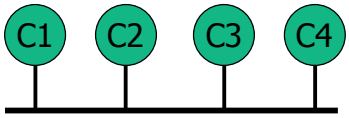
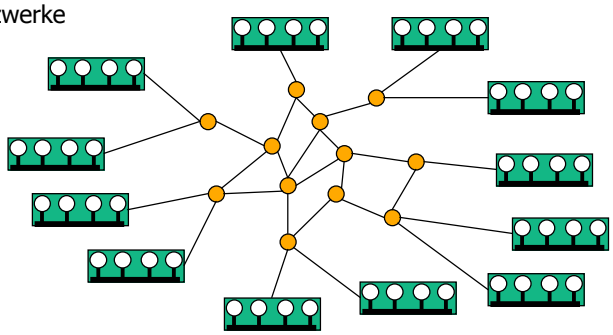
Anwendung: Routing in Netzwerken

- Netzwerk: miteinander verbundene Computer



- Internet: miteinander verbundene Netzwerke



Routing und Forwarding

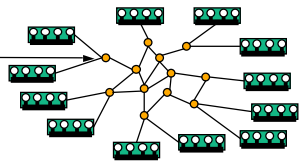
- Wie gelangt eine Nachricht von Computer A zu Computer B?
  - Beispiel Brief-Adresse: Straße/Hausnummer, Postleitzahl/Ort
    - 2 Schritte: zuerst die richtige Postleitzahl, dann das richtige Haus
  - Bei Computernetzwerken:
    - Zuerst muß das richtige Netzwerk ermittelt werden, dann der richtige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

IP-Adressen

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

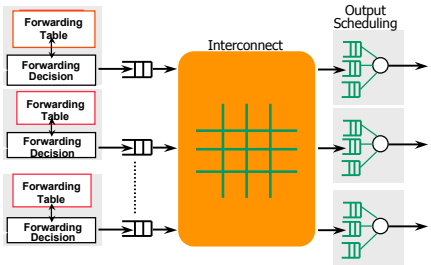
Weiterleitung (*Forwarding*)

- Weiterleitung wird erledigt durch sog. *Router*



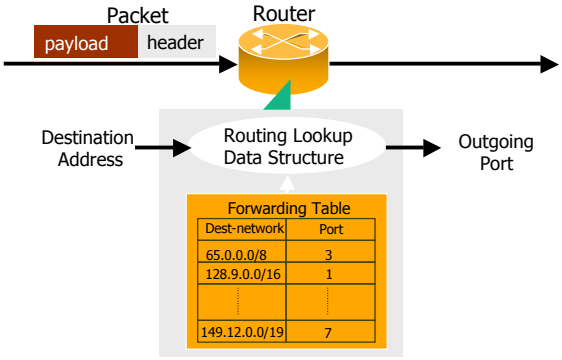
- Router hat viele sog. *Ports*, die mit Computern oder anderen Netzwerken verbunden sind
- Für eingehende Nachricht muß der Router entschieden, zu welchem Ausgangsport sie geschickt wird: hop-by-hop-Weiterleitung

- Aufbau:



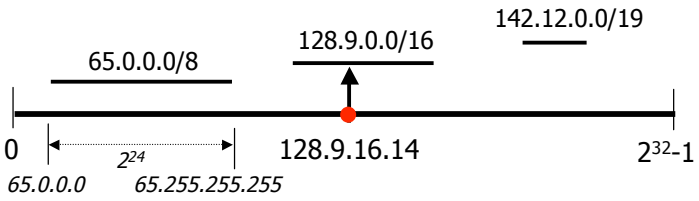
Routing

- In der „Mitte" des Internets spielt nur der Netzwerkteil eine Rolle
- Erstelle Routing-Tabelle mit allen Netzwerknummern
- Routing-Tabelle verknüpft Netzwerknummern mit Ausgangsports
- Die Routing-Tabelle zu erstellen ist ein anderes Problem

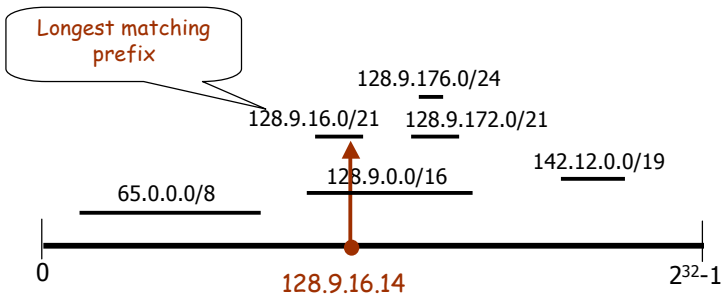


Beispiel: Routing-Tabelle

Destination IP Prefix	Outgoing Port
65.0.0.0/8 (Präfix-Länge)	3
128.9.0.0/16	1
142.12.0.0/19	7



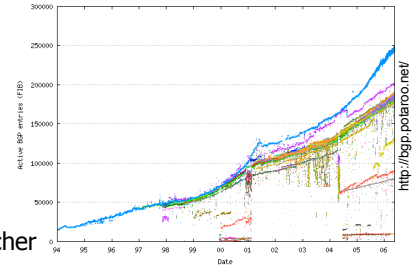
- Problem: Präfixe können überlappen!



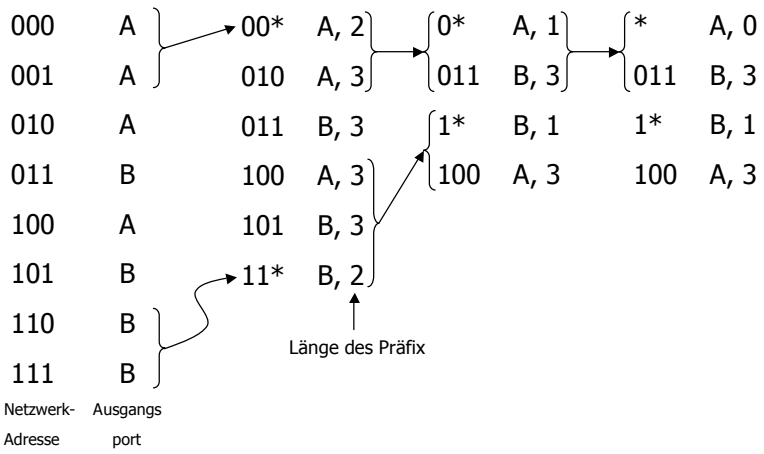
- Eindeutigkeit wird durch Longest-Prefix-Regel gewährleistet:
  1. Vergleiche Adresse mit allen Präfixen
  2. Gibt es mehrer Präfixe, die matchen (bis zu ihrer Länge), wähle denjenigen Präfix, der am längsten ist (d.h., am "genauesten")

Anforderungen

- Geschwindigkeit:
  - Ermittlung der Route so schnell wie möglich
  - Update der Routing-Tabelle:
    - Bursty: einige 100 Routes auf einen Schlag hinzufügen/löschen → Insert/Delete-Operationen
    - Frequenz: im Mittel ca. 100 Updates / Sekunde
- Speicherbedarf:
  - Anzahl der Netzwerknummern ist groß
  - hätte man für jede mögliche Netzwerknummer eine Zeile in der Tabelle, dann bräuchte jeder Router eine große Menge an Speicher
  - das wiederum hat 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: $O(n)$

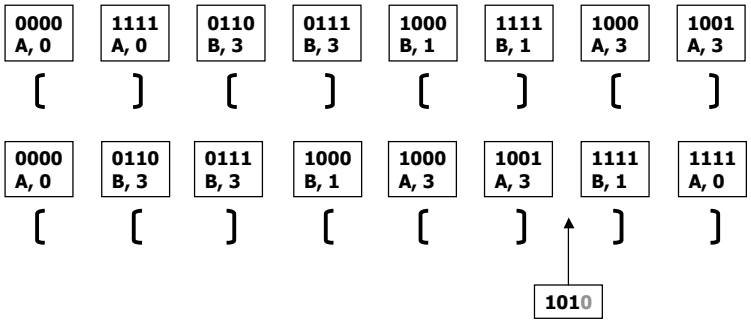
2. Ansatz: Sortierte Bereiche

- erstelle von jedem Tabelleneintrag zwei "Marker":
  - jeder Marker ist 1 Bit länger als der längste Präfix
  - linker Marker ([): mit 0 auffüllen
  - rechter Marker (]): mit 1 auffüllen
  - Beider Marker zusammen definieren den Bereich, den der Präfix abdeckt (abzüglich eventueller Intervalle in dessen Innerem, die von längeren Präfixen belegt werden)
  - verbinde 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

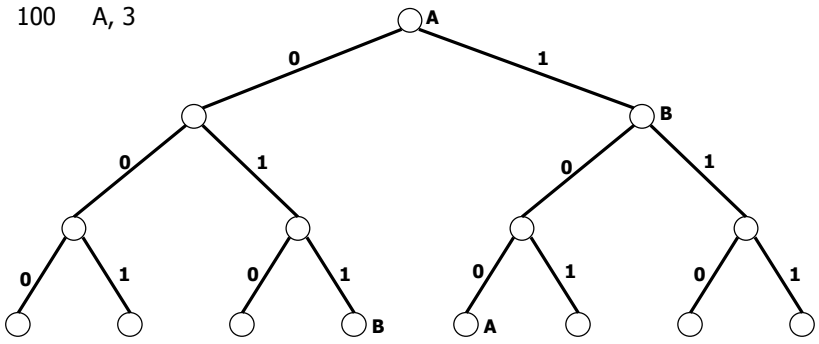
- Zeitkomplexität Einfügen: $O(n \log(n))$
- Zeitkomplexität Löschen: $O(n \log(n))$
- Zeitkomplexität Lookup: $O(\log(n))$
- Speicherkomplexität: $O(2n)$

3. Ansatz: Lösung mit DST / Trie

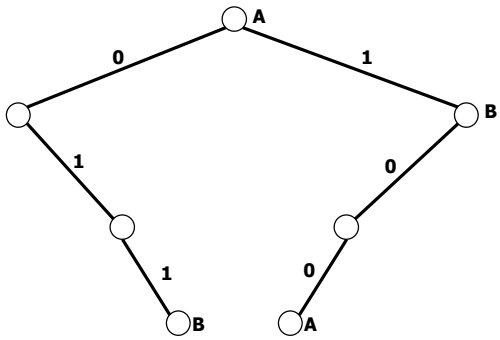
- wird in aktuellen Routern verwendet
- erstelle einen Binärbaum
- jede Stufe des Baumes wird jeweils mit dem nächsten Bit indiziert
- der Baum wird nur so weit aufgebaut, wie nötig
- bezeichne jeden Knoten im Baum mit dem Port, der dem Präfix zugeordnet ist

Beispiel

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



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



Komplexität

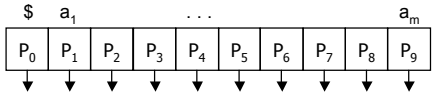
- b = maximale Anzahl Bits der Einträge
- Zeitkomplexität Lookup: $O(b)$
- Zeitkomplexität Einfügen: $O(b)$
- Zeitkomplexität Löschen: $O(b)$

Allgemeine Tries

- Verwaltung von Schlüsseln verschiedener Länge:
  - Füge spezielles Zeichen (z.B. $) zum Alphabet hinzu (Terminierungszeichen, *Terminator*)
  - Füge dieses Zeichen am Ende jedes Schlüssels hinzu
  - Effekt: die Menge der Schlüssel wird präfixfrei, d.h., kein Schlüssel 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

- 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 Schlüssel 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)



Bemerkungen

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

m-Wege-Baum	m-way Trie
Schlüssel durch Separatoren in die Unterbäume aufgeteilt	Schlüssel durch "Selektoren" in 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

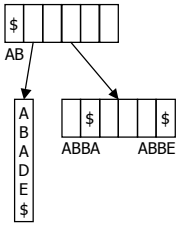
- Analog zum Suchen: verfolge Zeiger
- Falls Key zu Ende: teste Terminator-Flag ($-Feld)
  - Falls schon gesetzt → Key war schon im Trie
  - Sonst: setzen, Daten dort speichern
- Falls Zeiger nicht vorhanden (Key noch nicht zu Ende): erzeuge neue Knoten
- Spezialbehandlung, falls man die reinen "Terminierungsknoten"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

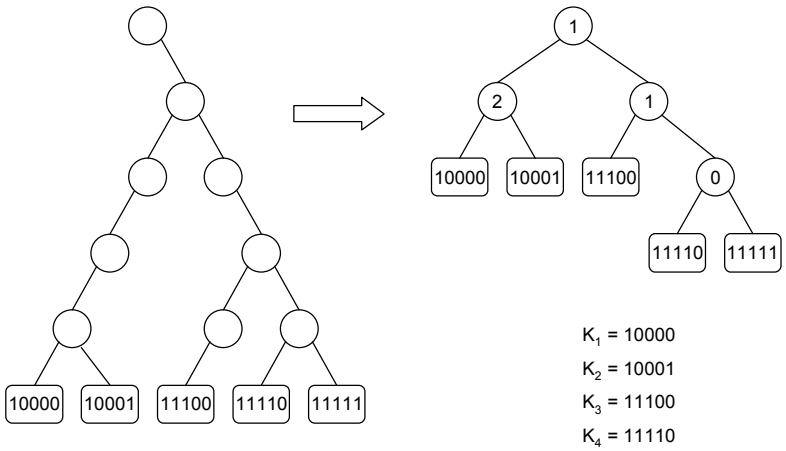
- Die Struktur eines Trie's hängt nur von den vorliegenden Schlüsseln ab, nicht von der Reihenfolge der Einfügungen!
- Keine optimale Speichernutzung, weil die Knoten eine feste Länge haben, auch bei minimaler Belegung
- Häufig One-Way-Branching (z.B. BEA und ABADE)
  - Lösung: zeigt ein Zeiger auf einen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



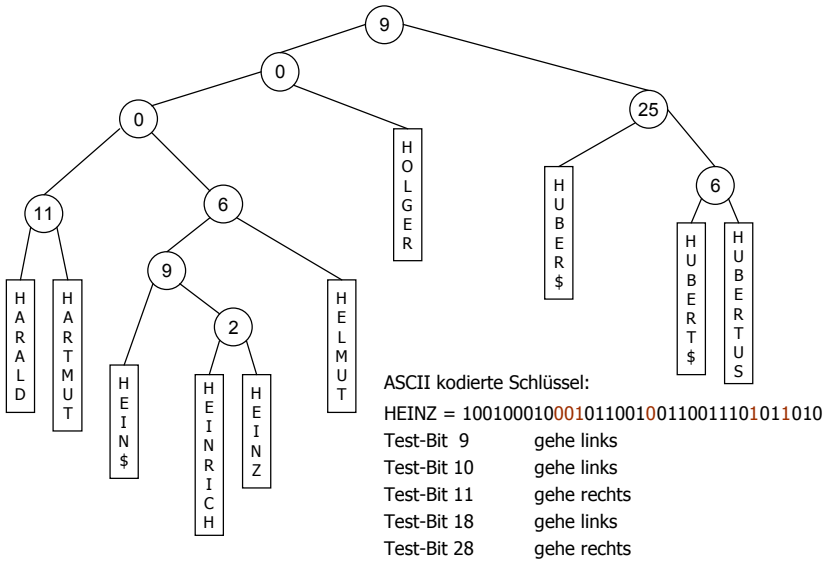
PATRICIA

- Variante der binären Tries:
  - vermeidet Pfade im Baum ohne Gabelung ("Einweg-Pfade")
  - hat nur einen Typ Knoten
  - beschleunigt Suche auf das Minimum
- PATRICIA = "Practical Algorithm to Retrieve Information Coded in Alphanumeric" [Morrison, 1968]
- Idee:
  - speichere an inneren Knoten die Anzahl Bits, die übersprungen werden können und nicht getestet werden brauchen (weil es sowieso keine Gabelung auf dem Pfad darunter gibt, d.h., alle Keys in diesem Teilbaum bis dahin den gleichen Präfix haben)
  - speichere Keys/Daten an ("irgendeinem") inneren Knoten

Einweg-Pfade vermeiden



K_1 = 10000
K_2 = 10001
K_3 = 11100
K_4 = 11110
K_5 = 11111



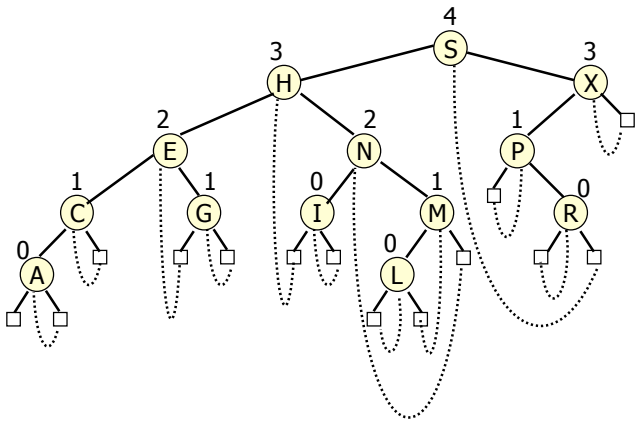
ASCII kodierte Schlüssel:

HEINZ = 10010001000101100100110011101011010

Test-Bit 9	gehe links
Test-Bit 10	gehe links
Test-Bit 11	gehe rechts
Test-Bit 18	gehe links
Test-Bit 28	gehe rechts
Test-Bit 31	gehe rechts, Blatt, vergleiche, OK

- Im folgenden folgende Modifikation:
  - Nummeriere Bits der Keys von rechts nach links (0 = erstes Bit)
  - Speichere im Knoten die Nummer desjenigen Bits, das an diesem Knoten getestet werden muß
    - Daher manchmal auch der Name *"crit bit tree"* für *"critical bit tree"*
  - Konsequenz: auf jedem Pfad durch den Baum von oben nach unten nehmen diese Nummern ab
- Lösung für zweites Problem (zwei verschiedene Knoten-Arten):
  - Speichere Keys (bislang in Blättern) in ("irgend einem") inneren Knoten
  - Geht gut, weil #Blätter = #innere Knoten + 1
  - Knoten haben jetzt 2 verschiedene Funktionen zu verschiedenen Zeiten
    - Separator während Traversierung
    - Daten-Container am Ende der Traversierung

Beispiel



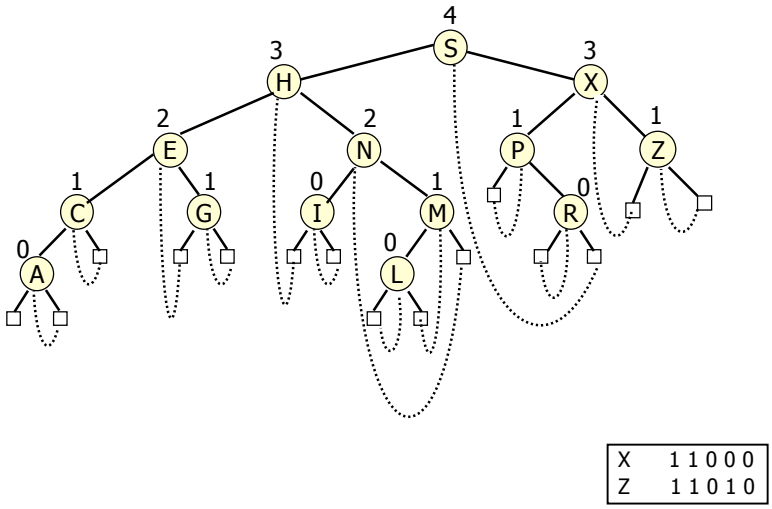
A	0 0 0 0 1
S	1 0 0 1 1
E	0 0 1 0 1
R	1 0 0 1 0
C	0 0 0 1 1
H	0 1 0 0 0
I	0 1 0 0 1
N	0 1 1 1 0
G	0 0 1 1 1
X	1 1 0 0 0
M	0 1 1 0 1
P	1 0 0 0 0
L	0 1 1 0 0

Suche in PATRICIAs

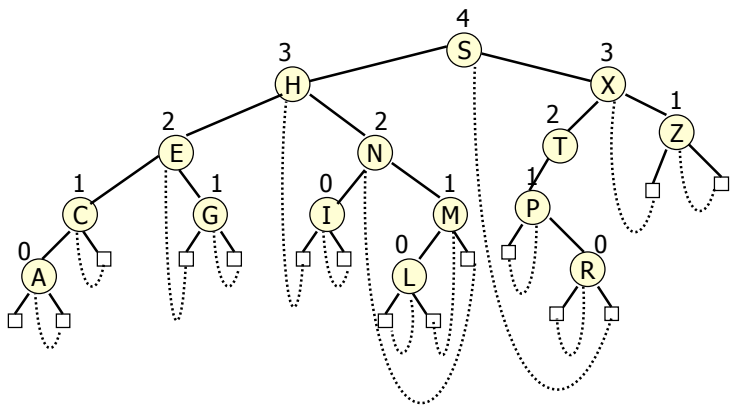
- Laufe im Baum abwärts, wobei man den Bitindex in jedem Knoten benutzt, um festzustellen, welches Bit im Schlüssel zu testen ist;
  - laufe nach rechts, falls das Bit =1 ist, und nach links, falls es =0 ist.
- Die Schlüssel in den Knoten werden auf dem Weg im Baum abwärts überhaupt nicht betrachtet!
- Schließlich wird ein aufwärts zeigender Pointer auf einen inneren Knoten vorgefunden: führe vollständigen Vergleich zwischen gesuchtem Key und Key in diesem inneren Knoten durch.
- Es ist leicht zu testen, ob ein Zeiger nach oben zeigt, da die Bitindizes in den Knoten (per Definition) kleiner werden, wenn man sich im Baum abwärts bewegt.

Einfügen in PATRICIAs

- 2 Fälle
- Zunächst: Äußeres Einfügen in einen Patricia-Baum
- Beispiel: Z = 11010 einfügen
  - rechten Zeiger von X (der ja nach oben zeigt) ersetzen durch einen Zeiger, der auf einen (neuen) "Testknoten" zeigt
  - neuen Testknoten erzeugen, der X und Z unterscheiden kann (Bit 1 testen, da X = 11000)
  - von dort aus linken Zeiger nach X hoch, rechten Zeiger auf Z hoch



- Zweiter Fall: Inneres Einfügen
- Etwas komplizierterer Fall:
  - Wenn der Key auf dem Weg im Inneren eingefügt werden muß
  - D.h., das Bit, das diesen Key von den anderen unterscheidet, wurde bei der Suche übersprungen
- Beispiel: T = 10100 einfügen
  - Suche endet bei P = 10000.
  - T und P unterscheiden sich in Bit 2, einer Position, die während der Suche übersprungen wurde. Die Forderung, daß die Bitindizes fallen müssen, wenn man sich im Baum abwärts bewegt, macht es notwendig, T zwischen X und P einzufügen, mit einem nach oben auf T selbst gerichteten Zeiger, der seinem eigenen Bit 2 entspricht.
  - Beachte: die Tatsache, daß Bit 2 vor dem Einfügen von T übersprungen wurde, impliziert, daß P und R den gleichen Wert von Bit 2 besitzen.



Eigenschaften

- Sedgewick: "Patricia stellt die Quintessenz der digitalen Suchmethoden dar"
- Knuth: "Patricia is a little tricky, and she requires careful scrutiny before all of her beauties are revealed."

- Haupttrick: Patricia identifiziert diejenigen Bits, die die Suchschlüssel von anderen unterscheiden, und baut sie in eine Datenstruktur (ohne überflüssige Knoten) ein, so daß man schnell von einem beliebigen Suchschlüssel zu dem einzigen Schlüssel in der Datenstruktur kommt, der gleich sein könnte.

- Satz:
  Ein Patricia-Trie, der aus N zufälligen Schlüsseln mit b Bits erzeugt wurde, hat N Knoten und erfordert für eine durchschnittliche Suche lg(N) Bitvergleiche.
- Bemerkung:
  - Die Länge der Schlüssel spielt keine Rolle!
  - Die o.g. Komplexität ist eine Bitkomplexität.
  - Bei allen anderen Suchmethoden ist die Länge der Schlüssel in irgendeiner Weise in die Suchprozedur "eingebaut" (z.B. beim Vergleich von kompletten Keys)