Kapitel 3.8: Laufzeit von DFS und BFS

Algorithmen und Datenstrukturen
WS 2021/22

Prof. Dr. Sándor Fekete

Satz 3.13

Der Graphen-Scan-Algorithmus 3.7 lässt sich so implementieren, dass die Laufzeit $O(n+m)$ ist.

Algorithmus 3.7

INPUT: Graph $G = (V,E)$, Knoten s

OUTPUT: Knotenmenge $Y \subseteq V$, die von s aus erreichbar ist,

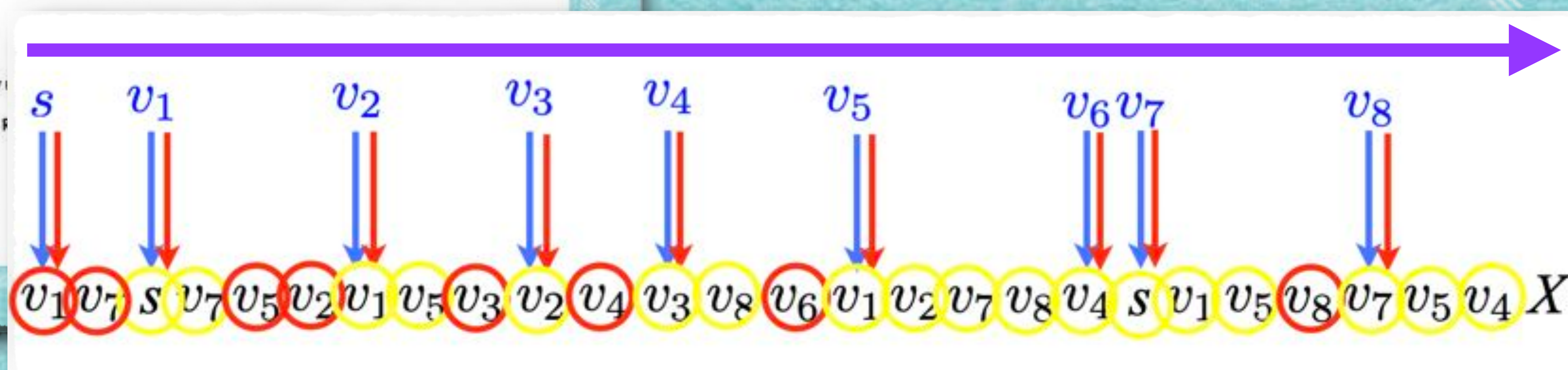
Kantenmenge $T \subseteq E$, die die Erreichbarkeit sicherstellt

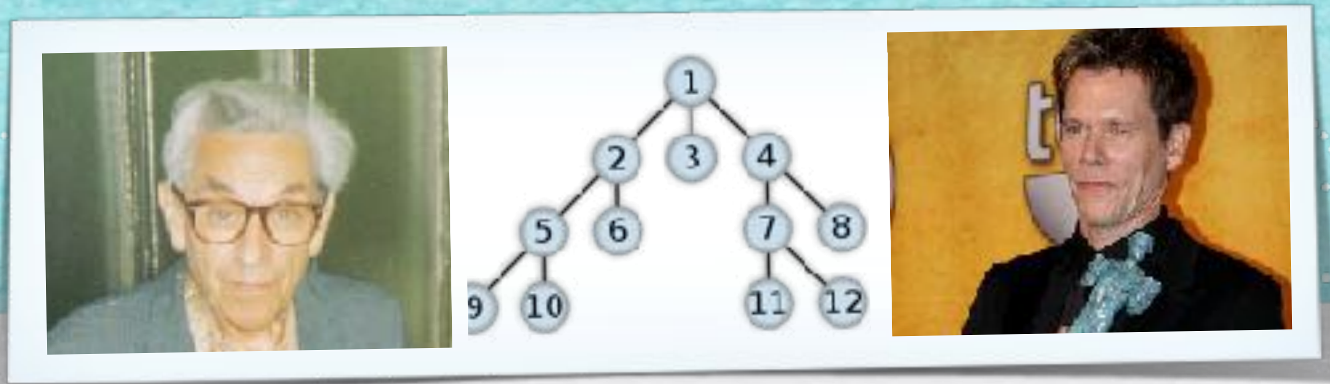
1. Sei $R:=\{s\}$, $Y:=\{s\}$, $T:=\emptyset$
2. WHILE $(R\neq\emptyset)$ DO {
   - 2.1. Wähle $v \in R$
   - 2.2. IF (es gibt kein $w \in V \setminus Y$ mit $e=\{v,w\} \in E$) THEN
     - 2.2.1. $R:=R\setminus\{v\}$
   - 2.3. ELSE {
     - 2.3.1. Wähle ein w
     - 2.3.2. Setze R := R

     }

   }
3. STOP

Adjazenzliste!





Kapitel 3.9: Eigenschaften von DFS und BFS

Algorithmen und Datenstrukturen
WS 2021/22

Prof. Dr. Sándor Fekete

Algorithmus 3.17

INPUT: Graph $G = (V,E)$, Knoten s

OUTPUT: Knotenmenge $Y \subseteq V$, die von s aus erreichbar ist,

für jeden Knoten $v \in Y$ die Länge $l(v)$ eines kürzesten s-v-Weges,

Kantenmenge $T \subseteq E$, die die Erreichbarkeit sicherstellt

1. Sei $R:=\{s\}$, $Y:=\{s\}$, $T:=\emptyset$, $l(s):=0$

2. WHILE ($R\neq\emptyset$) DO {

    2.1. wähle Element $v \in R$

    2.2. IF (es gibt kein w $V\setminus Y$ mit $e=\{v,w\} \in E$) THEN

        2.2.1. $R:=R\setminus\{v\}$

    2.3. ELSE {

        2.3.1. wähle ein $w \in V\setminus R$ mit $e=\{v,w\} \in E$;

        2.3.2. setze $R := R$ u $\{w\}$, $Y := Y$ u $\{w\}$, $T := T$ u $\{e\}$;

        2.3.3. setze $l(w):=l(v)+1$

        }

}

Satz 3.18

(1) Das Verfahren 3.17 ist endlich.

(2) Die Laufzeit ist O(n+m).

(3) Am Ende ist für jeden erreichbaren Knoten v ∈ Y die Länge eines kürzesten Weges von s nach v im Baum (Y,T) durch l(v) gegeben.

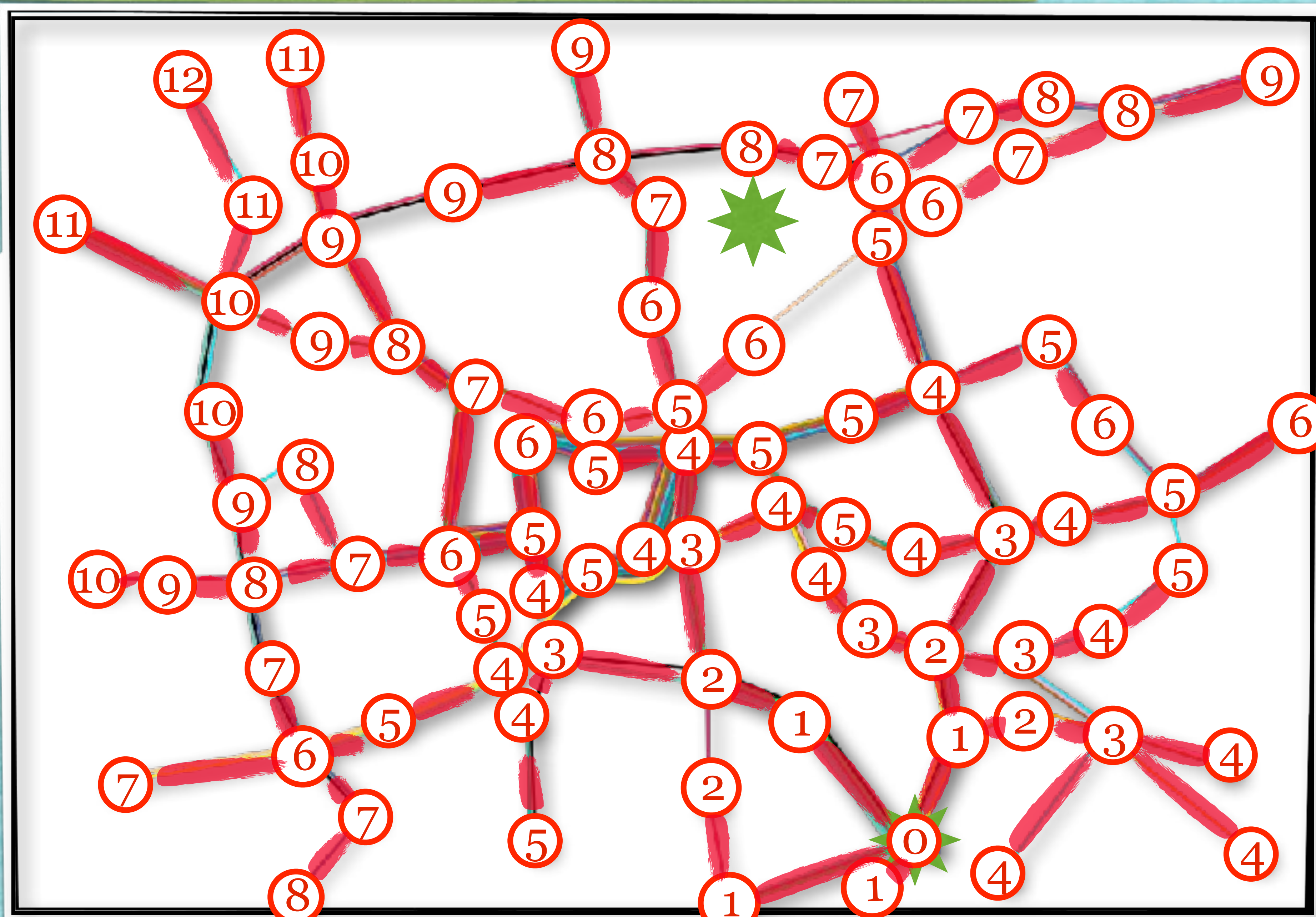
(4) Am Ende ist für jeden erreichbaren Knoten v ∈ Y die Länge eines kürzesten Weges von s nach v im Graphen (V,E) durch l(v) gegeben.

Beweis:

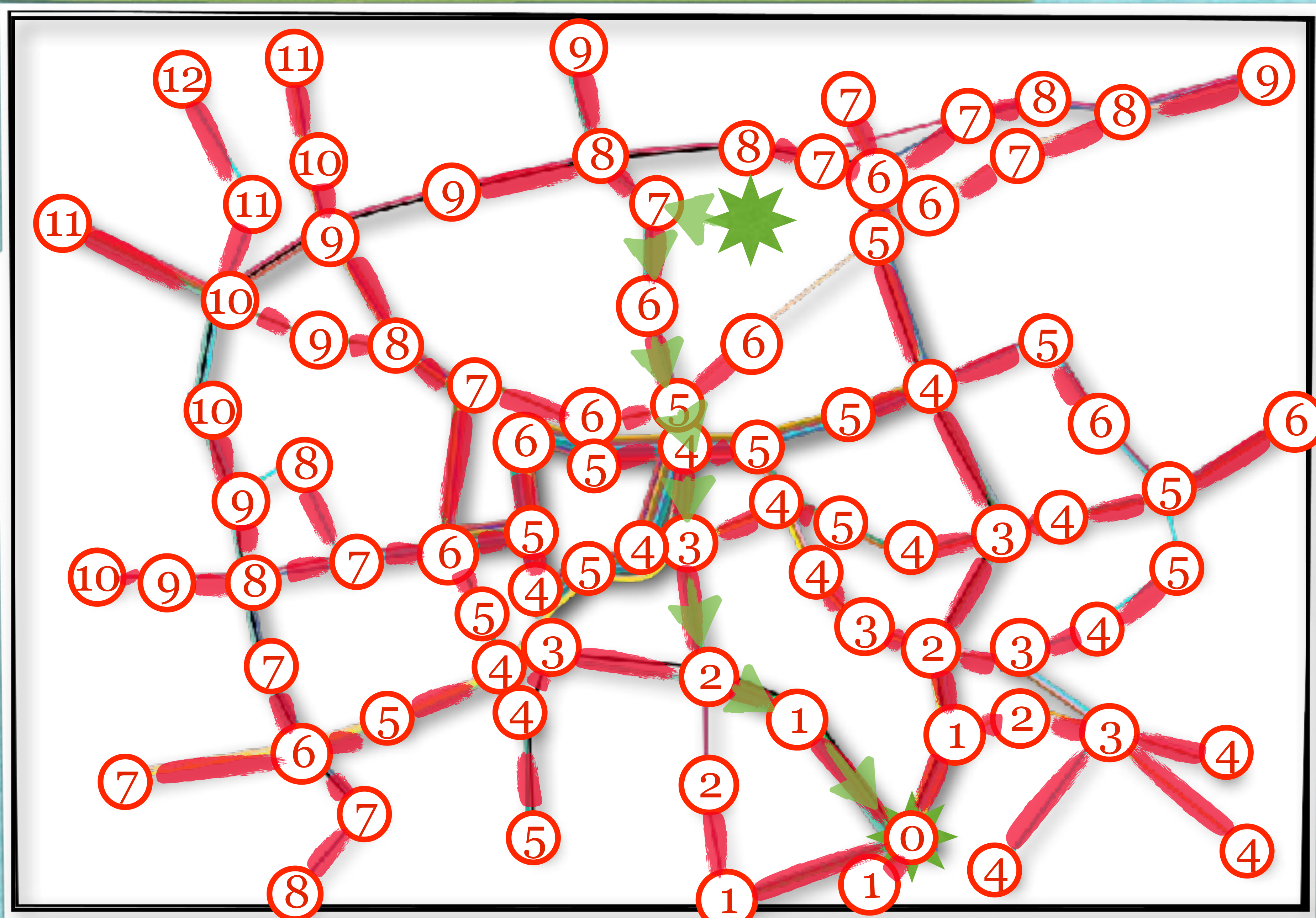
(1) Wie für Algorithmus 3.7 gelten alle Eigenschaften. zusätzlich ist für jeden Knoten v ∈ Y per Induktion, der Wert l(v) tatsächlich definiert.

(2) Die Laufzeit bleibt von Algorithmus 3.7 erhalten.

Wellenreiten in Graphen



Wellenreiten in Graphen



Breitensuche

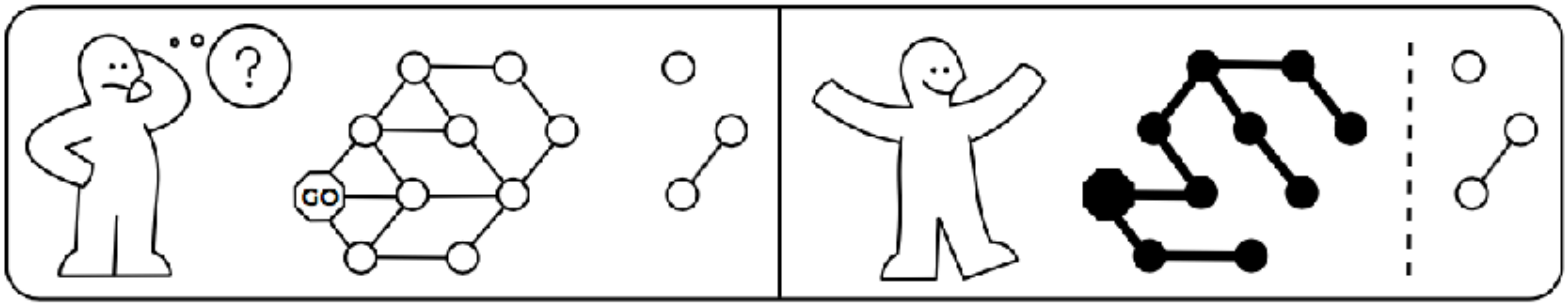
Mehr Details!

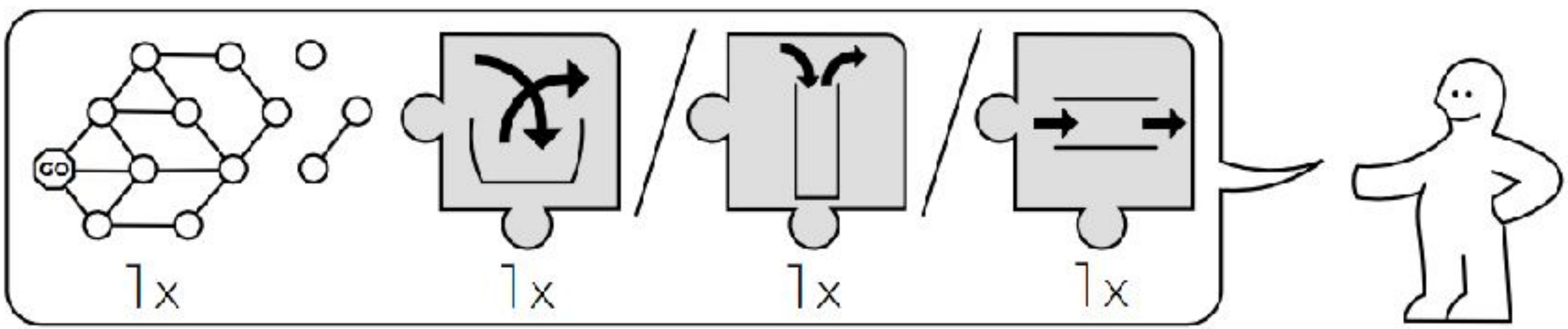
s.fekete@tu-bs.de

GRÅPH SCÄN

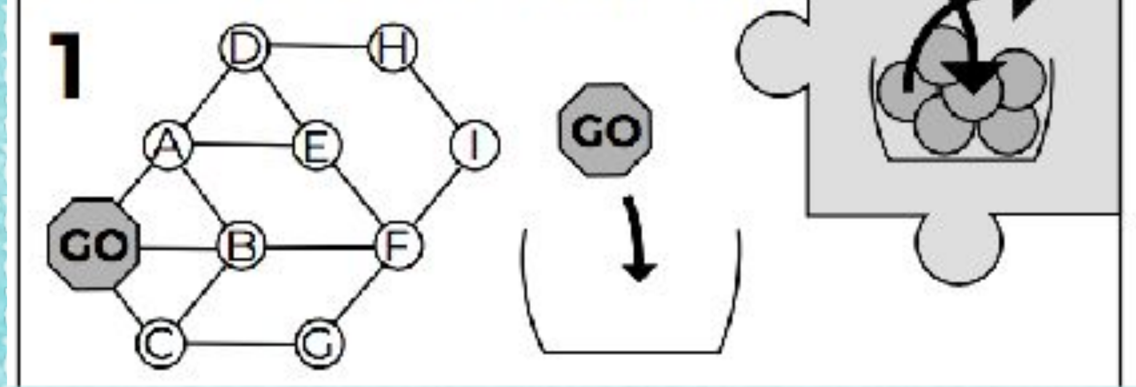
idea-instructions.com/graph-scan/

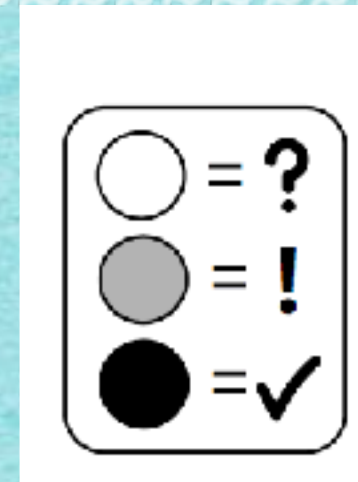
IDEA

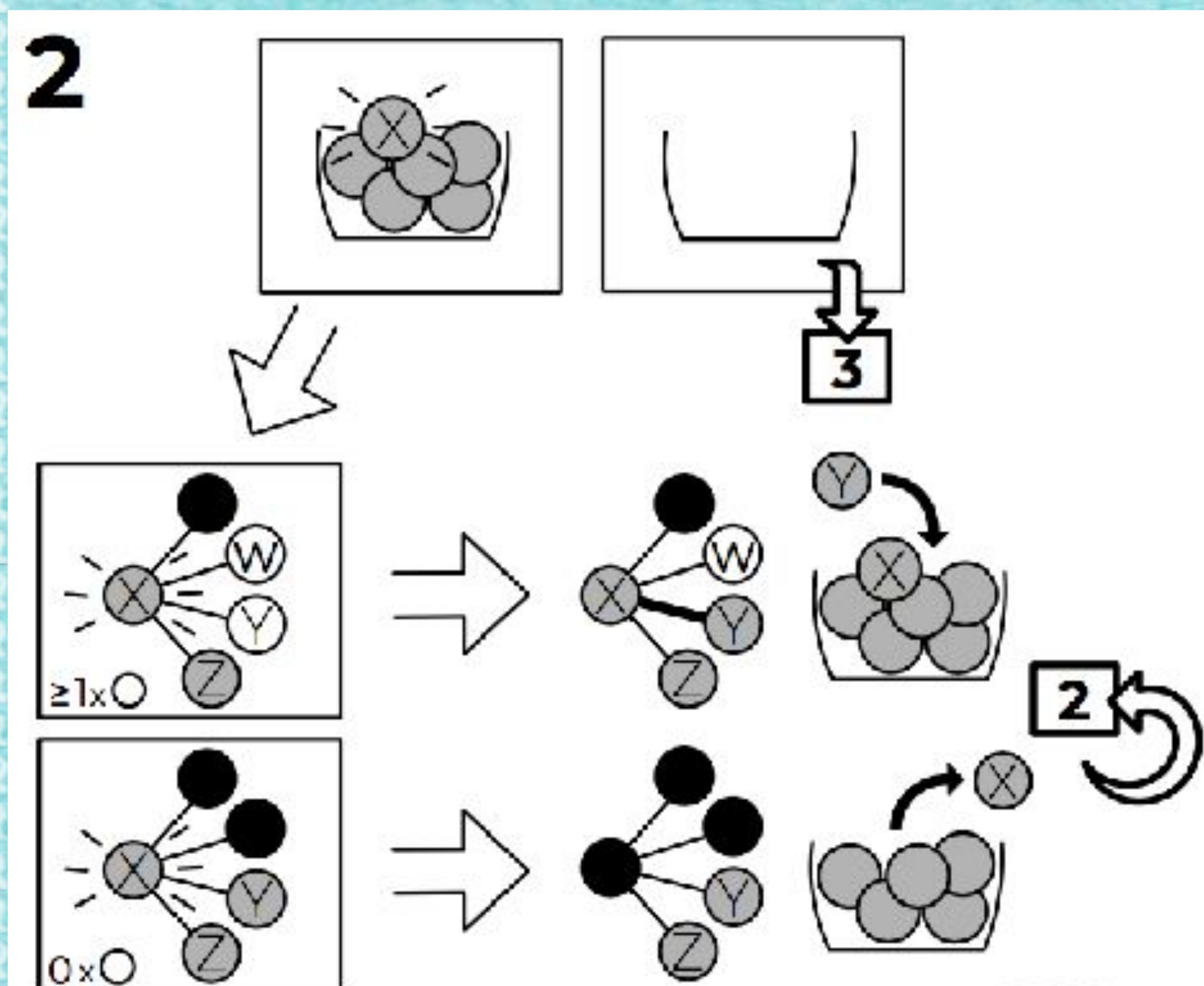


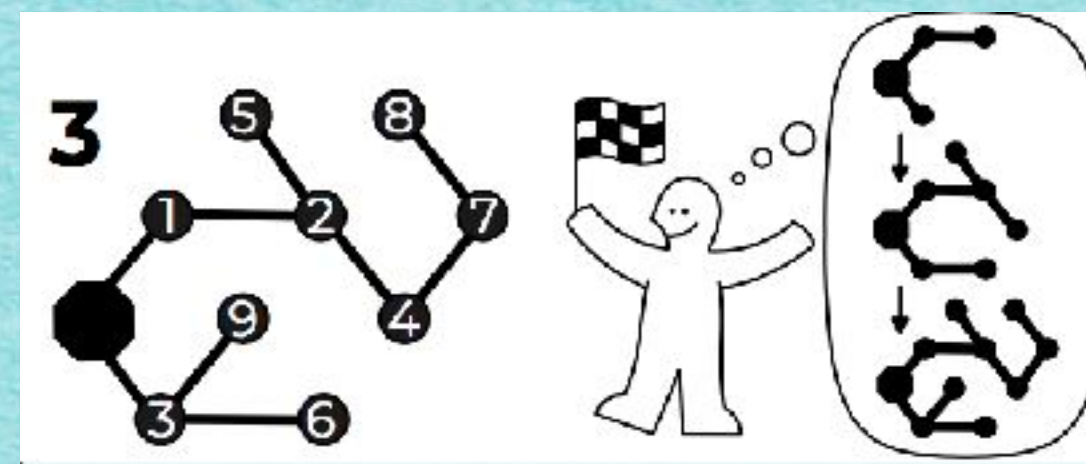


AD-HOC SEARCH



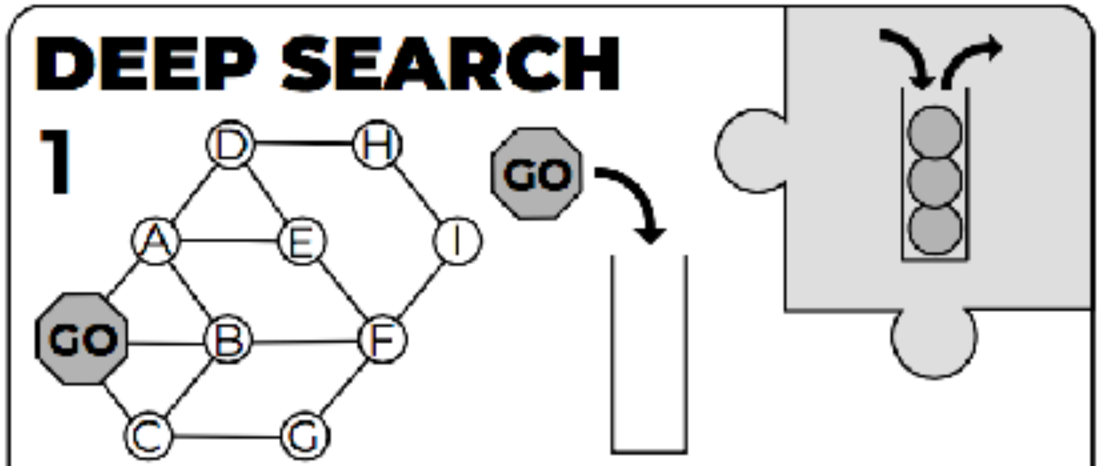


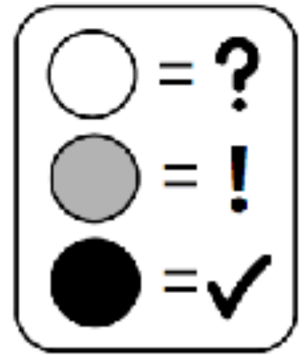




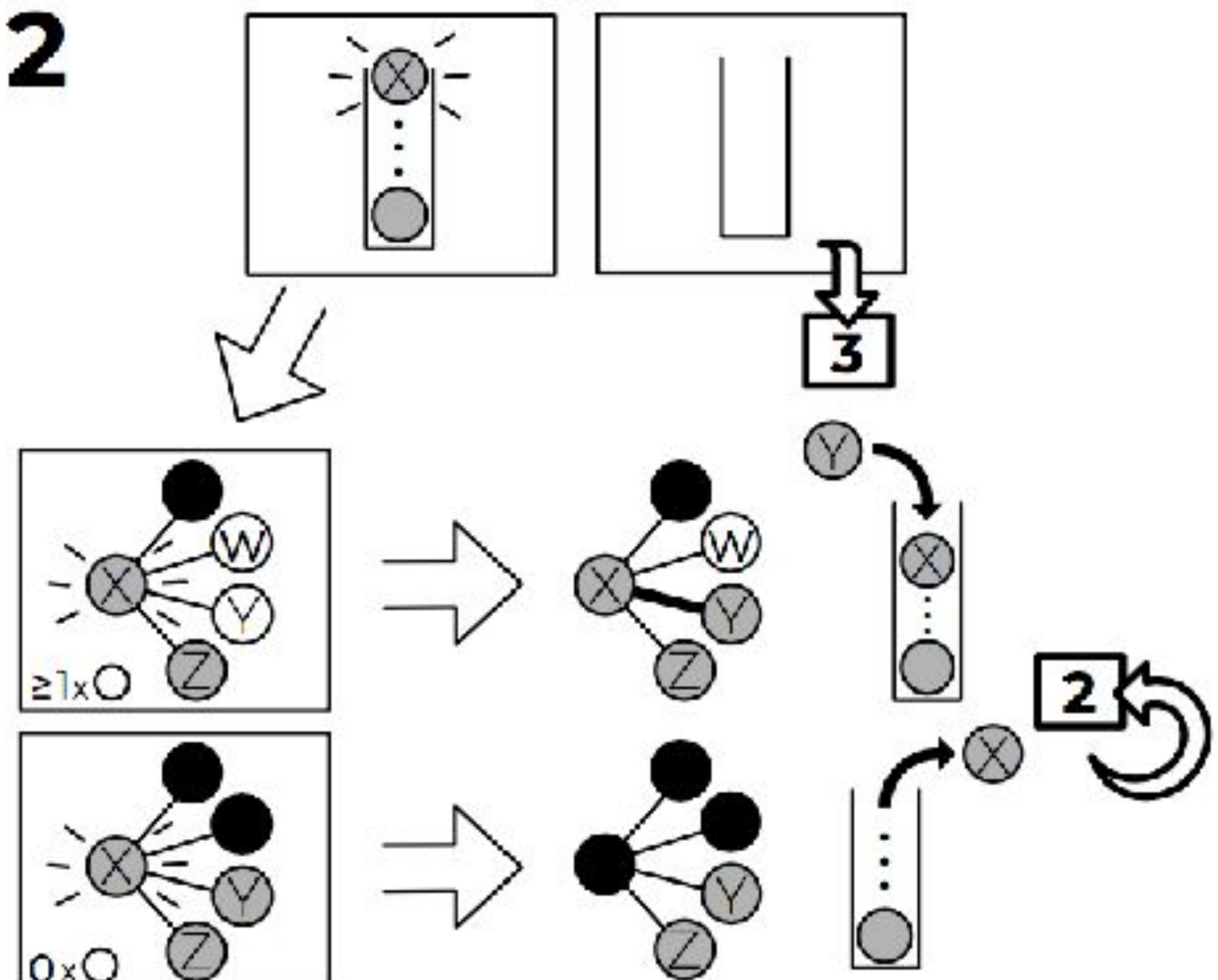
DEEP SEARCH

1

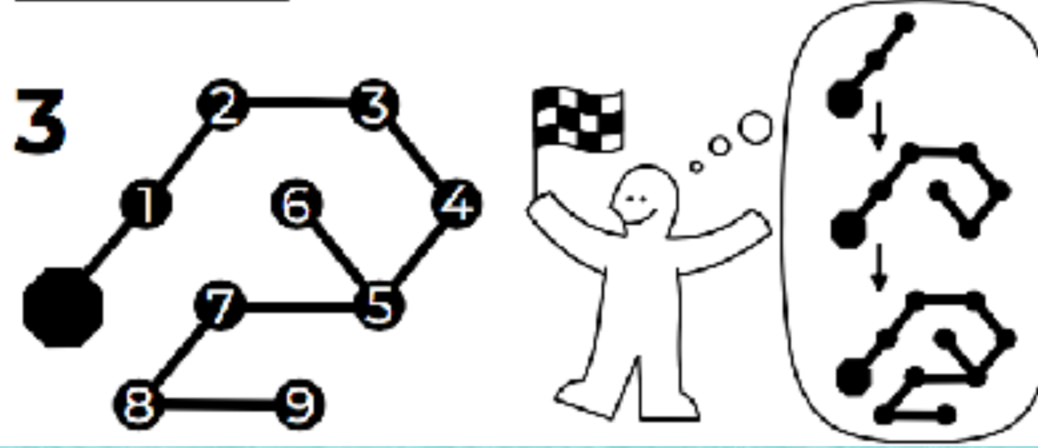




2

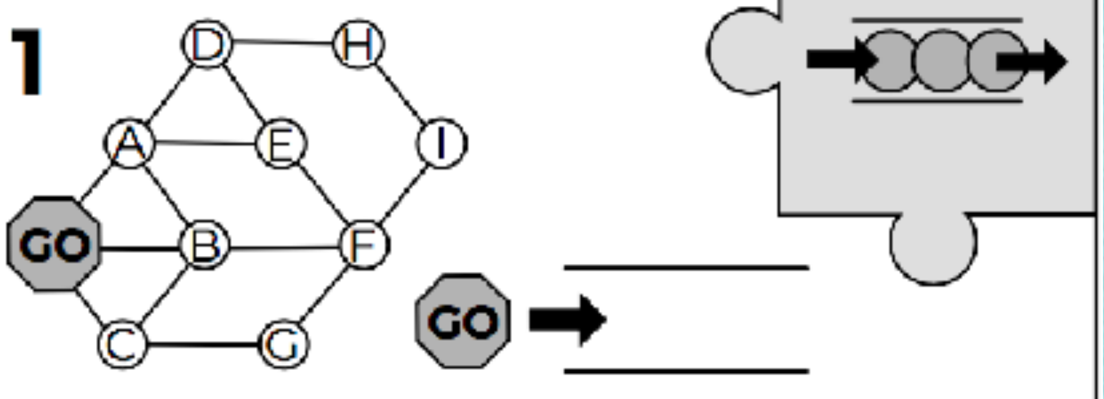


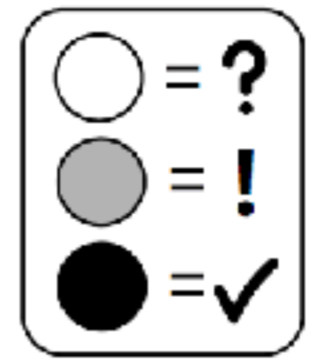
3



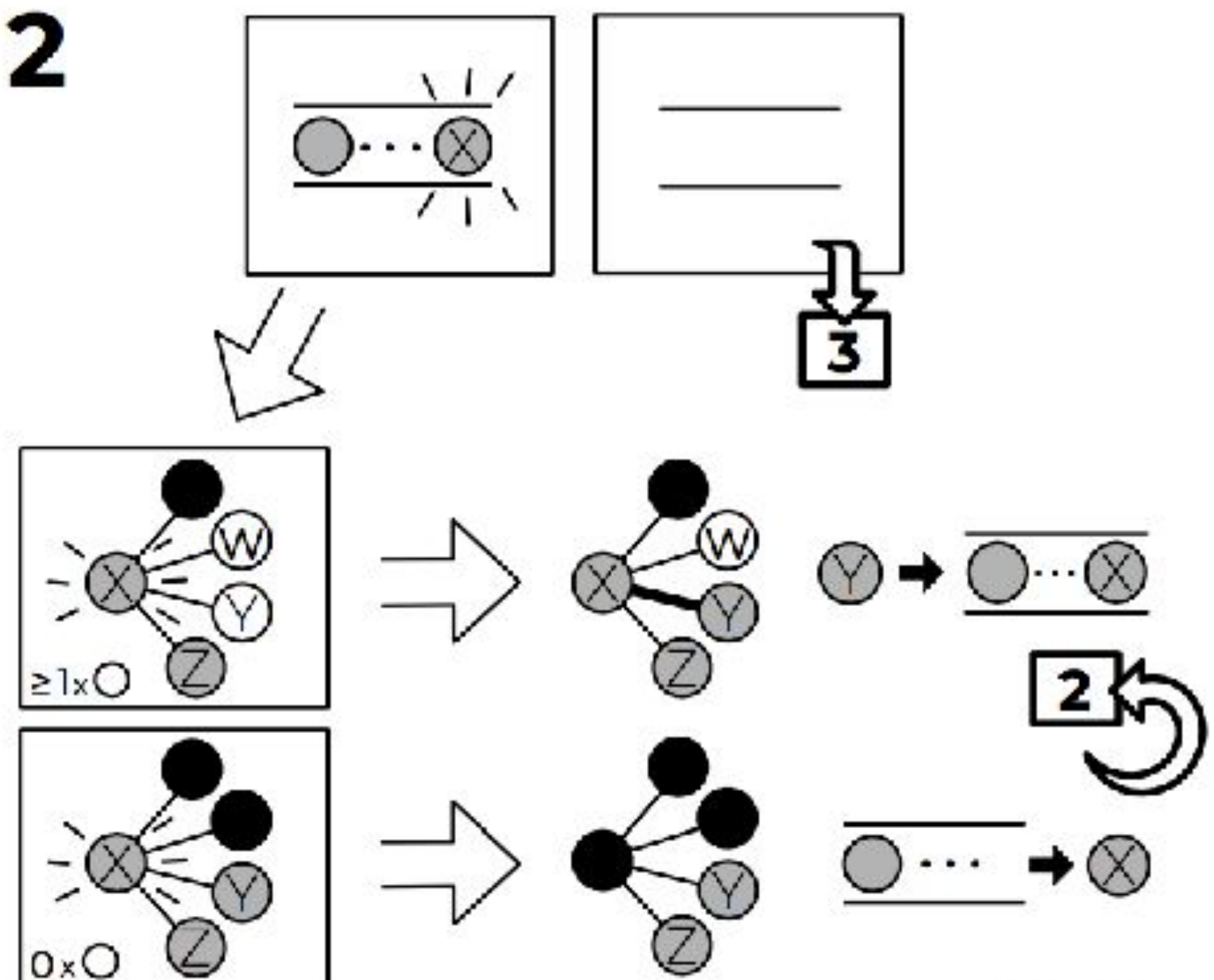
BROAD SEARCH

1

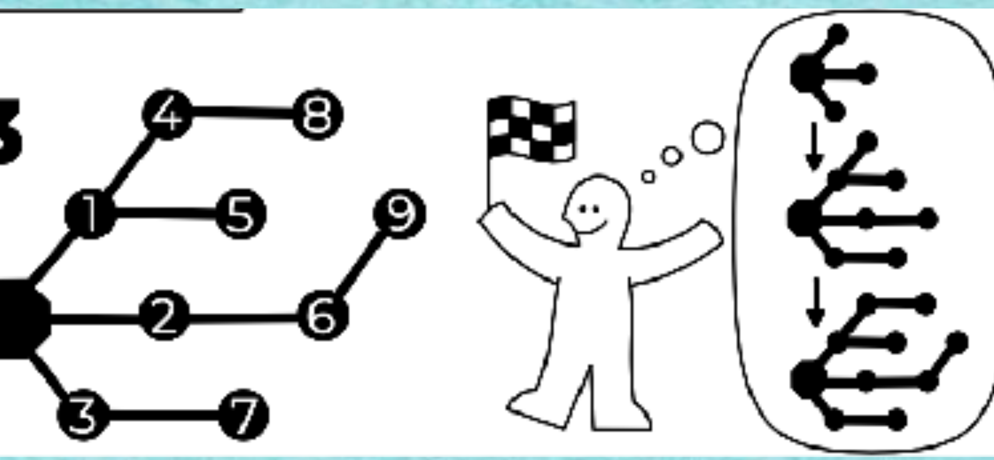




2



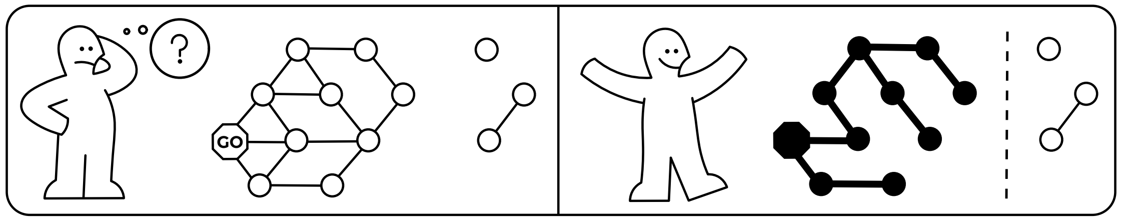
3

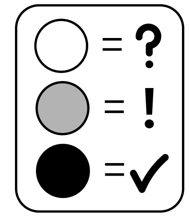


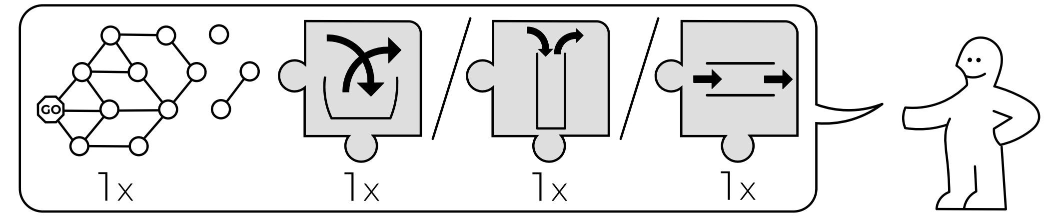
GRÅPH SKÄN

idea-instructions.com/graph-scan/
v1.3, CC by-nc-sa 4.0

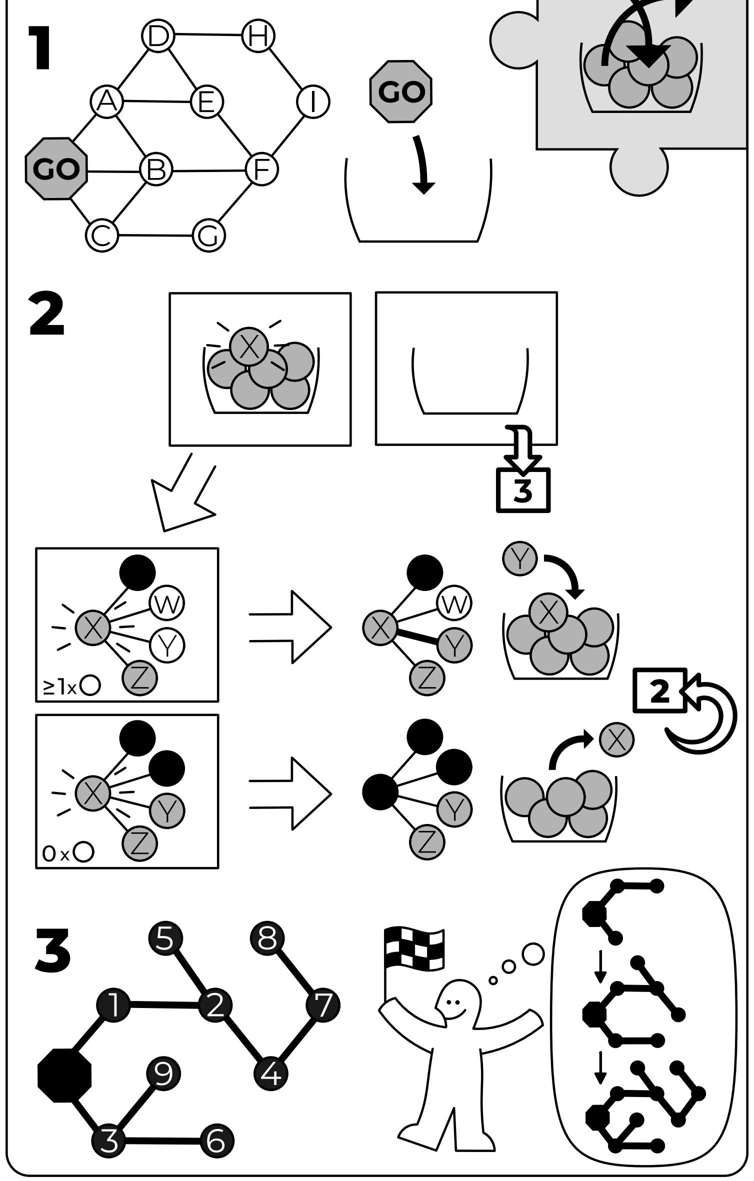
IDEA



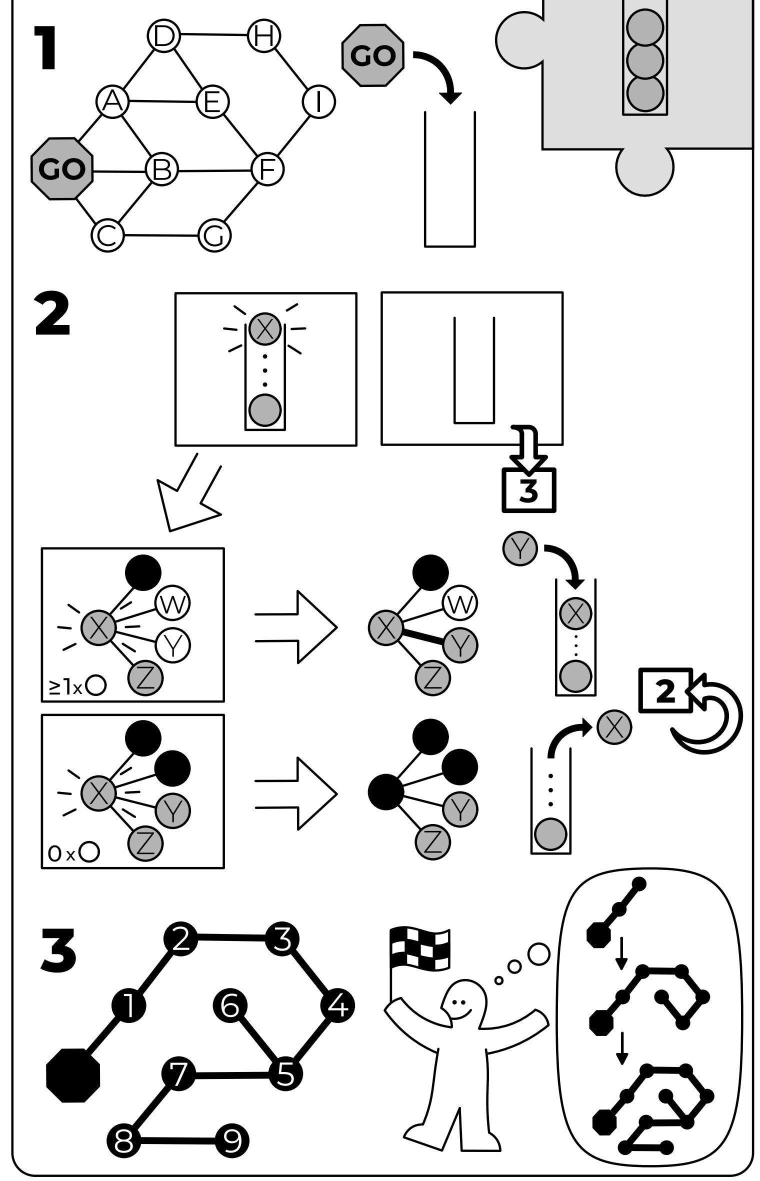




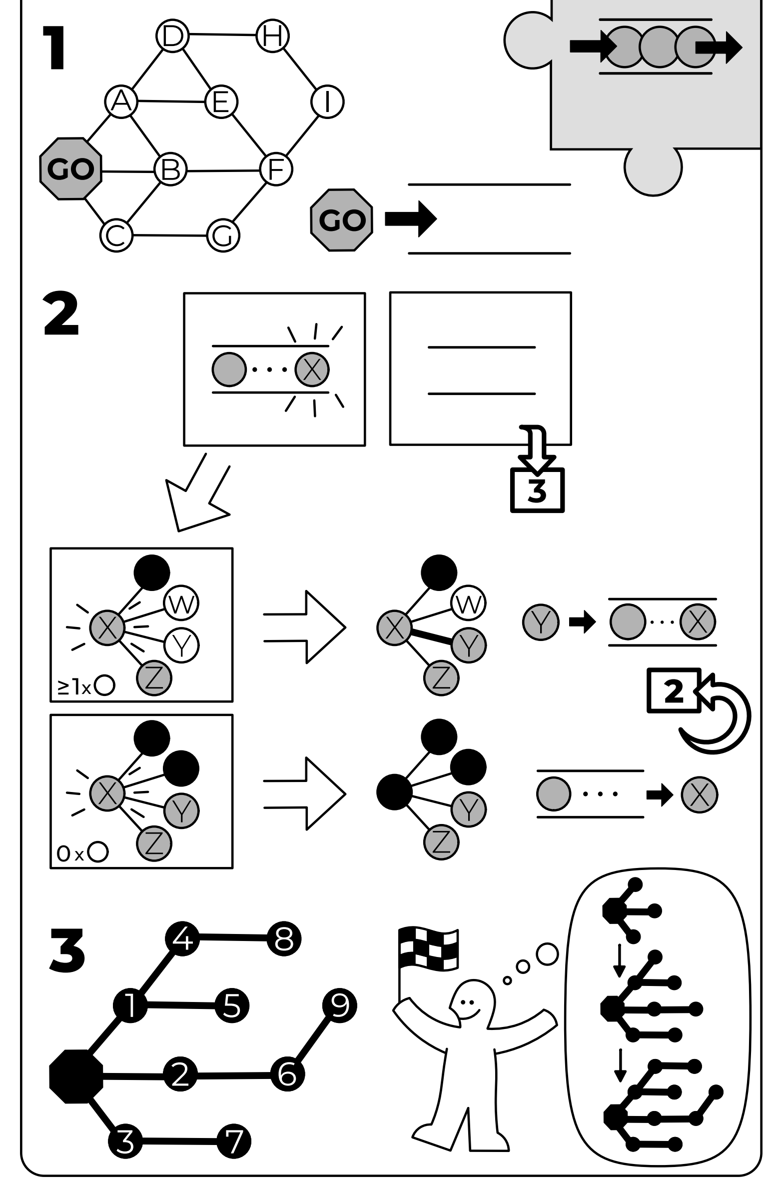
AD-HOC SEARCH



DEEP SEARCH

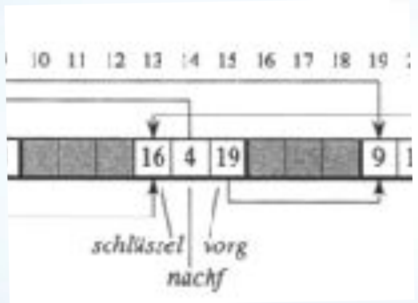


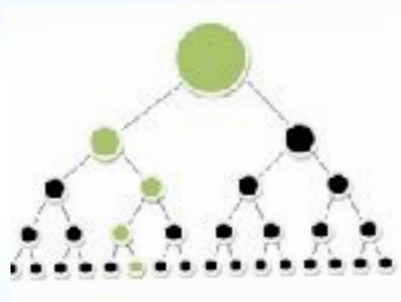
BROAD SEARCH



Kapitelende!

s.fekete@tu-bs.de






Kapitel 4: Dynamische Datenstrukturen

Algorithmen und Datenstrukturen
WS 2021/22

Prof. Dr. Sándor Fekete

Wie verwalten wir dynamische Mengen von Objekten?

Waschkorb

4.1 Grundoperationen

Aufgabenstellung:

- ***Verwalten einer Menge S von Objekten***
- ***Ausführen von verschiedenen Operationen (s.u.)***

Im Folgenden:

S	**Menge von Objekten**
k	**Wert eines Elements ("Schlüssel")**
x	**Zeiger auf Element**
NIL	**spezieller, "leerer" Zeiger**

<u>SEARCH(S,k):</u> **"Suche in S nach k"**

Durchsuche die Menge S nach einem Element von Wert k.

Ausgabe: Zeiger x, falls x existent
NIL, falls kein Element Wert k hat.

<u>INSERT(S,x):</u> **“Füge x in S ein”**

Erweitere S um das Element, das unter der Adresse x steht.

<u>DELETE(S,x):</u> **"Entferne x aus S"**

Lösche das unter der Adresse x stehende Element aus der Menge S.

<u>MINIMUM(S):</u> **“Suche das Minimum in S”**

Finde in S ein Element von kleinstem Wert. (Annahme: Die Werte lassen sich vollständig vergleichen!)

Ausgabe: Zeiger x auf solch ein Element

MAXIMUM(S): **“Suche das Maximum in S”**

Finde in S ein Element von größtem Wert. (Annahme: Die Werte lassen sich vollständig vergleichen!)

Ausgabe: Zeiger x auf solch ein Element

PREDECESSOR(S,x):
“Finde das nächstkleinere Element”

Für ein in x stehendes Element in S, bestimme ein Element von nächstkleinerem Wert in S.

Ausgabe: Zeiger y auf Element
NIL, falls x Minimum von S angibt

<u>SUCCESSOR(S,x):</u>
“Finde das nächstgrößere Element”

Für ein in x stehendes Element in S, bestimme ein Element von nächstgrößerem Wert in S.

Ausgabe: Zeiger y auf Element
NIL, falls x Maximum von S angibt

4.1 Grundoperationen

Wie nimmt man das vor?

**Wie lange dauert das,
in Abhängigkeit von der Größe von S?**

Unsortierte Unterlagen:
Immer alles durchgehen, also: O(n)

Sortierte Unterlagen: Geht schneller!

4.1 Grundoperationen

Langsam:

- ***O(n):*** ***lineare Zeit***

  Alle Objekte anschauen

Sehr schnell:

- ***O(1):*** ***konstante Zeit***

  Immer gleich schnell, egal wie groß S ist.

Schnell:

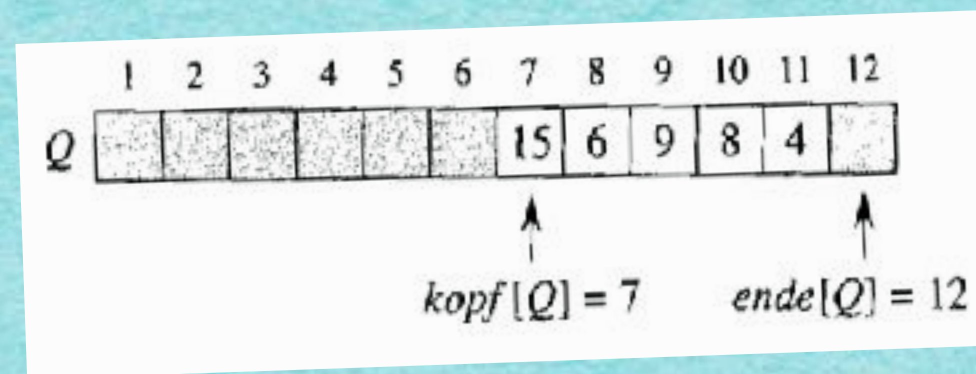
- ***O(log n):*** ***logarithmische Zeit***

  Wiederholtes Halbieren

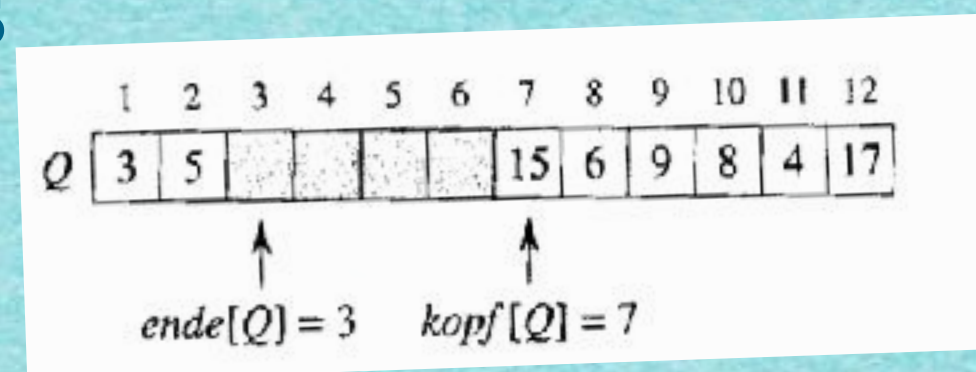
4.2 Stapel und Warteschlange

OUT

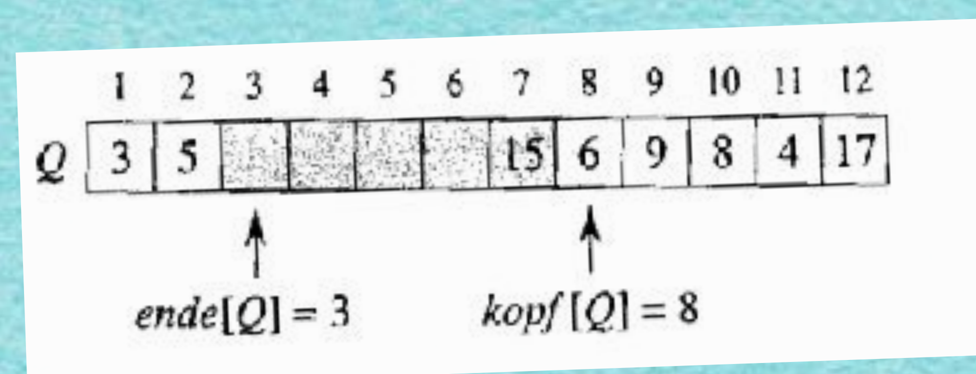
IN



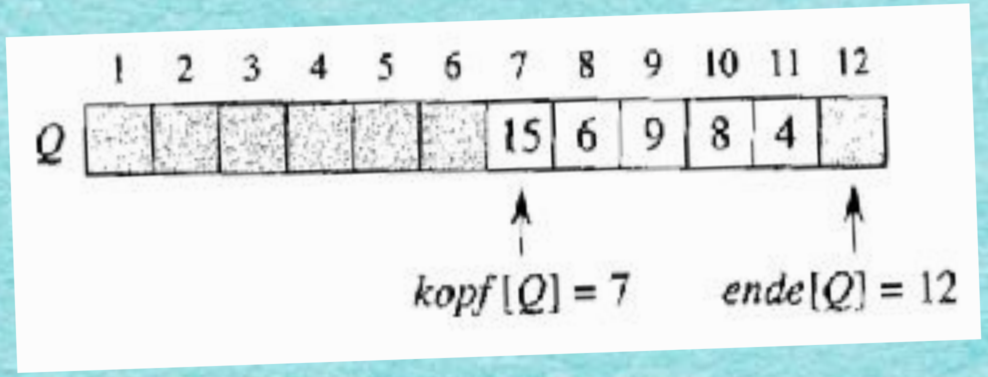
Enqueue: 17, 3, 5

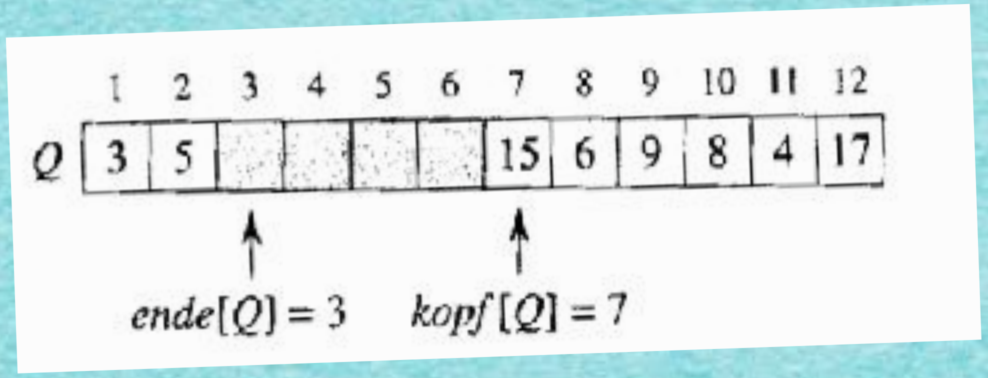


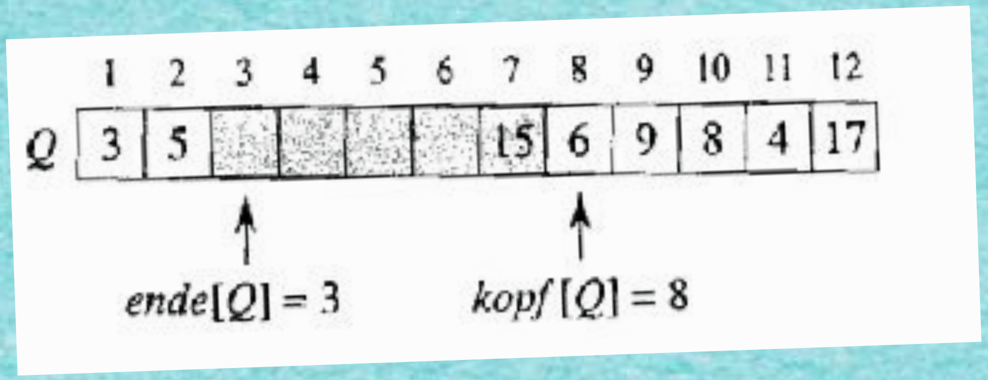
Dequeue:



Warteschlange auf Array umgesetzt







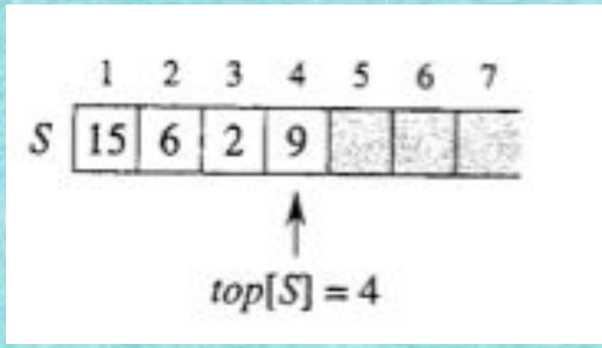
ENQUEUE(Q, x)

```
Q[ende[Q]] ← x
if ende[Q] = länge[Q]
    then ende[Q] ← 1
    else  ende[Q] ← ende[Q] + 1
```

DEQUEUE(Q)

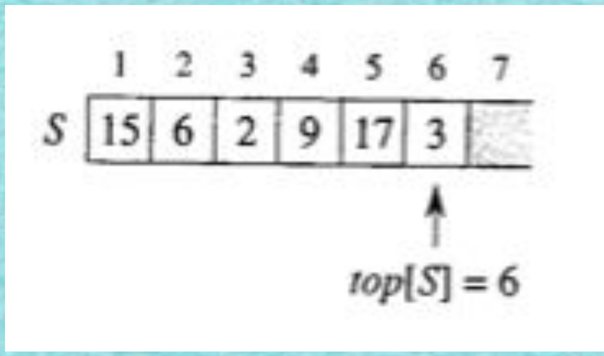
```
x ← Q[kopf[Q]]
if kopf[Q] = länge[Q]
    then kopf[Q] ← 1
    else  kopf[Q] ←  kopf[Q] + 1
return x
```

Stack auf Array umgesetzt



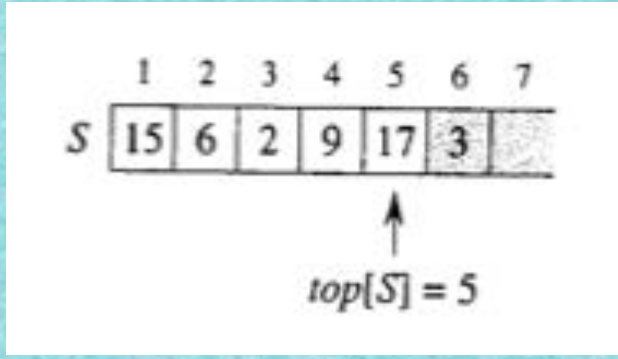
Push: 17, 3

IN

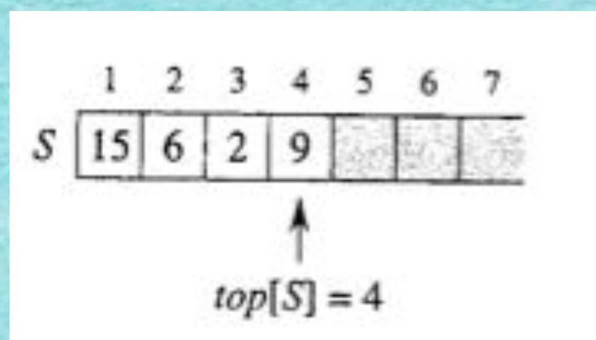


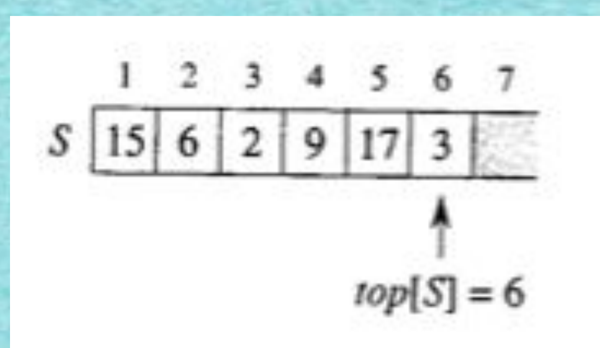
Pop

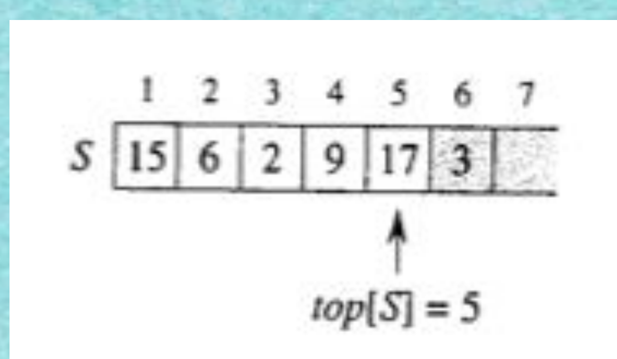
OUT



Stack auf Array umgesetzt



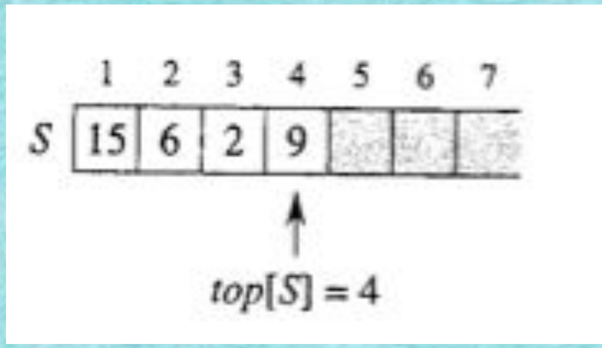




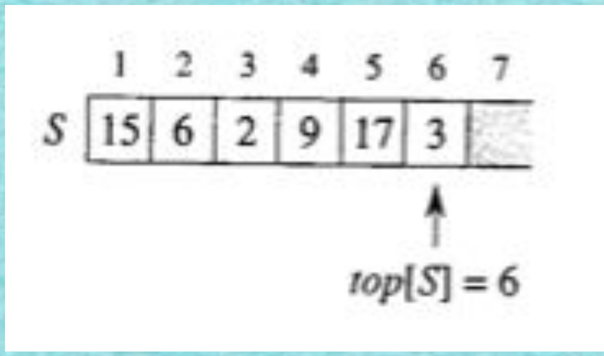
IN

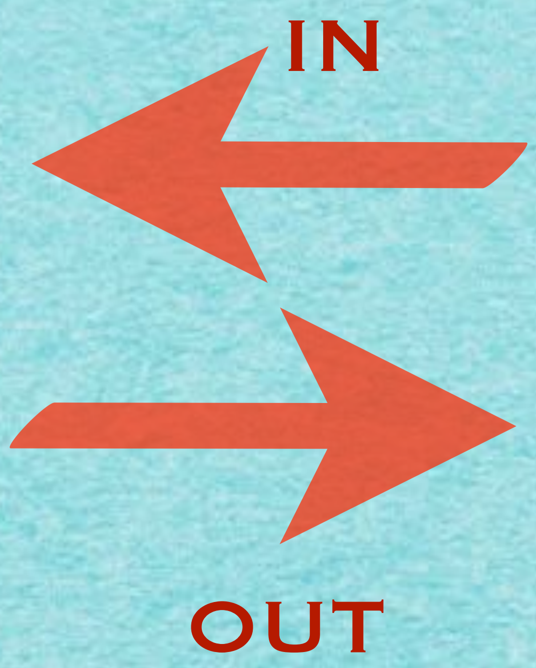
OUT

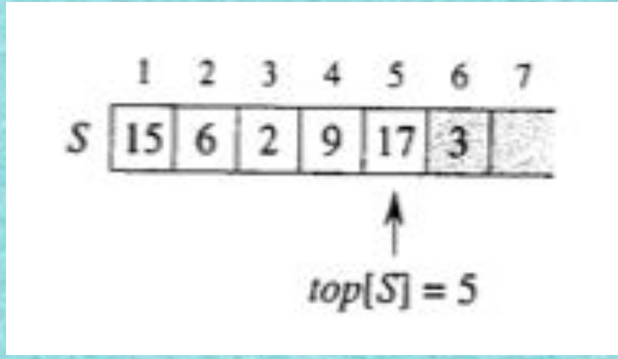
Stack auf Array umgesetzt



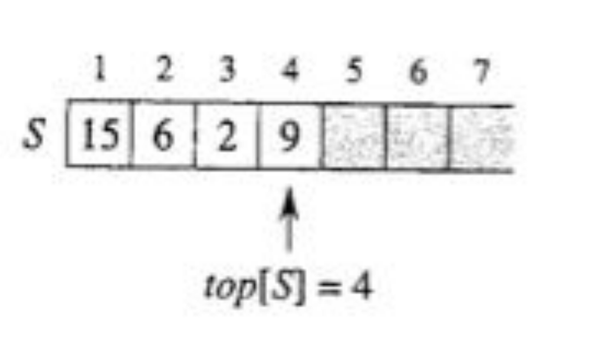

```
STACK-EMPTY(S)
if top[S] = 0
    then return WAHR
    else return FALSCH
```



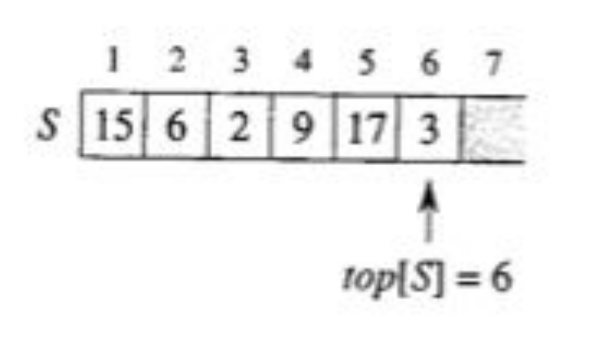




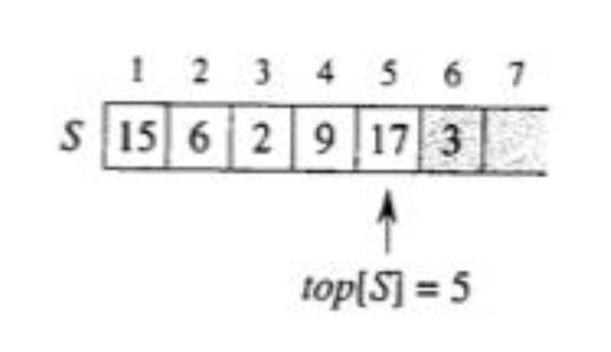
Stack auf Array umgesetzt



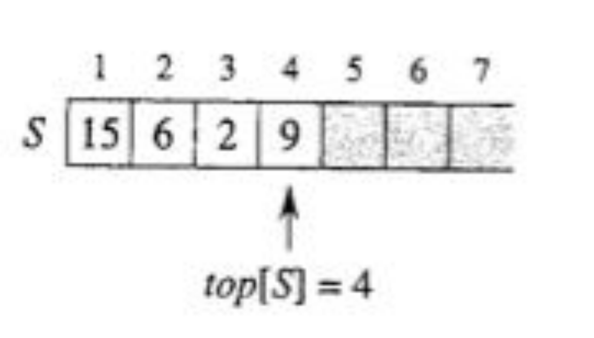

```
STACK-EMPTY(S)
if top[S] = 0
    then return WAHR
    else return FALSCH
```



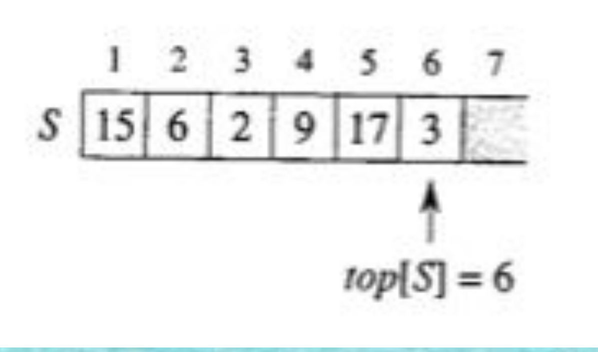

```
PUSH(S, x)
top[S] ← top[S] + 1
S[top[S]] ← x
```



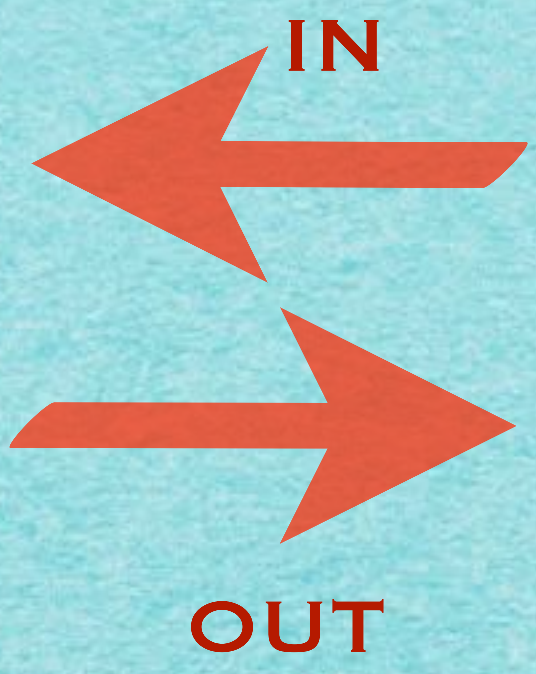
Stack auf Array umgesetzt

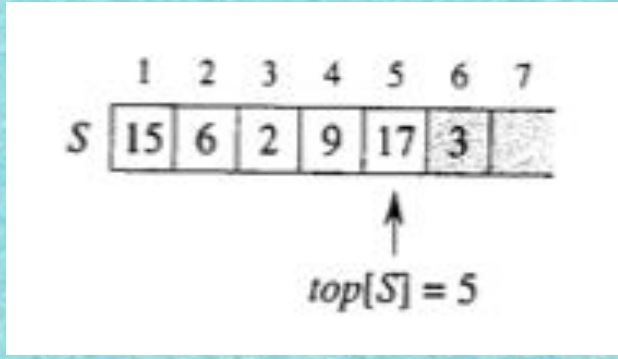



```
STACK-EMPTY(S)
if top[S] = 0
    then return WAHR
    else return FALSCH
```




```
PUSH(S, x)
top[S] ← top[S] + 1
S[top[S]] ← x
```





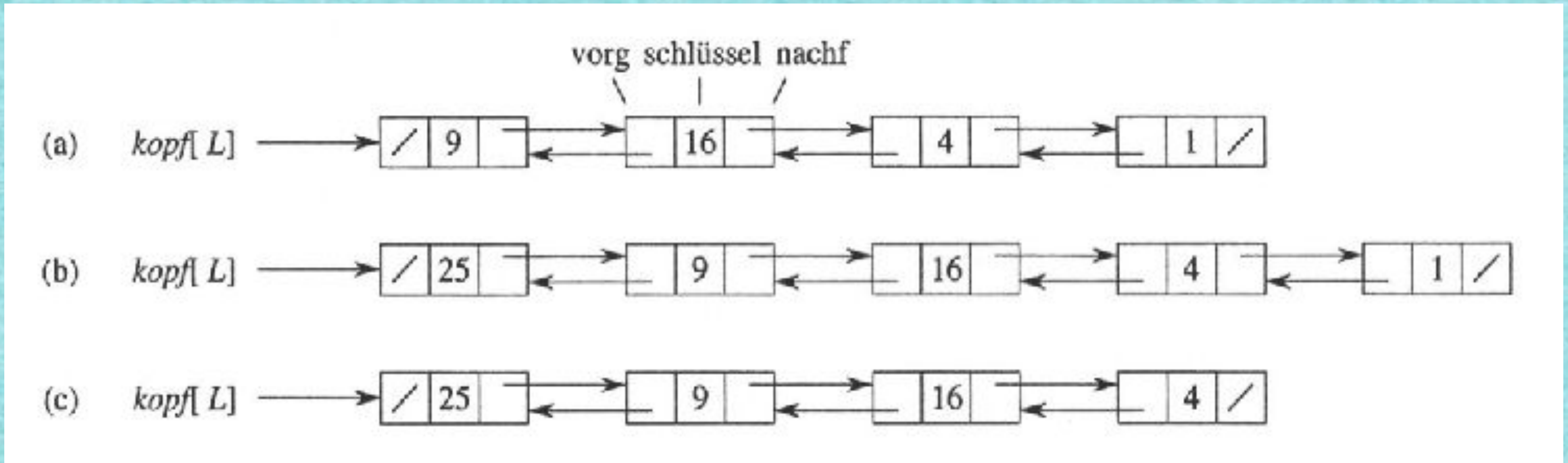

```
POP(S)
if STACK-EMPTY(S)
    then error "Unterlauf"
    else top[S] ← top[S] − 1
         return S[top[S] + 1]
```

4.3 Verkettete Listen

Idee:

Ordne Objekte nicht explizit in aufeinanderfolgenden Speicherzellen an, sondern gib jeweils Vorgänger und Nachfolger an.

Struktur einer doppelt verketteten Liste

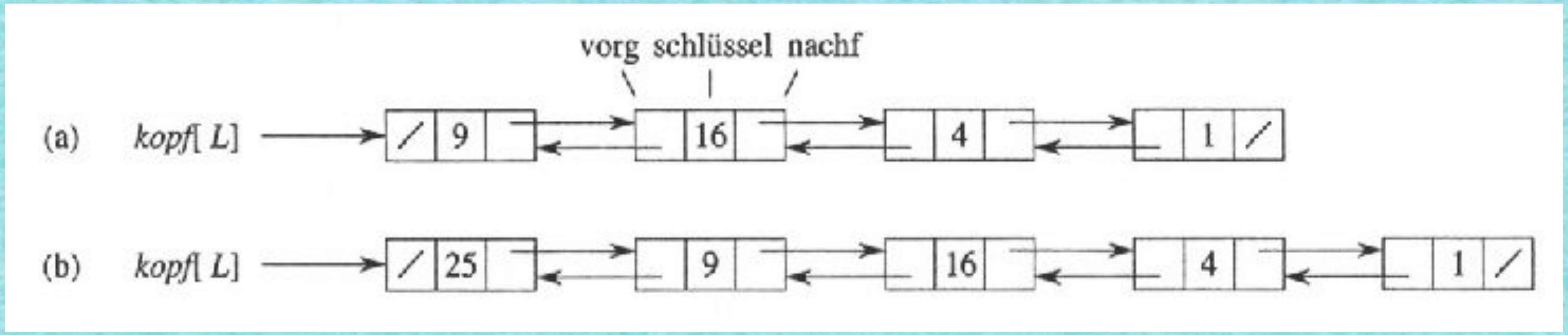


- **Füge vorne das Element mit Schlüssel 25 ein.**

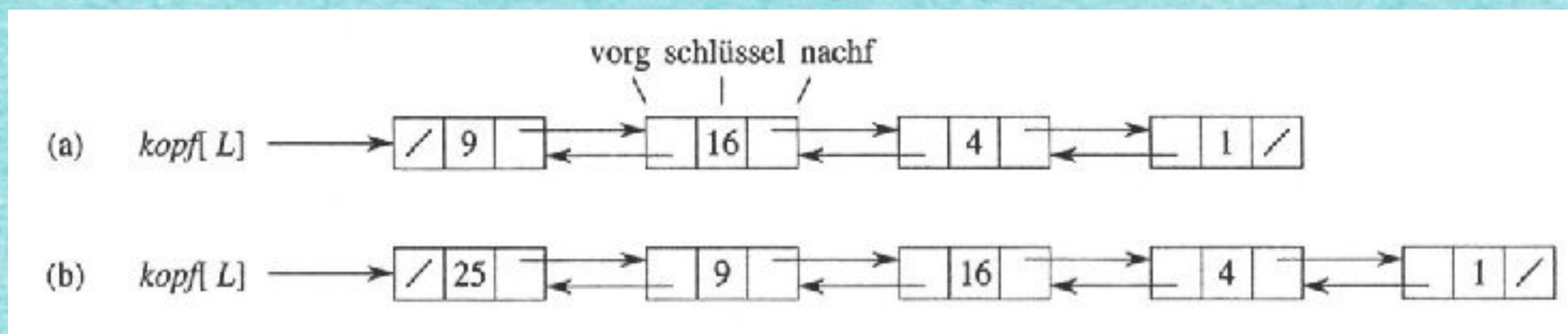
- **Finde ein Element mit Schlüssel 1 und lösche es.**

Einfügen in eine doppelt verkettete Liste

Einfügen in eine doppelt verkettete Liste

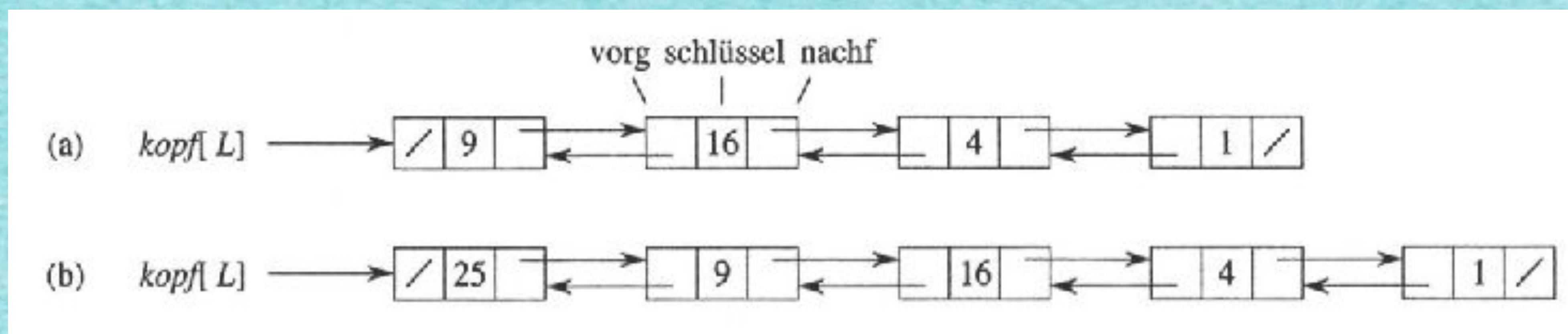


Einfügen in eine doppelt verkettete Liste




```
LIST-INSERT(L, x)
  nachf[x] ← kopf[L]
  if kopf[L] ≠ NIL
      then vorg[kopf[L]] ← x
  kopf[L] ← x
  vorg[x] ← NIL
```

Einfügen in eine doppelt verkettete Liste

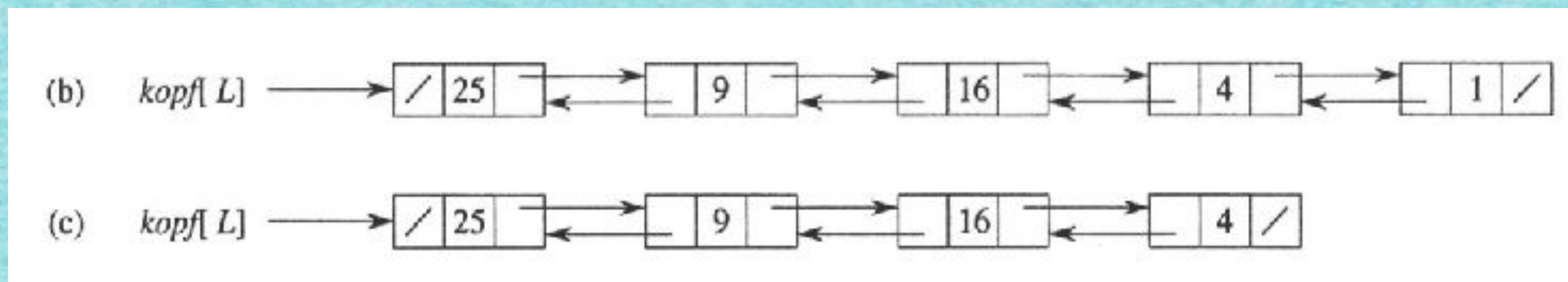


$\textsc{List-Insert}(L, x)$

$nachf[x] \leftarrow kopf[L]$
if $kopf[L] \neq \text{NIL}$
  then $vorg[kopf[L]] \leftarrow x$
$kopf[L] \leftarrow x$
$vorg[x] \leftarrow \text{NIL}$

<u>Laufzeit:</u> O(1)

Löschen aus einer doppelt verketteten Liste



<u>Laufzeit:</u> O(n)

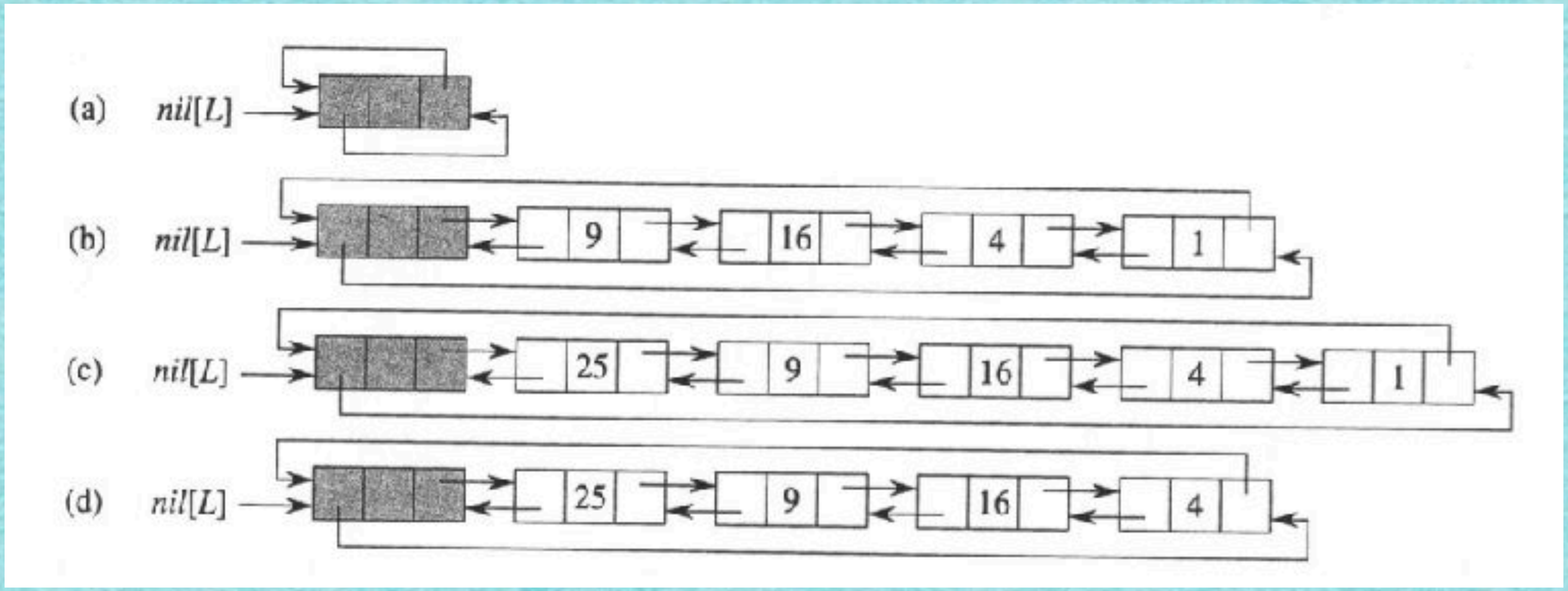
```
LIST-SEARCH(L, k)
x ← kopf[L]
while x ≠ NIL und schlüssel[x] ≠ k
        do x ← nachf[x]
return x
```

<u>Laufzeit:</u> O(1)

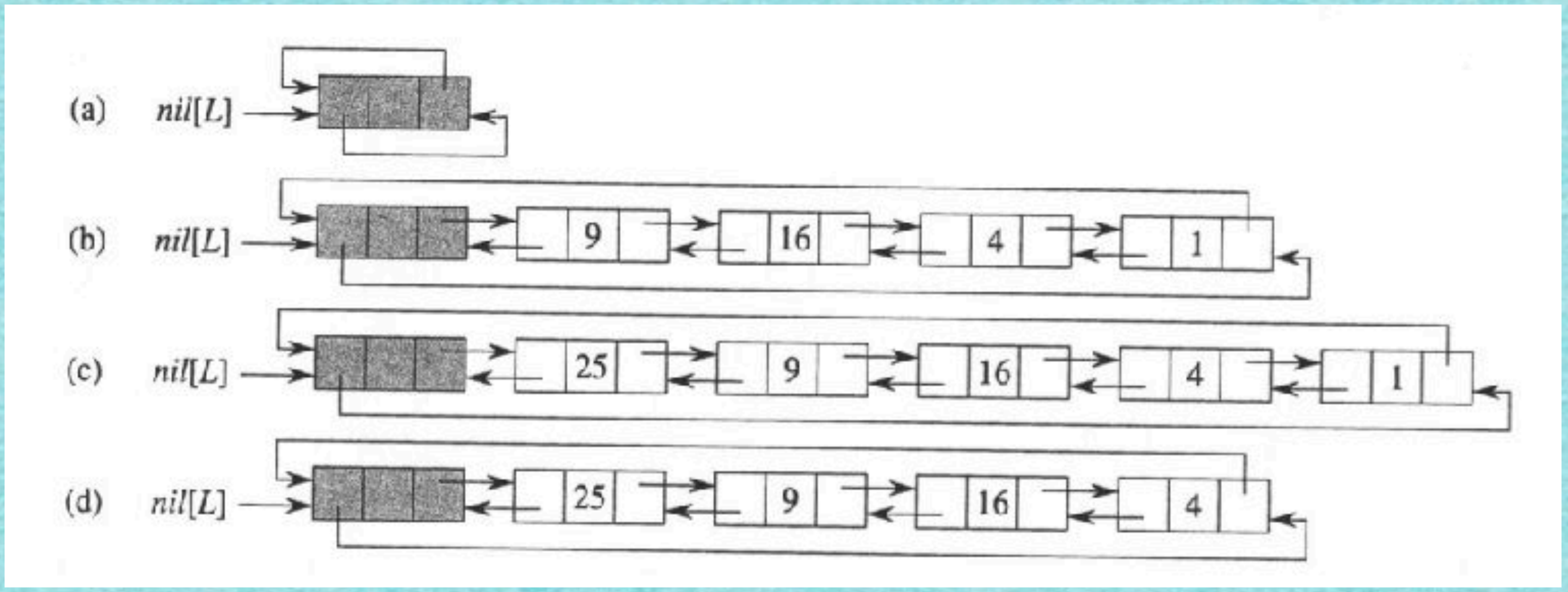
```
LIST-DELETE(L, x)
if vorg[x] ≠ NIL
    then nachf[vorg[x]] ← nachf[x]
    else   kopf[L] ← nachf[x]
if nachf[x] ≠ NIL
    then vorg[nachf[x]] ← vorg[x]
```

Alternative: Zyklische Struktur mit "Wächter" nil[L]

Alternative: Zyklische Struktur mit "Wächter" nil[L]



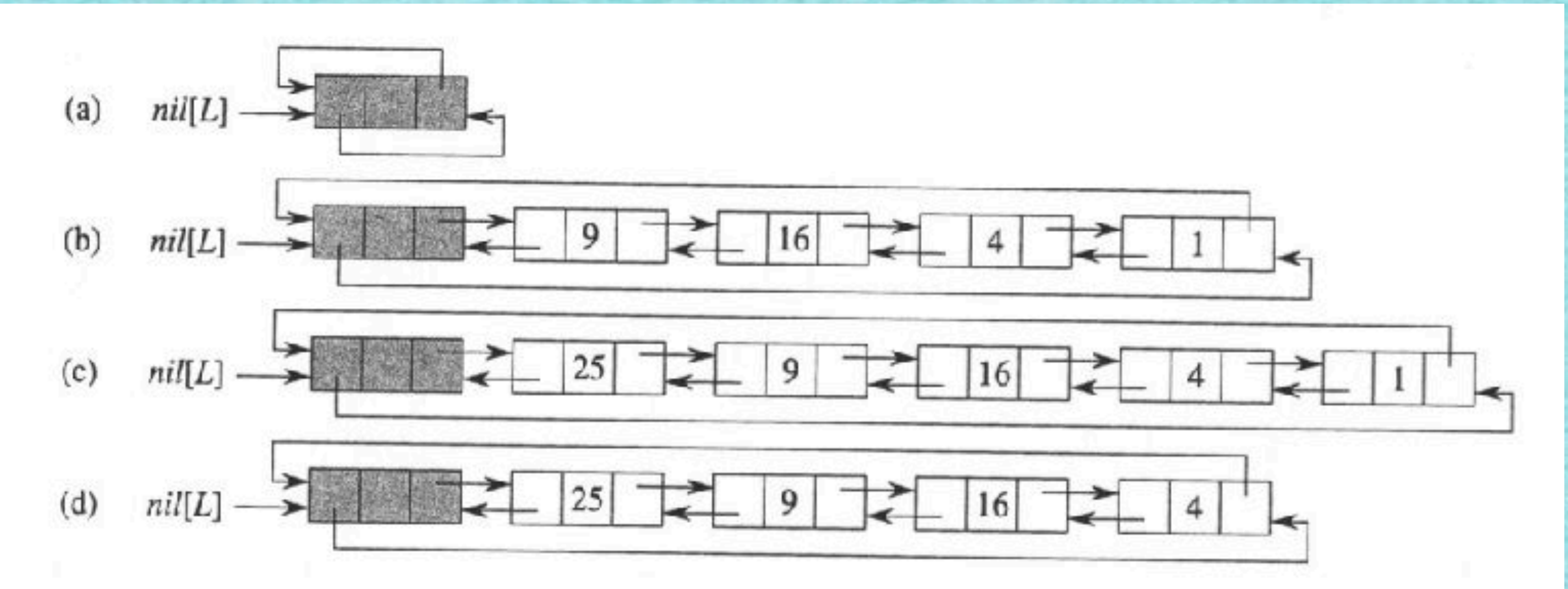
Alternative: Zyklische Struktur mit "Wächter" nil[L]



LIST-INSERT′(L, x)

$nachf[x] \leftarrow nachf[nil[L]]$
$vorg[nachf[nil[L]]] \leftarrow x$
$nachf[nil[L]] \leftarrow x$
$vorg[x] \leftarrow nil[L]$

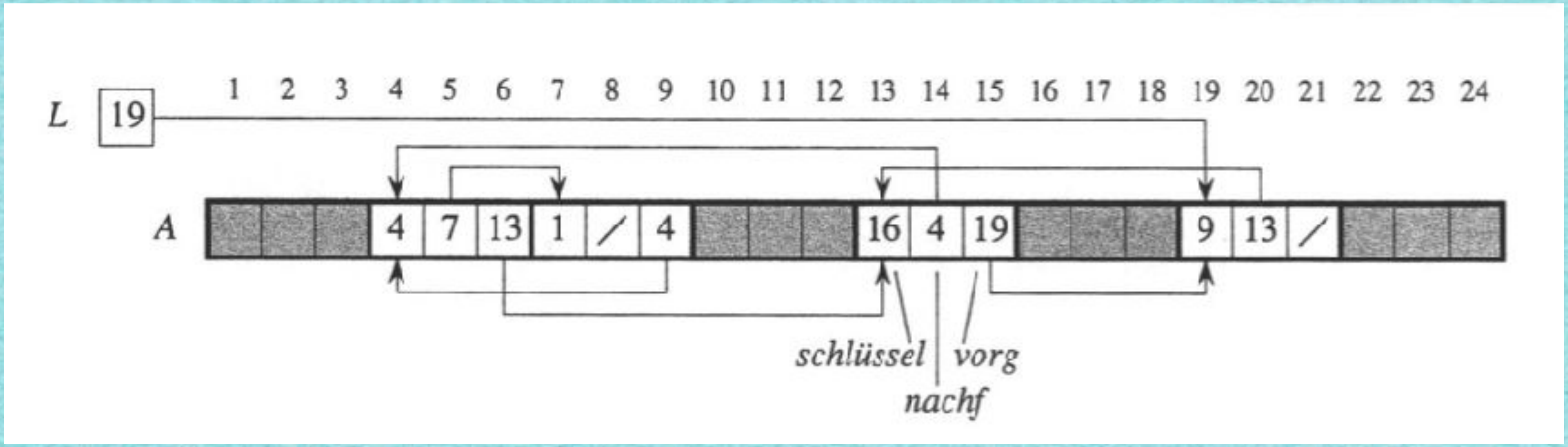
Alternative: Zyklische Struktur mit Wächter "nil[L]"

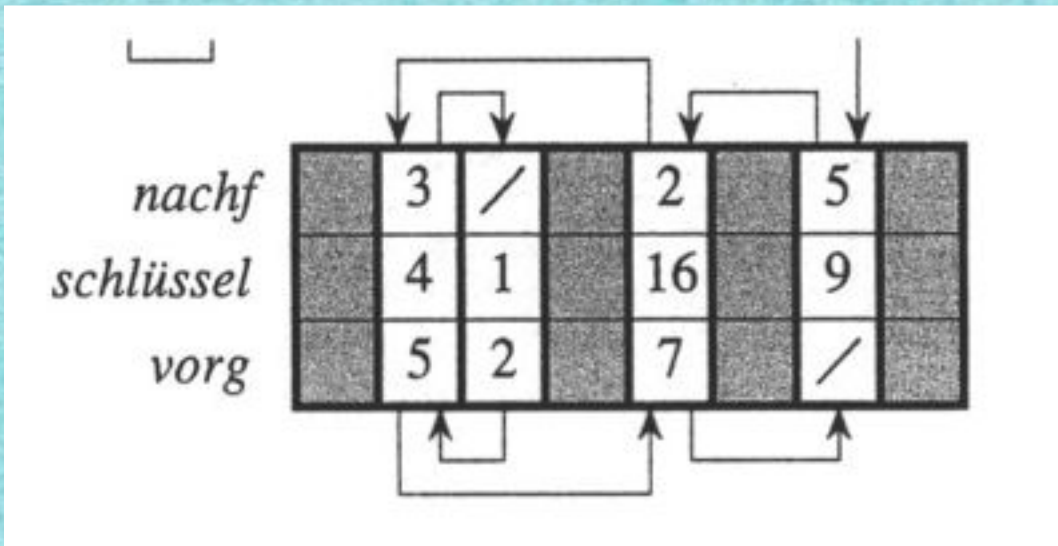



```
LIST-SEARCH'(L, k)
  x ← nachf[nil[L]]
  while x ≠ nil[L] und schlüssel[x] ≠ k
        do x ← nachf[x]
  return x
```

```
LIST-DELETE'(L, x)
  nachf[vorg[x]] ← nachf[x]
  vorg[nachf[x]] ← vorg[x]
```

Speicherung kann irgendwo erfolgen!





Mehr demnächst!

s.fekete@tu-bs.de