18.11.2010

ADS: große Menge von Worten speichern
↳ Bäume: nat. bin, AVL, B (log n Zugriff)
Hashtabelle: wenn richtig org.: konstante Zeit

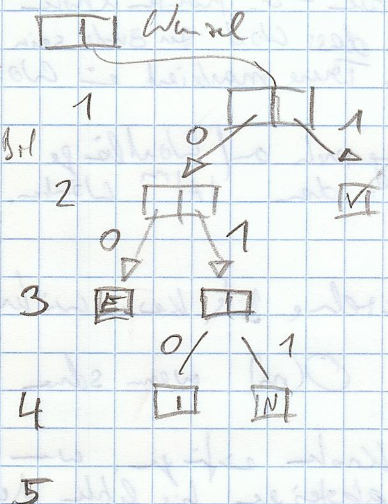
Ziel: größere Menge, mögl. effizient kompakte (Bsp. TS Suchmaschine, Rechtschreibprüfung, ...)

↳ TRIES   immer eindeutig

Variante: Radix-Trie: im Wesentl. binärer Baum - spezielle Form mit Routerknoten u. Datenknoten

	1 2 3 4 5
E	0 0 1 0 1
I	0 1 0 0 1
N	0 1 1 1 0
V	1 0 1 1 0
U	1 0 1 0 1

i-tes Bit gibt an welches Bit auf i-ter Ebene



soll so kompakt wie mögl. sein, unnötige Routerknoten vermeiden

nur 1 Wort, das mit 1 beginnt

maximal m Ebenen (m = Wortlänge)
↳ max. 2^m Datensätze

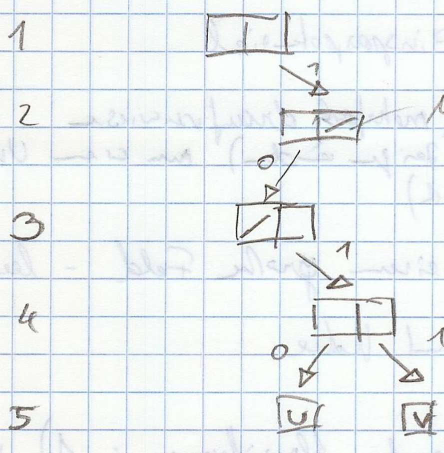
Router: nur 1 Nachfolger → redundant, wenn nicht weiterer Routerknoten
↳ so kompakt wie möglich
Sortiereigenschaft: links nur 0, rechts nur 1

Aufruf: RTS (root, key, i) rekursiv, so lange in die Tiefe, bis kein Router mehr oder Ende: NULL

⊖ WC: $\Theta(m)$ jedes Bit muss angesehen werden, Aufwand nur durch Länge des Strings gegeben (bin. Suchbaum: immer von n abh.)
⊕ nur von Schlüssellänge abhängig
WC hieß: Schlüssel gefunden

BC konstante Zeit, weil schon früher abgebrochen (Wort nicht gefunden)

neuen Datensatz einfügen: zuerst suchen Bsp. U 10101 → müsste rechts stehen



Aufwand: $\Theta(m)$ WC

* gleiches Datum noch mal?

Aufruf: rekursiver Abstieg wie bei Suche: so lange bis Datenknoten mit dem in Konflikt → Knoten sichern, neuen Router einfügen, anderes Feld mit NULL init., gesicherten Knoten wieder einfügen, Prozedur wieder rekursiv aufrufen
Ende für NULL-Pointer wieder rek. Aufruf → Datenknoten einhängen

Entfernen: auch nicht mehr benötigte Routenknoten
zuerst Suche => bis gesuchten Datenknoten, NULL eintragen
eine Ebene drüber – prüfen ob Routenknoten noch nötig (der über gelöschten Null und danach Verweis auf Datenknoten: Verbindung wird neu gesetzt nach oben – Route nicht mehr nötig

Laufzeit: $\Theta(m)$ (Aufräumarbeit: nur konstanter Aufwand, nicht relevant!)

längere Worte => Datenstruktur wird relativ hoch

Indexed Trie: Knoten kann > 2 Nachfolger haben (max. Anz. Zeichen im Alphabet)
nicht mehr zw. Daten- u. Routenknoten unterscheiden
End-Flag: zeigt, dass Wort zu Ende sein kann
jedes True markiert ein Wort

Keine Beschränkung mehr auf Wortlänge
eig. Wortlänge m: dann $|A|^m$ Wörter, aber in der Praxis Wortlänge nicht beschränkt

Vorteil: überall einheitl. Bits, keine weiteren Datenknoten

Suche $\Theta(m)$ WC , $O(m)$ wenn schon vorher abbricht

Einfügen: neuen leeren Knoten einfügen, wenn letzter Buchstabe => True,
sonst rekursiv absteigen, bis letzter Buchstabe

$\Theta(m)$, $O(m \cdot |A|)$ Größe des Alphabets berücksichtigt, zwar Konstant, kann bei großem Alphabet relevant sein

Entfernen: auch hier unnötige Knoten entfernen: alle End-Flags False, keine Pointer
rekursiv: in jedem Knoten darüber prüfen, wenn unnötig, dann in übergeordn. Pointer = NULL

$\Theta(m \cdot |A|)$ prüfen, ob Knoten unnötig

aber: die meisten Knoten nur wenige Nachfolger, vor allem weiter unten
=> nicht mehr als Array, sondern nur mehr die relevanten: statt Array lin. Liste,
Index mit speichern: Buchstaben auch, zus. Laufzeit: durch lin. Liste sequentiell durch. bei großen Alphabeten: trotzdem effizienter
auch Variante: die ersten k Ebenen als Array, dann lin. Liste
"Linked Trie"

Suffix Compression: noch viel Einsparpotential

gleiche Endung: ein Mal gespeichert, mehrfach draufverweisen (Bsp. S 76: ... da => 2 gleiche Teilbäume – nur 1 speichern u. zeigen drauf) nur wenn Unterbäume ident (Bsp. S 76: oder zusätzl. daran: nicht möglich)

Packed Trie: alle Knoten in einem großen Feld – leere Knoten durch andere ausnützen

$k = k + ord(\omega_i)$: berechnet Index

(Komplexere schwierige Algorithmen)

Kompaktierung: einfacher Greedy-Algorithmus: 1) sort nach Anz. nichtleerer Knoten müssen unterschiedliche Startindizes haben (Definitionssache: auch negativer Startindex möglich) fallend

Anwendung: Siehe Homepage