# Algorithmen und Datenstrukturen

## B7. Binäre Suchbäume

Marcel Lüthi and Gabriele Röger

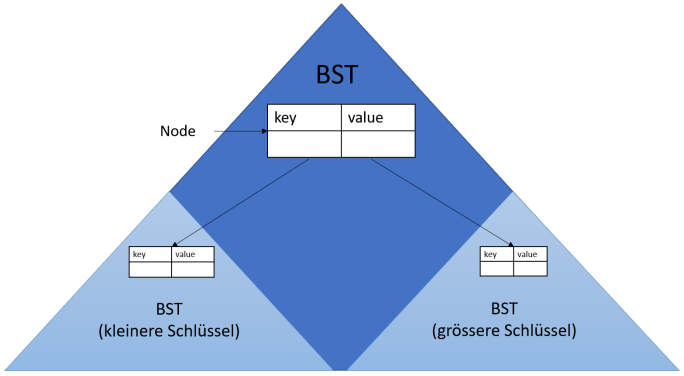
Universität Basel

# Binäre Suchbäume

Ein binärer Suchbaum ist ein Binärbaum mit symmetrischer Ordnung

## Ein Binärbaum ist

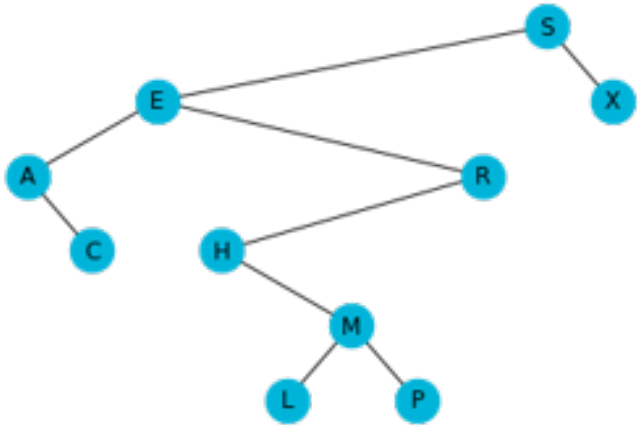
- der leere Baum, oder
- eine Wurzel mit einem linken und einem rechten Unterbaum




## Symmetrische Ordnung

Der Schlüssel jedes Knotens ist

- grösser als alle Schlüssel im linken Unterbaum
- kleiner als alle Schlüssel im rechten Unterbaum



# Repräsentation in Code (mit Zähler)

```
class Node[Key, Value]:
    key : Key  # Key unterstuetzt Ordnungsrelation
    value : Value
    left : Node[Key, Value]
    right : Node[Key, Value]
    count : Int
```

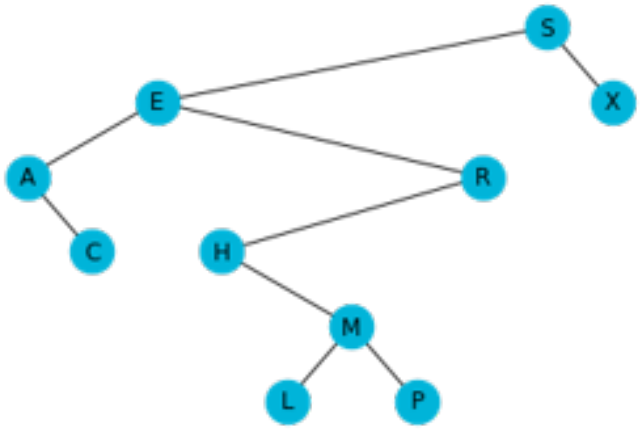
- Feld `count` zählt die Anzahl Knoten im Unterbaum
- Erlaubt effiziente Implementation von Operation `size`
  - Kein Traversieren vom Baum nötig.

# Suche in binären Suchbaum

Um get zu implementieren, müssen wir effizient suchen können.

Suche nach Schlüssel $k$:

- Fall 1: $k <$ Schlüssel in Knoten
  - Gehe nach links
- Fall 2: $k >$ Schlüssel in Knoten
  - Gehe nach rechts
- Fall 3: $k =$ Schlüssel in Knoten
  - Gefunden



# Suche in Binärbaum

Die Suche, ausgehend von Knoten `root` kann einfach rekursiv implementiert werden.

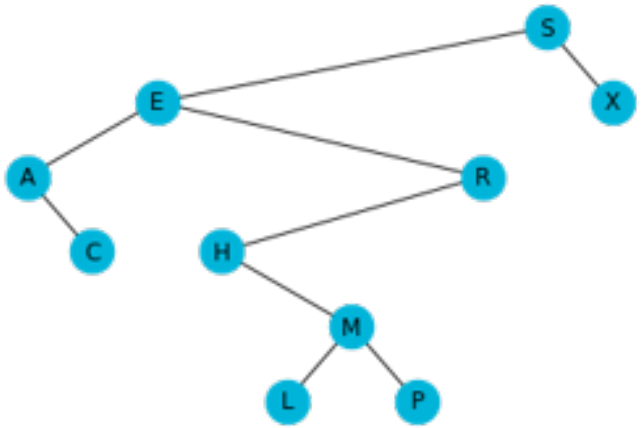
- Suche wird einfach in "richtigem" Teilbaum fortgesetzt.

```
def get(key, root):
    if root == None:
        return None
    elif key < root.key:
        return get(key, root.left)
    elif key > root.key:
        return get(key, root.right)
    elif key == root.key:
        return root.value
```

# Einfügen in binären Suchbaum

put lässt sich fast so einfach wie get implementieren.

- Suche nach Schlüssel.
  - Schlüssel gefunden → Wert neu setzen
  - Schlüssel nicht in Baum → Neuen Knoten hinzufügen.



# Einfügen in binären Suchbaum

Rekursive Implementation:

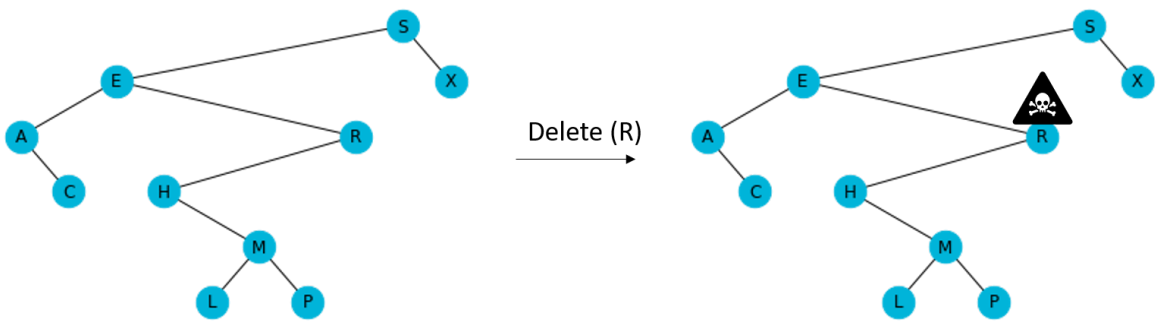
```
def put(key, value, root):
    if (root == None):
        return Node(key, value, count = 1)
    elif key < root.key:
        root.left = put(key, value, root.left)
    elif key > root.key:
        root.right = put(key, value, root.right)
    elif key == root.key:
        root.value = value
    root.count = 1 + size(root.left) + size(root.right)
    return root
```

- Auf dem "Rückweg" wird der Zähler für die Anzahl Knoten im Unterbaum aktualisiert.
- Beachte: Teilbaum wird in jeder Rekursion neu gesetzt.

# Löschen von Knoten: Einfache Methode

Einfachste Methode zum Löschen: Tombstone

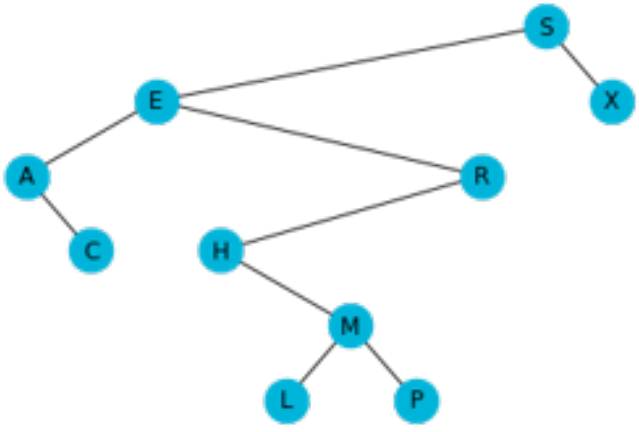
- Finde Knoten
- Markiere diesen als gelöscht
  - z.B. Wert auf `null` setzen
- Schlüssel bleibt im Baum




Problem: Speicherverschwendung bei vielen gelöschten Elementen.

# Löschen von minimalem Schlüssel

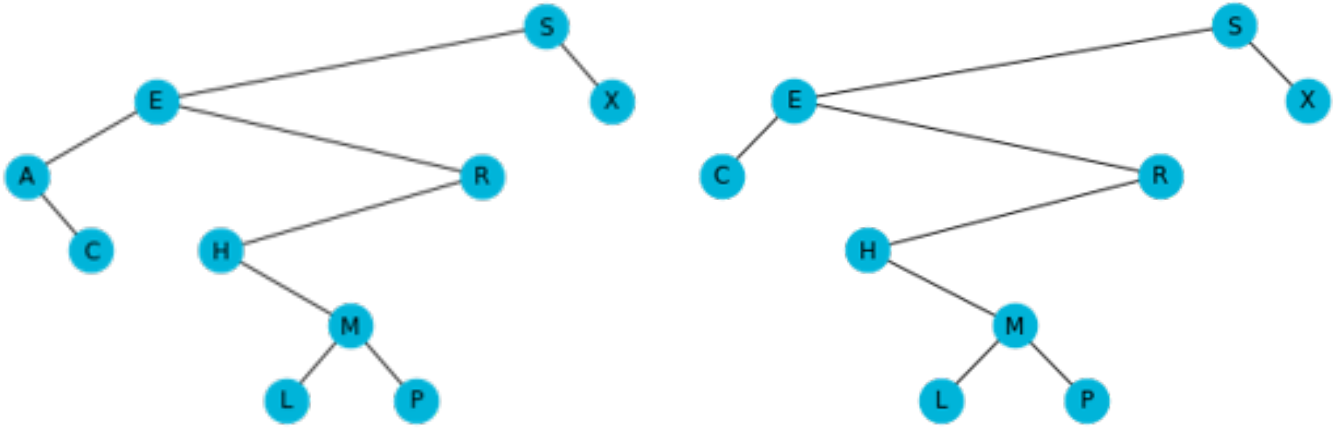
- Nach Links bis linker Knoten null ist
- Diesen Knoten durch rechten Knoten ersetzten
- Knotenzähler neu setzen




```
def deleteMin(root):
  if root.left == None:
    return root.right
  else:
    root.left = deleteMin(x.left);
    root.count = 1 + size(root.left)    + size(root.right);
    return root
```

# Löschen von minimalem Schlüssel

- Nach Links bis linker Knoten null ist
- Diesen Knoten durch rechten Knoten ersetzten
- Knotenzähler neu setzen

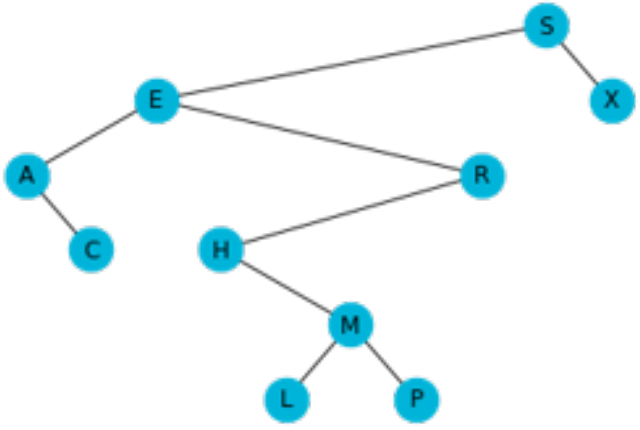



```
def deleteMin(root):
  if root.left == None:
    return root.right
  else:
    root.left = deleteMin(x.left);
    root.count = 1 + size(root.left)    + size(root.right);
    return root
```

# Löschen nach Hibbard

- Knoten $t$ mit zu löschendem Schlüssel suchen.

Fall 1: Keine Kinder

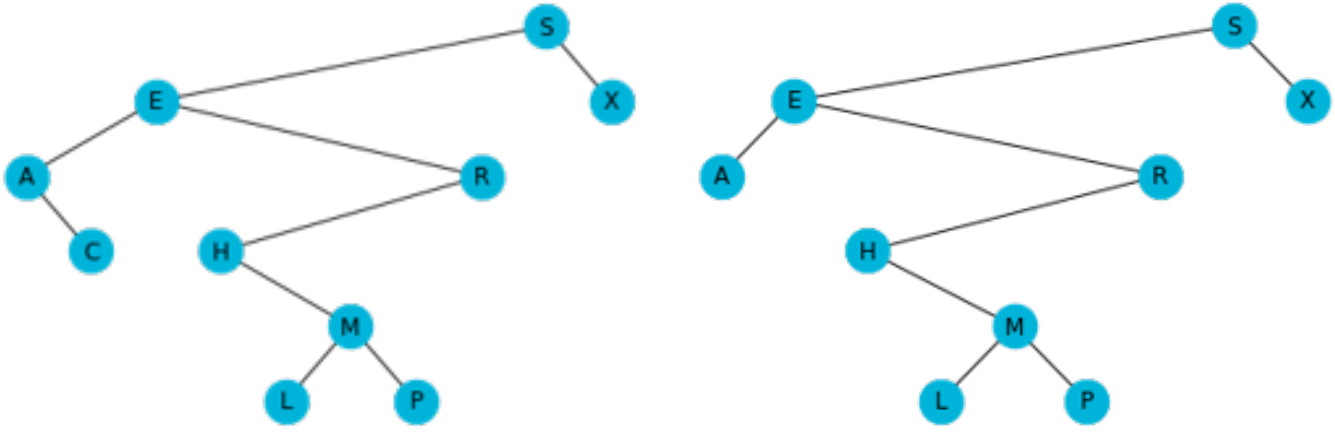



- Entsprechendes Kind von Elternknoten von $t$ auf null setzen.

# Löschen nach Hibbard

- Knoten $t$ mit zu löschendem Schlüssel suchen.

Fall 1: Keine Kinder

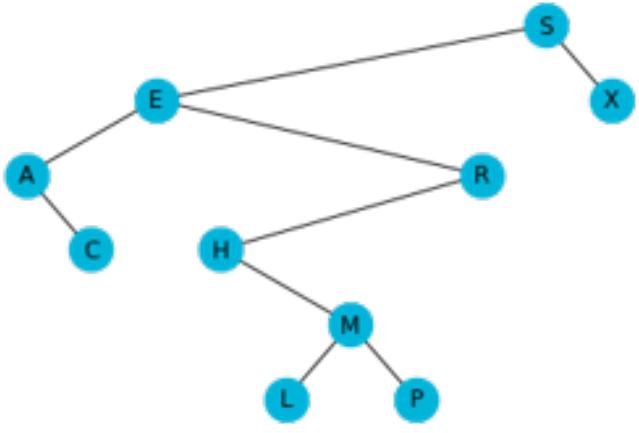



- Entsprechendes Kind von Elternknoten von $t$ auf null setzen.

# Löschen nach Hibbard

- Knoten $t$ mit zu löschendem Schlüssel suchen.

Fall 2: 1 Kind

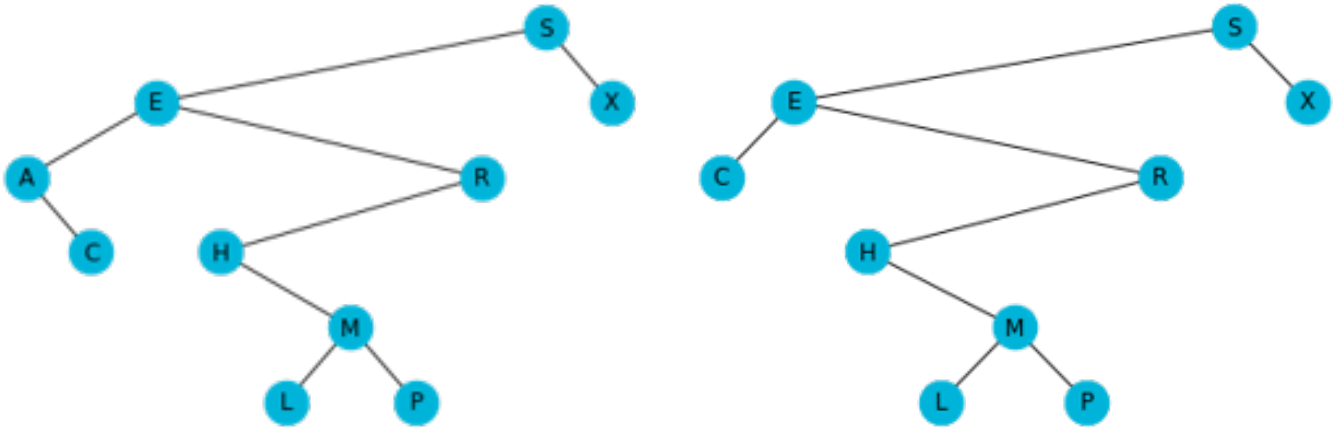



- Parent von Kind auf Parent von $t$ setzen

# Löschen nach Hibbard

- Knoten $t$ mit zu löschendem Schlüssel suchen.

Fall 2: 1 Kind

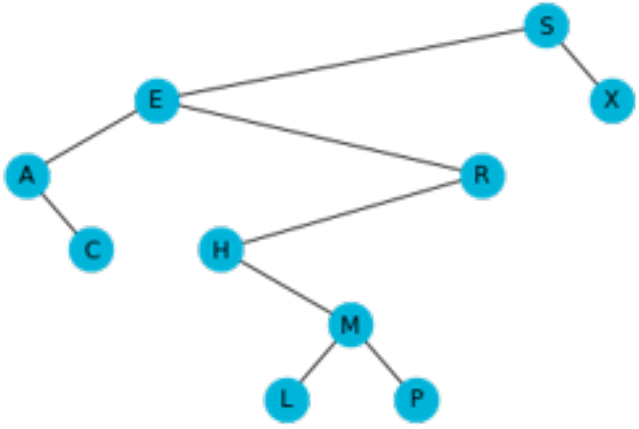



- Parent von Kind auf Parent von $t$ setzen

# Löschen nach Hibbard

- Knoten $t$ mit zu löschendem Schlüssel suchen.

Fall 3: 2 Kinder

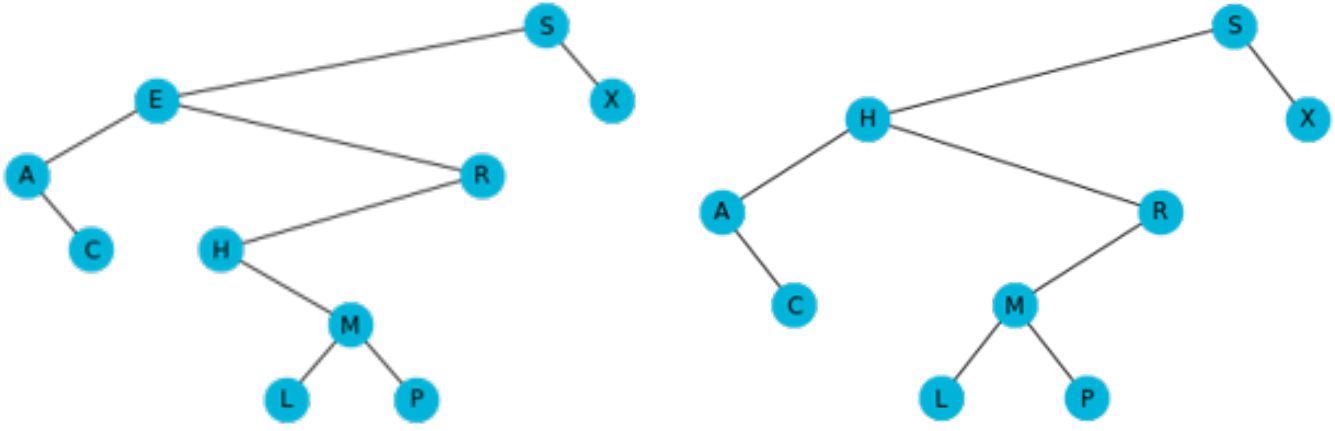



- Kleinster Knoten $x$ im rechten Unterbaum von $t$ suchen
- Kleinster Knoten im Unterbaum löschen (`deleteMin`)
- $x$ anstelle von t setzten

# Löschen nach Hibbard

- Knoten $t$ mit zu löschendem Schlüssel suchen.

Fall 3: 2 Kinder




- Kleinster Knoten $x$ im rechten Unterbaum von $t$ suchen
- Kleinster Knoten im Unterbaum löschen (`deleteMin`)
- $x$ anstelle von t setzten

# Löschen nach Hibbard: Anmerkungen

Warum wird Knoten durch Nachfolger und nicht Vorgänger ersetzt?

- Entscheidung willkürlich und unsymmetrisch.
- Asymmetrie führt zu Performanceeinbussen

Einfache Verbesserung: Manchmal Vorgänger und manchmal Nachfolger verwenden.