IIF
INSTITUT FÜR
INFORMATIK
FREIBURG

Vorlesung Informatik 2
Algorithmen und Datenstrukturen

(06 - Anwendungen von Stapeln und Schlangen)

Prof. Dr. Susanne Albers

Lineare Listen (1)

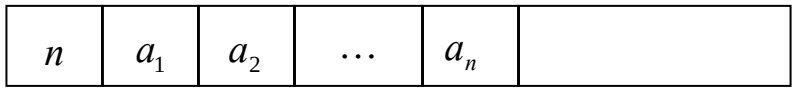
- Lineare Anordnung von Elementen eines Grundtyps
  (elementarer Datentyp oder Grundklasse, etwa Object)
- Grundtyp ist oft weiter strukturiert, etwa

```
class Grundklasse {
        int key; // eindeutiger Schlüssel
        Infoklasse info; // weitere Informationen
        }
```

- Bsp: Liste von Zahlen, Zeichenkette
- Operationen (Methoden)
  - Initialisieren (leere Liste, . . .)
  - Länge bestimmen
  - Suchen (inhaltsbasiert), kann aktuelles Element festlegen
  - Positionieren (nach laufender Nummer), . . .
  - Einfügen, an gegebener oder aktueller Position
  - Entfernen, . . .
  - Verketten, Ausgeben, Teilliste bilden, Umkehren, . . .

Implementierungen für Lineare Listen (1)

Mit Hilfe von Arrays:

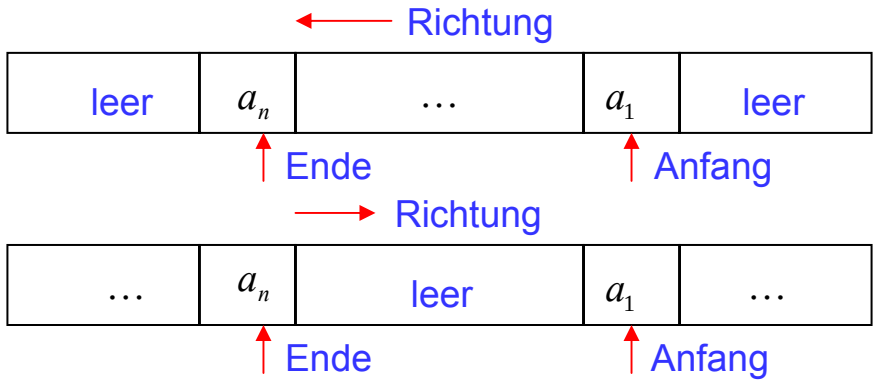


Eigenschaften dieser Lösung:

- Maximale Anzahl von Elementen vorgegeben (Länge des Arrays)
  Abhilfe: Dynamisierung
- Schlechte Speicherplatzausnutzung für kurze Listen
- Teure Verschiebeoperationen beim Einfügen/Entfernen von Elementen
+ Direkter Zugriff auf feste Positionen

Implementierungen für Lineare Listen (2)

Zyklische Speicherung mit Hilfe von Arrays:



Besondere Eigenschaften dieser Lösung:

+ Einfügen/Entfernen an den Listenenden in konstanter Zeit
+ Umkehren der Liste in konstanter Zeit

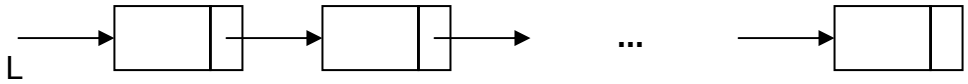
Implementierungen für Lineare Listen (3)

Mit Hilfe einfach verketteter (verlinkter) Elemente:

• speichere mit jedem Element einen Verweis auf das nächste oder null, falls das Element nicht existiert:


```
class node {
    Grundklasse content;
    node next;
}
```

• Liste L: Verweis auf das erste Element der Kette



Besondere Eigenschaften dieser Lösung:

\+ Länge der Liste kann sich beliebig ändern

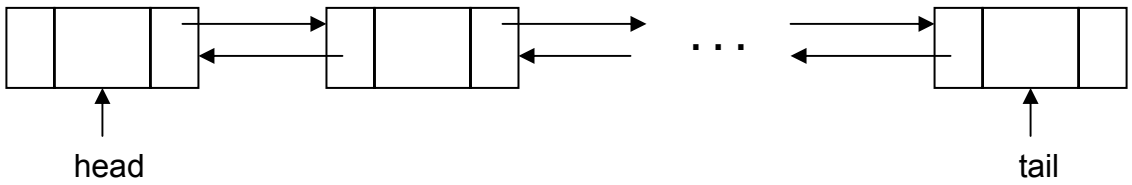
\- Kein direkter Zugriff auf Element an der Position p mehr möglich; nur Durchlaufen der Liste in O(p)

Implementierungen für Lineare Listen (4)

Mit Hilfe doppelt verketteter (verlinkter) Elemente:

• speichere mit jedem Element zwei Verweise auf das vorherige und nächste Element oder null, falls ein Element nicht existiert.

• Liste:



Besondere Eigenschaften dieser Lösung:

+ Umkehren der Liste in Θ(1)
+ Entfernen von Elementen leichter machbar Θ(1)
+ Einfügen/Entfernen/Zugriff am Anfang/Ende in Θ(1)
+ Hintereinanderhängen von Listen in Θ(1)

Stacks (Stapel) und Queues (Schlangen) (1)

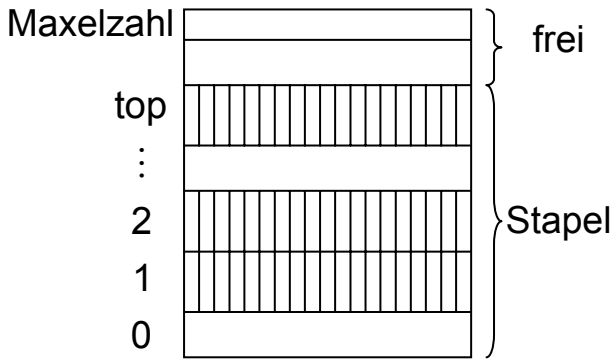
$L = < a_1, ..., a_n >$

Operation	Wirkung	Stack	Queue
Einfügungen L.pushHead(x) L.pushTail(x)	$< x, a_1, ..., a_n >$ $< a_1, ..., a_n, x >$	⊗	⊗
Enfernungen L.popHead() L.popTail()	$< a_2, ..., a_n > \rightarrow a_1$ $< a_2, ..., a_{n-1}, x > \rightarrow a_n$	⊗	⊗
Lesen L.top() L.bottom()	$\rightarrow a_1$ $\rightarrow a_n$	⊗	⊗
Diverses L.init() L.empty()	$<>$ $(L == <>)$	⊗ ⊗	⊗ ⊗

Stapel/Stack

LIFO Speicher

Operationen empty, push, pop , top



Implementation: In Java in der Klassenbibliothek vorhanden: java.util.Stack

```
   public class Stack extends Vector {
// Default Constructor: public Stack()
// Public Instance methods
public boolean empty();
public Object top() throws EmptyStackException;
public Object pop() throws EmptyStackException;
public Object push(Object item);
public int search(Object o)
}
```

Anwendungen für Stacks (Stapel) (1)

- Erkennen balancierter Klammerausdrücke

  ```
  (a(b))
  ```

  Lesen von „(“ → push ("("), Lesen von „)“ → pop ("("), Ausdruck korrekt, wenn Stack am Ende leer.

- Evaluierung korrekt geklammerter Ausdrücke

  ```
  ((a + b)*(c + d)) + f
  ```

- Iterierte Auswertung rekursiver Funktionen (Methoden). Jede Instanz der Funktion kann auf dem Stack Übergabeparameter, lokale Variablen und Returnwerte ablegen oder entnehmen. Schema dabei:

  1. Initialisiere Stack mit dem Anfangsproblem (der zu lösenden Aufgabe)

  2. Solange Stack nicht leer:
  Nimm das oberste Problem vom Stack und löse es. Fallen bei der Bearbeitung wieder Teilprobleme an, lege sie ebenfalls auf den Stack.

  3. Wenn Stack leer: Das Problem ist gelöst.

Anwendungen für Stacks (Stapel) (2)

Auswerten einer rekursiv definierten Funktion: Binomialkoeffizient

$$\binom{n}{k} = \begin{cases} 1, \text{ falls } & n = k \\ 1, \text{ falls } & k = 0 \\ \binom{n-1}{k-1} + \binom{n-1}{k}, \text{ sonst} & \end{cases}$$

Anwendungen für Stacks (Stapel) (3)

Ackermann-Funktion:
A(0,y) = y+1
A(x+1,0) = A(x,1)
A(x+1,y+1) = A(x, A(x+1,y))

Anwendungen für Stacks (Stapel) (4)

Beispiel: Die Türme von Hanoi (TOH)

- Bewege *n* unterschiedlich große Scheiben von Pfahl 1 nach Pfahl 2 mit Hilfe von Pfahl 3.
- Nie darf eine größere Scheibe auf einer kleineren liegen.
- Zu jedem Zeitpunkt darf nur eine Scheibe bewegt werden.

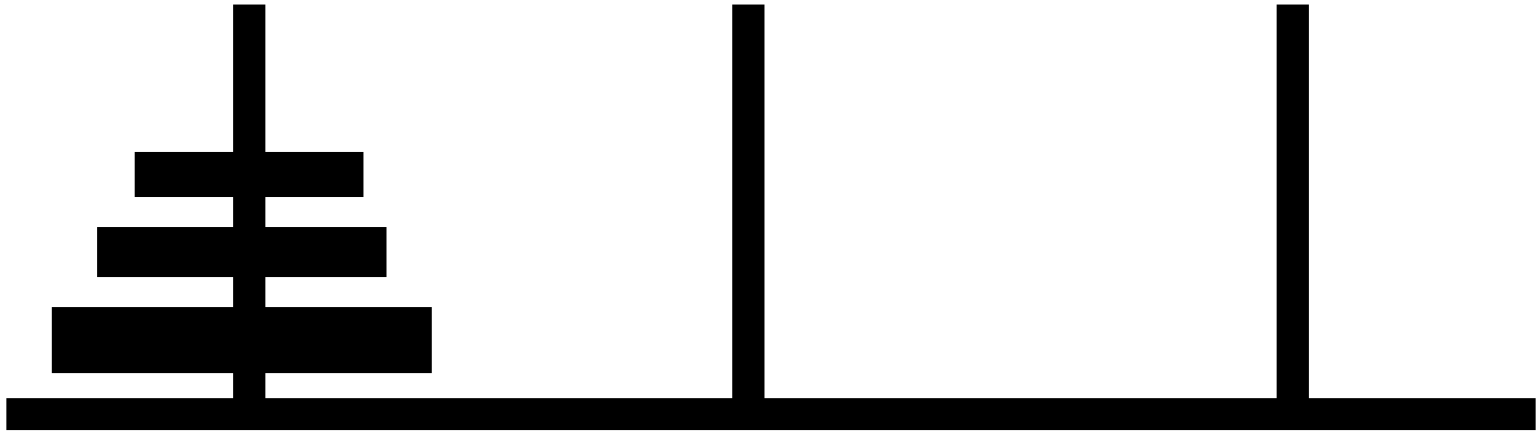
Codierung für die genannte Problemstellung: *n*, 1, 2, 3

Bewege Scheibe Nr. *n* direkt (falls frei) von 1 nach 2, Codierung: -*n*, 1, 2, 0

1. Initialisierung: s.push(*n*,1,2,3)
2. Wenn s.pop() *n, a, b, c* liefert, dann

falls (*n* < 0): „Bewege Scheibe Nr. |*n*| von *a* nach *b*.“
falls (*n* = 0): nichts
falls (*n* > 0):
s.push(*n-1*,*c*,*b*,*a*); (zuletzt)
s.push(-*n*,*a*,*b*,*0*); (dann)
s.push(*n-1*,*a*,*c*,*b*); (zuerst)

Die Türme von Hanoi



Bsp.: Die Türme von Hanoi (1)

```
class ArrNode {                          // Knoten fuer TOH-Stack
    private int [] content;
    private ArrNode next;
    ArrNode (int [] a, ArrNode n){ // Konstruktor
        content = a;
        next = n;
    }
    public int [] getContent (){
        return content;
    }
    public ArrNode getNext(){
        return next;
    }
} // ArrNode
```

Bsp.: Die Türme von Hanoi (2)

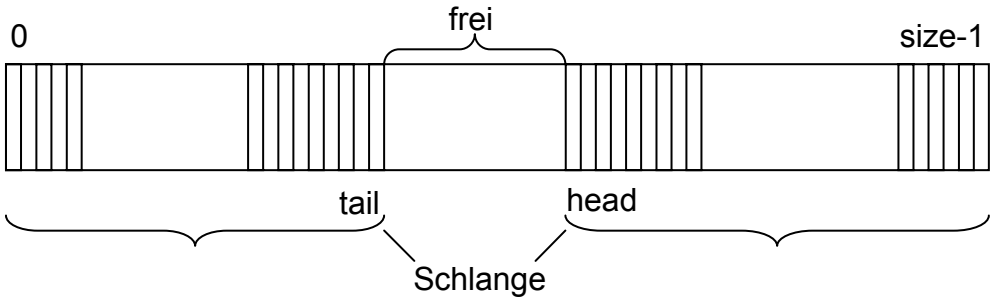
```
class TohStack {                    // der Stack fuer TOH
    ArrNode Head;
    TohStack (){                    // Konstruktor
        Head = null;
    }
    public boolean empty (){  // Test ob leer
        return Head == null;
    }
    public void push (int a, int b, int c, int d){
        int [] arr = { a, b, c, d };
        Head = new ArrNode (arr, Head);
    }
    public int [] pop (){
        ArrNode h = Head;
        Head = Head.getNext();
        return h.getContent ();
    }
} // TohStack
```

Bsp.: Die Türme von Hanoi (3)

```
public class TohStackTest {
    public static void main (String args[]){
        TohStack s = new TohStack ();
        s.push (Integer.parseInt (args[0]),1,2,3);   // Startproblem
        while ( !s.empty () ) {                      // Iteration statt Rekursion
            int [] a = s.pop ();
            if (a[0] < 0) System.out.println (
                "bewege Scheibe Nr. "+(-a[0])+
                " von Pfeiler "+a[1]+" nach "+a[2]);
            else if (a[0] != 0) {
                s.push (a[0]-1, a[3], a[2], a[1]);
                s.push ( -a[0], a[1], a[2], 0 );
                s.push (a[0]-1, a[1], a[3], a[2]);
            }
        }
    }
} // TohStackTest
```

Schlange/Queue

FIFO Speicher: empty, enqueue, dequeue




```
class Queue {
Object element[];
int lastOp, tail, head, size;

static final int deque = 1;
static final int enque = 0;

Queue(int sz) {
size = sz; head = 0; tail = size-1; lastOp = deque;
element = new Object[sz];
}
public boolean empty(){
return head==(tail+ 1)%size;}
public void enqueue(Object elem) throws Exception
...
public Object dequeue() throws Exception ...
}
```

Operationen

```
public void enqueue(Object elem) throws Exception {
    if ((head == (tail+1)%size) && lastOp == enque)
        throw new Exception("Queue full");
    lastOp = enque;
    tail = (tail + 1) % size;
    element[tail] = elem;
}
public Object dequeue() throws Exception {
    if ((head == (tail+1)%size) && (lastOp == deque))
        throw new Exception("Queue empty");
    Object temp = element[head];
    lastOp = deque;
    head = (head + 1) % size;
    return temp;
}
```

Test

```
public static void main(String argv[]) {
    Queue Q = new Queue(10);
    try {
        Q.enqueue(new String("Haus"));
        Q.enqueue(new Integer(10));
        Object o = Q.dequeue();
        System.out.println(o);
    }
    catch (Exception e) {
    System.out.println(e);
    }
}
```

Anwendung: Topologische Sortierung

Definition: Die Relation ≤ ist eine vollständige bzw. partielle Ordnung auf M, wenn (1)-(4) bzw. (2)-(4) gelten.

(1) $\forall x, y \in M: x \leq y$ oder $y \leq x$ (Vollständigkeit)

(2) $\forall x \in M: x \leq x$ (Reflexivität)

(3) $\forall x, y \in M$: Wenn $x \leq y$ und $y \leq x$, so ist $x = y$. (Antisymmetrie)

(4) $\forall x, y, z \in M$: Wenn $x \leq y$ und $y \leq z$, dann ist $x \leq z$ (Transitivität)

Beispiel: Potenzmenge 2^M einer Menge M bezüglich Teilmengenrelation partiell aber nicht vollständig geordnet.

Topologische Sortierung: Gegeben M und ≤ partiell. Gesucht: Einbettung in vollständige Ordnung, d.h. die Elemente von M sind in eine Reihenfolge $m_1,...,m_n$ zu bringen, wobei $m_i \leq m_j$ für $i < j$ gilt.

Lemma: Jede partiell geordnete Menge lässt sich topologisch sortieren.

Beweisidee: Es genügt, die Existenz eines minimalen Elements z in M zu zeigen. z muss nicht eindeutig sein. Wähle $m_1 = z$ und sortiere $M - \{z\}$ topologisch.

Verwendete Datenstrukturen

A : Array der Länge n, $A[i]$ natürliche Zahl

$L[i]$: Liste von Zahlenpaaren

Q : Queue (Schlange), die ebenfalls Zahlen enthält

Algorithmus :

Eingabe : p Paare (i,j) mit $x_i \leq x_j$ und $M = \{x_1,...,x_n\}$

(1) Setze alle $A[i]$ auf 0, alle $L[i]$ und Q seien leer

(2) Wird (j,k) gelesen, wird $A[k]$ um 1 erhöht und in $L[j]$ eingetragen.

(3) Durchlaufe A: Schreibe alle k mit $A[k] = 0$ in Q

(4) While $Q \neq \{\}$

$j = Q$.dequeue()

Gebe x_j aus

Durchlaufe $L[j]$: Wird (j,k) gefunden, wird $A[k]$ um 1 verringert.

Falls $A[k] = 0$ ist, Q.enqueue(k)

Satz : Der Algorithmus löst das Problem Topologische Sortierung in $O(n+p)$ Zeit.

Beispiel

Eingabe: (1,3), (3,7), (7,4), (4,6), (9,2), (9,5),
(2,8), (5,8), (8,6), (9,7), (9,4)

Datenstruktur nach der Vorverarbeitung:

Beispiel

Eingabe: (1,3), (3,7), (7,4), (4,6), (9,2), (9,5),
(2,8), (5,8), (8,6), (9,7), (9,4)

Datenstruktur nach der Vorverarbeitung:

```
      1    2    3    4    5    6    7    8    9
A:  [0    1    1    2    1    2    2    2    0]
L:  1,3  2,8  3,7  4,6  5,8       7,4  8,6  9,2
                                             9,5
                                             9,7
                                             9,4
```

Ablauf des Algorithmus:

```
     Q     |
-----------+-----------------------------
    Out    |
```