



## Datenstrukturen, Algorithmen und Programmierung 2 (DAP2)

# Datenstrukturen

## *Was ist eine Datenstruktur?*

- Eine Datenstruktur ist eine Anordnung von Daten im Speicher eines Rechners, die effizienten Zugriff auf die Daten ermöglicht
- Datenstrukturen für viele unterschiedliche Anfragen vorstellbar

# Datenstrukturen

## *Ein grundlegendes Datenbank-Problem*

Speicherung von Datensätzen

### *Beispiel*

Kundendaten (Name, Adresse, Wohnort, Kundennummer, offene Rechnungen, offene Bestellungen,...)

### *Anforderungen*

- Schneller Zugriff
- Einfügen neuer Datensätze
- Löschen bestehender Datensätze

## Datenstrukturen

### *Zugriff auf Daten*

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

### *Beispiel*

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

## Datenstrukturen

### *Zugriff auf Daten*

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

### *Beispiel*

- Kundendaten (Name, Adresse, Kundennummer)
- Schlüssel: Kundennummer
- Totale Ordnung:  $, \leq$

## Datenstrukturen

### *Problem*

Gegeben sind  $n$  Objekte  $O_1, \dots, O_n$  mit zugehörigen Schlüsseln  $s(O_i)$

### *Operationen*

- **Suche( $x$ )**; Ausgabe  $O$  mit Schlüssel  $s(O) = x$ ;  
**nil**, falls kein Objekt mit Schlüssel  $x$  in Datenbank
- **Einfügen( $O$ )**; Einfügen von Objekt  $O$  in Datenbank
- **Löschen( $O$ )**; Löschen von Objekt  $O$  mit aus der Datenbank

## Datenstrukturen

### *Vereinfachung*

- Schlüssel sind natürliche Zahlen
- Eingabe nur aus Schlüsseln

### *Analyse von Datenstrukturen*

- Platzbedarf in  $\Theta$ - bzw.  $O$ -Notation
- Laufzeit der Operationen in  $\Theta$ - bzw.  $O$ -Notation

## Aufgabe

### *Datenstrukturen*

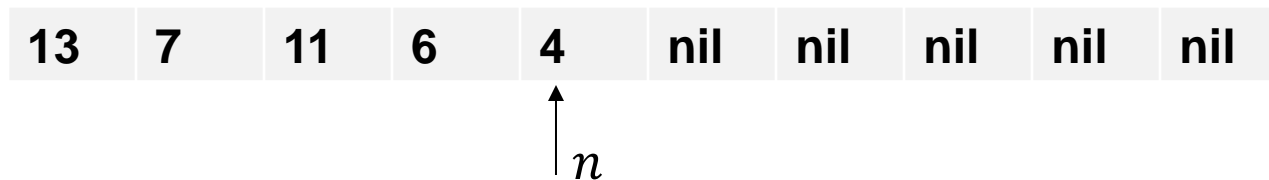
- Welche einfachen Datenstrukturen kennen Sie, mit denen man das Wörterbuchproblem lösen kann?
- Was sind die Laufzeiten für Suchen, Einfügen und Löschen?  
( $n$  bezeichne die Anzahl der Objekte in der Datenstruktur)



## Datenstrukturen

### *Einfaches Feld*

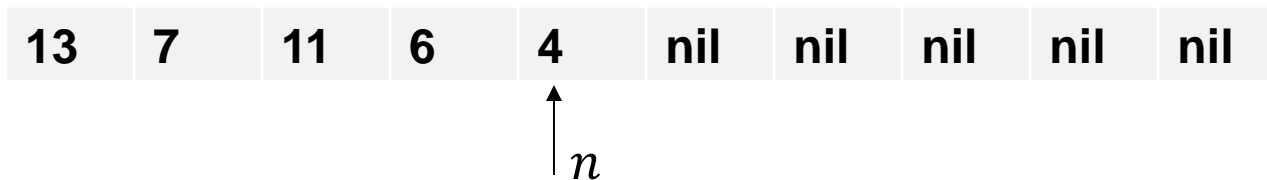
- Feld  $A[1..max]$
- Integer  $n$ ,  $1 \leq n \leq max$
- $n$  bezeichnet Anzahl Elemente in Datenstruktur



## Datenstrukturen

### Einfügen(s)

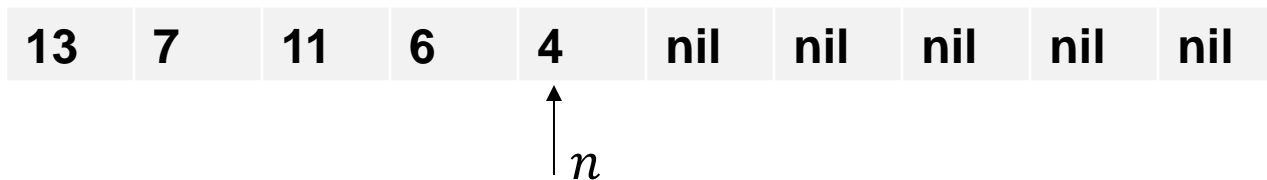
1. **if**  $n = \text{max}$  **then** Ausgabe „Fehler: Kein Platz in Datenstruktur“
2. **else**
3.      $n \leftarrow n + 1$
4.      $A[n] \leftarrow s$



## Datenstrukturen

Einfügen(s)

1. **if**  $n = \text{max}$  **then** Ausgabe „Fehler: Kein Platz in Datenstruktur“
2. **else**
3.      $n \leftarrow n + 1$
4.      $A[n] \leftarrow s$

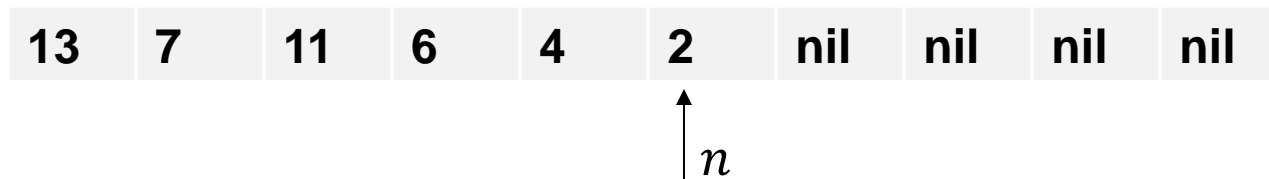


Einfügen(2)

## Datenstrukturen

Einfügen(s)

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

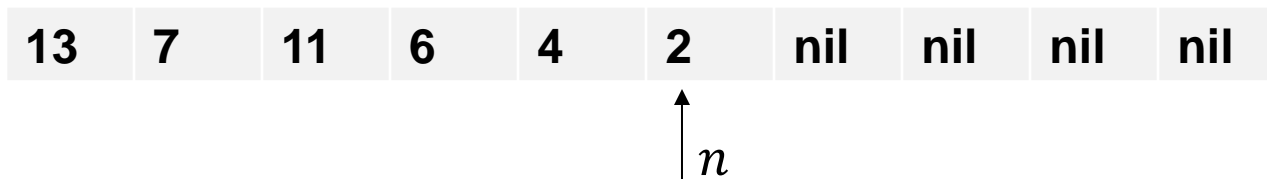


Einfügen(2)

## Datenstrukturen

Suche( $x$ )

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

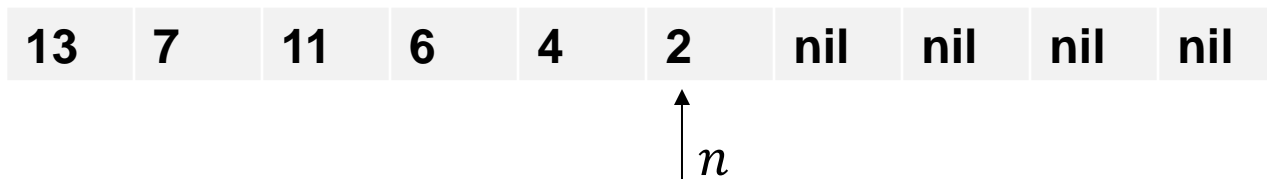


## Datenstrukturen

Löschen( $i$ )

1.  $A[i] \leftarrow A[n]$
2.  $A[n] \leftarrow \mathbf{nil}$
3.  $n \leftarrow n - 1$

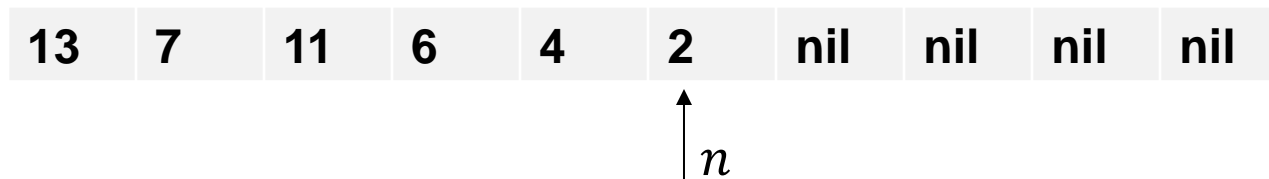
Annahme:  
Wir bekommen  
Index  $i$  des zu  
löschenden Objekts



## Datenstrukturen

Löschen( $i$ )

1.  $A[i] \leftarrow A[n]$
2.  $A[n] \leftarrow \mathbf{nil}$
3.  $n \leftarrow n - 1$

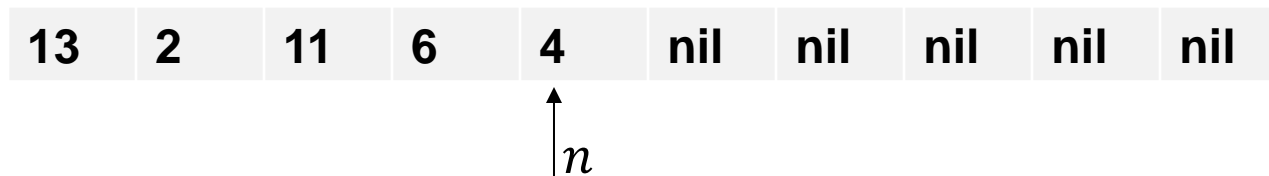


Löschen(2)

## Datenstrukturen

Löschen( $i$ )

1.  $A[i] \leftarrow A[n]$
2.  $A[n] \leftarrow \mathbf{nil}$
3.  $n \leftarrow n - 1$



Löschen(2)



## Datenstrukturen

### *Datenstruktur Feld*

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

### *Vorteile*

- Schnelles Einfügen und Löschen

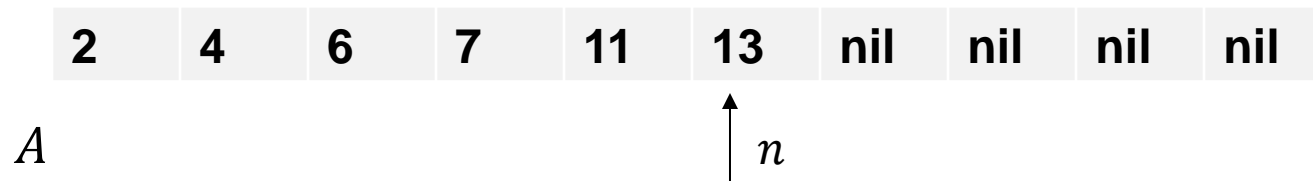
### *Nachteile*

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

## Datenstrukturen

### *Datenstruktur „sortiertes Feld“*

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



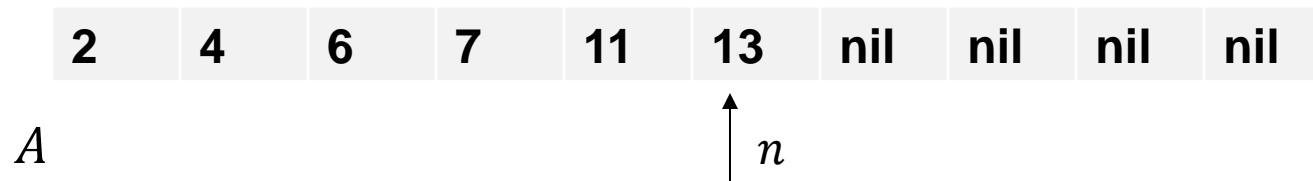
## Datenstrukturen

### *Datenstruktur „sortiertes Feld“*

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

Welchen Vorteil bietet ein sortiertes Feld?

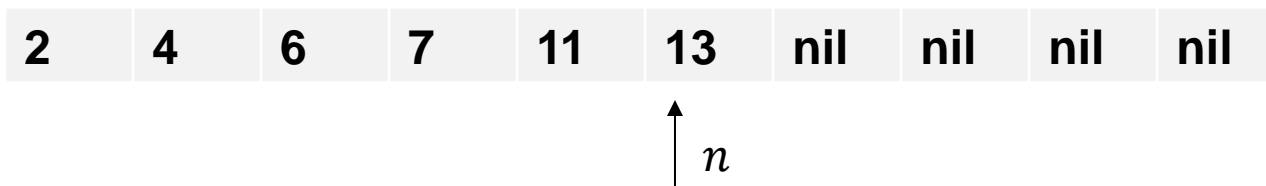
- A) Schnelleres Einfügen
- B) Weniger Speicherverbrauch
- C) Schnelleres Löschen
- D) Schnelleres Suchen



## 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.  $A[i] \leftarrow s$

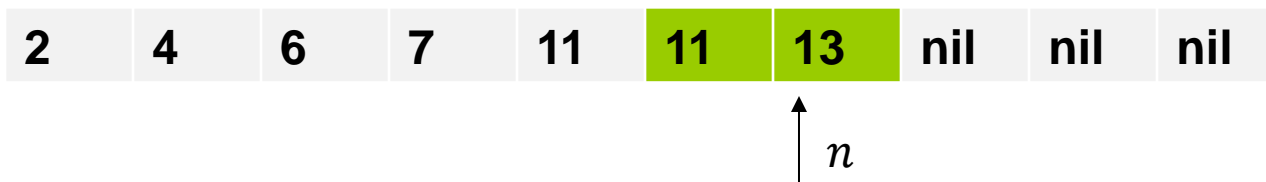


Einfügen(10)

## 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)

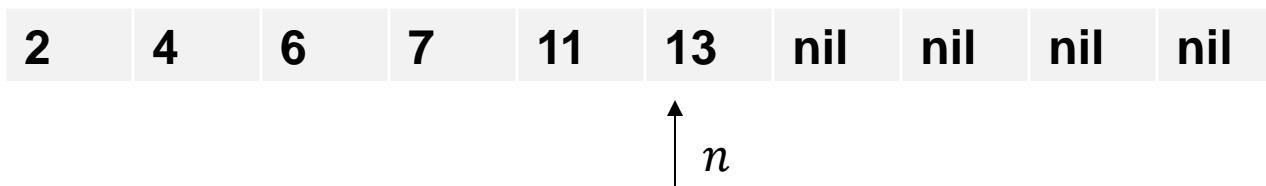


## Datenstrukturen

Löschen( $i$ )

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

Parameter ist der  
Index  $i$  des zu  
löschenden Objekts

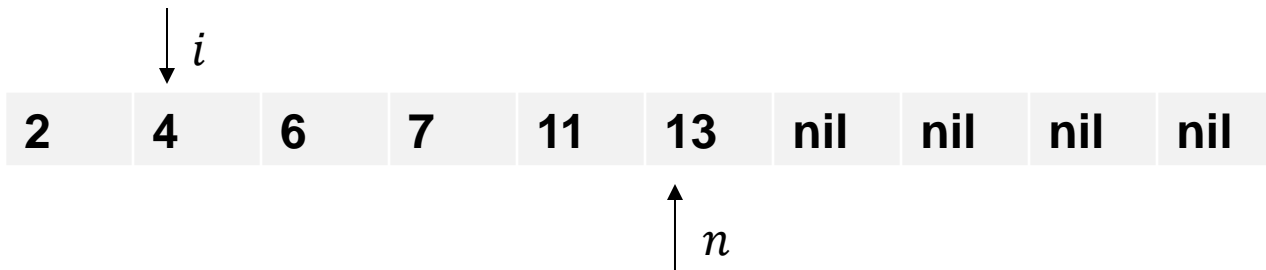


## Datenstrukturen

Löschen( $i$ )

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

Parameter ist der  
Index  $i$  des zu  
löschenden Objekts

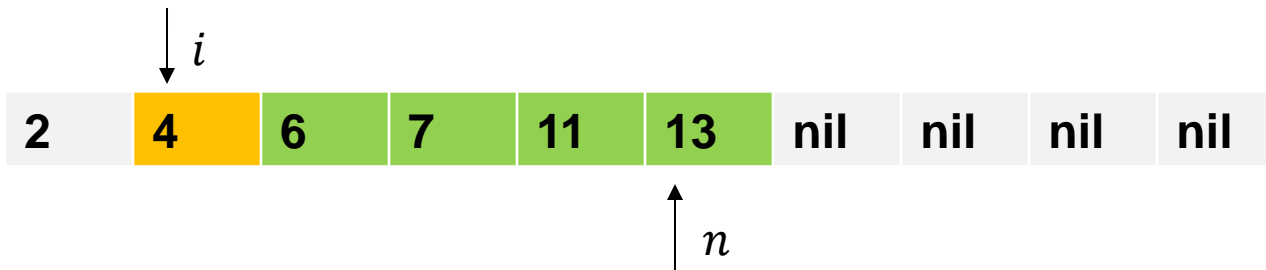


Löschen(2)

## Datenstrukturen

Löschen( $i$ )

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

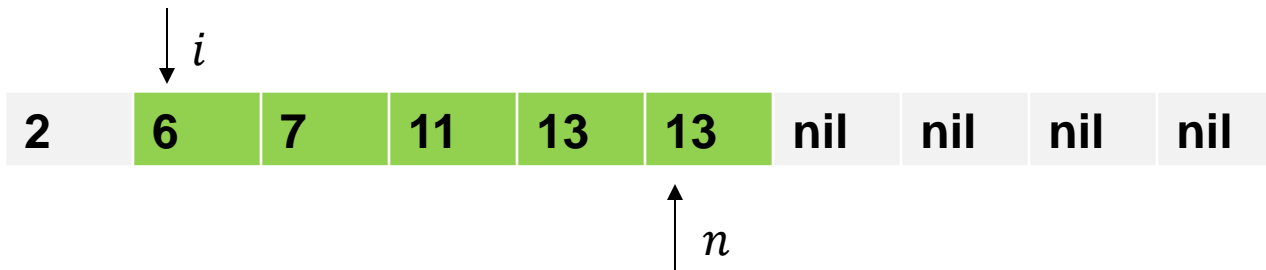


Löschen(2)

## Datenstrukturen

Löschen( $i$ )

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

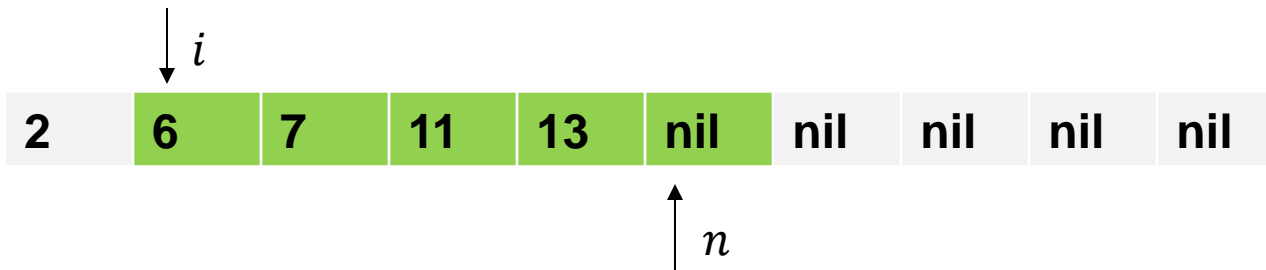


Löschen(2)

## 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.      $n \leftarrow n - 1$

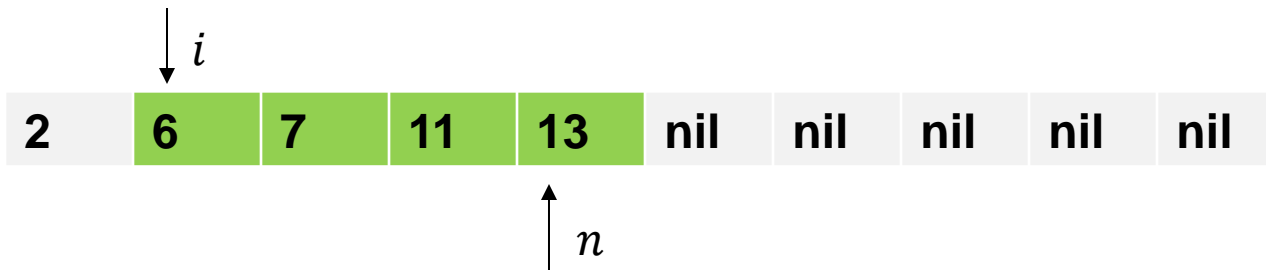


Löschen(2)

## Datenstrukturen

Löschen( $i$ )

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

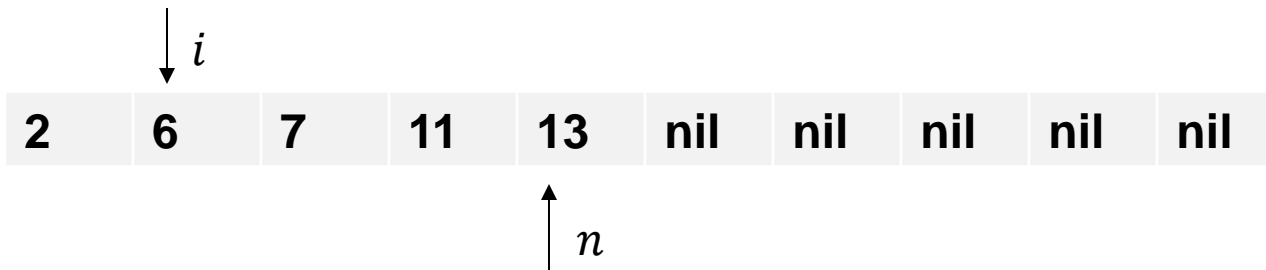


Löschen(2)

## Datenstrukturen

Suchen( $x$ )

- Binäre Suche
- Laufzeit  $\mathbf{O}(\log n)$



Löschen(2)

## Datenstrukturen

### *Datenstruktur sortiertes Feld*

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

### *Vorteile*

- Schnelles Suchen

### *Nachteile*

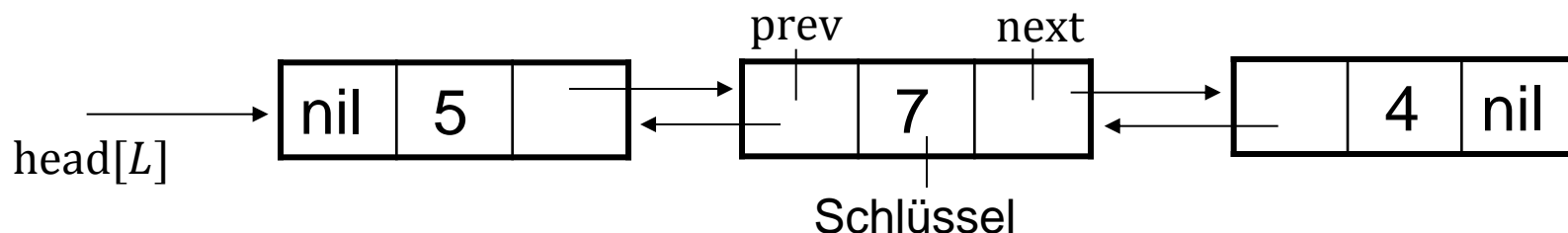
- Speicherbedarf abhängig von max (nicht vorhersagbar)
- Hohe Laufzeit für Einfügen/Löschen



## Datenstrukturen

### *Doppelt verkettete Listen*

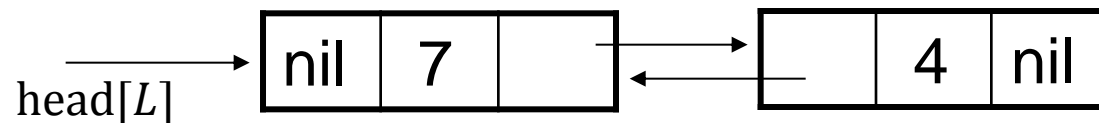
- Listenelement  $x$  ist Objekt bestehend aus **Schlüssel** und zwei Zeigern **prev** und **next**
- next verweist auf Nachfolger von  $x$
- prev verweist auf Vorgänger von  $x$
- prev/next sind **nil**, wenn Vorgänger/Nachfolger nicht existiert
- head[L] zeigt auf das erste Element



## Datenstrukturen

Einfügen( $L, x$ )

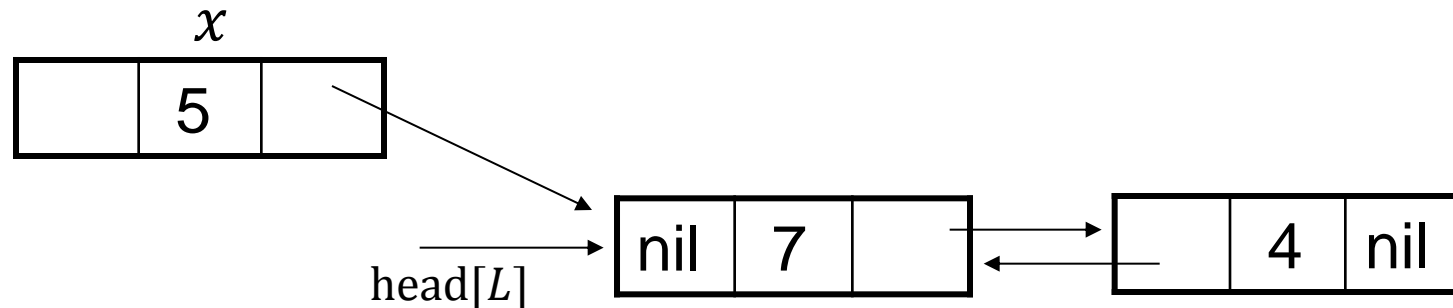
1.  $\text{next}[x] \leftarrow \text{head}[L]$
2. **if**  $\text{head}[L] \neq \text{nil}$  **then**  $\text{prev}[\text{head}[L]] \leftarrow x$
3.  $\text{head}[L] \leftarrow x$
4.  $\text{prev}[x] \leftarrow \text{nil}$



## Datenstrukturen

Einfügen( $L, x$ )

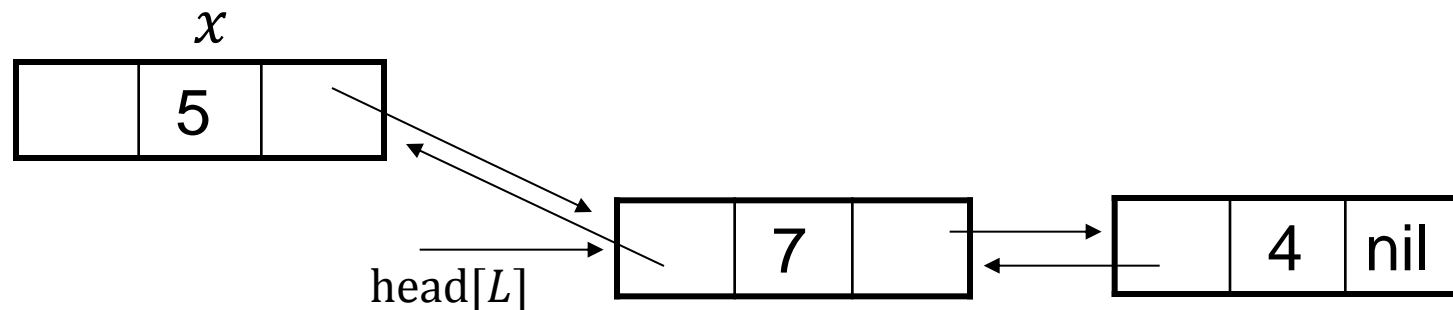
1.  $\text{next}[x] \leftarrow \text{head}[L]$
2. **if**  $\text{head}[L] \neq \text{nil}$  **then**  $\text{prev}[\text{head}[L]] \leftarrow x$
3.  $\text{head}[L] \leftarrow x$
4.  $\text{prev}[x] \leftarrow \text{nil}$



## Datenstrukturen

Einfügen( $L, x$ )

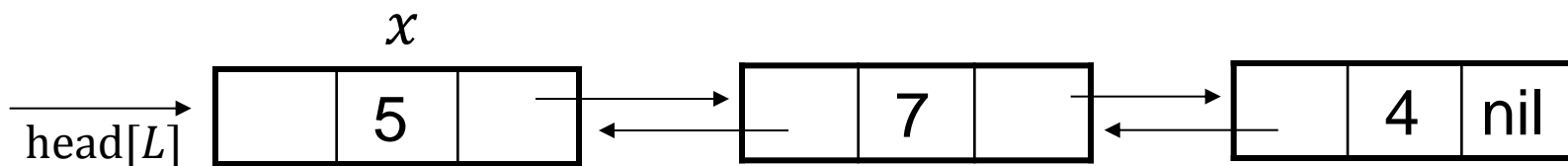
1.  $\text{next}[x] \leftarrow \text{head}[L]$
2. **if**  $\text{head}[L] \neq \text{nil}$  **then**  $\text{prev}[\text{head}[L]] \leftarrow x$
3.  $\text{head}[L] \leftarrow x$
4.  $\text{prev}[x] \leftarrow \text{nil}$



## Datenstrukturen

Einfügen( $L, x$ )

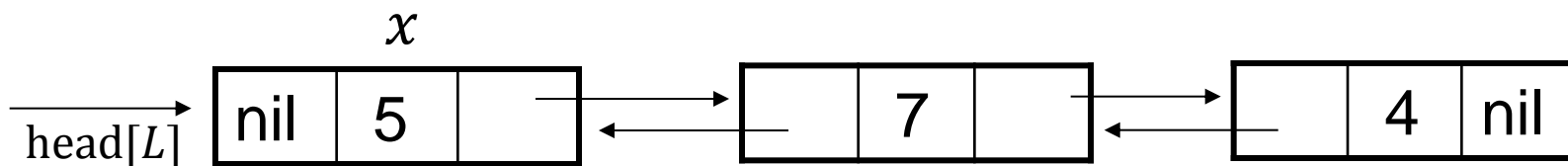
1.  $\text{next}[x] \leftarrow \text{head}[L]$
2. **if**  $\text{head}[L] \neq \text{nil}$  **then**  $\text{prev}[\text{head}[L]] \leftarrow x$
3.  $\text{head}[L] \leftarrow x$
4.  $\text{prev}[x] \leftarrow \text{nil}$



## Datenstrukturen

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.  $\text{prev}[x] \leftarrow \text{nil}$



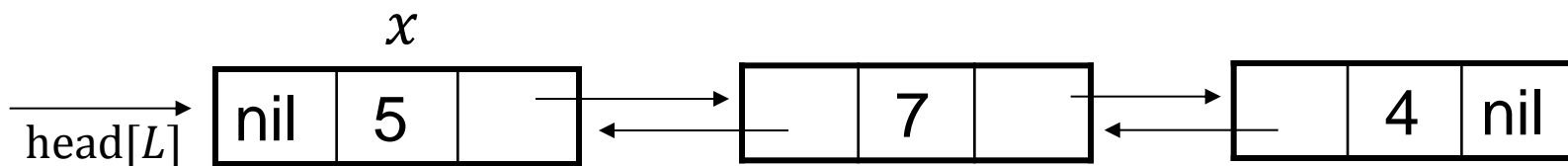
## Datenstrukturen

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*

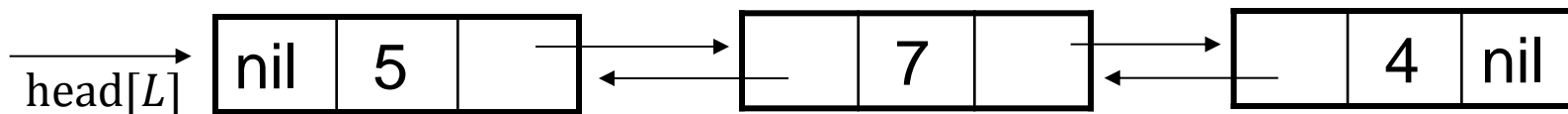
- $\mathbf{O}(1)$



## Datenstrukturen

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**  $x$

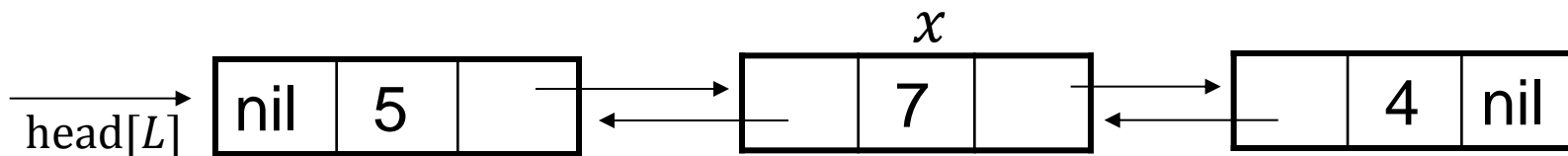




## Datenstrukturen

Löschen( $L, x$ )

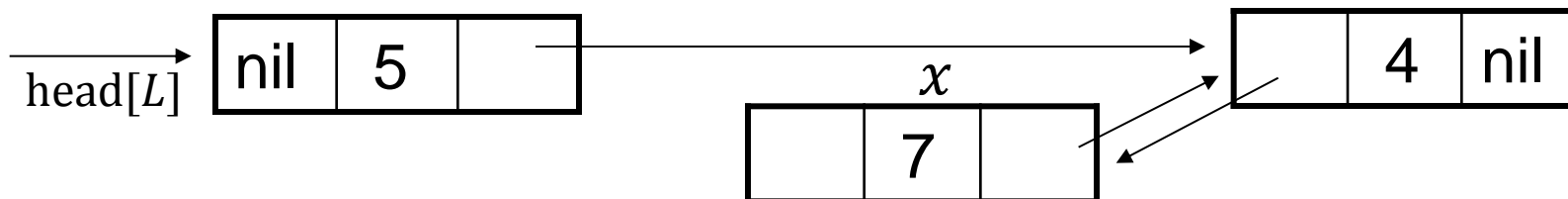
1. **if**  $\text{prev}[x] \neq \text{nil}$  **then**  $\text{next}[\text{prev}[x]] \leftarrow \text{next}[x]$
2. **else**  $\text{head}[L] \leftarrow \text{next}[x]$
3. **if**  $\text{next}[x] \neq \text{nil}$  **then**  $\text{prev}[\text{next}[x]] \leftarrow \text{prev}[x]$
4. **delete**  $x$



## Datenstrukturen

Löschen( $L, x$ )

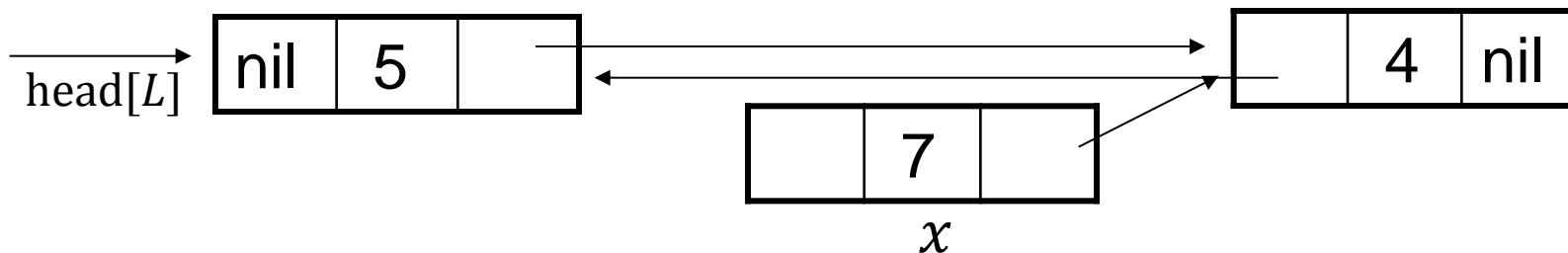
1. **if**  $\text{prev}[x] \neq \text{nil}$  **then**  $\text{next}[\text{prev}[x]] \leftarrow \text{next}[x]$
2. **else**  $\text{head}[L] \leftarrow \text{next}[x]$
3. **if**  $\text{next}[x] \neq \text{nil}$  **then**  $\text{prev}[\text{next}[x]] \leftarrow \text{prev}[x]$
4. **delete**  $x$



## Datenstrukturen

Löschen( $L, x$ )

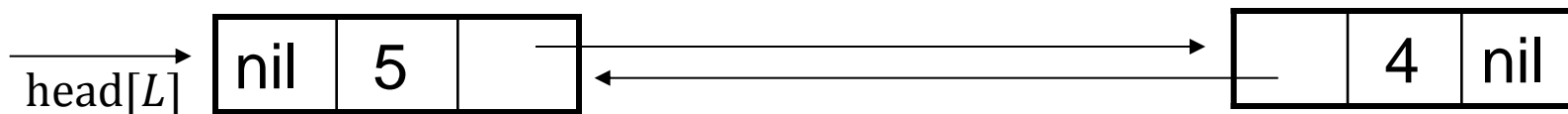
1. **if**  $\text{prev}[x] \neq \text{nil}$  **then**  $\text{next}[\text{prev}[x]] \leftarrow \text{next}[x]$
2. **else**  $\text{head}[L] \leftarrow \text{next}[x]$
3. **if**  $\text{next}[x] \neq \text{nil}$  **then**  $\text{prev}[\text{next}[x]] \leftarrow \text{prev}[x]$
4. **delete**  $x$



## Datenstrukturen

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**  $x$



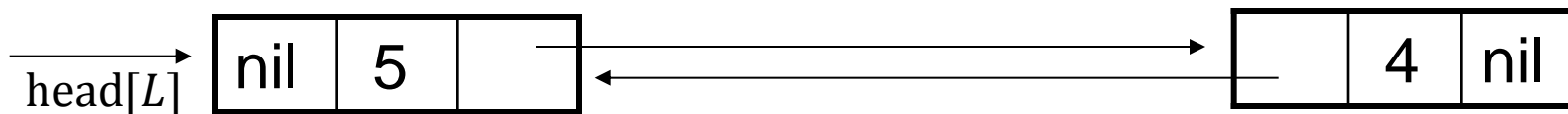
## Datenstrukturen

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**  $x$

*Laufzeit*

- $O(1)$



## Datenstrukturen

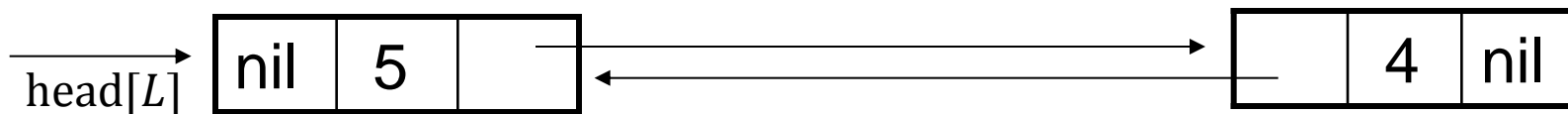
Suche( $L, k$ )

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**  $x$

## Datenstrukturen

Suche( $L, k$ )

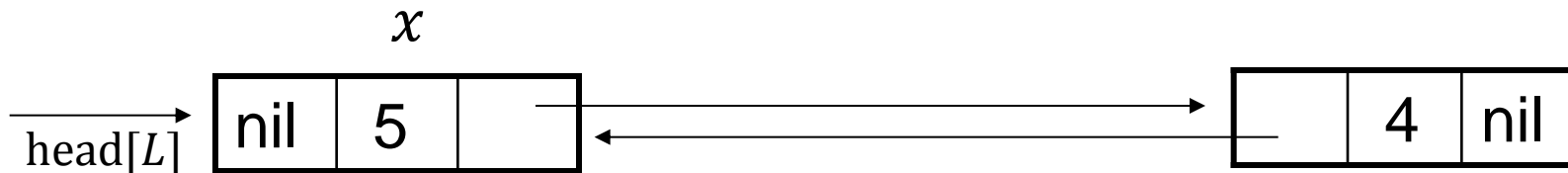
1.  $x \leftarrow \text{head}[L]$
2. **while**  $x \neq \text{nil}$  and  $\text{key}[x] \neq k$  **do**
3.      $x \leftarrow \text{next}[x]$
4. **return**  $x$



## 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**  $x$



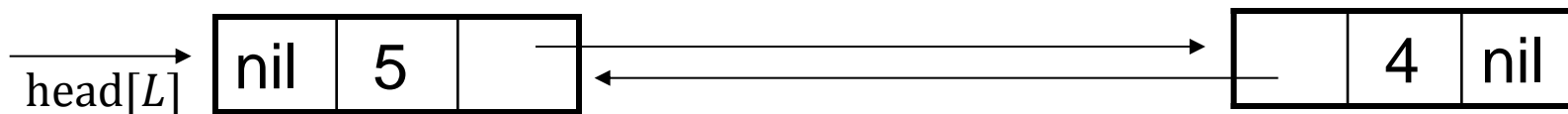
Suche( $L, 4$ )



## 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**  $x$

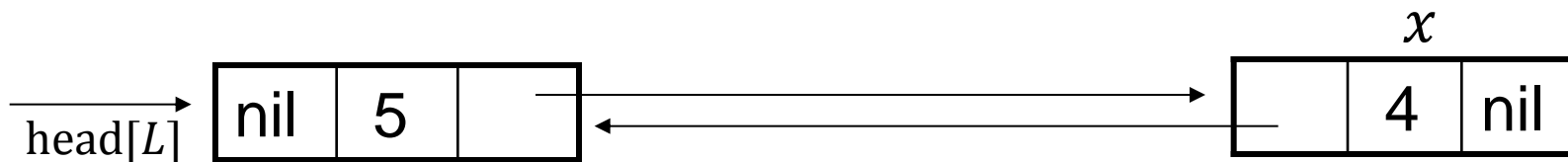


Suche( $L, 4$ )

## 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**  $x$

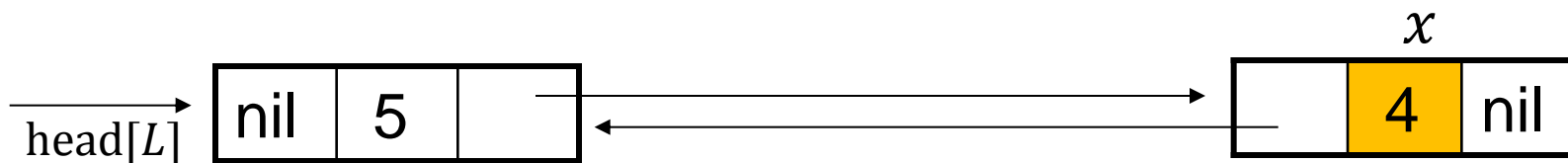


Suche( $L, 4$ )

## 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**  $x$



Suche( $L, 4$ )

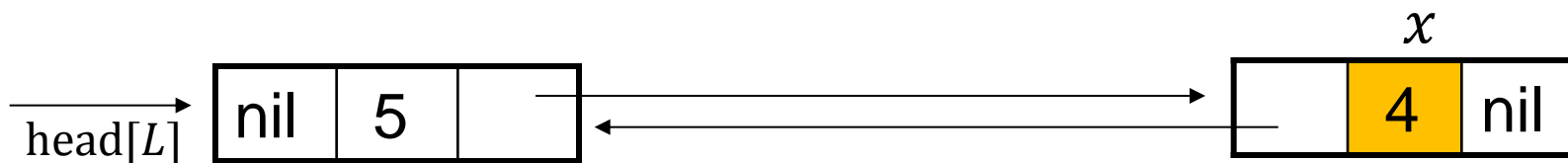
## Datenstrukturen

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*

- $\mathbf{O}(n)$



Suche( $L, 4$ )

## Datenstrukturen

### *Datenstruktur Liste*

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

### *Vorteile*

- Schnelles Einfügen/Löschen
- $\Theta(n)$  Speicherbedarf

### *Nachteile*

- Hohe Laufzeit für Suche

# Datenstrukturen

## *Ein grundlegendes Datenbank-Problem*

Speicherung von Datensätzen

### *Beispiel*

Kundendaten (Name, Adresse, Wohnort, Kundennummer, offene Rechnungen, offene Bestellungen,...)

### *Anforderungen*

- Schneller Zugriff
- Einfügen neuer Datensätze
- Löschen bestehender Datensätze

## Datenstrukturen

### *Zugriff auf Daten*

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

### *Beispiel*

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

## Datenstrukturen

### *Problem*

Gegeben sind  $n$  Objekte  $O_1, \dots, O_n$  mit zugehörigen Schlüsseln  $s(O_i)$

### *Operationen*

- **Suche( $x$ )**; Ausgabe  $O$  mit Schlüssel  $s(O) = x$ ;  
**nil**, falls kein Objekt mit Schlüssel  $x$  in Datenbank
- **Einfügen( $O$ )**; Einfügen von Objekt  $O$  in Datenbank
- **Löschen( $O$ )**; Löschen von Objekt  $O$  mit aus der Datenbank



# 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

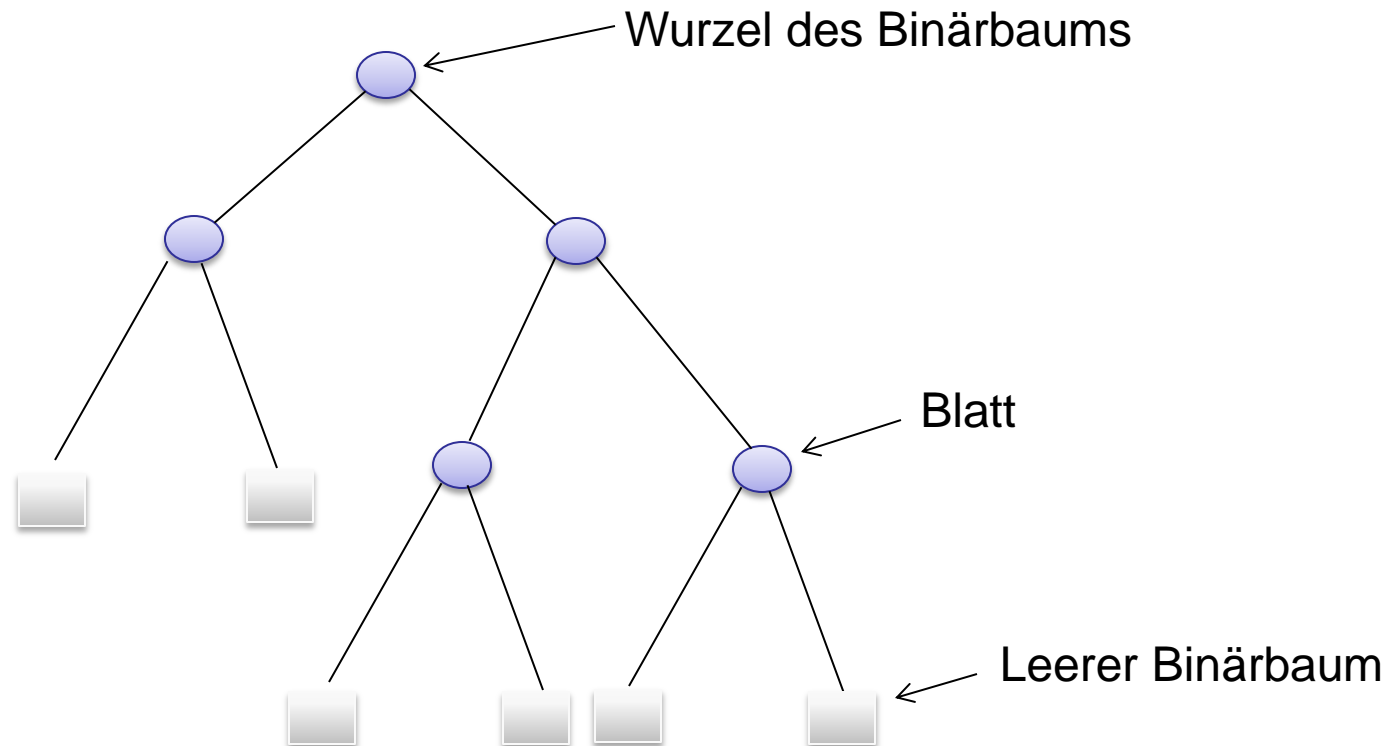
## Datenstrukturen

### *Definition (Binärbaum)*

- Ein Binärbaum  $T$  ist eine Struktur, die auf einer endlichen Menge definiert ist. Diese Menge nennt man auch die Knotenmenge des Binärbaums.
- Die leere Menge ist ein Binärbaum. Dieser wird auch als leerer Baum bezeichnet.
- Ein Binärbaum ist ein Tripel  $(v, T_1, T_2)$ , wobei  $T_1$  und  $T_2$  Binärbäume mit disjunkten Knotenmengen  $V_1$  und  $V_2$  sind und  $v \notin V_1 \cup V_2$  Wurzelknoten heißt. Die Knotenmenge des Baums ist dann  $\{v\} \cup V_1 \cup V_2$ .  
 $T_1$  heißt linker Unterbaum von  $v$  und  $T_2$  heißt rechter Unterbaum von  $v$ .

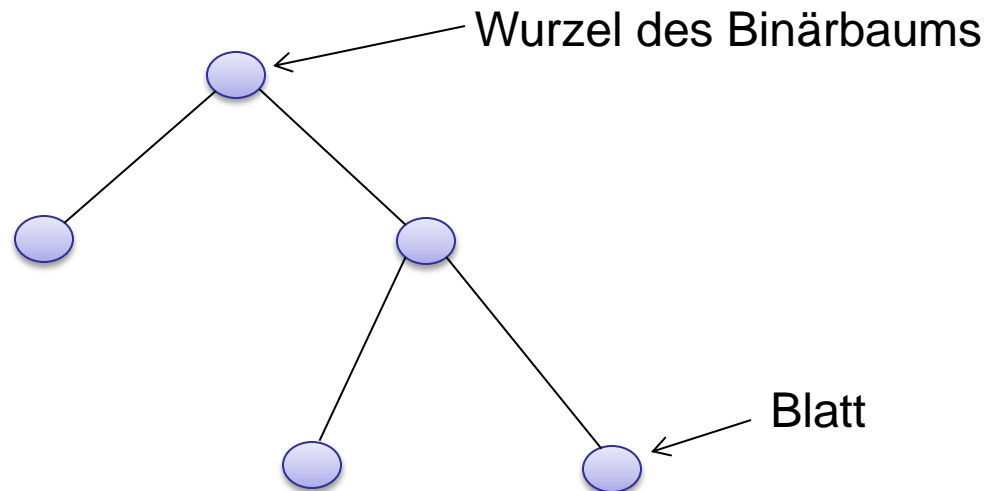
## Datenstrukturen

### *Darstellung von Binärbäumen*



## Datenstrukturen

### *Darstellung von Binärbäumen*



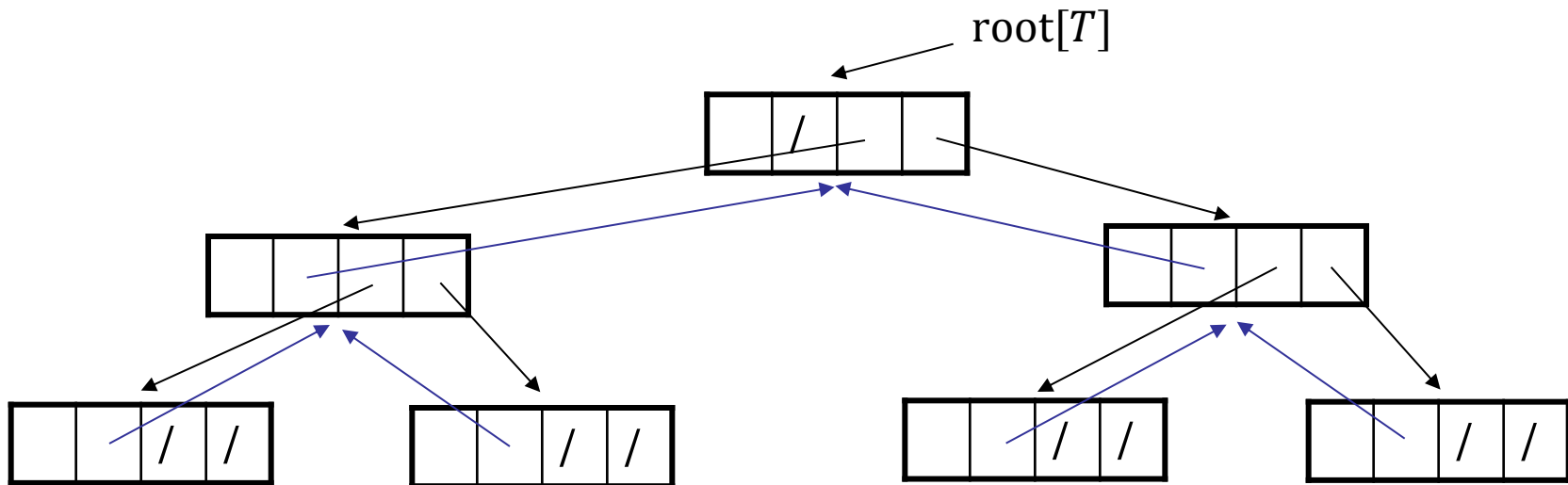
- Häufig lässt man die leeren Bäume in der Darstellung eines Binärbaums weg

## Datenstrukturen

### *Binärbäume(Darstellung im Rechner)*

- 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  $root[T]$

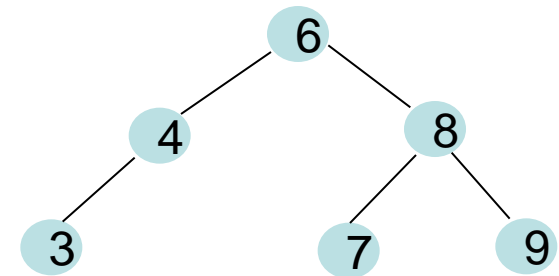
key	$p[v]$	$lc[v]$	$rc[v]$
-----	--------	---------	---------



## Datenstrukturen

### *Binäre Suchbäume*

- Verwende Binärbaum
- Speichere Schlüssel „geordnet“



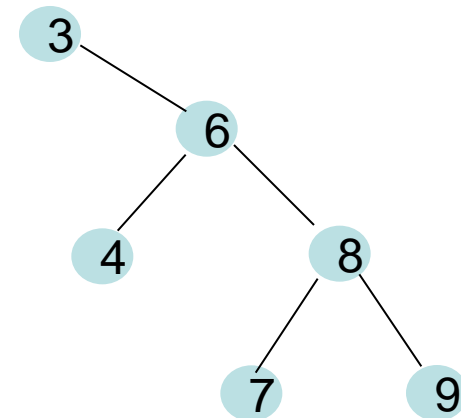
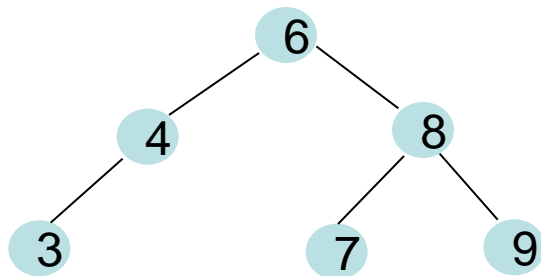
### *Binäre Suchbaumeigenschaft*

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

## Datenstrukturen

### *Unterschiedliche Suchbäume*

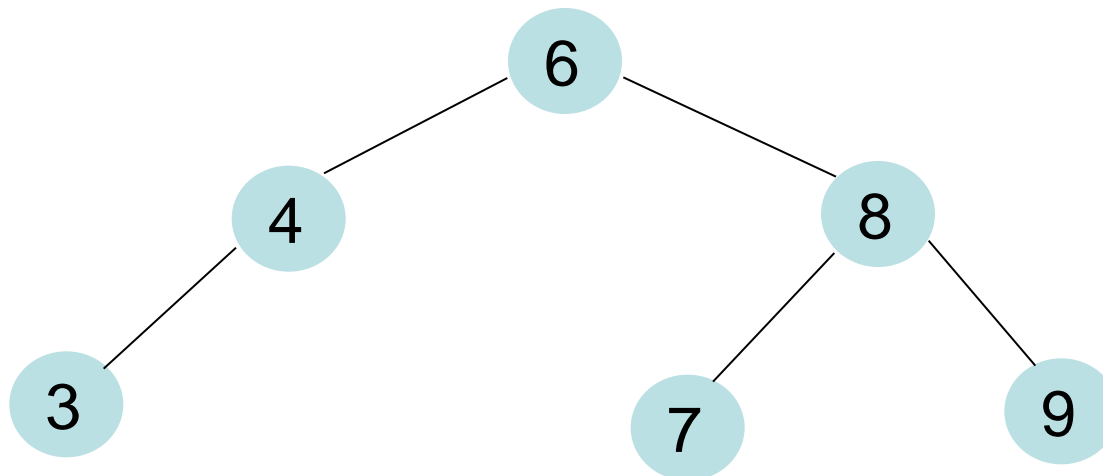
- Schlüsselmenge 3,4,6,7,8,9
- Wir erlauben kein mehrfaches Vorkommen desselben Schlüssels



## Datenstrukturen

### *Ausgabe aller Schlüssel*

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



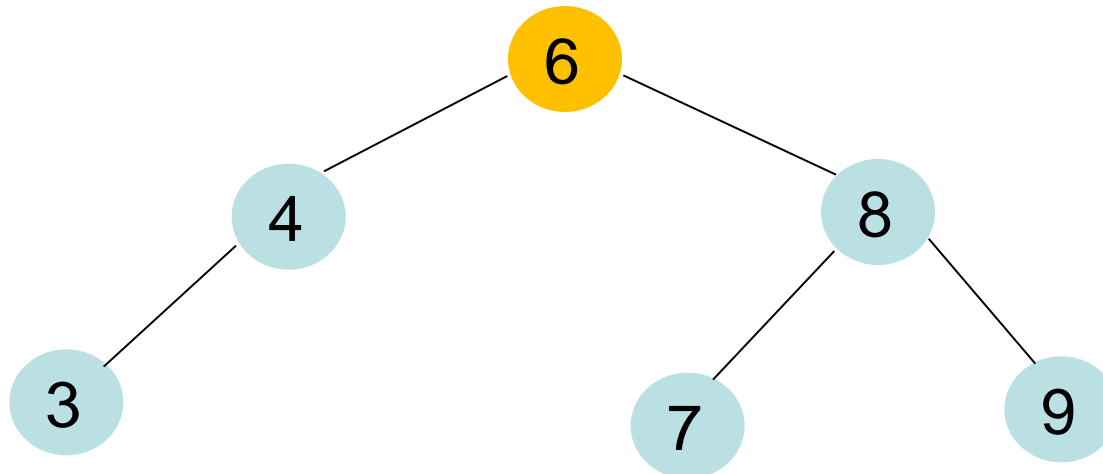


## Datenstrukturen

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[ $x$ ])

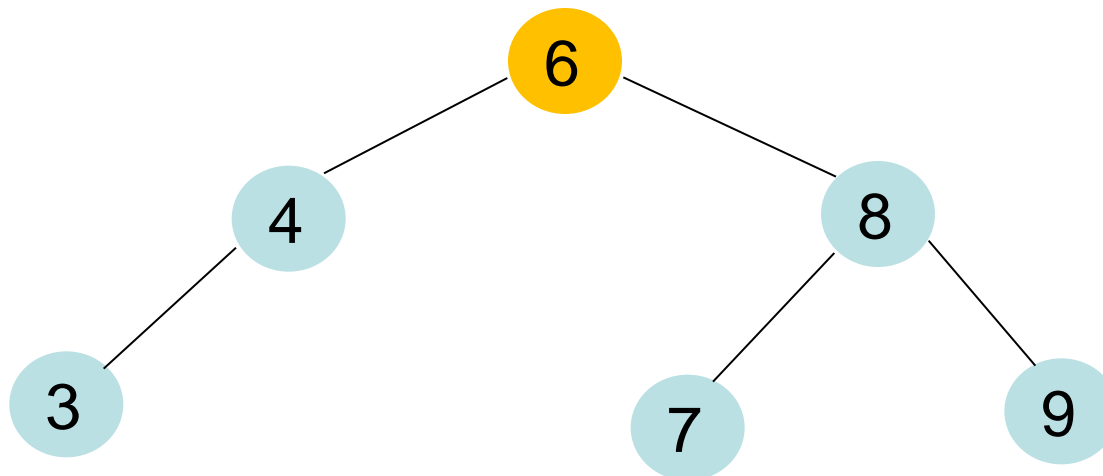
Aufruf über  
Inorder-Tree-Walk(root[ $T$ ])



## Datenstrukturen

Inorder-Tree-Walk( $x$ )

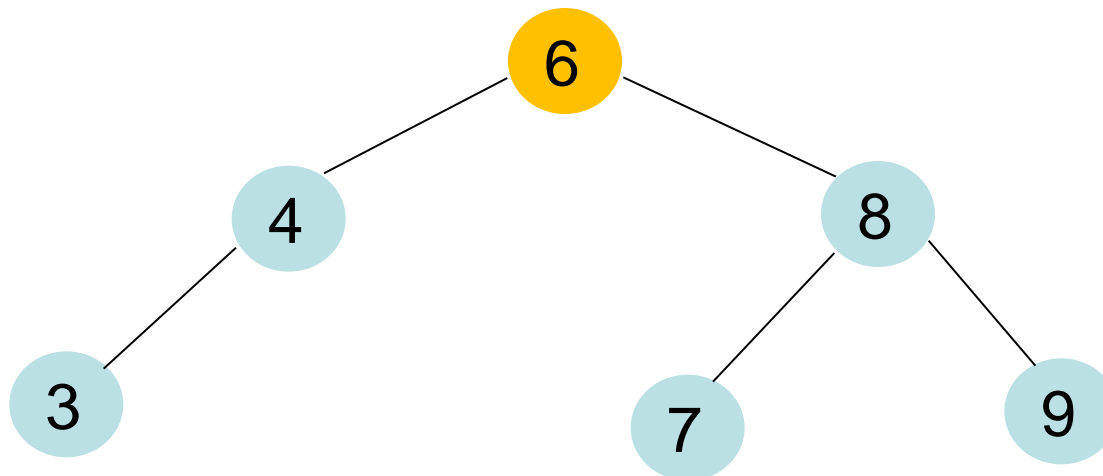
1. **if  $x \neq \text{nil}$  then**
2.     Inorder-Tree-Walk(lc[ $x$ ])
3.     Ausgabe key[ $x$ ]
4.     Inorder-Tree-Walk(rc[ $x$ ])



## Datenstrukturen

Inorder-Tree-Walk( $x$ )

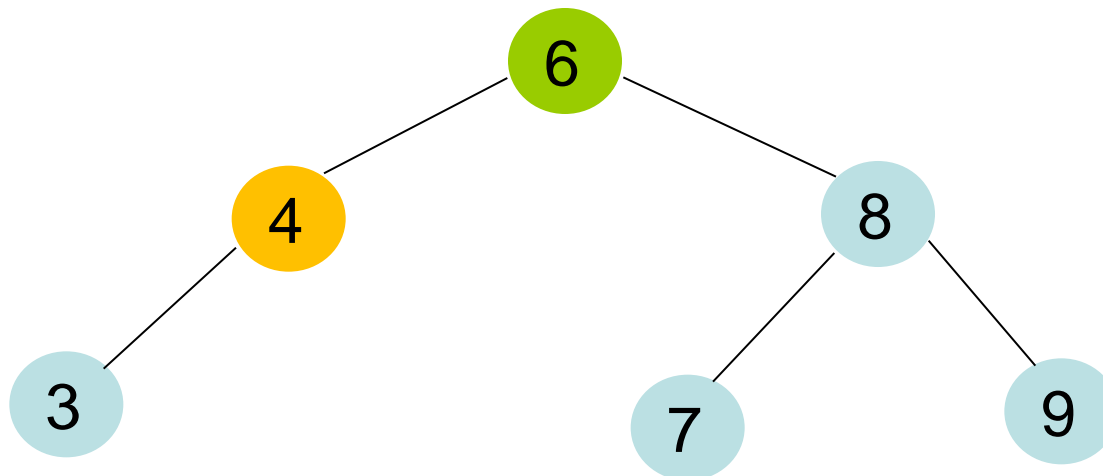
1. **if  $x \neq \text{nil}$  then**
2.   Inorder-Tree-Walk(lc[ $x$ ])
3.   Ausgabe key[ $x$ ]
4.   Inorder-Tree-Walk(rc[ $x$ ])



## Datenstrukturen

Inorder-Tree-Walk( $x$ )

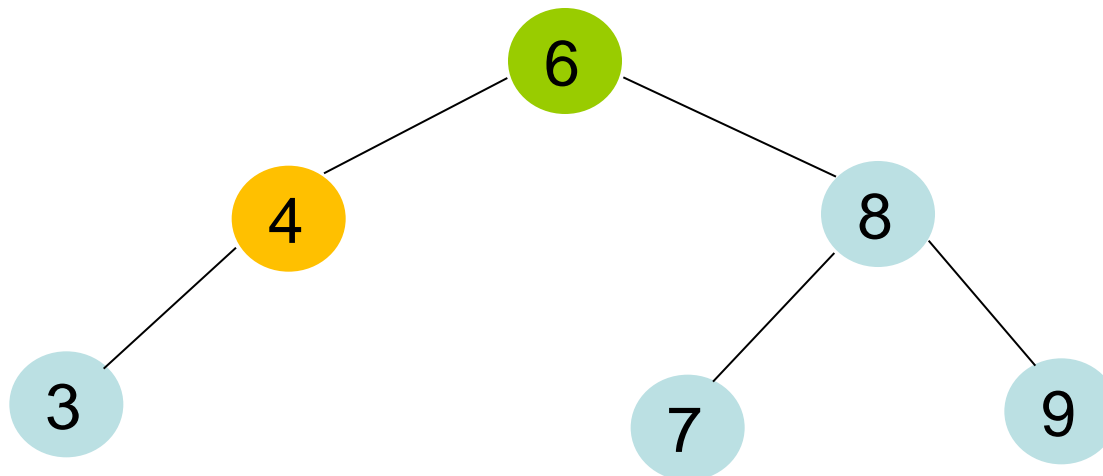
1. **if  $x \neq \text{nil}$  then**
2.     Inorder-Tree-Walk(lc[ $x$ ])
3.     Ausgabe key[ $x$ ]
4.     Inorder-Tree-Walk(rc[ $x$ ])



## Datenstrukturen

Inorder-Tree-Walk( $x$ )

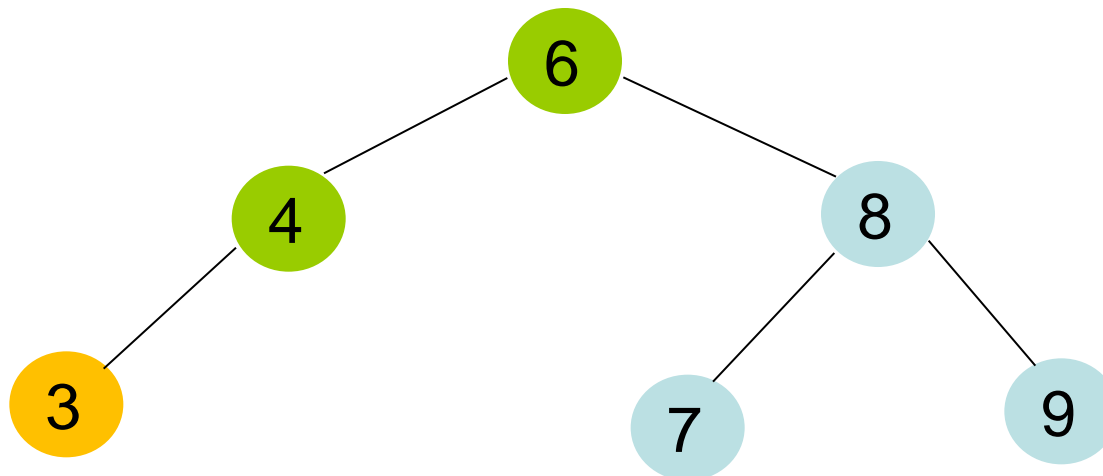
1. **if  $x \neq \text{nil}$  then**
2.   Inorder-Tree-Walk(lc[ $x$ ])
3.   Ausgabe key[ $x$ ]
4.   Inorder-Tree-Walk(rc[ $x$ ])



## Datenstrukturen

Inorder-Tree-Walk( $x$ )

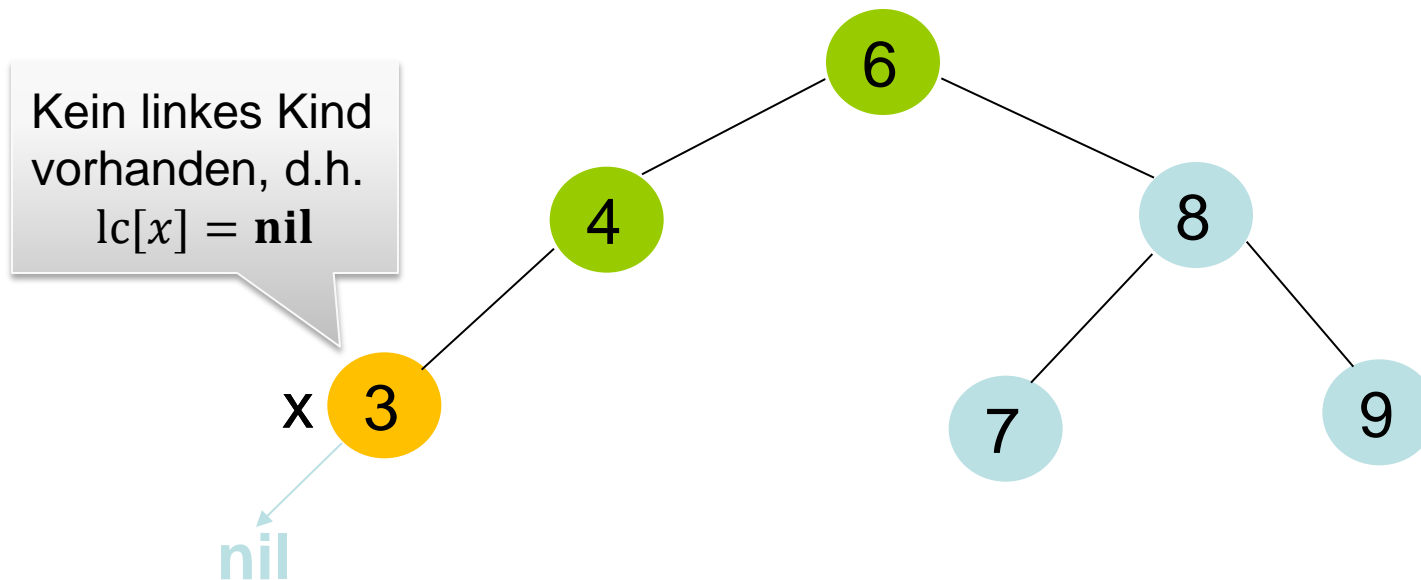
1. **if  $x \neq \text{nil}$  then**
2.     Inorder-Tree-Walk(lc[ $x$ ])
3.     Ausgabe key[ $x$ ]
4.     Inorder-Tree-Walk(rc[ $x$ ])



## Datenstrukturen

Inorder-Tree-Walk( $x$ )

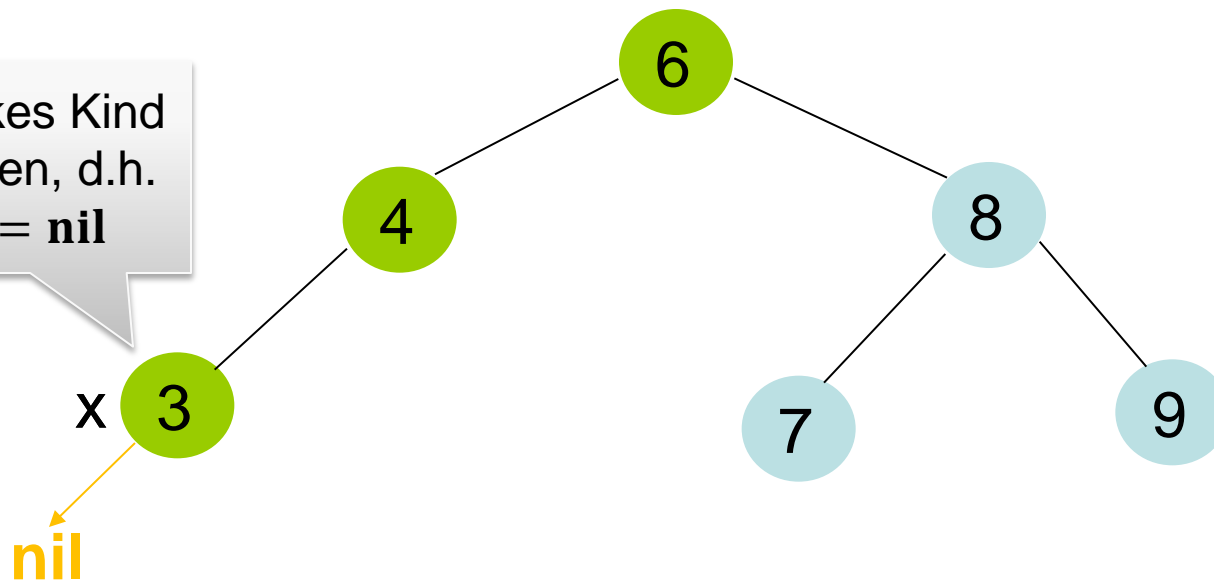
1. **if  $x \neq \text{nil}$  then**
2. **Inorder-Tree-Walk( $\text{lc}[x]$ )**
3. Ausgabe  $\text{key}[x]$
4. Inorder-Tree-Walk( $\text{rc}[x]$ )



## Datenstrukturen

Inorder-Tree-Walk( $x$ )

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





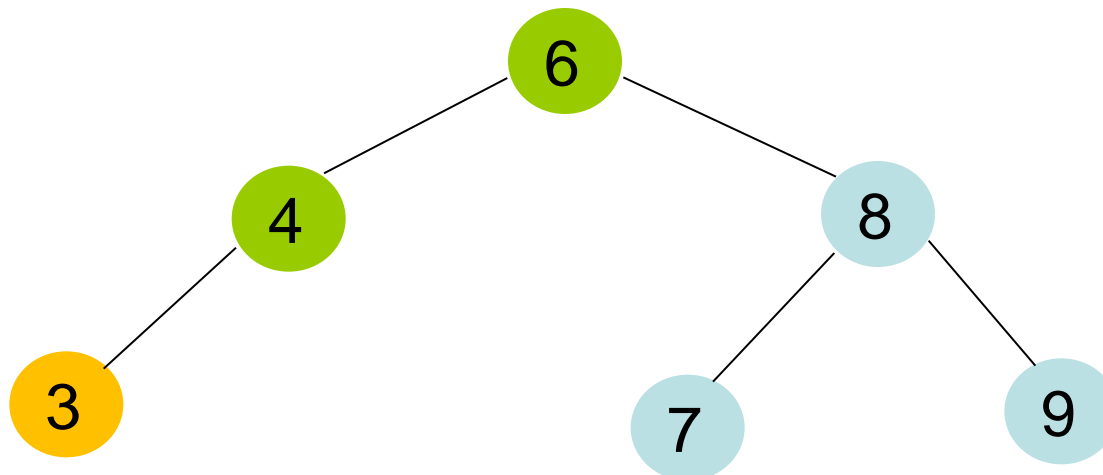
## Datenstrukturen

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



## Datenstrukturen

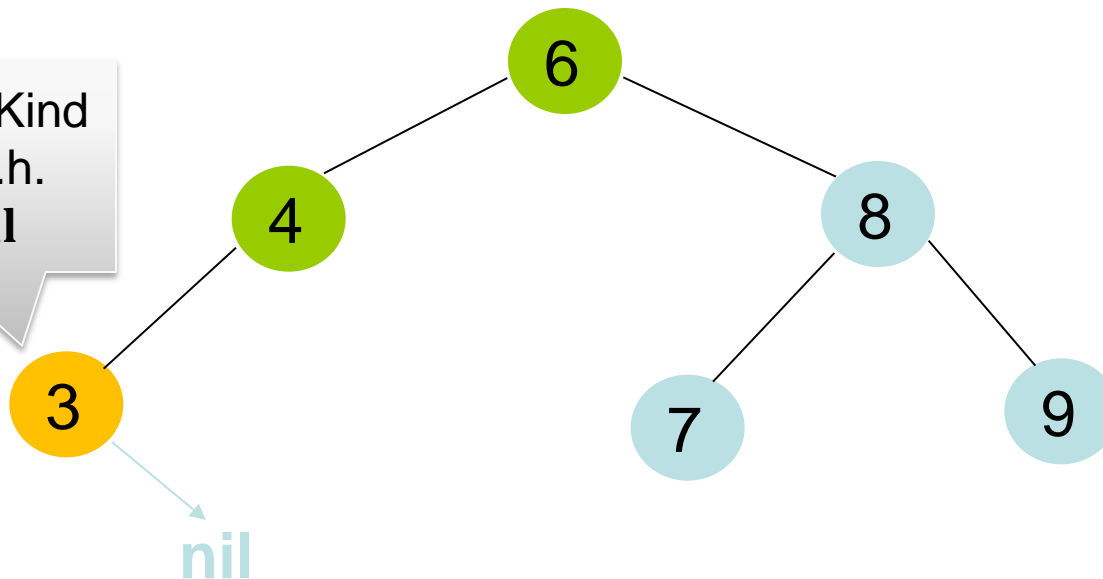
Inorder-Tree-Walk( $x$ )

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

*Ausgabe:*

3

Kein rechtes Kind  
vorhanden, d.h.  
 $\text{rc}[x] = \text{nil}$



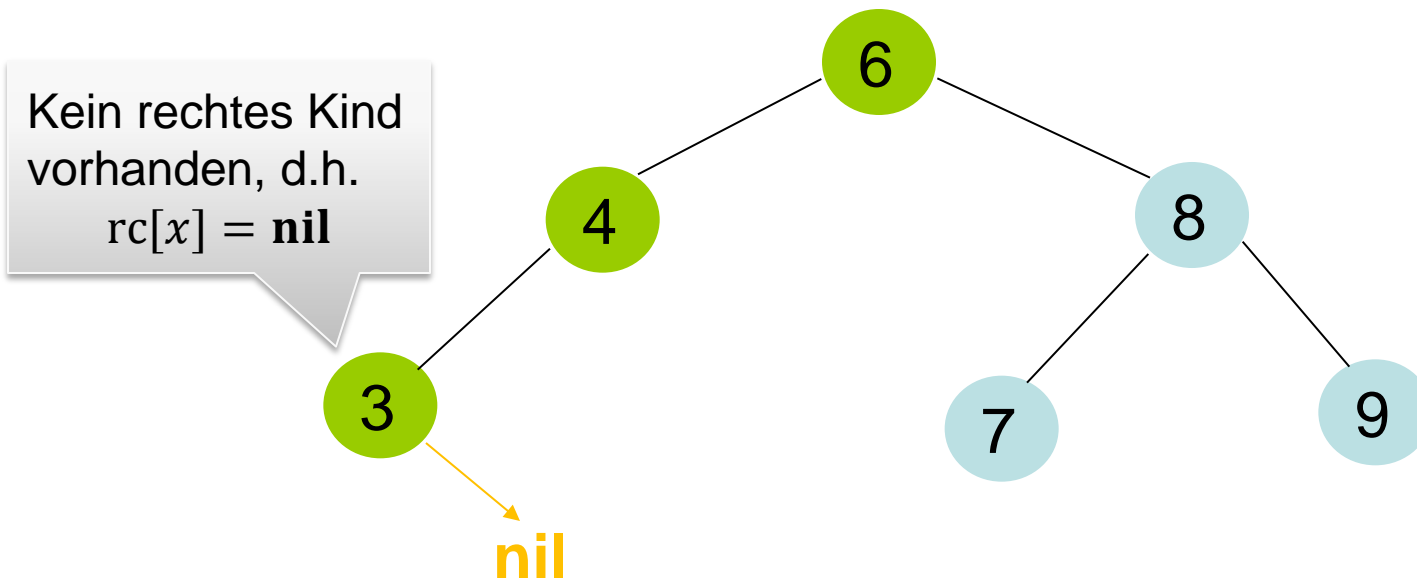
## Datenstrukturen

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



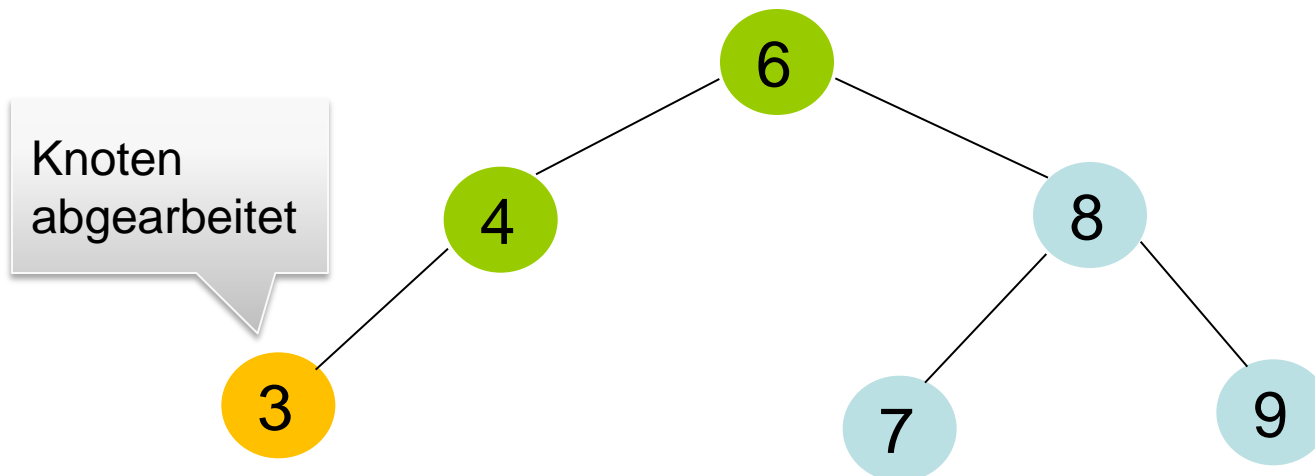
## Datenstrukturen

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



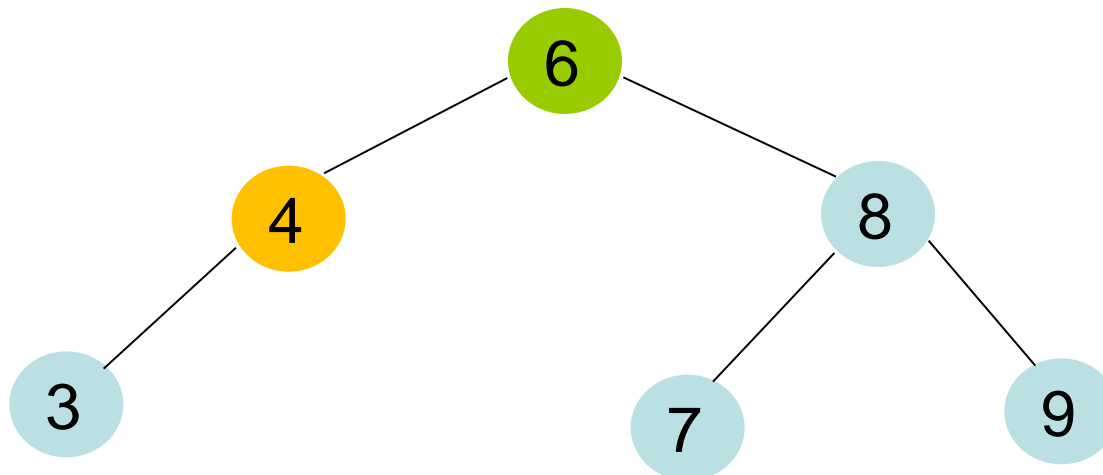
## Datenstrukturen

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



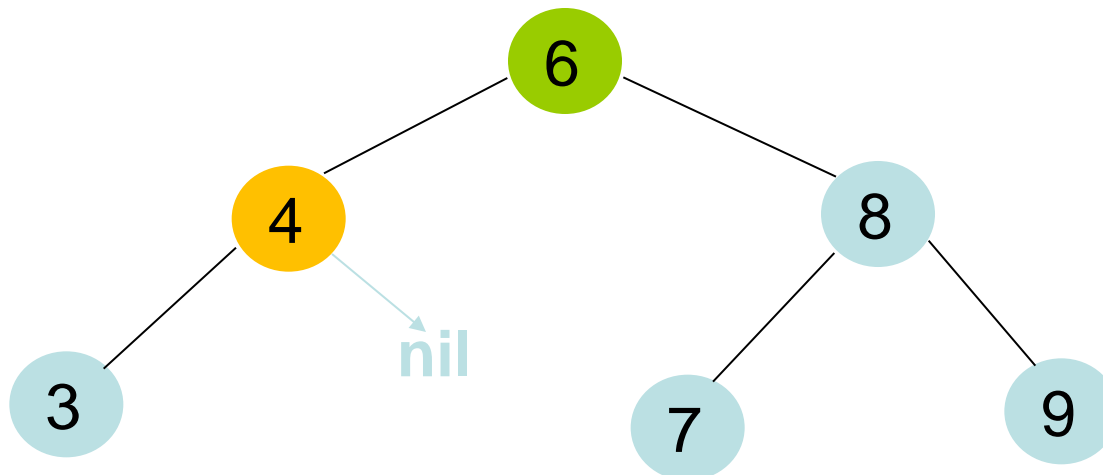
## Datenstrukturen

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



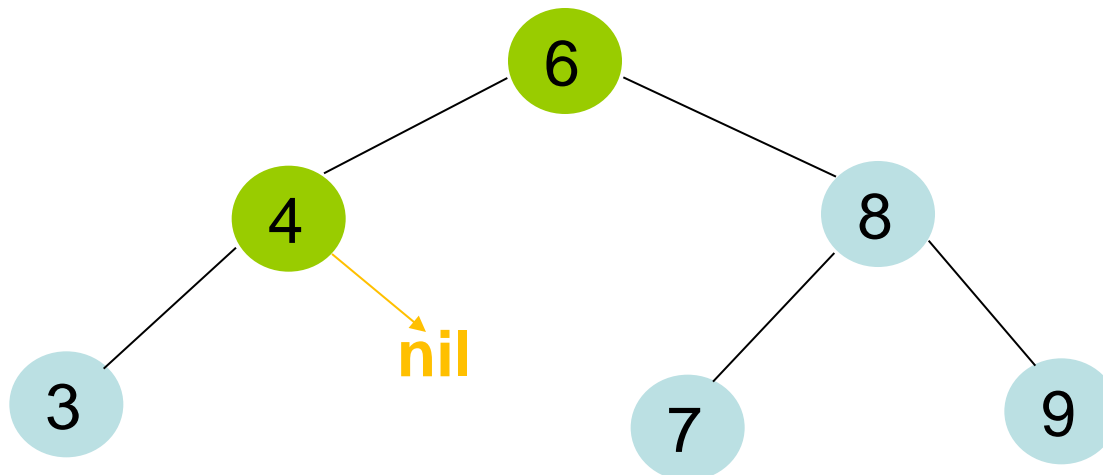
## Datenstrukturen

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



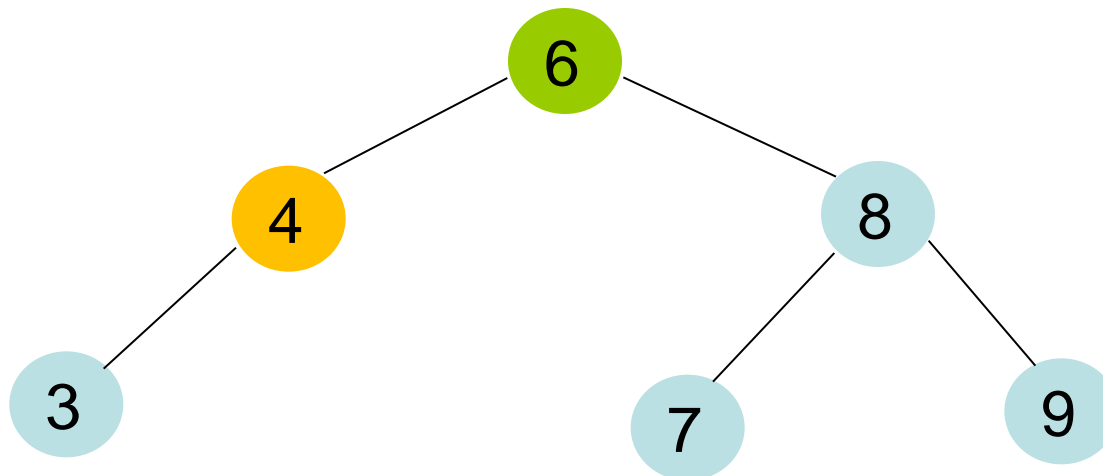
## Datenstrukturen

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





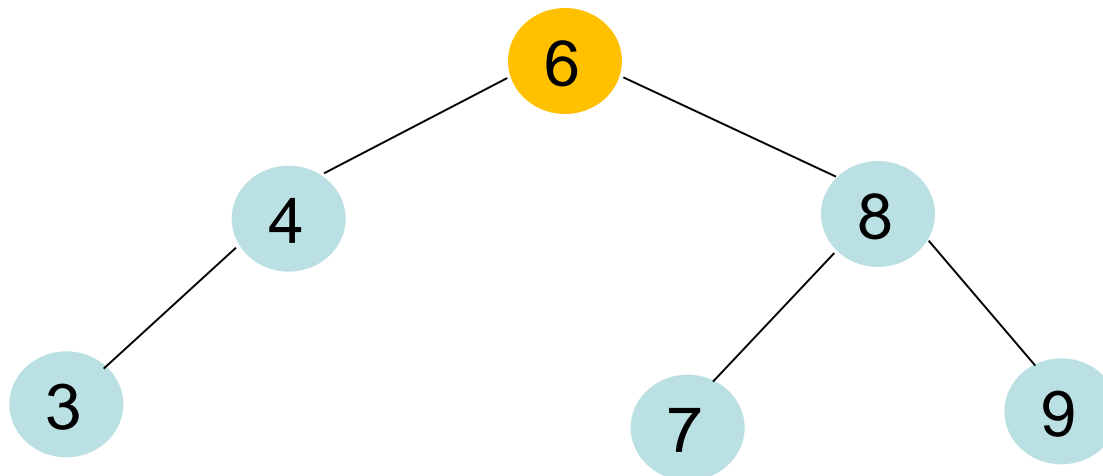
## Datenstrukturen

Inorder-Tree-Walk( $x$ )

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

*Ausgabe:*

3, 4, 6



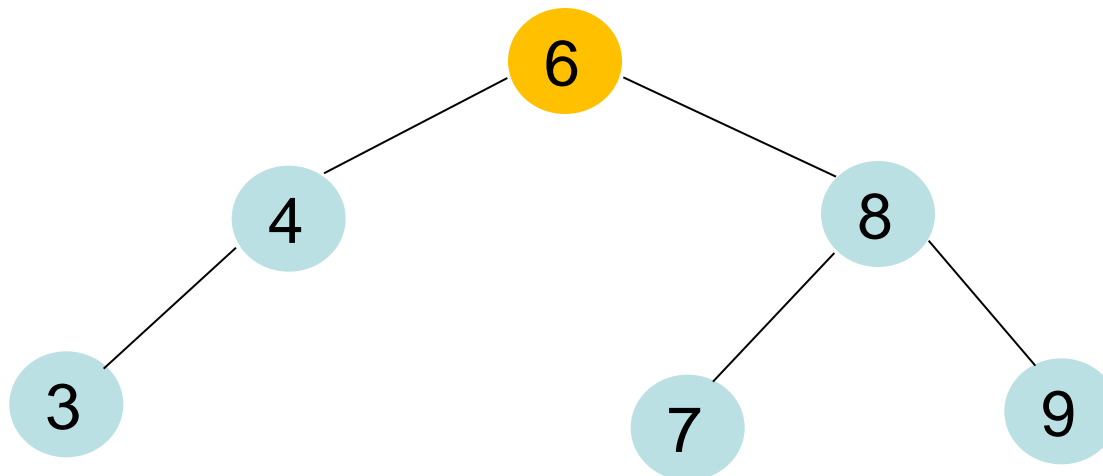
## Datenstrukturen

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



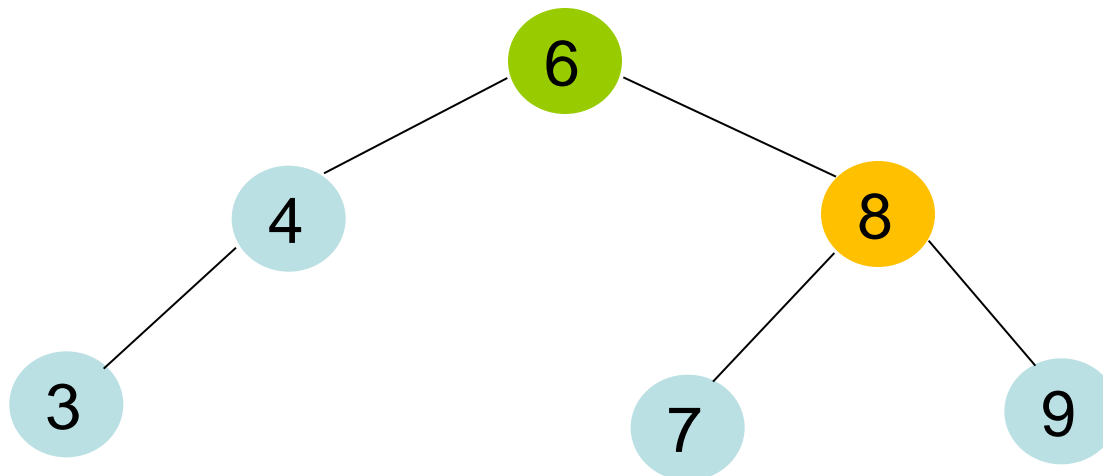
## Datenstrukturen

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



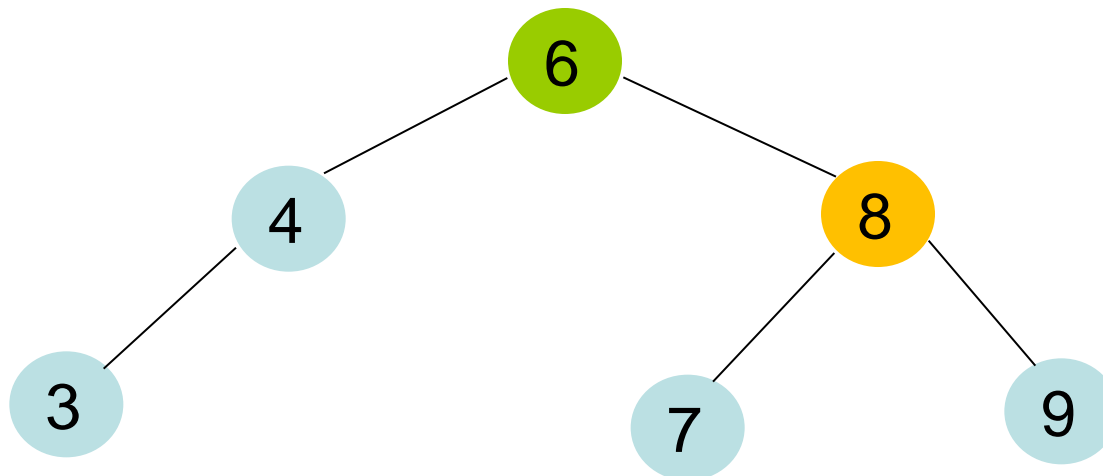
## Datenstrukturen

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



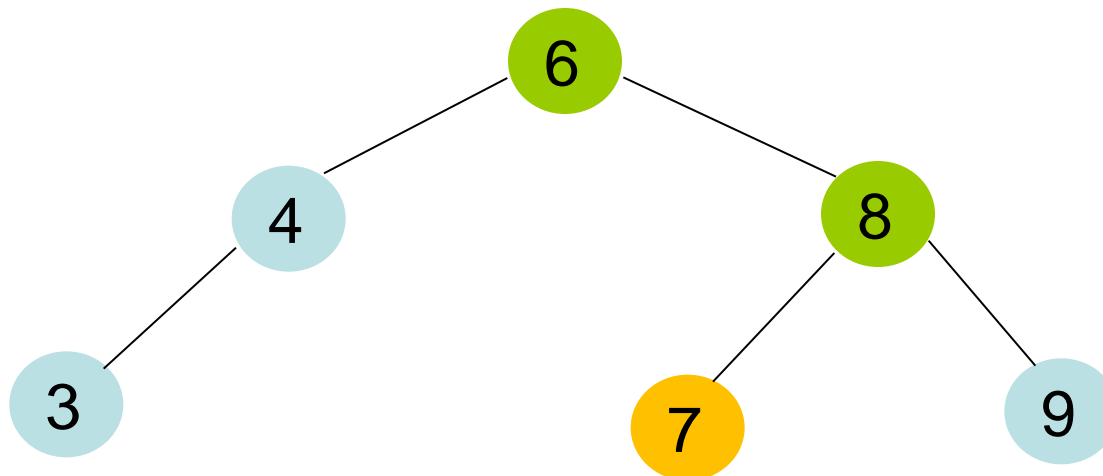
## Datenstrukturen

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



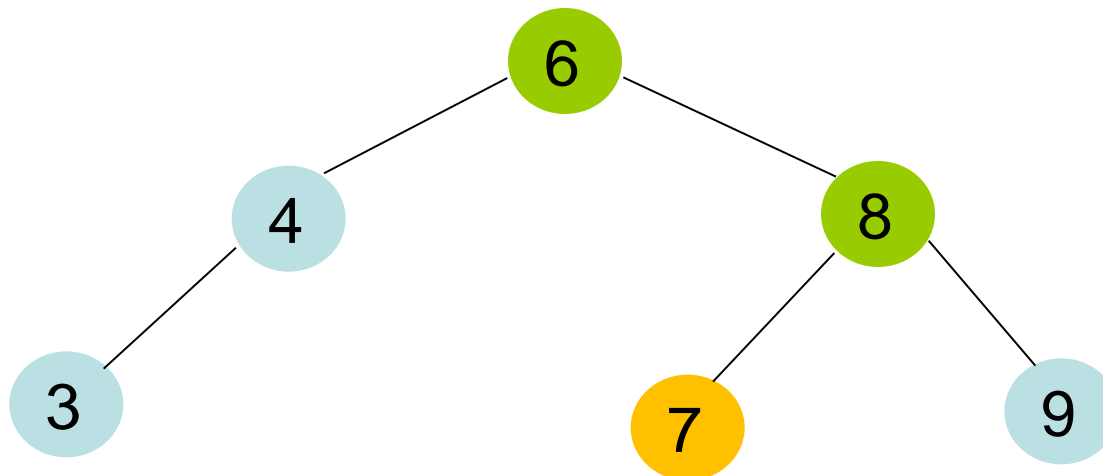
## Datenstrukturen

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



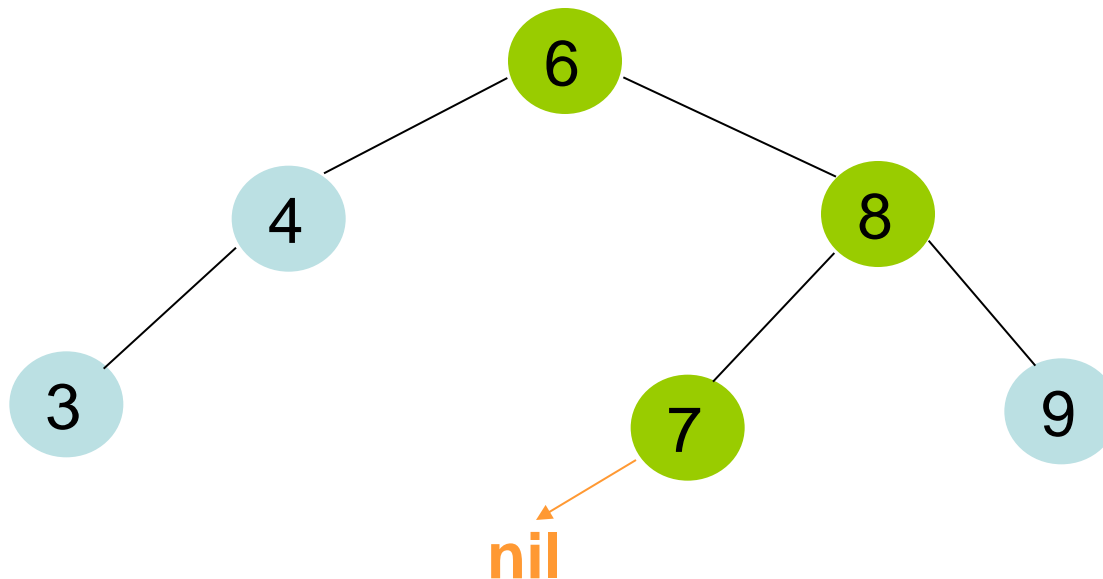
## Datenstrukturen

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



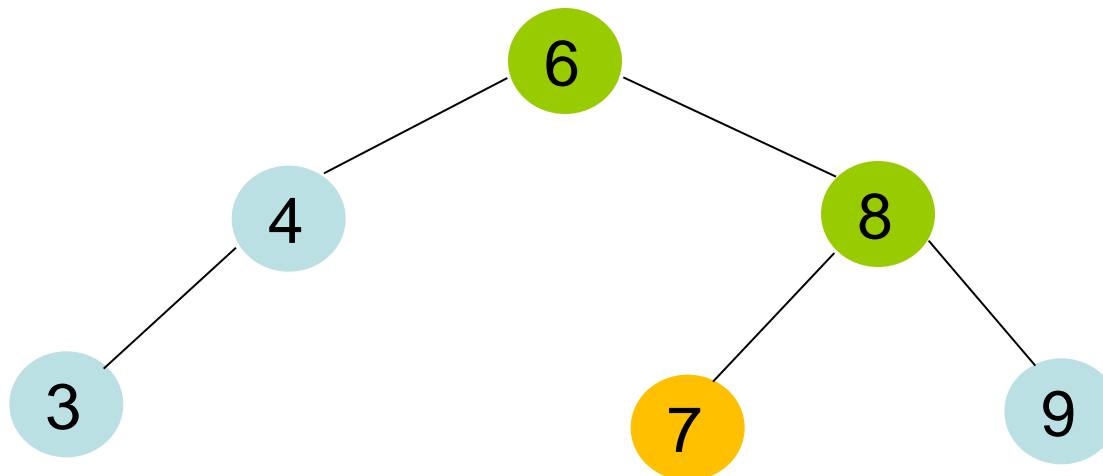
## Datenstrukturen

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





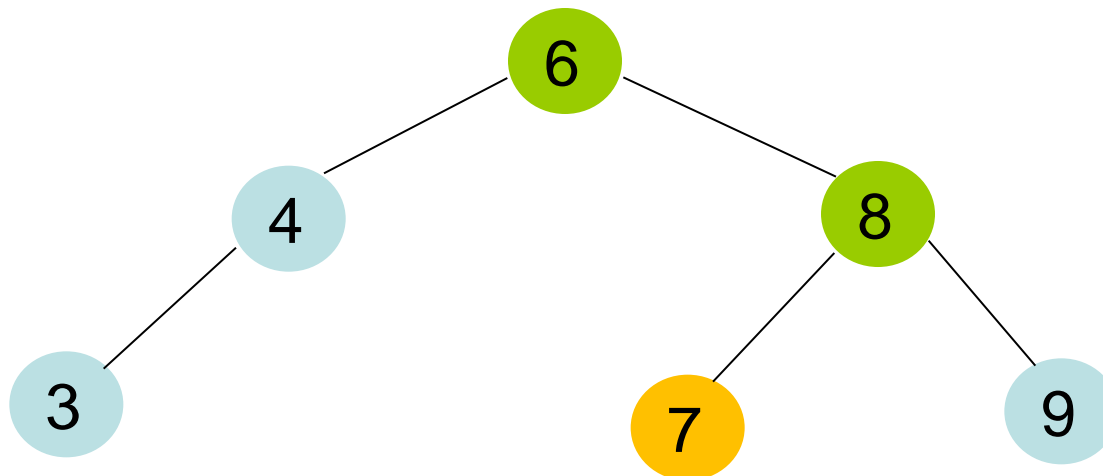
## Datenstrukturen

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



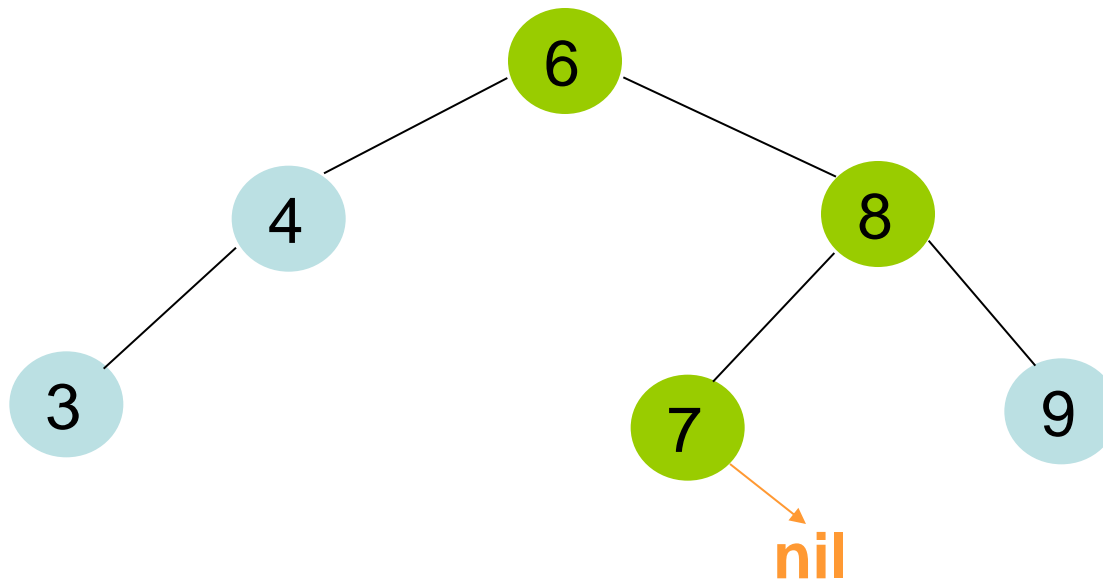
## Datenstrukturen

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



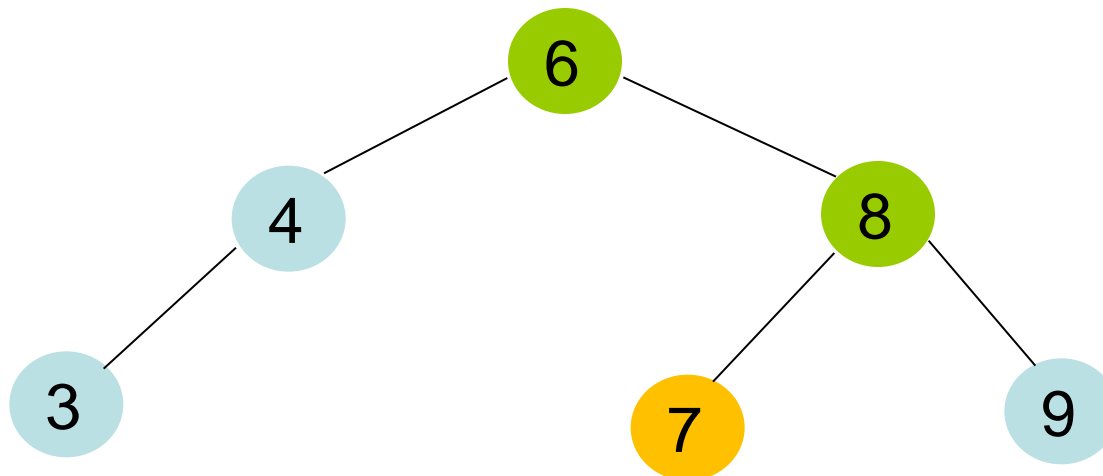
## Datenstrukturen

Inorder-Tree-Walk( $x$ )

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

*Ausgabe:*

3, 4, 6, 7

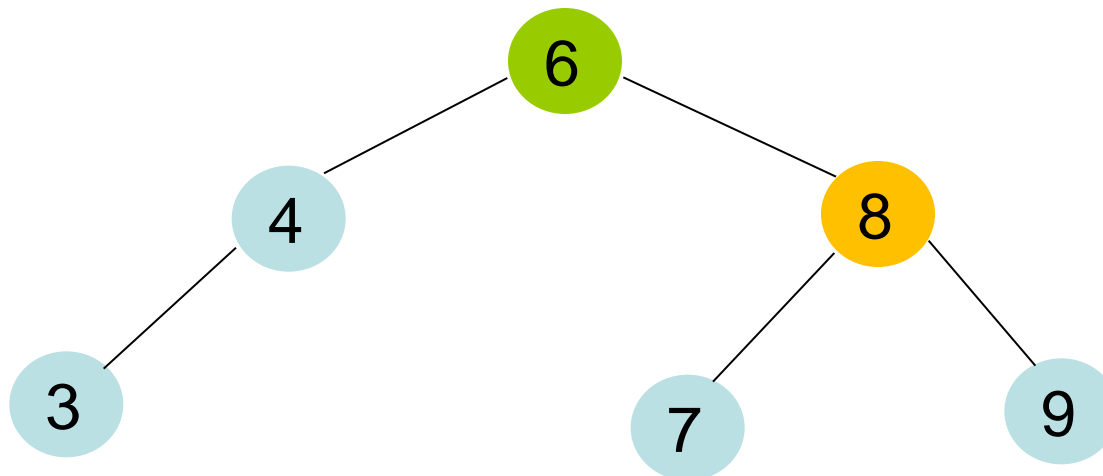


## Datenstrukturen

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, 8

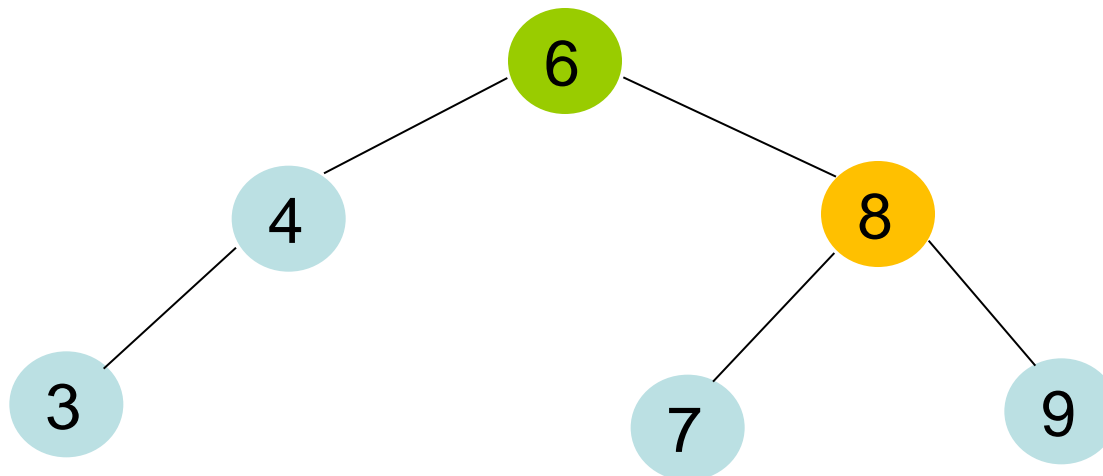


## Datenstrukturen

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, 8

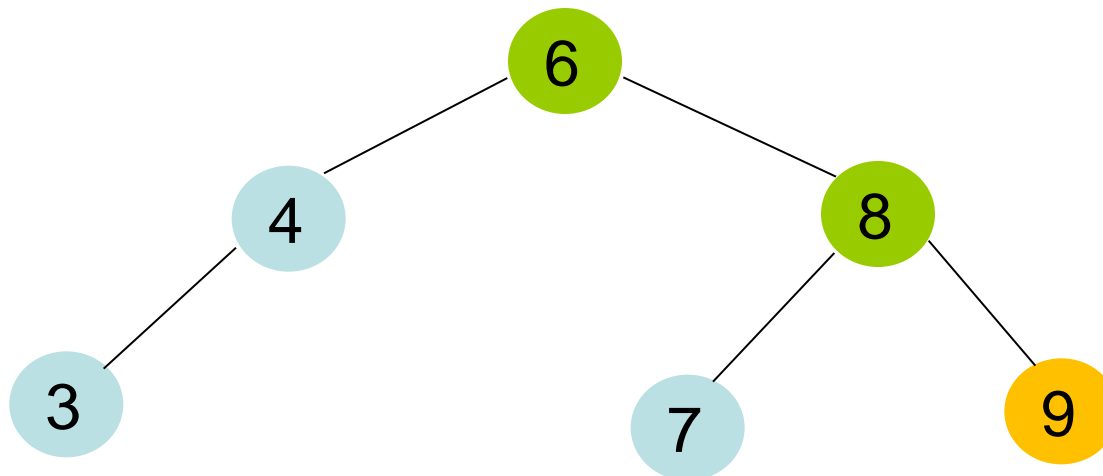


## Datenstrukturen

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, 8

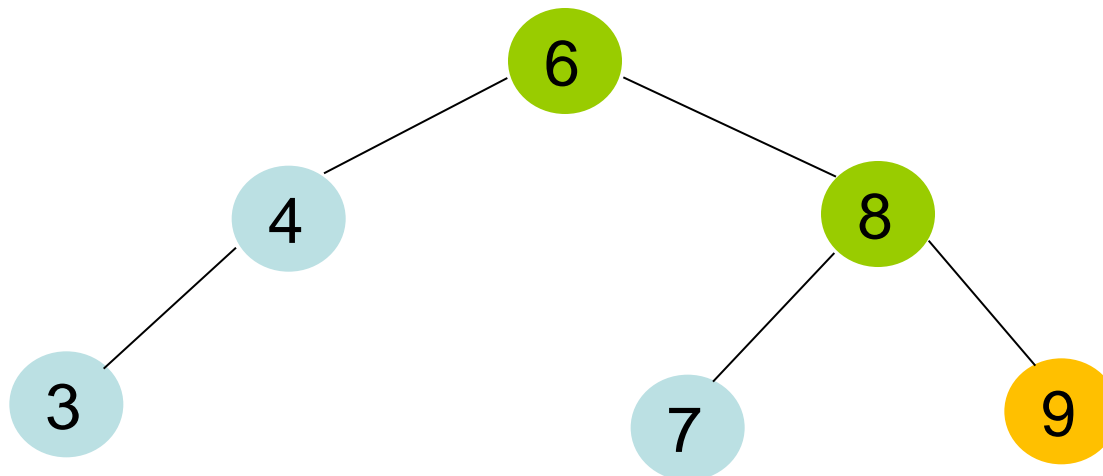


## Datenstrukturen

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, 8

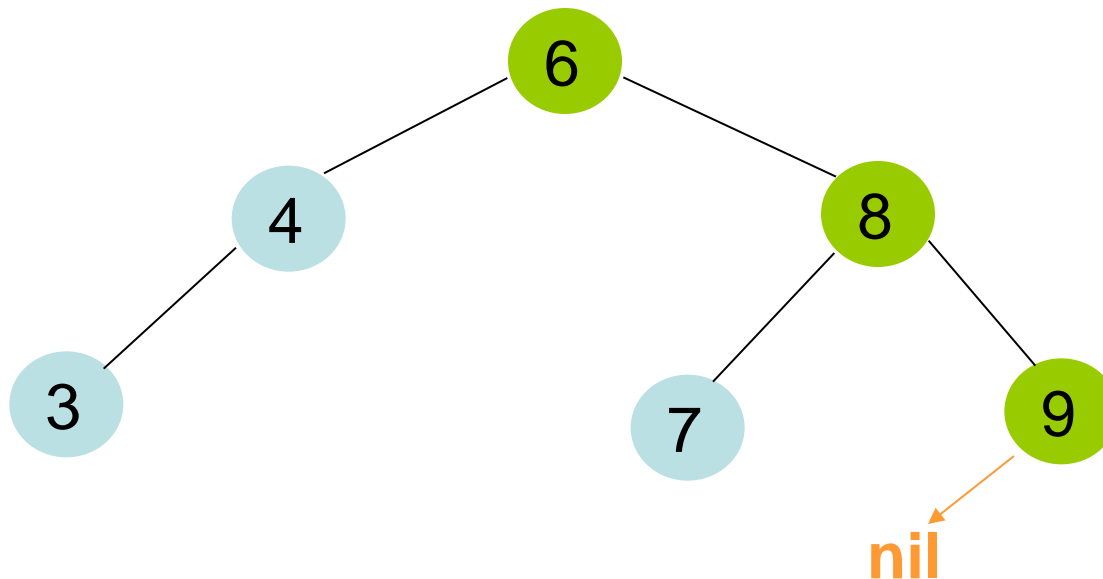


## Datenstrukturen

Inorder-Tree-Walk( $x$ )

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

*Ausgabe:*  
3, 4, 6, 7, 8





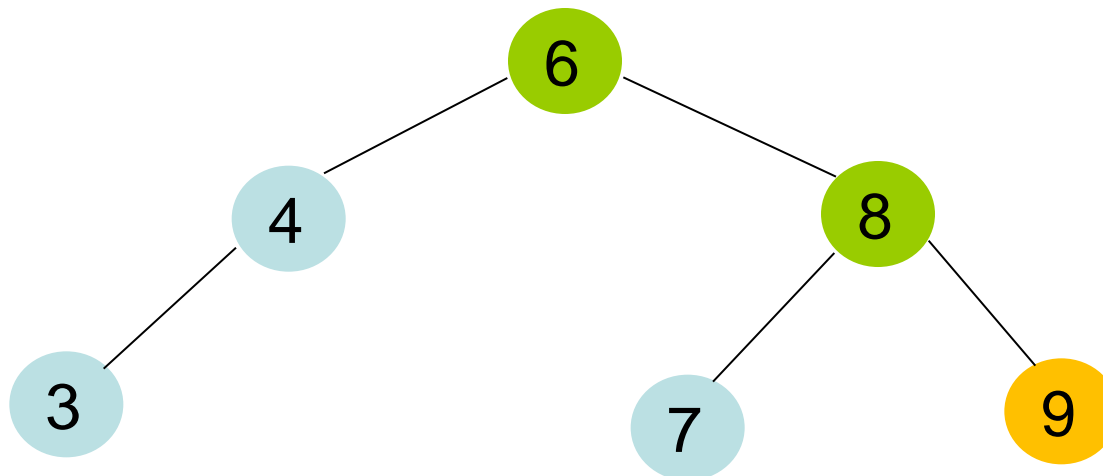
## Datenstrukturen

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, 8, 9



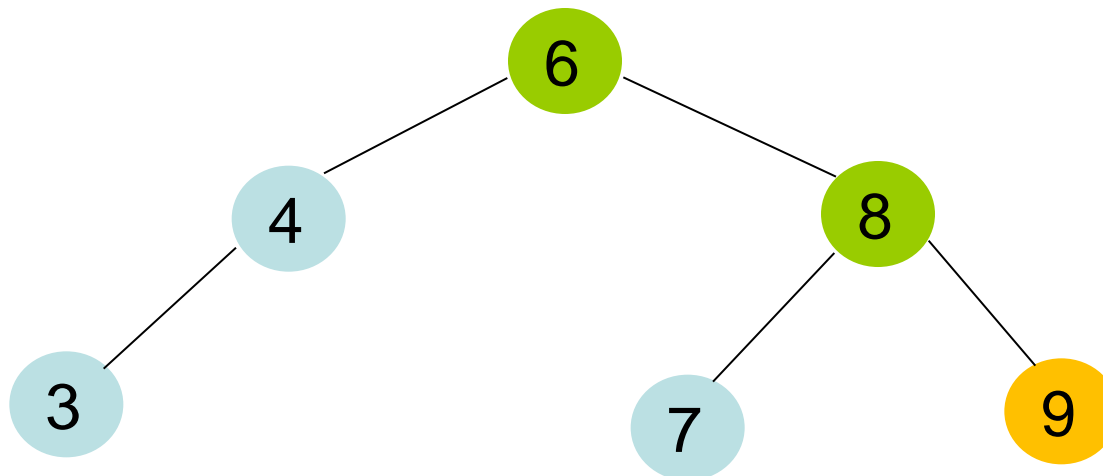
## Datenstrukturen

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, 8, 9



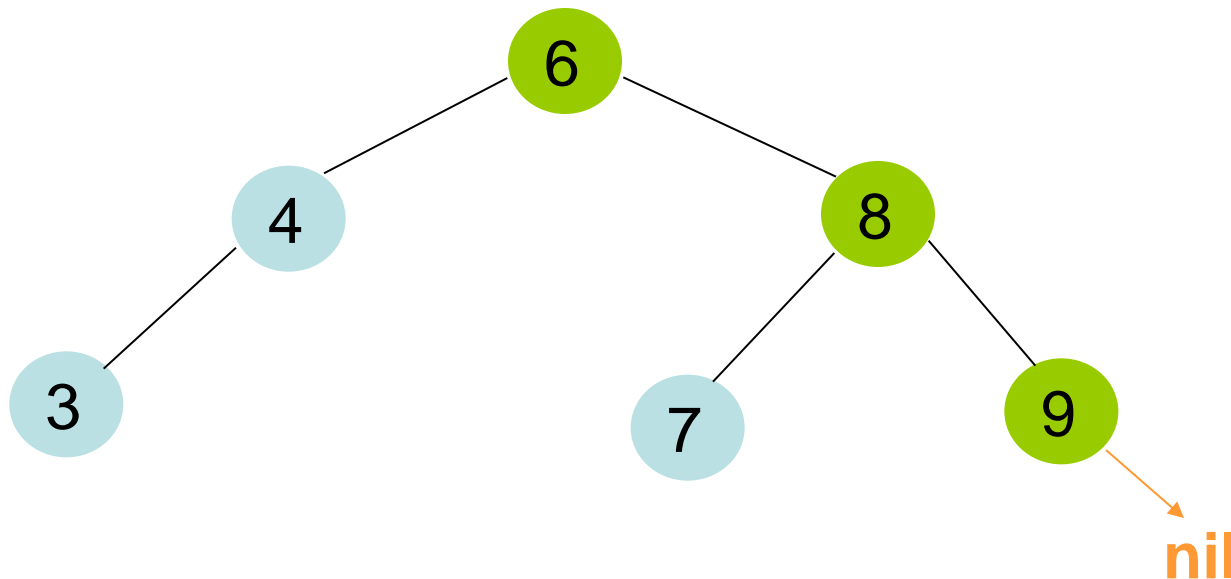
## Datenstrukturen

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, 8, 9



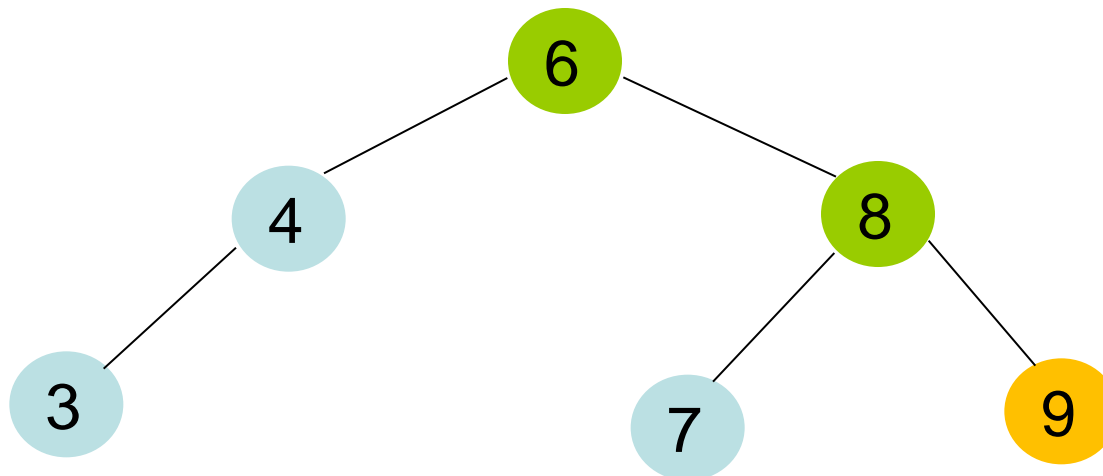
## Datenstrukturen

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, 8, 9



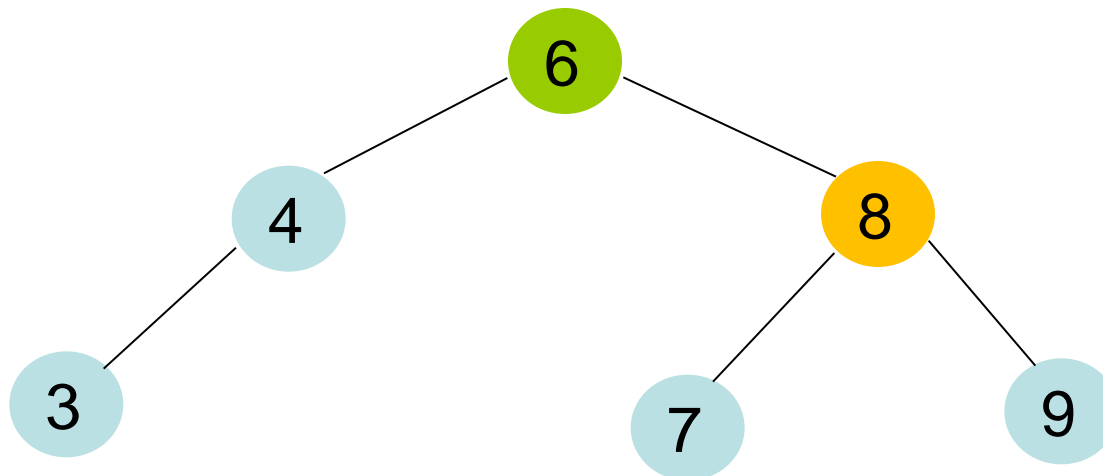
## Datenstrukturen

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, 8, 9



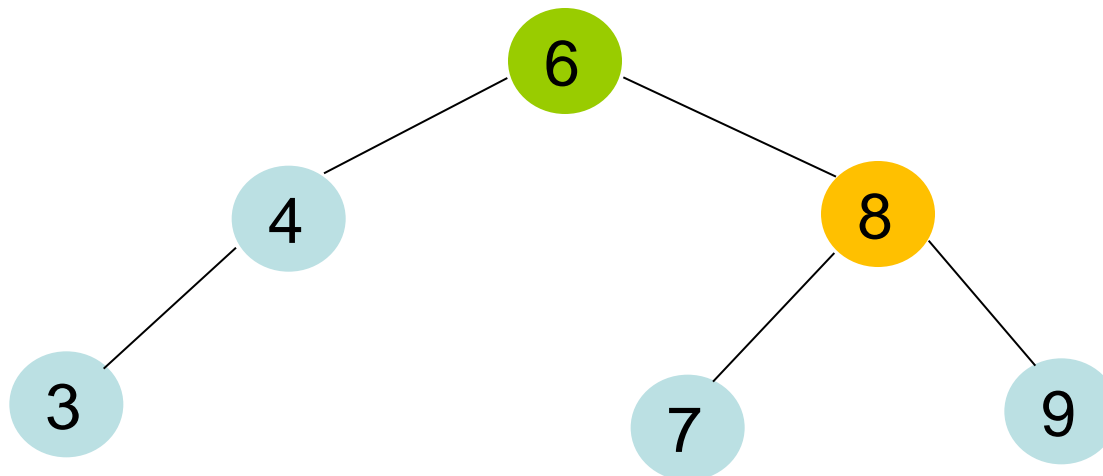
## Datenstrukturen

Inorder-Tree-Walk( $x$ )

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

*Ausgabe:*

3, 4, 6, 7, 8, 9



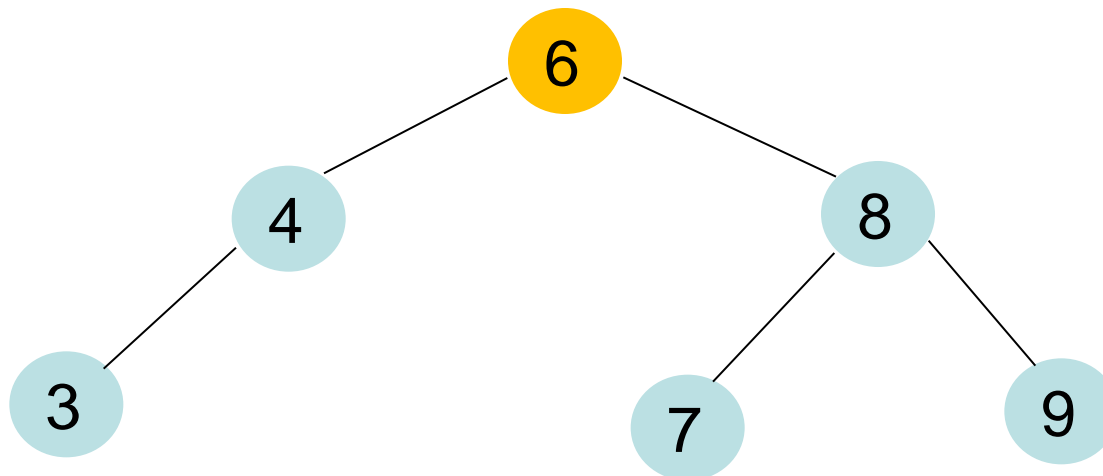
## Datenstrukturen

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, 8, 9



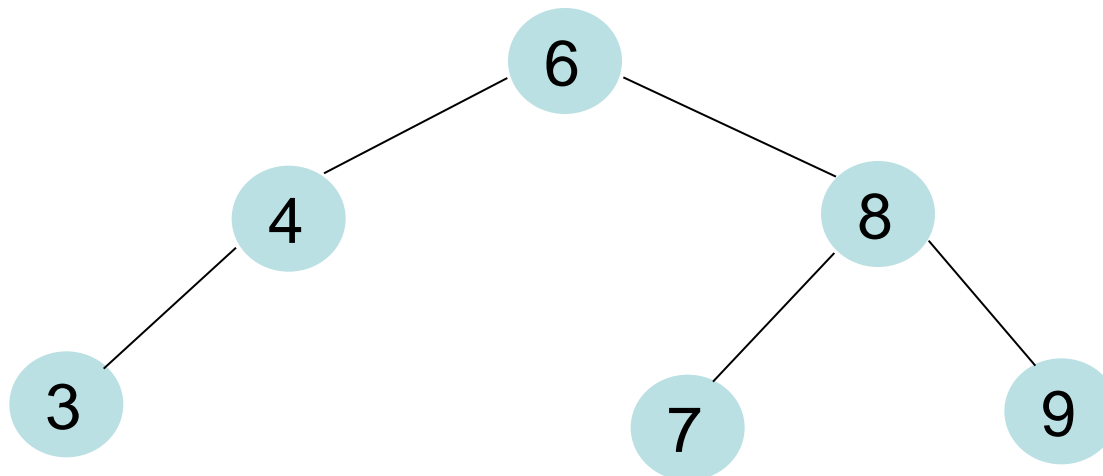
## Datenstrukturen

Inorder-Tree-Walk( $x$ )

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

*Ausgabe:*

3, 4, 6, 7, 8, 9





## Datenstrukturen

### *Lemma 33*

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

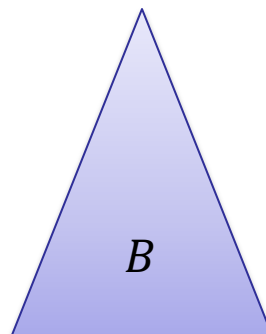
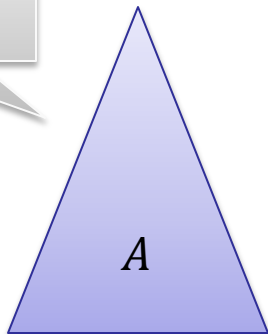
## Datenstrukturen

### *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*



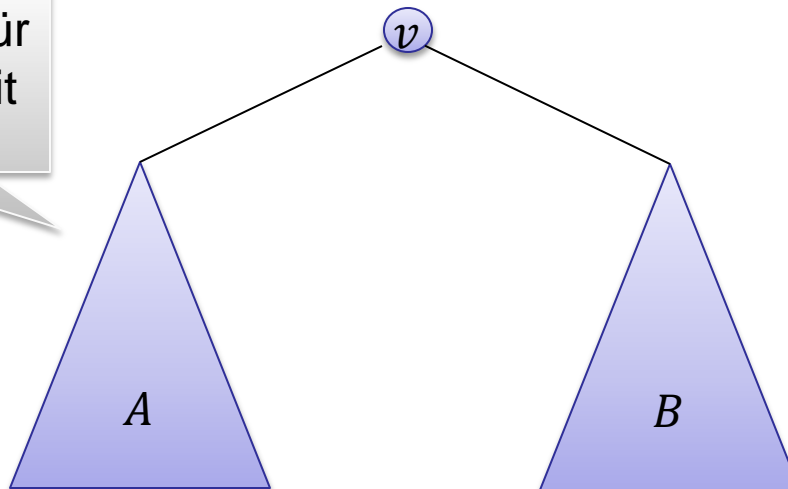
## Datenstrukturen

### *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  $v$



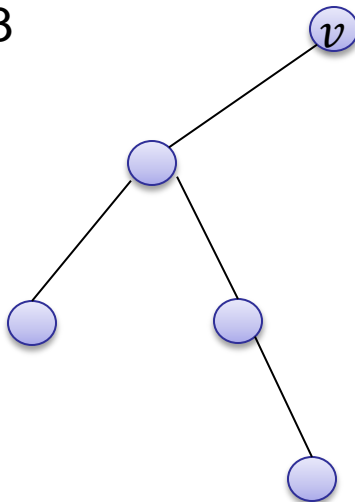
## Datenstrukturen

### 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



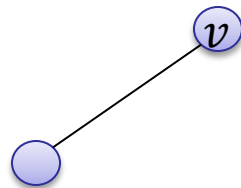
## Datenstrukturen

### 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 1



## 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 0



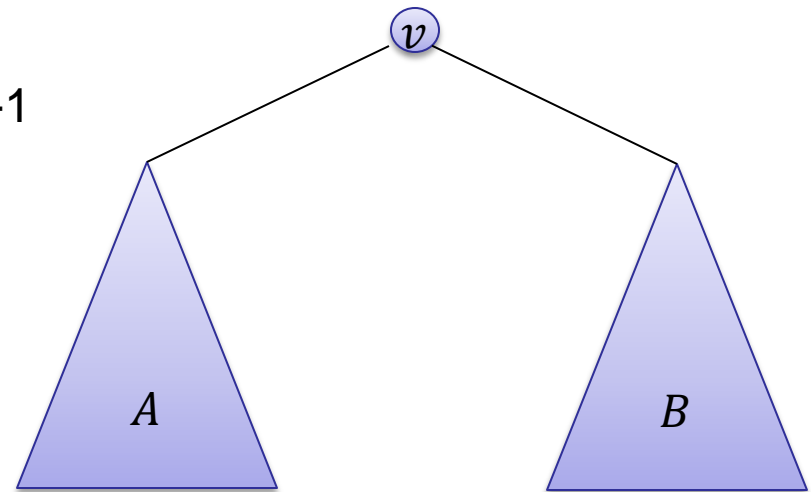
## Datenstrukturen

### 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  
 $1 + \max\{\text{Höhe}(A), \text{Höhe}(B)\}$

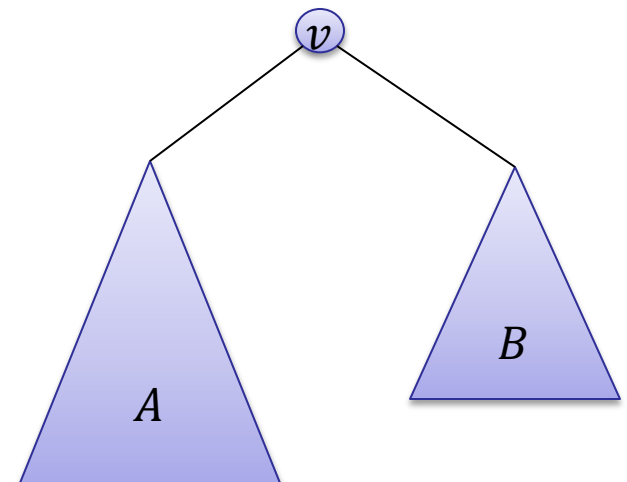




## Datenstrukturen

### *Induktion über die Struktur von Binärbäumen*

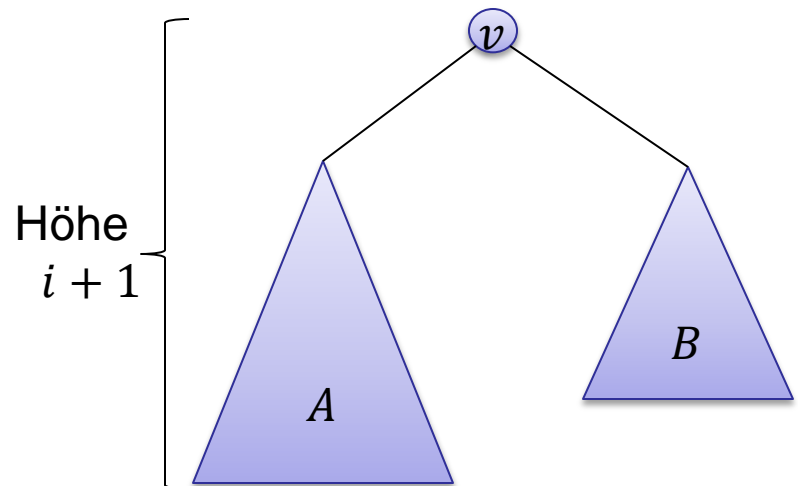
- Wir wollen Aussage  $A(i)$  durch Induktion über die Höhe von Bäumen zeigen
- (a) Zeige die Aussage für leere Bäume (Bäume der Höhe -1)
- (b) Zeige: Gilt die Aussage für Bäume der Höhe  $i$ , so gilt sie auch für Bäume der Höhe  $i + 1$



## Datenstrukturen

### Induktion über die Struktur von Binärbäumen

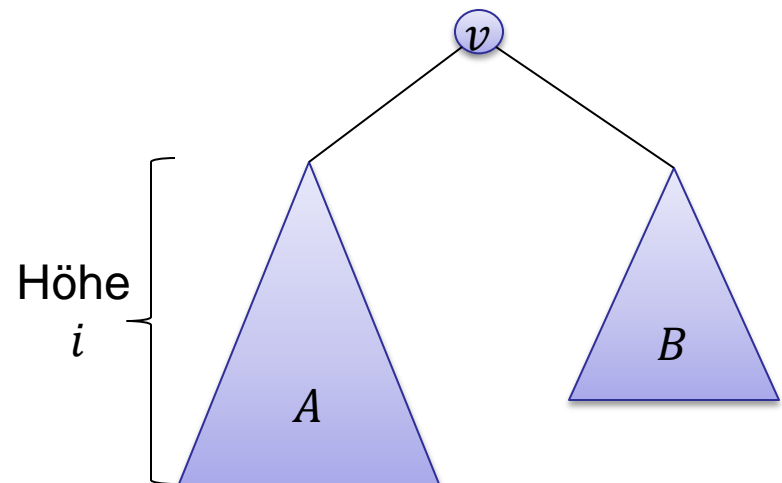
- Wir wollen Aussage  $A(i)$  durch Induktion über die Höhe von Bäumen zeigen
- (a) Zeige die Aussage für leere Bäume (Bäume der Höhe -1)
- (b) Zeige: Gilt die Aussage für Bäume der Höhe  $i$ , so gilt sie auch für Bäume der Höhe  $i + 1$
- Dabei können wir immer annehmen, dass ein Baum der Höhe  $i + 1$  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 33*

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

### *Beweis*

- (I.A.) Leerer Baum: Keine Ausgabe, also korrekt

Inorder-Tree-Walk( $x$ )

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

## Datenstrukturen

### Lemma 33

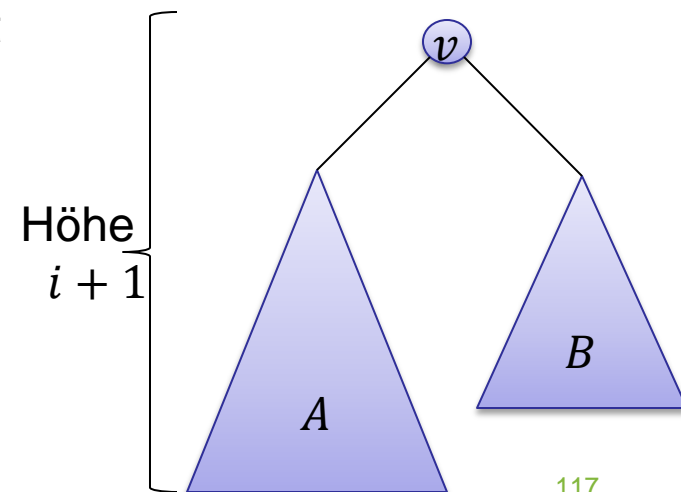
Inorder-Tree-Walk gibt die Schlüssel eines binären Suchbaums in aufsteigender Reihenfolge aus.

### Beweis

- (I.A.) Leerer Baum: Keine Ausgabe, also korrekt
- (I.V.) Lemma gilt für Bäume der Höhe  $\leq i$ .

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 33

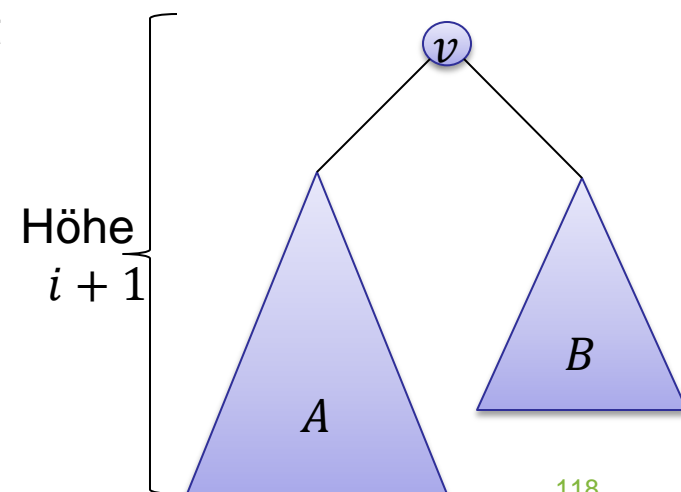
Inorder-Tree-Walk gibt die Schlüssel eines binären Suchbaums in aufsteigender Reihenfolge aus.

### Beweis

- (I.A.) Leerer Baum: Keine Ausgabe, also korrekt
- (I.V.) Lemma gilt für Bäume der Höhe  $\leq i$ .
- (I.S.) z.z.: Lemma gilt auch für Höhe  $i + 1 \geq 0$ .

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 33

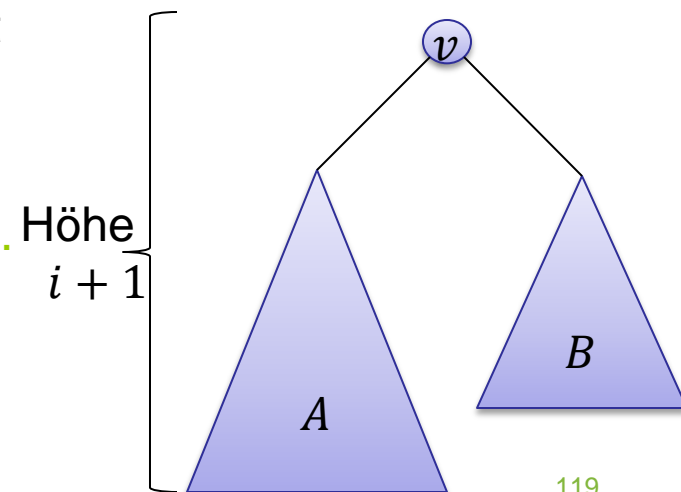
Inorder-Tree-Walk gibt die Schlüssel eines binären Suchbaums in aufsteigender Reihenfolge aus.

### Beweis

- (I.A.) Leerer Baum: Keine Ausgabe, also korrekt
- (I.V.) Lemma gilt für Bäume der Höhe  $\leq i$ .
- (I.S.) z.z.: Lemma gilt auch für Höhe  $i + 1 \geq 0$ .
- Betrachte Inorder-Tree-Walk auf 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 33

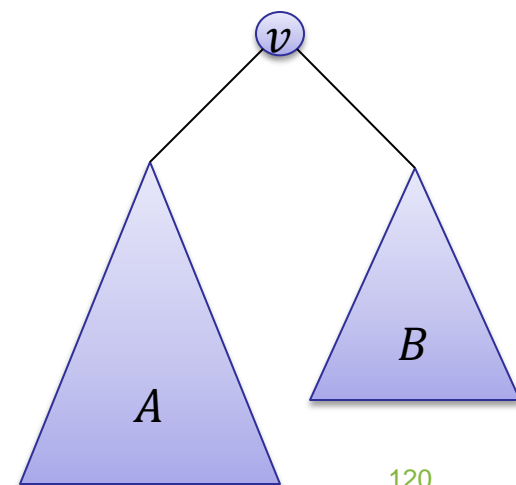
Inorder-Tree-Walk gibt die Schlüssel eines binären Suchbaums in aufsteigender Reihenfolge aus.

### Beweis

- (I.A.) Leerer Baum: Keine Ausgabe, also korrekt
- (I.V.) Lemma gilt für Bäume der Höhe  $\leq i$ .
- (I.S.) z.z.: Lemma gilt auch für Höhe  $i + 1 \geq 0$ .
- Betrachte Inorder-Tree-Walk auf solchem Baum.
- Nach Suchbaumeigenschaft sind alle Schlüssel in  $A$  kleiner oder gleich Schlüssel von  $v$

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 33

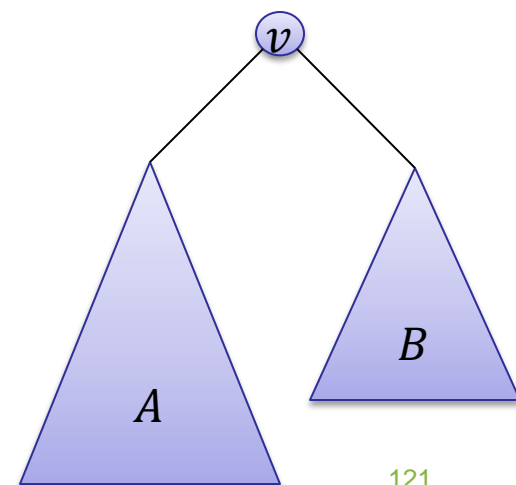
Inorder-Tree-Walk gibt die Schlüssel eines binären Suchbaums in aufsteigender Reihenfolge aus.

### Beweis

- (I.A.) Leerer Baum: Keine Ausgabe, also korrekt
- (I.V.) Lemma gilt für Bäume der Höhe  $\leq i$ .
- (I.S.) z.z.: Lemma gilt auch für Höhe  $i + 1 \geq 0$ .
- Betrachte Inorder-Tree-Walk auf solchem Baum.
- Nach Suchbaumeigenschaft sind alle Schlüssel in  $A$  kleiner oder gleich Schlüssel von  $v$
- Zeile 2: Aufruf für Teilbaum  $A$  der Höhe  $\leq i$   
Nach (I.V.): Schlüssel aus  $A$  werden 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 33

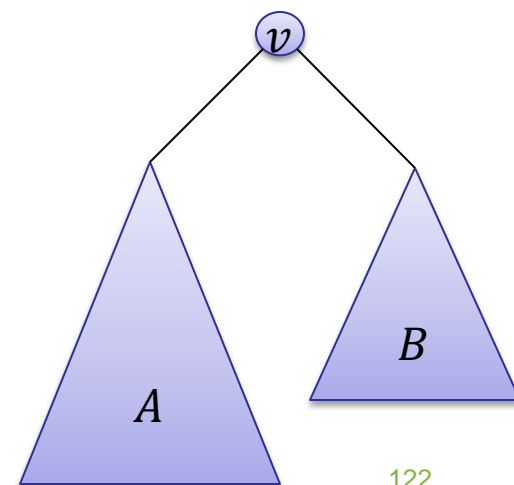
Inorder-Tree-Walk gibt die Schlüssel eines binären Suchbaums in aufsteigender Reihenfolge aus.

### Beweis

- (I.A.) Leerer Baum: Keine Ausgabe, also korrekt
- (I.V.) Lemma gilt für Bäume der Höhe  $\leq i$ .
- (I.S.) z.z.: Lemma gilt auch für Höhe  $i + 1 \geq 0$ .
- Betrachte Inorder-Tree-Walk auf solchem Baum.
- Nach Suchbaumeigenschaft sind alle Schlüssel in  $A$  kleiner oder gleich Schlüssel von  $v$
- Zeile 2: Aufruf für Teilbaum  $A$  der Höhe  $\leq i$   
Nach (I.V.): Schlüssel aus  $A$  werden 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 33*

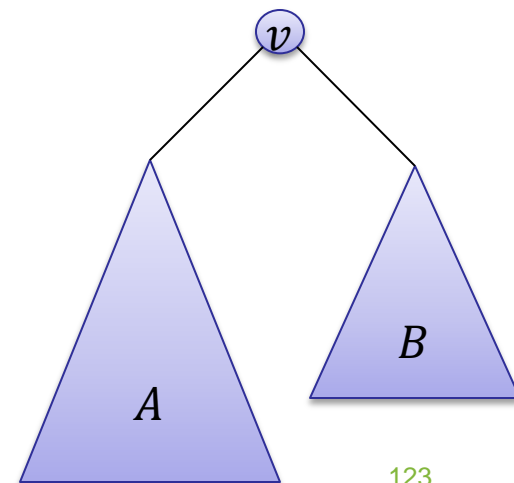
Inorder-Tree-Walk gibt die Schlüssel eines binären Suchbaums in aufsteigender Reihenfolge aus.

### *Beweis*

- Zeile 2: Aufruf für Teilbaum  $A$  der Höhe  $\leq i$   
Nach (I.V.): Schlüssel aus  $A$  werden 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 33

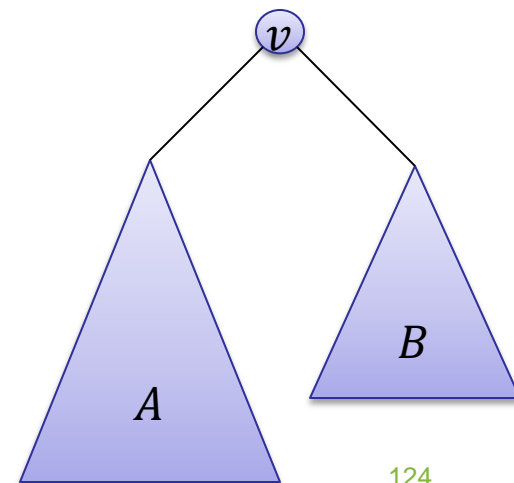
Inorder-Tree-Walk gibt die Schlüssel eines binären Suchbaums in aufsteigender Reihenfolge aus.

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[ $x$ ])

### Beweis

- Zeile 2: Aufruf für Teilbaum  $A$  der Höhe  $\leq i$   
Nach (I.V.): Schlüssel aus  $A$  werden in aufsteigender Reihenfolge ausgegeben
- Zeile 3: key[ $v$ ] wird ausgegeben
- Alle Schlüssel in Teilbaum  $B$  sind größer als Schlüssel von  $v$  (Suchbaumeigenschaft)



## Datenstrukturen

### Lemma 33

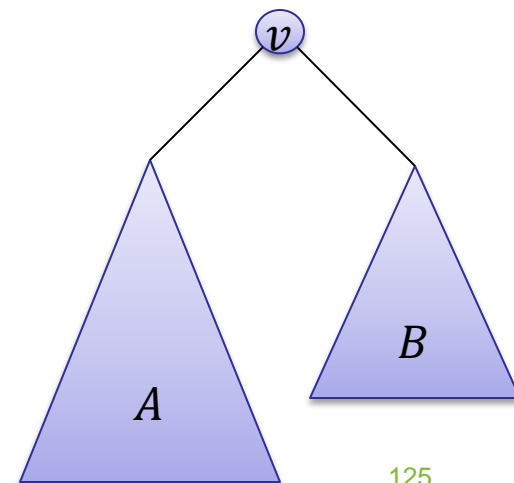
Inorder-Tree-Walk gibt die Schlüssel eines binären Suchbaums in aufsteigender Reihenfolge aus.

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[ $x$ ])

### Beweis

- Zeile 2: Aufruf für Teilbaum  $A$  der Höhe  $\leq i$   
Nach (I.V.): Schlüssel aus  $A$  werden in aufsteigender Reihenfolge ausgegeben
- Zeile 3: key[ $v$ ] wird ausgegeben
- Alle Schlüssel in Teilbaum  $B$  sind größer als Schlüssel von  $v$  (Suchbaumeigenschaft)
- Zeile 4: Aufruf für Teilbaum  $B$  der Höhe  $\leq i$   
Nach (I.V.): Schlüssel aus  $B$  werden in aufsteigender Reihenfolge ausgegeben



## Datenstrukturen

### Lemma 33

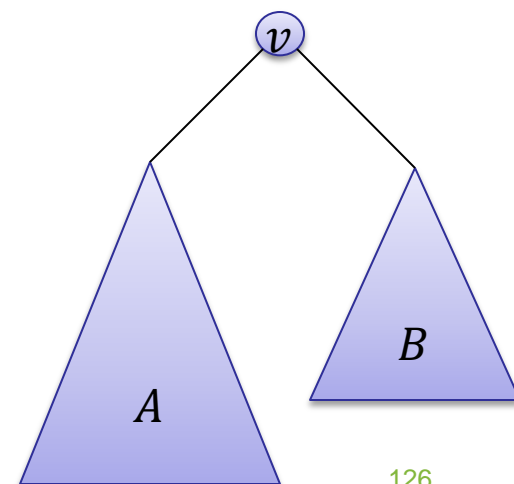
Inorder-Tree-Walk gibt die Schlüssel eines binären Suchbaums in aufsteigender Reihenfolge aus.

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[ $x$ ])

### Beweis

- Zeile 2: Aufruf für Teilbaum  $A$  der Höhe  $\leq i$   
Nach (I.V.): Schlüssel aus  $A$  werden in aufsteigender Reihenfolge ausgegeben
- Zeile 3: key[ $v$ ] wird ausgegeben
- Alle Schlüssel in Teilbaum  $B$  sind größer als Schlüssel von  $v$  (Suchbaumeigenschaft)
- Zeile 4: Aufruf für Teilbaum  $B$  der Höhe  $\leq i$   
Nach (I.V.): Schlüssel aus  $B$  werden in aufsteigender Reihenfolge ausgegeben



## Datenstrukturen

### Lemma 33

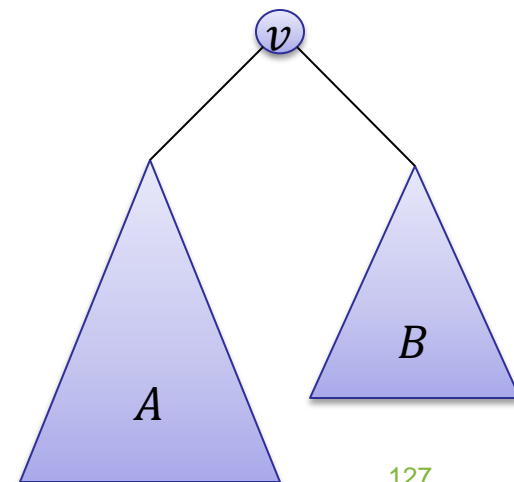
Inorder-Tree-Walk gibt die Schlüssel eines binären Suchbaums in aufsteigender Reihenfolge aus.

### Beweis

- Insgesamt:
- Schlüssel aus  $A$  aufsteigend, Schlüssel von  $v$ , Schlüssel aus  $B$  aufsteigend
- Nach Suchbaumeigenschaft ist dies aufsteigende Folge

Inorder-Tree-Walk( $x$ )

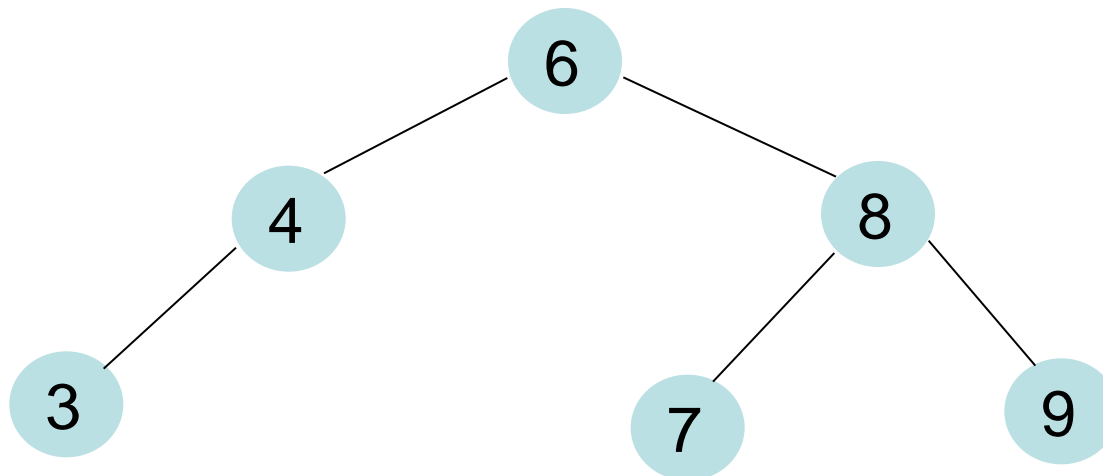
1. **if**  $x \neq \text{nil}$  **then**
2.   Inorder-Tree-Walk(lc[ $x$ ])
3.   Ausgabe key[ $x$ ]
4.   Inorder-Tree-Walk(rc[ $x$ ])



## Datenstrukturen

### *Suchen in Binärbäumen*

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

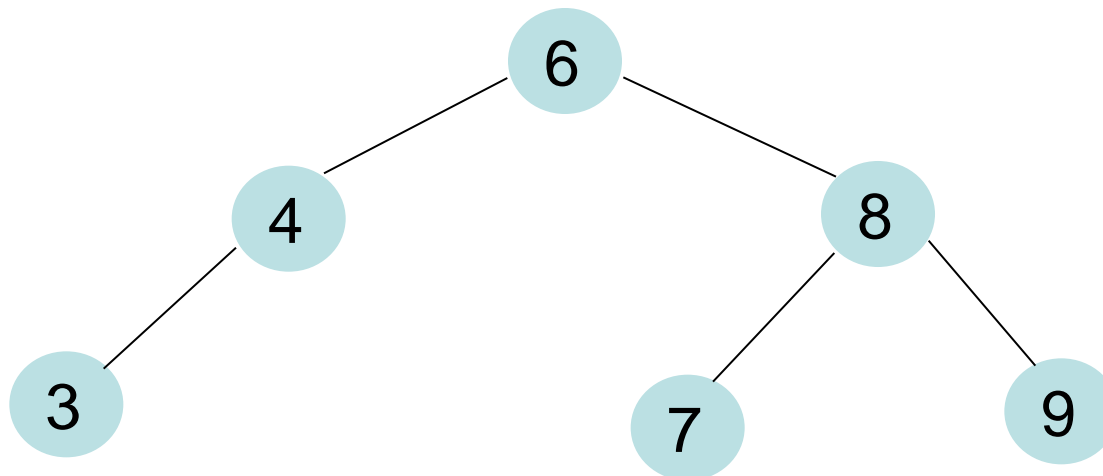




## Datenstrukturen

Baumsuche( $x, k$ )

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

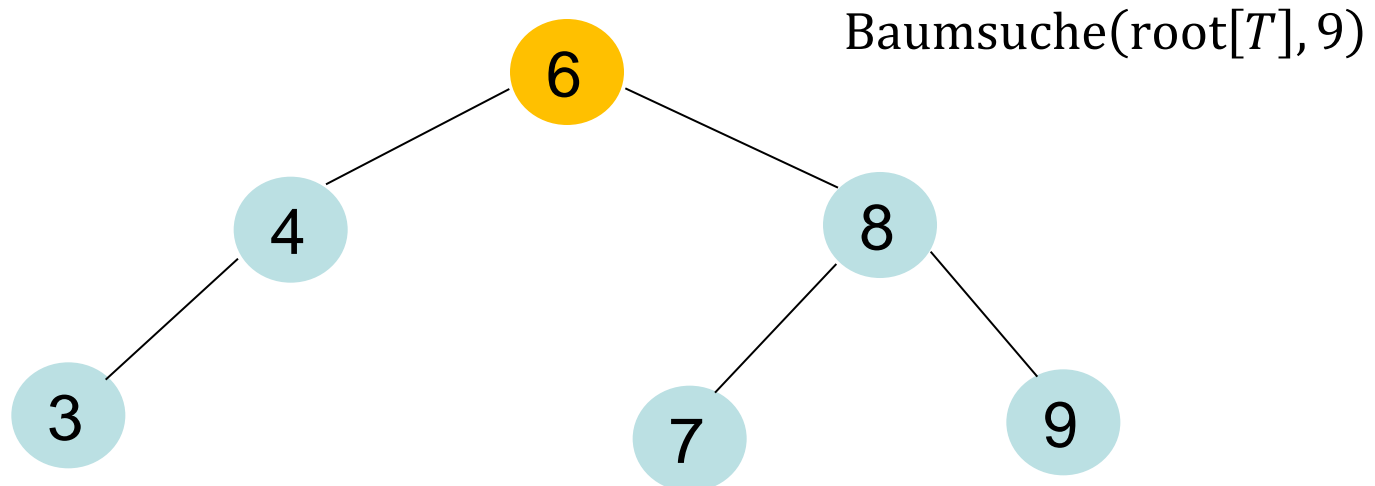


## Datenstrukturen

Baumsuche( $x, k$ )

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

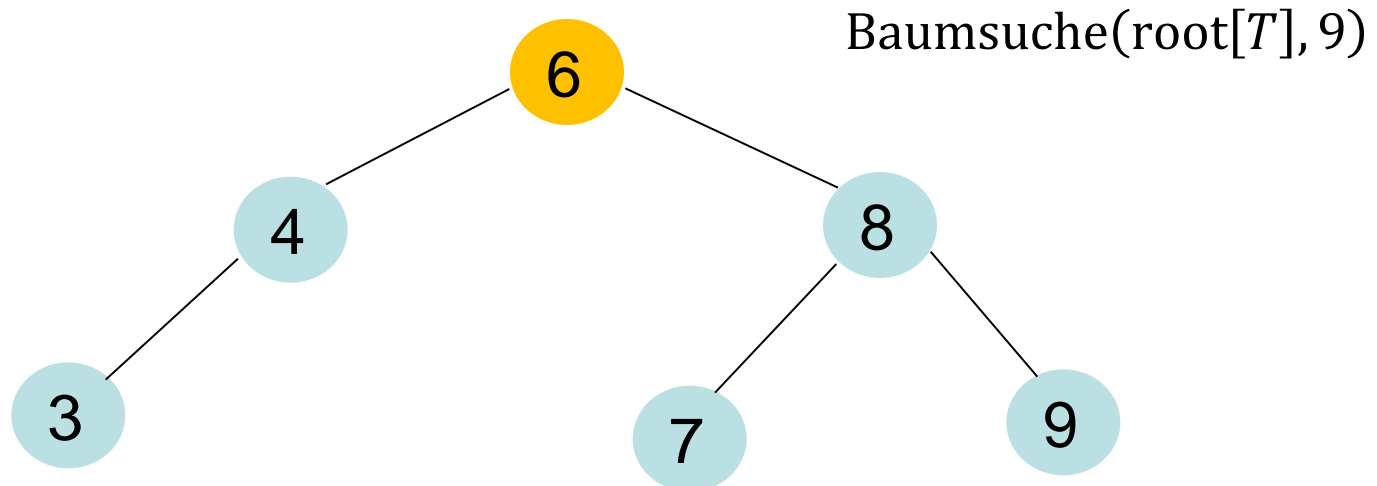
1. **if**  $x = \text{nil}$  **or**  $k = \text{key}[x]$  **then return**  $x$
2. **if**  $k < \text{key}[x]$  **then return** Baumsuche( $\text{lc}[x], k$ )
3. **else return** Baumsuche( $\text{rc}[x], k$ )



## Datenstrukturen

Baumsuche( $x, k$ )

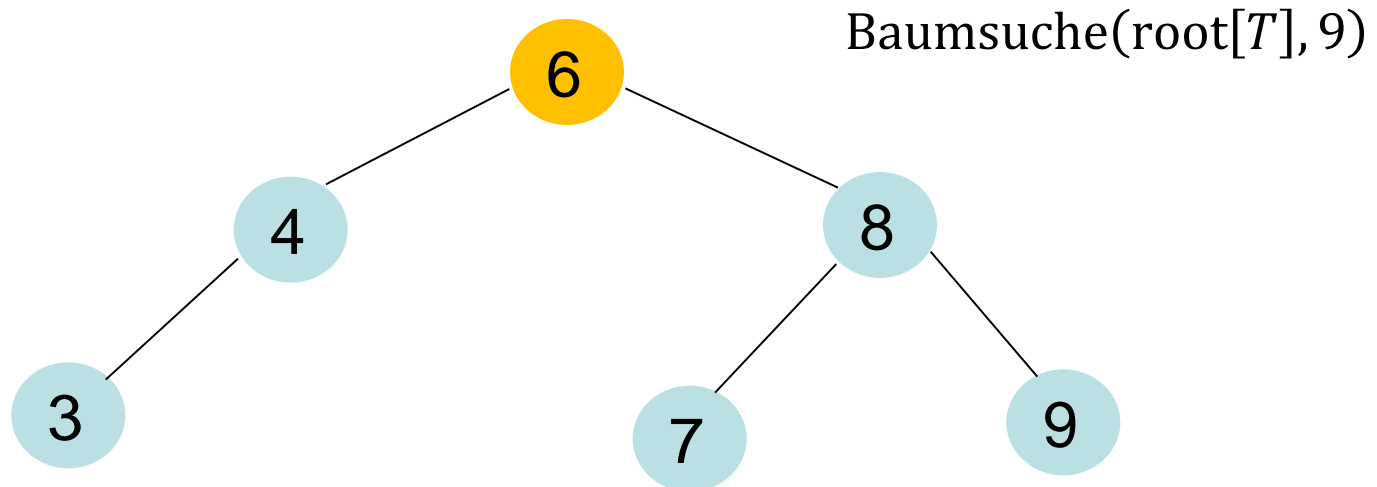
1. **if  $x = \text{nil}$  or  $k = \text{key}[x]$  then return  $x$**
2. **if  $k < \text{key}[x]$  then return Baumsuche(lc[ $x$ ],  $k$ )**
3. **else return Baumsuche(rc[ $x$ ],  $k$ )**



## Datenstrukturen

Baumsuche( $x, k$ )

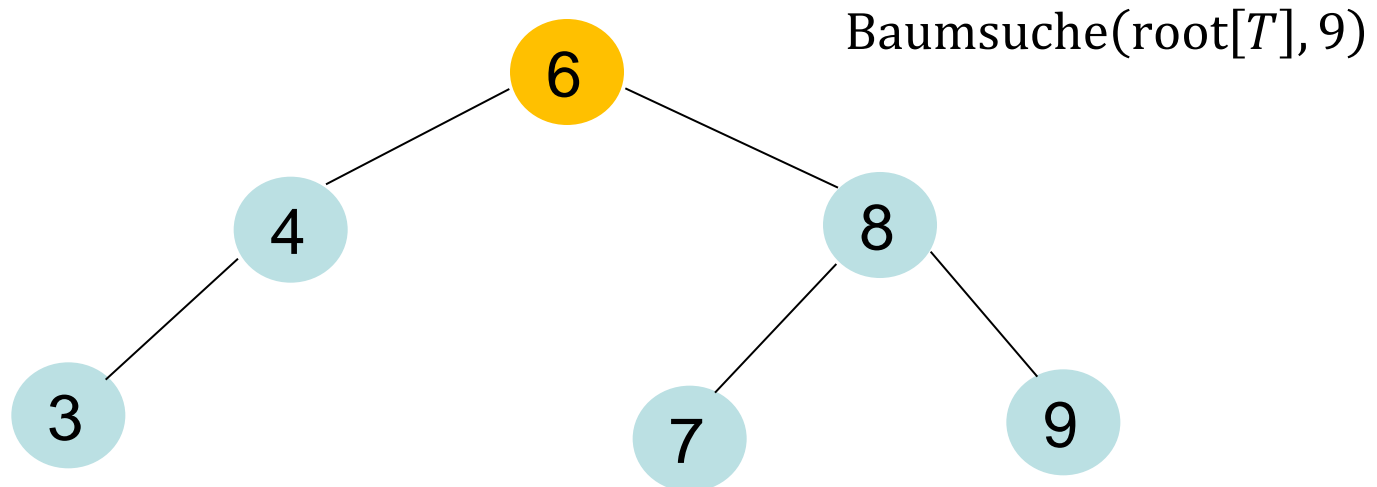
1. **if**  $x = \text{nil}$  **or**  $k = \text{key}[x]$  **then return**  $x$
2. **if**  $k < \text{key}[x]$  **then return** Baumsuche(lc[ $x$ ],  $k$ )
3. **else return** Baumsuche(rc[ $x$ ],  $k$ )



## Datenstrukturen

Baumsuche( $x, k$ )

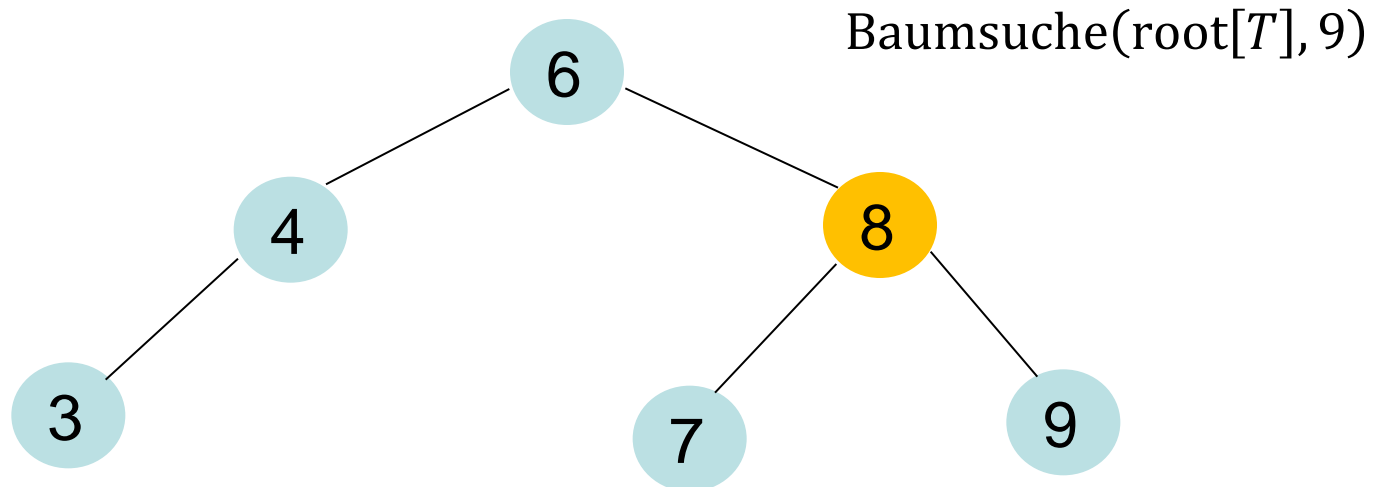
1. **if**  $x = \text{nil}$  **or**  $k = \text{key}[x]$  **then return**  $x$
2. **if**  $k < \text{key}[x]$  **then return** Baumsuche(lc[ $x$ ],  $k$ )
3. **else return** Baumsuche(rc[ $x$ ],  $k$ )



## Datenstrukturen

Baumsuche( $x, k$ )

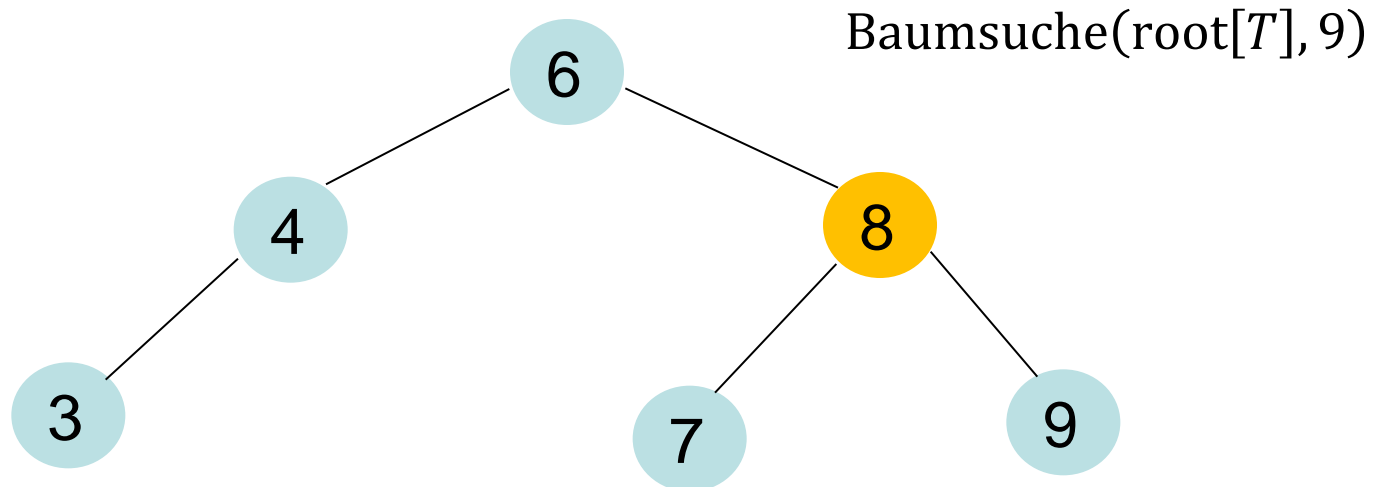
1. **if  $x = \text{nil}$  or  $k = \text{key}[x]$  then return  $x$**
2. **if  $k < \text{key}[x]$  then return Baumsuche(lc[ $x$ ],  $k$ )**
3. **else return Baumsuche(rc[ $x$ ],  $k$ )**



## Datenstrukturen

Baumsuche( $x, k$ )

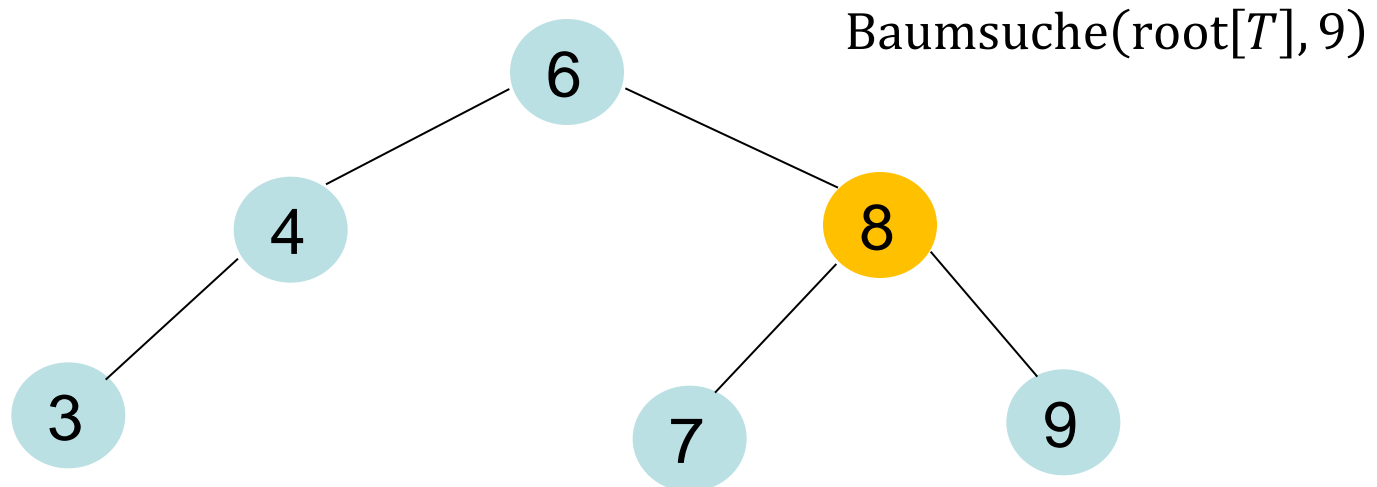
1. **if**  $x = \text{nil}$  **or**  $k = \text{key}[x]$  **then return**  $x$
2. **if**  $k < \text{key}[x]$  **then return** Baumsuche(lc[ $x$ ],  $k$ )
3. **else return** Baumsuche(rc[ $x$ ],  $k$ )



## Datenstrukturen

Baumsuche( $x, k$ )

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



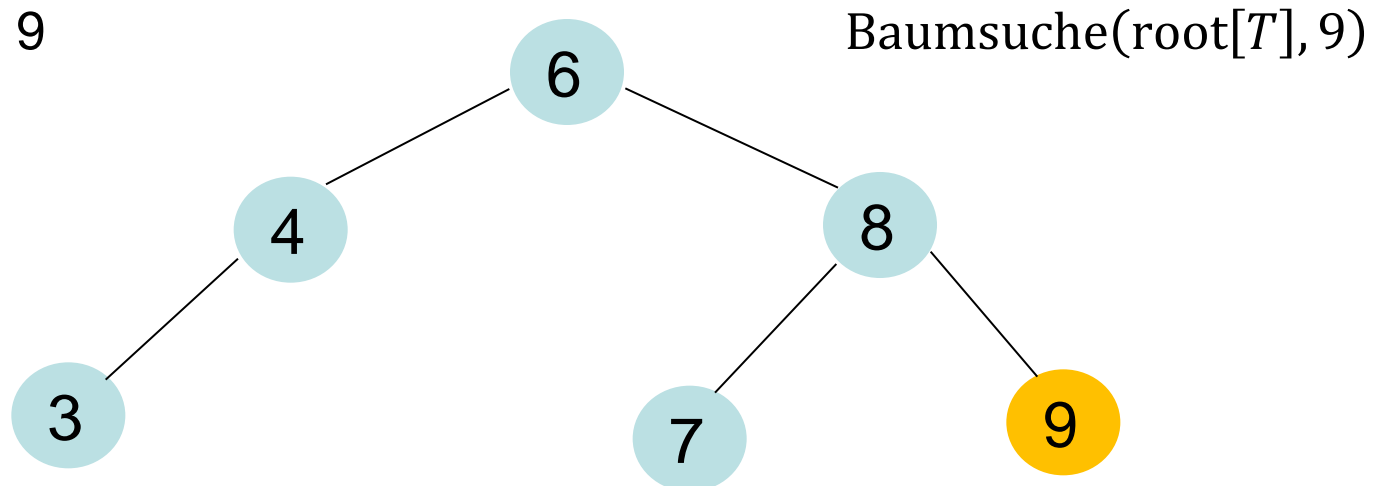


## Datenstrukturen

Baumsuche( $x, k$ )

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

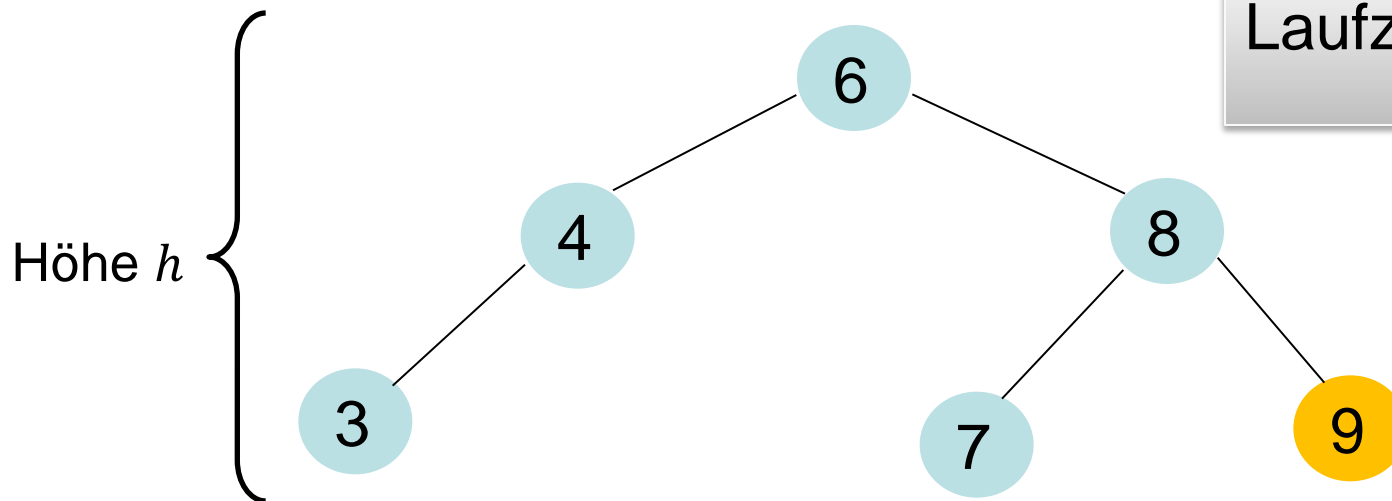
Ausgabe: 9



## Datenstrukturen

Baumsuche( $x, k$ )

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

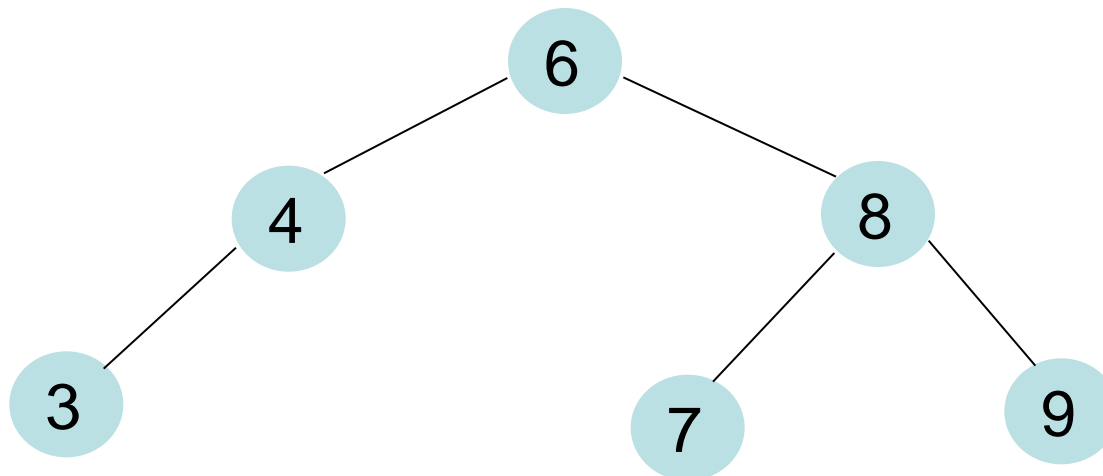


Laufzeit:  $O(h)$

## Datenstrukturen

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**  $x$

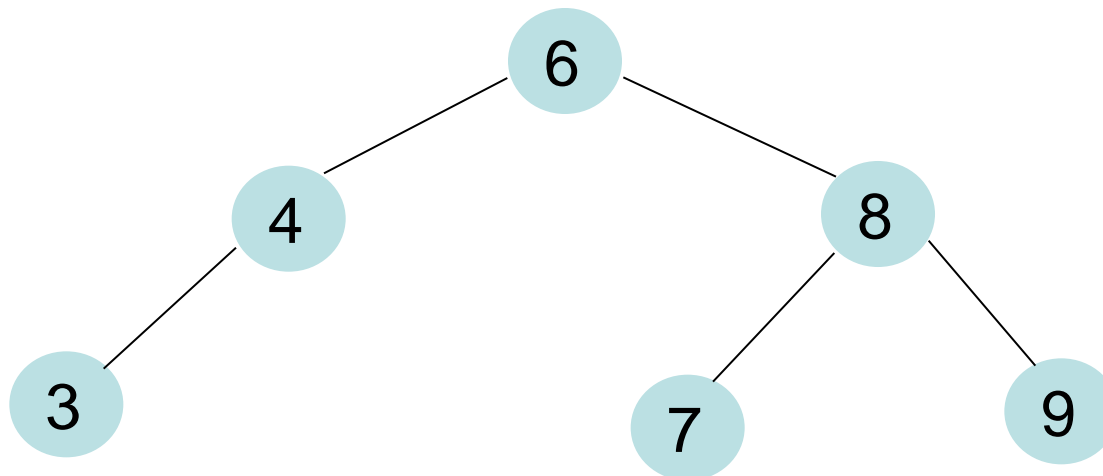


## Datenstrukturen

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

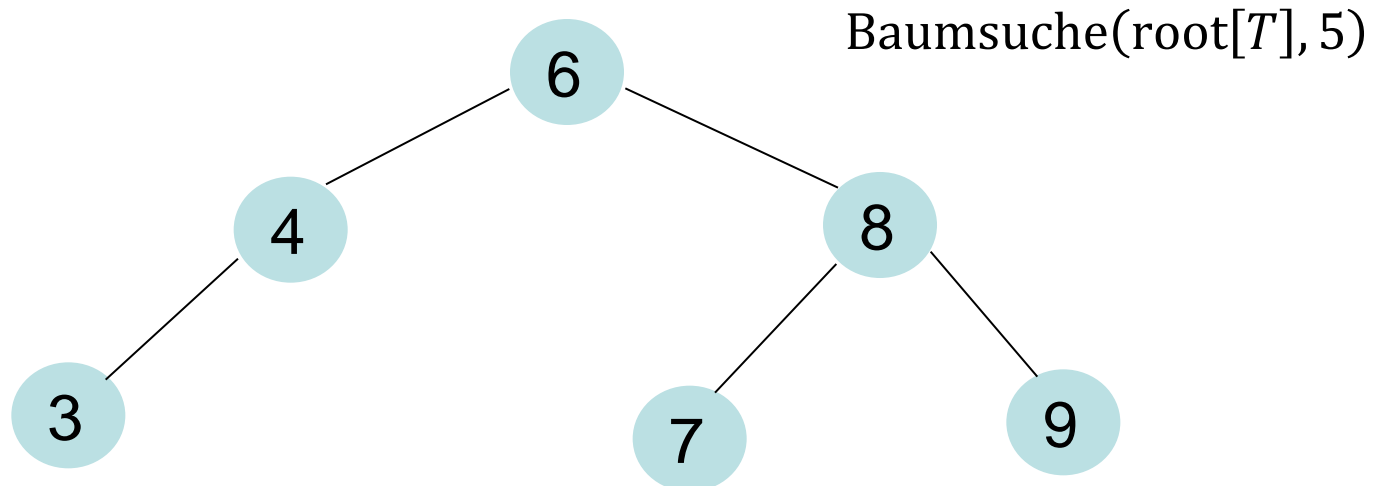


## Datenstrukturen

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**  $x$

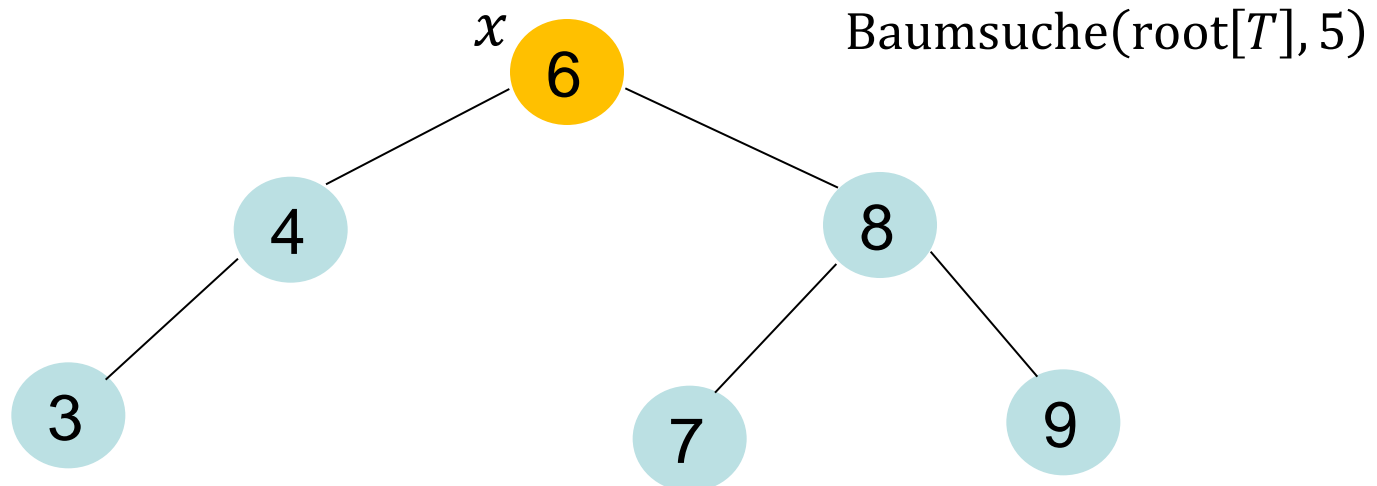
Aufruf mit  
 $x = \text{root}[T]$



## Datenstrukturen

IterativeBaumsuche( $x, k$ )

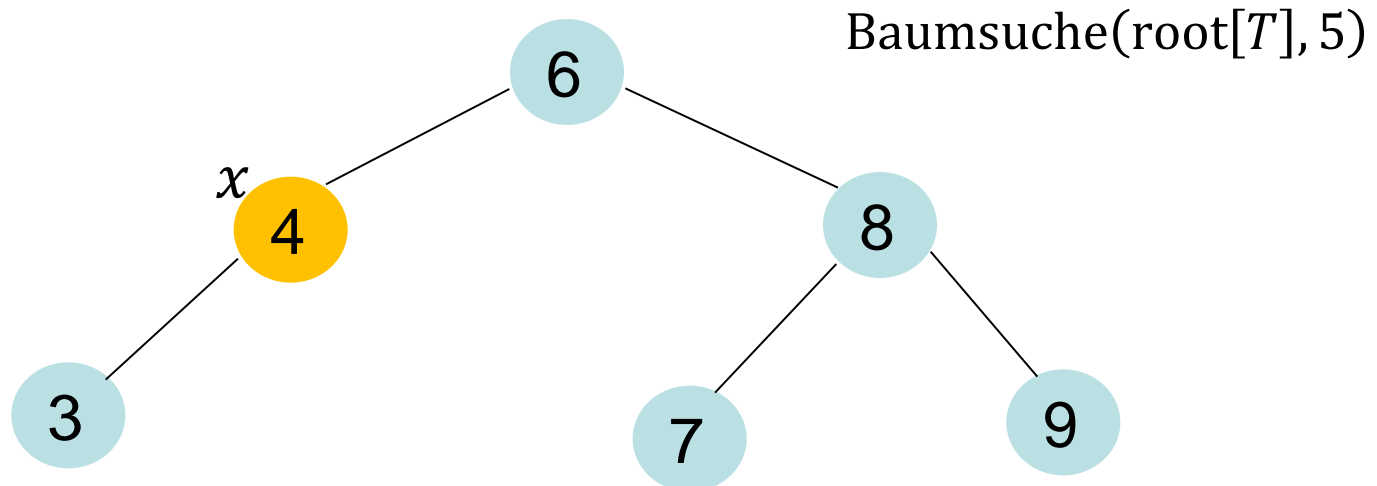
1. **while**  $x \neq \text{nil}$  **and**  $k \neq \text{key}[x]$  **do**
2.     **if**  $k < \text{key}[x]$  **then**  $x \leftarrow \text{lc}[x]$
3.     **else**  $x \leftarrow \text{rc}[x]$
4. **return**  $x$



## Datenstrukturen

IterativeBaumsuche( $x, k$ )

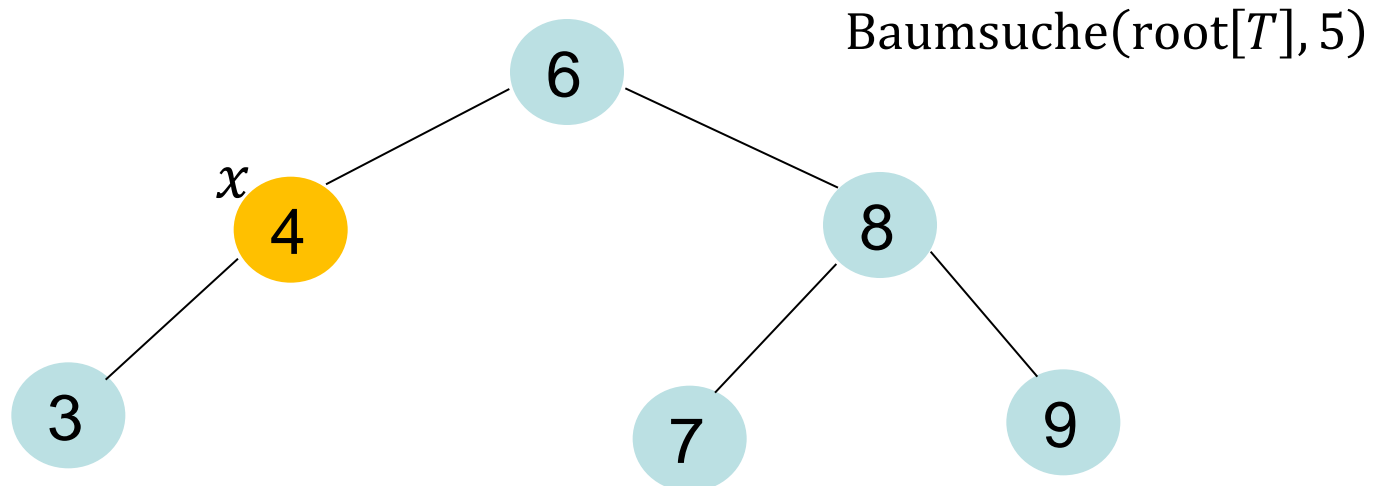
1. **while**  $x \neq \text{nil}$  **and**  $k \neq \text{key}[x]$  **do**
2.     **if**  $k < \text{key}[x]$  **then**  $x \leftarrow \text{lc}[x]$
3.     **else**  $x \leftarrow \text{rc}[x]$
4. **return**  $x$



## Datenstrukturen

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**  $x$



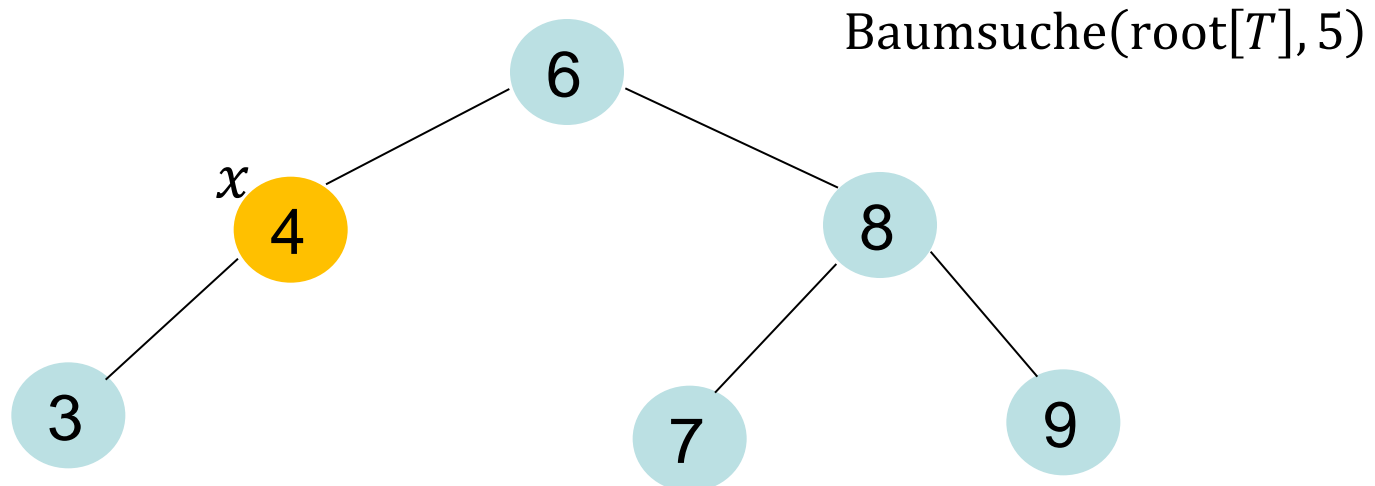


## Datenstrukturen

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**  $x$

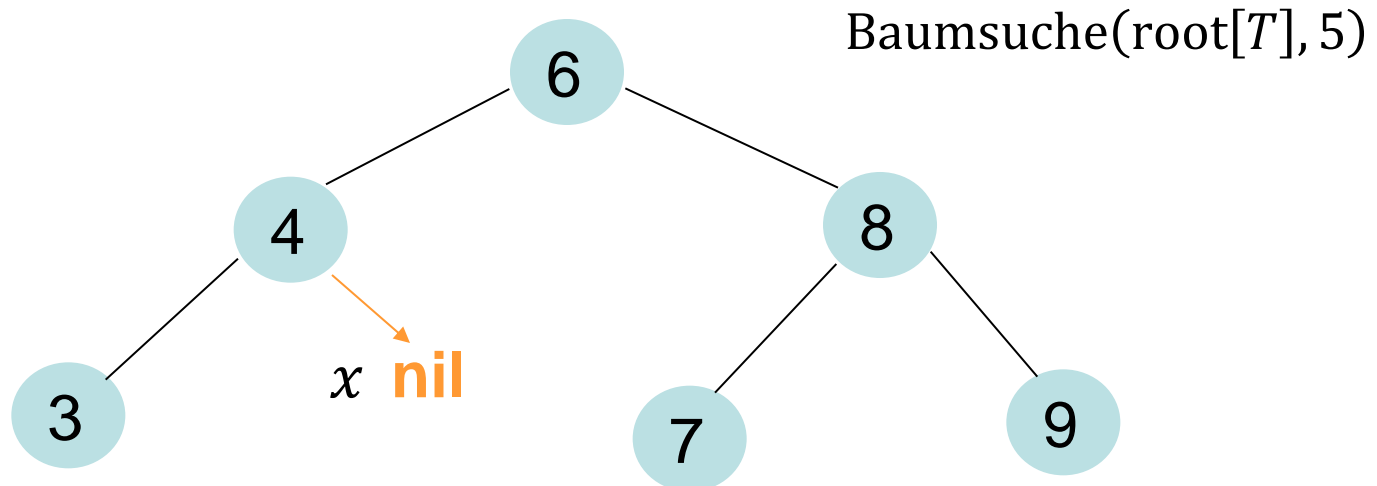
Aufruf mit  
 $x = \text{root}[T]$



## Datenstrukturen

IterativeBaumsuche( $x, k$ )

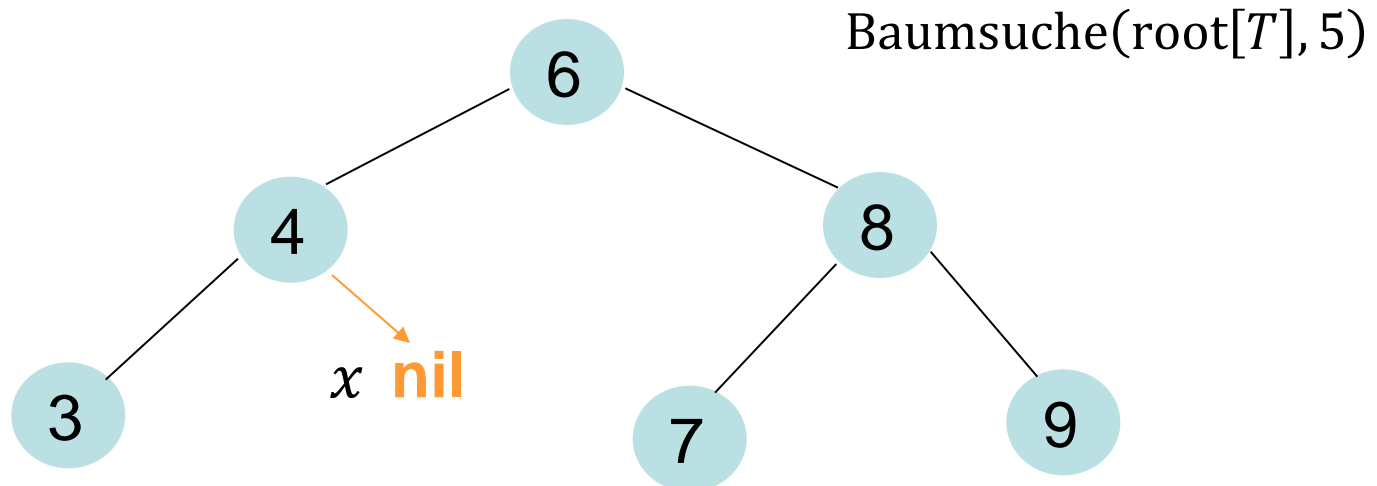
1. **while**  $x \neq \text{nil}$  **and**  $k \neq \text{key}[x]$  **do**
2.     **if**  $k < \text{key}[x]$  **then**  $x \leftarrow \text{lc}[x]$
3.     **else**  $x \leftarrow \text{rc}[x]$
4. **return**  $x$



## Datenstrukturen

IterativeBaumsuche( $x, k$ )

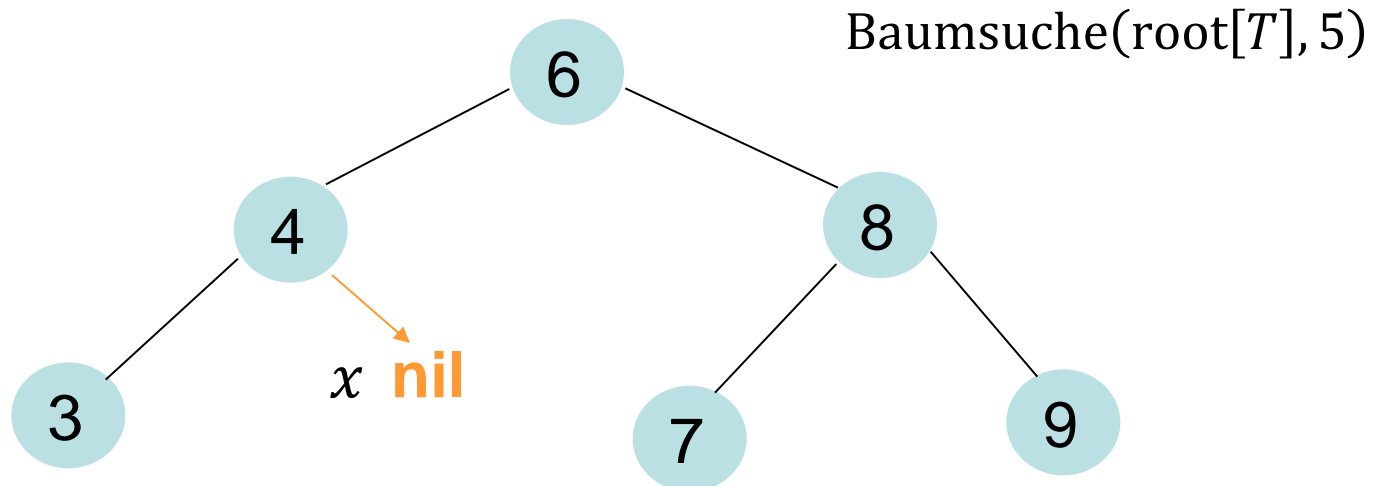
1. **while**  $x \neq \text{nil}$  **and**  $k \neq \text{key}[x]$  **do**
2.     **if**  $k < \text{key}[x]$  **then**  $x \leftarrow \text{lc}[x]$
3.     **else**  $x \leftarrow \text{rc}[x]$
4. **return**  $x$



## Datenstrukturen

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**  $x$

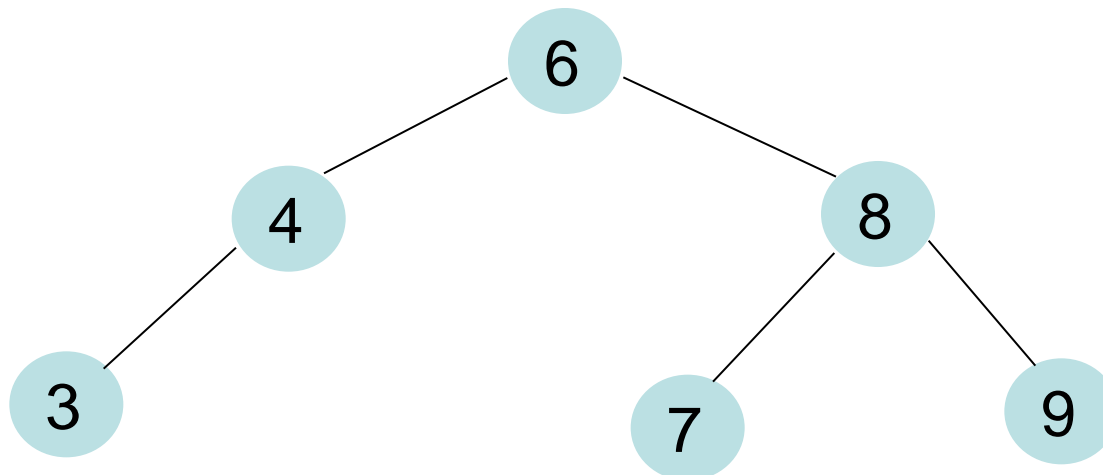


## Datenstrukturen

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  $O(h)$ .



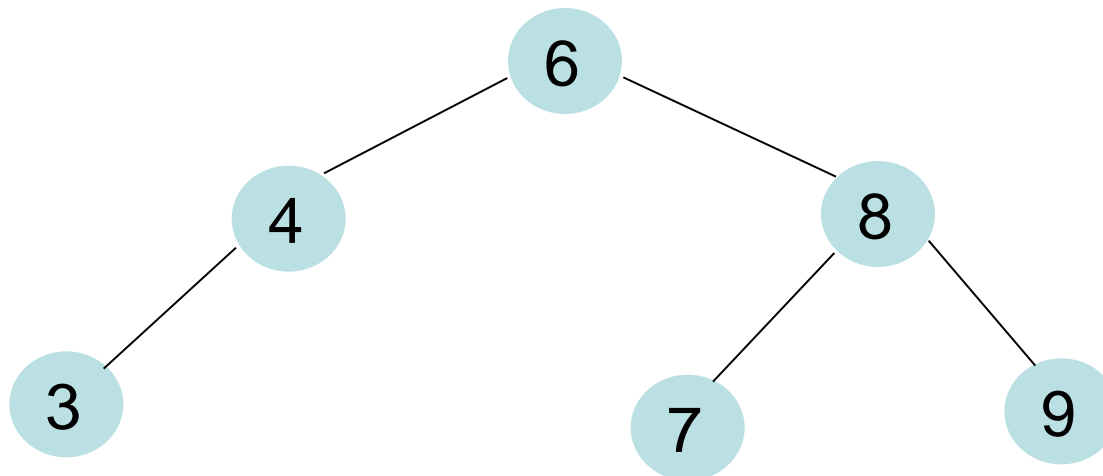
## Datenstrukturen

### *Minimum- und Maximumsuche*

Suchbaumeigenschaft:

Alle Knoten im rechten Unterbaum eines Knotens  $x$  sind größer als  $\text{key}[x]$

Alle Knoten im linken Unterbaum von  $x$  sind kleiner als  $\text{key}[x]$

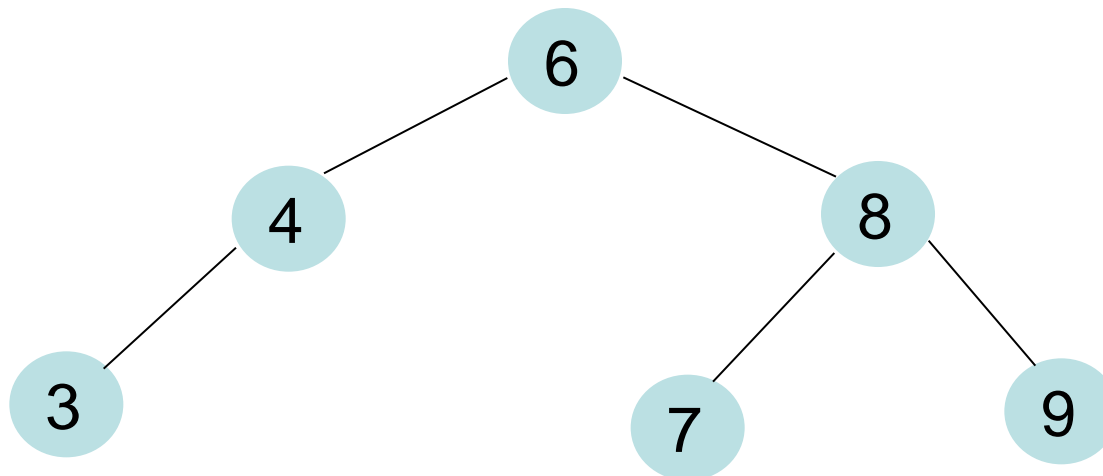


## Datenstrukturen

Wird mit Wurzel  
aufgerufen

MinimumSuche( $x$ )

1. **while**  $lc[x] \neq \text{nil}$  **do**  $x \leftarrow lc[x]$
2. **return**  $x$



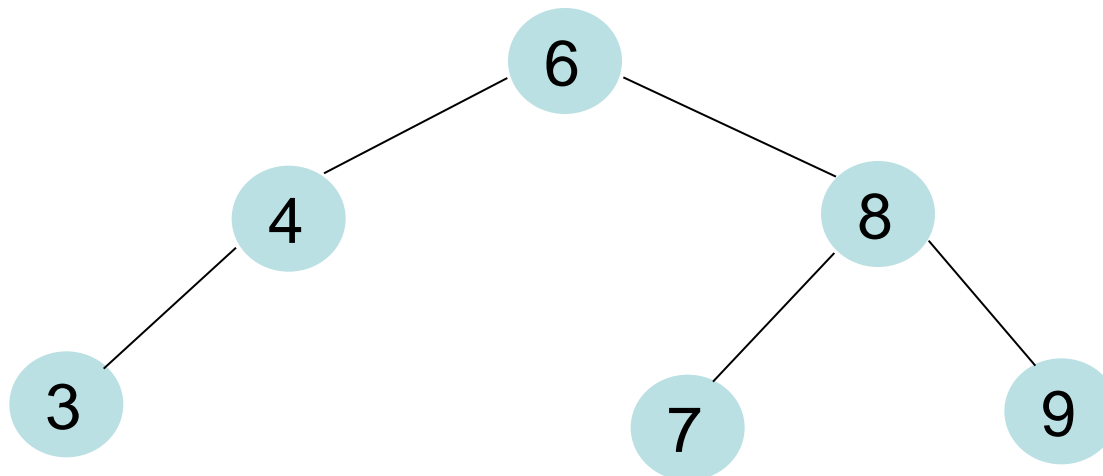
## Datenstrukturen

Wird mit Wurzel  
aufgerufen

Laufzeit  $O(h)$

MinimumSuche( $x$ )

1. **while**  $lc[x] \neq \text{nil}$  **do**  $x \leftarrow lc[x]$
2. **return**  $x$

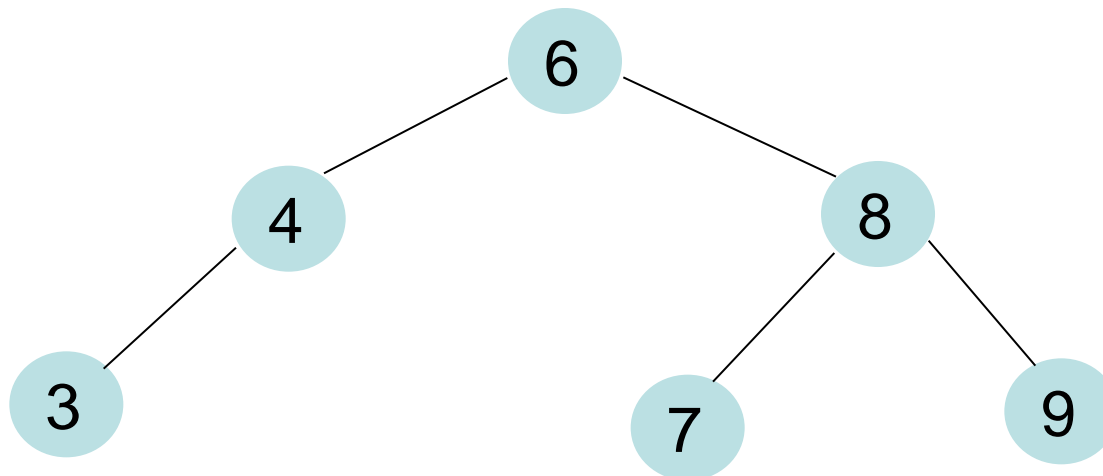




## Datenstrukturen

### *Nachfolgersuche*

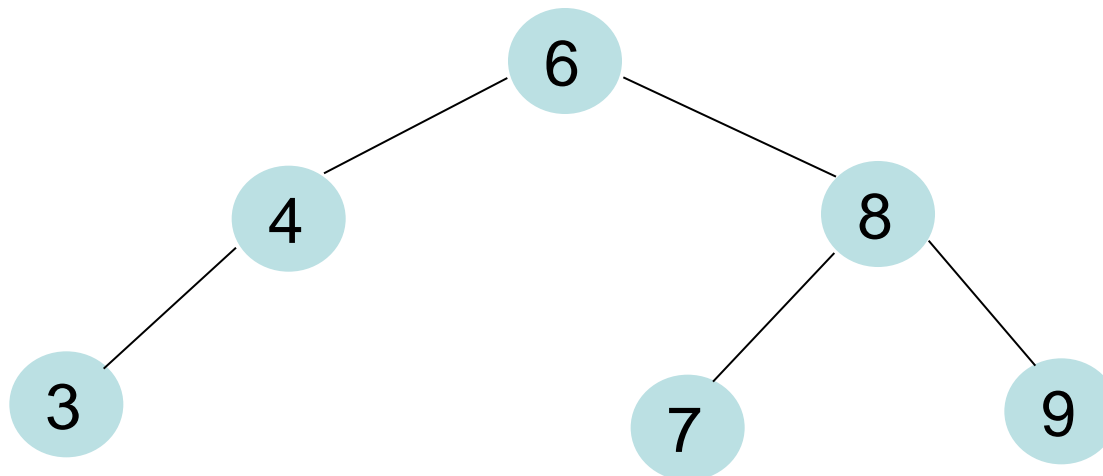
- Nachfolger bzgl. Inorder-Tree-Walk
- Wenn alle Schlüssel unterschiedlich, dann ist das der nächstgrößere Schlüssel



## Datenstrukturen

### *Nachfolgersuche*

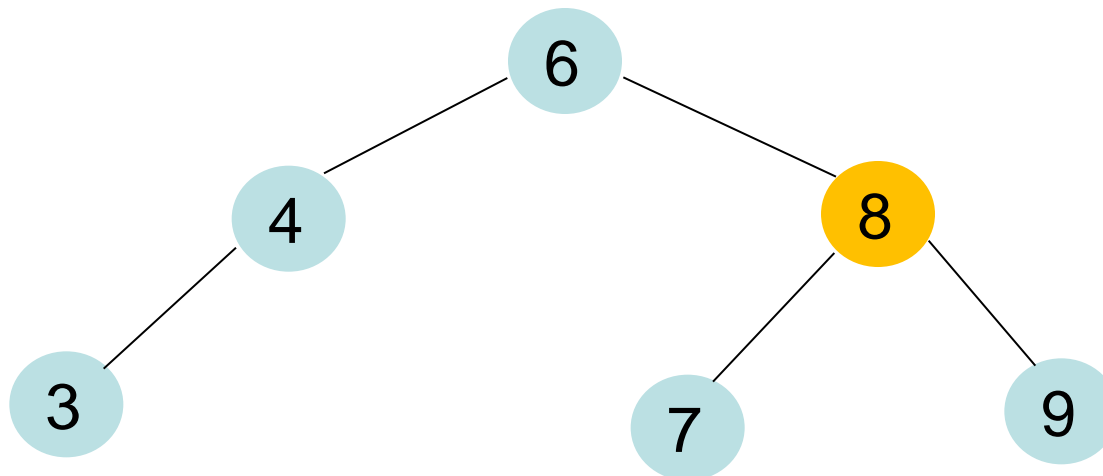
- Fall 1 (rechter Unterbaum von  $x$  nicht leer):  
Dann ist der linkeste Knoten im rechten Unterbaum der Nachfolger von  $x$



## Datenstrukturen

### *Nachfolgersuche*

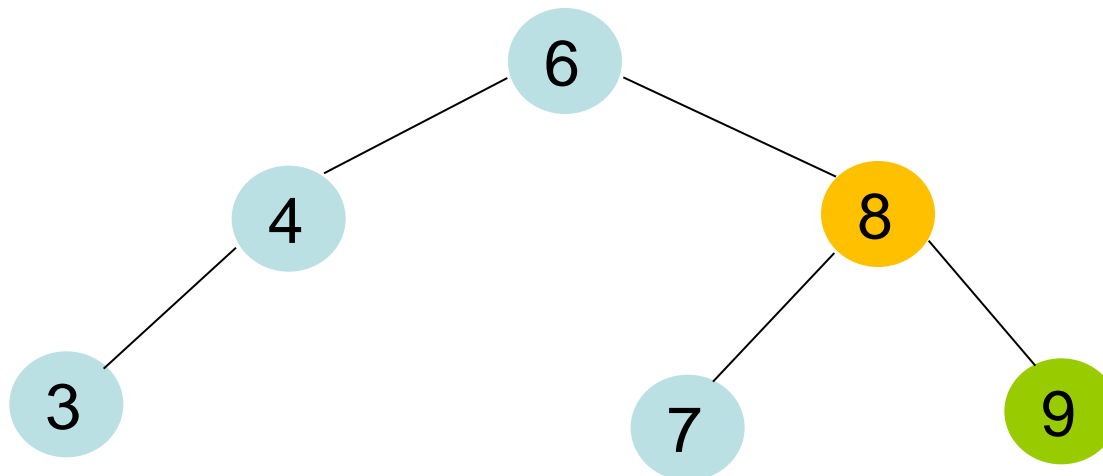
- Fall 1 (rechter Unterbaum von  $x$  nicht leer):  
Dann ist der linkeste Knoten im rechten Unterbaum der Nachfolger von  $x$



## Datenstrukturen

### *Nachfolgersuche*

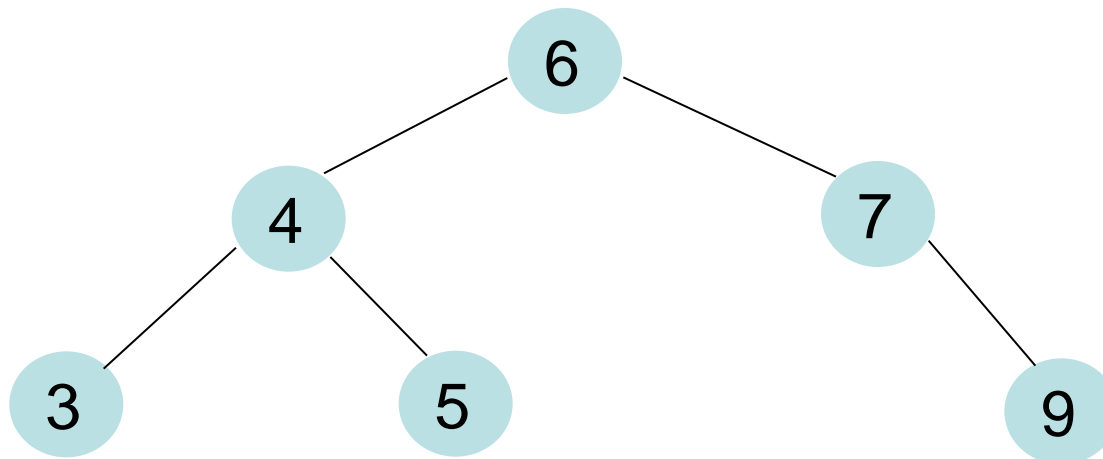
- Fall 1 (rechter Unterbaum von  $x$  nicht leer):  
Dann ist der linkeste Knoten im rechten Unterbaum der Nachfolger von  $x$



## Datenstrukturen

### *Nachfolgersuche*

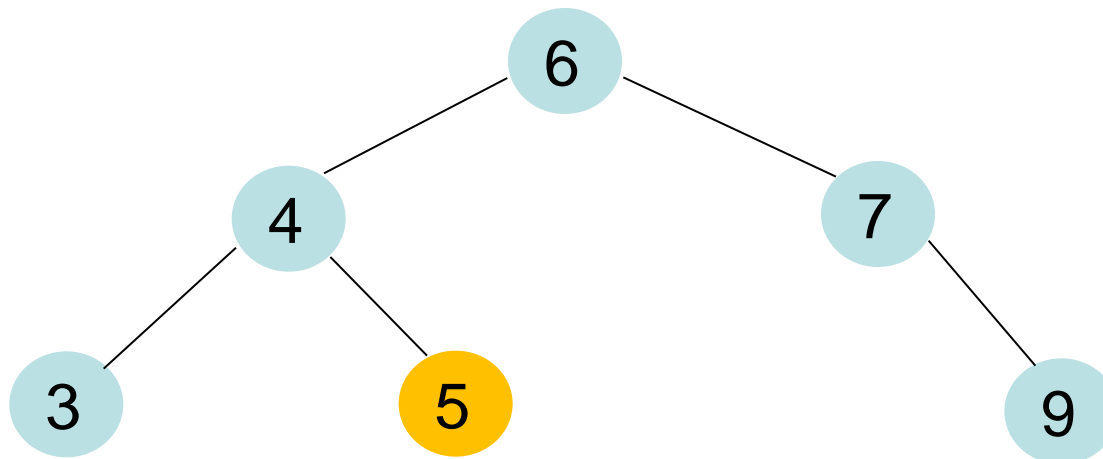
- Fall 2 (rechter Unterbaum von  $x$  leer und  $x$  hat Nachfolger  $y$ ):  
Dann ist  $y$  der erste Knoten auf dem Pfad zur Wurzel, der größer als  $x$  ist



## Datenstrukturen

### *Nachfolgersuche*

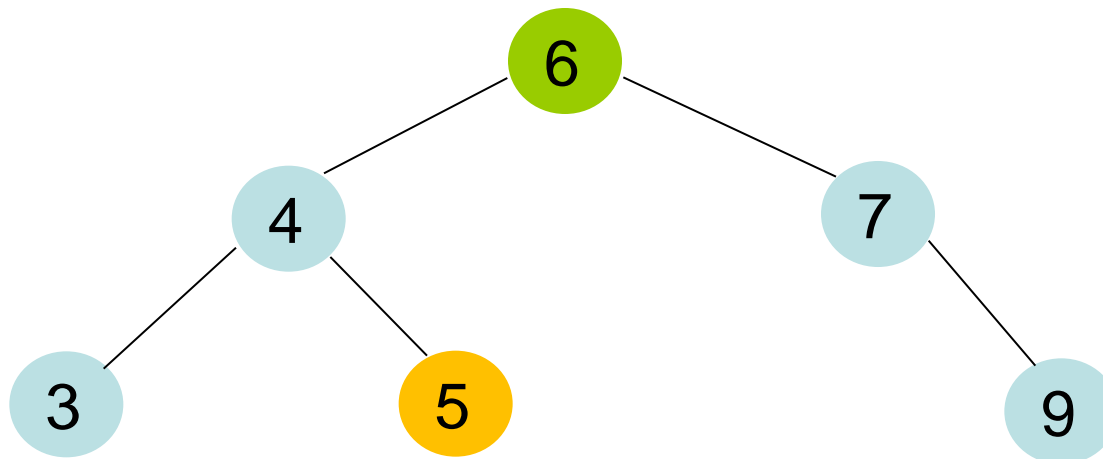
- Fall 2 (rechter Unterbaum von  $x$  leer und  $x$  hat Nachfolger  $y$ ):  
Dann ist  $y$  der erste Knoten auf dem Pfad zur Wurzel, der größer als  $x$  ist



## Datenstrukturen

### *Nachfolgersuche*

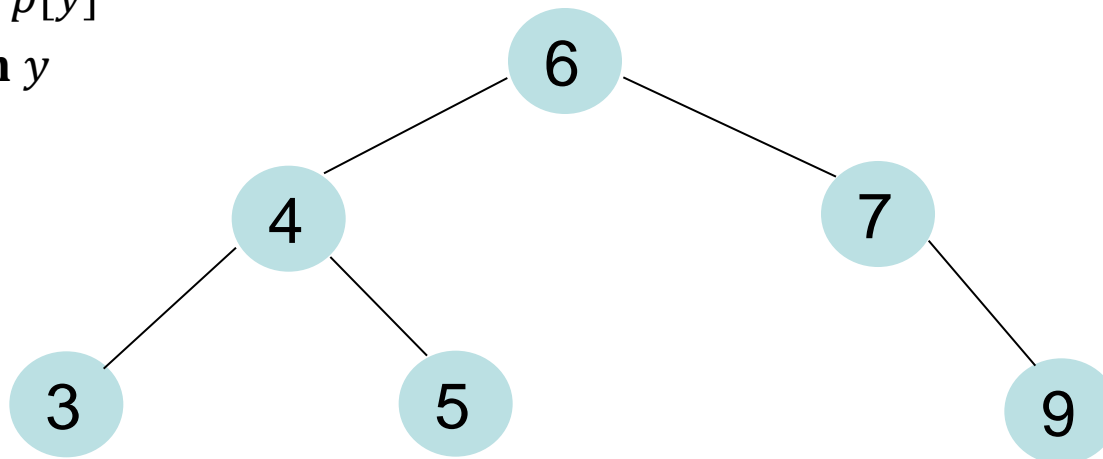
- Fall 2 (rechter Unterbaum von  $x$  leer und  $x$  hat Nachfolger  $y$ ):  
Dann ist  $y$  der erste Knoten auf dem Pfad zur Wurzel, der größer als  $x$  ist



## Datenstrukturen

Nachfolgersuche( $x$ )

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

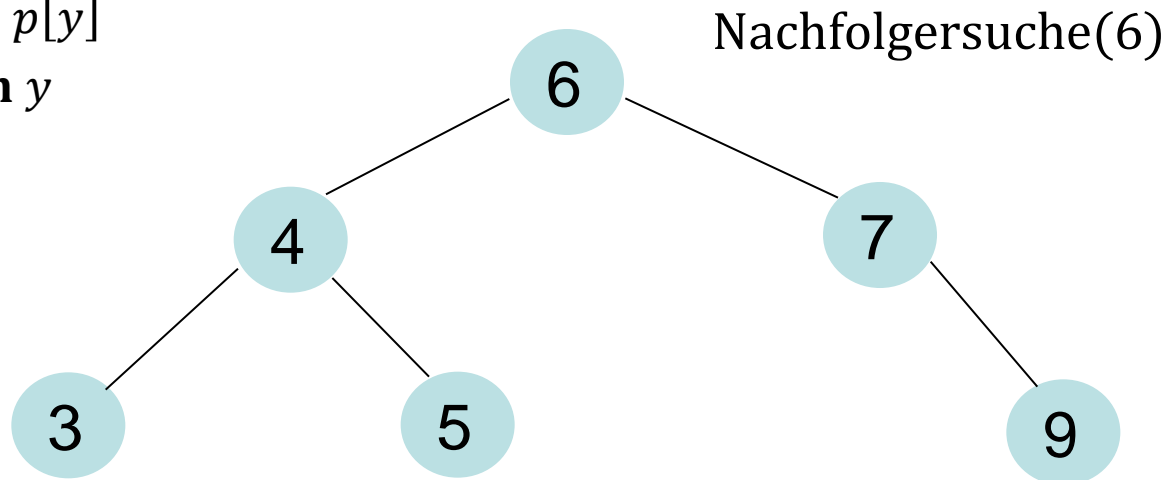




## Datenstrukturen

Nachfolgersuche( $x$ )

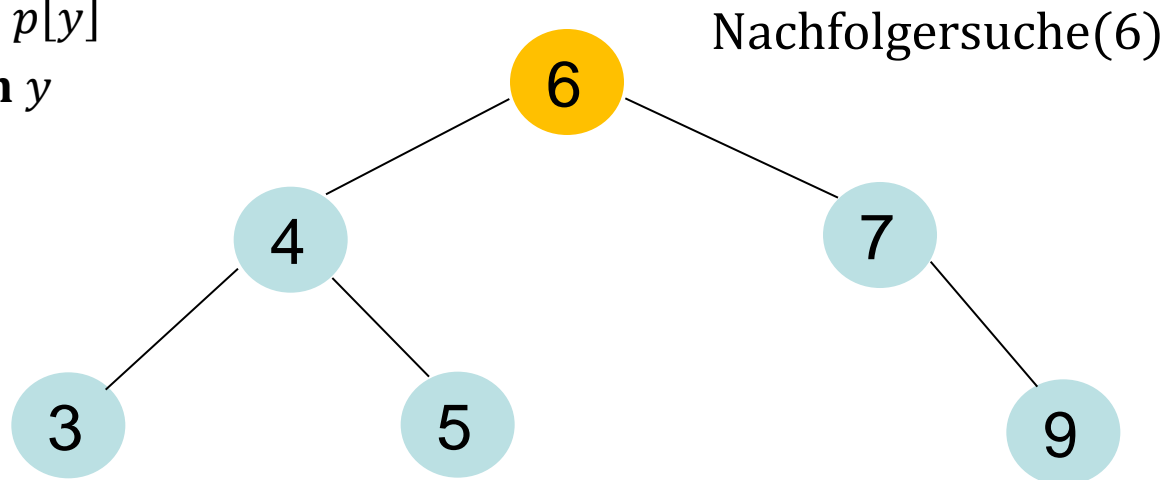
1. **if**  $rc[x] \neq \text{nil}$  **then return** MinimumSuche( $rc[x]$ )
2.  $y \leftarrow p[x]$
3. **while**  $y \neq \text{nil}$  **and**  $x = rc[y]$  **do**
4.      $x \leftarrow y$
5.      $y \leftarrow p[y]$
6. **return**  $y$



## Datenstrukturen

Nachfolgersuche( $x$ )

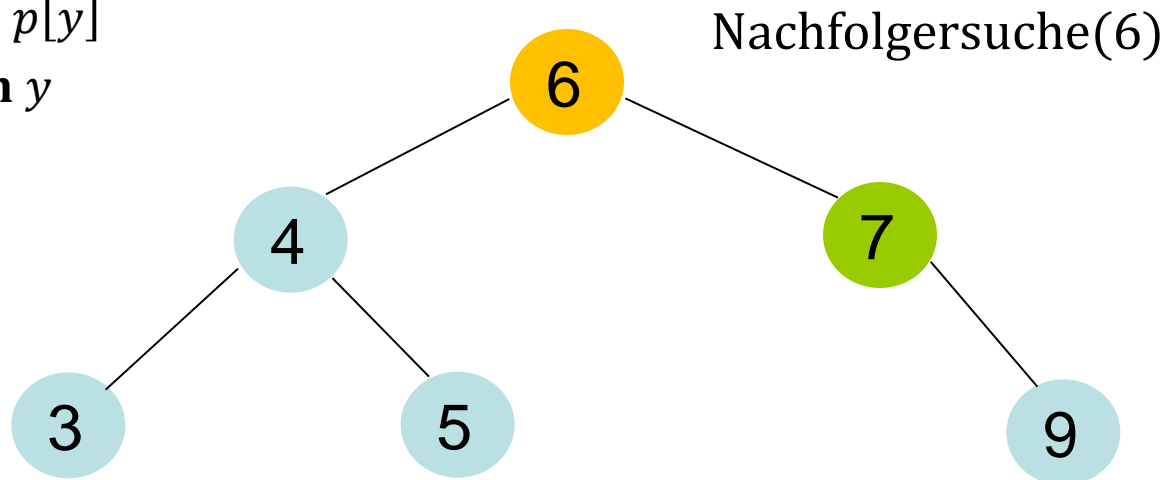
1. **if**  $rc[x] \neq \text{nil}$  **then return** MinimumSuche( $rc[x]$ )
2.  $y \leftarrow p[x]$
3. **while**  $y \neq \text{nil}$  **and**  $x = rc[y]$  **do**
4.      $x \leftarrow y$
5.      $y \leftarrow p[y]$
6. **return**  $y$



## Datenstrukturen

Nachfolgersuche( $x$ )

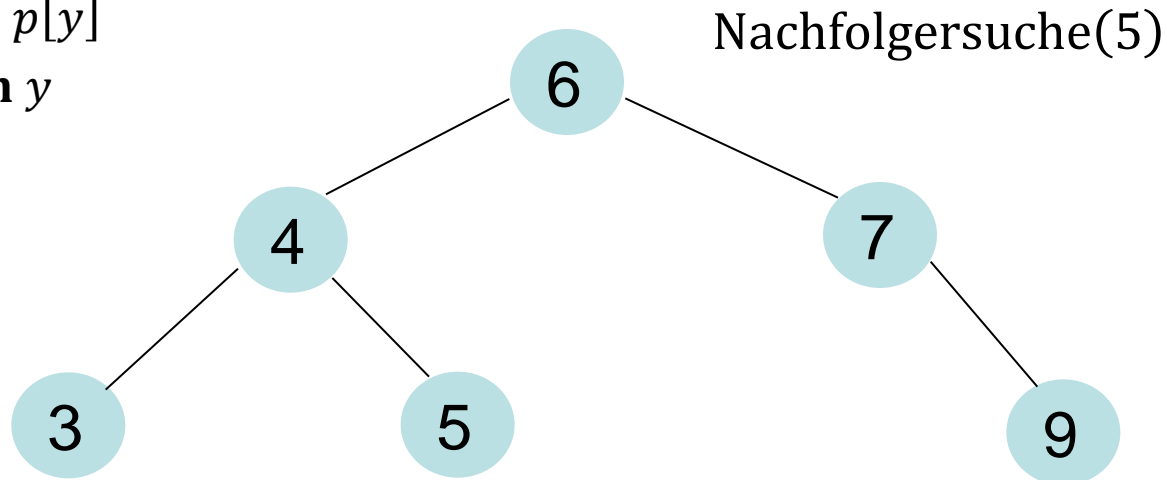
1. **if**  $rc[x] \neq \text{nil}$  **then return** MinimumSuche( $rc[x]$ )
2.  $y \leftarrow p[x]$
3. **while**  $y \neq \text{nil}$  **and**  $x = rc[y]$  **do**
4.      $x \leftarrow y$
5.      $y \leftarrow p[y]$
6. **return**  $y$



## Datenstrukturen

Nachfolgersuche( $x$ )

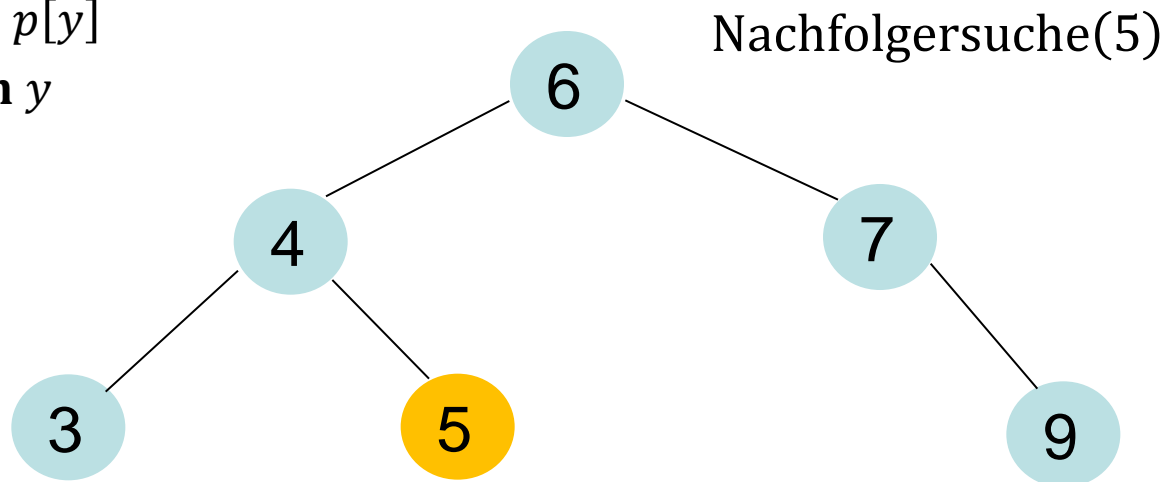
1. **if**  $rc[x] \neq \text{nil}$  **then return** MinimumSuche( $rc[x]$ )
2.  $y \leftarrow p[x]$
3. **while**  $y \neq \text{nil}$  **and**  $x = rc[y]$  **do**
4.      $x \leftarrow y$
5.      $y \leftarrow p[y]$
6. **return**  $y$



## Datenstrukturen

Nachfolgersuche( $x$ )

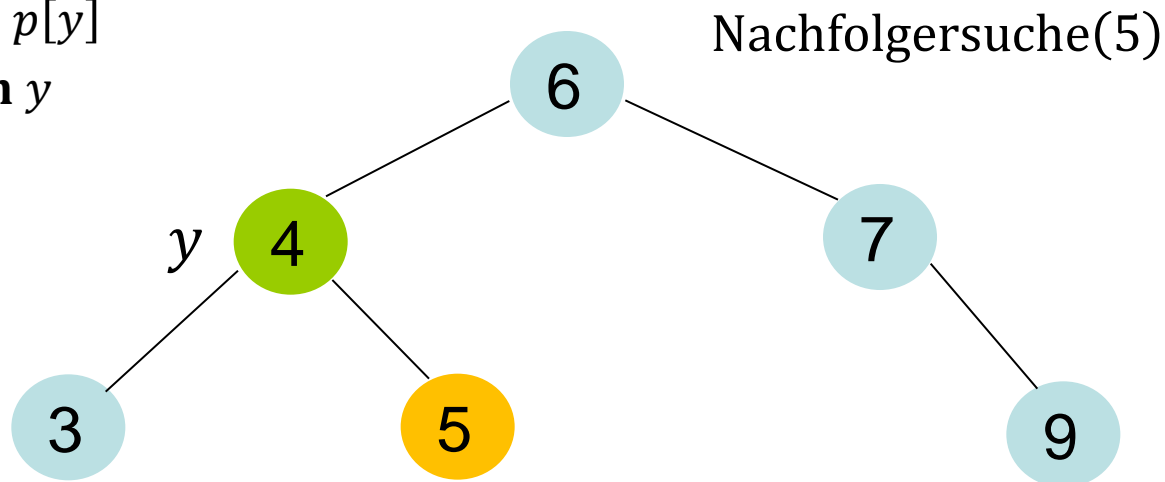
1. **if**  $rc[x] \neq \text{nil}$  **then return** MinimumSuche( $rc[x]$ )
2.  $y \leftarrow p[x]$
3. **while**  $y \neq \text{nil}$  **and**  $x = rc[y]$  **do**
4.      $x \leftarrow y$
5.      $y \leftarrow p[y]$
6. **return**  $y$



## Datenstrukturen

Nachfolgersuche( $x$ )

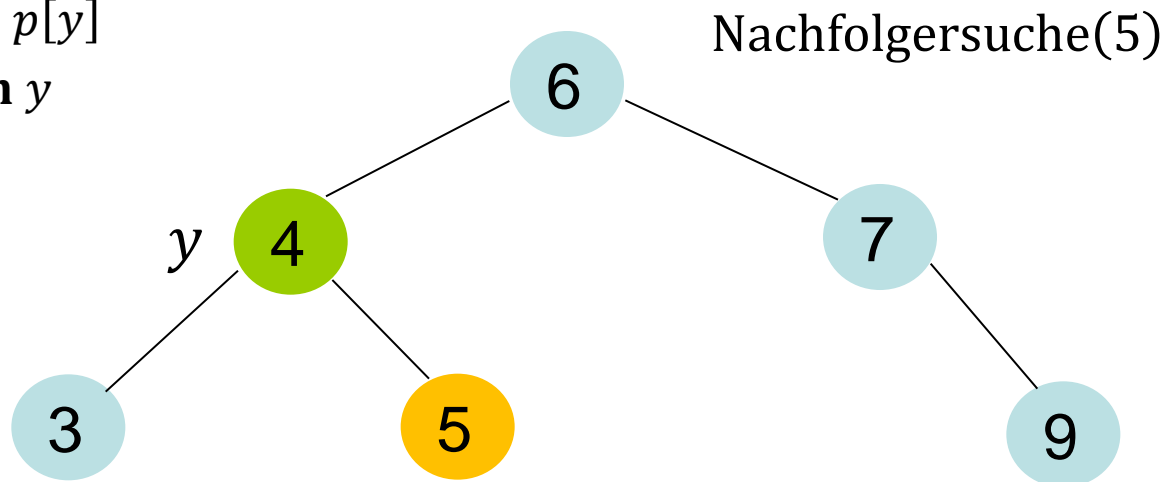
1. **if**  $rc[x] \neq \text{nil}$  **then return** MinimumSuche( $rc[x]$ )
2.  $y \leftarrow p[x]$
3. **while**  $y \neq \text{nil}$  **and**  $x = rc[y]$  **do**
4.      $x \leftarrow y$
5.      $y \leftarrow p[y]$
6. **return**  $y$



## Datenstrukturen

Nachfolgersuche( $x$ )

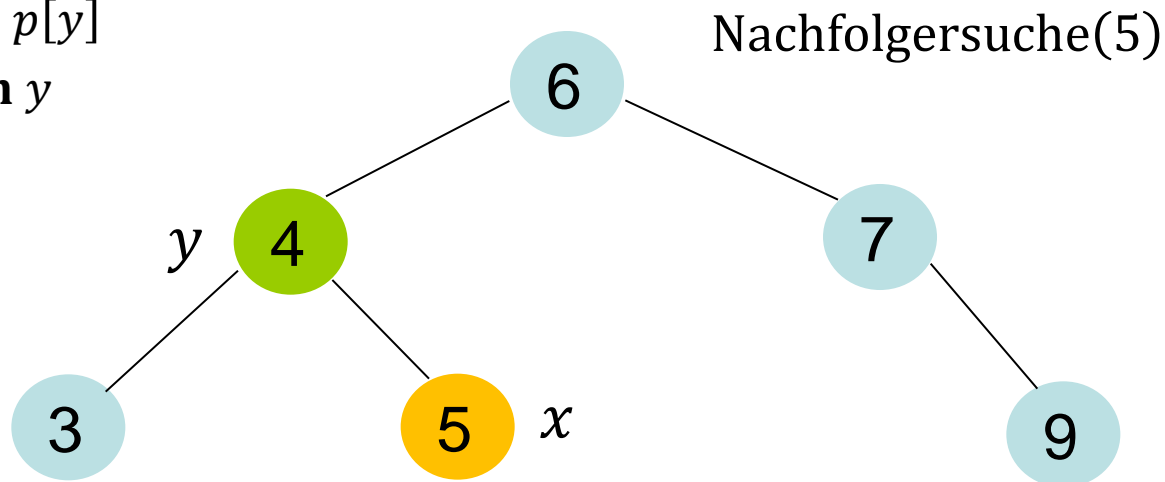
1. **if**  $rc[x] \neq \text{nil}$  **then return** MinimumSuche( $rc[x]$ )
2.  $y \leftarrow p[x]$
3. **while**  $y \neq \text{nil}$  **and**  $x = rc[y]$  **do**
4.      $x \leftarrow y$
5.      $y \leftarrow p[y]$
6. **return**  $y$



## Datenstrukturen

Nachfolgersuche( $x$ )

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

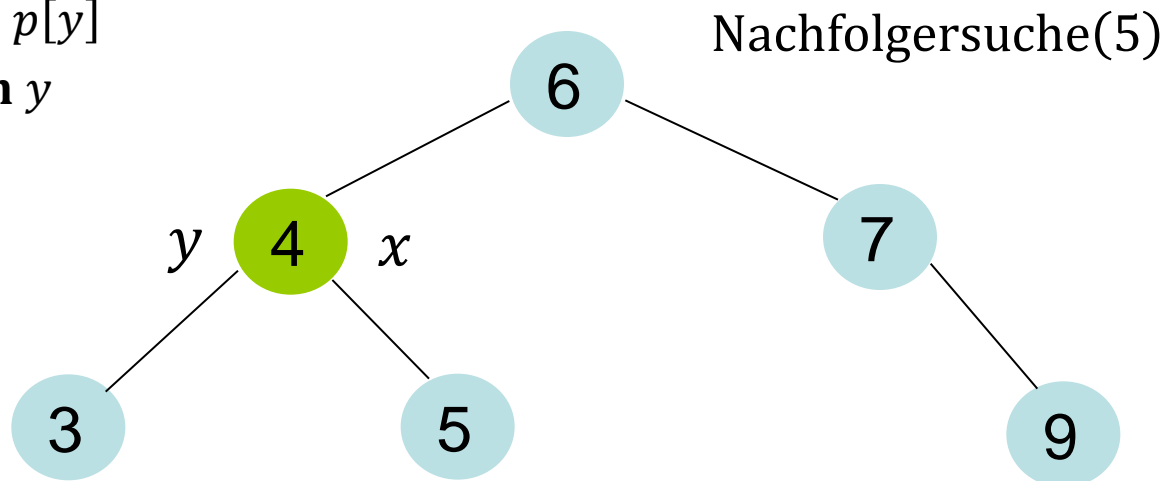




## Datenstrukturen

Nachfolgersuche( $x$ )

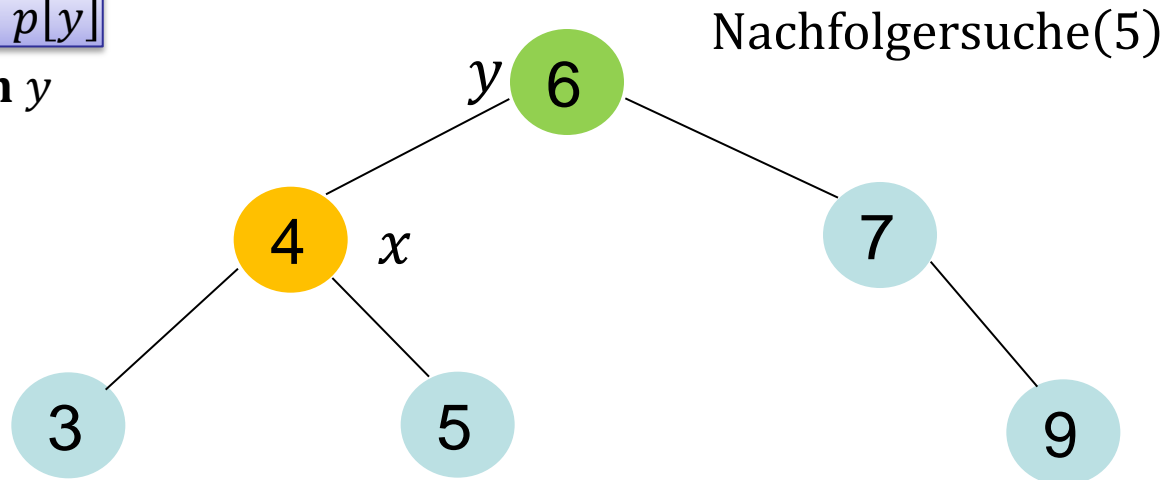
1. **if**  $rc[x] \neq \text{nil}$  **then return** MinimumSuche( $rc[x]$ )
2.  $y \leftarrow p[x]$
3. **while**  $y \neq \text{nil}$  **and**  $x = rc[y]$  **do**
4.  $x \leftarrow y$
5.  $y \leftarrow p[y]$
6. **return**  $y$



## Datenstrukturen

Nachfolgersuche( $x$ )

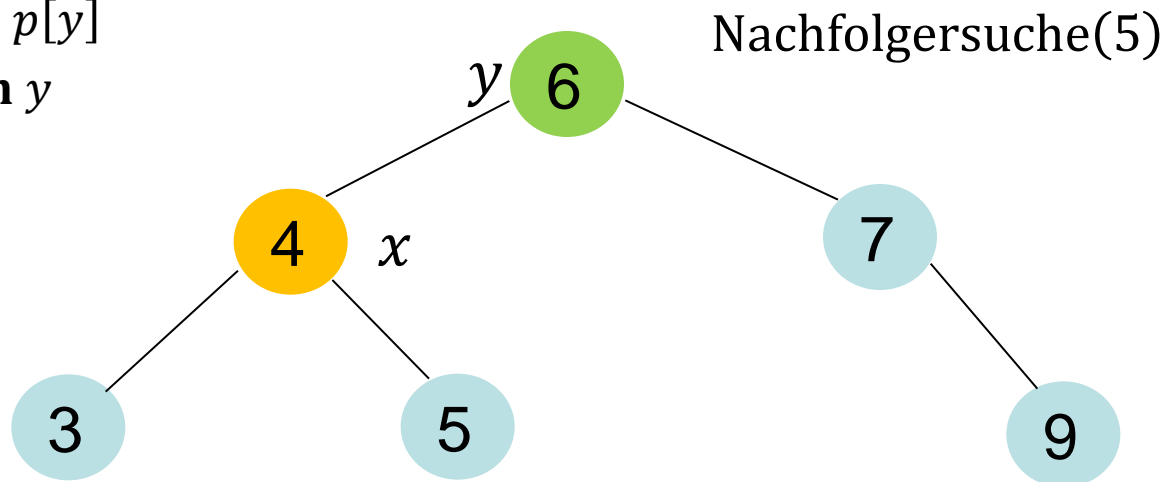
1. **if**  $rc[x] \neq \text{nil}$  **then return** MinimumSuche( $rc[x]$ )
2.  $y \leftarrow p[x]$
3. **while**  $y \neq \text{nil}$  **and**  $x = rc[y]$  **do**
4.      $x \leftarrow y$
5.      $y \leftarrow p[y]$
6. **return**  $y$



## Datenstrukturen

Nachfolgersuche( $x$ )

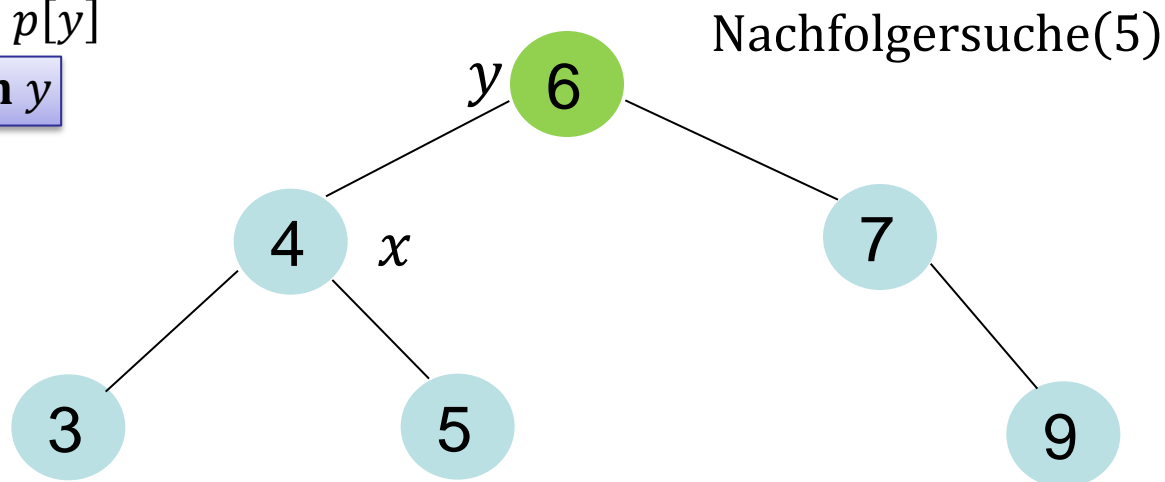
1. **if**  $rc[x] \neq \text{nil}$  **then return** MinimumSuche( $rc[x]$ )
2.  $y \leftarrow p[x]$
3. **while**  $y \neq \text{nil}$  **and**  $x = rc[y]$  **do**
4.      $x \leftarrow y$
5.      $y \leftarrow p[y]$
6. **return**  $y$



## Datenstrukturen

Nachfolgersuche( $x$ )

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

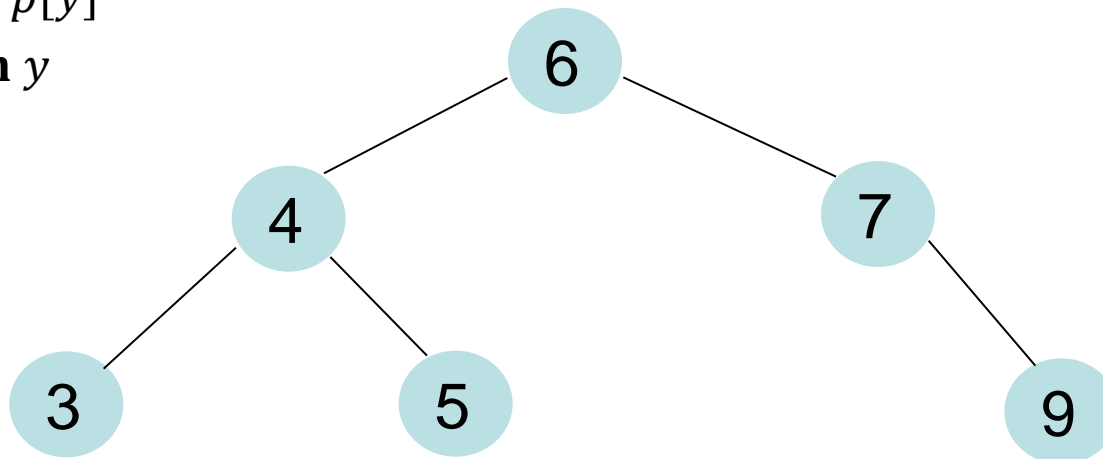


## Datenstrukturen

Nachfolgersuche( $x$ )

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

Laufzeit  $O(h)$

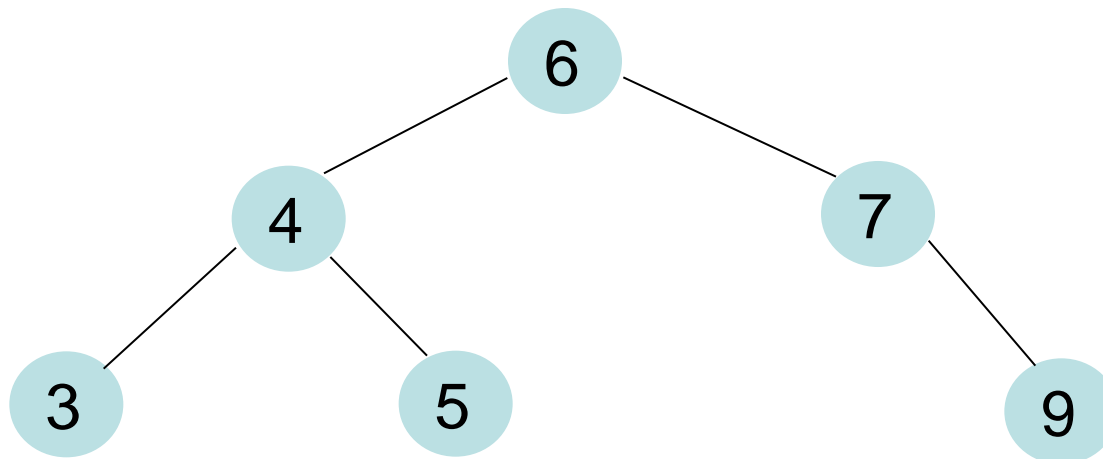


## Datenstrukturen

### *Vorgängersuche*

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

Laufzeit  $O(h)$



## Datenstrukturen

### *Binäre Suchbäume*

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

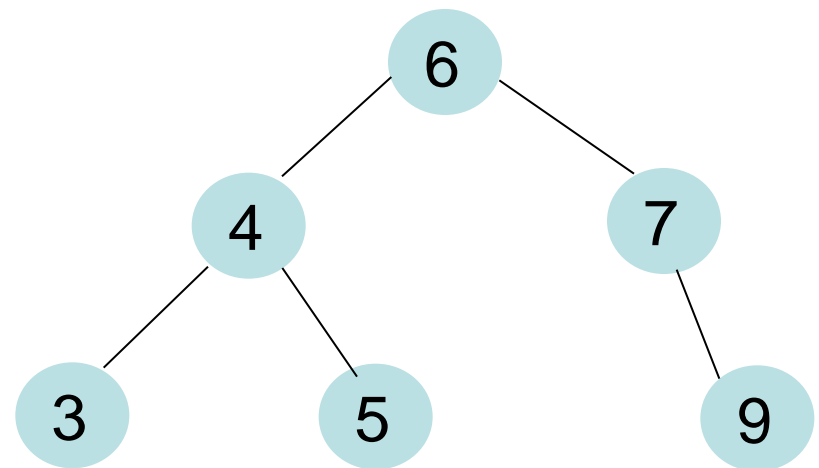
### *Dynamische Operationen?*

- 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

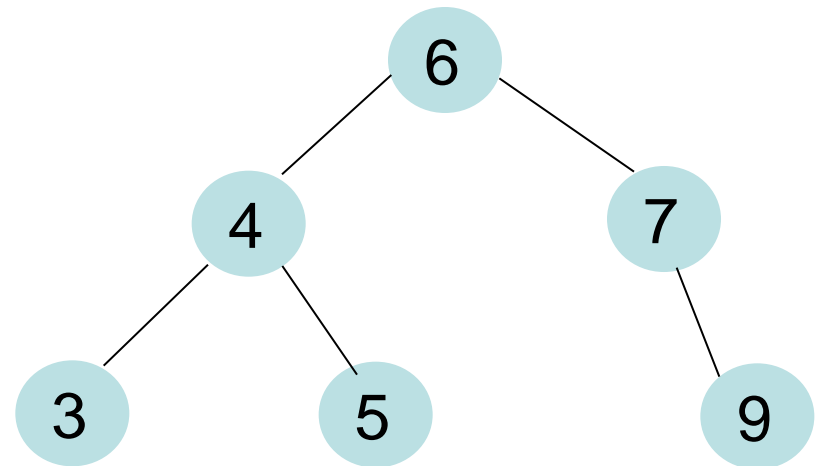




## Datenstrukturen

Einfügen( $T, z$ )

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



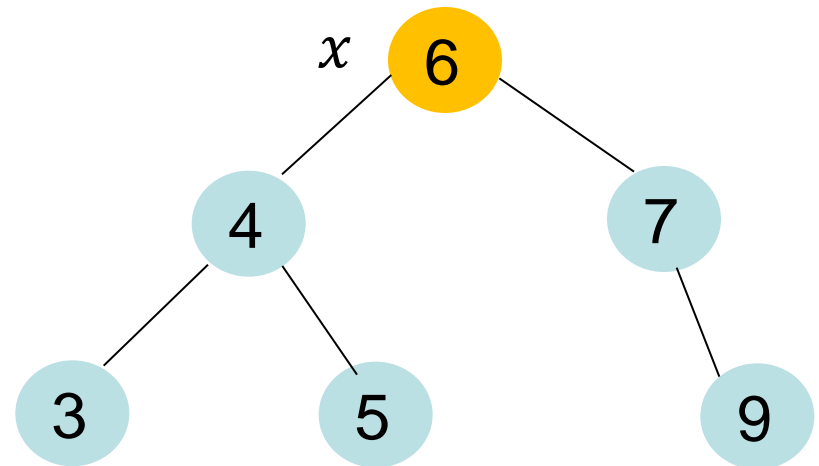
## Datenstrukturen

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)

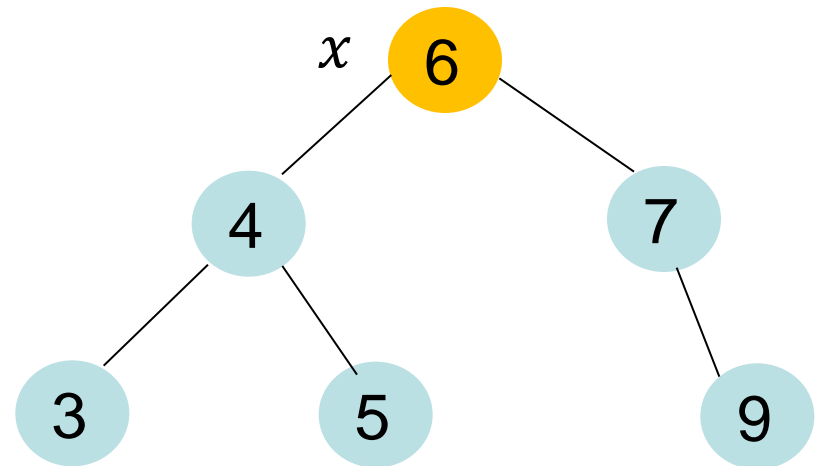


## Datenstrukturen

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**  $\text{rc}[y] \leftarrow z$

Einfügen(8)

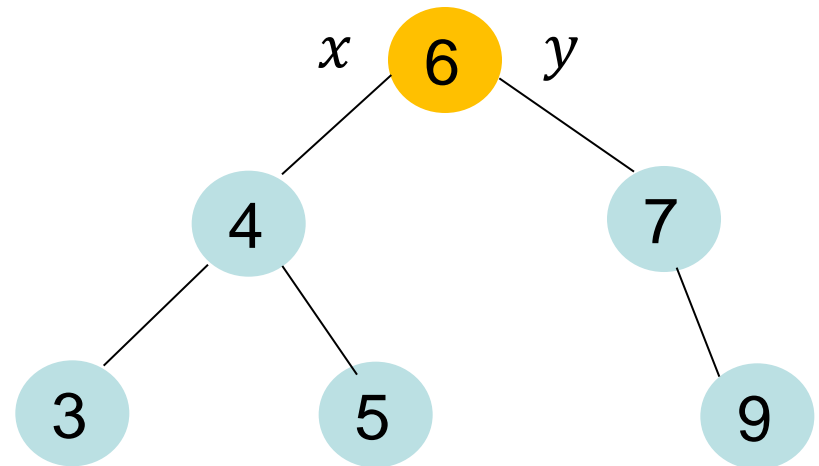


## Datenstrukturen

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**  $\text{rc}[y] \leftarrow z$

Einfügen(8)

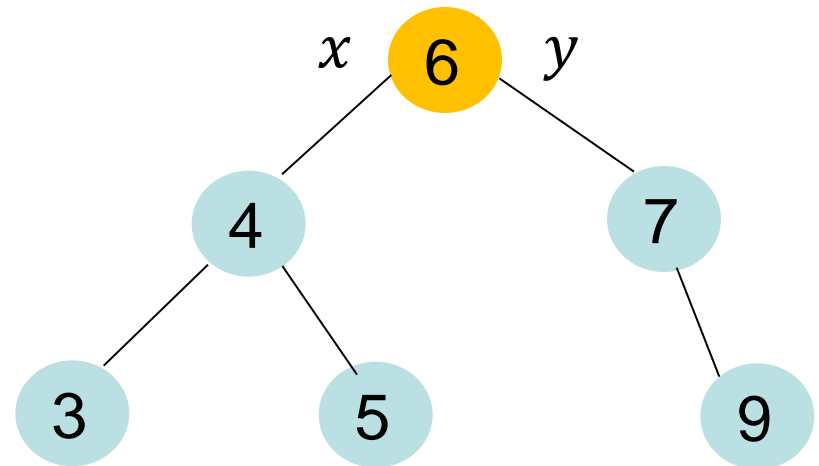


## Datenstrukturen

Einfügen( $T, z$ )

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

Einfügen(8)

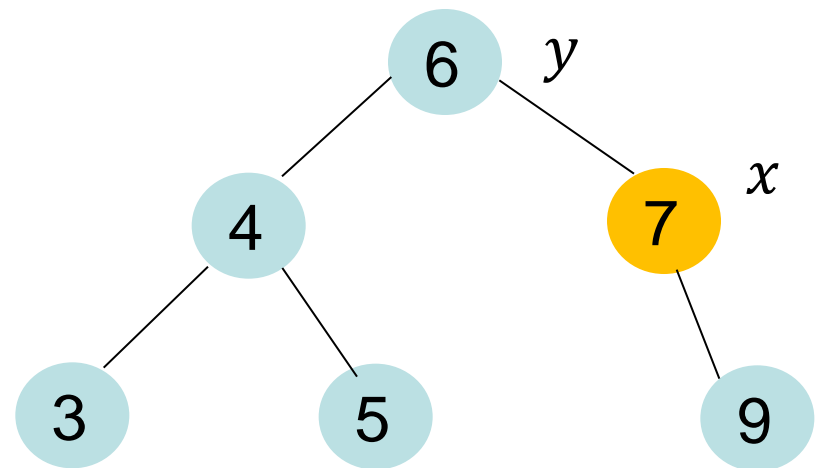


## Datenstrukturen

Einfügen( $T, z$ )

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

Einfügen(8)

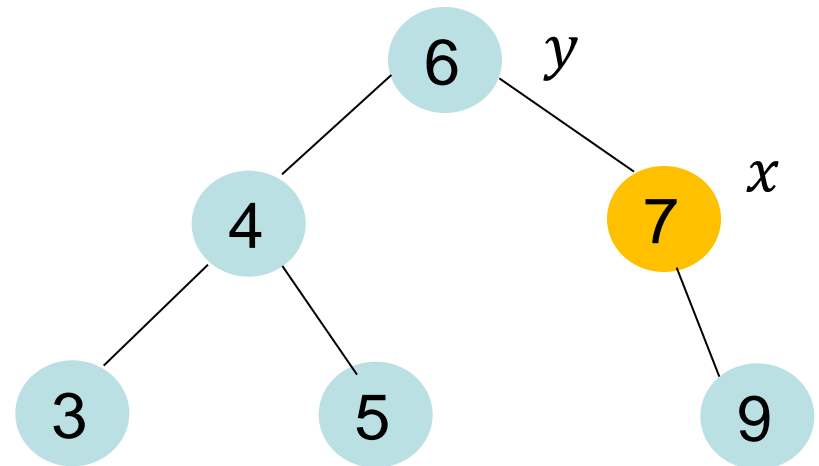


## Datenstrukturen

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**  $\text{rc}[y] \leftarrow z$

Einfügen(8)

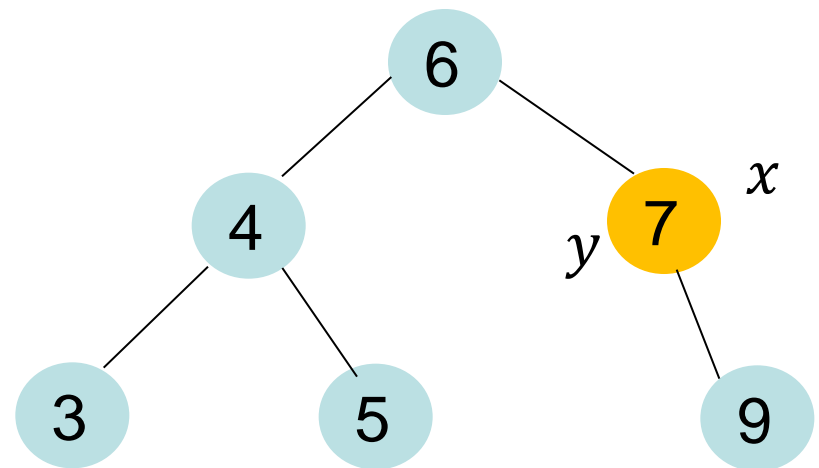


## Datenstrukturen

Einfügen( $T, z$ )

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

Einfügen(8)



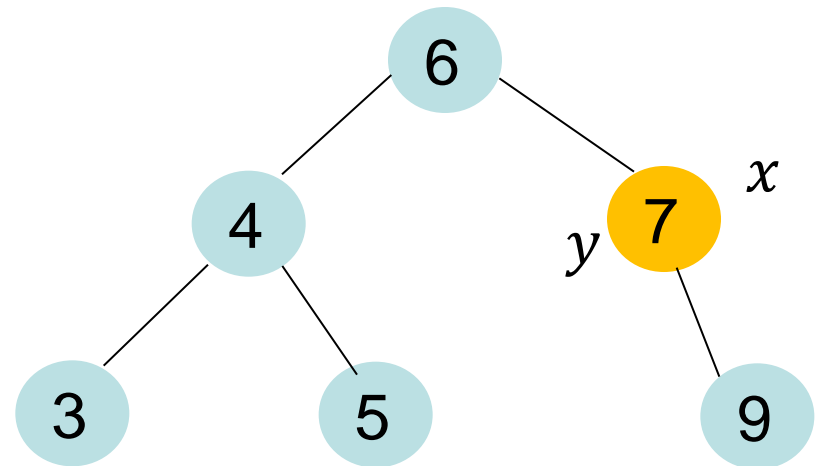


## Datenstrukturen

Einfügen( $T, z$ )

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

Einfügen(8)

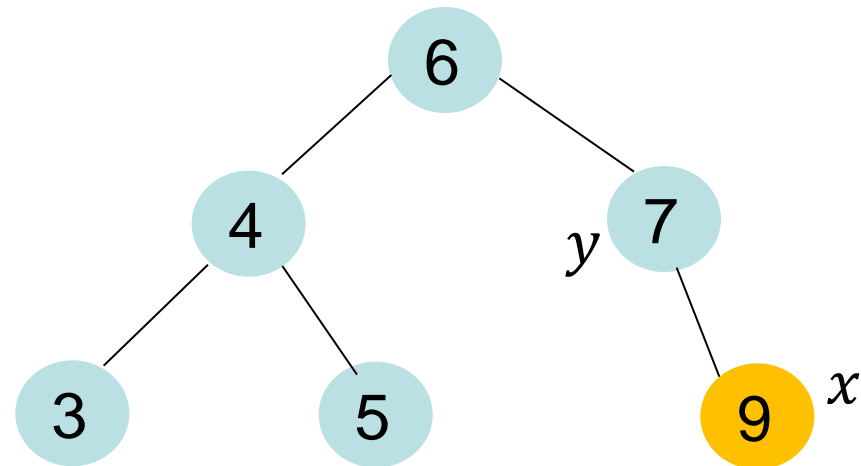


## Datenstrukturen

Einfügen( $T, z$ )

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

Einfügen(8)

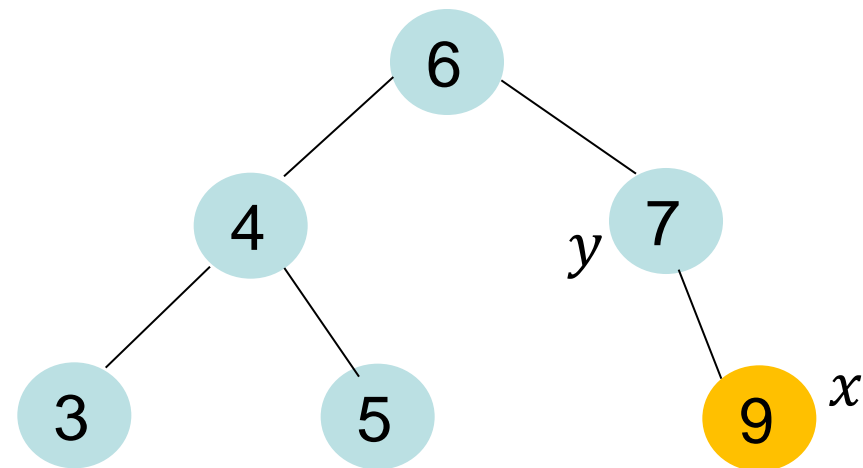


## Datenstrukturen

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**  $\text{rc}[y] \leftarrow z$

Einfügen(8)

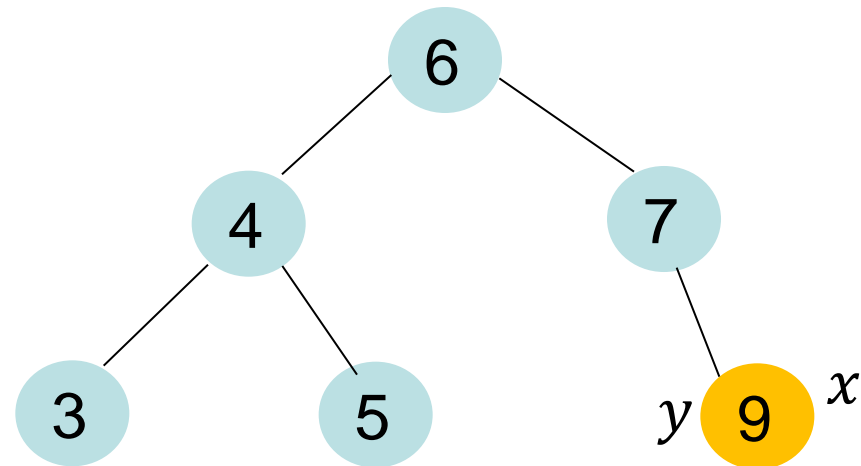


## Datenstrukturen

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**  $\text{rc}[y] \leftarrow z$

Einfügen(8)

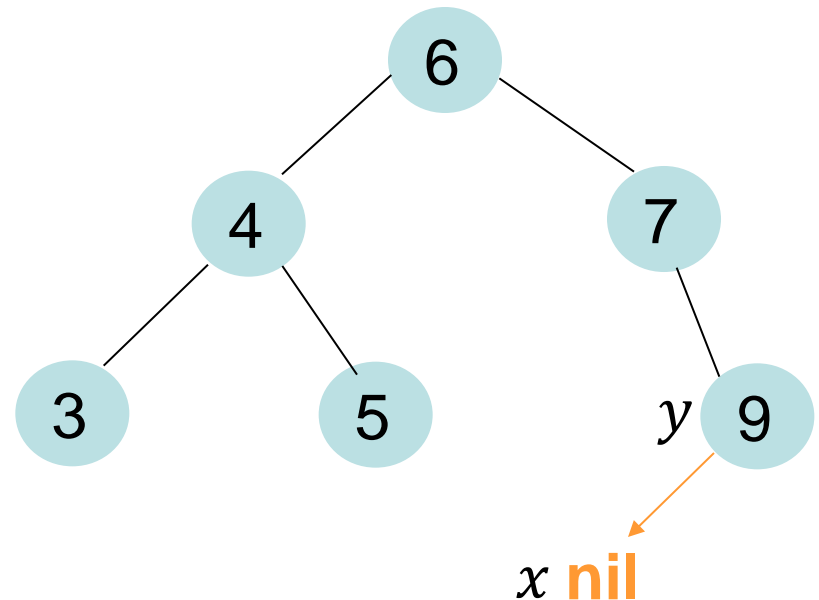


## Datenstrukturen

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**  $\text{rc}[y] \leftarrow z$

Einfügen(8)

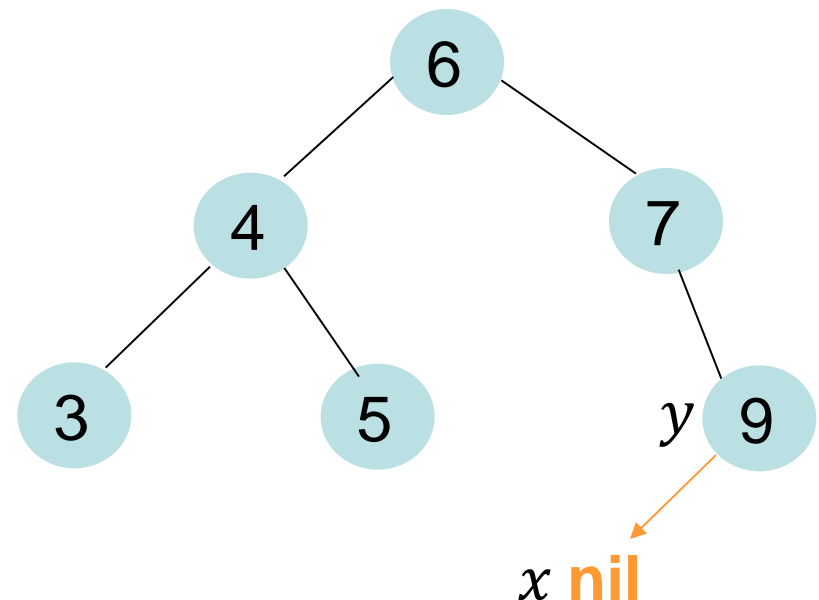


## Datenstrukturen

Einfügen( $T, z$ )

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

Einfügen(8)

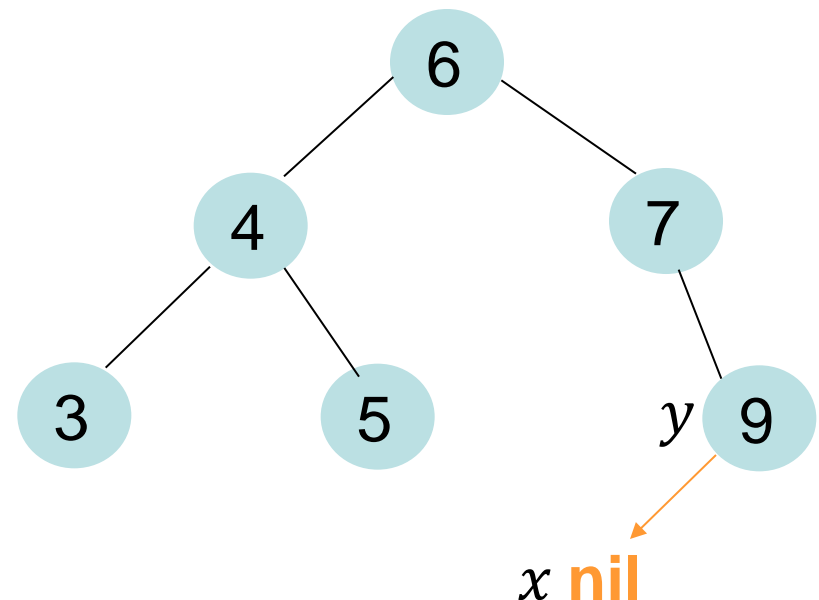


## Datenstrukturen

Einfügen( $T, z$ )

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

Einfügen(8)

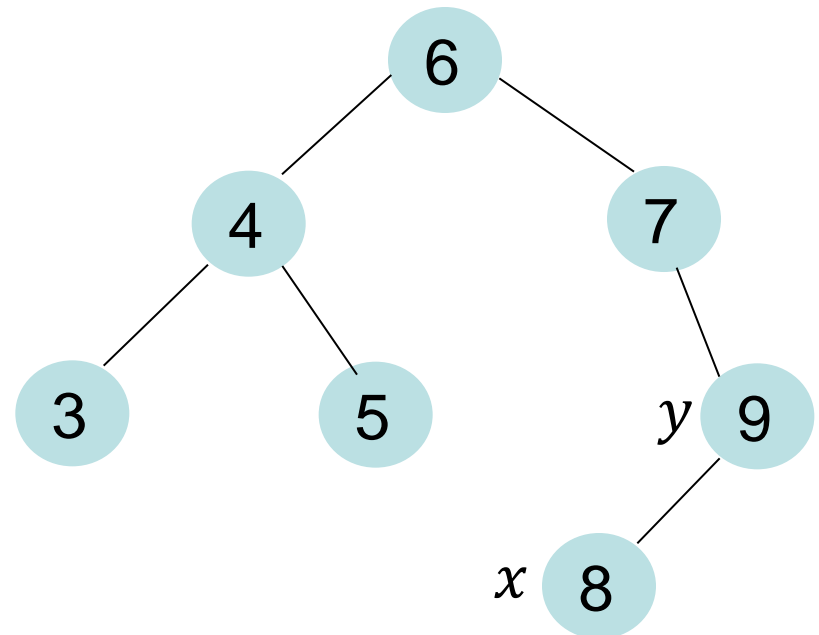


## Datenstrukturen

Einfügen( $T, z$ )

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

Einfügen(8)



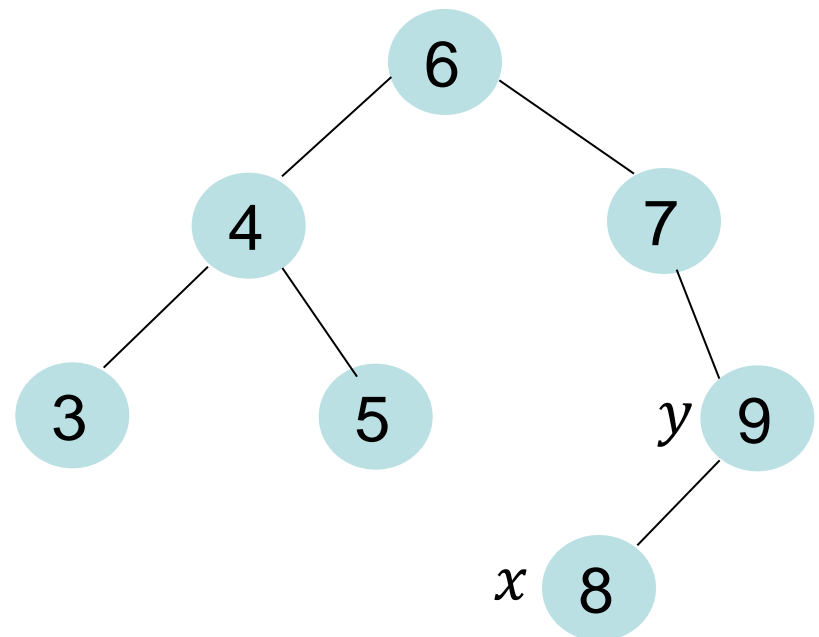


## Datenstrukturen

Einfügen( $T, z$ )

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

Einfügen(8)

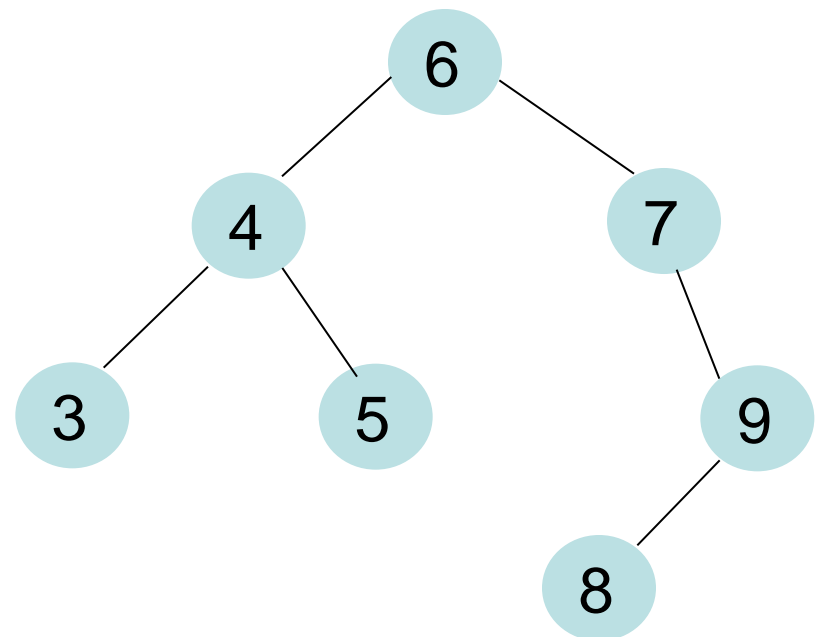


## Datenstrukturen

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**  $\text{rc}[y] \leftarrow z$

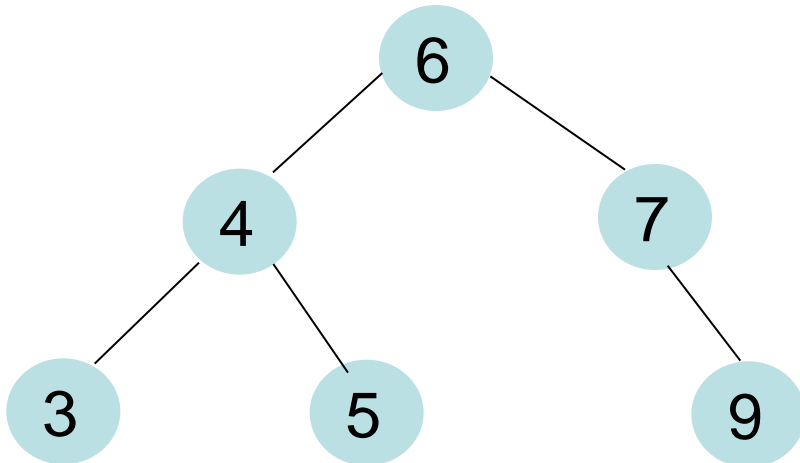
Laufzeit  $\mathbf{O}(h)$



## Datenstrukturen

### Löschen

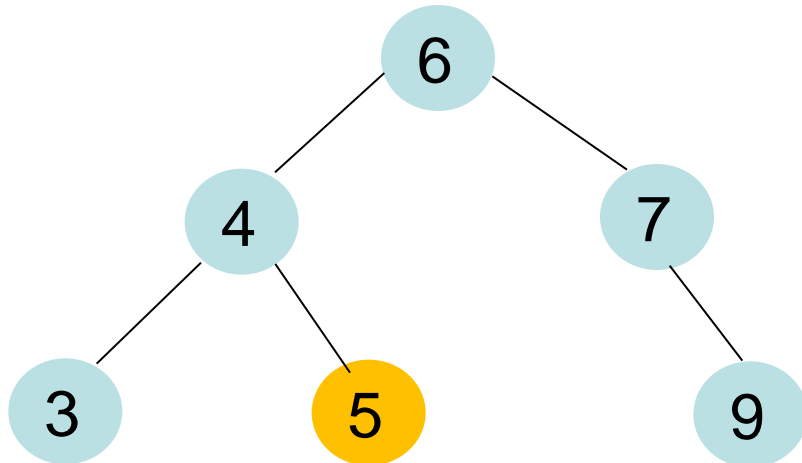
- 3 unterschiedliche Fälle
- (a) zu löschendes Element  $z$  hat keine Kinder
- (b) zu löschendes Element  $z$  hat ein Kind
- (c) zu löschendes Element  $z$  hat zwei Kinder



## Datenstrukturen

### *Fall (a)*

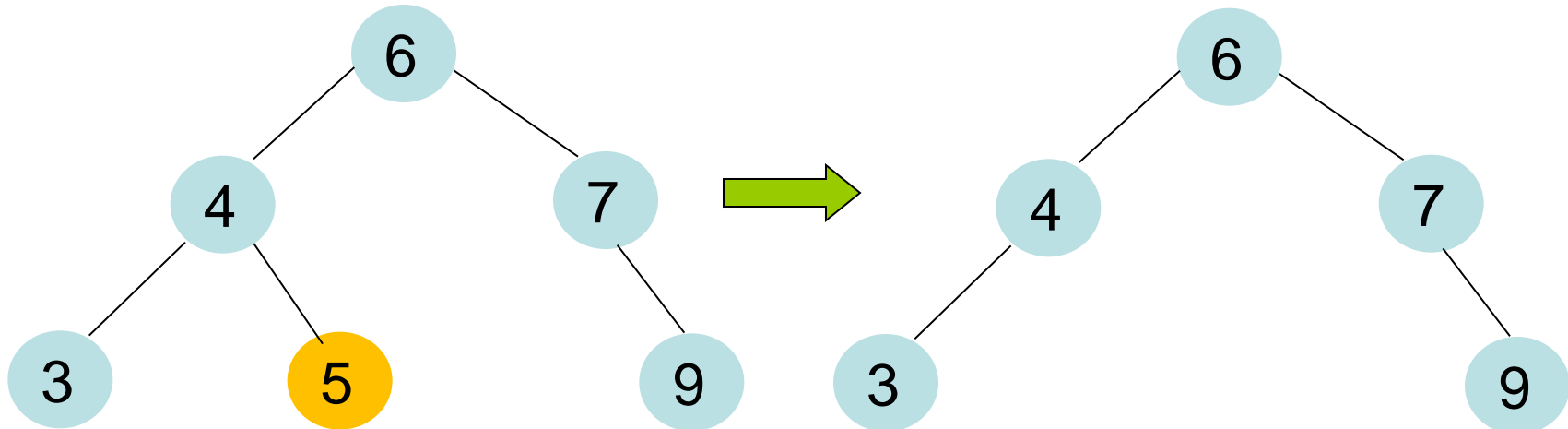
- zu löschendes Element  $z$  hat keine Kinder



## Datenstrukturen

### *Fall (a)*

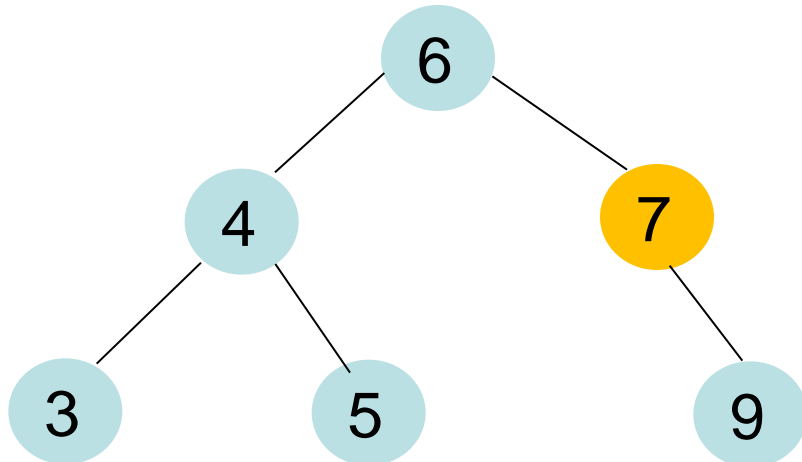
- zu löschendes Element  $z$  hat keine Kinder
- Entferne Element



## Datenstrukturen

### *Fall (b)*

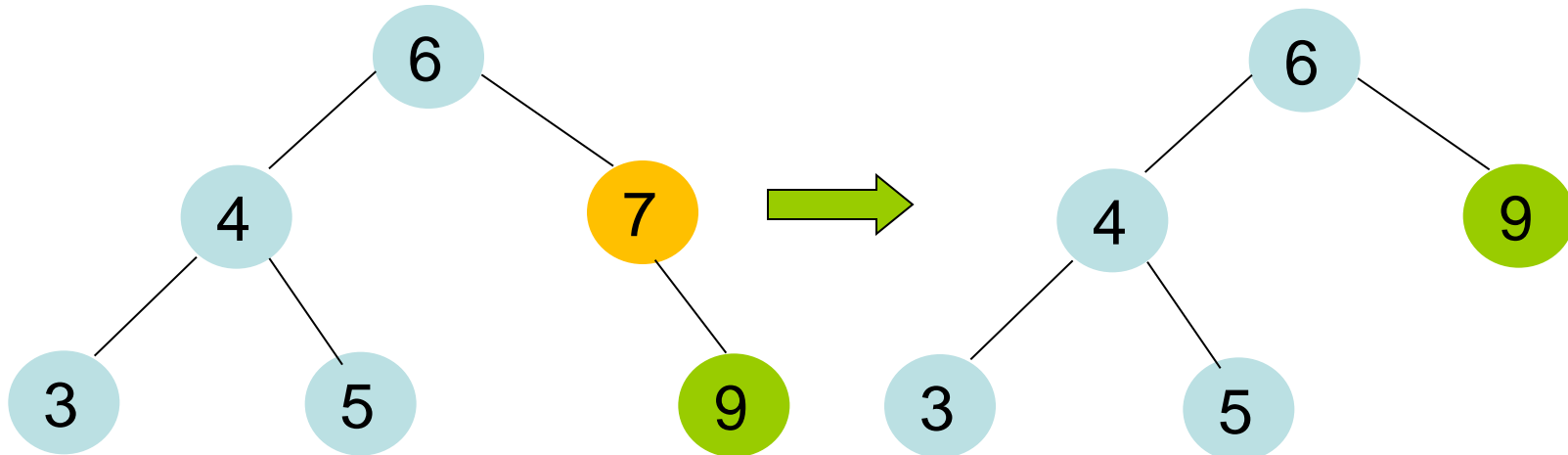
- Zu löschendes Element  $z$  hat 1 Kind



## Datenstrukturen

### *Fall (b)*

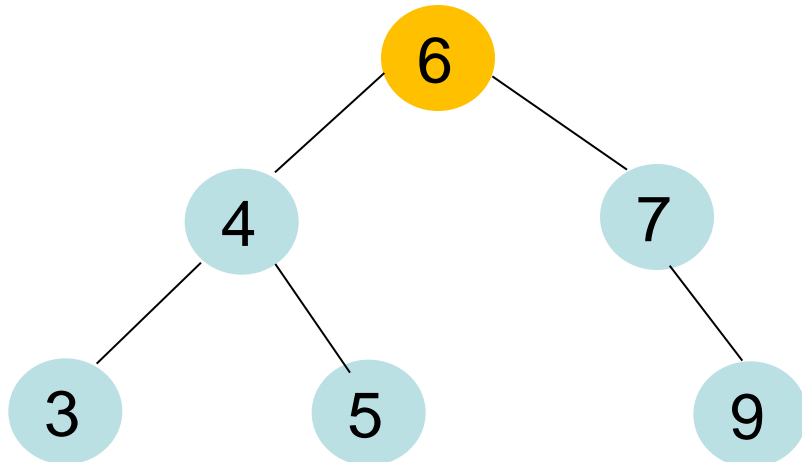
- Zu löschendes Element  $z$  hat 1 Kind



## Datenstrukturen

### *Fall (c)*

- Zu löschendes Element  $z$  hat 2 Kinder

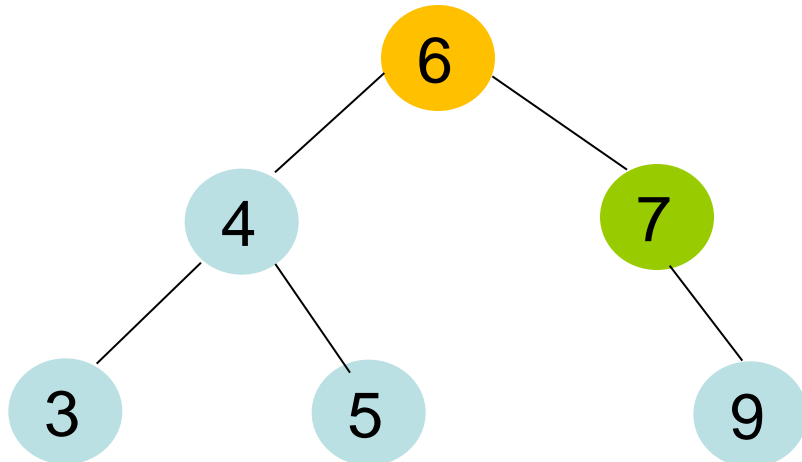




## Datenstrukturen

### *Fall (c)*

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

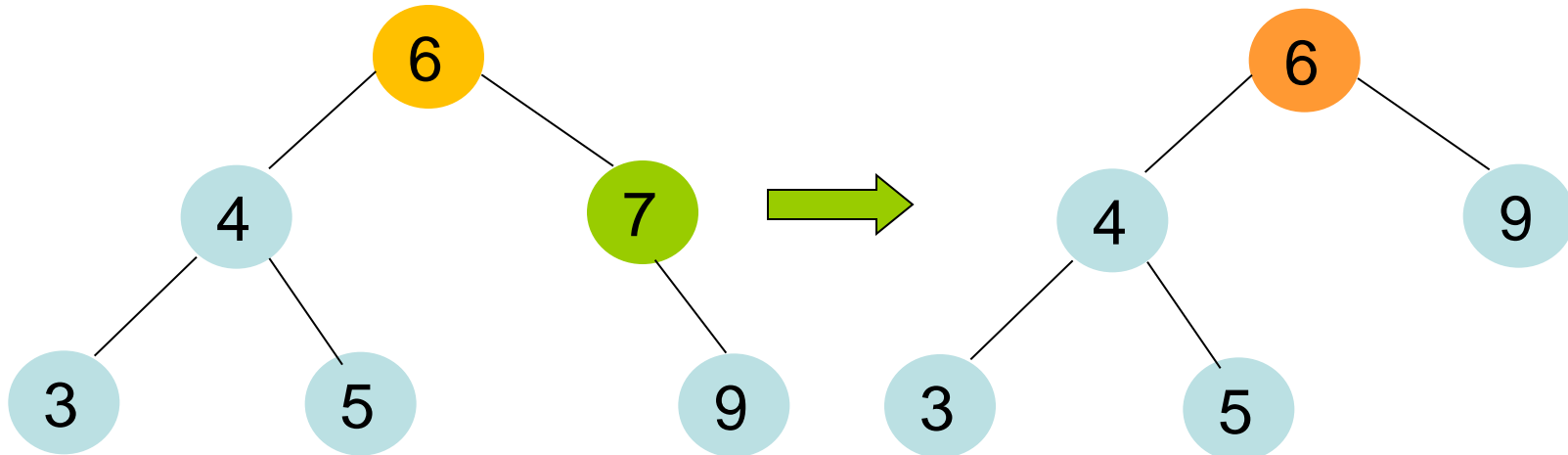


## Datenstrukturen

### Fall (c)

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

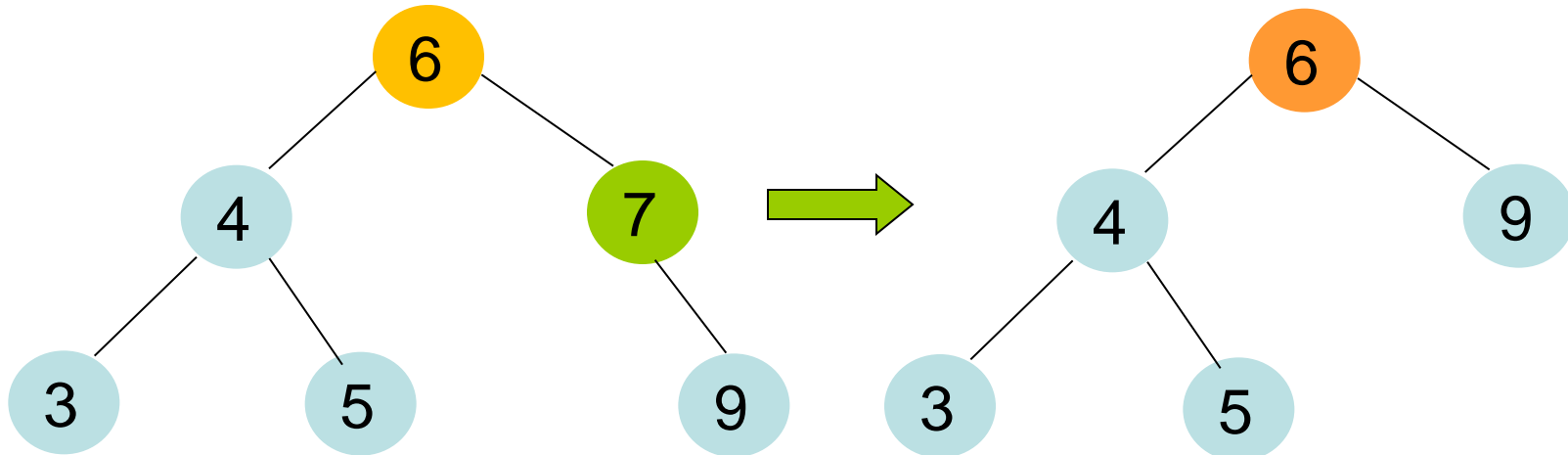
Nachfolger  
hat nur ein  
Kind



## Datenstrukturen

### *Fall (c)*

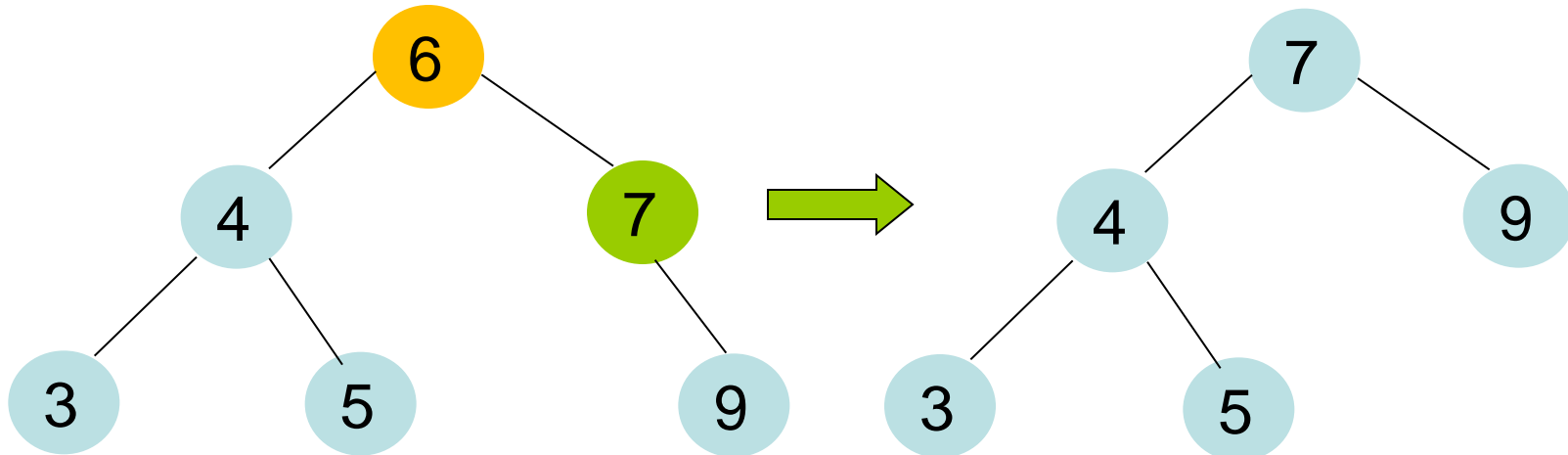
- Zu löschendes Element  $z$  hat 2 Kinder
- Schritt 1: Bestimme Nachfolger von  $z$
- Schritt 2: Entferne Nachfolger



## Datenstrukturen

### Fall (c)

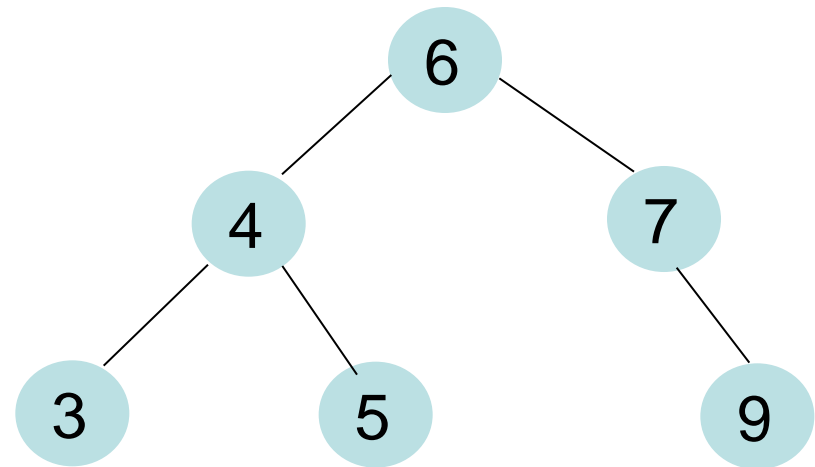
- Zu löschendes Element  $z$  hat 2 Kinder
- Schritt 1: Bestimme Nachfolger von  $z$
- Schritt 2: Entferne Nachfolger
- Schritt 3: Ersetze  $z$  durch Nachfolger



## Datenstrukturen

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.  $\text{key}[z] \leftarrow \text{key}[y]$



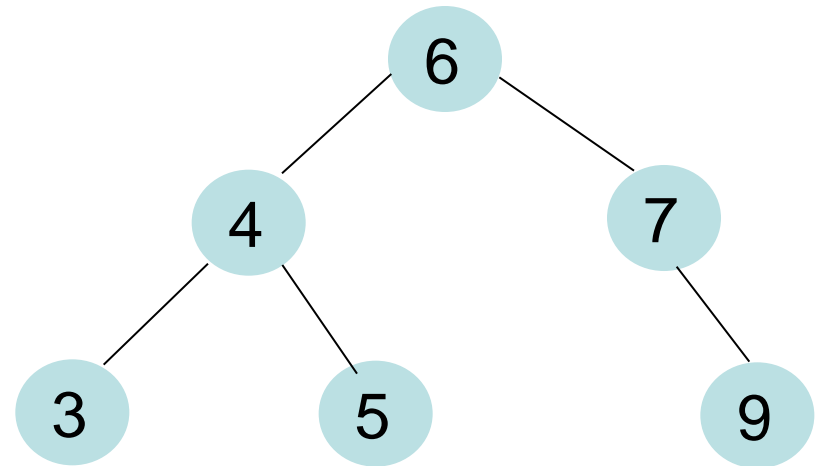
## Datenstruk

Referenz auf  $z$  wird  
übergeben!

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)



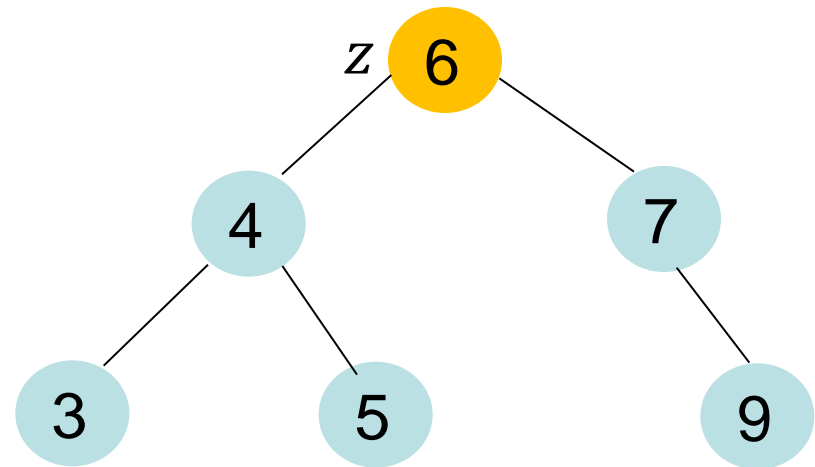
## Datenstrukturen

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.  $\text{key}[z] \leftarrow \text{key}[y]$

Bestimme Knoten, der gelöscht werden soll. Der Knoten hat nur einen Nachfolger

Löschen(6)



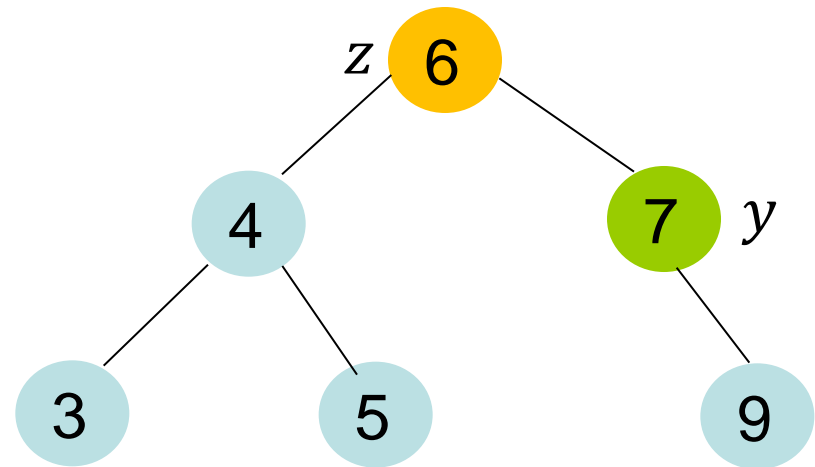
## Datenstrukturen

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.  $\text{key}[z] \leftarrow \text{key}[y]$

Bestimme Knoten, der gelöscht werden soll. Der Knoten hat nur einen Nachfolger

Löschen(6)





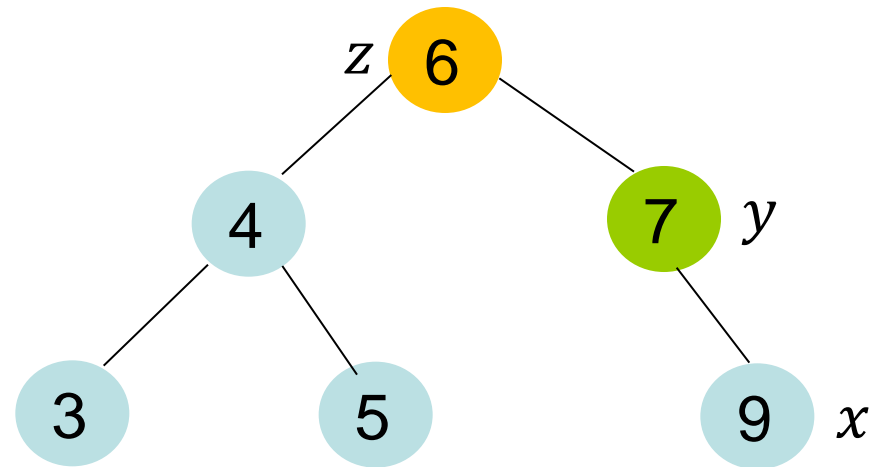
## Datenstrukturen

Löschen( $T, z$ )

1. **if**  $lc[z] = \text{nil}$  **or**  $rc[z] = \text{nil}$  **then**
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.    $\text{key}[z] \leftarrow \text{key}[y]$

Bestimme das Kind  
von  $y$ , falls existent

en(6)



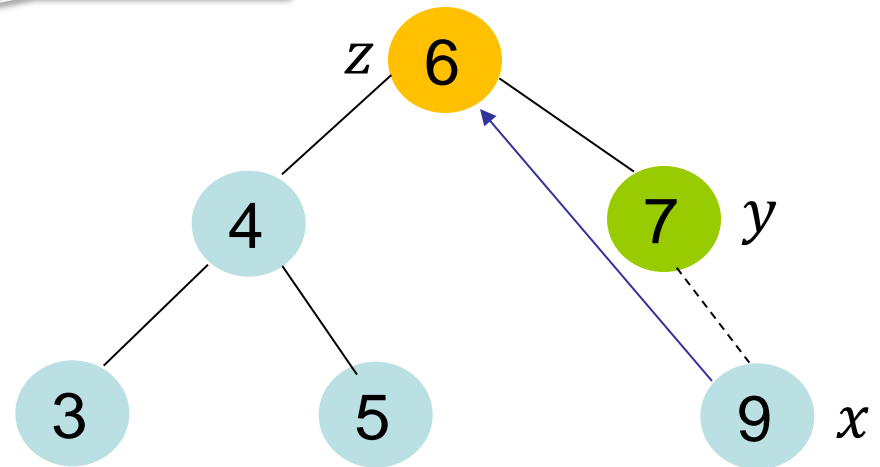
## Datenstrukturen

Löschen( $T, z$ )

1. **if**  $lc[z] = \text{nil}$  **or**  $rc[z] = \text{nil}$  **then**
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.    $\text{key}[z] \leftarrow \text{key}[y]$

Aktualisiere  
Vaterzeiger von  $x$

Löschen(6)

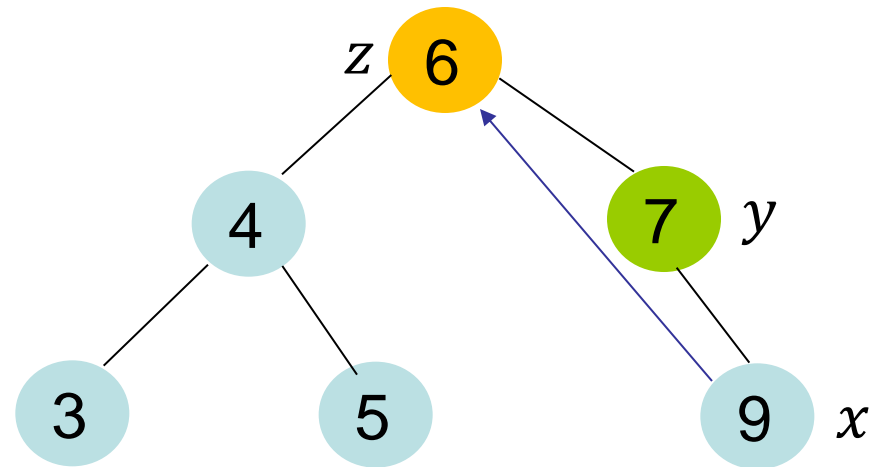


## Datenstrukturen

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)



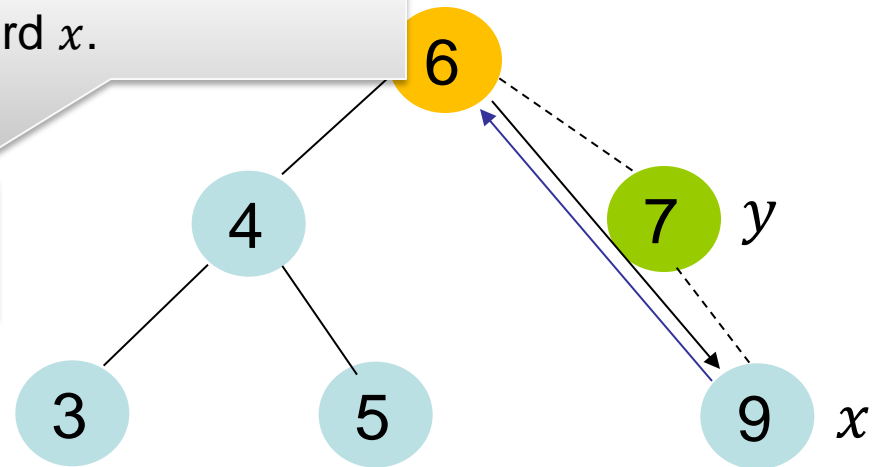
## Datenstrukturen

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)

Das rechte Kind von  
 $z$  wird  $x$ .



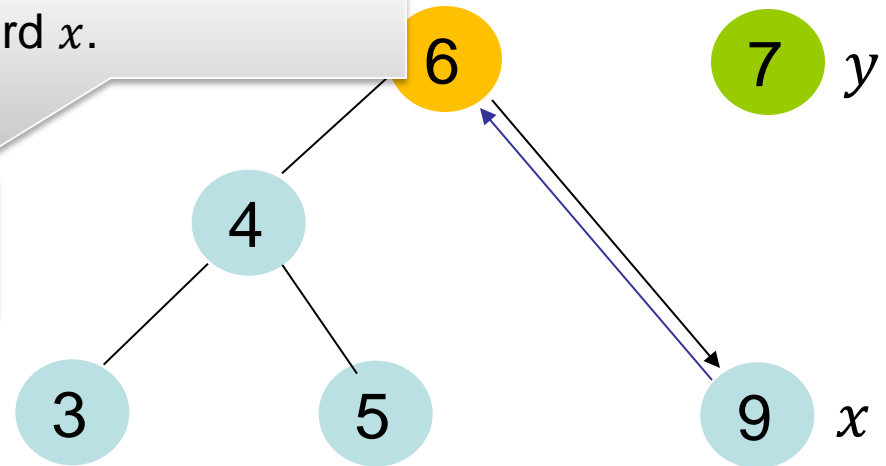
## Datenstrukturen

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)

Das rechte Kind von  
 $z$  wird  $x$ .



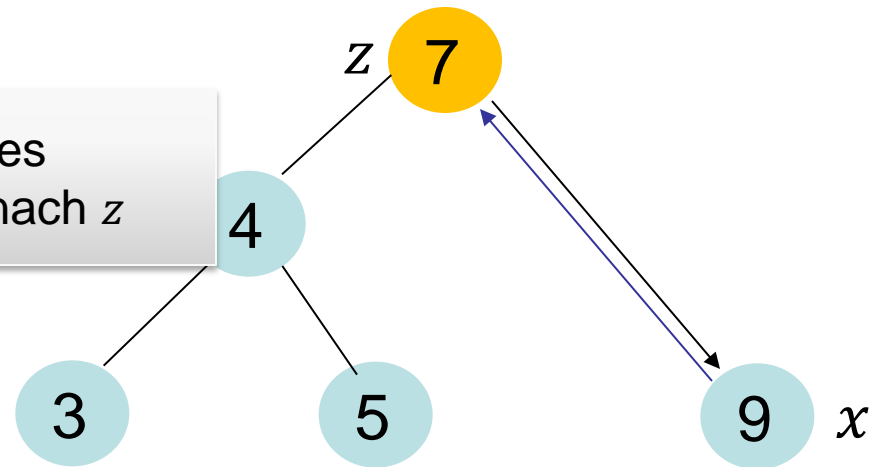
## Datenstrukturen

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**  $p[x] \leftarrow n[v]$
6. **if**  $p[y] = \text{nil}$  **then**  $\text{root} \leftarrow y$
7. **else if**  $y = lc[p[y]]$  **then**  $lc[p[y]] \leftarrow x$
8. **else**  $rc[p[y]] \leftarrow x$
9.  $\text{key}[z] \leftarrow \text{key}[y]$

Umkopieren des  
Inhalts von  $y$  nach  $z$

Löschen(6)

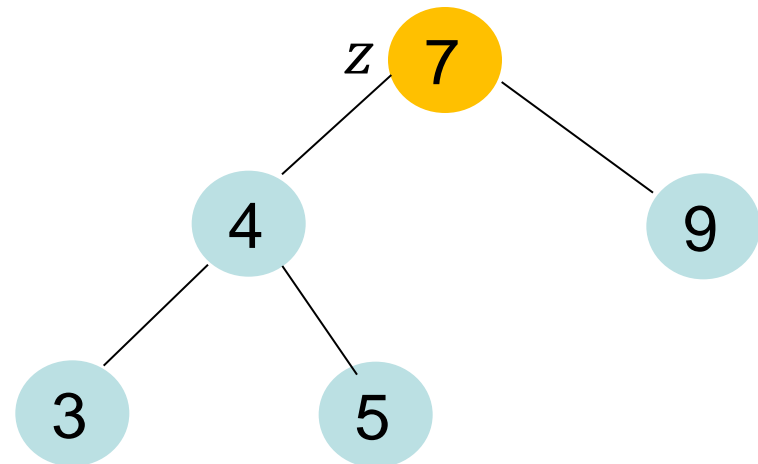


## Datenstrukturen

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

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)

