Πανεπιστήμιο Κρήτης, Τμήμα Επιστήμης Υπολογιστών
Άνοιξη 2006

ΗΥ463 - Συστήματα Ανάκτησης Πληροφοριών
Information Retrieval (IR) Systems

Ευρετηριασμός, Αποθήκευση και Οργάνωση Αρχείων Κειμένων (Indexing, Storage and File Organization)

Γιάννης Τζίτζικας
Διάλεξη : 6
Ημερομηνία : 10-3-2006

Δομές Ευρετηρίου: Διάρθρωση Διάλεξης

Ευρετηριασμός Κειμένου:Εισαγωγή

- Σκοπός
  - **Σχεδιασμός δομών δεδομένων που επιτρέπουν την αποδοτική υλοποίηση της γλώσσας επερώτησης**
- Απλοϊκή προσέγγιση: σειριακή αναζήτηση (online sequential search)
  - Ικανοποιητική μόνο αν η συλλογή των κειμένων είναι **μικρή**
  - Είναι η **μόνη** επιλογή αν η συλλογή κειμένων είναι **ευμετάβλητη**
- Εδώ
  - **σχεδιασμός δομών δεδομένων, που ονομάζονται ευρετήρια (called *indices*), για επιτάχυνση της αναζήτησης**

Ανάγκες Γλωσσών Επερώτησης

- Απλές
  - βρες έγγραφα που **περιέχουν** μια λέξη t
  - βρες **πόσες φορές** εμφανίζεται η λέξη t σε ένα έγγραφο
  - βρες τις **θέσεις** των εμφανίσεων της λέξης t στο έγγραφο
- Πιο σύνθετες
  - Boolean queries
  - phrase/proximity queries
  - pattern matching
  - Regular expressions
  - Structure-based queries
  - ...

Σχεδιάζουμε το ευρετήριο ανάλογα με το μοντέλο ανάκτησης και τη γλώσσα επερώτησης

Γενική (Λογική) μορφή ενός ευρετηρίου

Indexing Items

Documents

	k_1	**k_2**	...	**k_j**	...	**k_t**
d_1	$c_{1,1}$	$c_{2,1}$	...	$c_{i,1}$	...	$c_{t,1}$
d_2	$c_{1,2}$	$c_{2,2}$	...	$c_{i,2}$	...	$c_{t,2}$
...	...	...	...	...	...	...
d_i	$c_{1,j}$	$c_{2,j}$	...	$c_{i,j}$	...	$c_{t,j}$
...	...	...	...	...	...	...
d_N	$c_{1,N}$	$c_{2,N}$	...	$c_{i,N}$	...	$c_{t,N}$

c_{ij}: το κελί που αντιστοιχεί στο έγγραφο di και στον όρο kj, το οποίο μπορεί να περιέχει:
- ένα w_{ij} που να δηλώνει την παρουσία ή απουσία του kj στο di (ή τη σπουδαιότητα του kj στο di)
- τις θέσεις στις οποίες ο όρος kj εμφανίζεται στο di (αν πράγματι εμφανίζεται)

Ερωτήματα:

Τι πρέπει να έχει το κάθε c_{ij}

Πώς να υλοποιήσουμε αυτή τη λογική δομή ώστε να έχουμε απόδοση;

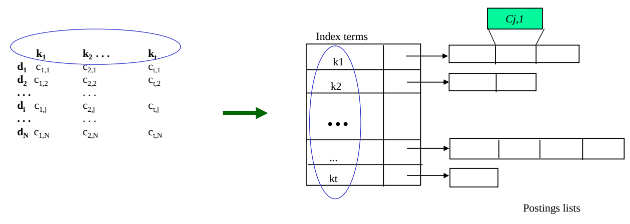
Τεχνικές Ευρετηριασμού (Indexing Techniques)

- Ανεστραμένα Αρχεία (Inverted files)
  - η πιο διαδομένη τεχνική
- Δένδρα και Πίνακες Καταλήξεων (Suffix trees and arrays)
  - γρήγορες για “phrase queries” αλλά η κατασκευή και η συντήρησή τους είναι δυσκολότερη και ακριβότερη
- Αρχεία Υπογραφών (Signature files)
  - Χρησιμοποιήθηκαν πολύ τη δεκαετία του 80. Σπανιότερα σήμερα.

Ανεστραμμένα Αρχεία (Inverted Files)

Ανεστραμμένο Αρχείο

Λογική Μορφή Ευρετηρίου — Μορφή Ανεστραμμένου Ευρετηρίου



Άρα δεν αποθηκεύουμε τα «μηδενικά κελιά»

Inverted Files (Ανεστραμμένα αρχεία)

Inverted file = a word-oriented mechanism for indexing a text collection in order to speed up the searching task.

- An inverted file consists of:
  - Vocabulary: is the set of all distinct words in the text
  - Occurrences: lists containing all information necessary for each word of the vocabulary (documents where the word appears, frequency, text position, etc.)

Ανεστραμμένο αρχείο για ένα μόνο έγγραφο και αποθήκευση θέσεων εμφάνισης κάθε λέξης

Κείμενο

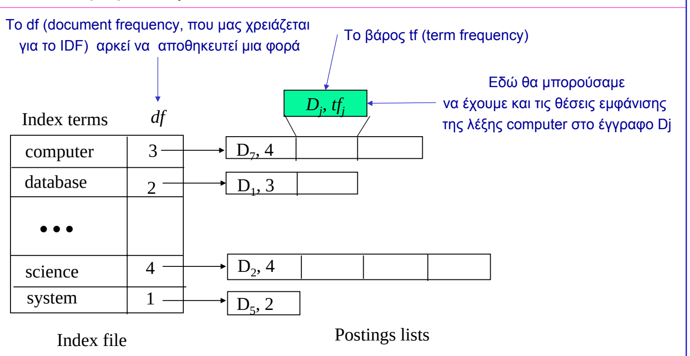
That house has a garden. The garden has many flowers. The flowers are beautiful
1 6 12 16 18 25 29 36 40 45 54 58 66 70

Inverted File:

Vocabulary	Occurrences
beautiful	70
flowers	45, 58
garden	18, 29
house	6

Τι άλλο θα κάνατε αν είχαμε πολλά έγγραφα και θέλαμε να υλοποιήσουμε το Διανυσμ. Μοντέλο;

Ανεστραμμένο αρχείο για πολλά έγγραφα, και βάρυνση tf-idf



Ανεστραμμένο αρχείο και βάρυνση freq

Document Corpus

Doc	Text
1	Pease porridge hot
2	Pease porridge cold
3	Pease porridge in the pot
4	Pease porridge hot, pease porridge not cold
5	Pease porridge cold, pease porridge not hot
6	Pease porridge hot in the pot

Inverted File

Vocabulary	Inverted Lists
cold	<2,1> <4,1> <5,1>
hot	<1,1> <4,1> <5,1> <6,1>
in	<3,1> <6,1>
not	<4,1> <5,1>
pease	<1,1> <2,1> <3,1> <4,2> <5,2> <6,1>
porridge	<1,1> <2,1> <3,1> <4,2> <5,2> <6,1>
pot	<3,1> <6,1>
the	<3,1> <6,1>

Αναστραμμένο Αρχείο: Κατασκευή και Αναζήτηση

Υπόβαθρο/Επανάληψη: **Tries**

Tries
- multiway trees for stroring strings
- able to retrieve any string in time proportional to its length (independent from the number of all stored strings)

Description
- – every edge is labeled with a letter
- – searching a string s
  - start from root and for each character of *s* follow the edge that is labeled with the same letter.
  - continue, until a leaf is found (which means that s is found)

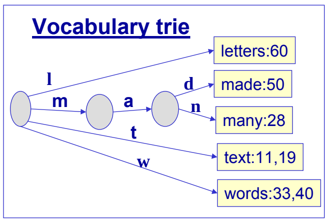
Tries: Παράδειγμα

1 6 9 11 17 19 24 28 33 40 46 50 55 60

This is a **text**. A **text** has **many words**. **Words** are **made** from **letters.**

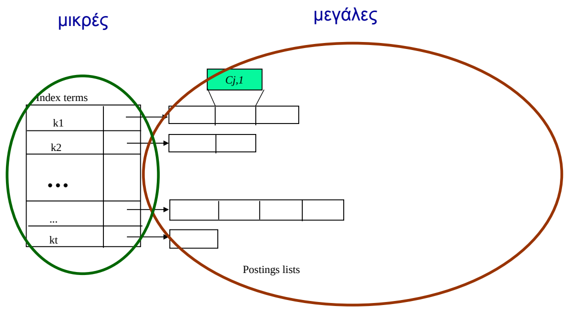
Vocabulary
- text (11)
- text (19)
- many (28)
- words (33)
- words (40)
- made (50)
- letters (60)

Vocabulary (ordered)
- letters (60)
- made (50)
- many (28)
- text (11,19)
- words (33,40)



Note: Lecture 5: Successor variety ?

Ανεστραμμένα Αρχεία: Απαιτήσεις Χώρου



Ανεστραμμένα Αρχεία: Απαιτήσεις Χώρου

Notations
- *n*: the size of the text
- V: the size of the vocabulary

For the Vocabulary:
- Rather **small**.
- According to *Heaps'* law (*to be described in a subsequent lecture*) the vocabulary grows as $O(n^{\beta})$, where β is a constant between 0.4 and 0.6 in practice. So V ~ sqrt(n)

For Occurrences:
- **Much more** space.
- Since each word appearing in the text is referenced once in that structure, the extra space is *O(n)*
- To reduce space requirements, a technique called *block addressing* is used

how?

Block Addressing

- The text is divided in blocks
- The occurrences point to the blocks where the word appears
- Advantages:
  - – the number of pointers is smaller than positions
  - – all the occurrences of a word inside a single block are collapsed to one reference
  - – (**indices of only 5% overhead over the text size can be obtained with this technique. Of course this depends on the block size**)
- Disadvantages:
  - – online sequential search over the qualifying blocks if exact positions are required
    - e.g. for finding the sentence where the word occurs
    - e.g. for evaluating a context (phrasal or proximity) query

Block Addressing: Example

That house has a garden. The garden has many flowers. The flowers are beautiful

1 6 12 16 18 25 29 36 40 45 54 58 66 70

Vocabulary	Occurrences
beautiful	70
flowers	45, 58
garden	18, 29
house	6

Block 1	Block 2	Block 3	Block 4
That house has a	garden. The garden has	many flowers. The flowers	are beautiful

Vocabulary	Occurrences
beautiful	4
flowers	3
garden	2
house	1

Size of Inverted Files as percentage of the size of the whole collection

45% of all words are stopwods

Index	Small collection (1Mb)		Medium collection (200Mb)		Large collection (2Gb)	
	Without stopwords	All words	Without stopwords	All words	Without stopwords	All words
Addressing words	45%	73%	36%	64%	35%	63%
Addressing 64K blocks	27%	41%	18%	32%	5%	9%
Addressing 256 blocks	18%	25%	1.7%	2.4%	0.5%	0.7%

Addressing words: 4 bytes per pointer (2^32 ~ giga)
Addressing 64K blocks: 2 bytes per pointer
Addressing 256 blocks: 1 byte per pointer

Searching an inverted index

Searching an inverted index

General Steps:

1/ Vocabulary search:
- the words present in the query are searched in the vocabulary

2/Retrieval occurrences:
- the lists of the occurrences of all words found are retrieved

3/Manipulation of occurrences:
- The occurrences are processed to solve the query
- If block addressing is used we have to search the text of the blocks in order to get the exact positions and number of occurences

1/ Vocabulary search

As Searching task on an inverted file always starts in the vocabulary, it is better to **store the vocabulary in a separate file**
- this file is not so big so it is possible keep it at main memory at search time

Suppose we want to search for a word of length *m.*

Options:
- *Cost of searching a sequential file: O(V)*
- *Cost of searching assuming hashing: O(m)*
- *Cost of searching assuming tries: O(m)*
- *Cost of searching assuming the file is ordered (lexicographically): O(log V)*
  - this option is cheaper in space and very competitive

The structures most used to store the vocabulary are ***hashing, tries*** or ***B-trees.***

1/ Vocabulary Search (II)

Remarks
- prefix and range queries
  - can also be solved with binary search, tries or B-trees buts not with hashing
- context queries
  - are more difficult to solve with inverted indices
    - 1. each element must be searched separately and
    - 2. a list (in increasing positional order) is generated for each one
    - 3. The lists of all elements are traversed in synchronization to find places where all the words appear in sequence (for a phrase) or appear close enough (for proximity).

Inverted Index: A general remark

Experiments show that both the space requirements and the amount of text traversed can be close to O(n^0.85). Hence, inverted indices allow us to have sublinear search time and sublinear space requirements. This is not possible on other indices.

Constructing an Inverted File

Constructing an Inverted File

- All the vocabulary is kept in a suitable data structure storing for each word a list of its occurrences
  - e.g. in a trie data structure
- Each word of the text is read and searched in the vocabulary
  - if a trie data structure is used the this search costs O(m) where m the size of the word
- If it is not found, it is added to the vocabulary with a empty list of occurrences and the new position is added to the end of its list of occurrences

Constructing an Inverted File (II)

- Once the text is exhausted the vocabulary is written to disk with the list of occurrences. Two files are created:
  - in the first file, the list of occurrences are stored contiguously
  - in the second file, the vocabulary is stored in lexicographical order and, for each word, a pointer to its list in the first file is also included.
- The overall process is *O(n)* time

Trie: O(1) per text character
Since positions are appended (in the postings file) O(1) time
It follows that the overall process is O(n)

What if the Inverted Index does not fit in main memory ?

A technique based on **partial Indexes:**

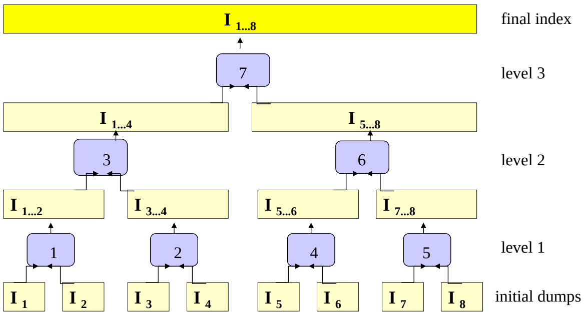
- Use the previous algorithm until the main memory is exhausted.
- When no more memory is available, **write to disk** the **partial index I_i** obtained up to now, and **erase it from main memory**
- Continue with the rest of the text

- Once the text is exhausted, a number of partial indices I_i exist on disk
- The partial indices are **merged** to obtain the final index

Merging two partial indices I1 and I2

- Merge the sorted vocabularies and whenever the same word appears in both indices, merge both list of occurences
- By construction, the occurences of the smaller-numbered index are before those ot the larger-numbered index, the therefore the lists are just **concatenated**
- Complexity: O(n1+n2) where n1 and n2 the sizes of the indices

Merging partial indices to obtain the final



Merging all partial indices: Time Complexity

Notations

- *n*: the size of the text
- V: the size of the vocabulary
- *M*: the amount of main memory available

- The total time to generate partial indices is *O(n)*
- The number of partial indices is *O(n/M)*
- To merge the *O(n/M)* partial indices are necessary $log_2(n/M)$ merging levels
- The total cost of this algorithm is *O(n log(n/M))*

Maintaining the Inverted File

- Addition of a new doc
  - build its index and merge it with the final index (as done with partial indexes)
- Delete a doc of the collection
  - scan index and delete those occurrences that point into the deleted file (complexity: **O(n)**)

Evaluating Phrasal and Proximity Queries with Inverted Indices

- Phrasal Queries
  - Must have an inverted index that also stores *positions* of each keyword in a document.
  - Retrieve documents and positions for each individual word, intersect documents, and then finally check for ordered contiguity of keyword positions.
  - Best to start contiguity check with the least common word in the phrase.
- Proximity Queries
  - Use approach similar to phrasal search to find documents in which all keywords are found in a context that satisfies the proximity constraints.
  - During binary search for positions of remaining keywords, find closest position of k_i to *p* and check that it is within maximum allowed distance.

Αποτίμηση Boolean επερωτήσεων με χρήση ανεστραμμένων αρχείων

Αποτίμηση με χρήση ανεστραμμένων αρχείων

- Primitive keyword: Retrieve containing documents using the inverted index.
- OR: Recursively retrieve e_1 and e_2 and take union of results.
- AND: Recursively retrieve e_1 and e_2 and take intersection of results.
- BUT: Recursively retrieve e_1 and e_2 and take set difference of results.

Inverted Index: Κατακλείδα

- Is probably the most adequate indexing technique
- Appropriate when the text collection is large and semi-static
- If the text collection is volatile online searching is the only option
- Some techniques combine online and indexed searching

Δένδρα και Πίνακες Καταλήξεων (Suffix Trees and Suffix Arrays)

Δένδρα και Πίνακες Καταλήξεων (Suffix Trees and Arrays)

- Κίνητρο
  - Γρήγορη αποτίμηση των phrase queries
  - Η έννοια της **λέξης** (στην οποία βασίζονται τα inverted files) δεν υπάρχει σε άλλες εφαρμογές (π.χ. στις γενετικές βάσεις δεδομένων), άρα υπάρχει ανάγκη για διαφορετικές δομές δεδομένων.

Μια αλυσίδα DNA είναι μια ακολουθία από διατεταγμένα ζευγάρια βάσεων.
Υπάρχουν 4 βάσεις: η αδενίνη (A), η γουανίνη (G), η κυτοσίνη (C) και η θυμίνη (T).
Κάθε ζευγάρι βάσεων του DNA αποτελείται από διαφορετικές βάσεις.
Συγκεκριμένα, η αδενίνη (A) μπορεί να συνδέεται μόνο με τη θυμίνη (T), ενώ η γουανίνη (G) μπορεί να συνδέεται μόνο με την κυτοσίνη (C). Ένα παράδειγμα αποσπάσματος αλυσίδας DNA ακολουθεί:

A G G C T A C C C T T A
T C C G A T G G G A A T

Δένδρα και Πίνακες Καταλήξεων (Suffix Trees and Arrays)

Γενική ιδέα
- Βλέπουμε όλο το κείμενο ως μία μακριά συμβολοσειρά (long string)
- Θεωρούμε κάθε θέση του κειμένου ως **κατάληξη κειμένου (text suffix)**
- Δύο καταλήξεις που εκκινούν από διαφορετικές θέσεις είναι λεξικογραφικά διαφορετικές
  - άρα κάθε κατάληξη προσδιορίζεται μοναδικά από τη θέση της αρχής της
- Επιλογές
  - Ευρετηριάζουμε όλες τις θέσεις του κειμένου
  - Ευρετηριάζουμε κάποιες θέσεις του κειμένου (π.χ. την αρχή των λέξεων)
    - Άρα εδώ έχουμε την έννοια του σημείου ευρετηρίου (index point)
    - Τα σημεία που δεν είναι σημεία ευρετηρίου δεν είναι παραδόσιμα (deliverable)

Παράδειγμα καταλήξεων (θεωρώντας ως σημεία ευρετηρίου τις αρχές των λέξεων)

This is a text. A text has many words. Words are made from letters.

letters.
made from letters.
Words are made from letters.
words. Words are made from letters.
many words. Words are made from letters.
text has many words. Words are made from letters.
text. A text has many words. Words are made from letters.

Δένδρα Καταλήξεων (Suffix Trees)

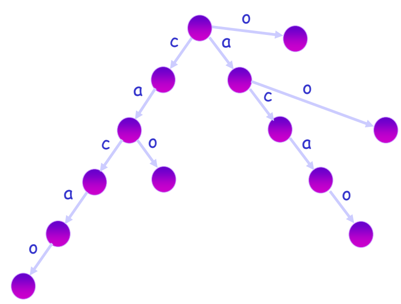
Δένδρο Καταλήξεων:
- Το **δένδρο καταλήξεων** ενός κειμένου είναι ένα **trie** πάνω σε όλες τις καταλήξεις του κειμένου.
  - **Suffix tree** = **trie** built over **all the suffixes** of the text
- Οι δείκτες προς το κείμενο αποθηκεύονται στα φύλλα του δένδρου.

Για μείωση του χώρου, το trie συμπυκνώνεται ως ένα Patricia tree
- Patricia = Practical Algorithm To Retrieve Information Coded in Alphanumerical

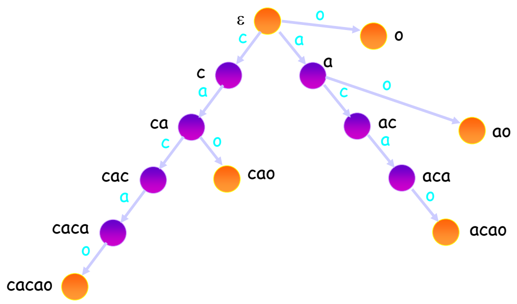
Suffix Trie για τη λέξη "cacao" (θεωρώντας κάθε θέση ως σημείο ευρετηρίου)

Καταλήξεις:
o
ao
cao
acao
cacao

Trie Καταλήξεων



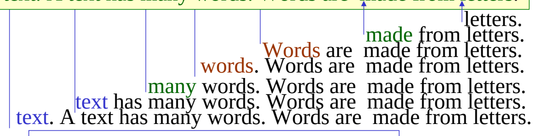
Καταλήξεις:
o
ao
cao
acao
cacao



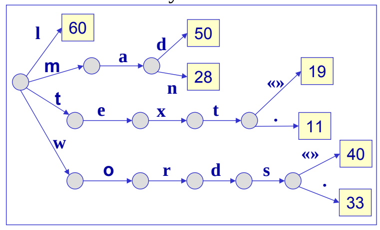
Παράδειγμα καταλήξεων και του αντίστοιχου Suffix Trie

1 6 9 11 17 19 24 28 33 40 46 50 55 60

This is a text. A text has many words. Words are made from letters.



Suffix Trie



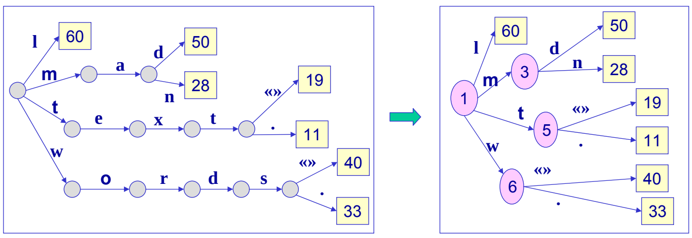
Suffix tree
= Suffix trie compacted into a Patricia tree

This involves compressing unary paths, i.e. paths where each node has just one child.

If unary paths are not present, the tree has O(n) nodes instead of the worst-case O(n^2) of the trie.

Suffix Trie

Suffix Tree



Πίνακες Καταλήξεων (Suffix arrays)

Πίνακες Καταλήξεων (Suffix arrays)
(Space efficient implementation of suffix trees)

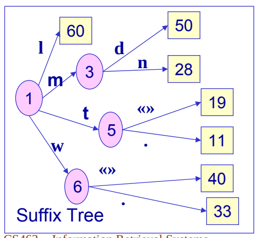
- Suffix trees have a space overhead of 120%-240% over the text size (index points = word beginnings)
- Here we present a data structure with space requirements like those of the vector space model (~40% overhead over the text size)

Πίνακες Καταλήξεων (Suffix arrays)
(Space efficient implementation of suffix trees)

Πίνακας Καταλήξεων:
- Πίνακας με δείκτες προς όλες τις «καταλήξεις» σε λεξικογραφική σειρά
- Για να τον δημιουργήσουμε αρκεί μια depth-fist-search διάσχιση του suffix tree.

1 6 9 11 17 19 24 28 33 40 46 50 55 60

This is a text. A text has many words. Words are made from letters.



Suffix Tree

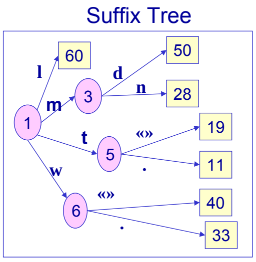
Suffix Array

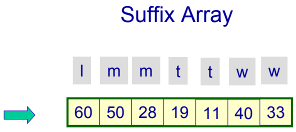
l	m	m	t	t	w	w
60	50	28	19	11	40	33

Πίνακες Καταλήξεων(II)

1 6 9 11 17 19 24 28 33 40 46 50 55 60

This is a text. A text has many words. Words are made from letters.





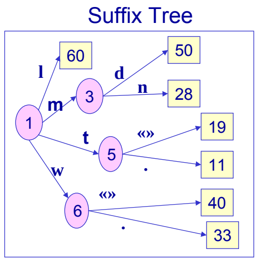
Οφέλη:

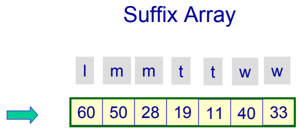
- Μείωση χώρου
  - 1 δείκτη ανά κατάληξη (7 καταλήξεις, πίνακας 7 κελιών)
  - (overhead ~ that of inverted files)
- Δυνατότητα **binary search**

Πίνακες Καταλήξεων(III)

1 6 9 11 17 19 24 28 33 40 46 50 55 60

This is a text. A text has many words. Words are made from letters.

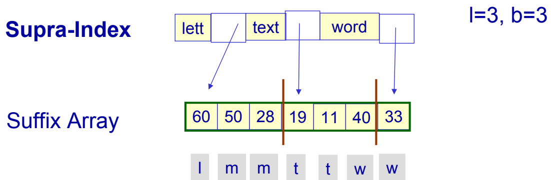




Για να δούμε αν υπάχει μια κατάληξη στο κείμενο κάνουμε διαδική αναζήτηση (binary search) στο περιεχόμενο των δεικτών

Πίνακες Καταλήξεων (IV)

- If vocabulary is big (and the suffix array does not fit in main memory), **supra indices** are employed
  - they store the first l characters for each of every b entries



Δένδρα και Πίνακες Καταλήξεων
Κόστος Αποτίμησης Επερωτήσεων

- Κόστος αναζήτησης μιας συμβολοσειράς μήκους m χαρακτήρων
  - O(m) στην περίπτωση των δένδρων καταλήξεων (suffix tree)
  - O(log n) στην περίπτωση των πινάκων καταλήξεων (suffix array)
- Αποτίμηση phrase queries
  - Η φράση αναζητείται ωσάν να ήταν μια συμβολοσειρά
- Αποτίμηση proximity queries
  - proximity queries have to be resolved element wise