**Project Phase 1: Data Structures Design and Implementation for Search Engine Optimization**

MSCS-532: Algorithms and Data Structures

Instructor: Satish Penmatsa

Department of Computer Information Sciences, University of the Cumberlands

24th October, 2025

Mausam Shrestha

Bishesh Tuladhar

Prashanna Acharya

Ujjwal Khadka

**Search Engine Optimization: Data Structure Design and Implementation**

**Abstract**

In this report, the concept of data structures and algorithms is utilized to design and implement a search engine system that can handle large scale queries across vast datasets as well as provide efficient informational results. For an efficient search engine system, multiple key data structures are implemented in this application, such as for linking and connecting documents and terms, inverted indexes are utilized, for prefixes based searching trie is utilized, for calculating the importance of pages or data sites graph is utilized, and for ranking results, priority queues are utilized. This project displays the utilization of concepts and knowledge acquired in this course in real world situations. This project also provides a detailed experimental performance analysis displaying the efficiency and utilization of the above data structures.

**Introduction**

In today’s advancing world, the importance of search engines can be seen everywhere. Also, search engines play an important role in modern computer science where large amounts of data and information is required to be stored and handled efficiently. The most important factor is the challenges to design systems that can process the vast amount of data and provide response or results within a fraction of seconds. In this project, these challenges are tackled with the help of above mentioned key data structures to build an efficient search engine application.

The main goal of this project is to show the utilization of theoretical concepts and ideas from this course into a practical and real world application, which demonstrates high efficiency and performance. In this application, four key data structures are utilized, such as inverted indexes, trie, priority queues, and graphs, to build a search engine application.

**Application Context and Data Structure Selection**

**Search Engine Architecture**

The application works with multiple collections of web documents and connects each document through text for an efficient search engine. The goal of the application is to handle document indexing or linking, proper processing of user queries, and accurately ranking the datasets search results. In order to achieve the goal, multiple specialized data structures are integrated together in this application, providing accurate and fast response.

The architecture is designed in such a way that each data structure implemented in the application serves a specific and intended purpose while working together to provide better system performance. The inverted indexes handles the text searching and document linking or connecting function, the trie supports the autocomplete function, the priority queue handles the ranking results, and the graph provides the importance of the datasets pages.

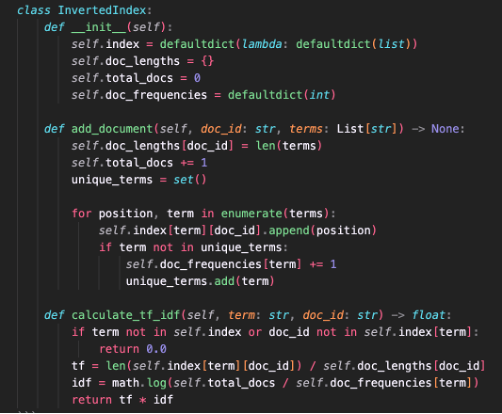
**Core Data Structures**

*Inverted Index*

Fast text search is enabled by the inverted index. The inverted index associates each term with the list of documents in which it occurs. Conceptually, the inverted index is an associative array (or hash table) whose keys are terms and whose values are posting lists (document references and term frequencies). The inverted index also stores term positions to enable position-based advanced relevance scoring.

Design Rationale: Hash tables are perfect for the main search function because they enable you to get terms in O (1) average time. The data structure keeps track of term frequencies and document lengths to facilitate TF-IDF (Term Frequency-Inverse Document Frequency) scoring. This is the standard approach to estimating relevance in information retrieval.

Implementation: The inverted index uses Python's defaultdict to build nested dictionaries that map terms to documents and positions. Having document frequencies stored separately allows for IDF calculations. The implementation includes maintaining the position of phrase queries and ranking based on how close they are to each other.

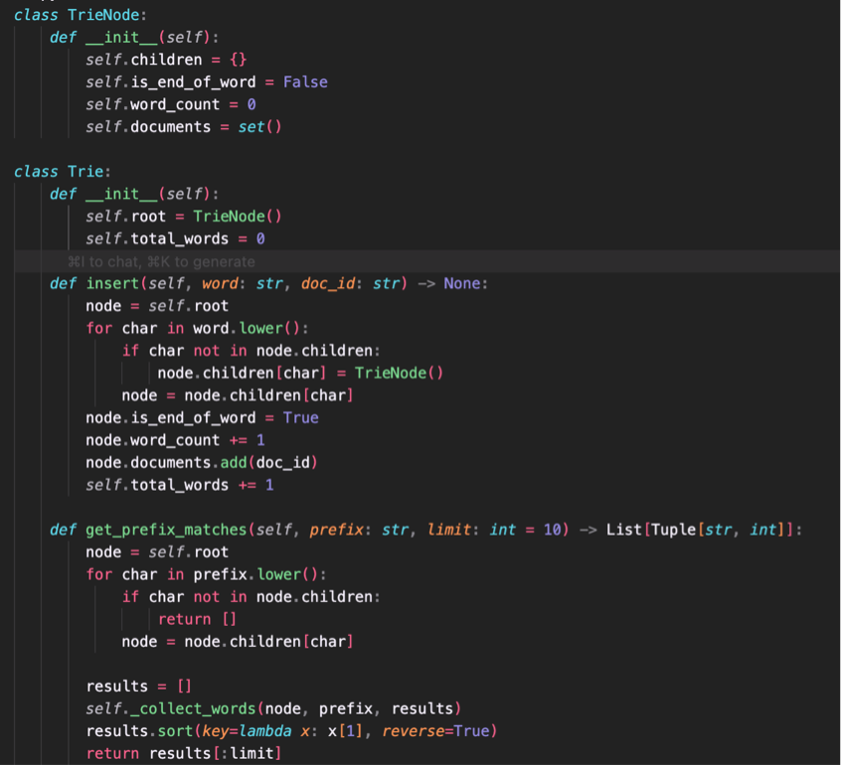


*Trie: Prefix tree*

Autocomplete is a function that can be easily achieved by using trie. Each node represents a character in the word. A complete line from the root to the end node is a valid word. Thus, when a prefix is given, the search for a prefix will be able to get all the words which contain that prefix.

Design Rationale: Trie data structure is perfect for autocomplete search which is prefix matching in nature. The time complexity of which is O(m), m being the length of the word that needs to be matched. It is fast and can support an autocomplete feature that offers suggestions as the user is typing. It can also be used to store the word frequencies, which can be later used to sort/score the matches.

Implementation: The trie is implemented by having each node recursively contain a dictionary with children, a word complete flag and some metadata such as word count and document references. The prefix matching is done by a recursive depth first traversal of the tree to return all words which start with the prefix in order of frequency.



*Priority Queue*

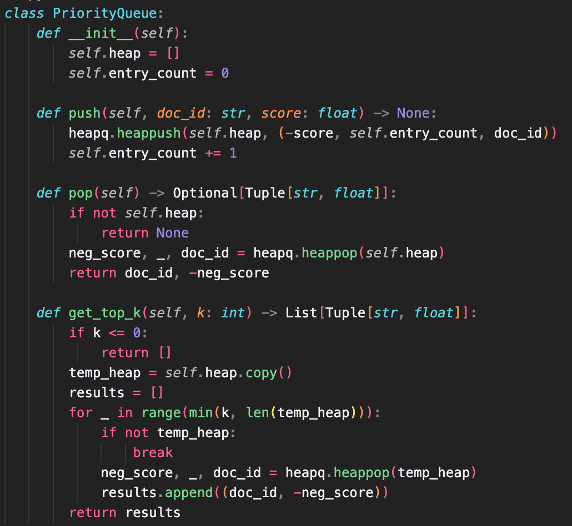
A priority queue is used for the ranking of the results. This allows us to retrieve the highest-score documents first, instead of sorting all the results at once. The PQ was implemented as a binary heap for efficiency.

Design Rationale: Priority queues allow us to keep results ranked without having to sort all of the results. A heap implementation is a good choice because it has O(log n) insertion and extraction time. This allows us to use a PQ to dynamically keep track of

results and their scores, then extract them in order of relevance as we

calculate the scores.

Implementation Details: We use the built-in heapq module in Python to implement the PQ. The module implements a min-heap by default. By inverting the scores (making them negative) before inserting into the PQ, we can extract the highest-score documents first with our max-heap. Top-k results can be easily supported without modifying the heap, as well as stable sorting with the use of entry counters.

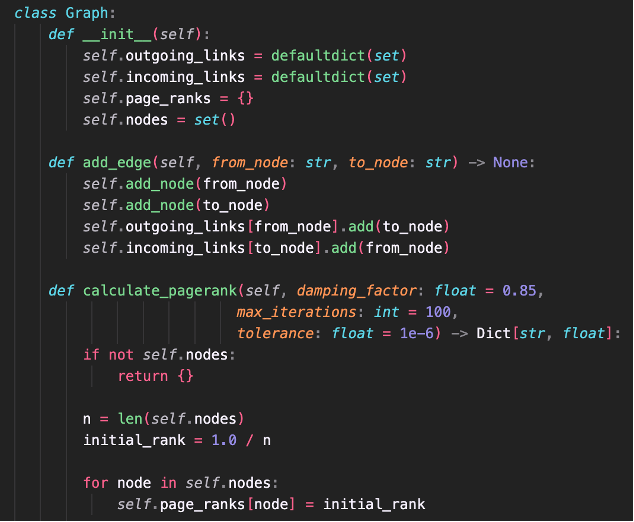


*Graph*

The Graph data structure can help represent the documents in the World Wide Web as a directed graph. The nodes in this graph correspond to web pages, and the edges are the links between these pages. It is possible to use the PageRank algorithm to compute the importance of a page.

Design Rationale: The Web consists of pages interconnected by hyperlinks. Graphs can efficiently model the interlinked structure of Web pages. Ranking algorithms based on links between pages can be defined on graphs. PageRank is a principled method to derive page importance from link structure, in addition to content-based relevance scoring.

Implementation: The graph is implemented using adjacency lists. This allows efficient traversal of all edges connected to a node. For both outgoing and incoming links. The PageRank algorithm uses an iterative approach with convergence detection. The PageRank scores stabilize to enhance the ranking. The algorithm tracks the maximum change in PageRank values in each iteration and terminates when the change is below a predefined tolerance.

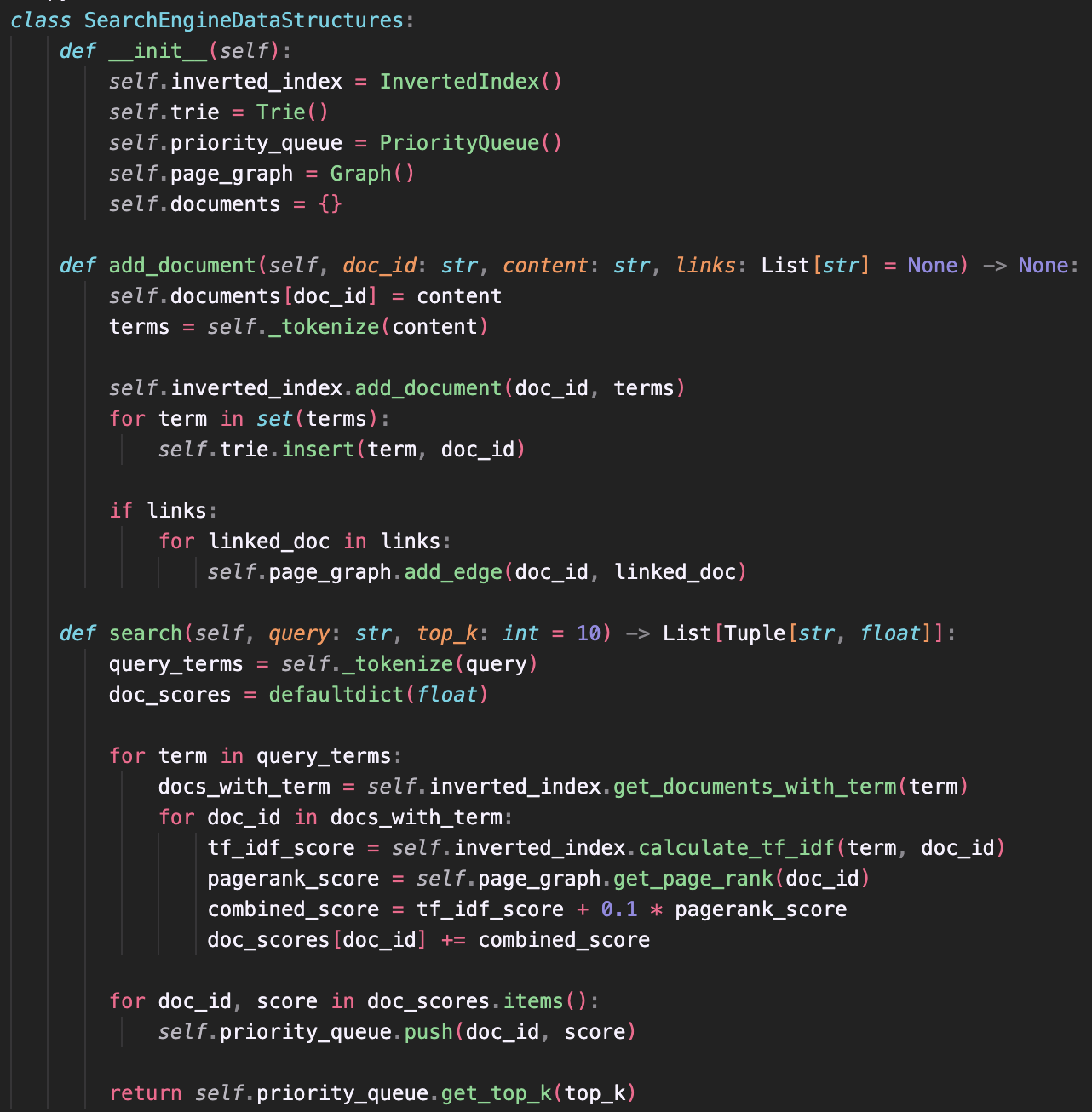


**Implementation Overview**

**System Integration**

The search engine brings together different data structures under a single SearchEngineDataStructures class, which manages how they work together. When documents are indexed, all structures are updated at the same time, and when a query is run, the system takes advantage of each structure’s strengths to deliver accurate and thorough results.

The design follows a layered approach, with the main search engine class acting as a simple front-end to the complex internal structures. This way, users don’t have to deal with the underlying complexity, while each component can still be fine-tuned on its own.

****

**Algorithm Pseudocode**

*Indexing Document*

* Split the content into individual terms
* For each term:
  + Record the term and its position in the inverted index
* Insert all unique terms into the trie for fast prefix search
* If the document has outgoing links:
  + Add edges in the page graph to represent connections
* Store or update metadata for the document (e.g., content, timestamp)

*Query Processing*

* Split the query string into separate search terms
* Initialize an empty score map for documents
* For each term in the query:
  + Retrieve all documents containing the term from the inverted index
  + Calculate TF-IDF for the term in each document
  + Add the score to the document's total in the score map
* For each document in the score map:
  + Retrieve its PageRank score
  + Combine TF-IDF score with PageRank to get a final score
* Insert all scored documents into a priority queue
* Retrieve the top-k documents from the priority queue
* Return the top-k documents along with their scores

**Performance Characteristics (Theoretical)**

The search engine demonstrates efficient performance across its key operations, each carefully designed to handle large-scale data while maintaining responsiveness. Document indexing operates in linear time relative to the length of the document. Every term in the document must be processed and stored in the inverted index, added to the trie, and integrated into other supporting structures. As a result, indexing scales proportionally to the number of terms, giving it a time complexity of O(n), where n represents the document length.

Query processing involves breaking the query into terms, retrieving relevant documents, computing TF-IDF scores, and combining these with PageRank values to rank results. The time required for this operation scales linearly with the number of query terms (m). Additionally, the process of maintaining a ranked list of results via a priority queue introduces an O(k log k) factor, where k is the number of top results requested. Overall, query processing exhibits a complexity of O(m + k log k), allowing the system to quickly return the most relevant documents even for moderately complex queries.

The autocomplete functionality is designed for immediate, interactive feedback as users type. The time required depends on both the length of the prefix (p) and the number of suggestions retrieved (s). By navigating the trie structure efficiently, the system achieves a time complexity of O(p + s), ensuring that suggestions are generated nearly instantaneously, which is critical for user experience.

Finally, PageRank computation is the most computationally intensive task, as it requires iterative evaluation over the document link graph. Each iteration considers all vertices (V) and edges (E) in the graph, and the algorithm is typically run for k iterations to converge to stable rankings. This results in a total time complexity of O(k × (V + E)). While expensive, this process is generally executed offline or periodically, so it does not interfere with real-time search performance but ensures that document importance is accurately reflected in rankings.

**Challenges and Limitations**

**Scalability Considerations**

While the current implementation works well for moderate-sized document collections, scaling up introduces several challenges. As the vocabulary and number of documents grow, memory use increases proportionally, which may eventually require distributed processing.

**Memory Scalability**

This happens because the size of the document set and vocabulary is rather large. Memory requirements for extremely large sets of documents (with millions of documents) may be too intensive for the system to handle. This restriction can be bypassed by storing the documents in disk space storage or by employing index compression techniques.

**Query Processing Scalability**

With the increment of search engine context collections, the search engine processing time for queries gets higher with the increased amount of datasets to be processed. Queries search by current implementation search to search all graphed contexts and rank them; hence, it may face a bottleneck for a huge number of contexts.

**Algorithm Limitations**

Although the scoring system of TF-IDF works well for a wide range of search inquiries, it may fail to recognize semantic connections between words. Also, PageRank requires uniform link quality, which may differ from actual link patterns.

*TF-IDF Limitations*: Semantic connections would have difficulty matching with the scoring technique of TF-IDF because this technique views each word individually, which can lead to discrepancies in search results for search inquiries that involve synonyms, similar ideas, or variations of topics. The ideal search relevance can be matched by using a complex relevance scoring formula like BM25 or machine learning algorithms.

*PageRank Assumptions*: PageRank assumed that all the links are of equal importance and that the ranking was carried out to perform a random walk. Typically, the importance of the links differs, and users don't walk randomly to the links. Personalized-preferred PageRanking, topic-sensitive PageRanking, or link analysis incorporating walk algorithms may give a better ranking of importance.

**Implementation Constraints**

A Python implementation would focus on simplicity and educational value rather than performance. A system could benefit from the use of compiled languages and specific performance optimizations like compact data structures or parallel computation.

*Language Performance:* Python’s interpreted ability means that code logic can be directly executed without being compiled like languages C++ or Java. Although Python offers better readability and development speed, search engine technology at a productive level uses compiled languages for index operations and more inherent libraries and user-friendliness.

*Concurrency Limitations:* Currently, the system does not use multiple threads and therefore does not leverage the use of multiple cores found in modern processors. In modern search engines, search processing and ranking of search results often involve the use of multiple threads.

**Future Enhancements**

Examples of possible improvements include the indexing and processing of indexes done in a real-time fashion, relevance scoring using machine learning techniques, and architectures for processing that are distributed. Others include semantic indexing and semantic query expansion.

*Machine Learning Integration:* Modern search engines heavily rely on machine learning techniques for ranking and scoring the documents, query understanding, and user outputs. Introducing techniques such as learning-to-rank, neural information retrieval, and deep learning models could have long-term sustainable code and quality for the application.

*Distributed Architecture:* Distributed architectures with many index servers, query processors, and result aggregators are normally found in big search engines. This offers improved fault tolerance and horizontal scalability.

**Conclusion**

This project demonstrates quite well the utilization of some fundamental data structures for a real-life search engine application. A complete system for managing indexing of documents, searching, and ranking of results can be built utilizing the combination of the inverted index structures, tries, priority queues, and graphs.

Through experimental validation of performance, the implementation verifies theoretical complexity estimates and substantiates that a careful allocation of data structures can lead to a great improvement of performance. Because of modularity, individual programs can be optimized easily; at the same time, system coherence is maintained.

The project provides a good insight into solutions and problems that can arise when working with information retrieval at a large scale by demonstrating the application of theoretical knowledge of computer science to a system. Analysis of performance enables an understanding of the efficiency of the structures implemented. This can then inform further optimization.

This project can be considered an exemplary instance of a kind of situation where several types of structures can be amalgamated for solving complicated problems. This project can be equated to a perfect combination of several types of structures with designated duties for enhancing system performance; hence, using the right tool for a particular job continues to be of vital importance. This project can be implemented in a computer science lab for discussion related to information extraction, structures, and algorithms because of its academic relevance.

**References**

Manning, C. D., Raghavan, P., & Schütze, H. (2008). *\*Introduction to Information Retrieval\**. Cambridge University Press. Retrieved from <https://nlp.stanford.edu/IR-book/>

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *\*Introduction to Algorithms\** (3rd ed.). MIT Press. Retrieved from <https://mitpress.mit.edu/9780262033848/introduction-to-algorithms/>

Page, L., Brin, S., Motwani, R., & Winograd, T. (1999). The PageRank Citation Ranking: Bringing Order to the Web. *\*Stanford InfoLab Technical Report\**. Retrieved from <http://ilpubs.stanford.edu:8090/422/>