11. Elementare Datenstrukturen

Definition 11.1: Eine dynamische Menge ist gegeben durch eine oder mehrer Mengen von Objekten sowie Operationen auf diesen Mengen und den Objekten der Mengen. Dynamische Mengen werden auch abstrakte Datentypen (ADT) genannt.

Definition 11.2:

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

2. Werden auf einer Menge von Objekten die Operationen Einfügen, Entfernen und Suchen des Maximums betrachtet, so spricht man von einer Warteschlange.

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

Objekte in dynamischen Mengen (1)

- Objekte bestehen aus verschiedenen Feldern.
- Ein Feld speichert den *Schlüssel* des Objekts.
- Identifikation eines Objekts erfolgt über Schlüssel.
- Schlüssel von Objekten sind nicht notwendig paarweise verschieden.
- Falls Schlüssel paarweise verschieden sind, ist Menge von Objekten identisch zur Menge von Schlüsseln.

Objekte in dynamischen Mengen (2)

- Weitere Felder relevant für Datenstruktur, z.B. Felder für Referenzen auf andere Objekte.
- Andere Felder nur relevant für Anwendung, nicht für Datenstruktur – die Daten dieser Felder nennt man Satellitendaten.

Datenstrukturen

Zugriff auf Daten:

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

Beispiel:

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

Datenstrukturen

Zugriff auf Daten:

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

Beispiel:

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

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

Operationen in dynamischen Mengen

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

Stacks (Stapel) und Queues (Schlangen)

Definition 11.4:

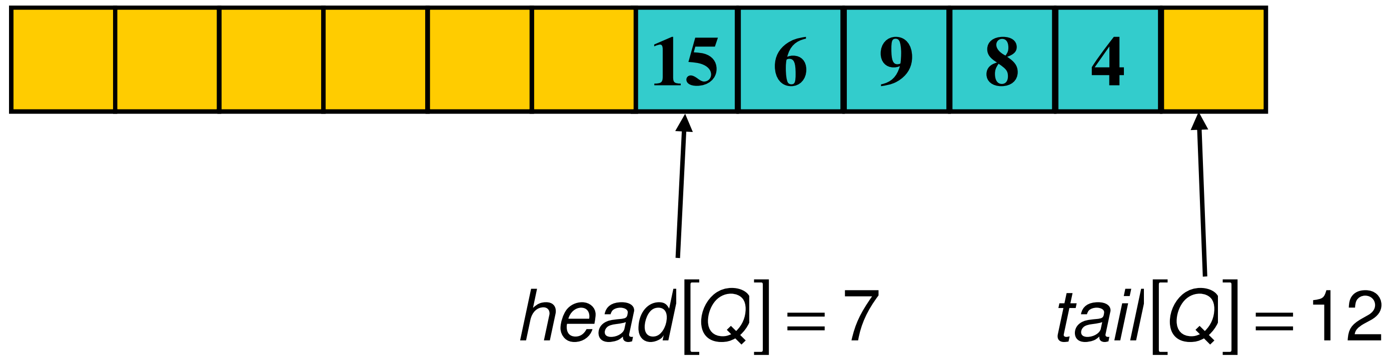
1. Stacks (Stapel) sind eine Datenstruktur, die die dynamische Menge LIFO (last-in-first-out) implementiert.
   In LIFOs sollen beliebige Objekte eingefügt und das zuletzt eingefügte Objekt ernfernt werden können.

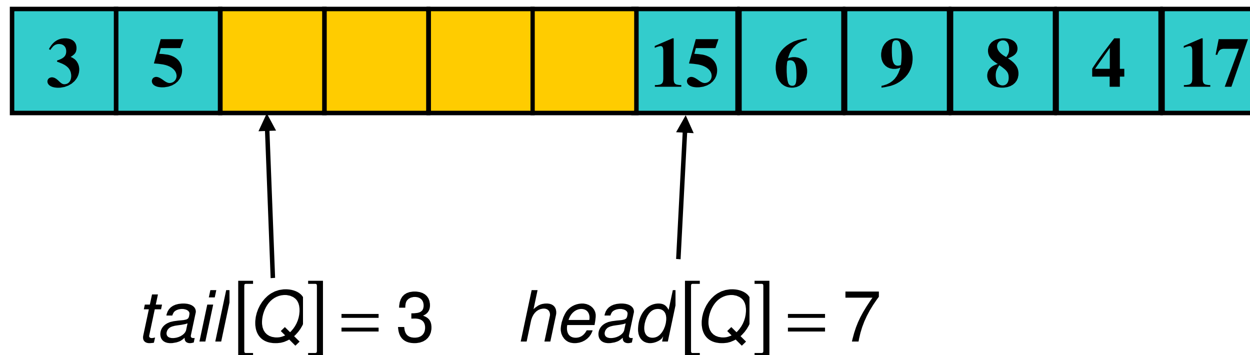
2. Queues (Schlangen) sind eine Datenstruktur, die die dynamische Menge FIFO (first-in-first-out) implementiert.
   In FIFOs sollen beliebige Objekte eingefügt und das am längsten in der Menge befindliche Objekt ernfernt werden können.

Queues

- Einfügen eines Objekts wird Enqueue genannt.
- Entfernen des am längsten in der Queue befindlichen Objekts wird Dequeue genannt.
- Queue mit maximal $n-1$ Objekten wird realisiert durch ein Array $Q[1\ldots n]$ mit zusätzlichen Feldern $head[Q]$, $tail[Q]$.
- $head[Q]$: Position des am längsten in Queue befindlichen Objekts
- $tail[Q]$: erste freie Position.
- Alle Indexberechnungen modulo n (+1), betrachten Array kreisförmig. Auf Position n folgt wieder Position 1.

Queue - Beispiele



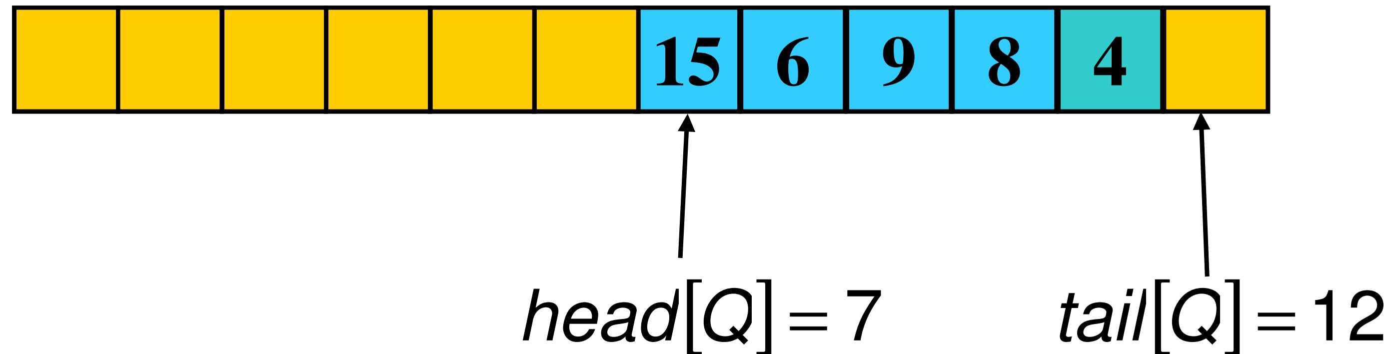


Queue - Operationen

Enqueue(Q, x)
1. $Q[tail[Q]] \leftarrow x$
2. **if** $tail[Q] = length[Q]$
3. **then** $tail[Q] \leftarrow 1$
4. **else** $tail[Q] \leftarrow tail[Q] + 1$

Dequeue(Q)
1. $\mathrm{x} \leftarrow Q[head[Q]]$
2. **if** $head[Q] = length[Q]$
3. **then** $head[Q] \leftarrow 1$
4. **else** $head[Q] \leftarrow head[Q] + 1$
5. **return** x

Illustration Enqueue



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

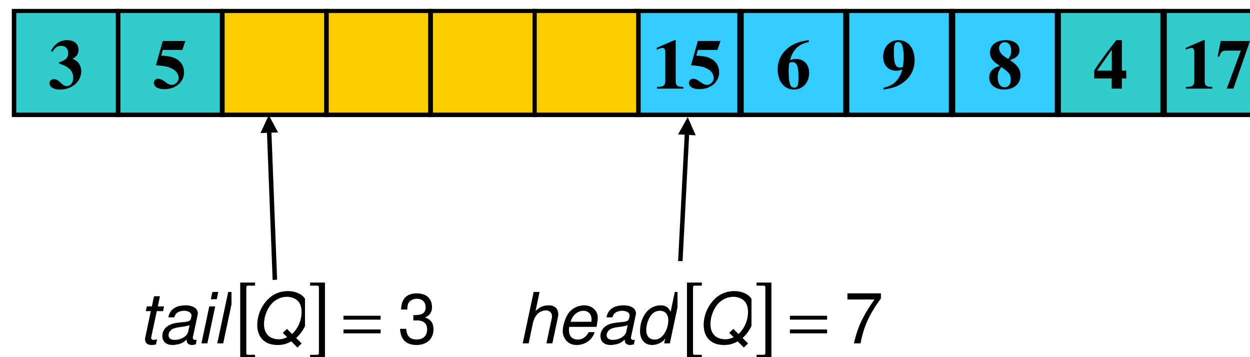
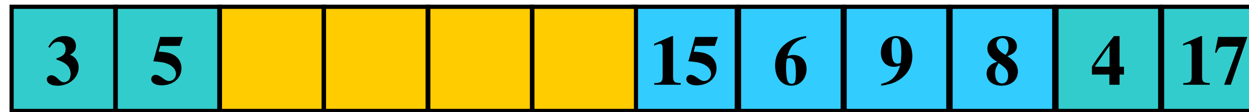


Illustration Dequeue



$tail[Q] = 3$ $head[Q] = 7$

Nach Dequeue:

$tail[Q] = 3$ $head[Q] = 8$

Laufzeit Queue-Operationen

Satz 11.6: Mit Queues können die Operationen einer FIFO in Zeit **O**(1) ausgeführt werden.

Doppelt verkettete Listen (1)

- Verkettete Listen bestehen aus einer Menge linear angeordneter Objekte.
- Anordnung realisiert durch Verweise.
- Unterstützen alle dynamischen Mengen mit den Operationen Insert, Delete, Search, Search-Minimum, usw. Unterstützung ist nicht unbedingt effizient.
- Objekte in doppelt verketteter Liste *L* besitzen mindestens drei Felder: key, next, prev. Außerdem Felder für Satellitendaten möglich.
- Zugriff auf Liste *L* durch Verweis/Zeiger head[*L*].

Doppelt verkettete Listen (2)

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

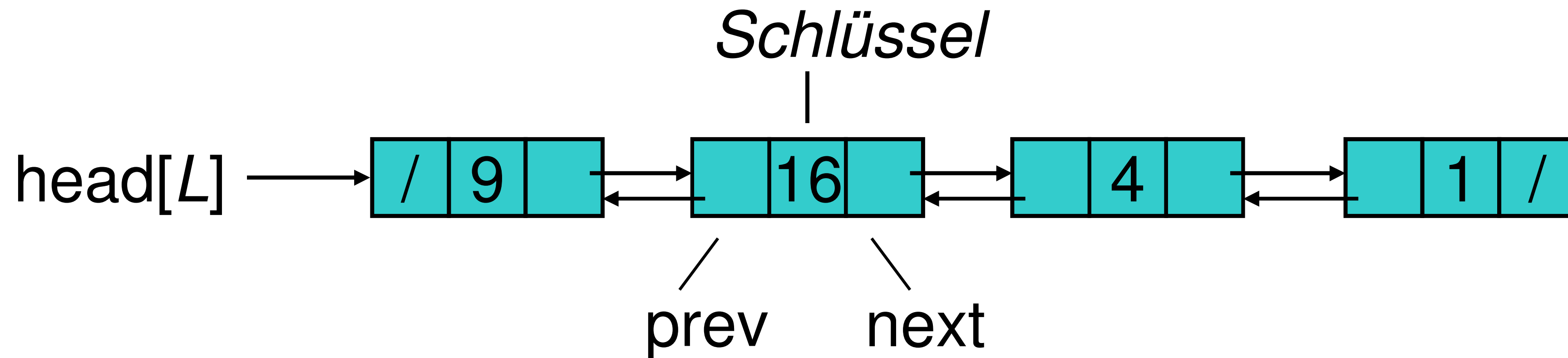
Varianten verketteter Listen

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

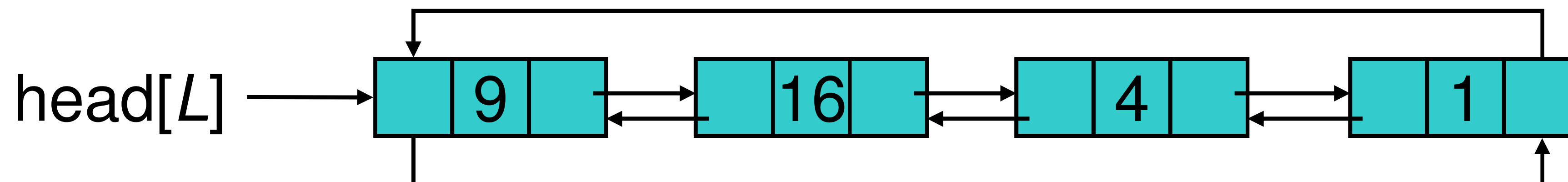
- Sortierte verkettete Liste: Schlüssel können sortiert werden. Reihenfolge in Liste entspricht sortierter Reihenfolge der Schlüssel.

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

Doppelt verkettete Listen - Beispiel



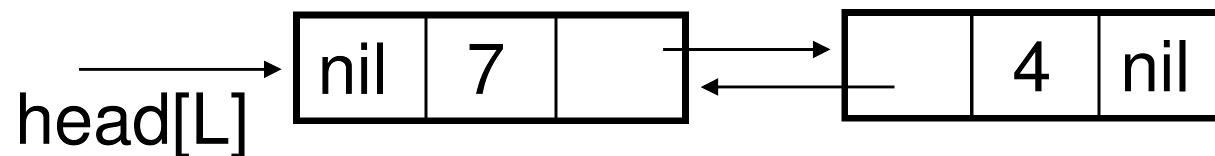
zyklisch doppelt verkette Liste:



Doppelt verkettete Listen - Beispiel

Einfügen(L,x)

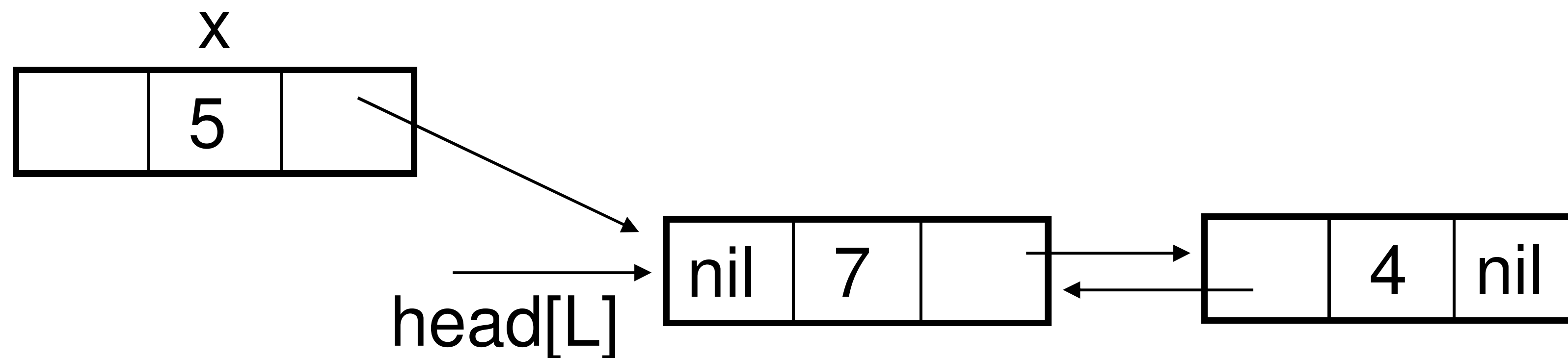
1. next[x] $\leftarrow$ head[L]
2. **if** head[L] $\neq$ **nil then** prev[head[L]] $\leftarrow$ x
3. head[L] $\leftarrow$ x
4. prev[x] $\leftarrow$ **nil**



Doppelt verkettete Listen - Beispiel

Einfügen(L,x)

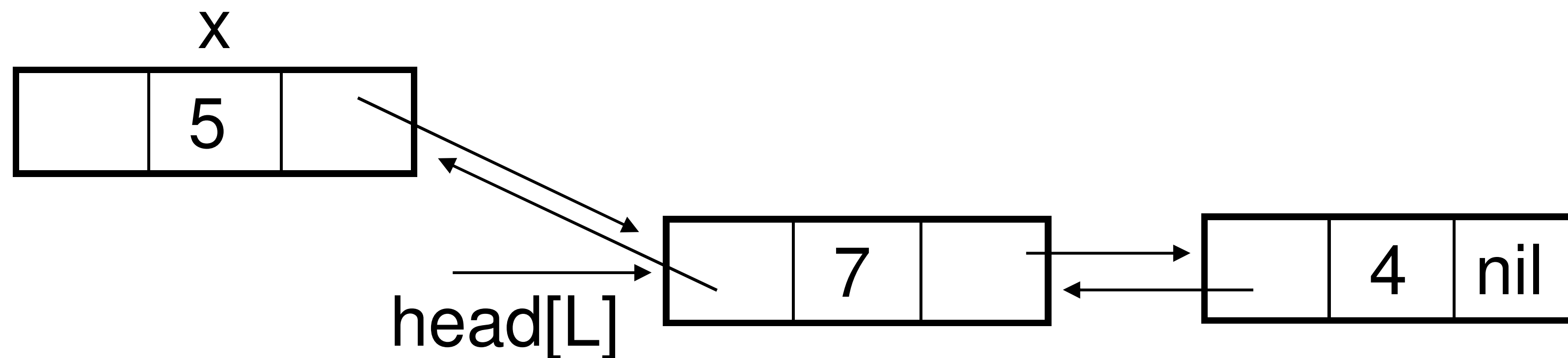
1. next[x] ← head[L]
2. **if** head[L] ≠ **nil then** prev[head[L]] ← x
3. head[L] ← x
4. prev[x] ← **nil**



Doppelt verkettete Listen - Beispiel

Einfügen(L,x)

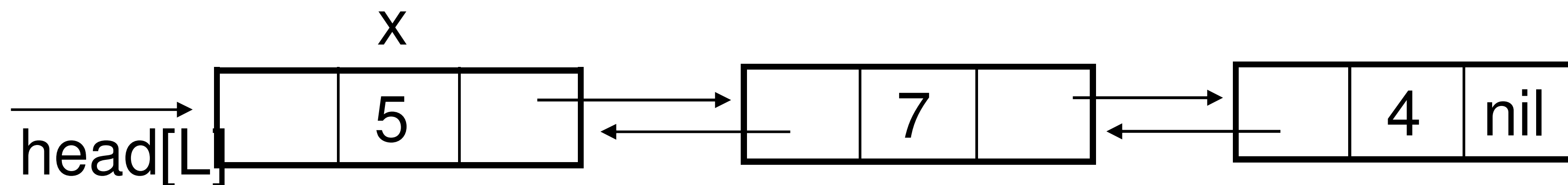
1. next[x] $\leftarrow$ head[L]
2. **if** head[L] $\neq$ **nil then** prev[head[L]] $\leftarrow$ x
3. head[L] $\leftarrow$ x
4. prev[x] $\leftarrow$ **nil**



Doppelt verkettete Listen - Beispiel

Einfügen(L,x)

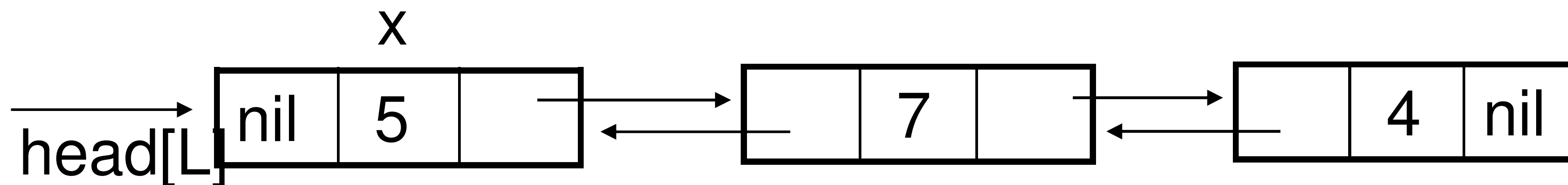
1. next[x] $\leftarrow$ head[L]
2. **if** head[L] $\neq$ **nil then** prev[head[L]] $\leftarrow$ x
3. head[L] $\leftarrow$ x
4. prev[x] $\leftarrow$ **nil**



Doppelt verkettete Listen - Beispiel

Einfügen(L,x)

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



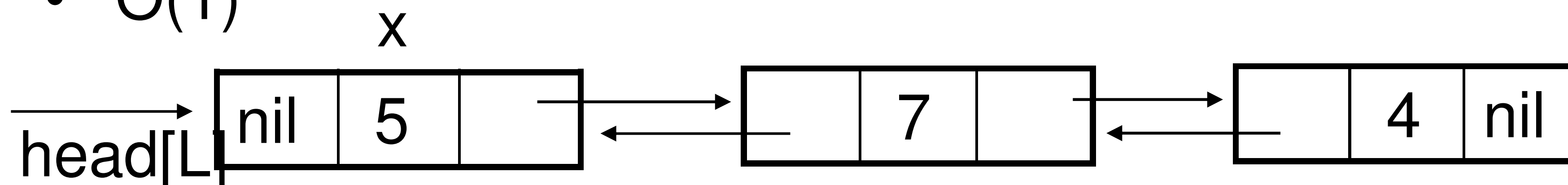
Doppelt verkettete Listen - Beispiel

Einfügen(L,x)

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

Laufzeit:

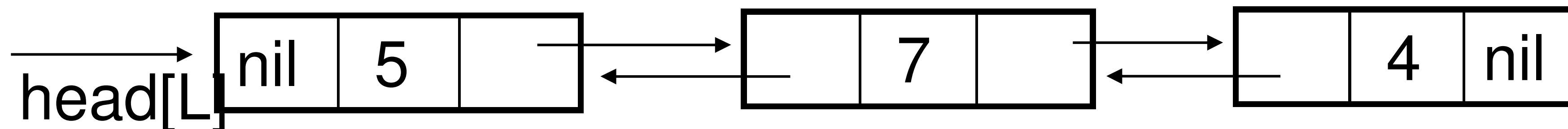
- O(1)



Doppelt verkettete Listen - Beispiel

Löschen(L,x)

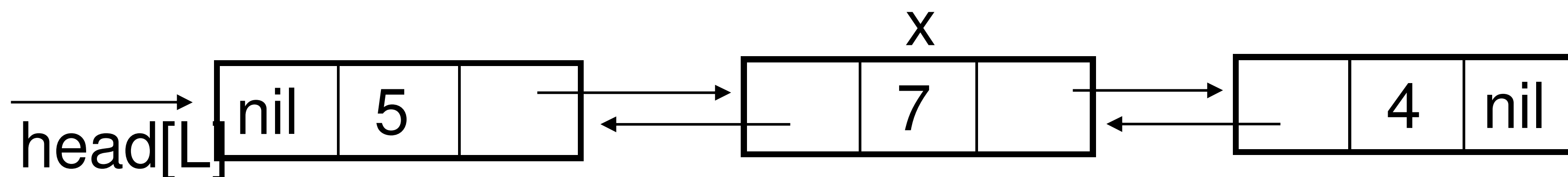
1. **if** prev[x] $\neq$ **nil then** next[prev[x]] $\leftarrow$ next[x]
2. **else** head[L] $\leftarrow$ next[x]
3. **if** next[x] $\neq$ **nil then** prev[next[x]] $\leftarrow$ prev[x]



Doppelt verkettete Listen - Beispiel

Löschen(L,x)

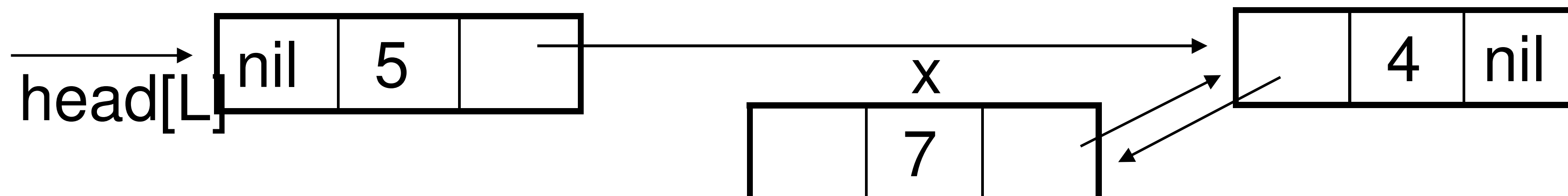
1. **if** prev[x] $\neq$ **nil then** next[prev[x]] $\leftarrow$ next[x]
2. **else** head[L] $\leftarrow$ next[x]
3. **if** next[x] $\neq$ **nil then** prev[next[x]] $\leftarrow$ prev[x]



Doppelt verkettete Listen - Beispiel

Löschen(L,x)

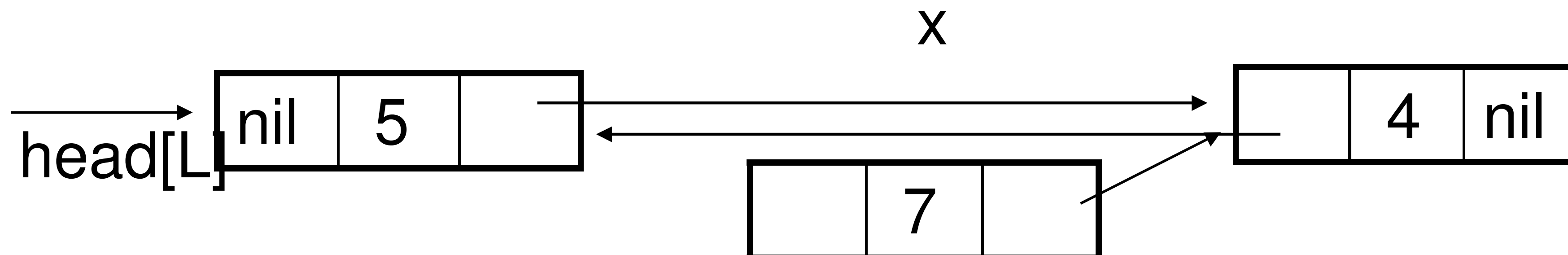
1. **if** prev[x] $\neq$ **nil then** next[prev[x]] $\leftarrow$ next[x]
2. **else** head[L] $\leftarrow$ next[x]
3. **if** next[x] $\neq$ **nil then** prev[next[x]] $\leftarrow$ prev[x]



Doppelt verkettete Listen - Beispiel

Löschen(L,x)

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



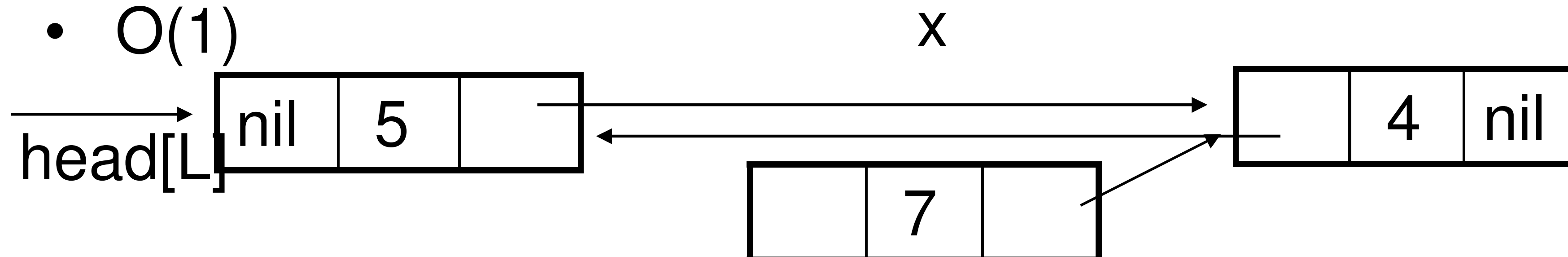
Doppelt verkettete Listen - Beispiel

Löschen(L,x)

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

Laufzeit:

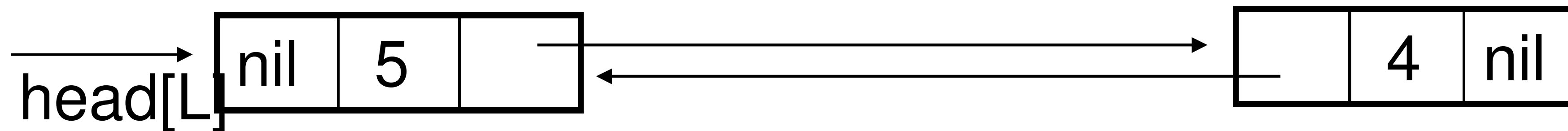
- O(1)



Doppelt verkettete Listen - Beispiel

Suche(L,k)

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

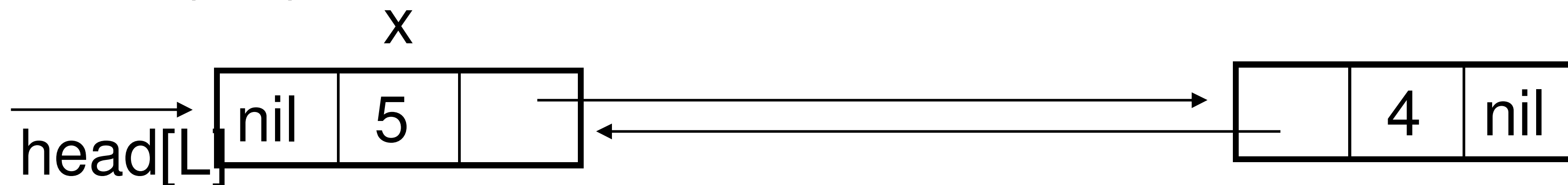


Doppelt verkettete Listen - Beispiel

Suche(L,k)

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

Suche(L,4)

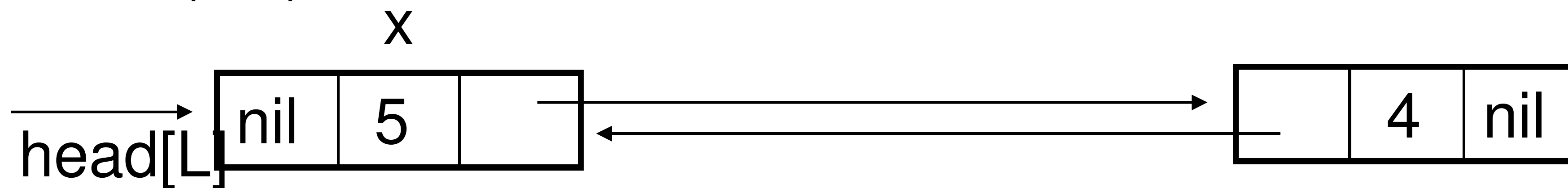


Doppelt verkettete Listen - Beispiel

Suche(L,k)

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

Suche(L,4)

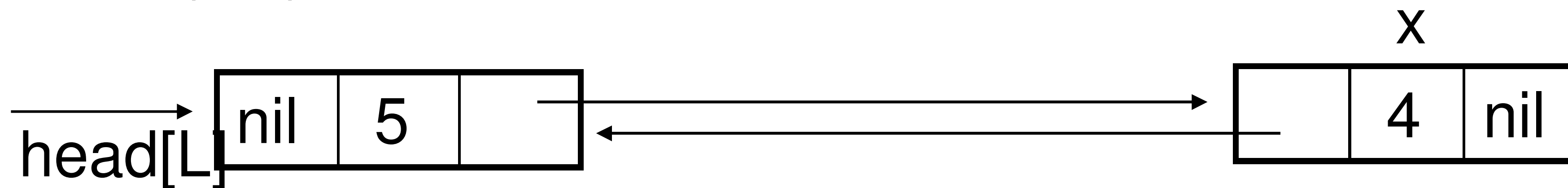


Doppelt verkettete Listen - Beispiel

Suche(L,k)

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

Suche(L,4)

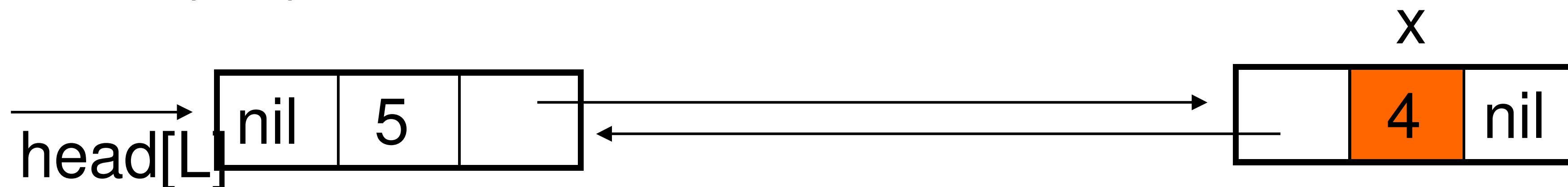


Doppelt verkettete Listen - Beispiel

Suche(L,k)

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

Suche(L,4)

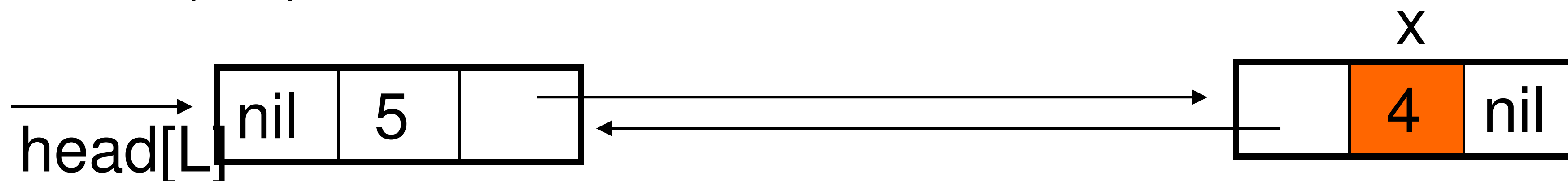


Doppelt verkettete Listen - Beispiel

Suche(L,k)

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

Suche(L,4)



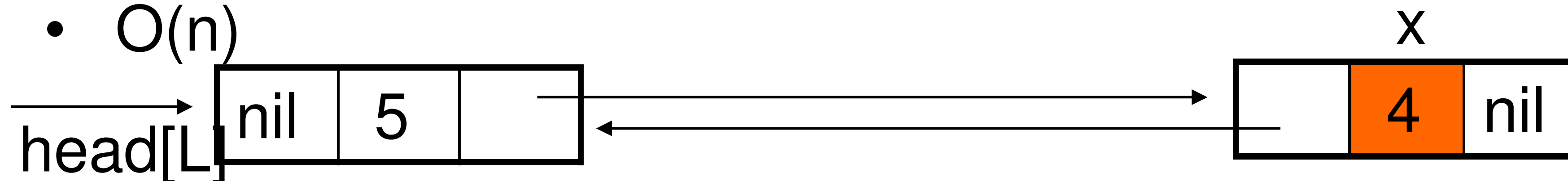
Doppelt verkettete Listen - Beispiel

Suche(L,k)

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

Laufzeit:

- O(n)



Doppelt verkettete Listen - Beispiel

Datenstruktur Liste:

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

Vorteile:

- Schnelles Einfügen/Löschen
- $O(n)$ Speicherbedarf

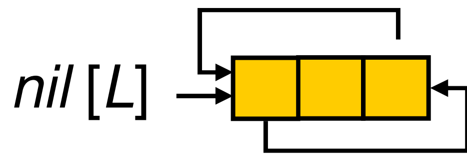
Nachteile:

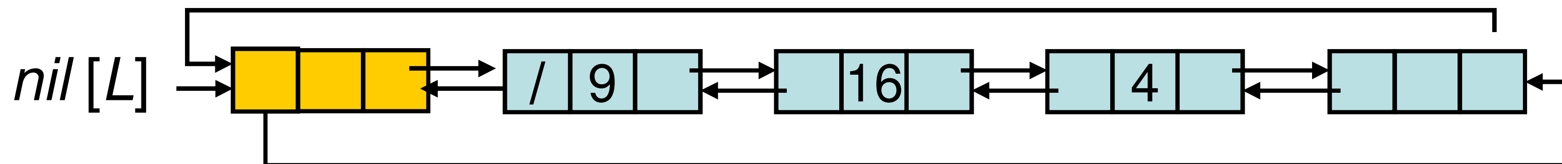
- Hohe Laufzeit für Suche

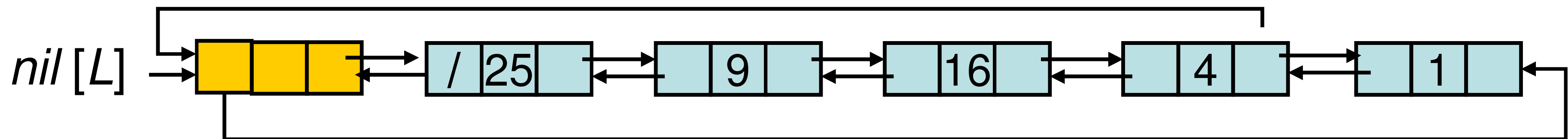
Vereinfachung durch Sentinels (Wächter)

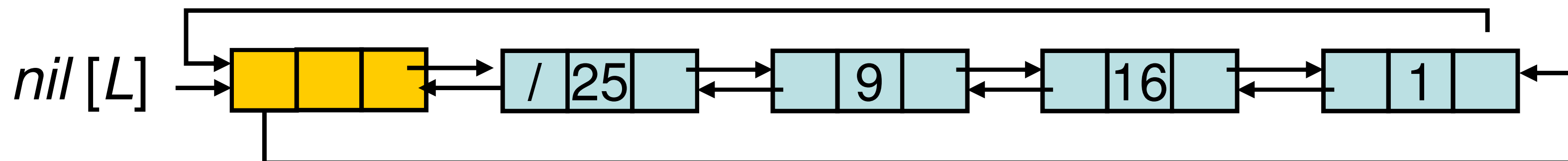
- Vermeidung von Abfragen head[L] ≠ NIL und x ≠ NIL durch Einfügen eines zusätzlichen Objekts *nil*[*L*].
- *nil*[L] besitzt drei Felder key, prev, next, aber key[*nil*[*L*]]= NIL.
- next[*nil*[*L*]] verweist auf erstes (richtiges) Objekt in Liste.
- prev[*nil*[*L*]] verweist auf letztes Objekt in Liste.
- Ersetzten in Pseudocode NIL durch *nil*[*L*].
- head[*L*] nicht mehr benötigt, ersetzt durch next[*nil*[*L*]].
- Objekt zur Vereinfachung von Randbedingungen wird Sentinel oder Wächter genannt.

Zyklisch verkette Listen mit Sentinel









Durchsuchen verketteter Liste (mit Sentinel)

List-Search´(L,k)
1. $x \leftarrow \text{next}[nil[L]]$
2. **while** $x \neq nil[L] \wedge \text{key}[x] \neq k$
3. **do** $x \leftarrow \text{next}[x]$
4. **return** x

Einfügen in verkettete Liste (mit Sentinel)

$$
\begin{aligned}
&\text{List-Insert´}(L,x)\\
&1.\ \text{next}[x] \leftarrow \text{next}[nil[L]]\\
&2.\ \text{prev}[\text{next}[nil[L]]] \leftarrow x\\
&3.\ \text{next}[nil[L]] \leftarrow x\\
&4.\ \text{prev}[x] \leftarrow nil[L]
\end{aligned}
$$

Löschen aus verketteter Liste (mit Sentinel)

$$\begin{aligned}
&\text{List - Delete}'(L,x)\\
&1.\ \text{next}[\text{prev}[x]] \leftarrow \text{next}[x]\\
&2.\ \text{prev}[\text{next}[x]] \leftarrow \text{prev}[x]
\end{aligned}$$

Doppelt verkettete Listen - Beispiel

Drei grundlegende Datenstrukturen

- Feld
- Stack/Queue
- (doppelt) verkettete Liste

Diskussion

- Alle drei Strukturen haben gewichtige Nachteile
- Suche: $\Theta(n)$
- Zeiger/Referenzen helfen beim Speichermanagement

Binäre Bäume

- Neben Felder für Schlüssel und Satellitendaten Felder p, lc, rc.

- x Knoten: p[*x*] Verweis auf Elternknoten, lc[*x*] Verweis auf linkes Kind, rc[*x*] Verweis auf rechtes Kind.

- Falls p[*x*]=NIL, dann ist *x* Wurzel des Baums.

- lc[*x*] / rc[*x*]=NIL: kein linkes/rechtes Kind.

- Zugriff auf Baum *T* durch Verweis root[*T*] auf Wurzel-knoten.

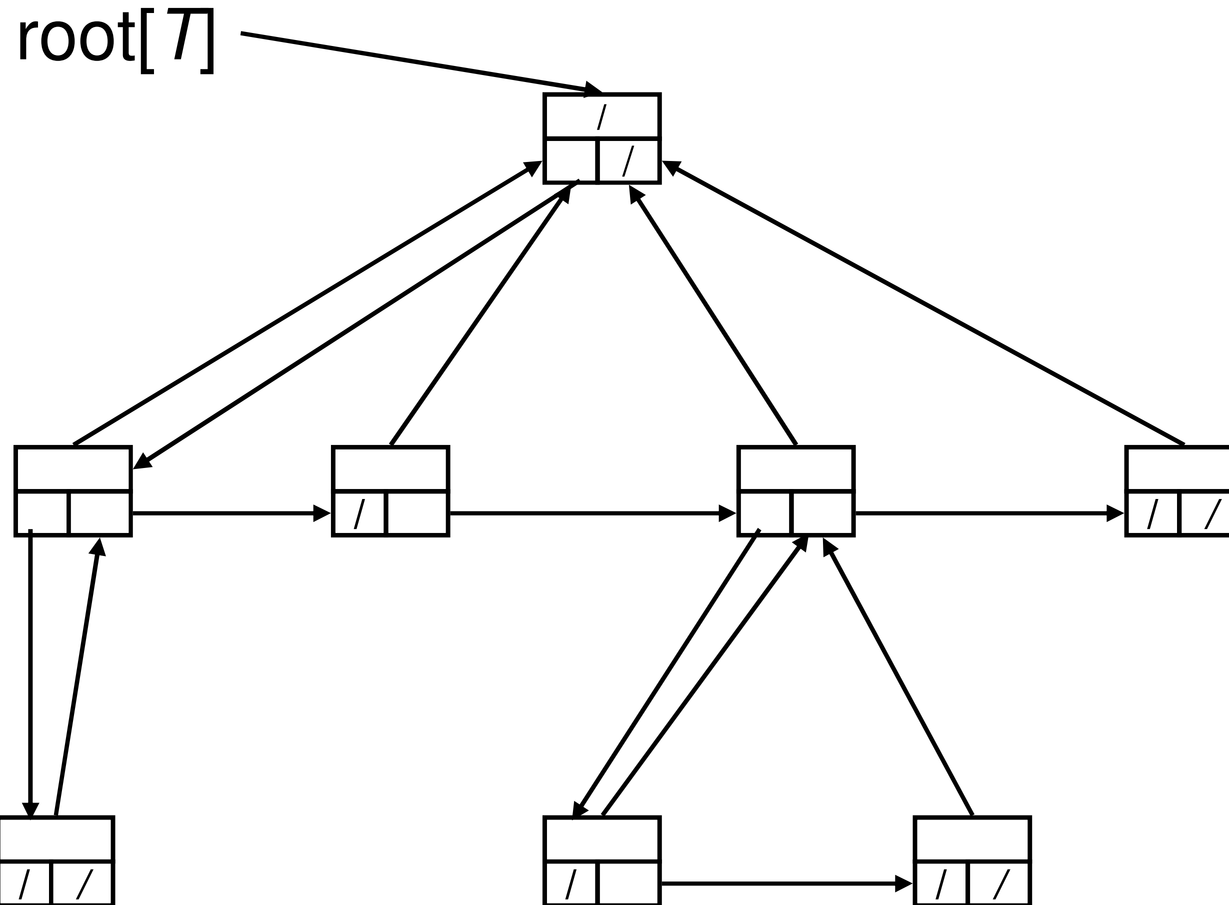
Binäre Bäume - Illustration

root[T]

Allgemeine Bäume

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

Allgemeine Bäume



Binäre Suchbäume

Binäre Suchbäume

- Verwende Binärbaum
- Speichere Schlüssel „geordnet“

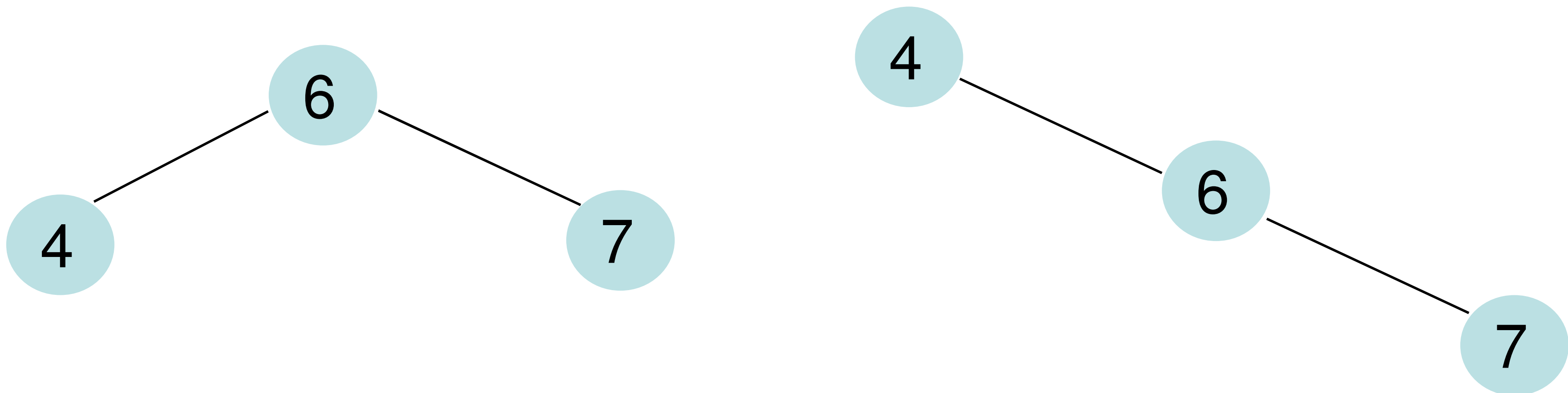
Binäre Suchbaumeigenschaft:

- Sei x Knoten im binären Suchbaum
- Ist y Knoten im linken Unterbaum von x, dann gilt $key[y] \leq key[x]$
- Ist y Knoten im rechten Unterbaum von x, dann gilt $key[y] \geq key[x]$

Binäre Suchbäume

Unterschiedliche Suchbäume

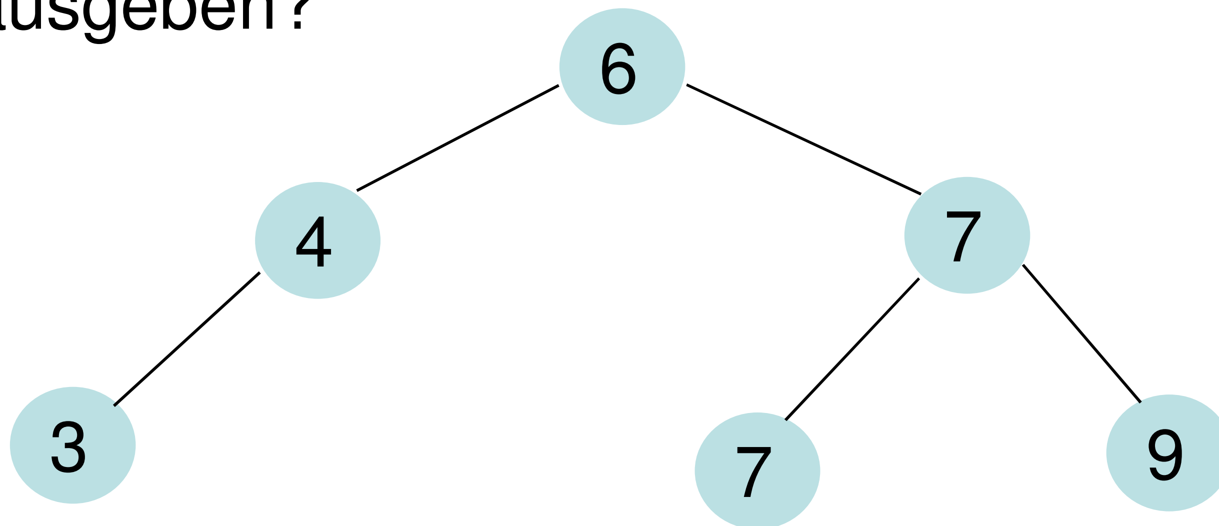
- Schlüsselmenge 4,6,7
- Wir erlauben mehrfache Vorkommen desselben Schlüssels



Binäre Suchbäume

Ausgabe aller Schlüssel

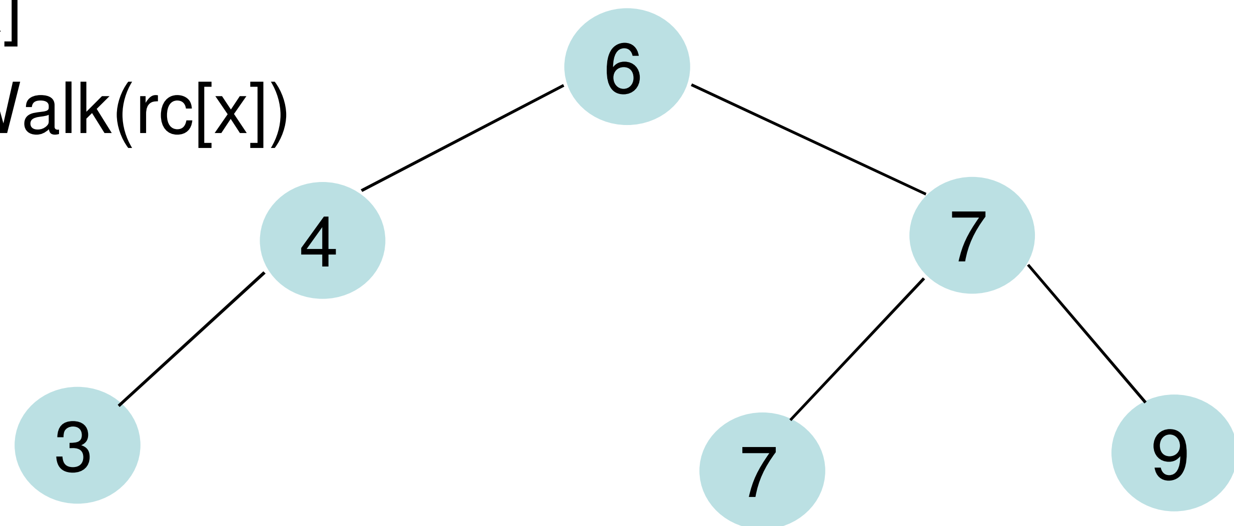
- Gegeben binärer Suchbaum
- Wie kann man alle Schlüssel aufsteigend sortiert in $\Theta(n)$ Zeit ausgeben?



Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

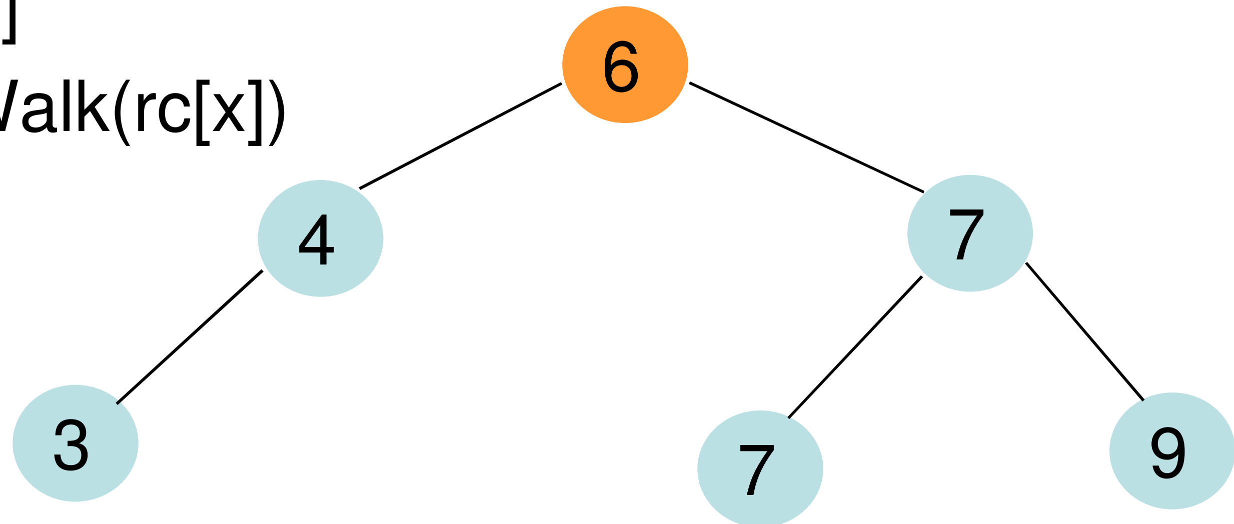


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

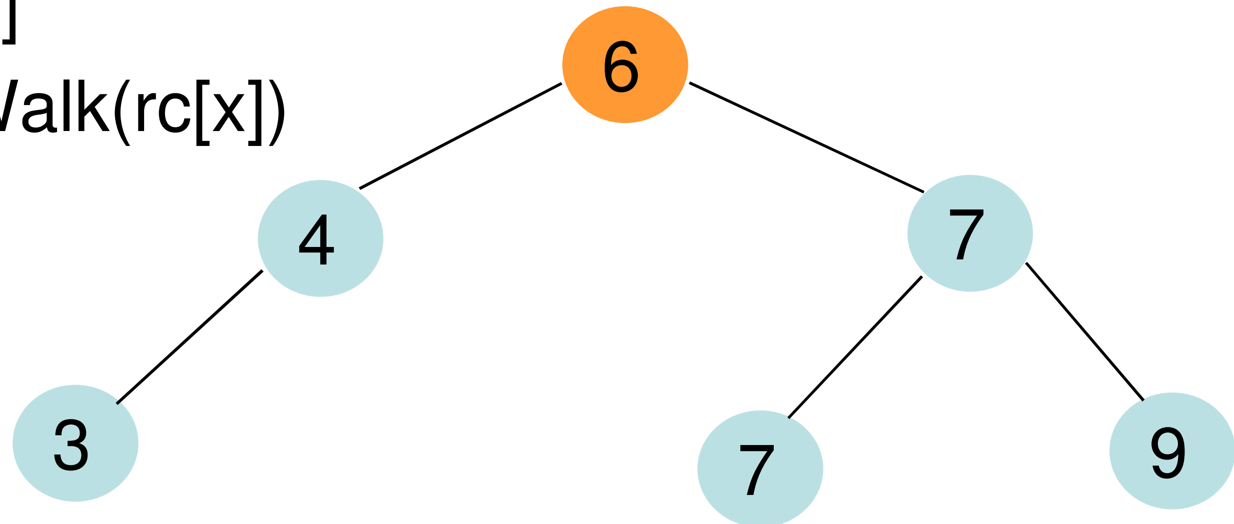
Aufruf über Inorder-Tree-Walk(root[T])



Binäre Suchbäume

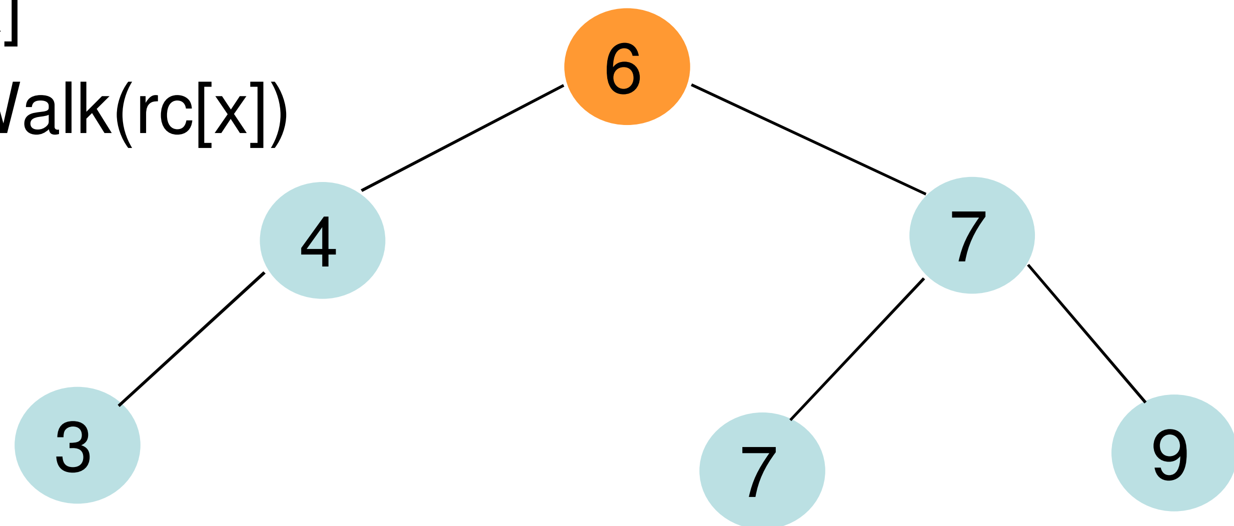
Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])



Binäre Suchbäume

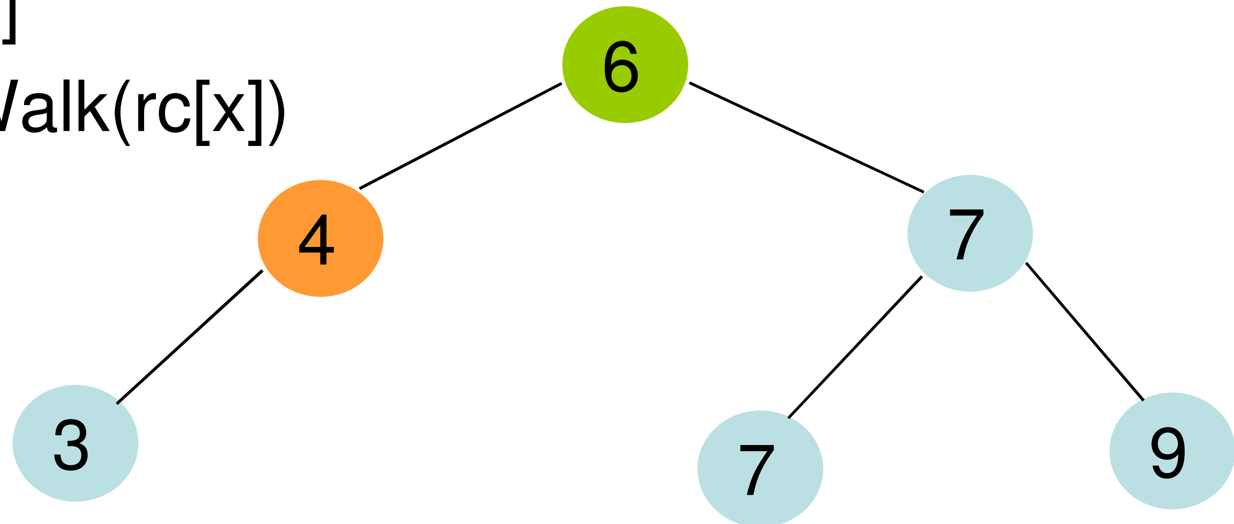
Inorder-Tree-Walk(x)
1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])



Binäre Suchbäume

Inorder-Tree-Walk(x)

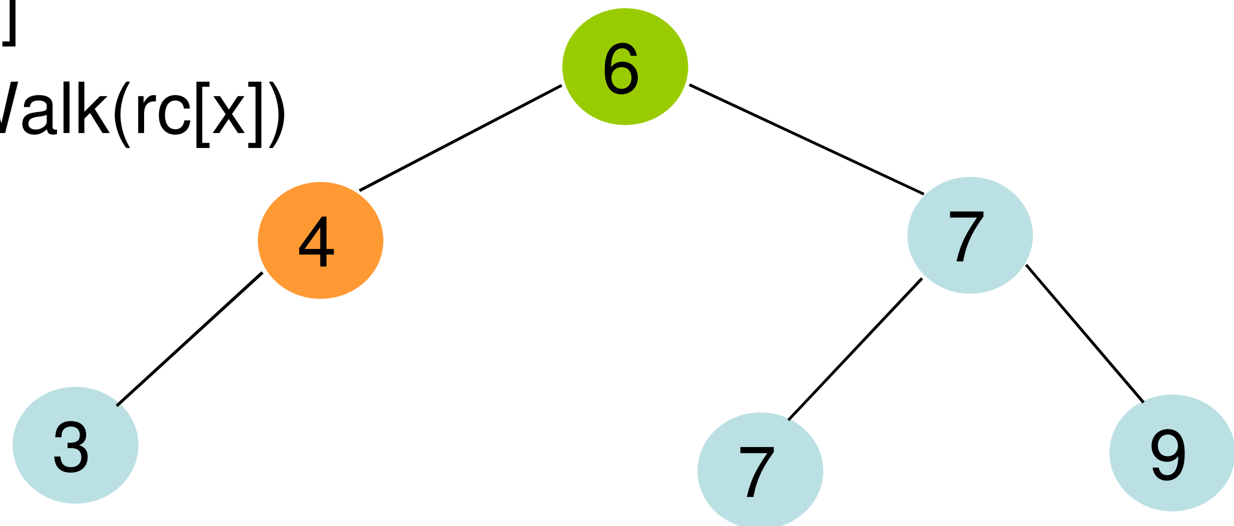
1. **if** x≠**nil** **then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])



Binäre Suchbäume

Inorder-Tree-Walk(x)

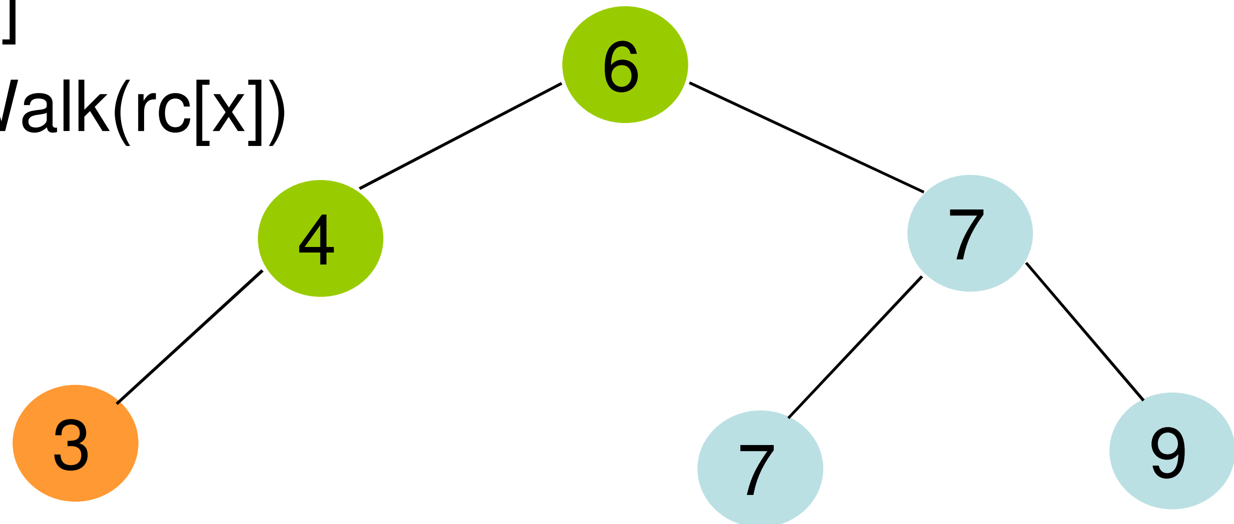
1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])



Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

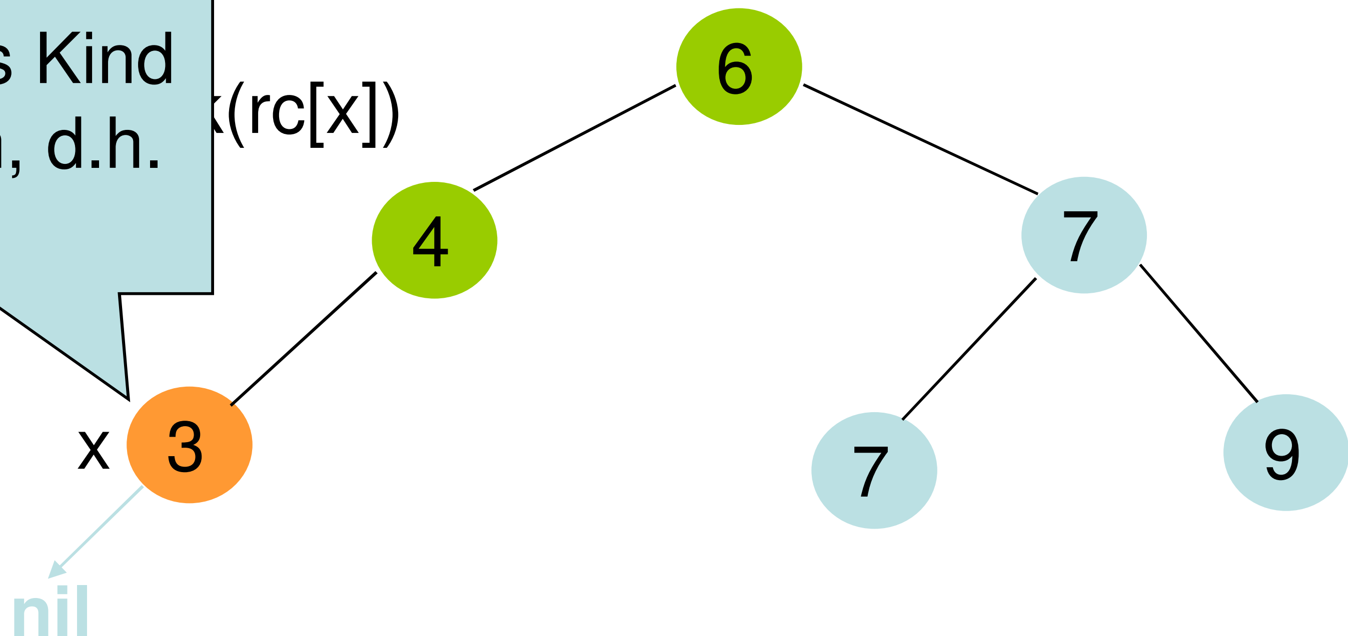


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3.
4. k(rc[x])

Kein linkes Kind vorhanden, d.h. lc[x]=**nil**

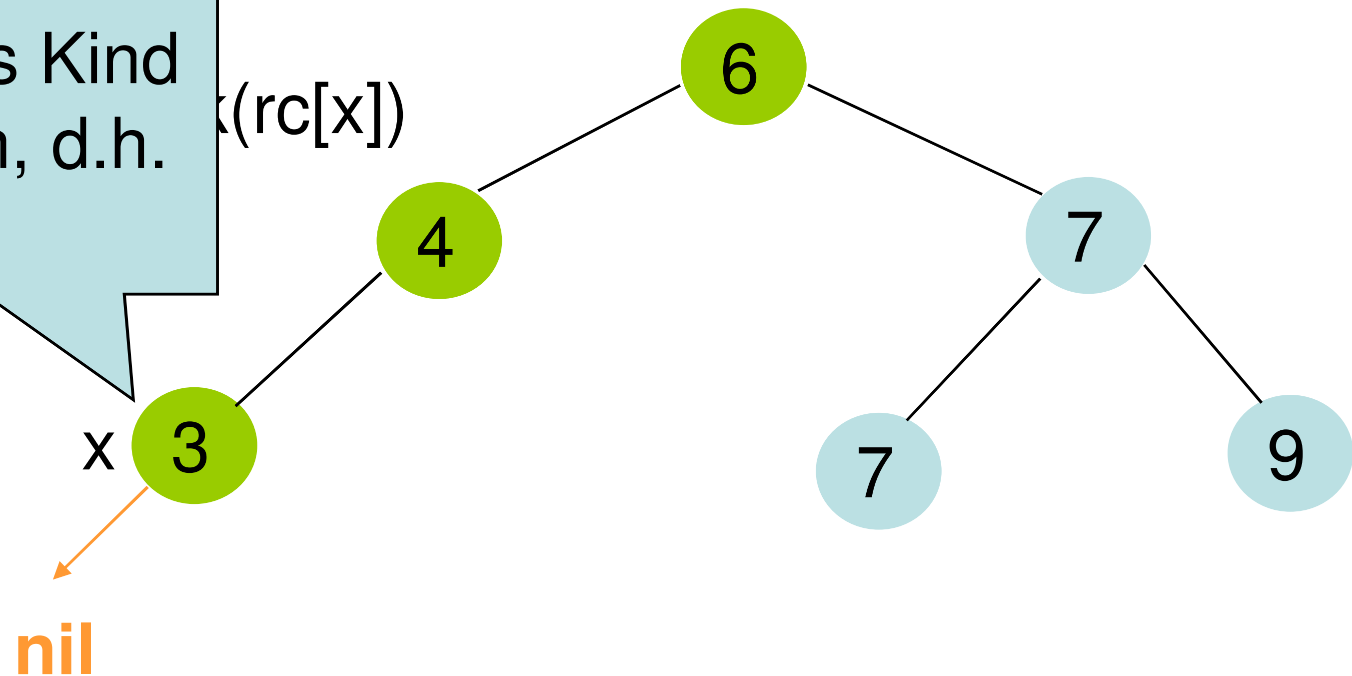


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3.
4. (rc[x])

Kein linkes Kind vorhanden, d.h. lc[x]=**nil**

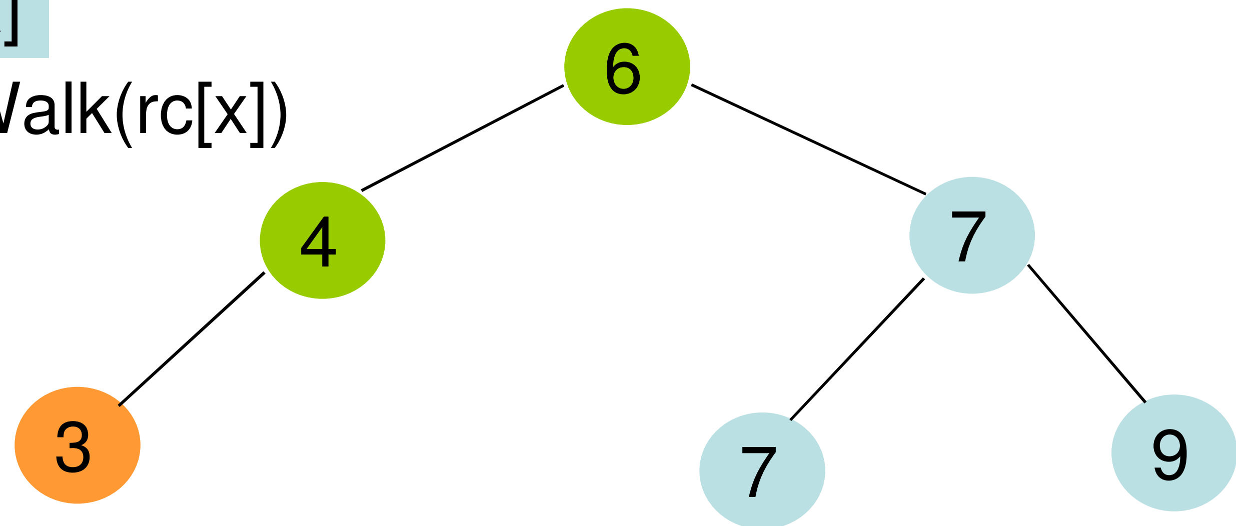


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3

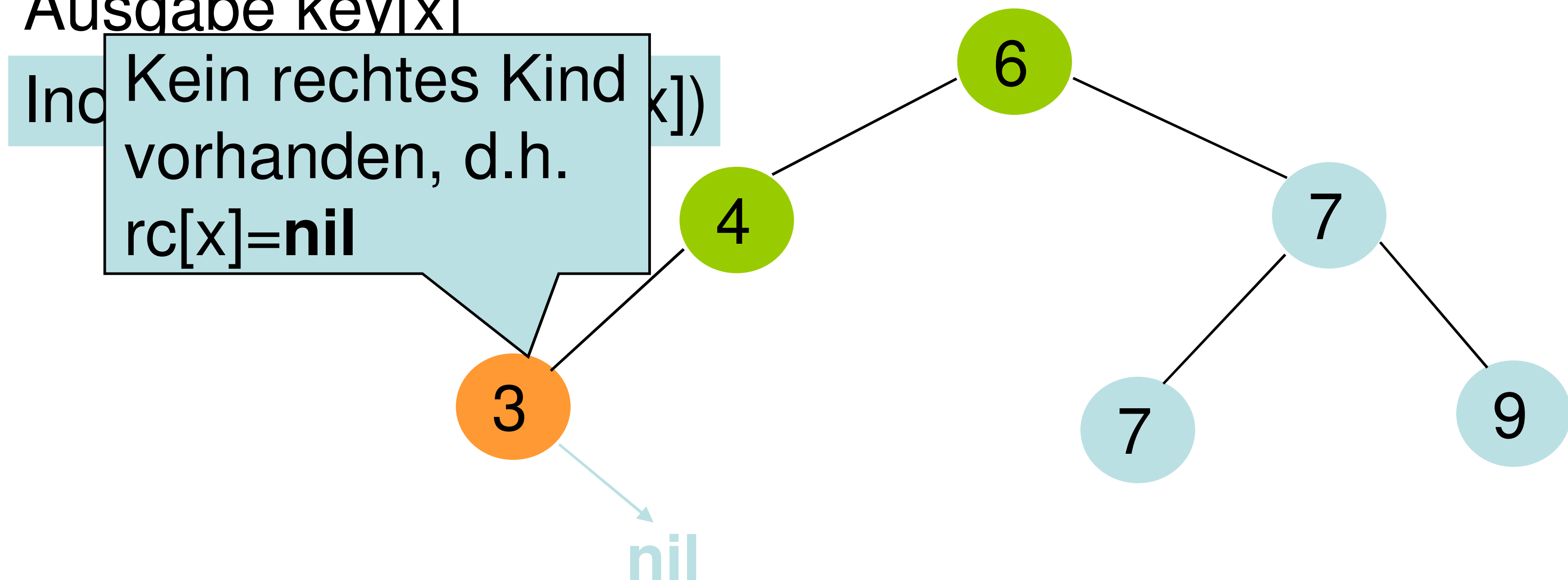


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4.

Ausgabe:
3

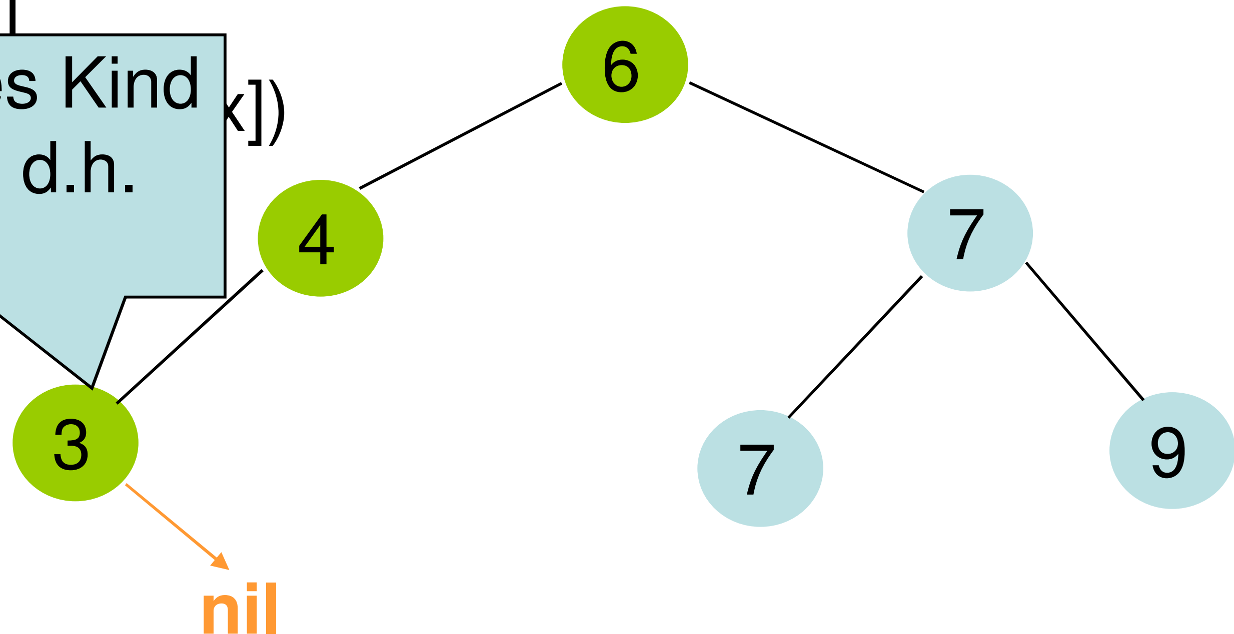


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2.   Inorder-Tree-Walk(lc[x])
3.   Ausgabe key[x]
4.   Inorder-Tree-Walk(rc[x])

Ausgabe:
3

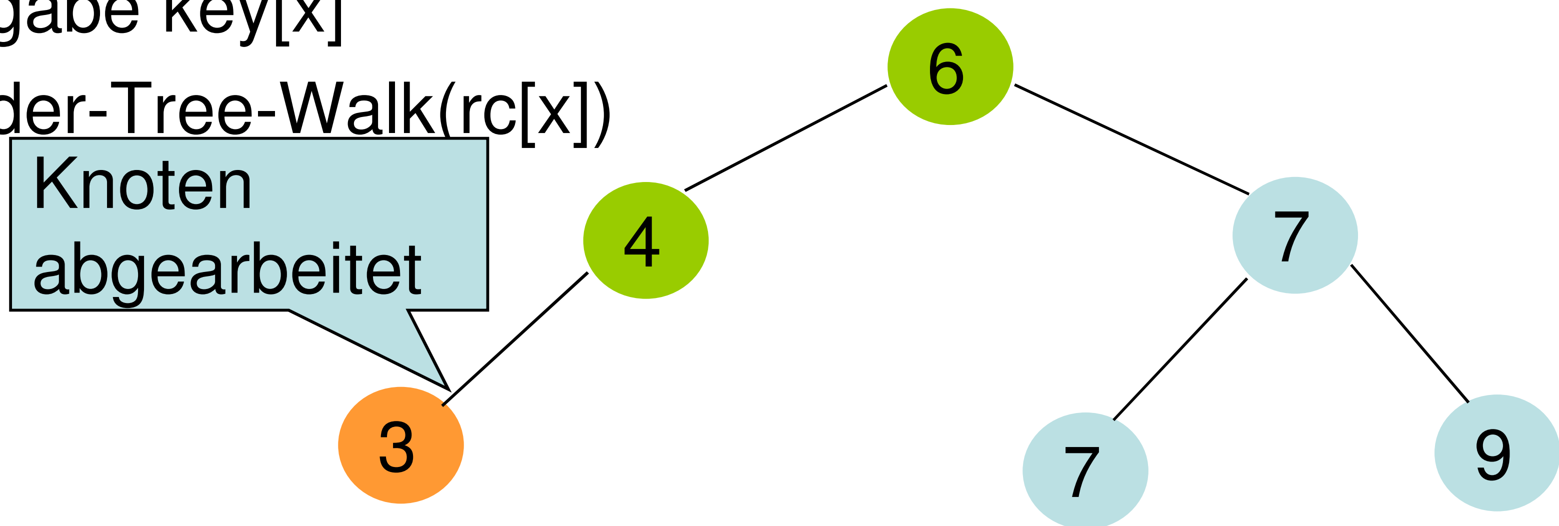


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3

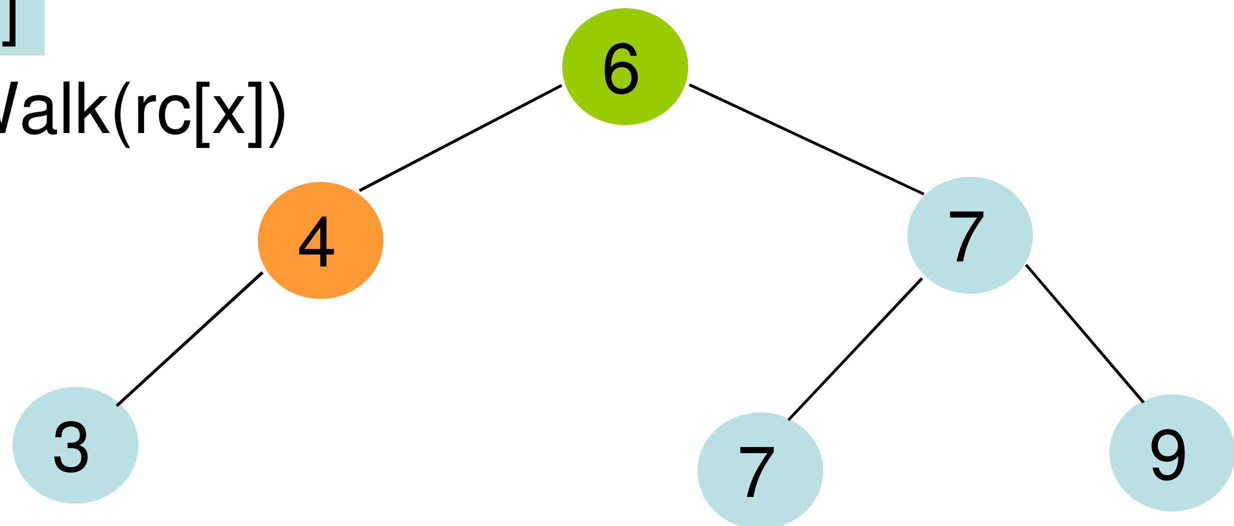


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4

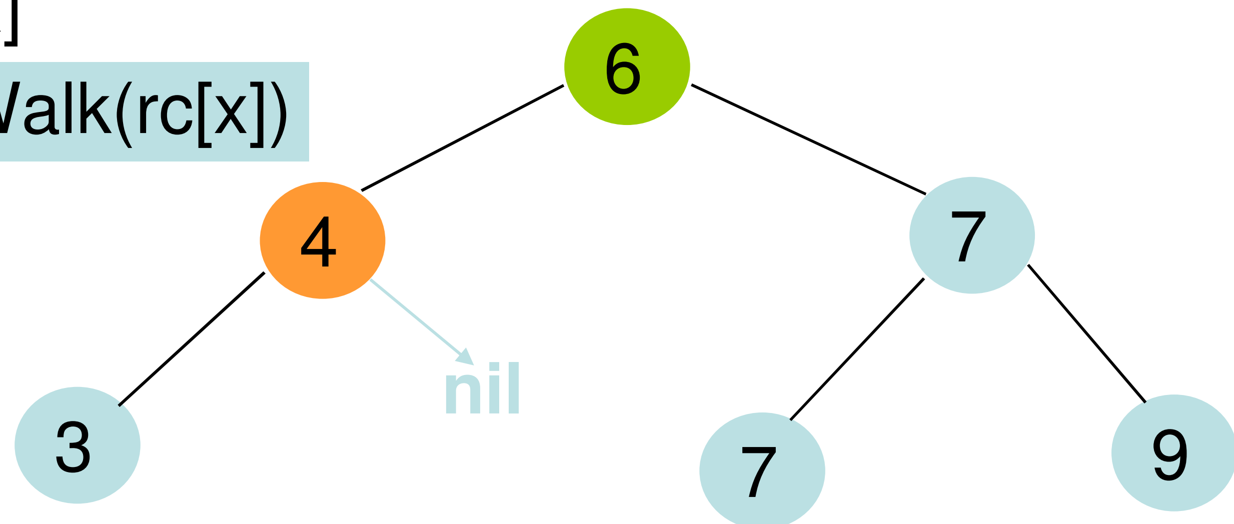


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2.   Inorder-Tree-Walk(lc[x])
3.   Ausgabe key[x]
4.   Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4

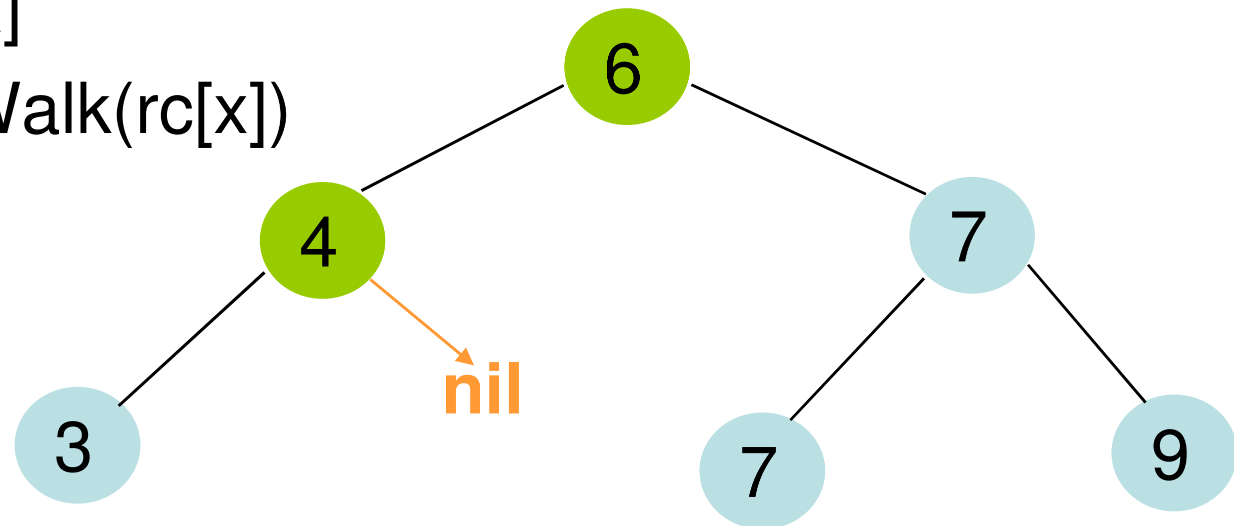


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4

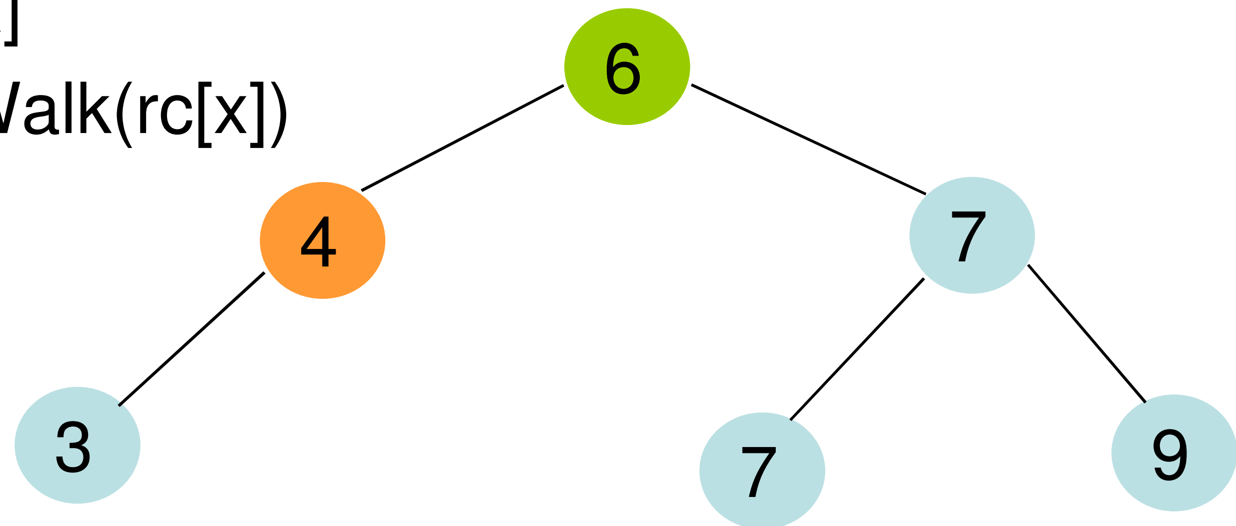


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2.   Inorder-Tree-Walk(lc[x])
3.   Ausgabe key[x]
4.   Inorder-Tree-Walk(rc[x])

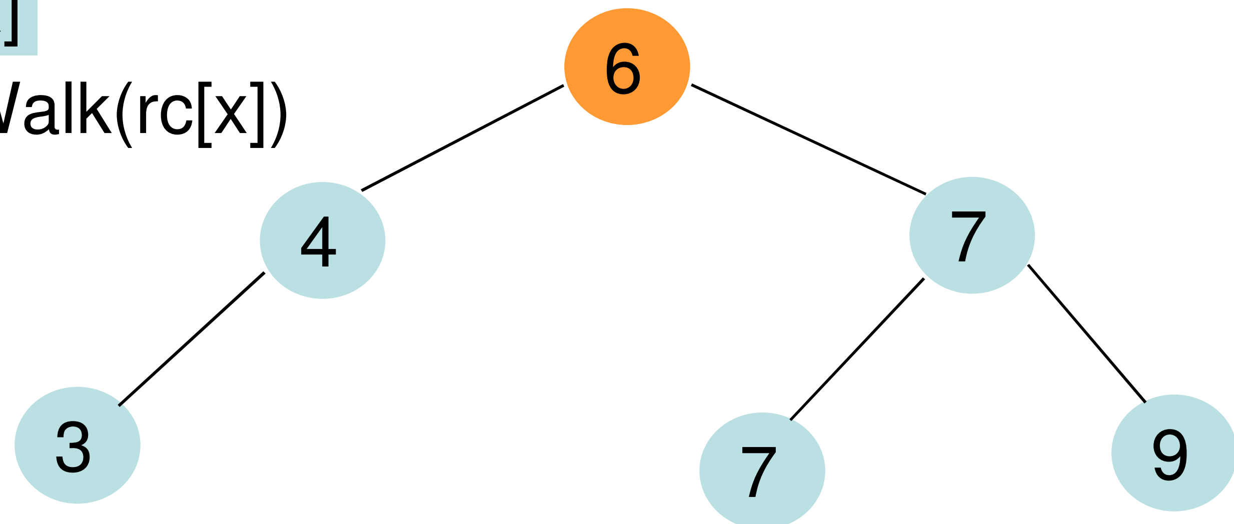
Ausgabe:
3, 4



Binäre Suchbäume

Inorder-Tree-Walk(x)
1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6

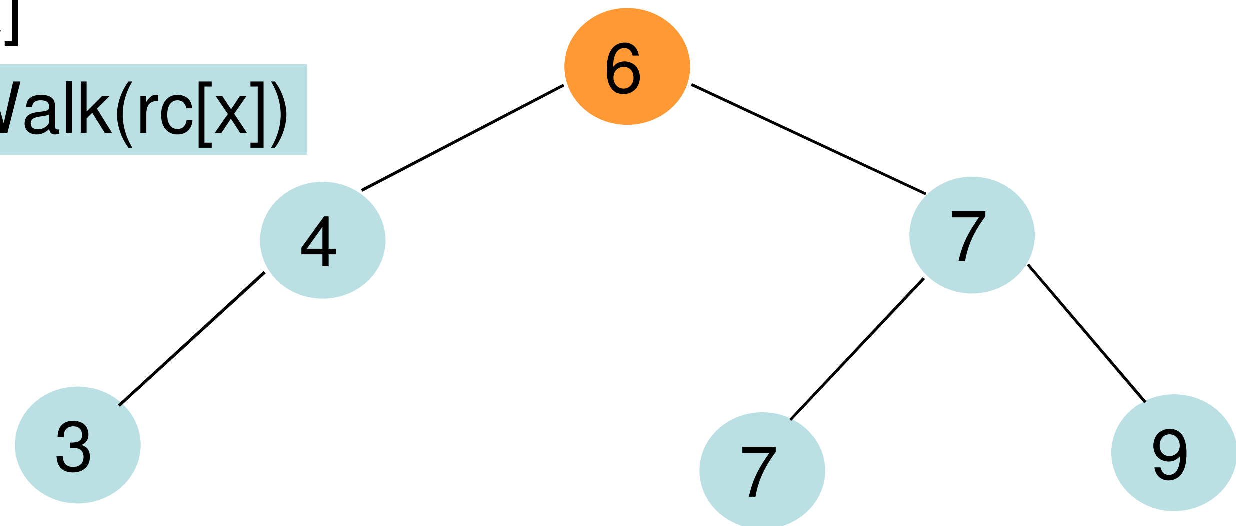


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6

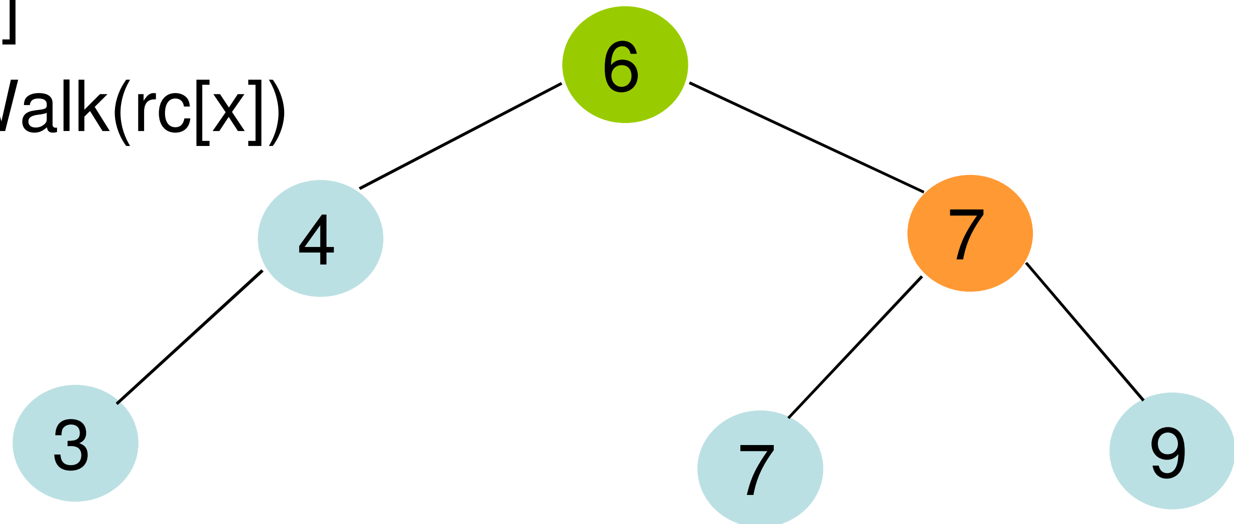


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2.   Inorder-Tree-Walk(lc[x])
3.   Ausgabe key[x]
4.   Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6

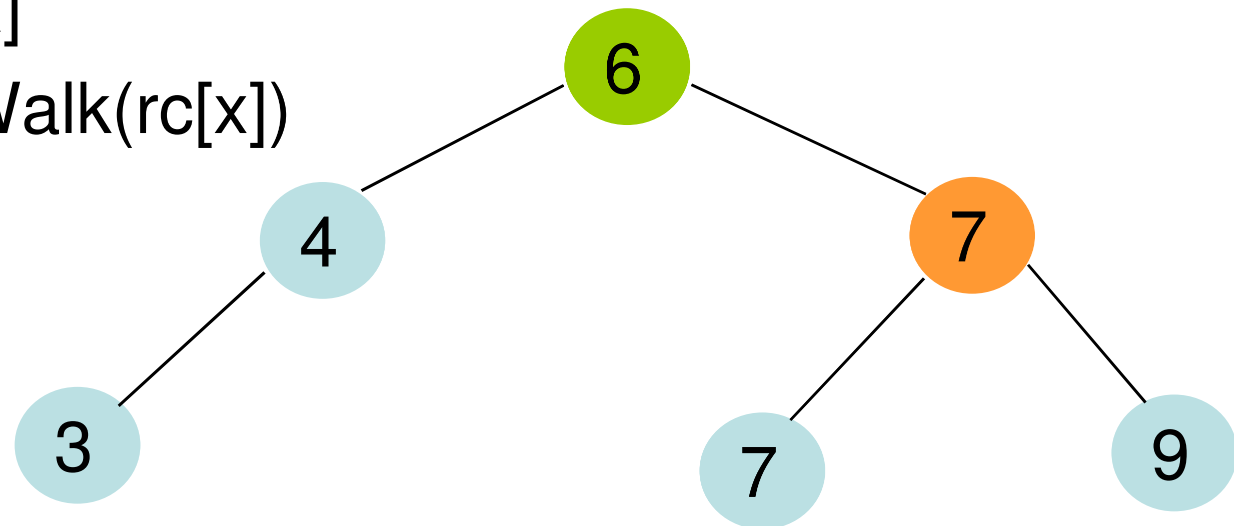


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6

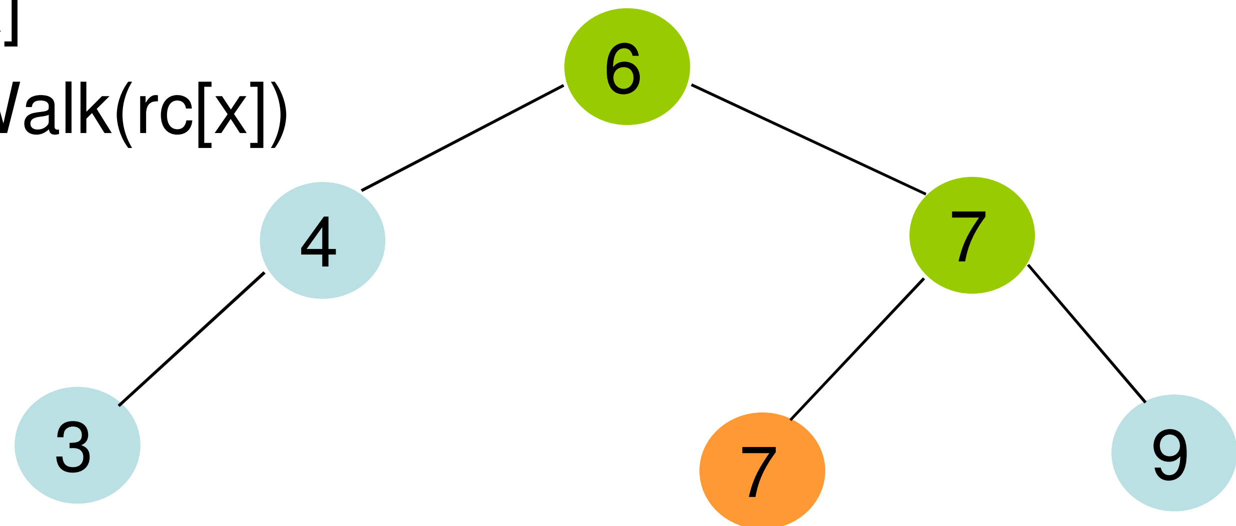


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6

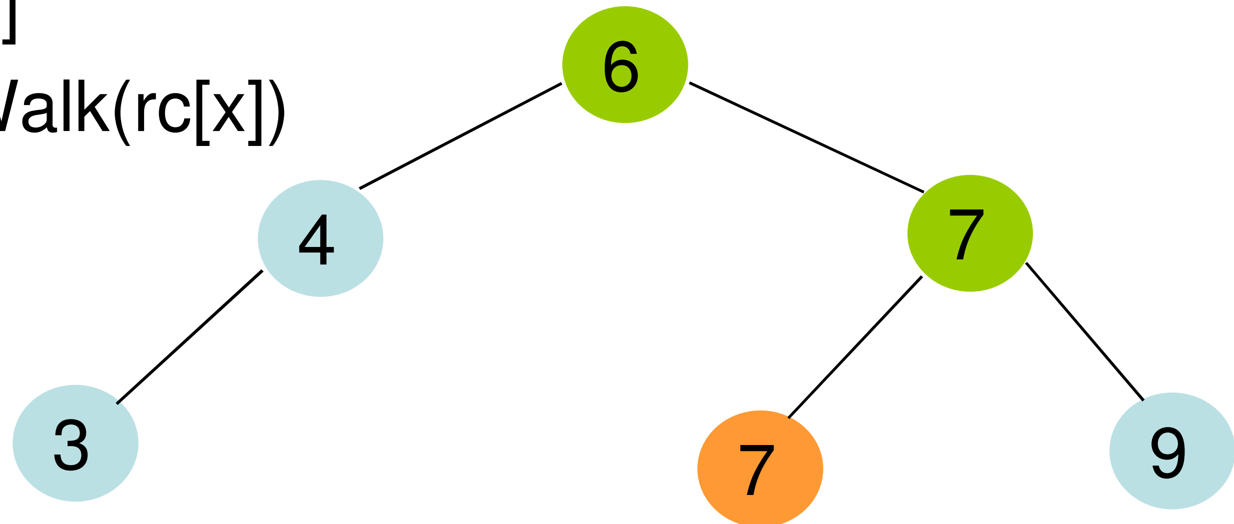


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6

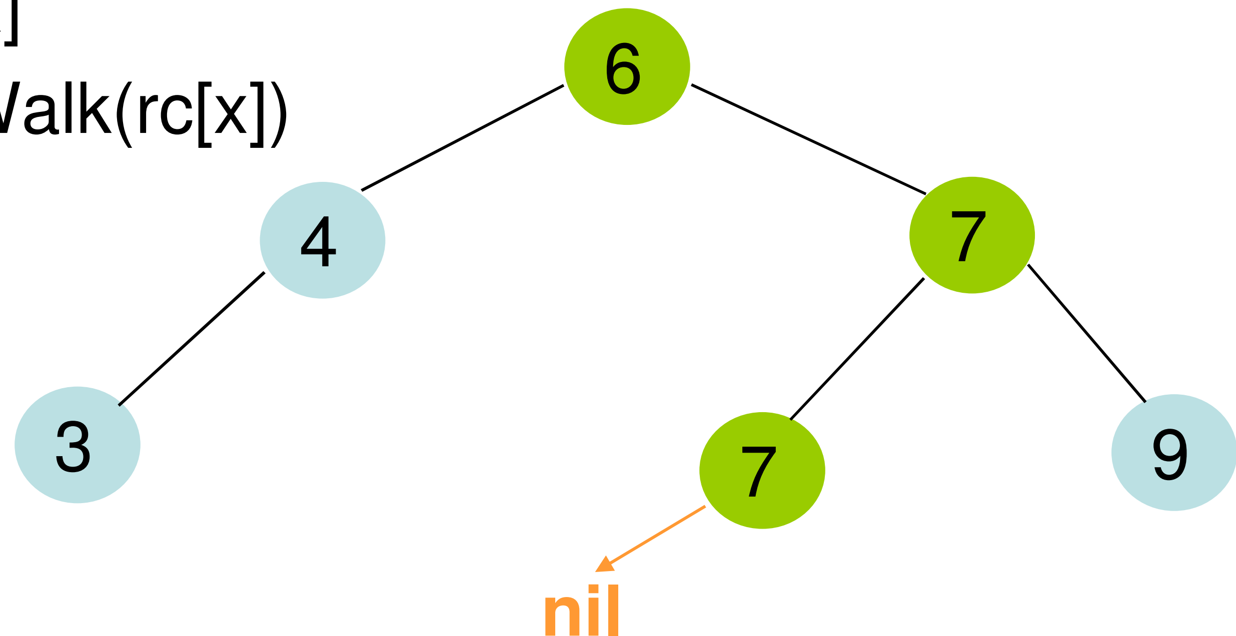


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6

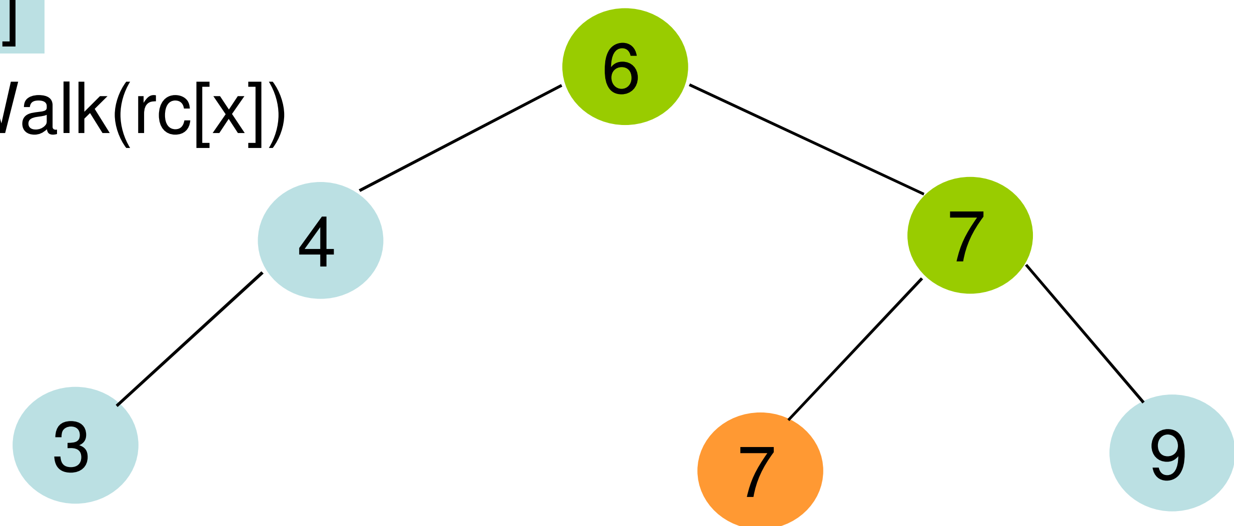


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6, 7

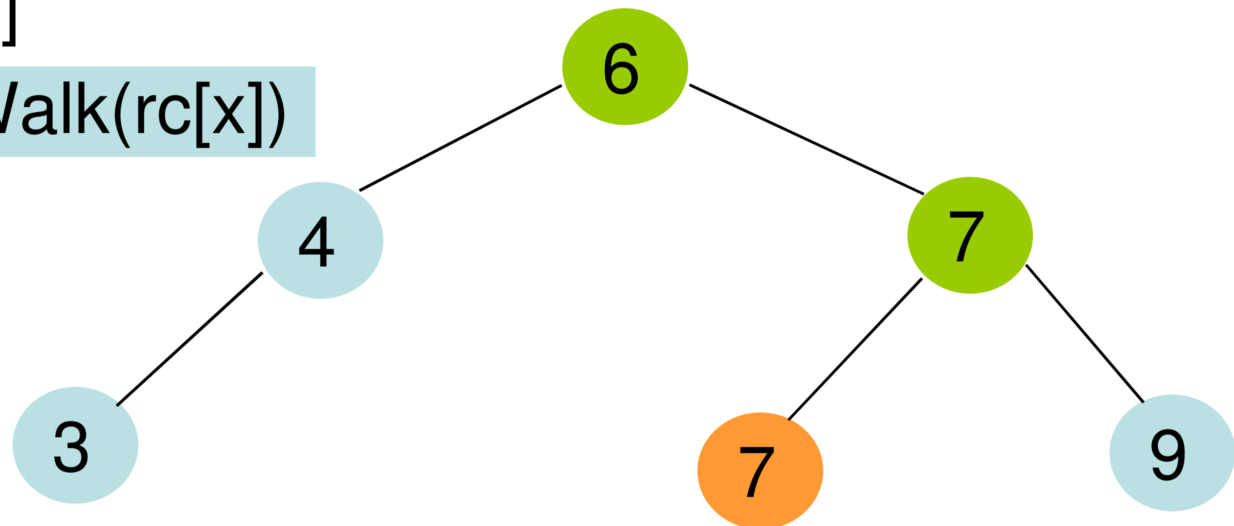


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6, 7

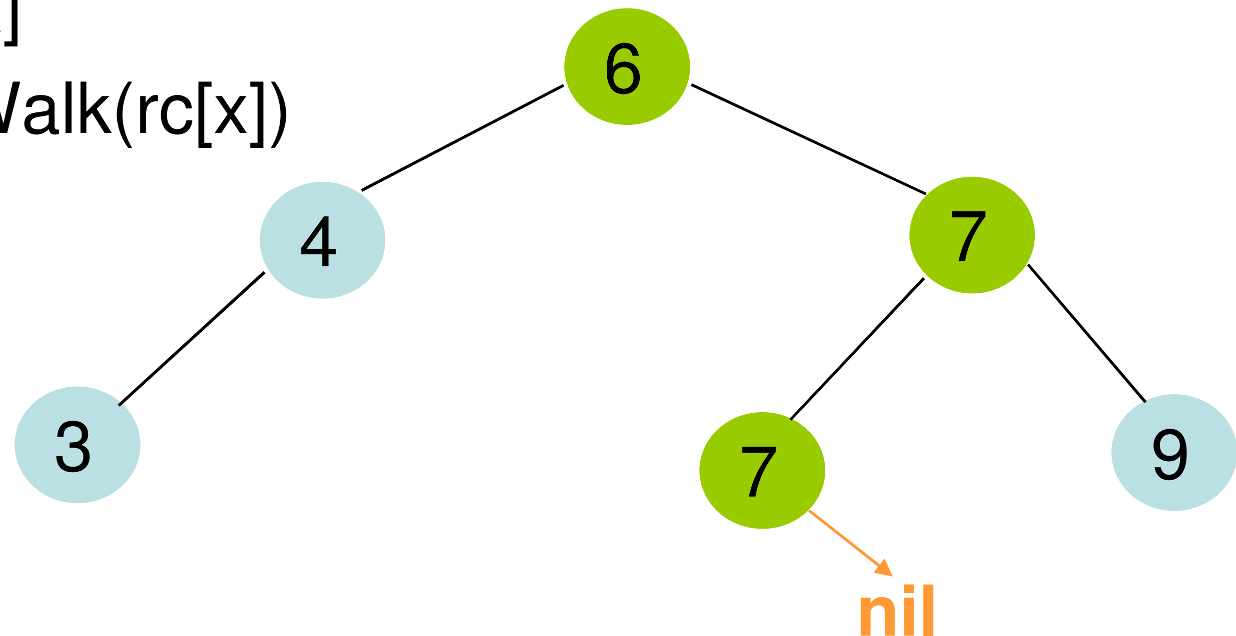


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6

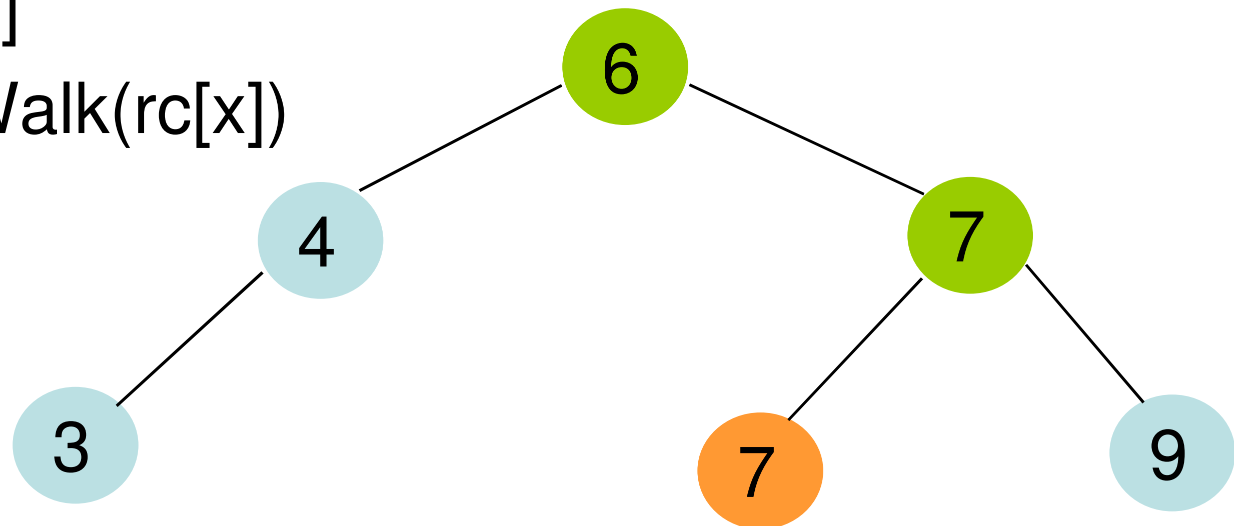


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

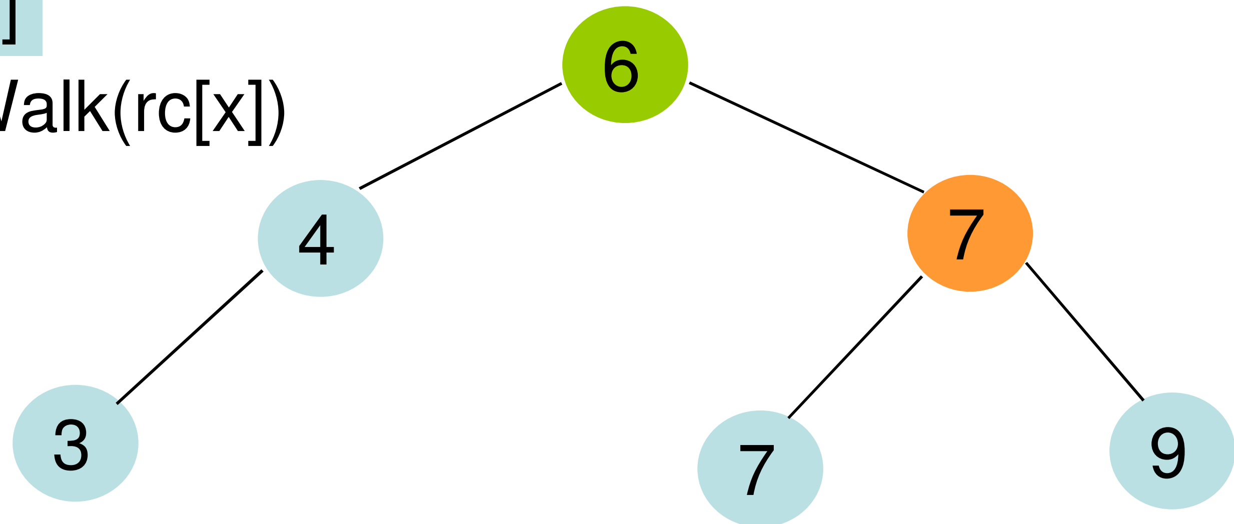
Ausgabe:
3, 4, 6, 7



Binäre Suchbäume

Inorder-Tree-Walk(x)
1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6, 7, 7

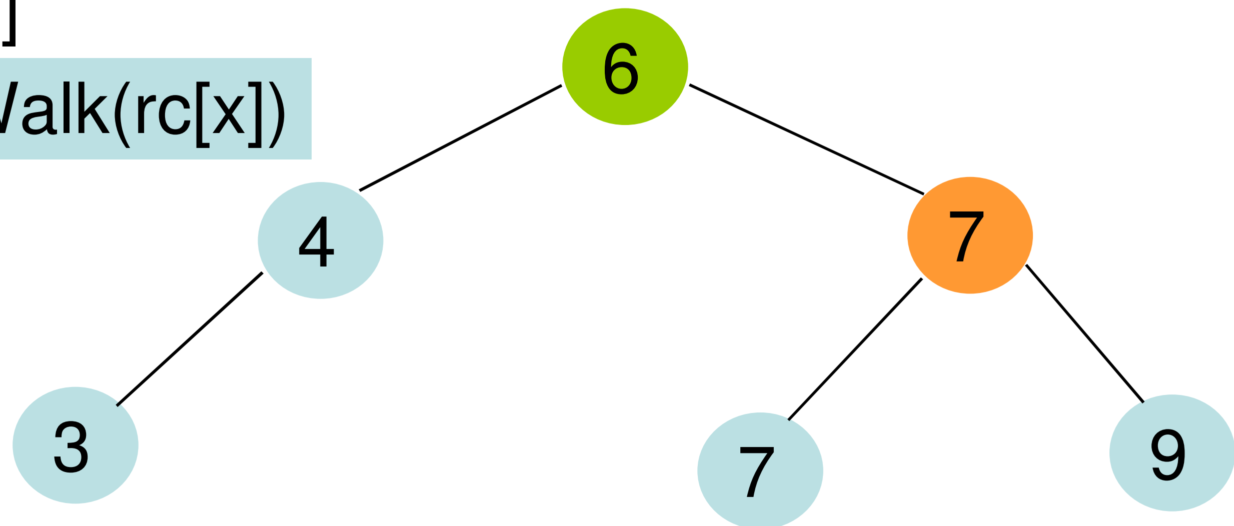


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6, 7, 7

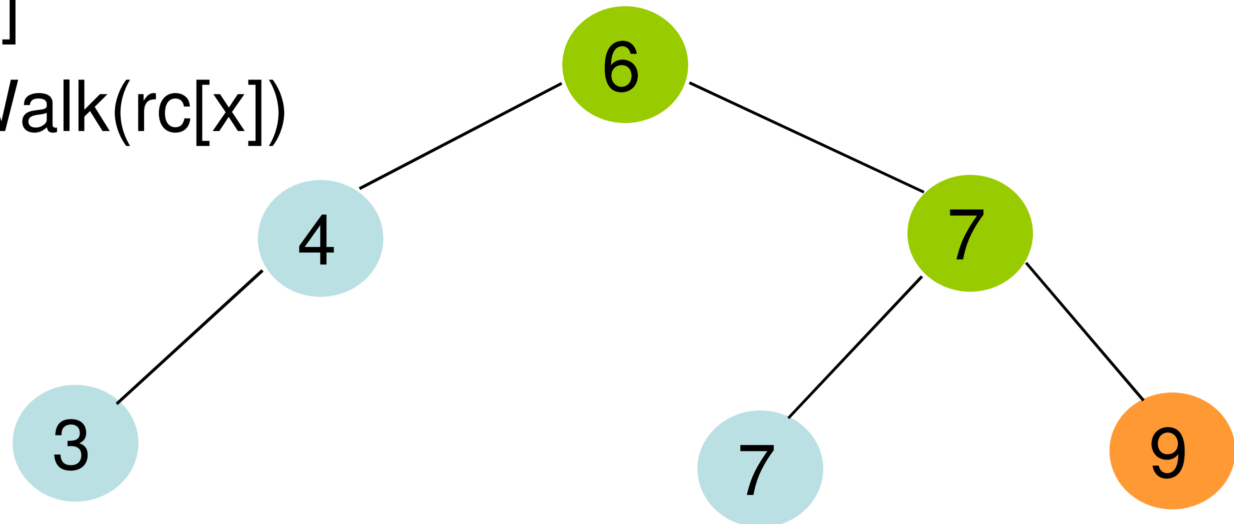


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6, 7, 7

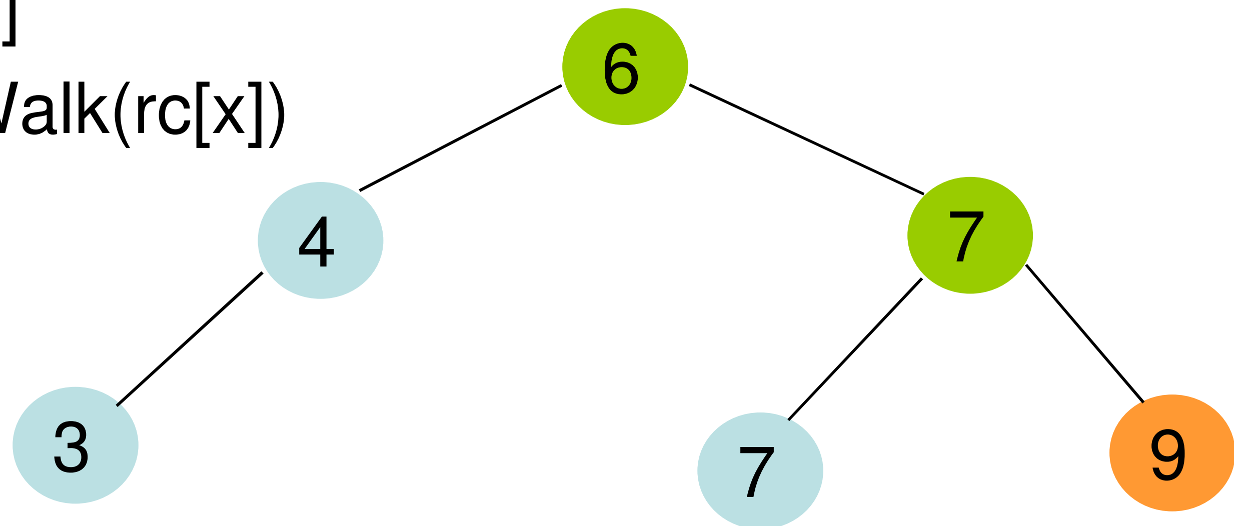


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6, 7, 7

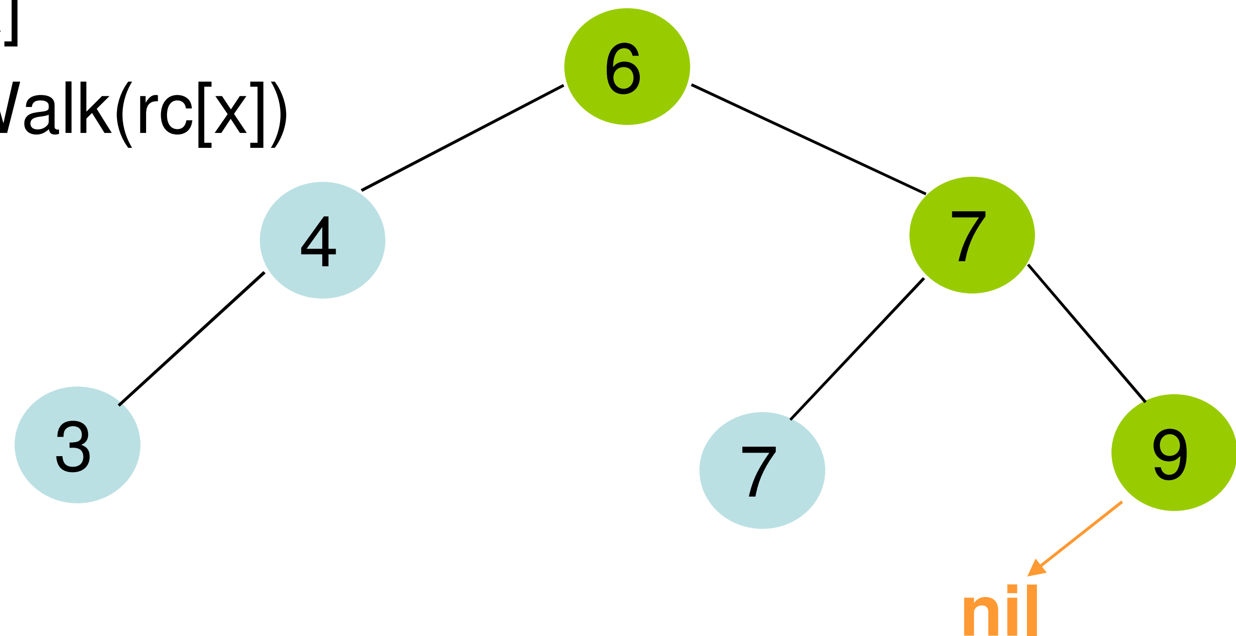


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6, 7, 7

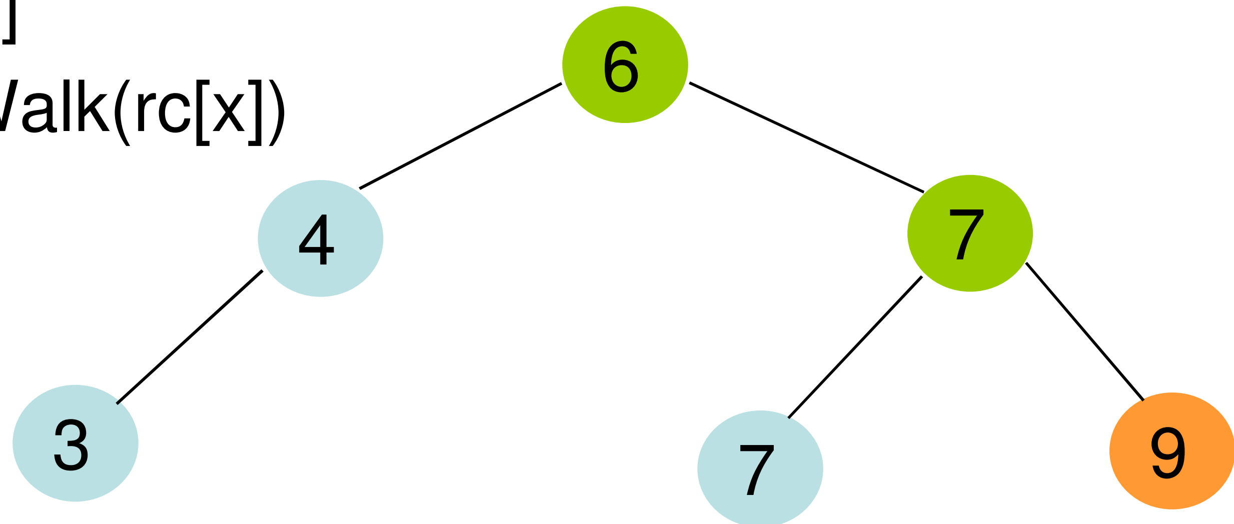


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6, 7, 7

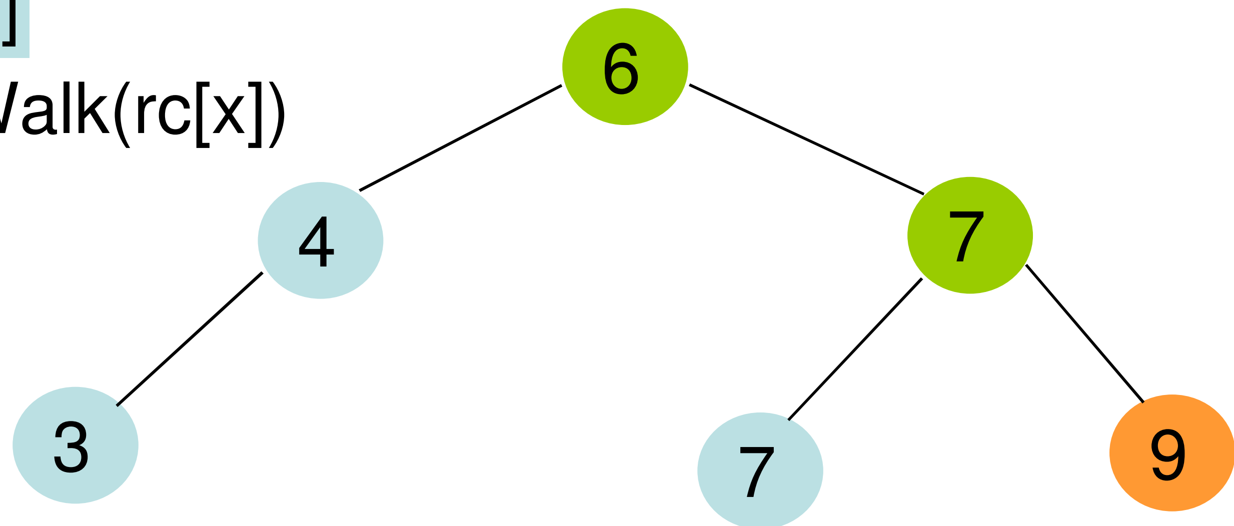


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6, 7, 7, 9

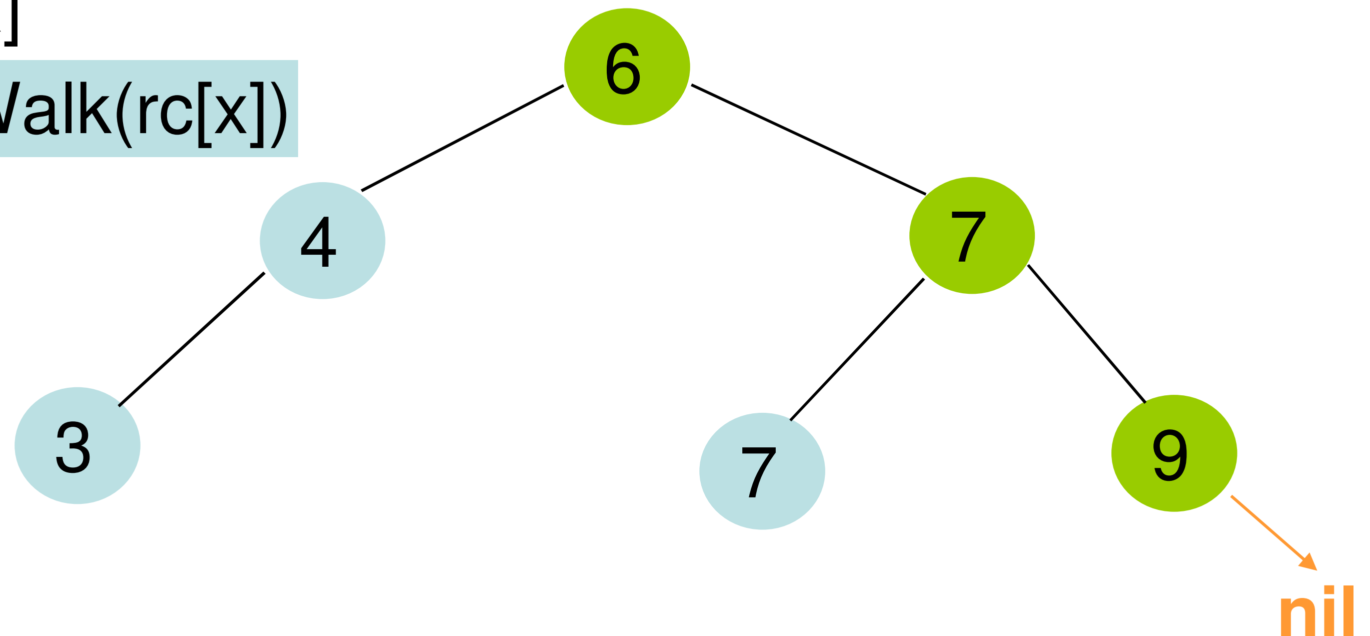


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6, 7, 7, 9

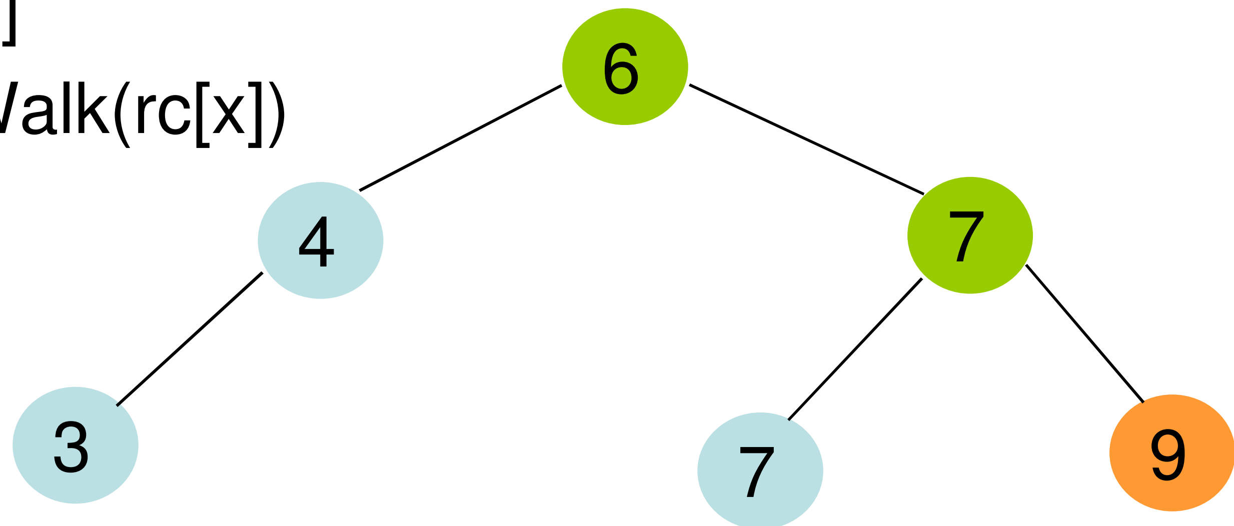


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6, 7, 7, 9

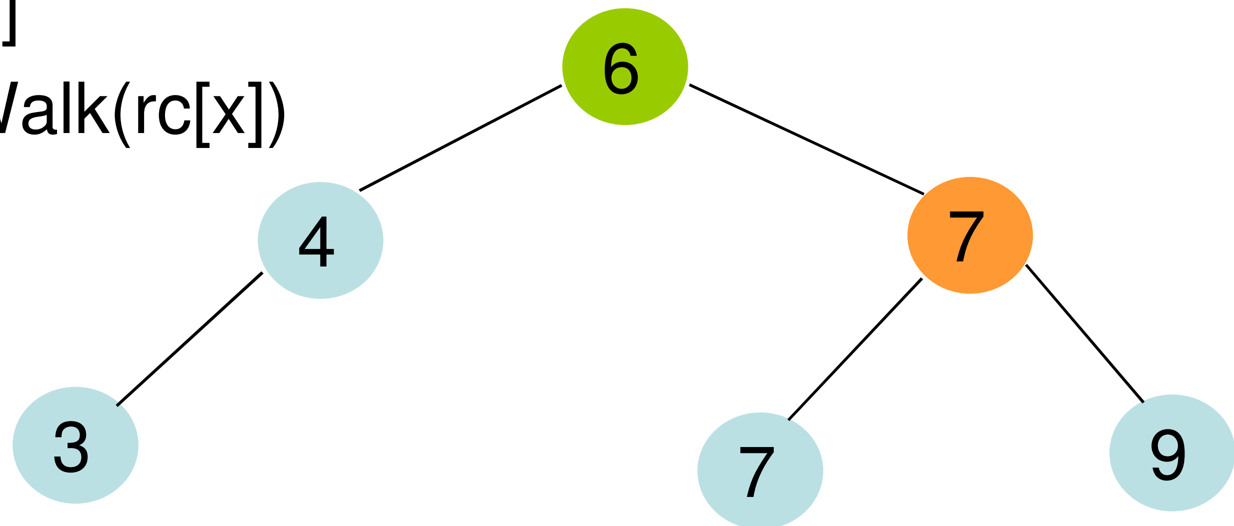


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6, 7, 7, 9

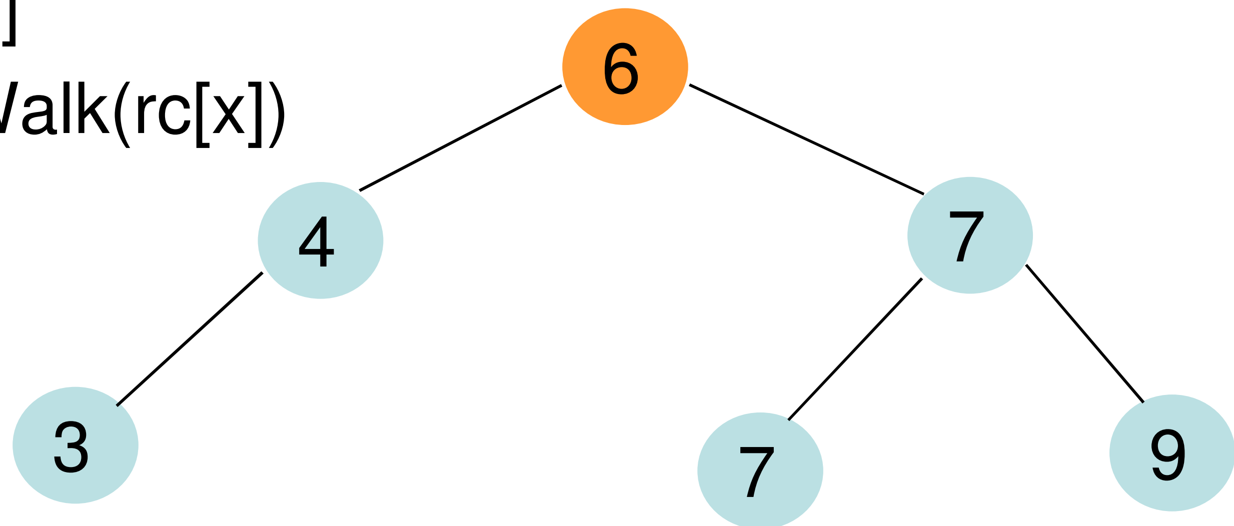


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6, 7, 7, 9

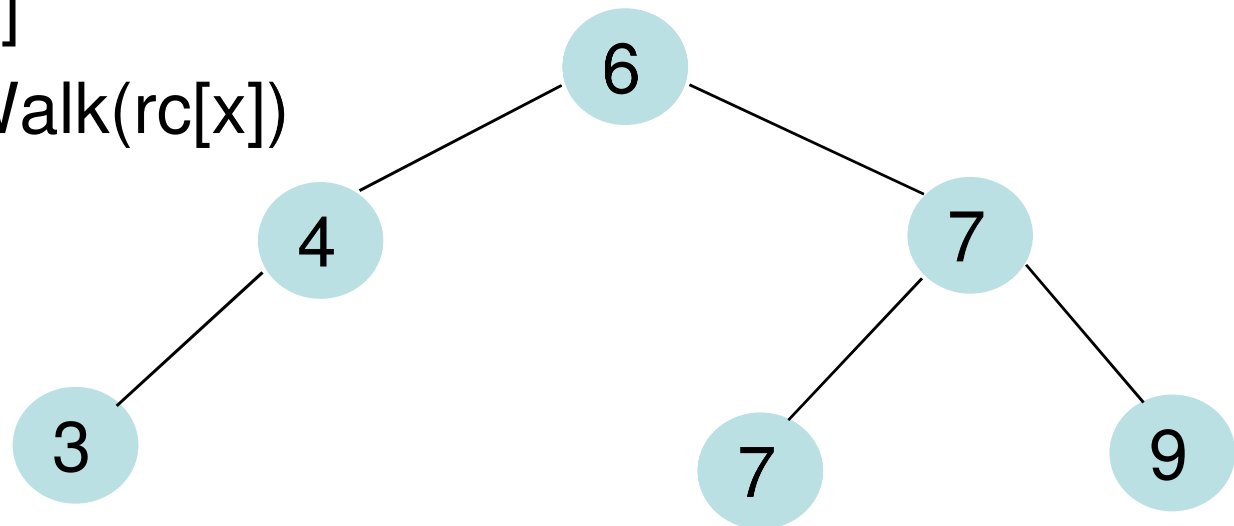


Binäre Suchbäume

Inorder-Tree-Walk(x)

1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Ausgabe:
3, 4, 6, 7, 7, 9



Binäre Suchbäume

Inorder-Tree-Walk(x)
1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Laufzeit $\Theta(n)$

Binäre Suchbäume

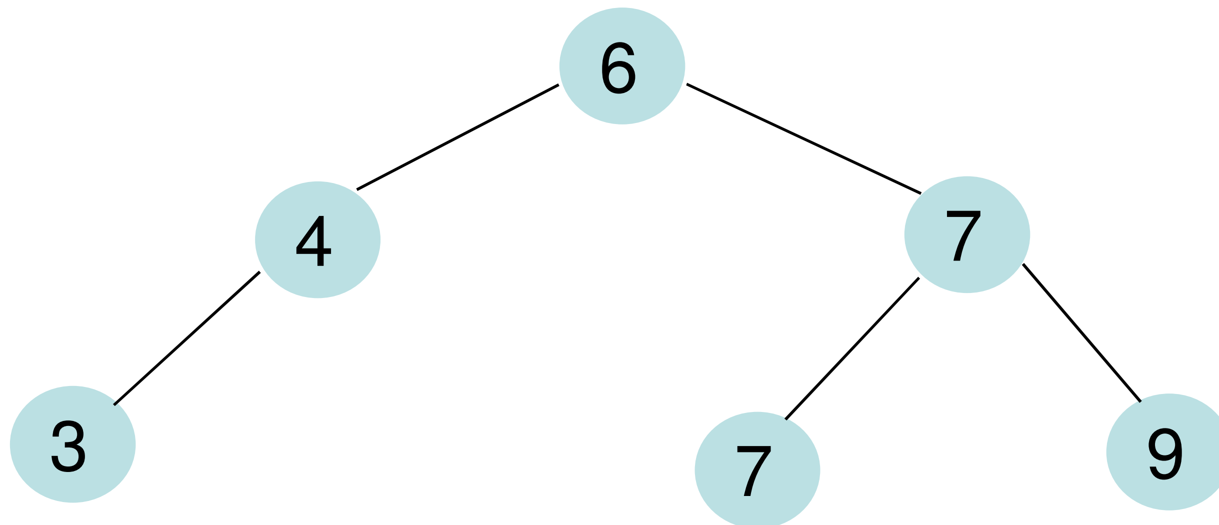
Suchen in Binärbäumen

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

Binäre Suchbäume

Baumsuche(x,k)

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

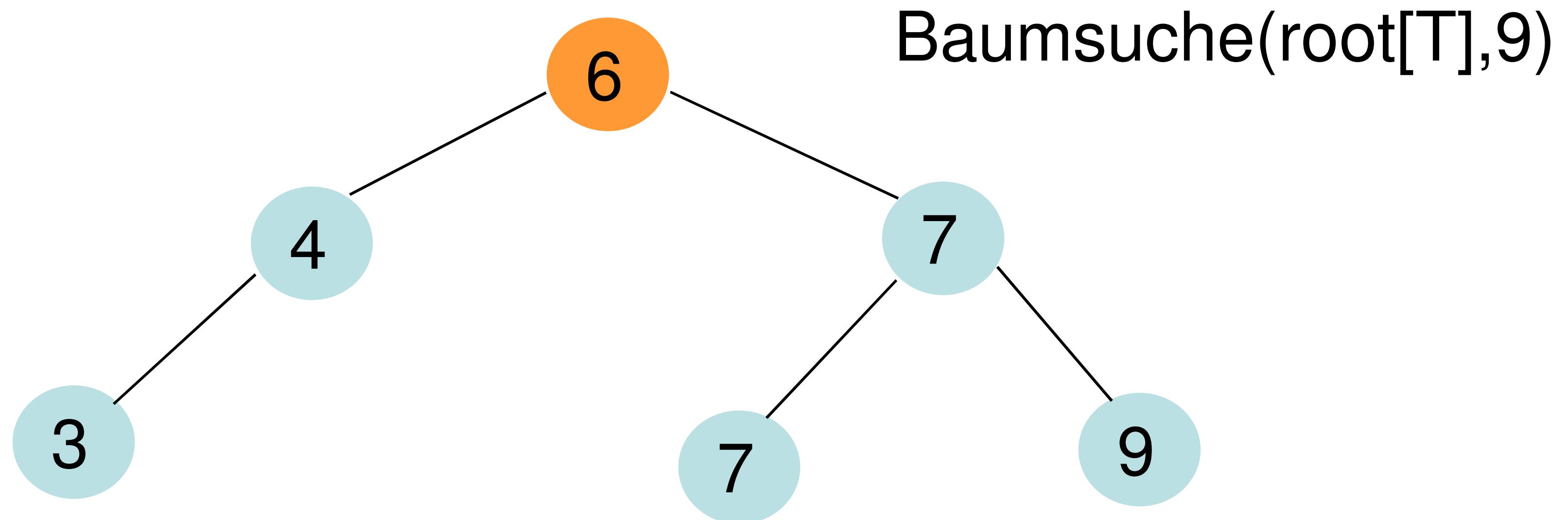


Binäre Suchbäume

Baumsuche(x,k)

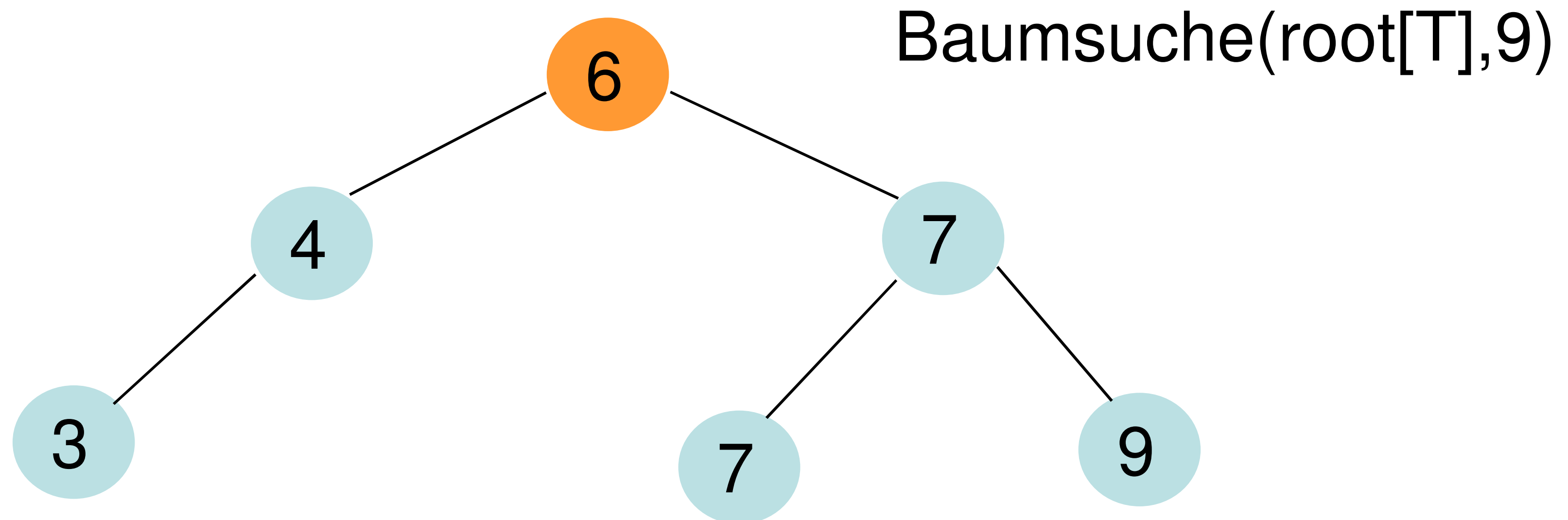
Aufruf mit x=root[T]

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



Baumsuche(x,k)

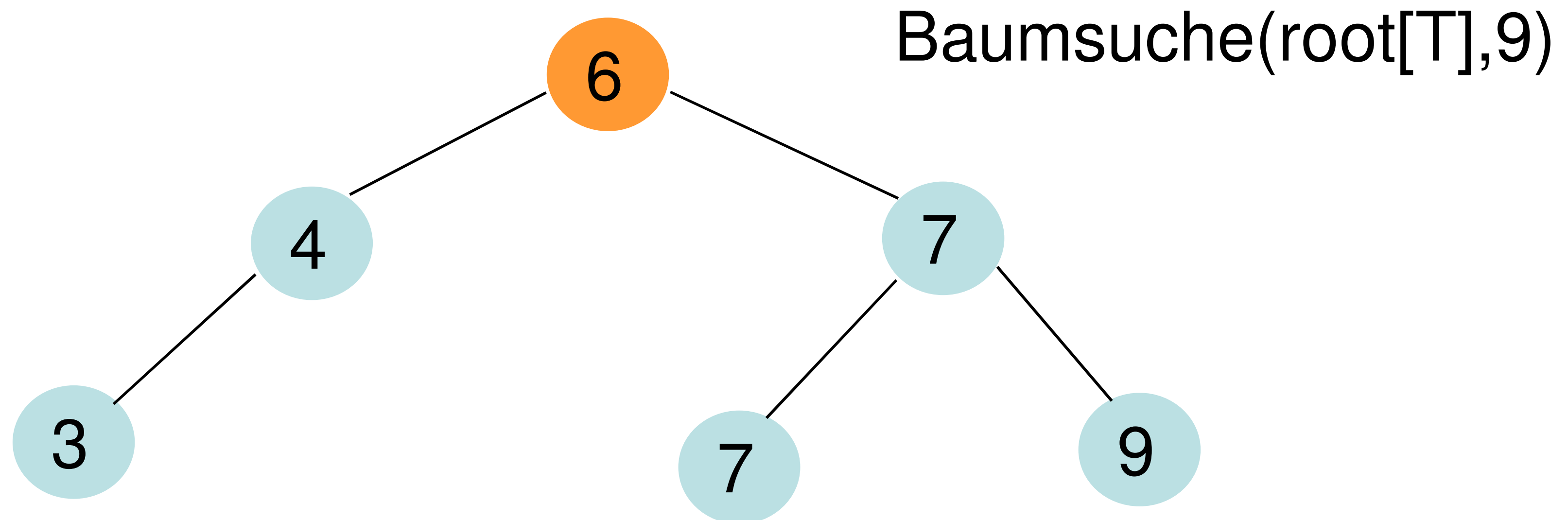
1. **if** x=**nil or** k=key[x] **then return** x
2. **if** k<key[x] **then return** Baumsuche(lc[x],k)
3. **else** return Baumsuche(rc[x],k)



Binäre Suchbäume

Baumsuche(x,k)

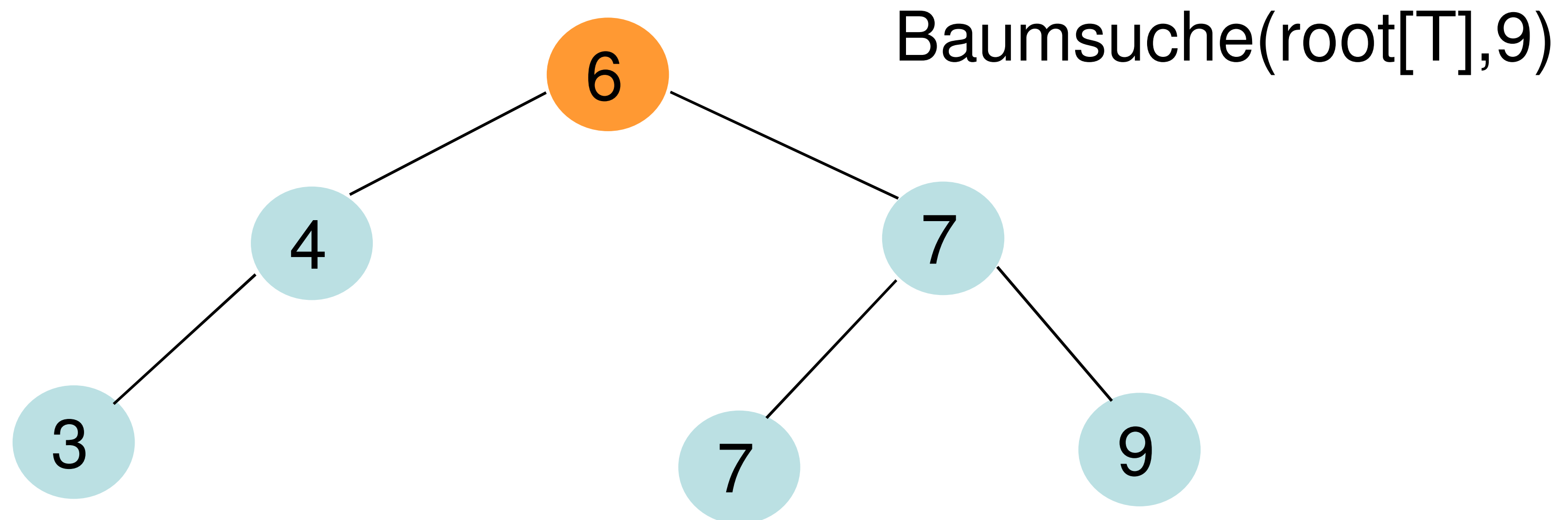
1. **if** x=**nil or** k=key[x] **then return** x
2. **if** k<key[x] **then return** Baumsuche(lc[x],k)
3. **else** return Baumsuche(rc[x],k)



Binäre Suchbäume

Baumsuche(x,k)

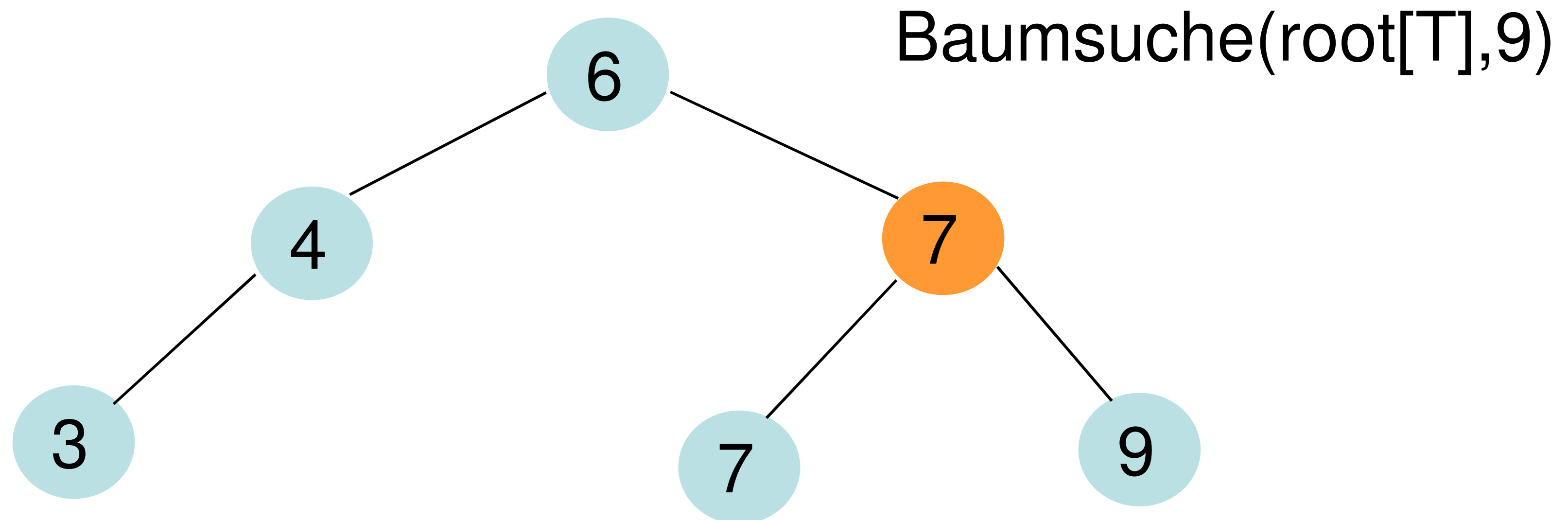
1. **if** x=**nil or** k=key[x] **then return** x
2. **if** k<key[x] **then return** Baumsuche(lc[x],k)
3. **else** return Baumsuche(rc[x],k)



Binäre Suchbäume

Baumsuche(x,k)

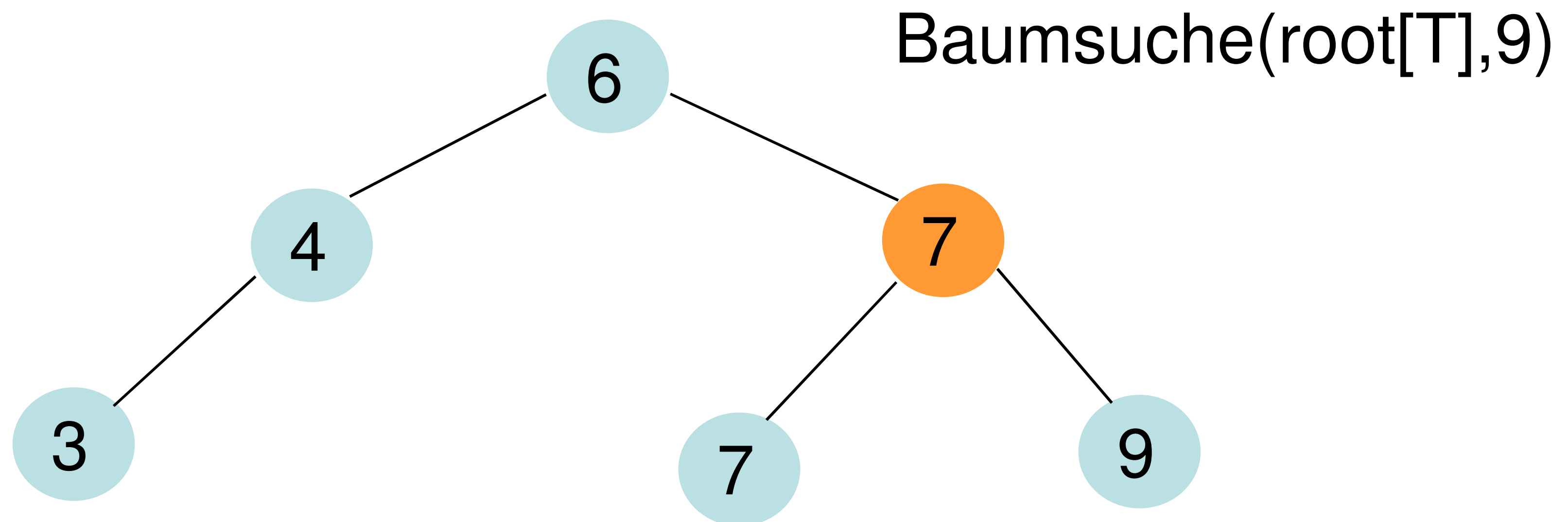
1. **if** x=**nil or** k=key[x] **then return** x
2. **if** k<key[x] **then return** Baumsuche(lc[x],k)
3. **else** return Baumsuche(rc[x],k)



Binäre Suchbäume

Baumsuche(x,k)

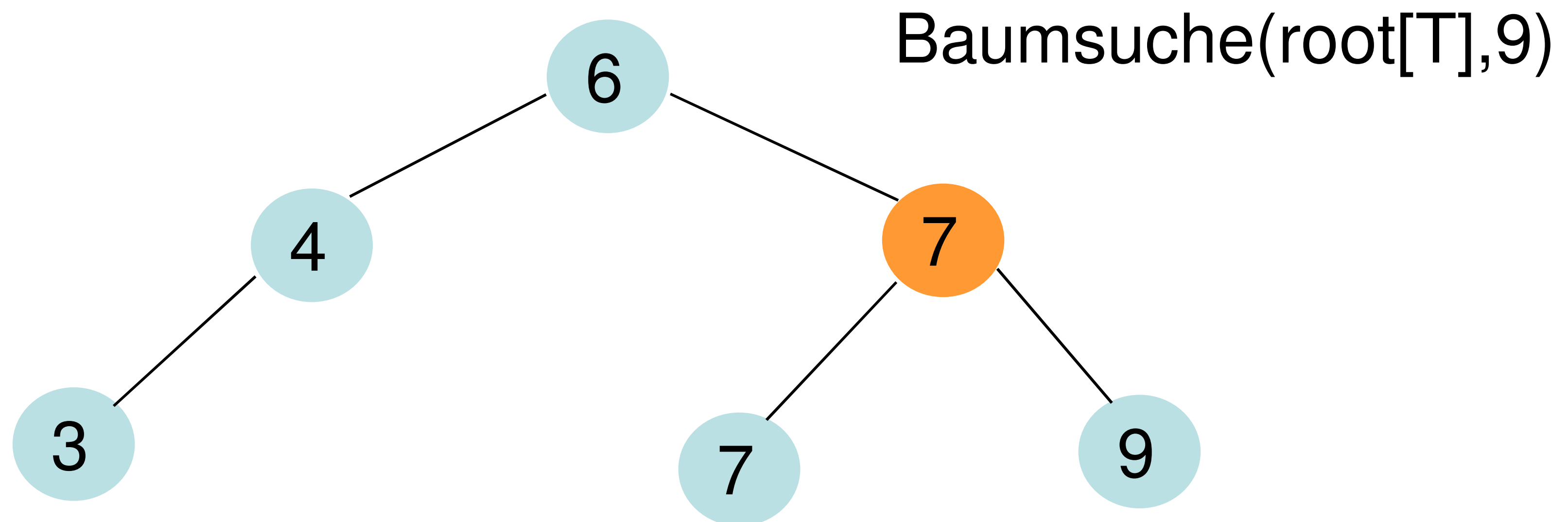
1. **if** x=**nil or** k=key[x] **then return** x
2. **if** k<key[x] **then return** Baumsuche(lc[x],k)
3. **else** return Baumsuche(rc[x],k)



Binäre Suchbäume

Baumsuche(x,k)

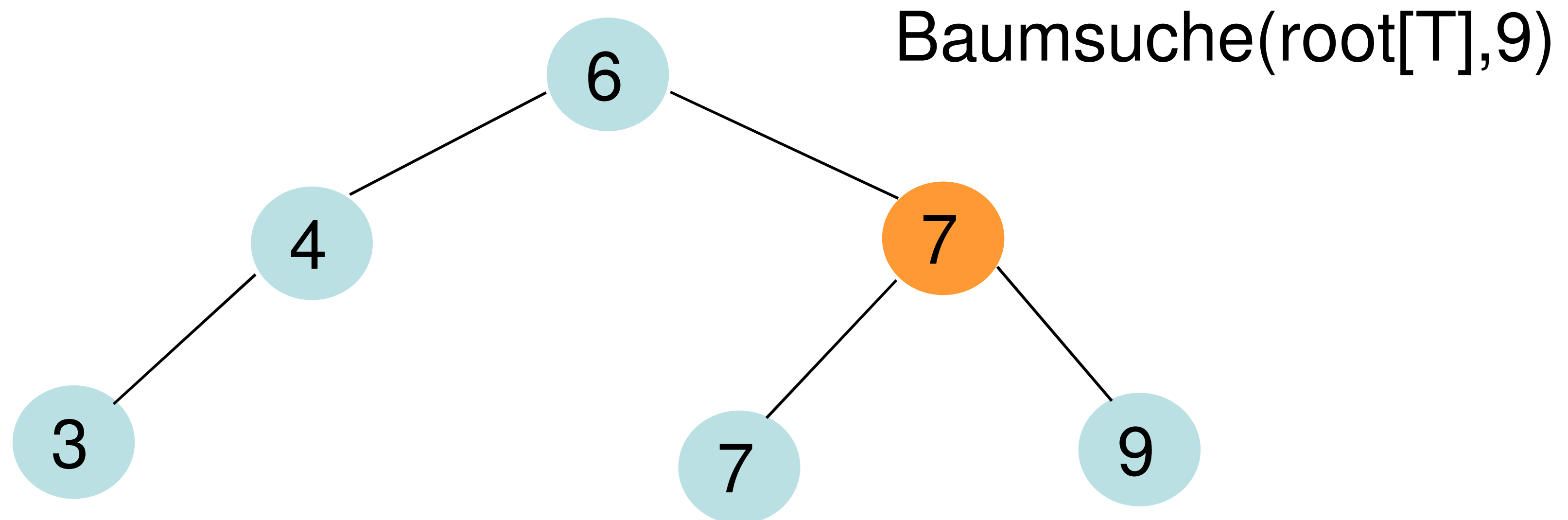
1. **if** x=**nil or** k=key[x] **then return** x
2. **if** k<key[x] **then return** Baumsuche(lc[x],k)
3. **else** return Baumsuche(rc[x],k)



Binäre Suchbäume

Baumsuche(x,k)

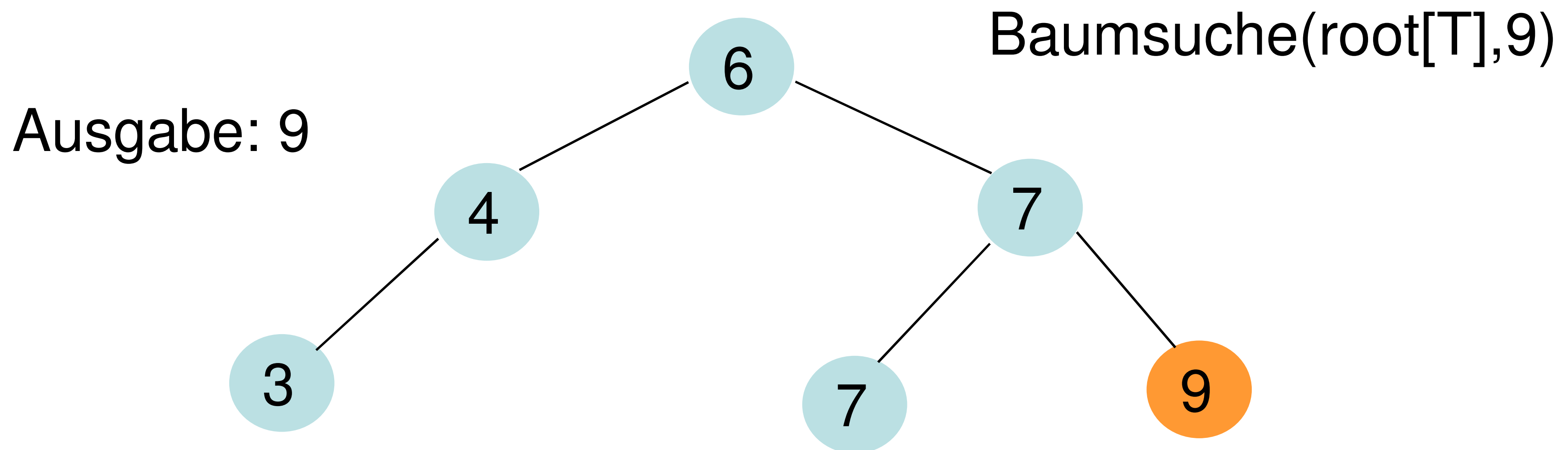
1. **if** x=**nil or** k=key[x] **then return** x
2. **if** k<key[x] **then return** Baumsuche(lc[x],k)
3. **else** return Baumsuche(rc[x],k)



Binäre Suchbäume

Baumsuche(x,k)

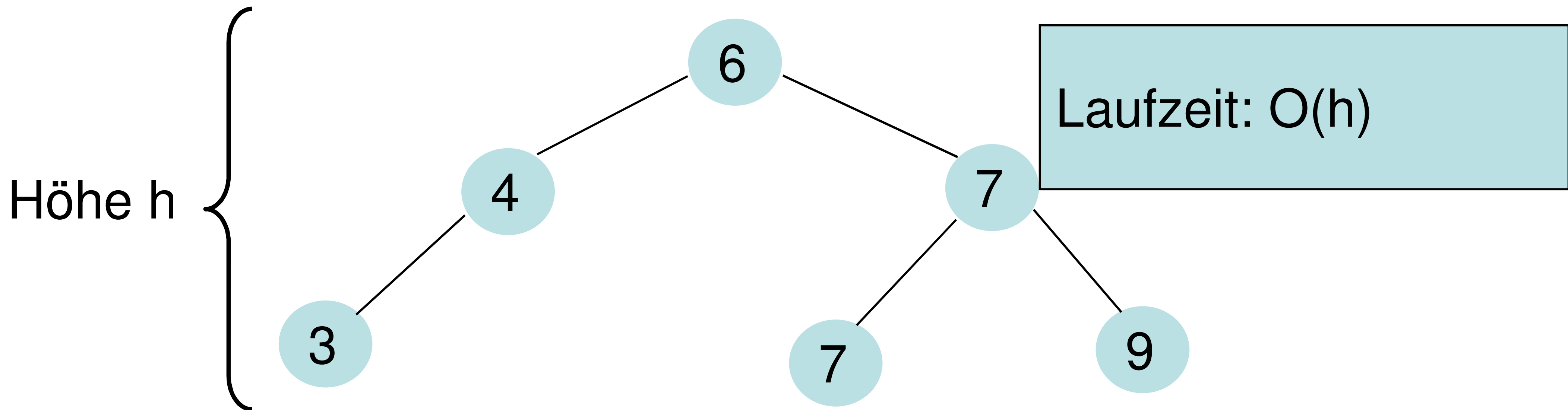
1. **if** x=**nil or** k=key[x] **then return** x
2. **if** k<key[x] **then return** Baumsuche(lc[x],k)
3. **else** return Baumsuche(rc[x],k)



Binäre Suchbäume

Baumsuche(x,k)

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



Binäre Suchbäume

Weitere Operationen in Binärbäumen

- Minimum/Maximumsuche
- Nachforlger/Vorgänger

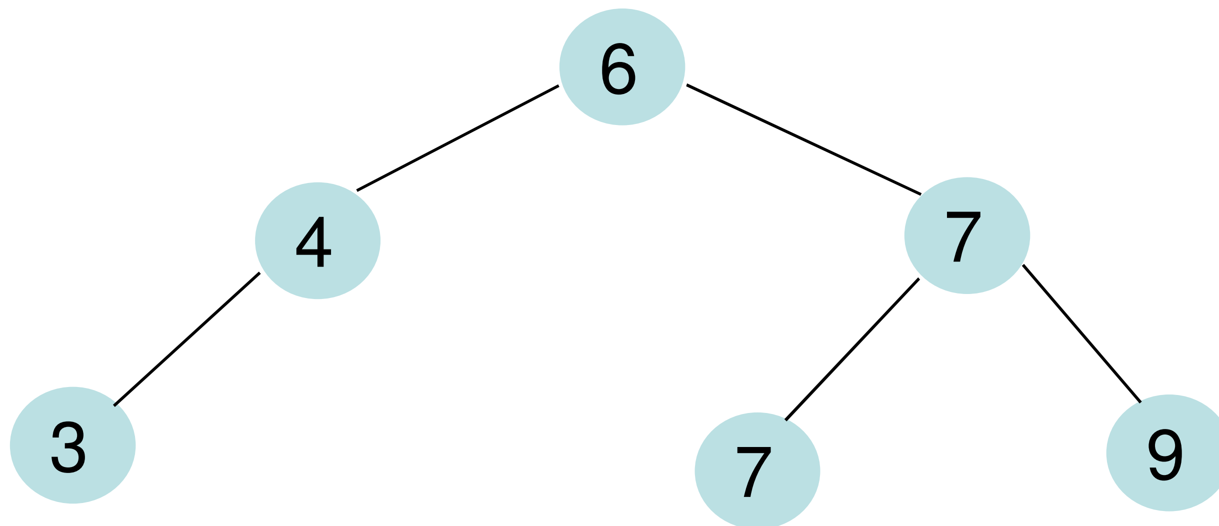
Minimum- und Maximumsuche:

- Suchbaumeigenschaft:
  Alle Knoten im rechten Unterbaum eines Knotens x sind größer gleich key[x]
- Alle Knoten im linken Unterbaum von x sind ≤ key[x]

Binäre Suchbäume

MinimumSuche(x)

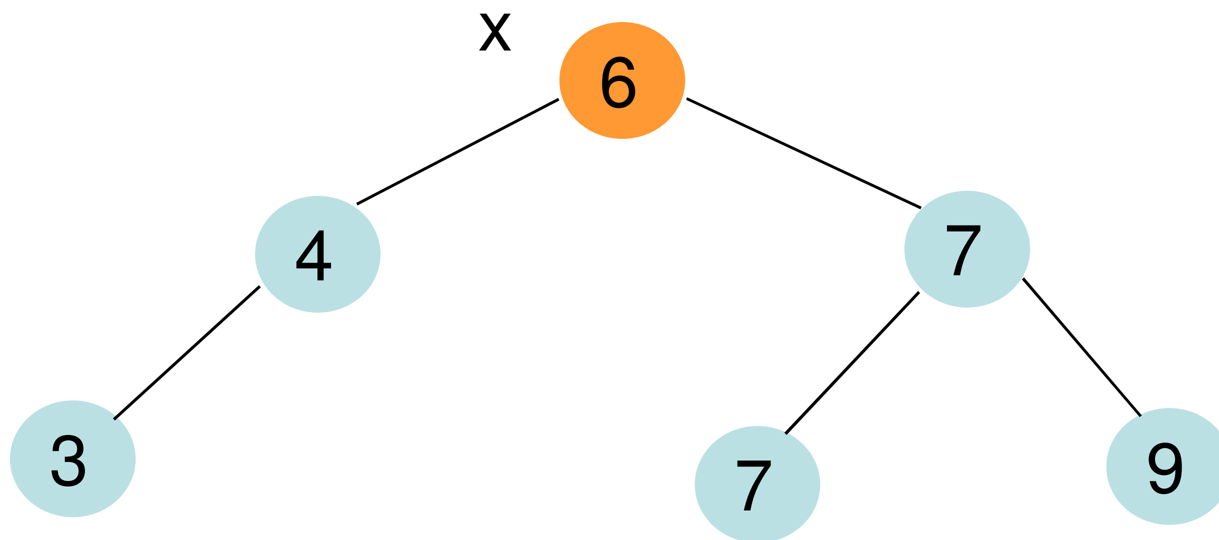
1. **while** lc[x]≠**nil do** x ← lc[x]
2. **return** x



Binäre Suchbäume

MinimumSuche(x)

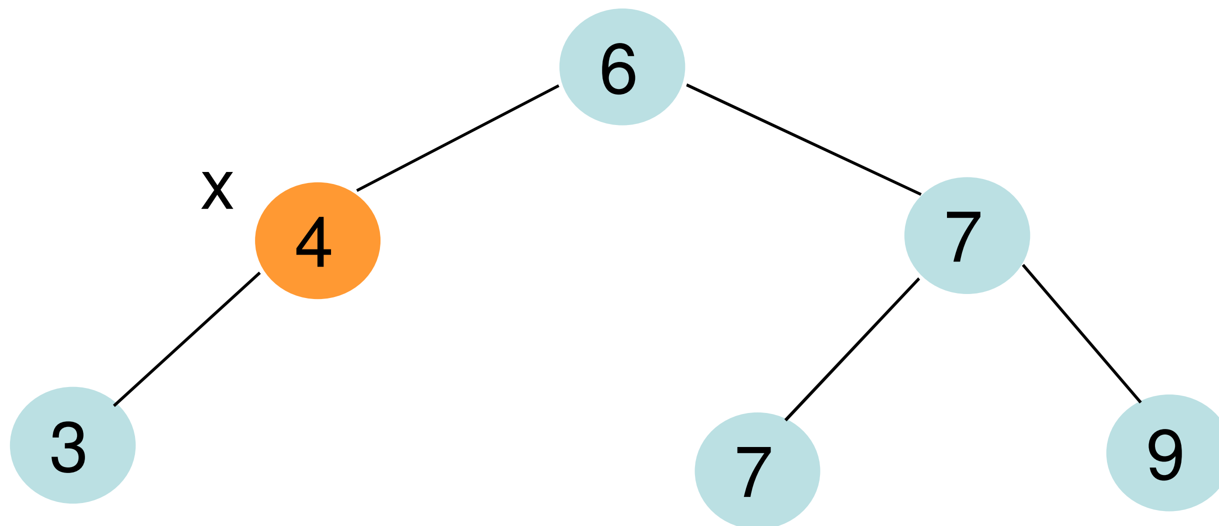
1. **while** lc[x]≠**nil** **do** x ← lc[x]
2. **return** x



Binäre Suchbäume

MinimumSuche(x)

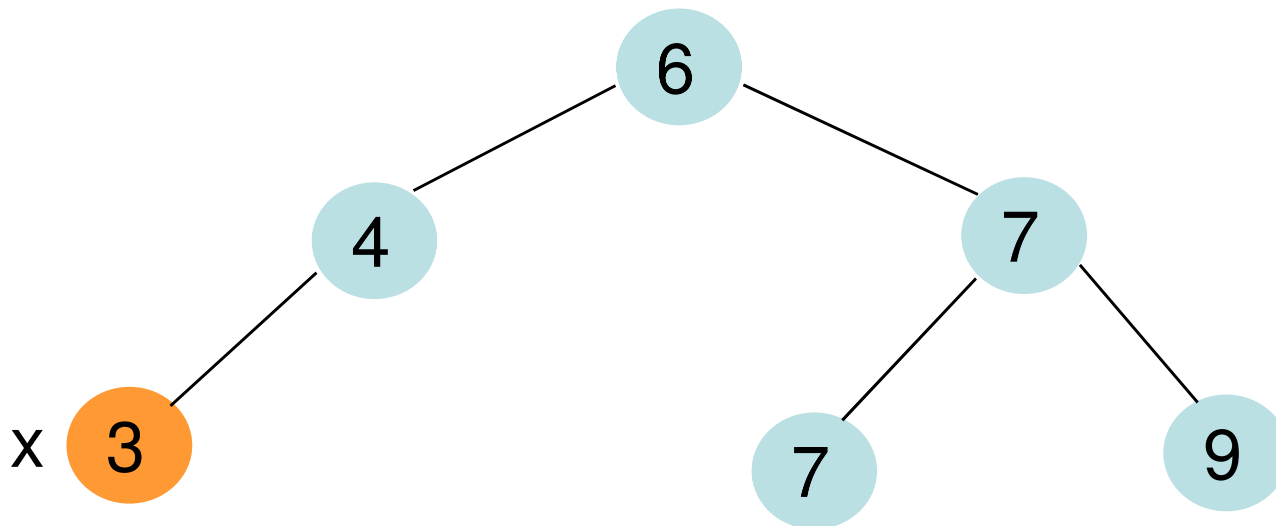
1. **while** lc[x]≠**nil do** x ← lc[x]
2. **return** x



Binäre Suchbäume

MinimumSuche(x)

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

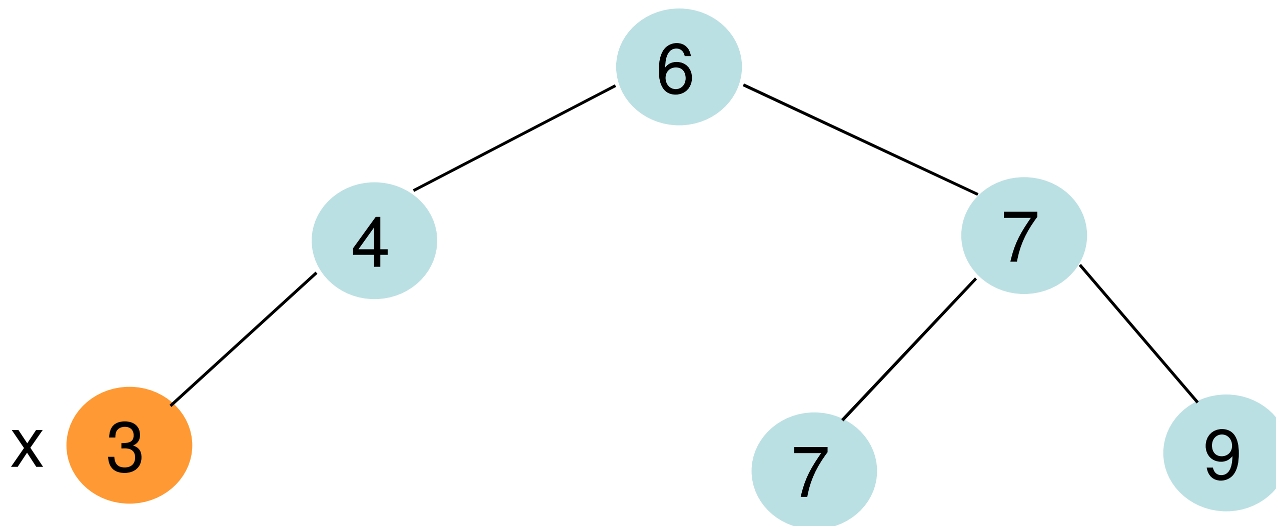


Binäre Suchbäume

MinimumSuche(x)

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

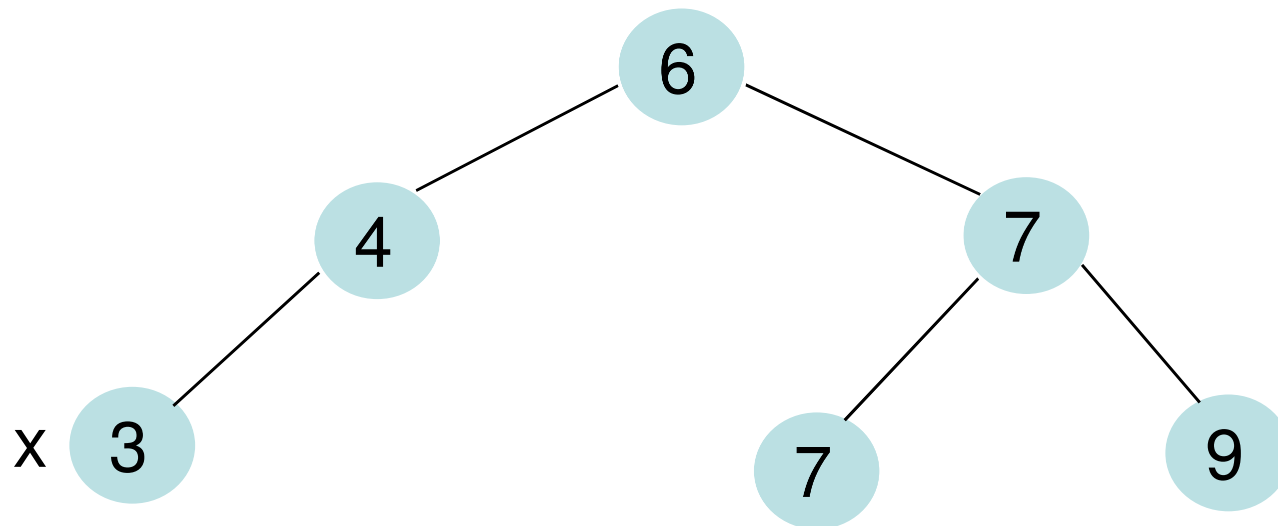
Laufzeit O(h)



Binäre Suchbäume

MaximumSuche(x)

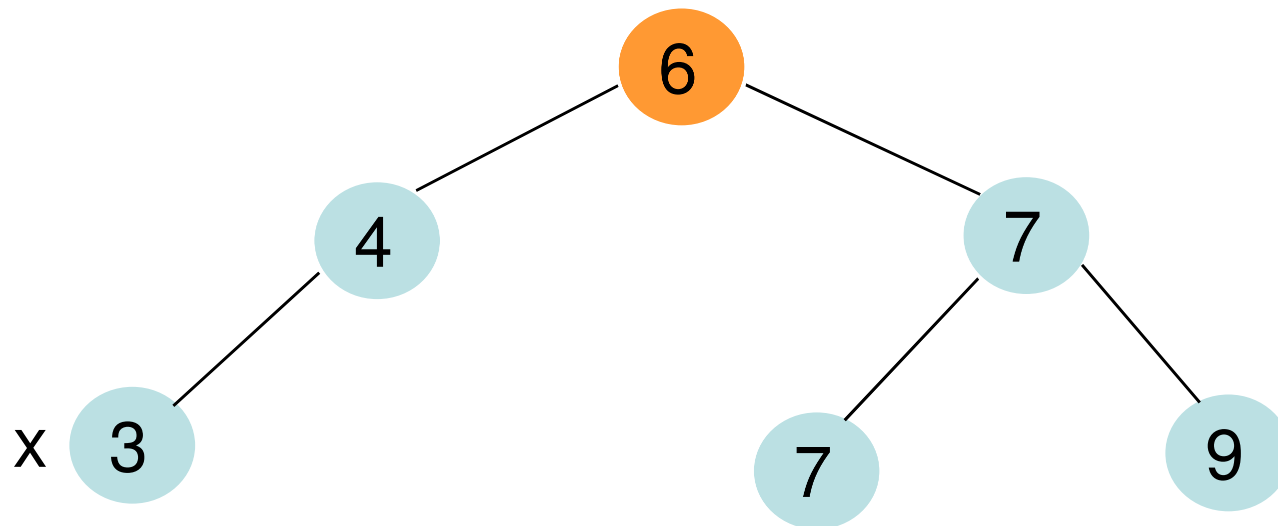
1. **while** rc[x]$\neq$**nil do** x $\leftarrow$ rc[x]
2. **return** x



Binäre Suchbäume

MaximumSuche(x)

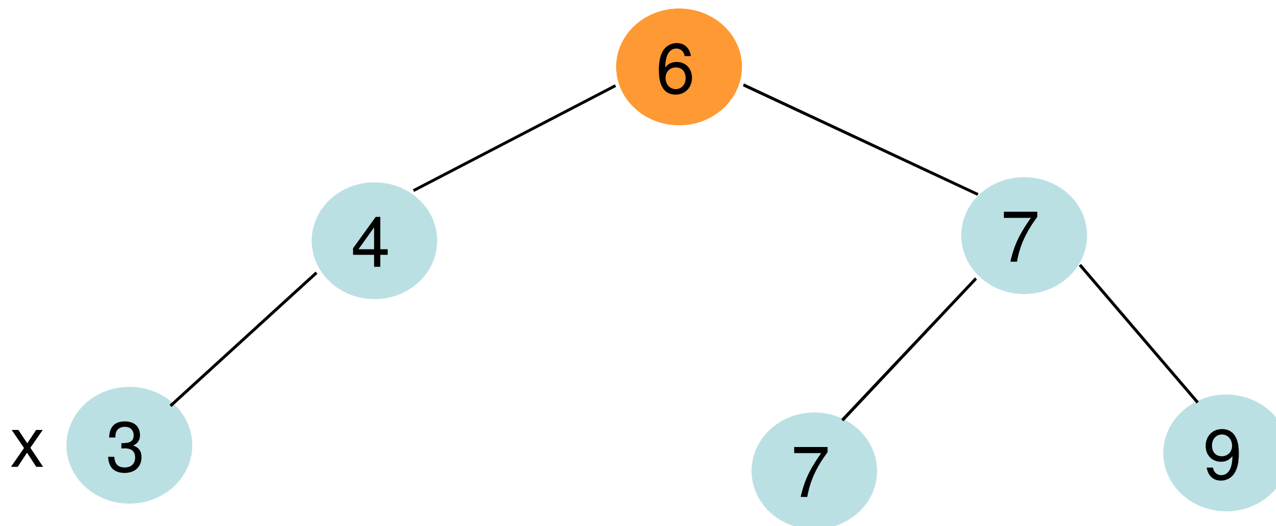
1. **while** rc[x]$\neq$**nil do** x $\leftarrow$ rc[x]
2. **return** x



Binäre Suchbäume

MaximumSuche(x)

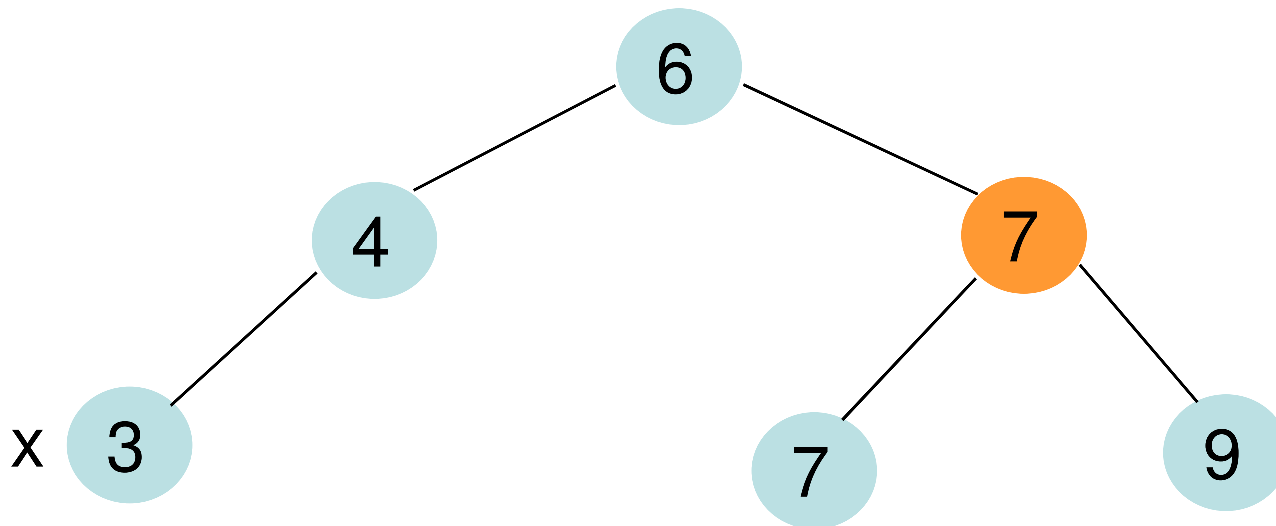
1. **while** rc[x]$\neq$**nil do** x $\leftarrow$ rc[x]
2. **return** x



Binäre Suchbäume

MaximumSuche(x)

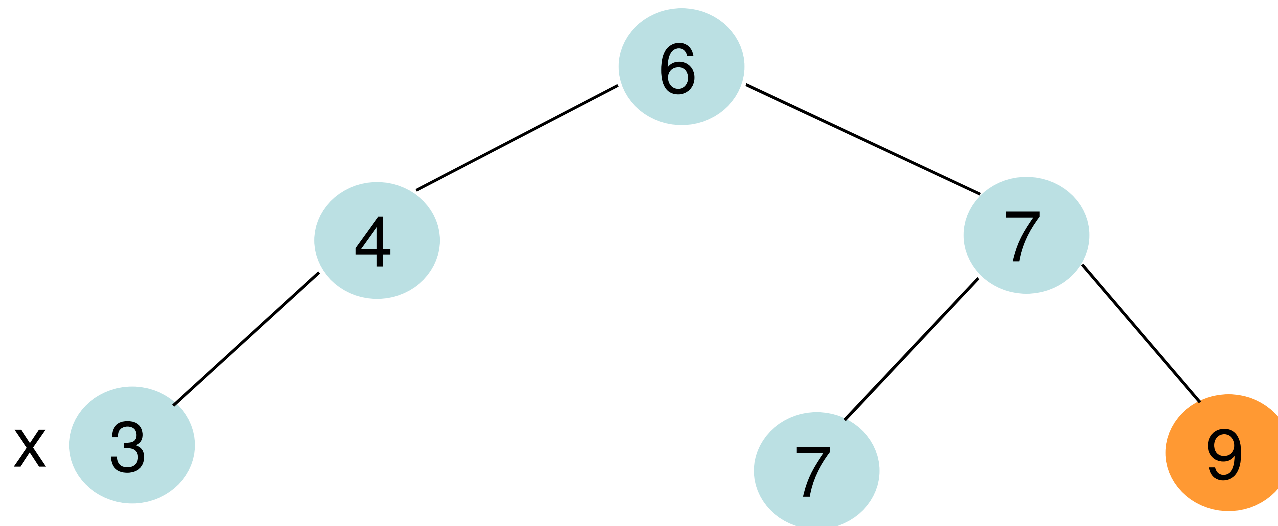
1. **while** rc[x]≠**nil do** x ← rc[x]
2. **return** x



Binäre Suchbäume

MaximumSuche(x)

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

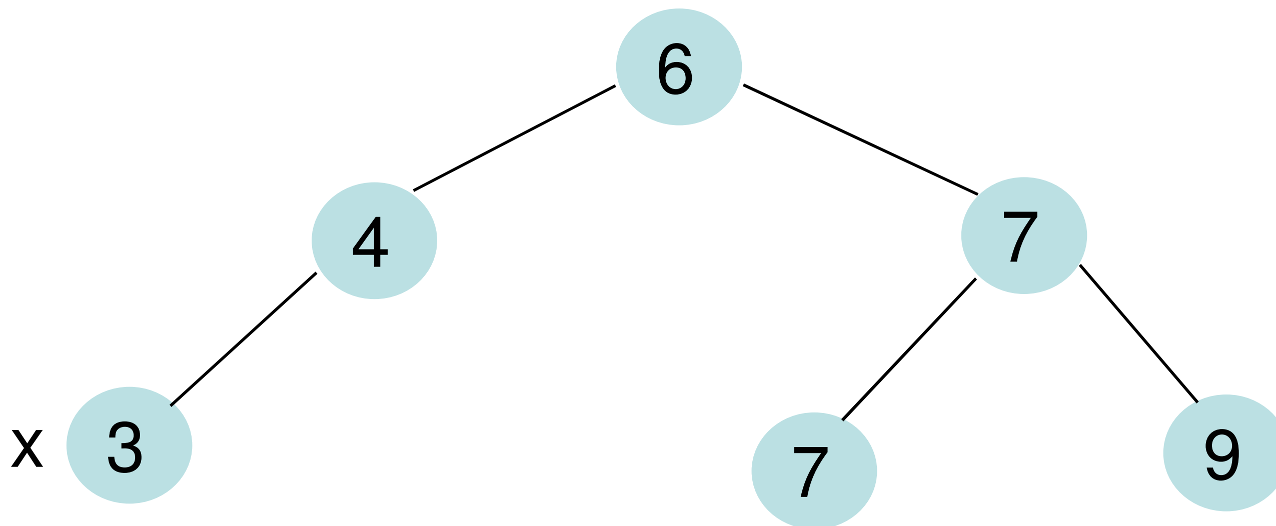


Binäre Suchbäume

MaximumSuche(x)

1. **while** rc[x]$\neq$**nil do** x $\leftarrow$ rc[x]
2. **return** x

Laufzeit O(h)

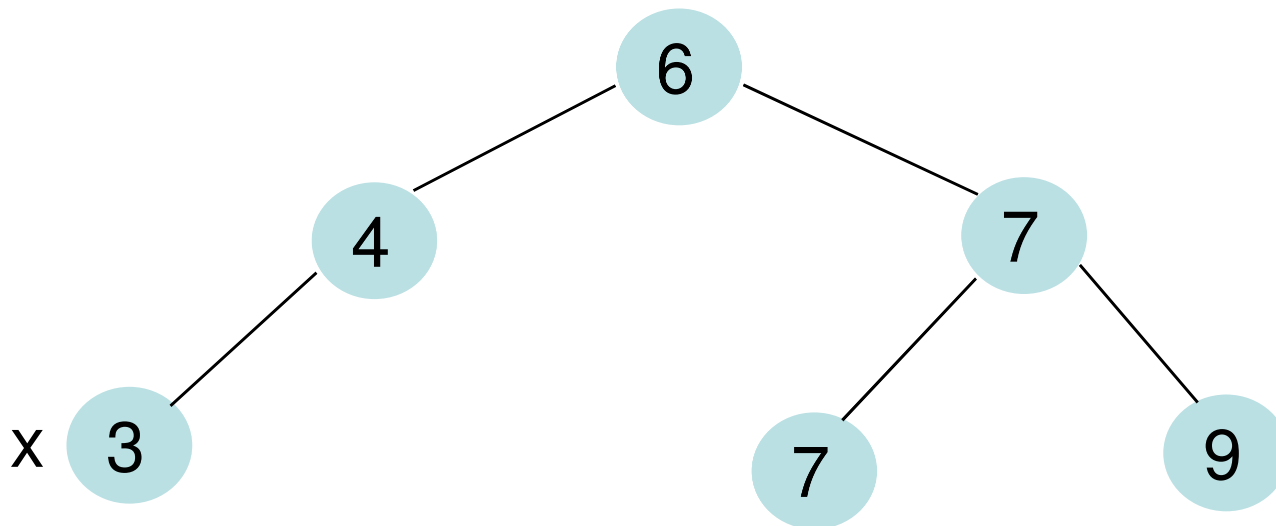


Binäre Suchbäume

MaximumSuche(x)

1. **while** rc[x]$\neq$**nil do** x $\leftarrow$ rc[x]
2. **return** x

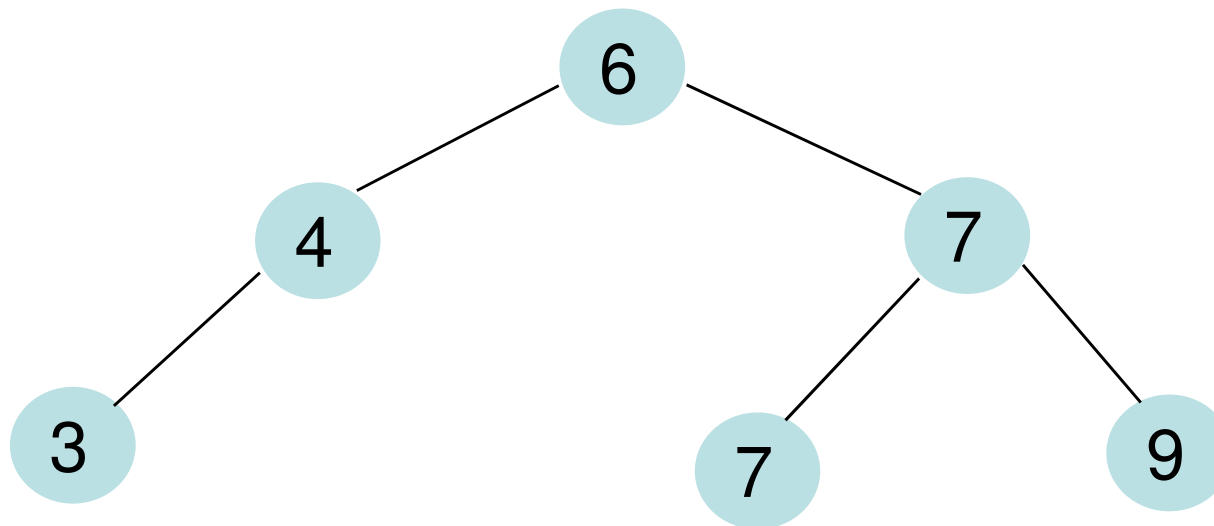
Laufzeit O(h)



Binäre Suchbäume

Nachfolgersuche:

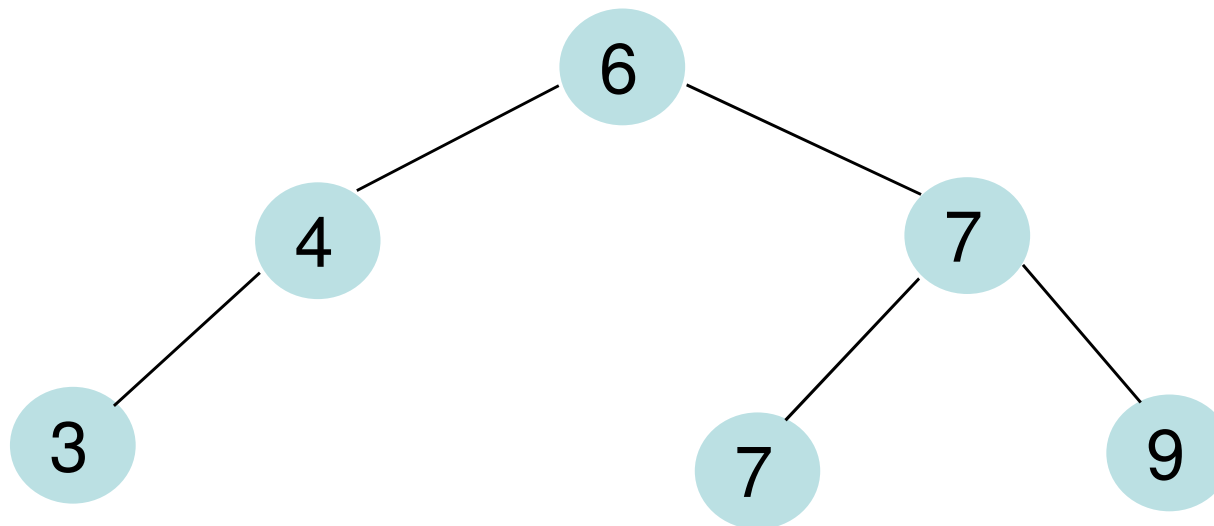
- Nachfolger bzgl. Inorder-Tree-Walk
- Wenn alle Schlüssel unterschiedlich, dann ist das der nächstgrößere Schlüssel



Binäre Suchbäume

Nachfolgersuche:

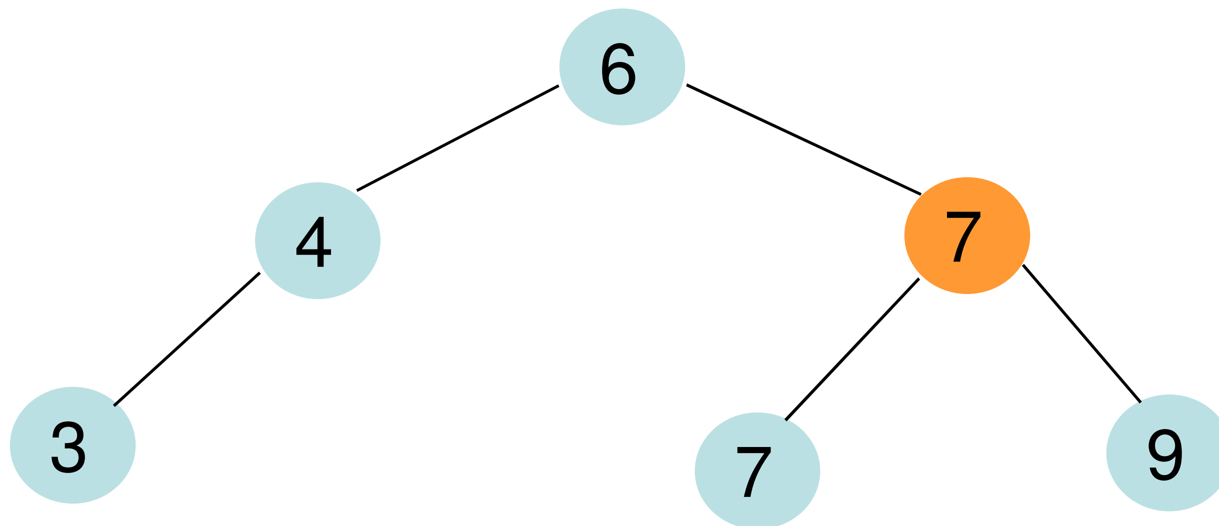
- Fall 1 (rechter Unterbaum von x nicht leer):
  Dann ist der linkeste Knoten im rechten Unterbaum der Nachfolger von x



Binäre Suchbäume

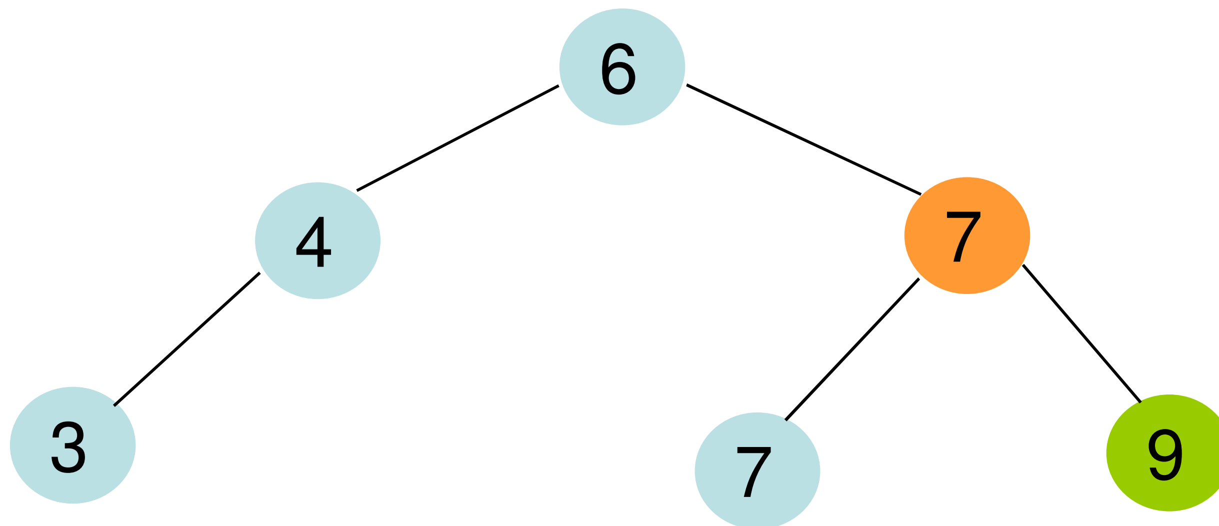
Nachfolgersuche:

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



Nachfolgersuche:

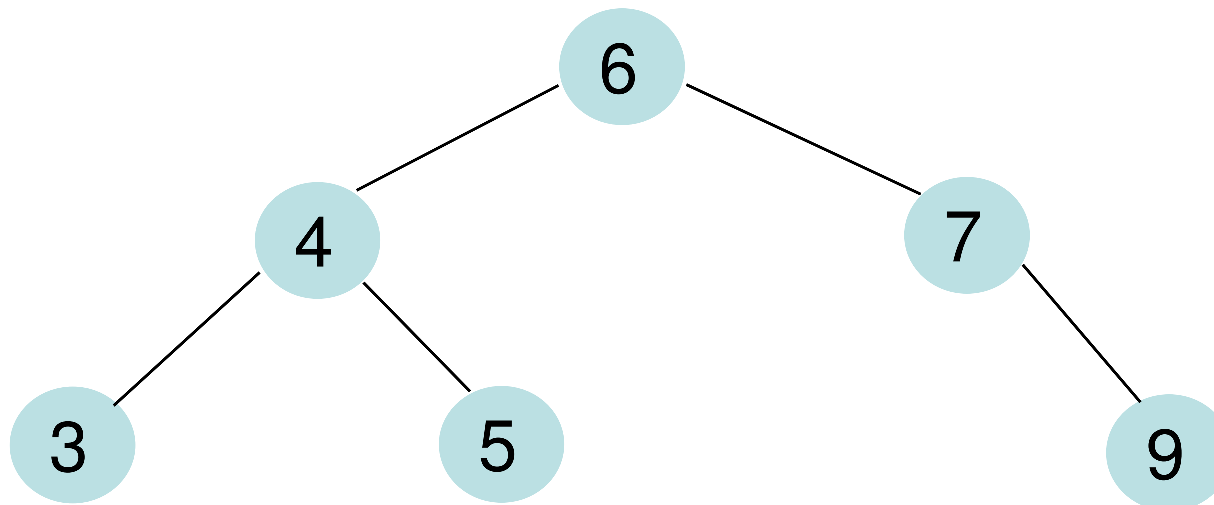
- Fall 1 (rechter Unterbaum von x nicht leer):
  Dann ist der linkeste Knoten im rechten Unterbaum der Nachfolger von x



Binäre Suchbäume

Nachfolgersuche:

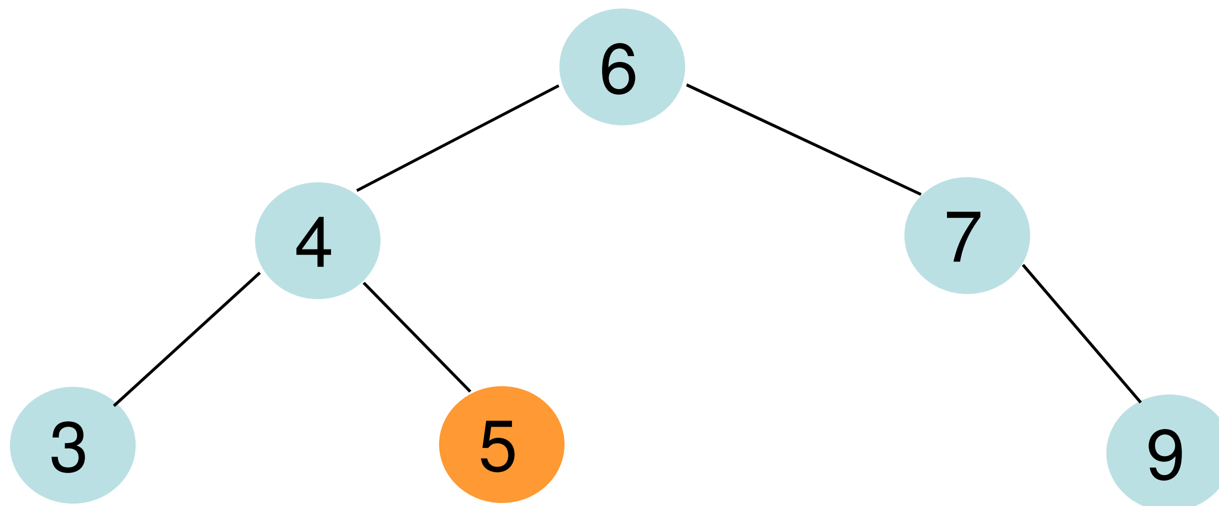
- Fall 2 (rechter Unterbaum von x leer und x hat Nachfolger y): Dann ist y der niedrigste Vorgänger von x, dessen linkes Kind ebenfalls Vorgänger von x ist



Binäre Suchbäume

Nachfolgersuche:

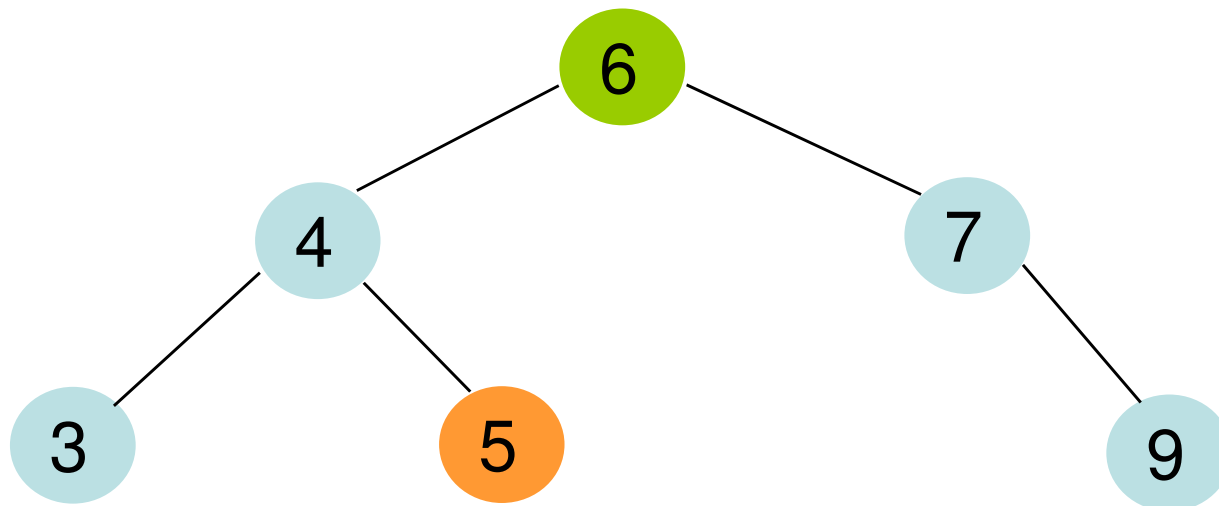
- Fall 2 (rechter Unterbaum von x leer und x hat Nachfolger y): Dann ist y der niedrigste Vorgänger von x, dessen linkes Kind ebenfalls Vorgänger von x ist



Binäre Suchbäume

Nachfolgersuche:

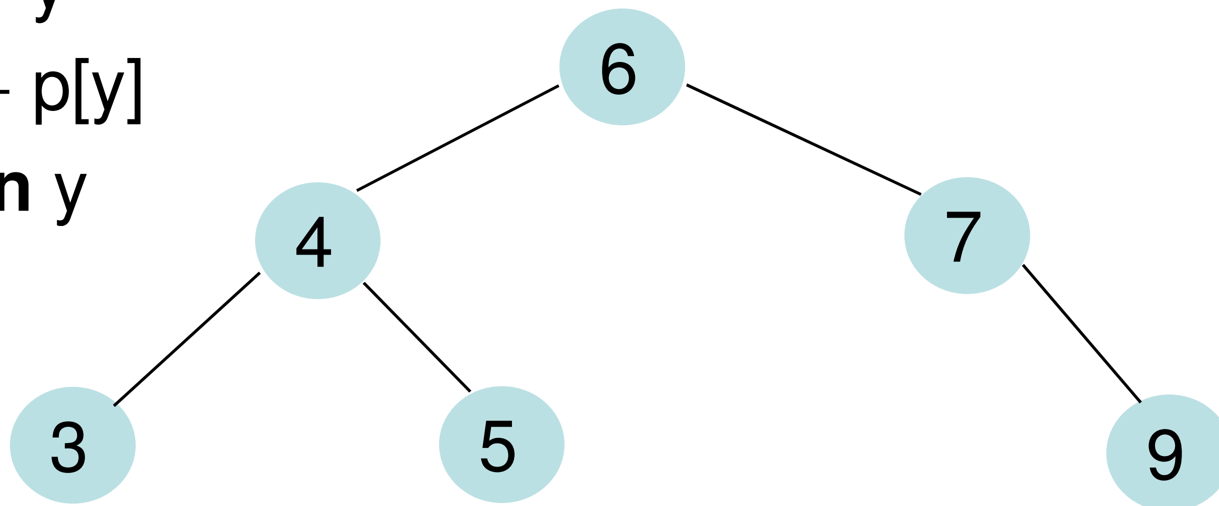
- Fall 2 (rechter Unterbaum von x leer und x hat Nachfolger y): Dann ist y der niedrigste Vorgänger von x, dessen linkes Kind ebenfalls Vorgänger von x ist



Binäre Suchbäume

Nachfolgersuche(x)

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

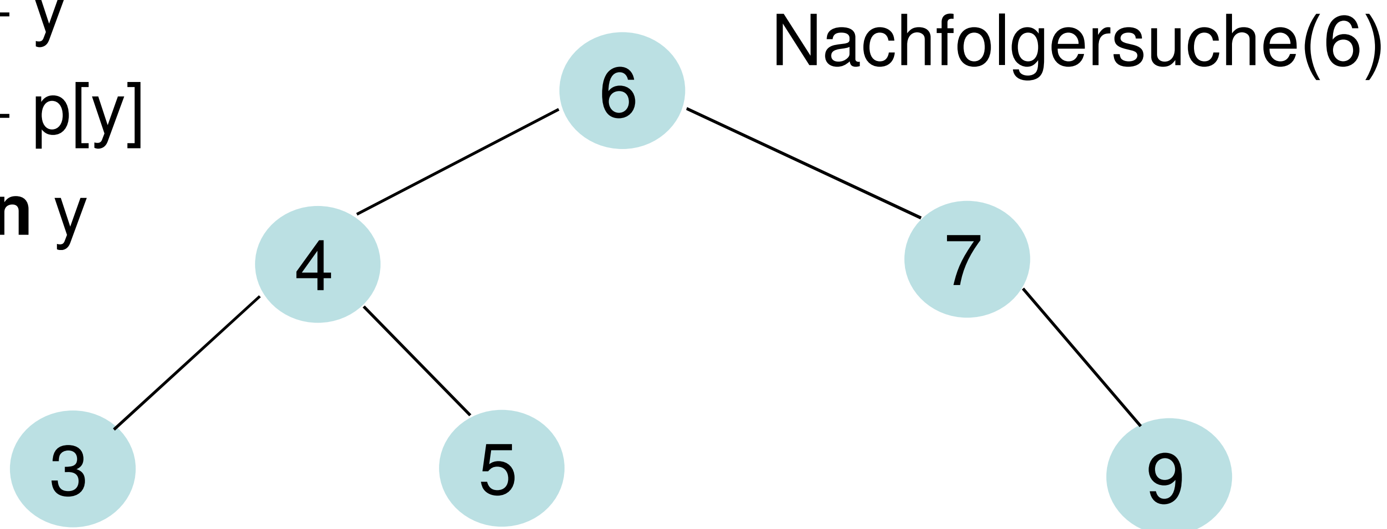


Binäre Suchbäume

Nachfolgersuche(x)

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

Nachfolgersuche(6)

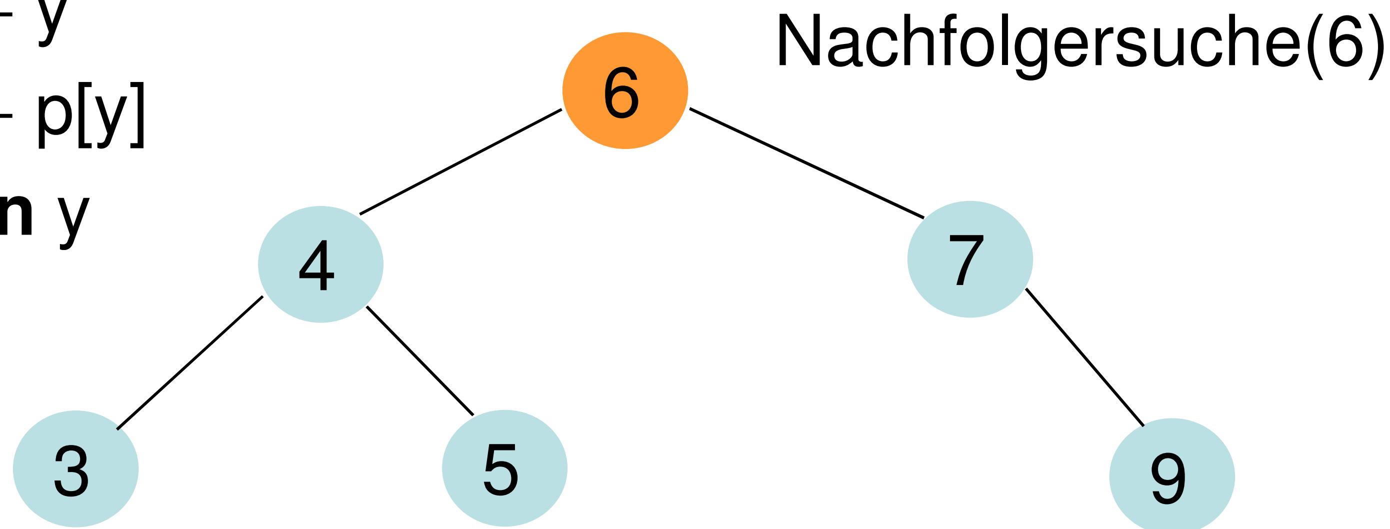


Binäre Suchbäume

Nachfolgersuche(x)

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

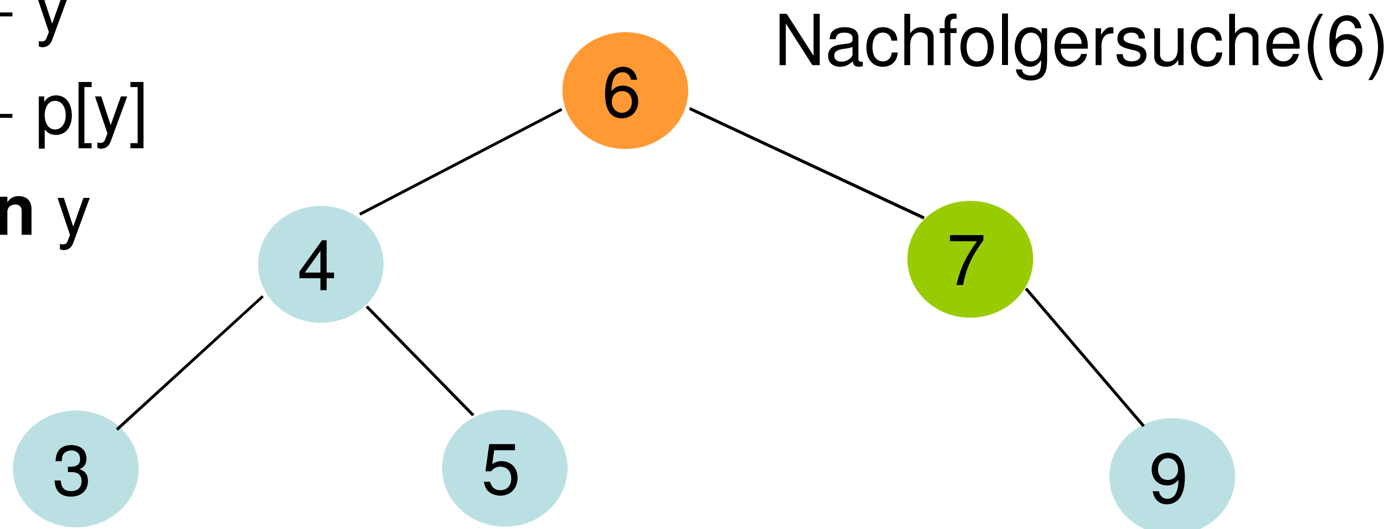
Nachfolgersuche(6)



Binäre Suchbäume

Nachfolgersuche(x)

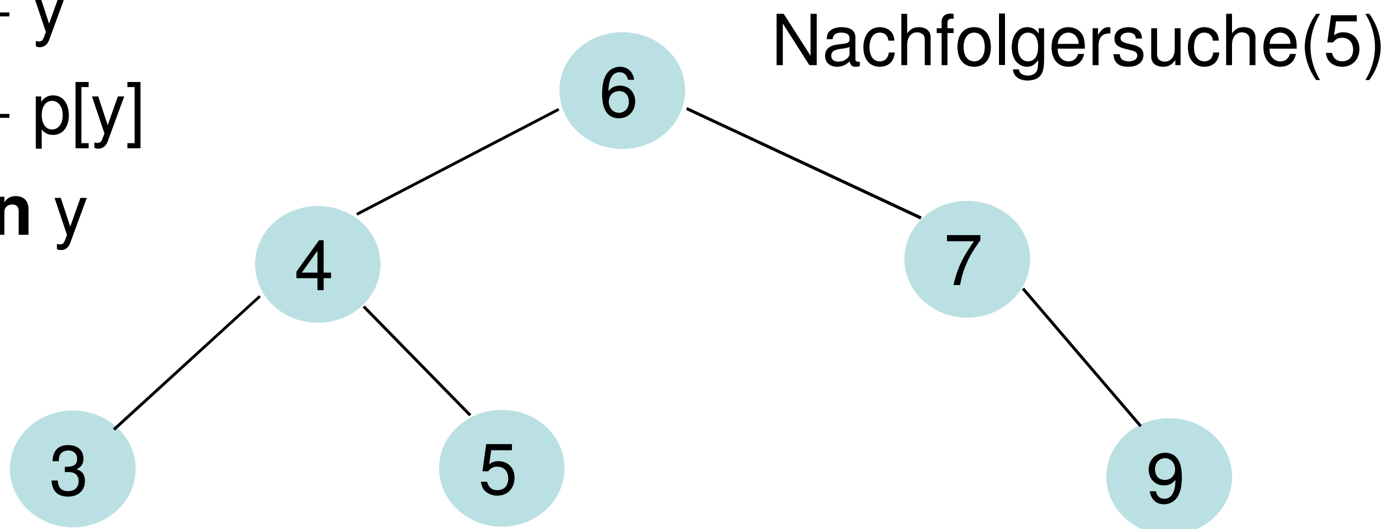
1. **if** rc[x] ≠ **nil then return** MinimumSuche(rc[x])
2. y ← p[x]
3. **while** y ≠ **nil** and x=rc[y] **do**
4. x ← y
5. y ← p[y]
6. **return** y



Binäre Suchbäume

Nachfolgersuche(x)

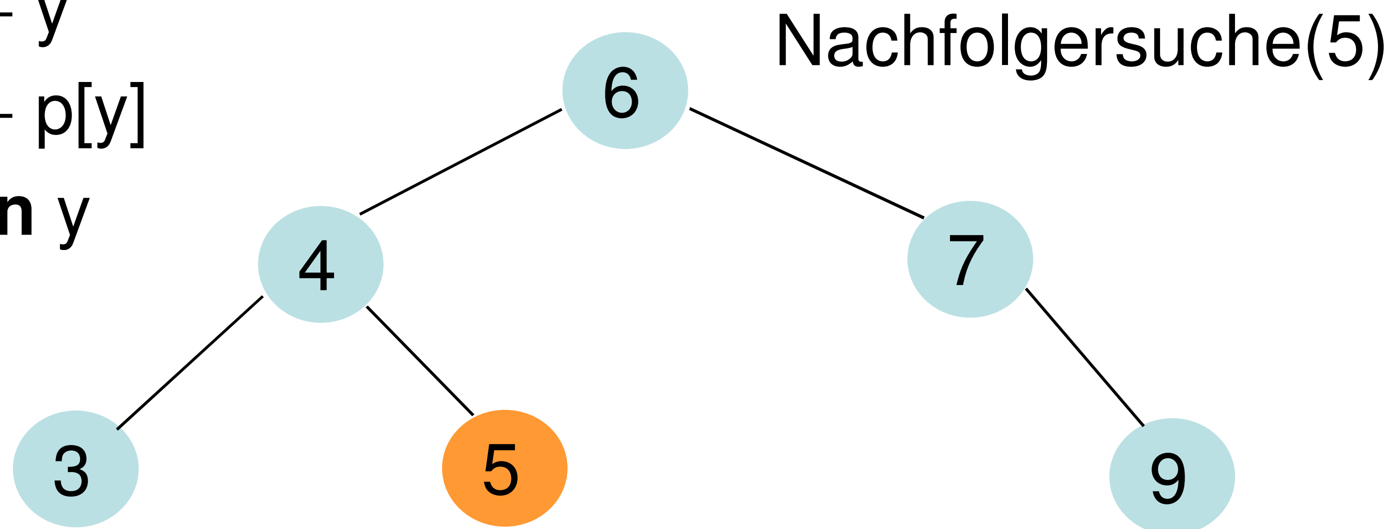
1. **if** rc[x] $\neq$ **nil then return** MinimumSuche(rc[x])
2. y $\leftarrow$ p[x]
3. **while** y $\neq$ **nil** and x=rc[y] **do**
4. x $\leftarrow$ y
5. y $\leftarrow$ p[y]
6. **return** y



Binäre Suchbäume

Nachfolgersuche(x)

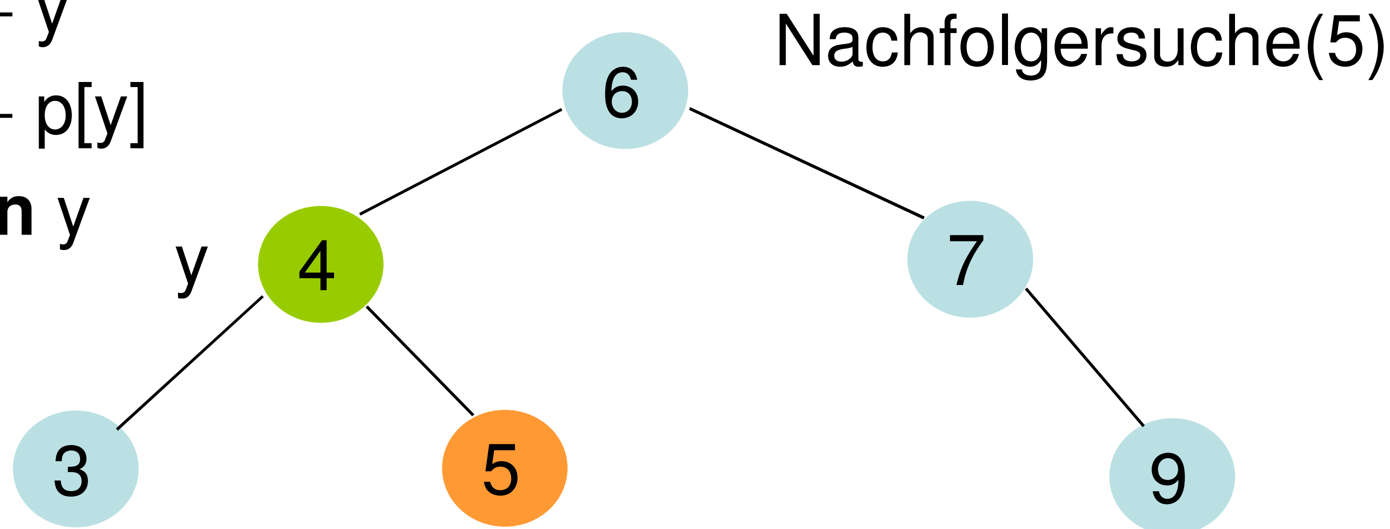
1. **if** rc[x] ≠ **nil then return** MinimumSuche(rc[x])
2. y ← p[x]
3. **while** y ≠ **nil** and x=rc[y] **do**
4. x ← y
5. y ← p[y]
6. **return** y



Binäre Suchbäume

Nachfolgersuche(x)

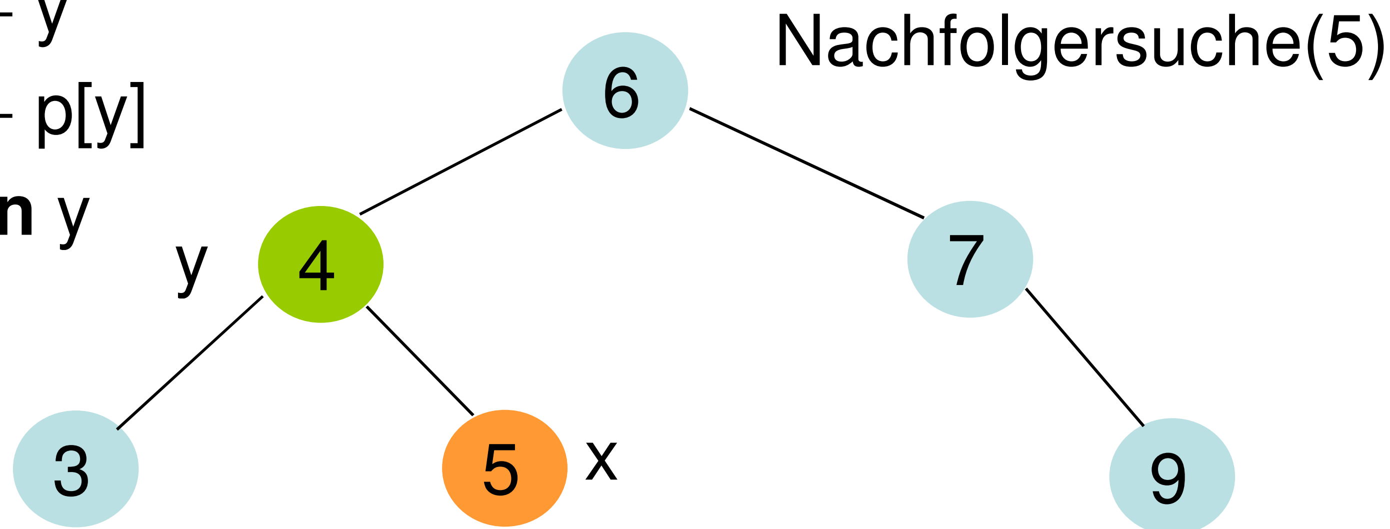
1. **if** rc[x] ≠ **nil then return** MinimumSuche(rc[x])
2. y ← p[x]
3. **while** y ≠ **nil** and x=rc[y] **do**
4. x ← y
5. y ← p[y]
6. **return** y



Binäre Suchbäume

Nachfolgersuche(x)

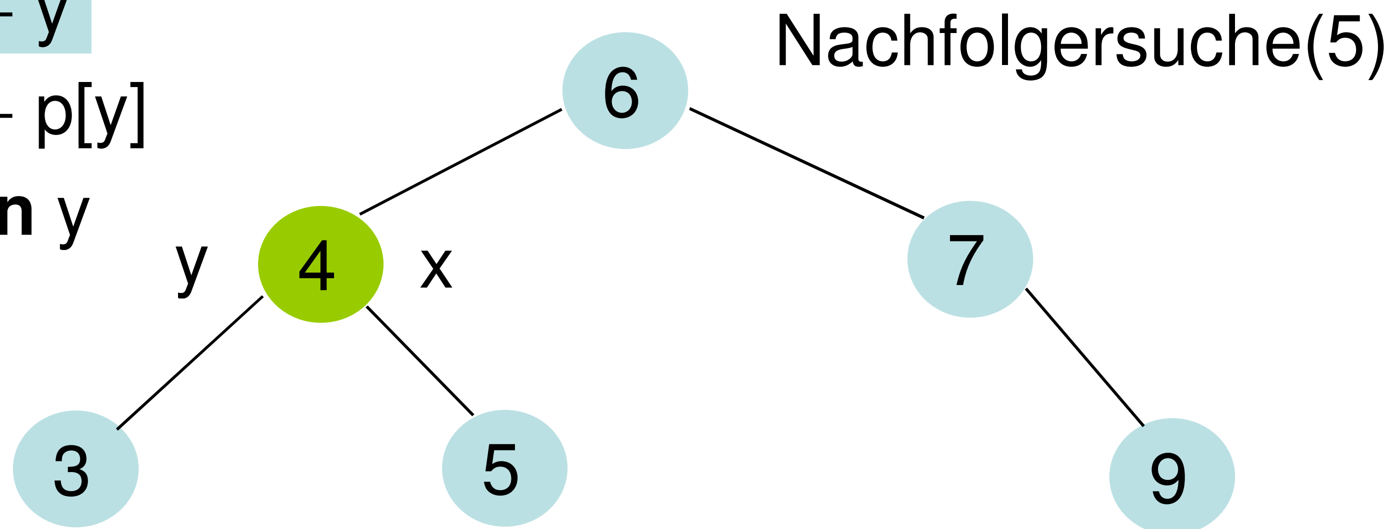
1. **if** rc[x] ≠ **nil then return** MinimumSuche(rc[x])
2. y ← p[x]
3. **while** y ≠ **nil** and x=rc[y] **do**
4.   x ← y
5.   y ← p[y]
6. **return** y



Binäre Suchbäume

Nachfolgersuche(x)

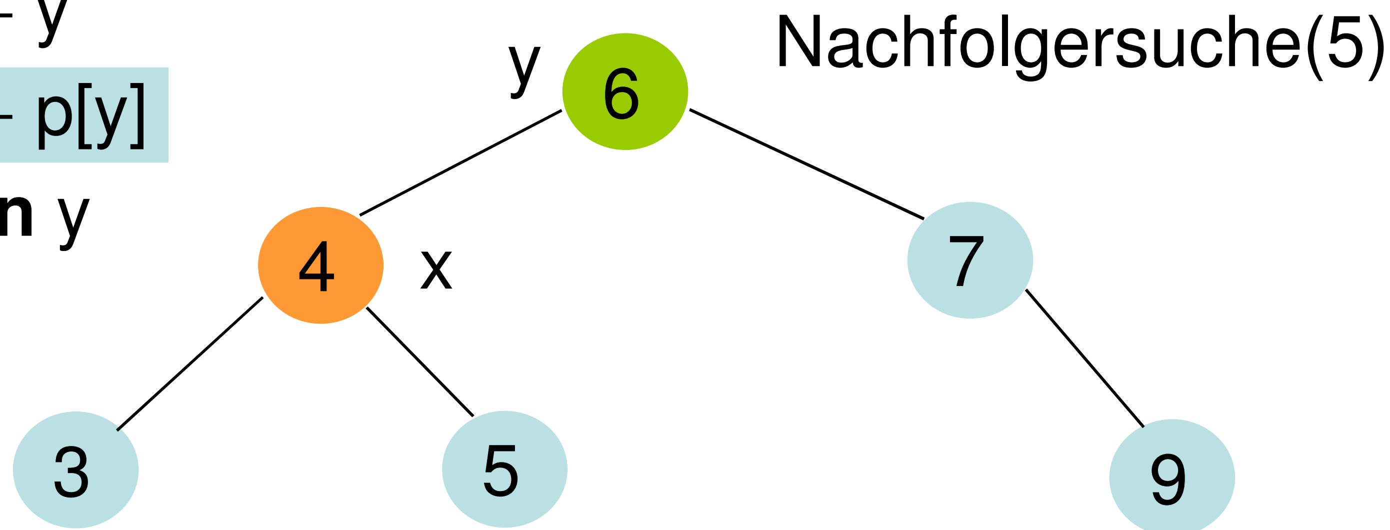
1. **if** rc[x] ≠ **nil then return** MinimumSuche(rc[x])
2. y ← p[x]
3. **while** y ≠ **nil** and x=rc[y] **do**
4. x ← y
5. y ← p[y]
6. **return** y



Binäre Suchbäume

Nachfolgersuche(x)

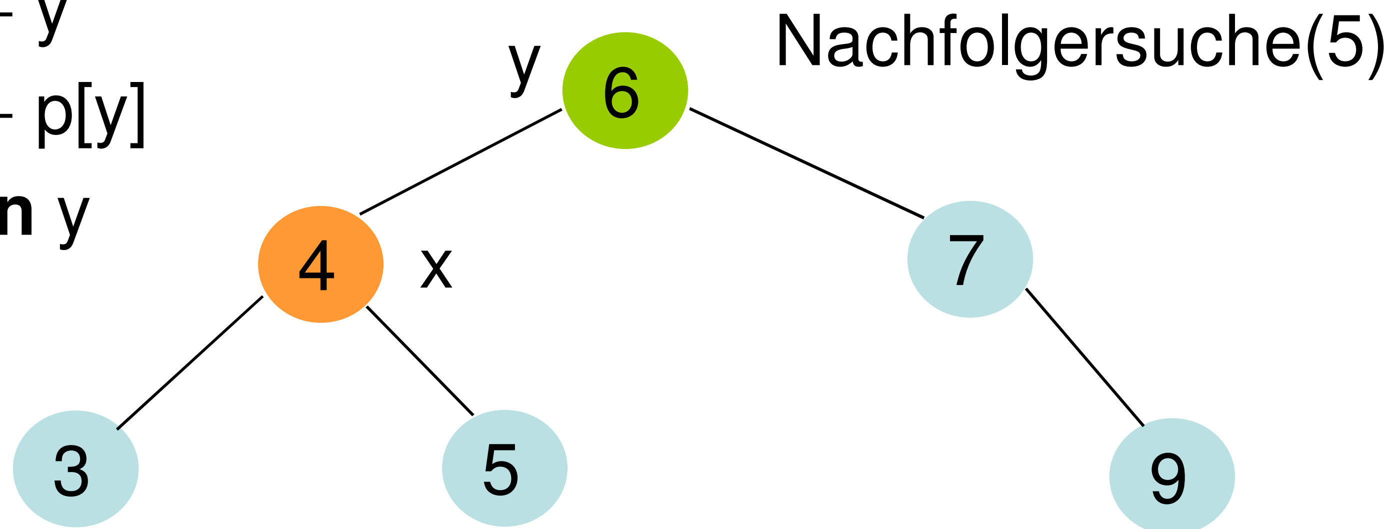
1. **if** rc[x] ≠ **nil then return** MinimumSuche(rc[x])
2. y ← p[x]
3. **while** y ≠ **nil** and x=rc[y] **do**
4. x ← y
5. y ← p[y]
6. **return** y



Binäre Suchbäume

Nachfolgersuche(x)

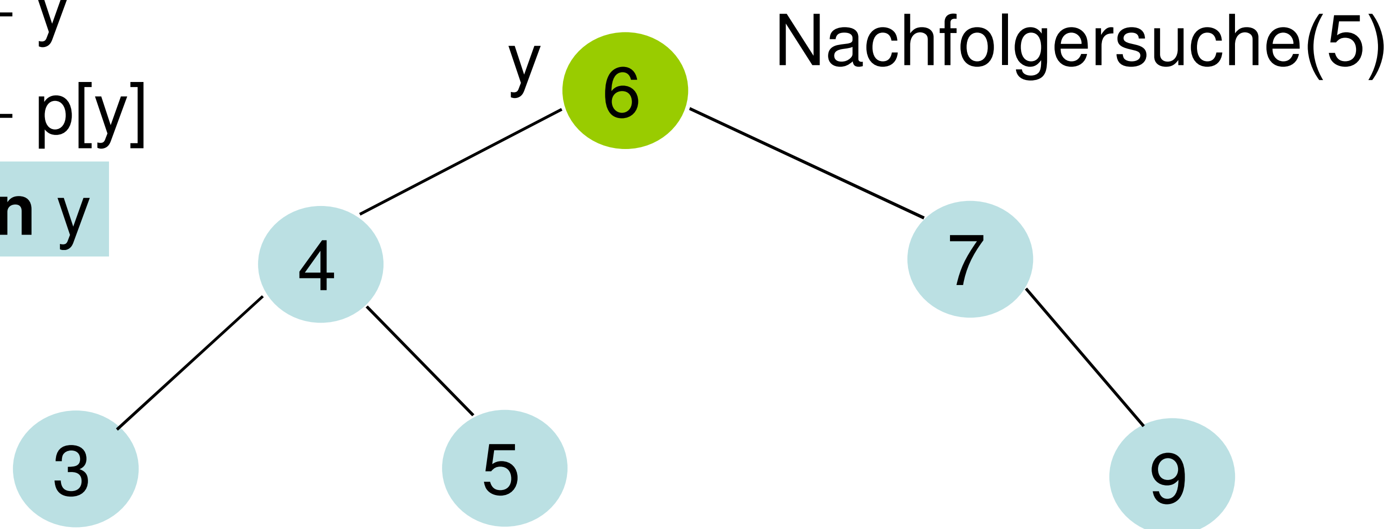
1. **if** rc[x] $\neq$ **nil then return** MinimumSuche(rc[x])
2. y $\leftarrow$ p[x]
3. **while** y $\neq$ **nil** and x=rc[y] **do**
4. x $\leftarrow$ y
5. y $\leftarrow$ p[y]
6. **return** y



Binäre Suchbäume

Nachfolgersuche(x)

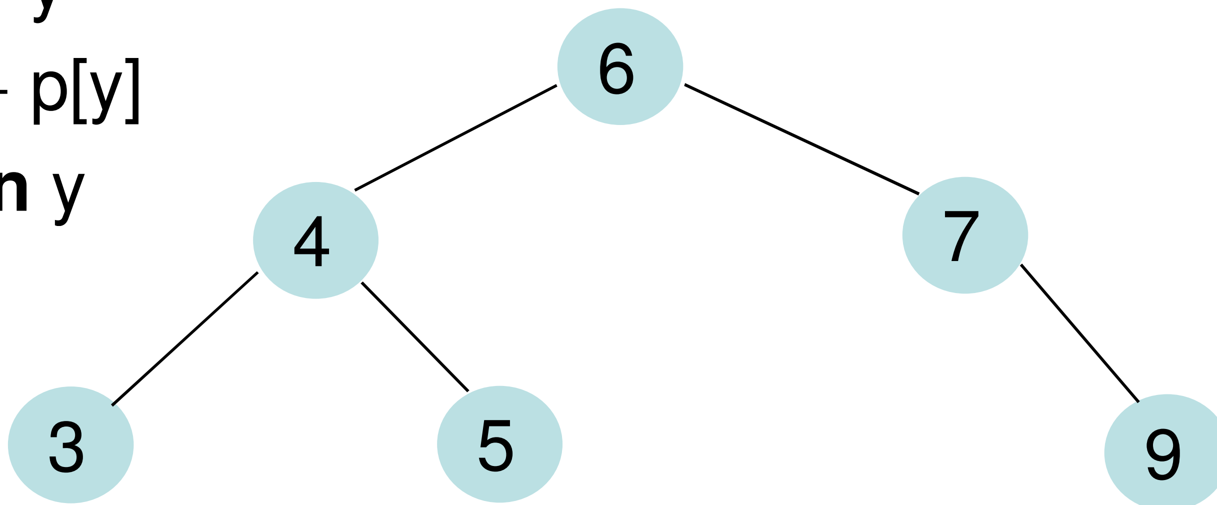
1. **if** rc[x] $\neq$ **nil then return** MinimumSuche(rc[x])
2. y $\leftarrow$ p[x]
3. **while** y $\neq$ **nil** and x=rc[y] **do**
4. x $\leftarrow$ y
5. y $\leftarrow$ p[y]
6. **return** y



Binäre Suchbäume

Nachfolgersuche(x)
1. **if** rc[x] ≠ **nil then return** MinimumSuche(rc[x])
2. y ← p[x]
3. **while** y ≠ **nil** and x=rc[y] **do**
4. x ← y
5. y ← p[y]
6. **return** y

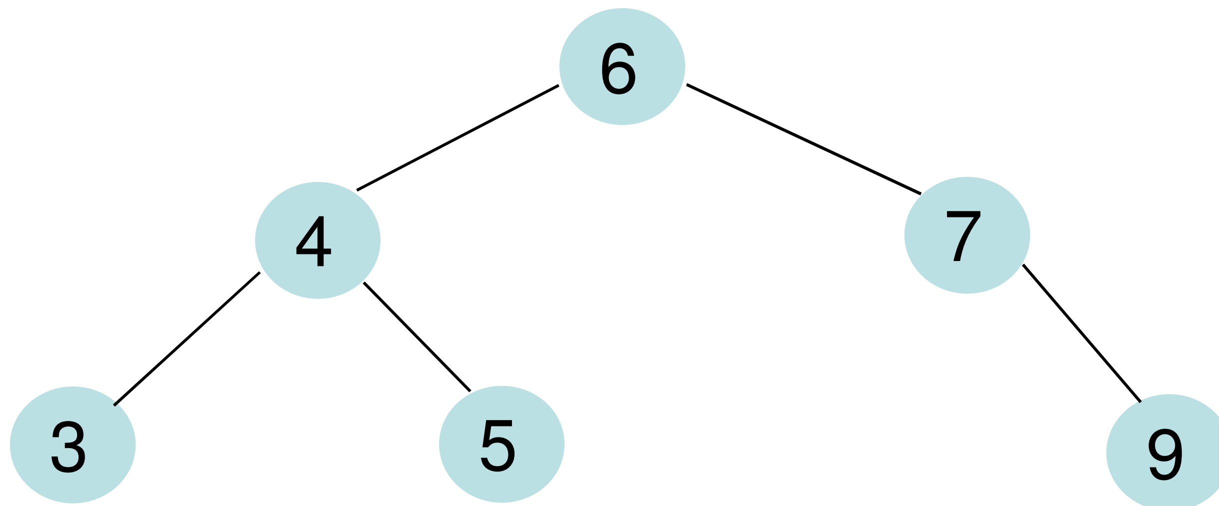
Laufzeit O(h)



Binäre Suchbäume

Vorgängersuche:

- Symmetrisch zu Nachfolgersuche
- Daher ebenfall O(h) Laufzeit



Binäre Suchbäume

Binäre Suchbäume:

- Aufzähler der Elemente mit Inorder-Tree-Walk in O(n) Zeit
- Such in O(h) Zeit
- Minimum/Maximum in O(h) Zeit

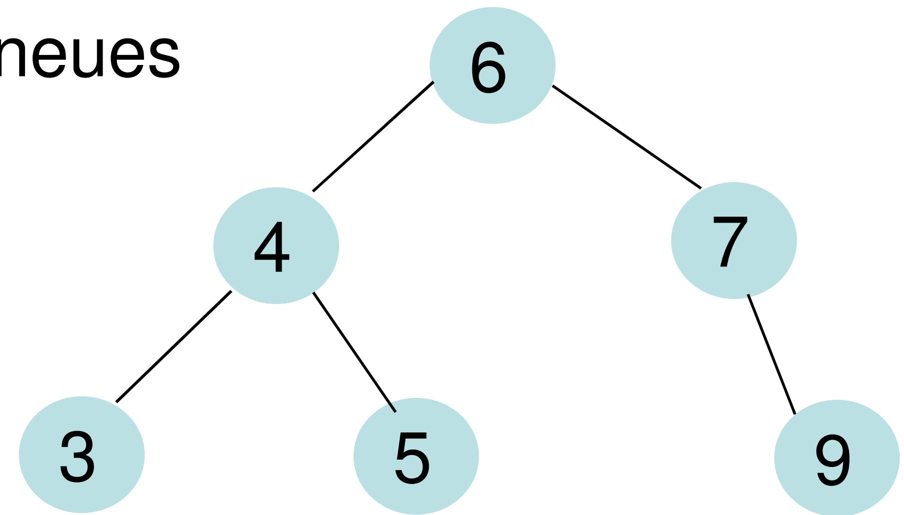
Dynamische Operationen?

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

Binäre Suchbäume

Einfügen:

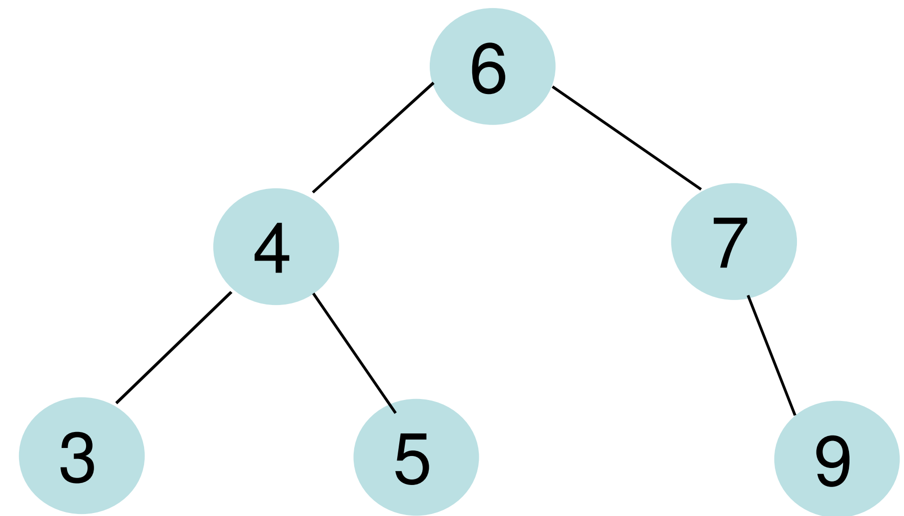
- Ähnlich wie Baumsuche: Finde Blatt, an das neuer Knoten angehängt wird
- Danach wird **nil**-Zeiger durch neues Element ersetzt



Binäre Suchbäume

Einfügen(T,z)

1. y← **nil**; x ← root[T]
2. **while** x≠**nil** **do**
3. y ← x
4. **if** key[z] < key[x] **then** x ← lc[x]
5. **else** x ← rc[x]
6. p[z] ← y
7. **if** y=**nil** **then** root[T] ← z
8. **else**
9. **if** key[z]< key[y] **then** lc[y] ← z
10. **else** rc[y] ← z



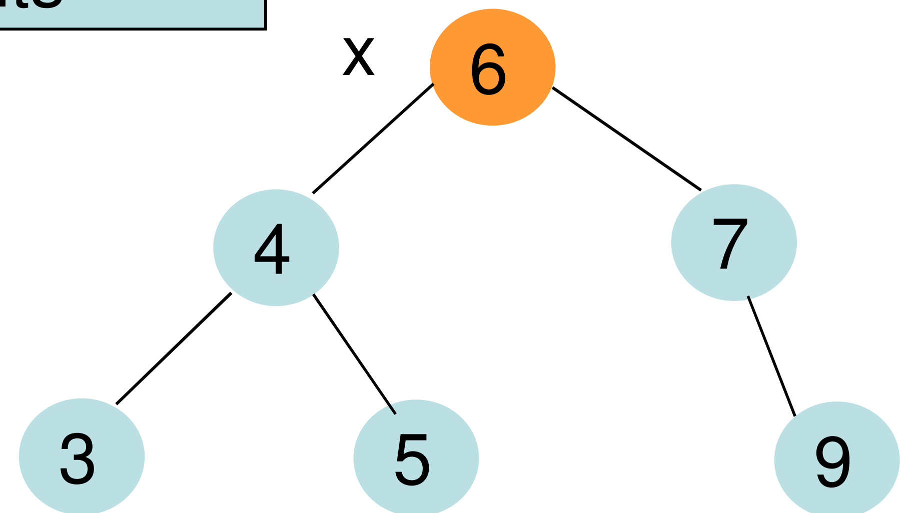
Binäre Suchbäume

Einfügen(T,z)

1. y← **nil**; x ← root[T]
2. **while** x≠**nil** **do**
3.   y ← x
4.   **if** key[z] < key[x] **then** x ← lc[x]
5.   **else** x ← rc[x]
6. p[z] ← y
7. **if** y=**nil** **then** root[T] ← z
8. **else**
9.   **if** key[z]< key[y] **then** lc[y] ← z
10.   **else** rc[y] ← z

y ist Vater des einzufügenden Elements

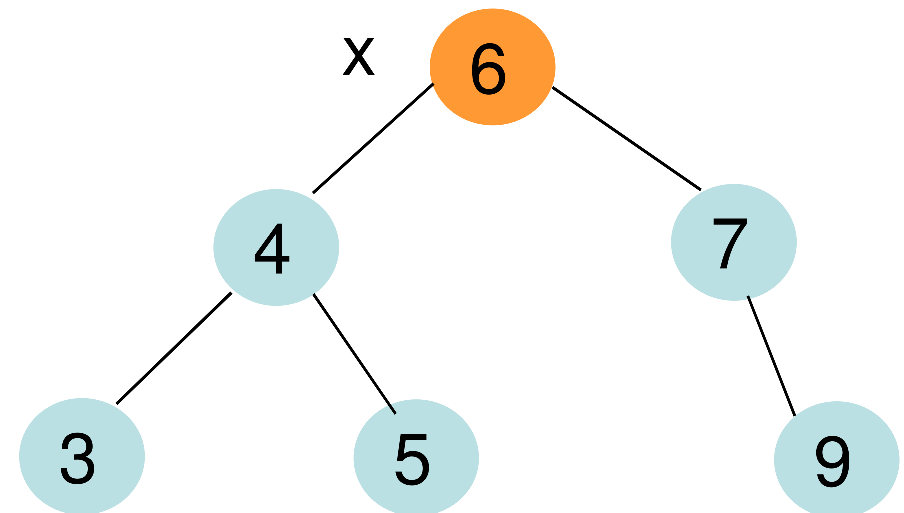
Einfügen(8)



Binäre Suchbäume

Einfügen(T,z)

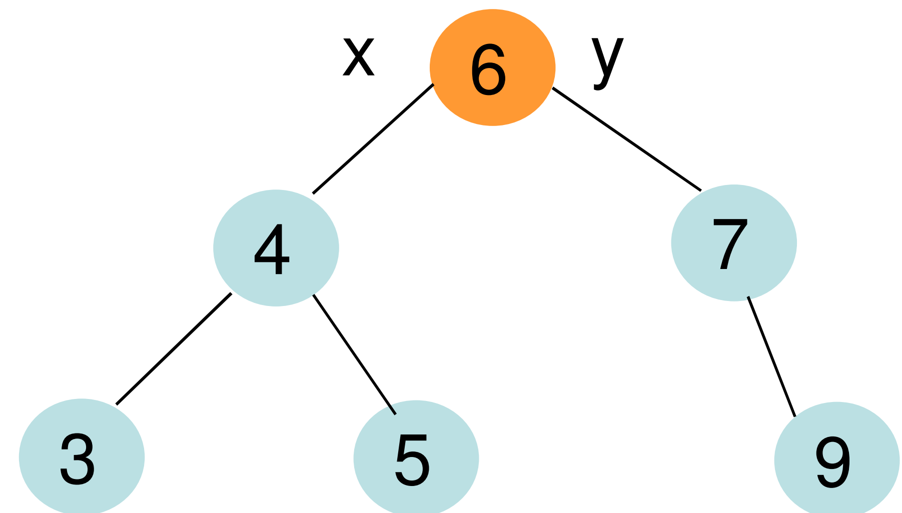
1. y← **nil**; x ← root[T]
2. **while** x≠**nil do**
3.   y ← x
4.   **if** key[z] < key[x] **then** x ← lc[x]
5.   **else** x ← rc[x]
6. p[z] ← y
7. **if** y=**nil then** root[T] ← z
8. **else**
9.   **if** key[z]< key[y] **then** lc[y] ← z
10.   **else** rc[y] ← z



Binäre Suchbäume

Einfügen(T,z)

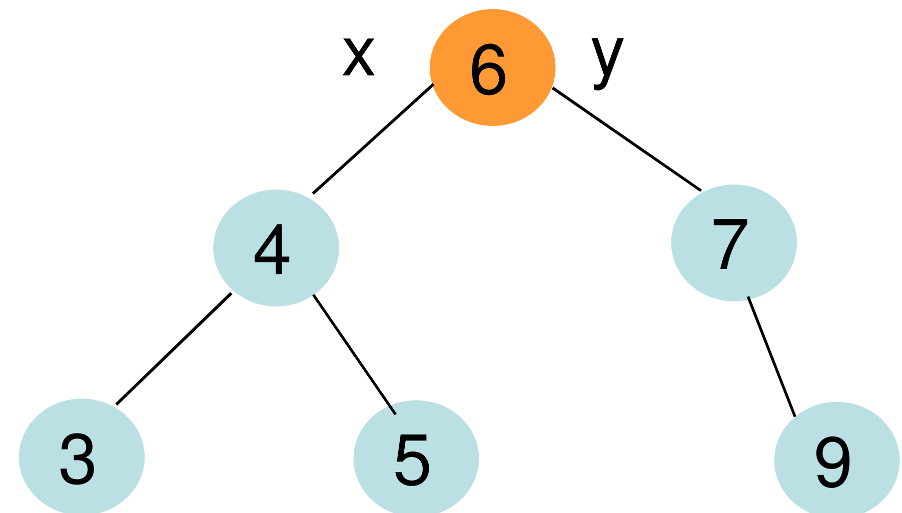
1. y← **nil**; x ← root[T]
2. **while** x≠**nil do**
3. y ← x
4. **if** key[z] < key[x] **then** x ← lc[x]
5. **else** x ← rc[x]
6. p[z] ← y
7. **if** y=**nil then** root[T] ← z
8. **else**
9. **if** key[z]< key[y] **then** lc[y] ← z
10. **else** rc[y] ← z



Binäre Suchbäume

Einfügen(T,z)

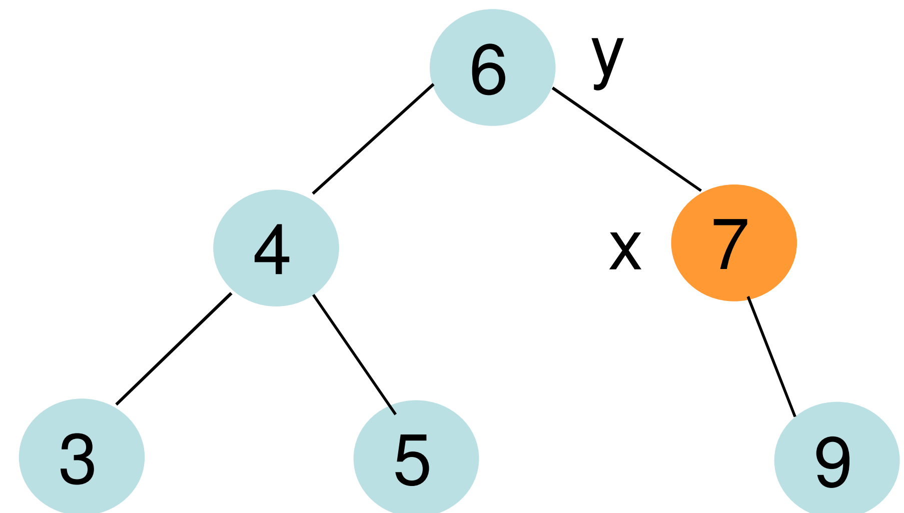
1. y← **nil**; x ← root[T]
2. **while** x≠**nil do**
3. y ← x
4. **if** key[z] < key[x] **then** x ← lc[x]
5. **else** x ← rc[x]
6. p[z] ← y
7. **if** y=**nil then** root[T] ← z
8. **else**
9. **if** key[z]< key[y] **then** lc[y] ← z
10. **else** rc[y] ← z



Binäre Suchbäume

Einfügen(T,z)

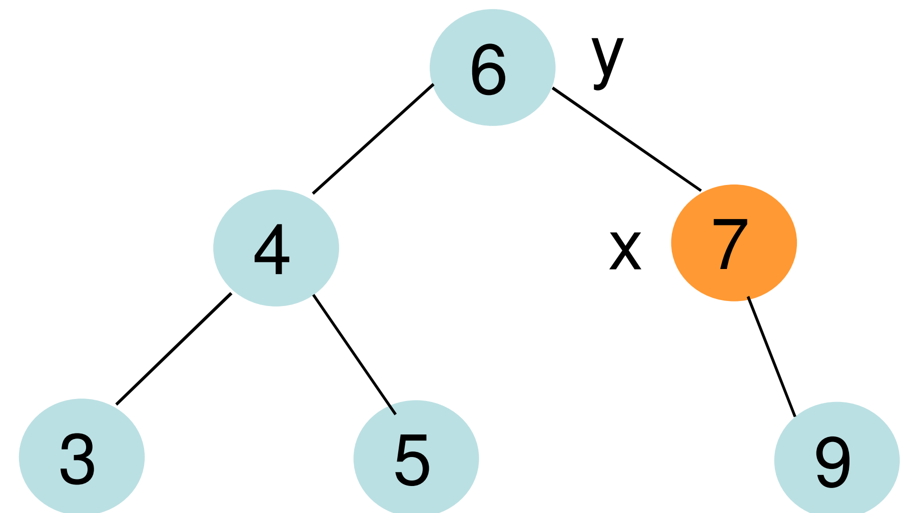
1. y← **nil**; x ← root[T]
2. **while** x≠**nil do**
3. y ← x
4. **if** key[z] < key[x] **then** x ← lc[x]
5. **else** x ← rc[x]
6. p[z] ← y
7. **if** y=**nil then** root[T] ← z
8. **else**
9. **if** key[z]< key[y] **then** lc[y] ← z
10. **else** rc[y] ← z



Binäre Suchbäume

Einfügen(T,z)

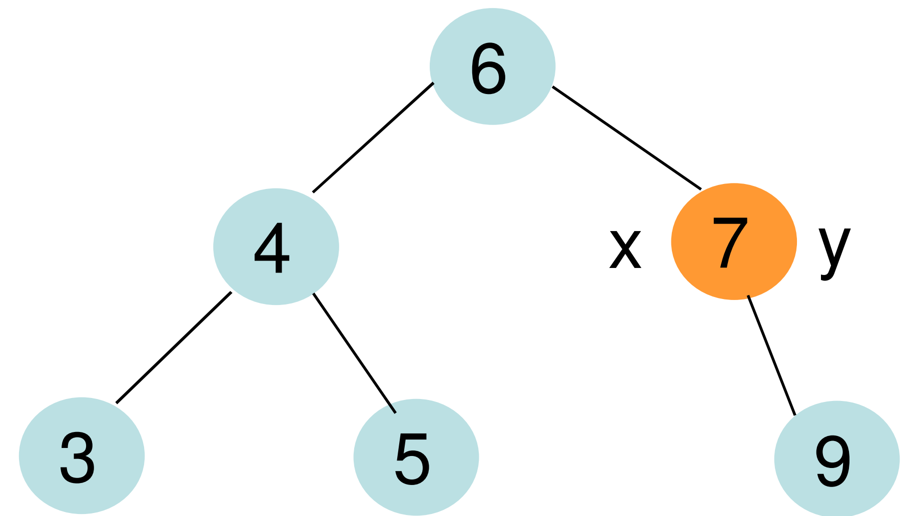
1. y← **nil**; x ← root[T]
2. **while** x≠**nil do**
3. y ← x
4. **if** key[z] < key[x] **then** x ← lc[x]
5. **else** x ← rc[x]
6. p[z] ← y
7. **if** y=**nil then** root[T] ← z
8. **else**
9. **if** key[z]< key[y] **then** lc[y] ← z
10. **else** rc[y] ← z



Binäre Suchbäume

Einfügen(T,z)

1. y← **nil**; x ← root[T]
2. **while** x≠**nil** **do**
3. y ← x
4. **if** key[z] < key[x] **then** x ← lc[x]
5. **else** x ← rc[x]
6. p[z] ← y
7. **if** y=**nil** **then** root[T] ← z
8. **else**
9. **if** key[z]< key[y] **then** lc[y] ← z
10. **else** rc[y] ← z



Binäre Suchbäume

Einfügen(T,z)

```
1. y← nil; x ← root[T]
2. while x≠nil do
3.    y ← x
4.    if key[z] < key[x] then x ← lc[x]
5.    else x ← rc[x]
6. p[z] ← y
7. if y=nil then root[T] ← z
8. else
9.     if key[z]< key[y] then lc[y] ← z
10.    else rc[y] ← z
```

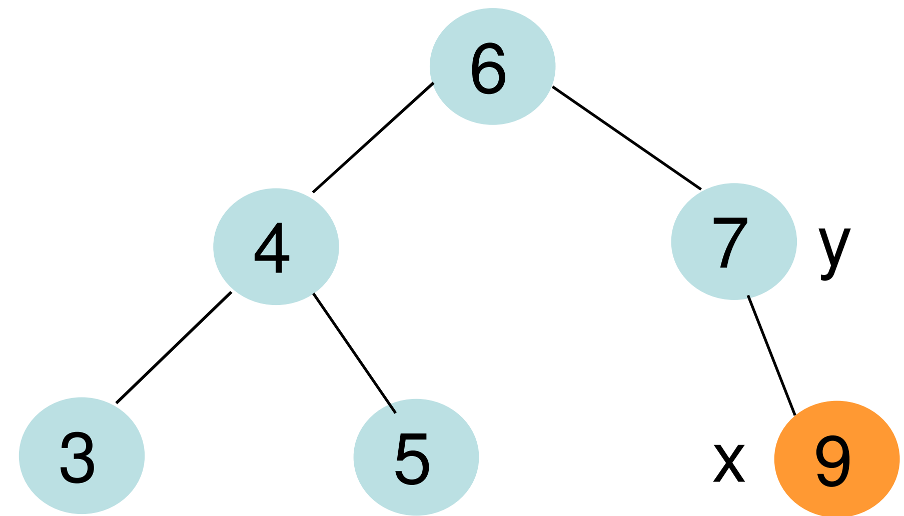
Einfügen(8)

6, 4, 3, 5, x 7 y, 9

Binäre Suchbäume

Einfügen(T,z)

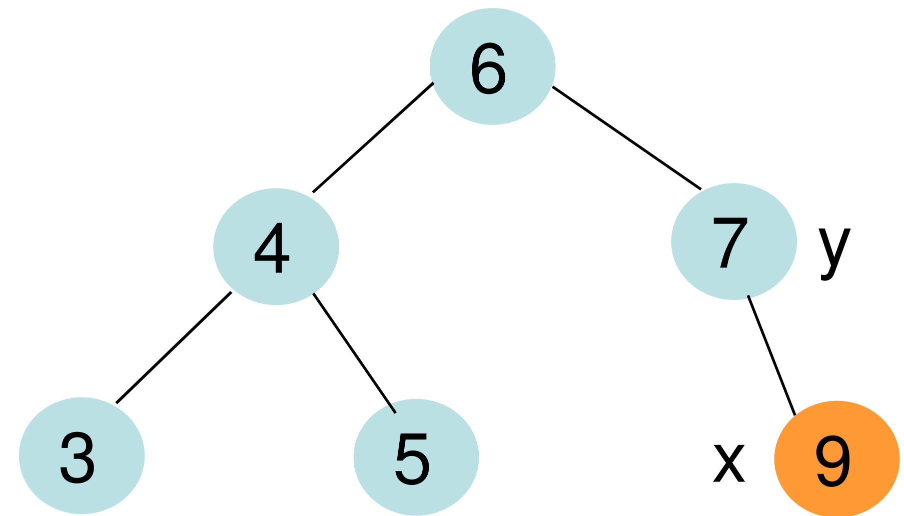
1. y← **nil**; x ← root[T]
2. **while** x≠**nil do**
3. y ← x
4. **if** key[z] < key[x] **then** x ← lc[x]
5. **else** x ← rc[x]
6. p[z] ← y
7. **if** y=**nil then** root[T] ← z
8. **else**
9. **if** key[z]< key[y] **then** lc[y] ← z
10. **else** rc[y] ← z



Binäre Suchbäume

Einfügen(T,z)

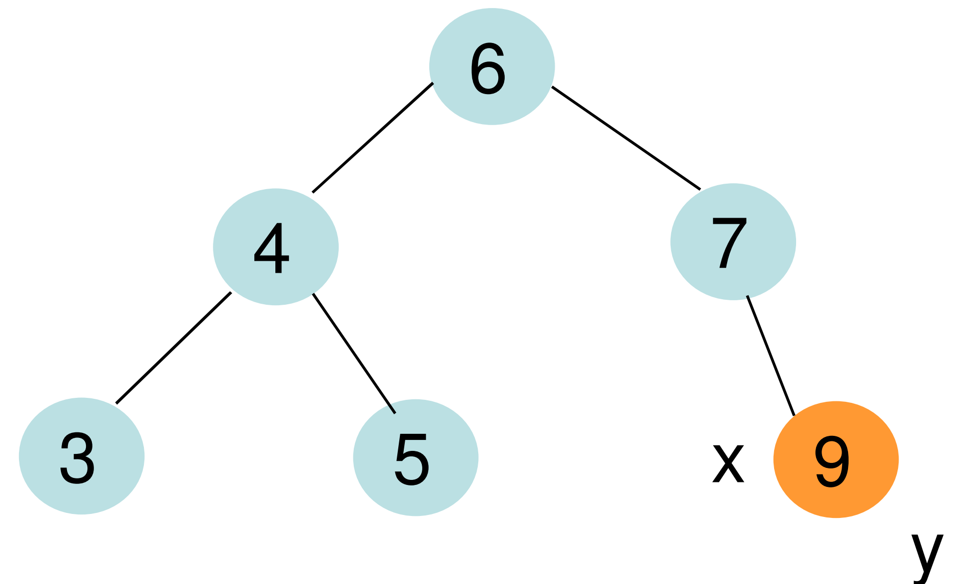
1. y← **nil**; x ← root[T]
2. **while** x≠**nil** **do**
3.   y ← x
4.   **if** key[z] < key[x] **then** x ← lc[x]
5.   **else** x ← rc[x]
6. p[z] ← y
7. **if** y=**nil** **then** root[T] ← z
8. **else**
9.   **if** key[z]< key[y] **then** lc[y] ← z
10.   **else** rc[y] ← z



Binäre Suchbäume

Einfügen(T,z)

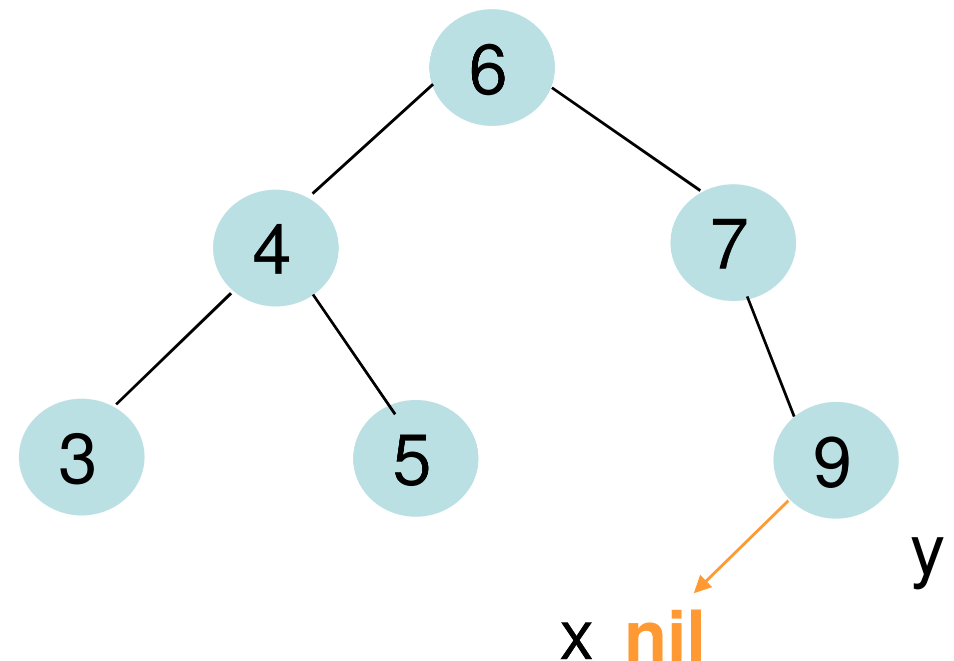
1. y← **nil**; x ← root[T]
2. **while** x≠**nil** **do**
3. y ← x
4. **if** key[z] < key[x] **then** x ← lc[x]
5. **else** x ← rc[x]
6. p[z] ← y
7. **if** y=**nil** **then** root[T] ← z
8. **else**
9. **if** key[z]< key[y] **then** lc[y] ← z
10. **else** rc[y] ← z



Binäre Suchbäume

Einfügen(T,z)

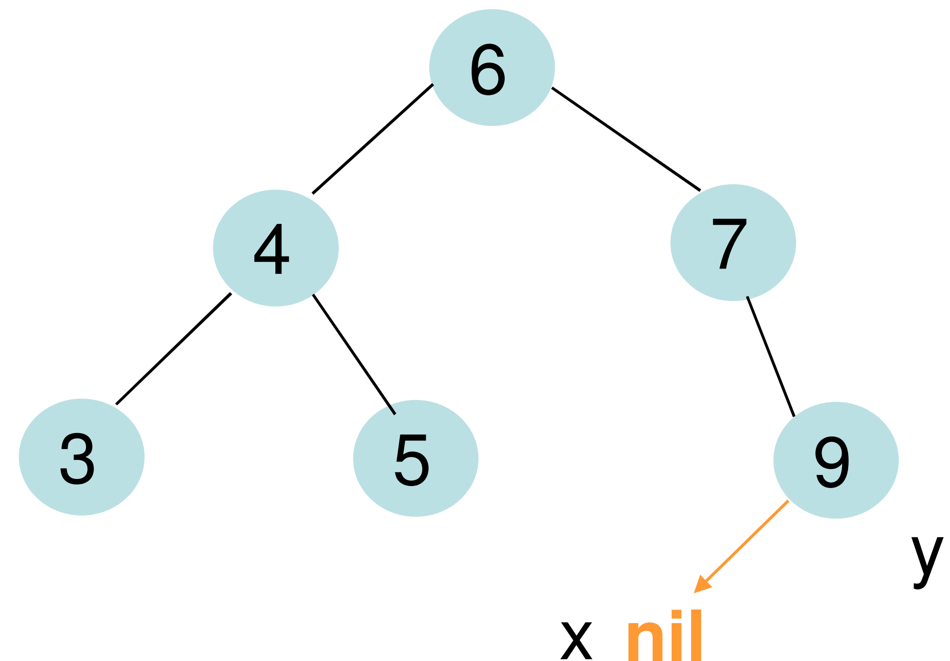
1. y← **nil**; x ← root[T]
2. **while** x≠**nil do**
3. y ← x
4. **if** key[z] < key[x] **then** x ← lc[x]
5. **else** x ← rc[x]
6. p[z] ← y
7. **if** y=**nil then** root[T] ← z
8. **else**
9. **if** key[z]< key[y] **then** lc[y] ← z
10. **else** rc[y] ← z



Binäre Suchbäume

Einfügen(T,z)

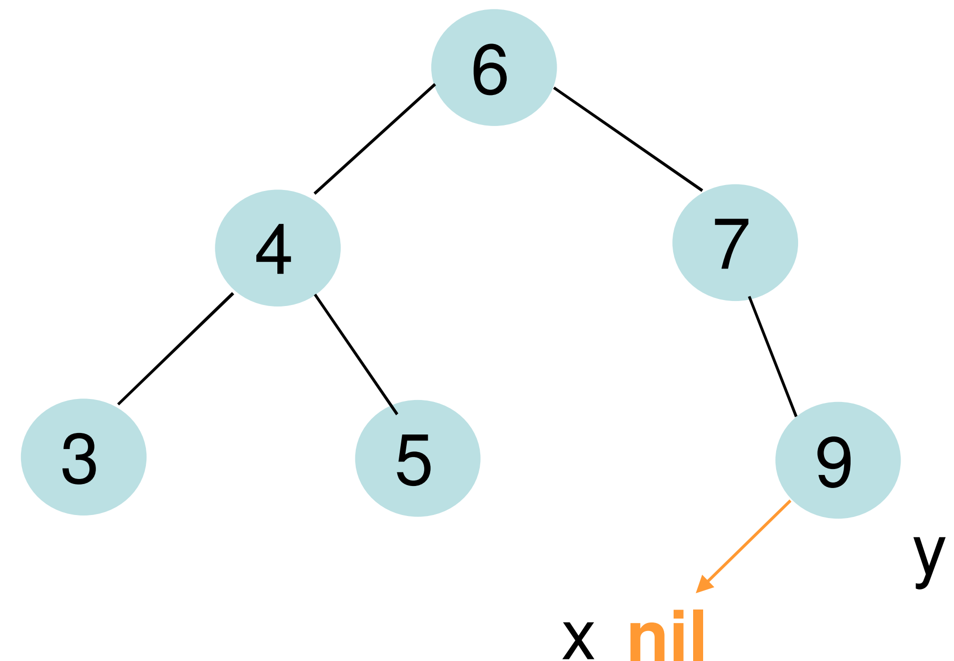
1. y← **nil**; x ← root[T]
2. **while** x≠**nil do**
3. y ← x
4. **if** key[z] < key[x] **then** x ← lc[x]
5. **else** x ← rc[x]
6. p[z] ← y
7. **if** y=**nil then** root[T] ← z
8. **else**
9. **if** key[z]< key[y] **then** lc[y] ← z
10. **else** rc[y] ← z



Binäre Suchbäume

Einfügen(T,z)

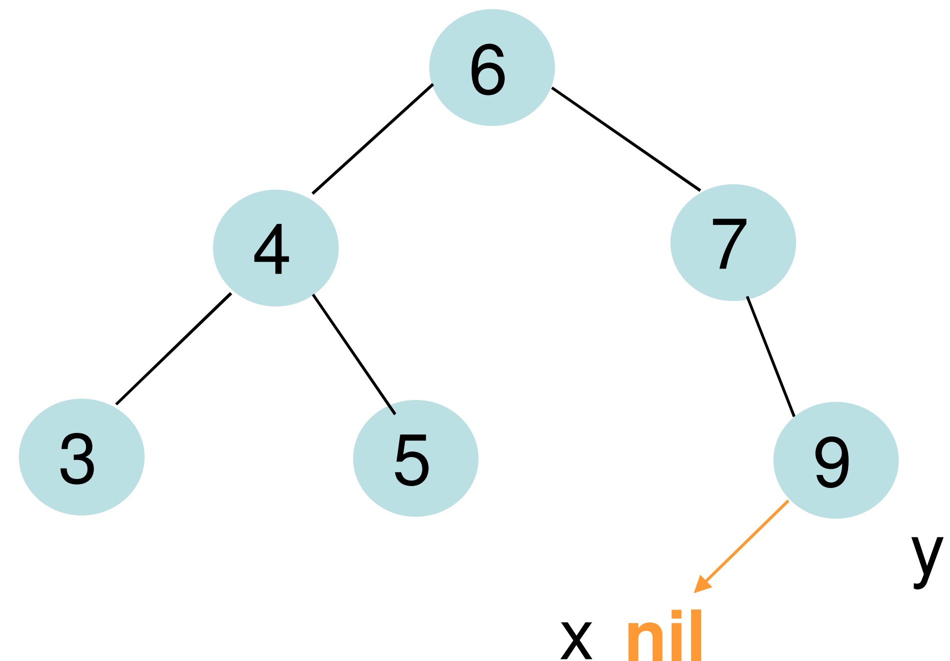
1. y← **nil**; x ← root[T]
2. **while** x≠**nil do**
3.   y ← x
4.   **if** key[z] < key[x] **then** x ← lc[x]
5.   **else** x ← rc[x]
6. p[z] ← y
7. **if** y=**nil then** root[T] ← z
8. **else**
9.   **if** key[z]< key[y] **then** lc[y] ← z
10.   **else** rc[y] ← z



Binäre Suchbäume

Einfügen(T,z)

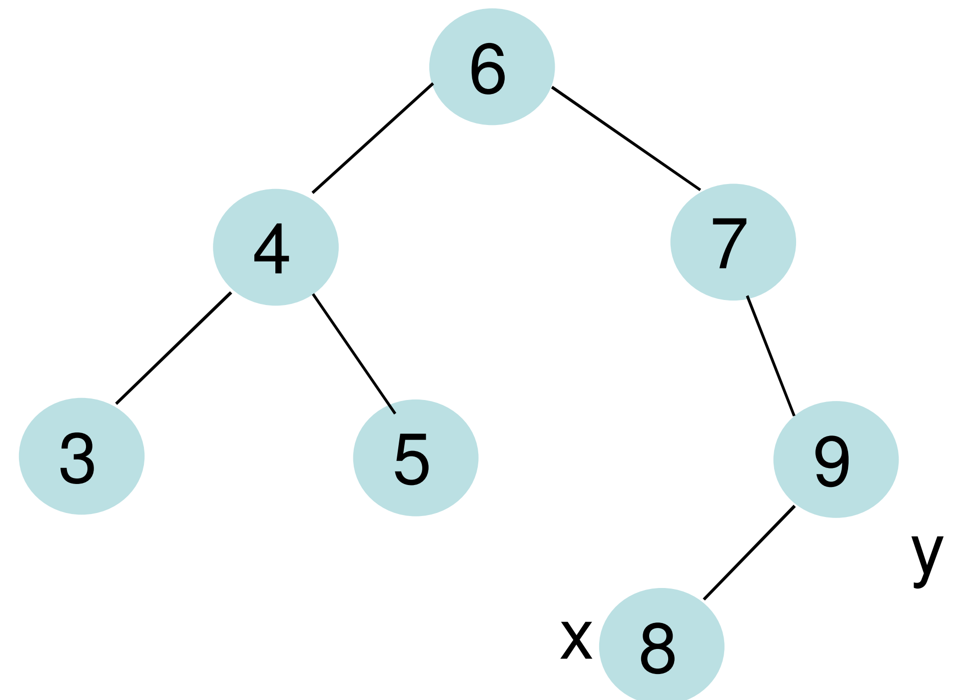
1. y← **nil**; x ← root[T]
2. **while** x≠**nil do**
3. y ← x
4. **if** key[z] < key[x] **then** x ← lc[x]
5. **else** x ← rc[x]
6. p[z] ← y
7. **if** y=**nil then** root[T] ← z
8. **else**
9. **if** key[z]< key[y] **then** lc[y] ← z
10. **else** rc[y] ← z



Binäre Suchbäume

Einfügen(T,z)

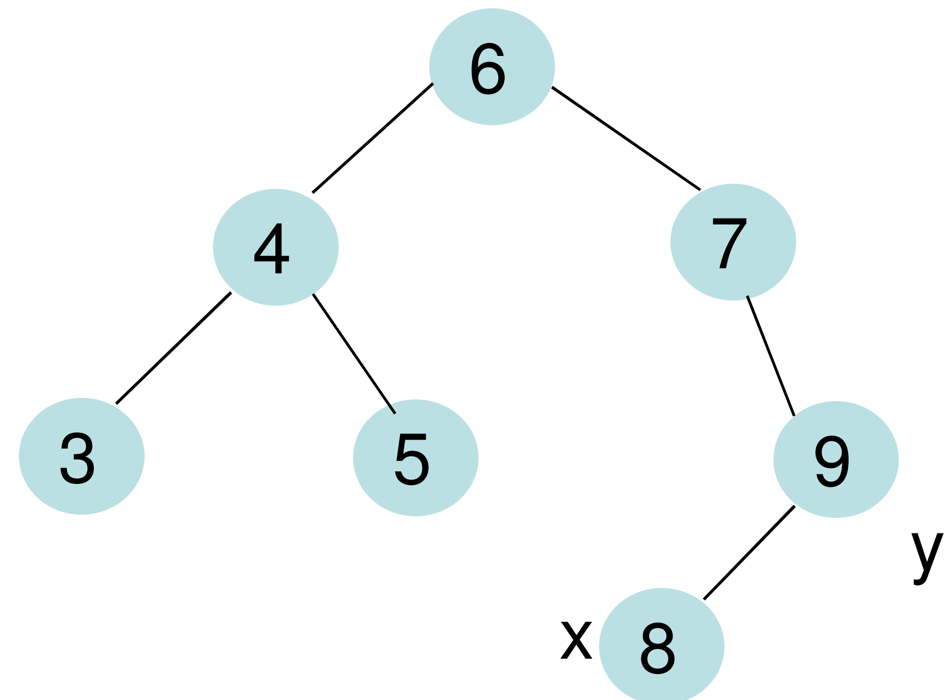
1. y← **nil**; x ← root[T]
2. **while** x≠**nil do**
3. y ← x
4. **if** key[z] < key[x] **then** x ← lc[x]
5. **else** x ← rc[x]
6. p[z] ← y
7. **if** y=**nil then** root[T] ← z
8. **else**
9. **if** key[z]< key[y] **then** lc[y] ← z
10. **else** rc[y] ← z



Binäre Suchbäume

Einfügen(T,z)

1. y← **nil**; x ← root[T]
2. **while** x≠**nil do**
3. y ← x
4. **if** key[z] < key[x] **then** x ← lc[x]
5. **else** x ← rc[x]
6. p[z] ← y
7. **if** y=**nil then** root[T] ← z
8. **else**
9. **if** key[z]< key[y] **then** lc[y] ← z
10. **else** rc[y] ← z



Binäre Suchbäume

Einfügen(T,z)

1. y← **nil**; x ← root[T]
2. **while** x≠**nil do**
3.   y ← x
4.   **if** key[z] < key[x] **then** x ← lc[x]
5.   **else** x ← rc[x]
6. p[z] ← y
7. **if** y=**nil then** root[T] ← z
8. **else**
9.   **if** key[z]< key[y] **then** lc[y] ← z
10.   **else** rc[y] ← z

Einfügen(8)

Laufzeit:

O(h)

Binäre Suchbäume

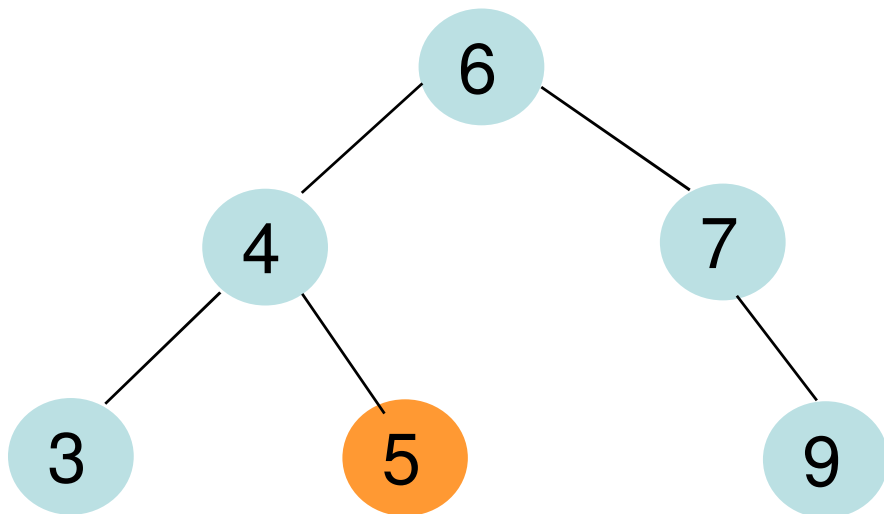
Löschen:

- 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

Binäre Suchbäume

Fall (a):

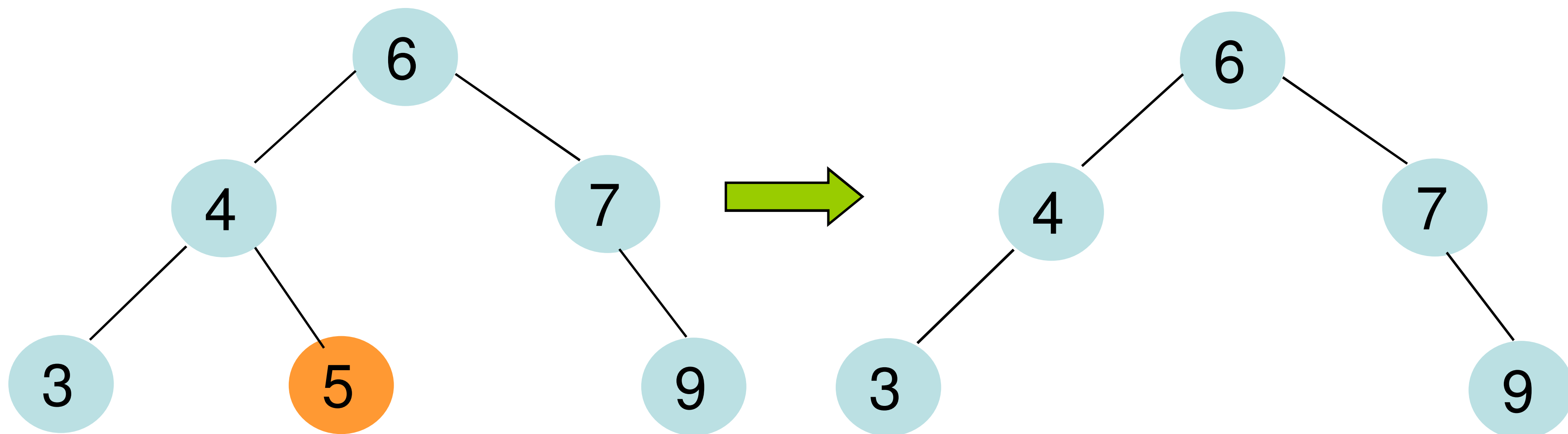
- zu löschendes Element z hat keine Kinder



Binäre Suchbäume

Fall (a):

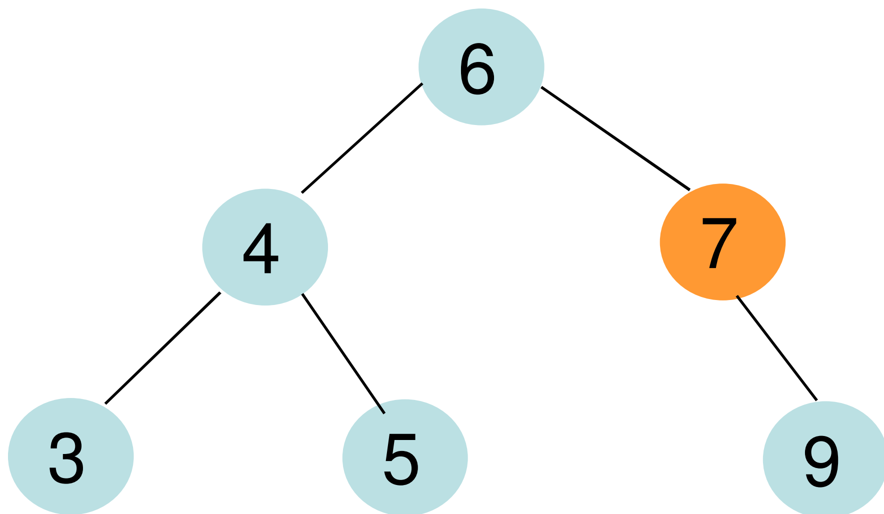
- zu löschendes Element z hat keine Kinder
- Entferne Element



Binäre Suchbäume

Fall (b):

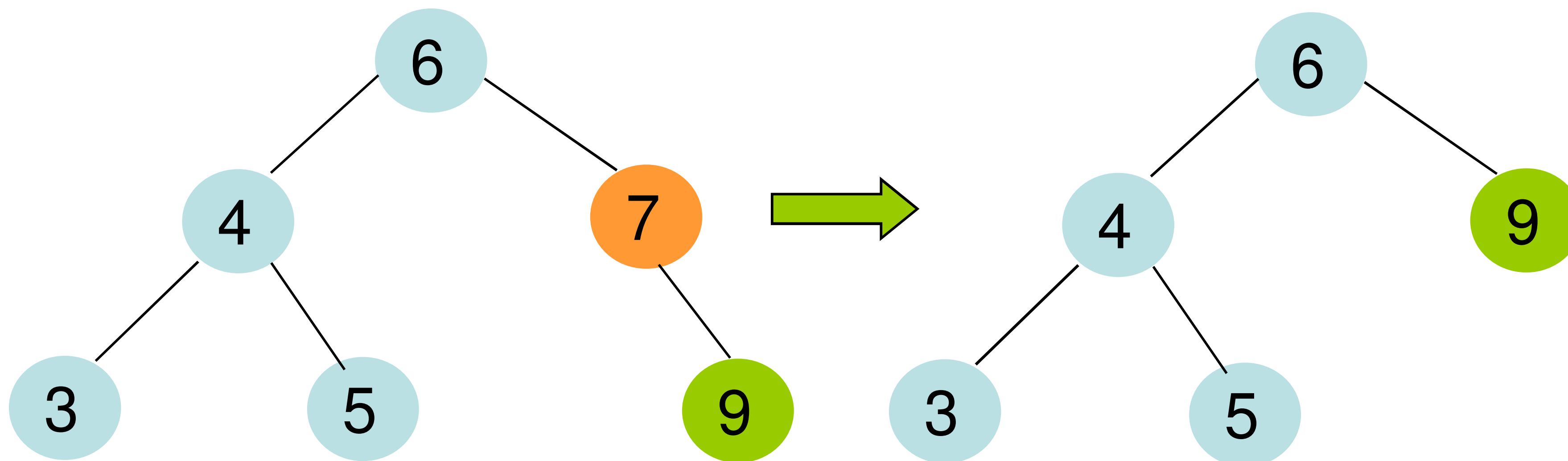
- zu löschendes Element z hat ein Kind



Binäre Suchbäume

Fall (b):

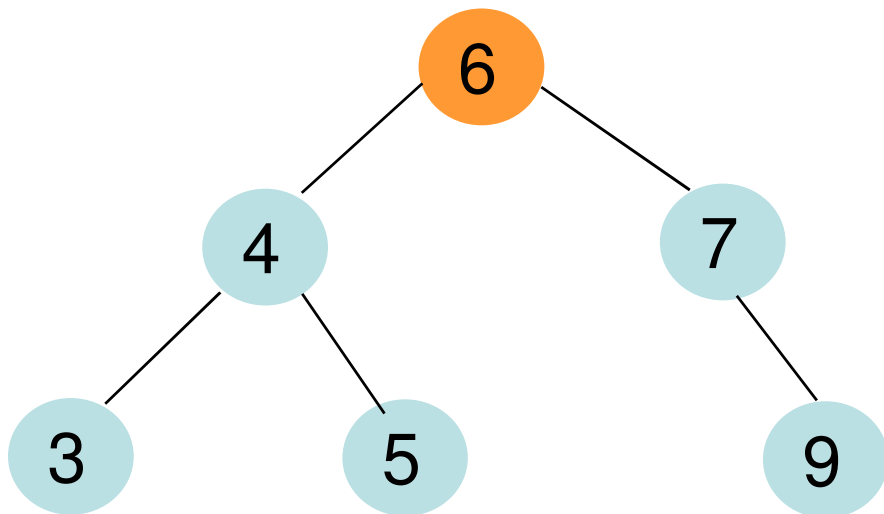
- zu löschendes Element z hat ein Kind
- Hänge der Unterbaum von z an den Vater von z



Binäre Suchbäume

Fall (c):

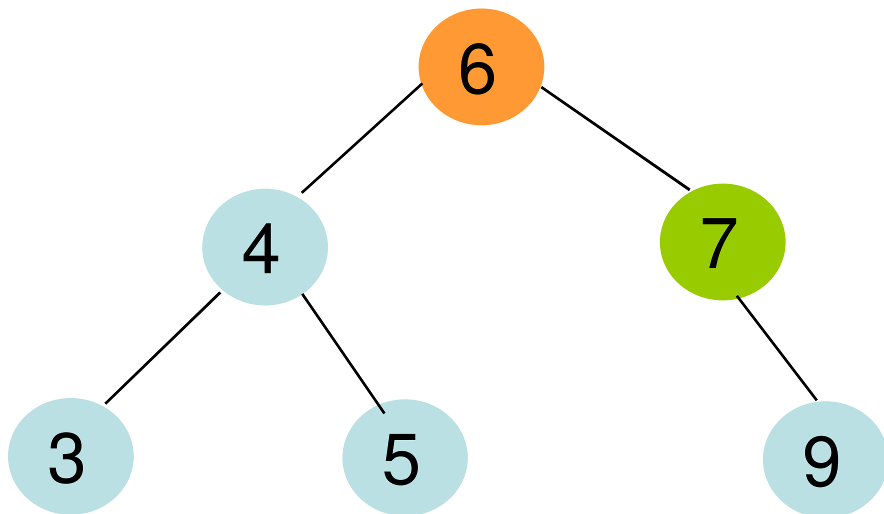
- zu löschendes Element z hat zwei Kinder



Binäre Suchbäume

Fall (c):

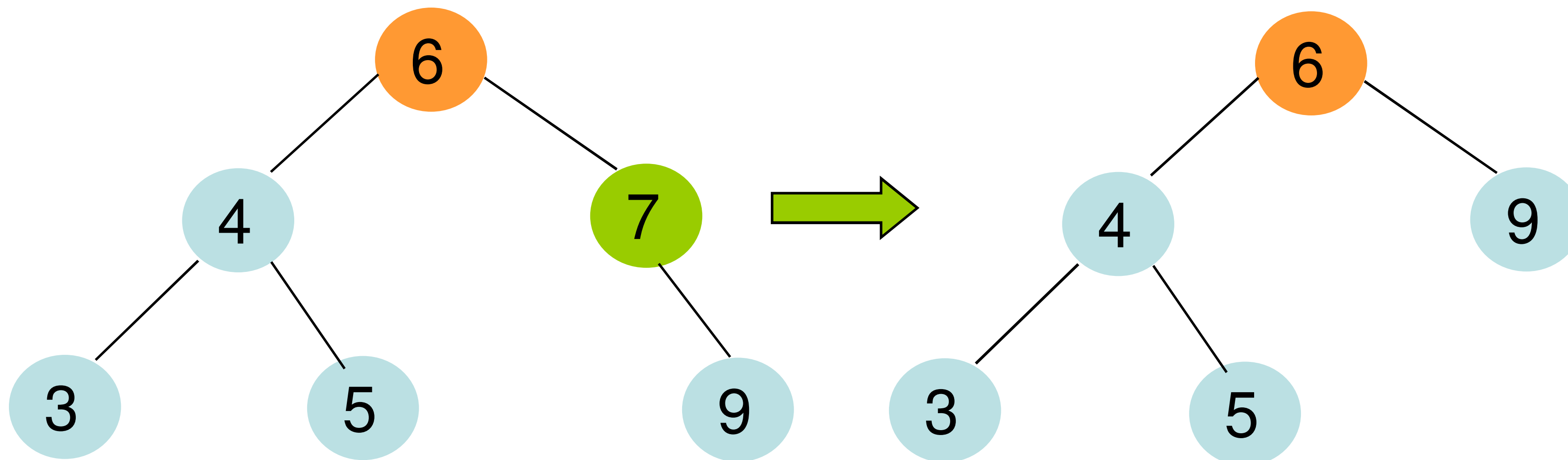
- zu löschendes Element z hat zwei Kinder
- Schritt 1: Bestimme Nachfolger von z



Binäre Suchbäume

Fall (c):

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

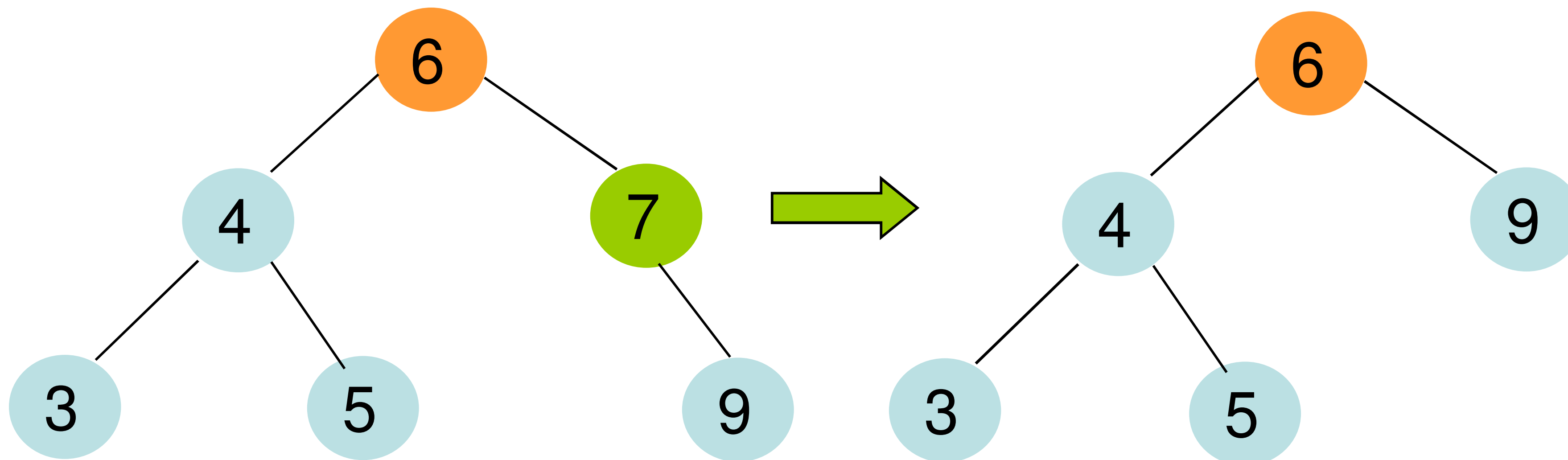


Binäre Suchbäume

Fall (c):

Nachfolger hat nur ein Kind!

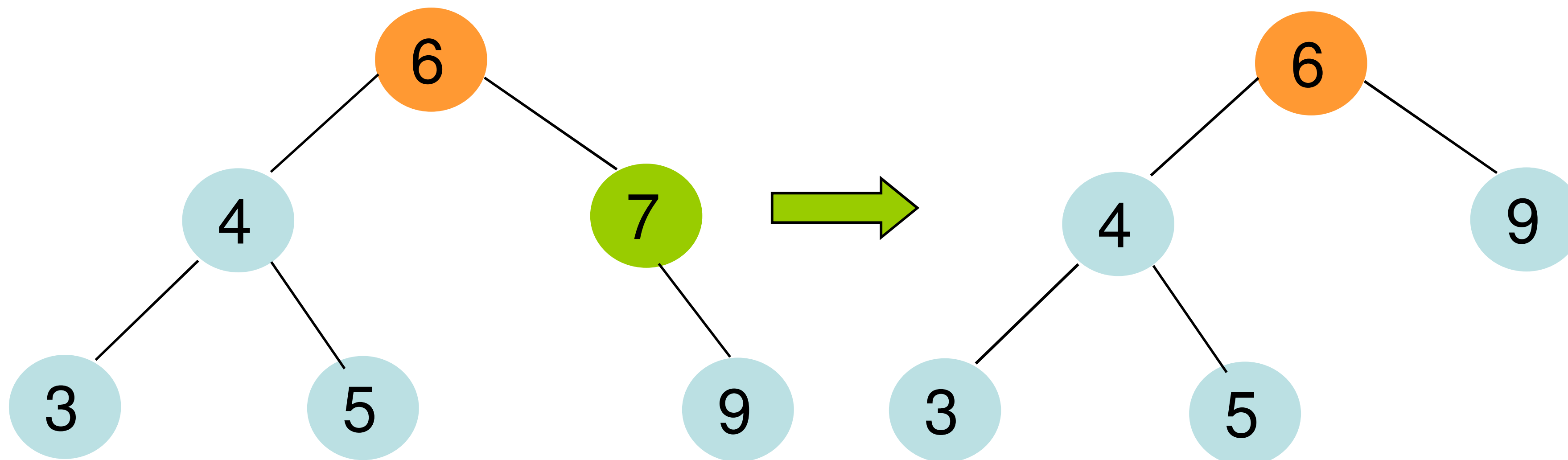
- zu löschendes Element z hat zw
- Schritt 1: Bestimme Nachfolger vo z
- Schritt 2: Entferne Nachfolger



Binäre Suchbäume

Fall (c):

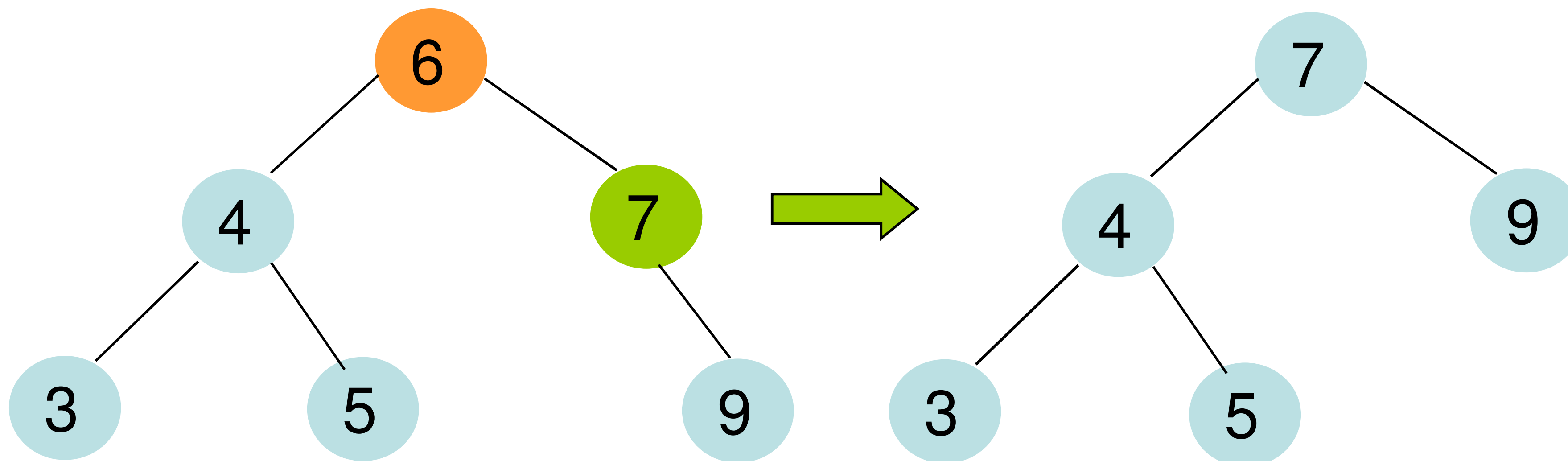
- zu löschendes Element z hat zwei Kinder
- Schritt 1: Bestimme Nachfolger von z
- Schritt 2: Entferne Nachfolger



Binäre Suchbäume

Fall (c):

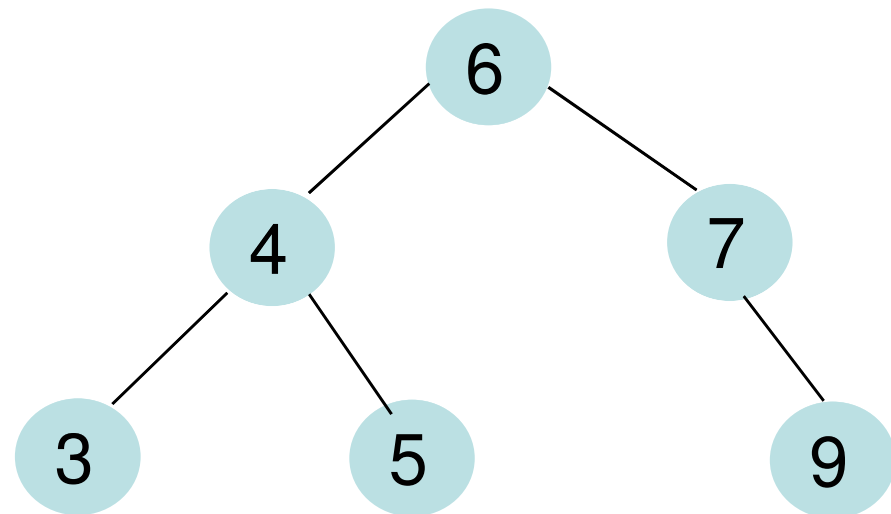
- zu löschendes Element z hat zwei Kinder
- Schritt 1: Bestimme Nachfolger von z
- Schritt 2: Entferne Nachfolger; ersetze durch Nachfolger



Binäre Suchbäume

Löschen(T,z)

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

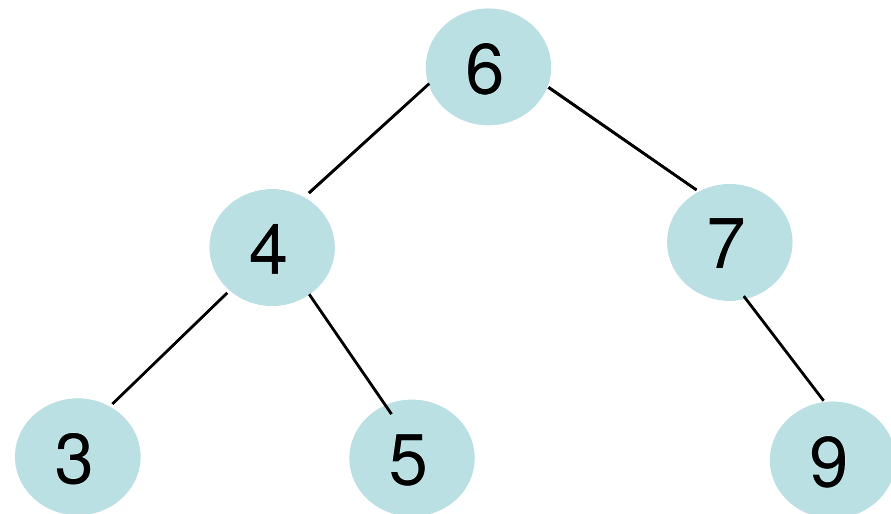


Binäre Suchbäume

Löschen(T,z)

1. **if** lc[z]=**nil** or rc[z]=**nil** **then** y $\leftarrow$ z
2. **else** y $\leftarrow$ NachfolgerSuche(z)
3. **if** lc[y] $\neq$ nil **then** x $\leftarrow$ lc[y]
4. **else** x $\leftarrow$ rc[y]
5. **if** x $\neq$ **nil** **then** p[x] $\leftarrow$ p[y]
6. **if** p[y]=**nil** **then** 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. **if** y $\neq$ z **then** key[z] $\leftarrow$ key[y]
10. **return** y

Löschen(6)



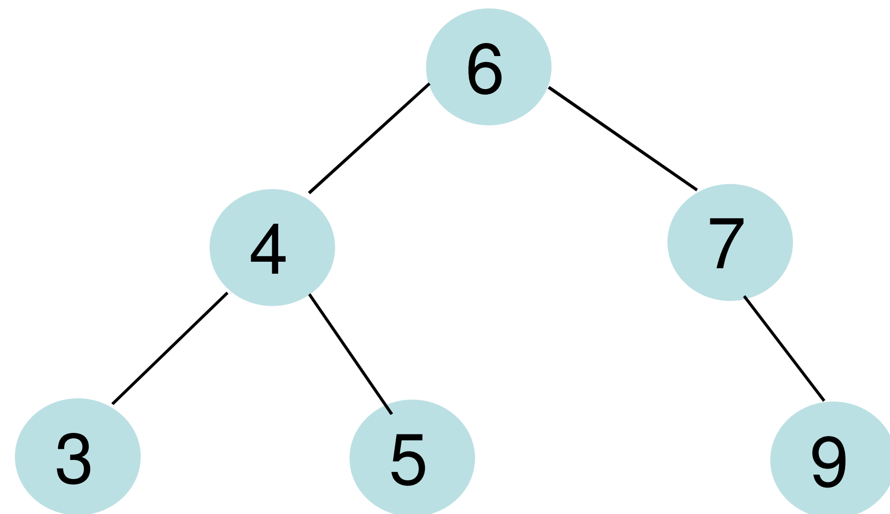
Binäre Suchbäume

Referenz auf z wird übergeben!

Löschen(T,z)

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

Löschen(6)



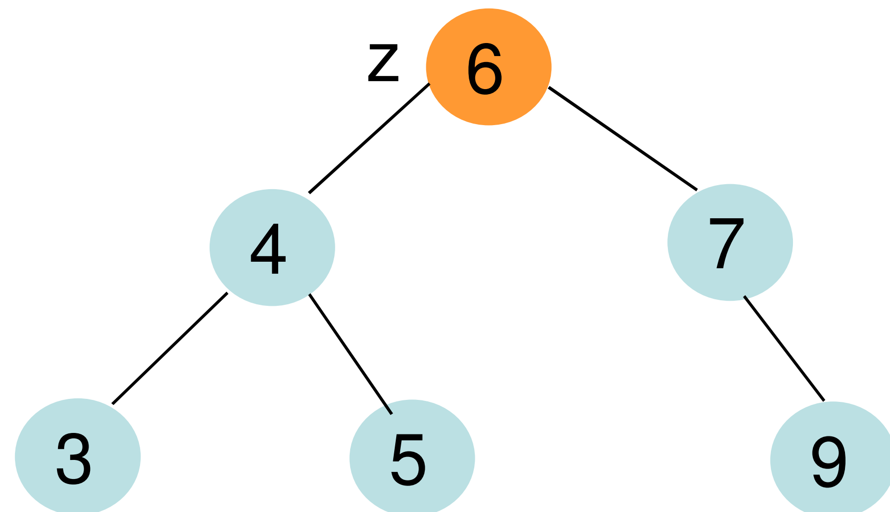
Binäre Suchbäume

Löschen(T,z)

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

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

Löschen(6)



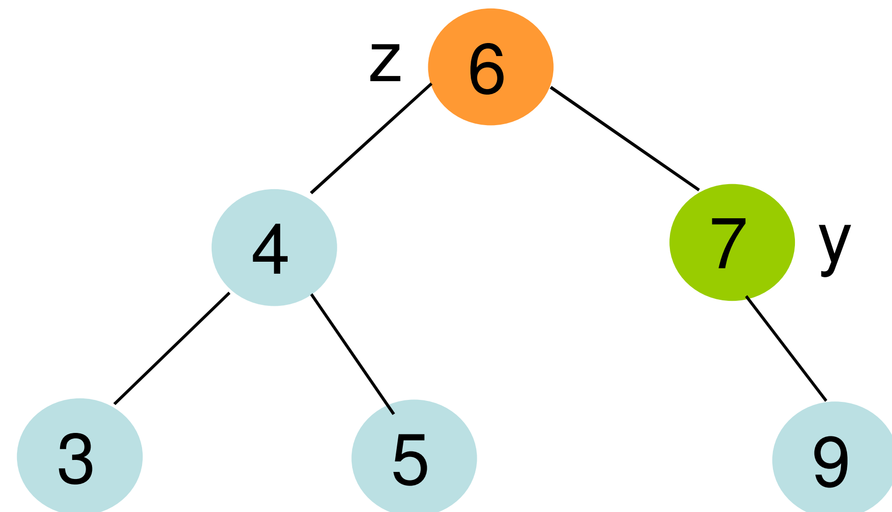
Binäre Suchbäume

Löschen(T,z)

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

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

Löschen(6)



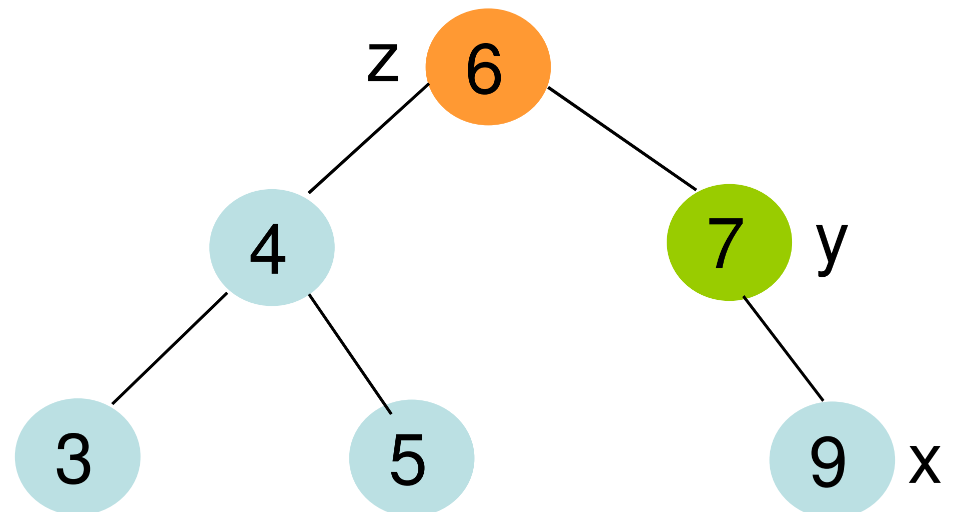
Binäre Suchbäume

Löschen(T,z)

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

Bestimme den Nachfolger von y (falls dieser existiert).

Löschen(6)



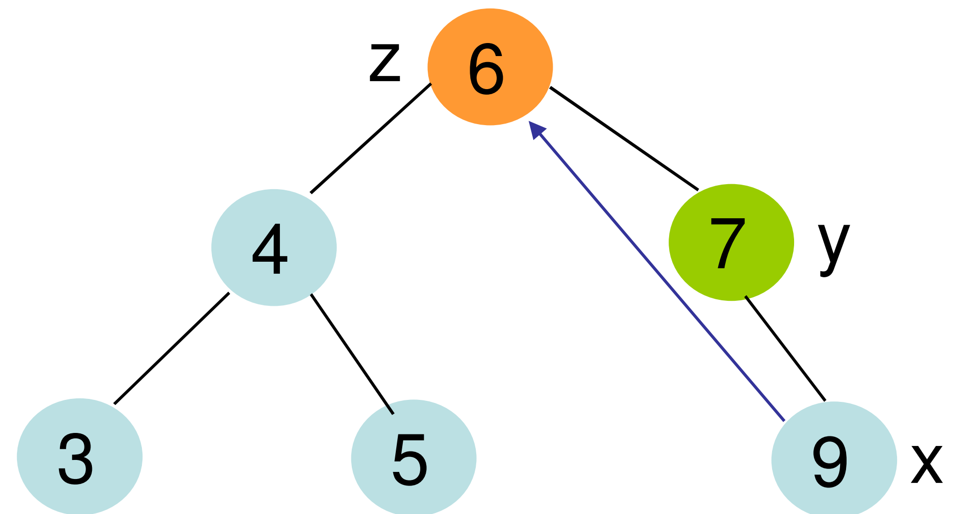
Binäre Suchbäume

Löschen(T,z)

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

Aktualisiere Vaterzeiger von x

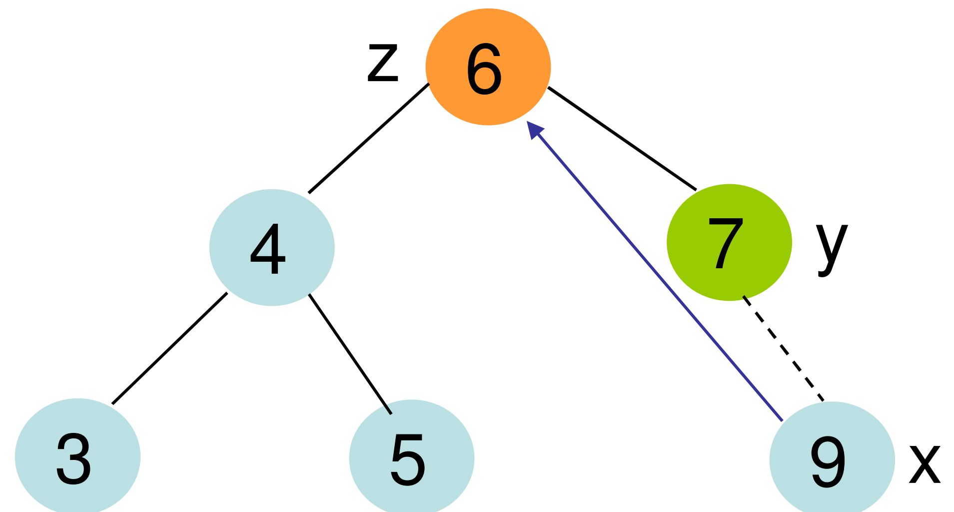
en(6)



Binäre Suchbäume

Löschen(T,z)

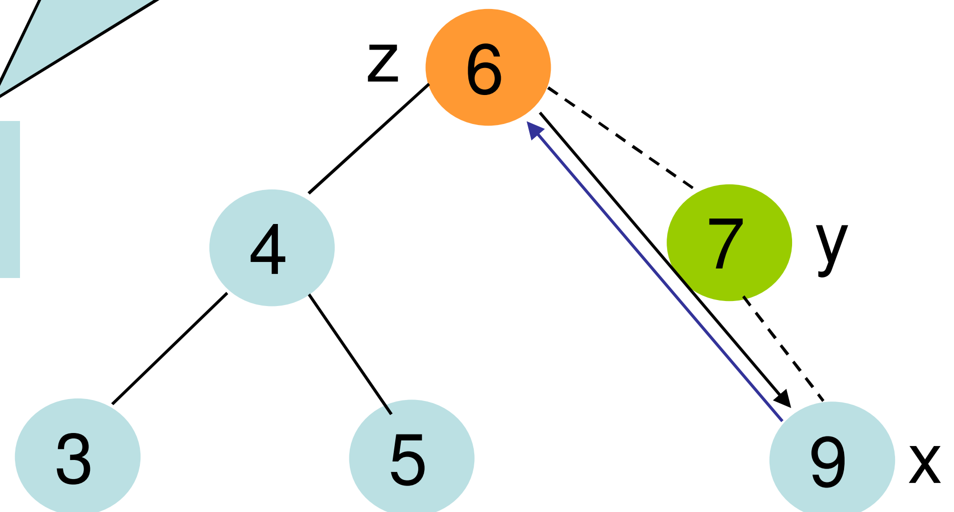
1. **if** lc[z]=**nil** or rc[z]=**nil then** y ← z
2. **else** y ← NachfolgerSuche(z)
3. **if** lc[y] ≠ nil **then** x ← lc[y]
4. **else** x ← rc[y]
5. **if** x ≠ **nil then** p[x] ← p[y]
6. **if** p[y]=**nil then** root[T] ← x
7. **else if** y=lc[p[y]] **then** lc[p[y]] ← x
8. **else** rc[p[y]] ← x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



Binäre Suchbäume

Löschen(T,z)

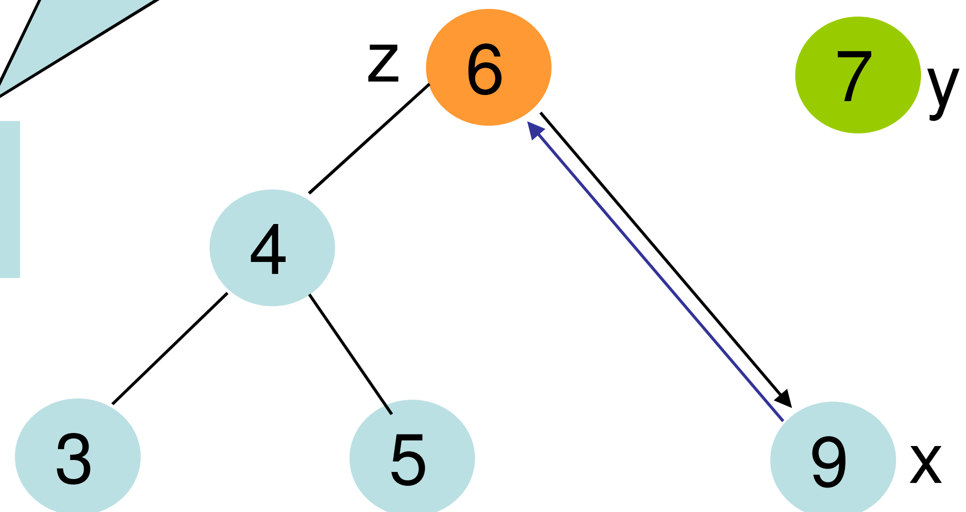
1. **if** lc[z]=**nil** or rc[z]=**nil** **then** y ← z
2. **else** y ← NachfolgerSuche(z)
3. **if** lc[y] ≠ nil **then** x ← lc[y]
4. **else** x ← rc[y]
5. **if** x ≠ **nil** **then** p[x] ← p[y]
6. **if** p[y]=**nil** **then** root[T] ← x
7. **else if** y=lc[p[y]] **then** lc[p[y]] ← x
8. **else**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



Binäre Suchbäume

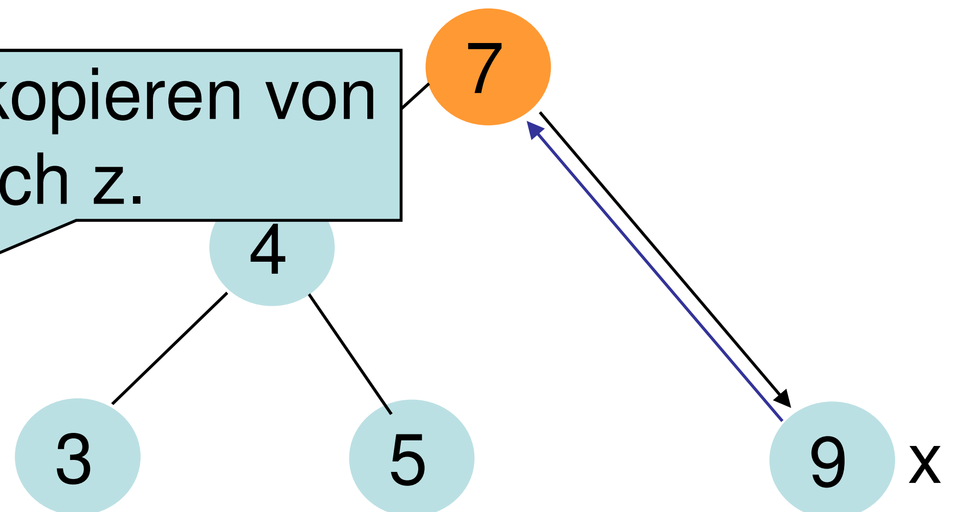
Löschen(T,z)
1. **if** lc[z]=**nil** or rc[z]=**nil then** y ← z
2. **else** y ← NachfolgerSuche(z)
3. **if** lc[y] ≠ nil **then** x ← lc[y]
4. **else** x ← rc[y]
5. **if** x ≠ **nil then** p[x] ← p[y]
6. **if** p[y]=**nil then** root[T] ← x
7. **else if** y=lc[p[y]] **then** lc[p[y]] ← x
8. **else**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

Das rechte Kind von z wird x.



Binäre Suchbäume

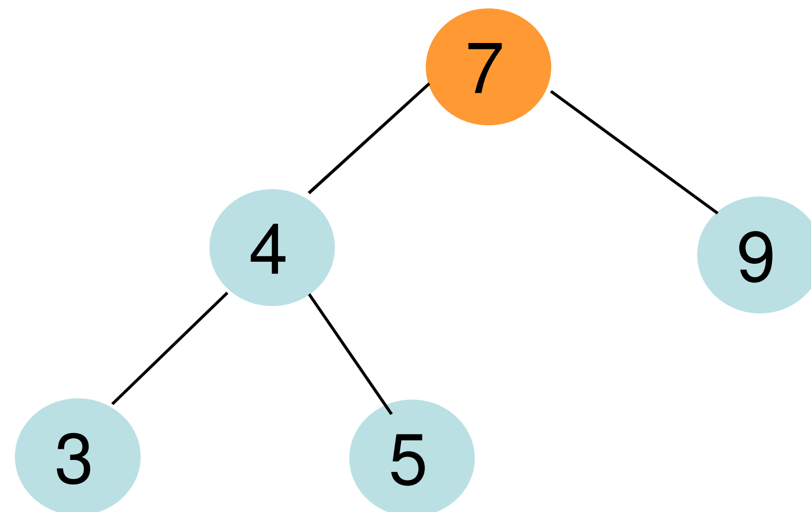
Löschen(T,z)
1. **if** lc[z]=**nil** or rc[z]=**nil** **then** y ← z
2. **else** y ← NachfolgerSuche(z)
3. **if** lc[y] ≠ nil **then** x ← lc[y]
4. **else** x ← rc[y]
5. **if** x ≠ **nil then** p[x] ← p[y]
6. **if** p[y]=**nil then** root[T] ← x
7. **else if** y=lc[p[y]] **then** lc[p[y
8. **else** rc[p[y]]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



Binäre Suchbäume

Löschen(T,z)

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

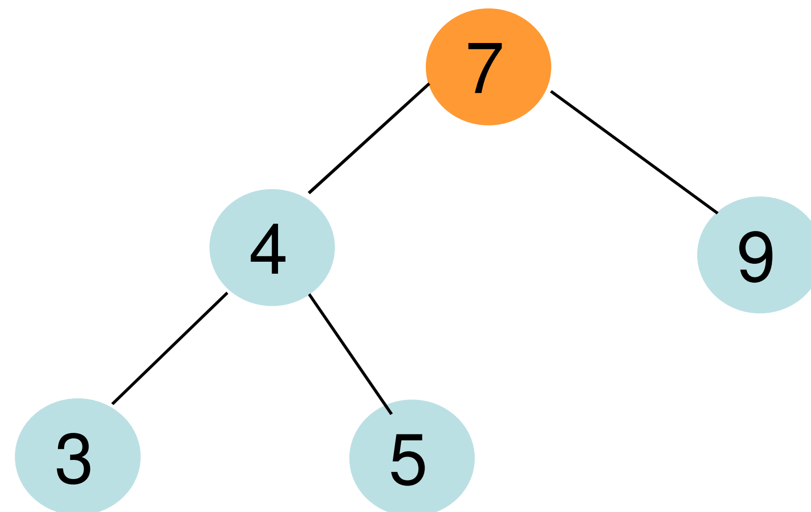


Binäre Suchbäume

Löschen(T,z)

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

Löschen(6)



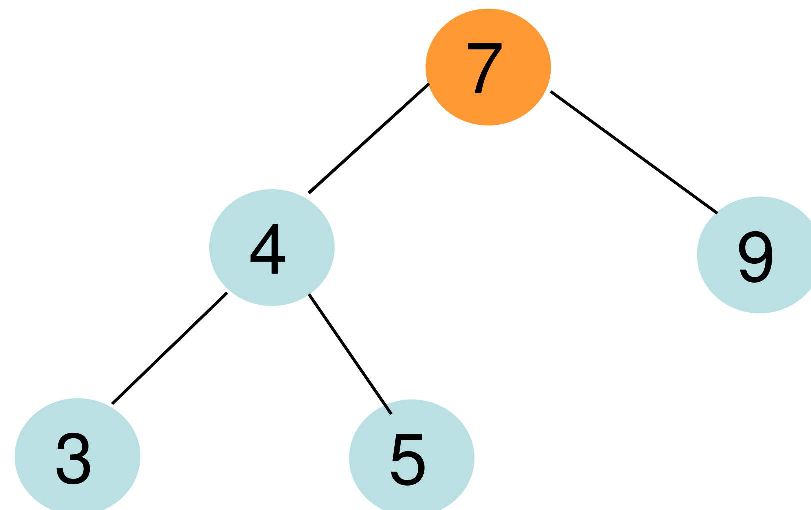
Binäre Suchbäume

Löschen(T,z)

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

Laufzeit O(h)

Löschen(6)



Binäre Suchbäume

Binäre Suchbäume:

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

Frage:

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