

8

Dynamische Listen

Algorithmen & Datenstrukturen · Sommersemester 2026

Prof. Dr. Sebastian Wild

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!

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

außer Union-Find!



Dynamische Datenstrukturen

Bisher: im Wesentlichen alles Arrays!

▶ Sortieren

↖ außer Union-Find!

▶ Suchen

▶ Subsequence Sums

▶ 2D-Felder

Jetzt: Dynamische Strukturen!

↪ Hier: Sequenzen von Elementen („Listen“)

8.1 Abstract Data Types

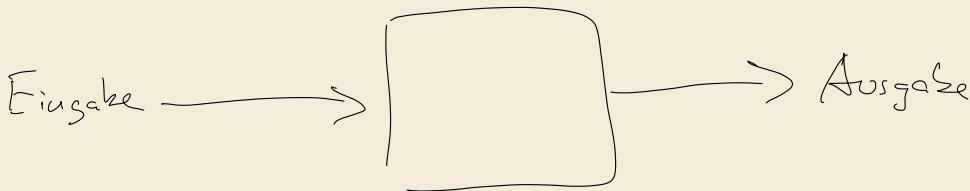
Abstract Data Types

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.



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

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

≈ Java Interface

≠ Implementierung/Datenstruktur: ADT gibt nur an *was* passieren soll, nicht *wie*

Beispiel *Dynamische Decrementelle*

ADT *Union-Find* (Disjunkte Mengen)

- ▶ Operationen: $\text{construct}(n)$, $\text{union}(p, q)$, $\text{find}(p)$

Beispiel

ADT *Union-Find* (*Disjunkte Mengen*)

- ▶ Operationen: $\text{construct}(n)$, $\text{union}(p, q)$, $\text{find}(p)$
- ▶ Semantik
 - ▶ construct gibt Größe n des Universums fest vor (unveränderlich)
 - ▶ Zu jedem Zeitpunkt haben wir eine Partition S_0, \dots, S_{k-1} von $\{0, \dots, n-1\}$ für ein k .

Beispiel

ADT *Union-Find* (Disjunkte Mengen)

- ▶ Operationen: $\text{construct}(n)$, $\text{union}(p, q)$, $\text{find}(p)$
- ▶ Semantik
 - ▶ construct gibt Größe n des Universums fest vor (unveränderlich)
 - ▶ Zu jedem Zeitpunkt haben wir eine Partition S_0, \dots, S_{k-1} von $\{0, \dots, n-1\}$ für ein k .
 - ▶ Initial ist $k = n$ und $S_i = \{i\}$

Beispiel

ADT *Union-Find* (Disjunkte Mengen)

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

Beispiel

ADT *Union-Find* (Disjunkte Mengen)

- ▶ Operationen: construct(n), $\text{union}(p, q)$, $\text{find}(p)$
- ▶ Semantik
 - ▶ construct gibt Größe n des Universums fest vor (unveränderlich)
 - ▶ Zu jedem Zeitpunkt haben wir eine Partition S_0, \dots, S_{k-1} von $\{0, \dots, n-1\}$ für ein k .
 - ▶ Initial ist $k = n$ und $S_i = \{i\}$
 - ▶ Query $\text{find}(p)$ gibt das eindeutige i zurück mit $p \in S_i$
 - ▶ Update $\text{union}(p, q)$: mit $i = \text{find}(p)$ und $j = \text{find}(q)$ 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  $p \in S_i$  */
6     int find(int p);
7
8     /** ersetze  $S_i$  und  $S_j$  durch  $S_i \cup S_j$  für  $i = \text{find}(p)$ ,  $j = \text{find}(q)$  */
9     void union(int p, int q);
10 }
```

Clicker Question



Welche der folgenden Klassen haben Sie schon verwendet?

- A** `java.util.List`
- B** `java.util.ArrayList`
- C** `java.util.LinkedList`
- D** `java.util.Vector`
- E** `java.util.HashMap`
- F** keine davon



→ sli.do/cs210

Java Generics & Collections

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

- ▶ bestehen aus einer Sammlung von Objekten

Java Generics & Collections

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

- ▶ bestehen aus einer Sammlung von Objekten
- ▶ Objekte für Container eine "black box" \rightsquigarrow egal was drin steckt

Java Generics & Collections

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

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

\rightsquigarrow möchten „sortenrein“ spezifizieren können, ohne vorher „Sorte“ festzulegen

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

Java Generics & Collections

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

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

\rightsquigarrow möchten „sortenrein“ spezifizieren können, ohne vorher „Sorte“ festzulegen

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

Typparameter! (in Java: *Generics*)

`List<T>`

`List<String> s = ..`

- ▶ 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

java.util.List
X

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

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

```
1 interface LinearList<Elem, Position extends LinearList.ListPosition> {
2     Position start();
3     Position end();
4     Position next(Position p);
5     Position previous(Position p);
6
7     void insert(Position p, Elem x);
8     void delete(Position p);
9     Elem get(Position p);
10    void set(Position p, Elem x);
11
12    /** Represents a position of an element in the list */
13    interface ListPosition { }
14
15    // default methods (siehe unten)
16 }
```

Lineare Listen

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

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

```
1 interface LinearList<Elem, Position extends LinearList.ListPosition> {
2     Position start();
3     Position end();
4     Position next(Position p);
5     Position previous(Position p);
6
7     void insert(Position p, Elem x);
8     void delete(Position p);
9     Elem get(Position p);
10    void set(Position p, Elem x);
11
12    /** Represents a position of an element in the list */
13    interface ListPosition { }
14
15    // default methods (siehe unten)
16 }
```

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

Lineare Listen – Positionen

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

- ▶ $L.start() := 1$ erstes Element

Lineare Listen – Positionen

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

- ▶ `L.start()` := 1 erstes Element
- ▶ `L.end()` := $n + 1$ *hinter* letztem Element

Lineare Listen – Positionen

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

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

(Positionen erscheinen unnötig? \rightsquigarrow siehe unten)

Lineare Listen – Operationen

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

$$\blacktriangleright L.get(p) := \begin{cases} a_p & \text{falls } p \neq L.end(), \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

Lineare Listen – Operationen

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

► $L.get(p) := \begin{cases} a_p & \text{falls } p \neq L.end(), \\ \text{undefiniert} & \text{sonst.} \end{cases}$

► $L.set(p, x)$ wirft Exception falls $p = L.end()$ und terminiert sonst normal.

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

Lineare Listen – Operationen

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

► $L.get(p) := \begin{cases} a_p & \text{falls } p \neq L.end(), \\ \text{undefiniert} & \text{sonst.} \end{cases}$

► $L.set(p, x)$ wirft Exception falls $p = L.end()$ und terminiert sonst normal.

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

► Nach $L.insert(p, x)$ gilt $L = a_1, \dots, a_{p-1}, \underline{x}, a_p, a_{p+1}, \dots, a_n$.

Beachte: $p = L.end()$ erlaubt! $\rightsquigarrow L.insert(p, x)$ hängt x ans Ende von L an.

Lineare Listen – Operationen

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

► $L.get(p) := \begin{cases} a_p & \text{falls } p \neq L.end(), \\ \text{undefiniert} & \text{sonst.} \end{cases}$

► $L.set(p, x)$ wirft Exception falls $p = L.end()$ und terminiert sonst normal.

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

► Nach $L.insert(p, x)$ gilt $L = a_1, \dots, a_{p-1}, x, a_p, a_{p+1}, \dots, a_n$.

Beachte: $p = L.end()$ erlaubt! $\rightsquigarrow L.insert(p, x)$ hängt x ans Ende von L an.

► $L.delete(p)$ wirft Exception falls $p = L.end()$ und terminiert sonst normal.

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

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?

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 $\text{insert}(L, p)$ / $\text{delete}(L, p)$ mit der Position p ?

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?

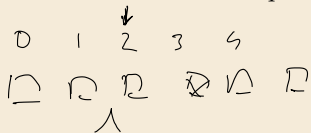
Jein ... wir haben nämlich eine Subtilität unterschlagen:

Was passiert bei $\text{insert}(L, p)$ / $\text{delete}(L, p)$ mit der Position p ?

Antwort: Implementierungen von Linearen Listen können Realisierungen wählen!

Index-basierte Positionen

- ▶ entspricht obigem Modell einer mathematischen Sequenz



Pointer-basierte Positionen

- ▶ entspricht einer verketteten Liste (Index vielleicht gar nicht bekannt)



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 $\text{insert}(L, p)$ / $\text{delete}(L, p)$ 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

Pointer-basierte Positionen

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

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 $\text{insert}(L, p)$ / $\text{delete}(L, p)$ 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 $\text{get}(p)$!

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 $\text{get}(p)$ **nicht**

Clicker Question

Listen im Collections Framework der Java Runtime Library realisieren unsere Positionen über Iterator-Objekte. Welcher Art von Positionen entsprechen Iteratoren?



- A** Index-basierte Positionen
- B** Pointer-basierte Positionen
- C** Beides
- D** Weder noch
- E** Iter..was?
- F** keine Ahnung



→ sli.do/cs210

Clicker Question

Listen im Collections Framework der Java Runtime Library realisieren unsere Positionen über Iterator-Objekte. Welcher Art von Positionen entsprechen Iteratoren?



- A** Index-basierte Positionen ✓
- B** Pointer-basierte Positionen ✓
- C** Beides ✓
- D** Weder noch ✓
- E** ~~Iter..was?~~
- F** ~~keine Ahnung~~



→ sli.do/cs210

Abgeleitete Operationen

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

Ultimativer wiederverwendbarer Code!

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); }
```

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 }
```

if then else Ausdruck

?

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
```

`for (String x : l)`

`l = List<String>`

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

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



LIFO

Last in First out

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

```
1 public interface Stack<Elem> extends Iterable<Elem> {  
2     void push(Elem x);  
3     Elem pop();  
4     Elem top();  
5     boolean empty();  
6 }
```

Queues

FIFO

ADT *Queue* (Warteschlange):

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



```
1 public interface Queue<Elem> extends Iterable<Elem> {  
2     void enqueue(Elem x);  
3     Elem dequeue();  
4     Elem peek();  
5     boolean empty();  
6 }
```

Bags

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

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()`

Stack

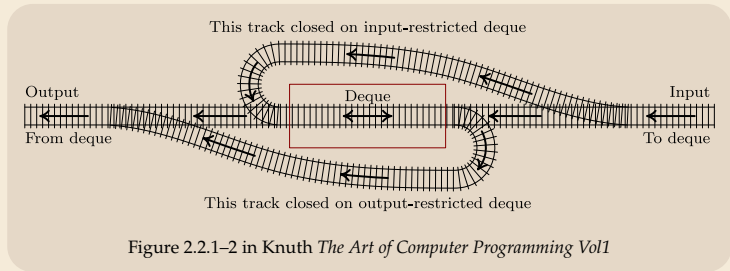
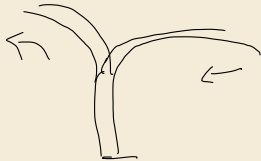


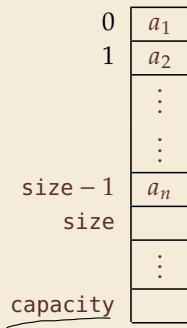
Figure 2.2.1-2 in Knuth *The Art of Computer Programming Vol1*

8.4 Array-basierte Listen

Array-basierte Listen

Idee: Speichere Elemente der Linearen Liste in einem Array.

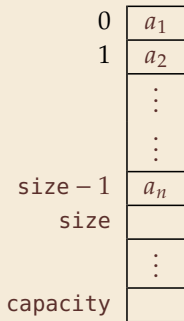
- ▶ Array-basierte Lineare Listen bleiben am nächsten an mathematischer Sequenz
- ▶ Array hat feste Größe \rightsquigarrow im Konstruktor fest gewählt.



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 \rightsquigarrow im Konstruktor fest gewählt.

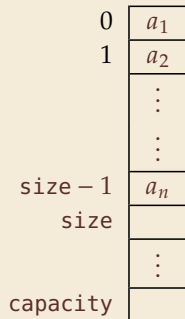


```
1 public class FixedSizeArrayList<Elem>
2     implements LinearList<Elem, FixedSizeArrayList.Position> {
3     private final int capacity;
4     private final Object[] elements;
5     private int size = 0;
6     /** Konstruktor */
7     public FixedSizeArrayList(int capacity) {
8         this.capacity = 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

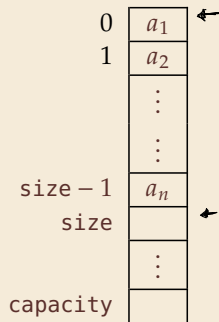
Idee: Speichere Elemente der Linearen Liste in einem Array.

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



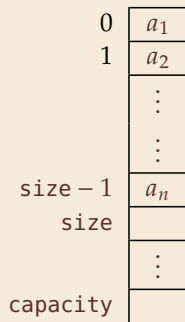
```
1 public class FixedSizeArrayList<Elem>
2     implements LinearList<Elem, FixedSizeArrayList.Position> {
3     private final int capacity;
4     private final Object[] elements;
5     private int size = 0;
6     /** Konstruktor */
7     public FixedSizeArrayList(int capacity) {
8         this.capacity = 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



```
1 public class FixedSizeArrayList<Elem> { // continued
2     public Position start() { return new Position(0); }
3     public Position end() { return new Position(size); }
4     public Position next(Position p) {
5         if (p.index == size) throw new IndexOutOfBoundsException();
6         return new Position(p.index+1);
7     }
8     public Position previous(Position p) {
9         if (p.index == 0) throw new IndexOutOfBoundsException();
10        return new Position(p.index-1);
11    }
12    @SuppressWarnings("unchecked")
13    public Elem get(Position p) { return (Elem) elements[p.index]; }
14    public void set(Position p, Elem x) { elements[p.index] = x; }
15    // more methods
16 }
```

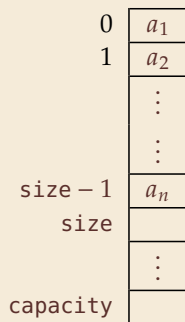
Array-basierte Listen – Günstige Operationen



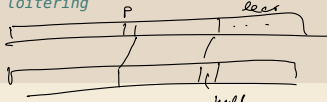
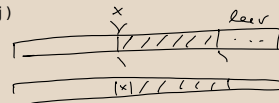
```
1 public class FixedSizeArrayList<Elem> { // continued
2     public Position start() { return new Position(0); }
3     public Position end() { return new Position(size); }
4     public Position next(Position p) {
5         if (p.index == size) throw new IndexOutOfBoundsException();
6         return new Position(p.index+1);
7     }
8     public Position previous(Position p) {
9         if (p.index == 0) throw new IndexOutOfBoundsException();
10        return new Position(p.index-1);
11    }
12    @SuppressWarnings("unchecked")
13    public Elem get(Position p) { return (Elem) elements[p.index]; }
14    public void set(Position p, Elem x) { elements[p.index] = x; }
15    // more methods
16 }
```

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

Array-basierte Listen – Teure Operationen



```
1 public class FixedSizeArrayList<Elem> { // continued
2     public void insert(Position p, Elem x) {
3         if (size >= capacity) throw new CapacityExceededException();
4         for (int j = size - 1; j >= p.index; --j)
5             elements[j + 1] = elements[j];
6         elements[p.index] = x; ++size;
7     }
8     public void delete(Position p) {
9         if (p.equals(end())) throw new IndexOutOfBoundsException();
10        for (int j = p.index; j <= size-2; ++j)
11            elements[j] = elements[j+1];
12        elements[size-1] = null; // avoid loitering
13        --size;
14    }
15 }
```

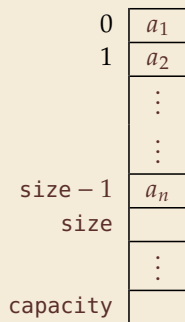


- ▶ können zwar direkt auf Position p zugreifen, müssen aber alle Elemente rechts von p um eine Position verschieben

↪ Worst-Case $O(n)$ Laufzeit

loitering: Speicher blockiert
durch eigentlich nicht
mehr verwendete Referenz

Array-basierte Listen – Teure Operationen



```
1 public class FixedSizeArrayList<Elem> { // continued
2     public void insert(Position p, Elem x) {
3         if (size >= capacity) throw new CapacityExceededException();
4         for (int j = size - 1; j >= p.index; --j)
5             elements[j + 1] = elements[j];
6         elements[p.index] = x; ++size;
7     }
8     public void delete(Position p) {
9         if (p.equals(end())) throw new IndexOutOfBoundsException();
10        for (int j = p.index; j <= size-2; ++j)
11            elements[j] = elements[j+1];
12        elements[size-1] = null; // avoid loitering
13        --size;
14    }
15 }
```

- ▶ 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

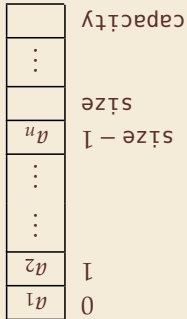
Array-Basierter Stack

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

Array-Basierter Stack

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

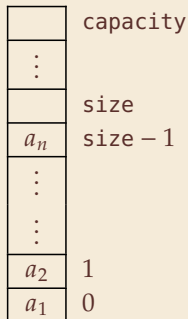
Müssen dafür nur konzeptionell das Array auf den Kopf stellen:



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:



```
1 public class ArrayStack<Elem> implements Stack<Elem> {
2     private final Object[] contents;
3     private int size = 0;
4
5     public ArrayStack(int capacity) { contents = new Object[capacity]; }
6
7     public void push(Elem x) {
8         if (size == contents.length) throw new Exception();
9         contents[size++] = x;
10    }
11    public Elem pop() {
12        Elem result = top();
13        contents[--size] = null; // avoid loitering
14        return result;
15    }
16    @SuppressWarnings("unchecked")
17    public Elem top() { return (Elem) contents[size-1]; }
18 }
```

if (size == capacity) throw new CapacityExceededException();

$O(1)$ für alle Operationen

Ringpuffer

Für Queues haben wir Updates an beiden Rändern . . . Kopieren nötig?

Nein! Wir müssen nur die Repräsentation flexibel machen.

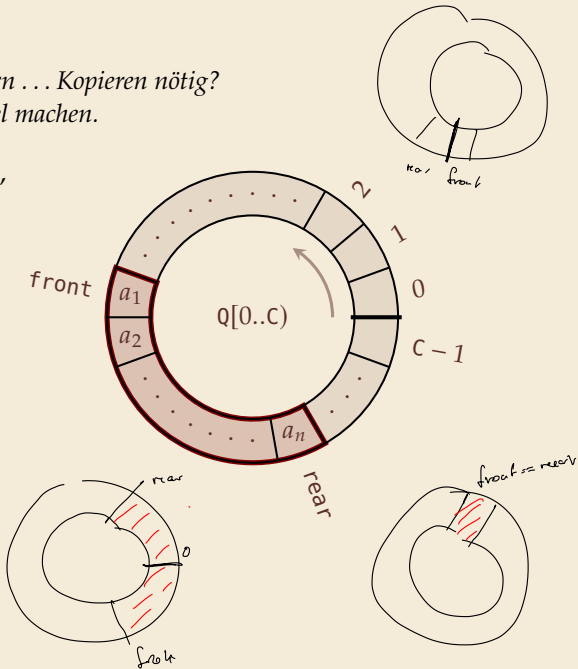
- ▶ enqueue hängt neues Element hinten an,
dequeue löscht vorne

↔ Belegter Bereich verschiebt sich durch
das Array nach hinten

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



Clicker Question



Was gilt für front und rear, wenn die Queue leer ist?



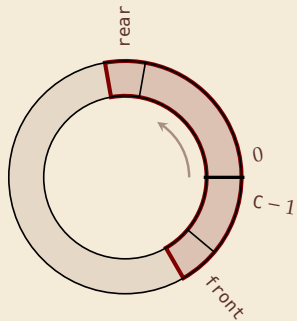
→ sli.do/cs210

Ringpuffer Randfälle

Invariante: i -tes Element in $Q[(\text{front} + i - 1) \bmod C]$ ($i = 0, \dots, n - 1$)

Ringpuffer Randfälle

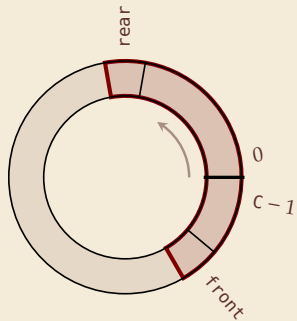
Invariante: i -tes Element in $Q[(\text{front} + i - 1) \bmod C]$ ($i = 0, \dots, n - 1$)



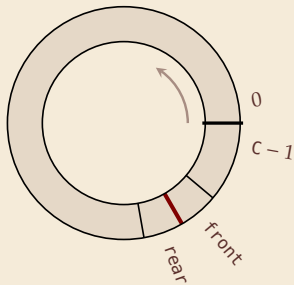
Overflow ✓

Ringpuffer Randfälle

Invariante: i -tes Element in $Q[(\text{front} + i - 1) \bmod C]$ ($i = 0, \dots, n - 1$)



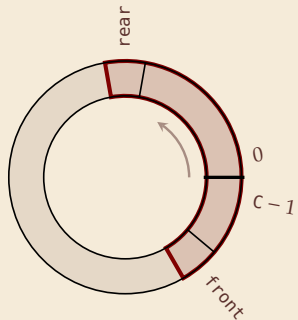
Overflow ✓



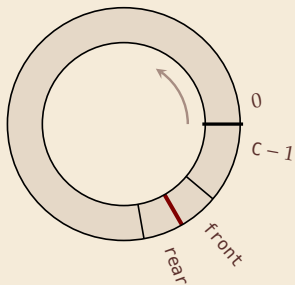
Leere Queue

Ringpuffer Randfälle

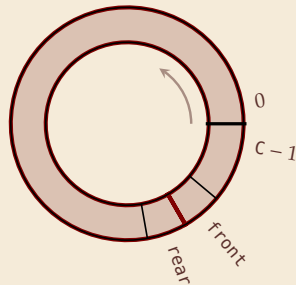
Invariante: i -tes Element in $Q[(\text{front} + i - 1) \bmod C]$ ($i = 0, \dots, n - 1$)



Overflow ✓



Leere Queue

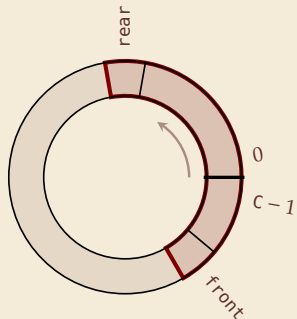


Volle = Leere Queue?

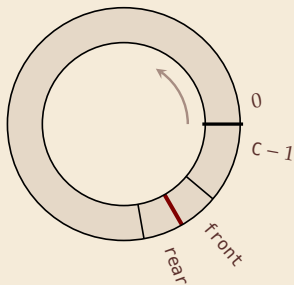
Ringpuffer Randfälle

Invariante: i -tes Element in $Q[(\text{front} + i - 1) \bmod C]$ ($i = 0, \dots, n - 1$)

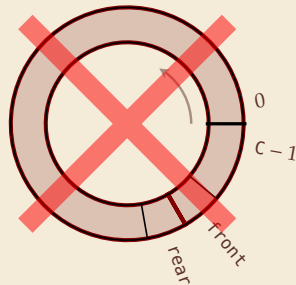
Erlaubte Anzahl Elemente ist $n \in [0..C - 1]$



Overflow ✓



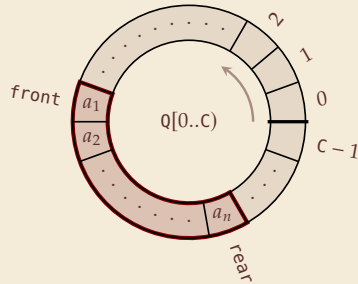
Leere Queue



Volle = Leere Queue?

Ringpuffer Implementierung

```
1 public class ArrayQueue<Elem> implements Queue<Elem> {
2     private final Object[] Q;
3     private int front, rear, C;
4     public ArrayQueue(final int capacity) {
5         C = capacity + 1; // allow capacity elems
6         Q = new Object[C];
7         front = 0; rear = Q.length-1;
8     }
9     public void enqueue(final Elem x) {
10        if (full()) throw new CapacityExceededException();
11        rear = (rear + 1) % C; Q[rear] = x;
12    }
13    public Elem dequeue() {
14        Elem res = peek();
15        Q[front] = null; // prevent loitering
16        front = (front + 1) % C;
17        return res;
18    }
19    @SuppressWarnings("unchecked")
20    public Elem peek() {
21        if (empty()) throw new NoSuchElementException();
22        return (Elem) Q[front];
23    }
24    public boolean empty() { return (rear + 1) % C == front; }
25    public boolean full() { return (rear + 2) % C == front; }
26 }
```



- ▶ alle Operationen $O(1)$ Zeit
- ▶ wieder: beschränkte Kapazität
 \rightsquigarrow keine allgemeine Queue
- ▶ Anwendung in real-time streams
 - ▶ entferne full-check
 - \rightsquigarrow 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

Clicker Question



Haben Sie schon einmal eine verkettete Liste (linked lists) implementiert?

- A** Ja
- B** Nein
- C** Erinnere mich nicht
- D** Verkettete Was?



→ sli.do/cs210

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

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 ...

Jetzt: Jedes Element verweist auf Nachfolger

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

Adresse	Inhalt
A	a_1 B
B	a_2 C
C	a_3 D
D	a_4 E
E	a_5 \perp $\hat{=}$ null

:

C	a_3 D
---	-----------

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 ...

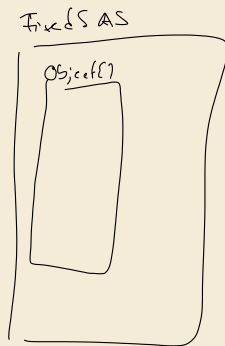
Jetzt: Jedes Element verweist auf Nachfolger

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

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

↪ Jede Zeile durch ein Objekt repräsentiert

► Adresse ist Objektreferenz / Pointer



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 ...

Jetzt: Jedes Element verweist auf Nachfolger

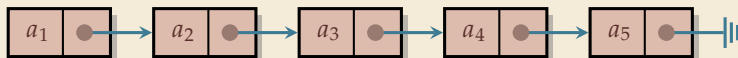
Adresse	Inhalt
1	a_1
2	a_2
3	a_3
4	a_4
5	a_5

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

↪ Jede Zeile durch ein Objekt repräsentiert

▶ Adresse ist Objektreferenz / Pointer

▶ graphisch:

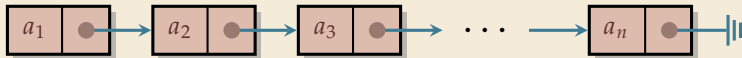


Einfach verkettete Listen

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

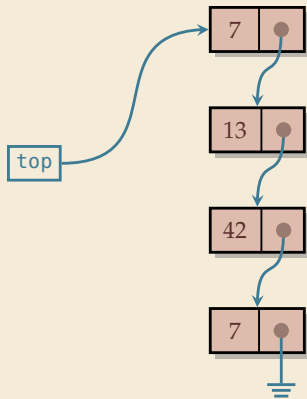
```
1 public class LinkedList<Elem>
2     implements LinearList<Elem, LinkedList.Position> {
3     // ...
4     private static final class ListNode extends Position {
5         Object element;
6         ListNode next;
7     public ListNode() { }
8     public ListNode(Object element, ListNode next) {
9         this.element = element; this.next = next;
10    }
11    }
12    public static abstract class Position
13        implements LinearList.ListPosition {
14        private Position() { } // keine Instanzen außerhalb
15    }
16 }
```

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



Verkettete Stacks

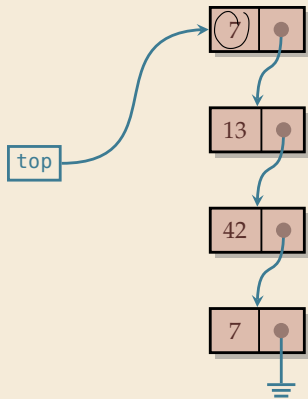
Funktioniert besonders bequem für *Stacks*!



```
1 public class LinkedStack<Elem> implements Stack<Elem> {
2     ListNode top = null;
3     private static final class ListNode {
4         Object element; ListNode next;
5         public ListNode(Object element) {
6             this.element = element;
7         }
8     }
9     @SuppressWarnings("unchecked")
10    public Elem top() {
11        if (empty()) throw new Exception();
12        return (Elem) top.element;
13    }
14    public void push(Elem x) {
15        ListNode newTop = new ListNode(x);
16        newTop.next = this.top;
17        this.top = newTop;
18    }
19    public Elem pop() {
20        Elem result = top();
21        top = top.next;
22        return result;
23    }
24    public boolean empty() { return top == null; }
25 }
```

Verkettete Stacks

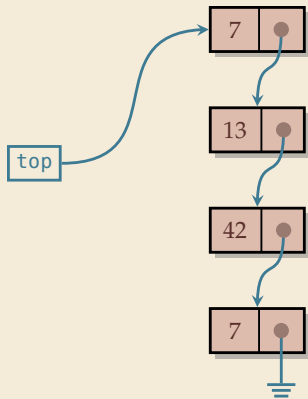
Funktioniert besonders bequem für *Stacks*!



```
1 public class LinkedStack<Elem> implements Stack<Elem> {  
2     ListNode top = null;  
3     private static final class ListNode {  
4         Object element; ListNode next;  
5         public ListNode(Object element) {  
6             this.element = element;  
7         }  
8     }  
9     @SuppressWarnings("unchecked")  
10    public Elem top() {  
11        if (empty()) throw new Exception();  
12        return (Elem) top.element;  
13    }  
14    public void push(Elem x) {  
15        ListNode newTop = new ListNode(x);  
16        newTop.next = this.top;  
17        this.top = newTop;  
18    }  
19    public Elem pop() {  
20        Elem result = top();  
21        top = top.next;  
22        return result;  
23    }  
24    public boolean empty() { return top == null; }  
25 }
```

Verkettete Stacks

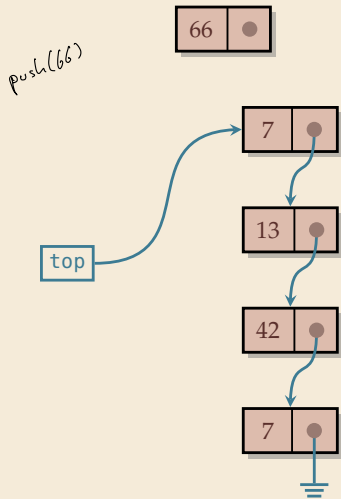
Funktioniert besonders bequem für *Stacks*!



```
1 public class LinkedStack<Elem> implements Stack<Elem> {  
2     ListNode top = null;  
3     private static final class ListNode {  
4         Object element; ListNode next;  
5         public ListNode(Object element) {  
6             this.element = element;  
7         }  
8     }  
9     @SuppressWarnings("unchecked")  
10    public Elem top() {  
11        if (empty()) throw new Exception();  
12        return (Elem) top.element;  
13    }  
14    public void push(Elem x) {  
15        ListNode newTop = new ListNode(x);  
16        newTop.next = this.top;  
17        this.top = newTop;  
18    }  
19    public Elem pop() {  
20        Elem result = top();  
21        top = top.next;  
22        return result;  
23    }  
24    public boolean empty() { return top == null; }  
25 }
```

Verkettete Stacks

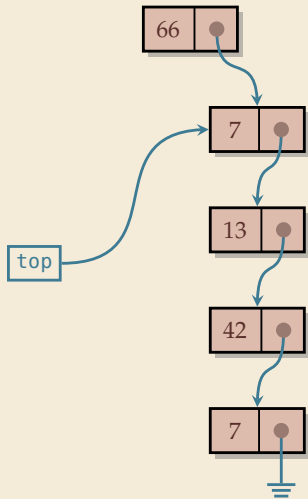
Funktioniert besonders bequem für *Stacks*!



```
1 public class LinkedStack<Elem> implements Stack<Elem> {
2     ListNode top = null;
3     private static final class ListNode {
4         Object element; ListNode next;
5         public ListNode(Object element) {
6             this.element = element;
7         }
8     }
9     @SuppressWarnings("unchecked")
10    public Elem top() {
11        if (empty()) throw new Exception();
12        return (Elem) top.element;
13    }
14    public void push(Elem x) {
15        ListNode newTop = new ListNode(x);
16        newTop.next = this.top;
17        this.top = newTop;
18    }
19    public Elem pop() {
20        Elem result = top();
21        top = top.next;
22        return result;
23    }
24    public boolean empty() { return top == null; }
25 }
```

Verkettete Stacks

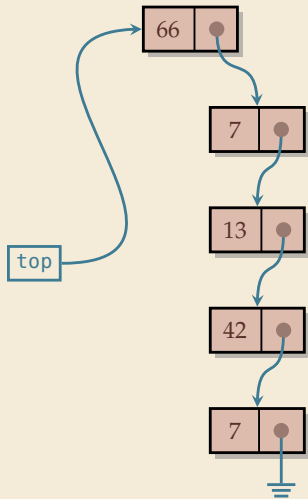
Funktioniert besonders bequem für *Stacks*!



```
1 public class LinkedStack<Elem> implements Stack<Elem> {  
2     ListNode top = null;  
3     private static final class ListNode {  
4         Object element; ListNode next;  
5         public ListNode(Object element) {  
6             this.element = element;  
7         }  
8     }  
9     @SuppressWarnings("unchecked")  
10    public Elem top() {  
11        if (empty()) throw new Exception();  
12        return (Elem) top.element;  
13    }  
14    public void push(Elem x) {  
15        ListNode newTop = new ListNode(x);  
16        newTop.next = this.top;  
17        this.top = newTop;  
18    }  
19    public Elem pop() {  
20        Elem result = top();  
21        top = top.next;  
22        return result;  
23    }  
24    public boolean empty() { return top == null; }  
25 }
```

Verkettete Stacks

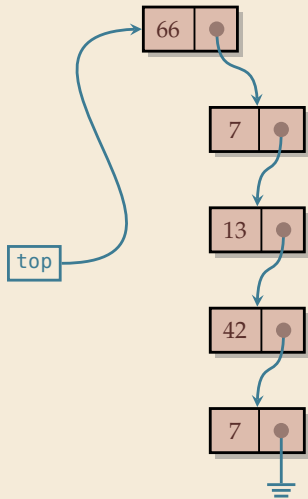
Funktioniert besonders bequem für *Stacks*!



```
1 public class LinkedStack<Elem> implements Stack<Elem> {  
2     ListNode top = null;  
3     private static final class ListNode {  
4         Object element; ListNode next;  
5         public ListNode(Object element) {  
6             this.element = element;  
7         }  
8     }  
9     @SuppressWarnings("unchecked")  
10    public Elem top() {  
11        if (empty()) throw new Exception();  
12        return (Elem) top.element;  
13    }  
14    public void push(Elem x) {  
15        ListNode newTop = new ListNode(x);  
16        newTop.next = this.top;  
17        this.top = newTop;  
18    }  
19    public Elem pop() {  
20        Elem result = top();  
21        top = top.next;  
22        return result;  
23    }  
24    public boolean empty() { return top == null; }  
25 }
```

Verkettete Stacks

Funktioniert besonders bequem für *Stacks*!



```
1 public class LinkedStack<Elem> implements Stack<Elem> {  
2     ListNode top = null;  
3     private static final class ListNode {  
4         Object element; ListNode next;  
5         public ListNode(Object element) {  
6             this.element = element;  
7         }  
8     }  
9     @SuppressWarnings("unchecked")  
10    public Elem top() {  
11        if (empty()) throw new Exception();  
12        return (Elem) top.element;  
13    }  
14    public void push(Elem x) {  
15        ListNode newTop = new ListNode(x);  
16        newTop.next = this.top;  
17        this.top = newTop;  
18    }  
19    public Elem pop() {  
20        Elem result = top();  
21        top = top.next;  
22        return result;  
23    }  
24    public boolean empty() { return top == null; }  
25 }
```

↪ keine inhärente Begrenzung für Stackgröße mehr!

Verkettete Queue

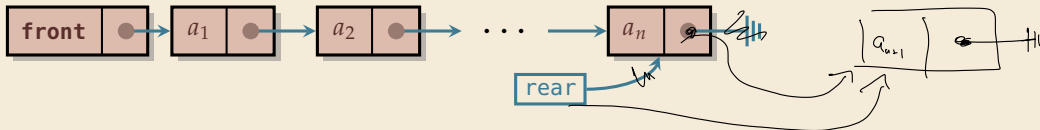
Queues sind fast genauso einfach.

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

↪ brauchen keine Fallunterscheidung für leere Queue in enqueue/dequeue 🤖

```
1 public class LinkedQueue<Elem>
2     implements Queue<Elem> {
3
4     ListNode front = new ListNode(null),
5         rear = front;
6     private static final class ListNode {
7         Object element; ListNode next;
8         public ListNode(Object e) { element = e; }
9     }
10    public void enqueue(Elem x) {
11        ListNode newRear = new ListNode(x);
12        rear.next = newRear;
13        rear = newRear;
14    }
```

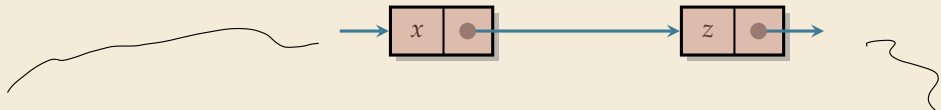
```
15    public Elem dequeue() {
16        Elem result = peek();
17        front = front.next;
18        return result;
19    }
20    @SuppressWarnings("unchecked")
21    public Elem peek() {
22        if (empty()) throw new Exception();
23        return (Elem) front.next.element;
24    }
25    public boolean empty() {
26        return front.next == null;
27    }
28 }
```



Updates in der Mitte

Updates mitten in einer Liste sind ebenso einfach und effizient!

Insert: Füge y zwischen x und z ein.

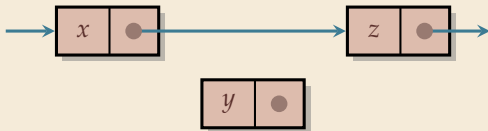


Updates in der Mitte

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.

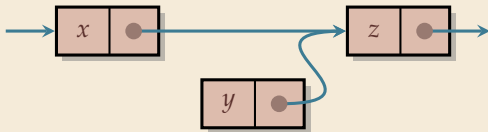


Updates in der Mitte

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 $y.next := z$

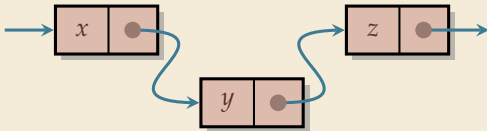


Updates in der Mitte

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 $x.next := y$

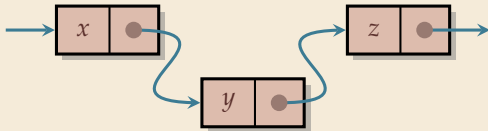


Updates in der Mitte

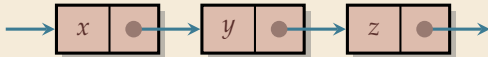
Updates mitten in einer Liste sind ebenso einfach und effizient!

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

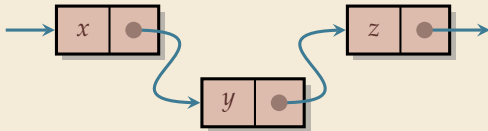


Updates in der Mitte

Updates mitten in einer Liste sind ebenso einfach und effizient!

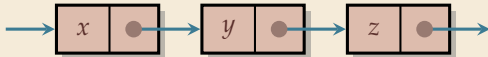
Insert: Füge y zwischen x und z ein.

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

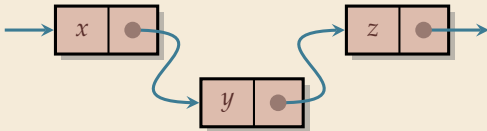


Updates in der Mitte

Updates mitten in einer Liste sind ebenso einfach und effizient!

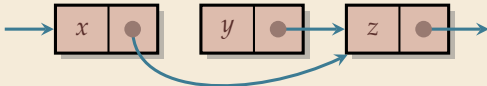
Insert: Füge y zwischen x und z ein.

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 $x.next := y$

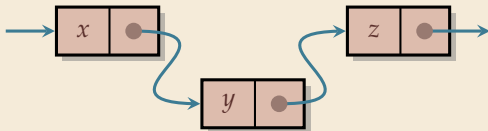


Updates in der Mitte

Updates mitten in einer Liste sind ebenso einfach und effizient!

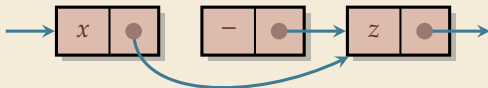
Insert: Füge y zwischen x und z ein.

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 $x.next := z$
3. Gebe y frei

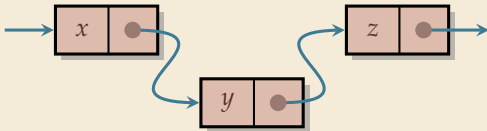


Updates in der Mitte

Updates mitten in einer Liste sind ebenso einfach und effizient!

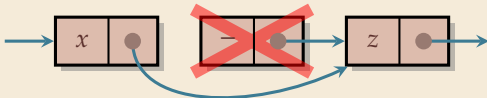
Insert: Füge y zwischen x und z ein.

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 $x.next := z$
3. Gebe y frei

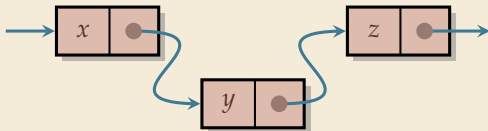


Updates in der Mitte

Updates mitten in einer Liste sind ebenso einfach und effizient!

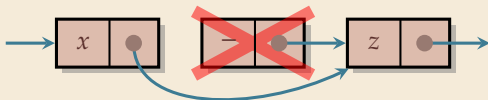
Insert: Füge y zwischen x und z ein.

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 $x.next := z$
3. Gebe y frei



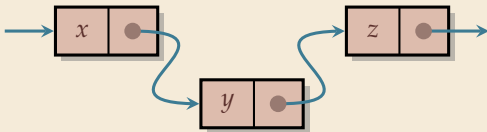
⚠ Brauchen Zugriff auf Vorgänger zum Löschen

Updates in der Mitte

Updates mitten in einer Liste sind ebenso einfach und effizient!

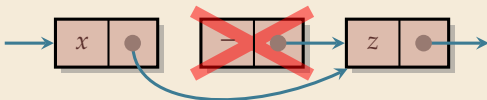
Insert: Füge y zwischen x und z ein.

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



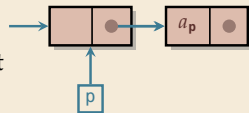
Delete: Lösche y aus Liste

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



⚠ Brauchen Zugriff auf Vorgänger zum Löschen

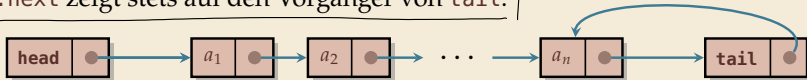
↪ **Konvention:** Position zeigt auf Knoten **vor** aktuellem Element



Einfach verkettete linear Liste

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

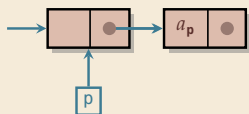
- ▶ Eine lineare Liste enthält immer die beiden Dummy-Knoten `head` und `tail`
- ▶ `tail.next` zeigt stets auf den Vorgänger von `tail`.



```
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.

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.

Vereinfachung:

alle Speicherblöcke

haben feste Größe

DIY Memory Management

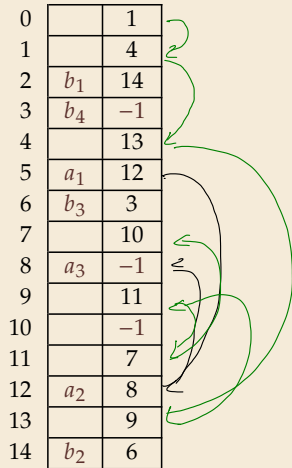
In Java findet JVM für `new ListNode()` (normalerweise) ein freies Stück Speicher auf dem Heap.

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.

Geteilter Speicher

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



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.

```
1 public class DIYMemoryManagement {
2     private MemoryCell[] mem;
3     public static final int NULL = -1;
4     // ...
5     public MemoryCell get(int address) {
6         return mem[address];
7     }
8     public static class MemoryCell {
9         Object element; int next;
10        public MemoryCell(Object element, int next) {
11            this.element = element;
12            this.next = next;
13        }
14    }
15 }
```

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

```
1 public class DIYMemoryManagement { // continued
2     private int nextFree = 0;
3
4     public DIYMemoryManagement(int memorySize) {
5         this.mem = new MemoryCell[memorySize];
6         for (int i = 0; i < mem.length; i++)
7             mem[i] = new MemoryCell(null,i+1);
8         mem[memorySize-1].next = NULL;
9     }
10    public int alloc() {
11        if (nextFree == NULL) throw new OutOfMemoryError();
12        int result = nextFree;
13        this.nextFree = mem[result].next;
14        return result;
15    }
16    public void free(int address) {
17        mem[address].element = null;
18        mem[address].next = nextFree;
19        nextFree = address;
20    }
21 }
```

8.6 Doppelt Verkettete Listen

Doppelte Verkettung

Einfach verkettete Listen können nicht effizient den Vorgänger finden.

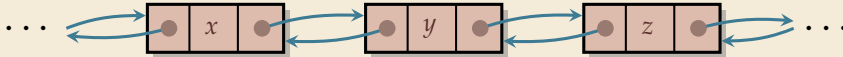
Unter Verdopplung der Anzahl Pointer können wir auch das unterstützen.



Doppelte Verkettung

Einfach verkettete Listen können nicht effizient den Vorgänger finden.

Unter Verdopplung der Anzahl Pointer können wir auch das unterstützen.

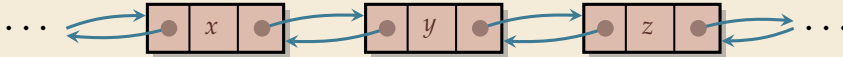


```
1 public class DoublyLinkedList<Elem> implements  
2     LinearList<Elem, DoublyLinkedList.Position> {  
3     private static final class ListNode extends Position {  
4         Object element; ListNode next, previous;  
5         ListNode(Object element) { this.element = element; }  
6     }
```

Doppelte Verkettung

Einfach verkettete Listen können nicht effizient den Vorgänger finden.

Unter Verdopplung der Anzahl Pointer können wir auch das unterstützen.

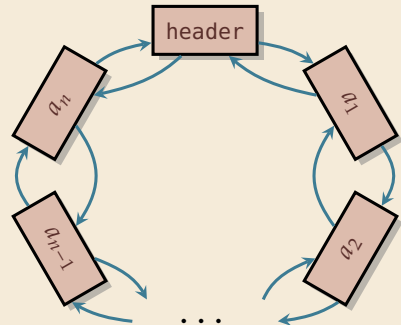


```
1 public class DoublyLinkedList<Elem> implements  
2   LinearList<Elem, DoublyLinkedList.Position> {  
3   private static final class ListNode extends Position {  
4     Object element; ListNode next, previous;  
5     ListNode(Object element) { this.element = element; }  
6   }  
}
```

Geschickte Konvention für Randfälle:

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

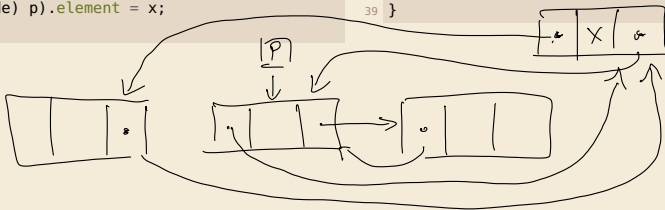
```
7 private final ListNode header;  
8 public DoublyLinkedList() {  
9   header = new ListNode(null);  
10  header.next = header; header.previous = header;  
11 }  
12 // ...  
13 }
```



Doppelt verkettete Listen

```
1 public class DoublyLinkedList<Elem> { // continued
2     public Position start() { return header.next; }
3     public Position end() { return header; }
4     public Position next(Position p) {
5         if (p.equals(end())) throw new Exception();
6         return ((ListNode) p).next;
7     }
8     public Position previous(Position p) {
9         if (p.equals(start())) throw new Exception();
10        return ((ListNode) p).previous;
11    }
12    @SuppressWarnings("unchecked")
13    public Elem get(Position p) {
14        if (p.equals(end())) throw new Exception();
15        return (Elem) ((ListNode) p).element;
16    }
17    public void set(Position p, Elem x) {
18        if (p.equals(end())) throw new Exception();
19        ((ListNode) p).element = x;
20    }
```


```
21
22     /** direkt vor p einfügen */
23     public void insert(Position p, Elem x) {
24         ListNode newNode = new ListNode(x);
25         final ListNode node = ((ListNode) p);
26         newNode.previous = node.previous;
27         newNode.next = node;
28         node.previous.next = newNode;
29         node.previous = newNode;
30     }
31
32     public void delete(Position p) {
33         if (p.equals(end())) throw new Exception();
34         final ListNode node = ((ListNode) p);
35         node.previous.next = node.next;
36         node.next.previous = node.previous;
37     }
38
39 }
```



Doppelt verkettete Listen

```
1 public class DoublyLinkedList<Elem> { // continued
2     public Position start() { return header.next; }
3     public Position end() { return header; }
4     public Position next(Position p) {
5         if (p.equals(end())) throw new Exception();
6         return ((ListNode) p).next;
7     }
8     public Position previous(Position p) {
9         if (p.equals(start())) throw new Exception();
10        return ((ListNode) p).previous;
11    }
12    @SuppressWarnings("unchecked")
13    public Elem get(Position p) {
14        if (p.equals(end())) throw new Exception();
15        return (Elem) ((ListNode) p).element;
16    }
17    public void set(Position p, Elem x) {
18        if (p.equals(end())) throw new Exception();
19        ((ListNode) p).element = x;
20    }
```

```
21
22     /** direkt vor p einfügen */
23     public void insert(Position p, Elem x) {
24         ListNode newNode = new ListNode(x);
25         final ListNode node = ((ListNode) p);
26         newNode.previous = node.previous;
27         newNode.next = node;
28         node.previous.next = newNode;
29         node.previous = newNode;
30     }
31
32     public void delete(Position p) {
33         if (p.equals(end())) throw new Exception();
34         final ListNode node = ((ListNode) p);
35         node.previous.next = node.next;
36         node.next.previous = node.previous;
37     }
38
39 }
```

- ▶ 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

Dünn besetzte Matrizen

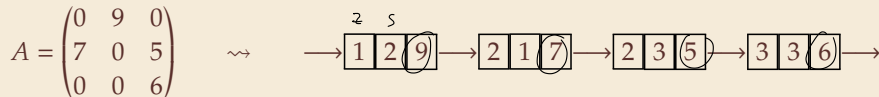
Matrizen und Vektoren enthalten in gewissen Anwendungen viele 0-Einträge.

- ▶ 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:



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:

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

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

Dünn besetzte Matrizen – Repräsentation

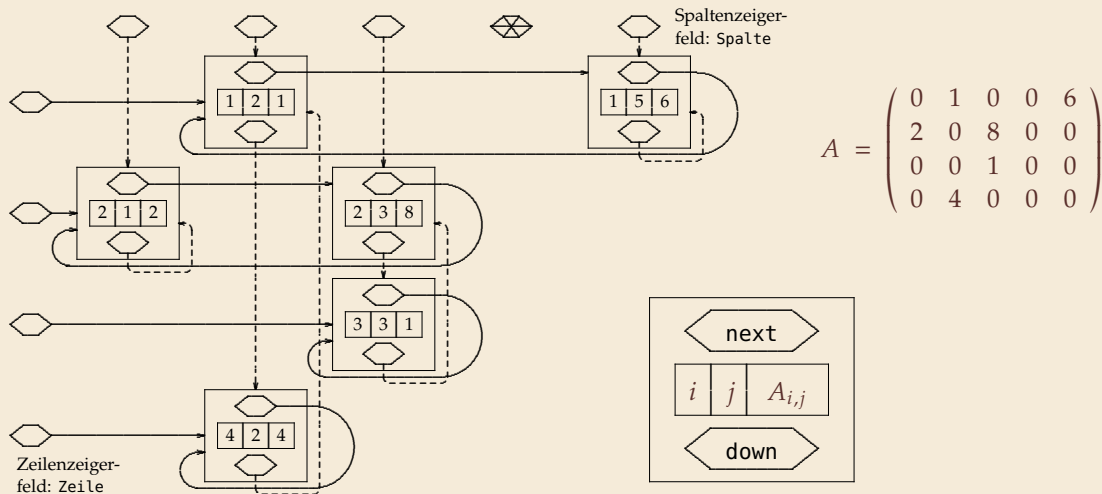


Abb. 2.17, Nebel & Wild, *Entwurf und Analyse von Algorithmen*, 2018

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
 - ▶ Allokieren 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 passt gut.*


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 passt 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?*

Clicker Question



Angenommen, wir haben aktuell ein Array der Größe n , welches nun vollgelaufen ist. Welche Größe würden Sie für das neue Array verwenden?



→ sli.do/cs210

Doubling Arrays

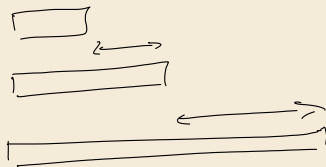
Lösung:

- ▶ Wir halten Array $A[0..C)$ mit Kapazität C vor
 - ▶ Speichern aktuell n Elemente
 - ▶ $\text{push}(x)$ fügt x am Ende an (in $A[n]$)
 - ▶ $\text{pop}()$ entfernt das letzte Element $A[n - 1]$
 - ▶ **Invariante:** $\frac{1}{4}C \leq n \leq C$
- ↪ Haben stets Platz für alle Elemente!

Doubling Arrays

Lösung:

- ▶ Wir halten Array $A[0..C)$ mit Kapazität C vor
 - ▶ Speichern aktuell n Elemente
 - ▶ $\text{push}(x)$ fügt x am Ende an (in $A[n]$)
 - ▶ $\text{pop}()$ entfernt das letzte Element $A[n-1]$
 - ▶ **Invariante:** $\frac{1}{4}C \leq n \leq C$
- ↔ Haben stets Platz für alle Elemente!



Wie erhalten wir die Invariante aufrecht?

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

Doubling Arrays

```
1 public class DoublingArray<Elem> {
2     private Object[] A = new Object[8]; // C=8
3     private int n = 0;
4     /** Stelle Invariante sicher, evtl. Kopie */
5     private void ensureCapacity() {
6         int C = A.length;
7         if (C/4 < n && n < C || n <= 4) return;
8         Object[] newA = new Object[2*n];
9         System.arraycopy(A, 0, newA, 0, n); // O(n)
10        A = newA;
11    }
12    public Integer start() { return 0; }
13    public Integer end() { return n; }
14    public Integer next(Integer p) {
15        if (p == n) throw new Exception();
16        return p + 1;
17    }
18    public Integer previous(Integer p) {
19        if (p == 0) throw new Exception();
20        return p - 1;
21    }
}
```

```
22
23     public void push(Elem x) {
24         ensureCapacity();
25         A[n++] = x;
26     }
27     public void pop() {
28         ensureCapacity();
29         A[--n] = null;
30     }
31
32     @SuppressWarnings("unchecked")
33     public Elem get(Integer p) {
34         if (p < 0 || p >= n) throw new Exception();
35         return (Elem) A[p];
36     }
37     public void set(final Integer p, final Elem x) {
38         if (p < 0 || p >= n) throw new Exception();
39         A[p] = x;
40     }
41 }
```

Doubling Arrays

```
1 public class DoublingArray<Elem> {
2     private Object[] A = new Object[8]; // C=8
3     private int n = 0;
4     /** Stelle Invariante sicher, evtl. Kopie */
5     private void ensureCapacity() {
6         int C = A.length;
7         if (C/4 < n && n < C || n <= 4) return;
8         Object[] newA = new Object[2*n];
9         System.arraycopy(A, 0, newA, 0, n); // O(n)
10        A = newA;
11    }
12    public Integer start() { return 0; }
13    public Integer end() { return n; }
14    public Integer next(Integer p) {
15        if (p == n) throw new Exception();
16        return p + 1;
17    }
18    public Integer previous(Integer p) {
19        if (p == 0) throw new Exception();
20        return p - 1;
21    }
}
```

```
22
23     public void push(Elem x) {
24         ensureCapacity();
25         A[n++] = x;
26     }
27     public void pop() {
28         ensureCapacity();
29         A[--n] = null;
30     }
31
32     @SuppressWarnings("unchecked")
33     public Elem get(Integer p) {
34         if (p < 0 || p >= n) throw new Exception();
35         return (Elem) A[p];
36     }
37     public void set(final Integer p, final Elem x) {
38         if (p < 0 || p >= n) throw new Exception();
39         A[p] = x;
40     }
41 }
```

worst-case : push | $\Theta(n)$
pop |

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)

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*

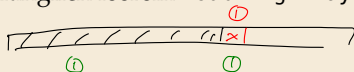
Betrachten hier anschauliche Version **ohne pop() Aufrufe**

Arrays und Arbeitszeitkonten

Theorem 8.2 (Amortisierte Kosten von dynamischen Arrays)

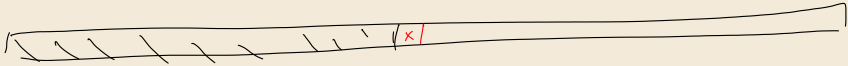
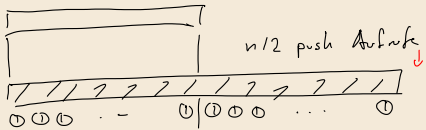
Eine beliebige Folge von m push-Operationen auf anfänglich leerem DoublingArray hat Gesamtkosten von $O(m)$ Arrayzugriffen.

$$\in 3m$$



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



Clicker Question



Welche der folgenden Aussagen über ein dynamisches Array mit aktuell n Elementen sind korrekt?

- A** Die Elemente sind in einem Array der Länge $2n$ gespeichert.
- B** Einfügen oder Löschen am Ende der Liste kostet $O(1)$ Zeit.
- C** Eine Folge von m Einfügungen am Ende der Liste kostet $O(n + m)$ Gesamtzeit.
- D** Einfügen in der Liste hat amortisierte Kosten von $O(1)$.



→ sli.do/cs210

Clicker Question



Welche der folgenden Aussagen über ein dynamisches Array mit aktuell n Elementen sind korrekt?

- A** ~~Die Elemente sind in einem Array der Länge $2n$ gespeichert.~~
- B** ~~Einfügen oder Löschen am Ende der Liste kostet $O(1)$ Zeit.~~
- C** Eine Folge von m Einfügungen am Ende der Liste kostet $O(n + m)$ Gesamtzeit. ✓
- D** ~~Einfügen in der Liste hat amortisierte Kosten von $O(1)$.~~



→ sli.do/cs210

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

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) \rightsquigarrow nicht nutzen
wie `Vector` sind die Methoden `synchronized` \rightsquigarrow langsam ohne Nutzen
 - \rightsquigarrow stattdessen `Deque` verwenden ...

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) \rightsquigarrow nicht nutzen
wie `Vector` sind die Methoden `synchronized` \rightsquigarrow langsam ohne Nutzen
 \rightsquigarrow stattdessen `Deque` verwenden ...
- Alle sind generische Typen, also z.B. `List<Integer>`

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) \rightsquigarrow nicht nutzen
wie `Vector` sind die Methoden `synchronized` \rightsquigarrow langsam ohne Nutzen
 \rightsquigarrow stattdessen `Deque` verwenden ...
- Alle sind generische Typen, also z.B. `List<Integer>`
- weitere Interfaces `java.util.Map` und `java.util.(Sorted)Set` \rightsquigarrow Unit 9–10

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) \rightsquigarrow nicht nutzen
wie `Vector` sind die Methoden `synchronized` \rightsquigarrow langsam ohne Nutzen
 \rightsquigarrow stattdessen `Deque` verwenden ...
- ▶ Alle sind generische Typen, also z.B. `List<Integer>`
- ▶ weitere Interfaces `java.util.Map` und `java.util.(Sorted)Set` \rightsquigarrow Unit 9–10

▶ Implementierungen

- ▶ `java.util.LinkedList`: doppelt verkettete Liste
- ▶ `java.util.ArrayList`: dynamische Arrays, Wachstumsfaktor 1.5 statt 2
 - ▶ **!** schrumpft nicht automatisch! \rightsquigarrow `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

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

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

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

statt

```
1 Object x = null;
```

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

Iteratoren für unsere Lineare Liste

Java erlaubt Kurzschreibweise

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

statt

```
1 Object x = null;
```

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

Wir können diese auch für unsere Collections ermöglichen:

```
1 interface LinkedList<Elem, Position> extends Iterable<Elem> {  
2     // ...  
3     default Iterator<Elem> iterator() {  
4         return new Iterator<Elem>() {  
5             Position cur = start();  
6             public boolean hasNext() { return !cur.equals(end()); }  
7             public Elem next() {  
8                 if (!hasNext()) throw new NoSuchElementException();  
9                 Elem elem = get(cur);  
10                cur = LinkedList.this.next(cur);  
11                return elem;  
12            }  
13        };  
14    }  
15 }
```

Zusammenfassung

Lineare Listen gut unterstützt.

oder real time Anforderung
✓

- ▶ 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