2.5 Bäume

Balancierte Bäume

- Nachteil bei normalen Suchbäumen: Worst-case Aufwand ist O(n)
- Tritt bei degenerierten Bäumen auf
- Kann durch Balancierung verhindert werden
- **Problem**: Stelle Balance in O(log n) wieder her

Balancierte Bäume

- **Gewichtsbalance**: Für jeden Knoten unterscheidet sich die Anzahl der Knoten im linken und rechten Teilbaum um maximal eins.
- **Höhenbalance**: Für jeden Knoten unterscheidet sich die Höhe des linken und rechten Teilbaums um maximal eins.

Balancierte Bäume

- Zur Erinnerung:
  - Maximale Zahl von Knoten in einem Baum der Höhe h ist $O(2^h)$
  - Minimale Zahl von Knoten in einem Baum der Höhe h ist $\Omega(h)$
  - Minimale Zahl von Knoten in einem balancierten Baum der Höhe h ist $\Omega(2^{h/2})$

Balancierte Bäume

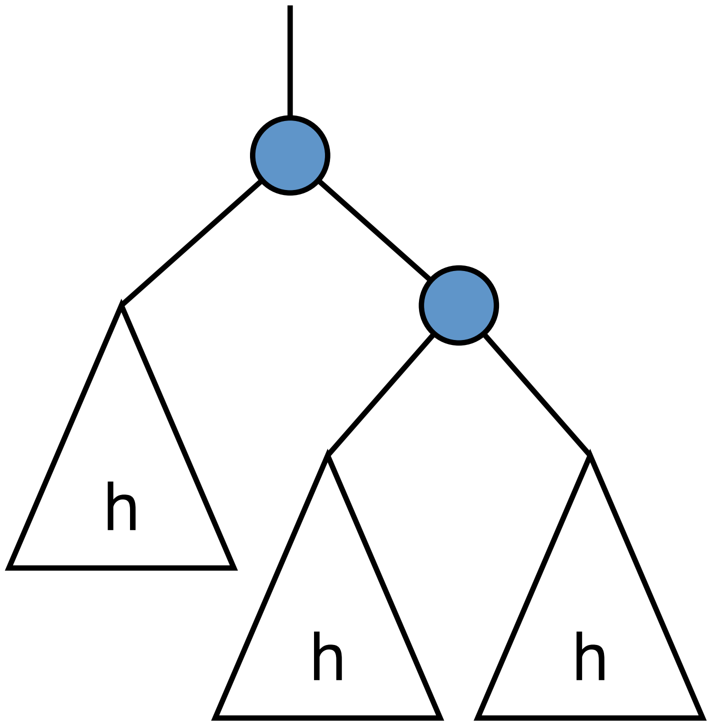
- Zur Erinnerung:
  - Minimale Höhe eines Baums mit n Knoten ist $\Omega(\log n)$
  - Maximale Höhe eines Baums mit n Knoten ist $O(n)$
  - Maximale Höhe eines balancierten Baums mit n Knoten ist $O(2 \times \log n) = O(\log n)$

2.5 Bäume

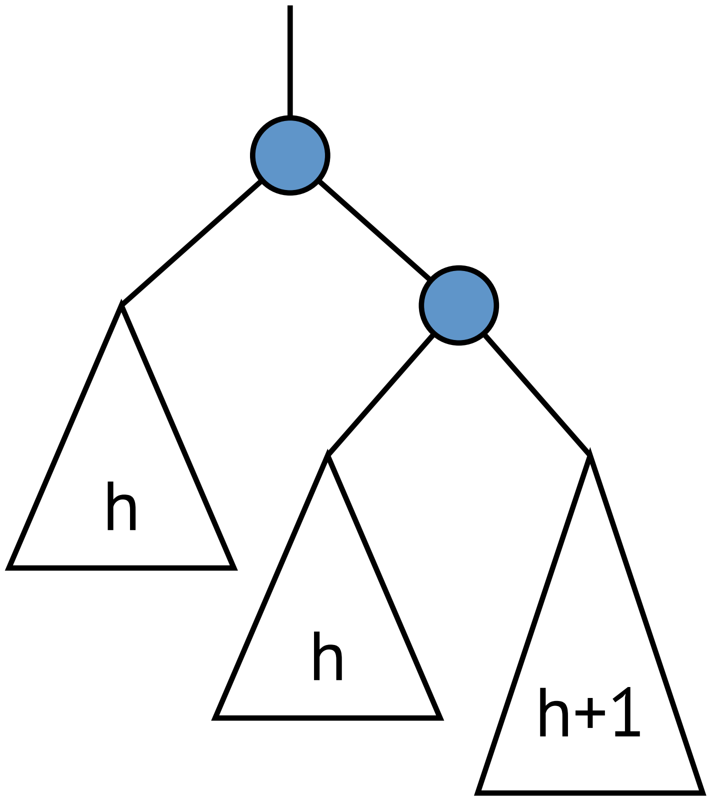
AVL-Bäume

- Werden wie normale binäre Suchbäume behandelt
- Jeder Knoten speichert sein Ungleichgewicht (+1,0,–1)
- Nach Insert() oder Delete() Wiederherstellung der Balance durch **Rotation**

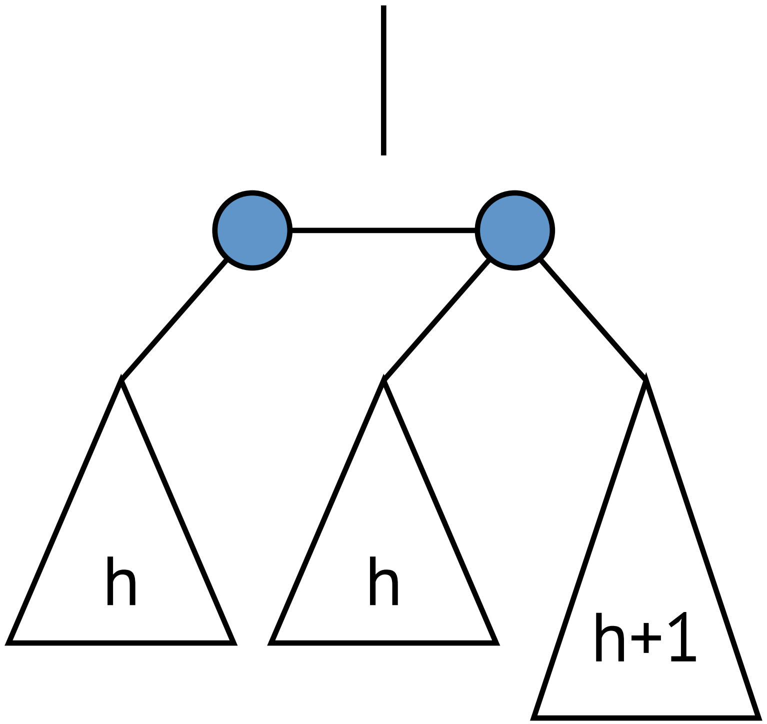
AVL-Bäume



AVL-Bäume

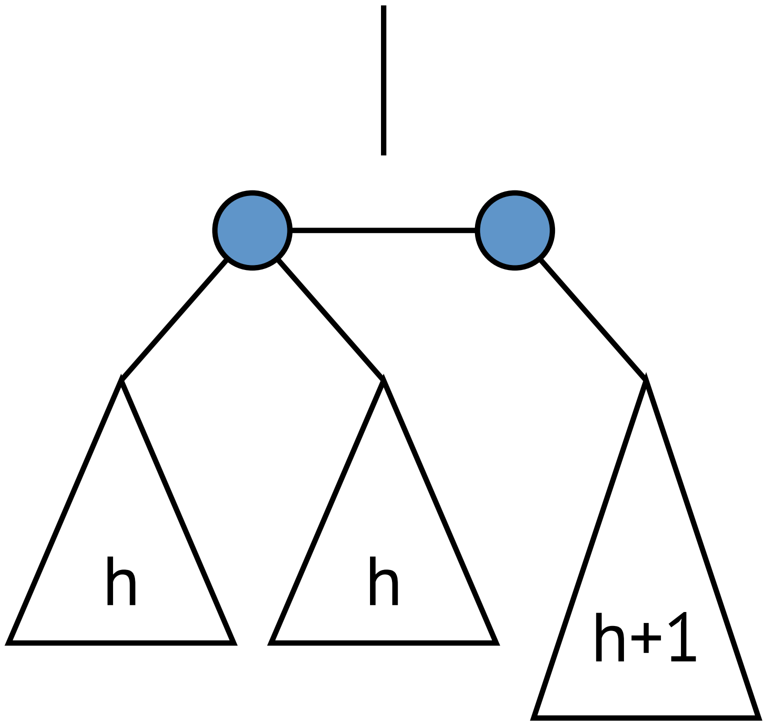


AVL-Bäume



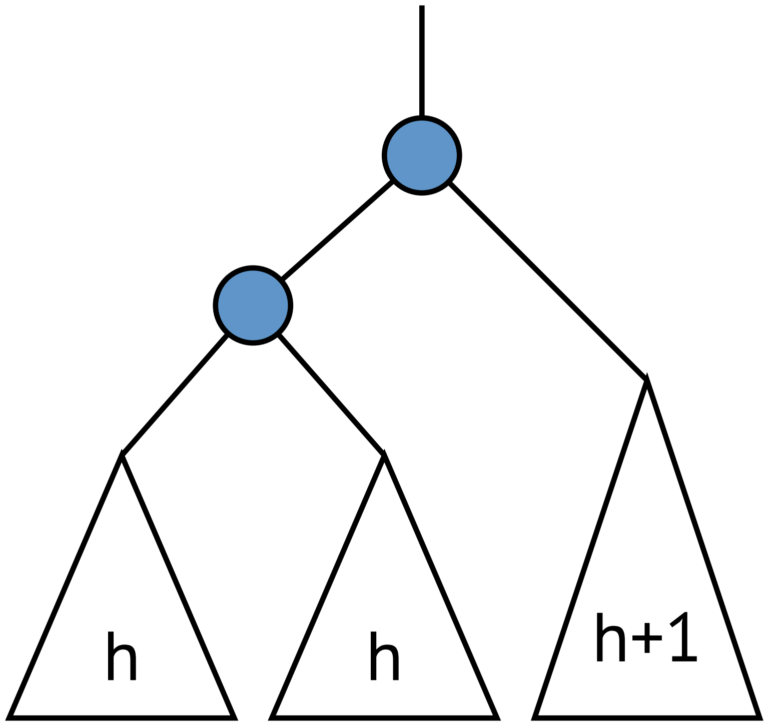
Einfache Rotation

AVL-Bäume



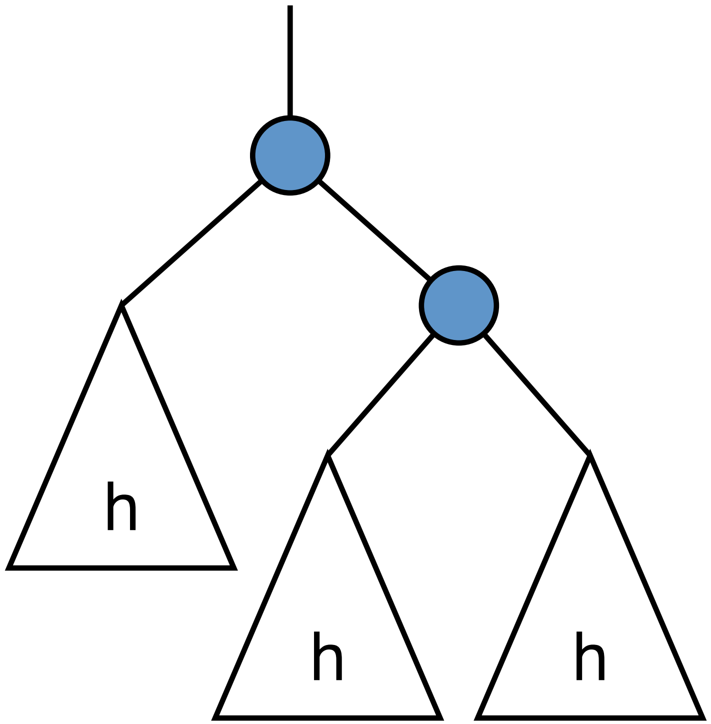
Einfache Rotation

AVL-Bäume

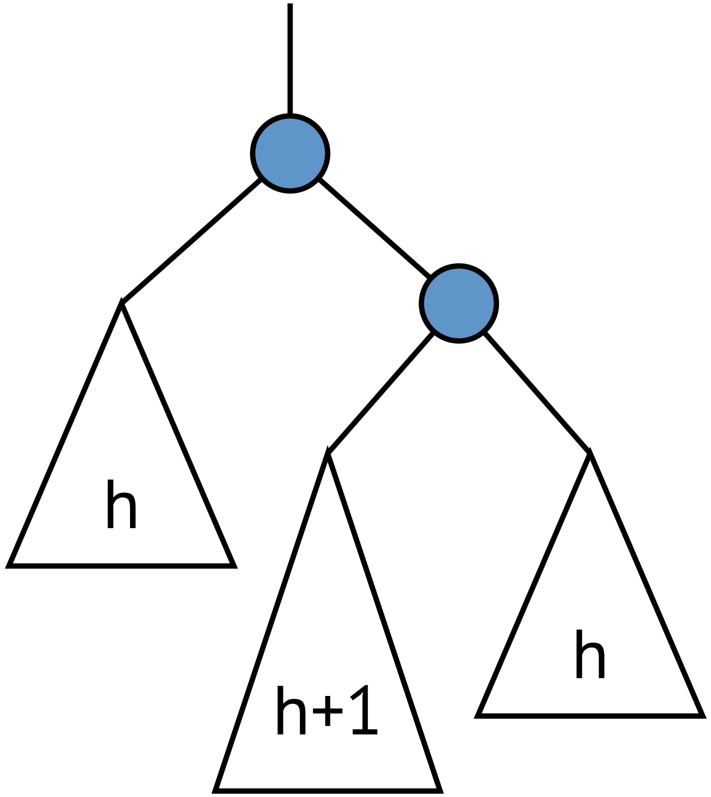


Einfache Rotation

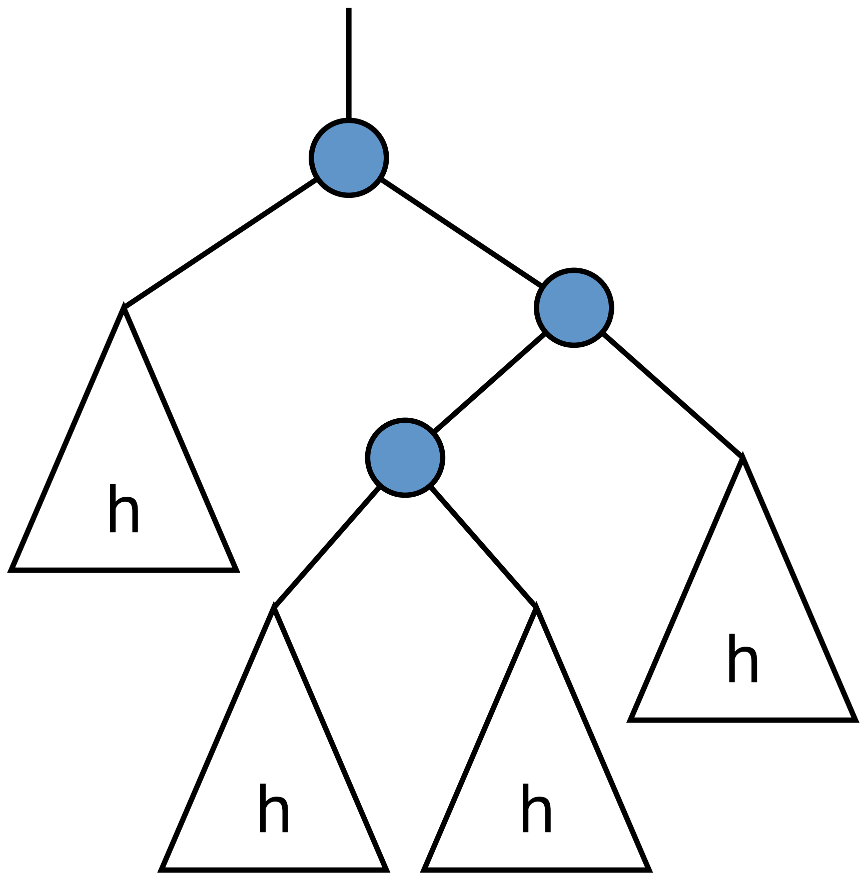
AVL-Bäume



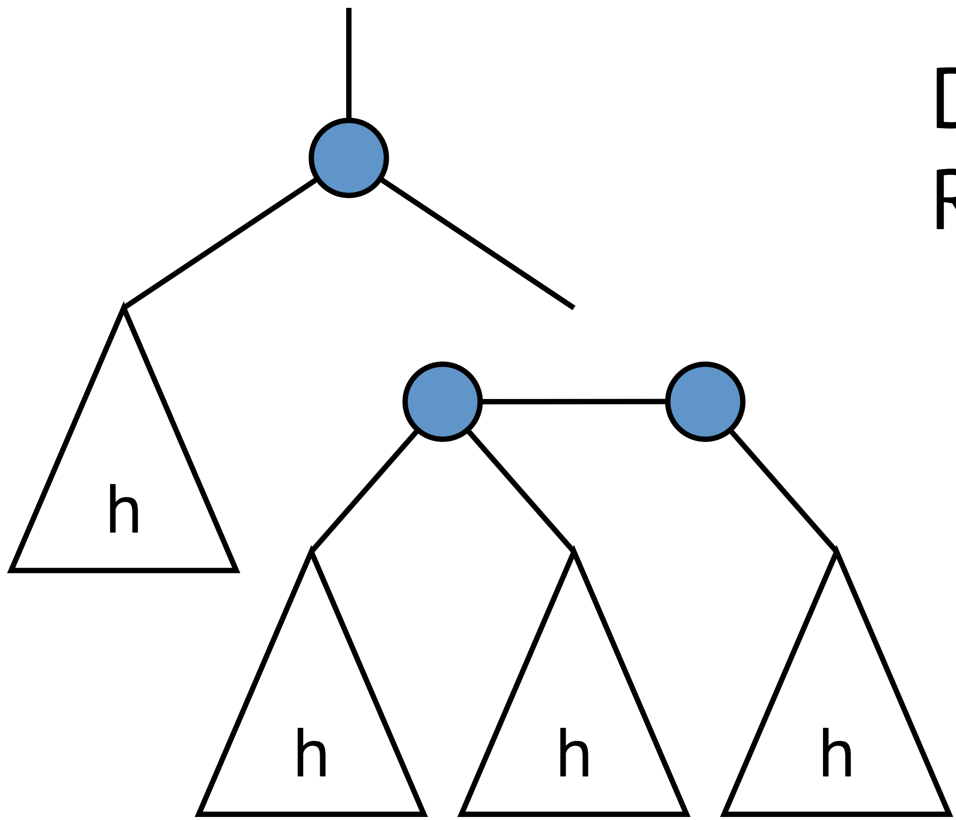
AVL-Bäume



AVL-Bäume

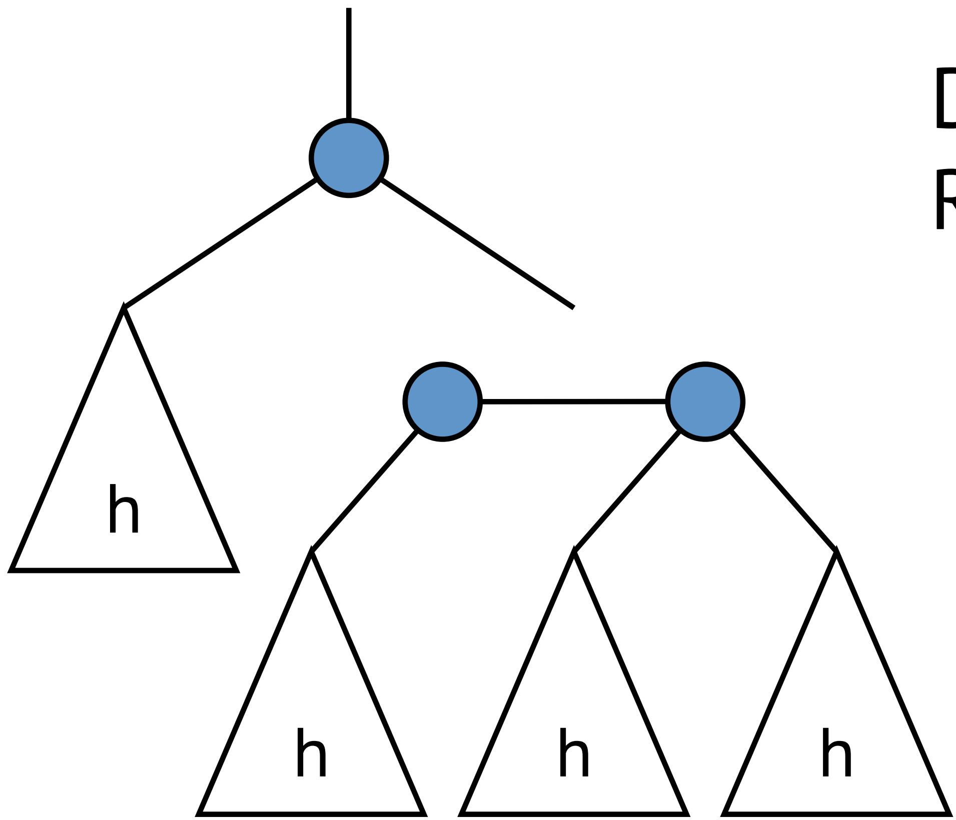


AVL-Bäume



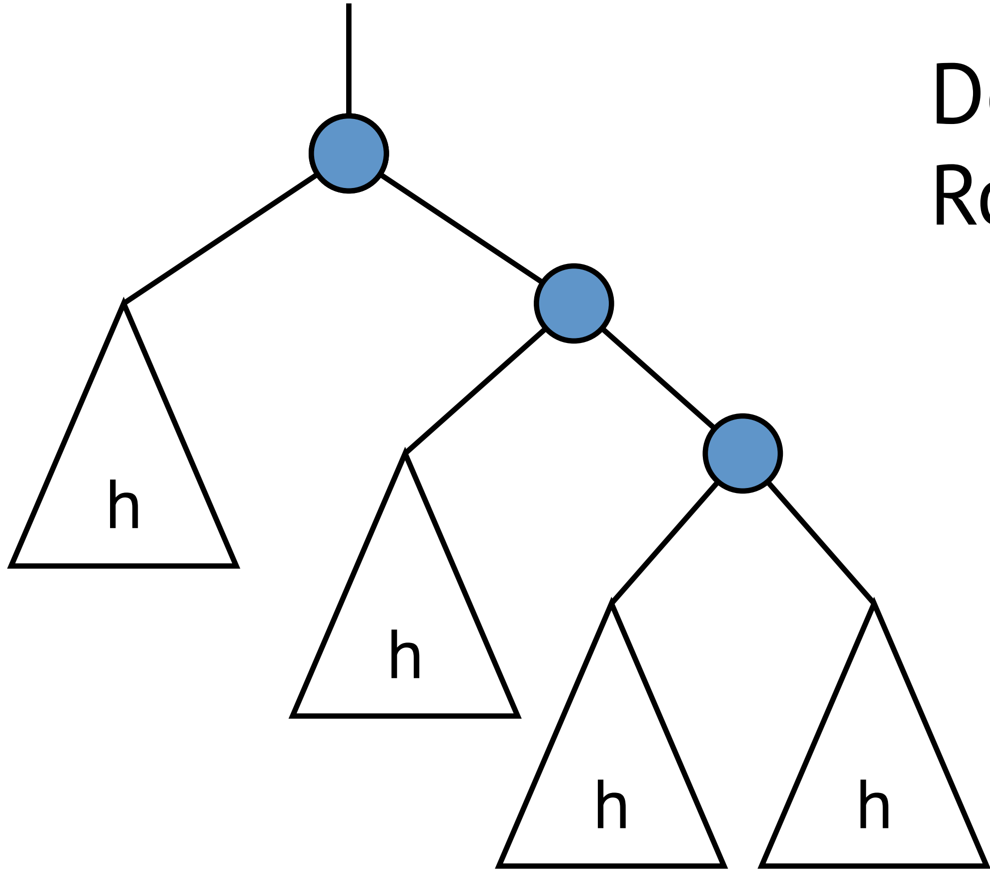
Doppelte Rotation

AVL-Bäume



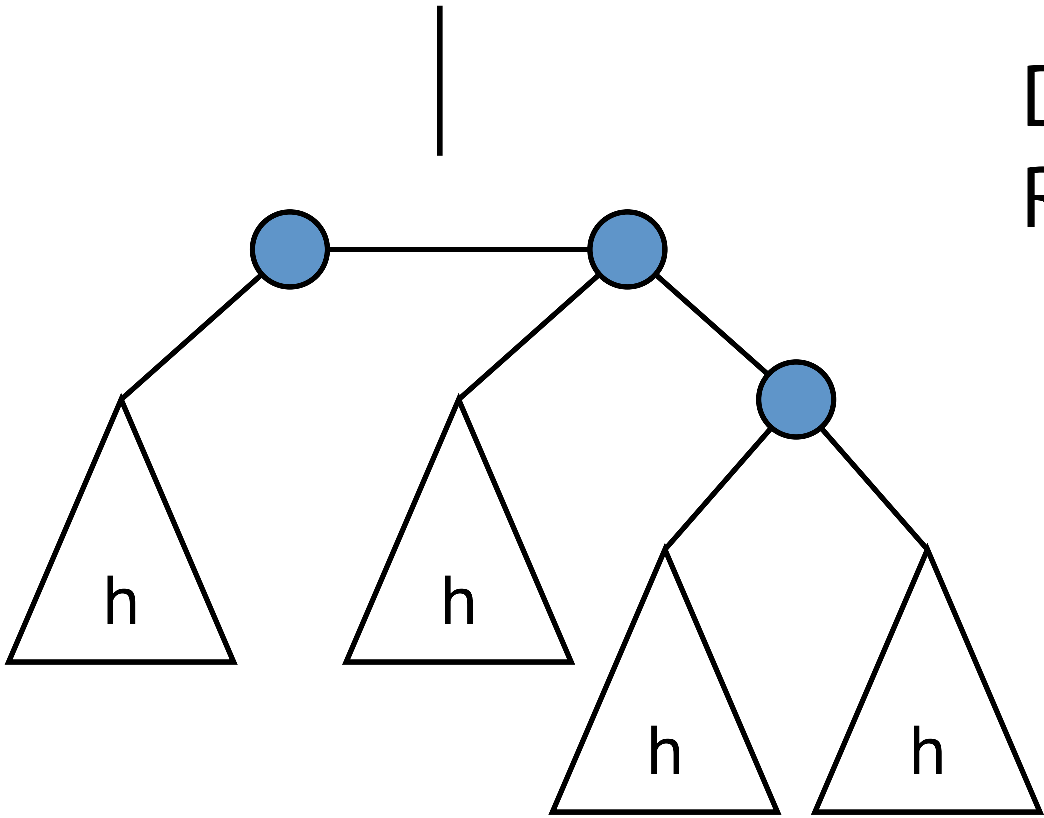
Doppelte Rotation

AVL-Bäume

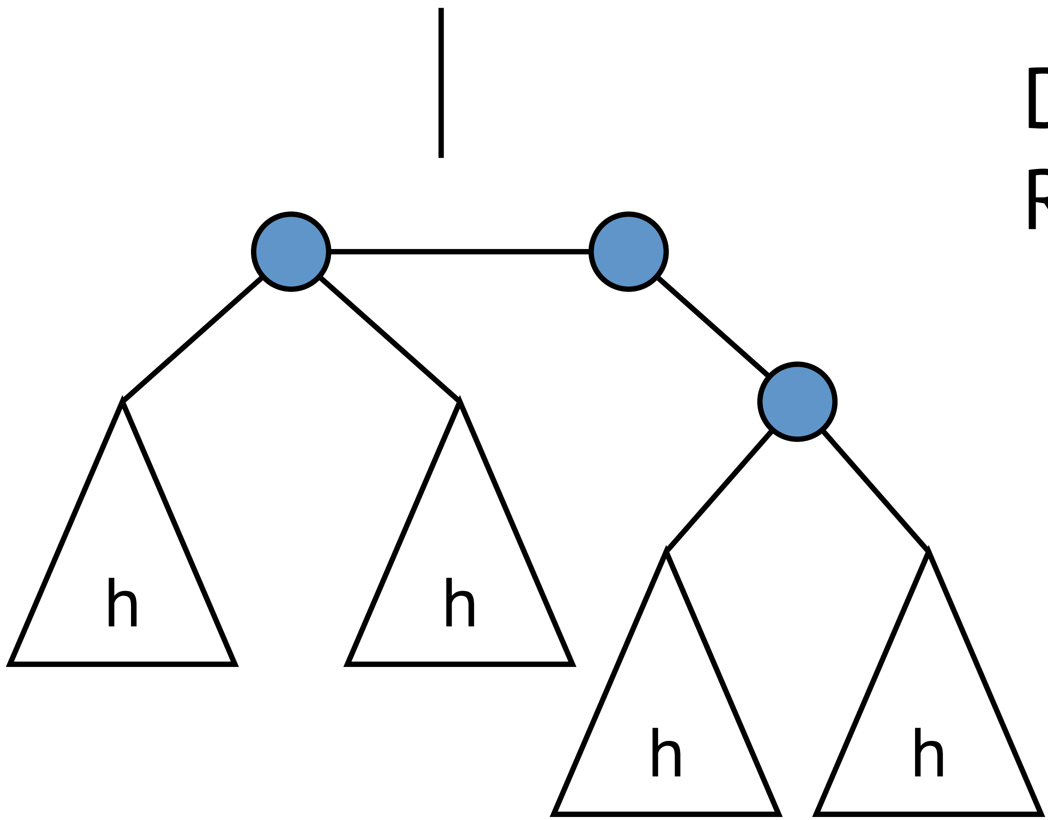


Doppelte Rotation

AVL-Bäume

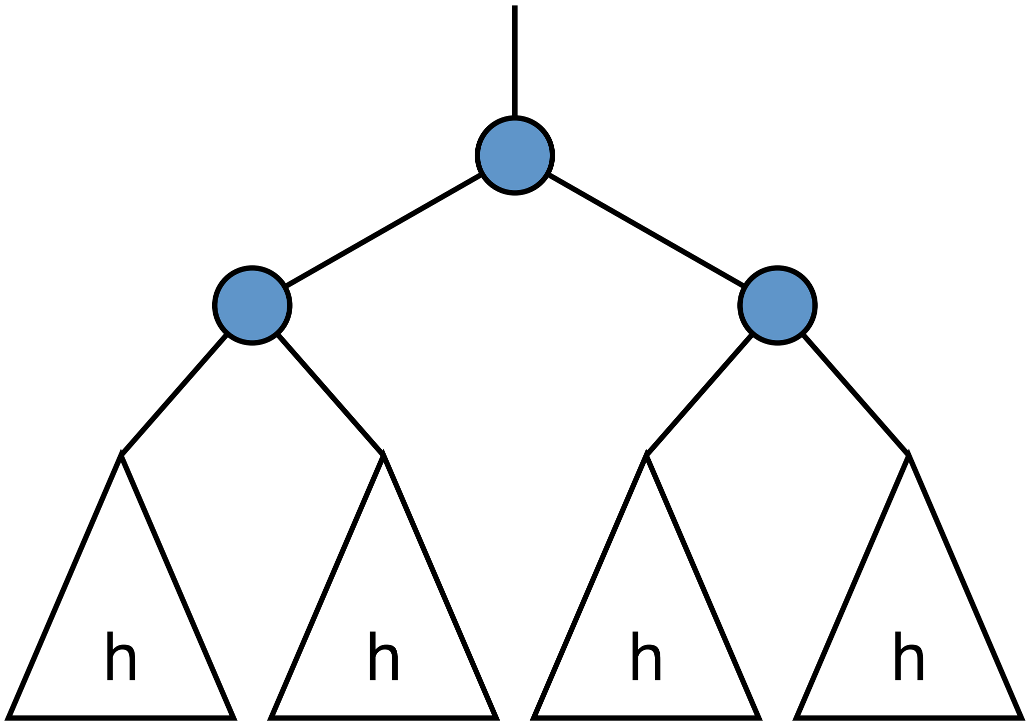


Doppelte Rotation



Doppelte Rotation

AVL-Bäume



Doppelte Rotation

AVL-Bäume

- Rotationen müssen ggf. nach oben propagiert werden
- Im schlechtesten Fall O(log n) Rotationen bei Delete()
- Nur praktikabel, wenn gesamte Datenstruktur im Hauptspeicher

2.5 Bäume

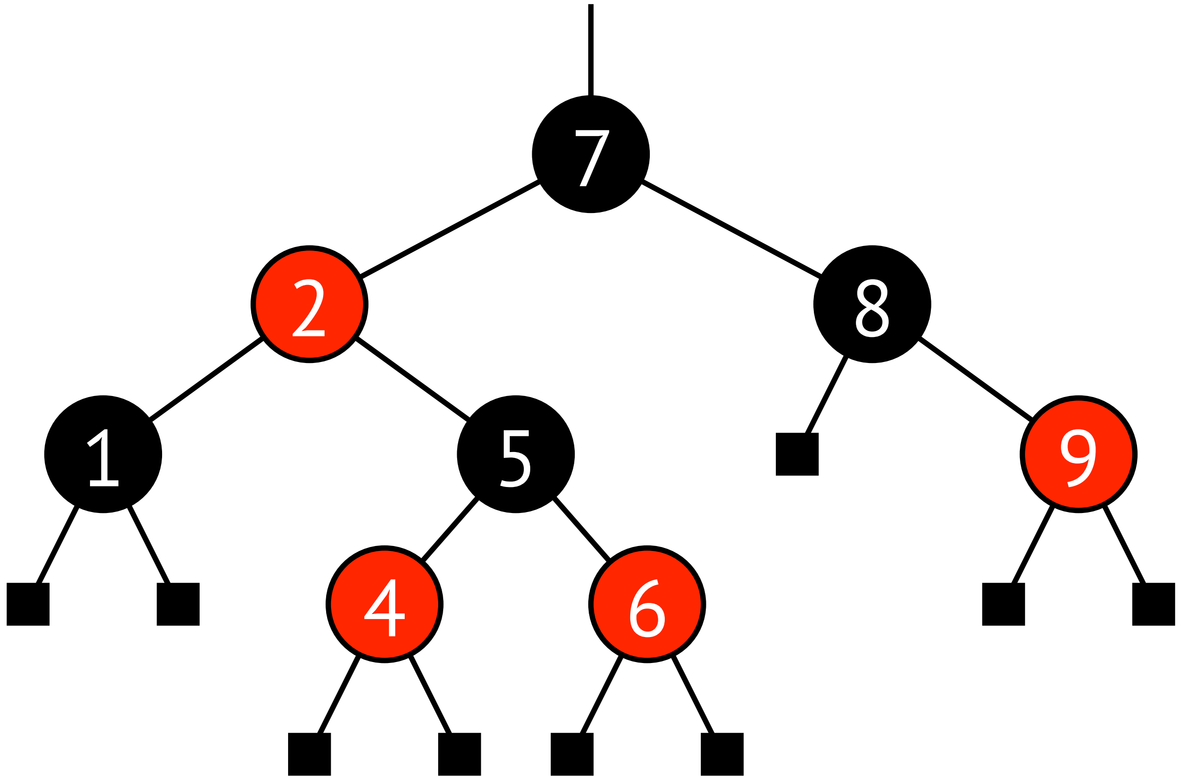
Rot-Schwarz-Bäume

- Binärer Suchbaum, jeder Knoten hat zwei Söhne oder keinen
- Rote und schwarze Knoten
- Re-Balancing durch Rotationen und Umfärben

Rot-Schwarz-Bäume

- Konsistenzregeln
  1. Jeder Knoten ist rot oder schwarz
  2. Die Wurzel ist schwarz
  3. Die Blätter sind schwarz
  4. Die Söhne eines roten Knotens sind schwarz.
  5. Alle Pfade von einem Knoten zu den nachfolgenden Blättern haben gleich viele schwarze Knoten

Rot-Schwarz-Bäume



Blätter werden im Folgenden nicht mehr dargestellt.

Rot-Schwarz-Bäume

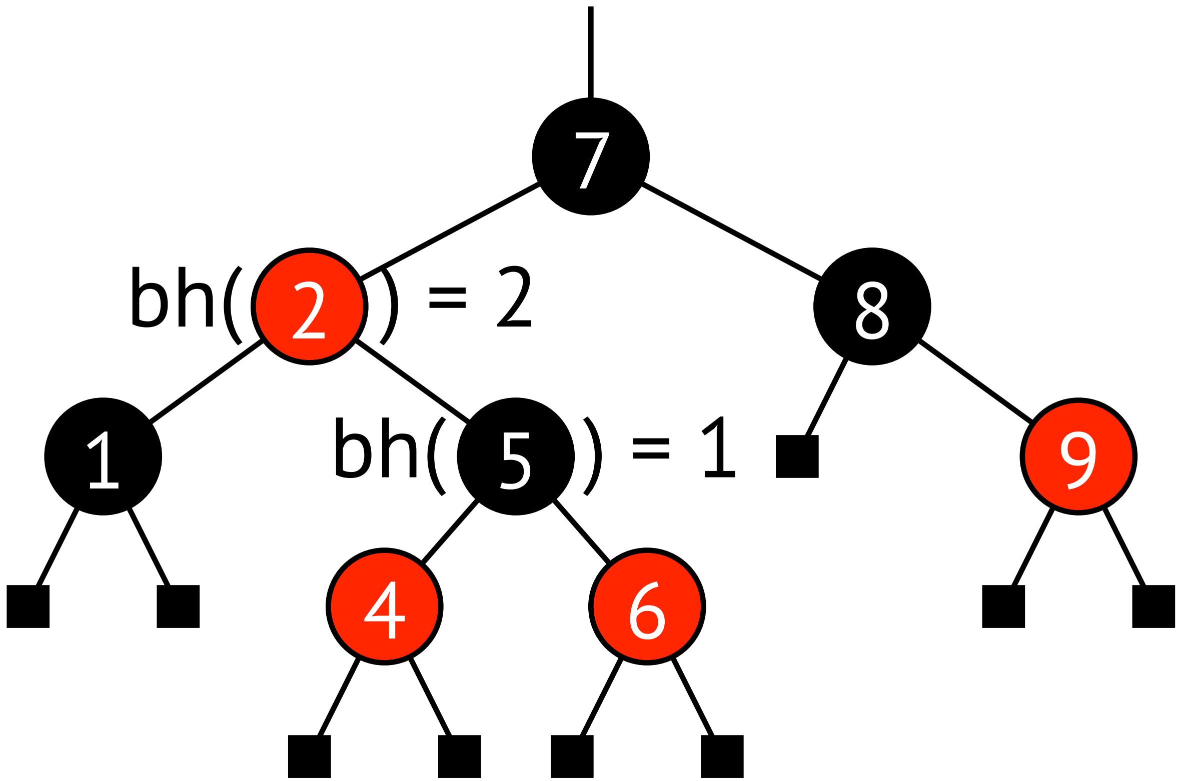
- Eigenschaften
  - Maximales Pfadlängenverhältnis 2:1
  - Pfadlängen $\Theta(\log n)$

- Lemma:

  Die Höhe eines Rot-Schwarz-Baums mit n inneren Knoten ist höchstens $2 \times \mathrm{ld}(n+1)$

- Die Zahl der schwarzen Knoten auf einem (dann jeden) Pfad von einem Knoten x (ausschließlich) bis zu einem Blatt (einschließlich) sei bh(x)

Rot-Schwarz-Bäume



Rot-Schwarz-Bäume

- Der Baum unter einem Knoten x enthält mindestens $2^{bh(x)}-1$ innere Knoten
  - I.A. x ist ein Blatt $\rightarrow$ $2^{bh(x)}-1=2^0-1=0$
  - I.V. Die Beh. gelte für alle y mit
    $height(y) < height(x)$
  - I.S. x sei ein innerer Knoten mit Söhnen x_1, x_2
    - ➡ $bh(x_1), bh(x_2) \geq bh(x)-1$
    - ➡ der Baum unter x enthält mindestens $(2^{bh(x)-1}-1)+(2^{bh(x)-1}-1)+1 = 2^{bh(x)}-1$ innere Knoten

Rot-Schwarz-Bäume

- Es sei $h = \text{height(root)}$ die Höhe des Baumes
  - Mindestens die Hälfte der Knoten auf einem Pfad von der Wurzel zu einem Blatt müssen schwarz sein.

➡ $\text{bh(root)} \geqslant h / 2$

➡ $n \geqslant 2^{h/2}-1$

➡ $h \leqslant 2 \times \text{ld}(n+1)$

Rot-Schwarz-Bäume

- Insert(), Delete()
  - Wie bei normalen binären Suchbäumen mit anschließender Wiederherstellung der Konsistenzbedingungen
  - Insert() → drei verschiedene Fälle
  - Delete() → vier verschiedene Fälle
  - Pseudo-Code: siehe Cormen et al.

Rot-Schwarz-Bäume: Insert()

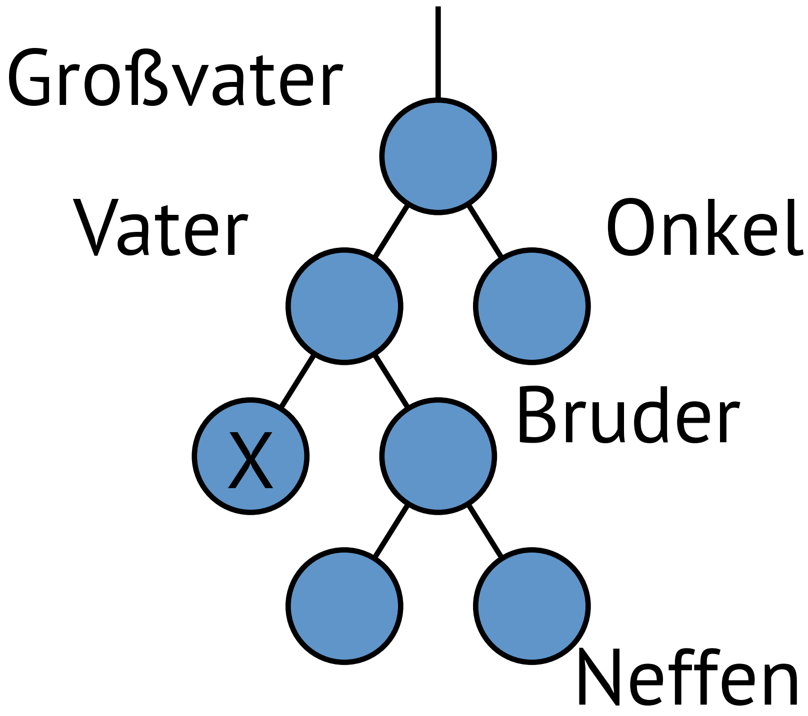
- Jeder eingefügte Knoten wird zunächst rot gefärbt
- Bei Wiederherstellung durch Umfärben oder Rotation kann genau eine weitere Konsistenzverletzung auftreten.
- Propagiere Modifikation nach oben.

Rot-Schwarz-Bäume: Insert()

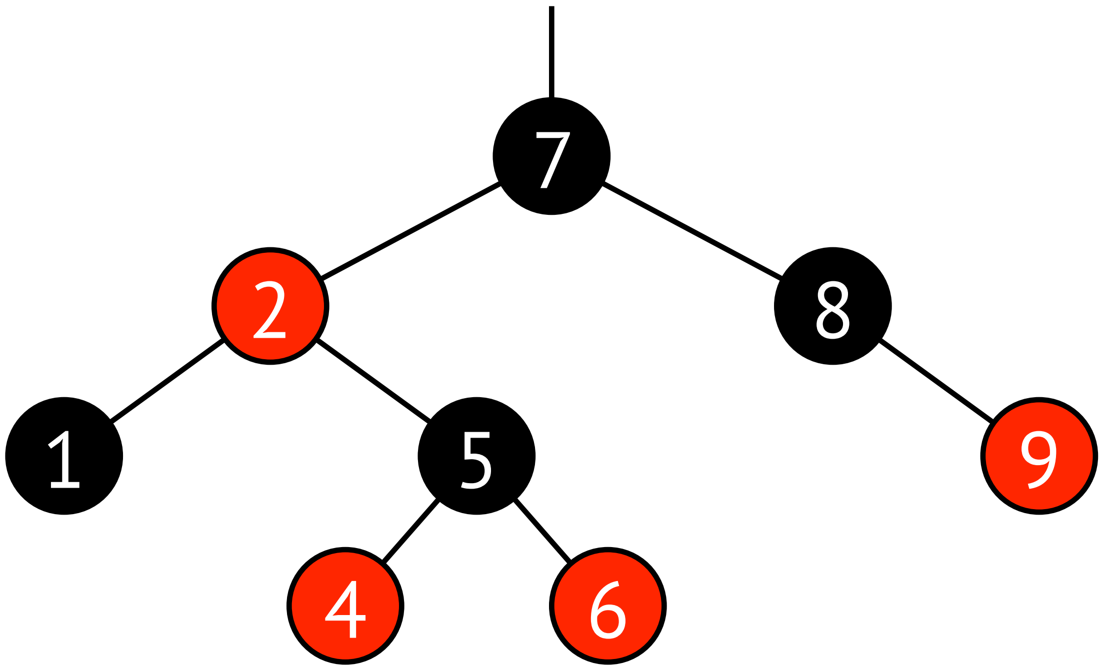
- Mögliche Konsistenzverletzungen
  - Neuer Knoten ist Wurzel
  - Neuer Knoten ist Nachfolger eines roten Knotens
- Fallunterscheidung nach der Farbe des **Onkels**

- Konsistenzregeln
  1. Jeder Knoten ist rot oder schwarz.
  2. Die Wurzel ist schwarz.
  3. Die Blätter sind schwarz.
  4. Die Söhne eines roten Knotens sind schwarz.
  5. Alle Pfade von einem Knoten zu den nachfolgenden Blättern haben gleich viele schwarze Knoten.

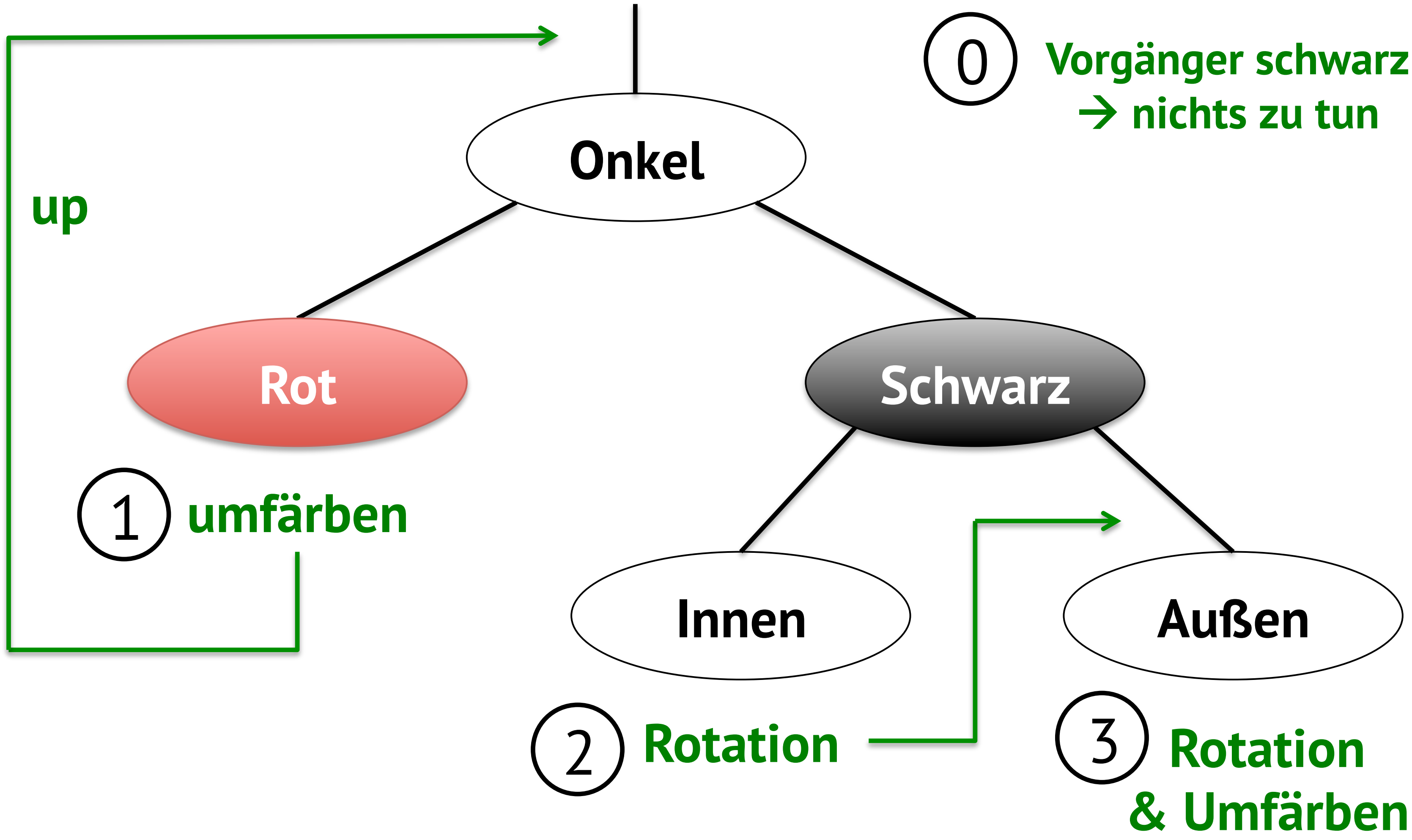
Rot-Schwarz-Bäume: Insert()



Rot-Schwarz-Bäume: Insert()

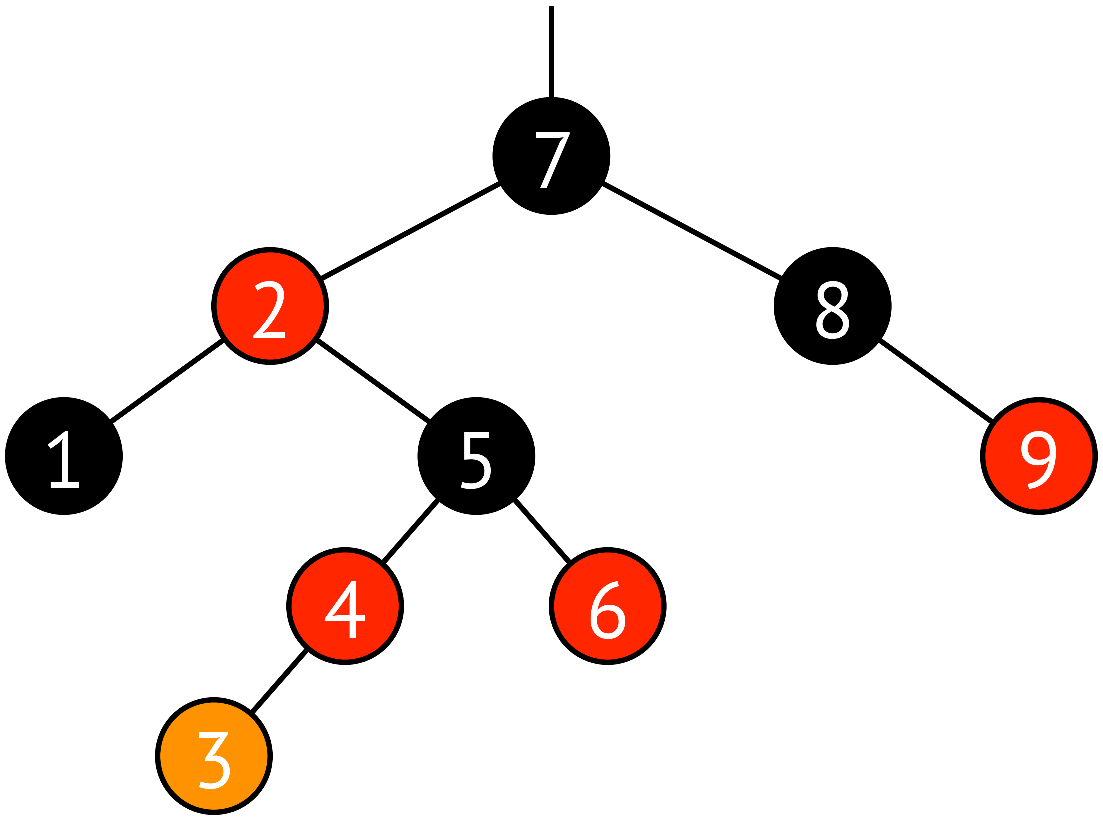


Rot-Schwarz-Bäume Insert – Scheme



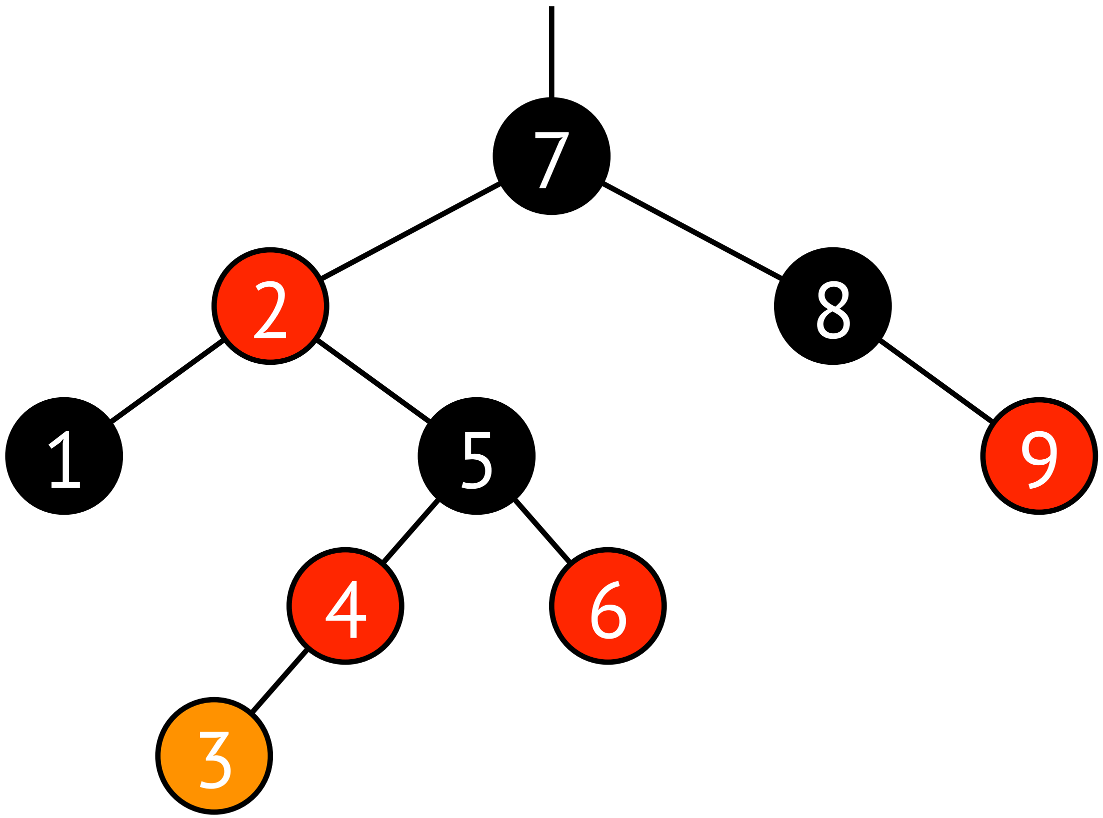
Rot-Schwarz-Bäume: Insert()

Einfügen zunächst wie bei normalen binären Suchbäumen



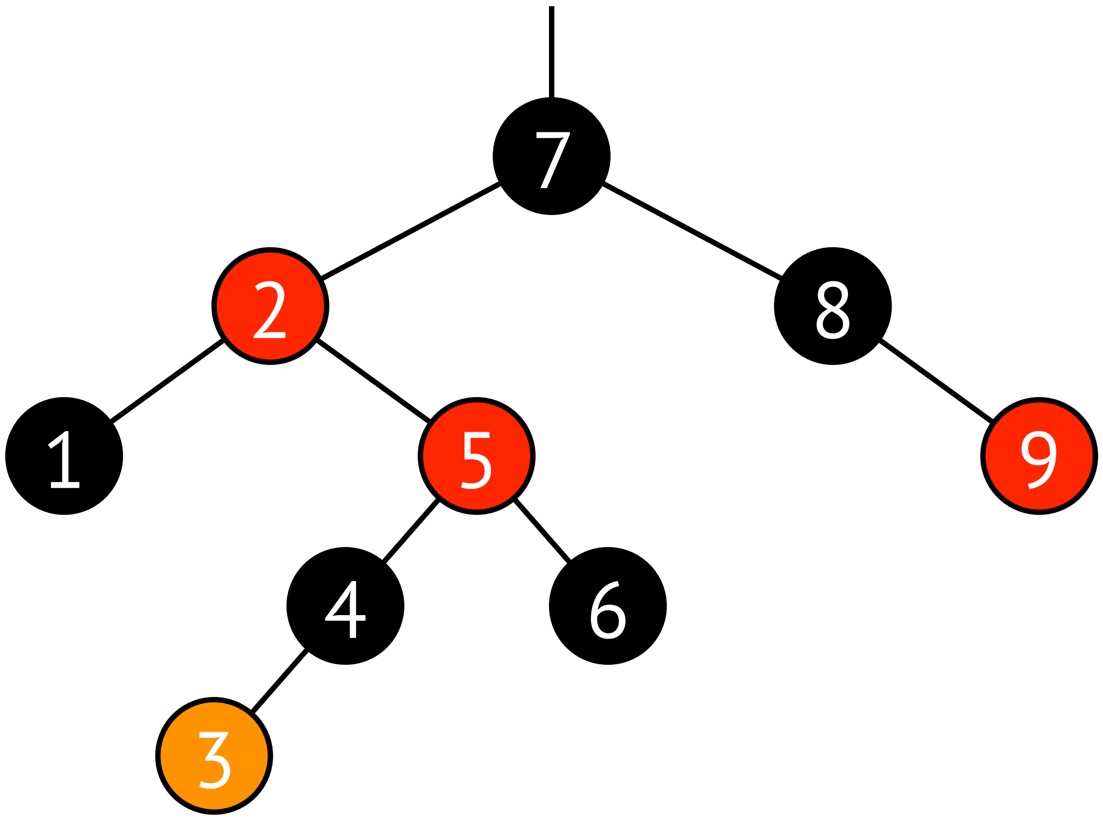
Rot-Schwarz-Bäume: Insert()

Fall 1: Der Onkel ist rot



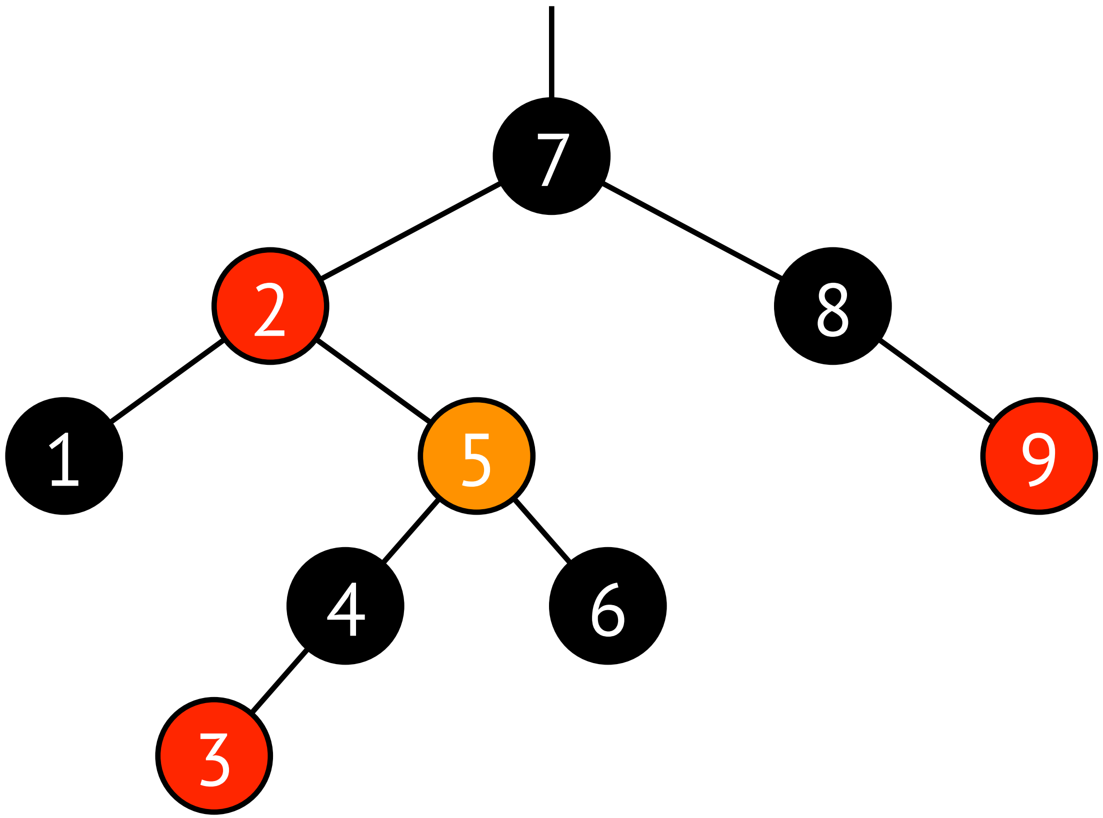
Rot-Schwarz-Bäume: Insert()

Fall 1: Der Onkel ist rot → umfärben

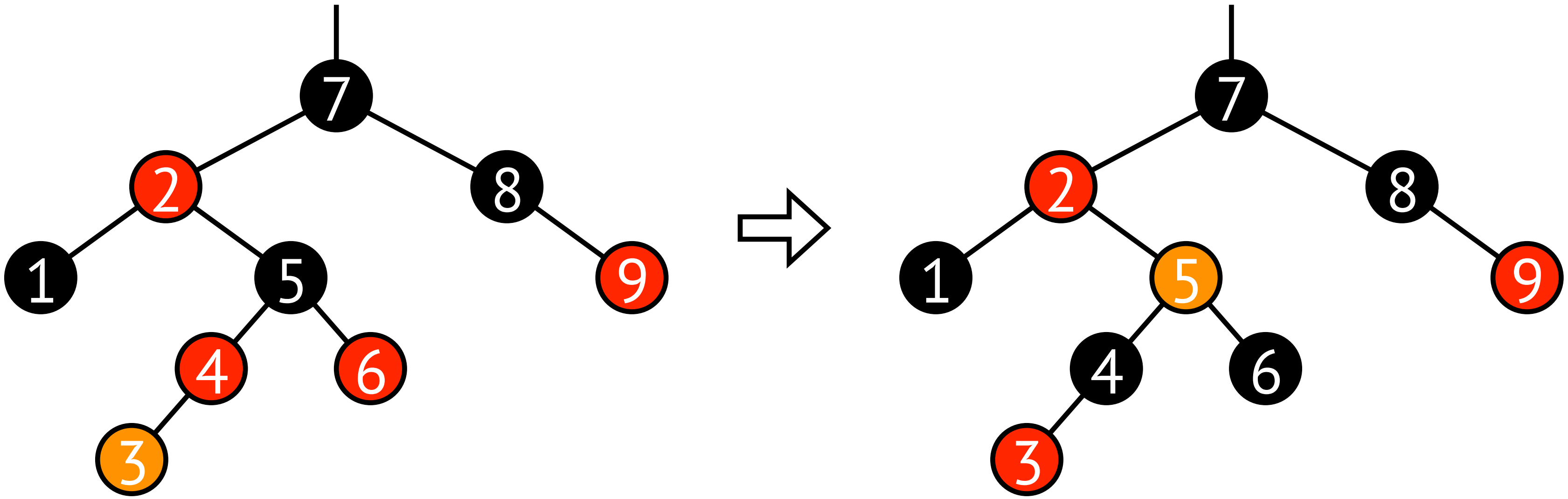


Rot-Schwarz-Bäume: Insert()

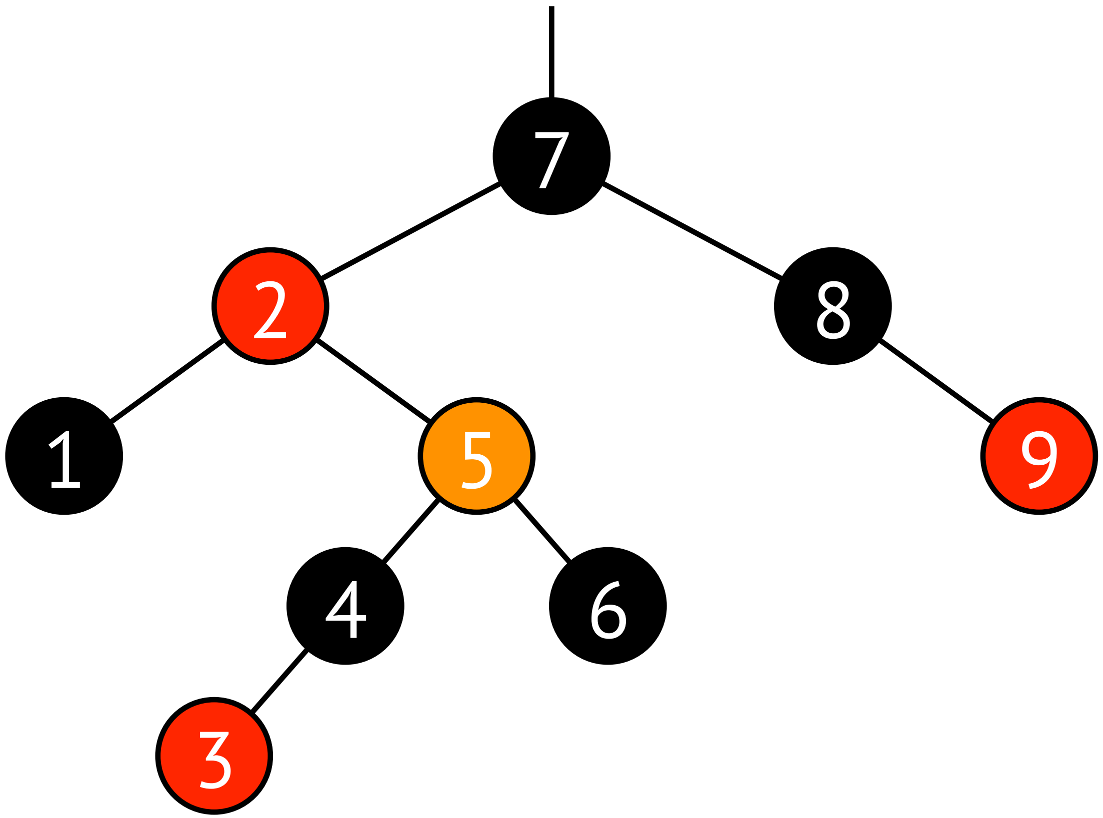
Fall 1: Der Onkel ist rot → umfärben und weiter mit Großvater



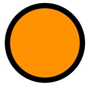
Fall 1: Der Onkel ist rot → umfärben und weiter mit Großvater

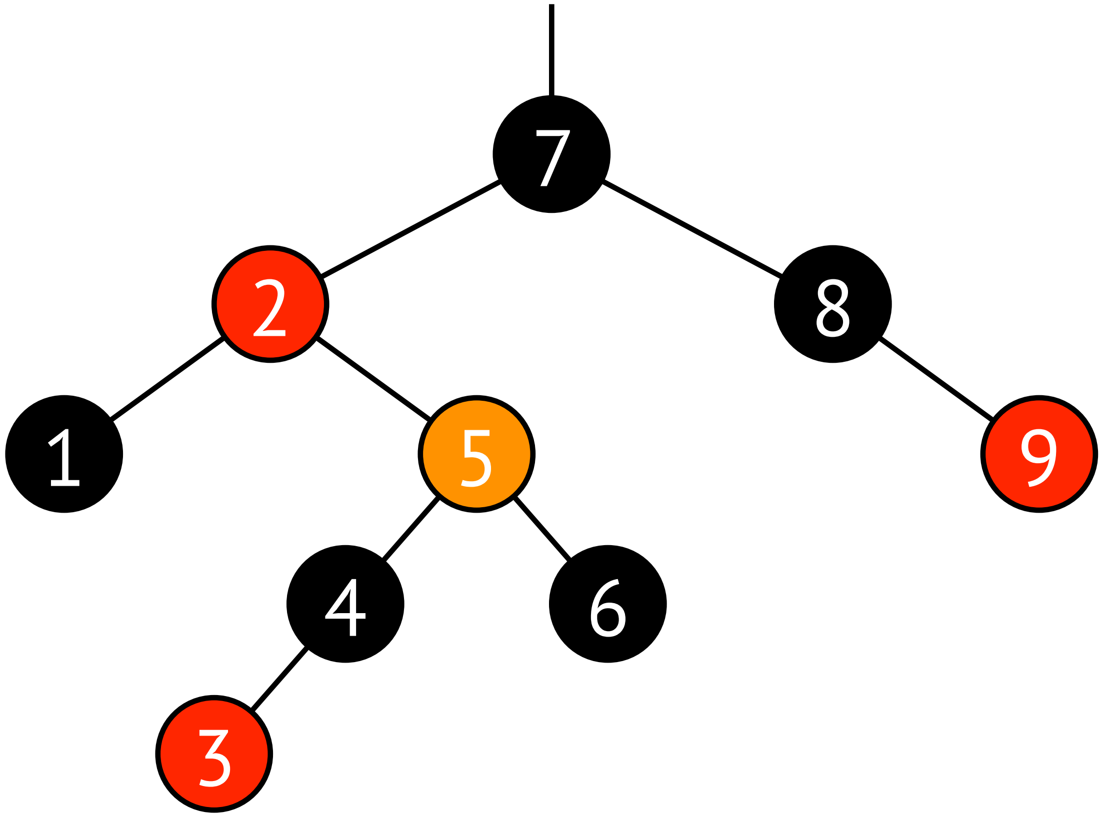


Rot-Schwarz-Bäume: Insert()



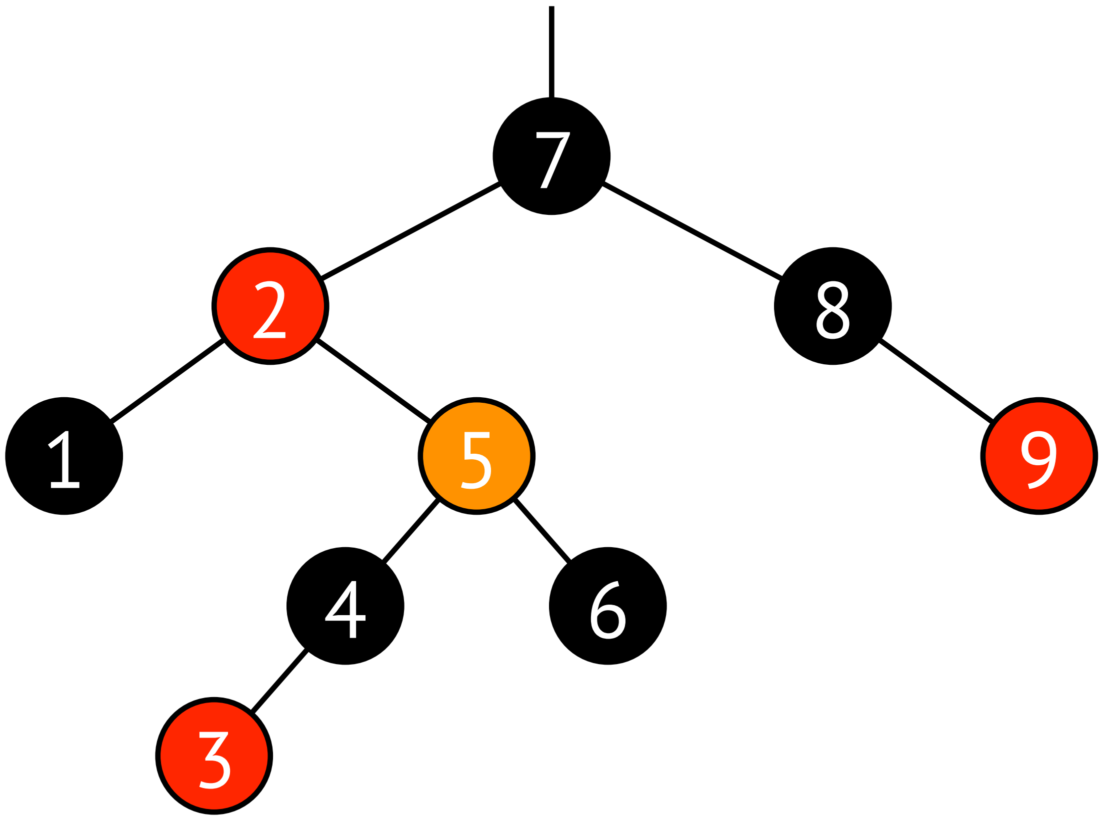
Rot-Schwarz-Bäume: Insert()

Fall 2: Der Onkel ist schwarz und  ist ein innerer Sohn

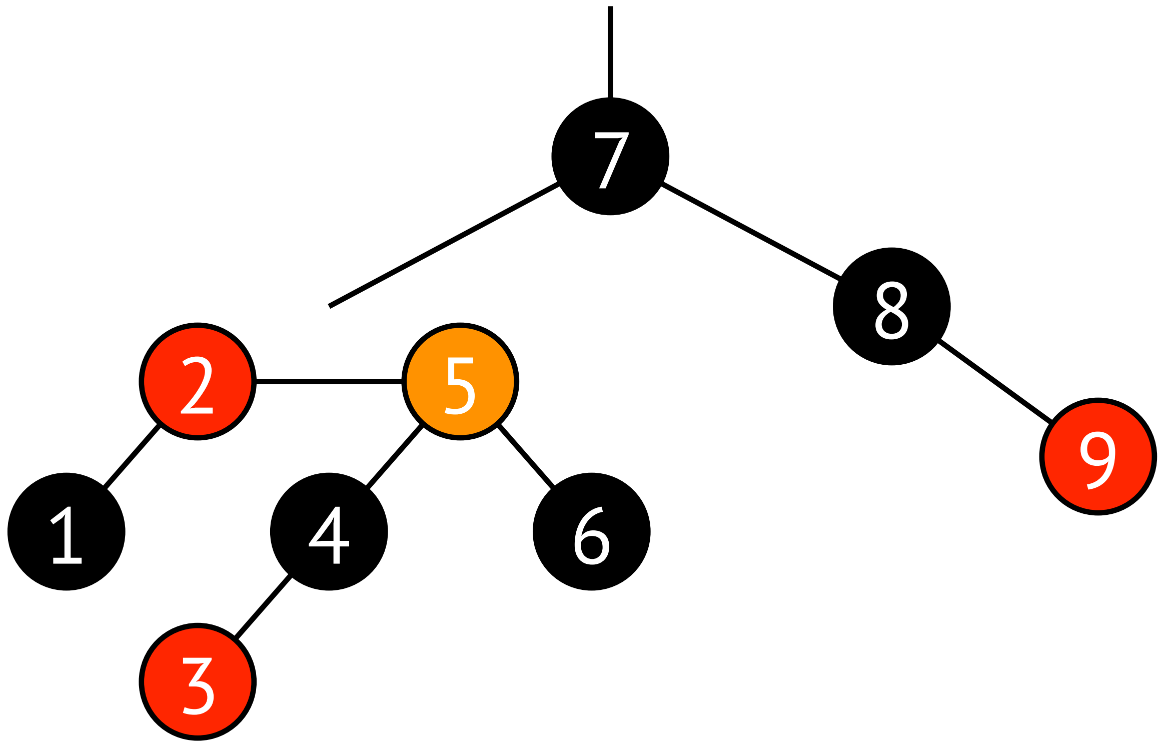


Rot-Schwarz-Bäume: Insert()

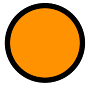
Fall 2: Der Onkel ist schwarz und ● ist ein innerer Sohn
→ Rotation nach außen

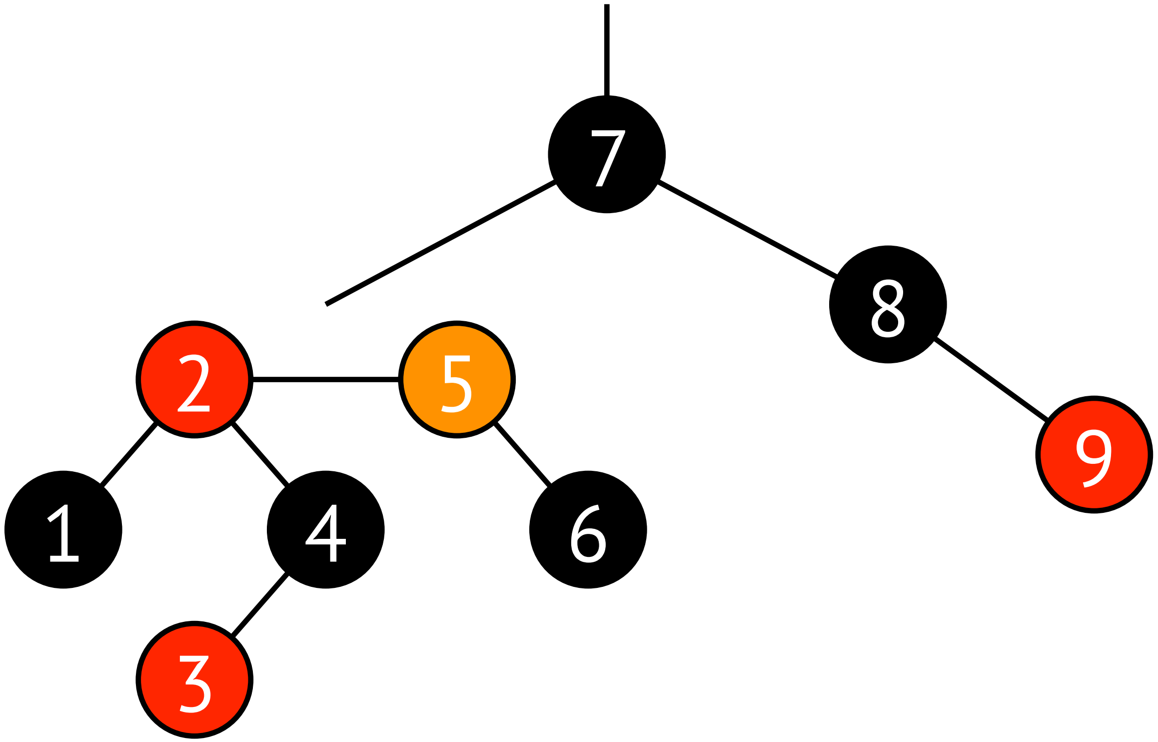


Fall 2: Der Onkel ist schwarz und ● ist ein innerer Sohn
→ Rotation nach außen



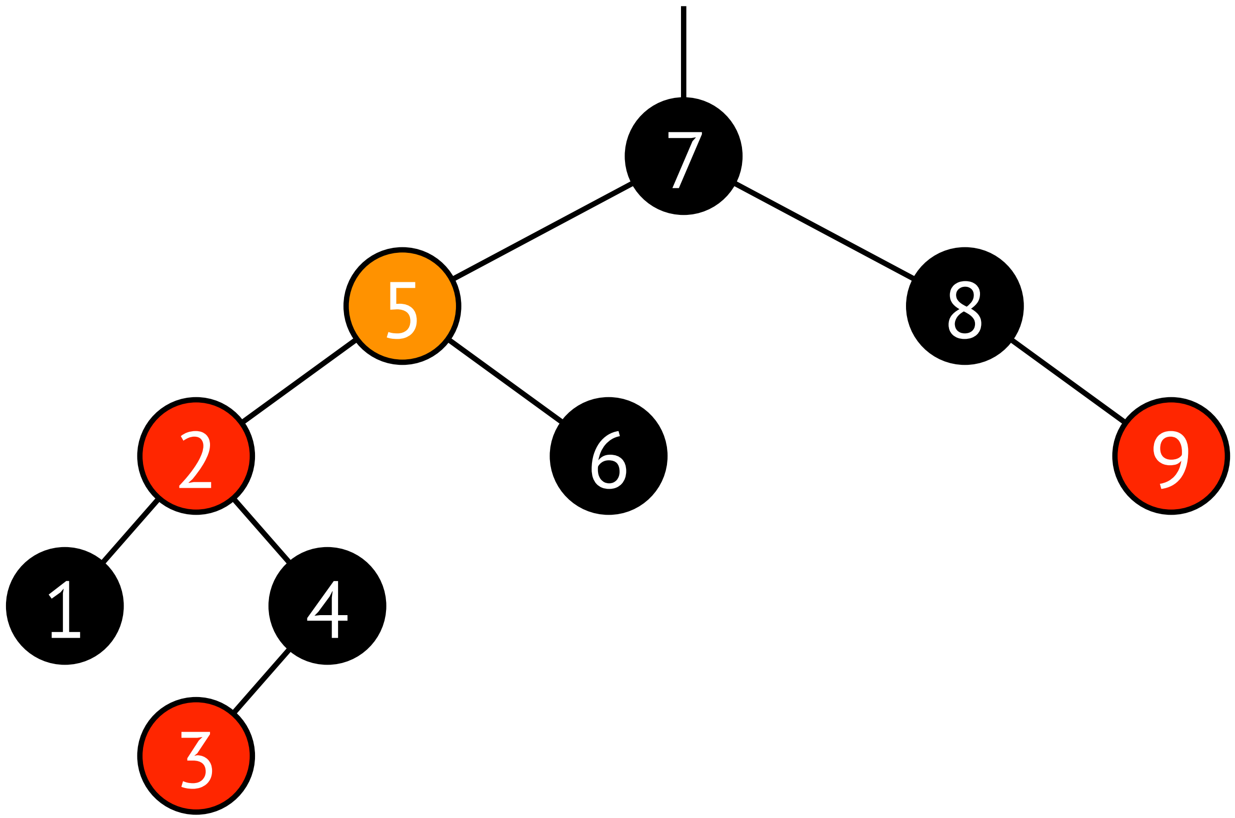
Rot-Schwarz-Bäume: Insert()

Fall 2: Der Onkel ist schwarz und  ist ein innerer Sohn
→ Rotation nach außen



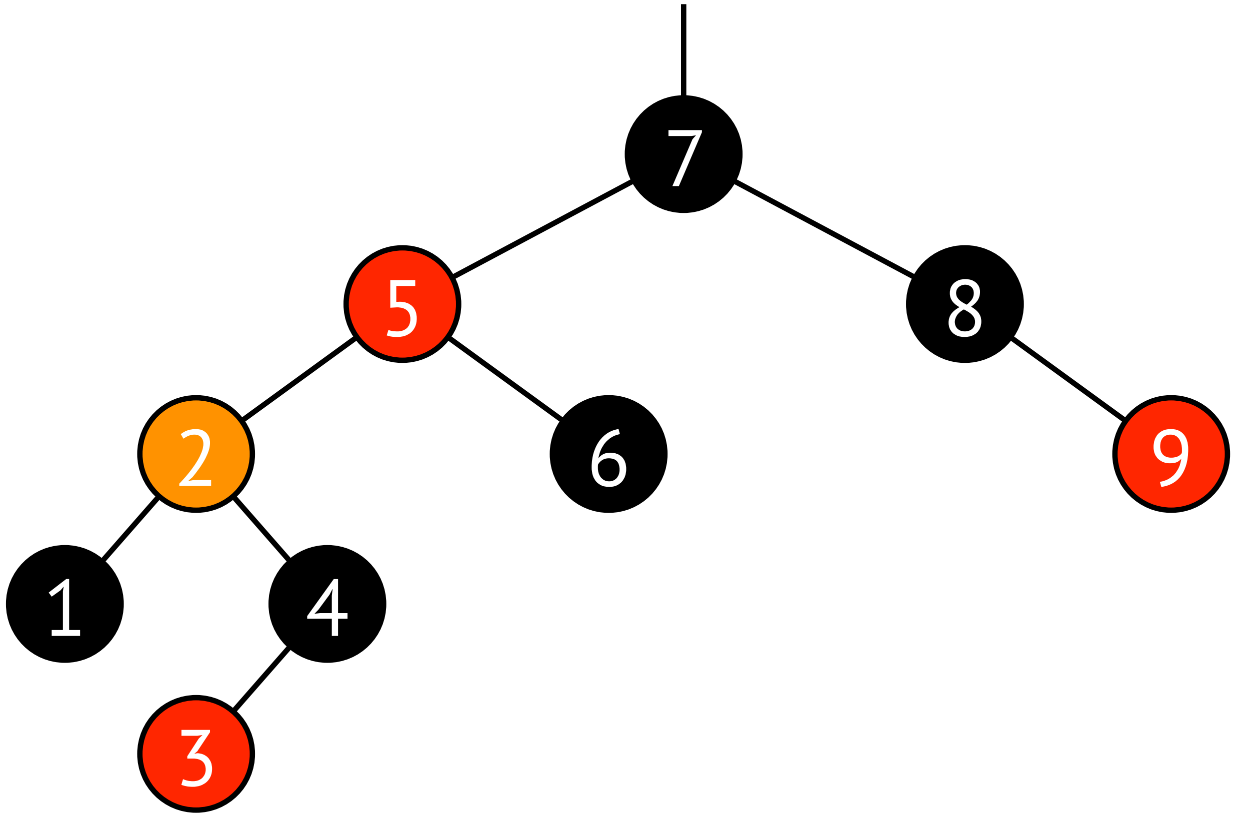
Rot-Schwarz-Bäume: Insert()

Fall 2: Der Onkel ist schwarz und ist ein innerer Sohn
→ Rotation nach außen



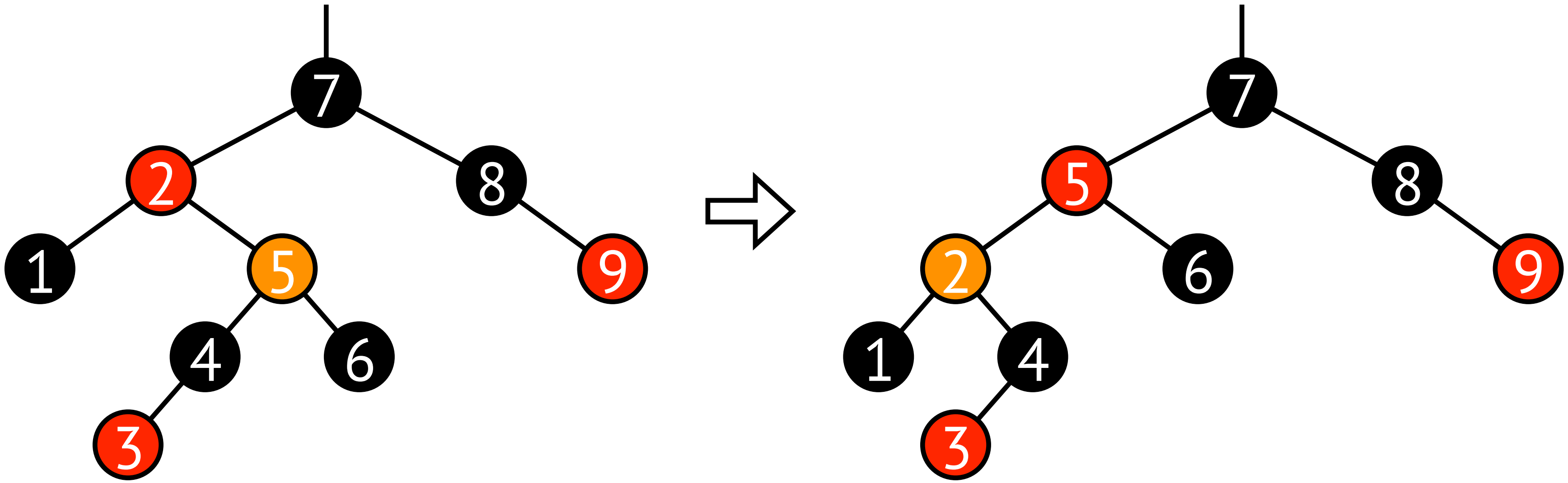
Rot-Schwarz-Bäume: Insert()

Fall 2: Der Onkel ist schwarz und ● ist ein innerer Sohn
→ Rotation nach außen und weiter mit äußeren Sohn (Fall 3)



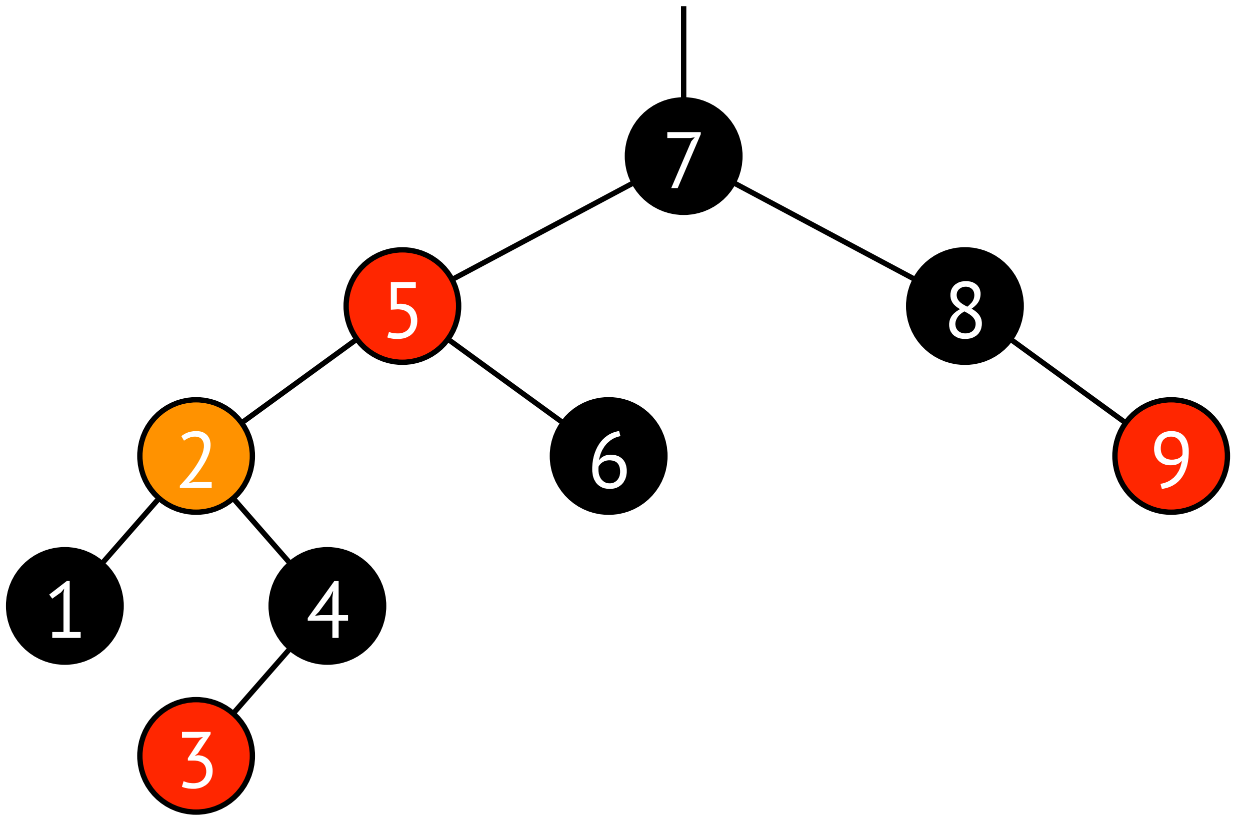
Rot-Schwarz-Bäume: Insert()

Fall 2: Der Onkel ist schwarz und ● ist ein innerer Sohn
→ Rotation nach außen und weiter mit äußeren Sohn (Fall 3)



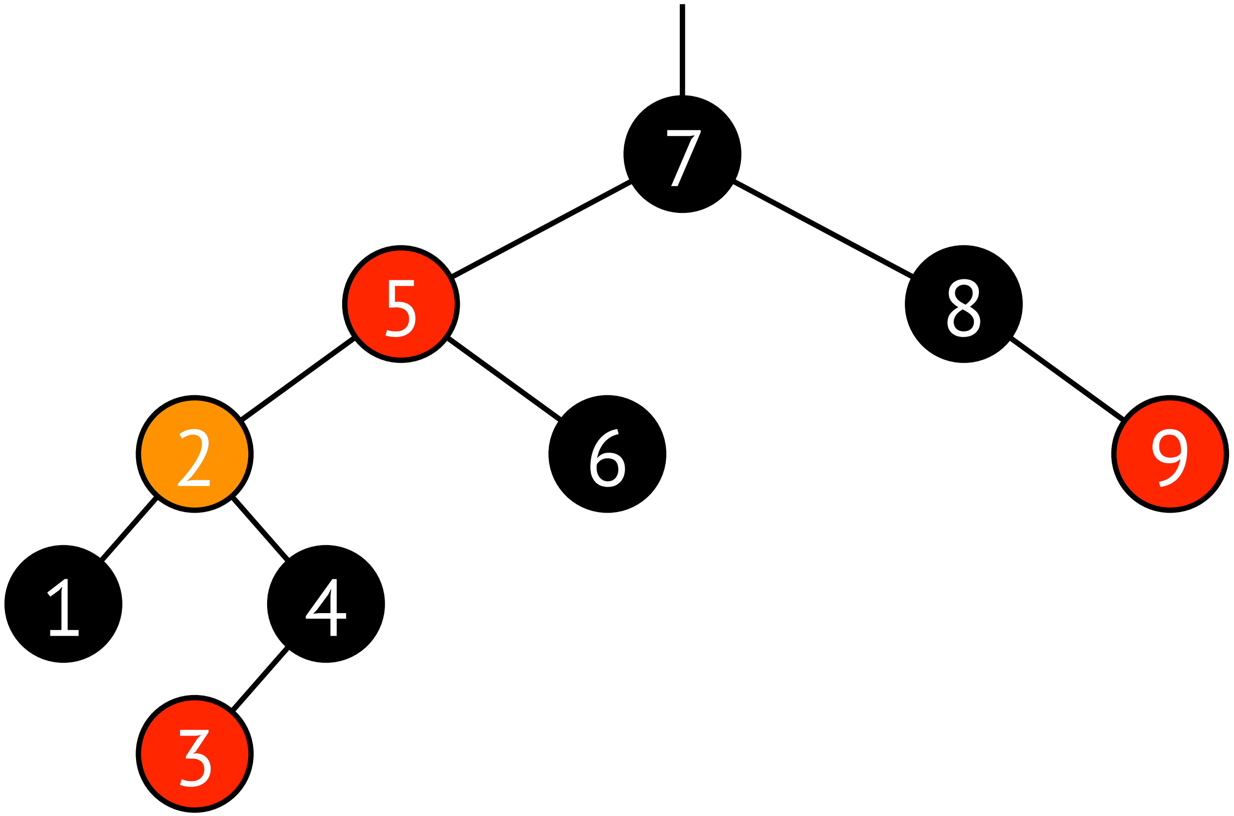
Rot-Schwarz-Bäume: Insert()

Fall 3: Der Onkel ist schwarz und ⬤ ist ein äußerer Sohn

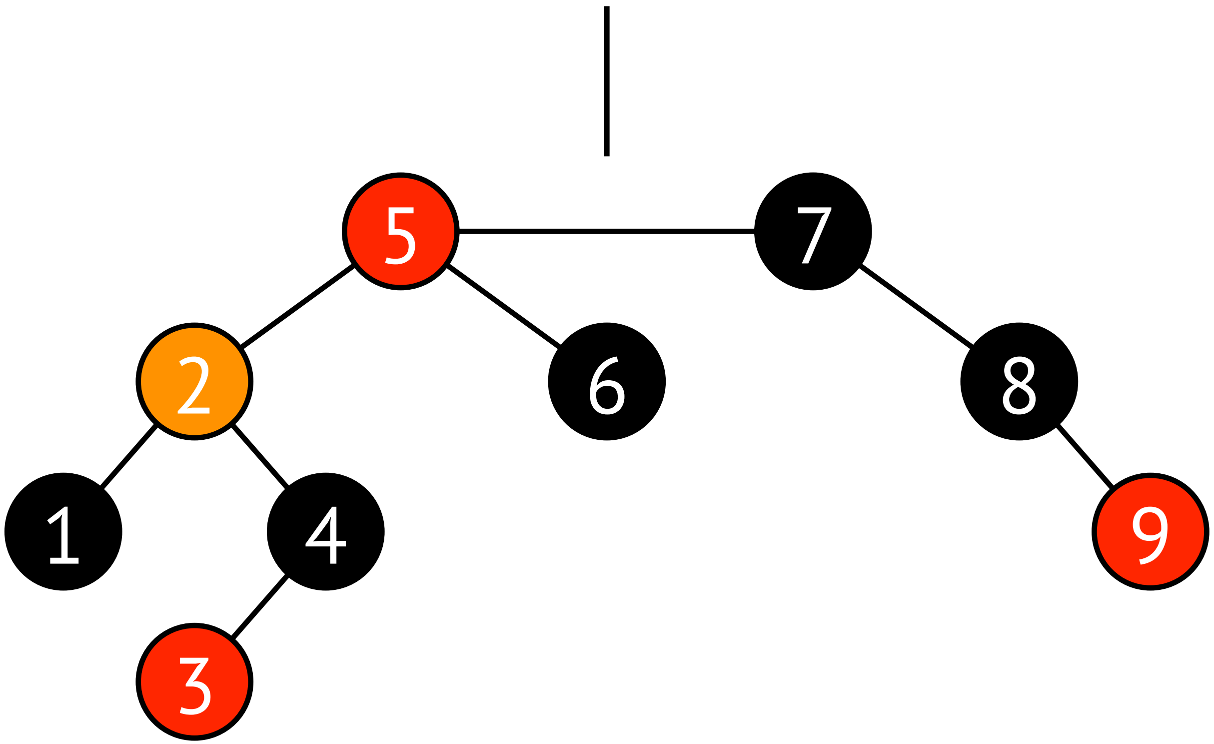


Rot-Schwarz-Bäume: Insert()

Fall 3: Der Onkel ist schwarz und ◯ ist ein äußerer Sohn
→ Rotation nach innen

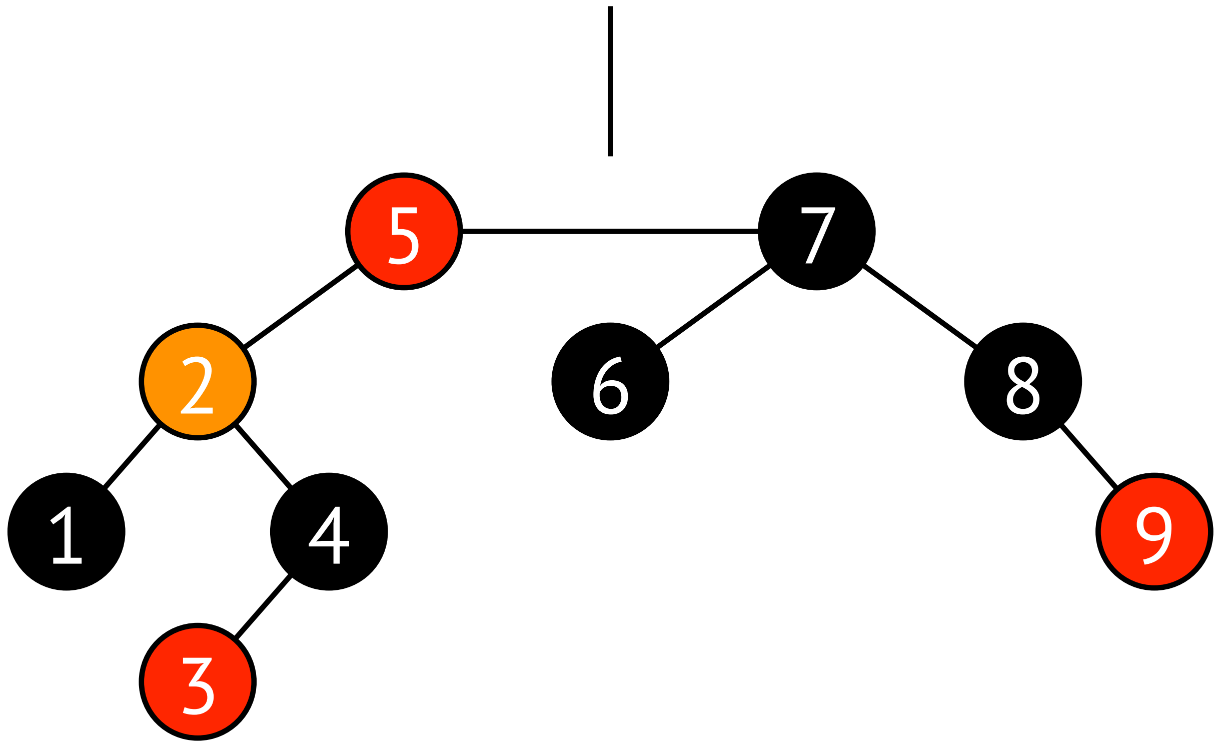


Fall 3: Der Onkel ist schwarz und ◯ ist ein äußerer Sohn
→ Rotation nach innen



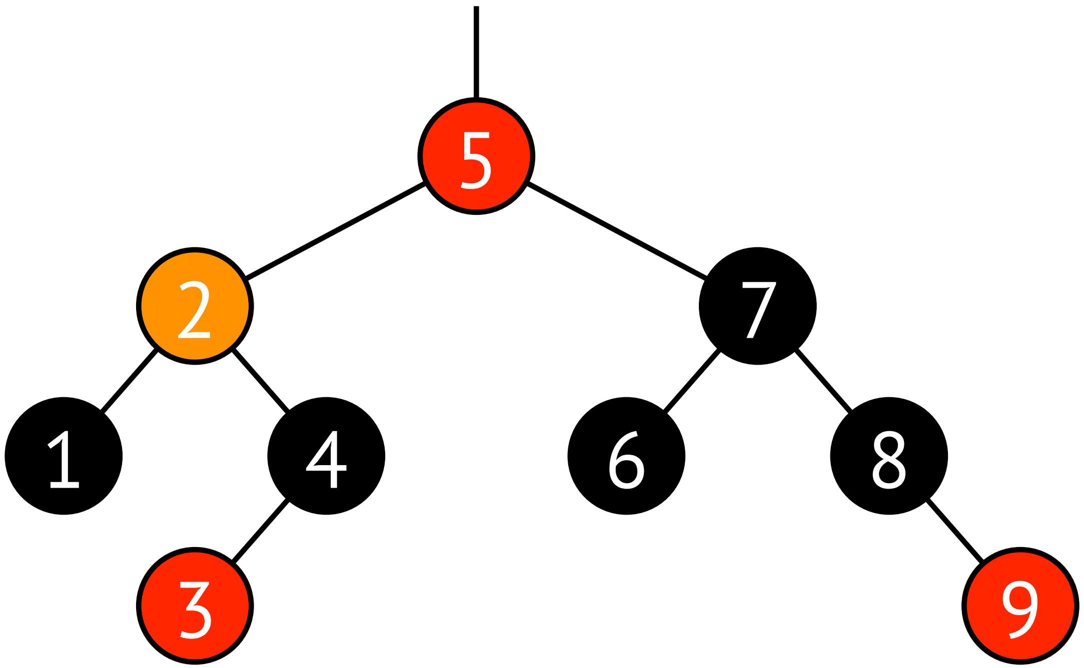
Rot-Schwarz-Bäume: Insert()

Fall 3: Der Onkel ist schwarz und ⬤ ist ein äußerer Sohn
→ Rotation nach innen



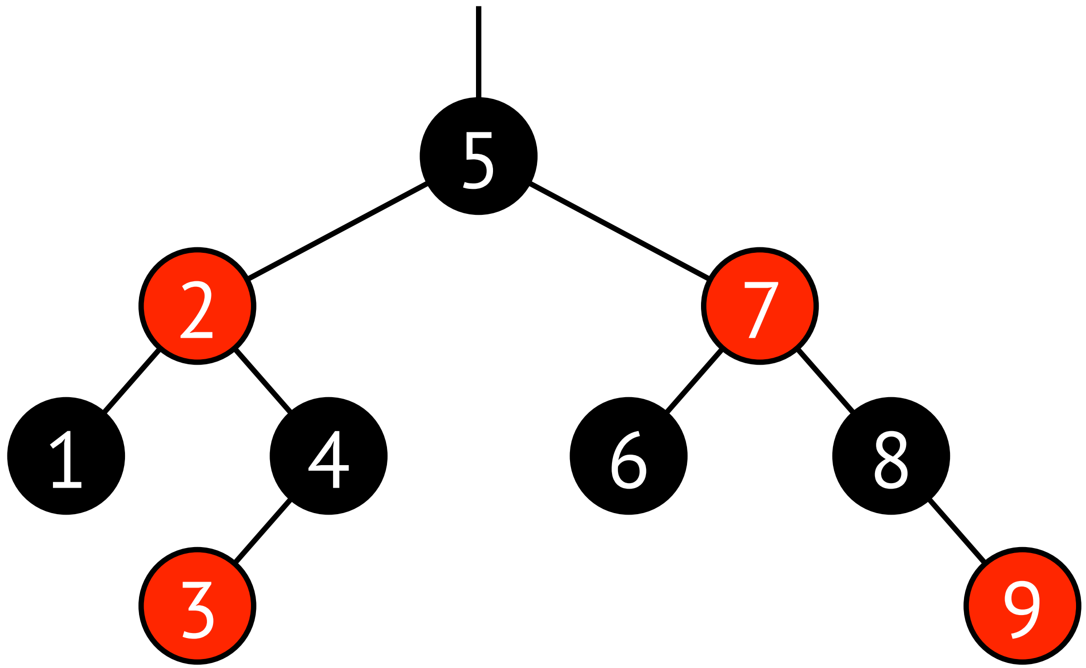
Rot-Schwarz-Bäume: Insert()

Fall 3: Der Onkel ist schwarz und ist ein äußerer Sohn
→ Rotation nach innen



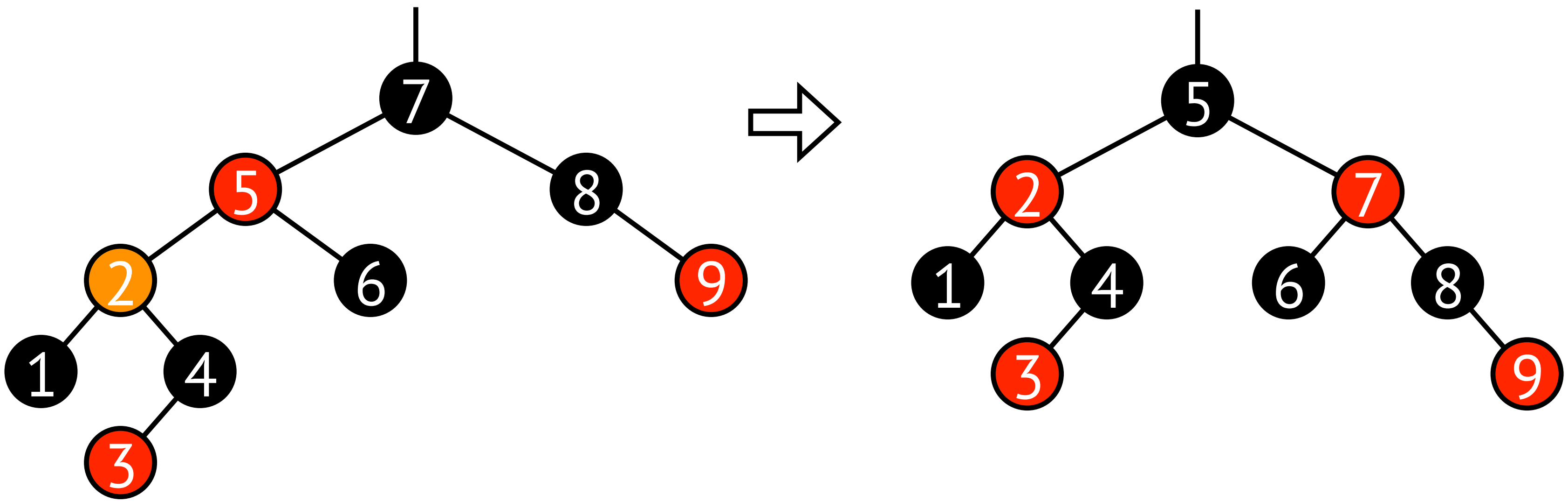
Rot-Schwarz-Bäume: Insert()

Fall 3: Der Onkel ist schwarz und ⬤ ist ein äußerer Sohn
→ Rotation nach innen und umfärben



Rot-Schwarz-Bäume: Insert()

Fall 3: Der Onkel ist schwarz und ◯ ist ein äußerer Sohn
→ Rotation nach innen und umfärben



Fallunterscheidung Insert()

1. Onkel ist rot
   – Umfärben, Rekursion nach oben
2. Sohn < Vater > Großvater oder
   Sohn > Vater < Großvater
   – Rotation, weiter mit Fall 3
3. Sohn < Vater < Großvater oder
   Sohn > Vater > Großvater
   – Rotation, Umfärben, Fertig

Aufwand Insert()

- Suchen der Einfügestelle O(log n)
- Einfügen O(1)
- Konsistenzwiederherstellung
  - Umfärbe-Schritte O(log n)
    (Schritt vom Enkel zum Großvater)
  - Rotationen O(1) (maximal zwei)
- Insgesamt: O(log n)

Rot-Schwarz-Bäume: Delete()

- Wie bei normalen binären Suchbäumen wird der Knoten entweder direkt gelöscht oder sein Successor wird gelöscht und der Inhalt wird kopiert.

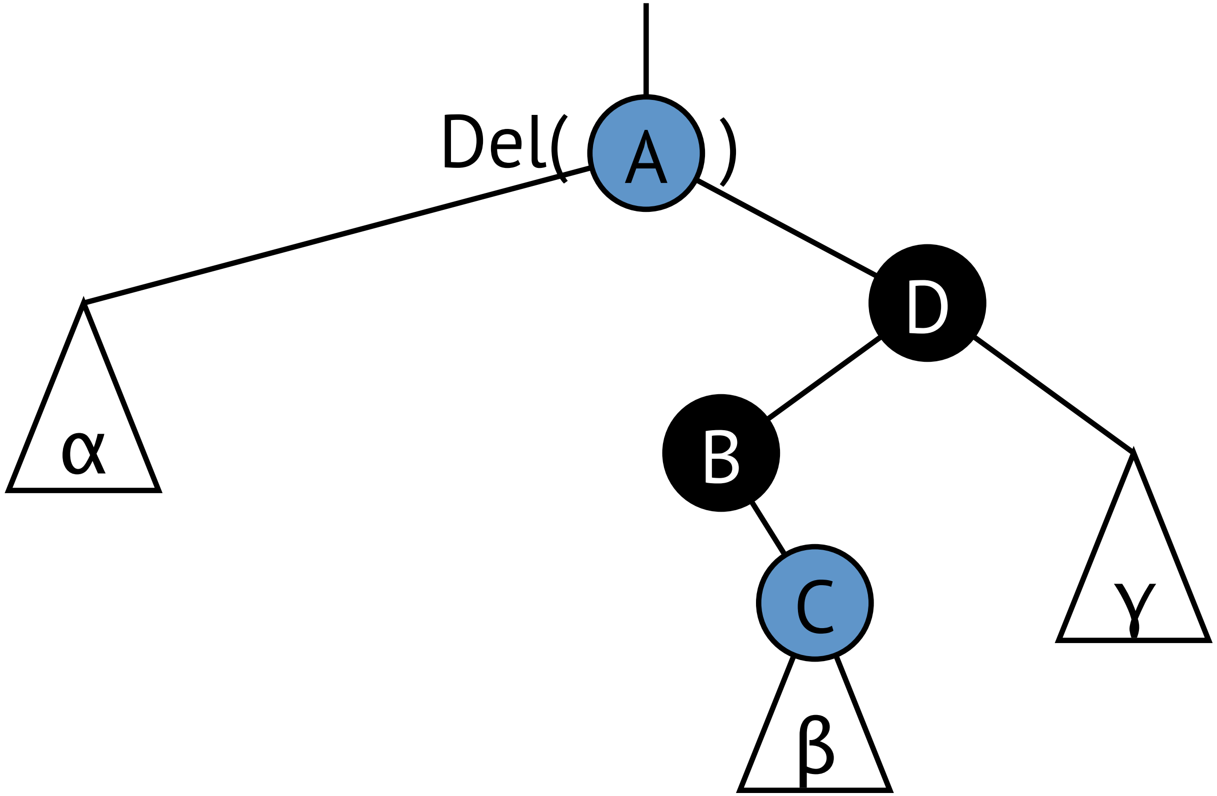
- Konsistenzwiederherstellung
  - Wenn der Knoten rot war, ist nichts zu tun
  - Wenn der Knoten schwarz war, gibt es **4** Fälle

Rot-Schwarz-Bäume: Delete()

- Falls der **tatsächlich** entfernte Knoten schwarz ist, fehlt auf allen Pfaden innerhalb des entsprechenden Teil-Baums eine schwarze Marke
- Weise diese schwarze Marke dem (einen) Nachfolger des gelöschten Knotens zu. Dadurch wird dieser **schwarz-rot** oder **doppel-schwarz**.

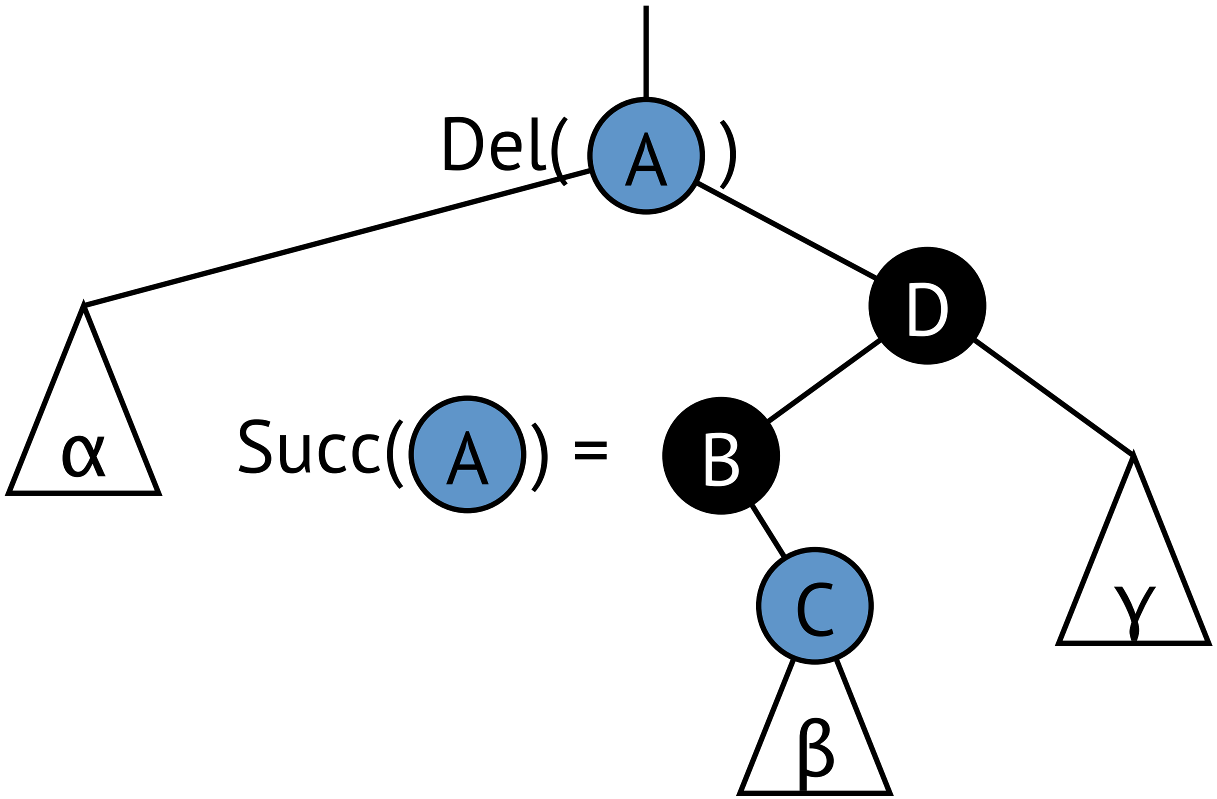
Rot-Schwarz-Bäume: Delete()

- Erinnerung



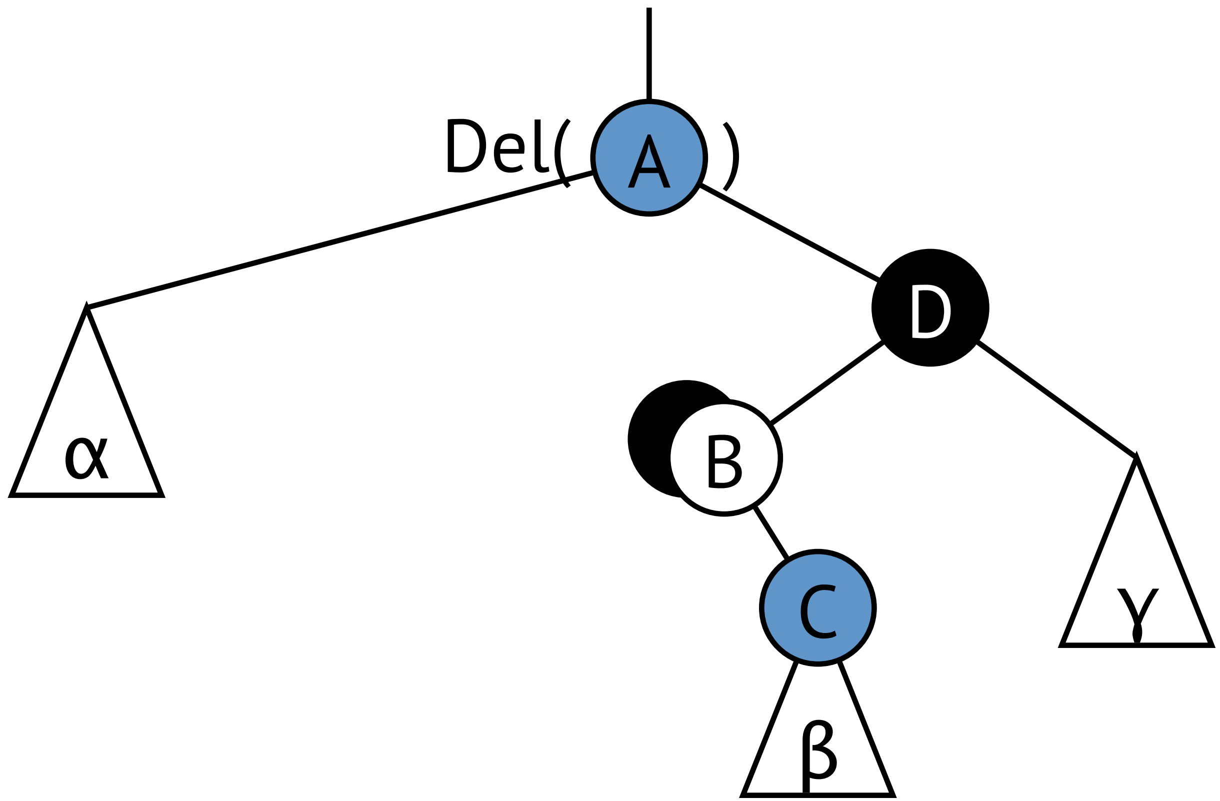
Rot-Schwarz-Bäume: Delete()

- Erinnerung



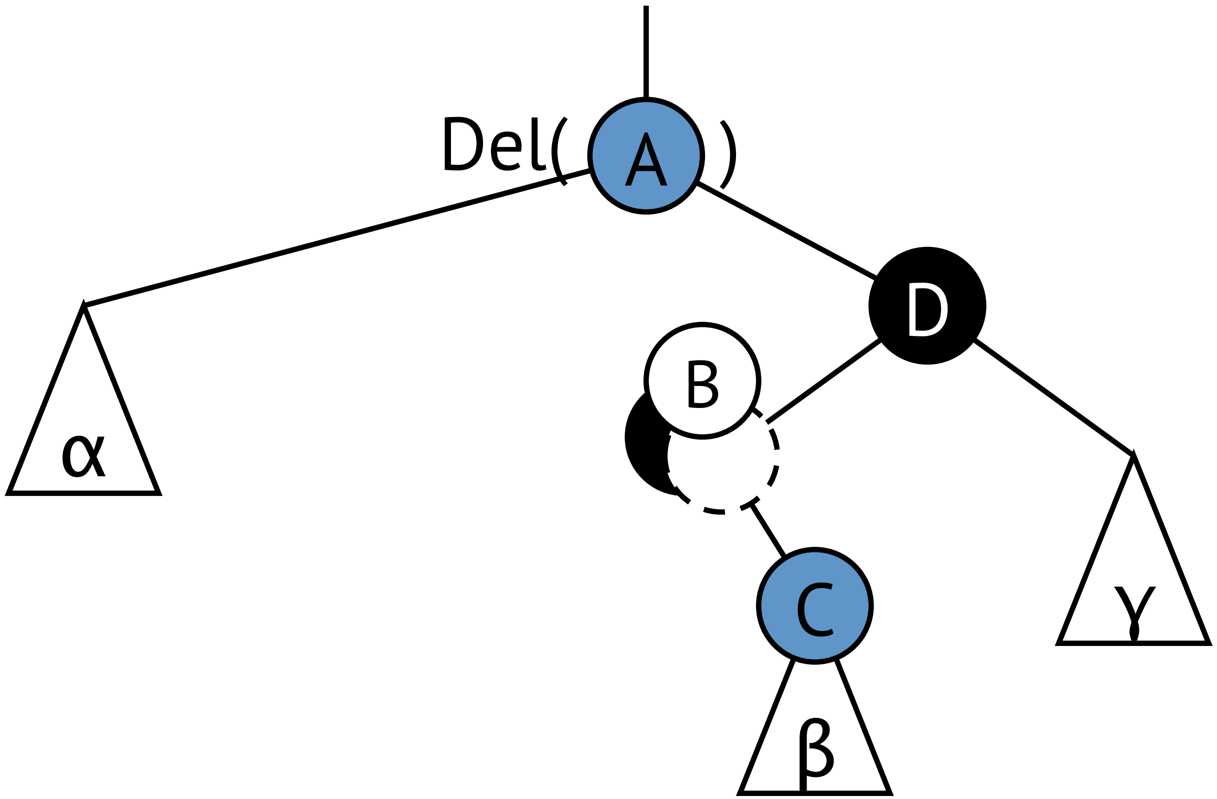
Rot-Schwarz-Bäume: Delete()

- Erinnerung



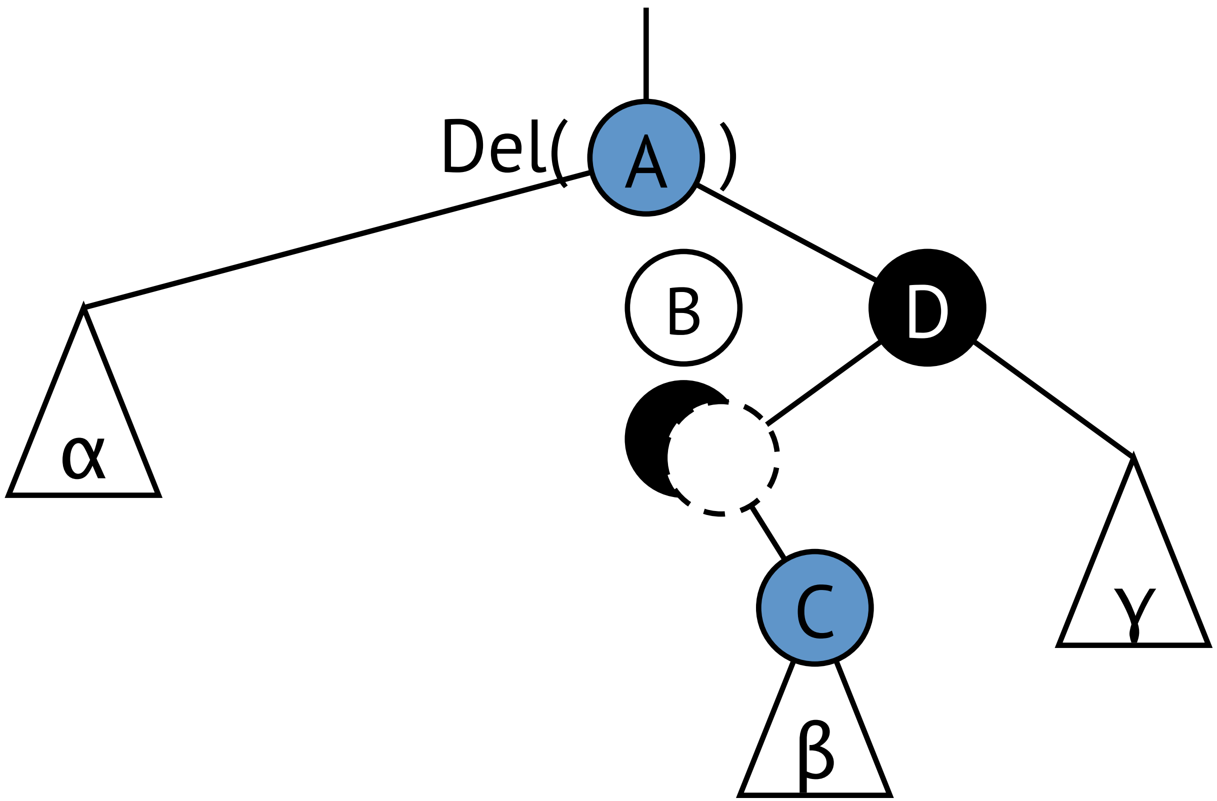
Rot-Schwarz-Bäume: Delete()

- Erinnerung



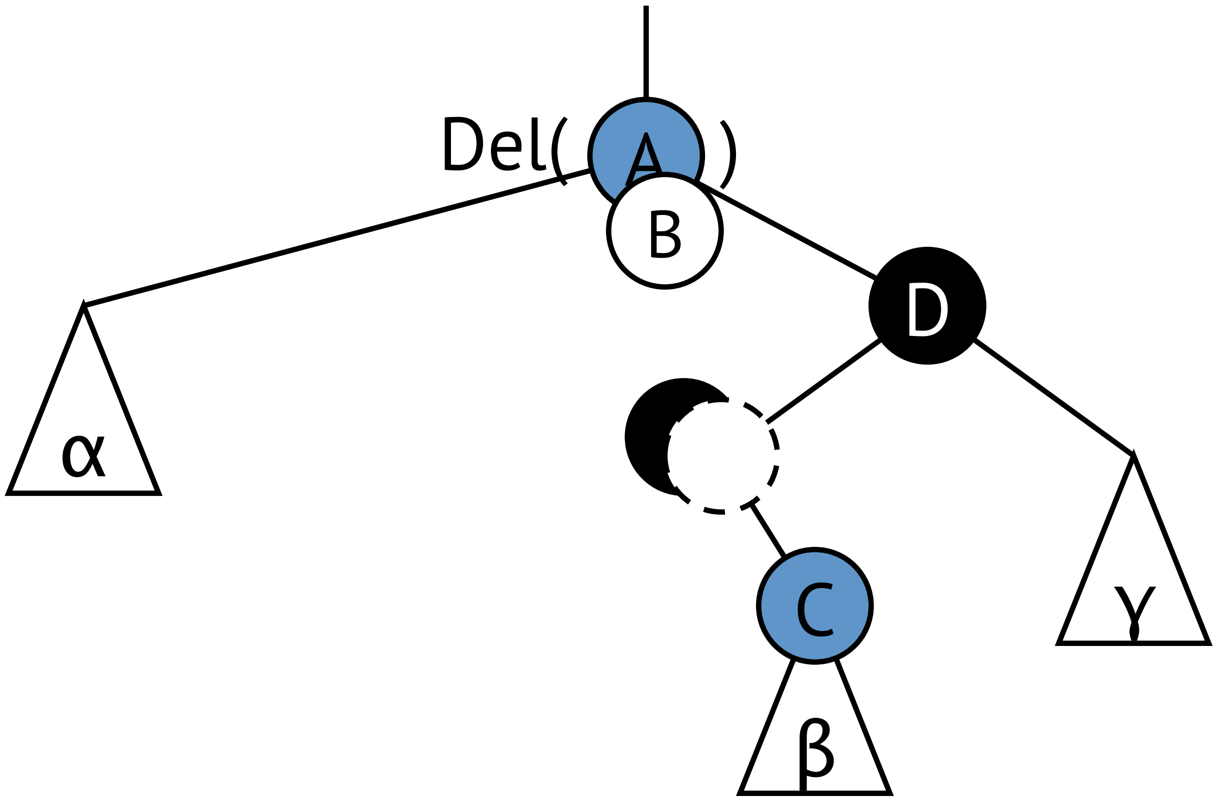
Rot-Schwarz-Bäume: Delete()

- Erinnerung



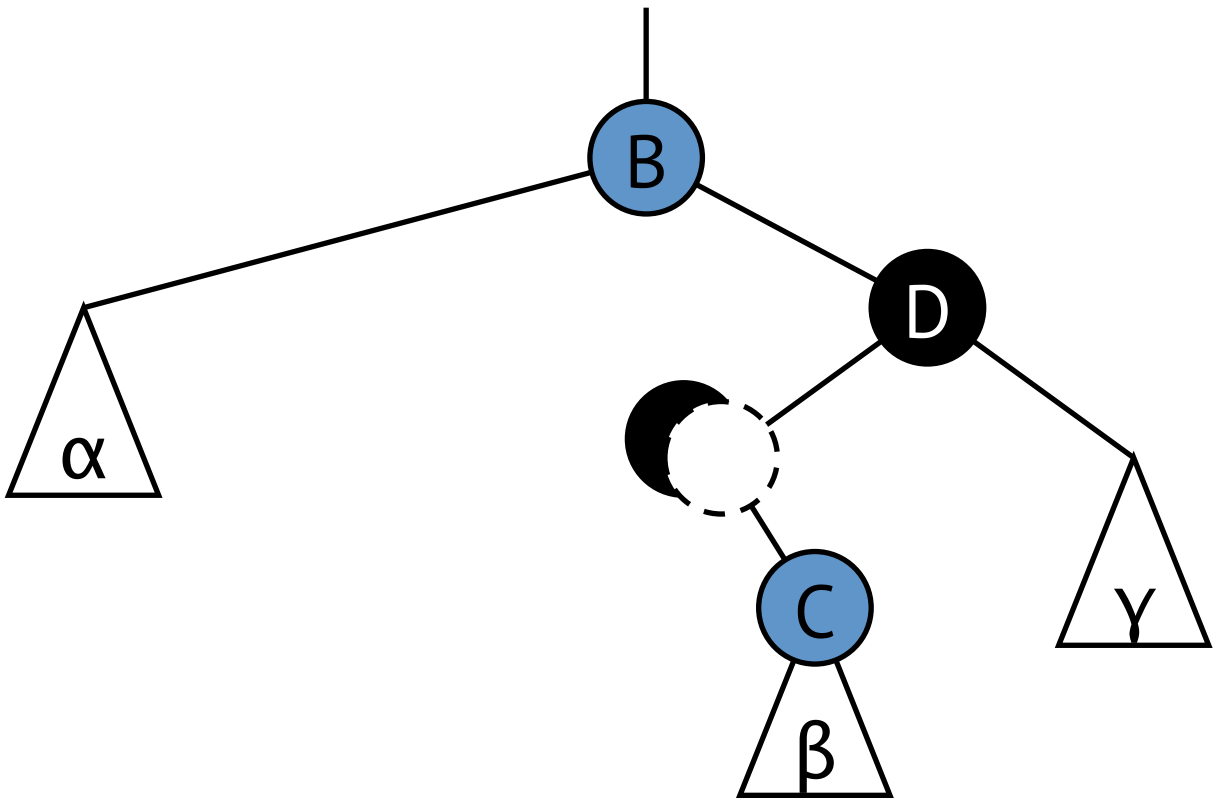
Rot-Schwarz-Bäume: Delete()

- Erinnerung



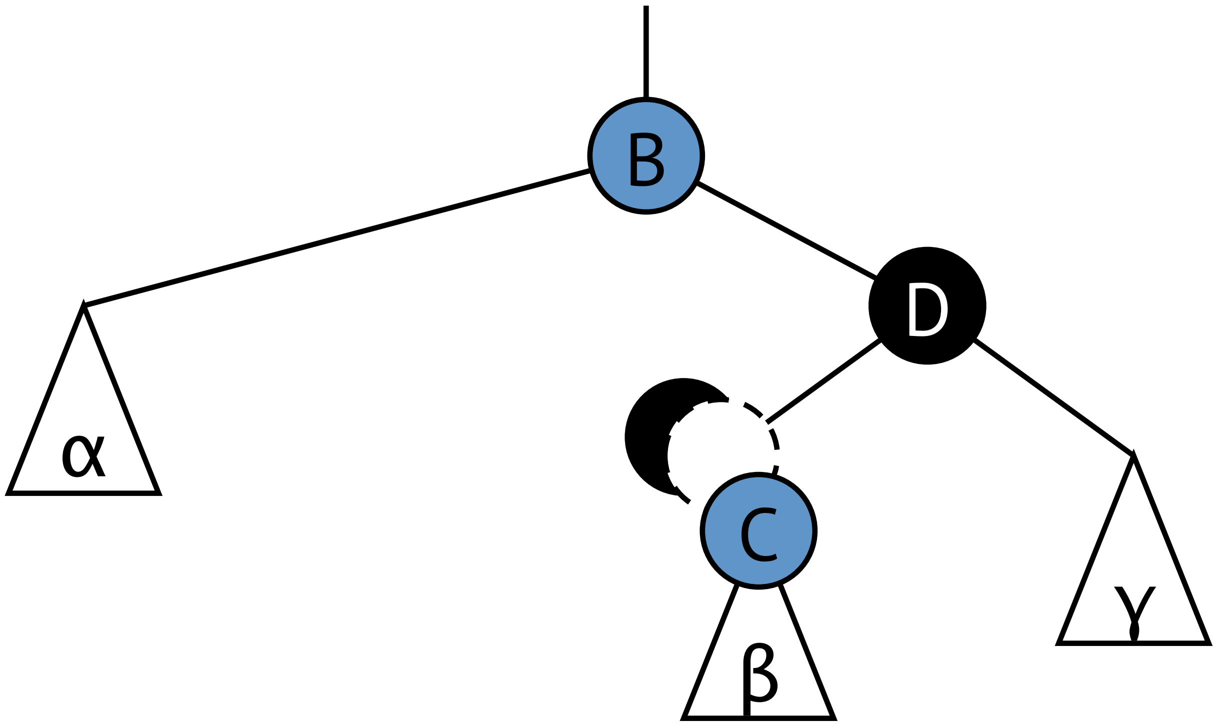
Rot-Schwarz-Bäume: Delete()

- Erinnerung



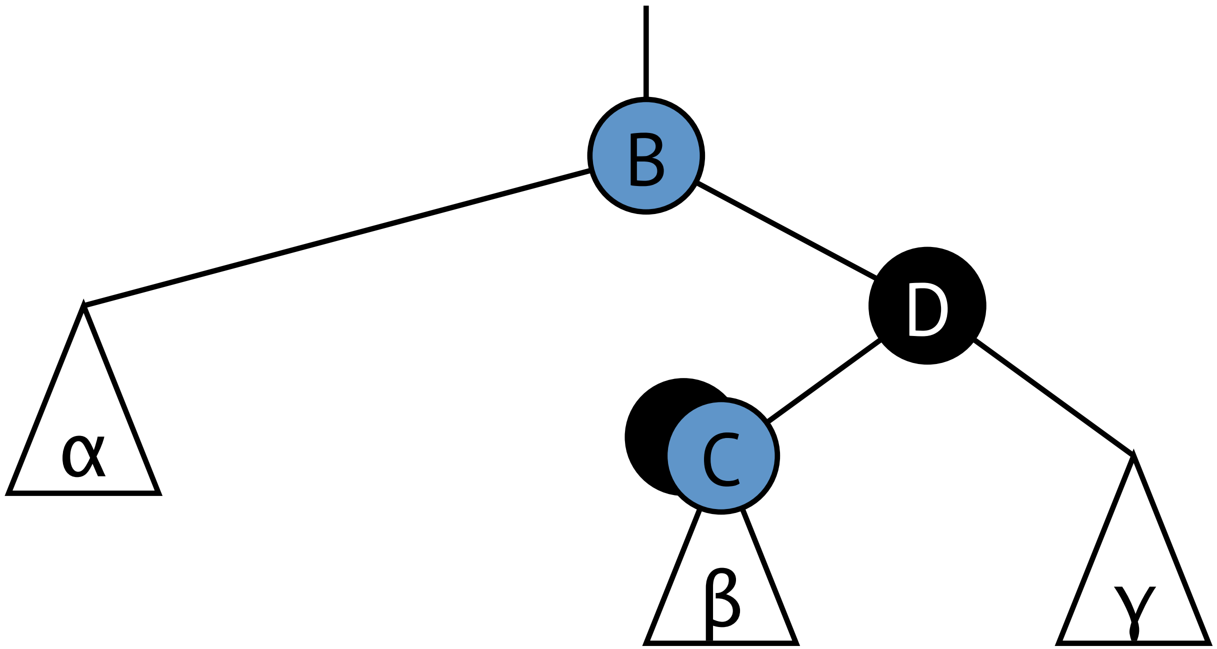
Rot-Schwarz-Bäume: Delete()

- Erinnerung



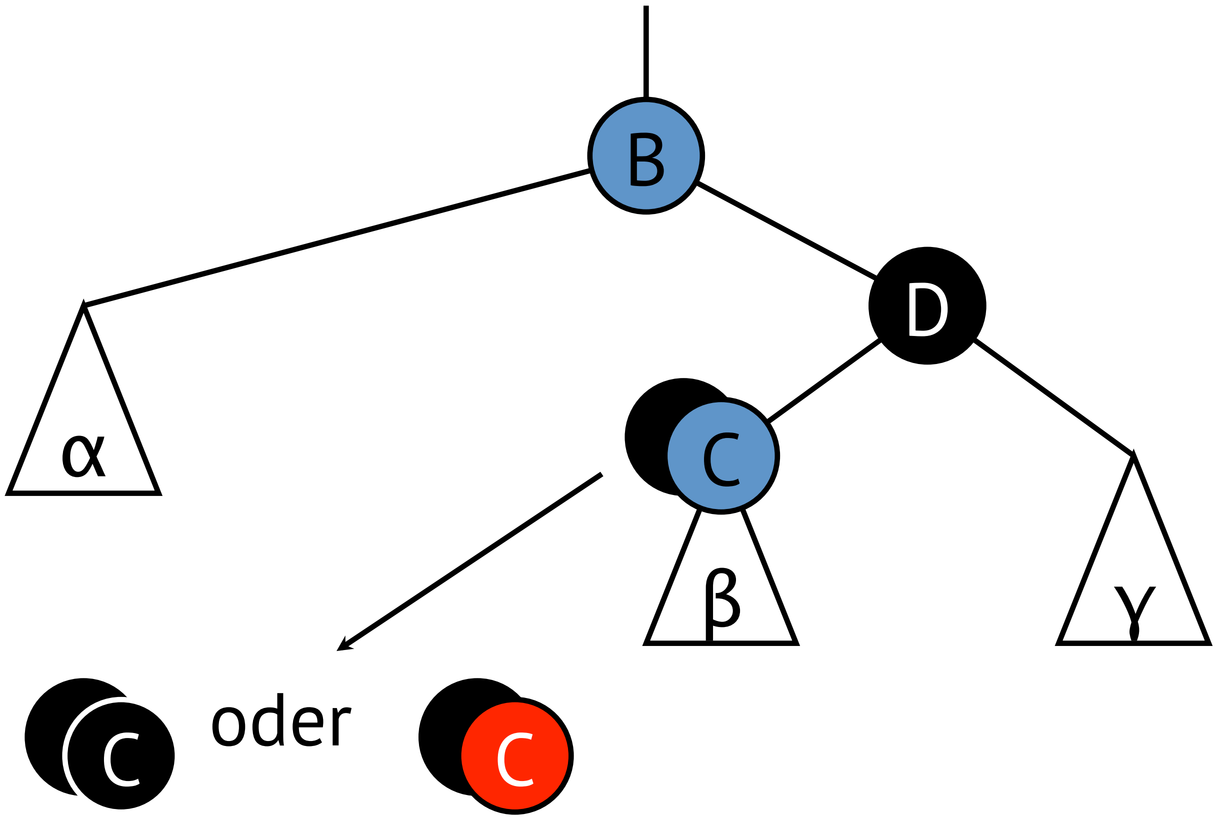
Rot-Schwarz-Bäume: Delete()

- Erinnerung



Rot-Schwarz-Bäume: Delete()

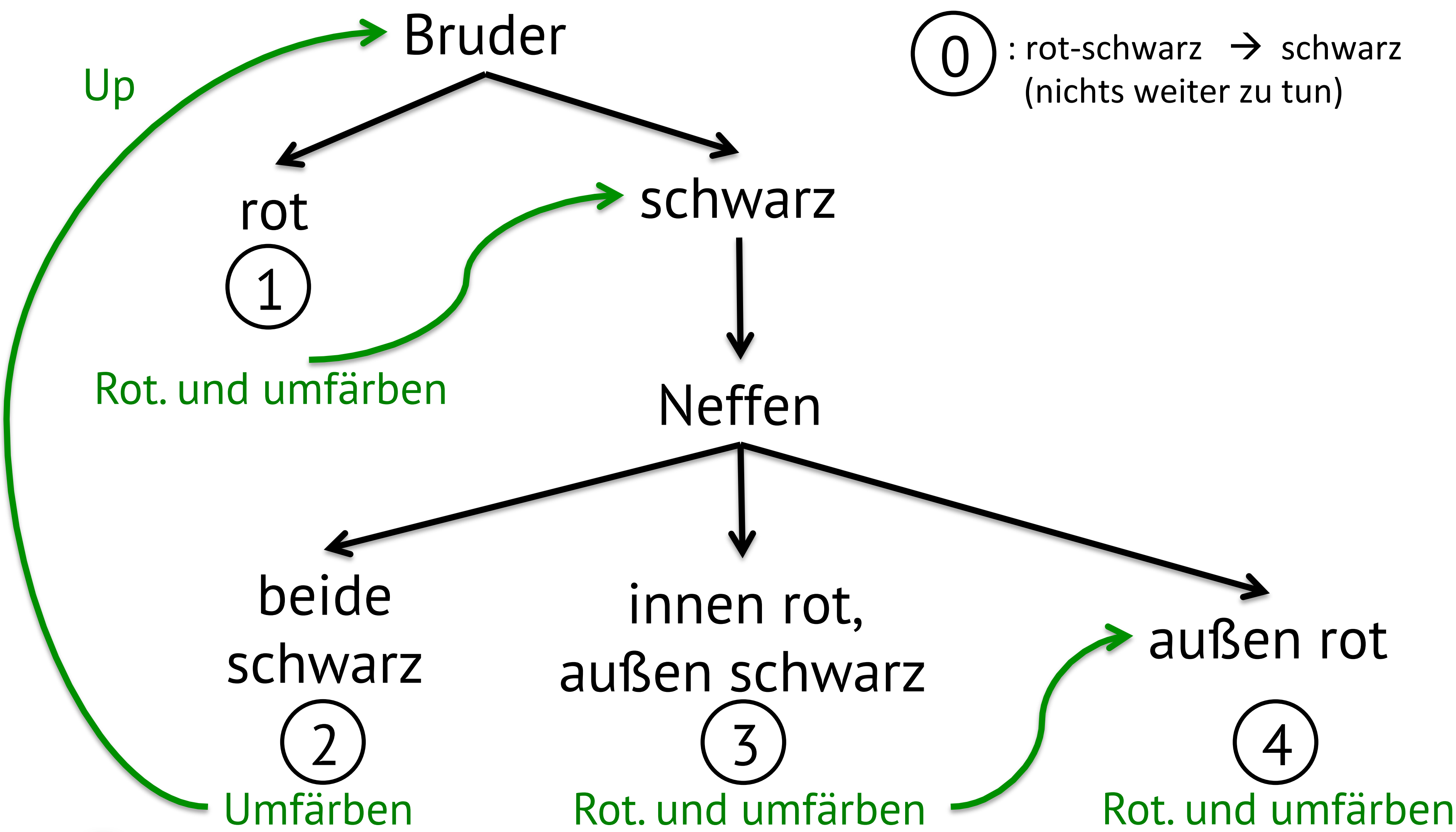
- Erinnerung



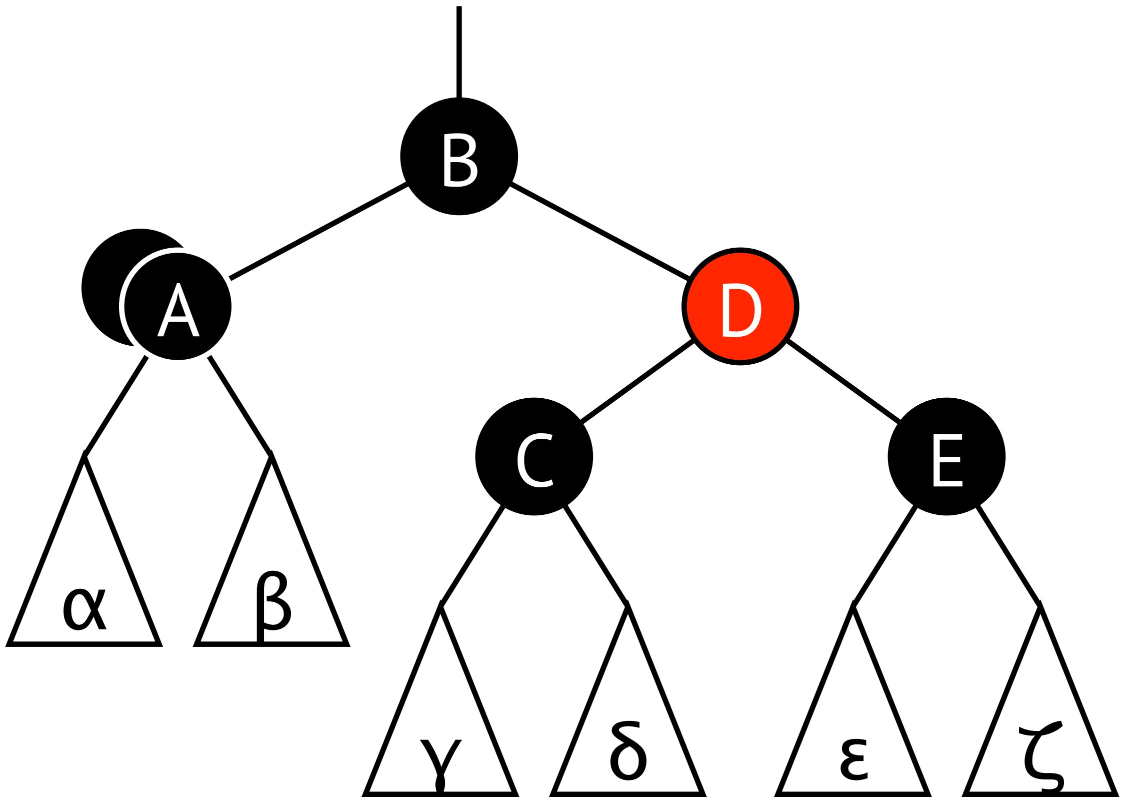
Rot-Schwarz-Bäume: Delete()

- Umfärben und Rotationen
  - Erhalte die Zahl schwarzer Marken der Sub-Bäume
- Abbruchbedingungen
  - Ein **schwarz-roter** Knoten wird **schwarz** gefärbt
  - Ein Wurzelknoten wird erreicht (**doppel-schwarz** kann einfach wegfallen, da die Wurzel auf allen Pfaden liegt)

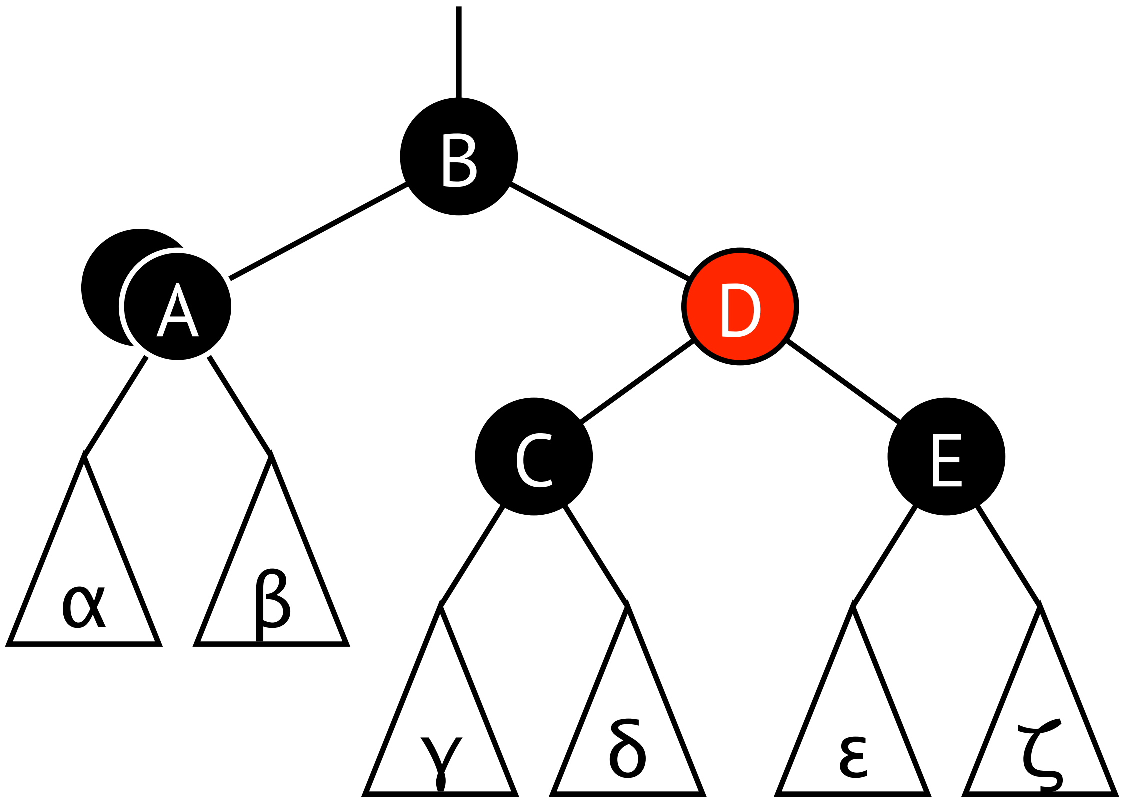
Rot-Schwarz-Bäume – Delete - Scheme



Fall 1: Der Bruder ist rot

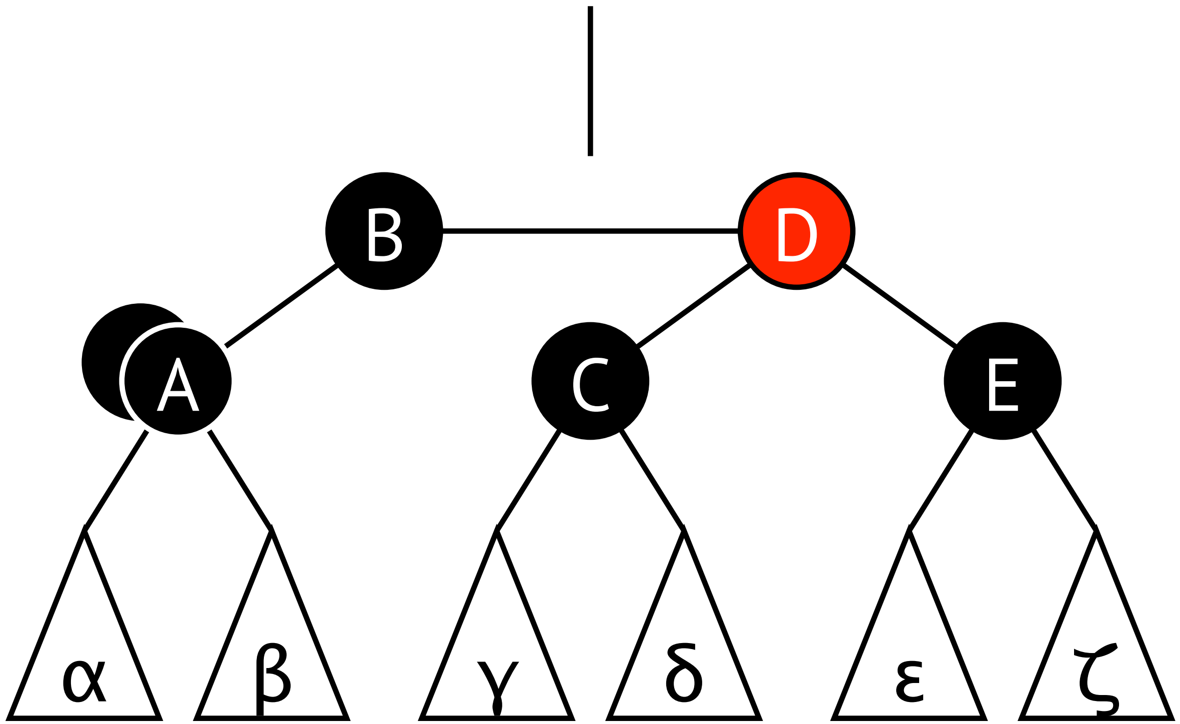


Fall 1: Der Bruder ist rot → Rotation nach links



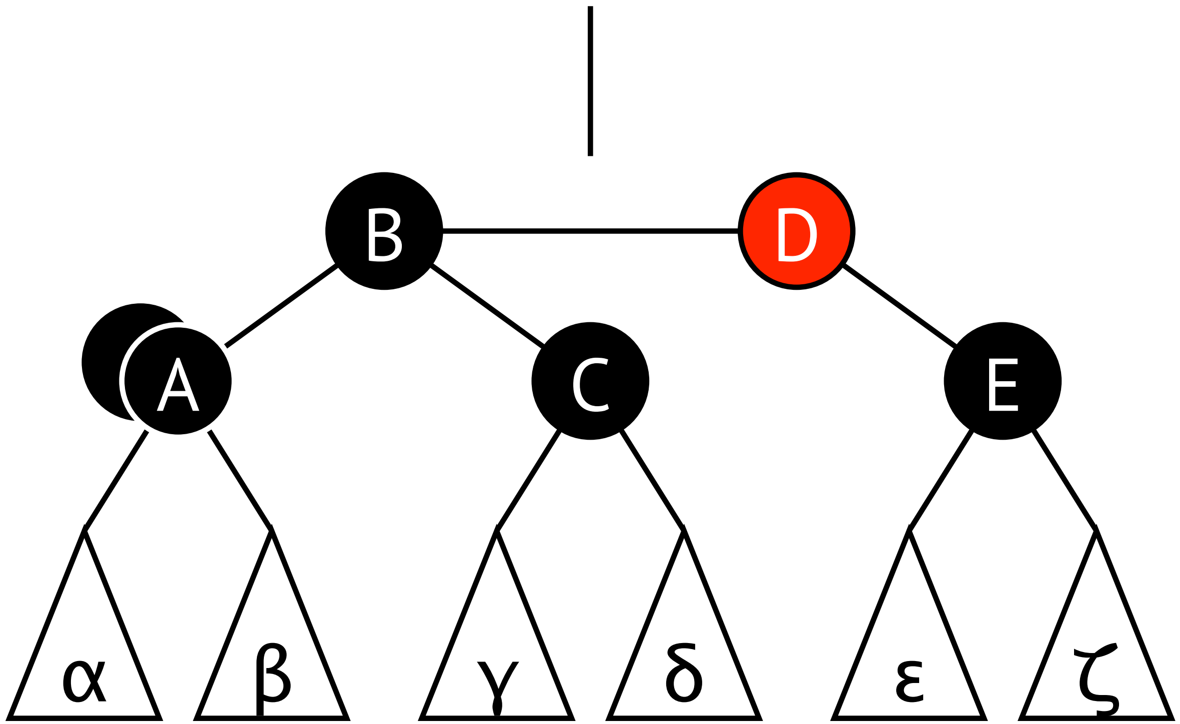
Rot-Schwarz-Bäume: Delete()

Fall 1: Der Bruder ist rot → Rotation nach links



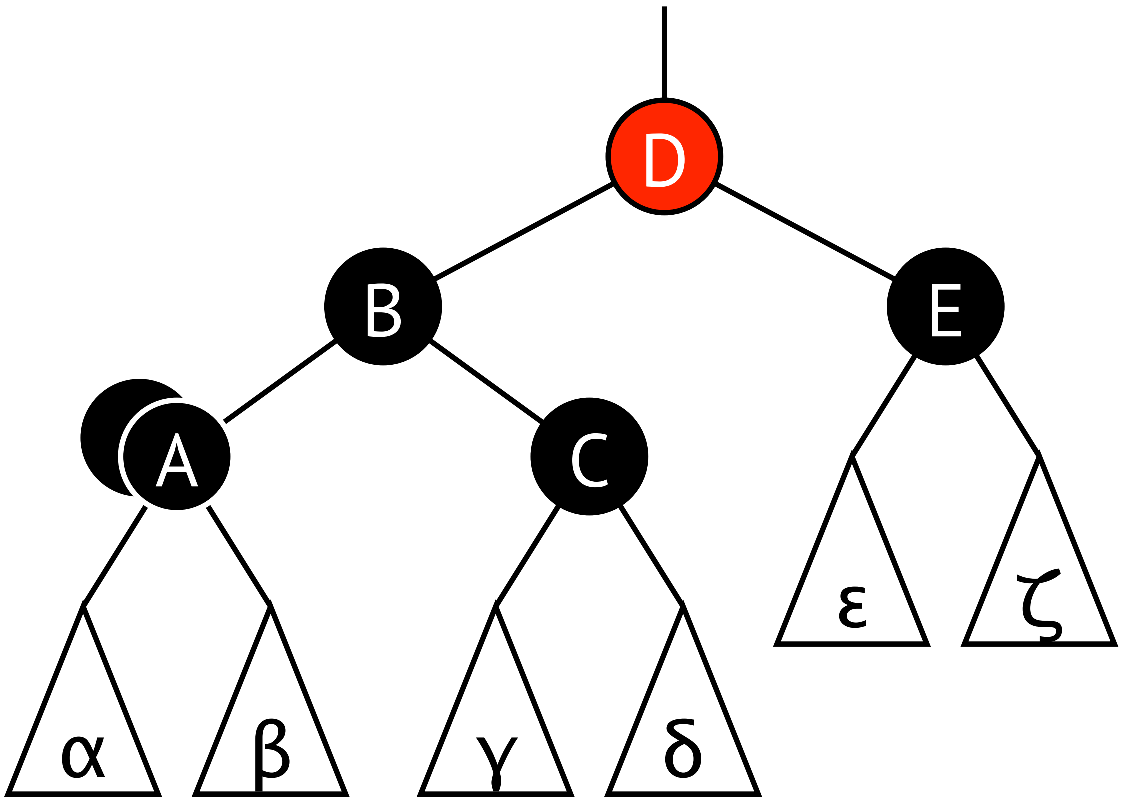
Rot-Schwarz-Bäume: Delete()

Fall 1: Der Bruder ist rot → Rotation nach links



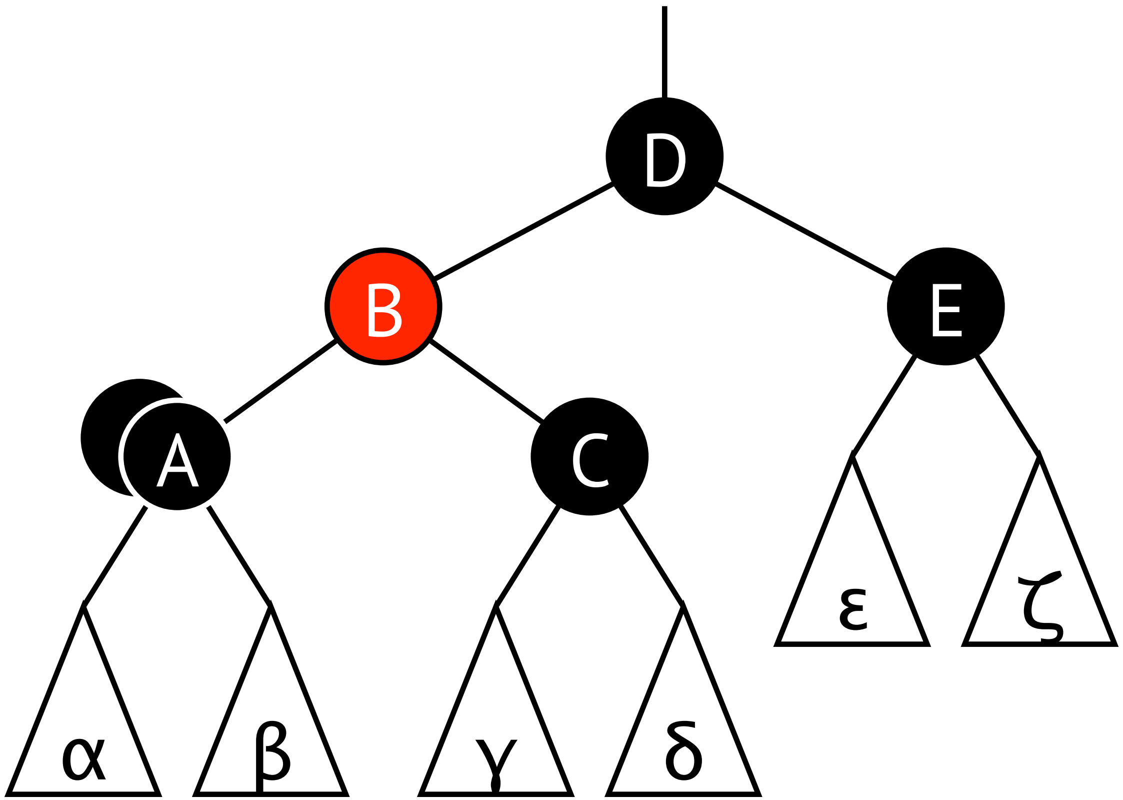
Rot-Schwarz-Bäume: Delete()

Fall 1: Der Bruder ist rot → Rotation nach links

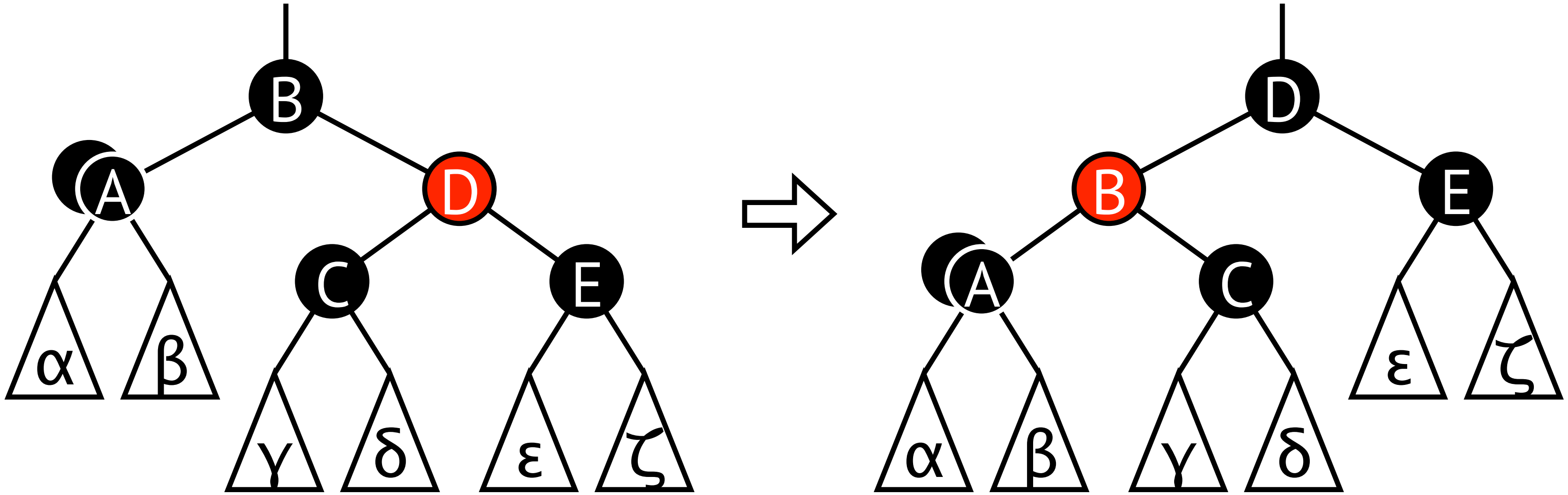


Rot-Schwarz-Bäume: Delete()

Fall 1: Der Bruder ist rot → Rotation nach links und umfärben



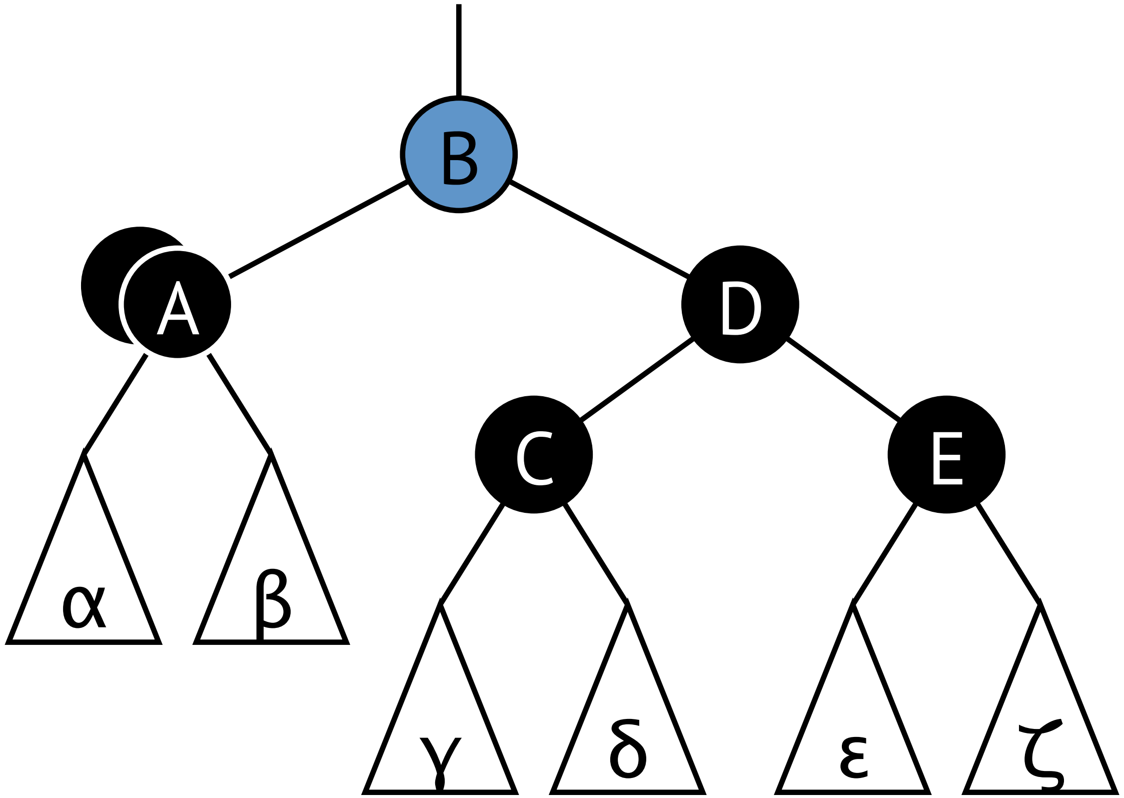
Fall 1: Der Bruder ist rot → Rotation nach links
und umfärben



weiter mit Fall 2,3,4: der Bruder ist **nicht** rot

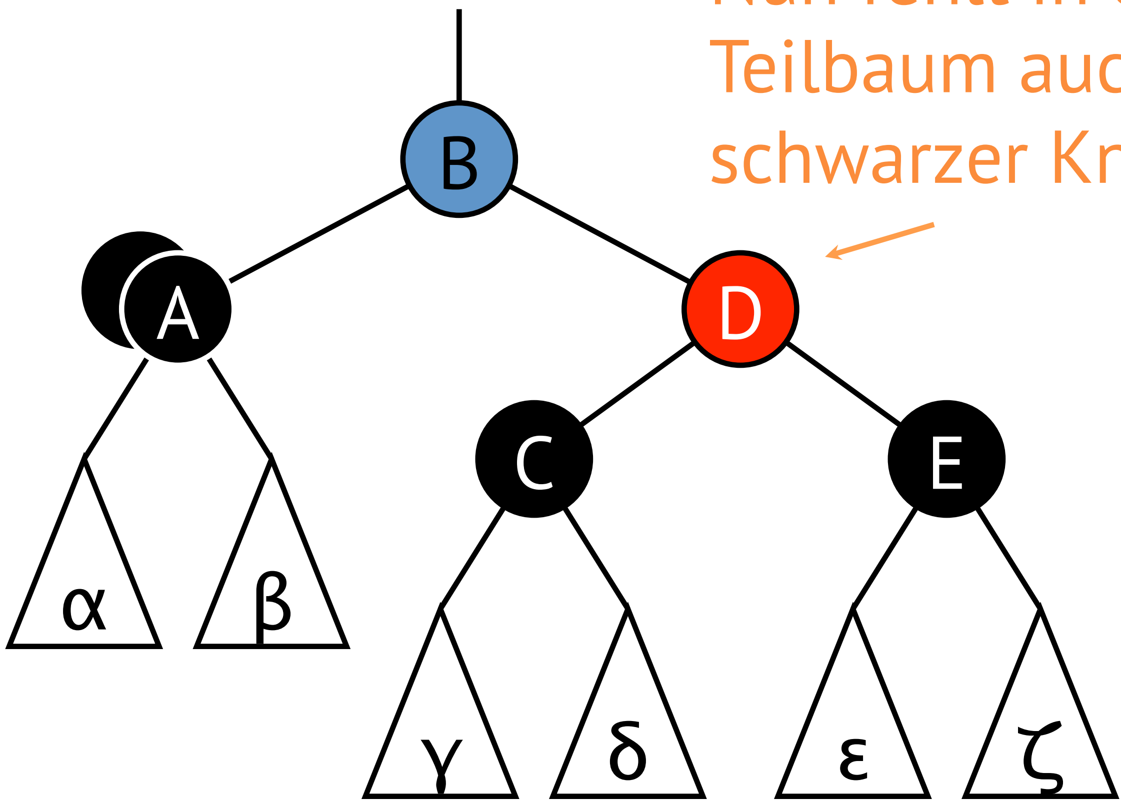
Rot-Schwarz-Bäume: Delete()

Fall 2: Der Bruder ist schwarz und dessen Söhne sind beide schwarz



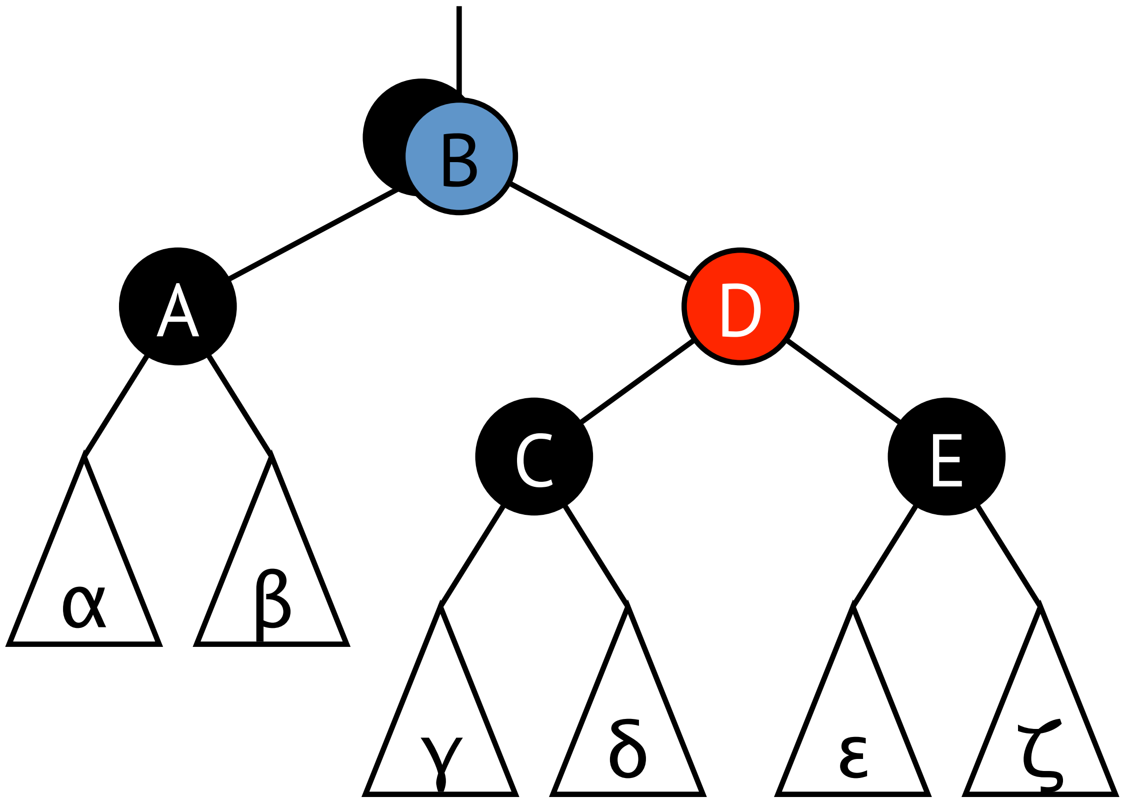
Fall 2: Der Bruder ist schwarz und dessen Söhne sind beide schwarz
→ umfärben

Nun fehlt in diesem Teilbaum auch ein schwarzer Knoten!



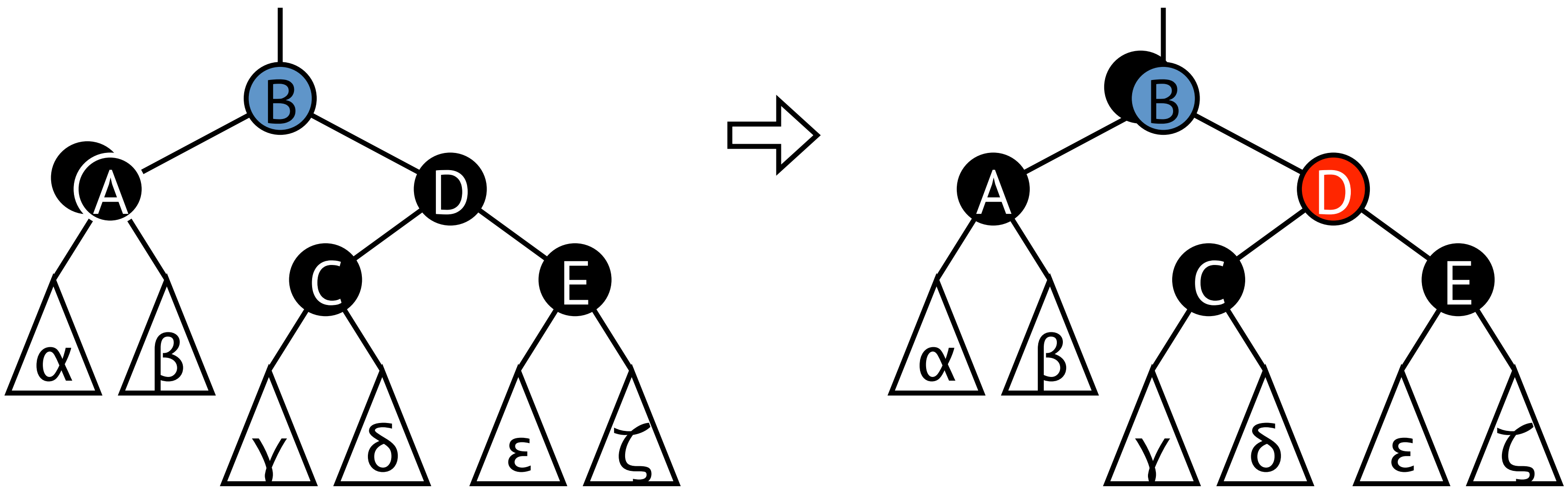
Rot-Schwarz-Bäume: Delete()

Fall 2: Der Bruder ist schwarz und dessen Söhne sind beide schwarz
→ umfärben und Marke nach oben weitergeben

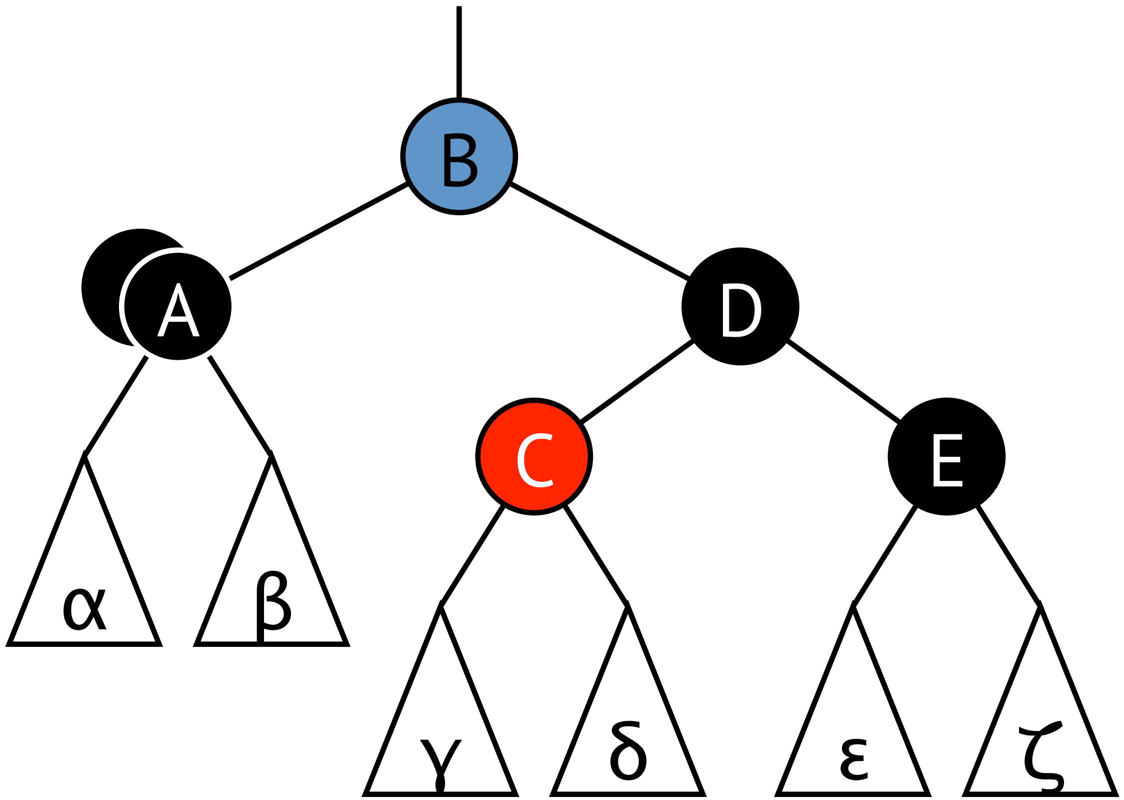


Rot-Schwarz-Bäume: Delete()

Fall 2: Der Bruder ist schwarz und dessen Söhne sind beide schwarz
→ umfärben und Marke nach oben weitergeben
(Ende, falls "B" rot war, sonst weiter mit "B")

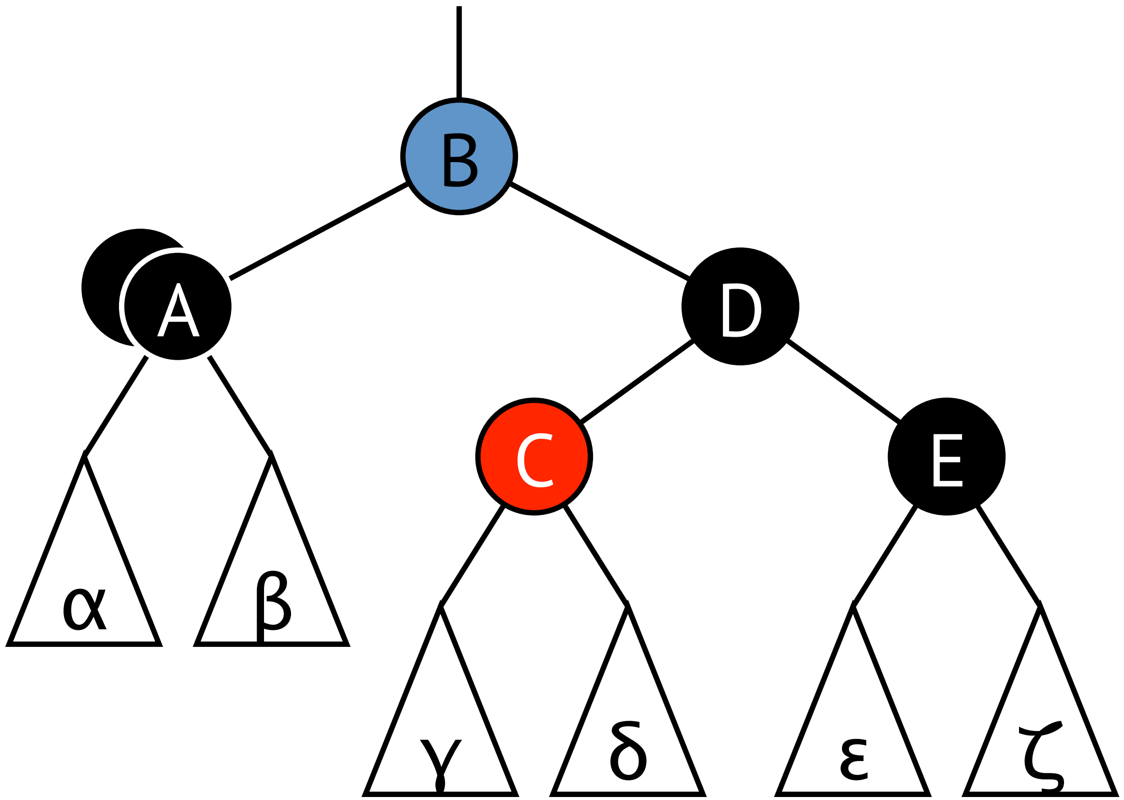


Fall 3: Bruder ist schwarz, dessen innerer Sohn ist rot und dessen äußerer Sohn ist schwarz



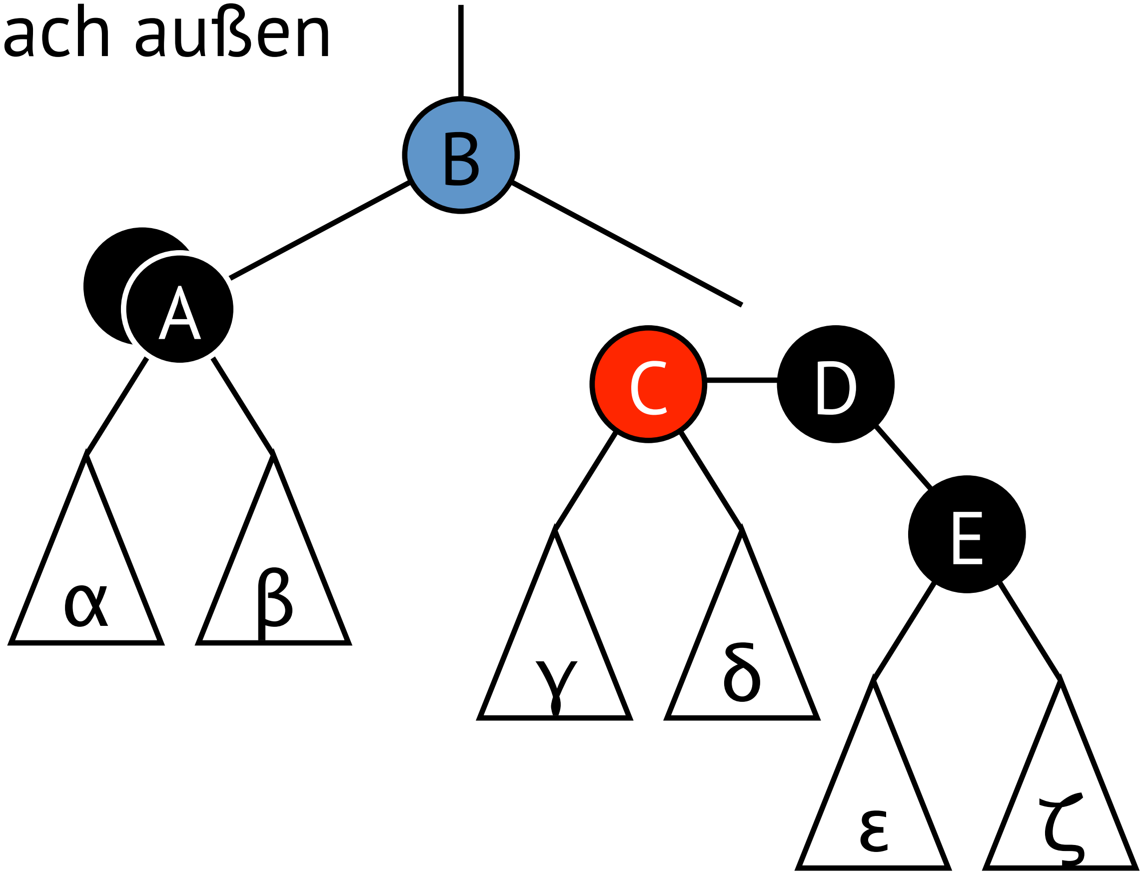
Fall 3: Bruder ist schwarz, dessen innerer Sohn ist rot und dessen äußerer Sohn ist schwarz

→ Rotation nach außen



Fall 3: Bruder ist schwarz, dessen innerer Sohn ist rot und dessen äußerer Sohn ist schwarz

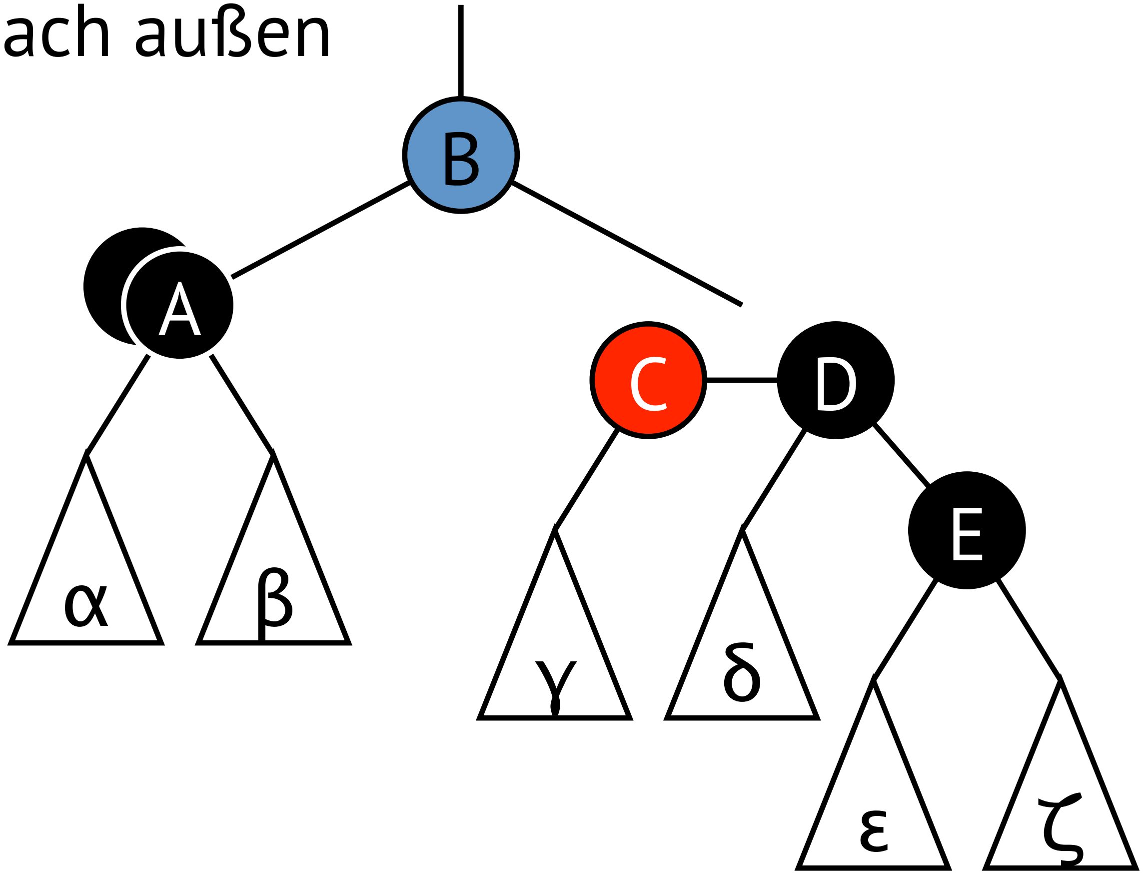
→ Rotation nach außen



Rot-Schwarz-Bäume: Delete()

Fall 3: Bruder ist schwarz, dessen innerer Sohn ist rot
und dessen äußerer Sohn ist schwarz

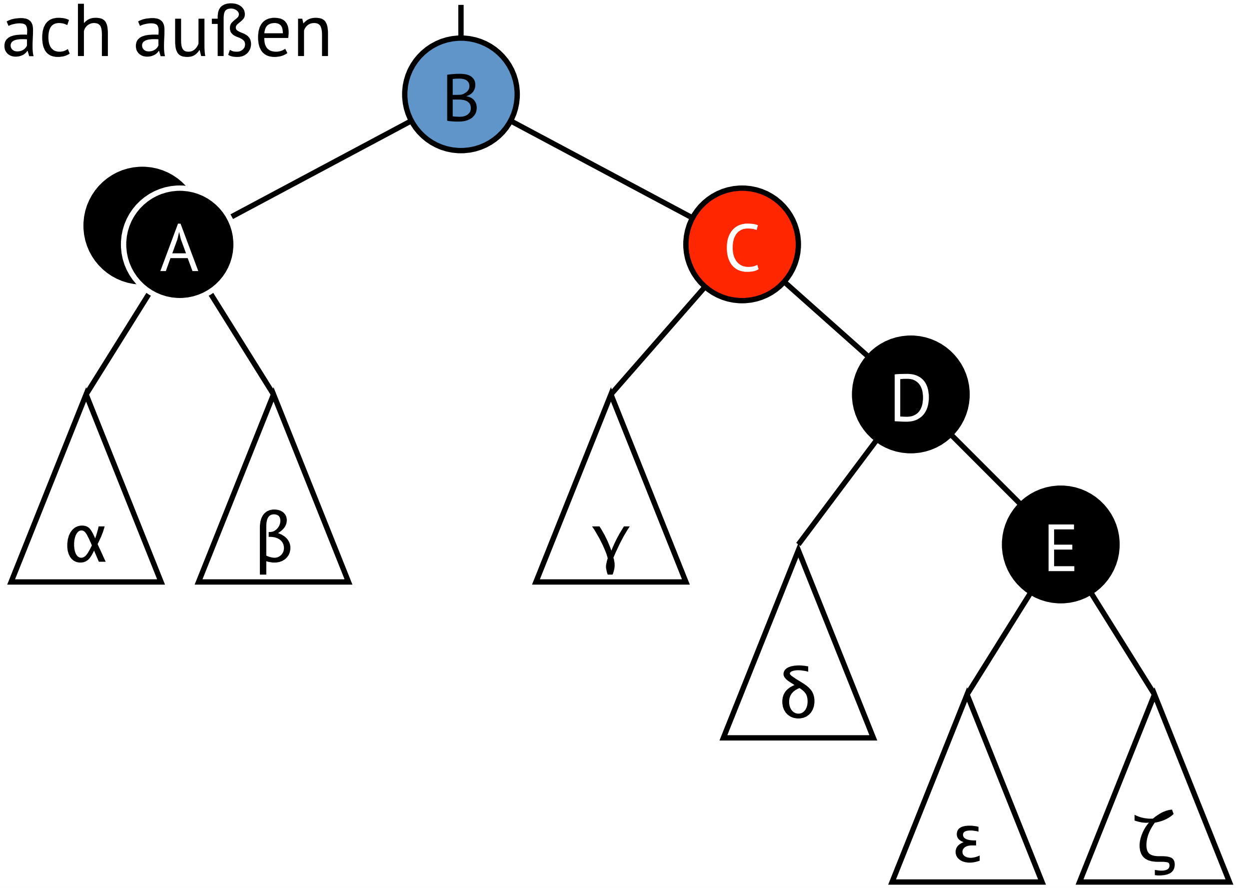
→ Rotation nach außen



Rot-Schwarz-Bäume: Delete()

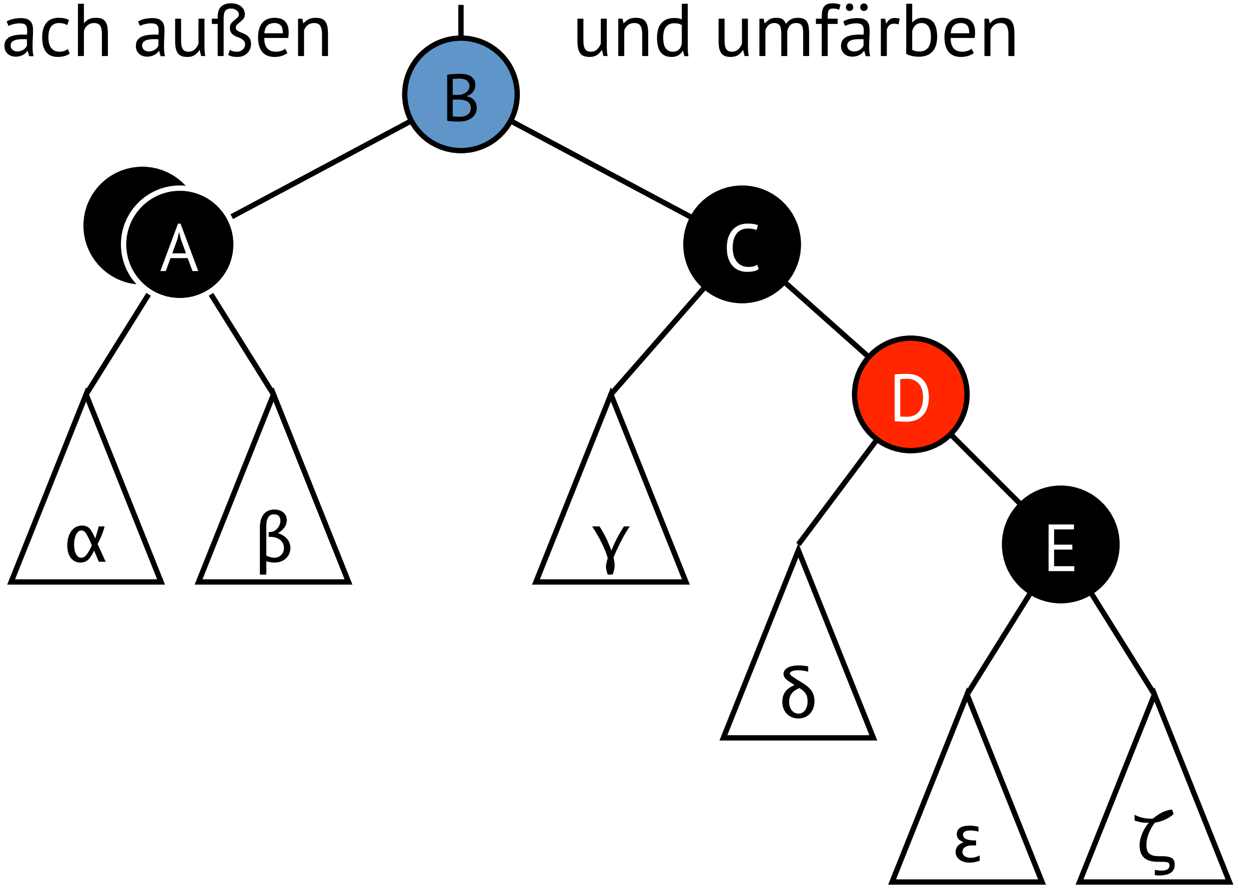
Fall 3: Bruder ist schwarz, dessen innerer Sohn ist rot
und dessen äußerer Sohn ist schwarz

→ Rotation nach außen



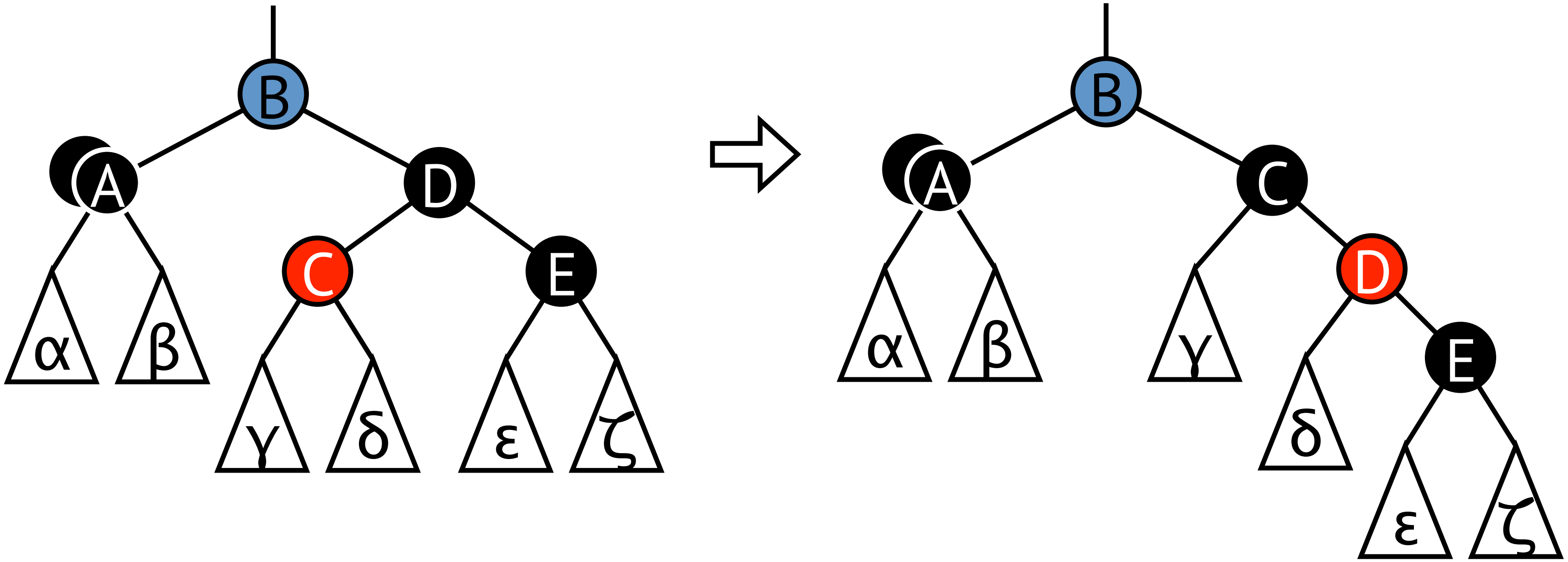
Fall 3: Bruder ist schwarz, dessen innerer Sohn ist rot und dessen äußerer Sohn ist schwarz

→ Rotation nach außen und umfärben



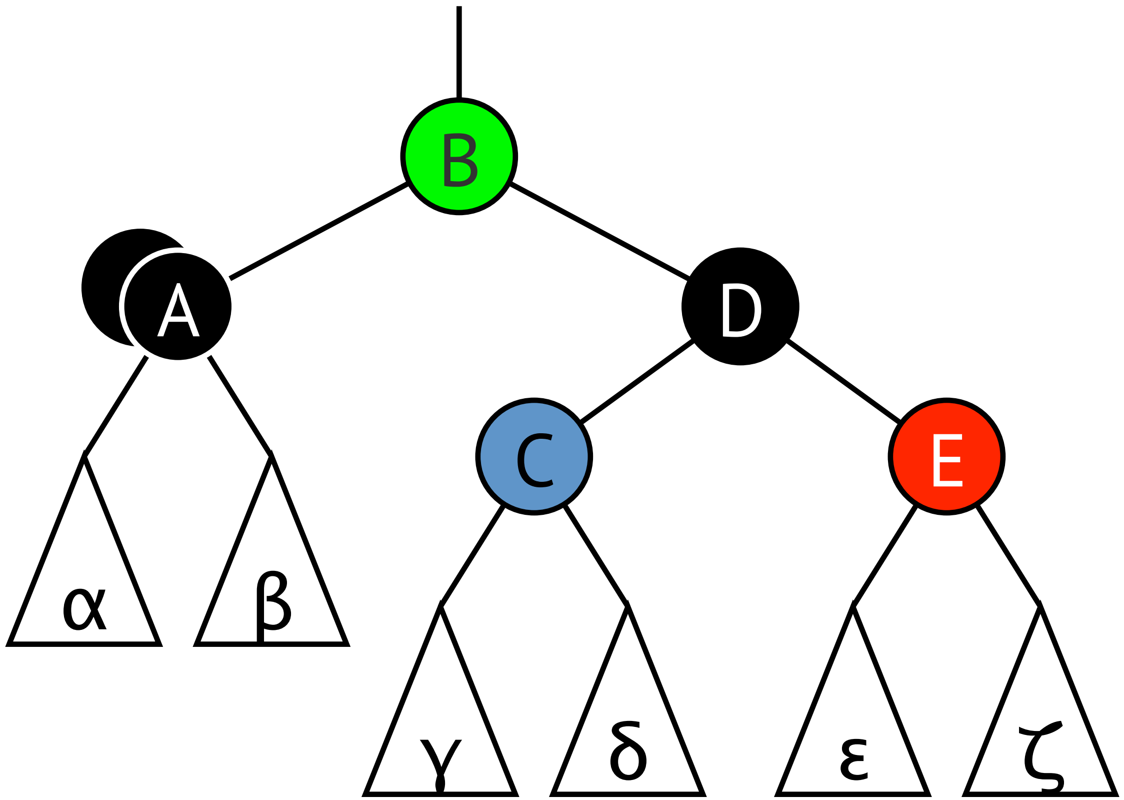
Fall 3: Bruder ist schwarz, dessen innerer Sohn ist rot und dessen äußerer Sohn ist schwarz

→ Rotation nach außen und umfärben

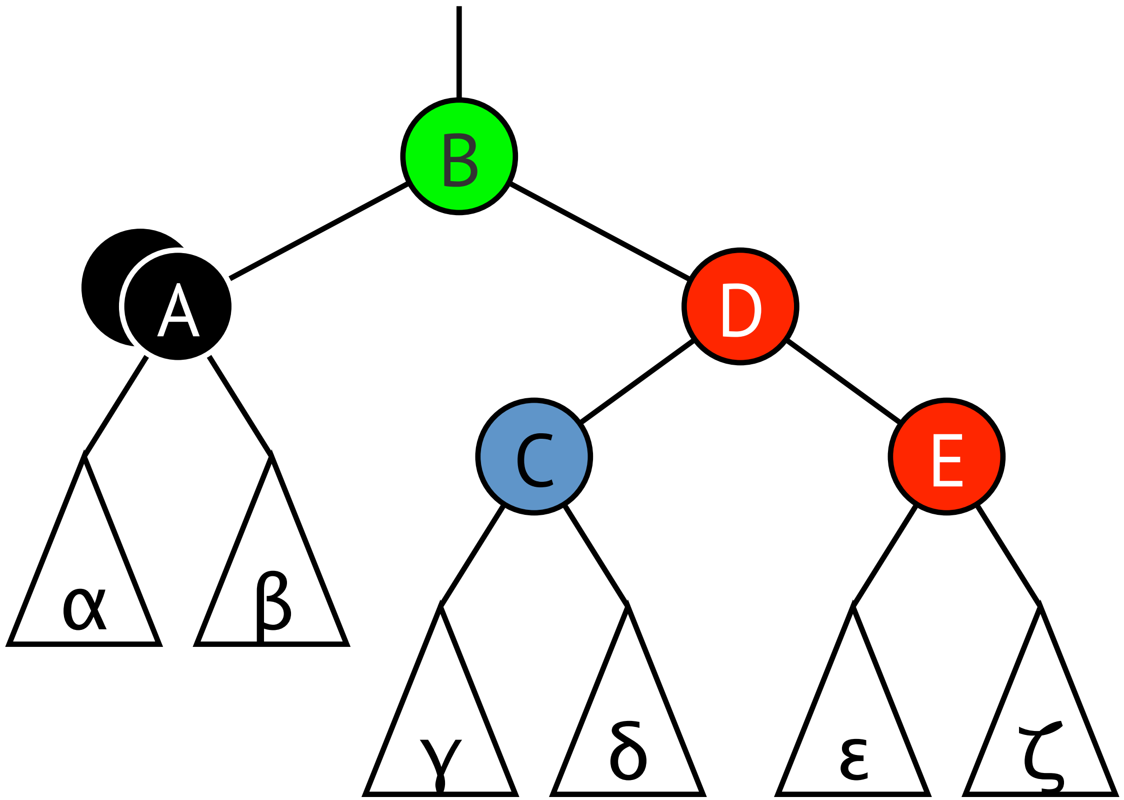


... weiter mit **Fall 4**

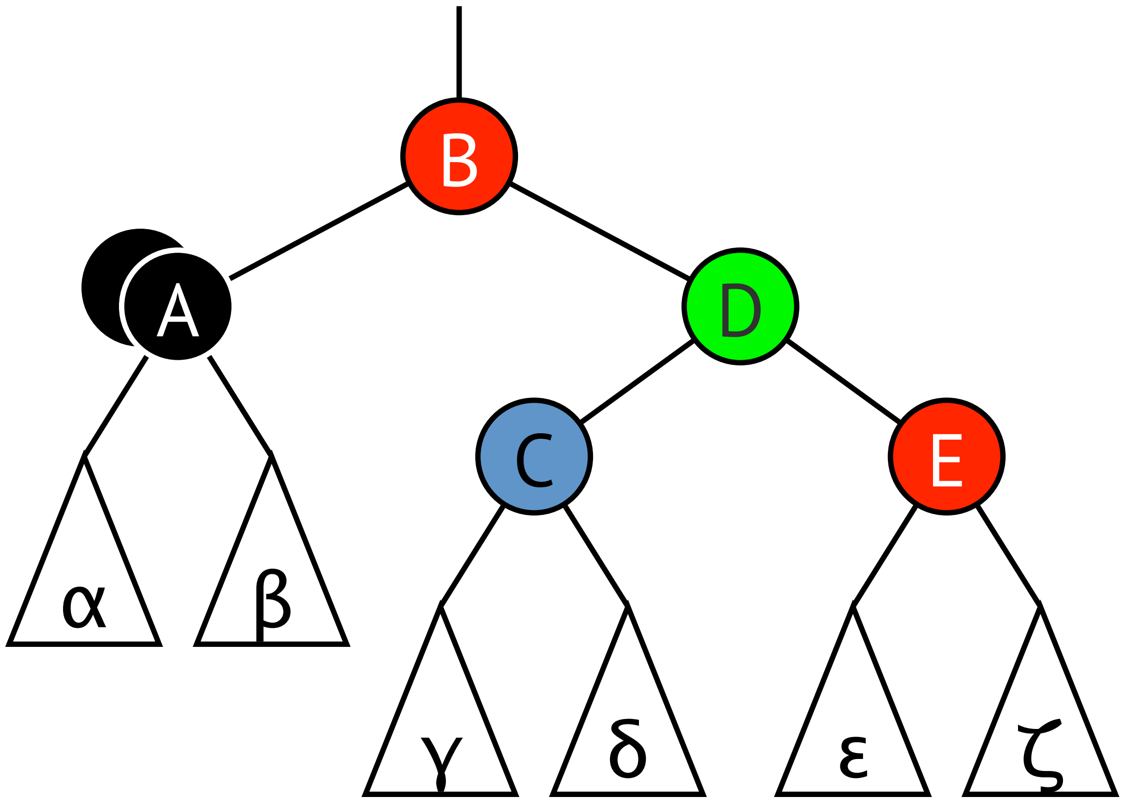
Fall 4: Der Bruder ist schwarz, dessen äußerer Sohn ist rot



Fall 4: Der Bruder ist schwarz, dessen äußerer Sohn ist rot

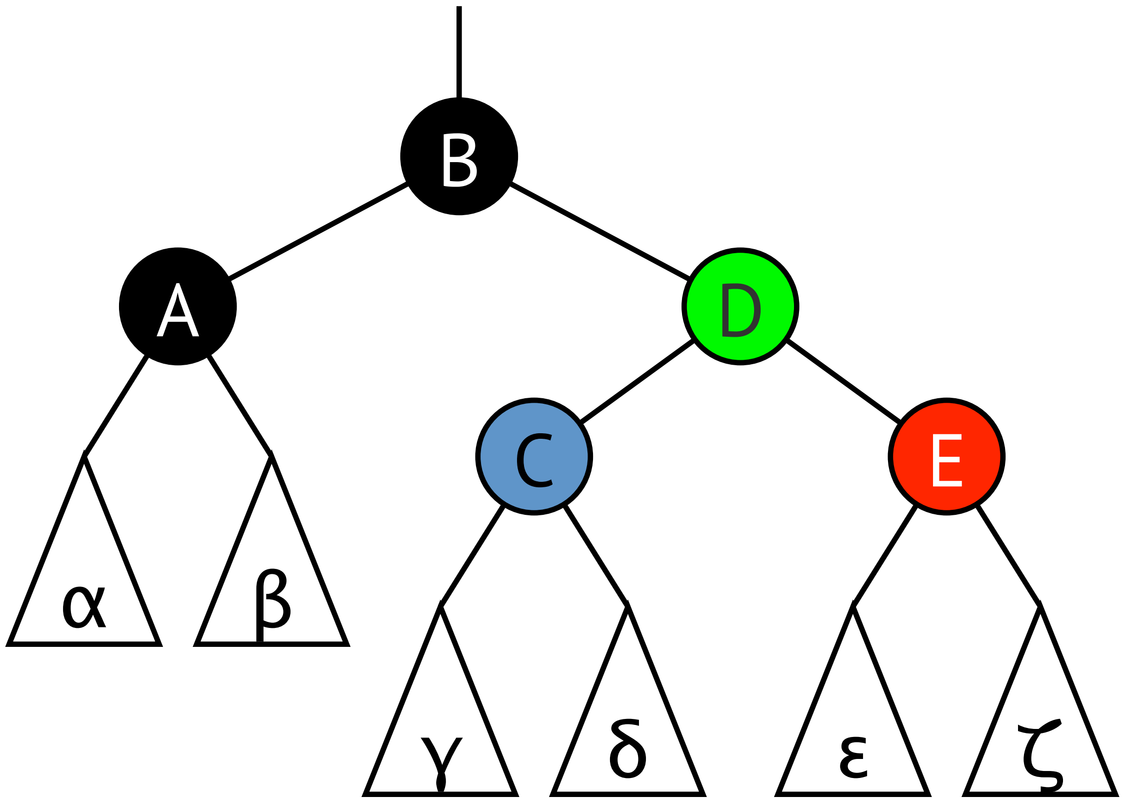


Fall 4: Der Bruder ist schwarz, dessen äußerer Sohn ist rot
→ umfärben



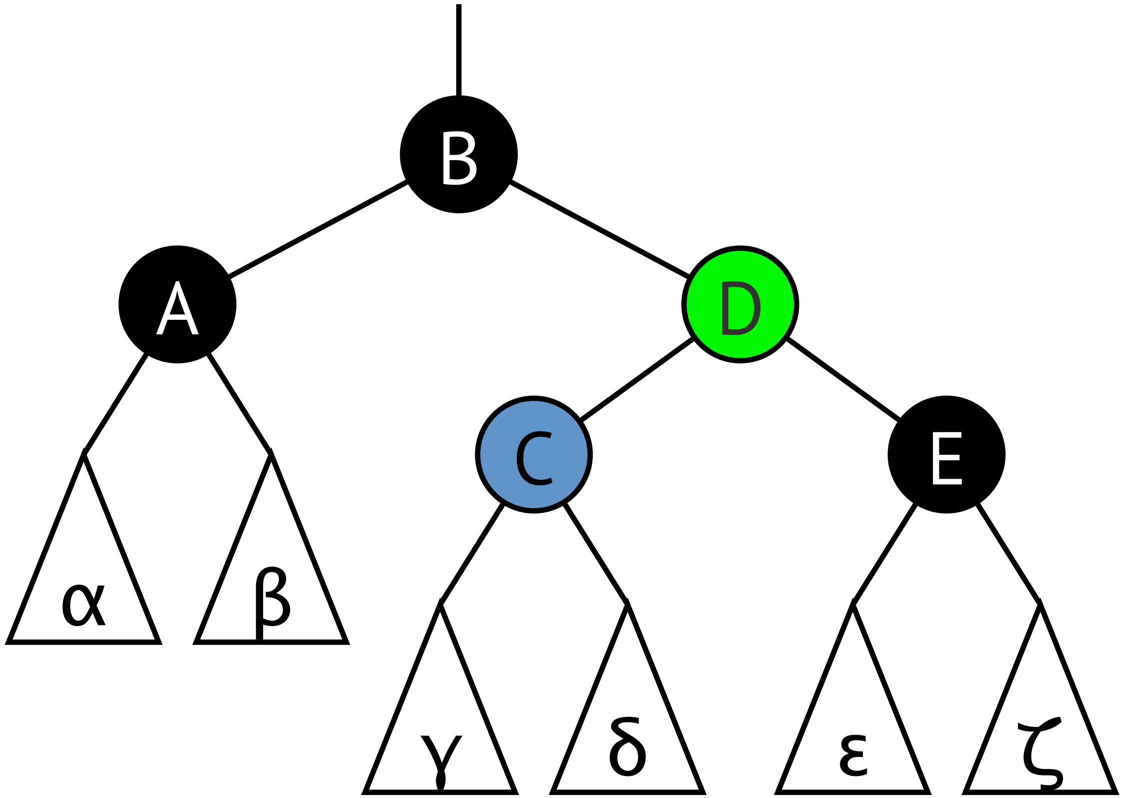
Rot-Schwarz-Bäume: Delete()

Fall 4: Der Bruder ist schwarz, dessen äußerer Sohn ist rot
→ umfärben



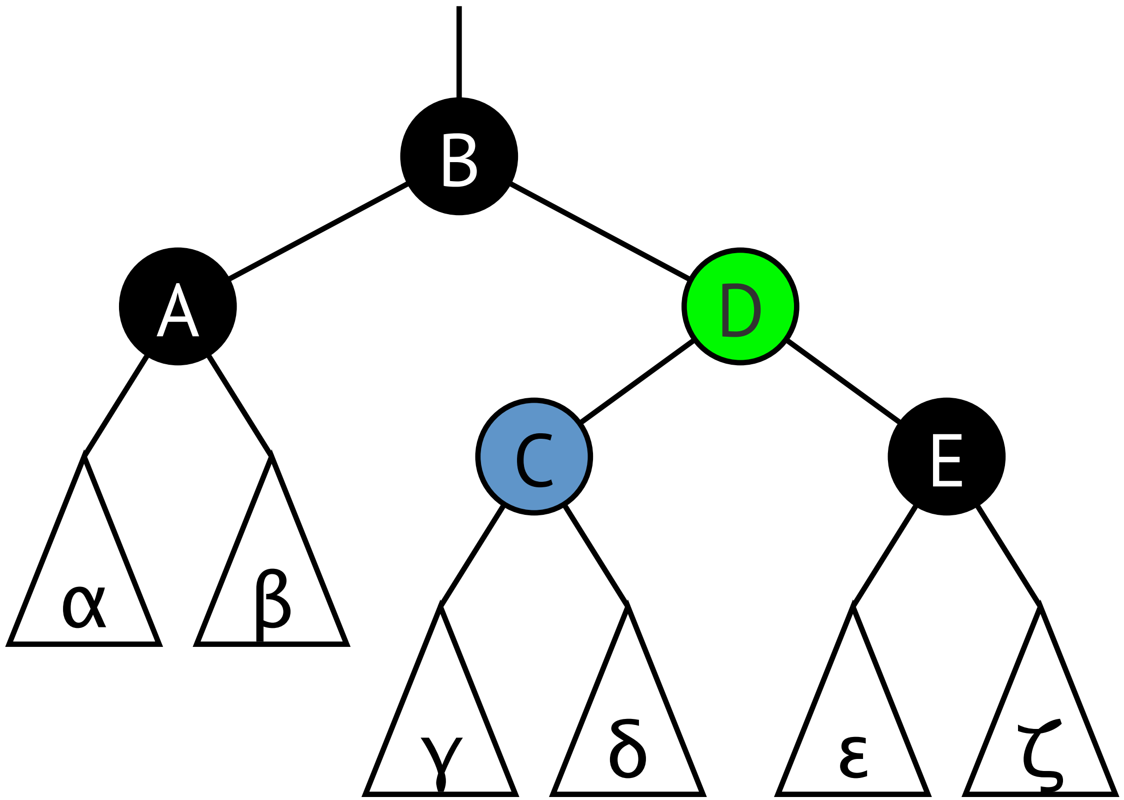
Rot-Schwarz-Bäume: Delete()

Fall 4: Der Bruder ist schwarz, dessen äußerer Sohn ist rot
→ umfärben

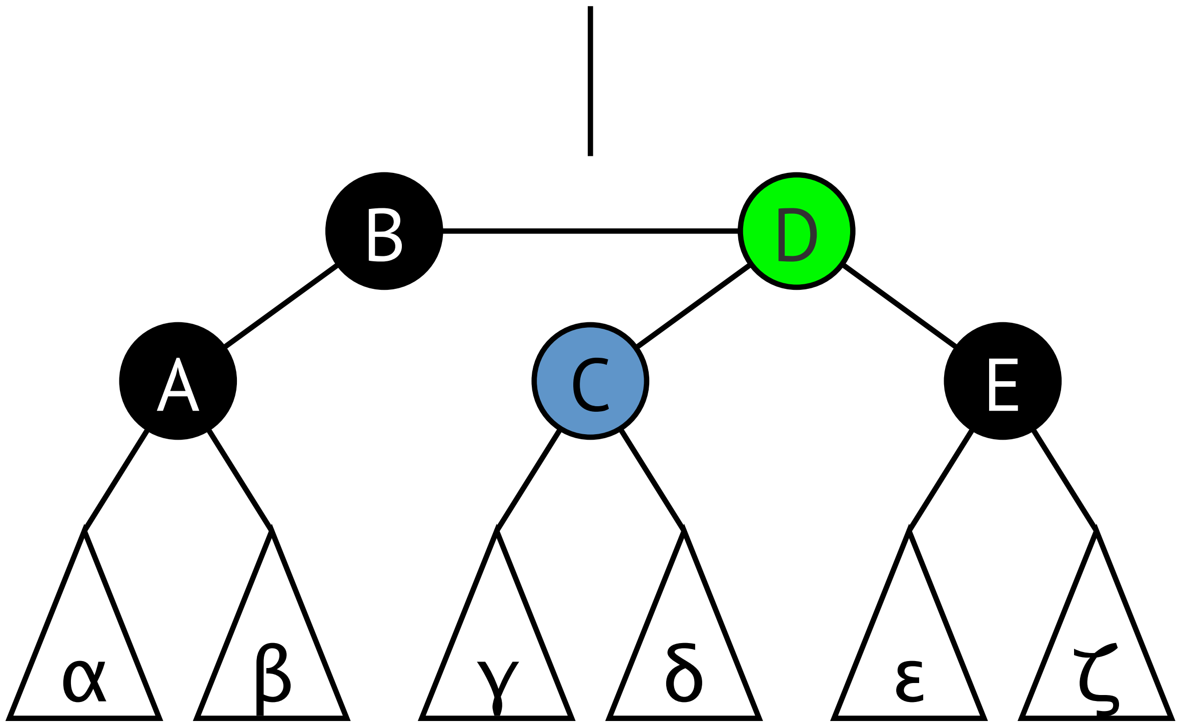


Rot-Schwarz-Bäume: Delete()

Fall 4: Der Bruder ist schwarz, dessen äußerer Sohn ist rot
→ umfärben und Rotation nach innen

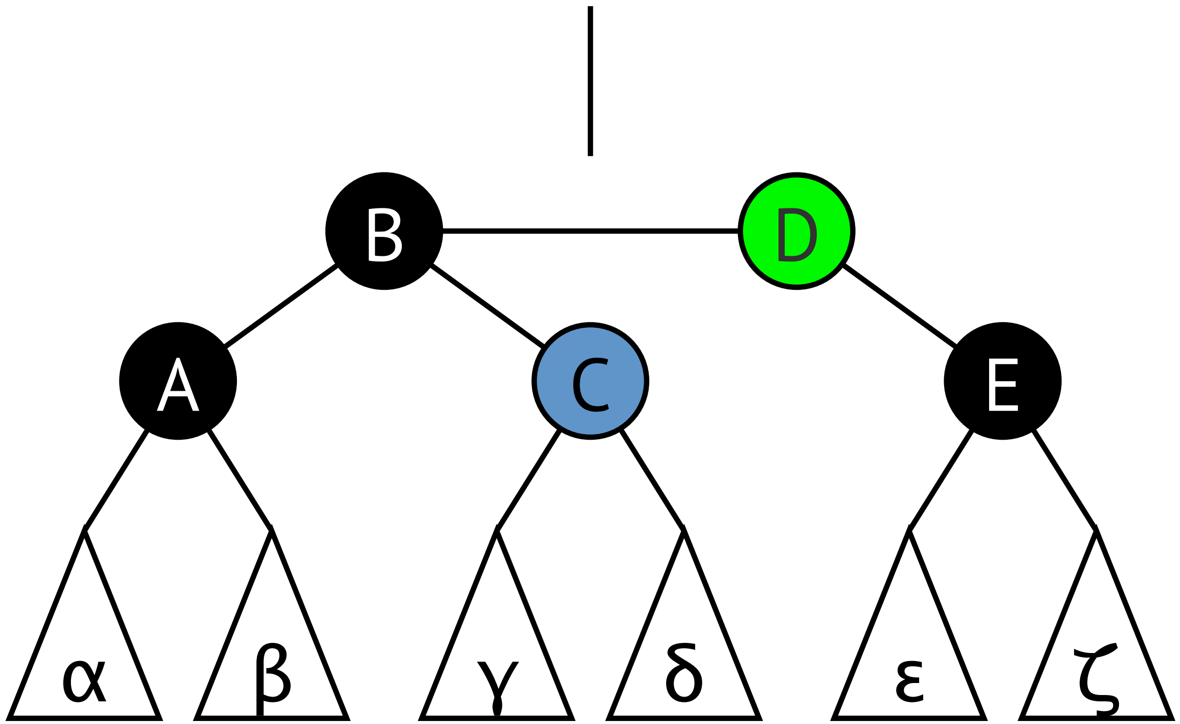


Fall 4: Der Bruder ist schwarz, dessen äußerer Sohn ist rot
→ umfärben und Rotation nach innen

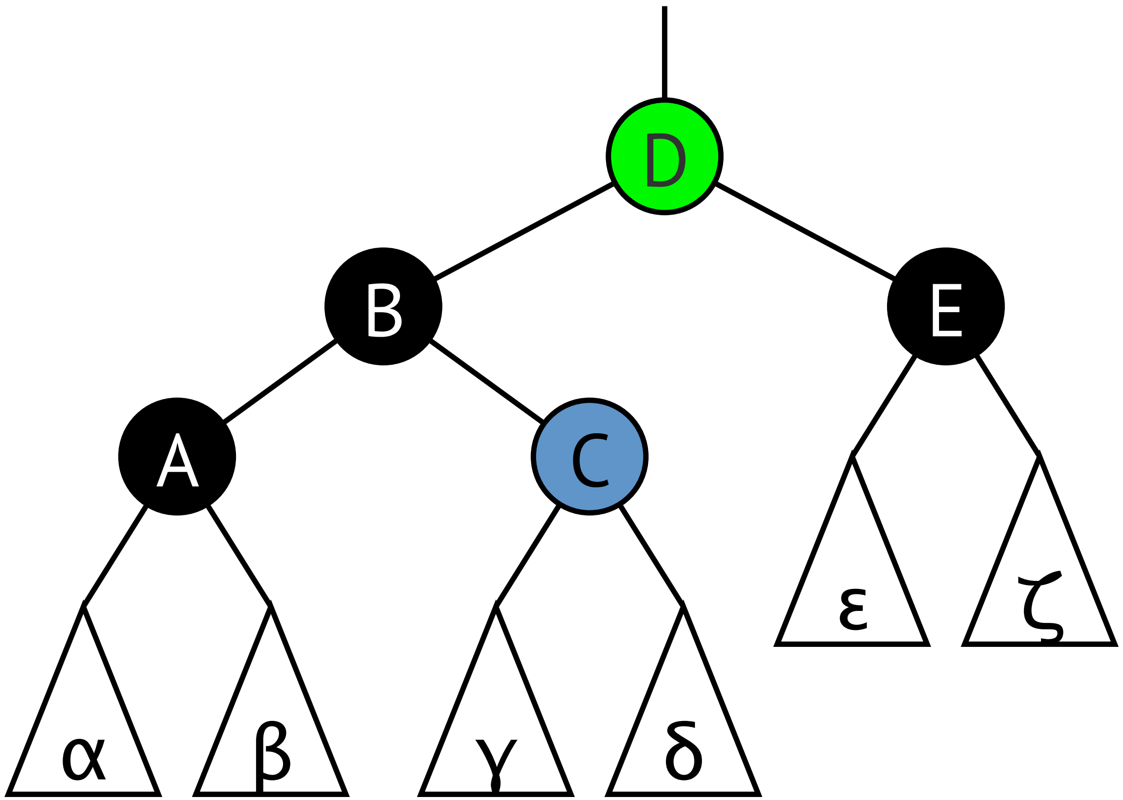


Rot-Schwarz-Bäume: Delete()

Fall 4: Der Bruder ist schwarz, dessen äußerer Sohn ist rot
→ umfärben und Rotation nach innen

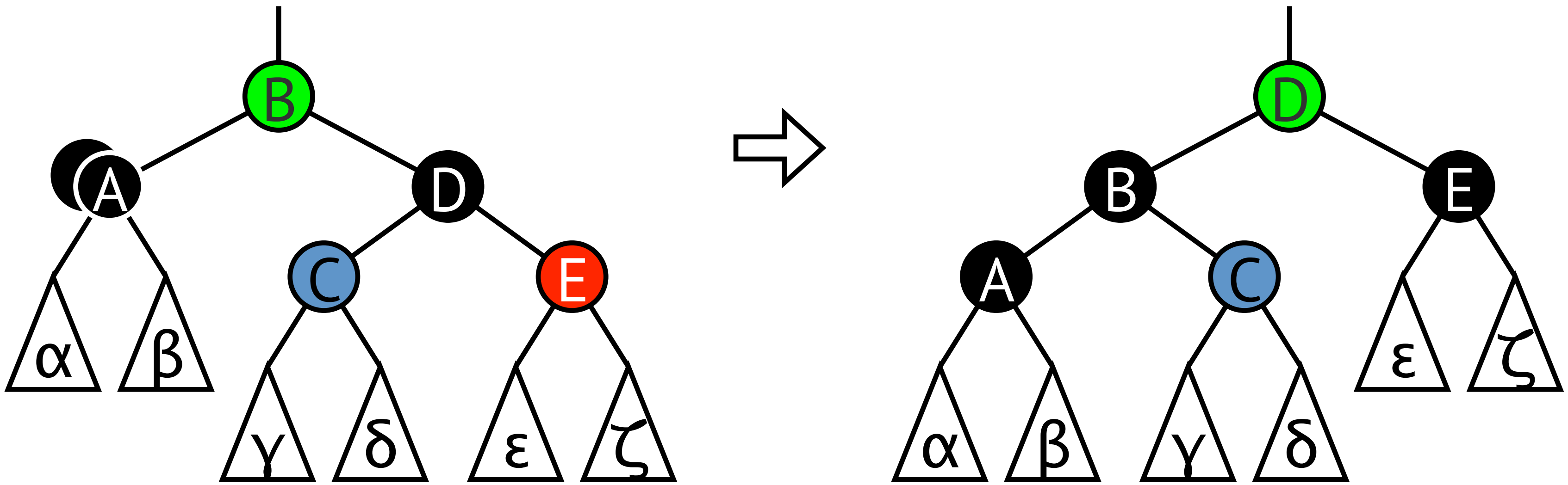


Fall 4: Der Bruder ist schwarz, dessen äußerer Sohn ist rot
→ umfärben und Rotation nach innen



Rot-Schwarz-Bäume: Delete()

Fall 4: Der Bruder ist schwarz, dessen äußerer Sohn ist rot
→ umfärben und Rotation nach innen



Aufwand Delete()

- Suchen des Löschknotens $O(\log n)$
- Löschen $O(1)$
- Konsistenzwiederherstellung
  - Umfärbe-Schritte $O(\log n)$
  - Rotationen $O(1)$ (maximal drei)
- Insgesamt: $O(\log n)$

2.5 Bäume

B-Bäume

- Alternative Interpretation/Implementierung der Rot-Schwarz-Bäume führt auf 2,4-Bäume
- Diese sind ein Spezialfall des allgemeineren Konzepts von B-Bäumen
- Vorteil: größere Mengen von Knoten werden zusammengefasst und passen dadurch besser in einen Festplatten-Block
- **Hysterese**: Nicht in jedem Schritt muss rebalanciert werden.

Speicherhierarchie

- Mit zunehmendem Abstand vom Prozessor …
  - … steigt die Speicherkapzität
  - … wächst die zugreifbare Blockgröße
  - … sinkt die Zugriffsgeschwindigkeit

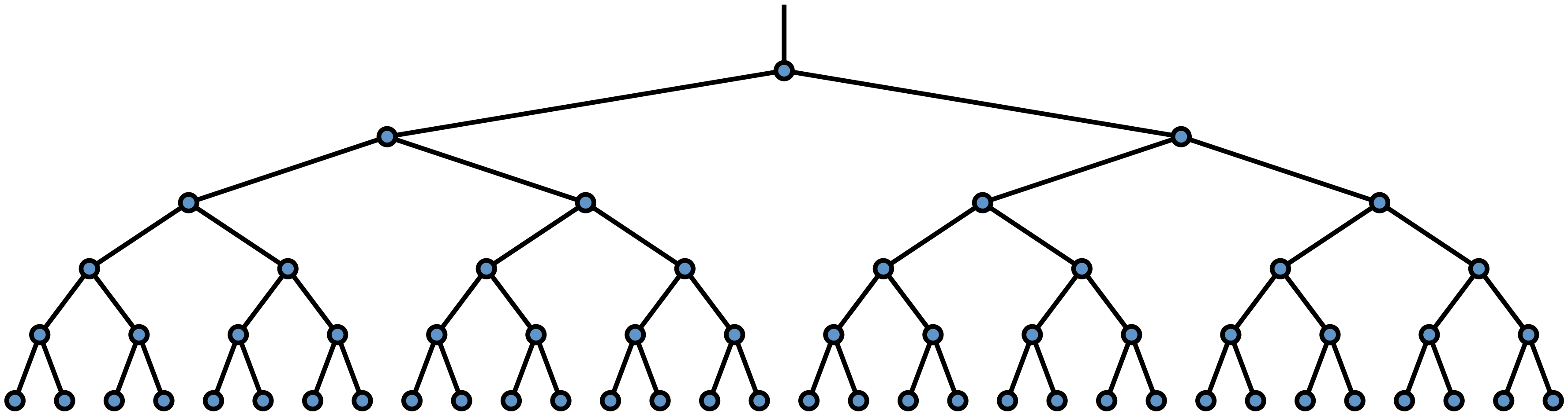
- Effiziente Algorithmen …
  - … führen häufige Berechnungen auf kleinen Datenmengen durch ("innere Schleife")
  - … minimieren die Zugriffe auf externe Speicherebenen
  - … passen die Größe der Datenstrukturen an die jeweiligen Blockgrößen an

Speicherhierarchie

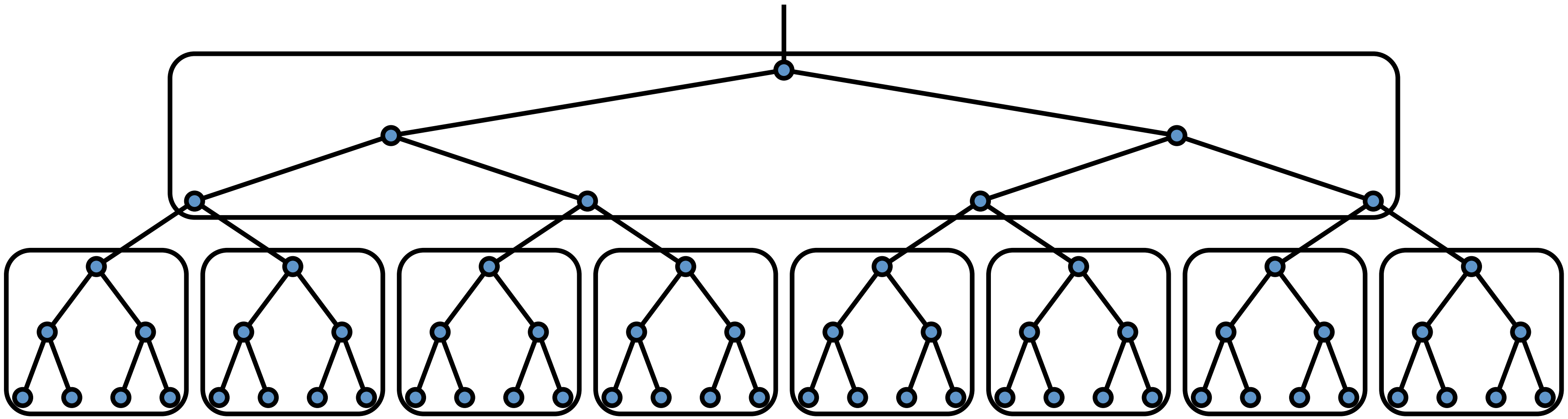
- Bei der Suche in **externen** Binärbaum-strukturen hängt die Zugriffszeit im Wesentlichen von der Verteilung der Knoten auf Festplatten-Sektoren ab.

- Fasse Teilbäume in Sektoren zusammen

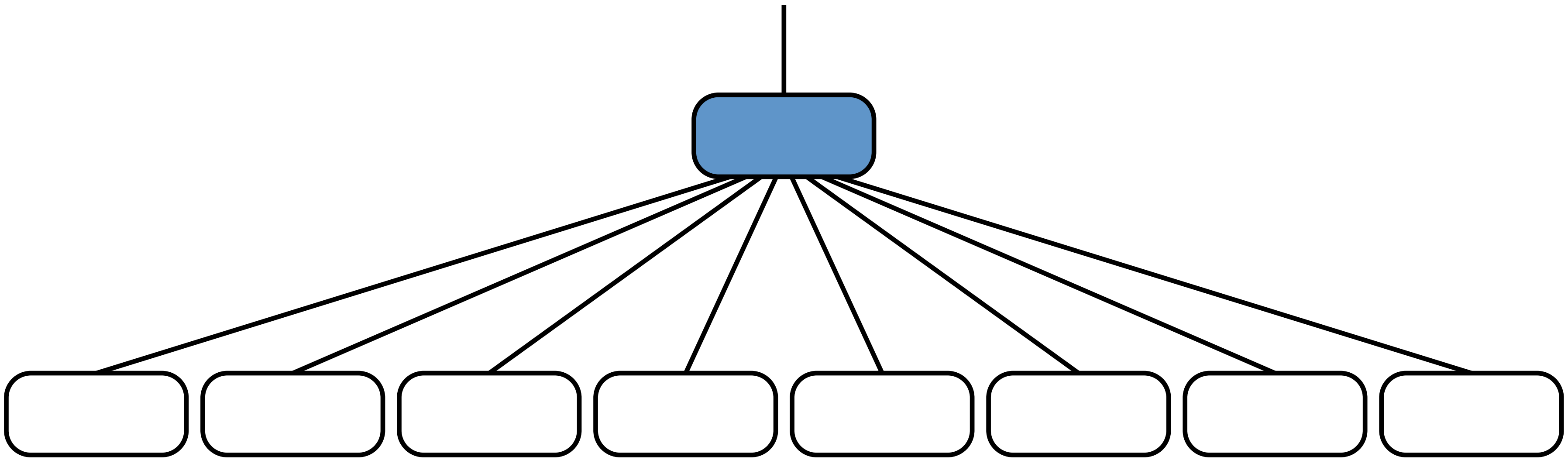
Speicherhierarchie



Speicherhierarchie



Speicherhierarchie



Definition : B-Bäume

- Jeder Knoten x hat die folgenden Felder
  - Die Zahl $n[x]$ der aktuell im Knoten gespeicherten Schlüssel ("Füllgrad")
  - $n[x]$ Schlüssel
    $key_1[x] \leqslant \ldots \leqslant key_{n[x]}[x]$
  - $n[x]+1$ Zeiger auf die Söhne
    $c_0[x],\ c_1[x],\ \ldots,\ c_{n[x]}[x]$
  - Der Bool-Wert $leaf[x]$ zeigt an, ob x ein Blatt ist

Definition : B-Bäume

- Bedingungen
  - Ist k_i ein Schlüssel im Unterbaum mit Wurzel $c_i[x]$ so ist

    $$k_0 \leqslant key_1[x] \leqslant k_1 \leqslant key_2[x] \leqslant \ldots \leqslant key_{n[x]}[x] \leqslant k_{n[x]}$$

  - Alle Blätter haben dieselbe Tiefe h

Definition : B-Bäume

- Bedingungen (Fortsetzung)
  - Es gibt eine feste Zahl $t \geqslant 2$, den minimalen Grad, mit
    - Jeder Knoten *außer der Wurzel* enthält mindestens $t-1$ **Schlüssel**
    - Jeder Knoten enthält höchstens $2 \times t-1$ **Schlüssel**

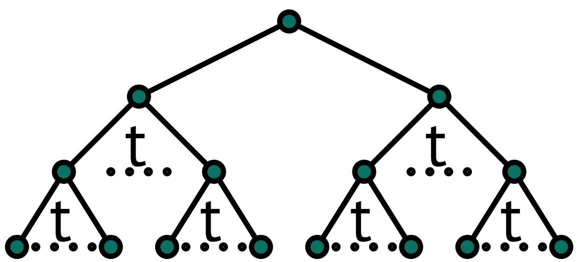
Definition : B-Bäume

- Bedingungen (Fortsetzung, alternativ)
  - Es gibt eine feste Zahl $t \geqslant 2$, den minimalen Grad, mit
    - Jeder Knoten *außer der Wurzel* hat mindestens t **Söhne**
    - Jeder Knoten hat höchstens $2{\times}t$ **Söhne**

B-Bäume

- Für die Höhe h eines B-Baums mit n Schlüsseln gilt

$$h \leq \log_t \frac{n+1}{2}$$



Tiefe	#Knoten
0	1
1	⩾2
2	⩾2t
3	$\geqslant 2t^2$

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=0}^{h-1} 2t^i \\ &= 1 + 2(t-1)\frac{t^h - 1}{t-1} \\ &= 2t^h - 1 \end{aligned}$$

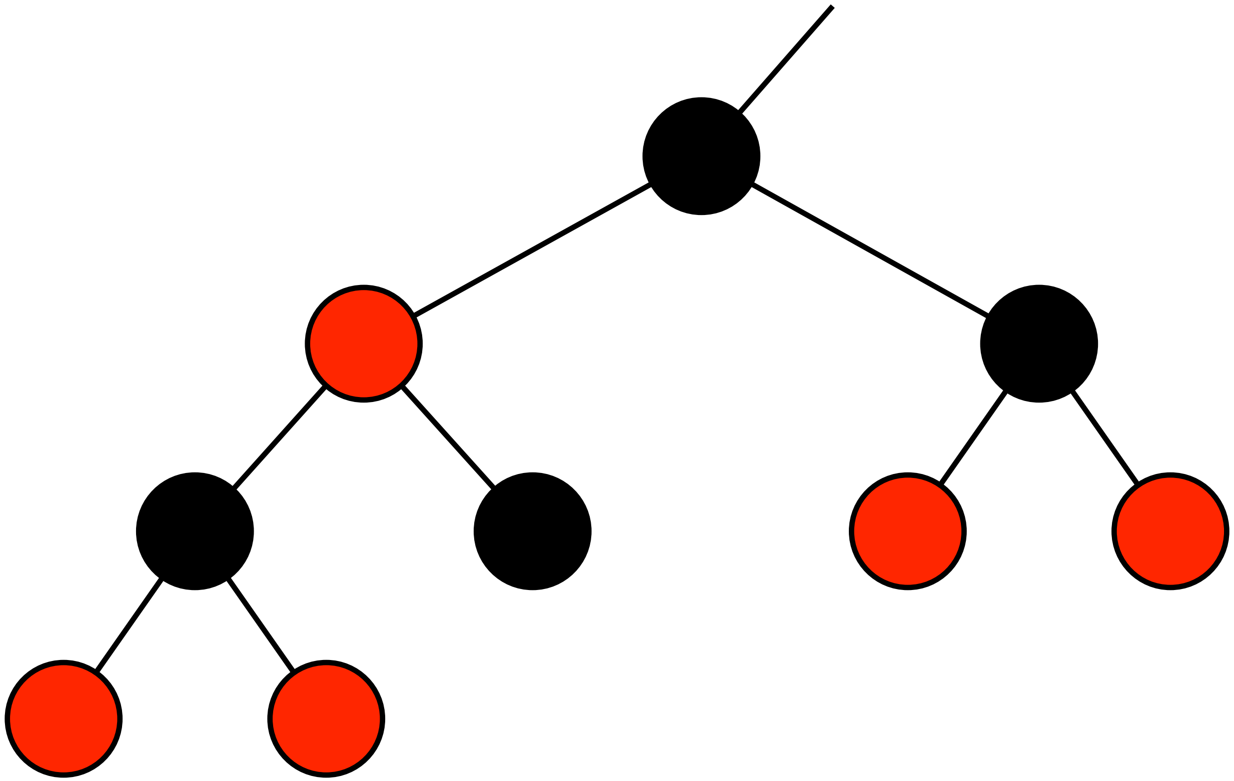
B-Bäume

- Halte den Wurzelknoten immer im Hauptspeicher (da für jeden Zugriff notwendig)
- Erwartete Zugriffe $\approx \log_t n$
- Beispiel
  - 10^8 Telefonnummern in Deutschland
  - Blockgröße 512
  - maximal 2 Festplattenzugriffe

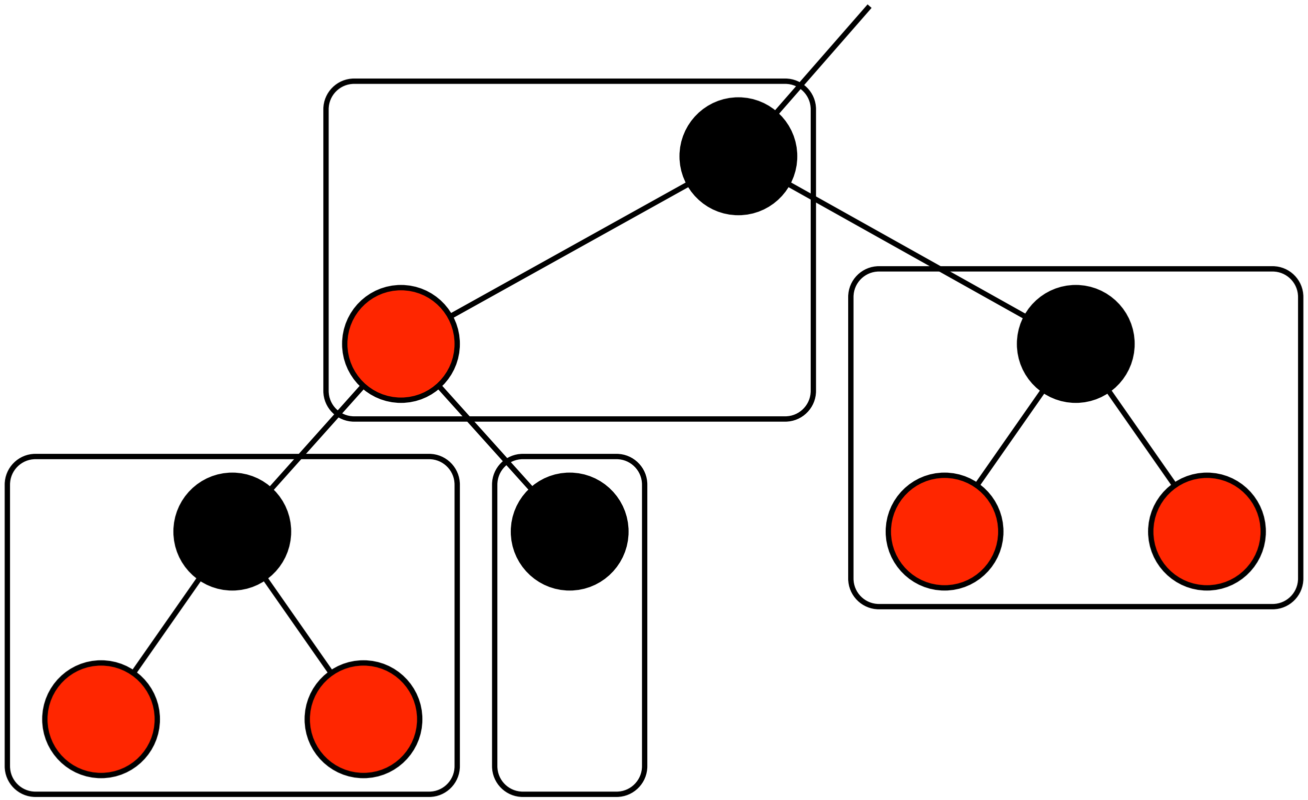
B-Bäume

- Zusammenhang mit Rot-Schwarz-Bäumen
  - Jeder schwarze Knoten absorbiert seine roten Söhne
  - Minimaler Füllgrad: $t = 2$ Söhne
  - Maximaler Füllgrad: $2t = 4$ Söhne
  - (2,4)-Bäume

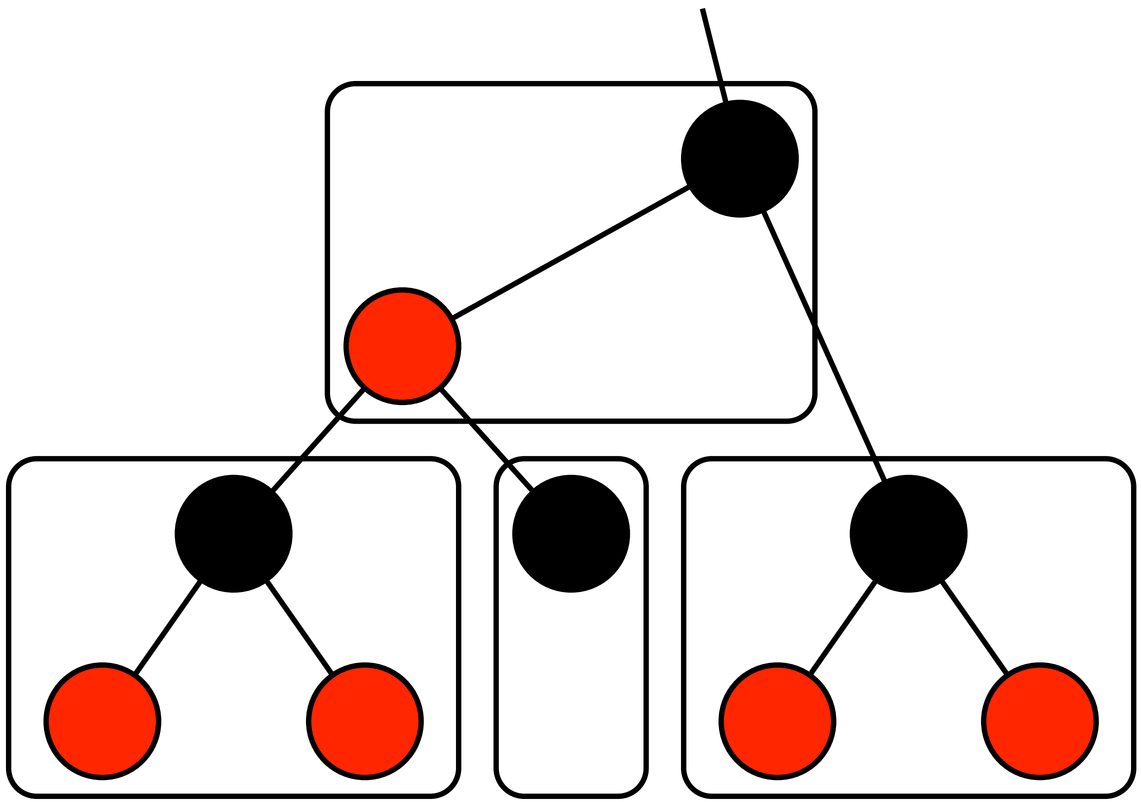
Rot-Schwarz-Bäume



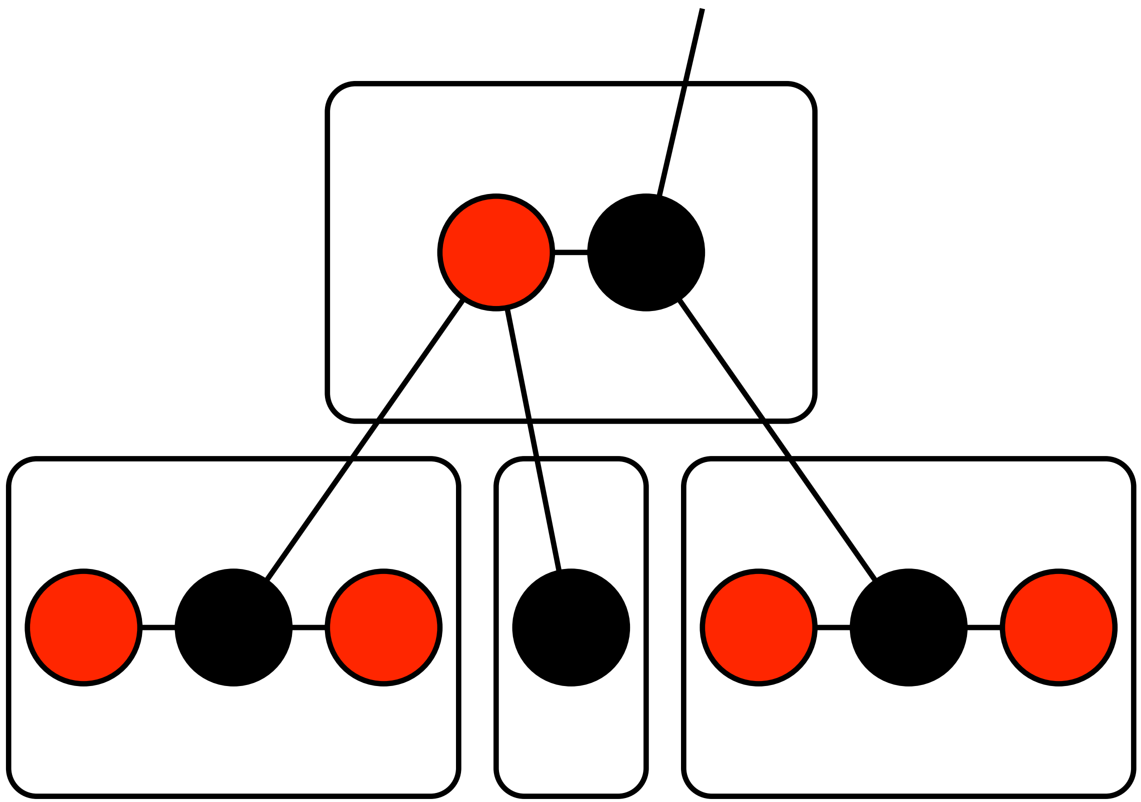
Rot-Schwarz-Bäume



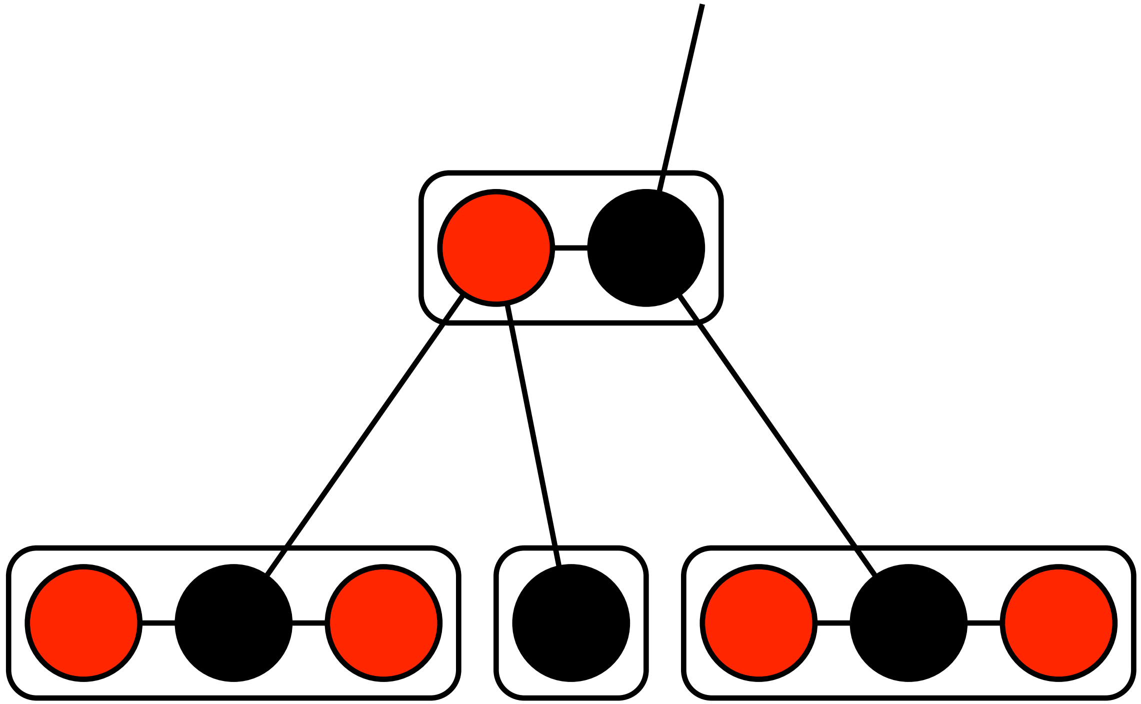
Rot-Schwarz-Bäume



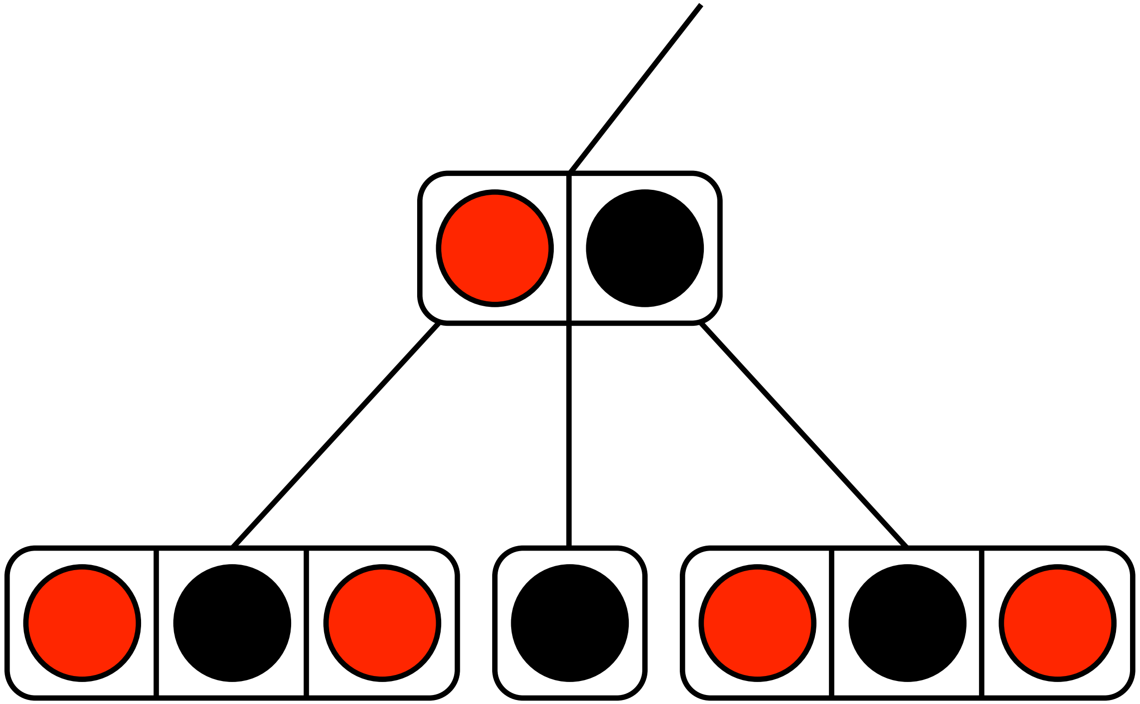
Rot-Schwarz-Bäume



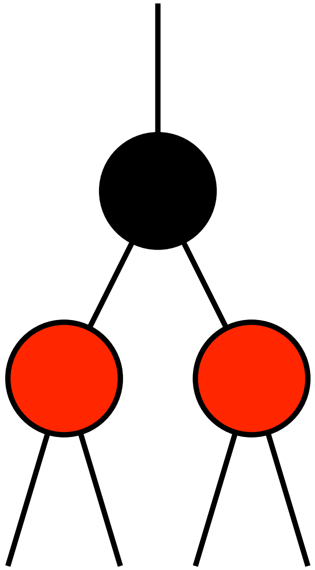
Rot-Schwarz-Bäume

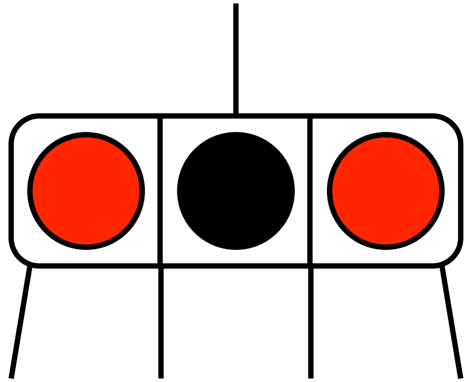


Rot-Schwarz-Bäume

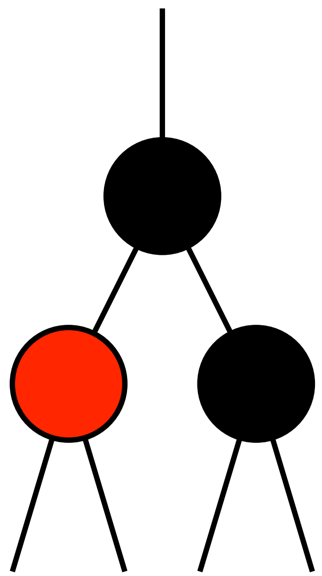


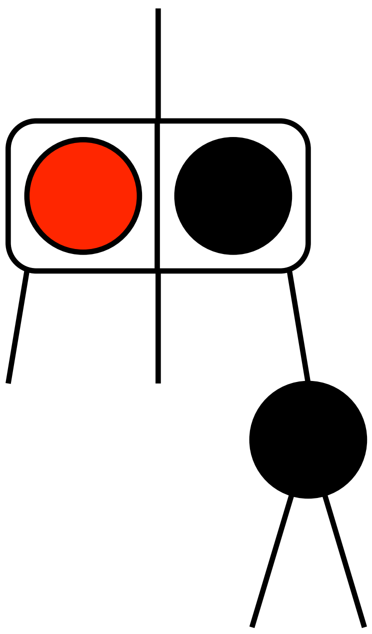
Rot-Schwarz-Bäume



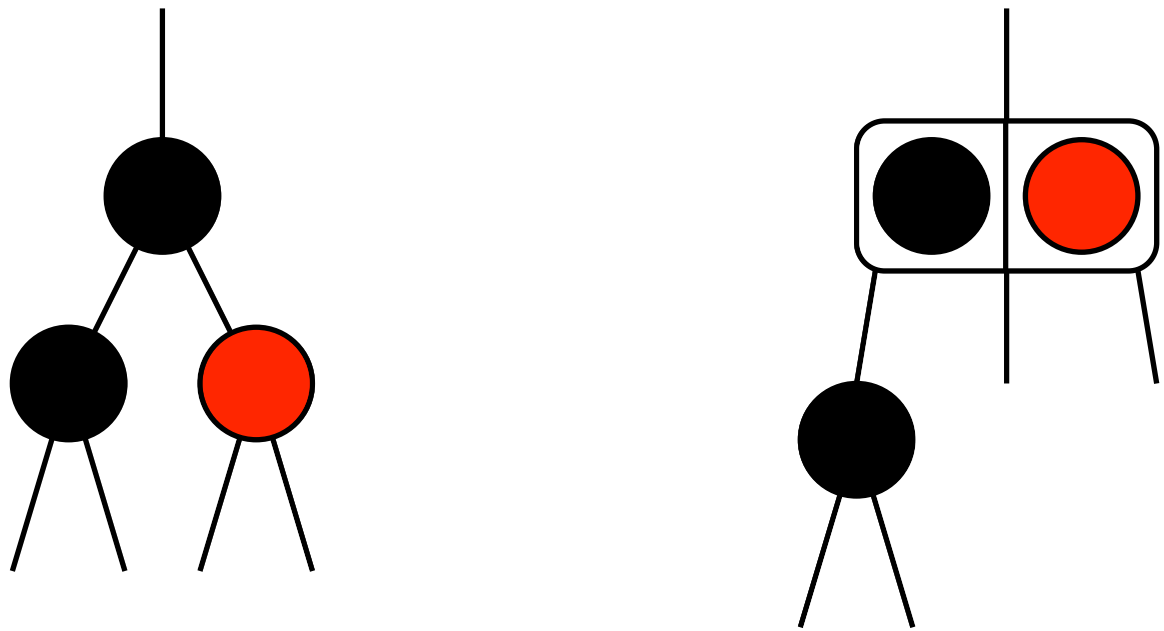


Rot-Schwarz-Bäume

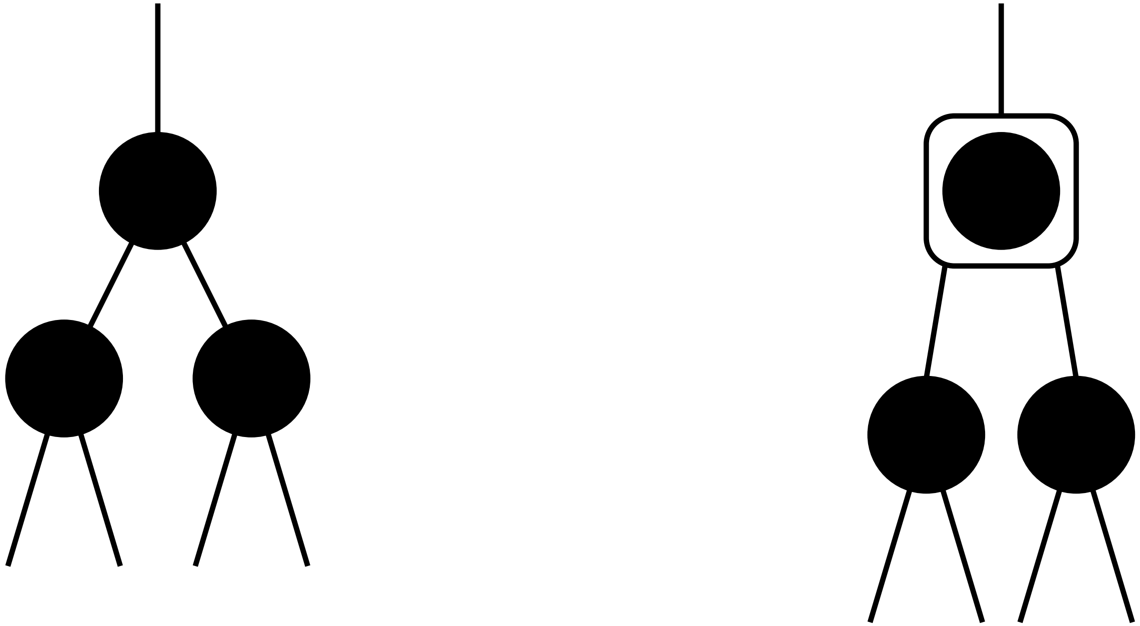




Rot-Schwarz-Bäume



Rot-Schwarz-Bäume



B-Bäume

- Operationen
  - Search()
  - Insert()
  - Delete()
- In den meisten Fällen kommen die Operationen ohne Restrukturierung des Baums aus
- Sonst: $O(1)$ verallgemeinerte Rotationen

Zugriff auf externen Speicher

- DiskRead(x), DiskWrite(x)

- x ← pointer to some object
  DiskRead(x);
  access/modify the contents of x
  DiskWrite(x) // only if modified
  access but don't modify the contents of x

Create()

- Create()

  x ← AllocateNode()

  leaf[x] ← true

  n[x] ← 0

  DiskWrite(x)

Search()

- SEARCH(x, k)

```
i ← 1
while i ⩽ n[x] and k > key_i[x]
    i ← i + 1
if i ⩽ n[x] and k = key_i[x] then
    return (x,i)
if leaf[x] then
    return NIL
else
    DiskRead( c_{i−1}[x] )
    Search( c_{i−1}[x], k )
```

Insert()

- Bei Binärbäumen wird immer ein neuer Blattknoten eingefügt
- Bei B-Bäumen ist kein neuer Knoten nötig, solange der Blattknoten noch nicht voll ist
- Steigt die Zahl der Schlüssel über $2 \times t - 1$, so wird das Blatt geteilt

Insert()

- Naiver Ansatz
  - Suche das entsprechende Blatt
  - Split() falls überfüllt
  - Propagiere nach oben, falls Vaterknoten ebenfalls überläuft
  - Problem: Auf die Knoten entlang des Pfades wird jeweils zweimal zugegriffen

Insert()

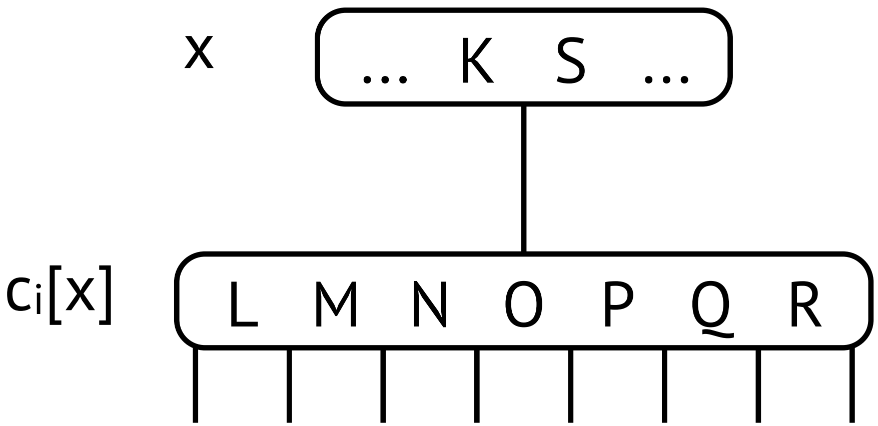
- One-Pass Algorithmus
  - Teile die Knoten entlang des Suchpfades bereits auf dem Hinweg wenn diese schon voll sind
  - Evtl. unnötige Split-Operationen amortisieren sich durch den schnelleren Average-Case

SplitChild()

- SplitChild(x,i)
  // x : Vaterknoten
  // i : Index des Sohns
  // Vorbedingung: $c_i[x]$ ist voll

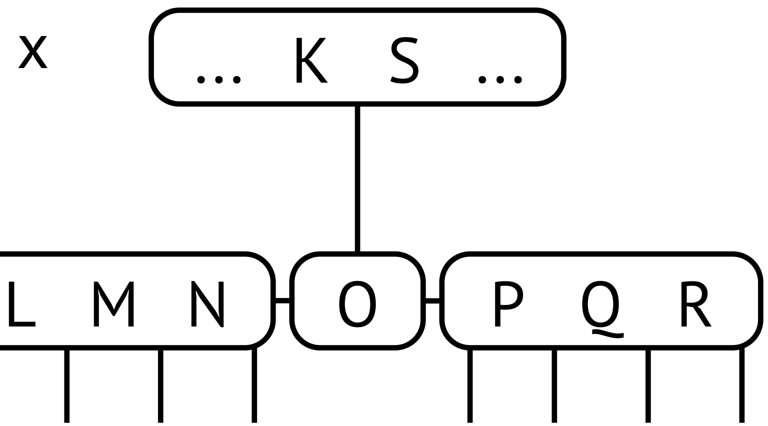
SplitChild()

- Beispiel $t = 4$



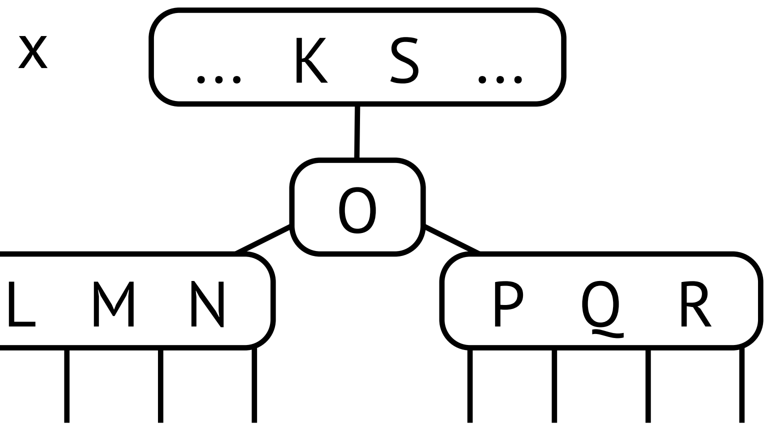
SplitChild()

- Beispiel t = 4



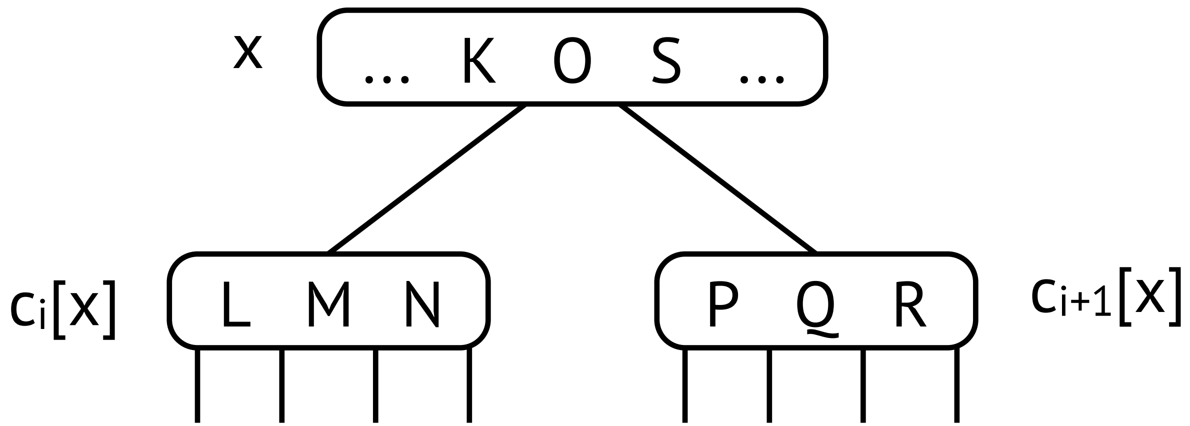
SplitChild()

- Beispiel t = 4



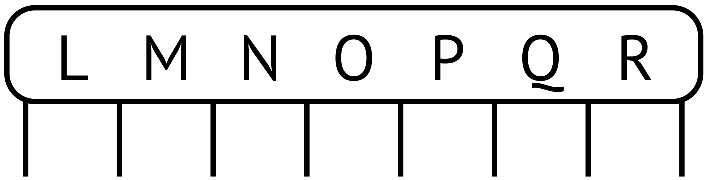
SplitChild()

- Beispiel $t = 4$



SplitChild()

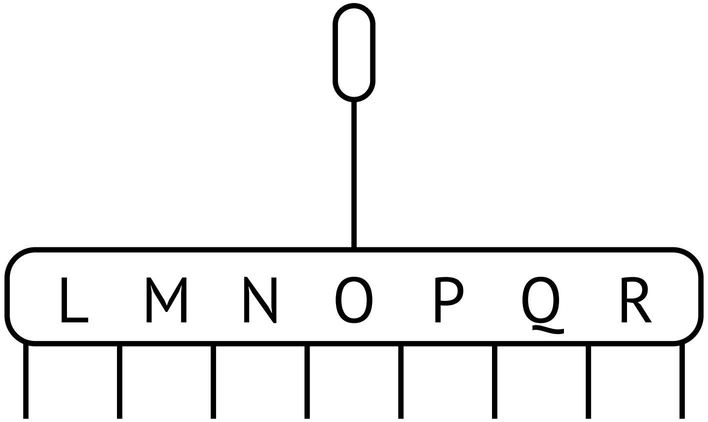
- Spezialfall: Wurzel



SplitChild()

- Spezialfall: Wurzel

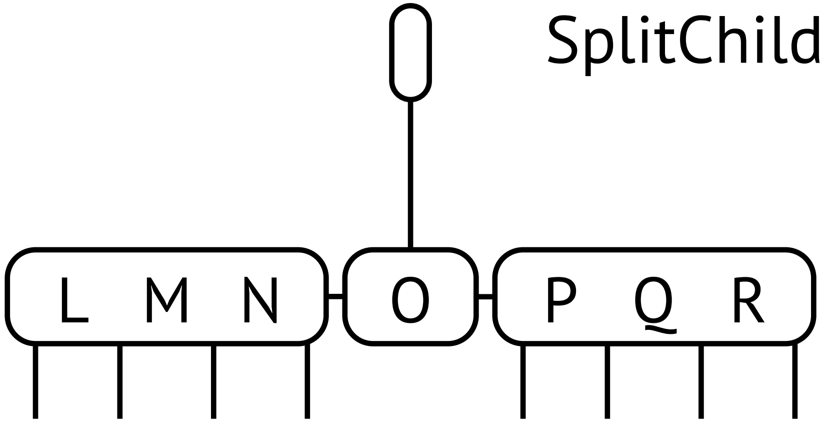
Erzeuge zuerst einen leeren Knoten …



SplitChild()

- Spezialfall: Wurzel

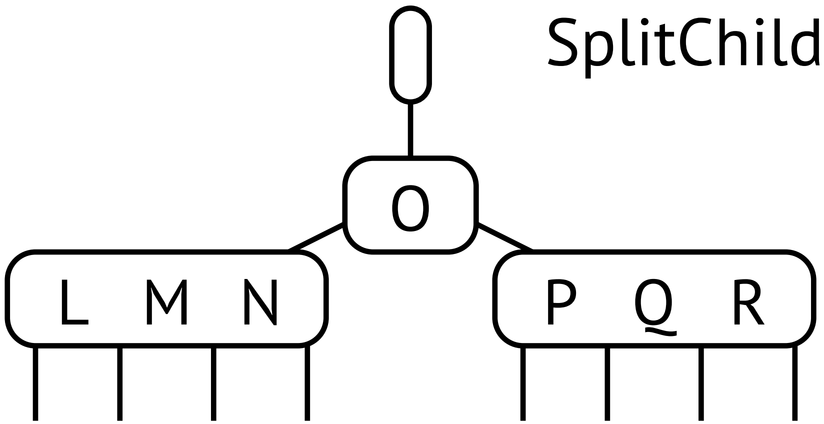
Erzeuge zuerst einen leeren Knoten ... dann SplitChild()



SplitChild()

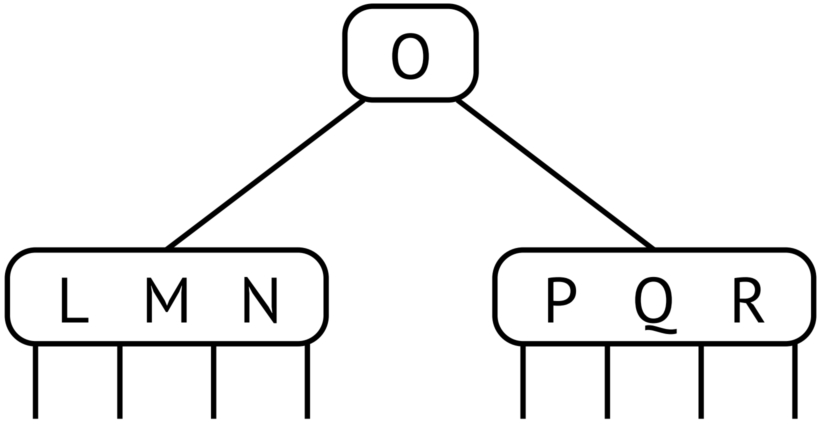
- Spezialfall: Wurzel

Erzeuge zuerst einen leeren Knoten ... dann SplitChild()



SplitChild()

- Spezialfall: Wurzel
- B-Bäume wachsen an der Wurzel



Insert()

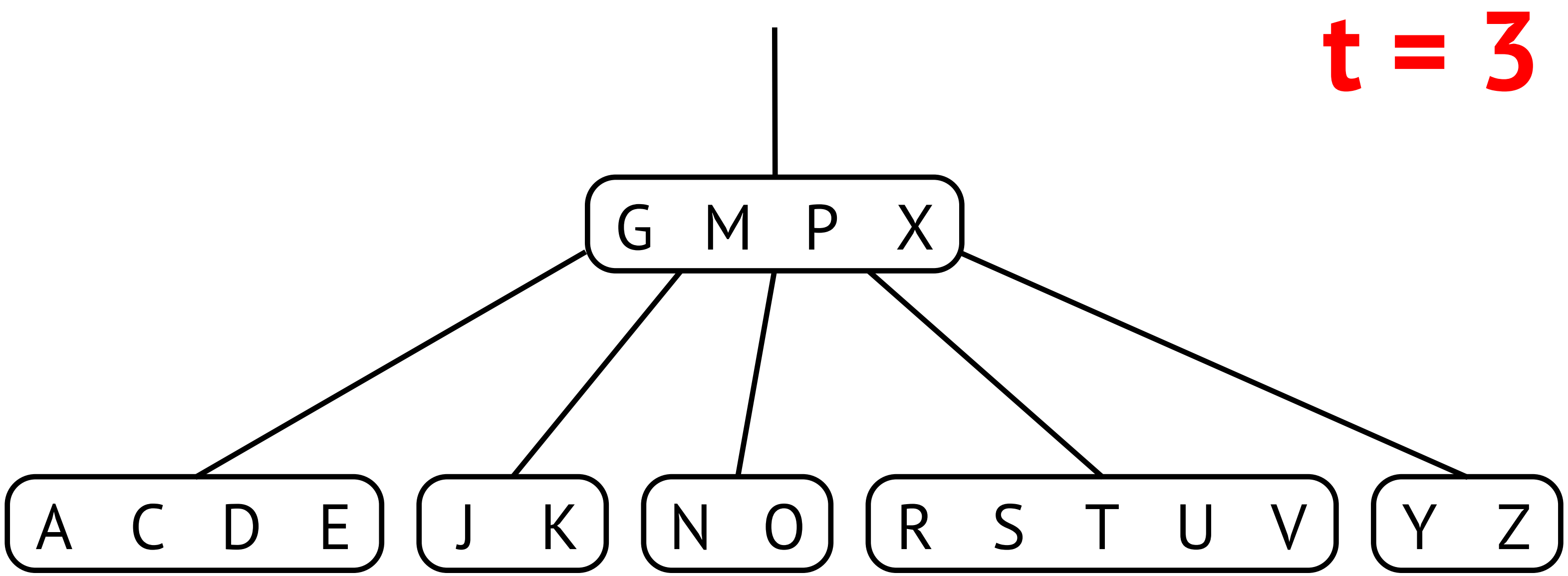
- Insert(T,k)

```
r ← root[T]
if n[r] = 2×t−1 then
    s ← AllocateNode()
    root[T] ← s
    leaf[s] ← false
    n[s] ← 0
    c0[s] ← r
    SplitChild(s,0)
    InsertNonFull(s,k)
else
    InsertNonFull(r,k)
```

InsertNonfull()

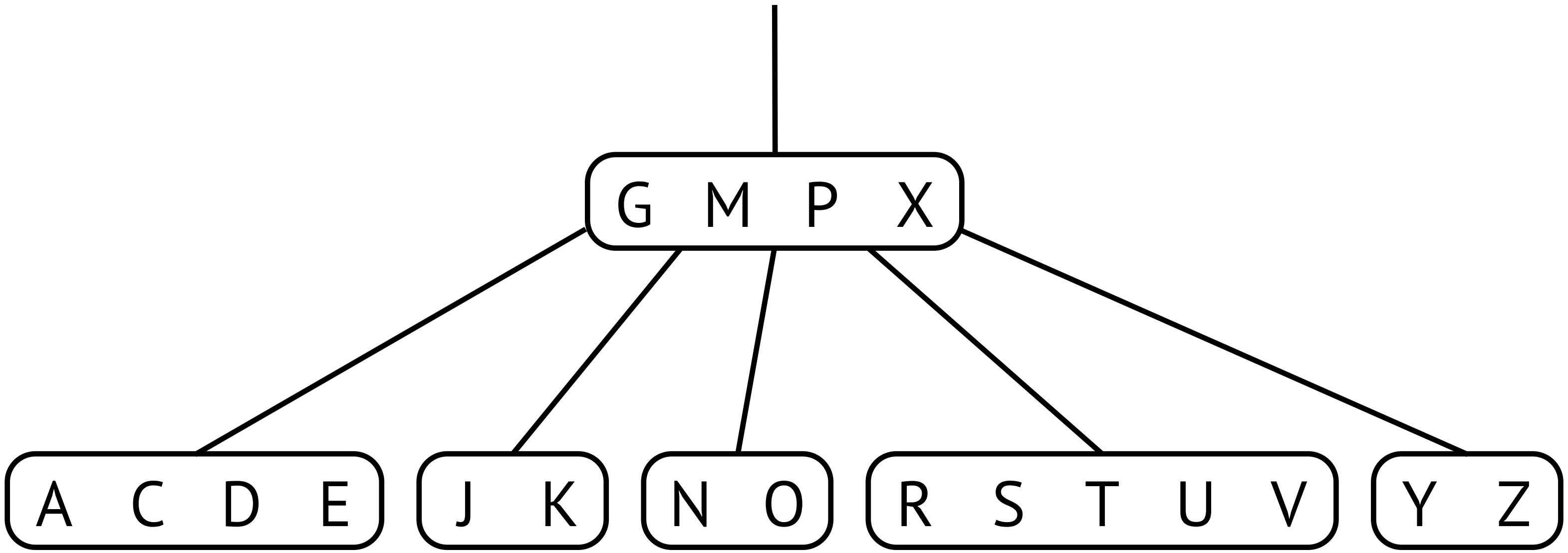
- InsertNonFull(x,k)
  if leaf[x] then
      find correct position for k and insert
      DiskWrite(x)
  else
      find correct subtree $c_i[x]$
      DiskRead($c_i[x]$)
      if $n[c_i[x]] = 2 \times t - 1$ then
          SplitChild(x,i)
          if $k > key_i[x]$ then
              $i \leftarrow i + 1$
      InsertNonFull($c_i[x]$,k)

Beispiel



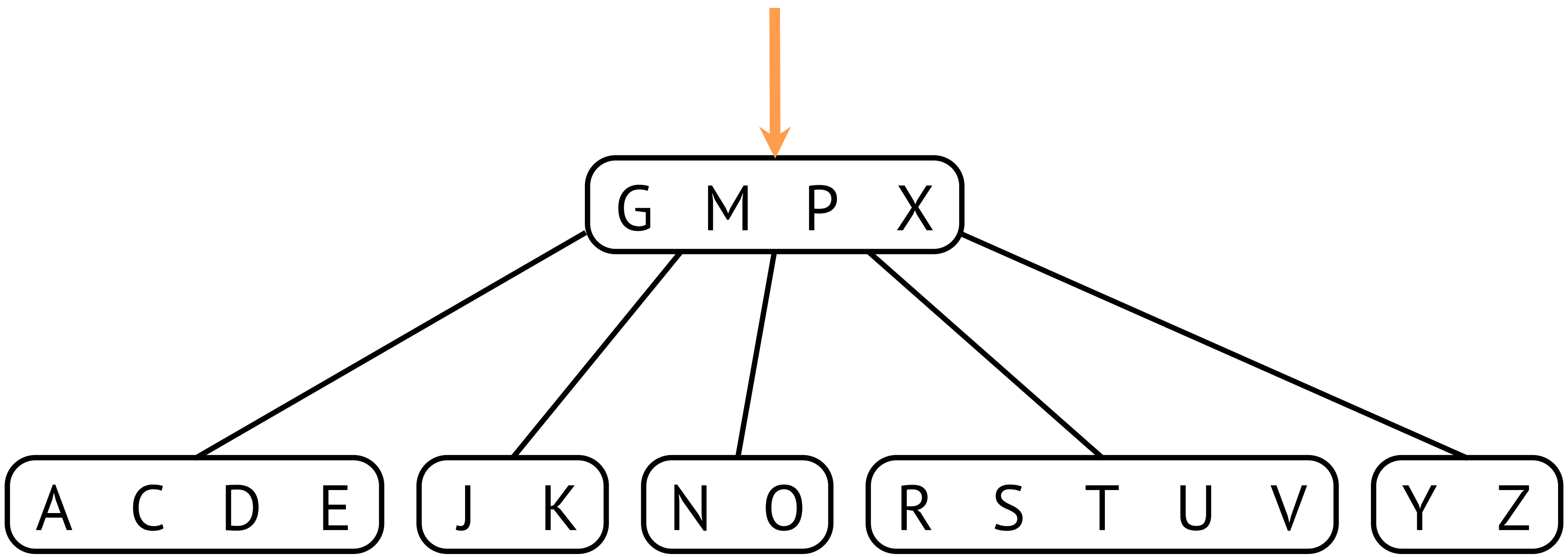
Beispiel

- Insert(B)



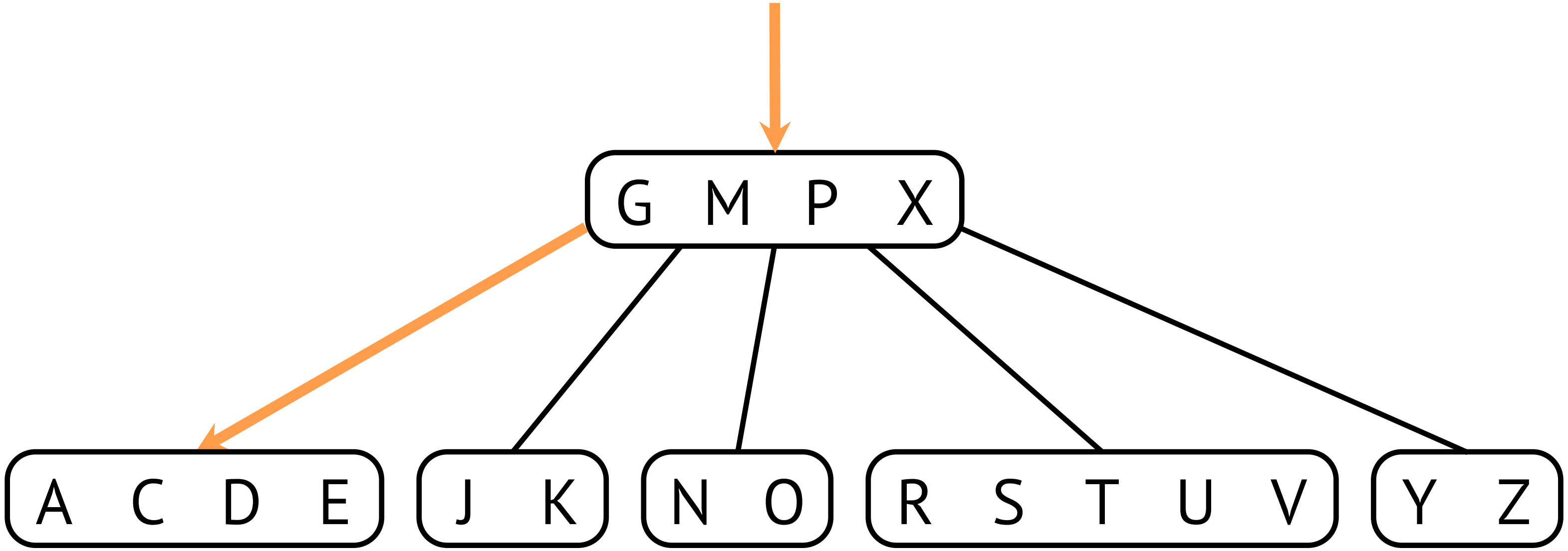
Beispiel

- Insert(B)



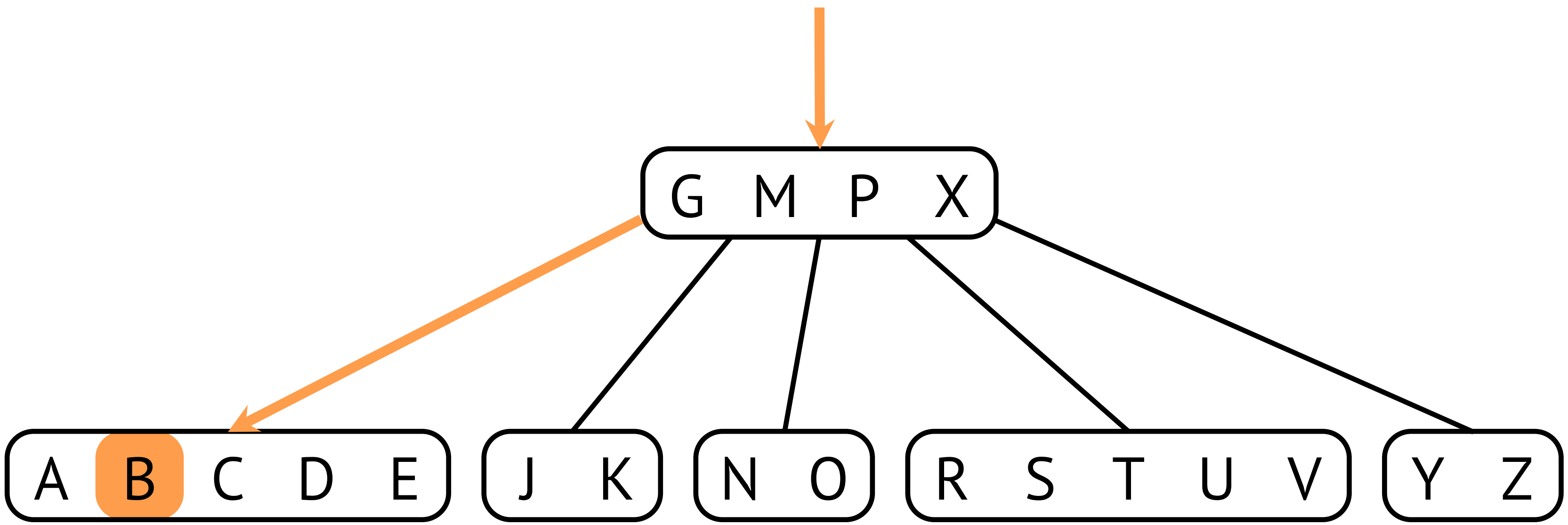
Beispiel

- Insert(B)



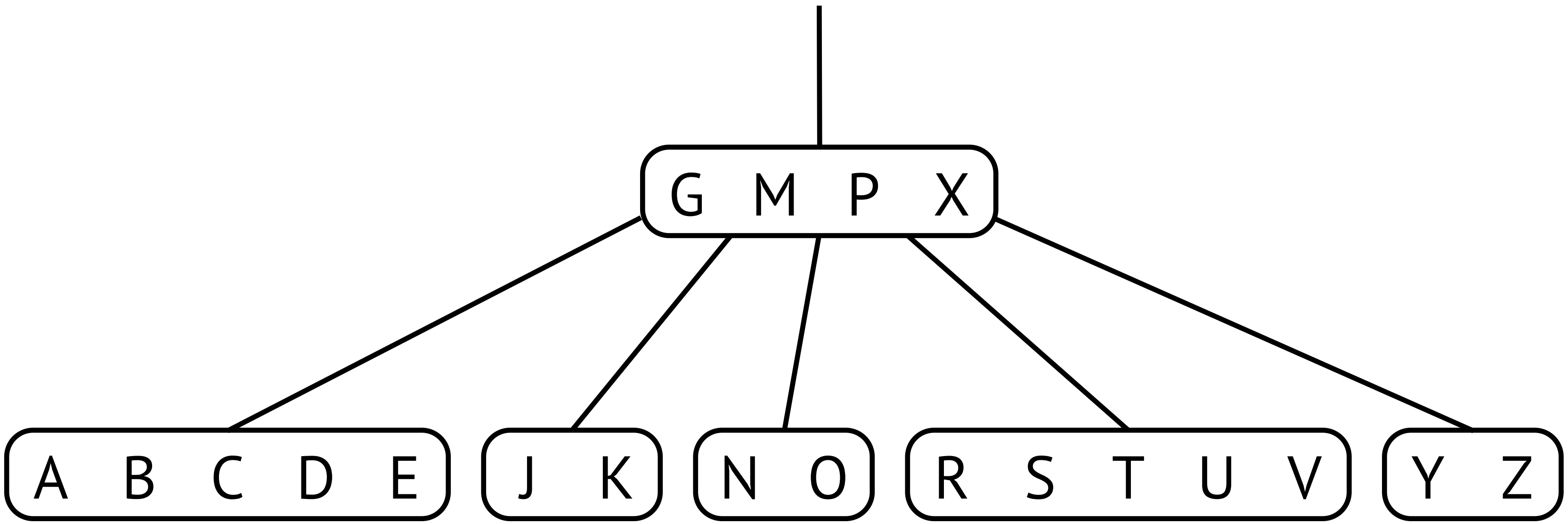
Beispiel

- Insert(B)



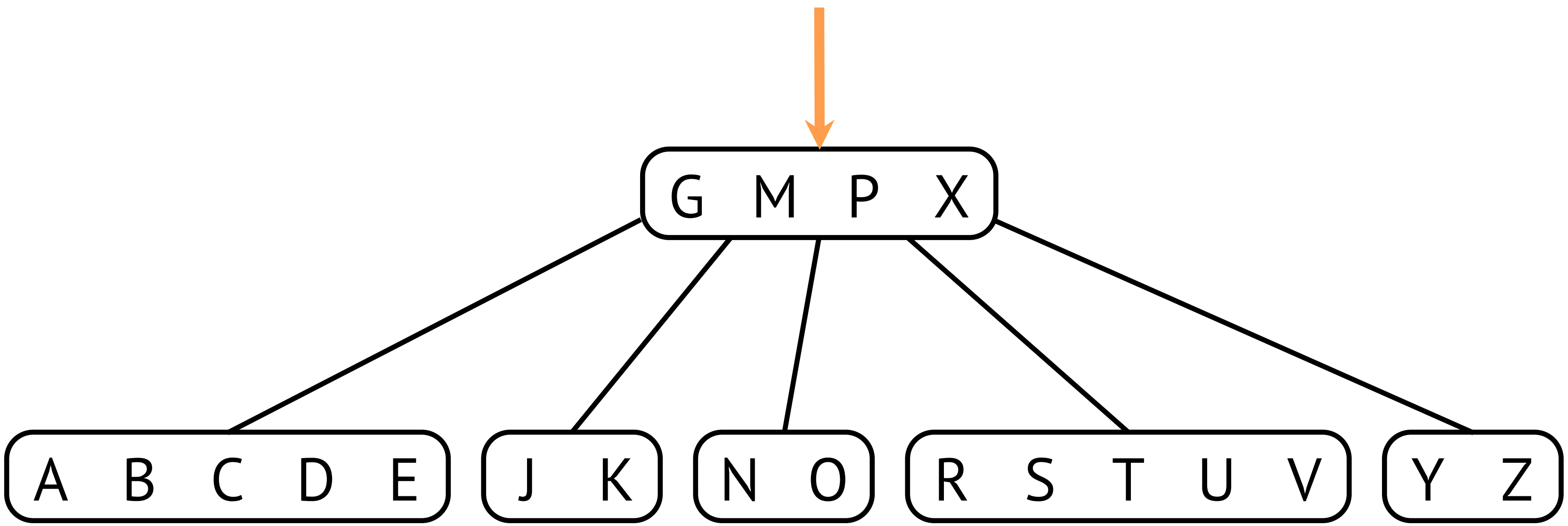
Beispiel

- Insert(Q)



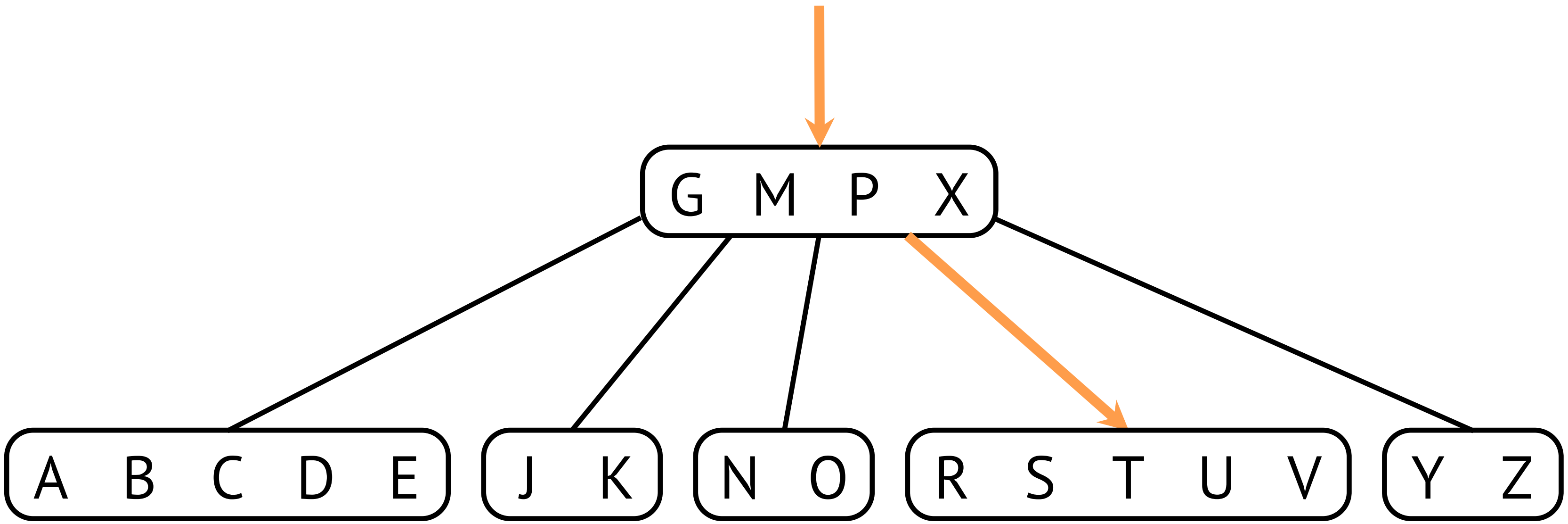
Beispiel

- Insert(Q)



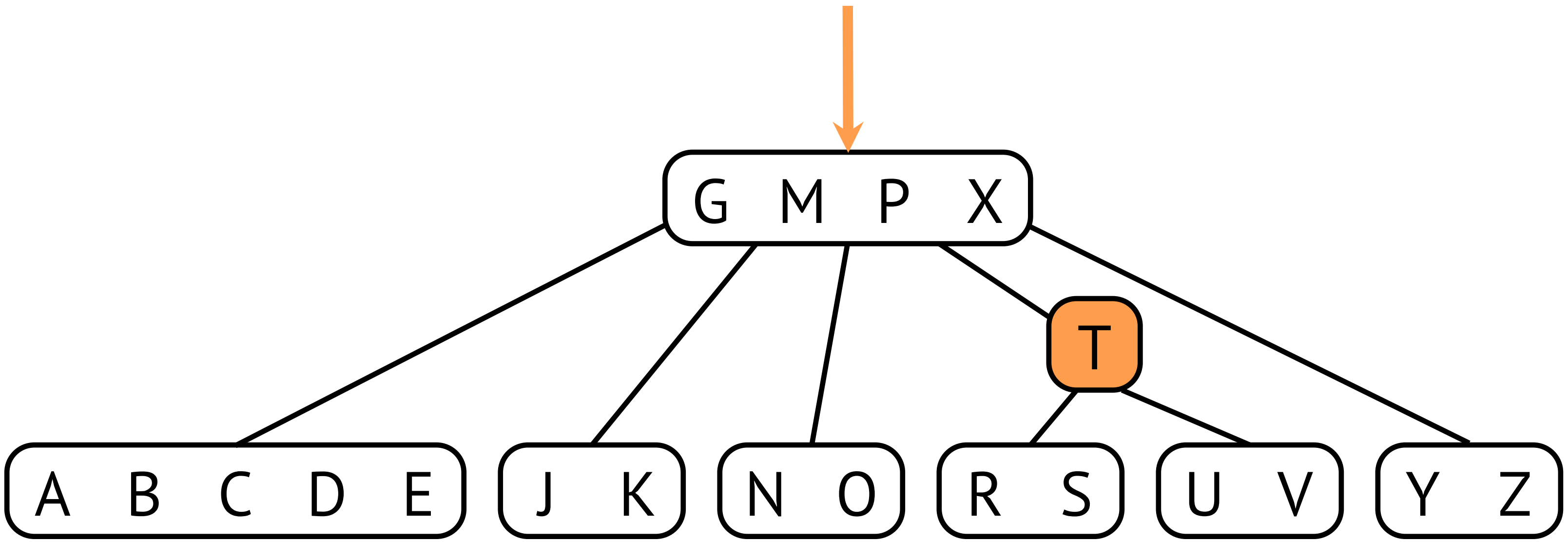
Beispiel

- Insert(Q)



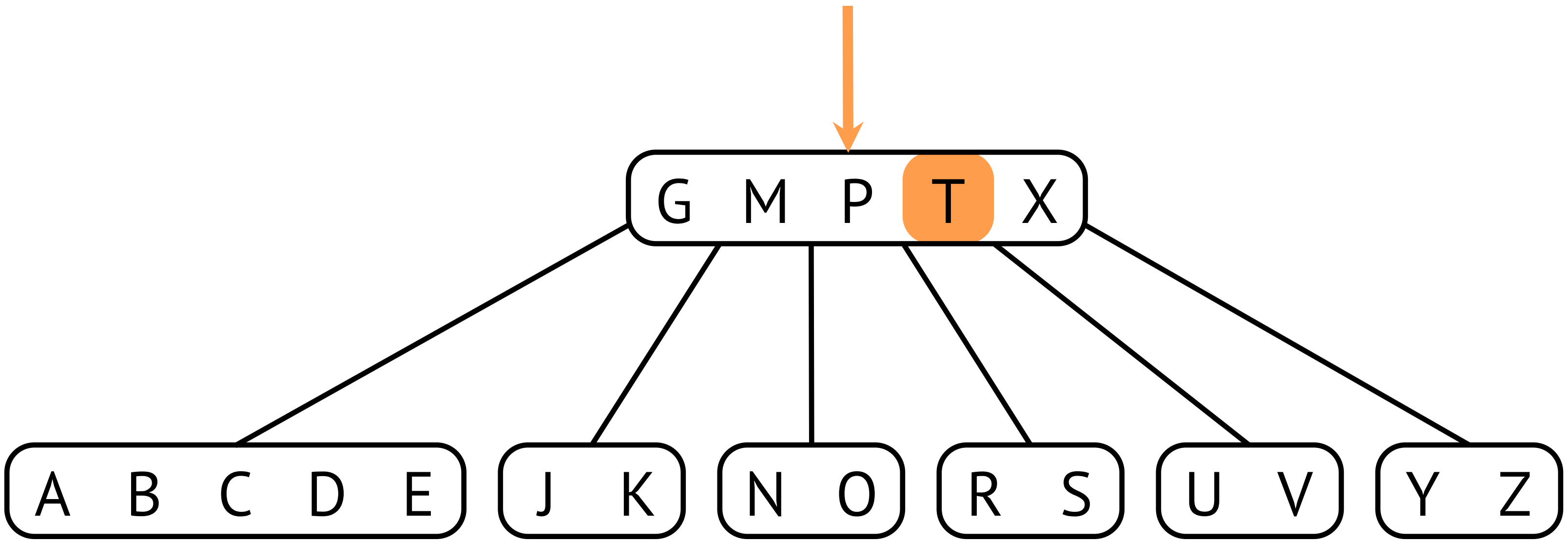
Beispiel

- Insert(Q)



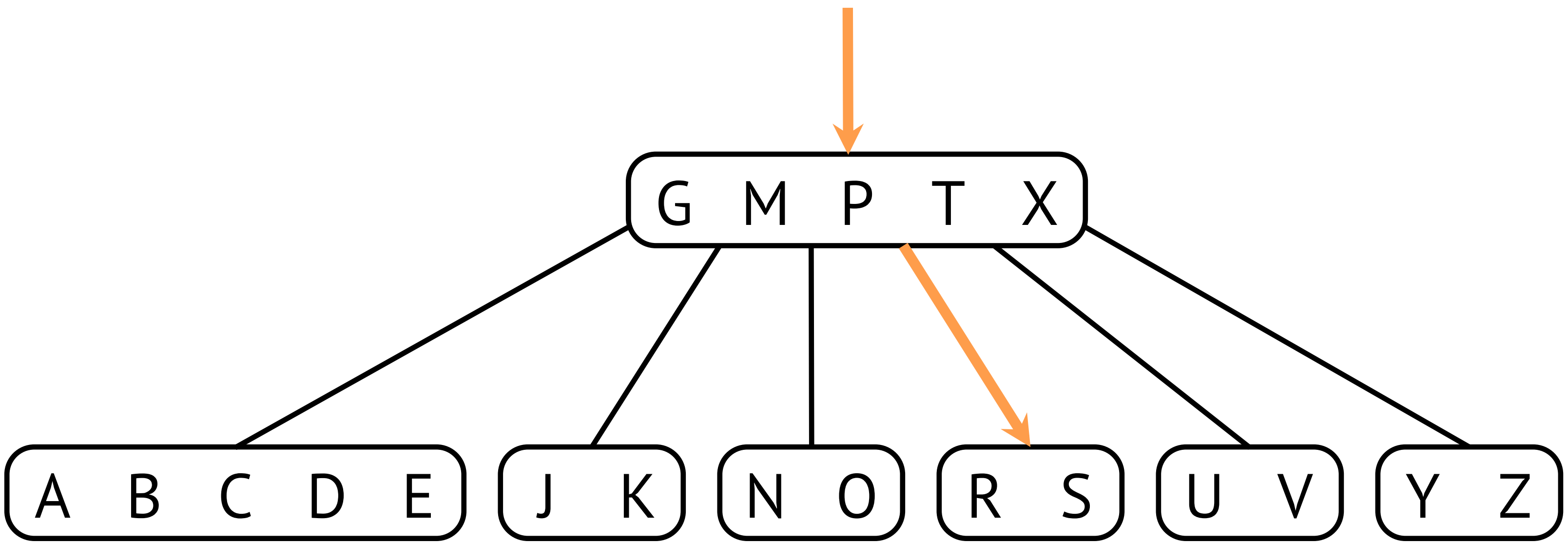
Beispiel

- Insert(Q)



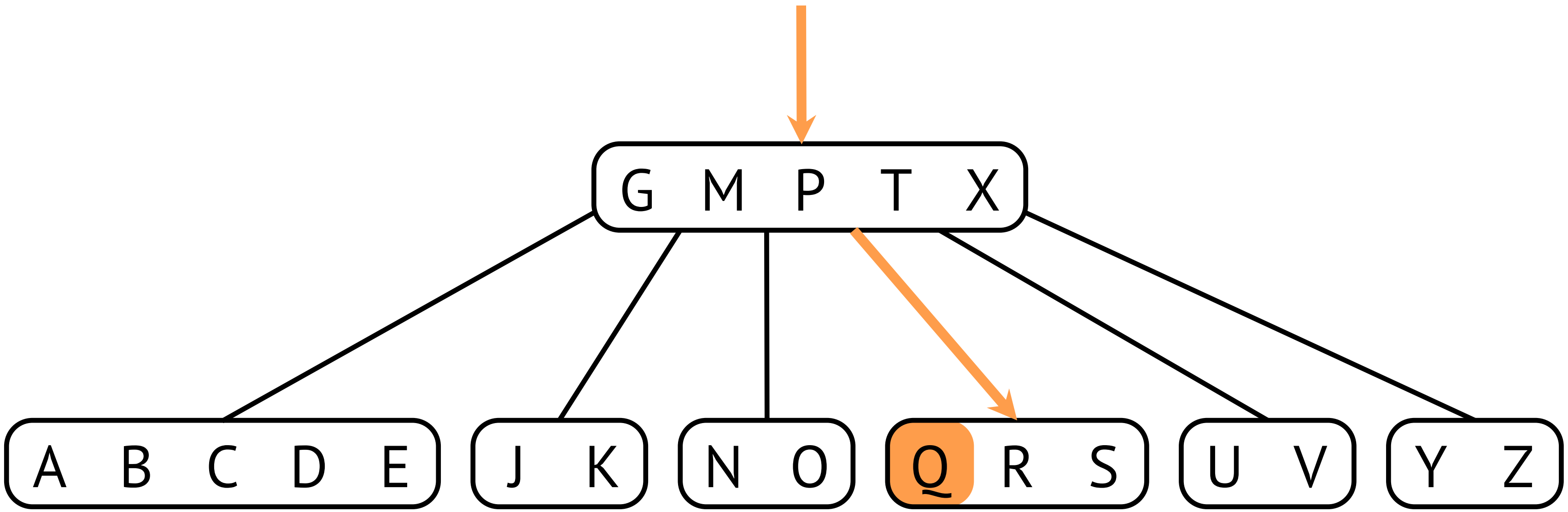
Beispiel

- Insert(Q)



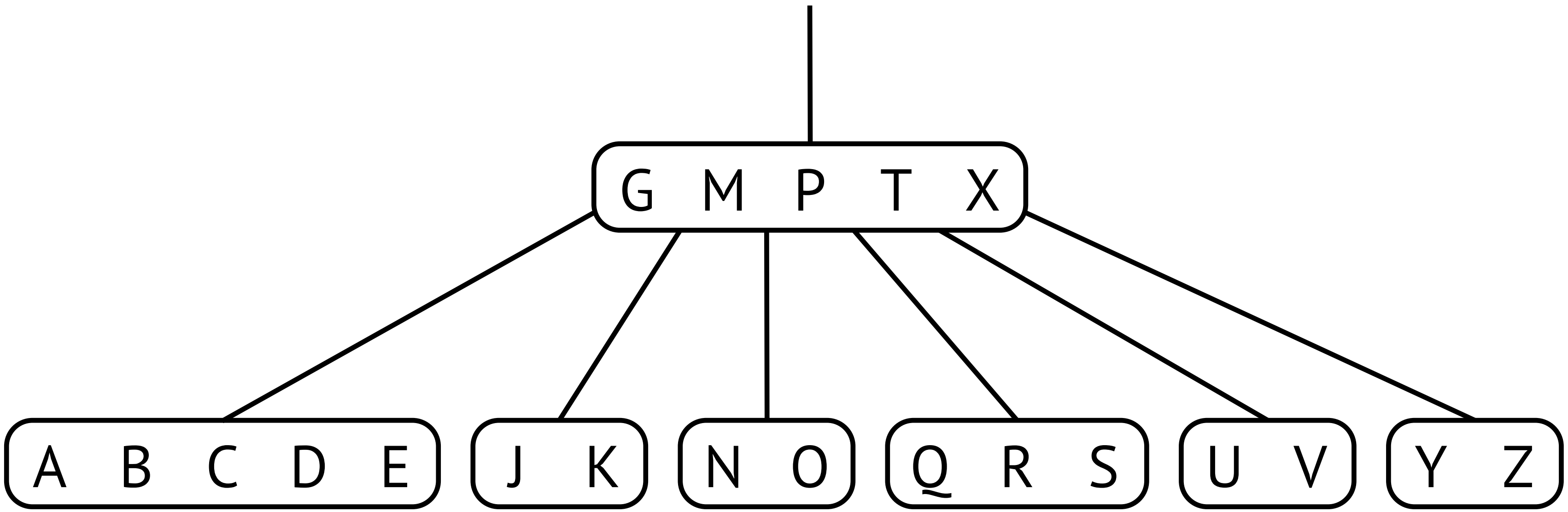
Beispiel

- Insert(Q)



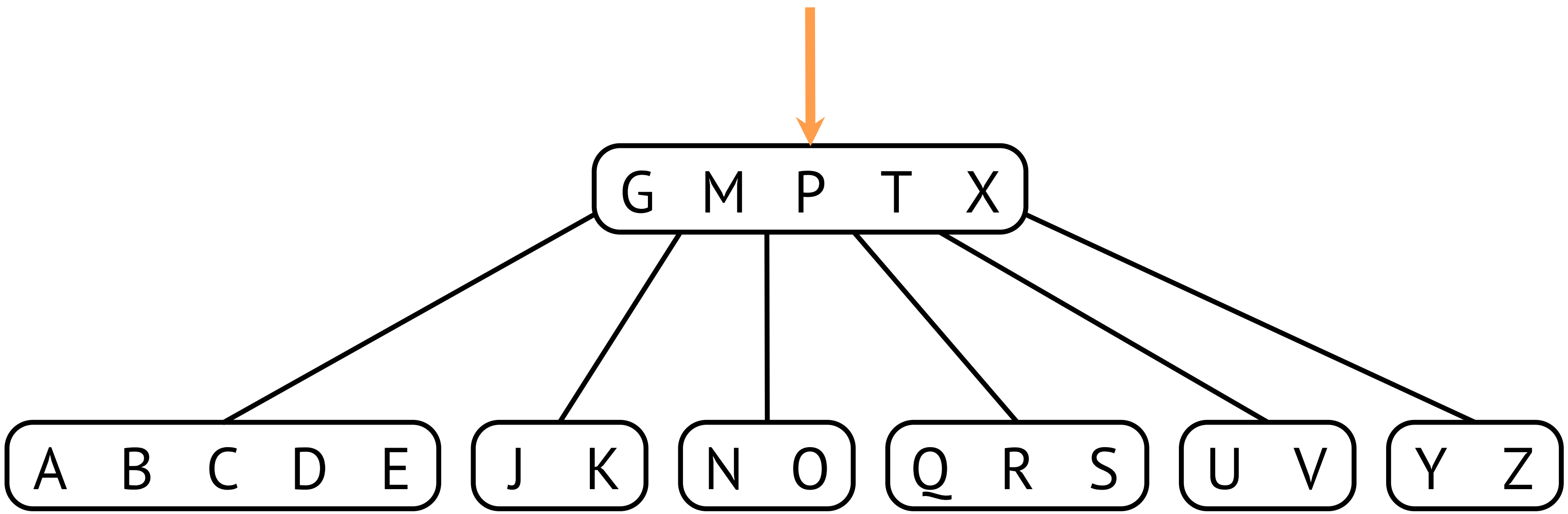
Beispiel

- Insert(L)



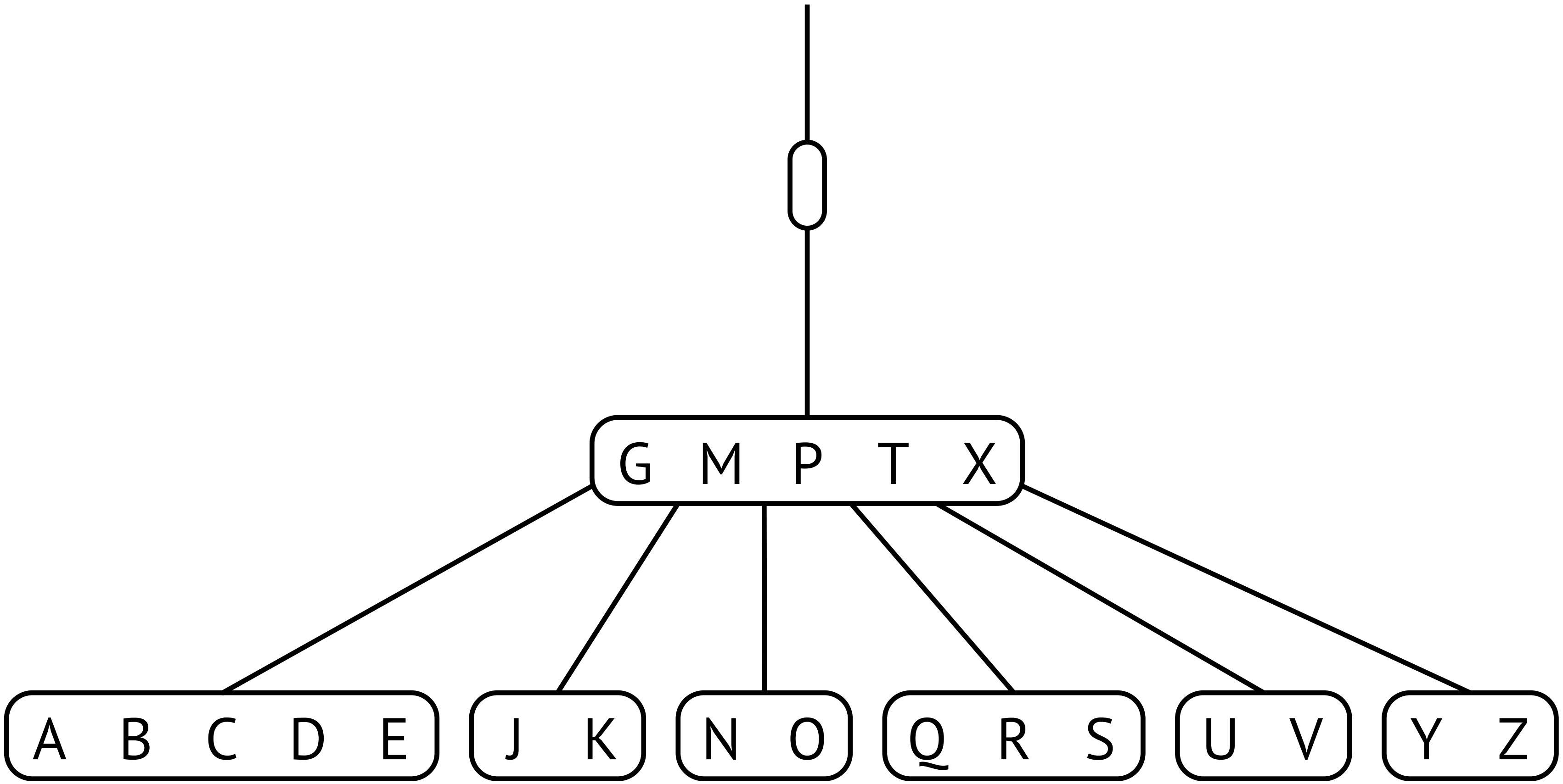
Beispiel

- Insert(L)



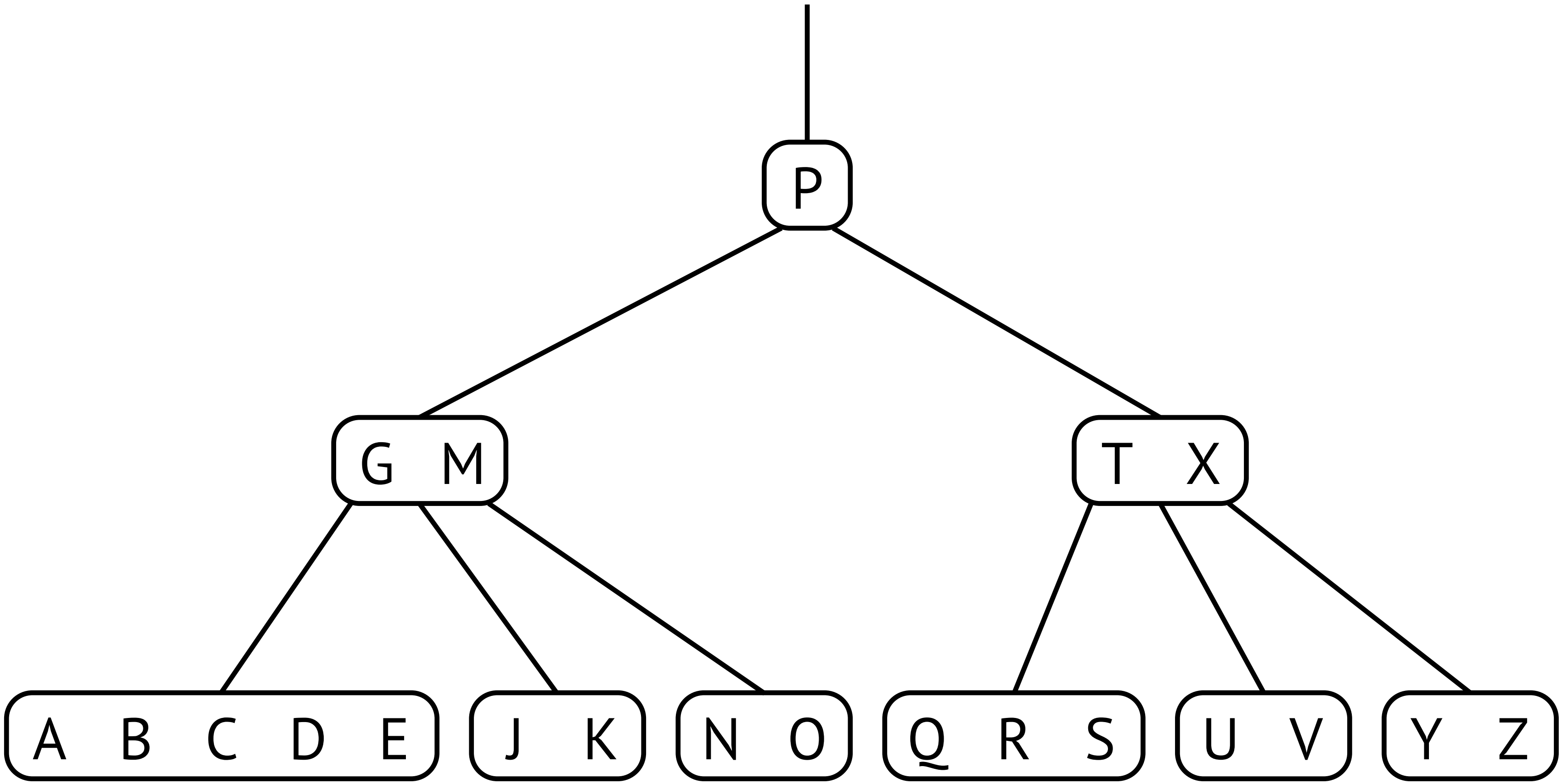
Beispiel

- Insert(L)



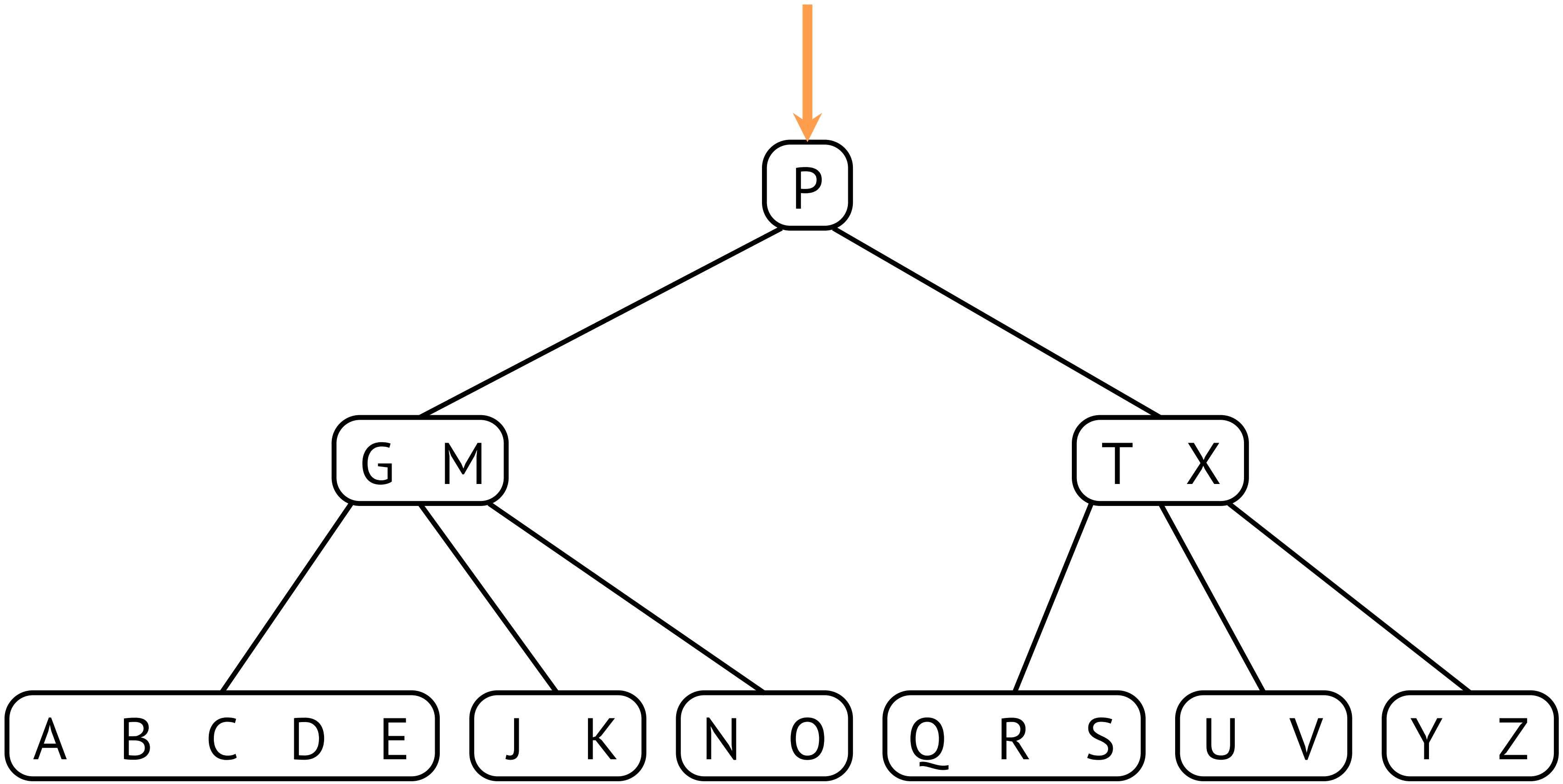
Beispiel

- Insert(L)



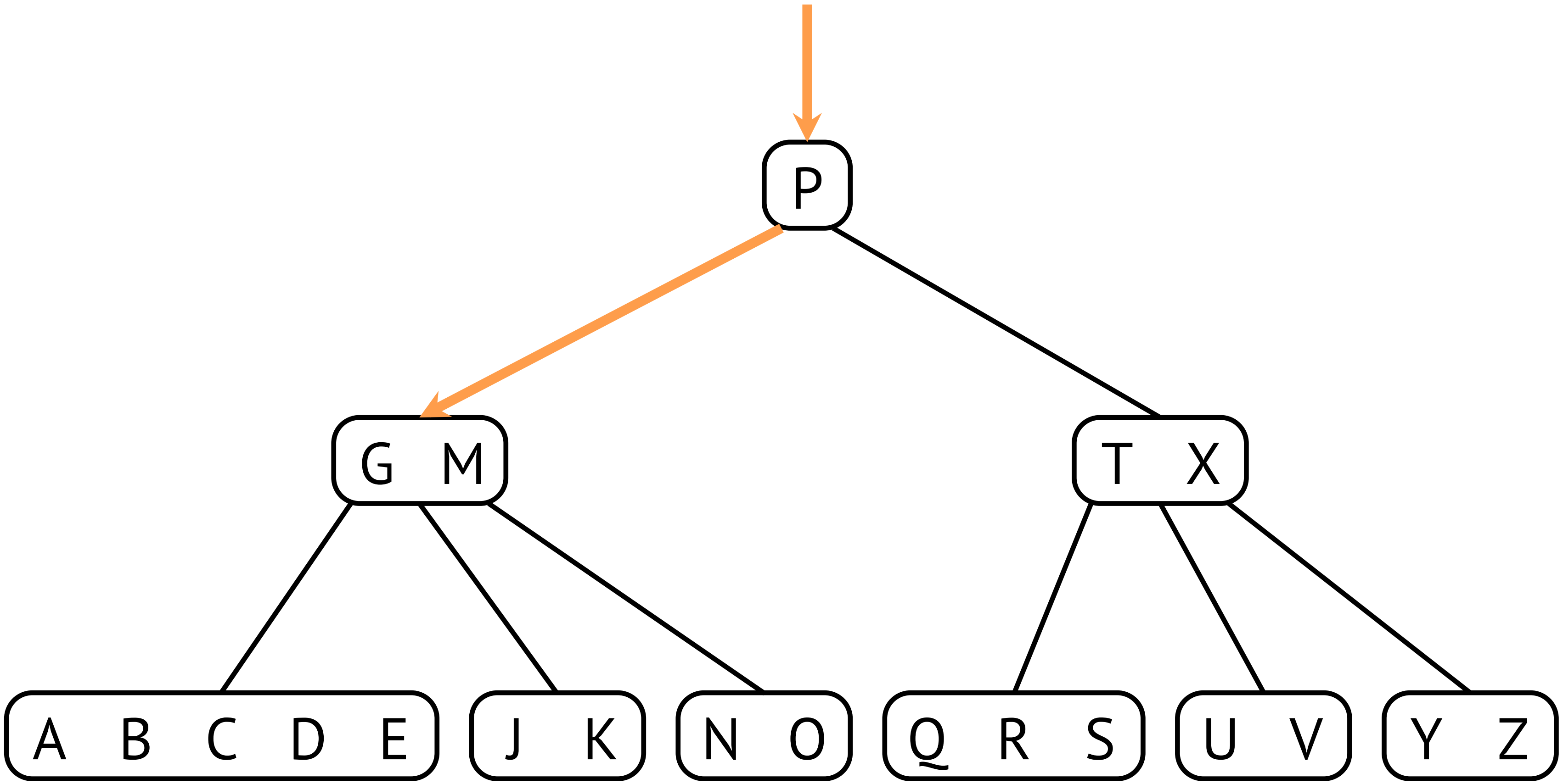
Beispiel

- Insert(L)



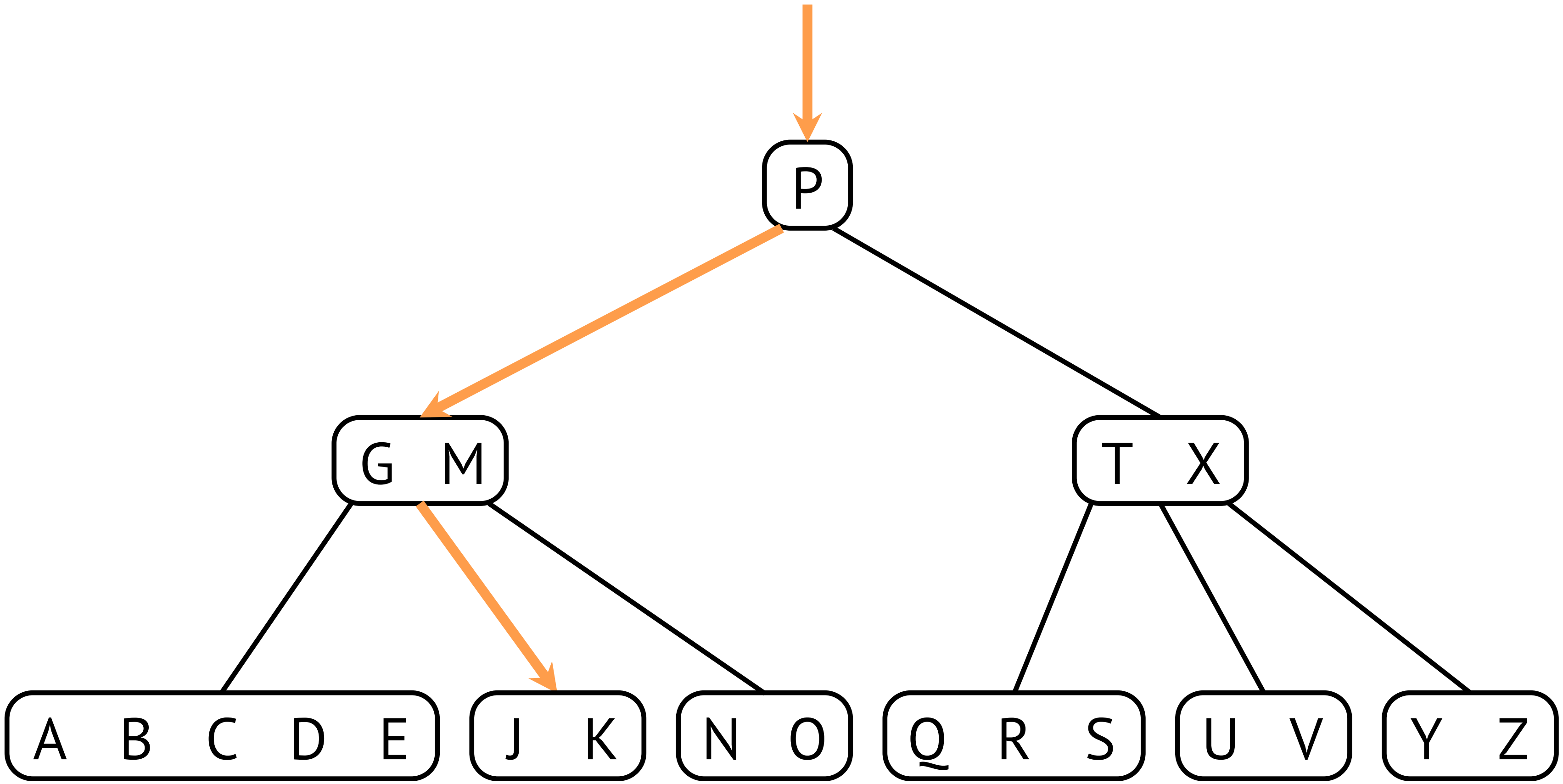
Beispiel

- Insert(L)



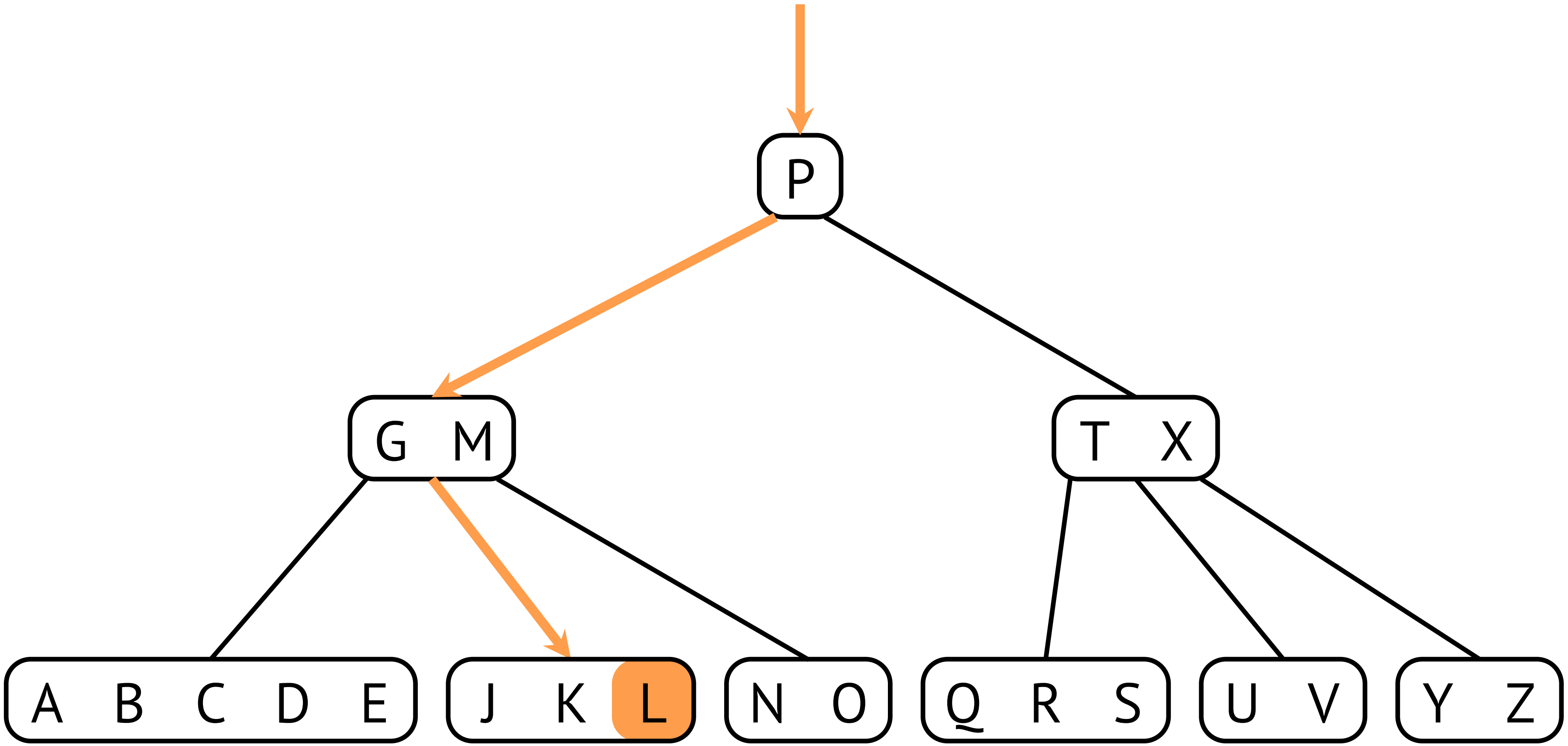
Beispiel

- Insert(L)



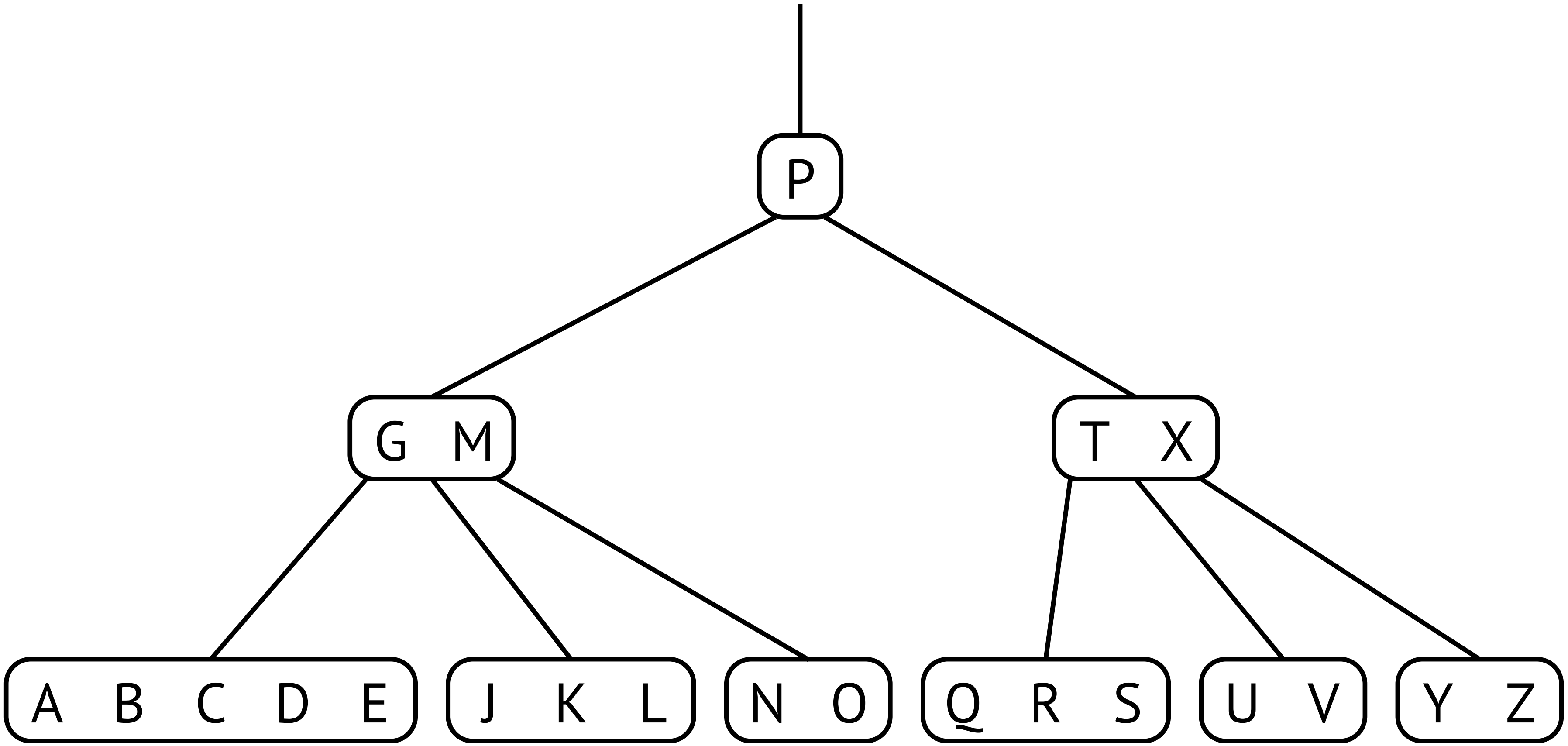
Beispiel

- Insert(L)



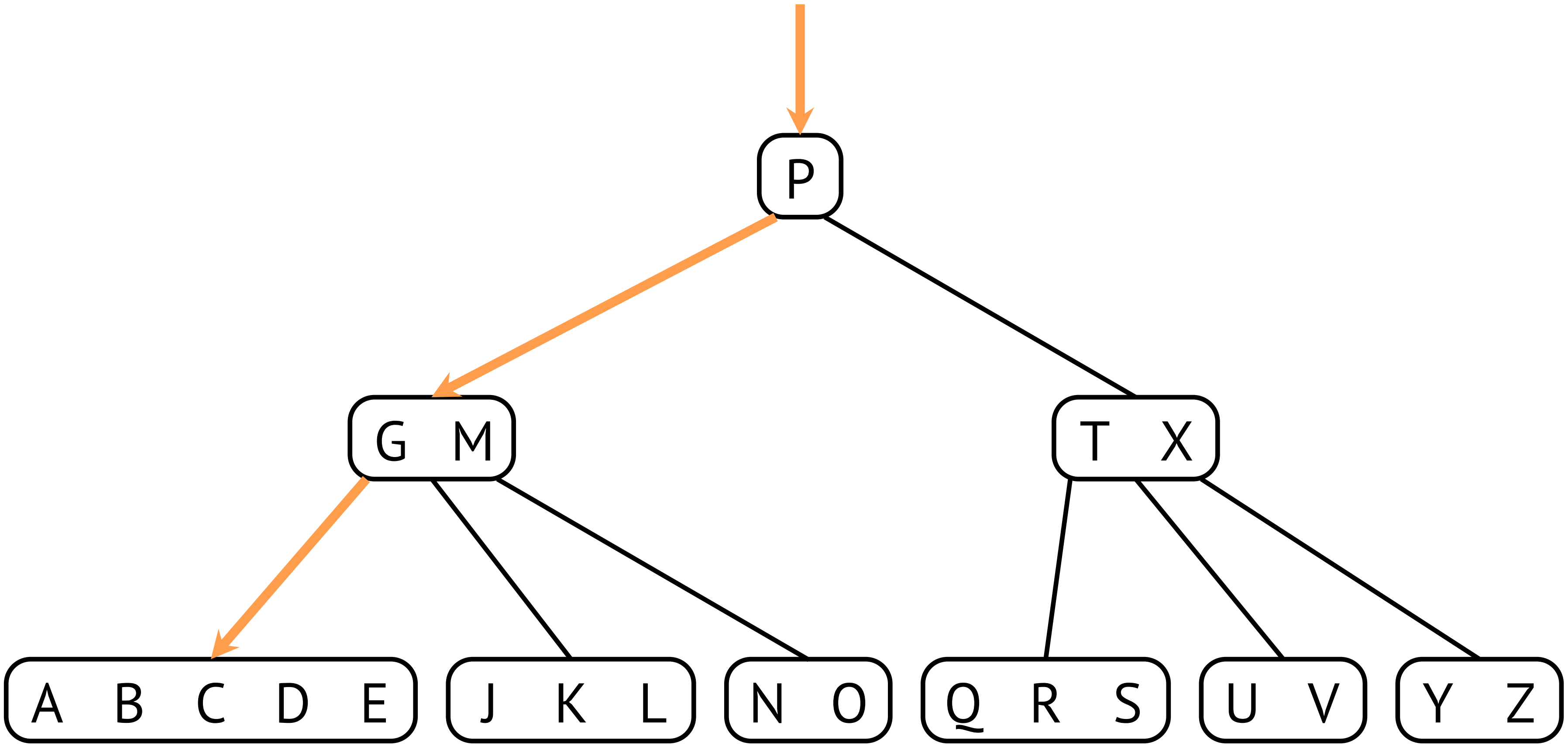
Beispiel

- Insert(F)



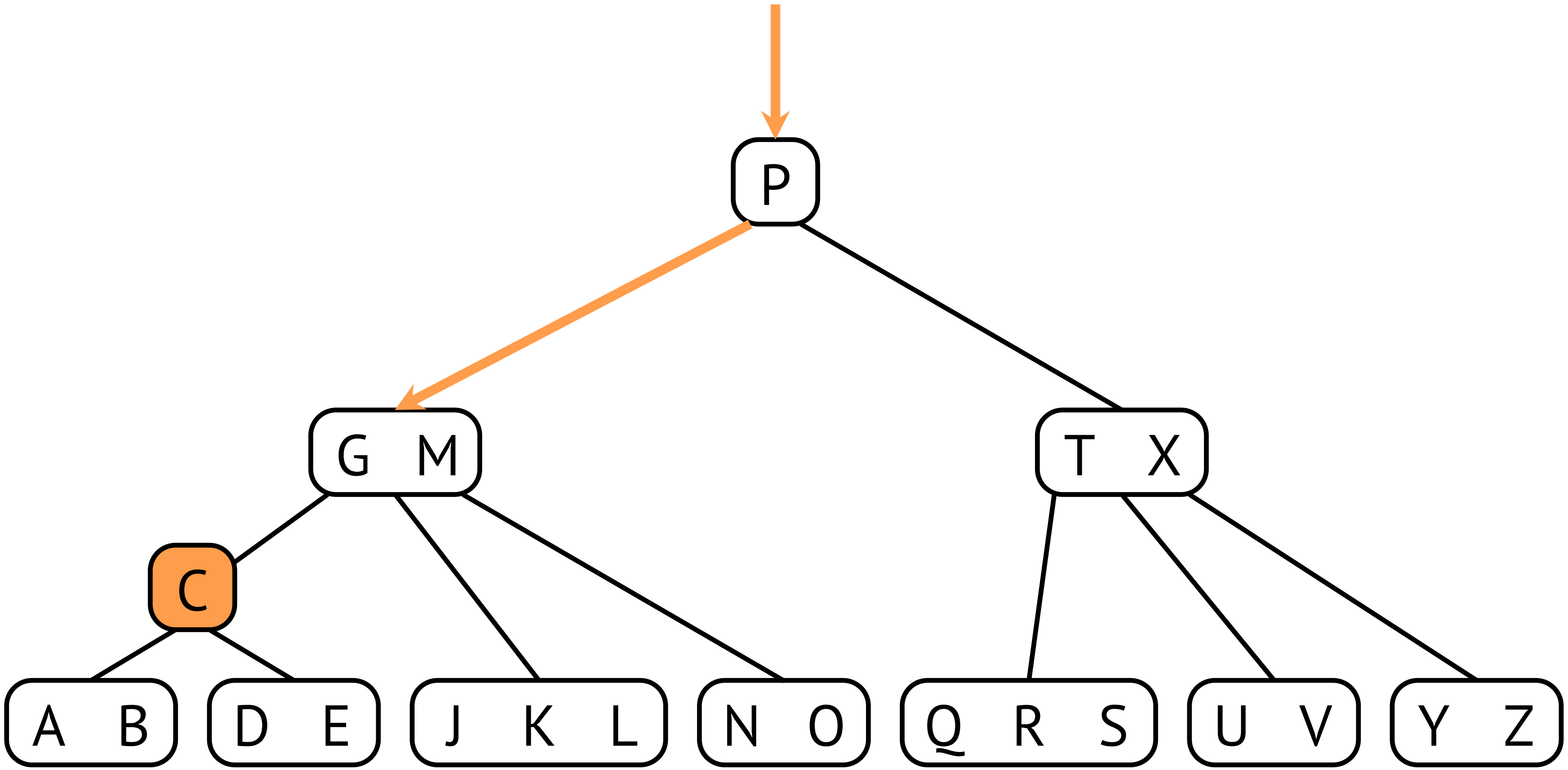
Beispiel

- Insert(F)



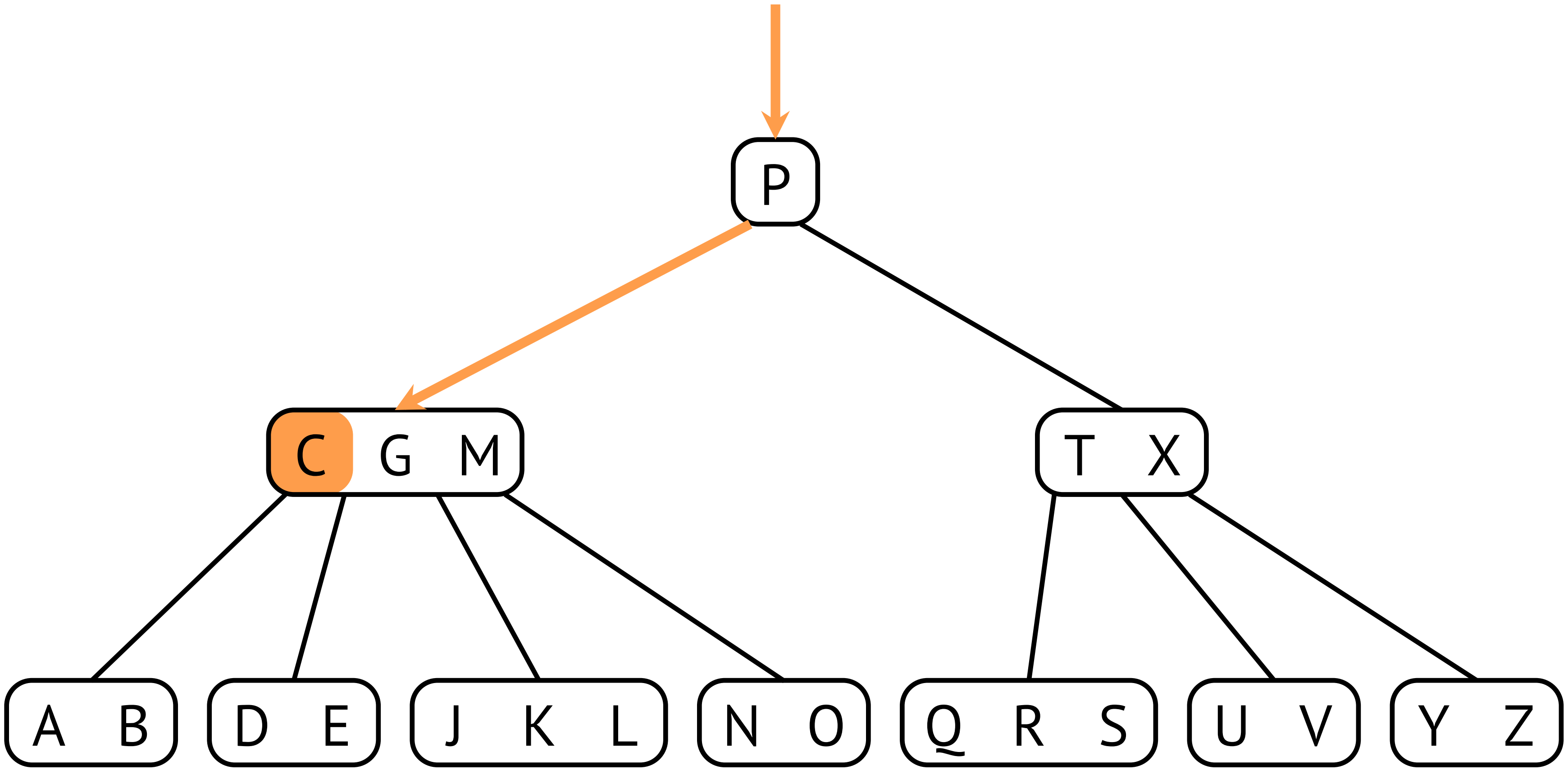
Beispiel

- Insert(F)



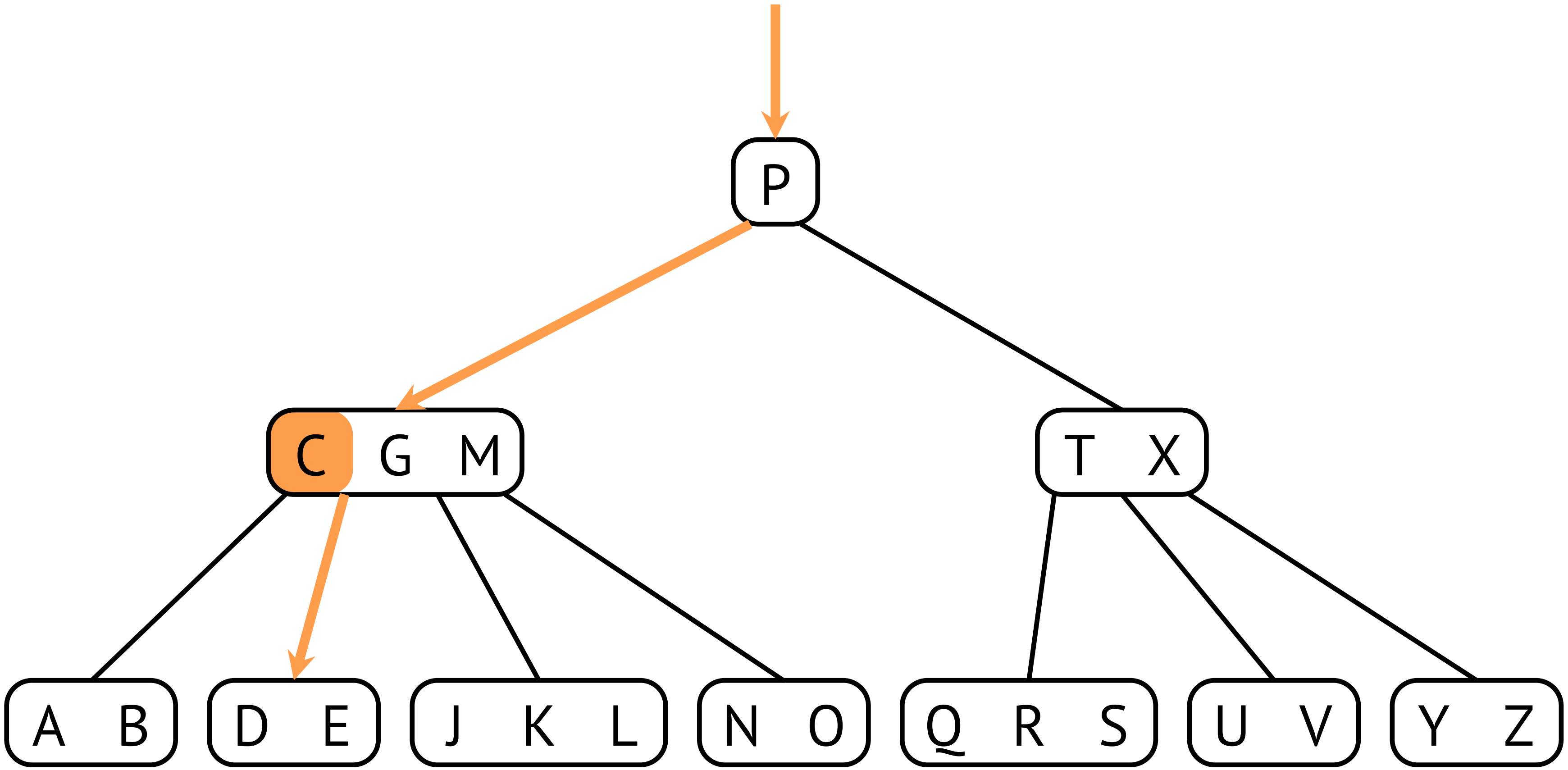
Beispiel

- Insert(F)



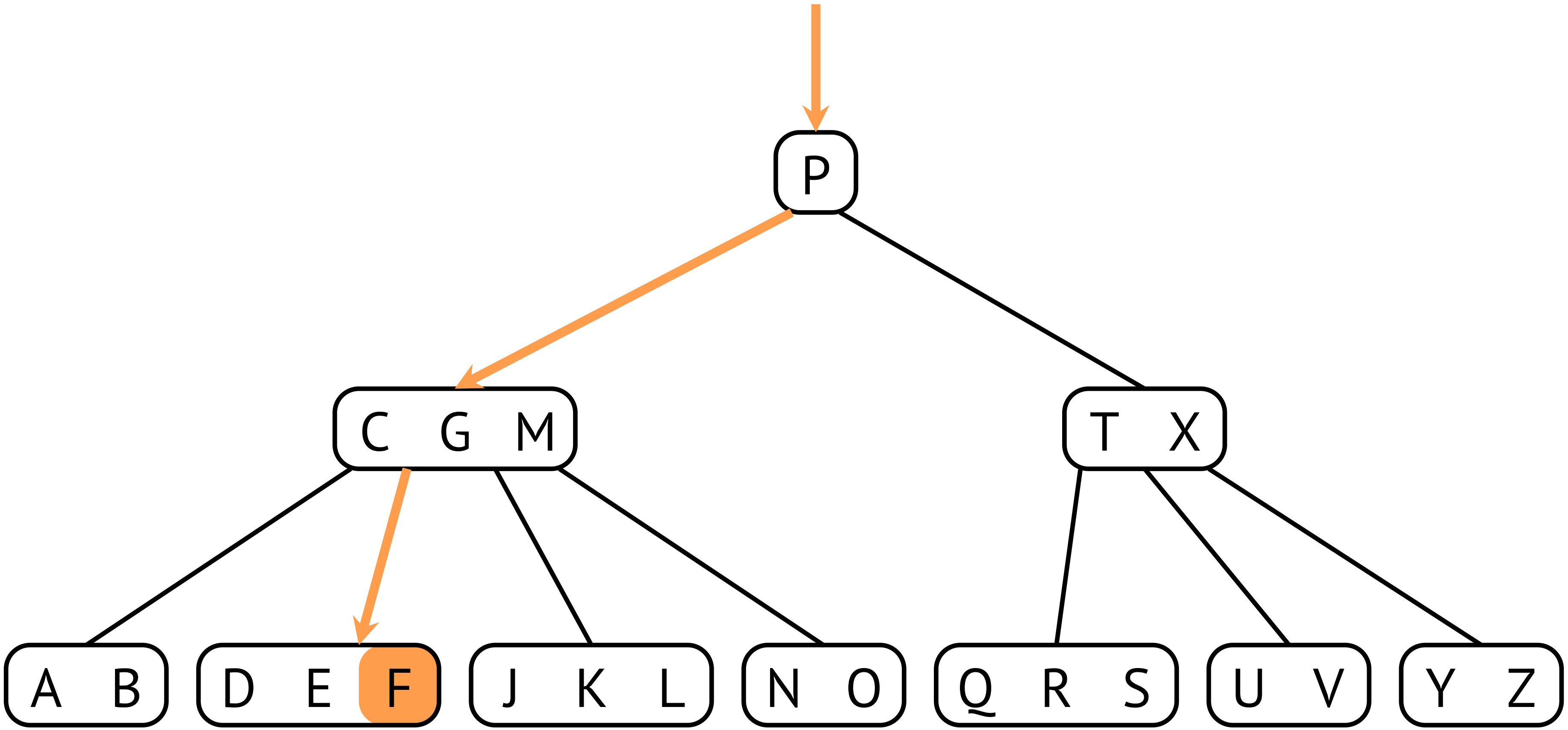
Beispiel

- Insert(F)



Beispiel

- Insert(F)



Delete()

- Unkritisch für Blattknoten mit hinreichendem Füllgrad (häufigster Fall)
- Bei Unterlauf müssen Knoten verschmolzen werden
- Kann/muss nach oben propagiert werden
- One-Pass-Formulierung?

Delete()

- Delete() sucht den Schlüssel entlang eines Pfades
- Eine Verschmelzung findet nur statt, wenn die entsprechenden Knoten beide einen Füllgrad $\leqslant t-1$ haben.
- Teste vor dem Abstieg, ob der Nachfolger Füllgrad $\geqslant t$ hat (dann ist Verschmelzung ausgeschlossen)
- Falls dies nicht der Fall ist, erzwinge dies durch direkte Umstrukturierung

Delete()

- Wie bei Insert() sorgen wir dafür, dass der aktuelle Knoten hinreichend gefüllt ist, so dass keine Verschmelzung notwendig wird (Schleifeninvariante)
- Falls ein Knoten nicht hinreichend gefüllt wäre, hätten wir das bereits bei der Bearbeitung des Vaterknotens bemerkt und behoben.

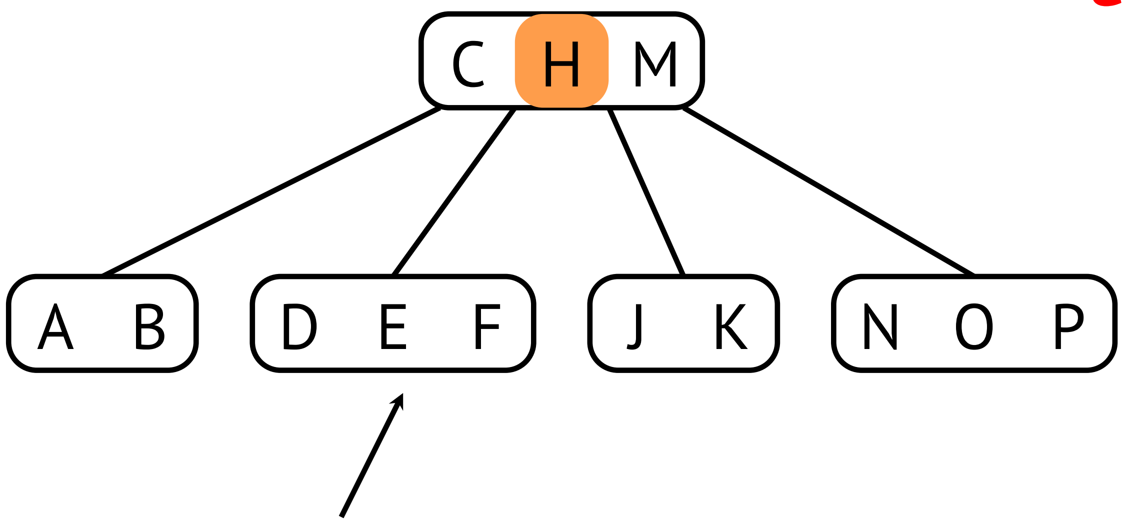
Delete()

- Hilfsoperationen
  - Steal()
  - Merge()
  - Rotate()

Steal()

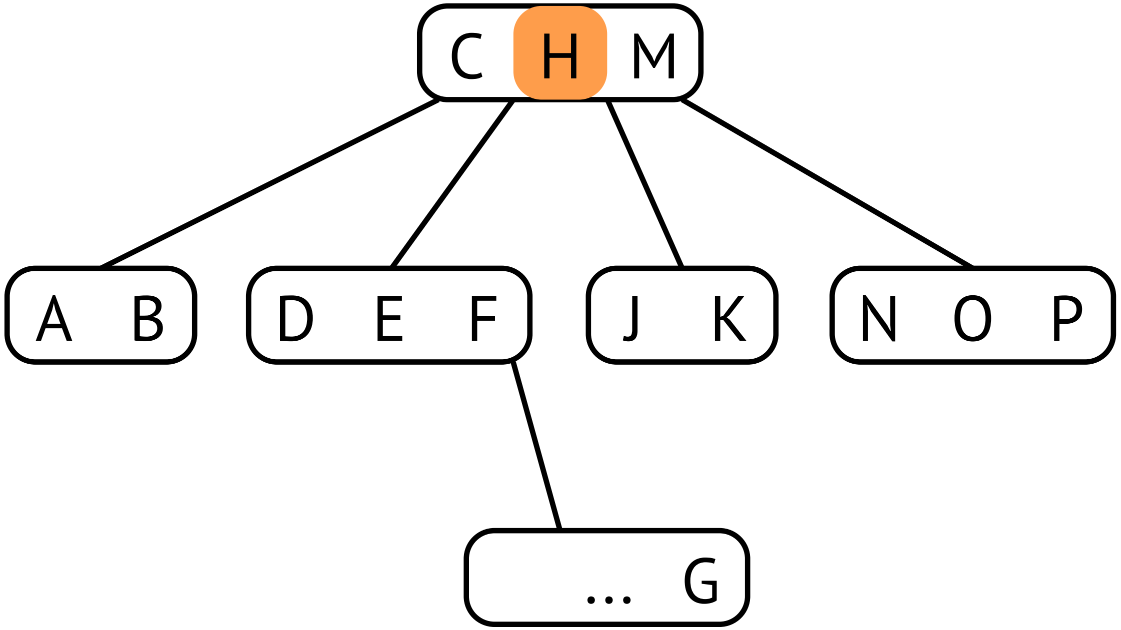
- Lösche Element aus **aktuellem** Knoten

t = 3

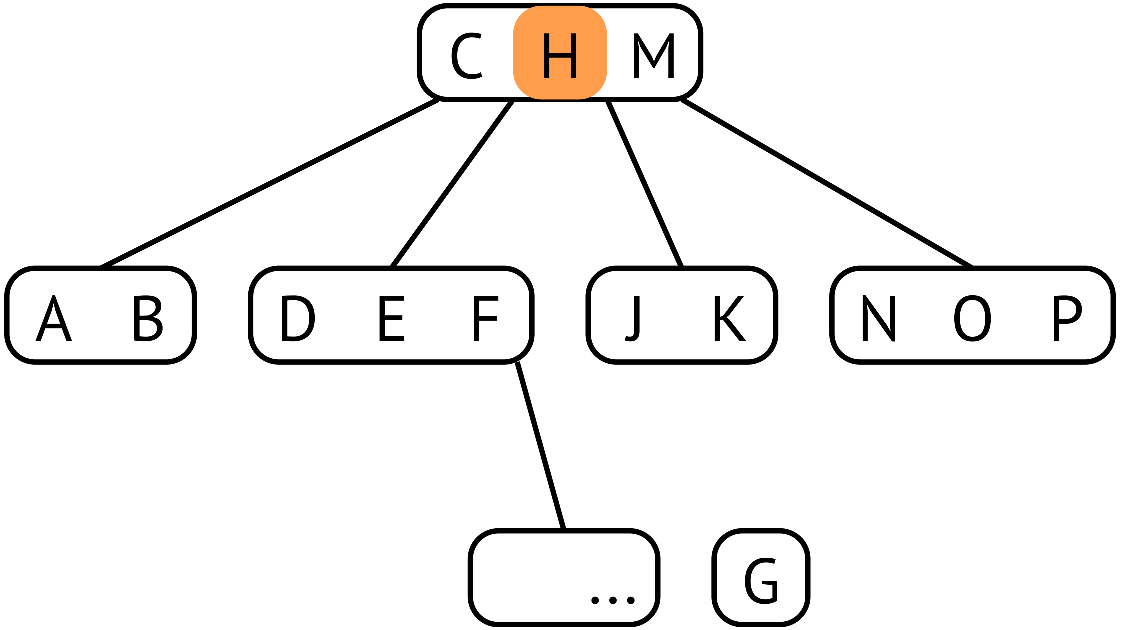


mehr als t−1 Knoten → rekursiver Abstieg möglich

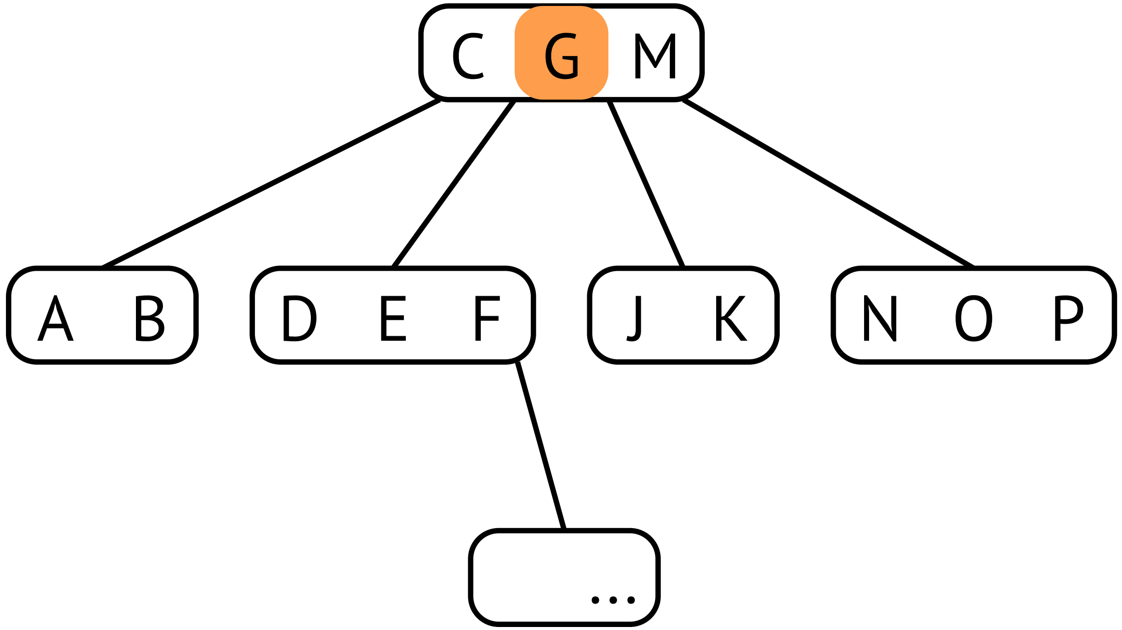
Steal()



Steal()

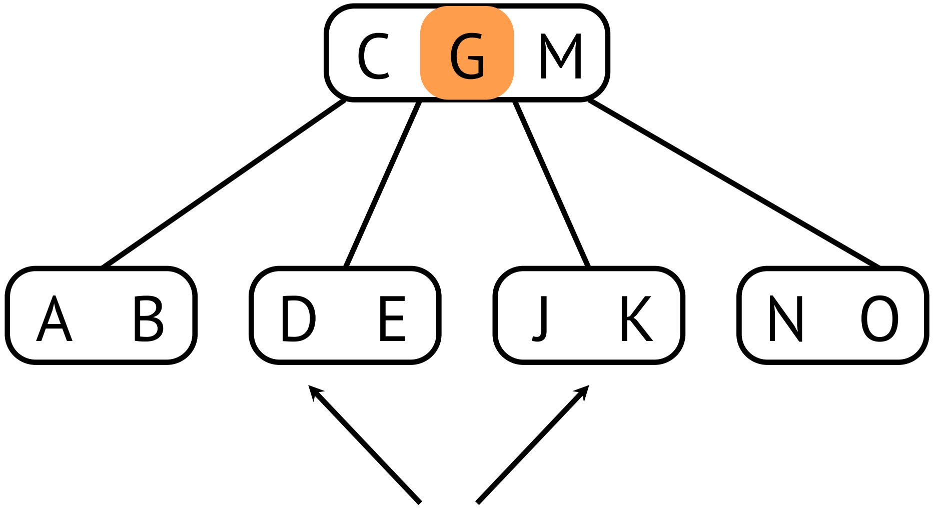


Steal()



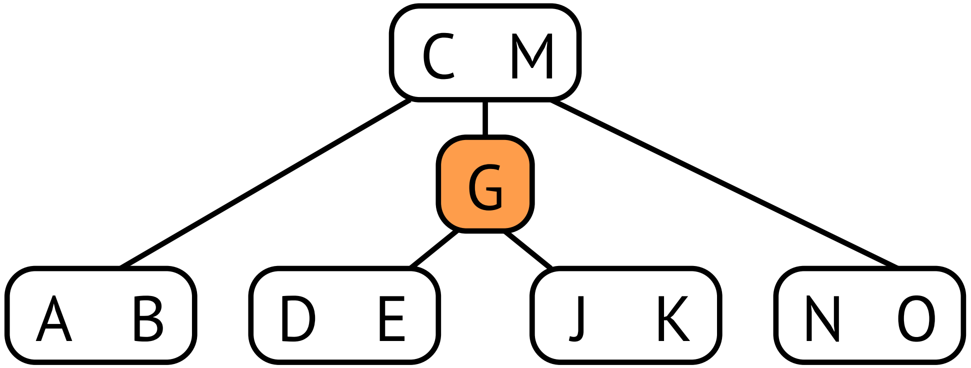
Merge()

- Lösche Element aus **aktuellem** Knoten

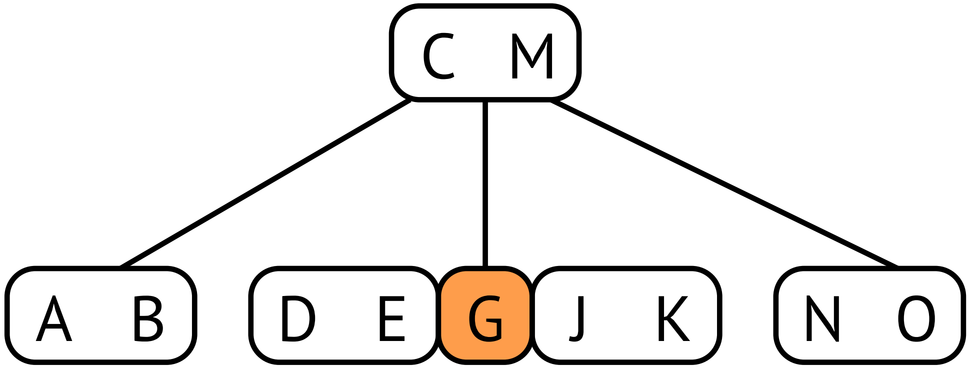


nur t−1 Knoten → rekursiver Abstieg **nicht** möglich

Merge()

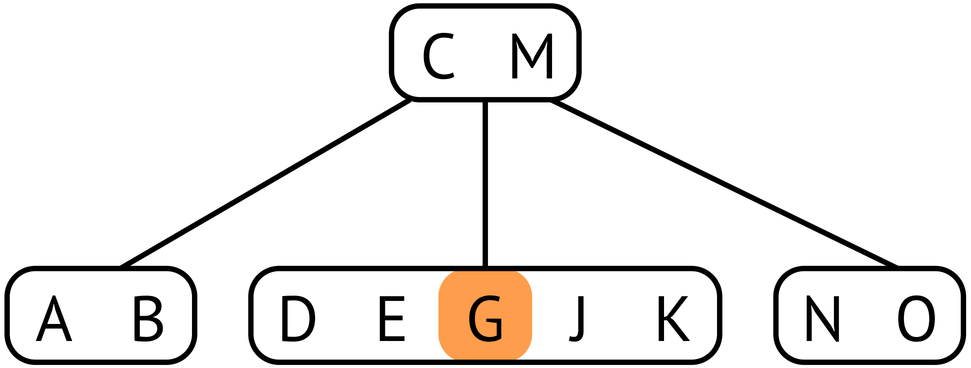


Merge()



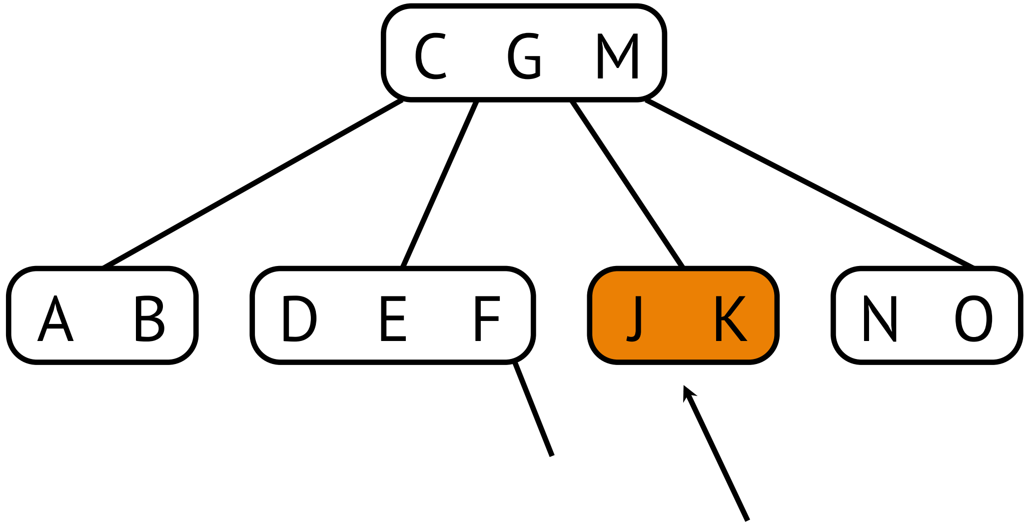
Merge()

- Lösche Element aus **aktuellem** Knoten



Rotate()

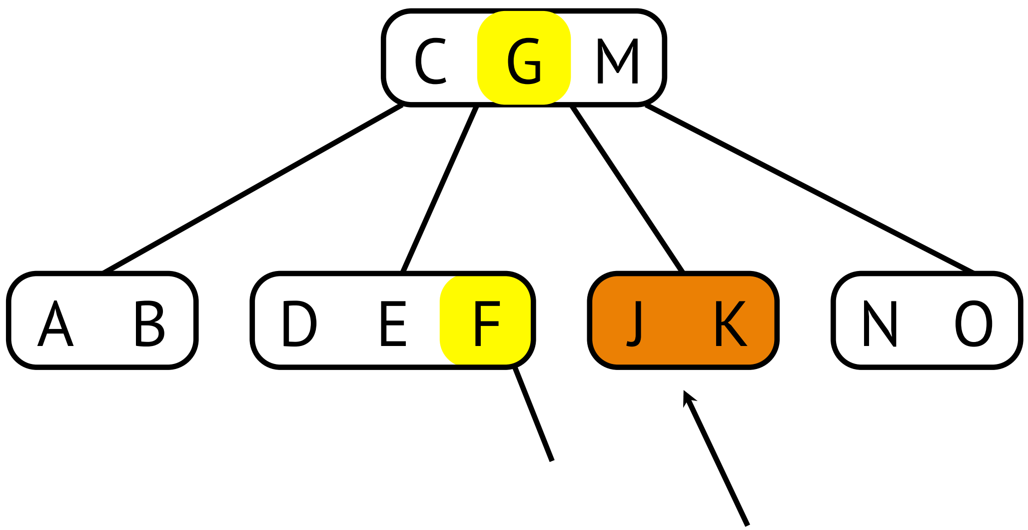
- Lösche Element aus **rechtem** Teilbaum



nur t−1 Knoten → rekursiver Abstieg nicht möglich

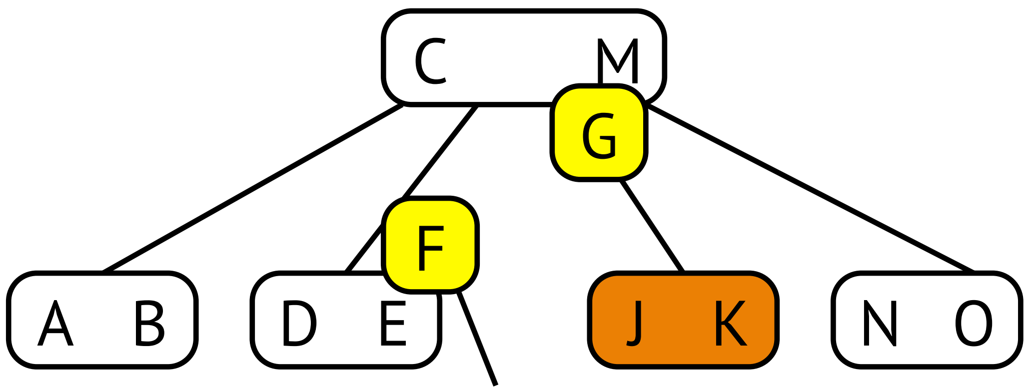
Rotate()

- Lösche Element aus **rechtem** Teilbaum

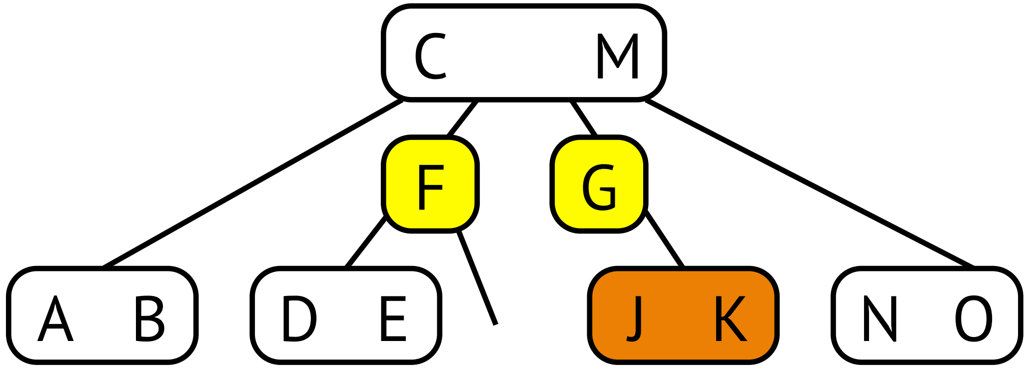


nur t−1 Knoten → rekursiver Abstieg nicht möglich

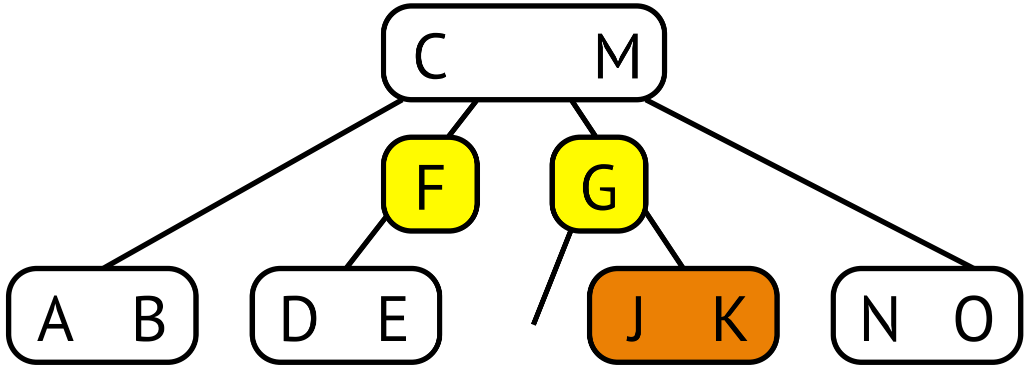
Rotate()



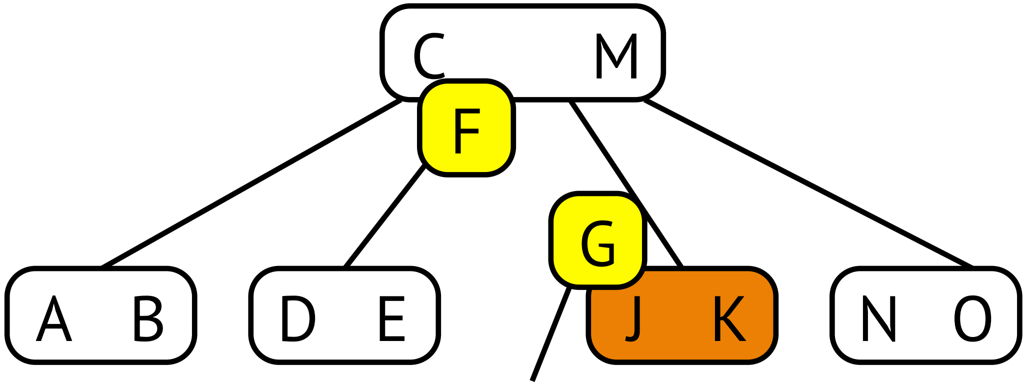
Rotate()



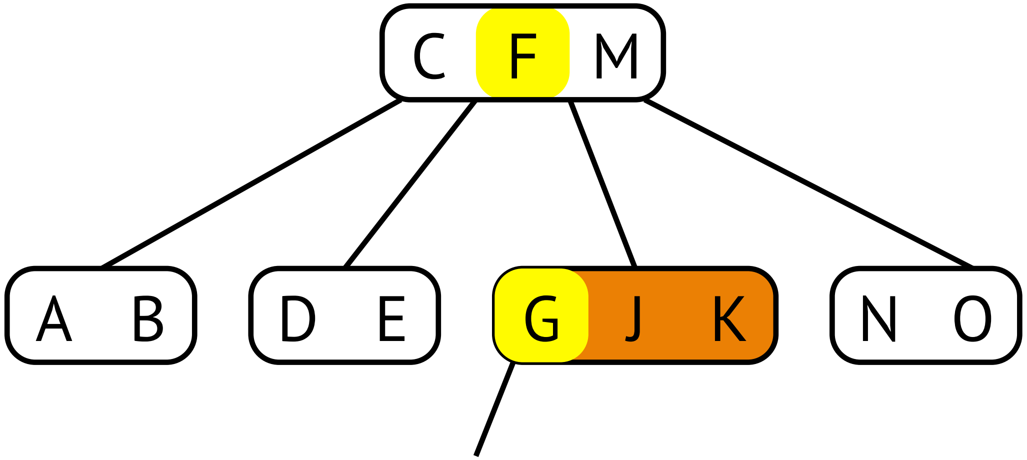
Rotate()



Rotate()



Rotate()

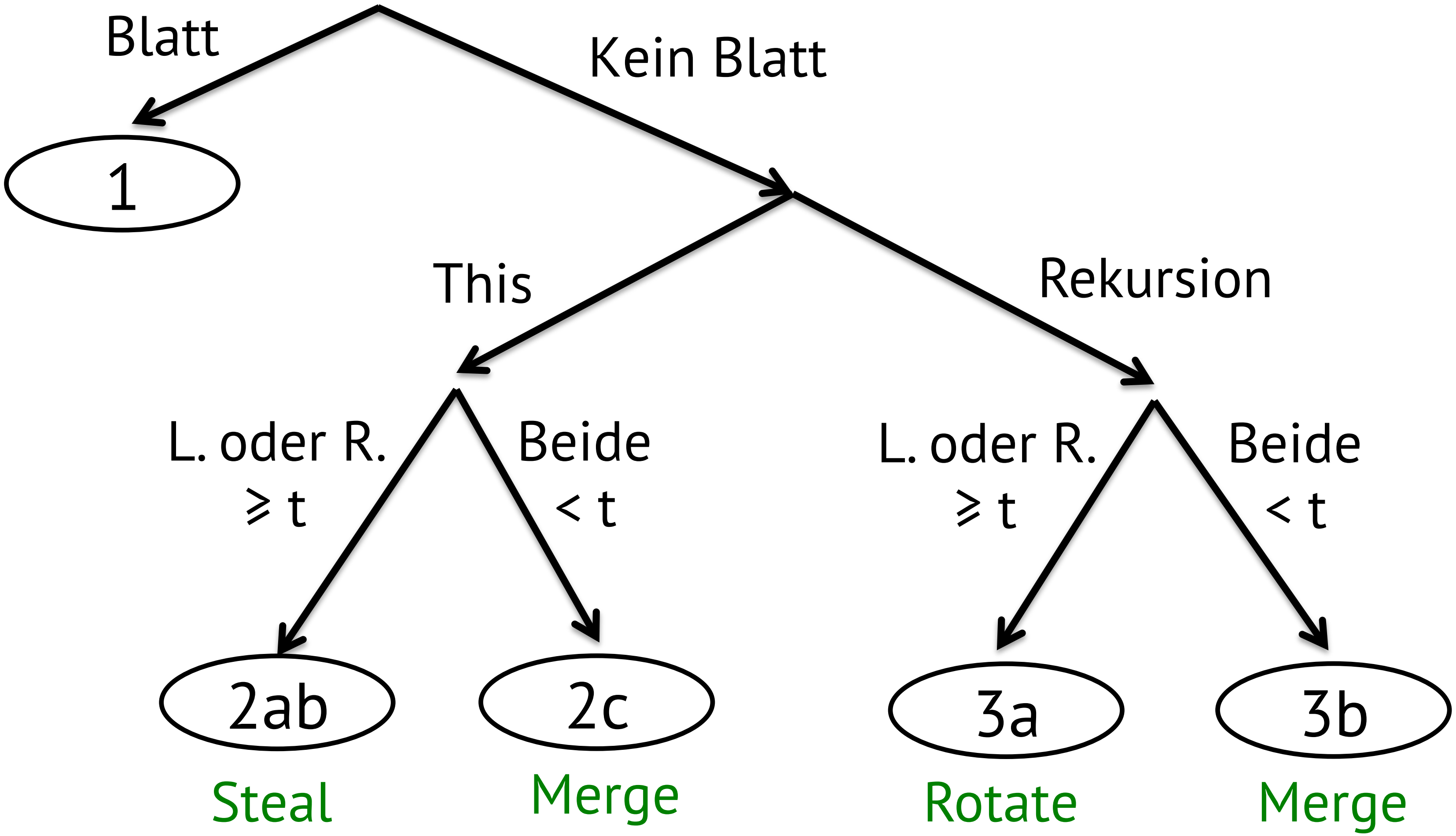


Delete()

- Standardfall: Schlüssel nicht im Knoten, Rekursion in den entsprechenden Teilbaum
- Nur in Knoten absteigen, die hinreichend gefüllt sind → wenn nicht, vorher den Baum mit Merge(), Steal(), Rotate() umstrukturieren.
- Nicht-Standardfälle: 1, 2a, 2b, 3a, 3b

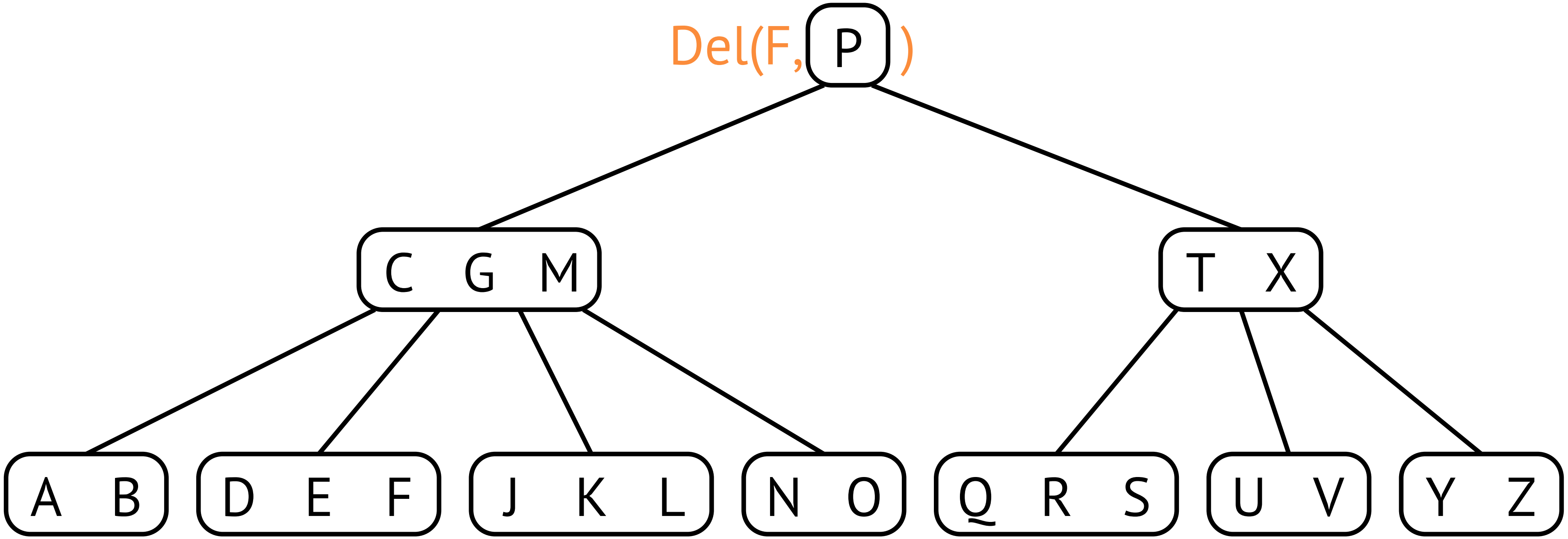
  (s. Cormen)

B-Bäume – Delete – Scheme

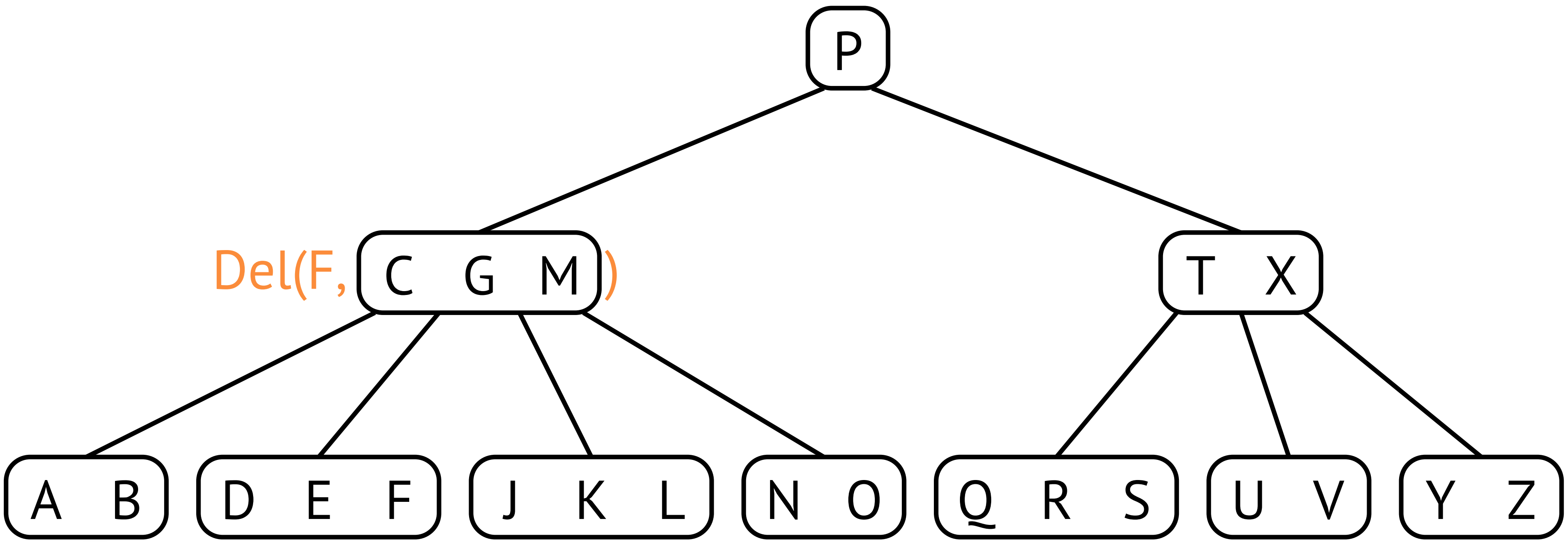


Delete()

- **Fall 1**: Schlüssel in einem Blatt

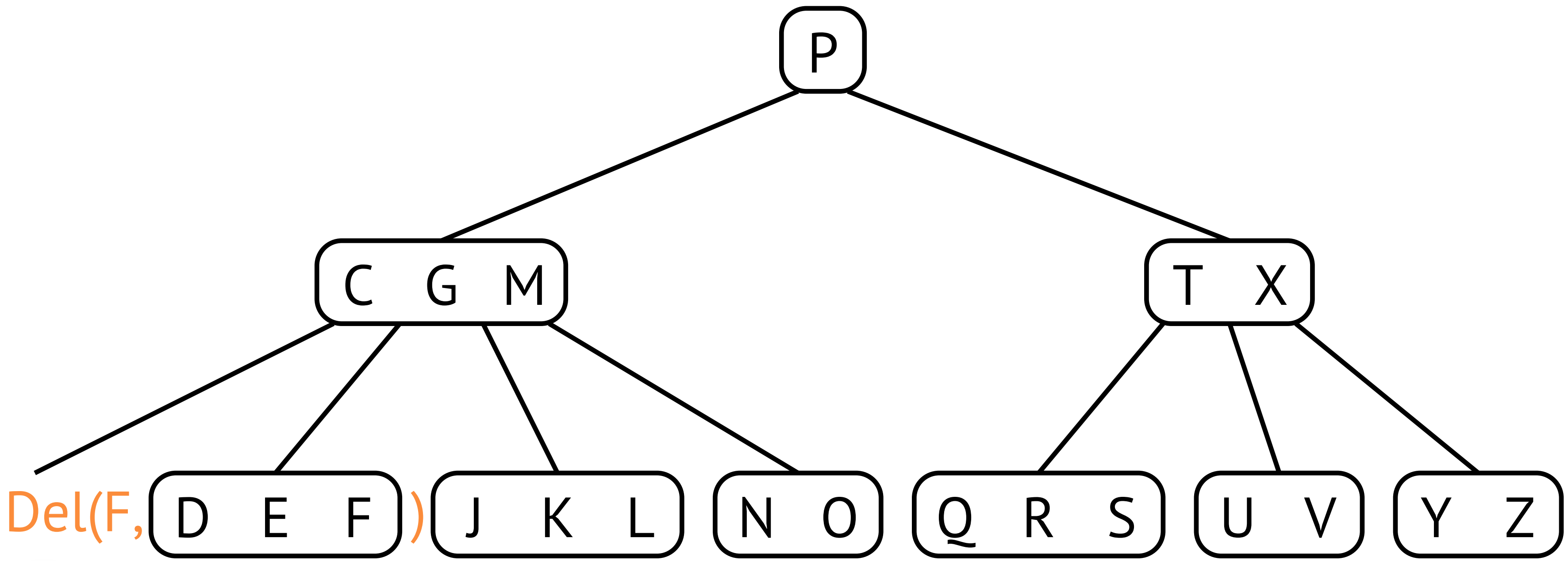


- **Fall 1**: Schlüssel in einem Blatt



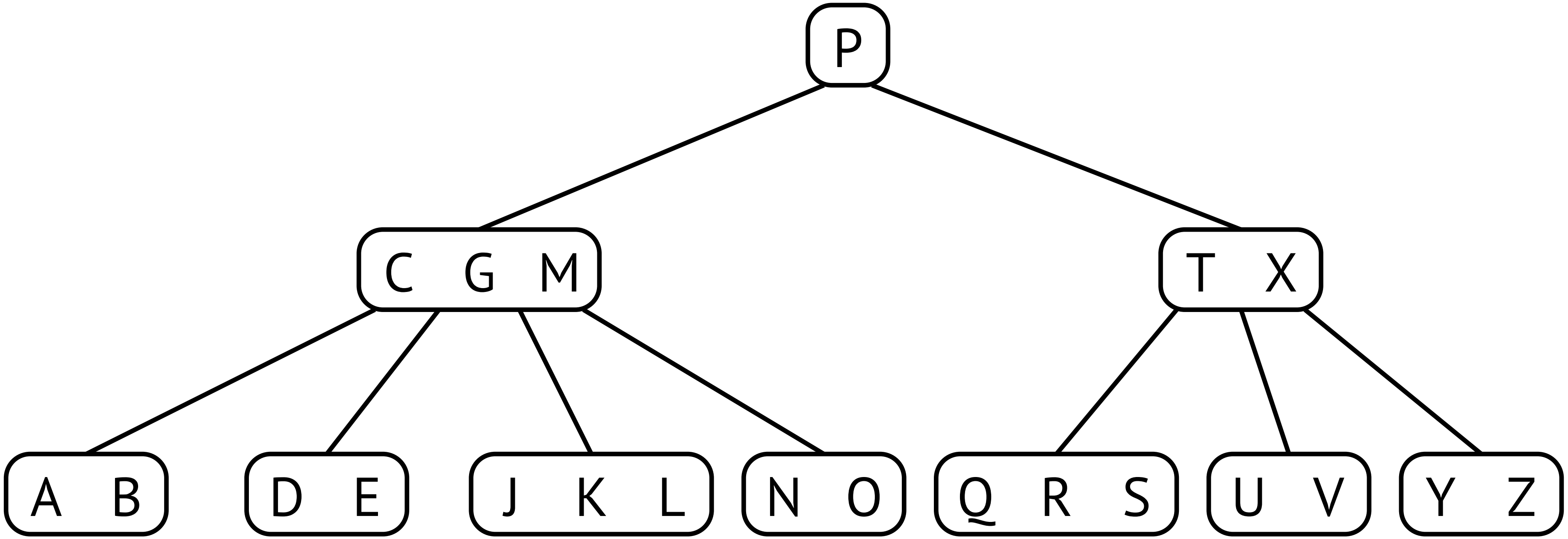
Delete()

- **Fall 1**: Schlüssel in einem Blatt



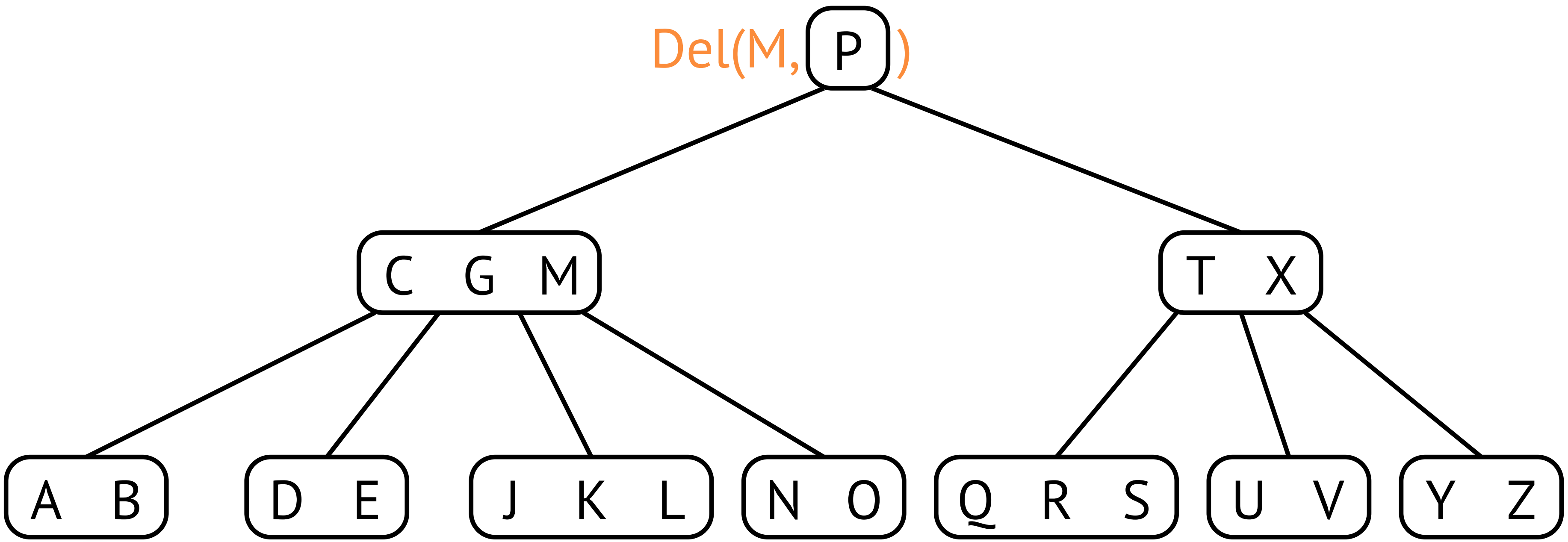
Delete()

- **Fall 1**: Schlüssel in einem Blatt
- Schlüssel wird gelöscht



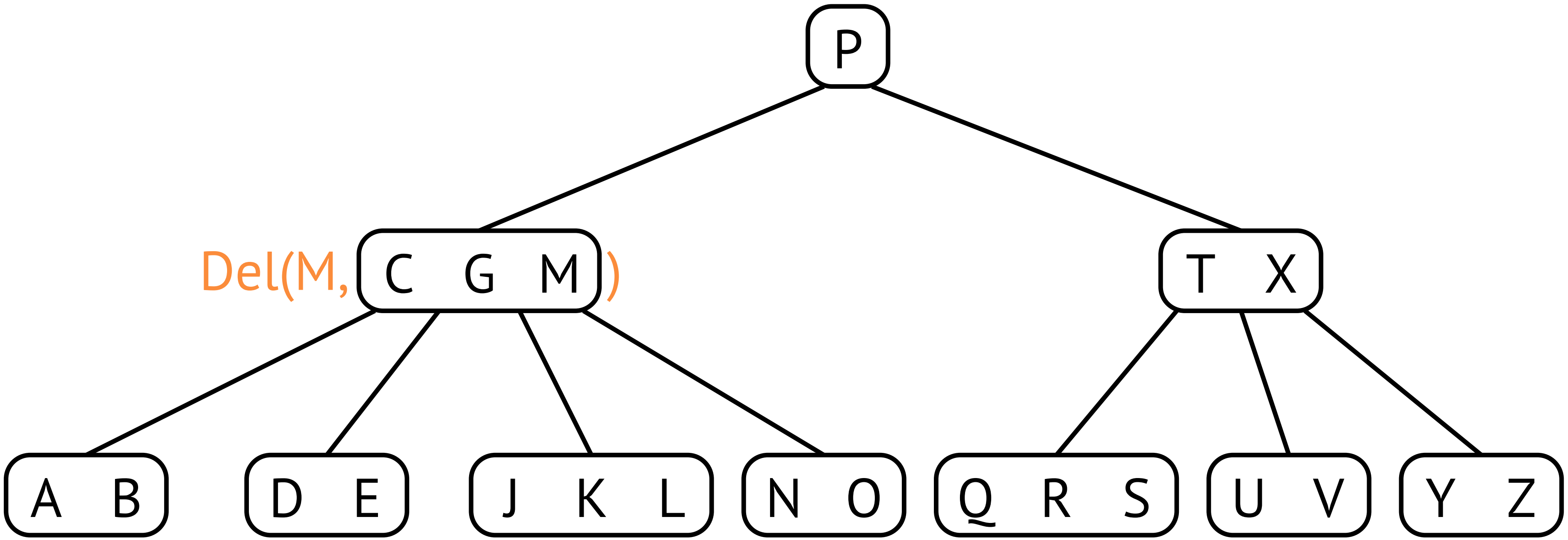
Delete()

- **Fall 2a**: Schlüssel in innerem Knoten, linker Sohn hat ⩾ t Schlüssel

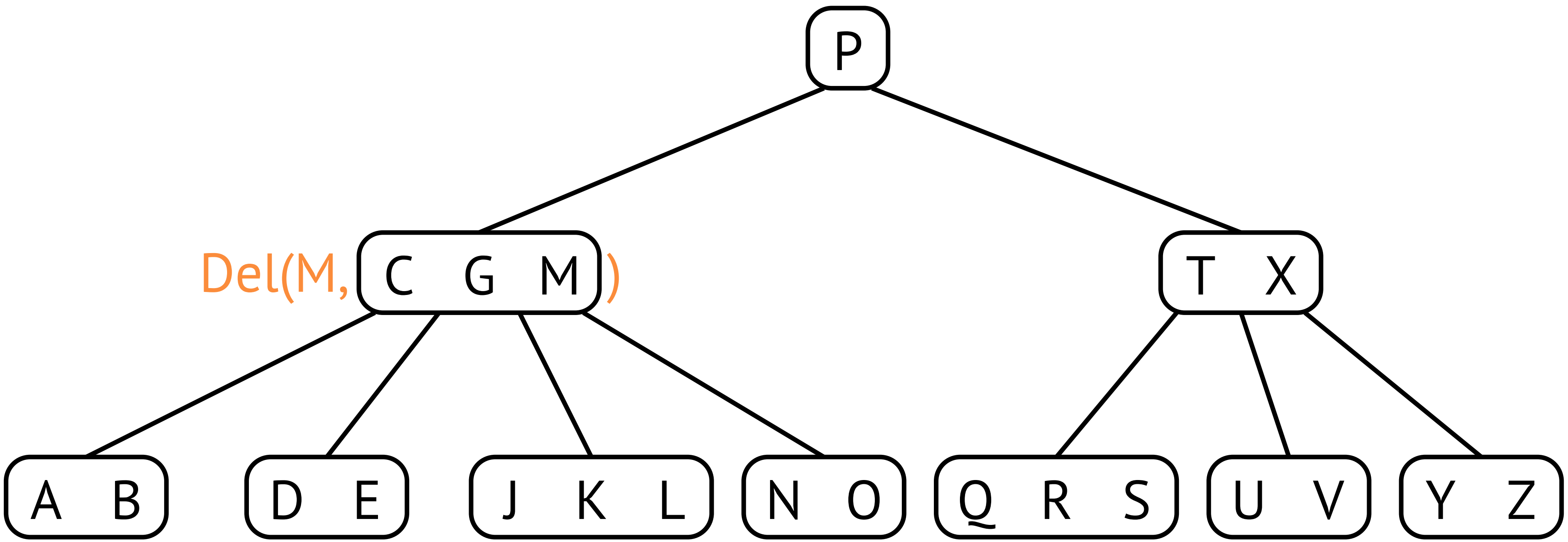


Delete()

- **Fall 2a**: Schlüssel in innerem Knoten, linker Sohn hat ≥ t Schlüssel

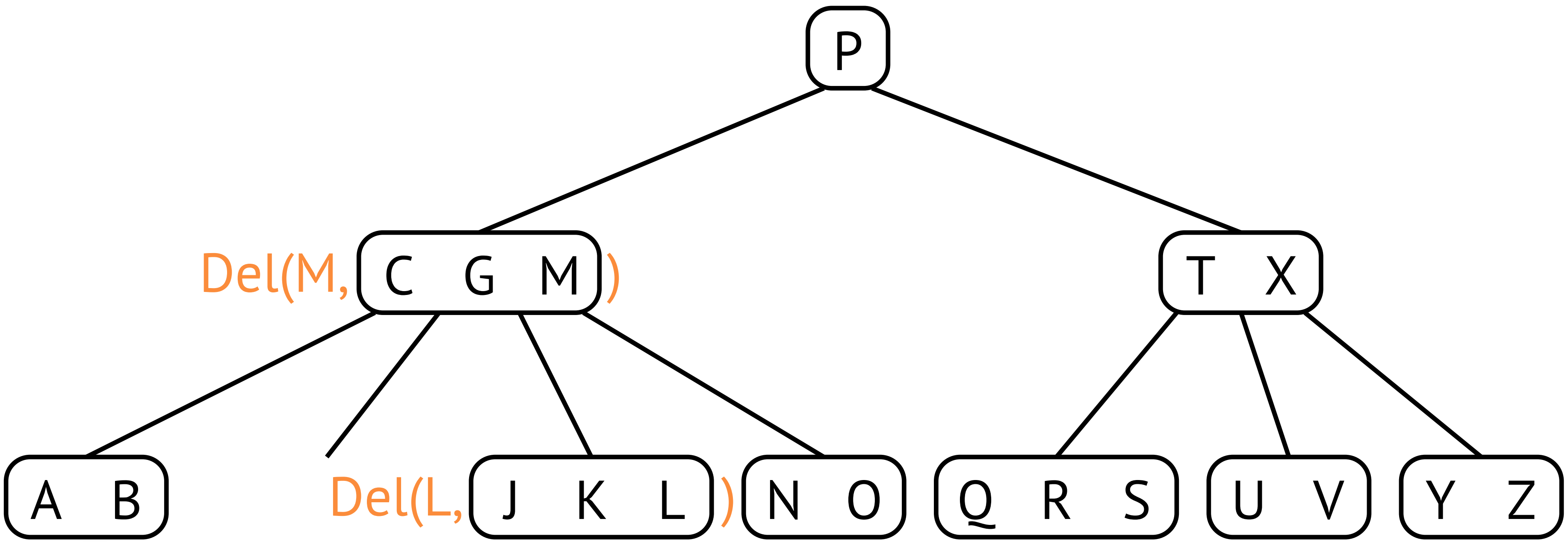


- **Fall 2a**: Schlüssel in innerem Knoten, linker Sohn hat ≥ t Schlüssel
- Stehle **Vorgänger** aus linkem Teilbaum



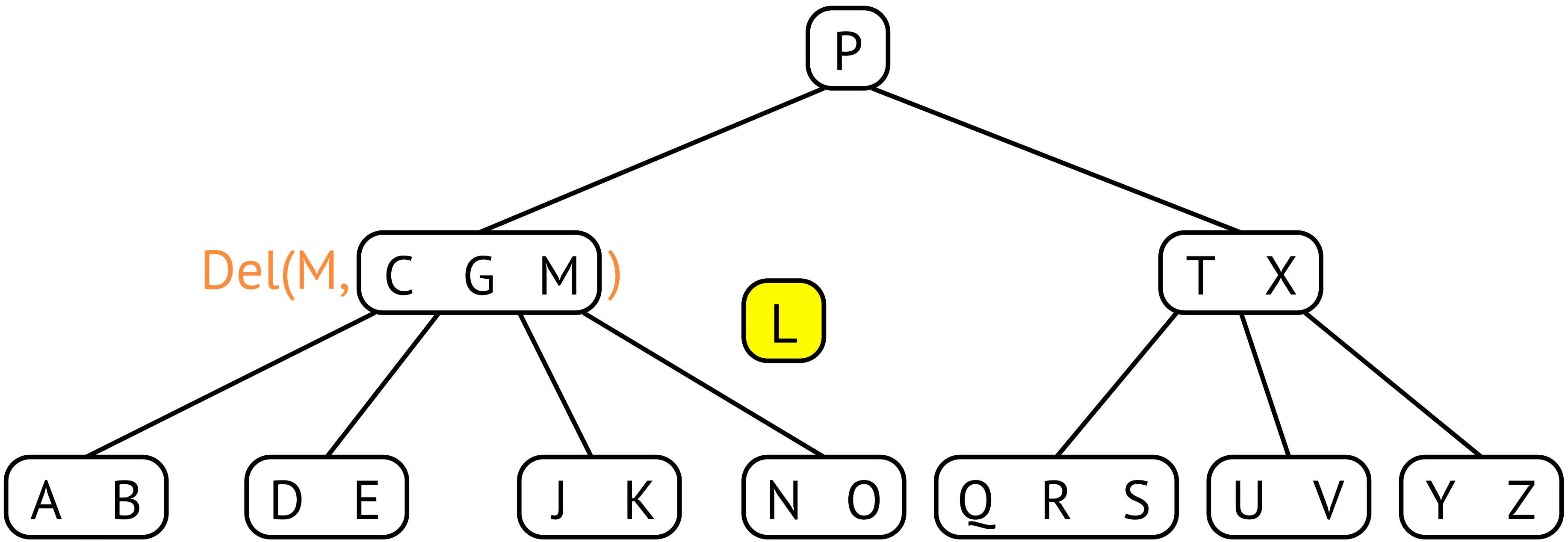
Delete()

- **Fall 2a**: Schlüssel in innerem Knoten, linker Sohn hat ⩾ t Schlüssel
- Stehle **Vorgänger** aus linkem Teilbaum



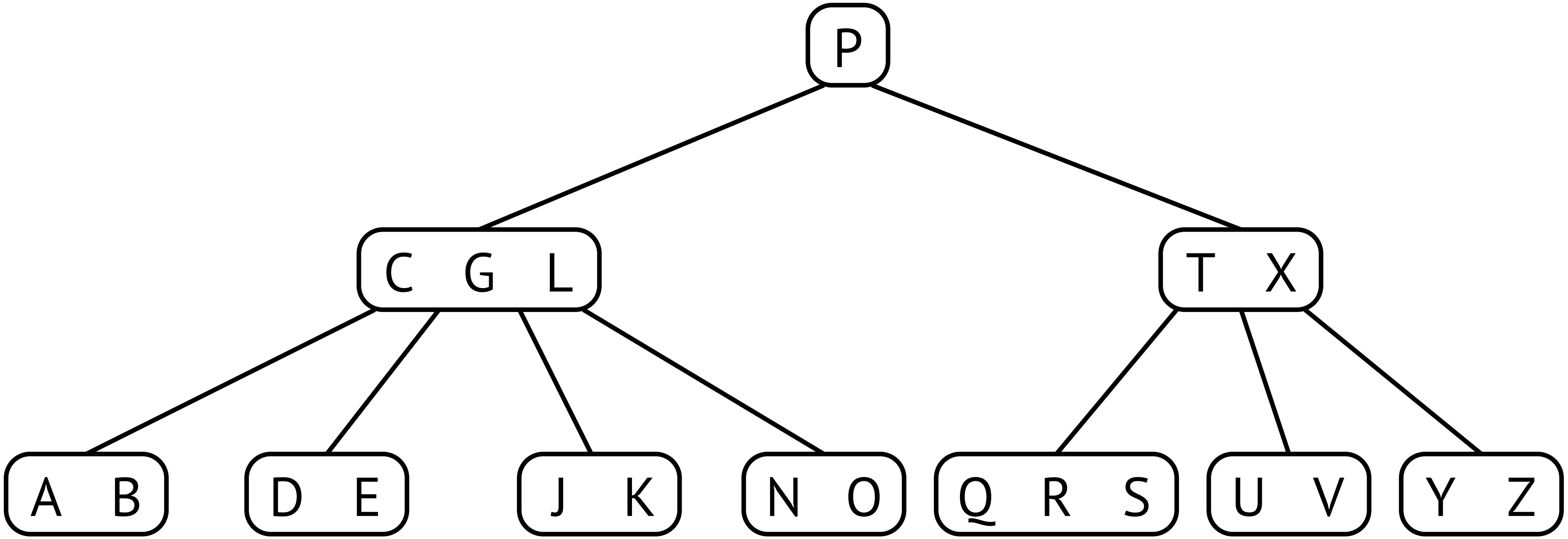
Delete()

- **Fall 2a**: Schlüssel in innerem Knoten, linker Sohn hat ≥ t Schlüssel
- Stehle **Vorgänger** aus linkem Teilbaum



Delete()

- **Fall 2a**: Schlüssel in innerem Knoten, linker Sohn hat $\geq t$ Schlüssel
- Stehle **Vorgänger** aus linkem Teilbaum



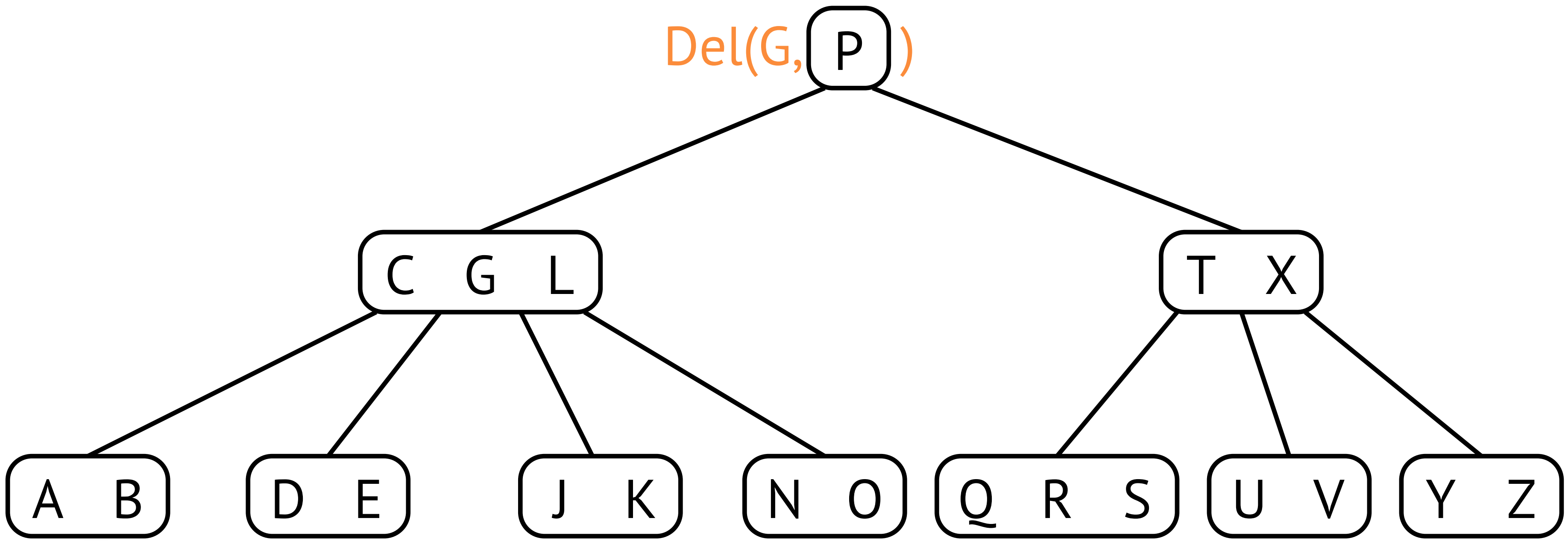
Delete()

- **Fall 2b**: Schlüssel in innerem Knoten, rechter Sohn hat $\geqslant t$ Schlüssel

  → analog zu **Fall 2a**

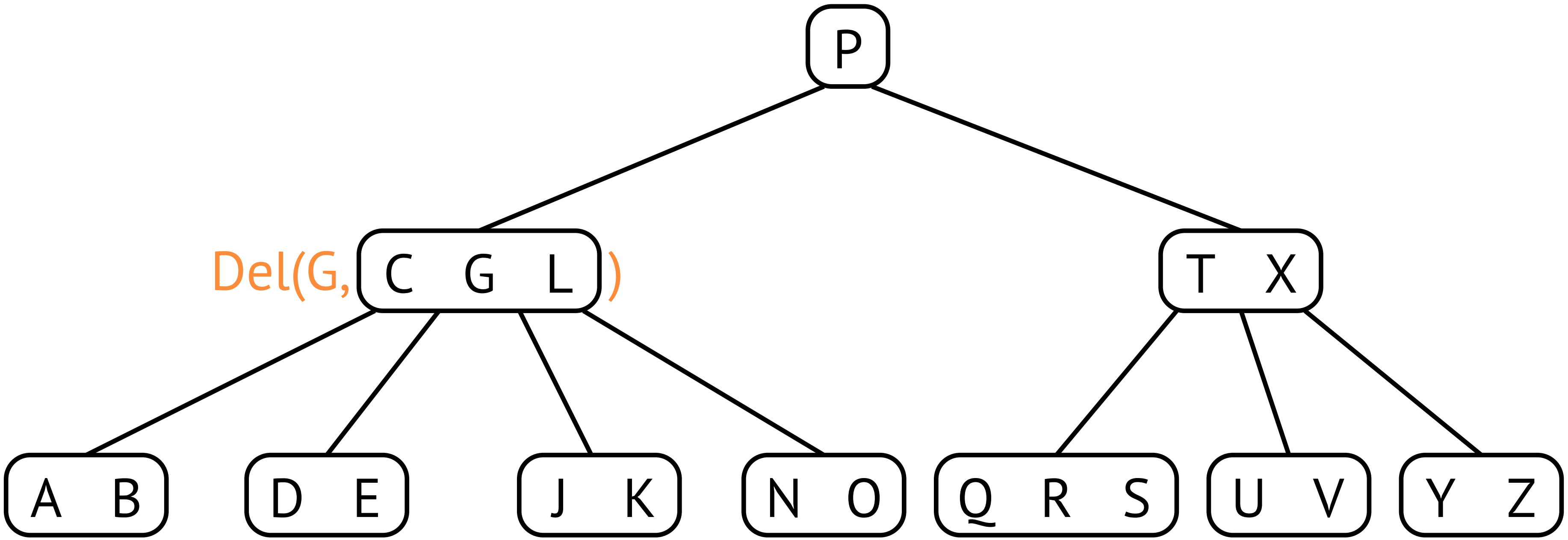
Delete()

- **Fall 2c:** Schlüssel in innerem Knoten, beide Söhne haben t−1 Schlüssel



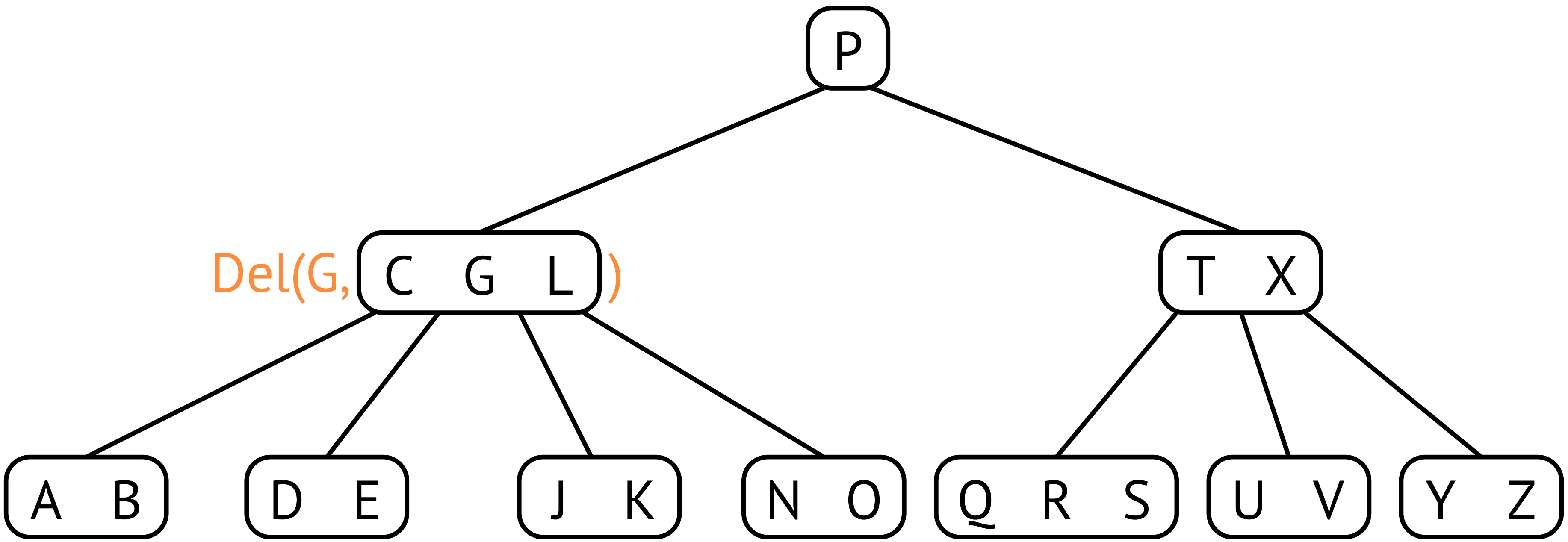
Delete()

- **Fall 2c**: Schlüssel in innerem Knoten, beide Söhne haben t−1 Schlüssel



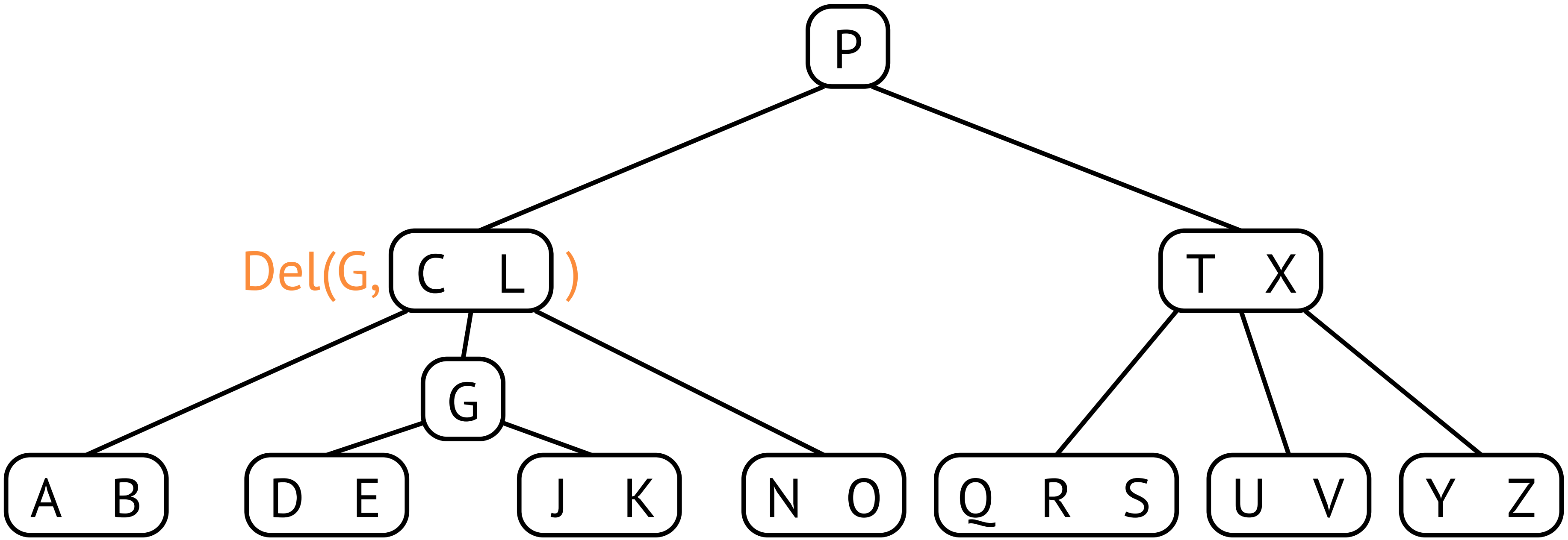
Delete()

- **Fall 2c**: Schlüssel in innerem Knoten, beide Söhne haben t−1 Schlüssel
- Stehlen nicht möglich → Merge()



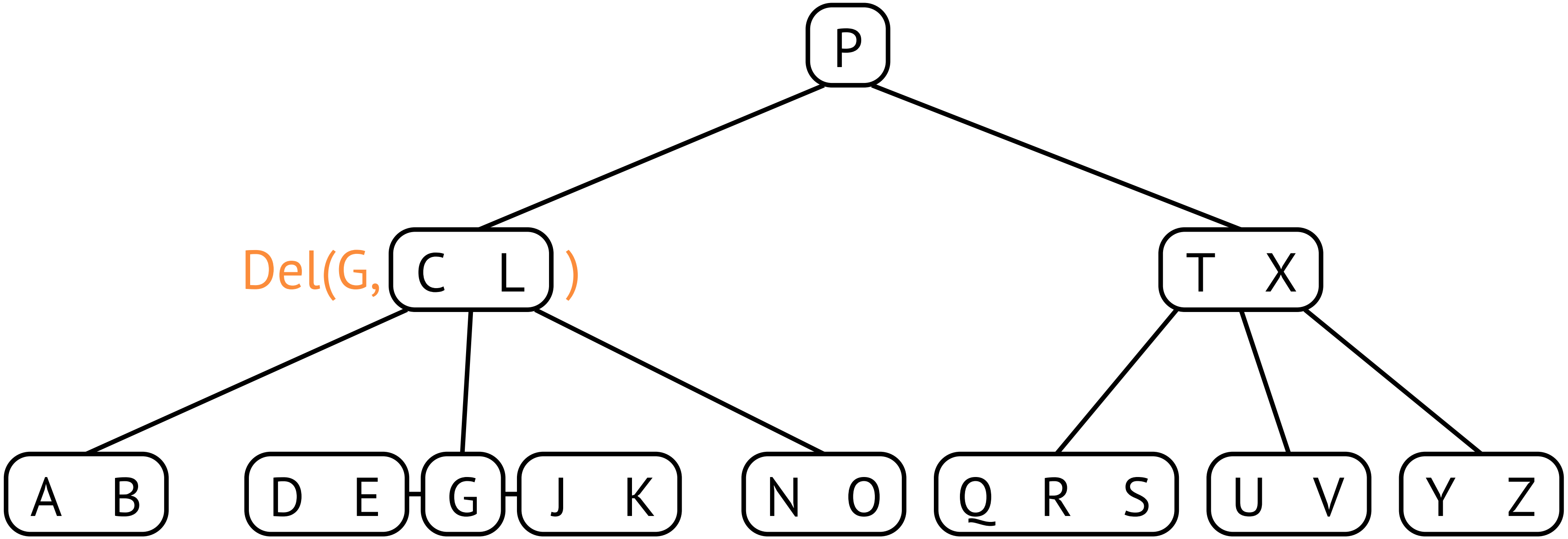
Delete()

- **Fall 2c**: Schlüssel in innerem Knoten, beide Söhne haben t−1 Schlüssel
- Stehlen nicht möglich → Merge()



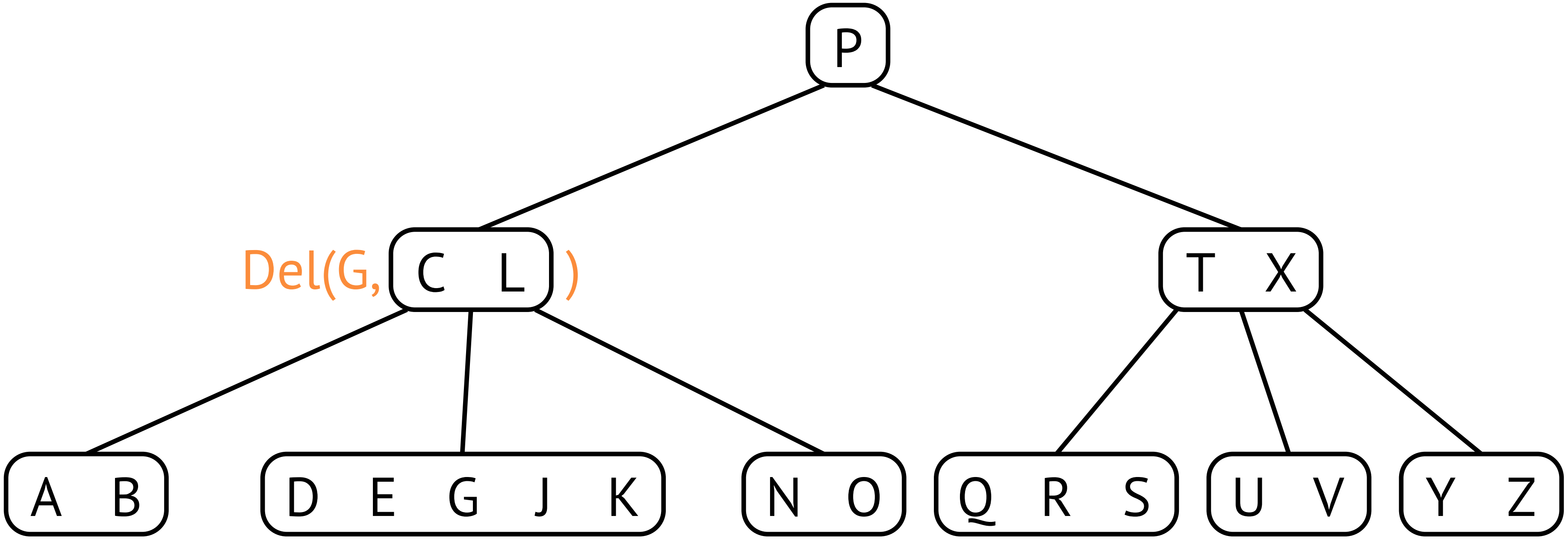
Delete()

- **Fall 2c**: Schlüssel in innerem Knoten, beide Söhne haben t−1 Schlüssel
- Stehlen nicht möglich → Merge()



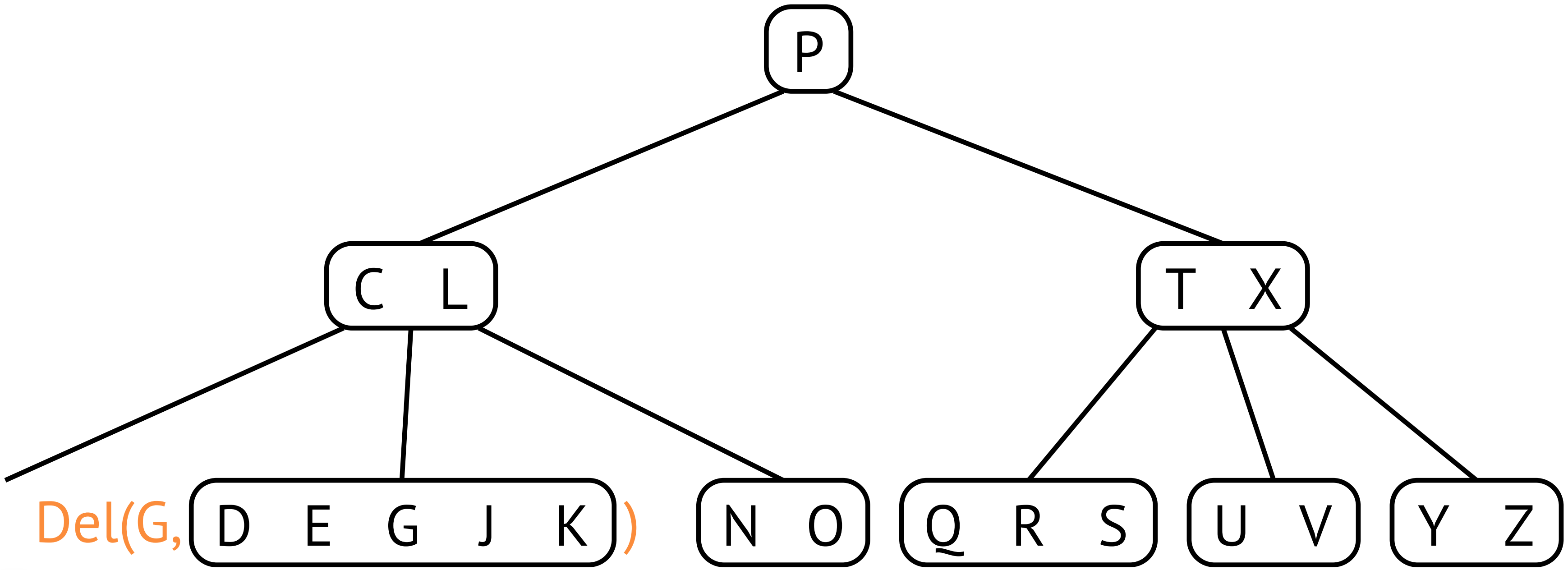
Delete()

- **Fall 2c**: Schlüssel in innerem Knoten, beide Söhne haben t–1 Schlüssel
- Stehlen nicht möglich → Merge()



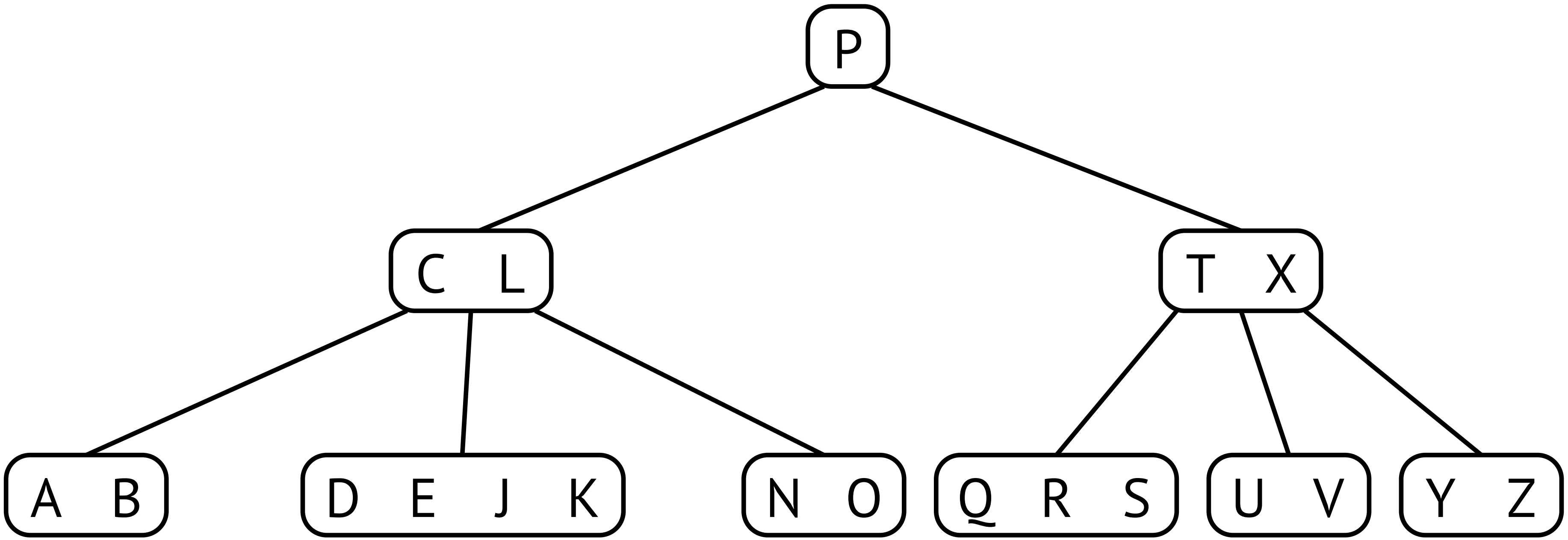
Delete()

- **Fall 2c**: Schlüssel in innerem Knoten, beide Söhne haben t−1 Schlüssel
- Stehlen nicht möglich → Merge()
- Rekursion



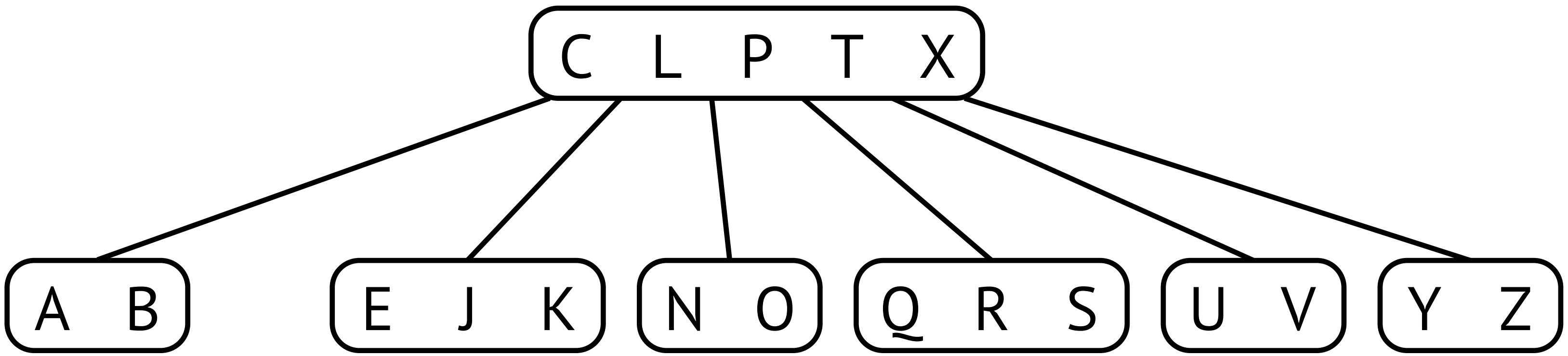
Delete()

- **Fall 2c**: Schlüssel in innerem Knoten, beide Söhne haben t−1 Schlüssel
- Stehlen nicht möglich → Merge()
- Rekursion



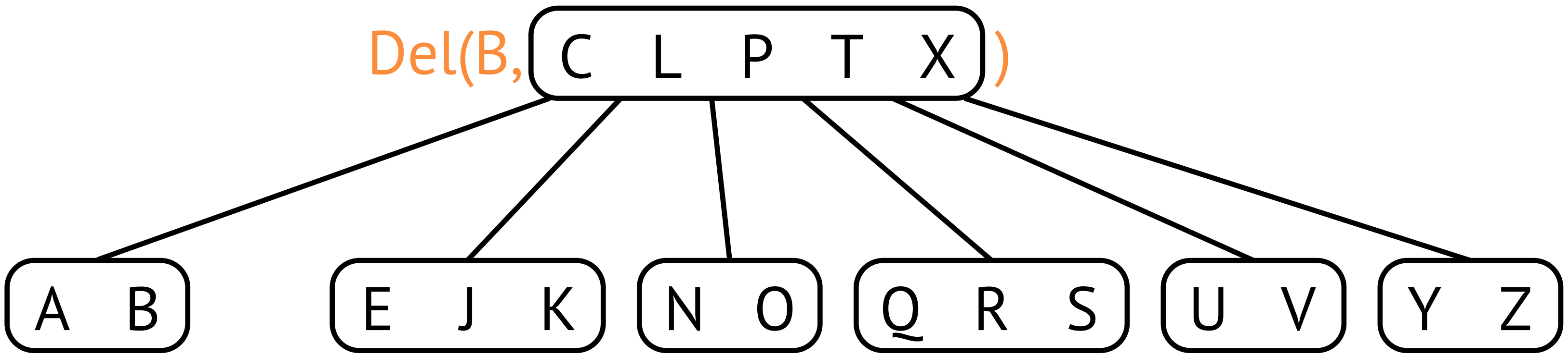
Delete()

- **Fall 3a**: Schlüssel nicht im aktuellen Knoten, der entspr. Unterbaum hat nur t−1 Schlüssel, sein Bruder aber t Schlüssel



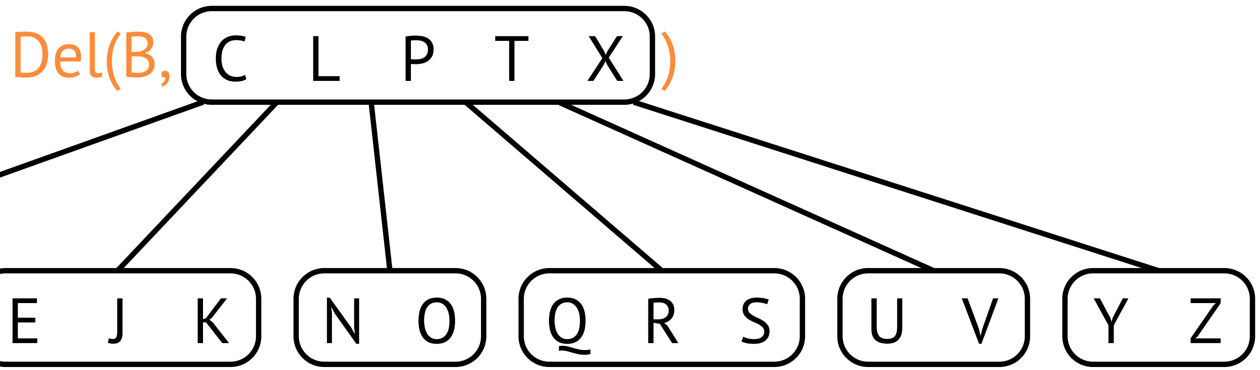
Delete()

- **Fall 3a**: Schlüssel nicht im aktuellen Knoten, der entspr. Unterbaum hat nur t−1 Schlüssel, sein Bruder aber t Schlüssel



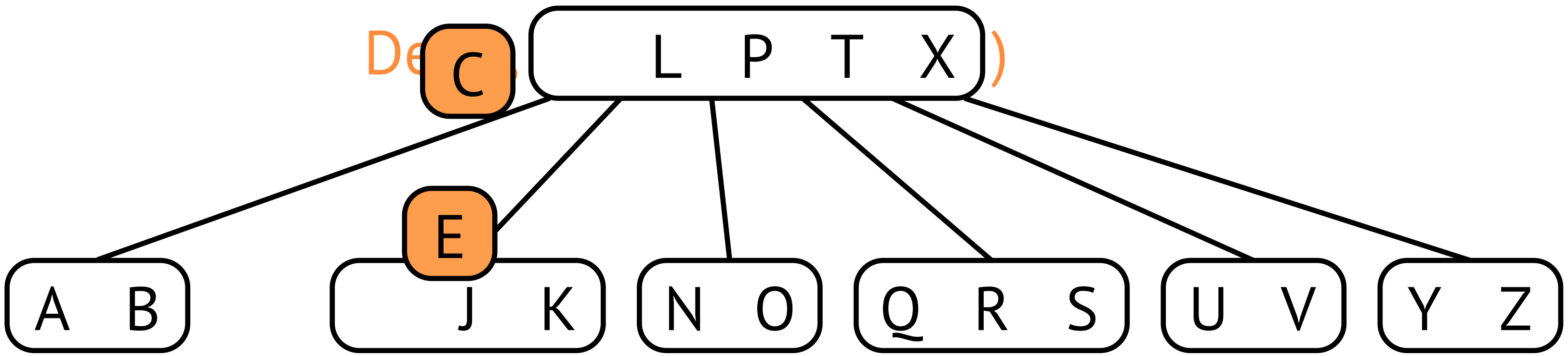
Delete()

- **Fall 3a**: Schlüssel nicht im aktuellen Knoten, der entspr. Unterbaum hat nur t−1 Schlüssel, sein Bruder aber t Schlüssel
- Rotate()



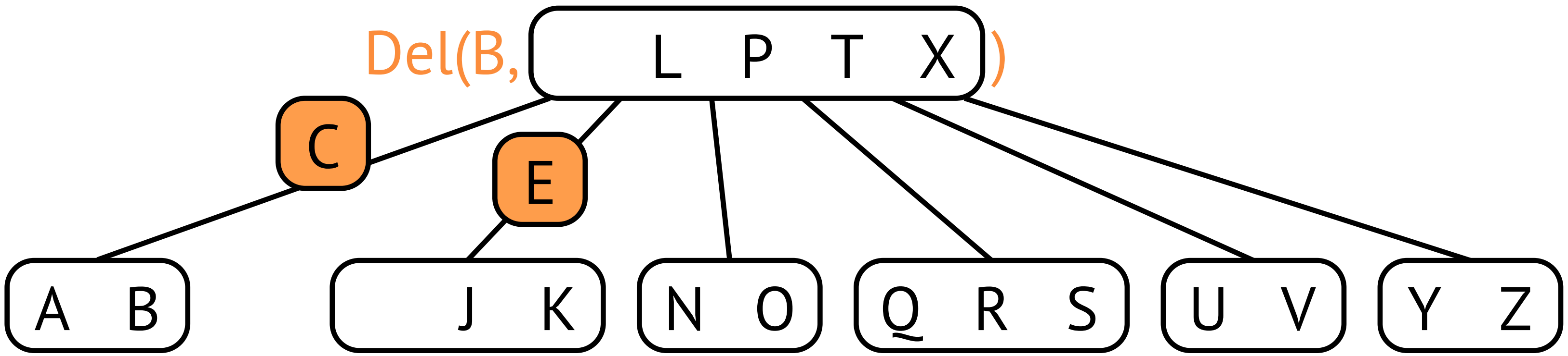
Delete()

- **Fall 3a**: Schlüssel nicht im aktuellen Knoten, der entspr. Unterbaum hat nur t−1 Schlüssel, sein Bruder aber t Schlüssel
- Rotate()



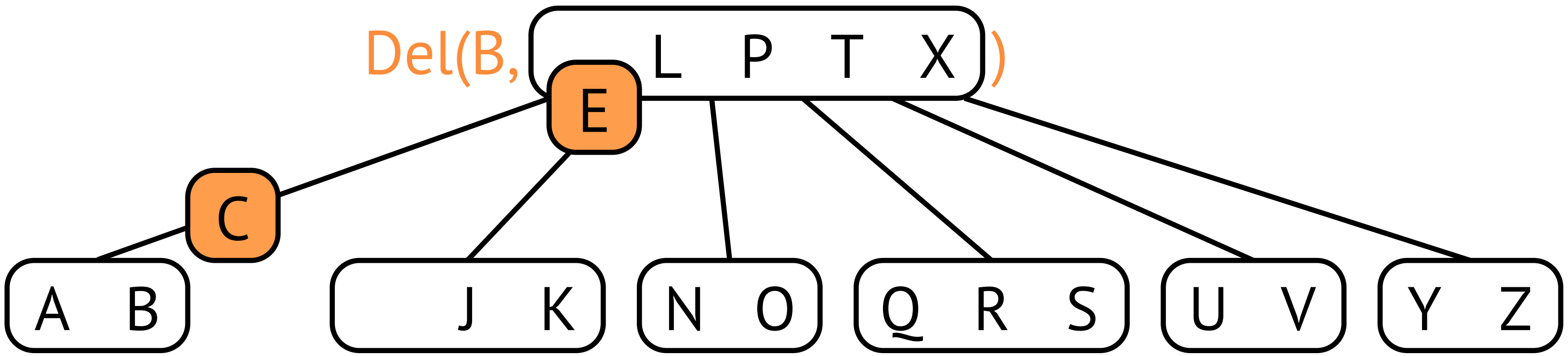
Delete()

- **Fall 3a**: Schlüssel nicht im aktuellen Knoten, der entspr. Unterbaum hat nur t−1 Schlüssel, sein Bruder aber t Schlüssel
- Rotate()



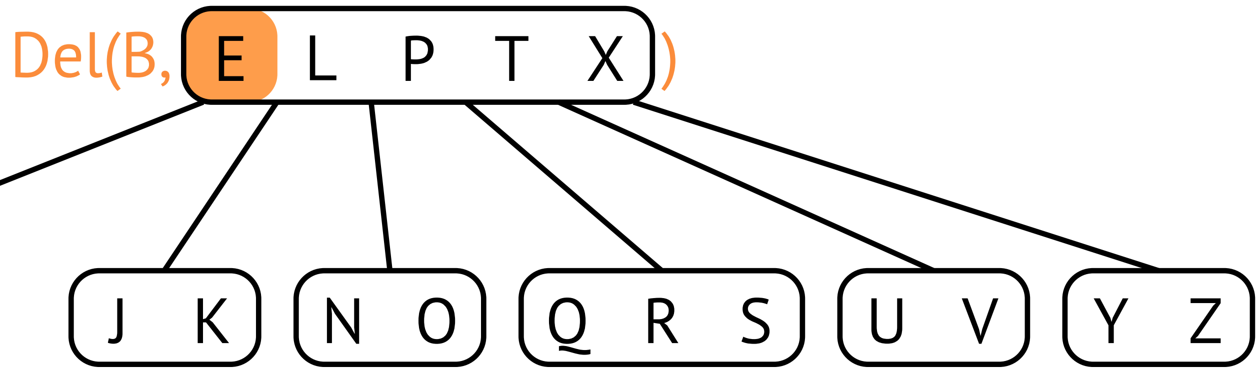
Delete()

- **Fall 3a**: Schlüssel nicht im aktuellen Knoten, der entspr. Unterbaum hat nur t−1 Schlüssel, sein Bruder aber t Schlüssel
- Rotate()



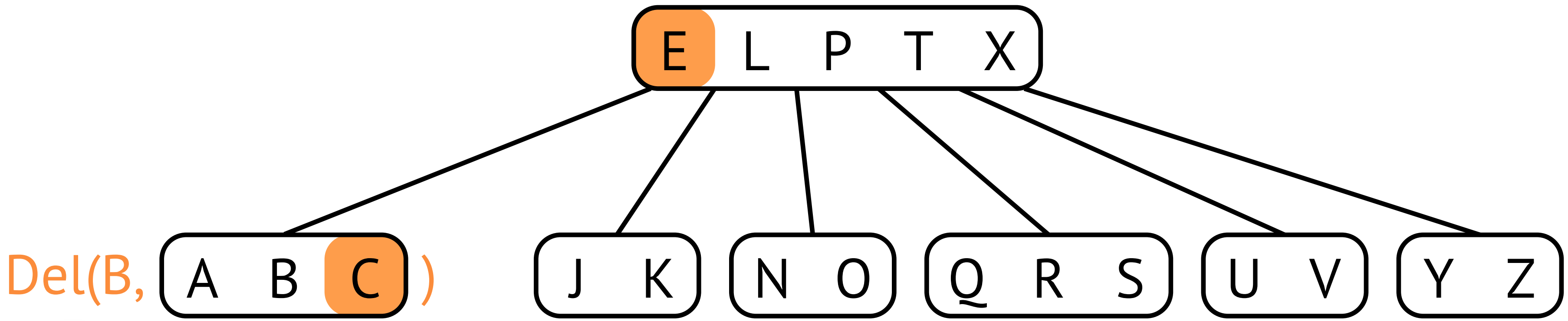
Delete()

- **Fall 3a**: Schlüssel nicht im aktuellen Knoten, der entspr. Unterbaum hat nur t−1 Schlüssel, sein Bruder aber t Schlüssel
- Rotate()



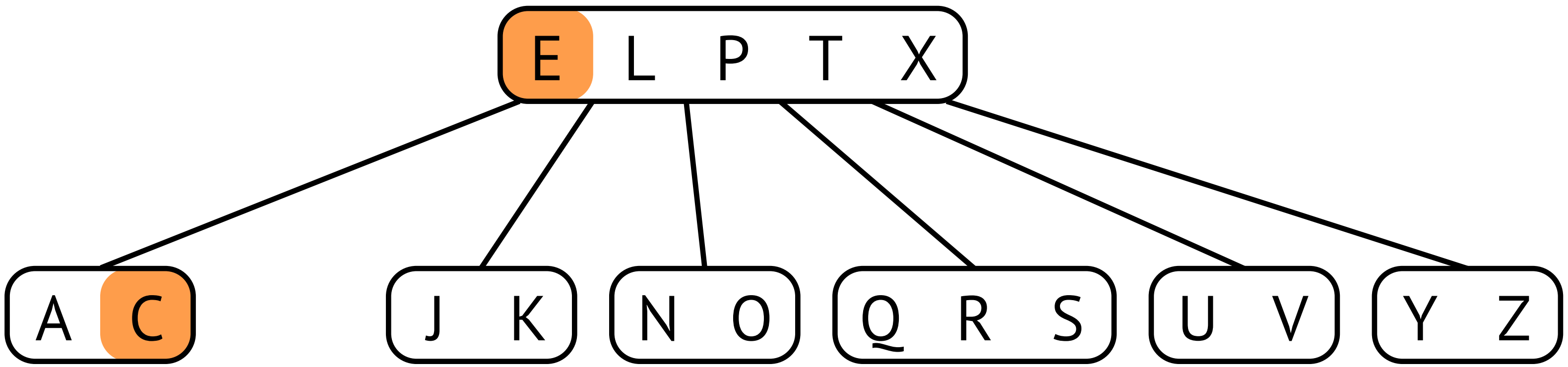
Delete()

- **Fall 3a**: Schlüssel nicht im aktuellen Knoten, der entspr. Unterbaum hat nur t−1 Schlüssel, sein Bruder aber t Schlüssel
- Rotate(), Rekursion



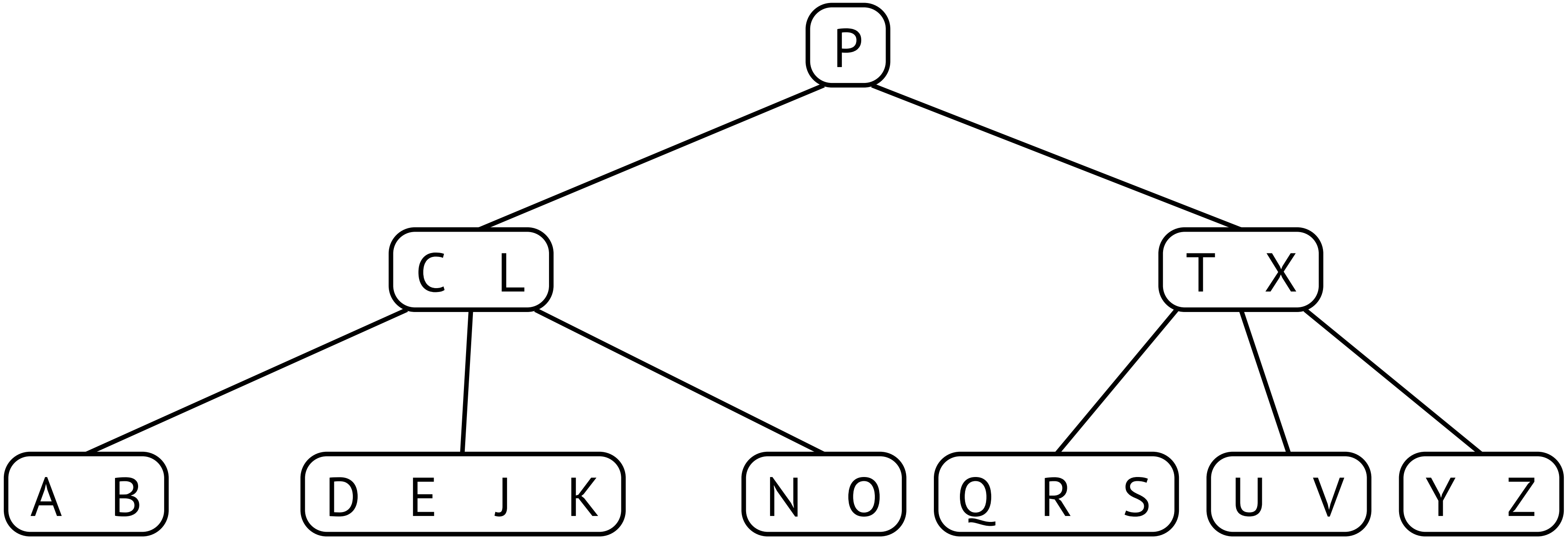
Delete()

- **Fall 3a**: Schlüssel nicht im aktuellen Knoten, der entspr. Unterbaum hat nur t−1 Schlüssel, sein Bruder aber t Schlüssel
- Rotate(), Rekursion



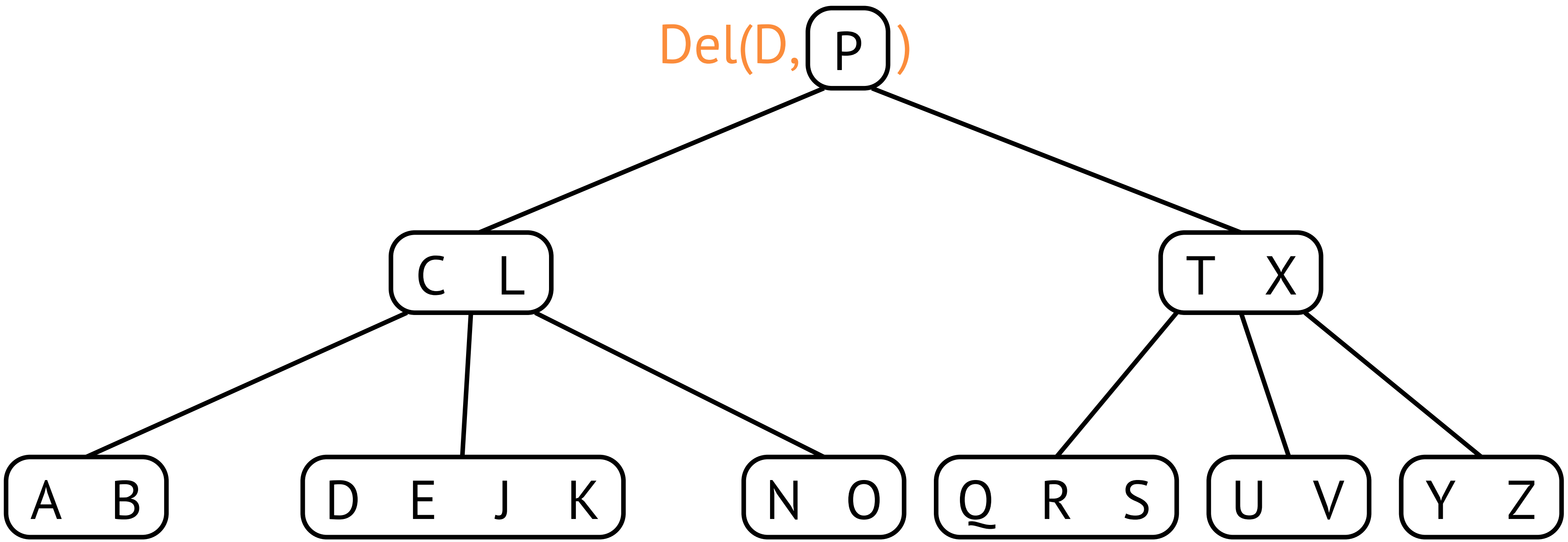
Delete()

- **Fall 3b**: Schlüssel nicht im aktuellen Knoten, sowohl der entspr. Unterbaum als auch dessen Brüder haben t−1 Schlüssel



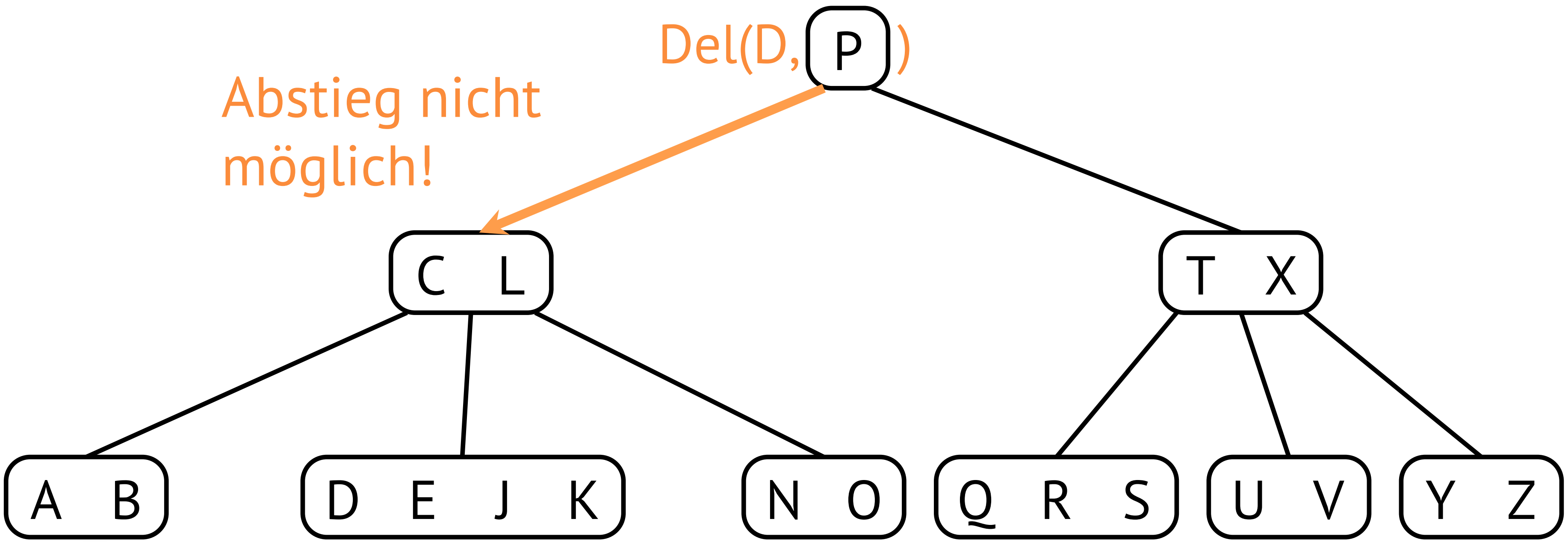
Delete()

- **Fall 3b**: Schlüssel nicht im aktuellen Knoten, sowohl der entspr. Unterbaum als auch dessen Brüder haben t−1 Schlüssel



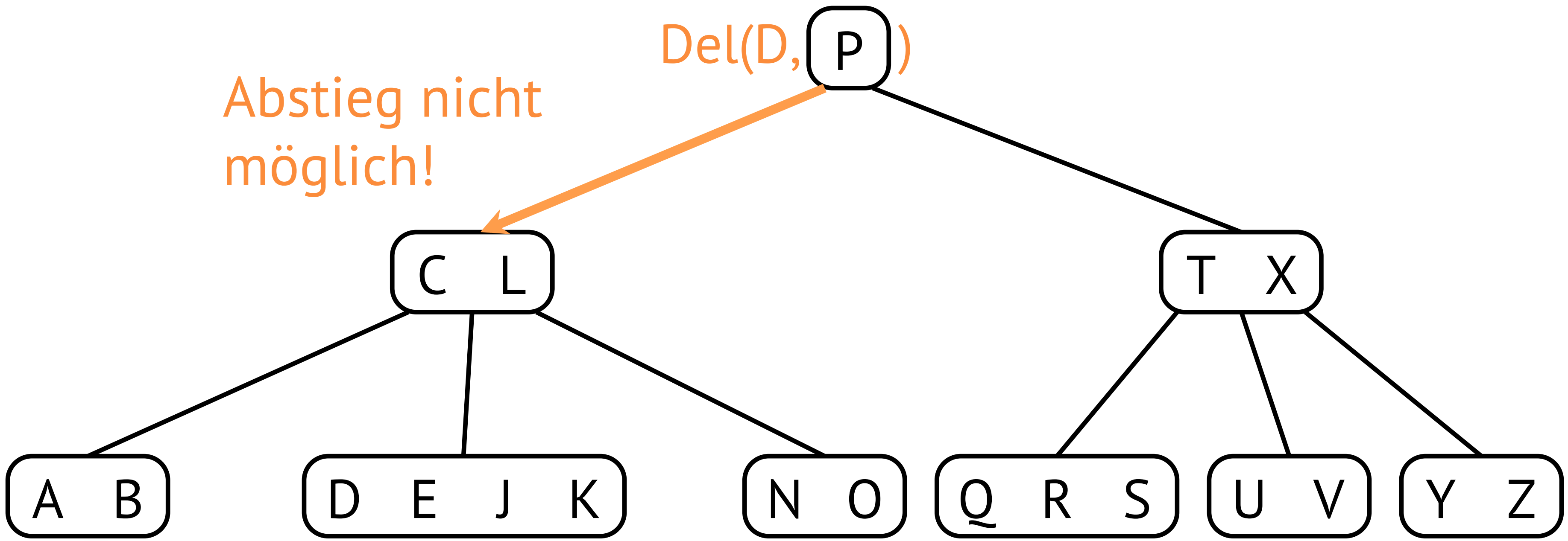
Delete()

- **Fall 3b**: Schlüssel nicht im aktuellen Knoten, sowohl der entspr. Unterbaum als auch dessen Brüder haben t−1 Schlüssel



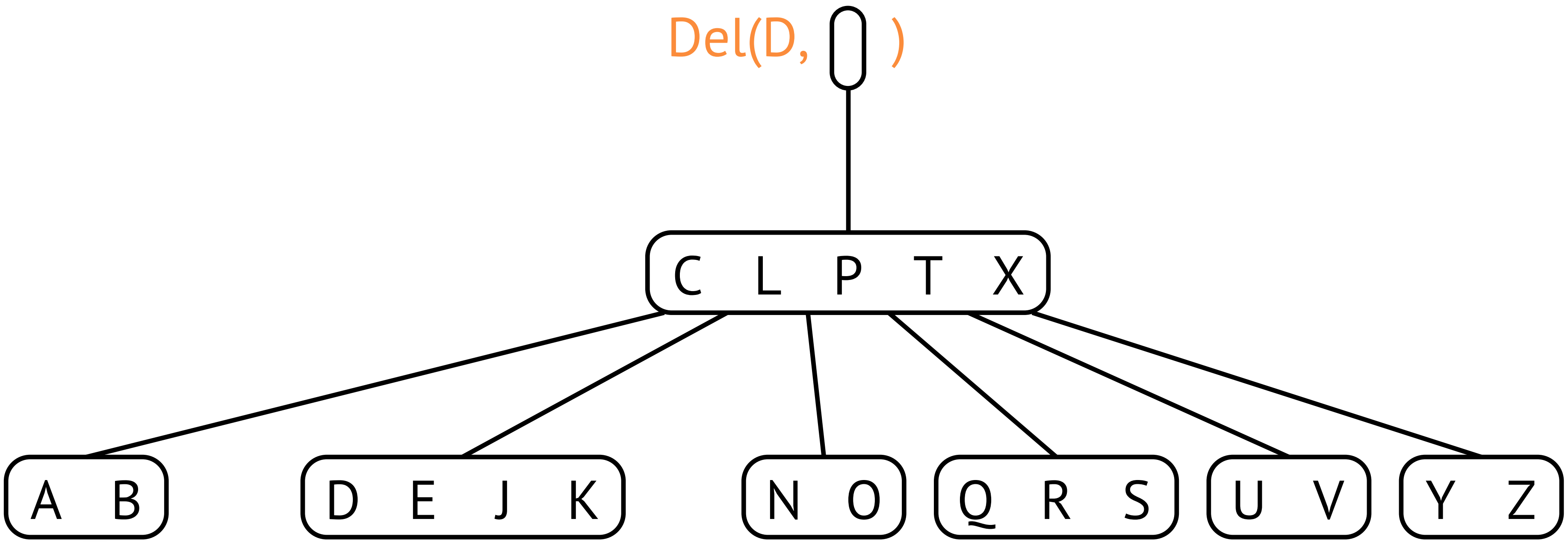
Delete()

- **Fall 3b**: Schlüssel nicht im aktuellen Knoten, sowohl der entspr. Unterbaum als auch dessen Brüder haben t−1 Schlüssel
- Merge zwei der Unterbäume



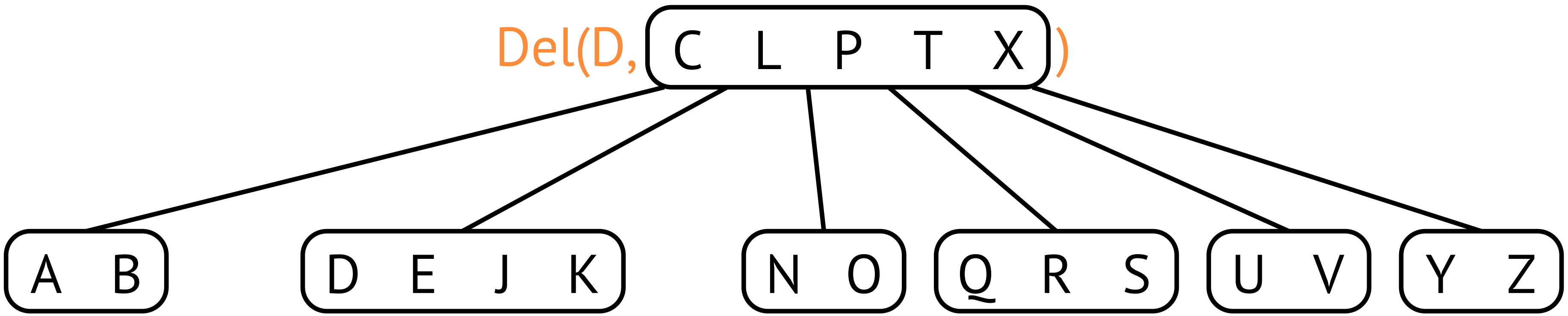
Delete()

- **Fall 3b**: Schlüssel nicht im aktuellen Knoten, sowohl der entspr. Unterbaum als auch dessen Brüder haben t−1 Schlüssel
- Merge zwei der Unterbäume



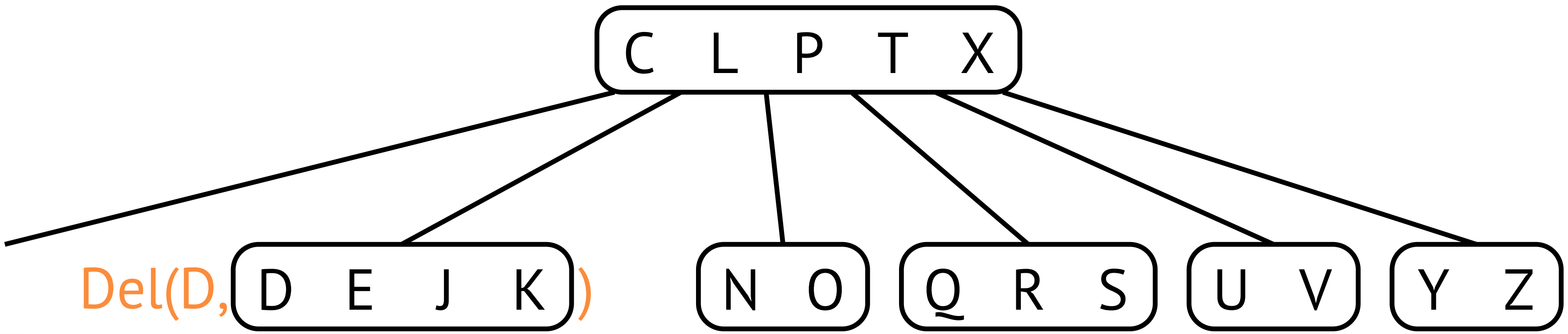
Delete()

- **Fall 3b**: Schlüssel nicht im aktuellen Knoten, sowohl der entspr. Unterbaum als auch dessen Brüder haben t−1 Schlüssel
- Merge zwei der Unterbäume



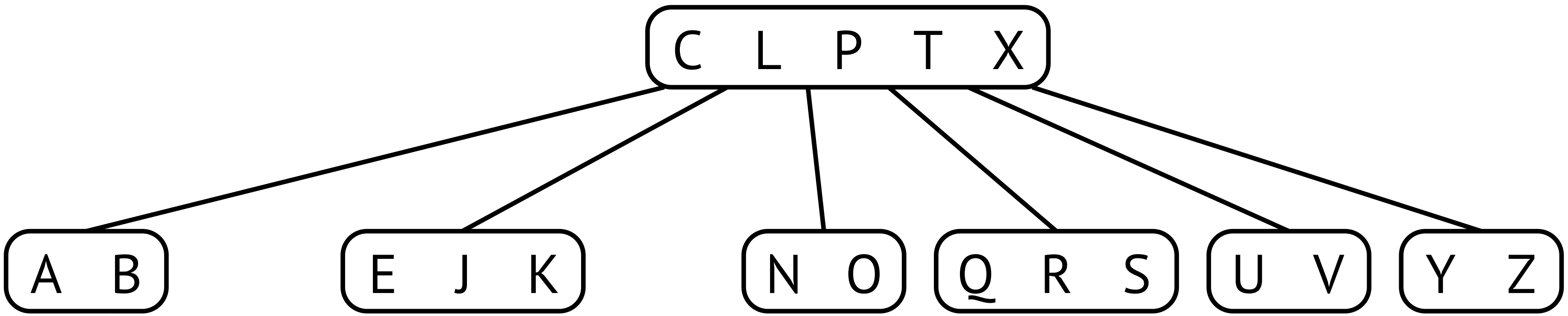
Delete()

- **Fall 3b**: Schlüssel nicht im aktuellen Knoten, sowohl der entspr. Unterbaum als auch dessen Brüder haben t−1 Schlüssel
- Merge zwei der Unterbäume



Delete()

- **Fall 3b**: Schlüssel nicht im aktuellen Knoten, sowohl der entspr. Unterbaum als auch dessen Brüder haben t−1 Schlüssel
- Merge zwei der Unterbäume



Zusammenfassung

- B-Bäume
  - Optimierter Zugriff auf externen Speicher durch Blockung mehrerer Schlüssel
  - Minimaler/maximaler Füllgrad
  - Garantierte Balancierung
  - Hysterese bei der Umstrukturierung
  - Steal(), Merge(), Rotate()