Algorithmen und Datenstrukturen

Kapitel 4: Datenstrukturen

Prof. Dr. Peter Kling

Wintersemester 2020/21

Übersicht

1 Elementare Datenstrukturen

2 Binäre Suchbäume

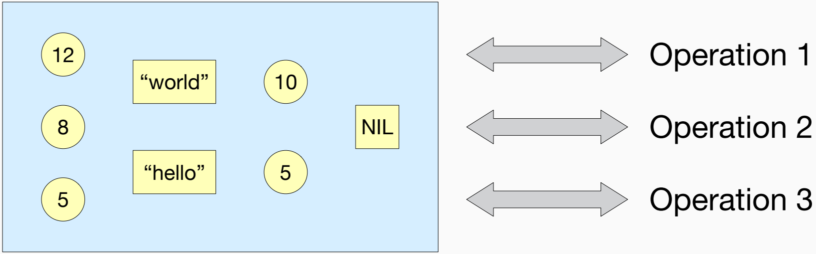
3 Balancierte Suchbäume

4 Hashing

1) Elementare Datenstrukturen

Definition 4.1

Eine Datenstruktur ist gegeben durch eine Menge von Objekten sowie eine Menge von Operationen auf diesen Objekten.



Definition 4.2

Ein Dictionary ist eine Datenstruktur, welche die Operationen INSERT (Einfügen), REMOVE (Entfernen) sowie SEARCH (Suchen) unterstützt.

Wörterbuch

Definition 4.3

Eine Priority Queue ist eine Datenstruktur, welche die Operationen INSERT (Einfügen), REMOVE (Entfernen) sowie SEARCHMIN (Suchen des Minimums) bzw. SEARCHMAX (Suchen des Maximums) unterstützt.

Prioritätswarteschlange

Ein grundlegendes Datenbankproblem

Speicherung und Verarbeitung von Datensätzen!

Beispiel

Verwalten von Kundendaten wie:

- Name, Adresse, Wohnort
- Kundennummer
- offene Bestellungen oder Rechnungen
- …

Anforderungen

- schneller Zugriff
- Einfügen neuer Datensätze
- Löschen bestehender Datensätze

Zugriff auf Objekte

- Objekte meist durch Schlüssel identifiziert
- Eingabe des Schlüssels liefert gewünschten Datensatz
- über den Schlüsseln gibt es eine totale Ordnung

Ver-
gleich-
barkeit

Beispiel

- Objekt Kundendaten (Name, Adresse, Kundennummer)
- Schlüssel: ~~Name~~ Kundennummer
- Totale Ordnung: ~~lexikographische Ordnung~~ „≤“

Typische elementare Operationen

- $\underline{\text{INSERT}(S, x)}$: Füge Objekt x in S ein.
- $\underline{\text{SEARCH}(S, k)}$: Finde Objekt x in S mit Schlüssel k. Falls kein solches Objekt in S existiert, gib NIL zurück.
- $\underline{\text{REMOVE}(S, x)}$: Entferne Objekt x aus S.
- $\underline{\text{SEARCHMIN}(S)}$: Finde das Objekt mit minimalem Schlüssel in S. Hierbei muss eine Ordnung auf den Schlüsseln existieren.
- $\underline{\text{SEARCHMAX}(S)}$: Finde das Objekt mit maximalem Schlüssel in S. Hierbei muss eine Ordnung auf den Schlüsseln existieren.

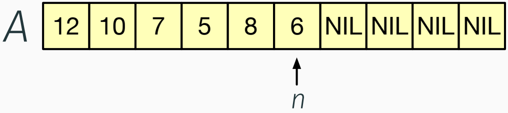
Eine einfache Datenstruktur: statisches Feld

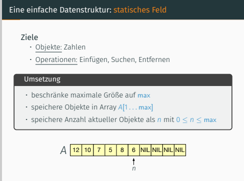
Ziele

- Objekte: Zahlen
- Operationen: Einfügen, Suchen, Entfernen

Umsetzung

- beschränke maximale Größe auf max
- speichere Objekte in Array $A[1 \dots \mathrm{max}]$
- speichere Anzahl aktueller Objekte als n mit $0 \leq n \leq \mathrm{max}$





- in diesem Beispiel sind die Schlüssel gleich den Objekten

Implementierung statischer Felder

INSERT(A, x)

if $n = \max$: **return** „Error: out of space“
$n \leftarrow n + 1$
$A[n] \leftarrow x$

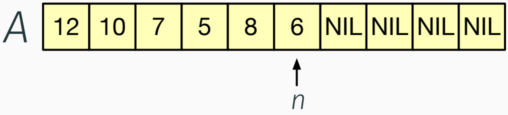
SEARCH(A, x)

for $i \leftarrow 1$ **to** n
    if $A[i] = x$: **return** i
return NIL

REMOVE(A, i)

$A[i] \leftarrow A[n]$
$A[n] \leftarrow$ NIL
$n \leftarrow n - 1$

REMOVE bekommt Index



Wie gut sind statische Felder?

Charakteristiken

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

Vorteile

- schnelles Einfügen
- schnelles Löschen

Nachteile

- Speicherbedarf hängt von max ab…
  …und ist nicht vorhersagbar
- hohe Laufzeit für Suche

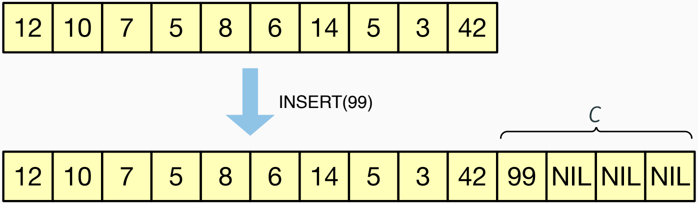
Dynamische Felder

- aktuell maximale Länge sei ℓ

length

Idee

Wenn Array A zu klein ($n > \ell$), generiere neues Array der Größe $\ell + c$ für feste Konstante c.



Ist das eine gute Implementierung eines dynamischen Feldes?

Überschlagsrechnung

- Zeitaufwand der Erweiterung ist $\Theta(\ell)$
- Zeitaufwand für n INSERT Operationen:
  - Aufwand $\Theta(\ell)$ für je c INSERT Operationen
    - also $\Theta(c)$ für die ersten c INSERT Operationen, …
    - …$\Theta(2c)$ für die zweiten c INSERT Operationen, …
    - …$\Theta(3c)$ für die dritten c INSERT Operationen, …
    - …
  - Gesamtaufwand:

$$\sum_{i=1}^{n/c} i \cdot c = \Theta(n^2)$$

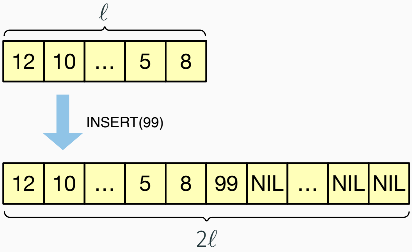
- Also durchschnittliche lineare Laufzeit für INSERT!

Das muss doch besser gehen!?!

Bessere dynamische Felder

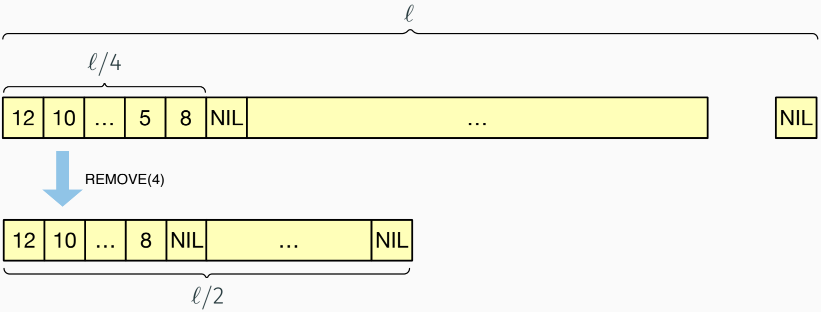
Idee

Wenn Array A zu klein ($n > \ell$), generiere neues Array der doppelten Größe 2ℓ.



Idee

Wenn Array A zu groß ($n \leq \ell/4$), generiere neues Array der halben Größe $\ell/2$.



Laufzeit solch dynamischer Felder

Lemma 4.1:

Betrachte ein Anfangs leeres dynamisches Feld A. Jede Folge σ von n INSERT und REMOVE Operationen auf A kann in Zeit $\Theta(n)$ bearbeitet werden.

Beweis später

- also im worst-case nur durchschnittlich konstante Laufzeit
- man spricht von amortisierter Laufzeit

Idee der Analyse

- vor jeder Verdopplung mit Kosten $\Theta(\ell)$ müssen...
- ...$\Theta(\ell)$ INSERT Operationen stattfinden

$\rightsquigarrow$ verrechne Kosten für Reallokierung mit INSERT Kosten

- Kosten für Halbierung können ähnlich verrechnet werden

Lemma 4.1:

Betrachte ein Anfangs leeres dynamisches Feld A. Jede Folge σ von n Insert und Remove Operationen auf A kann in Zeit $\Theta(n)$ bearbeitet werden.

Beweis später

- also im worst-case nur durchschnittlich konstante Laufzeit
- man spricht von amortisierter Laufzeit

Idee der Analyse

- vor jeder Verdopplung mit Kosten $\Theta(\ell)$ müssen…
- …$\Theta(\ell)$ Insert Operationen stattfinden

$\rightsquigarrow$ verrechne Kosten für Reallokierung mit Insert Kosten

- Kosten für Halbierung können ähnlich verrechnet werden

- man spricht hier auch von einem charging argument
- die Kosten der Reallokierungen werden den entsprechenden Insert Operationen „gecharged“ (zugewiesen)

Wie gut sind dynamische Felder?

Charakteristiken

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

Vorteile

- schnelles Einfügen
- schnelles Löschen
- Speicherbedarf linear in n

Nachteile

- hohe Laufzeit für Suche

Mögliche Verbesserungen

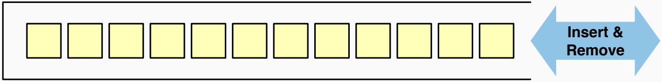
- Sortiertes dynamisches Feld?
  - schnellere Suche, aber linearer LZ beim Einfügen/Löschen
- Idee: sortiertes dynamisches Feld mit Lücken
  - geschickt verteilte Lücken erlauben Einfügen/Löschen in amortisierter LZ $\Theta(\log^2 n)$

- mittels Tricks auch worst-case Laufzeit $\Theta(1)$ (Stichwort „progressives Umkopieren“)
- Grund für lineare LZ: vergleiche mit innerer Schleife von InsertionSort
- Das ist das Prinzip einer Bibliothek! (Bücher alphabetisch sortiert; es gibt immer ein paar Lücken; wenn es eng wird, werden neue Regale angeschafft)
- die Analyse hiervon ist allerdings recht komplex

Stack

Stapel

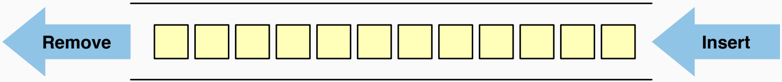
Eine Datenstruktur die das LIFO (last-in-first-out) Prinzip implementiert.



Queue

(Warte-) Schlange

Eine Datenstruktur die das FIFO (first-in-first-out) Prinzip implementiert.



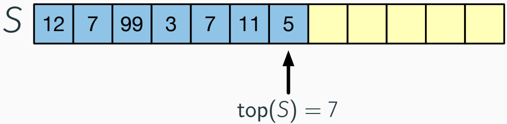
Stacks

Operationen

- PUSH: Einfügen eines Objektes
- POP: Entfernen des zuletzt eingefügten Objektes
- EMPTY: Überprüft ob Stack leer

Implementierung

- Stack mit maximal max Elementen
  - speichere Objekte in Array $S[1 \ldots \text{max}]$
  - $\text{top}(S)$ speichert Index des zuletzt eingefügten Objektes
- maximale Größe nicht bekannt $\rightsquigarrow$ dynamisches Feld



Implementierung der Stack-Operationen

EMPTY(S)

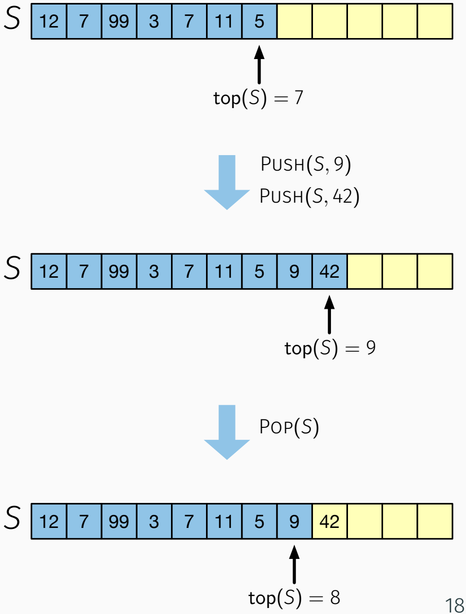
```
if top(S) = 0: return TRUE
else:          return FALSE
```

PUSH(S, x)

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

POP(S)

```
if EMPTY(S)
    return „Error: underflow“
top(S) ← top(S) − 1
return S[top(S) + 1]
```



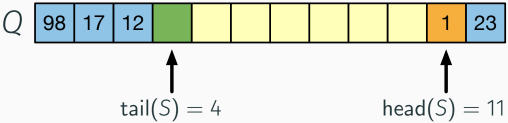
Queues

Operationen

- ENQUEUE: Einfügen eines Objektes
- DEQUEUE: Entfernen des ältesten Objektes in der Queue
- EMPTY: Überprüft ob Queue leer

Implementierung

- Queue mit maximal max Elementen
  - speichere Objekte in Array $Q[1 \dots \max + 1]$
  - $\mathrm{head}(Q)$ Index des ältesten Objektes in der Queue
  - $\mathrm{tail}(Q)$ „erste“ freie Position
    - interpretieren Array Kreisförmig (auf Position $\max + 1$ folgt Position 1)
- maximale Größe nicht bekannt ⇝ dynamisches Feld

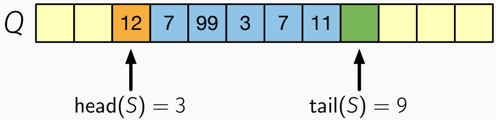


Implementierung der Queue-Operationen

EMPTY(Q)

```
if head(Q) = tail(Q)
    return TRUE
else
    return FALSE
```

ENQUEUE(Q, 42)
ENQUEUE(Q, 3)



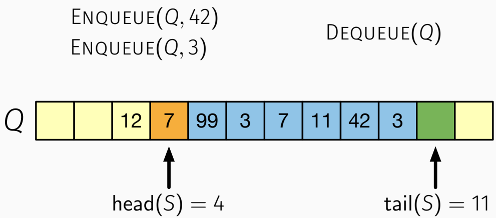
DEQUEUE(Q)

```
if EMPTY(Q): return „Error: underflow“
x ← Q[head(Q)]
if head(Q) = length(Q)
    head(Q) ← 1
else
    head(Q) ← head(Q) + 1
return x
```

ENQUEUE(Q, x)

```
Q[tail(Q)] ← x
if tail(Q) = length(Q)
    tail(Q) ← 1
else
    tail(Q) ← tail(Q) + 1
```

Implementierung der Queue-Operationen



EMPTY(Q)

```
if head(Q) = tail(Q)
    return TRUE
else
    return FALSE
```

DEQUEUE(Q)

```
if EMPTY(Q): return „Error: underflow“
x ← Q[head(Q)]
if head(Q) = length(Q)
    head(Q) ← 1
else
    head(Q) ← head(Q) + 1
return x
```

ENQUEUE(Q, x)

```
Q[tail(Q)] ← x
if tail(Q) = length(Q)
    tail(Q) ← 1
else
    tail(Q) ← tail(Q) + 1
```

Effizienz von Stacks & Queues

Theorem 4.1

Die Operationen eines statischen Stacks können mit Laufzeit $\Theta(1)$ implementiert werden.

Theorem 4.2

Die Operationen einer statischen Queue können mit Laufzeit $\Theta(1)$ implementiert werden.

Für dynamische Stacks/Queues:

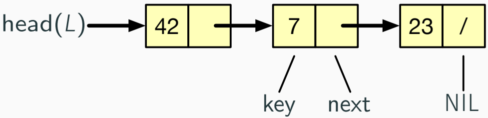
- dynamische Felder statt Arrays $\rightsquigarrow$ amortisierte Laufzeit $\Theta(1)$
- alternativ: Datenstrukturen mit Zeigern

Objekte, Referenzen, Zeiger

- Zugriff auf komplexe Objekte erfolgt meist per Referenz
    - effizienter für komplexe Datensätze
    - Standard für Java Objekte
    - in C/C++ durch explizite Zeiger
- wir verwenden folgende Begriffe synonym:
    - Referenzen
    - Zeiger
    - Verweise

Einfach verkettete Liste

- Menge von Objekten die über Verweise linear verkettet sind
- Verweise zeigen immer auf nächstes Element
- spezielle Informationen für Liste L und Objekte $x \in L$:
  - head(L): erstes Objekt der Liste L (oder NIL falls L leer)
  - next(x): das Objekt nach x in L (oder NIL falls x letztes Objekt)
  - key(x): Schlüssel von Objekt x

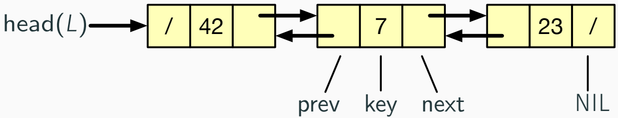


Einfach verkettete Liste

- Menge von Objekten die über Verweise linear verkettet sind
- Verweise zeigen immer auf nächstes Element
- spezielle Informationen für Liste L und Objekte $x \in L$:
  - head(L): erstes Objekt der Liste L (oder NIL falls L leer)
  - next(x): das Objekt nach x in L (oder NIL falls x letztes Objekt)
  - key(x): Schlüssel von Objekt x

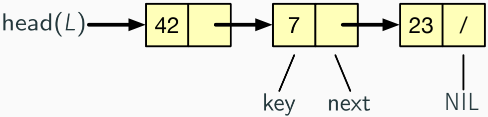
Doppelt verkettete Liste

- wie eine einfach verkettete Liste, aber...
- ...zusätzlich Verweis auf vorheriges Element:
  - prev(x): das Objekt vor x in L (oder NIL falls x erstes Objekt)

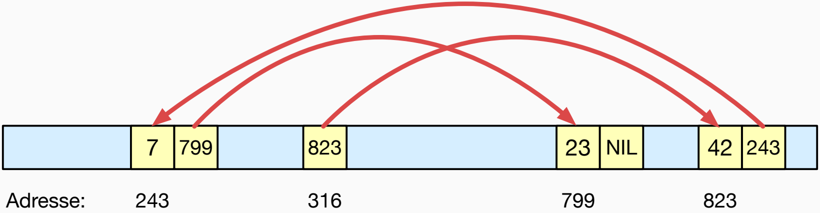


Darstellung: Abstract vs Speicher

Abstrakt

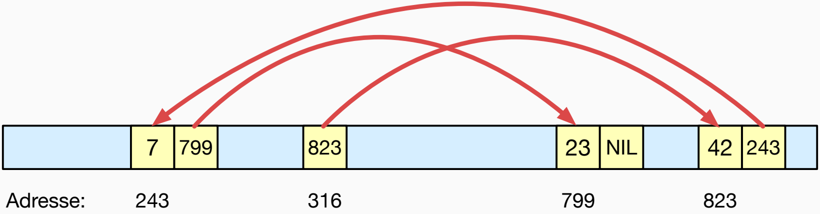


Im Speicher (linear adressierbar)



head(L) $= 316$

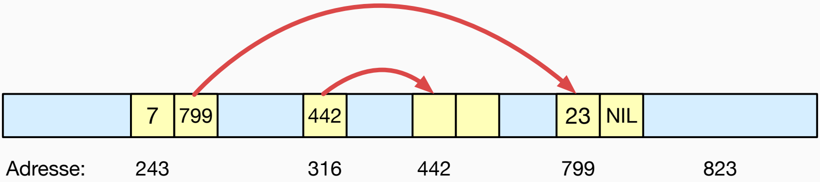
Auswirkungen von (De-)Allokationen im Speicher



$\mathrm{head}(L) = 316$

- $\mathrm{head}(L) \leftarrow \mathrm{NIL}$:
  Liste noch da, aber nicht mehr über **head**(L) erreichbar
- delete **head**(L):
  Speicherplatz des Ziels der Referenz wird freigegeben
- new **head**(L):
  Speicherplatz für neues Objekt wird angelegt und **head**(L) darauf verwiesen

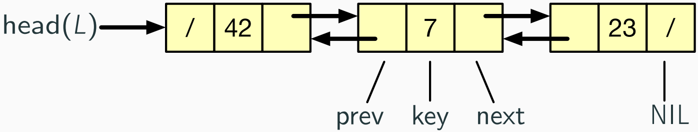
Auswirkungen von (De-)Allokationen im Speicher



head$(L) = 316$

- head$(L) \leftarrow$ NIL:
  Liste noch da, aber nicht mehr über **head**(L) erreichbar
- delete **head**(L):
  Speicherplatz des Ziels der Referenz wird freigegeben
- new **head**(L):
  Speicherplatz für neues Objekt wird angelegt und **head**(L) darauf verwiesen

Zeiger und Objekte



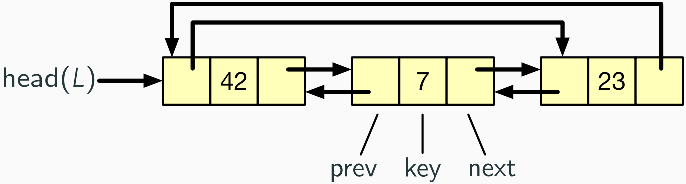
Zeigervariablen stehen stellvertretend für ihr Zielobjekt:

- $\text{key}(\text{head}(L)) = 42$
- $\text{key}(\text{prev}(\text{next}(\text{next}(\text{head}(L))))) = 7$

Varianten

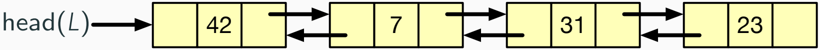
- Sortierte (doppelt) verkettete Liste:
  Listenreihenfolge entspricht sortierter Reihenfolge der Schlüssel (keys).

- zyklisch (doppelt) verkettete Liste:
  Ende der Liste zeigt auf Anfang.
    - **next** des letzten Objektes ist **head**(L)
    - **prev** von **head**(L) zeigt auf letztes Objekt



Operationen auf (doppelt) verketteten Listen

- INSERT(L, x): Hänge Objekt auf das Zeigervariable x zeigt an die Liste an.
- REMOVE(L, x): Lösche das Objekt auf das Zeigervariable x zeigt.
- SEARCH(L, k): Gib Zeiger auf (erstes) Objekt mit Schlüssel k zurück oder NIL, falls so ein Objekt nicht in L gespeichert ist.

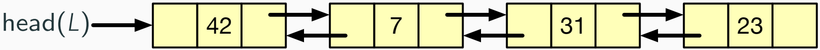


Pseudocode: Einfügen in doppelt verketteten Listen

```
INSERT(L, x)
next(x) ← head(L)
if head(L) ≠ NIL
    prev(head(L)) ← x
head(L) ← x
prev(x) ← NIL
```

Lemma 4.2

Algorithmus $\textsc{Insert}(L, x)$ hängt das Objekt x (vorne) an die Liste L an und hat Laufzeit $\Theta(1)$.



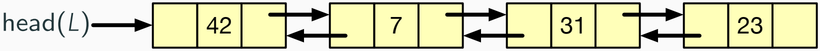
Pseudocode: Entfernen aus doppelt verketteten Listen

Remove(L, x)

```
if prev(x) ≠ NIL
    next(prev(x)) ← next(x)
else
    head(L) ← next(x)
if next(x) ≠ NIL
    prev(next(x)) ← prev(x)
```

Lemma 4.3

Algorithmus Remove(L, x) entfernt das Objekt x aus der Liste L an und hat Laufzeit $\Theta(1)$.



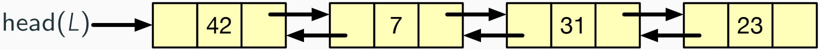
Pseudocode: Suchen in doppelt verketteten Listen

Search(L, k)

```
x ← head(L)
while x ≠ NIL and key(x) ≠ k
    x ← next(x)
return x
```

Lemma 4.4

Enthält die Liste L der Länge n ein Objekt mit Schlüssel k, dann gibt Search(L, k) solch ein Objekt zurück, andernfalls NIL. Die worst-case Laufzeit beträgt $\Theta(n)$.



Anmerkungen zur Analyse von INSERT/REMOVE

Für jede (korrekte) Liste L mit der aktuellen Menge $M = \{ O_1, O_2, \ldots, O_n \}$ der in L gespeicherten Objekte existiert eine Permutation π auf $\{ 1, 2, \ldots, n \}$, so dass:

- $\mathbf{head}(L) = O_{\pi(1)}$
- $\mathbf{prev}(O_{\pi(1)}) = \mathrm{NIL}$ und $\mathbf{prev}(O_{\pi(i)}) = O_{\pi(i-1)}$ für alle $i > 1$
- $\mathbf{next}(O_{\pi(i)}) = O_{\pi(i+1)}$ für alle $i < n$ und $\mathbf{next}(O_{\pi(n)}) = \mathrm{NIL}$

INSERT / REMOVE

- Korrektheit: obige Eigenschaft bleibt erhalten

Invari-
ante!

- Laufzeit: klar (nur Basisoperationen, keine Schleifen/Aufrufe)

Anmerkungen zur Analyse von Search

Search(L, k)

```
x ← head(L)
while x ≠ NIL and key(x) ≠ k
    x ← next(x)
return x
```

Invariante für Korrektheit

Zu Beginn des i-ten Schleifendurchlaufs gilt $\mathrm{key}(O_{\pi(j)}) \neq k$ für alle $j < i$.

Laufzeit

- Potentialfunktion: Position von x in der Liste

Vereinfachte Implementierung: Sentinels

Muss das so kompliziert sein?

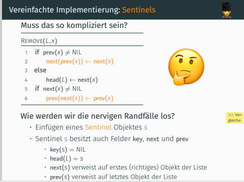
REMOVE(L, x)

```
if prev(x) ≠ NIL
    next(prev(x)) ← next(x)
else
    head(L) ← next(x)
if next(x) ≠ NIL
    prev(next(x)) ← prev(x)
```

Wie werden wir die nervigen Randfälle los?

NIL Vergleiche

- Einfügen eines Sentinel Objektes s
- Sentinel s besitzt auch Felder key, next und prev
    - $\text{key}(s) = \text{NIL}$
    - $\text{head}(L) = s$
    - $\text{next}(s)$ verweist auf erstes (richtiges) Objekt der Liste
    - $\text{prev}(s)$ verweist auf letztes Objekt der Liste

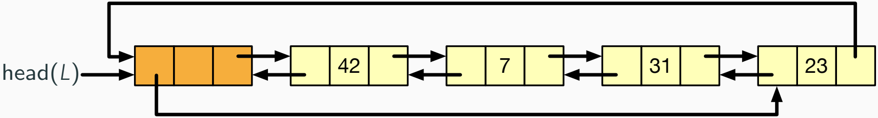


- solche Sentinel Objekte bezeichnet man auch als „Dummy Objects“

Vereinfachung per Sentinel: REMOVE

REMOVE(L, x)

next(prev(x)) $\leftarrow$ next(x)
prev(next(x)) $\leftarrow$ prev(x)

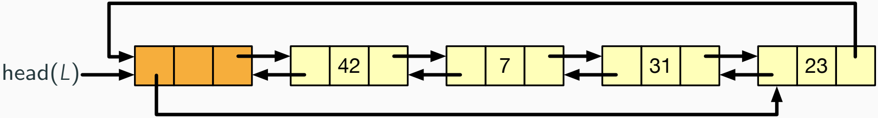


Im Pseudocode:

- Tests ob Nachfolger/Vorgänger vorhanden sind entfallen
- **head**(L) wird zu **next**(**head**(L))
- NIL wird zu **head**(L)

Vereinfachung per Sentinel: INSERT

```
INSERT(L, x)
next(x) ← next(head(L))
prev(next(head(L))) ← x
next(head(L)) ← x
prev(x) ← head(L)
```



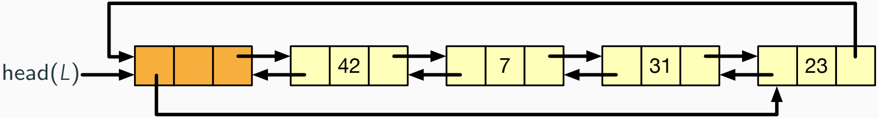
Im Pseudocode:

- Tests ob Nachfolger/Vorgänger vorhanden sind entfallen
- head(L) wird zu next(head(L))
- NIL wird zu head(L)

Vereinfachung per Sentinel: Search

SEARCH(L, k)

```
x ← next(head(L))
while x ≠ head(L) and key(x) ≠ k
      x ← next(x)
return x
```



Im Pseudocode:

- Tests ob Nachfolger/Vorgänger vorhanden sind entfallen
- head(L) wird zu next(head(L))
- NIL wird zu head(L)

Wie gut sind doppelt verkettete Listen?

Charakteristiken

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

Vorteile

- schnelles Einfügen
- schnelles Löschen
- Speicherbedarf linear in n

Nachteile

- hohe Laufzeit für Suche

Wie können wir eine schnelle Suche unterstützen?

Mehr Struktur durch Bäume!

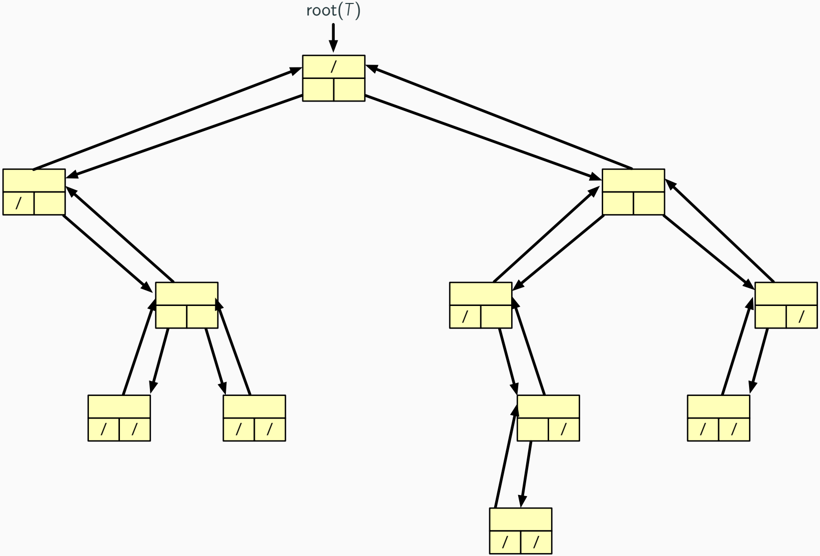
- Im Vergleich zu dynamischen Feldern gilt die LZ für Einfügen/Löschen also nicht nur amortisiert!

Binäre Bäume

- Menge von Objekten die über Verweise verkettet sind
- Verkettung hat Struktur eines Binärbaums
- spezielle Informationen für Binärbaum T und Objekte $x \in T$:
  - root(T): Verweis auf Objekt an der Wurzel des Binärbaums T
  - parent(x): Verweis auf Objekt im Elternknoten von x.
  - left(x): Verweis auf Objekt im linken Kind von x.
  - right(x): Verweis auf Objekt im rechten Kind von x.

- Zugriff auf T durch Verweis auf Wurzelknoten **root**(T)
- $\textbf{parent}(x) = \text{NIL} \iff x$ ist Wurzelknoten
- $\textbf{left}(x)/\textbf{right}(x) = \text{NIL} \iff$ kein linkes/rechtes Kind

Illustration eines binären Baumes



Allgemeine Bäume

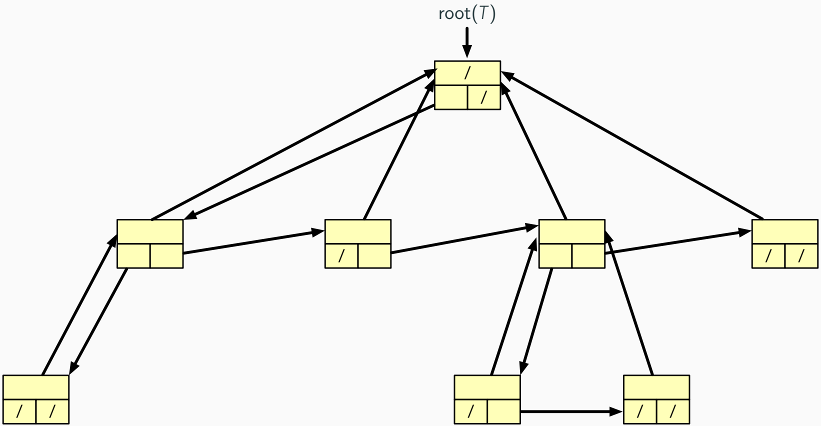
- Prinzip überträgt sich direkt auf k-näre Bäume
- aber nur für festes k
- statt **left** und **right** entsprechend **child**$_1$, …, **child**$_k$

Wie können wir Bäume mit unbeschränktem Grad abbilden?

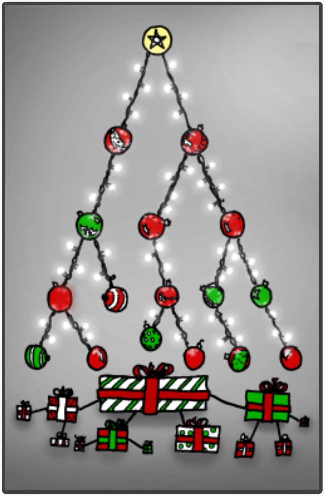
Idee

- nutze leftChild und rightSibling
- Knoten zeigt nur noch auf sein linkes Kind, …
- …restliche Kinder werden durch „Geschwister"-Links erreicht

Illustration eines allgemeinen Baumes



- Wurzel mit vier Kindern
- erstes Kind der Wurzel hat ein eigenes Kind
- drittes Kind der Wurzel hat zwei Kinder



https://xkcd.com/835/

Frohe Weihnachten!

Let’s hope for a heap of presents!

2) Binäre Suchbäume

Eigenschaften binärer Suchbäume

- schnelle Suche mit einem Binärbaum der...
- ...zusätzliche Struktur auf Schlüssel herstellt
  - also ähnlich zu Heaps, aber...
  - ...garantieren stärkere Ordnungseigenschaft

Binäre Suchbaumeigenschaft

- betrachte Knoten x und y im Binärbaum
- ist y im linken Unterbaum von x, dann gilt $\mathsf{key}(y) \leq \mathsf{key}(x)$
- ist y im rechten Unterbaum von x, dann gilt $\mathsf{key}(y) \geq \mathsf{key}(x)$

- schnelle Suche mit einem Binärbaum der...
- ...zusätzliche Struktur auf Schlüssel herstellt
  - also ähnlich zu Heaps, aber...
  - ...garantieren stärkere Ordnungseigenschaft

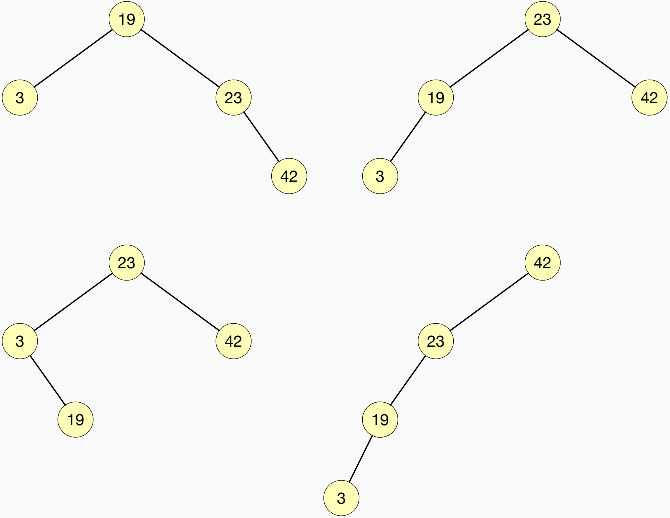
Binäre Suchbaumeigenschaft

- betrachte Knoten x und y im Binärbaum
- ist y im linken Unterbaum von x, dann gilt $\mathsf{key}(y) \leq \mathsf{key}(x)$
- ist y im rechten Unterbaum von x, dann gilt $\mathsf{key}(y) \geq \mathsf{key}(x)$

- unsere Heap Implementierung hat natürlich auch keine Zeigerstruktur sondern ein Array benutzt

Verschiedene Suchbäume für die gleichen Daten

- Schlüsselmenge: 42, 23, 3, 19



Statische Operationen auf binären Suchbäumen

- InorderTreeWalk(x): Ausgabe aller Schlüssel des in x gewurzelten binären Suchbaumes in (aufsteigend) sortierter Reihenfolge.
- Search(x, k): Gib Knoten mit Schlüssel k im in x gewurzelten binären Suchbaum aus.
- SearchMin(x): Suche Knoten mit minimalem Schlüssel im in x gewurzelten binären Suchbaum.
- SearchMax(x): Suche Knoten mit maximalem Schlüssel im in x gewurzelten binären Suchbaum.

- INORDERTREEWALK(x): Ausgabe aller Schlüssel des in x gewurzelten binären Suchbaumes in (aufsteigend) sortierter Reihenfolge.
- SEARCH(x, k): Gib Knoten mit Schlüssel k im in x gewurzelten binären Suchbaum aus.
- SEARCHMIN(x): Suche Knoten mit minimalem Schlüssel im in x gewurzelten binären Suchbaum.
- SEARCHMAX(x): Suche Knoten mit maximalem Schlüssel im in x gewurzelten binären Suchbaum.

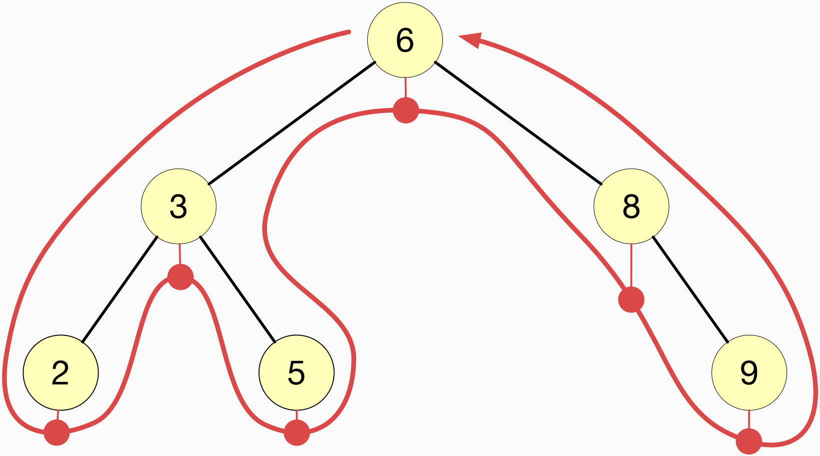
- Natürlich geben den entsprechenden Zeiger aus, da wir Bäume als verkettete Datenstruktur implementieren!

Statische Operationen auf binären Suchbäumen (weitere)

- SearchSuccessor(x): Suche Nachfolger von x bzgl. InorderTreeWalk des in x gewurzelten binären Suchbaumes.
- SearchPredecessor(x): Suche Vorgänger von x bzgl. InorderTreeWalk des in x gewurzelten binären Suchbaumes.

Idee: InorderTreeWalk

(Ausgabe in sortierter Reihenfolge)



Wie erhält man eine absteigend sortierte Ausgabe?

Pseudocode: InorderTreeWalk (Ausgabe in sortierter Reihenfolge)

```
InorderTreeWalk(x)
if x ≠ NIL
    InorderTreeWalk(leftChild(x))
    Print(key(x))
    InorderTreeWalk(rightChild(x))
```

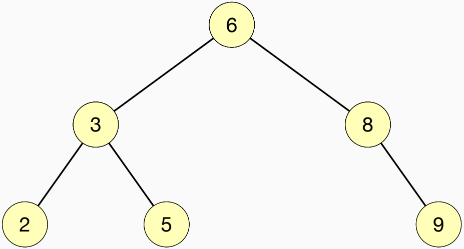
Lemma 4.5

InorderTreeWalk gibt alle Schlüssel in aufsteigend sortierter Reihenfolge aus und hat Laufzeit $\Theta(n)$.

(n: Anzahl der Elemente im Baum)

- Aufruf: InorderTreeWalk(root(T))
- Ausgabe im Beispiel:

  2, 3, 5, 6, 8, 9



Analyseskizze zu Lemma 4.5

Korrektheit

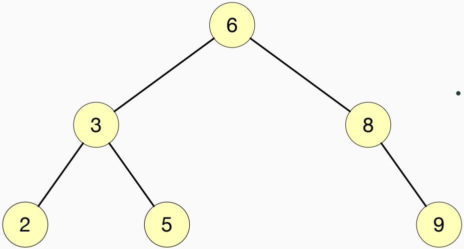
- mittels vollständiger Induktion (über Baumhöhe)
- Ausnutzen der Suchbaumeigenschaft

Laufzeit

- betrachte Aufruf $\textsc{InorderTreeWalk}(x)$
- sei n die Anzahl der Knoten im Teilbaum von x
- für Laufzeit $T(n)$ von $\textsc{InorderTreeWalk}(x)$ gilt:
    - Basisfall: $T(0) \leq c$ für eine Konstante $c > 0$
    - es existiert ein $i \in \{1, 2, \ldots, n\}$, so dass
$$T(n) \leq T(i-1) + c + T(n-i)$$
- zeige per vollständiger Induktion über n: $T(n) \leq 2c \cdot n + c$

Pseudocode: SEARCH

```
SEARCH(x, k)
if x = NIL or k = key(x)
    return x
if k < key(x)
    return SEARCH(leftChild(x), k)
else
    return SEARCH(rightChild(x), k)
```

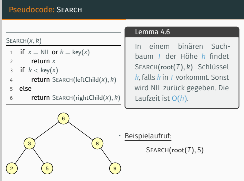


Lemma 4.6

In einem binären Suchbaum T der Höhe h findet $\textsc{Search}(\mathrm{root}(T), k)$ Schlüssel k, falls k in T vorkommt. Sonst wird NIL zurück gegeben. Die Laufzeit ist $O(h)$.

- Beispielaufruf:

$$\textsc{Search}(\mathrm{root}(T), 5)$$



- Korrektheit wieder per Induktion über Höhe + Suchbaumeigenschaft
- Laufzeit folgt, da Höhe des betrachteten Teilbaums mit jedem rekursiven Aufruf um eins abnimmt

Pseudocode: SearchMin

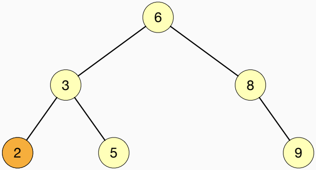
$\textsc{SearchMin}(x)$

```
while leftChild(x) ≠ NIL
    x ← leftChild(x)
return x
```

Lemma 4.7

In einem binären Suchbaum T der Höhe h findet $\textsc{SearchMin}(\mathrm{root}(T))$ den Knoten mit minimalem Schlüssel in Laufzeit $O(h)$.

- $\forall y$ im linken Teilbaum von x: $\mathrm{key}(y) \leq \mathrm{key}(x)$
  $\Longrightarrow$ Minimum ist „links unten“
- $\forall y$ im rechten Teilbaum von x: $\mathrm{key}(y) \geq \mathrm{key}(x)$
  $\Longrightarrow$ Maximum ist „rechts unten“



Pseudocode: SearchMax

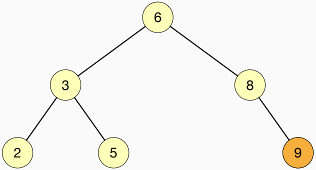
SearchMax(x)

```
while rightChild(x) ≠ NIL
    x ← rightChild(x)
return x
```

Lemma 4.7

In einem binären Suchbaum T der Höhe h findet SearchMin(root(T)) den Knoten mit minimalem Schlüssel in Laufzeit $O(h)$.

- $\forall y$ im linken Teilbaum von x: $\text{key}(y) \leq \text{key}(x)$
  $\implies$ Minimum ist „links unten“
- $\forall y$ im rechten Teilbaum von x: $\text{key}(y) \geq \text{key}(x)$
  $\implies$ Maximum ist „rechts unten“

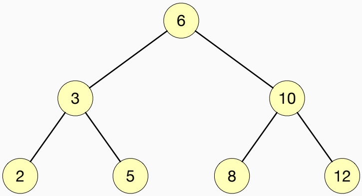


Pseudocode: SEARCHSUCCESSOR (Nachfolgersuche)

```
SEARCHSUCCESSOR(x)
if rightChild(x) ≠ NIL
    return SEARCHMIN(rightChild(x))
y ← parent(x)
while y ≠ NIL and x = rightChild(y)
    x ← y
    y ← parent(y)
return y
```

Lemma 4.8

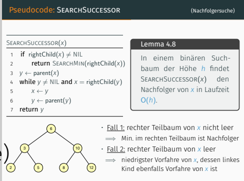
In einem binären Suchbaum der Höhe h findet SEARCHSUCCESSOR(x) den Nachfolger von x in Laufzeit $O(h)$.



- Fall 1: rechter Teilbaum von x nicht leer

  $\Longrightarrow$ Min. im rechten Teilbaum ist Nachfolger
- Fall 2: rechter Teilbaum von x leer

  $\Longrightarrow$ niedrigster Vorfahre von x, dessen linkes Kind ebenfalls Vorfahre von x ist



- Sind wir in Fall 2 und es gibt keinen solchen gesuchten Vorfahren, so hat x keinen Nachfolger.

Dynamische Operationen auf binären Suchbäumen

- $\underline{\textsc{Insert}(T, z)}$: Füge Knoten z zum binären Suchbaum T hinzu.
- $\underline{\textsc{Remove}(T, z)}$: Lösche Knoten z aus binärem Suchbaum T.

Müssen dabei die Suchbaumeigenschaft aufrecht erhalten!

Dynamische Operationen auf binären Suchbäumen

- Insert(T, z): Füge Knoten z zum binären Suchbaum T hinzu.
- Remove(T, z): Lösche Knoten z aus binärem Suchbaum T.

Müssen dabei die Suchbaumeigenschaft aufrecht erhalten!

- diese Operationen beeinflussen die Höhe des Suchbaumes
- im nächsten Unterkapitel beschäftigen wir uns damit, wie wir eine möglichst kleine Höhe garantieren können

Pseudocode: INSERT

```
INSERT(T, z)
y ← NIL; x ← root(T)
while x ≠ NIL
    y ← x
    if key(z) < key(y)
        x ← leftChild(y)
    else
        x ← rightChild(y)
parent(z) ← y
if y = NIL: root(T) ← z
else
    if key(z) < key(y)
        leftChild(y) ← z
    else
        rightChild(y) ← z
```

Lemma 4.9

In einem binären Suchbaum T der Höhe h fügt INSERT(T, z) den Knoten z korrekt ein. Die Laufzeit ist $O(h)$.

Idee: ähnlich zur Suche

- finde Einfügeposition („Suche“ nach key(z))
- ersetze NIL-Zeiger durch z…
  …und aktualisiere entsprechende Zeiger

Pseudocode: Remove

```
Remove(T, z)
if leftChild(z) = NIL or rightChild(z) = NIL
    y ← z
else
    y ← SearchSuccessor(z)
if leftChild(y) ≠ NIL
    x ← leftChild(y)
else
    x ← rightChild(y)
if x ≠ NIL: parent(x) ← parent(z)
if parent(z) = NIL
    root(T) ← x
else
    if y = leftChild(parent(y))
        leftChild(parent(y)) ← x
    else
        rightChild(parent(y)) ← x
if y ≠ z: key(z) ← key(y)
return y
```

Lemma 4.10

In einem binären Suchbaum T der Höhe h löscht Remove(T, z) den Knoten z und hält die Suchbaumeigenschaft aufrecht. Die Laufzeit ist $O(h)$.

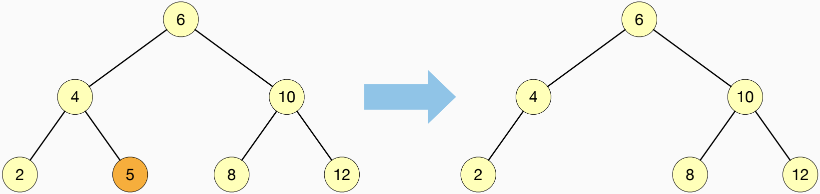
Idee: Betrachte 3 Fälle

(a) z hat keine Kinder

(b) z hat ein Kind

(c) z hat zwei Kinder

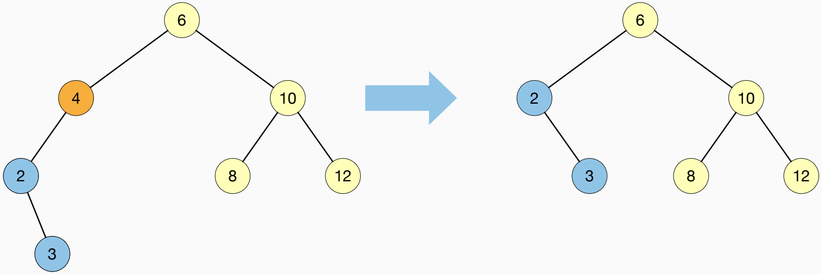
Entfernen eines Knoten: Fall 1



Zu löschendes Element z hat keine Kinder

- können Element einfach entfernen
- setze dazu entsprechenden Child-Zeiger des Parent auf NIL

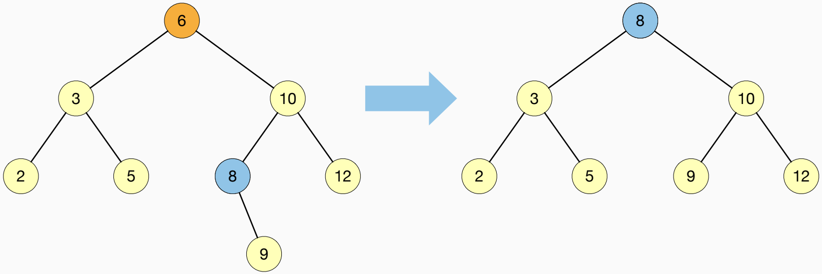
Entfernen eines Knoten: Fall 2



Zu löschendes Element *z* hat ein Kind

- ersetze *z* durch en Teilbaum seines Kindes

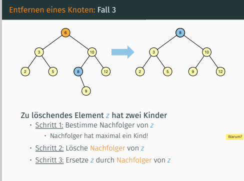
Entfernen eines Knoten: Fall 3



Zu löschendes Element z hat zwei Kinder

- Schritt 1: Bestimme Nachfolger von z
  - Nachfolger hat maximal ein Kind!
- Schritt 2: Lösche Nachfolger von z
- Schritt 3: Ersetze z durch Nachfolger von z

Warum?



- Nachfolger von z kann kein linkes Kind haben, sonst wäre dieses Nachfolger von z!
- Anders ausgedrückt: Nachfolger von z ist Minimum im Teilbaum von rightChild(z), liegt in diesem Teilbaum also ganz „links unten“ (und hat folglich kein leftChild).

Entfernen eines Knoten: Aufbau des Pseudocodes

```
REMOVE(T, z)
1   if leftChild(z) = NIL or rightChild(z) = NIL
2       y ← z
3   else
4       y ← SEARCHSUCCESSOR(z)
5   if leftChild(y) ≠ NIL
6       x ← leftChild(y)
7   else
8       x ← rightChild(y)
9   if x ≠ NIL: parent(x) ← parent(y)
10  if parent(y) = NIL
11      root(T) ← x
12  else
13      if y = leftChild(parent(y))
14          leftChild(parent(y)) ← x
15      else
16          rightChild(parent(y)) ← x
17  if y ≠ z: key(z) ← key(y)
18  return y
```

Finde zu löschenden Knoten

- Zeilen 1 bis 4: Bestimme zu löschenden Knoten y
- Zeilen 5 bis 8: Bestimme das Kind x von y (kann NIL sein)

Löschen von y

- Zeile 9: Aktualisiere Vaterzeiger von x
- Zeilen 10 bis 11: Aktualisiere entweder den Zeiger auf die Wurzel…
- Zeilen 12 bis 16: …oder den Child-Zeiger des Vaters von y

Abgleich mit zu löschendem Knoten

- Zeile 17: Aktualisiere ggfs. Daten des eigentlich zu löschenden Knotens z

Wie gut sind binäre Suchbäume?

Charakteristiken

- Platzbedarf: $\Theta(n)$
- Laufzeit Suche: $O(h)$
    - egal ob Suche nach...
    - ...Element, Miminimum, Maximum, Nachfolger oder Vorgänger
- Laufzeit Einfügen / Löschen: $O(h)$

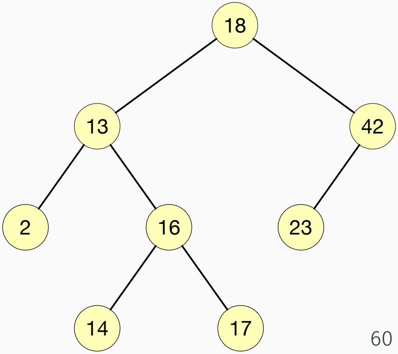
Wie können wir eine „kleine“ Höhe unter Einfügen und Löschen garantieren?

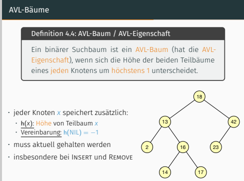
3) Balancierte Suchbäume

Definition 4.4: AVL-Baum / AVL-Eigenschaft

Ein binärer Suchbaum ist ein AVL-Baum (hat die AVL-Eigenschaft), wenn sich die Höhe der beiden Teilbäume eines jeden Knotens um höchstens 1 unterscheidet.

- jeder Knoten x speichert zusätzlich:
  - $\mathrm{h}(x)$: Höhe von Teilbaum x
  - Vereinbarung: $\mathrm{h}(\mathrm{NIL}) = -1$
- muss aktuell gehalten werden
- insbesondere bei INSERT und REMOVE





- benannt nach den Autoren Georgy Adelson-Velsky und Evgenii Landis; siehe Wikipedia für weitere Details
- $\mathsf{h}(x)$ ist also der Längste Pfad von x zu einem Blatt
- $\mathsf{h}(\mathrm{NIL}) = -1$ erlaubt schlicht einfacheren Pseudocode

Wie hoch sind AVL-Bäume?

Theorem 4.3

Für die Höhe h eines AVL-Baums mit n Knoten gilt

$$\left(\frac{3}{2}\right)^h \leq n \leq 2^{h+1} - 1.$$

Korollar 4.1

Ein AVL-Baum mit n Knoten hat Höhe $\Theta(\log n)$.

Operationen für Dynamische AVL-Bäume

- Standardoperationen wie bei binären Suchbäumen
  - Suchen, Such-Varianten, Einfügen, Löschen, ...
- Laufzeit $O(h)$ in Baum der Höhe h
  - für AVL-Bäume nach Korollar 4.1: $O(h) = O(\log n)$

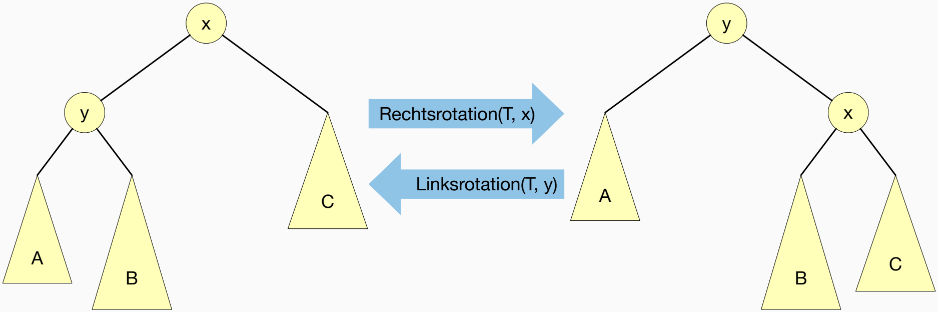
Problem

Einfügen und Löschen können AVL-Eigenschaft invalidieren!

Wie können wir die AVL-Eigenschaft wieder herstellen?

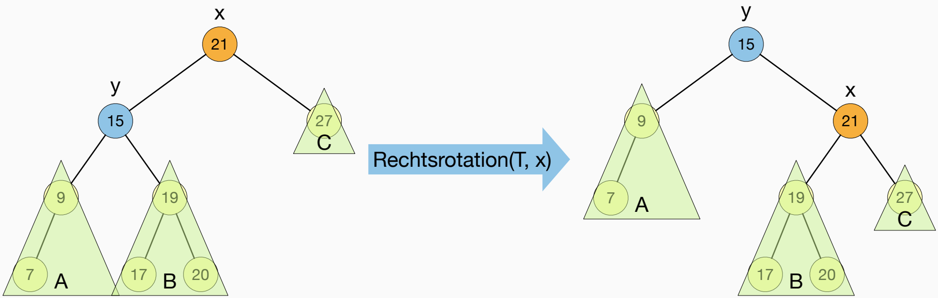
Rotationen in binären Suchbäumen

- Rotationen sind lokale Operationen auf einem binären Suchbaum, …
- …welche die Suchbaumeigenschaft erhalten
- wir betrachten zwei Arten von Rotationen:



- einfache Implementation in Laufzeit $\Theta(1)$

Konkrete Beispielrotation



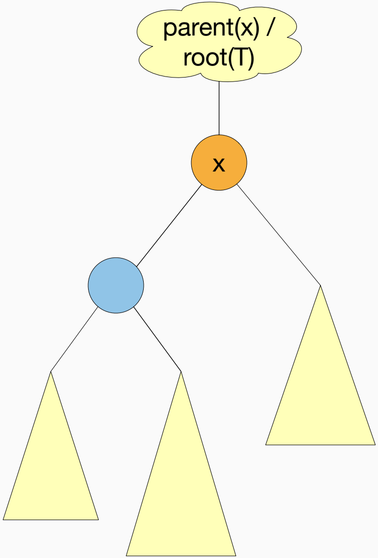
- Aufrechterhaltung der Suchbaumeigenschaft folgt leicht…
- …durch Betrachtung der veränderten Teilbäume + Suchbaumeigenschaft vor Rotation

Pseudocode: RightRotation

RightRotation(T, x)

```
y ← leftChild(x)
leftChild(x) ← rightChild(y)
if rightChild(y) ≠ NIL
    parent(rightChild(y)) ← x
parent(y) ← parent(x)
if parent(x) = NIL
    root(T) ← y
else
    if x = leftChild(parent(x))
        leftChild(parent(x)) ← y
    else
        rightChild(parent(x)) ← y
rightChild(y) ← x
parent(x) ← y
h(x) ← max { h(leftChild(x)), h(rightChild(x)) } + 1
h(y) ← max { h(leftChild(y)), h(rightChild(y)) } + 1
```

(LeftRotation wird analog implementiert)

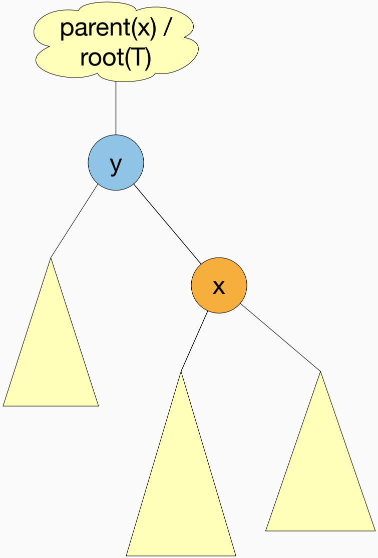


Pseudocode: RightRotation

RightRotation(T, x)

```
y ← leftChild(x)
leftChild(x) ← rightChild(y)
if rightChild(y) ≠ NIL
    parent(rightChild(y)) ← x
parent(y) ← parent(x)
if parent(x) = NIL
    root(T) ← y
else
    if x = leftChild(parent(x))
        leftChild(parent(x)) ← y
    else
        rightChild(parent(x)) ← y
rightChild(y) ← x
parent(x) ← y
h(x) ← max { h(leftChild(x)), h(rightChild(x)) } + 1
h(y) ← max { h(leftChild(y)), h(rightChild(y)) } + 1
```

(LeftRotation wird analog implementiert)



Aufrechterhaltung der AVL-Eigenschaft

- wir betrachten zunächst ein vereinfachtes Problem
- nutzen dieses später zur Wiederherstellung der AVL-Eigenschaft

Definition 4.5

Ein Baum heißt fast-AVL-Baum (hat die fast-AVL-Eigenschaft), wenn die AVL-Eigenschaft in jedem Knoten außer der Wurzel gilt und sich die Höhen der Unterbäume der Wurzel um höchstens 2 unterscheiden.

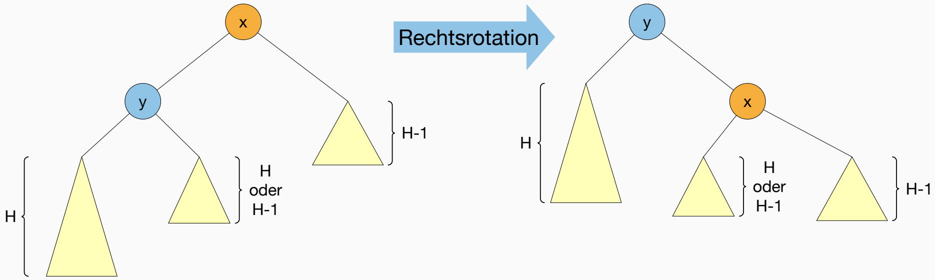
Können wir wenige Rotationen benutzen um aus einem fast-AVL-Baum einen AVL-Baum zu machen?

fast-AVL → AVL: Illustration der Operation

- betrachte fast-AVL-Baum gewurzelt in x
- o. B. d. A. sei linker Teilbaum höher
- 2 Fälle, je nachdem welcher Teilbaum von y höher ist

sonst analog

Fall 1

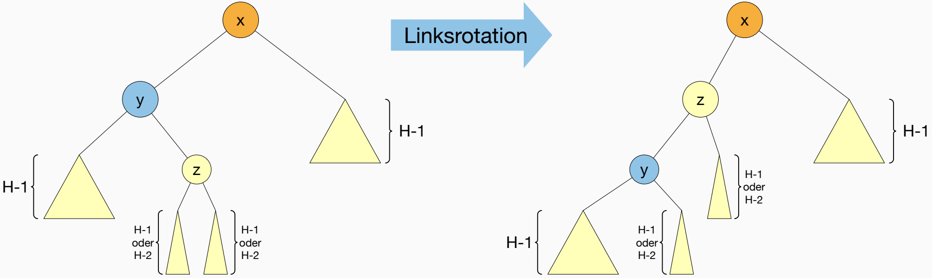


fast-AVL → AVL: Illustration der Operation

- betrachte fast-AVL-Baum gewurzelt in x
- o. B. d. A. sei linker Teilbaum höher
- 2 Fälle, je nachdem welcher Teilbaum von y höher ist

sonst analog

Fall 2

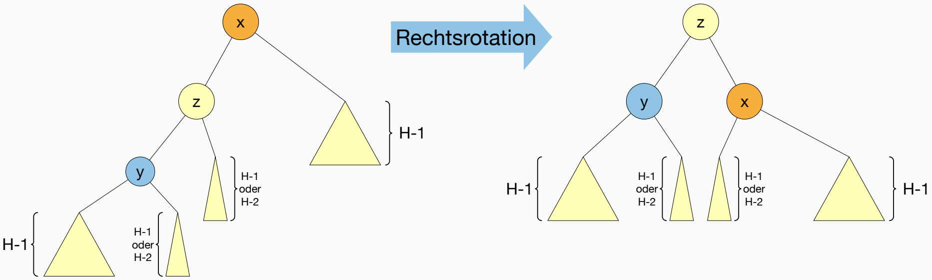


fast-AVL → AVL: Illustration der Operation

- betrachte fast-AVL-Baum gewurzelt in x
- o. B. d. A. sei linker Teilbaum höher
- 2 Fälle, je nachdem welcher Teilbaum von y höher ist

sonst analog

Fall 2

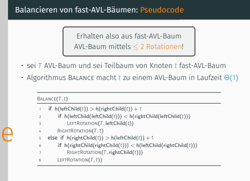


Balancieren von fast-AVL-Bäumen: Pseudocode

Erhalten also aus fast-AVL-Baum
AVL-Baum mittels $\leq$ 2 Rotationen!

- sei T AVL-Baum und sei Teilbaum von Knoten t fast-AVL-Baum
- Algorithmus Balance macht t zu einem AVL-Baum in Laufzeit $\Theta(1)$

```
Balance(T, t)
if h(leftChild(t)) > h(rightChild(t)) + 1
    if h(leftChild(leftChild(t))) < h(rightChild(leftChild(t)))
        LeftRotation(T, leftChild(t))
    RightRotation(T, t)
else if h(rightChild(t)) > h(leftChild(t)) + 1
    if h(rightChild(rightChild(t))) < h(leftChild(rightChild(t)))
        RightRotation(T, rightChild(t)))
    LeftRotation(T, t))
```



- BALANCE ist der Grund, weshalb Knoten in AVL-Bäumen auch die Höhe ihrer jeweiligen Teilbäume speichern!

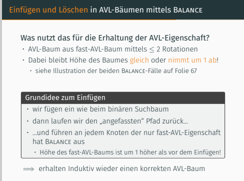
Was nutzt das für die Erhaltung der AVL-Eigenschaft?

- AVL-Baum aus fast-AVL-Baum mittels ≤ 2 Rotationen
- Dabei bleibt Höhe des Baumes gleich oder nimmt um 1 ab!
  - siehe Illustration der beiden BALANCE-Fälle auf Folie 67

Grundidee zum Einfügen

- wir fügen ein wie beim binären Suchbaum
- dann laufen wir den „angefassten“ Pfad zurück...
- ...und führen an jedem Knoten der nur fast-AVL-Eigenschaft hat BALANCE aus
  - Höhe des fast-AVL-Baums ist um 1 höher als vor dem Einfügen!

$\Longrightarrow$ erhalten Induktiv wieder einen korrekten AVL-Baum



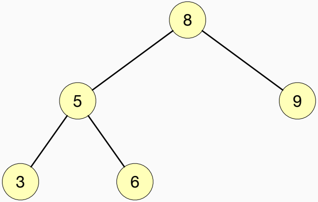
- „angefasster“ Pfad: alle Vorfahren des eingefügten Knoten
- Beachte: nur die Teilbäume von Vorfahren können die AVL-Eigenschaft verletzen!

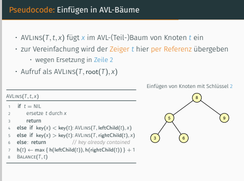
Pseudocode: Einfügen in AVL-Bäume

- AVLINS(T, t, x) fügt x im AVL-(Teil-)Baum von Knoten t ein
- zur Vereinfachung wird der Zeiger t hier per Referenz übergeben
  - wegen Ersetzung in Zeile 2
- Aufruf als AVLINS$(T, \mathbf{root}(T), x)$

```
AVLINS(T, t, x)
1  if t = NIL
2      ersetze t durch x
3      return
4  else if key(x) < key(t): AVLINS(T, leftChild(t), x)
5  else if key(x) > key(t): AVLINS(T, rightChild(t), x)
6  else: return                    // key already contained
7  h(t) ← max { h(leftChild(t)), h(rightChild(t)) } + 1
8  BALANCE(T, t)
```

Einfügen von Knoten mit Schlüssel 2





- Zeile 2 „schummelt“: wir verstecken hier das korrekte setzen der Child-Zeiger und des Parent-Zeigers
- echte Implementierungen könnten dazu z. B. Zeiger p auf Parent bei jedem Aufruf von AVLINS „mitschleppen“
- Zeile 7 nutzt unsere Definition von $\mathrm{h}(\mathrm{NIL}) = -1$

Analyseskizze zu AVLINS

Laufzeit

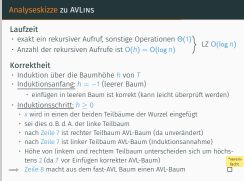
$$\left.\begin{array}{l}\text{• exakt ein rekursiver Aufruf, sonstige Operationen } \Theta(1) \\ \text{• Anzahl der rekursiven Aufrufe ist } O(h) = O(\log n)\end{array}\right\} \text{ LZ } O(\log n)$$

Korrektheit

- Induktion über die Baumhöhe h von T
- Induktionsanfang: $h = -1$ (leerer Baum)
    - einfügen in leeren Baum ist korrekt (kann leicht überprüft werden)
- Induktionsschritt: $h \geq 0$
    - x wird in einen der beiden Teilbäume der Wurzel eingefügt
    - sei dies o. B. d. A. der linke Teilbaum
    - nach Zeile 7 ist rechter Teilbaum AVL-Baum (da unverändert)
    - nach Zeile 7 ist linker Teilbaum AVL-Baum (Induktionsannahme)
    - Höhe von linkem und rechtem Teilbaum unterscheiden sich um höchstens 2 (da T vor Einfügen korrekter AVL-Baum)

$\Longrightarrow$ Zeile 8 macht aus dem fast-AVL Baum einen AVL-Baum $\square$

*vereinfacht



- man kann sogar zeigen, dass maximal eine Rebalancierung stattfindet
- das Argument, dass sich die Höhen der beiden Teilbäume um höchstens 2 unterscheiden, nutzt dass BALANCE die Höhe nicht vergrößert und maximal um 1 verringert
- formal müsste man diese Eigenschaft der Höhe in die Induktion mit aufnehmen

Löschen in AVL-Bäumen

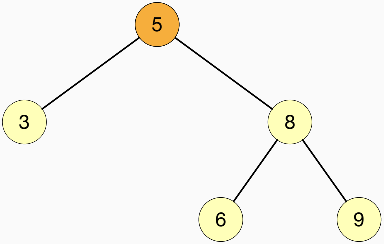
Grundidee zum Löschen

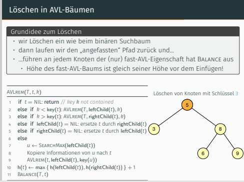
- wir Löschen ein wie beim binären Suchbaum
- dann laufen wir den „angefassten“ Pfad zurück und...
- ...führen an jedem Knoten der (nur) fast-AVL-Eigenschaft hat BALANCE aus
  - Höhe des fast-AVL-Baums ist gleich seiner Höhe vor dem Einfügen!

AVLREM(T, t, k)

```
if t = NIL: return // key k not contained
else if k < key(t): AVLREM(T, leftChild(t), k)
else if k > key(t): AVLREM(T, rightChild(t), k)
else if leftChild(t) = NIL: ersetze t durch rightChild(t)
else if rightChild(t) = NIL: ersetze t durch leftChild(t)
else
    u ← SEARCHMAX(leftChild(t))
    Kopiere Informationen von u nach t
    AVLREM(t, leftChild(t), key(u))
h(t) ← max { h(leftChild(t)), h(rightChild(t)) } + 1
BALANCE(T, t)
```

Löschen von Knoten mit Schlüssel 3





- Zeilen 4 und 5 sind wieder vereinfacht; auch hier müssen wieder die entsprechenden Zeiger im Baum aktualisiert werden

Löschen in AVL-Bäumen

Grundidee zum Löschen

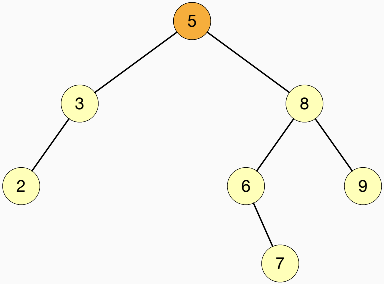
- wir Löschen ein wie beim binären Suchbaum
- dann laufen wir den „angefassten“ Pfad zurück und…
- …führen an jedem Knoten der (nur) fast-AVL-Eigenschaft hat BALANCE aus
  - Höhe des fast-AVL-Baums ist gleich seiner Höhe vor dem Einfügen!

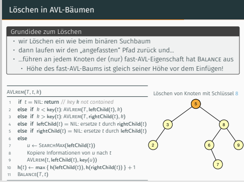
AVLREM(T, t, k)

---

if $t =$ NIL: **return** *// key k not contained*
else if $k <$ **key**(t): AVLREM$(T,$ **leftChild**$(t), k)$
else if $k >$ **key**(t): AVLREM$(T,$ **rightChild**$(t), k)$
else if **leftChild**$(t) =$ NIL: ersetze t durch **rightChild**(t)
else if **rightChild**$(t) =$ NIL: ersetze t durch **leftChild**(t)
else
    $u \leftarrow$ SEARCHMAX(**leftChild**(t))
    Kopiere Informationen von u nach t
    AVLREM$(t,$ **leftChild**$(t),$ **key**$(u))$
h$(t) \leftarrow$ **max** $\{$ **h**(**leftChild**(t)), **h**(**rightChild**(t)) $\} + 1$
BALANCE(T, t)

Löschen von Knoten mit Schlüssel 8





- Zeilen 4 und 5 sind wieder vereinfacht; auch hier müssen wieder die entsprechenden Zeiger im Baum aktualisiert werden

Ergebnis für Einfügen und Löschen in AVL-Bäumen

- Laufzeit von Löschen ist wieder $O(h) = O(\log n)$
- Analyse ähnlich zur Analyse für Einfügen

Theorem 4.4

AVL-Bäume für n Elemente unterstützen Suchen, Min-/Max-Bestimmung, Einfügen und Löschen in Laufzeit $\Theta(\log n)$.

Wie gut sind AVL-Bäume?

Charakteristiken

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

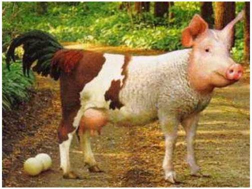
Vorteile

- schnelles Suchen
- Speicherbedarf linear in n

Nachteile

- mittelmäßiges Einfügen
- mittelmäßiges Löschen

Was können wir noch verbessern?



4) Hashing

Was ist Hashing?

- einfache Methode, um Wörterbücher zu implementieren
- unterstützt also die Operationen
  - INSERT
  - REMOVE
  - SEARCH

Eigenschaften

- worst-case Zeit für Suche $\Theta(n)$ 😞
- in der Praxis jedoch sehr gut
- unter gewissen Annahmen erwartete Suchzeit $\Theta(1)$ 😀

Hashing ist eine Verallgemeinerung von direkter Adressierung durch Arrays!

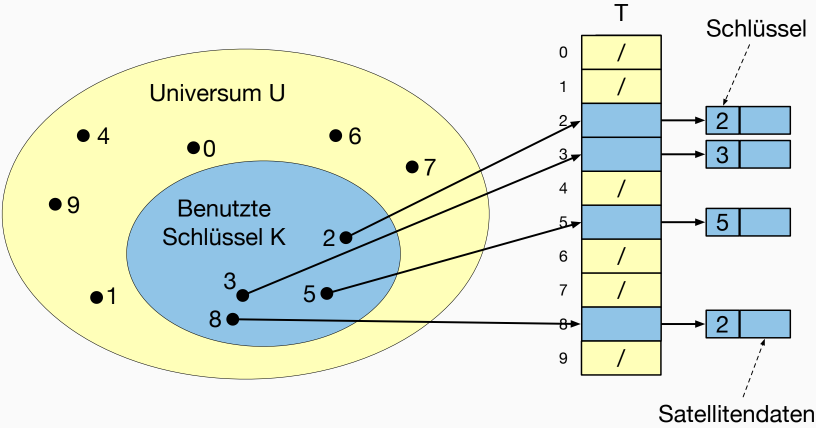
Direkte Adressierung mit Arrays

- Datenstruktur speichert Objekte mit Schlüsseln
- Schlüssel aus Universum $U = \{ 0, 1, \ldots, u - 1 \}$
- nehmen an, dass die Schlüssel eindeutig sind
  - es gibt keine zwei Objekte mit dem gleichen Schlüssel

Idee

- lege Array $T[0 \ldots u - 1]$ an
- Position k in T reserviert für Objekt mit Schlüssel k
- Eintrag $T[k]$ verweist auf Objekt mit Schlüssel k
- Objekt mit Schlüssel k nicht vorhanden $\implies T[k] = \mathrm{NIL}$

Illustration: Direkte Adressierung



Operationen bei direkter Adressierung

INSERT(T, x)

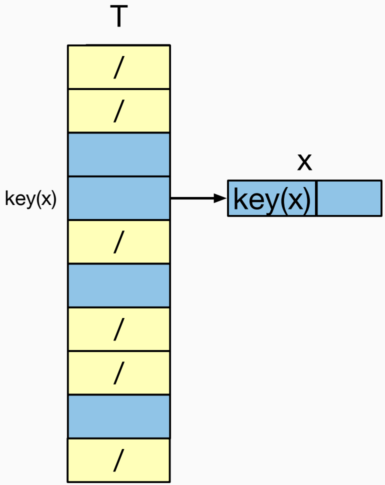
1 $T[\text{key}(x)] \leftarrow x$

REMOVE(T, k)

1 $T[k] \leftarrow \text{NIL}$

SEARCH(T, k)

1 $\mathit{return} T[k]$



Charakteristiken direkter Adressierung

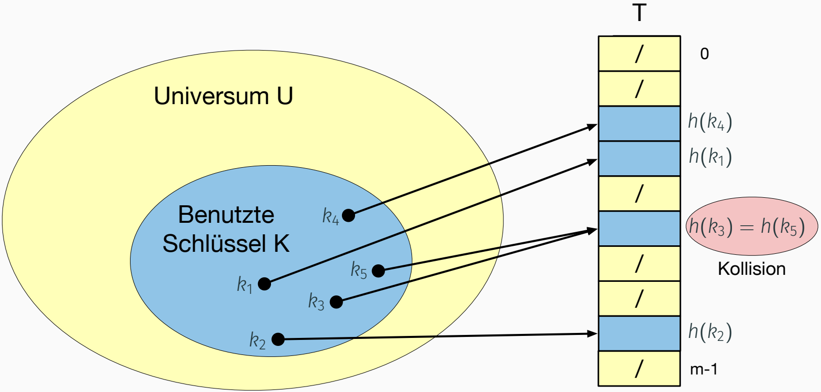
- einfach, schnell (alle Operationen in Laufzeit $\Theta(1)$)
- nicht möglich für unendlich großes Universum U
- Platzbedarf: $\Theta(|U|)$
  - speicherineffizient, wenn Universum sehr groß...
  - ...im Vergleich zur Anzahl der zu speichernden Objekte

Idee

- aktuelle Schlüsselmenge sei $K \subseteq U$ mit $m := |K|$
- verwende Hashfunktionen $h\colon U \to \{0, 1, \ldots, m-1\}$...
- ...zur Abbildung der Objekte auf kleine Hashtabelle

Aber wie gehen wir mit Kollisionen um?

Illustration: Hashing und Kollisionen

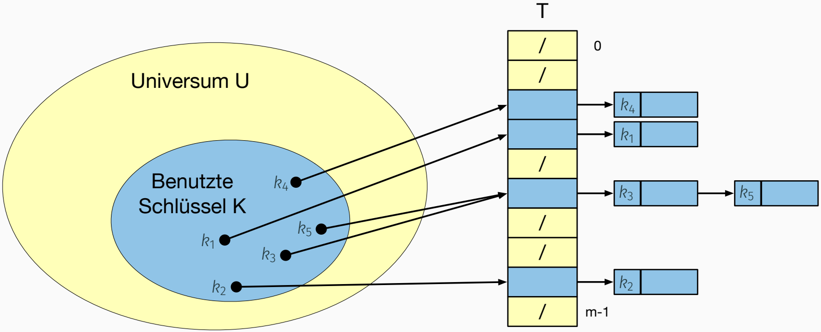


Kollisionen können für $m < |U|$ nicht vermieden werden!

Strategien zur Kollisionsauflösung

1. Geschlossene Adressierung:
   - Kollisionsauflösung durch Listen
2. Offene Adressierung:
   - lineares / quadratisches Hashing
   - Verfahren zur Suche nach „nächster“ freien Stelle in T
3. ...und viele mehr!

Strategie 1: Geschlossene Adressierung



Strategie 1: Operationen

CHINSERT(T, x)

Füge x vorne in Liste $T[h(\text{key}(x))]$ ein.

CHREMOVE(T, k)

Entferne alle x mit $\text{key}(x) = k$ aus Liste $T[h(k)]$.

CHSEARCH(T, x)

Suche nach Element mit Schlüssel k in Liste $T[h(k)]$.

- CHXY = „Chained-Hash-XY“
- Laufzeit für CHINSERT: $\Theta(1)$
- Laufzeit für CHREMOVE und CHSEARCH: proportional zu $T[h(k)]$

Wie gut/schlecht ist Hashing mit Listen?

- m Größe der Hashtabelle T, n Anzahl der gespeicherten Objekte
- definiere Lastfaktor von T als $\alpha := n/m$
  - die durchschnittliche Anzahl an Objekten in einer verketteten Liste

Laufzeit der Suche

- Worst-case:
  - Hashfunktion könnte alle Objekte auf denselben Wert hashen

$\Longrightarrow$ dann Laufzeit $\Theta(n)$

- Aber:
  - gute Hashfunktionen haben im Durchschnitt kurze Listen!

Gute Hashfunktionen verhalten sich wie echt zufällige Funktionen. Aber was genau heißt das?

Definition 4.6: Universelles Hashing

Sei $c > 0$ konstant. Eine Familie H von Hashfunktionen $h \colon U \to \{0, 1, \ldots, m-1\}$ heißt c-universell falls für ein beliebiges Paar $x, y \in U$ mit $x \neq y$

$$|\{h \in H \mid h(x) = h(y)\}| \leq \frac{c}{m} \cdot |H|$$

gilt.

- insbesondere gilt bei uniform zufälliger Wahl von $h \in H$:

$$\Pr[h(x) = h(y)] \leq \frac{c}{m} \cdot |H| \cdot \frac{1}{|H|} = \frac{c}{m}$$

Erwartete Listenlänge unter universellem Hashing

Theorem 4.5

Sei H eine c-universelle Familie von Hashfunktionen. Die Menge $K \subseteq U$ mit $n = |K|$ werde in einer Hashtabelle T der Größe m mittels einer uniform zufällig gewählten Hashfunktion $h \in H$ gespeichert. Dann hat $T[i]$ für alle $i \in \{0, 1, \ldots, m-1\}$ erwartete Länge $\mathrm{O}(1 + c \cdot \alpha)$.

Beweis.

- betrachte beliebigen Schlüssel $k_0 \in K$ mit $h(k_0) = i$
- $\forall k \in K \setminus \{k_0\}$ sei Zufallsvariable $X_k \in \{0, 1\}$ mit $X_k = 1 \iff h(k) = i$
- sei $X := \sum_{k \neq k_0} X_k$: Länge der Liste $h(k_0)$ minus 1
- dann gilt

$$\mathbb{E}[X] = \sum_{k \neq k_0} \mathbb{E}[X_k] = \sum_{k \neq k_0} 1 \cdot \Pr[X_k = 1] \leq \sum_{k \neq k_0} c/m = (n-1) \cdot c/m$$

- also Länge von $h(k_0)$ erwartet $\leq (n-1) \cdot c/m + 1 \leq 1 + c \cdot \alpha$ $\square$

Erwartete Listenlänge unter universellem Hashing

Theorem 4.5

Sei H eine c-universelle Familie von Hashfunktionen. Die Menge $K \subseteq U$ mit $n = |K|$ werde in einer Hashtabelle T der Größe m mittels einer uniform zufällig gewählten Hashfunktion $h \in H$ gespeichert. Dann hat $T[i]$ für alle $i \in \{0, 1, \dots, m-1\}$ erwartete Länge $O(1 + c \cdot \alpha)$.

Beweis.

- betrachte beliebigen Schlüssel $k_0 \in K$ mit $h(k_0) = i$
- $\forall k \in K \setminus \{k_0\}$ sei Zufallsvariable $X_k \in \{0, 1\}$ mit $X_k = 1 \iff h(k) = i$
- sei $X := \sum_{k \neq k_0} X_k$: Länge der Liste $h(k_0)$ minus 1
- dann gilt

$$\mathbb{E}[X] = \sum_{k \neq k_0} \mathbb{E}[X_k] = \sum_{k \neq k_0} 1 \cdot \Pr[X_k = 1] \leq \sum_{k \neq k_0} c/m = (n-1) \cdot c/m$$

- also Länge von $h(k_0)$ erwartet $\leq (n-1) \cdot c/m + 1 \leq 1 + c \cdot \alpha$ □

- Ungleichung folgt, da H c-universell ist

Was bringt uns das?

Direkte Folge aus Theorem 4.5

Wenn Hashtabelle Größe $\Theta(|K|)$ hat, haben alle Operationen konstante (erwartete) Laufzeit!

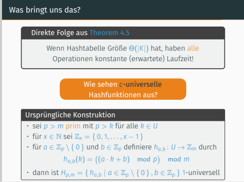
Wie sehen c-universelle Hashfunktionen aus?

Ursprüngliche Konstruktion

- sei $p > m$ prim mit $p > k$ für alle $k \in U$
- für $x \in \mathbb{N}$ sei $\mathbb{Z}_x = \{ 0, 1, \ldots, x-1 \}$
- für $a \in \mathbb{Z}_p \setminus \{ 0 \}$ und $b \in \mathbb{Z}_p$ definiere $h_{a,b} \colon U \to \mathbb{Z}_m$ durch

$$h_{a,b}(k) = \big((a \cdot k + b) \mod p\big) \mod m$$

- dann ist $H_{p,m} = \{ h_{a,b} \mid a \in \mathbb{Z}_p \setminus \{ 0 \}, b \in \mathbb{Z}_p \}$ 1-universell



- für Details zur ursprünglichen Konstruktion siehe Wikipedia (dort findet sich auch ein einfacher Beweis der 1-Universalität von $H_{p,m}$

Strategien zur Kollisionsauflösung

1. Geschlossene Adressierung:
    - Kollisionsauflösung durch Listen
2. Offene Adressierung:
    - lineares / quadratisches Hashing
    - Verfahren zur Suche nach „nächster“ freien Stelle in T
3. ...und viele mehr!

Strategie 2: Offene Adressierung

Haben gesehen: Geschlossene Adressierung

- Objekte haben eine feste Position in der Hashtabelle
- diese Position hängt insbesondere nur vom Schlüssel ab

Offene Adressierung

- Objekte haben keine feste Position
- die Position ist nun Abhängig vom Schlüssel…
- …und der aktuellen Belegung der Hashtabelle

Endposition „offen“

Grundidee

- für neues Objekt wird erste freie Position gesucht
- verschiedene Strategien zur Wahl der „nächsten“ Position

Strategie 2: Offene Adressierung und Listen

- typischerweise keine Listen zur Kollisionsvermeidung
- d. h. Hashtabelle voll $\Longrightarrow$ Einfügen nicht mehr möglich

Warum?

- Listen zur Kollisionsvermeidung auch hier denkbar
- aber: wollen lange „Verweisketten" vermeiden
  - erlaubt (in der Regel) schnellere Suche…
  - …opfert aber konstante worst-case Laufzeit zum Einfügen
- außerdem: Datenstruktur ist ohne Liste schön einfach 😀

Strategie 2: Formalisierung der „ersten freien“ Position

- betrachten Hashfunktion
  $h\colon U \times \{0, 1, \ldots\} \to \{0, 1, \ldots, m-1\}$
- definiert für jeden Schlüsssel $k \in U$ seine Testfolge

$$\big(h(k,0), h(k,1), \ldots\big)$$

- ideal: Testfolge ist eine Permutation von $\{0, 1, \ldots, m-1\}$

Wir können annehmen, dass jede Testfolge höchstens Länge m hat.
Warum?

Strategie 2: Formalisierung der „ersten freien“ Position

- betrachten Hashfunktion $h\colon U \times \{0, 1, \dots\} \to \{0, 1, \dots, m-1\}$
- definiert für jeden Schlüssel $k \in U$ seine Testfolge
  $$(h(k, 0), h(k, 1), \dots)$$
- ideal: Testfolge ist eine Permutation von $\{0, 1, \dots, m-1\}$

Wir können annehmen, dass jede Testfolge höchstens Länge m hat. Warum?

- ist die Testfolge eine Permutation von m, so wird jede Position der Hashtabelle genau einmal getestet

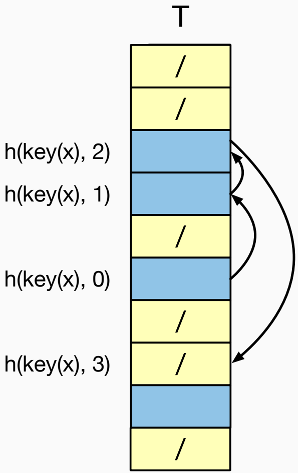
Offene Adressierung: Einfügen & Suchen

$\text{OHINSERT}(T, x)$

```
for i ← 0 to m − 1
    j ← h(key, i)
    if T[j] = NIL
        T[j] ← x
        return
error „out of space“
```

$\text{OHSEARCH}(T, k)$

```
i ← 0
repeat
    j ← h(k, i)
    if key(T[j]) = k: return T[j]
    else: i ← i + 1
until T[j] = NIL or i = m
return NIL
```



(OHXY = „Open-Hash-XY“)

Offene Adressierung: Was ist mit Löschen?

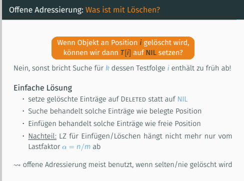
Wenn Objekt an Position i gelöscht wird, können wir dann $T[i]$ auf NIL setzen?

Nein, sonst bricht Suche für k dessen Testfolge i enthält zu früh ab!

Einfache Lösung

- setze gelöschte Einträge auf DELETED statt auf NIL
- Suche behandelt solche Einträge wie belegte Position
- Einfügen behandelt solche Einträge wie freie Position
- <u>Nachteil:</u> LZ für Einfügen/Löschen hängt nicht mehr nur vom Lastfaktor $\alpha = n/m$ ab

$\rightsquigarrow$ offene Adressierung meist benutzt, wenn selten/nie gelöscht wird



- Trick für effizientes Entfernen bei offener Adressierung: Zähle Anzahl der Deleted Einträge
- übersteigt die Anzahl solcher Einträge die Anzahl der Elemente in der Hashtabelle, baue Hashtabelle nochmal komplett neu auf
- für $m = \Omega(|K|)$ erhält man amortisierte konstante Laufzeit für Einfügen, Löschen und Suchen

- gegeben eine Hashfunktion $h' \colon U \to \{ 0, 1, \ldots, m-1 \}$
- sowie natürliche Zahlen $c_1, c_2 \neq 0$

Lineares Hashing

$$h(k,i) := \big(h'(k) + i\big) \mod m$$

- Problem: es entstehen lange Ketten besetzter Plätze
- man spricht von primärem Clustering

Quadratisches Hashing

$$h(k,i) := \big(h'(k) + c_1 \cdot i + c_2 \cdot i^2\big) \mod m$$

- volle Ausnutzung der Hashtabelle benötigt spezielle c_1, c_2, m
- Problem: $h'(k_1) = h'(k_2) \implies k_1$ und k_2 die gleiche Testfolge
- man spricht von sekundärem Clustering

Konstruktionen für Hashing mit offener Adressierung

- gegeben eine Hashfunktion $h'\colon U \to \{0, 1, \dots, m-1\}$
- sowie natürliche Zahlen $c_1, c_2 \neq 0$

Lineares Hashing

$$h(k,i) := (h'(k) + i) \mod m$$

- Problem: es entstehen lange Ketten besetzter Plätze
- man spricht von primärem Clustering

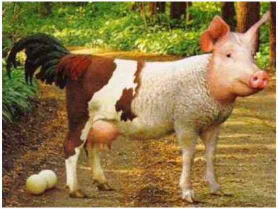
Quadratisches Hashing

$$h(k,i) := (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \mod m$$

- volle Ausnutzung der Hashtabelle benötigt spezielle c_1, c_2, m
- Problem: $h'(k_1) = h'(k_2) \implies k_1$ und k_2 die gleiche Testfolge
- man spricht von sekundärem Clustering

- lange Ketten entstehen, da eine Kette der Länge i mit Wahrscheinlichkeit $(i+1)/m$ beim Einfügen um eins verlängert wird

Haben wir mit Hashing unser Ziel erreicht?



Charakteristiken

- erlaubt sehr effiziente Dictionaries…
- …solange Lastfaktor nicht zu groß
- Lastfaktor α konstant
  $\implies$ erw. LZ $\Theta(1)$ für alle Operationen

Aber

- Kollisionen nicht vermeidbar
- worst-case Laufzeiten sind $\Theta(n)$
- Auswahl/Verteilung der eingefügten Schlüssel entscheidend

Ausblick auf weitere Hashverfahren

Perfektes Hashing

- worst-case LZ $\Theta(1)$ für statische Dictionaries
- Kollisionsverwaltung nutzt statt Listen wieder Hashtabellen

Dynamic Hashing

- Reallokation der Hashtabelle wenn zu voll/leer
- ähnlich zu dynamischen Feldern
- Fallstrick: Für manche Verfahren muss m prim sein!

Cuckoo Hashing

- erlaubt Suche in worst-case LZ $\Theta(1)$
- clevere Nutzung zweier Hashfunktionen
- Varianten extrem gut in Praxis und Theorie
- siehe Wikipedia