Einführung in Programmierung und Datenbanken

joern ploennigs

Algorithmen: Sortieren und Suchen

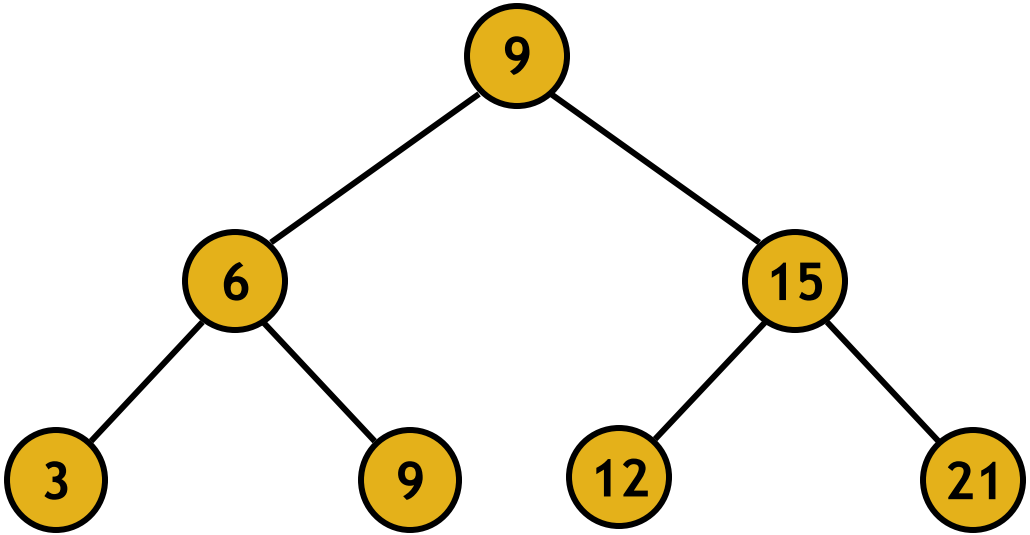
Midjourney: A bear economist in front of a stock chart crashing, digital art.

Rekursion - Beispiel: Tree Traversal

- Baum: Datenstruktur, in der jedes Element auf weitere Unterelemente verweisen kann.
- Nun wollen wir eine Funktion auf jedes Element im Baum anwenden.

- Simpelste Repräsentation eines Baumes:
  - Ein Tupel mit einem Wert und einer Liste
  - Diese Liste enthält Tupel mit jeweils einem Wert und einer Liste (Rekursive Datenstruktur!)

```
baum = (12, [
    (6,[
        (3,[]),
        (9,[])
        ]),
    (18,[
        (15,[]),
        (21,[])
        ])
])
```



Rekursion - Beispiel: Tree Traversal

Traversals sind Funktionen welche den Baum durchlaufen, z. B. um Elemente zu suchen.

```
def traverse(tree):
    wert=tree[0]                # der aktuelle Wert
    print(wert)                 # gebe den aktuellen Wert aus
    if tree[1]:                 # wenn Kinder definiert sind
        for child in tree[1]:   # iteriere durch alle Kinder
            traverse(child)     # und rufe die Funktion rekursiv auf

traverse(baum)
```

Ausgabe

```
12, 6, 3, 9, 18, 15, 21
```

Informatik-Exkurs: Sortieren

- Speziell das Sortieren von Listen, unabhängig vom Datentyp

- Eine der Standard-Anwendungen für Schleifen und Rekursion
- Die Lösung im Programmieralltag: `sorted()`

- Aber: Gutes Beispiel für strukturiertes Informatik-Problem

Sortieren - Algorithmus 1: Bubble Sort

- Jedes Element der Liste wird durchlaufen und mit dem nächsten Element verglichen.
- Ist das zweite Element kleiner als das erste wird die Position getauscht.
- Die Liste wird immer wieder durchlaufen bis dieser Fall nicht mehr auftritt.

```
def bubbleSort(numbers):
    for i in range(len(numbers)-1):
        for j in range(0, len(numbers)-i-1):
            if numbers[j] > numbers[j + 1] :
                numbers[j], numbers[j + 1] = numbers[j + 1], numbers[j]
```

Sortieren - Algorithmus 2: Quick Sort

- Das Grundprinzip ist das Aufteilen der Liste

- Das erste Element der Liste als „Pivot“-Element abspeichern.
- Dann wird die Liste durchlaufen und jedes Element mit dem Pivot verglichen.
- Einsortiert in eine von drei Listen: Kleiner, Gleich und Größer.
- Dann wird Quick Sort rekursiv auf die Listen Kleiner und Größer ausgeführt.
- Am Ende werden alle Listen rekursiv wieder zusammengeführt.

Informatik-Exkurs: Suchen

- Speziell das Suchen auf sortierten Listen und Bäumen

- Meist kennt man die Anzahl an Elementen, weiß also genau wo z. B. die Mitte ist.

- Spätere Vorlesungen: In echten Anwendungen haben Datensätze oft mehr als nur einen Wert (Tabellen anstatt Listen), hier arbeitet man meist mit Indexierung.

Suchen - Suche in Listen

- Die einfachste Methode: lineare Suche

```
def contains(list, x):
    for l in list:
        if(l == x):
            return True
    return False
```

Suchen - Suche in Listen

- Das optimale Verfahren: Binäre Suche
- Deutlich komplexerer Code

- Idee: Mittleres Element der Liste finden, dann mit dem gesuchten Wert vergleichen
  - Wert kleiner: Selbes Verfahren für die erste Hälfte der Liste
  - Wert größer: Selbes Verfahren für die zweite Hälfte der Liste

- Implementation meistens über Rekursion

Suchen - Suchen in Bäumen

- Strategien um einen unsortierten Baum zu durchsuchen:
  - Breitensuche: Vom Ursprung beginnend jede „Ebene“ des Baumes von links nach rechts durchlaufen.
  - Tiefensuche: Einem Pfad vom Ursprung bis zum Ende folgen, dann schrittweise rückwärts gehen bis sich weitere Pfade anbieten.

Suchen - Suchen in Bäumen

- In sortierten Bäumen (Suchbäumen) kann das Ergebnis extrem effizient gefunden werden, da nur ein Pfad durchlaufen werden muss
- Wie sortiert man einen Baum? Erstellen eines binären Suchbaums

- Beispiel:

12 6 18 15 3 21 9

SUCHBAUM

12 6 18 15 3 21 9

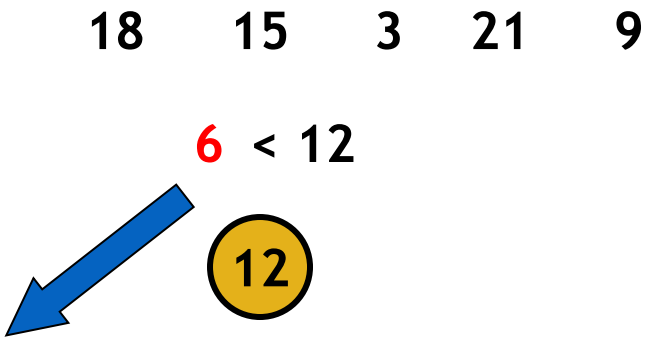
SUCHBAUM

6 18 15 3 21 9

SUCHBAUM

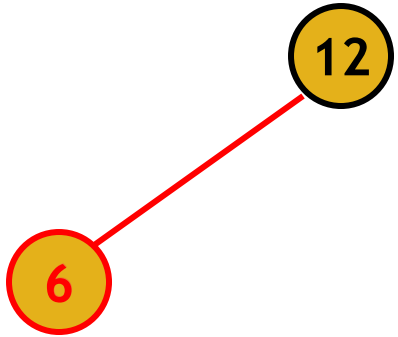
6 18 15 3 21 9

SUCHBAUM



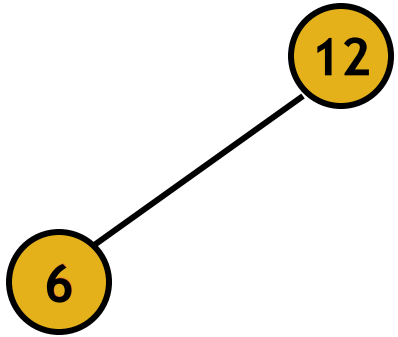
Suchbaum

18 15 3 21 9



SUCHBAUM

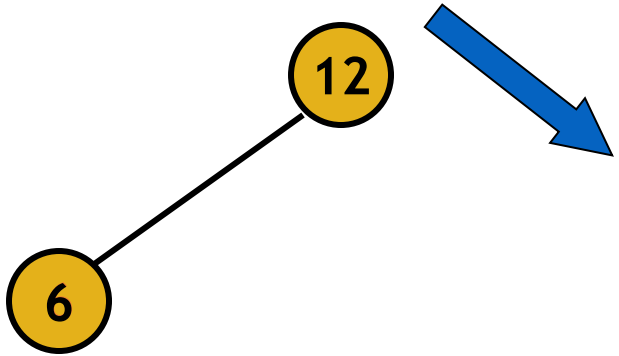
18 15 3 21 9



SUCHBAUM

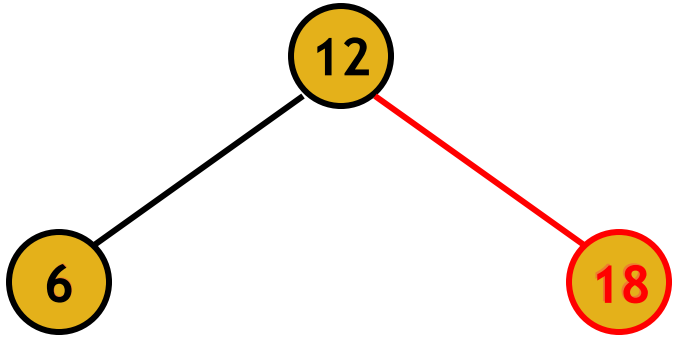
15 3 21 9

18 > 12



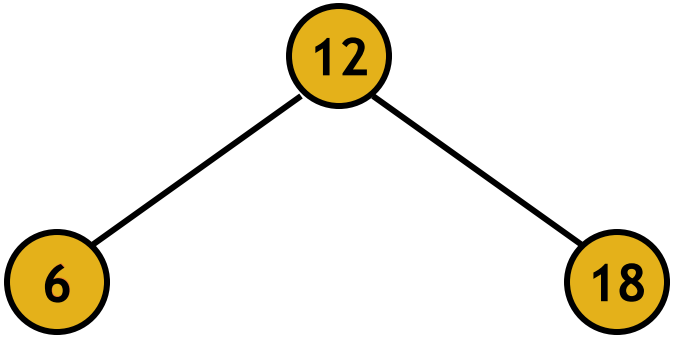
Suchbaum

15 3 21 9



Suchbaum

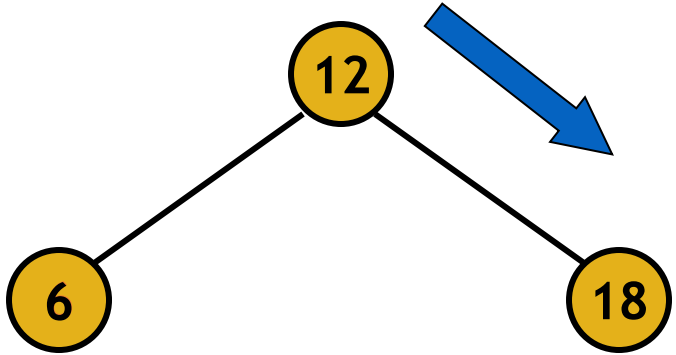
15 3 21 9



SUCHBAUM

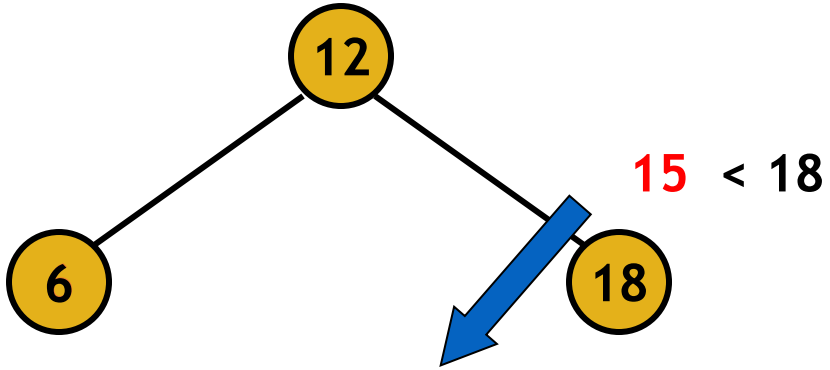
3 21 9

15 > 12



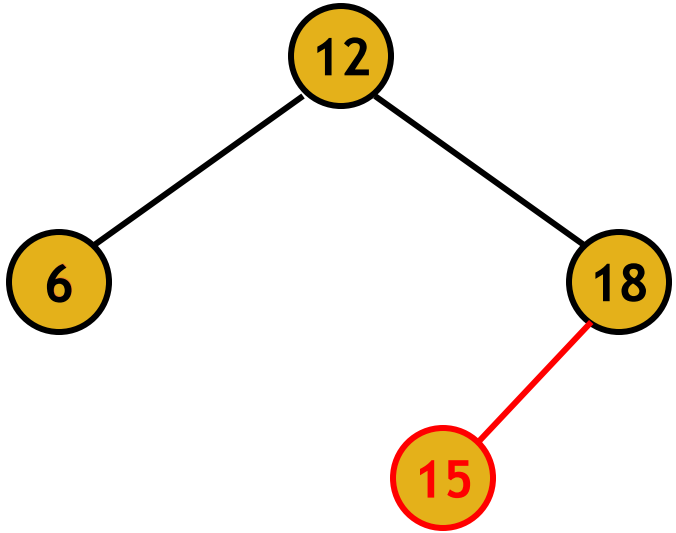
SUCHBAUM

3 21 9



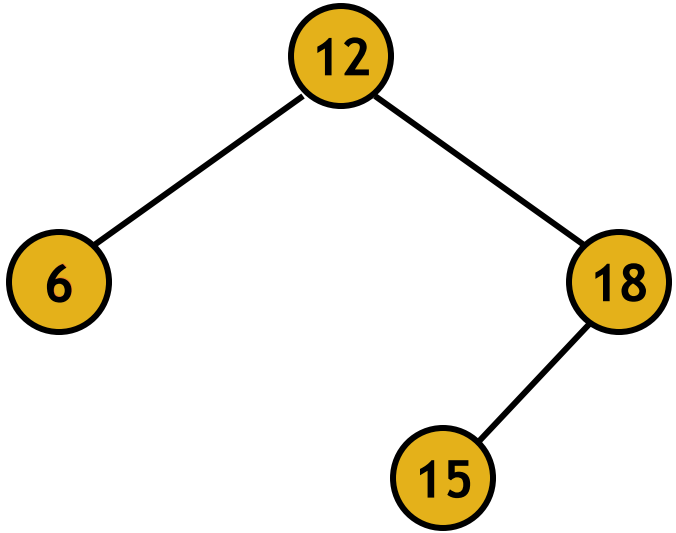
SUCHBAUM

3 21 9



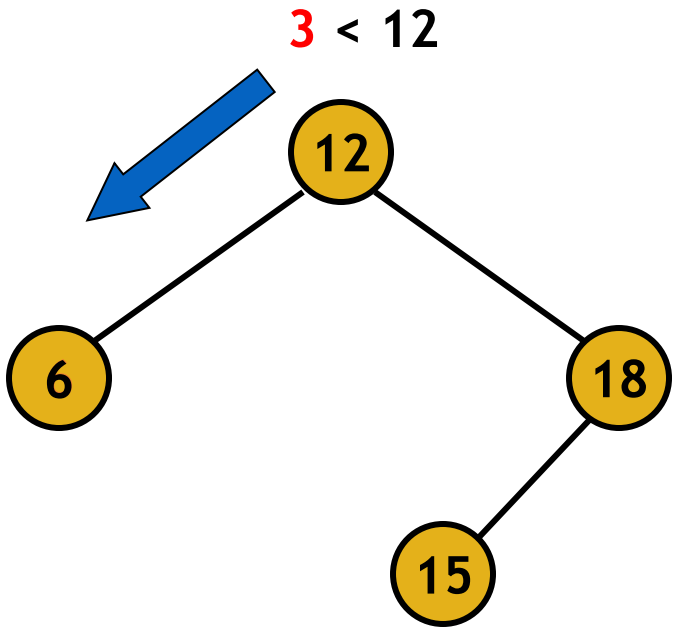
SUCHBAUM

3 21 9



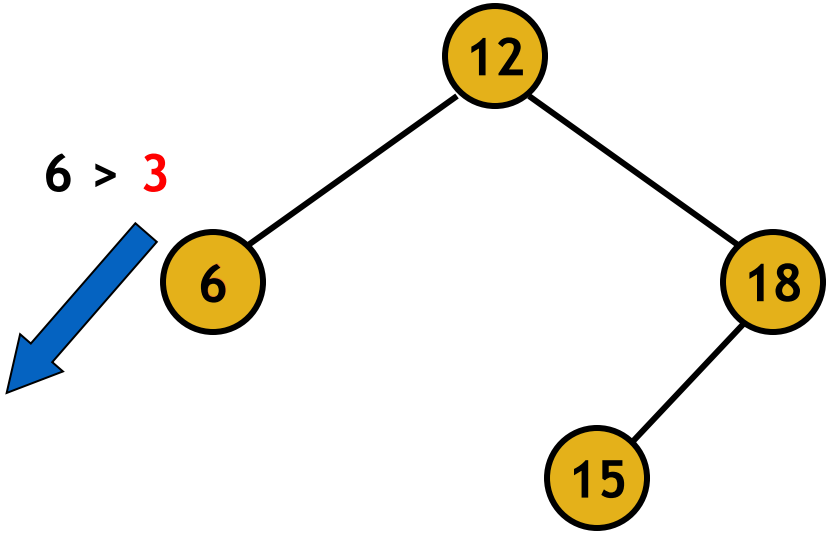
SUCHBAUM

21 9



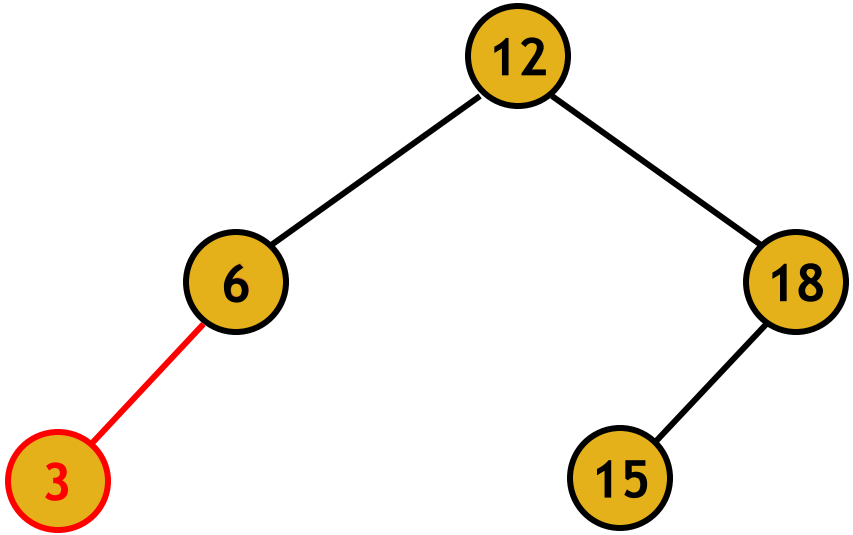
SUCHBAUM

21 9



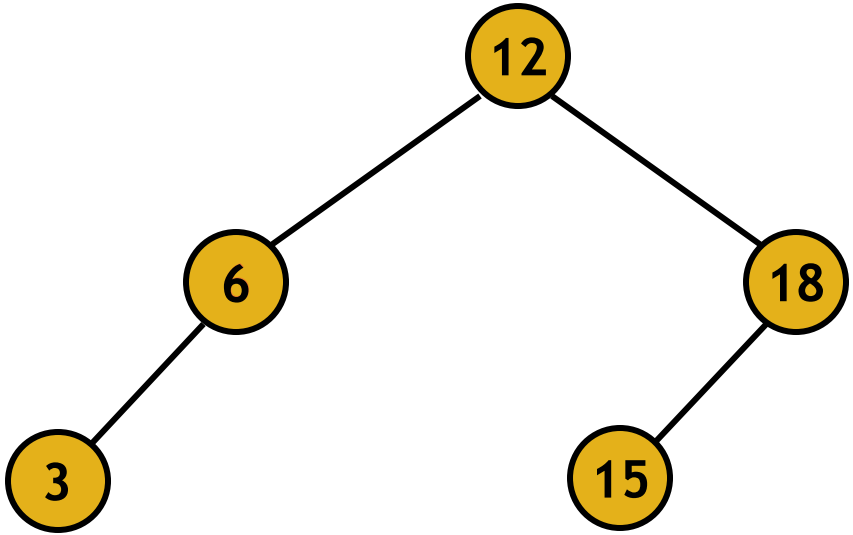
SUCHBAUM

21 9



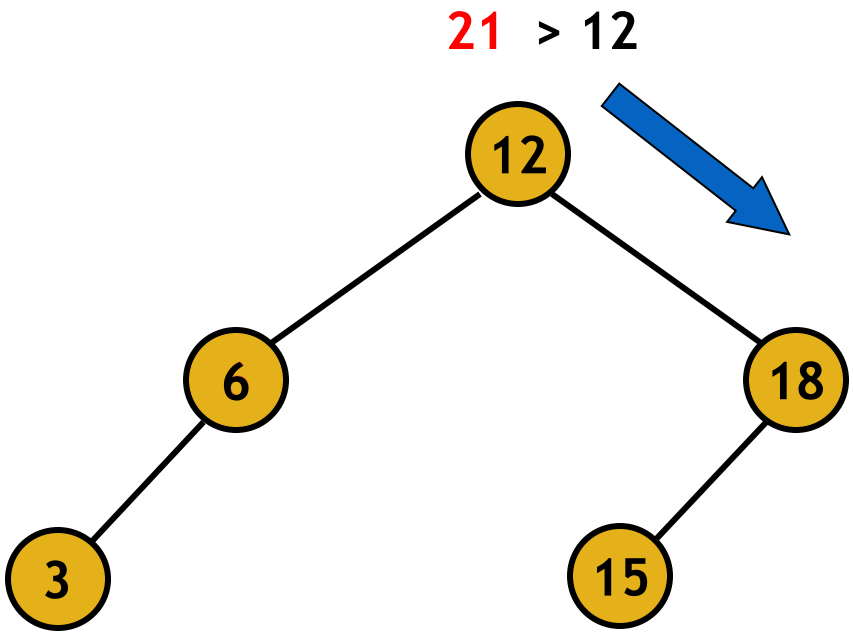
SUCHBAUM

21 9

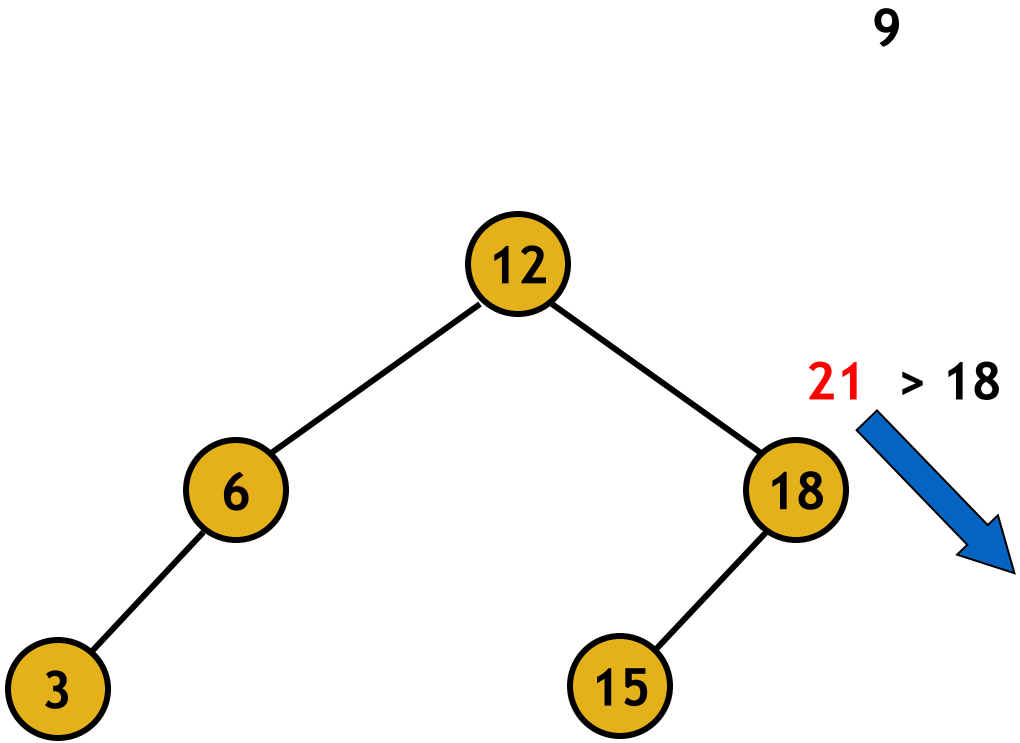


SUCHBAUM

9

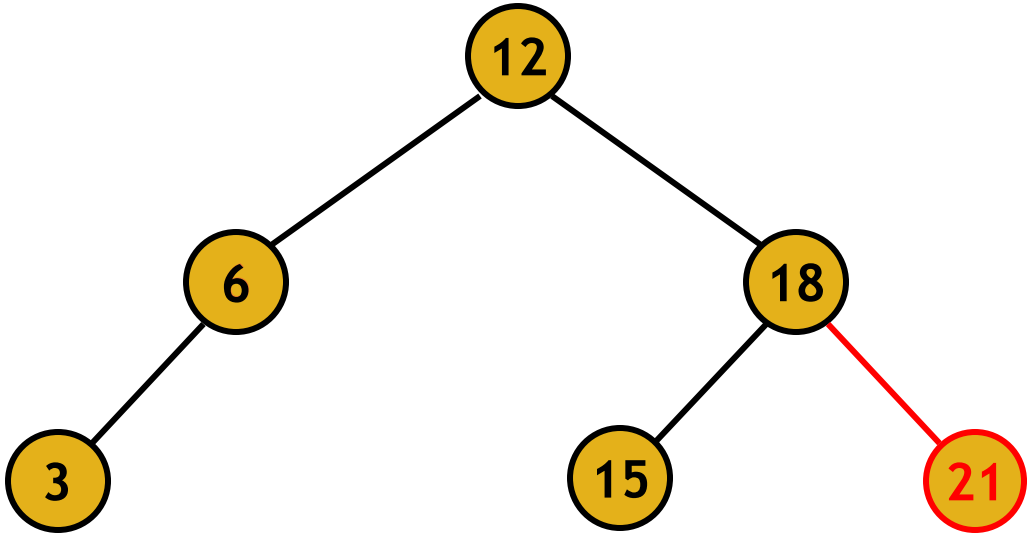


SUCHBAUM



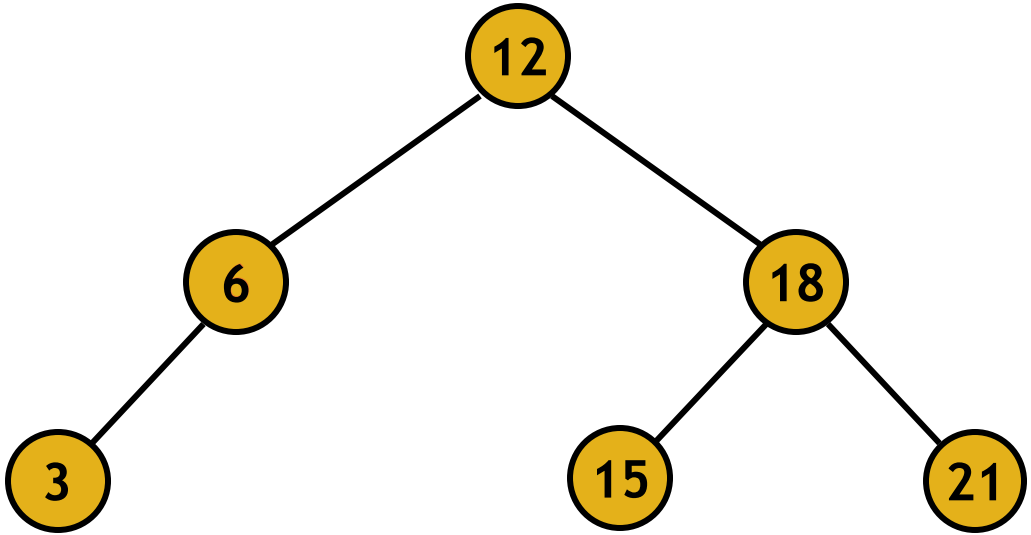
Suchbaum

9

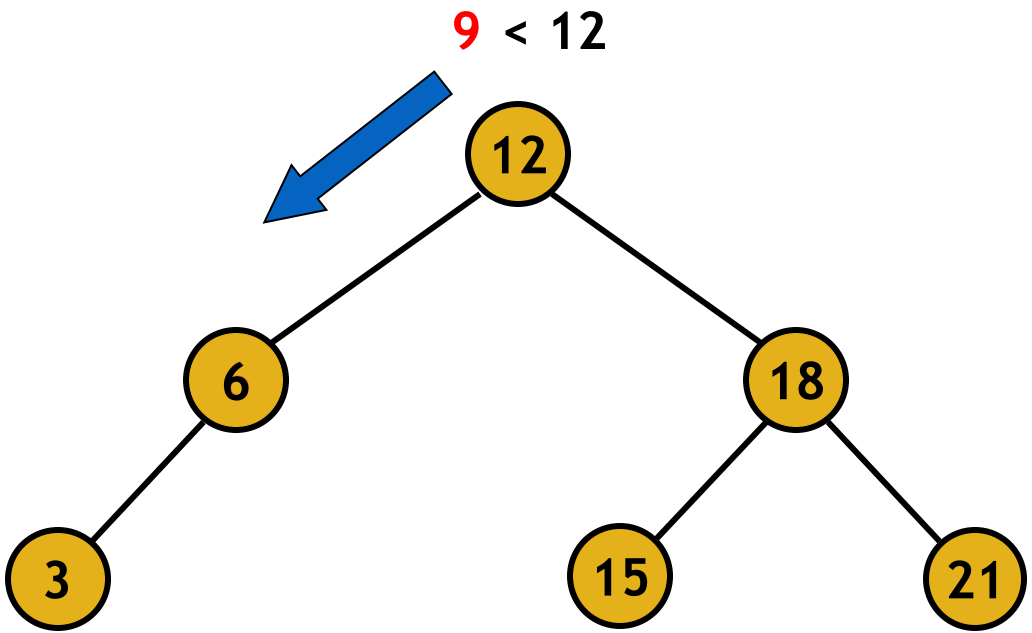


Suchbaum

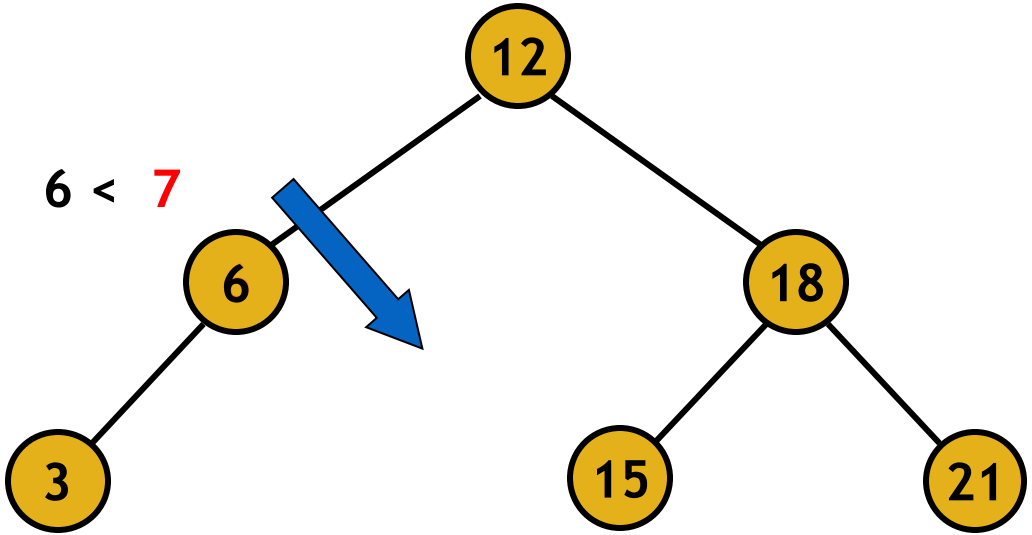
9



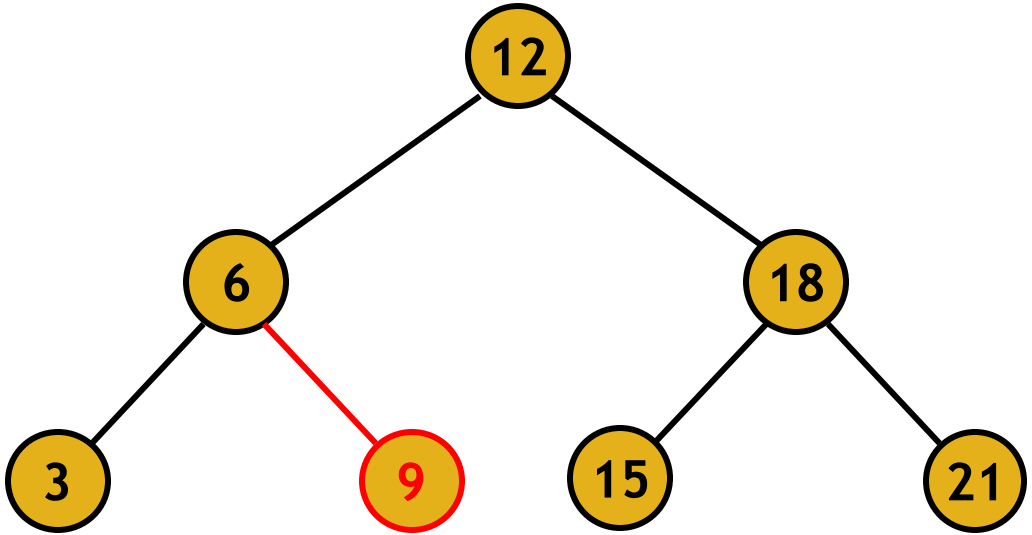
Suchbaum



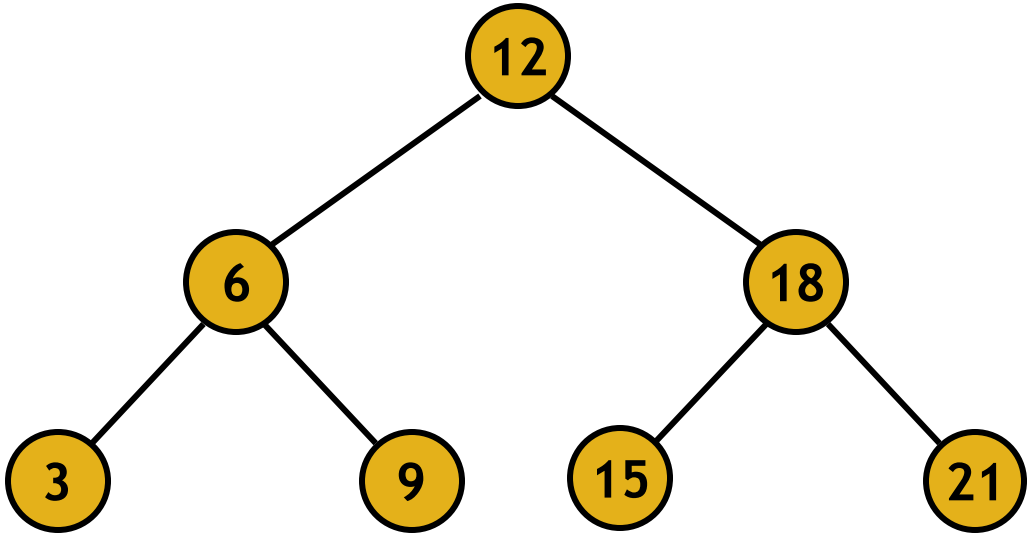
Suchbaum



Suchbaum



Suchbaum



Exkurs: Komplexität und effiziente Algorithmen

- Die eigentliche Berechnungszeit ergibt sich erst durch die genauen Eingabedaten und den genutzten Prozessor.
- Formal ist Effizienz stattdessen eher eine Anstiegskurve, welche eine relative Schätzung abgibt wie viel länger ein Algorithmus bei mehr Eingabedaten rechnen muss.
- Notation: O(x), wobei x die Menge der nötigen Rechenschritte angibt.
- Für Bäume und Listen hängt diese von der Anzahl an Elementen ab (Variable n).

  - Lineare Suche: O(n) Bubble Sort: $O(n^2)$
  - Binäre Suche: O(log(n)) Quick Sort: O(n*log(n))

Hörsaalfrage

FRAGEN?

DALL·E 2: A psychedelic DJ with a question mark for a head