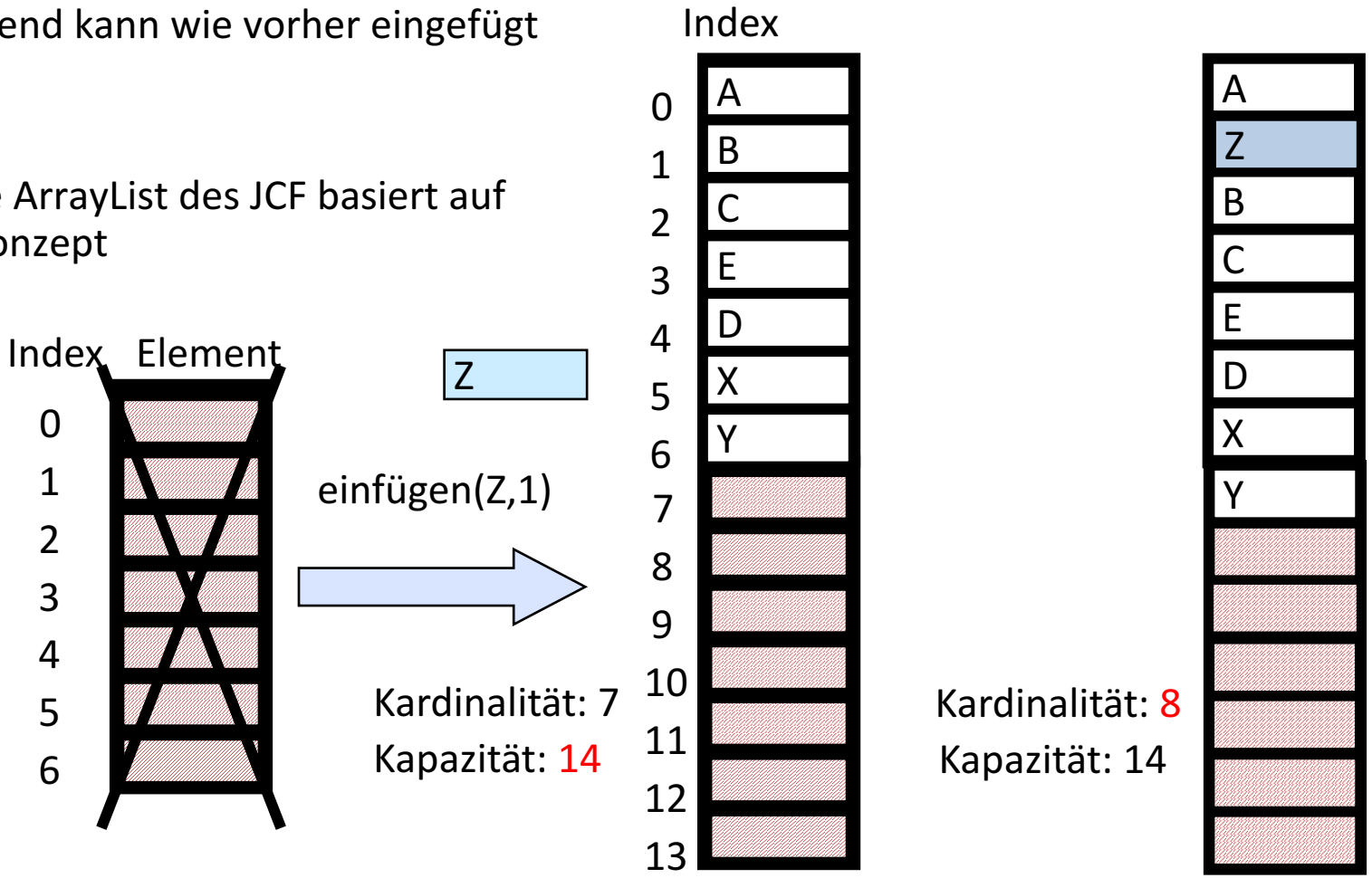
Altes Array voll!

Größeres Array erzeugen…



… und umkopieren

- Alte Elemente werden kopiert
- Anschließend kann wie vorher eingefügt werden
- Die Klasse ArrayList des JCF basiert auf diesem Konzept



Vergleich der Listen-Implementationen

- **Einfügen in eine Liste**
  - **LinkedList**:
    - – Zielposition ist erst durch Traversieren der Liste zu erreichen
    - – Objekterzeugung für jede Einfügung
    - \+ Das Einfügen ist sehr einfach
  - **ArrayList**:
  - – Alle Elemente nach der Einfügeposition müssen um eine Position verschoben werden
  - – Wenn die Kapazität ausgeschöpft ist, muss ein neues Array angelegt und alle Elemente müssen umkopiert werden
  - \+ Die Position zum Einfügen kann direkt angesprochen werden

- **Zugriff**
  - \+ Bei **ArrayList** erfolgt der Zugriff in konstanter Zeit
  - – Bei der **LinkedList** wird durchschnittlich die halbe Liste durchlaufen

Welche Listen-Implementation?

- Je nach Anwendungsfall eignen sich unterschiedliche Implementationen:
  - Bei Listen mit relativ **konstanter Größe** mit **häufigen Zugriffen** auf beliebige Positionen, ist die **ArrayList** geeignet
  - Bei Listen mit **dynamischer Größe**, bei denen viel eingefügt und entfernt wird (insbesondere am Listenanfang), ist die **LinkedList** die bessere Wahl

- Pragmatik für Java: Bei den meisten Anwendungen mit eher kleinen Listen ist die ArrayList ausreichend

List-Implementationen im Vergleich

LinkedList

- Knoten, die mit einander verbunden werden und eine **Kette** bilden
- Indizierung durch „Abzählen“
- Einfügen legt ein neues Objekt an, das eingekettet wird
- Löschen kettet ein Kettenglied aus der Kette aus

ArrayList

- Dynamisch „wachsendes“ **Array**
- Direkt indizierbar
- Einfügen erfordert Verschieben von Folgeelementen und eventuelle Neuerzeugung eines kompletten Arrays plus Umkopieren
- Löschen erfordert Verschieben von Folgeelementen

Aufwand für Operationen

- Aufwand ist die Menge an elementaren Schritten, die für eine zusammengesetzte Operation ausgeführt werden müssen
  - Bsp.: Operation Einfügen eines Elementes an Position i auf einer verketteten Liste erfordert mehrere elementare Schritte

- Elementare Schritte sind:
  - Zuweisungen (x = y, i++, …)
  - Vergleiche (a <= b, next != null, …)
  - Aufrufe mit konstantem Zeitbedarf (Objekterzeugung kleiner Objekte, sondierende Methoden, …)

Konstanter und variabler Anteil des Aufwandes

- Der Aufwand einer Operation setzt sich aus einem konstanten Anteil und einem variablen Anteil zusammen:
  - Der konstante Anteil ist für jede Ausführung der Operation gleich
  - Der variable Anteil ist abhängig von der Menge der zu verarbeitenden Daten

- Beispiel: Einfügen in verkettete Liste
  - Aufwand für Erzeugen und Verketten immer gleich (Konstant)
  - Der Durchlauf ist abhängig vom angefragten Index (Variabel)

Abschätzungen des Aufwandes

- Bei Abschätzungen wird häufig vom schlechtesten Fall (engl.: **worst case**) ausgegangen
- Ebenfalls verbreitet ist die Abschätzung des Aufwandes **im Mittel**
- Üblicherweise wird von **großen Datenmengen** ausgegangen, sodass der konstante Anteil vernachlässigt wird
- Der Aufwand ist eine Funktion, die von der Anzahl N der zu verarbeitenden Datenelemente abhängt:
  - Aufwand = f(N)
- Im Zusammenhang von Aufwandsbetrachtungen für Algorithmen wird auch von ihrer **Komplexität** gesprochen.

„O-Notation“

- Auch **Landau-Notation**, nach dem deutschen Zahlentheoretiker Edmund Landau

- In der Informatik für Komplexitätstheorie verwendet um Probleme in **Komplexitätsklassen** einzuteilen

- Typische Komplexitätsklassen sind:
  - $O(1)$ konstanter Aufwand (u.a. alle elementaren Schritte)
  - $O(\log n)$ logarithmischer Aufwand (u.a. Baumsuche)
  - $O(n)$ linearer Aufwand (u.a. Suche in Listen)
  - $O(n \cdot \log n)$ (u.a. gute Sortierverfahren)
  - $O(n^2)$ quadratischer Aufwand (u.a. einfache Sortierverfahren)
  - $O(2^n)$ exponentieller Aufwand (u.a. Erzeugen der Potenzmenge)

- Die Klassen geben eine Größenordnung für den Aufwand in Abhängigkeit von der zu verarbeitenden Datenmenge n

Mehr in FGI1

Komplexitäten der Listenoperationen

- **Einfügen in eine Liste**:
  - LinkedList: O(n)
    (linearer Aufwand, da bis zu n Elemente durchlaufen werden müssen)
  - ArrayList: O(n)
    (linear Aufwand, da bis zu n Elemente verschoben werden müssen)

- **Zugriff auf ein Element** über einen Index:
  - LinkedList: O(n) (linearer Aufwand, siehe oben)
  - ArrayList: O(1)
    (konstanter Aufwand, da direkte Abbildung auf indizierten Zugriff der unterliegenden Rechnerarchitektur)

Zwischenfazit

1. Für Listen gibt es zwei klassische imperative Implementationen: **verkettete Listen** und **wachsende Arrays**.

2. Verkettete Listen können **einfach** oder **doppelt verkettet** sein.

3. Wachsende Arrays basieren auf Arrays, die bei Bedarf mit größerer Kapazität angelegt werden.

4. Beide Implementationen haben Stärken und Schwächen. Mit Hilfe der **O-Notation** können wir diese Unterschiede formal greifbar machen.

Überblick

1. Listenimplementierung
   - Einfach verkettet
   - Doppelt verkettet
2. **Mengenimplementierung**
   - **Bäume**
   - **Hashing**

Einfügen in Mengen: hoher Aufwand?

- Insbesondere bei Sets (Mengen) ist der Test auf Enthaltensein wichtig
- Beim Einfügen muss auf Duplikate geprüft werden
- Bei einer Implementation der Menge als Liste, der Aufwand für das Einfügen linear von der Größe der Menge abhängen: **O(n)**

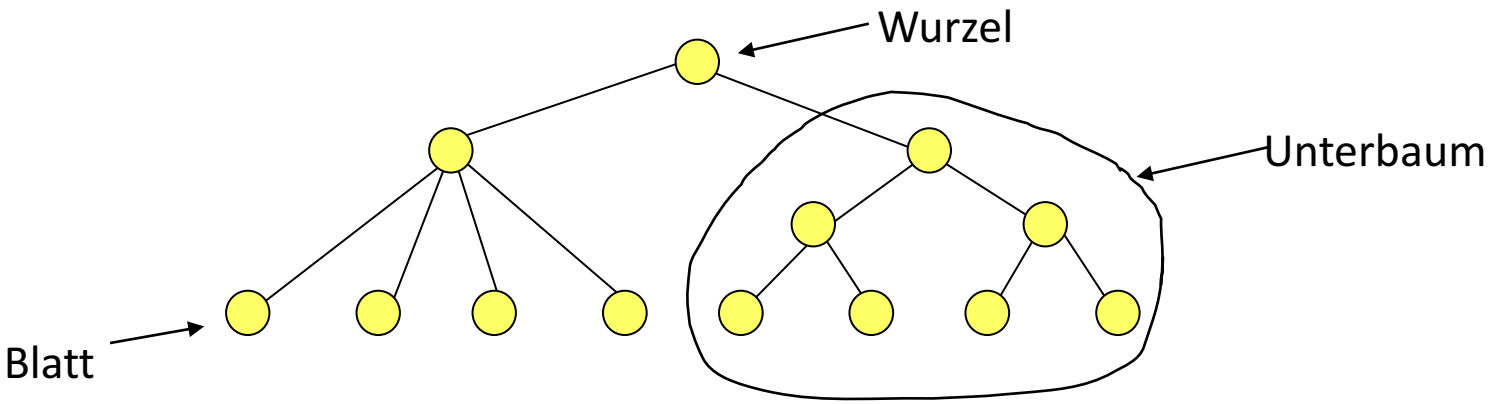
- Möglich sind Realisierungen mit O(log n) und sogar mit O(1), also konstantem Aufwand

Anforderungen an effiziente Suchverfahren

- Der Test auf Enthaltensein wird als Suche bezeichnet
- Es soll nicht jedes Element in der Menge mit dem zu suchenden Element vergleichen werden müssen (linearer Aufwand, **O(n)**)
- Für eine geeignete Struktur müssen die Elemente Anforderungen erfüllen. 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

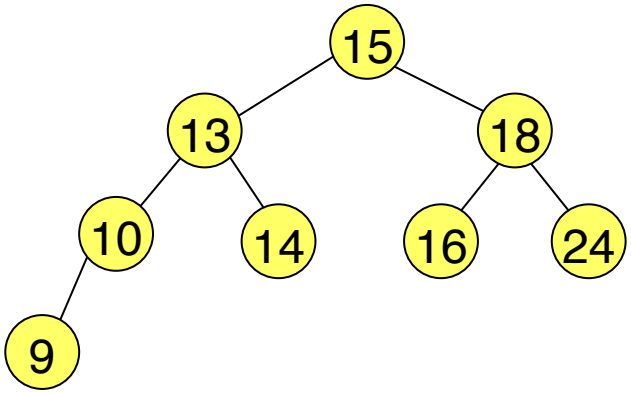
Bäume

- Ein **Baum** (engl.: tree) ist eine Struktur, in der Knoten miteinander durch (gerichtete) Kanten verbunden sind
- Knoten haben über Kanten beliebig viele Kindknoten (Nachfolger)
- 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

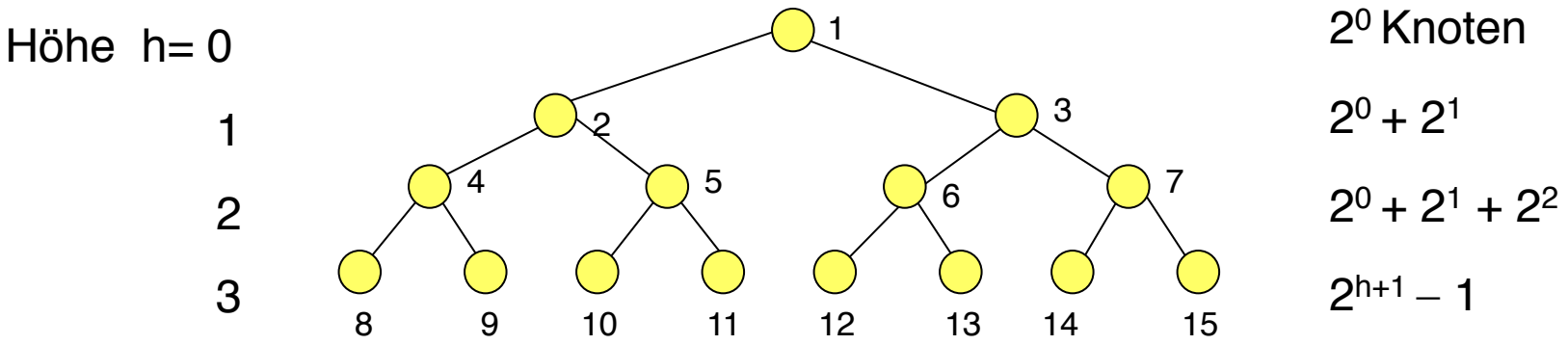
- In einem **binären Baum** hat ein Knoten maximal zwei Kindknoten
- Jeder Knoten enthält ein sortierbares Element
- Im linken Unterbaum sind alle „kleineren“ Elemente
- Im rechten Unterbaum sind alle „größeren“ Elemente



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

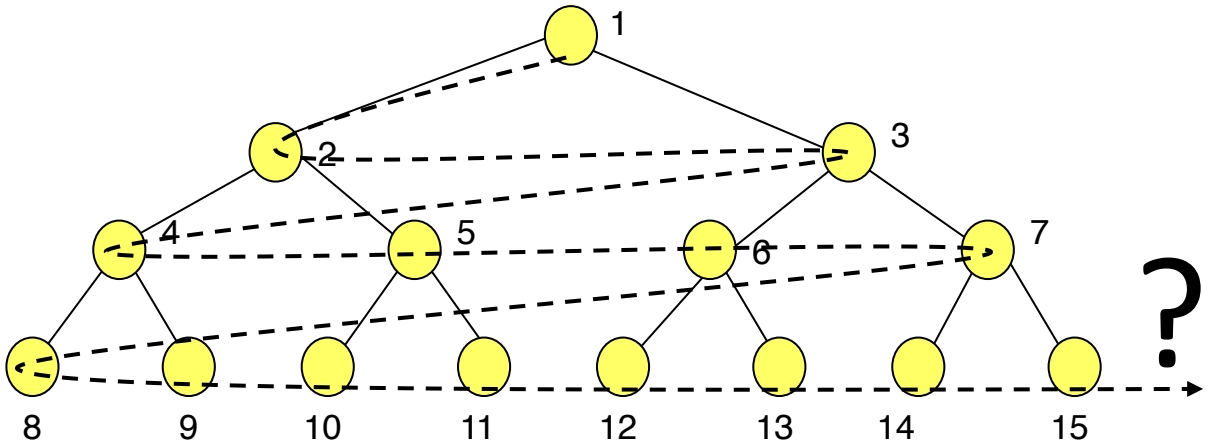
Merkmale von binären Bäumen

- 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 $2^{h+1} - 1$



Traversieren von Bäumen

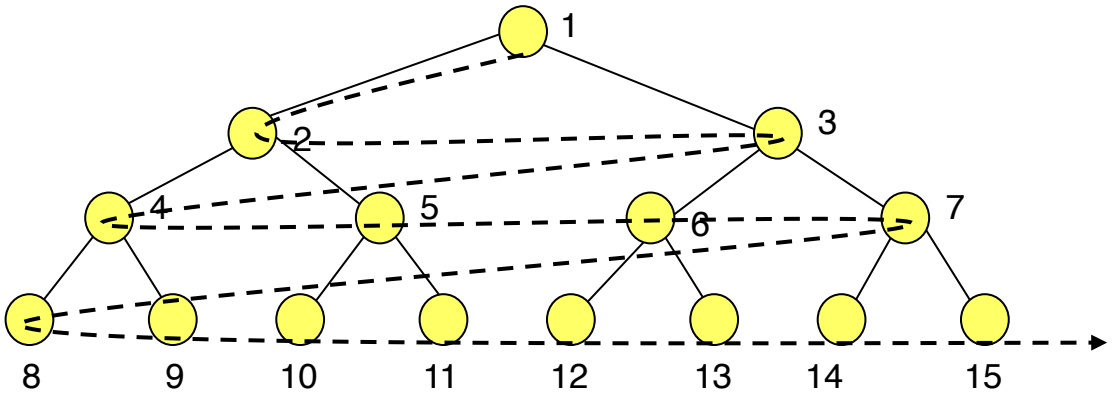
- Oft müssen alle Knoten eines Baumes der Reihe nach bearbeitet werden
- Dazu muss ein Ordnungsprinzip gewählt werden
- Die bekanntesten Traversierungsstrategien sind:
  - Breitendurchlauf
  - Tiefendurchlauf



nach © Neumann

Breitendurchlauf

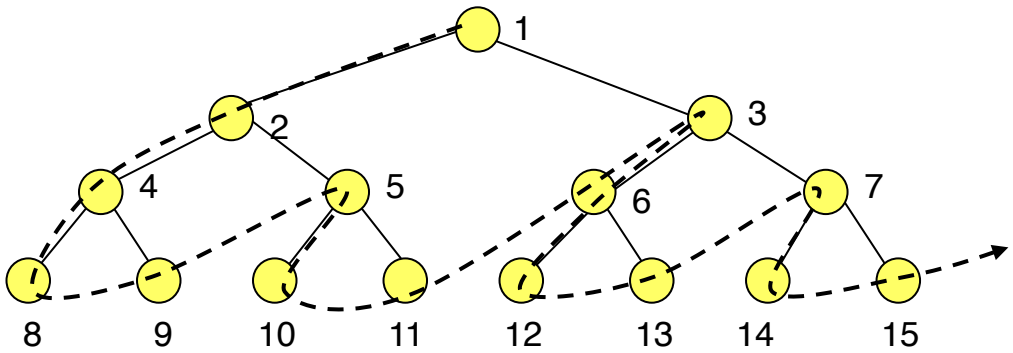
- Breitendurchlauf (level-order tree traversal)
- Die Idee:
  Verfahren, um einen Baum "schichtenweise" abzuarbeiten,
  z.B. bei der Suche nach Zielknoten mit minimalem Abstand zur Wurzel
- Auch auf allgemeinen Bäumen anwendbar
- Strategie:
  - Breite vor Tiefe,
  - links vor rechts



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

Tiefendurchlauf

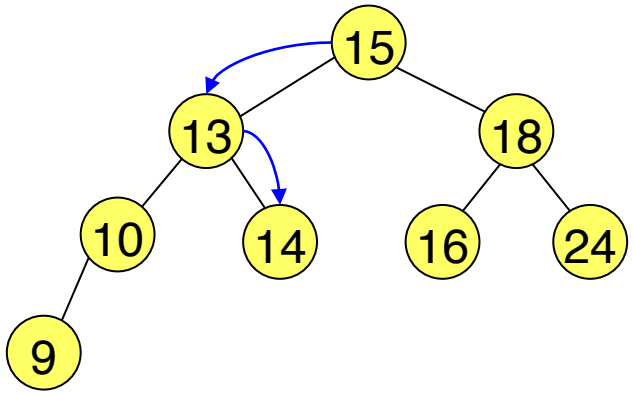
- Die Idee:
  Allgemeines Verfahren, um einen Baum "astweise" in Richtung seiner Blätter abzuarbeiten,
  z.B. bei der Suche nach Planschritten, die zu einem Ziel führen sollen
- Auch auf allgemeinen Bäumen anwendbar
- Strategie:
  - Tiefe vor Breite,
  - links vor rechts



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

Suchalgorithmus für binäre Suchbäume

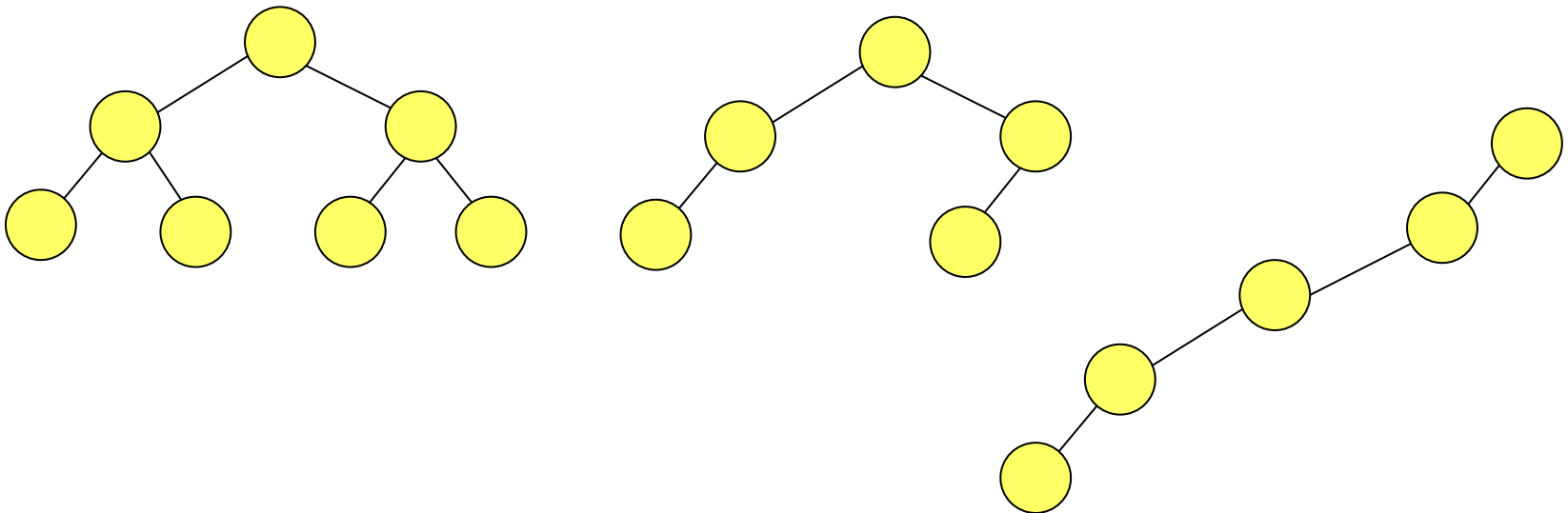
- Die Suche nach einem Element ist (rekursiv) folgendermaßen möglich:
  - Ist der Baum leer?
    - Ja → Suche erfolglos
    - Nein: Enthält der Wurzelknoten das gesuchte Element?
      - Ja → Suche erfolgreich
      - Nein: Ist das gesuchte Element kleiner als des aktuelle Element?
        - Ja: Weitersuchen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



Suche nach k = 14

Bäume können entarten

- Bei einem ausbalancierten Baum ist die Suche recht schnell
- Binäre Bäume können „entarten“: Eine verkettete Liste kann gesehen werden als ein verkümmerter binärer Baum mit jeweils einem Kindknoten
- Es fehlen geeignete Bedingungen...



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 Logarithmus Dualis, also der Logarithmus zur Basis 2)

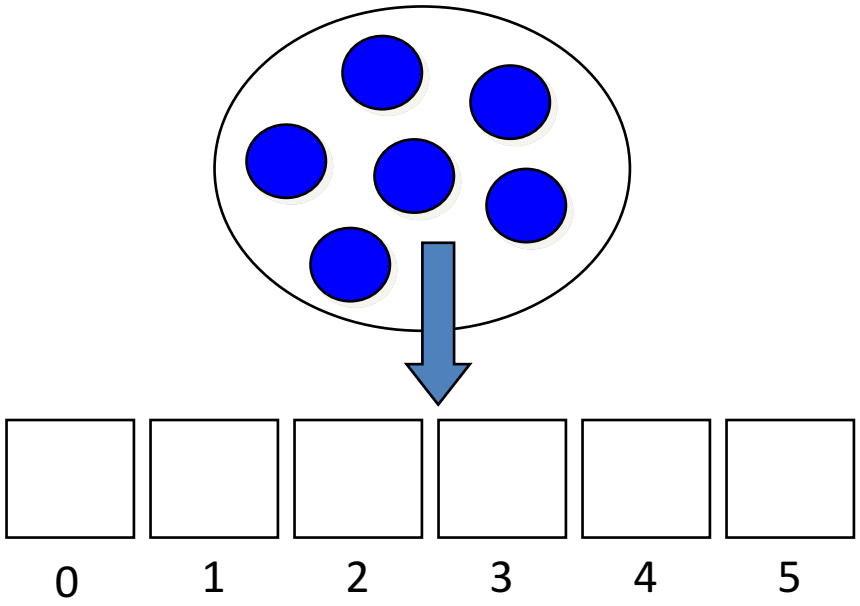
- Die Suche muss von der Wurzel bis zu jedem Blatt höchstens ld(n) Vergleiche vornehmen; der Aufwand ist damit in seiner Größenordnung **O(log(n))**

- Die Baum-Implementationen des JCF (TreeSet und TreeMap) benutzen binäre, balancierte Bäume (die Elemente müssen sortierbar sein)

Grundidee von Hash-Verfahren

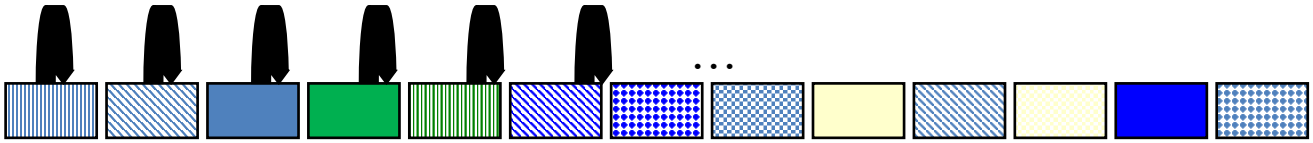
- Grundidee von Hash-Verfahren (auch: Hashing) ist, Elemente auf eine Indexstruktur abzubilden

- Aus Elementen ist unmittelbar ein Index berechenbar

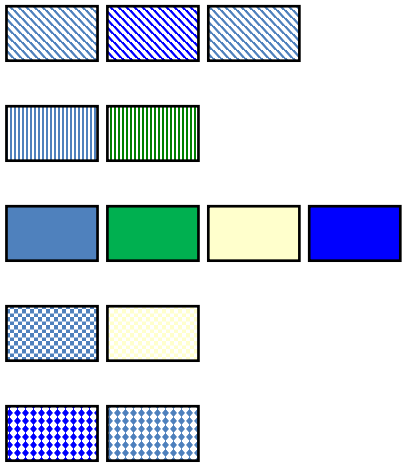


Lösungsansatz für Hashing

- Statt eine Liste komplett zu durchsuchen:



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

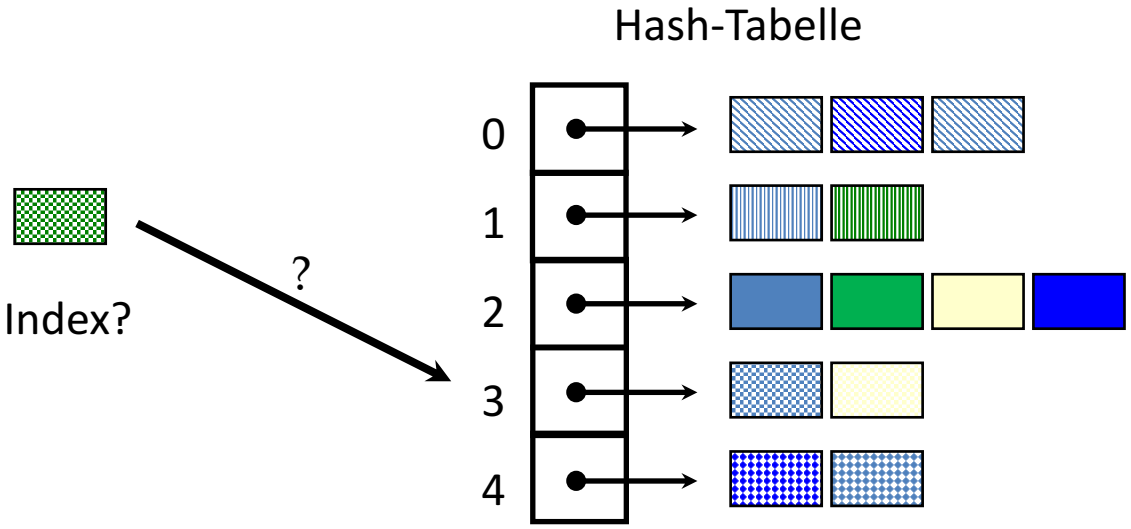


Jede Liste enthält die Elemente einer Kategorie

Jedes Element kann Auskunft geben, zu welcher Kategorie es gehört

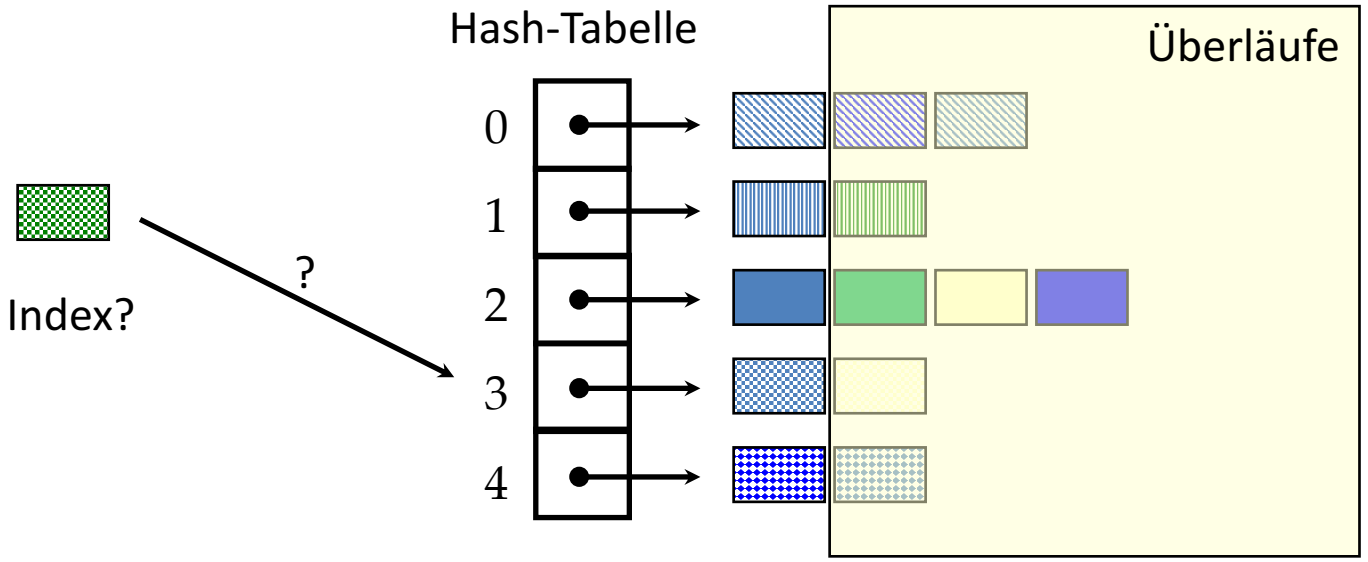
Hash-Tabelle

- Im Kern ist eine Hash-Tabelle von (möglichst kurzen) Listen
- Für das zu suchende Element wird zuerst der Index der Liste in der Tabelle ermittelt, in der Elemente der gleichen Kategorie liegen
- Nach dem (schnellen) indexbasierten Zugriff auf die Liste wird diese dann (linear) durchsucht



Ziel: möglichst wenige Überläufe

- Im Idealfall enthalten die Listen maximal ein Element enthalten
- Nach der Indexberechnung ist dann für eine Suche maximal ein Vergleich notwendig
- Sind mehr als ein Element enthalten, werden die überschüssigen Elemente als Überläufe bezeichnet
  - Die Listen werden auch Auch Überlaufbehälter (engl.: bucket) bezeichnet

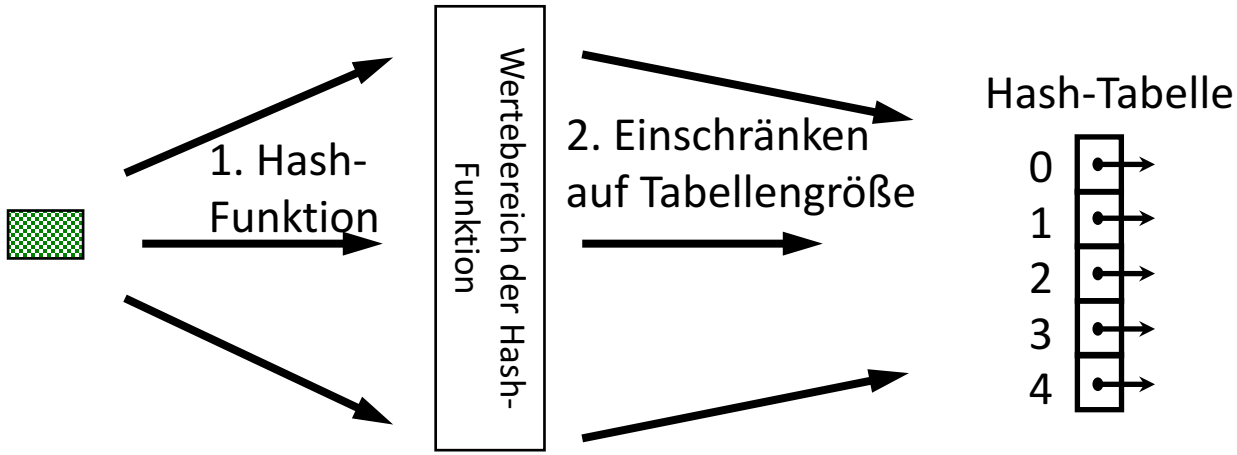


Die Hash-Funktion

- Sie bildet ein Element auf einen ganzzahligen Wert ab (int)

- Der berechnete Wert bildet eine Kategorie: Alle Elemente mit demselben Wert fallen in dieselbe Kategorie
  - Wenn 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:

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

nach © Neumann

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.

Hash-Verfahren im Java Collections Framework

- Die Implementation HashSet für das Interface Set im JCF basiert auf einem Hash-Verfahren
- Als 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!

Hash-Verfahren im Java Collections Framework (II)

- 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
- Geschwindigkeit hängt auch von der Güte der Hash-Werte ab.

Zusammenfassung

1. Bei **Mengen** ist der Test auf Enthaltensein (**Suche**) typischerweise die wichtigste Operation.

2. Mit einer naiven Implementation ist der Such-Aufwand ***O(n)***.

3. Bei einem **balancierten binären Suchbaum** reduziert sich der Such-Aufwand auf ***O(log(n)).***

4. Die **Set**-Implementation **TreeSet** des JCF erfordert, dass die Elemente das Interface **Comparable** implementieren.