technische universität
dortmund

LS 2 /
Informatik

Datenstrukturen, Algorithmen und Programmierung 2 (DAP2)

Datenstrukturen

Ein grundlegendes Datenbank-Problem

- Speicherung von Datensätzen

Beispiel

- Kundendaten (Name, Adresse, Wohnort, Kundennummer, offene Rechnungen, offene Bestellungen,…)

Anforderungen

- Schneller Zugriff
- Einfügen neuer Datensätze
- Löschen bestehender Datensätze

Datenstrukturen

Zugriff auf Daten

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

Beispiel

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

Datenstrukturen

Problem:

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

Operationen:

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

Datenstrukturen

Drei grundlegende Datenstrukturen

- Feld
- sortiertes Feld
- doppelt verkettete Liste

Diskussion

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

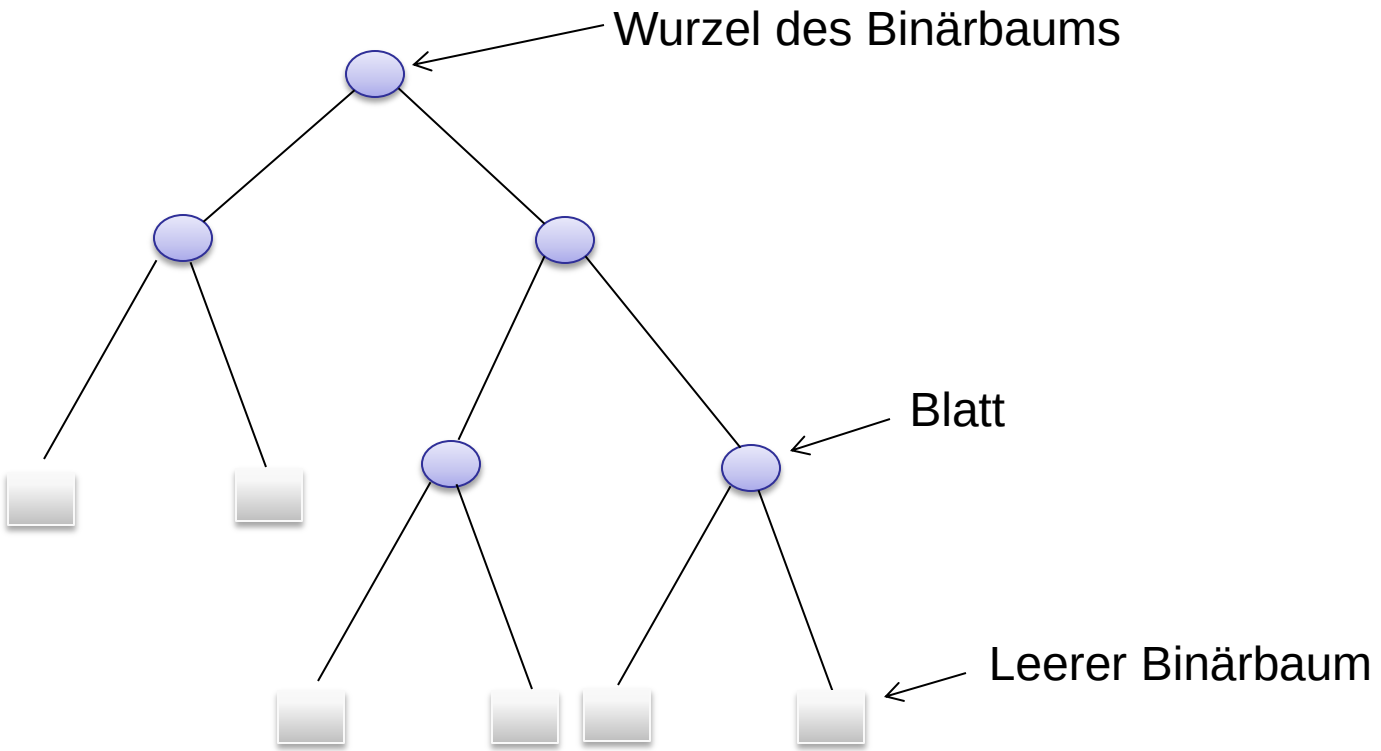
Datenstrukturen

Definition (Binärbaum)

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

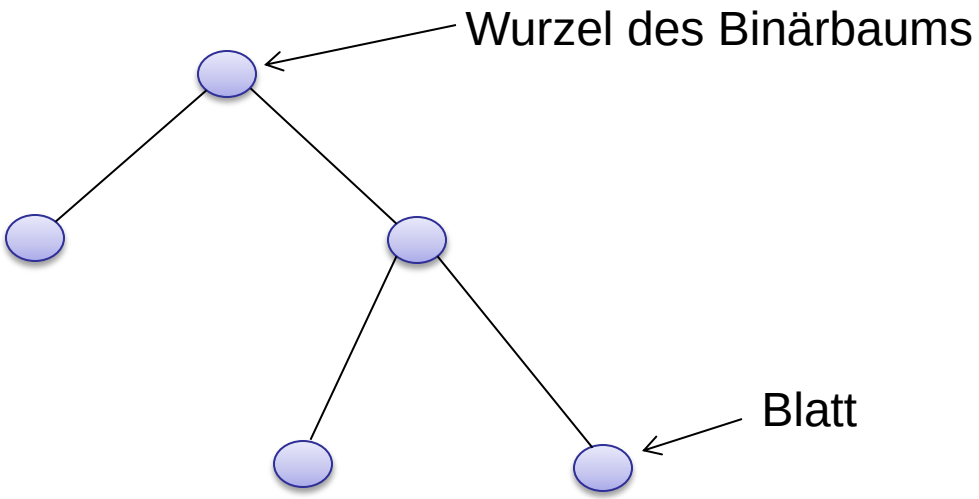
Datenstrukturen

Darstellung von Binärbäumen



Datenstrukturen

Darstellung von Binärbäumen

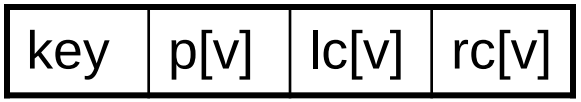


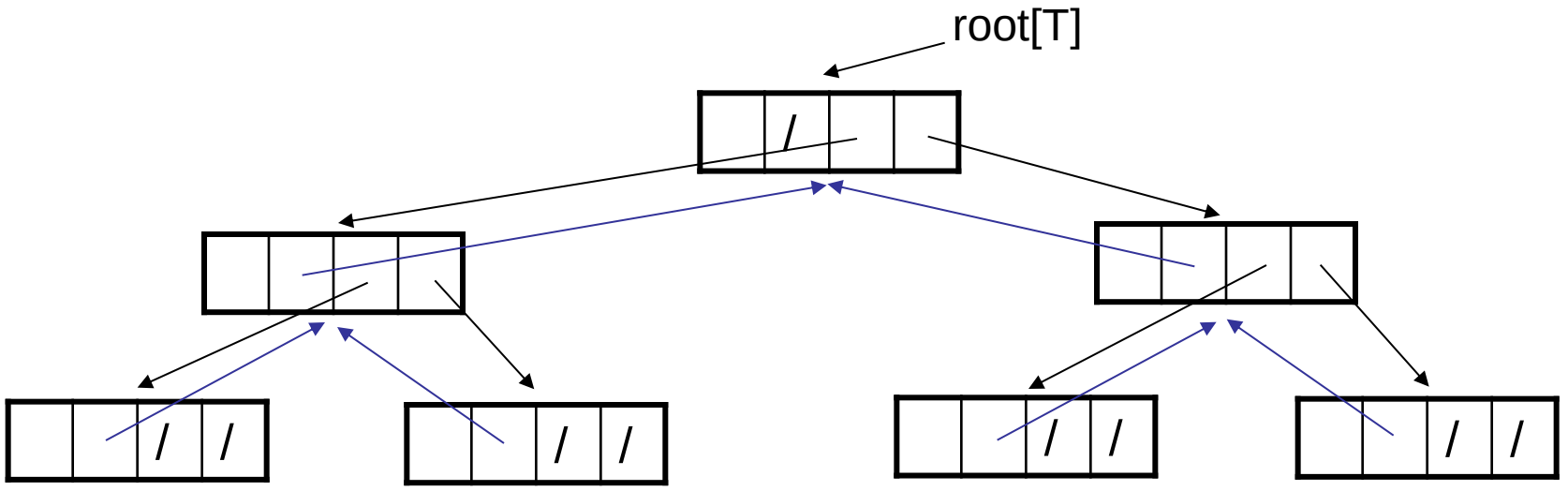
- Häufig lässt man die leeren Bäume in der Darstellung eines Binärbaums weg

Datenstrukturen

Binärbäume(Darstellung im Rechner)

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

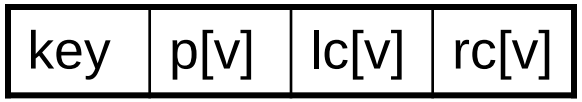


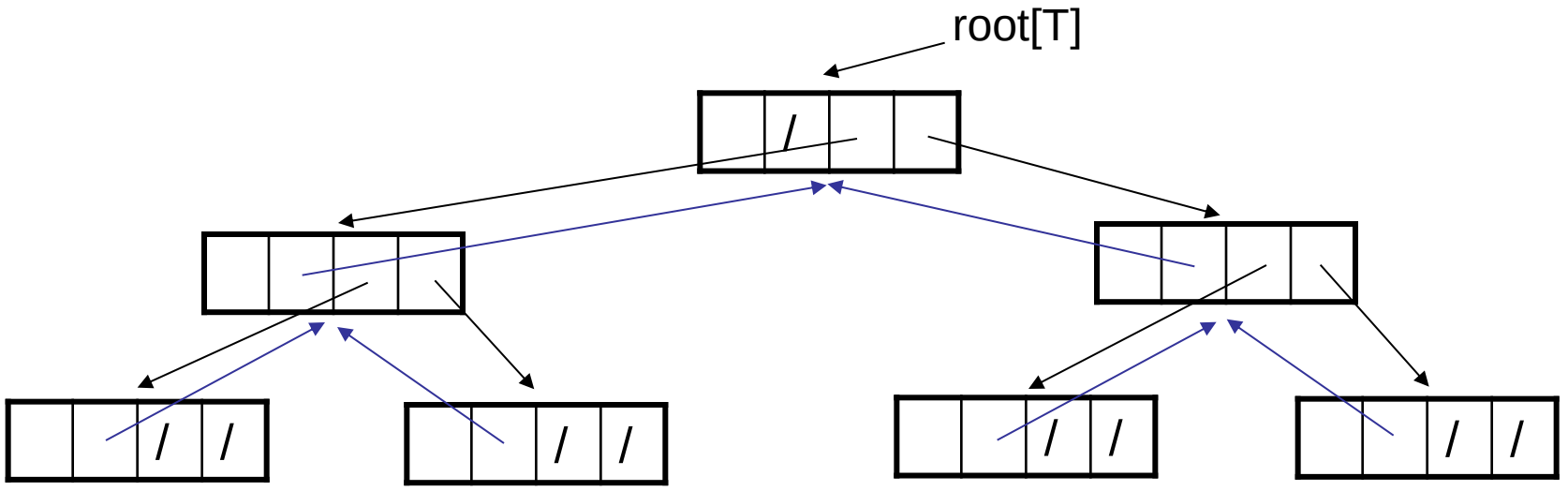


Datenstrukturen

Binärbäume(Darstellung im Rechner)

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

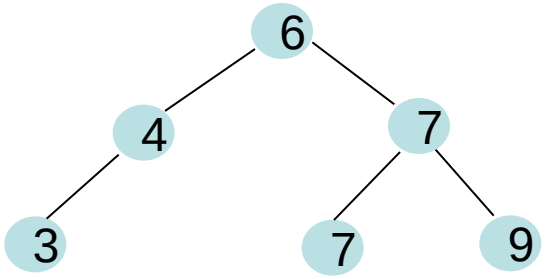




Datenstrukturen

Binäre Suchbäume

- Verwende Binärbaum
- Speichere Schlüssel „geordnet“



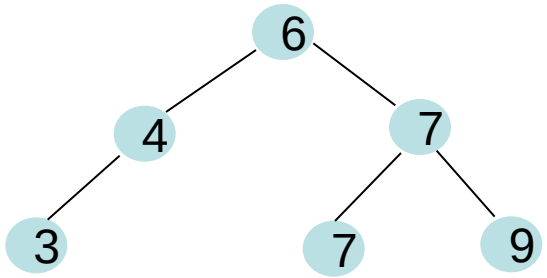
Binäre Suchbaumeigenschaft:

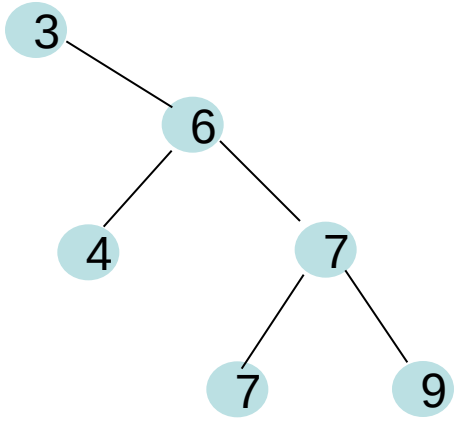
- Sei x Knoten im binären Suchbaum
- Ist y Knoten im linken Unterbaum von x, dann gilt key[y]≤key[x]
- Ist y Knoten im rechten Unterbaum von x, dann gilt key[y]>key[x]

Datenstrukturen

Unterschiedliche Suchbäume

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

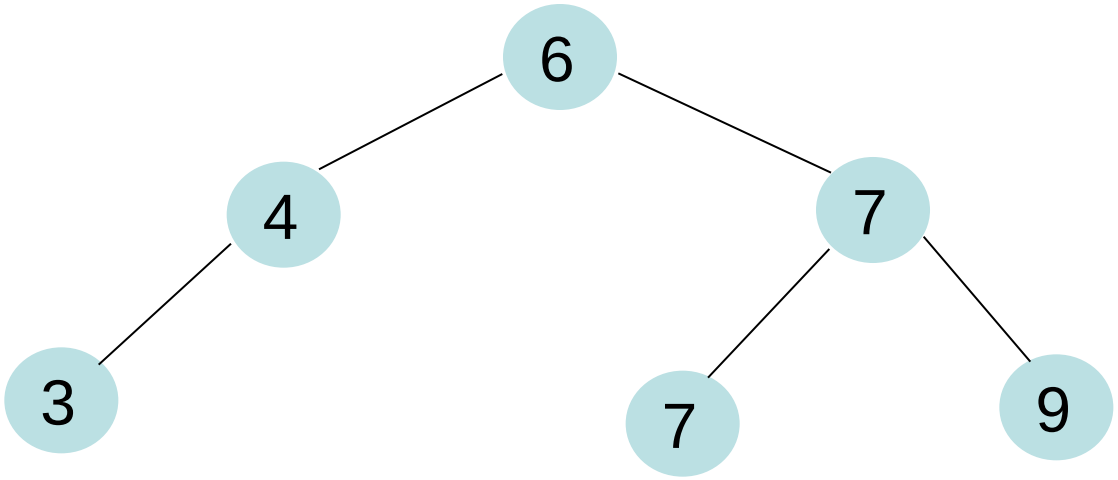




Datenstrukturen

Ausgabe aller Schlüssel

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

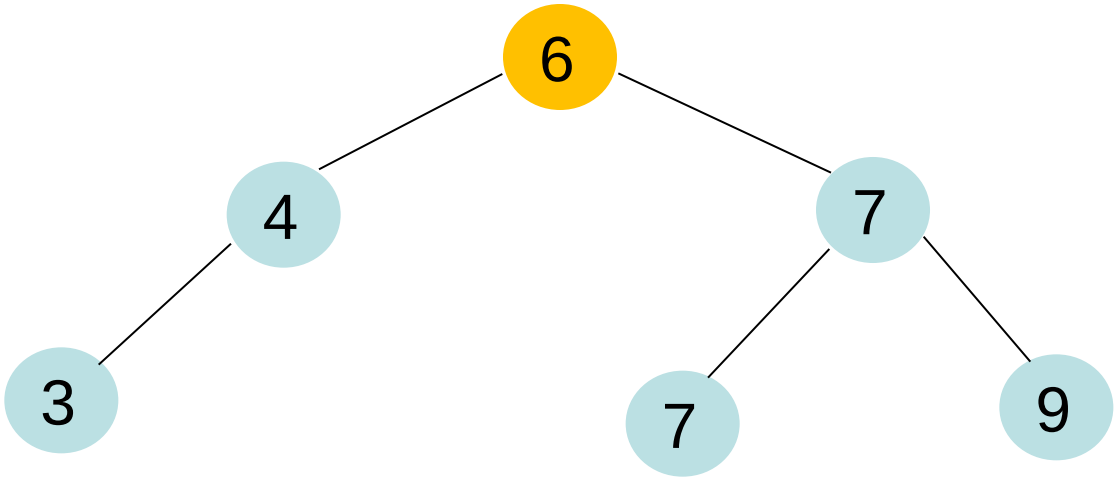


Datenstrukturen

Inorder-Tree-Walk(x)

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

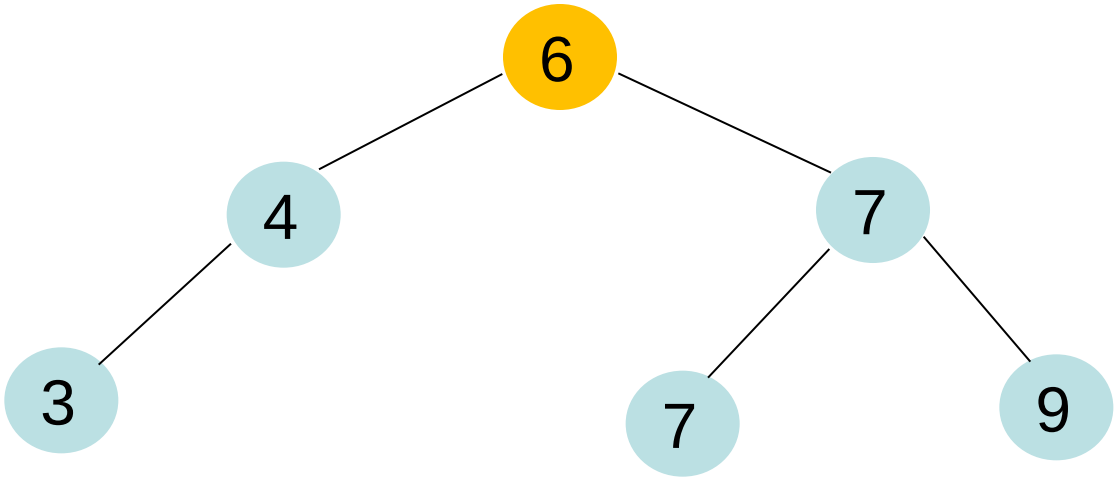
Aufruf über
Inorder-Tree-Walk(root[T])



Datenstrukturen

Inorder-Tree-Walk(x)

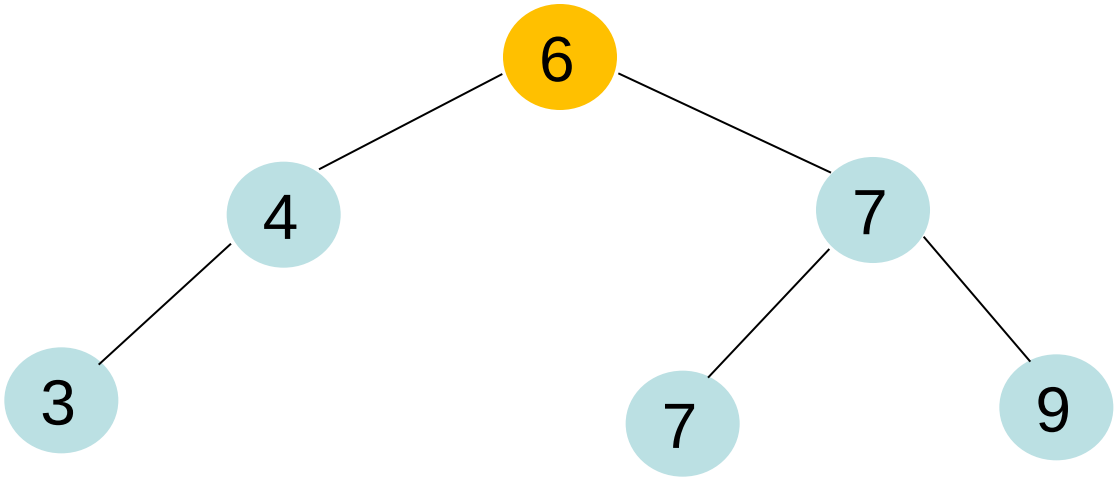
1. **if x≠nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])



Datenstrukturen

Inorder-Tree-Walk(x)

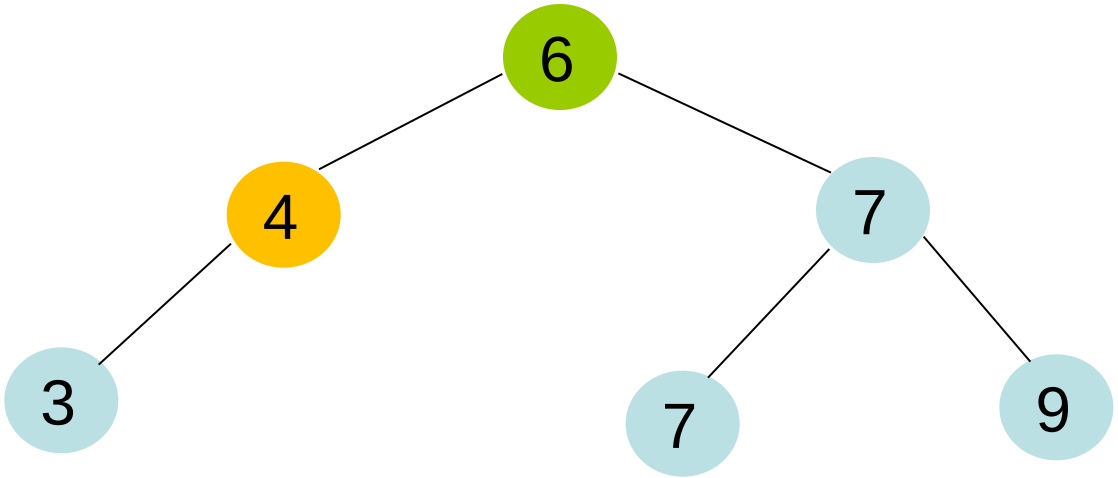
1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])



Datenstrukturen

Inorder-Tree-Walk(x)

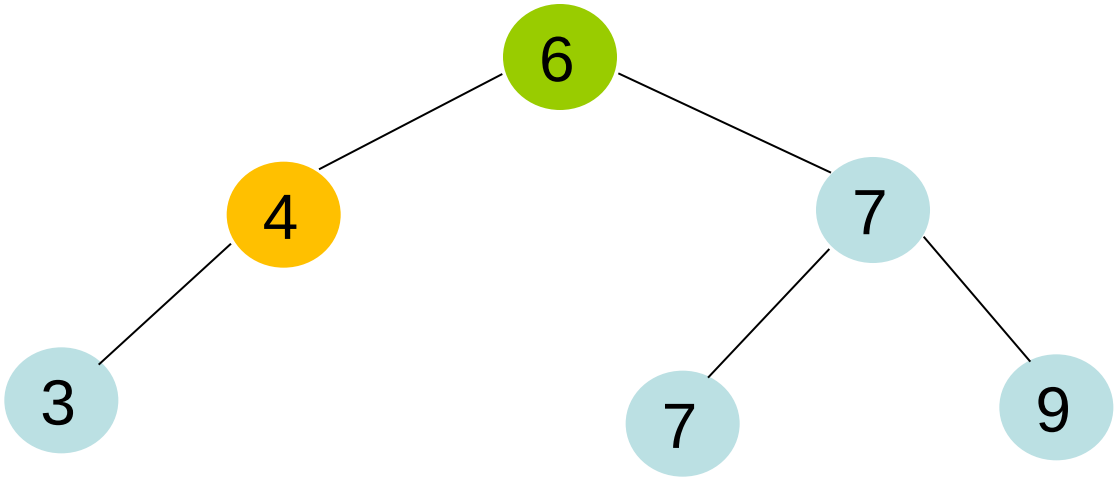
1. **if x≠nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])



Datenstrukturen

Inorder-Tree-Walk(x)

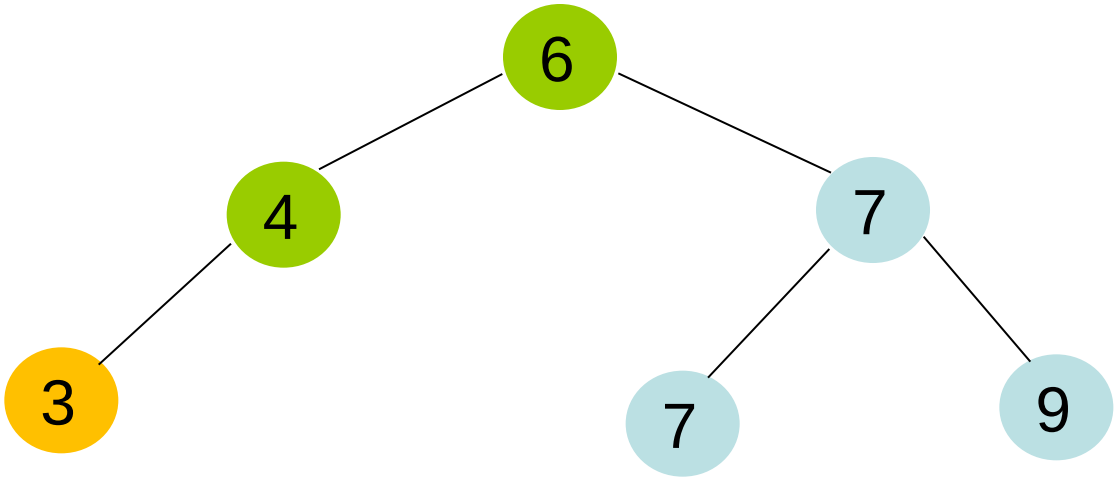
1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])



Datenstrukturen

Inorder-Tree-Walk(x)

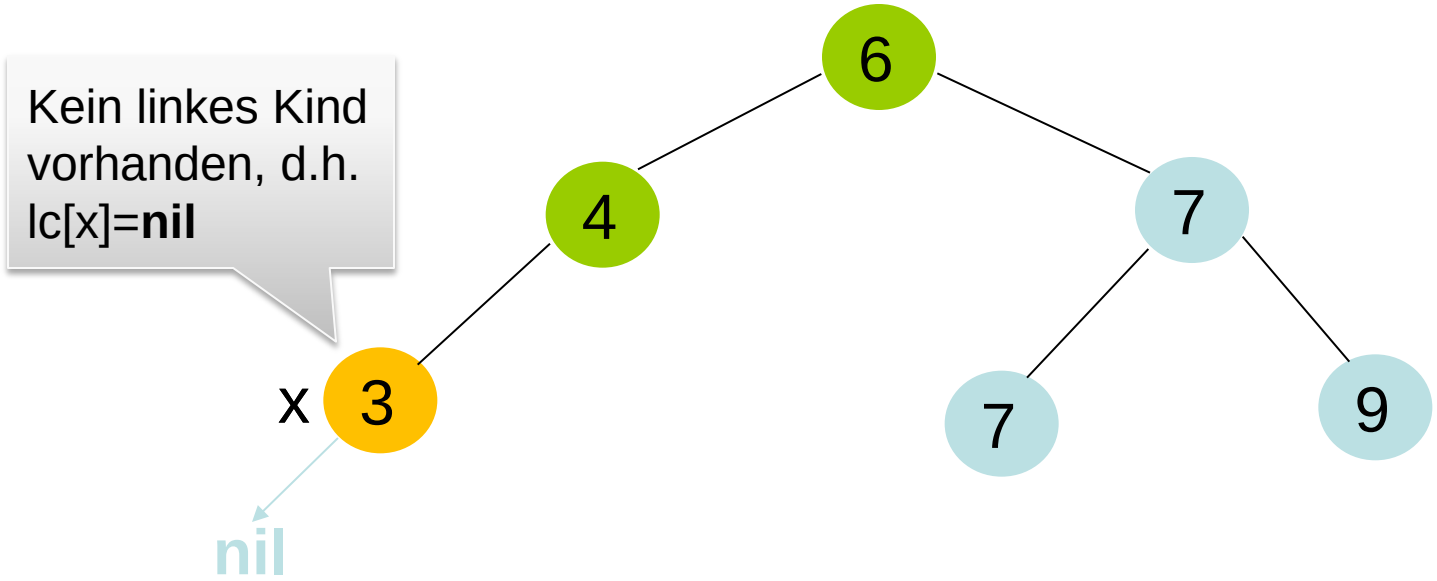
1. **if x≠nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])



Datenstrukturen

Inorder-Tree-Walk(x)

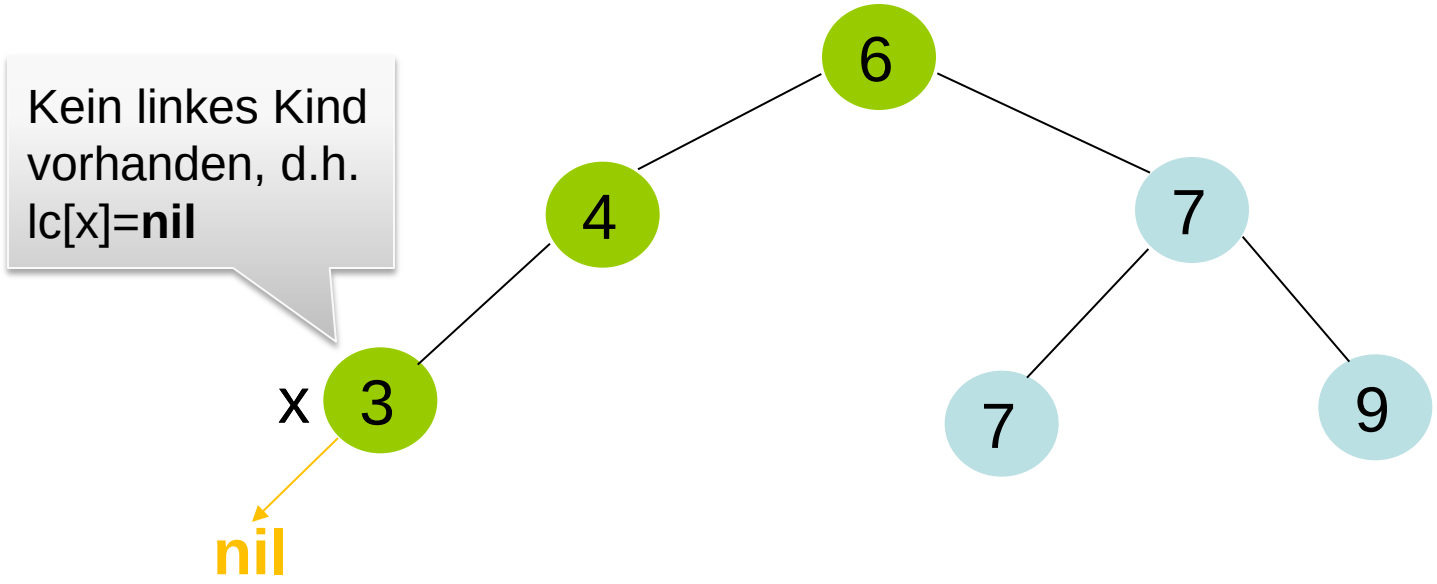
1. **if** x≠**nil then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])



Datenstrukturen

Inorder-Tree-Walk(x)

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

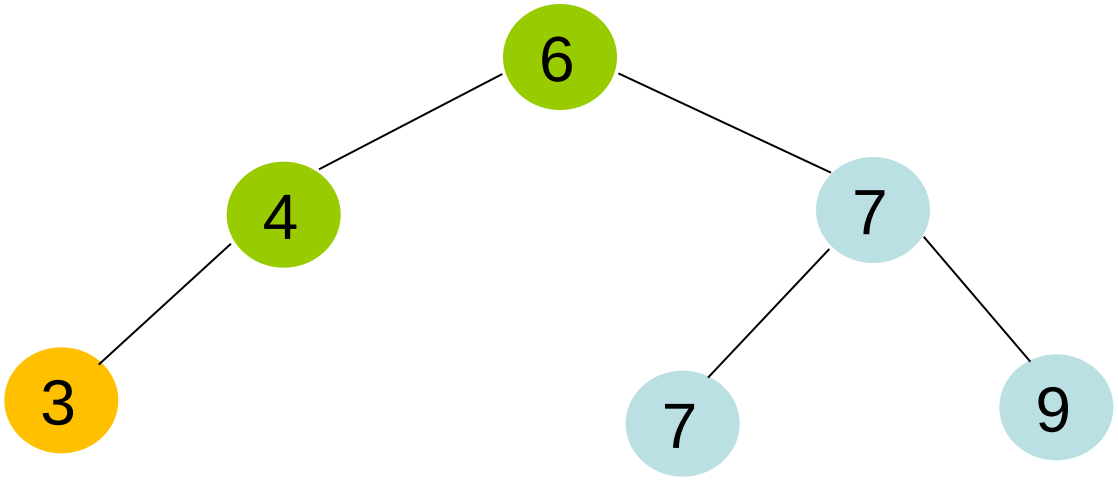


Datenstrukturen

Inorder-Tree-Walk(x)

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

Ausgabe:
3



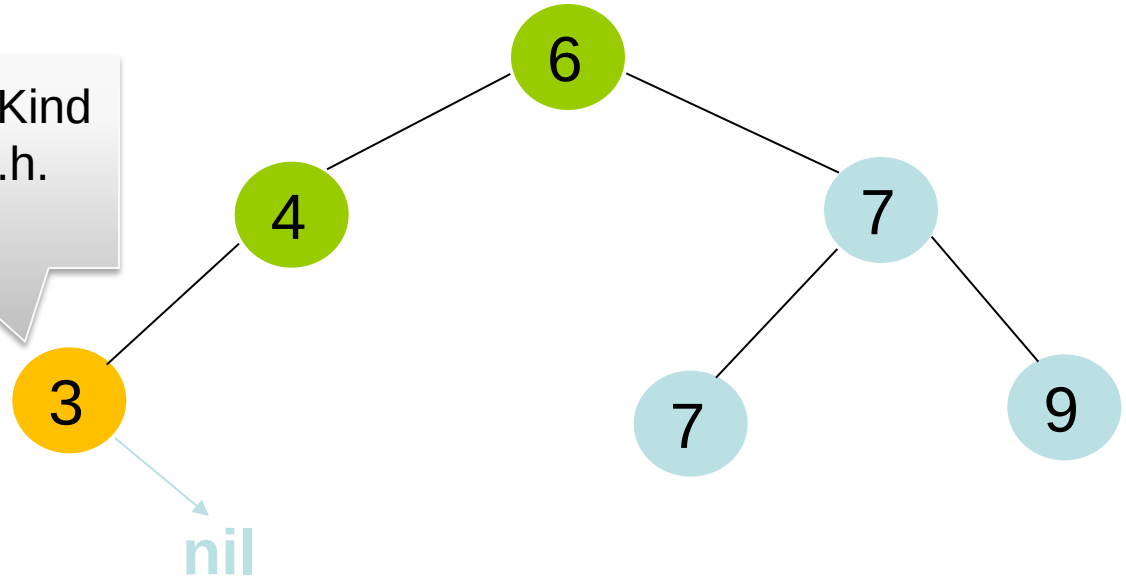
Datenstrukturen

Inorder-Tree-Walk(x)

1. **if** x≠**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**



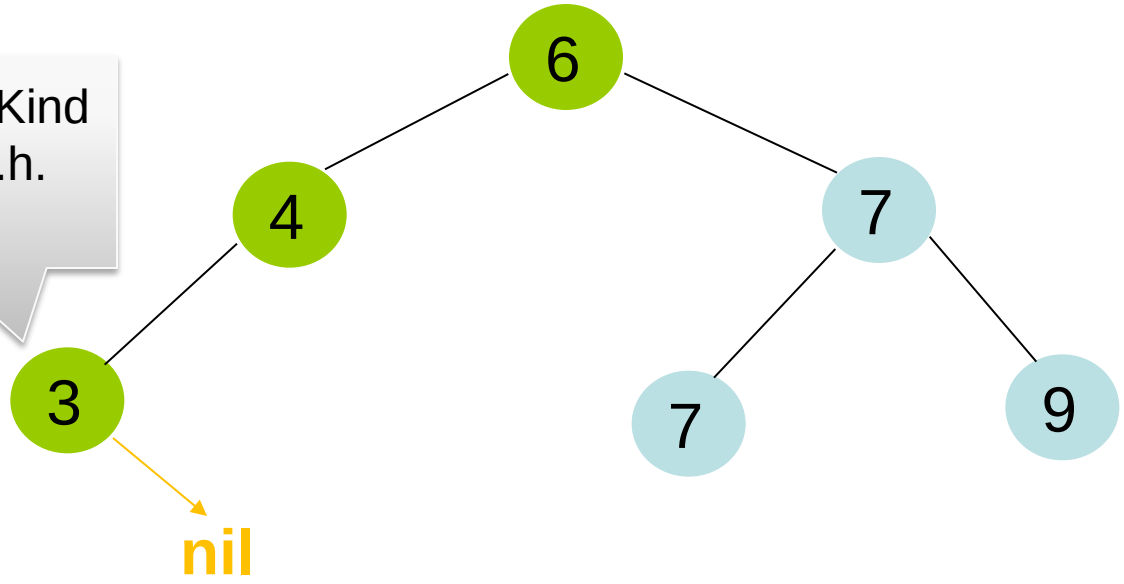
Datenstrukturen

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**

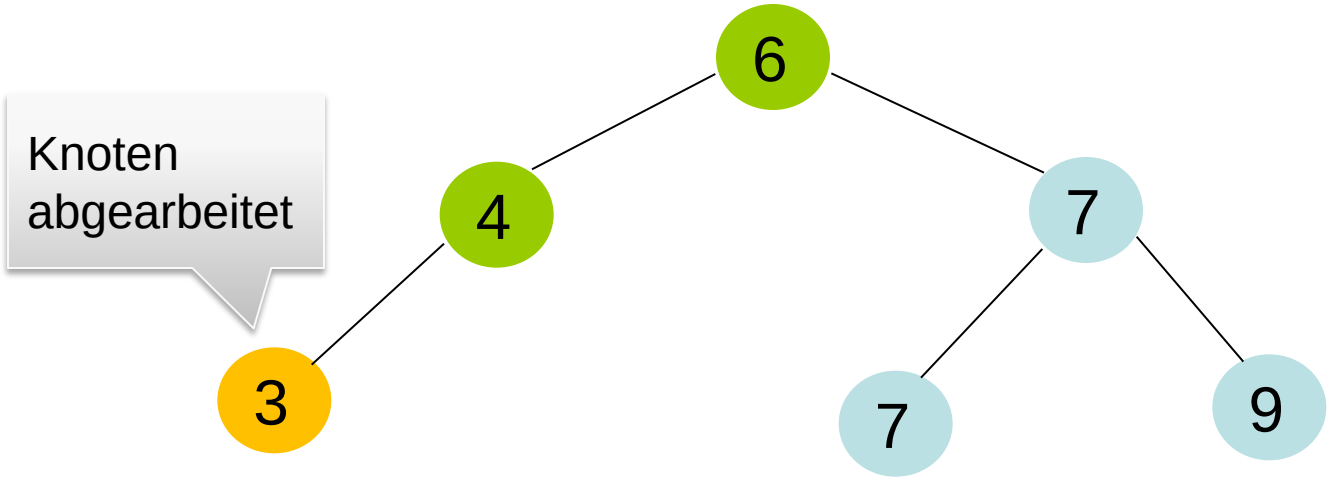


Datenstrukturen

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

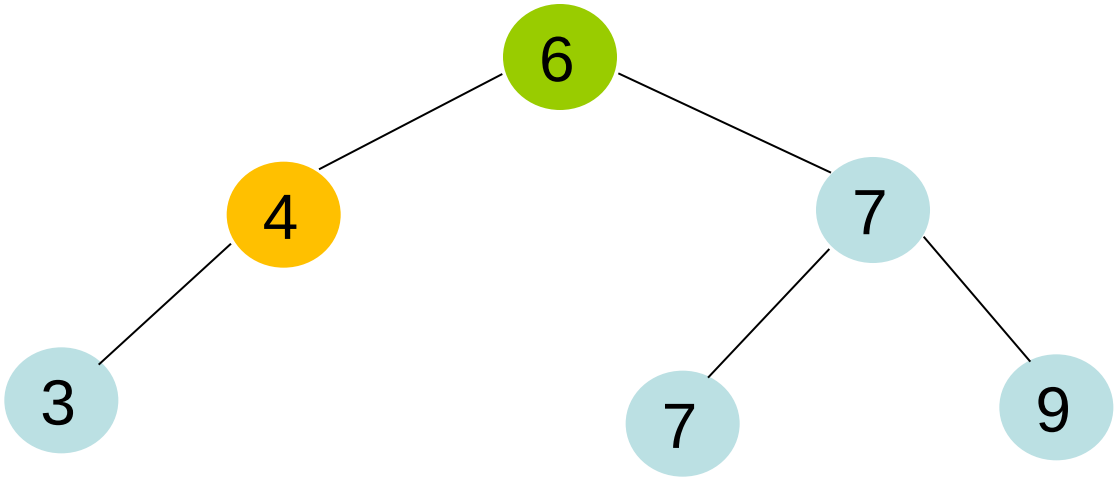


Datenstrukturen

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

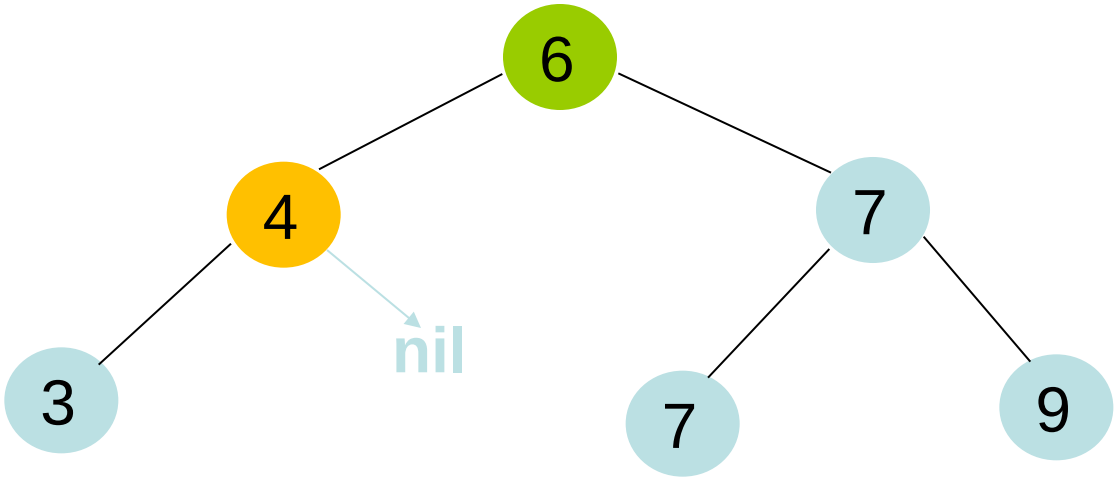


Datenstrukturen

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

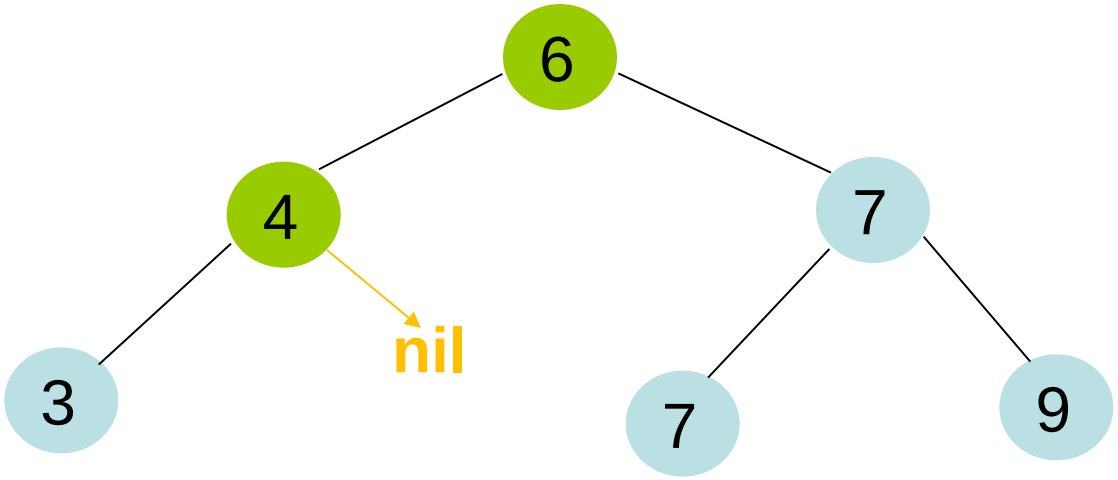


Datenstrukturen

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

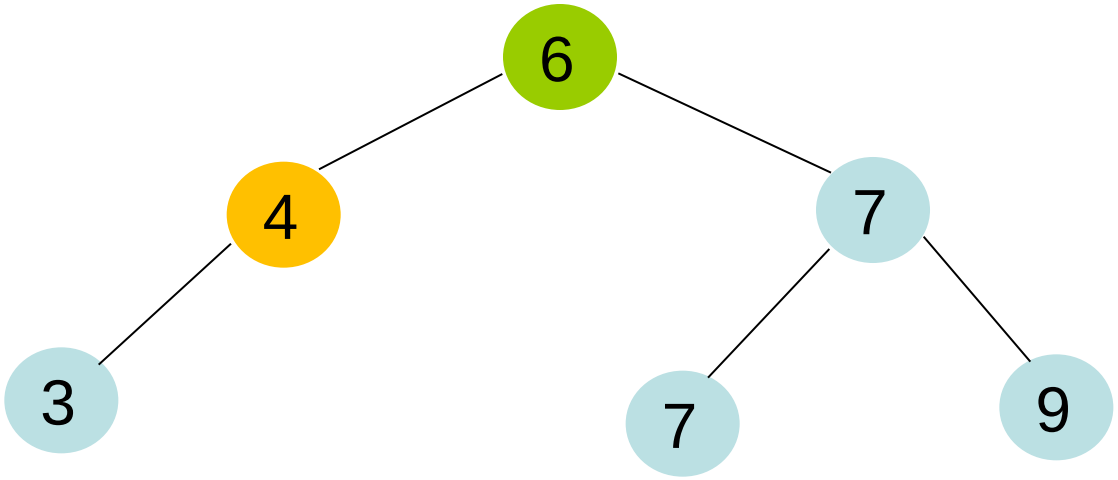


Datenstrukturen

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

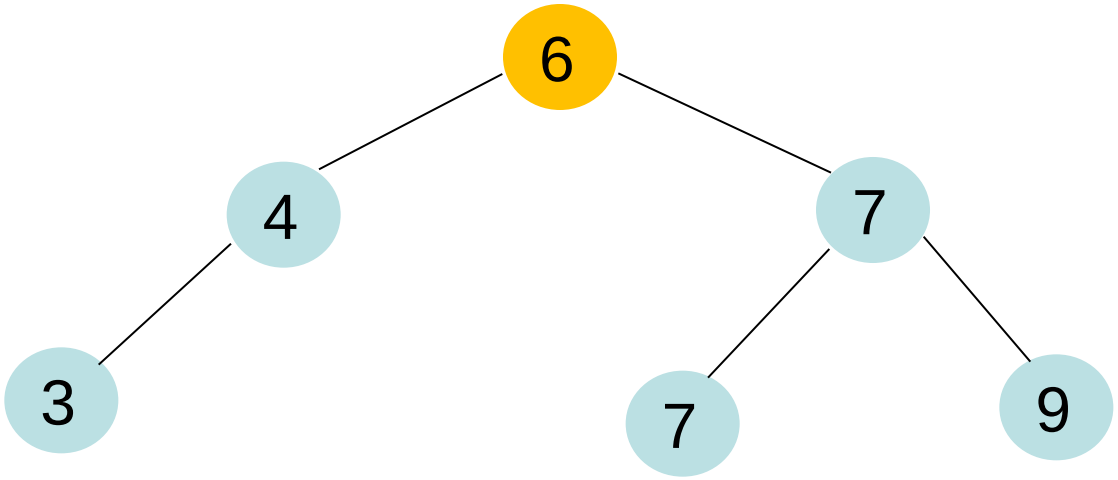


Datenstrukturen

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

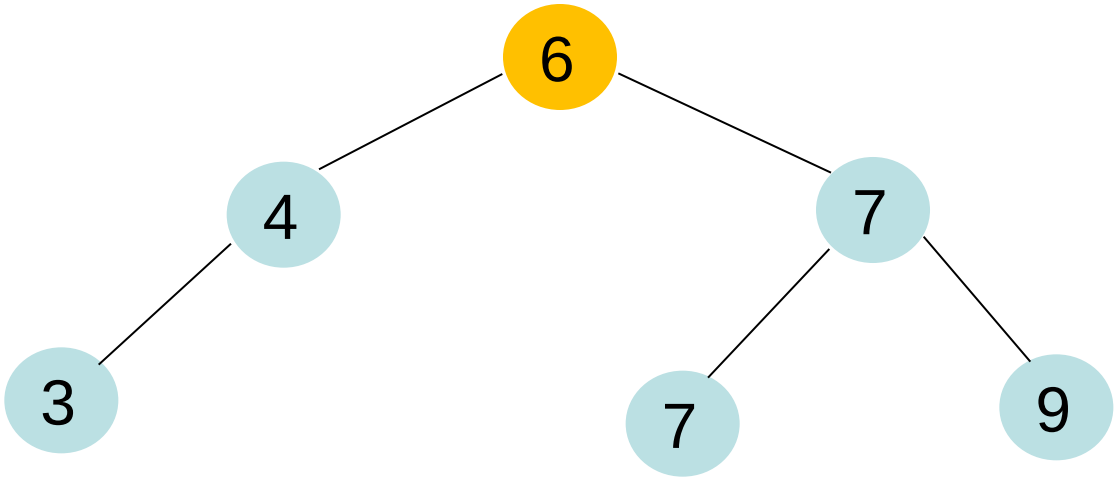


Datenstrukturen

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

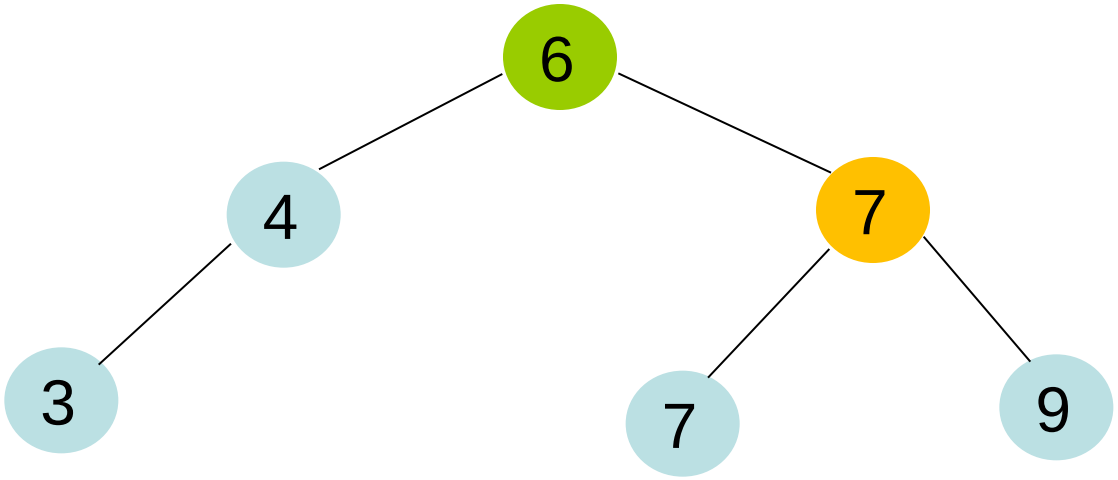


Datenstrukturen

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

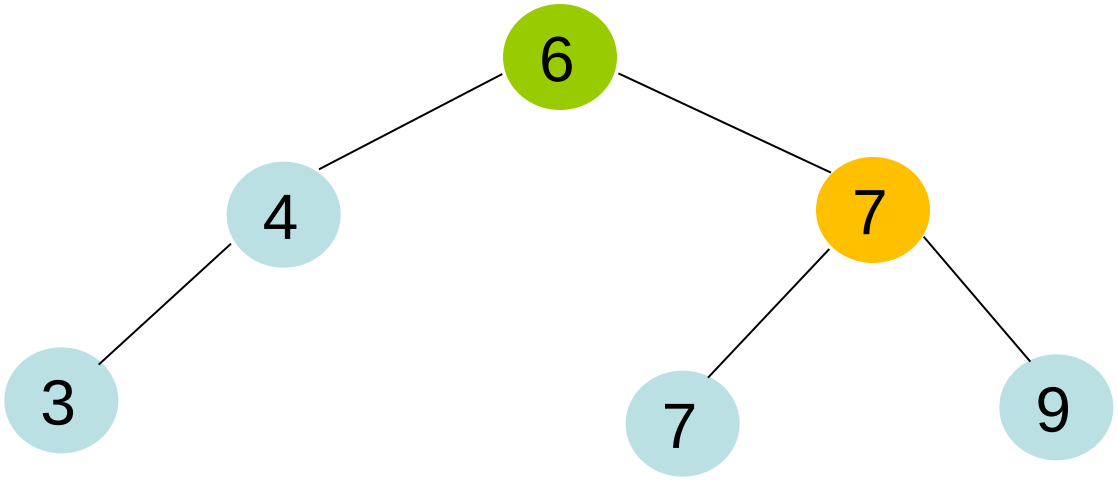


Datenstrukturen

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

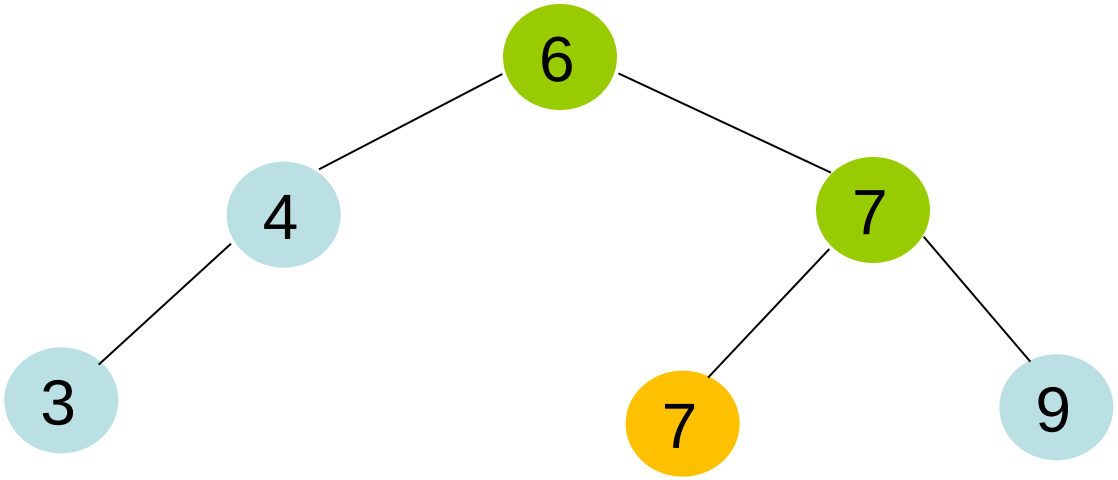


Datenstrukturen

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

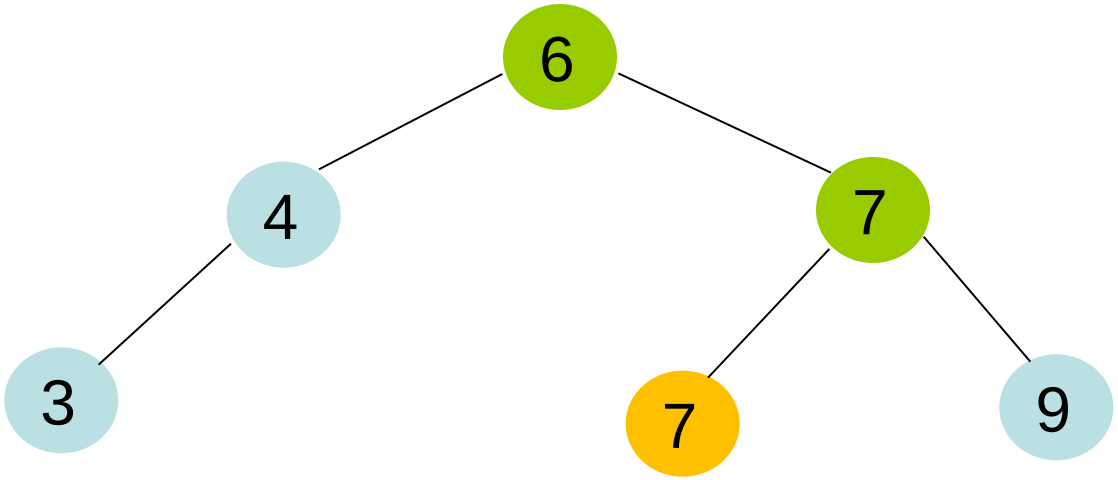


Datenstrukturen

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

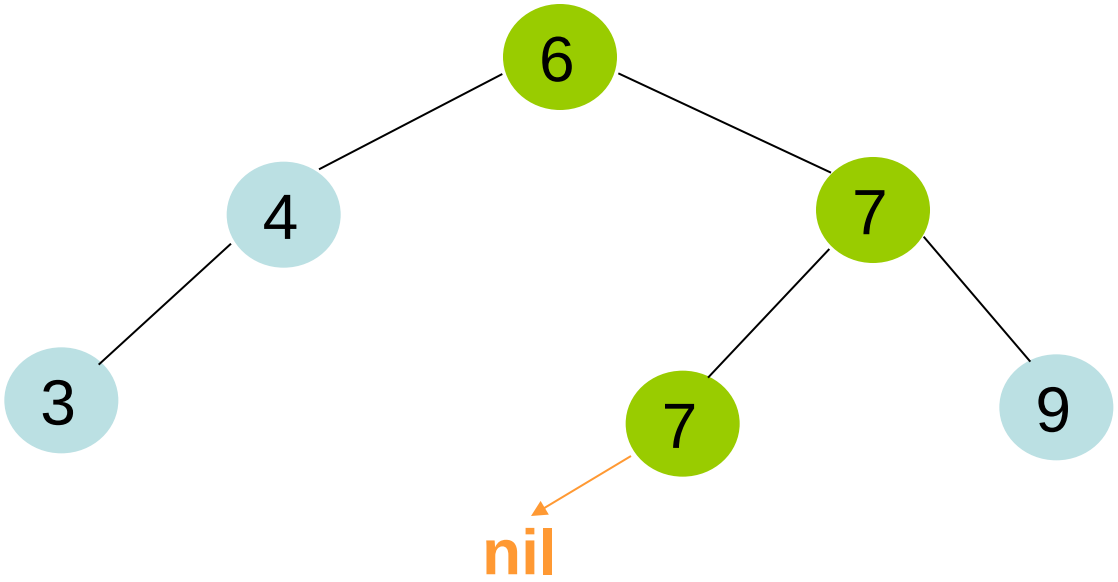


Datenstrukturen

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

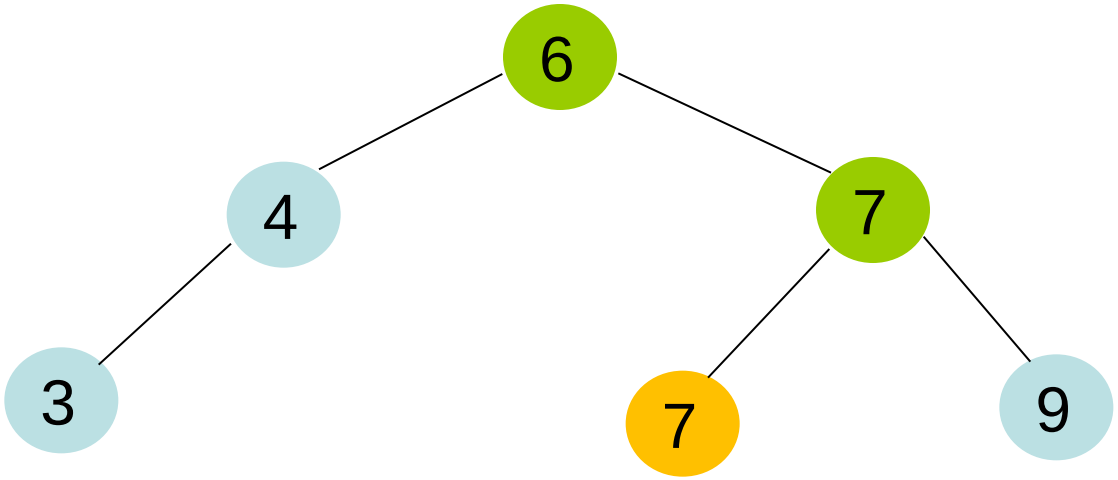


Datenstrukturen

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

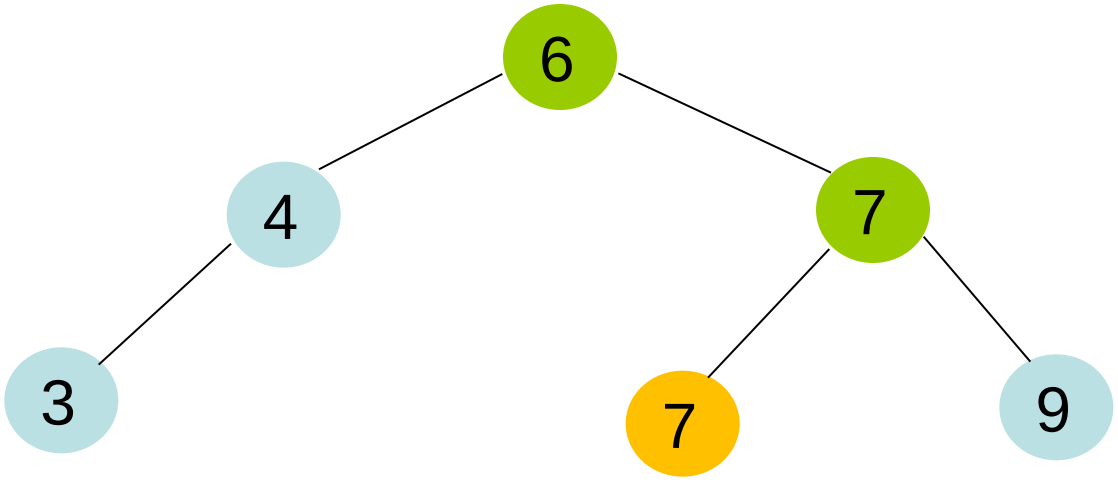


Datenstrukturen

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

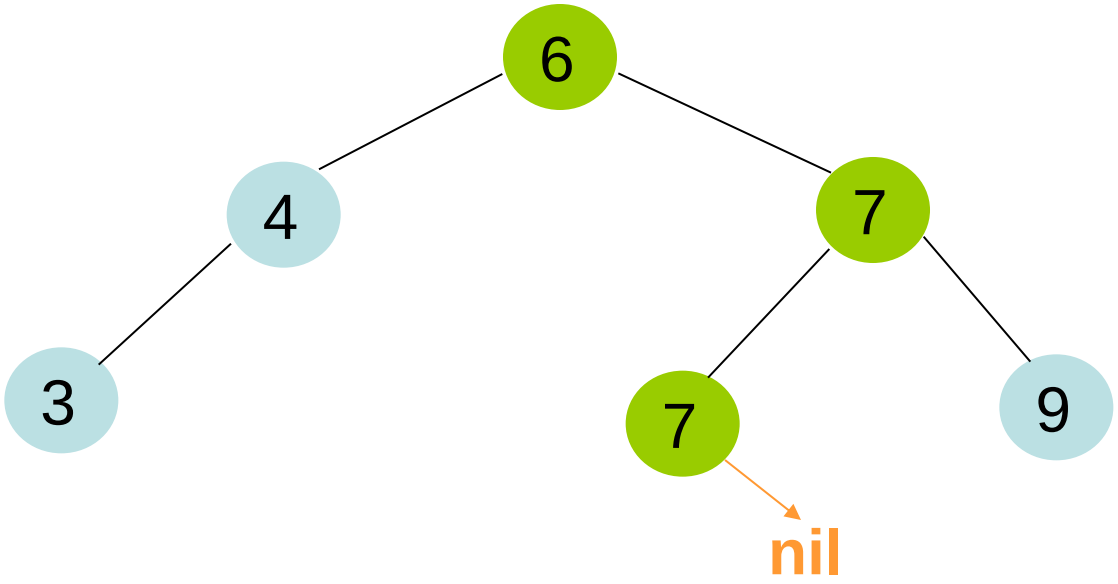


Datenstrukturen

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

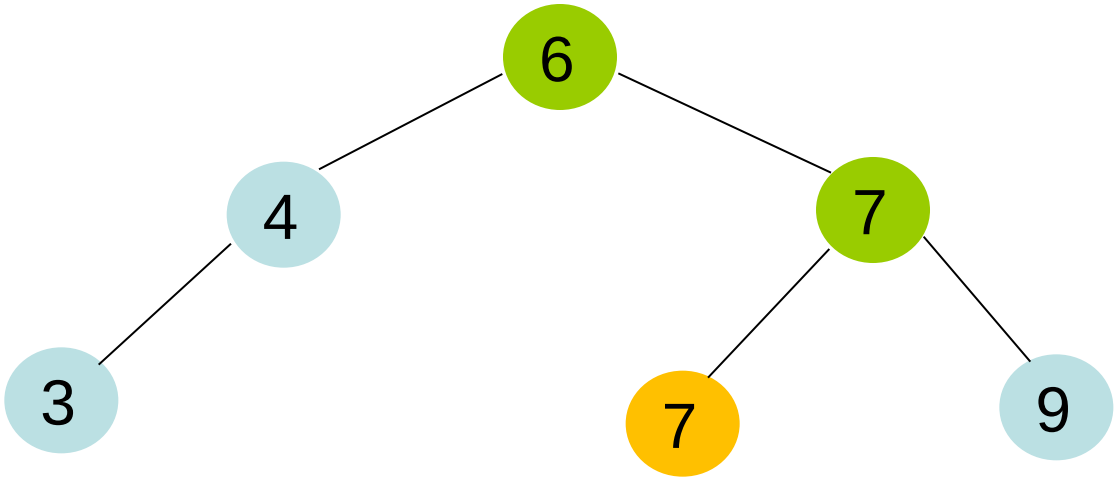


Datenstrukturen

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

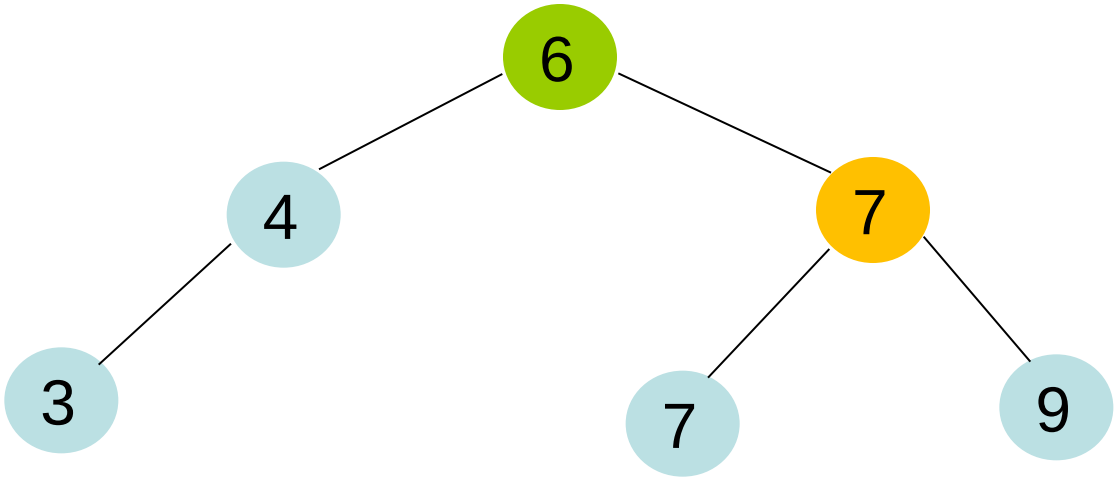


Datenstrukturen

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

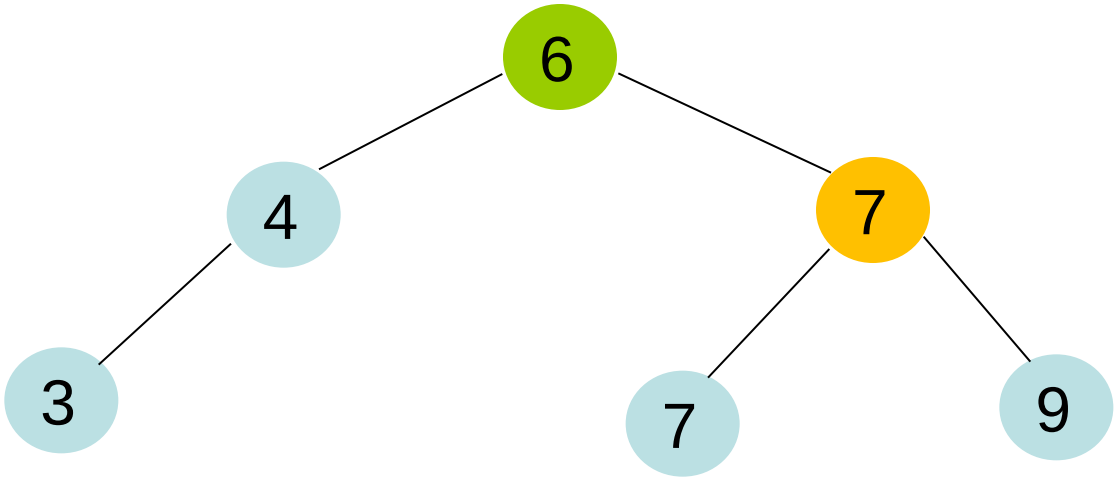


Datenstrukturen

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

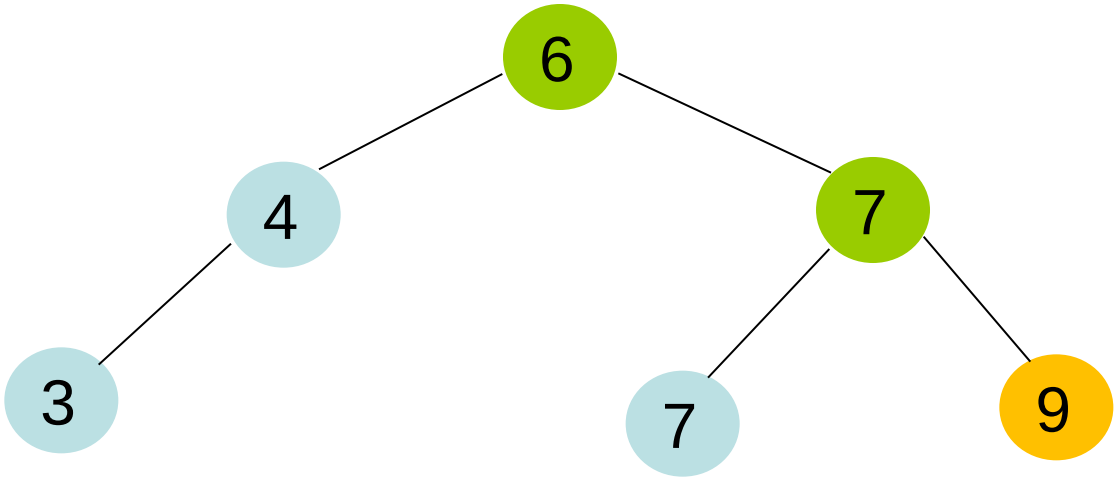


Datenstrukturen

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

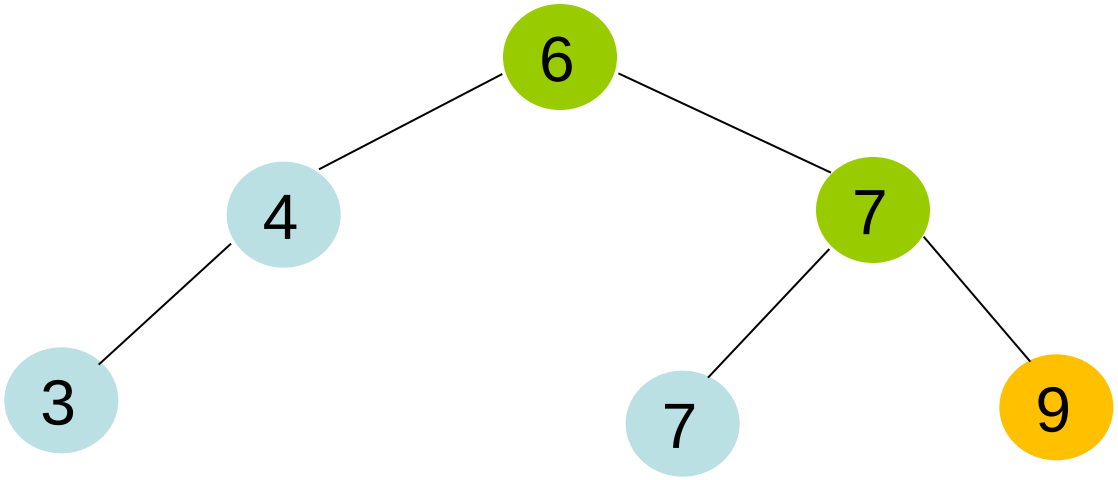


Datenstrukturen

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

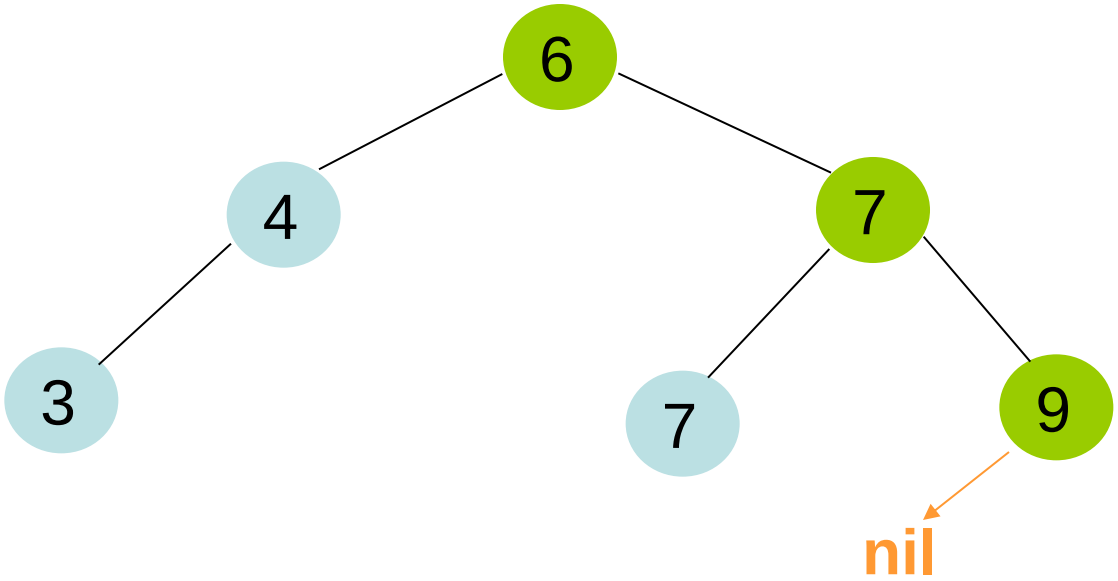


Datenstrukturen

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

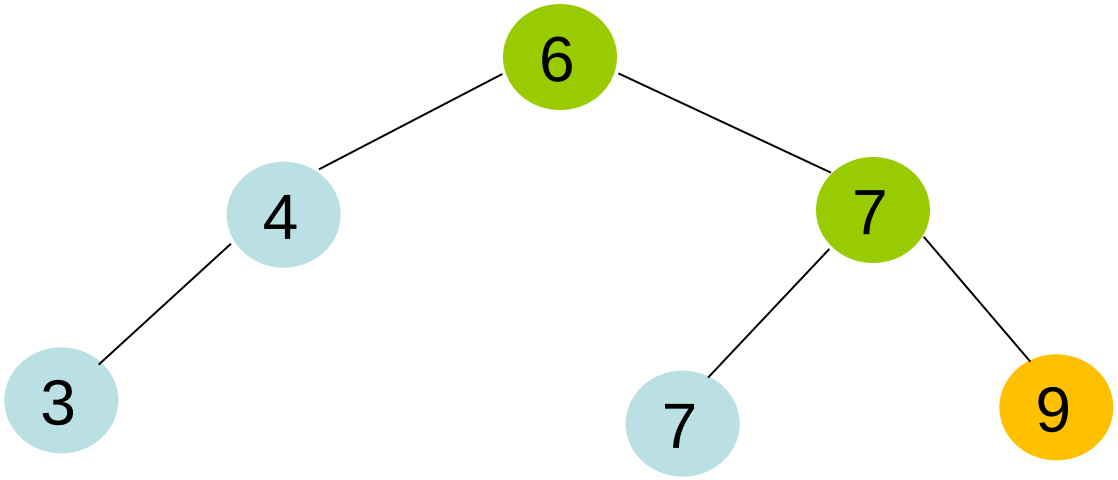


Datenstrukturen

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

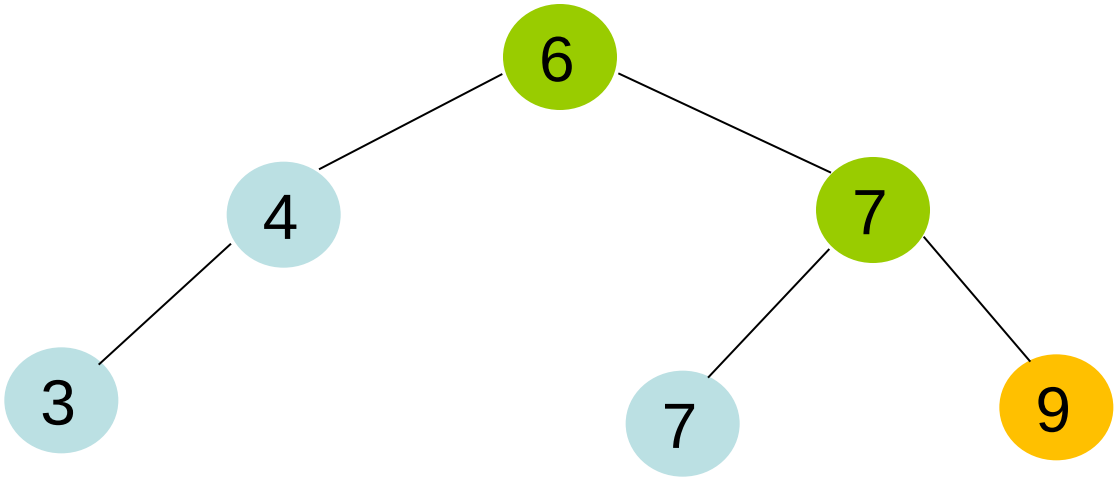


Datenstrukturen

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

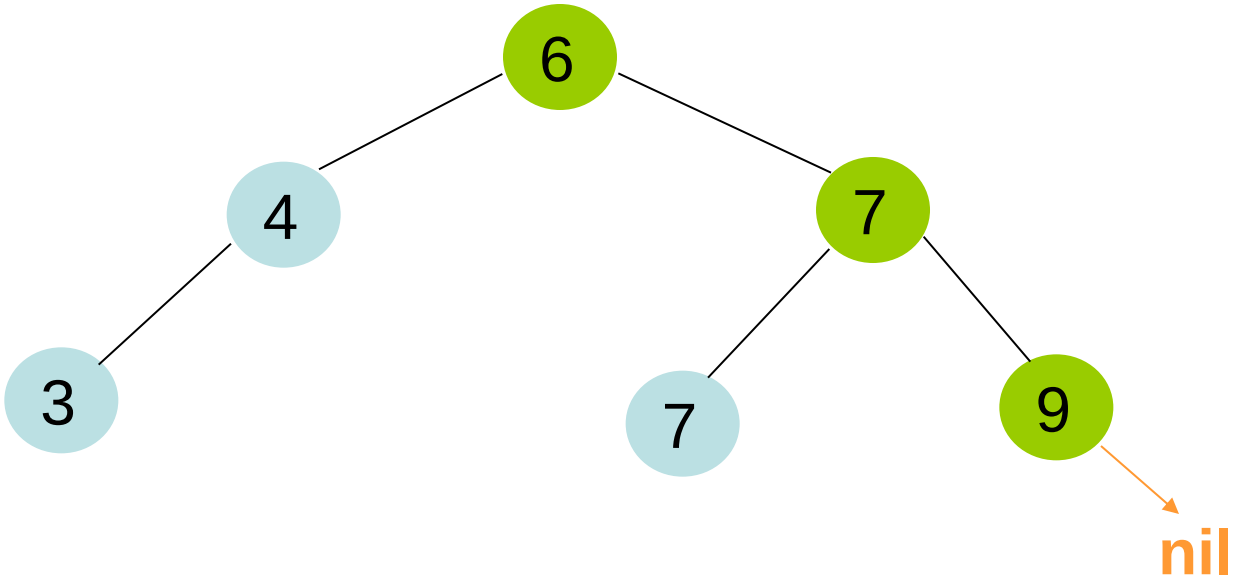


Datenstrukturen

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

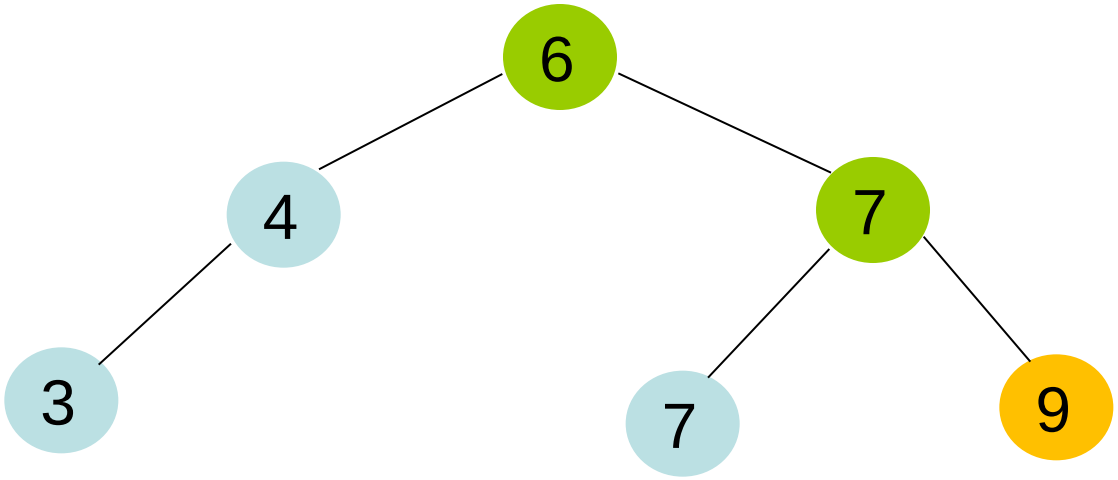


Datenstrukturen

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

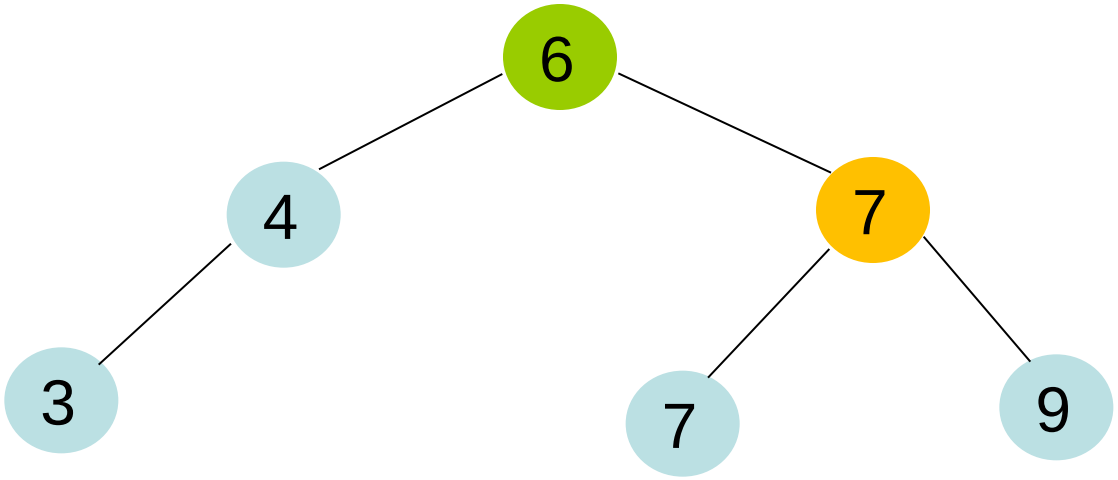


Datenstrukturen

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

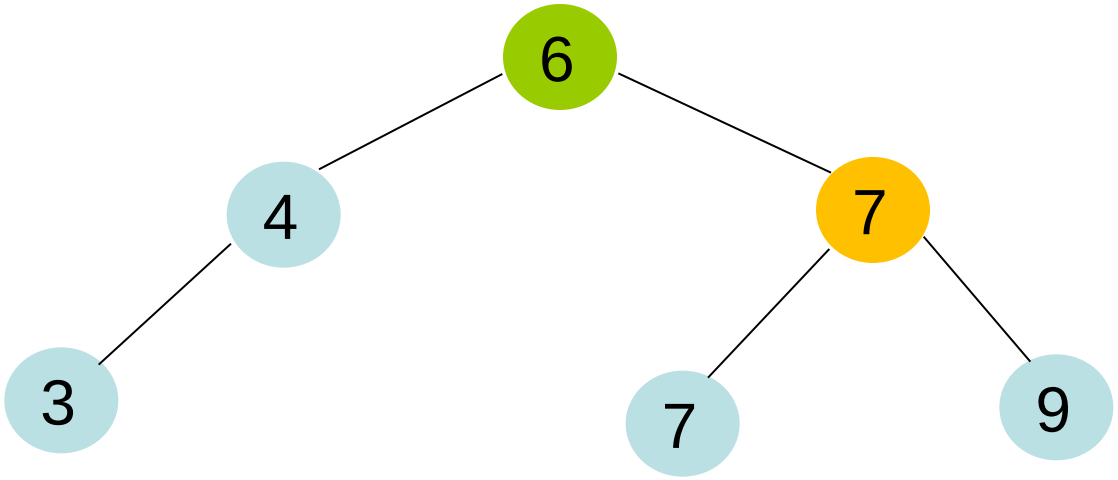


Datenstrukturen

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

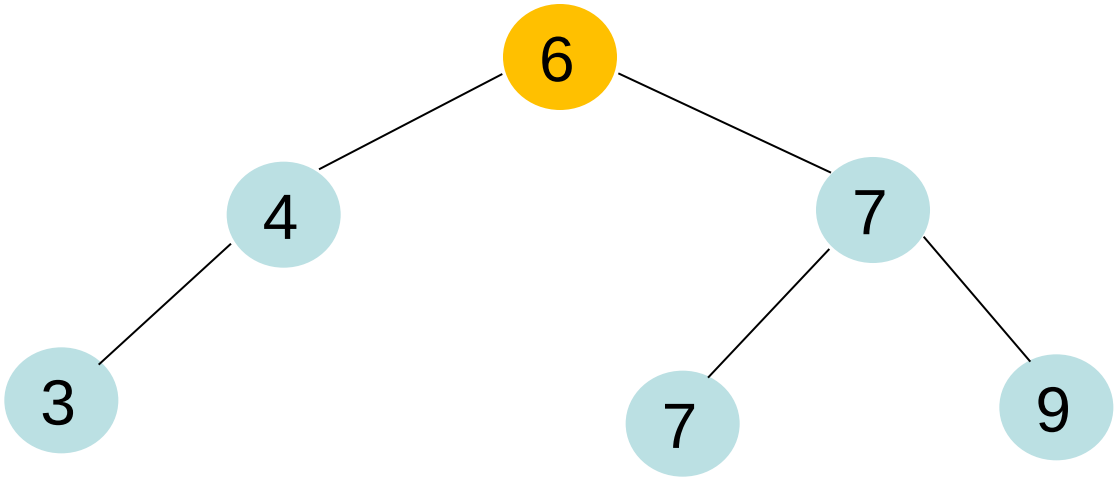


Datenstrukturen

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

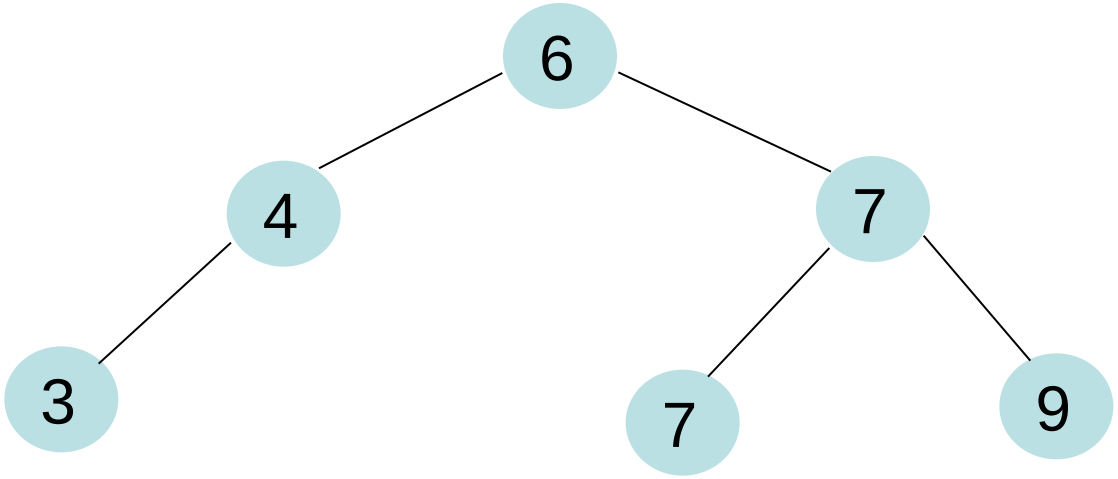


Datenstrukturen

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



Datenstrukturen

Lemma

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

Datenstrukturen

„Normale“ Induktion

- Wir wollen zeigen, dass Aussage A(i) für alle natürlichen Zahlen i gilt
- Dazu beweisen wir, dass

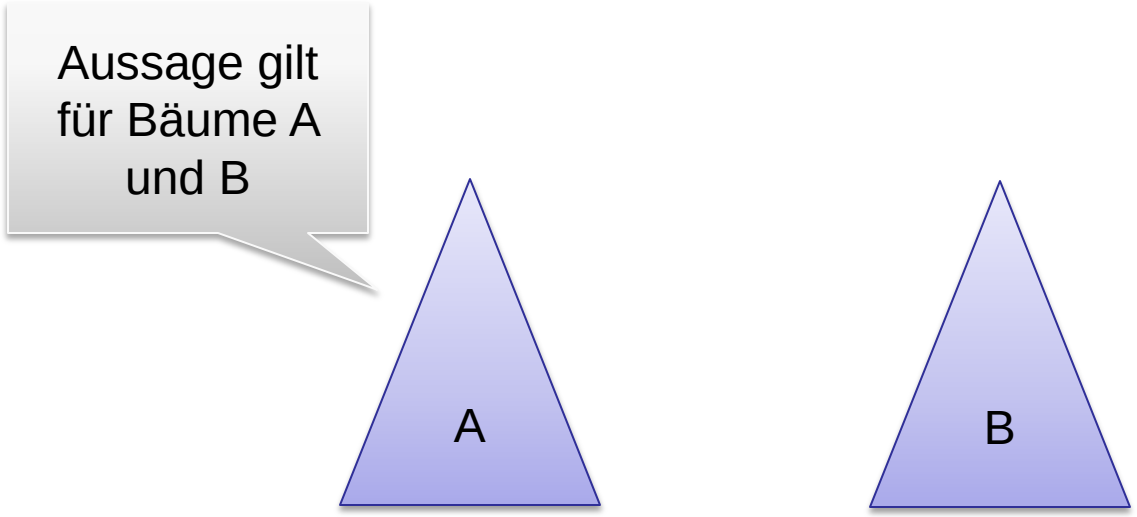
- *(a) A(1) gilt*
- *(b) Wenn A(i) gilt, dann gilt auch A(i+1)*

- (a) heißt Induktionsanfang
- (b) nennt man Induktionsschluss (oder auch Induktionsschritt)
- Die Vorraussetzung in (b) (also A(i)) heißt Induktionsvoraussetzung

Datenstrukturen

Induktion über die Struktur von Binärbäumen

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

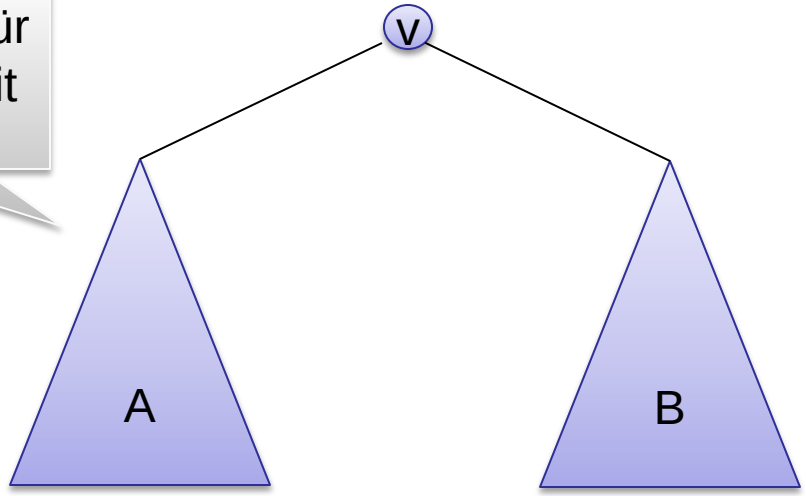


Datenstrukturen

Induktion über die Struktur von Binärbäumen

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

Dann gilt Aussage auch für neuen Baum mit Wurzel v



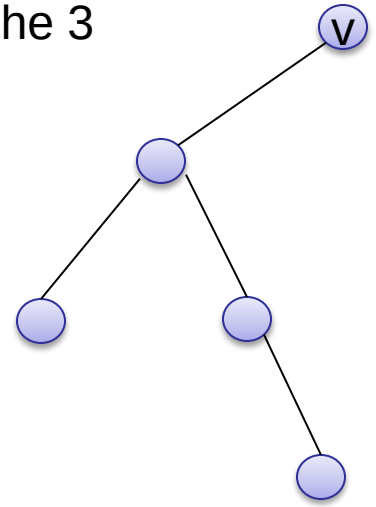
Datenstrukturen

Definition

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

Beispiel

- Baum der Höhe 3



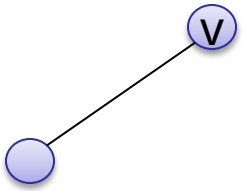
Datenstrukturen

Definition

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

Beispiel

- Baum der Höhe 1



Datenstrukturen

Definition

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

Beispiel

- Baum der Höhe 0

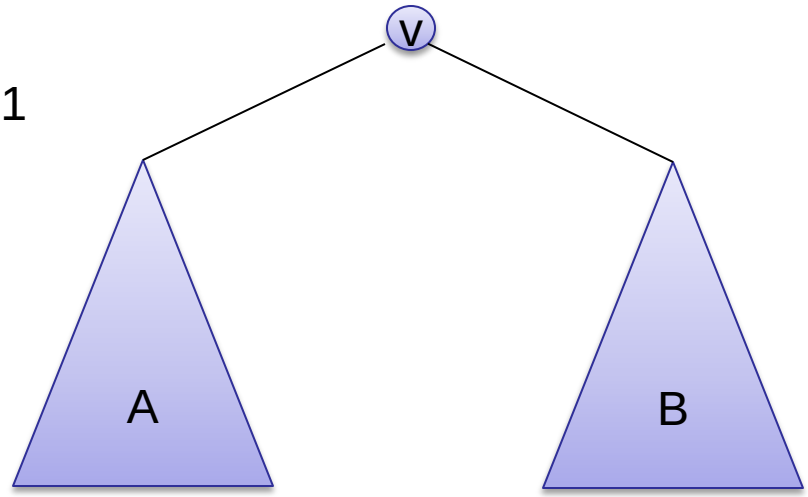
Datenstrukturen

Definition

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

Beispiel

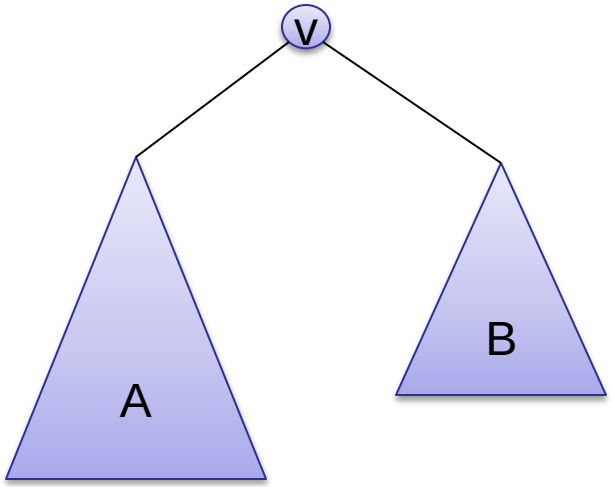
- Übereinkunft: Ein leerer Baum hat Höhe -1
- Damit gilt:
  Höhe eines Baumes mit Wurzel v und Teilbäumen A und B ist
  1 + max{ Höhe(A), Höhe(B)}



Datenstrukturen

Induktion über die Struktur von Binärbäumen

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

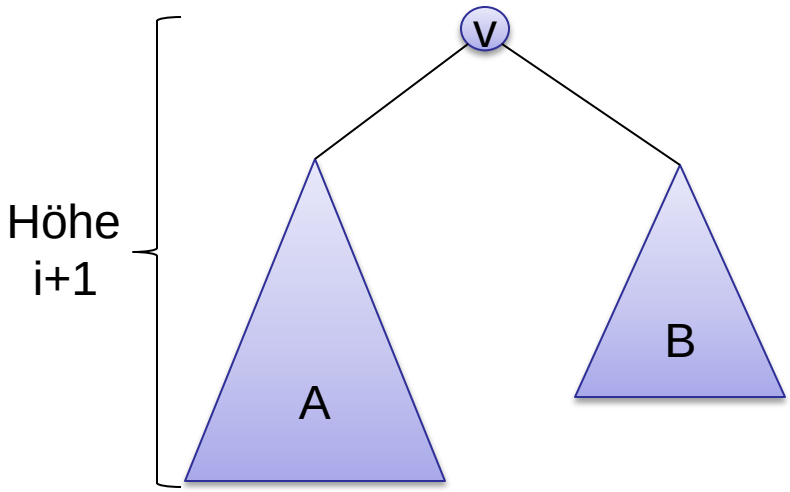


Datenstrukturen

Induktion über die Struktur von Binärbäumen

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

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

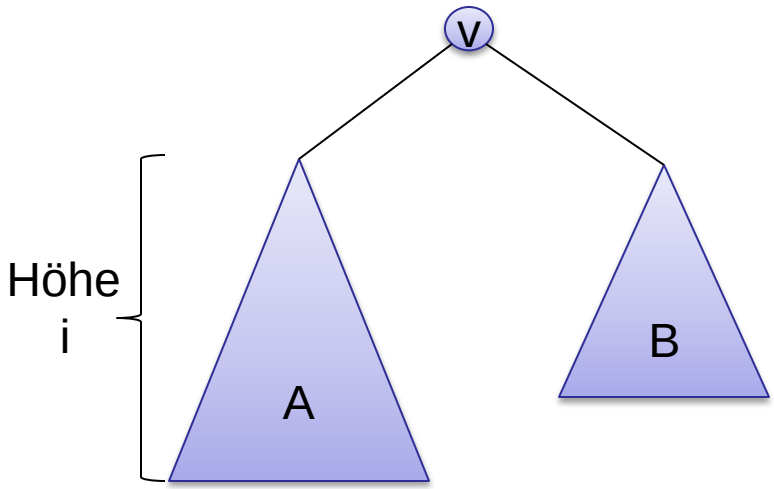


Datenstrukturen

Induktion über die Struktur von Binärbäumen

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

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



Datenstrukturen

```
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])
```

Lemma

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

Beweis

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

Datenstrukturen

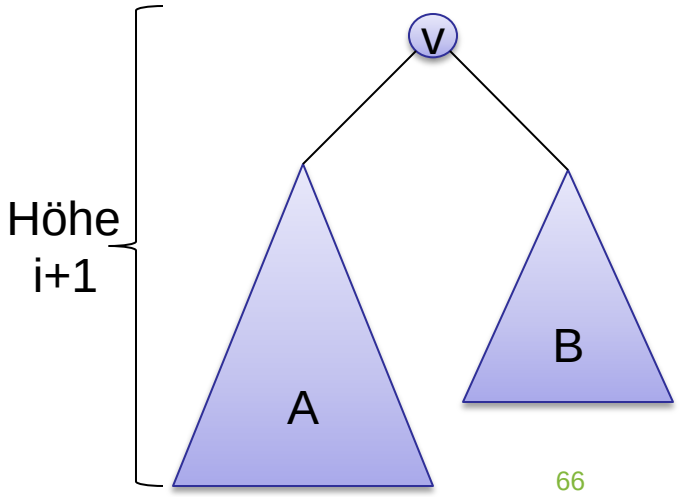
```
Inorder-Tree-Walk(x)
1. if x≠nil then
2.   Inorder-Tree-Walk(lc[x])
3.   Ausgabe key[x]
4.   Inorder-Tree-Walk(rc[x])
```

Lemma

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

Beweis

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



Datenstrukturen

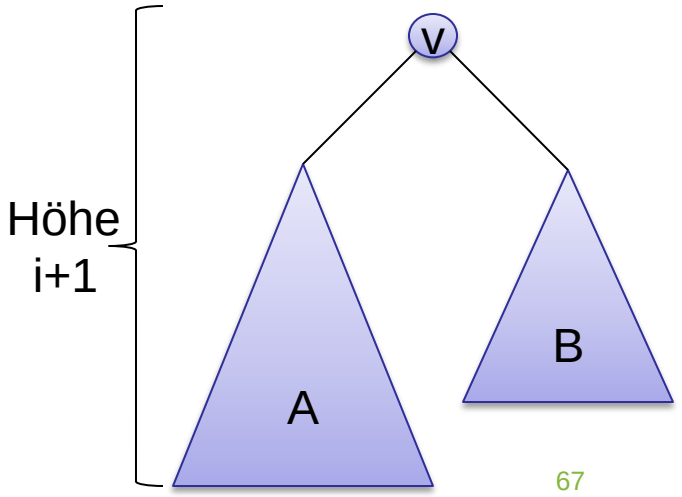
```
Inorder-Tree-Walk(x)
1. if x≠nil then
2.   Inorder-Tree-Walk(lc[x])
3.   Ausgabe key[x]
4.   Inorder-Tree-Walk(rc[x])
```

Lemma

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

Beweis

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



Datenstrukturen

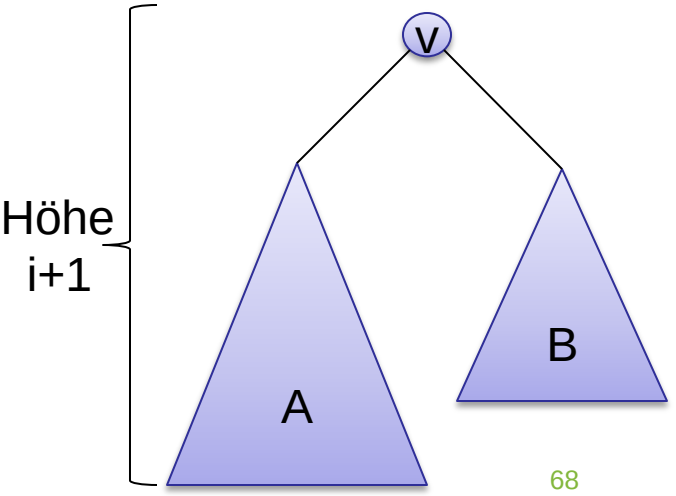
```
Inorder-Tree-Walk(x)
1. if x≠nil then
2.   Inorder-Tree-Walk(lc[x])
3.   Ausgabe key[x]
4.   Inorder-Tree-Walk(rc[x])
```

Lemma

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

Beweis

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



Datenstrukturen

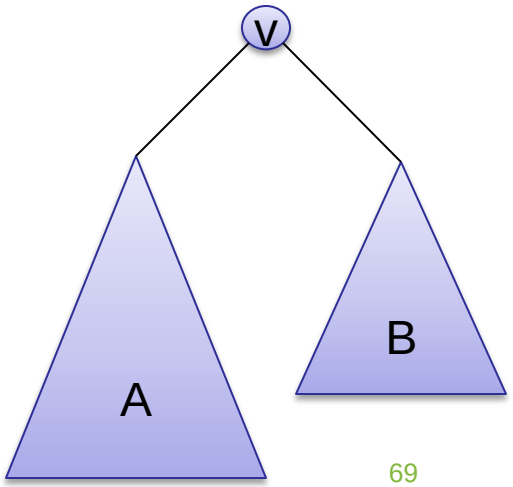
```
Inorder-Tree-Walk(x)
1. if x≠nil then
2.   Inorder-Tree-Walk(lc[x])
3.   Ausgabe key[x]
4.   Inorder-Tree-Walk(rc[x])
```

Lemma

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

Beweis

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



Datenstrukturen

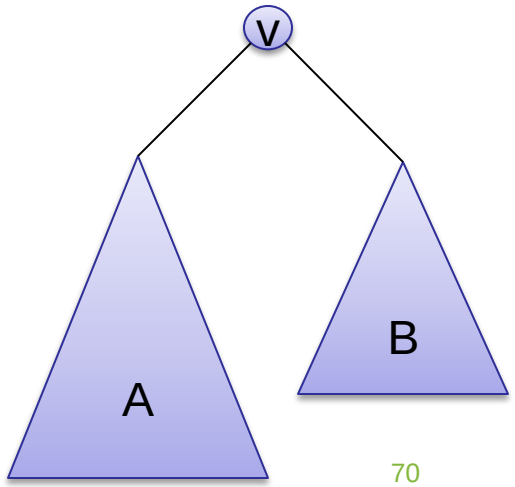
```
Inorder-Tree-Walk(x)
1.  if x≠nil then
2.    Inorder-Tree-Walk(lc[x])
3.    Ausgabe key[x]
4.    Inorder-Tree-Walk(rc[x])
```

Lemma

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

Beweis

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



Datenstrukturen

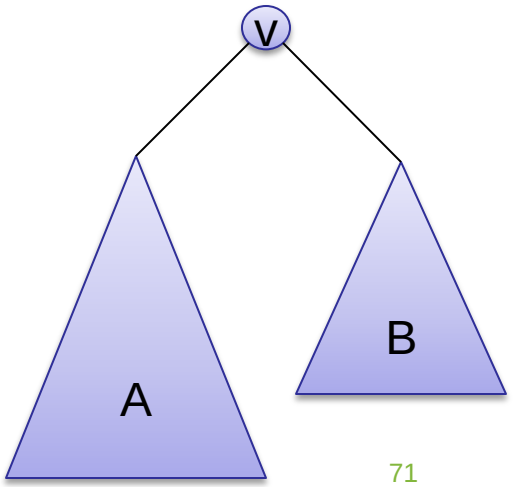
```
Inorder-Tree-Walk(x)
1.  if x≠nil then
2.    Inorder-Tree-Walk(lc[x])
3.    Ausgabe key[x]
4.    Inorder-Tree-Walk(rc[x])
```

Lemma

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

Beweis

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



Datenstrukturen

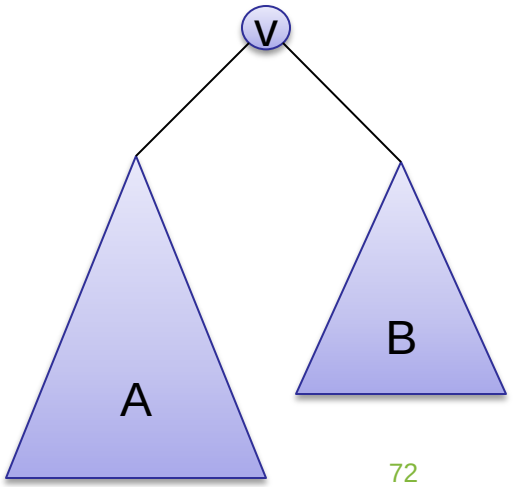
```
Inorder-Tree-Walk(x)
1.  if x≠nil then
2.    Inorder-Tree-Walk(lc[x])
3.    Ausgabe key[x]
4.    Inorder-Tree-Walk(rc[x])
```

Lemma

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

Beweis

- Zeile 2: Aufruf für Teilbaum A der Höhe ≤i
  Nach (I.V.): Schlüssel aus A werden in aufsteigender Reihenfolge ausgegeben



Datenstrukturen

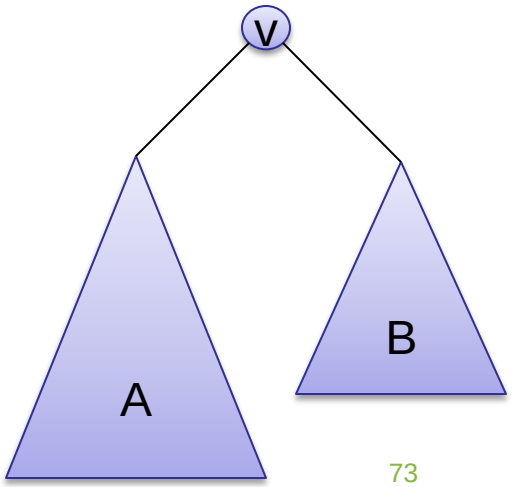
```
Inorder-Tree-Walk(x)
1.  if x≠nil then
2.    Inorder-Tree-Walk(lc[x])
3.    Ausgabe key[x]
4.    Inorder-Tree-Walk(rc[x])
```

Lemma

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

Beweis

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



Datenstrukturen

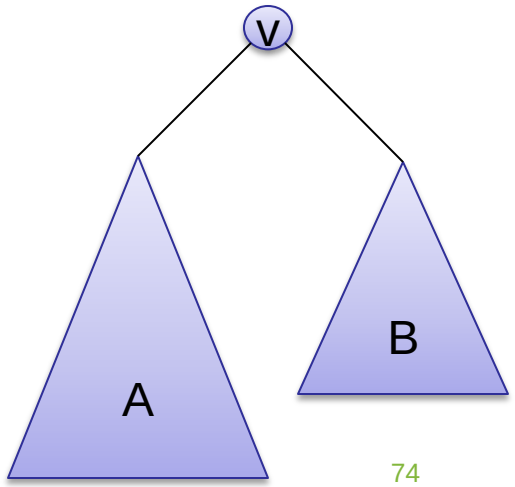
```
Inorder-Tree-Walk(x)
1.  if x≠nil then
2.    Inorder-Tree-Walk(lc[x])
3.    Ausgabe key[x]
4.    Inorder-Tree-Walk(rc[x])
```

Lemma

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

Beweis

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



Datenstrukturen

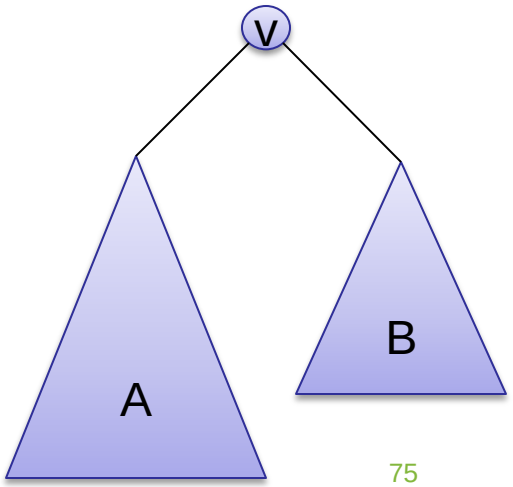
```
Inorder-Tree-Walk(x)
1.  if x≠nil then
2.    Inorder-Tree-Walk(lc[x])
3.    Ausgabe key[x]
4.    Inorder-Tree-Walk(rc[x])
```

Lemma

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

Beweis

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



Datenstrukturen

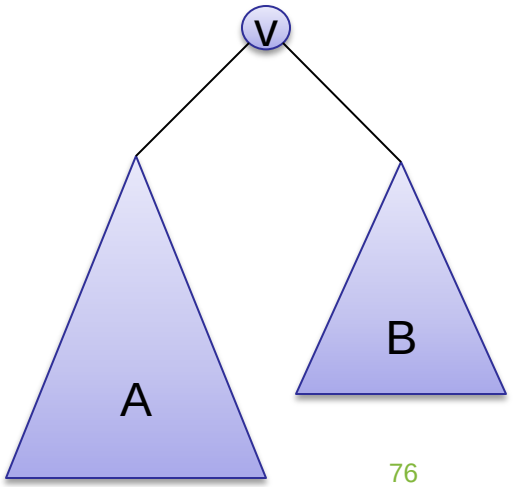
```
Inorder-Tree-Walk(x)
1. if x≠nil then
2.   Inorder-Tree-Walk(lc[x])
3.   Ausgabe key[x]
4.   Inorder-Tree-Walk(rc[x])
```

Lemma

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

Beweis

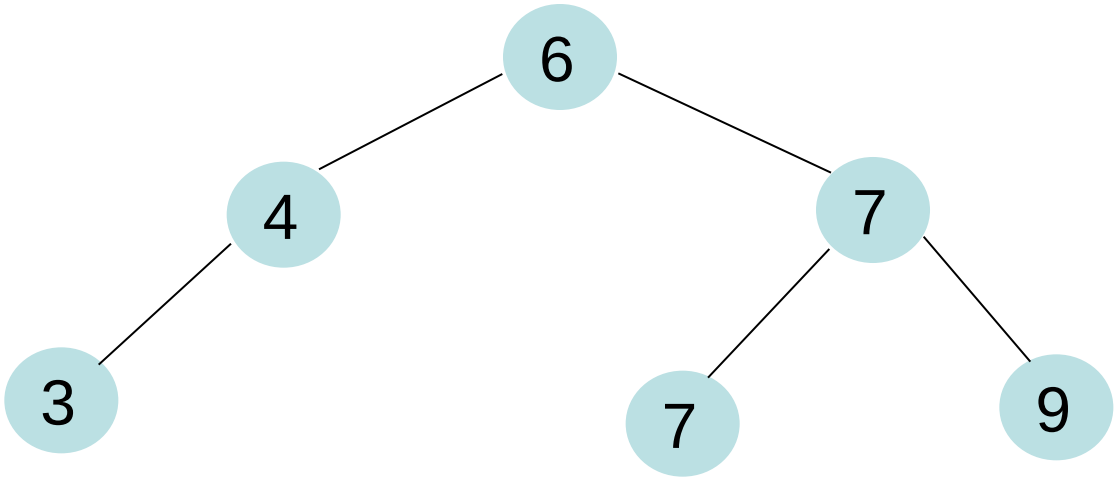
- Insgesamt:
- Schlüssel aus A aufsteigend, Schlüssel von v, Schlüssel aus B aufsteigend
- Nach Suchbaumeigenschaft ist dies aufsteigende Folge



Datenstrukturen

Suchen in Binärbäumen

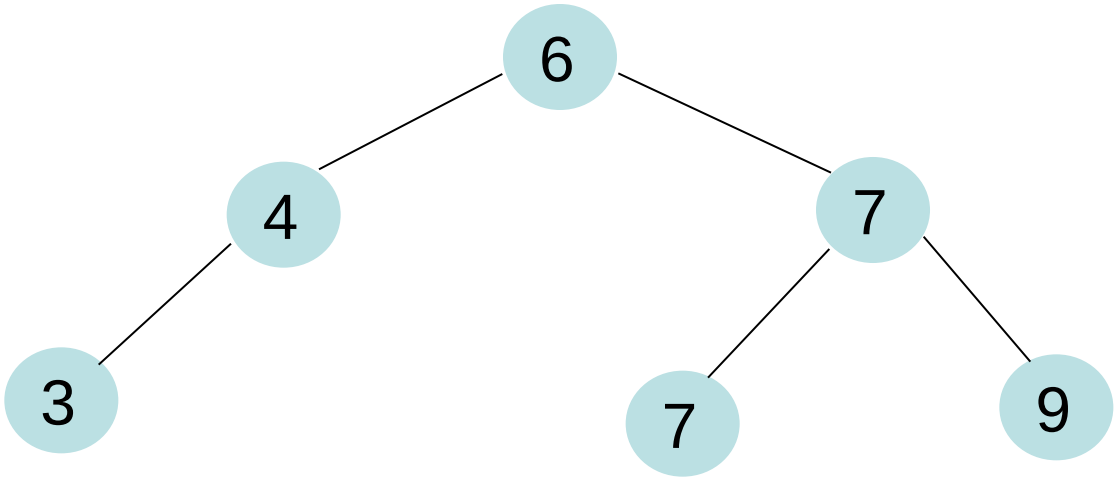
- Gegeben ist Schlüssel k
- Gesucht ist ein Knoten mit Schlüssel k



Datenstrukturen

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)

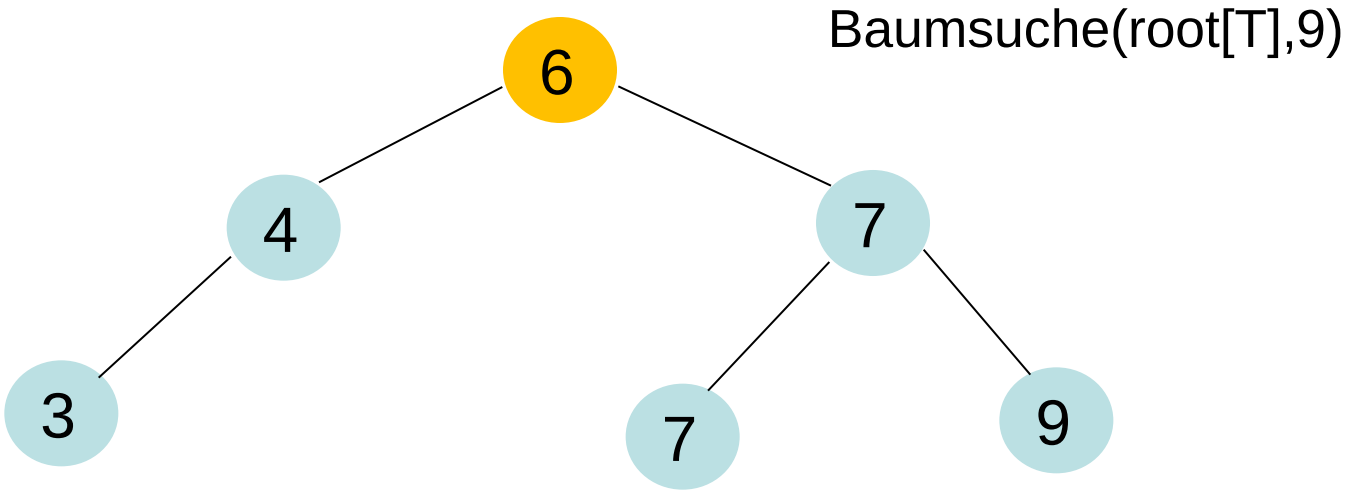


Datenstrukturen

Baumsuche(x,k)

Aufruf mit x=root[T]

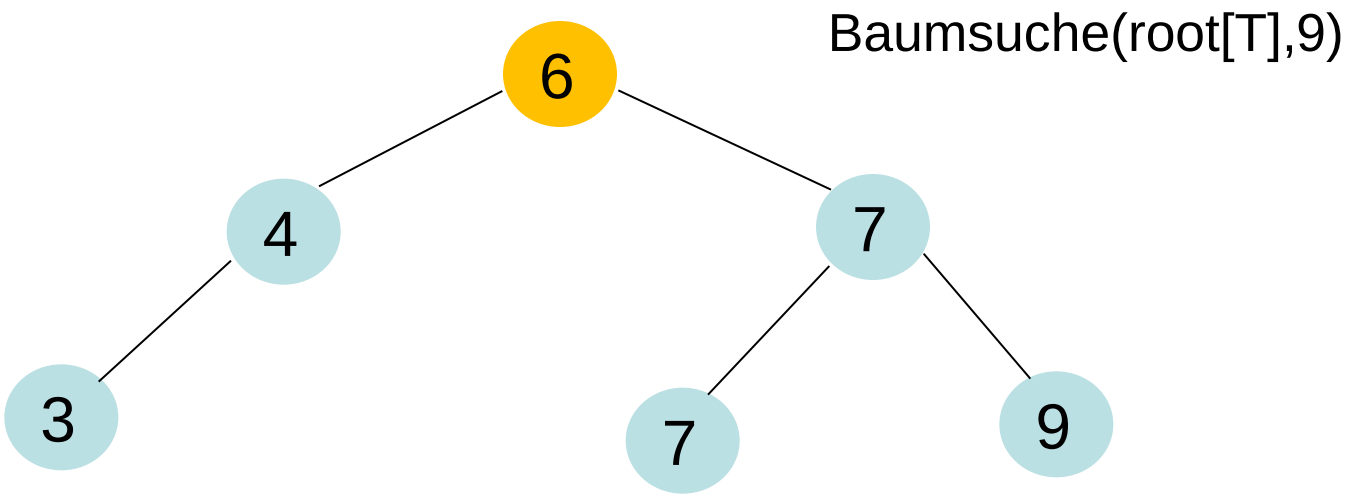
1. **if** x=**nil or** k=key[x] **then return** x
2. **if** k<key[x] **then return** Baumsuche(lc[x],k)
3. **else** return Baumsuche(rc[x],k)



Datenstrukturen

Baumsuche(x,k)

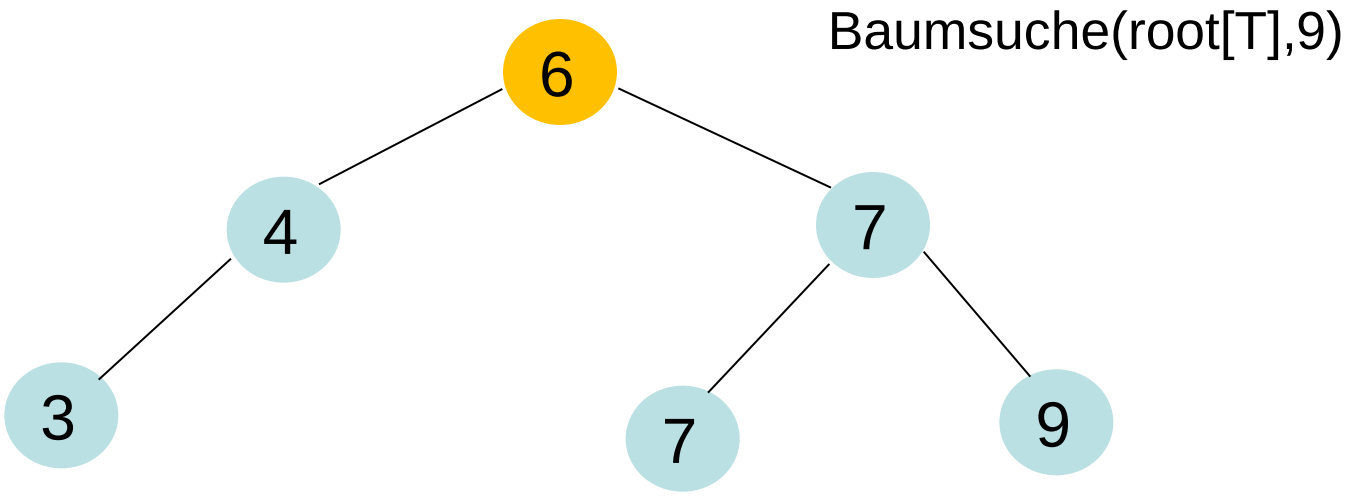
1. **if** x=**nil or** k=key[x] **then return** x
2. **if** k<key[x] **then return** Baumsuche(lc[x],k)
3. **else** return Baumsuche(rc[x],k)



Datenstrukturen

Baumsuche(x,k)

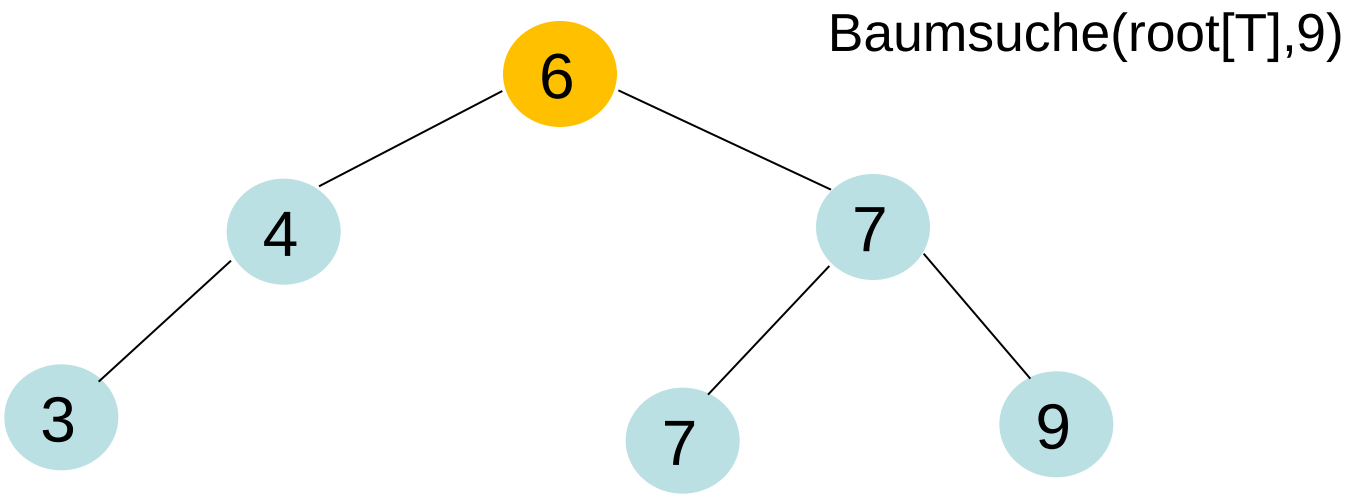
1. **if** x=**nil or** k=key[x] **then return** x
2. **if** k<key[x] **then return** Baumsuche(lc[x],k)
3. **else** return Baumsuche(rc[x],k)



Datenstrukturen

Baumsuche(x,k)

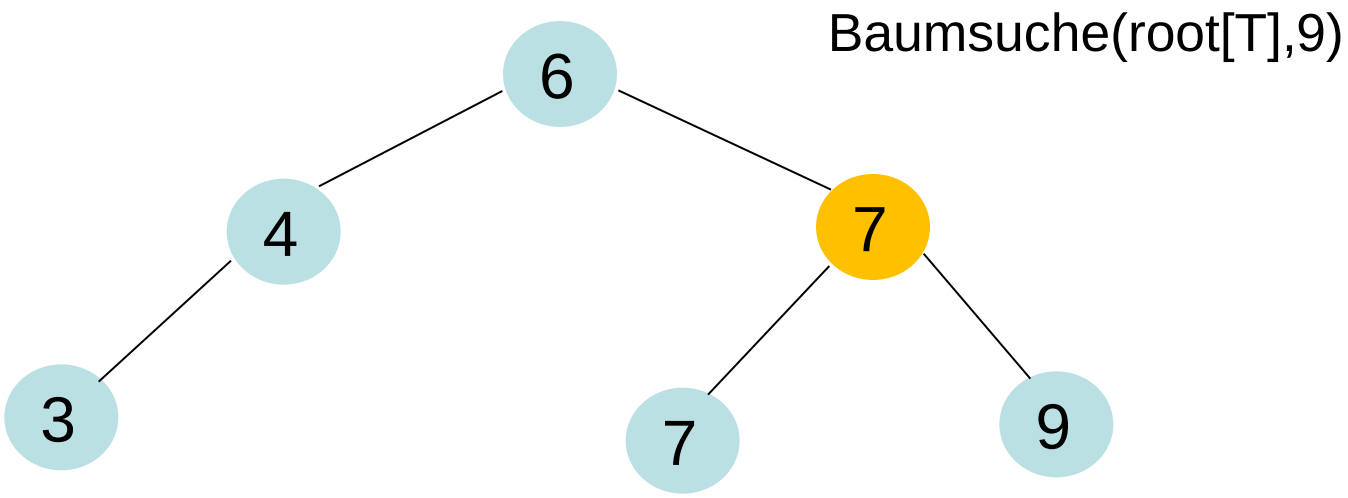
1. **if** x=**nil or** k=key[x] **then return** x
2. **if** k<key[x] **then return** Baumsuche(lc[x],k)
3. **else** return Baumsuche(rc[x],k)



Datenstrukturen

Baumsuche(x,k)

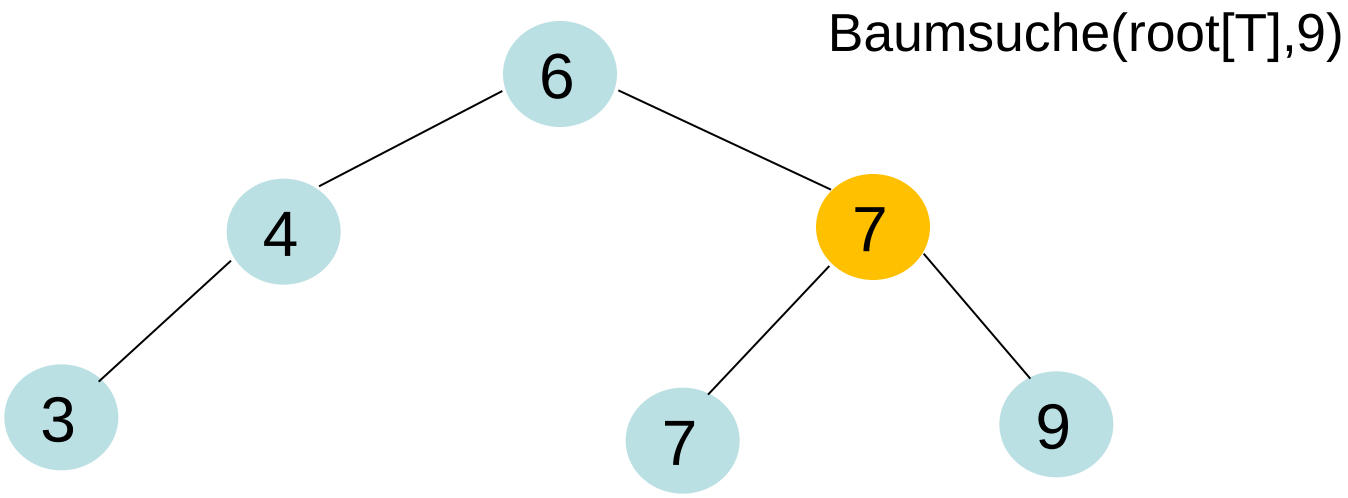
1. **if** x=**nil or** k=key[x] **then return** x
2. **if** k<key[x] **then return** Baumsuche(lc[x],k)
3. **else** return Baumsuche(rc[x],k)



Datenstrukturen

Baumsuche(x,k)

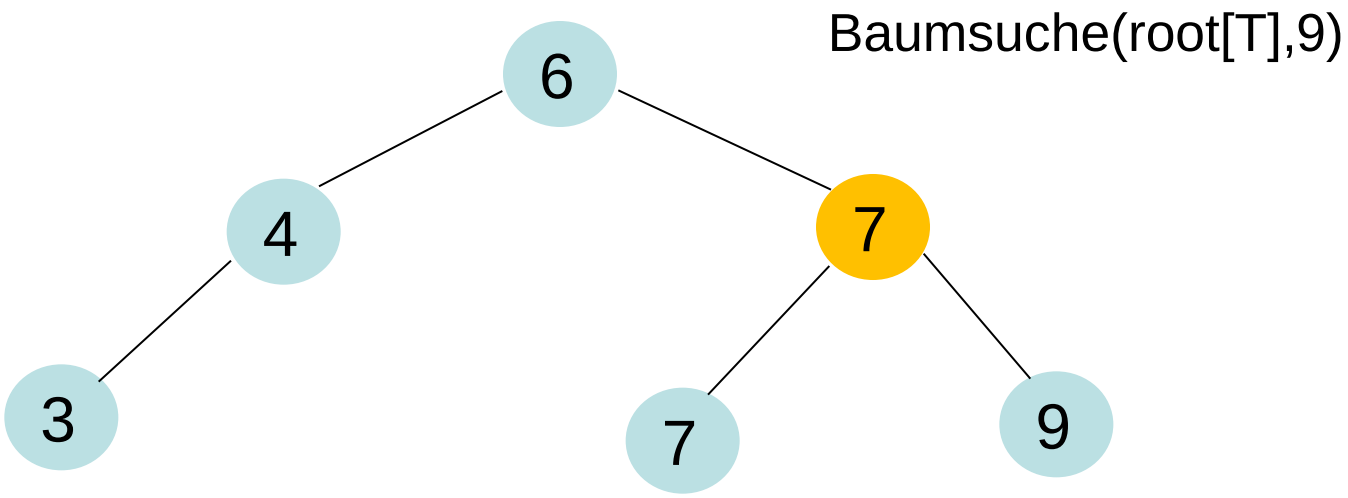
1. **if** x=**nil or** k=key[x] **then return** x
2. **if** k<key[x] **then return** Baumsuche(lc[x],k)
3. **else** return Baumsuche(rc[x],k)



Datenstrukturen

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)



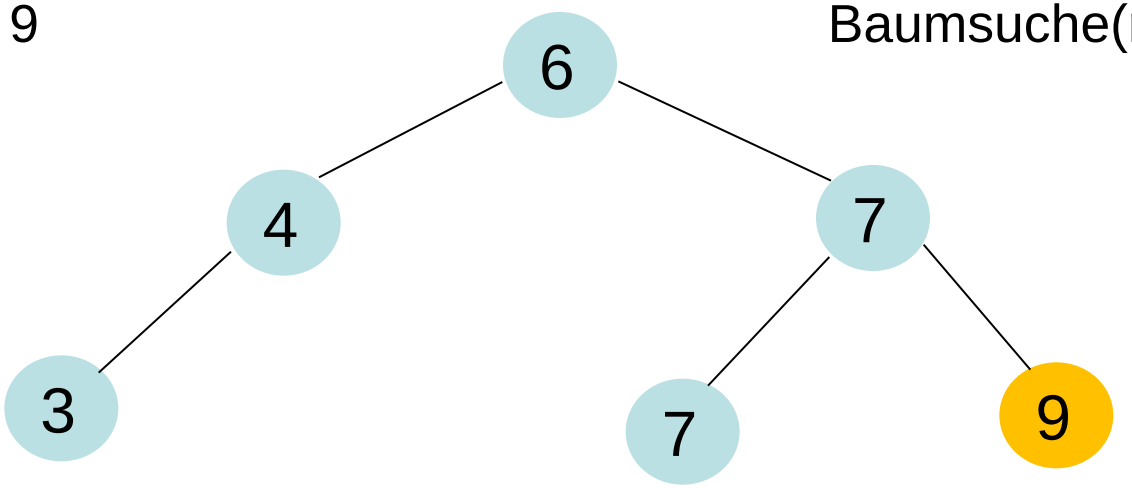
Datenstrukturen

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)

Ausgabe: 9

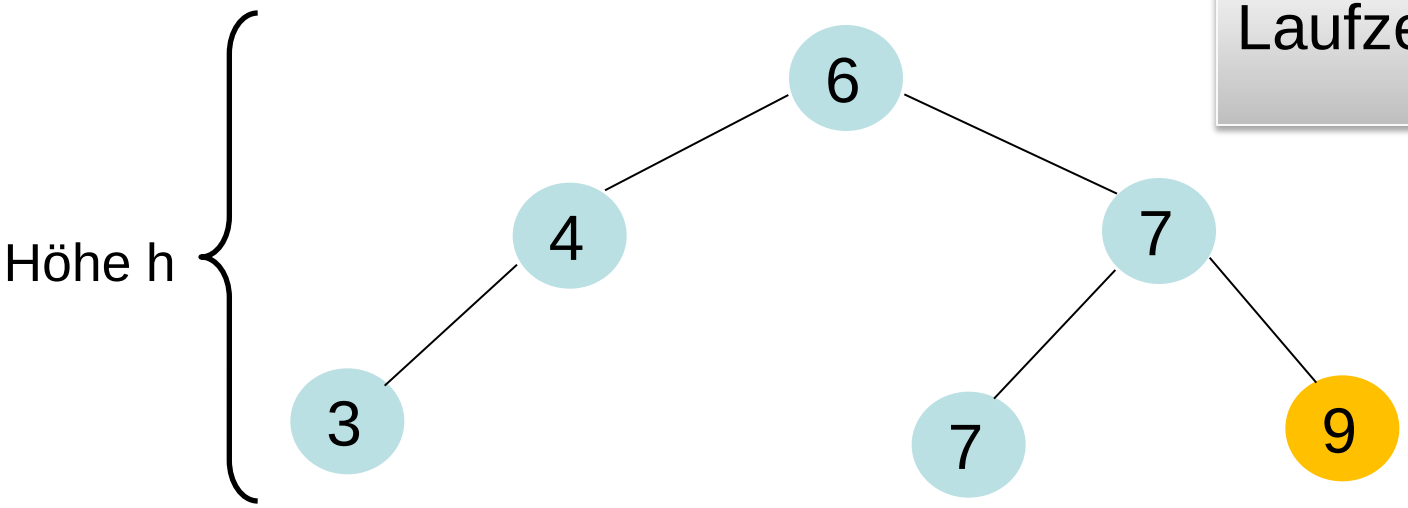
Baumsuche(root[T],9)



Datenstrukturen

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)

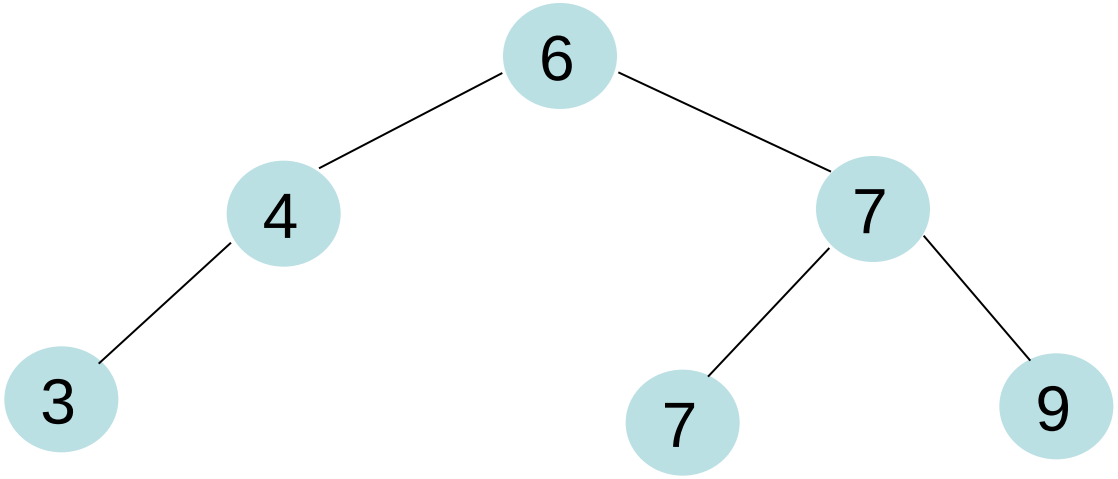


Laufzeit: O(h)

Datenstrukturen

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

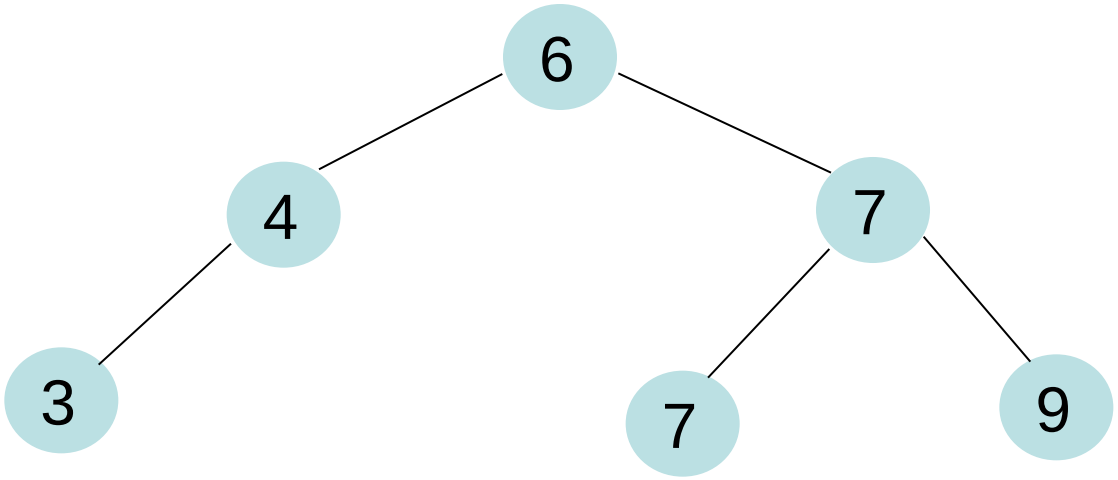


Datenstrukturen

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

Aufruf mit x=root[T]



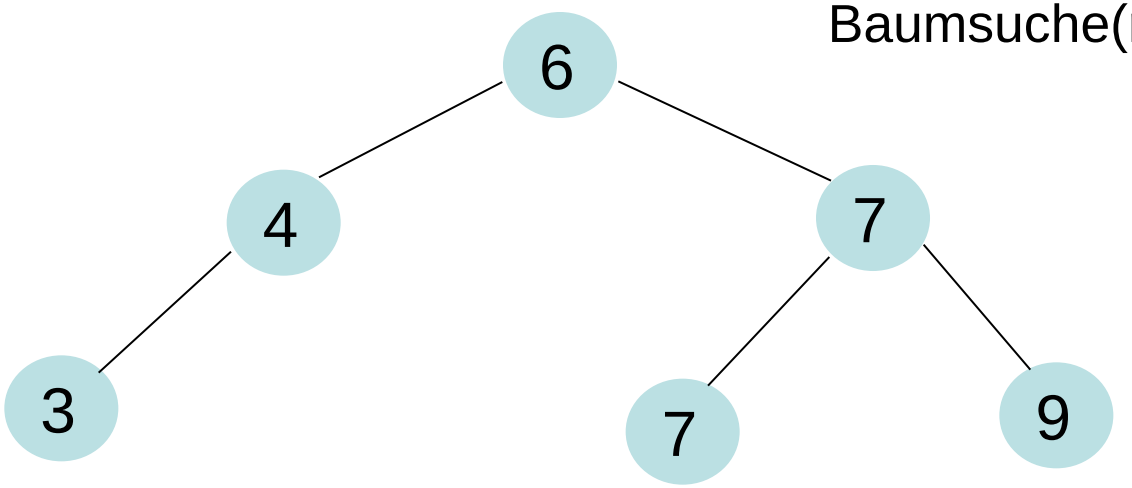
Datenstrukturen

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

Aufruf mit x=root[T]

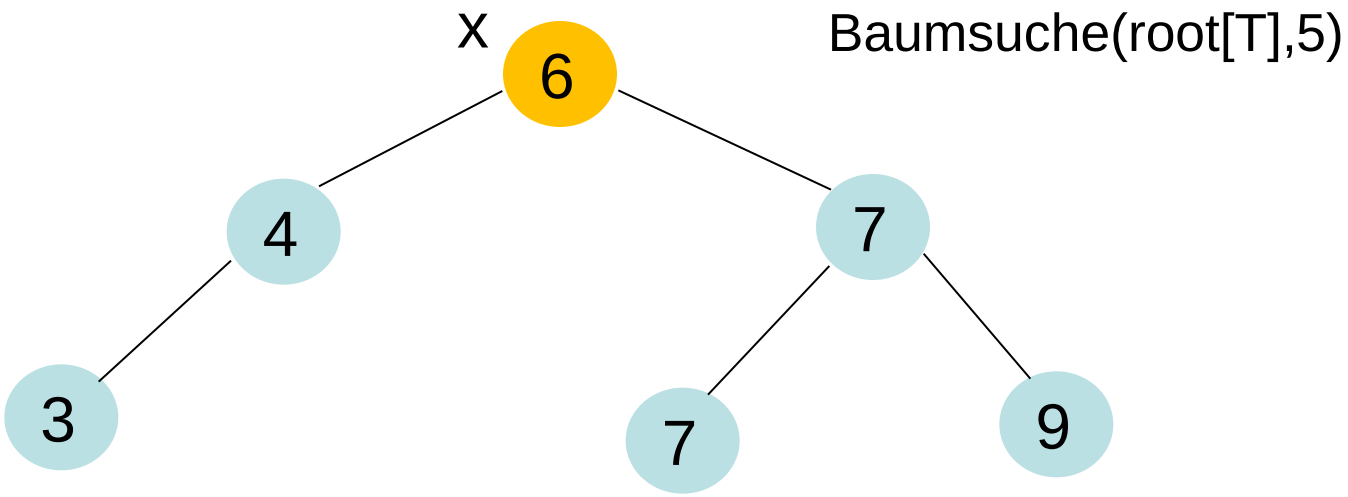
Baumsuche(root[T],5)



Datenstrukturen

IterativeBaumsuche(x,k)

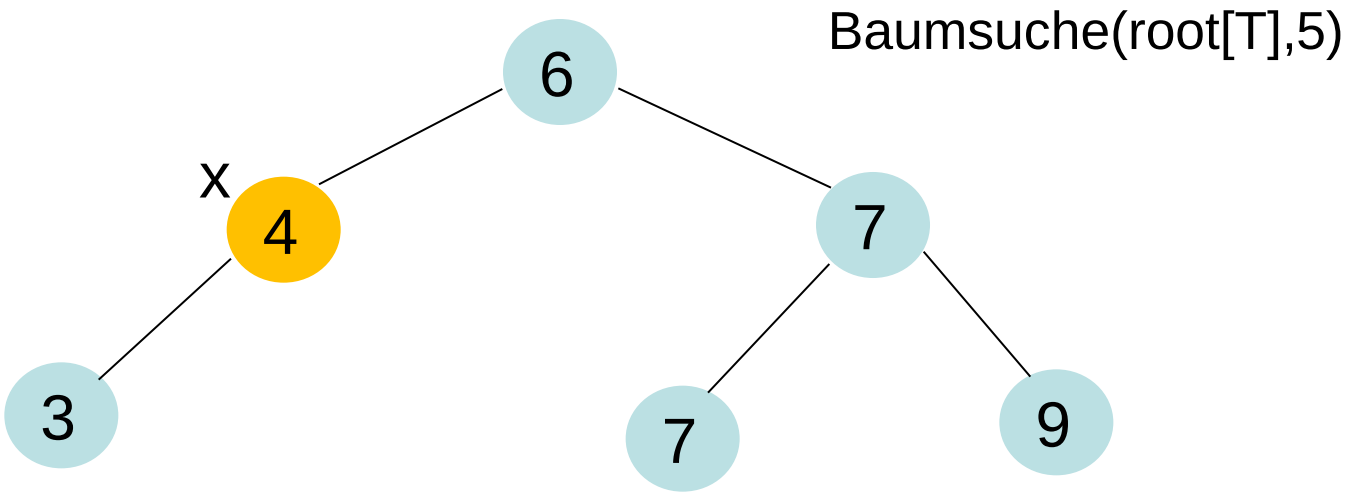
1. **while** x≠**nil** and k≠key[x] **do**
2. **if** k<key[x] **then** x ← lc[x]
3. **else** x ← rc[x]
4. **return** x



Datenstrukturen

IterativeBaumsuche(x,k)

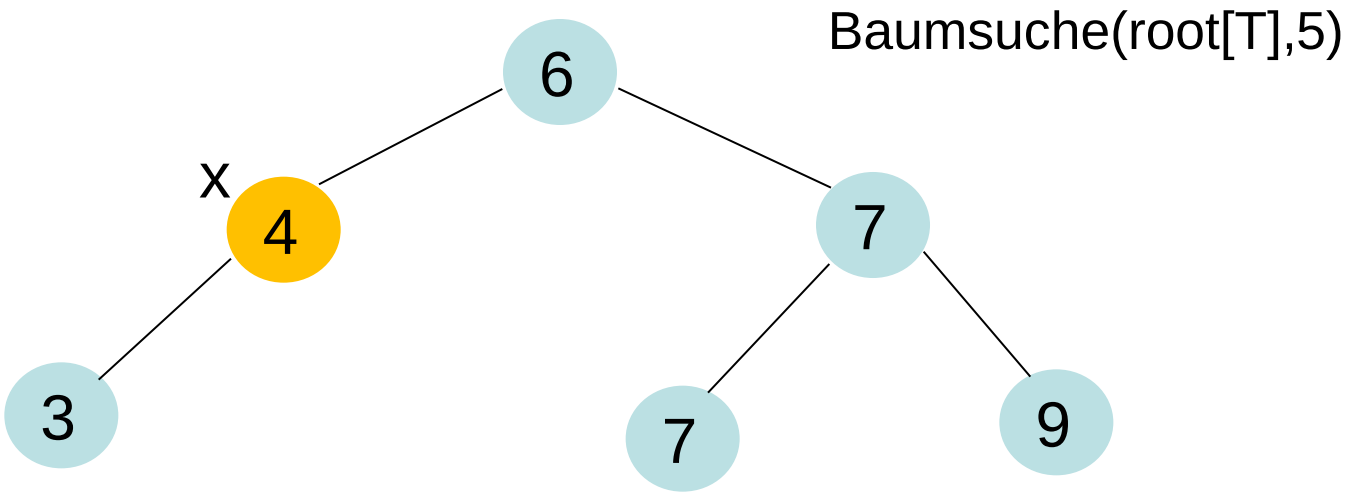
1. **while** x≠**nil** and k≠key[x] **do**
2. **if** k<key[x] **then** x ← lc[x]
3. **else** x ← rc[x]
4. **return** x



Datenstrukturen

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

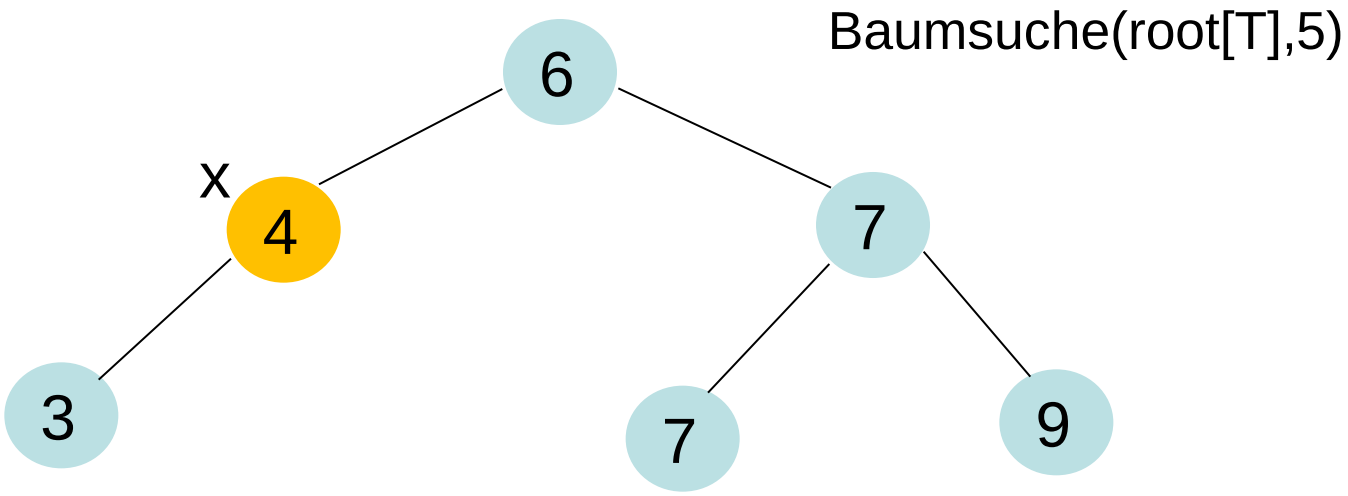


Datenstrukturen

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

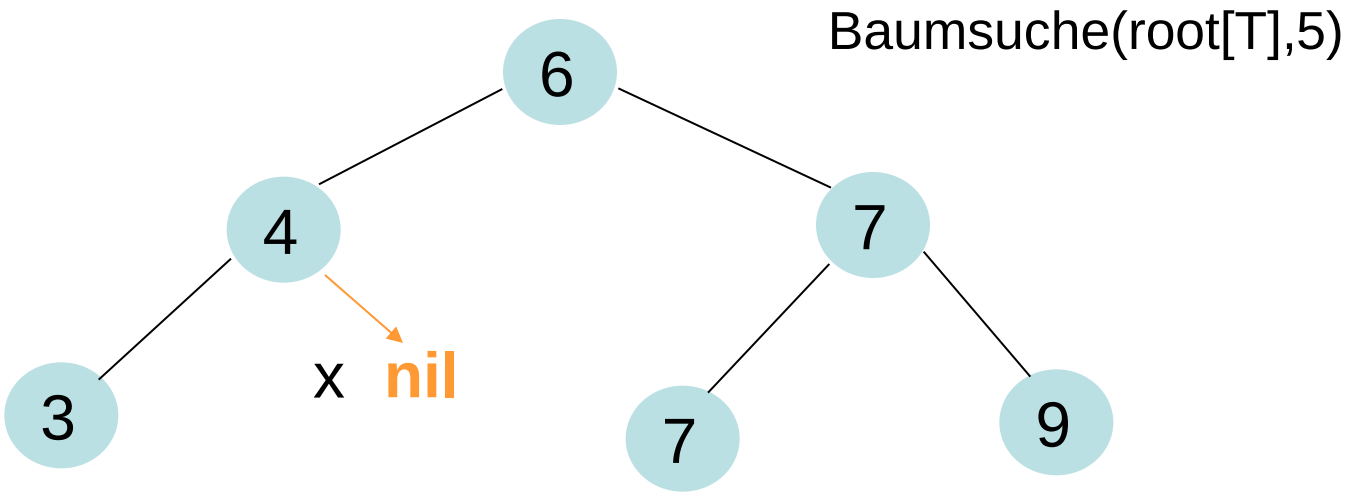
Aufruf mit x=root[T]



Datenstrukturen

IterativeBaumsuche(x,k)

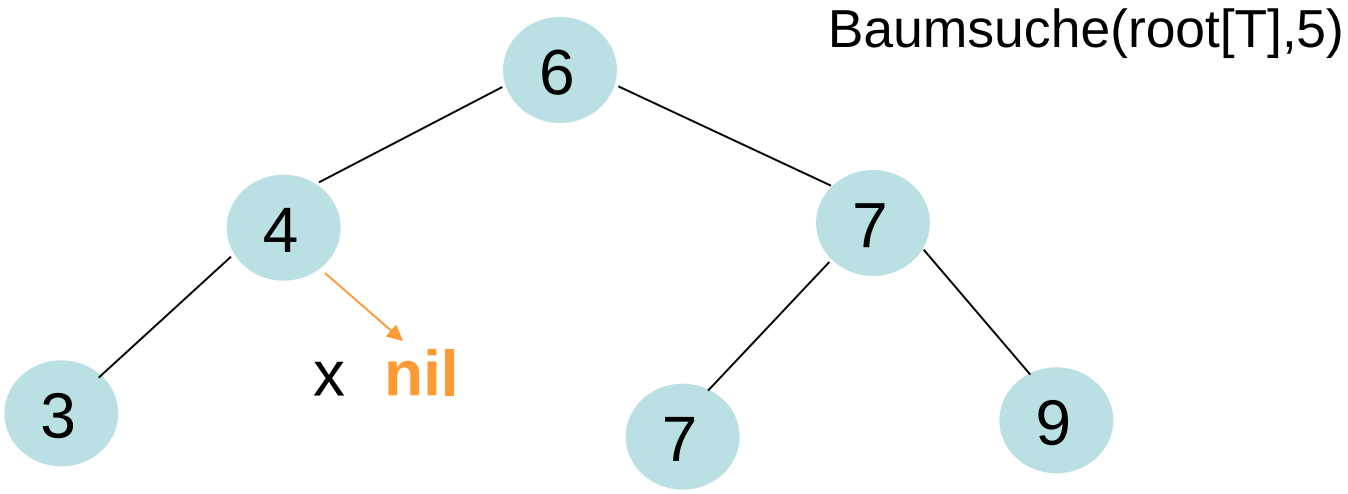
1. **while** x≠**nil** and k≠key[x] **do**
2. **if** k<key[x] **then** x ← lc[x]
3. **else** x ← rc[x]
4. **return** x



Datenstrukturen

IterativeBaumsuche(x,k)

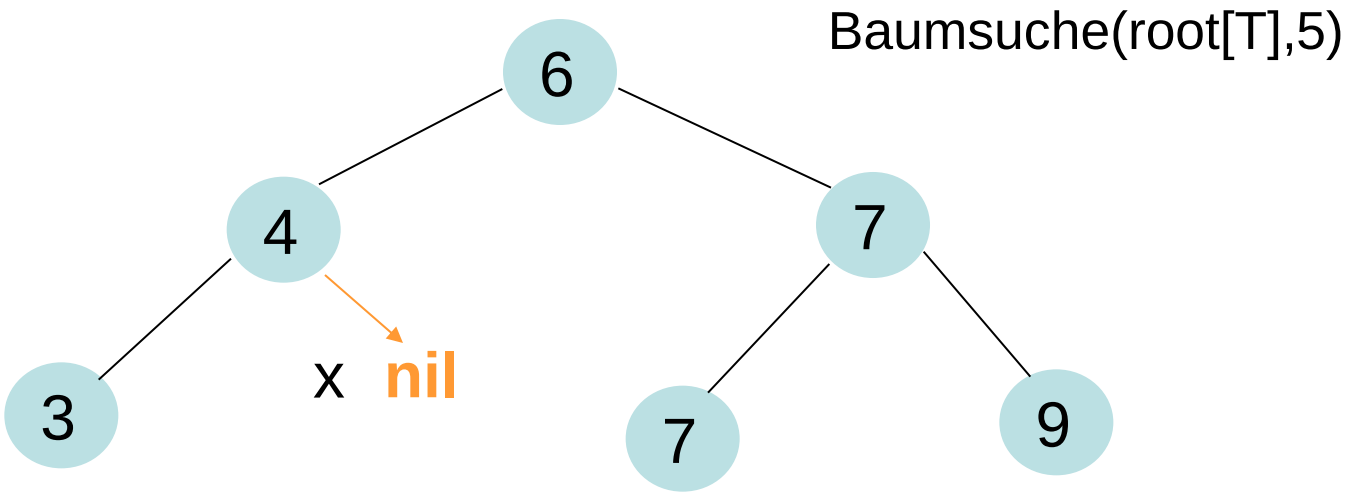
1. **while** x≠**nil** and k≠key[x] **do**
2. **if** k<key[x] **then** x ← lc[x]
3. **else** x ← rc[x]
4. **return** x



Datenstrukturen

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

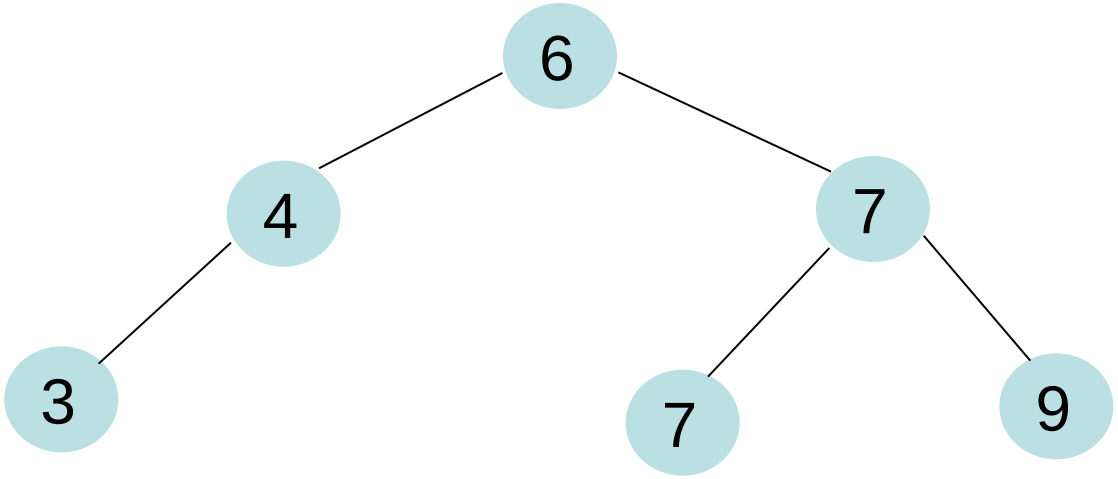


Datenstrukturen

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

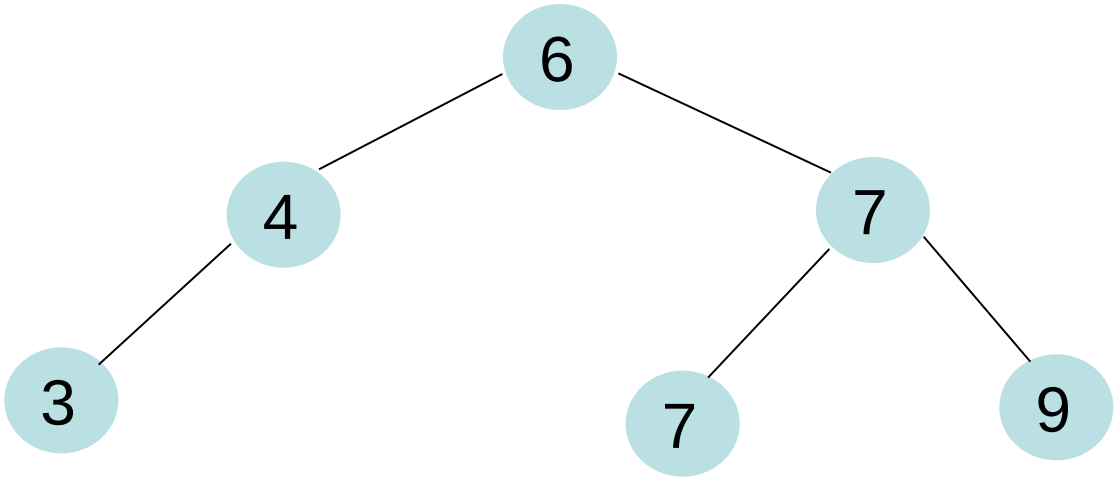
Funktionsweise wie (rekursive) Baumsuche. Laufzeit ebenfalls O(h).



Datenstrukturen

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]

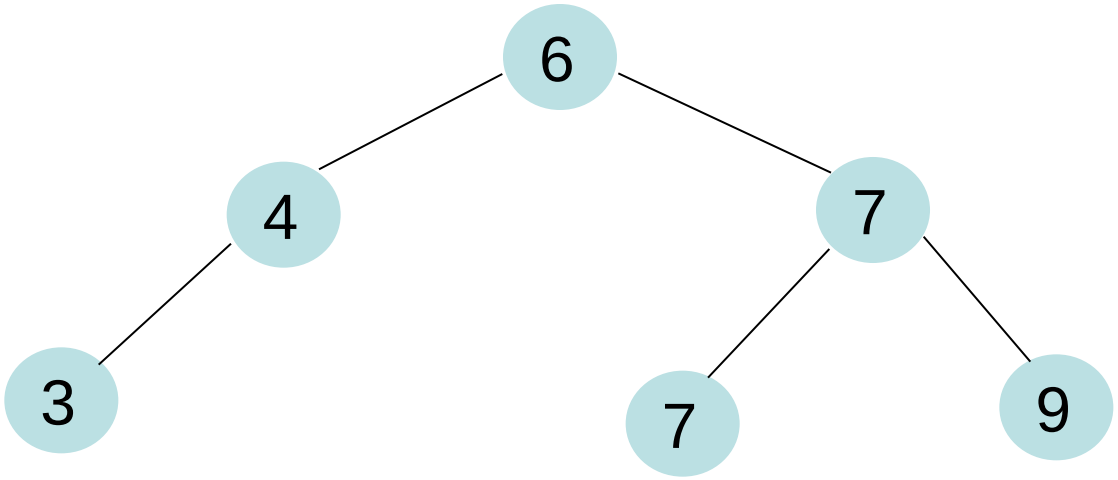


Datenstrukturen

Wird mit Wurzel aufgerufen

MinimumSuche(x)

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



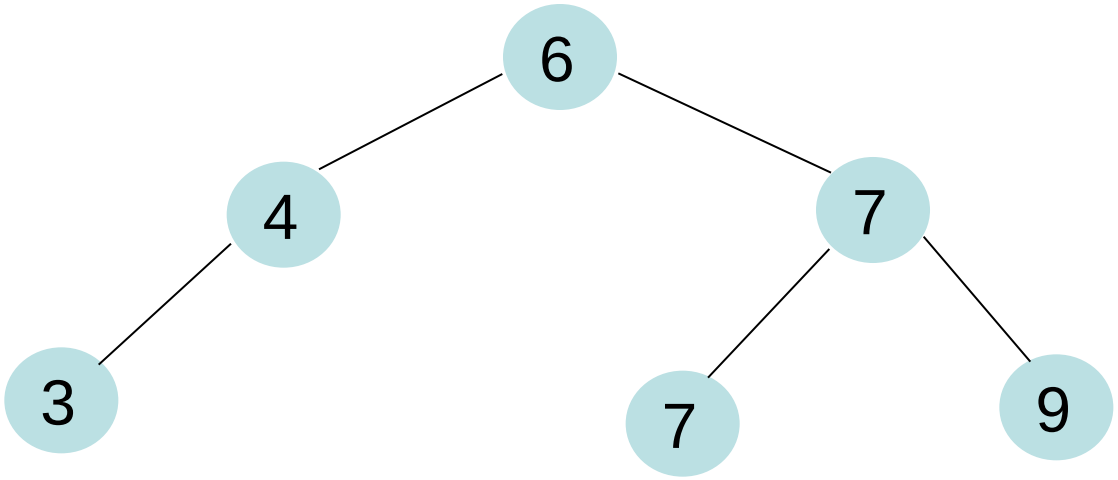
Datenstrukturen

Wird mit Wurzel aufgerufen

Laufzeit O(h)

MinimumSuche(x)

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



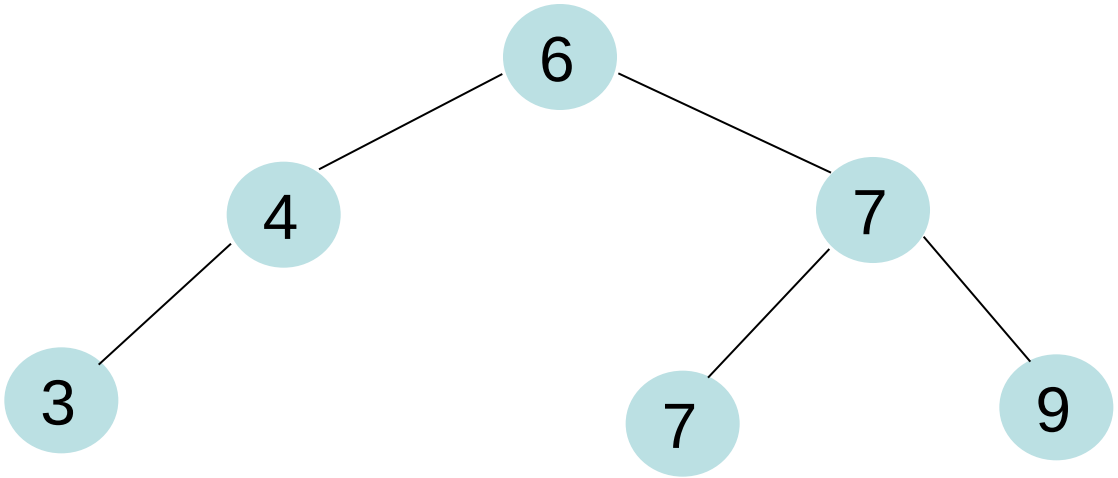
Datenstrukturen

Wird mit Wurzel aufgerufen

Laufzeit O(h)

MaximumSuche(x)

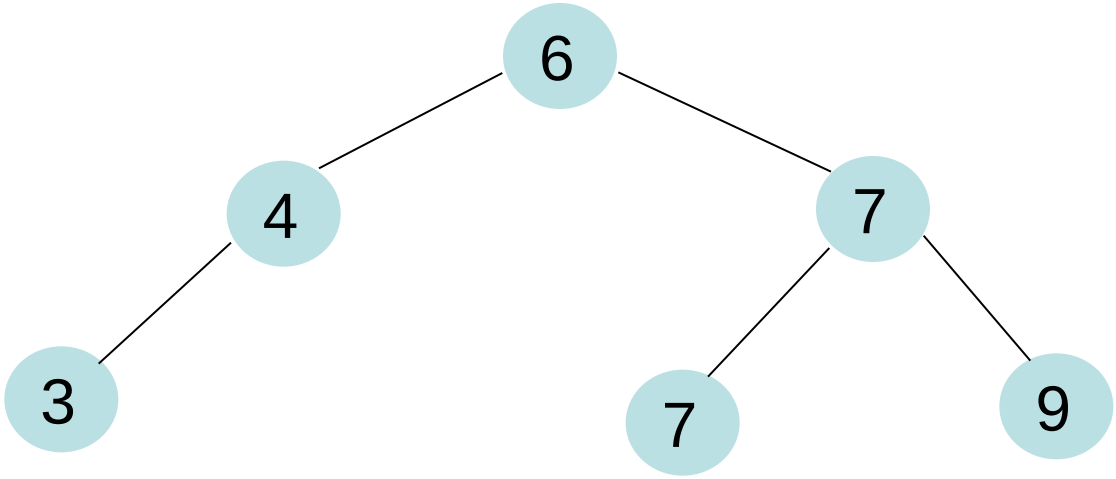
1. **while** rc[x]≠**nil do** x ← rc[x]
2. **return** x



Datenstrukturen

Nachfolgersuche

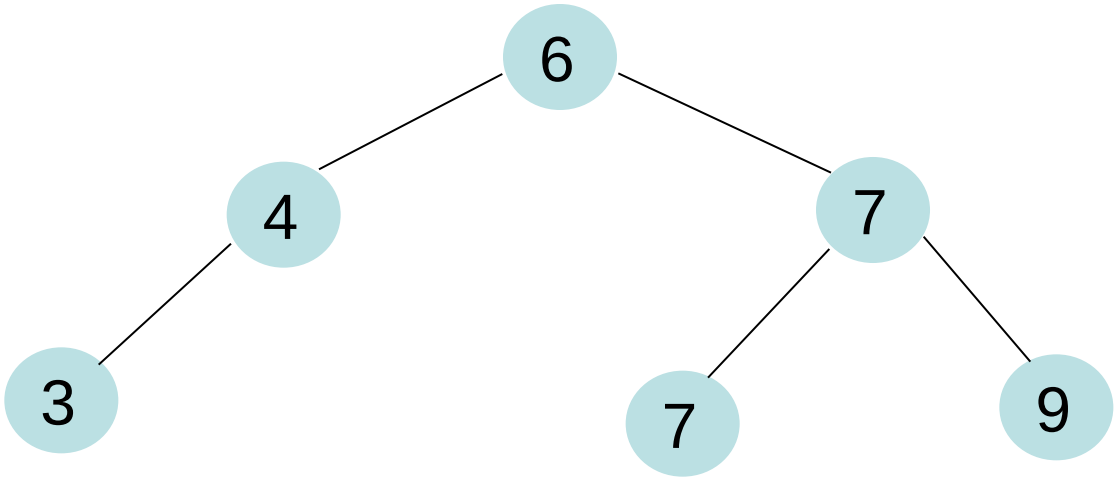
- Nachfolger bzgl. Inorder-Tree-Walk
- Wenn alle Schlüssel unterschiedlich, dann ist das der nächstgrößere Schlüssel



Datenstrukturen

Nachfolgersuche

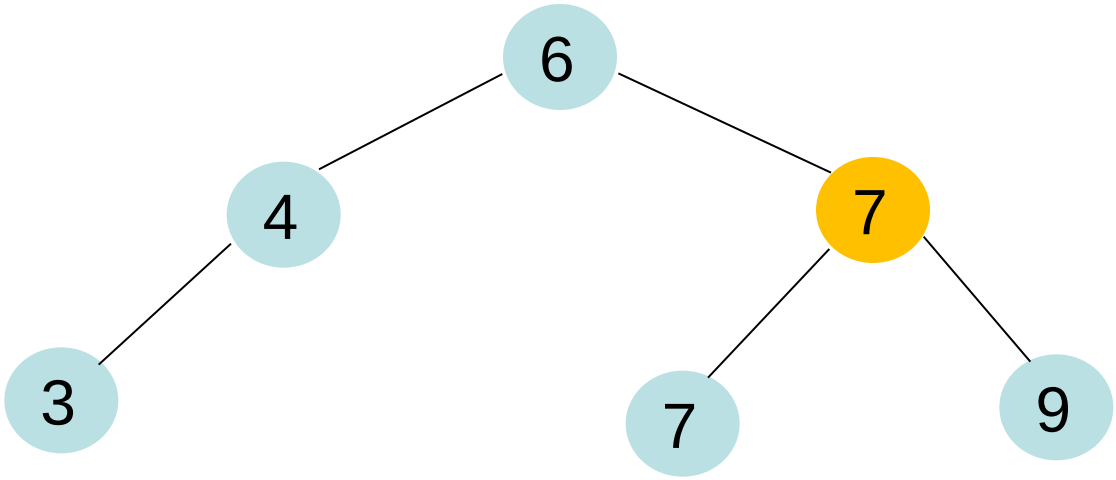
- Fall 1 (rechter Unterbaum von x nicht leer):
  Dann ist der linkeste Knoten im rechten Unterbaum der Nachfolger von x



Datenstrukturen

Nachfolgersuche

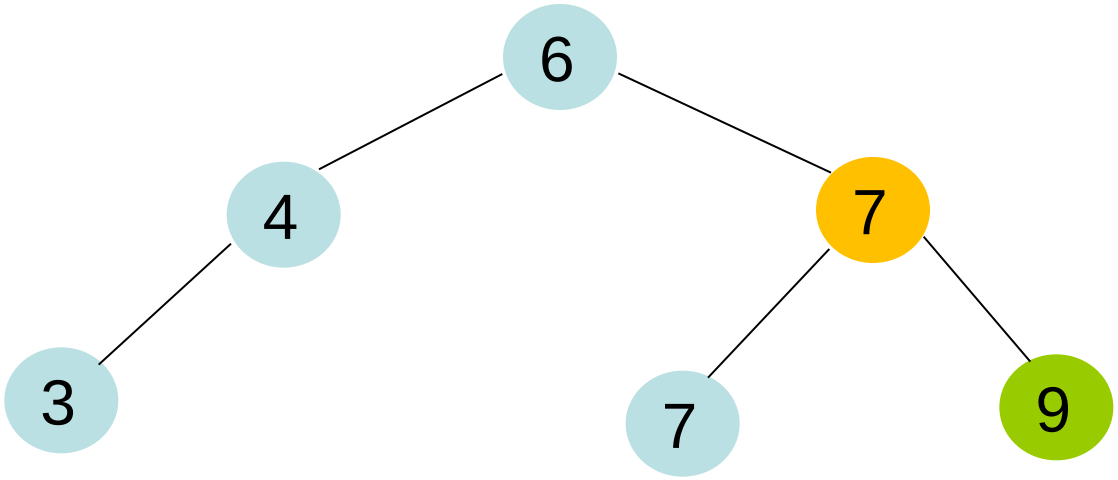
- Fall 1 (rechter Unterbaum von x nicht leer):
  Dann ist der linkeste Knoten im rechten Unterbaum der Nachfolger von x



Datenstrukturen

Nachfolgersuche

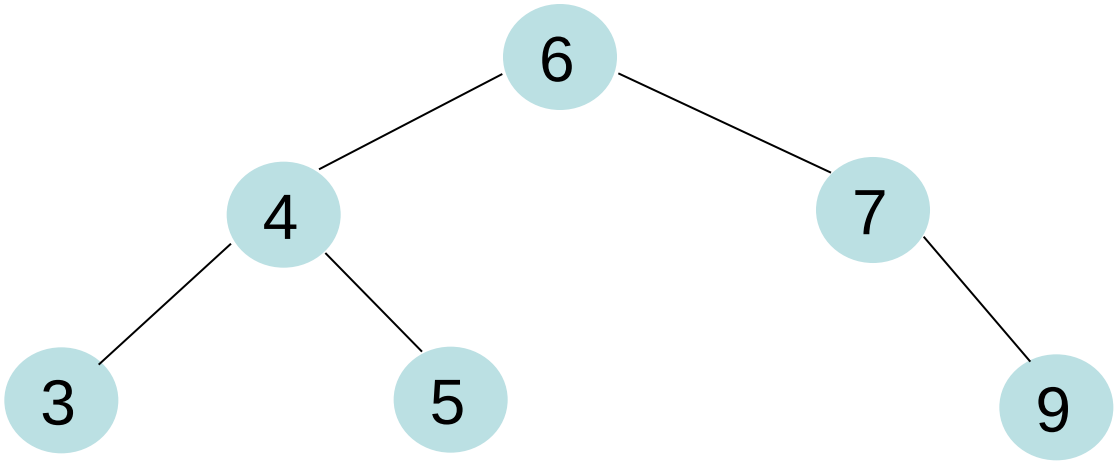
- Fall 1 (rechter Unterbaum von x nicht leer):
  Dann ist der linkeste Knoten im rechten Unterbaum der Nachfolger von x



Datenstrukturen

Nachfolgersuche

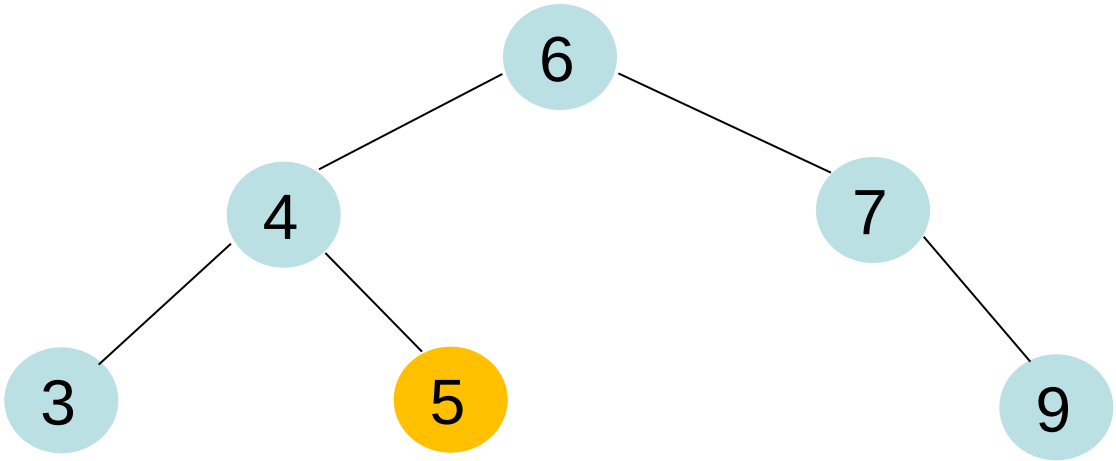
- Fall 2 (rechter Unterbaum von x leer und x hat Nachfolger y):
  Dann ist y der erste Knoten auf dem Pfad zur Wurzel, der größer als x ist



Datenstrukturen

Nachfolgersuche

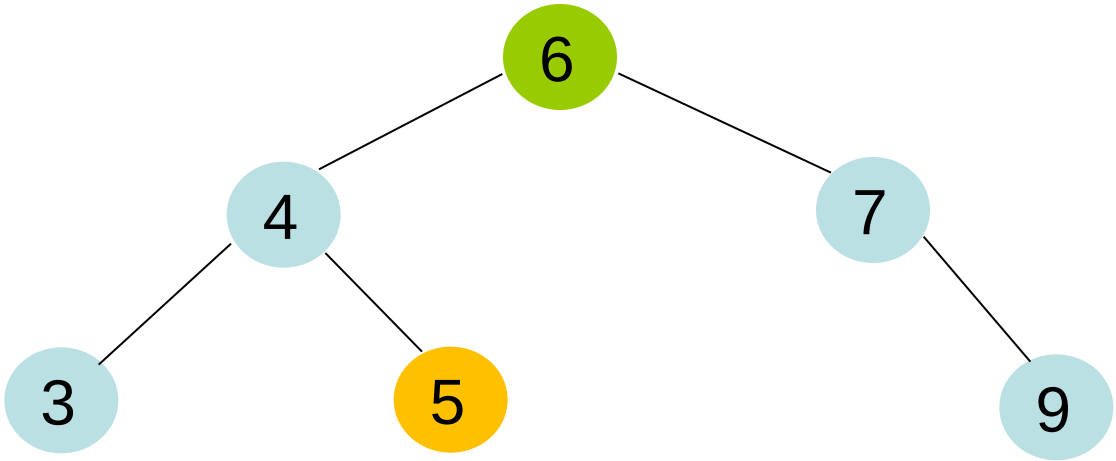
- Fall 2 (rechter Unterbaum von x leer und x hat Nachfolger y):
  Dann ist y der erste Knoten auf dem Pfad zur Wurzel, der größer als x ist



Datenstrukturen

Nachfolgersuche

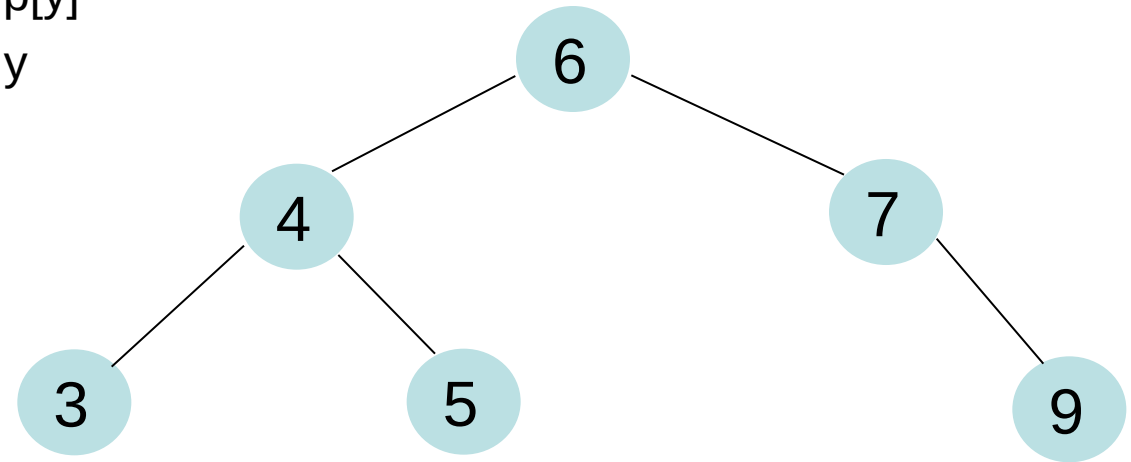
- Fall 2 (rechter Unterbaum von x leer und x hat Nachfolger y):
  Dann ist y der erste Knoten auf dem Pfad zur Wurzel, der größer als x ist



Datenstrukturen

Nachfolgersuche(x)

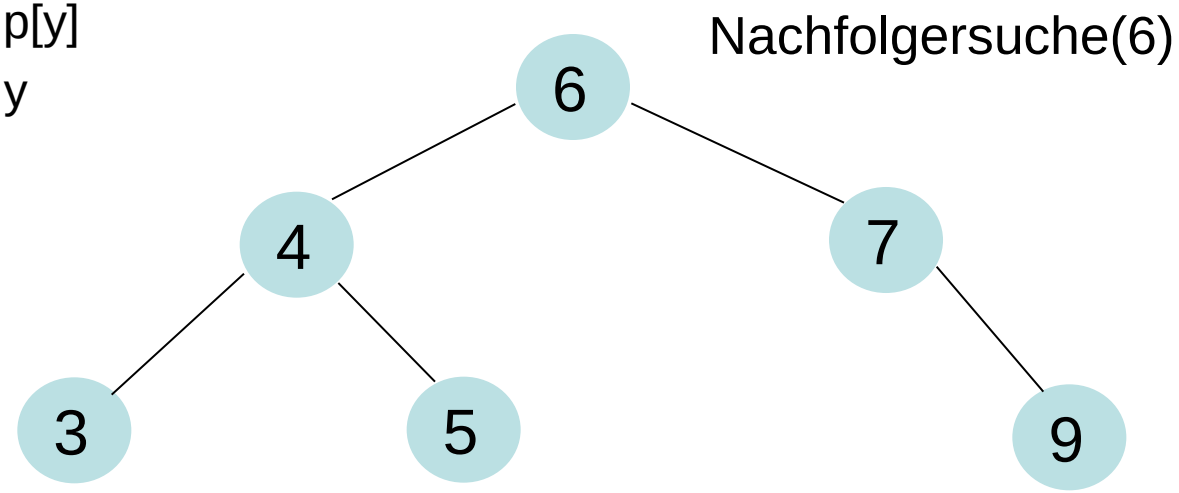
1. **if** rc[x] ≠ **nil then return** MinimumSuche(rc[x])
2. y ← p[x]
3. **while** y ≠ **nil** and x=rc[y] **do**
4. x ← y
5. y ← p[y]
6. **return** y



Datenstrukturen

Nachfolgersuche(x)

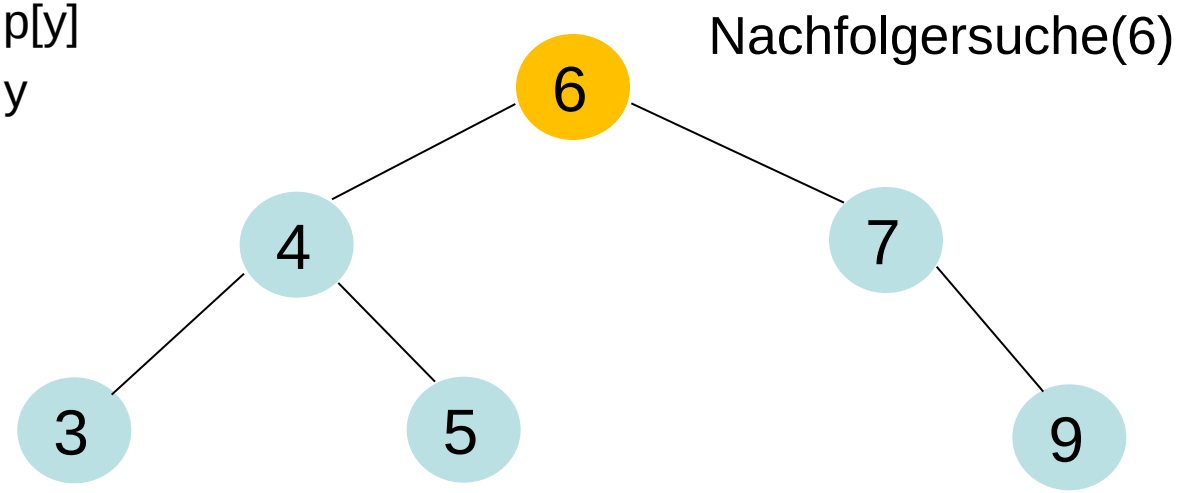
1. **if** rc[x] ≠ **nil then return** MinimumSuche(rc[x])
2. y ← p[x]
3. **while** y ≠ **nil** and x=rc[y] **do**
4. x ← y
5. y ← p[y]
6. **return** y



Datenstrukturen

Nachfolgersuche(x)

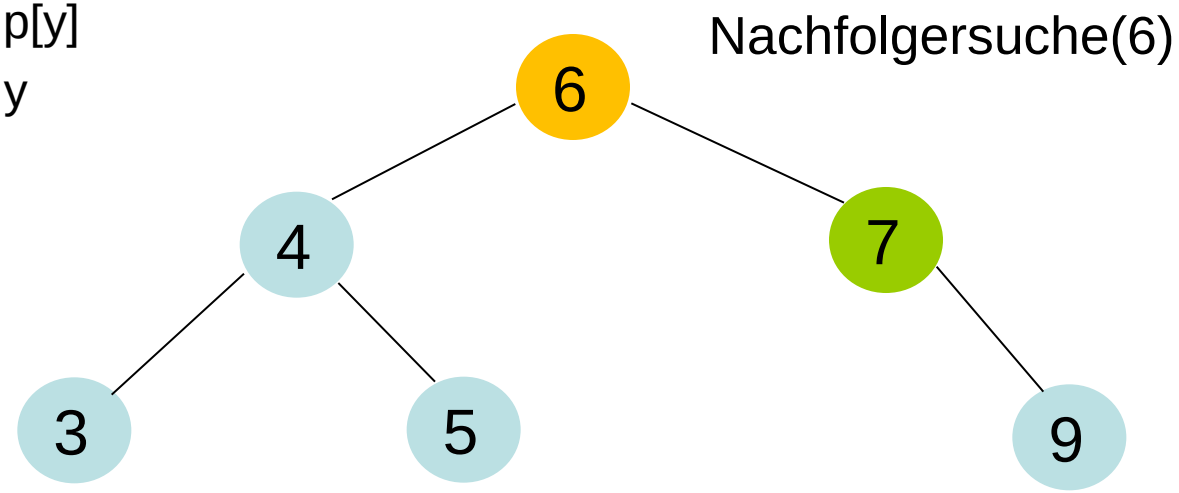
1. **if** rc[x] ≠ **nil then return** MinimumSuche(rc[x])
2. y ← p[x]
3. **while** y ≠ **nil** and x=rc[y] **do**
4. x ← y
5. y ← p[y]
6. **return** y



Datenstrukturen

Nachfolgersuche(x)

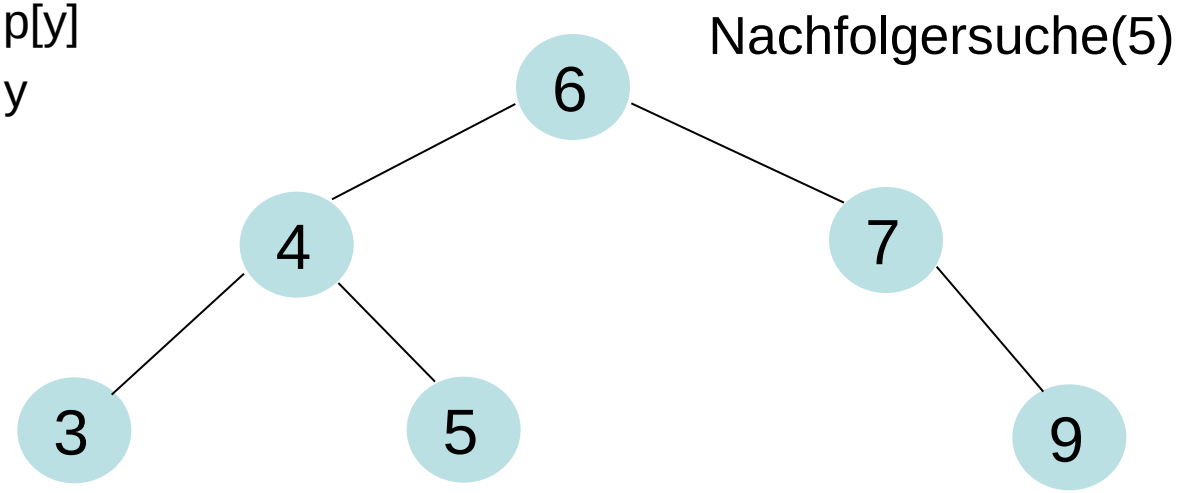
1. **if** rc[x] ≠ **nil then return** MinimumSuche(rc[x])
2. y ← p[x]
3. **while** y ≠ **nil** and x=rc[y] **do**
4. x ← y
5. y ← p[y]
6. **return** y



Datenstrukturen

Nachfolgersuche(x)

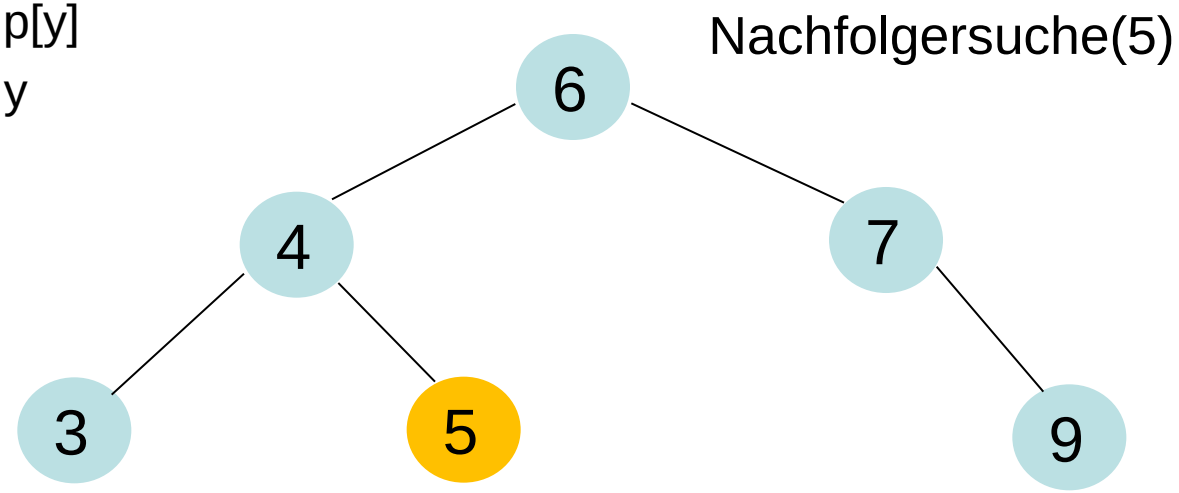
1. **if** rc[x] ≠ **nil then return** MinimumSuche(rc[x])
2. y ← p[x]
3. **while** y ≠ **nil** and x=rc[y] **do**
4. x ← y
5. y ← p[y]
6. **return** y



Datenstrukturen

Nachfolgersuche(x)

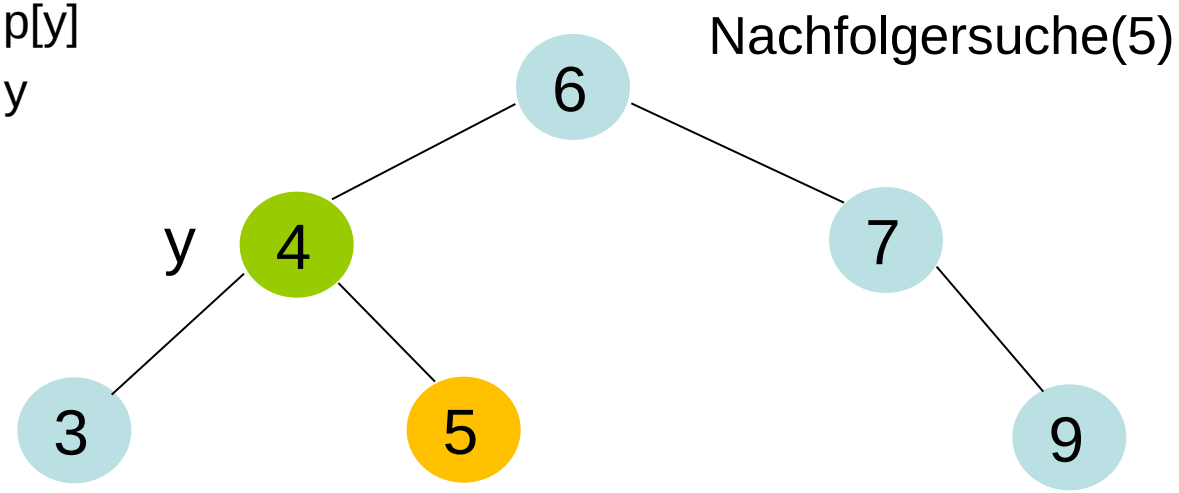
1. **if** rc[x] ≠ **nil then return** MinimumSuche(rc[x])
2. y ← p[x]
3. **while** y ≠ **nil** and x=rc[y] **do**
4. x ← y
5. y ← p[y]
6. **return** y



Datenstrukturen

Nachfolgersuche(x)

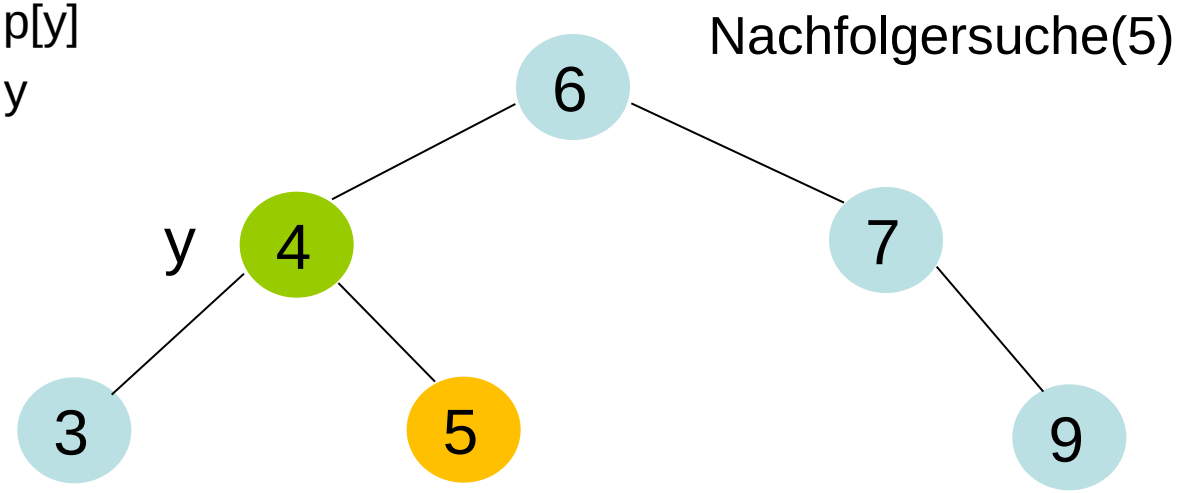
1. **if** rc[x] ≠ **nil then return** MinimumSuche(rc[x])
2. y ← p[x]
3. **while** y ≠ **nil** and x=rc[y] **do**
4. x ← y
5. y ← p[y]
6. **return** y



Datenstrukturen

Nachfolgersuche(x)

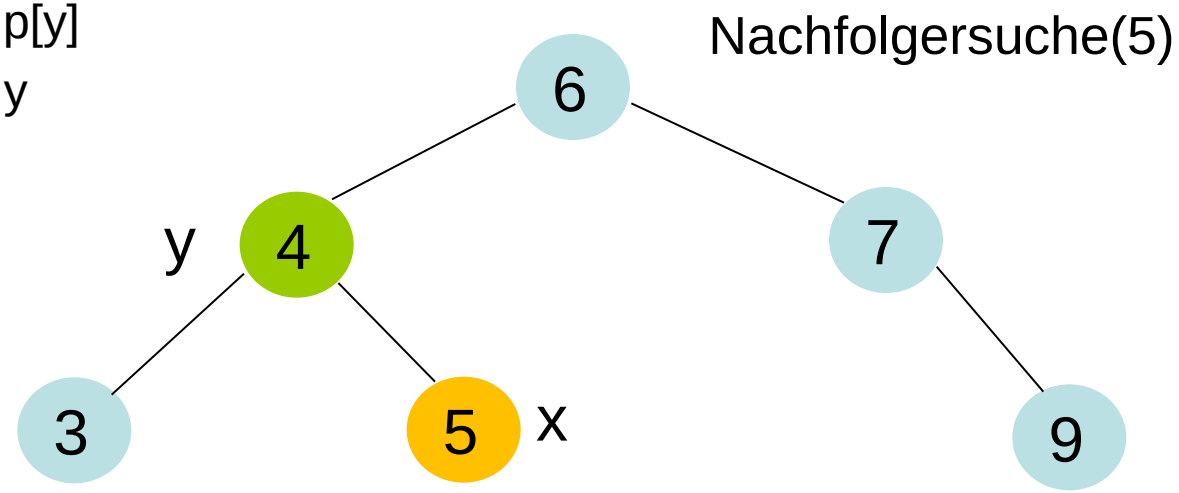
1. **if** rc[x] ≠ **nil then return** MinimumSuche(rc[x])
2. y ← p[x]
3. **while** y ≠ **nil** and x=rc[y] **do**
4. x ← y
5. y ← p[y]
6. **return** y



Datenstrukturen

Nachfolgersuche(x)

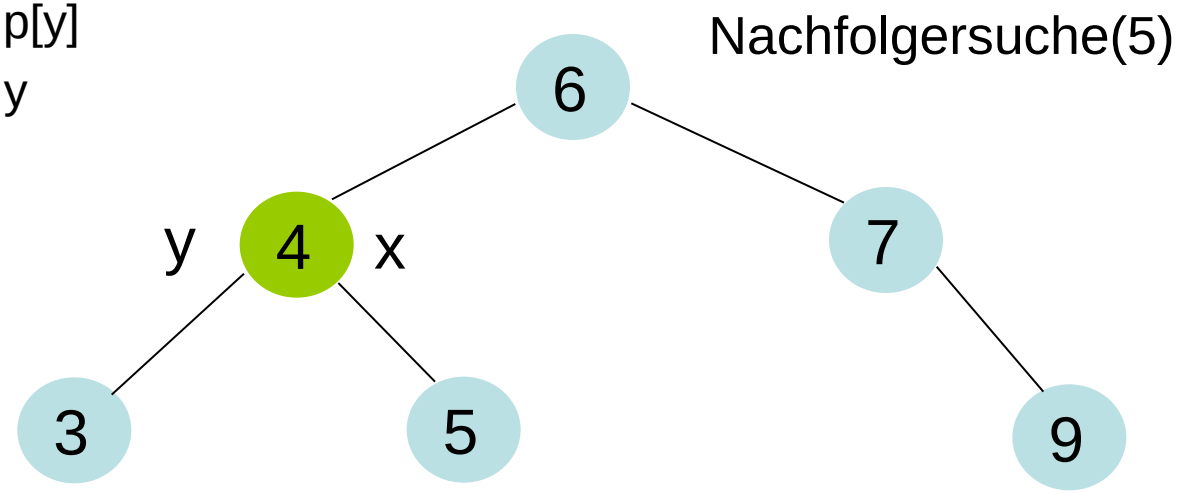
1. **if** rc[x] ≠ **nil then return** MinimumSuche(rc[x])
2. y ← p[x]
3. **while** y ≠ **nil** and x=rc[y] **do**
4. x ← y
5. y ← p[y]
6. **return** y



Datenstrukturen

Nachfolgersuche(x)

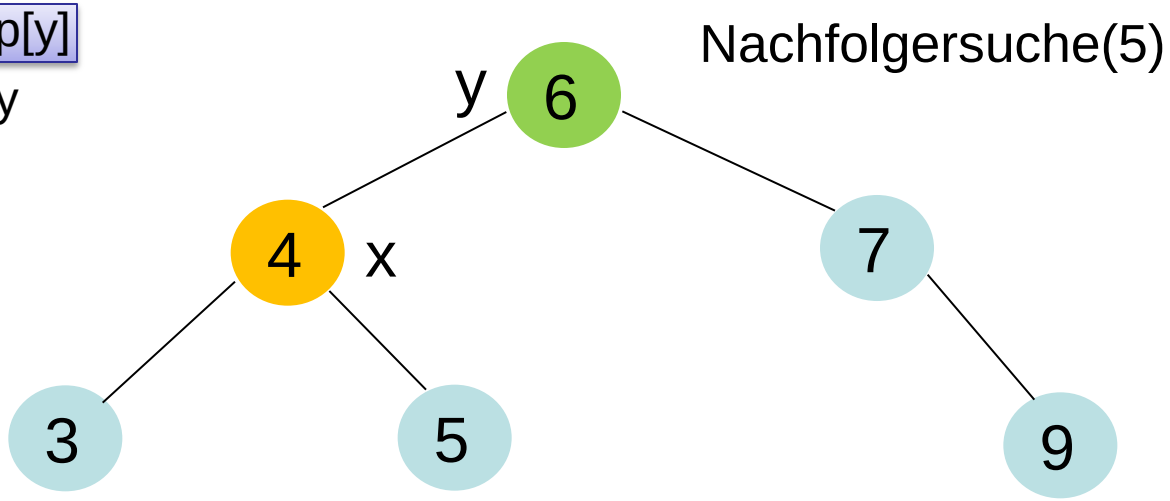
1. **if** rc[x] ≠ **nil then return** MinimumSuche(rc[x])
2. y ← p[x]
3. **while** y ≠ **nil** and x=rc[y] **do**
4. x ← y
5. y ← p[y]
6. **return** y



Datenstrukturen

Nachfolgersuche(x)

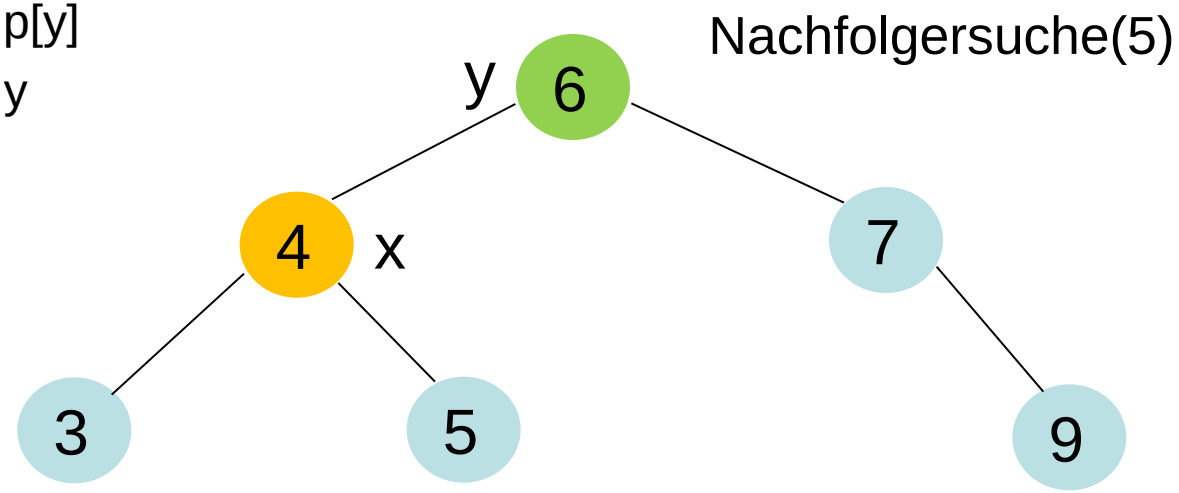
1. **if** rc[x] ≠ **nil then return** MinimumSuche(rc[x])
2. y ← p[x]
3. **while** y ≠ **nil** and x=rc[y] **do**
4. x ← y
5. y ← p[y]
6. **return** y



Datenstrukturen

Nachfolgersuche(x)

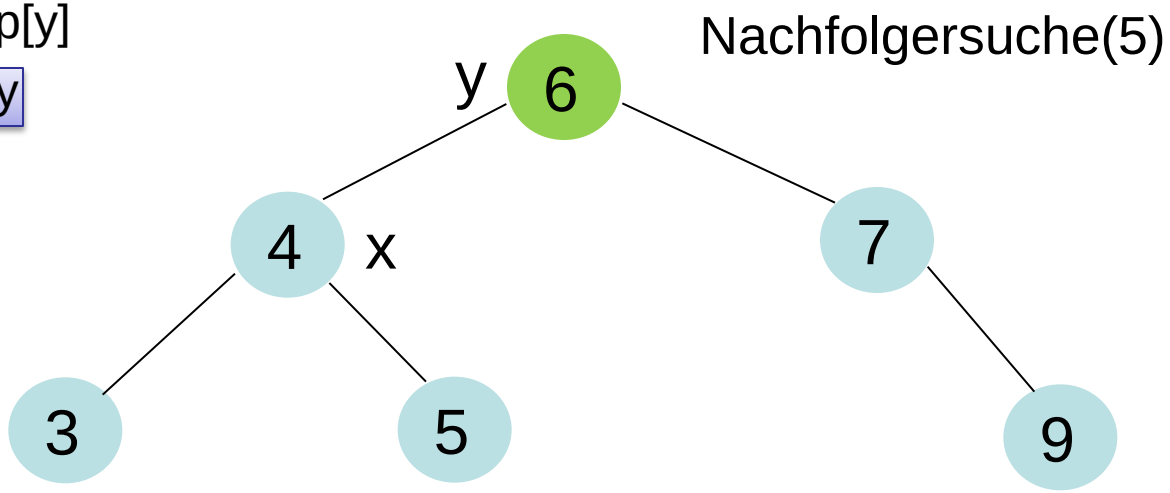
1. **if** rc[x] ≠ **nil then return** MinimumSuche(rc[x])
2. y ← p[x]
3. **while** y ≠ **nil** and x=rc[y] **do**
4. x ← y
5. y ← p[y]
6. **return** y



Datenstrukturen

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

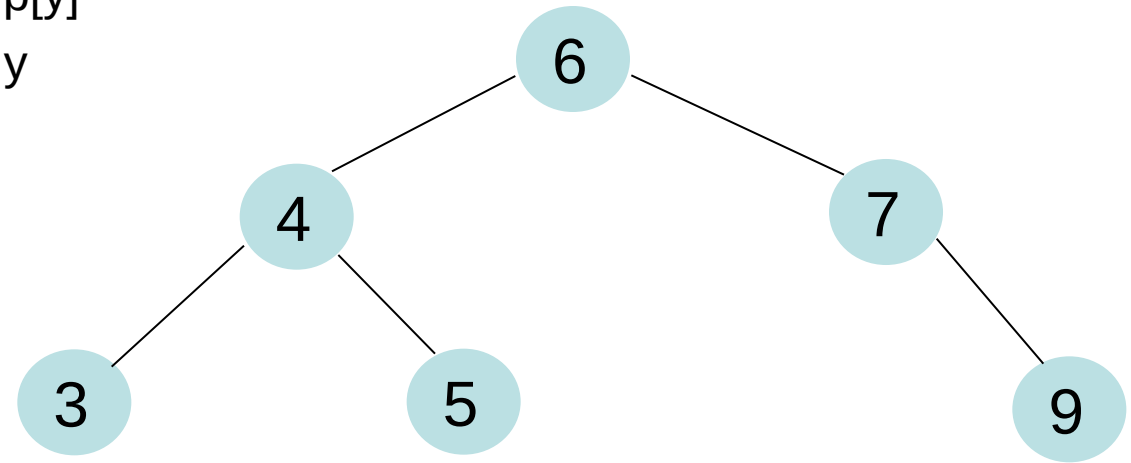


Datenstrukturen

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)

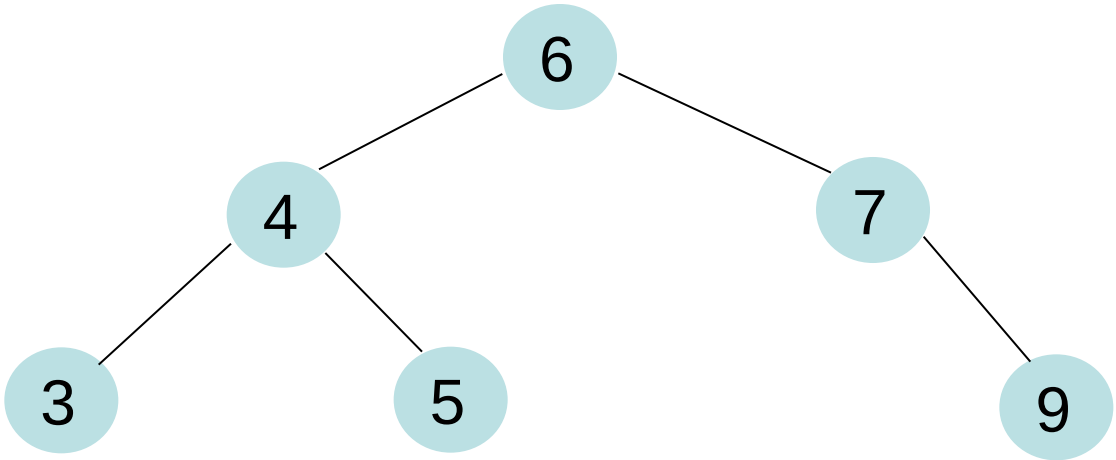


Datenstrukturen

Vorgängersuche

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

Laufzeit O(h)



Datenstrukturen

Binäre Suchbäume

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

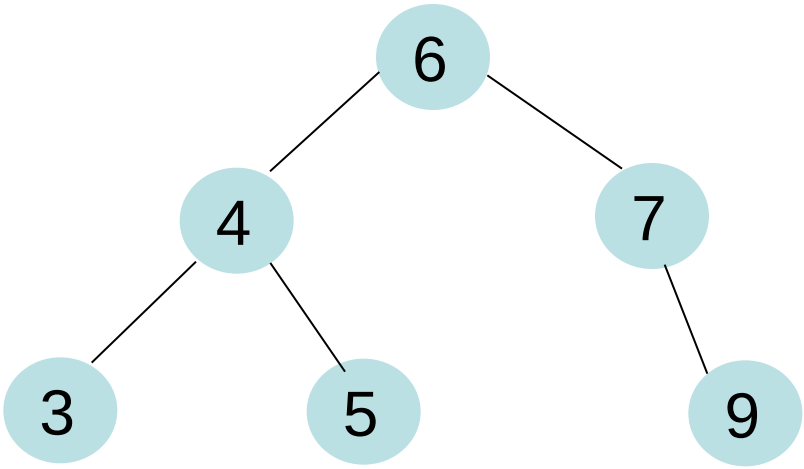
Dynamische Operationen?

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

Datenstrukturen

Einfügen

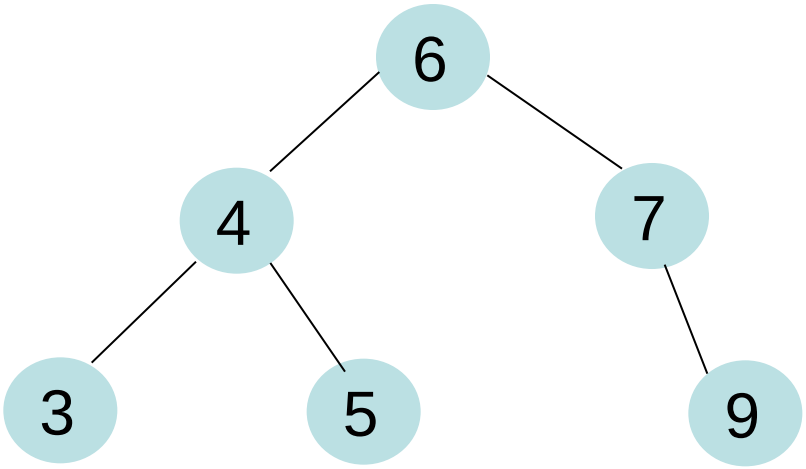
- Ähnlich wie Baumsuche: Finde Blatt, an das neuer Knoten angehängt wird
- Danach wird **nil**-Zeiger durch neues Element ersetzt



Datenstrukturen

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



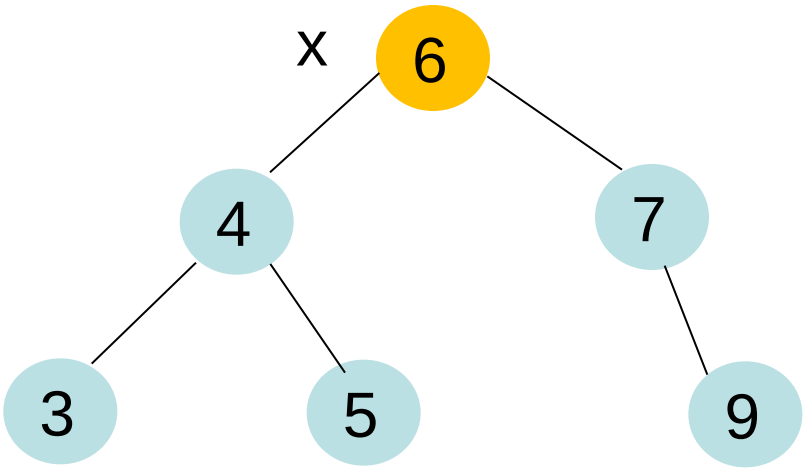
Datenstrukturen

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 wird Vater des einzufügenden Elements

Einfügen(8)

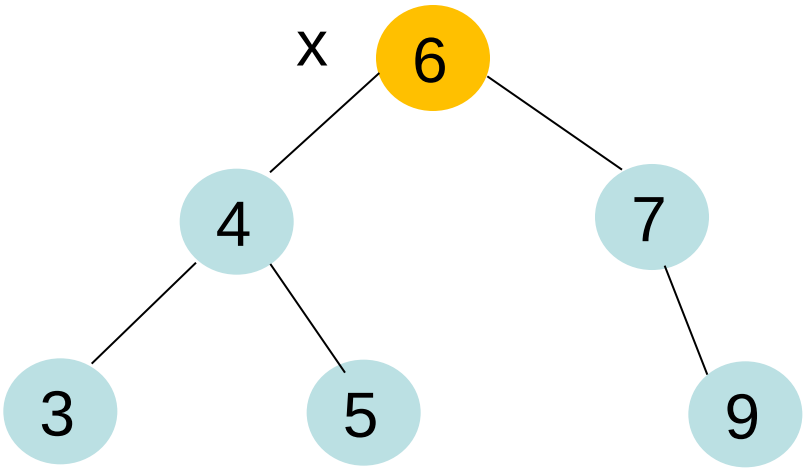


Datenstrukturen

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)

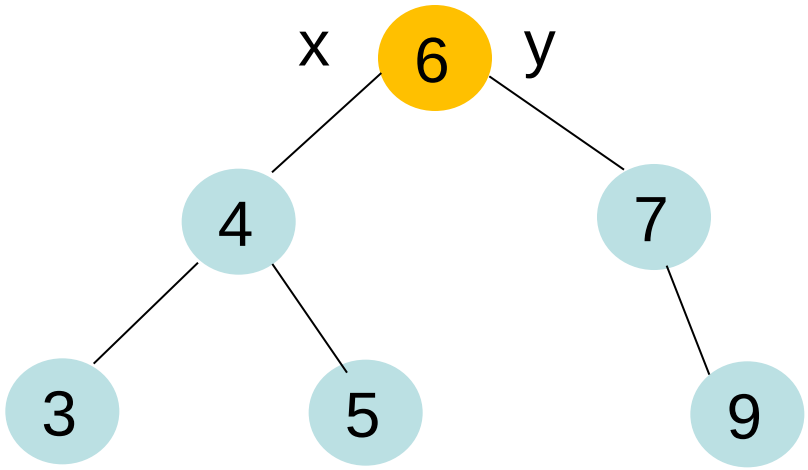


Datenstrukturen

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)

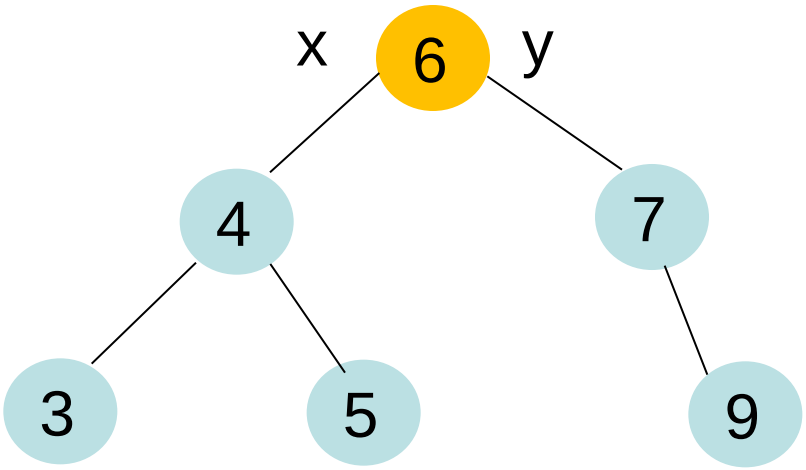


Datenstrukturen

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)

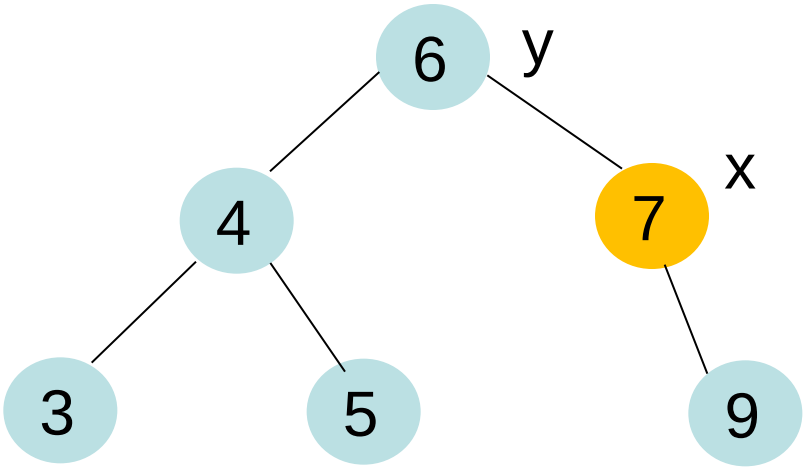


Datenstrukturen

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)

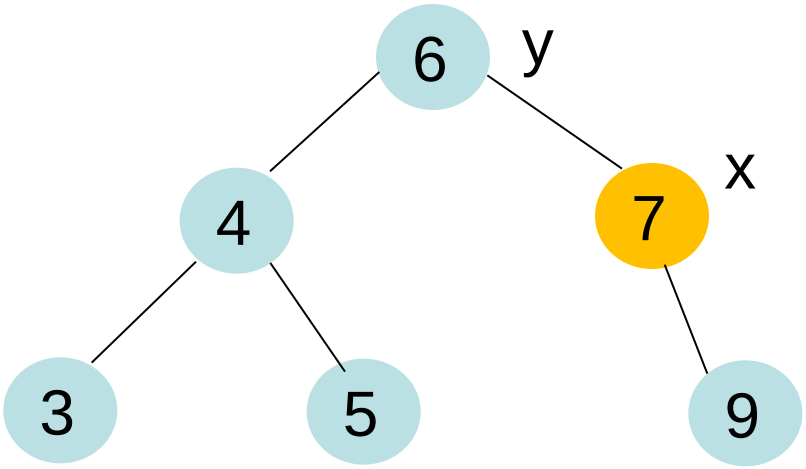


Datenstrukturen

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)

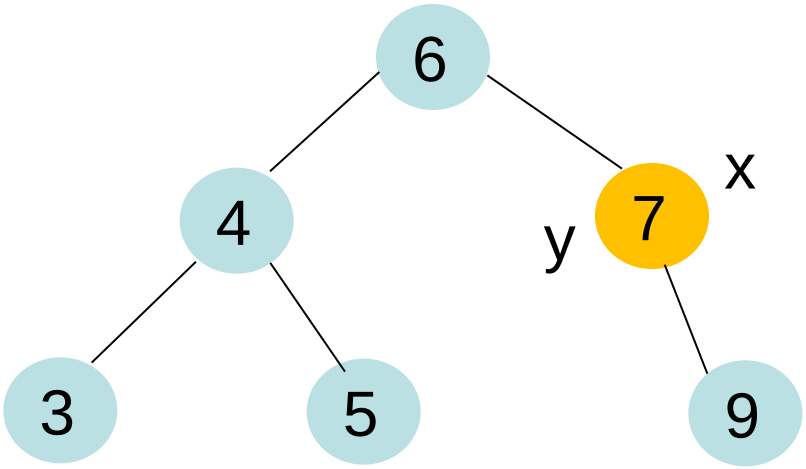


Datenstrukturen

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)

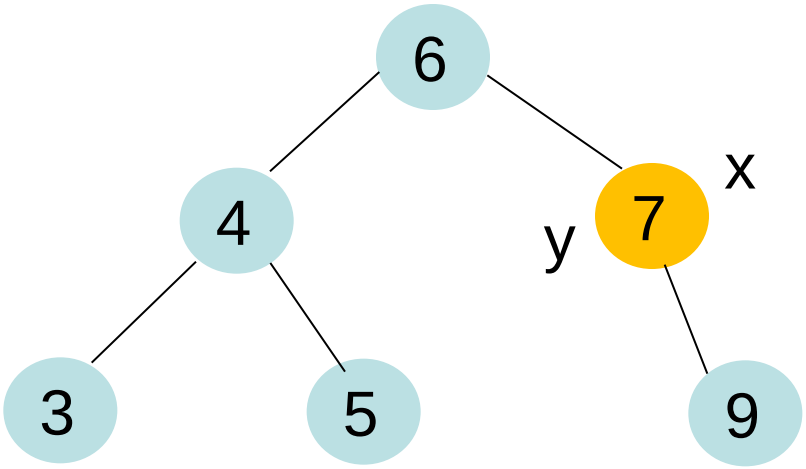


Datenstrukturen

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)

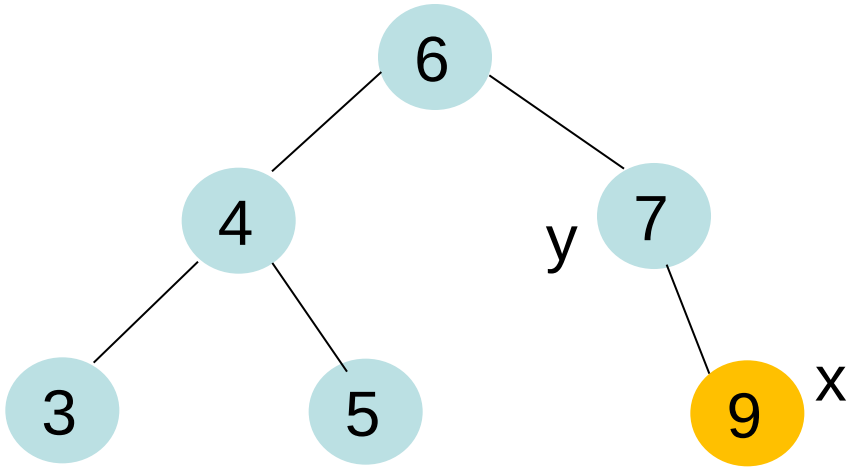


Datenstrukturen

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)

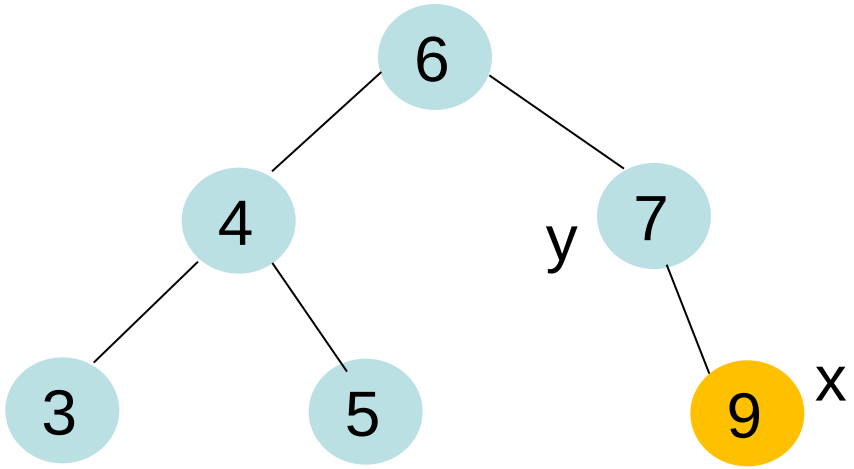


Datenstrukturen

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)

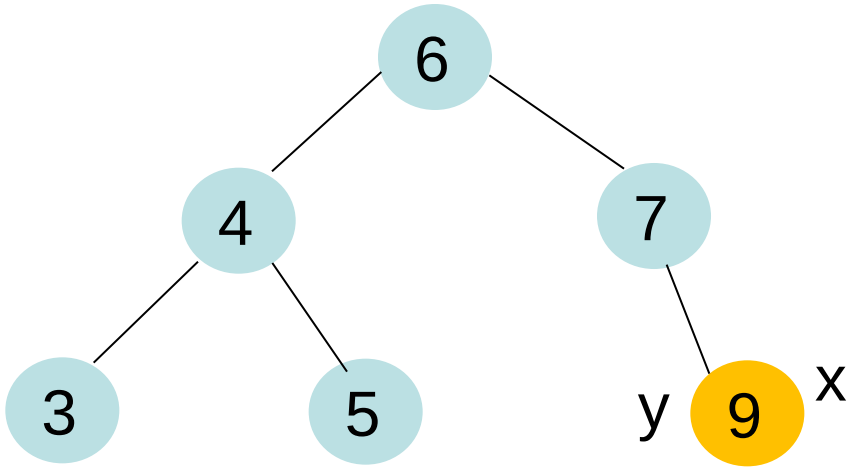


Datenstrukturen

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)

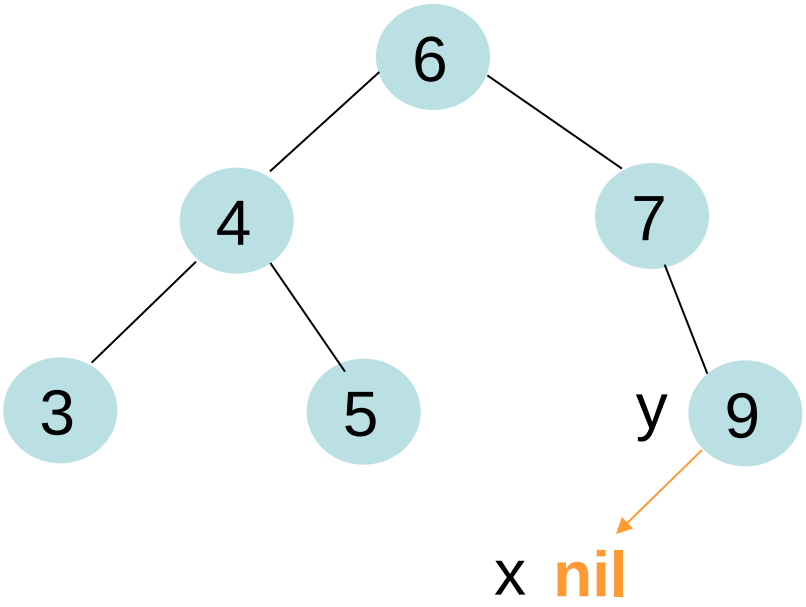


Datenstrukturen

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)

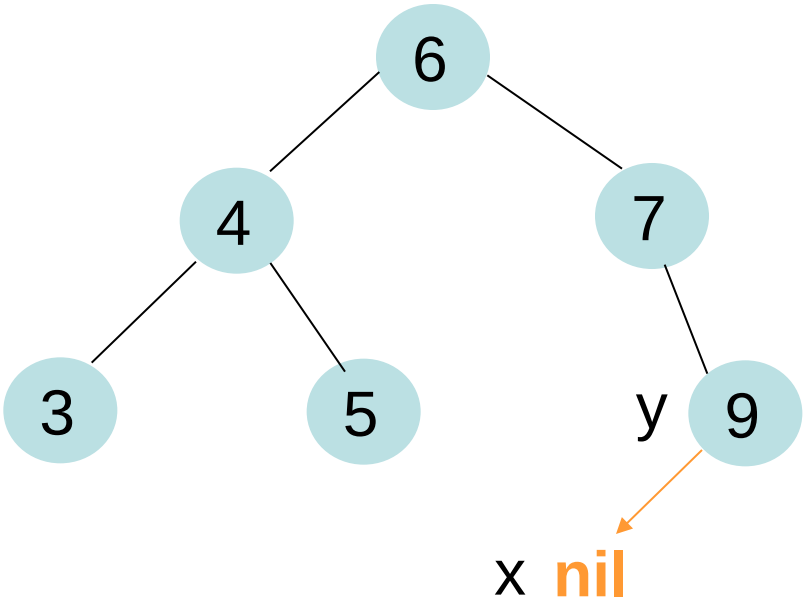


Datenstrukturen

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)

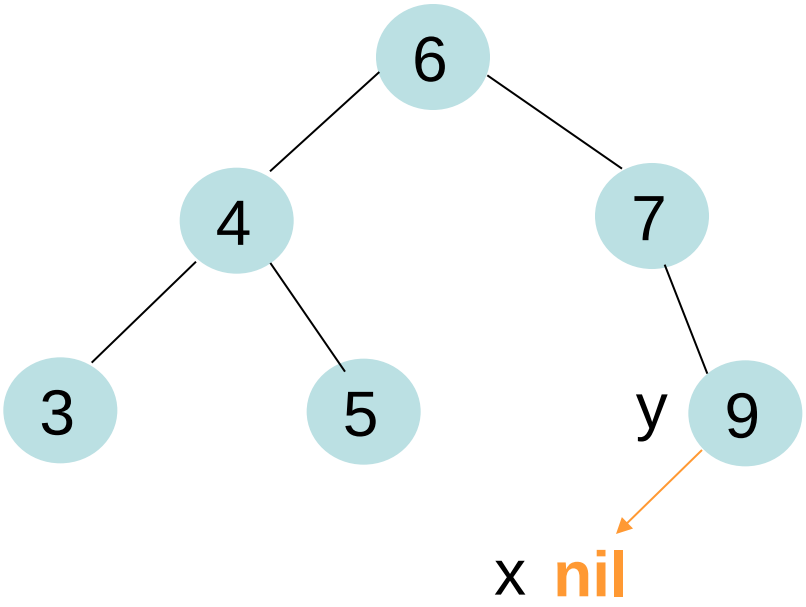


Datenstrukturen

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)

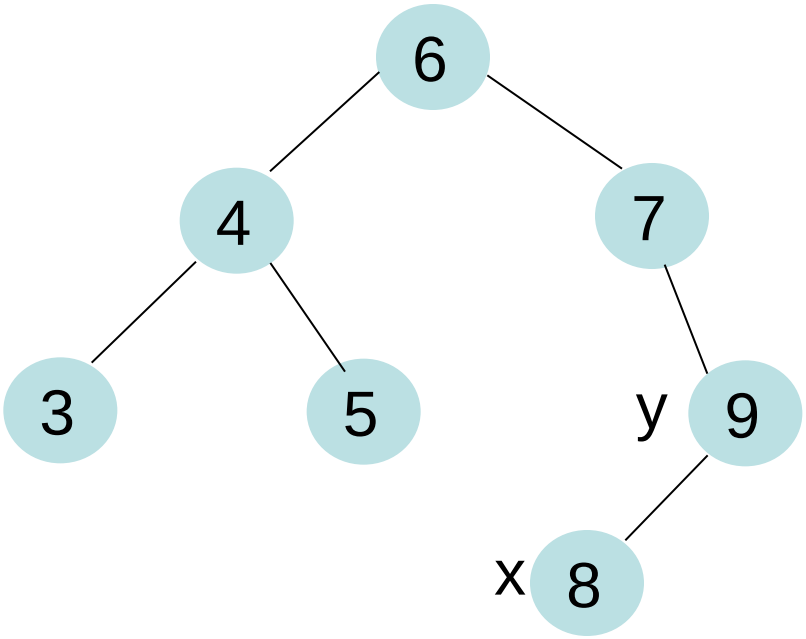


Datenstrukturen

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)

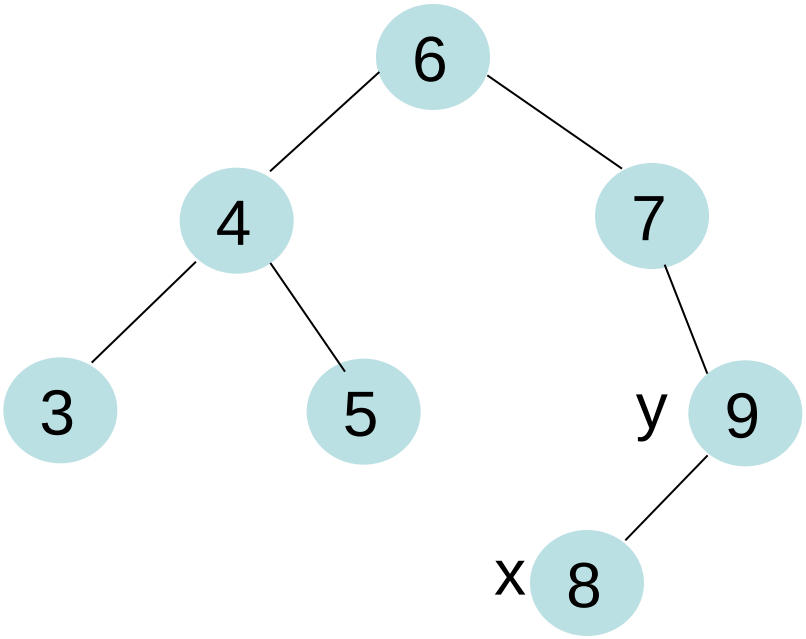


Datenstrukturen

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)

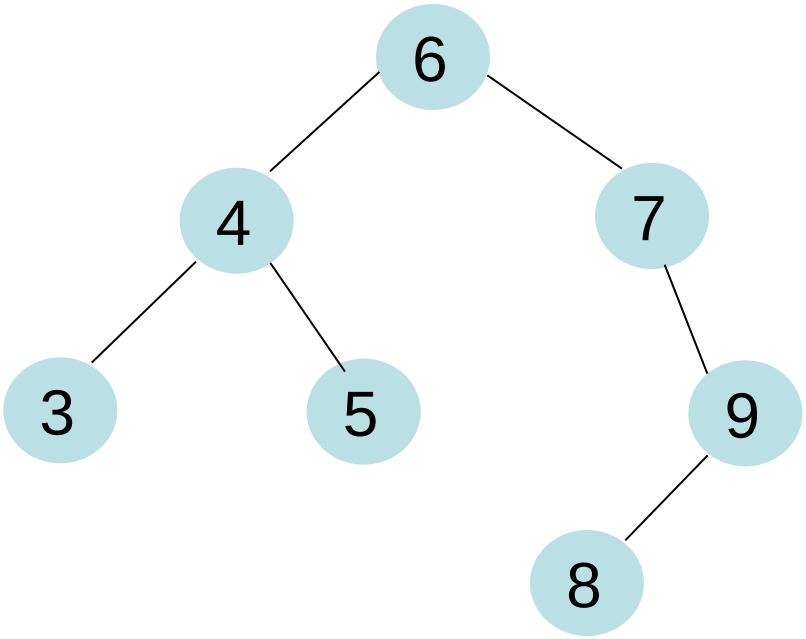


Datenstrukturen

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

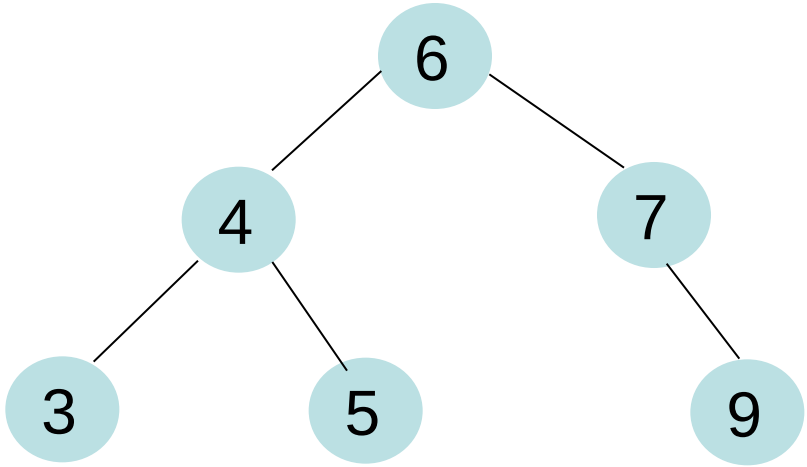
Laufzeit O(h)



Datenstrukturen

Löschen

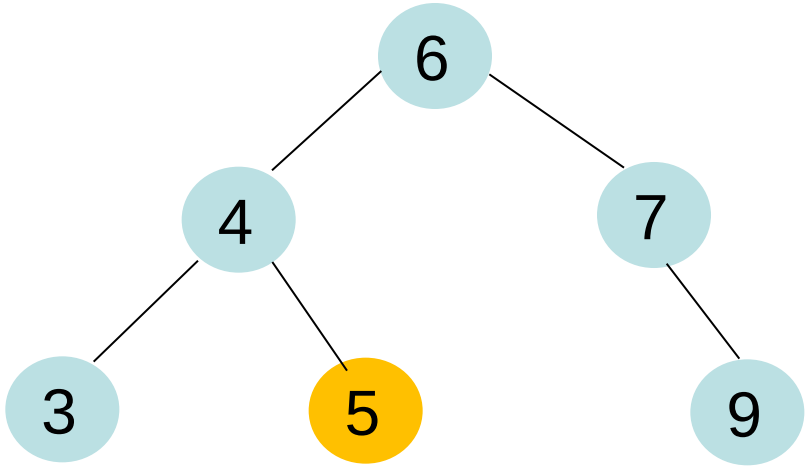
- 3 unterschiedliche Fälle
- (a) zu löschendes Element z hat keine Kinder
- (b) zu löschendes Element z hat ein Kind
- (c) zu löschendes Element z hat zwei Kinder



Datenstrukturen

Fall (a)

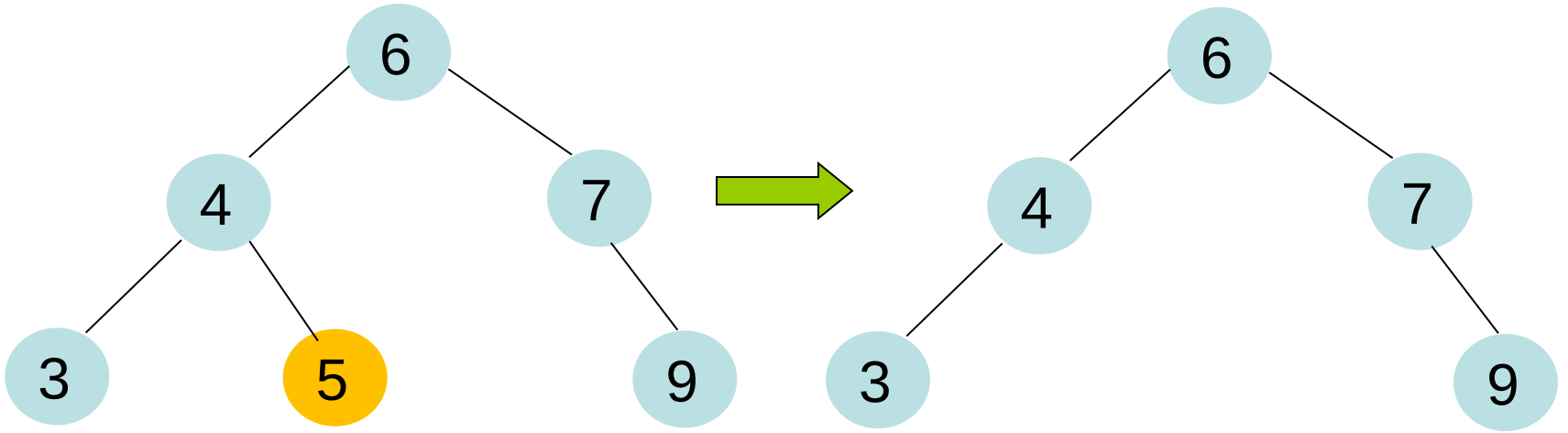
- zu löschendes Element z hat keine Kinder



Datenstrukturen

Fall (a)

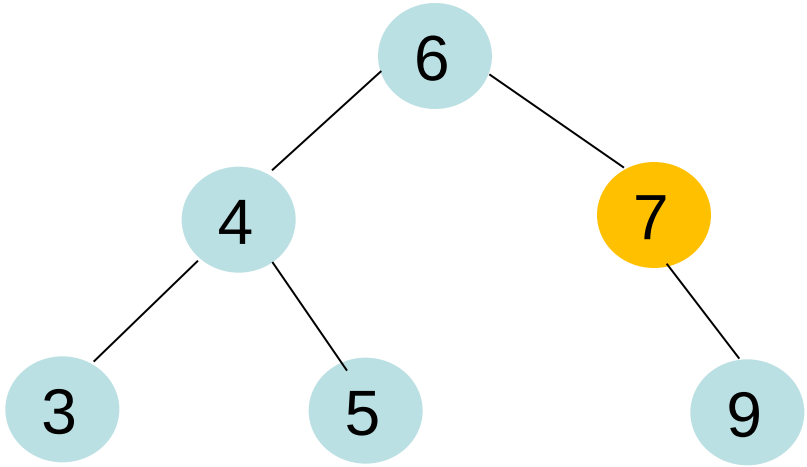
- zu löschendes Element z hat keine Kinder
- Entferne Element



Datenstrukturen

Fall (b)

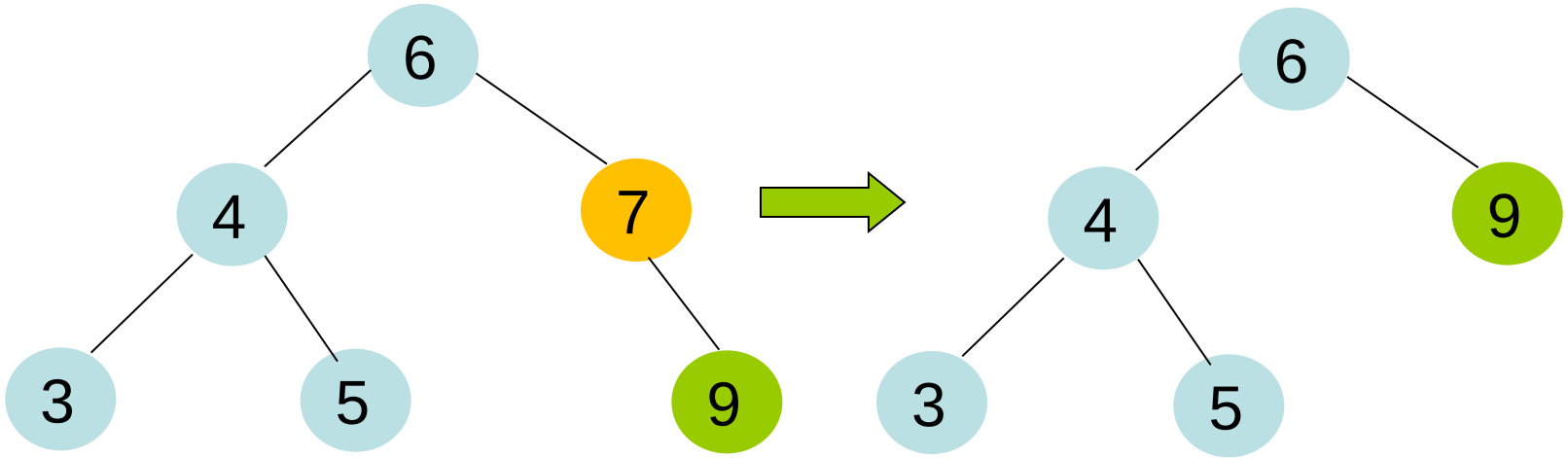
- Zu löschendes Element z hat 1 Kind



Datenstrukturen

Fall (b)

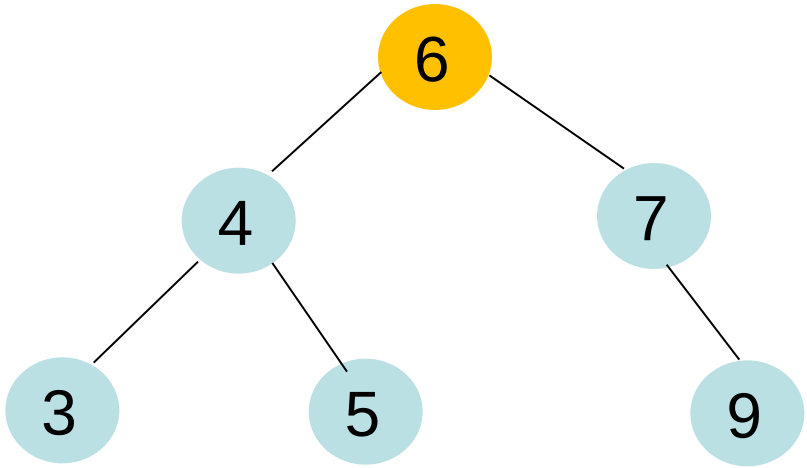
- Zu löschendes Element z hat 1 Kind



Datenstrukturen

Fall (c)

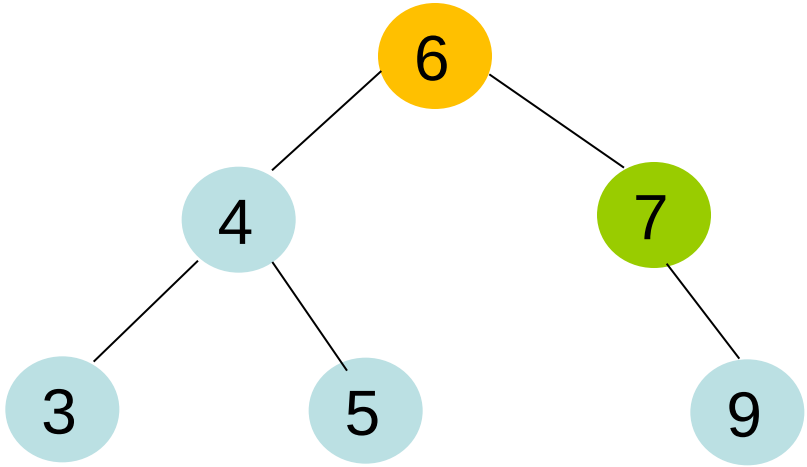
- Zu löschendes Element z hat 2 Kinder



Datenstrukturen

Fall (c)

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

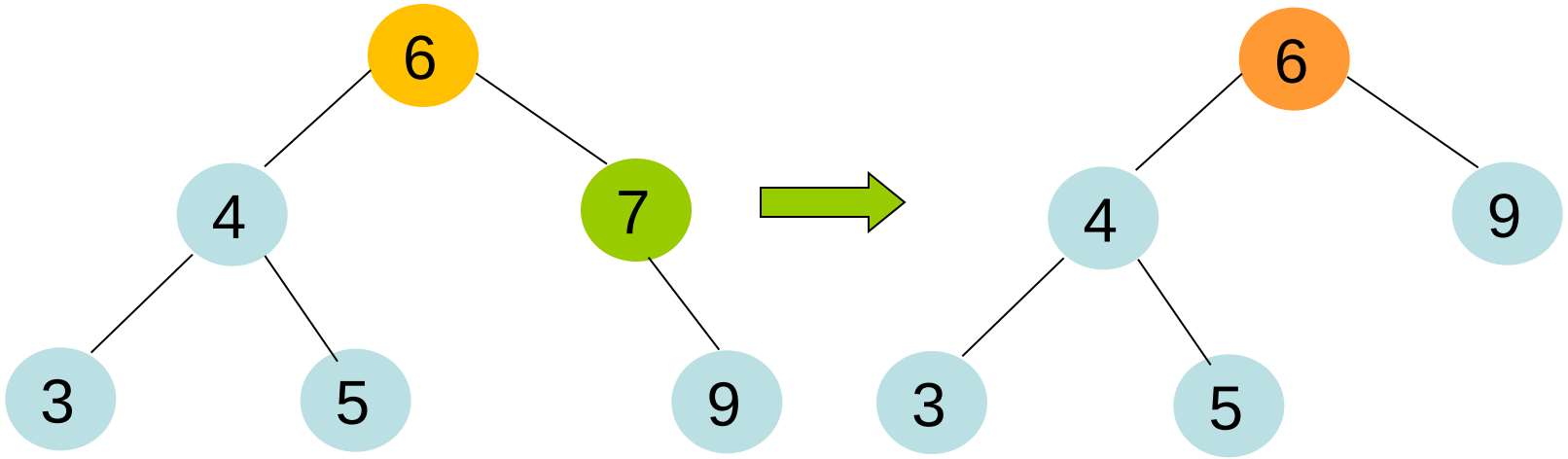


Datenstrukturen

Fall (c)

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

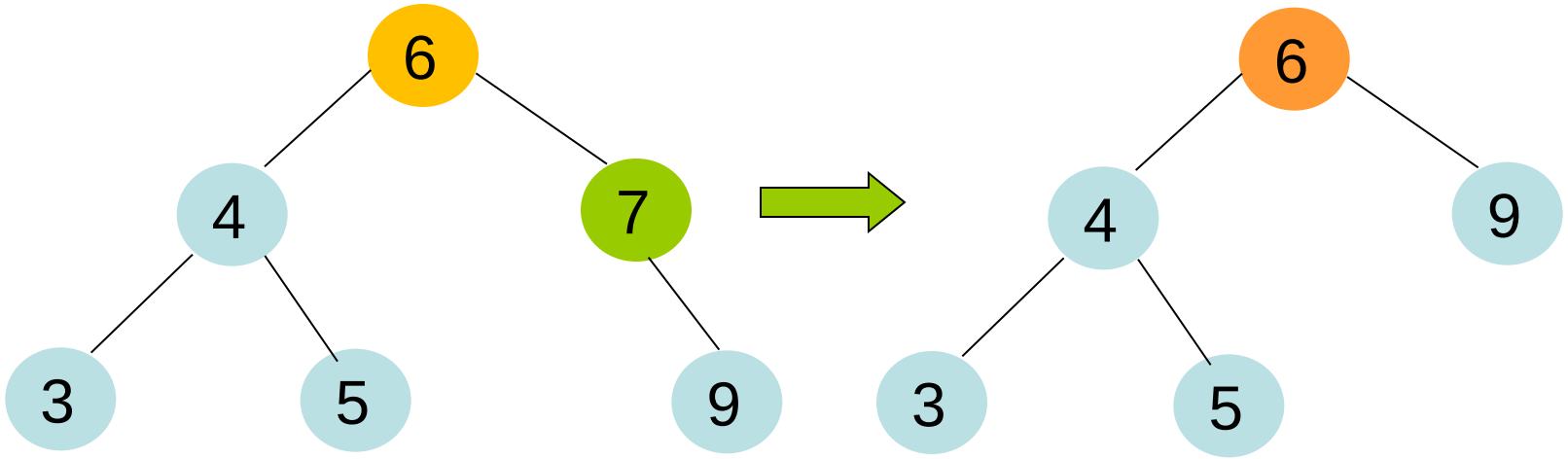
Nachfolger hat nur ein Kind



Datenstrukturen

Fall (c)

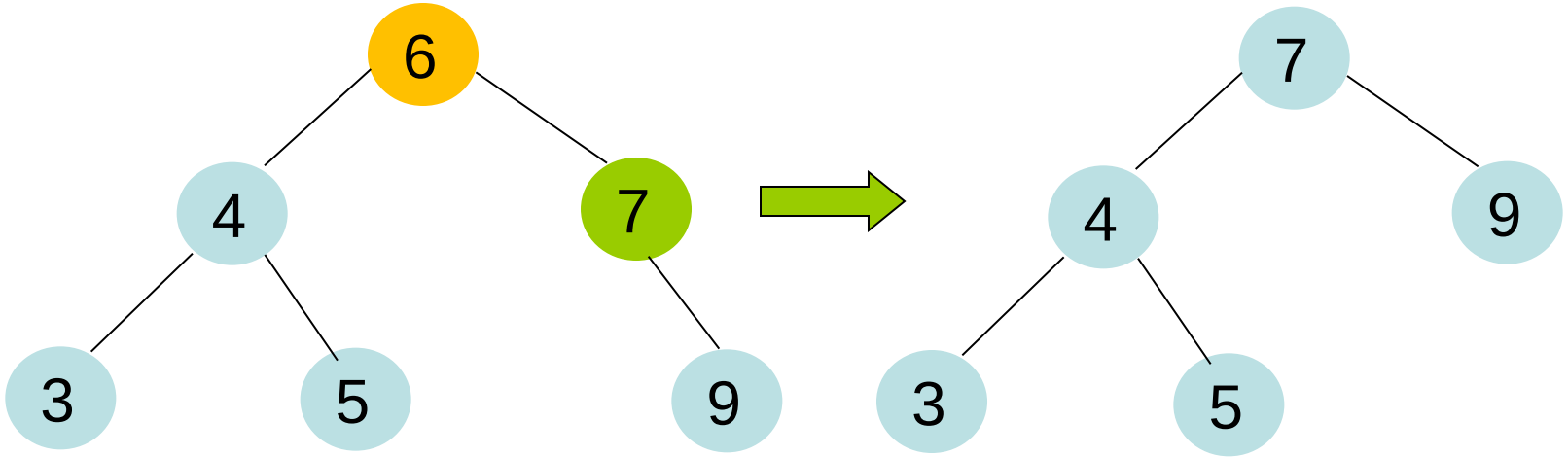
- Zu löschendes Element z hat 2 Kinder
- Schritt 1: Bestimme Nachfolger von z
- Schritt 2: Entferne Nachfolger



Datenstrukturen

Fall (c)

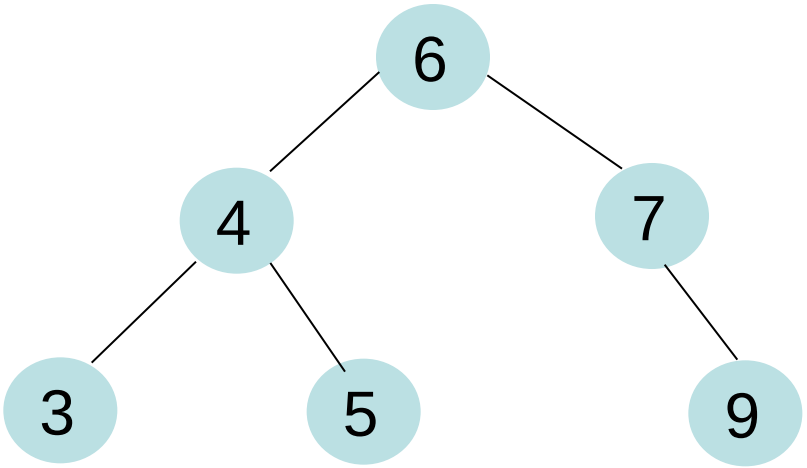
- Zu löschendes Element z hat 2 Kinder
- Schritt 1: Bestimme Nachfolger von z
- Schritt 2: Entferne Nachfolger
- Schritt 3: Ersetze z durch Nachfolger



Datenstrukturen

Löschen(T,z)

1. **if** lc[z]=**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]



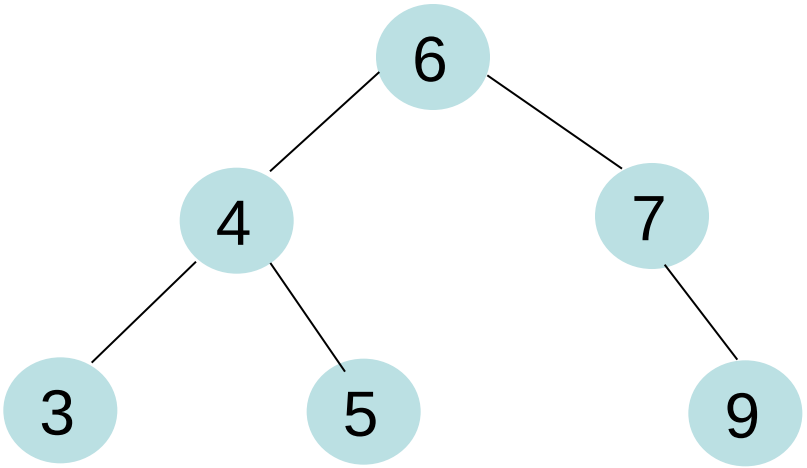
Datenstruk

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]

Löschen(6)



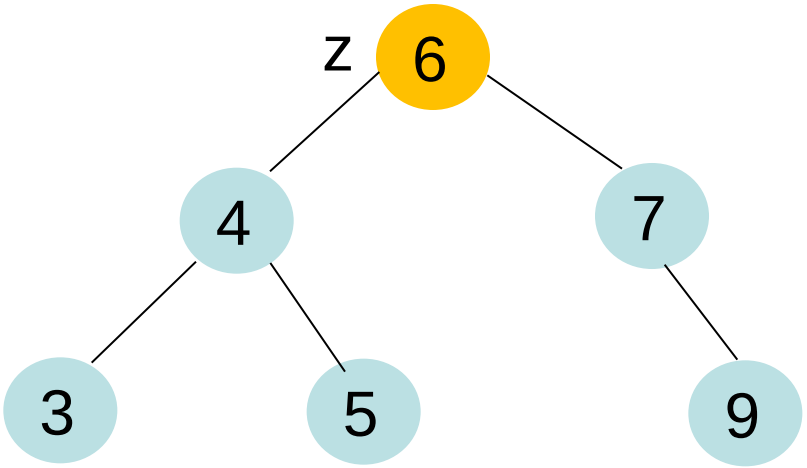
Datenstrukturen

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)



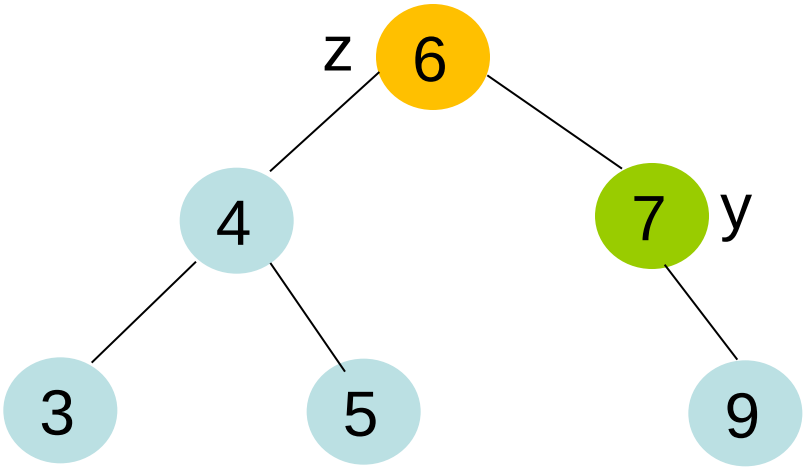
Datenstrukturen

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)



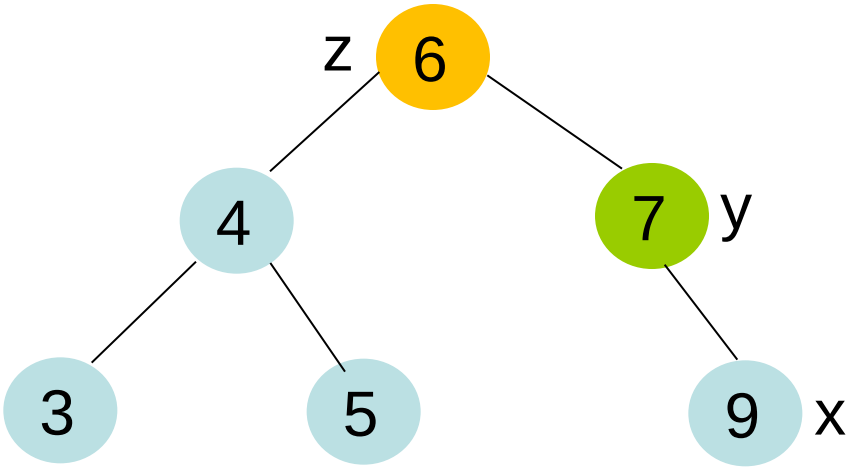
Datenstrukturen

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

…en(6)



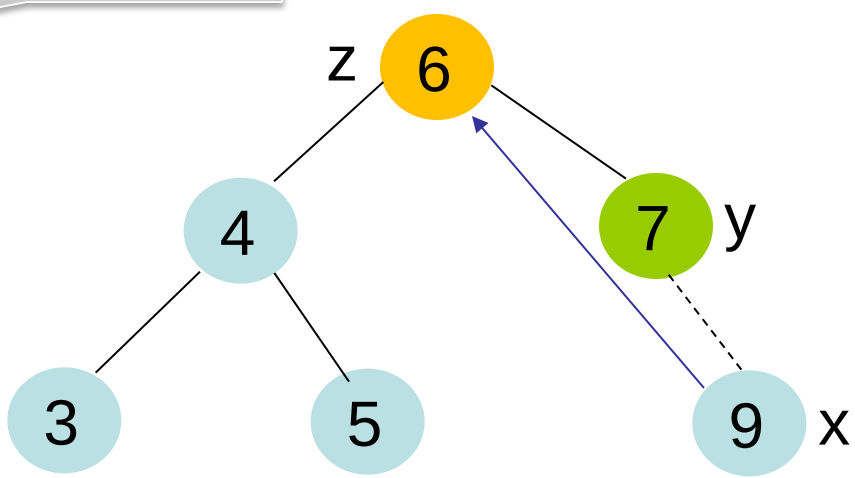
Datenstrukturen

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 Vaterzeiger von x

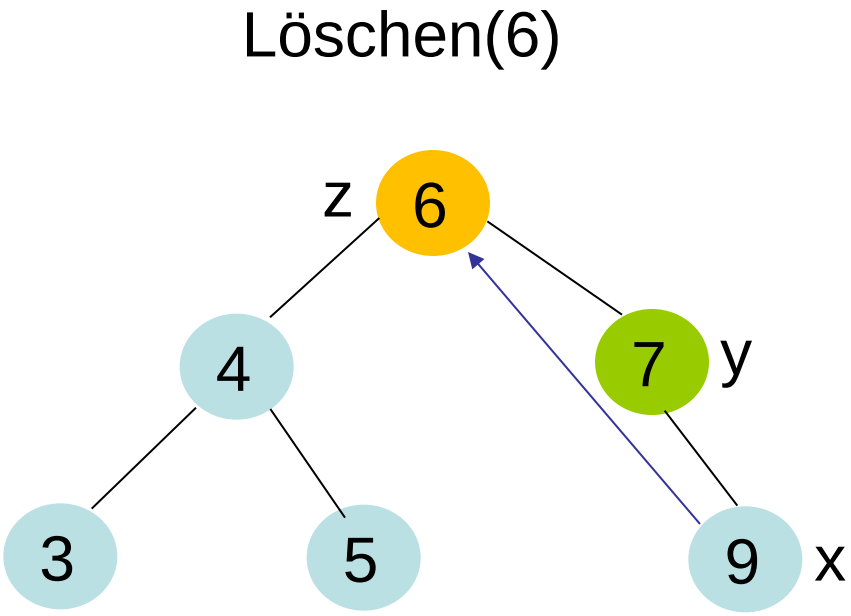
öschen(6)



Datenstrukturen

Löschen(T,z)

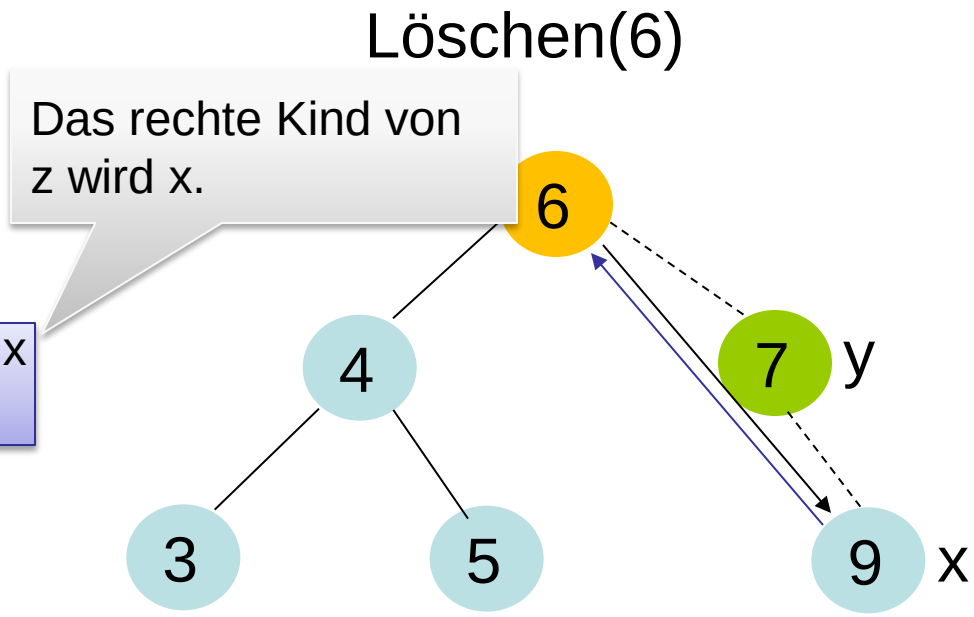
1. **if** lc[z]=**nil** or rc[z]=**nil then** y ← z
2. **else** y ← NachfolgerSuche(z)
3. **if** lc[y] ≠ nil **then** x ← lc[y]
4. **else** x ← rc[y]
5. **if** x ≠ **nil then** p[x] ← p[y]
6. **if** p[y]=**nil then** root[T] ← x
7. **else if** y=lc[p[y]] **then** lc[p[y]] ← x
8. **else** rc[p[y]] ← x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



Datenstrukturen

Löschen(T,z)

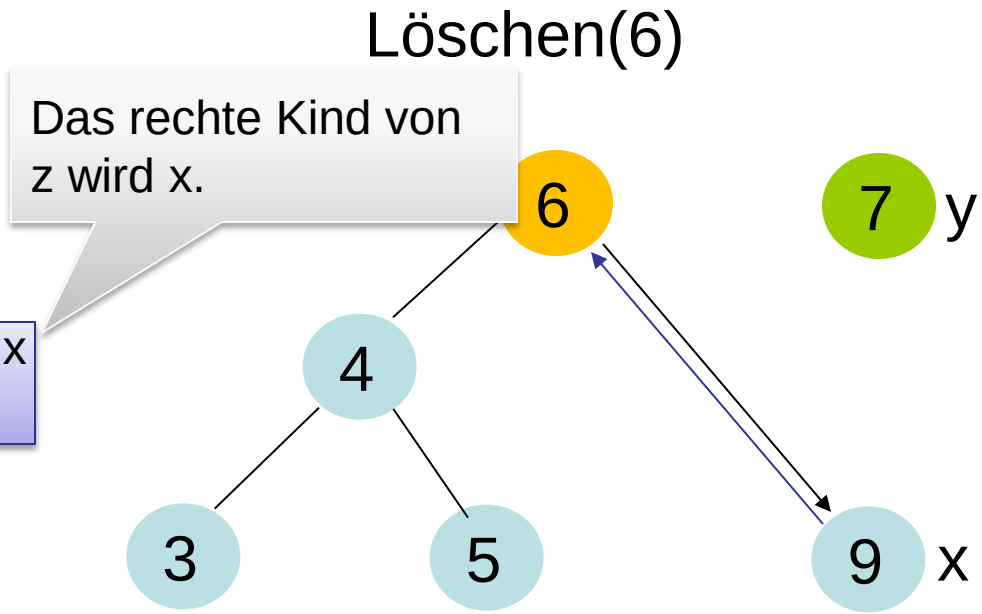
1. **if** lc[z]=**nil** or rc[z]=**nil then** y ← z
2. **else** y ← NachfolgerSuche(z)
3. **if** lc[y] ≠ nil **then** x ← lc[y]
4. **else** x ← rc[y]
5. **if** x ≠ **nil then** p[x] ← p[y]
6. **if** p[y]=**nil then** root[T] ← x
7. **else if** y=lc[p[y]] **then** lc[p[y]] ← x
8. **else** rc[p[y]] ← x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



Datenstrukturen

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]

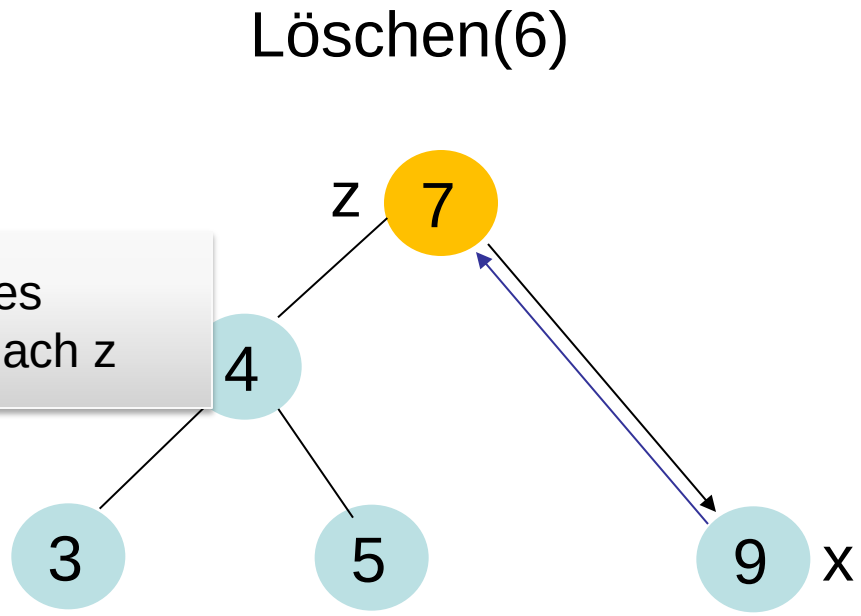


Datenstrukturen

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]
7. **else if** y=lc[p[y]] **then**
8. **else** rc[p[y]] ← x
9. key[z] ← key[y]

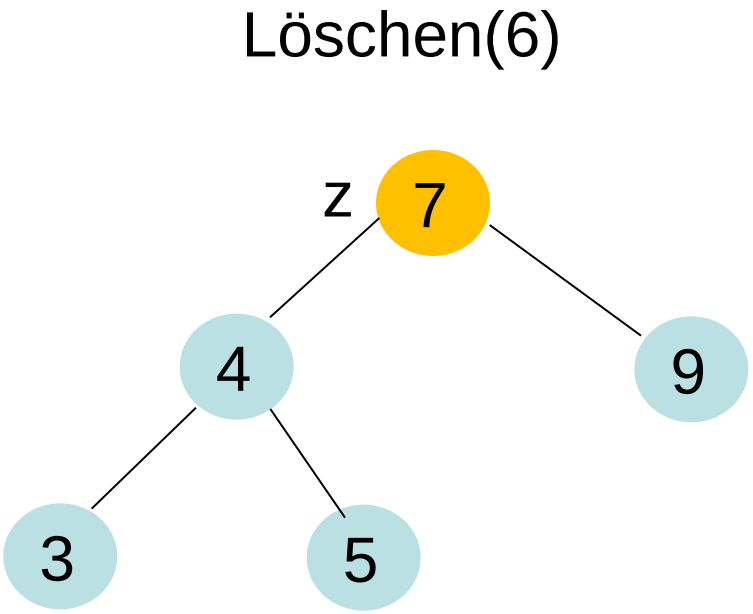
Umkopieren des Inhalts von y nach z



Datenstrukturen

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]



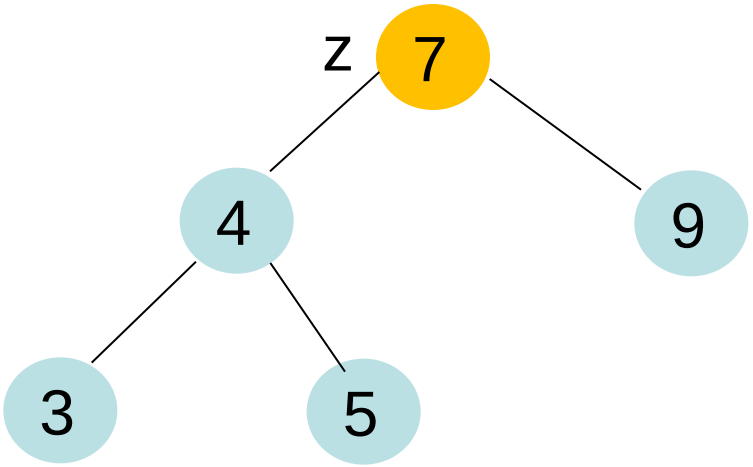
Datenstrukturen

Laufzeit O(h)

Löschen(T,z)

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

Löschen(6)



Datenstrukturen

Binäre Suchbäume

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

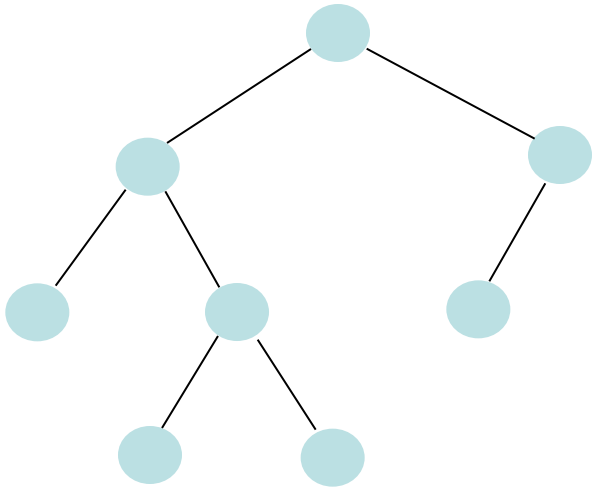
Frage

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

Datenstrukturen

AVL-Bäume [Adelson-Velsky und Landis]

- Ein Binärbaum heißt AVL-Baum, wenn für jeden Knoten gilt: Die Höhe seines linken und rechten Teilbaums unterscheidet sich höchstens um 1.



Datenstrukturen

Satz

Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Datenstrukturen

Satz

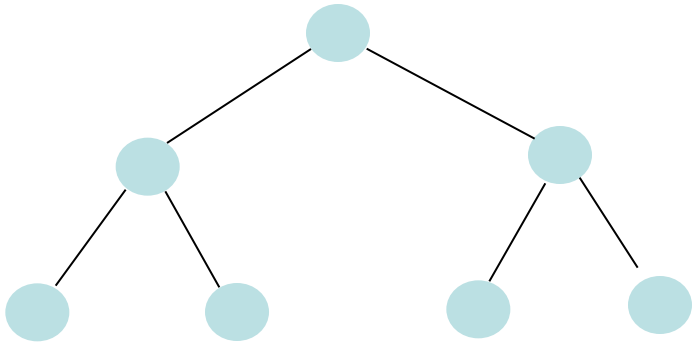
Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

a) $n \leq 2^{h+1} -1$:

- AVL-Baum ist Binärbaum



Datenstrukturen

Satz

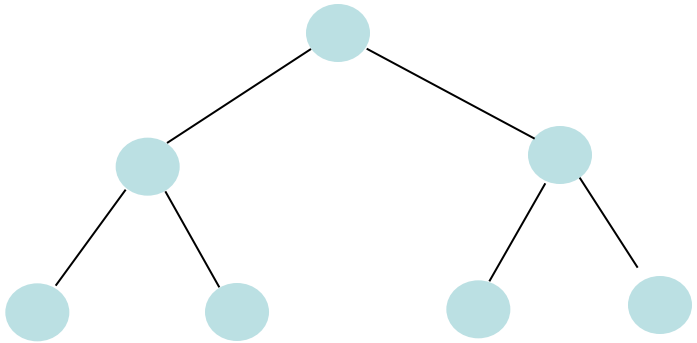
Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

a) $n \leq 2^{h+1} -1$:

- AVL-Baum ist Binärbaum
- Ein vollständiger Binärbaum hat eine maximale Anzahl Knoten unter allen Binärbäumen der Höhe h



Datenstrukturen

Satz

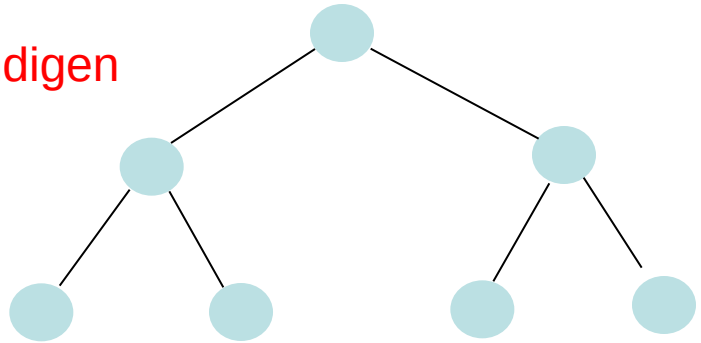
Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

a) $n \leq 2^{h+1} -1$:

- AVL-Baum ist Binärbaum
- Ein vollständiger Binärbaum hat eine maximale Anzahl Knoten unter allen Binärbäumen der Höhe h
- N(h) = Anzahl Knoten eines vollständigen Binärbaums der Höhe h



Datenstrukturen

Satz

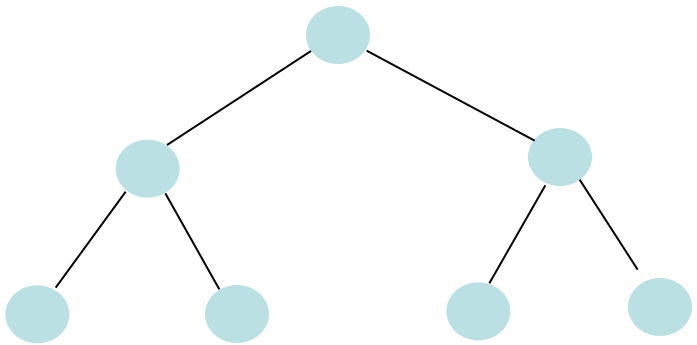
Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

a) $n \leq 2^{h+1} -1$:

- N(h) = Anzahl Knoten eines vollständigen Binärbaums der Höhe h



Datenstrukturen

Satz

Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

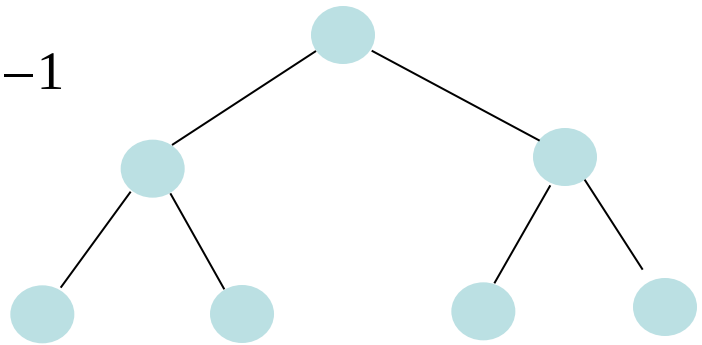
$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

a) $n \leq 2^{h+1} - 1$:

- N(h) = Anzahl Knoten eines vollständigen Binärbaums der Höhe h

- $N(h) = 1 + 2 + 4 ... + 2^h = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$



Datenstrukturen

Satz

Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

b) $(3/2)^h \leq n$:

- Beweis per Induktion über die Struktur von AVL-Bäumen

Datenstrukturen

Satz

Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

b) $(3/2)^h \leq n$:

- Beweis per Induktion über die Struktur von AVL-Bäumen
- (I.A.) Wir betrachten alle AVL-Bäume der Höhe 0 und 1.

Datenstrukturen

Satz

Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

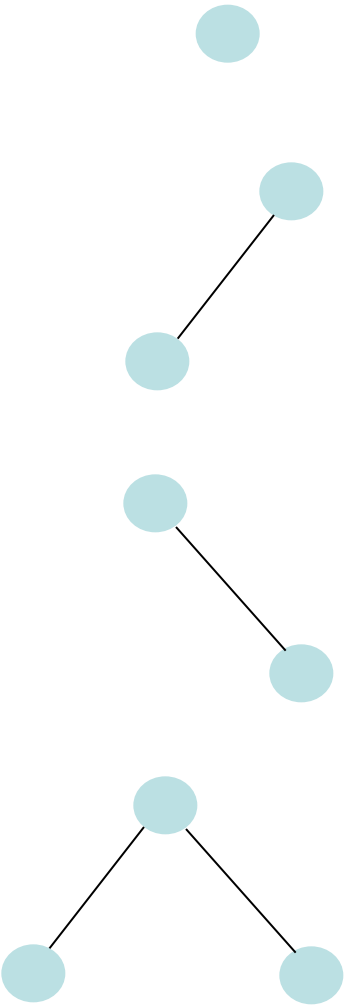
$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

b) $(3/2)^h \leq n$:

- Beweis per Induktion über die Struktur von AVL-Bäumen
- (I.A.) Wir betrachten alle AVL-Bäume der Höhe 0 und 1.
- h=0: Der Baum hat einen Knoten. Es gilt $(3/2)^h = 1 \leq 1$.

Datenstrukturen



Satz

Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

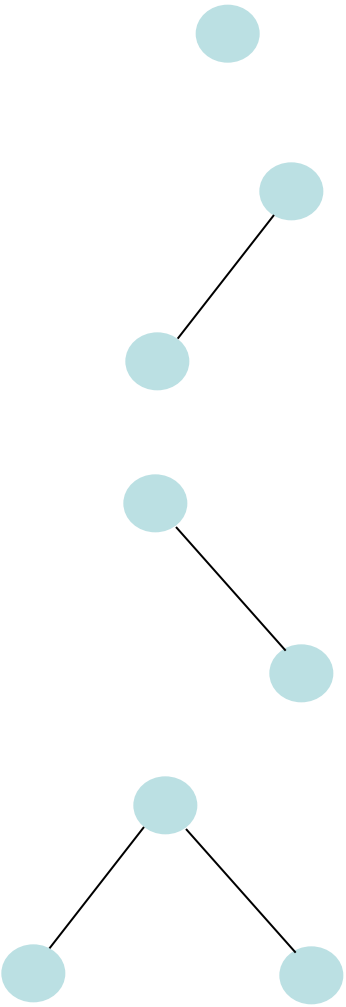
$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

b) $(3/2)^h \leq n$:

- Beweis per Induktion über die Struktur von AVL-Bäumen.
- (I.A.) Wir betrachten alle AVL-Bäume der Höhe 0 und 1.
- h=0: Der Baum hat einen Knoten. Es gilt $(3/2)^h = 1 \leq 1$.
- h=1: Der Baum hat 2 oder 3 Knoten. Es gilt $(3/2)^h = 3/2 \leq 2 \leq 3$.

Datenstrukturen



Satz

Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

b) $(3/2)^h \leq n$:

- Beweis per Induktion über die Struktur von AVL-Bäumen.
- (I.A.) Wir betrachten alle AVL-Bäume der Höhe 0 und 1.
- h=0: Der Baum hat einen Knoten. Es gilt $(3/2)^h = 1 \leq 1$.
- h=1: Der Baum hat 2 oder 3 Knoten. Es gilt $(3/2)^h = 3/2 \leq 2 \leq 3$.

Datenstrukturen

Satz

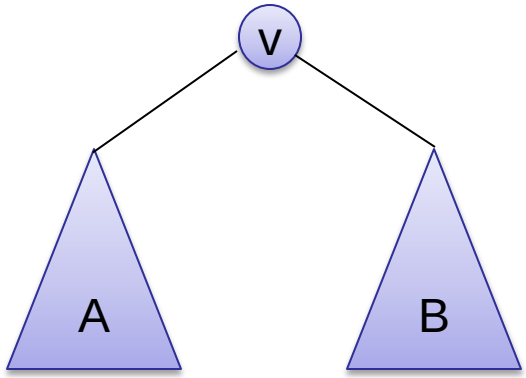
Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

b) $(3/2)^h \leq n$:

- (I.V.) Für jeden AVL-Baum der Höhe j, $0 \leq j \leq h$, gilt der Satz.



Datenstrukturen

Satz

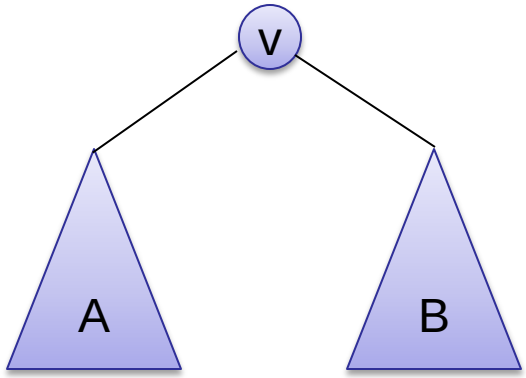
Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

b) $(3/2)^h \leq n$:

- (I.V.) Für jeden AVL-Baum der Höhe j, $0 \leq j \leq h$, gilt der Satz.
- (I.S.) Sei h≥1. Betrachte AVL-Baum T der Höhe h+1 mit Wurzel v.
- Seien A,B linker bzw. rechter Teilbaum von v.



Datenstrukturen

Satz

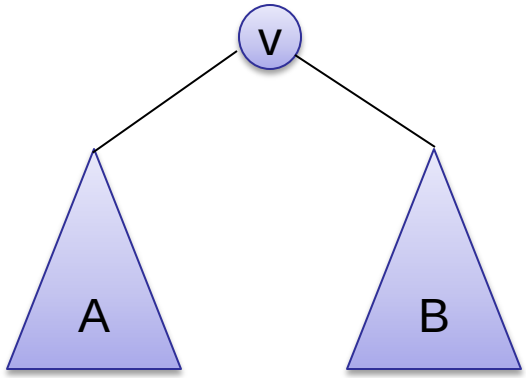
Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

b) $(3/2)^h \leq n$:

- (I.V.) Für jeden AVL-Baum der Höhe j, $0 \leq j \leq h$, gilt der Satz.
- (I.S.) Sei h≥1. Betrachte AVL-Baum T der Höhe h+1 mit Wurzel v.
- Seien A,B linker bzw. rechter Teilbaum von v.
- A oder B (oder beide) hat Tiefe h.



Datenstrukturen

Satz

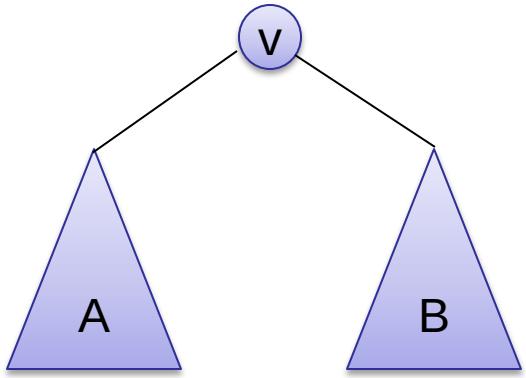
Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

b) $(3/2)^h \leq n$:

- (I.V.) Für jeden AVL-Baum der Höhe j, $0 \leq j \leq h$, gilt der Satz.
- (I.S.) Sei h≥1. Betrachte AVL-Baum T der Höhe h+1 mit Wurzel v.
- Seien A,B linker bzw. rechter Teilbaum von v.
- A oder B (oder beide) hat Tiefe h.
- Wegen AVL-Eigenschaft haben A und B Tiefe mindestens h-1≥0.



Datenstrukturen

Satz

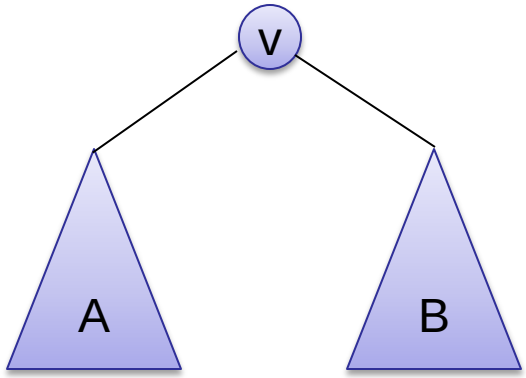
Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

b) $(3/2)^h \leq n$:

- (I.V.) Für jeden AVL-Baum der Höhe j, $0 \leq j \leq h$, gilt der Satz.
- (I.S.) Sei h≥1. Betrachte AVL-Baum T der Höhe h+1 mit Wurzel v.
- Seien A,B linker bzw. rechter Teilbaum von v.
- A oder B (oder beide) hat Tiefe h.
- Wegen AVL-Eigenschaft haben A und B Tiefe mindestens h-1≥0.



Datenstrukturen

Satz

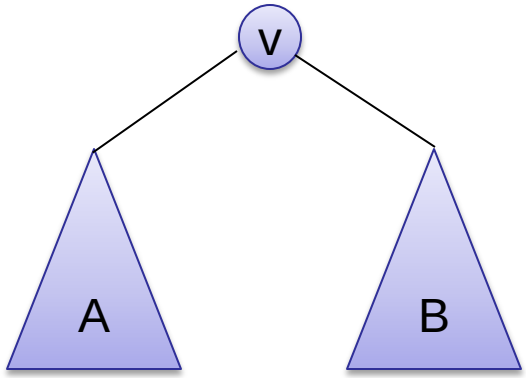
Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

b) $(3/2)^h \leq n$:

- Wegen AVL-Eigenschaft haben A und B Tiefe mindestens h-1≥0.
- Da T ein AVL-Baum ist, sind auch A und B AVL-Bäume.



Datenstrukturen

Satz

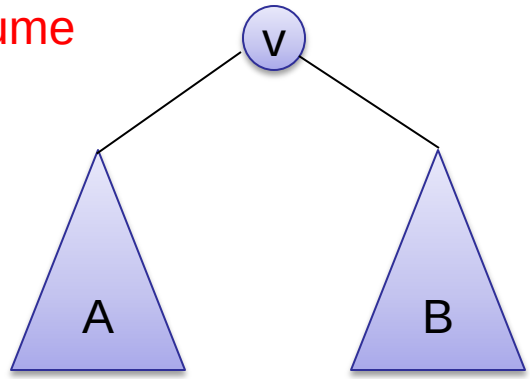
Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

b) $(3/2)^h \leq n$:

- Wegen AVL-Eigenschaft haben A und B Tiefe mindestens h-1≥0.
- Da T ein AVL-Baum ist, sind auch A und B AVL-Bäume.
- Kann also (I.V.) anwenden, da A und B AVL-Bäume der Tiefe ≥0 sind



Datenstrukturen

Satz

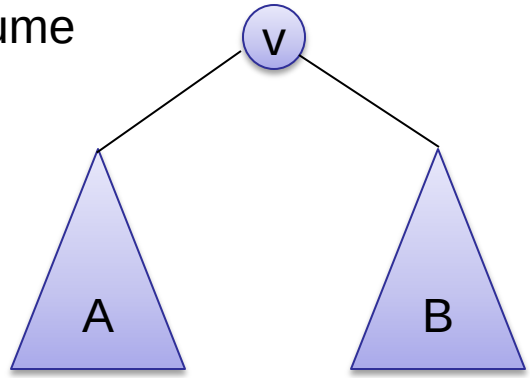
Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

b) $(3/2)^h \leq n$:

- Wegen AVL-Eigenschaft haben A und B Tiefe mindestens h-1≥0.
- Da T ein AVL-Baum ist, sind auch A und B AVL-Bäume.
- Kann also (I.V.) anwenden, da A und B AVL-Bäume der Tiefe ≥0 sind
- Es gibt drei Fälle:
  1) A,B haben Höhe h
  2) A hat Höhe h und B Hat Höhe h-1
  3) A hat Höhe h-1 und B hat Höhe h



Datenstrukturen

Satz

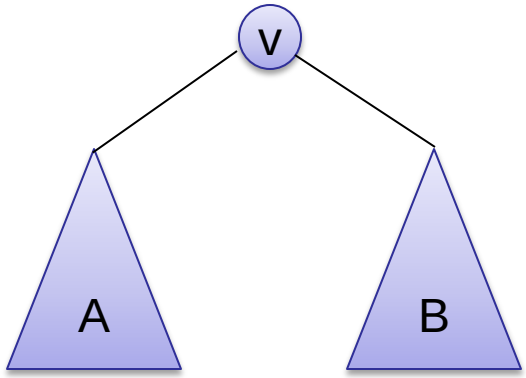
Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

b) $(3/2)^h \leq n$:

- Sei T(h) die minimale Anzahl Knoten in einem AVL-Baum der Tiefe h.



Datenstrukturen

Satz

Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

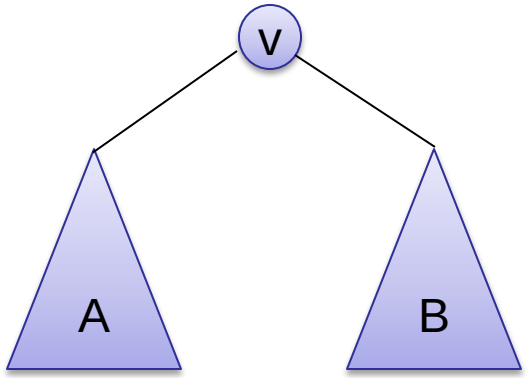
$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

b) $(3/2)^h \leq n$:

- Sei T(h) die minimale Anzahl Knoten in einem AVL-Baum der Tiefe h.
  Nach (I.V.) gilt in allen drei Fällen

$$T(h+1) \geq T(h) + T(h-1) + 1 \geq \left(\frac{3}{2}\right)^h + \left(\frac{3}{2}\right)^{h-1} + 1$$



Datenstrukturen

Satz

Für jeden AVL-Baum der Höhe h≥0 mit n Knoten gilt:

$$(3/2)^h \leq n \leq 2^{h+1}-1$$

Beweis

b) $(3/2)^h \leq n$:

- Sei T(h) die minimale Anzahl Knoten in einem AVL-Baum der Tiefe h. Nach (I.V.) gilt in allen drei Fällen

$$T(h+1) \geq T(h) + T(h-1) + 1 \geq \left(\frac{3}{2}\right)^h + \left(\frac{3}{2}\right)^{h-1} + 1$$

$$\geq (1+3/2)\cdot\left(\frac{3}{2}\right)^{h-1} \geq \left(\frac{3}{2}\right)^2 \cdot \left(\frac{3}{2}\right)^{h-1} = \left(\frac{3}{2}\right)^{h+1}$$