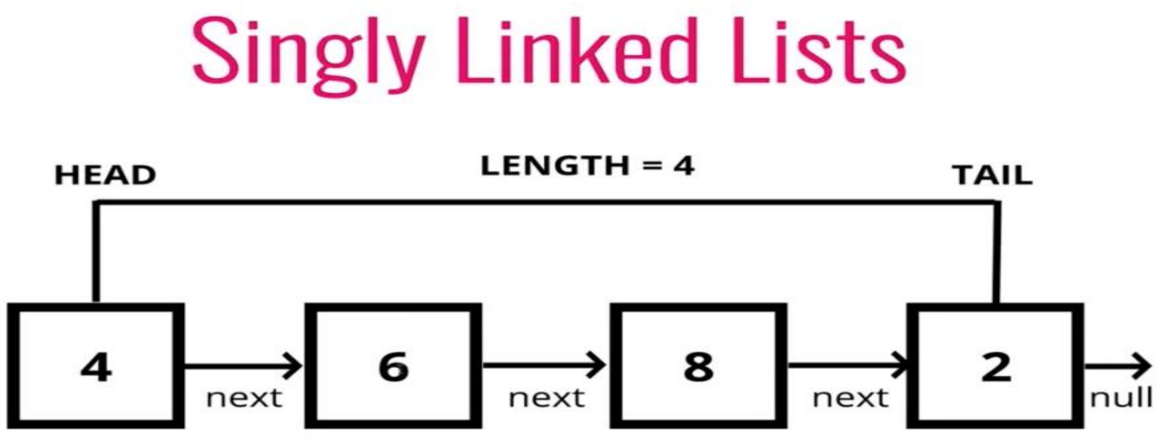
LINKED LIST

Linked lists are essential data structures in computer science, allowing for efficient data management and manipulation. Unlike arrays, linked lists are dynamic, meaning their size can change during runtime, which provides flexibility in handling data. This unit covers the concepts of singly linked lists, operations on linked lists, linked stacks and queues, polynomial representations using linked lists, and doubly linked lists.

1. Singly Linked List

Definition

A **singly linked list** is a linear collection of elements, called nodes, where each node contains two components: a data field and a pointer to the next node in the sequence. The first node is known as the **head**, and the last node points to null, indicating the end of the list.



Structure of a Singly Linked List

- **Node**: Each node consists of two parts:
    - **Data**: The value or information stored in the node.
    - **Next Pointer**: A reference to the next node in the list.
- **Head**: The starting point of the list, which points to the first node.

Operations on Singly Linked Lists

Singly linked lists support several operations, including:

1. **Insertion**:
    - **At the Beginning**: A new node is created and its next pointer points to the current head. The head is then updated to this new node.
    - **At the End**: Traverse the list until the last node is reached, then set the last node's next pointer to the new node.
    - **At a Specific Position**: Traverse to the desired position, adjust pointers accordingly to include the new node.
2. **Deletion**:
    - **From the Beginning**: Update the head to point to the second node.
    - **From the End**: Traverse to the second-to-last node and set its next pointer to null.

    - **From a Specific Position**: Traverse to the node before the target node and adjust pointers to bypass the target node.

3. **Traversal**: Accessing each node in the list sequentially, starting from the head and moving through each next pointer.
4. **Search**: Finding a node with a specific value by traversing the list and comparing data values.

Advantages and Disadvantages

Advantages:

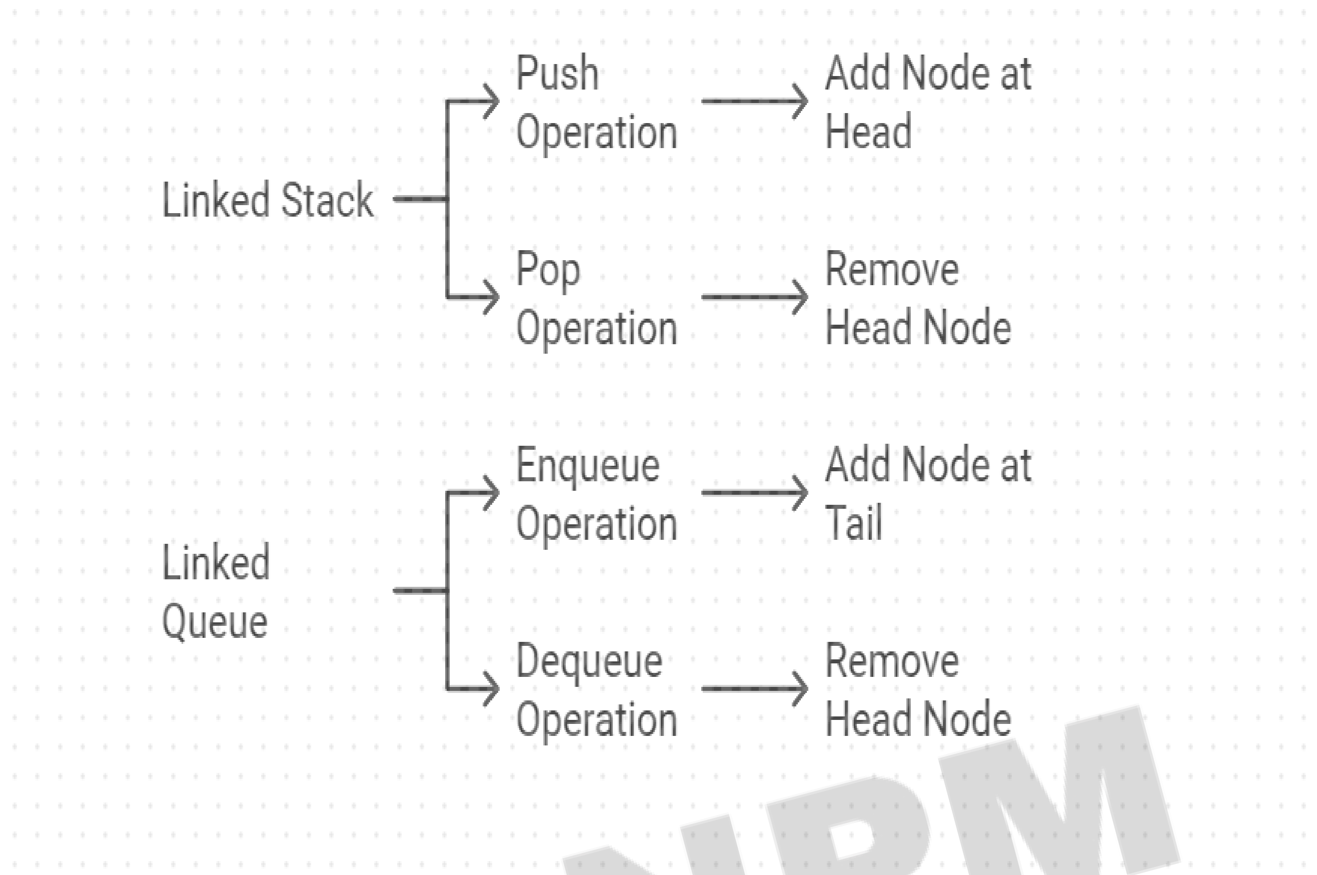
- Dynamic size: The list can grow or shrink as needed.
- Efficient insertions and deletions at any position.

Disadvantages:

- No direct access to elements; must traverse from the head.
- Extra memory usage due to pointers.

2. Linked Stacks and Queues

Linked lists are commonly used to implement **stacks** and **queues**, both of which are fundamental data structures.



Linked Stack

A **stack** operates on a Last-In-First-Out (LIFO) principle, where the last element added is the first to be removed. In a linked stack, nodes are added or removed from the head of the linked list.

Operations on a Linked Stack:

1. **Push**: Add a new node at the head of the linked list.
2. **Pop**: Remove the head node and return its data.
3. **Peek**: Access the data of the head node without removing it.

Linked Queue

A **queue** operates on a First-In-First-Out (FIFO) principle, where the first element added is the first to be removed. A linked queue typically uses two pointers: one for the head (front) and one for the tail (rear).

Operations on a Linked Queue:

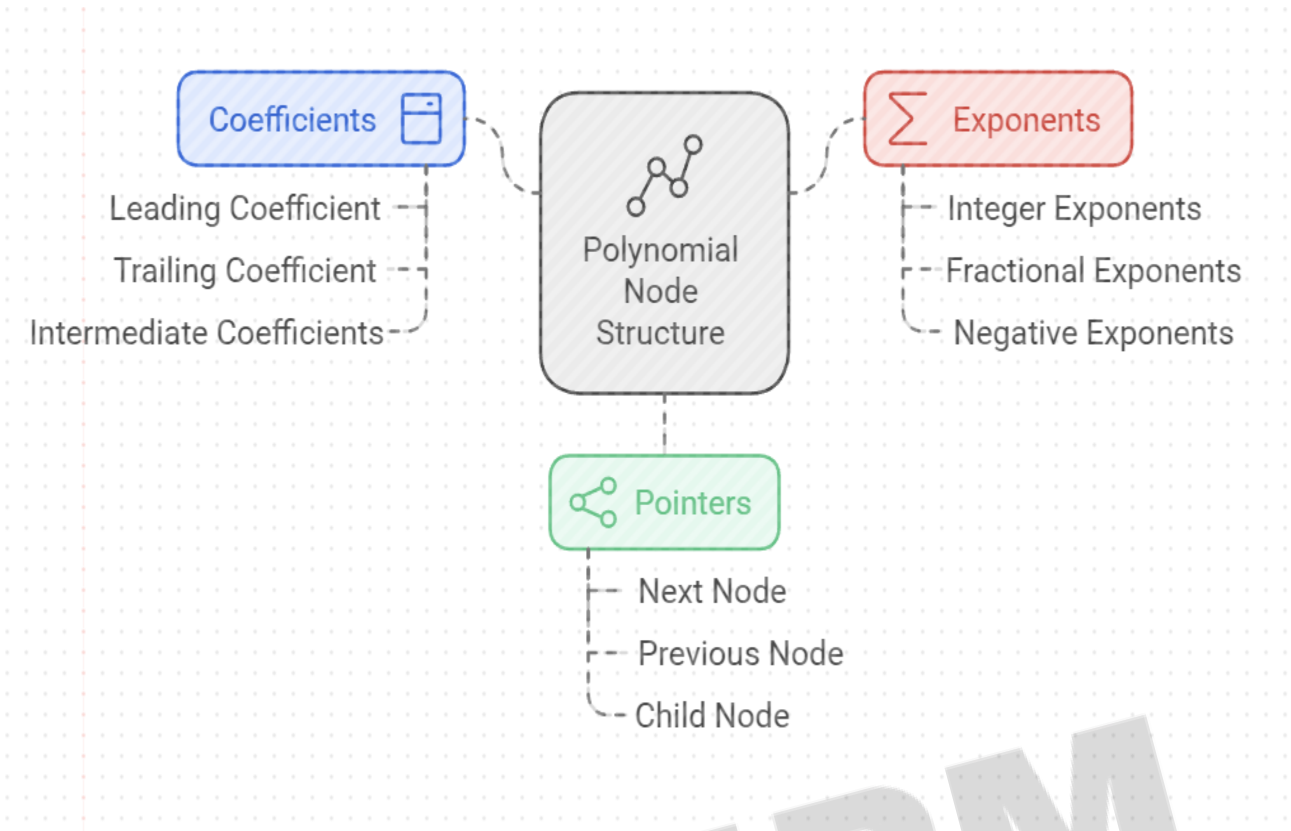
1. **Enqueue**: Add a new node at the tail of the linked list.
2. **Dequeue**: Remove the head node and return its data.
3. **Front**: Access the data of the head node without removing it.

Visualization of Stacks and Queues

- **Linked Stack**: A diagram shows how nodes are added or removed from the top of the stack.
- **Linked Queue**: A diagram illustrates the addition of nodes at the rear and removal from the front.

3. Polynomial Representations Using Linked Lists

Polynomials can be represented using linked lists, where each node corresponds to a term of the polynomial. Each node typically contains the coefficient and the exponent of the term.



Structure of a Polynomial Node

- **Coefficient**: Numeric value representing the term's coefficient.
- **Exponent**: The power associated with the variable in the term.
- **Next Pointer**: Reference to the next term in the polynomial.

Operations on Polynomial Linked Lists

1. **Addition of Polynomials**: To add two polynomial lists:
    - Traverse both lists simultaneously.
    - Compare the exponents:

        - If exponents are equal, sum the coefficients and create a new node for the result.
        - If one exponent is greater, add that term directly to the result.
    - Continue until all terms from both lists have been processed.
2. **Subtraction of Polynomials**: Similar to addition, but subtract coefficients when exponents match.
3. **Multiplication of Polynomials**: Multiply each term of one polynomial by every term of the other, then combine like terms in the result.
4. **Evaluation of Polynomials**: To evaluate a polynomial for a specific value of the variable:
    - Traverse the linked list and compute the result using the formula result=coefficient×x exponent

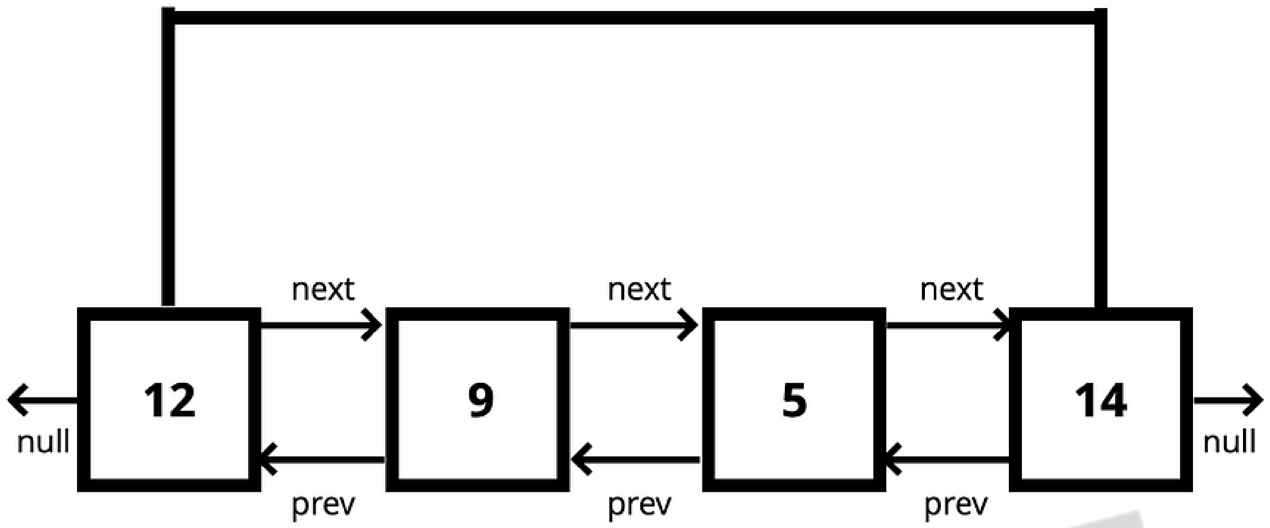
Visualization of Polynomial Operations

- **Polynomial Node Structure**: A diagram illustrating how a polynomial term is represented in a linked list.
- **Addition of Polynomials**: A flowchart showing the steps involved in adding two polynomial lists.

4. Doubly Linked List

A **doubly linked list** enhances the singly linked list by including an additional pointer in each node, allowing traversal in both directions (forward and backward).

Doubly Linked Lists



Structure of a Doubly Linked List

- **Node**: Contains three parts:
    - **Previous Pointer**: Reference to the previous node.
    - **Data**: The value stored in the node.
    - **Next Pointer**: Reference to the next node.
- **Head**: Points to the first node, and the last node's next pointer points to null.

Advantages of Doubly Linked Lists

- **Bidirectional Traversal**: Allows easy navigation in both forward and backward directions.
- **Easier Deletion**: Can directly access the previous node without traversing from the head.

Operations on Doubly Linked Lists

1. **Insertion**: Add nodes at the beginning, end, or a specific position while updating both next and previous pointers.
2. **Deletion**: Remove nodes from any position, adjusting pointers for both adjacent nodes.
3. **Traversal**: Access nodes in both forward and backward directions.

Visualization of Doubly Linked Lists

- **Doubly Linked List Structure**: A diagram illustrating the two pointers in each node and how they connect.

5. Addition of Two Polynomial Lists Using Doubly Linked Lists

Adding two polynomial lists represented as doubly linked lists follows a similar process to singly linked lists but benefits from the bidirectional traversal.

Steps to Add Two Polynomial Lists

1. **Initialization**: Create a new doubly linked list to store the result.
2. **Traversal**: Use two pointers to traverse both polynomial lists.
3. **Comparison**: Compare the exponents of the current nodes:
    - If they match, sum the coefficients and create a new node in the result list.

    - If one exponent is greater, add that term to the result.

4. **Completion**: After processing all terms, link any remaining terms from either polynomial to the result.

Visualization of Polynomial Addition

- **Addition Process Diagram**: A step-by-step illustration of how two polynomial linked lists are added, showing the comparison and node creation process.