Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Project Report
on
"A Mini Search Engine"

[Code No:COMP 202]

(For partial fulfillment of II Year/I in Computer Engineering)

Submitted by:

Pratik Dhungana (037968-24)

Submitted to:

Er. Sagar Acharya

Department of Computer Science and Engineering

Kathmandu University, Nepal

Submission Date: 25/02/2026

Abstract

This project presents the design and implementation of a console-based Mini Search Engine developed in C using fundamental Data Structures and Algorithms. The system constructs an inverted index to enable efficient document retrieval based on user queries. A hash table with separate chaining is used to store unique words, while linked lists maintain posting information containing document identifiers and term frequencies. During indexing, input documents are processed through tokenization, normalization (case folding and punctuation removal), and stop-word elimination to improve relevance and reduce storage overhead. The search engine supports single-keyword queries and Boolean AND queries, ranking results based on term frequency. Additionally, the system provides analytical features such as top frequent word identification and hash table performance statistics, including load factor and collision analysis. The implementation emphasizes algorithmic efficiency, dynamic memory management, and modular design. Although simplified compared to industrial search engines, this project effectively demonstrates core concepts of hashing, linked lists, sorting, and Boolean retrieval in a practical information retrieval system implemented entirely in C.

Keywords: Inverted Index, Hash Table, Information Retrieval, Posting List, Term Frequency, Data Structures

Acknowledgements

I would like to express my sincere gratitude to everyone who supported and guided me throughout the completion of this project.

First, I would like to thank my project supervisor for providing valuable guidance, constructive feedback, and continuous encouragement during the design and implementation phases of the Mini Search Engine. Their insights helped me refine the project objectives, improve the system design, and overcome technical challenges.

I would also like to acknowledge the authors and contributors of the books, tutorials, and online resources I referred to during this project. The guidance provided by *Introduction to Information Retrieval* by Manning et al., *Introduction to Algorithms* by Cormen et al., and online platforms like GeeksforGeeks and TutorialsPoint was instrumental in understanding core data structures, algorithms, and information retrieval techniques.

Finally, I would like to thank my family and friends for their moral support and encouragement throughout the project. Their patience and motivation were invaluable during moments of intensive coding and debugging.

This project would not have been possible without the support and resources provided by all of the above.

Table of Contents

Acronyms/Abbreviations

ISE: Inverted Search Engine

TF: Term Frequency

IDF: Inverse Document Frequency

IR: Information Retrieval

HT: Hash Table

PN: Posting Node

HN: Hash Node

DOC: Document

AND: Boolean AND Operation

TOP: Top Words

Chapter 1 : Introduction

Information retrieval systems are fundamental in managing and searching large volumes of textual data. Search engines rely on efficient indexing mechanisms to retrieve relevant documents quickly. At the core of such systems lies the inverted index, which enables direct mapping between words and the documents containing them. This project implements a simplified mini search engine in C that demonstrates the practical application of core Data Structures and Algorithms, particularly hashing and linked lists, in building an efficient document retrieval system.

1.1 Background

Search engines rely on specialized indexing techniques to retrieve documents efficiently. One of the most fundamental structures used in information retrieval systems is the inverted index, which maps words to the documents in which they appear. Instead of scanning every document for each query, the inverted index allows direct access to relevant documents through indexed word entries.

To implement an efficient inverted index, hashing is commonly used. A hash table enables average constant-time lookup, insertion, and retrieval of words. Collisions are handled using separate chaining, where linked lists store multiple entries within the same bucket. Each indexed word maintains a posting list containing document identifiers and frequency counts, allowing ranking of results based on term occurrence.

This project builds upon these foundational principles to demonstrate how core data structures can be applied in real-world retrieval systems.

1.2 Objectives

The main objectives of the project are:

- To design and implement an inverted index using a hash table with separate chaining in C.

- To implement efficient keyword and Boolean AND search functionality with frequency-based ranking.

- To apply linked lists for managing posting lists and dynamic memory allocation.

- To analyze the performance of the indexing system using statistical metrics such as load factor and collision distribution.

1.3 Motivation and Significance

The motivation behind this project is to transform theoretical knowledge of data structures into a practical, working system that demonstrates real-world application. Concepts such as hashing, linked lists, sorting, and frequency counting are often studied in isolation; however, integrating them into a search engine provides a comprehensive understanding of how these structures interact within a larger system. This project is significant because it illustrates how efficient indexing reduces computational overhead, how hash tables improve retrieval performance, and how memory management plays a critical role in systems programming. Although simplified compared to large-scale commercial search engines, the project captures the essential mechanisms used in information retrieval systems and strengthens problem-solving and algorithmic design skills.

1.4 Expected Outcomes

The project expected the following outcomes:

- A functional console-based mini search engine capable of indexing multiple documents.

- Accurate retrieval of documents using single-keyword and AND-based search queries.

- Ranked search results based on term frequency within documents.

- Performance analysis output detailing hash table utilization, load factor, and collision behavior.

Chapter 2 : Related Works/ Existing Works

Early information retrieval systems relied on sequential scanning of documents, where each document was searched linearly for query terms. Although simple to implement, sequential search methods were computationally expensive and inefficient for large document collections. As datasets grew, researchers introduced indexing techniques to reduce retrieval time and improve scalability.

The development of inverted indexing marked a major advancement in retrieval efficiency. By mapping terms directly to the documents in which they appear, inverted indexes significantly reduced search complexity and became the standard structure used in text retrieval systems.

2.1 Inverted Index-Based Search Systems

Most modern search engines are built upon inverted index structures. Academic and industrial research has demonstrated that inverted indexes provide efficient query processing, especially for keyword-based and Boolean search models.

Inverted index implementations typically include:

- A vocabulary of unique terms
- Posting lists containing document identifiers
- Frequency or positional information

Large-scale search engines extend this concept by incorporating distributed indexing and compression techniques. However, the fundamental structure remains the inverted index, which is also implemented in this project in a simplified form.

2.2 Hash Table-Based Indexing

Hash tables are widely used in indexing systems to provide fast lookup performance. Many dictionary-based search systems use hashing to map terms to

posting lists efficiently. Separate chaining is a common collision resolution technique because it maintains flexibility and avoids clustering issues seen in open addressing methods.

Research in hash-based indexing emphasizes:

- Selection of an effective hash function
- Maintaining optimal load factor
- Reducing collision chains

The use of the djb2 hash function in this project aligns with common practices for string-based hashing due to its good distribution properties and computational efficiency.

Numerous academic projects and instructional materials demonstrate mini search engines implemented using C, C++, or Java to teach core data structures. These implementations typically include:

- Hash tables for vocabulary storage
- Linked lists for posting lists
- Basic Boolean query support
- Simple ranking mechanisms

The Mini Search Engine developed in this project aligns with these educational implementations but additionally includes statistical analysis of the hash table, stop-word removal, and top-frequency word analysis to provide a more comprehensive demonstration of data structure applications.

Chapter 3: Design And Implementation

3.1 System Design

3.1.1 Architectural Design

The system consists of the following logical modules:

- Document Management Module
- Hash Table Indexing Module
- Posting List Management Module
- Query Processing Module
- Ranking Module
- Statistics Module
- Memory Management Module

Each module performs a specific function, ensuring separation of concerns and modularity.

3.1.2 Data Structure Design

A. Document Structure

Each document is assigned a unique document ID and stored in a document registry:

- Document ID (integer)
- Filename (string)

This structure allows efficient mapping from search results to actual filenames.

B. Hash Table

A fixed-size hash table of 1024 buckets is used:

- Each bucket stores a linked list of hash nodes.

- Collisions are handled using separate chaining.

The hash function used is the djb2 algorithm, which provides good distribution for string keys.

Hash index computation:

[
index = hash(word) \mod TABLE_SIZE
]

C. Hash Node Structure

Each hash node contains:

- The word (string key)
- Pointer to posting list
- Pointer to next hash node (for collision chaining)

This allows multiple words to exist within the same bucket.

D. Posting List Structure

Each word maintains a linked list of posting nodes. Each posting node contains:

- Document ID
- Frequency of occurrence in that document
- Pointer to next posting node

This structure forms the inverted index representation:

[
word \rightarrow (doc_id, frequency)
]

3.2 Implementation Details

3.2.1 Hash Function Implementation

The system uses the djb2 hash algorithm:

- Initializes hash value to 5381
- Iteratively updates hash using bit shifting and addition

This ensures uniform word distribution across buckets.

Average time complexity:
O(1) for lookup and insertion.

3.2.2 Document Processing

When a file is loaded:

1. The file is opened in read mode.
2. Words are extracted using formatted input.
3. Each word undergoes preprocessing:
    - Non-alphabetic characters removed
    - Converted to lowercase
4. Stop words are filtered.
5. Valid words are inserted into the index.

This ensures normalized and consistent indexing.

3.2.3 Word Insertion Algorithm

For each processed word:

1. Compute hash index.
2. Traverse bucket to check if word exists.
3. If word exists:
    - Traverse posting list.

    - Increment frequency if document already present.
4. If word does not exist:
    - Create new hash node.
    - Create new posting node.
    - Insert at head of bucket.

This guarantees correct frequency tracking and efficient updates.

3.2.4 Single Keyword Search Implementation

The single-word search process includes:

1. Normalize user input.
2. Compute hash index.
3. Locate word in bucket.
4. Retrieve posting list.
5. Sort posting list in descending frequency.
6. Display ranked results with document names.

If the word does not exist, the system reports no results found.

3.2.5 Boolean AND Search Implementation

The AND search follows these steps:

1. Normalize both query terms.
2. Retrieve posting lists for both words.
3. Compute intersection of document IDs.
4. Combine frequencies of both words.
5. Sort results by combined frequency.
6. Display ranked results.

This implements basic Boolean retrieval logic.

3.2.6 Ranking Mechanism

The system uses Term Frequency (TF) as the ranking metric:

- Higher frequency implies higher relevance.
- For AND search, combined frequency determines ranking.

Sorting is implemented using a comparison-based method suitable for small datasets.

3.2.7 Top Frequent Words Feature

The system calculates global word frequency by:

1. Traversing the entire hash table.
2. Summing posting frequencies for each word.
3. Maintaining a top-10 sorted list.
4. Displaying ranked words with total frequency.

This provides insight into term distribution across documents.

3.2.8 Statistical Analysis Implementation

The statistics module computes:

- Total used buckets
- Load factor
- Unique word count
- Total word-document pairs
- Longest collision chain
- Number of indexed documents

These metrics evaluate the performance and efficiency of the hash table.

3.2.9 Memory Management

Dynamic memory allocation is used for:

- Hash nodes

- Posting nodes

Before program termination:

- All posting nodes are freed.
- All hash nodes are freed.
- Buckets are reset to NULL.

This ensures no memory leaks and proper resource deallocation.

Chapter 4: Results and Discussion

4.1 Implemented Features

Several key features were successfully implemented in the Mini Search Engine. These features form the core functionality of the project and directly support the goal of efficient document retrieval.

The main implemented features are:

- **Document Indexing:** Multiple text files can be loaded, tokenized, and indexed efficiently using a hash-based inverted index.
- **Single Keyword Search:** The system retrieves all documents containing a queried word and ranks results based on frequency.
- **Boolean AND Search:** Users can search for documents containing two keywords simultaneously, with combined frequency-based ranking.
- **Top Frequent Words Analysis:** The system identifies and displays the top 10 most frequent words across all indexed documents.
- **Index Statistics:** Provides detailed information about hash table utilization, unique words, total postings, and collision chains.

4.1.1 Results and Performance Analysis

The implemented system produced reliable and consistent results during execution.

Functional Performance:

- Words were correctly tokenized, normalized, and inserted into the inverted index.
- Stop words were filtered successfully to improve result relevance.
- Single-word search returned accurate and correctly ranked results.
- Boolean AND search correctly computed intersections and ranked results by combined frequency.

- Top frequent words feature correctly identified words with the highest cumulative frequency.

System Performance:

- Indexing time scaled approximately linearly with the total number of words processed.
- Single keyword searches exhibited near constant-time lookup due to hashing.
- AND search operations performed efficiently for small to medium datasets, though nested traversal is O(n × m).
- Memory usage scaled with the number of unique words and postings; all dynamically allocated memory was freed upon program termination.

Overall, the system functioned correctly and met the expected performance requirements for a small-scale search engine.

4.2 Comparison with Objectives

Comparison with Original Objectives:

- **Efficient Document Indexing:** Achieved using a hash table-based inverted index.
- **Keyword Search:** Achieved with single-word search functionality.
- **Boolean Search Support:** Achieved using AND search on posting lists.
- **Analysis of Word Frequencies and Statistics:** Implemented through top frequent words and index statistics modules.

4.3 Challenges and Limitations

Several challenges were encountered during development:

- **Hash Table Collisions:** Properly handling collisions while maintaining fast insertion and retrieval required careful implementation of separate chaining.
- **Text Normalization:** Removing non-alphabetic characters, converting to lowercase, and handling long words required careful preprocessing.
- **AND Search Traversal:** Computing intersections of posting lists in an efficient way posed challenges for larger lists.
- **Memory Management:** Ensuring all dynamically allocated memory was freed correctly required careful traversal and deallocation of linked lists.

Limitations of the system include:

- Fixed hash table size limits scalability; performance may degrade with very large datasets.
- Ranking is based solely on term frequency; advanced metrics like TF-IDF are not used.
- AND search uses nested traversal rather than a more efficient merge-based algorithm.
- Only single-word and two-word searches are supported; OR, NOT, and phrase searches are not implemented.

4.4 Discussion

The goal of the project was to implement a functional mini search engine demonstrating practical applications of data structures such as hash tables and linked lists.

During development, the scope expanded to include statistical analysis, top frequent word identification, and document indexing with stop-word removal. Preprocessing of text proved crucial to ensure accurate indexing, and the use of a hash table enabled fast word lookup.

Single-word and AND search operations confirmed the effectiveness of the inverted index structure. Although the ranking mechanism is simple, it effectively demonstrates the concept of frequency-based relevance.

The system encountered minor challenges with memory management and posting list traversal, but these were resolved through careful implementation of dynamic allocation and deallocation.

While the system is not designed for large-scale or internet-scale retrieval, it provides a clear and practical demonstration of core information retrieval techniques. Overall, the project met its objectives and provided valuable insight into implementing search engines using fundamental data structures.

Chapter 5: Conclusion and Future Works

5.1 Conclusion

The Mini Search Engine project successfully demonstrates the practical application of fundamental data structures—hash tables and linked lists—in the field of information retrieval. Through the implementation of an inverted index, the system allows efficient indexing and retrieval of multiple text documents.

Key outcomes of the project include:

- Accurate document indexing with proper text preprocessing, including tokenization, normalization, and stop-word removal.
- Efficient single keyword search, returning ranked results based on term frequency.
- Boolean AND search functionality for retrieving documents containing multiple keywords.
- Computation and display of top frequent words and index statistics, providing insights into dataset characteristics.

The system meets the original project objectives, performing reliably on small to medium-sized datasets. It provides a clear demonstration of how theoretical concepts from data structures and algorithms can be translated into a functional, working system. While simple in design, the Mini Search Engine is modular, maintainable, and provides a solid foundation for further enhancements.

5.2 Future Works

Although the system achieves its intended functionality, several areas for improvement and extension exist:

1. **Dynamic Hash Table Resizing:** Implementing a dynamically resizable hash table would improve scalability and reduce collisions in larger datasets.

2. **Advanced Ranking Techniques:** Incorporating metrics such as TF-IDF (Term Frequency–Inverse Document Frequency) or BM25 could improve the relevance of search results.
3. **Extended Boolean Search:** Support for OR, NOT, and phrase queries would make the system more flexible and closer to real-world search engines.
4. **Performance Optimization:** Replace the current sorting algorithm for posting lists with more efficient algorithms, especially for larger datasets, and implement merge-based AND search to reduce computational complexity.
5. **Graphical User Interface (GUI):** Adding a lightweight GUI could improve usability and provide a more user-friendly experience.
6. **Support for Large Datasets:** Implementing persistent storage and memory-mapped indexing could allow the system to handle thousands of documents efficiently.

By addressing these areas, the Mini Search Engine could evolve into a more robust, scalable, and feature-rich information retrieval system while maintaining its educational value.

References

1. Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.
2. GeeksforGeeks. (n.d.). *Inverted Index Implementation in C/C++*. Retrieved from https://www.geeksforgeeks.org/inverted-index/
3. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
4. TutorialsPoint. (n.d.). *Hash Table Data Structure in C*. Retrieved from https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithm.htm