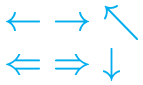
Algorithmen und Datenstrukturen (Th. Ottmann und P. Widmayer)

Folien: Elementare Datenstrukturen

Autor: Stefan Edelkamp

Institut für Informatik
Georges-Köhler-Allee
Albert-Ludwigs-Universität Freiburg

1 Überblick



Überblick

Datenstrukturen

Arrays

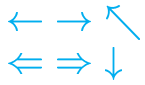
Listen

Stapel/Stack

Schlange/Queue:

Mengen

2 Datenstrukturen



Motivation: Wozu sind Datenstrukturen nützlich?

Arrays mit Adressierung

Listen mit

- 4 verschiedenen Implementierungen

Stapel/Schlangen

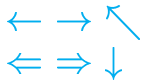
- Anwendung Stapel: Towers of Hanoi

- Anwendung Schlange: Topologische Sortierung

Mengen dargestellt als

- Bitvektoren

Algorithmen und Datenstrukturen



Algorithmus: Endliche Sequenz von Instruktionen, die angeben, wie ein Problem gelöst werden soll.

Datenstrukturen (DS): Dienen zur Organisation von Daten, um bestimmte Operationen (effizient) zu unterstützen.

Hier und heute: Werden DS isoliert betrachtet, in der Praxis sind sie wichtige Bestandteile von Algorithmen.

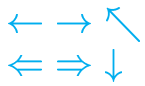
Verbindung: Datenstrukturen lassen sich vielfältig ineinander schachteln.

Eignung: Demnach ist es stets nützlich, sich Anwendungs- und Nichtanwendungssituationen zu überlegen

Unabhängigkeit: Darstellung im Grunde unabhängig von der Programmiersprache

Konkretisierung: Imperative Programmiersprachen, speziell: Programmiersprache Java

3 Arrays



1-dimensional: Array a für den Bereich $[1..n]$ sieht n Positionen für Daten des gewählten Typs vor.

Darstellung im Rechner: Array der Länge n blegt die Speicherplätze an den n Positionen $N+1,\ldots N+n$. Mit $a[i]$ sprechen wir das Datum an Position $N+i$ an.

Wesentlich: Arrays erlauben den Zugriff auf Komponenten über einen (berechenbaren) Index

Zugriffe: In Zeit $O(1)$. Keine Unterstützung von Operationen vom Typ Nimm $a[i]$ und füge es zwischen $a[j]$ und $a[j+1]$ ein.

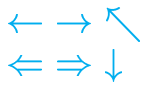
2-dimensional: Array $[1..n_1] \times [1..n_2]$. Arrayelement A[i][j] findet man an der Speicherplatzposition

$$N + (i-1)n_2 + j$$

k-dimensional: Array $[1..n_1] \times \ldots \times [1..n_k]$ Element $a[i_1]\ldots[i_k]$ findet sich an Speicherplatzposition

$$N + \left\{ \sum_{1 \le j \le k-1} (i_j - 1) \prod_{j < l \le k} n_l \right\} + i_k$$

4 Listen



Definition: Sei $L = (a_1, a_2, \ldots, a_n)$ geordnete Folge von Objekten eines bestimmten Typs, dann heißt: a_1 Anfangsobjekt (Kopf), a_n Endobjekt (Schwanz), a_{i+1} Nachfolger von a_i, a_i Vorgänger von a_{i+1}. Nachfolger von a_n und Vorgänger von a_1 ist `null`. Falls $n = 0$ heißt die Liste leer.

Operationen:

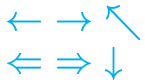
`insert`(x, L): Füge das Objekt x in L ein. Ergebnis ist eine Liste.

`find`(x, L): Falls $x \in L$ ist Antwort true, sonst false

`delete`(x, L): Entferne x aus L. Falls $x \notin L$ Fehlerbehandlung. Ergebnis ist eine Liste.

Weiterhin: `pred`(x, L), `succ`(x, L), `null`(L), `first`(L), `last`(L), `length`(L), `union`(L_1, L_2, L), `copy`(L, L'), `split`(x, L, L_1, L_2)

Listenoperationen mit Zeigern



Zeiger (Handle): p zum Zugriff, z.B. vom Typ `Object`

Operationen:

`insert`(x, p, L): Füge das Objekt x an Stelle p in L ein. Ergebnis ist eine Liste.

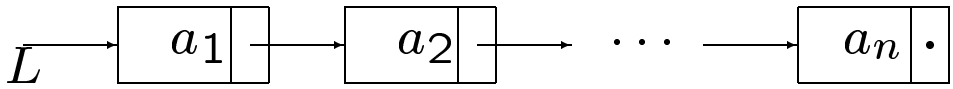
`delete`(p, L): Entferne Element an Stelle p aus L. Ergebnis ist eine Liste.

`search`(x, L): Liefert die erste Position als Zeiger, an der x in L vorkommt, und `null`, falls x in L nicht vorkommt

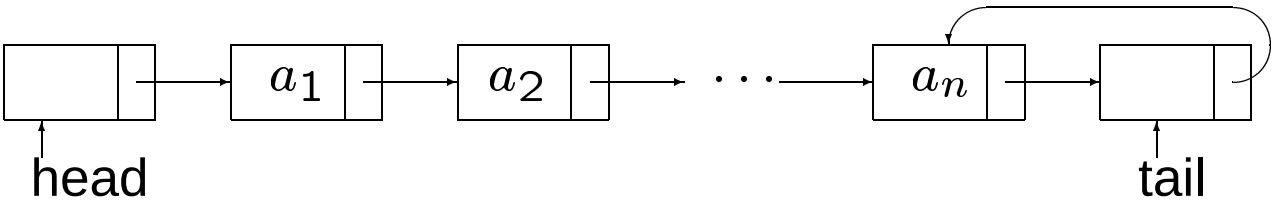
Weiterhin: `pred`(p, L), `succ`(p, L), `null`(L),
`first`(L), `last`(L), `length`(L),
`union`(L_1, L_2, L), `copy`(L, L'),
`split`(p, L, L_1, L_2)

Implementationen von Listen

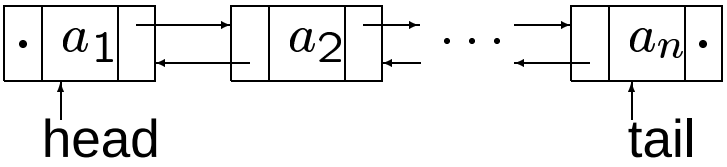
- Version 1: Als Array. Zeiger p vom Typ `int` (in `search`: $p = 0$ falls nicht vorhanden).
- Version 2: Einfach verkettet, Listenende `null`



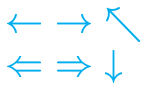
- Version 3: Einfach verkettet, am Listenanfang und -ende ein “dummy” Element



- Version 4: Doppelt verkettet, Listenanfang und Listenende `null`



Grundtypen und Knoten



Der Grundtyp beinhaltet Schlüssel bzw. Datensatz

```
public class Grundtyp {
  int key; // eventuell weitere Komponenten

  Grundtyp()         { this.key = 0; }
  Grundtyp(int key) { this.key = key; }

  public boolean equals(Grundtyp dat) {
    return this.key == dat.key;
  }
  public String toString() {
    StringBuffer st = new StringBuffer(""+key);
    return st.toString();
  }
}
```

Knoten dient zur Speicherung des Dateninhaltes und der (einfachen) Verkettung von Listenelementen. Wird ausschliesslich in der zugehörigen Listenklasse genutzt.

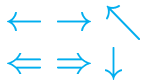
```
class Knoten {

  Grundtyp dat;   // Grundtyp
  Knoten next;    // Zeiger auf Nachfolgerknoten

  Knoten(Grundtyp dat, Knoten next) {
    this.dat = dat; this.next = next;
  }

}
```

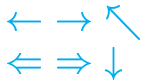
Arraydarstellung



Liste repräsentiert als Array:

```
public class Liste {
  Grundtyp[] L;
  int elzahl;
  Liste(int maxelzahl) {
    L = new Grundtyp[maxelzahl+1]; elzahl = 0;
  }
  public int search(Grundtyp x) {
    L[0] = x;
    int pos = elzahl;
    while (!L[pos].equals(x)) pos--;
    return pos;
  }
  public void insert(Grundtyp x, int p) {
    for (int pos=elzahl; pos >= p; pos--)
      L[pos+1] = L[pos];
    L[p] = x; elzahl++;
  }
  public void delete(int p) {
    elzahl--;
    for(int pos=p;pos<=elzahl;pos++)
      L[pos] = L[pos+1];
  }
  public String toString () {
    StringBuffer st = new StringBuffer("");
    for (int pos = 1; pos <= elzahl; pos++)
      st.append(pos == elzahl ?
                L[pos].toString():
                L[pos].toString()+", ");
    return st.toString();
  }
}
```

Dynamisch



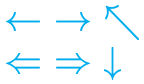
Liste implementiert als einfach verkettete Struktur:

```
public class Liste {

  Knoten head;              // Kopfelement
  Liste() { head = null; } // leere Liste

  public void insert (Grundtyp x) {
    head = new Knoten(x,head);
  }
  public boolean find(Grundtyp x) {
    for (Knoten i = head;i != null;i = i.next)
      if (i.dat.equals(x)) return true;
    return false;
  }
  public void delete(Grundtyp x) throws Exception {
    if (head.dat.equals(x)) head = head.next;
    for (Knoten i = head;i.next != null;i = i.next)
     if (i.next.dat.equals(x)) {
       i.next = i.next.next; return
    }
    throw new Exception("x kommt nicht vor");
  }
  public String toString() {
    StringBuffer st = new StringBuffer("");
    for (Knoten i = head;i != null;i = i.next)
      st.append(i.next == null
                        ? i.dat.toString()
                        : i.dat.toString()+", ");
    return st.toString();
  }
}
```

Verfeinerung



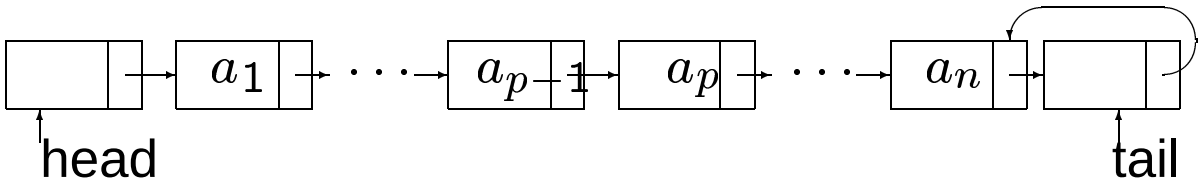
... nach (Ottmann / Widmayer):

```
public class Liste {
  Knoten head, tail;
  Liste() {
    Grundtyp dummy = new Grundtyp();
    head = new Knoten(dummy,null);
    tail = new Knoten(dummy,head);
    head.next = tail;
  }
  public void insert (Grundtyp x, Knoten p) {
    Knoten hilf;
    if (p == tail) hilf = tail; else hilf = p.next;
    p.next = new Knoten(p.dat,hilf);
    p.dat = x;
    if (p == tail)    tail = tail.next;
    if (hilf == tail) tail.next = p.next;
  }
  public void delete (Grundtyp x) throws Exception {
    tail.dat = x;
    Knoten pos = head;
    while(!pos.next.dat.equals(x)) pos = pos.next;
    if (pos.next != tail) pos.next = pos.next.next;
    else throw new Exception("x kommt nicht vor");
    if (pos.next == tail) tail.next = pos;
  }
  public Knoten search(Grundtyp x) {
    tail.dat = x;
    Knoten pos = head;
    do pos = pos.next; while (!pos.dat.equals(x));
    if (pos == tail) return null;
    return pos;
  }
}
```

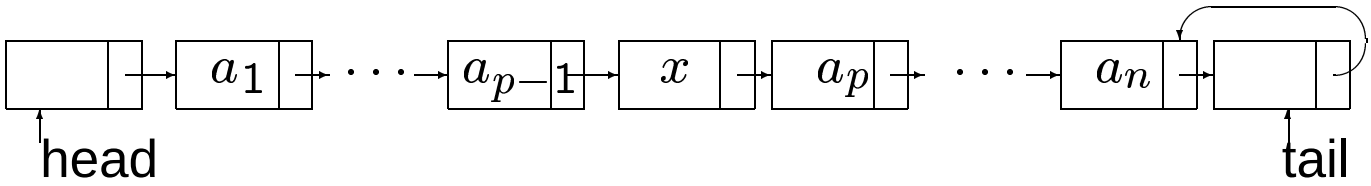
Veranschaulichung von Listenoperationen

Einfache Verkettung

Vor dem Einfügen

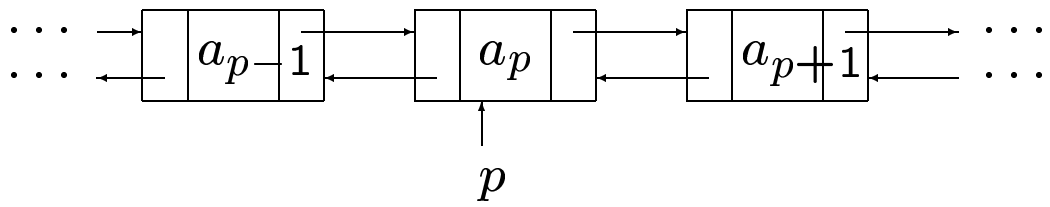


Nach dem Einfügen

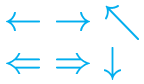


Doppelte Verkettung

Löschen in doppelt verketteter Liste:



Doppelte Verkettung



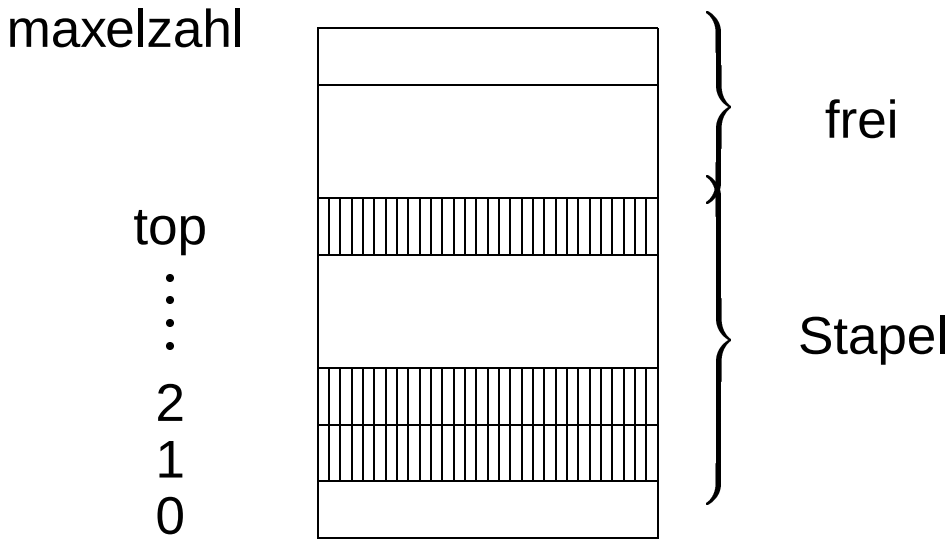
```
public class Knoten {
  Grundtyp dat;   // Grundtyp
  Knoten next;    // Zeiger auf Nachfolgerknoten
  Knoten prev;    // Zeiger auf Vorgaengerknoten
  ...
}
public class Liste {
  Knoten head, tail;
  Liste() { head = null; tail = null; }

  public void insert (Grundtyp x) {
    if (head == null)
      head = tail = new Knoten(x,null,null);
    else insert(x,head);
  }
  public void insert (Grundtyp x, Knoten p) {
    Knoten hilf = new Knoten(x,p.next,p);
    p.next = hilf;
    if (hilf.next == null) tail = hilf;
    else hilf.next.prev = hilf;
  }
  public void delete (Grundtyp x, Knoten p) {
    if (p == head) head = p.next;
    p.prev.next = p.next;
    if (p == tail) tail = p.prev;
    p.next.prev = p.prev;
  }
  public Knoten search(Grundtyp x) {
    for (Knoten i = head;i != null;i = i.next)
      if (i.dat.equals(x)) return i;
    return null;
  }
}
```

5 Stapel/Stack

LIFO Speicher

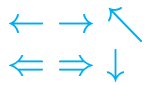
Operationen empty, push, pop (, peek)



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

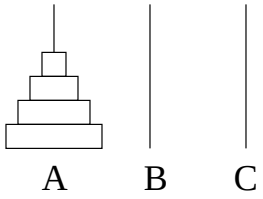
```
  public class Stack extends Vector {
    // Default Constructor: public Stack()
    // Public Instance methods
    public boolean empty();
    public Object peek() throws EmptyStackException;
    public Object pop() throws EmptyStackException;
    public Object push(Object item);
    public int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

Beispiel: Tower-of-Hanoi Problem



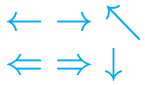
Problem: n Scheiben: $1, 2, 3, \ldots, n$,
3 Pfeiler: A,B,C (1,2,3).

Invarianz: Immer kleinere Scheibe auf der Größeren.



TOH: Live-Vorführung

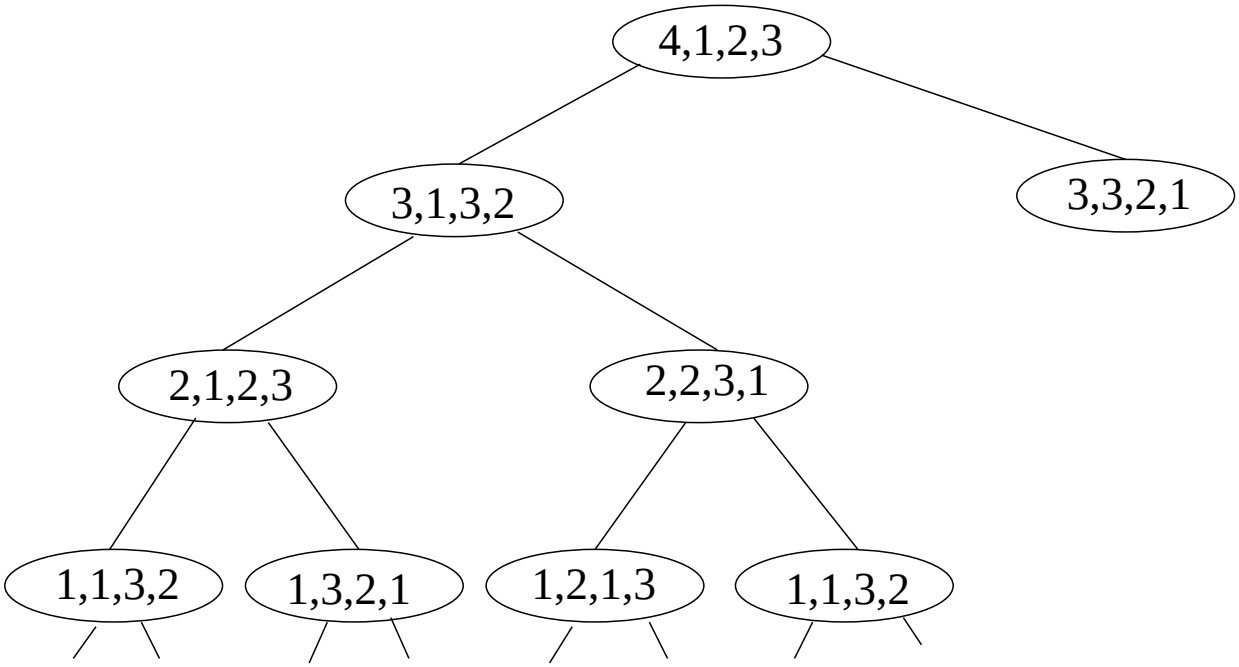
Rekursive Lösung



```
class Pfeiler {
  String name;
  Pfeiler(String n) { name=n;}
  public String toString(){ return name; }
}
public class ToHP {
  static long count = 0;
  static Pfeiler A, B, C;
  public static void main (String args[]){
    int n = Integer.parseInt (args[3]);
    A = new Pfeiler (args[0]);
    B = new Pfeiler (args[1]);
    C = new Pfeiler (args[2]);
    umschichten (n, A, B, C);
    System.out.println(count+" Scheibenbewegungen.");
  }
  public static void umschichten
   (int n, Pfeiler von, Pfeiler nach, Pfeiler mit) {
    if (n > 0) {
      umschichten(n-1,von, mit, nach);
      System.out.println(von+" --> "+nach);
      count++;
      umschichten(n-1,mit, nach, von);
    }
  }
}
```

Aufrufbaum

Tupel der Form: $(n, von, nach, mit)$:

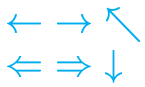


Stackreihenfolge:

```
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
+---+   +---+   +---+   +---+
```

Problem: Ausgabe in inneren Knoten

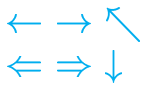
Spezielle Stackimplementierung für TOH



Genauer: Lineare Liste ohne explizites Knotenelement

```
class TohStack {
  int [] arr;
  TohStack next;
  TohStack (){ next = null; }      // Konstruktor 1
  TohStack (int [] a, TohStack n){ // Konstruktor 2
    arr = a;
    next = n;
  }
  public boolean empty () { return next == null; }
  public void push (int a, int b, int c, int d) {
    int [] A = { a, b, c, d };
      if ( empty () ) next = this;
      else next = new TohStack(arr, next);
      arr = A;
  }
  public int [] pop (){
      int [] a = arr;
      if ( next == this ) next = null;
      else { arr = next.arr; next = next.next; }
      return a;
  }
}
```

Testen: Tower-of-Hanoi-Stack



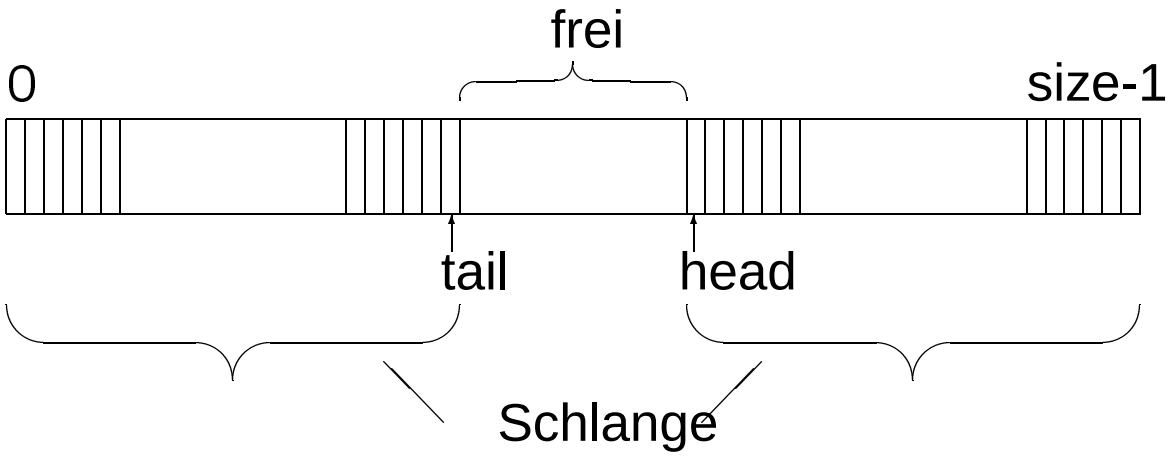
Idee: Auflösung der Rekursion durch Verwaltung der lokalen Variablen auf einem Stack.

```
public class tohStackTest {
  public static void main (String args[]) {
    TohStack s = new TohStack ();
    s.push (Integer.parseInt(args[0]), 1, 2, 3);
    while (!s.empty ()) {
      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]);
      }
    }
  }
}


//  > javac tohStackTest.java
//  > java tohStackTest 4
// bewege Scheibe Nr. 1 von Pfeiler 1 nach 3
// bewege Scheibe Nr. 2 von Pfeiler 1 nach 2
// bewege Scheibe Nr. 1 von Pfeiler 3 nach 2
// ...
```

6 Schlange/Queue:

FIFO Speicher: empty, enqueue, dequeue

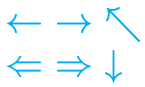



```
class Queue {
 Object element[];
 int lastOp, tail, head, size;

 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

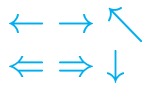


```
  public void enqueue(Object elem) throws Exception {
    if ((head == (tail+1)%size) && lastOp == enque)
      throw new Exception("Queue full");
    lastOp = enque;
    tail = (tail + 1) % size;
    element[tail] = elem;
  }
  public Object dequeue() throws Exception {
    if ((head == (tail+1)%size) && (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 $\leq$ 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 $y \leq x$
(2) $\forall x \in M : x \leq x$
(3) $\forall x, y \in M : x \leq y$ und $y \leq x \Rightarrow x = y$
(4) $\forall x, y \in M : x \leq y$ und $y \leq z \Rightarrow x \leq z$

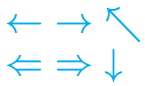
Beispiel: Potenzmenge 2^M einer Menge M bezüglich Teilmengenrelation partiell aber nicht vollständig geordnet.

Topologische Sortierung: Gegeben M und $\leq$ partiell. Gesucht: Einbettung in vollständige Ordnung, d.h. die Elemente von M sind in eine Reihenfolge $m_1, \ldots, m_n$ 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
V: Array der Länge n, $V[i]$ Zeiger auf $L[i]$
$L[i]$: Liste von Zahlenpaaren
Q: Queue (Schlange), die ebenfalls Zahlen enthält

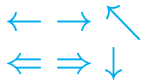
Algorithmus:

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

(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 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)

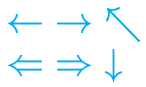
Datenstruktur nach der Vorverarbeitung:

```
   1    2    3    4    5    6    7    8    9
A:[0    1    1    2    1    2    2    2    0]
V: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       |  1   2   3   4   5   6   7   8   9
--------------+-----------------------------------
    {1,9}     |  0   1   1   2   1   2   2   2   0
              |
              |
              |
```

7 Mengen



Unterscheidung: Bitvektordarstellung und Listendarstellung

Eignung: Bitvektordarstellung für feste Grundmenge $G = \{1, \ldots, n\}$. und (relativ) großen Teilmengen M von G.

Bitvektor-Darstellung: M durch Array A mit

$$A[i] = 1 \Leftrightarrow i \in M, \text{ und } A[i] = 0 \text{ sonst.}$$

Operationen:

- $i \in M$?
- füge i zu M hinzu?
- entferne i aus M?

- $M = M_1 \cup M_2$?
- $M = M_1 \cap M_2$?
- $M = M_1 - M_2$?
- $M = M_1 \Delta M_2$?

Machinenwortdarstellung: w Bits in einem Machinenwort. Komplexität entweder $O(1)$ oder $O(\lceil n/w \rceil)$