



Datenstrukturen, Algorithmen und Programmierung 2 (DAP2)

Organisatorisches

Übungen

- Ab der nächsten Heimübung ist die Teilnahme an der Besprechung der Heimübung nicht mehr verpflichtend

Lernraumbetreuung

- Mo 14:30-17:30 in U04 zur C-Betreuung (Praktikum)
- Dafür fällt aus: Mo 14:30 – 17:30

Zweiter Test

- Aufgabe 1: Nachvollziehen eines Algorithmus oder einer Datenstruktur
- Aufgabe 2: Entwurf und Analyse eines Algorithmus zur dynamischen Programmierung
- Aufgabe 3: Entwurf und Analyse eines gierigen Algorithmus

Datenstrukturen

Datenstruktur Feld

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

Vorteile

- Schnelles Einfügen und Löschen

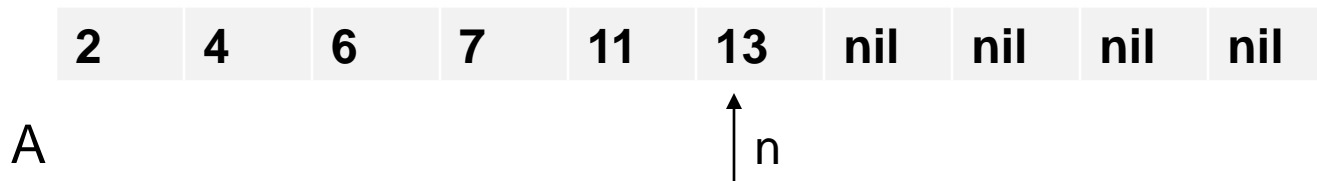
Nachteile

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

Datenstrukturen

Datenstruktur „sortiertes Feld“

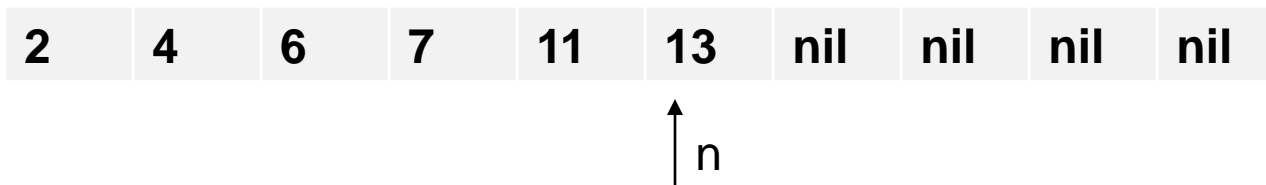
- **Sortiertes** Feld $A[1, \dots, \max]$
- Integer n , $1 \leq n \leq \max$
- n bezeichnet Anzahl Elemente in Datenstruktur



Datenstrukturen

Einfügen(s)

1. $n \leftarrow n+1$
2. $i \leftarrow n$
3. **while** $s < A[i-1]$ **do**
4. $A[i] \leftarrow A[i-1]$
5. $i \leftarrow i - 1$
6. $A[i] \leftarrow s$



Einfügen(10)

Datenstrukturen

Einfügen(s)

1. $n \leftarrow n+1$
2. $i \leftarrow n$
3. **while** $s < A[i-1]$ **do**
4. $A[i] \leftarrow A[i-1]$
5. $i \leftarrow i-1$
6. $A[i] \leftarrow s$



Einfügen(10)

Datenstrukturen

Einfügen(s)

1. $n \leftarrow n+1$
2. $i \leftarrow n$
3. **while** $s < A[i-1]$ **do**
4. $A[i] \leftarrow A[i-1]$
5. $i \leftarrow i - 1$
6. $A[i] \leftarrow s$

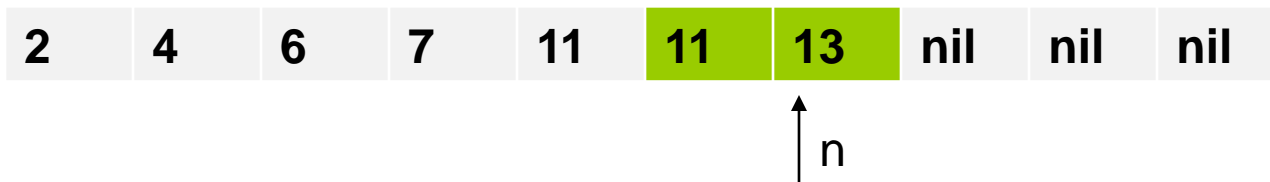


Einfügen(10)

Datenstrukturen

Einfügen(s)

1. $n \leftarrow n+1$
2. $i \leftarrow n$
3. **while** $s < A[i-1]$ **do**
4. $A[i] \leftarrow A[i-1]$
5. $i \leftarrow i-1$
6. $A[i] \leftarrow s$



Einfügen(10)

Datenstrukturen

Einfügen(s)

1. $n \leftarrow n+1$
2. $i \leftarrow n$
3. **while** $s < A[i-1]$ **do**
4. $A[i] \leftarrow A[i-1]$
5. $i \leftarrow i-1$
6. $A[i] \leftarrow s$

Laufzeit $O(n)$



Einfügen(10)

Datenstrukturen

Löschen(*i*)

1. **for** $j \leftarrow i$ **to** $n-1$ **do**
2. $A[j] \leftarrow A[j+1]$
3. $A[n] \leftarrow \text{nil}$
4. $n \leftarrow n-1$

Parameter ist der
Index des zu
löschenden Objekts

2	4	6	7	11	13	nil	nil	nil	nil
					↑ n				

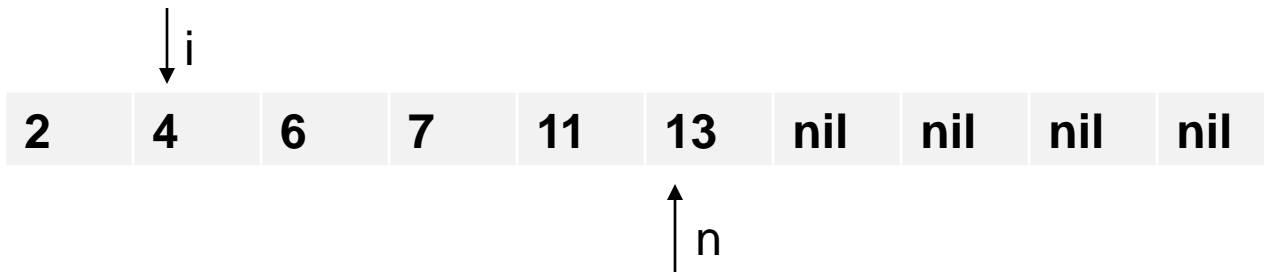
Löschen(2)

Datenstrukturen

Löschen(*i*)

1. **for** $j \leftarrow i$ **to** $n-1$ **do**
2. $A[j] \leftarrow A[j+1]$
3. $A[n] \leftarrow \text{nil}$
4. $n \leftarrow n-1$

Parameter ist der
Index des zu
löschenden Objekts

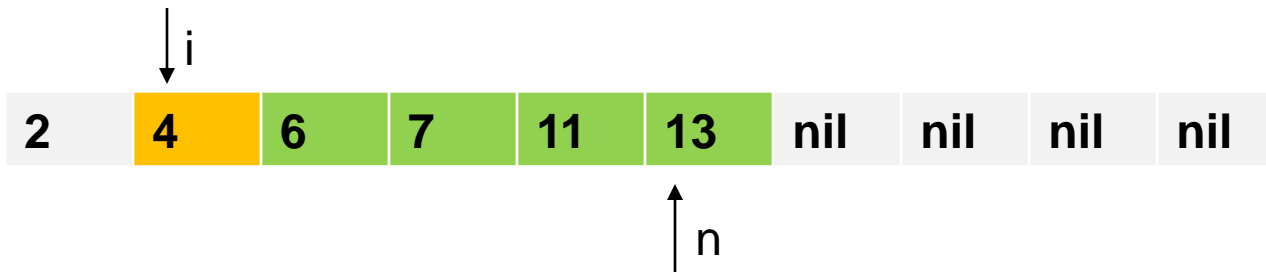


Löschen(2)

Datenstrukturen

Löschen(*i*)

1. **for** $j \leftarrow i$ **to** $n-1$ **do**
2. $A[j] \leftarrow A[j+1]$
3. $A[n] \leftarrow \text{nil}$
4. $n \leftarrow n-1$

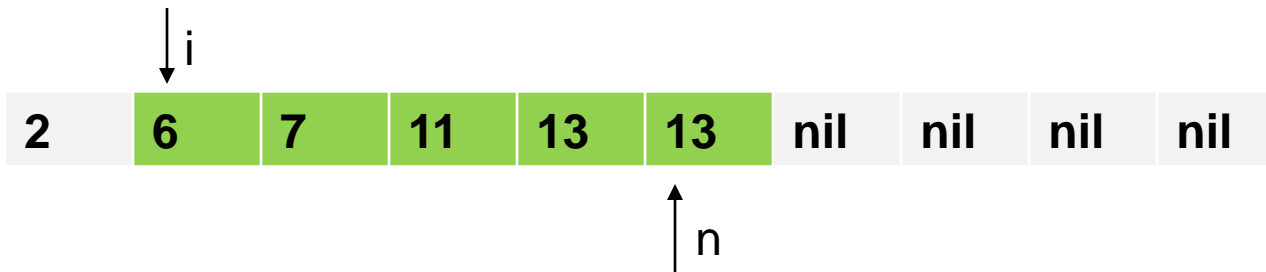


Löschen(2)

Datenstrukturen

Löschen(*i*)

1. **for** $j \leftarrow i$ **to** $n-1$ **do**
2. $A[j] \leftarrow A[j+1]$
3. $A[n] \leftarrow \text{nil}$
4. $n \leftarrow n-1$

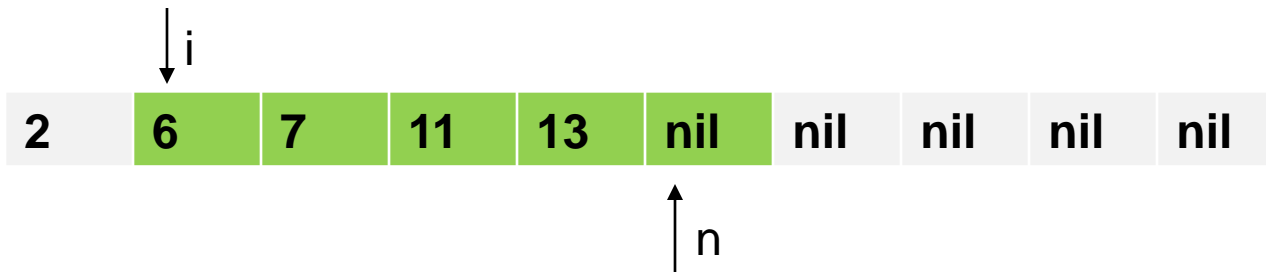


Löschen(2)

Datenstrukturen

Löschen(*i*)

1. **for** $j \leftarrow i$ **to** $n-1$ **do**
2. $A[j] \leftarrow A[j+1]$
3. $A[n] \leftarrow \text{nil}$
4. $n \leftarrow n-1$

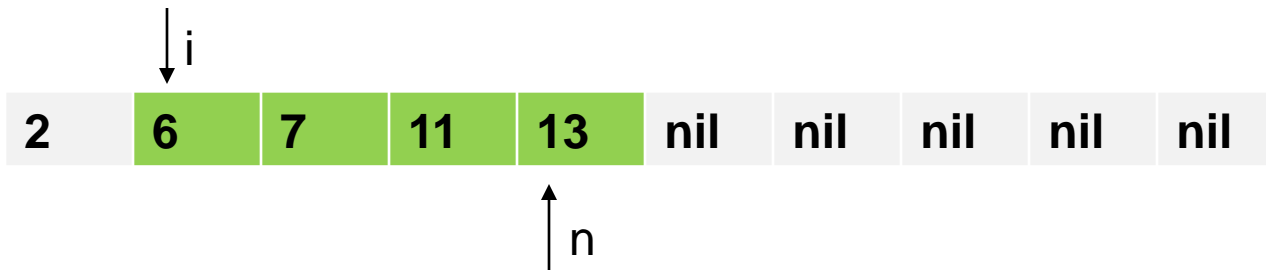


Löschen(2)

Datenstrukturen

Löschen(*i*)

1. **for** $j \leftarrow i$ **to** $n-1$ **do**
2. $A[j] \leftarrow A[j+1]$
3. $A[n] \leftarrow \text{nil}$
4. $n \leftarrow n-1$

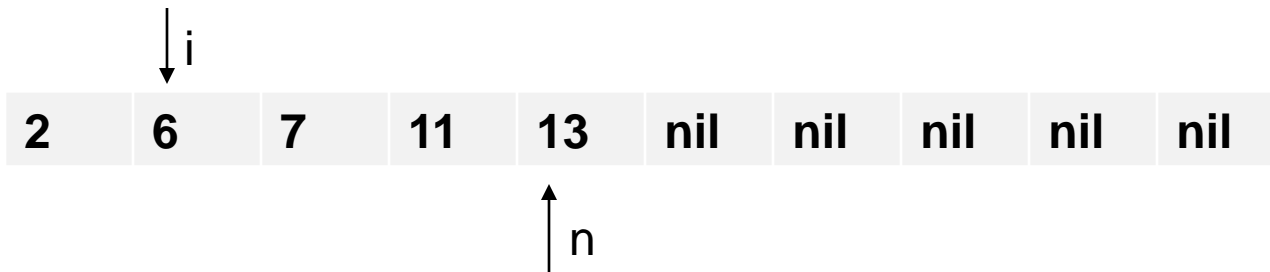


Löschen(2)

Datenstrukturen

Suchen(x)

- Binäre Suche
- Laufzeit $O(\log n)$



Löschen(2)

Datenstrukturen

Datenstruktur sortiertes Feld

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

Vorteile

- Schnelles Suchen

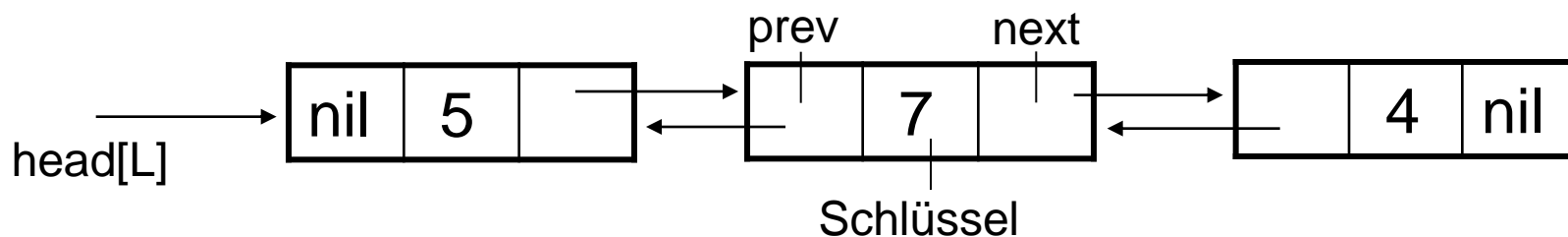
Nachteile

- Speicherbedarf abhängig von max (nicht vorhersagbar)
- Hohe Laufzeit für Einfügen/Löschen

Datenstrukturen

Doppelt verkettete Listen

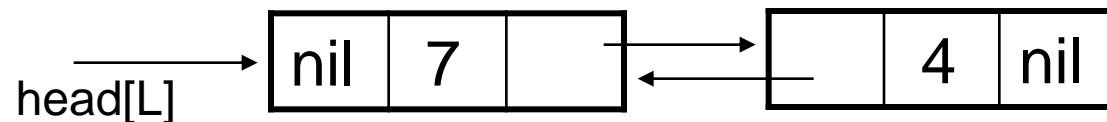
- Listenelement x ist Objekt bestehend aus **Schlüssel** und zwei Zeigern **prev** und **next**
- next verweist auf Nachfolger von x
- prev verweist auf Vorgänger von x
- prev/next sind **nil**, wenn Vorgänger/Nachfolger nicht existiert
- head[L] zeigt auf das erste Element



Datenstrukturen (siehe Vollversion)

Einfügen(L,x)

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



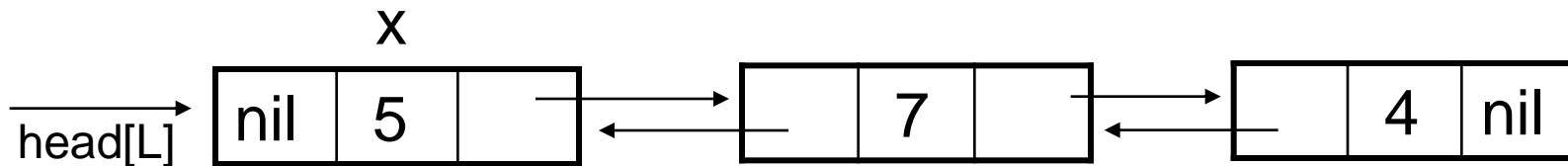
Datenstrukturen (siehe Vollversion)

Einfügen(L,x)

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

Laufzeit

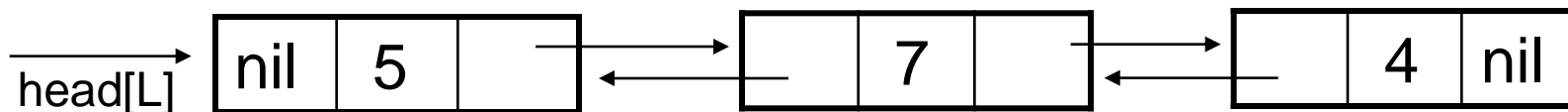
- $O(1)$



Datenstrukturen (siehe Vollversion)

Löschen(L,x)

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



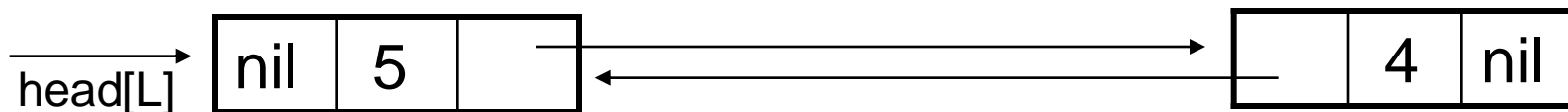
Datenstrukturen (siehe Vollversion)

Löschen(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]
4. **delete** x

Laufzeit

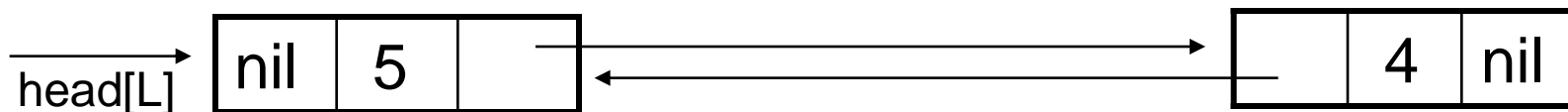
- O(1)



Datenstrukturen

Suche(L,k)

1. $x \leftarrow \text{head}[L]$
2. **while** $x \neq \text{nil}$ and $\text{key}[x] \neq k$ **do**
3. $x \leftarrow \text{next}[x]$
4. **return** x



Suche(L,4)

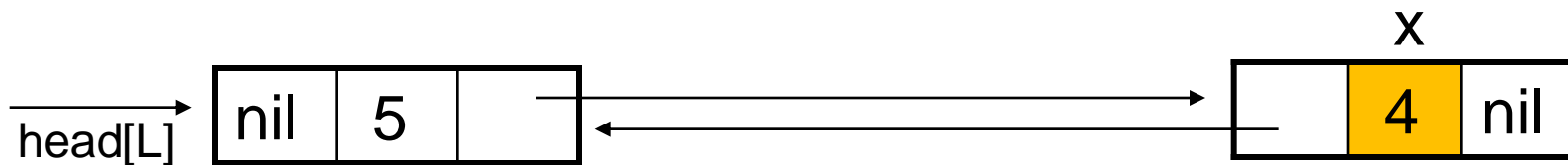
Datenstrukturen

Suche(L,k)

1. $x \leftarrow \text{head}[L]$
2. **while** $x \neq \text{nil}$ and $\text{key}[x] \neq k$ **do**
3. $x \leftarrow \text{next}[x]$
4. **return** x

Laufzeit

- $O(n)$



Suche(L,4)

Datenstrukturen

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

Datenstrukturen

Drei grundlegende Datenstrukturen

- Feld
- sortiertes Feld
- doppelt verkettete Liste

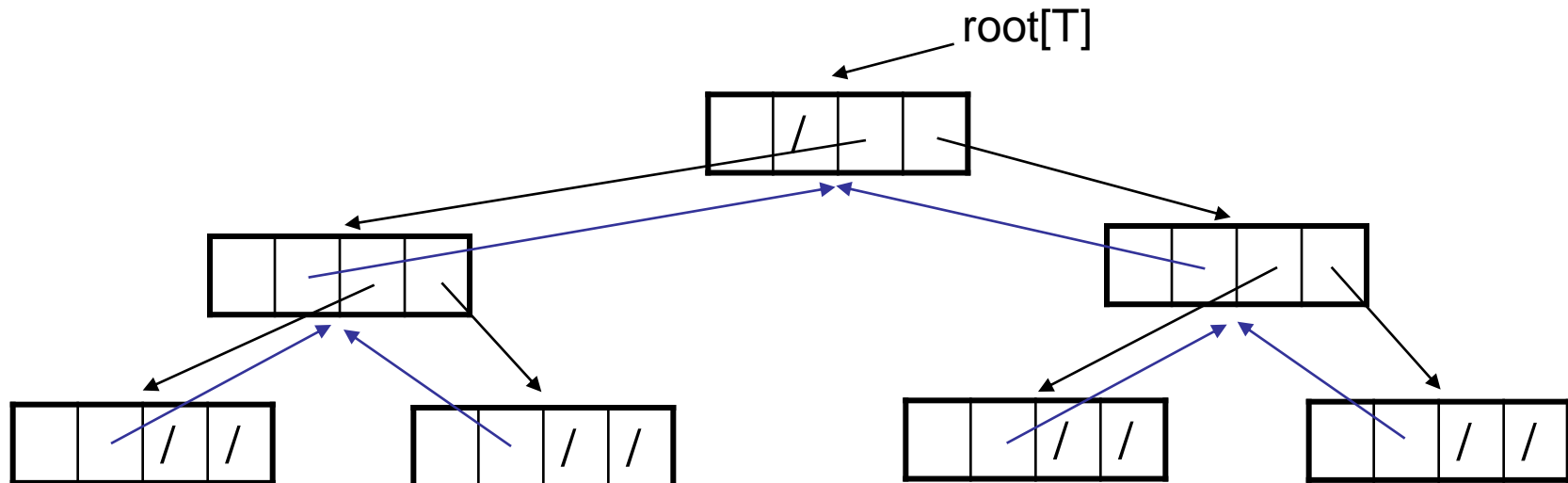
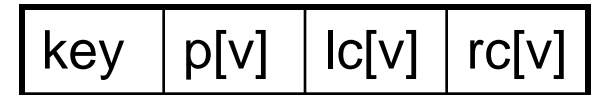
Diskussion

- Alle drei Strukturen haben gewichtige Nachteile
- Zeiger/Referenzen helfen beim Speichermanagement
- Sortierung hilft bei Suche ist aber teuer aufrecht zu erhalten

Datenstrukturen

Binärbäume

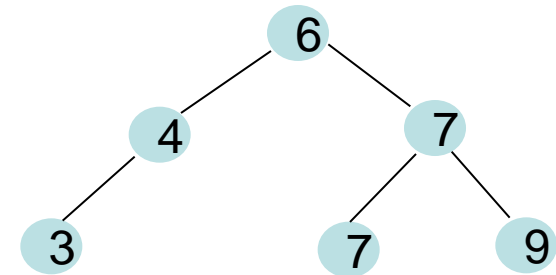
- Schlüssel key und ggf. weitere Daten
- Vaterzeiger $p[v]$ auf Vater von v (blau)
- Zeiger $lc[v]$ ($rc[v]$) auf linkes (rechtes) Kind von v
- Wurzelzeiger $root[T]$



Datenstrukturen

Binäre Suchbäume

- Verwende Binärbaum
- Speichere Schlüssel „geordnet“



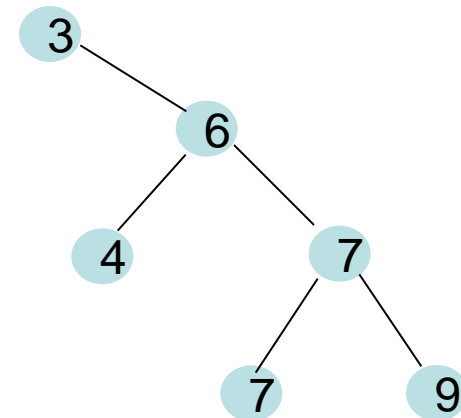
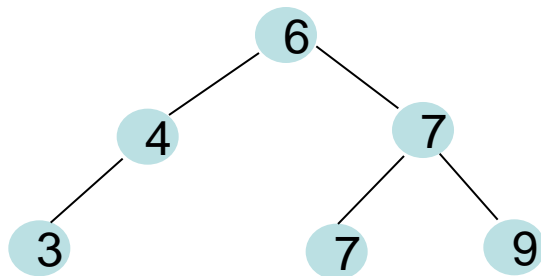
Binäre Suchbaumeigenschaft:

- Sei x Knoten im binären Suchbaum
- Ist y Knoten im **linken Unterbaum** von x, dann gilt $\text{key}[y] \leq \text{key}[x]$
- Ist y Knoten im **rechten Unterbaum** von x, dann gilt $\text{key}[y] > \text{key}[x]$

Datenstrukturen

Unterschiedliche Suchbäume

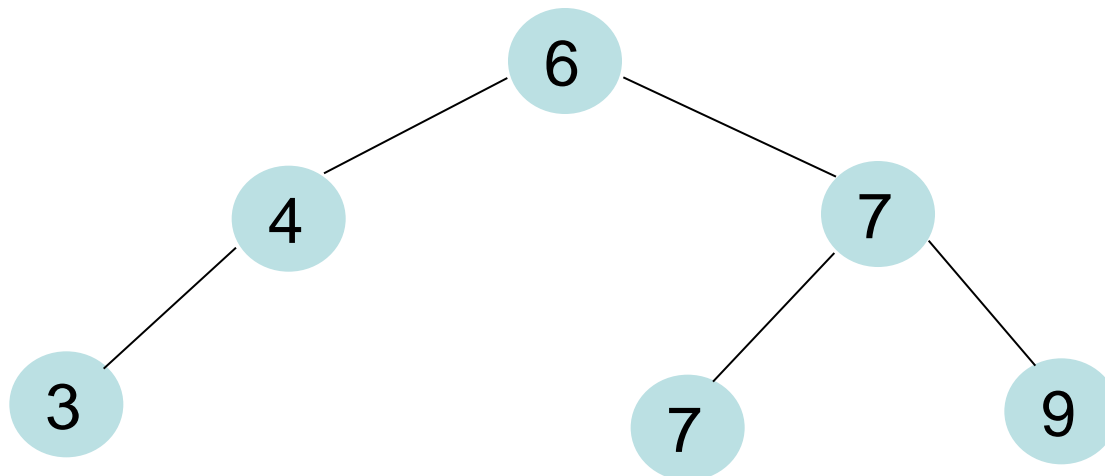
- Schlüsselmenge 3,4,6,7,7,9
- Wir erlauben mehrfache Vorkommen desselben Schlüssels



Datenstrukturen

Ausgabe aller Schlüssel

- Gegeben binärer Suchbaum
- Wie kann man alle Schlüssel aufsteigend sortiert in $\Theta(n)$ Zeit ausgeben?

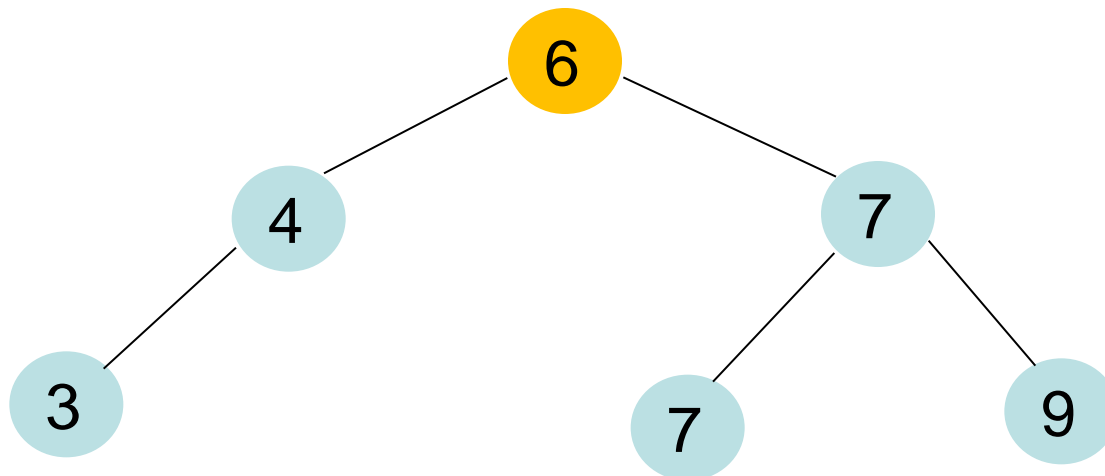


Datenstrukturen

Inorder-Tree-Walk(x)

1. **if** $x \neq \text{nil}$ **then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Aufruf über
Inorder-Tree-Walk(root[T])



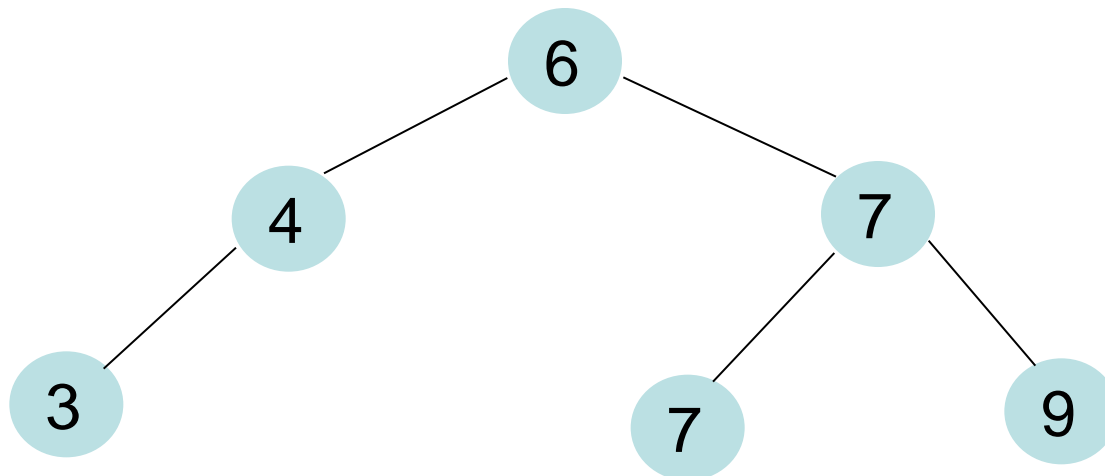
Datenstrukturen

Inorder-Tree-Walk(x)

1. **if** $x \neq \text{nil}$ **then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:

3, 4, 6, 7, 7, 9



Datenstrukturen

Lemma 37

- Inorder-Tree-Walk gibt die Schlüssel eines binären Suchbaums in aufsteigender Reihenfolge aus.

Datenstrukturen

„Normale“ Induktion

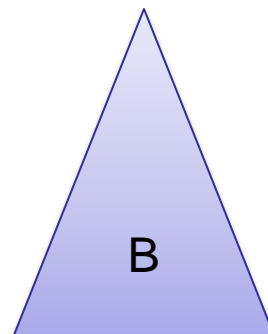
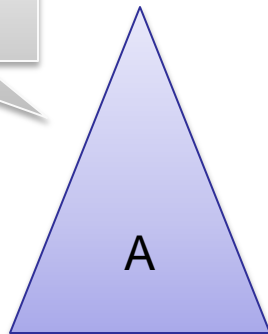
- Wir wollen zeigen, dass Aussage $A(i)$ für alle natürlichen Zahlen i gilt
- Dazu beweisen wir, dass
 - *(a) $A(1)$ gilt*
 - *(b) Wenn $A(i)$ gilt, dann gilt auch $A(i+1)$*
- (a) heißt Induktionsanfang
- (b) nennt man Induktionsschluss (oder auch Induktionsschritt)
- Die Voraussetzung in (b) (also $A(i)$) heißt Induktionsvoraussetzung

Datenstrukturen

Induktion über die Struktur von Binärbäumen

- Wollen zeigen, dass Aussage für alle Binärbäume gilt:
- (a) Zeige Induktionsanfang für „kleine Binärbäume“
- (b) Setze größere Bäume aus kleinen Binärbäumen zusammen, d.h.

Aussage gilt
für Bäume A
und B

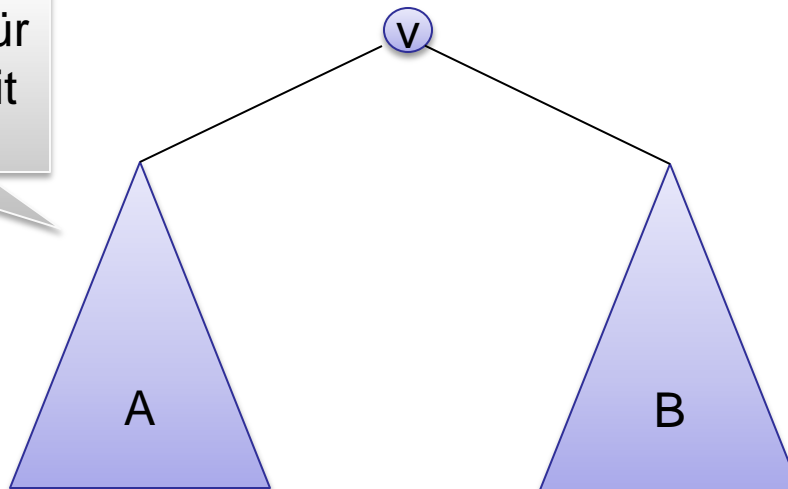


Datenstrukturen

Induktion über die Struktur von Binärbäumen

- Wollen zeigen, dass Aussage für alle Binärbäume gilt:
- (a) Zeige Induktionsanfang für „kleine Binärbäume“
- (b) Setze größere Bäume aus kleinen Binärbäumen zusammen, d.h.

Dann gilt
Aussage auch für
neuen Baum mit
Wurzel v



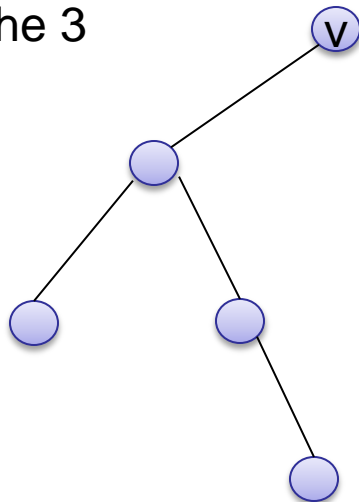
Datenstrukturen

Definition

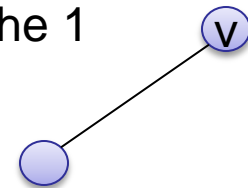
- Die Höhe eines Binärbaums mit Wurzel v ist die Länge (Anzahl Kanten) des längsten einfachen Weges (keine mehrfach vorkommenden Knoten) von der Wurzel zu einem Blatt.

Beispiel

- Baum der Höhe 3



- Baum der Höhe 1



- Baum der Höhe 0



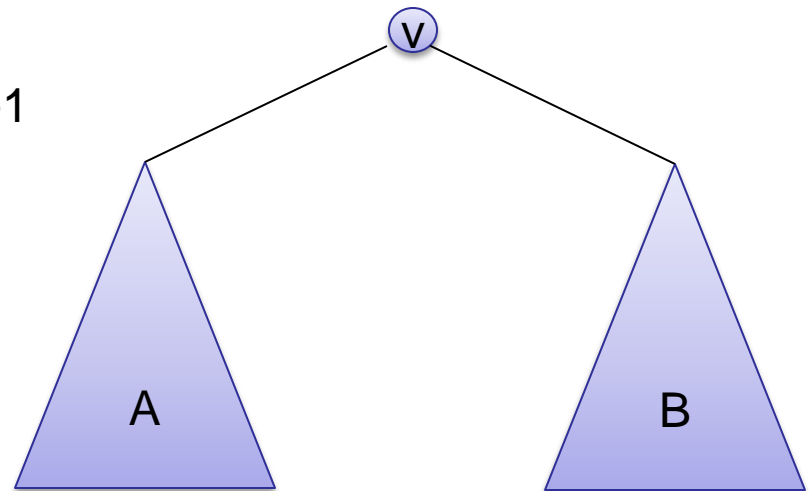
Datenstrukturen

Definition

- Die Höhe eines Binärbaums mit Wurzel v ist die Länge (Anzahl Kanten) des längsten einfachen Weges (keine mehrfach vorkommenden Knoten) von der Wurzel zu einem Blatt.

Beispiel

- Übereinkunft: Ein leerer Baum hat Höhe -1
- Damit gilt:
Höhe eines Baumes mit Wurzel v und Teilbäumen A und B ist
 $1 + \max\{\text{Höhe}(A), \text{Höhe}(B)\}$

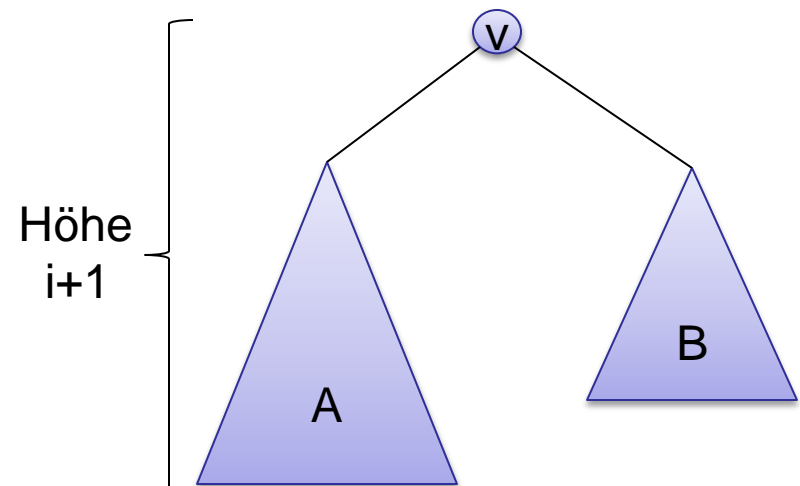


Datenstrukturen

Induktion über die Struktur von Binärbäumen

- Wir wollen Aussage $A(i)$ durch Induktion über die Höhe von Bäumen zeigen
- (a) Zeige die Aussage für leere Bäume (Bäume der Höhe -1)
- (b) Zeige: Gilt die Aussage für Bäume der Höhe i , so gilt sie auch für Bäume der Höhe $i+1$

- Dabei können wir immer annehmen, dass ein Baum der Höhe $i+1$ aus einer Wurzel v und zwei Teilbäumen A, B besteht, so dass
 - (1) A und B Höhe maximal i haben und
 - (2) A oder B Höhe i hat

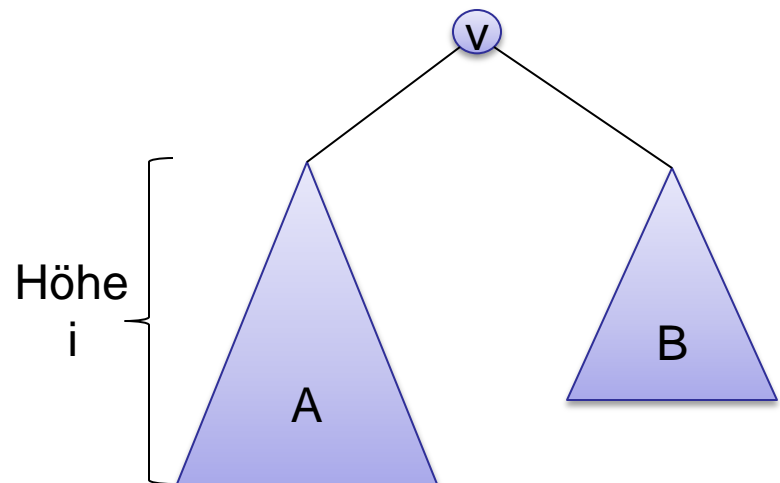


Datenstrukturen

Induktion über die Struktur von Binärbäumen

- Wir wollen Aussage $A(i)$ durch Induktion über die Höhe von Bäumen zeigen
- (a) Zeige die Aussage für leere Bäume (Bäume der Höhe -1)
- (b) Zeige: Gilt die Aussage für Bäume der Höhe i , so gilt sie auch für Bäume der Höhe $i+1$

- Dabei können wir immer annehmen, dass ein Baum der Höhe $i+1$ aus einer Wurzel v und zwei Teilbäumen A, B besteht, so dass
 - (1) A und B Höhe maximal i haben und
 - (2) A oder B Höhe i hat



Datenstrukturen

Lemma 37

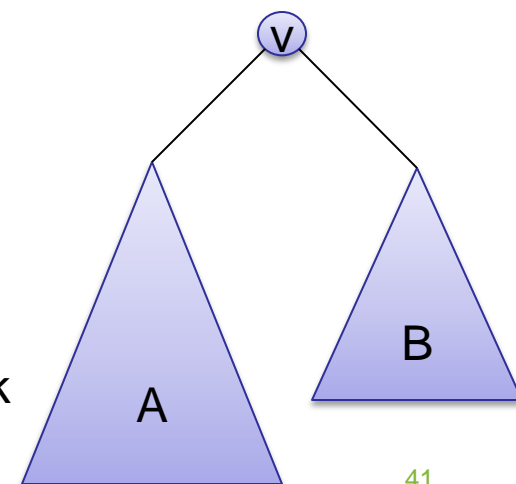
- Inorder-Tree-Walk gibt die Schlüssel eines binären Suchbaums in aufsteigender Reihenfolge aus.

Beweis

- (I.A.) Ist die Eingabe ein leerer Baum, so wird Inorder-Tree-Walk nil übergeben. Damit bricht der Algorithmus sofort ab. Es gibt also keine Ausgabe, was auch korrekt ist.
- (I.V.) Das Lemma gilt für Bäume der Höhe $\leq i$.
- (I.S.) Wir müssen zeigen, dass das Lemma auch für Höhe $i+1 \geq 0$. Dazu betrachten wir den Inorder-Tree-Walk auf einem solchem Baum.

Inorder-Tree-Walk(x)

1. **if** $x \neq \text{nil}$ **then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])



Datenstrukturen

Lemma 37

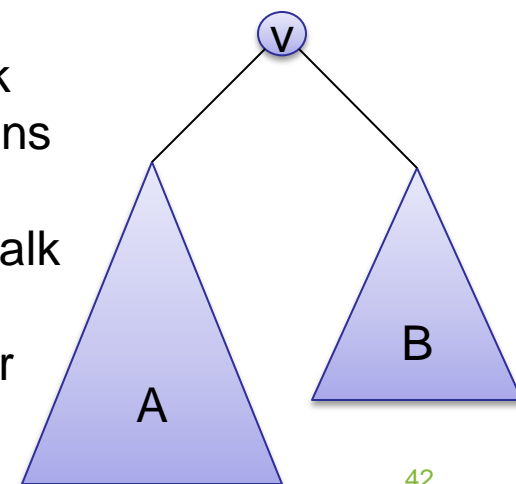
- Inorder-Tree-Walk gibt die Schlüssel eines binären Suchbaums in aufsteigender Reihenfolge aus.

Beweis

- (I.S.) Wir müssen zeigen, dass das Lemma auch für Höhe $i+1 \geq 0$. Dazu betrachten wir den Inorder-Tree-Walk auf einem solchem Baum. Da der Baum Höhe mindestens 0 hat, wird der Algorithmus nicht mit nil aufgerufen. Damit wird Zeile 2 ausgeführt. Dort wird Inorder-Tree-Walk rekursiv mit dem Teilbaum A der Höhe $\leq i$ ausgeführt. Nach (I.V.) werden die Schlüssel aus A in aufsteigender Reihenfolge ausgegeben

Inorder-Tree-Walk(x)

1. **if** $x \neq \text{nil}$ **then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])



Datenstrukturen

Lemma 37

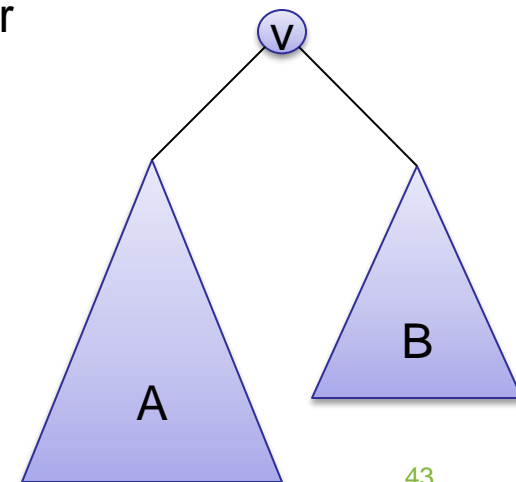
- Inorder-Tree-Walk gibt die Schlüssel eines binären Suchbaums in aufsteigender Reihenfolge aus.

Beweis

- Nach (I.V.) werden die Schlüssel aus A in aufsteigender Reihenfolge ausgegeben. Dann wird in Zeile 3 $\text{key}[v]$ ausgegeben. Wegen der Sucheigenschaft ist $\text{key}[v]$ größer gleich allen Schlüssel in A und kleiner als alle Schlüssel in B. In Zeile 4 wird dann Inorder-Tree-Walk rekursiv für Teilbaum B aufgerufen. Die dort gespeicherten Schlüssel werden nach (I.V.) ebenfalls in aufsteigender Reihenfolge ausgegeben.

Inorder-Tree-Walk(x)

1. **if** $x \neq \text{nil}$ **then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe $\text{key}[x]$
4. Inorder-Tree-Walk(rc[x])



Datenstrukturen

Lemma 37

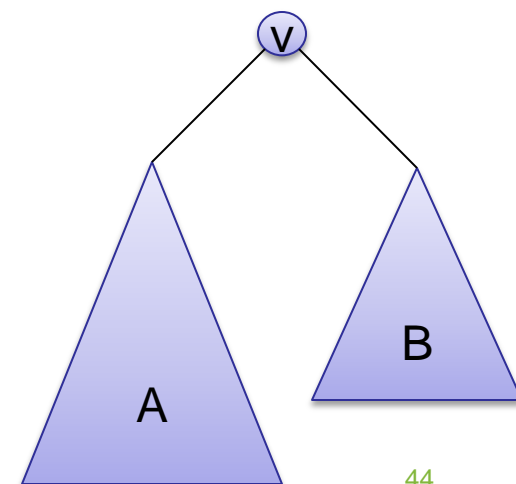
- Inorder-Tree-Walk gibt die Schlüssel eines binären Suchbaums in aufsteigender Reihenfolge aus.

Beweis

- Zusammenfassend werden also erst die Schlüssel in Teilbaum A in aufsteigender Reihenfolge ausgegeben. Dann folgt $\text{key}[v]$ und dann die Schlüssel aus B in aufsteigender Reihenfolge. Aufgrund der Suchbaumeigenschaft ist die gesamte Folge aufsteigend sortiert.

Inorder-Tree-Walk(x)

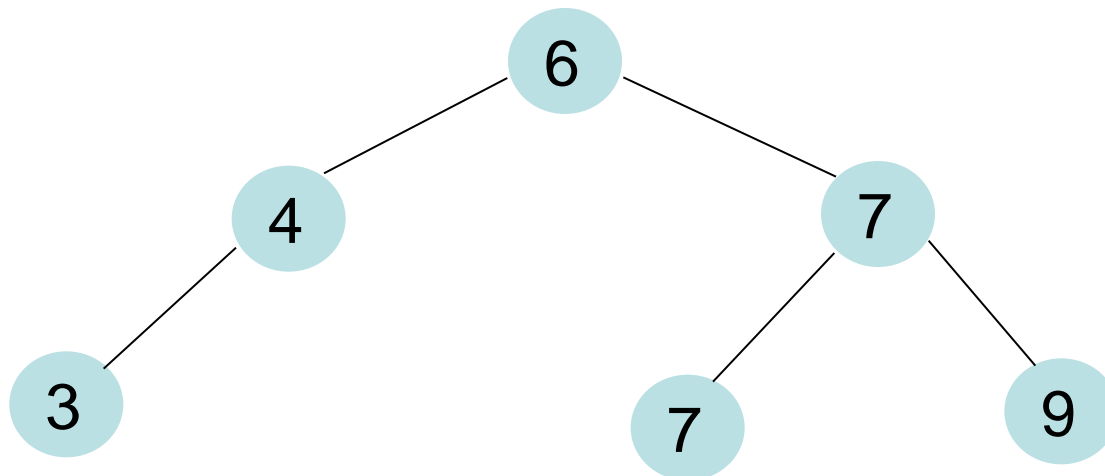
1. **if** $x \neq \text{nil}$ **then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe $\text{key}[x]$
4. Inorder-Tree-Walk(rc[x])



Datenstrukturen

Suchen in Binärbäumen

- Gegeben ist Schlüssel k
- Gesucht ist ein Knoten mit Schlüssel k

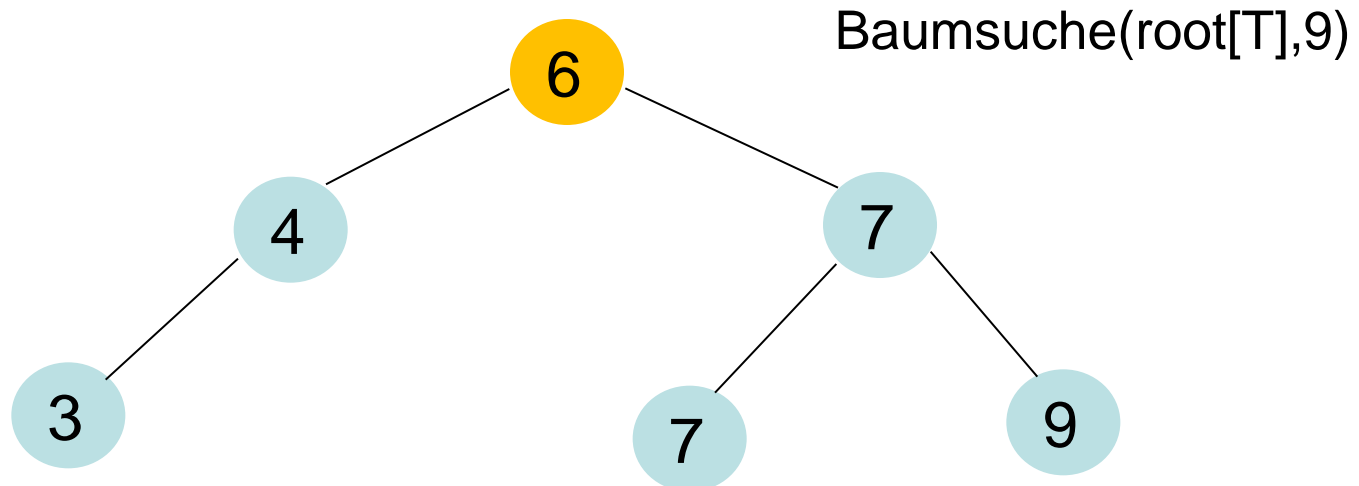


Datenstrukturen

Baumsuche(x,k)

Aufruf mit
x=root[T]

1. **if** x=nil **or** k=key[x] **then return** x
2. **if** k<key[x] **then return** Baumsuche(lc[x],k)
3. **else return** Baumsuche(rc[x],k)

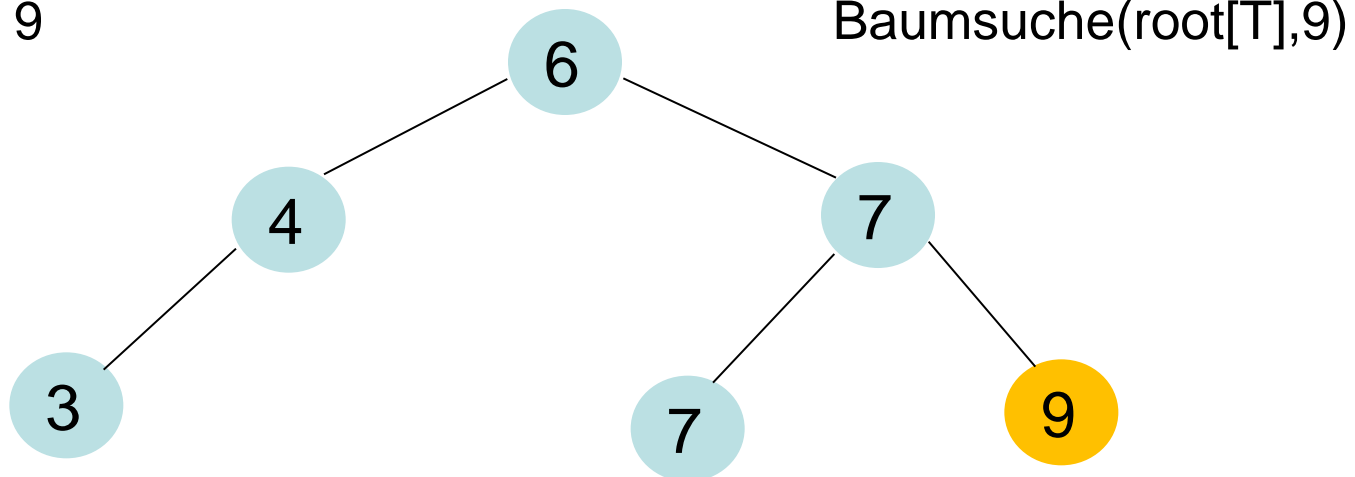


Datenstrukturen (siehe Vollversion)

Baumsuche(x,k)

1. **if** $x = \text{nil}$ **or** $k = \text{key}[x]$ **then return** x
2. **if** $k < \text{key}[x]$ **then return** $\text{Baumsuche}(\text{lc}[x], k)$
3. **else** **return** $\text{Baumsuche}(\text{rc}[x], k)$

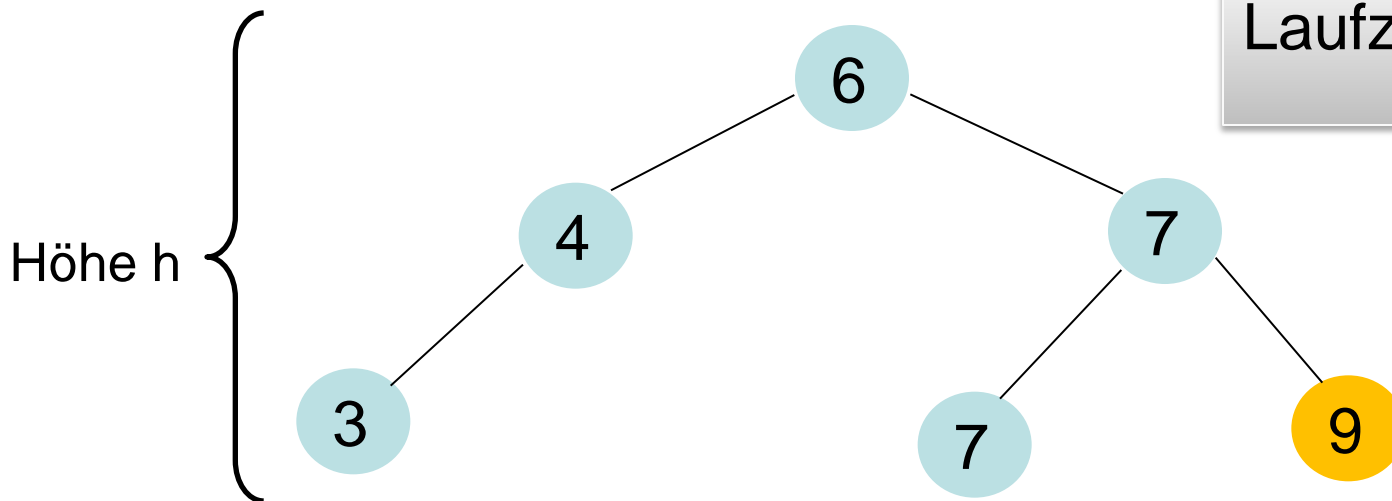
Ausgabe: 9



Datenstrukturen (siehe Vollversion)

Baumsuche(x,k)

1. **if** $x = \text{nil}$ **or** $k = \text{key}[x]$ **then return** x
2. **if** $k < \text{key}[x]$ **then return** $\text{Baumsuche}(\text{lc}[x], k)$
3. **else** **return** $\text{Baumsuche}(\text{rc}[x], k)$

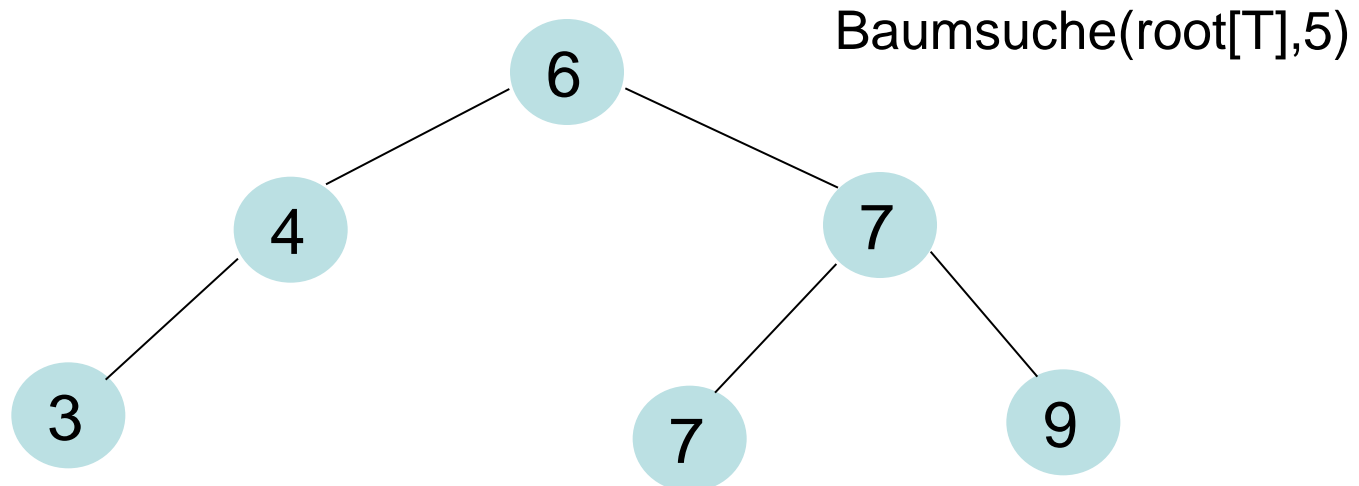


Datenstrukturen (siehe Vollversion)

IterativeBaumsuche(x,k)

1. **while** $x \neq \text{nil}$ and $k \neq \text{key}[x]$ **do**
2. **if** $k < \text{key}[x]$ **then** $x \leftarrow \text{lc}[x]$
3. **else** $x \leftarrow \text{rc}[x]$
4. **return** x

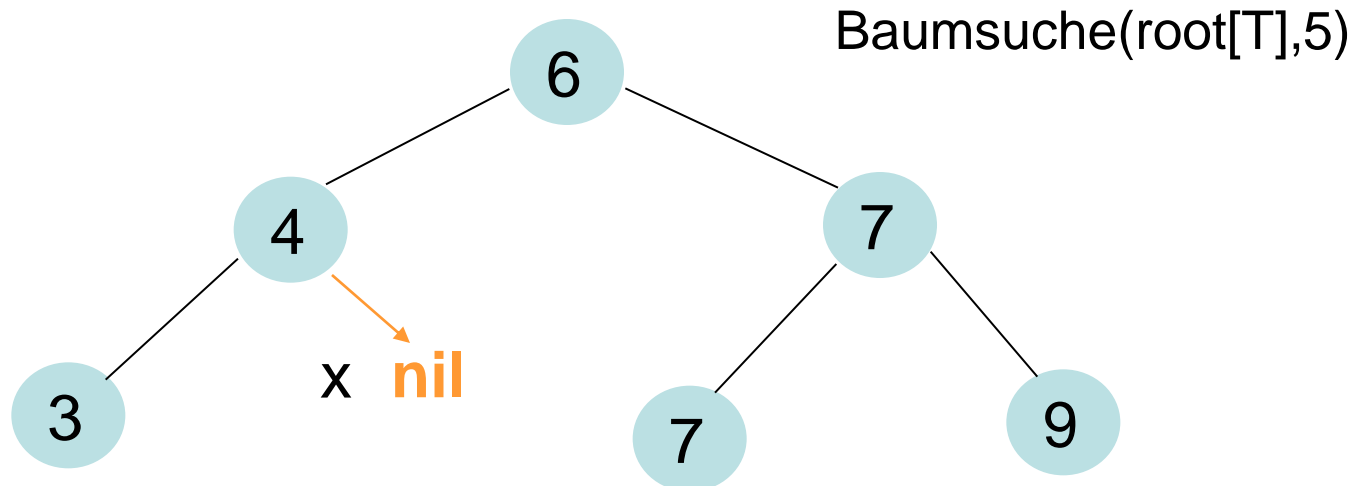
Aufruf mit
 $x = \text{root}[T]$



Datenstrukturen (siehe Vollversion)

IterativeBaumsuche(x,k)

1. **while** $x \neq \text{nil}$ and $k \neq \text{key}[x]$ **do**
2. **if** $k < \text{key}[x]$ **then** $x \leftarrow \text{lc}[x]$
3. **else** $x \leftarrow \text{rc}[x]$
4. **return** x

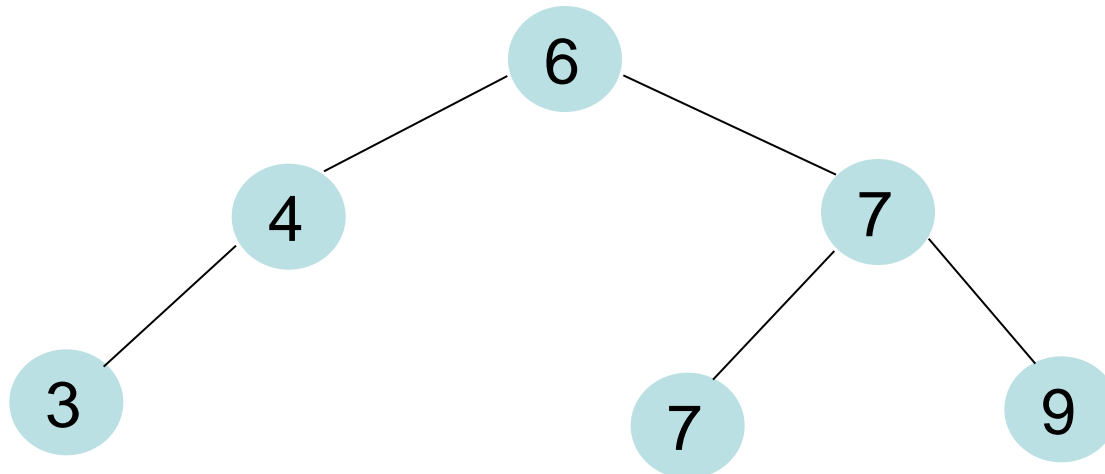


Datenstrukturen (siehe Vollversion)

IterativeBaumsuche(x,k)

1. **while** $x \neq \text{nil}$ and $k \neq \text{key}[x]$ **do**
2. **if** $k < \text{key}[x]$ **then** $x \leftarrow \text{lc}[x]$
3. **else** $x \leftarrow \text{rc}[x]$
4. **return** x

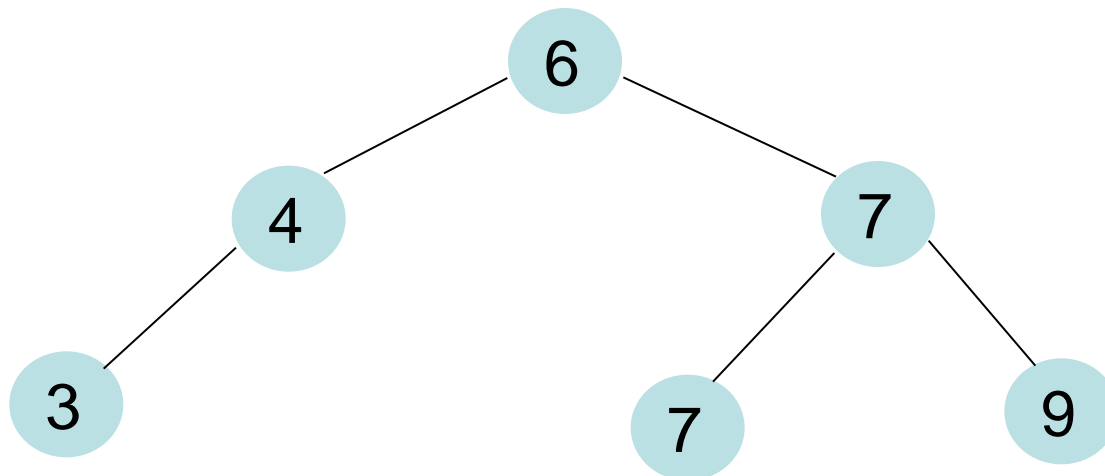
Funktionsweise wie (rekursive)
Baumsuche. Laufzeit ebenfalls $O(h)$.



Datenstrukturen

Minimum- und Maximumsuche

- Suchbaumeigenschaft:
Alle Knoten im rechten Unterbaum eines Knotens x sind größer $\text{key}[x]$
- Alle Knoten im linken Unterbaum von x sind $\leq \text{key}[x]$



Datenstrukturen

Wird mit Wurzel aufgerufen

Laufzeit $O(h)$

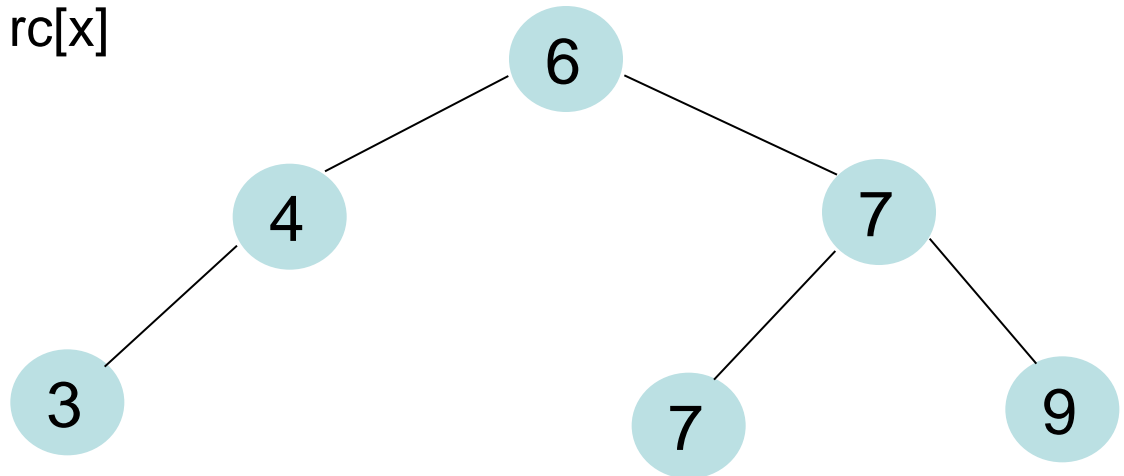
MinimumSuche(x)

1. **while** lc[x]≠nil **do** x ← lc[x]
2. **return** x

Wird mit Wurzel aufgerufen

MaximumSuche(x)

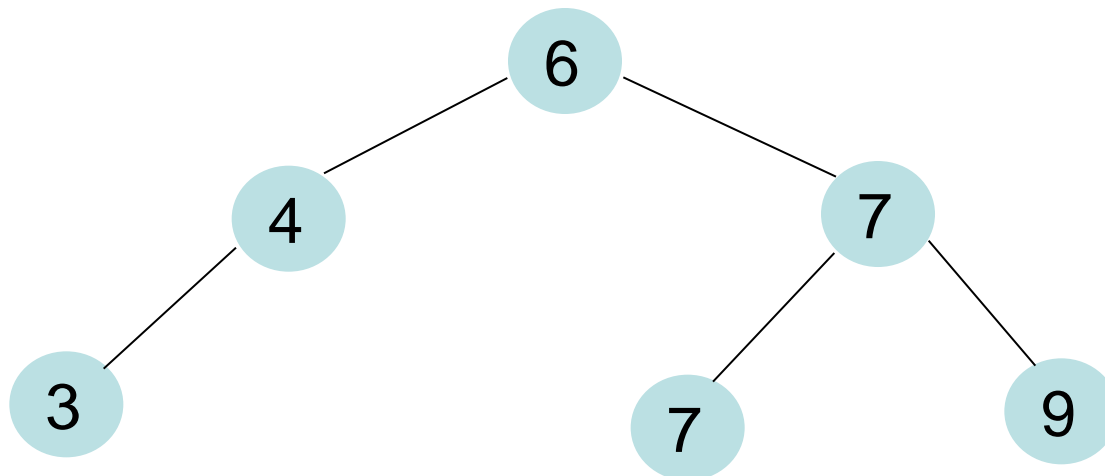
1. **while** rc[x]≠nil **do** x ← rc[x]
2. **return** x



Datenstrukturen

Nachfolgersuche

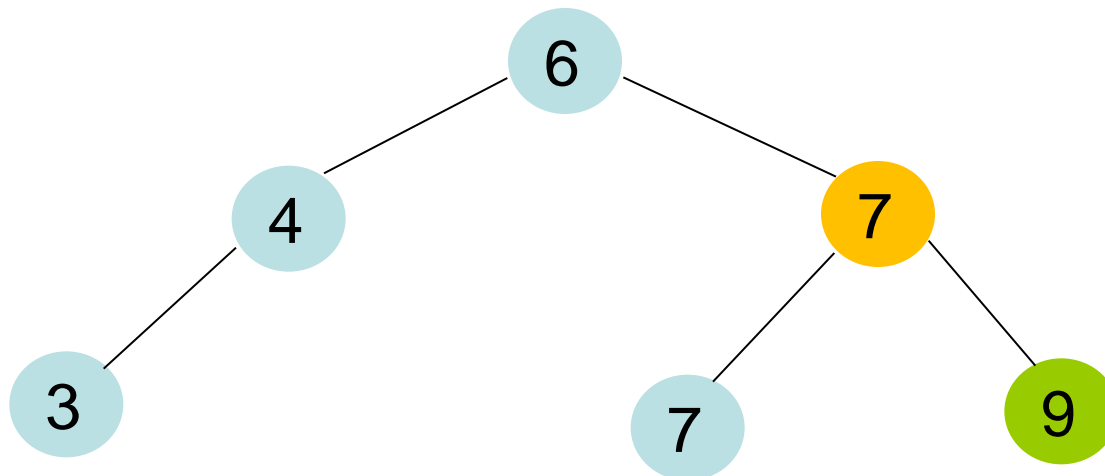
- Nachfolger bzgl. Inorder-Tree-Walk
- Wenn alle Schlüssel unterschiedlich, dann ist das der nächstgrößere Schlüssel



Datenstrukturen

Nachfolgersuche

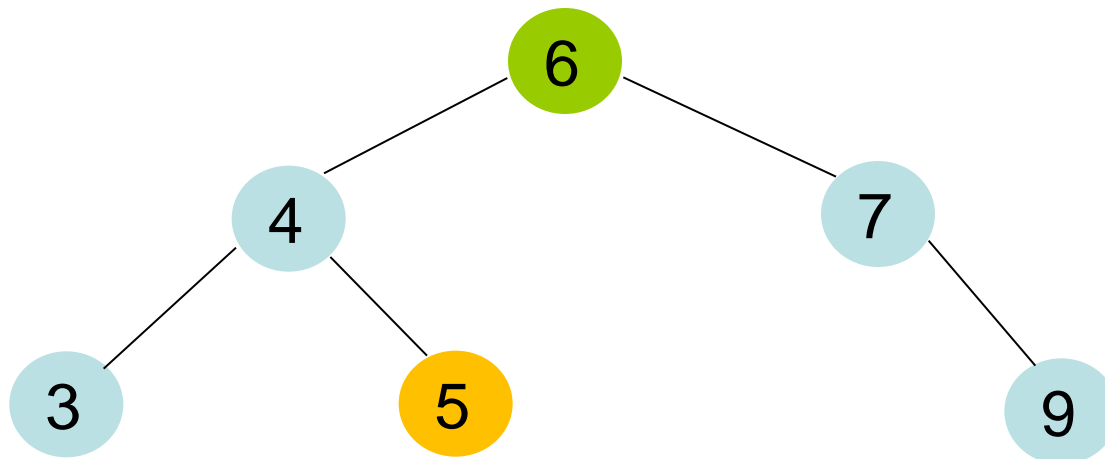
- Fall 1 (rechter Unterbaum von x nicht leer):
Dann ist der linkeste Knoten im rechten Unterbaum der Nachfolger von x



Datenstrukturen

Nachfolgersuche

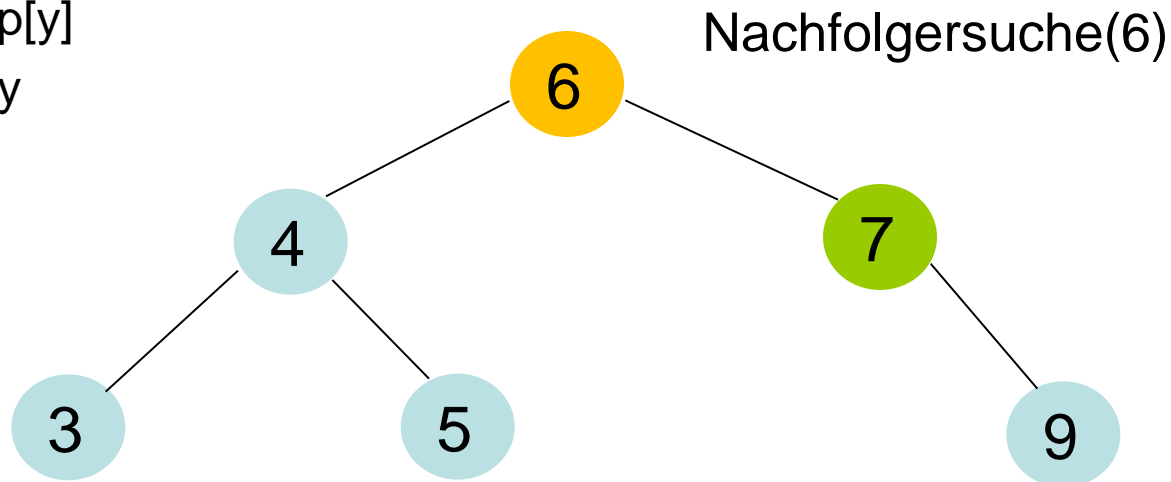
- Fall 2 (rechter Unterbaum von x leer und x hat Nachfolger y):
Dann ist y der erste Knoten auf dem Pfad zur Wurzel, der größer als x ist



Datenstrukturen (siehe Vollversion)

Nachfolgersuche(x)

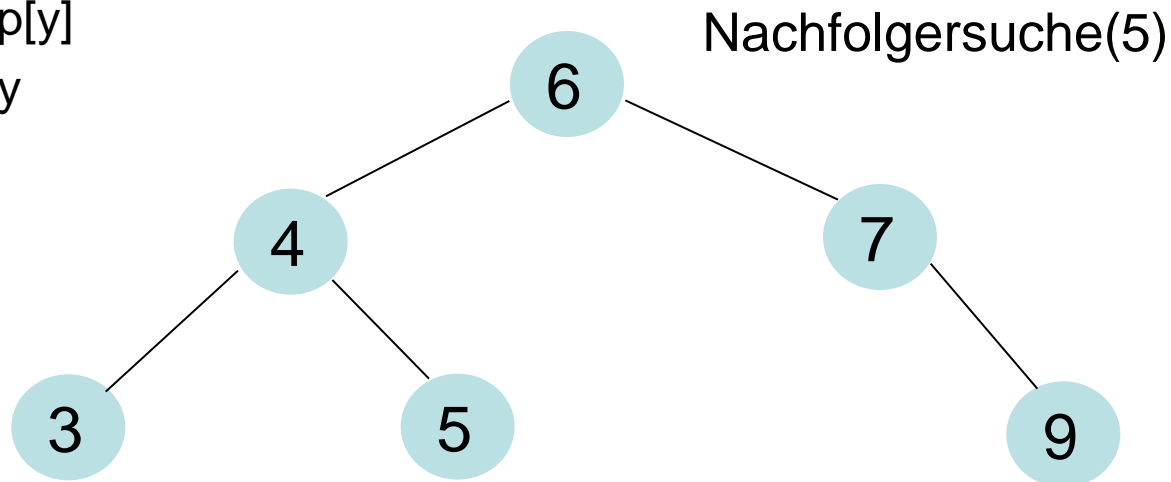
1. **if** rc[x] \neq nil **then return** MinimumSuche(rc[x])
2. $y \leftarrow p[x]$
3. **while** y \neq nil and x=rc[y] **do**
4. $x \leftarrow y$
5. $y \leftarrow p[y]$
6. **return** y



Datenstrukturen (siehe Vollversion)

Nachfolgersuche(x)

1. **if** rc[x] \neq nil **then return** MinimumSuche(rc[x])
2. $y \leftarrow p[x]$
3. **while** y \neq nil and x=rc[y] **do**
4. $x \leftarrow y$
5. $y \leftarrow p[y]$
6. **return** y

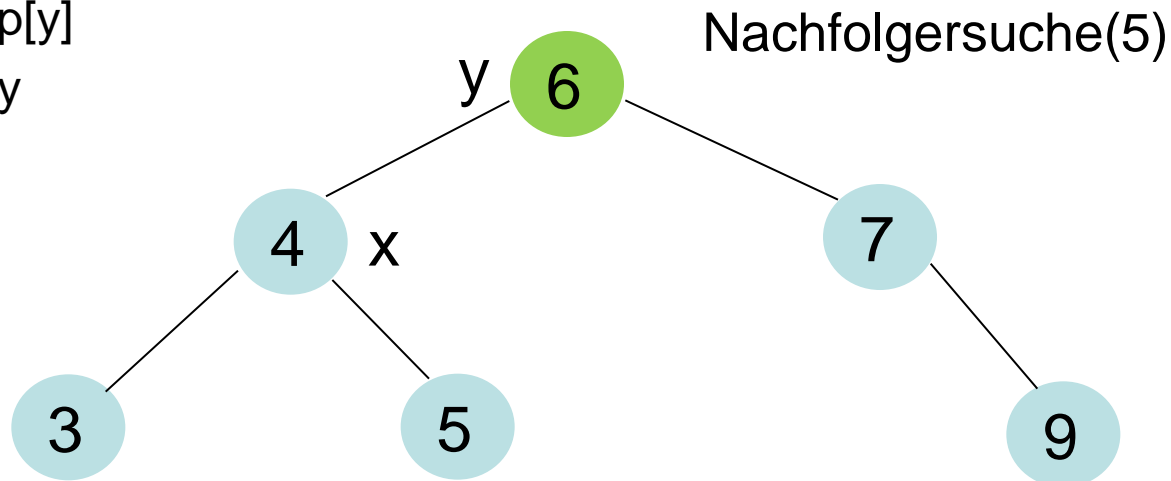


Datenstrukturen (siehe Vollversion)

Nachfolgersuche(x)

1. **if** rc[x] \neq nil **then return** MinimumSuche(rc[x])
2. $y \leftarrow p[x]$
3. **while** y \neq nil and x=rc[y] **do**
4. $x \leftarrow y$
5. $y \leftarrow p[y]$
6. **return** y

Laufzeit $O(h)$

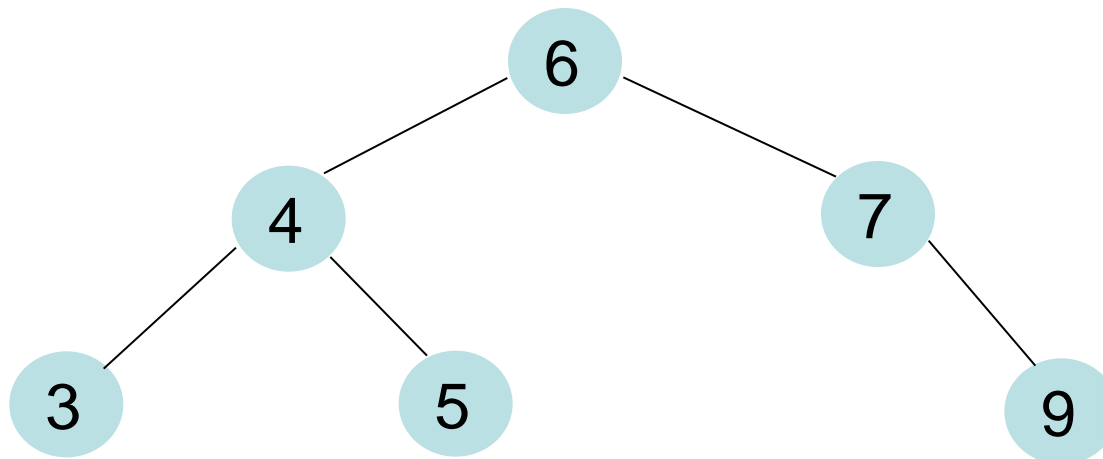


Datenstrukturen

Vorgängersuche

- Symmetrisch zu Nachfolgersuche
- Daher ebenfalls $O(h)$ Laufzeit

Laufzeit $O(h)$



Datenstrukturen

Binäre Suchbäume

- Aufzählen der Elemente mit Inorder-Tree-Walk in $O(n)$ Zeit
- Suche in $O(h)$ Zeit
- Minimum/Maximum in $O(h)$ Zeit
- Vorgänger/Nachfolger in $O(h)$ Zeit

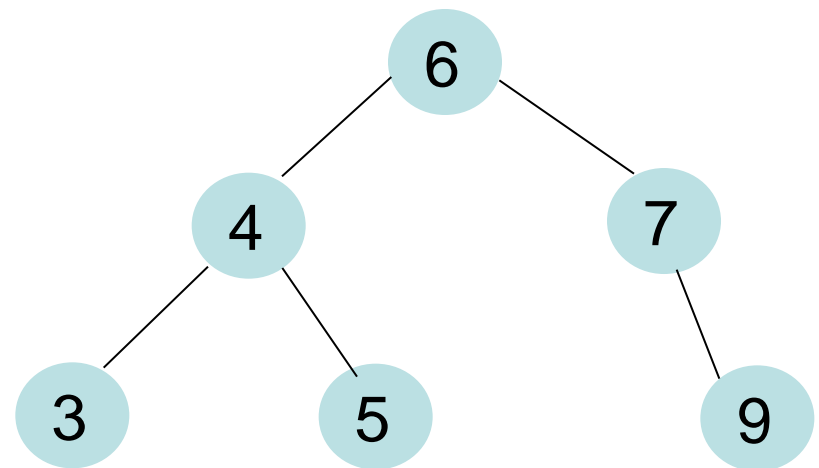
Dynamische Operationen?

- Einfügen und Löschen
- Müssen Suchbaumeigenschaft aufrecht erhalten
- Auswirkung auf Höhe des Baums?

Datenstrukturen

Einfügen

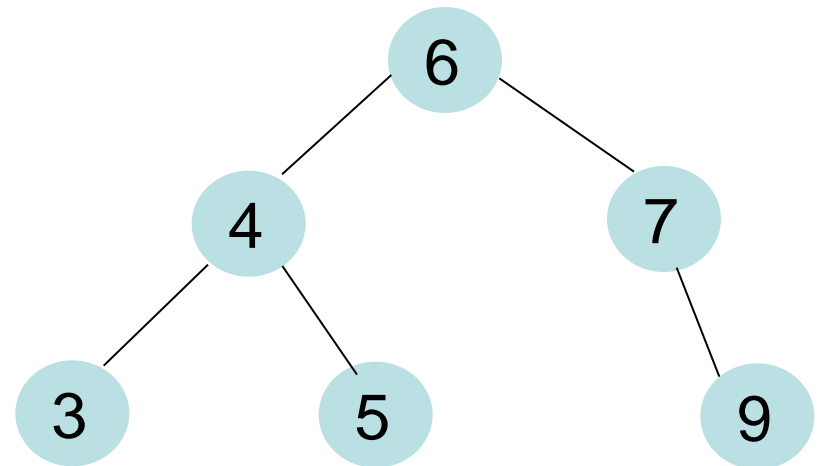
- Ähnlich wie Baumsuche: Finde Blatt, an das neuer Knoten angehängt wird
- Danach wird **nil**-Zeiger durch neues Element ersetzt



Datenstrukturen (siehe Vollversion)

Einfügen(T,z)

1. $y \leftarrow \text{nil}; x \leftarrow \text{root}[T]$
2. **while** $x \neq \text{nil}$ **do**
3. $y \leftarrow x$
4. **if** $\text{key}[z] < \text{key}[x]$ **then** $x \leftarrow \text{lc}[x]$
5. **else** $x \leftarrow \text{rc}[x]$
6. $p[z] \leftarrow y$
7. **if** $y = \text{nil}$ **then** $\text{root}[T] \leftarrow z$
8. **else**
9. **if** $\text{key}[z] < \text{key}[y]$ **then** $\text{lc}[y] \leftarrow z$
10. **else** $\text{rc}[y] \leftarrow z$



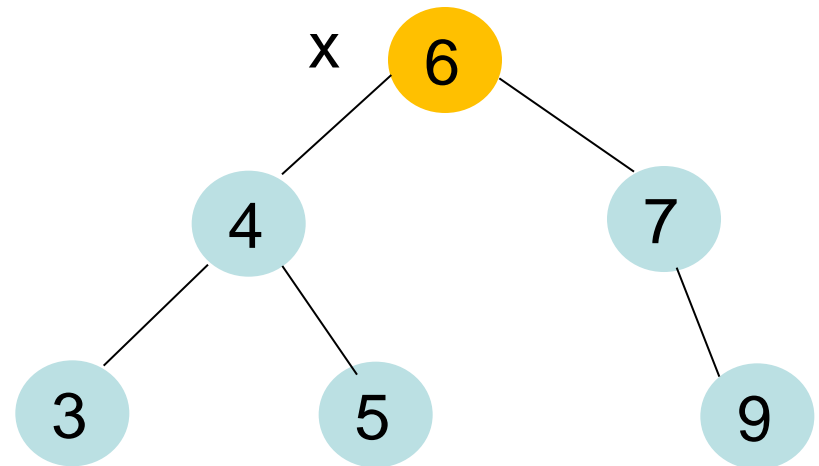
Datenstrukturen (siehe Vollversion)

Einfügen(T,z)

1. $y \leftarrow \text{nil}; x \leftarrow \text{root}[T]$
2. **while** $x \neq \text{nil}$ **do**
3. $y \leftarrow x$
4. **if** $\text{key}[z] < \text{key}[x]$ **then** $x \leftarrow \text{lc}[x]$
5. **else** $x \leftarrow \text{rc}[x]$
6. $p[z] \leftarrow y$
7. **if** $y = \text{nil}$ **then** $\text{root}[T] \leftarrow z$
8. **else**
9. **if** $\text{key}[z] < \text{key}[y]$ **then** $\text{lc}[y] \leftarrow z$
10. **else** $\text{rc}[y] \leftarrow z$

y wird Vater des einzufügenden Elements

Einfügen(8)

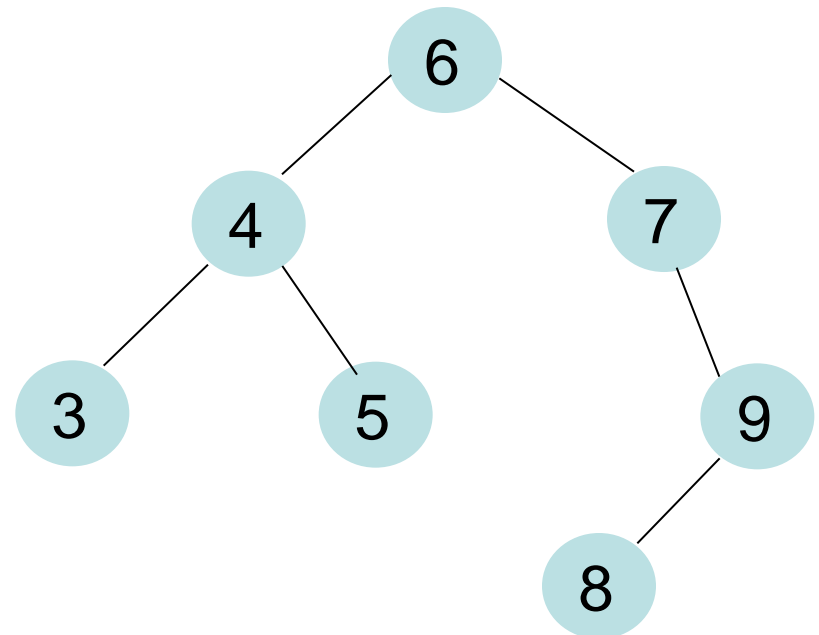


Datenstrukturen (siehe Vollversion)

Einfügen(T,z)

1. $y \leftarrow \text{nil}$; $x \leftarrow \text{root}[T]$
2. **while** $x \neq \text{nil}$ **do**
3. $y \leftarrow x$
4. **if** $\text{key}[z] < \text{key}[x]$ **then** $x \leftarrow \text{lc}[x]$
5. **else** $x \leftarrow \text{rc}[x]$
6. $p[z] \leftarrow y$
7. **if** $y = \text{nil}$ **then** $\text{root}[T] \leftarrow z$
8. **else**
9. **if** $\text{key}[z] < \text{key}[y]$ **then** $\text{lc}[y] \leftarrow z$
10. **else** $\text{rc}[y] \leftarrow z$

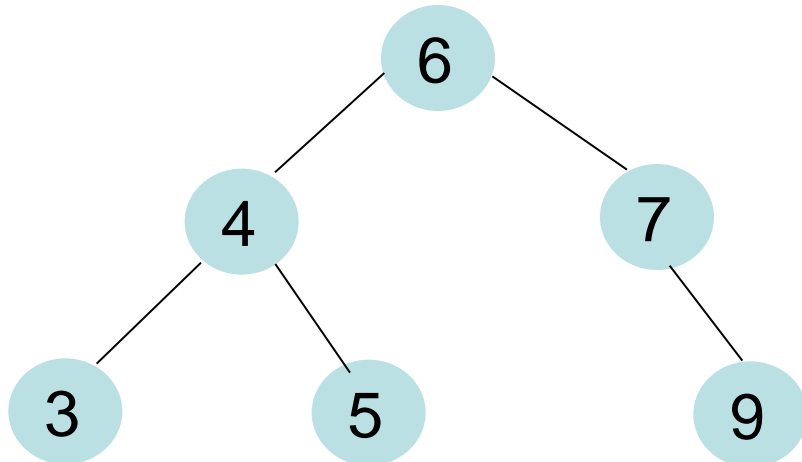
Laufzeit $O(h)$



Datenstrukturen

Löschen

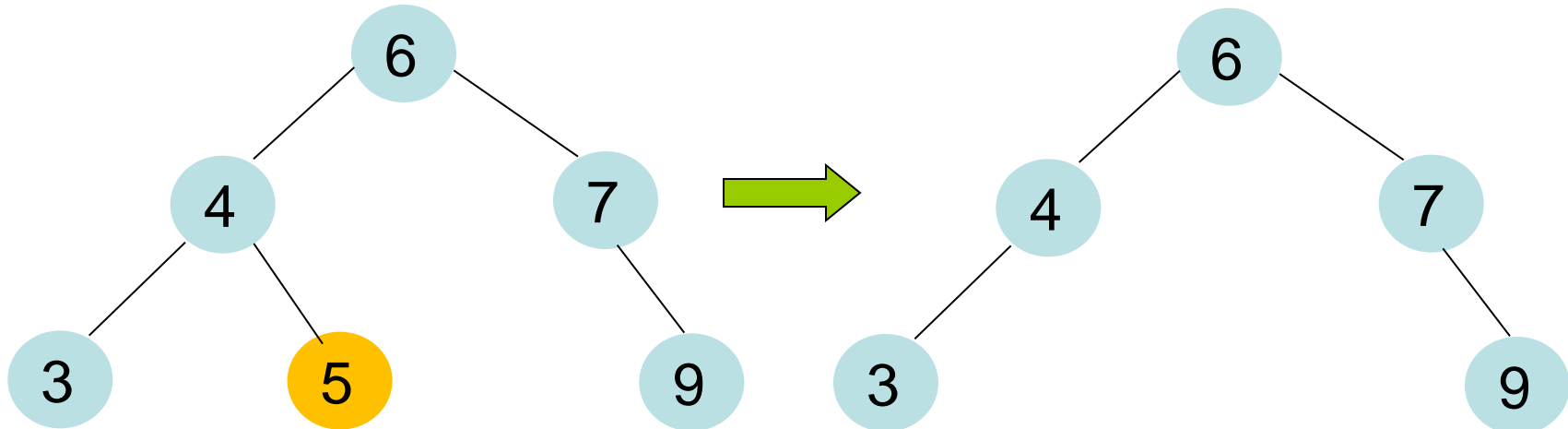
- 3 unterschiedliche Fälle
- (a) zu löschendes Element z hat keine Kinder
- (b) zu löschendes Element z hat ein Kind
- (c) zu löschendes Element z hat zwei Kinder



Datenstrukturen

Fall (a)

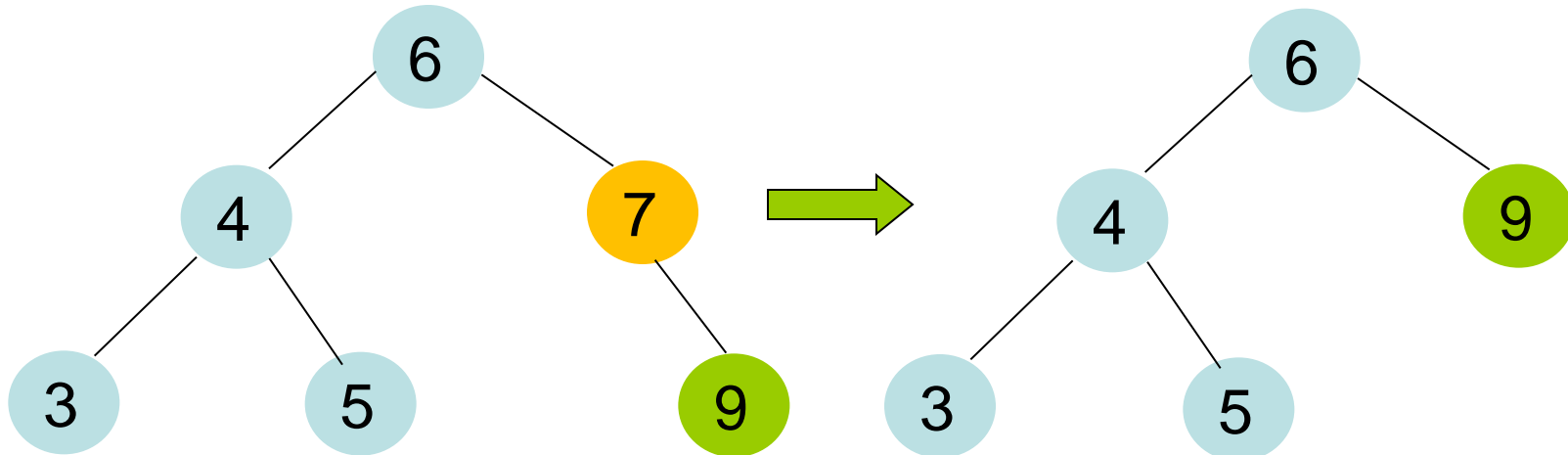
- zu löschendes Element z hat keine Kinder
- Entferne Element



Datenstrukturen

Fall (b)

- Zu löschendes Element z hat 1 Kind

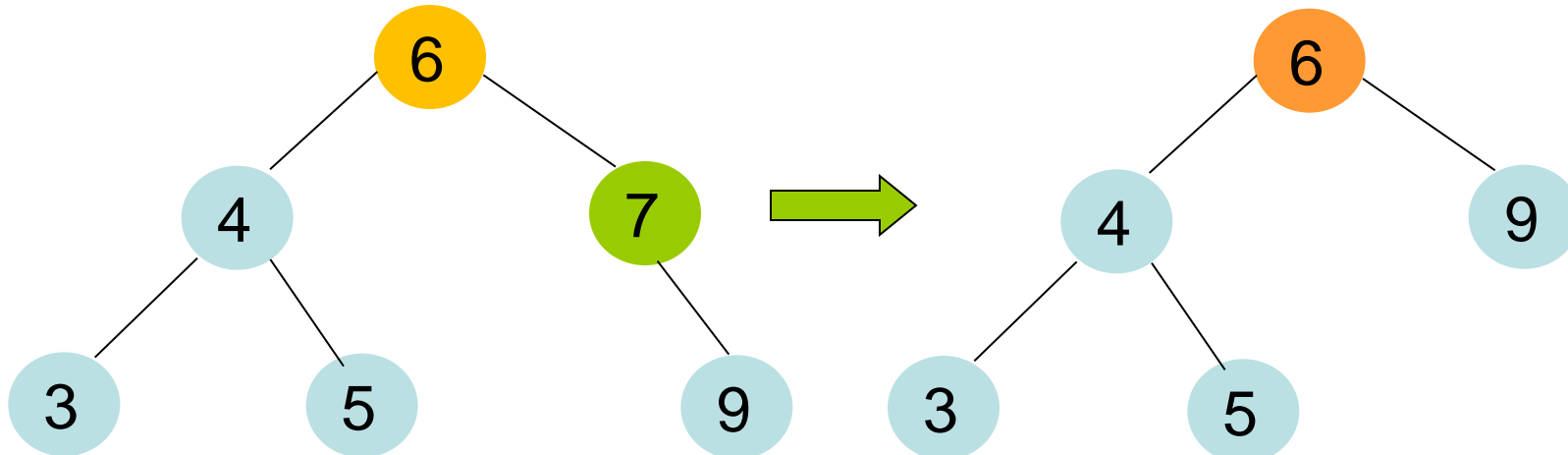


Datenstrukturen

Fall (c)

- Zu löschendes Element z hat 2 Kinder
- Schritt 1: Bestimme Nachfolger von z

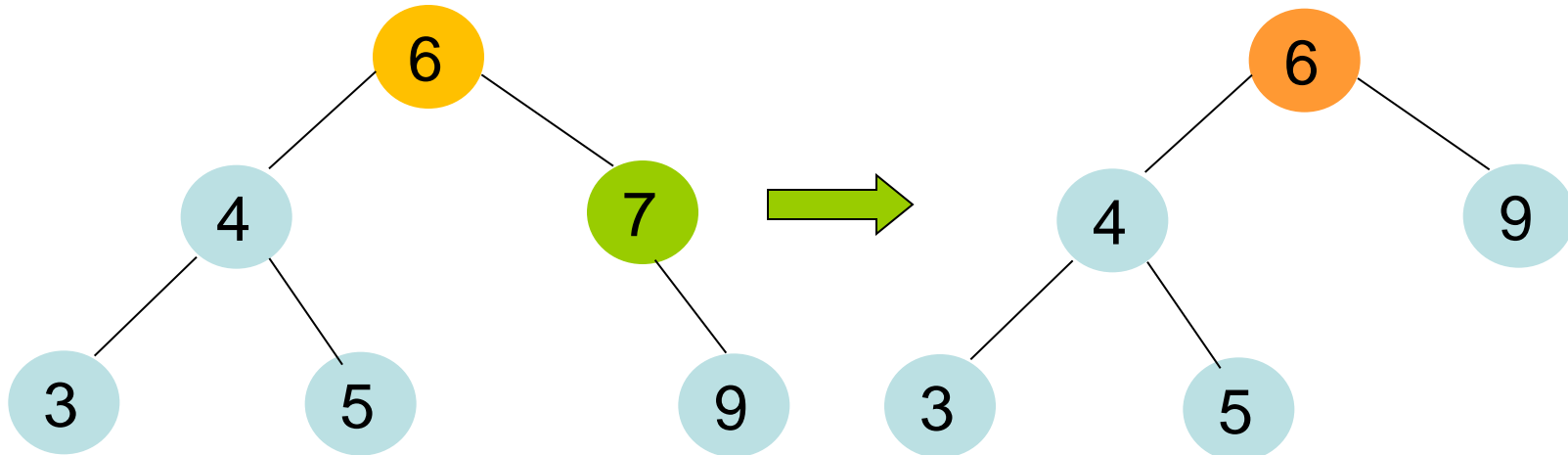
Nachfolger hat nur ein Kind



Datenstrukturen

Fall (c)

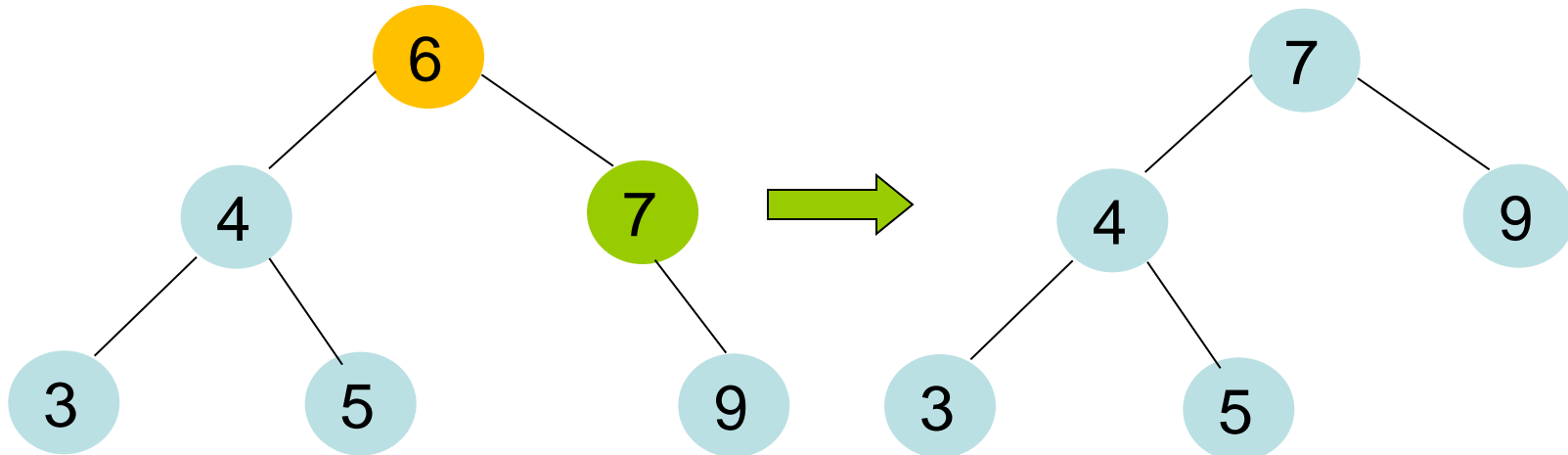
- Zu löschendes Element z hat 2 Kinder
- Schritt 1: Bestimme Nachfolger von z
- Schritt 2: Entferne Nachfolger



Datenstrukturen

Fall (c)

- Zu löschendes Element z hat 2 Kinder
- Schritt 1: Bestimme Nachfolger von z
- Schritt 2: Entferne Nachfolger
- Schritt 3: Ersetze z durch Nachfolger

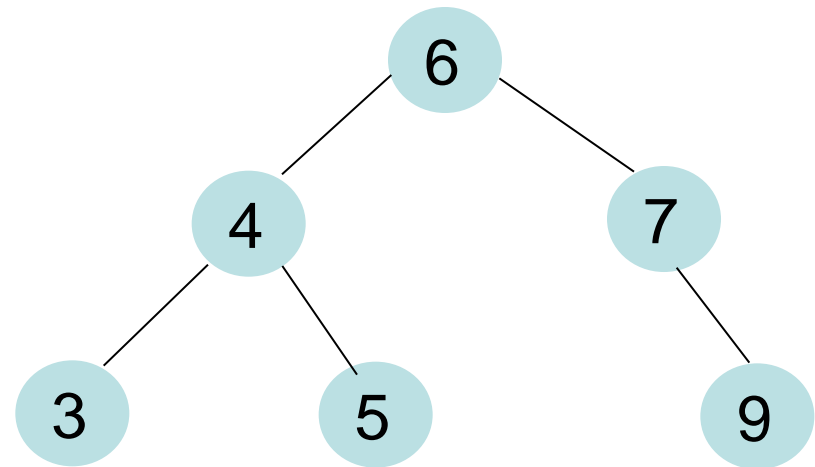


Datenstruk (Referenz auf z wird übergeben! (Rekursion))

Löschen(T,z)

1. **if** lc[z]=nil or rc[z]=nil **then** y ← z
2. **else** y ← NachfolgerSuche(z)
3. **if** lc[y] ≠ nil **then** x ← lc[y]
4. **else** x ← rc[y]
5. **if** x ≠ nil **then** p[x] ← p[y]
6. **if** p[y]=nil **then** root[T] ← x
7. **else if** y=lc[p[y]] **then** lc[p[y]] ← x
8. **else** rc[p[y]] ← x
9. key[z] ← key[y]

Löschen(6)



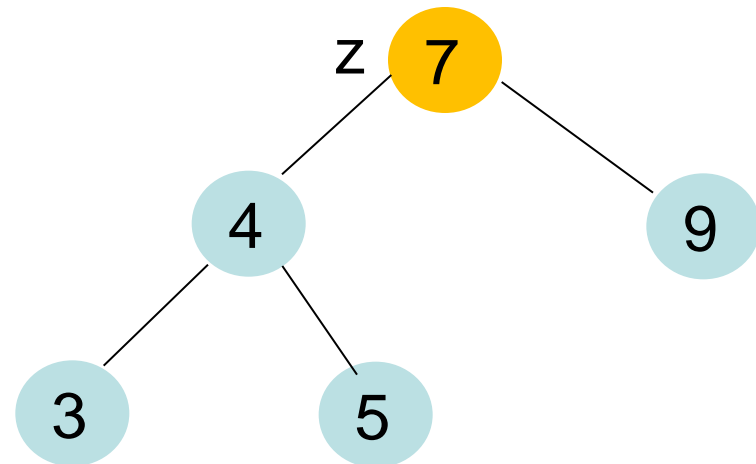
Datenstrukturen (siehe Vollversion)

Laufzeit $O(h)$

Löschen(T, z)

1. **if** $lc[z]=\text{nil}$ or $rc[z]=\text{nil}$ **then** $y \leftarrow z$
2. **else** $y \leftarrow \text{NachfolgerSuche}(z)$
3. **if** $lc[y] \neq \text{nil}$ **then** $x \leftarrow lc[y]$
4. **else** $x \leftarrow rc[y]$
5. **if** $x \neq \text{nil}$ **then** $p[x] \leftarrow p[y]$
6. **if** $p[y]=\text{nil}$ **then** $\text{root}[T] \leftarrow x$
7. **else if** $y=lc[p[y]]$ **then** $lc[p[y]] \leftarrow x$
8. **else** $rc[p[y]] \leftarrow x$
9. $\text{key}[z] \leftarrow \text{key}[y]$

Löschen(6)



Datenstrukturen

Binäre Suchbäume

- Ausgabe aller Elemente in $O(n)$
- Suche, Minimum, Maximum, Nachfolger in $O(h)$
- Einfügen, Löschen in $O(h)$

Frage

- Wie kann man eine „kleine“ Höhe unter Einfügen und Löschen garantieren?