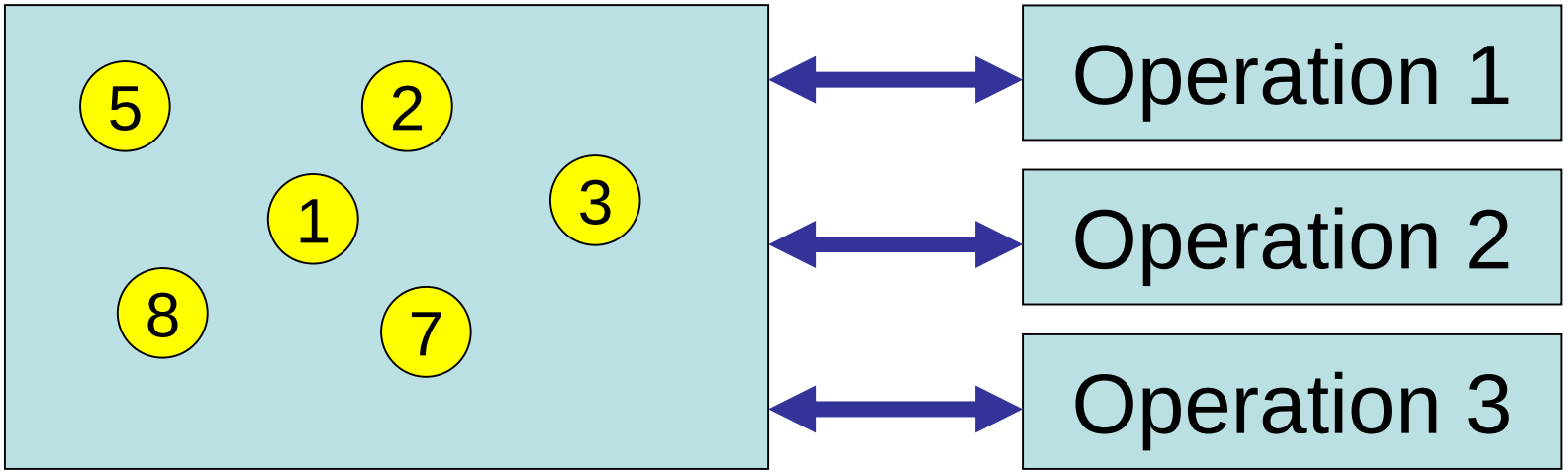
10. Elementare Datenstrukturen

Definition 10.1: Eine Datenstruktur ist gegeben durch eine Menge von Objekten sowie einer Menge von Operationen auf diesen Objekten.



10. Elementare Datenstrukturen

Definition 10.1: Eine Datenstruktur ist gegeben durch eine Menge von Objekten sowie einer Menge von Operationen auf diesen Objekten.

Definition 10.2:

1. Werden auf einer Menge von Objekten die Operationen Einfügen, Entfernen und Suchen betrachtet, so spricht man von einem Wörterbuch (engl. Dictionary).

2. Werden auf einer Menge von Objekten die Operationen Einfügen, Entfernen und Suchen des Maximums (Minimums) betrachtet, so spricht man von einer Prioritätswarteschlange (engl. Priority Queue).

Datenstrukturen

Ein grundlegendes Datenbank-Problem

- Speicherung von Datensätzen

Beispiel:

- Kundendaten (Name, Adresse, Wohnort, Kundennummer, offene Rechnungen, offene Bestellungen,…)

Anforderungen:

- Schneller Zugriff
- Einfügen neuer Datensätze
- Löschen bestehender Datensätze

Datenstrukturen

Zugriff auf Daten:

- Jedes Objekt hat einen Schlüssel
- Eingabe des Schlüssels liefert Datensatz
- Schlüssel sind vergleichbar (es gibt totale Ordnung der Schlüssel)

Beispiel:

- Kundendaten (Name, Adresse, Kundennummer)
- Schlüssel: Name
- Totale Ordnung: Lexikographische Ordnung

Datenstrukturen

Zugriff auf Daten:

- Jedes Objekt hat einen Schlüssel
- Eingabe des Schlüssels liefert Datensatz
- Schlüssel sind vergleichbar (es gibt totale Ordnung der Schlüssel)

Beispiel:

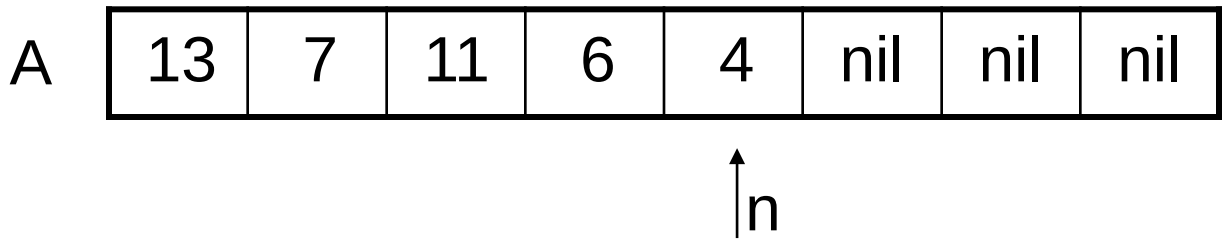
- Kundendaten (Name, Adresse, Kundennummer)
- Schlüssel: Kundennummer
- Totale Ordnung: ‚≤‘

Elementare Operationen

- Insert(*S*,*x*): Füge Objekt *x* in Menge *S* ein.
- Search(*S*,*k*): Finde Objekt *x* mit Schlüssel *k*. Falls kein solches Objekt vorhanden Ausgabe NIL
- Remove(*S*,*x*): Entferne Objekt *x* aus Menge *S*.
- Minimum(*S*): Finde Objekt mit minimalem Schlüssel in *S* (es muss eine Ordnung auf Schlüsseln existieren).
- Maximum(*S*): Finde Objekt mit maximalem Schlüssel in *S* (es muss eine Ordnung auf Schlüsseln existieren).

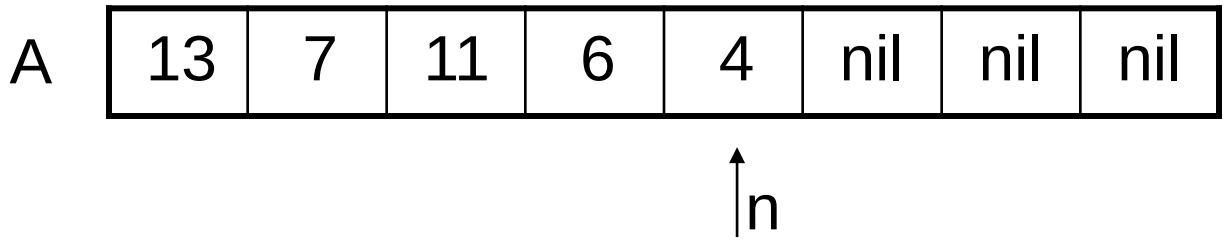
Unsere erste Datenstruktur:

- Array A[1,…,max]
- Integer n, $1 \leq n \leq max$
- n bezeichnet Anzahl Elemente in Datenstruktur



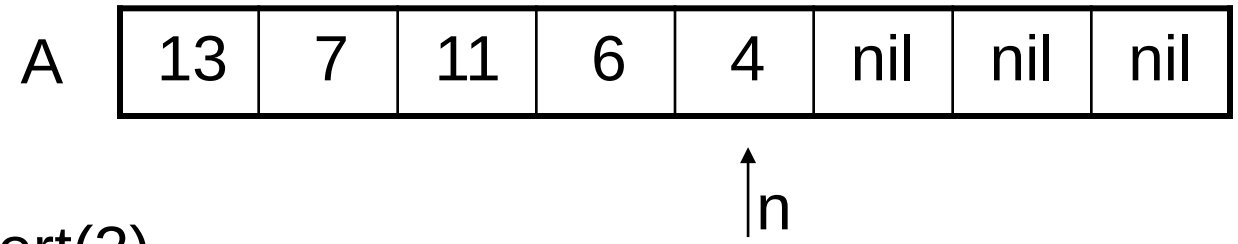
Insert(s)

1. **if** n=max **then return** „Fehler: Kein Platz in Datenstruktur“
2. **else**
3. n ← n+1
4. A[n] ← s



Insert(s)

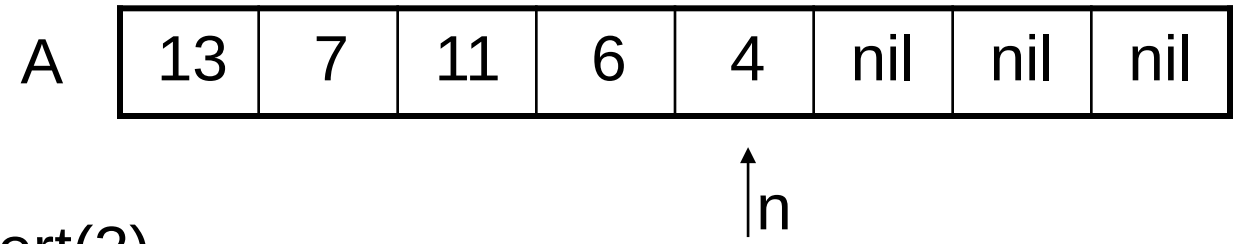
1. **if** n=max **then return** „Fehler: Kein Platz in Datenstruktur“
2. **else**
3. n ← n+1
4. A[n] ← s



Insert(2)

Insert(s)

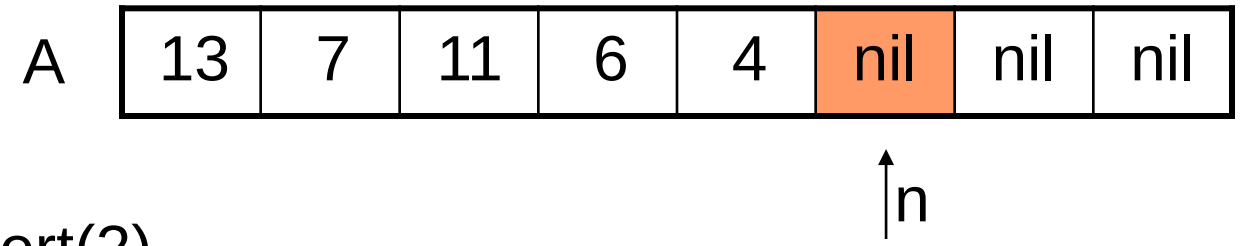
1. **if** n=max **then return** „Fehler: Kein Platz in Datenstruktur“
2. **else**
3. n ← n+1
4. A[n] ← s



Insert(2)

Insert(s)

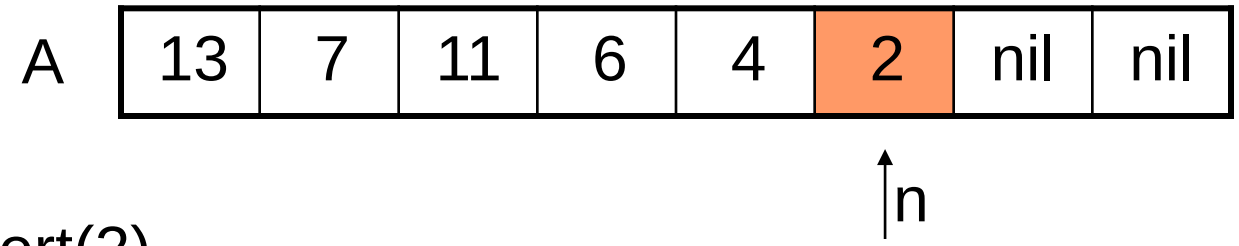
1. **if** n=max **then return** „Fehler: Kein Platz in Datenstruktur“
2. **else**
3. n ← n+1
4. A[n] ← s



Insert(2)

Insert(s)

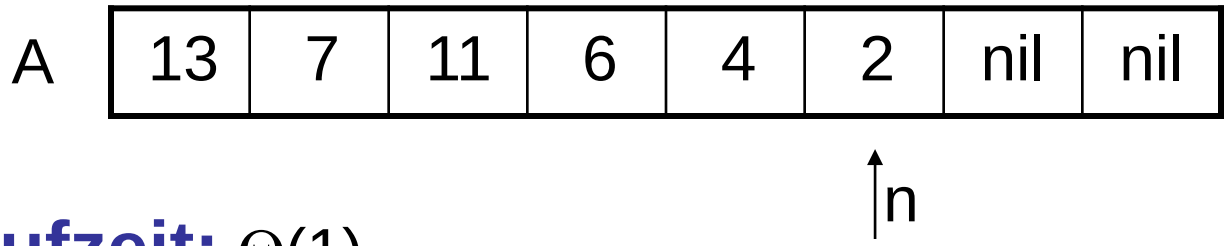
1. **if** n=max **then return** „Fehler: Kein Platz in Datenstruktur“
2. **else**
3. n ← n+1
4. A[n] ← s



Insert(2)

Insert(s)

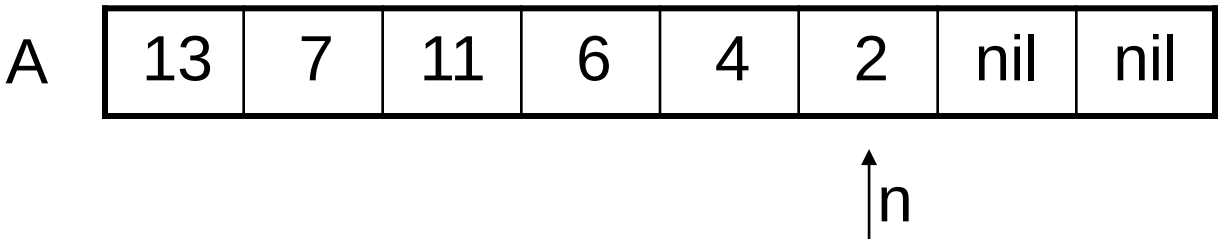
1. **if** n=max **then return** „Fehler: Kein Platz in Datenstruktur“
2. **else**
3. n ← n+1
4. A[n] ← s



Laufzeit: $\Theta(1)$

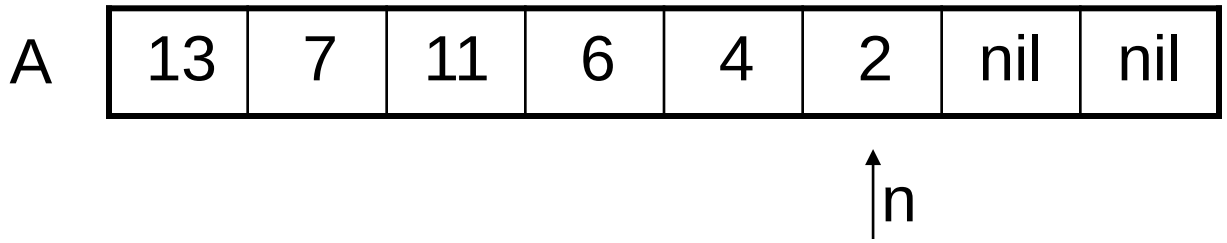
Search(x)

1. **for** i $\leftarrow$ 1 **to** n **do**
3. **if** A[i] = x **then return** i
4. return **nil**



Search(x)

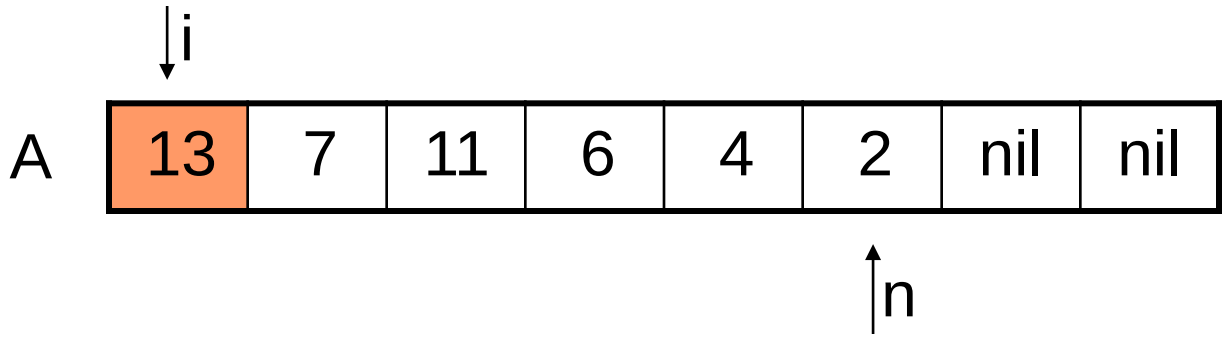
1. **for** i $\leftarrow$ 1 **to** n **do**
3. **if** A[i] = x **then return** i
4. return **nil**



Search(11)

Search(x)

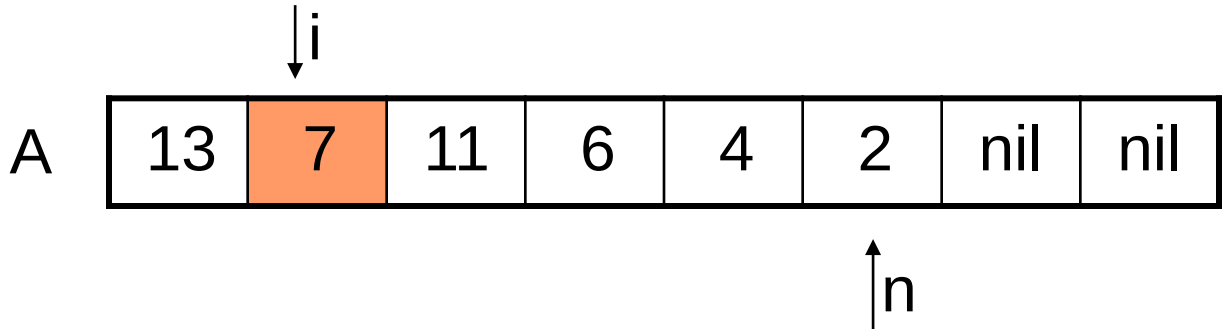
1. **for** i ← 1 **to** n **do**
3. **if** A[i] = x **then return** i
4. return **nil**



Search(11)

Search(x)

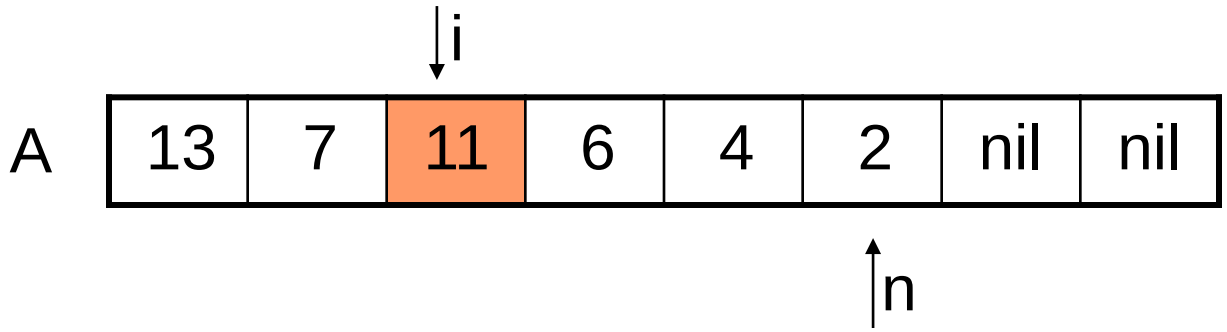
1. **for** i ← 1 **to** n **do**
3. **if** A[i] = x **then return** i
4. return **nil**



Search(11)

Search(x)

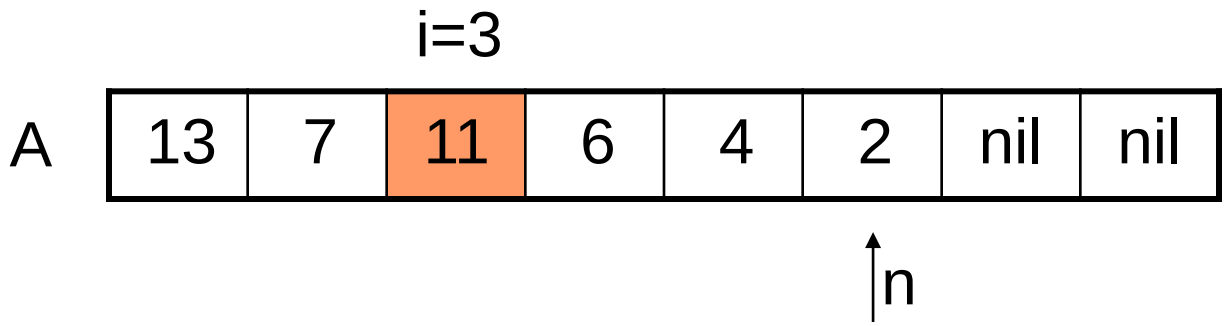
1. **for** i ← 1 **to** n **do**
3. **if** A[i] = x **then return** i
4. return **nil**



Search(11)

Search(x)

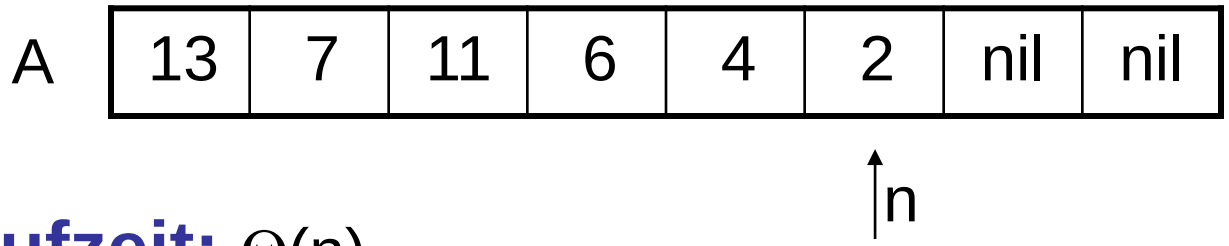
1. **for** i ← 1 **to** n **do**
3. **if** A[i] = x **then** **return** i
4. return **nil**



Search(11)

Search(x)

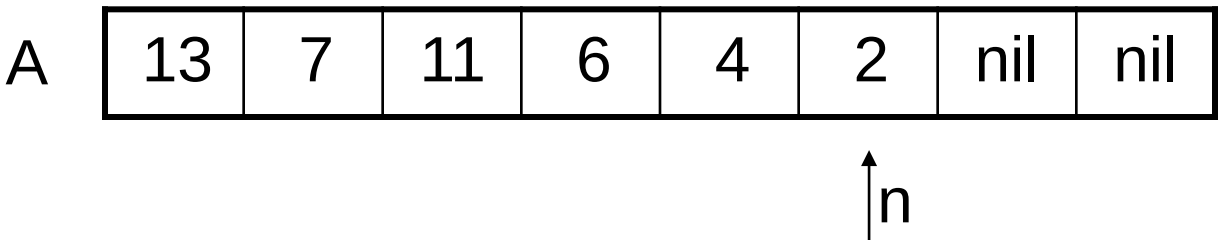
1. **for** i ← 1 **to** n **do**
3. **if** A[i] = x **then** **return** i
4. return **nil**



Laufzeit: $\Theta(n)$

Remove(i)

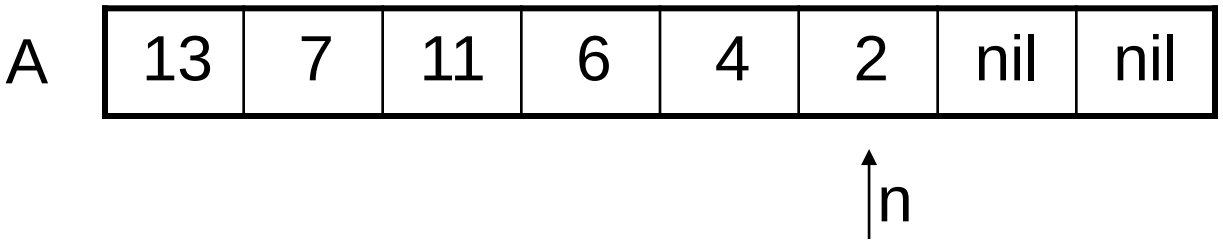
1. A[i] ← A[n]
2. A[n] ← **nil**
3. n ← n-1



Remove(i)

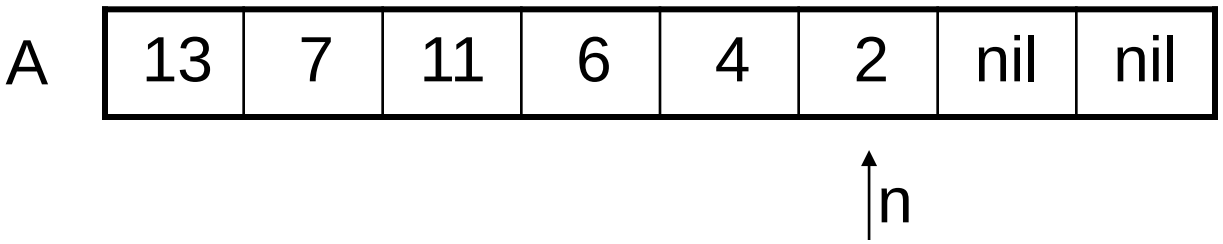
1. A[i] ← A[n]
2. A[n] ← **nil**
3. n ← n-1

Annahme:
Wir bekommen Index i des zu löschenden Objekts



Remove(i)

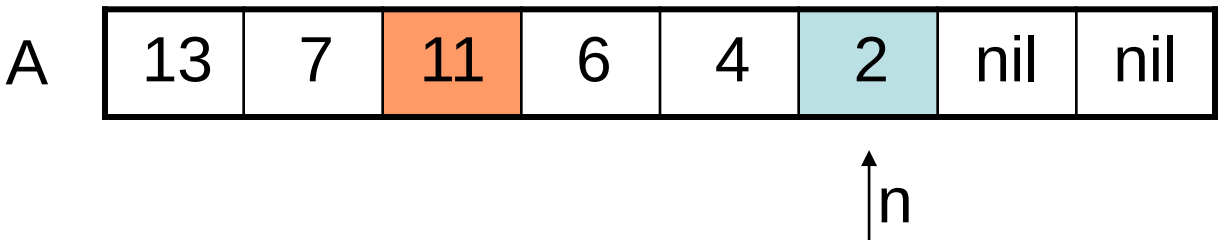
1. A[i] ← A[n]
2. A[n] ← **nil**
3. n ← n-1



Remove(3)

Remove(i)

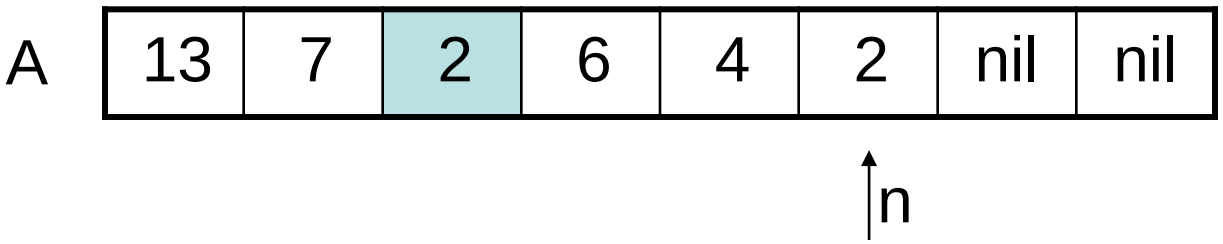
1. A[i] ← A[n]
2. A[n] ← **nil**
3. n ← n-1



Remove(3)

Remove(i)

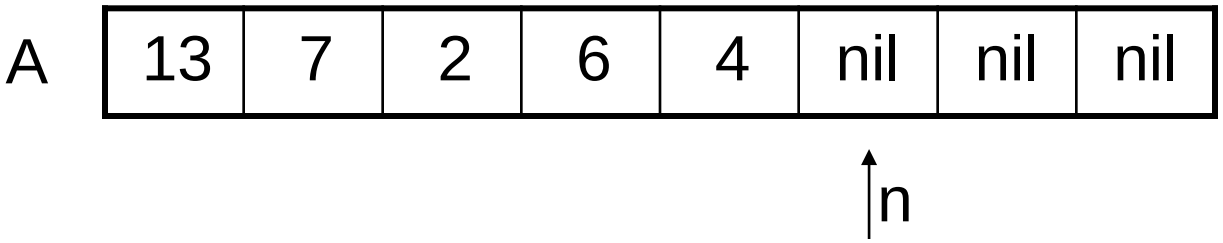
1. A[i] ← A[n]
2. A[n] ← **nil**
3. n ← n-1



Remove(3)

Remove(i)

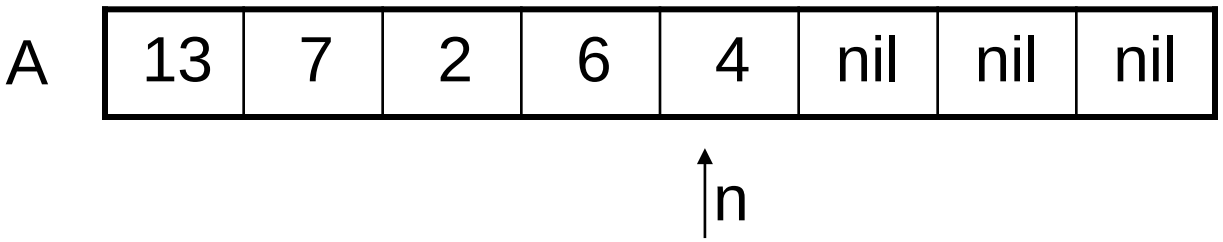
1. A[i] ← A[n]
2. A[n] ← **nil**
3. n ← n-1



Remove(3)

Remove(i)

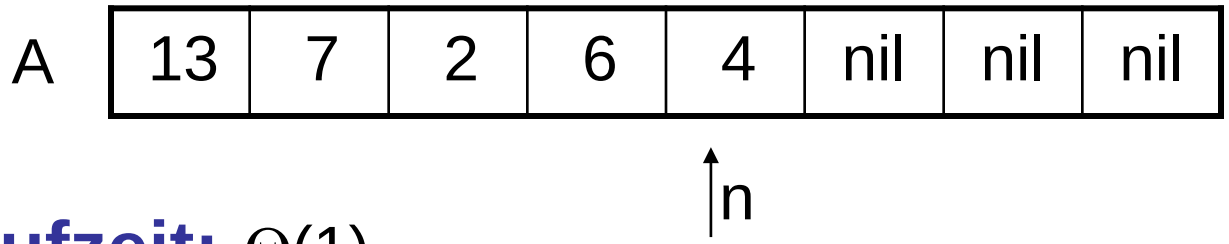
1. A[i] ← A[n]
2. A[n] ← **nil**
3. n ← n-1



Remove(3)

Remove(i)

1. A[i] ← A[n]
2. A[n] ← **nil**
3. n ← n-1



Laufzeit: $\Theta(1)$

Datenstrukturen

Datenstruktur Feld:

- Platzbedarf $\Theta(max)$
- Laufzeit Suche: $\Theta(n)$
- Laufzeit Einfügen/Löschen: $\Theta(1)$

Vorteile:

- Schnelles Einfügen und Löschen

Nachteile:

- Speicherbedarf abhängig von max (nicht vorhersagbar)
- Hohe Laufzeit für Suche

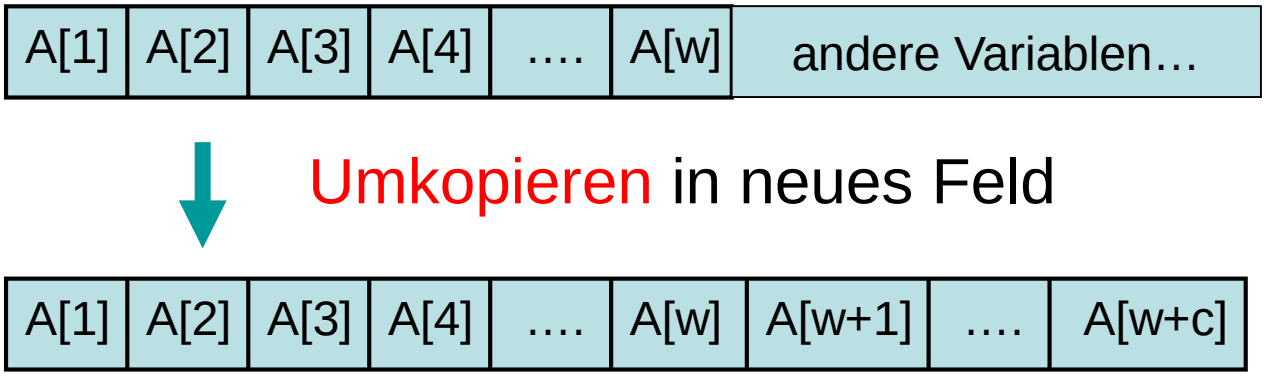
Dynamisches Feld

Im folgenden: w=max.

Operationen: new (fordere Speicher an) und delete (gib Speicher frei)

Erste Idee:

- Jedesmal, wenn Feld A nicht mehr ausreicht (n>w), generiere neues Feld der Größe w+c für ein festes c.



Dynamisches Feld

Zeitaufwand für Erweiterung ist $\Theta(w)$:

Zeitaufwand für n Insert Operationen:

- Aufwand von $\Theta(w)$ je c Operationen
- Gesamtaufwand: $\Theta(\sum_{i=1}^{n/c} c \cdot i) = \Theta(n^2)$

Dynamisches Feld

Bessere Idee:

- Jedesmal, wenn Feld A nicht mehr ausreicht (n>w), generiere neues Feld der doppelten Größe 2w.

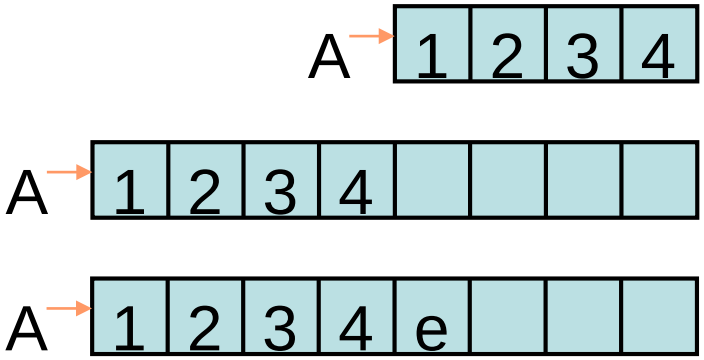
- Jedesmal, wenn Feld A zu groß ist (n≤w/4), generiere neues Feld der halben Größe w/2.

Dynamisches Feld

Seien $\alpha=1/4$ und $\beta=2$.

```
Insert(e):
   if n=w then
      A←reallocate(A,βn)
   n←n+1
   A[n]←e
```

n=w=4:



Dynamisches Feld

Seien $\alpha=1/4$ und $\beta=2$.

Remove(i):

A[i]←A[n]

A[n]←nil

n←n-1

if $n \leq \alpha w$ and $n>0$ then

A←reallocate(A,n/β)

n=5, w=16:

Dynamisches Feld

```
reallocate(A,w´):
    w←w´
    A´←new Array[1..w] of Element
    for i←1 to n do
        A´[i] ←A[i]      } Umkopieren
    delete A
    return A´
```

Dynamisches Feld

Lemma 10.3: Betrachte ein anfangs leeres dynamisches Feld A. Jede Folge σ von m Insert und Remove Anfragen kann auf A in Zeit O(m) bearbeitet werden.

- Erste Idee: Laufzeit $O(m^2)$
- Genauer: Im worst case nur durchschnittlich konstante Laufzeit pro Operation (Fachbegriff dafür: amortisiert)

Amortisierte Analyse

- Feldverdopplung:

- Feldhalbierung:

- Von 

  - Nächste Verdopplung: ≥n Insert Operationen
  - Nächste Halbierung: ≥n/2 Remove Ops

Amortisierte Analyse

- Von 

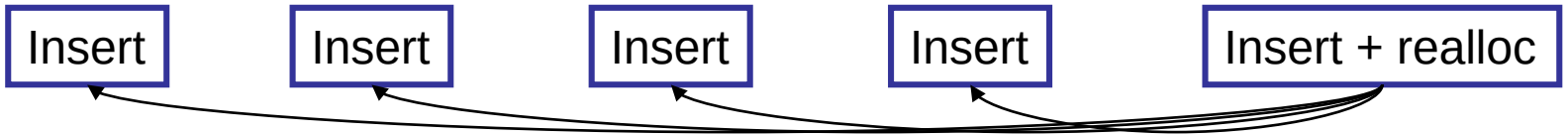
  - Nächste Verdopplung: $\geq n$ Insert Ops
  - Nächste Halbierung: $\geq n/2$ Remove Ops

- Idee: verrechne reallocate-Kosten mit Insert/Remove Kosten (ohne realloc)
  - Kosten für Insert/Remove: $O(1)$
  - Kosten für reallocate(βn): $O(n)$

Amortisierte Analyse

- Idee: verrechne reallocate-Kosten mit Insert/Remove Kosten
  - Kosten für Insert/Remove: O(1)
  - Kosten für reallocate(βn): O(n)

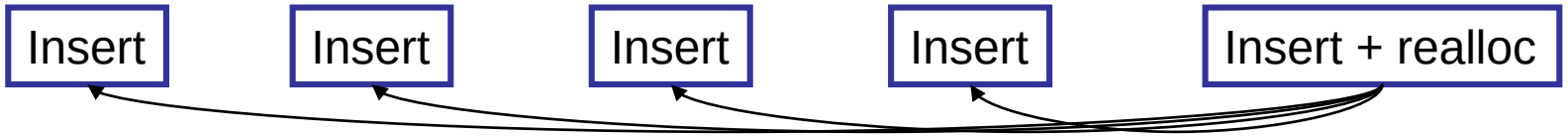
- Formale Verrechnung: Zeugenzuordnung



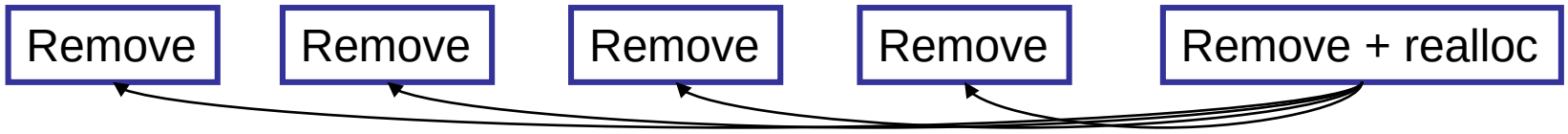
Reallokation bei n Elementen: bezeugt durch letzte n/2 Insert Operationen

Amortisierte Analyse

- Formale Verrechnung: Zeugenzuordnung



Reallokation bei n Elementen: bezeugt durch letzte n/2 Insert Operationen



Reallokation bei n Elementen: bezeugt durch letzte n Remove Operationen

- Dann jede Ins/Rem Op nur 1x Zeuge

Amortisierte Analyse

- Idee: verrechne reallocate-Kosten mit Insert/Remove Kosten
  - Kosten für Insert/Remove: $O(1)$
  - Kosten für reallocate(βn): $O(n)$
- Konkret:
  - $\Theta(n)$ Zeugen pro reallocate(βn)
  - verteile $O(n)$ Aufwand gleichmäßig auf Zeugen
- Gesamtaufwand: $O(m)$ bei m Operationen

Amortisierte Analyse

Alternative Sicht zur Zeugenmethode:
Kontenmethode

Kontenmethode: Spiel mit Zeittokens

- Günstige Operationen zahlen Tokens ein
- Teure Operationen entnehmen Tokens
- Tokenkonto darf nie negativ werden!

Amortisierte Analyse

Kontenmethode: Spiel mit Zeittokens

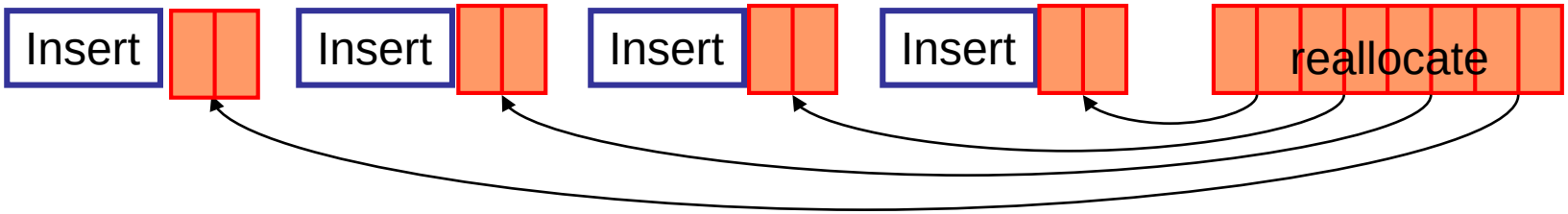
Annahme: Zeittoken sind genügend groß gewählte Zeiteinheiten, so dass Insert und Remove (ohne reallocate) eine Laufzeit von höchstens einem Zeittoken haben.

- Günstige Operationen zahlen Tokens ein
  $\rightarrow$ pro Insert 2 zusätzliche Tokens
  $\rightarrow$ pro Remove 1 zusätzlichen Token
- Teure Operationen entnehmen Tokens
  $\rightarrow$ pro reallocate(βn) –n Tokens
- Tokenkonto darf nie negativ werden!
  $\rightarrow$ erfüllt über Zeugenargument

Amortisierte Analyse

Laufzeit über Zeittoken:

- Ausführung von Insert/Remove kostet 1 Token
  → Tokenkosten für Insert: 1+2 = 3
  → Tokenkosten für Remove: 1+1 = 2
- Ausführung von reallocate(βn) kostet n Tokens
  → Tokenkosten für reallocate(βn): $n-n=0$



Gesamtlaufzeit = O(Summe der Tokenlaufzeiten)

Amortisierte Analyse

Wir haben schon kennengelernt:

- Zeugenmethode
- Kontenmethode

Im folgenden:

- allgemeine Herangehensweise
- Potenzialmethode

Amortisierte Analyse

- S: Zustandsraum einer Datenstruktur
- F: beliebige Folge von Operationen Op_1, Op_2, Op_3,…,Op_n
- s_0: Anfangszustand der Datenstruktur

$$s_0 \xrightarrow{Op_1} s_1 \xrightarrow{Op_2} s_2 \xrightarrow{Op_3} \ldots \xrightarrow{Op_n} s_n$$

- Zeitaufwand $T(F) = \sum_{i=1}^{n} T_{Op_i}(s_{i-1})$

Amortisierte Analyse

- Zeitaufwand $T(F) = \sum_{i=1}^{n} T_{Op_i}(s_{i-1})$
- Eine Familie von Funktionen $A_X(s)$, eine pro Operation X, heißt Familie amortisierter Zeitschranken falls für jede Sequenz F von Operationen gilt

  $$T(F) \leq A(F) := c + \sum_{i=1}^{n} A_{Op_i}(s_{i-1})$$

  für eine Konstante c unabhängig von F
- Wir können also für die Abschätzung der worst-case Laufzeit einer Folge von Operationen so tun, also ob $A_X(s)$ die worst-case Laufzeit von Operation X bei Zustand s ist. Es ist aber durchaus möglich, dass für eine einzelne Operation X $A_X(s) << T_X(s)$ ist.

Amortisierte Analyse

- Triviale Wahl von $A_X(s)$:
  $A_X(s) := T_X(s)$
- Dynamisches Feld (Zeittoken gen. groß):
  $A_{Insert}(s):=3$, $A_{Remove}(s):=2$, $A_{rellocate}(s):=0$
- alternative Wahl von $A_X(s)$:
  über Potenzial $\phi: S \rightarrow \mathbb{R}_{\geq 0}$
  $\rightarrow$vereinfacht Beweisführung

Amortisierte Analyse

Satz 10.4: Sei S der Zustandsraum einer Datenstruktur, sei s_0 der Anfangszustand und sei $\phi: S \to \mathbb{R}_{\geq 0}$ eine nichtnegative Funktion. Für eine Operation X und einen Zustand s mit $s \to s'$ definiere

$$A_X(s) := T_X(s) + (\phi(s') - \phi(s)).$$

Dann sind die Funktionen $A_X(s)$ eine Familie amortisierter Zeitschranken.

Amortisierte Analyse

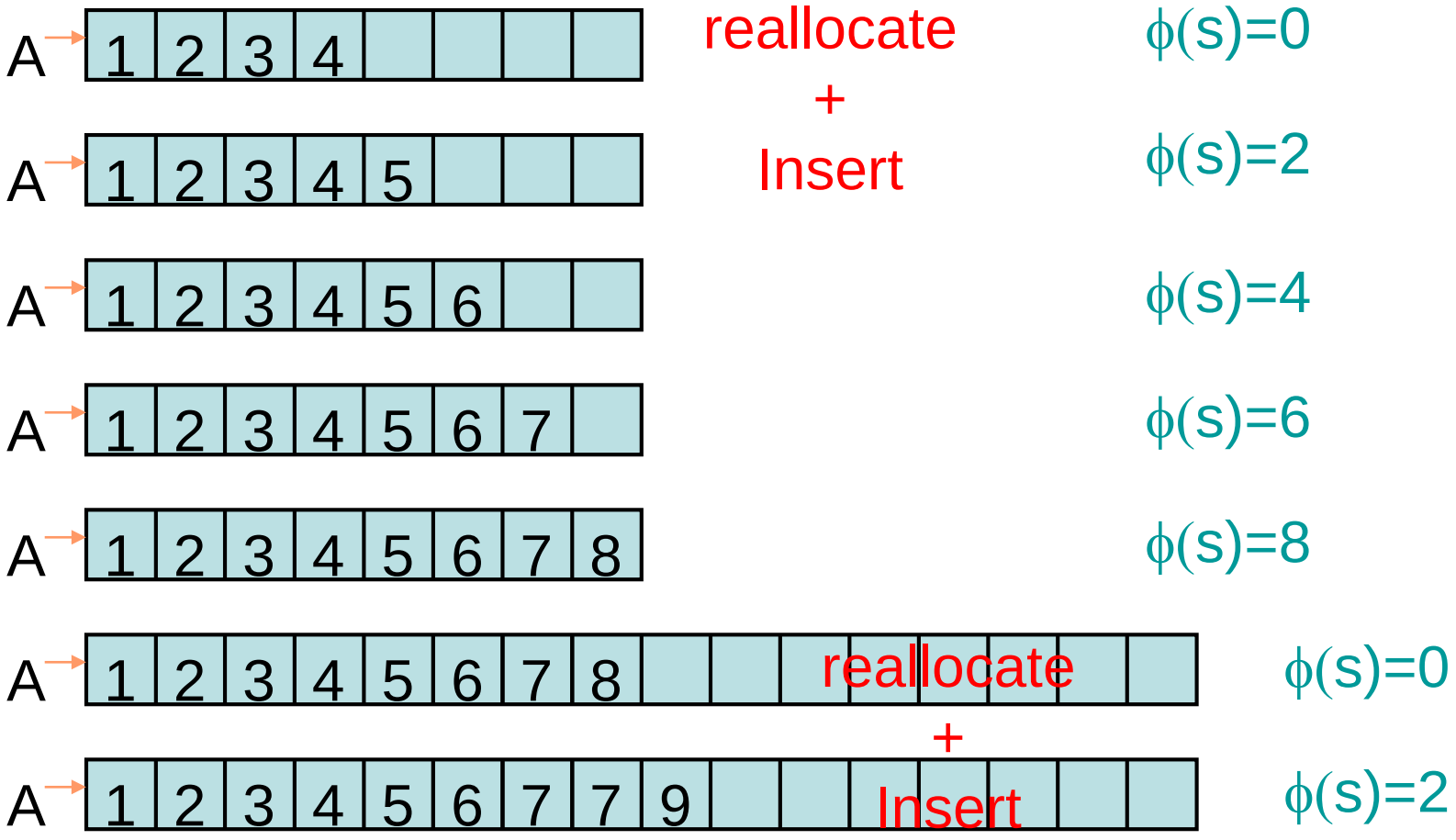
Zu zeigen: $T(F) \leq c + \sum_{i=1}^{n} A_{Op_i}(s_{i-1})$

Beweis:

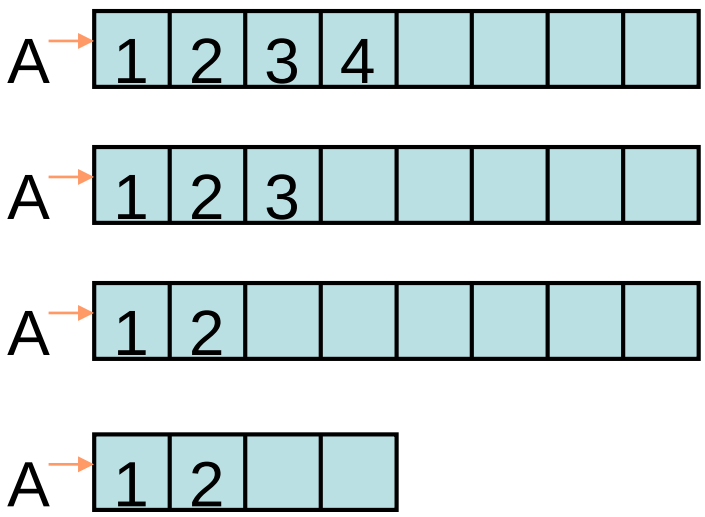
$$\sum_{i=1}^{n} A_{Op_i}(s_{i-1}) = \sum_{i=1}^{n} [T_{Op_i}(s_{i-1}) + \phi(s_i) - \phi(s_{i-1})]$$
$$= T(F) + \sum_{i=1}^{n} [\phi(s_i) - \phi(s_{i-1})]$$
$$= T(F) + \phi(s_n) - \phi(s_0)$$
$$\Rightarrow T(F) = \sum_{i=1}^{n} A_{Op_i}(s_{i-1}) + \phi(s_0) - \phi(s_n)$$
$$\leq \sum_{i=1}^{n} A_{Op_i}(s_{i-1}) + \phi(s_0)$$

konstant

Beispiel: Dynamisches Feld



Beispiel: Dynamisches Feld



Remove
+
reallocate

$\phi(s)=0$

$\phi(s)=2$

$\phi(s)=4$

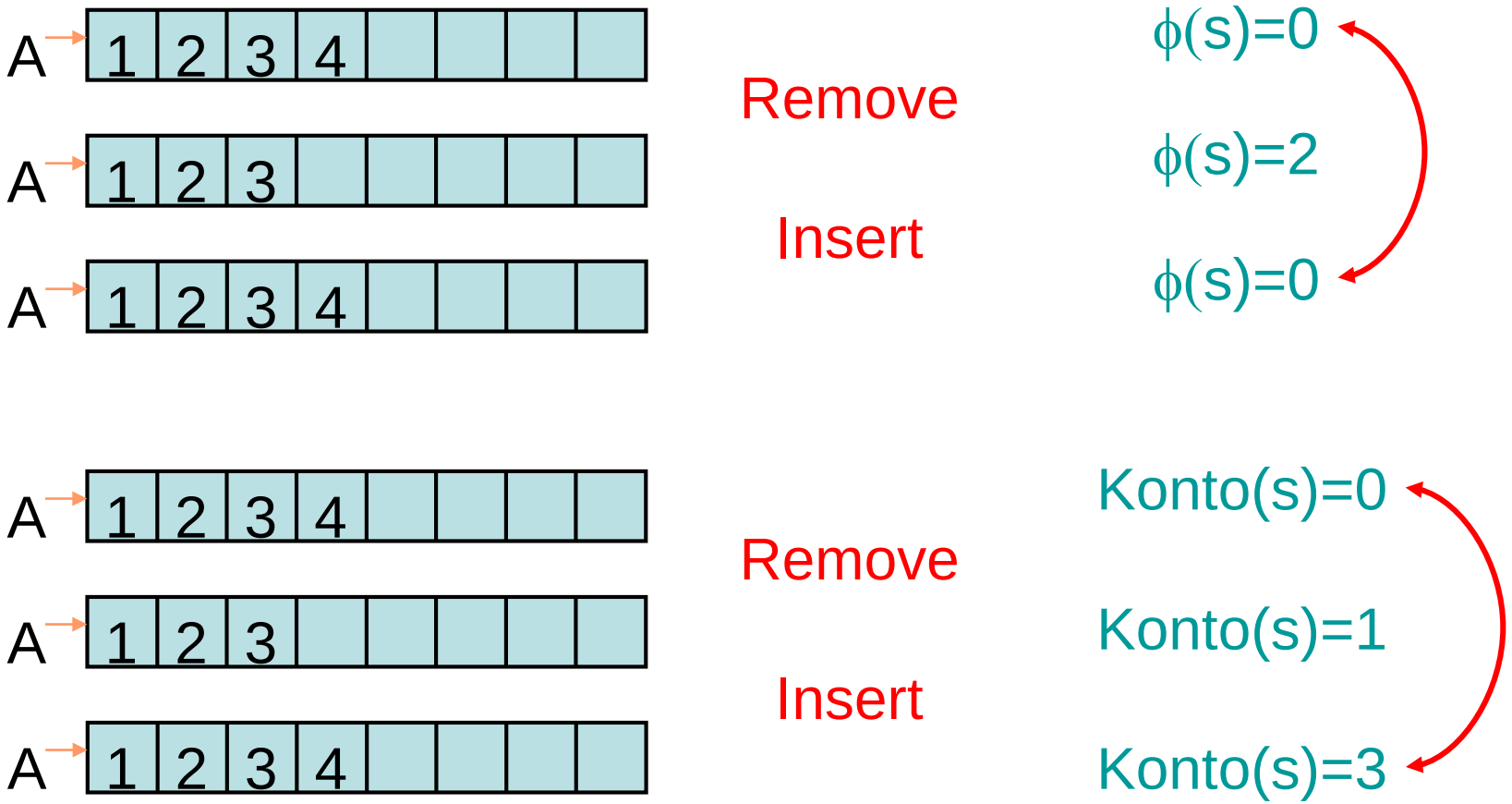
$\phi(s)=0$

Generelle Formel für $\phi(s)$:
(w_s: Feldgröße von A, n_s: Anzahl Einträge)

$$\phi(s) = 2|w_s/2 - n_s|$$

Beispiel: Dynamisches Feld

Potenzial ist nicht gleich Konto!



Beispiel: Dynamisches Feld

Formel für $\phi(s)$:

(w_s: Feldgröße von A in Zustand s,
n_s: Anzahl Einträge)

$$\phi(s) = 2|w_s/2 - n_s|$$

Satz 10.5:

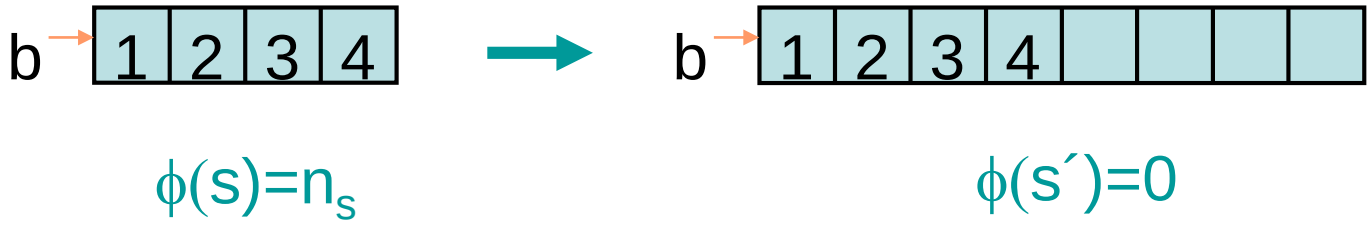
Sei $\Delta\phi = \phi(s') - \phi(s)$ für $s \rightarrow s'$

- ϕ nicht negativ, $\phi(s_0)=1$ (w=1, n=0)
- $A_{Insert}(s) = \Delta\phi + T_{Insert}(s) \leq 2+1 = 3$
- $A_{Remove}(s) = \Delta\phi + T_{Remove}(s) \leq 2+1 = 3$
- $A_{realloc}(s) = \Delta\phi + T_{realloc}(s) \leq (0-n_s)+n_s = 0$

Beispiel: Dynamisches Feld

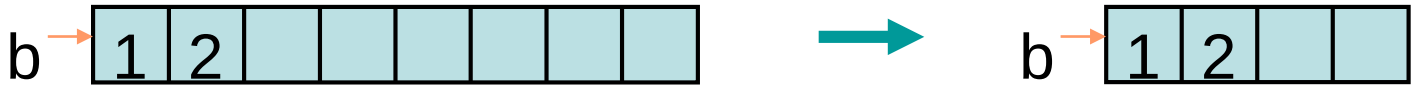
Beweis für $A_{realloc}(s) \leq 0$:

- Fall 1:



$\phi(s)=n_s$ $\phi(s')=0$

- Fall 2:



$\phi(s)=2n_s$ $\phi(s')=0$

Amortisierte Analyse

Die Potenzialmethode ist universell!

Satz 10.6: Sei $B_X(s)$ eine beliebige Familie amortisierter Zeitschranken. Dann gibt es eine Potenzialfunktion ϕ, so dass $A_X(s) \leq B_X(s)$ für alle Zustände s und alle Operationen X gilt, wobei $A_X(s)$ definiert ist wie in Satz 10.4.

Problem: finde geeignetes Potenzial!

Wenn erstmal gefunden, dann Rest einfach.

Datenstruktur Dynamisches Feld:

- Platzbedarf $\Theta(n)$
- Laufzeit Suche: $\Theta(n)$
- Amortisierte Laufzeit Einfügen/Löschen: $\Theta(1)$

Kann durch Tricks (progressives Umkopieren) in worst case Laufzeit O(1) umgewandelt werden.

Vorteile:

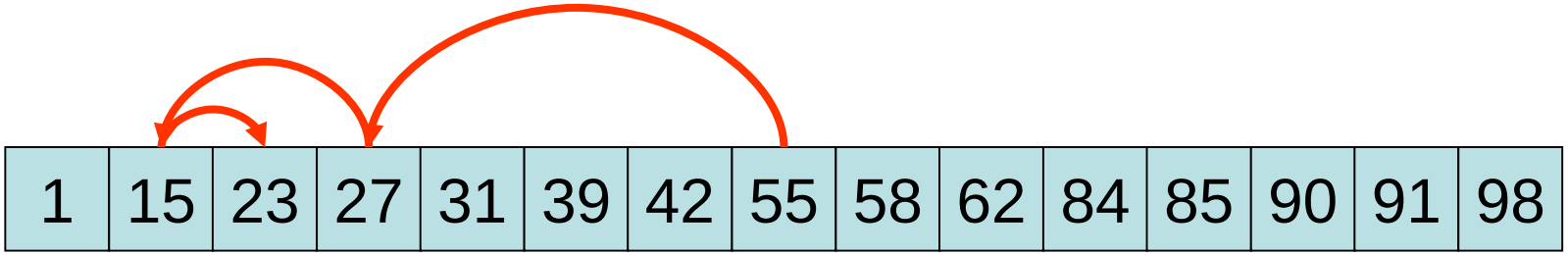
- Schnelles Einfügen und Löschen
- Speicherbedarf abhängig von n

Nachteile:

- Hohe Laufzeit für Suche

Warum nicht sortiertes Feld?

Mit sortiertem Feld binäre Suche möglich.

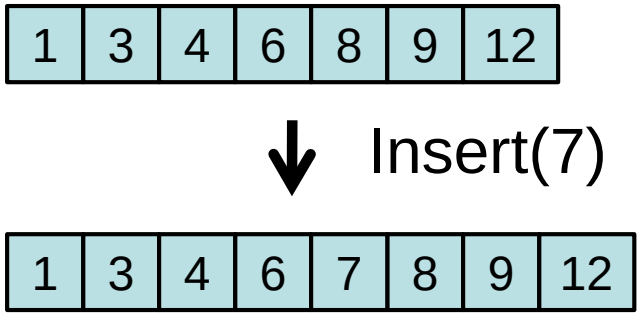


Im n-elementigen sortierten Feld wird jedes Element in max. log n Schritten gefunden.

Datenstruktur Sortiertes Dynamisches Feld:

- Platzbedarf $\Theta(n)$
- Laufzeit Suche: $\Theta(\log n)$
- Laufzeit Einfügen/Löschen: $\Theta(n)$

Grund: Folge muss auseinander (beim Einfügen) oder zusammen (beim Löschen) geschoben werden, was $\Theta(n)$ Laufzeit verursachen kann (siehe Insertionsort).



Sortiertes Feld

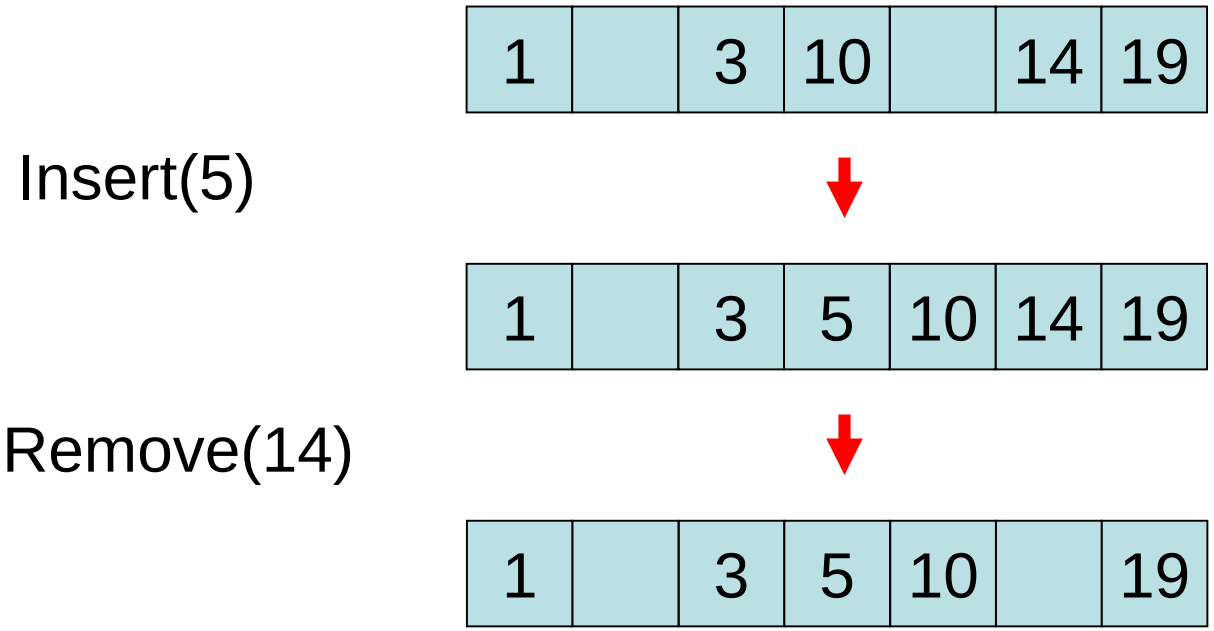
Kann man Insert und Remove besser mit einem sortierten Feld realisieren?

- folge Beispiel der Bibliothek!

Sortiertes Feld

Bibliotheksprinzip: lass Lücken!

Angewandt auf sortiertes Feld:

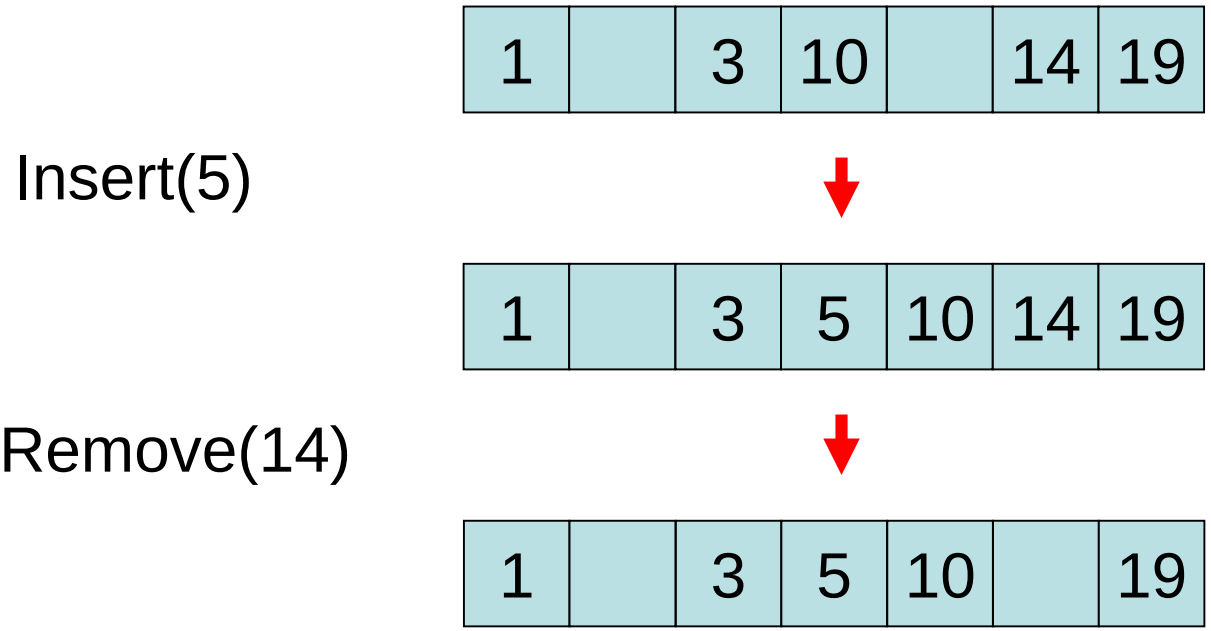


Sortiertes Feld

Durch geschickte Verteilung der Lücken:

amortierte Kosten für insert und remove $\Theta(\log^2 n)$

Analyse allerdings komplex!



Datenstruktur Sortiertes Dynamisches Feld mit Lücken:

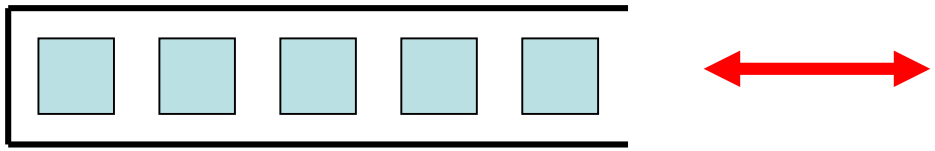
- Platzbedarf $\Theta(n)$
- Laufzeit Suche: $\Theta(\log n)$
- Amortisierte Laufzeit Einfügen/Löschen: $\Theta(\log^2 n)$

Noch bessere Laufzeiten möglich mit geeigneten Zeigerstrukturen oder Hashtabellen.

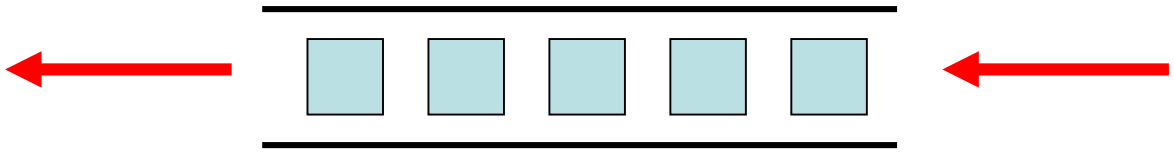
Stacks (Stapel) und Queues (Schlangen)

Definition 10.7:

1. Stacks (Stapel) sind eine Datenstruktur, die die LIFO (last-in-first-out) Regel implementiert.
   Bei der LIFO Regel soll das zuletzt eingefügte Objekt entfernt werden.



2. Queues (Schlangen) sind eine Datenstruktur, die die FIFO (first-in-first-out) Regel implementiert.
   Bei der FIFO Regel soll das am längsten in der Menge befindliche Objekt entfernt werden.

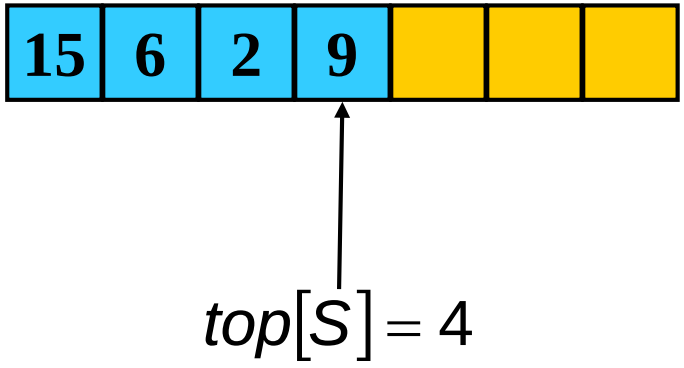


Stacks

- Einfügen eines Objekts wird bei Stacks Push genannt.
- Entfernen des zuletzt eingefügten Objekts wird Pop genannt.
- Zusätzliche Hilfsoperation ist Stack-Empty, die überprüft, ob ein Stack leer ist.
- Stack mit maximal n Objekten wird realisiert durch ein Array $S[1 \ldots n]$ mit einer zusätzlichen Variablen $top[S]$, die den Index des zuletzt eingefügten Objekts speichert.
- Maximale Anzahl Objekte a priori nicht bekannt: verwende dynamisches Array für S.

Stack - Beispiel

- Objekte sind hier natürliche Zahlen.
- Stack kann dann wie folgt aussehen:



Stack - Operationen

Stack - Empty(S)

1. **if** $top[S] = 0$
2. **then return** TRUE
3. **else return** FALSE

Push(S, x)

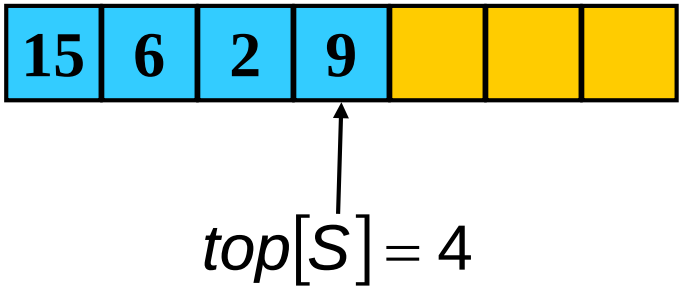
1. $top[S] \leftarrow top[S]+1$
2. $S[top[S]] \leftarrow x$

Pop(S)

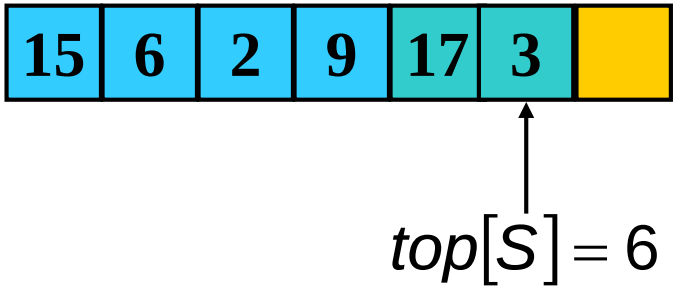
1. **if** Stack – Empty(S)
2. **then error** "underflow"
3. **else** $top[S] \leftarrow top[S]-1$
4. **return** $S[top[S]+1]$

Satz 10.8: Mit Stacks kann die LIFO Regel in Zeit **O**(1) ausgeführt werden.

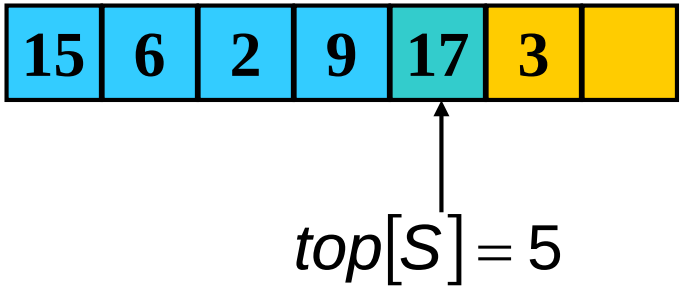
Illustration der Stackoperationen



Nach Push(*S*,17), Push(*S*,3):



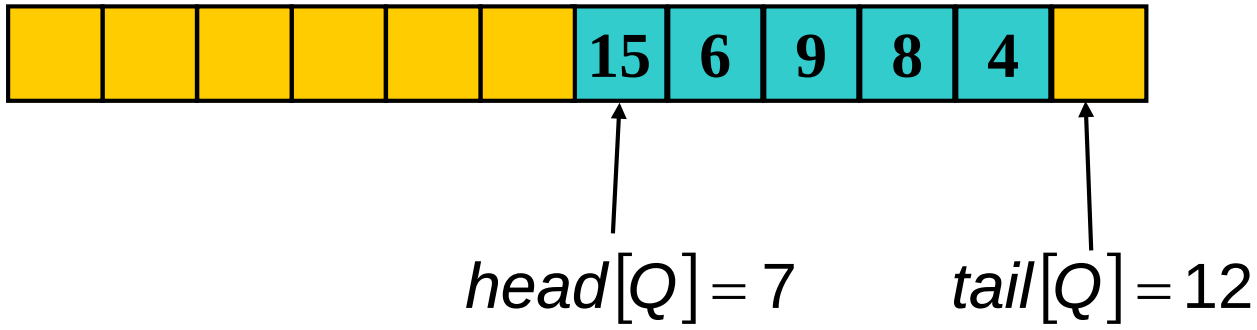
Nach Pop(*S*):

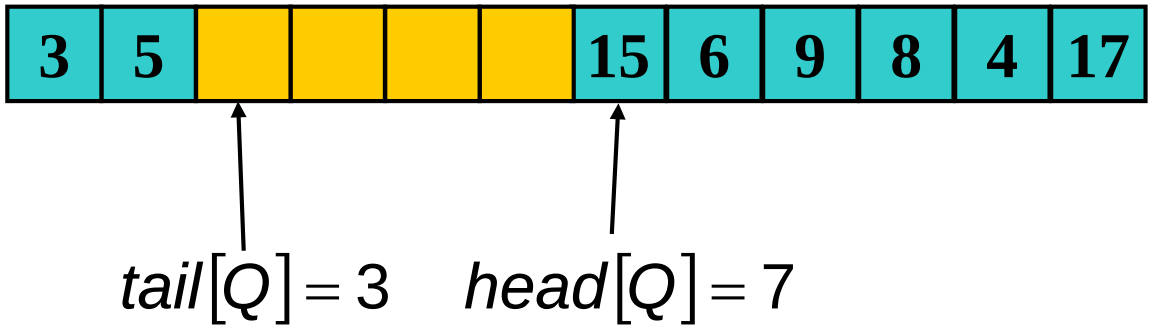


Queues

- Einfügen eines Objekts wird Enqueue genannt.
- Entfernen des am längsten in der Queue befindlichen Objekts wird Dequeue genannt.
- Zusätzliche Hilfsoperation ist Queue-Empty, die überprüft, ob eine Queue leer ist.
- Queue mit maximal *n-1* Objekten wird realisiert durch ein Feld *Q*[*1…n*] mit zusätzlichen Variablen *head*[*Q*], *tail*[*Q*].
  (Maximum nicht bekannt: verwende dynamisches Feld.)
- *head*[*Q*]: Position des am längsten in Queue befindlichen Objekts
- *tail*[*Q*]: erste freie Position.
- Alle Indexberechnungen modulo *n* (+1), betrachten Array kreisförmig. Auf Position *n* folgt wieder Position 1.

Queue - Beispiele





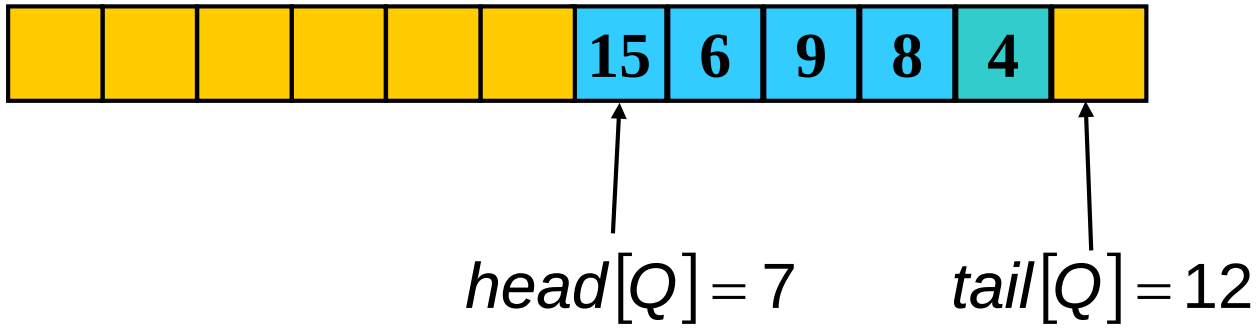
Queue - Operationen

```
Enqueue(Q,x)
Q[tail[Q]] ← x
if tail[Q] = length[Q] then
    then teil[Q] ← 1
    else tail[Q] ← tail[Q]+1
```

```
Queue-Empty(Q)
if head[Q]=tail[Q]
    then return TRUE
    else return FALSE
```

```
Dequeue(Q,x)
if Queue-Empty(Q) then error “underflow”
else
    x ← Q[head[Q]]
    if head[Q]=length[Q]
        then head[Q] ← 1
        else head[Q] ← head[Q]+1
    return x
```

Illustration Enqueue



Nach Enqueue(Q,17), Enqueue(Q,3), Enqueue(Q,5):

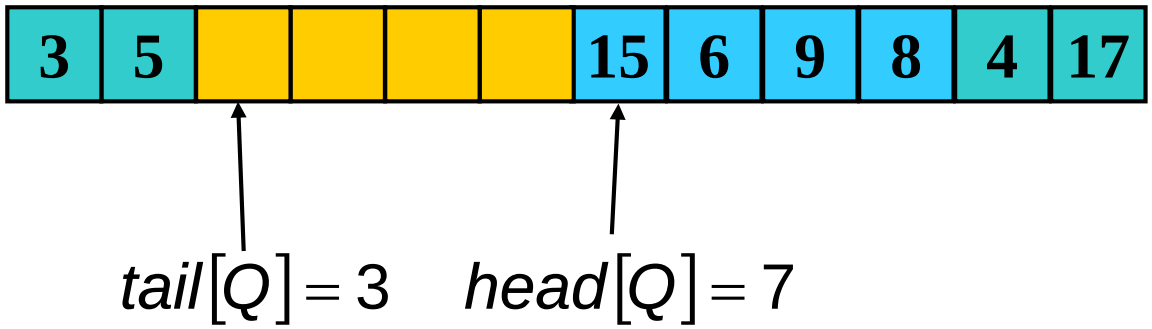
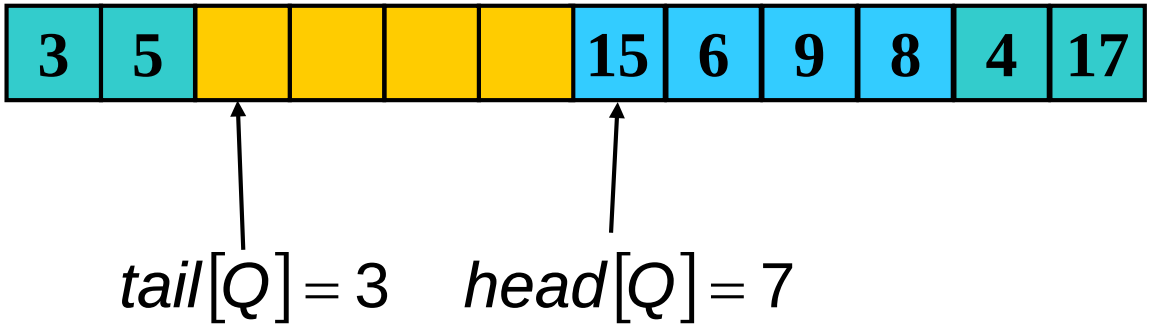
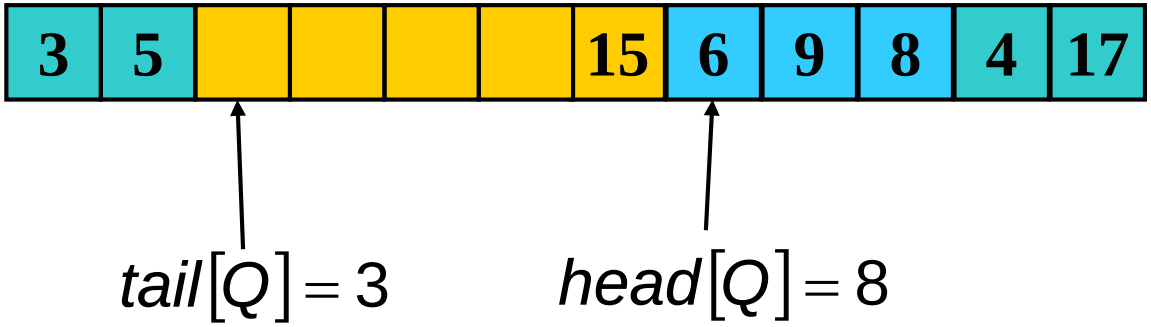


Illustration Dequeue



Nach Dequeue:



Laufzeit Queue-Operationen

Satz 10.9: Mit Queues kann die FIFO Regel in Zeit **O**(1) ausgeführt werden.

Problem: ein statisches Feld kann nur begrenzt viele Elemente speichern

Lösungsmöglichkeiten:

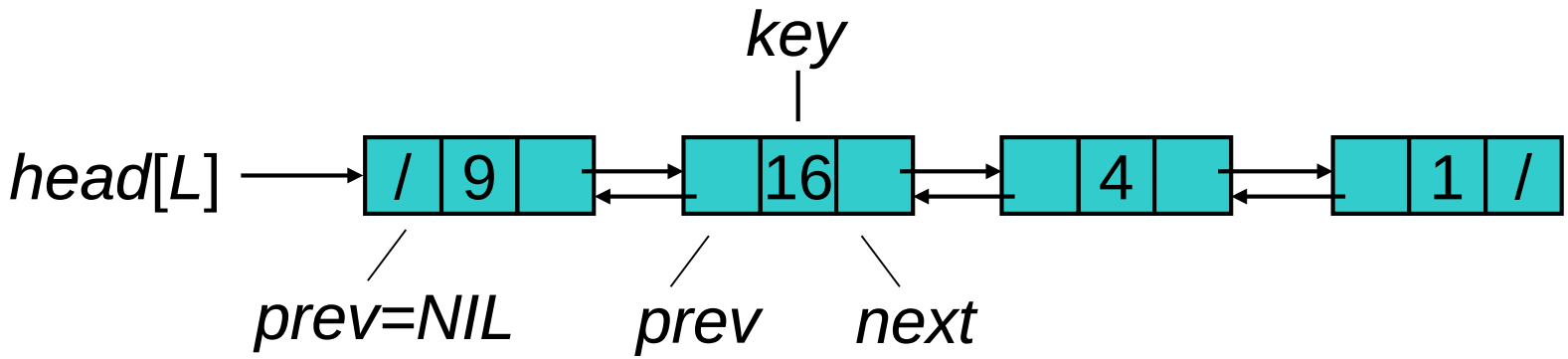
- Verwende ein dynamisches Feld wie oben angegeben. Dann sind die amortisierten Laufzeiten der Operationen nach wie vor konstant.
- Verwende Zeigerstrukturen.

Objekte, Referenzen, Zeiger

- Zugriff auf Objekte erfolgt in der Regel durch Referenzen oder Verweise auf Objekte wie in Java.
- In Sprachen wie C und C++ realisiert durch Zeiger, engl. Pointer.
- Zeiger/Pointer Notation aus Introduction to Algorithms.
- Verwenden Referenzen, Zeiger, Verweise synonym.
- Verweise zeigen oder verweisen oder referenzieren auf Objekte.

Doppelt verkettete Listen

- Verkettete Listen bestehen aus einer Menge von Objekten, die über Verweise linear verkettet sind.
- Objekte in doppelt verketteter Liste *L* besitzen mindestens drei Felder: *key, next, prev*. Außerdem sind Felder für Satellitendaten möglich.
- Zugriff auf Liste *L* durch Verweis/Zeiger *head*[*L*].



Doppelt verkettete Listen

- Verkettete Listen bestehen aus einer Menge von Objekten, die über Verweise linear verkettet sind.

- Objekte in doppelt verketteter Liste *L* besitzen mindestens drei Felder: *key, next, prev*. Außerdem sind Felder für Satellitendaten möglich.

- Zugriff auf Liste *L* durch Verweis/Zeiger *head*[*L*].

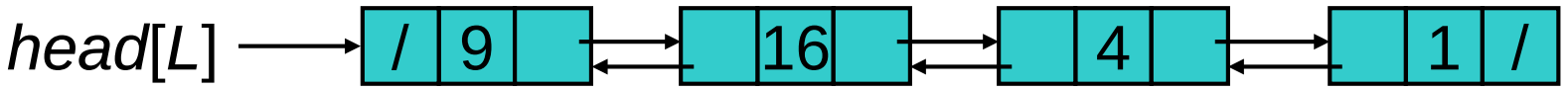
- Können dynamische Mengen mit den Operationen Insert, Remove, Search, Search-Minimum, usw unterstützen. Aber Unterstützung ist nicht unbedingt effizient.

Doppelt verkettete Listen

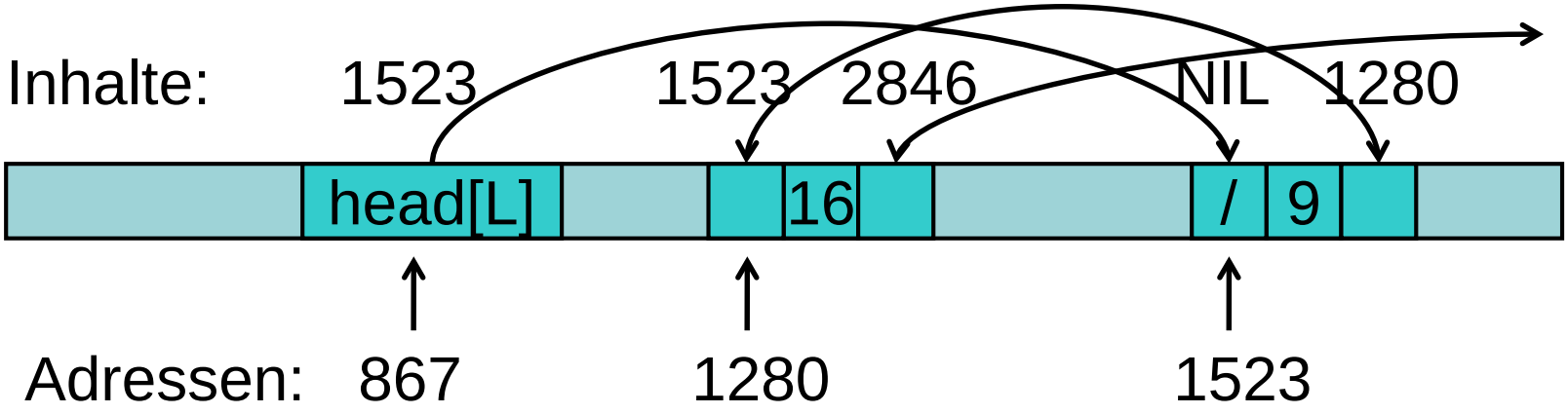
- *head*[*L*] verweist auf erstes Element der Liste *L*.
- *x* Objekt in Liste *L*: *next*[*x*] verweist auf nächstes Element in Liste, *prev*[*x*] verweist auf voriges Element in Liste.
- *prev*[*x*]=NIL: *x* besitzt keinen Vorgänger. Dann ist *x* erstes Element der Liste und *head*[*L*] verweist auf *x*.
- *next*[*x*]=NIL: *x* besitzt keinen Nachfolger. Dann ist x letztes Element der Liste.
- *head*[*L*]=NIL: Liste *L* ist leer.

Interne Darstellung

Abstrakt: verkettete Liste

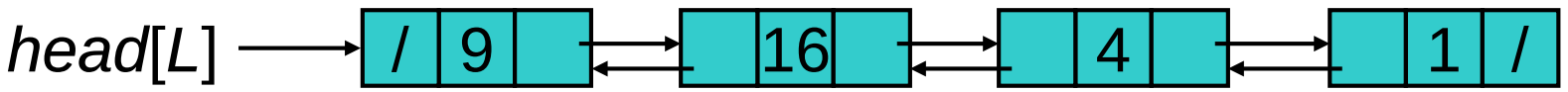


Intern: linear adressierbarer Speicher

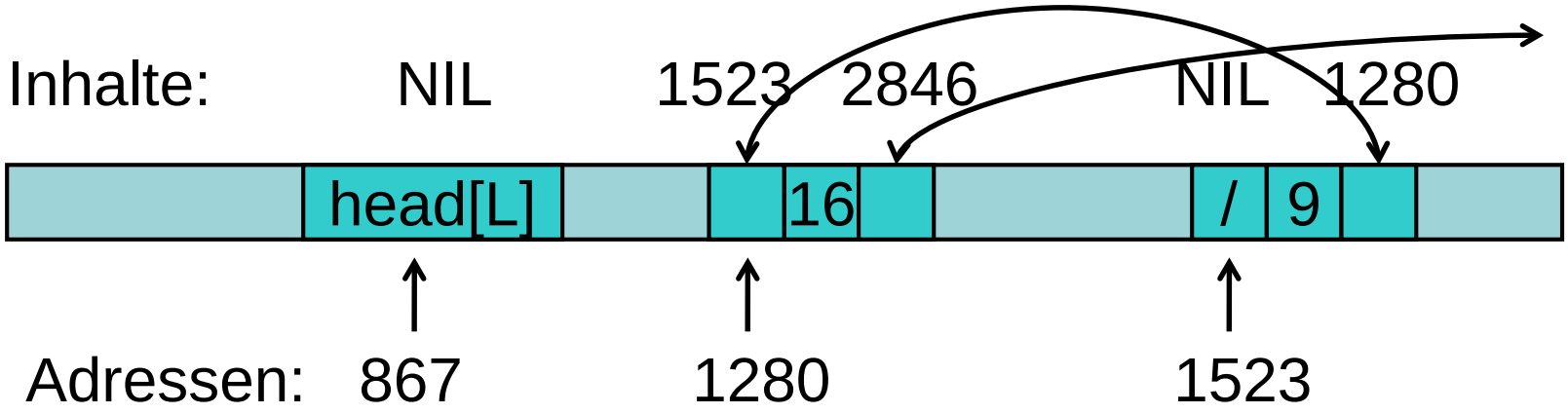


Interne Darstellung

Abstrakt: verkettete Liste

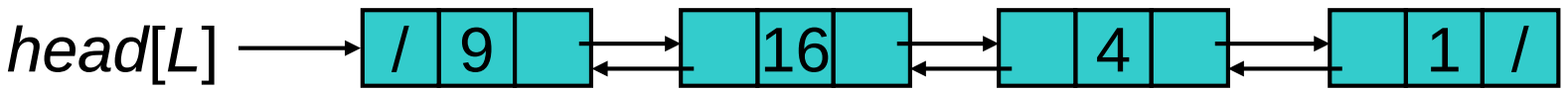


head[L] ← NIL: Liste noch da, aber nicht mehr über head[L] erreichbar.

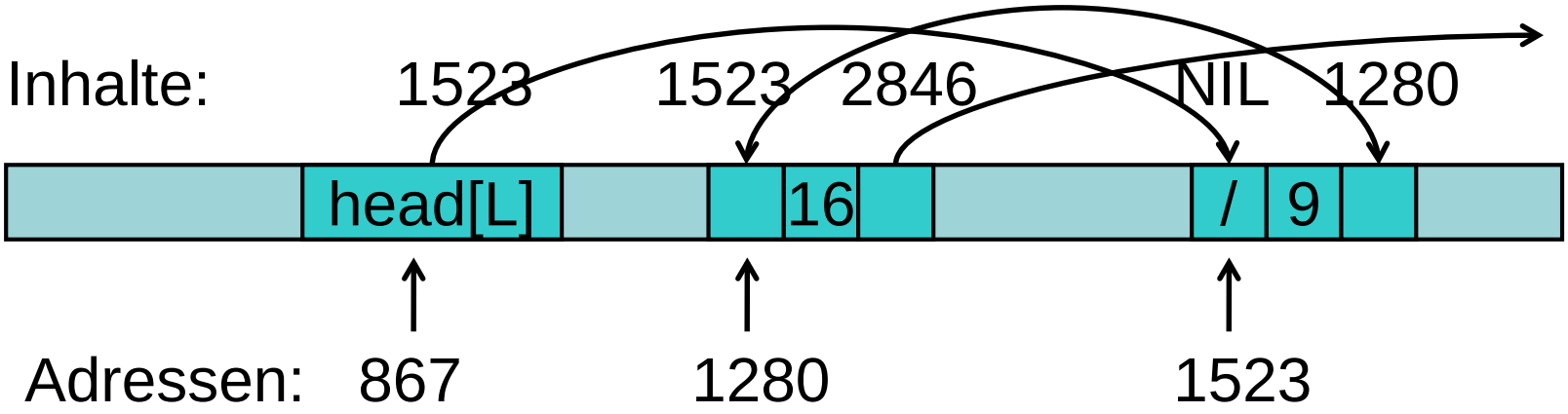


Interne Darstellung

Abstrakt: verkettete Liste

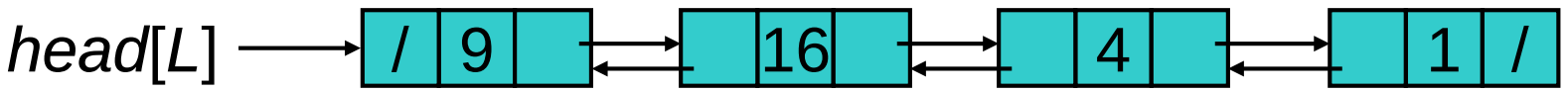


delete head[L]: Speicherplatz des Elements, auf das head[L] zeigt, wird freigegeben.

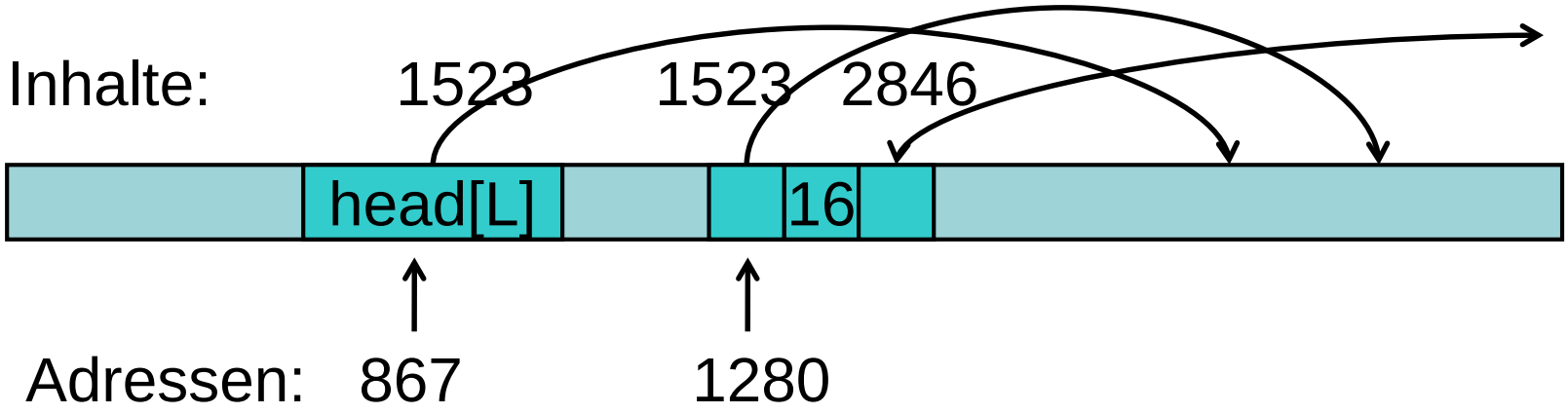


Interne Darstellung

Abstrakt: verkettete Liste

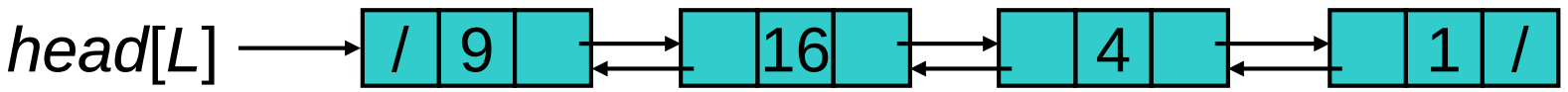


delete head[L]: Speicherplatz des Elements, auf das head[L] zeigt, wird freigegeben.

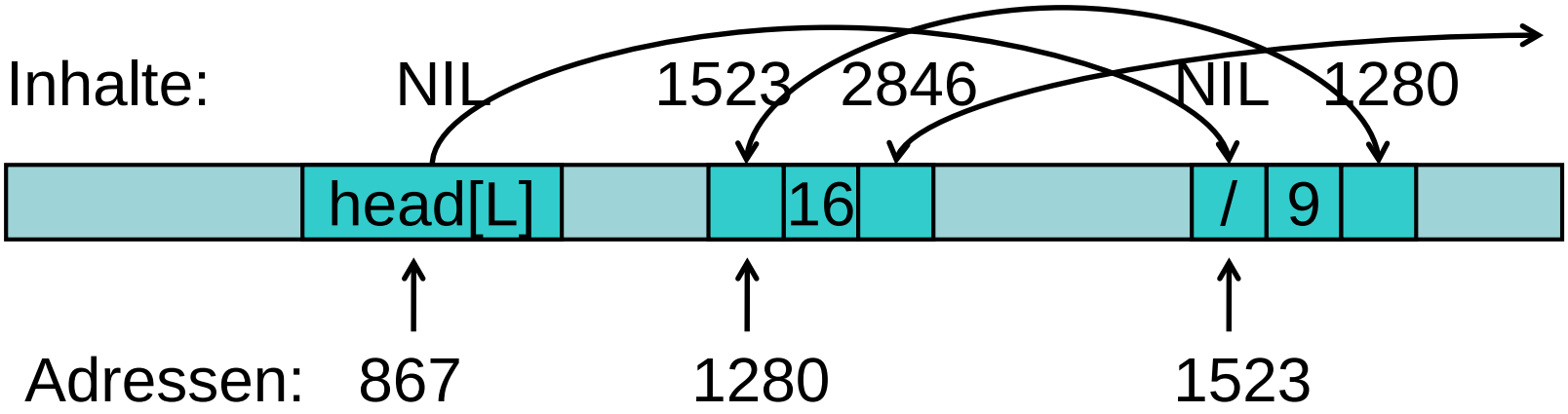


Interne Darstellung

Abstrakt: verkettete Liste

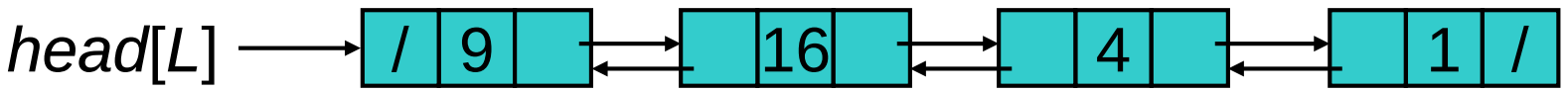


delete next[head[L]]: Speicherplatz des Elements, auf das next[head[L]] zeigt, wird freigegeben.

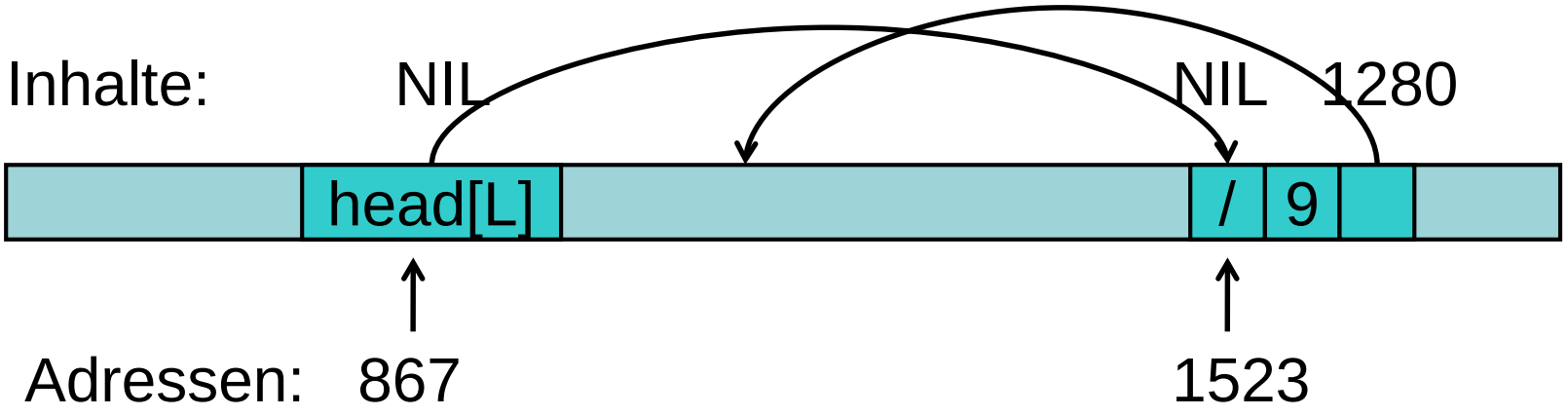


Interne Darstellung

Abstrakt: verkettete Liste

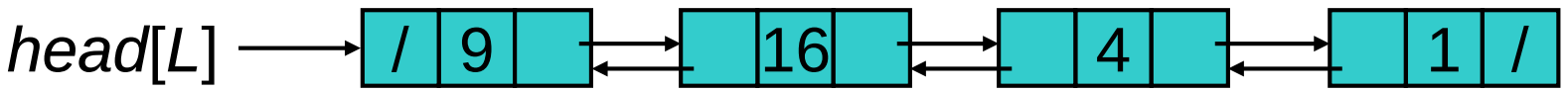


delete next[head[L]]: Speicherplatz des Elements, auf das next[head[L]] zeigt, wird freigegeben.

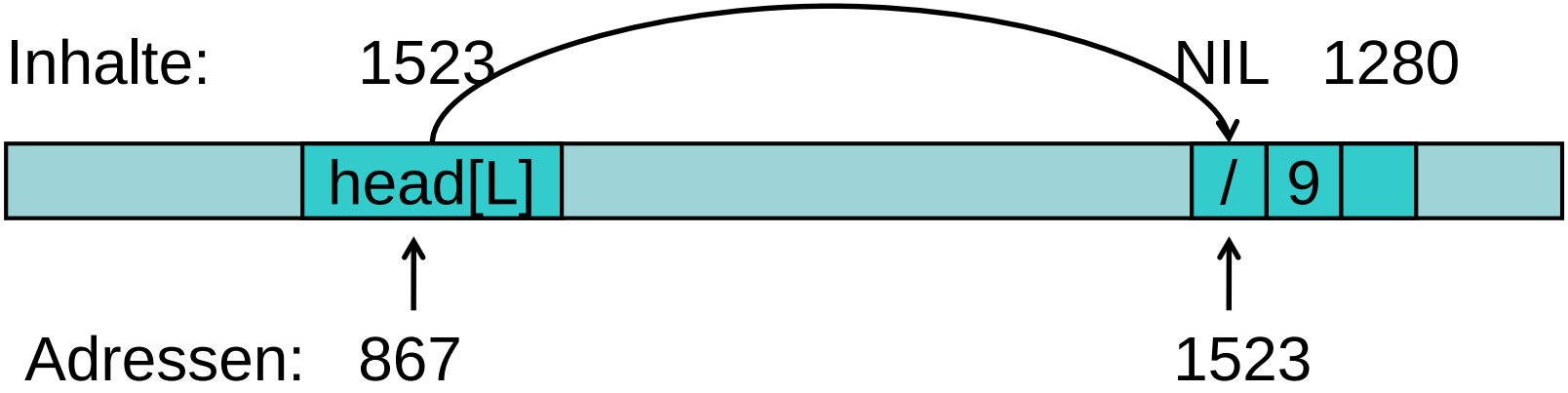


Interne Darstellung

Abstrakt: verkettete Liste

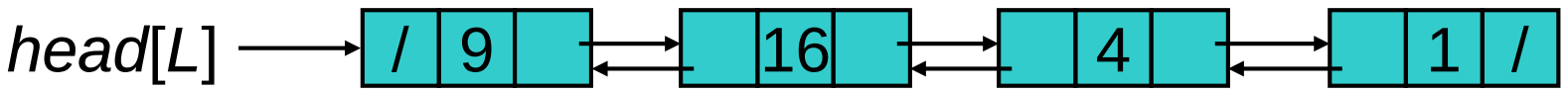


new head[L]: Speicherplatz für neues Element wird angelegt und head[L] darauf verwiesen.

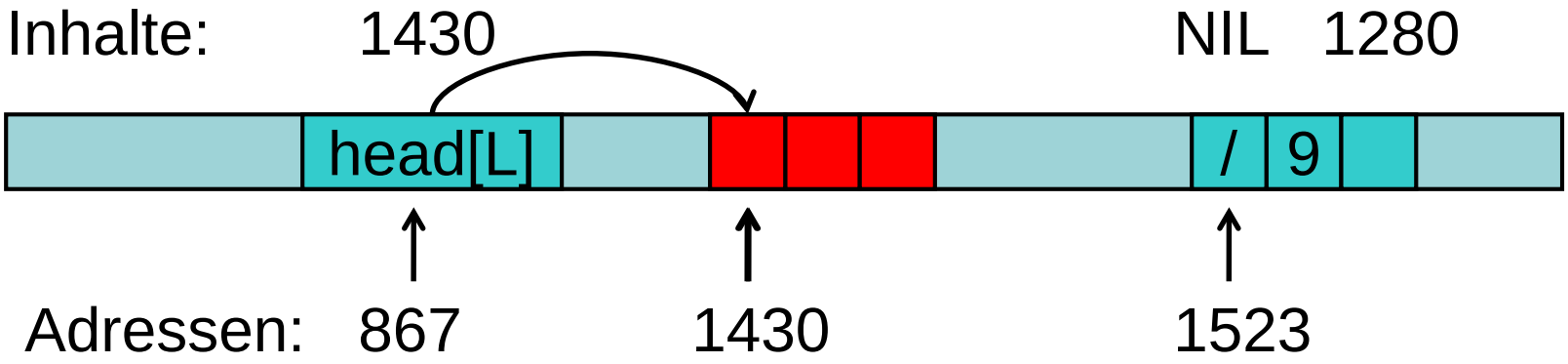


Interne Darstellung

Abstrakt: verkettete Liste

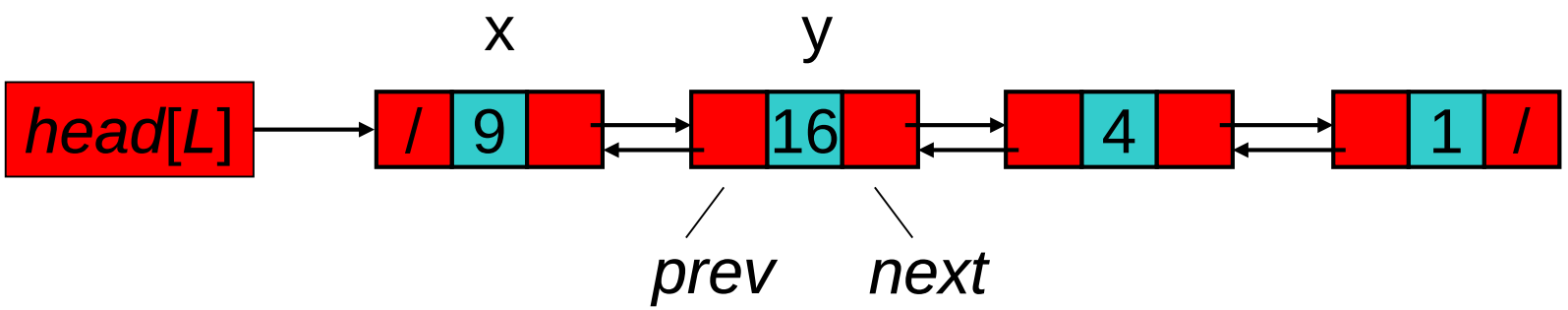


new head[L]: Speicherplatz für neues Element wird angelegt und head[L] darauf verwiesen.



Zeigervariablen

Rot: Zeigervariablen



Zeigervariablen sind wie Schattenvariablen, d.h. sie stehen stellvertretend für die Objekte, auf die sie zeigen.

- key[head[L]]: 9
- next[head[L]]: y
- prev[next[head[L]]]: x

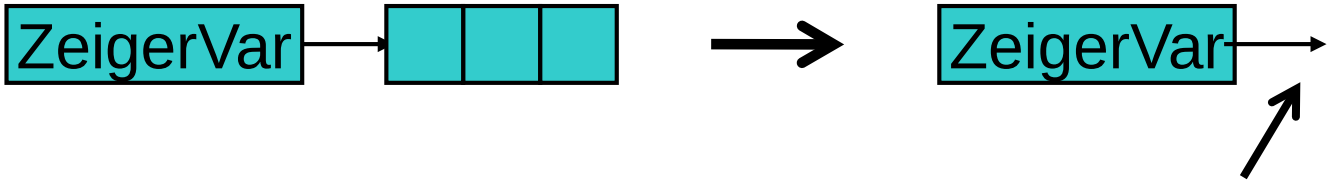
Allokation und Deallokation von Speicher

Befehle: new ZeigerVar, delete ZeigerVar

new ZeigerVar:

delete ZeigerVar:

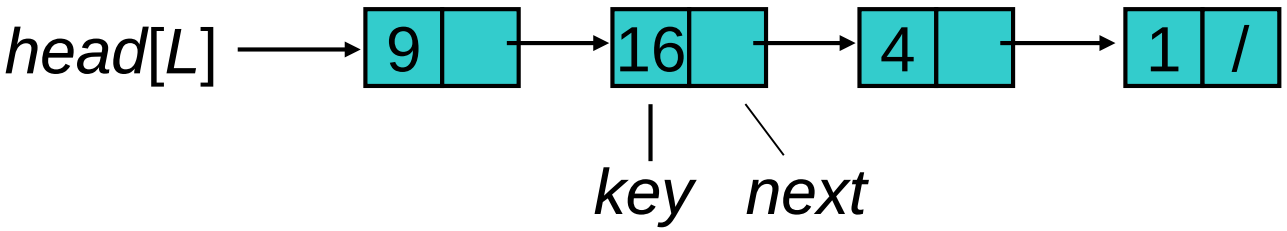
Größe hängt vom Typ ab, auf den ZeigerVar zeigen soll



Vorsicht! Adresse noch in ZeigerVar, aber Speicher für Objekt wieder freigegeben.

Varianten verketteter Listen

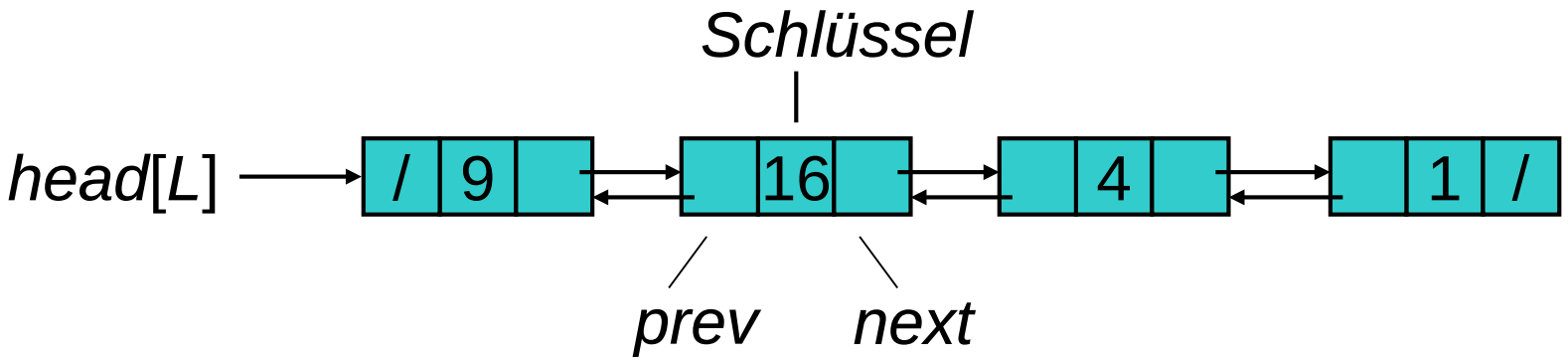
- Einfach verkettete Listen: Feld *prev* nicht vorhanden.



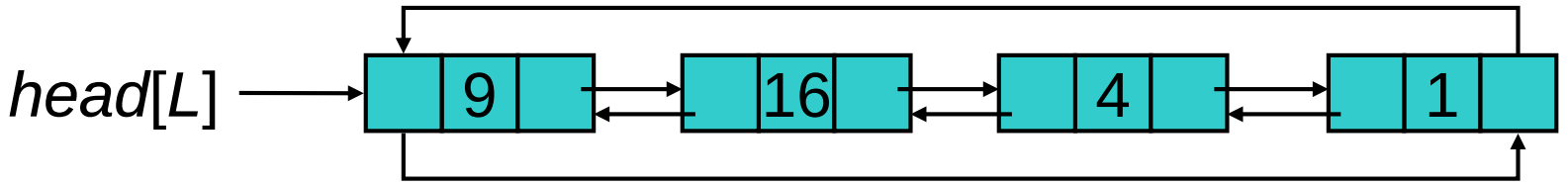
- Sortierte verkettete Liste: Reihenfolge in Liste entspricht sortierter Reihenfolge der Schlüssel.

- zykisch/kreisförmig verkettete Listen: *next* des letzten Objekts zeigt auf erstes Objekt. *prev* des ersten Objekts zeigt auf letztes Objekt.

Doppelt verkettete Listen - Beispiel



zyklisch doppelt verkette Liste:

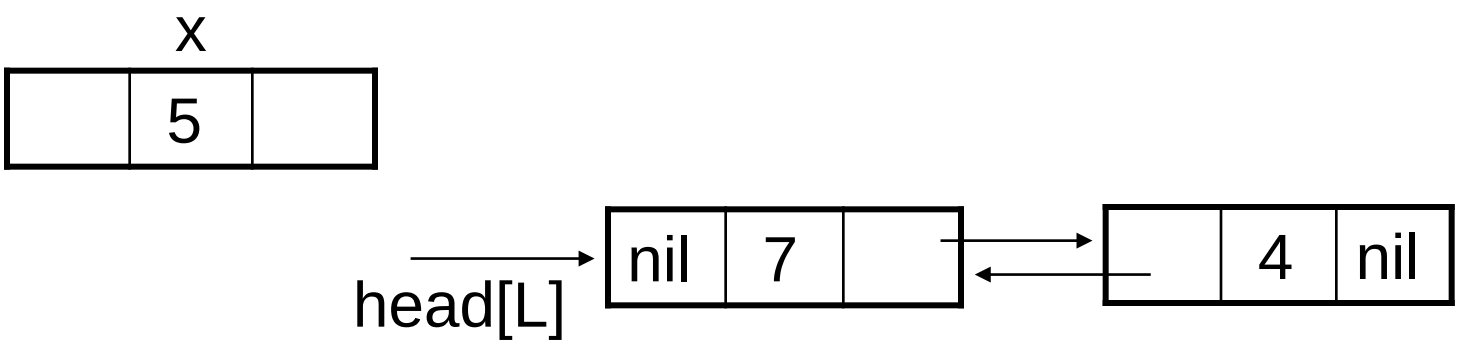


Doppelt verkettete Listen - Beispiel

List-Insert(L,x)

1. next[x] ← head[L]
2. **if** head[L] ≠ **nil then** prev[head[L]] ← x
3. head[L] ← x
4. prev[x] ← **nil**

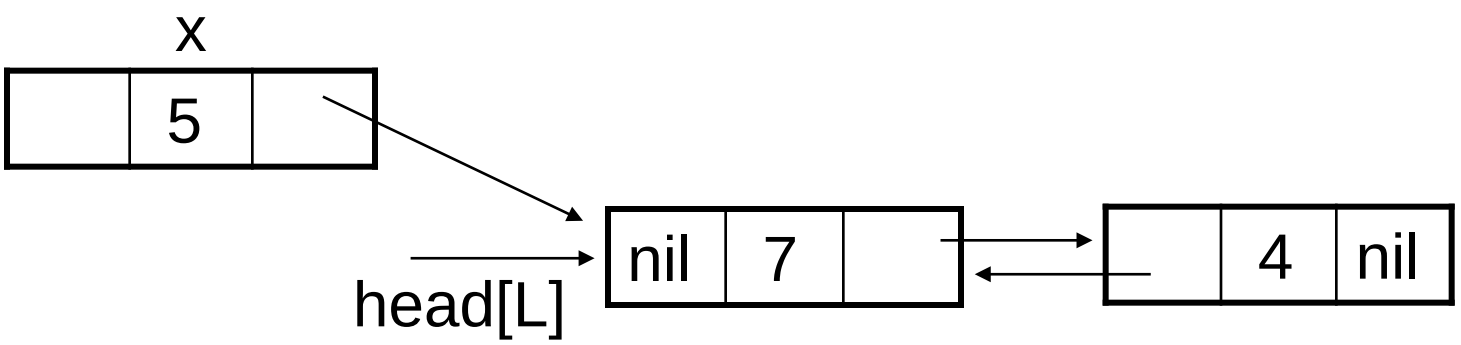
x: Zeigervariable, die Adresse des einzufügenden Elements enthält.



Doppelt verkettete Listen - Beispiel

List-Insert(L,x)

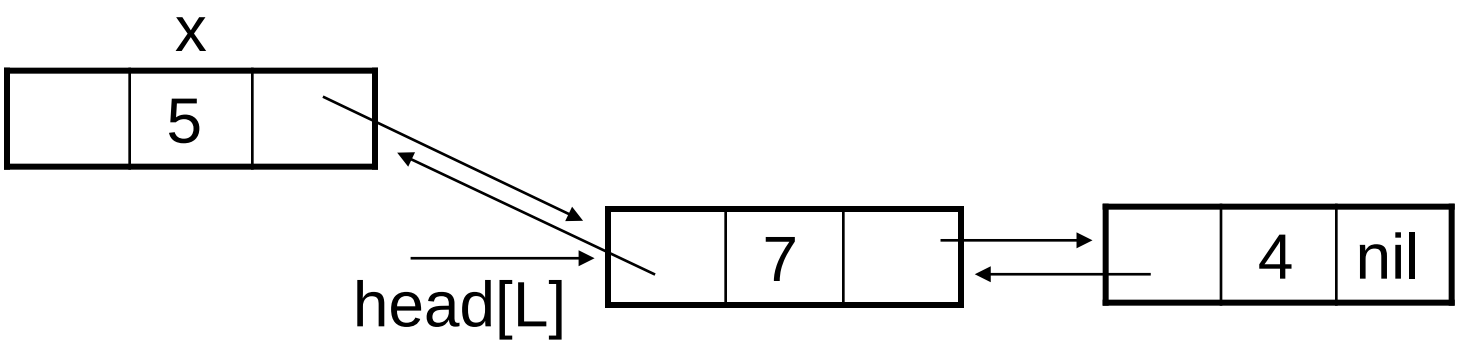
1. next[x] ← head[L]
2. **if** head[L] ≠ **nil then** prev[head[L]] ← x
3. head[L] ← x
4. prev[x] ← **nil**



Doppelt verkettete Listen - Beispiel

List-Insert(L,x)

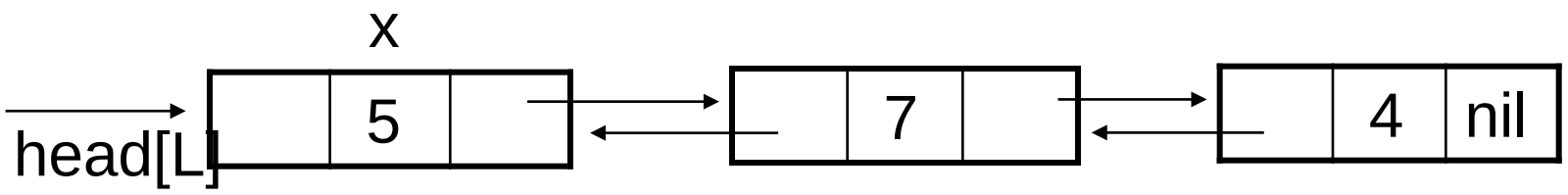
1. next[x] ← head[L]
2. **if** head[L] ≠ **nil then** prev[head[L]] ← x
3. head[L] ← x
4. prev[x] ← **nil**



Doppelt verkettete Listen - Beispiel

List-Insert(L,x)

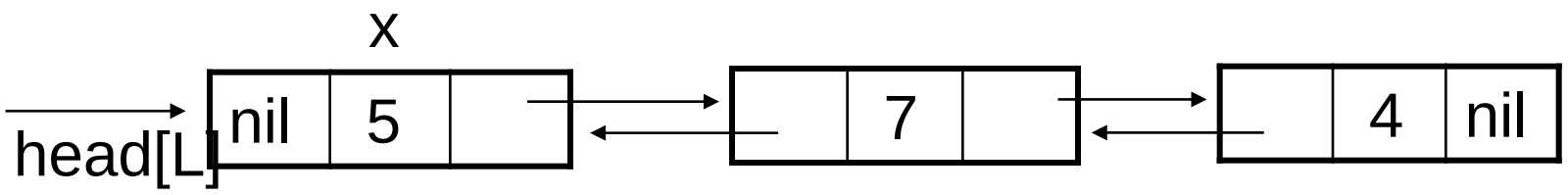
1. next[x] ← head[L]
2. **if** head[L] ≠ **nil then** prev[head[L]] ← x
3. head[L] ← x
4. prev[x] ← **nil**



Doppelt verkettete Listen - Beispiel

List-Insert(L,x)

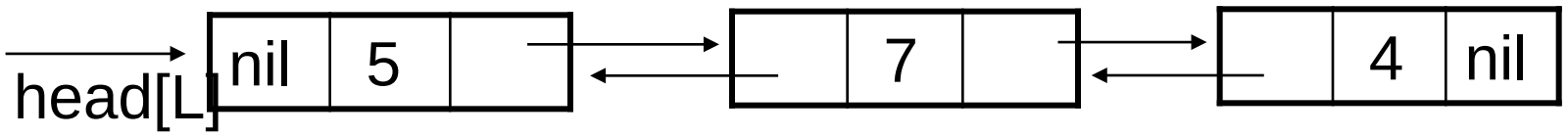
1. next[x] $\leftarrow$ head[L]
2. **if** head[L] $\neq$ **nil then** prev[head[L]] $\leftarrow$ x
3. head[L] $\leftarrow$ x
4. prev[x] $\leftarrow$ **nil**



Doppelt verkettete Listen - Beispiel

List-Remove(L,x)

1. **if** prev[x] ≠ **nil then** next[prev[x]] ← next[x]
2. **else** head[L] ← next[x]
3. **if** next[x] ≠ **nil then** prev[next[x]] ← prev[x]

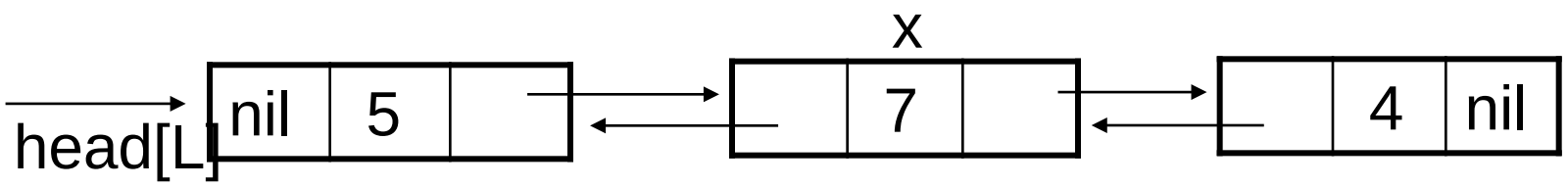


Doppelt verkettete Listen - Beispiel

List-Remove(L,x)

x: zeigt auf zu löschendes Objekt

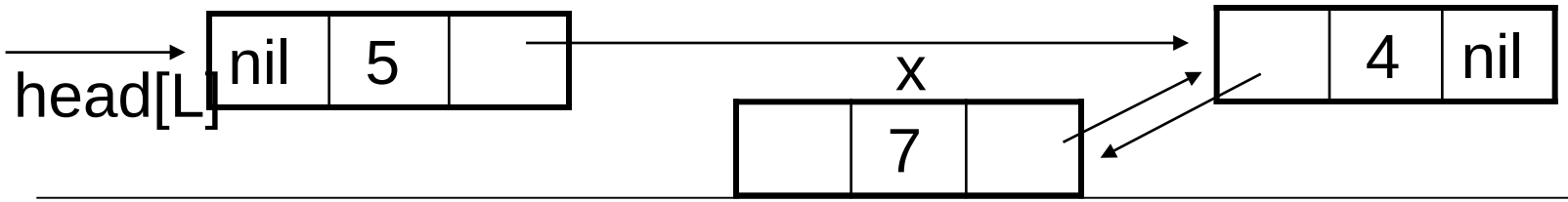
1. **if** prev[x] ≠ **nil then** next[prev[x]] ← next[x]
2. **else** head[L] ← next[x]
3. **if** next[x] ≠ **nil then** prev[next[x]] ← prev[x]



Doppelt verkettete Listen - Beispiel

List-Remove(L,x)

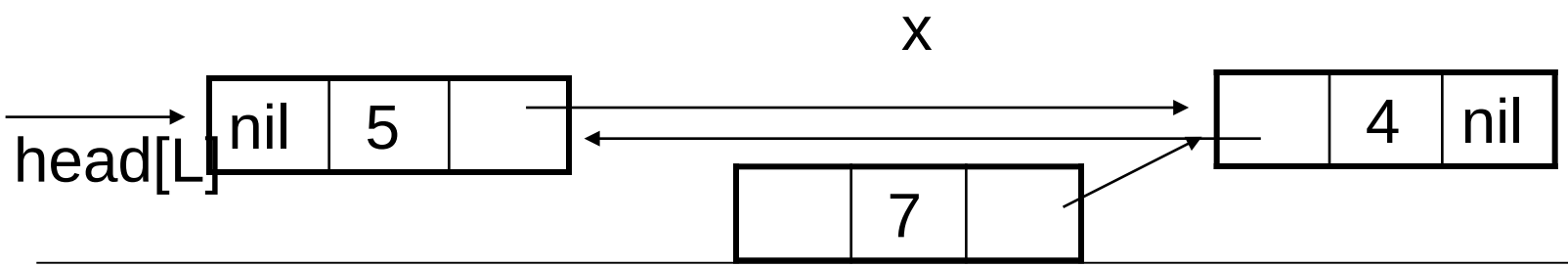
1. **if** prev[x] ≠ **nil then** next[prev[x]] ← next[x]
2. **else** head[L] ← next[x]
3. **if** next[x] ≠ **nil then** prev[next[x]] ← prev[x]



Doppelt verkettete Listen - Beispiel

List-Remove(L,x)

1. **if** prev[x] ≠ **nil then** next[prev[x]] ← next[x]
2. **else** head[L] ← next[x]
3. **if** next[x] ≠ **nil then** prev[next[x]] ← prev[x]



Doppelt verkettete Listen - Beispiel

Korrektheit von List-Insert und List-Remove:

Weise nach, dass die folgende Invariante nach jeder Operation für die aktuelle Menge $M=\{o_1,\ldots,o_n\}$ der Objekte in der Liste gilt:

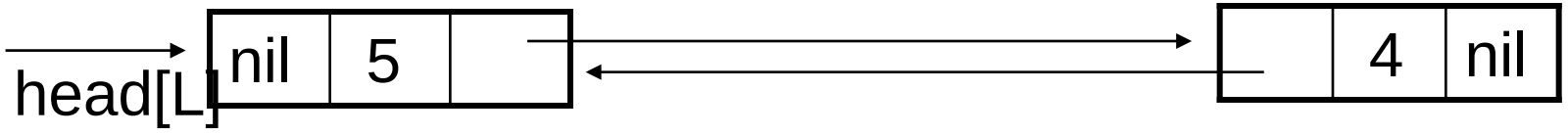
Es gibt eine Permutation π auf $\{1,\ldots,n\}$, so dass

- $head[L] = o_{\pi(1)}$
- $prev[o_{\pi(1)}]=NIL$ und $prev[o_{\pi(i)}]=o_{\pi(i-1)}$ für alle $i>1$
- $next[o_{\pi(i)}]=o_{\pi(i+1)}$ für alle $i<n$ und $next[o_{\pi(n)}]=NIL$

Doppelt verkettete Listen - Beispiel

List-Search(L,k)

1. x ← head[L]
2. **while** x≠nil and key[x]≠k **do**
3. x ← next[x]
4. **return x**

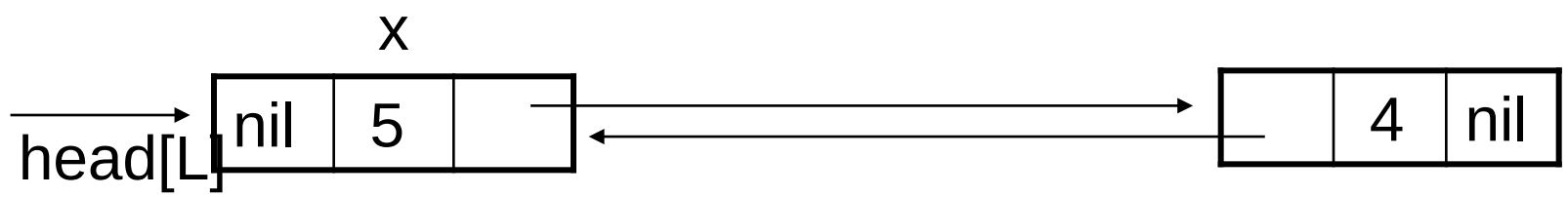


Doppelt verkettete Listen - Beispiel

List-Search(L,k)

1. x ← head[L]
2. **while** x≠nil and key[x]≠k **do**
3. x ← next[x]
4. **return x**

List-Search(L,4)

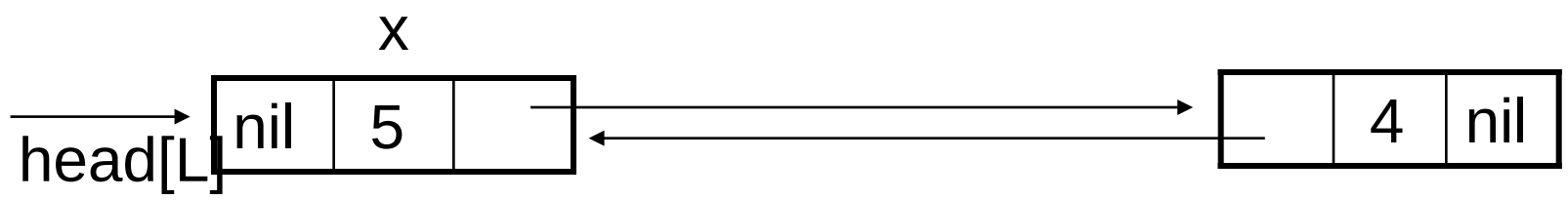


Doppelt verkettete Listen - Beispiel

List-Search(L,k)

1. x ← head[L]
2. **while** x≠nil and key[x]≠k **do**
3. x ← next[x]
4. **return x**

List-Search(L,4)

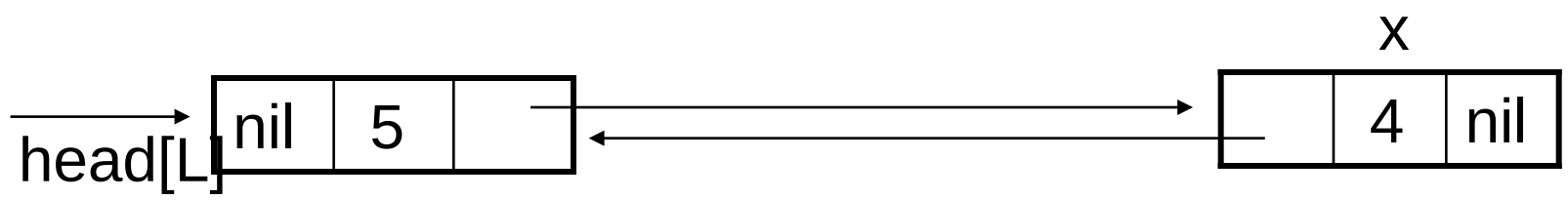


Doppelt verkettete Listen - Beispiel

List-Search(L,k)

1. x ← head[L]
2. **while** x≠nil and key[x]≠k **do**
3. x ← next[x]
4. **return x**

List-Search(L,4)

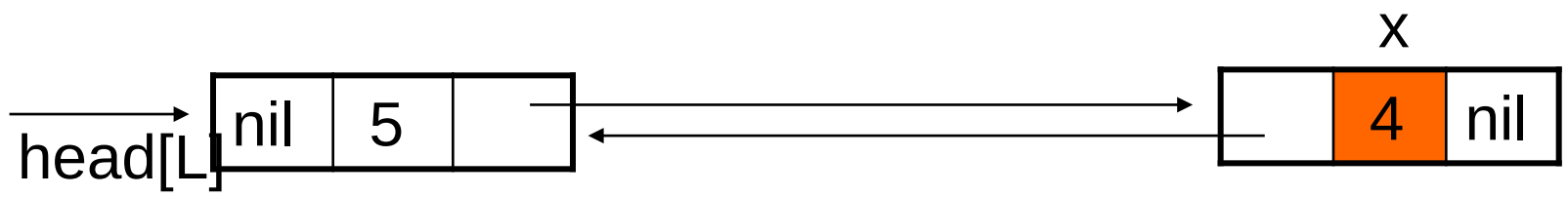


Doppelt verkettete Listen - Beispiel

List-Search(L,k)

1. x ← head[L]
2. **while** x≠nil and key[x]≠k **do**
3. x ← next[x]
4. **return x**

List-Search(L,4)

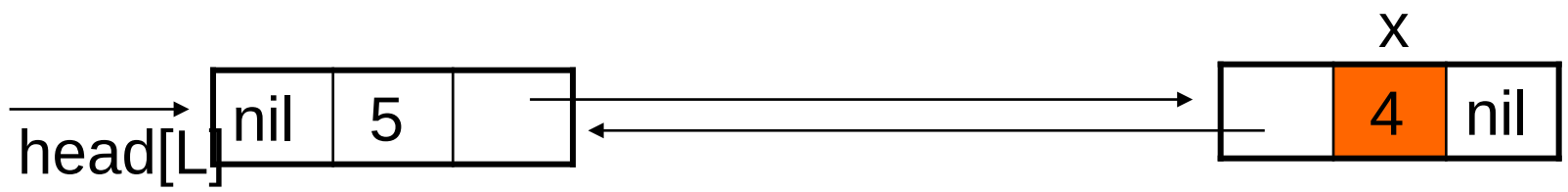


Doppelt verkettete Listen - Beispiel

List-Search(L,k)

1. x ← head[L]
2. **while** x≠nil and key[x]≠k **do**
3. x ← next[x]
4. **return x**

List-Search(L,4)



Einfügen in verkettete Liste

List-Insert(L,x)

1. next[x] $\leftarrow$ head[L]
2. **if** head[L] $\neq$ **nil then** prev[head[L]] $\leftarrow$ x
3. head[L] $\leftarrow$ x
4. prev[x] $\leftarrow$ **nil**

Lemma 10.10: List-Insert erfordert Zeit $\Theta(1)$.

Löschen aus verketteter Liste

List-Remove(L,x)

1. **if** prev[x] $\neq$ **nil then** next[prev[x]] $\leftarrow$ next[x]
2. **else** head[L] $\leftarrow$ next[x]
3. **if** next[x] $\neq$ **nil then** prev[next[x]] $\leftarrow$ prev[x]

Lemma 10.11: List-Remove erfordert Zeit $\Theta(1)$.

Durchsuchen einer verketteten Liste

List-Search(L,k)
1. $x \leftarrow head[L]$
2. **while** $x \neq NIL \ \wedge \ key[x] \neq k$
3. **do** $x \leftarrow next[x]$
4. **return** x

Lemma 10.12: List-Search erfordert bei einer Liste mit n Elementen Zeit $\Theta(n)$.

Beweis: Verwende Invariante für die Liste und als Potenzialfunktion die Ordnung des Elements in der Liste.

Doppelt verkettete Listen - Beispiel

Datenstruktur Liste:

- Platzbedarf $\Theta(n)$
- Laufzeit Suche: $\Theta(n)$
- Laufzeit Einfügen/Löschen: $\Theta(1)$

Vorteile:

- Schnelles Einfügen/Löschen
- O(n) Speicherbedarf

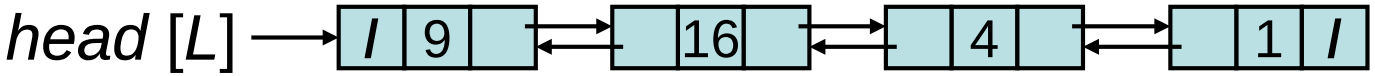
Nachteile:

- Hohe Laufzeit für Suche

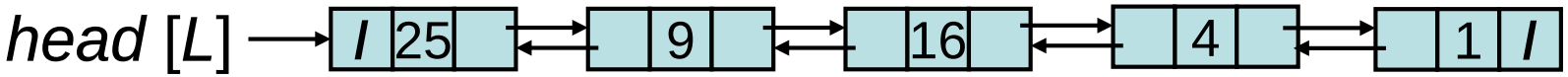
Vereinfachung durch Sentinels (Wächter)

- Vermeidung von Abfragen *head*[L] ≠ NIL und x ≠ NIL durch Einfügen eines zusätzlichen *dummy* Objekts.
- *dummy* besitzt drei Felder *key, prev, next*, aber *key*[*dummy*]= NIL.
- *next*[*dummy*] verweist auf erstes (richtiges) Objekt in Liste.
- *prev*[*dummy*] verweist auf letztes Objekt in Liste.
- Wir ersetzen in Pseudocode NIL durch *dummy*.
- *head*[*L*] zeigt jetzt auf dummy Objekt.
- Dummy Objekt zur Vereinfachung von Randbedingungen wird Sentinel oder Wächter genannt.

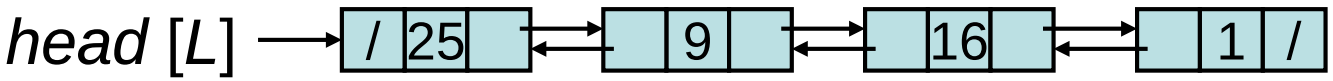
Liste ohne Sentinel



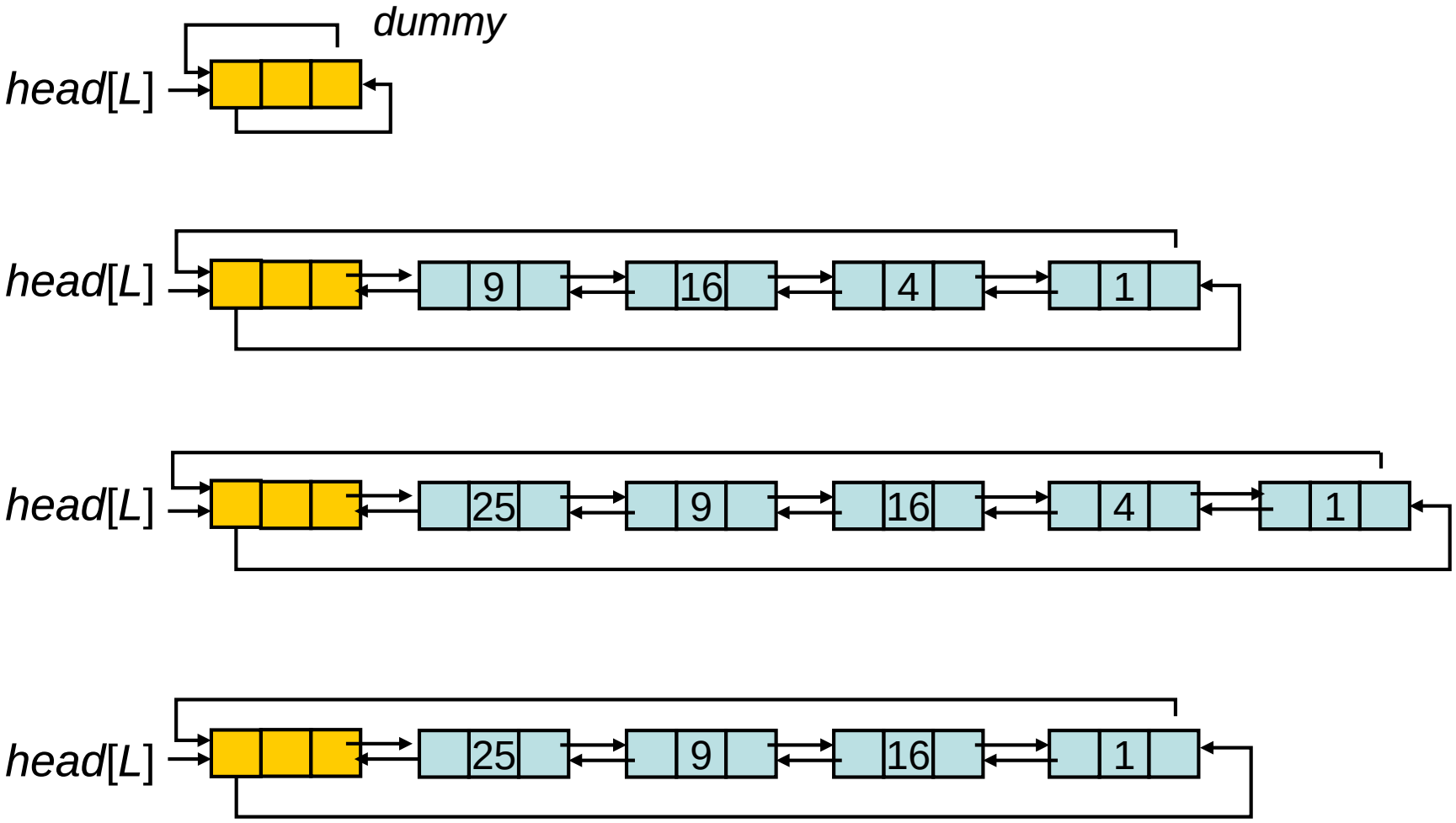
Nach List-Insert(*L,x*), wobei *key* [x]=25:



Nach List-Remove(*L,x*), wobei *key* [x]=4:



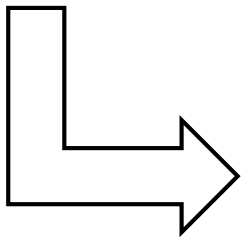
Zyklisch verkette Liste mit Sentinel



Durchsuchen verketteter Liste (mit Sentinel)

List-Search(L,k)

1. x ← head[L]
2. **while** x≠nil and key[x]≠k **do**
3. x ← next[x]
4. **return x**



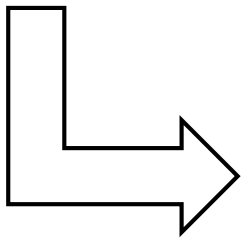
List-Search´(L,k)

1. x ← next[head[L]]
2. **while** x≠head[L] and key[x]≠k **do**
3. x ← next[x]
4. **return x**

Einfügen in verkettete Liste (mit Sentinel)

List-Insert(L,x)

1. next[x] $\leftarrow$ head[L]
2. **if** head[L] $\neq$ **nil then** prev[head[L]] $\leftarrow$ x
3. head[L] $\leftarrow$ x
4. prev[x] $\leftarrow$ **nil**



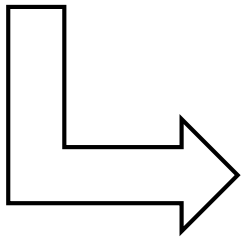
List-Insert´(L,x)

1. next[x] $\leftarrow$ next[head[L]]
2. prev[next[head[L]]] $\leftarrow$ x
3. next[head[L]] $\leftarrow$ x
4. prev[x] $\leftarrow$ head[L]

Löschen aus verketteter Liste (mit Sentinel)

List-Remove(L,x)

1. **if** prev[x] ≠ **nil then** next[prev[x]] ← next[x]
2. **else** head[L] ← next[x]
3. **if** next[x] ≠ **nil then** prev[next[x]] ← prev[x]



List-Remove´(L,x)

1. next[prev[x]] ← next[x]
2. prev[next[x]] ← prev[x]

Doppelt verkettete Listen

Datenstruktur Liste:

- Platzbedarf $\Theta(n)$
- Laufzeit Suche: $\Theta(n)$
- Laufzeit Einfügen/Löschen: $\Theta(1)$

Vorteile:

- Schnelles Einfügen/Löschen
- O(n) Speicherbedarf

Nachteile:

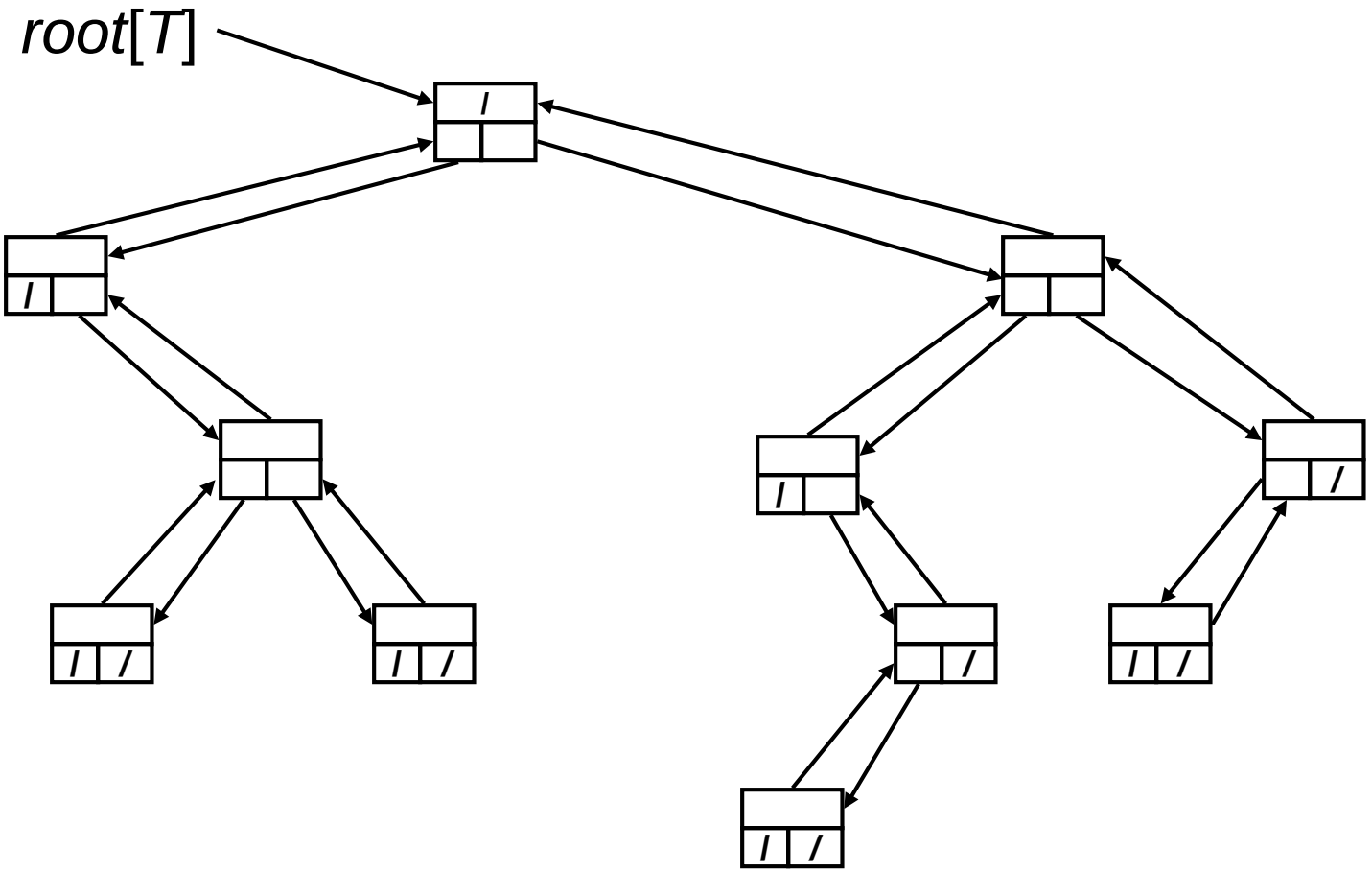
- Hohe Laufzeit für Suche

Besser: Verkettung der Elemente als Baum

Binäre Bäume

- Neben Felder für Schlüssel und Satellitendaten Felder *p*, *left*, *right*.
- x Knoten: *p*[*x*] Verweis auf Elternknoten, *left*[*x*] Verweis auf linkes Kind, *right*[*x*] Verweis auf rechtes Kind.
- Falls *p*[*x*]=NIL, dann ist *x* Wurzel des Baums.
- *left*[*x*] / *right*[*x*]=NIL: kein linkes/rechtes Kind.
- Zugriff auf Baum *T* durch Verweis *root*[*T*] auf Wurzelknoten.

Binäre Bäume - Illustration



Allgemeine Bäume

- Darstellung für binäre Bäume auch möglich für *k*-näre Bäume, *k* fest.
- Ersetze *left*[x]/*right*[*x*] durch *child1*[*x*],..,*childk*[*x*].
- Bei Bäumen mit unbeschränktem Grad nicht möglich, oder ineffizient, da Speicher für viele Felder reserviert werden muss.
- Nutzen linkes-Kind/rechtes-Geschwister- Darstellung.
- Feld *p* für Eltern bleibt. Dann Feld *left-child* für linkes Kind und Feld *right-sibling* für rechtes Geschwister.

Allgemeine Bäume - Illustration