

## **INFORMATION TO USERS**

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning  
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA  
800-521-0600

**UMI<sup>®</sup>**



**An Inverted Index Generator  
for CINDI**

**Hudong Li**

**A Major Report  
in  
The Department  
of  
Computer Science**

**Presented in Partial Fulfillment of the Requirements  
for the Degree of Master of Computer Science at  
Concordia University  
Montreal, Quebec, Canada**

**April 2003**

**© Hudong Li, 2003**



**National Library  
of Canada**

**Acquisitions and  
Bibliographic Services**

**395 Wellington Street  
Ottawa ON K1A 0N4  
Canada**

**Bibliothèque nationale  
du Canada**

**Acquisitions et  
services bibliographiques**

**395, rue Wellington  
Ottawa ON K1A 0N4  
Canada**

*Your file Votre référence*

*Our file Notre référence*

**The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.**

**The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.**

**L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.**

**L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.**

0-612-77916-5

**Canada**

# **Abstract**

## **An Inverted Index Generator for CINDI**

Hudong Li

Human maintained search engines are expensive, slow to update, and cannot cover all the web pages. Automated search engines that rely on keyword matching usually return too many low quality results, with most users only looking at the first few tens of the search results. Because search engine development has gone on at companies with little publication of technical details, it is a challenging task to develop a search engine. The use of hypertextual information can help to improve search quality. This report addresses the question of how to build an inverted index for a search system that can use the additional information presented in hypertext to produce better search results. This report is part of the work of the Concordia INdexing and DIscovery (CINDI) Digital Library System. In this report, we summarize the research work I have done; we present some implementation issues for the project; and present the data structures that can be used in indexing web pages. The design decision was driven by the desire to have a reasonable compact data structure, and the ability to fetch a record in few disk seeks during a search. This project has been implemented in C++ on Linux platform.

## **Acknowledgements**

I would like to express my sincere thanks to my major report supervisor, Dr. Bipin C. Desai for his guidance and encouragement during the period of research that has lead to this report.

I would also like to thank my friends, Mr. Jack Klein and Guoshun Zhan for their helpful suggestions during the writing of this report.

I gratefully acknowledge the Quebec government for the financial support during my studies at Concordia University.

# Contents

List of Figures	ix
<b>1. Introduction</b>	<b>1</b>
1.1 Discovery Problem .....	1
1.2 About This Project .....	2
1.3 About This Report .....	3
<b>2. Background</b>	<b>4</b>
2.1 Information Retrieval .....	4
2.2 Google .....	5
2.2.1 Web Crawling Subsystem .....	5
2.2.2 Indexing Subsystem .....	6
2.2.3 Searching Subsystem .....	9
2.3 Inverted Index .....	10
2.3.1 Original Meaning .....	10
2.3.2 Application .....	11
2.4 HTML Document .....	13
2.5 Data Structures and Algorithms .....	13
2.5.1 Linked List .....	13
2.5.2 Internal Sort .....	14
2.5.3 String-Based Pattern Matching .....	14
2.5.4 Bit Field Structure .....	15
2.5.5 File Operation .....	15

<b>3. Implementation of Inverted Index</b>	<b>20</b>
3.1 Major Data Structures .....	20
3.1.1 Repository .....	20
3.1.2 Lexicon .....	21
3.1.3 Hit Lists .....	21
3.1.4 Forward Index .....	22
3.1.5 Inverted Index .....	23
3.2 Document Processing .....	24
3.2.1 Interface .....	24
3.2.2 Extract a Page from Repository .....	26
3.2.3 Extract Items from a Page .....	26
3.2.4 Search a Given Word in the Temporary Array .....	28
3.2.5 Construct Nodes and Add Them to the Hit Lists .....	28
3.2.6 Construct Forward Barrels .....	31
3.2.7 Generate Inverted Barrel from Forward Barrel .....	32
3.2.8 Summary .....	32
3.3 Lexicon .....	33
3.3.1 Implementation Strategies .....	33
3.3.2 Lexicon Data .....	34
3.3.3 Computer Aided Lexicon Data Generation .....	34
3.3.4 Use and Maintain the Lexicon .....	37
3.3.5 Trimming of Punctuation .....	38



3.3.6	About Stemming Process .....	38
3.3.7	Stop Words .....	40
3.4	Parsing HTML Document .....	41
3.4.1	Extract Single Document .....	41
3.4.2	Extract Title .....	42
3.4.3	Extract Anchor Text and Meta Content .....	43
3.4.4	Extract Plain Text .....	46
3.5	Sorting .....	47
3.5.1	Build Hit Lists .....	47
3.5.2	Merge Sort Forward Barrels .....	47
3.6	Main Function .....	51
3.7	Test .....	52
<b>4.</b>	<b>Conclusion</b> .....	<b>54</b>
4.1	Contributions for the Project .....	54
4.2	Conclusion .....	54
	<b>References</b> .....	<b>56</b>
	<b>Appendices</b>	
A	Repository Data Sample .....	58
B	Lexicon Data Sample .....	64
C	Source Code and Data File List .....	67
D	Forward Barrel Setup Method .....	68



## List of Figures

Figure 2-1 Vector file and its inverted vector file .....	10
Figure 2-2 An inverted index example .....	12
Figure 3-1 Hits structure .....	22
Figure 3-2 Forward barrel structure .....	22
Figure 3-3 Inverted barrel structure .....	23
Figure 3-4 Search engine's internal interfaces .....	25
Figure 3-5 Data flow diagram .....	27
Figure 3-6 Element structure of the hit list .....	28
Figure 3-7 Temporary array and linked lists of a page. ....	29
Figure 3-8 Lexicon as a structured array in main memory .....	35
Figure 3-9 Relationship between the amount of words and documents .....	36
Figure 3-10 Lexicon as a record file in disk .....	39
Figure 3-11 Forward barrel example .....	48
Figure 3-12 Index and inverted barrel after the first step of merge .....	49
Figure 3-13 Index and inverted barrel after the second step of merge .....	50

# **Chapter 1**

## **Introduction**

### **1.1 Discovery Problem**

People usually use search engines to search needed material on the Internet. However, finding relevant resources on the Internet may take a fair amount of time. Human maintained search engines are expensive to maintain. Because the database cannot be updated frequently, people may find the content out of date. Automated search engines that rely on keyword matching usually return too many low quality matches. In order to search efficiently, Meta information must be used in the indexing process as well.

CINDI (Concordia INdexing and DIScovery System) system, which is under development, aims to provide a mechanism to support crawling, indexing, and searching. CINDI is a virtual library supporting advanced search and targeted to improve the web search quality.

## **1.2 About This Project**

The aim of the CINDI project, which is proposed by Dr. Desai [ 1, 2 ], is to build a system that can do effective search on documents of different formats.

There have been different steps in the development of CINDI. Firstly, the semantic header, which is used for indexing, was provided by the author of the resource. Secondly, a system to generate the semantic header and the subject of the document was developed [ 3 ].

The web is a vast collection of completely uncontrolled documents mainly based on hypertexts. Information retrieval techniques need to be extended to deal with the web. Our project is to provide CINDI system with hypertext indexing ability in order to support online query efficiently. CINDI supports two types of query requests. The search could be on title, author(s), subject, keyword(s), etc. In the search, user can search using a word, substring, or exact phrase. The inverted index presented in this report is an aid to support keyword search.

For the time being, very little academic research has been done on large-scale hypertext search engine. We seldom see a detailed technical report for a search engine's development, since much of it is a business secret.

## **1.3 About This Report**

In this report, the fundamentals of the search engine development including the structure of a search engine, string-based pattern matching, bit and file processing techniques are given in chapter 2. In chapter 3, we present project major data structures and project processing principals. Some implementation issues are also discussed in this chapter. Finally, chapter 4 is the conclusion.

## **Chapter 2**

### **Background**

#### **2.1 Information Retrieval**

Work in information retrieval goes back to many years before. Most of the research is on well-controlled collections such as collections of scientific papers. Things that work well on small and well-controlled collections often do not produce “good” results in the context of the web [ 4 ]. There are huge differences between the web and well-controlled collections. Information retrieval work needs to be extended to deal with the web.

For the web, we can make use of internal Meta information. For example, we can keep track of some visual presentation details, such as font size of words. Words in a large or bolder font are more important than others. We can also make use of external Meta information. External Meta information includes things like reputation of the source, update frequency, and citation. PageRank is used in Google search engine developed at Stanford University [ 4 ].

## **2.2 Google**

Google was originally a Stanford University project called BackRub carried out by students Larry Page and Sergey Brin [ 26 ]. By 1998, the name had been changed to Google, and the project jumped off the campus and became the private search engine company, Google Inc.

Google supports text-based searches. It belongs to automatic search engine. It can cover a much larger proportion of the web sites - 30% of the web or more [ 4 ]. Text-based search engines sift through the text of millions of web pages and look for the word(s) or phrase(s). If a page contains the word(s) or phrase(s), it is a hit.

### **2.2.1 Web Crawling Subsystem**

Google uses crawling subsystem to download web pages. It starts on a page and follows each link on the page to locate other pages or documents. This strategy is like the width-first search algorithm. The administration interface provides fields with multiple URLs and specifies which web hosts and domain names the robot is allowed to access, and keep it from crawling inappropriate hosts.

Google has gone through a number of different versions. In version 3.3, the crawling has some intelligent priorities, allowing it to crawl the pages with high PageRank .



The major function of the web crawling subsystem is to gather web pages/files from remote WWW servers. It may involve the whole Internet. This gathering task is usually carried out automatically by a program called *web robot*, whose major functions can be briefly described as follows:

1. Starting from a single URL, it downloads the web page from the remote web site. It parses and adds all the URLs in this page to a queue for future downloading.
2. For the web page from (1), the HTML source code is stored in the repository. If the web page is in other formats, a converter is applied in order to obtain HTML code.
3. After (2) is done, it gets a new URL from the queue, checks if the new URL is already downloaded. If so, it gets a next URL from the queue, and repeats (3). Otherwise, repeats (1).

In summary, a web robot continuously requests files from remote web sites and stores the HTML documents in the repository for further processing.

Crawling involves interacting with hundreds of thousands of web servers. In a fast-distributed crawling system, the URL server sends the URLs to a number of crawlers. A fast server side script language is used, for example, script language Python.

### **2.2.2 Index Subsystem**

The index subsystem defines how contents gathered from remote web sites is internally stored and managed in a text database to support search/query efficiently. It depends on the logical data structure of the text entities and the physical organization in the database.

The index subsystem is tightly coupled with the search subsystem, as its sole purpose is to speed up the free text search/query process conducted by the search subsystem. A common approach is to build an inverted index for the documents [ 7 ].

The major decisions to be made for the index subsystem are as follows:

1. What compression scheme will be used to store the text and its index.
2. Do we keep both original text and its index in the database, or just the index? In order to support certain advanced search capabilities such as exact phrase search, the original text must be stored together with its index. This will increase the need for storage space.
3. Index modes: batch indexing, or incremental indexing [ 17 ]. Batch indexing only updates index in a batch mode after a bulk of text has been loaded. Incremental indexing allows the indexing process to be done incrementally.
4. Case sensitivity, symbols, or numbers. Support of case sensitivity and symbols in the index will increase the storage space requirements.

Performance of a web search system will largely depend on its index structure. A good index structure should have small storage requirement, thus speeding query processing significantly.

Goggle's main steps in indexing are as follows[ 4 ]:

*Converting* Converters are applied to the downloaded files to obtain HTML documents.

For example, if the file is in Microsoft Word format, a converter is used to convert it from Word format to HTML format.

*Parsing HTML documents*    Extracting needed words from HTML documents.

*Indexing documents into forward barrels*    Every document is encoded into a number of barrels. We use a binary file as a barrel. By using an in-memory lexicon, a word (except noise word) occurrence is coded into a hit, and is written into a forward barrel. If a word is not in the lexicon, a log file is used to register this word.

*Sorting*    In order to generate the inverted index, a program processes each forward barrel, sorts it by wordID to produce an inverted barrel.

The index system needs to be designed to be robust in order to deal with vast amounts of errors.

Google does not do incremental updates in an index [ 4 ]. However, it is possible to remove URLs from the searchable index without reindexing. In version 3.3, Google can search on a combination of two collections: a main collection of documents is not changed often, and a changeable collection that is often updated, such as breaking news. The changeable collection can be indexed continuously, while the main collection index is updated monthly.

This search engine has reporting features. It provides a status report showing what was indexed and what went wrong. It provides options to see one or many hosts, the URLs, the errors and the successes, so it can explain what happened when indexing a site. For example, certain page of the site may be reported to have error.

### **2.2.3 Search Subsystem**

The search subsystem accepts queries from a Web-based search interface and schedules, partitions, and executes them against the indexed database to locate URLs and their associated attributes that satisfy the query criteria. Partitions are based on the wordid. It also deals with the I/O, caching, assembling, sorting, and performance-related system activity. Based on what content is indexed and the scheme used in the index subsystem, it implements and supports different search algorithms to provide one word, multi-words, and exact phrase search capabilities for a text retrieval system. Together with the crawling and index subsystems, the search subsystem supports basic and advanced search capabilities [ 5 ].

The search subsystem provides quality search results. There is a query evaluation process in searching. The main goal of the query evaluation process is to generate search results efficiently.

Because of transfer efficiency, the search result is in XML format. The result layout is HTML code generated by the sever.

## 2.3 Inverted Index

### 2.3.1 Original Meaning

Documents and words in these documents can be expressed as a invert vector file [ 12 ].

“Invert” means a vector file’s rows become columns and its columns become rows. Let us use the following example.

	Word 1	Word 2	...	Word j	....
Doc 1	O11	O12	...	O1j	....
Doc 2	O21	O22	...	O2j	....
.....					
Doc i	Oi1	Oi2	...	Oij	....
.....					

(a) Document-Word hit

	Doc 1	Doc 2	...	Doc i	....
Word 1	O11	O21	...	Oi1	....
Word 2	O12	O22	...	Oi2	....
.....					
Word j	O1j	O2j	...	Oij	....
.....					

(b) Word-Document hit

Figure 2-1 Vector file and its inverted vector file

Doc 1, Doc 2, ... Doc i .... are documents.

Word 1, Word 2, .... Word j .... are words.

$i \in [0, 1, 2, \dots, n]$ ,  $j \in [0, 1, 2, \dots, m]$

We use vector file (a) to express that in document i, there are  $O_{ij}$  hits for word j.

The inverted vector file (b) says word j occurs  $O_{1j}$  times in documents Doc 1,  $O_{2j}$  times in Doc 2, ..... In fact, the inverted file answers the problem of search engine, that is, what documents have the given word?

It is natural that the inverted file can be used as the major structure of the index for a search engine.

### **2.3.2 Application**

To find the occurrences of a given pattern in a text that has not been preprocessed is inefficient. Indexed searching speeds up the search by preprocessing the documents and storing the results in appropriate data structure. This data structure is called index. Inverted files and signature files are two kinds of indexing techniques. A signature file is a filtering mechanism to reduce the amount of data to be indexed. Zobel and Moffat compare these two methods and conclude that inverted file is a better method as it takes less time and space [ 9 ]. The inverted index file contains an entry for every unique document-word pair. Inverted file is a word-oriented mechanism for indexing a text collection. It uses vocabulary and occurrences. Vocabulary consists of the words in the

text, and occurrence is about the word position in the text. If the word appears multiple times in this document, they are different hits.

In inverted files, the index is split into posting part and vocabulary part. The posting part is the file in which the lists of word occurrences and documents are stored contiguously. The vocabulary part is kept in main memory. It is stored in the dictionary order. For each word, there is a pointer to its list in the posting file. Following is an example of an inverted index.

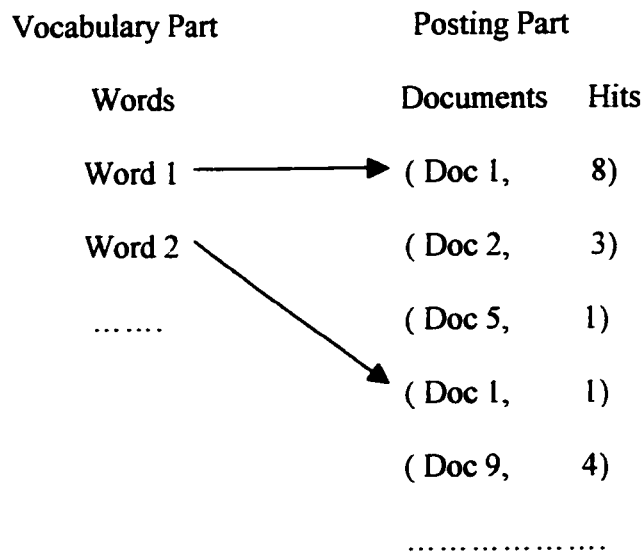


Figure 2-2 An inverted index example

We can implement inverted barrels as binary files. For the inverted file, the word occurrences need much storage space since every occurrence must be recorded. Because stop words are in all text, we do not index such words.

Google is a widely used search engine for querying hypertext data. It uses inverted index to support query efficiently[ 4 ]. Inverted index supports single word and multi-words queries.

## **2.4 HTML Document**

HTML is the language of publishing hypertext on the web. Every text-based web page has its HTML source code (also called HTML document). A repository is used to store HTML documents. We limit the scope in this project to scan HTML documents and extract title, anchor, Meta content, and plain text words from the HTML documents. So, we need to use some of the HTML tags.

## **2.5 Data Structures and Algorithms**

### **2.5.1 Linked List**

Single linked list is used to record the occurrences (hits) of a word. Although there are four kinds of hits (title, anchor, Meta content, and plain text), we use the integrated node structure, all the nodes in the linked list have same structure because this is required for the node in a linked list. Some fields of the node may be empty. Single linked list is pointed by the temporary array's pointer area. We use two operations: add a node to the rear of the linked list, and read every node's data and release the node.



### **2.5.2 Internal Sort**

In the project, we use computer aided generation method to produce the lexicon. That is to say, the lexicon is not generated completely automatically. This is because the lexicon is relative stable. When we generate the lexicon, we choose different words from a vast amount of web pages, then, we use an internal sort algorithm to make these words in dictionary order. We choose quick sort algorithm.

### **2.5.3 String-Based Pattern Matching**

String pattern matching technology is used to extract information between HTML tags. This issue is also known as searching a string in a text. The well-known character comparing based methods are simple shift pattern method and Boyer-Moore method[15]. The Boyer-Moore method is considered as the most efficient string-matching method in usual application. A simplified version is often used in editor program. The algorithm scans the characters of the pattern from right to left, beginning with the rightmost one. It uses two precomputed functions to shift the pattern to the right. These two shift functions are called *good matching shift* and *bad character shift*.

## 2.5.4 Bit Field Structure

In this project, we use bit field structure in order to store data in compact format in both forward and inverted barrels. A barrel is in fact a binary file. Bit field can be accessed individually. We can specify how many bits are used for each field. We can write binary data to a file and read data from a binary file by making use of bit field structure. In the following example, we first define a bit field structure variable, then assign two numbers to the two bit fields.

```
struct {  
    unsigned docid:24; // 24 bits is used to store documentID  
    unsigned hits: 8; // 8 bits to store the number of hits  
} dochits;
```

```
dochits.docid=5; // documentID is 5 (101)
```

```
dochits.hits=9; // the given word occurs 9 (1001) times in the document
```

Using the system call *write()*, binary data could be written to a binary file. The system call *write()* is given in section 2.5.6.

## 2.5.5 File Operation

File operations are essential tools for the project. When parsing HTML documents, we use file; the lexicon is maintained in a file; hit data is stored in files.

C++ supports file input and output through the following classes:

- `ofstream`: File class for writing operations (derived from `ostream`)
- `ifstream`: File class for reading operations (derived from `istream`)
- `fstream`: File class for both reading and writing operations (derived from `iostream`)

### **Open a file**

Open file is represented by a stream object (an instantiation of one of the above classes) and any input or output performed on this stream object will be applied to the physical file.

In order to open a file with a stream object, we use its member function `open ()`:

```
void open (const char * filename, openmode mode);
```

`mode` is a combination of the following flags:

```
ios::in ios::out ios::app ios::binary
```

These flags can be combined using bitwise operator OR: `|`. For example, if we want to open the file "example.dat" in binary mode to add data, we write:

```
ofstream file;
file.open ("example.dat", ios::out | ios::app | ios::binary);
```

We could also declare the previous object and conduct the same opening operation by writing:

```
ofstream file ("example.dat", ios::out | ios::app | ios::binary);
```

You can check if a file has been correctly opened by calling the member function

`is_open()`:

```
bool is_open();
```

It returns true in case the object has been correctly opened or false otherwise.

### **Close a file**

By calling member function: `void close ();`

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes.

We can verify the state of the stream (all of them return a bool value):

`bad()`

Returns true if a failure occurs in a reading or writing operation. For example, in case we try to write to a file that is not open for writing or if there is no space left on the disc where we try to write.

`fail()`

Returns true in the same cases as `bad()` , plus in case that a format error occurs.

`eof()`

Returns true if a file opened for reading has reached the end.

### **Get and put stream pointers**

All I/O streams objects have, at least, one stream pointer:

- `ifstream`, like `istream`, has a pointer known as get pointer that points to the next element to be read.
- `ofstream`, like `ostream`, has a pointer known as put pointer that points to the location where the next element to be written.

- `fstream`, like `iostream`, inherits both: `get` and `put`

These stream pointers that point to the reading or writing locations within a stream can be read and/or manipulated using the following member functions:

`tellg()` and `tellp()`

These two member functions don't have parameters and return a value of type `long int`.

`seekg()` and `seekp()`

This pair of functions serves respectively to change the position of stream pointers. Both functions are overloaded with two different prototypes:

```
seekg ( pos_type position );
```

```
seekp ( pos_type position );
```

Using this prototype, the stream pointer is changed to an absolute position from the beginning of the file. The type required is of the same as the one returned by functions `tellg` and `tellp`.

```
seekg ( off_type offset, seekdir direction );
```

```
seekp ( off_type offset, seekdir direction );
```

The direction could be:

`ios::beg` offset specified from the beginning of the stream

`ios::cur` offset specified from the current position of the stream pointer

`ios::end` offset specified from the end of the stream

## **Binary file**

File streams include two member functions used for input and output data sequentially: write and read. The first one (write) is a member function of ostream, also inherited from ofstream. And read is a member function of istream and it is inherited from ifstream. Objects of class fstream have both. Their prototypes are:

```
write ( char * buffer, streamsize size );
```

```
read ( char * buffer, streamsize size );
```

Here, buffer is the address of a memory block where read data are stored or where the data to be written are taken from. The size parameter is an integer value that specifies the number of characters to be read/written from/to the buffer.

## **Processing Unit**

From current position, we can read a character from the given file, or write a character to the given file. Then, the position moves forward by one character.

We can read a structure from the current position of the given file, or write a structure to the given file from the current position. The position moves forward by the size of the structure.

Read/write a character and read/write a structure can be used together. Doing these, we should remember that the positions moved forward are different.

## **Chapter 3**

# **Implementation of Inverted Index**

### **3.1 Major Data Structures**

Other than the usual data structures, such as stack, queue, string, and internal sort, some special data structures are used in this project. These data structures are in compacted format in order not to take up too much space, as we use compact method to store data in binary files.

#### **3.1.1 Repository**

The repository contains the web pages' source HTML documents. If the web page is not in HTML format, a converter may be applied to obtain HTML code. Converter is not used in this project. In the repository, the HTML documents are stored one after the other. A repository data sample is given in appendix A.

### **3.1.2 Lexicon**

The lexicon may have different forms. For example, lexicon may act as a dictionary or a file's index. Our major lexicon is kept in the main memory. In this way, we can reduce the search time. The lexicon is a list of words and wordIDs. Its function is mapping a word to a number (wordID). We choose sequential structure to save storage space thus to hold more words.

### **3.1.3 Hit Lists**

A hit list corresponds to a list of occurrences for a particular word in a particular document, including word position, font, and capitalization information. Hit lists take up most of the space used in both the forward and inverted indices. It is important to code the hit lists as compactly as possible. We use a program to implement compact encoding. Our aim is to reduce the space for storing these data.

As in Google, our compact encoding uses one machine word for every hit. There are two types of hits: fancy hits and plain hits.

Fancy hits include hits that occur in title, anchor text, and Meta tag. Plain hits include everything else. A plain hit consists of a capitalization bit, 3 bits for the word size, and 20 bits for word position in a document. A fancy hit consists of a capitalization bit, the three font bits are set to 111(distinguish from plain hit), and 20 bits for the position. The other 8 bits is used to identify that this is a hit data.



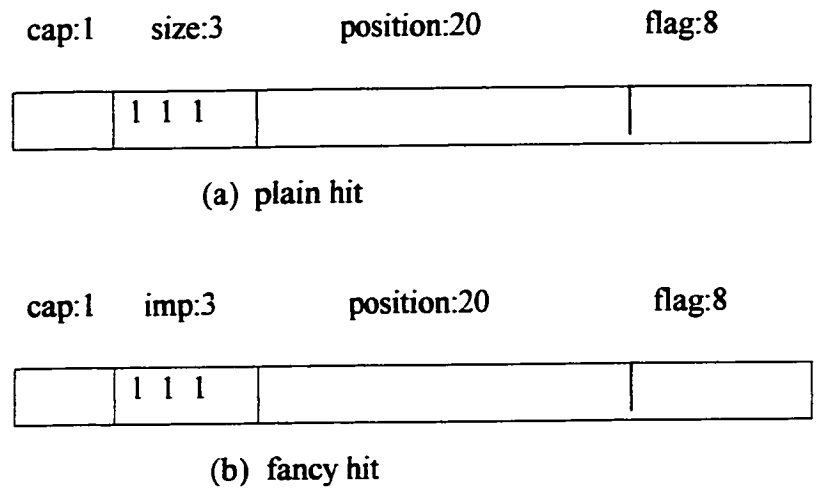


Figure 3-1 Hit structures

### 3.1.4 Forward Index

The forward index is partially sorted. It is stored in a number of barrels. Each barrel holds a range of wordid. If a document contains the word that fall into a particular barrel, the wordID is recorded into the barrel, followed by a list of hit data (capitalization bit, hit type, and word position). The length of a hit list is stored before the hits themselves.

Forward barrel's structure is given below:

<b>docid</b>	<b>wordid:24</b>	<b>hits:8</b>	<b>hit hit hit ...</b>
	<b>wordid:24</b>	<b>hits:8</b>	<b>hit hit hit ....</b>
	<b>null wordid</b>		
<b>docid</b>	<b>wordid:24</b>	<b>hits:8</b>	<b>hit hit hit ...</b>
	<b>wordid:24</b>	<b>hits:8</b>	<b>hit hit hit ...</b>
	<b>wordid:24</b>	<b>hits:8</b>	<b>hit hit hit</b>
	<b>null wordid</b>		
.....			

Figure 3-2 Forward barrel structure

24 bits for wordid.

8 bits for hits, can store  $2^{**8}=256$  hits.

Every hit is coded to 32 bits binary data. Hits data, including the first character's capitalization information (upper or lower case), hit type (the word occurs in title, anchor, Meta content, or plain text), and position of the word in correspond text are stored in forward barrel.

### 3.1.5 Inverted Index

From forward barrel, we obtain the inverted index, which includes inverted barrels and their indices. An inverted barrel is a binary file. Its size is variable based on the HTML documents that are indexed. For every valid wordid, the index contains a pointer which points to the barrel that wordid falls into. It points to a document list together with its hit list. This document list represents all the occurrences of that word in all the documents.

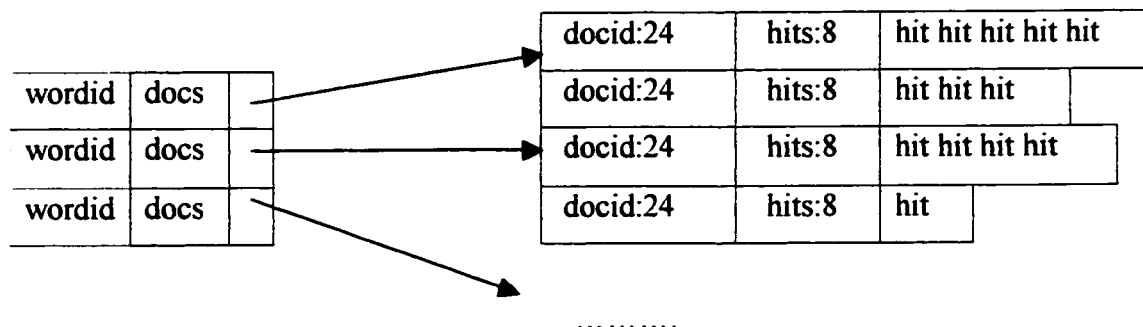


Figure 3-3 Inverted barrel structure

Docs field stores the number of documents that having the given word.  
24 bits for docid and 8 bits for hits. Every hit is coded with 16 bits.

## 3.2 Document Processing

### 3.2.1 Interface

Here, we talk about the interfaces between the indexing subsystem and other subsystems. Repository is the interface between crawling subsystem and indexing subsystem. Crawler downloads web pages' HTML source code from WWW. Crawler downloads HTML source code by making use of protocols, according width-first search algorithm. HTML source code is stored in the repository. Single page's download data include URL, protocol used, content type, connection status, content length, last modified date, server type, download date and time, and HTML source code. The download information is stored together with HTML code for this information can be used in special processing. A repository data sample from Google Inc. [ 6 ] is given in appendix A. The data format in the repository is given below:

```
URL of page 1
..... // Download information
<html>
.
.
.
</html> // end of page 1

URL of page 2
```

```

..... // Download information
<html>
.
.
.
</html> // end of page 2
.
.
.

URL of page n
..... // Download information
<html>
.
.
.
</html> // end of page n

```

Inverted index is the interface between the index subsystem and search subsystem. The indexer takes data from repository and processes these data. The result is stored in the inverted index.

The search subsystem takes data from inverted index and generates the query result.

The overview search engine's internal interfaces is given below:

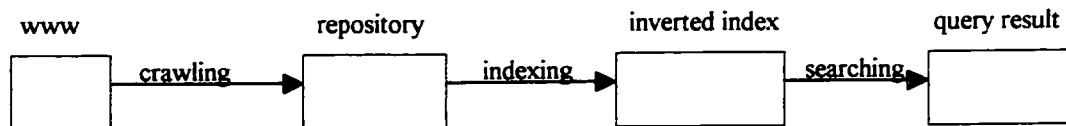


Figure 3-4 Search engine's internal interfaces

### 3.2.2 Extract a Page from Repository

Well formed page's HTML source code begins with `<html>` , or `<HTML>` , and ends with `</html>`, or `</HTML>` . We can obtain a single page's HTML source code by scanning characters between `<html>` and `</html>` , or between `<HTML>` and `</HTML>` . Since the repository is a file, we can extract the first page's source code from repository file's beginning, extract the second page's source code next, ... , until we meet the end of file character.

If a HTML document has no `</html>` tag, we use this page's `</body>` tag. If `</body>` is also missing, we scan until we meet the next page's `<html>` tag .

### 3.2.3 Extract Items from a Page

From single page's HTML source code, we can extract title, anchor text, Meta content, and plain text.

Title pattern is as

```
<title>*****</title>
```

Where `*****` is the title text. We get the characters between `<title>` and `</title>` . The title text may be empty or the title may be missing. There is only one title text (or none) in a page's source code.

Anchor text pattern is as

```
<a href= "URL">*****</a>
```

where \*\*\*\*\* is the anchor text. We obtain an anchor text by accessing the characters between <a href= "URL"> and </a> . Single page's source code may have several anchors, or none. We use loop function to extract anchors between <html> and </html>. In each iteration, we extract one anchor text. If </html> is missing, we can use tag </body>. If no </body> , we can use next document <html> tag, for the next page's anchor is after its <html> tag.

Like anchor text, single page may have zero, one or more Meta content. The Meta content pattern is as follows:

```
<meta name= "_____" content= "*****">
```

\*\*\*\*\* is the Meta content.

Single page may have zero, one or more paragraphs. A paragraph may begin with <p ...> . </p> tag is optional.

The data flow diagram is given below:

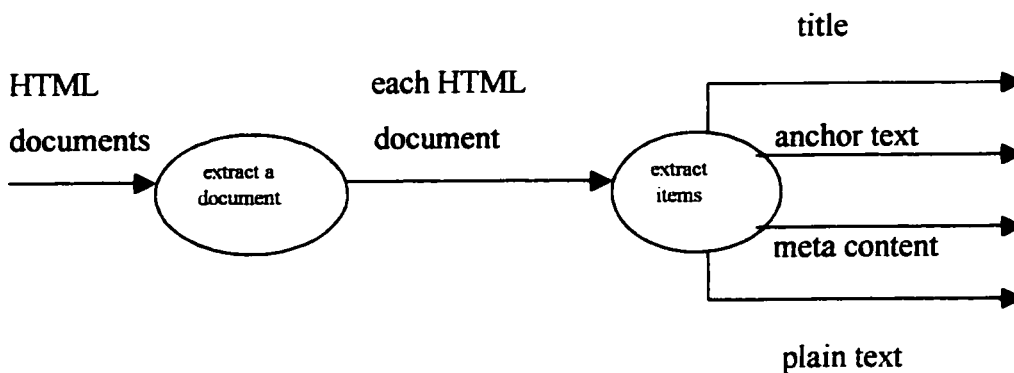


Figure 3-5 Data flow diagram

### 3.2.4 Search for a Given Word

Title, anchor text, Meta content, and plain text are made of words. From section 3.1.2, we know that lexicon is maintained in a database. Before using the lexicon, we load it into the main memory in the form of an array. Every element in this array has two parts: word and wordid. Since an array is a sequential storage structure, and words are in dictionary order, we can do binary search for a given word. If the word is in the array, program returns its index. If the word is not in the lexicon array, 0 is returned, we store it in a log file for future reference.

### 3.2.5 Construct Nodes and Add Them to the Hit Lists

We create a temporary array for each page. The temporary array and linked lists are given in figure 3-7. The element type of the temporary array is a structure. The structure has four fields:

- word : stores word
- wordid : stores word id
- hits : number of occurrence of the word in the document
- pointer: points to the hit list of the word

The linked list stores a word's occurrences in the document. The element type of the linked list is shown in figure 3-6 and its fields are described below.

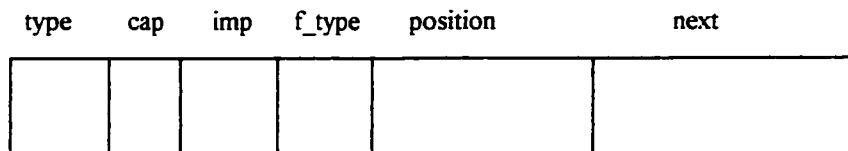


Figure 3-6 Element structure of the hit list

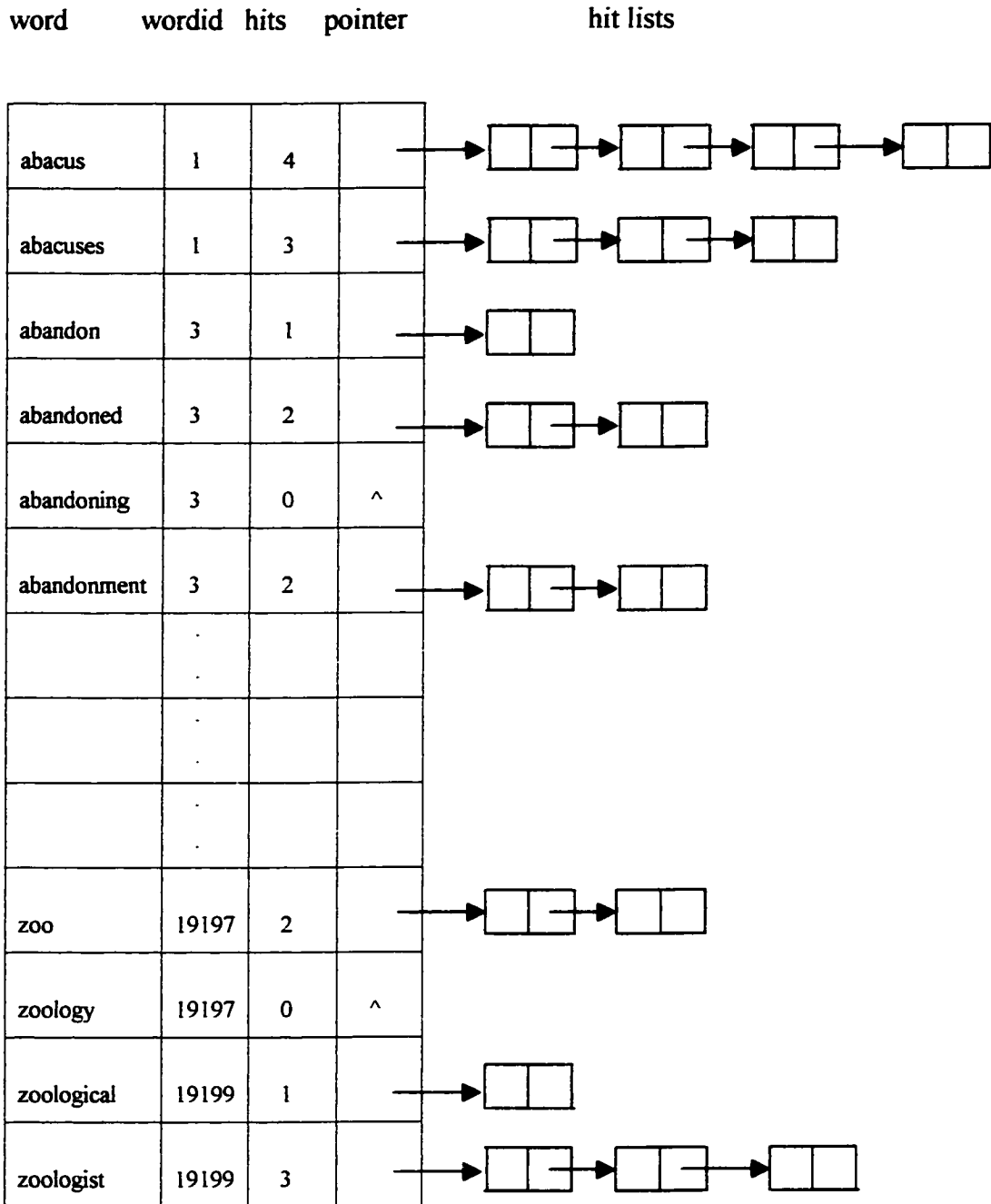


Figure 3-7 Temporary array and linked lists

The six fields used are as follows:

1. type: identify the type of the hit



0: fancy hit

1: plain hit

2. **cap** : the word first character's capitalization information

0: lower case

1: upper case

3. **imp** : for the fancy hit, it stores '111'

for the plain hit, it stores the word size

4. **f\_type**: identify kind of fancy hit

1: title hit

2: anchor hit

3: meta hit

5. **position**: it stores the word position in the document

6. **next**: points to the next element of the linked list, or null

For each word in the document, if the word is in title, anchor text, or Meta content hit(fancy hit), we need to fill fields: type, cap, imp= '111', position, and make up the node.

For the plain hit, fill areas: type, cap, imp, position, and make up the node.

Once the node is constructed, we search the temporary array, if we find the word in the array, a node is added to the hit list and the word's hits field is increased by 1. If we cannot find the word in the array, we do not index this word (it may not be an English word, or is a stop word, we use a log file to store it for later reference), and process the

next word in the document. When we finished processing all the words in the document, we go to the next step outlined below.

### **3.2.6 Constructing the Forward Barrels**

The construction of nodes here is different from the construction node discussed in 3.2.5. In there, we construct one kind of node, which is in the hit lists. Although there are different kinds of hits, the node type in 3.2.5 is the same (as figure 3-6). In this step, we construct four kinds of nodes, each of them is used to record the hits. Four kinds of nodes are used for recording title hit, anchor text hit, Meta content hit, and plain text hit. We should know that, the four kinds of nodes here are built based on the data that are in the nodes of the linked lists of the words..

For the fancy hit, the node here is like the structure in Figure 3-1 (b), and plain hit corresponds to Figure 3-1 (a). In constructing the node, we use a bit field structure. We first define a variable of bit field structure type, and then assign data to the bit fields, we manually compact the hits data, and store the data in the forward barrels.

There are 64 files that are used as forward barrels. Every barrel holds a range of wordid. From the temporary array, index 0 to 300, are put into the forward barrel 1. 301 to 600 are written to forward barrel 2, ... , array index 18901 to 19200(the number of words in the lexicon) are written to forward barrel 64. We may add more words into the lexicon and adjust the number of words in each forward barrel thus not increasing the number of files.

### **3.2.7 Generate Inverted Barrel from Forward Barrel**

For each forward barrel, we need to sort it. We use two-way-two-phase merge sort. In merging, temporary storage space is needed. From one forward barrel, we obtain one inverted barrel, plus one array as its index. Since we use 64 forward barrels, the final result includes 64 files as the inverted barrels and 64 arrays as their indices. The inverted index is the internal interface between the index subsystem and the search subsystem, the search subsystem takes data from the inverted index, based on the data to answer user's query.

### **3.2.8 Summary**

From repository, we extract each HTML document.

Form the document, we extract the title, anchor, Meta content, and plain text.

For each word in the above four kinds of text, we search the word. If the word is not in the lexicon, we register this word in the log file, and process next word. If the word is in the lexicon, we construct its hit node, and add the node to the word's hit list.

When all the words in the document, which are also in the lexicon, are added to the linked lists, documentID, wordID and the amount of hits (in the temporary array), and hits data( in the linked lists) are written to the forward barrels based on wordID.

When all the HTML documents are processed, forward barrels are setup. We sort every forward barrel, and obtain inverted barrel and an array as its index.

Finally, we obtain 64 inverted barrels and 64 arrays, called inverted index. These arrays are in the main memory thus support fast processing for the query subsystem.

### **3.3 Lexicon**

Lexicon (dictionary) is mainly used for mapping a word to a wordid. The operations are:

- Lookup a word
- Insert an item including a word and its id, and reserve the dictionary order

#### **3.3.1 Implementation Strategies**

The lexicon may be implemented as a B+ tree, a hash table, or as an array in the main memory.

B+ tree is efficient in search and insertion. Its disadvantage is the large space is needed to maintain pointers in the main memory. The advantage of using hash table is in insertion while searching. The disadvantages are pointers take up too much memory space and there are collisions during insertion which increases search time.

Our implementation is based on a structured array in the main memory. It can be maintained as a database file. The amount of words, or the capacity of the lexicon is the

size of the array. All the items placed in the lexicon must have the same type. For an array, the type for the items is specified when the array is declared. The lexicon stores word and wordid pairs, ordered by the dictionary order, so that the binary search could be used when searching.

Array-based lexicon has its advantages. First, it support fast search because words are in dictionary order. Second, it uses less storage space because it is a sequential structure. Third, we use this kind of lexicon to deal with stop words (see 3.3.7). Fourth, it can be used in the stemming process (see 3.3.6).

### **3.3.2 Lexicon Data**

In our lexicon, every item has two parts of data: word and wordid. Words in the lexicon are in common dictionary order. See figure 3-8.

You may find that *compute* (ID 10070) and *computer* (ID 10080) are divided into two items in the lexicon, this is because *compute* is a verb, and *computer* is the device for computing, which is a noun.

### **3.3.3 Computer Aided Lexicon Data Generation**

The lexicon data is stable compared to the indexing and querying. The lexicon data is not changed very often. We use computer aided generation method to generate the lexicon data. Words in the lexicon come from web pages. There are three phases:

<b>WORD</b>	<b>WORDID</b>
.....	.....
compress	10010
compressed	10010
compresses	10010
compression	10020
comprise	10030
comprised	10030
comprises	10030
compulsory	10040
compunction	10050
computation	10060
computations	10060
compute	10070
computed	10070
computes	10070
computer	10080
computers	10080
concatenation	10090
concatenations	10090
concede	10100
conceded	10100
concedes	10100
.....	.....

Figure 3-8 Lexicon as a structured array in main memory

You may find that *compute* (ID 10070) and *computer* (ID 10080) are divided into two items in the lexicon, this is because *compute* is a verb, and *computer* is the device for computing, which is a noun.

The first phase is to extract different words from certain amount of web pages. We use about 20 M bytes of data to generate 20,000 different words. The web pages cover business and economy, computer and Internet, news, entertainment, sports, health, military, law, taxes, environment, religion, history, and psychology fields. Following figure shows the relationship between the number of distinct words and the number of HTML documents. For the first 30 documents, samples used affect the words amount tremendously. When the number of documents increases, the increase is smaller.

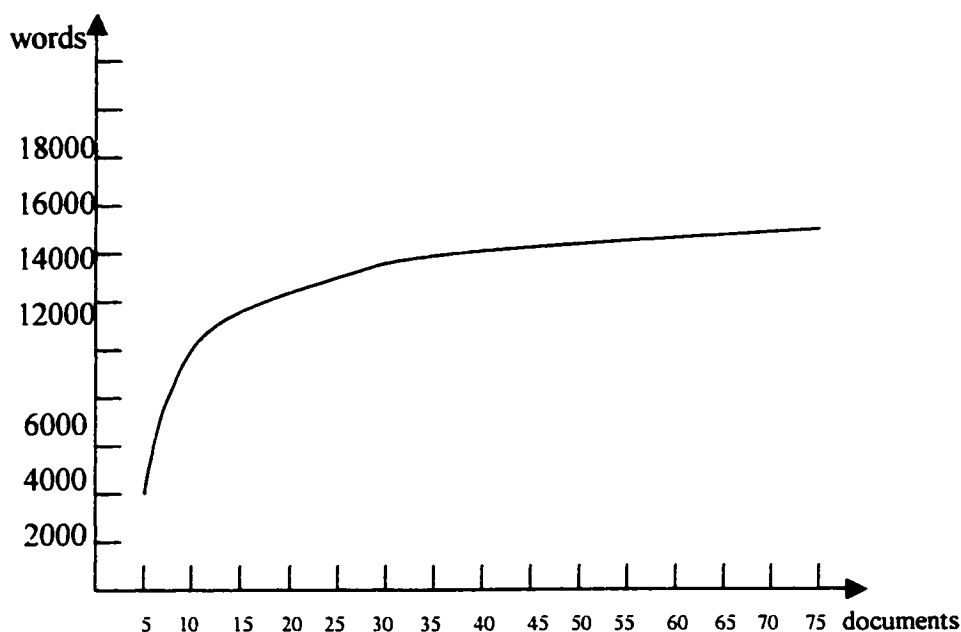


Figure 3-9 Relationship between the amount of words and documents

The algorithm is as follows:

Cal()

```
{save web page in file webpage.dat;
  open webpage.dat;
  read a word from webpage.dat and string copy to an array;
  read a word from webpage.dat and string copy to the tempstring;
  while (!eof())
    { if (tempstring is not in the array)
      search the position to insert and do the string copy to this position;
      read a word from webpage.dat and string copy to tempstring; }
}
```

The second phase is to sort the words in the array. We use quick sort to sort the array.

In the third phase, we write words and their array indices to a structured file, and performing the wordID manually, this is related to the stemming process (see 3.3.6) and eliminating the stop words (see 3.3.7).

### **3.3.4 Use and Maintain the Lexicon**

The lexicon is defined as:

```
typedef struct
{ char word[16];
  int wordid;
```



```
        } worditem;  
  
typedef worditem lexicon[19200];
```

When we want to use the lexicon, we first load it into the main memory as an array. We use a loop program structure. In each iteration, we read a string from the lexicon data file and do the string copy to the `worditem.word` area, read a number from the lexicon data file and assign it to the `worditem.wordid` area. In the main memory, we can use `lexicon[i].word` , `lexicon[i].wordid` to read or write a word and its id.

If we do not use the lexicon, the lexicon is stored as a record file. To do this, we use the loop function. In each iteration, write the contents of `worditem.word` and `worditem.wordid` to the lexicon file. Figure 3-10 shows the data sample stored in the file `lexicon.dat` (corresponding to the data in figure 3-8).

### **3.3.5 Trimming of Punctuation**

In a text, words are separated by punctuation or blanks. A word is a string of alphabetic characters. We should remove the trailing and leading punctuation before searching in the lexicon. The words are converted to lowercase before they are indexed.

### **3.3.6 About Stemming Process**

Stemming is a process to find out a word's stem or root. Some stemming algorithms, Okapi [ 22 ] for instance, uses both weak stemming to remove plural endings and grammatical suffixes like *ing* and *ed* and strong stemming to remove derivational

.....	.....
10010	compress
10010	compressed
10010	compresses
10020	compression
10030	comprise
10030	comprised
10030	comprises
10040	compulsory
10050	compunction
10060	computation
10060	computations
10070	compute
10070	computed
10070	computes
10080	computer
10080	computers
10090	concatenation
10090	concatenations
10100	concede
10100	conceded
10100	concedes
.....	.....

Figure 3-10 Lexicon as a record file in disk

suffixes like *ent*, *ence*, and *sion*. Other stemming algorithms use truncation to find the root. For example, Qpatus [ 3 ] uses sophisticated linguistic rules.

We think that using sophisticated linguistic rules inevitably causes problems. First, it is difficult to gather all linguistic rules. Second, error is inevitable. According to their linguistic rules, we can find an error example easily. Third, processing too many rules takes too much CPU time.

Stemming problem is avoided by using our method, that is to say, we need not solve stemming problem specially. Rather than using an stemming algorithm to generate the stem, we present a method, called lookup dictionary method. Let us go back to figure 3-10, the stem word and its grammatical variations have the same wordid. For example, verbs “compute”, “computed”, and “computes”, correspond to wordID 10070. Nouns “concatenation” and “concatenations” correspond to the wordID 10090. Table-lookup method allows quick indexing over the derivations of the stem error freely.

### **3.3.7 Stop Words**

Stop words are words that are commonly used in sentences, but they do not carry any features of the text. Stop words include “and”, “of”, “the”, etc. Stop words shouldn't be indexed because their occurrences are very frequent and their hits would take up too

much space. In our implementation, stop words are not in the lexicon, so they are not indexed.

Roughly speaking, stop words take up 20% of all the text words. Although the stop words have no inference upon the crawling subsystem for the search engine, for the index subsystem, they will take up more storage space.

## **3.4 Parsing HTML Document**

### **3.4.1 Extract Single Document**

Repository is a text file that stores HTML documents. From repository, we extract single HTML document one by one. We use following program segment:

```
ifstream fromr("repository.dat",ios::in);
fromr>>ch;
for(i=0;i<100;i++)    // if we store 100 pages in the repository
{docid++;
  ofstream totp("temppage.htm",ios::out);
  while(strncmp(ch,"</html>",6))
  {
    fromr>>ch; totp<<ch<<' ';
  };
}
totp.close();
fromr.close();
```

### 3.4.2 Extract Title

From single HTML document, we extract the title of the page. Each HTML document has a title. The title text may be empty. We use following program segment:

```
char ct;
char cht[7];
ifstream fromtp("temppage.htm");    // single html document
ofstream tott("temptitle.htm");     // file to store title text
cht[0]=fromtp.get();
cht[1]=fromtp.get();
cht[2]=fromtp.get();
cht[3]=fromtp.get();
cht[4]=fromtp.get();
cht[5]=fromtp.get();
cht[6]=fromtp.get();
while(!(cht[0]=='<' && cht[1]=='t' && cht[2]=='i'&&cht[3]=='t'&&
      cht[4]=='l' && cht[5]=='e' && cht[6]=='>')) // compare
    {
        // shift to right by one position
        cht[0]=cht[1];
        cht[1]=cht[2];
        cht[2]=cht[3];
        cht[3]=cht[4];
        cht[4]=cht[5];
        cht[5]=cht[6];
        cht[6]=fromtp.get();
    };
ct=fromtp.get();
while(ct!='<')
    {
        // extract title text
        tott.put(ct);
    }
```

```

        ct=fromtp.get(); // write to a file
    };
tott.close();
fromtp.close();

```

The processing unit here is single character.

### 3.4.3 Extract Anchor Text, Meta Content

The principal of extracting anchor and Meta content is the same. We take the anchor text for example. Similar to 3.4.3, we can extract an anchor text. We put the extraction of single anchor in a loop, loop starts from this page's beginning, ends in the page's end, then, we can extract all the anchors of the page. The program segment is as follows:

```

ifstream fromtp("temppage.htm"); // extract from a document
ofstream tota("tempanchor.htm"); // file to store anchor text
char ca, cha[7];
cha[0]=fromtp.get();
cha[1]=fromtp.get();
cha[2]=fromtp.get();
cha[3]=fromtp.get();
cha[4]=fromtp.get();
cha[5]=fromtp.get();
while(!(cha[0]=='<' && cha[1]=='/' && cha[2]=='h' && cha[3]=='t' &&
        cha[4]=='m' && cha[5]=='l'))
    { while(!(cha[0]==' ' && cha[1]=='h' && cha[2]=='r' &&
            cha[3]=='e' && cha[4]=='f' && cha[5]=='='))
        { // compair
          cha[0]=cha[1];

```

```

cha[1]=cha[2];
cha[2]=cha[3];
cha[3]=cha[4];
cha[4]=cha[5];
if(!fromtp.eof())
    cha[5]=fromtp.get(); // read a character
    else return(0);
};

// if not meet the end of file, shift to the right by one character
if(cha[0]=='&&
cha[1]=='h'&&
cha[2]=='r'&&
cha[3]=='e'&&
cha[4]=='f'&&
cha[5]=='')
{ ca=fromtp.get(); // read a character
while(ca!='<')
{ tota.put(ca);
ca=fromtp.get(); }; // store them in a file
tota.put('\n');
};
cha[0]=fromtp.get();
cha[1]=fromtp.get();
cha[2]=fromtp.get();
cha[3]=fromtp.get();
cha[4]=fromtp.get();
cha[5]=fromtp.get(); // read a character
};
tota.close();
fromtp.close();

```

Here, the processing unit is a single character, but in the further processing, we choose single word (a character string) as the processing unit.

*Filtering* Sometimes, we can not extract the exact term in one time. In these cases, we need to filter the extracted data one or more times. The algorithm is given below:

```
read a character from anchor text;
while(! end of anchor text)
    {if it is the noise character
        then push it to the noise stack;
        read a character from anchor text; }
clear the noise stack;
```

Here is the example of filtering. The HTML source code is from URL:

<http://www.cit.buffalo.edu/search.html>. One anchor is:

```
<a href= "http://www.cit.buffalo.edu/about.html"> About CIT</a>
```

After the first round of filtering, we get:

```
http://www.cit.buffalo.edu/about.html"> About CIT</a>
```

After the second round of filtering, we get:

```
About CIT
```

We obtain the anchor text.



### 3.4.4 Extract Plain Text

Plain text includes the words that are not in title, anchor text, or Meta tag. We extract plain text from the body of a HTML document. These words are between HTML tags (if any).

Following are some possible patterns for plain text.

`<li> ***** </li>`

`<dt> ***** </dt>`

`<p ...> ***** <br>`

`***** <p>`

`<h1> ***** </h1>`

`<h2> ***** </h2>`

Where \*\*\*\*\* is the plain text.

We cannot get plain text simply from pattern: `<p> ***** </p>` for two reasons:

1. `</p>` is optional, a paragraph may have a start tag `<p>`, but no end tag `</p>`.
2. Plain text may be in other patterns.

## **3.5 Sorting**

### **3.5.1 Build Hit Lists**

We have seen a method for building hit lists for single document from the web. They do the sequential search for all the words in the document. Then they sort the hit lists' index. In our implementation, we only do the binary search for all the words in the document. We believe that our method is better than the above mentioned method. For the same number of words, we simply do fast search instead of sequential search plus a sorting. In our implementation, the sorting is implied in the processing. We change a sequential searching plus a sorting problem to a fast searching problem by making use of the lexicon.

### **3.5.2 Merge Sort Forward Barrels**

Let us examine a forward barrel example in figure 3-11.

We use two-way-two-phase merge to sort each forward barrel, and get an inverted barrel and an array as the index.

In the first phase, we merge section Doc001 and section Doc002 (in figure 3-11), The merge is based on the wordID in these two sections. Suppose we use 0 stands for the nullword, variable *term1* stands for a wordID in section Doc001, term2 for a wordID in section Doc002, then we have 5 conditions:

Doc 001	Word 001	61	...
	Word 002	50	....
	Word 005	4	....
	Word 008	2	....
	Word 010	15	....
	Word 517	51	....
	Word 980	41	....
	Null word		
Doc 002	Word 001	31	....
	Word 002	1	....
	Word 003	4	....
	Word 008	5	....
	Word 057	11	....
	Word 110	3	....
	Null word		
Doc 004	Word 001	5	....
	Word 002	2	....
	Word 003	1	....
	Word 007	4	....
	Word 110	10	....
	Null word		

Figure 3-11 Forward barrel example

1. term1=0 and term2=0
2. term1=0 and term2>term1
3. term2=0 and term1>term2
4. term1!=0 and term2!=0 and term1<term2
5. term1!=0 and term2!=0 and term2<term1

Merging section Doc001 and section Doc002 (in figure 3-11), we obtain following array and file as in figure 3-12

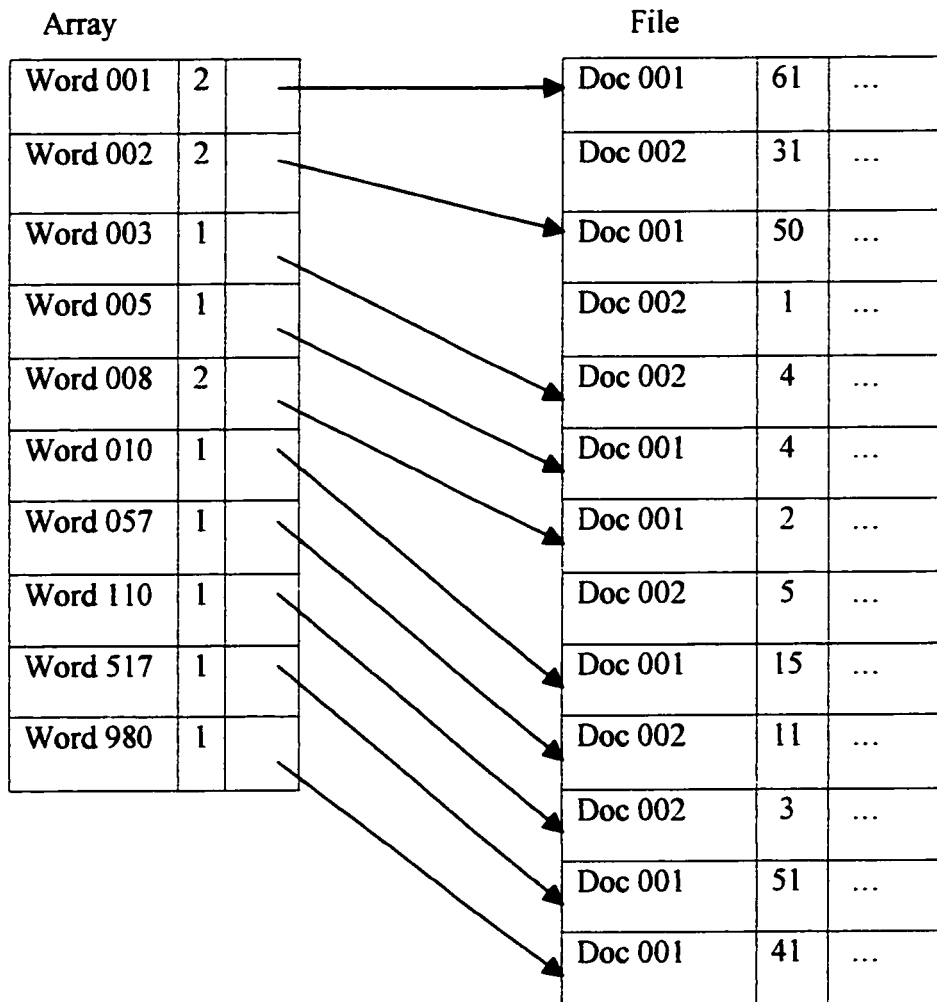


Figure 3-12 Index array and inverted barrel after the first step of merge

In the second phase, we merge structures in figure 3-12 and section Doc004 in figure 3-11, the merge is based on the wordID in the array in figure 3-12 and wordID in section Doc004 in figure 3-11. we obtain following array and file as in figure 3-13.

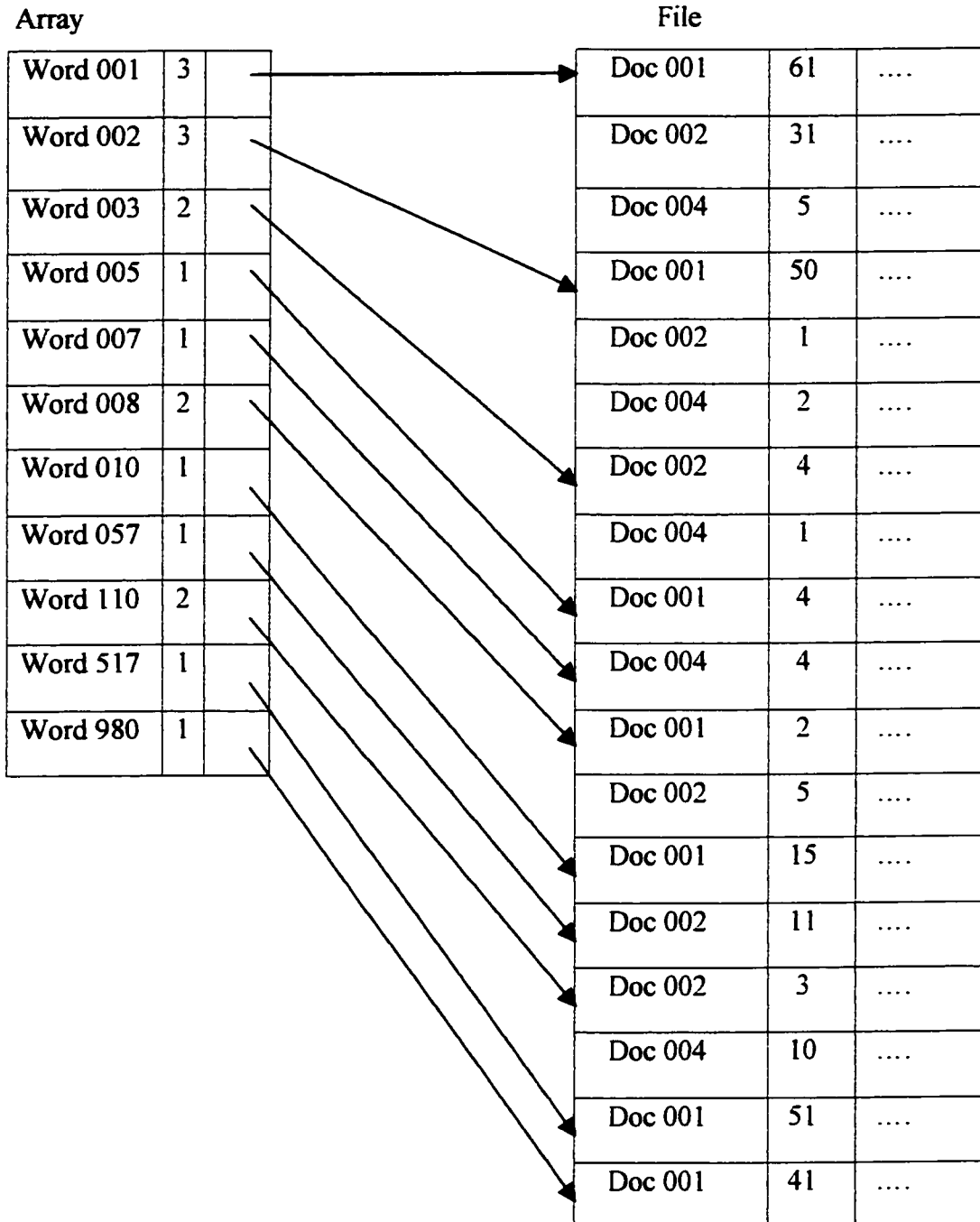


Figure 3-13 Index array and inverted barrel after the second step of merge

If there are more sections below the section Doc004, then the second phase needs reuse.

This two-way-two-phase merge program is the core part for the inverted index generator project.

### **3.6 Main Function**

The main function connects all the subroutines of the project. It calls subroutines directly or indirectly. We assume that all the HTML documents in the repository are error free. The final results are 64 files as inverted barrels and 64 in memory arrays as their indices. We maintain these arrays as structured files.

In the main function, we first load the lexicon from disk. In main memory, the lexicon is an array. We open the repository file for reading. We need to initialize the temporary array. Now, parsing begins. We obtain the title text, anchor text, meta content, and plain text. We use the words to build temporary array and linked lists. Then according to the wordid, we distribute the hits to 64 forward barrels. The last step is merge sort forward barrels until the inverted index is set up.

We give the algorithm of the main function.

```
main()
{
load lexicon[19200];
open("repository.dat");
docid=0;
```

```

while(block!=EOF)                                // repository not finish
{
while(block!= "<HTML>"|| "<html>") // proceed to the beginning of the page
    read a block;
docid++;
initialize temporary array;
read a block;
while(block!= "</HTML>"|| "</html>")
    {extract terms; read a block;};
search words and add hits to the linked lists;
array temp and linked lists=>forward barrels;
    read a block };
close("repository.dat");
gib(); } // forward barrels are change to inverted index

```

### 3.7 Test

During the implementation process of this project, we have tested the following items:

1. Extracted terms test
  - a. Whether the title is correct.
  - b. Whether the anchor is correct
  - c. Whether the Meta content is correct
  - d. Whether the plain text is correct
2. Filter result test
3. Verify the content in each forward barrel
4. Verify the contents both in each inverted barrel and its index
5. Project final test

- a. Whether the indexed word is in the document
- b. Whether the word in the document is indexed
- c. Whether each document in the repository is indexed
- d. Whether the indexed document is in the repository
- e. Each forward barrel increases as the number of documents increases
- f. Each inverted barrel and its index increase as the number of documents increases

For example, when test 5 (a), we indexed a word that ID is 1215, which is in the first document in the repository file. In the lexicon file, correspond to 1215 is the word “allow”, when we open repository file, “allow” is there.



# **Chapter 4**

## **Conclusion**

### **4.1 Contributions for the Project**

Following is our contributions for the CINDI project:

- The search engine's internal interface research
- Extraction of terms from HTML documents
- Design, implementation, and maintenance of the lexicon
- Present lookup lexicon stemming method
- Forward and inverted barrel research
- Prototype implementation

### **4.2 Conclusion**

CINDI is designed to be a scalable search engine. A large-scale web search engine is a complex system and much remains to be done. By taking part in this project, we find that

inverted index supports single word and multi-words queries. If we plan to answer exact phrase query, we need to store the whole text together with the inverted index. If we put words' synonyms together with the words themselves in the lexicon, then we can answer the synonym query. For example, if query is "programmer", the search engine could answer "programmer", "software developer", and "software engineer".

It is necessary to develop a PageRank subroutine to prioritize the search results. In this way, search subsystem may use two kinds of index results: semantic header-based index results and inverted index-based index results. We may prioritize by using the weights of the different factors, and provide user with the integrated search results.

In implementing inverted index generator, we have experienced complexity program structures. New and advanced technologies are needed for this project. So, we believe that CINDI will be a resource for searchers and researchers.

## References

- 1 Bipin C. Desai, Semantic Header aka Cover Page, Department of Computer Science, Concordia University  
<http://www.cs.concordia.ca/~faculty/bcdesai/web-publ/semantic-header.html>
- 2 Bipin C. Desai, The Semantic Header and Indexing and Searching on the Internet Department of Computer Science, Concordia University  
<http://www.cs.concordia.ca/~faculty/bcdesai/web-publ/cindi-system-1.0.html>
- 3 Sami Haddad, Automatic Semantic Header Generator, Department of Computer Science, Concordia University. September 1998
- 4 Brin, S. and Page, L. The anatomy of a large-scale hypertextual search engine. In *Proceedings of the 7th International World Wide Web Conference* (Brisbane, Australia, April 14-18), 107-117. 1998
- 5 Google Search Engine, <http://www.google.com>
- 6 First Annual Google Programming Contest, Google Inc. April 2002  
<http://www.google.com/programming-contest/>
- 7 Gang Cheng, Search Engine Subsystem, December 1996  
<http://www.npac.syr.edu/users/gcheng/homepage/thesis/node103.html>
- 8 Lycos Search Engine, <http://www.netscape.com>
- 9 Zobel, J., Moffat, A. and Rao, K. R., Inverted Files Versus Signature Files for Text Indexing, *ACM Transactions and Database Systems*, Vol. 23, 453-490, 1998.
- 10 Yahoo! <http://www.yahoo.com/>
- 11 Excite Search Engine, <http://www.excite.com/>
- 12 Marti Hearst and Ray Larson, Implementation Issues Information Seeking Behavior, Fall 1998.  
<http://www.sims.berkeley.edu/courses/is202/f98/Lecture19/tsld005.htm>
- 13 Nivio Ziviani, Edleno Silva de Moura, Gonzalo Navarro, Ricardo Baeza-Yates, *Compression: A Key for Next-Generation Text Retrieval Systems*, *IEEE Computer*, 33(11): 37-44, November 2000
- 14 Inverted Index Algorithm and Compression,  
<http://ir.iit.edu/~dagr/cs529/files/handouts/02aCompression-6per.PDF>

- 15 Donald E. Knuth, The Art of Computer Programming, Volumn 1, Fundamental Algorithms, Third Edition, Addison-Wesley, ISBN 0-201-89683-4. Volumn 3, Sorting and Searching, Second Edition, Addison-Wesley, ISBN 0-201-89585-0.
- 16 Markus Kuhn, Table-based/ inverted index text search algorithms, <http://mail.nl.linux.org/linux-utf8/2002-04/msg00117.html>
- 17 Ajith N., Jyothir G. R., Abhishek S., An Inverted Index Implementation Supporting Efficient Querying and Incremental Indexing, May 2002. <http://www.cs.wisc.edu/~jyothir/784/report.pdf>
- 18 Scott Weber and Sam Vann, Lehigh Search Engine, Indexer Implementation Document, October 2002. [http://www.lehigh.edu/~sdw3/csc397/indexer/implementation\\_doc.html](http://www.lehigh.edu/~sdw3/csc397/indexer/implementation_doc.html)
- 19 Christof Monz and Maarten de Rijke, Inverted Index Construction, Spring 2002 <http://remote.science.uva.nl/~christof/courses/ir/transparencies/clean-w-05.pdf>
- 20 Nazli Goharian, Introduction to Information Retrival Systems, course notes, February, 2002. <http://www.csam.iit.edu/~abet/CourseOverviews/cs495.doc>
- 21 Jeffrey D. Ullman, Data Mining Lecture Notes, Searching the Web, <http://www-db.stanford.edu/~ullman/mining/mining.html>
- 22 S. E. Robertson, Okapi at T1, <http://trec.nist.gov/pubs/trec2/papers/txt/02.txt>
- 23 JeanTreblay and Grant Cheston, Data Structure and Software Development, Prentice-Hall Inc. 2001
- 24 Uresh U., UNIX Internals: The New Frontiers, Prentice Hall, 1997. ISBN 0-13-101902-2
- 25 The American Heritage College Dictionary, Third Edition, Houghton Mifflin Co. 1993.
- 26 Cathleen Moore, Top Ten Technology Innovators: Larry Page and Sergey Brin <http://archive.infoworld.com/articles/fe/xml/02/03/04/020304fegoogle.xml>

## Appendix A    Repository Data Sample

---






URL: <http://www.yale.edu/yser/>

---

HTTP/1.1 200 OK  
X-Google-Crawl-Date: Thu, 13 Sep 2001 21:24:12 GMT  
Connection: close  
Content-length: 7655  
Last-modified: Thu, 30 Aug 2001 22:56:11 GMT  
Content-type: text/html  
Date: Thu, 13 Sep 2001 21:24:12 GMT  
Server: Netscape-Enterprise/3.6 SP3

```
<html>
<head>
<title>YSER Home Page</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-
  1">
<meta name="keywords" content="Yale, undergraduate research
  opportunities, science and engineering research, undergraduate
  research programs, undergraduate research fellowships ">
</head>

<body bgcolor="#FFFFFF" text="#000000">
<table width="100%" border="0">
  <tr>
    <td width="24">&nbsp;</td>
    <td>
<div align="center">
<table width="667" border="0" cellspacing="0" cellpadding="0">
  <tr>
    <td width="148"></td>
    <td width="10">&nbsp;</td>
    <td width="509"></td>
  </tr>
</table>
</div>
<table width="600" border="0" height="300" align="center">
  <tr>
    <td></td>
```

&nbsp;

**Original research is an integral part of undergraduate science education at Yale. Academic year and summer research opportunities bring classroom concepts to life and prepare Yale students for postgraduate training and careers in the sciences and engineering. With access to more than 800 faculty laboratories in 43 degree-granting programs in the Faculty of Arts and Sciences, Yale School of Medicine and Yale School of Forestry and Environmental Studies, Yale undergraduates perform research ranging from the observation of black holes to the development of the nervous system.**

**The YSER (Yale Science and Engineering Research) Program provides access to research opportunities via the YSER web site, [YSER workshops](workshops.html), [YSER News and links](news.html) and YSER-sponsored programs for the support of undergraduate research.**

&nbsp;

**Research Opportunities at Yale**

**Yale provides a variety of programs tailored to the individual needs of students, ranging from interdepartmental programs such as STARS and Perspectives on Science to departmentally-based programs and other opportunities to perform research in the laboratories of faculty throughout Yale University.**

**[Perspectives on Science](perspectives.html) is a yearlong course and summer program providing first year students with an introduction to contemporary scientific research and research opportunities.**

**[STARS](stars.html) (Science, Technology and Research Scholars) provides selected first year students through seniors with an integrated experience in research, course-based study and development of mentorship**

skills.</font></p>

<p><font face="Verdana, Arial, Helvetica, sans-serif">  
 <a href="facdept.html">Departmentally-based research programs</a>  
 provide a wide variety of research opportunities to students  
 throughout their time at Yale. While many students perform  
 research in conjunction with departmental major requirements, such  
 research may alternatively be carried out in laboratories of  
 faculty in other appropriate departments throughout Yale  
 University.</font></p>

<p>&nbsp;</p>

<p><font face="Verdana, Arial, Helvetica, sans-serif"><b><font  
 size="+1">Support for Undergraduate Research at  
 Yale</font></b></font></p>

<p><font face="Verdana, Arial, Helvetica, sans-serif">  
 Interdepartmental programs such as  
 <a href="perspectives.html">Perspectives on Science</a>  
 and <a href="stars.html">STARS</a> provide funding for research  
 during the summer and/or academic year.</font></p>

<p><font face="Verdana, Arial, Helvetica, sans-serif">More than fifteen  
 <a href="fellowships.html">individual fellowship programs</a> are  
 available for the support of undergraduate research.</font></p>

<p><font face="Verdana, Arial, Helvetica, sans-serif"> Many student  
 research projects are supported by <a  
 href="facdept.html#facgrants">individual faculty research  
 grants.</a></font></p>

<p>&nbsp;</p>

<p><font face="Verdana, Arial, Helvetica, sans-serif"><b><font  
 size="+1">Frequently Asked Questions</font></b></font></p>

<p><font face="Verdana, Arial, Helvetica, sans-serif"><a  
 href="faq.html#when">When can I start research?</a></font></p>

<p><font face="Verdana, Arial, Helvetica, sans-serif"><a  
 href="faq.html#term">Should I do research in the academic year or  
 the summer?</a></font></p>

<p><font face="Verdana, Arial, Helvetica, sans-serif"> <a  
 href="faq.html#fund">How will my research be funded? Can I receive  
 stipend support?</a></font></p>

<p><font face="Verdana, Arial, Helvetica, sans-serif"><a  
 href="faq.html#find">How do I find a research lab?</a></font></p>

<p><font face="Verdana, Arial, Helvetica, sans-serif"><a  
 href="faq.html#what">How will my research project be  
 defined?</a></font></p>

<p>&nbsp;</p>

<div align="left"></div>

<table width="616" border="0" cellspacing="0" cellpadding="0">

<tr>

<td rowspan="2" width="84" valign="bottom"><a

```

href="index.html"></a></td>
<td height="25" valign="bottom">
<div align="right">
<p align="center">&nbsp;</p>
</div>
</td>
</tr>
<tr>
<td height="50" valign="top" align="right"><a href="index.html"></a><a
href="news.html"></a><a href="facdept.html"></a><a
href="fellowships.html"></a><a href="faq.html"></a></td>
</tr>
</table>
<table width="616" border="0" cellspacing="0" cellpadding="0">
<tr>
<td width="84">&nbsp;</td>
<td align="right">
<div align="center"><font face="Verdana, Arial, Helvetica, sans-serif"><a
href="mailto:yser@yale.edu"><font size="-1">yser@yale.edu</font></a>
<font size="-1"><i> &#149; Last updated</i>: 30 August, 2001
&#149;&nbsp;&nbsp;&nbsp;Copyright
Yale University 2001</font></font></div>
</td>
</tr>
</table>
<table>
<center>
<center>
</center>
</center>
<p>
</td>
</tr>
<tr>
<td width="24">&nbsp;</td>
<td>&nbsp;</td>
</tr>
</table>
</body>
</html>

```



-----  
URL: <http://www.crk.umn.edu/newsevents/notices00-01/grad2001/>  
-----

HTTP/1.0 200 OK  
Content-length: 2506  
Last-modified: Mon, 21 May 2001 21:07:48 GMT  
Content-type: text/html  
Accept-ranges: bytes  
Server: WebSitePro/2.3.10  
Date: Mon, 06 Aug 2001 08:19:32 GMT

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">  
<meta name="GENERATOR" content="Microsoft FrontPage 4.0">  
<meta name="ProgId" content="FrontPage.Editor.Document">  
<title>Commencement 2001</title>  
</head>

<body>

<p align="center"><a href="../../index.htm"><font face="Arial"></font></a></p>  
<p align="center"><font face="Arial" size="4"> Commencement 2001<br>Saturday, May 5, 2001<br><br></font><font face="Arial" size="2"><span style="mso-bidi-font-size:10.0pt;mso-bidi-font-family: Arial">One of the largest groups ever—148 graduates—took part in commencement ceremonies, held May 5.<span style="mso-spacerun: yes">&nbsp;&nbsp;&nbsp;</span><br>The event was also notable because of the <a href="../../ROTC-Stanley.htm"> first ROTC commissioning ceremony</a> in UMC history.<span style="mso-spacerun: yes">&nbsp;&nbsp;&nbsp;</span><br>Colonel Clark P. Wigley, Commander of Air Force ROTC Detachment 610,&nbsp;&nbsp;&nbsp;<br>swore in cadet Lisa Marie Stanley as a Second Lieutenant in the U.S. Air Force.<span style="mso-spacerun: yes">&nbsp;&nbsp;&nbsp;</span><br>Graduating Crookston Student Association President Shane Roers was the student speaker.<span style="mso-spacerun: yes">&nbsp;&nbsp;&nbsp;</span><br>Regent Neel was guest speaker.<br></span><br></font><font face="Arial" size="4">Follow the links below for photos from



## **Appendix B    Lexicon Data Sample**

1 abacus

1 abacuses

2 abandon

2 abandoned

2 abandoning

2 abandonment

3 abase

3 abasement

4 abash

4 abashed

5 abate

5 abated

6 abatement

7 abattoir

8 abbess

9 abbey

9 abbeys

10 abbot

11 abbreviate

11 abbreviated

11 abbreviation

12 abdicate

12 abdicated

12 abdication  
13 abdomen  
14 abdominal  
15 abduct  
16 abduction  
17 abeam  
18 aberrant  
18 aberration  
19 abhor  
19 abhorrence  
19 abhorrent  
19 abhorred  
20 abide  
20 abided  
20 abiding  
21 abilities  
21 ability  
22 abject  
22 abjection  
22 abjectly  
23 abjuration  
23 abjure  
23 abjured

24 ablative  
25 ablaut  
26 ablaze  
27 able  
28 ablutions  
27 ably  
29 abnegation  
30 abnormal  
30 abnormalities  
30 abnormality  
30 abnormally  
31 aboard  
32 abode  
33 abolish  
33 abolished  
33 abolitionism  
34 abolitionist  
34 abolitionists  
35 abominable  
36 abominably  
37 abominate  
37 abominations

## **Appendix C    Source Code and Data File List**

**Makefile:** project makefile

**ligc.cpp:** project main function

**Agpl.cpp:** extract title from a HTML document

**Agta.cpp:** extract anchor text from a HTML document

**Agtm.cpp:** extract Meta content from a HTML document

**Agtp.cpp:** extract each HTML document from repository

**Agtt.cpp:** extract title from a HTML document

**Bs.cpp:** binary search

**Cal.cpp:** computer aided lexicon data generation

**Filter.cpp:** refine the extracted terms

**Initarr.cpp:** initialize the temporary array

**Insertsort.cpp:** insertion sort

**Linkedlist.cpp:** linked list operations

**Loadlexi.cpp:** load lexicon

**Rw.cpp:** read a character string from file

**Trh.cpp:** trimming of punctuation

**Twm.cpp:** two way merge to generate inverted index

**Data files**

**Repository.dat:** repository data file

**Lexicon.dat:** store lexicon data

**Fb1.dat , Fb2.dat , ... , Fb64.dat:** forward barrels

**Ib1.dat, Ib2.dat , ... , Ib64.dat:** inverted barrels

## Appendix D Forward Barrel Setup Method

We use an example to explain how to create a forward barrel.

The forward barrel data will be

DocumentID	WordID	Hits	Hitdata
9	3	2	1 111 0000 0000 0001, 0 100 0000 0000 0110
	10	2	0 011 0000 0000 1111, 0 010 0000 0001 0000 nullword(0000 0000 0000 0000 0000 0000 0000 0000)
15	11	4	1 111 0000 0000 1000, 0 000 0000 0000 0011
			0 010 0000 0000 0010, 0 000 0000 0001 1111 nullword(0000 0000 0000 0000 0000 0000 0000 0000)
0			

we use following code to write the data to the forward barrel fb.dat( binary file)

```
#include <stdio.h>
#include <iostream.h>

void main(void)
{
    struct                                // store wordid+hits
    {
        unsigned wordid:24;
        unsigned hits:8;
    }wh;

    struct                                // store one hit
    {
        unsigned cap1:1; // first word first letter's capitalization, 0 lowercase
                           //1 uppercase
        unsigned imp1:3; // word size or fancy type
        unsigned position1:12; // word position in the text
        unsigned cap2:1; // next word
        unsigned imp2:3;
        unsigned position2:12;
    }oh;

    struct                                // store docid
    {
        unsigned docid:24;
        unsigned :8;
    }doc;

    fstream fb1fpl("fb1.dat", ios::out|ios::in|ios::binary);
```

```

doc.docid=9;
fb l fp l .write((char *)&doc,4); // docid 9 =>fb.dat

wh.wordid=3; // wordid 3+ hits 2=>fb.dat
wh.hits=2;
fb l fp l .write((char *)&wh,4);

oh.cap1=1; // two hits =>fb.dat
oh.impl=7;
oh.position1=1;
oh.cap2=0;
oh.impl2=4;
oh.position2=6;
fb l fp l .write((char *)&oh,4);

wh.wordid=10; // wordid 10+ hits 2=>fb.dat
wh.hits=2;
fb l fp l .write((char *)&wh,4);

oh.cap1=0;
oh.impl=3;
oh.position1=15;
oh.cap2=0;
oh.impl2=2;
oh.position2=16;
fb l fp l .write((char *)&oh,4);

wh.wordid=0; // nullword=>fb.dat
wh.hits=0;
fb l fp l .write((char *)&wh,4);

doc.docid=15;
fb l fp l .write((char *)&doc,4); // docid 15 =>fb.dat

wh.wordid=11; // wordid 11+ hits 4=>fb.dat
wh.hits=4;
fb l fp l .write((char *)&wh,4);

oh.cap1=1; // two hits=>fb.dat
oh.impl=7;
oh.position1=8;

```



```
oh.cap2=0;
oh.imp2=0;
oh.position2=3;
fb1fp1.write((char *)&oh,4);

oh.cap1=0;
oh.imp1=2;
oh.position1=2;
oh.cap2=0;
oh.imp2=0;
oh.position2=31;
fb1fp1.write((char *)&oh,4);

wh.wordid=0;           // nullword=>fb.dat
wh.hits=0;
fb1fp1.write((char *)&wh,4);

doc.docid=0;          // nulldocid=>fb.dat
fb1fp1.write((char *)&doc,4);

fb1fp1.close();

}
```

## Appendix E Inverted Barrel Setup Method

An inverted barrel is associated with its index( an array). The inverted index(include an array and an inverted barrel) is as follows:

Array			Inverted barrel		
WordID	Docs	Offset	DocID	Hits	Hitdata
3	2	—————▶	9	2	1 111 0000 0000 0001 0 100 0000 0000 0110
			16	4	1 111 0000 0000 1000 0 000 0000 0000 0011 0 010 0000 0000 0010 0 000 0000 0001 1111
10	1	—————▶	8	1	0 011 0000 0000 1111

We use following code to create the inverted index(include array and inverted barrel)

```
#include <stdio.h>
#include <iostream.h>

void main(void)
{
    struct
    {
        int wordid;
        int docs;
        long offset;
    } ibindex[2];

    struct                                // store wordid+hits
    {
        unsigned docid:24;
        unsigned hits:8;
    } dh;

    struct                                // store one hit
    {
```

```

    unsigned cap1:1; // first word first letter's capitalization, 0 lowercase
                        //1 uppercase
    unsigned imp1:3; // word size or fancy type
    unsigned position1:12; // word position in the text
    unsigned cap2:1; // next word
    unsigned imp2:3;
    unsigned position2:12;
    }oh;

fstream ib1fp1("ib1.dat", ios::out|ios::in|ios::binary); // inverted barrel

ibindex[1].wordid=3; // write to the array
ibindex[1].docs=2;
ibindex[1].offset=ibfp1.tellp();

dh.docid=9; // docid 9 and hits 2 =>ib.dat
dh.hits=2;
ib1fp1.write((char *)&wh,4);

oh.cap1=1; // two hits =>ib.dat
oh.imp1=7;
oh.position1=1;
oh.cap2=0;
oh.imp2=4;
oh.position2=6;
ib1fp1.write((char *)&oh,4);

dh.docid=16; // docid 16 and hits 4 =>ib.dat
dh.hits=4;
ib1fp1.write((char *)&wh,4);

oh.cap1=1; // 4 hits=>ib.dat
oh.imp1=7;
oh.position1=8;
oh.cap2=0;
oh.imp2=0;
oh.position2=3;
ib1fp1.write((char *)&oh,4);

oh.cap1=0;
oh.imp1=2;
oh.position1=2;
oh.cap2=0;

```

```
oh.imp2=0;
oh.position2=31;
ib1fp1.write((char *)&oh,4);

ibindex[2].wordid=10; // write to the array
ibindex[2].docs=1;
ibindex[2].offset=ibfp1.tellp();

dh.docid=8; // docid 8 and hits 1 =>ib.dat
dh.hits=1;
ib1fp1.write((char *)&wh,4);

oh.cap1=0;
oh.imp1=3;
oh.position1=15;
ib1fp1.write((char *)&oh,4);

ib1fp1.close();
}
```