Implementing Rabin-Karp and Hash Functions for Streamlined Search Engine Architecture.

Ahmad Thoriq Saputra - 13522141[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13522141@std.stei.itb.ac.id

***Abstract*— This paper explores the implementation of the Rabin-Karp string matching algorithm and custom hash functions in the development of a streamlined search engine architecture. The incorporation of the Rabin-Karp algorithm enhances the search efficiency, allowing users to input queries and obtain relevant results swiftly. Custom hash functions contribute to overall performance by converting variable-length strings into numerical representations. The synergy between these algorithms provides an optimal solution for matching user queries with dataset entries, demonstrating the efficacy of algorithmic techniques in search engine design. The paper concludes by acknowledging the foundational role of Rabin-Karp and hash functions in streamlining search processes, highlighting their impact on search engine architecture.**

***Keywords*—Hash Function, Pattern Searching, Rabin-Karp, Search Engine, Algorithmic Techniques.**

I. Introduction

In the complex world of search engine design, advanced algorithms play a crucial role in determining overall effectiveness. This paper explores the use of the Rabin-Karp algorithm and hash functions to improve the efficiency of modern search engines.

As information retrieval methods continue to evolve, there's a constant need for cutting-edge approaches. This research aims to contribute to this ongoing effort by investigating how integrating the Rabin-Karp algorithm and hash functions can enhance search engine architecture.

Compared to traditional designs, this approach combines the strengths of the Rabin-Karp algorithm, known for efficient string matching, and hash functions, valued for quick data indexing and retrieval. The goal is to explain these algorithms, detailing how they can be practically integrated into search engine frameworks and how this integration impacts query processing.

Moving beyond theory, the research aims to bridge the gap between abstract algorithms and real-world use, providing specific steps for seamlessly incorporating these advanced techniques into search engine architecture.

The central research question driving this exploration is: How does combining the Rabin-Karp algorithm and hash functions improve search engine architecture? This question guides the research journey, focusing on not just identifying potential improvements but also highlighting the considerations and trade-offs involved in this integration.

Divided into well-structured sections, this paper provides a detailed analysis of the Rabin-Karp algorithm and hash functions, exploring both their theoretical foundations and practical implications in search engine architecture. By shedding light on both theory and practice, the research aims to offer valuable insights into optimizing search engine performance. Ultimately, the goal is to empower practitioners and researchers with practical knowledge, fostering a deeper understanding of how algorithmic advancements impact the evolving landscape of search engine technology.

II. Theory Foundation

A. Query Processing

Query processing transforms a high-level query into lower-level operations, involving stages like decomposition, optimization, code generation, and execution. Additional steps, including data localization and global query optimization, are crucial between decomposition and optimization. Data localization identifies query fragments, utilizing horizontal and vertical fragmentation to break the global relation into refined fragments, producing an improved query.

In the context of search engines, query processing is integral to retrieving information from user input. Steps like decomposition, optimization, and execution parallel how search engines analyze user queries. Data localization aligns with a search engine's task of identifying relevant data fragments. Understanding query processing unveils the behind-the-scenes mechanisms powering efficient search engine operations.

A.1. User Query Input

The initial step in the user-query journey involves scanning and parsing, where the search engine meticulously examines the input to ensure it adheres to syntactic and semantic standards. This process not only ensures the user's query is correctly interpreted but also sets the stage for an enhanced user experience by understanding user intent even in the presence of typos or colloquial language.

A.2. Query Understanding

Building on the principles of data localization, the search engine dynamically navigates through vast datasets, identifying relevant sources and fragments that align with the user's search intent. This process mirrors the quest to localize and pinpoint data, enhancing precision in information retrieval and ensuring the user is presented with the most contextually relevant results.

A.3. Indexing and Fragmentation

Global query optimization becomes the engine's strategy to break down the user's query into manageable fragments, akin to the optimization steps in traditional query processing. In the realm of search engines, this involves leveraging sophisticated indexing mechanisms and algorithms. These tools act as the backbone, allowing the engine to navigate through massive datasets swiftly and efficiently, significantly enhancing the speed at which relevant information is retrieved.

A.4. Optimized Retrieval

As the user's query progresses through the optimization phase, the search engine refines its strategy for retrieving information. It employs advanced techniques like relevance ranking, considering factors such as user behavior, context, and quality of content. This ensures that the user is presented with the most pertinent and valuable information, aligning with the evolving expectations of a modern and discerning user base.

A.5. Result Presentation

The final stage of code generation and execution in the context of search engines focuses on translating the optimized query into an understandable and user-friendly format. The search engine generates executable code to fetch and present search results, emphasizing an aesthetically pleasing and intuitive user interface. This stage bridges the technical complexities of the search process with the user's expectation of a seamless and visually appealing presentation of information.

In essence, by aligning with the principles of traditional query processing, search engines can transform user queries into an optimized and efficient journey, enriching the user experience and ensuring the swift delivery of relevant and valuable information. This symbiotic integration of query processing principles and search engine mechanisms shapes the landscape of digital information retrieval in a dynamic and user-centric manner.

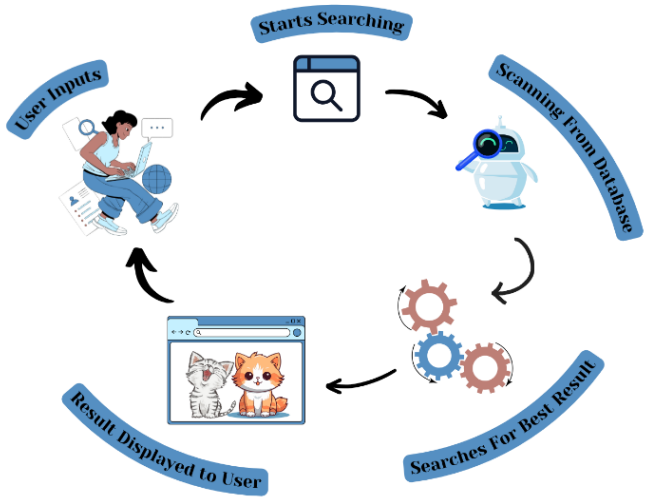

Fig.2.1. Query Processing Search Engine
(Source: Author's Documentation)

B. Modulo

Modulo, denoted by the symbol "mod," is a fundamental arithmetic operation representing the remainder when one integer is divided by another. It is expressed as the residual value of the division of a given integer (denoted by a) by another integer (denoted by m).

In mathematical terms, if we have two integers 'a' and 'm' (where m > 0), the operation a mod m, read as "a modulo m," yields the remainder when 'a' is divided by 'm.' This operation is defined by the equation a mod m = r, where 'r' is the remainder, and a can be expressed as the product of m and an integer quotient 'q' plus the remainder 'r.'

The modulus, denoted by 'm,' is crucial in this operation and determines the range of the result. The outcomes of arithmetic modulo m are confined to the set {0, 1, 2, ..., m – 1}. This set encompasses all possible remainders when an integer is divided by 'm,' creating a cyclical pattern where values repeat after reaching 'm - 1.'

Let's consider the example formula for the modulo operation:

$$a \bmod m = r$$

Where:

- a is the dividend (the integer being divided),
- m is the divisor (the modulus),
- r is the remainder.

This formula encapsulates the essence of modulo arithmetic, emphasizing its role in determining the remainder when 'a' is divided by 'm.' The resulting remainder 'r' is a vital component in various mathematical and computational processes, showcasing the broad utility of the modulo operation beyond simple arithmetic.

Modulo operations find versatile applications across various mathematical and computational domains. One notable application is in hash functions, where modulo is employed to confine hash codes within a specific range. This utilization ensures that the resulting hash values are bounded by the size of the hash table, providing an efficient and deterministic mapping of data.

C. Hash Function

In the ever-expanding landscape of data processing within local and global networks, the imperative for expedited data access and secure information exchange has driven computer scientists to seek innovative solutions. Among these, hash functions emerge as a cornerstone, working in conjunction with other security technologies to ensure both the speed and integrity of data transactions.

At its core, the verb "to hash," connoting the act of chopping or scrambling, encapsulates the essence of what hash functions accomplish. They intricately "scramble" data, transforming it into a numerical value of fixed length, irrespective of the input's original size. Commonly referred to as hashing algorithms or message digest functions, hash functions find versatile applications across computer science domains.

The properties of hash functions are designed with meticulous precision to ensure their efficacy in various applications. Firstly, they must be one-way, rendering it impossible to reverse the generation of a hash value back into the original data. This property is crucial for applications such as password storage, where irreversibility ensures enhanced security. [5]

Secondly, hash functions must be collision-free, meaning no two distinct input strings can produce the same output hash. This characteristic, synonymous with cryptographic hash functions,

safeguards the uniqueness of hash values and is integral to the prevention of unintended collisions.

Additionally, the speed of hash functions is paramount. To be effective, hash functions must compute hash values rapidly. In databases, where hash values are stored in hash tables to facilitate swift access, efficiency is critical.

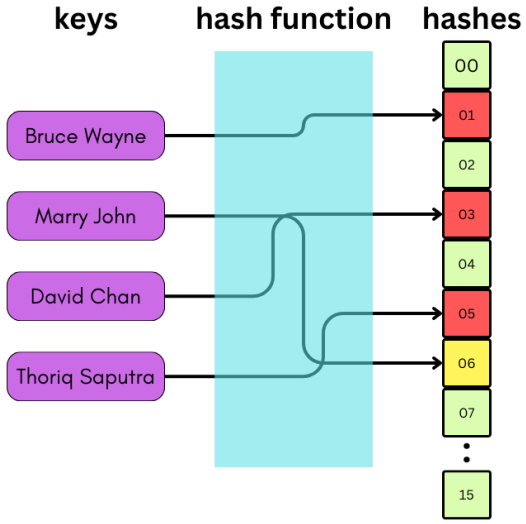

Fig.2.2. Hash Function Example
(Source: Author's Documentation)

Two primary heuristic methods exist for generating hash values: division and multiplication.

C.1. Mod Method

The mod method, a fundamental technique in crafting hash functions, offers a straightforward yet effective approach to mapping keys onto specific slots within a table. In this method, the hash function *h(key)* is defined as the remainder of the key when divided by the table size:

$$h(key) = key \ mod \ table\_size$$

This operation, denoted as *key* mod table_size or *key*% table_size, involves a single division operation, rendering hashing by division notably fast and computationally efficient.

However, to ensure the effectiveness of the division method, certain considerations come into play. It is advisable to avoid table sizes that are powers of a number *r*, as this would result in the hash function *h(key)* extracting only the lowest-order *p* bits of the key if *table_size* = r^p. This limitation could lead to an uneven distribution of keys if the low-order *p*-bit patterns are not equally likely.

Optimal results with the division method are often achieved when the table size is a prime number. Nevertheless, even when the table size is prime, an additional condition is imposed. If *r* represents the number of possible character codes on a computer, and *table_size* is a prime such that *r* mod table_size = 1, then the hash function *h*(*key*) = *key* mod table_size becomes a summation of the binary representation of the characters in the key, followed by a modulo operation with the table size.

For instance, let *r* = *256* and *table_size* = 17, wherein *r* mod table_size = 1 (i.e., *256* mod 17 = 1). Consequently, for the key 37599, the hash is computed as 37599 mod 17 = 12. Interestingly, for a different key, 573, the hash function yields the same result, 573 mod 17 = 12. This phenomenon, known as a collision, occurs when distinct keys produce identical hash values, highlighting a key challenge in hash function design.

To mitigate such collisions and enhance the overall robustness of the hash function, it is advisable to choose a prime table size that is not too close to an exact power of 2. This consideration contributes to a more even distribution of keys and minimizes the likelihood of clustering, thereby optimizing the effectiveness of the mod method in hash function implementation.

C.2. Multiplication Method

The multiplication method, a prominent technique in hash function design, introduces a strategic approach to transforming keys into hash values. In this method, a constant real number *c* within the range $0 < c < 1$ is multiplied by the key *k*, and the fractional part of $k \times c$ is extracted. The resulting value is then multiplied by the table size *m*, and the floor of the result is taken, represented by the equation:

$$h(k) = \text{floor}\ (m \times (k \times c \bmod 1))$$

Alternatively, it can be expressed as:

$$h(k) = \text{floor}\ (m \times (k \times c \bmod 1))$$

Here, the floor(*x*) function yields the integer part of the real number *x*, and (frac(*x*) = *x* - floor(*x*) yields the fractional part. An advantageous characteristic of the multiplication method is its flexibility regarding the choice of *m*, the table size. Typically, *m* is chosen to be a power of 2 ($m = 2^p$ for some integer *p*), facilitating easy implementation on most computers.

This method, grounded in mathematical precision and adaptability, showcases the nuanced considerations involved in crafting hash functions that are not only efficient but also resilient in diverse computational environments.

The use of hash functions extends across a myriad of applications in computer science, playing a pivotal role in ensuring data integrity, security, and expedited information retrieval. One prominent application lies in the encryption of communication between web servers and browsers, where hash functions contribute to generating secure session IDs for internet applications and facilitating data caching. Moreover, hash functions safeguard sensitive data, such as passwords, web analytics, and payment details, by converting them into irreversible and unique hash values. Digital signatures in emails benefit from hash functions, providing a means to verify the authenticity and integrity of electronic communications. Additionally, hash functions are instrumental in efficiently locating identical or similar data sets through lookup functions in databases, optimizing the speed of data access.

In conclusion, hash functions use precise design and mathematics to transform variable-length input into fixed-length hash values. Whether using the mod or multiplication method, the goal is to create one-way, collision-free, and efficient hash functions for diverse computer science applications. This exploration highlights the nuanced nature of hash function design, a crucial tool in modern computing adapting to the evolving landscape of data processing, security, and information

retrieval.

D. Rabin-Karp

The Rabin-Karp string matching algorithm introduces a significant advancement in the realm of pattern matching within a given text. It distinguishes itself from the Naive string-matching algorithm by employing a more strategic and efficient approach. Instead of scrutinizing every character in the initial phase, the Rabin-Karp algorithm harnesses the power of hash functions to streamline the pattern-matching process.

At its core, the algorithm calculates hash values for both the pattern and each M-character subsequence of the text. This calculated hash value serves as a unique identifier for the content within the pattern or substring, providing a rapid and effective means of comparison. The brilliance of Rabin-Karp lies in its ability to leverage these hash values, minimizing the need for extensive character-by-character matching and optimizing the overall search procedure.

To delve deeper into the steps of the algorithm, the initialization involves selecting a prime number 'p' as the modulus to prevent overflow and ensure a balanced distribution of hash values. The choice of a base 'b,' often a prime number corresponding to the character set size, further enhances the algorithm's efficiency. Setting the initial hash value 'hash' to zero marks the starting point for subsequent hash calculations.

The algorithm then proceeds to calculate the initial hash value for the pattern by iterating over each character from left to right. Each character's contribution to the hash value is determined by a formula that considers its position within the pattern. This step yields a hash value that uniquely represents the entire pattern.

The subsequent steps involve sliding the pattern over the text and continuously updating the hash value for each substring. As the pattern advances one position at a time, the algorithm efficiently adjusts the hash value by considering the contributions of the outgoing and incoming characters.

The distinctive feature of Rabin-Karp becomes particularly evident in the search phase. Iterating through the text, the algorithm calculates hash values for substrings of a specified length. When a match is potentially detected (i.e., the hash values of the pattern and substring coincide), the algorithm performs a more granular character-by-character comparison to confirm the match. If confirmed, the starting index of the substring is stored as a valid answer.

In essence, the Rabin-Karp algorithm capitalizes on the strengths of hash functions to expedite the pattern-matching process. By transforming the content into hash values and judiciously comparing these values, Rabin-Karp significantly reduces the computational load associated with exhaustive character matching, making it a powerful and efficient tool for various text processing applications.

D.1. Rabin-Karp Steps [10]

1. Initialize Hash Values
   - Choose a prime number 'p' as the modulus to avoid overflow and ensure a good distribution of hash values.
   - Select a base 'b,' typically a prime number and often the size of the character set (e.g., 256 for ASCII characters).
   - Set an initial hash value 'hash' to 0.
2. Calculate Initial Hash Value for the Pattern
   - Iterate over each character in the pattern from left to right.
   - For each character 'c' at position 'i', calculate its contribution to the hash value as,

$$(c \times b^{pattern\_length-i-1} \bmod p)$$

   then add it to 'hash.' This yields the hash value for the entire pattern.
3. Slide the Pattern Over the Text
   - Calculate the hash value for the first substring of the text that is the same length as the pattern.
4. Update the Hash Value for Each Subsequent Substring
   - To slide the pattern one position to the right, remove the contribution of the leftmost character and add the contribution of the new character on the right.
   - The formula for updating the hash value when moving from position 'i' to 'i+1' is:

$$hash = (hash - (text[i - pattern\_length] \times (b^{pattern\_length-1} \bmod p \times b + text[i]$$

5. Search for Pattern Matches
   - Start iterating from the beginning of the string.
   - For each substring of length 'm,' calculate its hash value.
   - If the hash value of the current substring matches the hash value of the pattern:
     - Perform a character-by-character comparison to confirm the match.
     - If the characters match, store the starting index as a valid answer.
   - Continue iterating for the next substrings.
6. Return Valid Starting Indices
   - Return the starting indices of the substrings where matches were confirmed as the required answer.

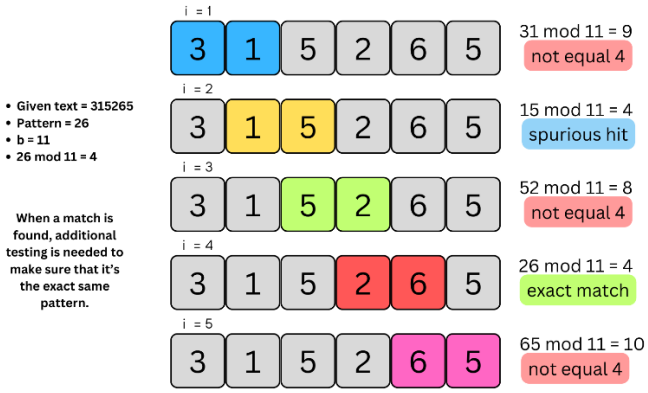

Fig.2.3. Rabin-Karp Process
(Source: Author's Documentation)

These steps help us understand how the Rabin-Karp algorithm works to find words or patterns in a block of text. Instead of checking each letter one by one, it uses a clever math trick called a hash function to quickly compare chunks of text and the pattern we're looking for. This way, it can find matches faster and more efficiently.

D.2. Application of Rabin-Karp

The Rabin-Karp algorithm finds practical applications in various areas where efficient pattern matching is crucial. One notable use is in text processing and search engines, where it aids in quickly identifying and locating keywords or phrases within large volumes of textual data. Additionally, the algorithm is employed in plagiarism detection systems, helping to detect similarities between documents or pieces of text. In bioinformatics, Rabin-Karp finds utility in DNA sequence matching and analysis, contributing to genetic research and identification of genetic patterns. Moreover, the algorithm has been applied in network security for intrusion detection, recognizing patterns indicative of potential threats or attacks. Its versatility and efficiency make the Rabin-Karp algorithm a valuable tool across domains where pattern matching is a fundamental requirement.

III. IMPLEMENTATION

A. Search Country

The Search Country Engine is a purpose-built application designed to streamline the exploration of an extensive dataset containing detailed information about countries. Rooted in the efficiency of the Rabin-Karp algorithm, the search engine is crafted to provide users with a swift and accurate means of discovering relevant country-related data. The dataset encompasses a variety of attributes, including country names, capitals, populations, and geographical details, making it a comprehensive resource for users seeking diverse information. The Search Country Engine prioritizes a user-friendly experience, allowing individuals to effortlessly retrieve information about specific countries or explore nations that match their search criteria. This section will delve into the specifics of data collection, preprocessing steps, and the distinctive features of the Search Country Engine, showcasing its capabilities in empowering users to seamlessly navigate and extract valuable insights from the rich dataset.

B. Country Data

```
countries = {
            "USA": {"capital": "Washington, D.C.", "population": 331002651, "area_sq_km": 9833517},
            "Canada": {"capital": "Ottawa", "population": 37742154, "area_sq_km": 9976140},
            "India": {"capital": "New Delhi", "population": 1380004385, "area_sq_km": 3287263},
            "UK": {"capital": "London", "population": 67886011, "area_sq_km": 243610},
            "China": {"capital": "Beijing", "population": 1444216107, "area_sq_km": 9706961},
            "Japan": {"capital": "Tokyo", "population": 126476461, "area_sq_km": 377975},
            "South Korea": {"capital": "Seoul", "population": 51269185, "area_sq_km": 100363},
            "Thailand": {"capital": "Bangkok", "population": 69799978, "area_sq_km": 513120},
            "Afghanistan": {"capital": "Kabul", "population": 38928346, "area_sq_km": 652230},
            ....
            }
```

Fig.3.1. Countries Dataset
(Source: Author's Documentation)

The provided dataset is a Python dictionary containing key-value pairs representing information about different countries. Each country, denoted as a key (e.g., "USA," "Canada"), is associated with an inner dictionary encapsulating details such as its capital city, population count, and land area in square kilometers. For instance, the entry for "USA" indicates that its capital is "Washington, D.C.," the population is 331,002,651, and the land area is 9,833,517 square kilometers. This structured format allows for organized storage and retrieval of essential country-specific data, making it suitable for various applications such as data analysis or the development of a search engine focused on countries. Additional countries can seamlessly be incorporated into the dataset using the same structure.

C. Hash Function

```
def hash_function(self, text, prime_mod):
    result = 0
    for char in text:
        result = (result * 256 + ord(char)) % prime_mod
    return result
```

Fig.3.2. Hash Function
(Source: Author's Documentation)

The 'hash_function' Python function employs a rolling hash mechanism to efficiently compute hash values for variable-length strings. It iterates through each character in the input text, updating the result by multiplying the current value by 256 and adding the Unicode code point of the character. The result is then taken modulo 'prime_mod' to constrain the hash value. This approach, utilizing a base of 256 for compatibility with ASCII characters, generates a numerical representation of the input text, making it suitable for tasks like hash table implementations or pattern matching algorithms such as Rabin-Karp.

D. Rabin-Karp Algorithm

```
def rabin_karp_search(self, pattern, text):
    prime_mod = 11
    pattern_hash = self.hash_function(pattern, prime_mod)
    pattern_length = len(pattern)
    text_length = len(text)

    for i in range(text_length - pattern_length + 1):
        substring = text[i:i + pattern_length]
        substring_hash = self.hash_function(substring, prime_mod)

        if pattern_hash == substring_hash and pattern == substring:
            return i

    return -1
```

Fig.3.3. Rabin-Karp Function
(Source: Author's Documentation)

The 'rabin_karp_search' Python function implements the Rabin-Karp string matching algorithm for searching a pattern within a given text. It employs a rolling hash technique using the 'hash_function'. The function initializes a prime modulus 'prime_mod' and calculates the hash value for the input pattern. It then iterates through the text, computing hash values for overlapping substrings of the same length as the pattern. If a match is found between the hash values and the actual substrings, indicating a potential pattern occurrence, the function returns the starting index of the match. If no match is found in the entire text, the function returns -1. This algorithm optimizes string matching by leveraging hash values, offering a more efficient alternative to exhaustive search methods.

E. Other Functions

```python
def search_country(self, query):
    query_lower = query.lower()
    matching_countries = []

    for country, info in self.countries.items():
        country_lower = country.lower()
        index = self.rabin_karp_search(query_lower, country_lower)
        if index != -1:
            if country_lower.startswith(query_lower):
                matching_countries.append((country, info))

    if matching_countries:
        country_count = len(matching_countries)
        if country_count == 1:
            print("\n ===== Country Found =====")
        else:
            print("=== Matching Countries ===")
            print(f"[{country_count} countries found]")
        for country, info in matching_countries:
            # Display matching countries
            self.display_country_info(country, info)
        print("\n==========================")
    else:
        print("Country not found.")
```

Fig.3.4. Search Country Function
(Source: Author's Documentation)

The 'search_country' Python method performs a case-insensitive search for countries that match a given query within a dataset. It initializes a lowercased version of the query and iterates through the countries in the dataset, applying the Rabin-Karp string matching algorithm to efficiently identify matches. If a match is found, the method verifies if the country name starts with the query to ensure accurate results. Matching countries and their corresponding information are appended to a list. The method then prints the results, displaying the found countries and their details, or indicating that no matches were found. This function provides a user-friendly interface for querying and retrieving information about countries, utilizing the Rabin-Karp algorithm for an optimized search experience.

```python
def display_country_info(self, country, info):
    print(f"\nCountry: {country}")
    print(f"Capital: {info['capital']}")
    print(f"Population: {info['population']}")
    print(f"Area (sq km): {info['area_sq_km']}")
```

Fig.3.5. Display Country Info Function
(Source: Author's Documentation)

The 'display_country_info' Python method takes a country name and its corresponding information as input and prints formatted details about the country. It prints the country name, capital, population, and land area in square kilometers. This function is designed to enhance the readability of the information displayed to the user, providing a clear and organized presentation of key country attributes. It is typically used in conjunction with other functions, such as in the 'search_country' method, to present detailed information about matching countries in a user-friendly manner.

```python
def run_search_engine(self):
    print("""
            SEARCH ENGINE ART
    """)
    while True:
        user_input = input("Enter a country name (or 'exit' to quit): ").strip().lower()

        if user_input == 'exit':
            print("Exiting the search engine. Goodbye Bang!")
            break

        if user_input:
            self.search_country(user_input)
        else:
            print("Please enter a valid country name.")
```

Fig.3.6. Search Engine Runer
(Source: Author's Documentation)

The 'run_search_engine' Python method initiates a simple command-line search engine for querying and retrieving information about countries. It displays an ASCII art title and enters a continuous loop prompting the user to input a country name. The user can type 'exit' to terminate the search engine. For valid inputs, the method invokes the 'search_country' function, which performs a case-insensitive search for matching countries in the dataset and displays relevant details. If the user provides an empty input, a prompt for a valid country name is issued. The loop continues until the user chooses to exit, providing an interactive and visually appealing interface for exploring country information.

F. User Interface and Program Output

Fig.3.7. Beginning of The Program
(Source: Author's Documentation)

```
Enter a country name (or 'exit' to quit): Bekasi

[Country not found]
```

Fig.3.8. Program Output Country Not Found
(Source: Author's Documentation)

```
Enter a country name (or 'exit' to quit):

[Please enter a valid country name]
```

Fig.3.9. Program Output Input is BLANK
(Source: Author's Documentation)

```
Enter a country name (or 'exit' to quit): Indo

 ===== Country Found =====

Country: Indonesia
Capital: Jakarta
Population: 273523615
Area (sq km): 1904569

==========================
```

Fig.3.10. Program Output of Country Detail
(Source: Author's Documentation)

Fig.3.11. Program Output of Countries Details
(Source: Author's Documentation)

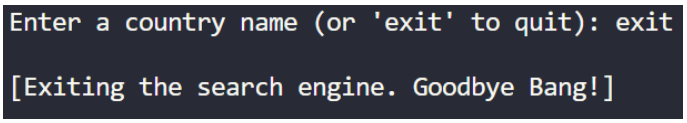

Fig.3.12. Program Output Exiting
(Source: Author's Documentation)

IV. Conclusion

In conclusion, the development of the "Search Country" program represents a successful integration of advanced algorithms into a practical application. By implementing the Rabin-Karp string matching algorithm and hash function, the program enables users to input a country name and promptly retrieve detailed information. The utilization of the Rabin-Karp algorithm enhances search efficiency, optimizing the process of matching user queries with country names in the dataset. Additionally, the hash function contributes to the overall speed and performance of the program by efficiently converting variable-length strings into numerical representations. This successful implementation underscores the importance of algorithmic strategies in real-world applications, showcasing their impact on user experience and program functionality. Overall, the "Search Country" program exemplifies the seamless integration of theoretical concepts into practical solutions, demonstrating the potential for algorithmic innovation in program development.

V. Acknowledgment

I would like to express my deepest gratitude to God, the source of all knowledge and wisdom, for granting me the strength and inspiration to complete this paper. My sincere appreciation extends to my esteemed lecturer, Dr. Ir. Rinaldi Munir, M.T., for his invaluable guidance, unwavering support, and insightful feedback throughout the research and writing process. I am also grateful to Monterico Adrian, S.T., M.T., for his assistance and encouragement. Their expertise and dedication have been instrumental in shaping this paper. I am truly blessed to have had the opportunity to learn and grow under their mentorship.

References

[1] A. D. Sawarkar and J. M. Waghmare, "Query Processing and Query Optimization in Distributed Database: A Survey," April 2014.
[2] "Bidadari Surga." Modulo: Konsep dan Penggunaannya dalam Matematika, May 2023, https://bidadarisurga.com/modulo. Accessed 8 December 2023.
[3] Munir, Rinaldi. "Teori Bilangan Bagian 1" Matematika Diskrit, 2023. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/matdis23-24.htm. Accessed 8 December 2023.
[4] Munir, Rinaldi. "Teori Bilangan Bagian 3" Matematika Diskrit, 2023. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/matdis23-24.htm. Accessed 8 December 2023.
[5] Malviya, Nitesh. "Introduction to Hash Functions." Infosec Institute, November 2020, https://resources.infosecinstitute.com/topics/cryptography/introduction-to-hash-functions/. Accessed 8 December 2023.
[6] 1&1 IONOS. "Hash Function: How Securely Do They Protect Data?" October 2020, https://www.ionos.com/digitalguide/server/security/hash-function/. Accessed 8 December 2023.
[7] ranadeepika2409. "What are Hash Functions and How to Choose a Good Hash Function?" GeeksforGeeks, n.d., https://www.geeksforgeeks.org/what-are-hash-functions-and-how-to-choose-a-good-hash-function/. Accessed 8 December 2023.
[8] javatpoint.com. "DAA: Rabin-Karp Algorithm." JavaTpoint, https://www.javatpoint.com/daa-rabin-karp-algorithm. Accessed 8 December 2023.
[9] programiz.com. "Rabin-Karp Algorithm." Programiz, https://www.programiz.com/dsa/rabin-karp-algorithm. Accessed 8 December 2023.
[10] GeeksforGeeks. "Rabin-Karp Algorithm for Pattern Searching." GeeksforGeeks, https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/. Accessed 8 December 2023.

Statement

I hereby declare that the paper I have written is my own work, not a reproduction or translation of someone else's paper, and it is free from plagiarism.

Bandung, 9 Desember 2023

Ahmad Thoriq Saputra, 13522141