STM JOURNALS

International Journal of
Data Structure Studies

Volume 2, Issue 2, 2024
July–December
DOI (Journal): 10.37591/IJDSS

https://journals.stmjournals.com/ijdss

Review IJDSS

Impact of Time Complexity Using Array and Linked List in Data Structure

Jigar Pandya[1], Poonam Chakravarty[2,*]

Abstract

Data structures are techniques for maintaining, manipulating, and storing data on a computer, enabling efficient access and modification. They support various operations, such as insertion, deletion, updating, and sorting. Examples of data structure include arrays, linked lists, graphs, heaps, stacks, and queues. Each data structure is designed to meet specific needs and solve particular problems. Typically, we identify the problem, devise a solution as an algorithm, and then write an efficient program. The program should be optimized for both time and space efficiency, which is why understanding 'time complexity' is essential. Time complexity measures the total execution time a program takes to complete and is expressed using Big O notation. Efficient data handling is crucial in modern computing applications, making it essential to understand the concept of time complexity in data structures. Time complexity measures the computational resources required by algorithms in relation to the size of their input. This abstract explores the fundamental time complexities of common data structures, offering insights into their operational efficiencies.

Keywords: Data structure, array and linked list, algorithm, time complexity

INTRODUCTION

Array

An array is a linear data structure that stores homogeneous data elements in contiguous memory locations. It is a static data structure, meaning that memory allocation is performed at the compile time. Although the array has a fixed size, it provides constant time access to elements [1].

Arrays are classified into three types based on their dimensionality, which refers to how the elements of the array are organized: One-dimensional (1-D), two-dimensional (2-D), and multidimensional (3-D) arrays. In this section, we discuss the time complexity of the common operations for 1-D and 2-D arrays.

Linked List

A linked list is a linear data structure that stores data in a linear sequence, similar to an array; however, each element is not stored in contiguous memory locations. Each data element, known as a node, consists of two parts: the first part, called 'data,' holds the actual value, and the second part, known as 'address,' stores the memory address of the next node [2]. Unlike arrays, linked lists allocate memory dynamically at runtime, thus making them more flexible. Linked lists do not have a fixed size, and their elements are stored in non-contiguous memory locations, allowing for efficient insertion and deletion operations. Here, we discuss the time complexity of the common operations for both singly and doubly linked lists.

***Author for Correspondence**
Poonam Chakravarty
E-mail: poonam.chakravarty@raiuniversity.edu

[1]Assistant Professor, Department of Computer Science, Rai University, Ahmedabad, Gujarat, India
[2]Assistant Professor, Department of CSE/IT, Rai University, Ahmedabad, Gujarat, India

Received Date: March 05, 2024
Accepted Date: October 16, 2024
Published Date: November 07, 2024

Citation: Jigar Pandya, Poonam Chakravarty. Impact of Time Complexity Using Array and Linked List in Data Structure. International Journal of Data Structure Studies. 2024; 2(2): 41–48p.

TIME COMPLEXITY

The time complexity for a specific program of a specific data structure varies. It is a standardized technique to describe the maximum running time increase rate, as well as the size of the input in the algorithm [3].

Time complexity in data structures is typically expressed using Big O notation, which specifies the highest running time of an algorithm that it takes to complete. Understanding the time complexity is crucial for an algorithm (program) that requires less memory and time.

- *O(1):* This denotes a constant time complexity, meaning that the algorithm's execution time remains the same regardless of the input size.
- *O(n):* This denotes the linear time complexity, meaning that the algorithm's execution time increases proportionally with the size of the input. Here, n represents the size of the input data.

TIME COMPLEXITY: ALGORITHM IN ARRAY OPERATIONS

During a linear search, each element of the array is iterated individually until the target element is located or the end of the array is reached [4]. In the worst-case scenario, the target element may be the last element in the array, or it may not be present at all, requiring the examination of every element. Consequently, the time complexity for searching an array using a linear search is linear, meaning that it is proportional to the size of the array [5].

Insertion at the End

- *Time Complexity:* O(1)
- *Explanation:* Any data element can be inserted into the last index of the array. As arrays allow direct access to their last element, adding a new element at the end can be performed in a constant time. Therefore, we do not need to shift any existing elements, resulting in constant time complexity for this operation.

Insertion at the Beginning

- *Time Complexity:* O(n)
- *Explanation:* We can insert any data element at the first index, but to do this, we need to shift all existing data elements by one position to the right to create space for the new element. This operation becomes costlier as the size of the array increases, as the shifting elements scale linearly with the array size. This is why inserting a data element at the beginning results in linear time complexity.

Insertion at a Specific Index

- *Time Complexity:* O(n)
- *Explanation:* Any data element can be inserted at a specific index in the array; however, all the existing elements must be shifted to create space for the new data element. If the desired index is closer to the beginning of the array, more elements will have to be shifted. Thus, the time complexity increased linearly with the distance of the insertion point from the start of the array.

Deletion at the End

- *Time Complexity:* O(1)
- *Explanation:* Removing an element from the end of an array is an operation that takes constant time. The last element is simply removed from the array. Because arrays allow direct access to their last element, there is no need to shift any other element, making the time complexity of this operation constant.

Deletion at the Beginning

- *Time Complexity:* O(n)

- *Explanation:* We can delete any data element from the beginning of the array; however, this requires shifting the remaining elements. When a data element is deleted, it is necessary to fill in the gap, which means that all subsequent elements are shifted to the left. This operation becomes more expensive as the size of the array increases because of the number of elements that need to be shifted scales linearly with the array size. Consequently, deleting an element from the beginning of the array results in linear time complexity.

Deletion at a Specific Index

- *Time Complexity:* O(n)
- *Explanation:* Deleting an element from a specific index in an array requires shifting the elements that follow the deleted element from one position to the left. The number of elements that must be shifted depends on the position of the deleted element within the array. If the index of the deleted element is closer to the beginning, more elements will need to be shifted, resulting in linear time complexity.

These detailed explanations illustrate how the time complexity of array operations varies based on a specific operation and its characteristics [6].

One-Dimensional Array Operations

1. Accessing an element
   - *Time Complexity:* O(1)
2. Traversing an element
   - *Time Complexity:* O(n)
3. Inserting an element at the end
   - *Time Complexity:* O(1)
4. Inserting an element at the beginning:
   - *Time Complexity:* O(n)
5. Deleting an element
   - *Time Complexity:* O(n)
6. Searching an element
   - *Time Complexity:* O(n)

Two-Dimensional Array Operations

1. Accessing an element
   - *Time Complexity:* O(1)
2. Traversing an element:
   - *Time Complexity:* O(m*n)
3. Inserting an element at a specific position
   - *Time Complexity:* O(1)
4. Deleting an element
   - *Time Complexity:* O(m*n)
5. Searching an element
   - *Time Complexity:* O(m*n)

TIME COMPLEXITY: LINKED LIST OPERATIONS

The complexity of the operations in linked lists directly affects their efficiency and performance in various scenarios. Here, the complexity impact linked lists [7].

Access Time

Linked lists have a linear access time complexity of O(n) for accessing elements by index. This means that the time required to access an element increases linearly with the input size of the list. Arrays provide constant time access O(1) because elements are stored contiguously in memory.

Search Time

Similar to the access time, searching for a data element in a linked list also has a linear time complexity of O(n). This is because one typically needs to traverse the list sequentially from the head (or tail) until the desired element is found [8]. In contrast, some search operations in arrays can be optimized to achieve better than linear time complexity by using techniques such as binary search.

Insertion and Deletion Time

Compared with arrays, the insertion and deletion of data elements in linked lists have better time complexity at the beginning of the list. The time complexity for insertion and deletion insertion at the beginning of a linked list is O(1) because it involves updating pointers without shifting elements. However, these operations become less efficient towards the end of the list, with both insertion and deletion taking O(n) time because of the need to traverse the list to find the insertion or deletion point. In arrays, inserting or deleting elements at the end is an operation that occurs in constant time (O(1)), whereas insertion or deletion at the beginning takes linear time (O(n)) because it involves shifting elements [9].

Memory Overhead

Linked lists typically have more memory overhead than arrays, because each node in the list requires additional memory to store pointers/references to the next (and possibly previous) nodes. This additional memory overhead can affect cache performance and memory usage, especially for large lists.

Dynamic Memory Allocation

Linked lists may incur overhead from dynamic memory allocation/deallocation when nodes are dynamically created and destroyed. This overhead can impact the performance and memory fragmentation in certain scenarios, especially for frequently changing lists [10].

In summary, although linked lists offer efficient insertion and deletion at the beginning and do not require contiguous memory allocation, their linear access and search time complexities can make them less efficient than arrays for certain operations, especially when random access or search is frequently performed. When deciding between the linked lists and other data structures, it is crucial to consider the particular needs of the application.

Accessing an Element (by Index)

- *Time Complexity:* O(n)
- *Explanation:* In contrast to arrays, linked lists do not provide immediate access to elements by using indices. To access an element at a specific index, the list must be traversed from the head (or tail, if doubly linked) until the desired index is reached. This traversal takes linear time, as each node may need to be visited in the worst-case scenario.

Searching for an Element

- *Time Complexity:* O(n)
- *Explanation:* Similar to accessing an element by index, searching for a specific value in a linked list requires traversing the list from the head (or tail) until the target value is obtained. In the worst-case scenario, each node in the list may need to be visited, resulting in a linear time complexity.

Insertion at the Beginning

- *Time Complexity:* O(1)
- *Explanation:* Adding a new node at the start of a linked list requires creating a new node and adjusting its next pointer of the new node to reference the existing head of the list. This operation takes a constant amount of time because it does not depend on the size of the list; only a few pointers need to be updated.

Insertion at the End

- *Time Complexity:* O(n)
- *Explanation:* Unlike insertion at the beginning, insertion of a new node at the end of a singly linked list requires traversing the entire list to reach the last node. Once the last node is reached, the new node can be appended, resulting in a linear time complexity owing to traversal.

Insertion at a Specific Index

- *Time Complexity:* O(n)
- *Explanation:* Inserting a new node at a specific index involves traversing the list until the desired index is achieved. Once the index is reached, the new node can be inserted, resulting in a linear time complexity owing to traversal.

Deletion at the Beginning

- *Time Complexity:* O(1)
- *Explanation:* Deleting the first node of a linked list involves updating the head pointer to point at the next node and freeing the memory allocated to the old head node. This operation takes a constant amount of time because it does not depend on the size of the list.

Deletion at the End

- *Time Complexity:* O(n)
- *Explanation:* Removing the last node from a singly linked list involves navigating the entire list until the second-to-last node is reached. Upon reaching the second-to-last node, the next pointer can be modified to null, thereby effectively eliminating the last node. This operation takes linear time owing to the traversal.

Deletion at a Specific Index

- *Time Complexity:* O(n)
- *Explanation:* Deleting a node in a specific index involves traversing the list until the desired index is reached. Once the index is reached, the node can be removed by updating the pointers of its neighboring nodes. This operation takes linear time owing to the traversal.

Singly Linked List Operations

1. Accessing an element
   - *Time Complexity:* O(n)
2. Traversing an element
   - *Time Complexity:* O(n)
3. Inserting a node at the end
   - *Time Complexity:* O(n)
4. Inserting a node at the beginning
   - *Time Complexity:* O(1)
5. Deleting a node at the end
   - *Time Complexity:* O(n)
6. Deleting a node at the beginning
   - *Time Complexity:* O(1)
7. Searching an element
   - *Time Complexity:* O(n)

Doubly Linked List Operations

1. Accessing a node
   - *Time Complexity:* O(n)

2. Traversing an element
   - *Time Complexity:* O(n)
3. Inserting a node at the end
   - *Time Complexity:* O(1)
4. Inserting a node at the beginning
   - *Time Complexity:* O(1)
5. Deleting a node at the end
   - *Time Complexity:* O(1)
6. Deleting a node at the beginning
   - *Time Complexity:* O(1)
7. Searching an element
   - *Time Complexity:* O(n)

TIME COMPLEXITY COMPARISON: ARRAY VERSUS LINKED LIST

Time Complexity

- Accessing an Element (by index):
  - *Array:* O(1)
  - *Linked List:* O(n)
- Insertion/Deletion at the Beginning:
  - *Array:* O(n)
  - *Linked List:* O(1)
- Insertion/Deletion at the End:
  - *Array:* O(1)
  - *Linked List:* O(1)
- Insertion/Deletion at a Specific Index:
  - *Array:* O(n)
  - *Linked List:* O(n)
- Searching for an Element:
  - *Both:* O(n) (for linear search)

Memory Usage

- Array:
  - Fixed size determined at initialization.
  - Contiguous memory allocation.
  - This may result in memory wastage if the array size is larger than required.
- Linked List:
  - Dynamic size; memory is allocated as nodes are created.
  - Non-contiguous memory allocation.
  - Can be more memory-efficient if the size of the list varies significantly.

Flexibility

- Array:
  - Fixed size, not easily resizable without creating a new array and copying elements.
  - Direct access to elements by index.
  - Better suited for scenarios where random access and fixed size collections are required.
- Linked List:
  - Dynamic size; can easily grow or shrink by adding or removing nodes.
  - No direct access to elements by index; traversal is required to access elements.
  - It is better suited for scenarios where frequent insertions and deletions are required, especially at the beginning of the collection.

Insertion/Deletion Efficiency

- Beginning of the Collection:
  - *Array:* O(n)
  - *Linked List:* O(1)
- End of the Collection:
  - *Both:* O(1)
- Specific Index:
  - *Both:* O(n)

Spatial Locality

- *Array*
  - Better spatial locality due to contiguous memory allocation, leading to improved cache performance.
- *Linked List*
  - Poor spatial locality owing to non-contiguous memory allocation potentially leads to cache misses.

Traversal Efficiency

- *Array*
  - More efficient traversal due to direct access to elements by index.
- *Linked List*
  - Less efficient traversal due to the need to follow pointers from node to node.

In summary, arrays offer better time complexity for accessing elements by index and insertion/deletion at the end, whereas linked lists excel in insertion/deletion at the beginning and provide flexibility in dynamic memory allocation. The priority of choosing arrays or linked lists depends on the specific requirements of the application, which may include various types of operations that are performed on it [11].

Table 1 represents, highlighting that it compares the performance of different data structures (1-D array, 2-D array, singly linked list, and doubly linked list) across various operations. Let me know if you need any changes.

Table 1. Performance comparison of data structures for various operations.

Operations	1-D Array	2-D Array	Singly Linked List	Doubly Linked List
Accessing	O(1)	O(1)	O(n)	O(n)
Traversing	O(n)	O(m*n)	O(n)	O(n)
Inserting	O(1)	O(1)	O(1)	O(1)
Deleting	O(n)	O(m*n)	O(n)	O(1)
Searching	O(n)	O(m*n)	O(n)	O(n)

CONCLUSION

This study primarily focused on determining which method is more efficient for data allocation by comparing arrays and linked lists. Accessing an element in an array takes O(1) time, whereas accessing an element in a linked list takes O(n). Consequently, arrays are generally preferred for efficient data allocation. However, if quick and frequent deletions are required, especially with large data volumes, a doubly linked list is more advantageous because it allows for O(1) deletion. In contrast, 1-D array and 2-D array require O(n) and O(m*n) times for deletions, respectively.

REFERENCES

1. Akinde Aderonke O, Okolie Samuel O, Kuyoro'Shade O. The S-linked list–A variant of the linked list data structure. J Emerg Trends Comput Inf Sci. 2013;4:571–6.

2. Agrawal SC, Singh S, Gautam AK, Singh MK. Basic concept of embedded ‘C’. Int J Comput Sci Inform. 2012;1:290–4.
3. Azar E, Alebicto ME. Swift Data Structure and Algorithms. Birmingham, United Kingdom: Packt Publishing Ltd.; 2016.
4. Mühlberg JT, White DH, Dodds M, Lüttgen G, Piessens F. Learning assertions to verify linked-list programs. In: Calinescu R, Rumpe B, editors. Software Engineering and Formal Methods. SEFM 2015. Lecture Notes in Computer Science. Vol. 9276. Cham: Springer; 2015. p. 37–52. DOI: 10.1007/978-3-319-22969-0_3.
5. Mridha P, Datta BK. An algorithm for analysis the time complexity for iterated local search (ILS). Res Appl Math. 2021;7:52–4.
6. Abhar MO, Gatuam N. A review data structure, algorithms & analysis. J Emerg Technol Innov Res. 2019;6:59–64.
7. GeeksforGeeks. (2023). Complete guide on complexity analysis data structure and algorithms tutorial [Online]. GeeksforGeeks. Available from: https://www.geeksforgeeks.org/complete-guide-on-complexity-analysis/.
8. Lokeshwar B, Zaid MM, Naveen S, Venkatesh J, Sravya L. Analysis of time and space complexity of array, linked list and linked array (hybrid) in linear search operation. 2022 International Conference on Data Science, Agents & Artificial Intelligence (ICDSAAI), Chennai, India. 2022. p. 1–6. DOI: 10.1109/ICDSAAI55433.2022.10028872.
9. Singh Chauhan A. A comparative study of various sorts of data structures. Int Res J Mod Eng Technol Sci. 2021;3:3011–6.
10. Devi KR. Analysis of arraylist and linked list. Int J Comput Sci Eng. 2019;7:1566–70. DOI: 10.26438/ijcse/v7i5.15661570.
11. Shastri S, Mansotra V, Bhadwal AS, Kumari M, Khajuria A, Singh DS. A GUI based run-time analysis of sorting algorithms and their comparative study. Int J Comput Sci Eng. 2017;5:217–21. DOI: 10.26438/ijcse/v5i11.217221.

