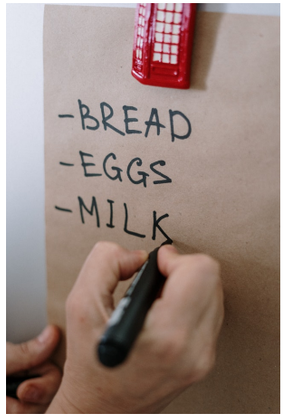
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Algorithmen und Datenstrukturen

Vorlesung 11:
Elementare Datenstrukturen
Stapel, Schlangen und Listen



Alexander Wolff

Wintersemester 2024

Zur Erinnerung

Datenstruktur.

Konzept, mit dem man Daten speichert und anordnet, so dass man sie schnell finden und ändern kann.

Zur Erinnerung

Datenstruktur.

Konzept, mit dem man Daten speichert und anordnet, so dass man sie schnell finden und ändern kann.

Zur Erinnerung

Datenstruktur.

Konzept, mit dem man Daten speichert und anordnet, so dass man sie schnell finden und ändern kann.

Abstrakter Datentyp.

Implementierung.

Zur Erinnerung

Datenstruktur.

Konzept, mit dem man Daten speichert und anordnet, so dass man sie schnell finden und ändern kann.

Abstrakter Datentyp.

beschreibt die „Schnittstelle“ einer Datenstruktur – welche Operationen werden unterstützt?

Implementierung.

Zur Erinnerung

Datenstruktur.

Konzept, mit dem man Daten speichert und anordnet, so dass man sie schnell finden und ändern kann.

Abstrakter Datentyp.

beschreibt die „Schnittstelle“ einer Datenstruktur – welche Operationen werden unterstützt?

Implementierung.

wie wird die gewünschte Funktionalität realisiert:
– wie sind die Daten gespeichert (Feld, Liste, . . .)?
– welche Algorithmen implementieren die Operationen?

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Implementierung	INSERT	FINDMIN	EXTRACTMIN	DECREASEKEY

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Implementierung	Insert	FindMin	ExtractMin	DecreaseKey
Karten in beliebiger Reihenfolge				

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

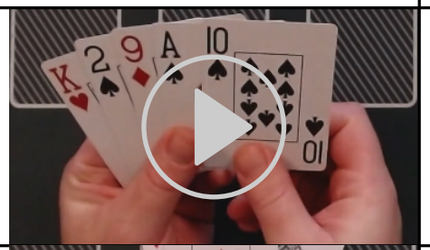
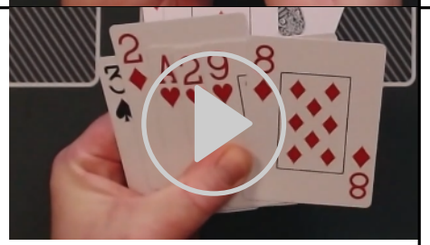
verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Implementierung	INSERT	FINDMIN	EXTRACTMIN	DECREASEKEY
Karten in beliebiger Reihenfolge	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

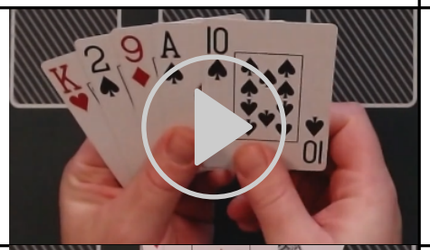
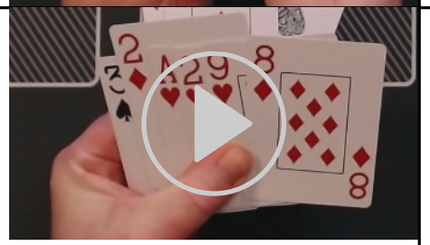
verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Implementierung	INSERT	FINDMIN	EXTRACTMIN	DECREASEKEY
Karten in beliebiger Reihenfolge 	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Karten in beliebiger Reihenfolge, kleinste markiert 				

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

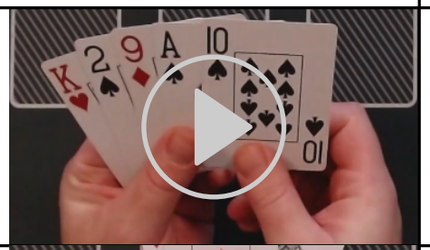
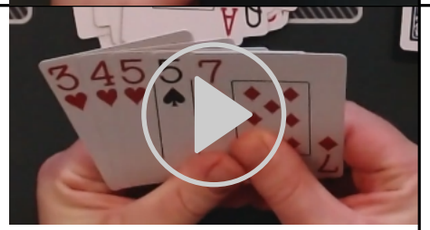
verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Implementierung	INSERT	FINDMIN	EXTRACTMIN	DECREASEKEY
Karten in beliebiger Reihenfolge 	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Karten in beliebiger Reihenfolge, kleinste markiert 	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

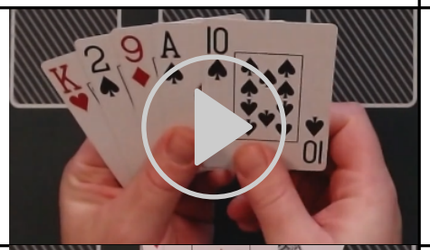
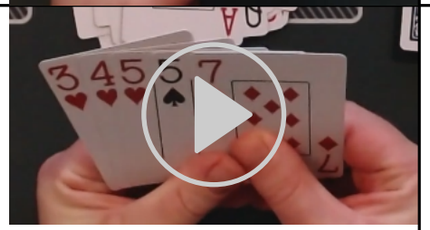
verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Implementierung	INSERT	FINDMIN	EXTRACTMIN	DECREASEKEY
Karten in beliebiger Reihenfolge 	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Karten in **sortierter** Reihenfolge 				

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

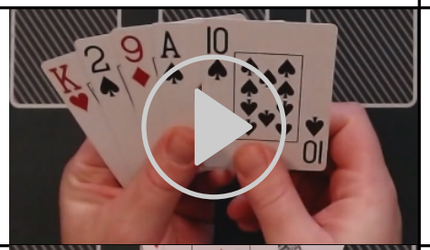
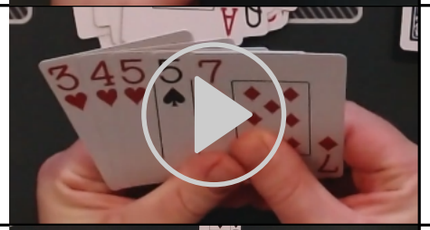
verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Implementierung	INSERT	FINDMIN	EXTRACTMIN	DECREASEKEY
Karten in beliebiger Reihenfolge 	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Karten in **sortierter** Reihenfolge 	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

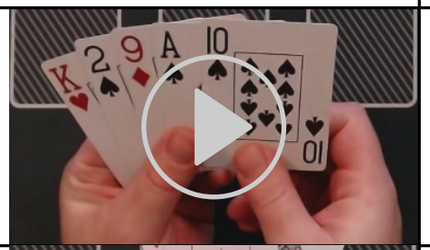
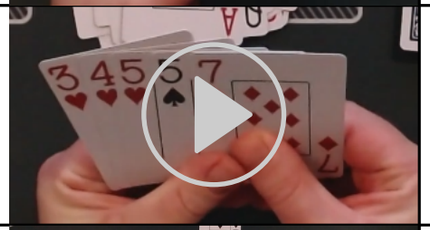
verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Implementierung	INSERT	FINDMIN	EXTRACTMIN	DECREASEKEY
Karten in beliebiger Reihenfolge 	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Karten in **sortierter** Reihenfolge 	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Karten als MINHEAP				

Prioritätsschlange

Abstrakter Datentyp: Prioritätsschlange

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ eine Priorität $x.key$ hat.

Implementierung	INSERT	FINDMIN	EXTRACTMIN	DECREASEKEY
Karten in beliebiger Reihenfolge 	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Karten in beliebiger Reihenfolge, kleinste markiert	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Karten in **sortierter** Reihenfolge 	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Karten als MINHEAP	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Operation	Funktionalität

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Operation	Funktionalität
INSERT(key k, value v)	

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Operation	Funktionalität
INSERT(key k, value v)	Füge Element (k, v) ein

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Operation	Funktionalität
INSERT(key k, value v)	Füge Element (k, v) ein
DELETE(x)	

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Operation	Funktionalität
INSERT(key k, value v)	Füge Element (k, v) ein
DELETE(x)	Entferne Element x

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Operation	Funktionalität
INSERT(key k, value v)	Füge Element (k, v) ein
DELETE(x)	Entferne Element x
SEARCH(key k)	

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Operation	Funktionalität
INSERT(key k, value v)	Füge Element (k, v) ein
DELETE(x)	Entferne Element x
SEARCH(key k)	Suche Element x mit $x.key = k$

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Operation	Funktionalität
INSERT(key k, value v)	Füge Element (k, v) ein
DELETE(x)	Entferne Element x
SEARCH(key k)	Suche Element x mit $x.key = k$
MINIMUM()	

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Operation	Funktionalität
INSERT(key k, value v)	Füge Element (k, v) ein
DELETE(x)	Entferne Element x
SEARCH(key k)	Suche Element x mit $x.key = k$
MINIMUM()	Suche Element x mit kleinstem Schlüssel $x.key$

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Operation	Funktionalität
INSERT(key k, value v)	Füge Element (k, v) ein
DELETE(x)	Entferne Element x
SEARCH(key k)	Suche Element x mit $x.key = k$
MINIMUM()	Suche Element x mit kleinstem Schlüssel $x.key$
MAXIMUM()	

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Operation	Funktionalität
INSERT(key k, value v)	Füge Element (k, v) ein
DELETE(x)	Entferne Element x
SEARCH(key k)	Suche Element x mit $x.key = k$
MINIMUM()	Suche Element x mit kleinstem Schlüssel $x.key$
MAXIMUM()	Suche Element x mit größtem Schlüssel $x.key$

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Operation	Funktionalität
INSERT(key k, value v)	Füge Element (k, v) ein
DELETE(x)	Entferne Element x
SEARCH(key k)	Suche Element x mit $x.key = k$
MINIMUM()	Suche Element x mit kleinstem Schlüssel $x.key$
MAXIMUM()	Suche Element x mit größtem Schlüssel $x.key$
PREDECESSOR(x)	

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Operation	Funktionalität
INSERT(key k, value v)	Füge Element (k, v) ein
DELETE(x)	Entferne Element x
SEARCH(key k)	Suche Element x mit $x.key = k$
MINIMUM()	Suche Element x mit kleinstem Schlüssel $x.key$
MAXIMUM()	Suche Element x mit größtem Schlüssel $x.key$
PREDECESSOR(x)	Suche Element y mit $y.key$ aus $\{(k, v) \mid k < x.key\}$

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Operation	Funktionalität
INSERT(key k, value v)	Füge Element (k, v) ein
DELETE(x)	Entferne Element x
SEARCH(key k)	Suche Element x mit $x.key = k$
MINIMUM()	Suche Element x mit kleinstem Schlüssel $x.key$
MAXIMUM()	Suche Element x mit größtem Schlüssel $x.key$
PREDECESSOR(x)	Suche Element y mit größtem Schlüssel $y.key$ aus $\{(k, v) \mid k < x.key\}$

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Operation	Funktionalität
INSERT(key k, value v)	Füge Element (k, v) ein
DELETE(x)	Entferne Element x
SEARCH(key k)	Suche Element x mit $x.key = k$
MINIMUM()	Suche Element x mit kleinstem Schlüssel $x.key$
MAXIMUM()	Suche Element x mit größtem Schlüssel $x.key$
PREDECESSOR(x)	Suche Element y mit größtem Schlüssel $y.key$ aus $\{(k, v) \mid k < x.key\}$
SUCCESSOR(x)	

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Operation	Funktionalität
INSERT(key k, value v)	Füge Element (k, v) ein
DELETE(x)	Entferne Element x
SEARCH(key k)	Suche Element x mit $x.key = k$
MINIMUM()	Suche Element x mit kleinstem Schlüssel $x.key$
MAXIMUM()	Suche Element x mit größtem Schlüssel $x.key$
PREDECESSOR(x)	Suche Element y mit größtem Schlüssel $y.key$ aus $\{(k, v) \mid k < x.key\}$
SUCCESSOR(x)	Suche Element y mit kleinstem Schlüssel $y.key$ aus $\{(k, v) \mid k > x.key\}$

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	Füge Element (k, v) ein
DELETE(ptr x)	Entferne Element x
ptr SEARCH(key k)	Suche Element x mit $x.key = k$
ptr MINIMUM()	Suche Element x mit kleinstem Schlüssel $x.key$
ptr MAXIMUM()	Suche Element x mit größtem Schlüssel $x.key$
ptr PREDECESSOR(ptr x)	Suche Element y mit größtem Schlüssel $y.key$ aus $\{(k, v) \mid k < x.key\}$
ptr SUCCESSOR(ptr x)	Suche Element y mit kleinstem Schlüssel $y.key$ aus $\{(k, v) \mid k > x.key\}$

Pointer

Dynamische Menge

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

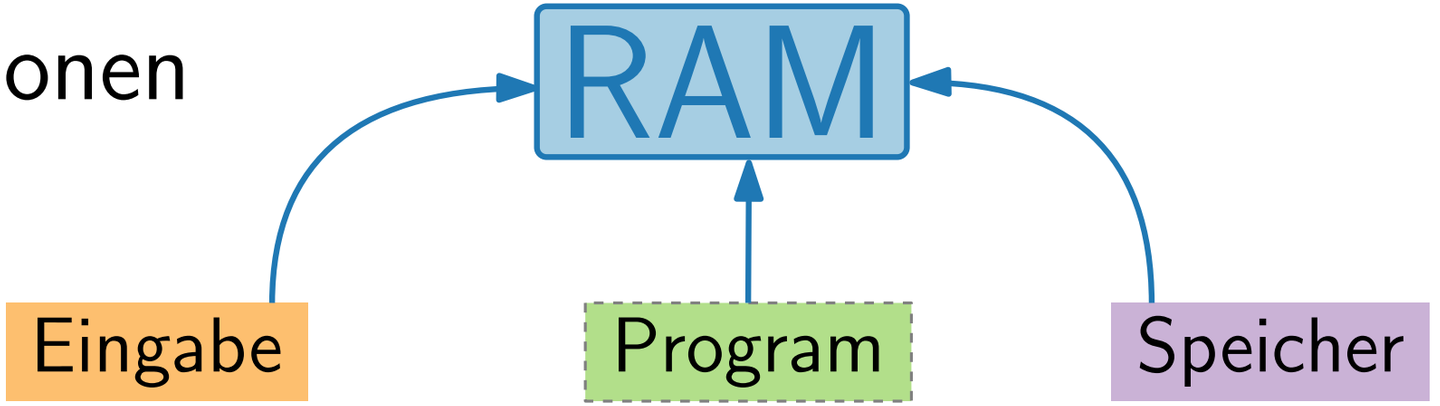
eindeutige Zahl

Beliebiges Objekt

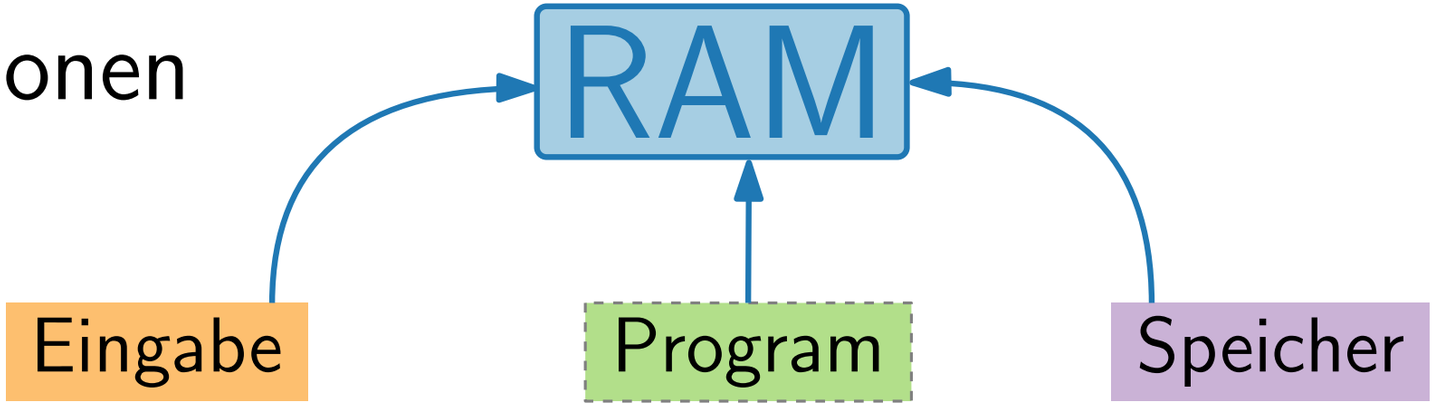
Operation	Funktionalität
ptr INSERT(key k, value v)	Füge Element (k, v) ein
DELETE(ptr x)	Entferne Element x
ptr SEARCH(key k)	Suche Element x mit $x.key = k$
ptr MINIMUM()	Suche Element x mit kleinstem Schlüssel $x.key$
ptr MAXIMUM()	Suche Element x mit größtem Schlüssel $x.key$
ptr PREDECESSOR(ptr x)	Suche Element y mit größtem Schlüssel $y.key$ aus $\{(k, v) \mid k < x.key\}$
ptr SUCCESSOR(ptr x)	Suche Element y mit kleinstem Schlüssel $y.key$ aus $\{(k, v) \mid k > x.key\}$

Pointer ?

Exkurs: Speicherregionen

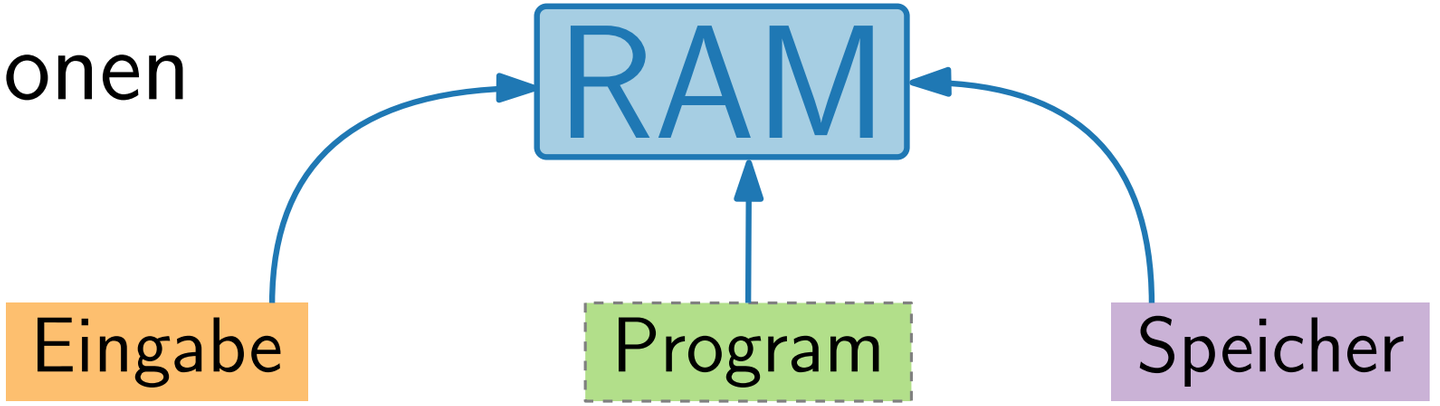


Exkurs: Speicherregionen



Der Speicher wird in 5 Regionen unterteilt:

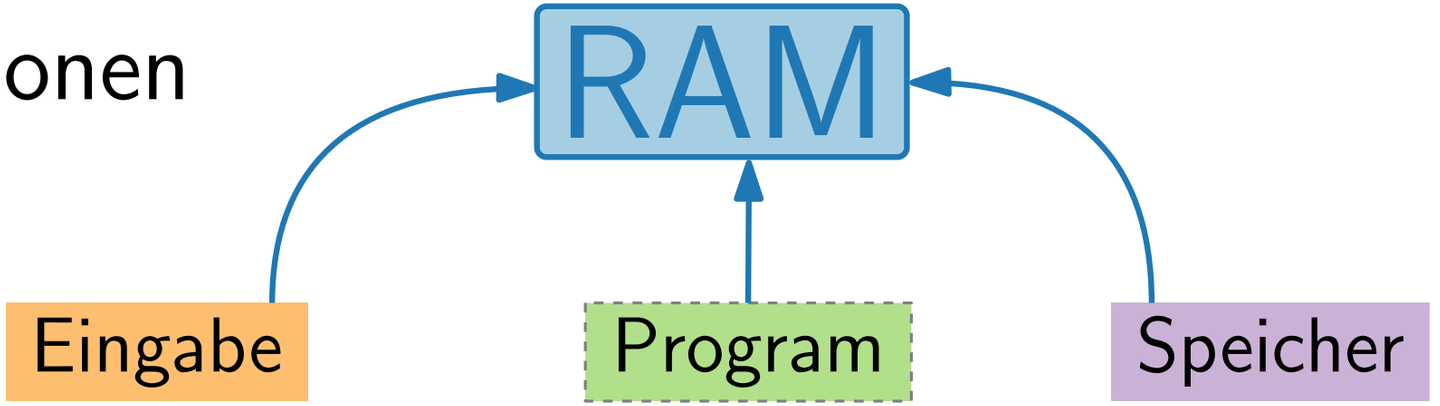
Exkurs: Speicherregionen



Der Speicher wird in 5 Regionen unterteilt:

Code	

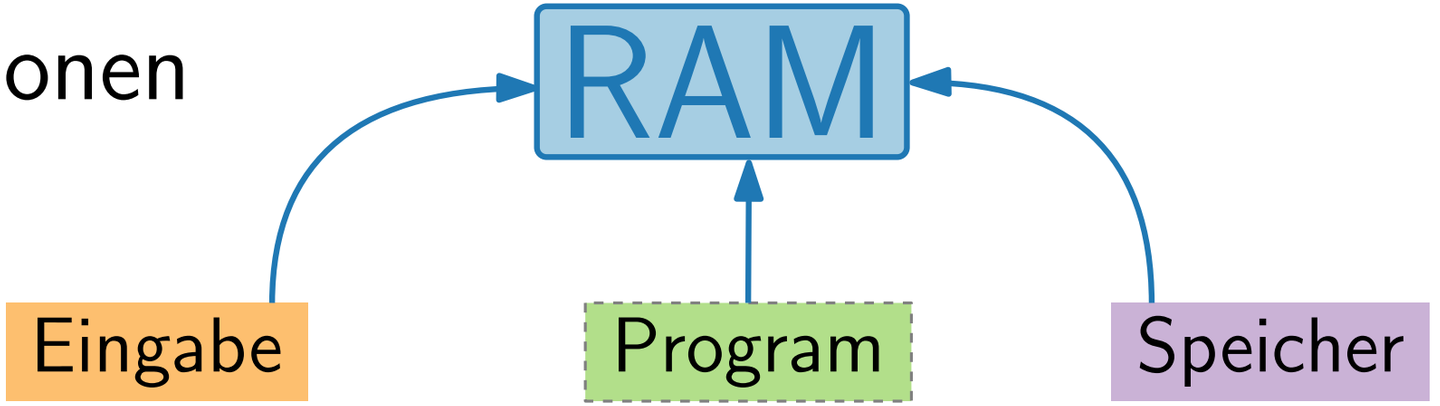
Exkurs: Speicherregionen



Der Speicher wird in 5 Regionen unterteilt:

Code	Maschinencode / kompilierter Code, vom Menschen nicht lesbar

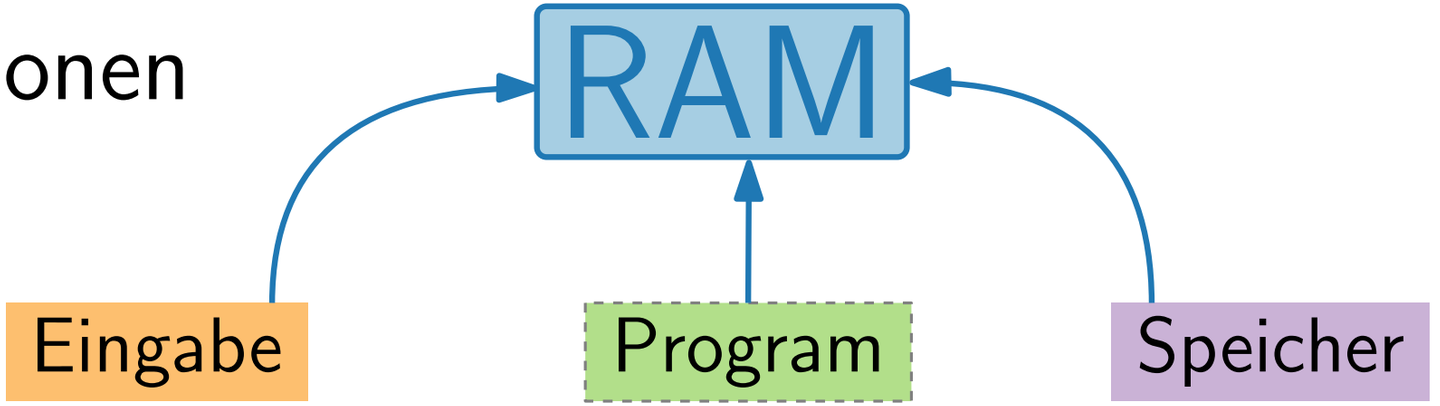
Exkurs: Speicherregionen



Der Speicher wird in 5 Regionen unterteilt:

Code	Maschinencode / kompilierter Code, vom Menschen nicht lesbar
Read-Only Data	

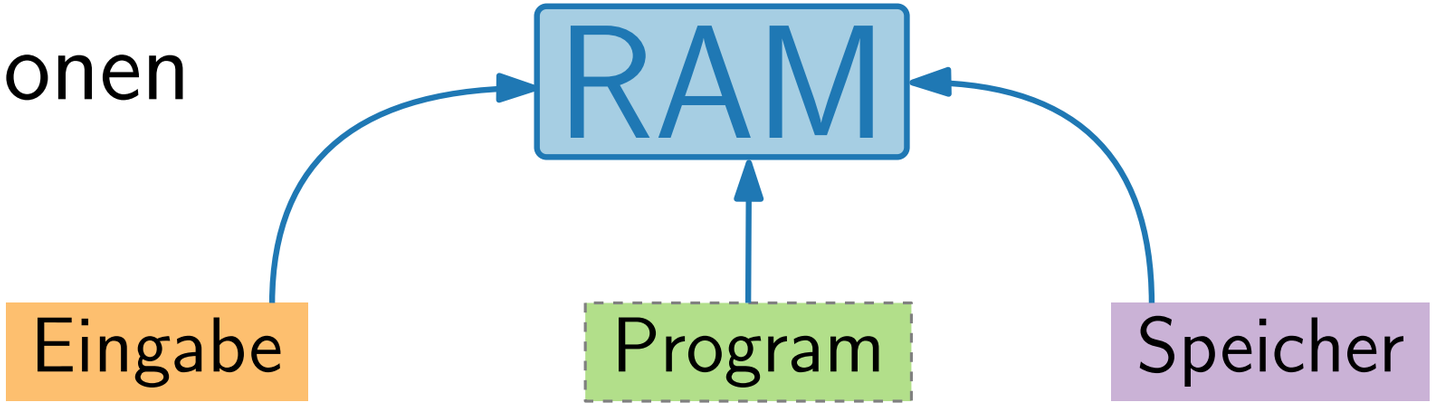
Exkurs: Speicherregionen



Der Speicher wird in 5 Regionen unterteilt:

Code	Maschinencode / kompilierter Code, vom Menschen nicht lesbar
Read-Only Data	Globale, nicht veränderbare Variablen

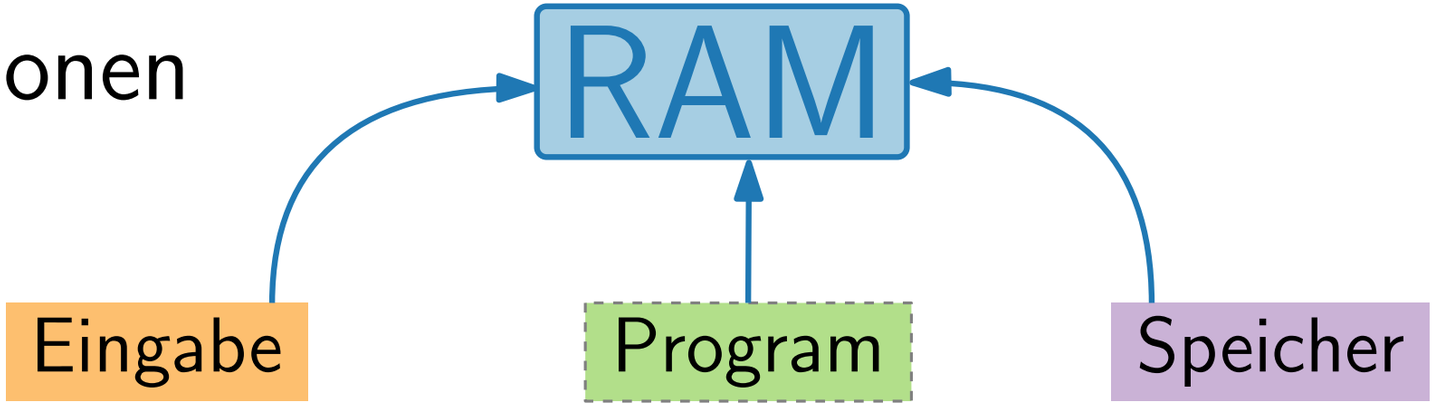
Exkurs: Speicherregionen



Der Speicher wird in 5 Regionen unterteilt:

Code	Maschinencode / kompilierter Code, vom Menschen nicht lesbar
Read-Only Data	Globale, nicht veränderbare Variablen
Global Data	

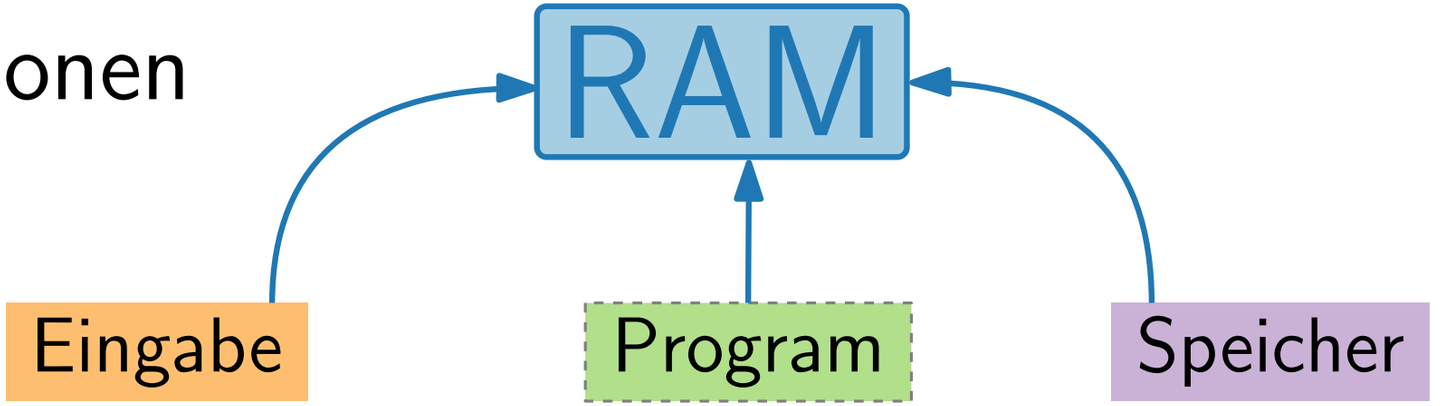
Exkurs: Speicherregionen



Der Speicher wird in 5 Regionen unterteilt:

Code	Maschinencode / kompilierter Code, vom Menschen nicht lesbar
Read-Only Data	Globale, nicht veränderbare Variablen
Global Data	Globale, veränderbare Variablen

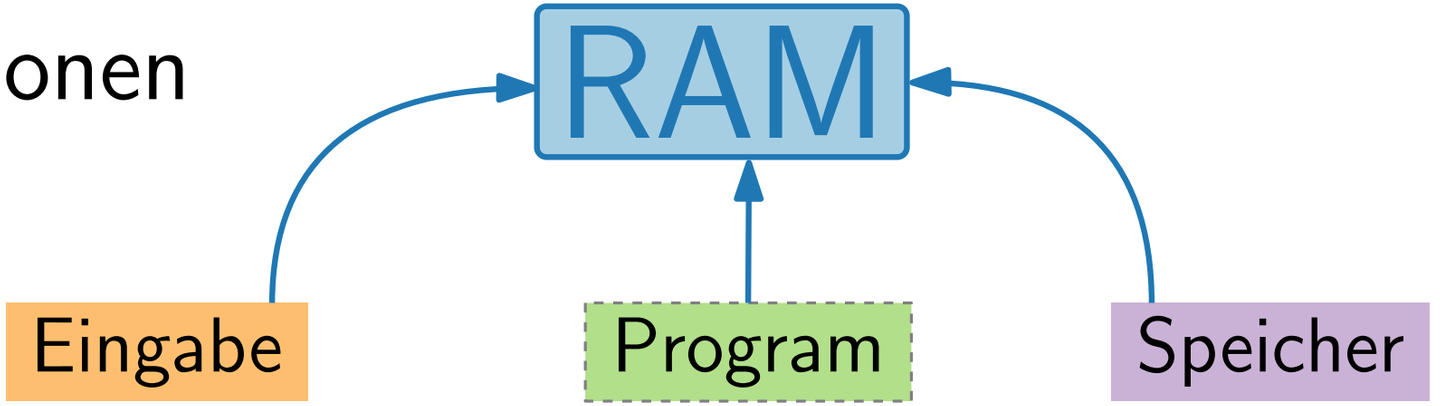
Exkurs: Speicherregionen



Der Speicher wird in 5 Regionen unterteilt:

Code	Maschinencode / kompilierter Code, vom Menschen nicht lesbar
Read-Only Data	Globale, nicht veränderbare Variablen
Global Data	Globale, veränderbare Variablen
Stack	

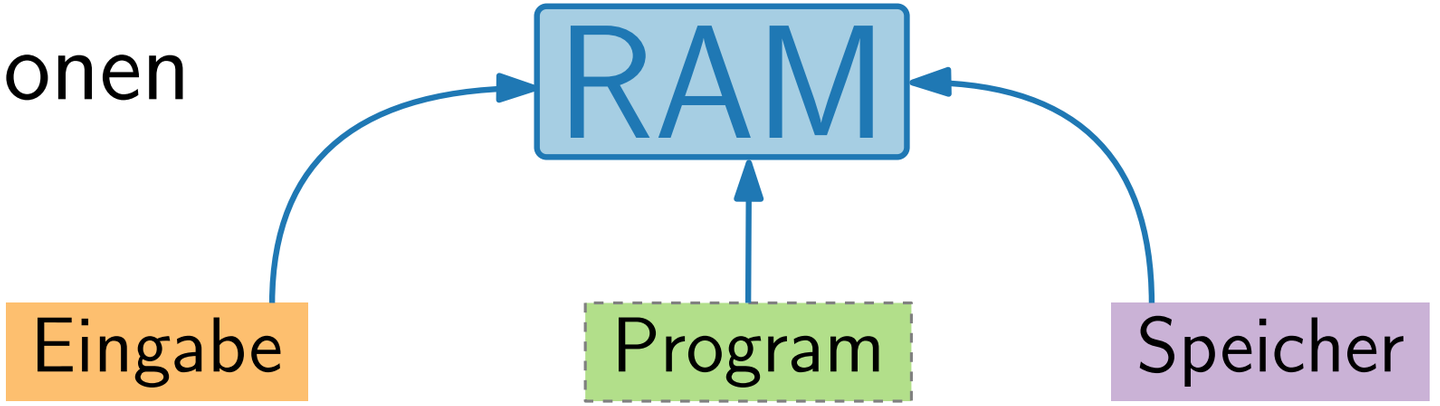
Exkurs: Speicherregionen



Der Speicher wird in 5 Regionen unterteilt:

Code	Maschinencode / kompilierter Code, vom Menschen nicht lesbar
Read-Only Data	Globale, nicht veränderbare Variablen
Global Data	Globale, veränderbare Variablen
Stack	■ Wenig, schneller Speicher

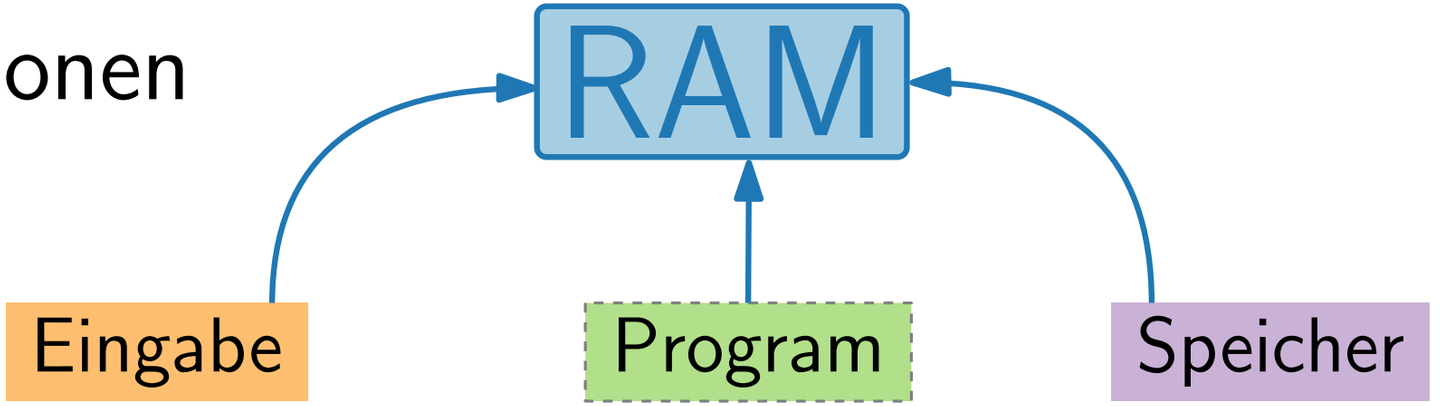
Exkurs: Speicherregionen



Der Speicher wird in 5 Regionen unterteilt:

Code	Maschinencode / kompilierter Code, vom Menschen nicht lesbar
Read-Only Data	Globale, nicht veränderbare Variablen
Global Data	Globale, veränderbare Variablen
Stack	■ Wenig, schneller Speicher ■ Pro aktivem Methodenaufruf ein **Stack Frame**:

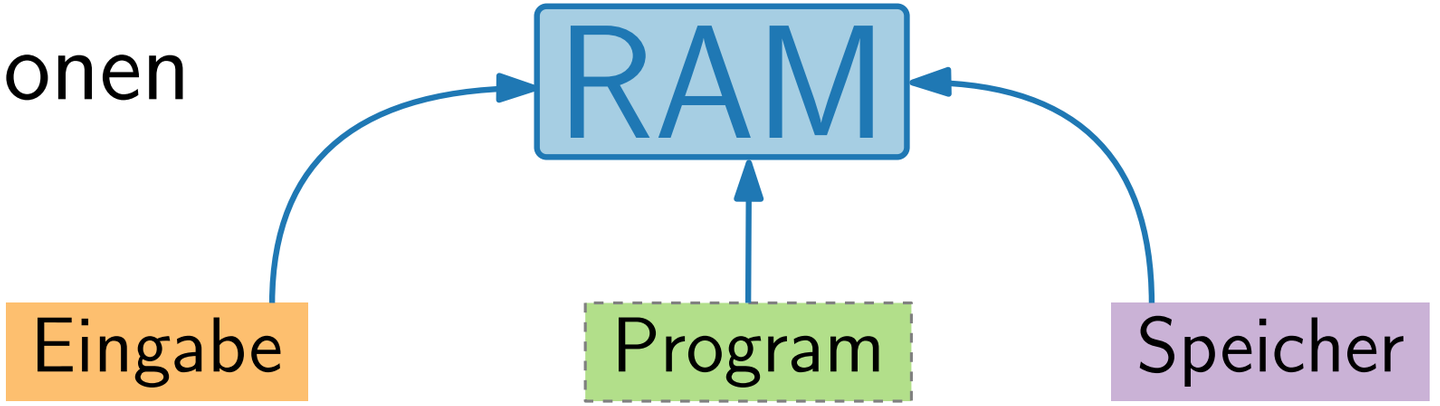
Exkurs: Speicherregionen



Der Speicher wird in 5 Regionen unterteilt:

Code	Maschinencode / kompilierter Code, vom Menschen nicht lesbar
Read-Only Data	Globale, nicht veränderbare Variablen
Global Data	Globale, veränderbare Variablen
Stack	■ Wenig, schneller Speicher ■ Pro aktivem Methodenaufruf ein **Stack Frame**: ■ Eingabe

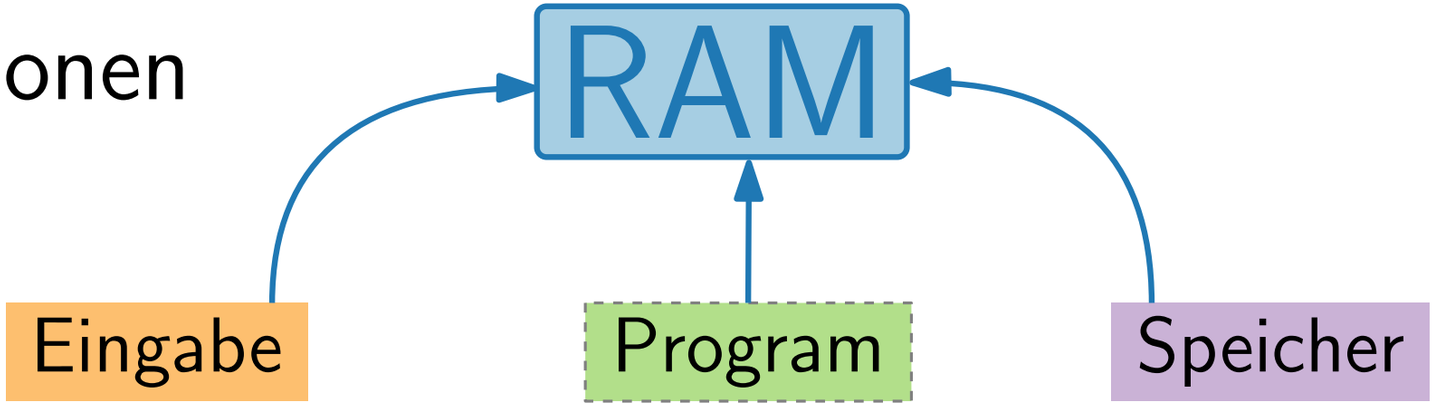
Exkurs: Speicherregionen



Der Speicher wird in 5 Regionen unterteilt:

Code	Maschinencode / kompilierter Code, vom Menschen nicht lesbar
Read-Only Data	Globale, nicht veränderbare Variablen
Global Data	Globale, veränderbare Variablen
Stack	■ Wenig, schneller Speicher ■ Pro aktivem Methodenaufruf ein **Stack Frame**: ■ Eingabe ■ lokale Variablen

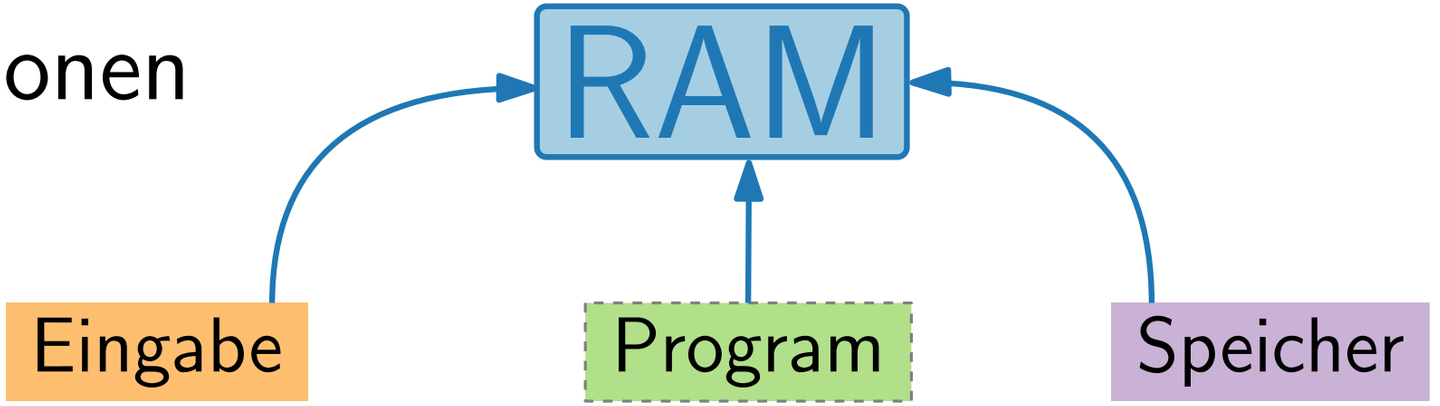
Exkurs: Speicherregionen



Der Speicher wird in 5 Regionen unterteilt:

Code	Maschinencode / kompilierter Code, vom Menschen nicht lesbar
Read-Only Data	Globale, nicht veränderbare Variablen
Global Data	Globale, veränderbare Variablen
Stack	■ Wenig, schneller Speicher ■ Pro aktivem Methodenaufruf ein **Stack Frame**: ■ Eingabe ■ lokale Variablen ■ **return** Adresse im Code

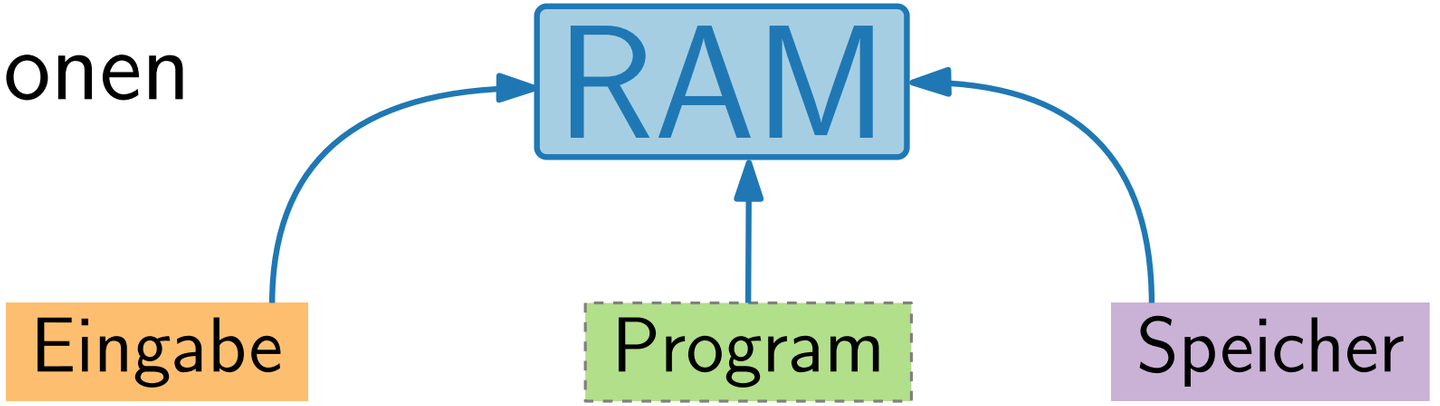
Exkurs: Speicherregionen



Der Speicher wird in 5 Regionen unterteilt:

Code	Maschinencode / kompilierter Code, vom Menschen nicht lesbar
Read-Only Data	Globale, nicht veränderbare Variablen
Global Data	Globale, veränderbare Variablen
Stack	■ Wenig, schneller Speicher ■ Pro aktivem Methodenaufruf ein **Stack Frame**: ■ Eingabe ■ lokale Variablen ■ **return** Adresse im Code ■ Wenn voll → stack**overflow**

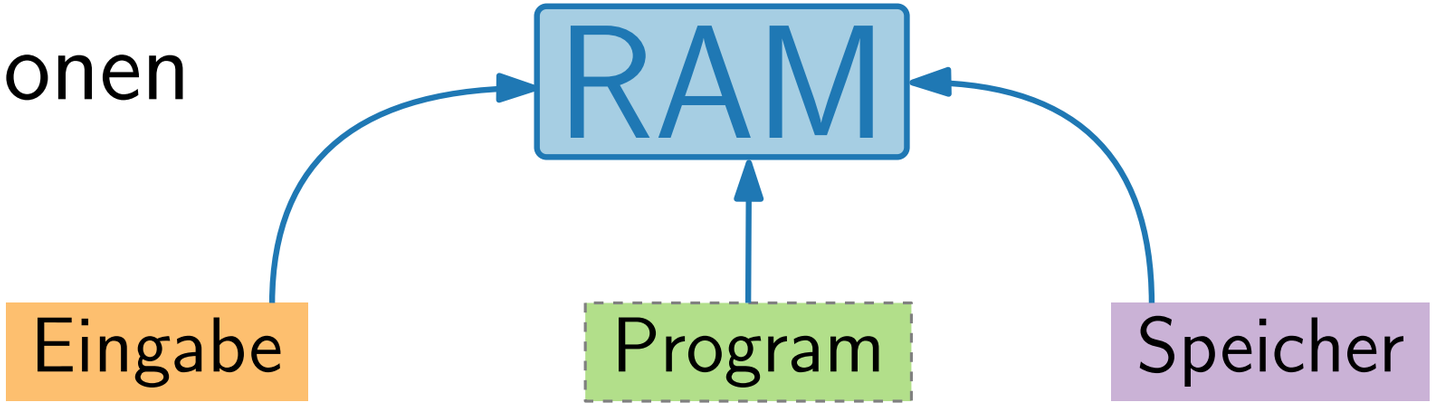
Exkurs: Speicherregionen



Der Speicher wird in 5 Regionen unterteilt:

Code	Maschinencode / kompilierter Code, vom Menschen nicht lesbar
Read-Only Data	Globale, nicht veränderbare Variablen
Global Data	Globale, veränderbare Variablen
Stack	■ Wenig, schneller Speicher ■ Pro aktivem Methodenaufruf ein **Stack Frame**: ■ Eingabe ■ lokale Variablen ■ **return** Adresse im Code ■ Wenn voll → stack**overflow**
Heap	

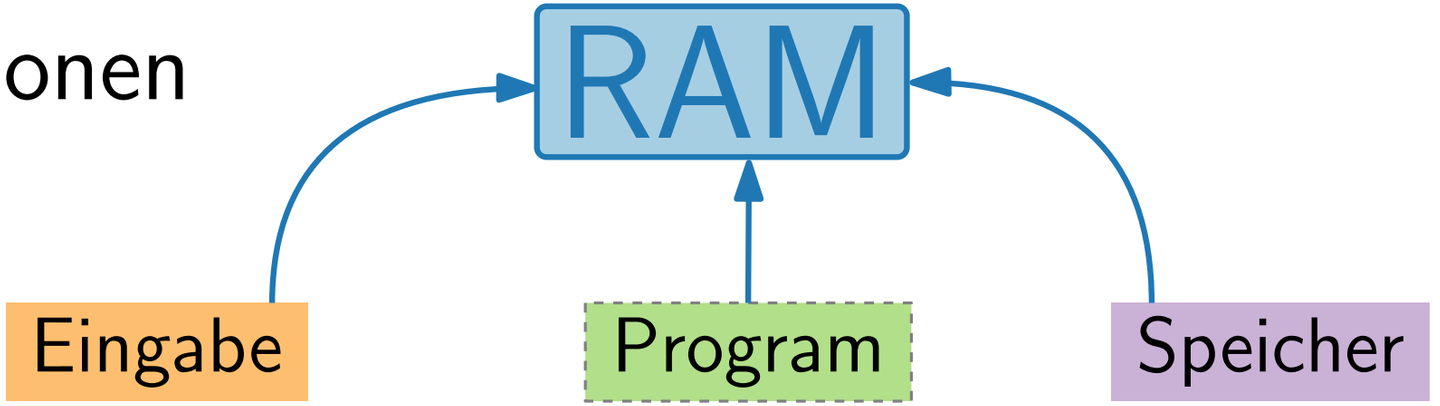
Exkurs: Speicherregionen



Der Speicher wird in 5 Regionen unterteilt:

Code	Maschinencode / kompilierter Code, vom Menschen nicht lesbar
Read-Only Data	Globale, nicht veränderbare Variablen
Global Data	Globale, veränderbare Variablen
Stack	■ Wenig, schneller Speicher ■ Pro aktivem Methodenaufruf ein **Stack Frame**: ■ Eingabe ■ lokale Variablen ■ **return** Adresse im Code ■ Wenn voll → stack**overflow**
Heap	■ Viel, langsamer Speicher

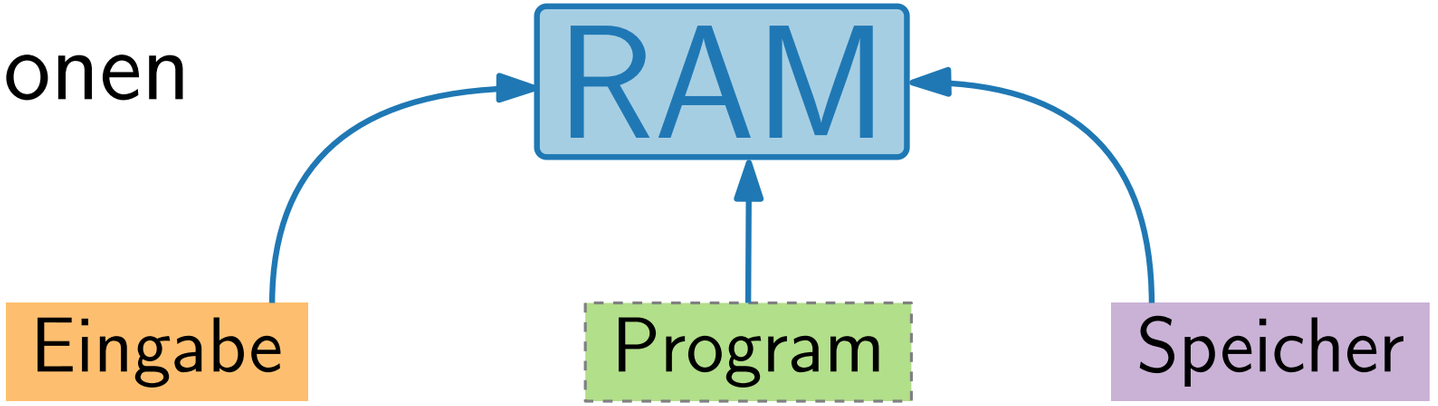
Exkurs: Speicherregionen



Der Speicher wird in 5 Regionen unterteilt:

Code	Maschinencode / kompilierter Code, vom Menschen nicht lesbar
Read-Only Data	Globale, nicht veränderbare Variablen
Global Data	Globale, veränderbare Variablen
Stack	■ Wenig, schneller Speicher ■ Pro aktivem Methodenaufruf ein **Stack Frame**: ■ Eingabe ■ lokale Variablen ■ **return** Adresse im Code ■ Wenn voll → stack**overflow**
Heap	■ Viel, langsamer Speicher ■ Nur über Pointer zugreifbar

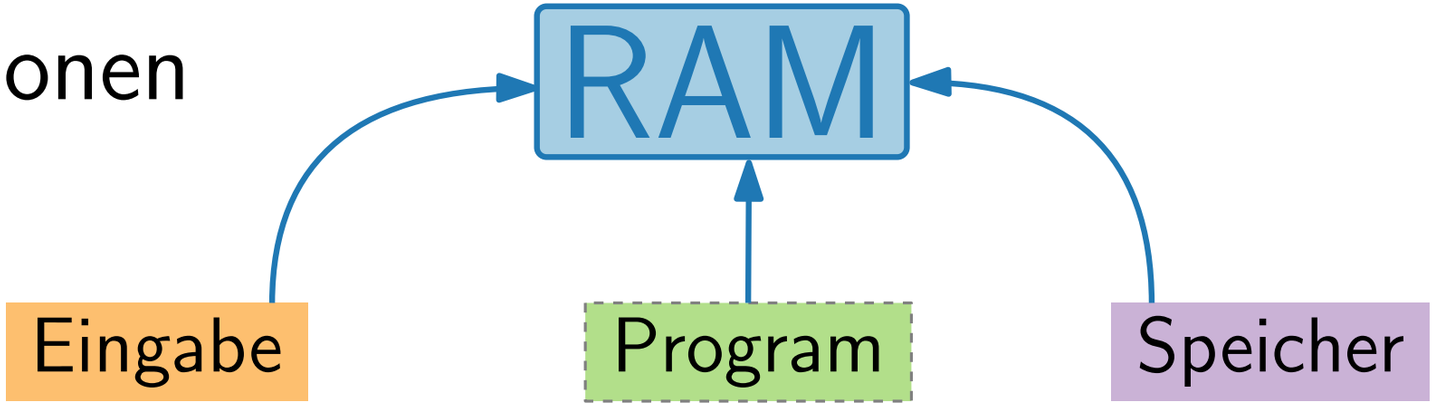
Exkurs: Speicherregionen



Der Speicher wird in 5 Regionen unterteilt:

Code	Maschinencode / kompilierter Code, vom Menschen nicht lesbar
Read-Only Data	Globale, nicht veränderbare Variablen
Global Data	Globale, veränderbare Variablen
Stack	■ Wenig, schneller Speicher ■ Pro aktivem Methodenaufruf ein **Stack Frame**: ■ Eingabe ■ lokale Variablen ■ **return** Adresse im Code ■ Wenn voll → stack**overflow**
Heap	■ Viel, langsamer Speicher ■ Nur über Pointer zugreifbar

Exkurs: Speicherregionen



Der Speicher wird in 5 Regionen unterteilt:

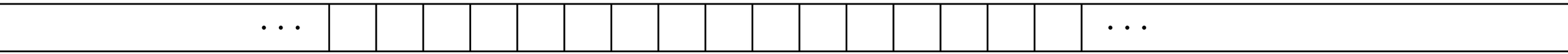
Code	Maschinencode / kompilierter Code, vom Menschen nicht lesbar
Read-Only Data	Globale, nicht veränderbare Variablen
Global Data	Globale, veränderbare Variablen
Stack	■ Wenig, schneller Speicher ■ Pro aktivem Methodenaufruf ein **Stack Frame**: ■ Eingabe ■ lokale Variablen ■ **return** Adresse im Code ■ Wenn voll → stack**overflow**
Heap	■ Viel, langsamer Speicher ■ Nur über Pointer zugreifbar Enthält alles, was nicht eine Zahl oder ein Array ist.

Pointer

Speicher ist ein sehr sehr langes Array aus Blöcken fester Größe (z.B. 64 bit).

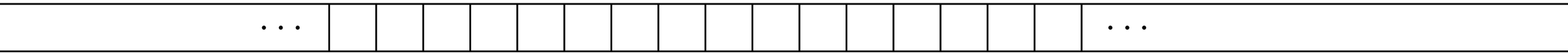
Pointer

Speicher ist ein sehr sehr langes Array aus Blöcken fester Größe (z.B. 64 bit).



Pointer

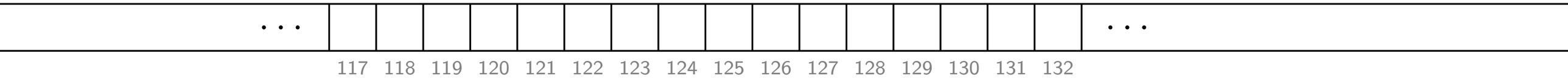
Speicher ist ein sehr sehr langes Array aus Blöcken fester Größe (z.B. 64 bit).



Jeder Block hat eine **Adresse** (der Platz im Array).

Pointer

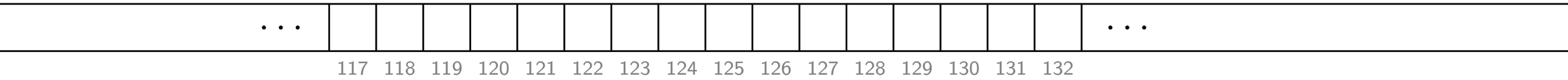
Speicher ist ein sehr sehr langes Array aus Blöcken fester Größe (z.B. 64 bit).



Jeder Block hat eine **Adresse** (der Platz im Array).

Pointer

Speicher ist ein sehr sehr langes Array aus Blöcken fester Größe (z.B. 64 bit).

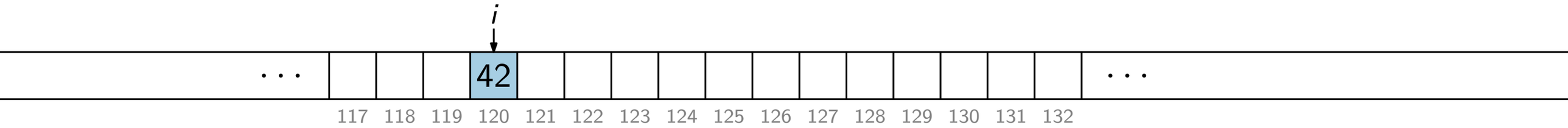


Jeder Block hat eine **Adresse** (der Platz im Array).

Ein Block kann eine Zahl enthalten

Pointer

Speicher ist ein sehr sehr langes Array aus Blöcken fester Größe (z.B. 64 bit).

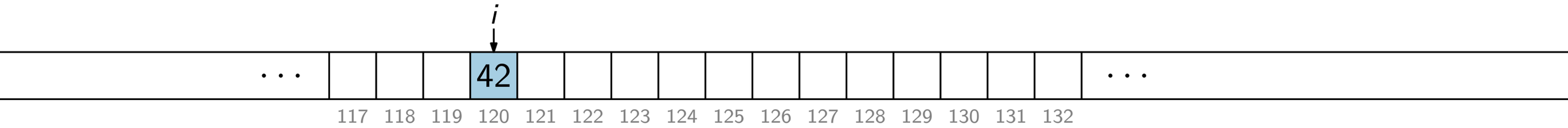


Jeder Block hat eine **Adresse** (der Platz im Array).

Ein Block kann eine Zahl enthalten z.B. $i = 42$

Pointer

Speicher ist ein sehr sehr langes Array aus Blöcken fester Größe (z.B. 64 bit).



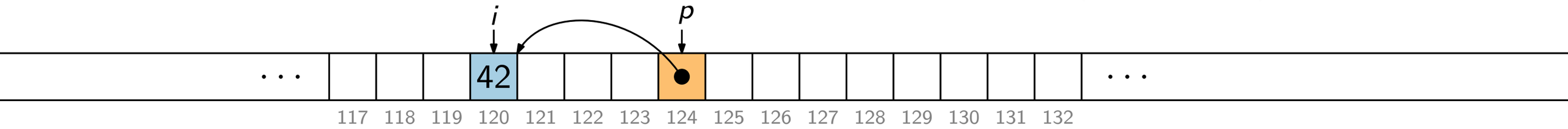
Jeder Block hat eine **Adresse** (der Platz im Array).

Ein Block kann eine Zahl enthalten z.B. $i = 42$

... oder einen Verweis (**Pointer**) auf eine andere Adresse

Pointer

Speicher ist ein sehr sehr langes Array aus Blöcken fester Größe (z.B. 64 bit).



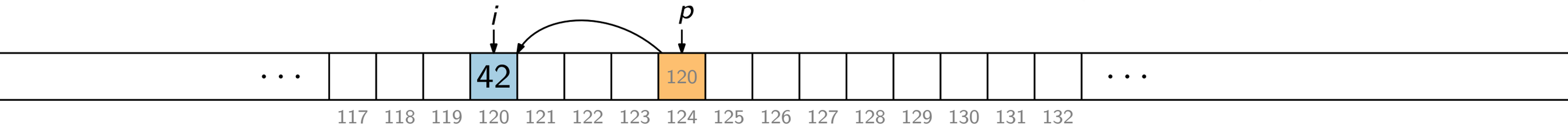
Jeder Block hat eine **Adresse** (der Platz im Array).

Ein Block kann eine Zahl enthalten — z.B. $i = 42$

... oder einen Verweis (**Pointer**) auf eine andere Adresse — z.B. $p =$ Pointer auf i

Pointer

Speicher ist ein sehr sehr langes Array aus Blöcken fester Größe (z.B. 64 bit).



Jeder Block hat eine **Adresse** (der Platz im Array).

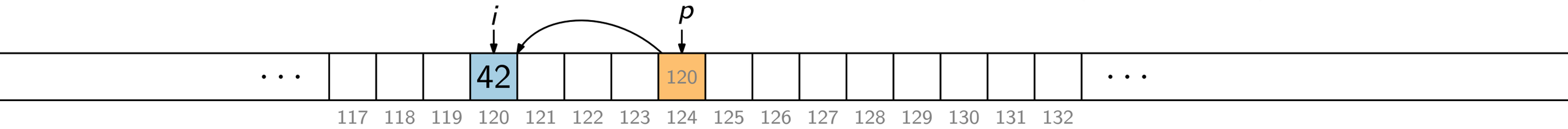
Ein Block kann eine Zahl enthalten — z.B. $i = 42$

... oder einen Verweis (**Pointer**) auf eine andere Adresse — z.B. $p =$ Pointer auf i

p enthält jetzt die Adresse von i

Pointer

Speicher ist ein sehr sehr langes Array aus Blöcken fester Größe (z.B. 64 bit).



Jeder Block hat eine **Adresse** (der Platz im Array).

Ein Block kann eine Zahl enthalten — z.B. $i = 42$

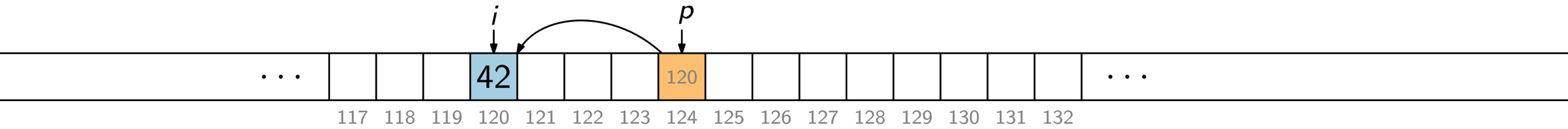
... oder einen Verweis (**Pointer**) auf eine andere Adresse — z.B. $p =$ Pointer auf i

In **C** gibt es dafür 2 Befehle: & und $*$

p enthält jetzt die Adresse von i

Pointer

Speicher ist ein sehr sehr langes Array aus Blöcken fester Größe (z.B. 64 bit).



Jeder Block hat eine **Adresse** (der Platz im Array).

Ein Block kann eine Zahl enthalten — z.B. $i = 42$

... oder einen Verweis (**Pointer**) auf eine andere Adresse — z.B. $p =$ Pointer auf i

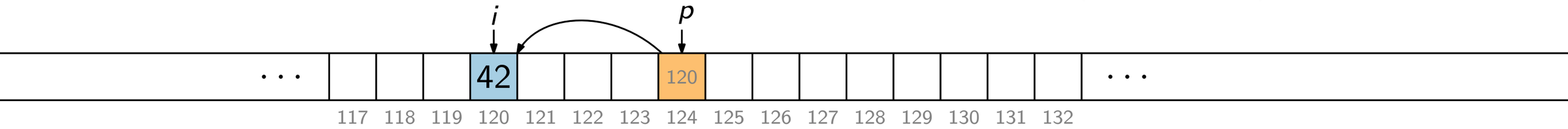
p enthält jetzt die Adresse von i

In **C** gibt es dafür 2 Befehle: & und $*$

- $\&i$: Adresse von i

Pointer

Speicher ist ein sehr sehr langes Array aus Blöcken fester Größe (z.B. 64 bit).



Jeder Block hat eine **Adresse** (der Platz im Array).

Ein Block kann eine Zahl enthalten — z.B. $i = 42$

... oder einen Verweis (**Pointer**) auf eine andere Adresse — z.B. $p =$ Pointer auf i

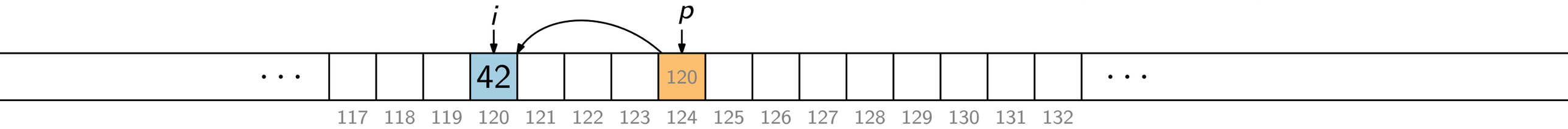
p enthält jetzt die Adresse von i

In **C** gibt es dafür 2 Befehle: & und $*$

- $\&i$: Adresse von i — z.B. $\&i = 120$

Pointer

Speicher ist ein sehr sehr langes Array aus Blöcken fester Größe (z.B. 64 bit).



Jeder Block hat eine **Adresse** (der Platz im Array).

Ein Block kann eine Zahl enthalten — z.B. $i = 42$

... oder einen Verweis (**Pointer**) auf eine andere Adresse — z.B. $p =$ Pointer auf i

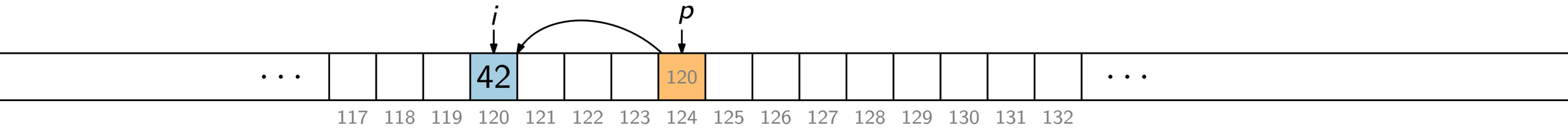
p enthält jetzt die Adresse von i

In **C** gibt es dafür 2 Befehle: & und $*$

- $\&i$: Adresse von i — z.B. $\&i = 120$, $\&p = 124$

Pointer

Speicher ist ein sehr sehr langes Array aus Blöcken fester Größe (z.B. 64 bit).



Jeder Block hat eine **Adresse** (der Platz im Array).

Ein Block kann eine Zahl enthalten — z.B. $i = 42$

... oder einen Verweis (**Pointer**) auf eine andere Adresse — z.B. $p =$ Pointer auf i

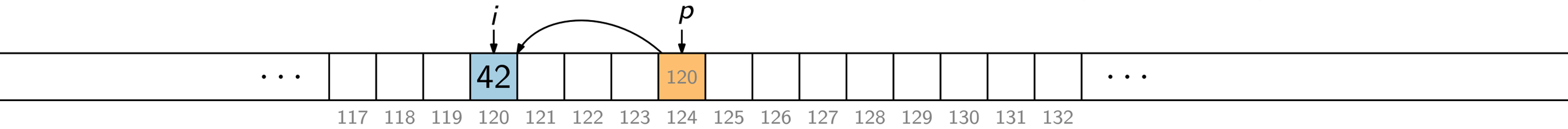
p enthält jetzt die Adresse von i

In **C** gibt es dafür 2 Befehle: & und $*$

- $\&i$: Adresse von i — z.B. $\&i = 120$, $\&p = 124$
- $*p$: Folge dem Pointer zur Adresse, die in p steht

Pointer

Speicher ist ein sehr sehr langes Array aus Blöcken fester Größe (z.B. 64 bit).



Jeder Block hat eine **Adresse** (der Platz im Array).

Ein Block kann eine Zahl enthalten — z.B. $i = 42$

... oder einen Verweis (**Pointer**) auf eine andere Adresse — z.B. $p =$ Pointer auf i

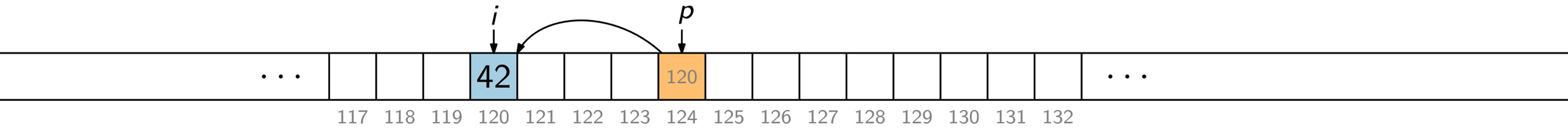
p enthält jetzt die Adresse von i

In **C** gibt es dafür 2 Befehle: & und $*$

- $\&i$: Adresse von i — z.B. $\&i = 120$, $\&p = 124$
- $*p$: Folge dem Pointer zur Adresse, die in p steht — z.B. $*p = 42$

Pointer

Speicher ist ein sehr sehr langes Array aus Blöcken fester Größe (z.B. 64 bit).



Jeder Block hat eine **Adresse** (der Platz im Array).

Ein Block kann eine Zahl enthalten — z.B. $i = 42$

... oder einen Verweis (**Pointer**) auf eine andere Adresse — z.B. $p =$ Pointer auf i

p enthält jetzt die Adresse von i

In **C** gibt es dafür 2 Befehle: & und $*$

- $\&i$: Adresse von i — z.B. $\&i = 120$, $\&p = 124$
- $*p$: Folge dem Pointer zur Adresse, die in p steht — z.B. $*p = 42$

Mit `pythontutor.com` lässt sich das schön visualisieren

Pointer

Speicher ist ei

Jeder Block ha

Ein Block kan

... oder einen

In **C** gibt es da

- $\&i$: Adress
- $*p$: Folge d

```c
#include <stdio.h>

int main(void) {
  int i = 42;
  printf("Wert von i (i) = %d\n", i);

  printf("Adresse von i (&i) = %p\n", &i);

  int *p = &i;  // Pointer p zeigt auf i
  printf("Wert von p = Adresse von i (p) = %p\n", p);
  printf("Wert, auf den p zeigt = Wert von i (*p) = %d\n", *p);

  *p = 7; // i veraendert sich...
  printf("i = %d\n", i);

  int *p2 = p;
  *p2 = 100; // i veraendert sich...
  printf("i = %d\n", i);
}
```

von *i*

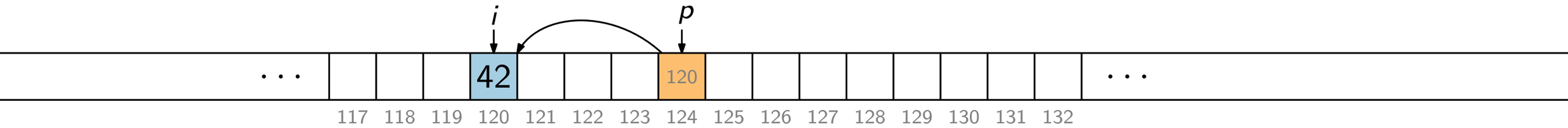
124

algo.uni-trier.de/demos/memvis.html

Mit pythontutor.com lässt sich das schön visualisieren

Pointer

Speicher ist ein sehr sehr langes Array aus Blöcken fester Größe (z.B. 64 bit).



Jeder Block hat eine **Adresse** (der Platz im Array).

Ein Block kann eine Zahl enthalten — z.B. $i = 42$

... oder einen Verweis (**Pointer**) auf eine andere Adresse — z.B. $p =$ Pointer auf i

p enthält jetzt die Adresse von i

In **C** gibt es dafür 2 Befehle: & und $*$

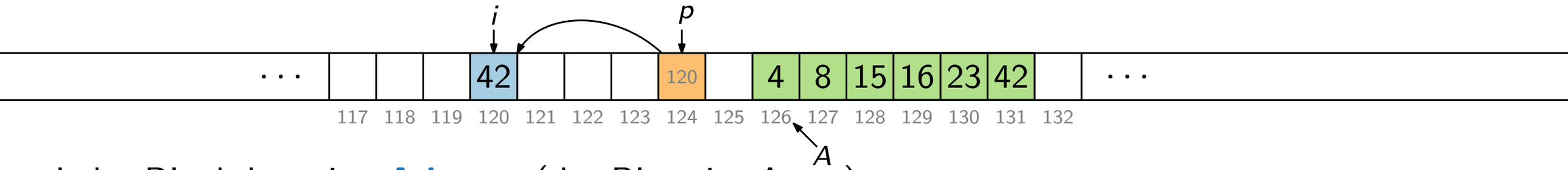
- $\&i$: Adresse von i — z.B. $\&i = 120$, $\&p = 124$
- $*p$: Folge dem Pointer zur Adresse, die in p steht — z.B. $*p = 42$

Wie schaut das bei Arrays aus?

Mit `pythontutor.com` lässt sich das schön visualisieren

Pointer

Speicher ist ein sehr sehr langes Array aus Blöcken fester Größe (z.B. 64 bit).



Jeder Block hat eine **Adresse** (der Platz im Array).

Ein Block kann eine Zahl enthalten — z.B. $i = 42$

... oder einen Verweis (**Pointer**) auf eine andere Adresse — z.B. $p =$ Pointer auf i

p enthält jetzt die Adresse von i

In **C** gibt es dafür 2 Befehle: & und $*$

- $\&i$: Adresse von i — z.B. $\&i = 120$, $\&p = 124$
- $*p$: Folge dem Pointer zur Adresse, die in p steht — z.B. $*p = 42$

Wie schaut das bei Arrays aus? — z.B. $A = [4, 8, 15, 16, 23, 24]$

Mit `pythontutor.com` lässt sich das schön visualisieren

Pointer

Speicher ist ei

Jeder Block ha

Ein Block kan

... oder einen

In **C** gibt es da

- $\&i$: Adress
- $*p$: Folge d

Wie schaut da

```c
#include <stdio.h>

int main(void) {
  int A[6] = {4, 8, 15, 16, 23, 42};

  printf("A: %p\n", A);
  printf("&A: %p\n", &A);
  printf("&A[0]: %p\n", &A[0]);
  printf("&A[2]: %p\n\n", &A[2]);

  printf("A[0]: %d\n", A[0]);
  printf("*A: %d\n\n", *A);

  printf("A[2]: %d\n", A[2]);
  printf("*(A+2): %d\n", *(A+2));
  printf("2[A]: %d\n", 2[A]);
}
```

von i

124

23, 24]

algo.uni-trier.de/demos/memvis.html

Mit pythontutor.com lässt sich das schön visualisieren

Pointer

Speicher ist ei

Jeder Block ha

Ein Block kan

... oder einen

In **C** gibt es da

- $\&i$: Adress
- $*p$: Folge d

Wie schaut da

von i

124

23, 24]

```c
#include <stdio.h>

int main(void) {
  int A[6] = {4, 8, 15, 16, 23, 42};

  printf("A: %p\n", A);
  printf("&A: %p\n", &A);
  printf("&A[0]: %p\n", &A[0]);
  printf("&A[2]: %p\n\n", &A[2]);

  printf("A[0]: %d\n", A[0]);
  printf("*A: %d\n\n", *A);

  printf("A[2]: %d\n", A[2]);
  printf("*(A+2): %d\n", *(A+2));
  printf("2[A]: %d\n", 2[A]);
}
```

output:

A :

algo.uni-trier.de/demos/memvis.html

Mit pythontutor.com lässt sich das schön visualisieren

Pointer

Speicher ist ei

Jeder Block ha

Ein Block kan

... oder einen

In **C** gibt es da

- $\&i$: Adress
- $*p$: Folge d

Wie schaut da

```c
#include <stdio.h>

int main(void) {
  int A[6] = {4, 8, 15, 16, 23, 42};

  printf("A: %p\n", A);
  printf("&A: %p\n", &A);
  printf("&A[0]: %p\n", &A[0]);
  printf("&A[2]: %p\n\n", &A[2]);

  printf("A[0]: %d\n", A[0]);
  printf("*A: %d\n\n", *A);

  printf("A[2]: %d\n", A[2]);
  printf("*(A+2): %d\n", *(A+2));
  printf("2[A]: %d\n", 2[A]);
}
```

output:

```
A: 126
```

von i

124

23, 24]

algo.uni-trier.de/demos/memvis.html

Mit pythontutor.com lässt sich das schön visualisieren

Pointer

Speicher ist ei

Jeder Block ha

Ein Block kan

... oder einen

In **C** gibt es da

- $\&i$: Adress
- $*p$: Folge d

Wie schaut da

von i

124

23, 24]

```c
#include <stdio.h>

int main(void) {
  int A[6] = {4, 8, 15, 16, 23, 42};

  printf("A: %p\n", A);
  printf("&A: %p\n", &A);
  printf("&A[0]: %p\n", &A[0]);
  printf("&A[2]: %p\n\n", &A[2]);

  printf("A[0]: %d\n", A[0]);
  printf("*A: %d\n\n", *A);

  printf("A[2]: %d\n", A[2]);
  printf("*(A+2): %d\n", *(A+2));
  printf("2[A]: %d\n", 2[A]);
}
```

output:

```
A: 126
&A:
```

algo.uni-trier.de/demos/memvis.html

Mit pythontutor.com lässt sich das schön visualisieren

Pointer

Speicher ist ei

Jeder Block ha

Ein Block kan

... oder einen

In **C** gibt es da

- $\&i$: Adress
- $*p$: Folge d

Wie schaut da

von i

124

23, 24]

```c
#include <stdio.h>

int main(void) {
  int A[6] = {4, 8, 15, 16, 23, 42};

  printf("A: %p\n", A);
  printf("&A: %p\n", &A);
  printf("&A[0]: %p\n", &A[0]);
  printf("&A[2]: %p\n\n", &A[2]);

  printf("A[0]: %d\n", A[0]);
  printf("*A: %d\n\n", *A);

  printf("A[2]: %d\n", A[2]);
  printf("*(A+2): %d\n", *(A+2));
  printf("2[A]: %d\n", 2[A]);
}
```

output:

```
A: 126
&A: 126
```

algo.uni-trier.de/demos/memvis.html

Mit pythontutor.com lässt sich das schön visualisieren

Pointer

Speicher ist ei

Jeder Block ha

Ein Block kan

... oder einen

In **C** gibt es da

- $\&i$: Adress
- $*p$: Folge d

Wie schaut da

```c
#include <stdio.h>

int main(void) {
  int A[6] = {4, 8, 15, 16, 23, 42};

  printf("A: %p\n", A);
  printf("&A: %p\n", &A);
  printf("&A[0]: %p\n", &A[0]);
  printf("&A[2]: %p\n\n", &A[2]);

  printf("A[0]: %d\n", A[0]);
  printf("*A: %d\n\n", *A);

  printf("A[2]: %d\n", A[2]);
  printf("*(A+2): %d\n", *(A+2));
  printf("2[A]: %d\n", 2[A]);
}
```

output:

```
A: 126
&A: 126
&A[0]:
```

von i

124

23, 24]

algo.uni-trier.de/demos/memvis.html

Mit pythontutor.com lässt sich das schön visualisieren

Pointer

Speicher ist ei

Jeder Block ha

Ein Block kan

... oder einen

In **C** gibt es da

- $\&i$: Adress
- $*p$: Folge d

Wie schaut da

```c
#include <stdio.h>

int main(void) {
  int A[6] = {4, 8, 15, 16, 23, 42};

  printf("A: %p\n", A);
  printf("&A: %p\n", &A);
  printf("&A[0]: %p\n", &A[0]);
  printf("&A[2]: %p\n\n", &A[2]);

  printf("A[0]: %d\n", A[0]);
  printf("*A: %d\n\n", *A);

  printf("A[2]: %d\n", A[2]);
  printf("*(A+2): %d\n", *(A+2));
  printf("2[A]: %d\n", 2[A]);
}
```

output:

```
A: 126
&A: 126
&A[0]: 126
```

von i

124

23, 24]

algo.uni-trier.de/demos/memvis.html

Mit pythontutor.com lässt sich das schön visualisieren

Pointer

Speicher ist ei

Jeder Block ha

Ein Block kan

... oder einen

In **C** gibt es da

- $\&i$: Adress
- $*p$: Folge

Wie schaut da

von i

124

23, 24]

```c
#include <stdio.h>

int main(void) {
  int A[6] = {4, 8, 15, 16, 23, 42};

  printf("A: %p\n", A);
  printf("&A: %p\n", &A);
  printf("&A[0]: %p\n", &A[0]);
  printf("&A[2]: %p\n\n", &A[2]);

  printf("A[0]: %d\n", A[0]);
  printf("*A: %d\n\n", *A);

  printf("A[2]: %d\n", A[2]);
  printf("*(A+2): %d\n", *(A+2));
  printf("2[A]: %d\n", 2[A]);
}
```

output:

```
A: 126
&A: 126
&A[0]: 126
&A[2]:
```

algo.uni-trier.de/demos/memvis.html

Mit pythontutor.com lässt sich das schön visualisieren

Pointer

Speicher ist ei

Jeder Block ha

Ein Block kan

... oder einen

In **C** gibt es da

- $\&i$: Adress
- $*p$: Folge d

Wie schaut da

```c
#include <stdio.h>

int main(void) {
  int A[6] = {4, 8, 15, 16, 23, 42};

  printf("A: %p\n", A);
  printf("&A: %p\n", &A);
  printf("&A[0]: %p\n", &A[0]);
  printf("&A[2]: %p\n\n", &A[2]);

  printf("A[0]: %d\n", A[0]);
  printf("*A: %d\n\n", *A);

  printf("A[2]: %d\n", A[2]);
  printf("*(A+2): %d\n", *(A+2));
  printf("2[A]: %d\n", 2[A]);
}
```

output:

```
A: 126
&A: 126
&A[0]: 126
&A[2]: 128
```

von i

124

23, 24]

algo.uni-trier.de/demos/memvis.html

Mit pythontutor.com lässt sich das schön visualisieren

Pointer

Speicher ist ei

Jeder Block ha

Ein Block kan

... oder einen

In **C** gibt es da

- $\&i$: Adress
- $*p$: Folge d

Wie schaut da

```c
#include <stdio.h>

int main(void) {
  int A[6] = {4, 8, 15, 16, 23, 42};

  printf("A: %p\n", A);
  printf("&A: %p\n", &A);
  printf("&A[0]: %p\n", &A[0]);
  printf("&A[2]: %p\n\n", &A[2]);

  printf("A[0]: %d\n", A[0]);
  printf("*A: %d\n\n", *A);

  printf("A[2]: %d\n", A[2]);
  printf("*(A+2): %d\n", *(A+2));
  printf("2[A]: %d\n", 2[A]);
}
```

output:

```
A: 126
&A: 126
&A[0]: 126
&A[2]: 128

A[0]:
```

von i

124

23, 24]

algo.uni-trier.de/demos/memvis.html

Mit pythontutor.com lässt sich das schön visualisieren

Pointer

Speicher ist ei

Jeder Block ha

Ein Block kan

... oder einen

In **C** gibt es da

- $\&i$: Adress
- $*p$: Folge d

Wie schaut da

```c
#include <stdio.h>

int main(void) {
  int A[6] = {4, 8, 15, 16, 23, 42};

  printf("A: %p\n", A);
  printf("&A: %p\n", &A);
  printf("&A[0]: %p\n", &A[0]);
  printf("&A[2]: %p\n\n", &A[2]);

  printf("A[0]: %d\n", A[0]);
  printf("*A: %d\n\n", *A);

  printf("A[2]: %d\n", A[2]);
  printf("*(A+2): %d\n", *(A+2));
  printf("2[A]: %d\n", 2[A]);
}
```

output:

```
A: 126
&A: 126
&A[0]: 126
&A[2]: 128

A[0]: 4
```

von i

124

23, 24]

algo.uni-trier.de/demos/memvis.html

Mit pythontutor.com lässt sich das schön visualisieren

Pointer

Speicher ist ei

Jeder Block ha

Ein Block kan

... oder einen

In **C** gibt es da

- $\&i$: Adress
- $*p$: Folge d

Wie schaut da

```c
#include <stdio.h>

int main(void) {
  int A[6] = {4, 8, 15, 16, 23, 42};

  printf("A: %p\n", A);
  printf("&A: %p\n", &A);
  printf("&A[0]: %p\n", &A[0]);
  printf("&A[2]: %p\n\n", &A[2]);

  printf("A[0]: %d\n", A[0]);
  printf("*A: %d\n\n", *A);

  printf("A[2]: %d\n", A[2]);
  printf("*(A+2): %d\n", *(A+2));
  printf("2[A]: %d\n", 2[A]);
}
```

output:

```
A: 126
&A: 126
&A[0]: 126
&A[2]: 128

A[0]: 4
*A:
```

von i

124

23, 24]

algo.uni-trier.de/demos/memvis.html

Mit pythontutor.com lässt sich das schön visualisieren

Pointer

Speicher ist ei

Jeder Block ha

Ein Block kan

... oder einen

In **C** gibt es da

- $\&i$: Adress
- $*p$: Folge d

Wie schaut da

```c
#include <stdio.h>

int main(void) {
  int A[6] = {4, 8, 15, 16, 23, 42};

  printf("A: %p\n", A);
  printf("&A: %p\n", &A);
  printf("&A[0]: %p\n", &A[0]);
  printf("&A[2]: %p\n\n", &A[2]);

  printf("A[0]: %d\n", A[0]);
  printf("*A: %d\n\n", *A);

  printf("A[2]: %d\n", A[2]);
  printf("*(A+2): %d\n", *(A+2));
  printf("2[A]: %d\n", 2[A]);
}
```

output:

```
A: 126
&A: 126
&A[0]: 126
&A[2]: 128

A[0]: 4
*A: 4
```

von i

124

23, 24]

algo.uni-trier.de/demos/memvis.html

Mit pythontutor.com lässt sich das schön visualisieren

Pointer

Speicher ist ei

Jeder Block ha

Ein Block kan

... oder einen

In **C** gibt es da

- $\&i$: Adress
- $*p$: Folge

Wie schaut da

von i

124

23, 24]

```c
#include <stdio.h>

int main(void) {
  int A[6] = {4, 8, 15, 16, 23, 42};

  printf("A: %p\n", A);
  printf("&A: %p\n", &A);
  printf("&A[0]: %p\n", &A[0]);
  printf("&A[2]: %p\n\n", &A[2]);

  printf("A[0]: %d\n", A[0]);
  printf("*A: %d\n\n", *A);

  printf("A[2]: %d\n", A[2]);
  printf("*(A+2): %d\n", *(A+2));
  printf("2[A]: %d\n", 2[A]);
}
```

output:

```
A: 126
&A: 126
&A[0]: 126
&A[2]: 128

A[0]: 4
*A: 4

A[2]:
```

algo.uni-trier.de/demos/memvis.html

Mit pythontutor.com lässt sich das schön visualisieren

Pointer

Speicher ist ei

Jeder Block ha

Ein Block kan

... oder einen

In **C** gibt es da

- $\&i$: Adress
- $*p$: Folge d

Wie schaut da

```c
#include <stdio.h>

int main(void) {
  int A[6] = {4, 8, 15, 16, 23, 42};

  printf("A: %p\n", A);
  printf("&A: %p\n", &A);
  printf("&A[0]: %p\n", &A[0]);
  printf("&A[2]: %p\n\n", &A[2]);

  printf("A[0]: %d\n", A[0]);
  printf("*A: %d\n\n", *A);

  printf("A[2]: %d\n", A[2]);
  printf("*(A+2): %d\n", *(A+2));
  printf("2[A]: %d\n", 2[A]);
}
```

output:

```
A: 126
&A: 126
&A[0]: 126
&A[2]: 128

A[0]: 4
*A: 4

A[2]: 15
```

von i

124

23, 24]

algo.uni-trier.de/demos/memvis.html

Mit pythontutor.com lässt sich das schön visualisieren

Pointer

Speicher ist ei

Jeder Block ha

Ein Block kan

... oder einen

In **C** gibt es da

- $\&i$: Adress
- $*p$: Folge d

Wie schaut da

```c
#include <stdio.h>

int main(void) {
  int A[6] = {4, 8, 15, 16, 23, 42};

  printf("A: %p\n", A);
  printf("&A: %p\n", &A);
  printf("&A[0]: %p\n", &A[0]);
  printf("&A[2]: %p\n\n", &A[2]);

  printf("A[0]: %d\n", A[0]);
  printf("*A: %d\n\n", *A);

  printf("A[2]: %d\n", A[2]);
  printf("*(A+2): %d\n", *(A+2));
  printf("2[A]: %d\n", 2[A]);
}
```

output:

```
A: 126
&A: 126
&A[0]: 126
&A[2]: 128

A[0]: 4
*A: 4

A[2]: 15
*(A+2):
```

von i

124

23, 24]

algo.uni-trier.de/demos/memvis.html

Mit `pythontutor.com` lässt sich das schön visualisieren

Pointer

Speicher ist ei

Jeder Block ha

Ein Block kan

... oder einen

In **C** gibt es da

- $\&i$: Adress
- $*p$: Folge d

Wie schaut da

```c
#include <stdio.h>

int main(void) {
  int A[6] = {4, 8, 15, 16, 23, 42};

  printf("A: %p\n", A);
  printf("&A: %p\n", &A);
  printf("&A[0]: %p\n", &A[0]);
  printf("&A[2]: %p\n\n", &A[2]);

  printf("A[0]: %d\n", A[0]);
  printf("*A: %d\n\n", *A);

  printf("A[2]: %d\n", A[2]);
  printf("*(A+2): %d\n", *(A+2));
  printf("2[A]: %d\n", 2[A]);
}
```

output:

```
A: 126
&A: 126
&A[0]: 126
&A[2]: 128

A[0]: 4
*A: 4

A[2]: 15
*(A+2): 15
```

von i

124

23, 24]

algo.uni-trier.de/demos/memvis.html

Mit pythontutor.com lässt sich das schön visualisieren

Pointer

Speicher ist ei

Jeder Block ha

Ein Block kan

... oder einen

In **C** gibt es da

- $\&i$: Adress
- $*p$: Folge

Wie schaut da

von i

124

23, 24]

```c
#include <stdio.h>

int main(void) {
  int A[6] = {4, 8, 15, 16, 23, 42};

  printf("A: %p\n", A);
  printf("&A: %p\n", &A);
  printf("&A[0]: %p\n", &A[0]);
  printf("&A[2]: %p\n\n", &A[2]);

  printf("A[0]: %d\n", A[0]);
  printf("*A: %d\n\n", *A);

  printf("A[2]: %d\n", A[2]);
  printf("*(A+2): %d\n", *(A+2));
  printf("2[A]: %d\n", 2[A]);
}
```

```
output:
A: 126
&A: 126
&A[0]: 126
&A[2]: 128

A[0]: 4
*A: 4

A[2]: 15
*(A+2): 15
2[A]:
```

algo.uni-trier.de/demos/memvis.html

Mit pythontutor.com lässt sich das schön visualisieren

Pointer

Speicher ist ei

Jeder Block ha

Ein Block kan

... oder einen

In **C** gibt es da

- $\&i$: Adress
- $*p$: Folge

Wie schaut da

```c
#include <stdio.h>

int main(void) {
  int A[6] = {4, 8, 15, 16, 23, 42};

  printf("A: %p\n", A);
  printf("&A: %p\n", &A);
  printf("&A[0]: %p\n", &A[0]);
  printf("&A[2]: %p\n\n", &A[2]);

  printf("A[0]: %d\n", A[0]);
  printf("*A: %d\n\n", *A);

  printf("A[2]: %d\n", A[2]);
  printf("*(A+2): %d\n", *(A+2));
  printf("2[A]: %d\n", 2[A]);
}
```

```
output:
A: 126
&A: 126
&A[0]: 126
&A[2]: 128

A[0]: 4
*A: 4

A[2]: 15
*(A+2): 15
2[A]: 15
```

von i

124

23, 24]

algo.uni-trier.de/demos/memvis.html

Mit pythontutor.com lässt sich das schön visualisieren

Zurück zu dynamischen Mengen

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	Füge Element (k, v) ein
DELETE(ptr x)	Entferne Element x
ptr SEARCH(key k)	Suche Element x mit $x.key = k$
ptr MINIMUM()	Suche Element x mit kleinstem Schlüssel $x.key$
ptr MAXIMUM()	Suche Element x mit größtem Schlüssel $x.key$
ptr PREDECESSOR(ptr x)	Suche Element y mit größtem Schlüssel $y.key$ aus $\{(k, v) \mid k < x.key\}$
ptr SUCCESSOR(ptr x)	Suche Element y mit kleinstem Schlüssel $y.key$ aus $\{(k, v) \mid k > x.key\}$

Pointer

Zurück zu dynamischen Mengen

Abstrakter Datentyp: Dynamische Menge

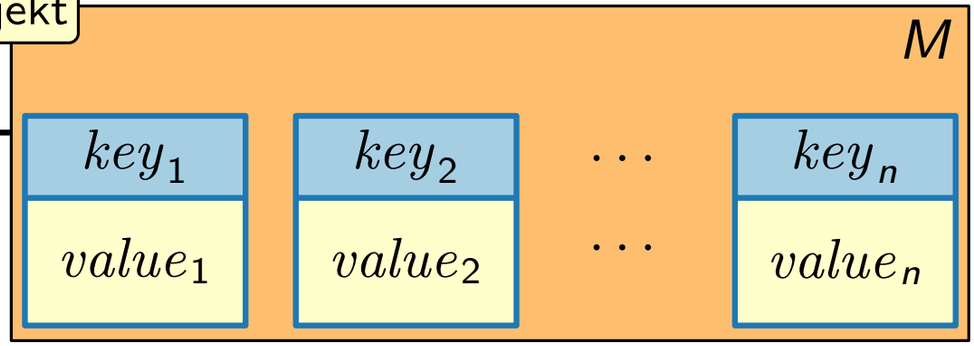
verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	Füge Element (k, v) ein
DELETE(ptr x)	Entferne Element x
ptr SEARCH(key k)	Suche Element x mit $x.key = k$
ptr MINIMUM()	Suche Element x mit kleinstem Schlüssel $x.key$
ptr MAXIMUM()	Suche Element x mit größtem Schlüssel $x.key$
ptr PREDECESSOR(ptr x)	Suche Element y mit größtem Schlüssel $y.key$ aus $\{(k, v) \mid k < x.key\}$
ptr SUCCESSOR(ptr x)	Suche Element y mit kleinstem Schlüssel $y.key$ aus $\{(k, v) \mid k > x.key\}$

Pointer



Zurück zu dynamischen Mengen

Abstrakter Datentyp: Dynamische Menge

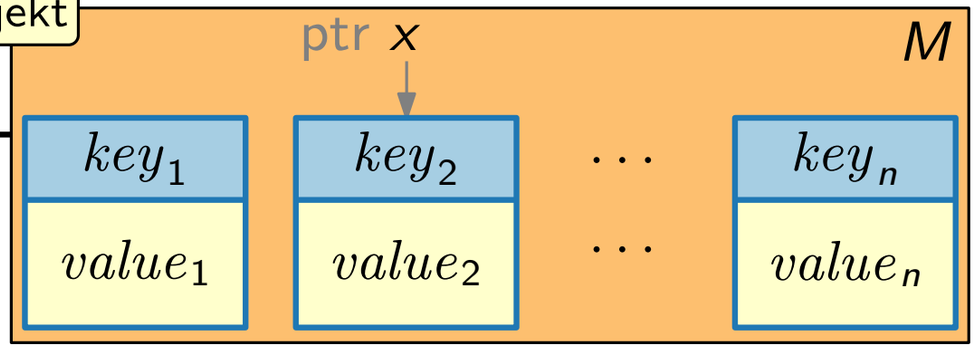
verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

eindeutige Zahl

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	Füge Element (k, v) ein
DELETE(ptr x)	Entferne Element x
ptr SEARCH(key k)	Suche Element x mit $x.key = k$
ptr MINIMUM()	Suche Element x mit kleinstem Schlüssel $x.key$
ptr MAXIMUM()	Suche Element x mit größtem Schlüssel $x.key$
ptr PREDECESSOR(ptr x)	Suche Element y mit größtem Schlüssel $y.key$ aus $\{(k, v) \mid k < x.key\}$
ptr SUCCESSOR(ptr x)	Suche Element y mit kleinstem Schlüssel $y.key$ aus $\{(k, v) \mid k > x.key\}$

Pointer



Zurück zu dynamischen Mengen

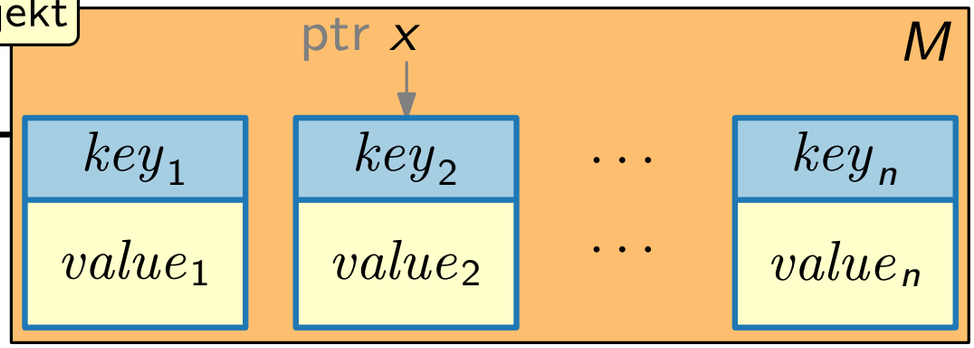
Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer



Zurück zu dynamischen Mengen

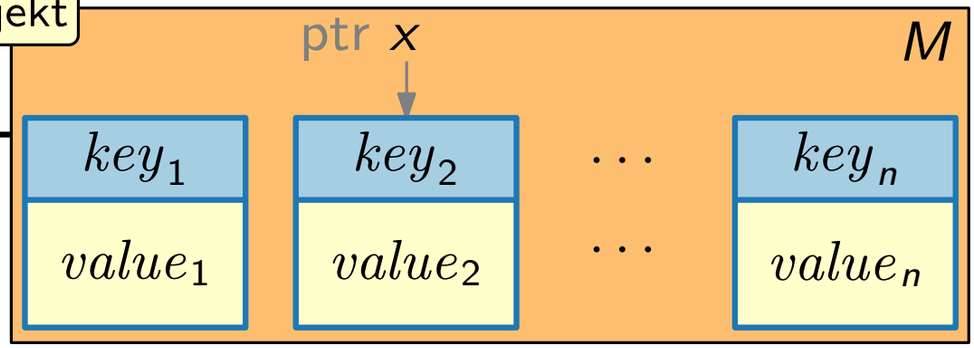
Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	$x = (k, v)$
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer



Zurück zu dynamischen Mengen

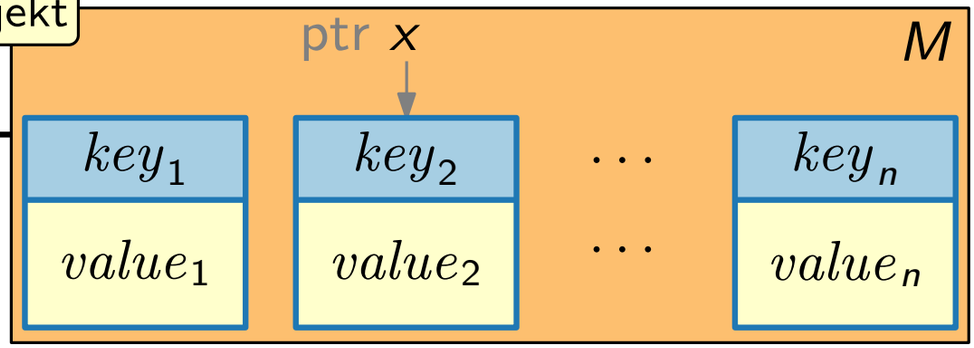
Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	$x = (k, v)$ $M = M \cup \{x\}$
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer



Zurück zu dynamischen Mengen

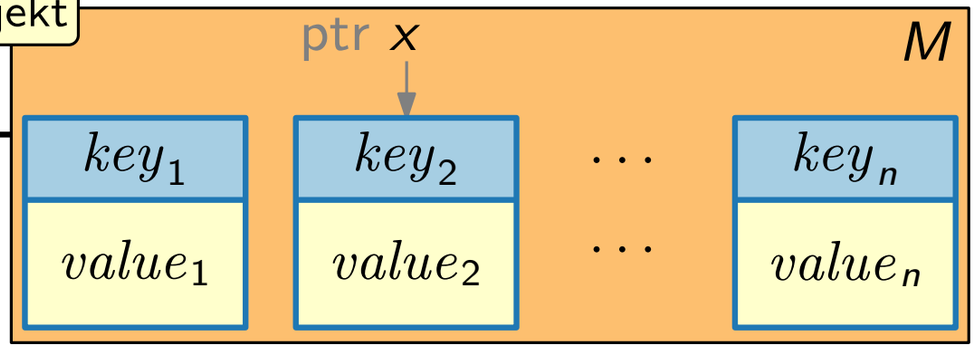
Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	$x = (k, v)$ $M = M \cup \{x\}$ **return** x
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer



Zurück zu dynamischen Mengen

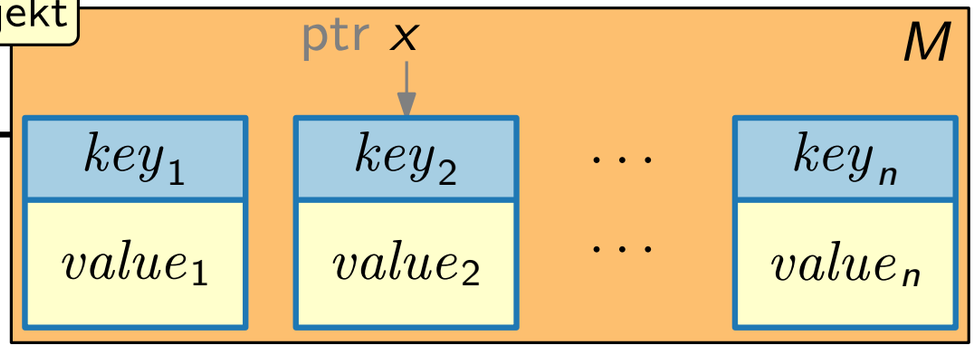
Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer



Zurück zu dynamischen Mengen

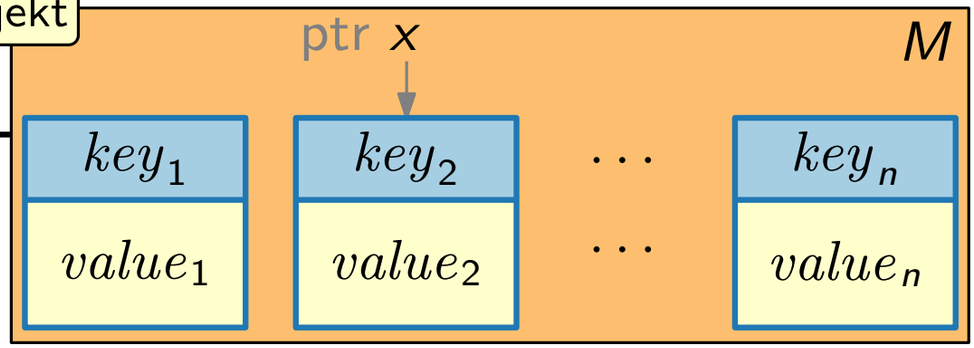
Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	$M = M \setminus \{x\}$
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer



Zurück zu dynamischen Mengen

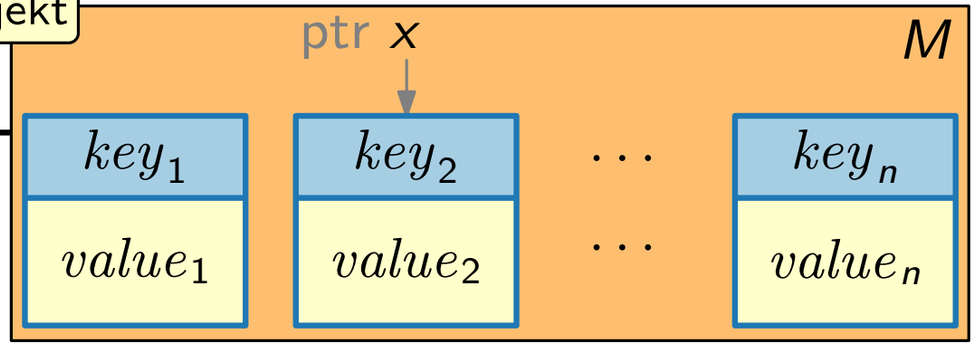
Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer



Zurück zu dynamischen Mengen

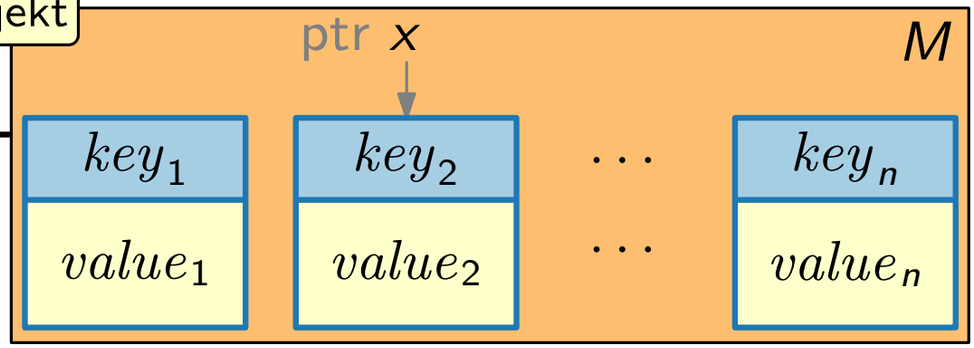
Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	**if** M enthält Element $x = (k, v)$ **then**
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer



Zurück zu dynamischen Mengen

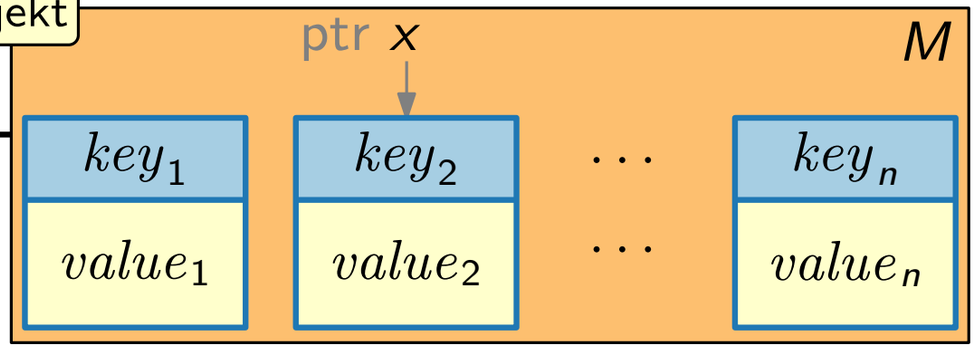
Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	**if** M enthält Element $x = (k, v)$ **then** **return** x
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer



Zurück zu dynamischen Mengen

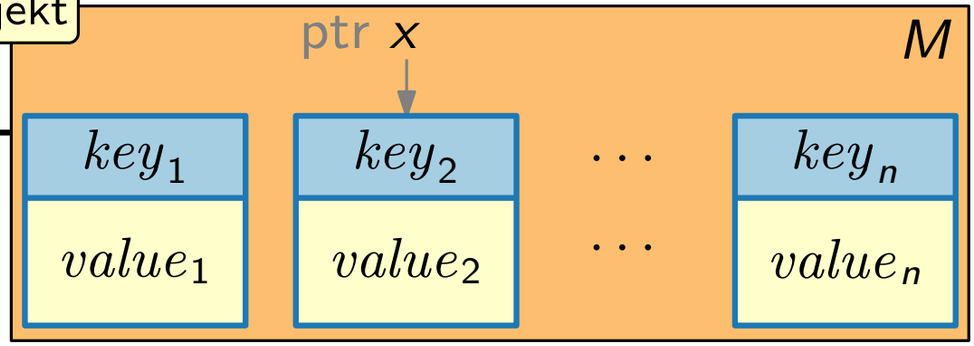
Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	**if** M enthält Element $x = (k, v)$ **then** return x **else** return *nil*
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer



Zurück zu dynamischen Mengen

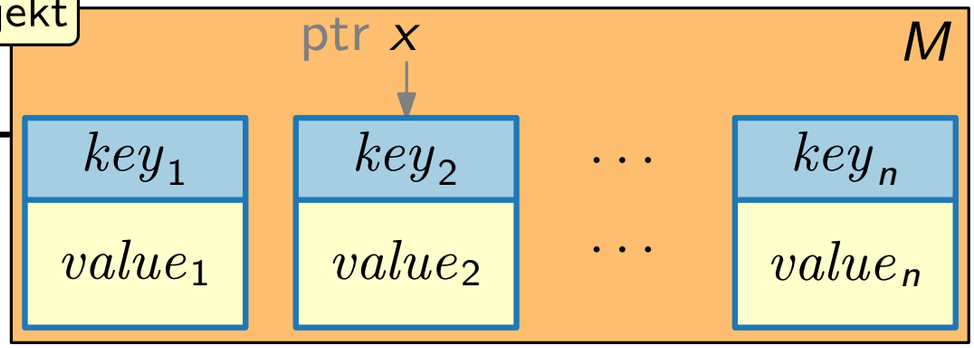
Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer

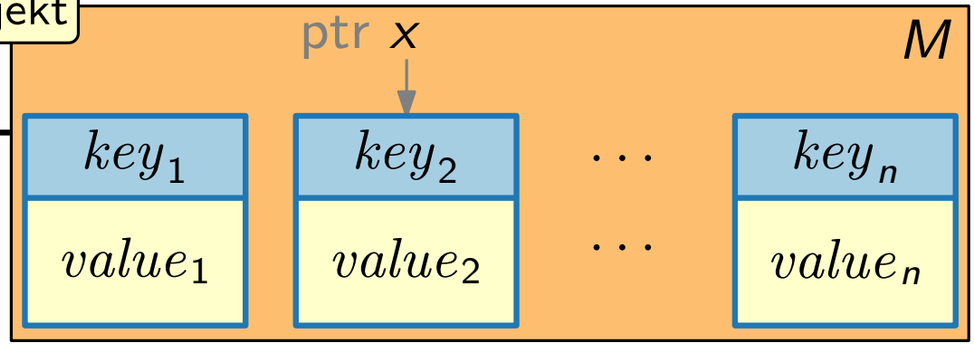


Zurück zu dynamischen Mengen

Abstrakter Datentyp: **Dynamische Menge**

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt



Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	**if** $M = \emptyset$ **then return** *nil*
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

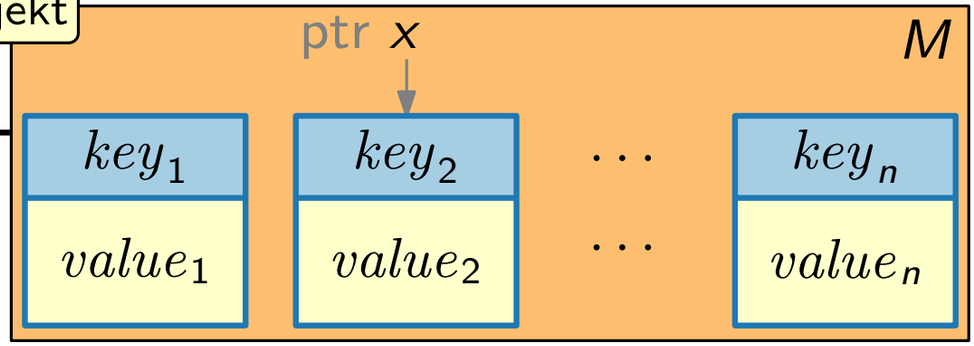
Pointer

Zurück zu dynamischen Mengen

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt



Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	**if** $M = \emptyset$ **then return** *nil* $k' = \min_{(k,v) \in M} k$
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

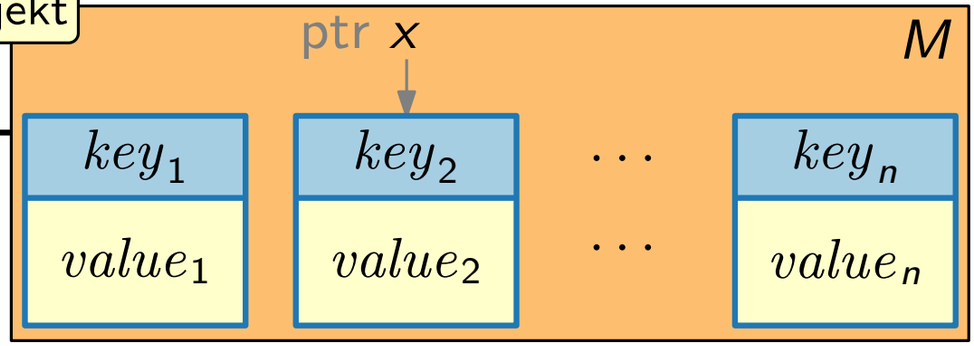
Pointer

Zurück zu dynamischen Mengen

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt



Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	**if** $M = \emptyset$ **then return** *nil* $k' = \min_{(k,v) \in M} k$ $x' = (k', v) \in M$
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

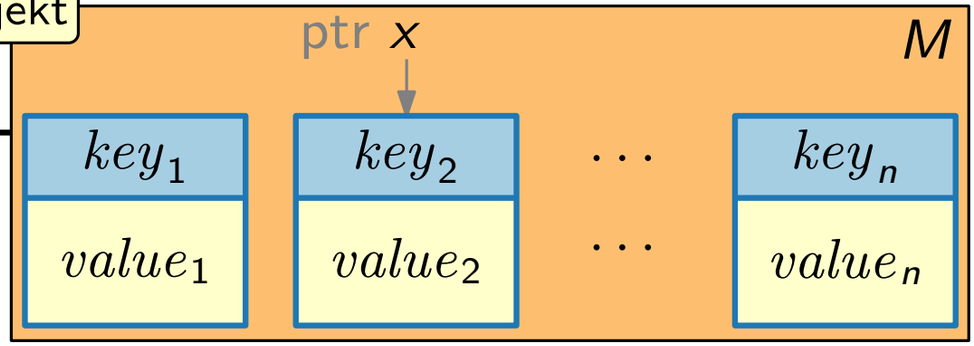
Pointer

Zurück zu dynamischen Mengen

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt



Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	**if** $M = \emptyset$ **then return** *nil* $k' = \min_{(k,v) \in M} k$ $x' = (k', v) \in M$ **return** x'
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

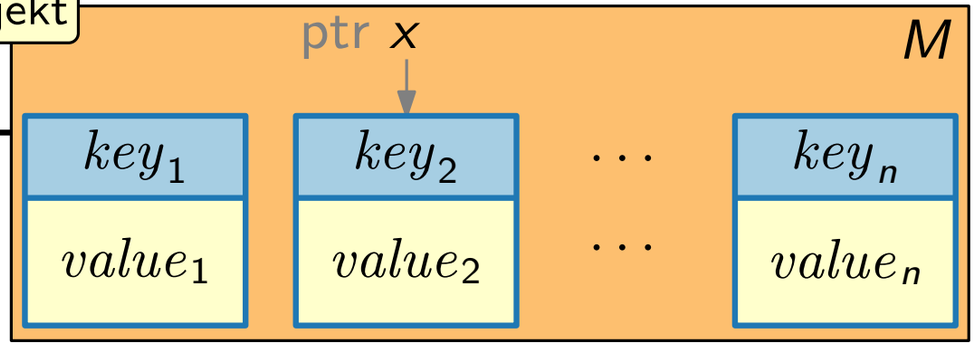
Pointer

Zurück zu dynamischen Mengen

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt



Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	**if** $M = \emptyset$ **then return** *nil* $k' = \max_{(k,v) \in M} k$ $x' = (k', v) \in M$ **return** x'
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

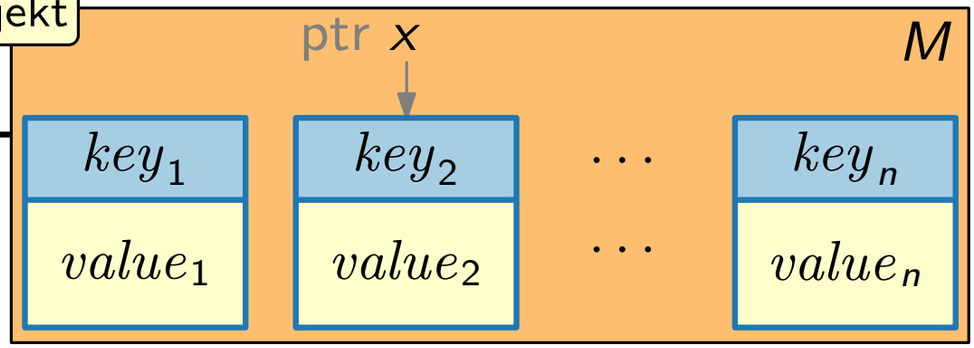
Pointer

Zurück zu dynamischen Mengen

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt



Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	$M' = \{(k, v) \in M \mid k < x.key\}$
ptr SUCCESSOR(ptr x)	

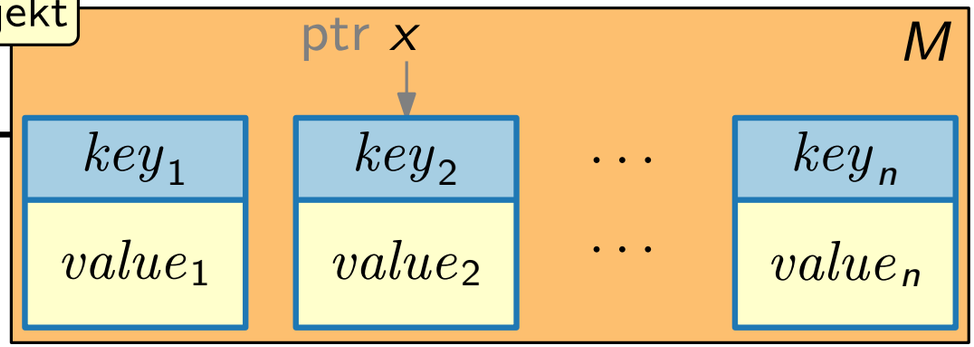
Pointer

Zurück zu dynamischen Mengen

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt



Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	$M' = \{(k, v) \in M \mid k < x.key\}$ **return** M'.MAXIMUM()
ptr SUCCESSOR(ptr x)	

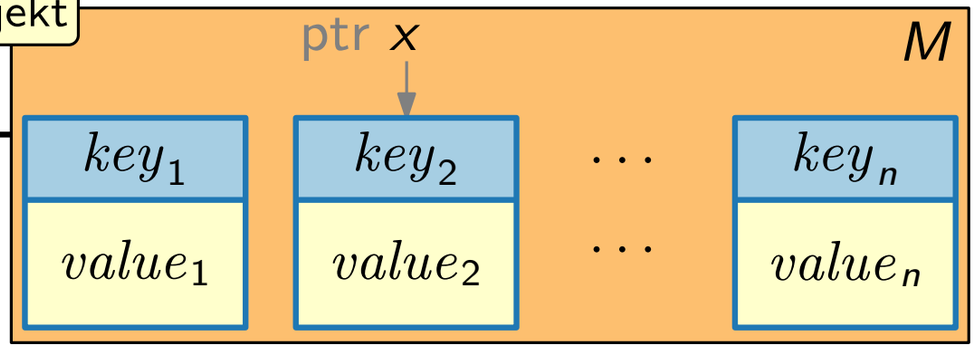
Pointer

Zurück zu dynamischen Mengen

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt



Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	$M' = \{(k, v) \in M \mid k > x.key\}$ **return** M'.MINIMUM()

Pointer

Zurück zu dynamischen Mengen

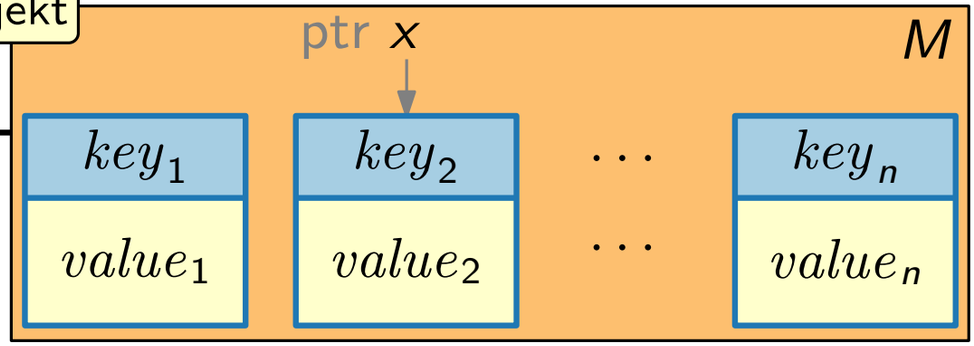
Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer



Anwendung: sortiere Elemente in M

Zurück zu dynamischen Mengen

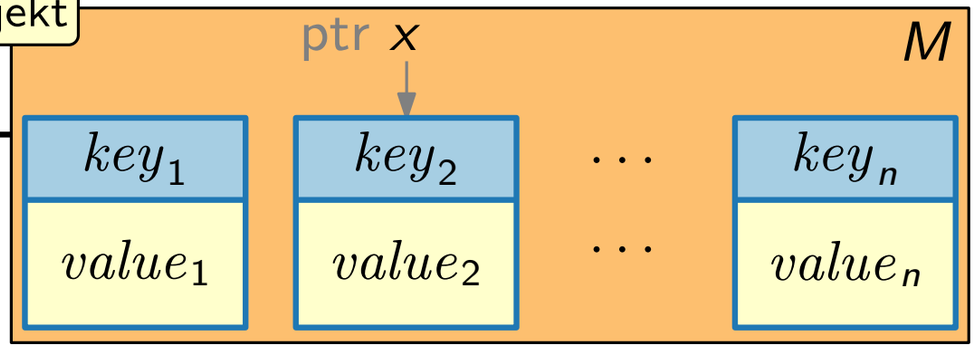
Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer



Anwendung: sortiere Elemente in M

ptr $x = M$.MINIMUM()

Zurück zu dynamischen Mengen

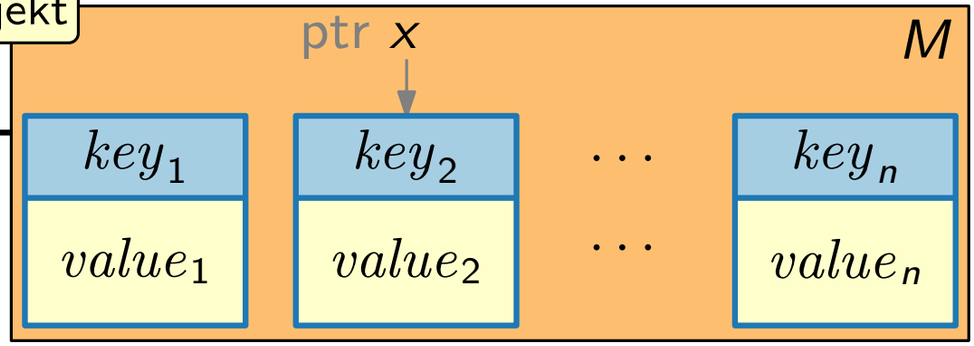
Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer



Anwendung: sortiere Elemente in M

ptr $x = M$.MINIMUM()
while $x \neq nil$ **do**

Zurück zu dynamischen Mengen

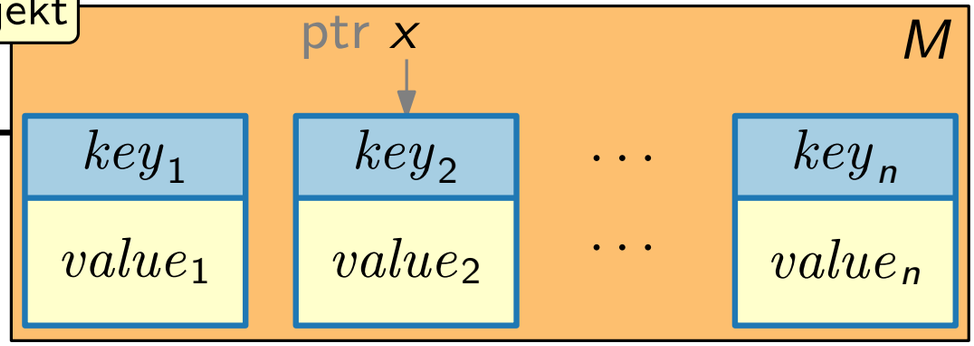
Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer



Anwendung: sortiere Elemente in M

```
ptr x = M.MINIMUM()
while x ≠ nil do
    gib x aus
```

Zurück zu dynamischen Mengen

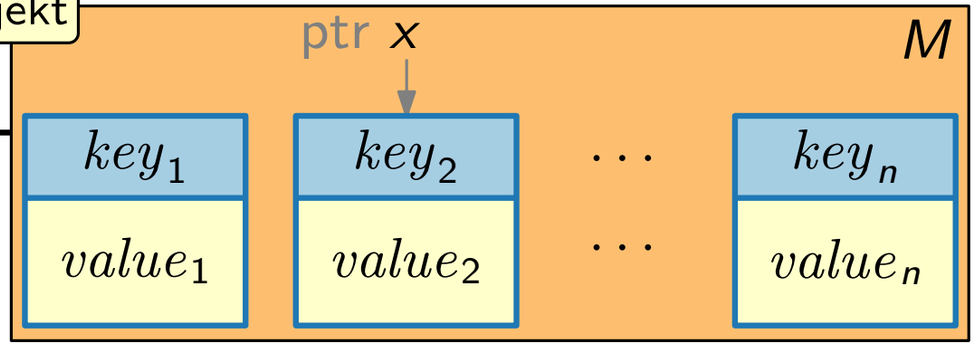
Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer



Anwendung: sortiere Elemente in M

```
ptr x = M.MINIMUM()
while x ≠ nil do
    gib x aus
    x = M.SUCCESSOR(x)
```

Zurück zu dynamischen Mengen

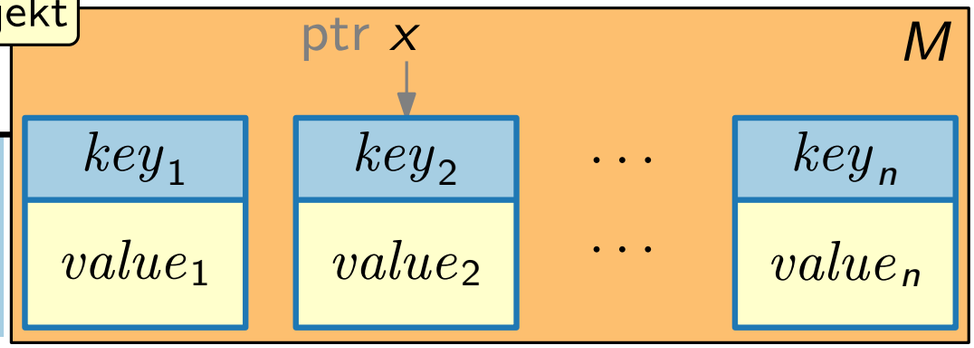
Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer



Zurück zu dynamischen Mengen

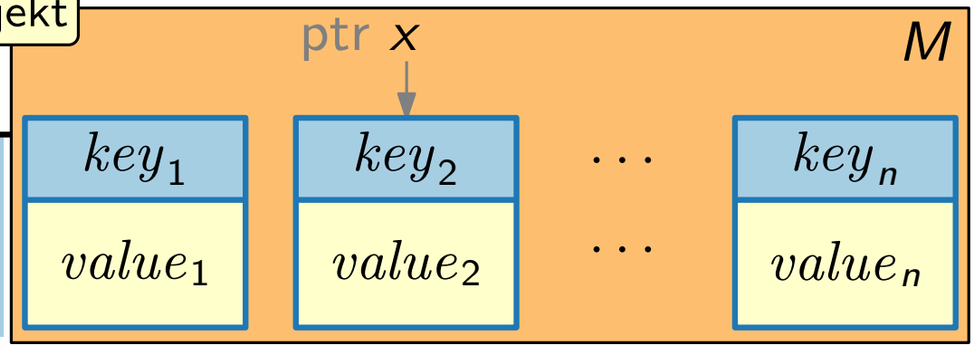
Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	**Veränderungen**
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer



Zurück zu dynamischen Mengen

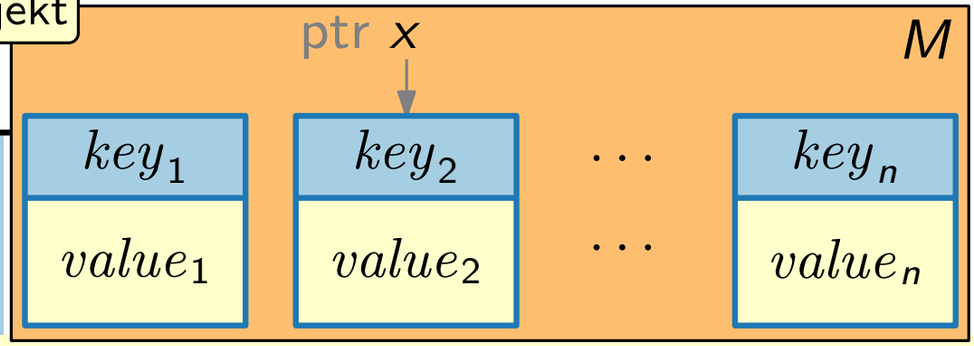
Abstrakter Datentyp: **Dynamische Menge**

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	**Veränderungen**
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer



Zurück zu dynamischen Mengen

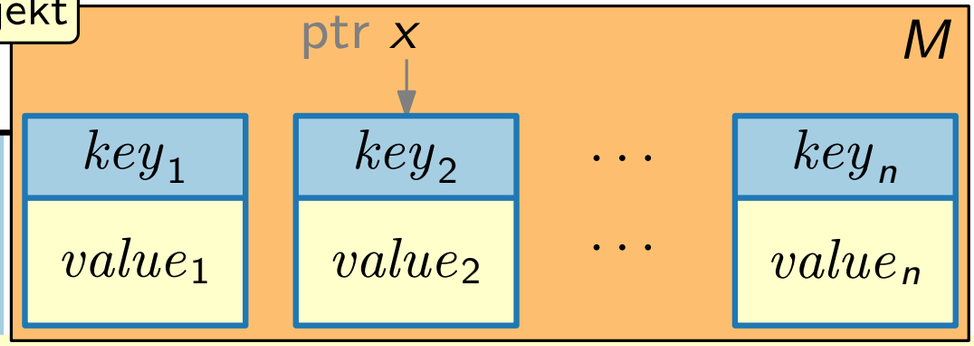
Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt

Operation	Funktionalität
ptr INSERT(key k, value v)	**Veränderungen**
DELETE(ptr x)	
ptr SEARCH(key k)	**Anfragen**
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer

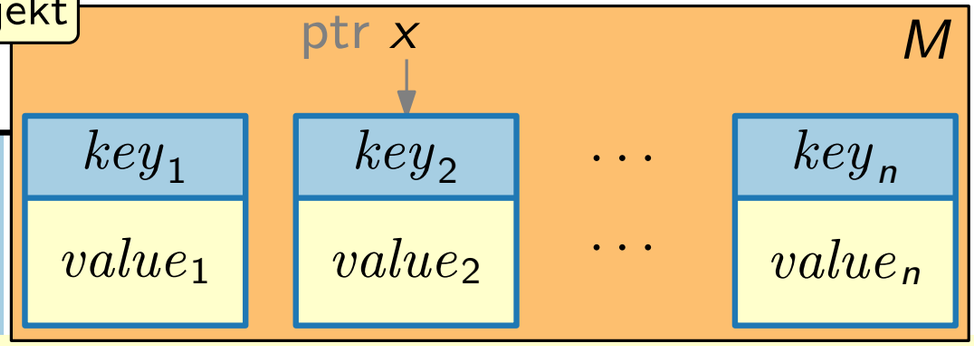


Zurück zu dynamischen Mengen

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt



Operation	Funktionalität
ptr INSERT(key k, value v)	Veränderungen
DELETE(ptr x)	
ptr SEARCH(key k)	Anfragen
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

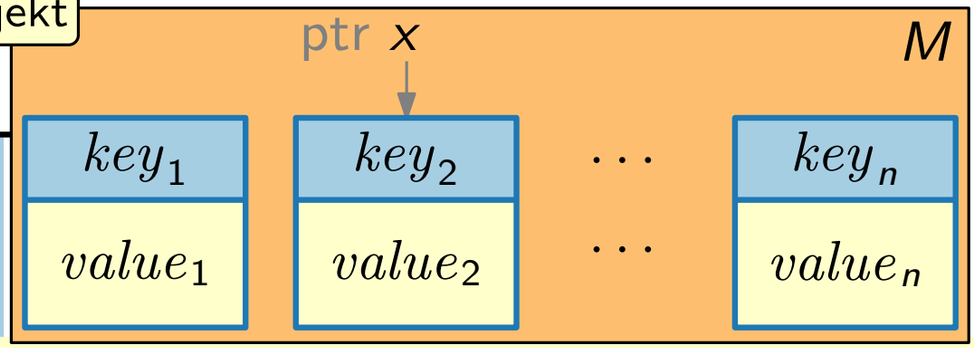
Pointer

Zurück zu dynamischen Mengen

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt



Operation	Funktionalität
ptr INSERT(key k, value v)	**Veränderungen**
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	**Anfragen**
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Wörterbuch

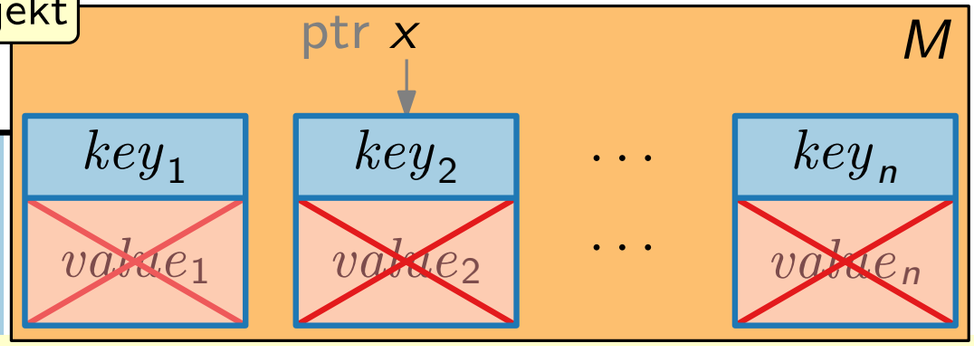
Pointer

Zurück zu dynamischen Mengen

Abstrakter Datentyp: Dynamische Menge

verwaltet Elemente einer Menge M, wobei jedes Element $x \in M$ aus einem Schlüssel $x.key$ und einem Wert $x.value$ besteht.

Beliebiges Objekt



Vereinfachung

Operation	Funktionalität
ptr INSERT(key k, value v)	**Veränderungen**
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	**Anfragen**
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Wörterbuch

Pointer

Implementierung Wörterbuch

Wörterbuch.

Spezialfall einer dynamischen Menge

Abstrakter Datentyp.

stellt folgende Operationen bereit:
INSERT, DELETE, SEARCH

Implementierung.

heute: 3 Varianten

Stapel, Schlange und Liste

I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Operation	Implementierung

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Operation	Implementierung

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Operation	Implementierung

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Operation	Implementierung

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Operation	Implementierung
boolean EMPTY()	
PUSH(key k)	
key POP()	
key TOP()	

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Operation	Implementierung
boolean EMPTY()	**if** $top == 0$ **then return** *true* **else return** *false*
PUSH(key *k*)	
key POP()	
key TOP()	

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Operation	Implementierung
boolean EMPTY()	**if** $top == 0$ **then return** *true* **else return** *false*
PUSH(key k)	$top = top + 1$ $A[top] = k$
key POP()	
key TOP()	

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Operation	Implementierung
boolean EMPTY()	**if** $top == 0$ **then return** *true* **else return** *false*
PUSH(key k)	$top = top + 1$ $A[top] = k$
key POP()	**Aufgabe.** Schreiben Sie Pseudocode, der das oberste Element vom Stapel nimmt und zurückgibt!
key TOP()	

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Operation	Implementierung
boolean EMPTY()	**if** $top == 0$ **then return** *true* **else return** *false*
PUSH(key k)	$top = top + 1$ $A[top] = k$
key POP()	$top = top - 1$ **return** $A[top + 1]$
key TOP()	

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Operation	Implementierung
boolean EMPTY()	**if** $top == 0$ **then return** *true* **else return** *false*
PUSH(key k)	$top = top + 1$ $A[top] = k$

key POP()	**if** EMPTY() **then error** „underflow“
	else
	  $top = top - 1$
	  return $A[top + 1]$

key TOP()	

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Operation	Implementierung
boolean EMPTY()	**if** $top == 0$ **then return** *true* **else return** *false*
PUSH(key k)	$top = top + 1$ $A[top] = k$

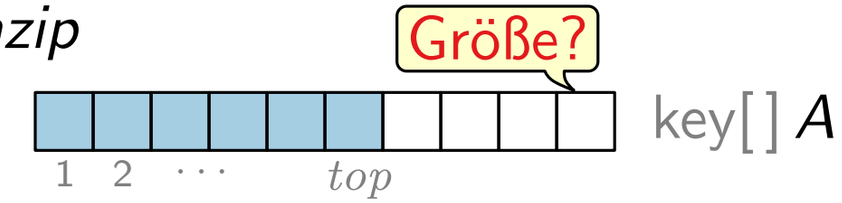
key POP()	**if** EMPTY() **then error** „underflow“
	else
	  $top = top - 1$
	  return $A[top + 1]$

key TOP()	**if** EMPTY() **then error** „underflow“ **else return** $A[top]$

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Operation	Implementierung
boolean EMPTY()	**if** $top == 0$ **then return** *true* **else return** *false*
PUSH(key k)	$top = top + 1$ $A[top] = k$

key POP()	**if** EMPTY() **then error** „underflow“
	else
	  $top = top - 1$
	  return $A[top + 1]$

key TOP()	**if** EMPTY() **then error** „underflow“ **else return** $A[top]$

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Operation	Implementierung
STACK(int n)	
boolean EMPTY()	**if** $top == 0$ **then return** *true* **else return** *false*
PUSH(key k)	$top = top + 1$ $A[top] = k$

key POP()	**if** EMPTY() **then error** „underflow“
	else
	  $top = top - 1$
	  return $A[top + 1]$

key TOP()	**if** EMPTY() **then error** „underflow“ **else return** $A[top]$

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Operation	Implementierung
STACK(int n)	$A =$ **new** key$[1 \ldots n]$ $top = 0$
boolean EMPTY()	**if** $top == 0$ **then return** *true* **else return** *false*
PUSH(key k)	$top = top + 1$ $A[top] = k$

key POP()	**if** EMPTY() **then error** „underflow“
	else
	  $top = top - 1$
	  return $A[top + 1]$

key TOP()	**if** EMPTY() **then error** „underflow“ **else return** $A[top]$

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Operation	Implementierung
STACK(int n)	$A =$ **new** key$[1 \ldots n]$ $top = 0$
boolean EMPTY()	**if** $top == 0$ **then return** *true* **else return** *false*
PUSH(key k)	$top = top + 1$ ← **if** $top > A.length$ **then error** „overflow“ $A[top] = k$

key POP()	**if** EMPTY() **then error** „underflow“
	else
	  $top = top - 1$
	  return $A[top + 1]$

key TOP()	**if** EMPTY() **then error** „underflow“ **else return** $A[top]$

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Operation	Implementierung
STACK(int n)	$A =$ **new** key$[1 \dots n]$ $top = 0$
boolean EMPTY()	**if** $top == 0$ **then return** *true* **else return** *false*
PUSH(key k)	$top = top + 1$ ← **if** $top > A.length$ **then error** „overflow“ $A[top] = k$

key POP()	**if** EMPTY() **then error** „underflow“
	else
	  $top = top - 1$
	  return $A[top + 1]$

key TOP()	**if** EMPTY() **then error** „underflow“ **else return** $A[top]$

Laufzeiten?

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Operation	Implementierung
STACK(int n)	$A =$ **new*** key$[1 \dots n]$ $top = 0$
boolean EMPTY()	**if** $top == 0$ **then return** *true* **else return** *false*
PUSH(key k)	$top = top + 1$ ← **if** $top > A.length$ **then error** „overflow“ $A[top] = k$

key POP()	**if** EMPTY() **then error** „underflow“
	else
	  $top = top - 1$
	  return $A[top + 1]$

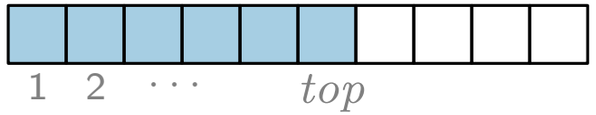
key TOP()	**if** EMPTY() **then error** „underflow“ **else return** $A[top]$

Laufzeiten?
Alle* $\mathcal{O}(1)$,
d.h. konstant.

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



key[ ] A

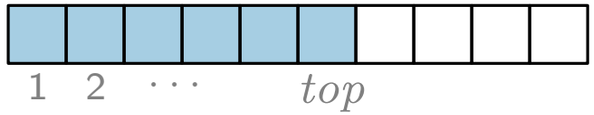
Operation	Implementierung
STACK(int n) **Konstruktor**	$A =$ **new*** key$[1 \dots n]$ $top = 0$
boolean EMPTY()	**if** $top == 0$ **then return** *true* **else return** *false*
PUSH(key k)	$top = top + 1$ ← **if** $top > A.length$ **then error** „overflow“ $A[top] = k$
key POP()	**if** EMPTY() **then error** „underflow“ **else** $top = top - 1$ **return** $A[top + 1]$
key TOP()	**if** EMPTY() **then error** „underflow“ **else return** $A[top]$

Laufzeiten?
Alle* $\mathcal{O}(1)$,
d.h. konstant.

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



key[ ] A

Operation	Implementierung
STACK(int n) **Konstruktor**	$A =$ **new*** key$[1 \dots n]$ $top = 0$
boolean EMPTY()	**if** $top == 0$ **then return** *true* **else return** *false*
PUSH(key k)	$top = top + 1$ ← **if** $top > A.length$ **then error** „overflow“ $A[top] = k$
key POP()	**if** EMPTY() **then error** „underflow“ **else** $top = top - 1$ **return** $A[top + 1]$
key TOP()	**if** EMPTY() **then error** „underflow“ **else return** $A[top]$

Laufzeiten?
Alle* $\mathcal{O}(1)$,
d.h. konstant.

Methoden

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Operation	Implementierung
STACK(int n)	$A =$ **new*** key$[1 \ldots n]$ $top = 0$
boolean EMPTY()	**if** $top == 0$ **then return** *true* **else return** *false*
PUSH(key k)	$top = top + 1$ $A[top] = k$ ← **if** $top > A.length$ **then error** „overflow“

key POP()	**if** EMPTY() **then error** „underflow“
	else
	  $top = top - 1$
	  return $A[top + 1]$

key TOP()	**if** EMPTY() **then error** „underflow“ **else return** $A[top]$

Konstruktor

Attribute

Laufzeiten?
Alle* $\mathcal{O}(1)$,
d.h. konstant.

Methoden

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Operation	
Stack(int	Konstru
boolean E	
Push(key	
key Pop()	
key Top()	**if** Empty() **then error** „underflow“ **else return** $A[top]$

```
public class PKStack {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();
        int[] A = {4, 8, 15, 16, 23, 42};

        for (int i = 0; i < A.length; i++) {
          stack.push(A[i]);
        }

        System.out.println("Stack enthaelt:");
        while (!stack.isEmpty()) {
          System.out.print(stack.pop() + " ");
        }
    }
}
```

algo.uni-trier.de/demos/memvis.html

thoden

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Operation

STACK(int

Konstrul

boolean E

PUSH(key

key POP()

key TOP()

```
public class PKStack {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();
        int[] A = {4, 8, 15, 16, 23, 42};

        for (int i = 0; i < A.length; i++) {
          stack.push(A[i]);
        }

        System.out.println("Stack enthaelt:");
        while (!stack.isEmpty()) {
          System.out.print(stack.pop() + " ");
        }
    }
}
```

output:

```
Stack enthaelt:
```

thoden

algo.uni-trier.de/demos/memvis.html

if EMPTY() **then error** „underflow“ **else return** $A[top]$

I. Stapel

last-in first-out

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

Operation

STACK(int

Konstrul

boolean E

PUSH(key

key POP()

key TOP()

```
public class PKStack {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();
        int[] A = {4, 8, 15, 16, 23, 42};

        for (int i = 0; i < A.length; i++) {
          stack.push(A[i]);
        }

        System.out.println("Stack enthaelt:");
        while (!stack.isEmpty()) {
          System.out.print(stack.pop() + " ");
        }
    }
}
```

output:

```
Stack enthaelt:
42 23 16 15 8 4
```

thoden

algo.uni-trier.de/demos/memvis.html

if EMPTY() **then error** „underflow“ **else return** $A[top]$

II. Schlange

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Operation	Implementierung

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Operation	Implementierung

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Operation	Implementierung

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Operation	Implementierung

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Operation	Implementierung

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Operation	Implementierung

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Operation	Implementierung

II. Schlange

first-in first-out

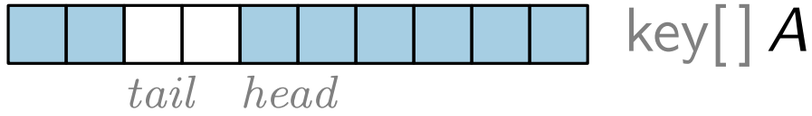
verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Operation	Implementierung

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

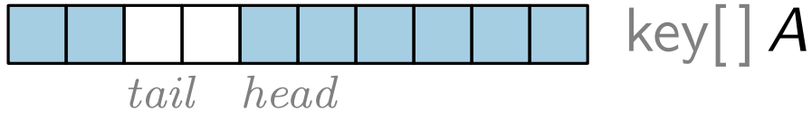


Operation	Implementierung

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*



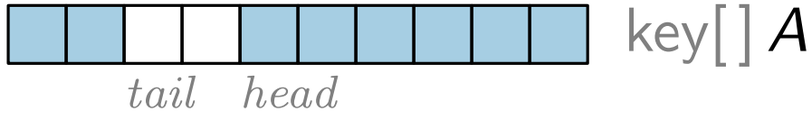
Operation	Implementierung
QUEUE(int n)	

Konstruktor

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

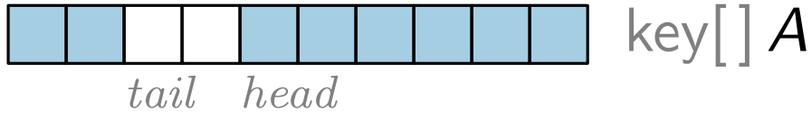


Operation	Implementierung
QUEUE(int n) **Konstruktor**	$A =$ **new** key$[1 \ldots n+1]$ $tail = head = 1$

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

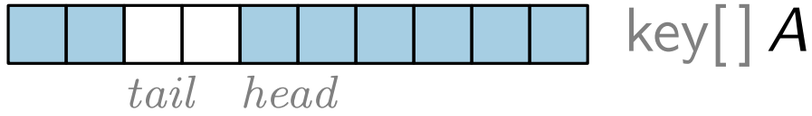


Operation	Implementierung
QUEUE(int n) **Konstruktor**	$A =$ **new** key$[1 \ldots n+1]$ $tail = head = 1$
boolean EMPTY()	

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

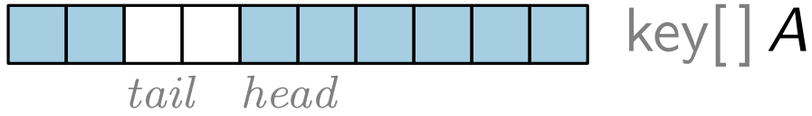


Operation	Implementierung
QUEUE(int n) **Konstruktor**	$A =$ **new** key$[1 \ldots n+1]$ $tail = head = 1$
boolean EMPTY()	**if** $head == tail$ **then return** *true* **else return** *false*

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

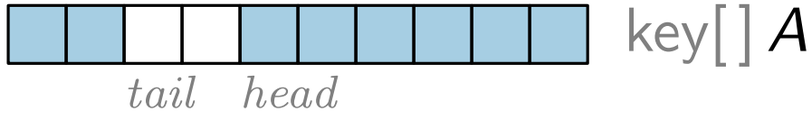


Operation	Implementierung
QUEUE(int n) Konstruktor	$A =$ **new** key$[1 \ldots n+1]$ $tail = head = 1$
boolean EMPTY()	**if** $head == tail$ **then return** *true* **else return** *false*
ENQUEUE(key k)	

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

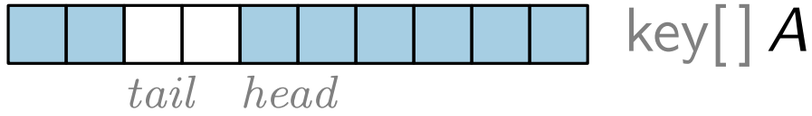


Operation	Implementierung
QUEUE(int n) **Konstruktor**	$A =$ **new** key$[1 \ldots n+1]$ $tail = head = 1$
boolean EMPTY()	**if** $head == tail$ **then return** *true* **else return** *false*
ENQUEUE(key k) stell neues Element an den Schwanz der Schlange an	

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

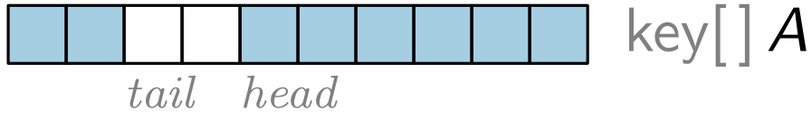


Operation	Implementierung
QUEUE(int n) **Konstruktor**	$A =$ **new** key$[1 \ldots n+1]$ $tail = head = 1$
boolean EMPTY()	**if** $head == tail$ **then return** *true* **else return** *false*
ENQUEUE(key k) stell neues Element an den Schwanz der Schlange an	$A[tail] = k$ **if** $tail == A.length$ **then** $tail = 1$ **else** $tail = tail + 1$

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

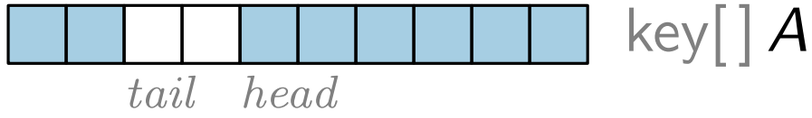


Operation	Implementierung
QUEUE(int n) **Konstruktor**	$A =$ **new** key$[1 \ldots n+1]$ $tail = head = 1$
boolean EMPTY()	**if** $head == tail$ **then return** *true* **else return** *false*
ENQUEUE(key k) stell neues Element an den Schwanz der Schlange an	$A[tail] = k$ **if** $tail == A.length$ **then** $tail = 1$ **else** $tail = tail + 1$
key DEQUEUE() entnimm Element am Kopf der Schlange	

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

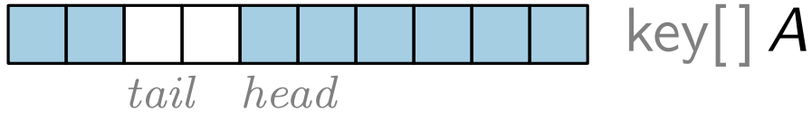


Operation	Implementierung
QUEUE(int n) **Konstruktor**	$A =$ **new** key$[1 \ldots n+1]$ $tail = head = 1$
boolean EMPTY()	**if** $head == tail$ **then return** *true* **else return** *false*
ENQUEUE(key k) stell neues Element an den Schwanz der Schlange an	$A[tail] = k$ **if** $tail == A.length$ **then** $tail = 1$ **else** $tail = tail + 1$
key DEQUEUE() entnimm Element am Kopf der Schlange	$k = A[head]$ **if** $head == A.length$ **then** $head = 1$ **else** $head = head + 1$ **return** k

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

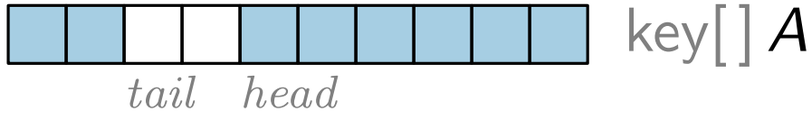


Operation	Implementierung
QUEUE(int n) **Konstruktor**	$A =$ **new** key$[1 \ldots n+1]$ $tail = head = 1$
boolean EMPTY()	**if** $head == tail$ **then return** *true* **else return** *false*
ENQUEUE(key k) stell neues Element an den Schwanz der Schlange an	$A[tail] = k$ **if** $tail == A.length$ **then** $tail = 1$ **else** $tail = tail + 1$
key DEQUEUE() entnimm Element am Kopf der Schlange	$k = A[head]$ **if** $head == A.length$ **then** $head = 1$ **else** $head = head + 1$ **return** k

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

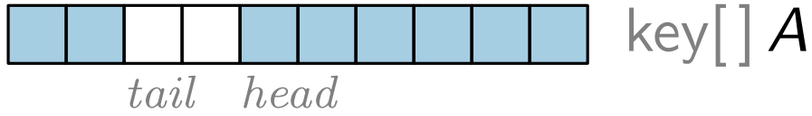


Operation	Implementierung
QUEUE(int n) **Konstruktor**	$A =$ **new** key$[1 \ldots n+1]$ $tail = head = 1$
boolean EMPTY()	**if** $head == tail$ **then return** *true* **else return** *false*
ENQUEUE(key k) stell neues Element an den Schwanz der Schlange an	$A[tail] = k$ **if** $tail == A.length$ **then** $tail = 1$ **else** $tail = tail + 1$
key DEQUEUE() entnimm Element am Kopf der Schlange	$k = A[head]$ **if** $head == A.length$ **then** $head = 1$ **else** $head = head + 1$ **return** k

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

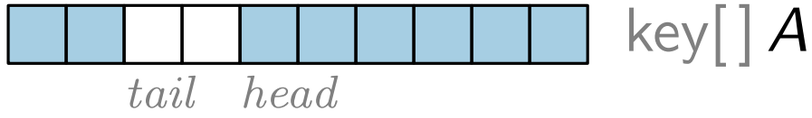


Operation	Implementierung
QUEUE(int n) **Konstruktor**	$A =$ **new** key$[1 \ldots n+1]$ $tail = head = 1$
boolean EMPTY()	**if** $head == tail$ **then return** *true* **else return** *false*
ENQUEUE(key k) stell neues Element an den Schwanz der Schlange an	$A[tail] = k$ **if** $tail == A.length$ **then** $tail = 1$ **else** $tail = tail + 1$
key DEQUEUE() entnimm Element am Kopf der Schlange	$k = A[head]$ **if** $head == A.length$ **then** $head = 1$ **else** $head = head + 1$ **return** k

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*



Operation	Implementierung
QUEUE(int n) **Konstruktor**	$A =$ **new** key$[1 \ldots n+1]$ $tail = head = 1$
boolean EMPTY()	**if** $head == tail$ **then return** *true* **else return** *false*
ENQUEUE(key k) stell neues Element an den Schwanz der Schlange an	$A[tail] = k$ **if** $tail == A.length$ **then** $tail = 1$ **else** $tail = tail + 1$
key DEQUEUE() entnimm Element am Kopf der Schlange	$k = A[head]$ **if** $head == A.length$ **then** $head = 1$ **else** $head = head + 1$ **return** k

Aufgabe.
Fangen Sie underflow & overflow ab!

II. Schlange

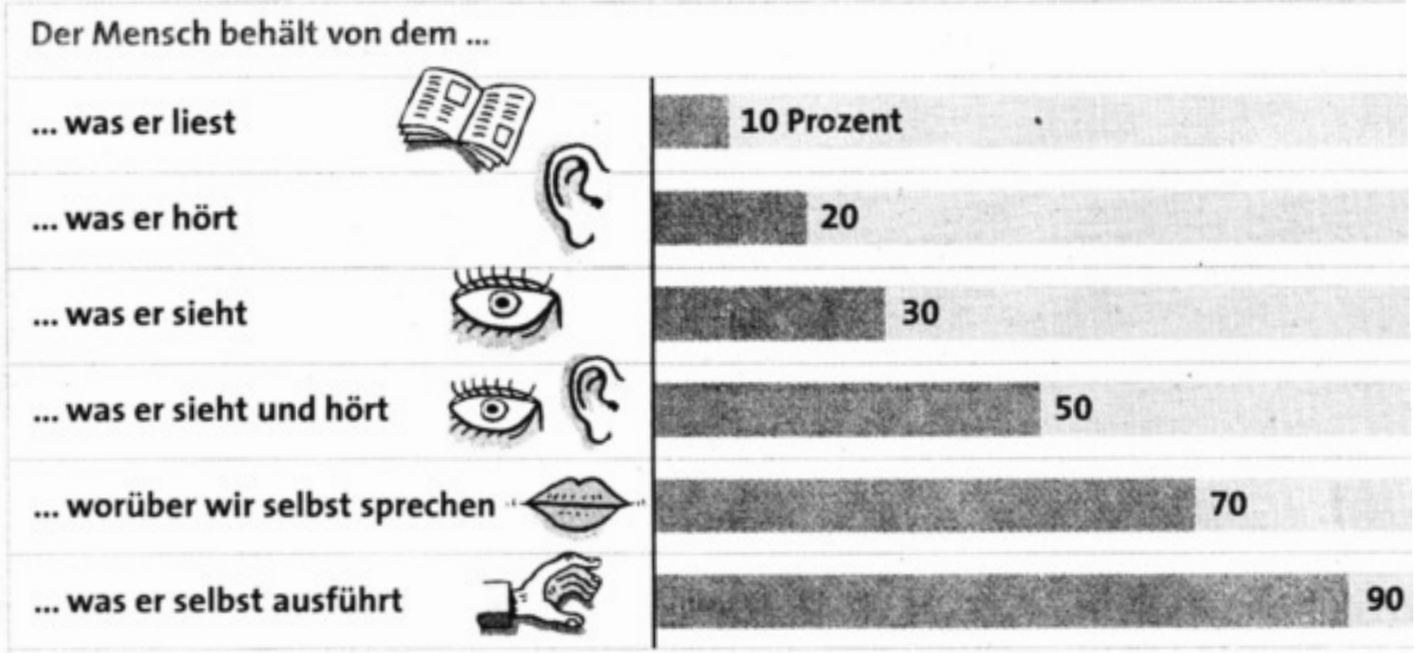
first-in first-out

key[ ] *A*

Aufgabe.
Fangen Sie
underflow &
overflow ab!

Die sinnliche Spur der Erinnerung

Wer lernen will, muss vor allem reden und be-greifen



HANDLUNGSORIENTIERTES LERNEN ist am effektivsten. Diese Einsicht ist seit fast 20 Jahren bekannt. Damals erschien diese Studie der American Audiovisual Society. Ergebnis: Von dem, was wir mit eigenen Händen tun, behalten wir 90 Prozent im Gedächtnis, von dem, worüber wir selbst sprechen, immerhin noch 70 Prozent. Von der reinen Lektüre eines Buches erinnern wir später nur noch 10 Prozent

II. Schlange

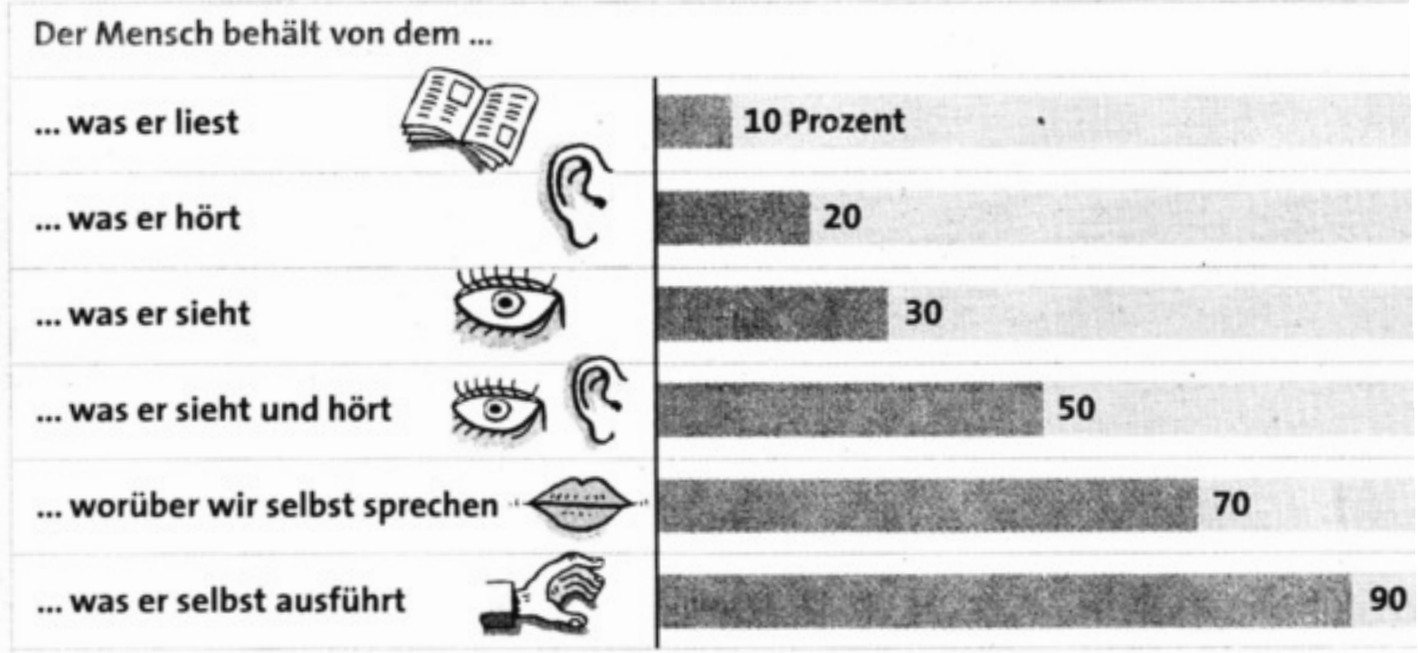
first-in first-out

key[ ] *A*

Aufgabe.
Fangen Sie
underflow &
overflow ab!

Die sinnliche Spur der Erinnerung

Wer lernen will, muss vor allem reden und be-greifen

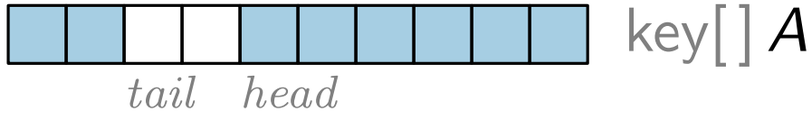


HANDLUNGSORIENTIERTES LERNEN ist am effektivsten. Diese Einsicht ist seit fast 20 Jahren bekannt. Damals erschien diese Studie der American Audiovisual Society. Ergebnis: Von dem, was wir mit eigenen Händen tun, behalten wir 90 Prozent im Gedächtnis, von dem, worüber wir selbst sprechen, immerhin noch 70 Prozent. Von der reinen Lektüre eines Buches erinnern wir später nur noch 10 Prozent

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*



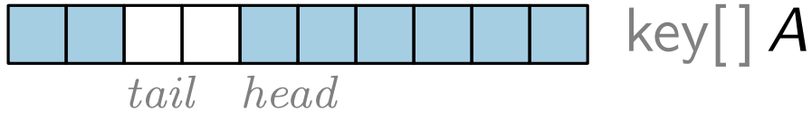
Operation	Implementierung
QUEUE(int n) **Konstruktor**	$A =$ **new** key$[1 \ldots n+1]$ $tail = head = 1$
boolean EMPTY()	**if** $head == tail$ **then return** *true* **else return** *false*
ENQUEUE(key k) stell neues Element an den Schwanz der Schlange an	$A[tail] = k$ **if** $tail == A.length$ **then** $tail = 1$ **else** $tail = tail + 1$
key DEQUEUE() entnimm Element am Kopf der Schlange	$k = A[head]$ **if** $head == A.length$ **then** $head = 1$ **else** $head = head + 1$ **return** k

Aufgabe.
Fangen Sie underflow & overflow ab!

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*



Operation	Implementierung
QUEUE(int n) **Konstruktor**	$A =$ **new** key$[1 \ldots n+1]$ $tail = head = 1$
boolean EMPTY()	**if** $head == tail$ **then return** *true* **else return** *false*
ENQUEUE(key k) stell neues Element an den Schwanz der Schlange an	$A[tail] = k$ **if** $tail == A.length$ **then** $tail = 1$ **else** $tail = tail + 1$
key DEQUEUE() entnimm Element am Kopf der Schlange	$k = A[head]$ **if** $head == A.length$ **then** $head = 1$ **else** $head = head + 1$ **return** k

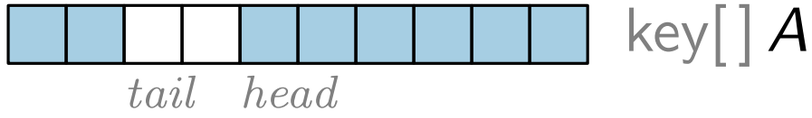
Aufgabe.
Fangen Sie underflow & overflow ab!

Laufzeiten?

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*



Operation	Implementierung
QUEUE(int n) **Konstruktor**	$A =$ **new*** key$[1 \ldots n+1]$ $tail = head = 1$
boolean EMPTY()	**if** $head == tail$ **then return** *true* **else return** *false*
ENQUEUE(key k) stell neues Element an den Schwanz der Schlange an	$A[tail] = k$ **if** $tail == A.length$ **then** $tail = 1$ **else** $tail = tail + 1$
key DEQUEUE() entnimm Element am Kopf der Schlange	$k = A[head]$ **if** $head == A.length$ **then** $head = 1$ **else** $head = head + 1$ **return** k

Aufgabe.
Fangen Sie underflow & overflow ab!

Laufzeiten?
Alle* $\mathcal{O}(1)$.

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Operatio

QUEUE(int

Konstr

boolean E

ENQUEUE(

stell neues Elem
Schwanz der Sch

key DEQU

entnimm Elemer
der Schlange

```
public class PKQueue {
    public static void main(String[] args) {
        Queue<Integer> queue = new Queue<>();
        int[] A = {4, 8, 15, 16, 23, 42};

        for (int i = 0; i < A.length; i++) {
          queue.enqueue(A[i]);
        }

        System.out.println("Queue enthaelt:");
        while (!queue.isEmpty()) {
          System.out.print(queue.dequeue() + " ");
        }
    }
}
```

algo.uni-trier.de/demos/memvis.html

return *k*

Laufzeiten?

Alle* $\mathcal{O}(1)$.

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Operatio

QUEUE(int

Konstr

boolean E

ENQUEUE(

stell neues Elem
Schwanz der Sch

key DEQU

entnimm Elemer
der Schlange

```
public class PKQueue {
    public static void main(String[] args) {
        Queue<Integer> queue = new Queue<>();
        int[] A = {4, 8, 15, 16, 23, 42};

        for (int i = 0; i < A.length; i++) {
          queue.enqueue(A[i]);
        }

        System.out.println("Queue enthaelt:");
        while (!queue.isEmpty()) {
          System.out.print(queue.dequeue() + " ");
        }
    }
}
```

output:

```
Queue enthaelt:
```

algo.uni-trier.de/demos/memvis.html

return *k*

Laufzeiten?

Alle* $\mathcal{O}(1)$.

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*

Operatio

QUEUE(int

Konstr

boolean E

ENQUEUE(

stell neues Elem
Schwanz der Sch

key DEQU

entnimm Elemer
der Schlange

```
public class PKQueue {
    public static void main(String[] args) {
        Queue<Integer> queue = new Queue<>();
        int[] A = {4, 8, 15, 16, 23, 42};

        for (int i = 0; i < A.length; i++) {
          queue.enqueue(A[i]);
        }

        System.out.println("Queue enthaelt:");
        while (!queue.isEmpty()) {
          System.out.print(queue.dequeue() + " ");
        }
    }
}
```

output:

```
Queue enthaelt:
4 8 15 16 23 42
```

algo.uni-trier.de/demos/memvis.html

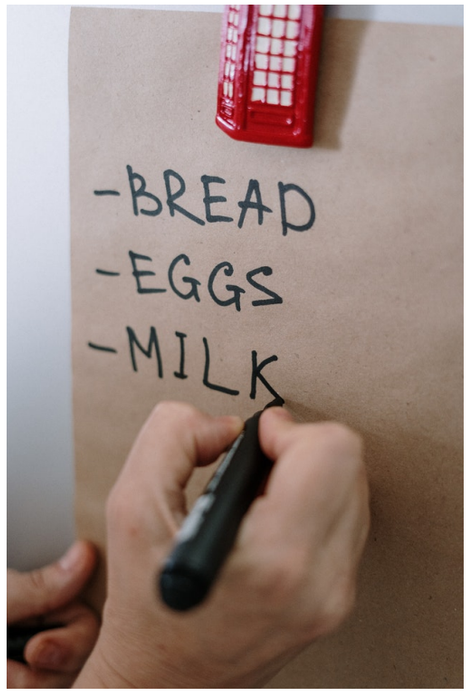
return *k*

Laufzeiten?

Alle* $\mathcal{O}(1)$.

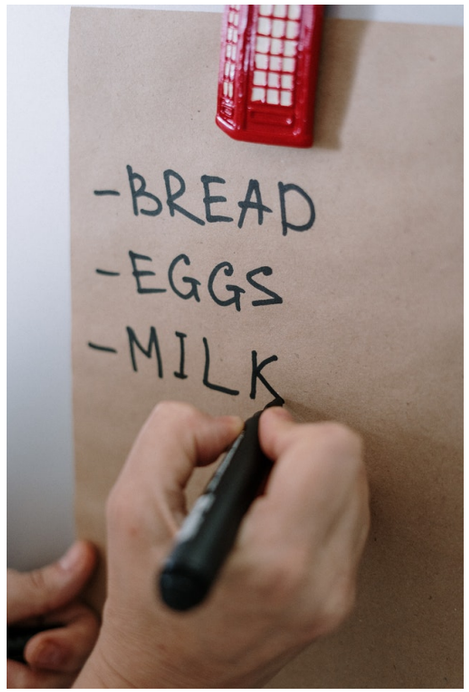
III. Liste

Operation	Implementierung

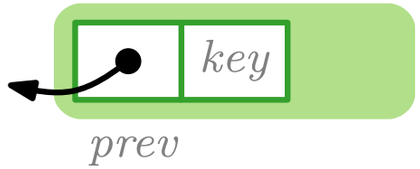


III. Liste

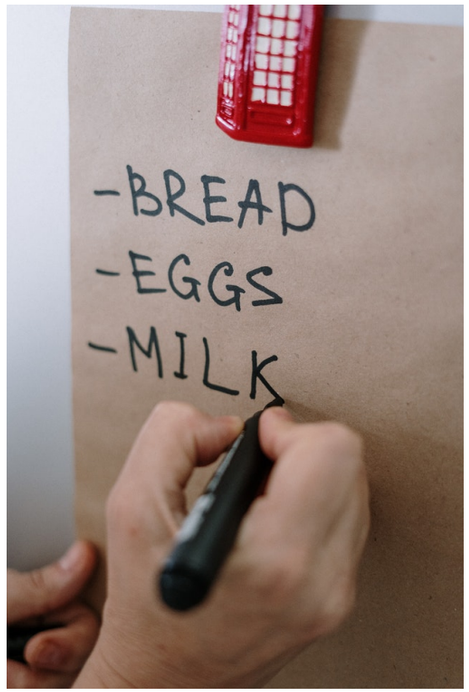
Operation	Implementierung



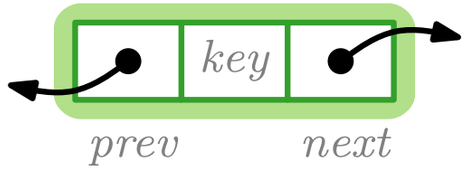
III. Liste



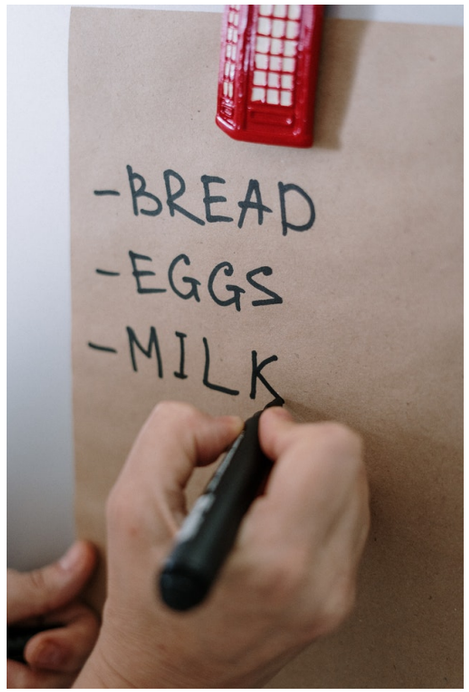
Operation	Implementierung



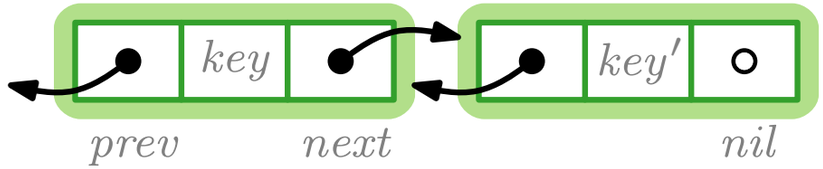
III. Liste



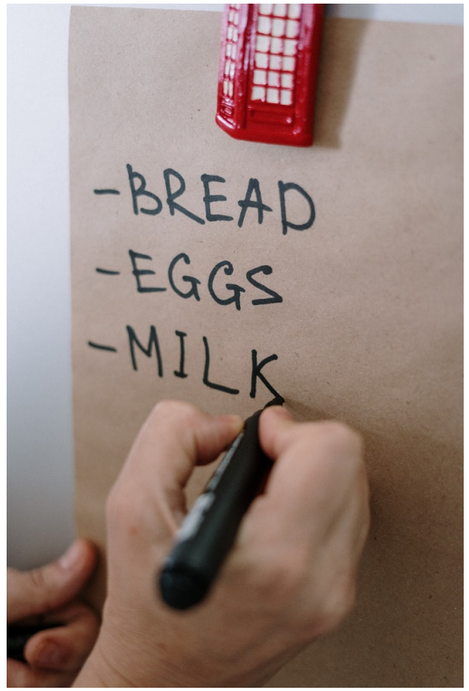
Operation	Implementierung



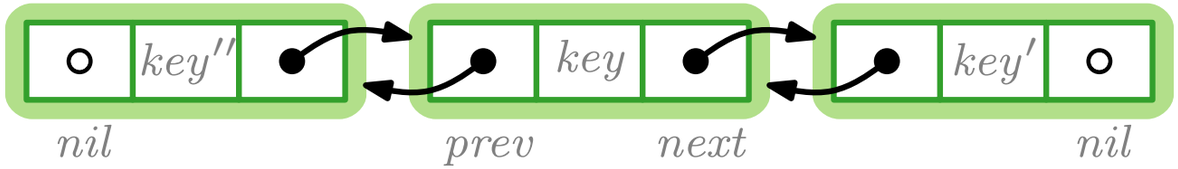
III. Liste



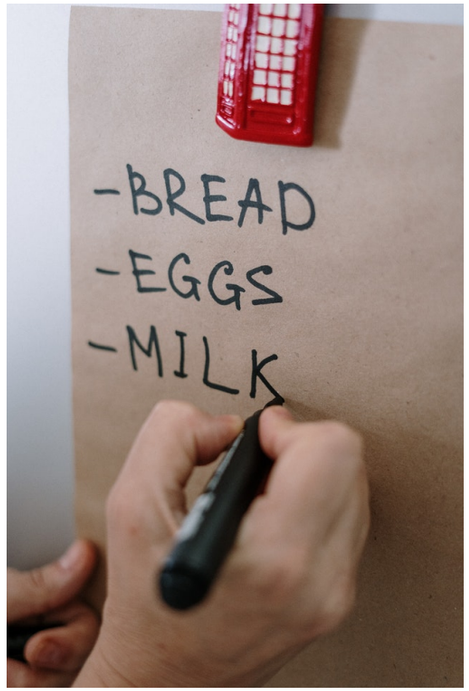
Operation	Implementierung



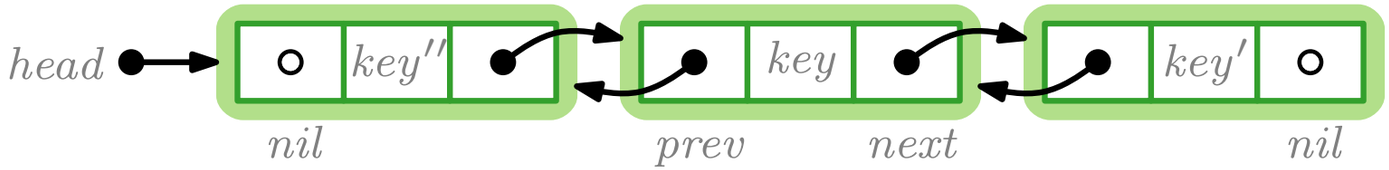
III. Liste



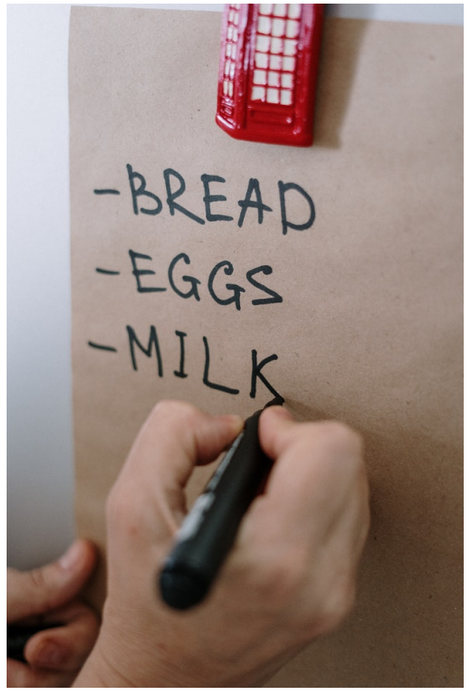
Operation	Implementierung



III. Liste

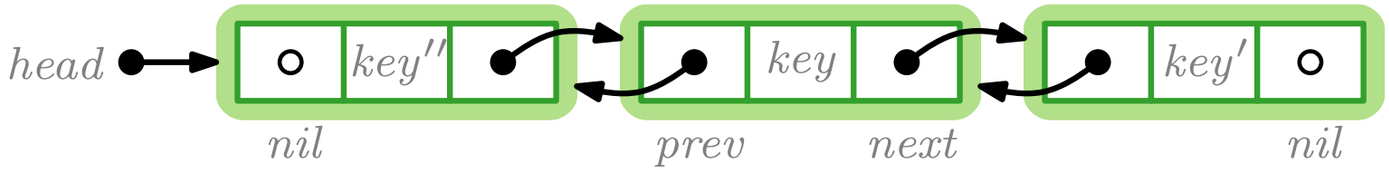


Operation	Implementierung

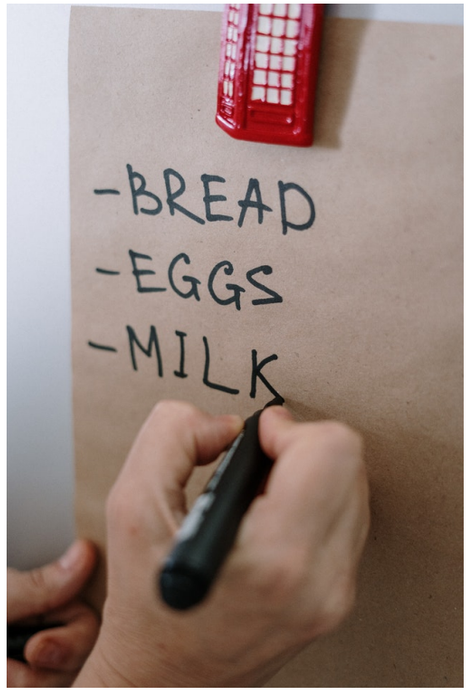


III. Liste

(doppelt verkettet)

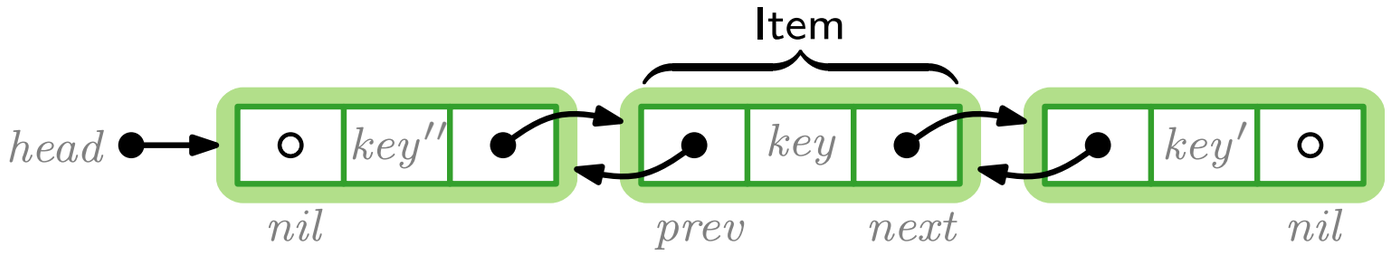


Operation	Implementierung

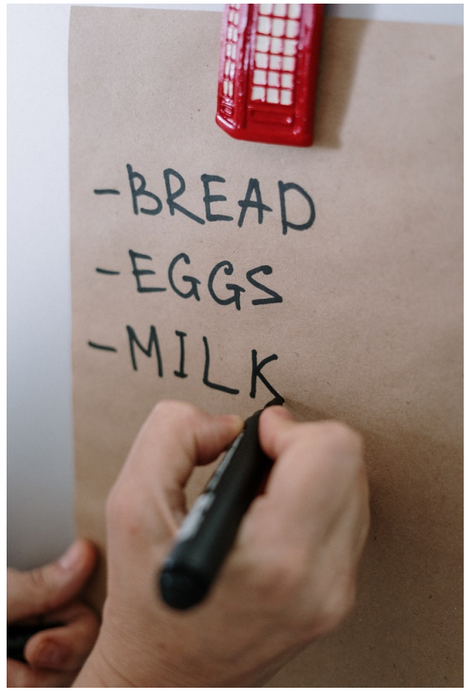


III. Liste

(doppelt verkettet)

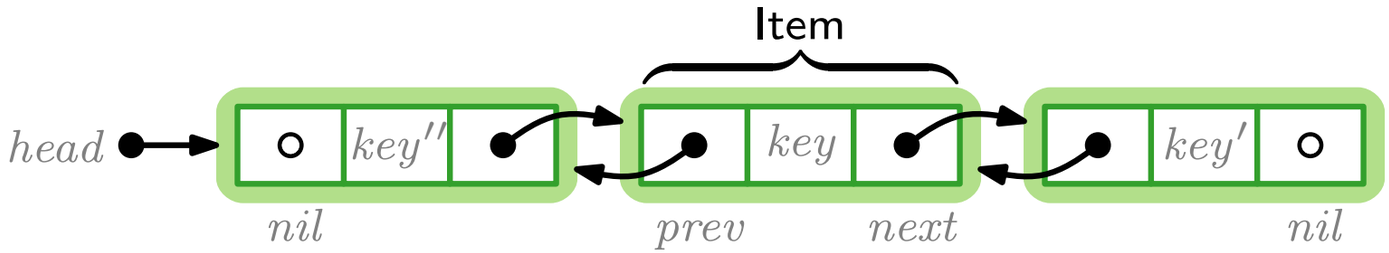


Operation	Implementierung

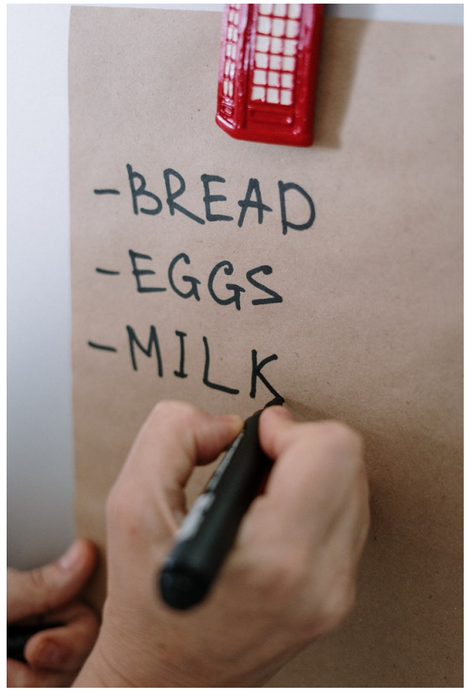


III. Liste

(doppelt verkettet)

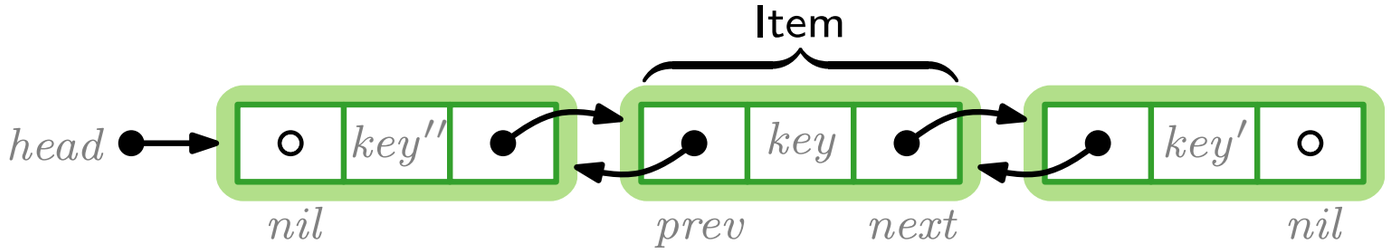


Operation	Implementierung
List()	

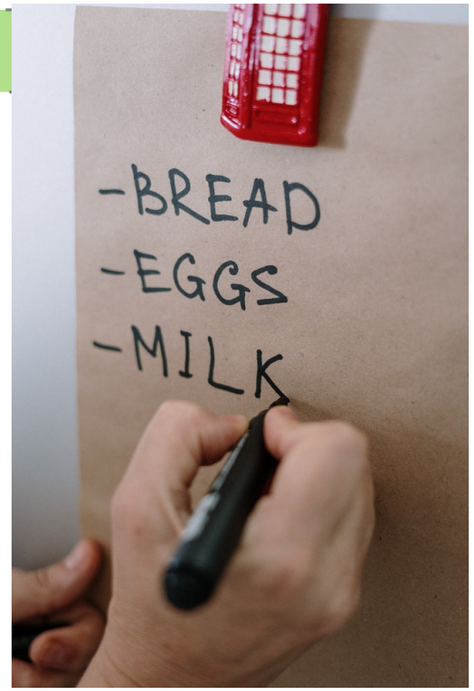


III. Liste

(doppelt verkettet)

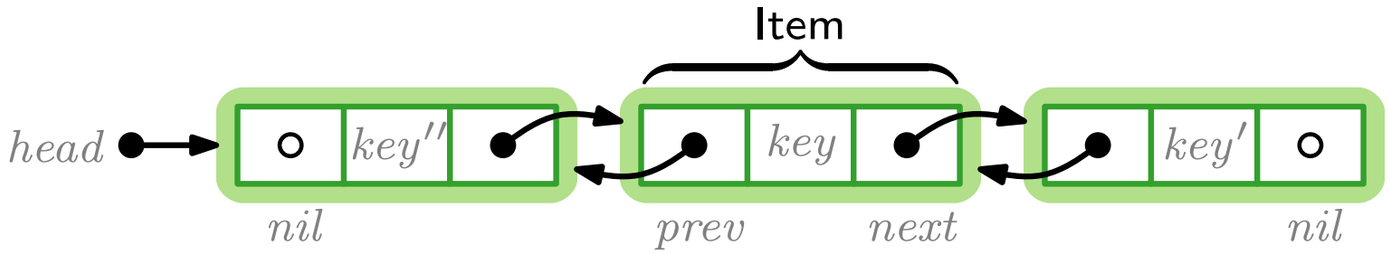


Operation	Implementierung
List()	$head = nil$

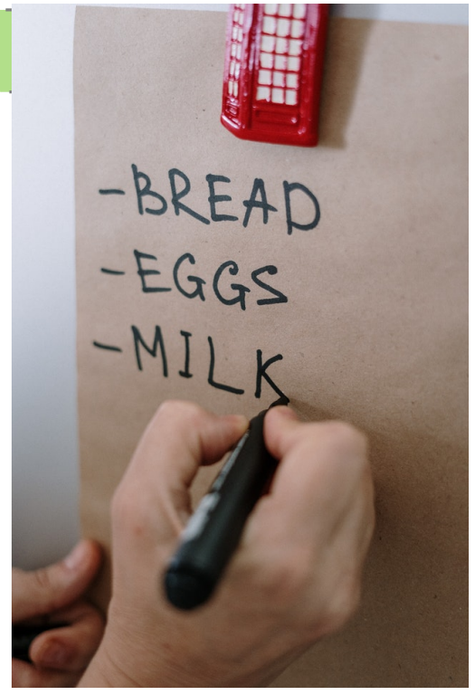


III. Liste

(doppelt verkettet)

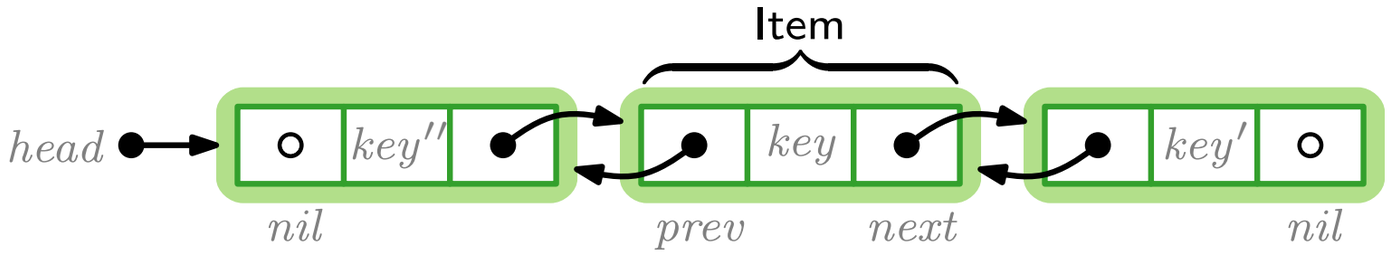


Operation	Implementierung
List()	$head = nil$
ptr Search(key k)	

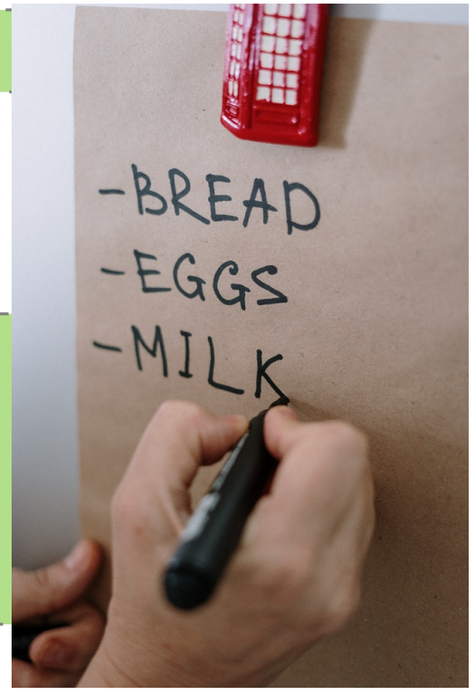


III. Liste

(doppelt verkettet)

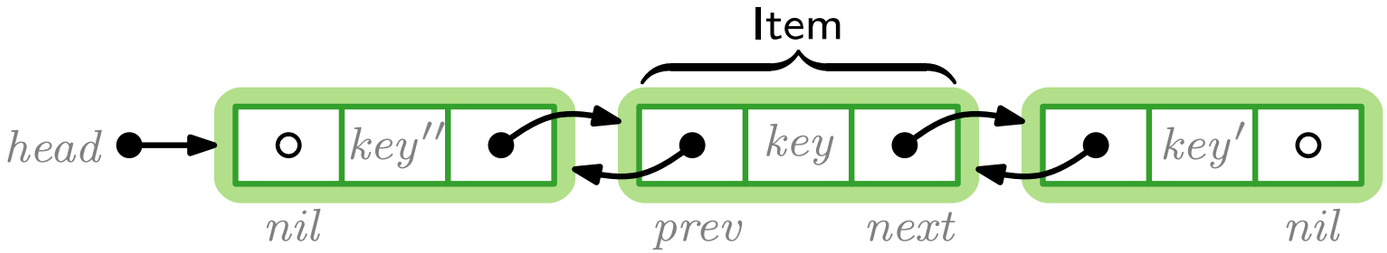


Operation	Implementierung
List()	$head = nil$
ptr Search(key k)	$x = head$

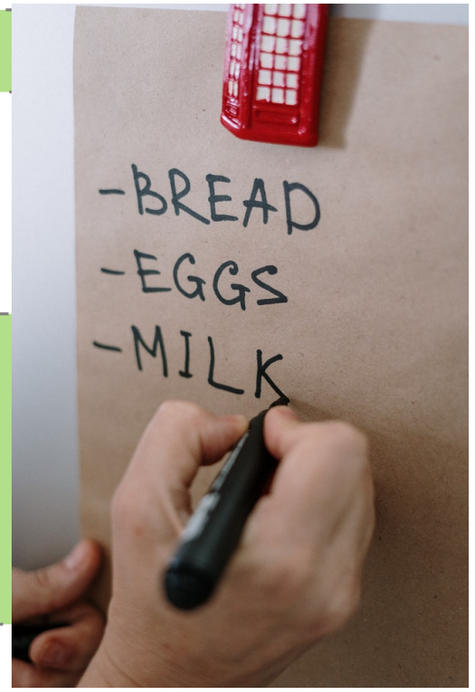


III. Liste

(doppelt verkettet)

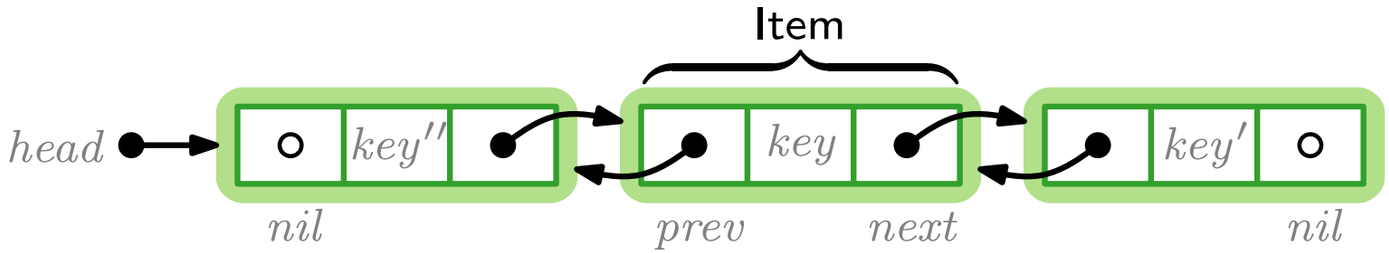


Operation	Implementierung
List()	$head = nil$
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $\quad x = x.next$

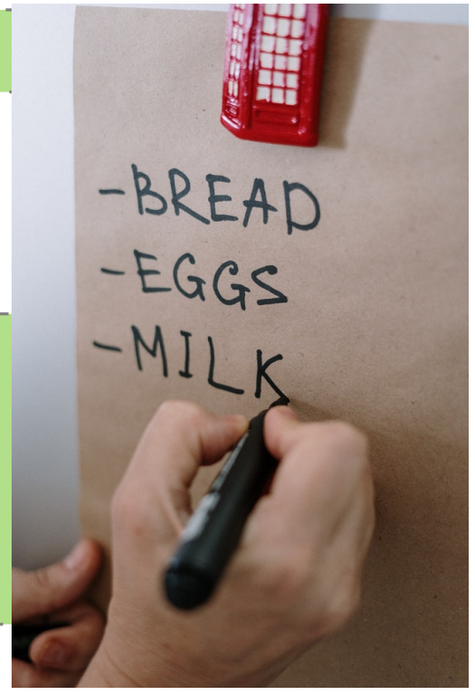


III. Liste

(doppelt verkettet)

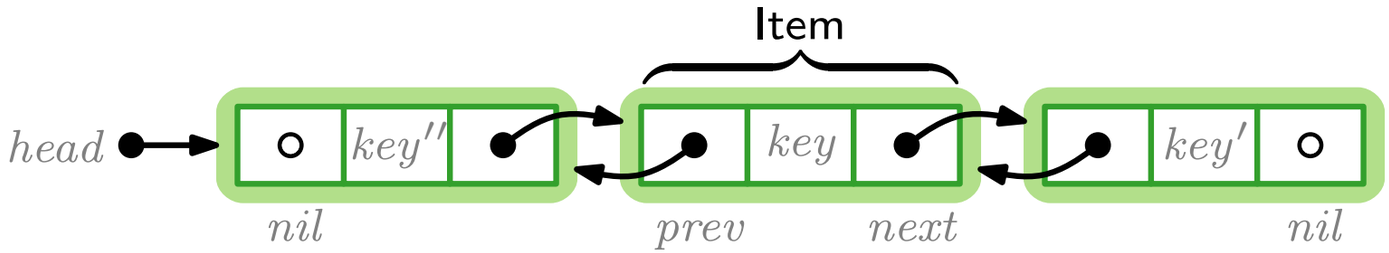


Operation	Implementierung
List()	$head = nil$
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $\quad x = x.next$ **return** x

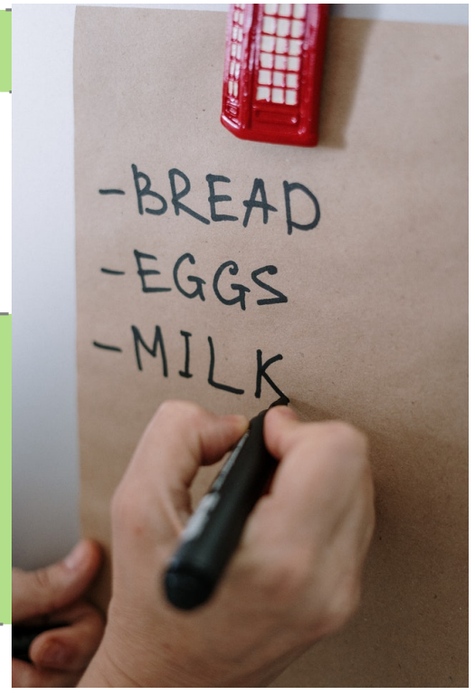


III. Liste

(doppelt verkettet)

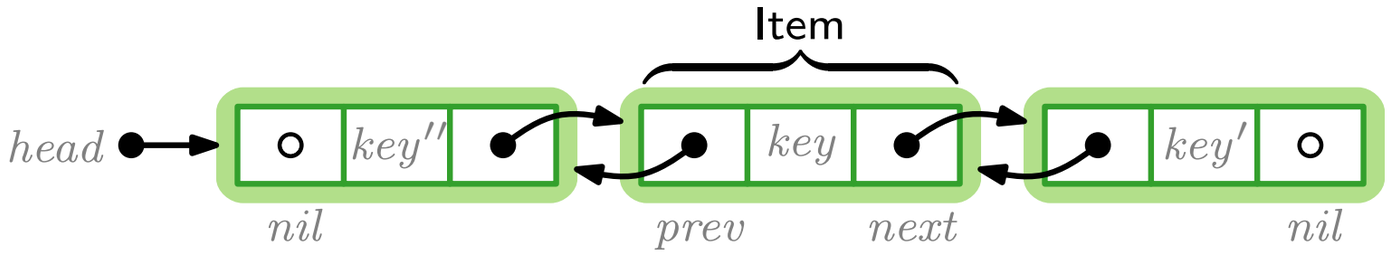


Operation	Implementierung
List()	$head = nil$
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $\quad x = x.next$ **return** x
ptr Insert(key k)	

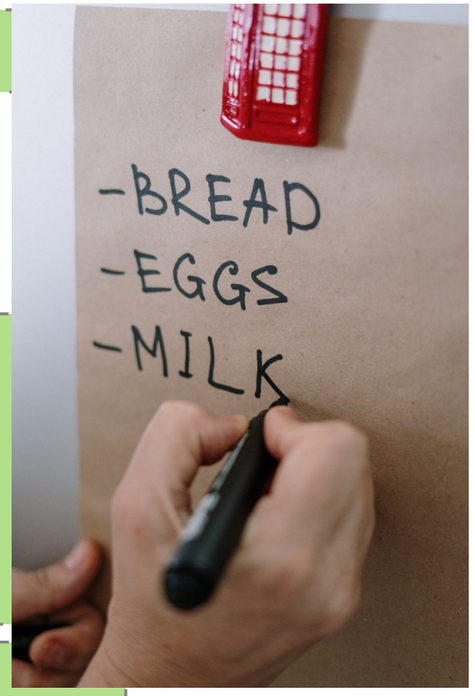


III. Liste

(doppelt verkettet)

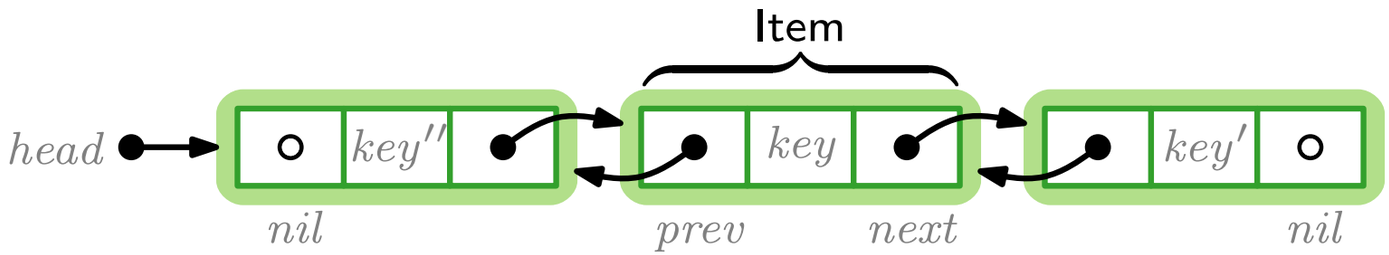


Operation	Implementierung
List()	$head = nil$
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $\quad x = x.next$ **return** x
ptr Insert(key k)	$x =$ **new** Item(k, $head$) **return** x

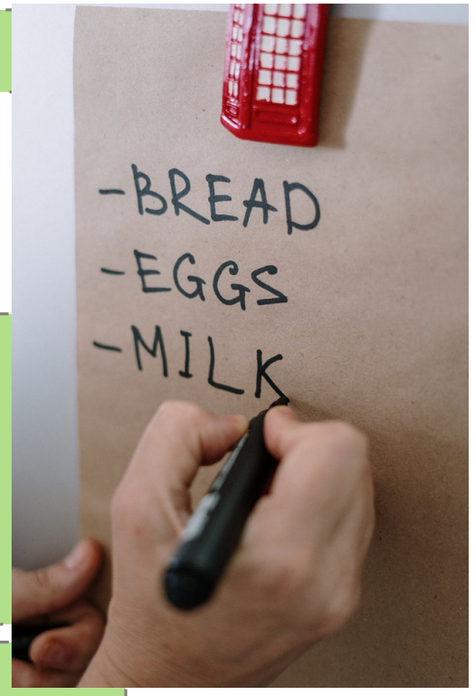


III. Liste

(doppelt verkettet)

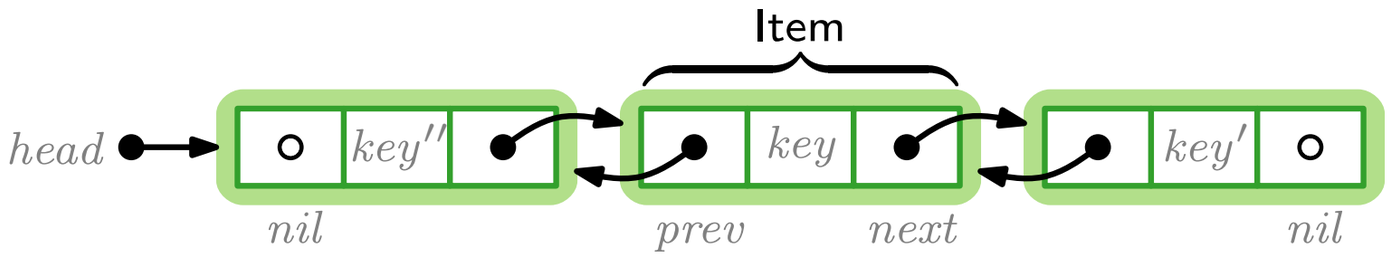


Operation	Implementierung
List()	$head = nil$
	Item(key k, ptr p) — Konstruktor
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $x = x.next$ **return** x
ptr Insert(key k)	$x =$ **new** Item(k, $head$) **return** x

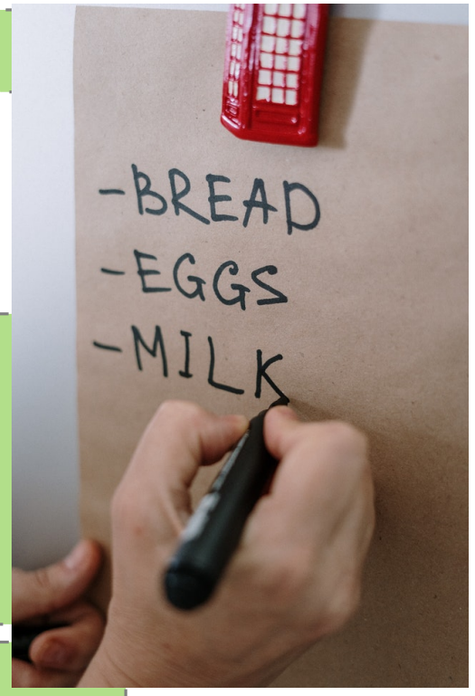


III. Liste

(doppelt verkettet)

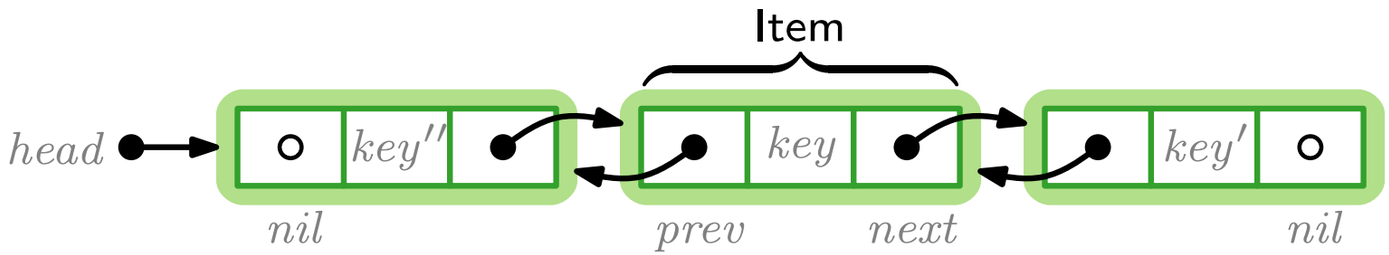


Operation	Implementierung
List()	$head = nil$
	Item(key k, ptr p) $key = k$, $next = p$, $prev = nil$ — Konstruktor
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $x = x.next$ **return** x
ptr Insert(key k)	$x =$ **new** Item(k, $head$) **return** x

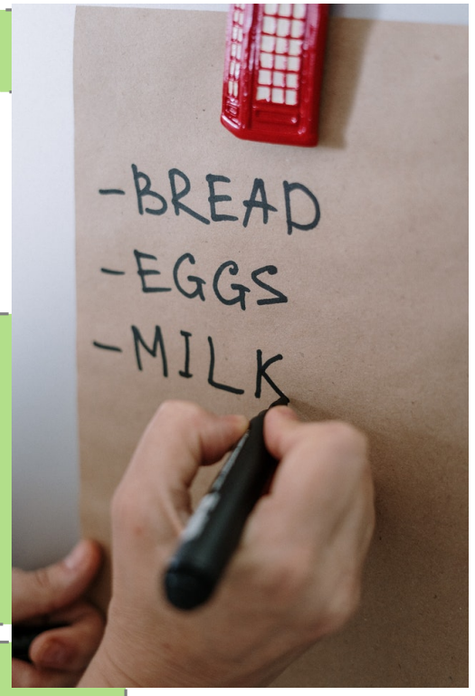


III. Liste

(doppelt verkettet)

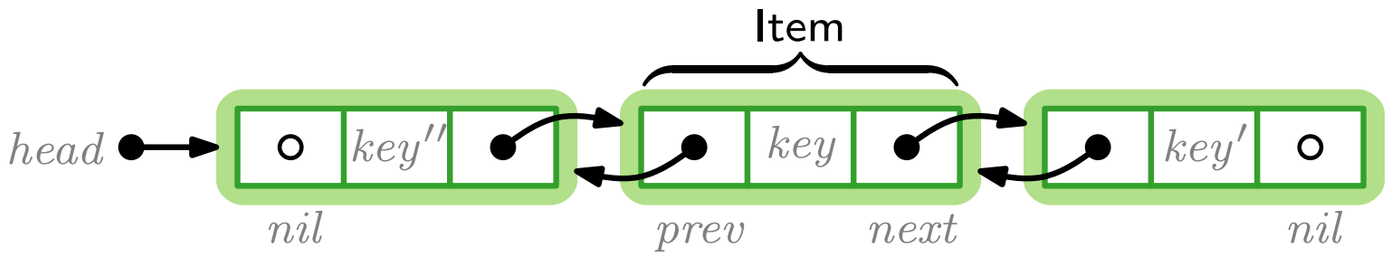


Operation	Implementierung
List()	$head = nil$
	Item(key k, ptr p) $key = k$, $next = p$, $prev = nil$ — Konstruktor
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $x = x.next$ **return** x
ptr Insert(key k)	$x =$ **new** Item(k, $head$) **if** $head \neq nil$ **then** $head.prev = x$ **return** x

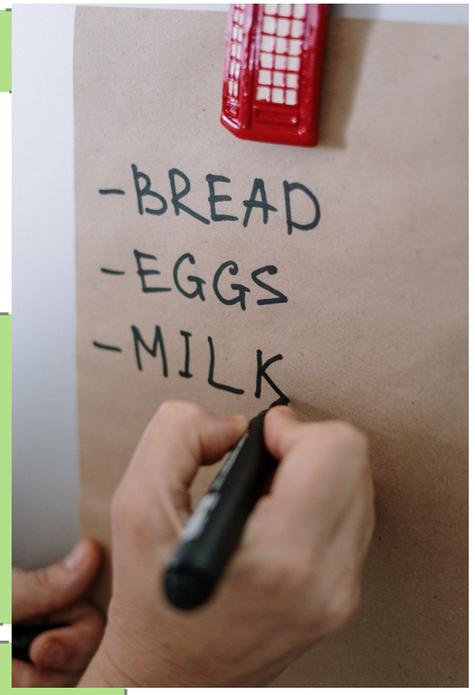


III. Liste

(doppelt verkettet)

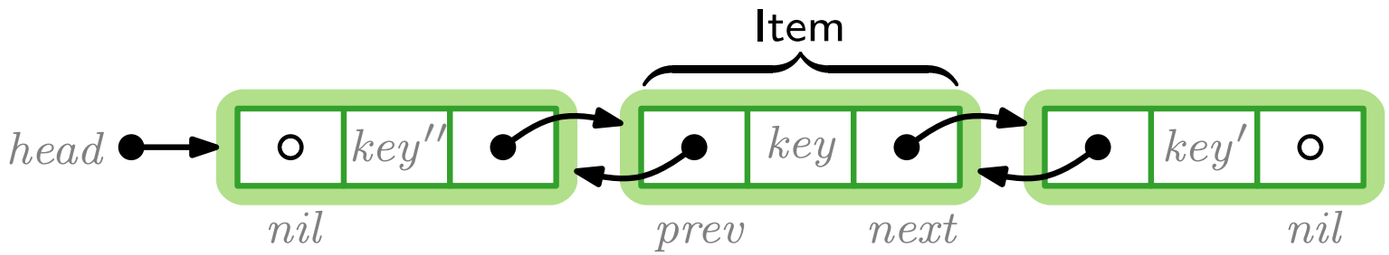


Operation	Implementierung
List()	$head = nil$
	Item(key k, ptr p) $key = k$, $next = p$, $prev = nil$ — Konstruktor
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $x = x.next$ **return** x
ptr Insert(key k)	$x =$ **new** Item(k, $head$) **if** $head \neq nil$ **then** $head.prev = x$ $head = x$ **return** x



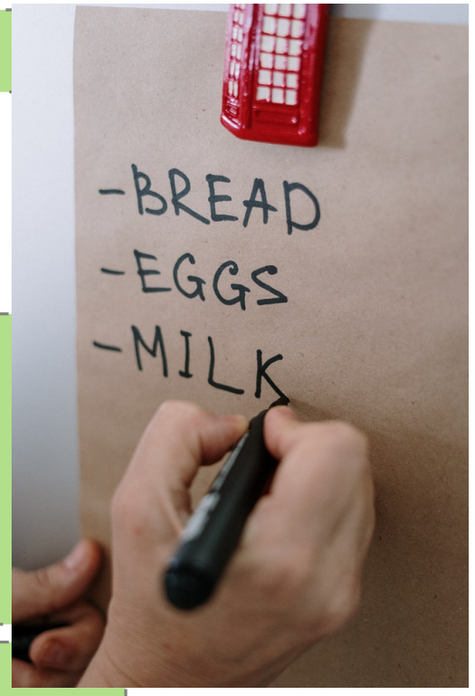
III. Liste

(doppelt verkettet)



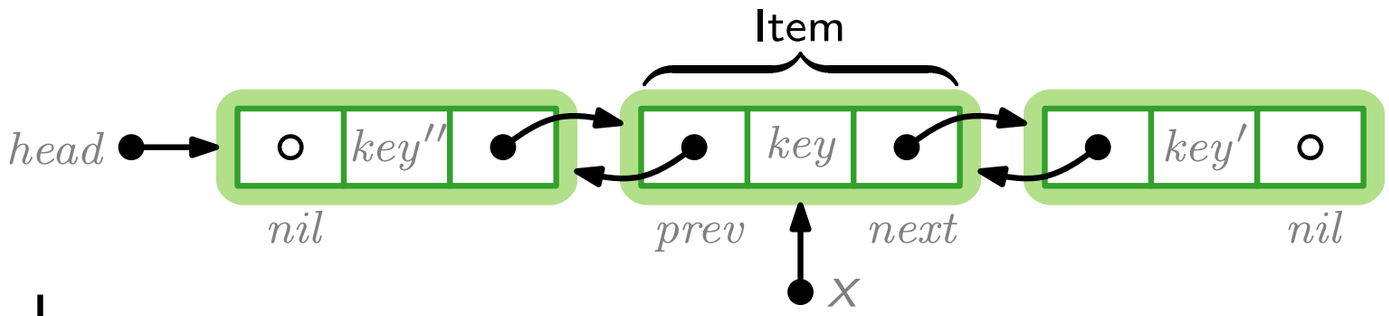
Operation	Implementierung
List()	$head = nil$
	Item(key k, ptr p) $key = k$, $next = p$, $prev = nil$ — Konstruktor
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $x = x.next$ **return** x
ptr Insert(key k)	$x =$ **new** Item(k, $head$) **if** $head \neq nil$ **then** $head.prev = x$ $head = x$ **return** x

Aufgabe.
Schreiben Sie
Delete(ptr k)



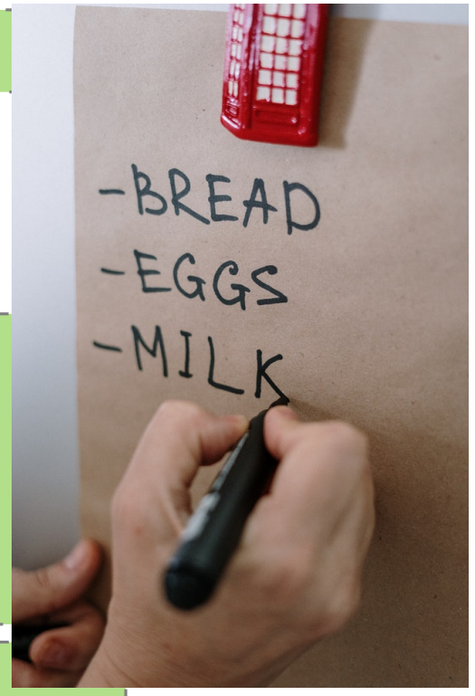
III. Liste

(doppelt verkettet)



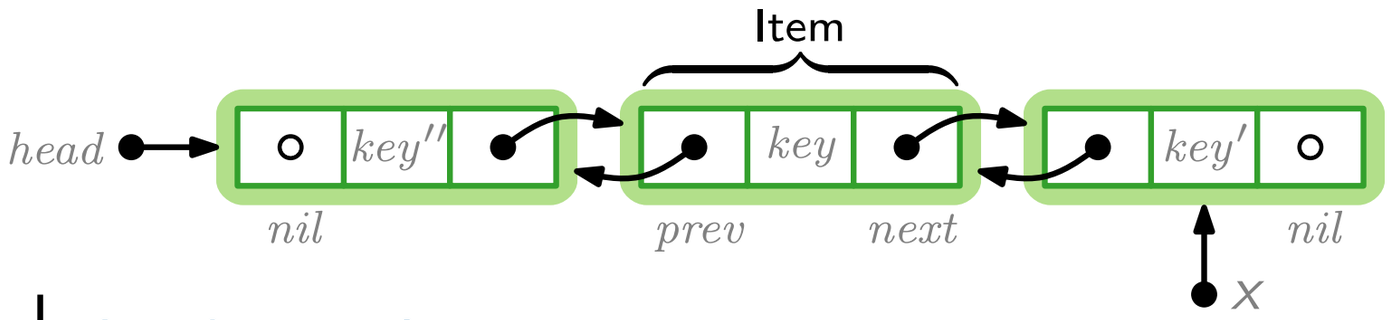
Operation	Implementierung
List()	$head = nil$
	Item(key k, ptr p) $key = k$, $next = p$, $prev = nil$ — Konstruktor
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $x = x.next$ **return** x
ptr Insert(key k)	$x =$ **new** Item(k, $head$) **if** $head \neq nil$ **then** $head.prev = x$ $head = x$ **return** x

Aufgabe.
Schreiben Sie
Delete(ptr k)



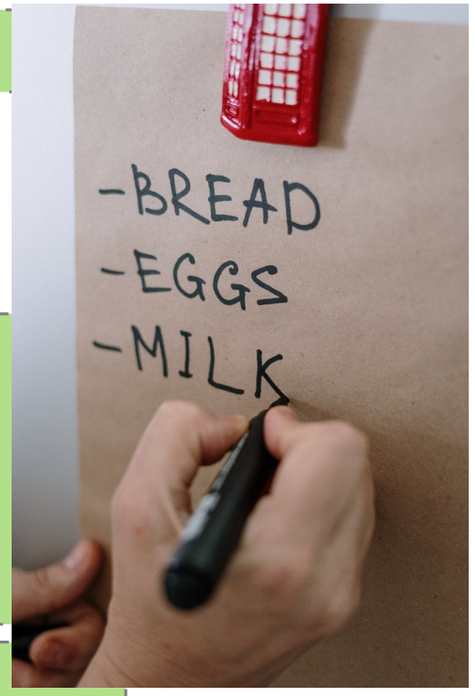
III. Liste

(doppelt verkettet)



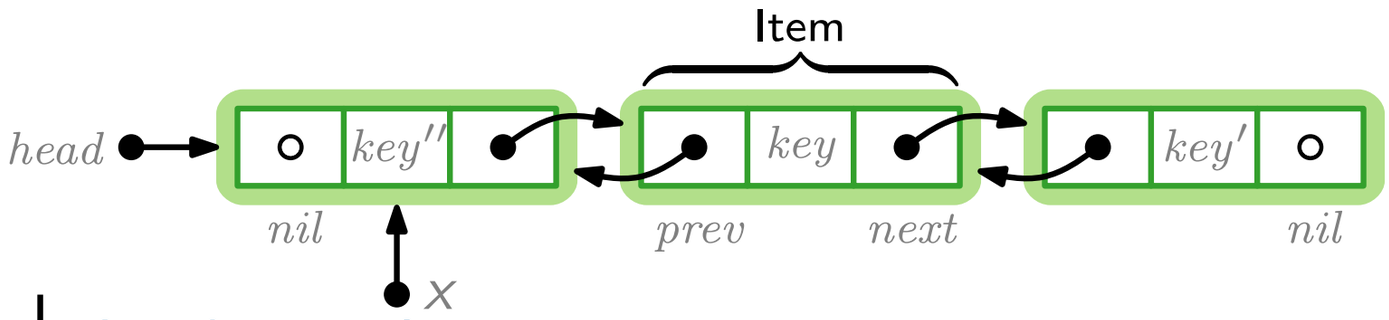
Operation	Implementierung
List()	$head = nil$
	Item(key k, ptr p) $key = k$, $next = p$, $prev = nil$ — Konstruktor
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $x = x.next$ **return** x
ptr Insert(key k)	$x =$ **new** Item(k, $head$) **if** $head \neq nil$ **then** $head.prev = x$ $head = x$ **return** x

Aufgabe.
Schreiben Sie
Delete(ptr k)



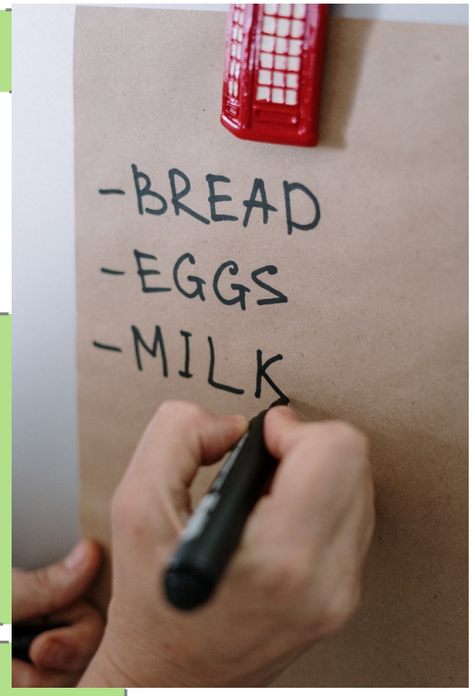
III. Liste

(doppelt verkettet)



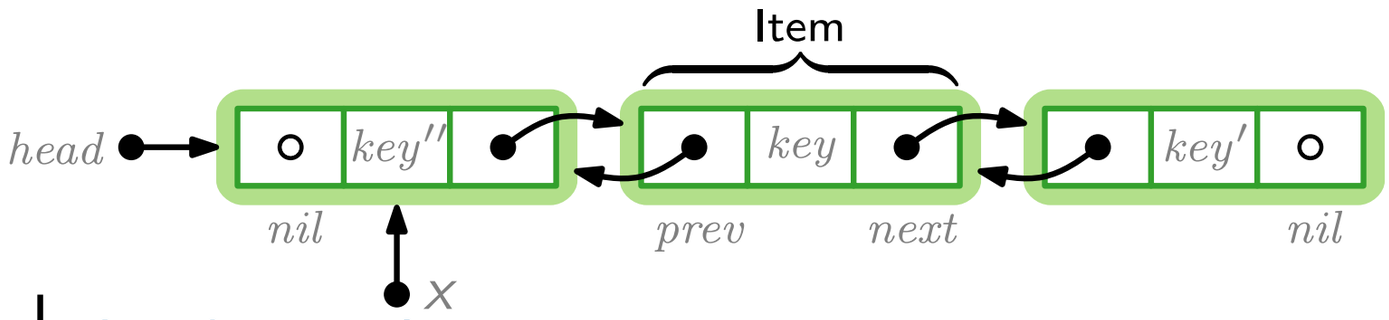
Operation	Implementierung
List()	$head = nil$
	Item(key k, ptr p) $key = k$, $next = p$, $prev = nil$ — Konstruktor
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $x = x.next$ **return** x
ptr Insert(key k)	$x =$ **new** Item(k, $head$) **if** $head \neq nil$ **then** $head.prev = x$ $head = x$ **return** x

Aufgabe.
Schreiben Sie
Delete(ptr k)

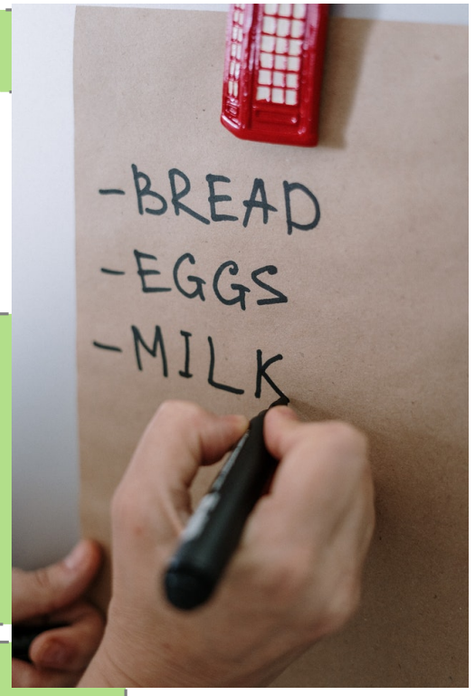


III. Liste

(doppelt verkettet)

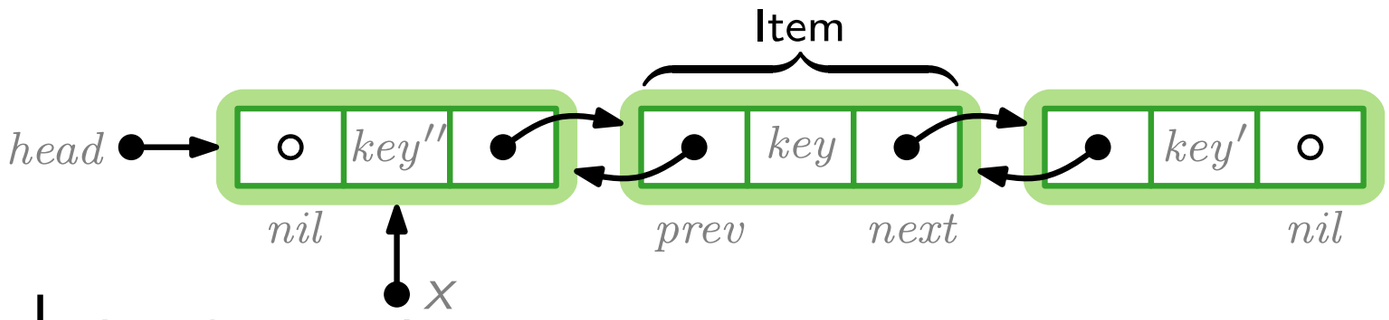


Operation	Implementierung
List()	$head = nil$
	Item(key k, ptr p) $key = k$, $next = p$, $prev = nil$ — Konstruktor
Laufzeiten?	
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $x = x.next$ **return** x
ptr Insert(key k)	$x =$ **new** Item(k, $head$) **if** $head \neq nil$ **then** $head.prev = x$ $head = x$ **return** x

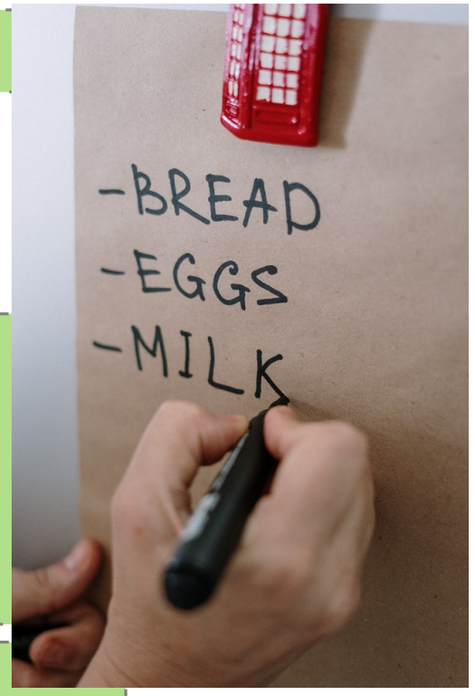


III. Liste

(doppelt verkettet)

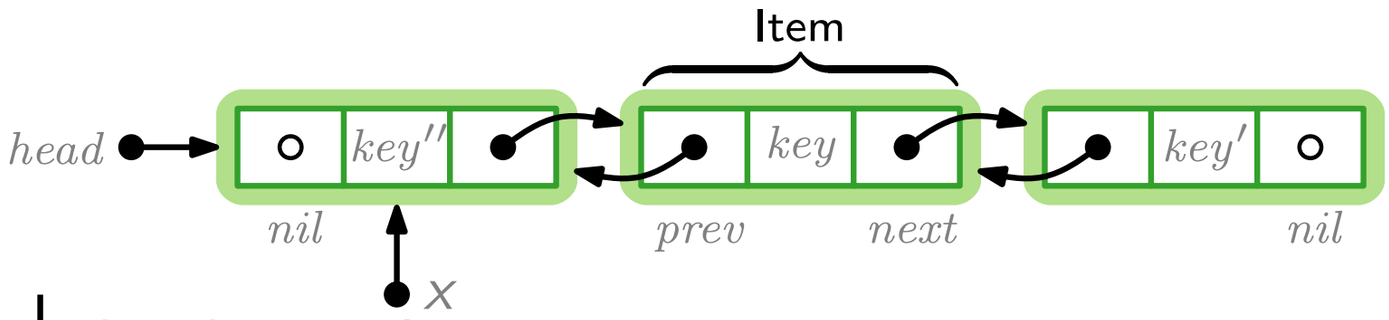


Operation	Implementierung
List()	$head = nil$
	Item(key k, ptr p) $key = k$, $next = p$, $prev = nil$ — Konstruktor
Laufzeiten?	
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $x = x.next$ **return** x
ptr Insert(key k)	$x =$ **new** Item(k, $head$) **if** $head \neq nil$ **then** $head.prev = x$ $head = x$ **return** x
Delete(ptr x)	



III. Liste

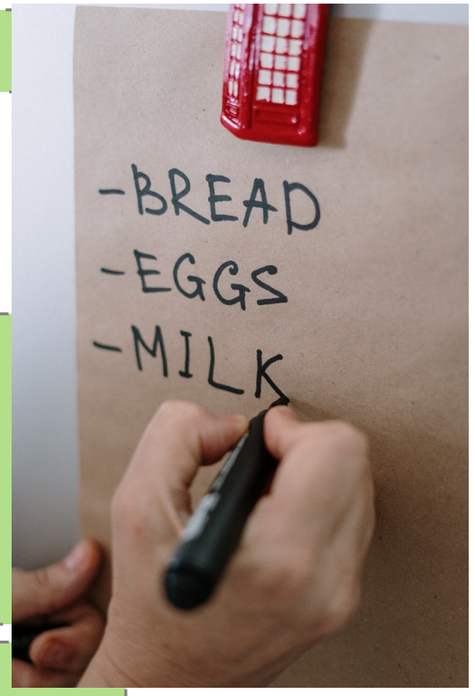
(doppelt verkettet)



Operation	Implementierung
List() $\mathcal{O}(1)$	$head = nil$
Item(key k, ptr p)	$key = k$ $next = p$ $prev = nil$
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $\quad x = x.next$ **return** x
ptr Insert(key k)	$x =$ **new** Item(k, $head$) **if** $head \neq nil$ **then** $head.prev = x$ $head = x$ **return** x
Delete(ptr x)	

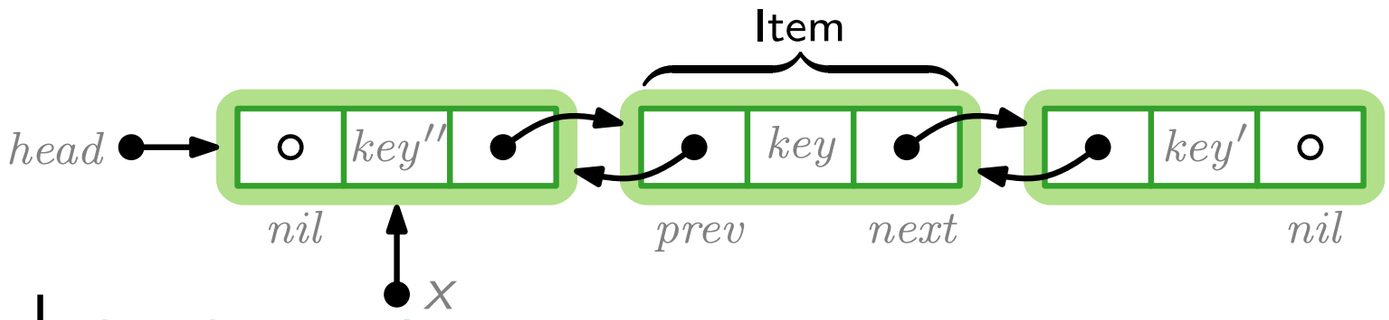
Laufzeiten?

Konstruktor

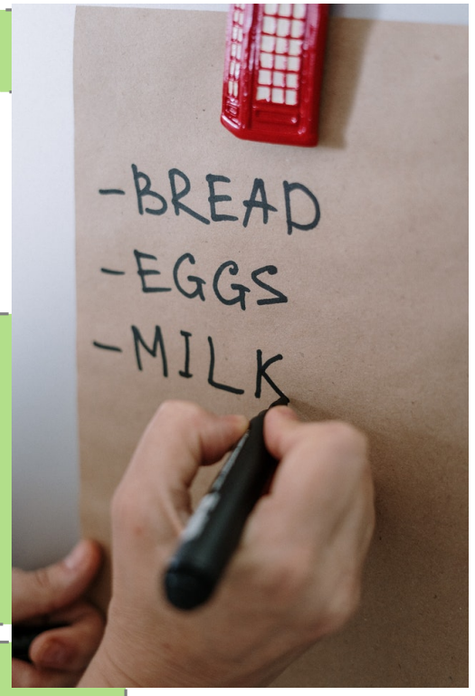


III. Liste

(doppelt verkettet)

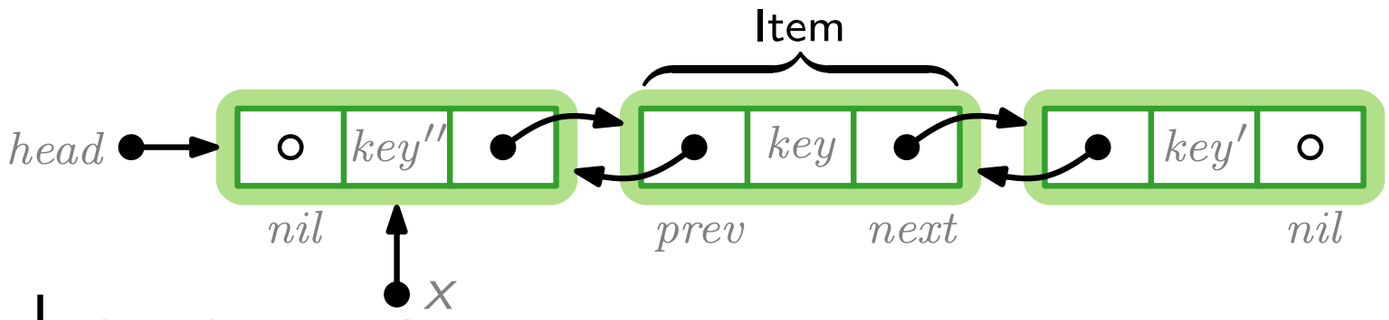


Operation		Implementierung
List()	$\mathcal{O}(1)$	$head = nil$
		Item(key k, ptr p) $key = k$, $next = p$, $prev = nil$ — **Konstruktor**
Laufzeiten?		
ptr Search(key k)	$\mathcal{O}(n)$	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $x = x.next$ **return** x
ptr Insert(key k)		$x =$ **new** Item(k, $head$) **if** $head \neq nil$ **then** $head.prev = x$ $head = x$ **return** x
Delete(ptr x)		



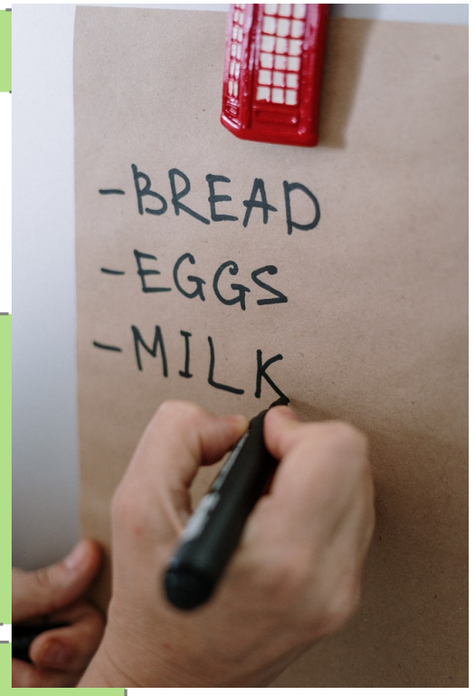
III. Liste

(doppelt verkettet)



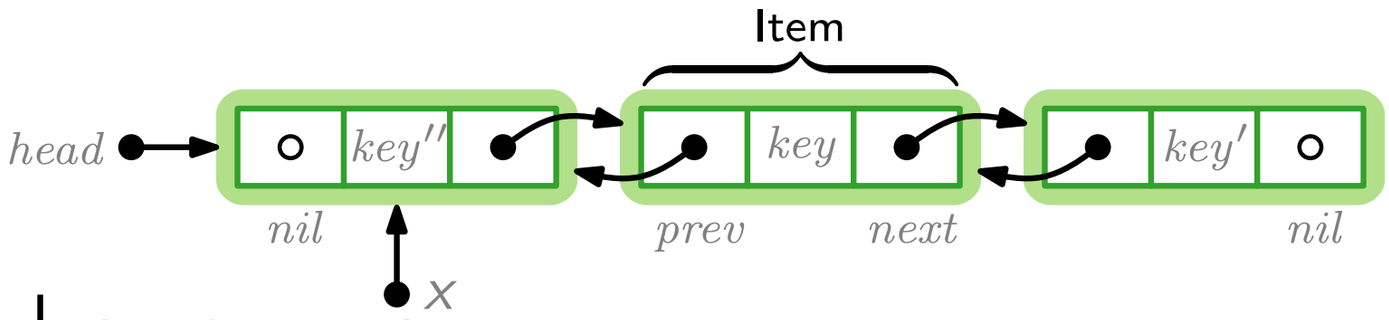
Operation	Implementierung
List() $\mathcal{O}(1)$	$head = nil$
Item(key k, ptr p)	$key = k$ $next = p$ $prev = nil$ Konstruktor
ptr Search(key k) $\mathcal{O}(n)$	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $\quad x = x.next$ **return** x
ptr Insert(key k) $\mathcal{O}(1)$	$x =$ **new** Item(k, $head$) **if** $head \neq nil$ **then** $head.prev = x$ $head = x$ **return** x
Delete(ptr x)	

Laufzeiten?

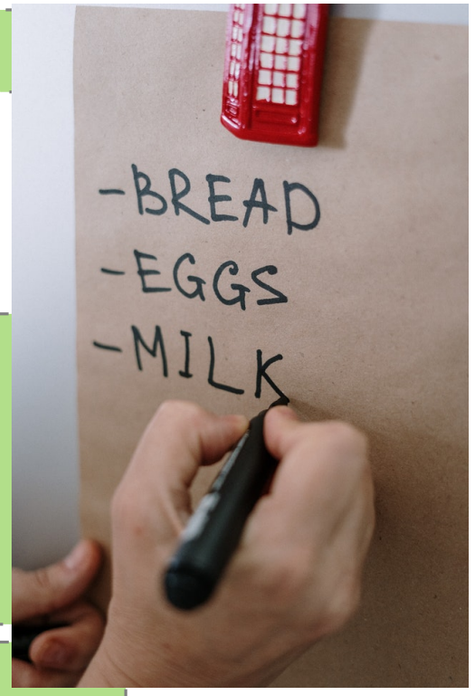


III. Liste

(doppelt verkettet)

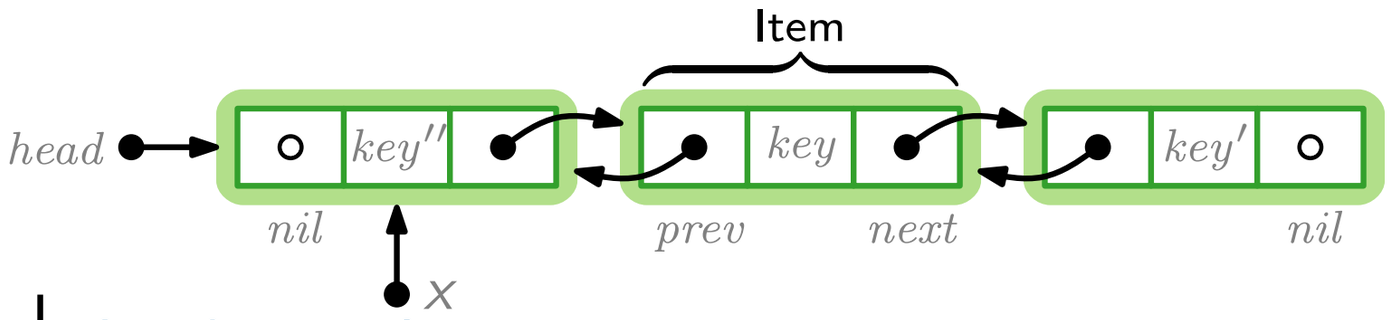


Operation		Implementierung
List()	$\mathcal{O}(1)$	$head = nil$
		Item(key k, ptr p) $key = k$, $next = p$, $prev = nil$ — **Konstruktor**
Laufzeiten?		
ptr Search(key k)	$\mathcal{O}(n)$	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $x = x.next$ **return** x
ptr Insert(key k)	$\mathcal{O}(1)$	$x =$ **new** Item(k, $head$) **if** $head \neq nil$ **then** $head.prev = x$ $head = x$ **return** x
Delete(ptr x)	$\mathcal{O}(1)$	



III. Liste

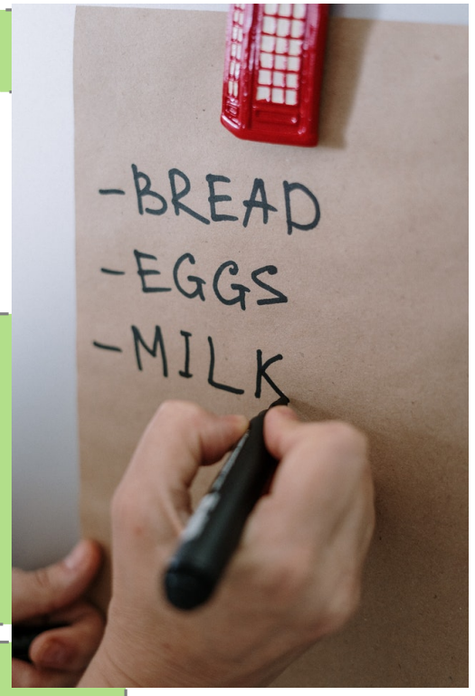
(doppelt verkettet)



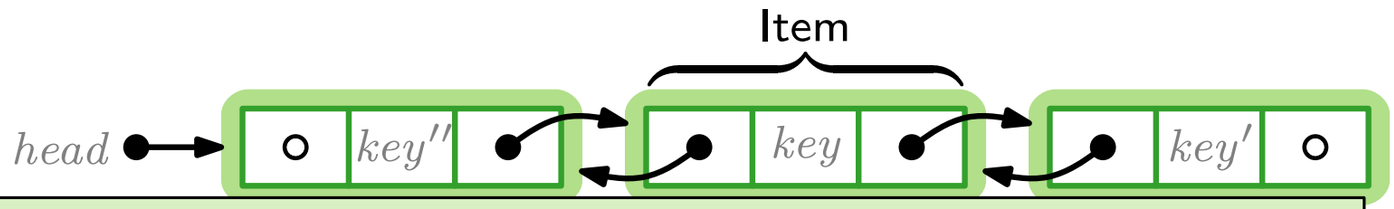
Operation		Implementierung
List()	$\mathcal{O}(1)$	$head = nil$
Item(key k, ptr p)		$key = k$ $next = p$ $prev = nil$
ptr Search(key k)	$\mathcal{O}(n)$	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $\quad x = x.next$ **return** x
ptr Insert(key k)	$\mathcal{O}(1)$	$x =$ **new** Item(k, $head$) **if** $head \neq nil$ **then** $head.prev = x$ $head = x$ **return** x
Delete(ptr x)	$\mathcal{O}(1)$	

Laufzeiten?

Konstruktor

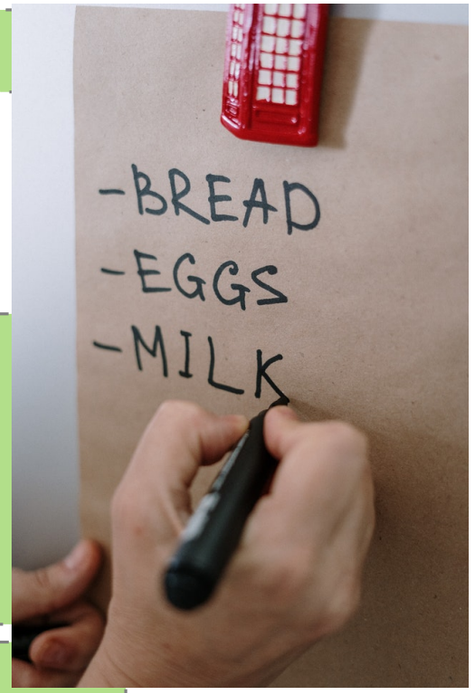


III. Liste




```
public class PKListDemo {

  static class Item<T> {
     T key;
     Item next;
     Item prev;

     // Konstruktor
     Item(T k, Item p) {
        key = k;
        next = p;
        prev = null;
     }
  }

  static class PKList<T> {

     public Item head;

     // Konstruktor
     public PKList() {
       head = null;
     }

     public boolean isEmpty() {
       return (head == null);
     }

     public Item search (T k) {
       Item x = head;
       while (x != null && x.key != k) {
         x = x.next;
       }
       return x;
     }

     public Item insert (T k) {
       Item x = new Item<T>(k, head);
       if (head != null) {
         head.prev = x;
       }
       head = x;
       return x;
     }

     public void delete (Item k) {
       if (k.prev == null) {
         // k ist head
         head = k.next;
       } else {
         // k.prev != null
         k.prev.next = k.next;
       }
       if (k.next != null) {
         k.next.prev = k.prev;
       }
     }
  }

  public static void main(String[] args) {
     int[] A = {4, 8, 15, 16, 23, 42};
     PKList<Integer> list = new PKList<>();

     for (int i = 0; i < A.length; i++) {
       list.insert(A[i]);
     }

     Item item = list.search(16);
     list.delete(item);
  }
}
```

Konstruktor



Übersicht Elementare Datenstrukturen

Operationen	Stapel	Schlange	Liste
Einfügen			
Entfernen			
weitere Oper.			

außer Konstruktor und EMPTY()

Übersicht Elementare Datenstrukturen

Operationen	Stapel	Schlange	Liste
Einfügen	Push()		
Entfernen	Pop()		
weitere Oper.	Top()		

außer Konstruktor und Empty()

Übersicht Elementare Datenstrukturen

Operationen	Stapel	Schlange	Liste
Einfügen	Push()	Enqueue()	
Entfernen	Pop()	Dequeue()	
weitere Oper.	Top()		

außer Konstruktor und Empty()

Übersicht Elementare Datenstrukturen

Operationen	Stapel	Schlange	Liste
Einfügen	Push()	Enqueue()	
Entfernen	Pop()	Dequeue()	
weitere Oper.	Top()	Head() Tail()	

außer Konstruktor und Empty()

Übersicht Elementare Datenstrukturen

Operationen	Stapel	Schlange	Liste
Einfügen	Push()	Enqueue()	Insert()
Entfernen	Pop()	Dequeue()	Delete()
weitere Oper.	Top()	Head() Tail()	Search()

außer Konstruktor und Empty()

Übersicht Elementare Datenstrukturen

Operationen	Stapel	Schlange	Liste
Einfügen ■ Einschränkung	Push()	Enqueue()	Insert()
Entfernen ■ Einschränkung	Pop()	Dequeue()	Delete()
weitere Oper. außer Konstruktor und Empty()	Top()	Head() Tail()	Search()

Übersicht Elementare Datenstrukturen

Operationen	Stapel	Schlange	Liste
Einfügen ■ Einschränkung	Push() nur oben	Enqueue()	Insert()
Entfernen ■ Einschränkung	Pop() nur oben	Dequeue()	Delete()
weitere Oper. außer Konstruktor und Empty()	Top()	Head() Tail()	Search()

Übersicht Elementare Datenstrukturen

Operationen	Stapel	Schlange	Liste
Einfügen	Push()	Enqueue()	Insert()
■ Einschränkung	nur oben	nur hinten	
Entfernen	Pop()	Dequeue()	Delete()
■ Einschränkung	nur oben	nur vorne	
weitere Oper.	Top()	Head() Tail()	Search()

außer Konstruktor und Empty()

Übersicht Elementare Datenstrukturen

Operationen	Stapel	Schlange	Liste
Einfügen	Push()	Enqueue()	Insert()
■ Einschränkung	nur oben	nur hinten	nur vorne
Entfernen	Pop()	Dequeue()	Delete()
■ Einschränkung	nur oben	nur vorne	beliebig
weitere Oper. (außer Konstruktor und Empty())	Top()	Head() Tail()	Search()

Übersicht Elementare Datenstrukturen

Operationen	**Stapel**	**Schlange**	**Liste**
Einfügen	Push()	Enqueue()	Insert()
■ Einschränkung	nur oben	nur hinten	(nur vorne) beliebig
Entfernen	Pop()	Dequeue()	Delete()
■ Einschränkung	nur oben	nur vorne	beliebig
weitere Oper.	Top()	Head() Tail()	Search()

außer Konstruktor und Empty()

Übersicht Elementare Datenstrukturen

Operationen	Stapel	Schlange	Liste
Einfügen	Push()	Enqueue()	Insert()
■ Einschränkung	nur oben	nur hinten	(nur vorne) beliebig
Entfernen	Pop()	Dequeue()	Delete()
■ Einschränkung	nur oben	nur vorne	beliebig
weitere Oper.	Top()	Head() Tail()	Search()

außer Konstruktor und Empty()

Alle hier aufgelisteten Operationen außer Search() laufen in $\mathcal{O}(1)$ Zeit.

Übersicht Elementare Datenstrukturen

Operationen	Stapel	Schlange	Liste
Einfügen	Push()	Enqueue()	Insert()
■ Einschränkung	nur oben	nur hinten	(nur vorne) beliebig
Entfernen	Pop()	Dequeue()	Delete()
■ Einschränkung	nur oben	nur vorne	beliebig
weitere Oper. (außer Konstruktor und Empty())	Top()	Head() Tail()	Search()

Alle hier aufgelisteten Operationen außer Search() laufen in $\mathcal{O}(1)$ Zeit.

Listen sind mächtiger als Stapel/Schlangen. Wozu also Stapel/Schlangen?