

# Abstrakte Datentypen und Datenstrukturen



Professor Dr. Petra Mutzel  
Lehrstuhl für Algorithm Engineering, LS11  
Fakultät für Informatik, TU Dortmund

3. VO DAP2 SS 2009 21. April 2009

# Praktikum zu DAP 2

- Beginn: Mittwoch 22. April
- Bitte das 1. Praktikumsblatt vorher durchlesen (nicht schon lösen ☺)

# Freiwilliger Linux-Kurs von Wilfried Rupflin und Sven Jörges

- **Termine:** Dienstag 14-16 Uhr und alternativ Donnerstag 16-18 Uhr
- **Beginn:** 21.04. (heute)
- GB V R. 014/015

# Überblick

- ADT Sequence
  - ADT Stack
  - ADT Queue
  - ADT Dictionary
- Realisierung mit  
- Feldern  
- verketteten Listen
- Theoretische Laufzeitanalyse

# Motivation

„Warum soll ich hier bleiben?“  
Grundlage für alles Weitere!!!

„Das kenne ich ja schon alles!“  
Auch die Analyse?

Tipp: Beliebte Klausuraufgabe: „Realisieren Sie den ADT XXX mit einfach verketteten Listen der Form YYY (nicht ringförmig). Analyse!“

Rückblick:

# Asymptotische Laufzeit: O-Notation

- $O(g(n))$  ist die Menge aller positiven Funktionen  $f(n)$ , für die ein  $c$  und ein  $n_0$  (beide positiv und unabhängig von  $n$ ) existieren, so dass für alle  $n \geq n_0$  gilt:  $f(n) \leq cg(n)$ .

$$O(g(n)) = \{f(n) : N \rightarrow R^+ \mid (\exists c, n_0 > 0), (\forall n \geq n_0) : f(n) \leq cg(n)\}$$

Beweis für die Aussage:

- $\theta(g(n)) = O(g(n)) \cap \Omega(g(n))$  (s. Tafel)

## Abstrakte Datentypen

- Ein **abstrakter Datentyp (ADT)** besteht aus einem Wertebereich (d.h. einer Menge von Objekten) und darauf definierten Operationen.
- Die Menge der Operationen bezeichnet man auch als **Schnittstelle** des Datentyps.
- Eine **Datenstruktur** ist eine Realisierung bzw. Implementierung eines ADT.

## Der ADT Sequence

- **Wertebereich:** Menge aller endlichen Folgen eines gegebenen Grundtyps.
- $S = \langle a_1, a_2, \dots, a_n \rangle$       Leere Sequenz:  $\langle \rangle$
- **Operationen:**
  - Im folgenden betrachten wir die Sequence  $S = \langle a_1, a_2, \dots, a_n \rangle$  mit Grundtyp val

## Operationen des ADT Sequence

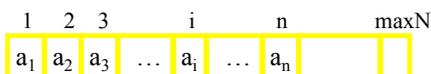
- **INSERT(val x, pos p): pos**
  - Einfügen eines Elements x vor dem Element an Position p
- **INSERT(x, pos(a<sub>i</sub>)):  $S = \langle a_1, a_2, \dots, a_{i-1}, x, a_i, \dots, a_n \rangle$**
- **INSERT(x, nil):  $S = \langle a_1, a_2, \dots, a_n, x \rangle$**
- **DELETE(pos p)**
  - Löscht das Element an Position p ( $p \neq \text{nil}$ ).
- **DELETE(pos(a<sub>i</sub>)):  $S = \langle a_1, a_2, \dots, a_{i-1}, a_{i+1}, \dots, a_n \rangle$**

## Operationen des ADT Sequence

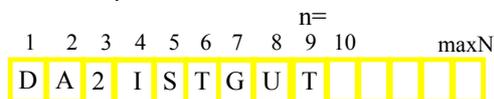
- **GET(int i): val**
  - Gibt das Element an i-ter Stelle zurück ( $1 \leq i \leq n$ )
- **GET(i):  $a_i$**
- **CONCATENATE(Seq S')**
  - Hängt die Sequence S' hinten an S an (müssen gleichen Grundtyp besitzen)
- **CONCATENATE(S')**: Falls  $S' = \langle a'_1, a'_2, \dots, a'_m \rangle$ , dann:  $\langle a_1, a_2, \dots, a_n, a'_1, \dots, a'_m \rangle$

## Realisierung durch Felder

- **Interne Repräsentation:**
  - Speicherung als Array der Dimension maxN
  - Position wird durch Feldindex angegeben
- Leeres Feld:  $n := 0$



## Beispiel: INSERT(P,3):3

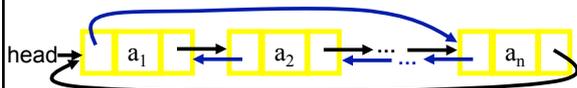


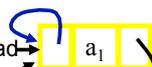
```

Procedure INSERT(val x, int t):int
if n=maxN then throw Overflow Exc.
for i:=n downto t { A[i+1]:=A[i] }
A[t]:=x
n:=n+1
return t
    
```

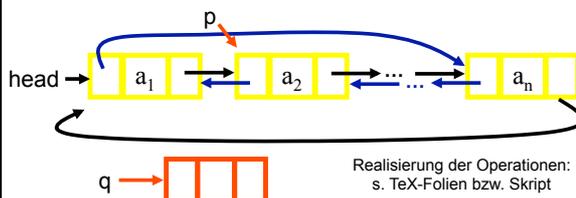
## Realisierung durch Listen

- Interne Repräsentation:
  - Speicherung als doppelt verkettete Liste
  - vorwärts und rückwärts ringförmig verkettet



- Leere Liste: head → nil
- 1-elementige Liste: head → 

## Beispiel: INSERT(x,p):q

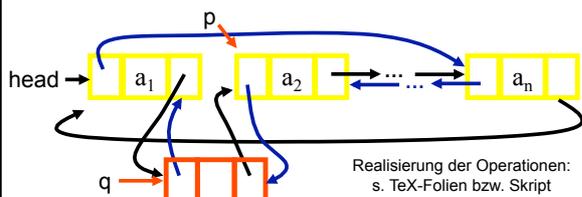


Realisierung der Operationen:  
s. TeX-Folien bzw. Skript

```

if head=nil then head:=q.next:=q.prev:=q else
if p=head then head:=q
if p=nil then p:=head // hinten einfügen = vor head
q.next:=p; q.prev:=p.prev
q.next.prev:=q.prev.next:=q ...
    
```

## Beispiel: INSERT(x,p):q



Realisierung der Operationen:  
s. TeX-Folien bzw. Skript

```

if head=nil then head:=q.next:=q.prev:=q else
if p=head then head:=q
if p=nil then p:=head // hinten einfügen = vor head
q.next:=p; q.prev:=p.prev
q.next.prev:=q.prev.next:=q ...
    
```

## Laufzeitanalyse des ADT Sequence

Operation	Best-Case		Average-Case Worst-Case	
	Felder	Listen	Felder	Listen
Initialisierung	$\Theta(1)+\text{Alloc}$	$\Theta(1)$	$\Theta(1)+\text{Alloc}$	$\Theta(1)$
INSERT	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
DELETE	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
GET	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
CONCAT	$\Theta(1+S \cdot n)$	$\Theta(1)$	$\Theta(1+S \cdot n)$	$\Theta(1)$

⇒ bei Einfügen und Entfernen: Listen besser als Arrays

## Der ADT Stack

„Stack“: Stapelspeicher, LIFO-Speicher

- **Wertebereich:** Menge aller endlichen Folgen eines gegebenen Grundtyps.

•  $S = \langle a_1, a_2, \dots, a_n \rangle$       Leerer Stack:  $\langle \rangle$

### Operationen:

- Im folgenden betrachten wir den Stack  $S = \langle a_1, a_2, \dots, a_n \rangle$  mit Grundtyp val

## Operationen des ADT Stack

### • PUSH(val x)

- Legt ein neues Element x auf den Stack

• PUSH(x):  $S = \langle a_1, a_2, \dots, a_n, x \rangle$

### • POP() : val

- Gibt das oberste Element des Stacks zurück und entfernt es ( $n > 0$ ).

• POP():  $a_n$

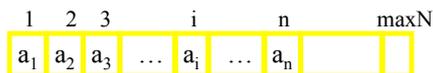
### • ISEMPY() : bool

- Gibt true zurück, falls S leer ist; sonst false.

## Realisierung des ADT Stack durch Felder

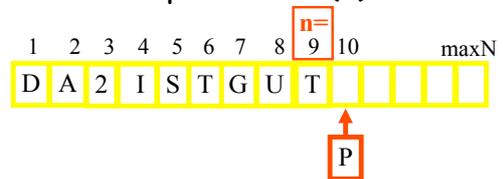
- **Interne Repräsentation:**
  - Speicherung als Array der Dimension  $\text{maxN}$
  - Position wird durch Feldindex angegeben

- **Leeres Feld:**  $n:=0$



ähnlich wie bei ADT Sequence, nur einfacher

## Beispiel: PUSH(P)

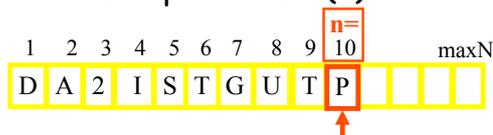


```

Procedure PUSH(val x)
if n=maxN then throw Overflow Exception
n:=n+1
A[n]:=x
    
```

Realisierung der Operationen: s. TeX-Folien bzw. Skript

## Beispiel: PUSH(P)



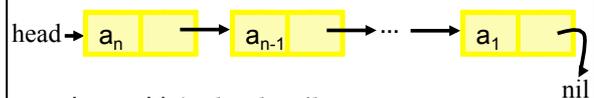
```

Procedure PUSH(val x)
if n=maxN then throw Overflow Exception
n:=n+1
A[n]:=x
    
```

Realisierung der Operationen: s. TeX-Folien bzw. Skript

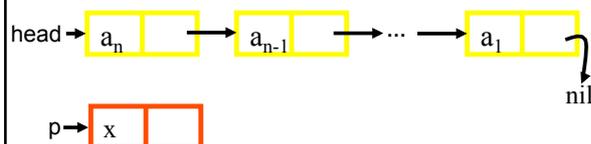
## Realisierung des ADT Stack durch Listen

- **Interne Repräsentation:**
  - Speicherung als **einfach** verkettete Liste
  - ~~vorwärts und rückwärts ringförmig~~ verkettet



- **Leere Liste:** head  $\rightarrow$  nil
- **1-elementige Liste:** head  $\rightarrow$  a<sub>1</sub>  $\rightarrow$  nil

## Beispiel: PUSH(x)

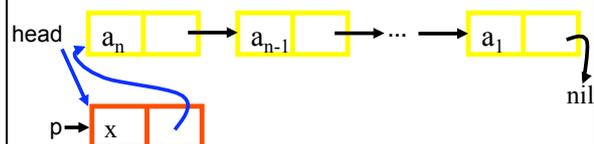


```

Procedure PUSH(val x)
var SListEl p:=new SListEl
p.value:=x
p.next:=head
head:=p
    
```

Realisierung der Operationen: s. TeX-Folien bzw. Skript

## Beispiel: PUSH(x)



```

Procedure PUSH(val x)
var SListEl p:=new SListEl
p.value:=x
p.next:=head
head:=p
    
```

Realisierung der Operationen: s. TeX-Folien bzw. Skript

## Laufzeitanalyse des ADT Stack

Worst-Case, Best-Case, Average Case

Operation	Felder	Listen
Initialisierung	$\Theta(1)+\text{Alloc}$	$\Theta(1)$
IEMPTY	$\Theta(1)$	$\Theta(1)$
PUSH	$\Theta(1)$	$\Theta(1)$
POP	$\Theta(1)$	$\Theta(1)$

⇒ falls  $\text{maxN}$  bekannt: Arrays besser, da dynamische Speicherallokierung (wie bei Listen) langsamer ist.

## Der ADT Queue

„Queue“: Warteschlangen, FIFO-Speicher

- **Wertebereich:** Menge aller endlichen Folgen eines gegebenen Grundtyps.

•  $S = \langle a_1, a_2, \dots, a_n \rangle$       Leere Queue:  $\langle \rangle$

### • Operationen:

– Im folgenden betrachten wir

die Queue:  $S = \langle a_1, a_2, \dots, a_n \rangle$  mit Grundtyp  $\text{val}$

## Operationen des ADT Queue

- **PUT (val x)**  
– Legt ein neues Element  $x$  an das Ende der Queue

• **PUT (x):**  $S = \langle a_1, a_2, \dots, a_n, x \rangle$

- **GET () : val**  
– Gibt das erste Element der Queue zurück und entfernt es ( $n > 0$ ).

• **GET ():**  $a_1$

- **IEMPTY () : bool**  
– Gibt **true** zurück, falls  $S$  leer ist; sonst **false**.

## Realisierung des ADT Queue durch Felder

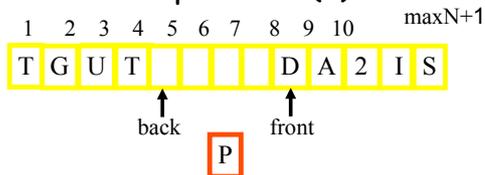
### • Interne Repräsentation:

- Speicherung als Array der Dimension  $\text{maxN}$
- Trick: Queue ist zyklisch im Feld gespeichert
- Position wird durch Feldindizes  $\text{front}$  und  $\text{back}$  angegeben



$\text{back}$  zeigt hinter das letzte Listenelement; um korrekt auf Überlauf zu testen, nutzen wir  $\text{maxN}+1$  Felder

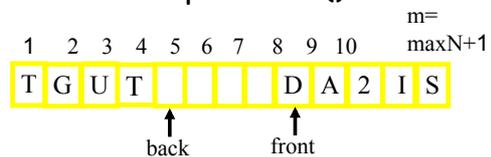
## Beispiel: PUT(P)



Initialisierung:  $\text{front} := \text{back} := 1$       Function **IEMPTY ()**: bool  
return  $\text{front} = \text{back}$

Procedure **PUT (val x)**  
 $A[\text{back}] := x$   
if  $\text{back} = m$  then  $\text{back} := 1$  else  $\text{back} := \text{back} + 1$   
if  $\text{front} = \text{back}$  then throw Overflow Exception

## Beispiel: GET()

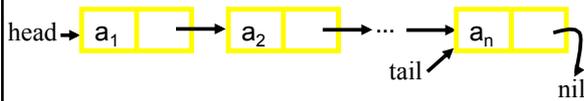


Procedure **GET ()**: val  
if  $\text{front} = \text{back}$  then throw Underflow Exception  
 $x := A[\text{front}]$   
if  $\text{front} = m$  then  $\text{front} := 1$  else  $\text{front} := \text{front} + 1$   
return  $x$

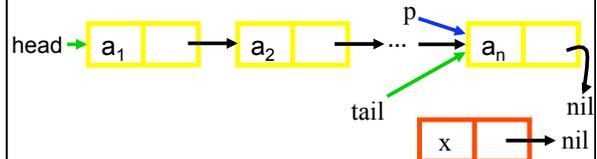
Realisierung aller Operationen: s. Skript

## Realisierung des ADT Queue durch Listen

- **Interne Repräsentation:**
  - Speicherung als einfach verkettete Liste
  - Zeiger head und tail auf Anfang und Ende



## Beispiel: PUT(x)

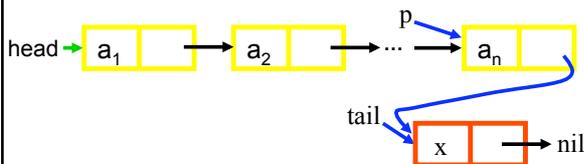


```

Procedure PUT(val x)
  var SListEl p:=tail
  tail:=new SListEl
  tail.value:=x; tail.next:=nil
  if head=nil then head:=tail else p.next:=tail
    
```

Realisierung der Operationen: s. Skript

## Beispiel: PUT(x)



```

Procedure PUT(val x)
  var SListEl p:=tail
  tail:=new SListEl
  tail.value:=x; tail.next:=nil
  if head=nil then head:=tail else p.next:=tail
    
```

Realisierung der Operationen: s. Skript

## Laufzeitanalyse des ADT Queue

Worst-Case, Best-Case, Average Case

Operation	Felder	Listen
Initialisierung	$\Theta(1)+\text{Alloc}$	$\Theta(1)$
ISEMPTY	$\Theta(1)$	$\Theta(1)$
PUT	$\Theta(1)$	$\Theta(1)$
GET	$\Theta(1)$	$\Theta(1)$

Vergleichen Sie diese Tabelle mit der von ADT Stack

## Der ADT Dictionary

„Dictionary“: Wörterbuch z.B. (Matrikelnummer,Name)

- **Wertebereich:**  $D \subseteq K \times V$ , wobei K Schlüssel (key) bezeichnet und V die Werte. Dabei ist jedem  $k \in K$  höchstens ein  $v \in V$  zugeordnet.

### • Operationen:

- Einfügen, Entfernen, Suchen (s. nächste Folie)
- Im folgenden sei Q ein Dictionary vor Anwendung der Operationen.

## Operationen des ADT Dictionary

- **INSERT(K k, V v)** z.B. (Matrikelnummer,Name)
  - Falls k nicht schon in D ist, dann wird ein neuer Schlüssel k mit Wert v in D eingefügt, andernfalls wird der Wert des Schlüssels k auf v geändert.

INSERT(k,v): Falls k neu:  $D := D \cup (k, v)$ , sonst

$D := D \setminus (k, v') \cup (k, v)$

### • DELETE(K k)

- Entfernt Schlüssel k mit Wert aus D (falls k in D)

### • SEARCH(K k): V

- Gibt den bei Schlüssel k gespeicherten Wert zurück (falls k in D)

## Der ADT Dictionary

- **Wörterbuchproblem:** Finde eine Datenstruktur mit möglichst effizienten Implementierungen für die Operationen eines Dictionary.
- **Naive Lösung:** als Paar von Feldern: → Lineare Laufzeit **für alle Operationen**
- **Im Laufe der Vorlesung:** einige weitaus bessere Verfahren!