Algorithmen und Datenstrukturen
Kapitel 6A: Binäre Suchbäume

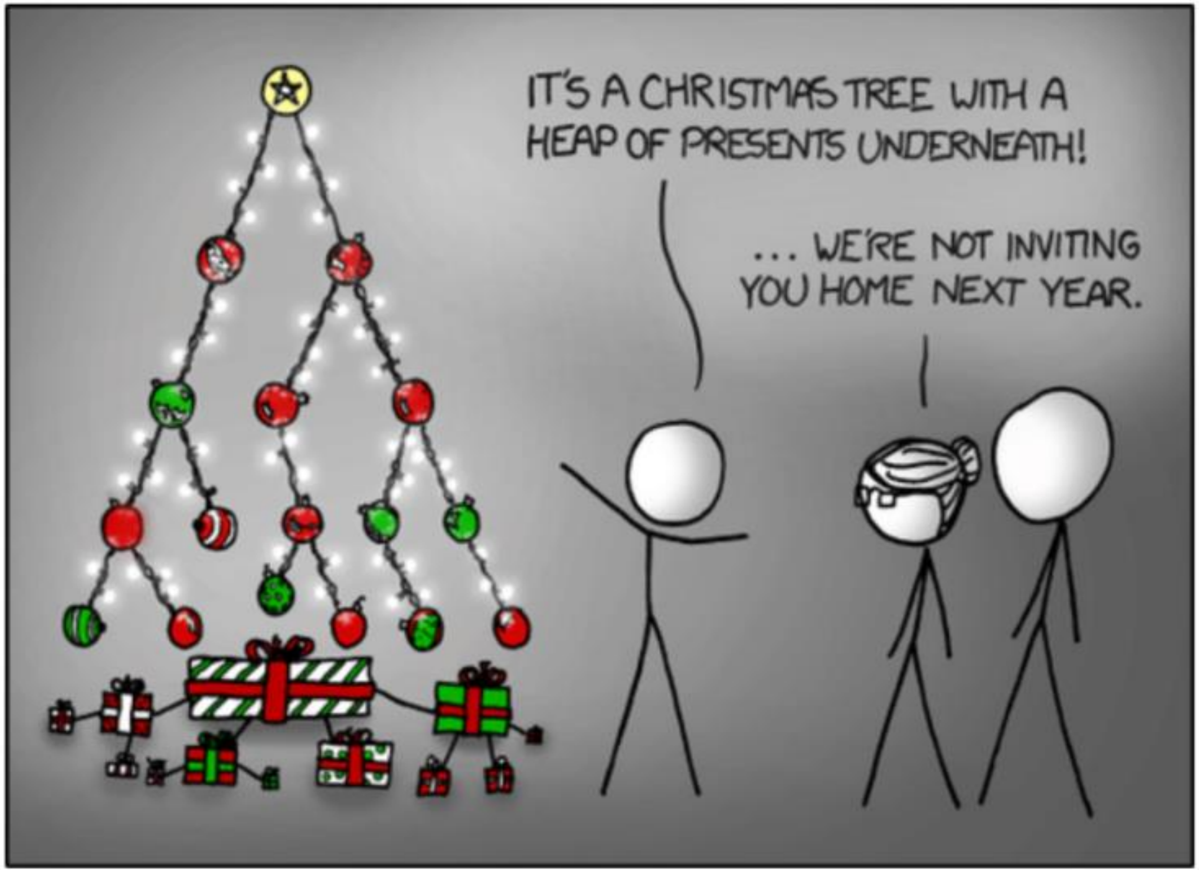
Prof. Dr. Wolfgang Mühlbauer

Fakultät für Informatik

wolfgang.muehlbauer@th-rosenheim.de

Wintersemester 2019/2020

Zur Auflockerung …



Quelle: [3]

- Hoffentlich passiert Ihnen das nicht nach dieser Vorlesung! ☺

Übersicht

- **Einführung**
  - Definition Binärer Suchbaum

- Operationen
  - Traversieren
  - Vorgänger- und Nachfolger
  - Suche und Navigation
  - Einfügen
  - Löschen

- Zusammenfassung

Wiederholung: ADT Map

- **Map, Dictionary, Symboltabelle (dt. "assoziatives Datenfeld")**
  - Speichert Key-Value Pairs (dt. "Schlüssel-Werte-Paare")
  - Bsp.: Alter von Personen → { (Trump, 73), (Merkel, 65), (Kurz, 33) }

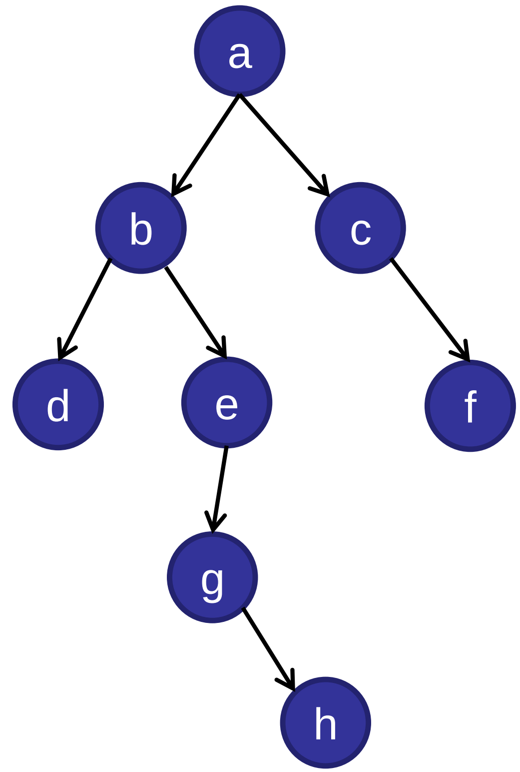
- **Typische Operationen**
  - `void put(Key key, Value value)` (oft auch "`insert`")
  - `Value get(Key key)`
  - `void delete(Key key)` (oft auch "`remove`")
  - `boolean contains (Key key)`
  - `boolean IsEmpty()`
  - `int size`
  - `Iterable<Key> keys` (durchlaufe alle Schlüssel)

- **Annahmen**
  - Jeder Wert hat einen Key. Keys sind eindeutig, keine Duplikate!

- Binäre Suchbäume sind eine Alternative zu Hashtabellen und implementieren ebenfalls die **ADT Map.**
  - Java: `TreeMap`

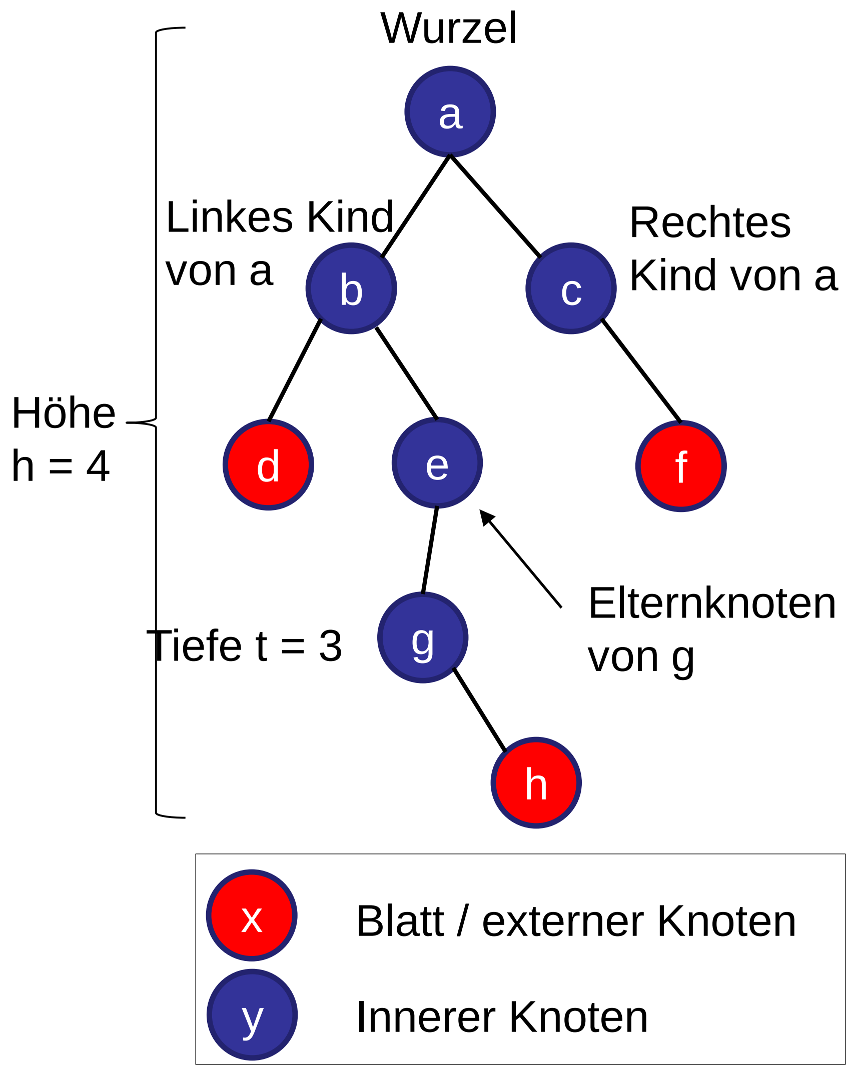
Baum

- Besteht aus
  - Menge ***V***: Knoten (engl. „Vertices“)
  - Menge ***E***: Kanten (engl. „Edges“)
- Jeder Knoten hat
  - 1 Vorgänger / Eltern
  - 1, 2, … Nachfolger / Kinder
- Verallgemeinerung von linearen Listen
  - Jedes Element hat >1 Nachfolger
- Spezieller Fall eines Graphen
  - Es gibt keine Kreise.
- Zahlreiche Anwendungen
  - Syntaxbäume, Entscheidungsbäume, Suchbäume, Kodebäume, etc
  - Struktur zum Speichern von (meist) ganzzahligen Schlüsseln und Schlüssel/Werte Paare (dieses Kapitel)



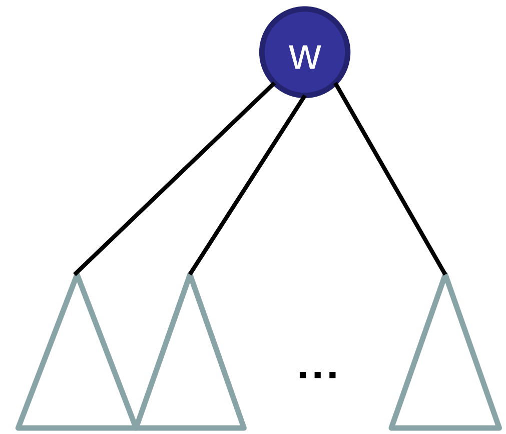
Begriffe

- Kinder und Eltern
- Innere Knoten und externe Knoten (=Blätter)
- ***Wurzel***
  - Einziger Knoten ohne Elternknoten
- ***Tiefe***
  - Entfernung eines Knotens zur Wurzel
- ***Höhe*** eines Baumes *h*
  - Längster Weg von Wurzel zu einem Blatt
- ***Ordnung*** *d*
  - Maximale Zahl von Kindern eines Knotens
  - d = 2: Binärbaum
  - d > 2: Vielwegbäume



Suchbäume

- **Rekursive Definition: Baum mit Ordnung *d*, Höhe *h***
  - Der aus einem einzigen Knoten bestehende Baum ist ein Baum der Ordnung *d*. Die Höhe *h* ist 0.
  - Sind $t_{1,}\ \ldots, t_d$ beliebige Bäume der Ordnung *d,* so erhält man einen (weiteren) Baum der Ordnung *d,* indem man die Wurzeln von $t_{1,}\ \ldots, t_d$ zu Kindern einer neugeschaffenen Wurzel *w* macht. Die Höhe *h* ist $\max\{h(t_1), \ldots, h(t_d)\} + 1$



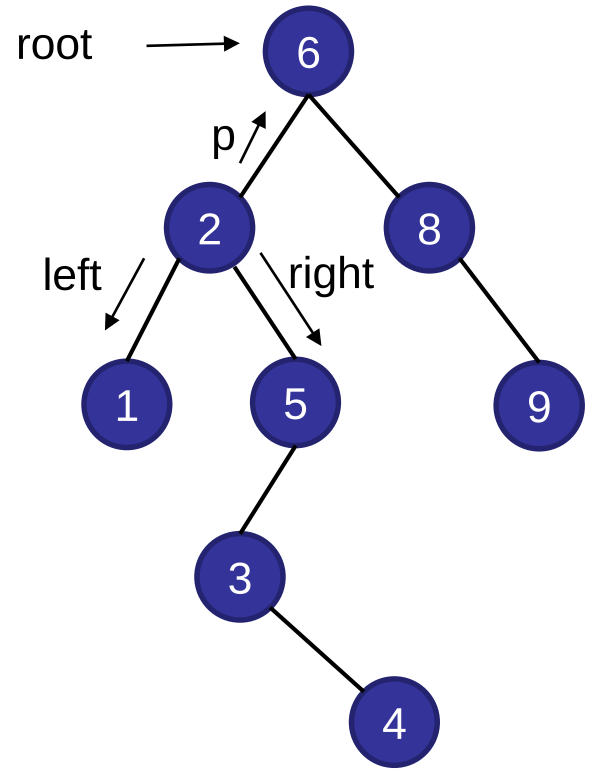
- Baum-basierte Datenstrukturen können Operationen auf dynamischen Mengen implementieren.
  - Beispiel: Binäre Suchbäume, B-Bäume, Rot-Schwarz-Bäume

Übersicht

- Einführung
  - Definition Binärer Suchbaum
- **Operationen**
  - Traversieren
  - Vorgänger- und Nachfolger
  - Suche
  - Einfügen
  - Löschen
- Zusammenfassung

Binäre Suchbäume: Definition

- Binärer Suchbaum **`T`** hat eine Wurzel
  - **`T.root`**: "Zeiger" auf Wurzel
- Jeder **Knoten *x*** ist Objekt und speichert die folgenden **Attribute**
  - **`x.key`**: Schlüssel
  - **`x.left`**: Zeiger auf linkes Kind
  - **`x.right`**: Zeiger auf rechtes Kind
  - **`x.p:`** Zeiger auf Elternknoten
  - und natürliche den Value
- **Definition: Binärer Suchbaum**
  - Baum der Ordnung 2
  - Der Schlüssel in ***jedem*** Knoten ist größer als ***alle*** Schlüssel im linken Teilbaum und kleiner als ***alle*** Schlüssel im rechten Teilbaum.



Operationen wie Einfügen, Suchen und Löschen haben die Laufzeit $O(h)$, wobei h die Höhe des Baumes ist.

In-Order Traversierung

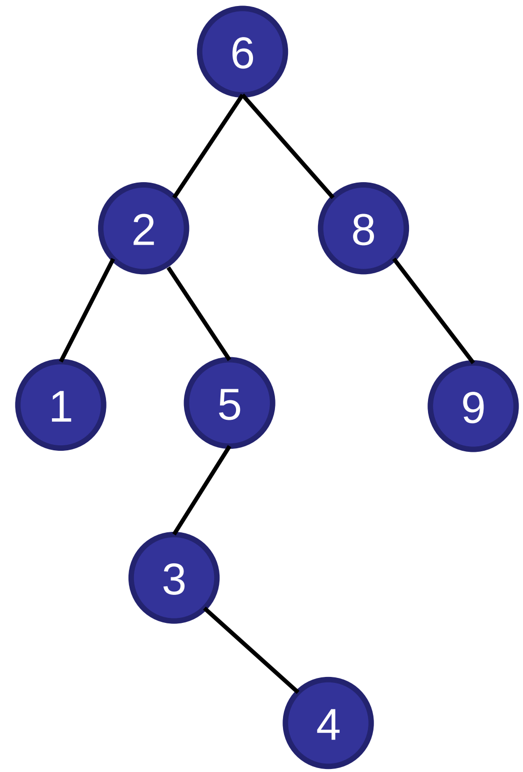
- **Ziel**
  - Gib alle Schlüssel des binären Suchbaums in **aufsteigender Reihenfolge** aus.
- **Idee**
  - Beginne bei Wurzel
  - Gib rekursiv zunächst alle Schlüssel des linken Teilbaumes von *x* aus.
  - Ausgabe des Schlüssels *x*.
  - Gib dann rekursiv alle Schlüssel des rechten Teilbaumes von *x* aus.
- **Laufzeit:** $\Theta(n)$

```
INORDER()
    if x ≠ NIL
        INORDER(x.left)
        print x.key
        INORDER(x.right)
```

Iteration: Durchlaufe Baum in In-Order Reihenfolge

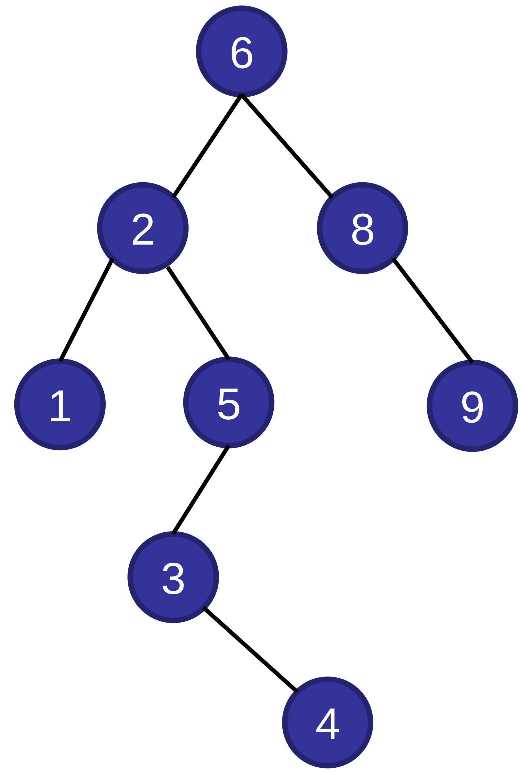
Quellcode: BST.java

In-Order Traversierung in folgendem Baum?



Traversierung eines binären Suchbaumes

- **Mögliche Reihenfolgen**
  - *Preorder*: 6, 2, 1, 5, 3, 4, 8, 9
  - *Inorder*: 1, 2, 3, 4, 5, 6, 8, 9
  - *Postorder*: 1, 4, 3, 5, 2, 9, 8, 6
  - *Levelorder*: 6, 2, 8, 1, 5, 9, 3, 4



- Implementierung von Preorder
  - Besuche Wurzel → linker Teilbaum → rechter Teilbaum

- Aus Angabe der Inorder-Reihenfolge der Knoten lässt sich ***nicht*** eindeutig auf Baum schließen.

Publikums-Joker: Traversierung

Sie kennen die Preorder Reihenfolge der Schlüssel: 1 3 4 6

Wie viele mögliche binäre Suchbäume gibt es?

A. 1

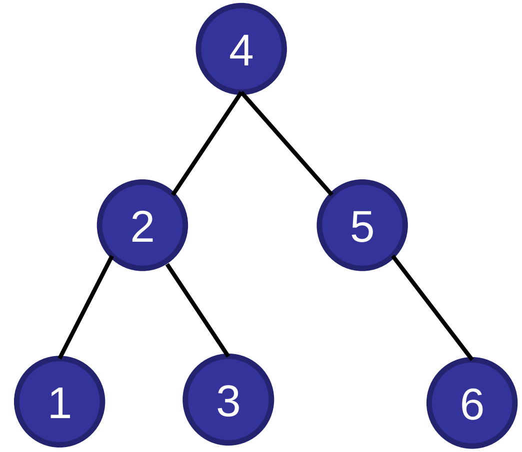
B. 2

C. 3

D. 4

Suche nach Schlüssel *x*

- **Beispiel**
  - Suche 3 und 6 im Beispiel!



- **`GET(Node x, Key key)`**
  - Suche `key` im Teilbaum der `node` als Wurzel hat.
  - *Rekursive* Variante, siehe Pseudocode und `BST.java`
  - Iterative Variante, ähnlich wie bei `put`.

Starte bei Knoten x, suche nach Schlüssel key im Teilbaum von Knoten x

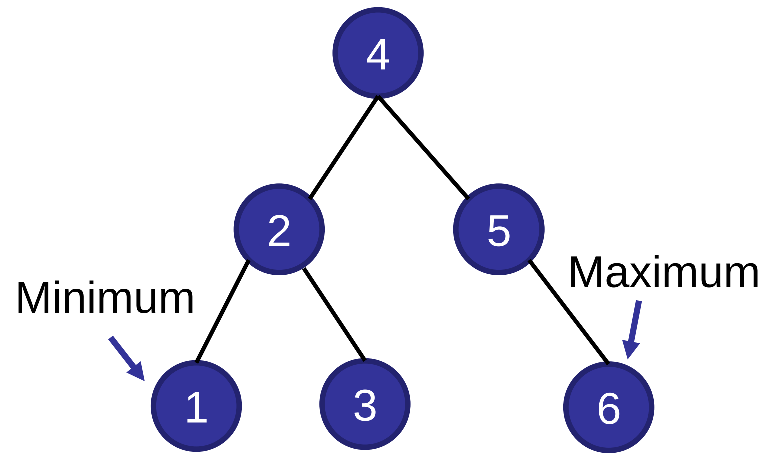
```
GET(Node x, Key key)
    if x == null or x == key[x]
        return x
    if key < x.key
        return GET(x.left, key)
    else
        return GET(x.right, key)

Erster Aufruf: GET(root, k)
```

Quellcode: BST.java

- **Laufzeit:** $O(h)$
  - Falls *h* balanciert: $\Theta(\log n)$
  - Worst Case: $\Theta(n)$
    - Wann tritt dieser ein?

Minimum und Maximum

- Ein binärer Suchbaum garantiert:
  - Der minimale Schlüssel ist "***ganz links"*** im Baum.
  - Der maximale Schlüssel ist "***ganz rechts*** im Baum.



Suche den Knoten mit dem minimalen/ maximalen Schlüssel i im Teilbaum von x

```
MIN(Node x)
    while x.left ≠ null
        x = x.left
    return x
```

- **Laufzeit:** $O(h)$
  - Falls h balanciert: $\Theta(\log n)$
  - Worst Case: $\Theta(n)$

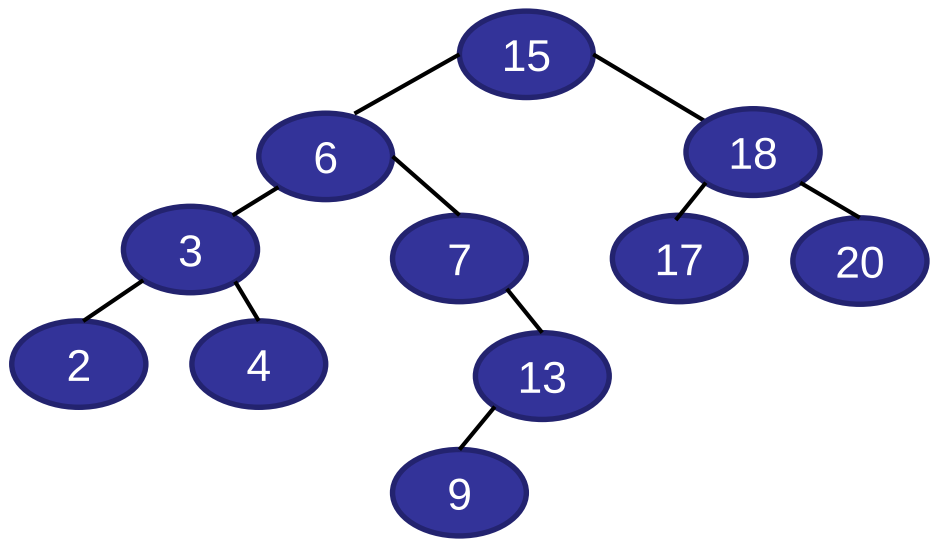
```
MAX(Node x)
    while x.right ≠ null
        x = x.right
    return x
```

Quellcode: BST.java

Vorgänger und Nachfolger

- **Definition:** Der ***Nachfolger*** (***Vorgänger***) eines Knotens *x* ist der Knoten *y*, so dass *y.key* der kleinste (größte) Schlüssel ist, der größer (kleiner) als *x.key* ist.

- **Beispiel:**
  - Nachfolger von 15? $\longrightarrow$ 17
  - Nachfolger von 6? $\longrightarrow$ 7
  - Nachfolger von 4? $\longrightarrow$ 6
  - Vorgänger von 6? $\longrightarrow$ 4



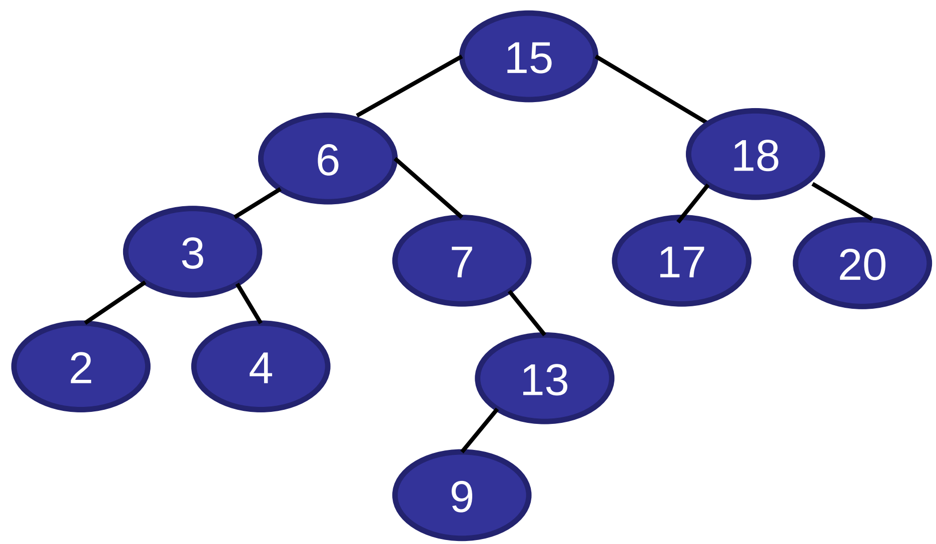
Binärer Suchbaum: Vorgänger und Nachfolger

- **2 Fälle** bei Bestimmung eines Nachfolgers von *x*
  - Fall 1: *x* hat ***nicht leeren rechten*** Teilbaum
    - Nachfolger von *x* ist Minimum im rechten Teilbaum von *x*.
  - Fall 2: *x* hat ***leeren rechten*** Teilbaum
    - Gehe nach oben bis zu einem Knoten *x*, der linkes Kind seines Elternknotens *y* ist.

```
SUCCESSOR(x)
    if x.right ≠ null          // Fall 1
        return MIN(x.right)
    // Fall 2
    y = x.p
    while y ≠ null and x==y.right
        x = y
        y = y.p
    return y
```

Suche Nachfolger von Knoten x

Quellcode: BST.java



- **Laufzeit:** $O(h)$

Übersicht

- Einführung
  - Definition Binärer Suchbaum
- Operationen
  - Traversieren
  - Vorgänger- und Nachfolger
  - Suche
  - **Einfügen**
  - Löschen
- Zusammenfassung

Einfügen eines Knotens *z*

- Beginne an Wurzel, folge Pfad in Richtung Blätter. Man merkt sich 2 "Zeiger"
  - *x*: aktueller Knoten
  - *y*: Elternknoten des aktuellen Knotens
- Vergleiche jeweils Schlüssel von *x* mit einzufügendem Schlüssel und gehe nach links oder rechts weiter.
- Sobald *x* gleich `null`, ist Einfügeposition gefunden.
- Pseudocode nicht auswendig lernen, sondern Idee dahinter verstehen!

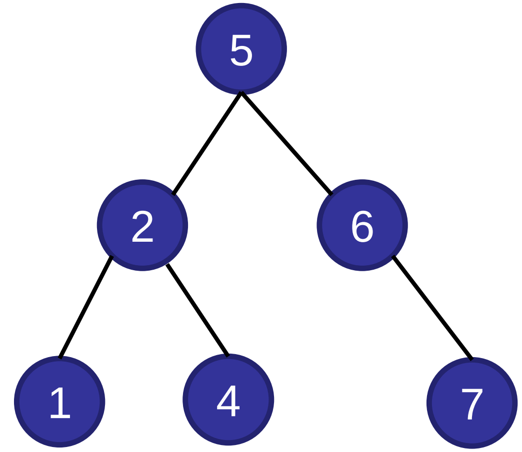
Füge (key, val) in Baum ein

```
PUT(Key key, Value val)
    y = null
    x = root
    while x ≠ null
        y = x
        if key < x.key
            x = x.left
        else
            x = x.right
    Node z = new Node(key, val)
    z.p = y
    if y == null
        root = z  // tree was empty
    elseif z.key < y.key
        y.left = z
    else
        y.right = z
```

Quellcode: BST.java

Binärer Suchbaum: Einfügen - Beispiel

- **Übung**
  - Ergebnis von `PUT(3,??)?`



- **Java**
  - `BST.java, put(.)`
- **Diskussion**
  - Einfügen führt zunächst eine Suche durch.
  - Laufzeit: $O(h)$
  - Worst Case: $O(h) = O(n)$
- ***Animation***
  - https://visualgo.net/bn/bst
  - https://www.cs.usfca.edu/~galles/visualization/BST.html

Übersicht

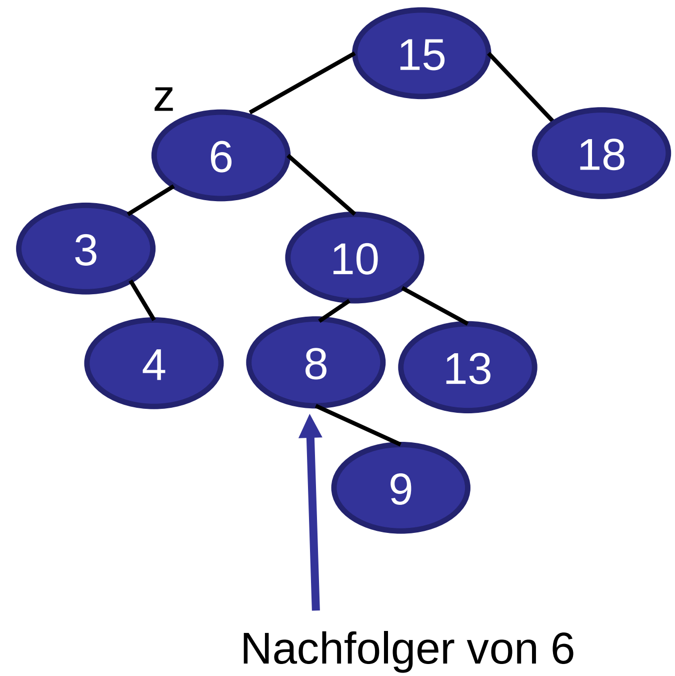
- Einführung
  - Definition Binärer Suchbaum
- Operationen
  - Traversieren
  - Vorgänger- und Nachfolger
  - Suche
  - Einfügen
  - **Löschen**
- Zusammenfassung

Binärer Suchbaum: Löschen eines Knoten *z*

- Löschen deutlich komplizierter als Einfügen

- **3 Fälle** bzgl. des zu löschenden Knotens ***z*** sind zu unterscheiden:
  - ***Fall 1:*** *z* hat **keine** Kinder (z=4)
    - Einfach *z* entfernen
  - ***Fall 2***: *z* hat **1** Kind (z=3)
    - Setze Kind an Position von *z (Bsp. z=3)*
    - *Nur linkes Kind (Fall 2b)*
    - *Nur rechtes Kind (Fall 2a)*
  - ***Fall 3***: *z* hat **2** Kinder (z=6)
    - Idee: Finde Nachfolger von *z*, setze diesen an die Stelle von *z*.
    - Hinweis: Nachfolger von z hat höchstens 1 Kind!

Löschen von 6!



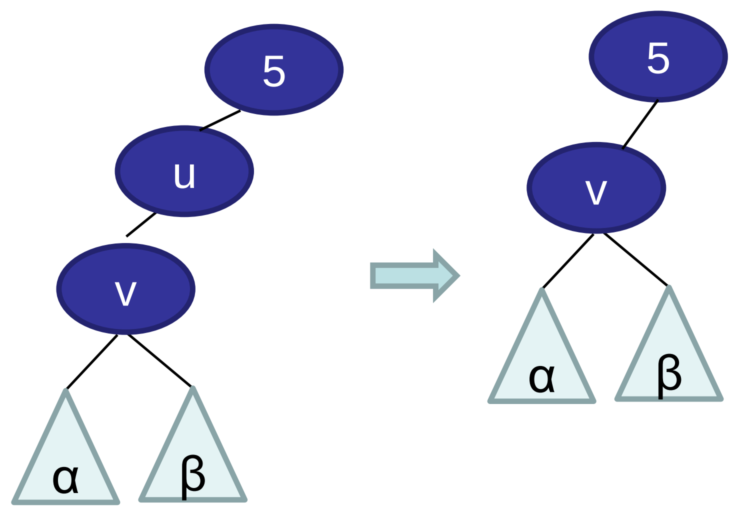
Hilfsoperation TRANSPLANT

- Beim Löschen müssen Teilbäume verschoben bzw. anderswo eingehängt werden.

- **TRANSPLANT(*u*,*v*)** verschiebt Teilbaum von *v* an die Stelle des Teilbaumes von *u*.
  - Zeile 7-8: Mache Elternknoten von *u* zu Elternknoten von *v*
  - Zeile 3-6: Elternknoten von *u* bekommt *v* als linkes oder rechtes Kind (je nachdem ob *u* linkes oder rechtes Kind war).
  - Passt *v.left* bzw. *v.right* **nicht** an.

Verschiebt Teilbaum von v an u

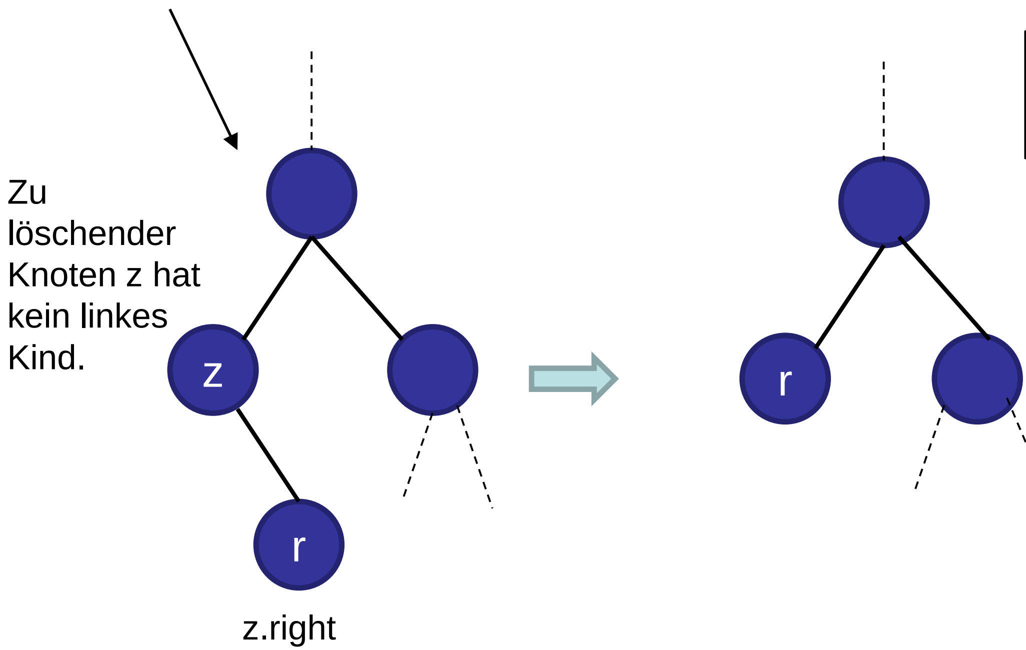
```
TRANSPLANT(u, v)
1    if u.p == null
2       root = v
3    elseif u == u.p.left
4       u.p.left = v
5    else
6       u.p.right = v
7    if v ≠ null
8       v.p = u.p
```



Fall 1 und Fall 2a

- **Annahme**
  - Der zu löschende Knoten z hat **kein linkes Kind.**
  - Schließt Fall, dass überhaupt kein Kind, mit ein.

- **Aktion**
  - Setze rechtes Kind von *z* an die Stelle von *z*.
  - Das rechte Kind kann auch NIL sein (Fall 1).



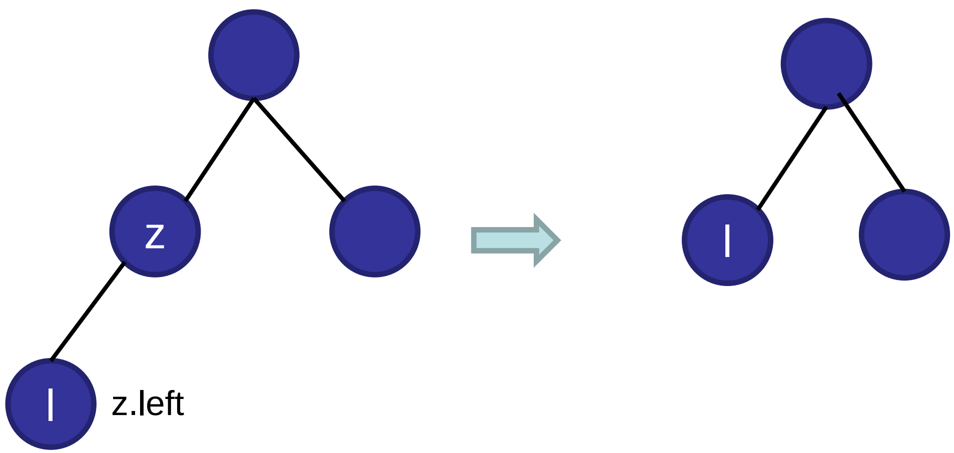

```
AKTION
TRANSPLANT(z, z.right)
```

Hinweis: Die folgenden Skizzen zeigen nur einen Teilausschnitt eines fiktiven Baumes.

Fall 2b

- **Annahme:**
  - *z* hat **nur 1 Kind** und dieses Kind ist das **linke Kind**.

- **Aktion**
  - Ersetze *z* durch das linke Kind.

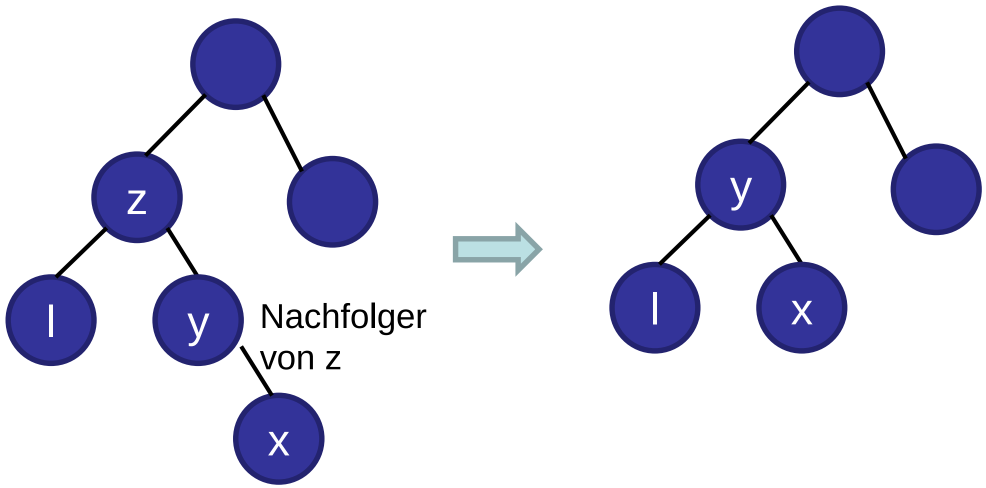


<u>AKTION</u>
TRANSPLANT(*z*, *z.left*)

Fall 3a

- **Annahme**
  - *z* hat ***2 Kinder***.
  - ***y*** sei ***Nachfolger*** von *z* im rechten Teilbaum.
  - UND: Der Nachfolger *y* ist ***gleichzeitig*** das ***rechte Kind von z***.

- **Aktion**
  - Setze Teilbaum von y an die Stelle von z.
  - Hinweis: *y* kann kein linkes Kind haben, denn sonst wäre es nicht der Nachfolger von z.




```
AKTION
TRANSPLANT(z, y)
y.Left = z.left
y.left.p = y
```

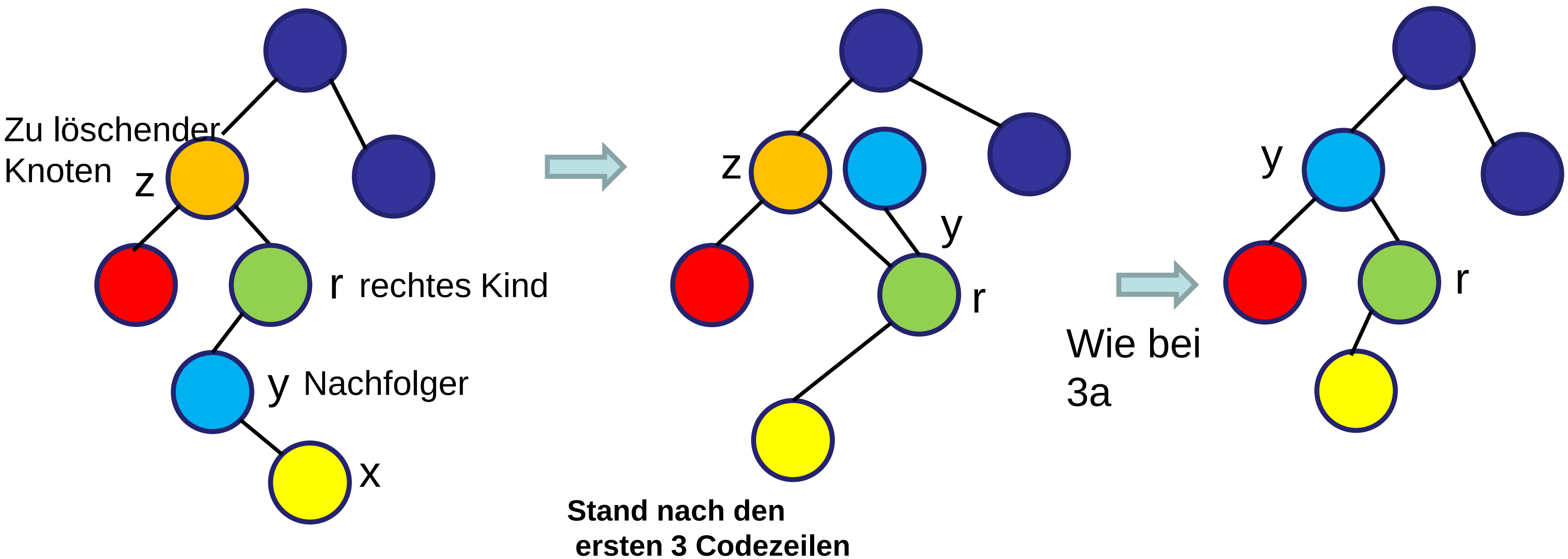
Fall 3b

- **Annahme**
  - ***z*** hat **2 Kinder**.
  - ***y*** sei **Nachfolge**r von *z*
  - ABER***: y*** NICHT rechtes Kind von **z** (siehe 3a)
  - Das rechte Kind von ***z*** sei ***r***.
- **Aktion**
  - "Mache y frei", x nimmt die Stelle von y ein.
  - Mache y zu Elter vom rechten Kind r des zu löschenden Knoten y.
  - Weiter wie bei 3a.

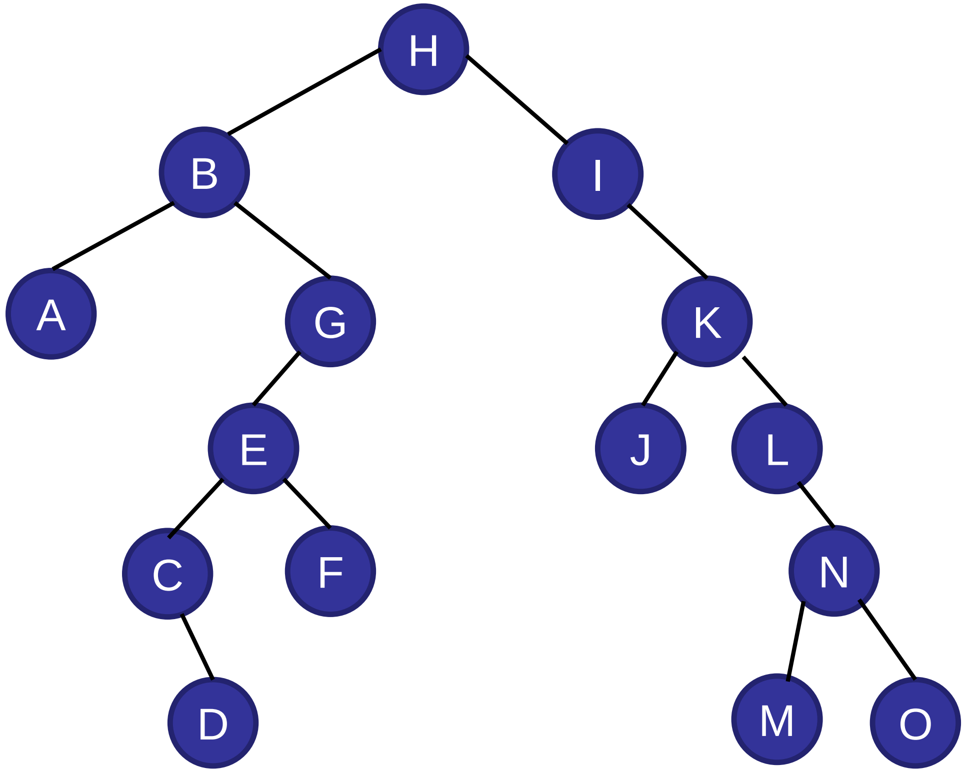
AKTION

```
TRANSPLANT(y, y.right)
y.right = z.right
y.right.p = y
TRANSPLANT(z, y)
y.left = z.left
y.left.p = y
```

Wie Fall 3a



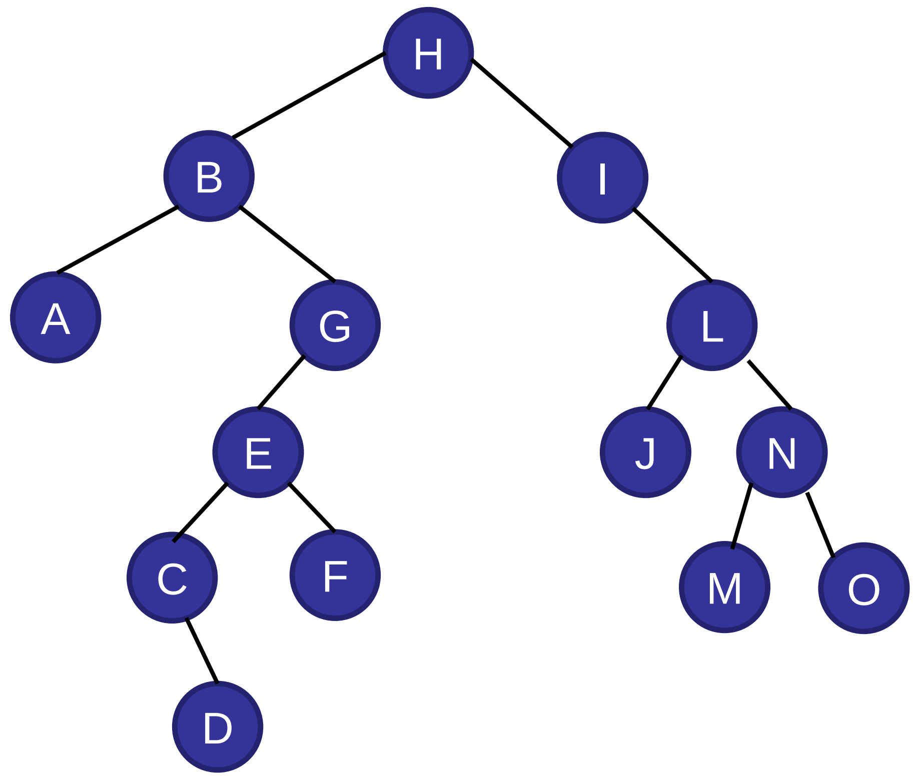
Übung (Fall 3a)



- Wie sieht der Baum nach `DELETE`(K) aus?

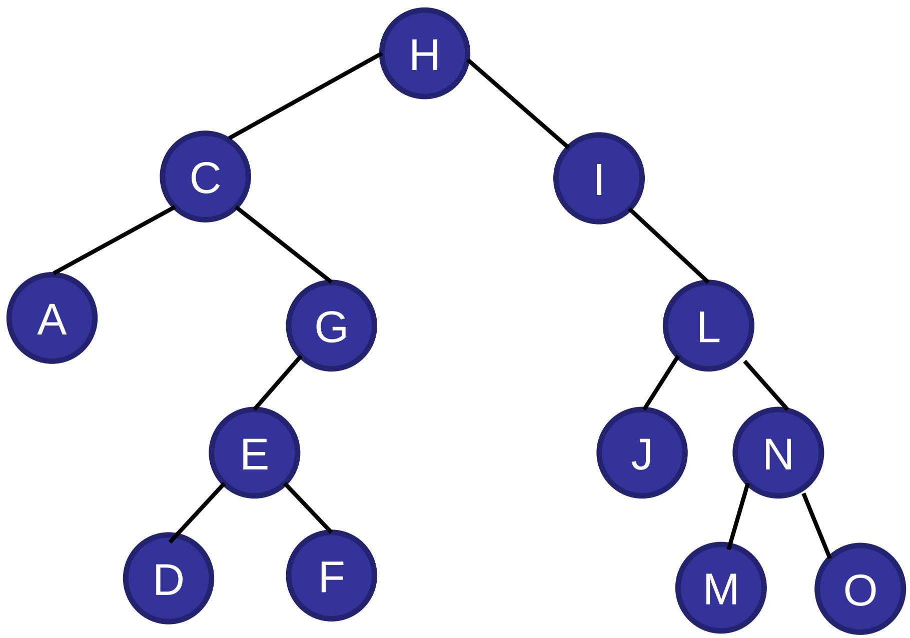
Übung (Fall 3b)

Ergebnis



- Nun soll als nächstes B gelöscht werden.
- Wie sieht der Baum nach $\texttt{DELETE}(B)$ aus?

Übung: Ergebnis



Publikums-Joker: Binäre Suchbäume

Was macht der unten abgebildete Code für einen binären Suchbaum?

A. Er zählt die Blätter.
B. Er zählt die inneren Knoten.
C. Er gibt die Höhe des Baumes zurück.
D. Er gibt die Länge des längsten Pfades im Baum zurück.


```
int function(Node root)
{
   if (root == null)
      return 0;
   if (root.left == null && root.right == NULL)
      return 0;
   return 1 + fun(root.left) + fun(root.right);
}
```

Diskussion

- Löschen ist trickreich!

- **Laufzeit:** $O(h)$
  - TRANSPLANT hat konstante Laufzeit: $O(1)$
  - Der Aufruf von MINIMUM zur Bestimmung des Nachfolgers hat $O(h)$

- **Implementierung**
  - Siehe Übung

- Allgemein: Laufzeit für Basisoperationen in binärem Suchbaum mit *n* Schlüsseln abhängig von der Höhe *h* des Baumes
  - Worst Case: O(*h*) = O(*n*)
  - Best Case : O(*h*) = O(log *n*)
  - Ziel: Halte binären Suchbaum balanciert!

Zusammenfassung

- Baum als Datenstruktur
- Binäre Suchbäume
  - Navigation: Vorgänger, Nachfolger, Traversieren
  - Suchen, Einfügen und Entfernen von Schlüsseln
  - Laufzeitanalyse
- Ausblick
  - Im schlimmsten Fall hat die Suche in einem binären Suchbaum lineare Laufzeit: $O(n)$
  - Wie kann man verhindern, dass dieser Worst Case auftritt?
  - Wie kann man binäre Suchbäume balanciert halten?
  - Lösung: Rot-Schwarz-Bäume, siehe nächstes Kapitel!

Quellenverzeichnis

[1] Cormen, Leiserson, Rivest and Stein. *Introduction to Algorithms*, Third Edition, The MIT Press, 2009.

[2] Ottmann, Widmayer. *Algorithmen und Datenstrukturen*, Kapitel 5.1, 5. Auflage, Spektrum Akademischer Verlag, 2012.

[3] Quelle: https://cs124.quora.com/xkcd-comics

[4} Sedgewick, Wayne. Algorithms, Fourth Edition, Addison-Wesley, 2011