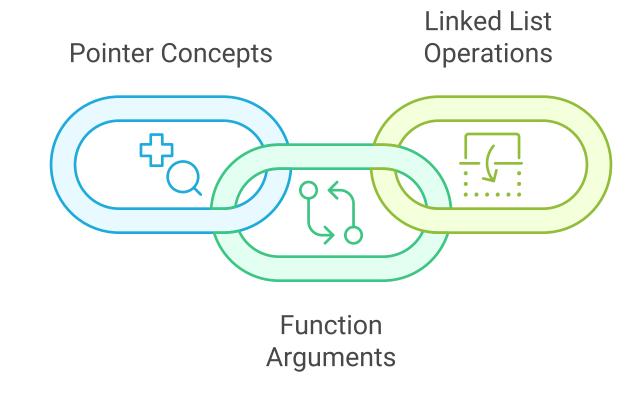
## Presentation on Linked Lists using Pointers

This document provides an overview of linked lists and their implementation using pointers in the C programming language. It covers the fundamental concepts of pointers, their usage in function arguments, and the various operations associated with linked lists. The goal is to highlight the efficiency and flexibility that pointers bring to dynamic memory management and data structures.

Mastering Linked Lists



## **Definition**

**Introduction to Pointers** 

### A pointer is a variable that stores the memory address of another variable. This allows for direct access to the variable's value and enables manipulation of data in memory.

Usage

## • **Dynamic Memory Allocation**: Allocating memory at runtime.

• Data Structures: Implementing complex data structures like linked lists, trees, etc.

In C, pointers are extensively used for:

- Efficient Function Argument Passing: Reducing overhead by passing addresses
- instead of copying large data structures.
- Passing Pointers as Function Arguments

## **Direct Access**

## that changes made to the variable inside the function reflect outside the function as well.

The Power of Pointers

Pointers allow functions to modify variables directly by passing their addresses. This means

## Pointers enable direct variable access and **Pointer Usage** modification. **Direct Variable** Modification Passing addresses **Address** allows external **Passing** function variable changes.

**NODE** 

**Efficiency** 

Linked List Operations (Based on Code)

A **node** in a linked list is the fundamental building block that stores data and a reference

**Linked List Node Structure** 

**Data** 

node

**Insert at End** 

Appends a new node at the

end, extending the list.

**Remove from End** 

list.

Deletes the last node, useful

for removing the tail of the

Information stored in the

Passing pointers instead of copying large structures saves both memory and execution time.

This is particularly beneficial in scenarios where large datasets are involved.

## (pointer) to the next node.



• Insert Before/After a Given Node: Place a new node before or after a specified node

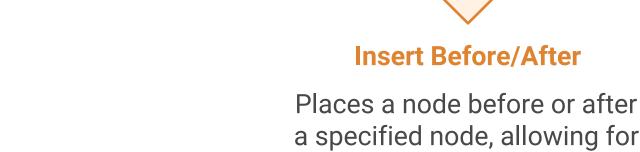
How to insert a node in a linked list?

• Insert at Front: Add a new node at the beginning of the list.

• Insert at End: Append a new node at the end of the list.

**Insertion Operations** 

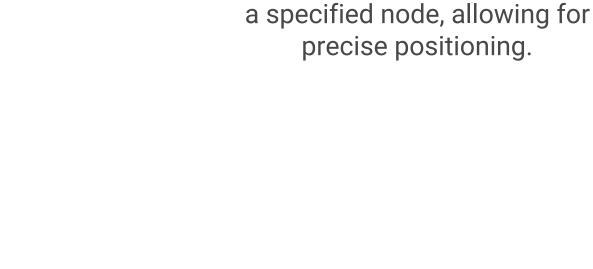
in the list.



# **Insert at Front**

head of the list.

beginning, making it the new



# Adds a new node at the

Node

How to delete a node in a linked list?

**Delete Specific Node** 

Removes a node based on

value or position, providing

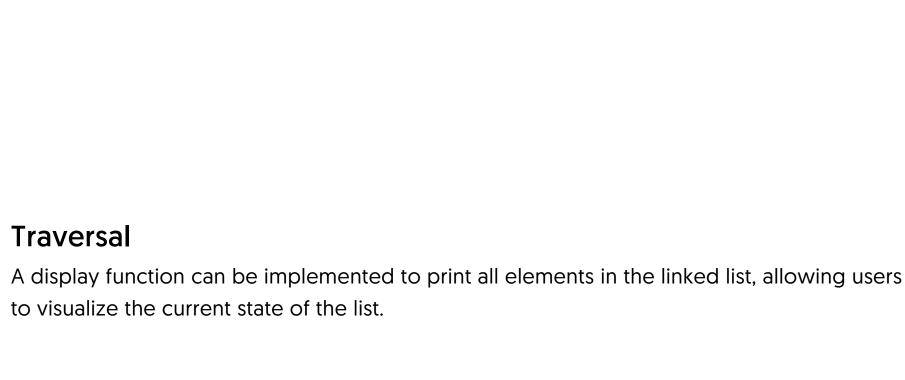
targeted deletion.

list.

**Remove from Front** 

Deletes the first node, useful

for removing the head of the



# Linked List Display Cycle

**Implement Display** 

**Function** 

Create a function to display list elements.

# DFX **Print Elements Traverse Linked** List Output the elements to the user. Move through each node in the list.

# **Key Takeaways**

- Pointers make dynamic memory management efficient by allowing direct access to memory locations.
  - Functions that accept pointer arguments allow for direct manipulation of data, enhancing performance and reducing memory overhead.
  - Linked lists utilize pointers for flexible memory allocation, enabling dynamic resizing and efficient data handling.

## **Deletion Operations** • Remove from Front: Delete the first node of the list. • Remove from End: Delete the last node of the list. • Delete a Specific Node: Remove a node based on a given value or position.