AVL Bäume

- AVL-Bäume sind *höhenbalancierte* Bäume

- Sei $bal(v) = h(v.left) - h(v.right)$ die Balance von v
- Ein AVL-Baum ist ein binärer Baum falls für alle Knoten v $bal(v) \in \{-1, 0, 1\}$
- Die Höhe eines AVL-Baums is begrenzt mit $h(T) < 1.44 \log_2(n + 2)$

Rebalancierung von AVL Bäume

- Einfügen und Insertieren führt zu unbalancierten Bäumen
- Nach jeder Insertion oder Deletion wird der Baum re-balanciert.
- Insertion und Deletion führt schlimmstenfalls zu eine Node mit $bal(v) = \pm 2$
- Es muss also nur eine kleine lokale Dis-balance ausgeglichen werden
- Dies erfolgt durch “Rotations” Operationen

$\rightarrow$ Tafel

Summary AVL Bäume

- Rotationsoperationen kosten $\mathcal{O}(1)$ Zeit
  meist ist nur eine, höchstens aber $\log n$ Rotationen notwendig
- Insertionen und Deletionen im AVL Baum kosten $\mathcal{O}(\log n)$ Zeit
- Auch Suchen, Minimum, Maximum, Successor, Predecessor brauchen je $\mathcal{O}(\log n)$ Zeit

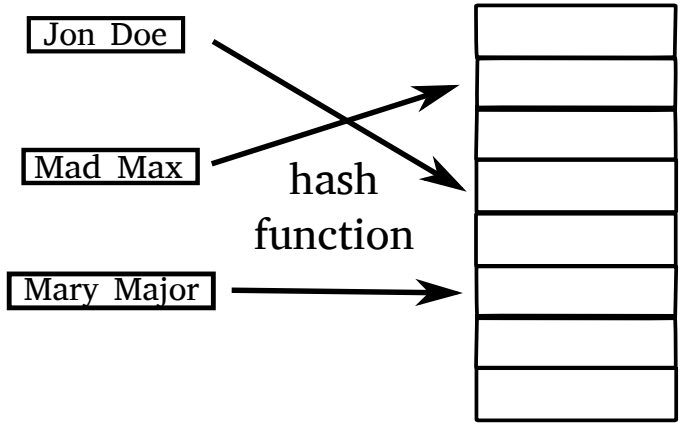
B-trees and B*-trees

B-Bäume sind effizienter im Speicherzugriff, v.a. wenn die Daten nicht mehr im Hauptspeicher (RAM) gehalten werden können.

- Alle Blätter auf gleicher Tiefe
- Höchstens m Kinder pro Knoten
- mindestens $\lceil \frac{m}{2} \rceil$ Kinder
- Knoten mit $l + 1$ Kindern hat l Schlüssel

- B* Baum: Daten in Blättern (auf Festplatte), nur Schlüssel in internen Knoten

Hashing



- Ein Hash speichert Daten in einer Tabelle der Grösse N
- N sollte grösser sein als die Anzahl der Daten $N > n$
- Hash function h bildet Schlüssel auf Positionen in der Tabelle ab
- Hash collisions treten auf wenn $h(k_1) = h(k_2)$

Ohne Kollisionen kann in $\mathcal{O}(1)$ gesucht werden!

Die Hash Funktion

- Die Hash Funktion soll Schlüssel quasi-zufällig auf das Intervall $[0 \ldots N-1]$ abbilden
- ähnliche Schlüssel sollen *nicht* ähnliche hashes ergeben
- Hash Funktion sollte schnell zu berechnen sein

- Beispiel Divisions-Rest-Methode: $h(k) = k \mod N$
  Funktioniert gut mit N Primzahl, schlecht falls $N = 2^k$
- Multiplikationsmethode:

$$h(k) = \lfloor N(k \cdot A - \lfloor k \cdot A \rfloor) \rfloor$$

  Gut mit A irrationale Zahl, z.b. goldene Schnitt $A = \frac{\sqrt{5}-1}{2}$

Ohne Kollisionen kann in $\mathcal{O}(1)$ gesucht werden!

Hash Collisions

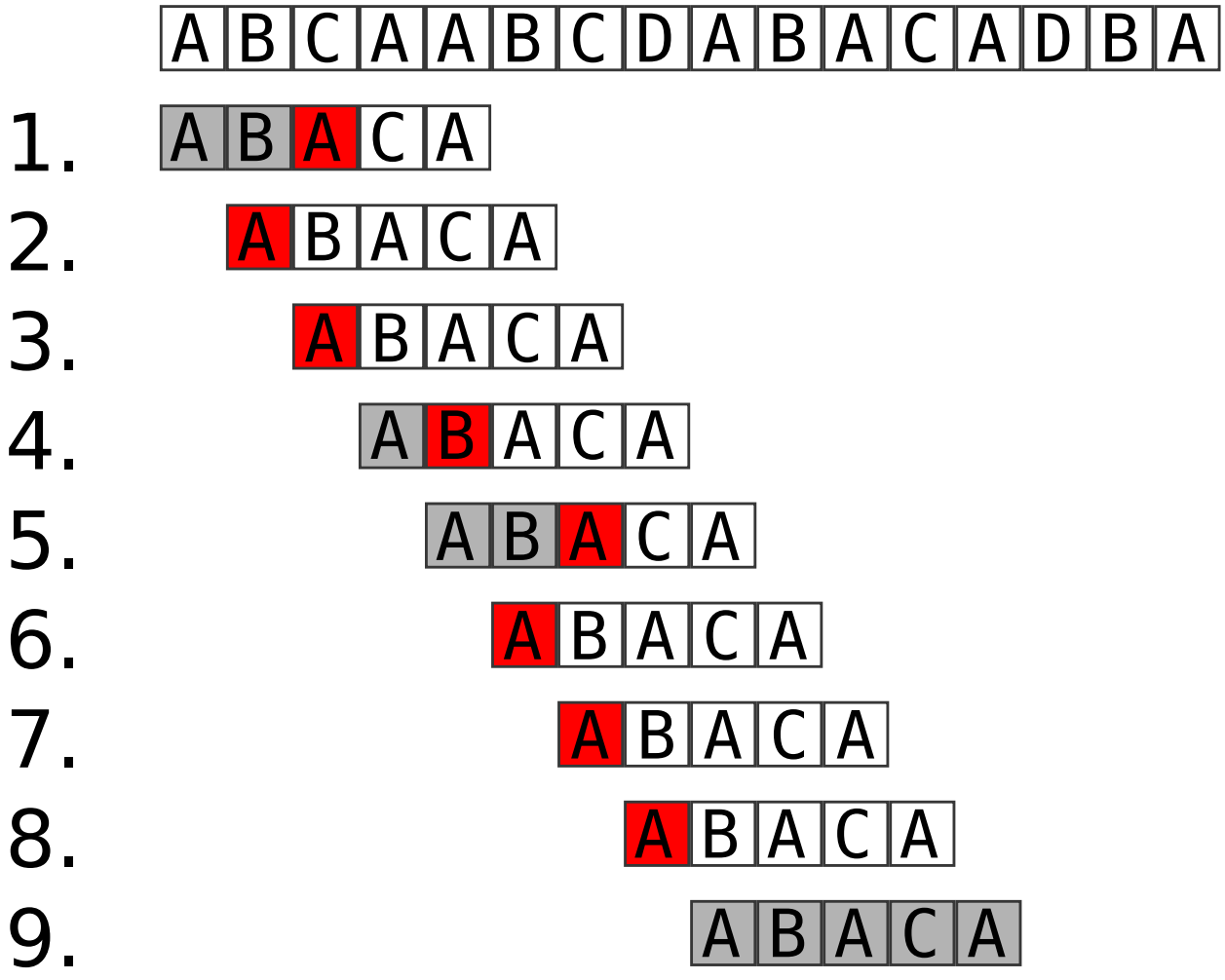
- Unterschiedliche Schlüssel können auf diegleiche Addresse abgebildet werden, i.e. $h(k_1) = h(k_2)$
- Einfachste Lösung: Jede Zelle der Hashtabelle ist der Anfang eine linked list ($\rightarrow$ Tafel)
- Solange $N > n$ sind diese linked lists kurz und kein Problem für die Performance
- Wird $n \gg N$ sind die Listen lang wächst der Suchaufwand von $\mathcal{O}(1)$ auf $\mathcal{O}(n)$
- Ein zu voller Hash kann nicht einfach vergössert werden. Daten müssen in einen neuen grösseren Hash übertragen werden

Facit: Hash Tabellen sind schneller als Suchbäume, wenn die Grösse von vorneherein bekannt ist oder kaum schwankt.

Suchen in Zeichenketten

Brute Force - Algorithmus

Naïver Ansatz:



Brute Force - Algorithmus

Algorithm 7 Finde ein gesuchtes Muster *pat* der Länge m in einem Text *text* der Länge n (BruteForce)

```
for i = 0 → n − m do
    j ← 0
    while j < m ∧ pat[j] = text[i + j] do
        j ← j + 1
    end while
    if j ≥ m then
        return i
    end if
end for
return −1
```

Zeitkomplexität: $\mathcal{O}(n \cdot m)$
Beispiel: 10GB Text, 1MB Pattern, 4GHz Rechner, 1 Vergleich pro Takt $\rightarrow 2500000s = 694.4h \approx 30d$

KMP - Algorithmus[1]

- 1977 von Donald Ervin Knuth und Vaughan Ronald Pratt + unabhängig von James Hiram Morris entwickelt
- Ausnutzen von vorangegangenen Vergleichen bei Nichtübereinstimmung
- Verschieben des Suchmusters *pat* um mehr als eine Position, da $pat[i \ldots j-1] = text[i \ldots i+j-1]$
- Verschieben des Textzeigers *i* nicht notwendig
- Anzahl an Positionen die das Suchmuster verschoben wird ist nur vom Suchmuster abhängig
- 2-Schritt Verfahren:
  1. Suchmusteranalyse:
     Bestimmung wie viele Positionen zurückgegangen werden muss, wenn an Position *j* im Muster ein Vergleich fehlschlägt ($N[j]$)
  2. Suche des Musters im eigentlichen Text

[1] siehe z.B.
`http://www.iti.fh-flensburg.de/lang/algorithmen/pattern/kmp.htm`

Idee des KMP Algorithmus

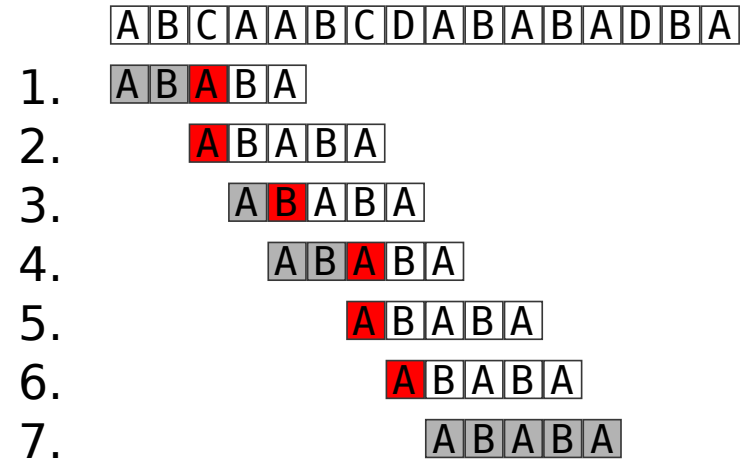
Beispiel:

		i			*i*
Text:	...	BBABB???	Text:	...	BBABB???
Query:		BBABA	Query:		...BBABA
		j			*j*

- Kann es einen match geben, wenn wir die query um eins nach rechts schieben?
- Nein. Nächster möglicher match verschiebt query um 3!
- Warum? Letzter match endet auf BB, query beginnt mit BB

KMP - Algorithmus

j		N[j]
0		- 1
1	A B A B A A B A B A	0
2	A B A B A A B A B A	0
3	A B A B A A B A B A	1
4	A B A B A A B A B A	2



KMP - Algorithmus

Algorithm 8 Finde ein gesuchtes Muster *pat* der Länge m in einem Text *text* der Länge n (KMP)

```
j ← 0
for i = 0 → n − 1 do
    while j ≥ 0 ∧ pat[j] ≠ text[i] do
        j ← N[j]
    end while
    j ← j + 1
    if j = m then
        return i − m
    end if
end for
return −1
```

Zeitkomplexität: $\mathcal{O}(n)$
Es fehlt: Berechnung von $N[0 \ldots m]$

KMP - Algorithmus

Die Suchmusteranalyse:

Algorithm 9 Erstelle Tabelle N (border/failure function) die für jede Position j die Verschiebedistanz bei einem Fehlschlag enthält

---

$i \leftarrow 0$
$j \leftarrow -1$
$N[i] \leftarrow j$
while $i < m$ **do**
  while $j \geq 0 \wedge pat[j] \neq pat[i]$ **do**
    $j \leftarrow N[j]$
  end while
  $i \leftarrow i + 1$
  $j \leftarrow j + 1$
  $N[i] \leftarrow j$
end while

Zeitkomplexität: $\mathcal{O}(m)$
Speicherkomplexität: $\mathcal{O}(m)$

KMP - Algorithmus

Gesamte Zeitkomplexität: $\mathcal{O}(n+m)$
Gesamte Speicherkomplexität: $\mathcal{O}(m)$

wieder Beispiel:
10GB Text, 1MB Pattern, 4GHz Rechner, 1 Vergleich pro Takt
$\rightarrow \approx 2.5s$

Tries (Präfixbäume)

- ’trie’ stammt aus dem Begriff ’Information Retrieval’
- Spezieller Suchbaum für Strings
- Alle Nachkommen eines Knoten haben das gleiche Prefix
- Blätter stehen für Wortende und können zusätzliche Daten speichern
- Einfügen/Löschen/Suchen kostet (fast) gleichviel Zeit $\mathcal{O}(m)$
- Sehr Speichereffizient
- Kein Balancieren des Baums notwendig nach Einfügeoperationen

Anwendungen:

- Predictive Text (Autocomplete Dictionary)
- Rechtschreibüberprüfung
- Datenstruktur für verschiedene (ungenaue) Suchalgorithmen, z.B. Radix sort, Burst sort, etc.
- Ersetzung anderer Datenstrukturen, z.B. Hashes, Binäre Bäume

Tries (Präfixbäume)

Beispielbild: Tafel

Tries (Präfixbäume)

Trie als Ersatz für Hash

- Geordnete Iteration (z.B. alphabetisch durch Pre-order traversal) über seine Elemente
- Suche von ähnlichen Suchmustern
- Schnelles Einfügen (keine hash tables)
- Schneller Zugriff bei kleinen Schlüsseln (keine hash function)
- Keine Kollisionen

Nachteile:

- Langsamer Zugriff wenn trie nicht in den Hauptspeicher passt
- Lange (unnütze) Pfade bei Verwendung von Schlüsseln die nicht natürlichsprachig sind (z.B. Zahlen, bits o. ä.)

Tries (Präfixbäume)

Eigenschaften:

- Höhe des trie durch längsten Schlüssel bestimmt
- Keine leeren Knoten
- Gestalt des tries nur abhängig von Schlüsselmenge
- Viele Einweg-Verzweigungen nahe der Blätter
- Dünne Besetung bei grossen Alphabeten

Komprimierung:

- Möglich bei statischen tries (nur Suche erlaubt)
- Knoten von denen nur eine Kante ausgeht werden speziell (im Vaterknoten) gespeichert (Kollabieren von Kanten)
- Kindknoten Vektor kann auf tatsächlich existente Verzweigungen beschränkt werden

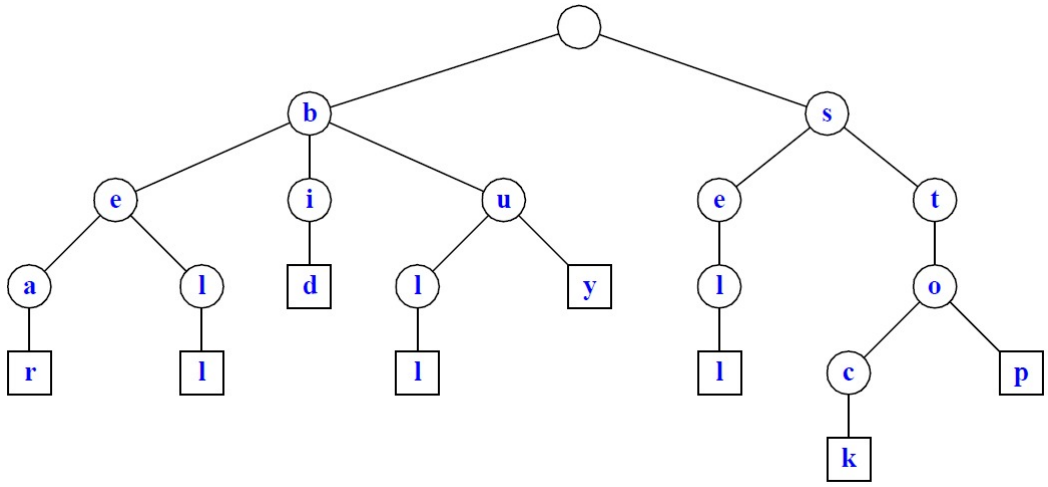
PATRICIA Tree (Prefix-/ Radix-Baum)

- **P**ractical **A**lgorithm **t**o **R**etrieve **I**nformation **C**oded **i**n **A**lphanumeric
- Jeder Knoten mit nur einem Kindknoten wird mit dem Kindknoten verschmolzen
- Kanten können mehr als ein Zeichen darstellen
- Knoten speichert das gemeinsame Präfix seiner Kindknoten
- Nichtexistente Ausgangskanten benötigen keinen extra Speicher (werden übersprungen)
- Binärer Baum aber auch alphanumerisch möglich
- Sehr Speichereffizient
- Aufwendige Einfüge-/Lösch-/Such-Operationen
- Schneller Suchabbruch bei nichtexistentem Schlüssel

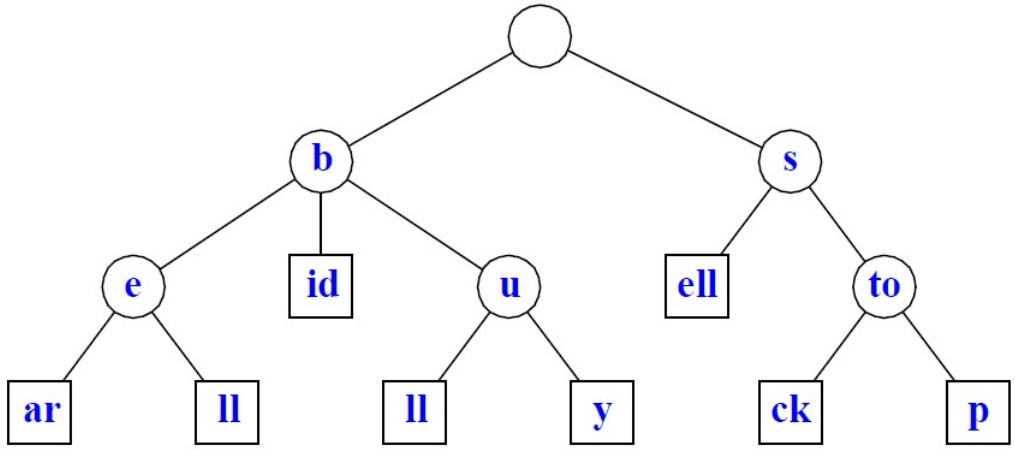
PATRICIA Tree (Prefix-/ Radix-Baum)

Beispiel: {bear, bell, bid, bull, buy, sell, stock, stop}

Standard Trie:



Compressed Trie:



Suffix Trees

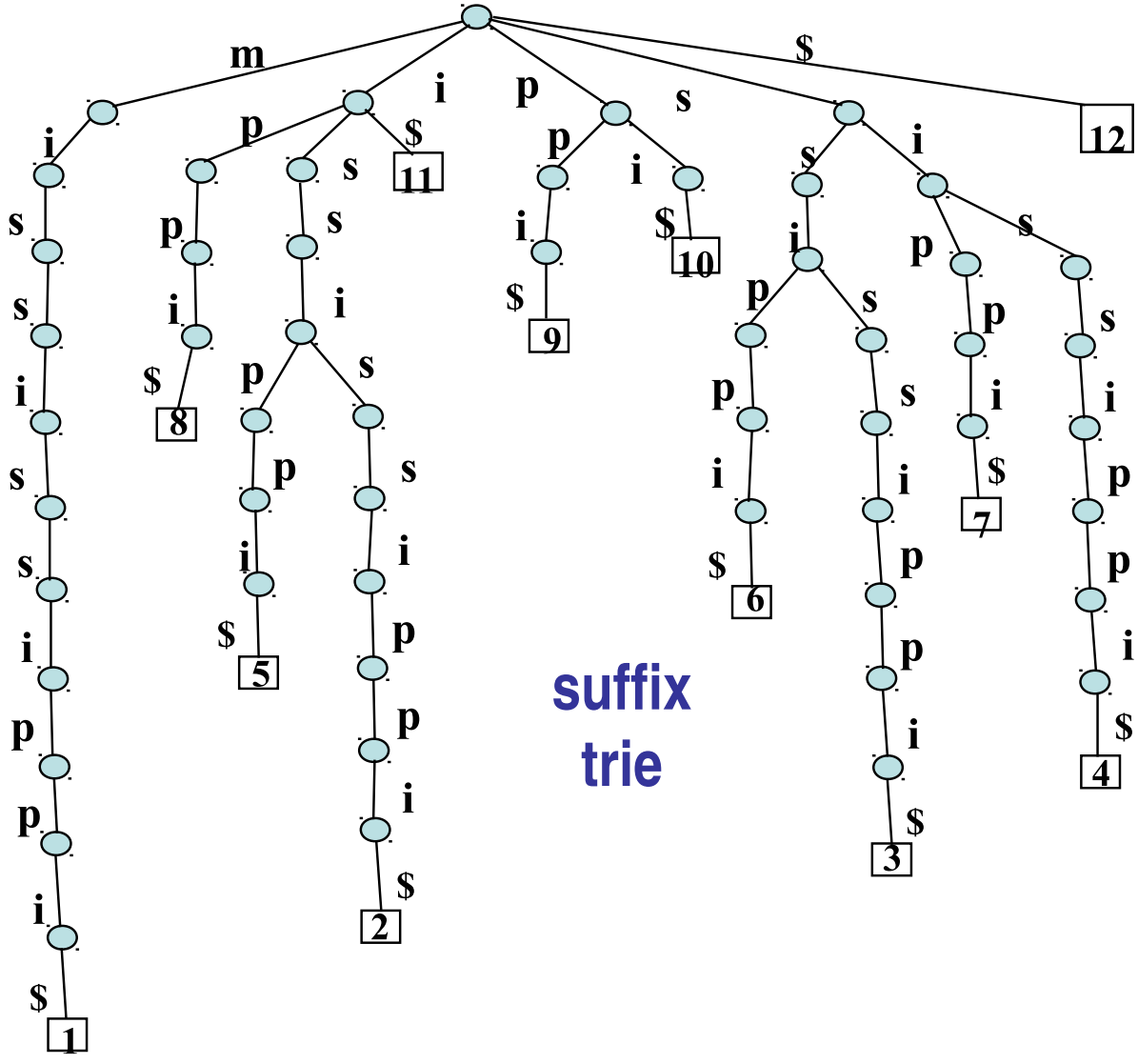
- Spezialfall eines Trie (PATRICIA Tree)
- Speichert alle Suffixes einer Zeichenkette
- Jeder Pfad von der Wurzel zu einem Blatt ist ein Suffix
- Kann in linearer Zeit mit linearem Speicher aufgebaut werden
- Naïver Ansatz: $\mathcal{O}(n^2)$
- Schnelles Auffinden von Teilzeichenketten (genaue, ungenaue Suche und reguläre Ausdrücke)
- Grosser Speicherplatzbedarf

Geschichte

- Morrison (1968): PATRICIA Tree
- Weiner (1973): Konstruktion mit linearer Laufzeit
- McCreight (1976): Speichereffizientere Algorithmus
- Ukkonen (1995): Einfacherer Algorithmus mit $\mathcal{O}(n)$ Laufzeit und Speicher, ermöglicht on-line Konstruktion
- Farach (1997): Optimal für alle Alphabete

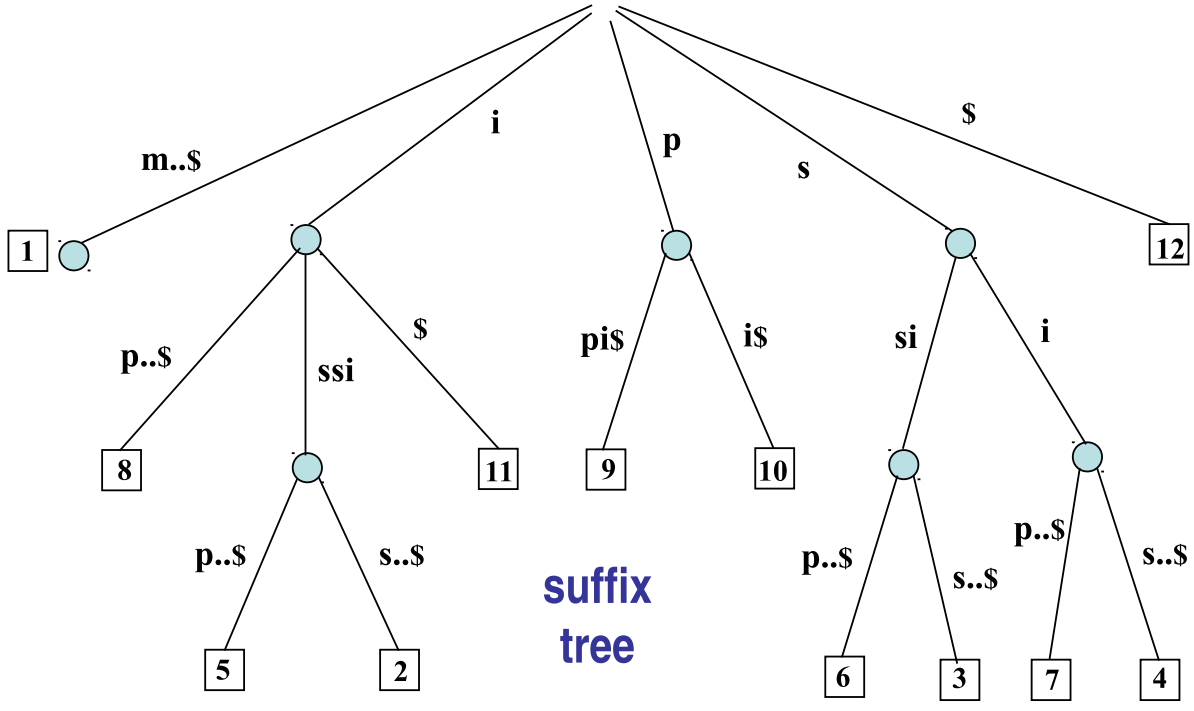
Suffix Trees

Suffix trie for “mississippi”

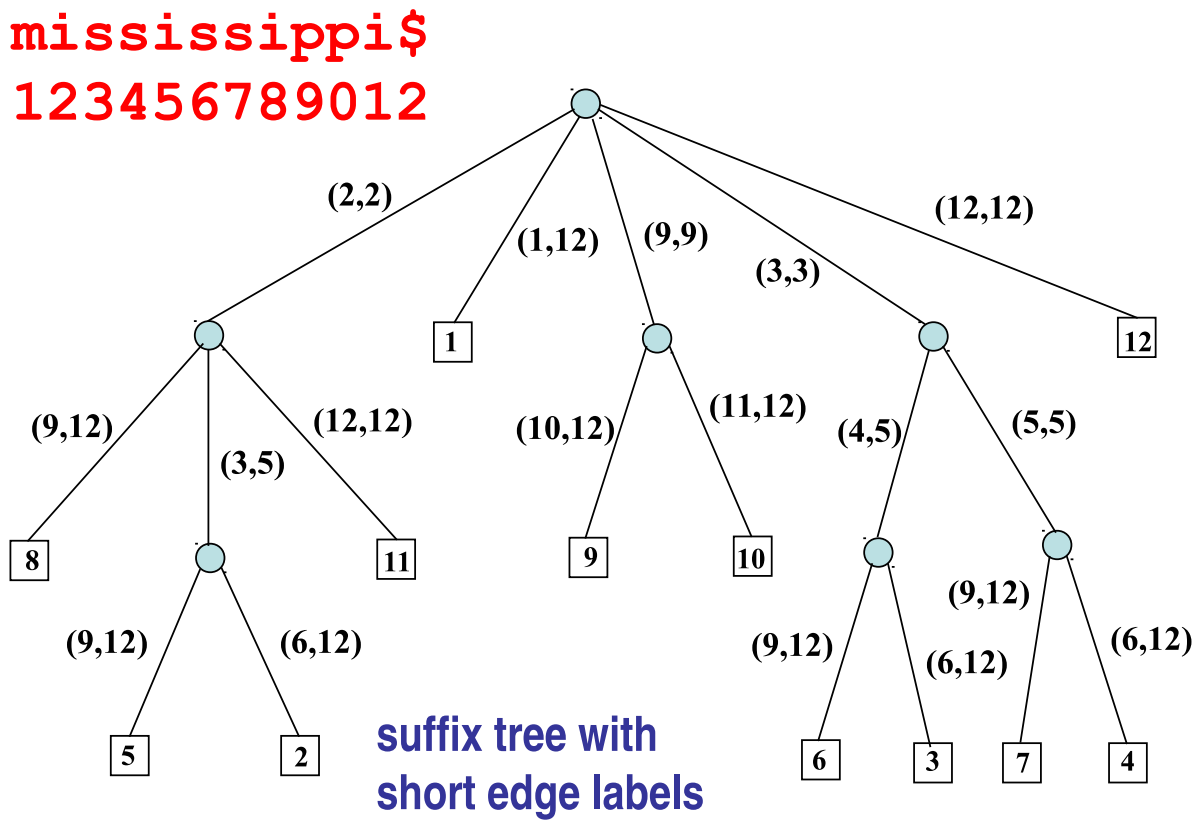


Suffix Trees

Suffix tree (compressed)



Suffix Trees



Suffix Tree Properties

A suffix tree can be stored in $\mathcal{O}(n)$ space

- There are exactly n leafs
- Each interior node has ≥ 2 children
- $\implies \mathcal{O}(n)$ nodes
- With short edge labels only 2 numbers per edge
- In practice at least 20n bytes

Suffix trees can be constructed in $\mathcal{O}(n)$ time

- e.g. Ukkonen (1995)
- from suffix array (Kärkkäinen & Sanders, 2003)

Searching in suffix tree

Search string of length m in text of length n

- build suffix tree of T in time $\mathcal{O}(n)$
- follow path from root through matching characters in $\mathcal{O}(m)$
- all occurrences are gicen by the leafs below the end point of the search
- suffix tree has to be buit only once
  r searches in $\mathcal{O}(n + r \cdot m)$ time

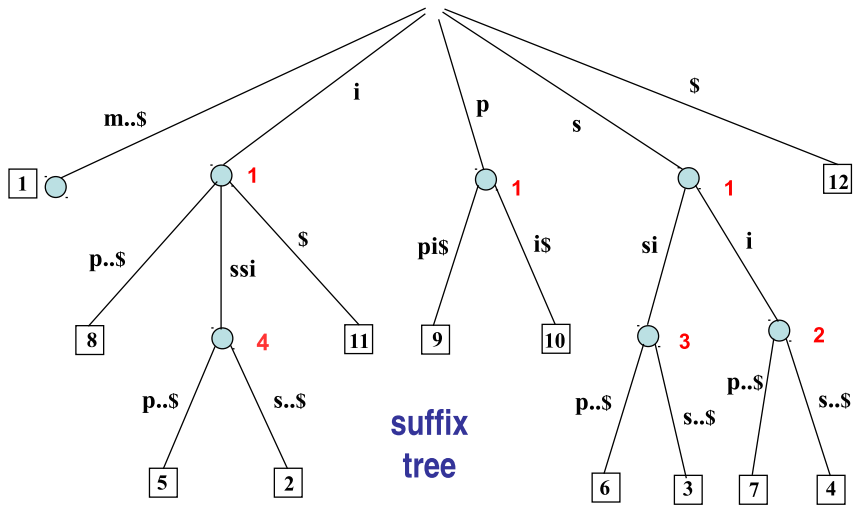
Search Method comparison:

	Preprocess query	Preprocess text	Added Space	Time
Brute force			$\mathcal{O}(1)$	$\mathcal{O}(nm)$
KMP	$\mathcal{O}(m)$		$\mathcal{O}(m)$	$\mathcal{O}(n)$
Suffix trie		$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(m)$

Longest repeated substring

Find the longest string that occurs at least twice in T

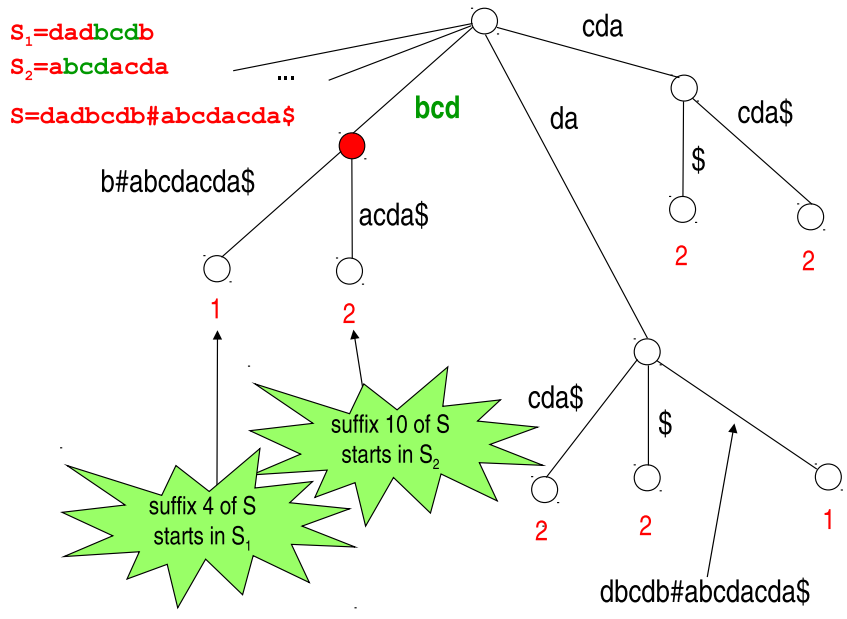
- build suffix tree of T in time $\mathcal{O}(n)$
- traverse tree, find branch node with greatest string depth
- traversal takes $\mathcal{O}(n)$ time



Longest common substring

Find the longest common string LCS that occurs in both S_1 and S_2

- build suffix tree for $S = S_1 \# S_2 \$$
- traverse tree
  find *common* branch node with greatest string depth
- *common* branch node has descendant leafs from S_1 *and* S_2
- takes $\mathcal{O}(|S_1| + |S_2|)$ time



Suffix Arrays

- Space efficient alternative to suffix trees
- Imagine all suffixes of a text sorted alphabetically
- Array S of numbers $1 \ldots n$, such that $S[i] = k$ implies the kth suffix ($x_{k \ldots n}$) appears at position i in the sorted list
- Typically needs $4n$ bytes space

Suffix Array Example

1:	mississippi
2:	ississippi
3:	ssissippi
4:	sissippi
5:	issippi
6:	ssippi
7:	sippi
8:	ippi
9:	ppi
10:	pi
11:	i

11:	i
8:	ippi
5:	issippi
2:	ississppi
1:	mississippi
10:	pi
9:	ppi
7:	sippi
4:	sissippi
6:	ssippi
3:	ssissippi

$$S = [11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3]$$