



Algorithmen und Datenstrukturen

Einfache Datenstrukturen

Matthias Teschner
Graphische Datenverarbeitung
Institut für Informatik
Universität Freiburg

SS 09

Lernziele der Vorlesung



- Algorithmen
 - Sortieren, Suchen, Optimieren
- Datenstrukturen
 - Repräsentation von Daten
 - Listen, Stapel, Schlangen, Bäume
- Techniken zum Entwurf von Algorithmen
 - Algorithmenmuster
 - Greedy, Backtracking, Divide-and-Conquer
- Analyse von Algorithmen
 - Korrektheit, Effizienz

Überblick



- Einführung
- Feld
- Verkettete Liste
- Stapel und Schlangen
- Anwendungen

Datenstruktur



- Algorithmen manipulieren **dynamische Mengen** von Elementen (Eingabe → Ausgabe)
 - Suchen, Einfügen, Löschen
 - Minimum, Maximum, nächstkleinstes oder nächstgrößtes Element
- Datenstrukturen werden zur Realisierung (Repräsentation) dynamischer Mengen verwendet.
- **Datenstrukturen sind unterschiedlich effizient in Bezug auf Manipulationen (Operationen).**
- Sinnvolle Wahl einer Datenstruktur hängt von der Effizienz der darin implementierten Operationen ab, die für einen gegebenen Algorithmus relevant sind.

Element



- Datenstrukturen repräsentieren Menge von Elementen / Datensätzen.
- Elemente bestehen aus Attributen.
- Schlüssel sind ein oder mehrere ausgezeichnete Attribute, über die ein Element identifiziert wird.
- Such- und Sortieralgorithmen verwenden Schlüssel als Kriterium.

- Beispiel:

```
class Element {  
    int key;  
    infoKlasse info;  
}
```

Schlüssel zur eindeutigen Identifikation eines Elements

weitere Informationen zum Element

Operationen auf einer Menge D von Elementen



- `INIT (D)`
Initialisierung von D als leere Menge
- `INSERT (D, x)`, `DELETE (D, x)`
Einfügen / Löschen von Element mit Schlüssel x
- `SEARCH (D, x)`
Suche Element mit Schlüssel x
- `SIZE (D)`
Anzahl der Elemente in D
- `MAX (D)`, `MIN (D)`
Maximum / Minimum von D liefert das Element mit dem größten / kleinsten Schlüssel
- `SUCC (D, x)`, `PRED (D, x)`
Nachfolger / Vorgänger von Element mit Schlüssel x liefert Element mit nächstgrößerem / nächstkleineren Schlüssel

Implementierung von Mengen



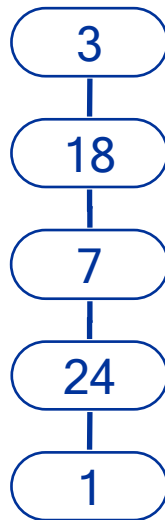
- Implementierungen von Datenstrukturen sind durch unterschiedliche Laufzeiten für verschiedene Operationen charakterisiert.
- **statische Datenstrukturen**
 - Feld

Die Größe eines Feldes kann während der Laufzeit eines Programms nicht verändert werden.
- **dynamische Datenstrukturen**
 - Liste, verkettet oder doppelt verkettet
 - Baum
 - Graph

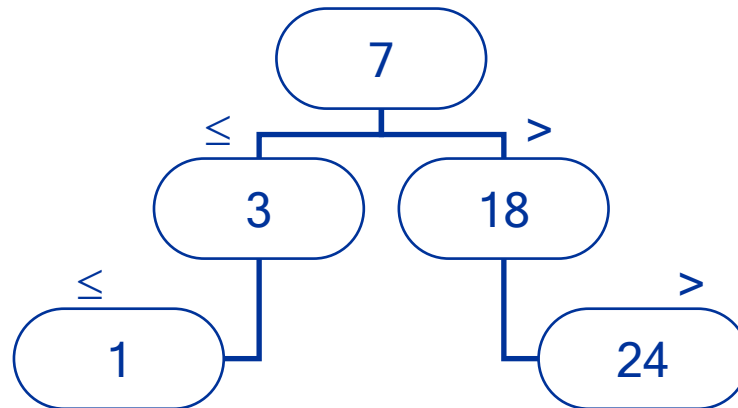
Motivation



- alternative Repräsentation einer Menge von Zahlen



Liste

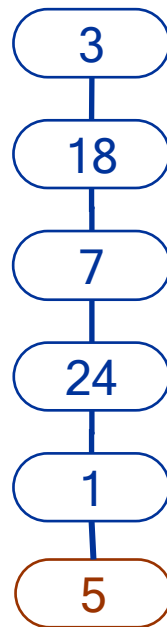


Binärbaum

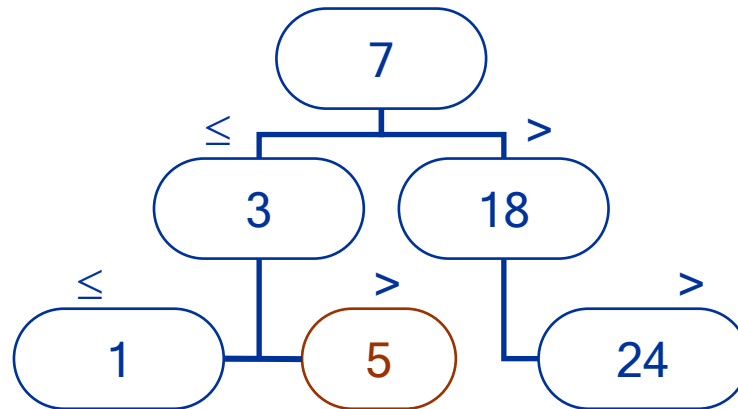
Motivation



- Einfügen eines weiteren Elements "5"



Liste



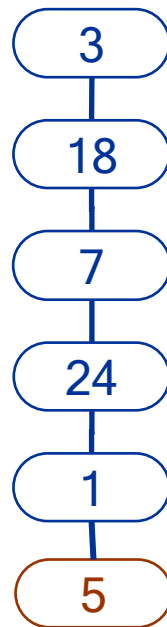
Binärbaum

- aufwändiger für den Binärbaum

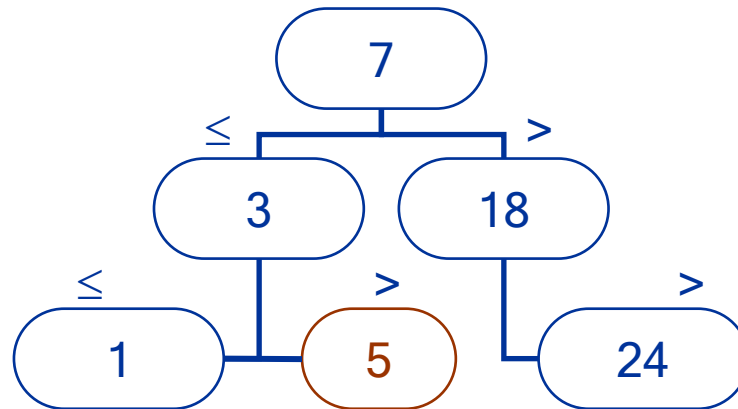
Motivation



- Suchen des Elements "5"



Liste



Binärbaum

- effizienter für den Binärbaum

Beispiele für Datenstrukturen



- **Feld**
 - Zugriff auf ein Element über einen Index
- **Liste**
 - Element besitzt Verweis auf das folgende Element
- **Stapel**
 - Elemente können nur in umgekehrter Reihenfolge des Einfügens gelesen oder gelöscht werden
- **Warteschlange**
 - Elemente können nur in gleicher Reihenfolge des Einfügens gelesen oder gelöscht werden
- **Graphen, Bäume**
 - Elemente besitzen variable Anzahl von Verweisen auf weitere Elemente

Überblick



- Einführung
- **Feld**
- Verkettete Liste
- Stapel und Schlangen
- Anwendungen

Feld



- Feld mit fester Zahl von Elementen kann für die Repräsentation einer dynamischen Menge verwendet werden.
- Beispiel: `int[] a = new int[amax+1];`
generiert Feld mit `amax+1` Elementen

0	1	2	...	n	...	amax
n	a_1	a_2	...	a_n	...	a_{amax}

Index für Feld a

$a[0]$ kann die Zahl der Elemente der Menge enthalten, die durch a repräsentiert wird (nicht zu verwechseln mit `a.length()`).
`a.length() = amax` gibt die maximale Zahl von Elementen an, die mit a repräsentiert werden können.

$a[1] \dots a[n]$ enthalten die Elemente der zu repräsentierenden Menge.
 $a[n+1] \dots a[amax]$ sind undefiniert.

- Elemente sind üblicherweise nach Kriterien angeordnet, z. B. sortiert

Operationen



- Initialisierung

```
int[] a = new int[amax+1];
```

```
a[0] = 0;          a[0] ist in unserem Beispiel der Zähler für die verwalteten Elemente
```

- Element *e* an Position *insertAt* in Feld *a* einfügen: $O(n)$

```
if (a[0]<amax)          Ist noch Platz für ein weiteres Element?
{
    if (insertAt > 0 && insertAt <= a[0]) Einfügen innerhalb oder am Anfang von a
    {
        for (int i=a[0]+1; i>insertAt; i--) Umkopieren von O ( a[0]=n ) Elementen
            a[i] = a[i-1];
        a[insertAt] = e;          Einfügen des Elements
        a[0]++;
    }
    else if (insertAt == a[0]+1)      Einfügen am Ende von a
    { a[insertAt] = e; a[0]++; }
}
else ... - erzeuge Feld doppelter Größe 2*(amax+1)
         - kopiere alle Elemente vom alten ins neue Feld, lösche altes Feld
         - füge weiteres Element ein
```

Operationen



- Element e an Position deleteAt löschen: $O(n)$

```
if (deleteAt > 0 && deleteAt <= a[0])
{
    for (int i=deleteAt+1; i<=a[0]; i++)
        a[i-1] = a[i];
    a[0]--;
}
```

durchschnittlich
 $O(a[0])$ Operationen

Operationen



- Element e suchen in $O(n)$

```
for (int i=1; i<=a[0]; i++)  
    if (a[i]==e) return i;  
return -1;
```

durchschnittlich
 $O(a[0])$ Operationen

- Zugriff auf das i -te Element: $O(1)$
- Anzahl der Elemente in $O(1)$: $a[0]$
- wenn das Feld geordnet ist, dann
 - Element mit größtem Schlüssel in $O(1)$: $a[a[0]]$
 - Element mit kleinstem Schlüssel in $O(1)$: $a[1]$

Überblick

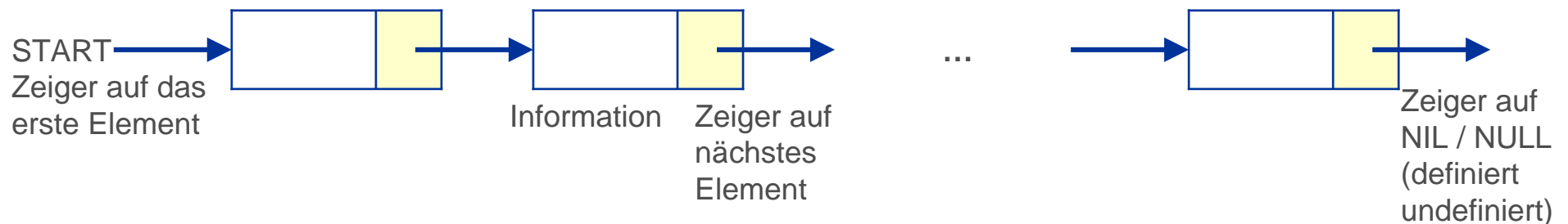


- Einführung
- Feld
- Verkettete Liste
- Stapel und Schlangen
- Anwendungen

Verkettete Liste



- dynamische Datenstruktur
- Zahl der Elemente während der Laufzeit frei wählbar
- Elemente bestehen aus einfachen oder zusammengesetzten Datentypen
- Elemente sind durch Zeiger / Referenzen auf ein folgendes Element verbunden
- einfache oder doppelte Verkettung



Eigenschaften im Vergleich zum Feld



- geringer Mehrbedarf an Speicher
- Einfügen und Löschen von Elementen erfolgt ohne Umkopieren anderer Elemente
- Zahl der Elemente kann beliebig verändert werden
- kein direkter Zugriff auf Elemente (Liste muss durchlaufen werden)

Varianten

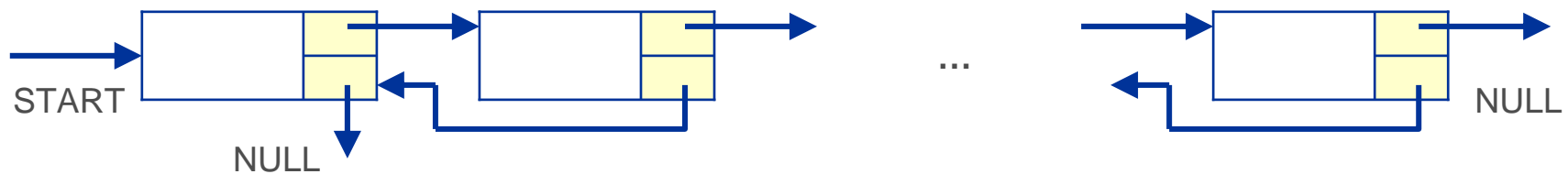


■ Liste mit Listenkopf und Ende-Zeiger



- enthält Zeiger auf erstes Element
- kann weitere Informationen enthalten (z. B. Zahl der Elemente)

■ doppelt verkettete Liste (ein Zeiger zum folgenden Element, ein zweiter Zeiger zum vorherigen Element)



Element / Knoten Implementierung



basiert auf Mary K. Vernon:
"Introduction to Data Structures"

<http://pages.cs.wisc.edu/~vernon/cs367/cs367.html>

```
public class Listnode  
{
```

```
    private int data;  
    private Listnode next;
```

```
    public Listnode(int d)  
    { data = d; next = null; }
```

```
    public Listnode(int d, Listnode n)  
    { data = d; next = n; }
```

```
    public int getData() { return data; }  
    public void setData(int d) { data = d; }
```

```
    public Listnode getNext() { return next; }  
    public void setNext(Listnode n) { next = n; }
```

```
}
```

2 Felder: Daten (hier lediglich ein Integer) und ein Zeiger / eine Referenz auf Listnode
private: nur innerhalb der Klasse zugreifbar

2 Konstruktoren: Initialisierung von Instanzen der Klasse

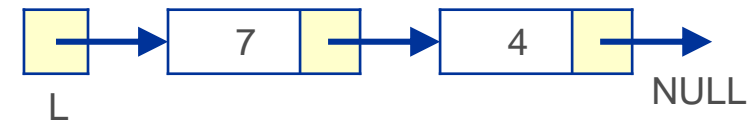
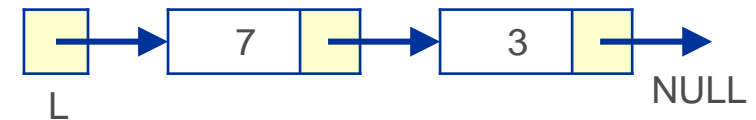
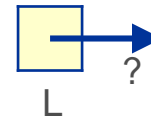
Funktionen zum Lesen und Schreiben der private-Felder.
Kapselung: Erlaubt die Einhaltung von Zusicherungen an den Inhalt der Felder.

Die Daten eines Knoten sind hier durch "int" vereinfacht repräsentiert. Üblicherweise verwendet man hier selbst definierte Referenzdatentypen, z. B. Object data; . In dem Fall reservieren die Konstruktoren lediglich Speicher für Zeiger / Referenzen auf den Datentyp Object.

Beispiele



- Listnode L;
- L = new Listnode(7);
- L.setNext(new Listnode(3));
- L.getNext().setData(4);

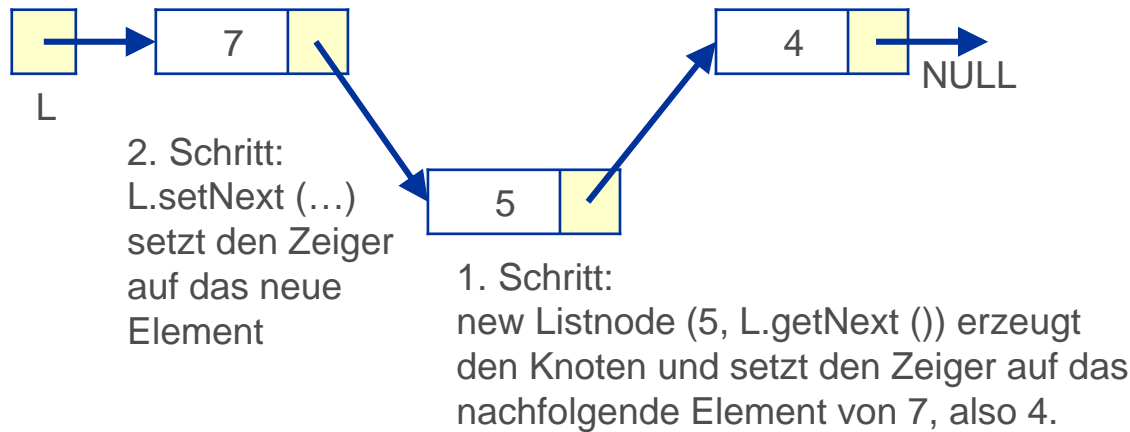


Beispiele



- Einfügen zwischen dem ersten und dem zweiten Element

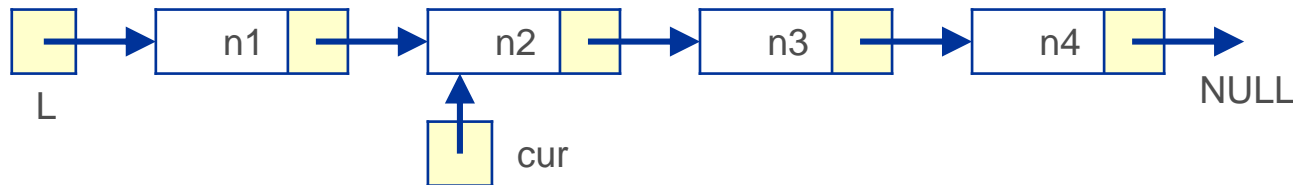
```
L.setNext( new Listnode(5, L.getNext()) );
```



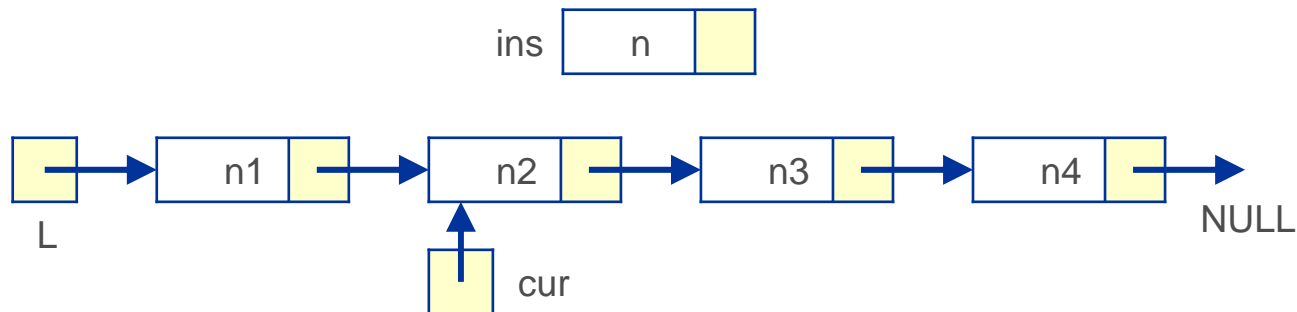
Einfügen eines Knotens



- Einfügen eines Knotens ins hinter einem Knoten cur



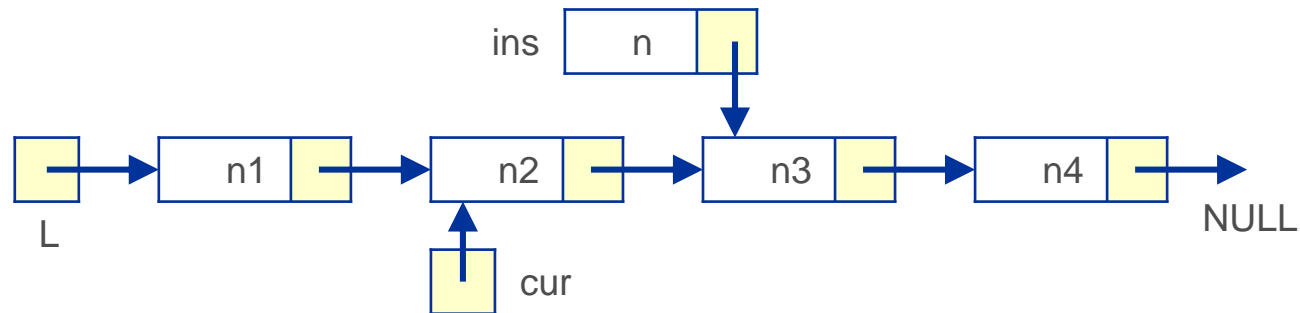
```
Listnode ins = new Listnode (n);
```



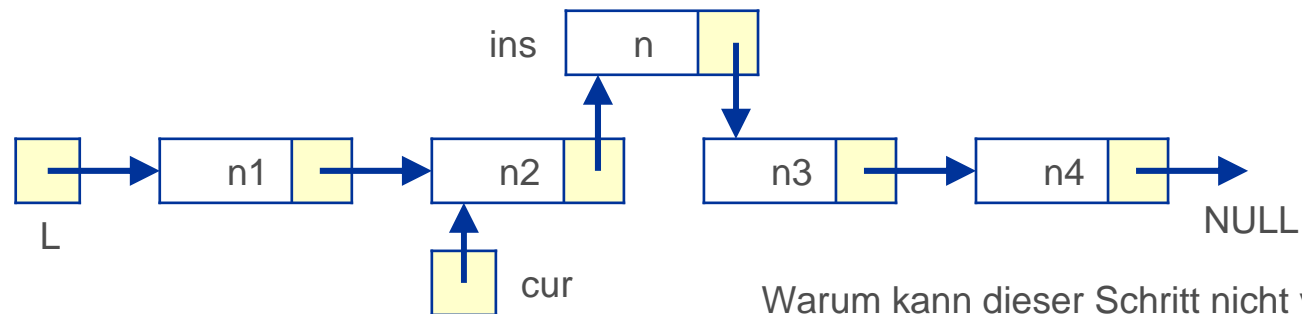
Einfügen eines Knotens



- `ins.setNext (cur.getNext());`



- `cur.setNext (ins);`

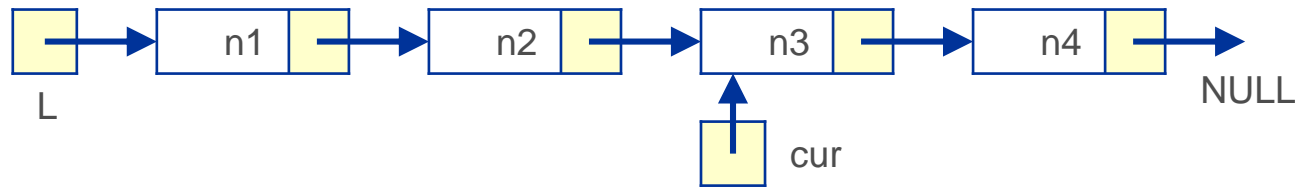


Warum kann dieser Schritt nicht vor `insert.setNext` durchgeführt werden?

Löschen eines Knotens



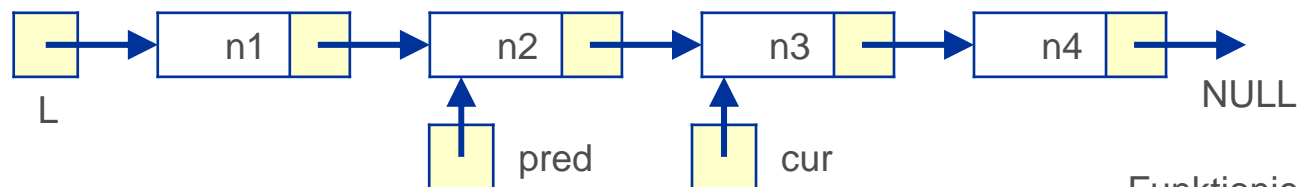
- Löschen eines Knotens cur



- Finde den Vorgängerknoten pred

```
Listnode pred = L;  
while (pred.getNext() != cur)  
    pred = pred.getNext();
```

Laufe elementweise durch die
Liste, bis pred.next auf cur zeigt.
 $O(n)$



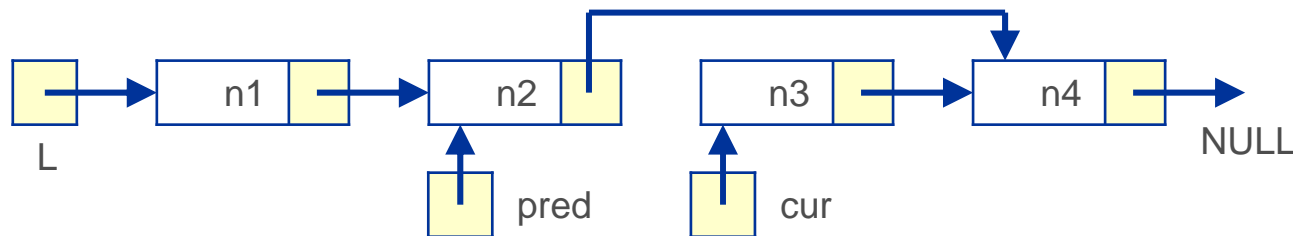
Funktioniert nicht für
den ersten Knoten!

Löschen eines Knotens



- setze den next-Zeiger des Vorgängers von cur auf den Nachfolger von cur

```
pred.setNext( cur.getNext() );
```

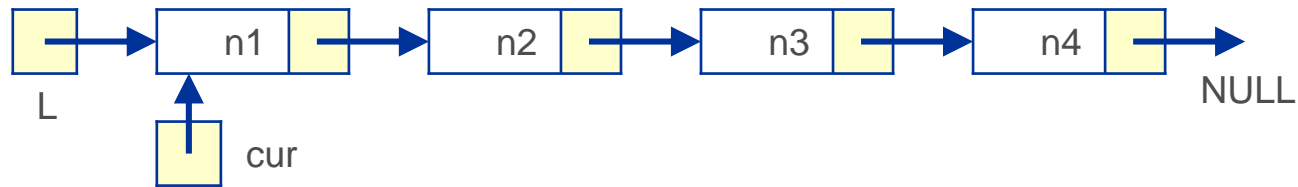


- Wenn keine weitere Referenz auf die durch cur angegebene Instanz existiert und cur auf eine andere Instanz gesetzt wird, ist das gelöschte Element nicht mehr zugreifbar. In C++ kann der belegte Speicher durch delete als Gegenstück zu new freigegeben werden.

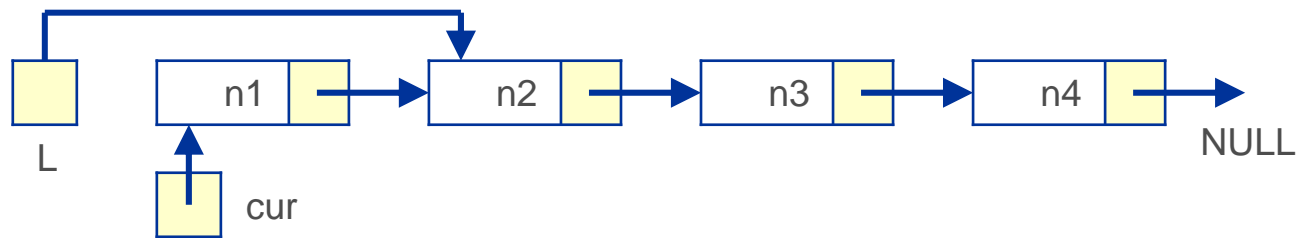
Sonderfall



- Löschen des ersten Knotens



- `if (cur == L) { L = cur.getNext(); }`



Löschen eines Knotens



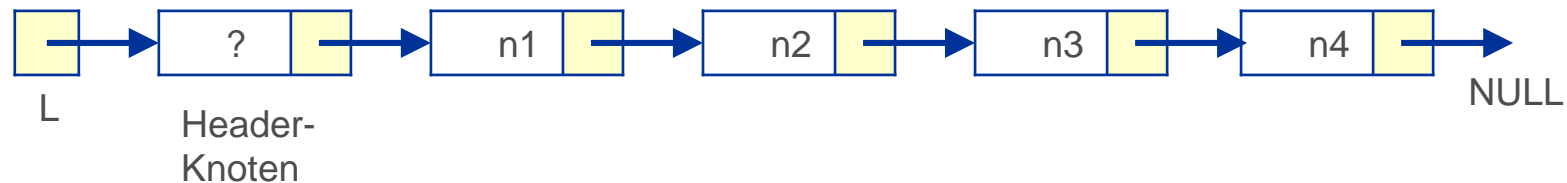
- generelles Löschen

```
if (cur == L)
{
    L = cur.getNext();
}
else
{
    Listnode pred = L;
    while (pred.getNext() != cur)
        pred = pred.getNext();
    pred.setNext(cur.getNext());
}
```

Vermeidung des Sonderfalls



- Header-Knoten



- Vorteil

- Löschen des ersten Knotens ist kein Sonderfall

- Nachteil

- weitere Funktionen müssen berücksichtigen, dass der erste Knoten nicht zur Liste gehört, z. B. Ausgabe aller Elemente, Zählen der Elemente

Verkettete Liste (mit Header-Knoten)



```
public class List
{
    private Listnode first, last;
    private int numItems;

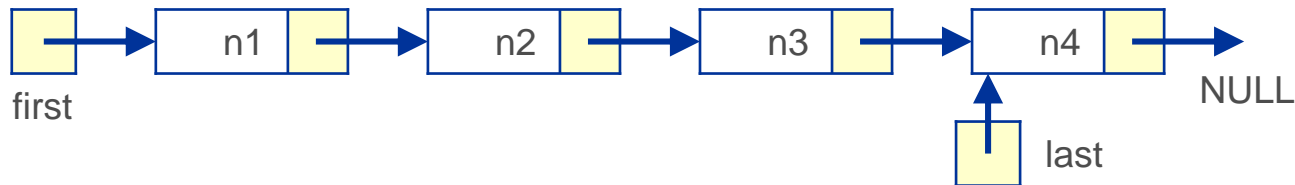
    public List() { ... }

    public int size() { return numItems; }
    public boolean isEmpty() { return first==last; }

    public void add (int data) { ... }
    public int add (int pos, int data) { ... }
    public int remove (int pos) { ... }
    public Listnode get (int pos) { ... }
    public boolean contains (int data) { ... }
}
```

isEmpty ist nur korrekt, wenn ein Header-Knoten vorhanden ist.

first, last, (cur)

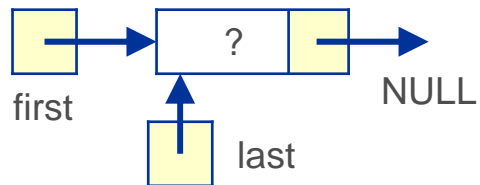


- First zeigt auf das erste Element, last auf das letzte.
- Last erlaubt Einfügen von Elementen am Ende in $O(1)$, Liste muss nicht durchlaufen werden.
- Last muss durch alle Operationen aktuell gehalten werden.
- Oft wird noch eine dritte Referenz *cur* auf einen aktuellen Knoten in der Liste mitgeführt.

Konstruktor (mit Header-Knoten)



```
■ public List ()  
  {  
    first = last = new Listnode(null);  
    numItems = 0;  
  }
```



Einfügen am Ende der Liste



```
■ public void add (int data)
  {
    last.setNext( new Listnode(data) );
    last = last.getNext();
    numItems++;
  }
```

Durchlaufen der Liste wird durch last-Zeiger vermieden.
last-Zeiger und Zahl der Elemente numItems müssen aktualisiert werden.

Einfügen an Position pos



```
public int add (int data, int pos)
{
    if (pos < 0 || pos > numItems)
        return -1;
    if (pos == numItems) add(data);
    else
    {
        Listnode cur = first;
        for (int k=0; k<pos; k++)
            cur = cur.getNext();
        cur.setNext
            (new Listnode(data, cur.getNext()));
        numItems++;
    }
    return 0;
}
```

Füge am Ende ein.
numItems und last
gehen uns nichts an.

Laufe durch die Liste
bis Position pos.

first wird nicht verändert,
da first auf den Header-
Knoten zeigt.

last wird nicht verändert,
da nicht am Ende
eingefügt wird.

Wie gesagt: cur könnte man durchaus als Feld in der Klasse berücksichtigen.
pos = 0 fügt ein Element nach dem "0"-ten, also vor dem ersten Element ein.

Löschen an Position pos



```
■ public int remove (int pos)
  {
    if (pos < 1 || pos > numItems)
      return -1;

    Listnode pred = first;
    for (int k=0; k<pos-1; k++)
      pred = pred.getNext();

    Listnode cur = pred.getNext();
    pred.setNext(cur.getNext());

    numItems--;
    if (cur.getNext()==null) last=pred;
    return 0;
  }
```

Lösche das
Element cur

Finde den Vorgänger des
Elementes an Position pos

Finde den Nachfolger des
Vorgängers, also das zu
löschende Element.

Referenz auf Element an Position pos



```
■ public Listnode get (int pos)
  {
    if (pos < 1 || pos > numItems)
      return null;

    Listnode cur = first;
    for (int k=0; k<pos; k++)
      cur = cur.getNext();

    return cur;
  }
```

Laufe durch die Liste
bis Position pos.

Referenz auf Element an Position pos



```
■ public boolean contains (int data)
{
    Listnode cur = first;

    for (int k=0; k<numItems; k++)
    {
        cur = cur.getNext();
        if (cur.getData() == data) return true;
    }

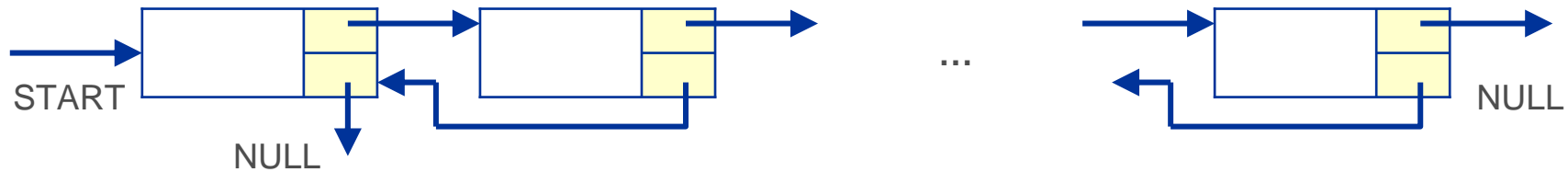
    return false;
}
```

Erstes Element ist
der Header, kein
Element unserer
Liste.

Variante



■ Doppelt verkettete Liste



```
public class Listnode
{
    private int data;
    private Listnode prev, next;

    public Listnode(int d)
    { data = d; prev = next = null; }

    public Listnode(int d, Listnode p, Listnode n)
    { data = d; prev = p; next = n; }

    public int getData() { return data; }
    public void setData(int d) { data = d; }

    public Listnode getNext() { return next; }
    public void setNext(Listnode n) { next = n; }
    public Listnode getPrev() { return prev; }
    public void setPrev(Listnode p) { prev = p; }
}
```

Es gibt auch zyklische Listen, bei denen dann das erste und das letzte Element verbunden werden. Ein Zeiger auf ein ausgewähltes Element (Wächter, first, START) existiert auch dort.

Überblick



- Einführung
- Feld
- Verkettete Liste
- Stapel und Schlangen
- Anwendungen

Schlange

(Warteschlange, Queue)



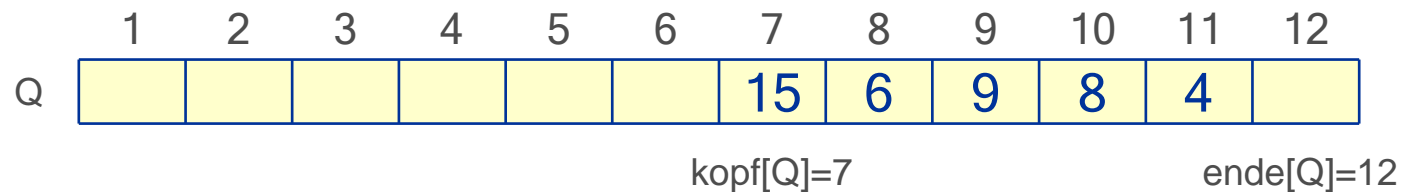
- dynamische Menge mit speziellen, optimierten Zugriffsoperationen
- Einfügen eines Elements (enqueue) in $O(1)$
- Zugriff / Entfernen eines Elements (dequeue) in $O(1)$
- FIFO-Prinzip (first in - first out)
 - Zugriff auf das Element, das am längsten in der Datenstruktur "wartet" (z. B. Warteschlange von Kunden an einer Kasse)
- Realisierung als verkettete Liste oder als Feld
- Kopf und Ende der Schlange sind bekannt
 - Elemente werden am Ende eingefügt
 - Elemente werden vom Kopf gelesen / gelöscht

Realisierung durch Feld

Ringpuffer



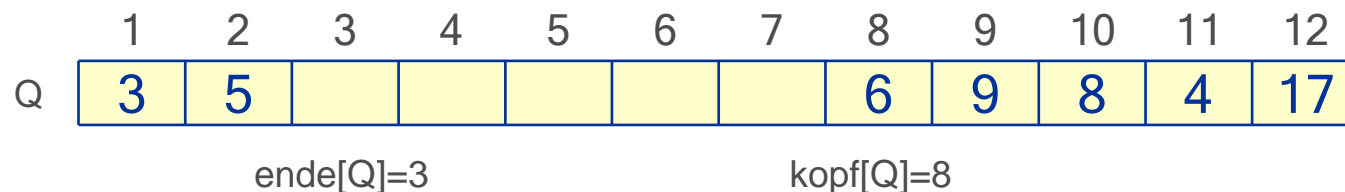
- **Feld Q fester Größe** Initialisierung der leeren Schlange: z. B. $\text{kopf}[Q]=-1$; $\text{ende}[Q]=1$



- **enqueue(Q,17); enqueue(Q,3); enqueue(Q,5);**



- **dequeue(Q);** liefert $Q[\text{kopf}[Q]]=15$



Implementierung



- enqueue (Q, x)
 Q [ende[Q]] = x;
 if (ende[Q] == laenge[Q])
 ende[Q] = 1;
 else
 ende[Q] = ende[Q]+1;
- dequeue (Q)
 x = Q [kopf[Q]];
 if (kopf[Q] == laenge[Q])
 kopf[Q] = 1;
 else
 kopf[Q] = kopf[Q]+1;
 return x;

Probleme



- Unterlauf (Puffer-Unterlauf, underflow, buffer underflow)
 - dequeue bei leerer Menge $\text{kopf}[Q] = -1$
- Überlauf (Puffer-Überlauf, overflow, buffer overflow)
 - enqueue bei vollem Feld $\text{ende}[Q] == \text{kopf}[Q]$
- Überlauf kann auch toleriert werden
 - z. B. wenn man lediglich an den zuletzt eingefügten Daten interessiert ist
 - bei $\text{ende}[Q] == \text{kopf}[Q]$ wird das alte Element $Q[\text{kopf}[Q]]$ durch ein neues Element überschrieben. $\text{ende}[Q]$ und $\text{kopf}[Q]$ werden aktualisiert.

Anwendungen



- Kommunikation asynchroner Funktionen
 - Ergebnisse einer Funktion dienen als Eingabe für eine weitere Funktion
 - Ergebnisse der ersten Funktion werden gepuffert (in die Warteschlange eingefügt) und von der zweiten Funktion aus dem Puffer entnommen (aus der Warteschlange gelesen und gelöscht)
- Kommunikation von Geräten mit Prozessen
 - Graphische Benutzeroberflächen puffern Maus- und Tastaturereignisse in einer Message Queue
 - Drucker-Queue
- Parallelisierung
 - Elemente einer Queue werden von mehreren parallelen Prozessen gelesen und verarbeitet

Anwendungen

Topologische Sortierung

Wikipedia
Topologische Sortierung

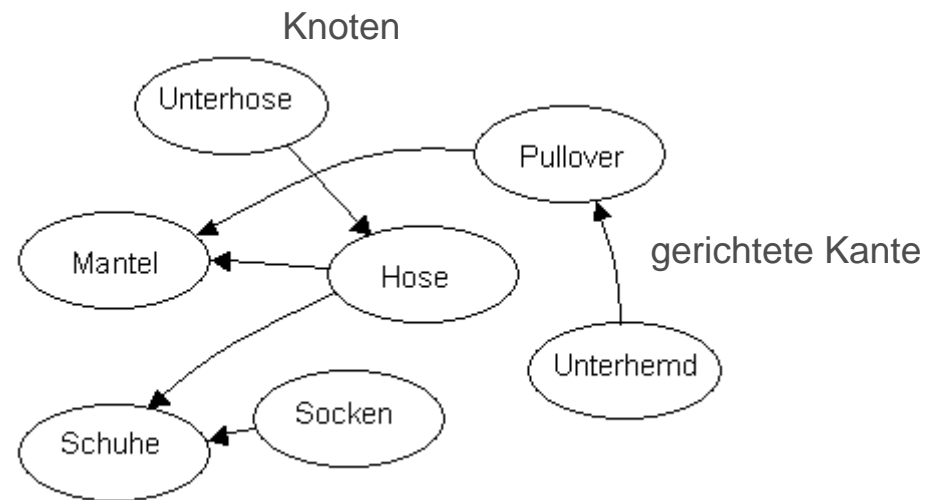


- Topologische Sortierung
 - Sortierung von Elementen, für die Abhängigkeiten gelten
- Beispiel:
 - Menge:
{Hose, Unterhemd, Pullover, Mantel, Socken, Unterhose, Schuhe}
 - Abhängigkeiten: $A \rightarrow B$ (erst A anziehen, dann B)
Unterhemd \rightarrow Pullover, Unterhose \rightarrow Hose, Pullover \rightarrow Mantel,
Hose \rightarrow Mantel, Hose \rightarrow Schuhe, Socken \rightarrow Schuhe
 - Topologische Sortierung (nicht eindeutig):
Unterhose, Socken, Hose, Unterhemd, Pullover, Mantel, Schuhe
oder auch
Unterhemd, Unterhose, Pullover, Socken, Hose, Schuhe, Mantel
nicht jedoch
Pullover, Unterhemd, ... (da Unterhemd \rightarrow Pullover)

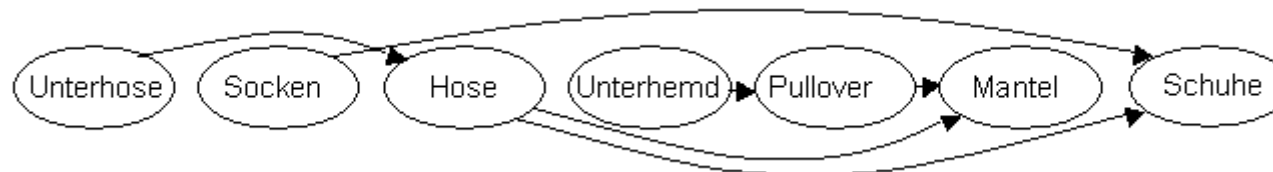
Darstellung als gerichteter Graph



- Abhängigkeiten



- Topologische Sortierung

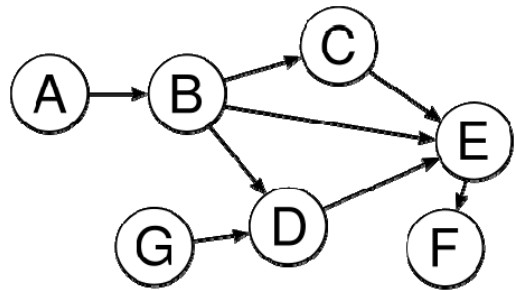


lineare Anordnung der Knoten, in der die gerichteten Kanten (Abhängigkeiten) immer von links nach rechts orientiert sind.

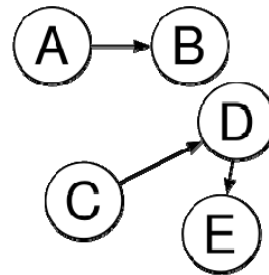
Sortierbarkeit



- sortierbare Graphen (azyklische Graphen)

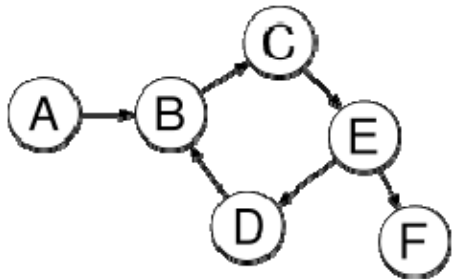


z. B. A, B, C, G, D, E, F

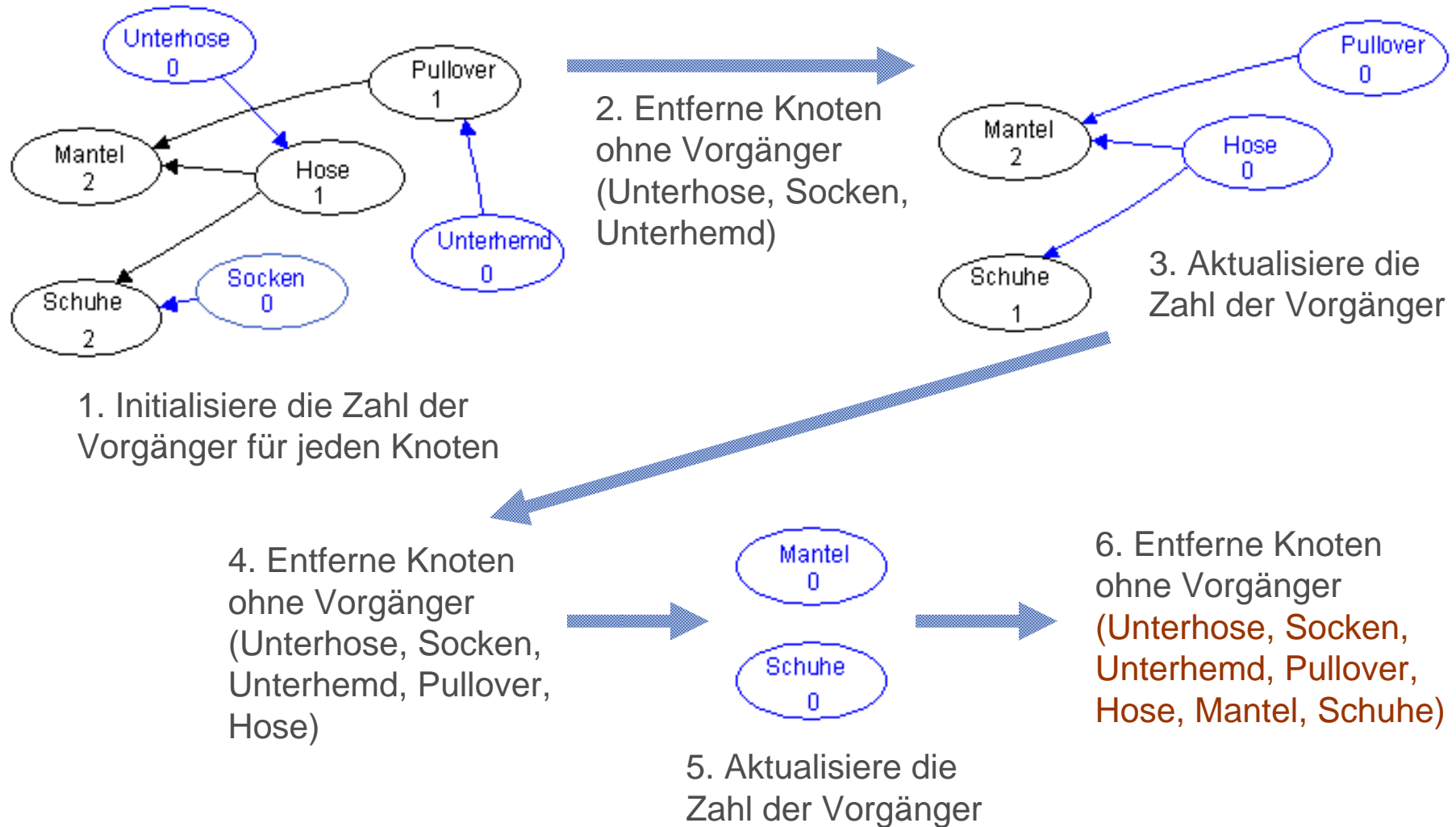


z. B. A, C, B, D, E

- nicht sortierbare Graphen (zyklische Graphen)



Prinzip der Sortierung



Implementierung mit einer Schlange



- Menge: {1, 2, 3, 4, 5, 6, 7, 8, 9}
- Abhängigkeiten: (1,3), (3,7), (7,4), (4,6), (9,2), (9,5), (2,8), (5,8), (8,6), (9,7), (9,4)
- Vorverarbeitung: Aufbau von zwei Feldern

	1	2	3	4	5	6	7	8	9
Feld A (Zahl der Vorgänger eines Knotens i)	0	1	1	2	1	2	2	2	0
Feld L (Liste der Nachfolger mit Knoten i als Vorgänger)	3	8	7	6	8		4	6	2 5 7 4

Implementierung mit einer Schlange



	1	2	3	4	5	6	7	8	9
Feld A (Zahl der Vorgänger eines Knotens i)	0	1	1	2	1	2	2	2	0
Feld L (Liste der Nachfolger mit Knoten i als Vorgänger)	3	8	7	6	8		4	6	2 5 7 4

- Lege Elemente ohne Vorgänger in eine Schlange: 1, 9
- Verarbeite alle Elemente der Schlange
 - reduziere die Zahl der Vorgänger für alle Nachfolger des Elements um eins (1 hat 3 als Nachfolger → $A[3]--$, analog für Nachfolger von 9)
 - gib das Element aus, markiere Element als verarbeitet: 1, 9

	1	2	3	4	5	6	7	8	9
A	-1	0	0	1	0	2	1	2	-1
L	3	8	7	6	8		4	6	2 ...

Implementierung mit einer Schlange



	1	2	3	4	5	6	7	8	9
A	-1	0	0	1	0	2	1	2	-1
L	3	8	7	6	8		4	6	2 ...

- Lege Elemente ohne Vorgänger in die Schlange: 2,3,5
- Verarbeite alle Elemente der Schlange
 - reduziere die Zahl der Vorgänger für alle Nachfolger des Elements
 - gib das Element aus, markiere Element als verarbeitet: 2,3,5
- Zwischenergebnis: 1,9,2,3,5

	1	2	3	4	5	6	7	8	9
A	-1	-1	-1	1	-1	2	0	0	-1
L	3	8	7	6	8		4	6	2 ...

Implementierung mit einer Schlange



	1	2	3	4	5	6	7	8	9
A	-1	-1	-1	1	-1	2	0	0	-1
L	3	8	7	6	8		4	6	2 ...

- Verarbeite 7 und 8, Zwischenergebnis: 1,9,2,3,5,7,8

	1	2	3	4	5	6	7	8	9
A	-1	-1	-1	0	-1	1	-1	-1	-1
L	3	8	7	6	8		4	6	2 ...

- Verarbeite 4, Zwischenergebnis: 1,9,2,3,5,7,8,4
- Verarbeite 6, Zwischenergebnis: 1,9,2,3,5,7,8,4,6
- keine neuen Elemente ohne Vorgänger → fertig

Pseudocode



- n Elemente 1...n, p Abhängigkeiten $\text{pred}[i] \rightarrow \text{succ}[i]$
- $Q := \emptyset$;
for $i:=1$ to n do { $A[i]:=0$; $L[i]:= \emptyset$; }
for $i:=1$ to p do {
 $A[\text{pred}[i]]++$; $L[i].\text{add}(\text{succ}[i])$; }
for $i:=1$ to n do
 if ($A[i]==0$) $Q.\text{enqueue}(i)$;
while ($Q.\text{empty}()==\text{false}$) {
 $j := Q.\text{dequeue}()$;
 print j ;
 forAllElements k in $L[j]$ do {
 $A[k]--$;
 if ($A[k]==0$) $Q.\text{enqueue}(k)$; } }
}

Überblick



- Einführung
- Feld
- Verkettete Liste
- Stapel und Schlangen
- Anwendungen

Stapel

(Kellerspeicher, Stack)

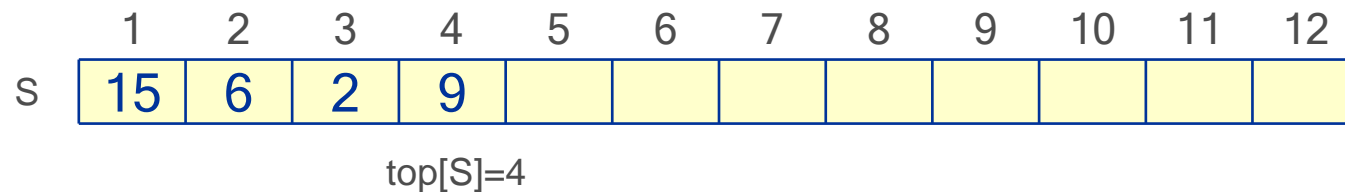


- dynamische Menge mit speziellen, optimierten Zugriffsoperationen
- Einfügen eines Elements (push) in $O(1)$
- Zugriff / Entfernen eines Elements (pop) in $O(1)$
- LIFO-Prinzip (last in - first out)
 - Elemente werden in umgekehrter Einfüge-Reihenfolge gelesen / gelöscht (Tellerstapel, Zugriff nur auf obersten Teller)
- Realisierung als verkettete Liste oder als Feld
- Oberstes Element des Stapels (top) ist bekannt
 - Elemente werden hinter / über dem obersten Element eingefügt
 - Oberstes Element wird gelesen / gelöscht

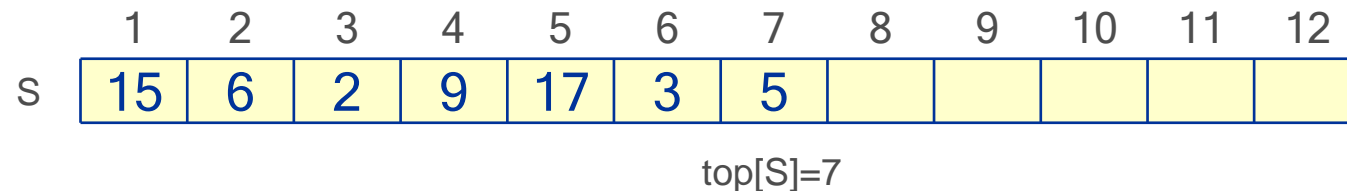
Realisierung durch Feld



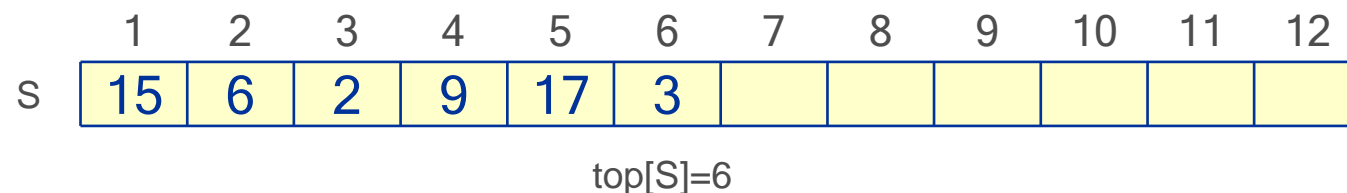
- **Feld S fester Größe** Initialisierung des leeren Stapels: z. B. $\text{top}[S] = -1$



- **push(S, 17); push(S, 3); push(S, 5);**



- **pop(S);** liefert $S[\text{top}[S]] = 5$



Implementierung



- `push (S, x)`
`top[S] = top[S] + 1;`
`S[top[S]] = x;`
- `pop (S)`
`x = S[top[S]];`
`top[S] = top[S] - 1;`
`return x;`
- Behandlung von Unter- / Überlauf muss noch ergänzt werden.

Anwendungen



- Evaluation der Verschachtelung von Klammersausdrücken
 - $a(a(b))$ korrekt, $a(a(b)$ fehlerhaft
- Auswertung korrekt geklammerter Ausdrücke
 - $(a + b) \cdot (c + d) + e$
- Iterative Auswertung rekursiver Funktionen

Überblick



- Einführung
- Feld
- Verkettete Liste
- Stapel und Schlangen
- **Anwendungen**
 - Evaluation von Verschachtelungen
 - Auswertung von Klammerausdrücken
 - Rekursive Funktionen

Evaluation von Verschachtelungen



- Realisierung durch Klammer-Stapel
- Ignorieren von Über- / Unterlauf

```
■ while ( x=leseZeichen() )  
  {  
    if ( x=='(' ) push (S,x);  
    if ( x==')' ) pop (S);  
  }
```

```
if ( empty(S)==true ) gleiche Anzahl öffnender und schließender Klammern ?  
  return inOrdnung;  
else  
  return fehlerhaft;
```

Beispiel



- Term
Stapel

(a · (b · c) + d) · (e + f)															
			((((
((((((((((((((((

gleiche
Zahl von
(und).

- Erweiterung für verschiedenartige Klammern
 - teste, ob oberstes Element mit schließender Klammer übereinstimmt

- Term
Stapel

{ a - [(b + c) - (d - e)] · f (x - y) }															
				((((((((
{	{	{	{	[[[[[[[[[[[[
{	{	{	{	{	{	{	{	{	{	{	{	{	{	{	{

Fehler !
] passt
nicht zu { .

Evaluation von Verschachtelungen



- verschiedenartige Klammern, ignorieren von Über- / Unterlauf
- ```
while (x=leseZeichen())
{
 if (x=='(' || x=='[' || x=='{')
 push (S,x);
 if (x==')' || x==']' || x=='}')
 {
 y = pop (S);
 if (!passtZusammen(x,y)) return fehlerhaft;
 }
}

if (empty(S)==true)
 return inOrdnung;
else
 return fehlerhaft;
```

# *Auswertung von Termen*

## *Prinzip*



- Realisierung durch zwei Stapel (Operator- und Operandenstapel)
- Lese Term zeichenweise
- Speicher Operatoren auf Operatorstapel
- Speicher Operanden auf Operandenstapel
- Wenn passende Zahl von Operanden für obersten Operator vorhanden, entferne die entsprechenden Operanden und den Operator, werte Term aus, speicher Ergebnis auf dem Operandenstapel

# Illustration



- $\text{max3}(a,b,c)$
- Operator  $\text{max3}$  auf Operatorstapel
- $a$ ,  $b$  und  $c$  auf Operandenstapel
- Zahl der gelesenen Operanden ( $a$ ,  $b$ ,  $c$ ) stimmt mit der benötigten Operandenzahl des obersten Operators ( $\text{max3}$ ) überein.
- $a$ ,  $b$ ,  $c$  und  $\text{max3}$  werden von den Stapeln gelesen und entfernt.
- Term  $\text{max3}(a,b,c)$  wird ausgewertet und auf Operandenstapel gespeichert.

# Beispiel



- $(a+b) \cdot c + (d+e) \cdot f + g$

| (a + b) |   |   | · c |    |   | + (d + e) |    |   | · f |    |   | + g |    |   |    |    |   |    |    |    |
|---------|---|---|-----|----|---|-----------|----|---|-----|----|---|-----|----|---|----|----|---|----|----|----|
| a       | a | b | t1  | t1 | c | t2        | t2 | d | t2  | t2 | e | t3  | t3 | f | t4 | t4 | g | t5 | t5 | t6 |
|         | + | + |     | ·  | · |           | +  | + |     | +  | + |     | ·  | · |    | +  | + |    | +  |    |

- t1 bis t6 sind Zwischenergebnisse, z. B.  $t1 = a + b$
- Klammern und Prioritätenregeln müssen in der Implementierung berücksichtigt werden.

# *Iterative Implementierung rekursiver Funktionen - Prinzip*



- Realisierung durch Funktionen-Stapel
- Speicher das Anfangsproblem auf dem Stapel
- Solange Stapel nicht leer
  - lese das oberste Problem und löse es
  - falls bei der Lösung neue Teilprobleme entstehen, speicher diese auf dem Stack
- Stack leer = Problem gelöst

# Beispiel



- Binomialkoeffizient

- gibt die Zahl der Möglichkeiten an, k Objekte aus einer Menge mit n verschiedenen Elementen auszuwählen

- $$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

- "49 über 6" gibt die Zahl der möglichen Ziehungen beim Lotto an.

- Binomischer Satz

$$(x+y)^n = \binom{n}{0} x^n y^0 + \binom{n}{1} x^{n-1} y^1 + \dots + \binom{n}{n-1} x^1 y^{n-1} + \binom{n}{n} x^0 y^n$$

# Rekursive Formulierung



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

# Beispiel

$$\begin{array}{l} \begin{pmatrix} 4 \\ 2 \end{pmatrix} \\ \begin{pmatrix} 3 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 3 \\ 2 \end{pmatrix} \\ \begin{pmatrix} 3 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 2 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 2 \\ 2 \end{pmatrix} \\ \begin{pmatrix} 3 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 2 \\ 1 \end{pmatrix} \\ \begin{pmatrix} 3 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ \begin{pmatrix} 3 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ \begin{pmatrix} 3 \\ 1 \end{pmatrix} \\ \begin{pmatrix} 2 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 2 \\ 1 \end{pmatrix} \\ \begin{pmatrix} 2 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ \begin{pmatrix} 2 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ \begin{pmatrix} 2 \\ 0 \end{pmatrix} \end{array}$$

ergebnis = 0

ergebnis = 0

ergebnis = 0

ergebnis += 1 = 1

ergebnis = 1

ergebnis += 1 = 2

ergebnis += 1 = 3

ergebnis = 3

ergebnis = 3

ergebnis += 1 = 4

ergebnis += 1 = 5

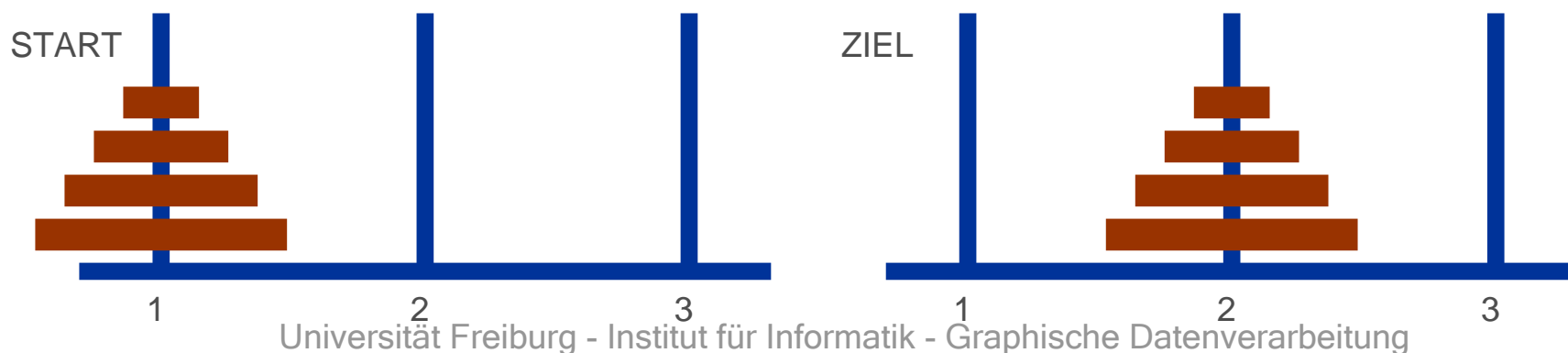
ergebnis += 1 = 6



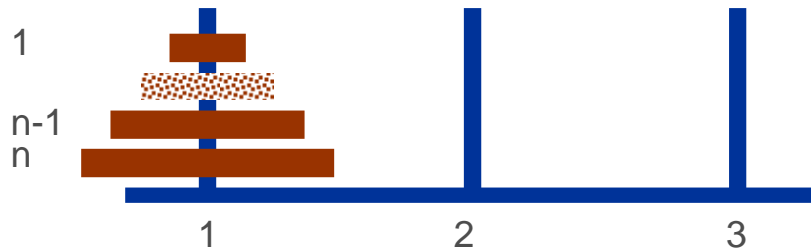
# Die Türme von Hanoi



- n Scheiben unterschiedlicher Größe liegen der Größe nach geordnet auf Pfahl 1, größte Scheibe unten
- Ziel: Transport der n Scheiben auf Pfahl 2 mit Hilfe von Pfahl 3
- Regeln: In jedem Schritt darf die jeweils oberste Scheibe von einem Pfahl a auf einen anderen Pfahl b transportiert werden, wenn die oberste Scheibe auf Pfahl b größer ist als die transportierte Scheibe

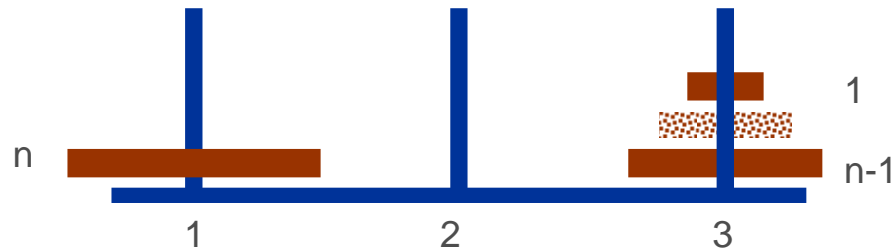


# Rekursive Formulierung



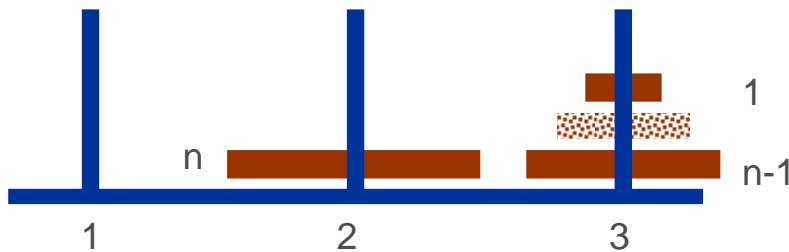
## Aufgabe:

Transportiere n Scheiben von 1 nach 2 mit 3

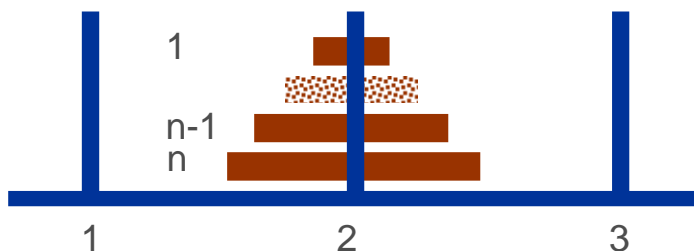


## Realisierung:

Transportiere n-1 Scheiben von 1 nach 3 mit 2



Transportiere Scheibe n von 1 nach 2



Transportiere n-1 Scheiben von 3 nach 2 mit 1

# Realisierung mit Stapel

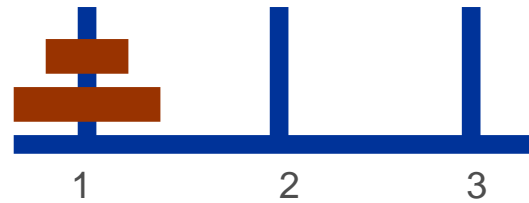


- Transportiere  $n-1$  Scheiben von 1 nach 3 mit 2 ( $n-1, 1, 3, 2$ )
  - Transportiere Scheibe  $n$  von 1 nach 2 ( $-n, 1, 2, 0$ )
  - Transportiere  $n-1$  Scheiben von 3 nach 2 mit 1 ( $n-1, 3, 2, 1$ )
- 
- ```
push (S,n,1,2,3);
while (empty(S)==false)
{
    (n,a,b,c) = pop (S);
    if (n<0) transportiereEineScheibe (n,a,b);
    if (n==0) ;
    if (n>0)
    {
        push(S,n-1,c,b,a);
        push(S,-n,a,b,0);
        push(S,n-1,a,c,b);
    }
}
```

Reihenfolge beachten

Beispiel

Aufgabe mit
2 Scheiben



- push (2,1,2,3);
- pop; n=2 → push (1,3,2,1); push (-2,1,2,0); push (1,1,3,2);
- pop; n=1 → push (0,2,3,1); push (-1,1,3,0); push (0,1,2,3);
- pop; n=0
- pop; n=-1 → Scheibe 1 von 1 nach 3
- pop; n=0
- pop; n=-2 → Scheibe 2 von 1 nach 2
- pop; n=1 → push (0,1,2,3); push (-1,3,2,0); push (0,3,1,2);
- pop; n=0
- pop; n=-1 → Scheibe 1 von 3 nach 2
- pop; n=0
- Stack empty

Zusammenfassung



- Datenstrukturen werden zur Realisierung (Repräsentation) dynamischer Mengen verwendet.
- Datenstrukturen sind unterschiedlich effizient in Bezug auf Manipulationen.
- Beispiele:
 - Feld
 - Verkettete Liste
 - Stapel und Schlangen mit speziellen Einfüge- und Entferne-Operationen (Stapel nach LIFO-Prinzip, Schlange nach FIFO-Prinzip)
- Anwendungen
 - topologische Sortierung, Klammersausdrücke, iterative Realisierung rekursiver Funktionen

Nächstes Thema



- Algorithmen
 - Sortierverfahren