Project: Developing and Optimizing Data Structures for Real-World Applications Using Python

Phase 2: Data Structure Design and Implementation (Deliverable 2)

Avijit Saha

Student ID #005023281

Algorithms and Data Structures (MSCS-532-B01)

Dr. Vanessa Cooper

November 17, 2024

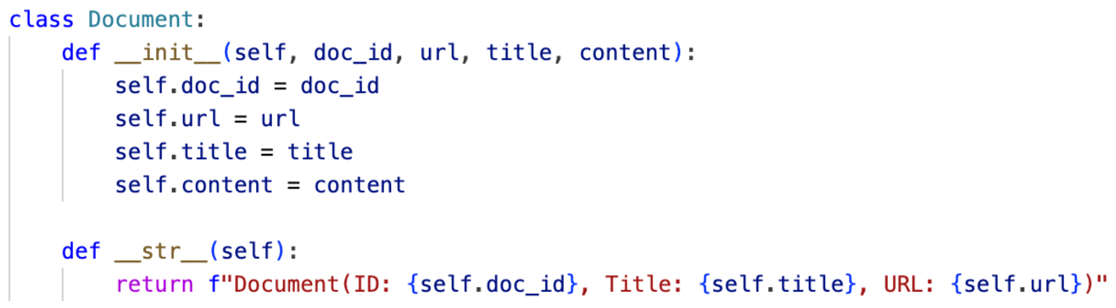
Implementation

In phase 2, I will use two primary data structures of the SEO application. This implementation has been designed to determine the critical functionality of these data structures: insertion, document retrieval, and keyword retrieval from the inverted index. The data structures implemented till now are:

Document Class: The Document class represents an individual document in the SEO system. Each document has the following attributes.

- **id_doc:** A unique identifier of a document; this is essential for distinguishing between different documents in the system.
- **url:** The URL from which the document is available. This would further facilitate the system to refer to the location of the document online.
- **Title:** The document's title may provide a quick content summary.
- **Content:** The main body content of the document. The content is essential, as it will be parsed and indexed for search operations.

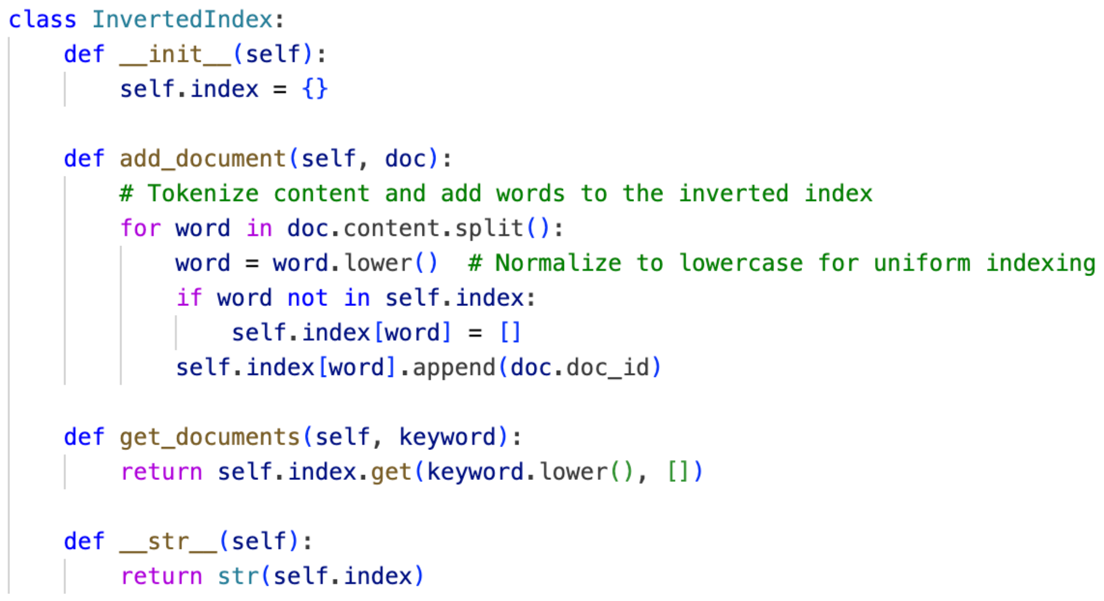
This class is essential because it wraps up metadata about a document and its textual content. It thus permits the efficient storage and retrieval of information related to individual documents for subsequent use in indexing and searching.



```python
class Document:
    def __init__(self, doc_id, url, title, content):
        self.doc_id = doc_id
        self.url = url
        self.title = title
        self.content = content

    def __str__(self):
        return f"Document(ID: {self.doc_id}, Title: {self.title}, URL: {self.url})"
```

Figure1: Document Class



```python
class InvertedIndex:
    def __init__(self):
        self.index = {}

    def add_document(self, doc):
        # Tokenize content and add words to the inverted index
        for word in doc.content.split():
            word = word.lower()  # Normalize to lowercase for uniform indexing
            if word not in self.index:
                self.index[word] = []
            self.index[word].append(doc.doc_id)

    def get_documents(self, keyword):
        return self.index.get(keyword.lower(), [])

    def __str__(self):
        return str(self.index)
```

Figure 2: InvertedIndex class

InvertedIndex Class: InvertedIndex is a class that links keywords to a list of document IDs. It acts as the primary entity in the efficient implementation of keyword-based searches for documents. The main methods that should be included in the class are:

- **add_document(doc):** This function will get the textual content of a document, break it up into individual words, and store the document's ID under each keyword found in the document. Each keyword is normalized to loIr case to handle case-insensitive searches.
- **get_documents(keyword):** This function will be used when searching for documents containing a specific keyword. It returns a list of document IDs that contain the keyword.

All the corresponding documents are fetched in constant time for any given keyword, given the inverted index. This makes searching for records that match keywords very efficient.

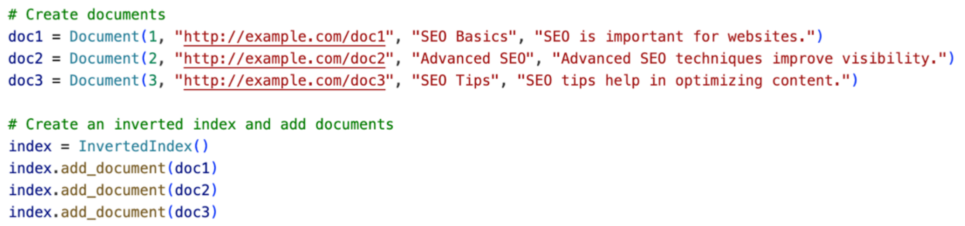
These two classes form the foundation of the SEO application by enabling the indexing of documents and supporting efficient searches based on keywords.

For now, I will implement these two classes because they are significant parts of this project for creating the documents and searching for the keywords inside the papers. These are going to be one of the main features of this project.

Later, I will add other features like Queue class in this project, where I will implement the heapsort technique. It will be helpful to rank the documents.

Demonstration and Testing

To demonstrate it, I have created a list of documents where I can search for a keyword in the papers. Then, I added it to the documents.

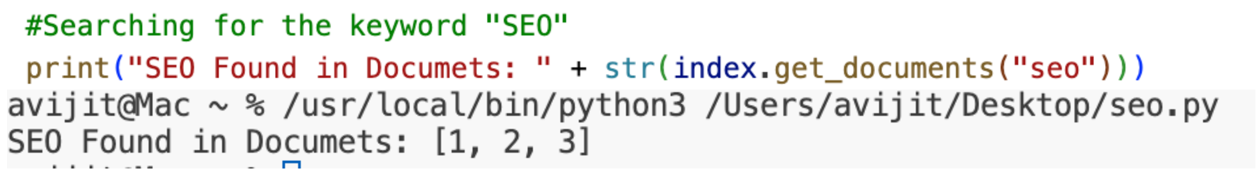


```python
# Create documents
doc1 = Document(1, "http://example.com/doc1", "SEO Basics", "SEO is important for websites.")
doc2 = Document(2, "http://example.com/doc2", "Advanced SEO", "Advanced SEO techniques improve visibility.")
doc3 = Document(3, "http://example.com/doc3", "SEO Tips", "SEO tips help in optimizing content.")

# Create an inverted index and add documents
index = InvertedIndex()
index.add_document(doc1)
index.add_document(doc2)
index.add_document(doc3)
```

Figure 3: Creating and Adding Documents

Then, I created a series of test cases to validate the core functionalities of these data structures. Testing is crucial in ensuring that the implementation works as expected and that edge cases are handled correctly. Here are the critical test cases that Ire created:



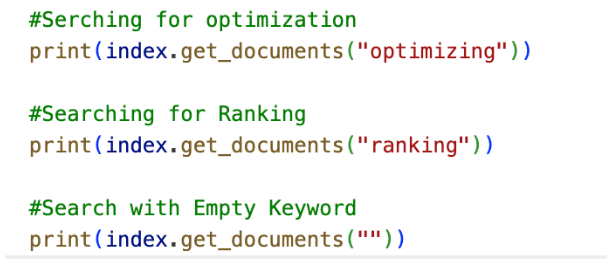
```python
#Searching for the keyword "SEO"
print("SEO Found in Documets: " + str(index.get_documents("seo")))
```

```
avijit@Mac ~ % /usr/local/bin/python3 /Users/avijit/Desktop/seo.py
SEO Found in Documets: [1, 2, 3]
```

Figure 4: Output for the Current implementation

Searching for Keyword SEO: This is where I have to query for documents containing the keyword "SEO." What is expected is for all the documents talking about "SEO" to appear in the result and for the output to contain only the IDs of relevant documents.

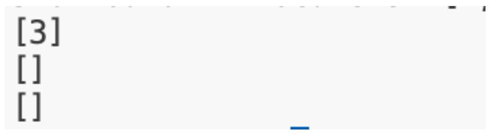
Searching another Keyword: In this case, the system was expected to check correctly and return only those documents containing the word "optimization." Since document ID 1 has the keyword "optimization" in its content, it is expected to return document ID 1.



```
#Serching for optimization
print(index.get_documents("optimizing"))

#Searching for Ranking
print(index.get_documents("ranking"))

#Search with Empty Keyword
print(index.get_documents(""))
```



```
[3]
[]
[]
```

Figure 5: Searching By keyword

This test exercises how the system would react in case of searching for a keyword that does not exist within the set of documents. The function should return an empty list since no document contains the word "ranking."

Implementation Challenges and Solutions:

Some challenges arose while implementing the classes Document and InvertedIndex. I gave some thought to each challenge and adjusted the design accordingly.

Challenge 1: Case Sensitivity

One such challenge I encountered was how to handle case sensitivity. The words within a document can come in various case forms, for example, "SEO," "Seo," and "SEO." This would lead to different entries of the same word being made without normalization.

Solution: To address this, I changed all words to lowercase before inserting them into the index. This makes all searches case insensitive so that all forms of a word have the duplicate entry in the inverted index.

Challenge 2: Handling Duplicates in Document Content

Another problem was dealing with the same keyword occurring more than once in the document. In SEO, a keyword's occurrence rate can affect a document's relevance, and I should handle this efficiently.

Solution: I kept several occurrences of the same document ID concerning each keyword. In this way, the system would represent correctly how often each keyword appeared inside the documents. I might have optimized the index to keep unique document IDs, but our driving factor was simplicity and correctness.

Challenge 3: Empty and Non-Existent Keywords

This is because users may search for an empty string or some keyword not present in any document. If not treated accordingly, it could raise an error or behave unexpectedly.

Solution: I added a check to ensure that empty strings and non-existent keywords return an empty list. This makes the system more robust and user-friendly.

Next Steps

Further steps will involve extending the SEO application and covering other critical features that real-world applications of SEO use. These could be:

Complete Implementation of Ranking and Relevance Scoring: Next, a priority queue could be implemented to sort the documents according to relevance scores. These scores can be pre-determined by the frequency of keyword occurrences, the document's title, and other SEO-related factors. The system would thus return documents in order of ranking according to their relevance to the search query.

Advanced Search Features: I will enhance the searching capability in the system by incorporating advanced features that will include:

> **Boolean searches**(AND, OR, NOT) allow the user to combine the terms being searched for in elaborate ways.
>
> **Phrase searches** allow for a search for an exact phrase rather than just a keyword.
>
> **Wildcard searches** to allow partial matches on keywords.

Scalability Testing: I will benchmark the system with more documents to guarantee it scales Ill. This entails indexing thousands of documents and ensuring that search operations remain efficient.

Integration with External Data: I will integrate the SEO application with real-life data sources such as Ib crawlers or third-party APIs in future work. This way, the system could index in real-time and thus provide updated search results.

References:

Baeza-Yates, R., & Ribeiro-Neto, B. (1999). Modern Information Retrieval: The Concepts and Technology behind Search. Addison-Wesley Longman.

Chakrabarti, S. (2003). Mining the Web: Discovering Knowledge from Hypertext Data. Morgan Kaufmann.

Manning, C. D., Raghavan, P., & Schütze, H. (2008). Introduction to Information Retrieval. Cambridge University Press.