Algorithmen und Datenstrukturen

B6. Binäre Suchbäume[1]

Marcel Lüthi and Gabriele Röger

Universität Basel

11. April 2018

[1] Folien basieren auf Vorlesungsfolien von Sedgewick & Wayne
https://algs4.cs.princeton.edu/lectures/32BinarySearchTrees-2x2.pdf

Introduction

Programm

- Binäre Suchbäume
- Analyse / Experimente
- Implementation von ordnungsbasierten Operationen
- Löschen von Knoten

Informatiker des Tages : Niklaus Wirth

Niklaus Wirth

- Schweizer Informatiker
  - Professor an der ETH
- Gewinner Turing Award (1984)
  - Arbeit an Programmiersprachen
- Entwickelte u.a. die Sprachen Pascal, Modula und Oberon
- Autor eines klassischen Buchs zu Algorithmen und Datenstrukturen

Wirth, Niklaus. Algorithms + Data Structures = Programs, Prentice-Hall Series in Automatic Computation. Prentice Hall, 1976.

Einfache Symboltabellen: Komplexität

	Worst-case			Average-case		
Implementation	suchen	einfügen	löschen	suchen (hit)	einfügen	löschen
Verkettete Liste	N	N	N	N/2	N	N/2
Binäre suche	$\log_2(N)$	N	N	$\log_2(N)$	$N/2$	N

Ziel

Entwickle Datenstruktur die effizientes Suchen und Einfügen erlaubt.

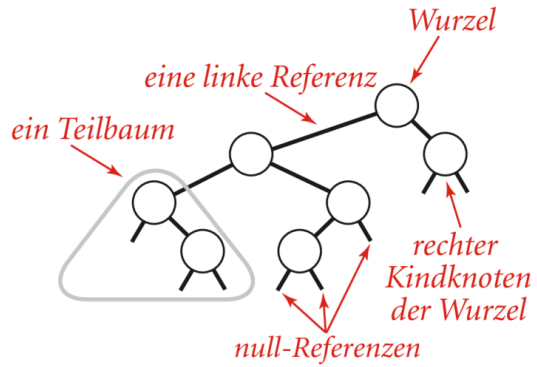
Binäre Suchbäume

Binäre Suchbäume

Ein Binärer Suchbaum ist ein Binärbaum mit symmetrischer Ordnung

Ein Binärbaum ist

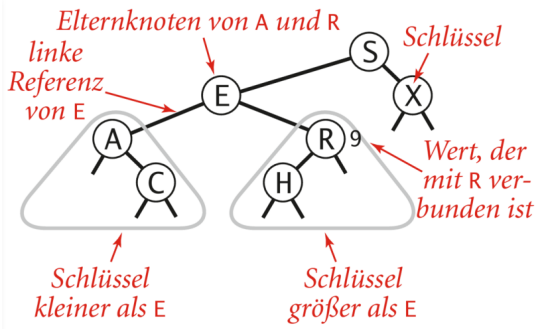
- der leere Baum, oder
- eine Wurzel mit einem linken und einem rechten Unterbaum



Symetrische Ordnung

Der Schlüssel jedes Knotens ist

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



Quelle: Abb. 3.8 / 3.9, Algorithmen, Wayne & Sedgewick

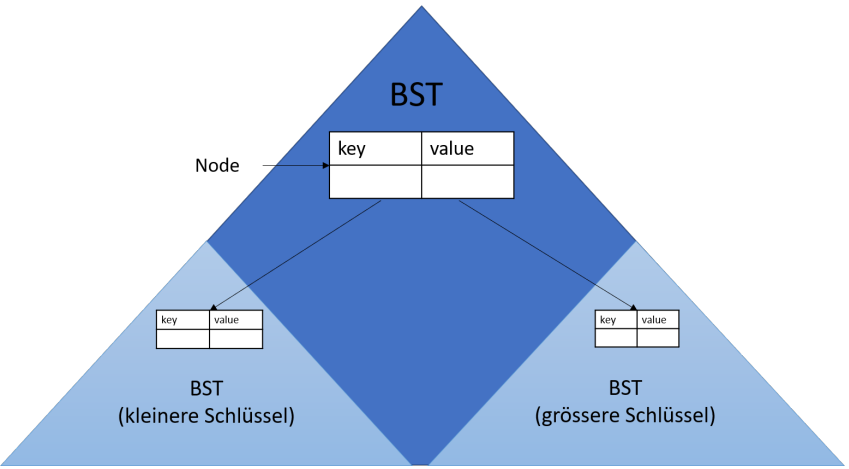
Binäre Suchbäume und Symboltabellen

Gut geeignet um eine Symboltabelle zu implementieren.

```
class ST[Key, Value]:

    ST()

    def put(key : Key, value : Value) -> None
    def get(key : Key) -> Value
    def contains(key : Key) -> Boolean
    def delete(key : Key) -> None
    def isEmpty() -> Boolean
    def size() -> Int
    def keys() : Iterator[Key]
```

Repräsentation in Code

- Implementation BST: Referenz zu Wurzel Knoten




```
class Node[Key, Value]:
    # Auf Key muss Ordnungsrelation
    # definiert sein

    Node(key : Key, value : Value)

    key : Key
    value : Value
    left : Node[Key, Value]
    right : Node[Key, Value]
```

Repräsentation in Code (mit Zähler)

- Implementation BST: Referenz zu Wurzel Knoten

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

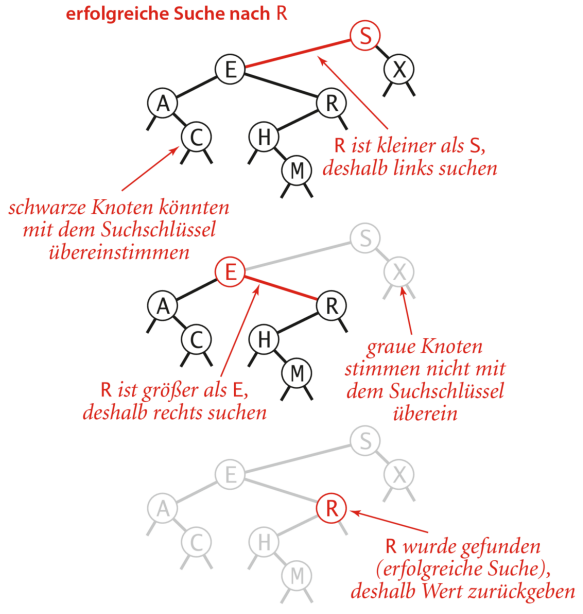
```
class Node[Key, Value]:
    # Auf Key muss Ordnungsrelation
    # definiert sein

    Node(key : Key, value : Value)

    key : Key
    value : Value
    left : Node[Key, Value]
    right : Node[Key, Value]
    count : Int
```

Suche in Binärbaum

- Um get zu implementieren, müssen wir effizient suchen können.

Suche nach Schlüssel k: Prinzip:

- 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



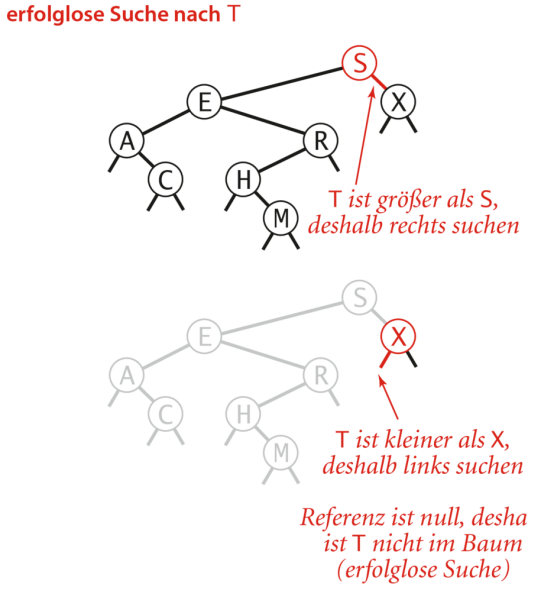
Quelle: Abb. 3.11, Algorithmen, Wayne & Sedgewick

Suche in Binärbaum

- Um get zu implementieren, müssen wir effizient suchen können.

Suche nach Schlüssel k: Prinzip:

- 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



Quelle: Abb. 3.11, Algorithmen, Wayne & Sedgewick

Suche in Binärbaum

- Die Suche, ausgehend von Knoten `root` kann einfach rekursiv implementiert werden.
  - Suche wird einfach in "richtigem" Teilbaum fortgesetzt.

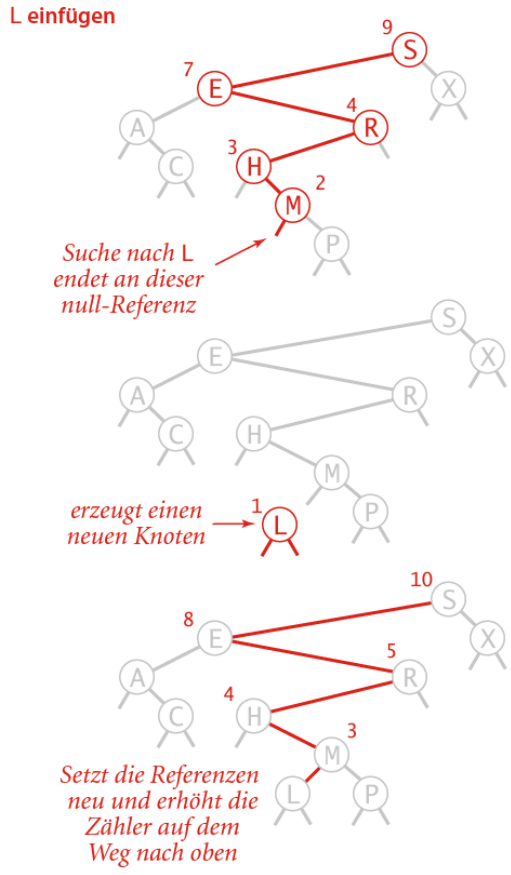
```
def get(key, root):
    if root == None:
        return None
    elif key < root.key:
        return get(key, root.left)
    elif key > root.key:
        return get(key, root.right)
    elif key == root.key:
        return root.value
```

Einfügen in Binärbaum

- put lässt sich fast so einfach wie get implementieren.

Suche nach Schlüssel.
Zwei Fälle:

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



Quelle: Abb. 3.12, Algorithmen, Wayne & Sedgewick

Einfügen in Binärbaum

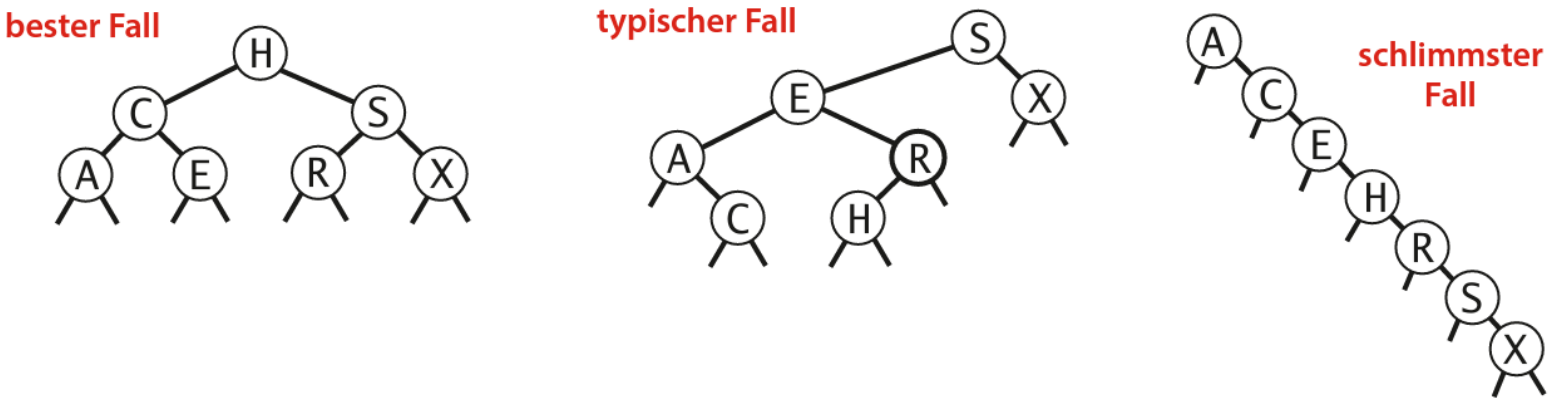
- Die operation `put` ausgehen von Knoten `root` kann einfach rekursiv implementiert werden.
  - Auf dem "Rückweg" wird der Zähler für die Anzahl Knoten im Unterbaum aktualisiert.
- Beachte: Teilbaum wird in jeder Rekursion neu gesetzt.

```
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
```

Analyse

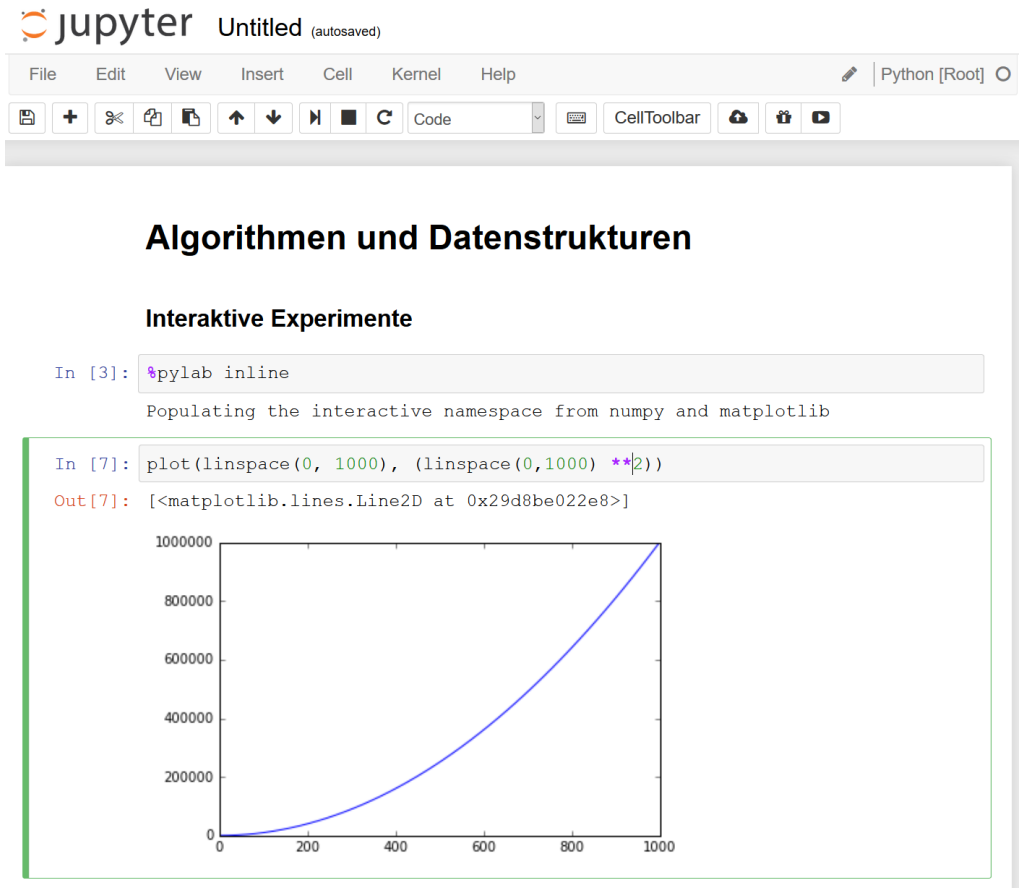
Ausprägung des Binärbaums

- Selbe Menge von Schlüsseln führt zu verschiedene Bäumen
  - hängt von Einfügereihenfolge ab.



Quelle: Abb. 3.14, Algorithmen, Wayne & Sedgewick

Implementation und Beispielanwendung



IPython Notebooks: BST.ipynb

Sortieralgorithmus

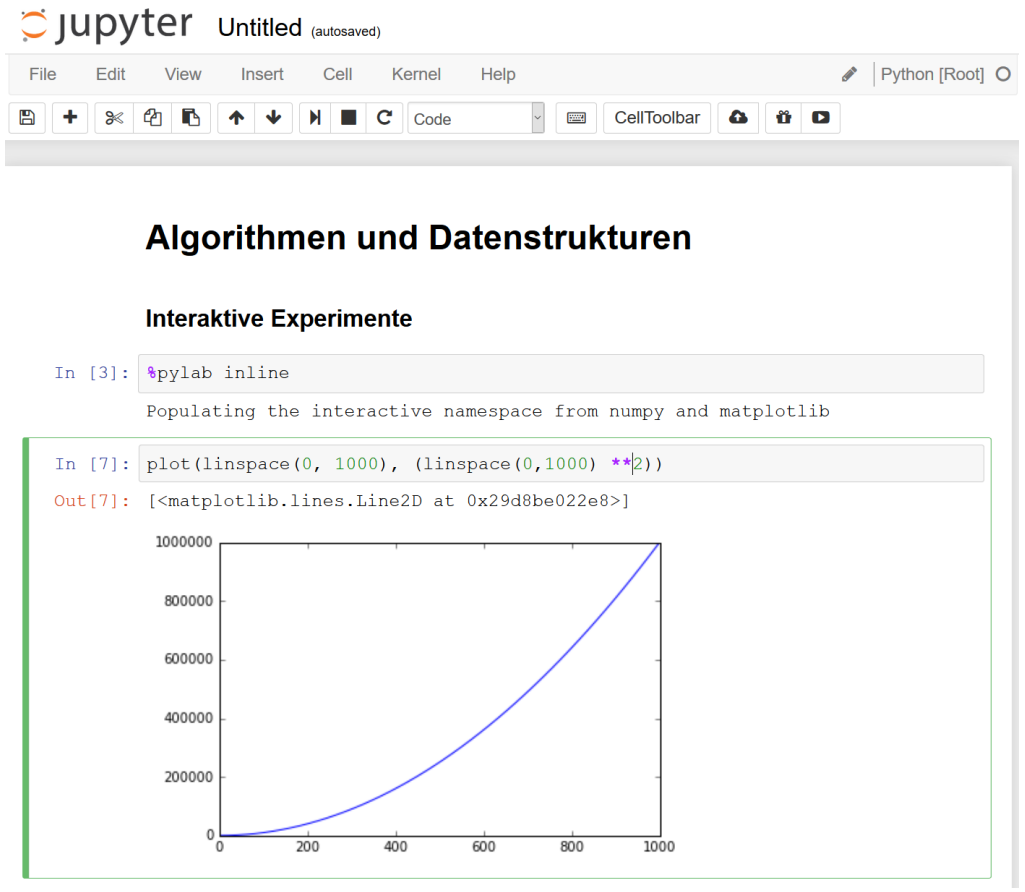
- Annahme: Keine doppelten Schlüssel!

Sortieralgorithmus

1. Permutiere zufällige alle Schlüssel
2. Füge Schlüssel in Binären Suchbaum ein

- Was passiert wenn wir doppelte Schlüssel haben?
- Welchem Sortieralgorithmus entspricht das?

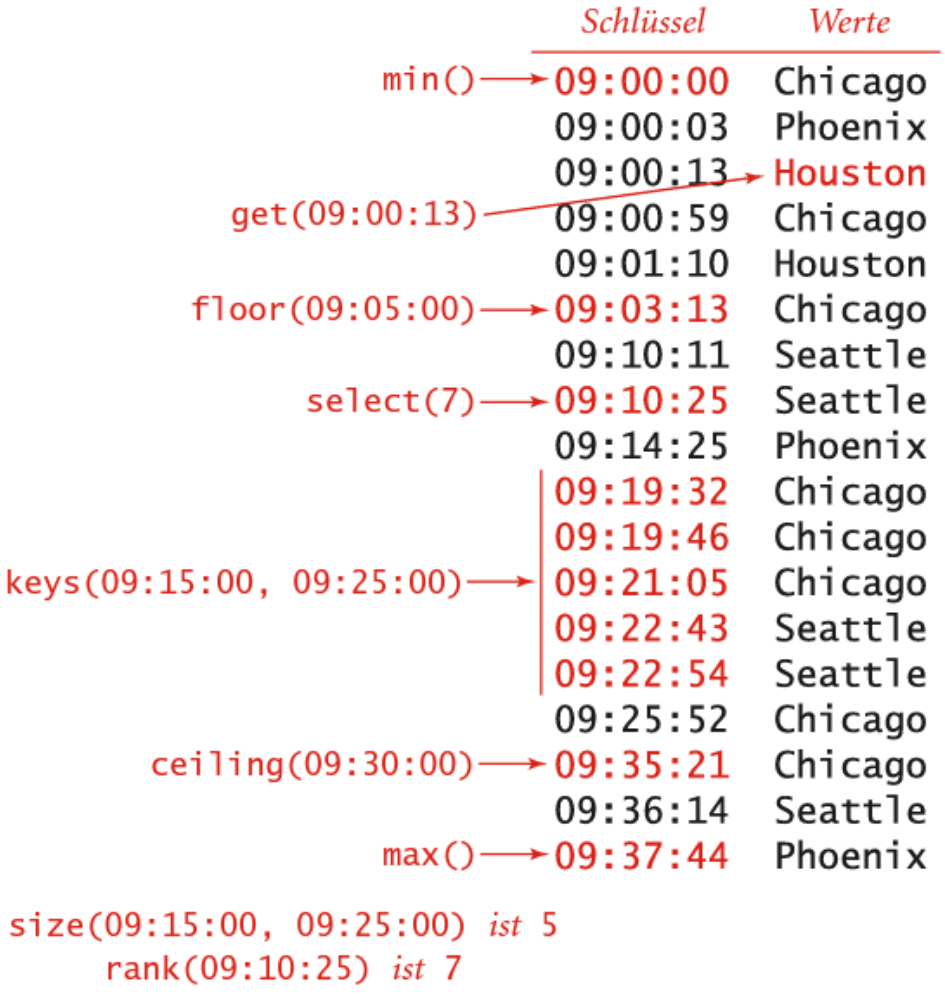
Implementation und Beispielanwendung



IPython Notebooks: BST.ipynb

Ordnungsbasierte Methoden

Geordnete Symboltabellen: API

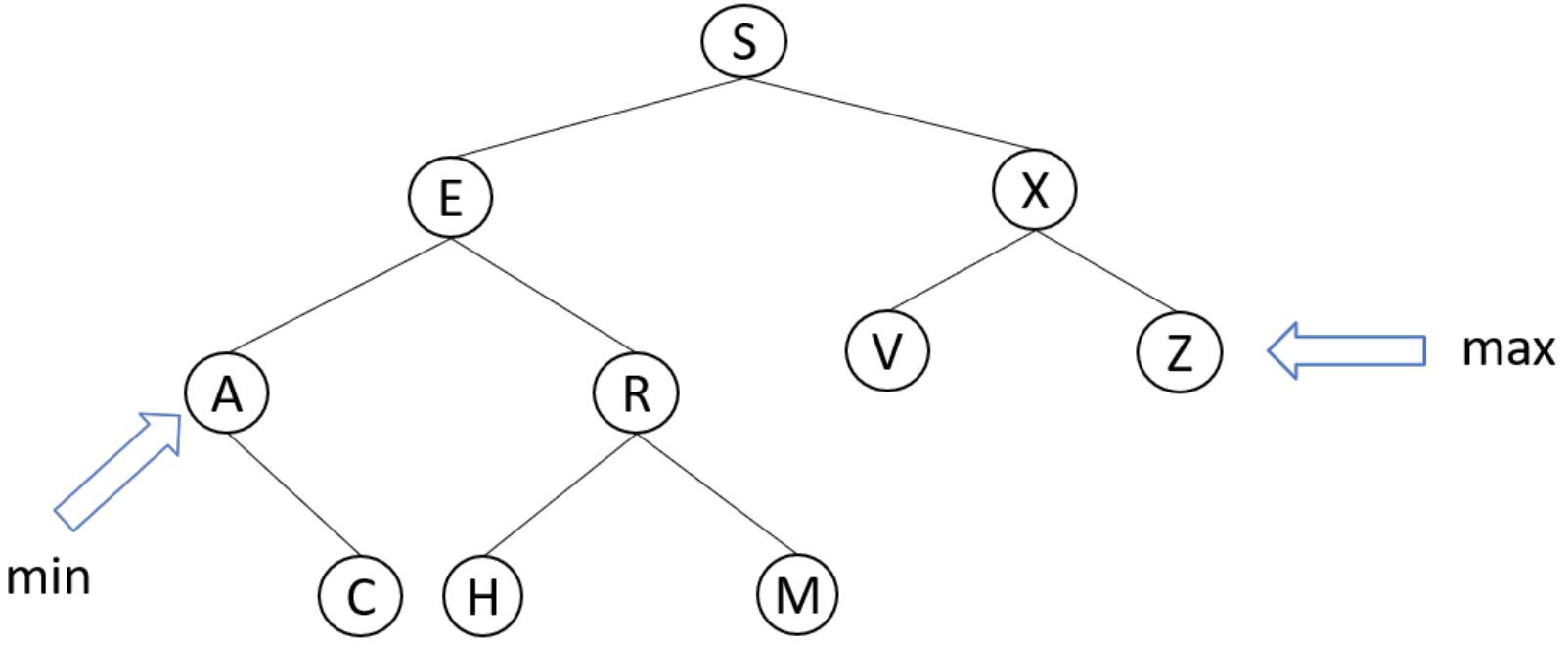


Quelle: Abbildung 3.1, Algorithmen, Wayne & Sedgewick

Quiz: Minimum und Maximum

Minimum Kleinster Schlüssel in Symboltabelle

Maximum Grösster Schlüssel in Symboltabelle

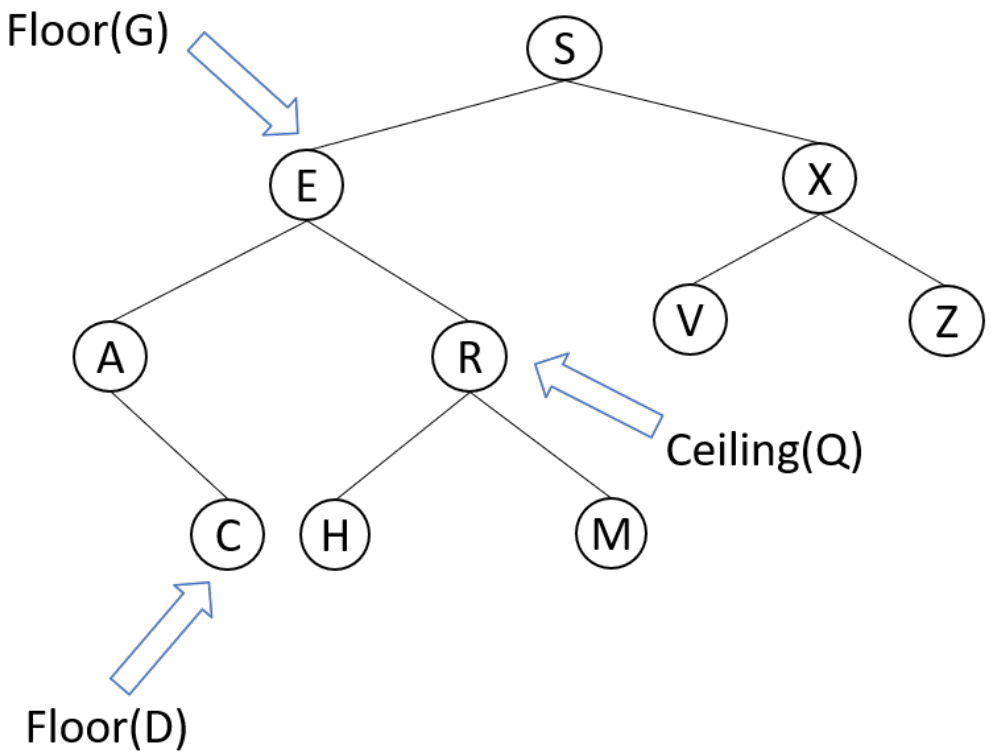


- Wie finden wir Minimum und Maximum?

Quiz: Floor und Ceiling

Floor Grösster Schlüssel $\leq$ gegebener Schlüssel

Ceiling Kleinster Schlüssel $\geq$ gegebener Schlüssel



- Wie finden wir Floor und Ceiling?

Floor berechnen

Fall 1: Gesuchter Schlüssel $=$ Schlüssel in Knoten
- Gefunden

Fall 2: Gesuchter Schlüssel $<$ Schlüssel in Knoten
- Im linken Unterbaum schauen

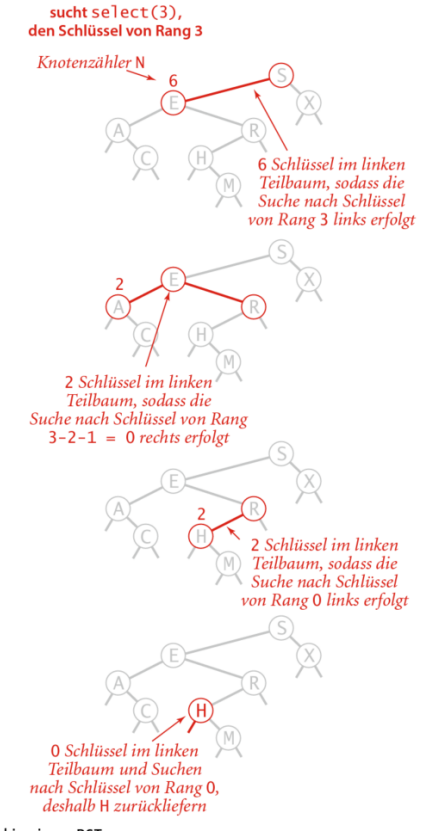
Fall 3: Gesuchter Schlüssel $>$ Schlüssel in Knoten
- Im rechten Unterbaum schauen
  - Falls es kleineren Schlüssel gibt, suche fortsetzen.
  - Ansonsten ist floor Wert im aktuellen Knoten.

Achtung: Falls Schlüssel $<$ kleinstes Element im Baum existiert floor nicht.

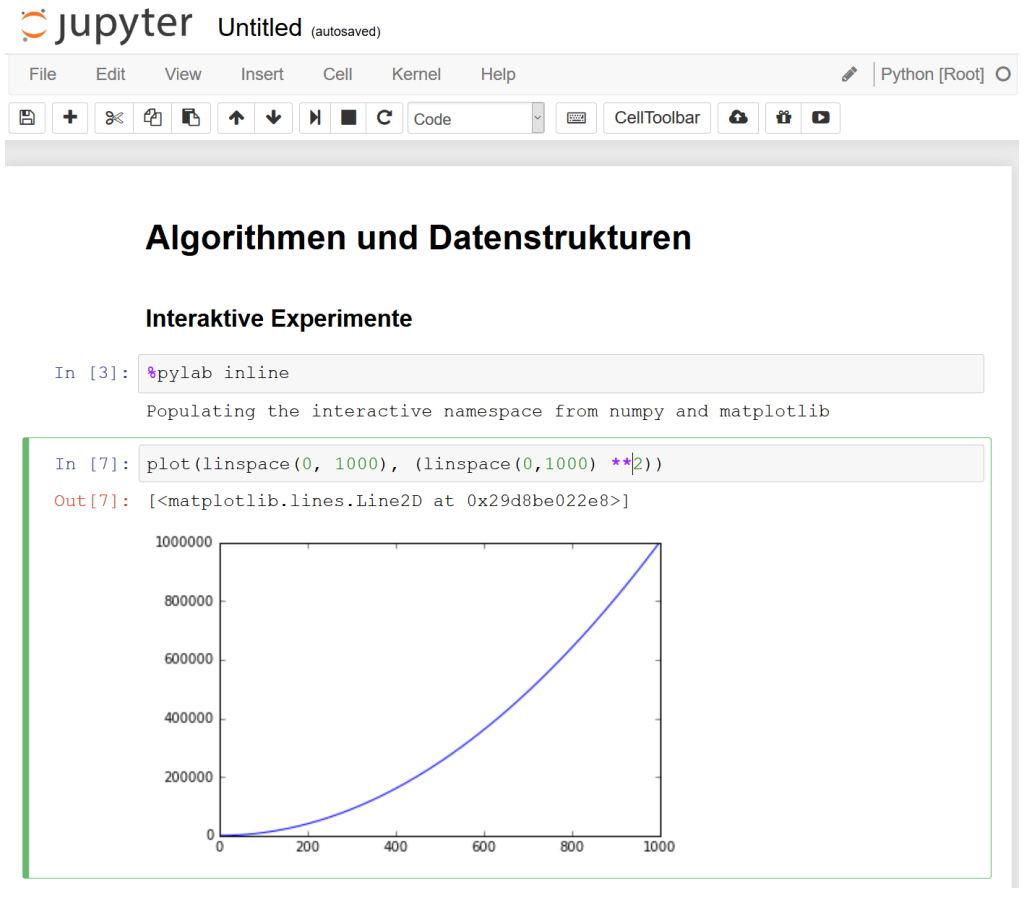
Select

Select Wert von Schlüssel mit Rang k zurückgeben

```
def select(k):
  node = self._select(root, k);
  return node.key;

def select(node, k):
  if node == None:
    return None
  t = size(node.left);
  if t > k:
    return select(node.left,  k);
  elif t < k:
    return select(node.right, k-t-1)
  else:
    return node;
```



Implementation und Beispielanwendung



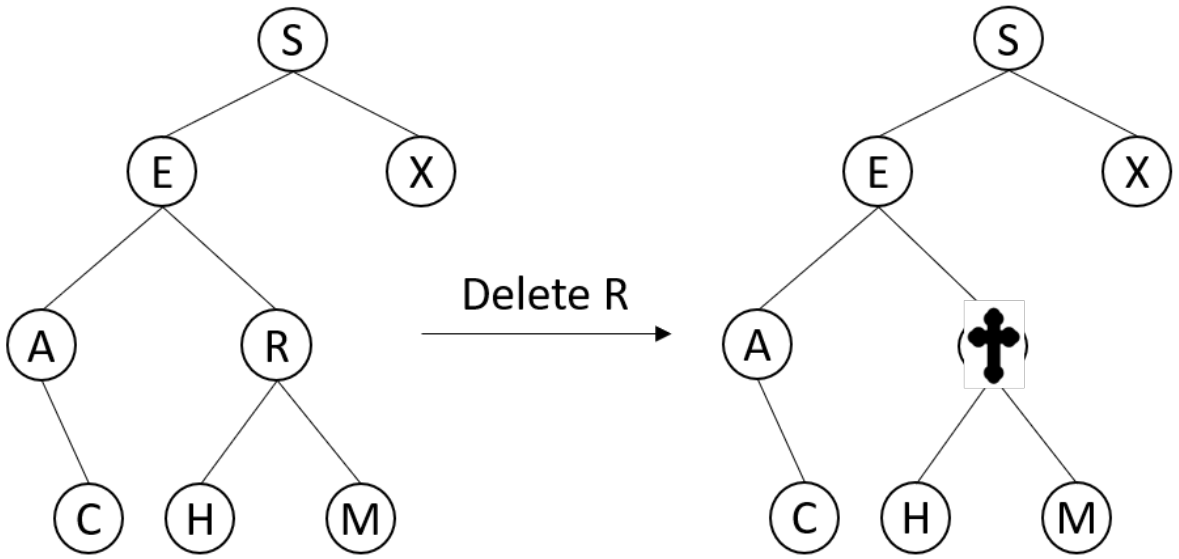
IPython Notebooks: BST.ipynb

Löschen

Löschen von Knoten: Einfache Methode

Einfachste Methode zum Löschen: Tombstone

- Finde Knoten
- Markiere diesen als gelöscht (z.B. indem Wert auf `null` gesetzt wird).
  - Schlüssel bleibt im Baum

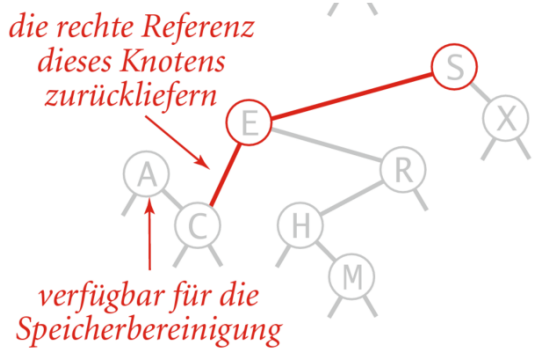


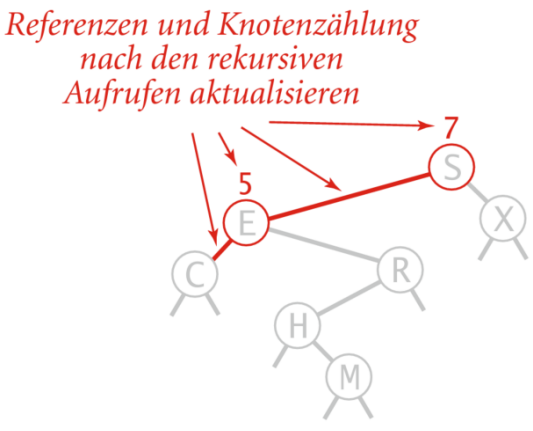
Problem: Speicherverschwendung bei vielen gelöschten Elementen.

Löschen von minimalem Key

- Nach Links bis linker Knoten null ist
- Diesen Knoten durch rechten Knoten ersetzten
- Knotenzähler count aktualisieren.

```
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
```

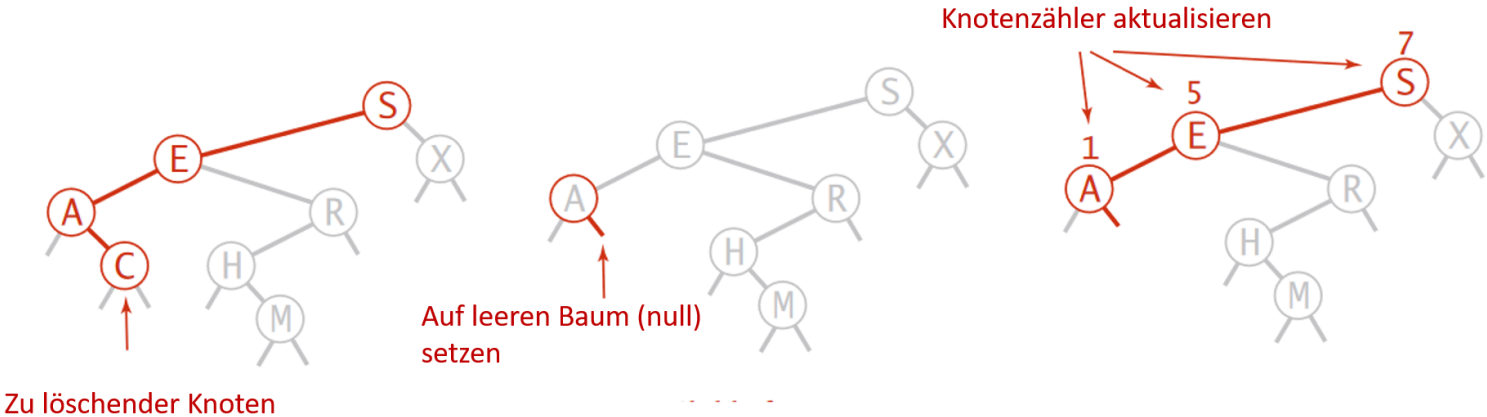




Löschen nach Hibbard

- Knoten t mit zu löschendem Schlüssel suchen.

Fall 1: Keine Kinder

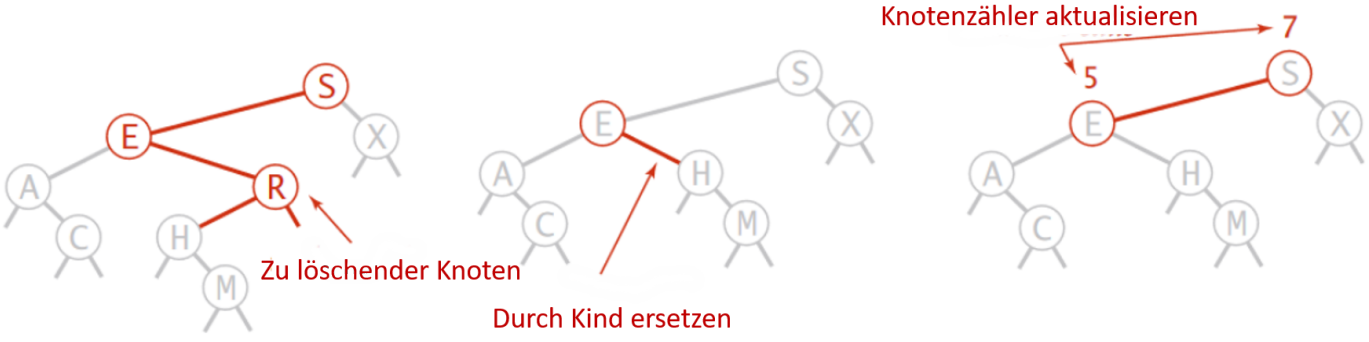


- Parent von t auf leeren Baum (null) setzen.
- Knotenzähler `count` aktualisieren.

Löschen nach Hibbard

- Knoten t mit zu löschendem Schlüssel suchen.

Fall 2: 1 Kind

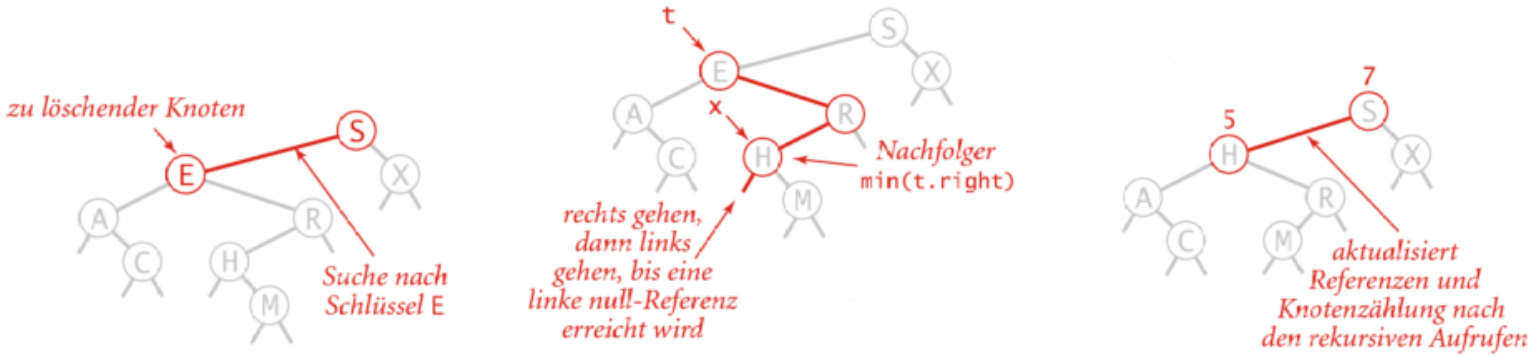


- Parent von t neu setzen
- Knotenzähler `count` aktualisieren.

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
- Knotenzähler `count` aktualisieren.

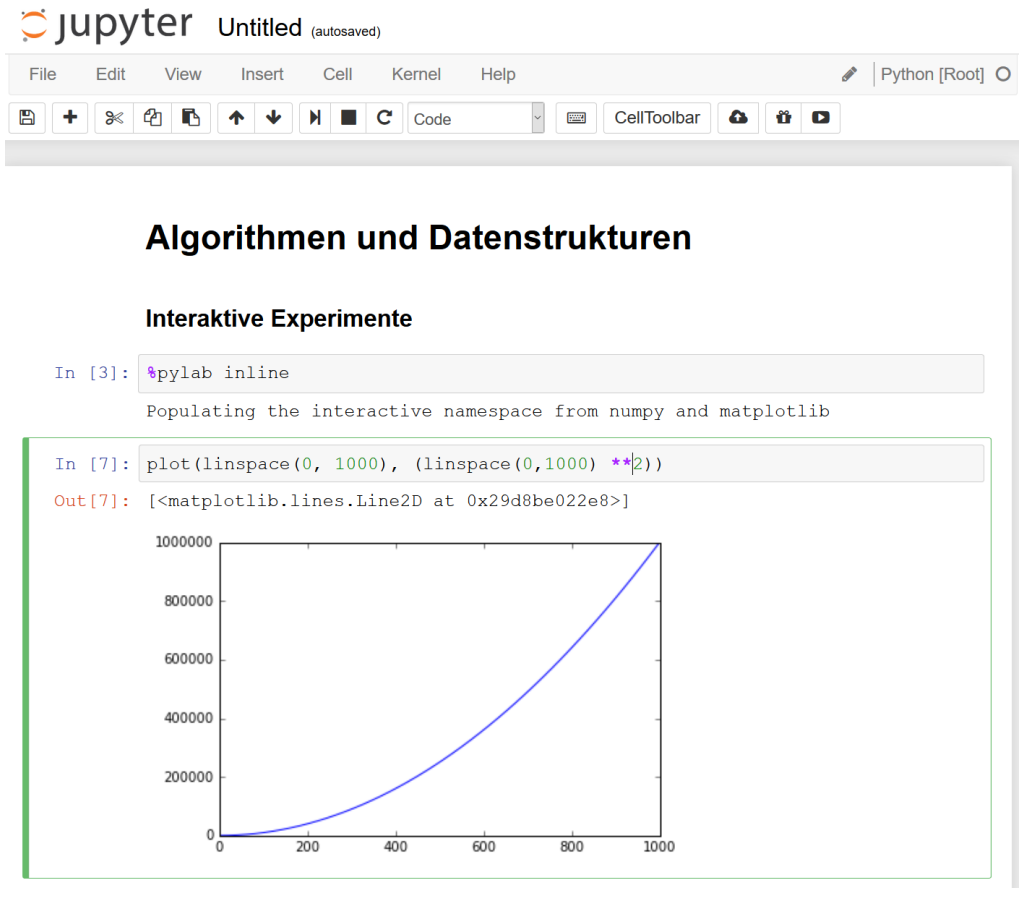
Löschen nach Hibbard: Probleme

- Warum wird durch Nachfolger und nicht Vorgänger ersetzt?
- Entscheidung willkürlich und unsymmetrisch.
- Konsequenz: Bäume nicht zufällig $\Rightarrow$ Performanceeinbussen
  - Praxis: Manchmal Vorgänger und manchmal Nachfolger verwenden.

Offenes Problem!

Elegante und effiziente Lösung für Löschen in Binärbaum.

Implementation und Beispielanwendung



IPython Notebooks: BST.ipynb

Komplexität

	Worst-case			Average-case		
Implementation	suchen	einfügen	löschen	suchen (hit)	einfügen	lösche
Verkettete Liste	N	N	N	N/2	N	N/2
Binäre suche	$\log_2(N)$	N	N	$\log_2(N)$	$N/2$	N
Binärer Suchbaum	N	N	N	$1.39\log_2(N)$	$1.39\log_2(N)$	$\sqrt{N}$