Linked Lists

A linked list is a linear data structure where each element (called a node) is connected to the next one using pointers. Unlike array, elements of linked list are stored in random memory locations.

In this article, we will learn about the linked list, its types, representation of the linked list in C, and discuss what link list offers as compared to the similar data structures.

What is a Linked List?

A linked list is a sequence of nodes in which each node is made up of two parts:

- **Data**: The value stored in the node.
- **Pointer**: A reference to the next node in the sequence. *(There can be multiple pointers for different kind of linked list.)*

Unlike arrays, linked lists do not store nodes in contiguous memory locations. Instead, each node contains pointer to the next node, forming a chain-like structure and to access any element (node), we need to first sequentially traverse all the nodes before it.

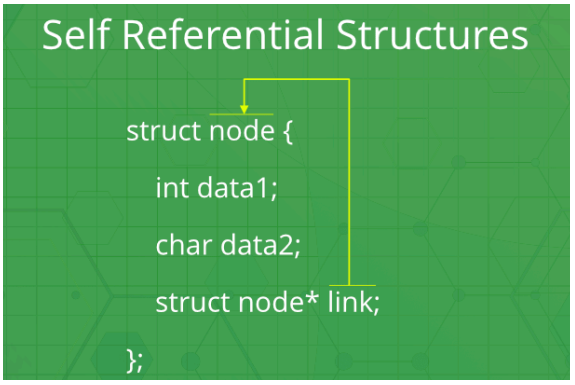
It is a recursive data structure in which any smaller part of it is also a linked list in itself.

Representation of Linked List in C

In C, linked lists are represented as the pointer to the first node in the list. For that reason, the first node is generally called **head** of the linked list. Each node of the linked list is represented by a structure that contains a data field and a pointer of the same type as itself. Such structure is called self-referential structures

Self Referential Structures

Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.



In other words, structures pointing to the same type of structures are self-referential in nature

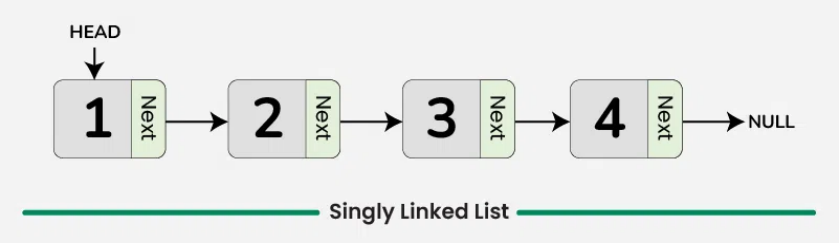
Types of Linked List in C

Linked list can be classified on the basis of the type of structure they form as a whole and the direction of access. Based on this classification, there are five types of linked lists:

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

Singly Linked List in C

A linked list or singly linked list is a linear data structure that is made up of a group of nodes in which each node has two parts: the data, and the pointer to the next node. The last node's (also known as tail) pointers point to NULL to indicate the end of the linked list.



Representation of Singly Linked List

A linked list is represented as a pointer to the first node where each node contains:

Data: Here the actual information is stored.

Next: Pointer that links to the next node.

```
// Structure to represent the
// singly linked list
struct Node {

  // Data field - can be of
  // any type and count
  int data;

  // Pointer to the next node
  struct Node* next;
}
```

Basic Operations on Singly Linked List

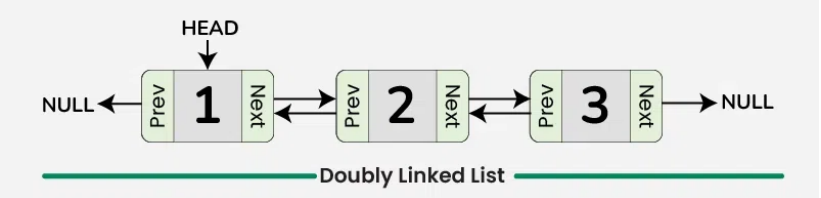
Operation	Operation Type	Description	Time Complexity	Space Complexity
Insertion	At Beginning	Insert a new node at the	O (1)	O (1)

		start of a linked list.		
	At the End	Insert a new node at the end of the linked list.	O (N)	O (1)
	At Specific Position	Insert a new node at a specific position in a linked list.	O (N)	O (1)
Deletion	**From Beginning**	Delete a node from the start of a linked list	O (1)	O (1)
	From the End	Delete a node at the end of a linked list.	O (N)	O (1)

	A Specific Node	Delete a node from a specific position of a linked list.	O (N)	O (1)
Traversal		Traverse the linked list from start to end.	O (N)	O (1)

Doubly Linked List

A doubly linked list is a bit more complex than singly linked list. In it, each node contains three parts: the data, a pointer to the next node, and one extra pointer which points to the previous node. This allows for traversal in both directions, making it more versatile than a singly linked list.



Doubly Linked List in C

A doubly linked list is a type of linked list in which each node contains 3 parts, a data part and two addresses, one points to the previous node and one for the next node. It differs from the singly linked list as it has an extra pointer called previous that points to the previous node, allowing the traversal in both forward and backward directions.

In this article, we will learn about the doubly linked list implementation in C. We will also look at the working of the doubly linked list and the basic operations that can be performed using the doubly linked list in C.

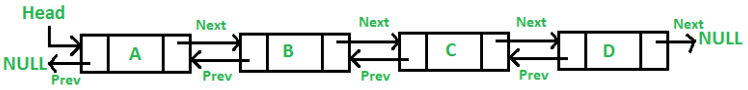
Doubly Linked List Representation in C

A doubly linked list is represented in C as the pointer to the head (i.e. first node) in the list. Each node in a doubly linked list contains three components:

Data: data is the actual information stored in the node.

Next: next is a pointer that links to the next node in the list.

Prev: previous is a pointer that links to the previous node in the list.

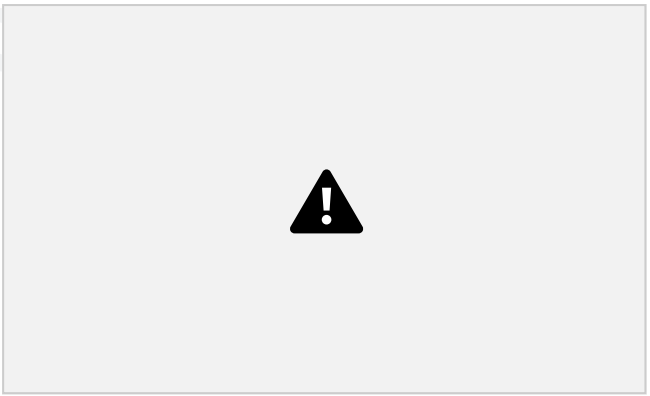


Implementation of Doubly Linked List in C

To implement a doubly linked list in C first, we need to define a node that has three parts: data, a pointer to the next node, and a pointer to the previous node, and then create a new node.

We can create a node using the structure which allows us to combine different data types into single type.

Define a Node in Doubly Linked List



To define a structure of a node in doubly linked list use the below format:

```
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
}
```

We can then dynamically create the node using malloc() and assign the values to the next, prev and data fields. It is generally preferred to create a function that creates a node, assign the data field and return the pointer to that node.

Basic Operations on C Doubly Linked List

We can perform the following basic operations in a doubly-linked list:

Insertion

Deletion

Traversal

1. Insertion in Doubly Linked List in C

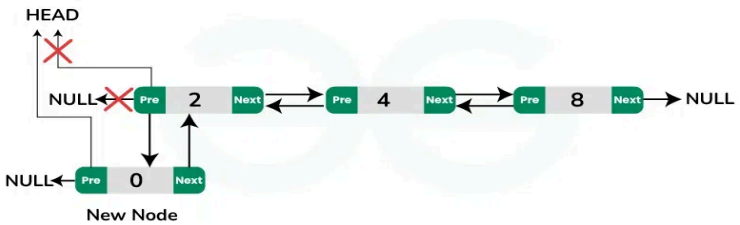
Inserting a new node in a doubly linked list can be done in a similar way like inserting a new node in a singly linked list but we have to maintain the link of previous node also. We have three scenarios while inserting a node in a doubly linked list:

Insertion at the beginning of the list.

Insertion at the end of the list.

Insertion in the middle of the list

Insertion at the Beginning of the Doubly Linked List



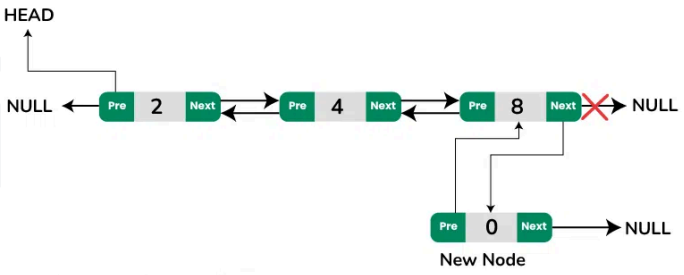
To insert a new node at the beginning of the doubly linked list follow the below approach:

Approach:

- *Create a newNode with **data** field assigned the given value.*
- *If the list is empty:*
    - *Set newNode->next to NULL.*
    - *Set newNode->prev to NULL.*
    - *Update the head pointer to point to newNode.*
- *If the list is not empty:*
    - *Set newNode->next to head.*
    - *Set newNode->prev to NULL.*
    - *Set head->prev to newNode.*
    - *Update the head pointer to point to newNode.*

Time Complexity: O(1), as insertion is done at front so we are not traversing through the list.
Space Complexity: O(1)

Insertion at the End of the Doubly Linked List



To insert a new node at the end of the doubly linked list, follow the below approach:

Approach:

Create a newNode with data field assigned the given value.

If the list is empty:

Set newNode->next to NULL.

Set newNode->prev to NULL.

Update the head pointer to point to newNode.

If the list is not empty:

Traverse the list to find the last node (where last->next == NULL).

Set last->next to newNode.

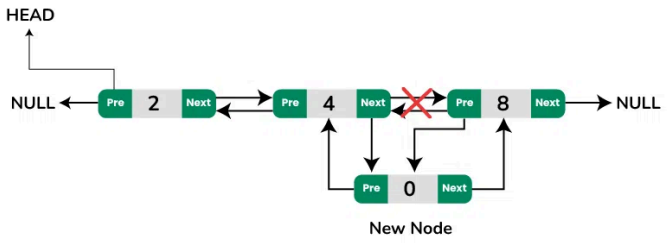
Set newNode->prev to last.

Set newNode->next to NULL.

Time Complexity: O(n), as insertion is done at the end, so we need to traverse through the list.

Space Complexty: O(1)

Insertion in the Middle of the Doubly Linked List



For inserting a node at a specific position in a doubly linked list follow the below approach:

Approach:

- *Create a newNode.*
- *Find the size of the list to ensure the position is within valid bounds.*

- *If the position is 0 (or 1 depending on your indexing):*
    - *Use the insertion at the beginning approach.*
- *If the position is equal to the size of the list:*
    - *Use the insertion at the end approach.*
- *If the position is valid and not at the boundaries:*
    - *Traverse the list to find the node immediately before the desired insertion point (prevNode).*
    - *Set newNode->next to prevNode->next.*
    - *Set newNode->prev to prevNode.*
    - *If prevNode->next is not NULL, set prevNode->next->prev to newNode.*
    - *Set prevNode->next to newNode.*