# Algorithmen und Datenstrukturen

## B6. Symboltabellen[1]

Marcel Lüthi and Gabriele Röger

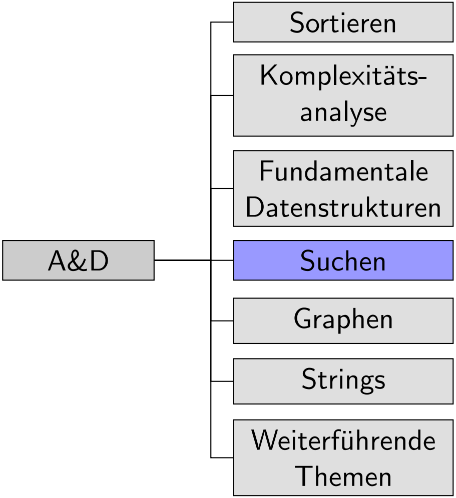
Universität Basel

03. April 2019

[1] Folien basieren Teilweise auf Vorlesungsfolien von Sedgewick & Wayne
https://algs4.cs.princeton.edu/lectures/31ElementarySymbolTables-2x2.pdf

# Einführung

# Übersicht


A&D
Sortieren
Komplexitäts-analyse
Fundamentale Datenstrukturen
Suchen
Graphen
Strings
Weiterführende Themen

# Übersicht über nächsten Vorlesungen

Thema: Symboltabellen

- Einführung und einfache Implementationen (Diese Woche)
- Binäre Suchbäume (Diese Woche)
- 2-3-Bäume und Rot-Schwarz Bäume (Nächste Woche)
- Hashtabellen (Nächste Woche)

# Symboltabellen

# Symboltabellen

**Abstraktion für Schlüssel/Werte Paar**

Grundlegende Operationen

- Speichere Schlüssel mit dazugehörendem Wert.
- Suche zu Schlüssel gehörenden Wert.
- Schlüssel und Wert löschen.

# Beispiel: DNS

- Einfügen von Domainname (Schlüssel) mit gegebener IP Adresse (Wert)
- Gegeben Domainname, finde IP Adresse

| Domainname | IP Adresse |
|---|---|
| informatik.cs.unibas.ch | 131.152.227.35 |
| www.unibas.ch | 131.152.228.33 |
| www.cs.princeton.edu | 128.112.136.11 |
| www.fsf.org | 208.118.235.174 |

# Andere Beispiele

| Anwendung | Zweck der Suche | Schlüssel | Wert |
|---|---|---|---|
| Wörterbuch | Definition finden | Wort | Definition |
| Websuche | Finde Webseite | Suchbegriff | Liste von Webseiten |
| Compiler | Eigenschaften von Variablen | Variablenname | Typ / Wert |
| Dateisystem | Finde Datei auf Disk | Dateiname | Ort auf Disk |
| Log | Finde Events | Timestamp | Logeintrag |

# Annahmen

- Jeder Schlüssel ist eindeutig.
  - Werte mit gleichem Schlüssel werden ersetzt.
- Schlüssel sind vergleichbar.
- Schlüsselgleichheit (Equality) ist definiert.
- Schlüssel sollen nicht mutierbar sein.

- Entspricht verallgemeinerung von Array (mit Schlüssel $\neq$ Index).
- Wird als Assoziatives Array bezeichnet.

# Umsetzung in Programmiersprachen

Symboltabelle werden auch als Map, Assoziatives Array oder Dictionary bezeichnet.

In Java: Teil der Standardbibliothek

- `AbstractMap` mit Subklassen `HashMap` und `TreeMap`

```
Map<String, Integer> st = new TreeMap<>();
st.put("aKey", 42);;
st.put("anotherKey", 17)
Integer value = st.get("aKey");
```

In Python: Teil der Sprache:

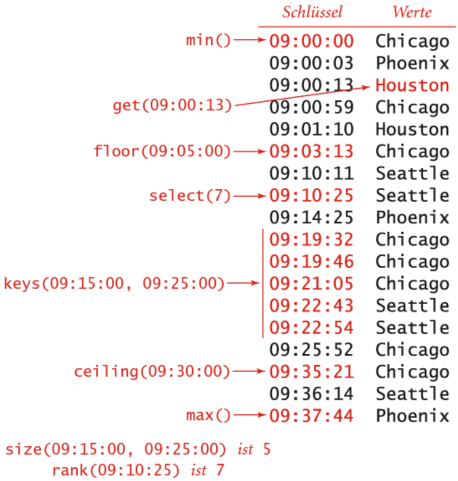
```
st = {"aKey" : 42, "anotherKey" : 17}
value = st["aKey"]
```

# Symboltabellen: API

```
class ST[Key, Value]:

    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]
```

# Geordnete Symboltabellen: API




Quelle: Abbildung 3.1, Algorithmen, Wayne & Sedgewick

# Geordnete Symboltabellen: API

- Wenn die Schlüssel geordnet werden können, lässen sich viele weitere Operationen definieren:

```
class ST[Key, Value]:
    ...
    def min() -> Key
    def max() -> Key

    def floor(key : Key) -> Key
    def ceiling(key : Key) -> Key

    def rank(key : Key) : Int
    def select(k : Int) -> None

    def deleteMin() -> None
    def deleteMax() -> None

    def size(lo : Key, hi : Key) -> Int

    def keys() : Iterator[Key]
    def keys(lo : Key, hi : Key) -> Iterator[Key]
```

# Warnung: Gleichheit von Objekten

- Zwei Arten von Gleichheit in OO Sprachen:

  Referenzgleichheit (==) `Referenzen sind gleich (gleiches Objekt)`

  Objektgleichheit (`equals`) Inhalt ist gleich

**Achtung!**

Implementation von benutzerdefinierten Klassen in Java und Python vergleicht per Default nur Objekt-Id und nicht Inhalt.

- Methoden `equals` (Java) und `__eq__` (Python) müssen implementiert werden.

# Einfache Implementationen

# Standard Testbeispiel

Bilde eine Symboltabelle bei der der $i-$te Input mit dem Wert $i$ assoziiert ist

Input:

| Schlüssel | S | E | A | R | C | H | E | X | A | M | P | L | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Werte | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

Symboltabelle:

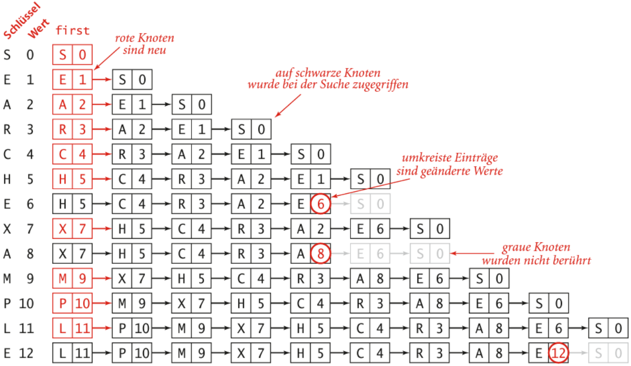
| Schlüssel | A | C | E | H | L | M | P | R | S | X |
|---|---|---|---|---|---|---|---|---|---|---|
| Werte | 8 | 4 | 12 | 5 | 11 | 9 | 10 | 3 | 0 | 7 |

# Einfache Implementation 1

**Datenstruktur** Verkettete Liste von Schlüssel/Werte-Paaren

**Suchen** Elemente durchlaufen bis gefunden oder Listenende

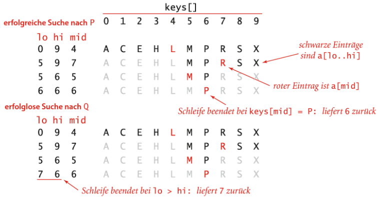
**Einfügen** Element in Liste? Wert ändern. Ansonsten: Am Anfang einfügen.




Quelle: Abbildung 3.3, Algorithmen, Wayne & Sedgewick

# Intermezzo: Binary search

- Klassischer Algorithmus zum Suchen in geordnetem Array
  - Vergleiche Element mit mittlerem Element des Arrays
  - Wiederhole in Teilarray, bis Element gefunden oder Teilarray leer.

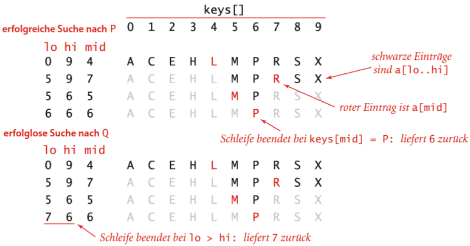



Quelle: Abbildung 1.9, Algorithmen, Wayne & Sedgewick

```
def binarysearch(a, value):
  lo, hi = 0, len(a) - 1
  while lo <= hi:
    mid = (lo + hi) // 2
    if a[mid] < value:
      lo = mid + 1
    elif value < a[mid]:
      hi = mid - 1
    else:
      return mid
  return None
```

# Die Rank Funktion

- Gibt Anzahl Elemente zurück die kleiner als Schlüssel sind
    - Entspricht genau Index in Array




Quelle: Abbildung 3.6, Algorithmen
Wayne & Sedgewick

```
def _rank(a, value):
  lo = 0
  hi = len(a) - 1
  while lo  <= hi:
    mid = (lo + hi) // 2
    if  a[mid] < value:
      lo = mid + 1
    elif  value  < a[mid]:
      hi = mid - 1
    else:
      return mid
  return lo
```

# Einfache Implementation 2

**Datenstruktur** Geordnetes Array von Schlüssel/Werte-Paaren

**Hilfsfunktion `rank`** Anzahl Elemente $< k$ (index in Array)

Operationen:

- `get`: Nutze `rank` um direkt auf richtiges Element zuzugreifen.
  - Teste ob wirklich richtiges Element an dieser Stelle ist
- `put`: Nutze `rank` um Stelle zu finden wo eingefügt/ersetzt werden muss.

Details: Jupyter Notebook: `Symboltable.ipynb`

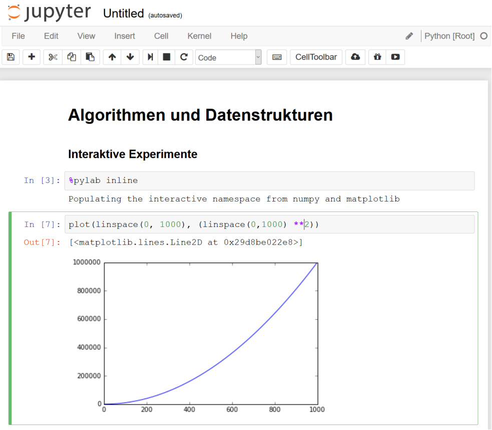
# Komplexität

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

# Geordnete Symboltabellen: Komplexität

| | Verkettete Liste | Binärsuche |
|---|---|---|
| suche | $O(N)$ | $O(\log N)$ |
| einfügen / löschen | $O(N)$ | $O(N)$ |
| min / max | $O(N)$ | $O(1)$ |
| floor /ceiling | $O(N)$ | $\log(N)$ |
| rank | $O(N)$ | $O(\log(N))$ |
| select | $O(N)$ | $O(1)$ |
| iteration (geordnet) | $N\log(N)$ | $N$ |

# Implementation




- Ausführliche Diskussion und Implementation Jupyter-Notebook: `Symboltable.ipynb`

# Binäre Suchbäume

# Binäre Suchbäume

Ein Binärer Suchbaum ist ein Binärbaum mit symmetrischer Ordnung

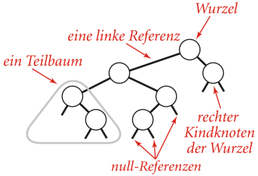
## Ein Binärbaum ist

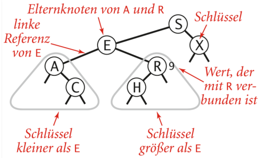
- der leere Baum, oder
- eine Wurzel mit einem linken und einem rechten Unterbaum

## 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

# Implementation

```
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]
```

- Implementation Symboltabelle: Referenz zu Wurzel Knoten

# Repräsentation in Code (mit Zähler)

- 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

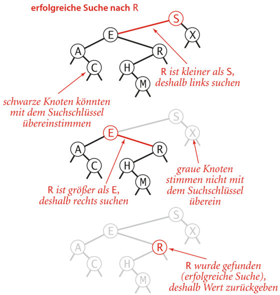
- Um get zu implementieren, müssen wir effizient suchen können.

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

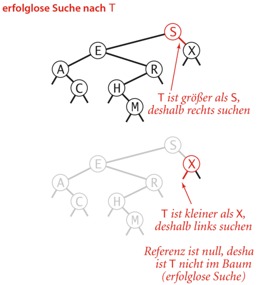
- Gehe nach links

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.

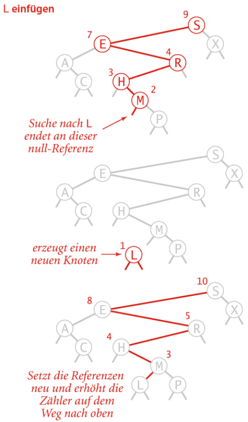
```
def get(key, root):
    if root == None:
        return None
    elif key < root.key:
        return get(key, root.left)
    elif key > root.key:
        return get(key, root.right)
    elif key == root.key:
        return root.value
```

# 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

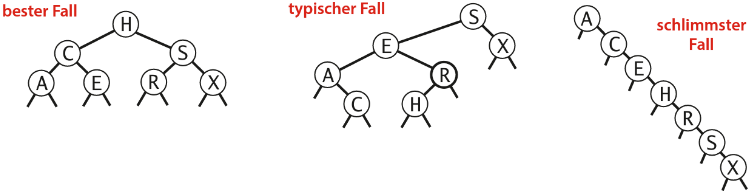
# Einfügen in Binärbaum

- 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
```

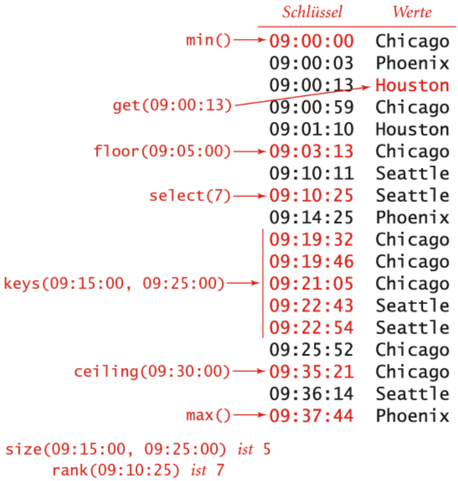
# Ausprägung des Binärbaums

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




Quelle: Abb. 3.14, Algorithmen, Wayne & Sedgewick

# Geordnete Symboltabellen: API

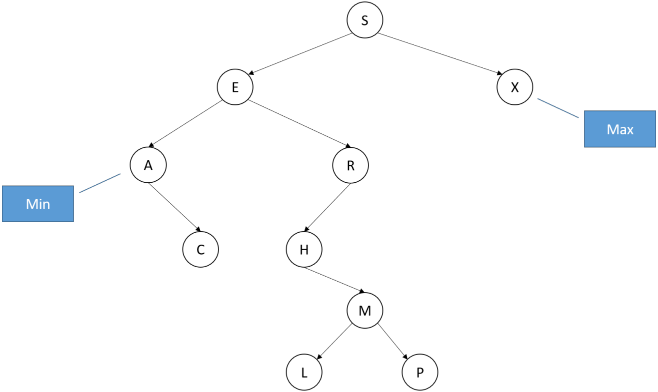



Quelle: Abbildung 3.1, Algorithmen, Wayne & Sedgewick

# Quiz: Minimum und Maximum

**Minimum** Kleinster Schlüssel in Symboltabelle

**Maximum** Grösster Schlüssel in Symboltabelle

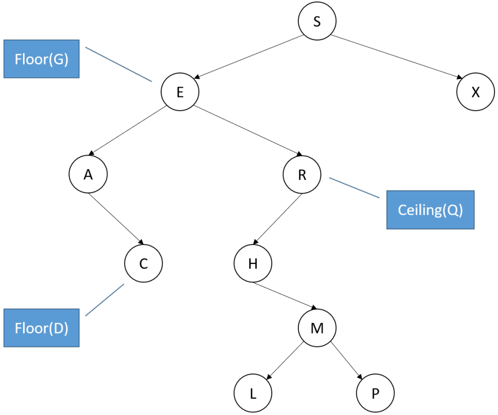



- Wie finden wir Minimum und Maximum?

# Quiz: Floor und Ceiling

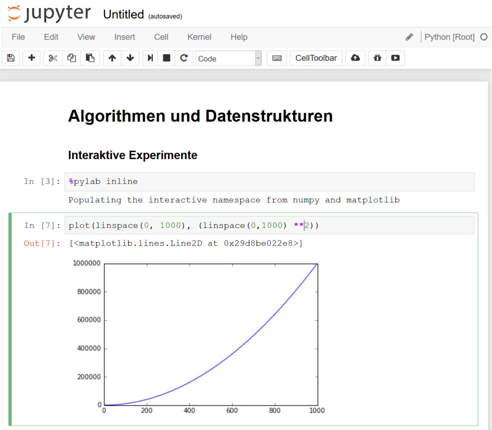
**Floor** Grösster Schlüssel $\leq$ gegebener Schlüssel

**Ceiling** Kleinster Schlüssel $\geq$ gegebener Schlüssel




- Wie finden wir Floor und Ceiling?

# Ordnungsbasierte Operationen

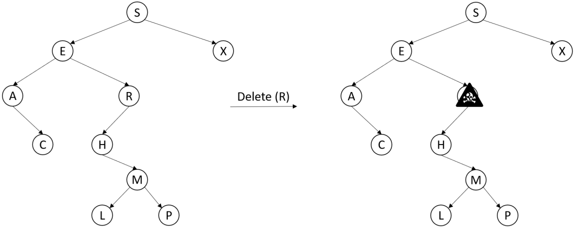



- Ordnungsbasierten Operationen sind einfach zu implementieren.
- Ausführliche Diskussion und Implementation Jupyter-Notebook: `Symboltable.ipynb`

# 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

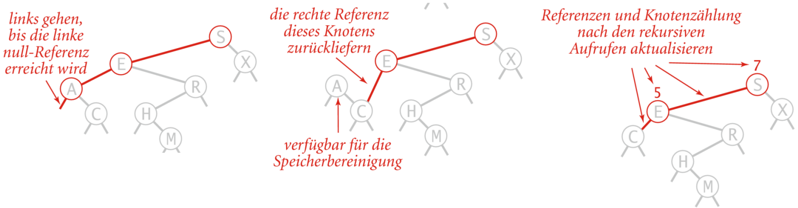



Problem: Speicherverschwendung bei vielen gelöschten Elementen.

# Löschen von minimalem Key

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

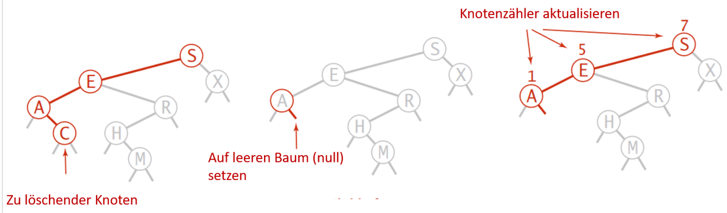
```
def deleteMin(root):
  if root.left == None:
    return root.right
  else:
    root.left = deleteMin(x.left);
    root.count = 1 + size(root.left)   + size(root.right);
    return root
```



# Löschen nach Hibbard

- Knoten $t$ mit zu löschendem Schlüssel suchen.

Fall 1: Keine Kinder

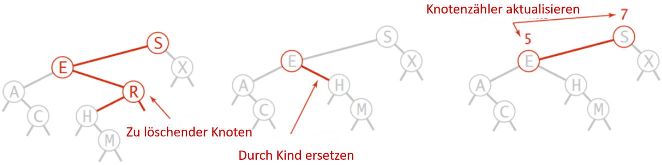



- Parent von $t$ auf leeren Baum (null) setzen.
- Knotenzähler `count` aktualisieren.

# Löschen nach Hibbard

- Knoten $t$ mit zu löschendem Schlüssel suchen.

Fall 2: 1 Kind

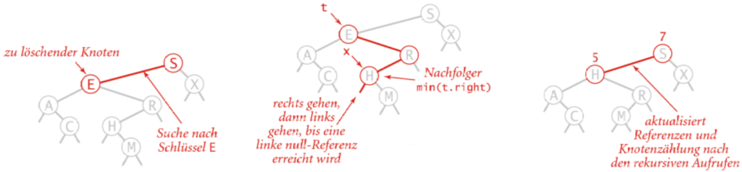



- Parent von $t$ neu setzen
- Knotenzähler `count` aktualisieren.

# 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.

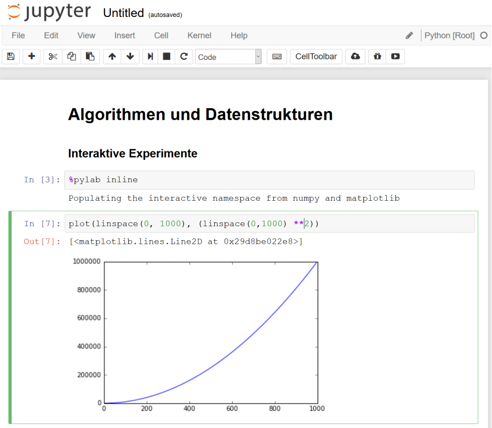
**Offenes Problem!**

Elegante und effiziente Lösung für Löschen in Binärbaum.

# 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 |
| Binärer Suchbaum | $N$ | $N$ | $N$ | $\log_2(N)$ | $\log_2(N)$ | $\sqrt{N}$ |

# Implementation




Jupyter-Notebook: `Symboltable.ipynb`