Doubly Linked List

* **What is** Doubly Linked List **Data Structure?**

A doubly linked list is a linear data structure that consists of a sequence of nodes, where each node contains a value and two pointers. These pointers, commonly known as "next" and "previous" pointers, allow for traversal in both forward and backward directions.

In a doubly linked list, each node contains three components:

- Value: The actual data or value that the node holds.

- Next Pointer: A pointer that points to the next node in the list.

- Previous Pointer: A pointer that points to the previous node in the list.

The doubly linked list maintains a logical order among its nodes, with the first node being called the head and the last node being called the tail. The previous pointer of the head node and the next pointer of the tail node are typically set to null or some sentinel value to indicate the end of the list in their respective directions.

The main advantage of a doubly linked list over a singly linked list is that it provides bidirectional traversal. This allows for efficient operations such as insertion and deletion at both the beginning and end of the list. In a singly linked list, such operations require traversing the list from the head to the desired position, which can be time-consuming for large lists.

The doubly linked list also enables backward traversal, which can be useful in certain scenarios. However, this flexibility comes at the cost of increased memory usage compared to singly linked lists, as each node requires an additional pointer to store the previous node's address.

Here's a visual representation of a doubly linked list:

```

null ⇐ [prev] Node 1 [next] ⇐ [prev] Node 2 [next] ⇐ [prev] Node 3 [next] ⇒ null

```

In the above example, Node 1 is the head, Node 3 is the tail, and each node contains a value along with the previous and next pointers.

Doubly linked lists are commonly used in scenarios where efficient insertion and deletion at both ends of the list are required, such as implementing a deque (double-ended queue), LRU (Least Recently Used) cache, or certain algorithms and data structures.

* **Why there is a need for** Doubly Linked List **Data Structure?**

**The doubly linked list data structure is used in scenarios where bidirectional traversal and efficient insertion and deletion at both ends of the list are required. Here are some reasons why a doubly linked list may be needed:**

**1. Bidirectional Traversal: Unlike singly linked lists, which allow traversal in only one direction (forward), doubly linked lists provide the ability to traverse in both forward and backward directions. This bidirectional traversal can be useful in various scenarios, such as implementing iterators, searching for elements from both ends, or implementing algorithms that require backward navigation.**

**2. Efficient Insertion and Deletion at Both Ends: Doubly linked lists excel when it comes to insertion and deletion operations at both the beginning and end of the list. In a singly linked list, inserting or deleting a node at the end of the list requires traversing the entire list to reach the tail. However, in a doubly linked list, the tail node's reference is readily available, allowing for constant-time insertion and deletion at the tail. Similarly, inserting or deleting a node at the beginning of the list can be done in constant time by updating the head node's reference.**

**3. Implementing Deques (Double-Ended Queues): Deques are data structures that allow insertion and deletion of elements at both ends. Doubly linked lists are commonly used to implement deque data structures efficiently. The bidirectional traversal and constant-time insertion and deletion operations at both ends make doubly linked lists an ideal choice for implementing deques.**

**4. LRU (Least Recently Used) Cache: LRU cache is a common caching technique that discards the least recently used items when the cache reaches its capacity. Doubly linked lists can be used to implement the LRU cache efficiently. The cache can be represented as a doubly linked list, with the most recently used item at the head and the least recently used item at the tail. This allows constant-time access, insertion, and deletion of items in the cache.**

**5. Undo/Redo Functionality: Doubly linked lists can be useful in implementing undo/redo functionality in applications where users can perform actions and then undo or redo them. Each action can be represented as a node in the doubly linked list, with the previous and next pointers maintaining the sequence of actions. Undoing an action involves traversing backward in the list, while redoing an action involves traversing forward.**

**These are just a few examples of scenarios where the features provided by doubly linked lists are beneficial. The choice of data structure depends on the specific requirements of the problem at hand, and doubly linked lists offer advantages in situations that demand bidirectional traversal and efficient insertion and deletion at both ends.**

Operations on Doubly Linked ListData Structure with Asymptotic Complexity and Complexity Analysis:

Here are the common operations performed on a doubly linked list data structure along with their asymptotic complexities:

1. Insertion at the beginning (prepend):a

- Complexity: O(1)

- Description: Add a new node at the beginning of the list by updating the head pointer and adjusting the necessary pointers.

2. Insertion at the end (append):

- Complexity: O(1)

- Description: Add a new node at the end of the list by updating the tail pointer and adjusting the necessary pointers.

3. Insertion at a specific position:

- Complexity: O(n)

- Description: Inserting a node at a specific position requires traversing the list to find the desired position and adjusting the pointers accordingly. In the worst case, it may require visiting all the nodes, resulting in linear time complexity.

4. Deletion from the beginning:

- Complexity: O(1)

- Description: Remove the first node in the list by updating the head pointer and adjusting the necessary pointers.

5. Deletion from the end:

- Complexity: O(1)

- Description: Remove the last node in the list by updating the tail pointer and adjusting the necessary pointers.

6. Deletion of a specific node by value:

- Complexity: O(n)

- Description: To delete a specific node by its value, the list needs to be traversed to find the node, and then the pointers are adjusted to remove it. In the worst case, it may require visiting all the nodes, resulting in linear time complexity.

7. Searching for a specific value:

- Complexity: O(n)

- Description: Searching for a specific value in a doubly linked list requires traversing the list and comparing the values until the desired value is found or the end of the list is reached. In the worst case, it may require visiting all the nodes, resulting in linear time complexity.

8. Traversing the list:

- Complexity: O(n)

- Description: Visiting or iterating over all the nodes in the list requires traversing from the head to the tail or vice versa, resulting in linear time complexity.

9. Accessing a specific node by index:

- Complexity: O(n)

- Description: Accessing a specific node by its index requires traversing the list from the head or tail until the desired index is reached. In the worst case, it may require visiting all the nodes, resulting in linear time complexity.

10. Reversing the list:

- Complexity: O(n)

- Description: Reversing a doubly linked list involves swapping the prev and next pointers of each node. It requires traversing the entire list once, resulting in linear time complexity.

11. Get the size of the list:

- Complexity: O(1)

- Description: Maintaining a size variable or counter allows retrieving the number of nodes in constant time.

It's worth noting that the time complexities mentioned above assume a basic implementation of the doubly linked list without any additional data structures or optimizations. Depending on the specific requirements and additional features implemented, the complexities can vary.

* **Implementation of Operations on** Doubly Linked List **Data Structure:**

**Look at the code ….**

* **Properties of** Doubly Linked List **Data Structure:**

**The LinkedList data structure has several notable properties:**

**The doubly linked list data structure has several properties that distinguish it from other types of data structures. Here are the key properties of a doubly linked list:**

**1. Doubly Linked Structure:**

**- Each node in a doubly linked list contains two pointers: one pointing to the previous node (prev) and one pointing to the next node (next).**

**- This bidirectional linkage allows for efficient traversal in both directions, from the head to the tail and vice versa.**

**2. Dynamic Size:**

**- Doubly linked lists can dynamically grow or shrink in size as nodes are added or removed.**

**- Unlike arrays, which have a fixed size, doubly linked lists can adapt to accommodate changing data requirements.**

**3. Head and Tail Pointers:**

**- A doubly linked list typically maintains two pointers: the head pointer that points to the first node and the tail pointer that points to the last node.**

**- These pointers provide easy access to the beginning and end of the list, enabling efficient insertion and deletion at both ends.**

**4. Insertion and Deletion Flexibility:**

**- Doubly linked lists offer flexibility in inserting and deleting nodes at various positions within the list.**

**- Nodes can be inserted or deleted at the beginning, end, or at any specific position in between.**

**- This flexibility allows for efficient operations such as inserting or deleting nodes adjacent to a given node.**

**5. Efficient Insertion and Deletion at Ends:**

**- Insertion and deletion at the beginning or end of a doubly linked list can be performed in constant time (O(1)).**

**- This is because the head and tail pointers can be updated directly, and the necessary adjustments to the adjacent nodes' pointers can be made easily.**

**6. Memory Overhead:**

**- Doubly linked lists require additional memory to store the prev and next pointers for each node.**

**- Compared to singly linked lists, which only require a single pointer per node, doubly linked lists have a higher memory overhead.**

**7. Reverse Traversal:**

**- Doubly linked lists allow efficient traversal in reverse order.**

**- Starting from the tail pointer, nodes can be traversed backward, which is useful in certain scenarios where backward traversal is required.**

**8. Versatility:**

**- Doubly linked lists are versatile and can be used in various applications and algorithms.**

**- They provide flexibility in managing and manipulating data, making them a suitable choice in scenarios where frequent insertion and deletion operations are required.**

**These properties make doubly linked lists a valuable data structure in situations where efficient insertion and deletion at both ends, as well as bidirectional traversal, are necessary. However, it's important to consider the trade-offs, such as the increased memory overhead, when choosing the appropriate data structure for a specific use case.**

* **Applications of** Doubly Linked List **Data Structure:**

**The doubly linked list data structure finds application in various scenarios where bidirectional traversal, efficient insertion and deletion at both ends, and flexibility in manipulating the data are required. Here are some common applications of doubly linked lists:**

**1. Implementation of Deque (Double Ended Queue):**

**- Doubly linked lists provide an efficient implementation of the deque data structure, which allows insertion and deletion at both ends.**

**- Deques are useful in scenarios where elements need to be added or removed from either end with constant time complexity.**

**2. Undo/Redo Functionality:**

**- Doubly linked lists are commonly used to implement undo/redo functionality in text editors or other applications.**

**- Each change is stored as a node in the list, and the prev and next pointers allow for efficient navigation through the history of changes.**

**3. Browser History:**

**- Doubly linked lists can be used to implement the back and forward navigation functionality in web browsers.**

**- Each visited page is stored as a node, and the prev and next pointers enable efficient traversal of the browsing history.**

**4. Cache Implementation:**

**- Doubly linked lists are utilized in cache implementations, such as LRU (Least Recently Used) or LFU (Least Frequently Used) caches.**

**- The list keeps track of the most recently accessed or frequently used elements, allowing for efficient eviction of least used elements.**

**5. Music or Video Playlist:**

**- Doubly linked lists are suitable for implementing playlists in music or video players.**

**- Each song or video is represented as a node, and the prev and next pointers enable easy navigation between the playlist items.**

**6. Memory Management:**

**- Doubly linked lists are used in memory management algorithms, such as the buddy system or free list management.**

**- The list maintains a collection of free memory blocks, and the pointers facilitate efficient allocation and deallocation of memory.**

**7. Hash Table Chaining:**

**- Doubly linked lists are employed in hash table chaining to handle collisions.**

**- When multiple elements hash to the same index, a doubly linked list is used to store them, allowing efficient search, insertion, and deletion within each chain.**

**8. Polynomial Representation:**

**- Doubly linked lists can be used to represent polynomials in algebraic computations.**

**- Each term of the polynomial can be stored as a node, and the list supports efficient addition, subtraction, and multiplication operations.**

**These are just a few examples of the applications of doubly linked lists. The flexibility and bidirectional traversal provided by this data structure make it suitable for various use cases where efficient manipulation and traversal of data in both directions are required.**

* **Advantages of** Doubly Linked List **Data Structure:**

**A Doubly Linked List is a type of linked list where each node contains pointers to both the next node and the previous node in the list. This additional reference to the previous node provides several advantages over a singly linked list. Here are some advantages of using a Doubly Linked List data structure:**

**1. Bi-directional Traversal: One of the primary advantages of a Doubly Linked List is the ability to traverse the list in both directions. Each node contains a reference to both the next and previous nodes, allowing efficient backward traversal. This feature enables operations like reverse traversal, searching from the end of the list, or iterating in both directions.**

**2. Efficient Insertion and Deletion: Doubly Linked List allows for efficient insertion and deletion operations at both ends of the list. Since each node contains references to both the next and previous nodes, inserting or deleting a node involves updating the pointers of only three nodes: the previous node, the node being inserted/deleted, and the next node. This makes these operations faster compared to singly linked lists, where inserting or deleting a node requires updating the pointers of only two nodes.**

**3. Flexibility in Operations: The bi-directional nature of a Doubly Linked List provides flexibility in performing various operations on the list. It allows for efficient insertion or deletion of nodes at any position within the list, including the beginning or end, without traversing the entire list. This flexibility is particularly useful in scenarios where frequent modifications or reordering of elements are required.**

**4. Tail Access: The reference to the last node (tail) in a Doubly Linked List allows convenient access to the end of the list. This makes operations like appending elements or accessing the last node efficient, as there is no need to traverse the entire list from the beginning.**

**5. Backtracking: With a Doubly Linked List, it is easier to backtrack or move backward in the list. The previous node reference simplifies operations that require moving back from the current node to the previous node, such as undoing or reversing operations.**

**6. Implementing Advanced Data Structures: Doubly Linked List serves as a fundamental building block for implementing more complex data structures like a deque (double-ended queue) or a circular doubly linked list. These data structures leverage the bi-directional traversal and efficient insertion/deletion characteristics of Doubly Linked List to provide additional functionalities.**

**It's important to consider that Doubly Linked List also has some disadvantages, such as increased memory usage due to the additional previous node reference and the complexity of managing the links between nodes. The choice of using a Doubly Linked List depends on the specific requirements and trade-offs of the problem at hand.**

* **Disadvantages of** Doubly Linked List **Data Structure**[:](https://docs.google.com/document/d/117SPrcFF80B2fxxYIisA5cTEdbVoxfgHV1gcGiOCn2w/edit" \l "heading=h.mtw228jwq389)

While Doubly Linked List offers several advantages, it also has some disadvantages compared to other data structures like singly linked lists or arrays. Here are some of the disadvantages of using a Doubly Linked List:

1. Increased Memory Usage: Doubly Linked List requires additional memory to store the reference to the previous node in each node. This additional reference increases the memory overhead compared to singly linked lists, where only the reference to the next node is needed. If memory usage is a concern, the extra memory requirement of a Doubly Linked List can be a disadvantage.

2. Complex Implementation and Maintenance: The presence of both next and previous node references in each node makes the implementation and maintenance of a Doubly Linked List more complex compared to singly linked lists. The additional pointers introduce additional complexity in handling operations like insertion, deletion, and updating links when nodes are added or removed.

3. Increased Time Complexity: The presence of the previous node reference in each node affects the time complexity of certain operations. While insertion and deletion at the beginning or end of the list are efficient, operations that involve traversing the list, such as searching for an element, can be slightly slower compared to singly linked lists. Traversing the list requires following both the next and previous pointers, which adds extra overhead.

4. Additional Overhead for Link Updates: In a Doubly Linked List, updating the links when adding or removing nodes requires updating both the next and previous references of multiple nodes. This additional overhead can impact the performance of operations that involve frequent modifications, especially in scenarios where elements are frequently added or removed from the list.

5. Fragile Links: The presence of two separate pointers in each node increases the potential for link-related issues. It introduces the possibility of inconsistent or broken links if the links are not managed correctly during insertion, deletion, or other operations. Ensuring the integrity of the links in a Doubly Linked List requires careful attention and proper handling.

6. Increased Code Complexity: The presence of the previous node reference in a Doubly Linked List can lead to increased code complexity. The extra reference complicates the code logic and makes it more error-prone, especially when implementing algorithms or performing complex operations on the list.

It's important to consider these disadvantages when choosing a data structure for a particular application. Depending on the specific requirements and characteristics of the problem, other data structures may be more suitable alternatives to Doubly Linked List.

Diff between linkedList and Doubly linkedList:

The main difference between a LinkedList and a DoublyLinkedList lies in the type of links between the nodes.

1. LinkedList:

In a LinkedList, each node contains a reference or pointer to the next node in the list. The nodes are connected in a linear fashion, forming a singly linked list. Each node holds a reference to the next node but does not have a reference to the previous node. Traversal in a LinkedList can only be done in one direction, typically from the head (first node) to the tail (last node).

2. DoublyLinkedList:

In a DoublyLinkedList, each node contains references or pointers to both the next node and the previous node in the list. The nodes are connected in a bi-directional manner, forming a doubly linked list. Each node holds a reference to the next node as well as a reference to the previous node. Traversal in a DoublyLinkedList can be done in both directions, allowing movement back and forth between nodes.

Key differences between LinkedList and DoublyLinkedList:

a. Traversal Direction:

- LinkedList: Traversal is unidirectional, typically from the head to the tail.

- DoublyLinkedList: Traversal is bidirectional, allowing movement in both forward and backward directions.

b. Link Structure:

- LinkedList: Each node has a reference to the next node only.

- DoublyLinkedList: Each node has references to both the next node and the previous node.

c. Flexibility in Operations:

- LinkedList: Insertion and deletion operations are efficient at the beginning or end of the list. However, operations in the middle of the list require traversal from the head.

- DoublyLinkedList: Insertion and deletion operations are efficient at both ends of the list. Additionally, operations in the middle of the list can be performed more efficiently due to the bidirectional links.

d. Memory Overhead:

- LinkedList: Requires less memory as it only needs a single reference per node (next node).

- DoublyLinkedList: Requires more memory as it needs two references per node (next node and previous node).

e. Implementation Complexity:

- LinkedList: Simpler to implement and manage due to the single reference per node.

- DoublyLinkedList: More complex to implement and manage due to the additional reference to the previous node.

The choice between LinkedList and DoublyLinkedList depends on the specific requirements of the application. If bidirectional traversal or efficient insertion/deletion at both ends is necessary, DoublyLinkedList is a better choice. For simpler scenarios where unidirectional traversal suffices, LinkedList may be sufficient and more memory-efficient.

Sure! Here's a table summarizing the differences between `LinkedList` and `DoublyLinkedList` in Java, along with their asymptotic complexities for common operations:

|  |  |  |
| --- | --- | --- |
| **Operation** | **LinkedList** | **DoublyLinkedList** |
| Access (by index) | O(n) | O(n/2) |
| Search (by value) | O(n) | O(n) |
| Insertion (at beginning) | O(1) | O(1) |
| Insertion (at end) | O(1) | O(1) |
| Insertion (at given index) | O(n) | O(n/2) (average case) |
| Deletion (at beginning) | O(1) | O(1) |
| Deletion (at end) | O(n) | O(1) |
| Deletion (at given index) | O(n) | O(n/2) (average case) |
| Traversal | O(n) | O(n) |

Note:

- "n" refers to the number of elements in the list.

- The complexities provided are average-case complexities, as the worst-case complexities can be slightly higher for certain scenarios.

In general, `LinkedList` and `DoublyLinkedList` have similar complexities for most operations. However, `DoublyLinkedList` provides the advantage of bidirectional traversal, which can be beneficial for certain operations such as accessing elements by index or performing insertions/deletions at specific positions.

It's worth noting that these complexities are theoretical and may vary depending on the specific implementation and the hardware environment in practice.