

Algorithmen und Datenstrukturen

Wintersemester 2023/24

11. Vorlesung

Elementare Datenstrukturen:
Stapel + Schlange + Liste

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?

Beispiel

Prioritätsschlange:

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

Abstrakter Datentyp

$O(1)$ stellt folgende Operationen bereit: $O(n)$
Insert, FindMax, ExtractMax, IncreaseKey

Implementierung 1

- Daten werden in einem Feld (oder Liste) gespeichert
- neue Elemente werden hinten angehängt (unsortiert)
- Maximum wird immer aufrechterhalten

Beispiel

Prioritätsschlange:

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

Abstrakter Datentyp

stellt folgende Operationen bereit:

Insert, FindMax, ExtractMax, IncreaseKey $O(\log n)$

$O(1)$



Implementierung 2

- Daten werden in einem Heap gespeichert
- neue Elemente werden angehängt und raufgereicht
- Maximum steht immer in der Wurzel des Heaps

Teil III [CLRS]



Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

Abstrakter Datentyp	<i>Funktionalität</i>								
ptr Insert(key k , info i)	<ul style="list-style-type: none">● lege neuen Datensatz (k, i) an● $M = M \cup \{(k, i)\}$● gib Zeiger auf (k, i) zurück <div data-bbox="946 954 2032 1284" style="text-align: center;">M <table border="1" style="margin: auto;"><tr><td>key_1</td><td>key_2</td><td>\dots</td><td>key_n</td></tr><tr><td>$info_1$</td><td>$info_2$</td><td>\dots</td><td>$info_n$</td></tr></table></div>	key_1	key_2	\dots	key_n	$info_1$	$info_2$	\dots	$info_n$
key_1	key_2	\dots	key_n						
$info_1$	$info_2$	\dots	$info_n$						

Teil III [CLRS]

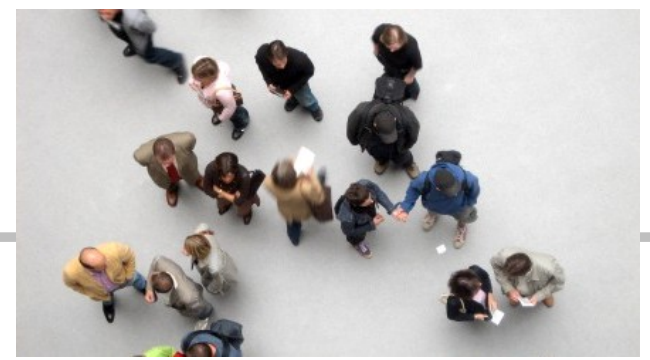


Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

Abstrakter Datentyp	<i>Funktionalität</i>								
ptr Insert(key k , info i) Delete(ptr x)	<ul style="list-style-type: none"><li data-bbox="953 715 1853 794">• $M = M \setminus \{(x.key, x.info)\}$ <div data-bbox="953 954 2034 1289"><p style="text-align: right;">M</p><table border="1"><tr><td>key_1</td><td>key_2</td><td>\dots</td><td>key_n</td></tr><tr><td>$info_1$</td><td>$info_2$</td><td>\dots</td><td>$info_n$</td></tr></table></div>	key_1	key_2	\dots	key_n	$info_1$	$info_2$	\dots	$info_n$
key_1	key_2	\dots	key_n						
$info_1$	$info_2$	\dots	$info_n$						

Teil III [CLRS]



Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

Abstrakter Datentyp	<i>Funktionalität</i>
ptr Insert(key k , info i) Delete(ptr x) ptr Search(key k)	<ul style="list-style-type: none">• falls vorhanden, gib Zeiger p mit $p.key = k$ zurück• sonst gib Zeiger nil zurück

Teil III [CLRS]



Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

Abstrakter Datentyp	<i>Funktionalität</i>
ptr Insert(key k , info i) Delete(ptr x) ptr Search(key k) ptr Minimum() ptr Maximum() ptr Predecessor(ptr x) ptr Successor(ptr x)	<ul style="list-style-type: none">• sei $M' = \{(k, i) \in M \mid k < x.key\}$• falls $M' = \emptyset$, gib <i>nil</i> zurück,• sonst gib Zeiger auf (k^*, i^*) zurück, wobei $k^* = \max_{(k,i) \in M'} k$

Teil III [CLRS]



Dynamische Menge:

verwaltet Elemente einer sich ändernden Menge M

Abstrakter Datentyp	Funktionalität	
<code>ptr Insert(key k, info i)</code> <code>Delete(ptr x)</code> <code>ptr Search(key k)</code>	} Änderungen	} Wörterbuch
<code>ptr Minimum()</code> <code>ptr Maximum()</code> <code>ptr Predecessor(ptr x)</code> <code>ptr Successor(ptr x)</code>	} Anfragen	

Implementierung: je nachdem... Drei Beispiele!

I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp

boolean Empty()

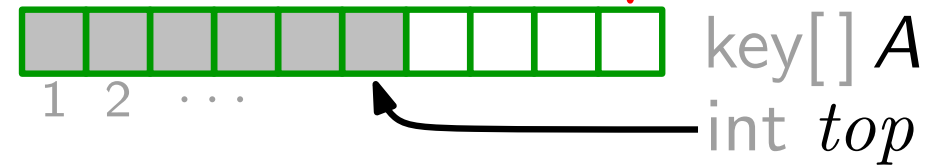
Push(key k)

key Pop()

key Top()

Implementierung

Größe?



if $top == 0$ **then return** *true*
else return *false*

$top = top + 1$
 $A[top] = k$

if Empty() **then error** „underflow“
else

$top = top - 1$
 return $A[top + 1]$

if Empty() **then ... else return** $A[top]$

I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*



Abstr. Datentyp	Implementierung	
Stack(int n)	$A = \mathbf{new}^* \text{key}[1..n]$ $top = 0$	$\text{key}[]$ A $\text{int } top$
boolean Empty()	if $top == 0$ then return true else return false	
Push(key k)	$top = top + 1$ { if $top > A.length$ then $A[top] = k$ { error „overflow“	
key Pop()	if Empty() then error „underflow“ else $top = top - 1$ return $A[top + 1]$	
key Top()	if Empty() then ... else return $A[top]$	

Laufzeiten?

Alle* $O(1)$,
d.h. konstant.

I. Stapel

verwaltet sich ändernde Menge nach *LIFO-Prinzip*

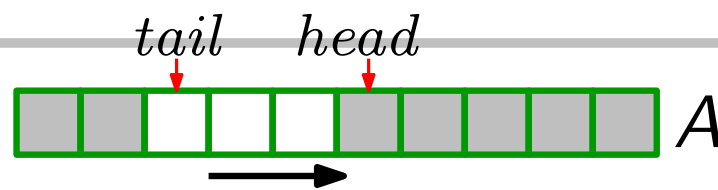


Abstr. Datentyp	Implementierung	
Stack(int <i>n</i>)	} <i>Konstruktor</i>	Attribute { key[] <i>A</i> int <i>top</i>
boolean Empty()	} <i>Methoden</i>	
Push(key <i>k</i>)		
key Pop()		
key Top()	} <i>Attribute</i>	Methoden { Stack - top: int - A: key[] + Empty(): boolean + Push(key) ...

Aufgabe:

Fertigen Sie ein UML-Diagramm für die Klasse *Stack* an!

II. Schlange



verwaltet sich ändernde Menge nach *FIFO-Prinzip*



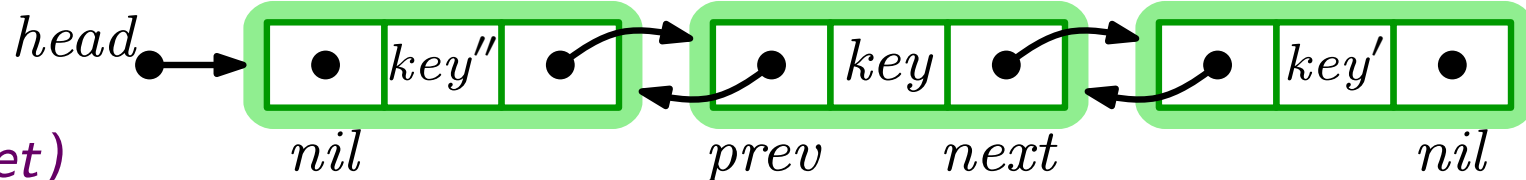
Abs. Datentyp	Implementierung	
Queue(int <i>n</i>)	$A = \mathbf{new}^* \text{key}[1..n]$ $tail = head = 1$	key[] <i>A</i> int <i>tail</i> int <i>head</i>
Aufgabe: Fangen Sie underflow & overflow ab!		
boolean Empty()	if $head == tail$ then return true else return false	
Enqueue(key <i>k</i>) <i>stell neues Element an den Schwanz der Schlange an</i>	$A[tail] = k$ if $tail == A.length$ then $tail = 1$ else $tail = tail + 1$	
key Dequeue() <i>entnimmt Element am Kopf der Schlange</i>	$k = A[head]$ if $head == A.length$ then $head = 1$ else $head = head + 1$ return k	

Laufzeiten?

Alle* $O(1)$.

III. Liste

(doppelt verkettet)



Abs. Datentyp

List()

ptr Search(key k)

ptr Insert(key k)

Implementierung

$head = nil$

Item

key	key
ptr	prev
ptr	next

ptr head

$x = head$

while $x \neq nil$ **and** $x.key \neq k$ **do**
 $x = x.next$

return x

$x = \mathbf{new}$ Item()

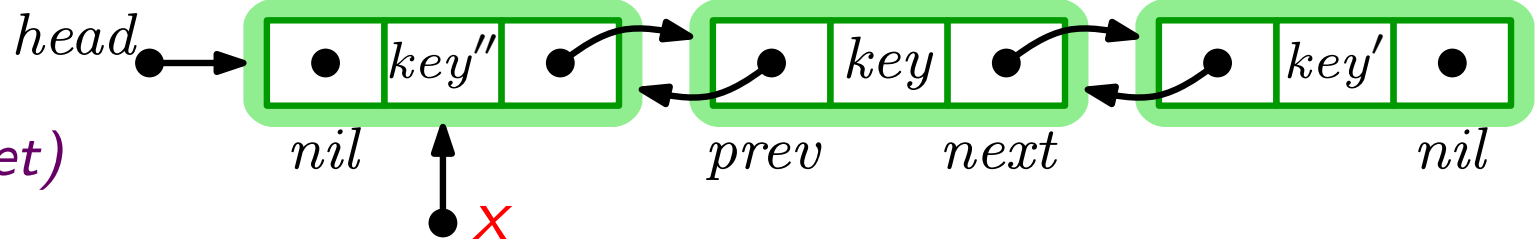
$x.key = k$; $x.prev = nil$; $x.next = head$

if $head \neq nil$ **then** $head.prev = x$

$head = x$; **return** x

III. Liste

(doppelt verkettet)



Abs. Datentyp

List()

ptr Search(key *k*)

Hausaufgabe:

Benutzen Sie
Stopper!

ptr Insert(key *k*)

Aufgabe:

Implementieren Sie
Delete(ptr *x*)

Implementierung

head = *nil*

Item(key *k*, ptr *p*)

key = *k*

next = *p*

prev = *nil*

Item

key *key*

ptr *prev*

ptr *next*

ptr *head*

x = *head*

while *x* ≠ *nil* **and** *x*.*key* ≠ *k* **do**

└ *x* = *x*.*next*

return *x*

x = **new** Item(*k*, *head*)

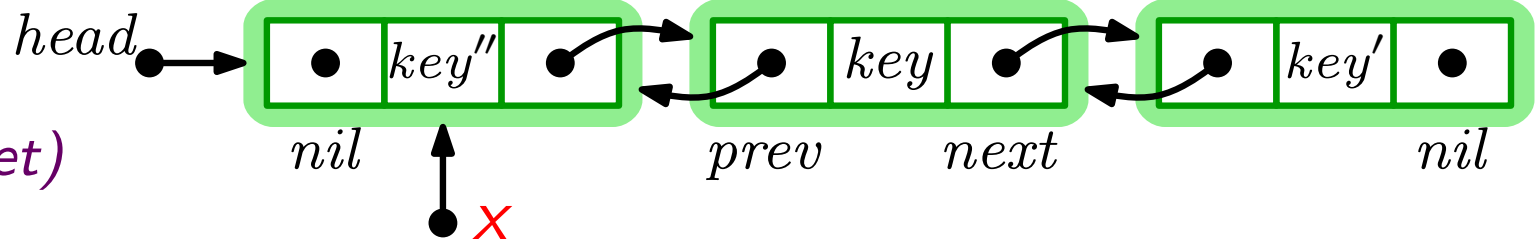
~~*x*.*key* = *k*, *x*.*prev* = *nil*; *x*.*next* = *head*~~

if *head* ≠ *nil* **then** *head*.*prev* = *x*

head = *x*; **return** *x*

III. Liste

(doppelt verkettet)



Abs. Datentyp

List() $O(1)$

Laufzeiten?

ptr Search(key k) $O(n)$

ptr Insert(key k) $O(1)$

$O(1)$
Delete(ptr x)

Implementierung

$head = nil$	Item(key k , ptr p) key = k next = p prev = nil	Item key key ptr prev ptr next
		ptr head

```

x = head
while x ≠ nil and x.key ≠ k do
  x = x.next
return x

```

```

x = new Item( $k$ , head)
x.key = k, x.prev = nil; x.next = head
if head ≠ nil then head.prev = x
head = x; return x

```


Von Pseudocode zu Javacode: (1) Item

```
public class Item {
```

```
private Object key;
private Item prev;
private Item next;
```

```
Item(key k, ptr p)
```

```
key = k
next = p
prev = nil
```

```
Item
```

```
key key
ptr prev
ptr next
```

```
public Item(Object k, Item p) {
    key = k;
    next = p;
    prev = null;
}
```

```
public void setPrev(Item p) { prev = p; }
public void setNext(Item p) { next = p; }
public Item getPrev() { return prev; }
public Item getNext() { return next; }
public Object getKey() { return key; }
```

setter-
und
getter-
Methoden

```
}
```

Von Pseudocode zu Javacode: (2) List

```
public class List {
```

```
private Item head;
```

```
ptr head
```

```
public List() {
    head = null;
}
```

```
List()
    head = nil
```

```
public Item insert(Object k) {
    Item x = new Item(k, head);
    if (head != null) {
        head.setPrev(x);
    }
    head = x;
    return x;
}
```

```
ptr Insert(key k)
```

```
x = new Item(k, head)
if head ≠ nil then
    | head.prev = x
head = x
return x
```

```
public Item getHead() { return head; }
```

Von Pseudocode zu Javacode: (2) List

```
ptr Search(key k)
```

```
  x = head
```

```
  while x ≠ nil and x.key ≠ k do
```

```
    ⊥ x = x.next
```

```
  return x
```



```
public Item search(Object k) {
```

```
  Item x = head;
```

```
  while (x != null && x.getKey() != k) {
```

```
    x = x.getNext();
```

```
  }
```

```
  return x;
```

```
}
```

Von Pseudocode zu Javacode: (2) List

```
Delete(ptr x)
```

```
  if  $x.prev \neq nil$  then  $x.prev.next = x.next$   
  else  $head = x.next$   
  if  $x.next \neq nil$  then  $x.next.prev = x.prev$ 
```



```
public void delete(Item x) {  
    if (x == null) System.out.println("Fehler!");  
    Item prev = x.getPrev();  
    Item next = x.getNext();  
    if (prev != null) prev.setNext(next);  
    else head = next;  
    if (next != null) next.setPrev(prev);  
}
```

```
}
```

Javacode: (3) Main

```
public class Listentest {  
    public static void main(String[] args) {  
        List myList = new List();  
  
        myList.insert(new Integer(10));  
        myList.insert(new Integer(16));  
  
        System.out.println("Die Liste enthaelt:");  
        for (Item it = myList.getHead(); it != null;  
            it = it.getNext()) {  
            System.out.println((Integer) it.getKey());  
        } Was wird hier ausgegeben?  
  
    }  
}
```

Javacode: (3) Main

```
public class Listentest {  
    public static void main(String[] args) {  
        List myList = new List();  
        myList.insert(new Integer(10));  
        myList.insert(new Integer(16));  
        System.out.println("Die Liste enthaelt:");  
        for (Item it = myList.getHead(); it != null;  
            it = it.getNext()) {  
            System.out.println((Integer) it.getKey());  
        }  
        Item it = myList.search(new Integer(16));  
        myList.delete(it);  
    }  
}
```

```
Die Liste enthaelt:  
16  
10  
Fehler!
```

Warum "Fehler!"?

Item.java

```
public Item search(Object k) {  
    Item x = head;  
    while (x != null && x.getKey() != k) {  
        x = x.getNext();  
    }  
    return x;  
}
```

Listentest.java

```
myList.insert(new Integer(16));  
...  
Item it = myList.search(new Integer(16));  
myList.delete(it);
```

```
public void delete(Item x) {  
    if (x == null) System.out.println("Fehler!");  
    Item prev = x.getPrev();  
    Item next = x.getNext();  
    if (prev != null) prev.setNext(next);  
    else head = next;  
    if (next != null) next.setPrev(prev);  
}
```

Warum "Fehler!" ?

Item.java

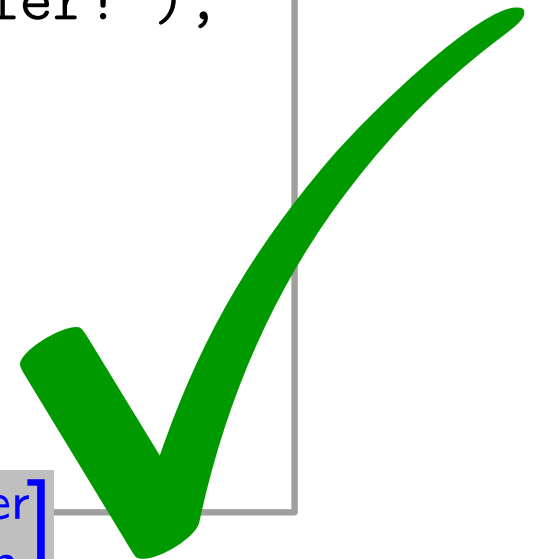
```
public Item search(Object k) {
    Item x = head;    !k.equals(x.getKey())
    while (x != null && x.getKey() != k) {
        x = x.getNext();
    }
    return x;
}
```

Listentest.java

```
myList.insert(new Integer(16));
... gleiche Zahlen, aber verschiedene Objekte!
Item it = myList.search(new Integer(16));
myList.delete(it);
```

```
public void delete(Item x) {
    if (x == null) System.out.println("Fehler!");
    Item prev = x.getPrev();
    Item next = x.getNext();
    if (prev != null) prev.setNext(next);
    else head = next;
    if (next != null) next.setPrev(prev);
}
```

**[Unschön: Klasse Item muss public sein, so dass Anwender
und Bibliotheksklasse List darüber kommunizieren können.]**



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 $O(1)$ Zeit!
 Listen sind mächtiger als Stapel/Schlangen. Wozu also Stapel/Schlangen?