

# Einführung in die Programmierung II

## Listen-Datentypen

Tobias Joschko

30. März 2022

# Themenüberblick – Listen-Datentypen

Arrays

Einfach verkettete Listen

Doppelt verkettete Listen

Stacks und Queues

Datensätze in Listen

Zusammenfassung: Entwurfsprinzipien

# Themenüberblick – Listen-Datentypen

Arrays

Einfach verkettete Listen

Doppelt verkettete Listen

Stacks und Queues

Datensätze in Listen

Zusammenfassung: Entwurfsprinzipien

# Arrays

Array: Zusammenhängender Bereich im Speicher

# Arrays

Array: Zusammenhängender Bereich im Speicher

## Vorteile

- ▶ Zugriffe i.d.R. sehr schnell
- ▶ wahlfreier Zugriff möglich
- ▶ kann leicht durchlaufen werden (Pointerberechnungen)

# Arrays

Array: Zusammenhängender Bereich im Speicher

## Vorteile

- ▶ Zugriffe i.d.R. sehr schnell
- ▶ wahlfreier Zugriff möglich
- ▶ kann leicht durchlaufen werden (Pointerberechnungen)

## Nachteile

- ▶ feste Größe
- ▶ bei Verletzung der Grenzen können Fehler auftreten
- ▶ Größenänderung unmöglich bzw. teuer
- ▶ evtl. schlechte Ausnutzung des Speichers

# Arrays

Lösungsansatz: Dynamische Arrays

# Arrays

Lösungsansatz: Dynamische Arrays

Idee: Zugriffe nicht direkt mittels [], sondern durch Funktionen:



# Arrays

## Lösungsansatz: Dynamische Arrays

Idee: Zugriffe nicht direkt mittels [], sondern durch Funktionen:

- ▶ z.B.: `array.get(i)` und `array.set(i,e1)`
  - ▶ `array.get(i)` liest im array an Stelle `i`
  - ▶ prüft dabei, ob `i` ein gültiger Index ist
  - ▶ `array.set(i,e1)` schreibt Element `e1` an Stelle `i`
  - ▶ verändert ggf. die Größe des Arrays

# Arrays

## Lösungsansatz: Dynamische Arrays

Idee: Zugriffe nicht direkt mittels [], sondern durch Funktionen:

- ▶ z.B.: `array.get(i)` und `array.set(i,e1)`
  - ▶ `array.get(i)` liest im array an Stelle `i`
  - ▶ prüft dabei, ob `i` ein gültiger Index ist
  - ▶ `array.set(i,e1)` schreibt Element `e1` an Stelle `i`
  - ▶ verändert ggf. die Größe des Arrays
- ▶ z.B.: `at(array,i)`
  - ▶ liefert das Element an Stelle `i`
  - ▶ prüft dabei die Arraygrenzen

# Arrays

## Lösungsansatz: Dynamische Arrays

Idee: Zugriffe nicht direkt mittels [], sondern durch Funktionen:

- ▶ z.B.: `array.get(i)` und `array.set(i,e1)`
  - ▶ `array.get(i)` liest im array an Stelle `i`
  - ▶ prüft dabei, ob `i` ein gültiger Index ist
  - ▶ `array.set(i,e1)` schreibt Element `e1` an Stelle `i`
  - ▶ verändert ggf. die Größe des Arrays
- ▶ z.B.: `at(array,i)`
  - ▶ liefert das Element an Stelle `i`
  - ▶ prüft dabei die Arraygrenzen
- ▶ z.B.: `push_back`, `pop_back`
  - ▶ `push_back` fügt ein Element am Ende der Liste hinzu.
  - ▶ `pop_back` löscht das letzte Element der Liste.
  - ▶ **Standardoperationen auf Listen**

# Arrays

## Implementierung dynamischer Arrays

Record-Datentyp (`struct`), der ein Array enthält

- ▶ Felder:
  - ▶ Pointer auf das Array
  - ▶ Größe des reservierten Speichers
  - ▶ Tatsächliche Anzahl der Elemente

# Arrays

## Implementierung dynamischer Arrays

### Record-Datentyp (struct), der ein Array enthält

- ▶ Felder:
  - ▶ Pointer auf das Array
  - ▶ Größe des reservierten Speichers
  - ▶ Tatsächliche Anzahl der Elemente
- ▶ Member-Funktionen für Zugriff:
  - ▶ `push_back`
  - ▶ `pop_back`
  - ▶ `get`
  - ▶ `reallocate` (intern von `push_back` benutzt)

# Themenüberblick – Listen-Datentypen

Arrays

Einfach verkettete Listen

Doppelt verkettete Listen

Stacks und Queues

Datensätze in Listen

Zusammenfassung: Entwurfsprinzipien

# Einfach verkettete Listen

Verkettete Liste: Elemente stehen verteilt im Speicher

# Einfach verkettete Listen

## Verkettete Liste: Elemente stehen verteilt im Speicher

- ▶ Idee: Elemente der Liste bestehen aus zwei Teilen
  - ▶ Daten
  - ▶ Pointer auf das nächste Element



# Einfach verkettete Listen

## Verkettete Liste: Elemente stehen verteilt im Speicher

- ▶ Idee: Elemente der Liste bestehen aus zwei Teilen
  - ▶ Daten
  - ▶ Pointer auf das nächste Element

## Vorteile:

- ▶ Größe ist nicht fest vorgegeben
- ▶ sehr effiziente Speicherausnutzung
- ▶ Einfügen und Löschen von Elementen sehr effizient

# Einfach verkettete Listen

## Verkettete Liste: Elemente stehen verteilt im Speicher

- ▶ Idee: Elemente der Liste bestehen aus zwei Teilen
  - ▶ Daten
  - ▶ Pointer auf das nächste Element

## Vorteile:

- ▶ Größe ist nicht fest vorgegeben
- ▶ sehr effiziente Speicherausnutzung
- ▶ Einfügen und Löschen von Elementen sehr effizient

## Nachteile:

- ▶ Kein wahlfreier Zugriff
- ▶ Durchlauf ineffizient

# Einfach verkettete Listen

Implementierung einfach verketteter Listen

Elemente: struct mit Daten und Pointer auf Nachfolger

Liste: struct mit Pointer auf den Anfang der Liste

- ▶ Felder:
  - ▶ Pointer auf das erste Element der Liste
  - ▶ evtl. weitere Pointer (für bessere Performance)

# Einfach verkettete Listen

Implementierung einfach verketteter Listen

Elemente: struct mit Daten und Pointer auf Nachfolger

Liste: struct mit Pointer auf den Anfang der Liste

- ▶ Felder:
  - ▶ Pointer auf das erste Element der Liste
  - ▶ evtl. weitere Pointer (für bessere Performance)
- ▶ Basisfunktionen für Zugriff:
  - ▶ push\_back
  - ▶ pop\_back
  - ▶ get

# Einfach verkettete Listen

Implementierung einfach verketteter Listen

Elemente: struct mit Daten und Pointer auf Nachfolger

Liste: struct mit Pointer auf den Anfang der Liste

- ▶ Felder:
  - ▶ Pointer auf das erste Element der Liste
  - ▶ evtl. weitere Pointer (für bessere Performance)
- ▶ Basisfunktionen für Zugriff:
  - ▶ push\_back
  - ▶ pop\_back
  - ▶ get
- ▶ Ende wird durch ein Dummy-Element markiert.
  - ▶ Sentinel-Prinzip, vgl. terminierende Null bei Strings

# Themenüberblick – Listen-Datentypen

Arrays

Einfach verkettete Listen

Doppelt verkettete Listen

Stacks und Queues

Datensätze in Listen

Zusammenfassung: Entwurfsprinzipien

# Doppelt verkettete Listen

Doppelte Verkettung: Pointer auf Nachfolger und Vorgänger

# Doppelt verkettete Listen

## Doppelte Verkettung: Pointer auf Nachfolger und Vorgänger

- ▶ Idee: Elemente der Liste bestehen aus drei Teilen
  - ▶ Daten
  - ▶ Pointer auf das nächste Element
  - ▶ Pointer auf das vorhergehende Element



# Doppelt verkettete Listen

## Doppelte Verkettung: Pointer auf Nachfolger und Vorgänger

- ▶ Idee: Elemente der Liste bestehen aus drei Teilen
  - ▶ Daten
  - ▶ Pointer auf das nächste Element
  - ▶ Pointer auf das vorhergehende Element

## Vor- und Nachteile:

- ▶ effizienter bei wiederholten Zugriffen auf benachbarte Elemente
- ▶ etwas mehr Speicherverbrauch als einfach verkettete Liste
  - ▶ kann i.d.R. vernachlässigt werden

# Doppelt verkettete Listen

## Doppelte Verkettung: Pointer auf Nachfolger und Vorgänger

- ▶ Idee: Elemente der Liste bestehen aus drei Teilen
  - ▶ Daten
  - ▶ Pointer auf das nächste Element
  - ▶ Pointer auf das vorhergehende Element

## Vor- und Nachteile:

- ▶ effizienter bei wiederholten Zugriffen auf benachbarte Elemente
- ▶ etwas mehr Speicherverbrauch als einfach verkettete Liste
  - ▶ kann i.d.R. vernachlässigt werden

## Verwaltung durch Dummy-Element

- ▶ Nutze ein spezielles leeres Element (HEAD)
- ▶ markiert Anfang und Ende der Liste

# Doppelt verkettete Listen

Implementierung doppelt verketteter Listen

Elemente: struct mit Daten und Pointer auf Nachbarn

Liste: struct mit Pointer auf den Kopf der Liste

- ▶ Felder:
  - ▶ Pointer auf das HEAD-Element

# Doppelt verkettete Listen

Implementierung doppelt verketteter Listen

Elemente: struct mit Daten und Pointer auf Nachbarn

Liste: struct mit Pointer auf den Kopf der Liste

- ▶ Felder:
  - ▶ Pointer auf das HEAD-Element
- ▶ Basisfunktionen für Zugriff:
  - ▶ push\_back
  - ▶ pop\_back
  - ▶ get

# Themenüberblick – Listen-Datentypen

Arrays

Einfach verkettete Listen

Doppelt verkettete Listen

Stacks und Queues

Datensätze in Listen

Zusammenfassung: Entwurfsprinzipien

# Stacks und Queues

## Stacks: *Stapel-* oder *Kellerspeicher*

- ▶ Elemente werden *gestapelt*.
- ▶ Nur das zuletzt eingefügte Element ist zugänglich
  - ▶ „Last-In-First-Out“ (LIFO)

# Stacks und Queues

## Stacks: *Stapel-* oder *Kellerspeicher*

- ▶ Elemente werden *gestapelt*.
- ▶ Nur das zuletzt eingefügte Element ist zugänglich
  - ▶ „Last-In-First-Out“ (LIFO)
- ▶ Standardoperationen: push, pop und top
  - ▶ push fügt ein Element hinzu
  - ▶ pop entfernt das oberste Element
  - ▶ top liefert das oberste Element zurück

# Stacks und Queues

## Stacks: *Stapel-* oder *Kellerspeicher*

- ▶ Elemente werden *gestapelt*.
- ▶ Nur das zuletzt eingefügte Element ist zugänglich
  - ▶ „Last-In-First-Out“ (LIFO)
- ▶ Standardoperationen: push, pop und top
  - ▶ push fügt ein Element hinzu
  - ▶ pop entfernt das oberste Element
  - ▶ top liefert das oberste Element zurück
- ▶ Implementierung durch Arrays oder verkettete Listen



# Stacks und Queues

## Stacks: *Stapel-* oder *Kellerspeicher*

- ▶ Elemente werden *gestapelt*.
- ▶ Nur das zuletzt eingefügte Element ist zugänglich
  - ▶ „Last-In-First-Out“ (LIFO)
- ▶ Standardoperationen: push, pop und top
  - ▶ push fügt ein Element hinzu
  - ▶ pop entfernt das oberste Element
  - ▶ top liefert das oberste Element zurück
- ▶ Implementierung durch Arrays oder verkettete Listen
- ▶ Anwendungsbeispiele:
  - ▶ Der *Stack* im Hauptspeicher
  - ▶ Pufferspeicher bei rekursiven Problemen (z.B. Damenproblem)
  - ▶ Einfaches Speichermodell in Kleinstrechnern (z.B. Taschenrechner)

# Stacks und Queues

Queues: *Warteschlange* oder *Pufferspeicher*

- ▶ Nur das zuerst eingefügte Element ist zugänglich
  - ▶ „First-In-First-Out“ (FIFO)

# Stacks und Queues

## Queues: *Warteschlange* oder *Pufferspeicher*

- ▶ Nur das zuerst eingefügte Element ist zugänglich
  - ▶ „First-In-First-Out“ (FIFO)
- ▶ Standardoperationen: push, pop
  - ▶ push fügt ein Element am Anfang der Liste hinzu
  - ▶ pop entfernt das letzte Element der Liste

# Stacks und Queues

## Queues: *Warteschlange* oder *Pufferspeicher*

- ▶ Nur das zuerst eingefügte Element ist zugänglich
  - ▶ „First-In-First-Out“ (FIFO)
- ▶ Standardoperationen: push, pop
  - ▶ push fügt ein Element am Anfang der Liste hinzu
  - ▶ pop entfernt das letzte Element der Liste
- ▶ Implementierung durch Arrays oder verkettete Listen

# Stacks und Queues

## Queues: *Warteschlange* oder *Pufferspeicher*

- ▶ Nur das zuerst eingefügte Element ist zugänglich
  - ▶ „First-In-First-Out“ (FIFO)
- ▶ Standardoperationen: push, pop
  - ▶ push fügt ein Element am Anfang der Liste hinzu
  - ▶ pop entfernt das letzte Element der Liste
- ▶ Implementierung durch Arrays oder verkettete Listen
- ▶ Anwendungsbeispiele:
  - ▶ Pufferspeicher bei der Kommunikation
    - ▶ Netzwerke
    - ▶ Ein-/Ausgabe von Computern
    - ▶ Kommunikation unterschiedlicher Threads

# Stacks und Queues

## Priority Queues: Warteschlangen mit Gewichten

- ▶ Jedes Element hat eine Gewichtung bzw. eine Priorität
- ▶ Nur das schwerste/wichtigste Element ist zugänglich.

# Stacks und Queues

## Priority Queues: Warteschlangen mit Gewichten

- ▶ Jedes Element hat eine Gewichtung bzw. eine Priorität
- ▶ Nur das schwerste/wichtigste Element ist zugänglich.
- ▶ Standardoperationen: push, pop, get
  - ▶ push fügt ein Element am Anfang der Liste hinzu
  - ▶ pop entfernt das eine zugängliche Element.

# Stacks und Queues

## Priority Queues: Warteschlangen mit Gewichten

- ▶ Jedes Element hat eine Gewichtung bzw. eine Priorität
- ▶ Nur das schwerste/wichtigste Element ist zugänglich.
- ▶ Standardoperationen: push, pop, get
  - ▶ push fügt ein Element am Anfang der Liste hinzu
  - ▶ pop entfernt das eine zugängliche Element.
- ▶ Implementierung durch spezielle Baumstrukturen (Heaps)



# Stacks und Queues

## Priority Queues: Warteschlangen mit Gewichten

- ▶ Jedes Element hat eine Gewichtung bzw. eine Priorität
- ▶ Nur das schwerste/wichtigste Element ist zugänglich.
- ▶ Standardoperationen: push, pop, get
  - ▶ push fügt ein Element am Anfang der Liste hinzu
  - ▶ pop entfernt das eine zugängliche Element.
- ▶ Implementierung durch spezielle Baumstrukturen (Heaps)
- ▶ Anwendungsbeispiele:
  - ▶ Routenplanungs- und Scheduling-Algorithmen
  - ▶ schnelle Sortierverfahren

# Themenüberblick – Listen-Datentypen

Arrays

Einfach verkettete Listen

Doppelt verkettete Listen

Stacks und Queues

**Datensätze in Listen**

Zusammenfassung: Entwurfsprinzipien

# Datensätze in Listen

## Daten sind abstrakter Bestandteil in Listen

- ▶ in theoretischen Beispielen meist `int`
- ▶ in der Praxis oft komplexer
  - ▶ zusammengesetzte Daten (Record-Datentypen)
  - ▶ Schlüssel können z.B. auch Strings o.Ä. sein.

# Datensätze in Listen

## Daten sind abstrakter Bestandteil in Listen

- ▶ in theoretischen Beispielen meist `int`
- ▶ in der Praxis oft komplexer
  - ▶ zusammengesetzte Daten (Record-Datentypen)
  - ▶ Schlüssel können z.B. auch Strings o.Ä. sein.

## Objektorientierter Ansatz für einheitlichen Entwurf:

- ▶ Daten werden immer in einem Datentyp `element` gekapselt.
- ▶ Elemente sind Bestandteile der Listen, Arrays etc.
- ▶ Zugriff auf die Daten mittels Funktionen.
- ▶ Die Implementierung der eigentlichen Liste, des Arrays etc. muss dabei nicht verändert werden.
  - ▶ In C++: Meist werden Templates verwendet.

# Themenüberblick – Listen-Datentypen

Arrays

Einfach verkettete Listen

Doppelt verkettete Listen

Stacks und Queues

Datensätze in Listen

Zusammenfassung: Entwurfsprinzipien

# Zusammenfassung: Entwurfsprinzipien

Arrays, Listen, Bäume etc. sind **abstrakte Datentypen**

- ▶ Verhalten wird nur durch Funktionen spezifiziert.
- ▶ Listen-Datentypen haben Funktionen wie `push`, `pop` etc.
- ▶ Die Implementierung / das Verhalten dieser Funktionen bestimmt, um was für einen Datentyp es sich genau handelt.

# Zusammenfassung: Entwurfsprinzipien

## Implementierung abstrakter Datentypen

- ▶ Grundlegende Datenstruktur ist immer ein `struct`, das die konkrete Implementierung enthält.
  - ▶ z.B. Pointer auf Listenelemente, ein C-Array etc.
  - ▶ Alternativ auch andere Datentypen-structs (z.B. bei Stacks und Queues)
  - ▶ zusätzlich ggf. Hilfsdaten wie z.B. die Länge der Liste.
- ▶ Zugriff auf den Datentyp mittels Funktionen.
- ▶ Elemente der Liste etc. auch als `struct`, in dem auch Daten verwaltet werden. Zugriff auch hier durch Funktionen.

# Zusammenfassung: Entwurfsprinzipien

## Vorteile dieses Ansatzes:

- ▶ Die Funktionen **verstecken** die eigentliche Implementierung.
  - ▶ Technische Details wie Pointer und Hilfsvariablen spielen für den Benutzer keine Rolle.
- ▶ Der Elementtyp bzw. die Daten sind leicht austauschbar.
  - ▶ Will man die zugrundeliegenden Daten ändern, muss man nur das Struct und die Funktionen für den Elementtyp ändern.