Assignment 1: Indexer Implementation

Session: 2021 – 2025

Submitted by:

Ghulam Mustafa (2021-CS-39)

Supervised by:

Sir. Khaldoon Khurshid

Department of Computer Science

University of Engineering and Technology Lahore Pakistan

Table of Contents

Overview

The Document Search Indexer is a Python-based application designed to index and search text documents within a specified directory. Using data structures like linked lists and dictionaries, it constructs an efficient index to store words and the documents where they appear. This program is useful for basic search functionality across a collection of documents, making it ideal for educational projects or small document indexing systems.

Indexers and Their Importance

An indexer is a data structure or algorithm used to efficiently map data to its associated values, enhancing search operations by organizing and storing information for quick access. In this application, the indexer allows users to search for specific words across multiple text documents and retrieve relevant documents and their counts efficiently.

Indexing is fundamental in many domains, from database management to information retrieval, as it dramatically reduces search times by pre-structuring the data.

Key Concepts

- Linked List: Utilized to store unique document names per word.
- Dictionaries: Used for quick lookup of words in the index.
- Text Processing: Text is processed for standardization, improving indexing accuracy.

Working Flow of the program:

1. Initialization of Linked List Structures
2. Creation of an Indexing System
3. Text Processing
4. Indexing Documents
5. Searching for words
6. Displaying Results

Difference Between using a Dictionary and a Linked List:

1. Memory Efficiency

Linked List: Each node stores data and a pointer, using more memory but avoiding reallocation.

List: Lists need occasional reallocation, consuming extra memory and impacting performance with frequent size changes.

2. Insertion Time Complexity

Linked List: Efficient sequential insertions, but checking for duplicates takes O(n) time.

List: Insertion is O(1) on average, but checking for duplicates requires O(n) time.

3. **Search Time Complexity**

   Linked List: Search requires traversing nodes (O(n)), but avoids shifting during insertions/deletions.

   List: Search is also O(n), becoming costly for large collections, especially with frequent searches.

4. **Deletion Time Complexity**

   Linked List: Deletion requires finding the node (O(n)) but no element shifting.

   List: Deletion also takes O(n), but requires shifting elements, reducing efficiency if deletions are frequent.

5. **Structure Complexity**

   Linked List: Requires a separate Node class and pointer management, adding complexity.

   List: Simpler, directly managing elements in a single array-like structure.

Code Explaination:

Define the Node Class

The Node class is a fundamental component of the linked list, representing an individual item in the list. Each node contains:

- data: The document name where the word is found.
- next: A pointer to the next node in the list.

Code:

```
class Node:

    def __init__(self, data=None):

        self.data = data  # Data to store (document name)

        self.next = None  # Pointer to the next node
```

Define the LinkedList Class

The LinkedList class manages Node instances, storing document names where a specific word appears. It includes methods for:

- Insertion (insert): Adds a document name to the list if it isn't already present.
- Searching (search): Checks if a document name exists in the list.
- Displaying documents (display_documents): Returns a list of document names in the linked list.
- Getting document count (get_document_count): Returns the count of documents containing the word.

Code:

```
class LinkedList:
 def __init__(self):
        self.head = None
        self.count = 0  # Track how many documents contain the word
    def insert(self, data):
        if not self.search(data):
            new_node = Node(data)
            if self.head is None:
                self.head = new_node
            else:
                current = self.head
                while current.next:
                    current = current.next
                current.next = new_node
            self.count += 1
    def search(self, data):
        current = self.head
        while current:
            if current.data == data:
                return True
            current = current.next
        return False
    def display_documents(self):
        documents = []
        current = self.head
        while current:
            documents.append(current.data)
            current = current.next
        return documents
    def get_document_count(self):
        return self.count
```

Create the DocumentIndex Class

The DocumentIndex class is the main indexing structure, storing words as keys in a dictionary and linking each word to a LinkedList of document names. This class supports:

- Adding a word (add_word): Inserts the document name into the linked list associated with the word.
- Searching for a word (search_word): Retrieves the list of documents where the word is found and the count of such documents.

Code:

```
class DocumentIndex:
    def __init__(self):
        self.index = {}  # Dictionary to store words with linked lists of documents
    def add_word(self, word, document):
        if word not in self.index:
            self.index[word] = LinkedList()
        self.index[word].insert(document)
    def search_word(self, word):
        if word in self.index:
            return {
                'documents': self.index[word].display_documents(),
                'count': self.index[word].get_document_count()
            }
        return {'documents': [], 'count': 0}
```

Text Preprocessing

- Removing punctuation: Uses regular expressions to remove non-alphabetic characters.
- Converting to lowercase: Ensures that the text is case-insensitive.
- Removing stop words: Filters out common, unimportant words like "and", "the", "is", etc.
- Stemming: A simple stemming function removes common suffixes like -ing, -ed, and -ly.
- Stop Words: A predefined set of words that are filtered out because they do not contribute significant meaning to the search.
- Simple Stemming: A basic approach to reduce words like "running" to "run" and "happily" to "happi".

Code:

```
def preprocess_text(text):
    """Preprocess text: remove punctuation, lowercase, remove stop words, and apply simple stemming."""
    # Define simple stopwords (a small set of common English words)
    stop_words = set([
        'a', 'an', 'the', 'and', 'but', 'or', 'so', 'for', 'nor', 'to', 'of', 'in', 'on', 'at', 'by', 'with', 'about', 'as', 'from', 'that', 'which', 'who', 'whom', 'this', 'it', 'its', 'i', 'you', 'he', 'she', 'we', 'they', 'them', 'their', 'ours', 'your', 'yours', 'is', 'are', 'was', 'were', 'be', 'been', 'being'
    ])
    # Simple stemming function: Remove common suffixes
    def simple_stem(word):
        if word.endswith('ing'):
            return word[:-3]
        elif word.endswith('ed'):
            return word[:-2]
        elif word.endswith('ly'):
            return word[:-2]
        return word
    # Remove non-alphabetic characters (punctuation) and tokenize
    text = re.sub(r'[^a-zA-Z\s]', '', text)
    # Convert text to lowercase and split into words
    words = text.lower().split()
    # Remove stop words and apply stemming
    processed_words = [simple_stem(word) for word in words if word not in stop_words]
    return ' '.join(processed_words)
```

Indexing Documents

Purpose: Reads all .txt files in a given folder, processes the text, and adds each word in the document to the index.

- Each document is read and preprocessed.
- After preprocessing, the words are added to the DocumentIndex object.
- A set is used to ensure each word appears only once per document.

Code:

```
# Step 5: Read and Index Documents using LinkedList
```

```
def index_documents(doc_index, folder):
"""Read all documents in the folder and index words."""
    for filename in os.listdir(folder):
        if filename.endswith(".txt"):
            with open(os.path.join(folder, filename), 'r', encoding='utf-8') as file:
                content = file.read()
                content = preprocess_text(content)
                words = set(content.split())  # Use set to avoid duplicate words in the same
document
                for word in words:
                    doc_index.add_word(word, filename)
```

Searching for a Word

Purpose: Allows the user to search for a specific word in the index.

- The query is preprocessed and then searched in the DocumentIndex.
- The function returns a dictionary with the documents containing the word and the count of documents.

Code:

```
def search_documents(doc_index, query):
    """Search for a word in the index and retrieve document information."""
    query = preprocess_text(query)
    return doc_index.search_word(query)
```

Display Results

- If the word is found, it shows the count of documents and the names of the documents.
- If no documents are found, it displays a message indicating no matches.

Code:

```
def display_results(results):
    """Display search results with document count."""
    print(Fore.CYAN + "-" * 40)
    if results['documents']:
        print(Fore.GREEN + f"Found in {results['count']} document(s):" +
Fore.YELLOW + f" {', '.join(results['documents'])}")
    else:
```

```
    print(Fore.RED + "No matching documents found.")
print(Fore.CYAN + "-" * 40)
```