Linked lists

**Introduction to Linked Lists**

Linked lists are a fundamental data structure used in computer science for storing and organizing data. Unlike arrays, which store elements in contiguous memory locations, linked lists store elements as separate objects called nodes, which are linked together by pointers.

**Main Components:**

1. **Node:** The building block of a linked list. Each node contains two parts:
   * Data: The actual information being stored.
   * Pointer: A reference or link to the next node in the sequence.
2. **Pointer:** A reference that points to the next node in the sequence. It essentially establishes the logical connection between nodes.

**Types of Linked Lists:**

1. **Singly Linked List:** Each node has a single pointer that points to the next node in the sequence. The last node points to NULL, indicating the end of the list.
2. **Doubly Linked List:** Each node has two pointers - one that points to the next node and another that points to the previous node. This allows traversal in both forward and backward directions.
3. **Circular Linked List:** In this type of linked list, the last node points back to the first node, forming a circular structure.

**Advantages of Linked Lists:**

* **Dynamic Size:** Linked lists can dynamically grow and shrink in size during program execution.
* **Insertion and Deletion:** Insertion and deletion of elements are efficient, especially compared to arrays, as they do not require shifting elements.
* **Flexibility:** Linked lists can be easily modified and rearranged without the need for contiguous memory.

**Disadvantages of Linked Lists:**

* **Access Time:** Random access to elements (like in arrays) is not efficient. Traversal is required to access individual elements.
* **Memory Overhead:** Extra memory is required for storing pointers, increasing memory overhead compared to arrays.

**Applications:**

* **Dynamic memory allocation:** Linked lists are commonly used in implementing dynamic data structures such as stacks, queues, and hash tables.
* **File Systems:** Linked lists are used to represent directory structures and manage file allocation.

**Key Concepts**

**Head:** The first node of the linked list. It serves as the starting point for traversing the list.

**Tail:** The last node of the linked list. In some implementations, it is useful to keep a reference to the tail for efficient insertion at the end.

**Traversal:** Moving through the linked list from one node to another. This is typically done sequentially, starting from the head or any other node, following the pointers until the end of the list.

**Null Pointer**: The reference in the last node of the list, which points to nothing (or NULL). It indicates the end of the list.

**Introduction to Singly Linked Lists**

A singly linked list is a linear data structure consisting of a sequence of elements called nodes. Each node contains two parts: data and a reference to the next node in the sequence. Unlike arrays, which have fixed sizes, singly linked lists allow for dynamic memory allocation, making them suitable for situations where the size of the data structure may change frequently.

**Basic Structure:**

In a singly linked list:

* Each node contains two fields:
  1. **Data Field:** This holds the value or data associated with the node.
  2. **Next Pointer:** This is a reference (pointer) to the next node in the sequence. It points to NULL if there is no next node, indicating the end of the list.
* The first node of the list is called the **head**. It contains the starting point for traversing the list.
* The last node of the list points to NULL, indicating the end of the list.

**Operations on Singly Linked Lists:**

1. **Creation:** Creating an empty singly linked list involves initializing the head pointer to NULL.
2. **Insertion:** Nodes can be inserted at the beginning, end, or at any specific position in the list.
3. **Deletion:** Nodes can be deleted from the list based on their value or position.
4. **Traversal:** Traversing the list involves visiting each node in sequence, starting from the head, until reaching the end (NULL).
5. **Searching:** Searching for a specific element involves traversing the list and comparing the data values until a match is found or reaching the end of the list.

**Advantages of Singly Linked Lists:**

* **Dynamic Size:** Singly linked lists can dynamically grow or shrink in size during runtime, making them flexible for various applications.
* **Efficient Insertion and Deletion:** Insertion and deletion operations can be performed in constant time complexity O(1) at the beginning or end of the list.

**Disadvantages of Singly Linked Lists:**

* **Lack of Random Access:** Accessing an arbitrary element in a singly linked list requires traversing from the beginning of the list, resulting in linear time complexity O(n).
* **Extra Memory Overhead:** Each node in the list requires additional memory for storing the next pointer, increasing memory usage compared to arrays.
* **Insertion in a Singly Linked List:**
* **Insertion at the Beginning:**
* Allocate memory for a new node.
* Set the data of the new node.
* Point the next pointer of the new node to the current head.
* Update the head pointer to point to the new node.
* **Insertion at the End:**
* Allocate memory for a new node.
* Set the data of the new node.
* Traverse the list to find the last node.
* Point the next pointer of the last node to the new node.
* Set the next pointer of the new node to NULL if the list was empty, else point it to NULL.
* **Insertion at a Specific Position:**
* Allocate memory for a new node.
* Set the data of the new node.
* Traverse the list to find the node before the desired position.
* Adjust pointers to insert the new node at the desired position.
* **Deletion in a Singly Linked List:**
* **Deletion of the First Node:**
* Update the head pointer to point to the next node.
* Free the memory allocated for the first node.
* **Deletion of the Last Node:**
* Traverse the list to find the second-to-last node.
* Set the next pointer of the second-to-last node to NULL.
* Free the memory allocated for the last node.
* **Deletion of a Specific Node:**
* Traverse the list to find the node before the node to be deleted.
* Adjust pointers to bypass the node to be deleted.
* Free the memory allocated for the node to be deleted.
* **Traversing a Singly Linked List:**
* **Iterative Traversal:**
* Start from the head node.
* While the current node is not NULL:
* Process the data of the current node.
* Move to the next node by following the next pointer.
* **Recursive Traversal:**
* Base case: If the current node is NULL, return.
* Process the data of the current node.
* Recursively traverse the rest of the list starting from the next node.

let's illustrate a singly linked list example without code:

Consider a singly linked list representing a list of integers: 10 -> 20 -> 30 -> NULL.

In this example:

* The first node contains the integer value 10.
* The second node contains the integer value 20, with a reference pointing to the next node.
* The third node contains the integer value 30, with a reference pointing to NULL, indicating the end of the list.

Visually, the linked list would look like this:

rust

Copy code

10 -> 20 -> 30 -> NULL

In this representation:

* Each arrow (**->**) indicates the link between nodes.
* Each node contains an integer value and a pointer to the next node.
* The last node points to NULL, indicating the end of the list.
* **Time Complexity:**
* **Accessing an Element (Search):**
* Time Complexity: O(n)
* Explanation: In the worst case, you may need to traverse the entire list to find the desired element, resulting in linear time complexity.
* **Insertion at the Beginning:**
* Time Complexity: O(1)
* Explanation: Inserting a node at the beginning involves updating the head pointer, which can be done in constant time.
* **Insertion at the End:**
* Time Complexity: O(n)
* Explanation: In the worst case, you need to traverse the entire list to find the last node, resulting in linear time complexity.
* **Deletion at the Beginning:**
* Time Complexity: O(1)
* Explanation: Deleting the first node involves updating the head pointer, which can be done in constant time.
* **Deletion at the End:**
* Time Complexity: O(n)
* Explanation: In the worst case, you need to traverse the entire list to find the second-to-last node, resulting in linear time complexity.
* **Traversing the List:**
* Time Complexity: O(n)
* Explanation: Traversing the entire list requires visiting each node once, resulting in linear time complexity.
* **Space Complexity:**
* **Node Allocation:**
* Space Complexity: O(n)
* Explanation: Each node in the linked list requires memory allocation, resulting in linear space complexity proportional to the number of nodes in the list.
* **Additional Overhead:**
* Space Complexity: O(1)
* Explanation: Besides the node allocations, the singly linked list implementation typically requires only a constant amount of additional memory for maintaining pointers and metadata.

* **Introduction to Doubly Linked Lists**
* In computer science, a doubly linked list is a linear data structure similar to a singly linked list. However, in addition to a reference to the next node, each node in a doubly linked list also contains a reference to the previous node. This bidirectional linkage allows traversal in both forward and backward directions, providing greater flexibility in data manipulation compared to singly linked lists.
* **Basic Structure:**
* Each node in a doubly linked list contains three fields:
* **Data:** This field holds the actual value or data associated with the node.
* **Next Pointer:** This pointer/reference points to the next node in the sequence.
* **Previous Pointer:** This pointer/reference points to the previous node in the sequence.
* The first node in the list is called the **head**, and it serves as the starting point for traversal in the forward direction.
* The last node in the list is called the **tail**, and it serves as the ending point for traversal in the backward direction.
* **Key Features:**
* **Bidirectional Traversal:** Doubly linked lists support traversal in both forward and backward directions, allowing efficient navigation through the list.
* **Dynamic Insertion and Deletion:** Operations like insertion and deletion can be performed at both ends of the list and at any position, providing flexibility in data manipulation.
* **Advantages of Doubly Linked Lists:**
* **Bidirectional Traversal:** Allows efficient traversal in both forward and backward directions, enabling operations like reverse traversal and efficient removal of nodes from the end.
* **Dynamic Insertion and Deletion:** Supports efficient insertion and deletion operations at both ends and at any position in the list.
* **Applications:**
* Doubly linked lists find applications in situations where bidirectional traversal is required, such as in text editors (undo/redo functionality), browser history management, and implementing data structures like deques (double-ended queues).
* **1. Insertion in a Doubly Linked List:**
* **Insertion at the Beginning:**
* Allocate memory for a new node.
* Set the data of the new node.
* Point the next pointer of the new node to the current head.
* If the current head is not NULL, update the previous pointer of the current head to point to the new node.
* Update the head pointer to point to the new node.
* **Insertion at the End:**
* Allocate memory for a new node.
* Set the data of the new node.
* Traverse the list to find the last node.
* Point the next pointer of the last node to the new node.
* Set the previous pointer of the new node to point to the last node.
* Set the next pointer of the new node to NULL.
* **Insertion at a Specific Position:**
* Allocate memory for a new node.
* Set the data of the new node.
* Traverse the list to find the node before the desired position.
* Adjust pointers to insert the new node at the desired position.
* **2. Deletion in a Doubly Linked List:**
* **Deletion of the First Node:**
* If the list is empty, return.
* Update the head pointer to point to the next node.
* If the new head is not NULL, update its previous pointer to NULL.
* Free the memory allocated for the deleted node.
* **Deletion of the Last Node:**
* If the list is empty, return.
* Traverse the list to find the last node.
* Set the previous pointer of the last node's next node to NULL.
* Free the memory allocated for the last node.
* **Deletion of a Specific Node:**
* Traverse the list to find the node to be deleted.
* Adjust pointers of adjacent nodes to bypass the node to be deleted.
* Free the memory allocated for the deleted node.
* **3. Traversing a Doubly Linked List:**
* **Forward Traversal:**
* Start from the head node.
* While the current node is not NULL:
* Process the data of the current node.
* Move to the next node by following the next pointer.
* **Backward Traversal:**
* Start from the tail node.
* While the current node is not NULL:
* Process the data of the current node.
* Move to the previous node by following the previous pointer.

Consider a doubly linked of integers: 10 <-> 20 <-> 30 <-> NULL

* The first node (head) contains the integer value 10 and has a NULL prev pointer, indicating the beginning of the list.
* The second node contains the integer value 20, with pointers to the previous node (10) and the next node (30).
* The third node contains the integer value 30, with a pointer to the previous node (20) and a NULL next pointer, indicating the end of the list.

Visually, the doubly linked list would look like this:

rust

Copy code

NULL <-> 10 <-> 20 <-> 30 <-> NULL

In this representation:

* Each **<->** indicates bidirectional links between nodes.
* Each node contains an integer value and references to both the next and previous nodes.
* The first and last nodes have NULL pointers in the appropriate direction, indicating the start and end of the list.

Doubly linked lists offer advantages such as efficient insertion and deletion at both ends of the list and the ability to traverse the list in both forward and backward directions. However, they require more memory due to the additional prev pointers.

Top of Form

* **Introduction to** **circular linked list:** 
* Each node contains two fields:
* **Data Field:** This holds the value or data associated with the node.
* **Next Pointer:** This is a reference (pointer) to the next node in the sequence.
* Unlike linear linked lists, where the last node's next pointer points to NULL, in a circular linked list, the last node's next pointer points back to the first node, forming a closed loop.
* **Key Features:**
* **Circular Structure:** The circular arrangement enables efficient looping through elements without encountering a NULL pointer.
* **Seamless Traversal:** Traversal can begin from any node and continue indefinitely by following the next pointers, ultimately returning to the starting point.
* **Advantages of Circular Linked Lists:**
* **Efficient Looping:** Circular linked lists facilitate continuous looping without the need for additional checks for NULL pointers.
* **Disadvantages:**
* **Infinite Loop Risk:** Care must be taken to avoid infinite loops during traversal, as the circular structure may result in endless iteration if not managed properly.
* **Complexity:** Implementing and manipulating circular linked lists may require additional consideration compared to linear linked lists.
* **Applications:**
* Circular linked lists find application in various domains, including operating systems, where they are used for task scheduling algorithms, and in memory management systems for implementing circular buffers and queues.
* **1. Insertion in a Circular Linked List:**
* **Insertion at the Beginning:**
* Allocate memory for a new node.
* Set the data of the new node.
* If the list is empty, set the next pointer of the new node to point to itself.
* Otherwise, set the next pointer of the new node to point to the current head node.
* Update the next pointer of the last node to point to the new node.
* Update the head pointer to point to the new node.
* **Insertion at the End:**
* Allocate memory for a new node.
* Set the data of the new node.
* If the list is empty, set the next pointer of the new node to point to itself and update the head pointer.
* Otherwise, traverse the list to find the last node.
* Set the next pointer of the last node to point to the new node.
* Set the next pointer of the new node to point to the head node.
* Update the head pointer if necessary.
* **Insertion at a Specific Position:**
* Similar to insertion at the end, but traverse the list to find the node before the desired position.
* Adjust pointers to insert the new node at the desired position.
* **2. Deletion in a Circular Linked List:**
* **Deletion of a Node:**
* Find the node to be deleted.
* Update the next pointer of the previous node to skip the node to be deleted.
* Free the memory allocated for the deleted node.
* **3. Traversing a Circular Linked List:**
* **Forward Traversal:**
* Start from the head node.
* Process the data of each node and move to the next node until reaching the head node again.
* **Backward Traversal:**
* Start from the tail node (if available).
* Process the data of each node and move to the previous node until reaching the head node again.
* Consider a circular linked list representing a sequence of integers: 10 -> 20 -> 30 -> 10
* In this example:
* Each node contains an integer value.
* The last node's next pointer points back to the first node, forming a circular structure.
* Traversal can start from any node and continue indefinitely by following the next pointers.
* Visually, the circular linked list would look like this:
* rust
* Copy code
* +------+------+ +------+------+ +------+------+ Head -> | 10 | \*- - -+-> | 20 | \*- - -+-> | 30 | \*- - -+ +------+------+ +------+------+ +------+------+ ↑ | |\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|
* In this representation:
* Each rectangle represents a node in the circular linked list.
* Inside each rectangle, the number represents the value stored in that node.
* The arrows (**->**) indicate the next pointers, showing the direction of traversal.
* The last node's next pointer points back to the head node, creating a circular structure.
* This circular linked list can be traversed from any node, and traversal will continue indefinitely in a loop, following the next pointers until reaching the starting point again.
* **Conclusion of linked lists**
* Top of Form

In conclusion, linked lists are versatile data structures with various advantages and applications in computer science and software engineering. Understanding their properties, operations, and trade-offs is essential for designing efficient algorithms and systems.

Top of Form

Top of Form