COMP 8547 Advanced Computing Concepts

Fall 2023

Final Project Report

Variant 5 – Hotel Price Analysis

School of Computer Science, University of Windsor

Team – Fantastic Five

Members:

- **Subhram Satyajeet – 110127932**
- **Kashyap Bharatbhai Prajapati – 110126934**
- **Sachreet Kaur – 110122441**
- **Aditya Kunal Bhate – 110125765**
- **Samiksha Arora – 110128455**

Contents

1. Objective:

The idea of the project is to analyze and understand hotel prices to get a better deal when booking tickets near the travel date specified. As per the instructions provided, we were supposed to crawl 3 hotel websites, collect data, and present the user with the best deals based on the parameters provided. Throughout the project, several data structures and algorithms had to be implemented to present the user with the best deals. The overall objective of this project was to understand the usage of various data structures and algorithms and their applications in the real world.

2. Features, Significance, and Individual Contribution

2.1. Web Scrapping/Crawler

A web crawler, also called a web spider or web bot, acts as a vital tool utilized by search engines such as Google, Bing, and others to systematically explore and index web pages scattered across the internet. Its primary functions include:

- **Indexing Content:** Web crawlers traverse websites methodically, scrutinizing the content within and indexing the information uncovered. This process significantly aids search engines in furnishing pertinent results to users when they search for specific queries.
- **Data Collection:** These crawlers gather diverse data types like text, images, links, metadata, and more from web pages. This amassed data serves as a valuable resource utilized by search engines and various applications for analysis, ranking, and other purposes.
- **Monitoring Changes:** With their ability to monitor alterations on websites, web crawlers excel in detecting new pages, updates in content, or broken links. This feature provides invaluable insights for website owners and marketers, enabling them to uphold and enhance the quality of their websites.

These capabilities make web crawlers instrumental in maintaining the accuracy, relevance, and currency of online information, benefiting both users and website stakeholders.

2.2. HTML Parsing

HTML parsing in Java involves extracting specific information or elements from an HTML document. It's commonly used to scrape data from web pages, analyze their content, or perform various operations on the HTML structure.

Java offers several libraries for HTML parsing, and one of the popular choices is Jsoup. Jsoup simplifies the process of parsing HTML by providing methods to fetch and manipulate HTML content.

Here's a step-by-step explanation of HTML parsing using Jsoup in Java:

- **Understanding Jsoup:** Jsoup serves as a helpful tool in Java for dealing with website content written in HTML. It's like a smart assistant that knows how to read and handle HTML information, making it simpler for programmers to work with web page content.
- **Fetching HTML Content:** Imagine you're asking Jsoup to check out a webpage. It goes to that webpage and looks at the code behind it, much like peeking into the engine of a car to see how it works. This code contains all the information displayed on the webpage.
- **Parsing HTML:** Jsoup isn't satisfied with just looking at the code; it takes this code and organizes it neatly. Think of it as sorting a messy drawer full of various items into distinct compartments. This organized structure is called a "Document" in Jsoup's language.
- **Extracting Specific Elements:** Once Jsoup has neatly organized the web page's code into a Document, it's easier to pinpoint specific parts. For instance, if you want to grab all the links present on that webpage, Jsoup knows precisely how to find and pull out these links for you.
- **Manipulating and Traversing Elements:** Jsoup offers some neat tricks once you've found the needed elements. It's like having tools to tweak, modify, or move things around within that organized drawer. With Jsoup, you can change attributes, grab text, or navigate through the webpage structure smoothly.
- **Using Extracted Data:** Once you've gathered the required information, it's like having a collection of useful items from that organized drawer. You can display this content, study it further, or even use it in other projects - making the information work for you in different ways.

In essence, Jsoup acts as a friend who helps Java developers work effectively with website content. It simplifies the process of accessing, organizing, and using data from web pages.

2.3. Word completion/suggestion(Trie Tree)

Word completion and suggestion in Java refer to the process of predicting or offering possible words or phrases based on the input provided by a user. This functionality is commonly employed in various applications, including text editors, search engines, messaging apps, and autocomplete features in forms or search bars.

- **Word Completion:** It involves suggesting or completing words or phrases as a user types. For instance, as you start typing a word or phrase, the application predicts and

displays potential completions based on what has been typed so far. It helps users save time by offering options that match or closely resemble what they intend to write.

- **Word Suggestion:** This feature provides suggestions for words or phrases a user might be looking for. It analyzes the input context or previous words and suggests likely options. For example, when composing an email, the application might suggest phrases or words that are commonly used in professional email writing.
- **Enhanced User Experience:** It significantly improves the user experience by reducing the effort required to input text. Users can complete their sentences faster, especially when dealing with lengthy or commonly used phrases.
- **Improved Efficiency:** In applications like search engines or text editors, word completion and suggestion speed up the process of finding information or composing messages/documents.
- **Error Prevention:** It assists in preventing spelling errors or typos by suggesting correctly spelled or commonly used words or phrases.

Implementation in Java:

- In Java, word completion and suggestion can be implemented using algorithms like Trie (prefix tree), Levenshtein distance, or other approaches that analyze the text input and suggest relevant words or phrases.
- Libraries or custom implementations can be used to manage and provide word suggestions based on user input or contextual information.
- User interfaces or components can be designed to display these suggestions or completions as the user types, providing a seamless and efficient interaction.

Overall, word completion and suggestion in Java applications aim to streamline the user input process, making it faster, more accurate, and user-friendly.

2.4.Edit Distance

The edit distance, or Levenshtein distance, is like a measuring tape for how alike or different two words are. It counts the smallest number of changes needed to turn one word into another - like adding, removing, or changing just one letter at a time.

- **Spell Checking:** Ever had a little red line under a misspelled word? That's the edit distance at work! It helps spell checkers by suggesting the closest correctly spelled words based on how similar they are to what you typed.
- **DNA Sequencing:** Think of it as comparing two DNA strands - it's used in biology to figure out how similar or different they are. This is crucial in understanding genetic codes and finding links between different species or individuals.

- **Information Hunting:** When you type something in a search bar, the edit distance is quietly working in the background, finding documents or words that are most similar to what you're looking for. It helps search engines be smarter in fetching relevant results.

So, in a nutshell, the edit distance is like a handy tool used in various fields - from catching typos to uncovering genetic similarities or helping search engines find just what you're searching for.

2.5. Search Frequency

Search frequency refers to the number of times a particular term or query appears within a dataset, such as a collection of documents, a database, or a set of user interactions. It's a metric used to understand the popularity or relevance of a specific keyword or phrase within a given context.

Implementation of Search Frequency:

- **Data Source Selection:** Choose the dataset or source where you want to analyze search frequencies. It could be a text file, a collection of documents, user logs, or any data containing text.
- **Read and Load Data:** Use Java's file handling (FileReader, BufferedReader) or database connections (JDBC) to access and load the dataset into your Java program.
- **Tokenization:** Tokenize the text data into words, phrases, or tokens. Split the text into individual words or phrases to facilitate counting.
- **Preprocessing:** Preprocess the text if needed, such as converting everything to lowercase, removing punctuation, or handling special characters. This ensures consistency in counting words.
- **Counting Word Occurrences:**
    - Use a data structure like a HashMap<String, Integer> to store words and their frequencies.
    - Iterate through the tokens and update the frequency count for each word encountered in the dataset.
- **Display or Use Results:** Optionally, display the word frequencies or use them for further analysis, search ranking, or any other relevant purpose.

2.6. Inverted Indexing

Inverted indexing is a technique used in information retrieval to quickly locate and access specific data within a large collection, such as a database or a set of documents. It's commonly

employed by search engines to efficiently find relevant documents containing particular words or phrases. Implementing inverted indexing in Java involves creating a data structure that maps words to the documents where they appear.

Use of Inverted Indexing:

- Efficient Information Retrieval: Inverted indexing enables rapid searching of documents containing specific words. Instead of scanning through all documents, it directly points to the documents containing the queried terms.
- Improved Search Performance: Search engines utilize inverted indexes to speed up searches. When a user enters a query, the search engine quickly references the index to retrieve relevant documents.
- Support for Boolean Queries: It facilitates complex search queries involving boolean operations (AND, OR, NOT) by combining indexes for multiple terms.

Implementing inverted indexing involves these key steps:

- Data Collection: Gather the dataset or documents you want to index. This could be text files, a collection of web pages, or any structured data.
- Tokenization and Preprocessing: Tokenize the text by breaking it into words or phrases and preprocess it (lowercasing, removing punctuation, etc.).
- Building the Inverted Index:
    - Create a data structure (often a HashMap or HashMap<String, Set<Integer>>) to map words to document IDs or positions.
    - For each word in the documents, update the index by recording where the word appears (document ID, position, etc.).
- Searching with the Index: When a query arrives, tokenize it and look up the index to find documents containing those terms.

2.7. Page Ranking

Page ranking is a method used by search engines to assign a relevance score to web pages based on various factors. It determines the importance of a webpage within the context of a query entered by a user. Implementing page ranking in Java involves algorithms such as PageRank that analyze links between web pages to determine their importance and relevance.

Uses of Page Ranking:

- Search Engine Result Ordering: Page ranking helps search engines display the most relevant and authoritative pages at the top of search results. Pages with higher rankings are deemed more relevant to a user's query.
- Prioritizing Information: It aids in prioritizing information, making it easier for users to find high-quality content, reducing the time required to sift through less relevant pages.
- Determining PageAuthority: Page ranking also contributes to assessing a webpage's authority within a specific domain or subject area.

Here's an overview of the implementation:

- Graph Representation: Represent web pages as nodes and their links as edges in a graph structure. Each page is a node, and links between pages are edges.
- Initial Scores: Assign initial scores or weights to each page in the graph. All pages might start with the same score initially.
- Iterative Calculation: Use an iterative algorithm (like the PageRank algorithm) to update the scores of web pages based on the scores of linking pages. Iterate through the graph, recalculating scores based on linking pages and their own scores, using a specific formula or algorithm.
- Convergence: Keep iterating until the scores converge or reach stability (i.e., they stop changing significantly).
- Display Results: Finally, display or use the page scores to rank pages in search results, with higher scores indicating higher relevance or authority.

Implementing PageRank in Java involves setting up a web graph, calculating scores iteratively based on connections, and refining these scores until convergence. The core concept revolves around assigning importance scores to web pages based on their connections and the importance of the linking pages.

Java provides tools to handle graph structures and mathematical computations necessary for implementing PageRank algorithms. Libraries like Apache Commons Math might aid in numerical calculations.

However, a full-fledged implementation of PageRank involves more intricate mathematics and handling of larger-scale web graphs than can be covered in a simple example. Real-world applications of PageRank also involve handling diverse factors beyond mere link structures.

For a comprehensive implementation, an advanced understanding of graph theory, numerical algorithms, and handling large datasets is required. Libraries or frameworks specializing in graph algorithms could be utilized for efficient computations.

2.8. Data Validation using regular expression

Data validation using regular expressions serves to check if a particular data entry conforms to a specified pattern or structure. It ensures that user-provided information aligns with predetermined rules, such as validating email addresses, phone numbers, passwords, or any other formatted data. Executing data validation through regular expressions in Java encompasses establishing patterns and employing regex techniques to verify if the input aligns with these predefined patterns.

Uses of Data Validation using Regular Expressions:

- Form Input Validation: Used in web forms to validate user-entered data, ensuring it meets specific criteria like email format, phone number structure, or password complexity.
- Data Formatting and Consistency: Helps maintain data integrity by ensuring data conforms to specific formats, preventing errors or inconsistencies.
- Input Sanitization: Filters out unwanted or malicious characters from user inputs.

Here's an overview of implementing data validation using regular expressions in Java:

- Defining Regular Expressions: Define regex patterns that represent the required format or structure. For instance, a regex pattern for email validation or a specific date format.
- Using Pattern and Matcher Classes: Utilize Java's Pattern and Matcher classes from the java.util.regex package. Compile the regex pattern into a Pattern object using Pattern.compile(). Use a Matcher object to match the input string against the compiled regex pattern using matcher.matches() or matcher.find() methods.
- Checking for Matches: Validate user input by checking if it matches the defined regex pattern using methods like matches() or find() from the Matcher class.
- Feedback or Handling Invalid Input: Provide feedback to users about input validation results. Handle invalid inputs appropriately, such as displaying error messages or prompting users to re-enter data.

2.9. Frequency Counting

Frequency counting involves tallying the occurrences of specific elements, such as words, characters, or values, within a dataset or collection. Its purpose is to analyze and determine the frequency or prevalence of these elements. In Java, frequency counting can be implemented by iterating through the dataset, keeping track of element occurrences using data structures like HashMaps or arrays.

Uses of Frequency Counting:

- **Word Frequency in Text:** Analyzing how often words appear in a text document or a set of documents. This is useful for tasks like determining word importance or assessing language usage.
- **Character Frequency:** Counting the occurrence of characters in strings or documents, aiding in tasks such as analyzing language distribution or identifying common characters.
- **Value Occurrences:** Tallying occurrences of specific values or elements in a dataset, helping in statistical analysis or understanding data distributions.

Implementation:

- **Choose Data Source:** Select the dataset or source where you want to count frequencies, such as text files, arrays, collections, or any data structure.
- **Iterate and Tally:**
    - Use loops or iteration techniques to traverse through the dataset.
    - For each element encountered, increment the corresponding count in a HashMap, array, or any suitable data structure.
- **Display or Use Results:** After tallying, you can display the frequencies, perform further analysis, or use the counts as needed in your Java application.

2.10. Finding patterns using regular expressions

Detecting patterns through regular expressions enables pinpointing particular sequences or structures within text-based information. This method proves invaluable for validating data formats, extracting specific details from text, or seeking out defined patterns within extensive datasets. In Java, the java.util.regex package offers functionalities to apply regular expressions, enabling pattern matching and information extraction according to predetermined rules.

Uses of Finding Patterns using Regular Expressions:

- **Data Validation:** Ensuring that user inputs (like emails, phone numbers, passwords) adhere to specified formats or rules.
- **Text Extraction:** Extracting specific information from text, such as dates, URLs, or keywords, for further processing or analysis.
- **Search and Replace:** Searching for patterns within text to replace or manipulate specific portions based on predefined criteria.

Implementing pattern finding using regular expressions in Java involves several steps:

- **Define the Pattern:** Create a regex pattern that represents the structure or sequence you're looking for within the text.
- **Compile the Pattern:** Compile the regex pattern into a Pattern object using Pattern.compile().
- **Match the Pattern:** Use the Matcher class to match the pattern against the input text using methods like matcher.find() or matcher.matches().
- **Extract or Process Matching Results:** If a match is found, extract or process the matched portions using methods like group() in the Matcher class.

3. Data Structures and Algorithms Implemented

3.1. Heapsort

The use of heap sort lies in efficiently sorting arrays or lists of elements by creating a binary heap data structure. This sorting algorithm leverages the properties of a heap – a specialized tree-based structure – to repeatedly extract the maximum (for a max-heap) or minimum (for a min-heap) element, resulting in a sorted collection. Heap sort is efficient, particularly for large datasets, due to its O(n log n) time complexity for sorting. [1]

Use of Heap Sort:
Efficient Sorting: Heap sort offers consistent performance in sorting large datasets, making it suitable for scenarios requiring reliable and efficient sorting.
Space Complexity:

It has an optimal space complexity of O(1) in-place sorting, meaning it doesn't require additional memory proportional to the input size, except for a constant amount.[2]

Implementing heap sort in Java involves several steps:

- **Heapify:** Build a heap from the input array. This involves rearranging the elements to satisfy the heap property (max-heap or min-heap).
- **Heapify Procedure:** The procedure involves repeatedly swapping elements to maintain the heap property, ensuring the root node (maximum or minimum) satisfies the heap condition.
- **Heap Sorting:** After constructing the heap, repeatedly extract the root element (which is the maximum or minimum) and reconstruct the heap to maintain the property.
- **Sorting Algorithm:** The sorting algorithm is essentially the heapify process followed by extraction of elements to sort the array in ascending or descending order.

Time Complexity: O(n logn), where *n* is the number of elements in the input array.

3.2. Trie Tree

A Trie, also known as a prefix tree, is a tree-like data structure used for efficient retrieval and storage of strings or keys in a dataset. It's particularly useful for tasks involving dictionaries, autocomplete features, search engines, and word-related problems. In Java, a Trie can be implemented using nodes to represent characters, where each node contains links to its child nodes representing subsequent characters in words.

Use of Trie Tree:

- Efficient String Searches: Trie provides quick searching, insertion, and deletion of strings. It's ideal for dictionary-like applications, where searching for words or prefixes is frequent.
- Autocomplete and Suggestions: Trie structures facilitate autocomplete features in search engines or applications, offering suggestions based on partial input.
- Storing and Retrieving Dictionary Words: Tries excel in scenarios where storage and retrieval of words or strings are core requirements, such as language processing tasks.[3]

Implementing a Trie in Java involves defining nodes to represent characters and implementing operations like insertion, search, deletion, and autocomplete features.

- **Node Structure:** Define a TrieNode class representing each node in the Trie. This node typically includes pointers to child nodes and flags to indicate the end of a word.
- **Trie Operations:** Implement methods to insert a word into the Trie, search for a word, delete a word, and retrieve suggestions or autocomplete options based on partial inputs.
- **Usage:** Utilize the Trie structure in applications where efficient string storage, retrieval, or autocomplete functionalities are required.

Time Complexity: O(n), *n* is the length of the key being searched for.

3.3. Hash-Map

A HashMap in Java is a data structure that stores key-value pairs and offers quick insertion, deletion, and retrieval of elements. It utilizes a hashing technique to map keys to their associated values. HashMaps are used when efficient access to elements based on keys is required, enabling fast lookup and manipulation of data. In Java, HashMaps are implemented as part of the java.util package. [4]

Use of HashMap:

- **Fast Retrieval:** HashMaps provide rapid retrieval of values associated with keys, making them suitable for scenarios where quick access to elements is crucial.
- **Key-Value Association:** They establish an association between keys and values, allowing for efficient storage and retrieval based on keys.
- **Flexible Data Storage:** HashMaps accommodate various types of keys and values, making them versatile for storing different kinds of data.

Java Implementation:

- **Create a HashMap:**

    - Begin by making an instance of the HashMap class available in the java.util package.
    - When creating the HashMap, specify the types of elements the keys and values will hold. For instance, you might have HashMap<KeyType, ValueType>.
- **Adding Information:**
    - To place data into the HashMap, utilize the put() method. This method requires both the key and its corresponding value as arguments.
    - By using the put() method, you can assign values to specific keys within the HashMap.
- **Accessing Information:**
    - Retrieve stored values based on their associated keys with the get() method.
    - When utilizing get(), simply provide the desired key as an argument to receive the corresponding value stored within the HashMap.

Time Complexity: O(n) for worst case and O(1) for best case, where n is the size of the hash-map.

3.4. Pattern Matching

Pattern matching is the process of identifying specific patterns or sequences within a larger set of data, often within strings. It's crucial for tasks like searching, validating, or extracting relevant information based on predefined patterns or rules. In Java, pattern matching is implemented using various techniques, including regular expressions, string manipulation, or algorithms like the Knuth-Morris-Pratt (KMP) algorithm.[5]

Uses of Pattern Matching:

- Text Processing: Finding specific words, phrases, or sequences within text for tasks like search, extraction, or validation.
- Data Validation: Verifying whether a given input adheres to specific patterns, like checking if an email address or phone number follows a valid format.
- Parsing and Tokenization: Breaking down text into tokens or parts based on defined patterns, facilitating further analysis or processing.
- Searching and Information Retrieval: Aiding in locating specific patterns or sequences within documents or data sets, enabling efficient search and retrieval operations.

In Java, pattern matching can be achieved through various methods:

- Regular Expressions (Regex): Utilizing the java. util.regex package to define and match patterns using regex expressions. This method provides extensive capabilities for pattern-based search and manipulation.

- String Methods: Utilizing built-in string methods like indexOf(), contains(), or substring() to identify specific substrings or patterns within strings.
- Algorithms (e.g., KMP Algorithm): Implementing specialized algorithms like the Knuth-Morris-Pratt (KMP) algorithm to efficiently search for patterns within strings or text.
- Third-Party Libraries: Employing external libraries or frameworks that offer advanced pattern-matching functionalities beyond the core Java features.

Time Complexity: O(n+m), where n is the word and m is the pattern.

3.5. Edit Distance

The edit distance, also known as Levenshtein distance, quantifies the similarity between two strings by measuring the minimum number of operations (insertion, deletion, or substitution of a single character) required to transform one string into another. It's a fundamental concept in various fields, including computer science, natural language processing, bioinformatics, and spell-checking.[6]

Edit Distance Usage:

- **Spell Checking and Correction:**
  - Detects misspelled words by proposing corrections from words that closely resemble the input with a minimal edit distance.
- **Genetic Analysis:**
  - In bioinformatics, it facilitates the comparison of DNA sequences, enabling the identification of mutations or similarities between genetic sequences.
- **Information Retrieval:**
  - Assesses textual similarity, which assists in refining search algorithms and ranking systems by measuring the resemblance between different texts.

  - Implementing edit distance involves using algorithms like dynamic programming, such as the Wagner–Fischer algorithm.
- **Initialization:**
  - Create a matrix (often a 2D array) to store edit distances between substrings of the two input strings.
  - Initialize the matrix based on the lengths of the input strings.
- **Base Cases:**
  - Fill in the first row and the first column of the matrix with incremental values representing the edit distances for empty strings to substrings of the inputs.

    - For instance, the first row will contain edit distances from an empty string to substrings of the second input string, and vice versa for the first column.
- **Dynamic Programming Approach:**
    - Iterate through the matrix, computing edit distances between all possible substrings of the two input strings.
    - For each cell, determine the minimum cost of converting one substring to another (insertion, deletion, or substitution).
    - Update the cell with the minimum edit distance required to transform one substring into another.
- **Final Result:**
    - The bottom-right cell of the matrix will contain the edit distance between the entire input strings.
    - This value represents the minimum number of operations needed to convert one string into the other.
- **Backtracking (Optional):**
    - If required, trace back through the matrix to reconstruct the sequence of edit operations (insertion, deletion, or substitution) that leads to the minimum edit distance.
- **Implementation:**
    - Write code based on the algorithm chosen (e.g., Wagner–Fischer) to perform these steps and calculate the edit distance between two given strings.

This approach of utilizing dynamic programming builds a matrix to efficiently compute and store edit distances for substrings, culminating in the overall edit distance between the complete input strings. Adjustments or optimizations can be made based on specific use cases or variations of the algorithm chosen for implementation.

Time Complexity: O(n*m)

4. Screenshots

Driver Class: The provided Java code represents a Selenium WebDriver implementation to perform web scraping operations. It includes functionalities to navigate to URLs, interact with web elements, handle tabs, and manage browser instances. The Driver class initializes the WebDriver, sets up ChromeDriver properties, and offers methods for actions like finding elements, waiting for element visibility, handling tabs, and performing interactions like clicks on web elements.

```java
import com.accommodation.pricing.analysis.constant.Constants;

/**
 * Driver class is responsible for performing operation selenium web driver and web driver wait instance.
 * This class contains various methods that are useful to performing scrap operations.
 * for example:
 * navigateToUrl(String url) :  naviagte to url inside selenium browser window
 *
 * @author Kashyap Prajapati (110126934)
 *
 */
public class Driver {

    private static final Duration EXPLICIT_WAIT_TIME_IN_SECONDS=Duration.ofSeconds(10);

    private WebDriver driver;
    private WebDriverWait webDriverWait;

    /**
     * Constructor is used to initialized the driver and web driver instance of
     * the selenium.
     * Also set the system properties of chrome driver and chromer driver path.
     *
     */
    public Driver() {
        System.setProperty(Constants.CHROME_DRIVER,"src/main/resources/chromedriver-117.exe");
        ChromeOptions opt = new ChromeOptions();
        opt.addArguments("start-maximized");
        opt.addArguments("disable-infobars");
        opt.addArguments("--disable-extensions");
        opt.addArguments("--remote-allow-origins=*");
        opt.addArguments("--ignore-certificate-errors");
        opt.addArguments("--allow-insecure-localhost");
        opt.addArguments("acceptInsecureCerts");
        this.driver=new ChromeDriver(opt);
        this.webDriverWait = new WebDriverWait(driver, EXPLICIT_WAIT_TIME_IN_SECONDS);
    }


    /**
     * This method is used to navigate to url inside browser.
     * @param url :  input url
     */
    public void navigateToUrl(String url) {
        driver.navigate().to(url);
```

```
59       */
60      public void navigateToUrl(String url) {
61          driver.navigate().to(url);
62          driver.manage().window().maximize();
63      }
64
65      /**
66       * This method is used to get the instance of the driver.
67       * @return the web driver instance of the selenium
68       */
69      public WebDriver getDriver() {
70          return this.driver;
71      }
72
73
74      /**
75       * close the driver instance of selenium.
76       *
77       */
78      public void close() {
79          this.driver.close();
80      }
81
82      /**
83       * This method is used to find single element in browsers
84       * @param xPath : x path of the elements
85       * @return : single element of browsers
86       */
87      public WebElement waitForElementToVisible(String xPath) {
88          return webDriverWait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath(xPath)));
89      }
90
91      /**
92       * This method is used to find multiple elements in browsers
93       * @param xPath : xpath of the respective elements
94       * @return : the list of web elements that represent x path
95       */
96      public List<WebElement> findElements(String xPath){
97          return webDriverWait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath(xPath))).findElemen
98      }
99
100     /**
101      * This method is used to check element is displayed or not in browser:
102      * @param xPath : xPath of the respective elements.
```

```
103      * @return : true element is available | false element is not available or throwing timeout exception.
104      */
105     public boolean isElementDisplayed(String xPath) {
106         try {
107             return webDriverWait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath(xPath))).isDisp
108         }catch (TimeoutException e) {
109             return false;
110         }
111     }
112
113     /**
114      * This method is used to perform click operation on web elements.
115      * Use until method of web driver wait to check element is available or not before performing click opera
116      * @param xPath :  XPATH of respective element where you want to perform click operations
117      */
118     public void clickOnWebElement(String xPath) {
119         webDriverWait.until(ExpectedConditions.visibilityOfElementLocated(By.xpath(xPath))).click();
120     }
121
122
123     /**
124      * This method is used to open a new tab in browsers
125      */
126     public void openNewTab() {
127         driver.switchTo().newWindow(WindowType.TAB);
128     }
129
130     /**
131      * This method close the current tab
132      */
133     public void closeCurrentTab() {
134         ArrayList<String> tabs = new ArrayList<> (driver.getWindowHandles());
135         driver.close();
136         driver.switchTo().window(tabs.get(0));
137     }
138
139     /**
140      * This method is used to switch the tabs in browsers
141      * @param tab :  name of tab where you want switch
142      */
```

```java

    /**
     * This method is used to open a new tab in browsers
     */
    public void openNewTab() {
        driver.switchTo().newWindow(WindowType.TAB);
    }

    /**
     * This method close the current tab
     */
    public void closeCurrentTab() {
        ArrayList<String> tabs = new ArrayList<> (driver.getWindowHandles());
        driver.close();
        driver.switchTo().window(tabs.get(0));
    }

    /**
     * This method is used to switch the tabs in browsers
     * @param tab :  name of tab where you want switch
     */
    public void switchToTab(String tab) {
        driver.switchTo().window(tab);
    }


    /**
     * This method is used to close the current tab and switch to parent tab.
     * return void
     */
    public void closeCurrentAndSwitchToParent() {
         ArrayList<String> tabs = new ArrayList<>(driver.getWindowHandles());
         if(tabs.size()>1) {
             driver.close();
             driver.switchTo().window(tabs.get(0));
         }
    }
}


```

Html Util: The `HtmlUtil` class provides utility methods for handling HTML parsing using the Jsoup library in Java. It offers functionalities to parse HTML strings into Document objects, find elements based on CSS selectors, retrieve text content of the first matching element, and load documents from URLs. This class encapsulates operations for parsing HTML content and extracting information from web pages through Jsoup library methods.

```
 *
 * This is the HtmlUtil class to parse HTML string to get Document object.
 *
 * @author KASHYAP PRAJAPATI 110126934
 *
 */
public class HtmlUtil {

    private HtmlUtil() {

    }

    /**
     * This method is used to parse the html string and get document objects
     * @param htmlString
     * @return
     */
    public static Document parse(String htmlString) {
        return Jsoup.parse(htmlString);
    }


    /**
     * This method is used to find element based on CSS Selector
     * @param document
     * @param selectedHref
     * @return
     */
    public static Elements select(Document document,String selectedHref) {
        return document.select(selectedHref);
    }


    /**
     * This method is used to get first element text based on CSS selector
     * @param document
     * @param selectedHref
     * @return
     */
    public static String getFirstElementText(Document document,String selectedHref) {
        Elements  elements = document.select(selectedHref);
        if(elements.isEmpty())
            return "not found";
        return elements.first().text();
    }


    /**
     * This method is used to load document from the url.
     * @param url
     * @return
     */
    public static Document loadDoument(String url) {
        try {
            return Jsoup.connect(url).get();
        } catch (IOException e) {
            return null;
        }
    }
```

Edit Distance: The `EditDistance` class implements an algorithm based on the Longest Common Subsequence (LCS) dynamic programming technique. It calculates the Levenshtein distance between two words, allowing suggestions for similar words based on their differences in characters (insertions, deletions, substitutions). The `suggest_corrections` method compares a given word with a list of valid words, providing suggestions by identifying the closest matching words within a given threshold.

```
54            String []atomic_words_hotels;
55            //Declaring hashmap to store the suggested words and their conversion cost
56            LinkedHashMap<String, Integer> sorted_suggestions = new LinkedHashMap<String, Integer>();
57            //Variable to store each word from the list of words
58            String word_to_compare = "";
59            //Looping through each word in the list to calculate the cost of conversion
60            for(int list_counter=0; list_counter<valid_words.size(); list_counter++) {
61                word_to_compare = valid_words.get(list_counter);
62                atomic_words_hotels = word_to_compare.split(" ");
63                int min_dist = 9999;
64                for(int atomic_counter = 0; atomic_counter<atomic_words_hotels.length; atomic_counter++ )
65                {
66                    String single_word = atomic_words_hotels[atomic_counter];
67                    int comparator_word_length = single_word.length();
68                    int[][] comparator_array = new int[user_word_length+1][comparator_word_length+1];
69                    //System.out.println(comparator_array);
70                    //Code to calculate the Levenshtein distance based on LCS programming technique
71                        for(int i=0; i<=user_word_length; i++){
72                            for(int j=0; j<=comparator_word_length; j++){
73                                if (i == 0) {
74                                    comparator_array[i][j] = j;
75                                }
76                                else if (j == 0) {
77                                    comparator_array[i][j] = i;
78                                }
79                                else {
80                                    //Calculating the Levenshtein distance
81                                    comparator_array[i][j] = find_min_operations(comparator_array[i - 1][j - 1]
82                                     + cost_to_nearest_word(user_literal.charAt(i - 1), single_word.charAt(j - 1)),
83                                     comparator_array[i - 1][j] + 1,
84                                     comparator_array[i][j - 1] + 1);
85                                }
86                            }
87                        }
88                        if(comparator_array[user_word_length][comparator_word_length]<min_dist)
89                            min_dist = comparator_array[user_word_length][comparator_word_length];
90                }
91
92                        //Inserting the words and their distances into the hashmap
93                        sorted_suggestions.put(word_to_compare, min_dist);
94                }
95            //Declaring an arraylist to sort the words in Linked Hash Map based on their distances in ascending order
96            List<Map.Entry<String, Integer> > sorter_array = new ArrayList<Map.Entry<String, Integer> >(
97                    sorted_suggestions.entrySet());
98
99            //Sorting the words in the Linked Hash Map
100           Collections.sort(
101                   sorter_array,
102                   new Comparator<Map.Entry<String, Integer> >() {
103                       // Comparing two entries by value
104                       public int compare(
105                           Map.Entry<String, Integer> entry1,
106                           Map.Entry<String, Integer> entry2)
107                       {
108
109                           // Subtracting the entries
110                           return entry1.getValue()
111                               - entry2.getValue();
112                       }
```

```
                                    comparator_array[i][j] = j;
                                }
                                else if (j == 0) {
                                    comparator_array[i][j] = i;
                                }
                                else {
                                    //Calculating the Levenshtein distance
                                    comparator_array[i][j] = find_min_operations(comparator_array[i - 1][j - 1]
                                     + cost_to_nearest_word(user_literal.charAt(i - 1), single_word.charAt(j - 1)),
                                     comparator_array[i - 1][j] + 1,
                                     comparator_array[i][j - 1] + 1);
                                }
                            }
                        }
                        if(comparator_array[user_word_length][comparator_word_length]<min_dist)
                            min_dist = comparator_array[user_word_length][comparator_word_length];
                    }

                    //Inserting the words and their distances into the hashmap
                    sorted_suggestions.put(word_to_compare, min_dist);
                }
            //Declaring an arraylist to sort the words in Linked Hash Map based on their distances in ascending order
            List<Map.Entry<String, Integer> > sorter_array = new ArrayList<Map.Entry<String, Integer> >(
                    sorted_suggestions.entrySet());

            //Sorting the words in the Linked Hash Map
            Collections.sort(
                    sorter_array,
                    new Comparator<Map.Entry<String, Integer> >() {
                        // Comparing two entries by value
                        public int compare(
                            Map.Entry<String, Integer> entry1,
                            Map.Entry<String, Integer> entry2)
                        {

                            // Subtracting the entries
                            return entry1.getValue()
                                - entry2.getValue();
                        }
                    });

            int display_counter = 1;
            for (Map.Entry<String, Integer> item : sorter_array) {

                // Printing the sorted map till the threshold value
                System.out.println(display_counter+" "+item.getKey()+" "+item.getValue());

                display_counter++;
                if(display_counter==threshold_val)
                    break;
            }


        }



}
```

```
/**
 * This program is based on LCS dynamic programming technique to suggest words based on Levenshtein distance.
 * It handles difference between 2 words in terms of insertion, deletion and substitution of characters among them.
 * @author Subhram Satyajeet (110127932)
 *
 */

import java.io.File;

public class EditDistance {

    /**
     * This function checks if 2 words are same or different
     * @param a: word 1
     * @param b: word 2
     * @return
     */
    public static int cost_to_nearest_word(char a, char b) {
        //Function to check the cost of converting one word to another
        return a == b ? 0 : 1;
    }

    /**
     * This function finds the min number from a range of values
     * @param input_nums: range of values
     * @return
     */
    public static int find_min_operations(int... input_nums) {
        //Function to calculate the minimum number of operations to convert one word to another
        return Arrays.stream(input_nums)
          .min().orElse(Integer.MAX_VALUE);
    }

    /**
     * This functions suggests corrections by calculating Levenshtein distance between 2 given words
     * @param user_literal: Word to compare
     * @param valid_words: List of valid given words in repository
     * @param threshold_val: The threshold number of suggestions provided
     */
    public static void suggest_corrections(String user_literal, ArrayList<String> valid_words, int threshold_val){
        //Function to suggest the nearest word based on the misspelled word
        //Variable to take user provided word
        int user_word_length = user_literal.length();
        String []atomic_words_hotels;
        //Declaring hashmap to store the suggested words and their conversion cost
        LinkedHashMap<String, Integer> sorted_suggestions = new LinkedHashMap<String, Integer>();
        //Variable to store each word from the list of words
        String word_to_compare = "";
        //Looping through each word in the list to calculate the cost of conversion
        for(int list_counter=0; list_counter<valid_words.size(); list_counter++) {
            word_to_compare = valid_words.get(list_counter);
            atomic_words_hotels = word_to_compare.split(" ");
            int min_dist = 9999;
            for(int atomic_counter = 0; atomic_counter<atomic_words_hotels.length; atomic_counter++ )
            {
                String single_word = atomic_words_hotels[atomic_counter];
                int comparator_word_length = single_word.length();
                int[][] comparator_array = new int[user_word_length+1][comparator_word_length+1];
                //System.out.println(comparator_array);
```

Heap Sort:

Heap sort implementation

```java
package com.accommodation.pricing.analysis.algorithms;
/**
 * This program performs heapsort to find the best deals for users from hotel price list.
 * @author Subhram Satyajeet (110127932)
 *
 */

//Code to sort the data on prices
import java.util.Collections;
import java.util.List;
import java.util.Map;

public class HeapSort {


    //sorting functionality
    /**
     * This function performs heapsort on the list of hotels and their prices using recursion
     * @param entryList: The list of hotel names and prices
     */
    public static void sortHotelPriceFuction(List<Map.Entry<Integer, String>> entryList) {
        int dim_size = entryList.size();

        //recursively sort the functionality
        for (int heapsize = dim_size / 2 - 1; heapsize >= 0; heapsize--) {
            remain_heapify(entryList, dim_size, heapsize);
        }


        for (int heapsize = dim_size - 1; heapsize > 0; heapsize--) {

            Collections.swap(entryList, 0, heapsize);


            remain_heapify(entryList, heapsize, 0);
        }
    }

    /**
     * This function performs heapify after insertion of elements
     * @param remainList: the remaining list passed after sorting
     * @param heapsize: the size of the remaining heap
     * @param node_pos: the root node value
     */
    private static void remain_heapify(List<Map.Entry<Integer, String>> remainList, int heapsize, int node_pos) {
        int node_large_val = node_pos;
        int left_small_val = 2 * node_pos + 1;
        int right_small_val = 2 * node_pos + 2;

        // Check if left child is greater than the root
        if (left_small_val < heapsize && remainList.get(left_small_val).getKey() > remainList.get(node_large_val).getKey()) {
            node_large_val = left_small_val;
        }

        // Check if right child is greater
        if (right_small_val < heapsize && remainList.get(right_small_val).getKey() > remainList.get(node_large_val).getKey()) {
            node_large_val = right_small_val;
        }

        // If the value of root is not the largest
```

```java
public class HeapSort {


    //sorting functionality
    /**
     * This function performs heapsort on the list of hotels and their prices using recursion
     * @param entryList: The list of hotel names and prices
     */
    public static void sortHotelPriceFuction(List<Map.Entry<Integer, String>> entryList) {
        int dim_size = entryList.size();

        //recursively sort the functionality
        for (int heapsize = dim_size / 2 - 1; heapsize >= 0; heapsize--) {
            remain_heapify(entryList, dim_size, heapsize);
        }


        for (int heapsize = dim_size - 1; heapsize > 0; heapsize--) {

            Collections.swap(entryList, 0, heapsize);


            remain_heapify(entryList, heapsize, 0);
        }
    }

    /**
     * This function performs heapify after insertion of elements
     * @param remainList: the remaining list passed after sorting
     * @param heapsize: the size of the remaining heap
     * @param node_pos: the root node value
     */
    private static void remain_heapify(List<Map.Entry<Integer, String>> remainList, int heapsize, int node_pos) {
        int node_large_val = node_pos;
        int left_small_val = 2 * node_pos + 1;
        int right_small_val = 2 * node_pos + 2;

        // Check if left child is greater than the root
        if (left_small_val < heapsize && remainList.get(left_small_val).getKey() > remainList.get(node_large_val).getKey()) {
            node_large_val = left_small_val;
        }

        // Check if right child is greater
        if (right_small_val < heapsize && remainList.get(right_small_val).getKey() > remainList.get(node_large_val).getKey()) {
            node_large_val = right_small_val;
        }

        // If the value of root is not the largest
        if (node_large_val != node_pos) {
            Collections.swap(remainList, node_pos, node_large_val);

            // Recursively heapify the affected sub-tree
            remain_heapify(remainList, heapsize, node_large_val);
        }
    }


}
```

Inverted Index: The code represents an implementation of an inverted index data structure using a HashMap in Java. It builds an inverted index from hotel data, allowing searches based on specific keys and retrieving associated hotel information efficiently. The index organizes hotels by their attributes like names, amenities, etc., aiding in quick search and retrieval.

```java
package com.accommodation.pricing.analysis.algorithms;

import java.util.ArrayList;

public class InvertedIndex {

    private Map<String,Set<Hotel>> invertedIndexData;

    private InvertedIndex() {

    }

    private void checkForInvertedIndexPresentOrNot(String key,Hotel hotel) {
        if(key!=null) {
            if(!invertedIndexData.containsKey(key)) {
                Set<Hotel> set = new HashSet<>();
                set.add(hotel);
                invertedIndexData.put(key,set);
            }else {
                Set<Hotel> set = invertedIndexData.get(key);
                set.add(hotel);
                invertedIndexData.put(key,set);
            }
        }
    }

    private void checkForInvertedIndexPresentOrNot(String[] keys,Hotel hotel) {
        for(String key : keys) {
            checkForInvertedIndexPresentOrNot(key,hotel);
        }
    }


    private void checkForInvertedIndexPresentOrNot(List<String> keys,Hotel hotel) {
        for(String key : keys) {
            checkForInvertedIndexPresentOrNot(key,hotel);
        }
    }

    public InvertedIndex(List<Hotel> hotels) {
        invertedIndexData = new LinkedHashMap<>();
        for(Hotel hotel:hotels) {
            String[] nameWords = hotel.getName().split("\\s");
            checkForInvertedIndexPresentOrNot(nameWords,hotel);
            checkForInvertedIndexPresentOrNot(hotel.getDescription(), hotel);
            checkForInvertedIndexPresentOrNot(hotel.getOverView().split("\\s"), hotel);
            checkForInvertedIndexPresentOrNot(hotel.getAmenities(), hotel);
            checkForInvertedIndexPresentOrNot(hotel.getAllAmenities(), hotel);
        }
    }


    public List<Hotel> getInvertedIndexSearchData(String userInput) {
        List<Hotel> list = new ArrayList<>();
        for(String key : invertedIndexData.keySet()){
            if(key.startsWith(userInput)) {
                list.addAll(invertedIndexData.get(key));
            }
        }
        System.out.println("list"+list);
```

Trie class

This Java class implements a Trie data structure used for word suggestions, offering autocomplete and word prediction functionalities. It enables efficient storage, retrieval, and suggestion of words based on prefixes, facilitating quick word recommendations. The `Trie` class initializes with a list of words, allowing for prefix-based word retrieval and generation of word suggestions for a given prefix.

```java
package com.accommodation.pricing.analysis.algorithms;

import java.util.ArrayList;
import java.util.List;
/**
 * This class implements the Trie data structure for word suggestion purposes that
 * enabling autocomplete and word prediction functions,
 * it offers functionality for effectively storing and retrieving words.
 *
 * @author sachreet kaur(110122441)
 *
 */
public class Trie {

    Node rootNode;

    /**
     * Constructor for Trie that initializes the Trie with a list of words.
     *
     * @param words List of words to be inserted into the Trie.
     *
     * @author sachreet kaur(110122441)
     */
    public Trie(List<String> words) {
        rootNode = new Node();
        for(String word:words) {
            rootNode.insertNode(word);
        }
    }
    /**
     *
     * @param prefix : The prefix  used to find for searching in Trie.
     * @param position : If the last node is the end of a word, see if it is true.
     * @return True if the Trie contains the node with the specified prefix.
     *
     * @author sachreet kaur(110122441)
     */
    public boolean findNode(String prefix,boolean position) {
        Node lastNode = rootNode;
        for(char character : prefix.toCharArray()) {
            lastNode = lastNode.childs.get(character);
            if(lastNode==null)
                return false;
        }
        return !position || lastNode.isWord;
    }

    /**
     *
     * @param prefix : The prefix in the Trie that has to be looked up.
     *
     * @return If the last node is the end of a word, see if it is true.
     *
     * @author sachreet kaur(110122441)
     */
    public boolean find(String prefix) {
        return findNode(prefix, false);
    }

```

Node class

```java
 */

public class Node {
    // Map to store child nodes, where the character is the key and the corresponding node is the value.
    Map<Character,Node> childs;

    // The character that is linked to the current node.
    char character;

    // Indicates whether the node is at the end of a word.
    boolean isWord;

    /**
     * A Trie node's default constructor. Initializes the map of child nodes.
     *
     *  @author sachreet kaur(110122441)
     */
    public Node() {
        childs = new HashMap<>();
    }

    /**
     * Constructs a Trie node with a given character. Initializes the map of child nodes.
     *
     * @param character The character associated with the node.
     *
     *  @author sachreet kaur(110122441)
     */
    public Node(char character) {
        this.character=character;
        childs =  new HashMap<>();
    }

    /**
     * Starts from the current node and inserts a word into the Trie.
     *
     * @param searchWord The word to be inserted into the Trie.
     *
     * @author sachreet kaur(110122441)
     */
    public void insertNode(String searchWord) {
        if(searchWord == null || searchWord.isEmpty())
            return;
        char firstCharacter = searchWord.charAt(0);
        Node childNode = childs.get(firstCharacter);
        if(childNode == null) {
            childNode = new Node(firstCharacter);
            childs.put(firstCharacter, childNode);
        }

        if(searchWord.length()>1)
            childNode.insertNode(searchWord.substring(1));
        else
            childNode.isWord = true;
    }
}

```

Page ranking class

This Java program calculates and ranks keywords based on their occurrences within a text document using a basic page ranking algorithm. It reads input from a database, counts keyword occurrences, assigns ranks, sorts the keywords based on rank.

```
 * @author SAMIKSHA ARORA
 *
 */
public class PageRanking {

    public void implementPageRank(List<Hotel> hotelList) {
        String content = "";
        for(Hotel hotel : hotelList) {
            content = content+ " " +Validator.removeSpecialCharacterFromText(hotel.getOverView());
        }
         String[] words = content.toLowerCase().split("\\s+");

         int[] keywordOccurrences = new int[words.length];
         for (int i = 0; i < words.length; i++) {
             for (int j = i + 1; j < words.length; j++) {
                 if (words[i].equals(words[j])) {
                     keywordOccurrences[i]++;
                 }
             }
         }

         int[] keywordRanks = new int[words.length];
         for (int i = 0; i < words.length; i++) {
             for (int j = 0; j < words.length; j++) {
                 if (keywordOccurrences[i] > keywordOccurrences[j]) {
                     keywordRanks[i]++;
                 }
             }
         }
         Keyword[] keywordArray = new Keyword[words.length];
         for (int i = 0; i < words.length; i++) {
             keywordArray[i] = new Keyword(words[i], keywordOccurrences[i], keywordRanks[i]);
         }
         Arrays.sort(keywordArray);
         System.out.println("|=================================================================|");
         System.out.println("|    Keyword        |     Occerrences     |         Rank          |");
         System.out.println("|=================================================================|");
         for (Keyword keyword : keywordArray) {
             System.out.println("|             "+keyword.getWord()+"       |  "+Integer.toString(keyword.getOccurrences())+"         |      "+Integer.toString(keyword.getRank()));
             break;
         }
         System.out.println("|=================================================================|");
    }

    static class Keyword implements Comparable<Keyword> {
        private String word;
        private int occurrences;
        private int rank;

        public Keyword(String word, int occurrences, int rank) {
            this.word = word;
            this.occurrences = occurrences;
            this.rank = rank;
        }

        public String getWord() {
            return word;
        }

        public int getOccurrences() {
            return occurrences;
        }

        public int getRank() {
            return rank;
        }

        @Override
        public int compareTo(Keyword other) {
            return Integer.compare(other.rank, this.rank);
        }
```

Validator

This Java class `Validator` contains methods using regular expressions to validate and process text efficiently. It includes functions to check if a string consists only of numeric characters, remove non-alphanumeric characters, eliminate non-alphabetic characters including spaces, filter out non-numeric characters, and strip whitespace characters from a given text. These methods facilitate text validation and manipulation for specific character types or patterns.

```java
package com.accommodation.pricing.analysis.validator;

import java.util.regex.Pattern;

/**
 * Utility class for validating and processing text.
 *
 * @author sachreet kaur(110122441)
 */

public class Validator {

    // Regular expression pattern to exclude all non-alphanumeric characters and spaces.
    private static final String EXCLUDE_ALPHANUM_PATTERN = "[^A-Za-z0-9\\s]";

    // Regular expression pattern to exclude all non-alphabetic characters.
    private static final String EXCLUDE_ALPHANUM_SPACE_PATTERN="[^A-Za-z]";

    // Regular expression pattern to exclude all non-numeric characters.
    private static final String EXCLUDE_NUMBER_PATTERN = "[^0-9]";

    // Regular expression pattern to match whitespace characters.
    private static final String WHITESPACE_PATTERN ="[\\s]";

    // Pattern to check if a string consists only of numeric characters.
    private static final Pattern numberPattern = Pattern.compile("^[0-9]+$");

    /**
     * Private constructor that prevent to make instance of the class.
     *
     * @author sachreet kaur(110122441)
     *
     */
    private Validator() {

    }

    /**
     * Checks if a given string consists only of numeric characters.
     *
     * @param inputNumber :  checking the input string.
     * @return True if the input consists only of numeric characters, false otherwise.
     *
     *  @author sachreet kaur(110122441)
     */
    public static boolean checkValidNumber(String inputNumber) {
        return numberPattern.matcher(inputNumber).matches();
    }


    /**
     * Removes all non-alphanumeric and space characters from a given text.
     *
     * @param text : The input text.
     * @return Special characters and spaces have been removed from the text.
     *
     * @author sachreet kaur(110122441)
     */
    public static String removeSpecialCharacterFromText(String text) {
        if(text==null)
```

```
52       * Removes all non-alphanumeric and space characters from a given text.
53       *
54       * @param text : The input text.
55       * @return Special characters and spaces have been removed from the text.
56       *
57       * @author sachreet kaur(110122441)
58       */
59      public static String removeSpecialCharacterFromText(String text) {
60          if(text==null)
61              return text;
62          return text.replaceAll(EXCLUDE_ALPHANUM_PATTERN, "");
63      }
64
65
66      /**
67       * Removes all non-alphabetic characters from a given text.
68       *
69       * @param text: The input text.
70       * @return Non-alphabetic characters have been removed from the text.
71       *
72       * @author sachreet kaur(110122441)
73       */
74      public static String removeSpecicalCharacterWithSpaceFromText(String text) {
75          if(text==null)
76              return text;
77          return text.replaceAll(EXCLUDE_ALPHANUM_SPACE_PATTERN, "");
78      }
79
80
81      /**
82       * Removes all non-numeric characters from a given text.
83       *
84       * @param text The input text.
85       * @return Text with non-numeric characters removed.
86       *
87       * @author sachreet kaur(110122441)
88       */
89      public static String removeSpecialCharacterFromNumber(String text) {
90          if(text==null)
91              return text;
92          return text.replaceAll(EXCLUDE_NUMBER_PATTERN, "");
93      }
94
95      /**
96       * Removes all whitespace characters from a given text.
97       *
98       * @param text The input text.
99       * @return Text with whitespace characters removed.
100      *
101      * @author sachreet kaur(110122441)
102      */
103     public static String removeWhiteSpace(String text) {
104         if(text==null)
105             return text;
106         return text.replaceAll(WHITESPACE_PATTERN, "");
107     }
108 }
109
```

KMP class

KMP (Knuth-Morris-Pratt) - String matching algorithm using prefix-suffix observations for efficient pattern search. It avoids unnecessary comparisons by utilizing the pattern's internal structure.

```
 */
public class KMP {

    /**
     * Empty constructor
     */
    public KMP() {

    }

    /**
     * Search pattern in text
     * @param pattern : pattern
     * @param text : text
     * @return -1 :  pattern is not found | index : found pattern.
     */
    public int searchKMP(String pattern, String text)
    {
        int M = pattern.length();
        int N = text.length();
        int lps[] = new int[M];
        int j = 0;

        lps(pattern, M, lps);

        int i = 0;
        while ((N - i) >= (M - j)) {
            if (pattern.charAt(j) == text.charAt(i)) {
                j++;
                i++;
            }
            if (j == M) {
                System.out.println("Found " +pattern + " at index "+ (i - j));
                j = lps[j - 1];
                return i - j;
            }
            else if (i < N && pattern.charAt(j) != text.charAt(i)) {
                if (j != 0)
                    j = lps[j - 1];
                else
                    i = i + 1;
            }
        }
        return -1;
    }

    /**
     * Search in LPS array
     * @param pattern
     * @param M
     * @param lps
     */
    void lps(String pattern, int M, int lps[]){
        int length = 0;
        int i = 1;
        lps[0] = 0;
        while (i < M) {
            if (pattern.charAt(i) == pattern.charAt(length)) {
                length++;
                lps[i] = length;
                i++;
            }
            else
            {
                if (length != 0) {
                    length = lps[length - 1];
                }
                else {
                    lps[i] = length;
                    i++;
                }
```

5. Output

```
|=====================================================================|
|                    MAIN MENU                                        |
|=====================================================================|
|Operation:                                                           |
|        1. Scrap the web site                                        |
|        2. Normal Database search                                    |
|        3. Search by word using inverted index                       |
|        4. Spell checker using edit distance                         |
|        5. Best deal suggestion using Heap Sort                      |
|        6. Finding Pattern in Text using KMP                         |
|        7. Page ranking                                              |
|        8. Exit                                                      |
|=====================================================================|
```

```
|=====================================================================|
|              Please select a site to scrap                          |
|=====================================================================|
| Operation:                                                          |
|        1. KAYAK (https://www.ca.kayak.com/)                         |
|        2. MOMONDO (https://www.momondo.ca/)                         |
|        3. VERBO (https://www.vrbo.com/)                             |
|        4. Previous Menu                                             |
|=====================================================================|
```

```
2
|=====================================================================|
|               Please select search criteria                         |
|=====================================================================|
| Operation:                                                          |
|        1. Search by City                                            |
|        2. Search by Name                                            |
|        3. Previous Menu                                             |
|=====================================================================|
```

```
4
|=====================================================================|
|     Please enter the word ?                                         |
|=====================================================================|
w
1 Sonder at The Liberty  TwoBedroom w Balcony 0
2 Sonder at The Liberty  OneBedroom w Balcony 0
3 Tropical Oasis w Hot Tub Near Astonishing Beaches 0
4 Private 1 Bed Apt Full Kitch Selfcheck  Spt Ent 1


|=====================================================================|
|                     MAIN MENU                                       |
|=====================================================================|
|Operation:                                                           |
|         1. Scrap the web site                                       |
|         2. Normal Database search                                   |
|         3. Search by word using inverted index                      |
|         4. Spell checker using edit distance                        |
|         5. Best deal suggestion using Heap Sort                     |
|         6. Finding Pattern in Text using KMP                        |
|         7. Page ranking                                             |
|         8. Exit                                                     |
|=====================================================================|
```

```
|=====================================================================|
 |   Name     : Sonder at The Liberty  OneBedroom w Balcony
 |   Price    : 154
 |   Address  : null
 |   City     : Toronto
 |   Score    : 9.2
 |   Review   : 5 reviews
 |   Amenties : [Washer, Dryer, Air conditioning, Gym]
 |=====================================================================|


|=====================================================================|
 |   Name     : Tropical Oasis w Hot Tub Near Astonishing Beaches
 |   Price    : 198
 |   Address  : null
 |   City     : West Palm Beach
 |   Score    : 5.0
 |   Review   : 1 review
 |   Amenties : [Hot Tub, Kitchen, Washer, Dryer, Air conditioning, Outdoor Space]
 |=====================================================================|


|=====================================================================|
 |   Name     : Sonder at The Liberty  TwoBedroom w Balcony
 |   Price    : 216
 |   Address  : null
 |   City     : Toronto
 |   Score    : 9.6
 |   Review   : 13 reviews
 |   Amenties : [Washer, Dryer, Air conditioning, Gym]
 |=====================================================================|
```

```
|================================================================|
|     Please enter the word ?                                    |
|================================================================|
dining
Found dining at index 329
```

```
|================================================================|
 |   Name     : PRIVATE 5 APARTMENT WITH 1 FREE PARKING  FULLY FURNISHED
 |   Price    : 71
 |   Address  : null
 |   City     : Mississauga
 |   Score    : 8.8
 |   Review   : 9 reviews
 |   Amenties : [Kitchen, Washer, Dryer, Air conditioning, Outdoor Space]
 |================================================================|
```

```
|================================================================|
|                     MAIN MENU                                  |
|================================================================|
|Operation:                                                      |
|        1. Scrap the web site                                   |
|        2. Normal Database search                               |
|        3. Search by word using inverted index                  |
|        4. Spell checker using edit distance                    |
|        5. Best deal suggestion using Heap Sort                 |
|        6. Finding Pattern in Text using KMP                    |
|        7. Page ranking                                         |
|        8. Exit                                                 |
|================================================================|
7
|================================================================|
|   Keyword       |    Occerrences    |        Rank              |
|================================================================|
|             the       |  1118        |      24683
|================================================================|
```

6. References

[1] Schaffer R, Sedgewick R. The analysis of heapsort. Journal of Algorithms. 1993 Jul 1;15(1):76-100.

[2] Carlsson S. Average-case results on heapsort. BIT Numerical Mathematics. 1987 Mar;27:2-17.

[3] Willard DE. New trie data structures which support very fast search operations. Journal of Computer and System Sciences. 1984 Jun 1;28(3):379-94.

[4] Laborde P, Feldman S, Dechev D. A wait-free hash map. International Journal of Parallel Programming. 2017 Jun;45:421-48.

[5] Hak T, Dul J. Pattern matching.

[6] Bar-Yossef Z, Jayram TS, Krauthgamer R, Kumar R. Approximating edit distance efficiently. In45th Annual IEEE Symposium on Foundations of Computer Science 2004 Oct 17 (pp. 550-559). IEEE.