CosmoSurf Search Engine Report

Team : CosmoSurf
Members : Pooja, Shreyas, Nikhil
Website: https://nickhegde.github.io/cosmosurf-team/

Introduction:

This report describes the architecture and design of the "KWIC Lexical Sort System". The system takes a given sentence, generates all possible circular shifts of the sentence, and then sorts them lexicographically. The implementation is based on a simple class design with encapsulated logic for generating shifts and sorting.

System Overview:

The system implements a 'Key Word in Context (KWIC)' system with a focus on circular shifts and lexicographical sorting. The circular shifts involve shifting words in the input sentence and generating all possible variations, where the first word moves to the end in each shift. After generating these shifts, the system sorts them in lexicographical order and displays the results.

Core Components:

The system comprises the following core components:

1. KWICLexicalSort Class : The main class that encapsulates the logic for generating circular shifts and sorting.
2. KWICEntry Class : A nested class used to represent each circular shift as an object. This class implements `Comparable` to enable sorting based on lexicographical order.
3. Main Method: The entry point of the program, where an input sentence is provided, shifts are generated, sorted, and printed.

Key Design Concepts:

KWICEntry Class:

Encapsulation of Shifted Sentence : The `KWICEntry` class represents each shifted sentence. It encapsulates the string (`shiftedSentence`) representing the shift and overrides the `compareTo` method to allow lexicographical sorting.
Sorting Mechanism : By implementing the `Comparable<KWICEntry>` interface, the `compareTo` method compares the `shiftedSentence` of each `KWICEntry`, enabling natural ordering.

Circular Shift Generation:

Shifting Logic : The `generateCircularShifts` method takes a sentence, splits it into words, and systematically generates circular shifts by reordering the words. It appends each circular shift to a list of `KWICEntry` objects.

Modular Arithmetic: The circular nature of the shifts is achieved using modular arithmetic (`j % words.length`) to loop back to the beginning of the word array when necessary.

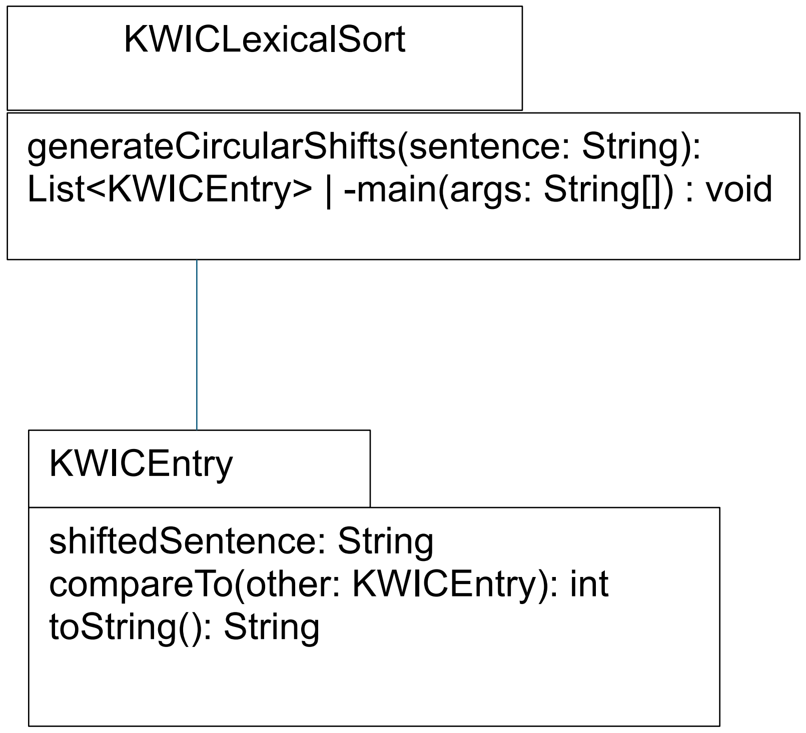
Lexicographical Sorting :

Sorting with Collections.sort() : Once all circular shifts are generated, the system uses `Collections.sort()` to sort the list of `KWICEntry` objects. Since `KWICEntry` implements `Comparable`, this sort is lexicographically based on the shifted sentences.

Sequence of Operations:

1. The user inputs a sentence (`inputSentence`).
2. The `generateCircularShifts` method splits the sentence into words and generates all possible circular shifts.
3. Each shift is stored as a `KWICEntry` object in a list.
4. The system outputs all generated circular shifts.
5. The list is then sorted lexicographically using the `Collections.sort()` method.
6. The sorted list is displayed as the final output.

Class Diagram:


KWICLexicalSort
generateCircularShifts(sentence: String): List<KWICEntry> | -main(args: String[]) : void
KWICEntry
shiftedSentence: String
compareTo(other: KWICEntry): int
toString(): String

Explanation of Class Diagram :

KWICLexicalSort Class :
1. Contains the `generateCircularShifts()` method that generates circular shifts from the input sentence.
2. The `main()` method serves as the entry point of the system.

KWICEntry Class :
1. Holds a single circularly shifted sentence (`shiftedSentence`).
2. Implements the `Comparable<KWICEntry>` interface to define the natural ordering for lexicographical sorting.
3. The `toString()` method is used to display each circular shift when printed.

Flow of Execution:

1. Initialization : The input sentence "Web browsers display web pages" is passed to the `generateCircularShifts()` method.

2. Shift Generation : For each word in the sentence, a new circular shift is created by rotating the words in the sentence. This continues until all possible shifts are generated.
3. Sorting : The list of generated circular shifts is passed to `Collections.sort()`, which sorts the shifts based on their lexicographical order.
4. Output : The system outputs both the unsorted and sorted lists of circular shifts.

Case-Sensitive Search:

Both the **Database** and **Search** modules store and retrieve content exactly as inputted. Searches are performed with case sensitivity, retrieving results only if the exact case matches the user's query.

Hyperlink Enforcement:

The **UserOutput** module displays URLs as clickable links, using HTML <a> tags in a web-based frontend to open URLs in a new tab, enhancing usability.

Boolean Search (OR/AND/NOT):

The **Search** module parses queries to allow:

- **OR**: Retrieves URLs with any of the keywords.
- **AND**: Retrieves URLs with all specified keywords.
- **NOT**: Excludes URLs with specific keywords. The **Database** filters results accordingly.

Multi-Engine Search:

The **Search** module integrates multiple search engines or APIs (e.g., Elasticsearch, Google Search API), running them asynchronously to gather comprehensive results.

Deletion of Outdated URLs:

A cleanup routine in the **Database** module identifies URLs that are no longer accessible or have not been accessed recently, optimizing storage by removing irrelevant entries.

Customizable Sorting:

The **Search** and **UserOutput** modules offer multiple sorting options:

- **Alphabetical Order**
- **Frequency of Access**
- **Payment Priority**: Sponsored URLs appear first. This allows users to view results in a preferred order.

Pagination and Results Per Page:

The **UserOutput** module supports pagination, allowing users to set the number of results displayed and move between pages for a better user experience.

Autofill and Typo Correction:

Requirement: Provide autofill suggestions and correct minor typos in search queries.

Implementation:
The **Search** module offers real-time suggestions and typo correction using fuzzy matching techniques and libraries like ElasticSearch or Google Suggestions API, helping users input accurate searches.

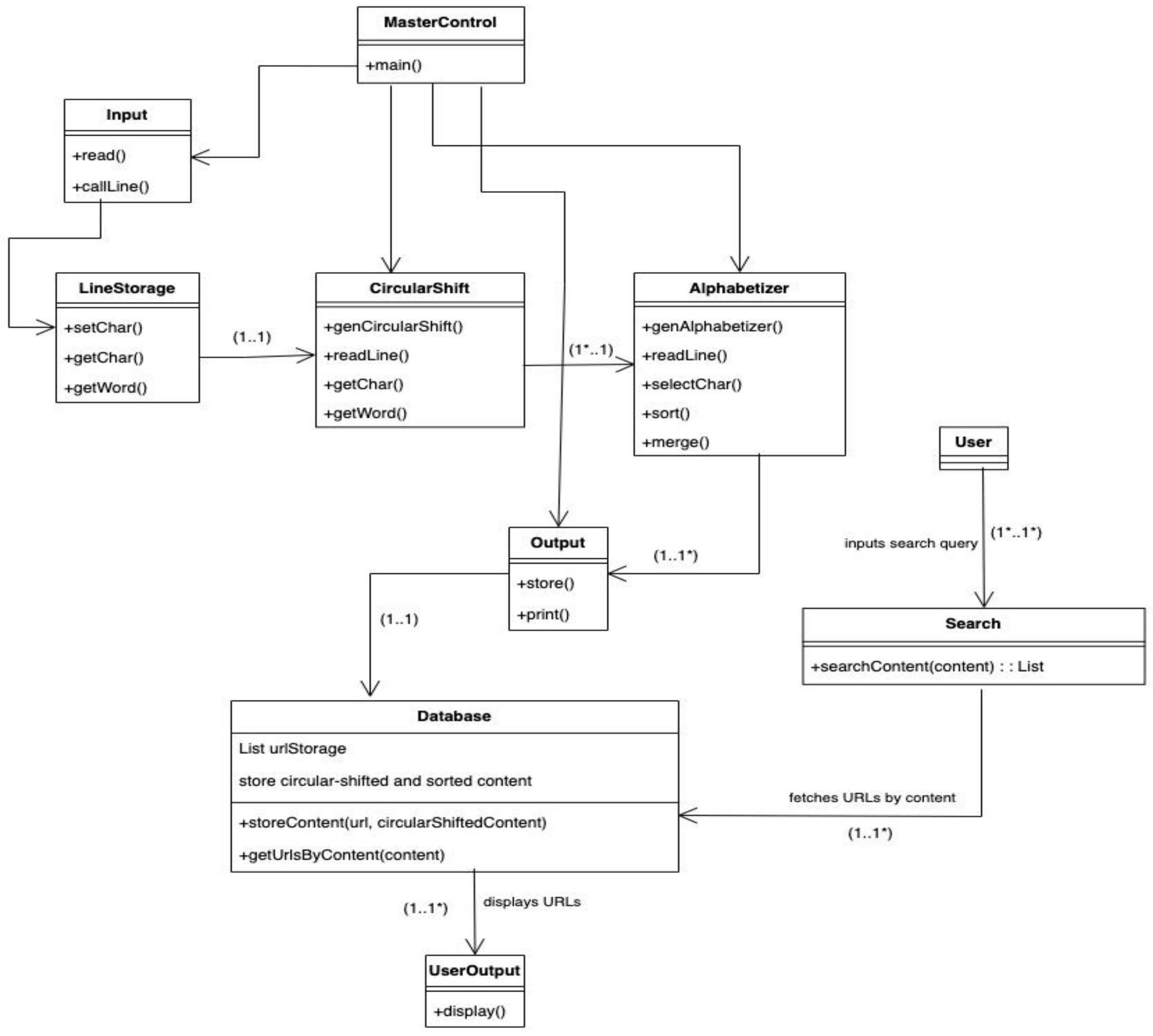
Symbol Filtering:

The **NoiseEliminator** module filters out unwanted symbols and words, making queries cleaner and results more relevant.

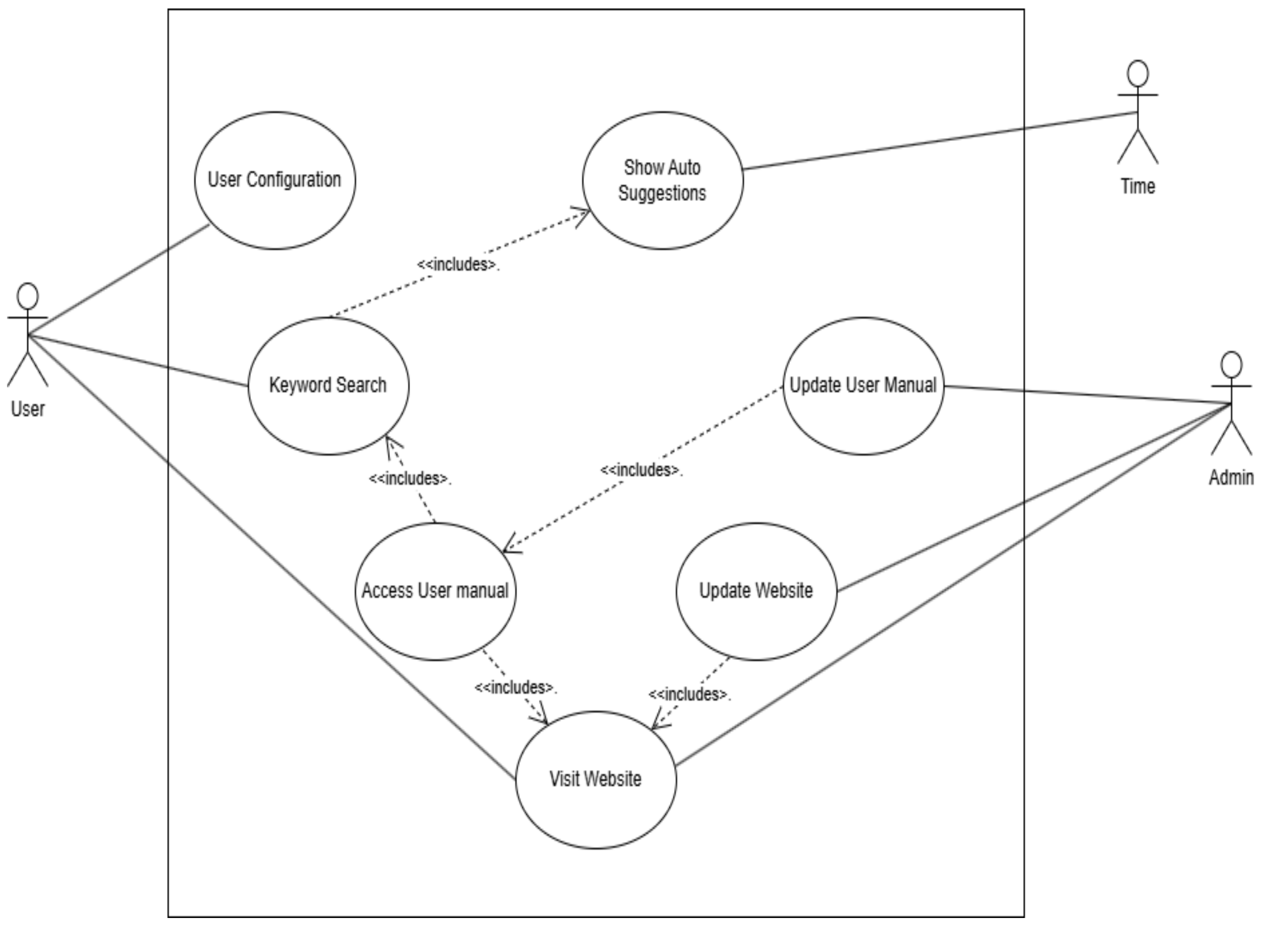
Potential Extensions:

1. Support for Multiple Sentences : The system could be extended to handle multiple sentences, storing shifts for multiple input sentences.
2. Different Sorting Criteria : In the future, the system could be extended to allow different sorting criteria, such as sorting based on word length or frequency of words.
3. File Input/Output : The system could be adapted to read input sentences from a file and write the sorted shifts to a file.

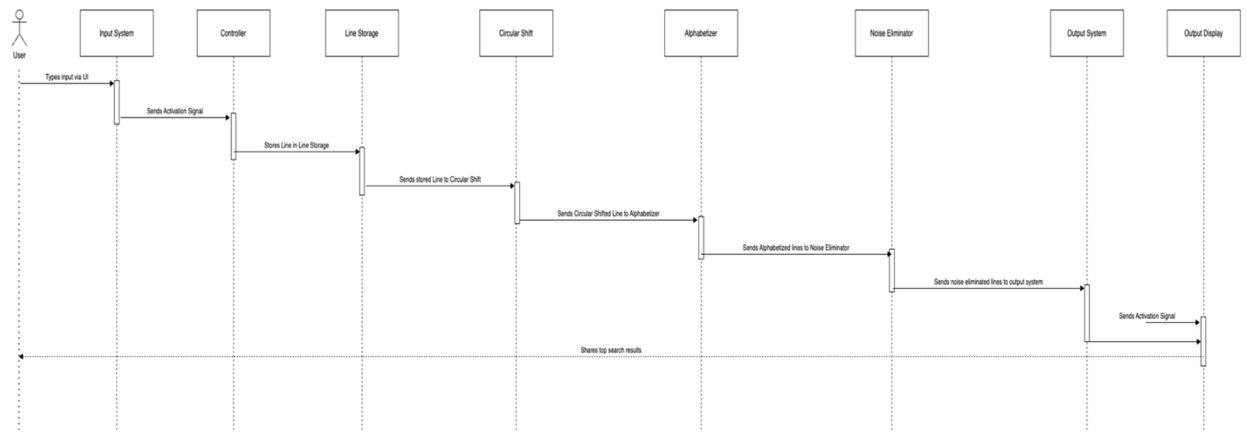
Class Diagram:



Use Case Diagram:



Sequence Diagram:


User
Input System
Controller
Line Storage
Circular Shift
Alphabetizer
Noise Eliminator
Output System
Output Display
Types input via UI
Sends Activation Signal
Stores Line in Line Storage
Sends stored Line to Circular Shift
Sends Circular Shifted Line to Alphabetizer
Sends Alphabetized lines to Noise Eliminator
Sends noise eliminated lines to output system
Sends Activation Signal
Shares top search results

Conclusion:

The KWIC Lexical Sort System efficiently generates and sorts circular shifts lexicographically using a modular design centered on the **KWICLexicalSort** and **KWICEntry** classes. This architecture is enhanced with features such as case-sensitive search, hyperlink support, Boolean operators, multi-engine search, URL deletion, sorting options, pagination, and symbol filtering, making it user-friendly and adaptable.

The system is designed for scalability, with potential extensions like support for multiple sentences, alternative sorting criteria, and file I/O. Overall, it provides a solid foundation for flexible and extendable text processing applications.