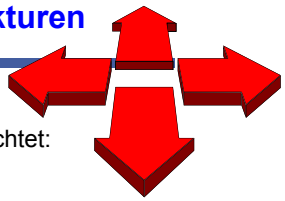
Softwareentwicklung I (SE1): Grundlagen objektorientierter Programmierung

- Vorlesung 12 -

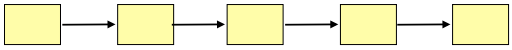
Axel Schmolitzky
Heinz Züllighoven
Guido Gryczan
et al.

1

Von Sammlungen zu dynamischen Datenstrukturen



- Wir haben bisher **Sammlungen** wie Mengen und Listen betrachtet:
  - Elemente können hinzugefügt und entnommen werden.
  - Es gibt unterschiedliche Organisationsprinzipien.
- Wir betrachten nun die **Implementationen** dieser Sammlungen und thematisieren damit erstmals **dynamische Datenstrukturen**, die in der Informatik eine große Tradition haben.
- Wir klären im Weiteren die Begriffe
  - „dynamisch“
  - „Datenstruktur“
- Wir geben einen Einstieg in die wichtigsten dynamischen Datenstrukturen mit ihren spezifischen Stärken und Schwächen.



2

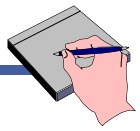
Implementationen für Sammlungen

- Das Java Collections Framework (JCF) stellt für seine Collection-Interfaces (wie `List`, `Set`, etc.) einige Implementationen zur Verfügung.
- Alle Implementationen basieren auf zwei grundlegenden Programmierkonstrukten:
  - **Arrays**
  - **verkettete Strukturen**
- Einige Implementationen machen nur von dem einen oder dem anderen Konzept Gebrauch, andere kombinieren sie. Insgesamt können im JCF vier Implementierungskonzepte unterschieden werden:
  - **verkettete Listen**
  - **wachsende Arrays**
  - **balancierte Bäume**
  - **Hash-Verfahren**

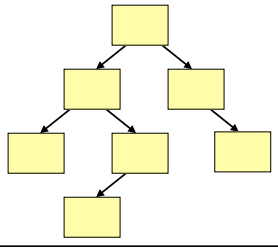
Dies sind Beispiele für dynamische Datenstrukturen!

3

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.
- **Ändern einer Struktur** bedeutet
  - Hinzufügen, Modifizieren und Löschen von Elementen
  - Ändern von Beziehungen
- Als **dynamisch** werden Datenstrukturen dann bezeichnet, wenn sie durch das Einfügen und Entfernen ihrer Elemente wachsen und schrumpfen.

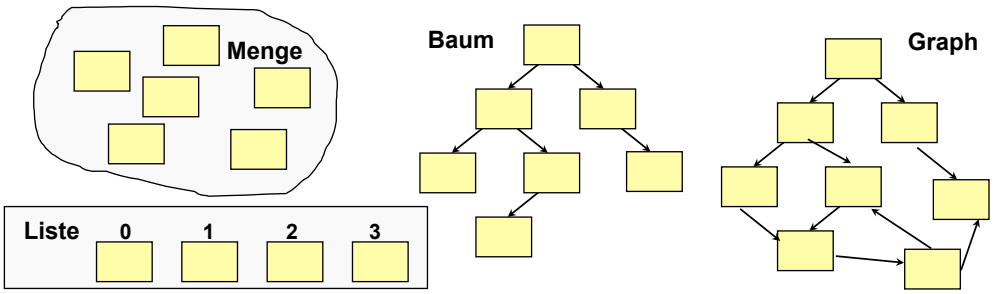


nach © Neumann

4

Einteilung dynamischer Datenstrukturen

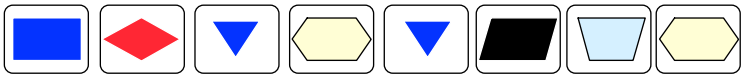
- Üblicherweise werden dynamische Datenstrukturen nach den Eigenschaften ihrer grundlegenden Struktur eingeteilt.
- Wir unterscheiden:
  - Strukturen von Elementen **ohne Relation** zueinander (z.B. für Mengen)
  - Lineare oder **sequenzielle Strukturen** (z.B. für Listen)
  - **Bäume**, in denen ein Element ein Vorgängerelement aber mehr als ein Nachfolgerelement haben kann.
  - **Graphen**, in denen ein Element beliebig mit anderen Elementen verbunden sein kann.



5

Sammlungen implementieren I: Listen

- Listen sind die grundlegendste (elementare) sequenzielle Struktur.
- Sobald wir dynamische Datenstrukturen zu ihrer Implementierung beherrschen, können wir auch weitere lineare Sammlungsarten wie *Stacks* und *Queues* implementieren.
- Wir werden zwei Implementationsformen betrachten:
  - eine basierend auf Verkettung
  - eine basierend auf Arrays



6

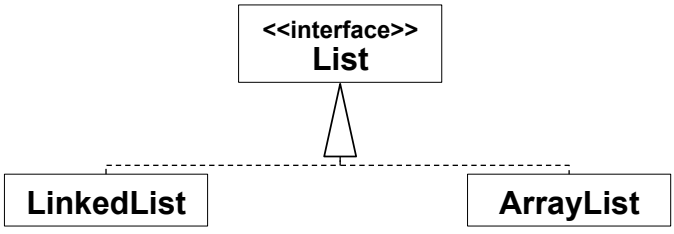
Zur Erinnerung: der Umgang mit einer Liste

- Aus Sicht des Klienten einer Liste sind relevant:
  - Eine Liste kann **beliebig viele Elemente** enthalten.
  - Über den Index kann direkt auf **beliebige Positionen** in der Liste zugegriffen werden.
  - Das **Einfügen** eines Elements **erhöht** den Index der nachfolgenden Elemente.
  - Das **Entfernen** eines Elements **verringert** den Index der nachfolgenden Elemente.
  - Häufig wird die Information benötigt, ob ein gegebenes Element bereits in der Liste enthalten ist („**Test auf Enthaltensein**“).

7

Listen-Implementationen im Java Collections Framework

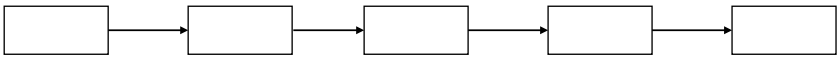
- Der Umgang mit einer Liste ist im JCF mit dem Interface `List` modelliert.
- Das JCF bietet zwei Implementationen für dieses Interface:
  - `LinkedList`
  - `ArrayList`
- `LinkedList` basiert auf dem Konzept **verkettete Liste**.
- `ArrayList` basiert auf dem Konzept **wachsender Arrays**.



8

Lineare Datenstrukturen für Listen: Verkettung

- Eine **verkettete Liste** ist das bekannteste Beispiel für eine sequenzielle dynamische Datenstruktur:
- 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 ist grundsätzlich als Struktur betrachtet eine **Sequenz** ihrer Elemente:
  - Jedes Listenelement ist **mit dem nächsten verbunden**.
  - Um vom Anfang zum Ende der Liste zu gelangen, muss **jedes Element traversiert** werden.



9

Einfach verkettete Liste

- Eine Liste als Referenzkette zwischen ihren Elementen kann auf zwei Arten realisiert werden. Wir unterscheiden:
  - **Einfach verkettete Liste**:
    - jedes Listenelement hat nur eine Referenz auf sein **Nachfolgerelement**.
    - Die Liste kann nur elementweise vom Anfang zum Ende traversiert werden.

10

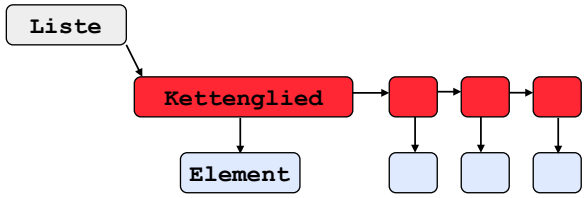
Doppelt verkettete Liste

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

11

Schema des objektorientierten Entwurfs einer Liste

- Üblicherweise besteht der objektorientierte Entwurf einer verketteten Liste aus verschiedenen Objekten:
  - Ein Objekt, das die **Liste insgesamt** für ihre Klienten repräsentiert.
  - Objekte als **Kettenglieder**, die die Verkettung der Liste realisieren. Sie sind für die Klienten nicht sichtbar.
  - Objekte, die als **Elemente** in der Liste gespeichert sind. Sie werden von den Klienten über die Umgangsformen der Liste verwaltet.

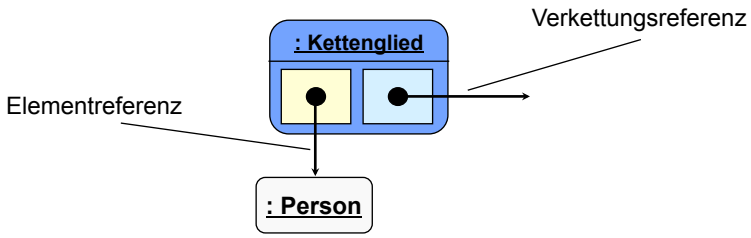


12

Konstruktion eines Kettenglieds

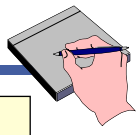
Ein Kettenglied kann objektorientiert so entworfen werden:

- Jedes **Kettenglied** wird als ein eigenes Objekt modelliert. Dazu wird eine eigene Klasse für die Kettenglieder definiert, etwa `Kettenglied`.
- Jedes Kettenglied hält eine Referenz auf das eigentliche **Element der Sammlung** (beispielsweise in einer Liste von Personen eine Referenz auf ein Exemplar der Klasse `Person`).
- Ein Kettenglied hält außerdem mindestens eine Referenz auf ein weiteres Kettenglied (das Nachfolgerelement) als **Verkettungsreferenz**.



13

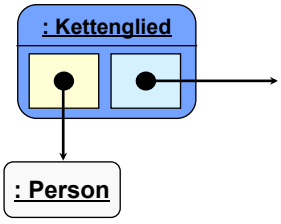
Definition einer Klasse für Kettenglieder



Hier wird eine Exemplarvariable deklariert, die den gleichen Typ hat wie die definierende Klasse!

Dies wird auch **strukturelle Rekursion** genannt.

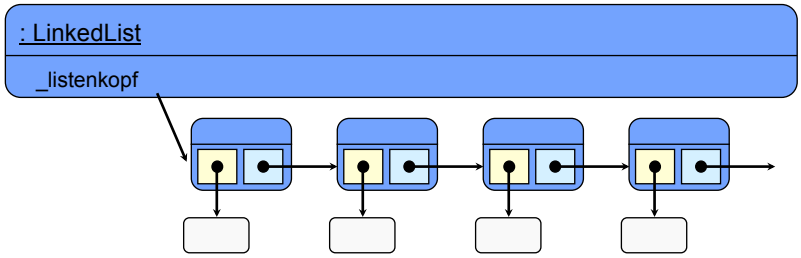
```
class Kettenglied {
    private Kettenglied _nachfolger;
    private Person _element;
    ...
}
```



14

Konstruktion einer einfach verketteten Liste

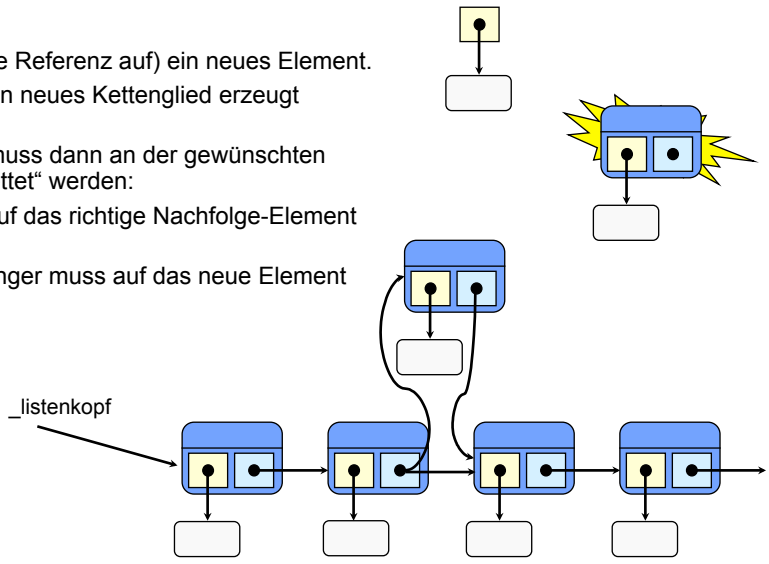
- Die Liste wird für ihre Klienten als Exemplar einer eigenen Klasse realisiert, etwa `LinkedList`.
- Die Kettenglieder (Exemplare von `Kettenglied`) bilden die innere Struktur der Liste.
- Die referenzierten Elemente werden üblicherweise **nicht** als Teil der Liste angesehen (beispielsweise wird beim Entfernen aus einer Liste nicht automatisch das Element selbst gelöscht).
- In der Klasse `LinkedList` wird üblicherweise die Referenz auf das erste `Kettenglied` gehalten („**Listenkopf**“).



15

Einfügen in einer verketteten Liste

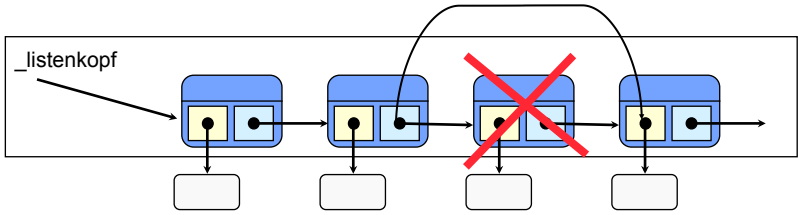
- Gegeben: (eine Referenz auf) ein neues Element.
- Zuerst muss ein neues Kettenglied erzeugt werden.
- Dieses Glied muss dann an der gewünschten Stelle „eingekettet“ werden:
  - Es muss auf das richtige Nachfolge-Element verweisen.
  - Der Vorgänger muss auf das neue Element verweisen.



16

Entfernen aus einer verketteten Liste

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

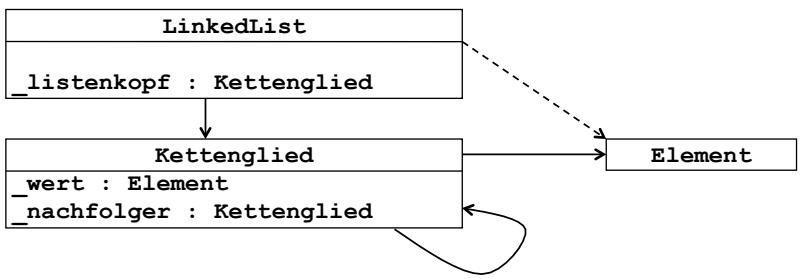


17

Klassendiagramm einer einfach verketteten Liste

Das Klassendiagramm zeigt wesentliche objektorientierte Konstruktionsmerkmale einer einfach verketteten Liste:

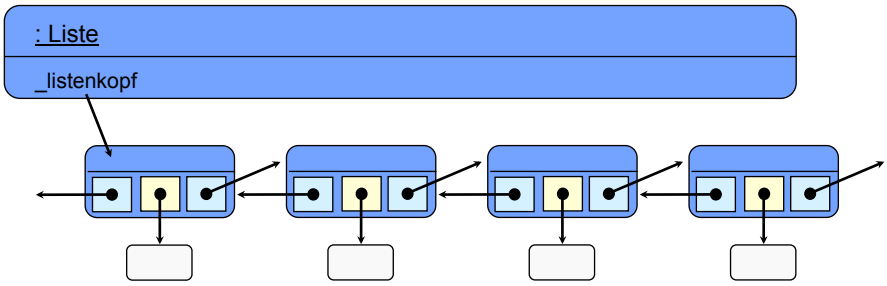
- Die Klasse `LinkedList` hält einen Verweis auf die Klasse `Kettenglied` im Attribut `_listenkopf`.
- Sie benutzt die Klasse `Element` in den Parametern ihrer Methoden.
- Die Klasse `Kettenglied` speichert jeweils ein Exemplar der Klasse `Element` im Attribut `_wert`.
- `Kettenglied` verweist auf sich selbst, um im Attribut `_nachfolger` das nächste Kettenglied referenzieren zu können.



18

Doppelt verkettete Liste

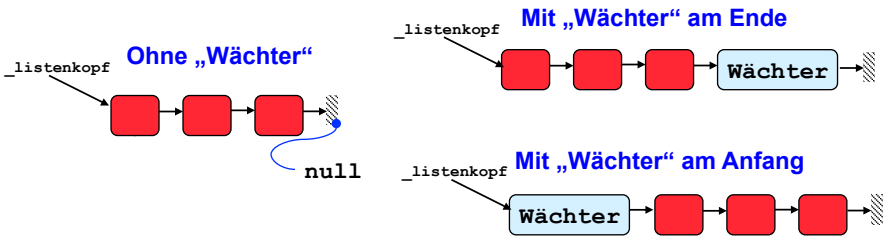
- Bei einer doppelt verketteten Liste hat ein Kettenglied außerdem eine Referenz auf ein weiteres Kettenglied (das "vorige"). Dies ermöglicht ein effizientes Durchlaufen der Liste in beide Richtungen.
- Einfügen und Entfernen werden vereinfacht.
- Die JCF-Implementation `LinkedList` basiert auf diesem Konzept.



19

Designalternative Listenenden

- An den **Listenenden** (Anfang und/oder Ende) können explizit leere Kettenelemente stehen, sog. **Wächter** (engl. sentinel).
- Vorteil eines Wächter-Objekts:
  - Es gibt weniger Sonderfälle zu programmieren (Beim Einfügen, Entfernen etc.).

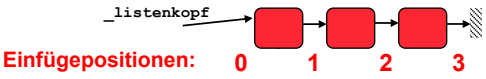


© Budd

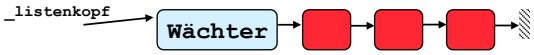
20

Sonderfälle: Beispiel Listenanfang

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



- **Ohne Wächter**: Es kann nicht in jedem Fall auf das Element **vor** der Einfügeposition positioniert werden, weil es am Listenanfang keines gibt (also im Fall `x == 0`). Die Behandlung dieses Sonderfalls (evtl. Umbiegen des Listenkopfes) muss explizit mit einer Abfrage im Quelltext berücksichtigt werden.
- **Mit Wächter**: Es wird in der Methode `einfuegen` **immer** auf das Listenelement positioniert, das unmittelbar vor der Einfügestelle liegt. Da ein Wächterelement am Anfang steht, gibt es für jede Einfügeposition immer ein solches Element. Das Einfügen sieht somit immer gleich aus, ohne bedingte Anweisung.

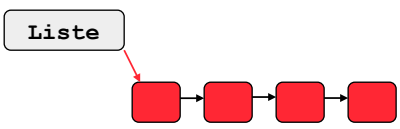


21

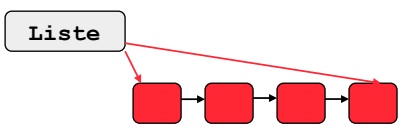
Designalternative Verweise

- 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



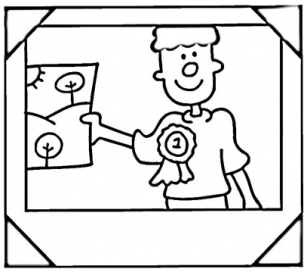
© Budd

22

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)

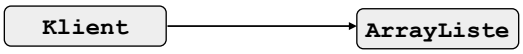
http://www.joelonsoftware.com/articles/fog0000000073.html



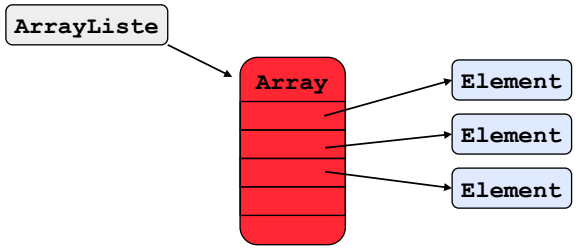
23

Lineare Strukturen für Listen: Array-Implementationen

- Grundidee: Eine Klasse, beispielsweise `ArrayListe`, repräsentiert auch hier Klienten gegenüber die gesamte Liste und bietet diesen alle Operationen einer Liste an ihrer Schnittstelle an (**Außensicht**, **Klientensicht**).



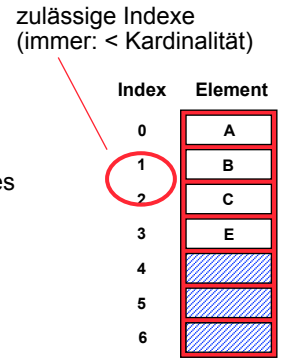
- Intern verwendet ein Exemplar dieser Klasse ein **Array**, in dem alle bisher eingefügten Elemente gehalten werden (**Innensicht**, **Implementation**).



24

Lineare Strukturen für Listen: „Wachsende“ Arrays

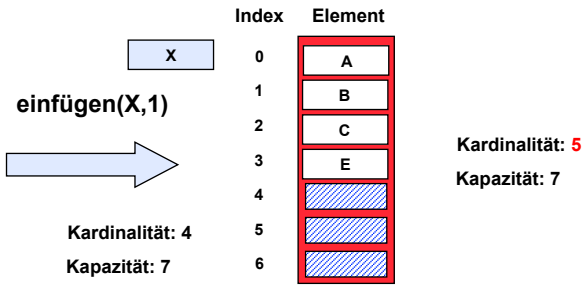
- Arrays sind als direkte Implementation für Listen ungeeignet, weil sie eine festgelegte Größe haben.
- Stattdessen wird das Konzept von **wachsenden Arrays** benutzt:
  - Erzeuge ein Array mit einer Anfangsgröße;
  - Befülle es mit den einzufügenden Elementen;
  - Wenn dieses Array voll ist, erzeuge ein größeres und kopiere alle Elemente des alten Arrays in das neue.
- Dies führt zur Unterscheidung von logischer Größe der Liste (Anzahl der Elemente, **Kardinalität**) und physikalischer Größe (**Kapazität**) des implementierenden Arrays.
- Es muss immer gelten: **Kapazität ≥ Kardinalität**



25

Wachsende Arrays: Zugriff schnell, Verschieben teuer

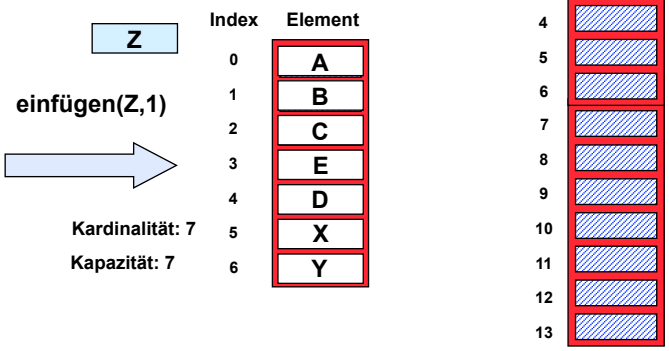
- Der Zugriff auf eine beliebige Indexposition ist sehr einfach und **schnell**: Es wird ein **indexbasierter Zugriff** auf das Array ausgeführt.
- Bei jedem Einfügen oder Löschen müssen die nachfolgenden Elemente innerhalb des Arrays **verschoben** werden. Im Extremfall (Operation am Listenanfang) müssen alle Elemente um eine Position verschoben werden.



26

Wachsende Arrays: Umkopieren in größeres Array

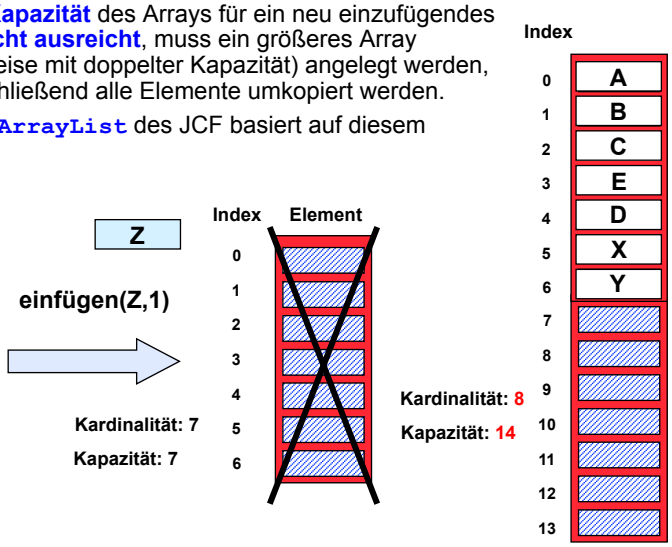
- Wenn die **Kapazität** des Arrays für ein neu einzufügendes Element **nicht ausreicht**, muss ein größeres Array (beispielsweise mit doppelter Kapazität) angelegt werden, in das anschließend alle Elemente umkopiert werden.
- Die Klasse `ArrayList` des JCF basiert auf diesem Konzept.



27

Wachsende Arrays: Umkopieren in größeres Array

- Wenn die **Kapazität** des Arrays für ein neu einzufügendes Element **nicht ausreicht**, muss ein größeres Array (beispielsweise mit doppelter Kapazität) angelegt werden, in das anschließend alle Elemente umkopiert werden.
- Die Klasse `ArrayList` des JCF basiert auf diesem Konzept.



28

Vergleich der Implementationen

- Beide Implementationen für Listen haben ihre Stärken und Schwächen.
- Um dies zu verdeutlichen, vergleichen wir die beiden Implementationen des JCF:
  - `LinkedList` (verkettete Liste)
  - `ArrayList` (wachsende Arrays)
  - Wir betrachten den Aufwand für zwei typische Operationen:
    - **Einfügen** eines Elementes
    - **Zugriff** auf ein Element an einer bestimmten Position

29

Einfügen in eine Liste

- `LinkedList`:
  - Nachteil: Position, an der eingefügt werden soll, ist erst durch **Traversieren** der Liste zu erreichen. Im Durchschnitt wird dabei die halbe Liste abgelaufen.
  - Nachteil: **Objekterzeugung** für jede Einfügung.
  - Vorteil: Das **Einfügen** ist sehr einfach (einfaches Umketten, konstanter Aufwand).
- `ArrayList`:
  - Nachteil: Alle Elemente nach der Einfügeposition müssen um eine Position verschoben werden. Im Durchschnitt wird dabei die **halbe Liste umkopiert**.
  - Nachteil: Wenn die Kapazität ausgeschöpft ist, muss ein **neues Array** angelegt und alle Elemente müssen umkopiert werden.
  - Vorteil: die Position zum Einfügen kann **direkt angesprochen** werden (konstanter Aufwand).

30

Zugriff auf eine beliebige Position; Fazit

- Beim Zugriff auf eine beliebige Position spielt die `ArrayList` ihre Stärke voll aus:
  - Der Zugriff erfolgt in konstanter Zeit, während bei der `LinkedList` durchschnittlich die halbe Liste durchlaufen werden muss.
- Insgesamt zeigt sich, dass die Implementationen für unterschiedliche Benutzungsprofile einer Liste unterschiedlich geeignet sind:
  - Für **relativ konstante** Listen, bei denen **häufig** wahlfrei auf beliebige Positionen **zugegriffen** wird, ist die `ArrayList` besser geeignet.
  - Für sehr **dynamische große** Listen, bei denen **viel eingefügt** und entfernt wird (insbesondere am Listenanfang), ist die `LinkedList` eventuell die bessere Wahl.
- **Pragmatik für Java**: Für die meisten Anwendungen mit eher kleinen Listen ist die `ArrayList` die Implementation der Wahl.

`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 lediglich 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 Folgelementen. Es findet keine Verkleinerung des Arrays statt!

Aufwand für Operationen formalisiert

- Unter **Aufwand** verstehen wir die Menge an **elementaren Schritten**, die für eine zusammengesetzte Operation ausgeführt werden müssen.
  - Bsp.: Die zusammengesetzte Operation **Einfügen an Position i** auf einer verketteten Liste erfordert mehrere elementare Schritte, die sich aus dem Durchlaufen der Liste bis zur Position i, dem Erzeugen eines neuen Kettenglieds und dem Setzen der Verkettungen ergeben.
- Elementare Schritte sind:
  - **Zuweisungen** (x = y, i++, …)
  - **Vergleiche** (a <= b, next != null, …)
  - **Aufrufe mit konstantem Zeitbedarf** (Objekterzeugung kleiner Objekte, sondierende Methoden, …)

33

Konstanter und variabler Anteil des Aufwandes

- Der Aufwand für eine Operation setzt sich üblicherweise aus einem **konstanten Anteil** und einem **variablen Anteil** zusammen.
  - Der konstante Anteil ist für jede Ausführung der Operation gleich.
  - Der variable Anteil hängt von der Menge der zu verarbeitenden Daten ab.

Bsp.: Beim Einfügen in eine verkettete Liste ist der Aufwand für Erzeugen und Verketten immer gleich, das Durchlaufen hingegen ist abhängig von dem gewünschten Index. Im schlechtesten Fall muss die gesamte Liste durchlaufen werden.

34

Abschätzungen des Aufwandes

- Bei Abschätzungen des Aufwandes wird häufig vom **schlechtesten Fall** (engl.: worst case) ausgegangen. Ebenfalls verbreitet ist die Abschätzung des **Aufwandes im Mittel**.
- Außerdem wird üblicherweise von **großen Datenmengen** ausgegangen, so dass der konstante Anteil irrelevant wird und vernachlässigt werden kann.
- Der Aufwand wird dadurch zu einer **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. Die zugehörige **Komplexitätstheorie** ist ein Teilgebiet der theoretischen Informatik.

35

Ein erster Blick auf die „O-Notation“

- Wir betrachten hier die sog. „O-Notation“ oder auch *Landau-Notation,* nach dem deutschen Zahlentheoretiker Edmund Landau. Sie wird in der Mathematik und in der Informatik verwendet, um das asymptotische Verhalten von Funktionen und Folgen zu beschreiben.
- In der Informatik finden wir die O-Notation insbesondere in der Komplexitätstheorie, um verschiedene Probleme und Algorithmen danach zu vergleichen, wie "schwierig" oder aufwändig sie zu berechnen sind. Mit der O-Notation können Aufwände für algorithmische Probleme in so genannte **Komplexitätsklassen** eingeteilt werden.
- 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)

- Diese Klassen (außer der ersten) geben eine **Größenordnung** für den Aufwand **in Abhängigkeit von der zu verarbeitenden Datenmenge n**.
- Diese Notation wird ausführlich in FGI 1 sowie in Algorithmen und Datenstrukturen thematisiert.

36

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)

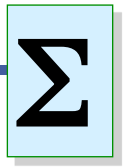
37

Test auf Enthaltensein

- Aufwand für Suchen in einer Liste: ***O(n)***
  - Die vorige Diskussion hat gezeigt, dass sowohl einfache verkettete Strukturen als auch Array-Implementationen keine günstigen Voraussetzungen für diesen Test haben: Bei beiden muss im Durchschnitt die halbe Liste durchsucht werden.
- Insbesondere bei Sets (Mengen) spielt der Test auf Enthaltensein in der Praxis häufig ein wichtige Rolle.
- Deshalb werden wir für Mengen Implementationen betrachten, die bei diesem Test erheblich effizienter sind.
- Möglich sind Realisierungen mit ***O(log n)*** und sogar mit ***O(1)***, also konstantem Aufwand!

38

Zusammenfassung



- Für Listen gibt es zwei klassische imperative Implementationen: **verkettete Listen** und **wachsende Arrays**.
- Verkettete Listen können **einfach** oder **doppelt verkettet** sein.
- Wachsende Arrays basieren auf Arrays, die bei Bedarf mit größerer Kapazität angelegt werden.
- Beide Implementationen haben Stärken und Schwächen. Mit Hilfe der **O-Notation** können wir diese Unterschiede formal greifbar machen.