Bäume

Manfred Hauswirth | Open Distributed Systems | Einführung in die Programmierung, WS 23/24

Rückblick

VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine

VL 1 „Algorithmen, Pseudocode, Sortieren I“: Insertion Sort

VL 2 „Algorithmen, Pseudocode, Sortieren II“: Selection Sort, Bubble Sort, Count Sort

VL 3 „Laufzeit und Speicherplatz“: Laufzeitanalyse der vorgestellten Sortierverfahren

VL 4 „Einfache Datenstrukturen“: Arrays, verkettete Listen, Structs in C, Stack, Queue

VL 5 „Bäume“: Binärbäume, Baumtraversierung, Laufzeitanalyse Baumoperationen

VL 6 „Dateien in C“: Dateien, Dateisysteme, Verzeichnisse, Dateiverwaltung mit C

VL 7 „Teile und Herrsche I“: Einführung der algorithmischen Methode, Merge Sort

VL 8 „Korrektheitsbeweise“: Rechnermodel, Beispielbeweise

VL 9 „Prioritätenschlangen/Halden/Heaps “: Heap Sort, Binärer Heap, Heap Operationen

VL 10 „Fortgeschrittene Sortierverfahren“: Quick Sort, Radix Sort

VL 11 „AVL Bäume“: Definition, Baumoperationen, Traversierung

VL 12 „Teile und Herrsche II“: Generalisierung des algorithmischen Prinzips, Mastertheorem

VL 13 „Q & A“: Offene Vorlesung/Wiederholung

Bäume

- Ein grundlegendes Problem
  - Speicherung von Datensätzen
  - Listen oft nicht ausreichend, da O(n) zu langsam ist (suchen, einfügen, löschen, etc.).
- Beispiele
  - Stammbaum
  - Dateisysteme
  - Entscheidungsbäume
  - Suchbäume, z.B. für Lexikon, Daten
  - Rekursionsbäume
- Anforderungen
  - Schneller Zugriff, d.h. schneller als O(n)
  - Schnelles Einfügen neuer Datensätze
  - Schnelles Löschen bestehender Datensätze

Zugriff auf Daten

- Zugriff auf Daten
  - Jedes Datum (Objekt) hat einen Schlüssel
  - Eingabe des Schlüssels liefert Datensatz
  - Schlüssel sind vergleichbar (es gibt eine totale Ordnung auf den Schlüsseln)

- Beispiel
  - Lexika (Begriff, Erläuterung)
  - Schlüssel: Begriff
  - Totale Ordnung: Lexikographische Ordnung

Datenzugriff

- Problem:
  - Gegeben seien n Objekte O_1, ..., O_n mit zugehörigen Schlüsseln $s(O_i)$

- Operationen:
  - Suche(x): Ausgabe O mit Schlüssel s(O) = x, sonst **nil (NULL)**, falls kein Objekt mit Schlüssel x in der Datenmenge vorhanden ist
  - Einfügen(O): Einfügen von Objekt O in die Datenmenge
  - Löschen(O): Löschen von Objekt O aus der Datenmenge

Datenstrukturen

- Bisher kennen wir folgende grundlegenden Datenstrukturen
  - Felder (Arrays)
  - Sortierte Arrays
  - Dynamische Arrays
  - Einfach/doppelt verkettete Listen

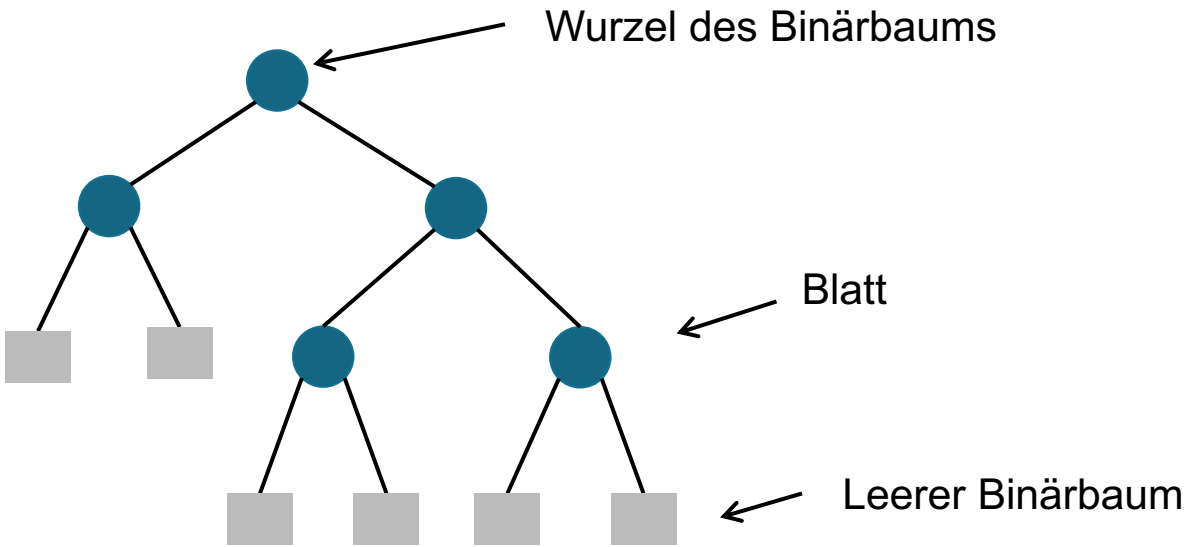
- Diskussion
  - Alle Datenstrukturen haben Nachteile
  - Listen helfen beim Speichermanagement
  - Sortierung hilft bei Suche ist aber teurer, aufrecht zu erhalten

Binärbaum

- 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 $\{v\} \cup V_1 \cup V_2$.
    T1 heißt linker Unterbaum von v und T_2 heißt rechter Unterbaum von v.

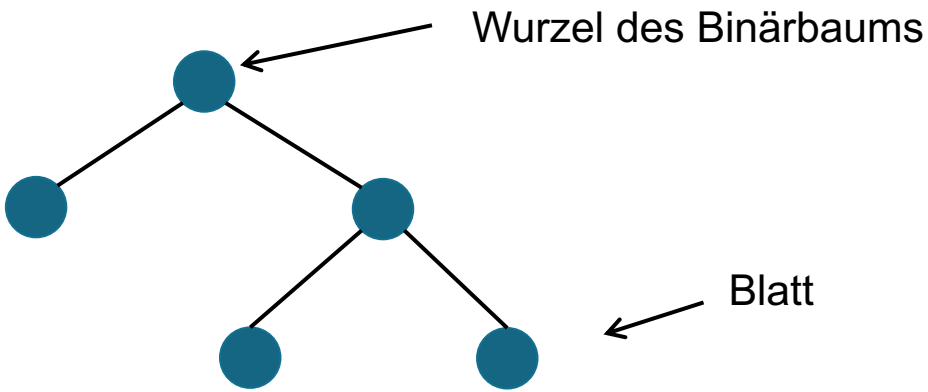
Binärbaum

- Darstellung von Binärbäumen



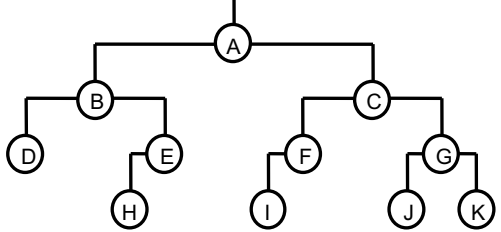
Binärbaum

- Darstellung von Binärbäumen



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

Binärbaum Begriffe



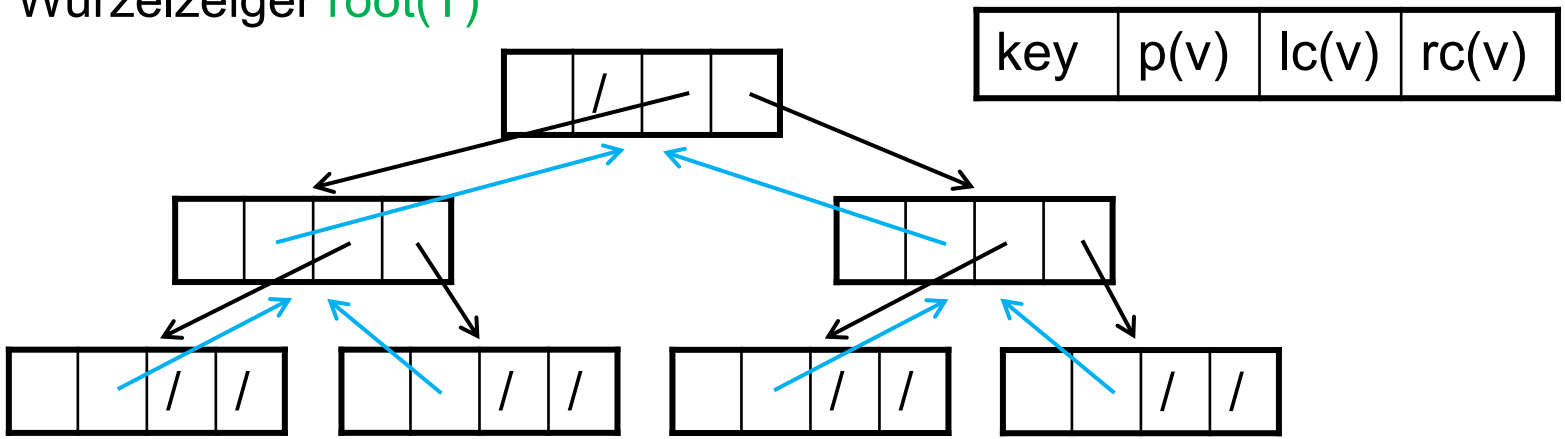
Technische Universität Berlin

Begriff	**Erläuterung**	**Beispiel**
Vorgänger (*predecessor*)	-	A ist Vorgänger von B
Nachfolger (*successor*)	-	B ist Nachfolger von A
Wurzel (*root*)	kein Vorgänger	A
Blatt (*leaf*)	kein Nachfolger	D, H, I, J, K
interner Knoten (*inner node*)	alle Nachfolger besetzt	A, B, C, G
Randknoten (*boundary node*)	nicht alle Nachfolger besetzt	D, E, F, H, I, J, K
Pfad (*path*)	Knotenfolge von einem Anfangs- bis zum Endknoten	Pfad von der Wurzel nach F: A,C,F
Pfadlänge (*path length*)	Anzahl der Kanten des Pfads	Pfadlänge von A nach F: 2
Tiefe eines Knotens (*depth*)	Pfadlänge zur Wurzel	Tiefe von F: 2
Höhe eines Baumes (*height*)	Größte Tiefe	3
Höhe eines Knotens *v*	Höhe des Teilbaums von *v*	Höhe von F: 1

Open Distributed Sytems
ODS

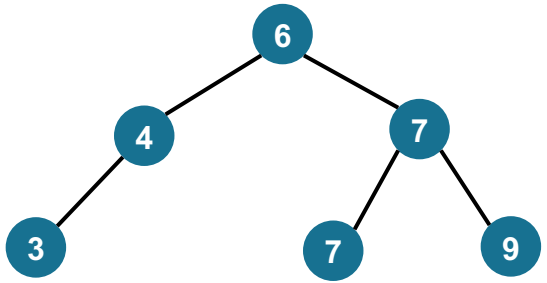
Binärbaum

- Binärbäume (Darstellung im Rechner)
  - Schlüssel key und ggf. weitere Daten
  - Zeiger lc(v) bzw. rc(v) auf linkes bzw. rechtes Kind von v
  - Elternzeiger p(v) auf Vater/Mutter von v (blau)
  - Wurzelzeiger root(T)



Binäre Suchbäume

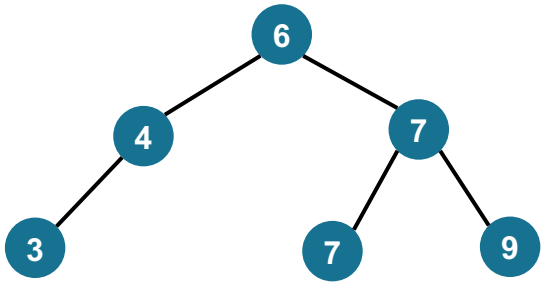
- Binäre Suchbäume
  - Verwende Binärbaum
  - Speichere Schlüssel „geordnet“

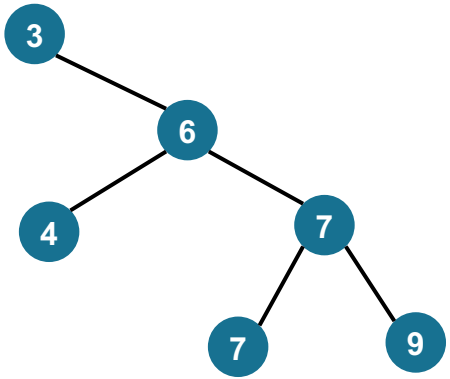


- Binäre Suchbaumeigenschaft:
  - Sei x ein Knoten im binären Suchbaum
  - Ist y ein Knoten im linken Unterbaum von x, dann gilt key(y) ≤ key(x)
  - Ist y ein Knoten im rechten Unterbaum von x, dann gilt key(y) > key(x)

Binäre Suchbäume

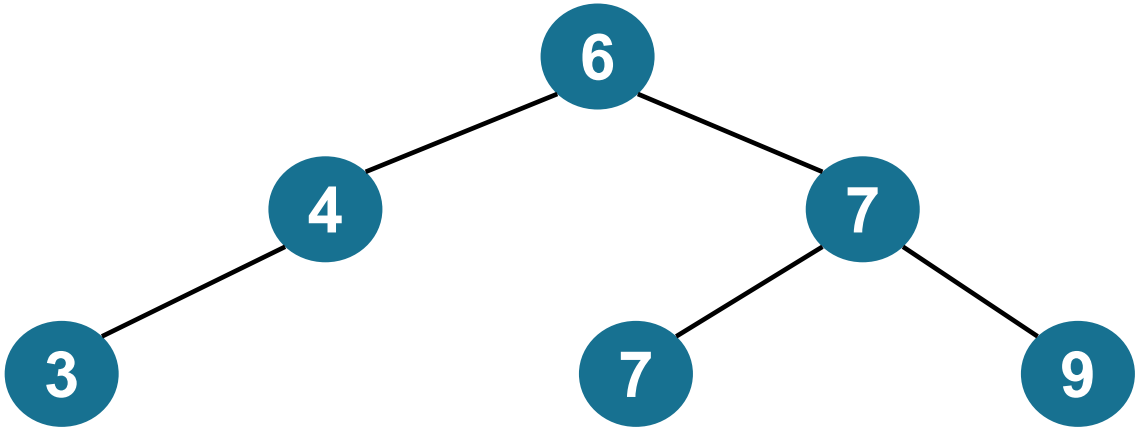
- Unterschiedliche Suchbäume
  - Schlüsselmenge 3,4,6,7,7,9
  - Wir erlauben mehrfache Vorkommen desselben Schlüssels





Binäre Suchbäume – Durchlaufen

- Ausgabe aller Schlüssel
  - Gegeben binärer Suchbaum
  - Wie kann man alle Schlüssel aufsteigend sortiert in O(n) Zeit ausgeben?



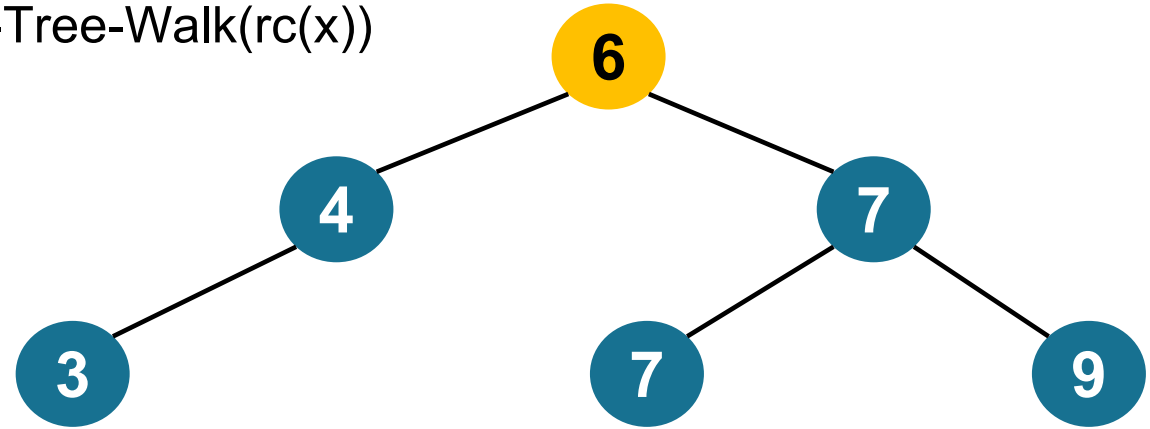
Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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))



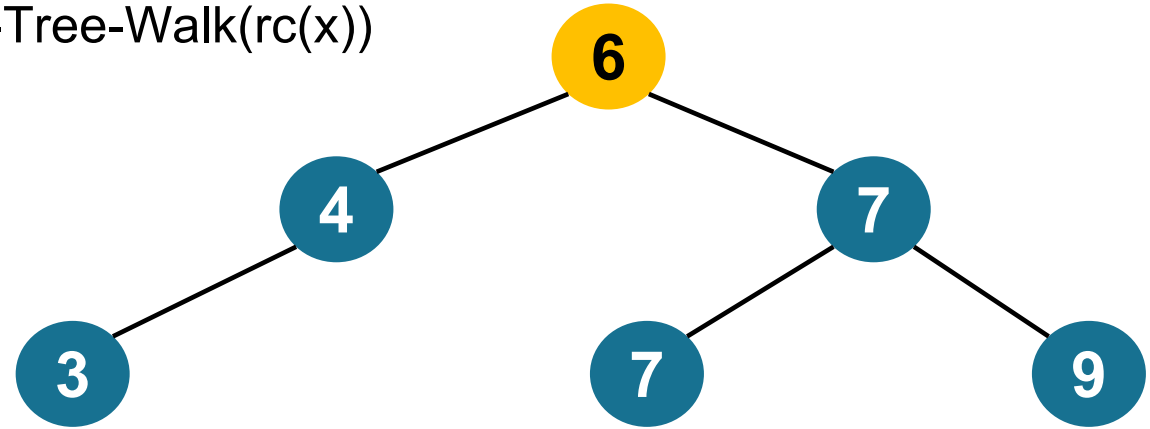
Open Distributed Sytems
ODS

Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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))



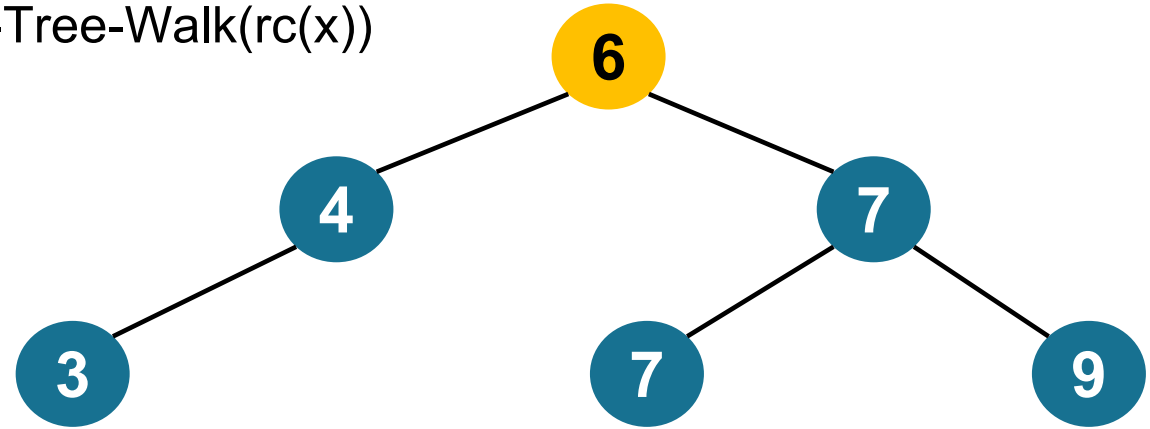
Open Distributed Sytems ODS

Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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))



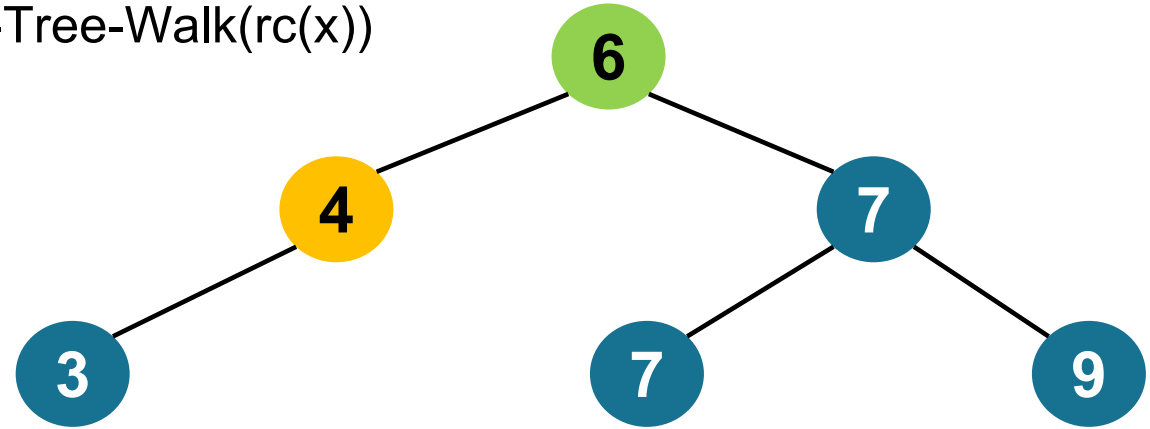
Open Distributed Sytems ODS

Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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))



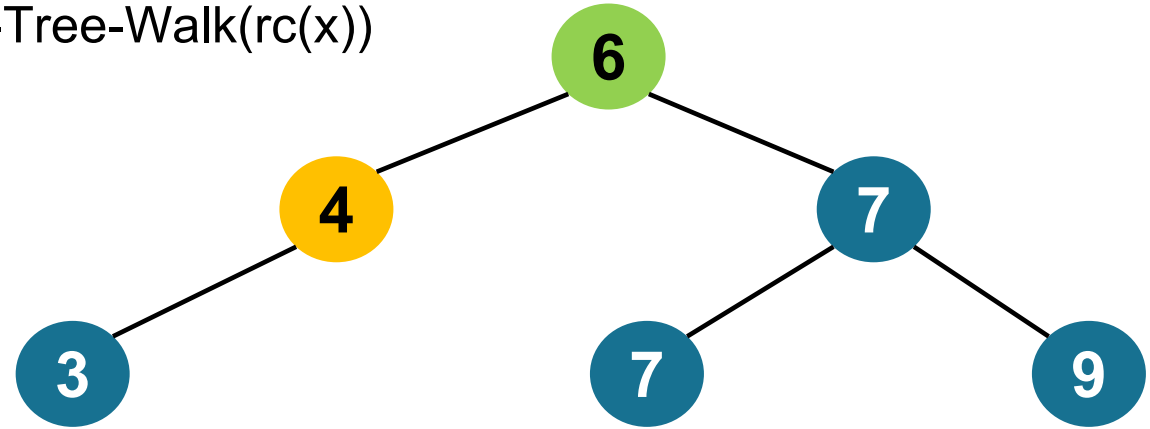
Open Distributed Sytems ODS

Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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))



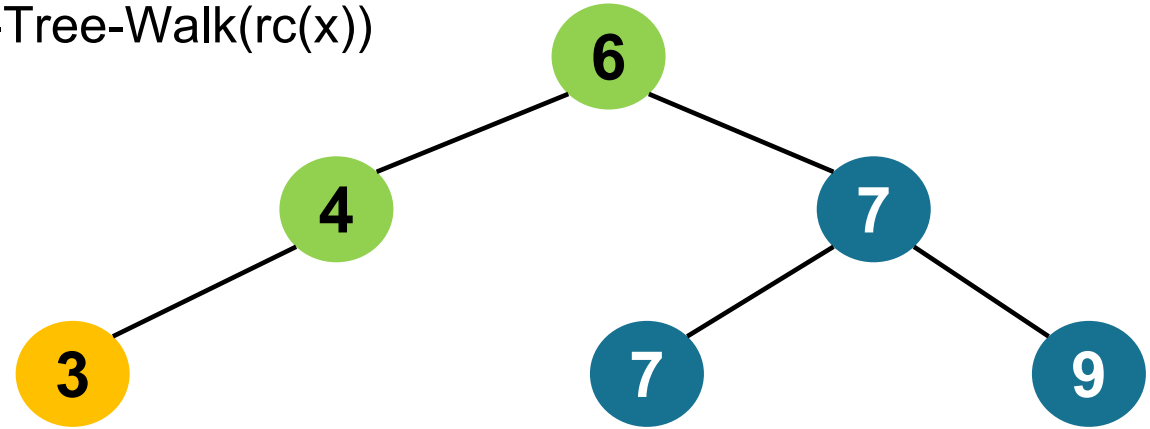
Open Distributed Sytems
ODS

Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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))



Open Distributed Sytems ODS

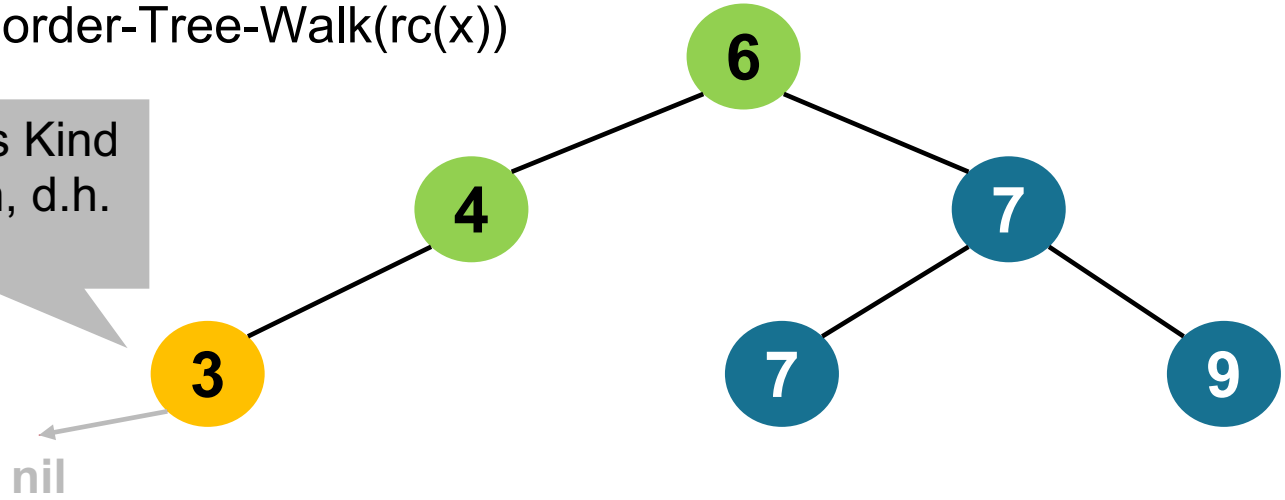
Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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))

Kein linkes Kind vorhanden, d.h. lc(x)=**nil**



Open Distributed Sytems
ODS

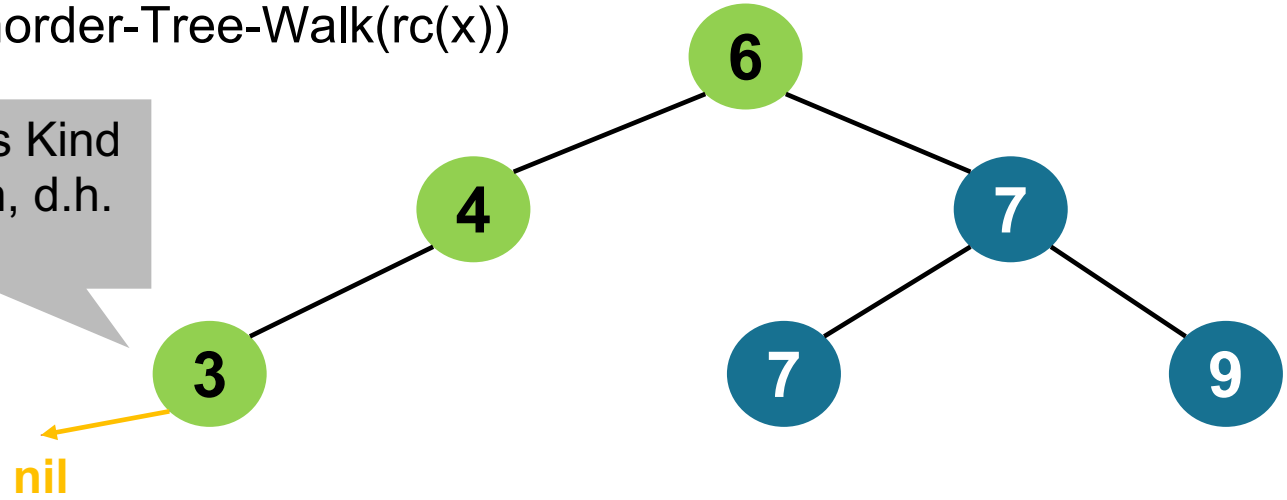
Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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))

Kein linkes Kind vorhanden, d.h. lc(x)=**nil**



Open Distributed Sytems ODS

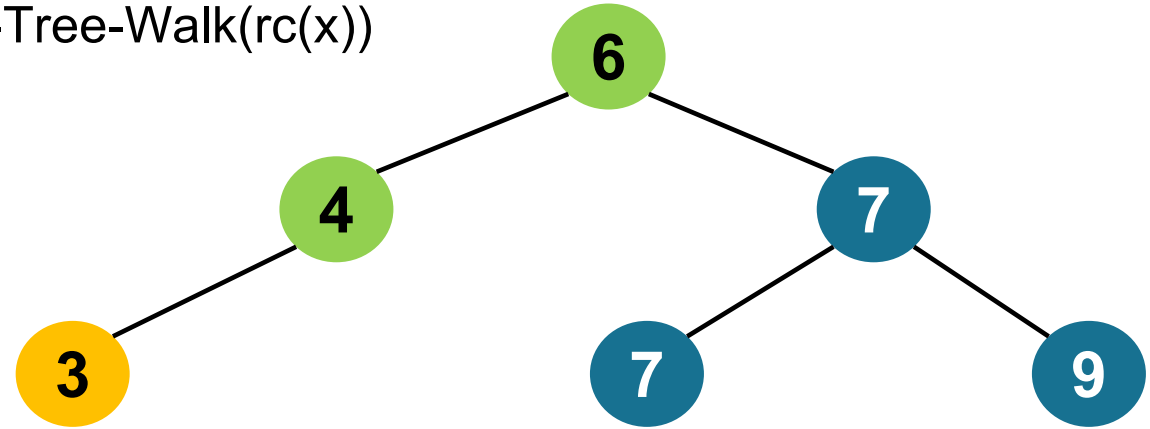
Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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



Open Distributed Sytems
ODS

Binäre Suchbäume – Durchlaufen

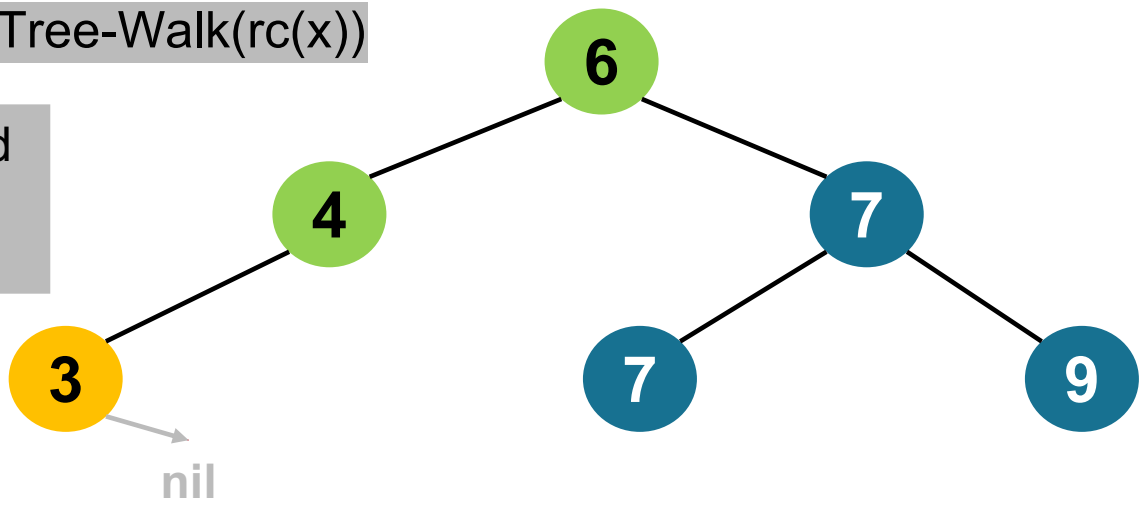
Technische Universität Berlin

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

Kein rechtes Kind vorhanden, d.h. rc(x)=**nil**



Open Distributed Sytems
ODS

Binäre Suchbäume – Durchlaufen

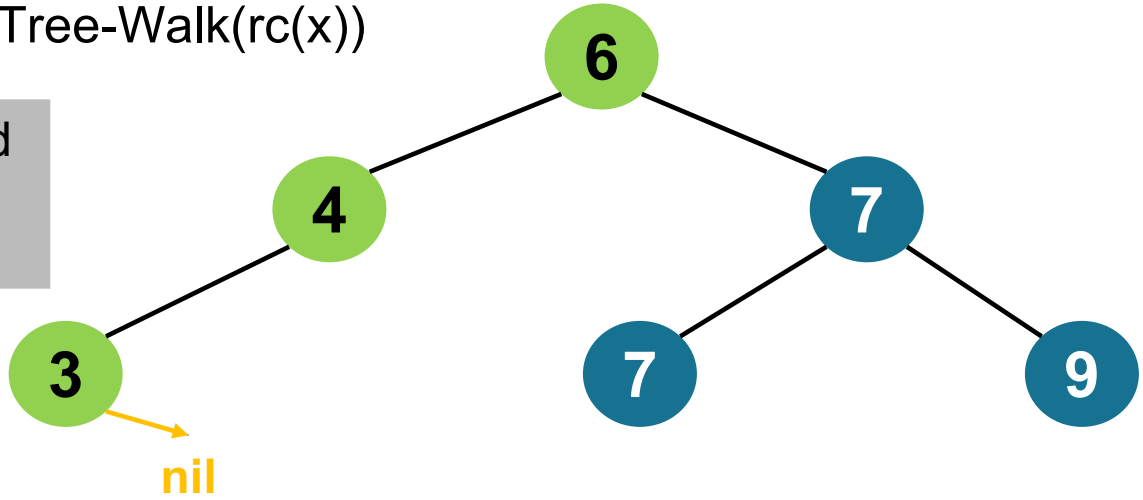
Technische Universität Berlin

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

Kein rechtes Kind vorhanden, d.h. rc(x)=**nil**



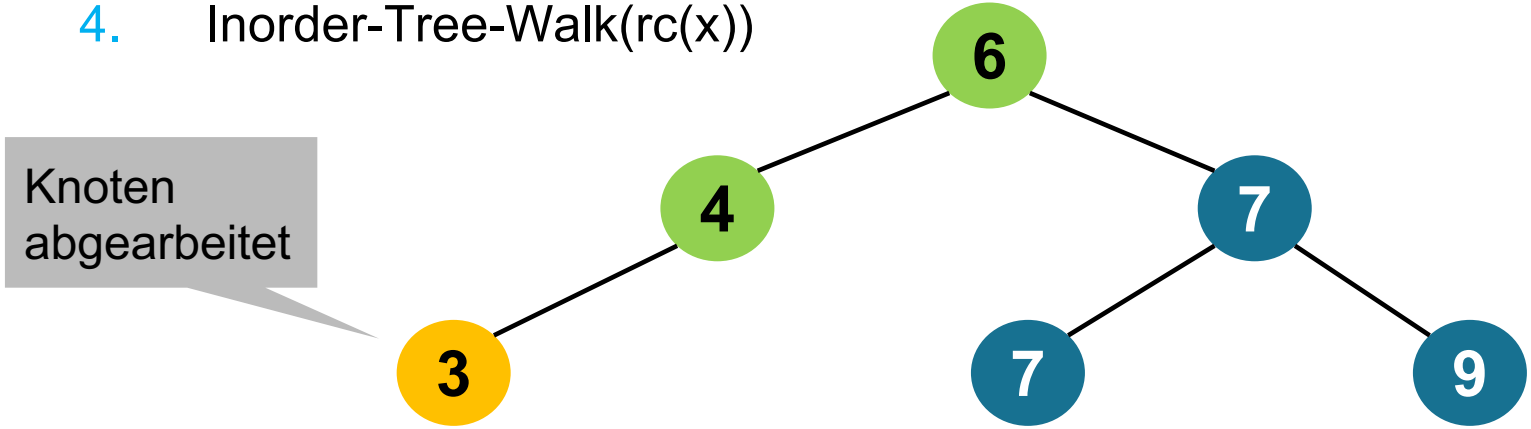
Open Distributed Sytems ODS

Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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



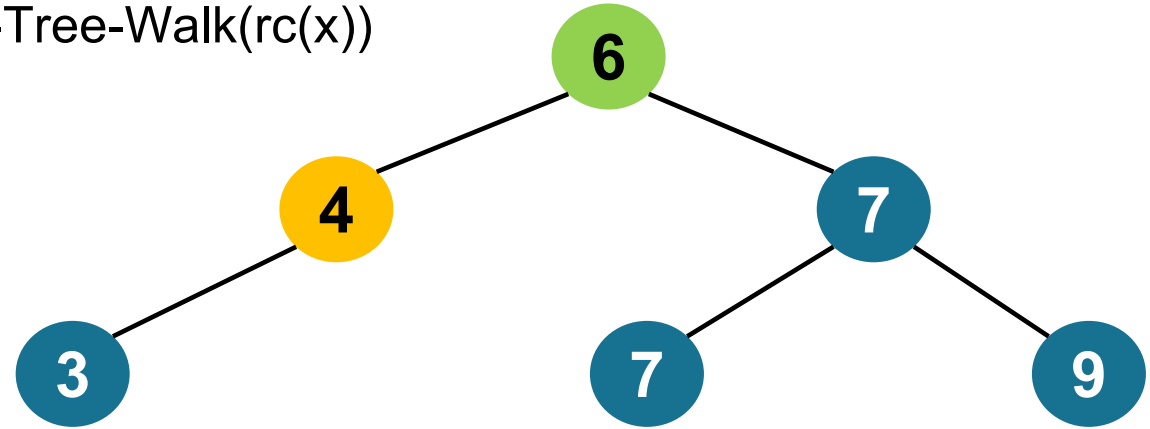
Open Distributed Sytems ODS

Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

```
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



Open Distributed Sytems ODS

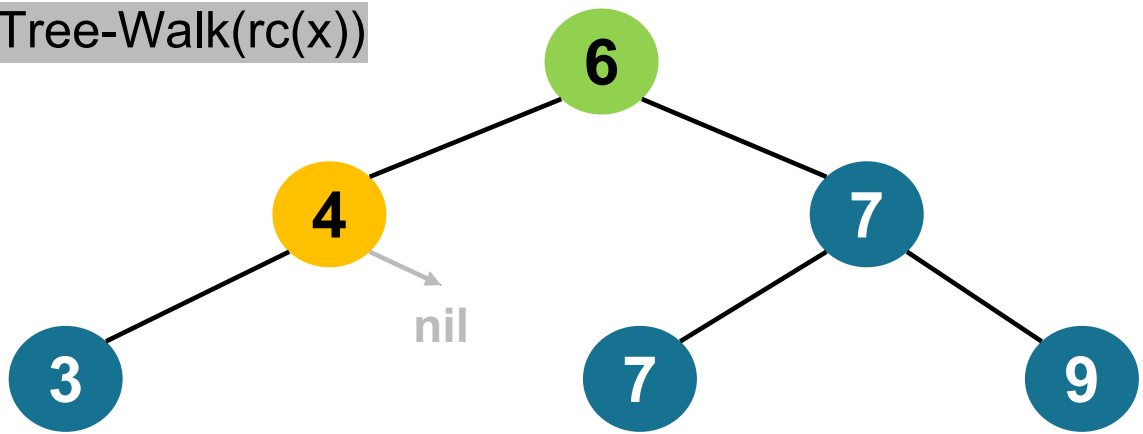
Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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



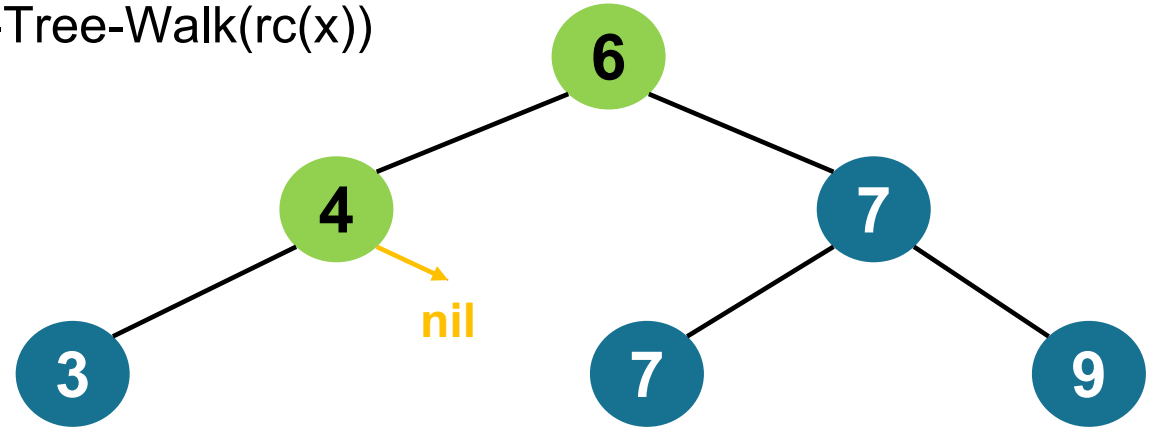
Open Distributed Sytems ODS

Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

```
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



Open Distributed Sytems
ODS

Binäre Suchbäume – Durchlaufen

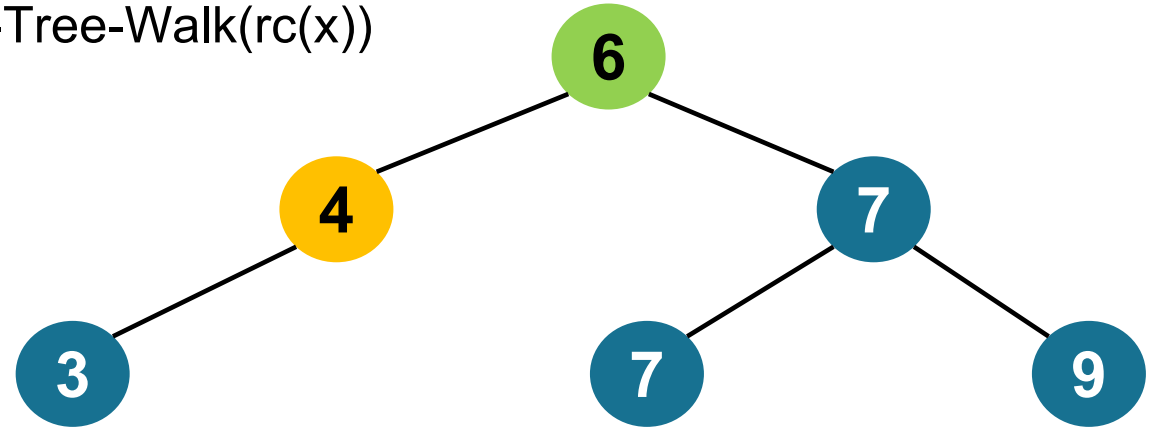
Technische Universität Berlin

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



Open Distributed Sytems ODS

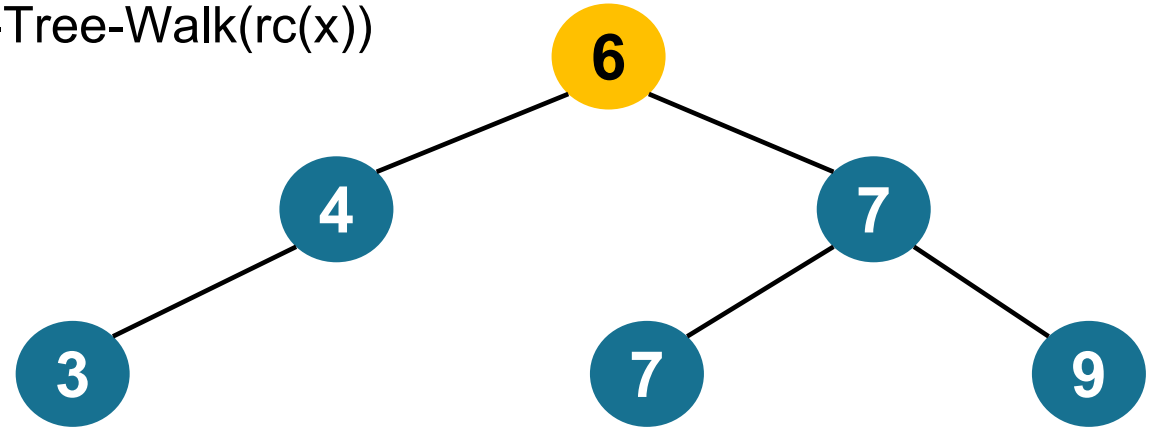
Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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



Open Distributed Sytems
ODS

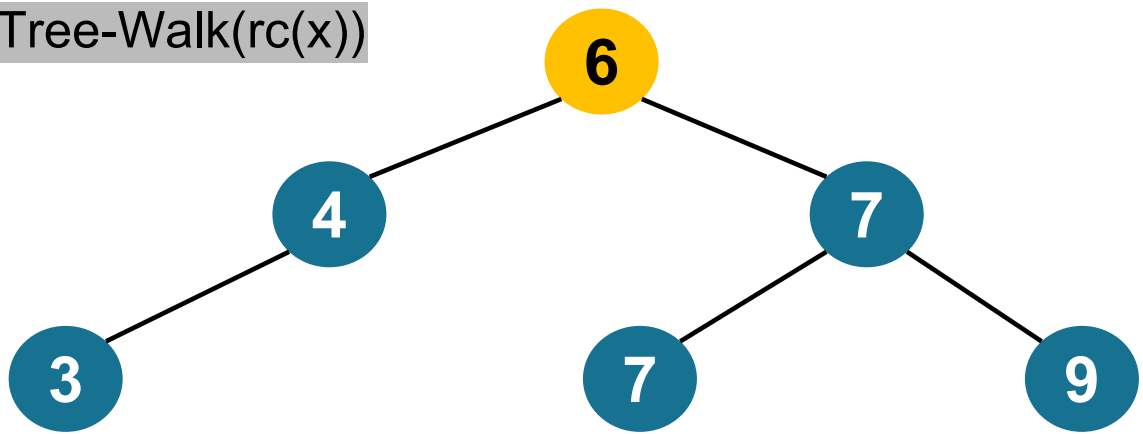
Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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



Open Distributed Sytems
ODS

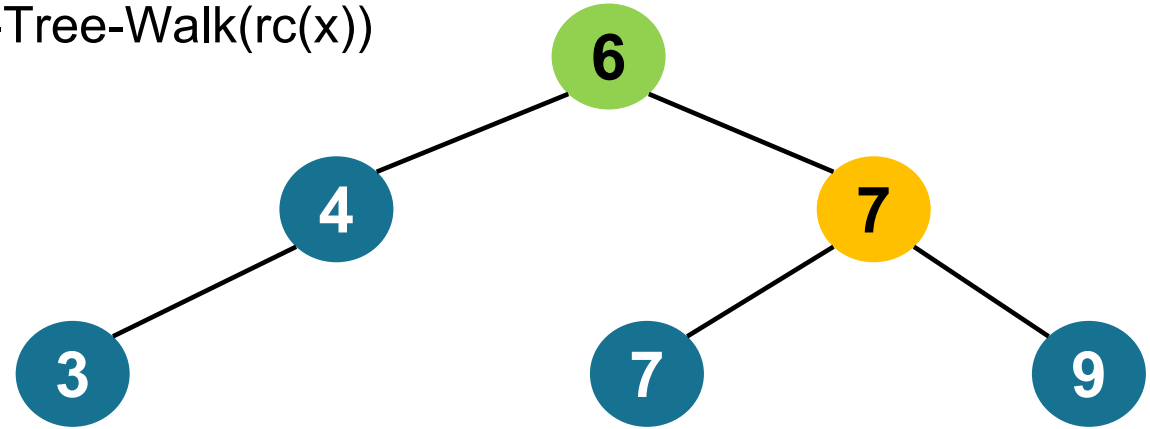
Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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



Open Distributed Sytems
ODS

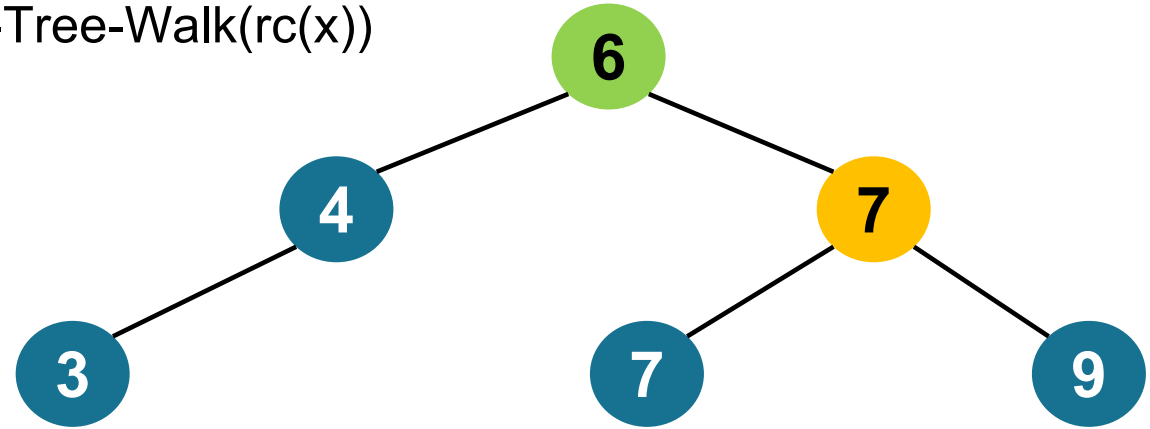
Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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



Open Distributed Sytems
ODS

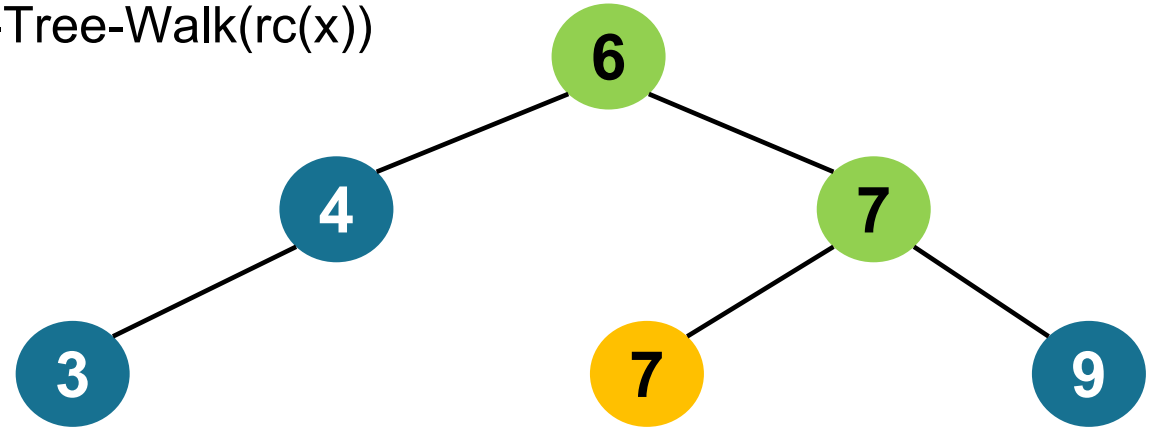
Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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



Open Distributed Sytems
ODS

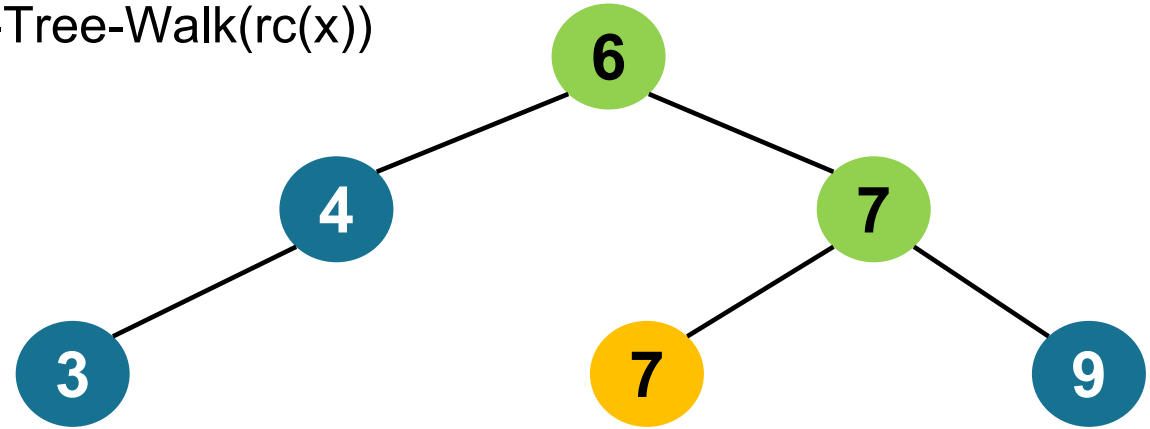
Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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



Open Distributed Sytems
ODS

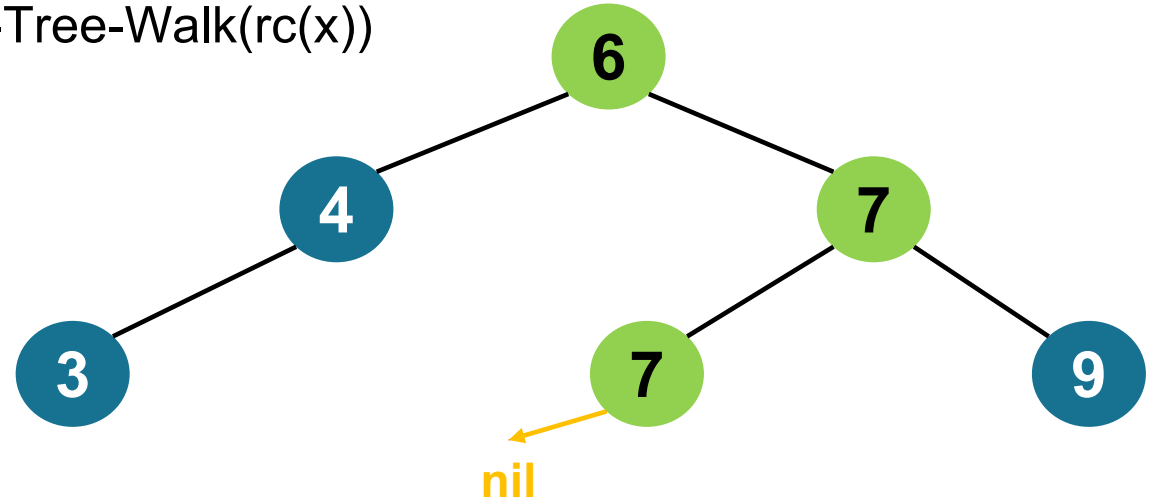
Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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



Open Distributed Sytems
ODS

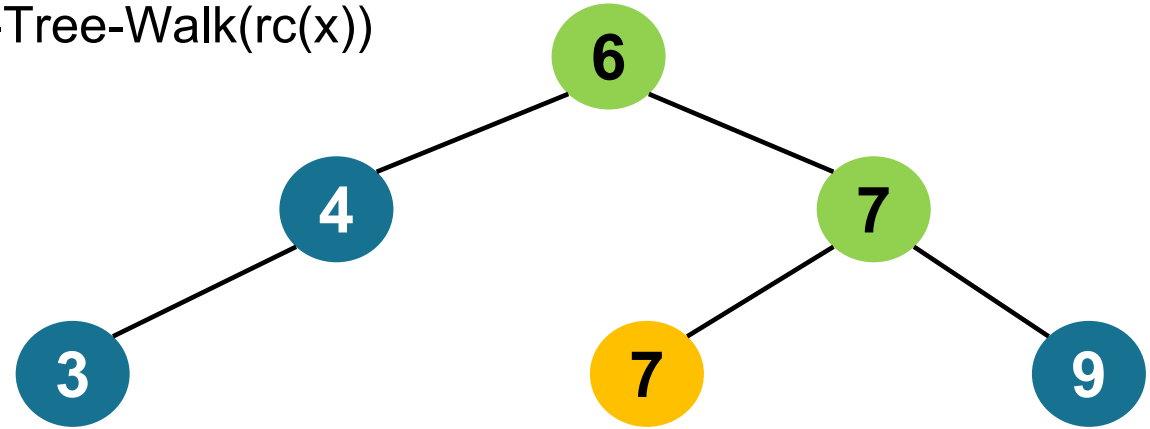
Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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



Open Distributed Sytems ODS

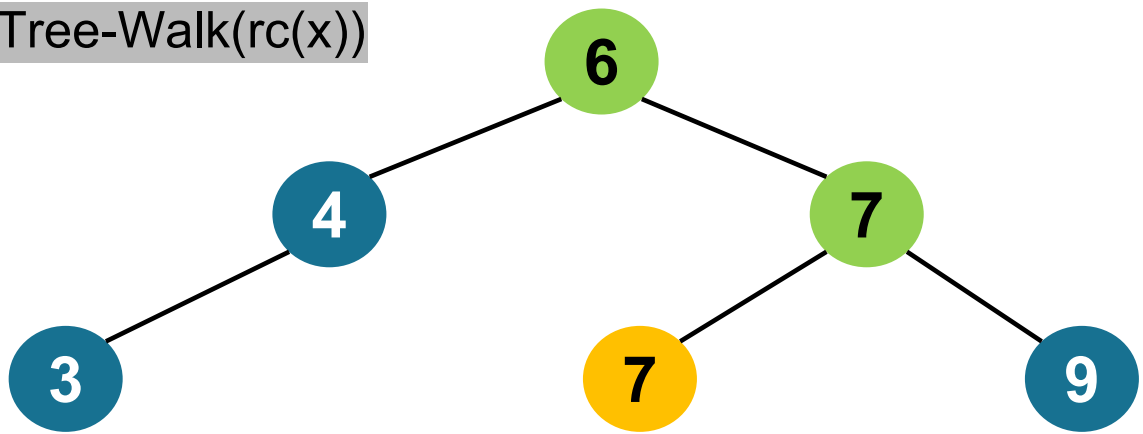
Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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



Open Distributed Sytems
ODS

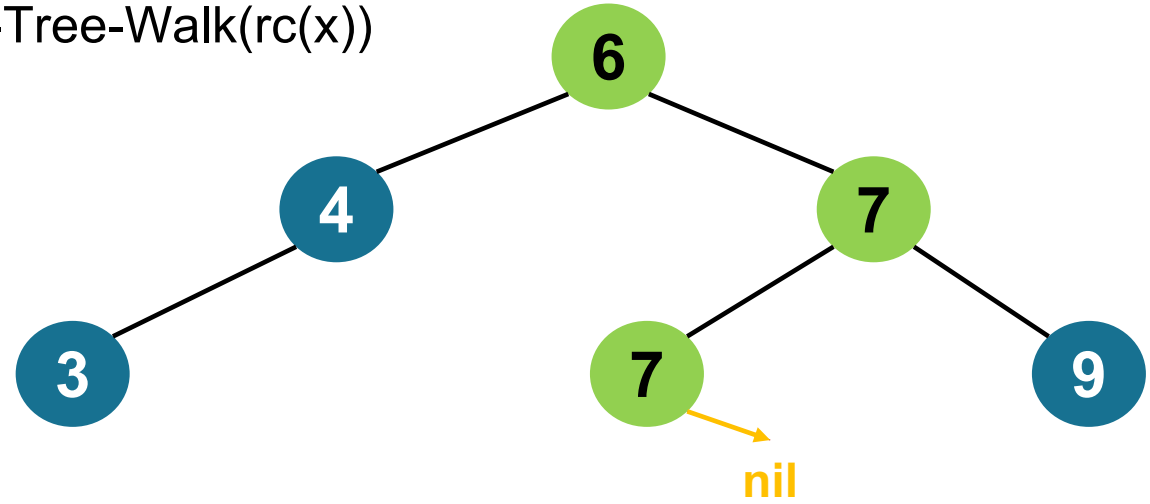
Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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



Open Distributed Sytems
ODS

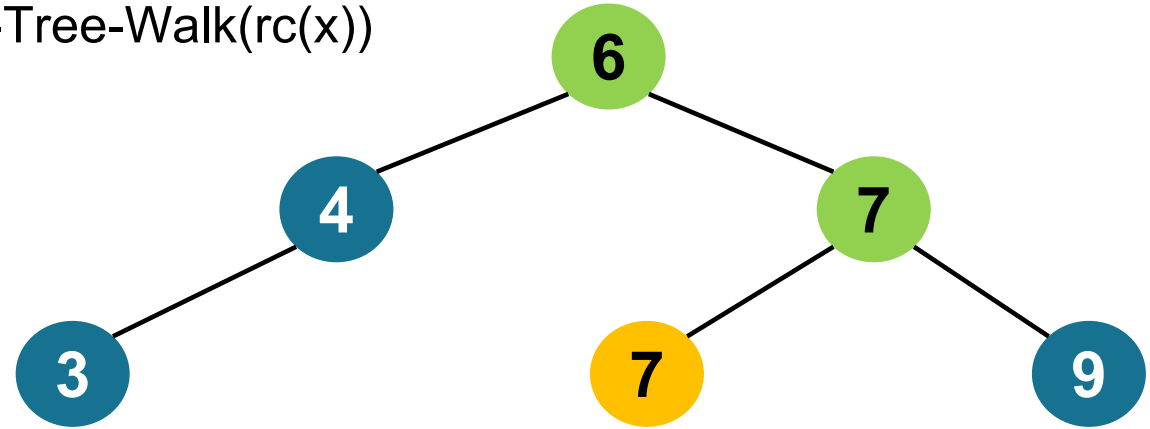
Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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



Open Distributed Sytems ODS

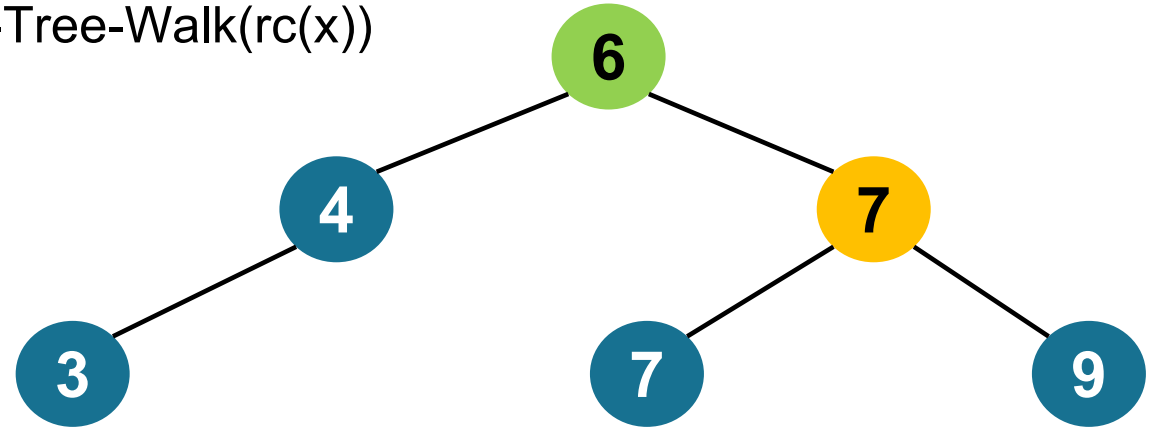
Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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



Open Distributed Sytems ODS

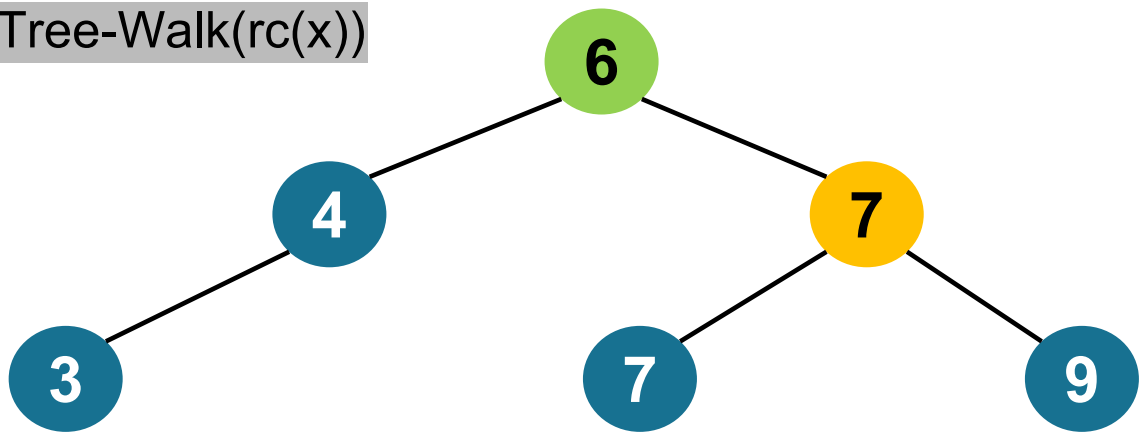
Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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



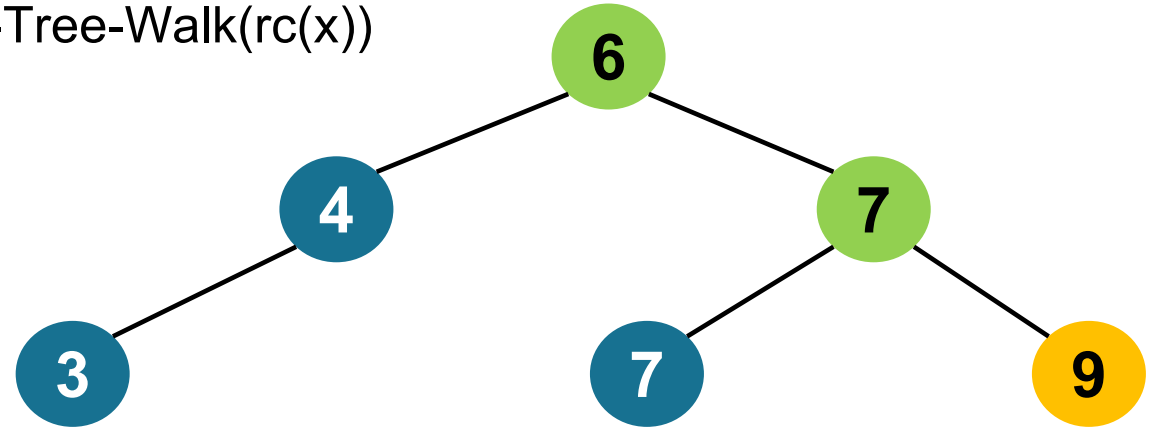
Open Distributed Sytems
ODS

Binäre Suchbäume – Durchlaufen

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



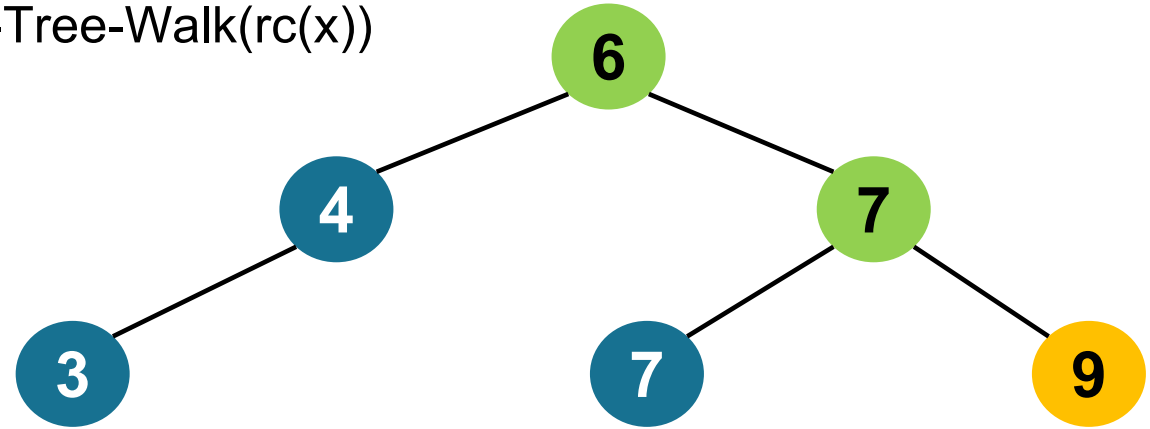
Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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



Open Distributed Sytems
ODS

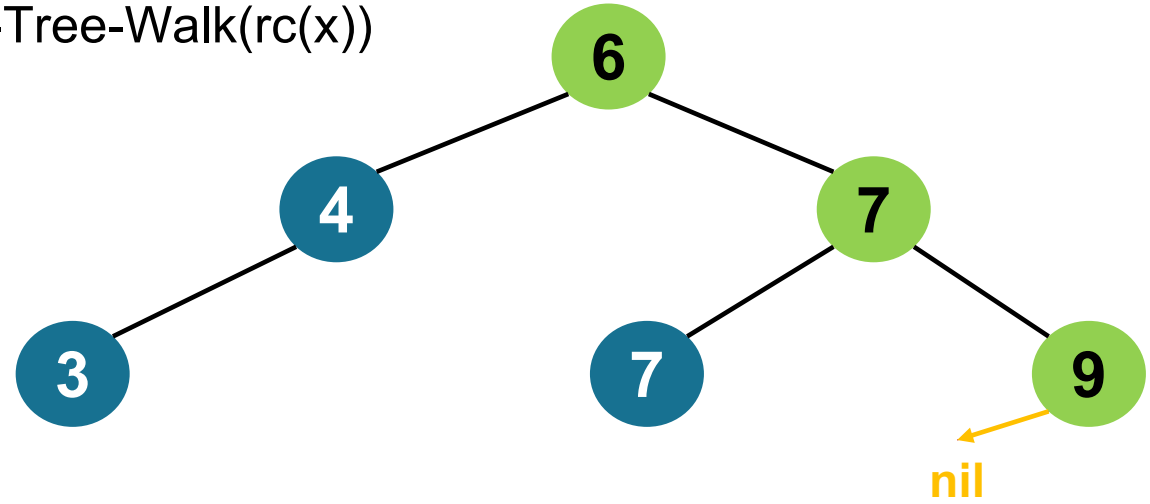
Binäre Suchbäume – Durchlaufen

Technische Universität Berlin

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



Open Distributed Sytems
ODS

Binäre Suchbäume – Durchlaufen

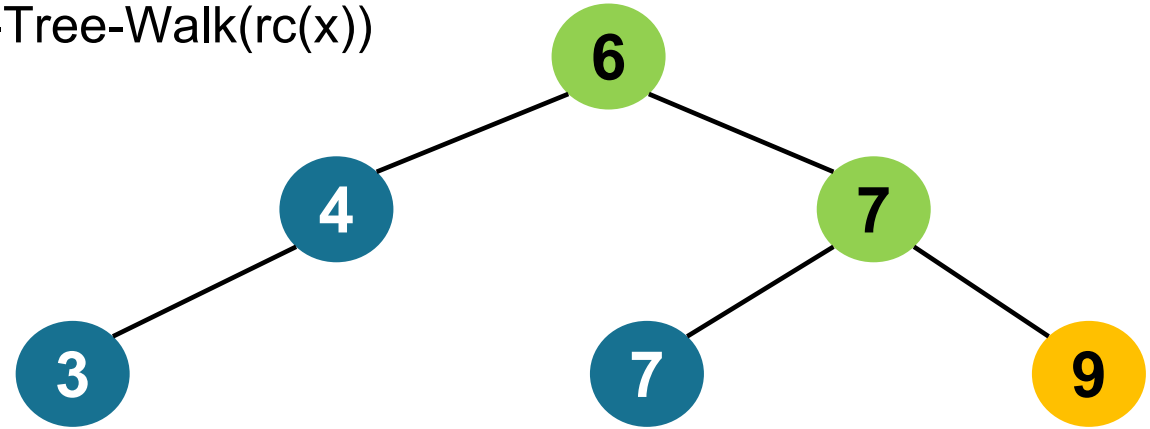
Technische Universität Berlin

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



Open Distributed Sytems ODS

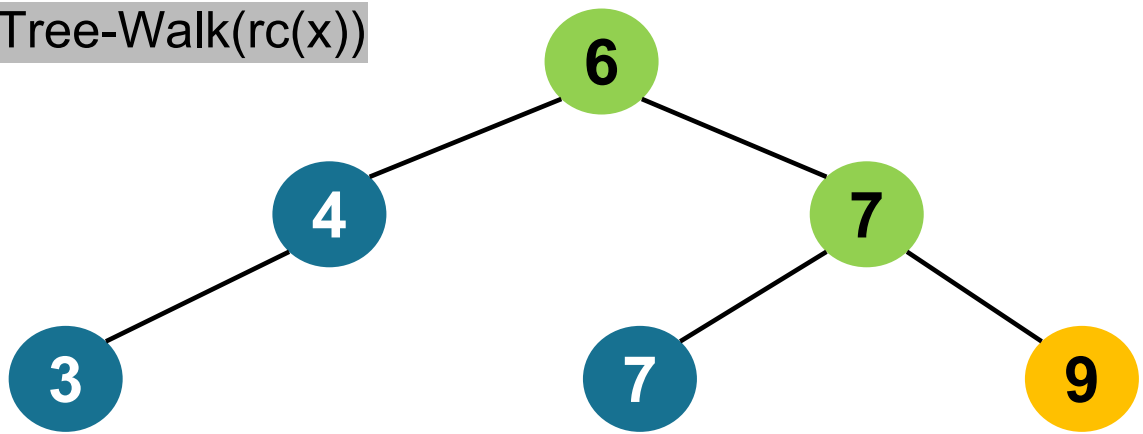
Binäre Suchbäume – Durchlaufen

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 – Durchlaufen

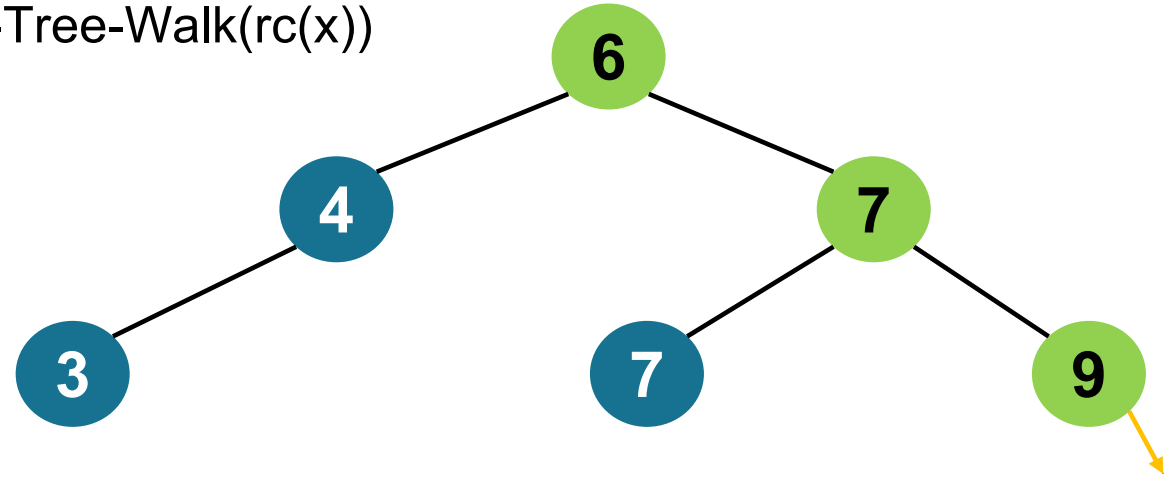
Technische Universität Berlin

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



Open Distributed Sytems ODS

Binäre Suchbäume – Durchlaufen

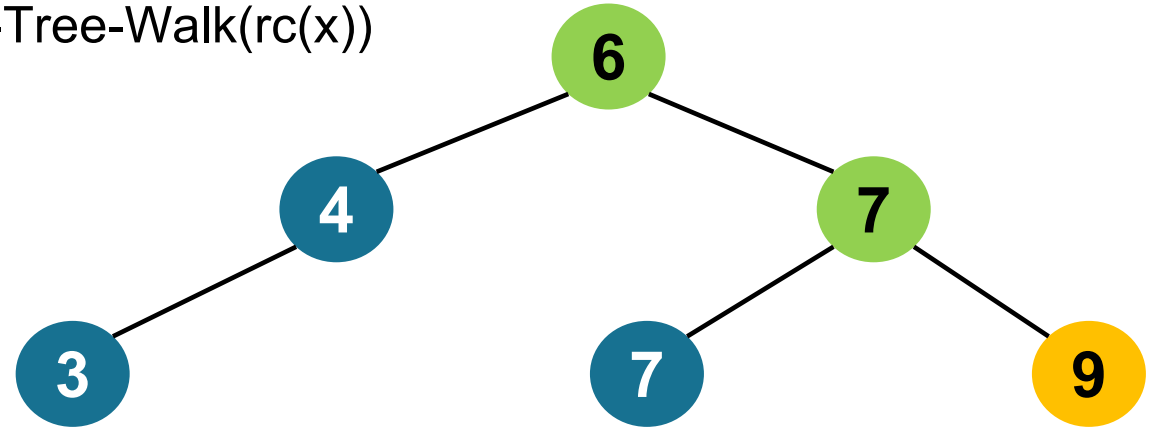
Technische Universität Berlin

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



Open Distributed Sytems ODS

Binäre Suchbäume – Durchlaufen

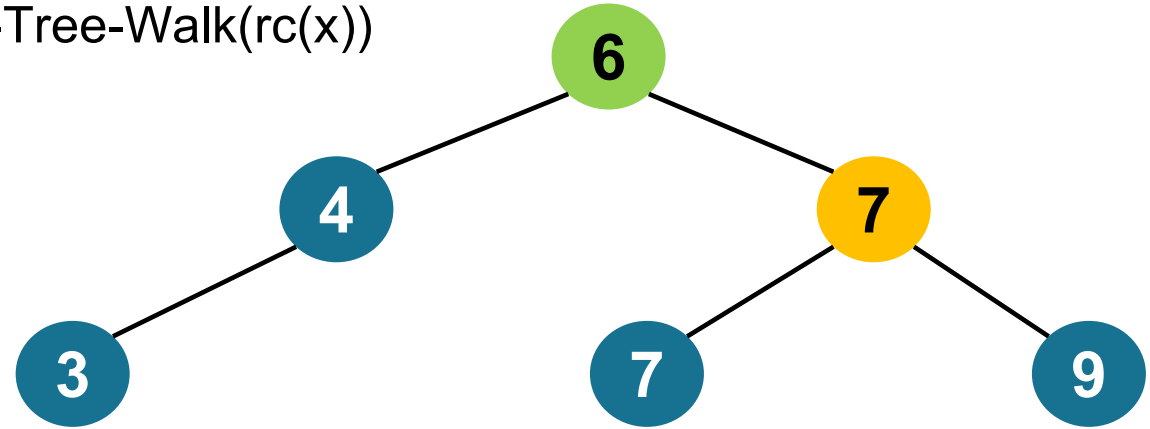
Technische Universität Berlin

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



Open Distributed Sytems ODS

Binäre Suchbäume – Durchlaufen

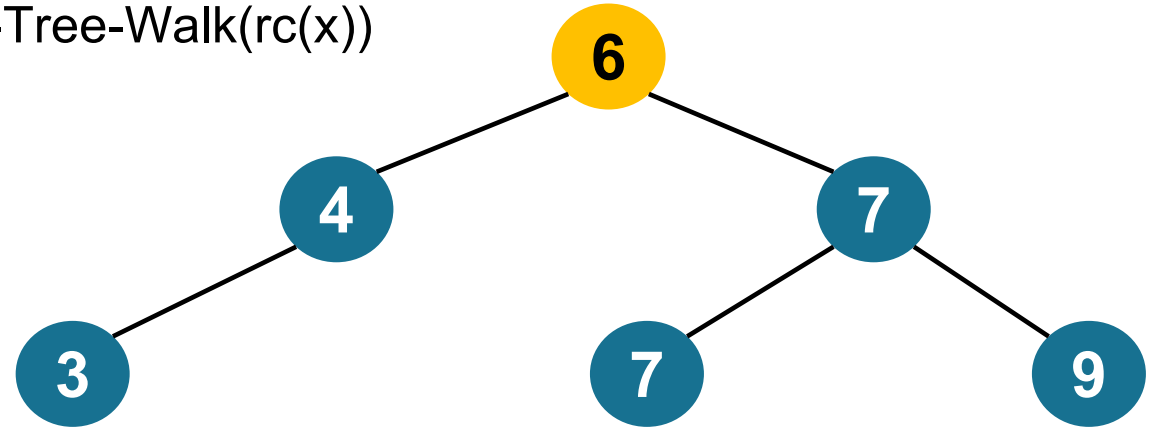
Technische Universität Berlin

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



Open Distributed Sytems ODS

Binäre Suchbäume – Durchlaufen

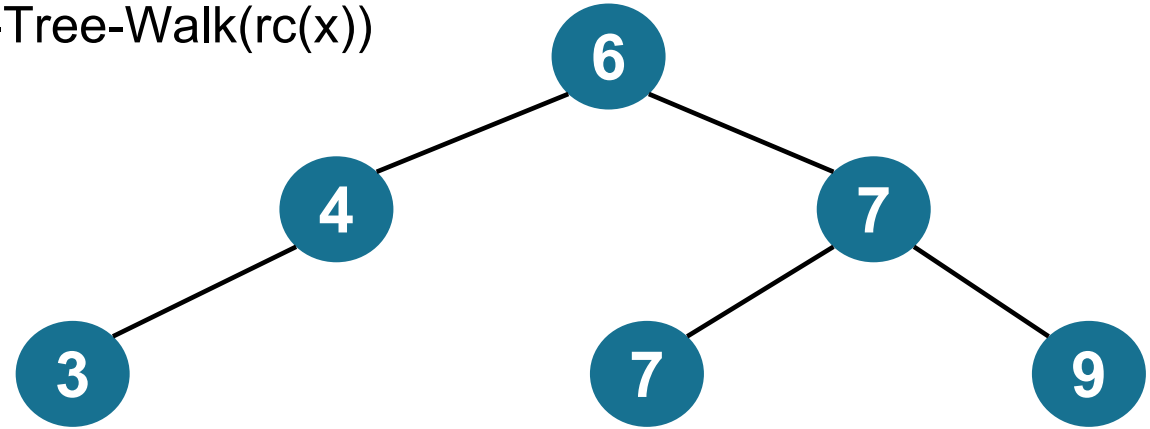
Technische Universität Berlin

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



Open Distributed Sytems ODS

Binäre Suchbäume Durchlaufen – Laufzeit

- Ausgabe aller Schlüssel
  - Gegeben binärer Suchbaum
  - Wie kann man alle Schlüssel aufsteigend sortiert in O(n) Zeit ausgeben?

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))

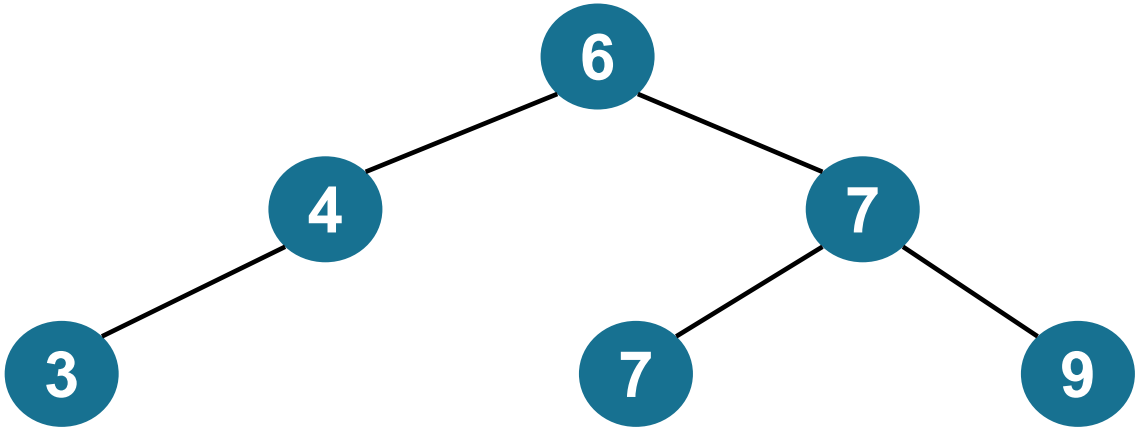
Anzahl der Elemente: n

- n + 2 * # Blätter
- < n
- n
- < n

=> Laufzeit Θ(n)

Binäre Suchbäume – Suchen

- Suchen in Binärbäumen
  - Gegeben ist Schlüssel k
  - Gesucht ist ein Knoten mit Schlüssel k

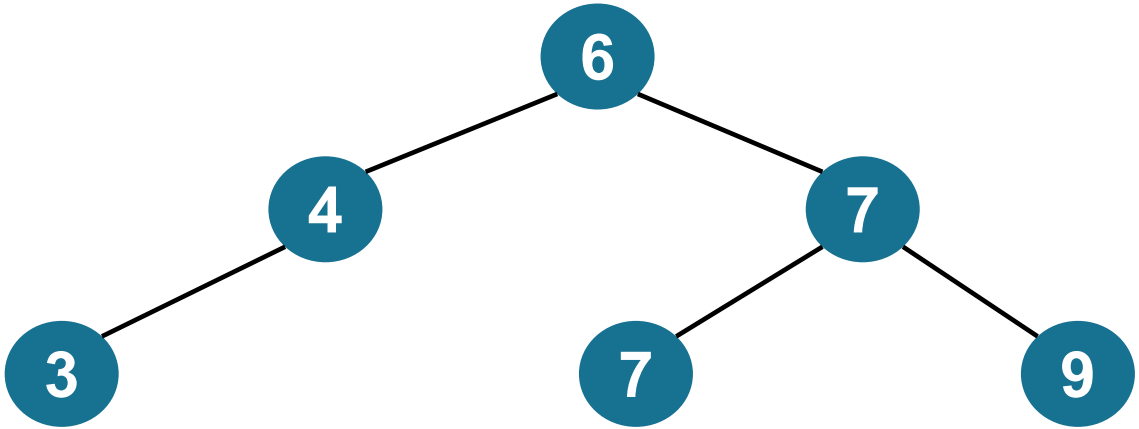


Binäre Suchbäume – Suchen

Technische Universität Berlin

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)

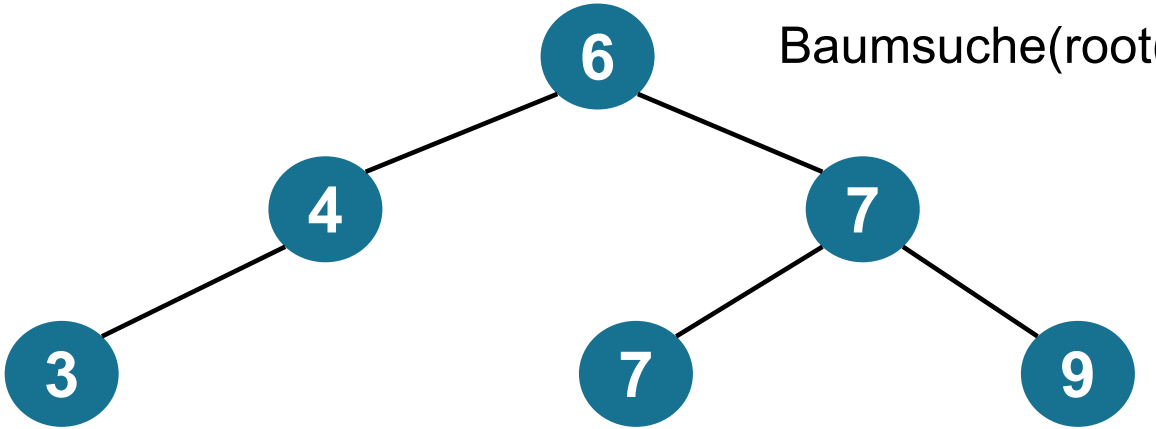


Open Distributed Sytems
ODS

Binäre Suchbäume – Suchen

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)

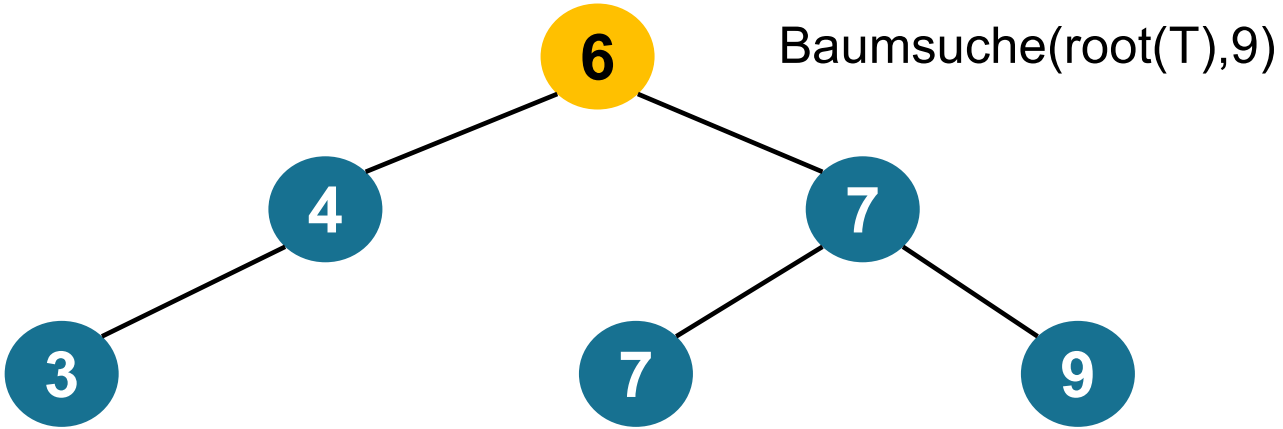


Baumsuche(root(T),9)

Binäre Suchbäume – Suchen

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)

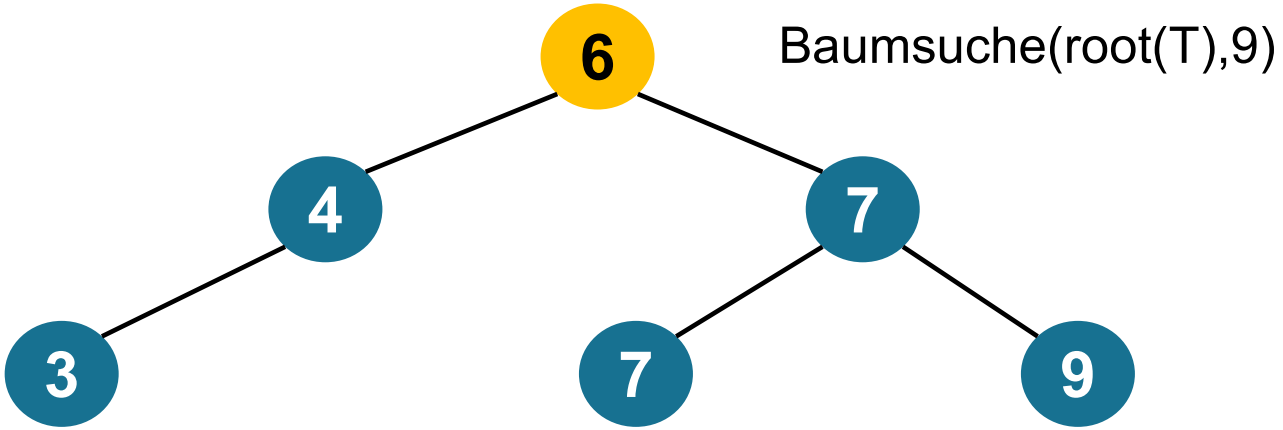


Baumsuche(root(T),9)

Binäre Suchbäume – Suchen

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)

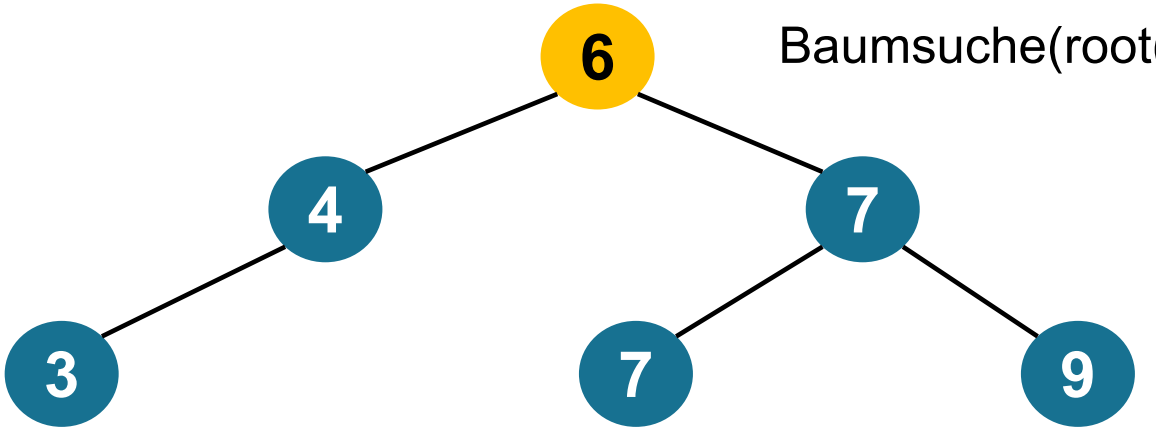


Baumsuche(root(T),9)

Binäre Suchbäume – Suchen

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)

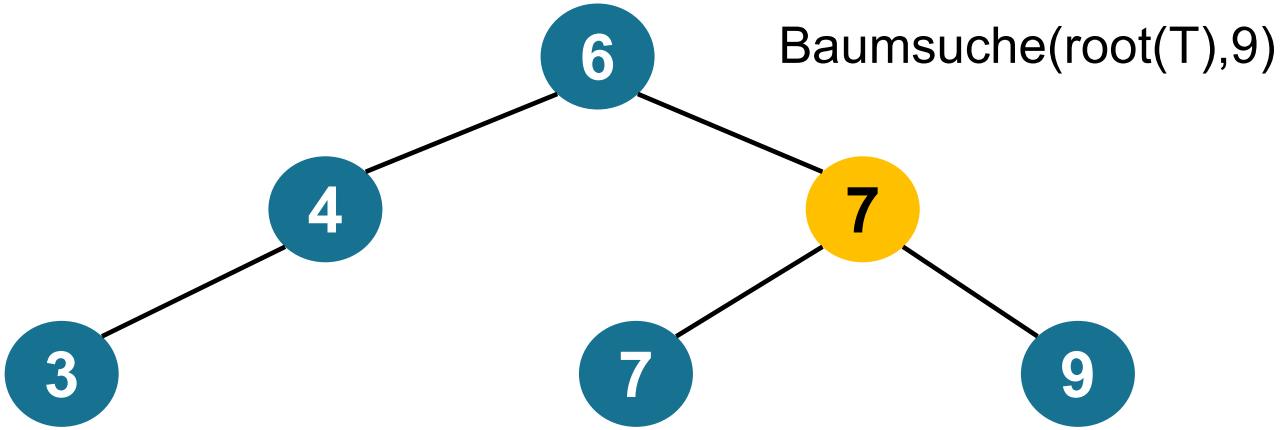


Baumsuche(root(T),9)

Binäre Suchbäume – Suchen

Baumsuche(x,k)

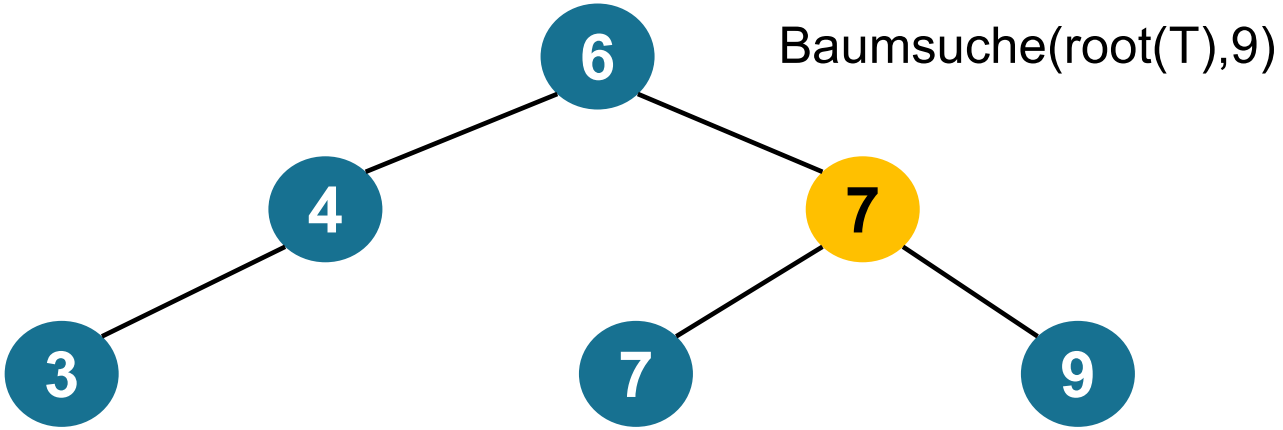
1. **if** x=**nil or** k=key(x) **then return** x
2. **if** k<key(x) **then return** Baumsuche(lc(x),k)
3. **else return** Baumsuche(rc(x),k)



Binäre Suchbäume – Suchen

Baumsuche(x,k)

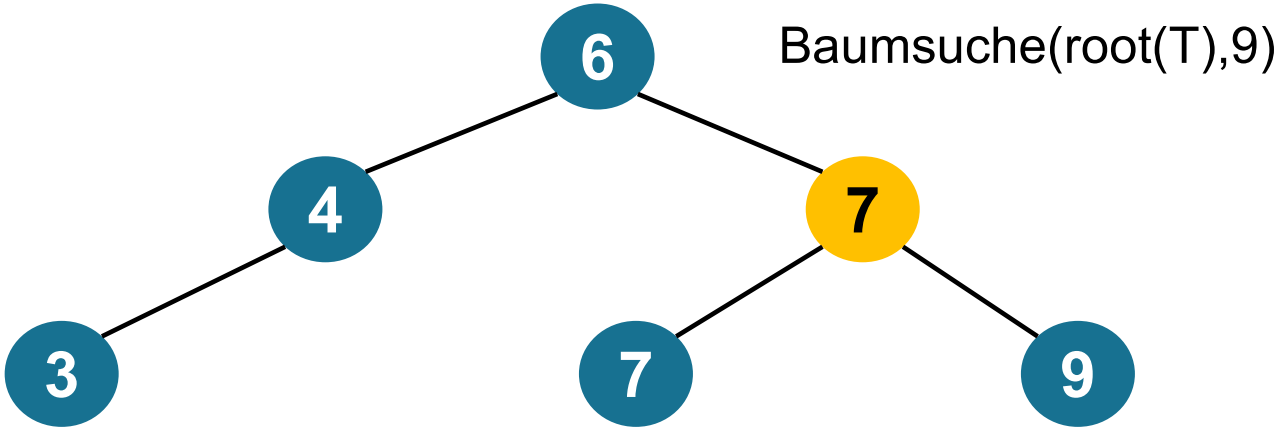
1. **if** x=**nil or** k=key(x) **then return** x
2. **if** k<key(x) **then return** Baumsuche(lc(x),k)
3. **else return** Baumsuche(rc(x),k)



Binäre Suchbäume – Suchen

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 – Suchen

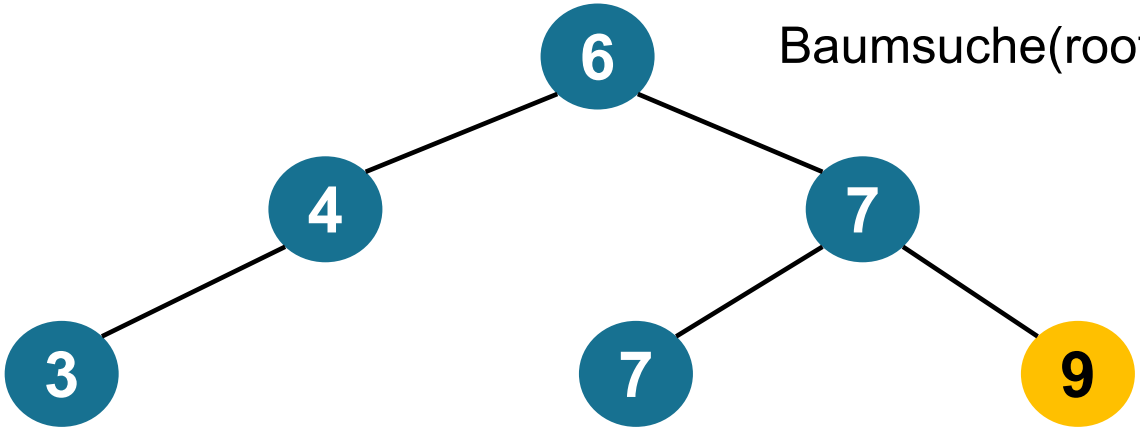
Technische Universität Berlin

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)

Baumsuche(root(T),9)

Ausgabe: 9



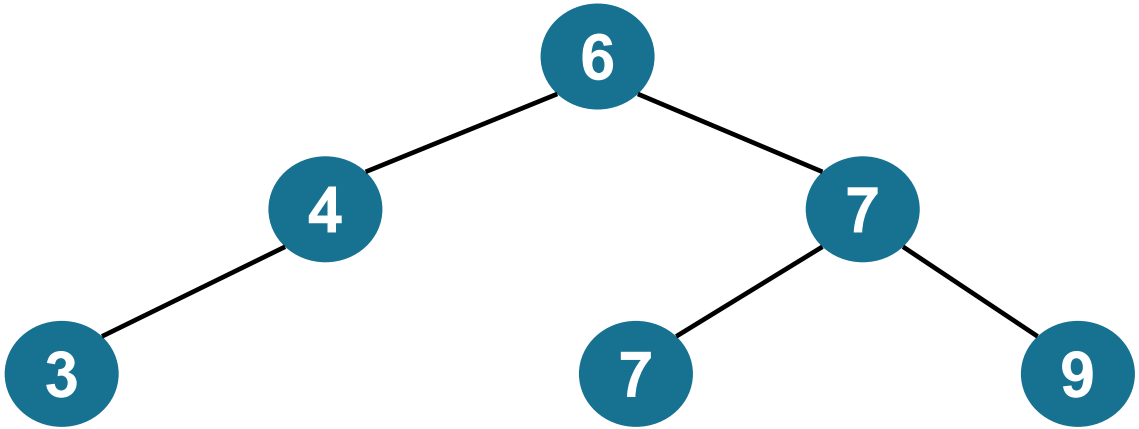
Open Distributed Sytems ODS

Binäre Suchbäume
Suchen – Laufzeit?

Technische Universität Berlin

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) // Entweder links
3. **else return** Baumsuche(rc(x),k) // oder rechts



Open Distributed Sytems
ODS

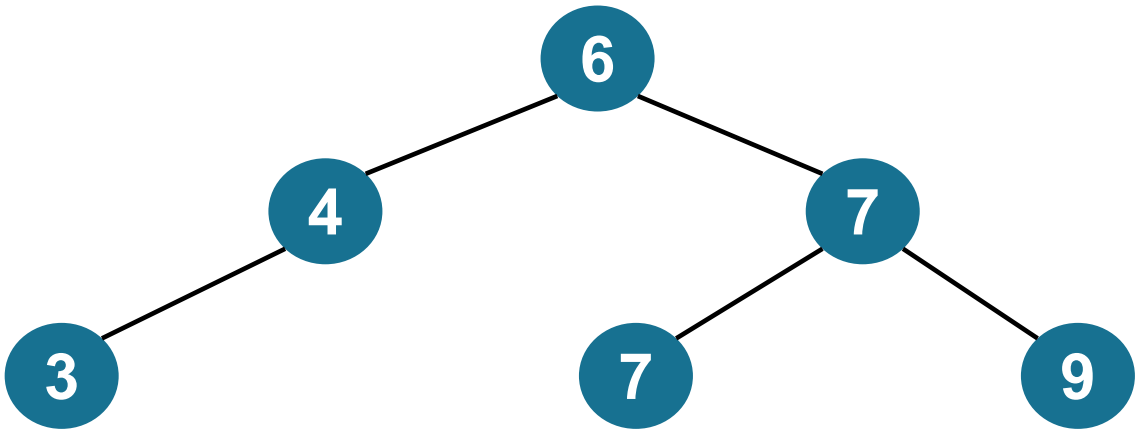
Binäre Suchbäume
Suchen – Laufzeit?

Technische Universität Berlin

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) // Entweder links
3. **else return** Baumsuche(rc(x),k) // oder rechts

Höhe h



Open Distributed Sytems
ODS

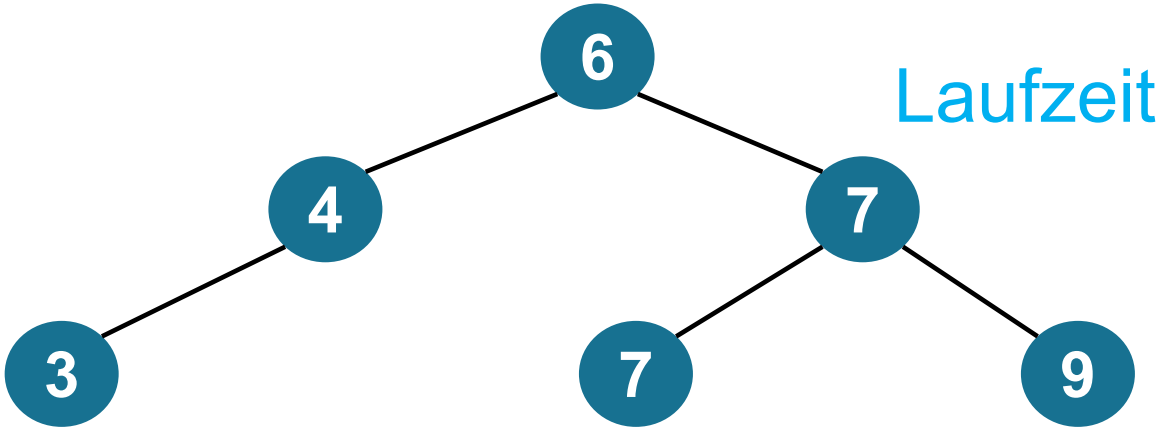
Binäre Suchbäume
Suchen – Laufzeit?

Technische Universität Berlin

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) // Entweder links
3. **else return** Baumsuche(rc(x),k) // oder rechts

Höhe h



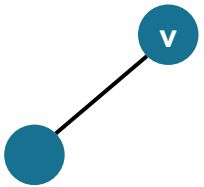
Laufzeit: O(h)

Open Distributed Sytems ODS

Höhe eines Baumes

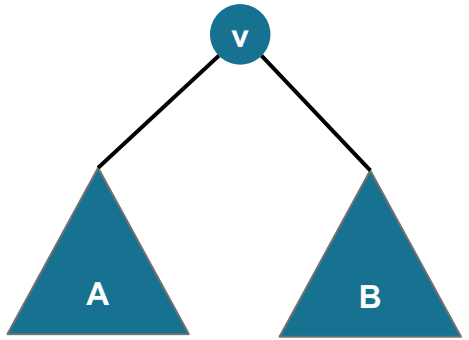
- Definition
  - Die Höhe eines Binärbaumes mit Wurzel v ist die Länge (Anzahl der Kanten) des längsten einfachen Weges (keine mehrfach vorkommenden Knoten) von der Wurzel zu einem Blatt.
- Beispiele
  - Baum der Höhe 0

  - Baum der Höhe 1



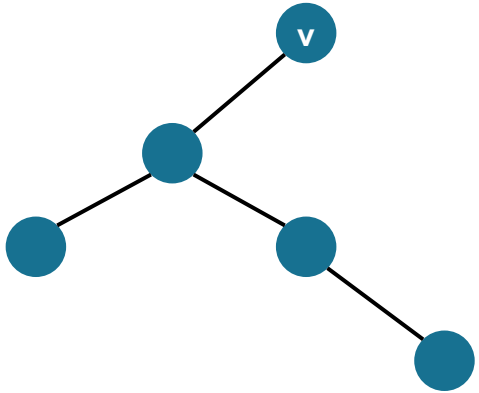
Höhe eines Baumes

- Definition
  - Die Höhe eines Binärbaumes mit Wurzel v ist die Länge (Anzahl der Kanten) des längsten einfachen Weges (keine mehrfach vorkommenden Knoten) von der Wurzel zu einem Blatt.
- Beispiel
  - Ein Baum mit einem Knoten hat Höhe 0
  - Damit gilt:
    Höhe eines Baumes mit Wurzel v und Teilbäumen A und B ist
    1 + max(Höhe(A), Höhe(B))



Höhe eines Baumes

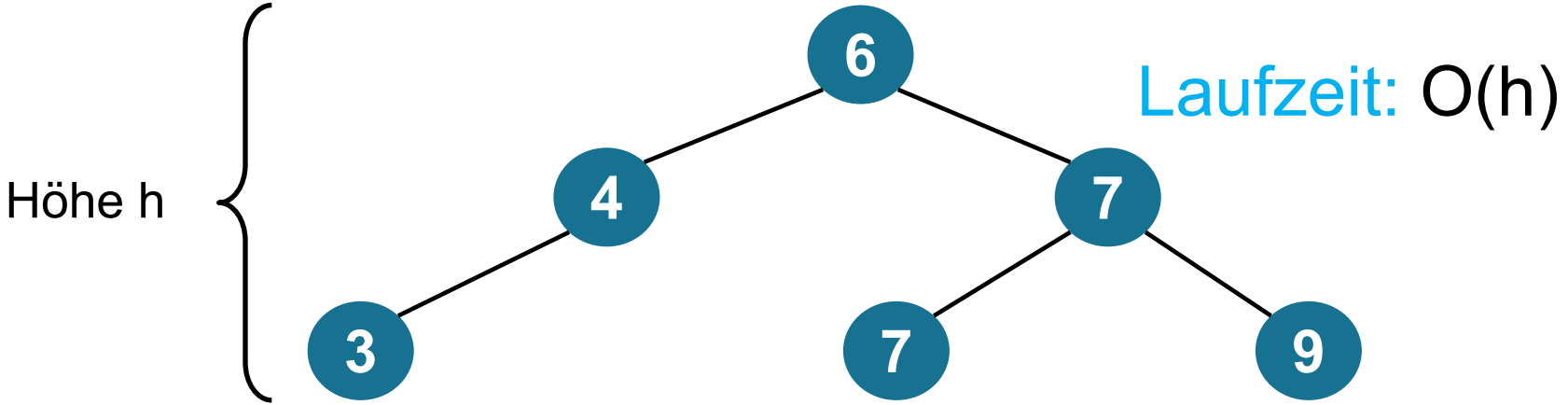
- Definition
  - Die Höhe eines Binärbaumes mit Wurzel v ist die Länge (Anzahl der Kanten) des längsten einfachen Weges (keine mehrfach vorkommenden Knoten) von der Wurzel zu einem Blatt.
- Beispiel
  - Baum der Höhe 3



Binäre Suchbäume – Suchen

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)

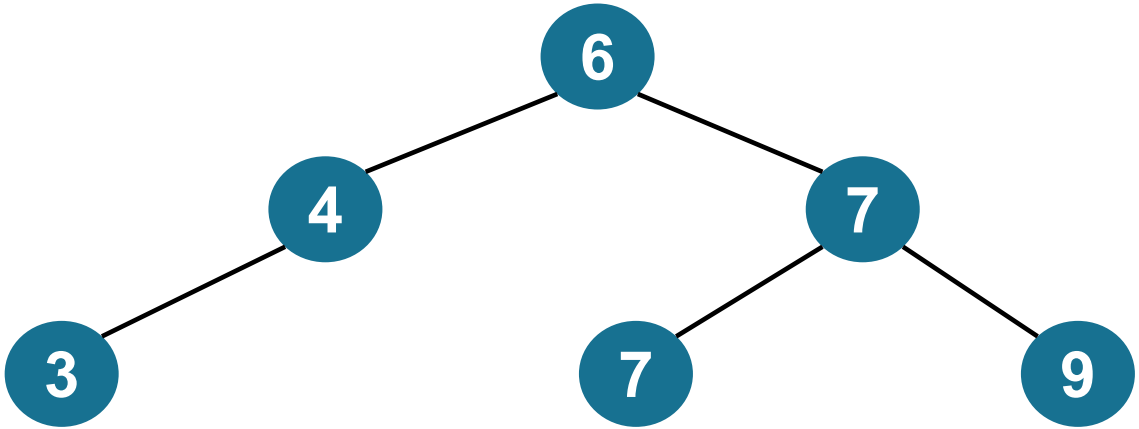


Laufzeit: O(h)

Binäre Suchbäume – Suchen

IterativeBaumsuche(x,k)

1. **while** x ≠ **nil and** k ≠ key(x) *do*
2. **if** k < key(x) **then** x ← lc(x)
3. **else** x ← rc(x)
4. **return** x

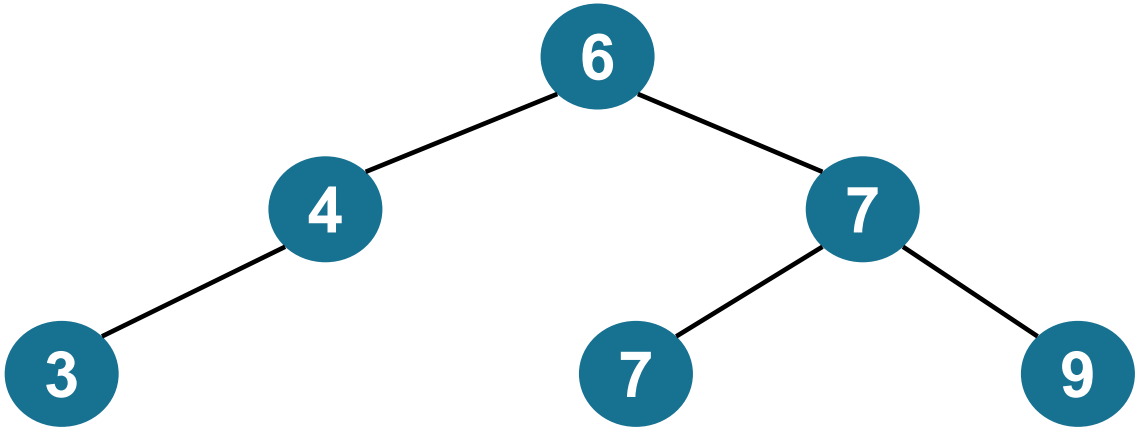


Binäre Suchbäume – Suchen

IterativeBaumsuche(x,k)

1. **while** x ≠ **nil and** k ≠ key(x) *do*
2. **if** k < key(x) **then** x ← lc(x)
3. **else** x ← rc(x)
4. **return** x

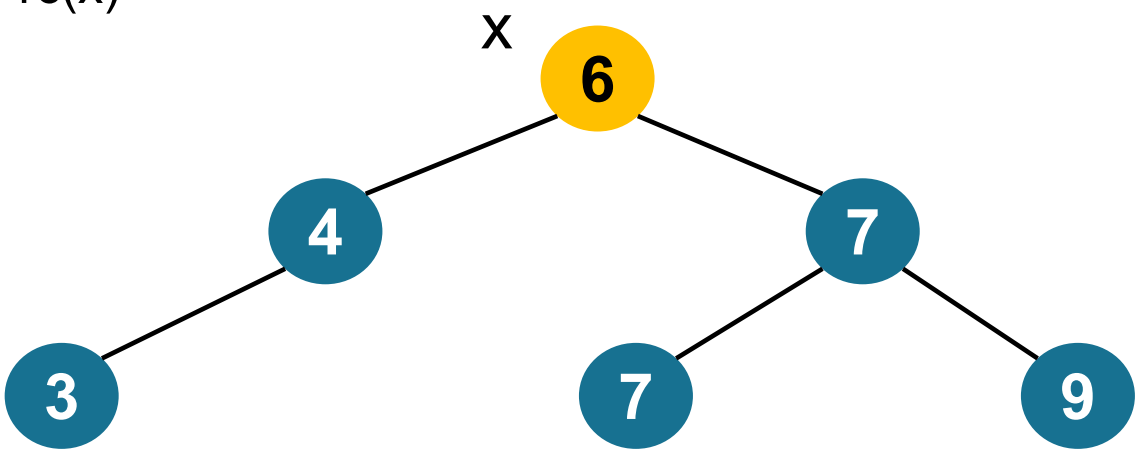
IterativeBaumsuche(root(T),5)



Binäre Suchbäume – Suchen

```
IterativeBaumsuche(x,k)
1. while x ≠ nil and k ≠ key(x) do
2.     if k < key(x) then x ← lc(x)
3.     else x ← rc(x)
4. return x
```

IterativeBaumsuche(root(T),5)

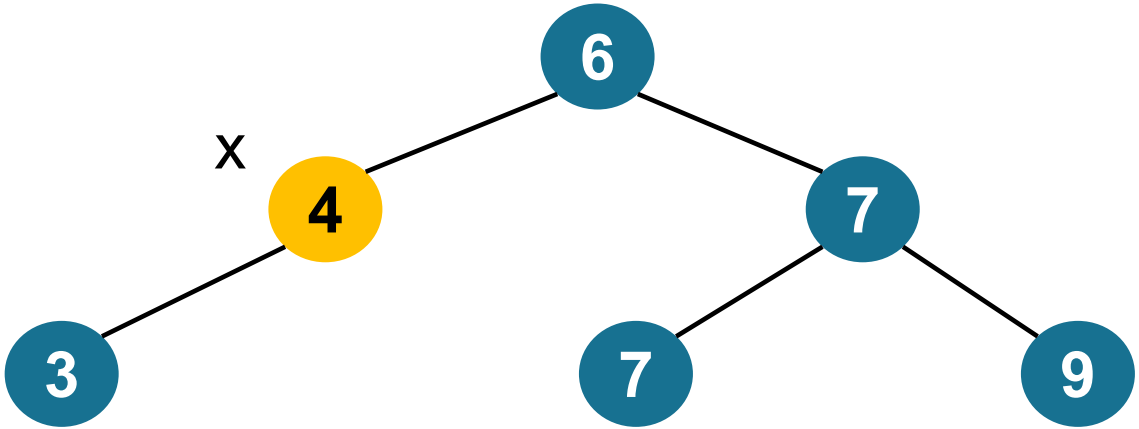


Binäre Suchbäume – Suchen

IterativeBaumsuche(x,k)

1. **while** x ≠ **nil and** k ≠ key(x) *do*
2. **if** k < key(x) **then** x ← lc(x)
3. **else** x ← rc(x)
4. **return** x

IterativeBaumsuche(root(T),5)

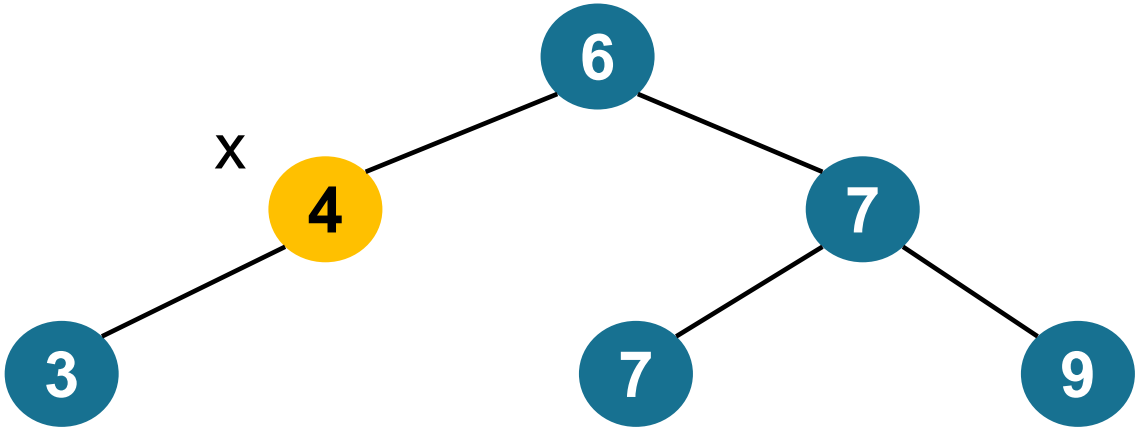


Binäre Suchbäume – Suchen

IterativeBaumsuche(x,k)

1. **while** x ≠ **nil and** k ≠ key(x) *do*
2. **if** k < key(x) **then** x ← lc(x)
3. **else** x ← rc(x)
4. **return** x

IterativeBaumsuche(root(T),5)

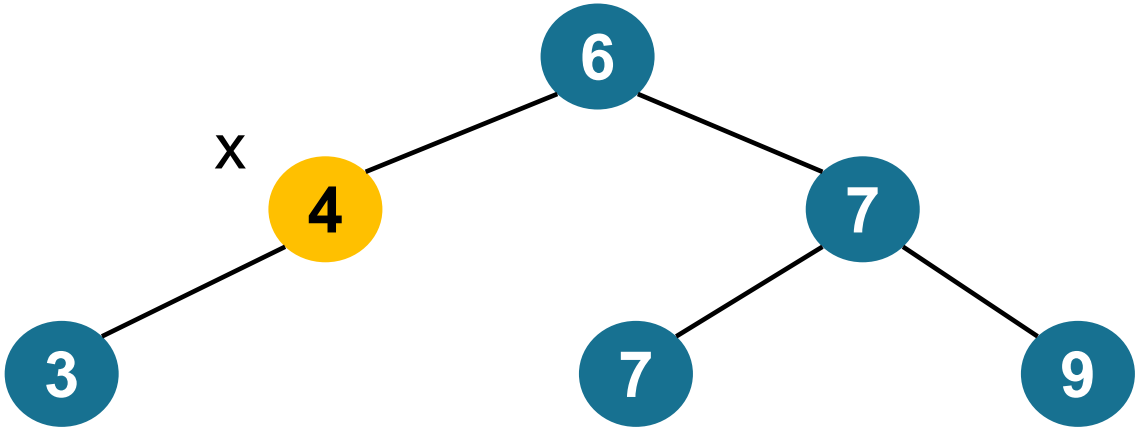


Binäre Suchbäume – Suchen

IterativeBaumsuche(x,k)

1. **while** x ≠ **nil and** k ≠ key(x) *do*
2. **if** k < key(x) **then** x ← lc(x)
3. **else** x ← rc(x)
4. **return** x

IterativeBaumsuche(root(T),5)

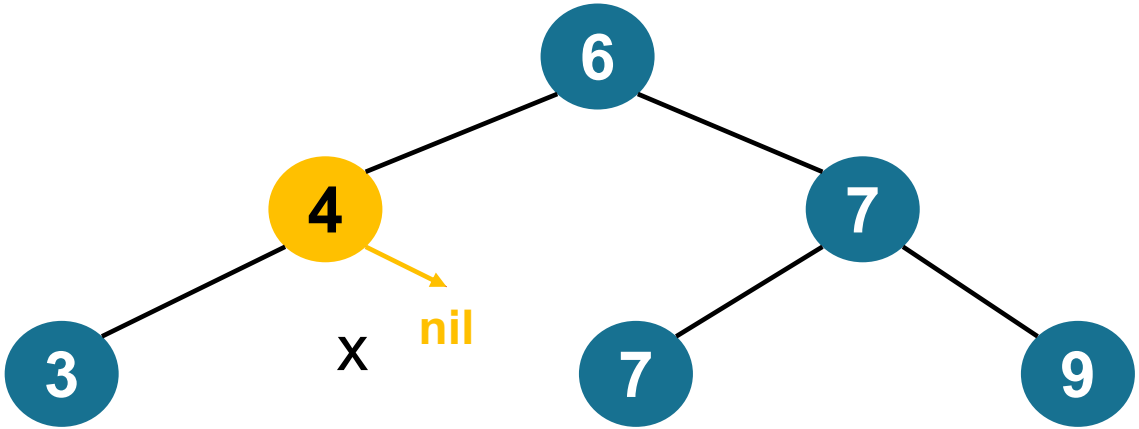


Binäre Suchbäume – Suchen

IterativeBaumsuche(x,k)

1. **while** x ≠ **nil and** k ≠ key(x) *do*
2. **if** k < key(x) **then** x ← lc(x)
3. **else** x ← rc(x)
4. **return** x

IterativeBaumsuche(root(T),5)

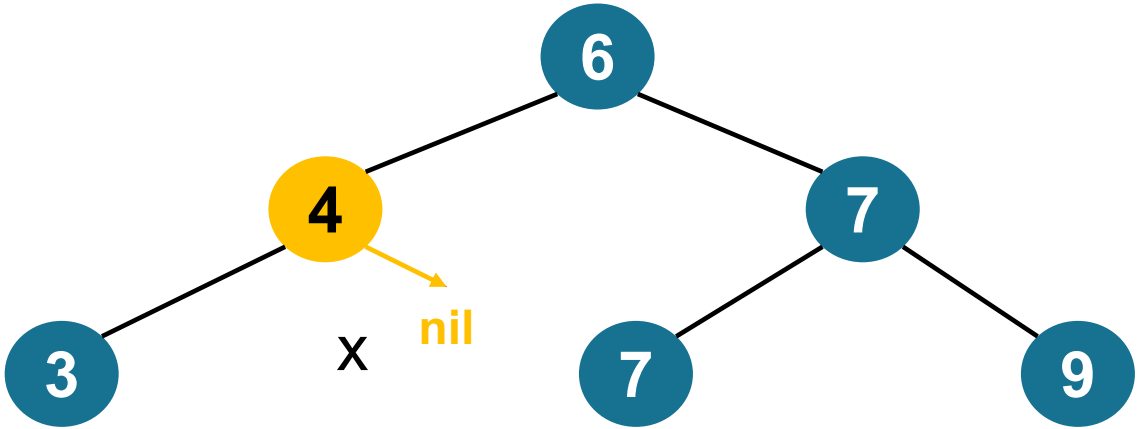


Binäre Suchbäume – Suchen

IterativeBaumsuche(x,k)

1. **while** x ≠ **nil and** k ≠ key(x) *do*
2. **if** k < key(x) **then** x ← lc(x)
3. **else** x ← rc(x)
4. **return** x

IterativeBaumsuche(root(T),5)

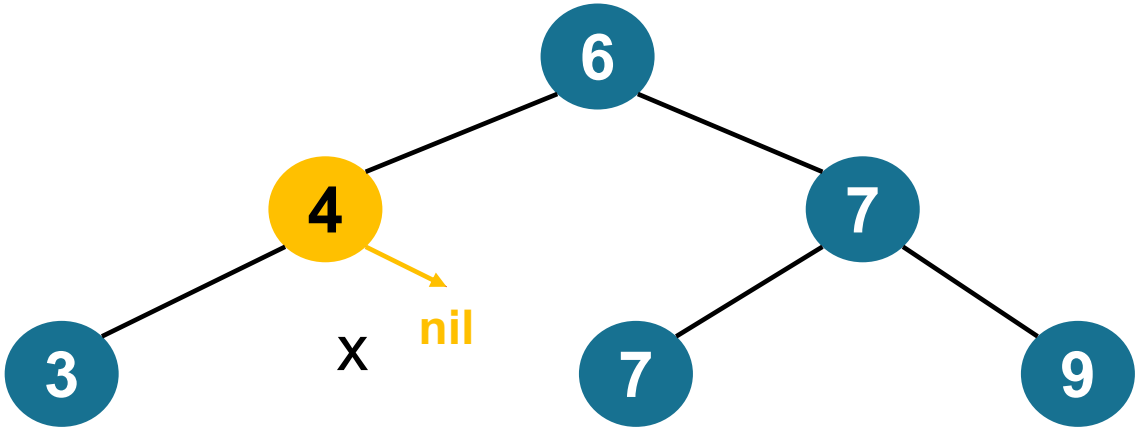


Binäre Suchbäume – Suchen

IterativeBaumsuche(x,k)

1. **while** x ≠ **nil and** k ≠ key(x) *do*
2. **if** k < key(x) **then** x ← lc(x)
3. **else** x ← rc(x)
4. **return** x

IterativeBaumsuche(root(T),5)

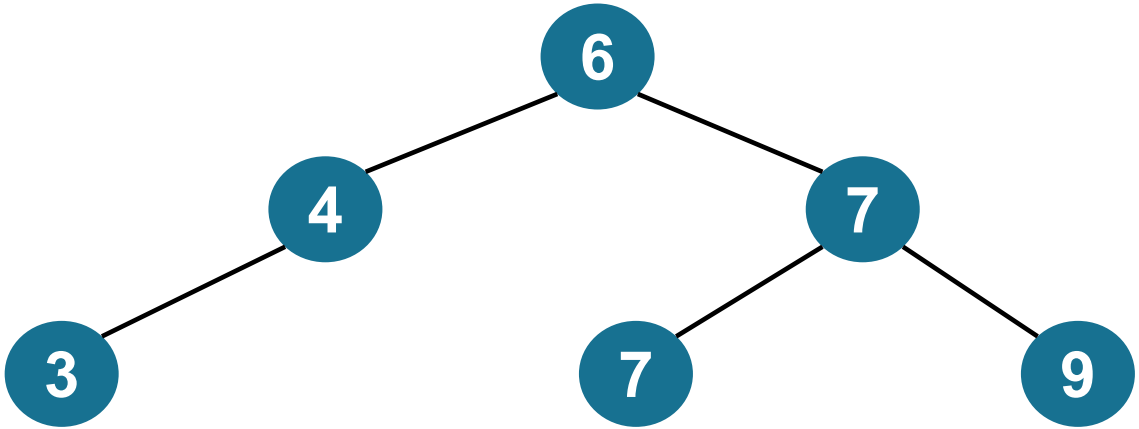


Binäre Suchbäume – Suchen

IterativeBaumsuche(x,k)

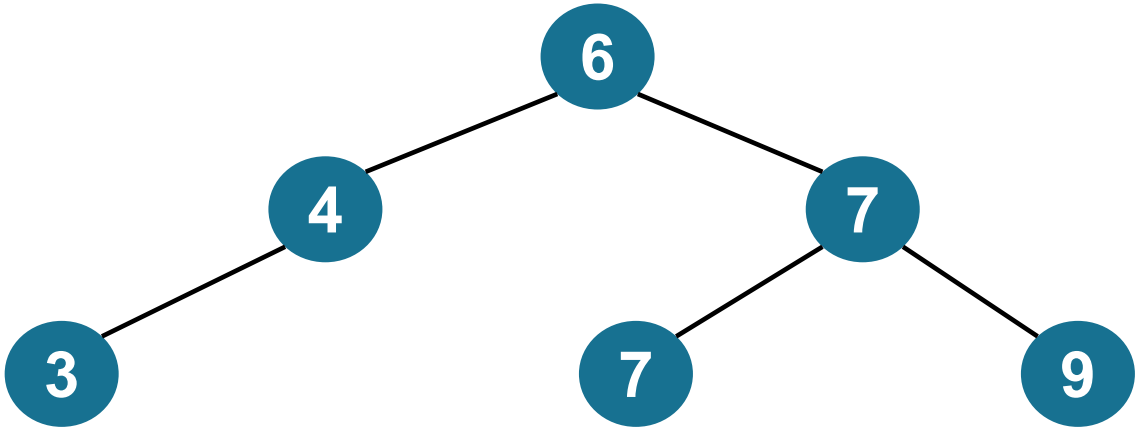
1. **while** x ≠ **nil and** k ≠ key(x) *do*
2. **if** k < key(x) **then** x ← lc(x)
3. **else** x ← rc(x)
4. **return** x

Funktionsweise wie (rekursive) Baumsuche. Laufzeit: O(h)



Binäre Suchbäume
Min/Max Suche

- Minimum- und Maximumsuche
  - Suchbaumeigenschaft:
    Alle Knoten im rechten Unterbaum eines Knotens x sind > key(x)
  - Alle Knoten im linken Unterbaum von x sind ≤ key(x)

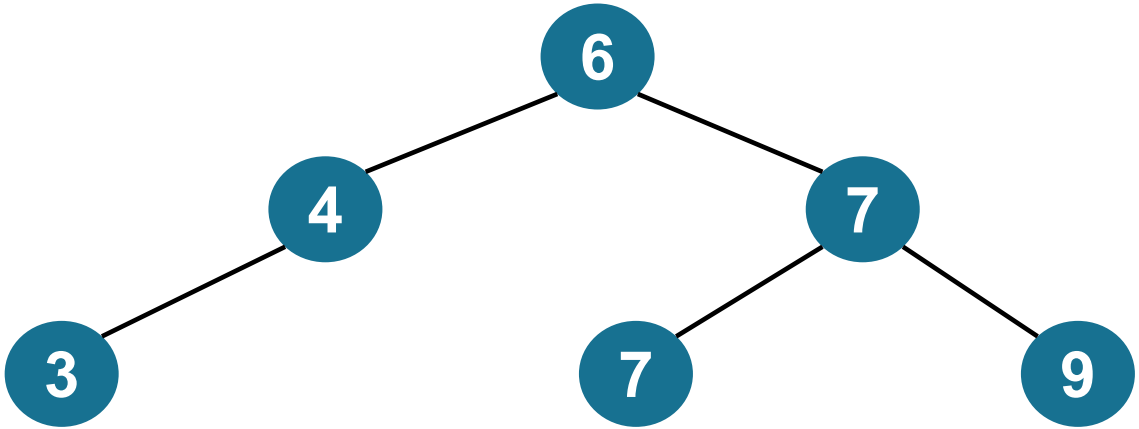


Binäre Suchbäume
Min/Max Suche

MinimumSuche(x)

Aufruf mit Wurzelknoten

1. **while** lc(x) ≠ **nil do** x ← lc(x)
2. **return** x



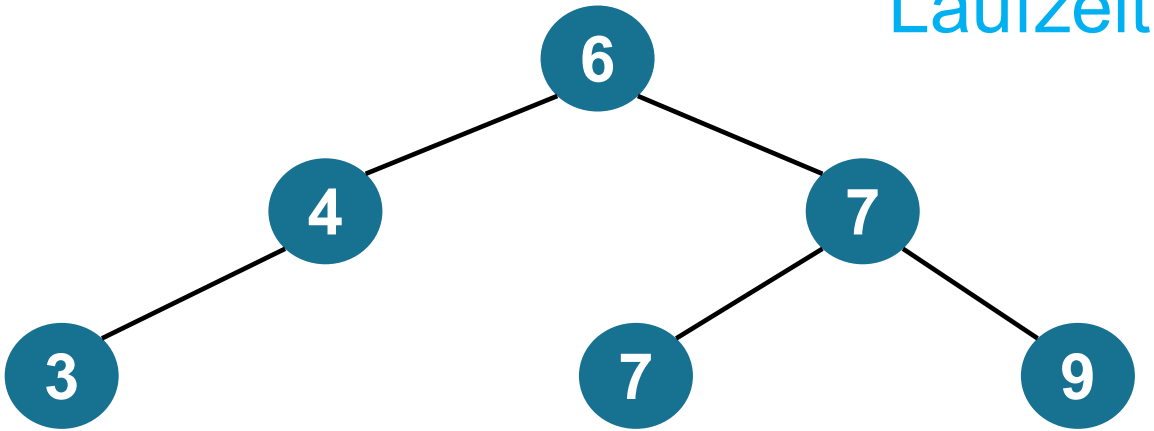
Binäre Suchbäume
Min/Max Suche

MinimumSuche(x)

Aufruf mit Wurzelknoten

1. **while** lc(x) ≠ **nil do** x ← lc(x)
2. **return** x

Laufzeit: O(h)



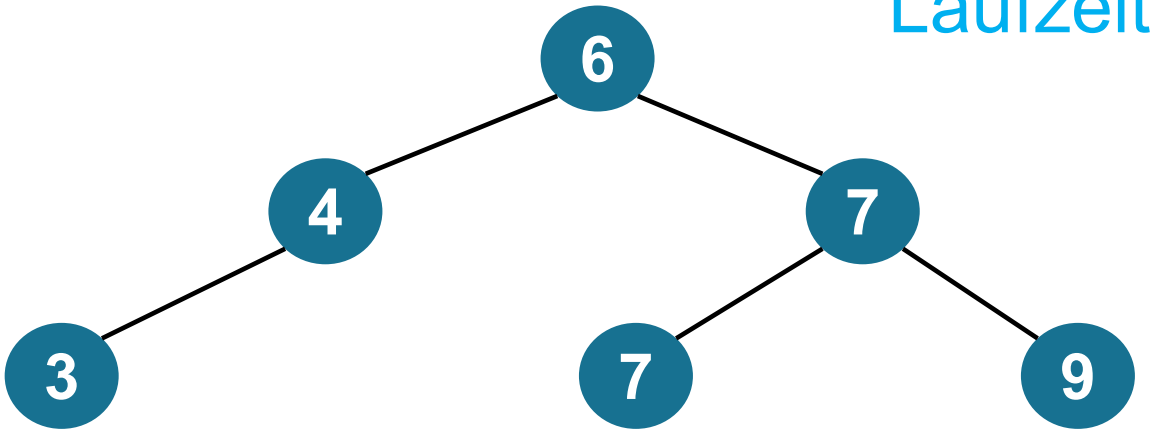
Binäre Suchbäume
Min/Max Suche

MaximumSuche(x)

Aufruf mit Wurzelknoten

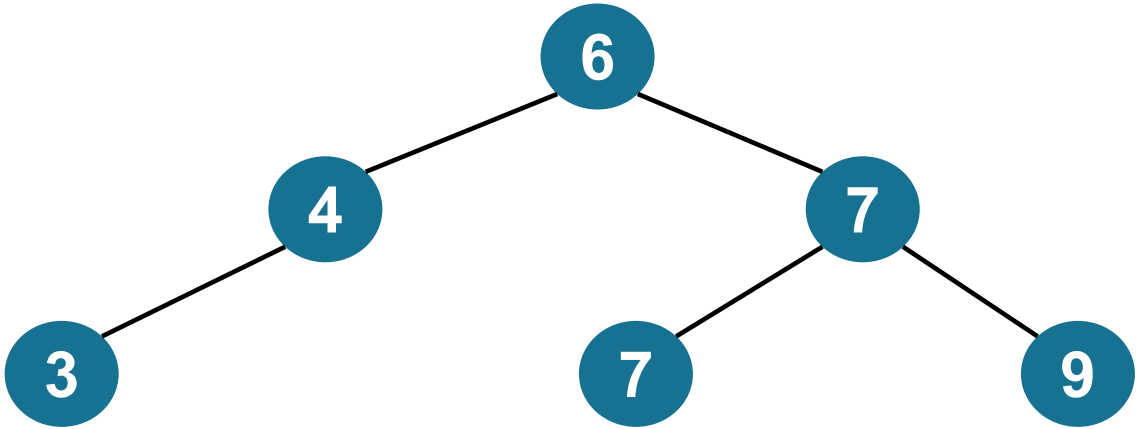
1. **while** rc(x) ≠ **nil do** x ← rc(x)
2. **return** x

Laufzeit: O(h)



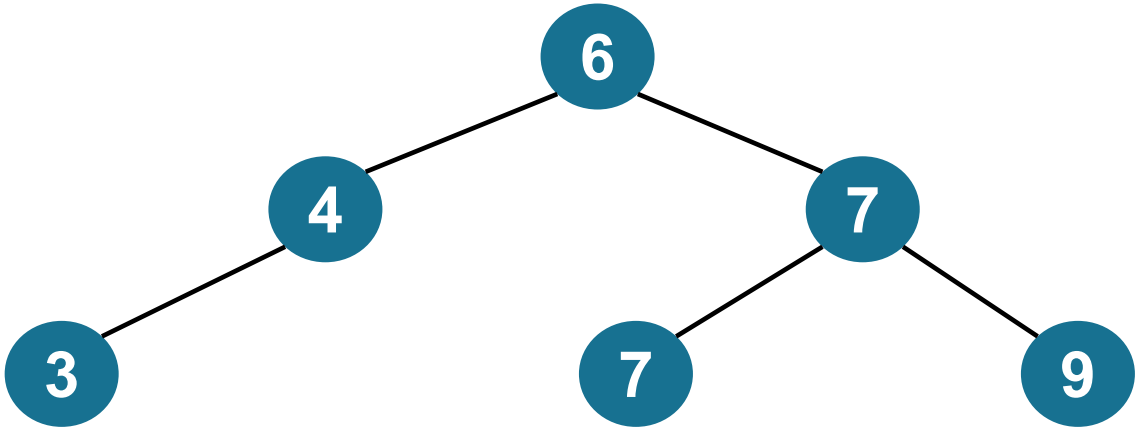
Binäre Suchbäume
Nachfolgersuche

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



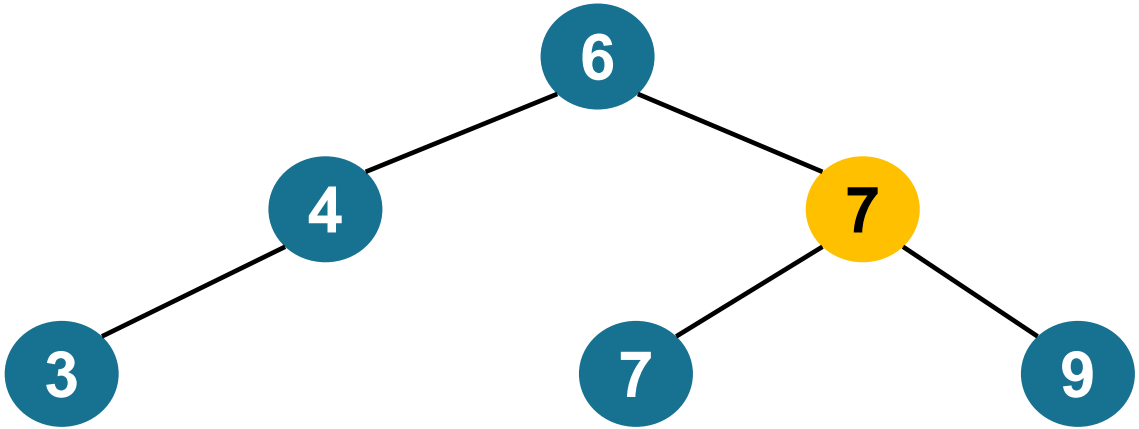
Binäre Suchbäume
Nachfolgersuche

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



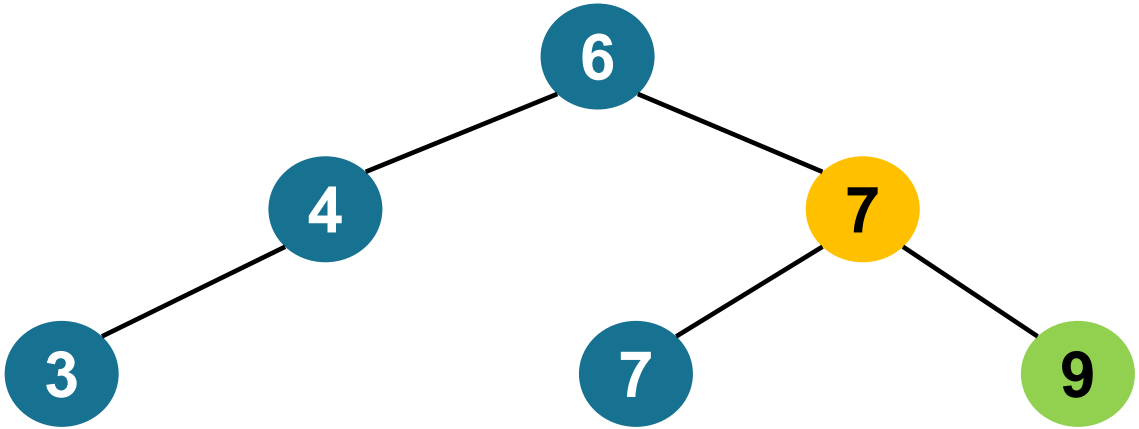
Binäre Suchbäume
Nachfolgersuche

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



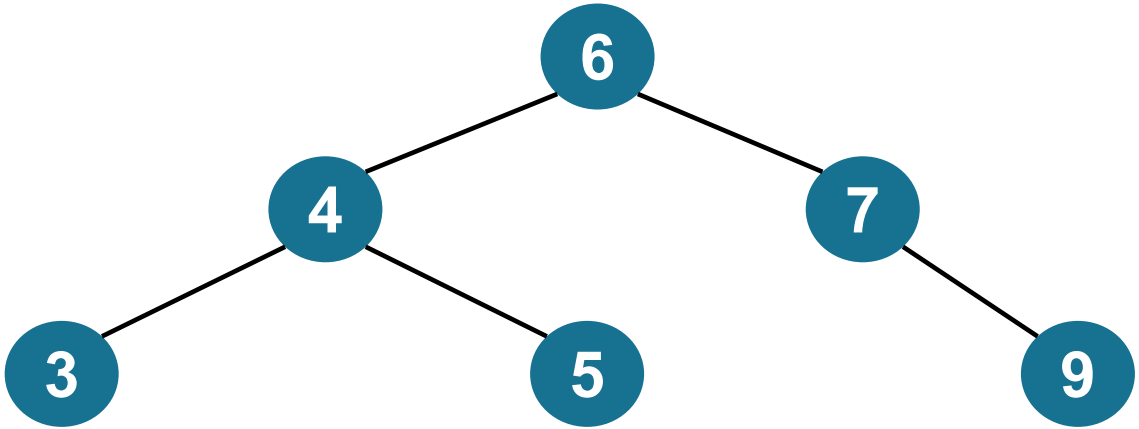
Binäre Suchbäume
Nachfolgersuche

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



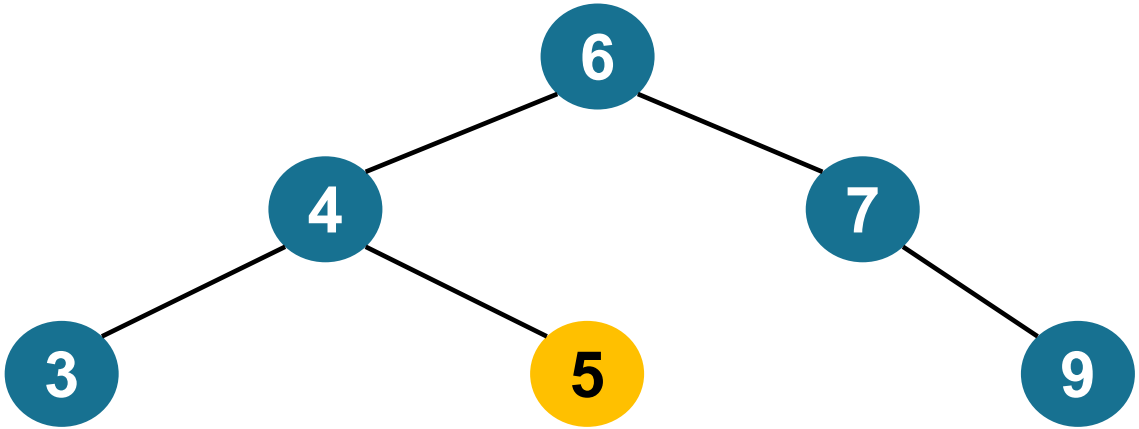
Binäre Suchbäume
Nachfolgersuche

- 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



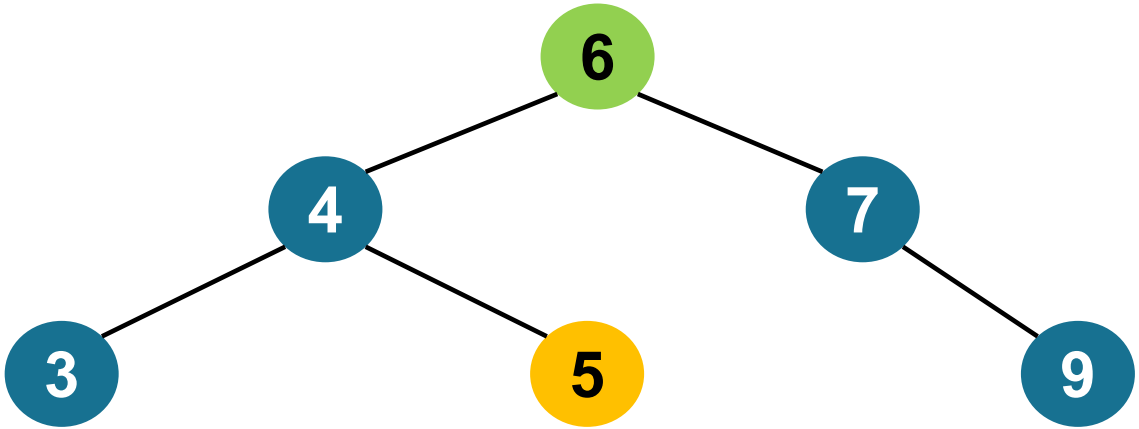
Binäre Suchbäume
Nachfolgersuche

- 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



Binäre Suchbäume
Nachfolgersuche

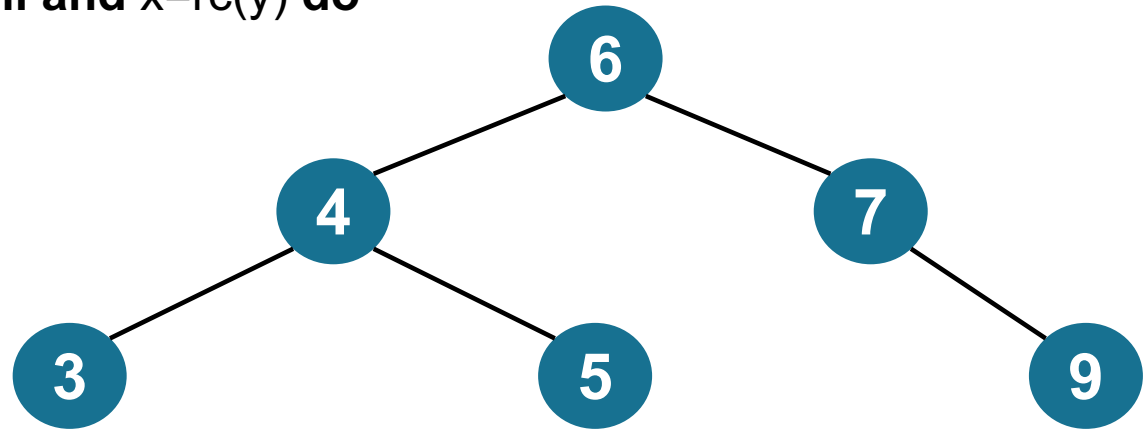
- Nachfolgersuche
  - Fall 2 (rechter Unterbaum von x leer und x hat Nachfolger y): Dann ist y der erste Knoten auf dem Pfad zur Wurzel, der größer als x ist



Binäre Suchbäume
Nachfolgersuche

Nachfolgersuche(x)

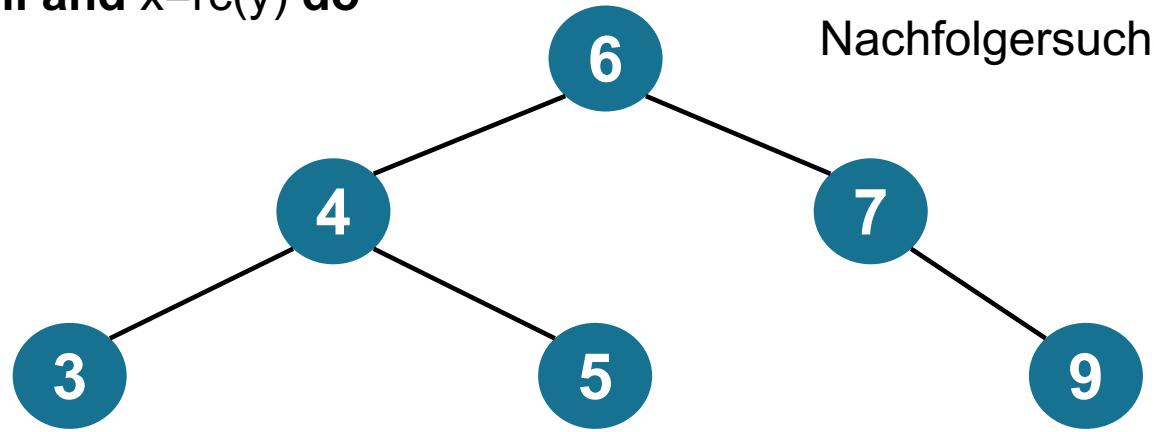
1. **if** rc(x) ≠ **nil then return** MinimumSuche(rc(x))
2. y ← p(x)
3. **while** y ≠ **nil and** x=rc(y) **do**
4. x ← y
5. y ← p(y)
6. **return** y



Binäre Suchbäume
Nachfolgersuche

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

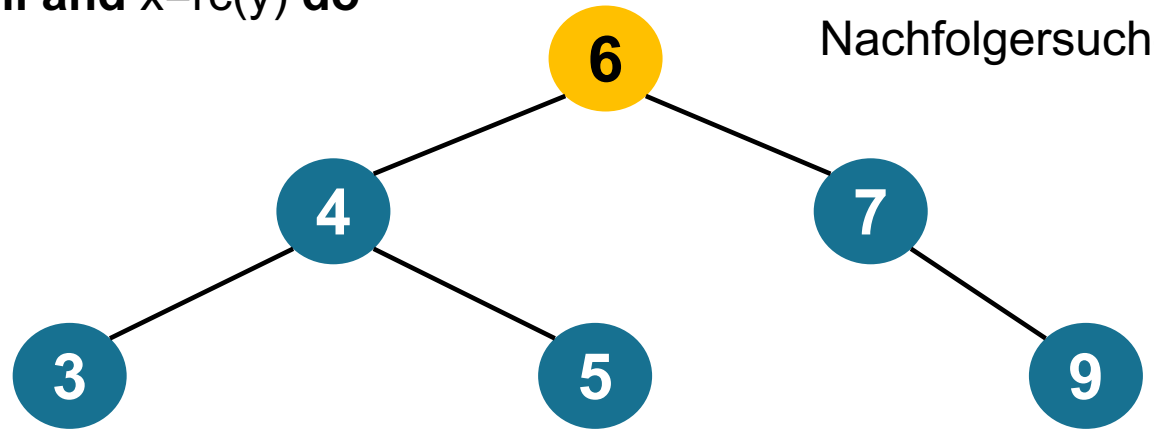


Nachfolgersuche(6)

Binäre Suchbäume
Nachfolgersuche

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

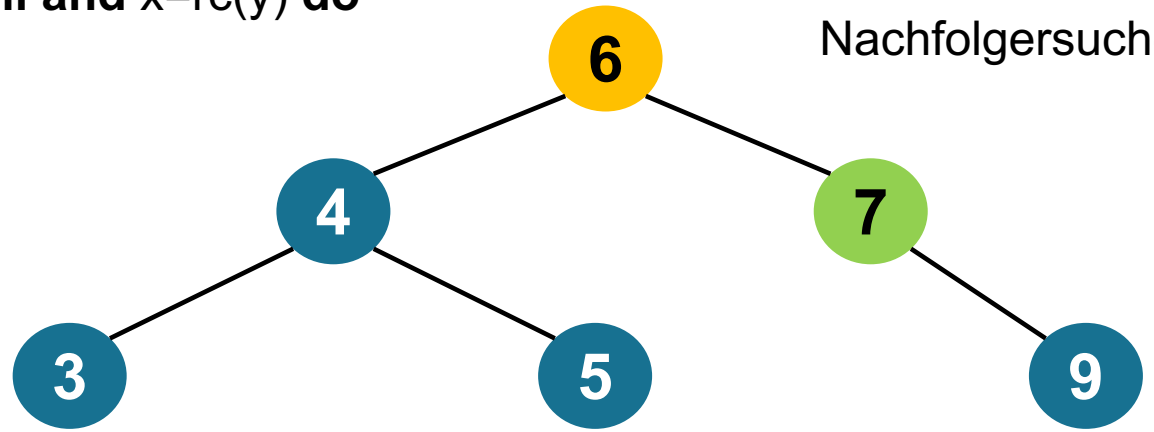


Nachfolgersuche(6)

Binäre Suchbäume
Nachfolgersuche

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

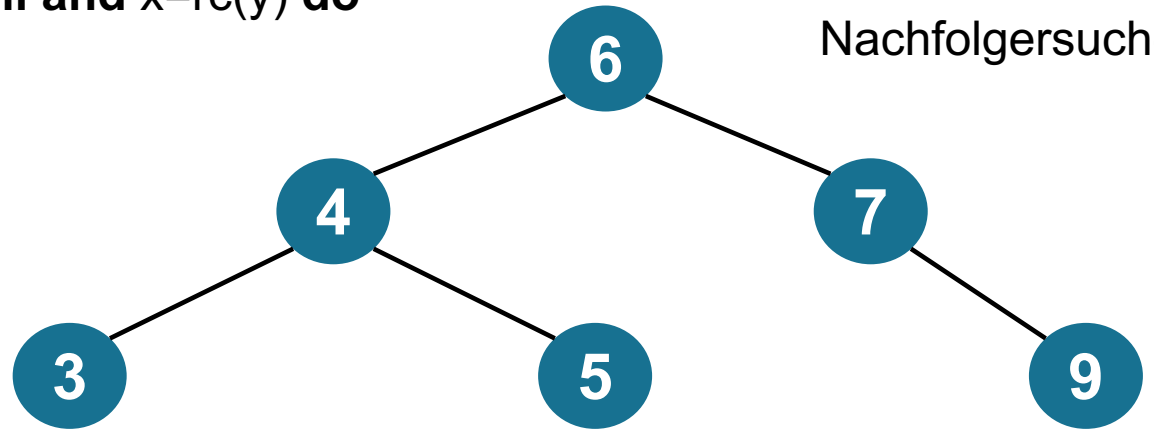


Nachfolgersuche(6)

Binäre Suchbäume
Nachfolgersuche

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



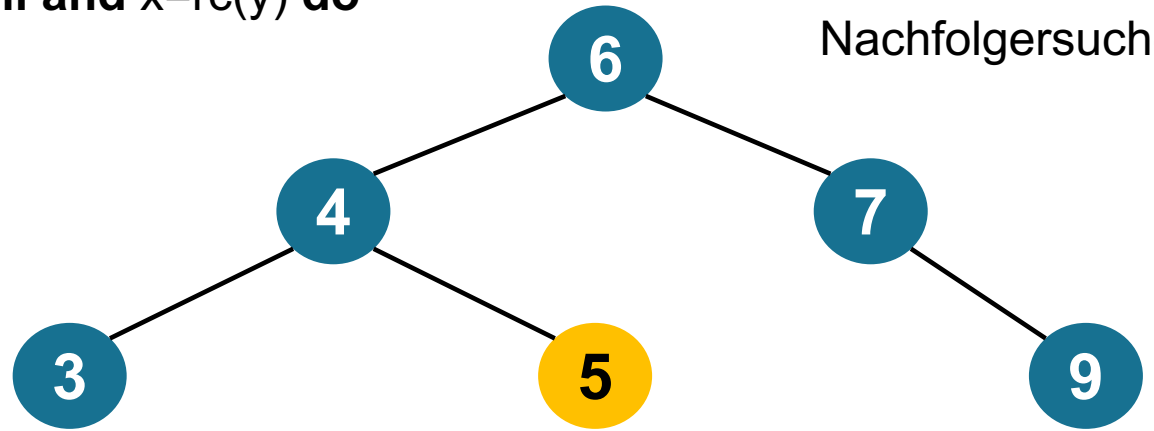
Nachfolgersuche(5)

Binäre Suchbäume
Nachfolgersuche

Technische Universität Berlin

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(5)

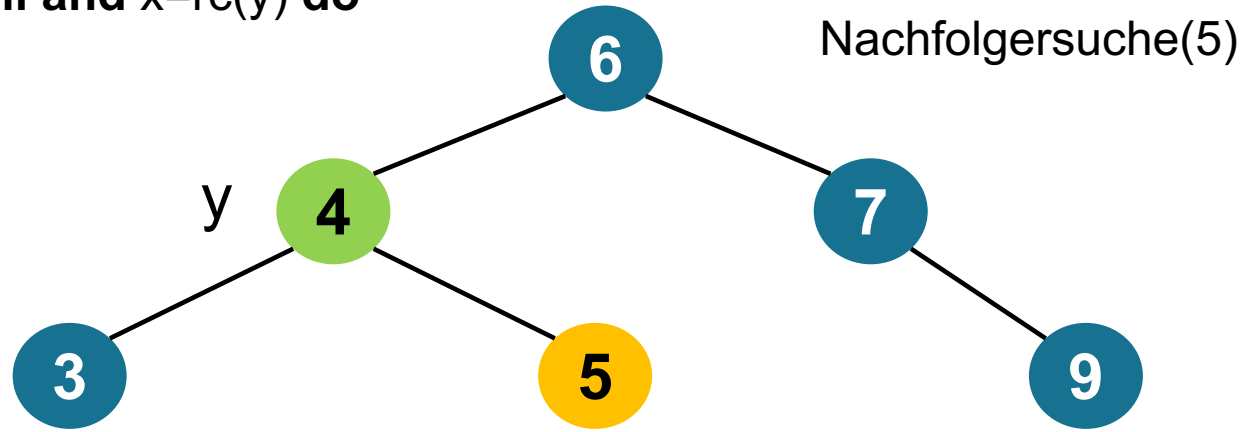
Open Distributed Sytems ODS

Binäre Suchbäume
Nachfolgersuche

Technische Universität Berlin

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

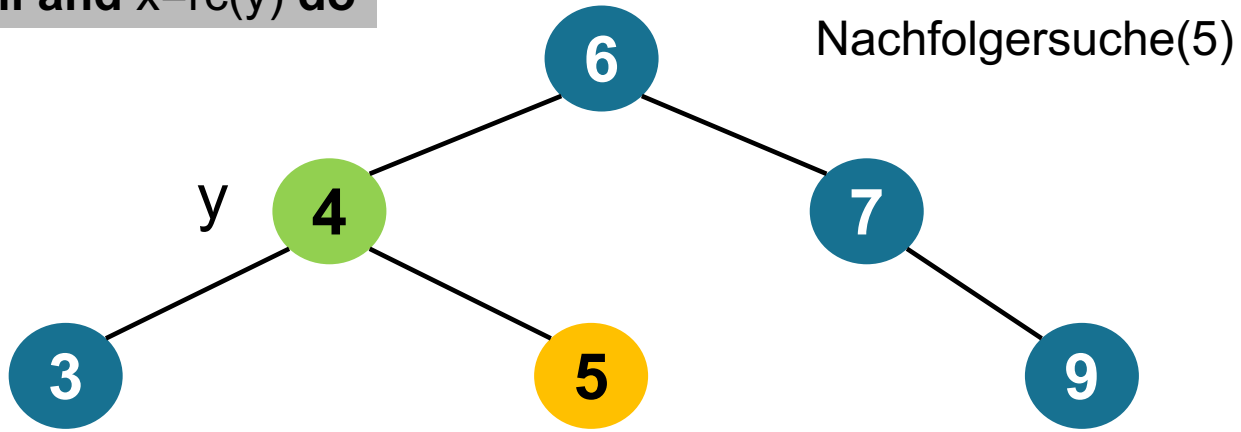


Open Distributed Sytems ODS

Binäre Suchbäume
Nachfolgersuche

Nachfolgersuche(x)

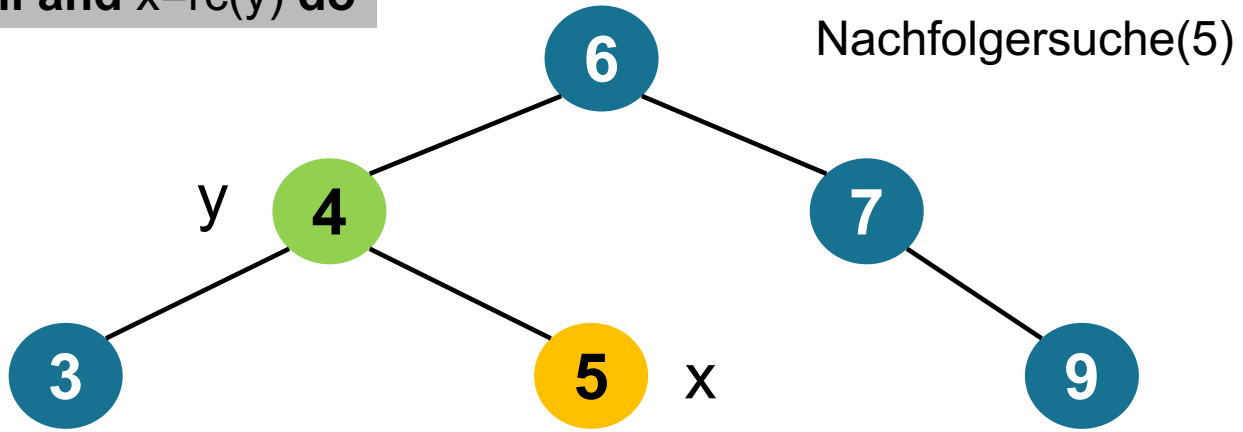
1. **if** rc(x) ≠ **nil then return** MinimumSuche(rc(x))
2. y ← p(x)
3. **while** y ≠ **nil and** x=rc(y) **do**
4. x ← y
5. y ← p(y)
6. **return** y



Binäre Suchbäume
Nachfolgersuche

Nachfolgersuche(x)

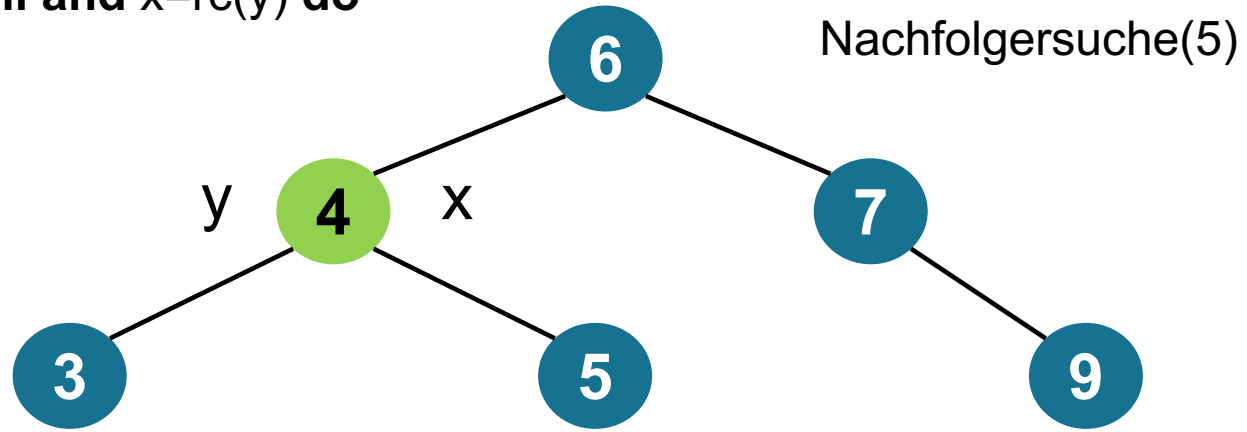
1. **if** rc(x) ≠ **nil then return** MinimumSuche(rc(x))
2. y ← p(x)
3. **while** y ≠ **nil and** x=rc(y) **do**
4. x ← y
5. y ← p(y)
6. **return** y



Binäre Suchbäume
Nachfolgersuche

Nachfolgersuche(x)

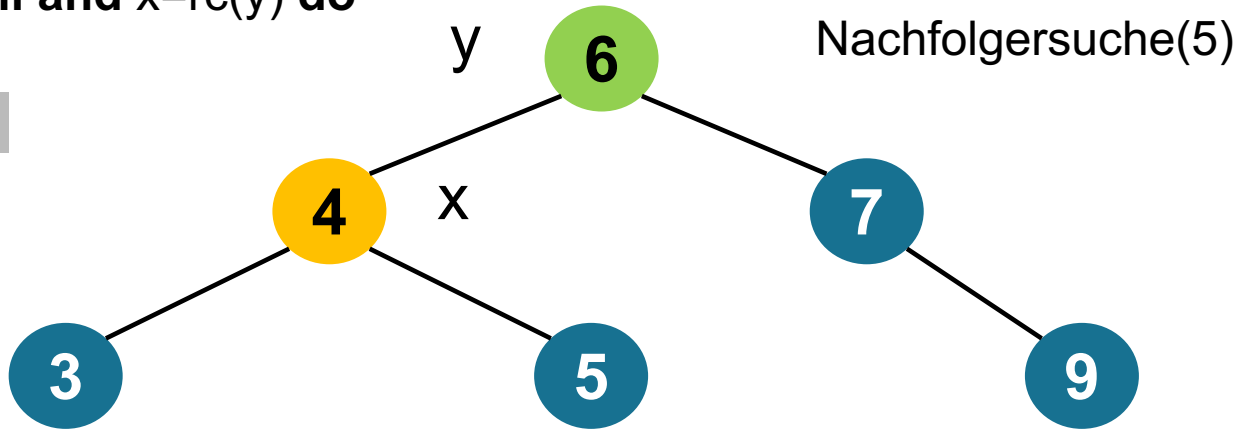
1. **if** rc(x) ≠ **nil then return** MinimumSuche(rc(x))
2. y ← p(x)
3. **while** y ≠ **nil and** x=rc(y) **do**
4. x ← y
5. y ← p(y)
6. **return** y



Binäre Suchbäume
Nachfolgersuche

Nachfolgersuche(x)

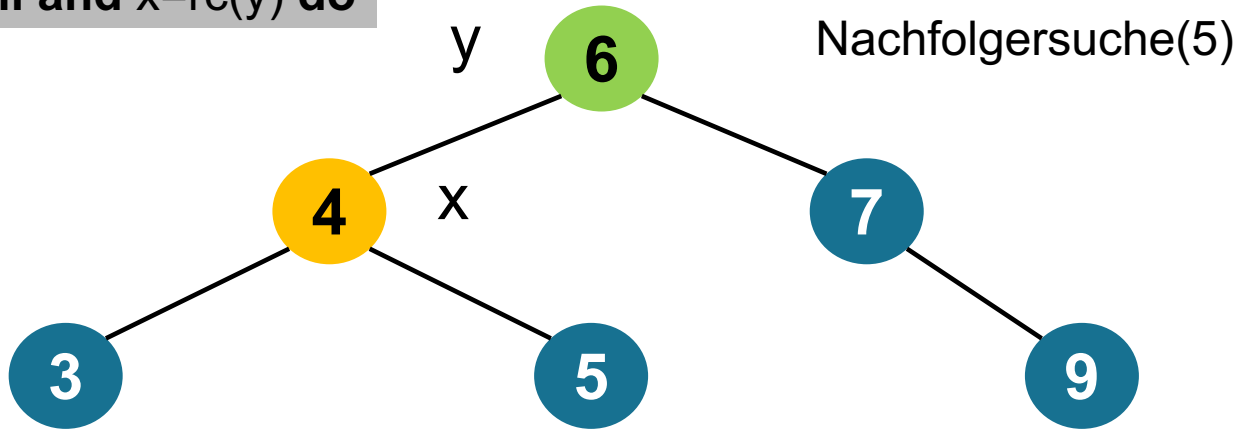
1. **if** rc(x) ≠ **nil then return** MinimumSuche(rc(x))
2. y ← p(x)
3. **while** y ≠ **nil and** x=rc(y) **do**
4. x ← y
5. y ← p(y)
6. **return** y



Binäre Suchbäume
Nachfolgersuche

Nachfolgersuche(x)

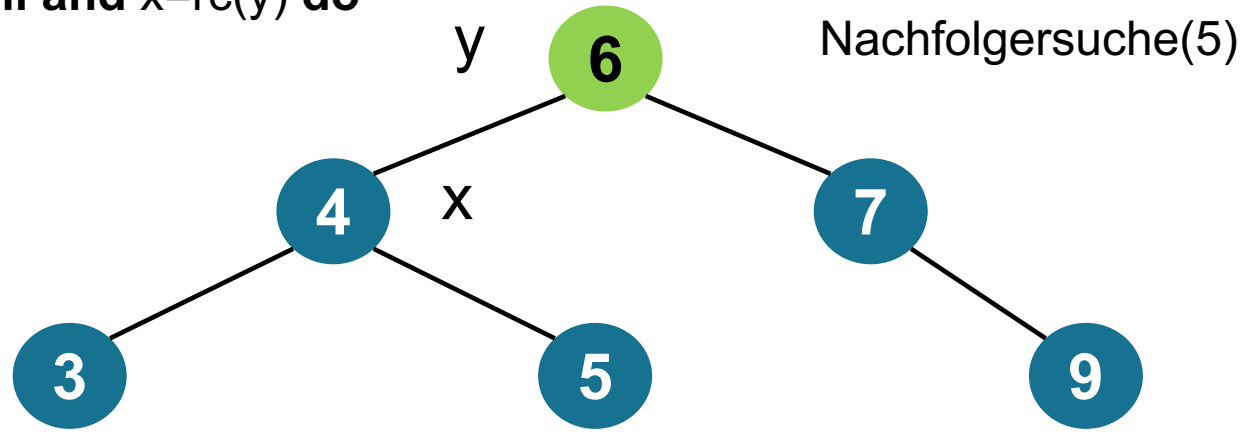
1. **if** rc(x) ≠ **nil then return** MinimumSuche(rc(x))
2. y ← p(x)
3. **while** y ≠ **nil and** x=rc(y) **do**
4. x ← y
5. y ← p(y)
6. **return** y



Binäre Suchbäume
Nachfolgersuche

Nachfolgersuche(x)

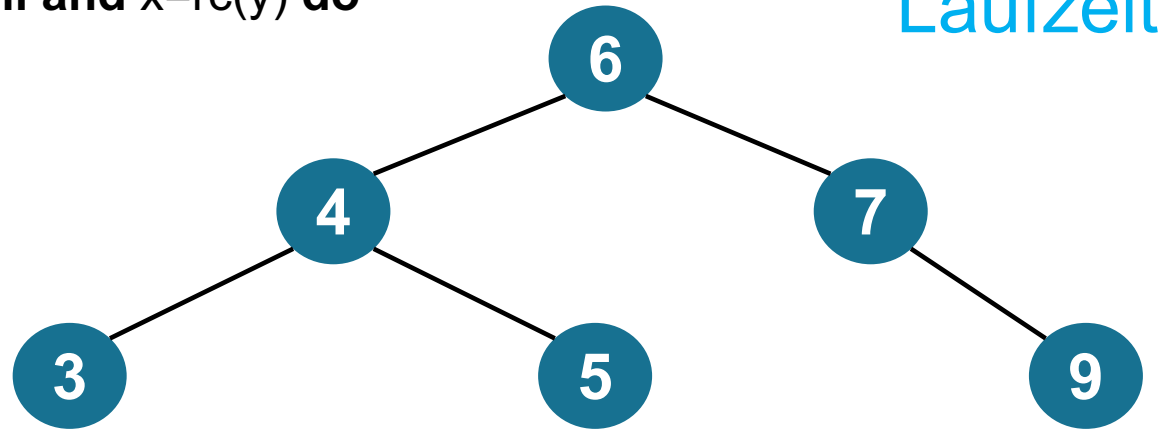
1. **if** rc(x) ≠ **nil then return** MinimumSuche(rc(x))
2. y ← p(x)
3. **while** y ≠ **nil and** x=rc(y) **do**
4. x ← y
5. y ← p(y)
6. **return** y



Binäre Suchbäume
Nachfolgersuche

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

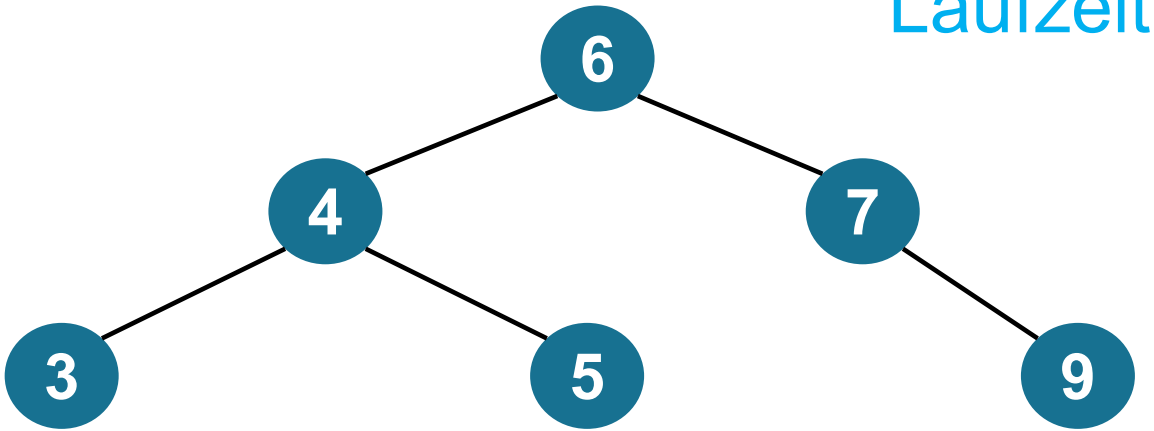


Laufzeit: O(h)

Binäre Suchbäume
Vorgängersuche

- Vorgängersuche
  - Analog zur Nachfolgersuche
  - Daher ebenfalls O(h) Laufzeit

Laufzeit: O(h)



Dynamische Bäume

- Ein grundlegendes Datenbank-Problem
  - Speicherung von Datensätzen
- Beispiel
  - Kundendaten (Name, Adresse, Wohnort, Kundennummer, offene Rechnungen, offene Bestellungen,…)
- Anforderungen
  - Schneller Zugriff
  - Schnelles Einfügen neuer Datensätze
  - Schnelles Löschen bestehender Datensätze

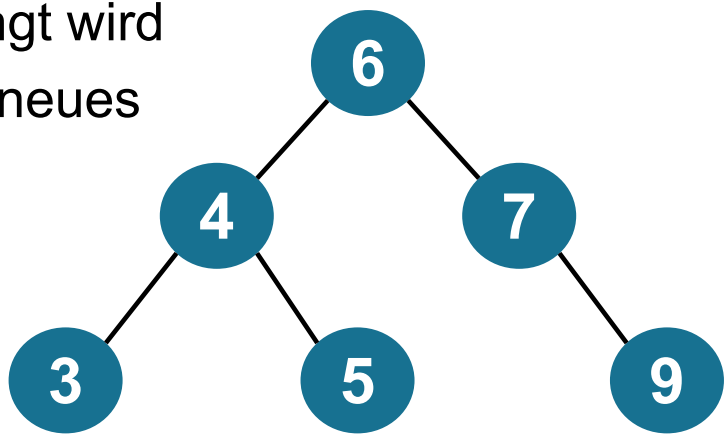
Dynamische Baumoperationen

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

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

Binäre Suchbäume – Einfügen

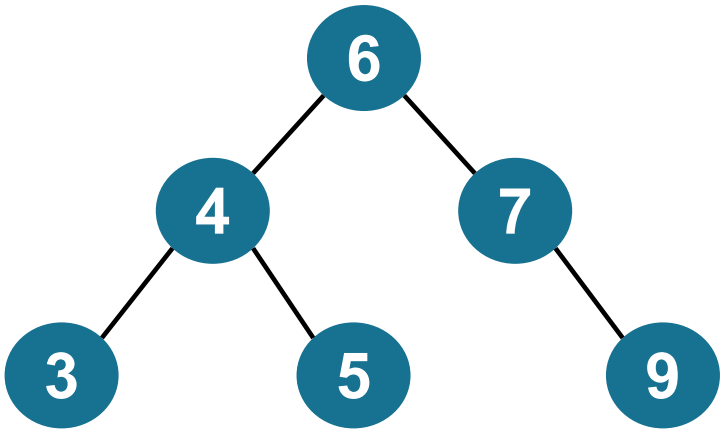
- Einfügen
  - Ähnlich wie Baumsuche: Finde Blatt, an das neuer Knoten angehängt wird
  - Danach wird **nil**-Zeiger durch neues Element ersetzt



Binäre Suchbäume – Einfügen

Technische Universität Berlin

```
Einfügen(T,z)
01. y ← nil; x ← root(T)
02. while x ≠ nil do
03.     y ← x
04.     if key(z) <= key(x) then x ← lc(x)
05.     else x ← rc(x)
06. p(z) ← y
07. if y=nil then root(T) ← z
08. else
09.     if key(z)<= key(y) then lc(y) ← z
10.     else rc(y) ← z
```



Open Distributed Sytems
ODS

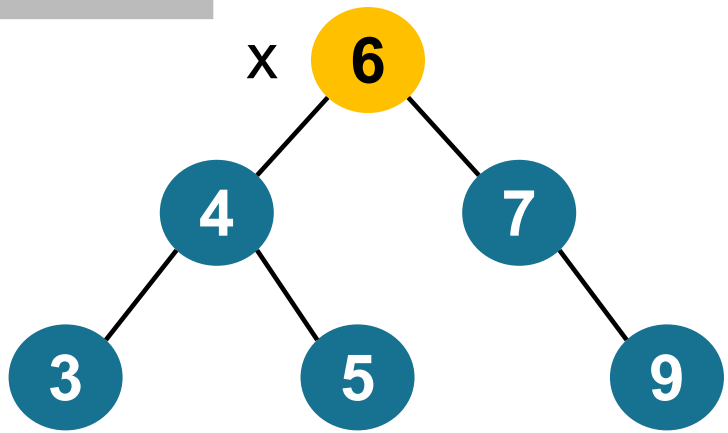
Binäre Suchbäume – Einfügen

Technische Universität Berlin

```
Einfügen(T,z)
01. y ← nil; x ← root(T)
02. while x ≠ nil do
03.     y ← x
04.     if key(z) <= key(x) then x ← lc(x)
05.     else x ← rc(x)
06. p(z) ← y
07. if y=nil then root(T) ← z
08. else
09.     if key(z)<= key(y) then lc(y) ← z
10.     else rc(y) ← z
```

y wird Elter des einzufügenden Elements.

Einfügen(8)



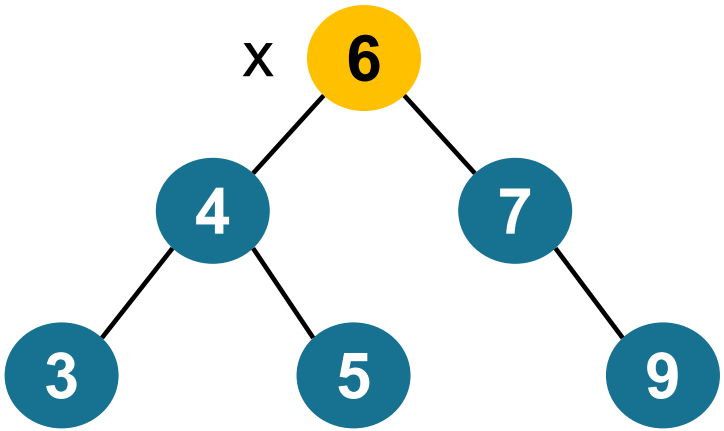
Open Distributed Sytems ODS

Binäre Suchbäume – Einfügen

Technische Universität Berlin

```
Einfügen(T,z)
01.  y ← nil; x ← root(T)
02.  while x ≠ nil do
03.      y ← x
04.      if key(z) <= key(x) then x ← lc(x)
05.      else x ← rc(x)
06.  p(z) ← y
07.  if y=nil then root(T) ← z
08.  else
09.      if key(z)<= key(y) then lc(y) ← z
10.      else rc(y) ← z
```

Einfügen(8)



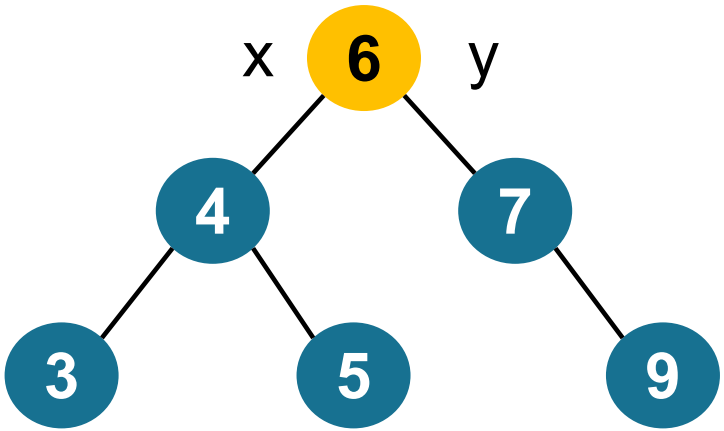
Open Distributed Sytems ODS

Binäre Suchbäume – Einfügen

Technische Universität Berlin

```
Einfügen(T,z)
01.  y ← nil; x ← root(T)
02.  while x ≠ nil do
03.      y ← x
04.      if key(z) <= key(x) then x ← lc(x)
05.      else x ← rc(x)
06.  p(z) ← y
07.  if y=nil then root(T) ← z
08.  else
09.      if key(z)<= key(y) then lc(y) ← z
10.      else rc(y) ← z
```

Einfügen(8)



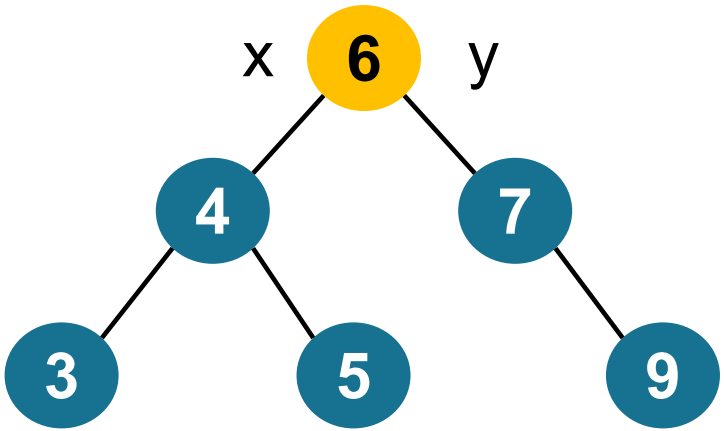
Open Distributed Sytems ODS

Binäre Suchbäume – Einfügen

Technische Universität Berlin

```
Einfügen(T,z)
01. y ← nil; x ← root(T)
02. while x ≠ nil do
03.     y ← x
04.     if key(z) <= key(x) then x ← lc(x)
05.     else x ← rc(x)
06. p(z) ← y
07. if y=nil then root(T) ← z
08. else
09.     if key(z)<= key(y) then lc(y) ← z
10.     else rc(y) ← z
```

Einfügen(8)



Open Distributed Sytems ODS

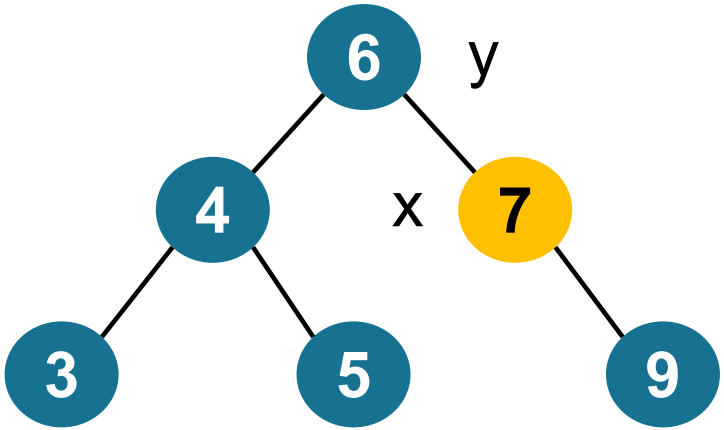
Binäre Suchbäume – Einfügen

Technische Universität Berlin

Einfügen(T,z)

01. y ← **nil**; x ← root(T)
02. **while** x ≠ **nil do**
03. y ← x
04. **if** key(z) <= key(x) **then** x ← lc(x)
05. **else** x ← rc(x)
06. p(z) ← y
07. **if** y=**nil then** root(T) ← z
08. **else**
09. **if** key(z)<= key(y) **then** lc(y) ← z
10. **else** rc(y) ← z

Einfügen(8)



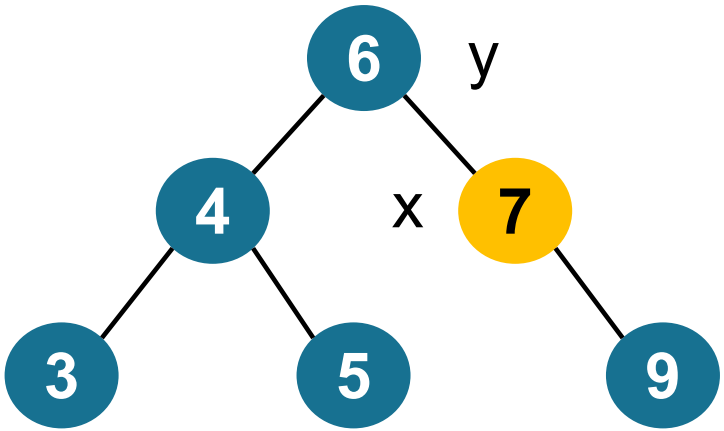
Open Distributed Sytems
ODS

Binäre Suchbäume – Einfügen

Technische Universität Berlin

```
Einfügen(T,z)
01.  y ← nil; x ← root(T)
02.  while x ≠ nil do
03.      y ← x
04.      if key(z) <= key(x) then x ← lc(x)
05.      else x ← rc(x)
06.  p(z) ← y
07.  if y=nil then root(T) ← z
08.  else
09.      if key(z)<= key(y) then lc(y) ← z
10.      else rc(y) ← z
```

Einfügen(8)



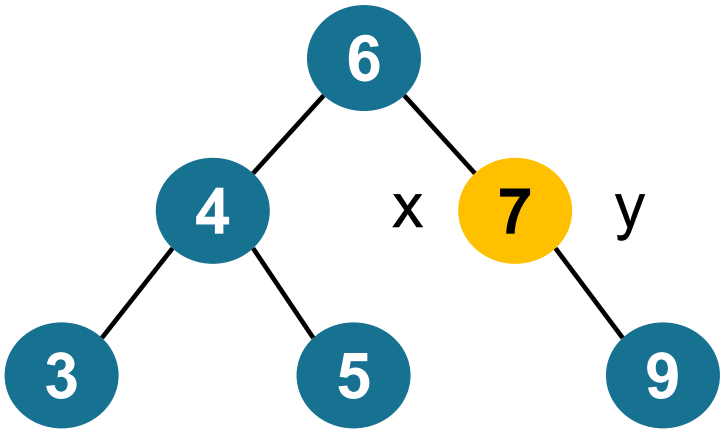
Open Distributed Sytems
ODS

Binäre Suchbäume – Einfügen

Technische Universität Berlin

```
Einfügen(T,z)
01.  y ← nil; x ← root(T)
02.  while x ≠ nil do
03.      y ← x
04.      if key(z) <= key(x) then x ← lc(x)
05.      else x ← rc(x)
06.  p(z) ← y
07.  if y=nil then root(T) ← z
08.  else
09.      if key(z)<= key(y) then lc(y) ← z
10.      else rc(y) ← z
```

Einfügen(8)



Open Distributed Sytems ODS

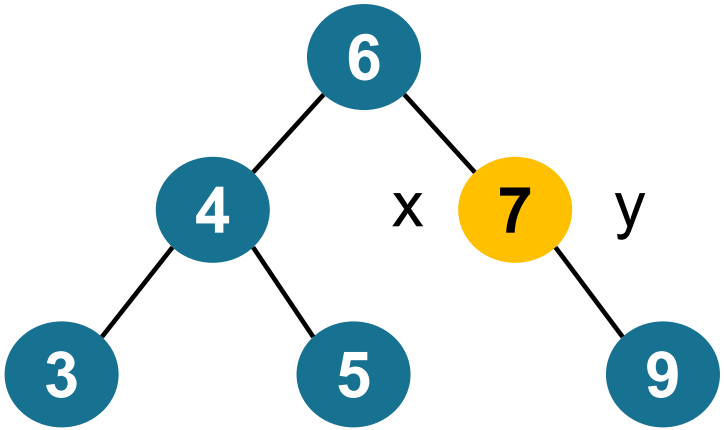
Binäre Suchbäume – Einfügen

Technische Universität Berlin

Einfügen(T,z)

01. y ← **nil**; x ← root(T)
02. **while** x ≠ **nil do**
03. y ← x
04. **if** key(z) <= key(x) **then** x ← lc(x)
05. **else** x ← rc(x)
06. p(z) ← y
07. **if** y=**nil then** root(T) ← z
08. **else**
09. **if** key(z)<= key(y) **then** lc(y) ← z
10. **else** rc(y) ← z

Einfügen(8)



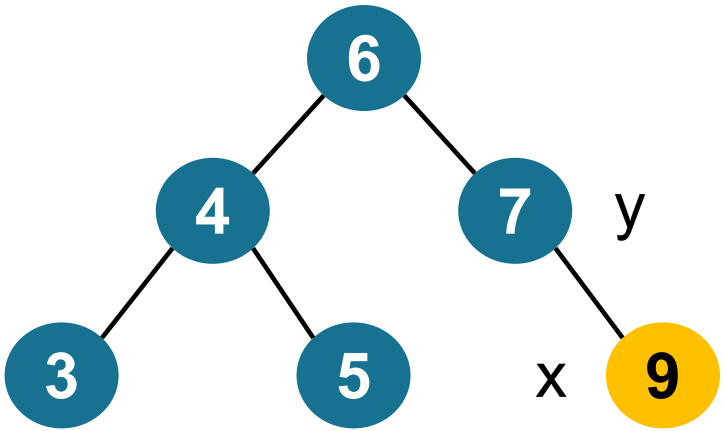
Open Distributed Sytems ODS

Binäre Suchbäume – Einfügen

Technische Universität Berlin

```
Einfügen(T,z)
01. y ← nil; x ← root(T)
02. while x ≠ nil do
03.     y ← x
04.     if key(z) <= key(x) then x ← lc(x)
05.     else x ← rc(x)
06. p(z) ← y
07. if y=nil then root(T) ← z
08. else
09.     if key(z)<= key(y) then lc(y) ← z
10.     else rc(y) ← z
```

Einfügen(8)



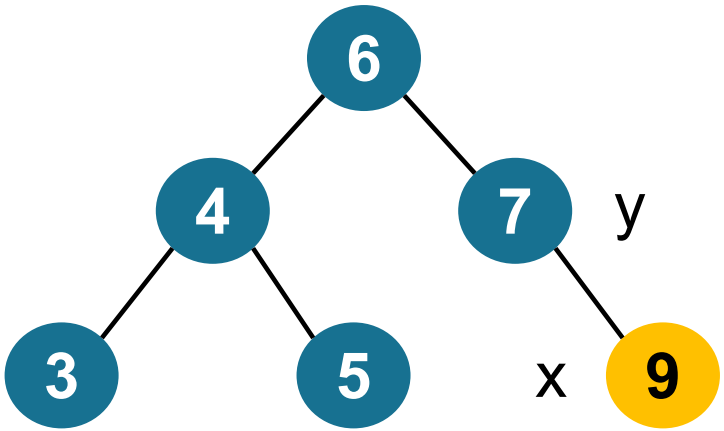
Open Distributed Sytems ODS

Binäre Suchbäume – Einfügen

Technische Universität Berlin

```
Einfügen(T,z)
01.  y ← nil; x ← root(T)
02.  while x ≠ nil do
03.      y ← x
04.      if key(z) <= key(x) then x ← lc(x)
05.      else x ← rc(x)
06.  p(z) ← y
07.  if y=nil then root(T) ← z
08.  else
09.      if key(z)<= key(y) then lc(y) ← z
10.      else rc(y) ← z
```

Einfügen(8)



Open Distributed Sytems
ODS

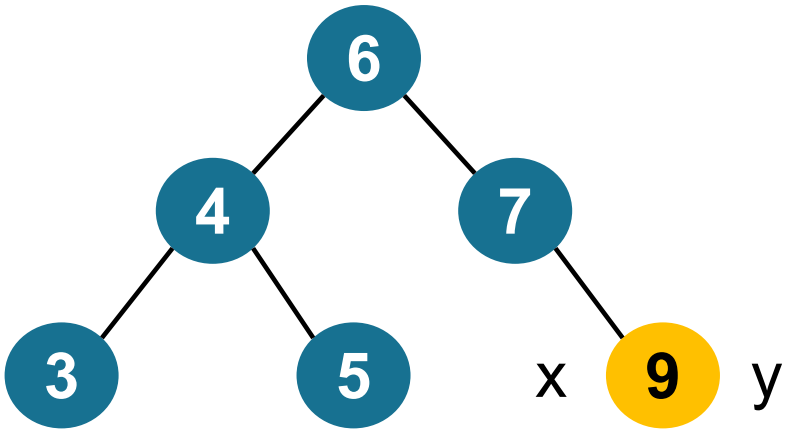
Binäre Suchbäume – Einfügen

Technische Universität Berlin

Einfügen(T,z)

01. y ← **nil**; x ← root(T)
02. **while** x ≠ **nil do**
03. y ← x
04. **if** key(z) <= key(x) **then** x ← lc(x)
05. **else** x ← rc(x)
06. p(z) ← y
07. **if** y=**nil then** root(T) ← z
08. **else**
09. **if** key(z)<= key(y) **then** lc(y) ← z
10. **else** rc(y) ← z

Einfügen(8)



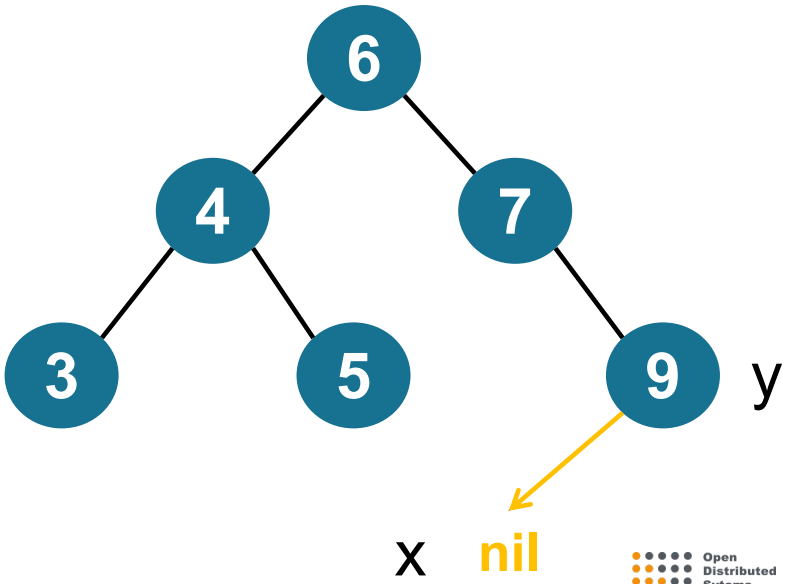
Open Distributed Sytems
ODS

Binäre Suchbäume – Einfügen

Technische Universität Berlin

```
Einfügen(T,z)
01. y ← nil; x ← root(T)
02. while x ≠ nil do
03.     y ← x
04.     if key(z) <= key(x) then x ← lc(x)
05.     else x ← rc(x)
06. p(z) ← y
07. if y=nil then root(T) ← z
08. else
09.     if key(z)<= key(y) then lc(y) ← z
10.     else rc(y) ← z
```

Einfügen(8)



Open Distributed Sytems ODS

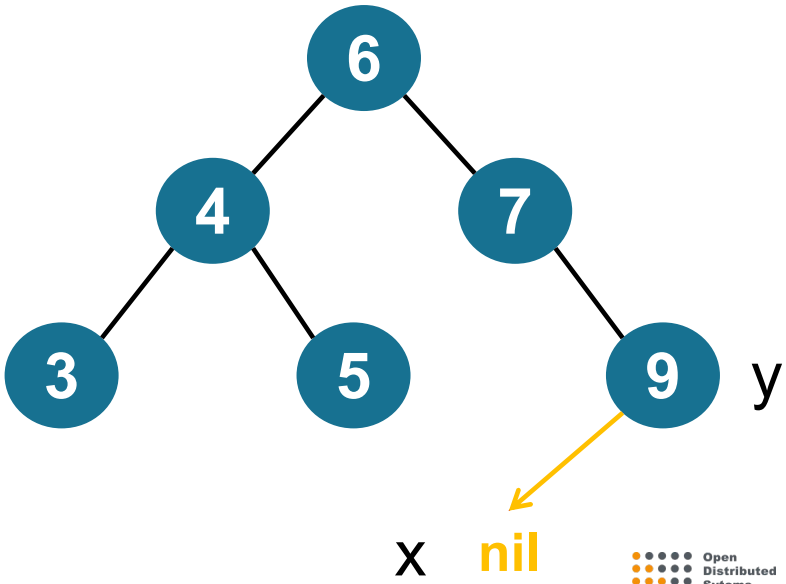
Binäre Suchbäume – Einfügen

Technische Universität Berlin

Einfügen(T,z)

01. y ← **nil**; x ← root(T)
02. **while** x ≠ **nil do**
03. y ← x
04. **if** key(z) <= key(x) **then** x ← lc(x)
05. **else** x ← rc(x)
06. p(z) ← y
07. **if** y=**nil then** root(T) ← z
08. **else**
09. **if** key(z)<= key(y) **then** lc(y) ← z
10. **else** rc(y) ← z

Einfügen(8)



Open Distributed Sytems
ODS

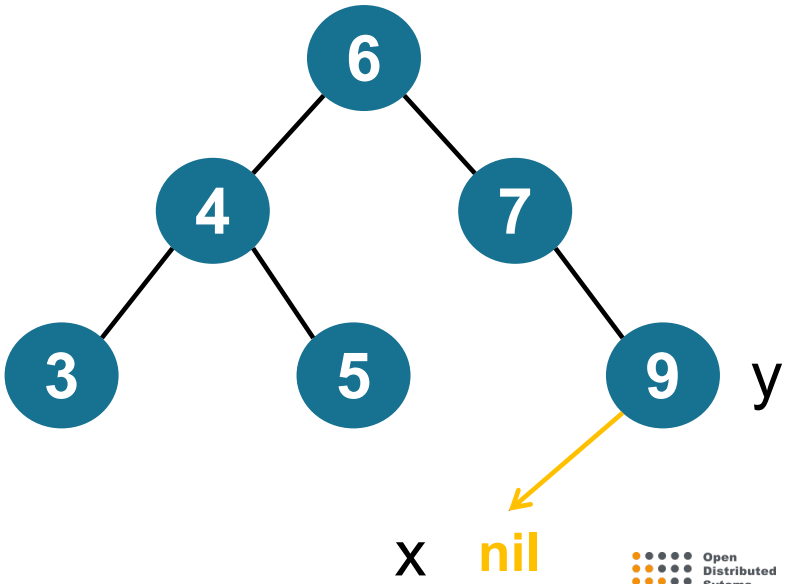
Binäre Suchbäume – Einfügen

Technische Universität Berlin

Einfügen(T,z)

01. y ← **nil**; x ← root(T)
02. **while** x ≠ **nil do**
03. y ← x
04. **if** key(z) <= key(x) **then** x ← lc(x)
05. **else** x ← rc(x)
06. p(z) ← y
07. **if** y=**nil then** root(T) ← z
08. **else**
09. **if** key(z)<= key(y) **then** lc(y) ← z
10. **else** rc(y) ← z

Einfügen(8)



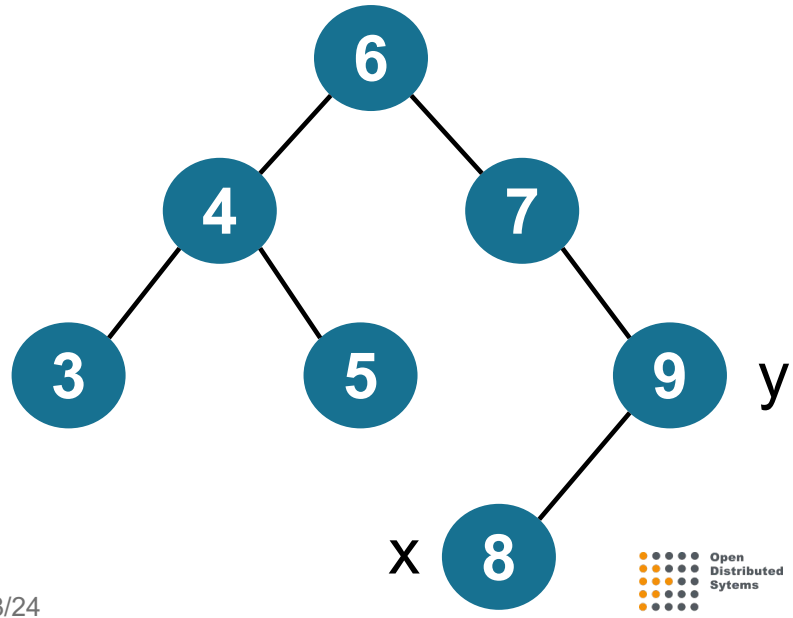
Open Distributed Sytems
ODS

Binäre Suchbäume – Einfügen

Technische Universität Berlin

```
Einfügen(T,z)
01.  y ← nil; x ← root(T)
02.  while x ≠ nil do
03.      y ← x
04.      if key(z) <= key(x) then x ← lc(x)
05.      else x ← rc(x)
06.  p(z) ← y
07.  if y=nil then root(T) ← z
08.  else
09.      if key(z)<= key(y) then lc(y) ← z
10.      else rc(y) ← z
```

Einfügen(8)



Open Distributed Sytems ODS

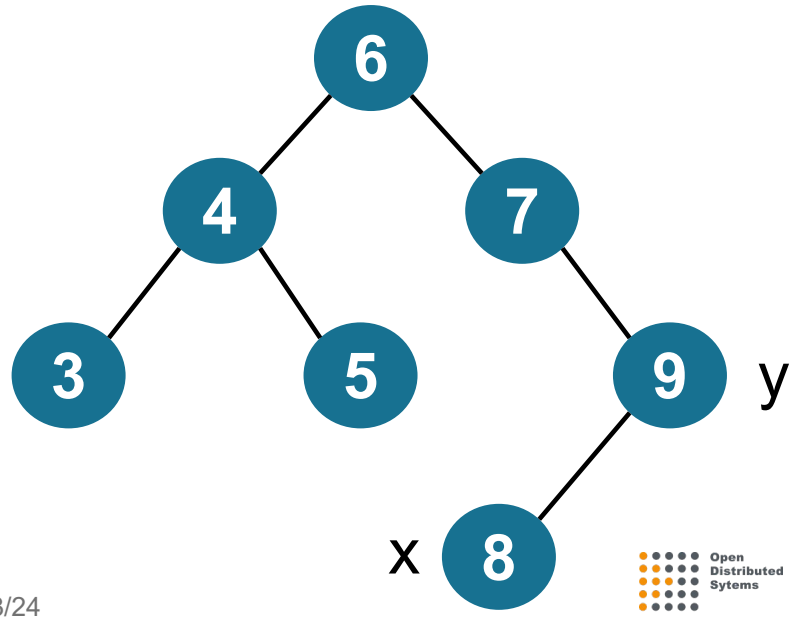
Binäre Suchbäume – Einfügen

Technische Universität Berlin

Einfügen(T,z)

01. y ← **nil**; x ← root(T)
02. **while** x ≠ **nil do**
03. y ← x
04. **if** key(z) <= key(x) **then** x ← lc(x)
05. **else** x ← rc(x)
06. p(z) ← y
07. **if** y=**nil then** root(T) ← z
08. **else**
09. **if** key(z)<= key(y) **then** lc(y) ← z
10. **else** rc(y) ← z

Einfügen(8)



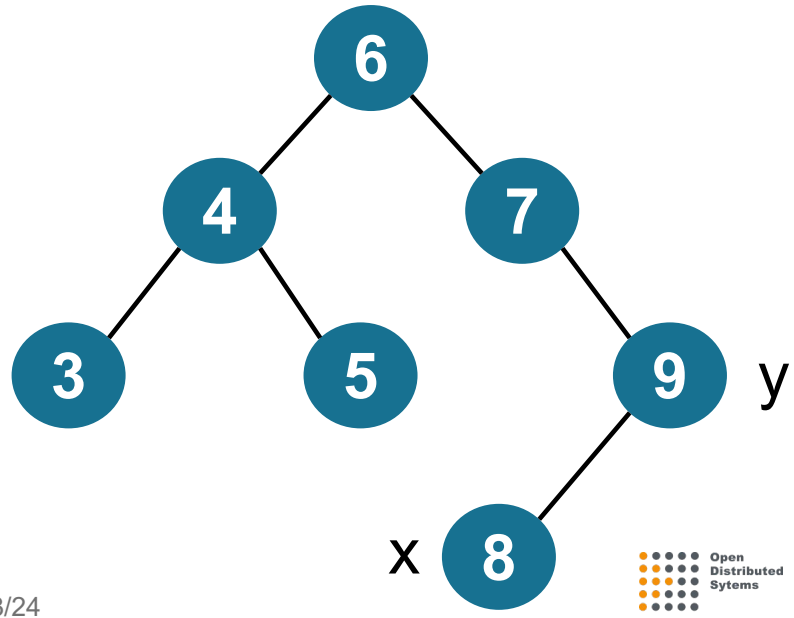
Open Distributed Sytems ODS

Binäre Suchbäume – Einfügen

Technische Universität Berlin

```
Einfügen(T,z)
01. y ← nil; x ← root(T)
02. while x ≠ nil do
03.     y ← x
04.     if key(z) <= key(x) then x ← lc(x)
05.     else x ← rc(x)
06. p(z) ← y
07. if y=nil then root(T) ← z
08. else
09.     if key(z)<= key(y) then lc(y) ← z
10.     else rc(y) ← z
```

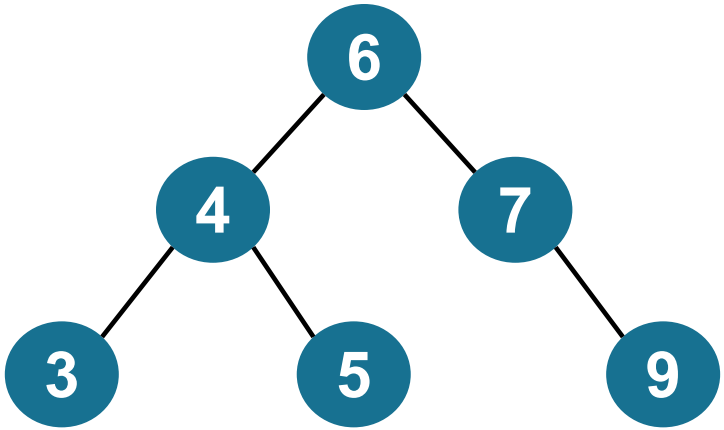
Laufzeit: O(h)



Open Distributed Sytems ODS

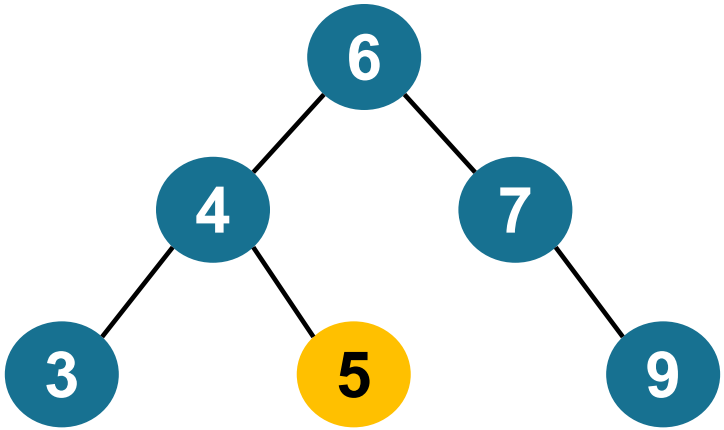
Binäre Suchbäume – Löschen

- 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



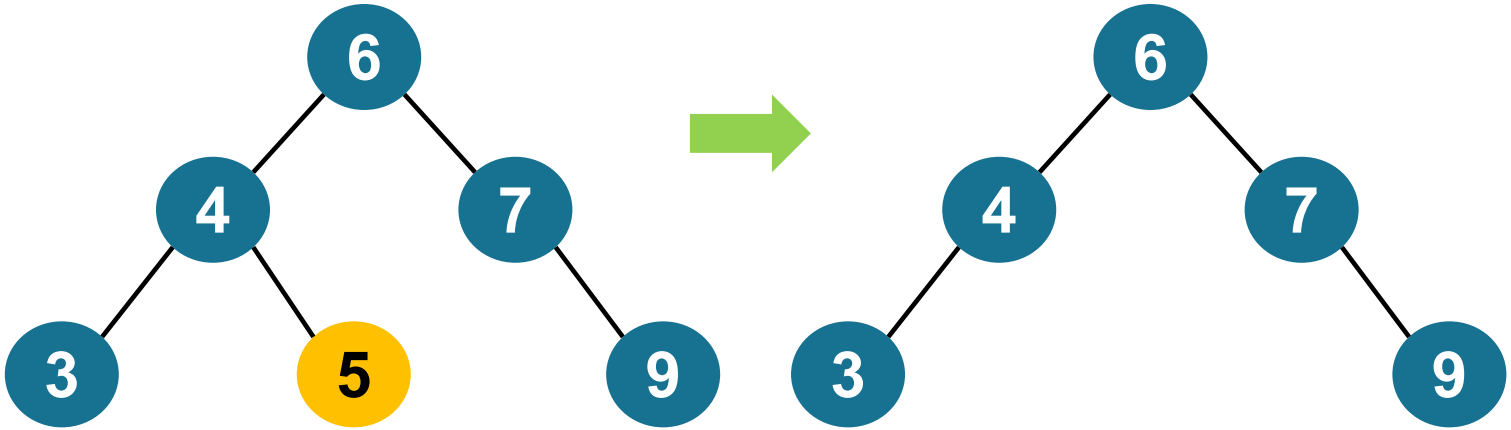
Binäre Suchbäume – Löschen

- Fall (a): Zu löschendes Element z hat keine Kinder



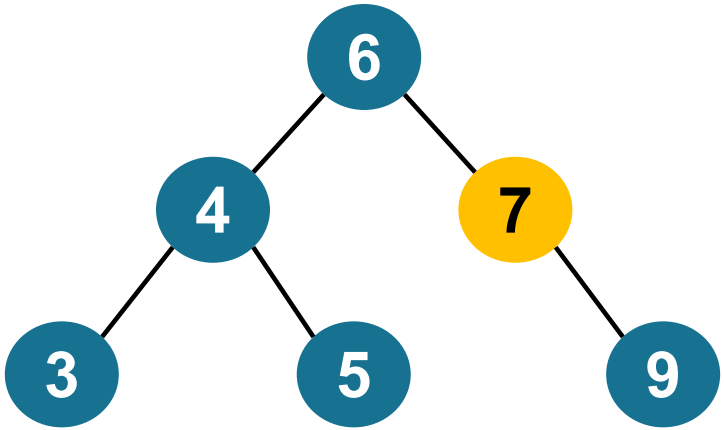
Binäre Suchbäume – Löschen

- Fall (a): Zu löschendes Element z hat keine Kinder
  - Entferne Element



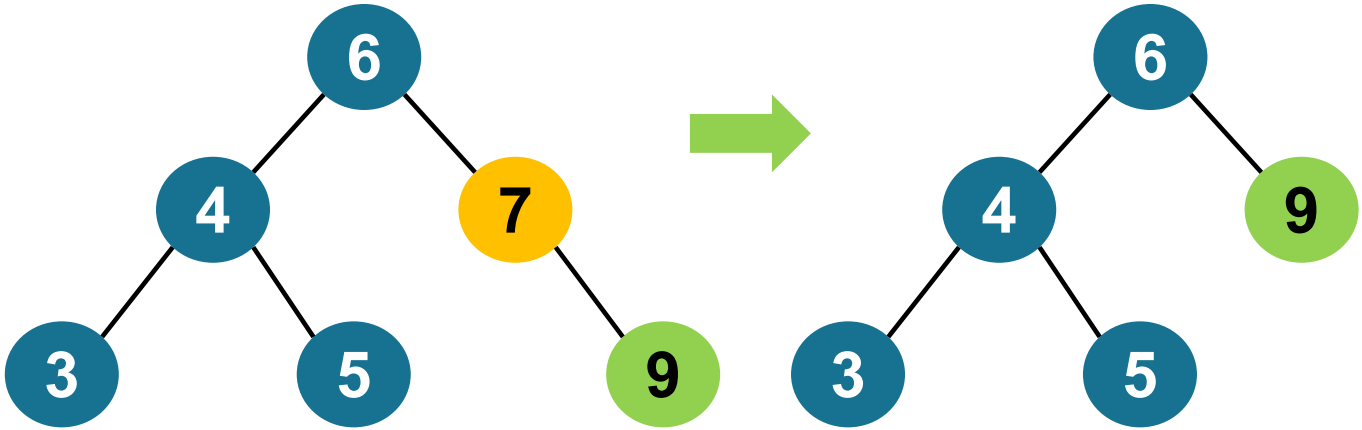
Binäre Suchbäume – Löschen

- Fall (b): Zu löschendes Element z hat 1 Kind



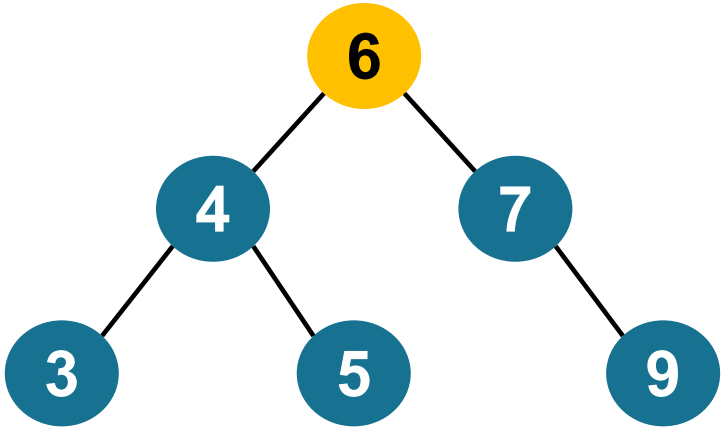
Binäre Suchbäume – Löschen

- Fall (b): Zu löschendes Element z hat 1 Kind
  - Ersetze Element durch das Kind



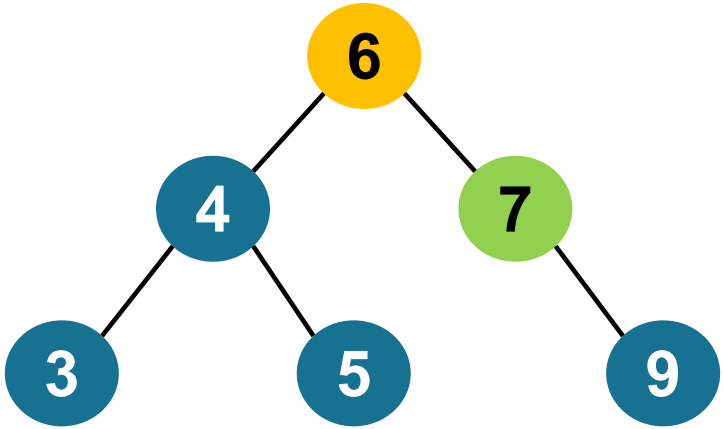
Binäre Suchbäume – Löschen

- Fall (c): Zu löschendes Element z hat 2 Kinder



Binäre Suchbäume – Löschen

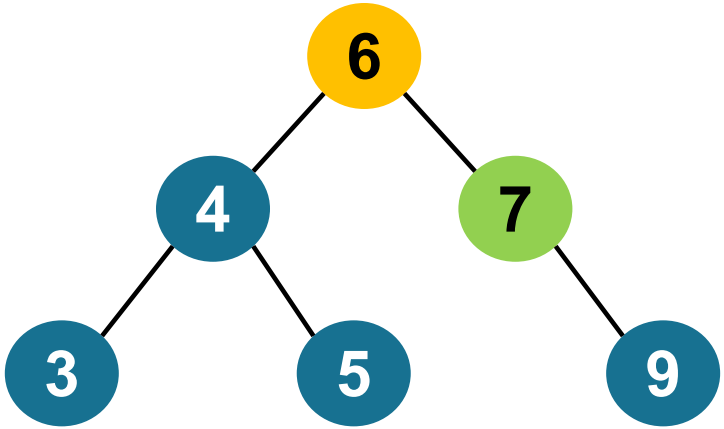
- Fall (c): Zu löschendes Element z hat 2 Kinder
  - Schritt 1: Bestimme Nachfolger von z



Binäre Suchbäume – Löschen

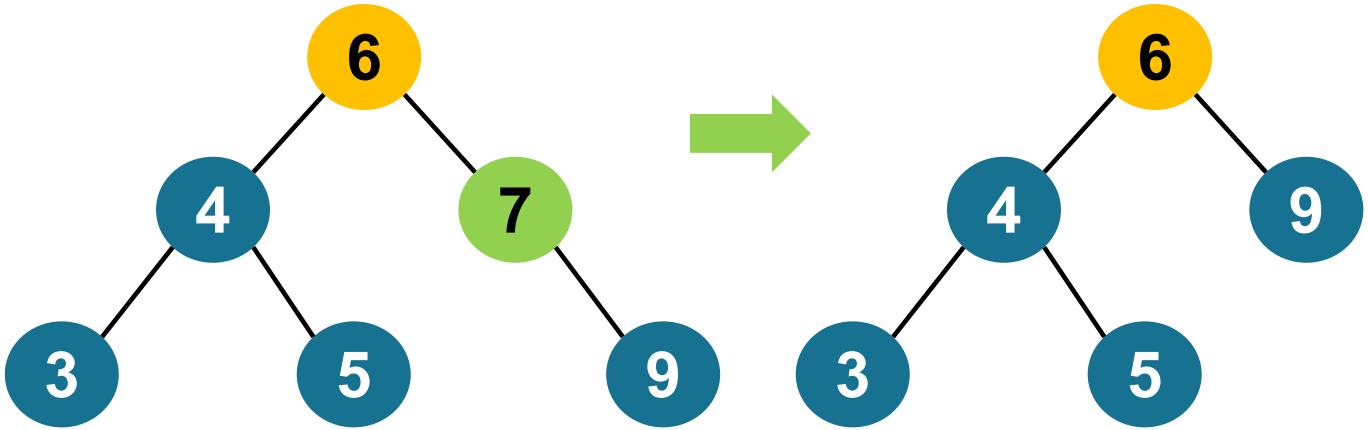
- Fall (c): Zu löschendes Element z hat 2 Kinder
  - Schritt 1: Bestimme Nachfolger von z

Nachfolger hat nur ein Kind.



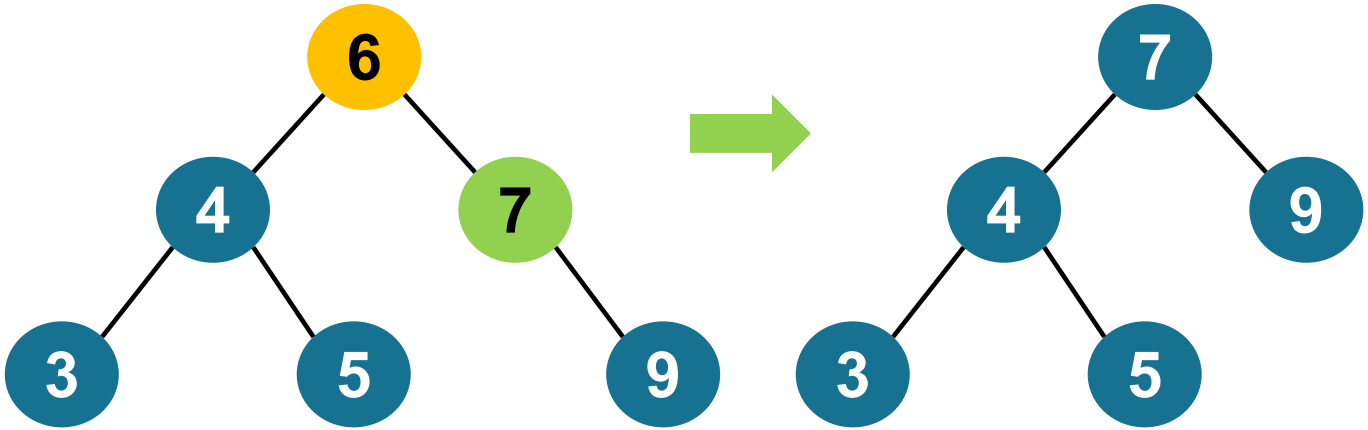
Binäre Suchbäume – Löschen

- Fall (c): Zu löschendes Element z hat 2 Kinder
  - Schritt 1: Bestimme Nachfolger von z
  - Schritt 2: Entferne Nachfolger von z



Binäre Suchbäume – Löschen

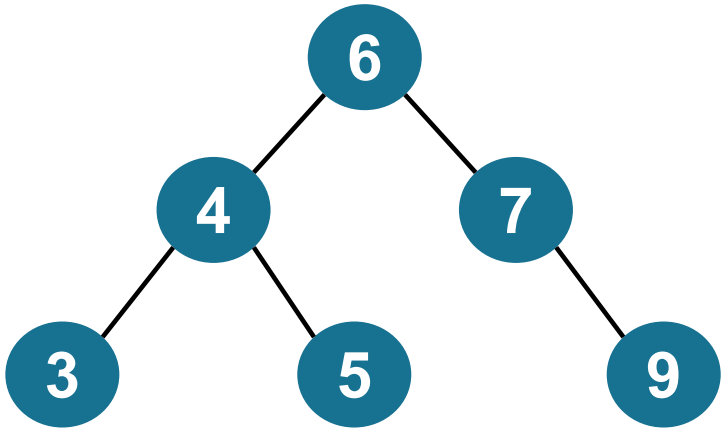
- Fall (c): Zu löschendes Element z hat 2 Kinder
  - Schritt 1: Bestimme Nachfolger von z
  - Schritt 2: Entferne Nachfolger von z
  - Schritt 3: Ersetze z durch Nachfolger



Binäre Suchbäume – Löschen

Löschen(T,z)

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

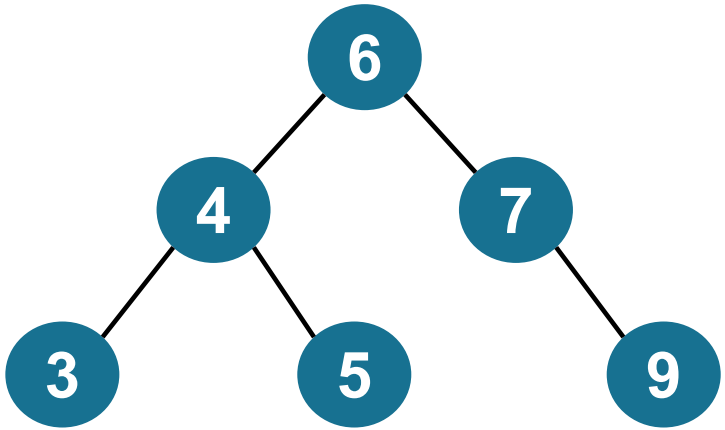


Binäre Suchbäume löschen

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. key(z) ← key(y)



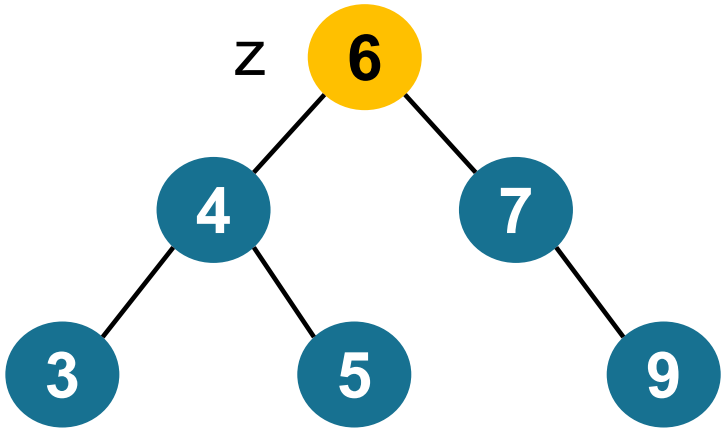
Binäre Suchbäume – Lösch

Löschen(T,z)

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

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

Löschen(6)



Open Distributed Sytems
ODS

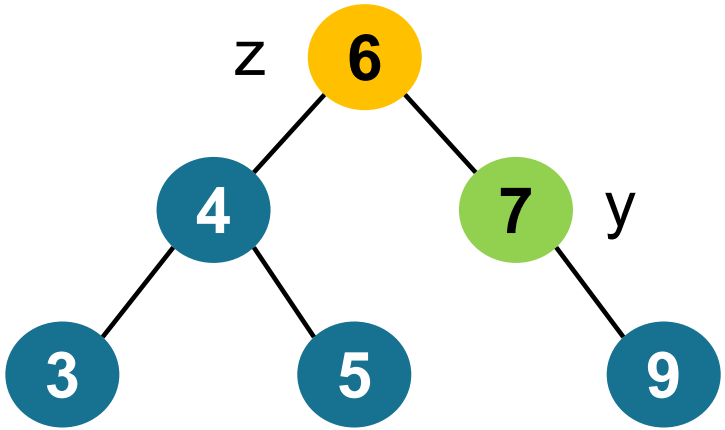
Binäre Suchbäume – Lösch

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

Löschen(T,z)

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

Löschen(6)



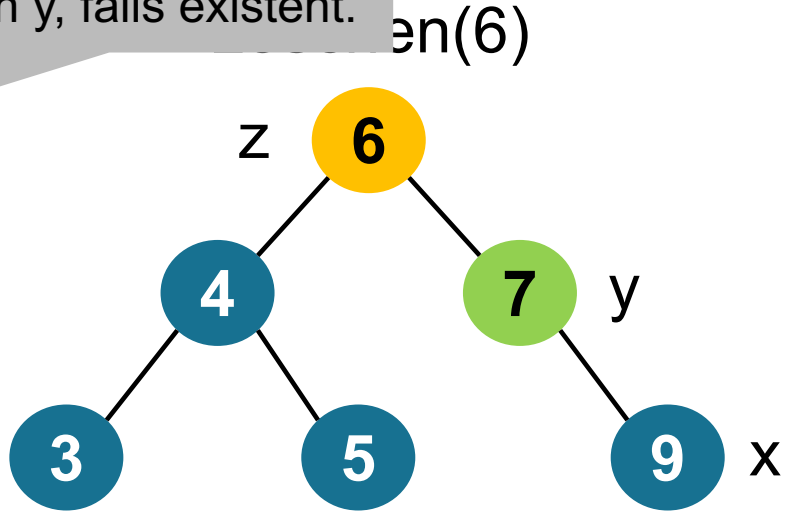
Open Distributed Sytems
ODS

Binäre Suchbäume – Löschen

Löschen(T,z)

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

Bestimme das Kind von y, falls existent.

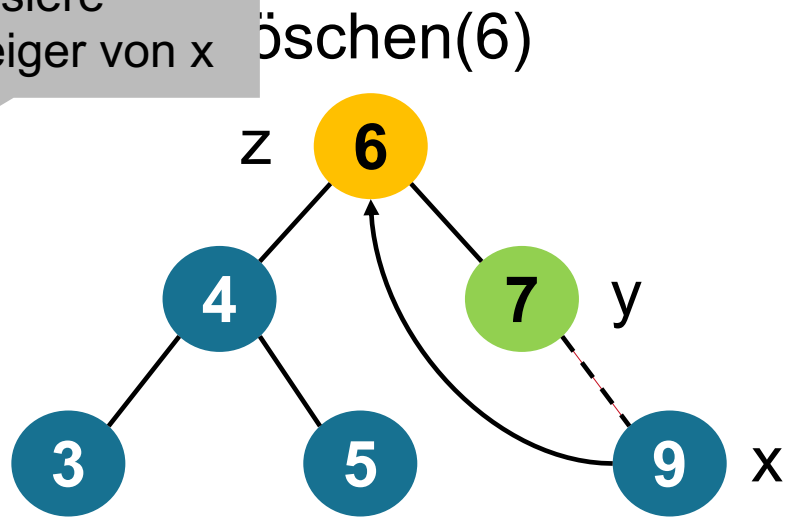


Binäre Suchbäume – Löschen

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. key(z) ← key(y)

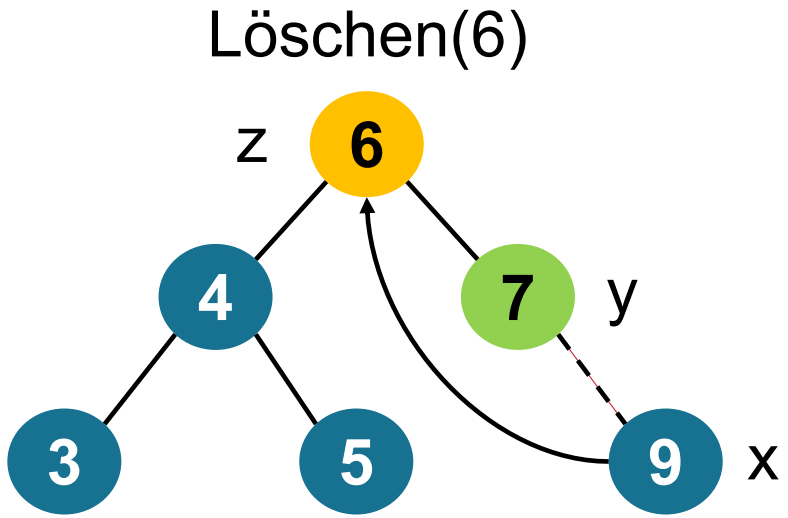
Aktualisiere Elterzeiger von x



Binäre Suchbäume – Löschen

Löschen(T,z)

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

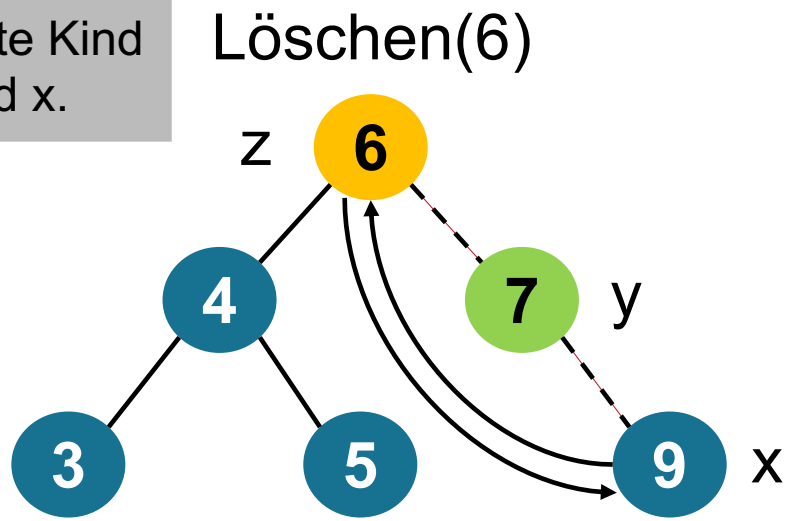


Binäre Suchbäume – Löschen

Löschen(T,z)

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

Das rechte Kind von z wird x.

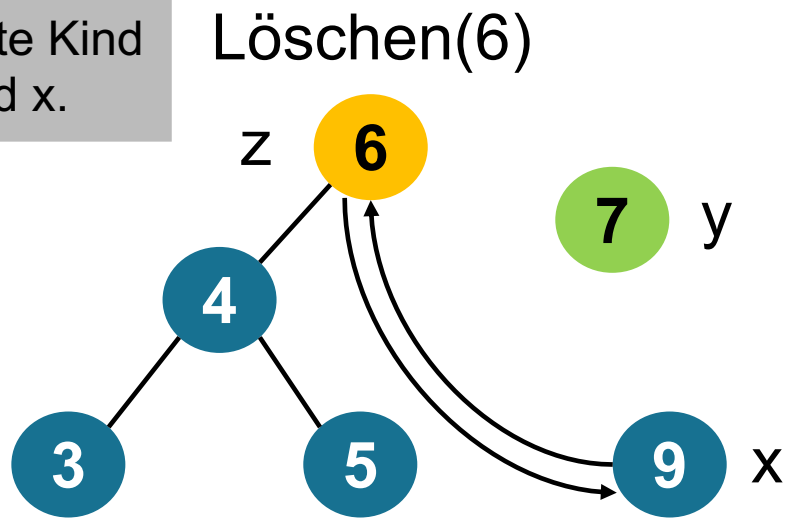


Binäre Suchbäume – Löschen

Löschen(T,z)

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

Das rechte Kind von z wird x.

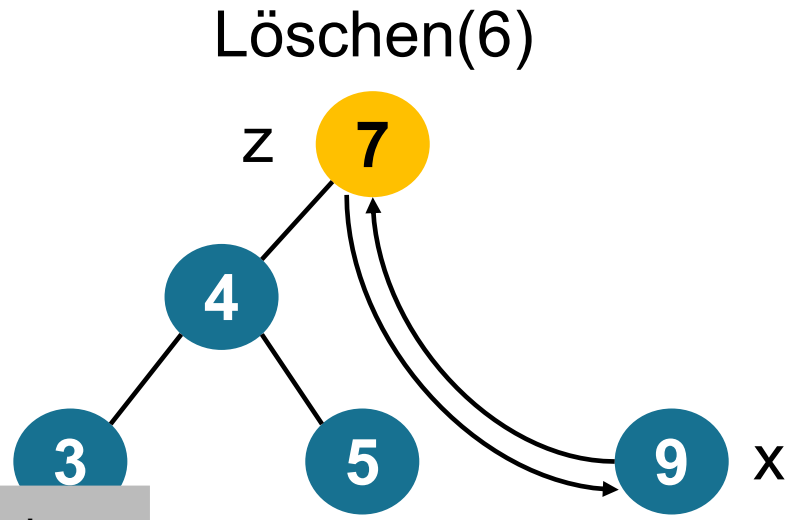


Binäre Suchbäume – Löschen

Löschen(T,z)

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

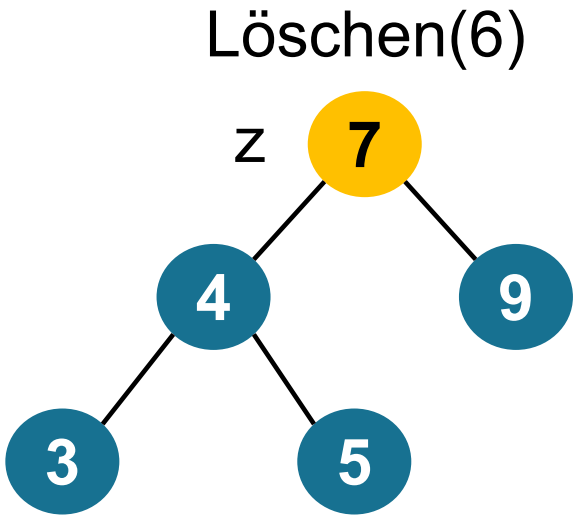
Umkopieren des Inhalts von y nach z



Binäre Suchbäume – Löschen

Löschen(T,z)

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



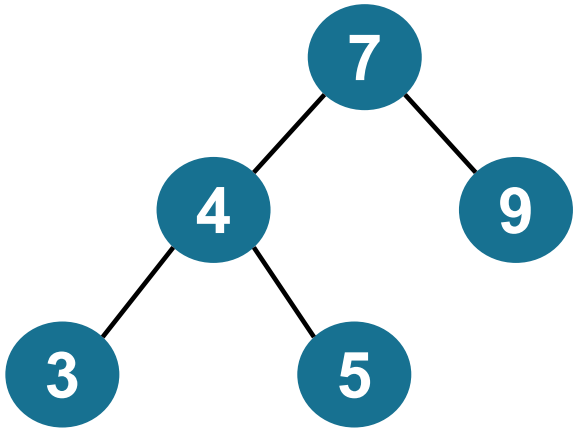
Binäre Suchbäume – Löschen

Technische Universität Berlin

Löschen(T,z)

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

Laufzeit: O(h)



Open Distributed Sytems ODS

Binäre Suchbäume

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

- Bester Fall
  - Beide Teilbäume sind vorhanden, d.h. beide Teilbäume sind gleich groß => der Baum ist vollständig balanciert
  - D.h. die Höhe ist O(log n)

- Schlechtester Fall
  - Ein Teilbaum ist immer leer, d.h. der Baum degeneriert zu einer Liste
  - D.h. die Höhe ist O(n)

Binäre Suchbäume

- Erhoffter/erwünschter Fall
  - Beide Teilbäume sind fast gleich groß => der Baum ist halbwegs balanciert
  - D.h. die Höhe ist immer noch O(log n)

- Frage
  - Wie kann man eine solche „kleine“ Höhe unter Einfügen und Löschen garantieren?
  - Dafür gibt es selbstbalancierende Suchbäume, z.B. AVL oder Rot/Schwarzbäume – dazu mehr später

Ausblick

VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine

VL 1 „Algorithmen, Pseudocode, Sortieren I“: Insertion Sort

VL 2 „Algorithmen, Pseudocode, Sortieren II“: Selection Sort, Bubble Sort, Count Sort

VL 3 „Laufzeit und Speicherplatz“: Laufzeitanalyse der vorgestellten Sortierverfahren

VL 4 „Einfache Datenstrukturen“: Arrays, verkettete Listen, Structs in C, Stack, Queue

VL 5 „Bäume“: Binärbäume, Baumtraversierung, Laufzeitanalyse Baumoperationen

VL 6 „Dateien in C“: Dateien, Dateisysteme, Verzeichnisse, Dateiverwaltung mit C

VL 7 „Teile und Herrsche I“: Einführung der algorithmischen Methode, Merge Sort

VL 8 „Korrektheitsbeweise“: Rechnermodel, Beispielbeweise

VL 9 „Prioritätenschlangen/Halden/Heaps “: Heap Sort, Binärer Heap, Heap Operationen

VL 10 „Fortgeschrittene Sortierverfahren“: Quick Sort, Radix Sort

VL 11 „AVL Bäume“: Definition, Baumoperationen, Traversierung

VL 12 „Teile und Herrsche II“: Generalisierung des algorithmischen Prinzips, Mastertheorem

VL 13 „Q & A“: Offene Vorlesung/Wiederholung