Algorithmen und Datenstrukturen 1

Prof. Dr. Carsten Lecon

Elementare Datenstrukturen

- Einführung
- Stapel
- Schlangen
- Verkette Listen
- Repräsentation von Bäumen
- Zusammenfassung

Bildquellen: presse.openface.de; www.bcm-news.de

Binäre Suchbäume

- Einleitung: Was sind binäre Suchbäume?
- Anfragen an binäre Suchbäume
- Einfügen und Löschen von Knoten in binären Suchbäumen

Binäre Suchbäume

- Bisherige Datenstrukturen (Felder, Listen, …) ineffizient für manche Operationen
- Beispiel Wörterbuch, Operationen:
  - Einfügen
  - Löschen
  - Suchen
  - Ermittlung Minimum/Maximum
  - Ermittlung Vorgänger/Nachfolger
- Es wird zu beobachten sein, dass binäre Suchbäume nicht immer (automatisch) in günstiger Struktur wachsen.

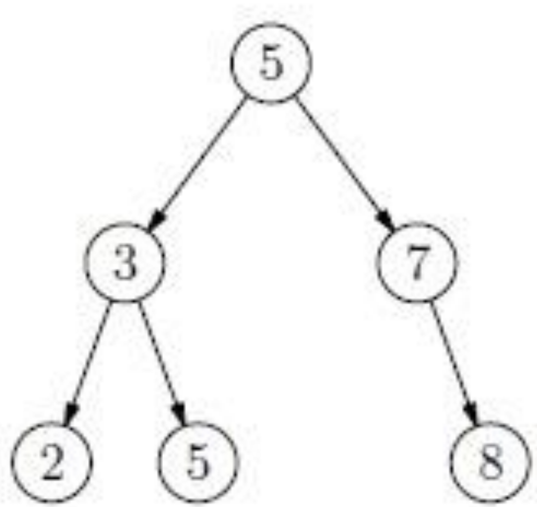
Binäre Suchbäume

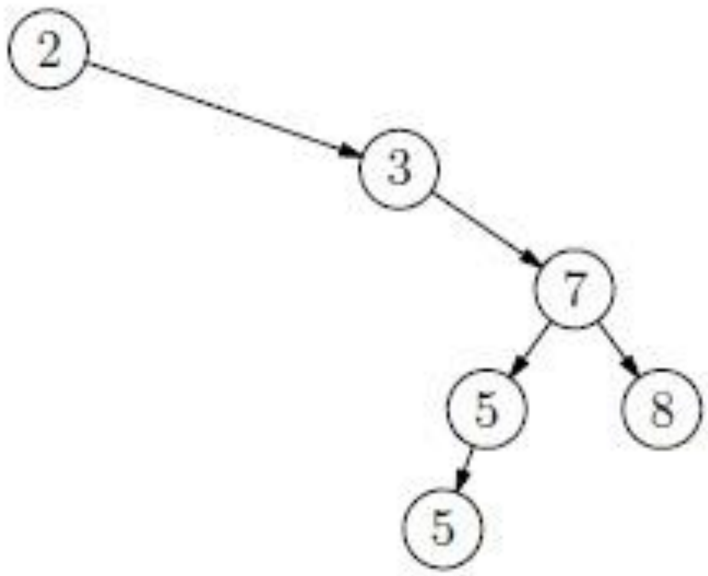
Definition

- Ein binärer Suchbaum (engl. *binary search tree* (BST)) ist ein binärer Baum, für den die binäre Suchbaumeigenschaft (BST-Eigenschaft) gilt:
  - Sei x ein Knoten in einem binären Suchbaum.
  - Ist y ein Knoten im linken Teilbaum von x, so gilt key(y) ≤ key(x).
  - Ist y ein Knoten im rechten Teilbaum von x, so gilt key(x) ≤ key(y).

Binäre Suchbäume

Beispiele





Binäre Suchbäume

Anmerkungen:

- Wir betrachten BST´s als verzeigerte Datenstruktur.
- Jeder Knoten ist ein Objekt mit einem Schlüsselwert.
  - Schlüssel
  - Satellitendaten
  - (Zeiger auf) linkes Kind
  - (Zeiger auf) rechtes Kind
  - (Zeiger auf) Elternknoten

Binäre Suchbäume

Notation:

T	Der BST selber
x	Knoten
key(x)	Schlüssel des Knotens x
left(x)	(Zeiger auf) linkes Kind
right(x)	(Zeiger auf) rechtes Kind
parent(x)	(Zeiger auf) Elternknoten
NIL	Leerer Zeiger

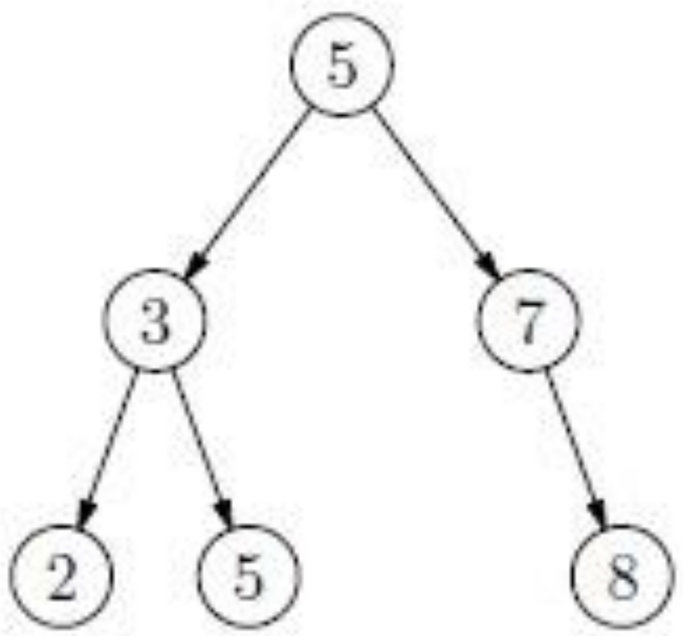
Binäre Suchbäume

Traversierung (in Ordnungsreihenfolge)

```
procedure inorderTreeWalk(x)
  if x≠NIL then
    inorderTreeWalk(left(x))
    print(key(x))
    inorderTreeWalk(right(x))
  end if
end procedure
```

Binäre Suchbäume

Beispiel für Inorder-Suche:



Binäre Suchbäume

Satz

- Sei x Wurzel eine Teilbaums mit n Knoten.
- Dann benötigt der Aufruf von inorderTreeWalk die Laufzeit $T(n) = O(n)$.

Binäre Suchbäume

```
procedure preorderTreeWalk(x)
  if x≠NIL then
    print key(x)
    preorderTreeWalk(left(x))
    preorderTreeWalk(right(x))
  end if
end procedure
```

Binäre Suchbäume

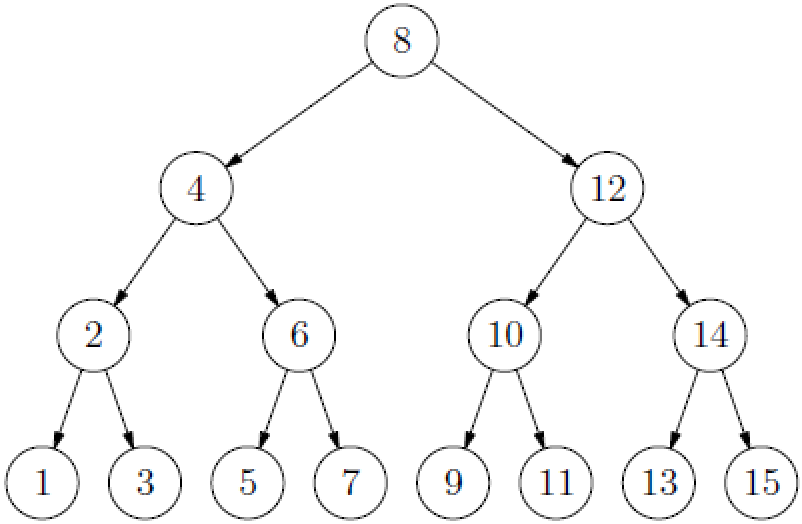
```
procedure postorderTreeWalk(x)
  if x≠NIL then
    postorderTreeWalk(left(x))
    postorderTreeWalk(right(x))
    print key(x)
  end if
end procedure
```

Binäre Suchbäume

Interessante Fragen:

- Wie viele Knoten passen in einen Binärbaum?
- Wie hoch ist der Binärbaum minimal/maximal?
- Laufzeit ist meist von der Höhe *h* des Baumes abhängig.

Binäre Suchbäume



- Abzählen der Knoten in jeder Ebene
- Verallgemeinern
- Summieren

Binäre Suchbäume

- Zählen: (s. Tafel)
- Verallgemeinern: (s. Tafel)
- Summieren:
  - Ein Binärbaum der Höhe h hat maximal

    $n(h) = 1+2+4+\ldots+2^h$ (geometrische Reihe)

    $n(h) = 2^{h+1}-1$

    Knoten.
- Man nennt diese maximale Knotenzahl auch die **Kapazität** des Baumes.

Binäre Suchbäume

- Vorhin: Anzahl der Knoten
- Nun: Höhe des Baumes

Anzahl Knoten	Min. Höhe
1	0
2	1
3	1
4	2
5	2
…	…

Binäre Suchbäume

Höhe eine Binärbaumes:

- T: Binärbaum
- n: Anzahl der Knoten von T

Satz

- T hat maximale Höhe
  - $h_{max} = n-1$
- T hat minimale Höhe
  - $h_{min} = \log_2(n+1)-1 \leq \log_2 n$ (nach oben abgerundet)

Binäre Suchbäume

Beweis des Satzes:

- h_{max}:
  - Bei n Knoten im Baum kann die maximale Höhe h_{max} nur erreicht werden, wenn er zur linearen Liste entartet. Dann ist die Pfadlänge n-1.

Binäre Suchbäume

Beweis des Satzes:

- h_{min}:
  - Der Baum hat die minimale Höhe, wenn er seine Kapazität voll ausschöpft:

$$n = 2^{h+1}-1$$

$$n+1 = 2^{h+1}$$

$$\log_2(n+1) - 1 = h$$

  - Die Höhe ist ganzzahlig →

$$h_{min} = \log_2(n+1) - 1 \leq \log_2 n \text{ (nach oben aufgerundet)}$$

Binäre Suchbäume

- Einleitung: Was sind binäre Suchbäume?
- Anfragen an binäre Suchbäume
- Einfügen und Löschen von Knoten in binären Suchbäumen

Anfragen an Binäre Suchbäume

Überblick

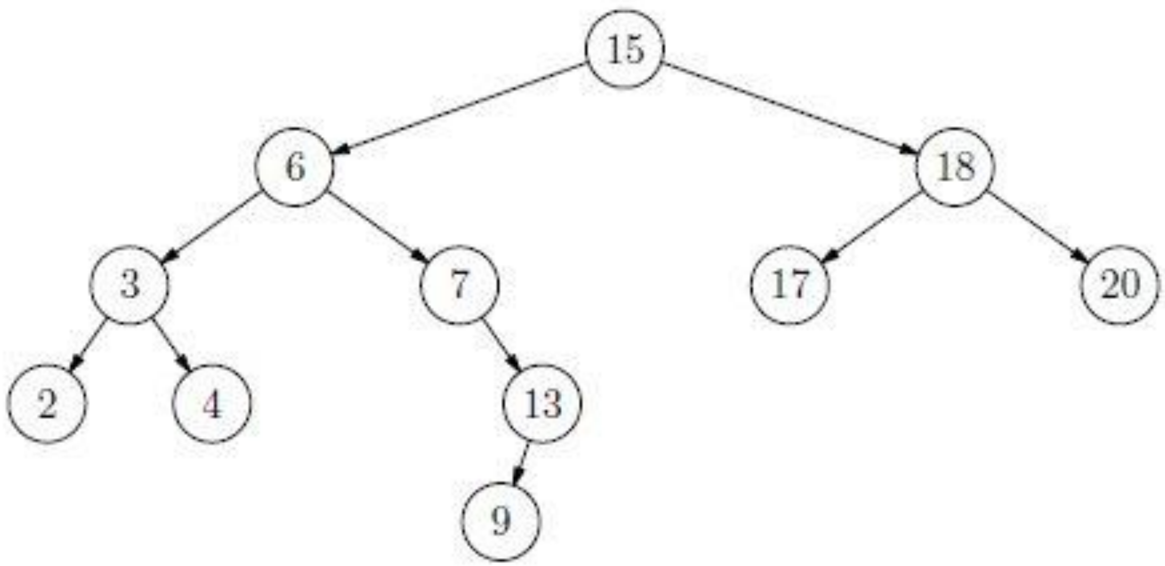
- Suchen (rekursiv, iterativ)
- Ermittlung Minimum/Maximum
- Ermittlung Vorgänger/Nachfolger

Anfragen an Binäre Suchbäume

- Einleitung: Was sind binäre Suchbäume?
- Anfragen an binäre Suchbäume
- Einfügen und Löschen von Knoten in binären Suchbäumen

Anfragen an Binäre Suchbäume

Beispiel: Suche nach Knoten mit Schlüssel 7 bzw. 8.



Anfragen an Binäre Suchbäume

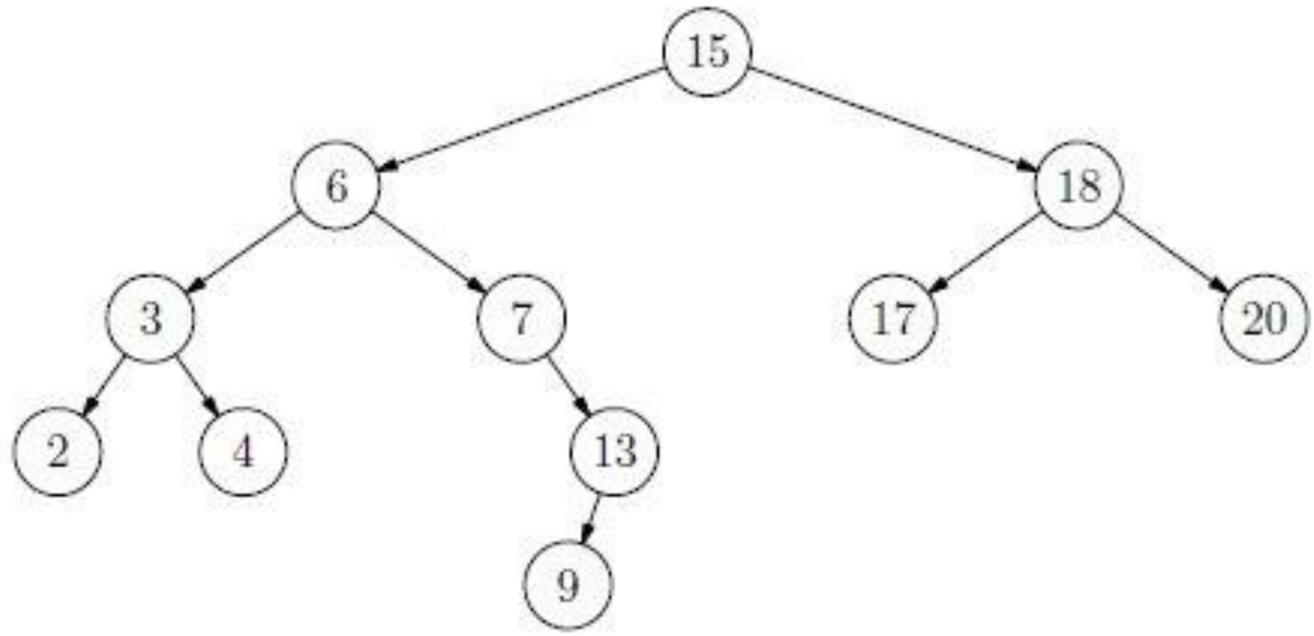
```
function treeSearch(x,k)
  if x=NIL or k=key(x) then return x
  end if
  if k<key(x) then
    return treeSearch(left(x),k)
  else
    return treeSearch(right(x),k)
  end if
end function
```

Anfragen an Binäre Suchbäume (iterativ)

```
function treeSearchIterativ(x,k)
  while x ≠ NIL and k ≠ key(x) do
    if k<key(x) then
      x = left(x)
    else
      x = right(x)
    end if
  end while
  return x
end function
```

Min./Max. in Binären Suchbäumen

- Wie finde ich das Element mit dem minimalen Schlüssel?
- Wie finde ich das Element mit dem maximalen Schlüssel?



Minimum/Maximum in BSTs

```
function treeMinimum(x)
  if x=NIL then return NIL
  end if
  while left(x) ≠ NIL do
    x = left(x)
  end while
  return x
end function
```

Voraussetzung: Eindeutige Schlüsselwerte

Minimum/Maximum in BSTs

```
function treeMaximum(x)
  if x=NIL then return NIL
  end if
  while right(x) ≠ NIL do
    x = right(x)
  end while
  return x
end function
```

Voraussetzung: Eindeutige Schlüsselwerte

Vorgänger/Nachfolger in BSTs

- Vorgänger:
  - **`pred(x)`**: Knoten y mit maximalem Schlüssel `k < key(x)`

- Nachfolger:
  - **`succ(x)`**: Knoten y mit minimalem Schlüssel `k > key(x)`

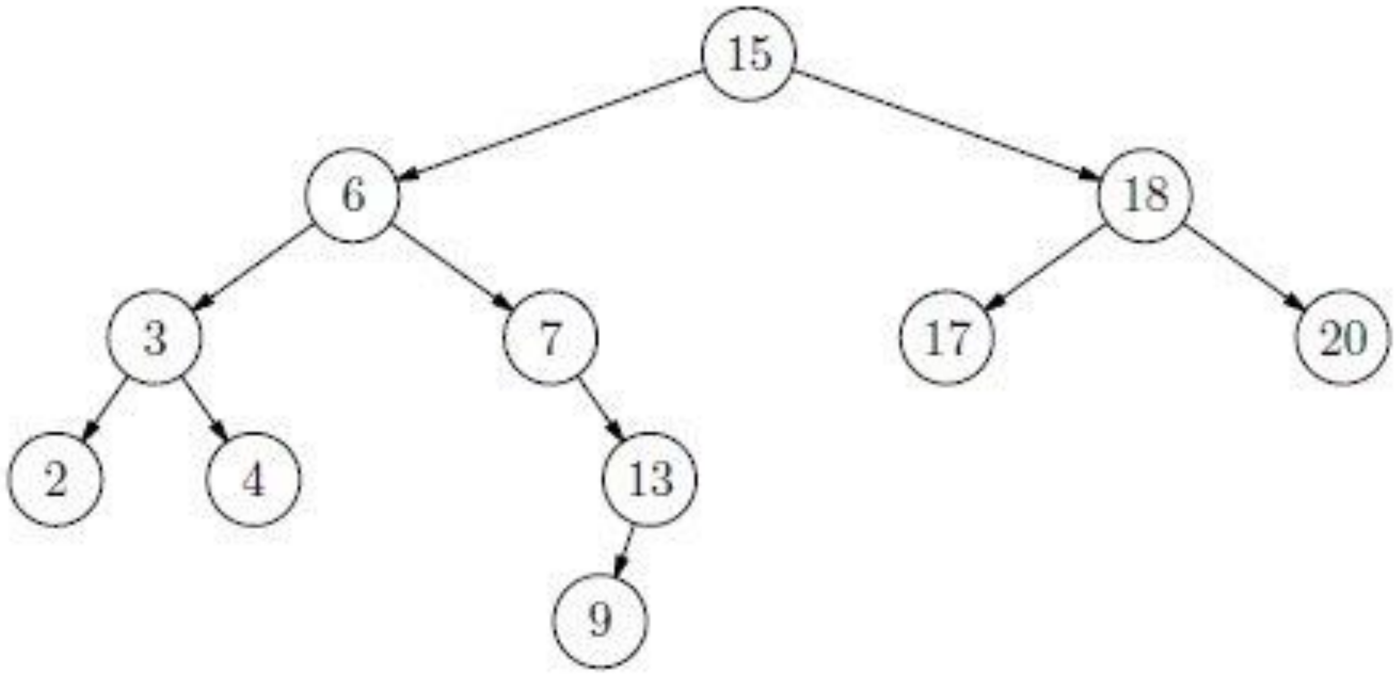
Vorgänger/Nachfolger in BSTs

Satz:

- Besitzt x ein rechtes Kind, so ist der direkte **Nachfolger** von x der Knoten mit minimalem Schlüssel im rechten Teilbaum.
- Besitzt x kein rechtes Kind, so ist der direkte **Nachfolger** von x der niedrigste Vorfahr von x, dessen linkes Kind ebenfalls Vorfahr von x ist (dabei ist ein Knoten Vorfahr von sich selbst).

Vorgänger/Nachfolger in BSTs

Nachfolger von 3, 6, 15, 13, 17, 4?



Vorgänger/Nachfolger in BSTs

```
function treeSuccessor(x)
  if x=NIL then return NIL
  end if
  if right(x) ≠ NIL then return treeMinimum(right(x))
  end if
  y = parent(x)
  while y ≠ NIL and x=right(y) do
    x = y
    y = parent(y)
  end while
  return y
end function
```

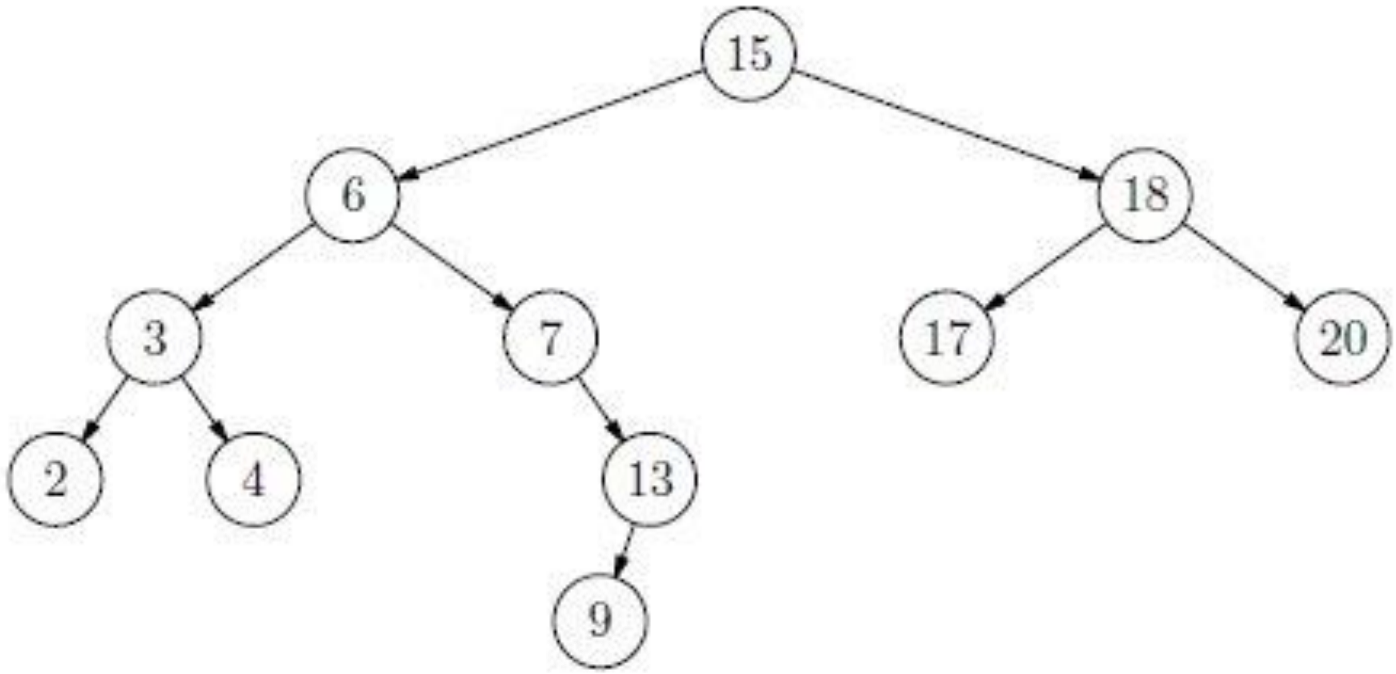
Vorgänger/Nachfolger in BSTs

Beobachtungen:

- Es sind keine Schlüsselvergleiche erforderlich, die Information steckt in der Struktur des Baumes.
- Wie viele Kinder hat *successor* (Nachfolger)?
  - 0 bzw. 1, falls x rechtes Kind hat
  - 0 bzw. 1 bzw. 2, sonst
- Annahme, dass alle Schlüssel paarweise verschieden sind
  - Falls nicht:
    - Festlegung, das *successor* der Knoten ist, der durch den Algorithmus zurückgeliefert wird.

Vorgänger/Nachfolger in BSTs

Vorgänger von 3, 6, 15, 7, 9, 2?



Vorgänger/Nachfolger in BSTs

- Vorgänger: Symmetrisch zum Nachfolger:
- Fall x linken Teilbaum hat:
  - *predecessor* ist das Maximum im linken Teilbaum (der am weitesten rechts stehende Knoten in dem Teilbaum)
- Falls x keinen linken Teilbaum hat:
  - Vorgänger muss oberhalb von x liegen.
  - Falls kein Knoten oberhalb: x hat keinen Vorgänger
  - Sonst: *predecssor(x)* ist der tiefste Knoten von x, dessen rechtes Kind x ist oder er einen Pfad zu x hat.
    - Falls es keinen solchen Knoten gibt: x hat keinen Vorgänger.

Vorgänger/Nachfolger in BSTs

Beobachtungen *predecessor*:

- Auch hier: Es sind keine Schlüsselvergleiche erforderlich, die Information steckt in der Struktur des Baumes.
- Auch hier: Annahme, dass alle Schlüssel paarweise verschieden sind
  - Falls nicht:
    - Festlegung, das *predecessor(x)* der Knoten ist, der durch den Algorithmus zurückgeliefert wird.

Vorgänger/Nachfolger in BSTs

```
function treePredeccessor(x)
  if x=NIL then return NIL
  end if
  if left(x) ≠ NIL then return treeMaximum(right(x))
  end if
  y = parent(x)
  while y ≠ NIL and x=left(y) do
    x = y
    y = parent(y)
  end while
  return y
end function
```

Binäre Suchbäume

Laufzeitanalyse:

- Algorithmen „verfolgen“ den Baum
  - Minimum/Maximum: abwärts
  - Vorgänger/Nachfolger: aufwärts
- Laufzeit von Höhe h abhängig:
  - $T(n) = O(h) = O(\log(n))$

Binäre Suchbäume

- Einleitung: Was sind binäre Suchbäume?
- Anfragen an binäre Suchbäume
- Einfügen und Löschen von Knoten in binären Suchbäumen

Binäre Suchbäume

- Modifizierende Operationen:
  - Suchbaumeigenschaft muss erhalten bleiben
  - **Einfügen**
    - Einfach, wenn „unten“ angefügt wird
  - **Löschen**
    - Voraussichtlich etwas schwieriger

Binäre Suchbäume

Prinzip Einfügen:

- Neuen Knoten z mit leeren Kindern erzeugen
- Wenn Baum leer:
  - Diesen Knoten z als Wurzelknoten nutzen
- Sonst:
  - je nach Schlüsselwert den Baum sukzessive nach links oder rechts im Baum traversieren
  - Knoten z an bisheriges Blatt anhängen

Binäre Suchbäume

```
function treeInsert(root, z)
  parent(z) = right(z) = left(z) = NIL
  if root=NIL then return z
  end if
  x=root
  while x ≠ NIL do
    y = x    // möglicher Elternknoten
    if key(z) < key(y) then x = left(x)
    else x = right(x)
   end if
  end while
```

Binäre Suchbäume

```
  parent(z) = y
  if key(z) < key(y) then
        left(y) = z
     else
     right(y) = z
  end if
  return root
end function
```

Binäre Suchbäume

Anmerkungen:

- Einfüge-Reihenfolge bestimmt Struktur des Baumes.
- So entstehende Bäume heißen „natürlich“.
- Günstigster/ungünstigster Fall:
  - Günstig: Vollständiger Baum → T(n) = O(log n)
  - Ungünstig: Lineare Liste → T(n) = O(n)
- Wichtige Fragen:
  - Was liegt dazwischen und wie häufig kommen welche Varianten vor?
  - Was ist der „mittlere“ Fall?
    → Die erwartete Höhe eines natürlichen BST mit n Knoten ist O(log n).

Binäre Suchbäume

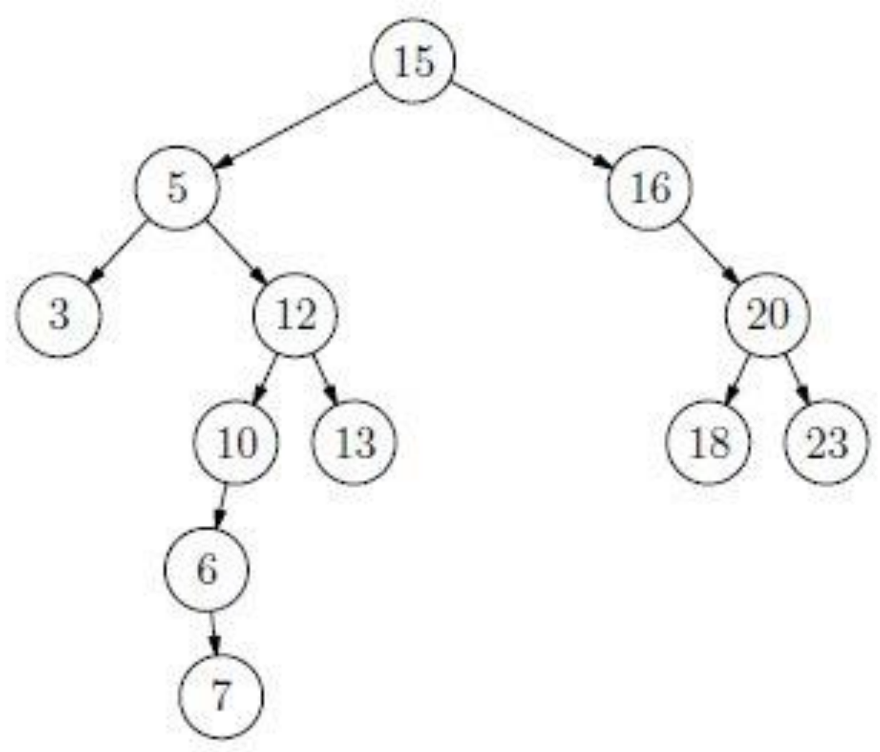
Einfügen – Beispiele:

3, 2, 1, 5, 4, 6

1, 2, 3, 4, 5, 6

Binäre Suchbäume

- Löschen von Knoten z mit
  - key(z)=13, key(z)=16, key(z)=5



Binäre Suchbäume

Löschen hängt ab von der Anzahl der Kinder:

- 0 Kinder (Blattknoten) → direkt entfernen
- 1 Kind → Herauslösen
- 2 Kinder →
  - Ersetzen durch Vorgänger, oder
  - Ersetzen durch Nachfolger

Binäre Suchbäume

Löschen – Funktionsweise

- Hat der zu löschende Knoten z max. ein Kind →
  - Herausnehmen
  - Ggf. bekommt Kind von z neuen Elternknoten
- Sonst (der Nachfolger rückt hoch):
  - Nachfolger (nächst höherer Schlüssel) bestimmen
    (Nachfolger hat max. ein Kind)
  - Nachfolger aus Baum herausnehmen
    (Korrektur der Kindschaftsverhältnisse)
  - Nachfolger ersetzt den zu löschenden Knoten

Binäre Suchbäume

```
function treeDelete(root,z)
  // Bestimme Knoten, der entfernt wird:

  if left(z) = NIL or right(z) = NIL then
    delNode = z
  else
    delNode=treeSuccessor(z)
  end if
```

Binäre Suchbäume

```
if left(delNode) ≠ NIL then x = left(delNode)
  else x = right(delNode)
end if
// x ist Kind von delNode

if x ≠ NIL then
  parent(x) = parent(delNode)
  // Kind von delNode bekommt neuen Elternknoten
end if
```

Binäre Suchbäume

```
if parent(delNode) = NIL then
    // falls delNode keinen Elternknoten hat,
   // hat auch x keinen
   root = x
else
    if delNode=left(parent(delNode)) then
       left(parent(delNode)) = x
    else
       right(parent(delNode)) = x
    end if
end if
```

Binäre Suchbäume

```
  if z ≠ NIL then
        key(z) = key(delNode)
        Satellitendaten(z) = Satellitendaten(delNode)
        // ggf. Aufräumarbeiten, bspw. left(z) = NIL
    end if
   return root
end function
```

Laufzeit?

Zusammenfassung BST

- Alle Wörterbuchoperationen konnten effizient realisiert werden:
  - Ungünstigste Laufzeit: O(n)
  - Günstigste Laufzeit: O(log n)
- Im allgemeinen Fall wissen wir über die Höhe des Baumes (Höhe) in der Regel wenig.
- Man kann zeigen, dass die erwartete Höhe eines BSTs im mittleren Fall O(log n) beträgt.
- Hinzufügen und Löschen kann im Verlauf des Lebenszeit des Baumes zu ungünstigen, unbalancierten Bäumen führen.

- Deshalb nun: Variante von BST-Bäumen, die einigermaßen balanciert sind.