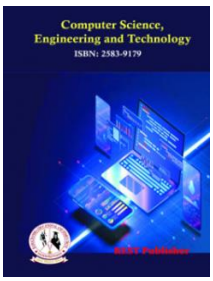
Computer Science, Engineering and Technology

Vol: 3(2), June 2025

REST Publisher; ISSN: 2583-9179

Website: https://restpublisher.com/journals/cset/

DOI: https://doi.org/10.46632/cset/3/2/20



Multi-Criteria Decision Making for Data Structure Selection Using the TOPSIS Method

***Penta Surya Prakash Rao, Sonu Kumar, Deepak Kumar Adhikari**

Netaji Subhas University Jamshedpur, Jharkhand, India.

Corresponding author email: surya164264@gmail.com

Abstract: *An introduction serves as the opening section of a piece of writing, setting the tone and context for the content that follows. It typically includes a brief overview of the main topic or subject, offering background information or outlining the purpose of the work. A strong introduction captures the reader's attention, provides essential context, and presents a clear thesis or argument that will be explored in the body of the text. It also helps to establish the writer's voice and gives a preview of the structure or approach the writing will take. The research significance section highlights the importance of a study and its potential impact on the field. It explains why the research is necessary, how it fills existing gaps in knowledge, and its relevance to current issues or problems. This section also emphasizes the contribution the study will make, whether through new insights, practical applications, or theoretical advancements. It addresses the broader implications for society, industry, or academia, showing how the findings could inform future research or policy decisions. Ultimately, it helps readers understand the value and potential outcomes of the research being conducted. The methodology section outlines the approach used to conduct research or a study. It details the procedures, techniques, and tools applied to gather and analyze data. This section typically includes the Study design, participant selection methods, data collection procedures, and evaluation techniques. It ensures the study's reliability and validity by explaining why specific methods were chosen and how they were implemented. By providing this level of detail, the methodology allows others to replicate or critically assess the research process. In essence, it explains* how *the research was conducted, ensuring transparency and rigor in the study. Alternative taken as Harry, Linked List, Stack, Queue, Hash Table, Graph. Evaluation preference taken as Memory Usage, Lookup Time, Insertion time, Deletion Time. Hash Table is getting first place of the table and Graph is getting last place of the table.*

Keywords: *Harry, Linked List, Stack, Memory Usage, Lookup Time, Insertion time*

1. INTRODUCTION

External memory (EM) algorithms and data structures are developed to improve data management and movement. Some researchers refer to them as I/O algorithms or out-of-core algorithms. A widely used technique in this domain is the bootstrapping paradigm, which uses weight-balanced B-trees as its underlying data structure. This method combines several static data structures, each designed for constrained search ranges, into a dynamic framework that solves the entire problem. However, because bootstrapping relies on specific data structures, this section explores an alternative strategy that focuses on the intrinsic properties of the problem rather than the data structure. [1] We propose a data structure to tackle the dynamic trees problem and introduce two of its variants. In Section 3, we summarize an initial version of our data structure and conduct a preliminary analysis of its running time. The main idea presented in this section involves partitions, where each tree is transformed into a set of independent partitions. As a result, when documenting how all components within an assembly are interconnected, it is necessary to create a data structure that retains both the topological and geometric properties of each component. [2] The data structure proposed in this paper offers several advantages. First, it determines the transformation matrices of components within an assembly by using the mating feature information between them. As discussed in the previous section, the data structure of Lieberman and Wesley, along with Eastman system, depend on each component's transformation matrix as input to determine its position and orientation within an assembly. [4] A data structure should have multiple versions. If we allow access to these multiple versions, we define the data structure as static. This data structure would be formed by a linked structure (or a more general form) that includes each version of the ephemeral structure, allowing access or update steps in any version to be reflected in the corresponding section of the persistent structure. [5] Achieving simultaneous elegance,

performance, and synchronization in functional programs that process extensive data structures. is challenging. This article focuses on the challenges of data structures. We begin by highlighting the difficulties of dealing with Functional language data structures were introduced with an alternative approach called I-structures. Our method involves testing different applications and comparing their implementations using functional data structures and I-structures. Start by exploring two data-structure operations that are traditional in functional languages. We aim to demonstrate the disadvantages and limitations for fine-grained operational data structure algorithms such as optimizations, especially in a parallel computing environment. [6] We introduce a set of innovative data structures designed for efficient management of various continuous and discrete properties of mobile systems. We refer to our data structures as dynamic, to distinguish them from traditional static or dynamically updated structures in the conventional sense. Ž We clarify their counterparts in the following text and use an abbreviated form for reference. We abbreviate the "operational data structure" as KDS. The fundamental characteristic of KDS is that its certificates provide ongoing proof of the correctness of the current structural operation. In short, our dynamic data structures differ from traditional dynamic data structures. While they can handle insertions and deletions, their primary focus is on continuous movement rather than discrete changes. [7] We introduce an extension of Hoare's logic to analyze programs that modify data structures. Our approach relies on a low-level storage model. using a heap with functions for searching, updating, allocation, and deallocation. This article builds on previous research on reasoning about data structures by Bur stall, Reynolds, Ishtar, and O'Meara. began by highlighting the primary challenge in verifying pointer programs: capturing Intuitive local reasoning or textbook style explanations of data structures. That programmers use. A specification and verification can only focus on memory cells. accessed by the program, without requiring a global view. [9] A new data structure is introduced by restructuring The correction equation is used in the proposed approach, which, compared to the conventional Newtonian data structure, reduces the number of complements by almost half and reduces CPU time by approximately 15% for large-scale systems. [14] The experimental The framework for these data structures is built on multi-temporal conjoint analysis (Coppin & Bauer, 1996). classification algorithm to a multi-date image to detect pixels with spectral characteristics that indicate sudden vegetation loss. Choosing an appropriate data structure is important for several reasons. Different data transformations behave differently for compositional properties (Cohen et al., 2001), and in the context of disturbance detection, it is necessary to use transformations that improve spectral distance and improve The difference between disturbed and undisturbed forests conditions. [17] The V-E and F-E data structures enable efficient access procedures and are structured to represent edges based on their topological proximity rather than as individual entities. This method eliminates the need for additional data processing to determine the representation of an edge's side or endpoint. [18] We introduce a new scheme for unstructured data. Unlike traditional schemas that resemble programming language type systems, unstructured data often has fewer constraints, requiring an alternative approach to expressing data constraints. We propose to represent both data and schema as edge-labeled graphs. In addition, we define the correspondence between a graph database and a graph scheme and prove that there is a natural, efficiently computable ordering between graph schemes. [19] Although various Various data structures have been introduced for feature indexing. they cannot scale effectively beyond 10–15 dimensions. This article introduces the Hybrid Tree, a multidimensional data structure developed for indexing high-dimensional feature spaces. Robust indexing techniques for high-dimensional spaces are essential to enable efficient similarity searches in a database system. Typically, feature spaces are indexed using multidimensional data structures, where similarity searches correspond to range searches in these structures. [20] The entity-relationship model is introduced as a data model that captures the main semantic details of the real world. A special graph-based approach is presented as a tool for database design. Keywords: database design, logical data representation, data semantics, data models, entity-relationship model, relational model, database task force, network model, entity set model, data definition and manipulation, data integrity and consistency. [23] Traditional file structures that implement multicity access, such as inverted files, are extensions of single-key access designs and exhibit various limitations, especially for high-performance files. This study investigates dynamic file structures that eliminate the distinction between primary and secondary keys and treat all keys symmetrically. Keywords: file structures, databases, dynamic storage allocation, multicity search, multidimensional data. The result of this approach is a symmetric and mutable file structure. Symmetric means that every key field is considered a primary key, while mutable, dynamically adjusts its shape based on the stored content, ensuring uniform bucket occupancy and access time, even when the data distribution across the data space is highly irregular. [27]

2. MATERIAL AND METHOD

Harry: An array is an organized collection of elements, usually of the same type, stored in contiguous memory locations.

Linked List: A linked list is an ordered data structure consisting of nodes, where each node contains data and a reference (or pointer) to the next node. allowing efficient insertion and deletion operations.

Stack: A stack is a structured collection of elements that operates on a last-in, first-out (LIFO) principle, allowing additions and removals only from the top. It is commonly used to handle function calls, undo instructions, and other operations that require reverse-order processing.

Queue: A queue is a structured A collection of elements that follows the first-in, first-out (FIFO) principle, in which items are added at the back and removed from the front. ensuring orderly processing.

Hash table: A hash table is a data structure that stores and manages key-value pairs. by assigning keys to specific codes in an array using a hash function. enabling fast access Insertion and deletion processes**.**

graph: armpit graph refers armpit descriptive label that summarizes the graph's content, providing context and helping the viewer understand what data or information is being presented

Evaluation preference:

Memory Usage: Memory usage refers to the portion of a computer's RAM that is consumed by programs, processes, and data. Effective memory management is essential to maintain performance and avoid slowdowns or system crashes caused by excessive consumption

Lookup Time: Seek time refers to the amount of time required access and retrieve data from A data structure such as an array or dictionary. It's crucial for performance, especially in large datasets or complex queries.

Insertion time: insertion time is The amount of time required to add an element to a data structure. like an array or list. It depends on the structure's type and its current state.

Deletion Time: Deletion time refers to the period it takes for data or information to be completely erased from a system, ensuring no recoverable traces remain, often used in digital security and data management

TOPSIS Method: Multi-criteria decision analysis (MCDA) or multi-criteria decision making (MCDM) is a well-established area of operations research that focuses on developing mathematical and computational techniques that support the subjective evaluation of decision alternatives based on multiple performance criteria. This evaluation can be performed by an individual decision maker or a group. Among the various MCDA/MCDM methods designed for real-world decision making, the Order Prioritization by Ideal Solution (TOPSIS) technique is widely used in many domains. One of the most common applications of TOPSIS is in supply chain management and logistics, addressing key aspects such as supplier selection, transportation, and location analysis. [1] his article presents a new approach based on the technique for order preference selection for unity. Ideal Solution (TOPSIS) for ranking and comparing algorithms. By running an algorithm for R methods with a sufficiently large value of R, the average result distribution can be approximated using a Gaussian distribution, which enables the application of Within the context of Hellinger-tops is for direct ranking algorithms. algorithm comparison, alternatives represent different algorithms, while criteria correspond to criterion evaluations. Simulation results confirm The performance of A-TOPSIS in establishing the ranking of evaluated algorithms. [2] This research paper aims to provide an overview of the developments in fuzzy TOPSIS methods. It begins with a literature review, examining various fuzzy models used in decision making. Finally, it highlights several applications of fuzzy TOPSIS. [3]

3. RESULT AND DISCUSSION

TABLE 1. Data structures

	Memory Usage	Lookup Time	Insertion Time	Deletion Time
Array	8	4	8	6
Linked List	7	3	8	7
Stack	8	2	9	8
Queue	9	4	7	6
Hash Table	9	7	10	7
Graph	6	3	6	7

The table provides a comparison of different data structures based on four performance metrics: Memory Usage, Lookup Time, Insertion Time, and Deletion Time. This represents the amount of memory each data structure requires. The Hash Table and Queue require the most memory (9 units), while the Graph requires the least (6 units). Memory consumption often correlates with the complexity of the structure and the data it stores. This is the time taken to search for an element in the structure. The Stack has the fastest lookup time (2 units), while the Hash Table takes the longest

(7 units). Stacks are simpler structures, which makes lookup faster, while Hash Tables require more time due to hashing mechanisms. This shows how long it takes to add an element. The Stack takes the least time (2 units), while the Hash Table takes the most (10 units). Stack operations are generally more efficient, as they only require adding to the top, while Hash Tables might involve rehashing. This indicates how quickly an element can be removed. The Stack again has the fastest deletion time (8 units), while the Queue has the slowest (6 units). Deletion time depends on the structure's complexity and how data is stored.

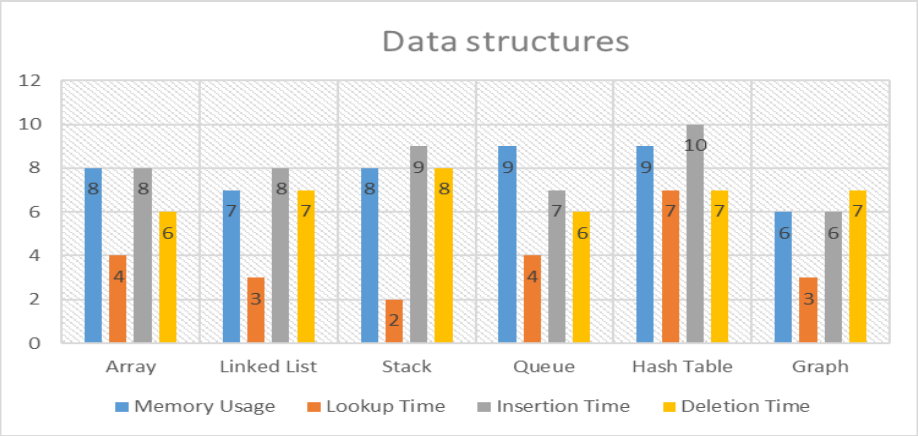


FIGURE 1. Data structures

The table presents a comparison of data structures based on four key performance metrics: Memory Usage, Lookup Time, Insertion Time, and Deletion Time. The Hash Table scores the highest in Lookup (7) and Insertion (10), making it highly efficient for quick data retrieval and storage. The Queue excels in Memory Usage (9) but has moderate performance in other aspects. Stacks and Arrays also perform well in Insertion but vary in Lookup and Deletion times. Graphs have the lowest Memory Usage (6) and moderate performance across all metrics. This analysis highlights the strengths and trade-offs of each data structure in different operations.

TABLE 2. Sort & Sum

	Memory Usage	Lookup Time	Insertion Time	Deletion Time
Array	64	16	64	36
Linked List	49	9	64	49
Stack	64	4	81	64
Queue	81	16	49	36
Hash Table	81	49	100	49
Graph	36	9	36	49

The table presents the memory usage and time complexities associated with different data structures for operations like lookup, insertion, and deletion. This refers to the amount of space the data structure occupies in memory. The array and hash table consume more memory (64 and 81, respectively) compared to others like linked lists (49) and graphs (36). This is the time taken to retrieve an element from the data structure. Arrays (16) and linked lists (9) have relatively low lookup times, while hash tables (49) take longer, potentially due to their hashing mechanism This indicates how long it takes to add an element. Stacks have the highest insertion time (81), while linked lists and arrays show moderate times (64 and 64). Deletion time shows how long it takes to remove an element. Stacks (64) and queues (36) have varying deletion times, with hash tables having moderate times (49). Linked lists are slightly slower in deletion (49), likely due to the need to traverse the structure.

TABLE 3. Normalized data

	Memory Usage	Lookup Time	Insertion Time	Deletion Time
Array	0.413118	0.394132	0.403034	0.475551
Linked List	0.361478	0.295599	0.403034	0.475551
Stack	0.413118	0.197066	0.453413	0.534994
Queue	0.464758	0.394132	0.352655	0.416107
Hash Table	0.464758	0.68973	0.503793	0.594438
Graph	0.309839	0.295599	0.302276	0.356663

The table provides normalized data for various data structures across four operations: memory usage, lookup time, insertion time, and deletion time. The values represent the relative performance of each data structure in comparison to others, where a higher value typically indicates a greater cost in terms of that operation. Arrays and stacks have similar memory consumption (0.413118), while linked lists use slightly less memory (0.361478). Graphs have the least memory usage (0.309839), reflecting their typically more complex, dynamic nature. Hash tables and queues use more memory (0.464758), likely due to the need for additional storage or indexing. Hash tables show the highest lookup time (0.68973), which is unusual compared to their typical constant-time lookup performance in real-world scenarios. Stacks (0.197066) are the fastest for lookup, followed by arrays (0.394132), linked lists (0.295599), and queues (0.394132). Queues have the shortest insertion time (0.352655), while stacks (0.453413) and arrays (0.403034) have slightly longer times. Linked lists and hash tables exhibit the same insertion time (0.403034), indicating similar performance for insertion in these structures. Stack (0.534994) and hash table (0.594438) have the highest deletion times, suggesting these structures are less efficient in deletion. Arrays, linked lists, and queues have similar deletion times (0.475551, 0.475551, and 0.416107 respectively), indicating consistent deletion performance.

TABLE 4. Weight

Array	0.25	0.25	0.25	0.25
Linked List	0.25	0.25	0.25	0.25
Stack	0.25	0.25	0.25	0.25
Queue	0.25	0.25	0.25	0.25
Hash Table	0.25	0.25	0.25	0.25
Graph	0.25	0.25	0.25	0.25

The content appears to list Various data structures including arrays, linked lists, stacks, queues, hash tables, and maps. alongside corresponding weights of 0.25 for each in four different contexts, which could represent different criteria or assessments (such as time complexity, space complexity, ease of implementation, etc.). A collection of elements stored in contiguous memory locations, offering constant time access (O (1)) for indexed elements, though insertion and deletion can be expensive (O(n)). A linear collection of elements, where each element points to the next. It allows efficient insertion and deletion (O(1)), but accessing an element requires O(n) time. A collection following the Last In First Out (LIFO) principle. It allows Implements efficient push and pop operations (O (1)), while allowing random element access is not possible.A collection following the First In First Out (FIFO) principle. Similar to stacks, it provides efficient enqueue and dequeue operations (O(1)) but does not support random access. A data structure that stores key-value pairs with an average time complexity of O(1) for search, insert, and delete operations, assuming no collisions. A collection of nodes (vertices) connected by edges. It can represent complex relationships and allows efficient traversal with algorithms like BFS and DFS.

TABLE 5. Weighted normalized decision matrix

	Memory Usage	Lookup Time	Insertion Time	Deletion Time
Array	0.10328	0.098533	0.100759	0.118888
Linked List	0.09037	0.0739	0.100759	0.118888
Stack	0.10328	0.049266	0.113353	0.133749
Queue	0.11619	0.098533	0.088164	0.104027
Hash Table	0.11619	0.172433	0.125948	0.14861
Graph	0.07746	0.0739	0.075569	0.089166

The table provided is a Weighted Normalized Decision Matrix that compares the performance of various data structures (Array, Linked List, Stack, Queue, Hash Table, and Graph) across four operations: Memory Usage, Lookup Time, Insertion Time, and Deletion Time. Each of these data structures is evaluated on these metrics, where the values represent normalized performance measures, typically scaled between 0 and 1. Lower values indicate better performance in a given operation. The memory required by the data structure. The time taken to access a specific element in the data structure. The time it takes to add an element to the structure. The time it takes to remove an element from the structure.

TABLE 6. Positive matrix

	Memory Usage	Lookup Time	Insertion Time	Deletion Time
Array	0.1162	0.1724	0.1259	0.1486
Linked List	0.1162	0.1724	0.1259	0.1486
Stack	0.1162	0.1724	0.1259	0.1486
Queue	0.1162	0.1724	0.1259	0.1486
Hash Table	0.1162	0.1724	0.1259	0.1486
Graph	0.1162	0.1724	0.1259	0.1486

The table presents a comparison of different data structures in terms of various performance metrics: Memory Usage, Lookup Time, Insertion Time, and Deletion Time. The data structures listed include Array, Linked List, Stack, Queue, Hash Table, and Graph. All the data structures have the same memory usage of 0.1162, indicating that they require an equal amount of memory for storage. However, the memory usage of a data structure can vary based on implementation and size of data being stored, but here, they are considered equivalent. This refers to the time it takes to search or retrieve an element from the data structure. All the structures have the same lookup time of 0.1724. This suggests that, in this comparison, searching an element has similar efficiency across all listed data structures The time it takes to insert an element into the data structure is also identical for all structures, at 0.1259. This indicates that inserting elements is equally fast across these data structures. Finally, the deletion time, which measures how long it takes to remove an element, is again the same for all data structures at 0.1486.

TABLE 7. Negative matrix

	Memory Usage	Lookup Time	Insertion Time	Deletion Time
Array	0.0775	0.1724	0.1259	0.1486
Linked List	0.0775	0.1724	0.1259	0.1486
Stack	0.0775	0.1724	0.1259	0.1486
Queue	0.0775	0.1724	0.1259	0.1486
Hash Table	0.0775	0.1724	0.1259	0.1486
Graph	0.0775	0.1724	0.1259	0.1486

The table compares the performance of different data structures (Array, Linked List, Stack, Queue, Hash Table, and Graph) in terms of four key metrics: Memory Usage, Lookup Time, Insertion Time, and Deletion Time. All the data structures listed have identical values for each of these metrics. Each data structure consumes the same amount of memory, with a value of 0.0775. This suggests that, in terms of memory consumption, there is no significant difference between any of the listed data structures. The time taken to look up or access an element in each data structure is identical at 0.1724. This indicates that the speed at which an element is retrieved is the same across all the structures in this comparison. The time taken to insert an element into each data structure is also the same at 0.1259, suggesting that each structure performs insertion operations with similar efficiency. Similarly, the time for deletion operations is uniform across all structures at 0.1486, meaning the time taken to remove an element is the same for each data structure.

TABLE 8. Si Positive, Si Negative and Ci value

	Si Positive	Si Negative	Ci
Array	0.084532	0.279852	0.768013
Linked List	0.109056	0.286469	0.724275
Stack	0.125364	0.309389	0.711643
Queue	0.094215	0.268726	0.740413
Hash Table	0	0.299732	1
Graph	0.131455	0.262267	0.666122

The table provided compares different data structures (Array, Linked List, Stack, Queue, Hash Table, and Graph) based on three variables: Si Positive, Si Negative, and Ci. the positive aspect or benefit of the data structure. For instance, an Array has a value of 0.084532, indicating its positive characteristics. The Graph has the highest value for Si Positive at 0.131455, reflecting potentially better benefits for certain use cases. measures the negative aspects or limitations of each data structure. The Linked List, for example, has a value of 0.286469, showing a significant negative tradeoff compared to others like Hash Table (with 0). refers to a comparative index, which might represent a metric of overall efficiency or suitability for particular operations. Hash Table has a Ci value of 1, indicating it's considered the most effective overall, likely due to its efficiency in handling key-value pair data and offering constant-time operations on average. Arrays and Linked Lists, on the other hand, have lower Ci values, signifying they might be less efficient or versatile for some operations.

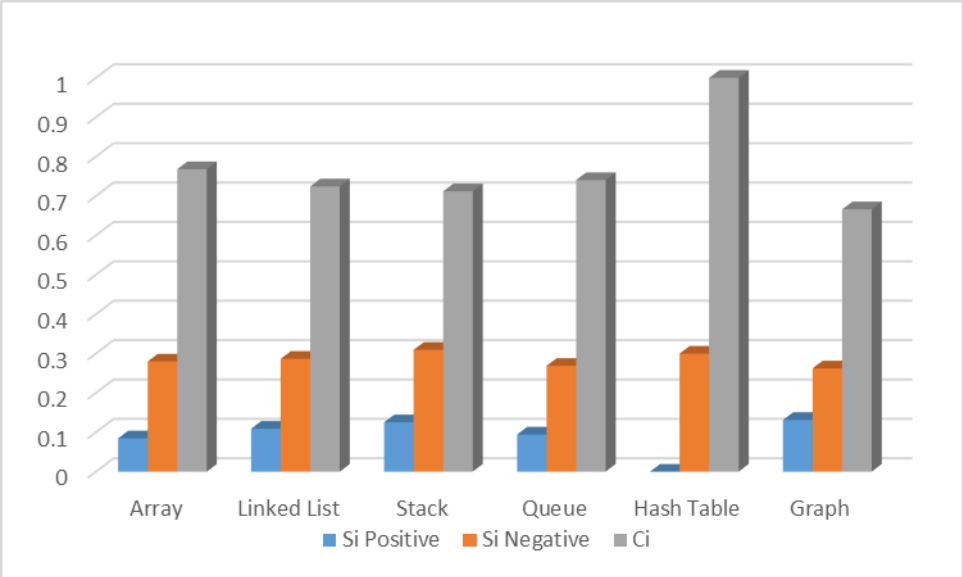


FIGURE 2. Si Positive, Si Negative and Ci value

In figure 2, the values for Si Positive, Si Negative, and Ci are provided for various data structures. The Hash Table stands out with a Si Positive value of 0 and the highest Ci value of 1, indicating optimal performance in certain contexts. The Array has a relatively low Si Positive (0.084532) and a moderate Ci value of 0.768013. The Graph has a higher Si Positive (0.131455) but a lower Ci of 0.666122, showing a trade-off between its complexity and efficiency. The Linked List, Stack, and Queue have values suggesting balanced performance with specific strengths depending on the use case

TABLE 9. Rank

	Rank
Array	2
Linked List	4
Stack	5
Queue	3
Hash Table	1
Graph	6

The following list ranks various data structures based on their common usage and importance in computer science. Hash tables rank highest due to their efficiency in data storage and retrieval, providing an average time complexity of O (1) for search, insertion, and deletion. They are widely used in applications thatrequirefast lookups, such as databases, caching, and subqueries. Arrays are basic data structures that provide constant-time element access via indexes. They are easy to implement and are commonly used in situations where fast random access is required, such as sorting algorithms and static data collections. Arrays follow the first-in-first-out (FIFO) principle, which makes them useful for scheduling, resource management, and buffering tasks such as print spooling and request handling in operating systems. Linked lists are dynamic structures that allow efficient insertion and deletion as memory requirements change. However, element access requires traversal, making them ideal for applications where memory allocation is unpredictable. Layers operate on the last-in-first-out (LIFO) principle and are essential for handling recursive function calls, expression evaluation, and undo instructions in software. Graphs, which represent relationships between objects, play a key role in areas such as network design, social networks, and pathfinding algorithms. Although highly complex, they provide significant flexibility in modeling interconnected data.

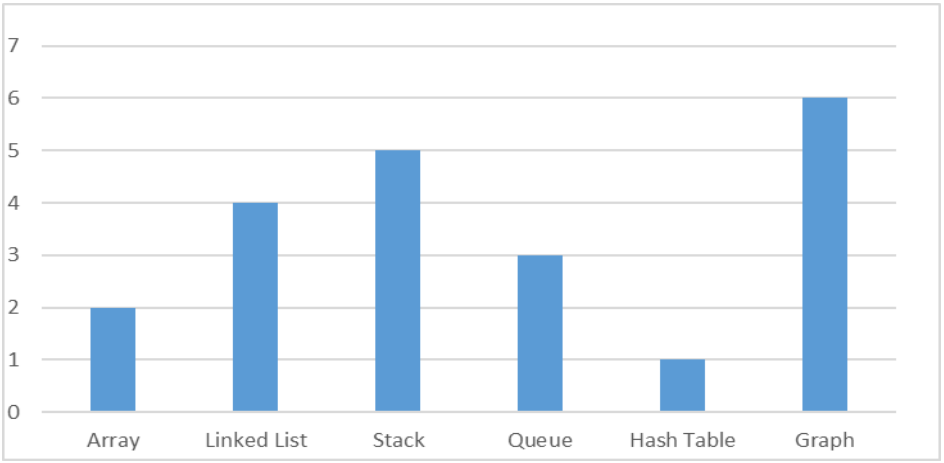


FIGURE 3. Rank

In figure 3, various data structures are ranked based on their performance or utility in different contexts. The Hash Table is ranked the highest at 1, likely due to its efficiency in providing fast lookups, insertions, and deletions with an average time complexity of O(1). The Array is ranked second at 2, known for its simplicity and direct access, though it can be less efficient for dynamic data manipulation. The Queue, Stack, and Linked List are ranked 3, 4, and 5, respectively, each excelling in specific use cases. Lastly, the Graph is ranked 6, highlighting its complexity and versatility in representing relationships between elements.

4. CONCLUSION

This survey examines key paradigms for designing and implementing efficient out-of-memory algorithms and data structures in a variety of problem domains, including sorting, FFT, scientific computing, computational geometry, mapping, databases, geographic information systems, and text and string processing. Despite significant progress, many challenges remain in these areas. A particularly difficult problem is determining lower bounds for sorting and ordering without relying on the assumption of separability.Quantitative civil war research increasingly acknowledges the wide gap between individual and group-level theories of mobilization and political violence and country-level empirical studies of armed conflict.This paper presents a novel interior point method and a unique data structure derived from the chaotic KKT conditions of the master problem. Extensive simulations of test systems ranging from 14 to 1047 buses demonstrate the effectiveness of the method for large-scale practical applications.When querying unstructured data, leveraging any available structural information can greatly improve performance. Several optimizations, especially for common path expressions, illustrate this impact (see [CACS94, CCM96], among others)We have developed a modified KD-tree traversal algorithm that eliminates the need for a par-ray stack. However, this change is driven Modern GPUs are constrained by limitations, which also reduce the working set size. required for ray tracing in kd-tree. This improvement enables greater parallelism in ray tracing, which we believe will be beneficial for a variety of architectures. Both model calculations and experimental data for a Zn-Al LDH confirm that the WT approach effectively distinguishes between two different atoms at the same distance, even though destructive interference causes partial data loss. This method is particularly valuable when the chemical composition of the sample is unknown, a common scenario in analyzing substates in environmental samples. While automated compiler-integrated prefetching has successfully reduced Memory latency is a challenge for array-based codes, and existing compiler technology has difficulty in prefetching pointer-based data structures. This paper proposes three prefetching schemes to solve the pointer-chasing problem, automates the most applicable approach (greedy prefetching) within the compiler, and evaluates their performance on a modern superscalar processor comparable to the MIPS R10000.The Technique for Ordered Priority the Technique for Order Preference Selection by Similarity to Ideal Solution (TOPSIS) is a widely used approach to decision-making problems. This paper provides a comprehensive overview of the development of fuzzy TOPSIS methods and emphasizes key applications such as location analysis, supplier selection, and sustainable and renewable energy. We expect this study to be a valuable resource for future research in this field. Traditional Fuzzy TOPSIS methods provide a robust framework for ranking competing alternatives by evaluating their overall performance across multiple criteria. Although these methods efficiently handle imprecise data through fuzzy set theory, they face challenges in handling other uncertainties, such as incompleteness. Addressing these uncertainties can lead to irrational and costly decision-making.

REFERENCES

[1]. Vitter, Jeffrey Scott. "External memory algorithms and data structures: Dealing with massive data." *ACM Computing surveys (CsUR)* 33, no. 2 (2001): 209-271.

[2]. Sleator, Daniel D., and Robert Endre Tarjan. "A data structure for dynamic trees." In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pp. 114-122. 1981.

[3]. Lee, Kunwoo, and David C. Gossard. "A hierarchical data structure for representing assemblies: part 1." *Computer-aided design* 17, no. 1 (1985): 15-19.

[4]. Driscoll, James R., Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. "Making data structures persistent." In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pp. 109-121. 1986.

[5]. Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. "I-structures: Data structures for parallel computing." In *Workshop on Graph Reduction*, pp. 336-369. Berlin, Heidelberg: Springer Berlin Heidelberg, 1986.

[6]. Jensen, Christian S., Torben Bach Pedersen, Laurynas Speičys, and Igor Timko. "Data modeling for mobile services in the real world." In *Advances in Spatial and Temporal Databases: 8th International Symposium, SSTD 2003, Santorini Island, Greece, July 2003. Proceedings 8*, pp. 1-9. Springer Berlin Heidelberg, 2003.

[7]. O'Hearn, Peter, John Reynolds, and Hongseok Yang. "Local reasoning about programs that alter data structures." In *Computer Science Logic: 15th International Workshop, CSL 2001 10th Annual Conference of the EACSL Paris, France, September 10–13, 2001, Proceedings 15*, pp. 1-19. Springer Berlin Heidelberg, 2001.

[8]. Wei, Hua, Hiroshi Sasaki, Junji Kubokawa, and R. Yokoyama. "An interior point nonlinear programming for optimal power flow problems with a novel data structure." *IEEE Transactions on Power Systems* 13, no. 3 (1998): 870-877.

[9]. Healey, Sean P., Warren B. Cohen, Yang Zhiqiang, and Olga N. Krankina. "Comparison of Tasseled Cap-based Landsat data structures for use in forest disturbance detection." *Remote sensing of environment* 97, no. 3 (2005): 301-310.

[10]. Weiler, Kevin. "Edge-based data structures for solid modeling in curved-surface environments." *IEEE Computer graphics and applications* 5, no. 1 (1985): 21-40.

[11]. Mishra, Suyash, and Anuranjan Misra. "Structured and unstructured big data analytics." In *2017 International Conference on Current Trends in Computer, Electrical, Electronics and Communication (CTCEEC)*, pp. 740-746. IEEE, 2017.

[12]. Chakrabarti, Kaushik, and Sharad Mehrotra. "The hybrid tree: An index structure for high dimensional feature spaces." In *Proceedings 15th International Conference on Data Engineering (Cat. No. 99CB36337)*, pp. 440-447. IEEE, 1999.

[13]. Chen, Peter Pin-Shan. "The entity-relationship model—toward a unified view of data." *ACM transactions on database systems (TODS)* 1, no. 1 (1976): 9-36.

[14]. Nievergelt, Jürg, Hans Hinterberger, and Kenneth C. Sevcik. "The grid file: An adaptable, symmetric multikey file structure." *ACM Transactions on Database Systems (TODS)* 9, no. 1 (1984): 38-71.

[15]. Sullivan, Kevin J., William G. Griswold, Yuanfang Cai, and Ben Hallen. "The structure and value of modularity in software design." *ACM SIGSOFT Software Engineering Notes* 26, no. 5 (2001): 99-108.

[16]. Cai, Yuanfang, Ben Hallen, Kevin Sullivan, and William Griswold. "The Structure and Value of Modularity in Software Design." (2001).

[17]. Becht, Etienne, Leland McInnes, John Healy, Charles-Antoine Dutertre, Immanuel WH Kwok, Lai Guan Ng, Florent Ginhoux, and Evan W. Newell. "Dimensionality reduction for visualizing single-cell data using UMAP." *Nature biotechnology* 37, no. 1 (2019): 38-44.

[18]. Liu, Jiesong, Feng Zhang, Lv Lu, Chang Qi, Xiaoguang Guo, Dong Deng, Guoliang Li et al. "G-learned index: Enabling efficient learned index on GPU." *IEEE Transactions on Parallel and Distributed Systems* (2024).

[19]. Pao, Derek, Cutson Liu, Angus Wu, Lawrence Yeung, and King Sun Chan. "Efficient hardware architecture for fast IP address lookup." In *Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 2, pp. 555-561. IEEE, 2002.

[20]. Frederickson, Greg N. "Data structures for on-line updating of minimum spanning trees, with applications." *SIAM Journal on Computing* 14, no. 4 (1985): 781-798.

[21]. Driscoll, James R., Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. "Making data structures persistent." In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pp. 109-121. 1986.

[22]. Wei, Hua, H. Sasaki, J. Kubokawa, and R. Yokoyama. "An interior point nonlinear programming for optimal power flow problems with a novel data structure." In *Proceedings of the 20th International Conference on Power Industry Computer Applications*, pp. 134-141. IEEE, 1997.

[23]. Behzadian, Majid, S. Khanmohammadi Otaghsara, Morteza Yazdani, and Joshua Ignatius. "A state-of the-art survey of TOPSIS applications." *Expert Systems with applications* 39, no. 17 (2012): 13051-13069.

[24]. Krohling, Renato A., and André GC Pacheco. "A-TOPSIS–an approach based on TOPSIS for ranking evolutionary algorithms." *Procedia Computer Science* 55 (2015): 308-317

[25]. Madi, Elissa Nadia, Jonathan M. Garibaldi, and Christian Wagner. "An exploration of issues and limitations in current methods of TOPSIS and fuzzy TOPSIS." In *2016 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pp. 2098-2105. IEEE, 2016.

[26]. Dymova, Ludmila, Pavel Sevastjanov, and Anna Tikhonenko. "An approach to generalization of fuzzy TOPSIS method." *Information Sciences* 238 (2013): 149-162.

[27]. Singh, Ritesh Kumar, and Lyes Benyoucef. "A fuzzy TOPSIS based approach for e-sourcing." *Engineering Applications of Artificial Intelligence* 24, no. 3 (2011): 437-448.

[28]. Singh, Ritesh Kumar, and Lyes Benyoucef. "A fuzzy TOPSIS based approach for e-sourcing." *Engineering Applications of Artificial Intelligence* 24, no. 3 (2011): 437-448.

[29]. Rahim, Robbi, S. Supiyandi, A. P. U. Siahaan, Tri Listyorini, Andy Prasetyo Utomo, Wiwit Agus Triyanto, Yudie Irawan et al. "TOPSIS method application for decision support system in internal control for selecting best employees." In *Journal of Physics: Conference Series*, vol. 1028, p. 012052. IOP Publishing, 2018.

[30]. Dos Santos, Bruno Miranda, Leoni Pentiado Godoy, and Lucila MS Campos. "Performance evaluation of green suppliers using entropy-TOPSIS-F." *Journal of cleaner production* 207 (2019): 498-509.

[31]. Yang, Z. L., Stephen Bonsall, and Jin Wang. "Approximate TOPSIS for vessel selection under uncertain environment." *Expert Systems with Applications* 38, no. 12 (2011): 14523-14534.

[32]. Tsaur, Ruey-Chyn. "Decision risk analysis for an interval TOPSIS method." *Applied Mathematics and Computation* 218, no. 8 (2011): 4295-4304.

[33]. de Farias Aires, Renan Felinto, and Luciano Ferreira. "A new approach to avoid rank reversal cases in the TOPSIS method." *Computers & Industrial Engineering* 132 (2019): 84-97.

[34]. Afful-Dadzie, Anthony, Eric Afful-Dadzie, and Charles Turkson. "A TOPSIS extension framework for re-conceptualizing sustainability measurement." *Kybernetes* 45, no. 1 (2016): 70-86.

[35]. Abdel-Basset, Mohamed, and Rehab Mohamed. "A novel plithogenic TOPSIS-CRITIC model for sustainable supply chain risk management." *Journal of Cleaner production* 247 (2020): 119586.

