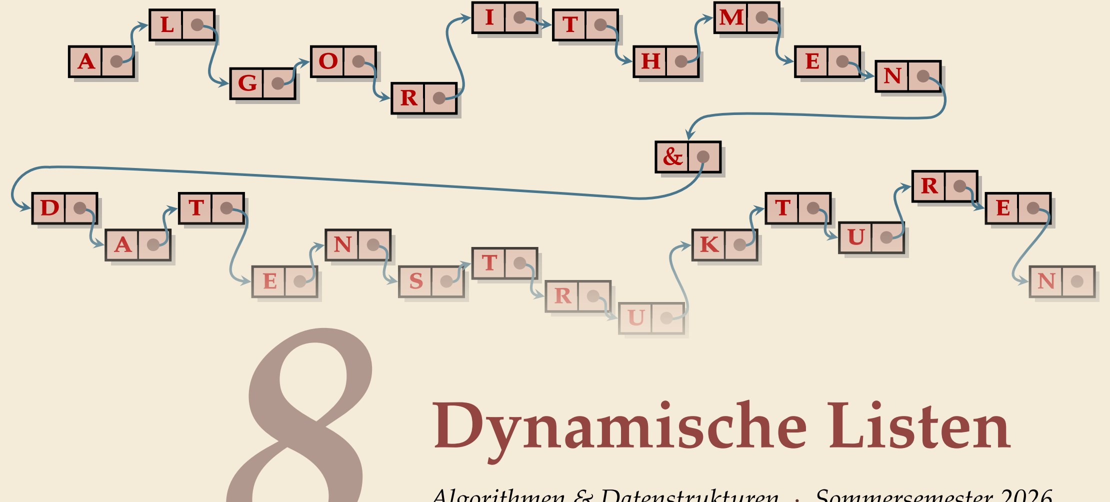


8 Dynamische Listen

Algorithmen & Datenstrukturen · Sommersemester 2026

Prof. Dr. Sebastian Wild

CS210 (Sommersemester 2026)
Philipps-Universität Marburg
version `2026-06-10 02:45 H`

Outline

8 Dynamische Listen

8.1 Abstract Data Types
8.2 Lineare Listen
8.3 Stacks & Queues
8.4 Array-basierte Listen
8.5 Einfach Verkettete Listen
8.6 Doppelt Verkettete Listen
8.7 Dynamische Arrays
8.8 Java Collections Framework

Dynamische Datenstrukturen

Bisher: *im Wesentlichen alles Arrays!* ← außer Union-Find!

- Sortieren
- Suchen
- Subsequence Sums
- 2D-Felder

Jetzt: *Dynamische Strukturen!*

$\leadsto$ Hier: Sequenzen von Elementen („Listen“)

8.1 Abstract Data Types

Abstract Data Types

Für algorithmische Probleme, starten mit Gegeben-Ziel-Spezifikation

- **Sortieren**

  Gegeben: Array $A[0..n)$ von n Objekten, Sortierkriterium

  Ziel: Array so permutieren, dass Objekte aufsteigend sortiert sind.

Abstrakter Datentyp (ADT) ist das Pendant für Datenstrukturaufgaben

- spezifiziert Operationen (Queries, Updates) auf Daten mit Semantik
- Bei Queries: Angabe welcher Rückgabewert produziert werden soll
- Bei Updates: Angabe, wie sich der Inhalt der Datenstruktur semantisch ändert

$\approx$ Java Interface

$\neq$ Implementierung/Datenstruktur: ADT gibt nur an *was* passieren soll, nicht *wie*

Beispiel

ADT *Union-Find (Disjunkte Mengen)*

- Operationen: `construct`(n), `union`(p, q), `find`(p)
- Semantik
  - `construct` gibt Größe n des Universums fest vor (unveränderlich)
  - Zu jedem Zeitpunkt haben wir eine Partition $S_0, \ldots, S_{k-1}$ von $\{0, \ldots, n-1\}$ für ein k.
  - Initial ist $k = n$ und $S_i = \{i\}$
  - Query `find`(p) gibt das eindeutige i zurück mit $p \in S_i$
  - Update `union`(p, q): mit $i =$ `find`(p) und $j =$ `find`(q) ersetze S_i und S_j durch $S_i \cup S_j$.
- Java Interface

```
1  public interface UnionFind {
2      // UnionFind(int n);
3      // (Konstruktoren können im interface angegeben werden.)
4
5      /** gibt das eindeutige i zurück mit p ∈ S_i */
6      int find(int p);
7
8      /** ersetze S_i und S_j durch S_i ∪ S_j für i = find(p), j = find(q) */
9      void union(int p, int q);
10 }
```

Java Generics & Collections

Weit-verbreitete Datenstrukturen: **Container / Collections**

- bestehen aus einer Sammlung von Objekten
- Objekte für Container eine "black box" ⇝ egal was drin steckt
- typische Anwendung: erlauben nur „sortenreine“ Elemente
  - Liste von Strings
  - Menge von Dateien
  - Matrix von ganzen Zahlen

⇝ möchten „sortenrein“ spezifizieren können, ohne vorher „Sorte“ festzulegen

Analog zu `int[]`, `String[]`, etc.!

Typparameter! (in Java: *Generics*)

- Klassen und Interfaces in Java können Typparameter haben
- wird in Klasse verwendet wie ein konkreter Klassenname
- erlaubt aber (bei Instanziierung) das Einsetzen beliebiger Klassen
- Beispiel `Comparator<T>`

8.2 Lineare Listen

Lineare Listen

Fundamentaler Container: **ADT *Lineare Liste (Dynamische Sequenz)***

- Semantik: zu jedem Zeitpunkt ist L eine Sequenz $a_1, a_2, \ldots, a_n$ von Elementen.

```java
interface LinearList<Elem, Position extends LinearList.ListPosition> {
    Position start();
    Position end();
    Position next(Position p);
    Position previous(Position p);

    void insert(Position p, Elem x);
    void delete(Position p);
    Elem get(Position p);
    void set(Position p, Elem x);

    /** Represents a position of an element in the list */
    interface ListPosition { }

    // default methods (siehe unten)
}
```

- zweiter Typparameter `Position` hier nötig, weil die Semantik von `Position` von Implementierung abhängt
  - ⇝ dazu gleich mehr!

Lineare Listen – Positionen

Für $L = a_1, \ldots, a_n$ können wir als Positionen einfach Indices $\texttt{p} \in \mathbb{N}_{\geq 1}$ angeben

- $\texttt{L.start()} \;:=\; 1$ erstes Element
- $\texttt{L.end()} \;:=\; n + 1$ *hinter* letztem Element
- $\texttt{L.next(p)} \;:=\; \begin{cases} \texttt{p} + 1 & \text{falls } \texttt{p} \neq \texttt{L.end()}, \\ \texttt{Exception} & \text{sonst.} \end{cases}$
- $\texttt{L.previous(p)} \;:=\; \begin{cases} \texttt{p} - 1 & \text{falls } \texttt{p} \neq \texttt{L.start()}, \\ \texttt{Exception} & \text{sonst.} \end{cases}$

(*Positionen erscheinen unnötig?* $\rightsquigarrow$ *siehe unten*)

Lineare Listen – Operationen

Eigentliche Operationen $\mathtt{L} = a_1, a_2, \ldots, a_n$ und $\mathtt{p} \in [1..n+1]$:

- $\mathtt{L.get(p)} := \begin{cases} a_{\mathtt{p}} & \text{falls } \mathtt{p} \neq \mathtt{L.end()}, \\ \textit{undefiniert} & \text{sonst.} \end{cases}$
- `L.set(p,x)` wirft `Exception` falls $\mathtt{p} = \mathtt{L.end()}$ und terminiert sonst normal.

  Danach gilt $\mathtt{L} = \begin{cases} a_1, \ldots, a_{\mathtt{p}-1}, \mathtt{x}, a_{\mathtt{p}+1}, \ldots, a_n & \text{falls } \mathtt{p} \neq \mathtt{L.end()}, \\ \textit{undefiniert} & \text{sonst.} \end{cases}$
- Nach `L.insert(p,x)` gilt $\mathtt{L} = a_1, \ldots, a_{\mathtt{p}-1}, \mathtt{x}, a_{\mathtt{p}}, a_{\mathtt{p}+1}, \ldots, a_n$.

  Beachte: $\mathtt{p} = \mathtt{L.end()}$ erlaubt! $\rightsquigarrow$ `L.insert(p,x)` hängt `x` ans Ende von `L` an.
- `L.delete(p)` wirft `Exception` falls $\mathtt{p} = \mathtt{L.end()}$ und terminiert sonst normal.

  Danach gilt $\mathtt{L} = \begin{cases} a_1, \ldots, a_{\mathtt{p}-1}, a_{\mathtt{p}+1}, \ldots, a_n & \text{falls } \mathtt{p} \neq \mathtt{L.end()}, \\ \textit{undefiniert} & \text{sonst.} \end{cases}$

Wozu Positionen?

Es scheint ganz und gar unnötig kompliziert, dass wir Operationen für Positionen haben.

Wir könnten doch einfach direkt den Index `p` übergeben; was soll all der Ärger?

Jein . . . wir haben nämlich eine Subtilität unterschlagen:

Was passiert bei `insert(L,p)` / `delete(L,p)` mit der Position `p`?

Antwort: Implementierungen von Linearen Listen können Realisierungen wählen!

Index-basierte Positionen

- entspricht obigem Modell einer mathematischen Sequenz
- $p \in \mathbb{N}_{\geq 1}$ stets das p-te Element vom Anfang auch nach `insert`/`delete`

⇝ *Updates links von p ändern Wert von `get(p)`!*

Pointer-basierte Positionen

- entspricht einer verketteten Liste (Index vielleicht gar nicht bekannt)
- Position verweilt auf „ihrem“ Element auch nach `insert`

⇝ *Updates anderswo ändern Wert von `get(p)` **nicht***

Abgeleitete Operationen

Einige Hilfsoperationen für Lineare Listen können wir implementieren, ohne zu wissen, welche Listen-Implementierung wir vorliegen haben!

Ultimativer wiederverwendbarer Code!

- `L.prepend(x)`: fügt `x` am Anfang (vor allen existierenden Elementen) von `L` an.

```
1     default void prepend(Elem x) { insert(start(), x); }
```

- `L.append(x)`: hängt `x` ans Ende von `L` an.

```
1 // public interface LinearList...
2     default void append(Elem x) {
3         insert(isEmpty() ? start() : end(), x);
4     }
```

- `L.isEmpty()`: liefert `true`, wenn `L` keine Elemente enthält und `false` sonst.

```
1     default boolean isEmpty() { return start().equals(end()); }
```

Abgeleitete Operationen [2]

- `L.firstOccurrence(x)`: erste Position in `L`, an der `x` in der Liste vorkommt bzw. `L.end()` falls `x` gar nicht in der Liste vorkommt.
- `L.firstOccurrenceAfter(p,x)`: erste Position in `L` hinter `p` (inklusive `p` selbst), an der `x` in der Liste vorkommt bzw. `L.end()` falls `x` dort nicht in der Liste vorkommt.

```
1      default Position firstOccurrence(Elem x) {
2          return firstOccurrenceAfter(start(), x);
3      }
4      default Position firstOccurrenceAfter(Position p, Elem x) {
5          for (Position cur = p; !cur.equals(end()); cur = next(cur))
6              if (get(cur).equals(x)) return cur;
7          return end();
8      }
9  // } end of interface LinearList
```

- Beachte: Können hier nur die lineare Suche verwenden.

8.3 Stacks & Queues

Eingeschränkte ADTs

Bevor zu Implementierungen von Linearen Listen kommen, betrachten wir zwei Spezialfälle: Stacks und Queues

- Da beide nur Teile der Operationen von Linearen Listen brauchen ist jede Implementierung von Linearen Listen auch ein Stack und eine Queue

⇝ *brauchen keine neuen Implementierungen*

Eingeschränkte ADTs trotzdem oft sinnvoll:

- Kommunizieren anderen Entwickler:innen, dass hier nur Spezialfall gebraucht
  - ⇝ mehr Information über Ihren Code
- (manchmal) bessere Implementierung für einfachere Anforderung möglich

Stacks

ADT *Stack* (*Stapel, Kellerspeicher*)

- `top()`: Gib oberstes Element zurück. (Stack unverändert)
- `push(`x`)`: Lege x oben auf den Stack (Einfügen).
- `pop()`: Entferne oberstes Element und gib es zurück
- `empty()`: `true` gdw. Stack leer

```
public interface Stack<Elem> extends Iterable<Elem> {
    void push(Elem x);
    Elem pop();
    Elem top();
    boolean empty();
}
```

Queues

ADT *Queue* (*Warteschlange*):

- `enqueue(`x`)`: Füge x am Ende der Queue ein.
- `dequeue()`: Entferne Element am Beginn der Queue und gib es zurück.

```java
public interface Queue<Elem> extends Iterable<Elem> {
    void enqueue(Elem x);
    Elem dequeue();
    Elem peek();
    boolean empty();
}
```

Bags

Manchmal wollen wir kommunizieren, dass die Reihenfolge der Elemente völlig egal ist.

ADT *Bag/Collection* (*Sack*)

- `insert(`x`)`: Füge x zur Bag hinzu
- `delAny()`: Entferne *beliebiges* Element aus Bag und gib es zurück.
  (Reihenfolge unspezifiziert)
- Java's `java.util.Collection`

Meist realisiert als Stack oder Queue, aber Reihenfolge soll für Benutzer der Bag keinen Unterschied machen.

Deque

ADT *Deque* (*Double-Ended Queue*)

- Einfügen und Entfernen von Elementen an beiden Enden erlaubt
  - `addFirst(`x`)`
  - `addLast(`x`)`
  - `removeFirst()`
  - `removeLast()`
  - `getFirst()`
  - `getLast()`

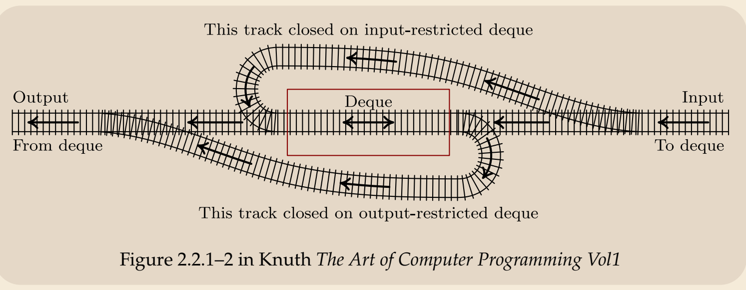


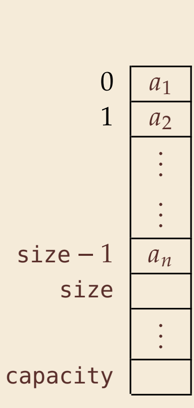
Figure 2.2.1–2 in Knuth *The Art of Computer Programming Vol1*

8.4 Array-basierte Listen

Array-basierte Listen

Idee: Speichere Elemente der Linearen Liste in einem Array.

- Array-basierte Lineare Listen bleiben am nächsten an mathematischer Sequenz
- Array hat feste Größe ⇝ im Konstruktor fest gewählt.




```
1  public class FixedSizeArrayList<Elem>
2            implements LinearList<Elem, FixedSizeArrayList.Position> {
3      private final int capacity;
4      private final Object[] elements;
5      private int size = 0;
6      /** Konstruktor */
7      public FixedSizeArrayList(int capacity) {
8          this.capacity = capacity;
9          elements = new Object[capacity];
10     }
11     /** Represents an array index */
12     public static class Position implements LinearList.ListPosition {
13         private final int index;
14         public Position(int index) { this.index = index; }
15         public boolean equals(Object o) {
16             return index == ((IndexPosition) that).index;
17         }
18         public int hashCode() { return index; }
19     }
20     // methods
21 }
```

Array-basierte Listen – Günstige Operationen

0	a_1
1	a_2
	⋮
	⋮
size − 1	a_n
size	
	⋮
capacity	

```java
public class FixedSizeArrayList<Elem> { // continued
    public Position start() { return new Position(0); }
    public Position end() { return new Position(size); }
    public Position next(Position p) {
        if (p.index == size) throw new IndexOutOfBoundsException();
        return new Position(p.index+1);
    }
    public Position previous(Position p) {
        if (p.index == 0) throw new IndexOutOfBoundsException();
        return new Position(p.index-1);
    }
    @SuppressWarnings("unchecked")
    public Elem get(Position p) { return (Elem) elements[p.index]; }
    public void set(Position p, Elem x) { elements[p.index] = x; }
    // more methods
}
```

- alle diese Operationen benötigen nur $O(1)$ Laufzeit

Array-basierte Listen – Teure Operationen

0	a_1
1	a_2
	⋮
	⋮
size − 1	a_n
size	
	⋮
capacity	

```java
public class FixedSizeArrayList<Elem> { // continued
    public void insert(Position p, Elem x) {
        if (size >= capacity) throw new CapacityExceededException();
        for (int j = size - 1; j >= p.index; --j)
            elements[j + 1] = elements[j];
        elements[p.index] = x; ++size;
    }
    public void delete(Position p) {
        if (p.equals(end())) throw new IndexOutOfBoundsException();
        for (int j = p.index; j <= size-2; ++j)
            elements[j] = elements[j+1];
        elements[size-1] = null;  // avoid loitering
        --size;
    }
}
```

- können zwar direkt auf Position `p` zugreifen, müssen aber alle Elemente rechts von `p` um eine Position verschieben

⇝ Worst-Case $O(n)$ Laufzeit

↯ keine allgemeine Implementierung: `insert` sollte keine Exception werfen

  - mit festem Array nicht vermeidbar

Array-Basierter Stack

Für Stacks können wir jegliches Kopieren vermeiden, da Updates nur an einem Ende passieren!

Müssen dafür nur konzeptionell das Array auf den Kopf stellen:

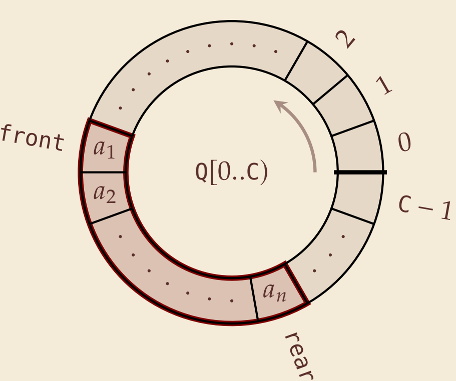
	capacity
⋮	
	size
a_n	size − 1
⋮	
⋮	
a_2	1
a_1	0

```java
public class ArrayStack<Elem> implements Stack<Elem> {
    private final Object[] contents;
    private int size = 0;

    public ArrayStack(int capacity) { contents = new Object[capacity]; }

    public void push(Elem x) {
        if (size == contents.length) throw new Exception();
        contents[size++] = x;
    }
    public Elem pop() {
        Elem result = top();
        contents[--size] = null; // avoid loitering
        return result;
    }
    @SuppressWarnings("unchecked")
    public Elem top() { return (Elem) contents[size-1]; }
}
```

Ringpuffer

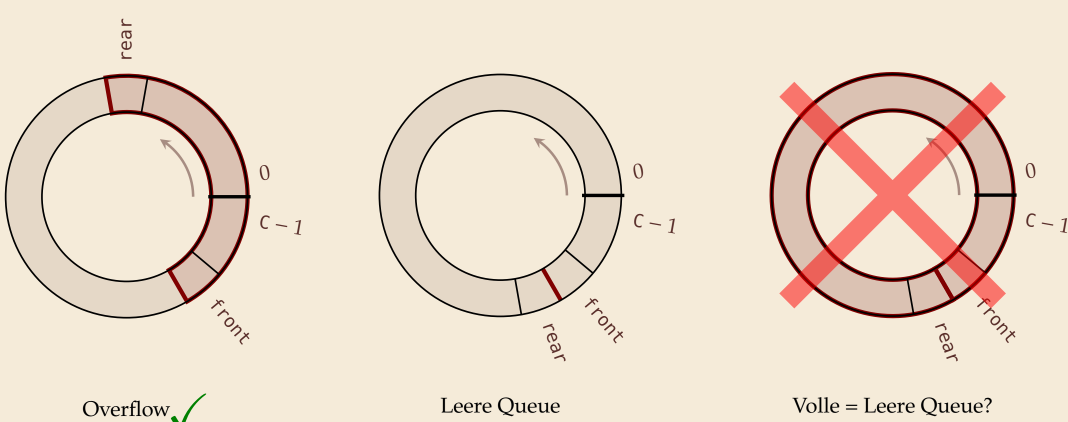
Für Queues haben wir Updates an beiden Rändern . . . Kopieren nötig?
Nein! Wir müssen nur die Repräsentation flexibel machen.

- ▶ `enqueue` hängt neues Element hinten an, `dequeue` löscht vorne
- ⇝ Belegter Bereich verschiebt sich durch das Array nach hinten
- ▶ Stellen uns Array **ringförmig** vor
- ▶ Elemente belegen ***zyklisch* zusammenhängenden** Bereich
- ⇝ Betrachte Positionen modulo `C`, der Länge des Arrays



Ringpuffer Randfälle

Invariante: i-tes Element in `Q`[(`front` $+ i - 1$) mod C] $\quad (i = 0, \ldots, n-1)$
Erlaubte Anzahl Elemente ist $n \in [0..C-1]$



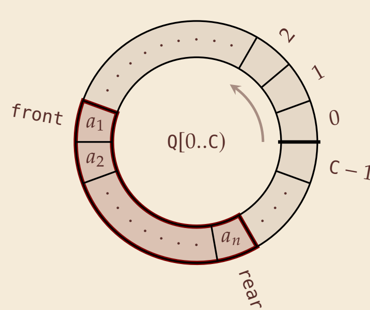
Overflow ✓

Leere Queue

Volle = Leere Queue?

Ringpuffer Implementierung

```
public class ArrayQueue<Elem> implements Queue<Elem> {
    private final Object[] Q;
    private int front, rear, C;
    public ArrayQueue(final int capacity) {
        C = capacity + 1;  // allow capacity elems
        Q = new Object[C];
        front = 0;  rear = Q.length-1;
    }
    public void enqueue(final Elem x) {
        if (full()) throw new CapacityExceededException();
        rear = (rear + 1) % C;  Q[rear] = x;
    }
    public Elem dequeue() {
        Elem res = peek();
        Q[front] = null; // prevent loitering
        front = (front + 1) % C;
        return res;
    }
    @SuppressWarnings("unchecked")
    public Elem peek() {
        if (empty()) throw new NoSuchElementException();
        return (Elem) Q[front];
    }
    public boolean empty() { return (rear + 1) % C == front; }
    public boolean full()  { return (rear + 2) % C == front; }
}
```



- alle Operationen $O(1)$ Zeit
- wieder: beschränkte Kapazität
  - ⇝ keine allgemeine Queue
- Anwendung in real-time streams
  - entferne full-check
  - ⇝ Überschreibt alte Daten

Array-basierte Listen – Zusammenfassung

- 👍 für einige Operationen sehr schnell
- 👎 Beschränkte Kapazität keine allgemeine Lösung.
- 👎 Können nicht effizient in der Mitte einfügen/löschen.

8.5 Einfach Verkettete Listen

Grenzenlose Listen

Für wirklich dynamische Listen brauchen wir mehr Flexibilität.

"All problems in computer science can be solved by another level of indirection" attributed to David Wheeler

Statt sequentieller Speicherung ...

Adresse	Inhalt
1	a_1
2	a_2
3	a_3
4	a_4
5	a_5

Jetzt: Jedes Element verweist auf Nachfolger

Adresse	Inhalt	
A	a_1	B
B	a_2	C
C	a_3	D
D	a_4	E
E	a_5	$\perp$

$\rightsquigarrow$ **Jede Zeile durch ein Objekt repräsentiert**

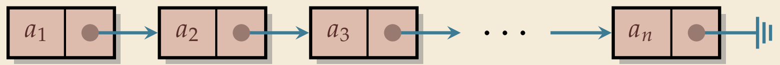
- Adresse ist Objektreferenz / Pointer
- graphisch: $a_1 \to a_2 \to a_3 \to a_4 \to a_5 \to$ ⊥

Einfach verkettete Listen

Einfachste verkettete Darstellung: jedes Element kennt Nachfolger `next`.

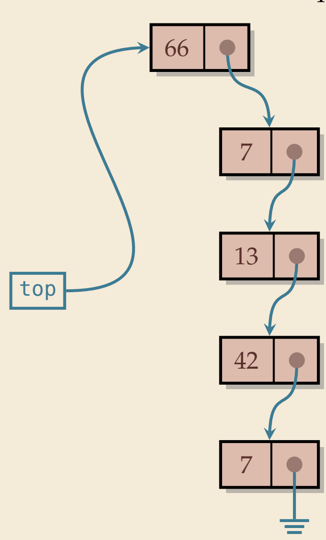
```
public class LinkedList<Elem>
        implements LinearList<Elem, LinkedList.Position> {
    // ...
    private static final class ListNode extends Position {
        Object element;
        ListNode next;
        public ListNode() { }
        public ListNode(Object element, ListNode next) {
            this.element = element;  this.next = next;
        }
    }
    public static abstract class Position
            implements LinearList.ListPosition {
        private Position() { }  // keine Instanzen außerhalb
    }
}
```

- Klasse `ListNode` repräsentiert Eintrag
- verwenden private innere Klasse für Kapselung
- `ListNode` fungiert gleichzeitig als Positions-Objekt
- (Positionen hier Pointer-basiert)



Verkettete Stacks

Funktioniert besonders bequem für *Stacks*!




```java
public class LinkedStack<Elem> implements Stack<Elem> {
    ListNode top = null;
    private static final class ListNode {
        Object element;  ListNode next;
        public ListNode(Object element) {
            this.element = element;
        }
    }
    @SuppressWarnings("unchecked")
    public Elem top() {
        if (empty()) throw new Exception();
        return (Elem) top.element;
    }
    public void push(Elem x) {
        ListNode newTop = new ListNode(x);
        newTop.next = this.top;
        this.top = newTop;
    }
    public Elem pop() {
        Elem result = top();
        top = top.next;
        return result;
    }
    public boolean empty() { return top == null; }
}
```

⇝ *keine inhärente Begrenzung für Stackgröße mehr!*

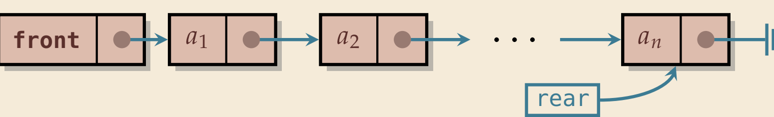
Verkettete Queue

Queues sind fast genauso einfach.

- Hier hilft es aber, einen Dummy-Knoten `front` stets vorhalten

⇝ brauchen keine Fallunterscheidung für leere Queue in `enqueue`/`dequeue` 🤓

```java
public class LinkedQueue<Elem>
                implements Queue<Elem> {

    ListNode front = new ListNode(null),
             rear = front;
    private static final class ListNode {
        Object element;  ListNode next;
        public ListNode(Object e) { element = e; }
    }
    public void enqueue(Elem x) {
        ListNode newRear = new ListNode(x);
        rear.next = newRear;
        rear = newRear;
    }
    public Elem dequeue() {
        Elem result = peek();
        front = front.next;
        return result;
    }
    @SuppressWarnings("unchecked")
    public Elem peek() {
        if (empty()) throw new Exception();
        return (Elem) front.next.element;
    }
    public boolean empty() {
        return front.next == null;
    }
}
```

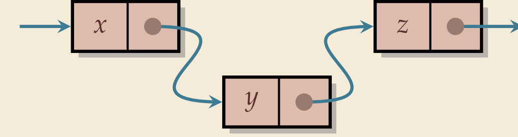


Updates in der Mitte

Updates mitten in einer Liste sind ebenso einfach und effizient!

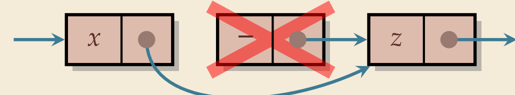
Insert: Füge y zwischen x und z ein.

1. Lege neuen Knoten für y an.
2. Setze y`.next` := z
3. Setze x`.next` := y



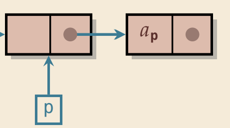
Delete: Lösche y aus Liste

1. Sei x Vorgänger von y
2. Setze x`.next` := y
3. Gebe y frei



⚠ *Brauchen Zugriff auf Vorgänger zum Löschen*

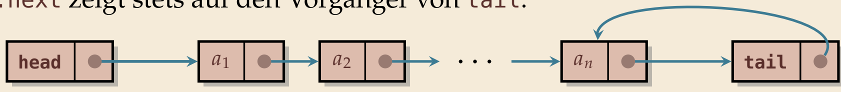
⇝ **Konvention:** Position zeigt auf Knoten **vor** aktuellem Element



Einfach verkettete linear Liste

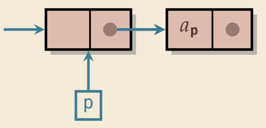
Viele Randfälle im Code kann man elegant mit folgender Konvention umgehen:

- Eine lineare Liste enthält immer die beiden Dummy-Knoten `head` und `tail`
- `tail.next` zeigt stets auf den Vorgänger von `tail`.




```
1  public class LinkedList<Elem> implements
2        LinearList<Elem, LinkedList.Position> {
3      int size = 0;
4      private final ListNode head = new ListNode(),
5                             tail = new ListNode();
6
7      // ... ListNode, Position (siehe oben)
8
9      public LinkedList() {  // initialize to empty
10         head.next = tail;  tail.next = head;
11     }
12
13     public Position start() { return head; }
14
15     public Position end() { return tail.next; }
16     public Elem get(Position p) {
17         if (p == end()) throw new Exception();
18         return (Elem) ((ListNode) p).next.element;
19     }
20     public void set(Position p, Elem x) {
21         if (p == end()) throw new Exception();
22         ((ListNode) p).next.element = x;
23     }
24     public Position next(final Position p) {
25         if (p == end()) throw new Exception();
26         return ((ListNode) p).next;
27     }
28     public int size() { return size; }
29     // ... insert / delete (nächste Folie)
30 }
```

Einfach verkettete lineare Liste [2]



Konvention: (1) Position zeigt auf Knoten **vor** aktuellem Element
(2) `tail.next` zeigt stets auf den Vorgänger von `tail`

```
1  public class LinkedList<Elem> // continued
2      public void insert(Position p, Elem x) {
3          ListNode pred = (ListNode) p;
4          ListNode newNode =
5              new ListNode(x, pred.next);
6          pred.next = newNode;  ++size;
7          // Invariante (2) herstellen
8          if (newNode.next == tail)
9              tail.next = newNode;
10     }
```

```
11     public void delete(Position p) {
12         if (p == end()) throw new Exception();
13         final ListNode pred = (ListNode) p;
14         pred.next.element = null; // no loitering
15         pred.next = pred.next.next;  --size;
16         // Invariante (2) herstellen
17         if (pred.next == tail)
18             tail.next = pred;
19     }
20 }
```

- `insert` fügt direkt vor `p` ein; `p` verweist danach auf das neue Element
- Nach `delete` zeigt `p` auf Nachfolger

⇝ ⚠ Löschen des Vorgängers macht Position ungültig!

DIY Memory Management

In Java findet JVM für `new ListNode()` (normalerweise) ein freies Stück Speicher auf dem Heap.

Wie eigentlich?

Für Performance-kritische Anwendungen oder beim Implementieren von Low-Level-Frameworks kann es sein, dass wir das selbst in die Hand nehmen müssen.

```
public class DIYMemoryManagement {
    private MemoryCell[] mem;
    public static final int NULL = -1;
    // ...
    public MemoryCell get(int address) {
        return mem[address];
    }
    public static class MemoryCell {
        Object element;  int next;
        public MemoryCell(Object element, int next) {
            this.element = element;
            this.next = next;
        }
    }
}
```

Geteilter Speicher

2 Listen a_1, a_2, a_3 und b_1, b_2, b_3, b_4 und *Free-List*

0		1
1		4
2	b_1	14
3	b_4	-1
4		13
5	a_1	12
6	b_3	3
7		10
8	a_3	-1
9		11
10		-1
11		7
12	a_2	8
13		9
14	b_2	6

DIY Memory Management – Operationen

- Verwalten *Free-List* als Stack von `MemoryCells`
- `alloc` entspricht `pop`; `free` entspricht `push`

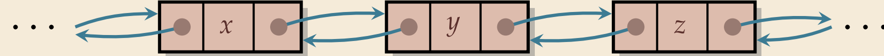
```java
public class DIYMemoryManagement { // continued
    private int nextFree = 0;

    public DIYMemoryManagement(int memorySize) {
        this.mem = new MemoryCell[memorySize];
        for (int i = 0; i < mem.length; i++)
            mem[i] = new MemoryCell(null,i+1);
        mem[memorySize-1].next = NULL;
    }
    public int alloc() {
        if (nextFree == NULL) throw new OutOfMemoryError();
        int result = nextFree;
        this.nextFree = mem[result].next;
        return result;
    }
    public void free(int address) {
        mem[address].element = null;
        mem[address].next = nextFree;
        nextFree = address;
    }
}
```

8.6 Doppelt Verkettete Listen

Doppelte Verkettung

Einfach verkettete Listen können nicht effizient den Vorgänger finden.

Unter Verdopplung der Anzahl Pointer können wir auch das unterstützen.

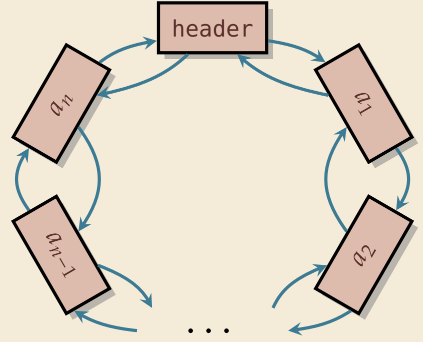



```
public class DoublyLinkedList<Elem> implements
        LinearList<Elem, DoublyLinkedList.Position> {
    private static final class ListNode extends Position {
        Object element;  ListNode next, previous;
        ListNode(Object element) { this.element = element; }
    }
```

Geschickte Konvention für Randfälle:

- Dummy-Knoten `header` existiert immer
- Liste ist **zyklisch geschlossen**.

```
    private final ListNode header;
    public DoublyLinkedList() {
        header = new ListNode(null);
        header.next = header; header.previous = header;
    }
    // ...
}
```



Doppelt verkettete Listen

```java
public class DoublyLinkedList<Elem> { // continued
    public Position start() { return header.next; }
    public Position end() { return header; }
    public Position next(Position p) {
        if (p.equals(end())) throw new Exception();
        return ((ListNode) p).next;
    }
    public Position previous(Position p) {
        if (p.equals(start())) throw new Exception();
        return ((ListNode) p).previous;
    }
    @SuppressWarnings("unchecked")
    public Elem get(Position p) {
        if (p.equals(end())) throw new Exception();
        return (Elem) ((ListNode) p).element;
    }
    public void set(Position p, Elem x) {
        if (p.equals(end())) throw new Exception();
        ((ListNode) p).element = x;
    }

    /** direkt vor p einfügen */
    public void insert(Position p, Elem x) {
        ListNode newNode = new ListNode(x);
        final ListNode node = ((ListNode) p);
        newNode.previous = node.previous;
        newNode.next = node;
        node.previous.next = newNode;
        node.previous = newNode;
    }

    public void delete(Position p) {
        if (p.equals(end())) throw new Exception();
        final ListNode node = ((ListNode) p);
        node.previous.next = node.next;
        node.next.previous = node.previous;
    }

}
```

- `insert` fügt direkt vor `p` ein; `p` verbleibt auf altem Element
  - ⇝ Andere Semantik als `LinkedList`!
- Nach `delete` zeigt `p` weiter auf gelöschten Knoten

⇝ ⚠ Löschen macht (diese) Position ungültig, aber andere nicht.

Dünn besetzte Matrizen

Matrizen und Vektoren enthalten in gewissen Anwendungen viele 0-Einträge.

- Übliche Darstellung als 2D-Array verschwendet Platz
- Operationen verschwenden Zeit, die 0-Einträge zu besuchen

Einfache Idee: speichere Liste der Nicht-Null-Einträge

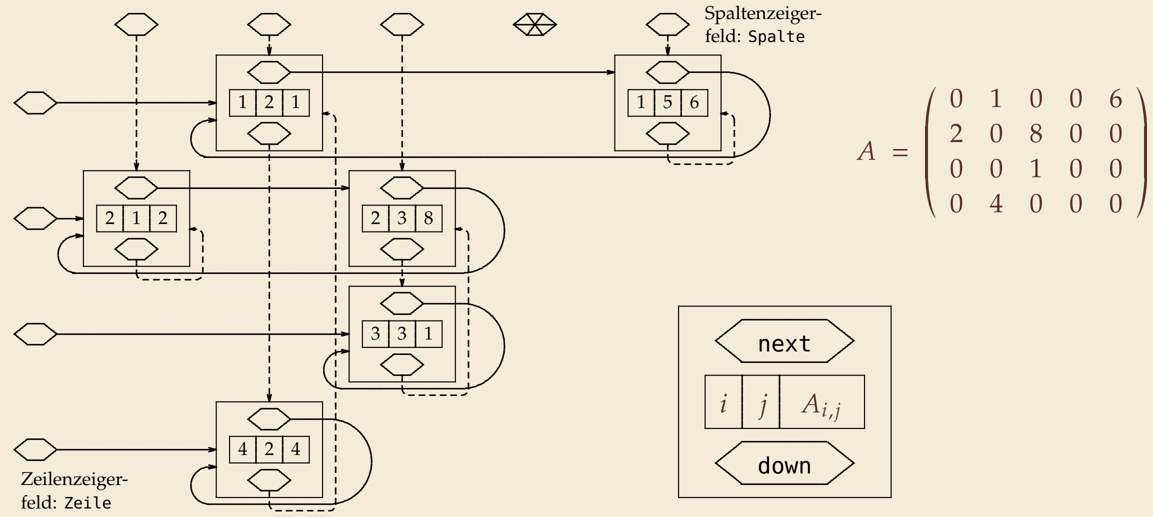
- Nullen brauchen gar keinen Platz!
- müssen aber Position / Indizes für Nicht-Nullen zusätzlich speichern

Beispiel:

$$A = \begin{pmatrix} 0 & 9 & 0 \\ 7 & 0 & 5 \\ 0 & 0 & 6 \end{pmatrix} \quad \leadsto \quad \longrightarrow \boxed{1}\boxed{2}\boxed{9} \longrightarrow \boxed{2}\boxed{1}\boxed{7} \longrightarrow \boxed{2}\boxed{3}\boxed{5} \longrightarrow \boxed{3}\boxed{3}\boxed{6} \longrightarrow$$

*Für Matrix-Multiplikation benötigen wir aber effizienten Zugriff auf Zeilen **und** Spalten.*

Dünn besetzte Matrizen – Reprästentation



$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 6 \\ 2 & 0 & 8 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 \end{pmatrix}$$

Abb. 2.17, Nebel & Wild, *Entwurf und Analyse von Algorithmen*, 2018

Verkettete Listen – Zusammenfassung

- 👍 Verkettete Stacks und Queues sind einfach und unbeschränkt in der Größe
  - gleichzeitig weiterhin $O(1)$ Zeit für alle Operationen!
- 👍 Verkettete Listen unterstützen $O(1)$ Einfügen und Löschen an **beliebiger** Position
- 👍 Können effizient über Liste iterieren

- 👎 viel Speicher geht für Pointer drauf
- 👎 kein (effizienter) *random access* („wahlfreier Zugriff“, Zugriff per Index)
- 👎 *“pointer chasing”* (Folgen von Referenzen) relativ langsam
  - typischerweise ein Cache Miss pro Schritt
  - Allozieren und garbage collection erzeugen Overhead

8.7 Dynamische Arrays

Dynamische Arrays

Relativ häufiges Szenario:

- Benötigen Liste, deren Gesamtlänge anfangs nicht bekannt ist
- Liste wird sukzessive gefüllt; (nur hinten anhängen genügt uns dabei)
- Würden gerne den Overhead von verketteten Listen vermeiden
  UND/ODER
  Benötigen random access in Liste

⇝ *Keine der bisherigen Lösungen passst gut.*

Wollen eigentlich wirklich ein Array – aber eines das mitwächst!

- **Idee:** Sobald Array voll, legen wir größeres Array an

müssen alle Elemente in neues Array kopieren

- *Können wir zumindest dafür sorgen, dass das teure Kopieren nur selten passiert?*

Doubling Arrays

Lösung:

- Wir halten Array $A[0..C)$ mit Kapazität C vor
- Speichern aktuell n Elemente
- `push(x)` fügt x am Ende an (in $A[n]$)
- `pop()` entfernt das letzte Element $A[n-1]$
- **Invariante:** $\frac{1}{4}C \le n \le C$

$\rightsquigarrow$ Haben stets Platz für alle Elemente!

Wie erhalten wir die Invariante aufrecht?

- vor `push`
  Falls $n = C$, alloziere neues Array der Größe $2n$, kopiere Elemente.
- after `pop`
  Falls $n < \frac{1}{4}C$, alloziere neues Array der Größe $2n$, kopiere Elemente.

Doubling Arrays

```java
public class DoublingArray<Elem> {
    private Object[] A = new Object[8]; // C=8
    private int n = 0;
    /** Stelle Invariante sicher, evtl. Kopie */
    private void ensureCapacity() {
        int C = A.length;
        if (C/4 < n && n < C || n <= 4) return;
        Object[] newA = new Object[2*n];
        System.arraycopy(A, 0, newA, 0, n); // O(n)
        A = newA;
    }
    public Integer start() { return 0; }
    public Integer end() { return n; }
    public Integer next(Integer p) {
        if (p == n) throw new Exception();
        return p + 1;
    }
    public Integer previous(Integer p) {
        if (p == 0) throw new Exception();
        return p - 1;
    }

    public void push(Elem x) {
        ensureCapacity();
        A[n++] = x;
    }
    public void pop() {
        ensureCapacity();
        A[--n] = null;
    }

    @SuppressWarnings("unchecked")
    public Elem get(Integer p) {
        if (p < 0 || p >= n) throw new Exception();
        return (Elem) A[p];
    }
    public void set(final Integer p, final Elem x) {
        if (p < 0 || p >= n) throw new Exception();
        A[p] = x;
    }
}
```

Funktioniert problemlos so . . . aber ist das Kopieren nicht sehr teuer?

Amortisierte Analyse

Theorem 8.1 (Amortisierte Kosten von dynamischen Arrays)

Eine beliebige Folge von m Operationen (`push`, `pop`, `get`, `set`) auf einem `DoublingArray` mit Anfangsgröße n hat Gesamtkosten von $O(n + m)$ Arrayzugriffen. ◀

Wir sagen in diesem Fall auch: `push` und `pop` haben *amortisierte Kosten* von $O(1)$.
(gemittelt über eine Folge von Operationen)

Formalen Beweis für Theorem 8.2 in *Effiziente Algorithmen*

Betrachten hier anschauliche Version **ohne `pop()` Aufrufe**

Arrays und Arbeitszeitkonten

Theorem 8.2 (Amortisierte Kosten von dynamischen Arrays)

Eine beliebige Folge von m `push`-Operationen auf anfänglich leerem `DoublingArray` hat Gesamtkosten von $O(m)$ Arrayzugriffen.

Beweis:

- `push` Aufruf kann „günstig" oder „teuer" sein
  - günstig = keine Vergrößerung von A nötig $\rightsquigarrow$ 1 Schreibzugriff auf Array
  - teuer = Verdoppeln A $\rightsquigarrow$ $n + 1$ Schreibzugriffe auf Array
- Jeder `push`-Aufruf muss 3 „Münzen" auf das Arbeitszeitkonto einzahlen.
  - günstiges `push` $\rightsquigarrow$ 2 Münzen übrig.
    Eine verbleibt auf dem neuen Element, andere geht an erstes Element ohne Münze.
  - teures `push`
    Behauptung: Alle Münzen einsammeln bezahlt die $n + 1$ Schreibzugriffe.
  - Zwischen letztem Kopieren und jetzt $n/2$ push-Aufrufe (sonst nicht voll)

  $\rightsquigarrow$ diese Aufrufe zahlen je 2 Münzen für zukünftiges Kopieren

  $\rightsquigarrow$ n Münzen auf dem Konto ∎

8.8 Java Collections Framework

Java Collections Framework

Viele der Datenstrukturen aus dieser Unit müssen Sie selten neu implementieren.

In der Regel kommen Sie mit den Versionen aus der Java Runtime aus!

- **Interfaces:**
  - `java.util.Collection` als Oberbegriff ($\approx$ Bag)
  - `java.util.List` entspricht unserer Linearen Liste
    unterstellt i. d. R. Index-basierte Positionen
  - eingeschränkte Interfaces `java.util.Queue` und `java.util.Deque`
    ⚠ ~~`java.util.Stack`~~ ist *deprecated* (veraltet) ⇝ nicht nutzen
    wie `Vector` sind die Methoden `synchronized` ⇝ langsam ohne Nutzen
    ⇝ stattdessen `Deque` verwenden . . .
  - Alle sind generische Typen, also z. B. `List<Integer>`
  - weitere Interfaces `java.util.Map` und `java.util.(Sorted)Set` ⇝ Unit 9–10
- **Implementierungen**
  - `java.util.LinkedList`: doppelt verkettete Liste
  - `java.util.ArrayList`: dynamische Arrays, Wachstumsfaktor 1.5 statt 2
    ⚠ schrumpft nicht automatisch! ⇝ `trimToSize()`

Methoden für Collections

Basis-Algorithmen für Collections

- `java.util.Collections.sort(list)`
- `java.util.Collections.reverse(list)`
- `java.util.Collections.shuffle(list)`

Fail-Fast Iteratoren

- **Iteratoren**
  - Positionen in Listen sind durch `java.util.Iterator` abgebildet
  - Basis-Iteratoren haben Methoden `hasNext()` und `next()` (plus `remove()`)
  - `ListIterator` kann auch rückwärts laufen und Elemente einfügen

- **Positionen**
  - Sie erinnern sich: Positionen in verketteten Listen und Array-basierten Listen verhalten sich inhärent anders
  - Java Collections enthalten `modCount`-Versionszähler
  - sobald ein Iterator eine Änderung bemerkt, erklärt er sich selbst für ungültig
  - ⇝ Operationen werfen `ConcurrentModificationException`
  - Ausnahme: Modifikation über Methoden des Iterators selbst

Iteratoren für unsere Lineare Liste

Java erlaubt Kurzschreibweise

```
for (Object x : collection) { ... }
```

statt

```
Object x = null;
for (Iterator it = collection.iterator(); it.hasNext(); x = it.next()) { ... }
```

Wir können diese auch für unsere Collections ermöglichen:

```
interface LinearList<Elem, Position> extends Iterable<Elem> {
    // ...
    default Iterator<Elem> iterator() {
        return new Iterator<Elem>() {
            Position cur = start();
            public boolean hasNext() { return !cur.equals(end()); }
            public Elem next() {
                if (!hasNext()) throw new NoSuchElementException();
                Elem elem = get(cur);
                cur = LinearList.this.next(cur);
                return elem;
            }
        };
    }
}
```

Zusammenfassung

Lineare Listen gut unterstützt.

- Wenn Einfügen oder Löschen in der Mitte wichtig ist $\rightsquigarrow$ `LinkedList`
- Sonst `ArrayList`

⚠ Java Collection Frameworks bietet auch Methoden an, die $O(n)$ Laufzeit haben