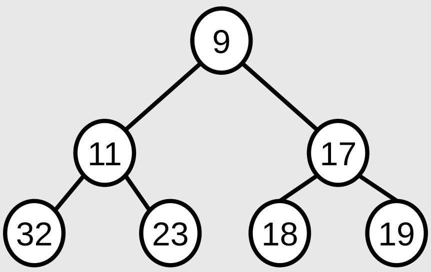
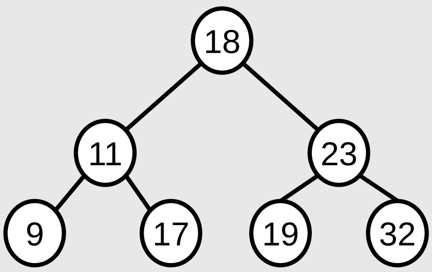
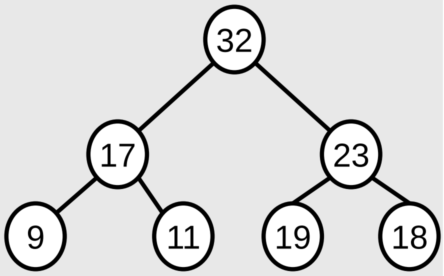
Binärer Suchbaum vs. Heap

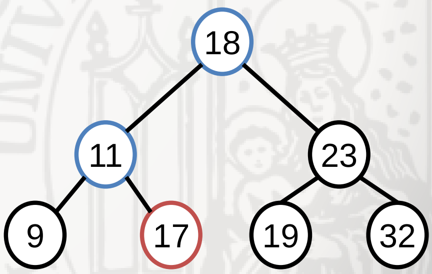
- Ein binärer Suchbaum und ein Heap unterscheiden sich durch ihre strukturellen Invarianten
  - Für Knoten v_i im linken Teilbaum und Knoten v_j im rechten Teilbaum von v_k gilt:

Min-Heap	Binärer Suchbaum	Max-Heap
$value(v_k) \leq value(v_i)$ $value(v_k) \leq value(v_j)$	$value(v_i) \leq value(v_k)$ $value(v_k) \leq value(v_j)$	$value(v_i) \leq value(v_k)$ $value(v_j) \leq value(v_k)$
		

Binärer Suchbaum: Suche

- Idee: Rekursive Erkundung eines Pfades
  - Suche nach Schlüssel s beginnt an der Wurzel $v = root$
  - Falls der aktuelle Knoten v Schlüssel s enthält → s gefunden!
  - Falls nicht:
    - v ist Blatt → s nicht enthalten!
    - Schlüssel ist kleiner als $value(v)$ → Suche im linken Teilbaum
    - Schlüssel ist größer als $value(v)$ → Suche im rechten Teilbaum

- Beispiel: Suche nach 17



Binäre Suchbäume: Implementierung Knoten

- Basierend auf Binärbäumen
- Schlüsselwert muss vergleichbar sein
- Jeder Knoten ist ein Binärbaum mit evtl. Subbäumen.

```
class node():
    def __init__(self, key = None, value = None):
        self.key = key
        self.value = value
        self.left = None
        self.right = None
```


Binäre Suchbäume: Suche nach Schlüssel

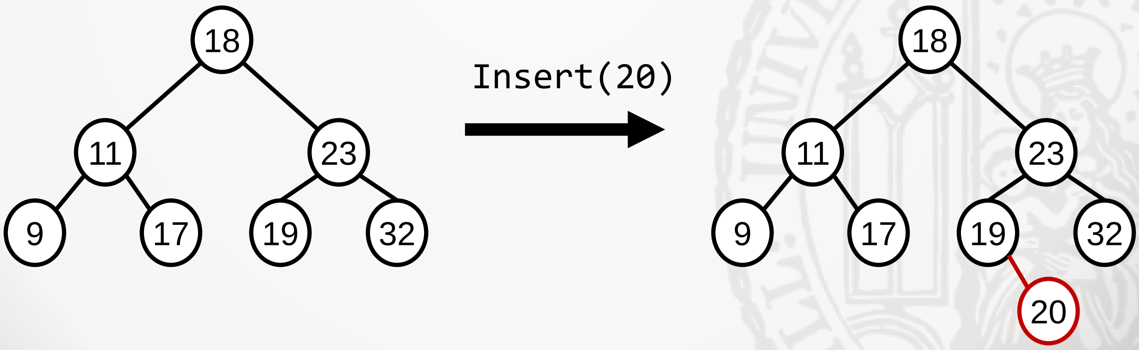
```
def search(self, key):
  if key == self.key:
    return self.value

  elif key < self.key and self.left != None:
    return self.left.search(key)

  elif key > self.key and self.right != None:
    return self.right.search(key)
```

Binäre Suchbäume: Einfügen Schlüssel+Objekt

- Intuition für *insert(key, value)*
  - Suche den Schlüssel *key* im Baum.
  - Falls *key* schon im Baum existiert, ersetze das vorherige Objekt.
  - Falls nicht, hält die Suche in einem Blatt oder innerem Knoten mit max. einem Nachfolger.
  - Füge einen neuen Knoten mit *(key, value)* als linker oder rechter Folgeknoten bei diesem Knoten ein.



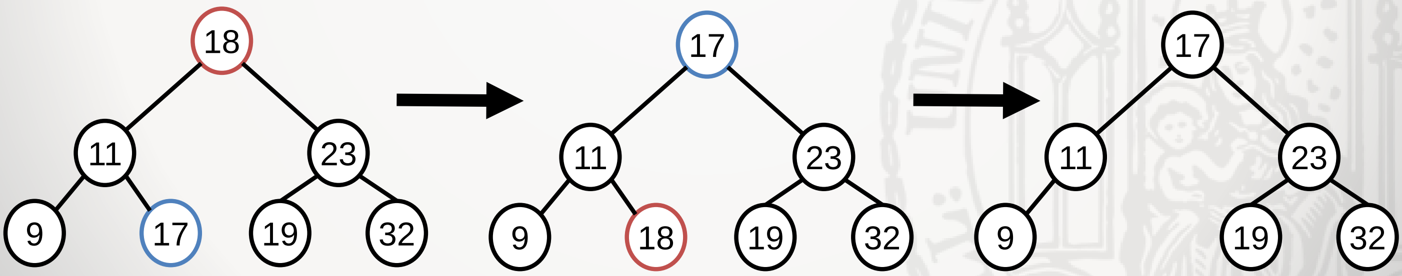
Binäre Suchbäume: Einfügen Schlüssel+Objekt

```
def insert(self, key, value):
  if self.key == None:
    self.key, self.value = key, value
  elif key == self.key:
    self.value = value
  elif key < self.key:
    if self.left == None:
      self.left = Node(key, value)
    else:
      self.left.insert(key, value)
  else:
    if self.right == None:
      self.right = Node(key, value)
    else:
      self.right.insert(key, value)
```

Binäre Suchbäume: Löschen

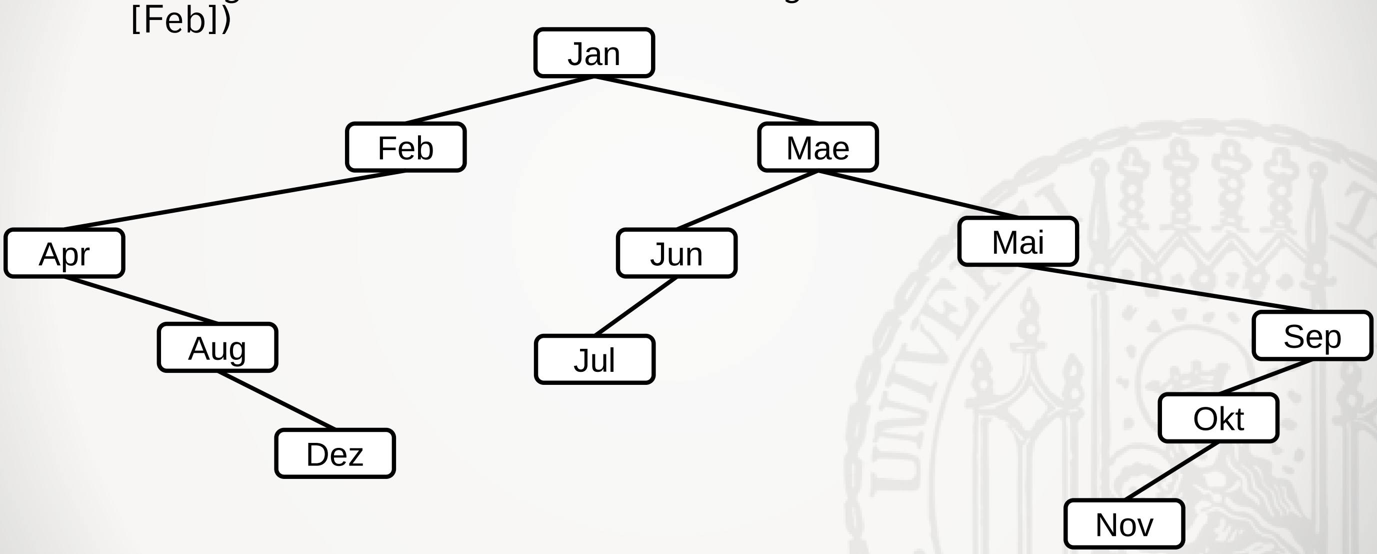
- Intuition für *remove(key)*
  - Suche den Knoten v mit Schlüssel *key* im Baum.
  - Falls *key* nicht im Baum existiert, passiert nichts.
  - Falls v ein Blatt ist, lösche den Zeiger darauf.
  - Falls v ein innerer Knoten ist, finde rechtesten Knoten v' (größten Schlüssel) im linken Teilbaum von v. Tausche v mit v'. Lösche dann den Blattknoten v.

`Remove(18)`



Suchbäume für lexikografische Schlüssel

- Beispiel: Deutsche Monatsnamen
  - Sortierung lexikographisch
  - Einfügen in kalendarischer Reihenfolge (nicht mehr ausbalanciert [Feb])



  - Ausgabe durch InOrder-Traversierung (siehe Kap. 1):
  - Apr - Aug - Dez - Feb - Jan - Jul - Jun - Mae - Mai - Nov - Okt - Sep

Binäre Suchbäume: Komplexität

- Analyse der Laufzeit Insert und Remove
  - Suchen der entsprechenden Position im Baum.
  - Lokale Änderungen im Baum in $O(1)$.

- Analyse des Suchverfahrens
  - Anzahl Vergleiche entspricht maximale Pfadtiefe des Baumes
  - Sei $h(t)$ die Höhe des Baumes t, dann ist die Komplexität der Suche $O\big(h(t)\big)$.
  - Wir wissen: Hat t genau n Knoten, dann gilt:
    $$h + 1 \;\leq\; n \;\leq\; 2^{h+1} - 1$$
  - Damit gilt im Worst-Case Komplexität $O(n)$ und im Best-Case $O(\log n)$.