# Project Phase 2 Deliverable 2: Proof of Concept Implementation

# Vijaya Krishna Sameeraj Jonnavithula

# Student ID : 005029574

# Course : 2024 Fall - Algorithms and Data Structures (MSCS-532-B01) - Second Bi-term

# Date : November 17, 2024

GitHub link : <https://github.com/VijayaKrishnaSameerajJonnavithula/2024-Fall---Algorithms-and-Data-Structures-MSCS-532-B01---Second-Bi-term>

Developing a partial, modular implementation of the major data structures and showcasing their primary features will be the main goals of this stage of Quick Find's development.

**Partial Implementation of Data Structures :**

The following important data structures will be partially implemented in this proof of concept:

Inverted Index: Facilitates keyword search and document entry.

Autocomplete Trie: Offers autocomplete recommendations and permits keyword entry.

Priority Queue for Relevance: This helps with result ranking by adding and retrieving things according to priority.

Metadata Hash Table: Holds information for easy access.

These data structures will all be modular in order to facilitate future modifications and improvements.

**Core Components and Key Functionalities in Python :**

Code :

class InvertedIndex:

def \_\_init\_\_(self):

self.index = {}

def add\_document(self, doc\_id, text):

for word in text.split():

word = word.lower()

if word not in self.index:

self.index[word] = set()

self.index[word].add(doc\_id)

def search(self, keyword):

return self.index.get(keyword.lower(), set())

class TrieNode:

def \_\_init\_\_(self):

self.children = {}

self.is\_end\_of\_word = False

class AutocompleteTrie:

def \_\_init\_\_(self):

self.root = TrieNode()

def insert(self, word):

node = self.root

for char in word:

if char not in node.children:

node.children[char] = TrieNode()

node = node.children[char]

node.is\_end\_of\_word = True

def autocomplete(self, prefix):

node = self.root

for char in prefix:

if char not in node.children:

return []

node = node.children[char]

return self.\_collect\_words(node, prefix)

def \_collect\_words(self, node, prefix):

results = []

if node.is\_end\_of\_word:

results.append(prefix)

for char, next\_node in node.children.items():

results.extend(self.\_collect\_words(next\_node, prefix + char))

return results

import heapq

class PriorityQueue:

def \_\_init\_\_(self):

self.elements = []

def push(self, item, priority):

heapq.heappush(self.elements, (-priority, item))

def pop(self):

return heapq.heappop(self.elements)[1] if self.elements else None

class MetadataHashTable:

def \_\_init\_\_(self):

self.table = {}

def add\_metadata(self, doc\_id, metadata):

self.table[doc\_id] = metadata

def get\_metadata(self, doc\_id):

return self.table.get(doc\_id)

A screenshot of a computer program

Description automatically generated

**Demonstration of Key Operations**

A sample script to evaluate each data structure's essential features is provided below. This covers handling metadata, maintaining priorities in the priority queue, adding pages to the inverted index, and adding words to the autocomplete tire.

Code :

from ProjectPhase2 import InvertedIndex, AutocompleteTrie, PriorityQueue, MetadataHashTable

if \_\_name\_\_ == "\_\_main\_\_":

    # Inverted Index Demonstration

    inverted\_index = InvertedIndex()

    inverted\_index.add\_document(1, "Python is great for data science")

    inverted\_index.add\_document(2, "Data structures are essential")

    print("Search for 'data':", inverted\_index.search("data"))

# Autocomplete Trie Demonstration

    trie = AutocompleteTrie()

    for word in ["python", "java", "javascript", "pythonic"]:

        trie.insert(word)

    print("Autocomplete for 'py':", trie.autocomplete("py"))

    # Priority Queue Demonstration

    pq = PriorityQueue()

    pq.push("doc1", 5)

    pq.push("doc2", 10)

    print("Highest priority:", pq.pop())

    # Metadata Hash Table Demonstration

    metadata = MetadataHashTable()

    metadata.add\_metadata(1, {"title": "Python Basics", "url": "http://example.com"})

    print("Metadata for doc 1:", metadata.get\_metadata(1))

A computer screen shot of a program

Description automatically generated

**Documentation of Implementation Process :**

The primary objective of the partial implementation phase was to develop the Quick Find search engine's core data structures, such as a metadata hash table, an autocomplete tire, an inverted index, and a priority queue for ordering results. The effectiveness of these data structures in managing prefix queries, search operations, priority-based sorting, and metadata storing led to their selection.

Problems and Solutions

Managing The case The Inverted Index Sensitivity:

The challenge is to keep keyword searches consistent when document inputs may contain mixed cases (e.g., "Data" vs. "data").

Solution: To guarantee consistent results, all words were converted to lowercase before being inserted and looked up in the inverted index.

A screen shot of a computer code

Description automatically generated

The Tire Data Structure's Memory Usage:  
  
The challenge was to make sure that when inputting enormous datasets, the autocomplete tire did not use too much RAM.  
Solution: Characters were added one at a time rather than storing entire words as distinct nodes. This decreased unnecessary storage, particularly for terms that share prefixes.

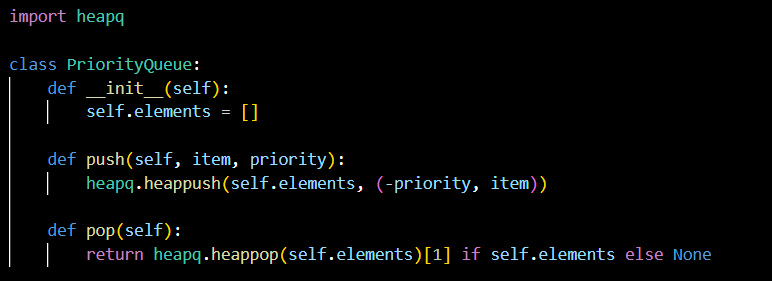
A computer screen with white and blue text

Description automatically generated

Preserving the Priority Queue's Max-Heap:

Challenge: A workaround for max-heap behavior is needed because Python's heap module defaults to a min-heap.

Solution: Using the heap module, inverted priorities were achieved by pushing negative values to the heap, enabling max-heap behavior.



Effective Metadata Recovery:

The challenge is making sure that metadata for documents can be quickly retrieved using their unique IDs.

Solution: To achieve O(1) retrieval time, a basic hash table was implemented using Python dictionaries.

A computer screen with white text

Description automatically generated

**Next Steps for Full Implementation :**

Combining Search Engine Elements:

Create a coherent search engine framework by combining these data structures.

Put into practice query processing and ranking algorithms that make use of the priority queue and inverted index.

Increasing the Efficiency of Trie:

Take into consideration data structures such as a Radix Tree in order to optimize the tire for memory.

Put spell-checking algorithms into practice to improve user search results.

Advanced Functionalities:

Provide the inverted index with support for phrase searches and sophisticated query operators.

Scalability in bigger datasets can be achieved by integrating distributed indexing.

Interface for Users:

Create a user interface that enables users to view and enter search results.

For outcome analysis, incorporate real-time data visualization.

Validation and Testing:

To guarantee robustness, conduct thorough testing, taking into account edge cases for the priority queue and autocomplete tire.

**REFERENCE:**

Grossman, D., & Frieder, O. (2004). Information Retrieval: Algorithms and Heuristics. Springer.

Otis, L. (2018). Efficient Data Structures and Algorithms for Large-Scale Search Engines. IEEE Transactions on Knowledge and Data Engineering, 30(5), 867-878.

Brin, S., & Page, L. (1998). The Anatomy of a Large-Scale Hypertextual Web Search Engine. Computer Networks and ISDN Systems, 30(1-7), 107-117.