

Chapter 3

Indexing structure

Designing an IR System

Our focus during IR system design:

- In improving **Effectiveness** of the system
 - The concern here is retrieving more relevant documents for users query
 - Effectiveness of the system is measured in terms of precision, recall, ...
 - Main emphasis: Stemming, stop words removal, weighting schemes, matching algorithms
- In improving **Efficiency** of the system
 - The concern here is reducing storage space requirement, enhancing searching time, indexing time, access time...
 - Main emphasis: Compression, indexing structures, space - time tradeoffs

Subsystems of IR system

The two subsystems of an IR system:

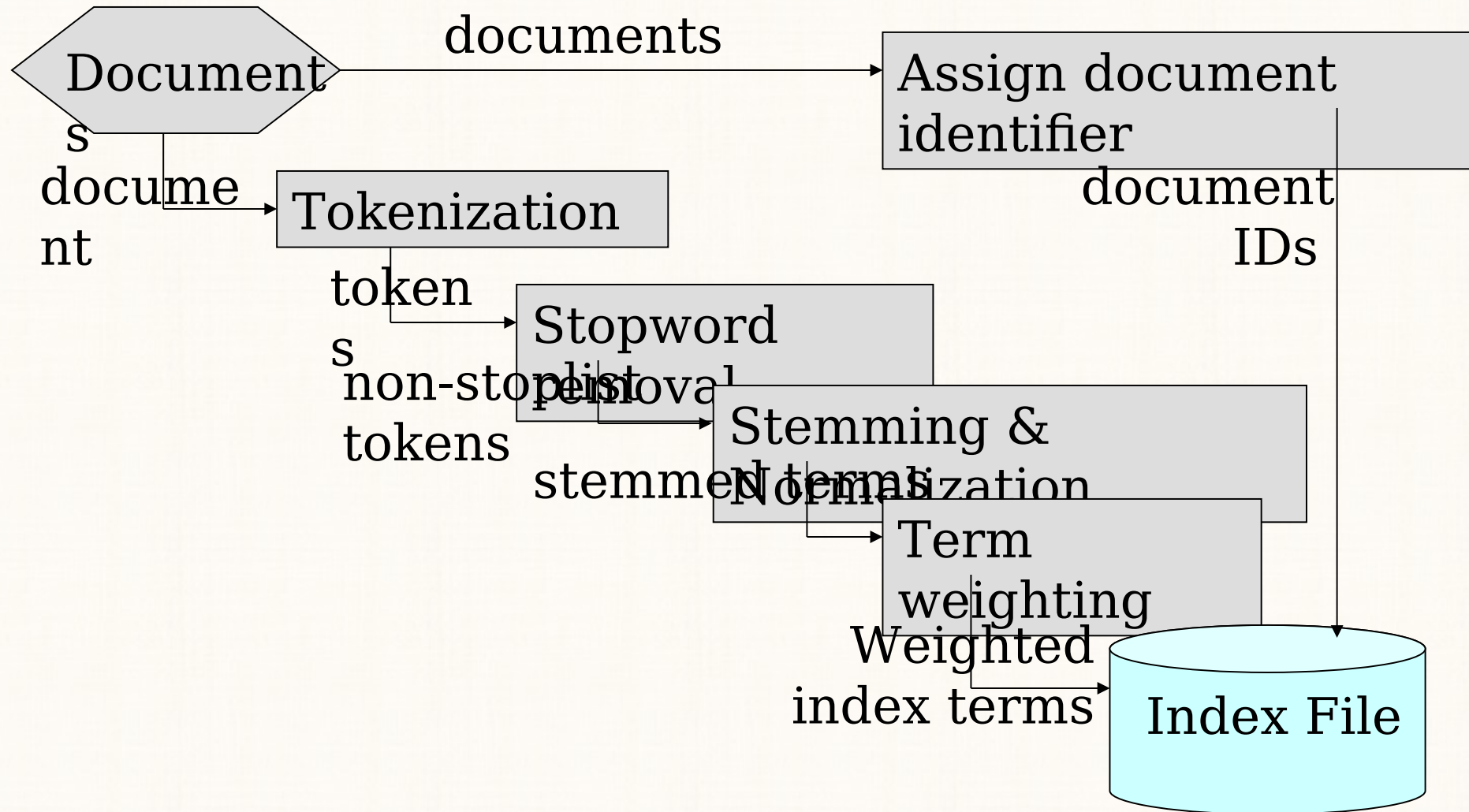
-Indexing:

- is an **offline process** of organizing documents using **keywords extracted** from the collection
- Indexing is used to speed up access to desired information from document collection as per users query

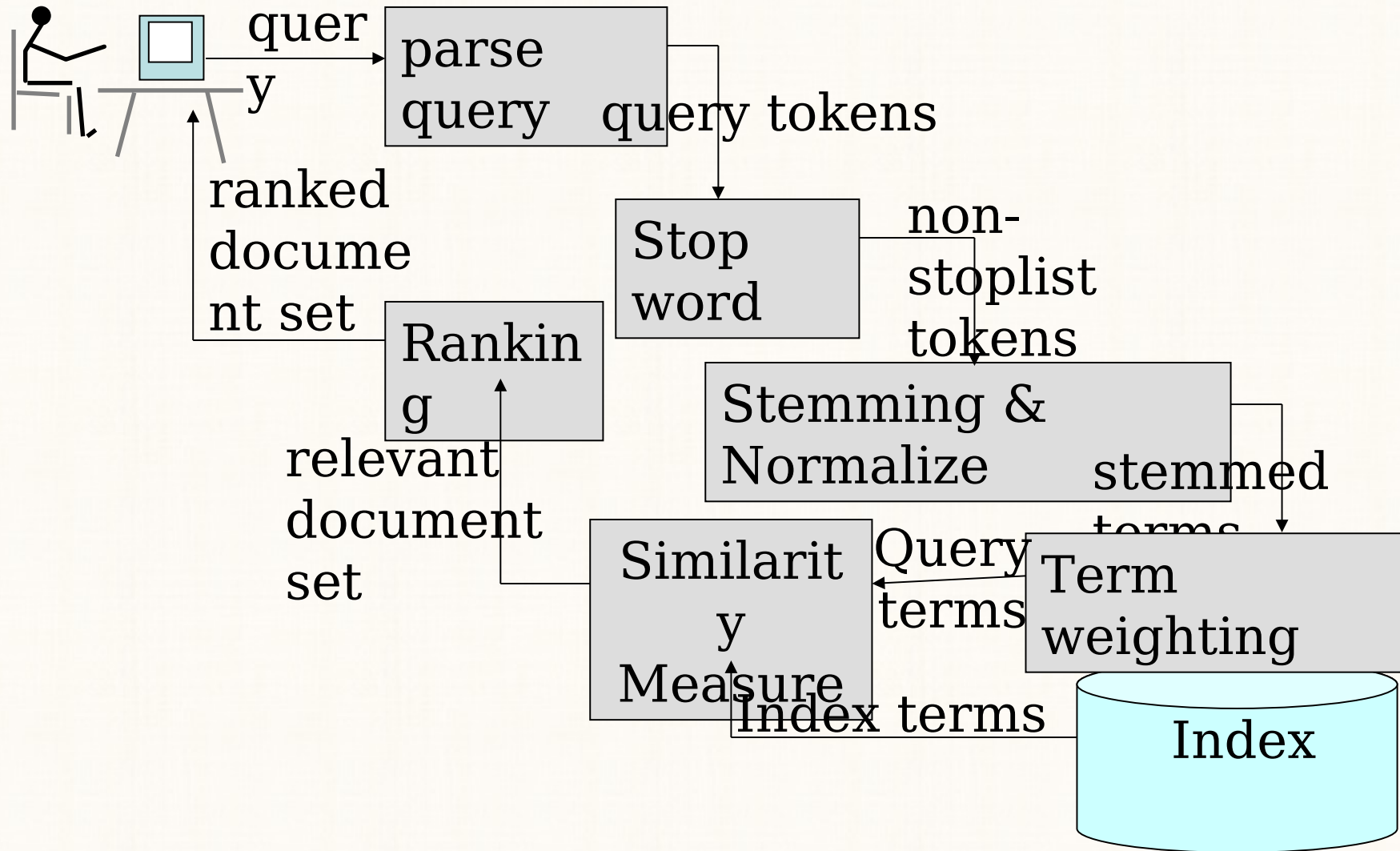
-Searching

- is an **online process** that scans document corpus to find relevant documents that matches users query

Indexing Subsystem



Searching Subsystem

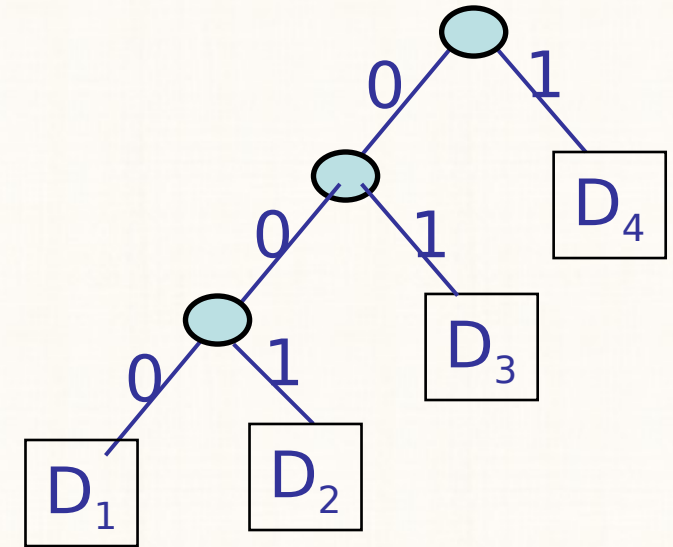


Text Compression

- Text compression is about finding ways to represent the text in **fewer bits** or **bytes**
- **Advantages:**
 - Save storage space requirement.
 - Speed up document transmission time
 - Takes less time to search the compressed text
- **Common compression methods**
 - **Statistical methods:** which requires **statistical information** about **frequency of occurrence** of symbols in the document
 - E.g. Huffman coding
 - Estimate **probabilities of symbols**, code one at a time, shorter codes for high probabilities
 - **Adaptive methods:** which **constructs dictionary** in the processing of compression
 - E.g. Ziv-Lempel compression:
 - Replace words or symbols with a pointer to dictionary entries

Huffman coding

- Developed in 1950s by David Huffman, widely used for **text compression**, **multimedia codec** and message transmission
- **The problem:** Given a set of n symbols and their weights (or frequencies), construct a tree structure (a binary tree for binary code) with the objective of reducing memory space & decoding time per symbol.
- Huffman coding is constructed based on **frequency of occurrence** of letters in text documents



Code of:

$D_1 = 000$

$D_2 = 001$

$D_3 = 01$

$D_4 = 1$

How to construct Huffman coding

Step 1: Create forest of trees for each symbol, $t_1, t_2, \dots, t_n$

Step 2: Sort forest of trees according to falling probabilities of symbol occurrence

Step 3: WHILE more than one tree exist DO

- Merge two trees t_1 and t_2 with least probabilities p_1 and p_2
- Label their root with sum $p_1 + p_2$
- Associate binary code: 1 with the right branch and 0 with the left branch

Step 4: Create a unique codeword for each symbol by traversing the tree from the root to the leaf.

- Concatenate all encountered 0s and 1s together during traversal

•The resulting tree has a prob. of 1 in its root and symbols in its leaf node.

Example

- Consider a 7-symbol alphabet given in the following table to construct the Huffman coding.

Symbol	Probability
a	45
b	13
c	12
d	16
e	9
f	5

- The Huffman encoding algorithm picks each time two symbols (with the smallest frequency) to combine

Exercise

1. Given the following, apply the Huffman algorithm to find an optimal binary code:

Character:	a	b	c	d	e	t
Frequency:	16	5	12	17	10	25

2. Given text: “for each rose, a rose is a rose”
 - Construct the Huffman coding

Ziv-Lempel compression

- The problem with Huffman coding is that it requires knowledge about the data before encoding takes place.
 - Huffman coding requires frequencies of symbol occurrence before codeword is assigned to symbols
- **Ziv-Lempel compression**
 - Not rely on previous knowledge about the data
 - Rather builds this knowledge in the course of data transmission/data storage
 - Ziv-Lempel algorithm (called LZ) uses a **table of code-words created** during data transmission;
 - each time it replaces strings of characters with a reference to a previous occurrence of the string.

Lempel-Ziv Compression Algorithm

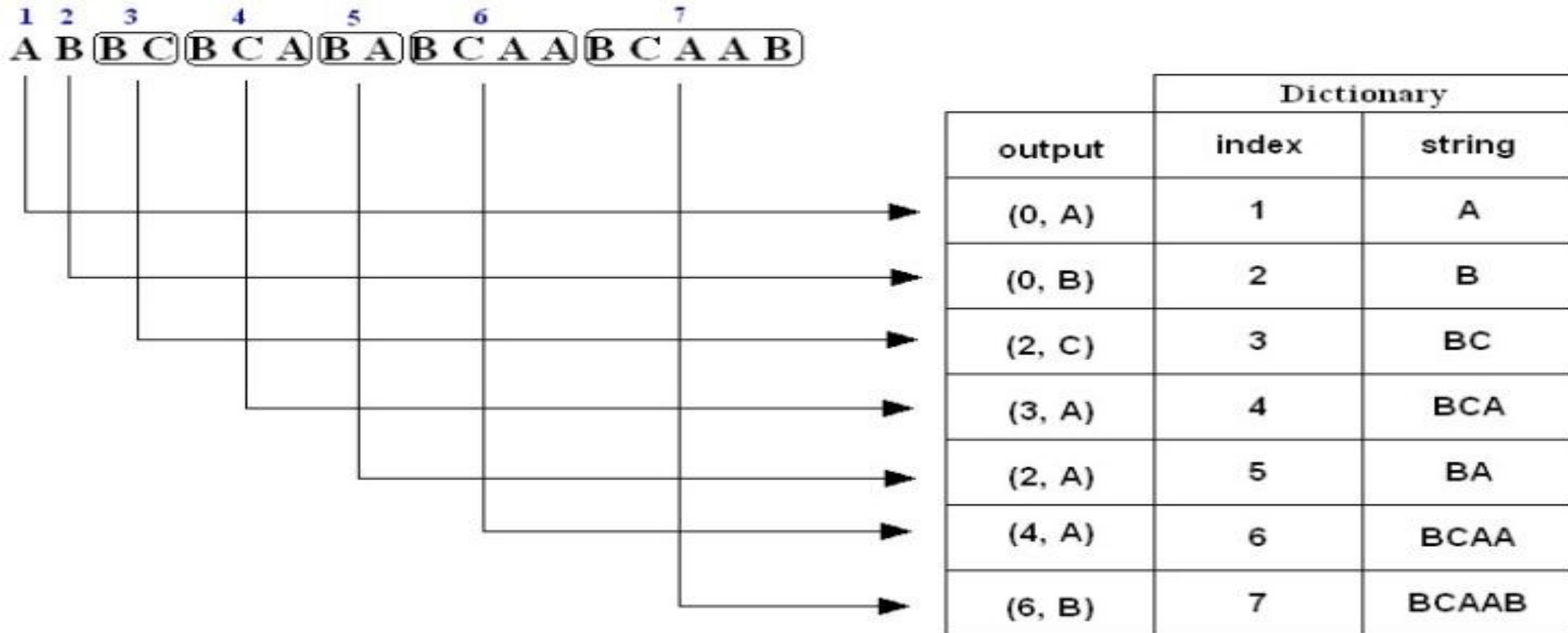
- The multi-symbol patterns are the form: $C_0C_1 \dots C_{n-1}C_n$. The **prefix** of a pattern consists of all the pattern symbols except the last: $C_0C_1 \dots C_{n-1}$

Lempel-Ziv Output: there are **three** options in assigning a code to each symbol in the list

- If one-symbol pattern is not in dictionary, assign (0, symbol)
- If multi-symbol pattern is not in dictionary, assign (dictionaryPrefixIndex, lastPatternSymbol)
- If the last input symbol or the last pattern is in the dictionary, assign (dictionaryPrefixIndex,)

Example: LZ Compression

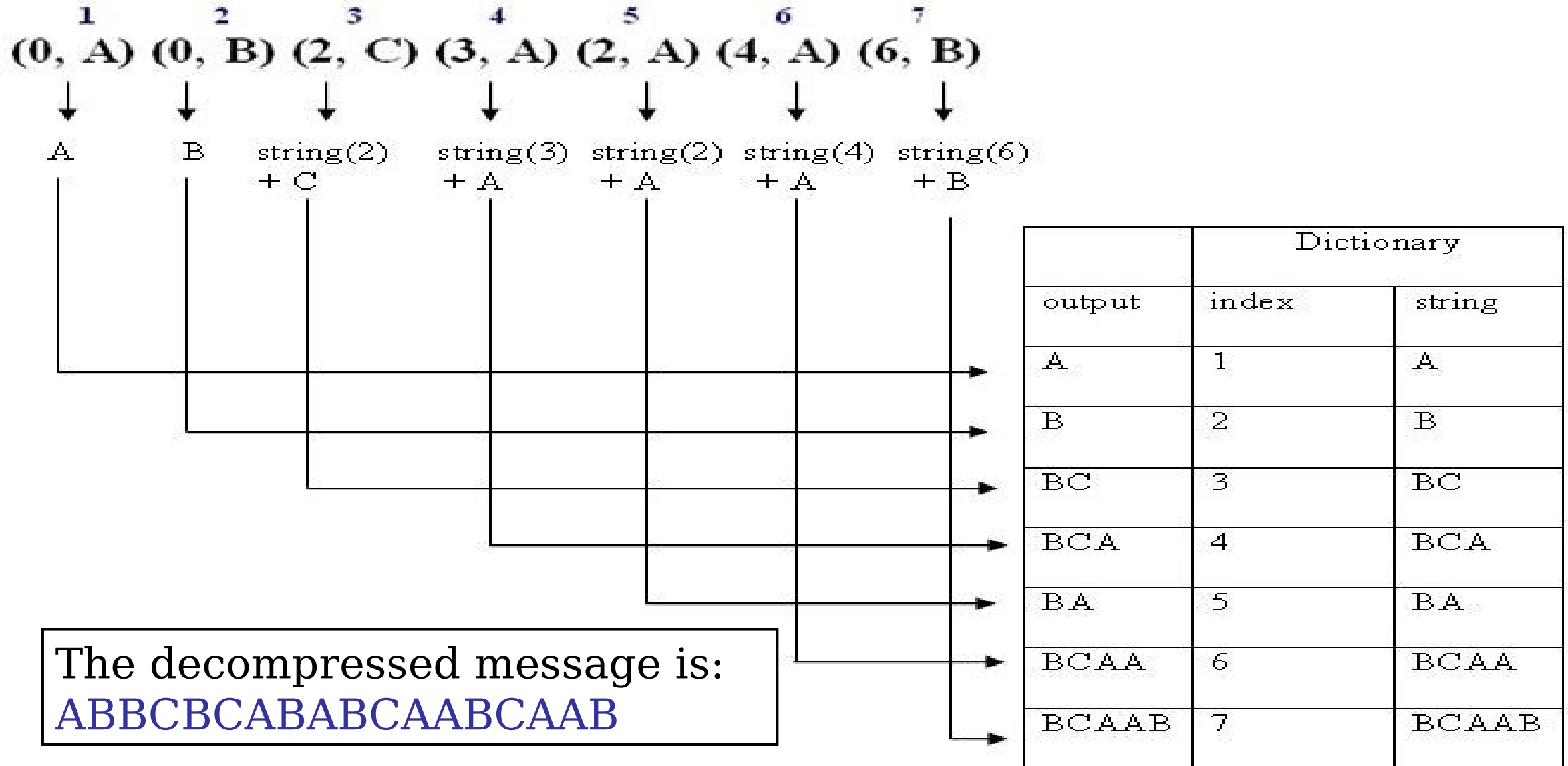
Encode (i.e., compress) the string **ABBCBCABABCAABCAAB** using the LZ algorithm.



The compressed message is: **0A0B2C3A2A4A6B**

Example: Decompression

Decode (i.e., decompress) the sequence: 0A0B2C3A2A4A6B



Exercise

Encode (i.e., compress) the following strings using the Lempel-Ziv algorithm.

1. Mississippi
2. ABBCBCABABCAABCAAB
3. SATATASACITASA.

Indexing structures: Exercise

- Discuss in detail theoretical and algorithmic concepts (including construction, various operations, complexity, etc.) on the following commonly used data structures

1. *Data structure vs. file structure*
2. *Arrays (fixed and dynamic arrays), sorted arrays*
3. *Records and linked list*
4. *Tree (AVL tree, Binary tree): balanced vs. unbalanced tree)*
5. *B tree and its variants (B+, B++ Tree, B* Tree),*
6. *Hierarchical Tree (like Quad Tree and its variants)*
7. *PAT-Tree and its variants*
8. *Disjoint tree: balanced and degenerate tree*
9. *Graph*
10. *Hashing,*
11. *Trie and its variants*

Indexing: Basic Concepts

- Indexing is used to speed up access to desired information from document collection as per users query such that
 - It enhances efficiency in terms of time for retrieval. Relevant documents are searched and retrieved quick

Example: author catalog in library

- An **index file** consists of records, called **index entries**.
- **Index files** are much smaller than the original file.
 - Remember **Heaps Law**: In 1 GB text collection the size of a vocabulary is only 5 MB (Baeza-Yates and Ribeiro-Neto, 2005)
 - This size may be further reduced by **Linguistic pre-processing** (like stemming & other normalization methods).
- The usual unit for indexing is the word
 - **Index terms** - are used to look up records in a file.

Major Steps in Index Construction

- **Source file:** Collection of text document
 - A document can be described by a set of representative keywords called index terms.
- **Index Terms Selection:**
 - Tokenize:** identify words in a document, so that each document is represented by a list of keywords or attributes
 - Stop words:** removal of high frequency words
 - Stop list of words is used for comparing the input text
 - Word stem and normalization:** reduce words with similar meaning into their stem/root word
 - Suffix stripping is the common method
 - Term relevance weight:** Different index terms have varying relevance when used to describe document contents.
 - This effect is captured through the **assignment of numerical weights to each index term** of a document.
 - There are different index terms weighting methods: **TF**, **TF*IDF**, ...
- **Output:** a set of index terms (vocabulary) to be used for **Indexing** the documents that each term occurs in.

Basic Indexing Process

Documents to
be indexed.



Friends, Romans,
countrymen.

Token
stream.

Tokenizer

Friends Roman countrymen
s

Modified
tokens.

Linguistic
preprocessin
g

friend roman countryma
n

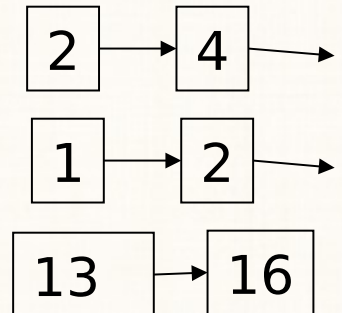
Index File
(Inverted
file).

Indexer

friend

roman

countryman



Building Index file

- An index file of a document is a file consisting of a list of index terms and a link to one or more documents that has the index term
 - A good **index file** maps each keyword K_i to a set of documents D_i that contain the keyword



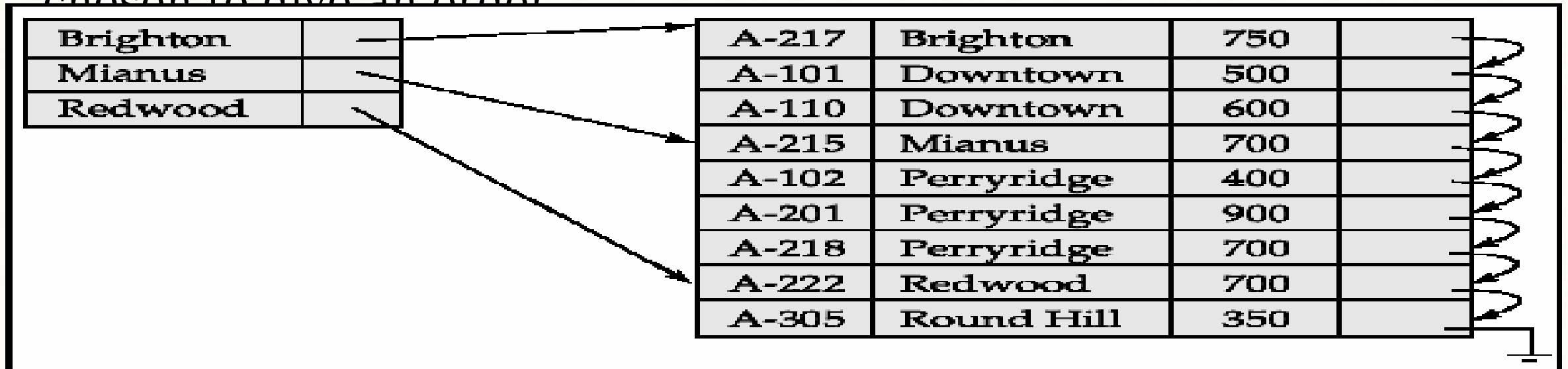
- Index file usually has index terms in a sorted order.
 - The sort order of the terms in the index file provides an order on a physical file
- An index file is list of search terms that are organized for associative look-up, i.e., to answer user's query:
 - In which documents does a specified search term appear?
 - Where within each document does each term appear? (There may be several occurrences.)
- For [organizing index file for a collection of documents](#), there are various options available:
 - Decide what data structure and/or file structure to use. Is it sequential file, inverted file, suffix array, signature file, etc. ?

Index file Evaluation Metrics

- Running time
 - Indexing time
 - Access/search time
 - Update time (Insertion time, Deletion time, modification time....)
- Space overhead
 - Computer storage space consumed.
- Access types supported efficiently.
 - is the indexing structure allows to access:
 - records with a specified term, or
 - records with terms falling in a specified range of values.

Sequential File

- Sequential file is the most primitive file structures.
 - It has no vocabulary as well as linking pointers.
- The records are generally arranged serially, one after another, but in **lexicographic** order on the value of some key field.
 - a particular attribute is chosen as primary key whose value will determine the order of the records.
 - when the first key fails to discriminate among records, a second key is chosen to give an order



Example:

- Given a collection of documents, they are parsed to extract words and these are saved with the Document ID.

Doc 1

I did enact Julius
Caesar I was killed
I the Capitol;
Brutus killed me.

Doc 2

So let it be with
Caesar. The noble
Brutus has told you
Caesar was ambitious

Sorting the Vocabulary

- After all documents have been tokenized, stopwords are removed, and normalization and stemming are applied, to generate index terms
- These index terms in sequential file are sorted in alphabetical order

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
I	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2



Sequential file

	Term	Doc No.
1	ambition	2
2	brutus	1
3	brutus	2
4	capitol	1
5	caesar	1
6	caesar	2
7	caesar	2
8	enact	1
9	julius	1
10	kill	1
11	kill	1
12	noble	2

Sequential File

- Its main advantages are:
 - easy to implement;
 - provides fast access to the next record using lexicographic order.
 - Instead of Linear time search, one can search in logarithmic time using binary search
- Its disadvantages:
 - difficult to update. Index must be rebuilt if a new term is added. Inserting a new record may require moving a large proportion of the file;
 - random access is extremely slow.
- The problem of update can be solved :
 - by ordering records by date of acquisition, than the key value; hence, the newest entries are added at the end of the file & therefore pose no difficulty to updating. But searching becomes very tough; it requires linear time

Inverted file

- A word oriented indexing mechanism based on sorted list of keywords, with each keyword having links to the documents containing it
 - Building and maintaining an inverted index is a relatively low cost risk.
 - On a text of n words an inverted index can be built in $O(n)$ time, n is number of keywords
- Content of the inverted file:
 - Data to be held in the inverted file includes :
 - The vocabulary (List of terms)
 - The occurrence (Location and frequency of terms in a document collection)

Inverted file

- **The occurrence:** contains one record per term, listing
 - Frequency of each term in a document, i.e. count number of occurrences of keywords in a document
 - TF_{ij} , number of occurrences of term t_j in document $\mathbf{d}_i$
 - DF_j , number of documents containing t_j
 - max_i , maximum frequency of any term in $\mathbf{d}_i$
 - N , total number of documents in a collection
 - CF_j , collection frequency of t_j in n_j
 -
 - Locations/Positions of words in the text

Inverted file

- **Why vocabulary?**

- Having information about vocabulary (list of terms) **speeds searching for relevant documents**

- **Why location?**

- Having information about the location of each term within the document **helps for:**
 - **user interface design: highlight location of search term**
 - **proximity based ranking: *adjacency* and *near* operators (in Boolean searching)**

- **Why frequencies?**

- Having information about frequency is used for:
 - calculating term weighting (like TF, TF*IDF, ...)
 - optimizing query processing

Inverted File

Documents are organized by the terms/words they contain

<i>Term</i>	<i>CF</i>	<i>Document ID</i>	<i>TF</i>	<i>Location</i>
auto	3	2	1	66
		19	1	213
		29	1	45
bus	4	3	1	94
		19	2	7, 212
		22	1	56
taxi	1	5	1	43
train	3	11	2	3, 70
		34	1	40

This is called an **index file**.

Text operations are performed before building the index.

Construction of Inverted file

- An inverted index consists of two files:
 - vocabulary file
 - Posting file

Advantage of dividing inverted file:

- Keeping a pointer in the vocabulary to the list in the posting file allows:
 - the vocabulary to be kept in memory at search time even for large text collection, and
 - Posting file to be kept on disk for accessing to documents

Inverted index storage

- Separation of inverted file into vocabulary and posting file is a good idea.
 - **Vocabulary:** For searching purpose we need only word list. This allows the vocabulary to be kept in memory at search time since the space required for the vocabulary is small.
 - The vocabulary grows by $O(n^\beta)$, where β is a constant between 0 – 1.
 - **Example:** from 1,000,000,000 documents, there may be 1,000,000 distinct words. Hence, the size of index is 100 MBs, which can easily be held in memory of a dedicated computer.
 - **Posting file** requires much more space.
 - For each word appearing in the text we are keeping statistical information related to word occurrence in documents.
 - Each of the postings pointer to the document requires an extra space of $O(n)$.
- How to speed up access to inverted file?

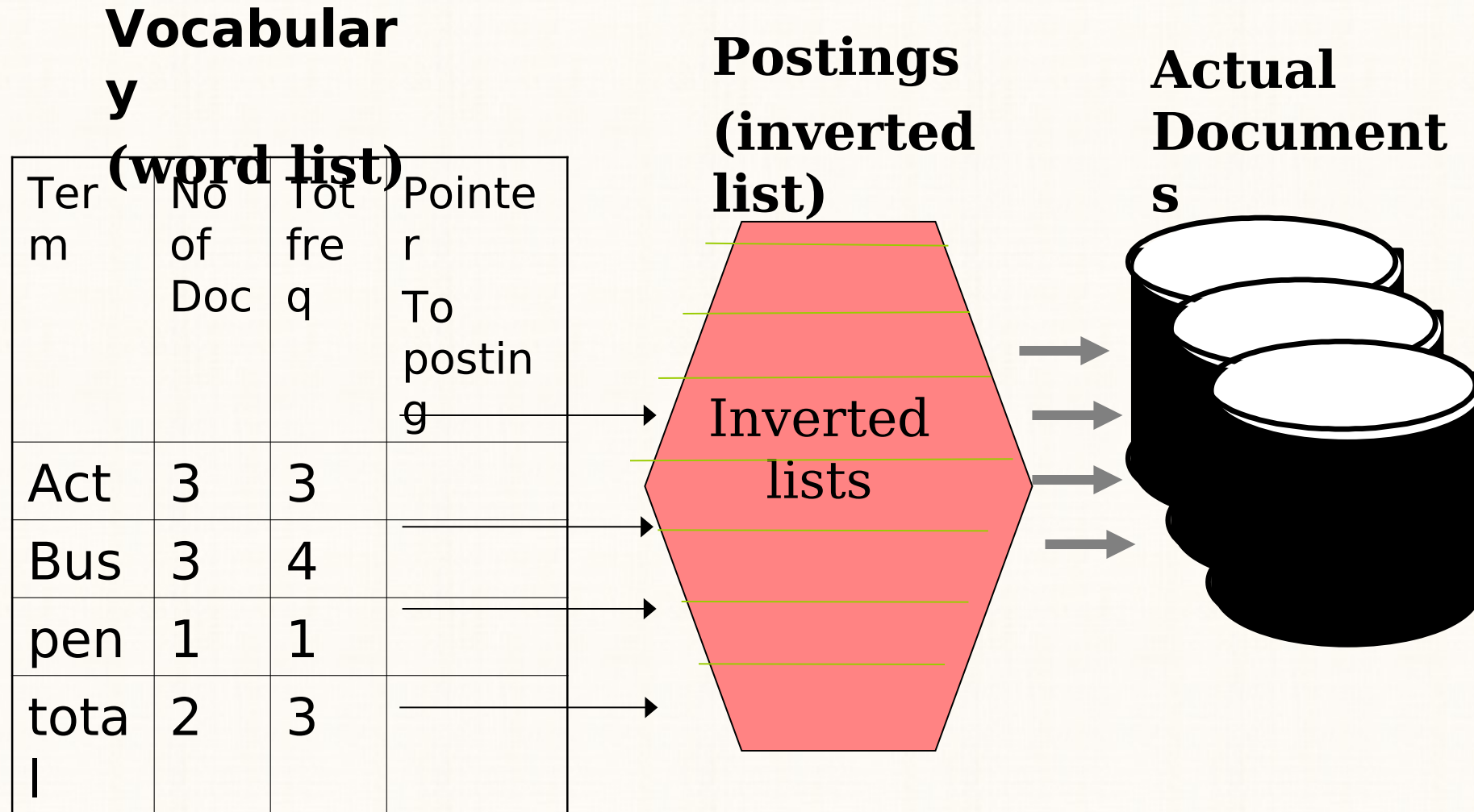
Vocabulary file

- A vocabulary file (Word list):
 - *stores* all of the **distinct** terms (keywords) that appear in any of the documents (in lexicographical order) and
 - For each word a pointer to posting file
- Records kept for each term j in the word list contains the following:
 - term j
 - number of documents in which term j occurs (DF_j)
 - Total frequency of term j (CF_j)
 - pointer to postings (inverted) list for term j

Postings File (Inverted List)

- For each distinct term in the vocabulary, stores a list of pointers to the documents that contain that term.
- Each element in an inverted list is called a posting, i.e., the occurrence of a term in a document
- It is stored as a separate inverted list for each column, i.e., a list corresponding to each term in the index file.
 - Each list consists of one or many individual postings related to Document ID, TF and location information about a given term i

Organization of Index File



Example:

- Given a collection of documents, they are parsed to extract words and these are saved with the Document ID.

Doc 1

I did enact Julius
Caesar I was killed
I the Capitol;
Brutus killed me.

Doc 2

So let it be with
Caesar. The noble
Brutus has told you
Caesar was ambitious

Sorting the Vocabulary

- After all documents have been tokenized the inverted file is sorted by terms

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
I	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

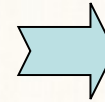


Term	Doc #
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
has	1
I	1
I	1
I	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Remove stopwords, apply stemming & compute term frequency

- Multiple term entries in a single document are merged and frequency information added
- Counting number of occurrence of terms in the collections helps to compute TF

Term	Doc #
ambition	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
enact	1
julius	1
kill	1
kill	1
noble	2



Term	Doc #	TF
ambition	2	1
brutus	1	1
brutus	2	1
capitol	1	1
caesar	1	1
caesar	2	2
enact	1	1
julius	1	1
kill	1	2
noble	2	1

Vocabulary and postings file

The file is commonly split into a Dictionary and a Posting file

vocabulary

Term	Doc #	TF
ambition	2	1
brutus	1	1
brutus	2	1
capitol	1	1
caesar	1	1
caesar	2	2
enact	1	1
julius	1	1
kill	1	2
noble	2	1

Term	DF	CF
ambitious	1	1
brutus	2	2
capitol	1	1
caesar	2	3
enact	1	1
julius	1	1
kill	1	2
noble	1	1

posting

Doc #	TF
2	1
1	1
2	1
1	1
1	1
2	2
1	1
1	1
1	2
2	1

Pointers

Complexity Analysis

- The inverted index can be built in $O(n)$ time.
 - n is number of vocabulary terms
- Since terms in vocabulary file are sorted searching takes logarithmic time.
- To update the inverted index it is possible to apply **Incremental indexing** which requires $O(k)$ time, k is number of new index terms

Exercises

- Construct the inverted index for the following document collections.

Doc 1 : New home to home sales forecasts

Doc 2 : Rise in home sales in July

Doc 3 : Home sales rise in July for new homes

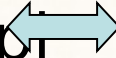
Doc 4 : July new home sales rise

Suffix Tree and Suffix Array

Reading Assignment

Suffix trie

- What is Suffix? A suffix is a substring that exists at the end of the given string.
 - Each position in the text is considered as a text suffix
 - If $\text{txt} = t_1 t_2 \dots t_i \dots t_n$ is a string, then $T_i = t_i, t_{i+1} \dots t_n$ is the *suffix* of txt that starts at position i ,

• **Example:** $\text{txt} = \text{mississippi}$ 

$T_1 = \text{mississippi};$

$T_2 = \text{ississippi};$

$T_3 = \text{ssissippi};$

$T_4 = \text{sissippi};$

$T_5 = \text{issippi};$

$T_6 = \text{ssippi};$

$T_7 = \text{sippi};$

$T_8 = \text{ippi};$

$T_9 = \text{ppi};$

$T_{10} = \text{pi};$

$T_{11} = \text{i};$

$\text{txt} = \text{GOOGOL}$

$T_1 = \text{GOOGOL}$

$T_2 = \text{OOGOL}$

$T_3 = \text{OGOL}$

$T_4 = \text{GOL}$

$T_5 = \text{OL}$

$T_6 = \text{L}$

Suffix trie

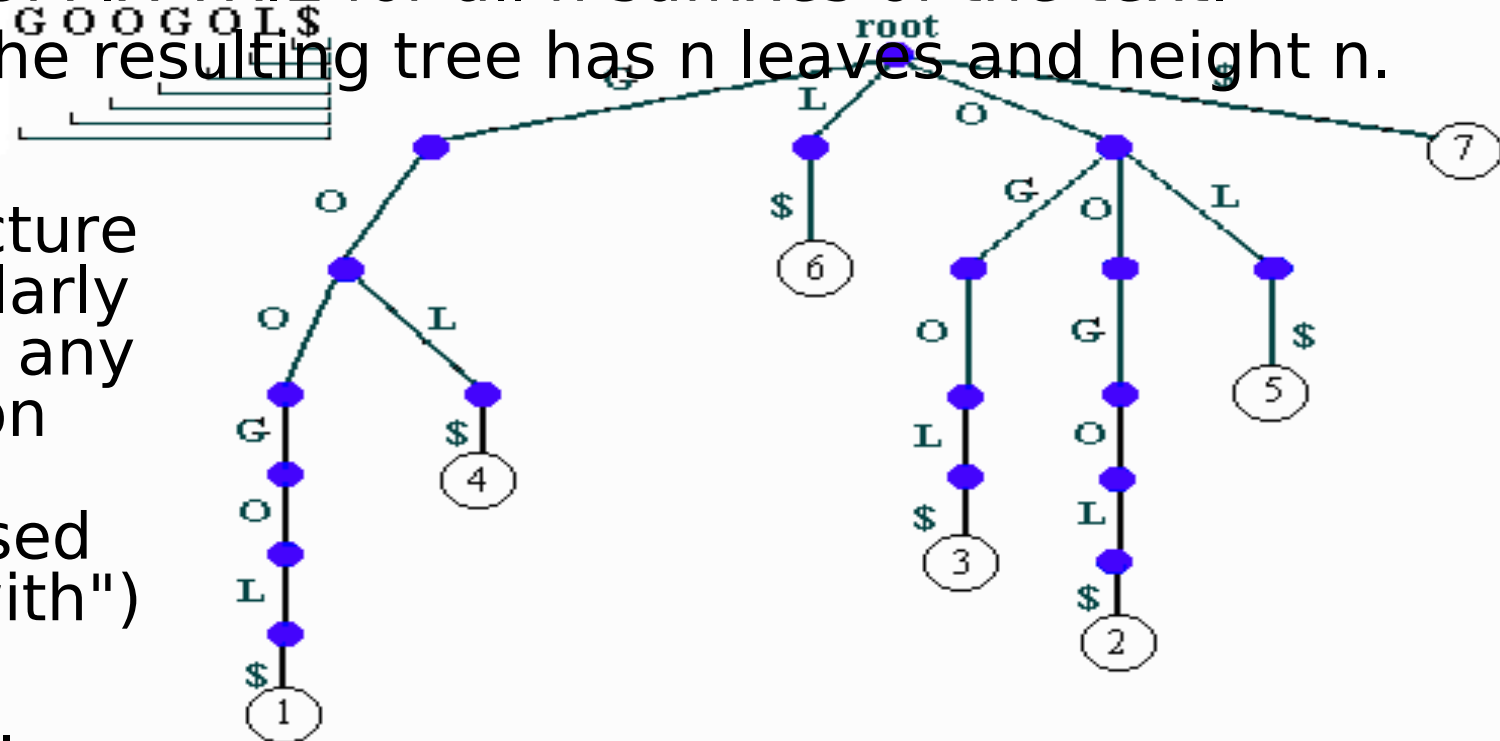
- A suffix trie is an ordinary trie in which the input strings are all possible suffixes.
 - **Principles:** The idea behind suffix TRIE is to assign to each symbol in a text an index corresponding to its position in the text. (i.e: First symbol has index 1, last symbol has index n (#of symbols in text)).
 - To build the suffix TRIE we use these indices instead of the actual object.
- The structure has several advantages:
 - It requires less storage space.
 - We do not have to worry how the text is represented (binary, ASCII, etc).
 - We do not have to store the same object twice (no duplicate).

Suffix Trie

- Construct SUFFIX TRIE for the following string:
GOOGOL
- We begin by giving a position to every suffix in the text starting from left to right as per characters occurrence in the string.

TEXT : G O O G O L \$
POSITION : 1 2 3 4 5 6 7

- Build a SUFFIX TRIE for all n suffixes of the text.
 - Note: The resulting tree has n leaves and height n.

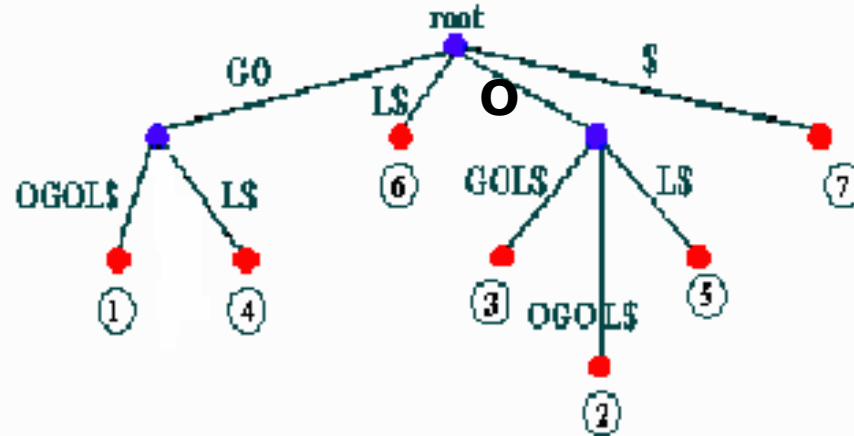


- This structure is particularly useful for any application requiring prefix based ("starts with") pattern matching.

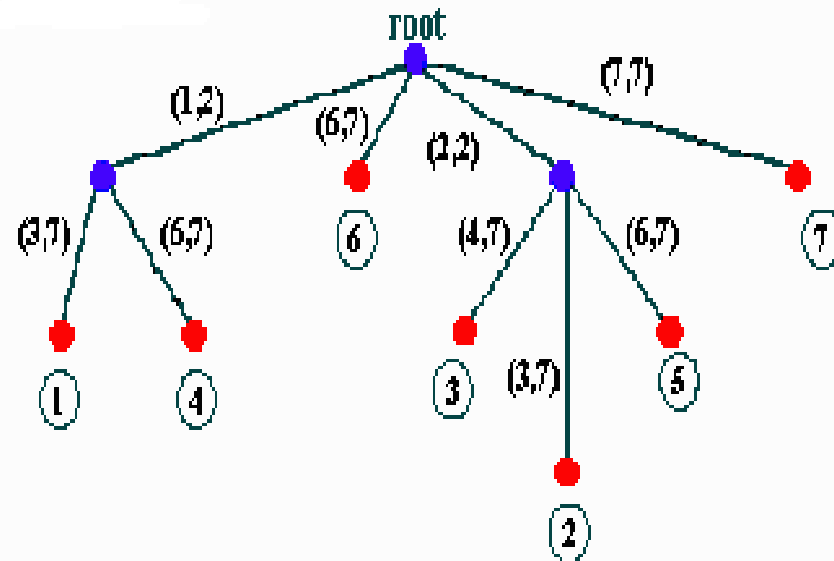
Suffix tree

- A suffix tree is a member of the trie family. It is a Trie of all the proper suffixes of S
 - The suffix tree is created by compacting unary nodes of the suffix TRIE.
- We store pointers rather than words in the leaves.
 - It is also possible to replace strings in every edge by a pair (a,b), where a & b are the beginning and end index of the string. i.e.
 - (3,7) for OGOL\$
 - (1,2) for GO
 - (7,7) for \$

Suffix Tree: GOOGOL\$



Suffix Tree with edges labeled (a,b)

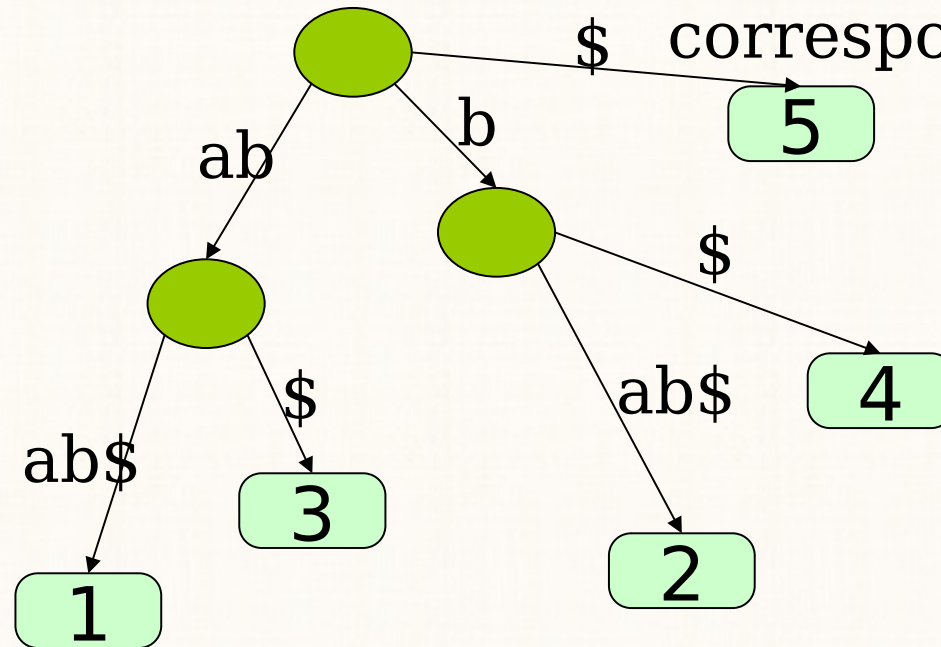


Example: Suffix tree

- Let $s = \text{abab}$, a suffix tree of s is a compressed trie of all suffixes of $s = \text{abab}\$$

- We label each leaf with the starting point of the corresponding suffix.

{
5 \$
4 b\$
3 ab\$
2
bab\$
1
abab\$
}



Complexity Analysis

- The suffix tree for a string has been built in $O(n^2)$ time.
- The search time is also linear in the length of string S .
- Searching for a substring[1..m], in string[1..n], can be solved in $O(m)$ time
 - It requires to search for the length of the string $O(|S|)$.

Generalized suffix tree

- Given a set of strings S , a generalized suffix tree of S is a compressed trie of all suffixes of $s \in S$
 - To make suffixes prefix-free we add a special char, $\$,$ at the end of s . To associate each suffix with a unique string in S add a different special symbol to each s
- Build a suffix tree for the string $s_1\$s_2\#$, where $\$$ and $\#$ are a special terminator for s_1, s_2 .
- Ex.: Let $s_1 = abab$ & $s_2 = aab$, a generalized suffix tree for s_1

& s_2 is:

• $\{$

• $\$$

$\#$

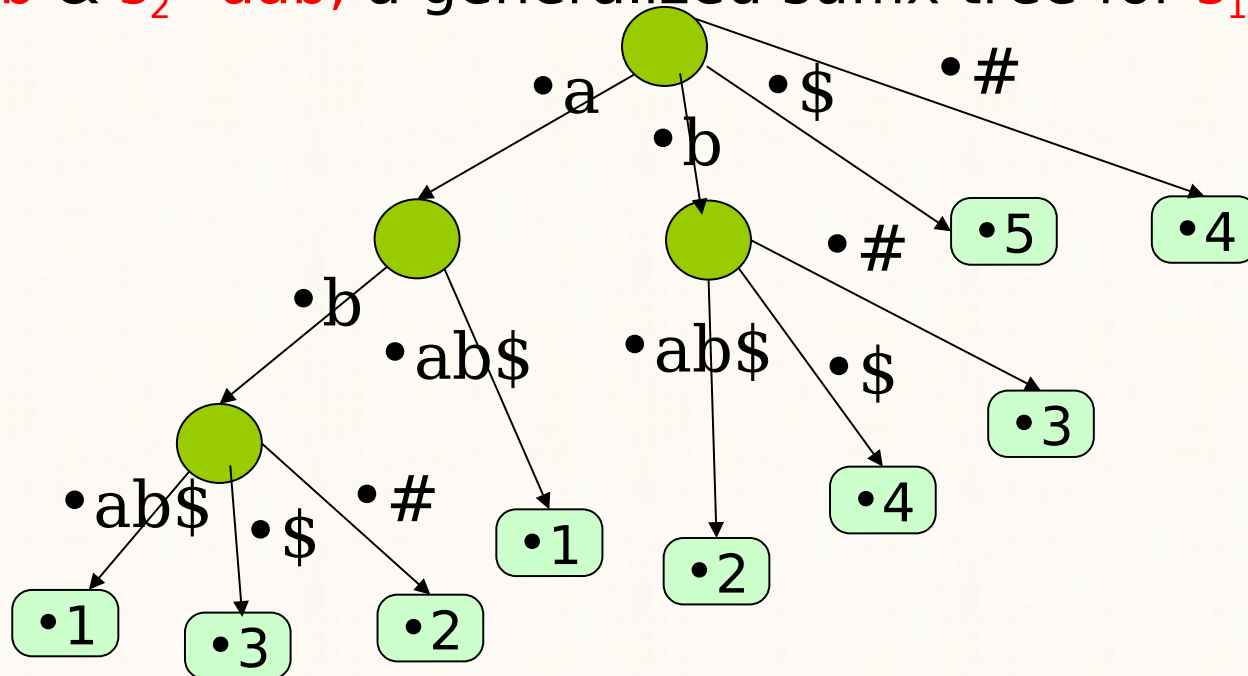
• $b\$$

$b\#$

• $ab\$$

$ab\#$

• $bab\$$



Search in suffix tree

- Searching for all instances of a substring S in a suffix tree is easy since any substring of S is the prefix of some suffix.
- Pseudo-code for searching in suffix tree:
 - Start at root
 - Go down the tree by taking each time the corresponding path
 - If S correspond to a node then return all leaves in sub-tree
 - the places where S can be found are given by the pointers in all the leaves in the subtree rooted at x.
 - If S encountered a NIL pointer before reaching the end, then S is not in the tree

Example:

- If S = "GO" we take the GO path and return: GOOGOL\$, GOL\$.
- If S = "OR" we take the O path and then we hit a NIL pointer so "OR" is not in the tree.

Suffix Tree Applications

- Suffix Tree can be used to solve a large number of string problems that occur in:
 - text-editing,
 - free-text search,
 - etc.
- Some examples of string problems are given below.
 - String matching
 - Longest Common Substring
 - Longest Repeated Substring
 - Palindromes
 - etc..

Drawbacks

- Suffix trees consume a lot of space
 - Even if word beginnings are indexed, space overhead of 120% - 240% over the text size is produced. Because depending on the implementation each nodes of the suffix tree takes a space (in bytes) equivalent to the number of symbols used.
 - How much space is required at each node for English word indexing based on alphabets a to z.
- How many bytes required to store MISSISSIPI ?

Suffix array

- A suffix array is more compact than a suffix tree.
 - Suffix arrays are a space efficient implementation of suffix trees
- Like suffix tree, a suffix array is a sorted list of the suffixes of a given string in *lexicographical order*.
 - The sorted list is presented as an array of integers that identify the suffixes in order.
 - This allows a *binary search* or fast substring search.
- Main drawbacks:
 - Its costly construction process.
 - The need for the document/text to be readily available at query time

Building suffix array

- Procedure:
 - Identify suffixes of the given string
 - Sort the suffixes lexicographically
 - Store indices of all the suffixes in a table.
- The suffix array gives the indices of the suffixes in sorted order
- A suffix array can be constructed in $O(n \log n)$ time, where n is the length of the string, by sorting the suffixes
- Example: Consider the string "good".
 - At the end, a special character is usually appended to the string.
 - In lexicographical order, the suffixes are "d\$", "good\$", "od\$" and "ood\$".
 - The suffix array is [4, 1, 3, 2, 5].

Building a suffix array

- **Example:**

- given the string $S = \text{GOOGOL}$, construct suffix array
- Sort the suffixes in *lexicographical order* and store

suffixes		Indices	ptr
GOL\$	S[0]	4	
GOOGOL\$	S[1]	1	
L\$	S[2]	6	
OGOL\$	S[3]	3	
OL\$	S[4]	5	
OOGOL\$	S[5]	2	
\$	S[6]	7	

Annotations:

- Double arrow from "Not stored" to OL\$ (S[4])
- Double arrow from "Stored" to OOGOL\$ (S[5])

Building suffix array

- How can we build suffix array for multiple strings, like GOOD and GOOGOL ?

Exercise

- Construct suffix array for the string

s = abab

- Identify suffixes and sort them lexicographically:

ab\$, abab\$, b\$, bab\$, \$

- The suffix array gives the indices of the suffixes in sorted order

3	1	4	2	5
---	---	---	---	---

How do we search for a pattern?

- If P occurs in T then all its occurrences are consecutive in the suffix array.
- Do a binary search on the suffix array
- Takes $O(\log n)$ time
- **Example 1:** search for 'good' in the suffix array constructed for 'GOOGOL'.
- **Exercise:** Let the string given is $S = \text{mississippi}$, construct suffix array and search for
 - (i) ppi
 - (ii) issa

Example

• Let $S =$
mississippi

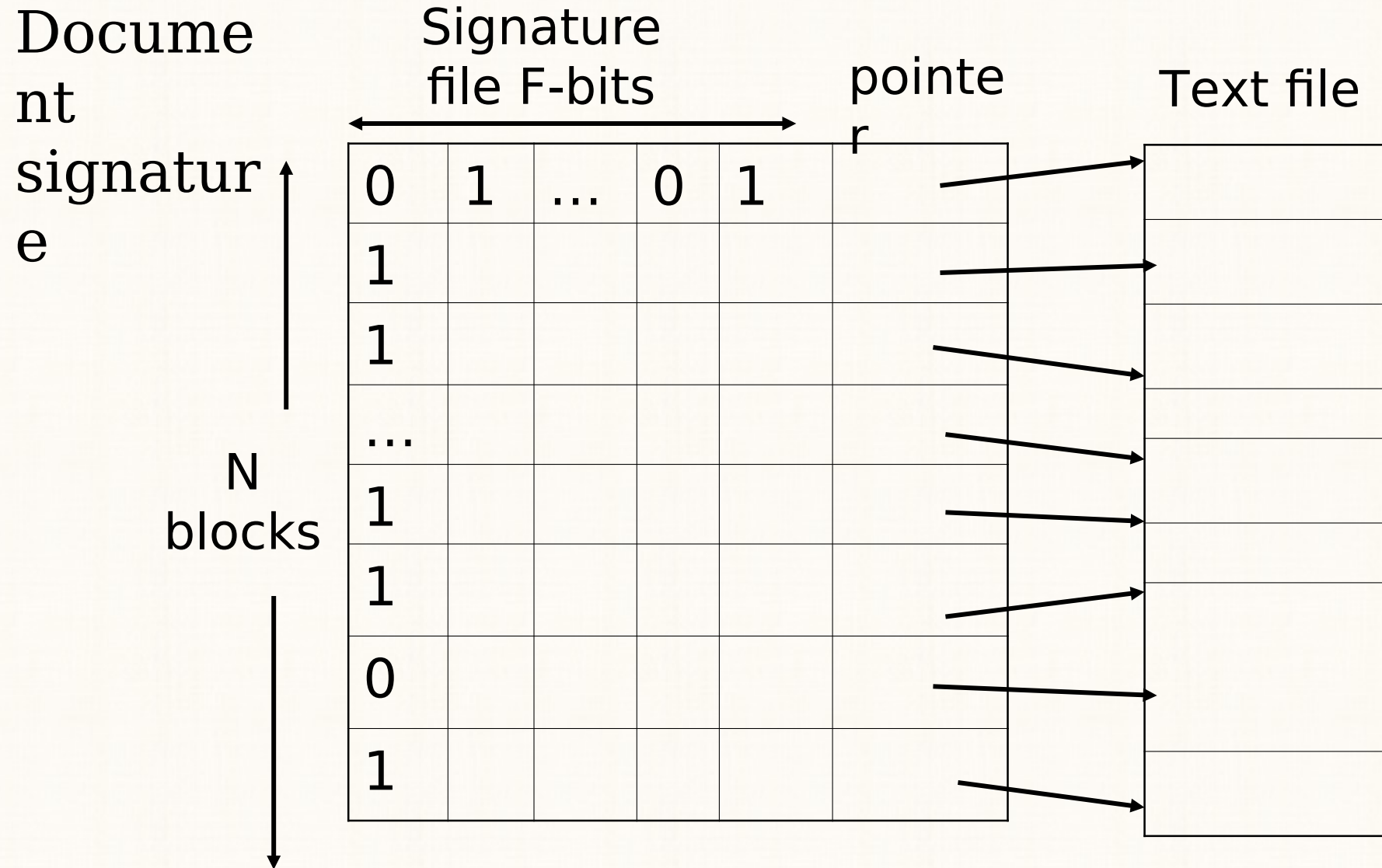
• Let $P =$ issa

L →	11	i
	8	ippi
	5	issippi
	2	ississippi
	1	mississippi
M →	10	pi
	9	ppi
	7	sippi
	4	sisippi
	6	ssippi
R →	3	ssissippi

Signature file

- Word-oriented index structures based on **hashing**
- How to build signature file
 - Hash each word to allocate fixed sized **F-bits** vector (word signature)
 - Divide the text in **blocks** of **N** words each
 - Assign **F-bits** masks for each text block of size **N** (document signature)
 - This is obtained by bitwise **ORing** the signatures of all the words in the text block.
- Efficient to search for phrases
- Hence the signature file is no more than the sequence of bit masks of all blocks (plus a pointer to each block).

Structure of Signature File



Example

- Given a text: "A text has many words. Words are made from letters"

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

- Text
Signature:

1110101	011110	1011111
	0	

- Signature (hash) function:

- $h(\text{text}) = 1000101$
- $h(\text{many}) = 0110101$
- $h(\text{word}) = 0111100$
- $h(\text{made}) = 0010111$
- $h(\text{letter}) = 1001011$

- Block 4: 001100

• OR 100001

- 101101

Searching

- During query processing:
 - Hash the query to a F-bit mask Q
 - Compare query signature with document signature of each block, that is
 - Bit-wise ANDing all the bits set in the query with bit masks B_i of all the text block
 - If all corresponding 1-bits are “on” in document signature, document *probably* contains that term, that is
 - If $Q \& B_i = Q$, all the bits set in Q are also set in B_i and therefore the text block may contain the word
- The main idea of signature file is that if a word is present in a text block, then all the bits set in its signature are also set in the bit mask of the text block
 - Hence if a bit is set in the mask of the query word and not in the mask of the text block, then the word is not present in the text block

Signature file trivia

- Signature files leads to possible mismatches.
 - It is possible that all the corresponding bits are set even though the word is not there. This is called false drop.
- False drop or false positive
 - Document that is retrieved by a search but is not relevant to the searcher's needs
 - False drops occur because of words that are written the same but have different meanings.
 - **Example:** 'squash' refer to a game, a vegetable or an action