

Algorithmen und Datenstrukturen

B6. Binäre Suchbäume¹

Marcel Lüthi and Gabriele Röger

Universität Basel

11. April 2018

¹Folien basieren auf Vorlesungsfolien von Sedgwick & 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

Implementation	suchen	Worst-case		Average-case		
		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**

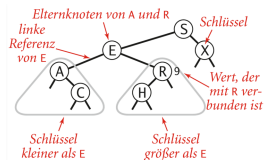
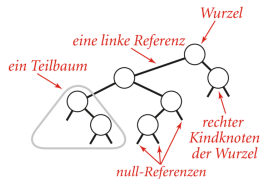
Ein Binärbaum ist

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

Symmetrische Ordnung

Der Schlüssel jedes Knotens ist

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



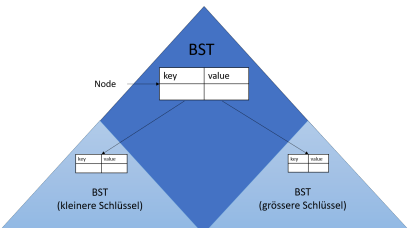
Binäre Suchbäume und Symboltabellen

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

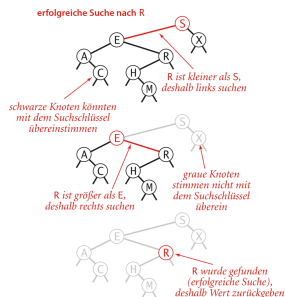
- Gehe nach links

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

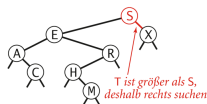
Fall 2: $k >$ Schlüssel in Knoten

- Gehe nach rechts

Fall 3: $k =$ Schlüssel in Knoten

- Gefunden

erfolgreiche Suche nach T



Referenz ist null, deshalb ist T nicht im Baum (erfolgreiche Suche)

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

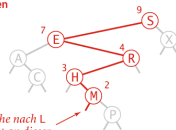
- `put` lässt sich fast so einfach wie `get` implementieren.

Suche nach Schlüssel.

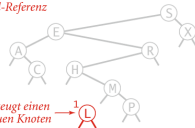
Zwei Fälle:

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

L einfügen



Suche nach L
endet an dieser
null-Referenz



erzeugt einen
neuen Knoten



Setzt die Referenzen
neu und erhöht die
Zähler auf dem
Weg nach oben

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.

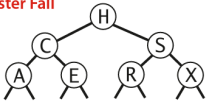
```
def put(key, value, root):  
    if (root == None):  
        return Node(key, value, count = 1)  
    elif key < root.key:  
        root.left = put(key, value, root.left)  
    elif key > root.key:  
        root.right = put(key, value, root.right)  
    elif key == root.key:  
        root.value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

Analyse

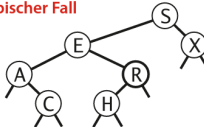
Ausprägung des Binärbaums

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

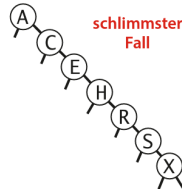
besten Fall



typischer Fall

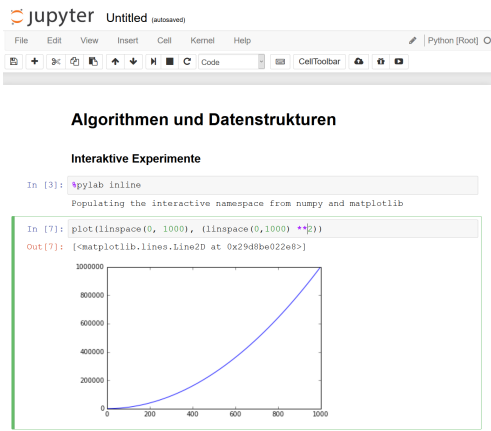


schlimmster Fall



Quelle: Abb. 3.14, Algorithmen, Wayne & Sedgwick

Implementation und Beispielanwendung



IPython Notebooks: BST.ipynb

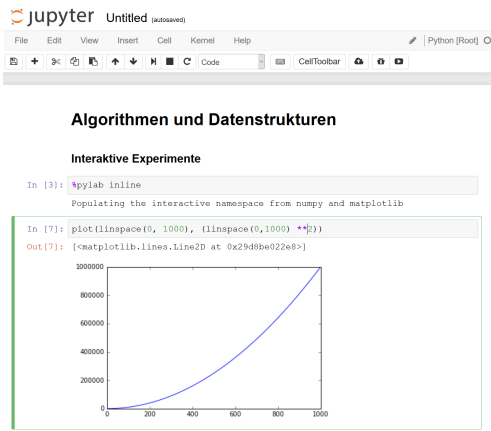
Sortieralgorithmus

- 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

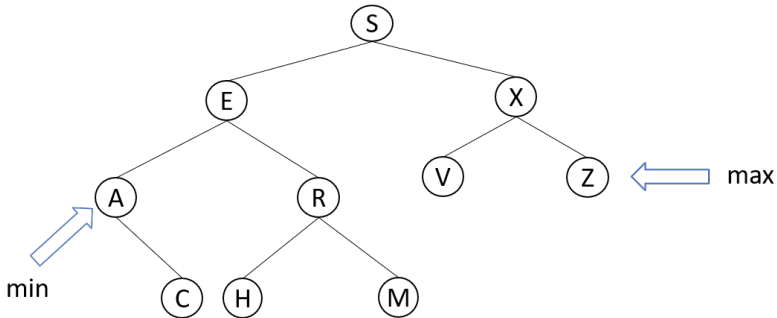
	<i>Schlüssel</i>	<i>Werte</i>
<code>min()</code> →	09:00:00	Chicago
	09:00:03	Phoenix
	09:00:13 →	Houston
<code>get(09:00:13)</code> →	09:00:59	Chicago
	09:01:10	Houston
<code>floor(09:05:00)</code> →	09:03:13	Chicago
	09:10:11	Seattle
<code>select(7)</code> →	09:10:25	Seattle
	09:14:25	Phoenix
	09:19:32	Chicago
	09:19:46	Chicago
<code>keys(09:15:00, 09:25:00)</code> →	09:21:05	Chicago
	09:22:43	Seattle
	09:22:54	Seattle
	09:25:52	Chicago
<code>ceiling(09:30:00)</code> →	09:35:21	Chicago
	09:36:14	Seattle
<code>max()</code> →	09:37:44	Phoenix
<code>size(09:15:00, 09:25:00)</code> ist 5		
<code>rank(09:10:25)</code> ist 7		

Quelle: Abbildung 3.1, Algorithmen, Wayne & Sedgwick

Quiz: Minimum und Maximum

Minimum Kleinsten Schlüssel in Symboltabelle

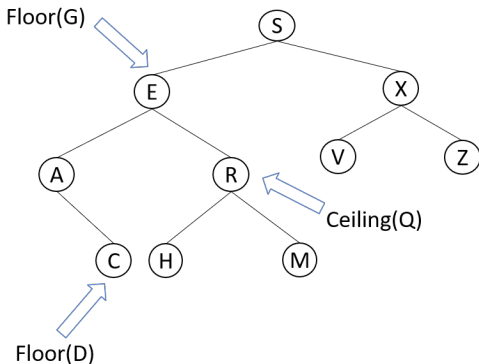
Maximum Grösster Schlüssel in Symboltabelle



- 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.

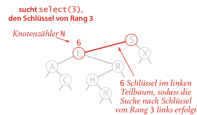
Achtung: Falls Schlüssel $<$ kleinstes Element im Baum existiert floor nicht.

Select

Select Wert von Schlüssel mit Rang k zurückgeben

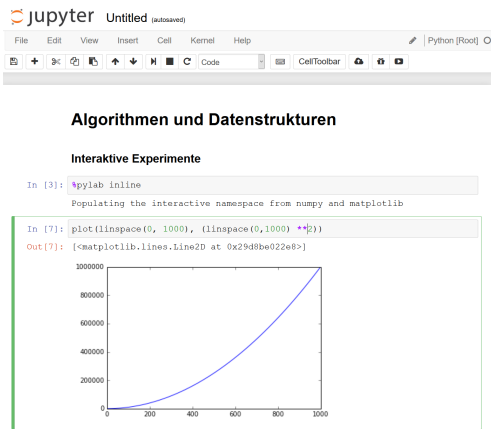
```
def select(k):
    node = self._select(root, k);
    return node.key;

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



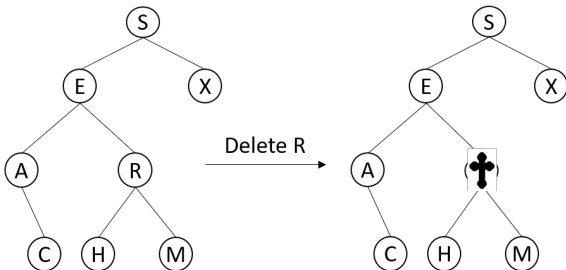
IPython Notebooks: BST.ipynb

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 ersetzen
- 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
```

links gehen,
bis die linke
null-Referenz
erreicht wird

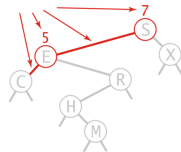


die rechte Referenz
dieses Knotens
zurückliefern



verfügbar für die
Speicherbereinigung

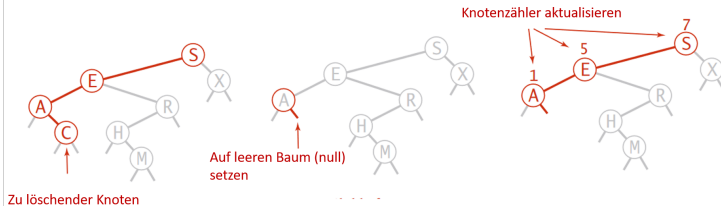
Referenzen und Knotenzählung
nach den rekursiven
Aufrufen aktualisieren



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 setzen
- Knotenzähler `count` aktualisieren.

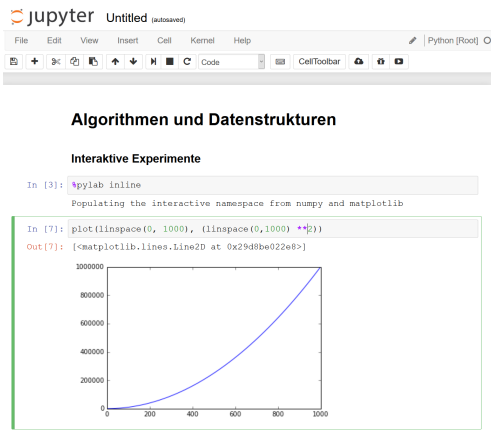
Löschen nach Hibbard: Probleme

- 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



The screenshot shows a Jupyter Notebook window titled "Untitled (autosaved)". The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Help) and a toolbar with various icons. The notebook content is as follows:

Algorithmen und Datenstrukturen

Interaktive Experimente

```
In [3]: %pylab inline
```

Populating the interactive namespace from numpy and matplotlib

```
In [7]: plot(linspace(0, 1000), (linspace(0,1000) **2))
```

```
Out [7]: [<matplotlib.lines.Line2D at 0x29d8be022e8>]
```

The plot displays a blue curve representing the function $y = x^2$ for x in the range $[0, 1000]$. The x-axis is labeled from 0 to 1000 in increments of 200. The y-axis is labeled from 0 to 1,000,000 in increments of 200,000. The curve starts at the origin (0,0) and rises to (1000, 1,000,000).

IPython Notebooks: BST.ipynb

Komplexität

Implementation	suchen	Worst-case			Average-case		
		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	$1.39 \log_2(N)$	$1.39 \log_2(N)$	\sqrt{N}	