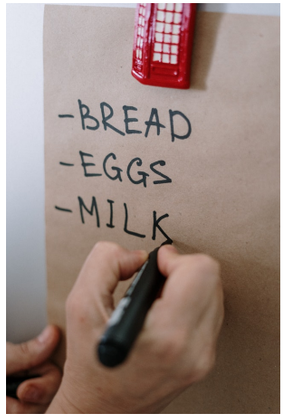
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

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.

Abstrakter Datentyp.

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

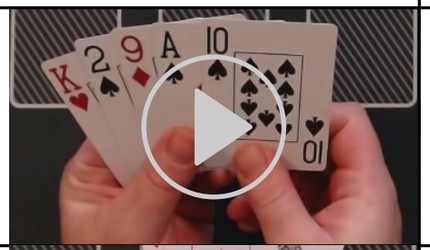
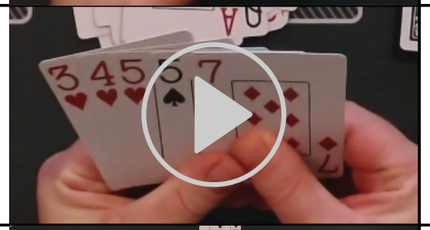
Implementierung.

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.

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.

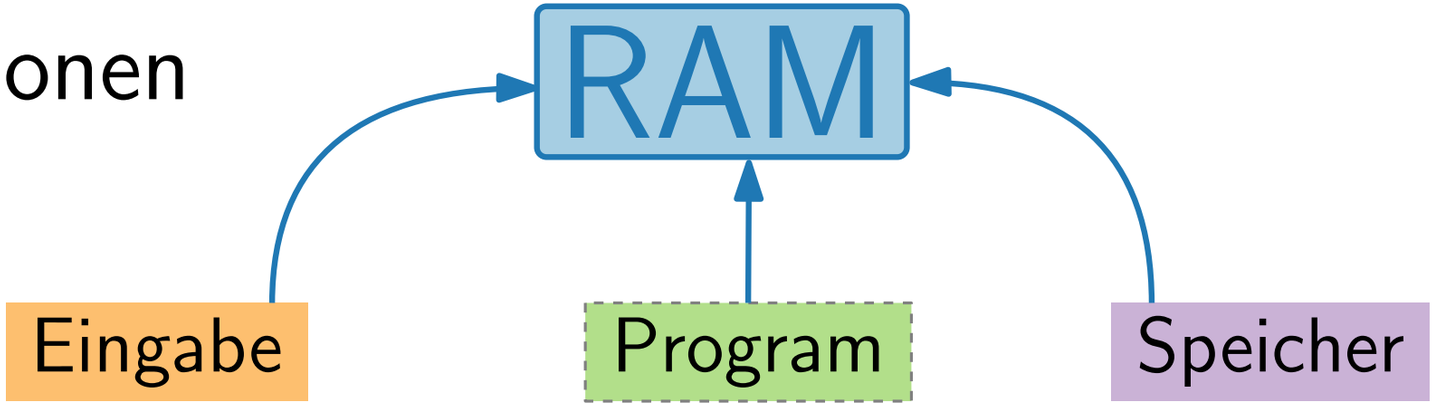
eindeutige Zahl

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

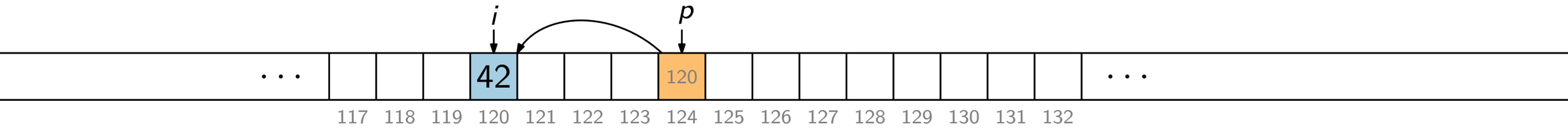


Der Speicher wird in 5 Regionen unterteilt:

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).



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*

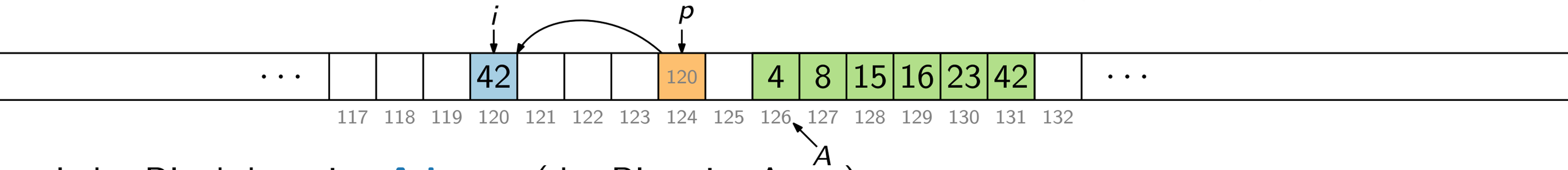
124

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? — 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

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

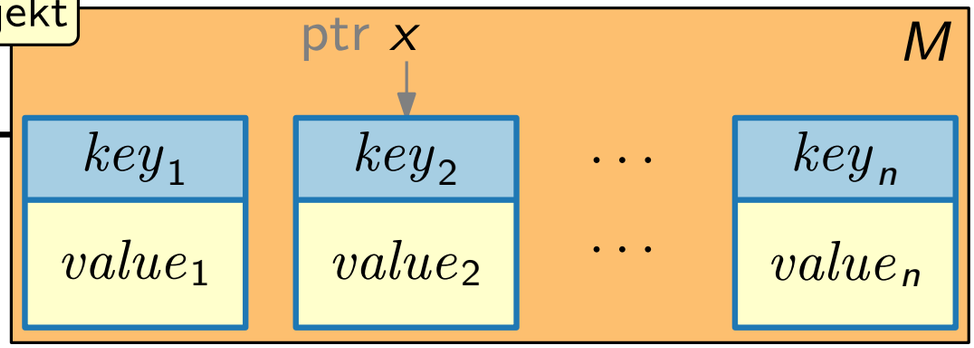
Abstrakter Datentyp: Dynamische Menge

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

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

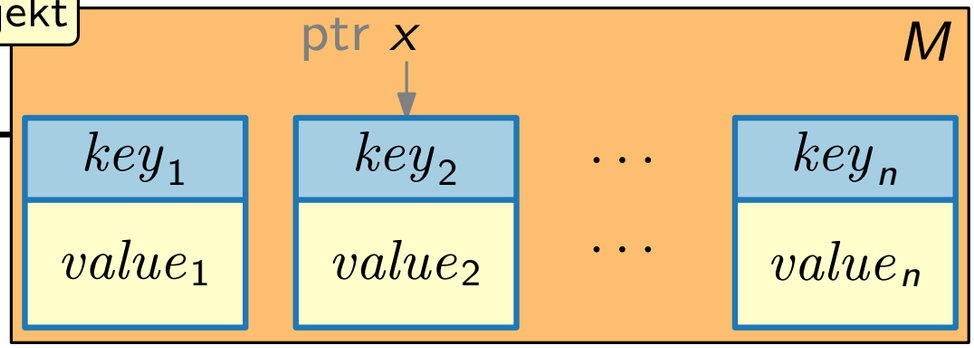
Abstrakter Datentyp: Dynamische Menge

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)	$M = M \setminus \{x\}$
DELETE(ptr x)	
ptr SEARCH(key k)	
ptr MINIMUM()	
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer



Zurück zu dynamischen Mengen

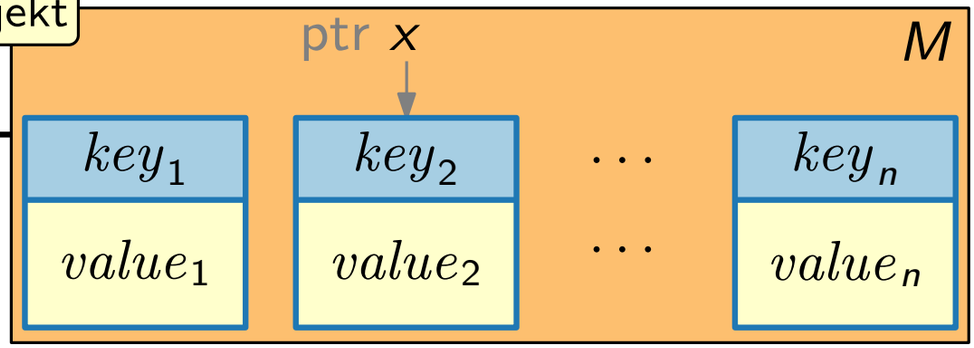
Abstrakter Datentyp: Dynamische Menge

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)	**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

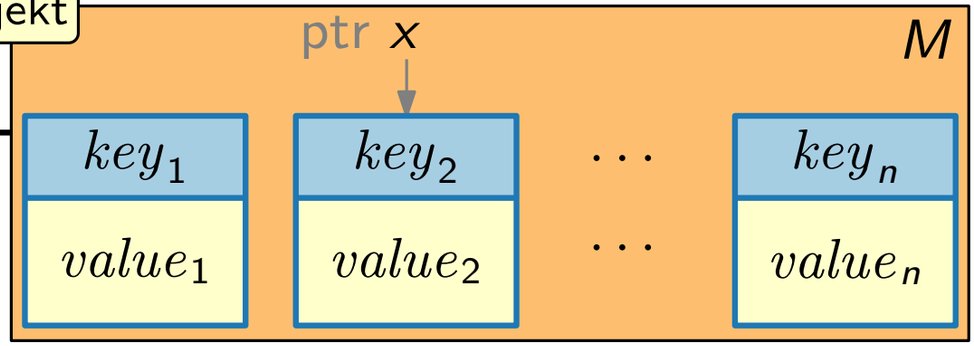


Zurück zu dynamischen Mengen

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()	**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

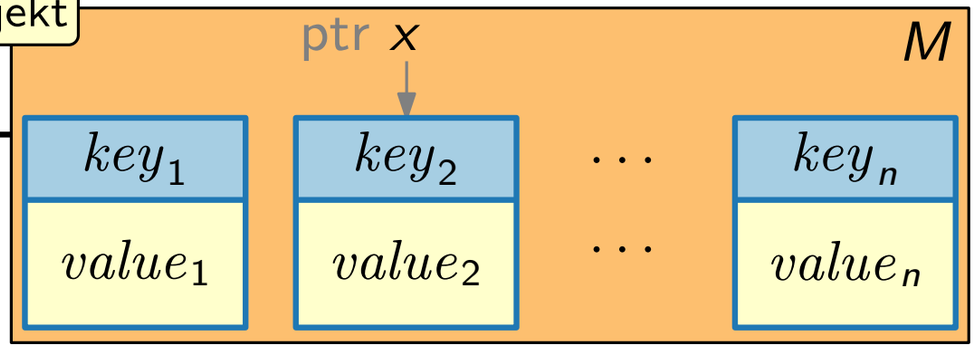
Abstrakter Datentyp: Dynamische Menge

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

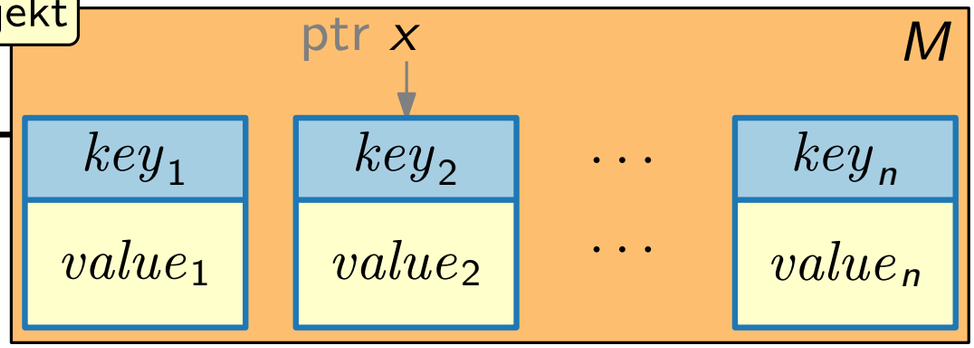


Zurück zu dynamischen Mengen

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)	$M' = \{(k, v) \in M \mid k < x.key\}$ **return** M'.MAXIMUM()
ptr SUCCESSOR(ptr x)	

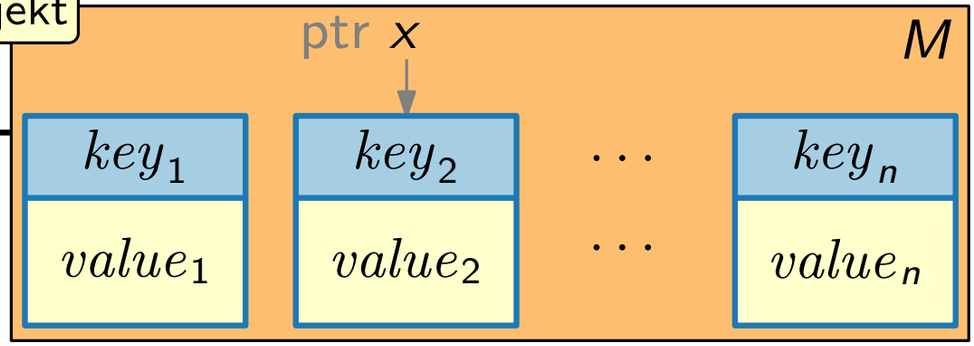
Pointer

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

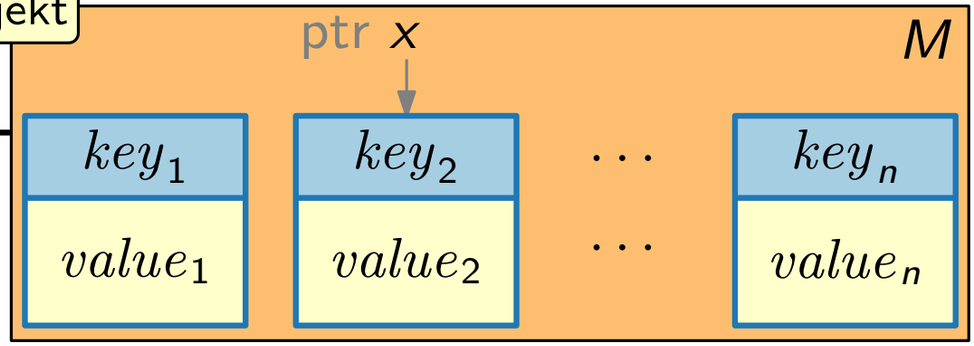
Abstrakter Datentyp: Dynamische Menge

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

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

Zurück zu dynamischen Mengen

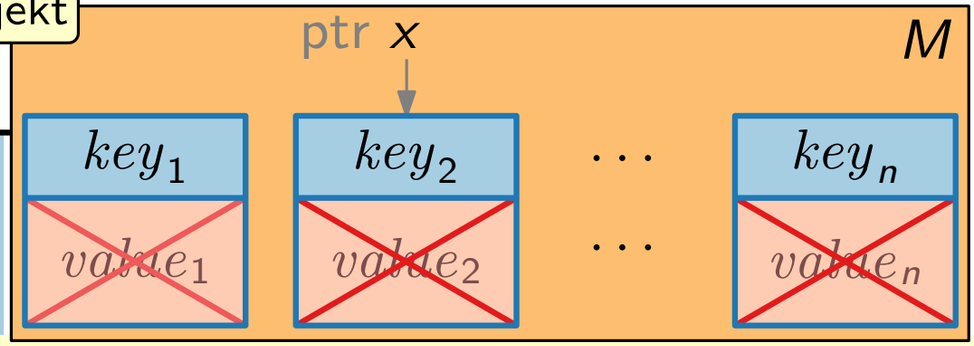
Abstrakter Datentyp: Dynamische Menge

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)	**Veränderungen**
DELETE(ptr x)	
ptr SEARCH(key k) (Wörterbuch)	
ptr MINIMUM()	**Anfragen**
ptr MAXIMUM()	
ptr PREDECESSOR(ptr x)	
ptr SUCCESSOR(ptr x)	

Pointer



Vereinfachung

Implementierung Wörterbuch

Wörterbuch.

Spezialfall einer dynamischen Menge

Abstrakter Datentyp.

stellt folgende Operationen bereit:
INSERT, DELETE, SEARCH

Implementierung.

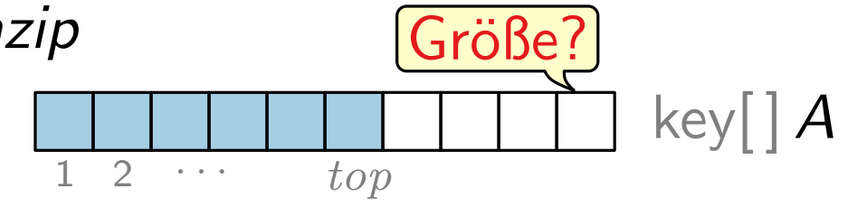
heute: 3 Varianten

Stapel, Schlange und Liste

I. Stapel

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) **Konstruktor**	$A =$ **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** $A[top]$

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

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

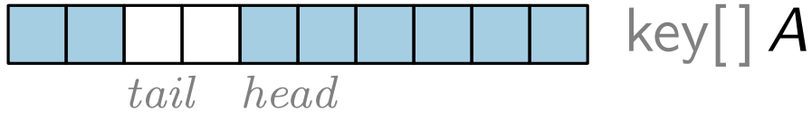
algo.uni-trier.de/demos/memvis.html

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

II. Schlange

first-in first-out

verwaltet sich ändernde Menge nach *FIFO-Prinzip*



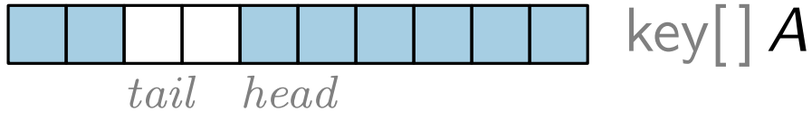
Operation	Implementierung
QUEUE(int n) **Konstruktor**	$A =$ **new** key$[1 \ldots n+1]$ $tail = head = 1$
boolean EMPTY()	**if** $head == tail$ **then return** *true* **else return** *false*
ENQUEUE(key k) stell neues Element an den Schwanz der Schlange an	$A[tail] = k$ **if** $tail == A.length$ **then**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



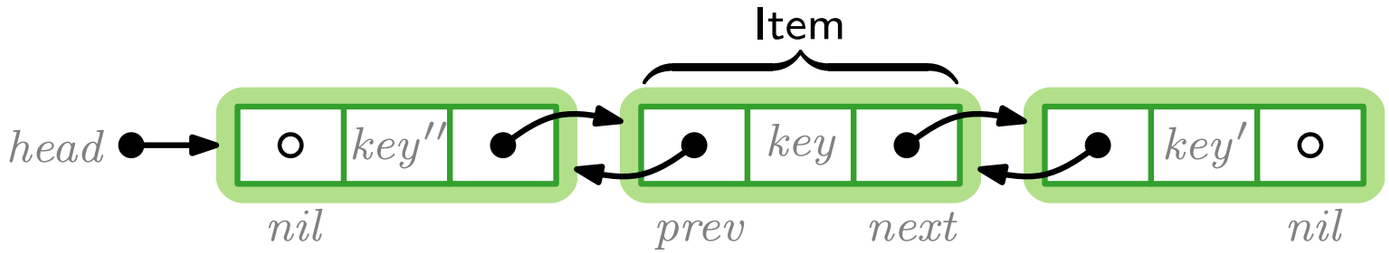
Operation	Implementierung
QUEUE(int n) **Konstruktor**	$A =$ **new*** key$[1 \ldots n+1]$ $tail = head = 1$
boolean EMPTY()	**if** $head == tail$ **then return** *true* **else return** *false*
ENQUEUE(key k) stell neues Element an den Schwanz der Schlange an	$A[tail] = k$ **if** $tail == A.length$ **then**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

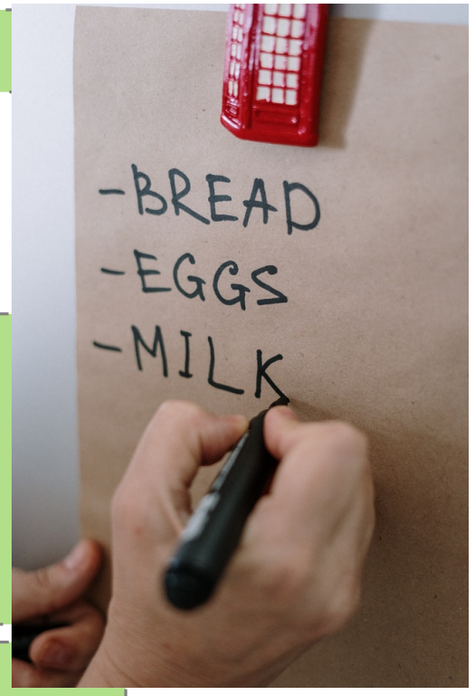
Laufzeiten?
Alle* $\mathcal{O}(1)$.

III. Liste

(doppelt verkettet)

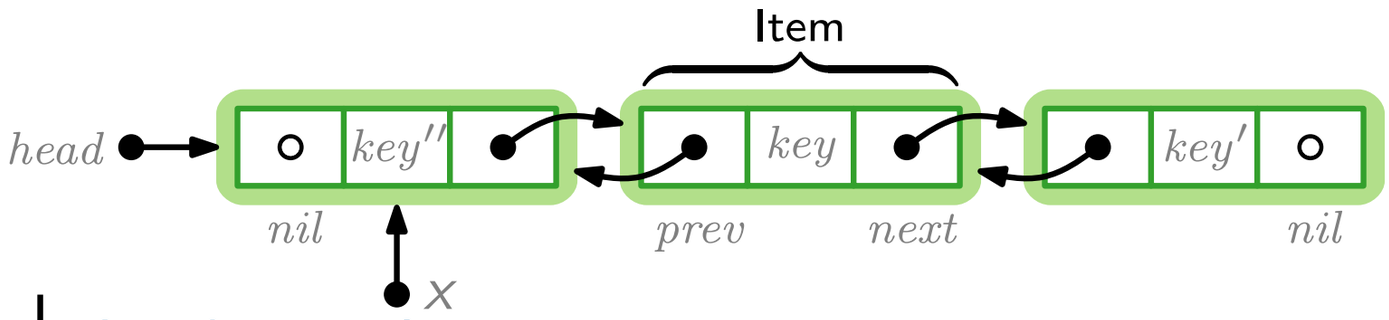


Operation	Implementierung
List()	$head = nil$
ptr Search(key k)	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $\quad x = x.next$ **return** x
ptr Insert(key k)	$x =$ **new** Item(k, $head$) **return** x



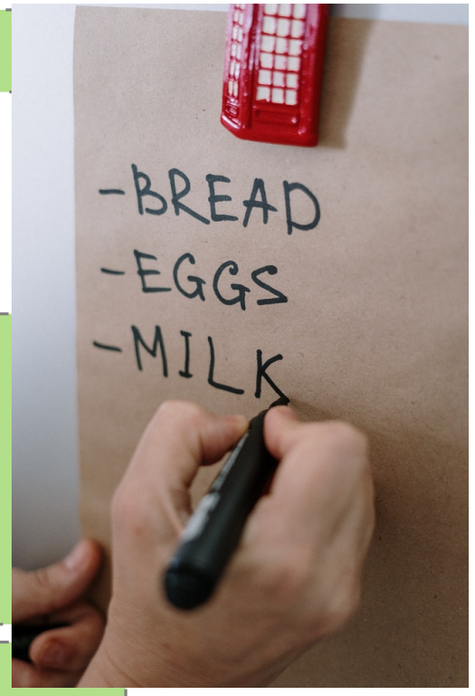
III. Liste

(doppelt verkettet)

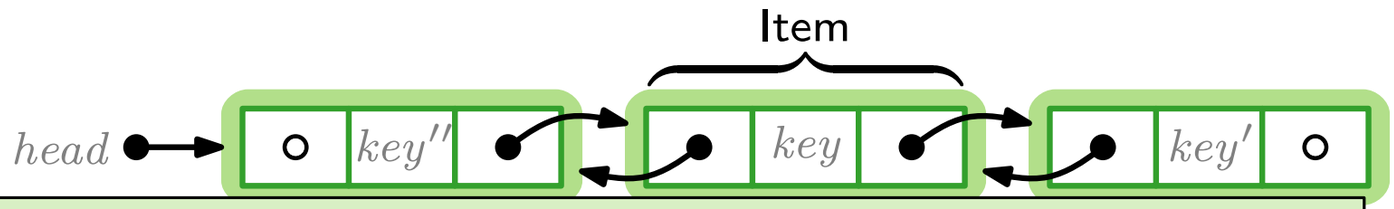


Operation		Implementierung
List()	$\mathcal{O}(1)$	$head = nil$
Item(key k, ptr p)		$key = k$ $next = p$ $prev = nil$ — Konstruktor
ptr Search(key k)	$\mathcal{O}(n)$	$x = head$ **while** $x \neq nil$ **and** $x.key \neq k$ **do** $x = x.next$ **return** x
ptr Insert(key k)	$\mathcal{O}(1)$	$x =$ **new**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

Laufzeiten?

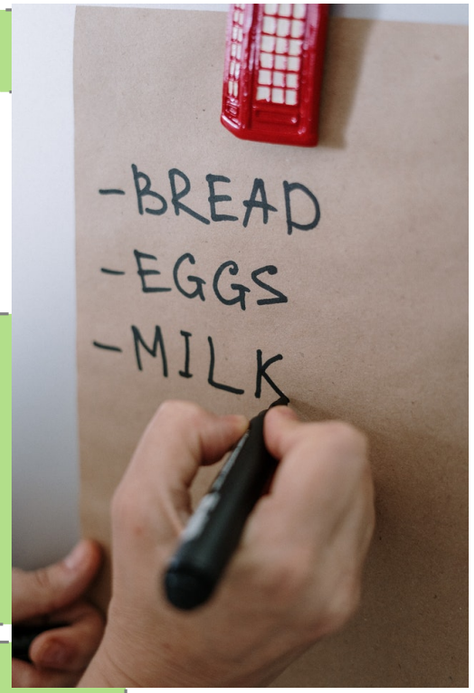


III. Liste




```
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	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?