Höhe eines B⁺-Baums

- Anzahl der Blätter bei minimaler Belegung

$$B_{\min}(k,h) = \begin{cases} 1 & h = 1 \\ 2(k+1)^{h-2} & h \geq 2 \end{cases}$$

- Anzahl von (eindeutigen) Elementen bei minimaler Belegung

$$n_{\min}(k,l,h) = \begin{cases} 1 & h = 1 \\ 2l(k+1)^{h-2} & h \geq 2 \end{cases}$$

- Anzahl der Blätter bei maximaler Belegung

$$B_{\max}(k,h) = (2k+1)^{h-1} \qquad h \leq 1$$

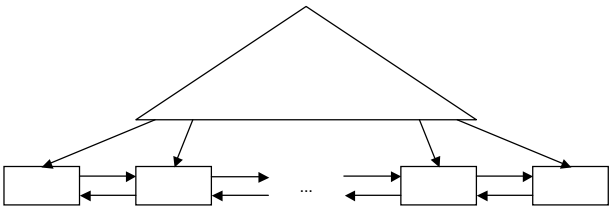
- Anzahl von Elementen bei maximaler Belegung

$$n_{\max}(k,l,h) = 2l(2k+1)^{h-1} \qquad h \leq 1$$

- Dann ergibt sich die Höhe eines B⁺-Baums

$$1+\log_{2k+1}\left(\frac{n}{2l}\right) \leq h \leq 2+\log_{k+1}\left(\frac{n}{2l}\right) \qquad h \geq 2$$

Zugriff auf Elemente im B⁺-Baum

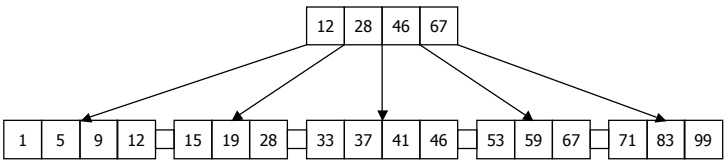


- da die Nutzdaten in den Blättern gespeichert sind, erfordert jeder Zugriff genau *h* Knotenzugriffe
- sequentieller Zugriff:
  - *h* interne Knoten + alle Blätter
  - Beispiel: „range query" (z.B. DB-Anfragen vom Typ „finde alle Studenten mit Matrikelnummner zwischen N_1 und N_2)

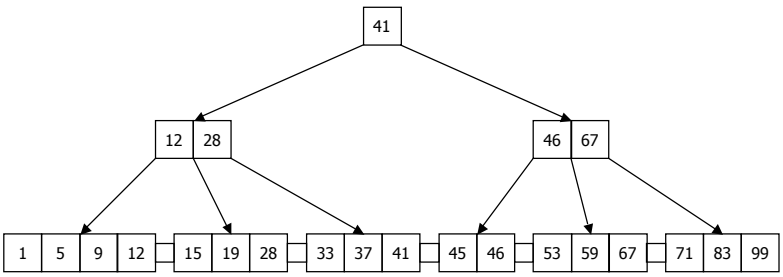
Einfache Operationen auf B$^+$-Bäumen

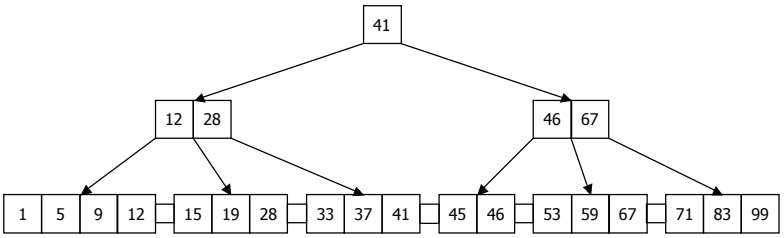
- Einfügen
  - interne Knoten genauso wie bei B-Bäumen
  - Blätter nach dem gleichen Prinzip, es muß garantiert werden, daß größere Schlüssel vom neuen Knoten als Seperator in den Vaterknoten kopiert werden
- Löschen
  - leichter als bei B-Bäumen
  - denn Schlüssel in den internen Knoten sind nur Seperatoren, sie dürfen nicht entfernt werden

Beispiel (k=l=2)

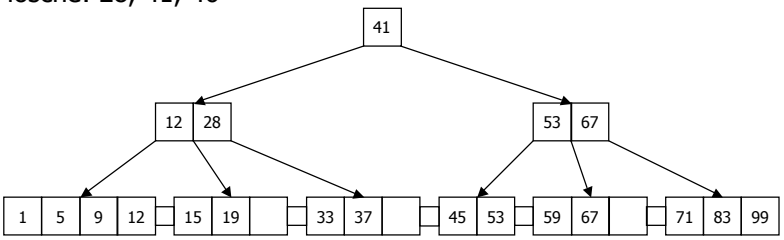


- füge ein: 45





- lösche: 28, 41, 46



Vergleich zwischen B-Bäumen und B$^+$-Bäumen

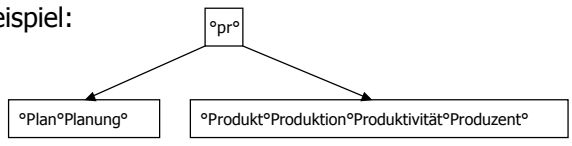
B-Baum	**B$^+$-Baum**
Nutzdaten in allen Knoten	Nutzdaten nur in den Blättern
Keine Schlüssel-Redundanz	Schlüssel-Redundanz
höhere Bäume wegen großer Seperatoren	niedrigere Bäume (große Breite) wegen kleiner Seperatoren
Sequentieller Zugriff auf Folge von Schlüsseln ist aufwendiger	Sequentieller Zugriff = Traversierung einer linearen Liste
	vereinfachter Lösch-Algorithmus
$n_{min} = 2(k+1)^{h-1} - 1$	$n_{min} = 2k(k+1)^{h-2}$
$n_{max} = (2k+1)^h - 1$	$n_{max} = 2k(2k+1)^{h-1}$

Fallstudie: B-Bäume im Vergleich zu B⁺-Bäumen

- Parameter: Seitengröße = 2048 Byte = 2kB
  Zeiger, Zähler, Schlüssel = 4 Byte
- Nutzdaten: eingebettete Speicherung L_D = 76 Byte
  separate Speicherung L_D = 4 Byte
- B-Baum (eingebettet):
  $k = \lfloor (L-L_M-L_P)/2\ (L_K+L_D+L_P) \rfloor = \lfloor (2048 - 4 - 4) / 2\ (4 + 76 + 4) \rfloor = 12$
- B-Baum (separate Speicherung):
  $k = \lfloor (L-L_M-L_P)/2\ (L_K+L_D+L_P) \rfloor = \lfloor (2048 - 4 - 4) / 2\ (4 + 4+ 4) \rfloor = 85$
- interne Knoten im B⁺-Baum:
  $k = \lfloor (L-L_M-L_P)/2\ (L_K+L_P) \rfloor = \lfloor (2048 - 4 - 4) / 2\ (4 + 4) \rfloor = 127$
- Blätter im B⁺-Baum (eingebettet):
  $l = \lfloor (L-L_M-2L_P)/2\ (L_K+L_D) \rfloor = \lfloor (2048 - 4 - 8) / 2\ (4 + 76) \rfloor = 12$
- Blätter im B⁺-Baum (separate Speicherung):
  $l = \lfloor (L-L_M-2L_P)/2\ (L_K+L_P) \rfloor = \lfloor (2048 - 4 - 8) / 2\ (4 + 4) \rfloor = 127$

Präfix-B-Bäume

- besonders günstig für Strings mit variabler Länge
- Präfix-B-Baum ist ein B-Baum, in dem kleine Präfixe der Schlüssel als Separator benutzt werden
- Präfix-B-Bäume benötigen weniger Platz und erlauben einen größeren Fan-Out
- für die Schlüssel Engels, Hegel, Kant, Marx, Plato, Schopenhauer genügt der erste Buchstabe
- Beispiel:



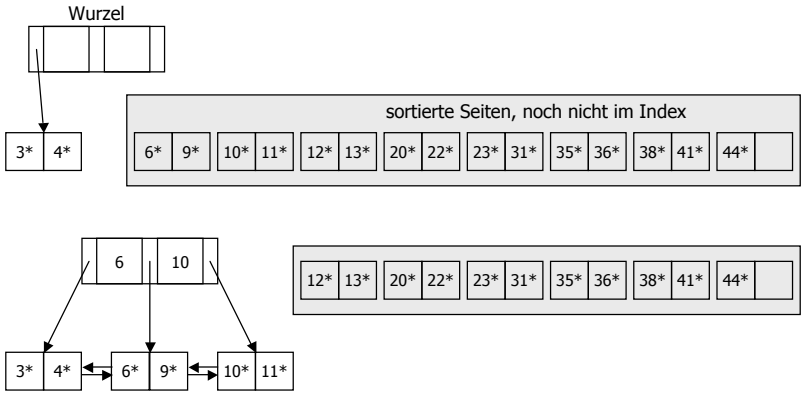
  - oft ist es günstiger, lange Zeichenketten zu komprimieren, z.B. Produkt, 7ion, 8vität, 5zent

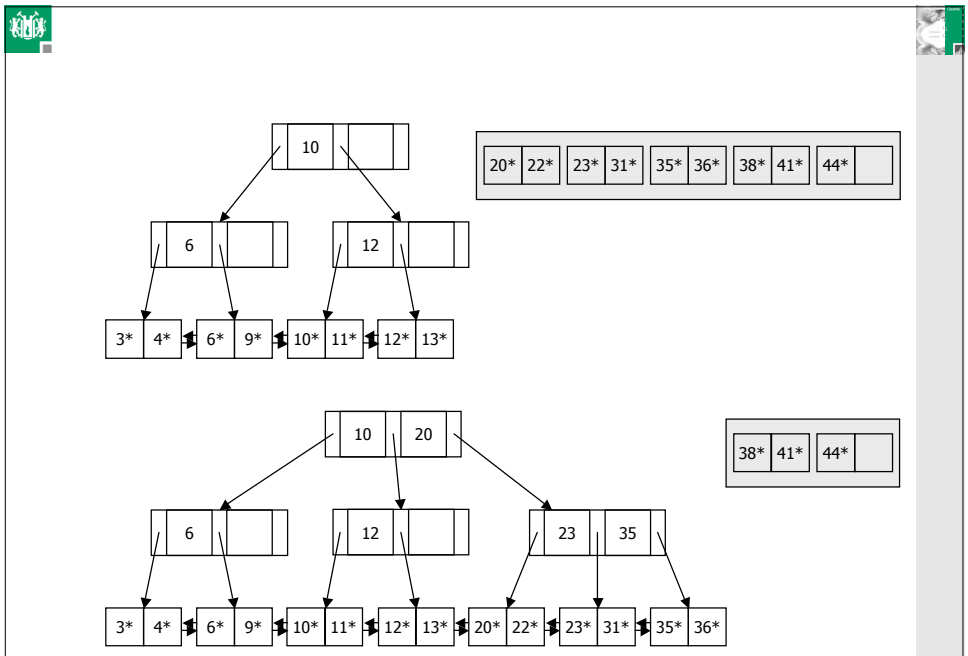
Bulk-Loading von B-Bäumen

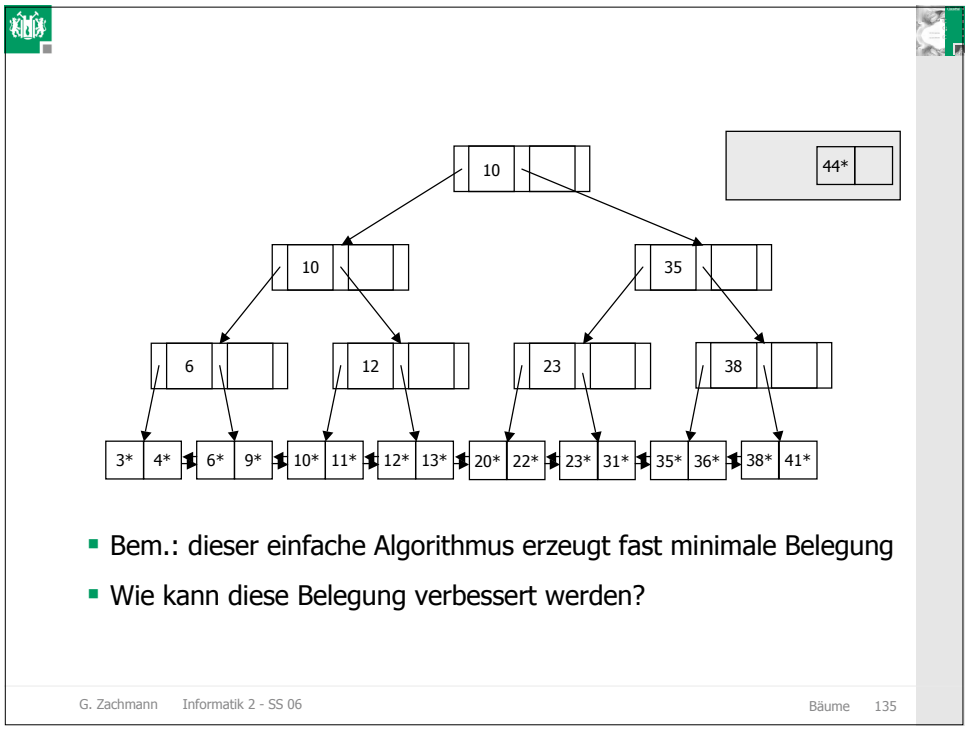
- B-Bäume können schrittweise (durch Einfügen eines einzelnen Schlüssels) oder als Index für bereits vorhanden Daten erstellt werden
- Schrittweise Erstellung ist nicht effizient → „Bulk-Loading"
- 3 Schritte
  - lese Daten und extrahiere die Schlüssel-Zeiger-Paare
  - sortiere die Paare
  - erstelle den Index und aktualisiere

Beispiel

- B^+-Baum mit $k = l = 1$ (aus Platzgründen)
- beginne mit den sortierten Schlüssel-Zeiger-Paaren und einer leeren Wurzel

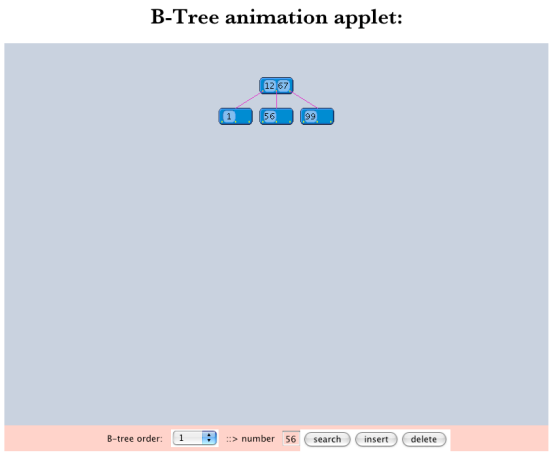






- Bem.: dieser einfache Algorithmus erzeugt fast minimale Belegung
- Wie kann diese Belegung verbessert werden?

Demo



http://slady.net/java/bt/

Komplexitätsanalyse für B-Bäume

- Annahmen: jeder Knoten benötigt nur einen Zugriff auf externe Daten und jeder modifizierte Knoten wird nur einmal geschrieben
- Anzahl der Schlüssel-Zugriffe:
  - Suche ≈ Pfad von Wurzel zu Knoten/Blatt
  - falls lineare Suche im Knoten: O(k) pro Knoten
  - insgesamt $O(k \log_k n)$ Zugriffe im Worst-Case
- Aber: wirklich teuer sind read/write auf Platte
- Grenzen für den Lesezugriff beim Suchen (f = „fetch"):

$f_{\min} = 1$	Schlüssel in der Wurzel
$f_{\max} = h \in O(\log_k(n))$	Schlüssel in einem Blatt
$h - \frac{1}{k} \leq f_{\text{avg}} \leq h - \frac{1}{2k}$	o.Bew.

Kosten für Insert

- f = Anzahl "Fetches", w = Anzahl "Writes"
- 2 Fälle:
  - kein Split: f_{min} = h, w_{min} = 1
  - Wurzel-Split: f_{max} = h, w_{max} = 2h+1
- bei maximalen Split-Operationen ist das Einfügen teuer, das passiert aber selten
- Abschätzung der Split-Wahrscheinlichkeit:
  - höchste Split-Zahl (rel. zur Anzahl Knoten) tritt auf bei der Konstruktion eines minimal belegten Baums
  - Anzahl der Knoten im minimal belegten Baum ist: $N(n) \leq \frac{n-1}{k} + 1$
  - im minimal belegten Baum gibt es höchstens $N(n)$-1 Splits, die Wahrscheinlichkeit für einen Split ist daher

$$p_{split} = \frac{N(n)-1}{n} = (\frac{n-1}{k} + 1 - 1)\frac{1}{n} = \frac{n-1}{nk} < \frac{1}{k}$$

- Eine Insert-Operation erfordert das Schreiben einer Seite, eine Split-Operation erfordert ungefähr 2 Schreib-Operationen
- durchschnittlicher Aufwand für Insert:

$$f_{avg} = h \qquad w_{avg} = 1 + 2p_{split} < 1 + \frac{2}{k}$$

Kosten für Löschen

- Mehrere Fälle:
  1. Best-Case: b > k, kein „Underflow"
     f_{min} = h w_{min} = 1
  2. "Underflow" wird durch Rotation beseitigt (keine Fortpflanzung):
     f_{rot} = h+1 w_{rot} = 3
  3. Mischung ohne Fortpflanzung:
     f_{mix} = h+1 w_{mix} = 2
  4. Worst-Case: Mischen bis hinauf zum Wurzel-Kind, Rotation beim Wurzel-Kind:
     f_{max} = 2h-1 w_{max} = h+1

Durchschnittliche Kosten für Löschen

- Obere Schranke unter der Annahme, daß alle Schlüssel nacheinander gelöscht werden
- kein Underflow: f_1 = h, $w_1 \leq 2$
- Zusätzliche Kosten für die Rotation (höchstens ein Underflow pro gelöschtem Schlüssel): $f_2 = 1$, $w_2 \leq 2$
- Zusätzliche Kosten für das Mischen:
  - 1 Mix-Operation = Elemente eines Knotens in Nachbarknoten mischen und Knoten löschen, d.h., Mix mit Propagation = bis zu h Mix-Op.
  - maximale Anzahl von Mix-Operationen: $N(n) - 1 = \frac{n-1}{k}$
  - Kosten pro Mix-Operation: 1 „read" und 1 „write"
  - zusätzliche Kosten pro Schlüssel:

$$f_3 = w_3 = \frac{\#\text{Mix-Op.}}{\#\text{Schlüssel}} = \frac{N(n)-1}{n} = \frac{(n-1)/k}{n} < \frac{1}{k}$$

- Addition ergibt:

$$f_{avg} \leq f_1 + f_2 + f_3 < h + 1 + \frac{1}{k}$$
$$w_{avg} \leq w_1 + w_2 + w_3 < 2 + 2 + \frac{1}{k} = 4 + \frac{1}{k}$$

Digitale Suchbäume & Binäre Tries

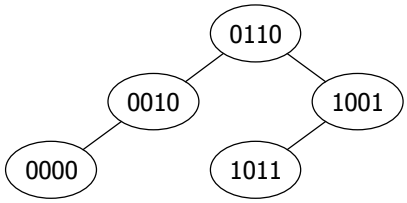
- Bisher: Traversierung durch Baum wurde gelenkt durch Vergleich zwischen gesuchtem Key und Key im Knoten
- Idee: lenke Traversierung durch Bits/Ziffern/Zeichen im Key
- Ab jetzt: Schlüssel sind Bitketten / Zeichenfolgen
  - feste Länge -> 0110, 0010, 1010, 1011
  - variable Länge -> 01$, 00$, 101$, 1011$
- Anwendung – IP-Routing, Paket-Klassifizierung, Firewalls
  - IPv4 – 32 bit IP-Adresse
  - IPv6 – 128 bit IP-Adresse

Digitale Suchbäume (DST = *digital search trees*)

- Annahme: feste Anzahl an Bits
- Konstruktion:
  - Wurzel enthält irgendeinen Schlüssel
  - alle Schlüssel, die mit 0 beginnen, sind im linken Unterbaum
  - alle Schlüssel, die mit 1 beginnen, sind im rechten Unterbaum
  - linker und rechter Unterbaum sind jeweils Unterbäume der verbleibenden Schlüssel

Beispiel

- Starte mit leerem Suchbaum:



  - füge den Schlüssel 0000 ein

- Bemerkungen:
  - Aussehen des Baumes hängt von der Reihenfolge des Einfügungens ab!
  - Es gilt nicht: Keys (ilnker Teilb.) < Key in Wurzel < Keys(rechter Teilb.) !
  - Es gilt aber: Keys (linker Teilbaum) < Keys(rechter Teilbaum)

- Suchen:

```
vergleiche gesuchten Key mit Key im Knoten
falls gleich:
    fertig
sonst:
    vergleiche auf der i-ten Stufe das i-te Bit
    1 → rechter Unterbaum
    0 → linker Unterbaum
```

- Anzahl der Schlüsselvergleiche: O(Höhe) = O(#Bits pro Schlüssel)
- Komplexität aller Operationen (Suchen, Einfügen, Löschen) : O((#Bits pro Schlüssel)^2)
- hohe Komplexität wenn Schlüssellänge sehr groß

Python-Code

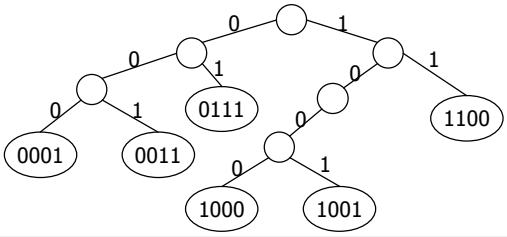
```
def digit(value, bitpos):
    return (value >> bitpos) & 0x01
```

```
def searchR(node, key, d):
    if node == None:
        return None
    if key == node.item.key:
        return node.item
    if digit(key, d) == 0:
        return searchR(node.left, key, d+1)
    else:
        return searchR(node.right, key, d+1)
```

```
class DigitalSearchTree:
    . . .
    def search(self,key):
        return searchR(self.head, key, 0)
```

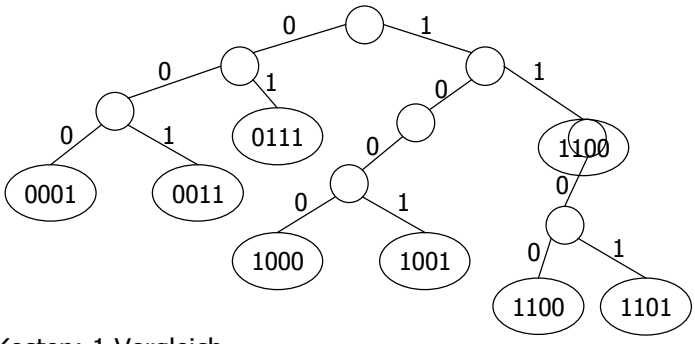
Binäre Tries

- Ziel: höchstens 1 Schlüsselvergleich pro Operation
- Zum Begriff:
  - *Information Re**trie**val*
  - Aussprache wie "*try*"
- 2 Arten von Knoten
  - Verzweigungsknoten: linke und rechte Kindsknoten, keine Datenfelder
  - Elementknoten: keine Kindsknoten, Datenfeld mit Schlüssel



Einfügen

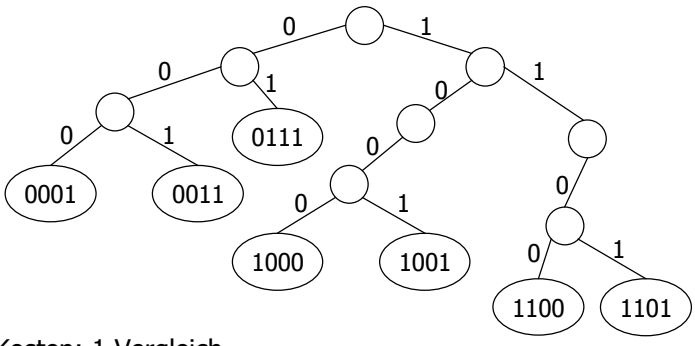
- Füge Schlüssel 1101 ein:



- Kosten: 1 Vergleich

Entfernen

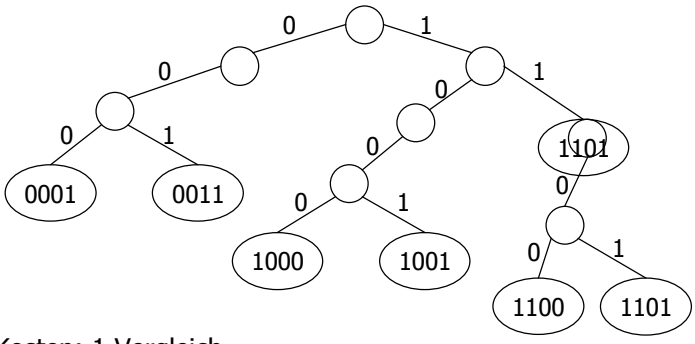
- Entferne Schlüssel 0111:



- Kosten: 1 Vergleich

Entfernen

- Entferne Schlüssel 1100:



- Kosten: 1 Vergleich

Implementierung

```
class Trie:
    . . .
    def insert(self,item):
        self.head = insertR(self.head, item, 0)
```

```
def insertR(node, item, d):
    if node == None:
        return TrieNode(item)
    if (node.left == None) and (node.right == None):
        return split( TrieNode(item), node, d )
    if digit(item.key, d) == 0:
        node.left = insertR(node.left, item, d+1)
    else:
        node.right = insertR(node.right, item, d+1)
    return node
```



```
def split(nodeP, nodeQ, d):
    nodeN = TrieNode(None)
    splitcase = 2 * digit(nodeP.item.key, d)
                  + digit(nodeQ.item.key, d)
    if   splitcase == 0: # 00
        nodeN.left  = split(nodeP, nodeQ, d+1)
    elif splitcase == 3: # 11
        nodeN.right = split(nodeP, nodeQ, d+1)
    elif splitcase == 1: # 01
        nodeN.left  = nodeP
        nodeN.right = nodeQ
    elif splitcase == 2: # 10
        nodeN.left  = nodeQ
        nodeN.right = nodeP
    else:
        print "Can't happen!"
return nodeN
```