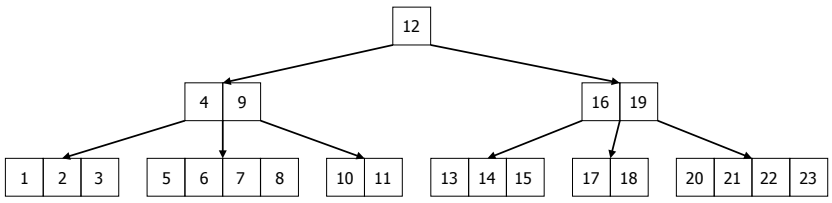
Bemerkungen

- CLRS benutzt eine etwas andere Definition:
  - Für einen B-Baum vom Grad t gilt:
    - jeder Knoten, außer der Wurzel, hat mindestens t-1 Schlüssel, also mindestens t Kinder
    - jeder Knoten besitzt höchstens $2t$-1 Schlüssel, also höchsten $2t$ Kinder
  - Formeln müssen „angepasst“ werden, damit sie stimmen
  - die Prüfung läuft gemäß den Folien, bei Unsicherheiten die Definition angeben

Eine Klasse von B-Bäumen

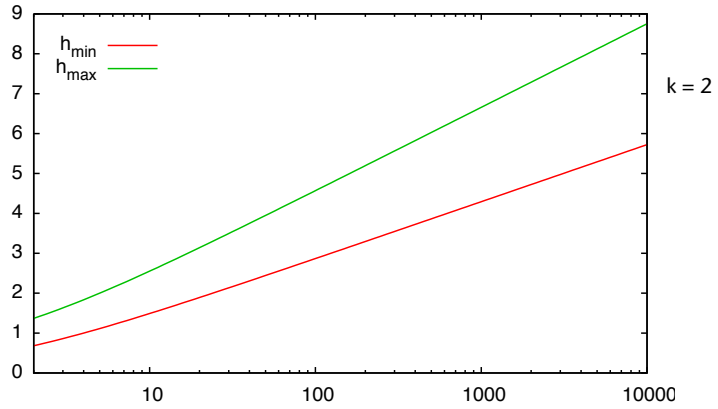
- Die Klasse $\tau(k,h)$ bezeichne alle B-Bäume mit dem Parameter k und der Höhe h ($h \geq 0$ und $k>0$)
  - Höhe der Wurzel = 1
- Beispiel: B-Baum der Klasse $\tau(2,3)$



Höhe eines B-Baumes

- Für die Höhe eines B-Baumes mit *n* Schlüsseln gilt:

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



Erklärung

- In einem minimalen B-Baum hat jeder Knoten die kleinstmögliche Anzahl an Kindern (= k+1); also gilt für die min. Anzahl Knoten

$$N_{\min}(k,h) = 1 + 2 + 2\left((k+1) + (k+1)^2 + \cdots + (k+1)^{h-2}\right)$$
$$= 1 + 2\sum_{i=0}^{h-2}(k+1)^i = 1 + 2\frac{(k+1)^{h-1}-1}{k}$$

- In einem maximalen B-Baum hat jeder Knoten die größtmögliche Anzahl an Kindern (= 2k+1); also gilt für die max. Anzahl Knoten

$$N_{\max}(k,h) = 1 + (2k+1) + (2k+1)^2 + \cdots + (2k+1)^{h-1}$$
$$= \sum_{i=0}^{h-1}(2k+1)^i = \frac{(2k+1)^h - 1}{2k}$$

- Die Höhe definiert eine obere und untere Schranke für die Anzahl der Knoten $N(B)$ eines beliebigen B-Baumes der Klasse $\tau(k,h)$:

$$1 + 2\frac{(k+1)^{h-1}-1}{k} \leq N(B) \leq \frac{(2k+1)^h-1}{2k} \quad \text{für } h \geq 1$$
$$N(B) = 0 \quad \text{für } h = 0$$

- Von $N_{min}(B)$ und $N_{max}(B)$ lässt sich ableiten:

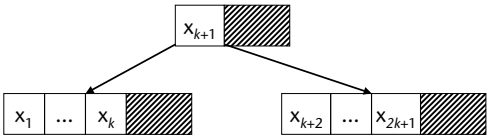
$$n \geq n_{min} = 1 + 2\frac{(k+1)^{h-1}-1}{k}k = 2(k+1)^{h-1} - 1$$
$$n \leq n_{max} = \frac{(2k+1)^h-1}{2k}2k = (2k+1)^h - 1$$

- Somit gilt für die Anzahl Keys n in einem B-Baum:

$$2(k+1)^{h-1} - 1 \leq n \leq (2k+1)^h - 1$$

Algorithmus zum Einfügen in einen B-Baum

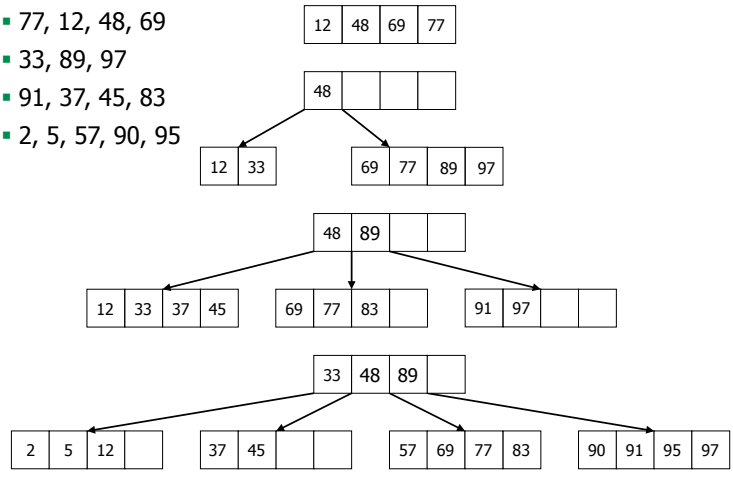
- Füge anfangs in ein leeres Feld der Wurzel ein
- Die ersten 2*k* Schlüssel werden sortiert in die Wurzel eingefügt
- Der nächste, der (2*k*+1)-te, Schlüssel passt nicht mehr in die Wurzel und erzeugt einen sog. "Überlauf" (*overflow*)
- Die Wurzel wird geteilt (ein sog. "Split"):
  - Jeder der beiden Kinder bekommt *k* Keys
    - Die ersten *k* Schüssel kommen in den linken Unterbaum
    - Die letzen *k* Schüssel kommen in den rechten Unterbaum
  - Die Wurzel bekommt 1 Key
    - Der Median der (2*k*+1) Schlüssel wird zum neuen Separator

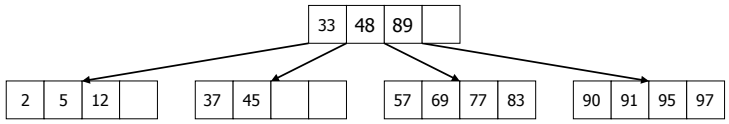


- Neue Schlüssel werden in den Blättern sortiert gespeichert
- Läuft ein Blatt über ($2k$+1 Keys), wird ein Split durchgeführt:
  - Das Blatt wird geteilt, ergibt 2 neue Blätter à k Keys;
  - diese werden im Vaterknoten anstelle des ursprünglichen Blattes verlinkt;
  - der Median-Key wandert in den Vaterknoten.
- Ist auch der Vaterknoten voll, wird das Splitten fortgesetzt
  - Im worst-case bis zur Wurzel

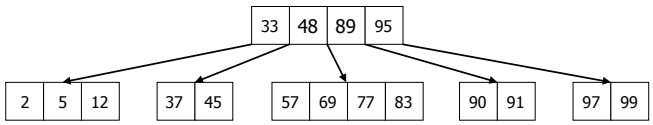
Beispiel

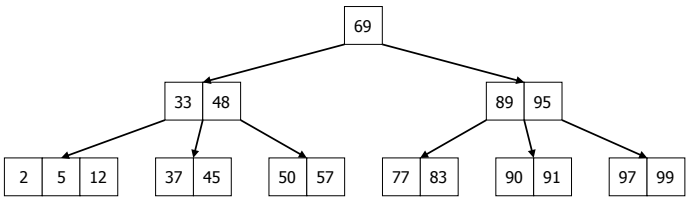
- k = 2 → 2 ≤ #Schlüssel ≤ 4 (außer Wurzel)
- Einzufügende Schlüssel:
  - 77, 12, 48, 69
  - 33, 89, 97
  - 91, 37, 45, 83
  - 2, 5, 57, 90, 95





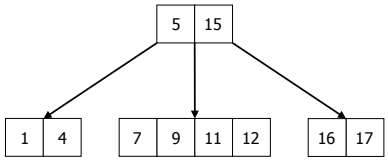
- Einzufügende Schlüssel:
  - 99
  - 50

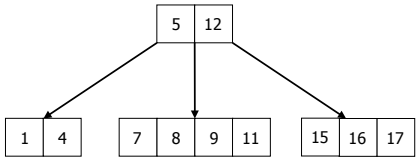


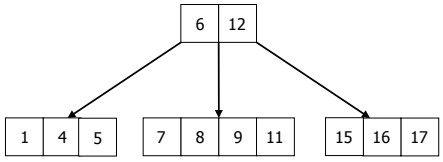


Der B*-Baum

- Variante des B-Baumes:
  - Selbe Datenstruktur
  - Algo zum Einfügen ist variiert
- Idee: zuerst Ausgleich mit Nachbarknoten statt Split
- Beispiel:
  - Füge ein:
  - 8
  - 6

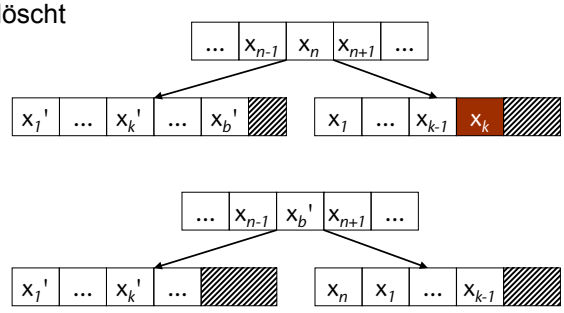




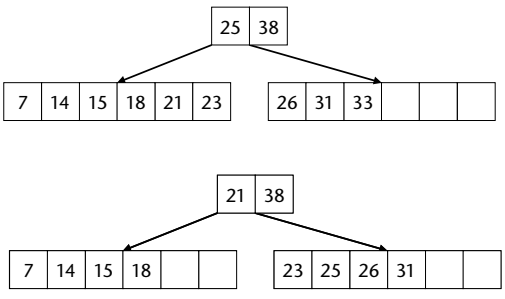


Löschen in einem B-Baum

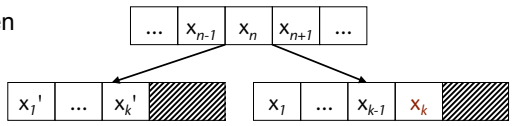
- Bedingung: minimale Belegung eines Knotens erhalten
  - Bei einem sog. *Underflow* muß ein Ausgleich geschaffen werden
- Methoden:
  - Rotation = Ausgleich mit Nachbarknoten
  - Merge = Gegenteil von Split
- Beispiel: x_k wird gelöscht
  - Wende Rotation an
  - Key x_b' wird in den Vaterknoten verschoben, Key x_n kommt in den rechten Knoten (mit k-1 Knoten)



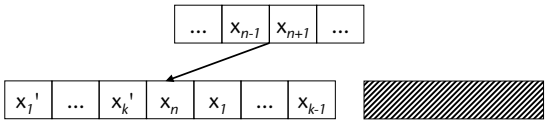
- Um wiederholte Rotationen (durch neue Löschoperationen) zu vermeiden, kann man mehr als einen Key rotieren
  - Beide Knoten haben danach ungefähr die gleiche Key-Anzahl
- Beispiel: 33 soll entfernt werden (k = 3)



- Zweite Art der Underflow-Behandlung: Merge
  - Wird benutzt, wenn es einen Underflow gibt und beide Nachbarknoten genau *k* Keys haben



- Beispiel: x_k löschen



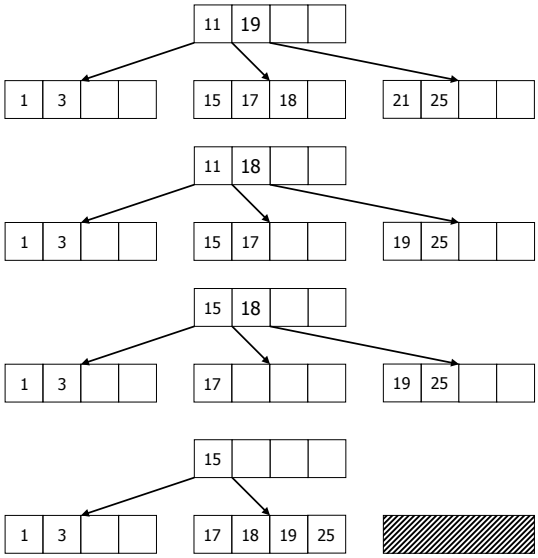
  - Die beiden Nachbarblätter und der Separator werden ge-merge-t; ergibt genau $k + 1 + (k-1) = 2k$ Keys

Allgemeiner Lösch-Algorithmus

- x = zu löschendes Element
- Suche x; unterscheide 2 Fälle:
  1. x ist in einem Blatt:
     - # Keys ≥ *k* (nach dem Löschen) → OK
     - # Keys = *k*-1 und # Keys in einem (direkten) Nachbarblatt > *k* → Rotation
     - Sonst → Merge
  2. x ist in einem internen Knoten:
     a) Ersetze den Separator x durch x_b oder x_1' des linken bzw. rechten Kind-Knotens (= nächstgößerer oder -kleiner Schlüssel)
     b) Lösche diesen Seperator (x_b oder x_1') aus dem entsprechenden Kind-Knoten (Rekursion)

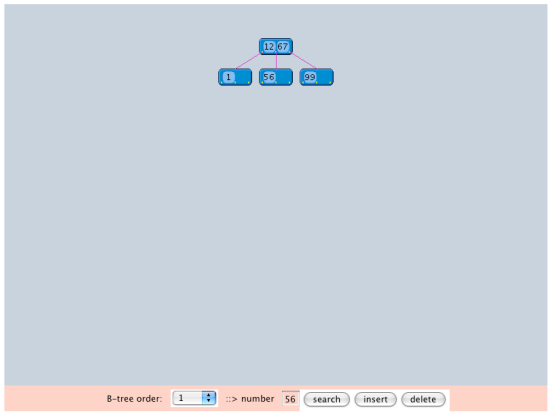
Beispiel (k = 2)

- Lösche:
- 21
- 11 → Underflow
- → Merge



Demo

B-Tree animation applet:



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

Komplexitätsanalyse für B-Bäume

- Anzahl der Key-Zugriffe:
  - Z.B. 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-Operationen auf die Platte
- Annahmen: jeder Knoten benötigt nur 1 Zugriff auf den externe Speicher und jeder modifizierte Knoten wird nur 1x geschrieben
- Schranken für die Anzahl Lesezugriffe f beim Suchen (f = „fetch“):

$$f_{\min} = 1 \qquad \text{Key ist in der Wurzel}$$

$$f_{\max} = h \in O\left(\log_k(n)\right) \qquad \text{Key ist in einem Blatt}$$

$$h - \tfrac{1}{k} \leq f_{\text{avg}} \leq h - \tfrac{1}{2k} \qquad \text{ohne Beweis}$$

Kosten für Insert

- f = Anzahl "Fetches", w = Anzahl "Writes"
- Unterscheide 2 extremale Fälle:
  - Kein Split: $f_{\min}$ = h, $w_{\min}$ = 1
  - Wurzel-Split: $f_{\max}$ = h, $w_{\max}$ = 2h+1
- Abschätzung der Split-Wahrscheinlichkeit:
  - Wahrscheinlichkeit, daß ein Knoten gesplittet werden muß = Wahrscheinlichkeit, daß der Knoten exakt $2k$ viele Keys enthält
  - Sei X = Anzahl Keys in einem Knoten
  - Es gilt: $X \in \{k, ..., 2k\}$
  - Für die Split-Wahrscheinlichkeit gilt also:

$$p_{\text{split}} = \frac{\text{Anzahl günstige}}{\text{Anzahl mögliche}} = \frac{1}{k}$$

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

$$
\begin{aligned}
w_{\text{avg}} &= 1 + p_{\text{split}} + \; p_{\text{split}}(1 + p_{\text{split}}) + \; p_{\text{split}}^2(\ldots) + \; \ldots \\
&= (1 + p_{\text{split}}) \sum_{i=0}^{h} p_{\text{split}}^i \\
&\leq (1 + p_{\text{split}}) \sum_{i=0}^{\infty} p_{\text{split}}^i \\
&= (1 + p_{\text{split}}) \frac{1}{1 - p_{\text{split}}} \\
&= \frac{k+1}{k-1}
\end{aligned}
$$

Kosten für Löschen

- Mehrere Fälle:
  1. Best-Case: kein „Underflow“
     f_{min} = h w_{min} = 1
  2. "Underflow" wird durch Rotation beseitigt (keine Fortpflanzung):
     f_{rot} = h+1 w_{rot} = 3
  3. Merge ohne Fortpflanzung:
     f_{mix} = h+1 w_{mix} = 2
  4. Worst-Case: Merge 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 Merge:
  - 1 Mix-Operation = Elemente eines Knotens in Nachbarknoten mergen 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:

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

Digitale Suchbäume & Binäre Tries

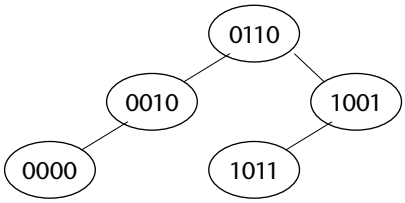
- Bisher: Traversierung durch Baum wurde gelenkt durch Vergleich zwischen gesuchtem Key und Key im Knoten
- Idee: lenke die Traversierung durch Bits/Ziffern/Zeichen im Key
  - Vergleiche die 2 großen Kategorien der Sortieralgorithmen
- Ab jetzt: Schlüssel sind Bitketten / Zeichenfolgen
  - Bei fester Länge → 0110, 0010, 1010, 1011
  - Bei variabler Länge → 01$, 00$, 101$, 1011$
- Anwendungen: z.B. 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 (= rekursive Definition):
  - Die Wurzel enthält irgendeinen Key
  - Alle Keys, die mit 0 beginnen, sind im linken Unterbaum
  - Alle Keys, die mit 1 beginnen, sind im rechten Unterbaum
  - Linker und rechter Unterbaum sind jeweils Digitale Suchbäume ihrer Keys

Beispiel

- Starte mit leerem Suchbaum:



  - füge den Schlüssel 0000 ein

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

- Algorithmus zum Suchen:

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

- Anzahl der Schlüsselvergleiche = O(Höhe) = O(# Bits pro Key)
- Komplexität aller Operationen (Suchen, Einfügen, Löschen) :
  $O((\#\text{ Bits pro Schlüssel})^2)$
- Ergibt hohe Komplexität wenn die Schlüssellänge sehr groß ist

Python-Implementierung

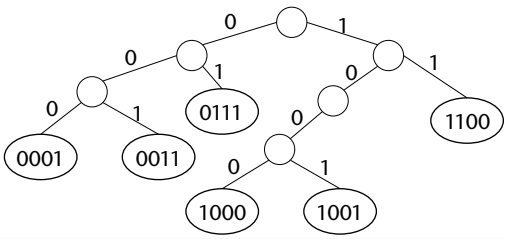
```
def digit(value, bitpos):
    return (value >> bitpos) & 0x01
```

```
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.root, key, 0)
```

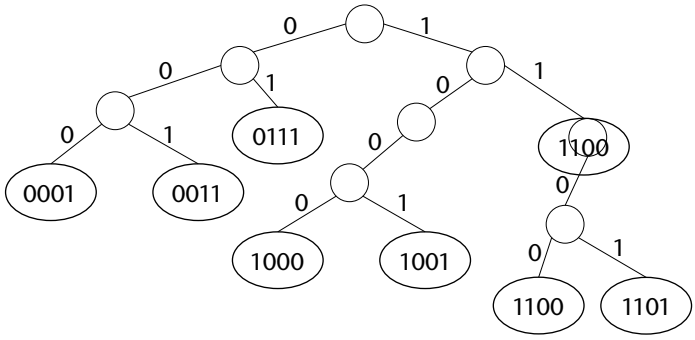

Binäre Tries

- Ziel: höchstens 1 kompletter Key-Vergleich pro Operation
- Zum Begriff:
  - "Trie" kommt von Information Re***trie***val
  - Aussprache wie "try"
- Es gibt 2 Arten von Knoten:
  - Verzweigungsknoten: hat nur linke und rechte Kindknoten, keine Keys
  - Elementknoten: keine Kindknoten, Datenfeld mit Schlüssel



Beispiel: Einfügen

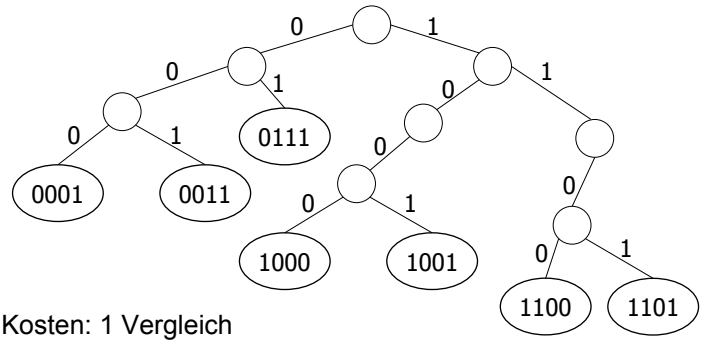
- Füge Schlüssel 1101 ein:



- Kosten: 1 Vergleich

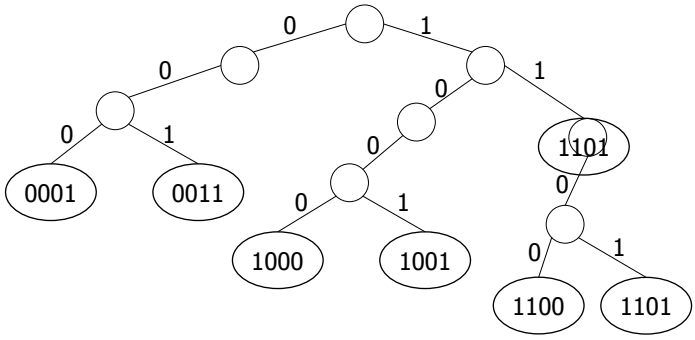
Beispiel: Entfernen

- Entferne Schlüssel 0111:



- Kosten: 1 Vergleich

- Entferne Schlüssel 1100:



- Kosten: 1 Vergleich

Implementierung

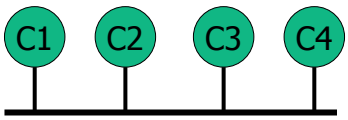
```
class Trie:
    . . .
    def insert(self,item):
        self.root = insertR(self.root, item, 0)
```

```
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
```

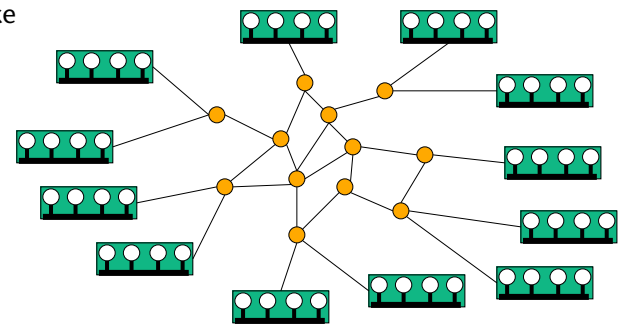
```
def split(nodeP, nodeQ, d):
    nodeN = TrieNode(None)
    splitcase = 2 * digit(nodeP.item.key, d)
                  + digit(nodeQ.item.key, d)
    if   splitcase == 0: # 00
        nodeN.left  = split(nodeP, nodeQ, d+1)
    elif splitcase == 3: # 11
        nodeN.right = split(nodeP, nodeQ, d+1)
    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
```


Anwendung: Routing in Netzwerken

- Netzwerk: miteinander verbundene Computer



- Internet = miteinander verbundene Netzwerke



Routing und Forwarding

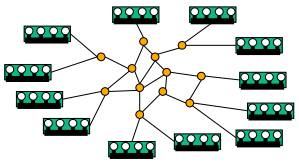
- Wie gelangt eine Nachricht von Computer A zu Computer B?
  - Beispiel Brief-Adresse: Straße/Hausnummer, Postleitzahl/Ort
    - 2 Schritte: zuerst die richtige Postleitzahl, dann das richtige Haus
  - Bei Computernetzwerken:
    - Zuerst muß das richtige Netzwerk ermittelt werden, dann der richtige Computer

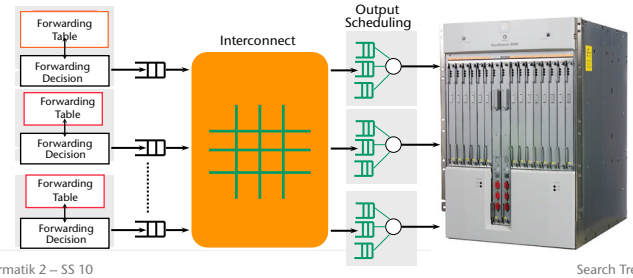
IP-Adressen

- Adresse eines Computers im Internet:
  - 32 Bits lang, meist in Punktnotation, z. B. 139.174.2.5
  - Besteht aus zwei Teilen: Netzwerkadresse und Host-Adresse
  - Beispiel: 139.174.2.5 =
    10001011101011100000001000000101
    Netzwerkadresse Host-Adresse
  - Anzahl der Bits für jeden Teil ist nicht festgelegt!
  - Um IP-Adressen richtig interpretieren zu können, muß man die Anzahl der Bits, mit denen das Netzwerk kodiert wird, kennen!
  - Wie man den Netzwerkanteil bestimmt, kommt demnächst
  - Länge der Netzwerkadresse bzw. die Netzwerkadresse selbst wird im folgenden Präfix heißen

Weiterleitung (Forwarding)

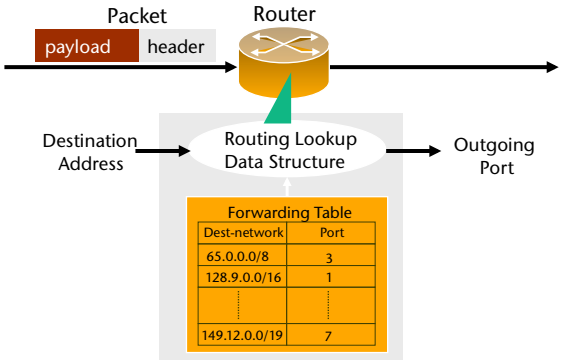
- Weiterleitung wird erledigt durch sog. *Router*
- Ein Router hat viele sog. *Ports*, die mit Computern oder anderen Routern verbunden sind
- Für eingehende Nachricht muß der Router entschieden, zu welchem Ausgangsport sie geschickt wird → hop-by-hop Weiterleitung
- Aufbau:





Der Router

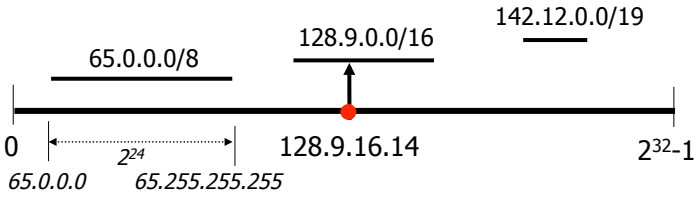
- Erstelle Routing-Tabelle mit "allen" Netzwerkadressen
- Routing-Tabelle verknüpft Netzwerkadressen mit Ausgangsports
- (Die Routing-Tabelle zu erstellen ist ein anderes Problem)



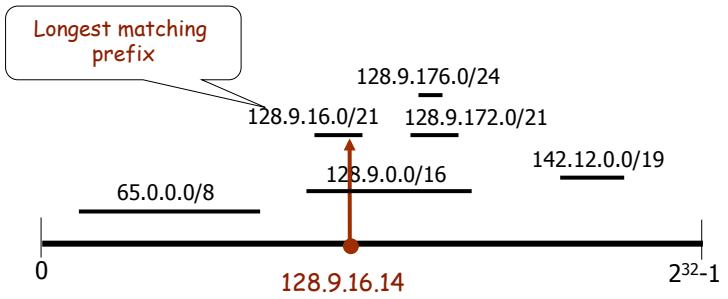
Beispiel einer Routing-Tabelle

Destination IP Prefix	Outgoing Port
65.0.0.0/8	3
128.9.0.0/16	1
142.12.0.0/19	7

Präfix-Länge



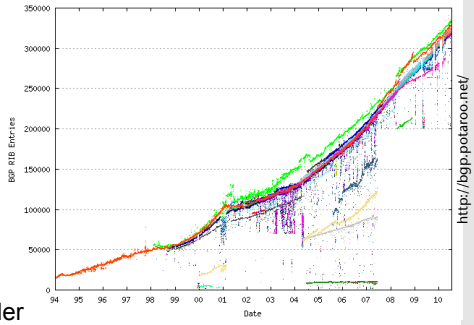
- Problem: Präfixe können überlappen!



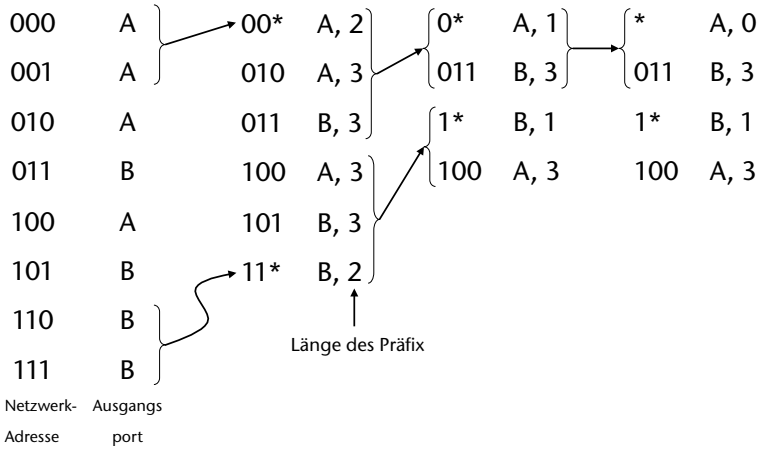
- Eindeutigkeit wird durch Longest-Prefix-Regel gewährleistet:
  1. Vergleiche Adresse mit allen Präfixen
  2. Gibt es mehrer Präfixe, die matchen (bis zu ihrer Länge), wähle denjenigen Präfix, der am längsten ist (d.h., am "genauesten")

Anforderungen

- Geschwindigkeit:
  - Muß ca. 500,000…15 Mio Pakete pro Sekunde routen/forwarden
  - Update der Routing-Tabelle:
    - Bursty: einige 100 Routes auf einen Schlag hinzufügen/löschen → effiziente Insert/Delete-Operationen
    - Frequenz: im Mittel ca. 100 Updates / Sekunde
- Speicherbedarf:
  - Anzahl der Netzwerk-adressen ist groß
  - Hätte man für jede mögliche Netzwerkadresse eine Zeile in der Tabelle, dann bräuchte jeder Router eine große Menge an Speicher
  - Das wiederum hätte Auswirkungen auf die Geschwindigkeit



Beispiel einer Routing-Tabelle



Problem

- Gegeben:
  - Menge von n Präfixen plus Länge jedes Präfix',
  - String zum Vergleich
- Ziel: effiziente Algorithmen zur
  - Bestimmung des *Longest Matching Prefix*
  - Einfügen von Präfixen in die Tabelle
  - Löschen von Präfixen in der Tabelle

1. (Brute-Force) Ansatz: Lineare Suche

- Jeden Eintrag prüfen
- Sich den longest match merken
- Zeitkomplexität Einfügen: O(1)
- Zeitkomplexität Löschen: O(n)
- Average-Case lookup: O(n/2) = O(n)
- Worst-Case lookup: O(n)
- Speicherkomplexität: O(n)

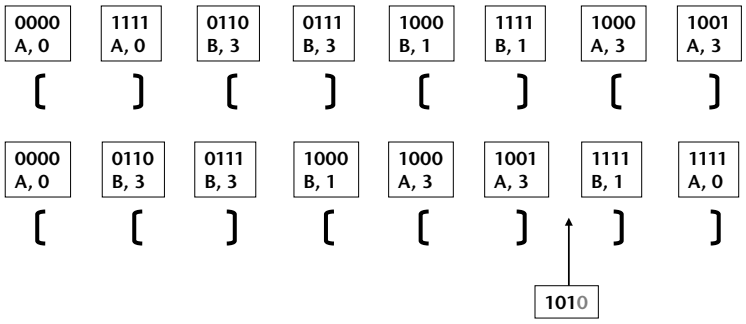
2. Ansatz: Sortierte Bereiche

- Erstelle von jedem Tabelleneintrag zwei "Marker":
  - Jeder Marker ist 1 Bit länger als der längste Präfix
  - Linker Marker ([) = Tabelleneintrag, aufgefüllt mit 0-en
  - Rechter Marker (]) = Tabelleneintrag, aufgefüllt mit 1-en
  - Beide Marker zusammen definieren den Bereich, den der Präfix abdeckt (abzüglich eventueller Intervalle in dessen Innerem, die von längeren Präfixen belegt werden)
  - Assoziiere Präfixlänge und Präfixe mit den Markern
  - Sortiere Marker

Beispiel

*	A, 0
011	B, 3
1*	B, 1
100	A, 3

Marker sind 4 Bits lang



Komplexität

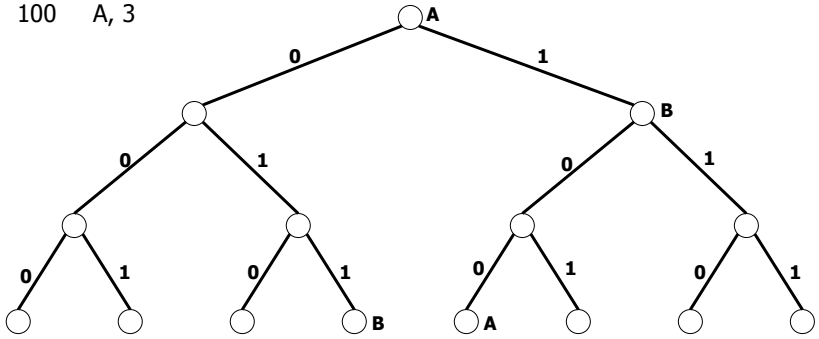
- Zeitkomplexität Einfügen: O(n log(n))
- Zeitkomplexität Löschen: O(n log(n))
- Zeitkomplexität Lookup: O(log(n))
- Speicherkomplexität: O(2n)

3. Ansatz: Lösung mit DST / Trie

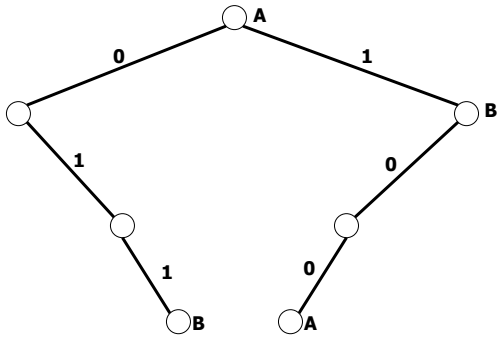
- Wird in aktuellen Routern verwendet
- Erstelle einen Binärbaum
- Jede Stufe des Baumes wird jeweils mit dem nächsten Bit indiziert
- Der Baum wird nur so weit aufgebaut, wie nötig
- Bezeichne jeden Knoten im Baum mit dem Port, der dem Präfix zugeordnet ist

Beispiel

*	A, 0
011	B, 3
1*	B, 1
100	A, 3



*	A, 0
011	B, 3
1*	B, 1
100	A, 3



Komplexität

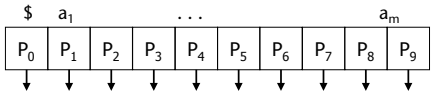
- b = maximale Anzahl Bits der Einträge
- Zeitkomplexität Lookup: O(b)
- Zeitkomplexität Einfügen: O(b)
- Zeitkomplexität Löschen: O(b)

Allgemeine Tries

- Verwaltung von Keys verschiedener Länge:
  - Füge spezielles Zeichen (z.B. $) zum Alphabet hinzu (Terminierungszeichen, Terminator)
  - Füge dieses Zeichen am Ende jedes Keys hinzu
  - Effekt: die Menge der Keys wird präfixfrei, d.h., kein Key ist Präfix eines anderen
  - (Das Ganze ist nur ein "Denkhilfsmittel"!)
- Keys werden als Zeichenfolgen eines Alphabets $\$,a_1,\ldots,a_m$ ausgedrückt:
  - Zahlen: m=10+1
  - Buchstaben: m=26+1
  - alpha-numerische Zeichen: m=36+1

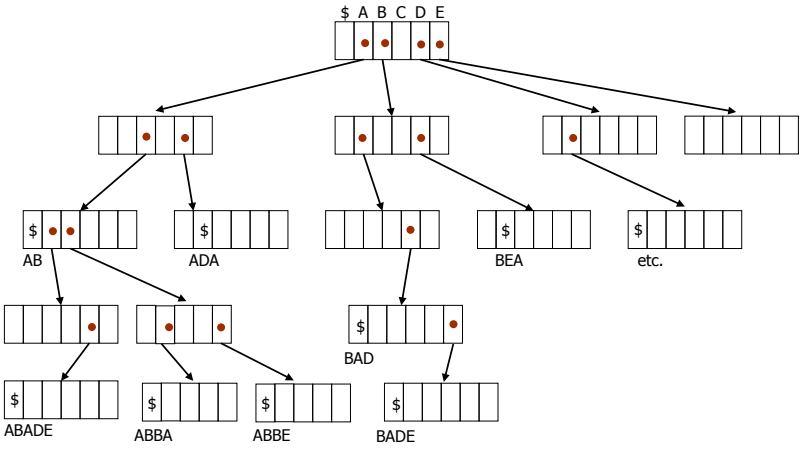
m-stufiger Trie

- Ein Knoten eines Trie über Alphabet der Mächtigkeit *m* ist ein Vektor mit *m*+1 Zeigern:
  - Jedes Vektorelement repräsentiert ein Zeichen des Alphabets
  - Für ein Zeichen a_k an der i-ten Stelle in einem Key gibt es einen Zeiger an der Stelle k in einem Vektor auf der i-ten Stufe des Baumes
  - Dieser Zeiger
    - zeigt auf einen Unterbaum für die "normalen" Zeichen ($P_1 \ldots P_m$), oder
    - es ist nur ein, von NULL verschiedener, Dummy-Wert (Fall P_0; zeigt also an, daß es einen Key gibt, der hier endet)



Beispiel

- Ein Trie für die Schlüssel AB, ABAD, ABBA, ADA, BAD, BADE, DA, DADA, EDA, EDDA, EDE über dem Alphabet {A,…,E} und $ als Terminator



Bemerkungen

- Grundlegende Struktur eines Tries (mit fester Schlüssellänge) ist einem B-Baum ähnlich:

m-Wege-Baum	m-way Trie
Key-Menge wird durch Separatoren in die Unterbäume aufgeteilt	Key-Menge wird durch "Selektoren" in die Unterbäume aufgeteilt
Blätter zeigen auf Nutzdaten	Knoten mit gesetztem Terminator zeigen auf Daten

- Belegung der Knoten nimmt zu den Blättern hin ab (schlechte Speicherausnutzung)
- Analog gibt es auch einen m-Wege-DST

Suchen im Multi-Way Trie

- Schema: verfolge den für das aktuelle Zeichen im Key "zuständigen" Zeiger

```
k = Key
Index i ← 0
starte bei x ← Wurzel
while i < Länge(k):
    teste Zeiger x[ k[i] ]
    if Zeiger == None:
        Key k ist nicht im Trie
    else:
        x ← x[ k[i] ]
    i += 1
teste, ob Flag in x["$"] gesetzt
```

- Anzahl der durchsuchten Knoten = Länge des Schlüssels + 1
- Vorteil: Such-Komplexität ist unabhängig von der Anzahl der gespeicherten Schlüssel

Einfügen in einen Trie

- Analog zum Suchen: verfolge Zeiger
- Falls Key zu Ende: teste Terminator-Flag ($-Feld)
  - Falls schon gesetzt → Key war schon im Trie
  - Sonst: setzen, Daten dort speichern
- Falls Zeiger nicht vorhanden (Key noch nicht zu Ende): erzeuge neue Knoten
- Spezialbehandlung, falls man die reinen "Terminierungsknoten" eingespart hat

Löschen aus einem Trie

- Baum durchsuchen, bis der Terminator ($) gefunden ist
- Terminator $ löschen
- alle Zeiger des Knotens überprüfen, ob sie alle auf NULL sind
  - nein → Lösch-Operation ist beendet
  - ja (alle sind NULL) → lösche den Knoten und überprüfe den Vaterknoten, falls der jetzt auch leer, dann wiederhole

Bemerkungen

- Die Struktur eines Trie's hängt nur von den vorliegenden Keys ab, nicht von der Reihenfolge der Einfügungen!
- Keine optimale Speichernutzung, weil die Knoten eine feste Länge haben, auch bei minimaler Belegung
- Häufig One-Way-Branching (z.B. BEA und ABADE)
  - Lösung: zeigt ein Zeiger auf einen Unterbaum, der nur einen Key enthält, wird der Key im Knoten gespeichert