**MODULE – 1**

**Syllabus: -**

Why do we need data structure? Concepts of data structures: a) Data and data structure b) Abstract Data Type and Data Type; Algorithms and programs, basic idea of pseudo-code. Algorithm efficiency and analysis, time, and space analysis of algorithms – Big O, Ω, Θ, notations.

**Array**: Different representations – row major, column major. ~~Sparse matrix - its implementation and usage. Array representation of polynomials.~~

**Linked List**: Singly linked list, circular linked list, doubly linked list, linked list representation of polynomial and applications.

**Why do we need data structure?**

* Data Structures and Algorithms are two of the important aspects of Computer Science.
* Data Structures allow us to organize and store the data, whereas Algorithms allow us to process that data meaningfully.
* Learning Data Structures and Algorithms will help us to become better Programmers.
* We will be able to write code that is more effective and reliable.

**Data** – Data are a collection of values and the relationships among them and the functions or operations that can be applied to the data.

**Data Structure**  - Data Structure is a way of organizing and storing data in a computer so that it can be accessed and used efficiently. It refers to the logical or mathematical representation of data, as well as the implementation in a computer program.



**Abstract Data Type (ADT)** – Abstract Data Type are like user defined data types which defines operations and values using functions without specifying what is there inside the function and how the operations are performed.

The process of only providing the essentials and hiding the details is known as **Abstraction**.

There are 3 types of abstract data types: -

1. **List ADT** - Lists are Linear data structures in which the data is stored in a non-continuous system. Lists consists of *“node”* where the data is stored. These nodes are linked to each other that is each node consists of an *“address box”* which contains the address of the next node. It may follow LIFO (Last In First Out) or FIFO (First In First Out).

**A diagram of a diagram

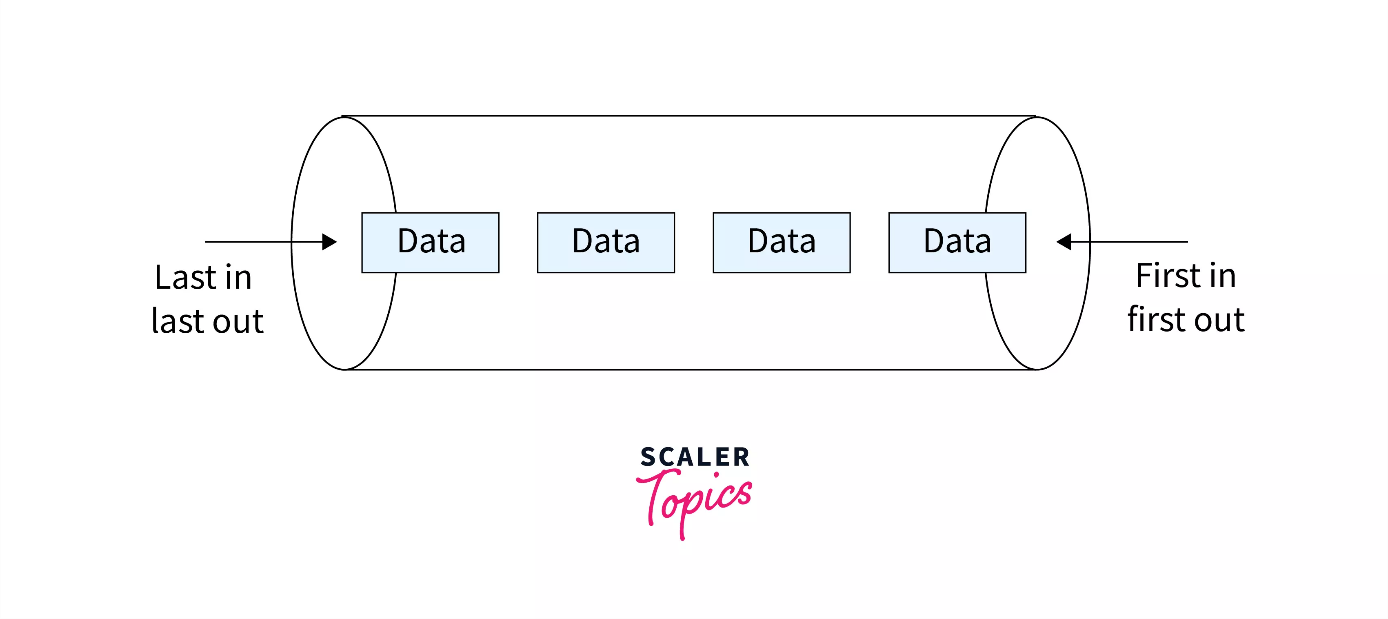
Description automatically generated with medium confidence**

1. **Stack ADT** - Stack is a Linear data structure in which data can be only accessed from its top only. It can only perform 2 operations – *push*(used to insert data to the top of the stack) & *pop*(used to remove data from the top of the stack). It follows LIFO (Last In First Out).

**A diagram of a diagram

Description automatically generated**

1. **Queue ADT** - Queue is Linear data structure in which the data can be accessed from both ends that is front and rear. It only has 2 operations – push(used to insert data from the rear end of the queue) & *pop*(used to remove data from the front end of the queue). It follows FIFO (First In First Out).

****

**Efficiency of an Algorithm**

Computer resources are limited so it should be used efficiently. The efficiency of an algorithm is defined as the no.of computational resources used by the algorithm. An algorithm must be analyzed to determine its different kinds of resource usage.

The efficiency of an algorithm depends on 2 parameters: -

1. Time Complexity
2. Space Complexity

**Time complexity** - It is a concept in computer science that deals with the amount of time taken by a set of code or algorithm to process or run as a function of the amount of input(n).

**Space Complexity** - The space complexity of algorithm is the amount of space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary Space and space used by input.

(Auxiliary Space - It is the extra or temporary space used by an algorithm.)



**Asymptotic Notations** - Asymptotic Notation is a mathematical notation used to analyze the time complexity and runtime of an algorithm for a large input. It describes the efficiency of algorithms based on the time and space with respect to the input size. They also provide information about the performance and scalability of the algorithm.

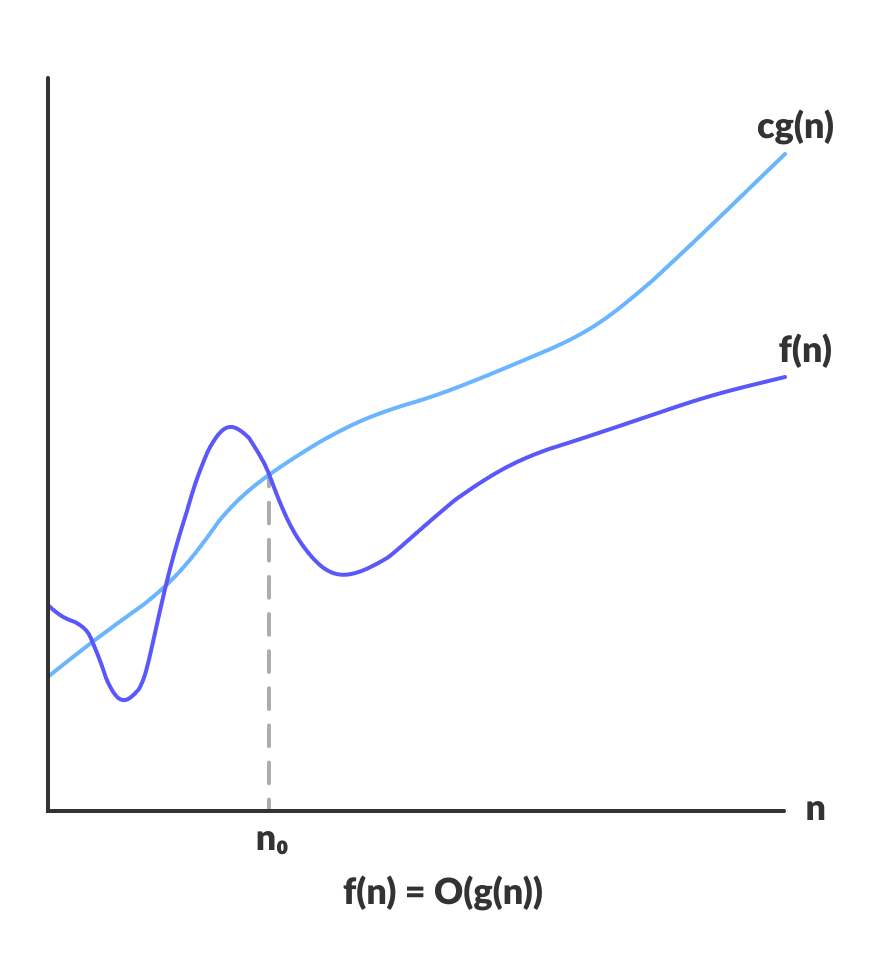
There are 3 main asymptotic notations: -

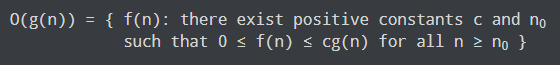
1. Big-O Notation
2. Omega(Ω) Notation
3. Theta(Θ) Notation

Note: - **‘n’** represents the size of the input.

**Big-O Notation** (Mostly used)

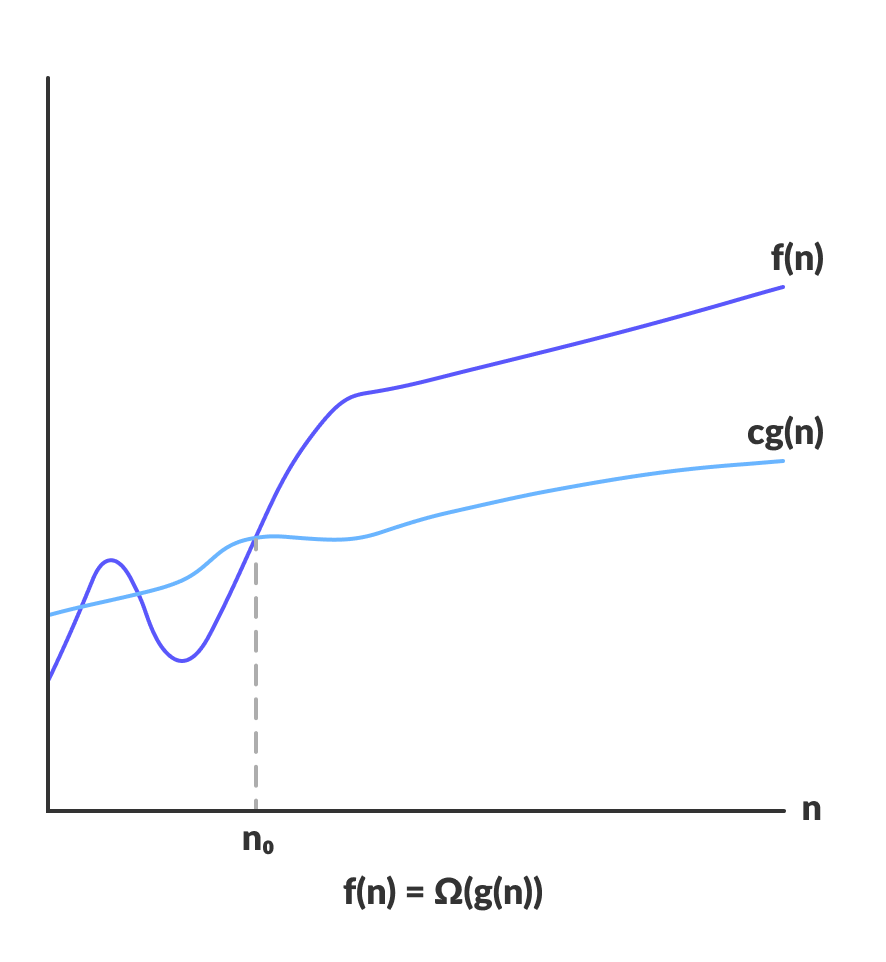
Big-O notation represents the upper bound of the running time of an algorithm. Thus it gives the **worst time** **complexity** of an algorithm.





**Omega(Ω) Notation**

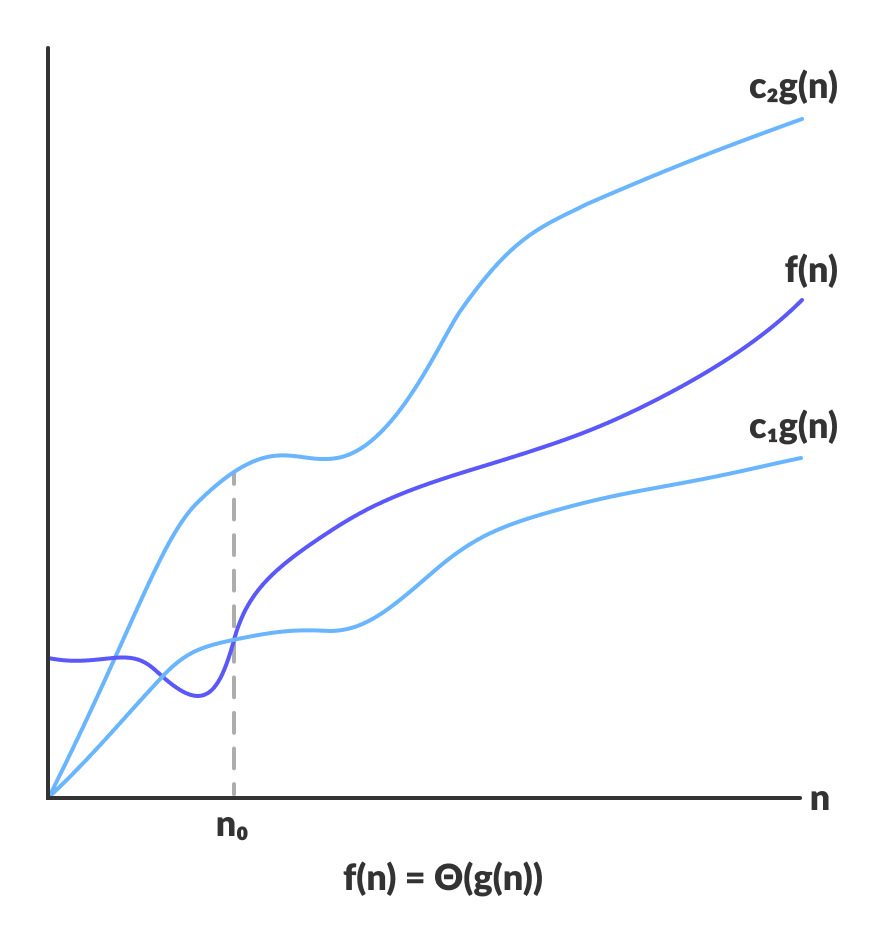
Omega notation represents the lower bound of the running time of an algorithm. Thus it provides the **best case complexity** of an algorithm.





**Theta(Θ) Notation**

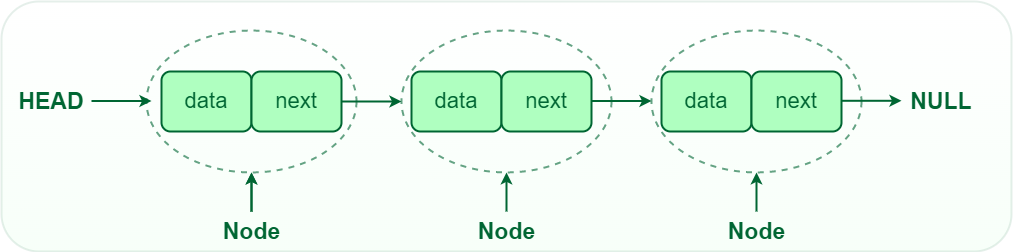
Theta notation encloses the function from above and below. Since it represents the upper and lower bound of the running time of an algorithm, it is used for analyzing the **average case complexity** of an algorithm.





**Linked List**

A Linked List is a data structure that stores a sequence of elements, where each element is called a node and each node contains the address of the next node. The first node is known as Head and the last node is known as Tail.



In linked list each node structure consists of 2 parts: -

1. Data - It holds the actual value / data
2. Next Pointer - It stores the memory address of the next node

Operations on Linked List

1. **Insertion** - dding a new node to a linked list involves adjusting the pointers of the existing nodes to maintain the proper sequence. Insertion can be performed at the beginning, end, or any position within the list
2. **Deletion** - Removing a node from a linked list requires adjusting the pointers of the neighboring nodes to bridge the gap left by the deleted node. Deletion can be performed at the beginning, end, or any position within the list.
3. **Searching** - Searching for a specific value in a linked list involves traversing the list from the head node until the value is found or the end of the list is reached.

Advantages of Linked List

1. Dynamic Size
2. Efficient insertion & deletion
3. Easy to implement
4. Flexibility
5. Easy to navigate

Disadvantages of Linked List

1. Slow Access Time
2. Pointers High Overhead
3. Cache Inefficiency
4. Extra Memory required

Applications of Linked List

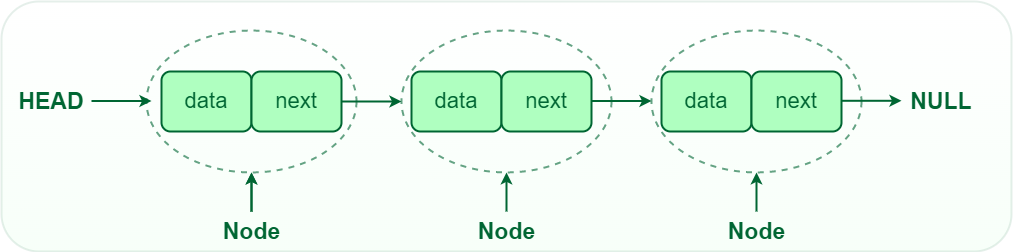
1. Used to represent Polynomials
2. Implementing stacks and queues
3. Arithmetic operations on long integers

There are 3 types of Linked List: -

1. Single Linked List
2. Double Linked List
3. Circular Linked List

**Single Linked List**

In a singly linked list, each node contains a reference to the next node in the sequence. Traversing a singly linked list is done in a forward direction.



PSEUDO CODE

1. Insert at beginning

Step 1: BEGIN

Step 2: Check overflow condition and create newnode

Step 3: newnode -> data = num

Step 4: newnode -> next = head

Step 5: head = newnode

Step 6: END

1. Insert after a node

Step 1: BEGIN

Step 2: Check overflow condition and create newnode

Step 3: newnode -> data = num

Step 4: ptr = head

Step 5: REPEAT Step 6, 7

Step 6: preptr = ptr

Step 7: ptr = ptr -> next

Step 8: UNTILL preptr -> data != value

Step 9: preptr -> next = newnode

Step 10: newnode -> next = ptr

Step 11: END

1. Insert at the end

Step 1: BEGIN

Step 2: Check overflow condition and create a newnode

Step 3: newnode -> data = num

Step 4: newnode -> next = NULL

Step 5: ptr = head

Step 6: REPEAT Step 7

Step 7: ptr = ptr -> next

Step 8: UNTILL ptr -> next != NULL

Step 9: ptr -> next = newnode

Step 10: END

1. Delete first node

Step 1: BEGIN

Step 2: Check underflow condition

Step 3: ptr = head

Step 4: head = ptr -> next

Step 5: FREE ptr

Step 6: END

1. Delete a certain node

Step 1: BEGIN

Step 2: Check underflow condition

Step 3: ptr = preptr = head

Step 4: REPEAT Step 5, 6

Step 5: preptr = ptr

Step 6: ptr = ptr -> next

Step 7: UNTILL preptr -> data != value

Step 8: preptr -> next = ptr -> next

Step 9: FREE ptr

Step 10: END

1. Delete last node

Step 1: BEGIN

Step 2: Check underflow condition

Step 3: ptr = preptr = head

Step 4: REPEAT Step 5, 6

Step 5: preptr = ptr

Step 6: ptr = ptr -> next

Step 7: UNTILL ptr -> next != NULL

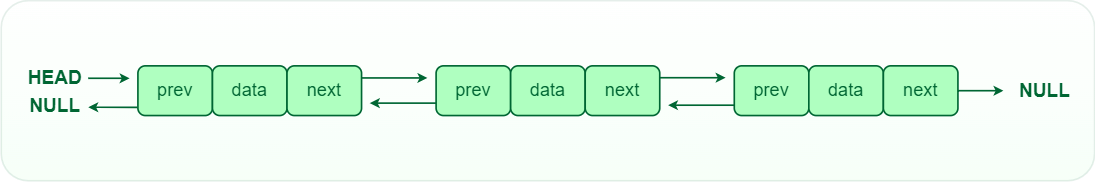
Step 8: preptr -> next = NULL

Step 9: FREE ptr

Step 10: END

**Double Linked List**

In a doubly linked list, each node contains references to both the next and previous nodes. This allows for traversal in both forward and backward directions, but it requires additional memory for the backward reference.



PSEUDO CODE

1. Insert at first

Step 1: BEGIN

Step 2: Check overflow condition and create newnode

Step 3: newnode -> data = num

Step 4: ptr = head

Step 5: newnode -> next = head

Step 6: newnode -> prev = NULL

Step 7: ptr -> prev = newnode

Step 8: head = newnode

Step 9: END

1. Insert after a node

Step 1: BEGIN

Step 2: Check ovewrflow condition and create a newnode

Step 3: newnode -> data = num

Step 4: ptr = preptr = head

Step 5: REPEAT Step 5, 6

Step 6: preptr = ptr

Step 7: ptr = ptr -> next

Step 8: UNTIL preptr -> data != value

Step 9: preptr -> next = newnode

Step 10: newnode -> next = ptr

Step 11: newnode -> prev = preptr

Step 12: ptr -> prev = newnode

Step 13: END

1. Insert at the end

Step 1: BEGIN

Step 2: Check overflow condition and create a newnode

Step 3: newndoe -> data = num

Step 4: newnode -> next = NULL

Step 5: ptr = head

Step 6: REPEAT Step 7

Step 7: ptr = ptr -> next

Step 8: UNTILL ptr -> next != NULL

Step 9: ptr -> next = newnode

Step 10: newnode -> prev = ptr

Step 11: END

1. Delete first node

Step 1: BEGIN

Step 2: Check underflow condition

Step 3: ptr = head

Step 4: head = ptr -> next

Step 5: head -> prev = NULL

Step 6: FREE ptr

Step 7: END

1. Delete a node

Step 1: BEGIN

Step 2: Check underflow condition

Step 3: ptr = preptr = head

Step 4: REPEAT step 5,6

Step 5: preptr = ptr

Step 6: ptr = ptr -> next

Step 7: UNTILL preptr -> data != value

Step 8: preptr -> next = ptr -> next

Step 9: (ptr -> next) -> prev = preptr

Step 9: FREE ptr

Step 10: END

1. Delete last node

Step 1: BEGIN

Step 2: Check underflow condition

Step 3: preptr = ptr = head

Step 4: REPEAT Step 5, 6

Step 5: preptr = ptr

Step 6: ptr = ptr -> next

Step 7: UNTILL ptr -> next != NULL

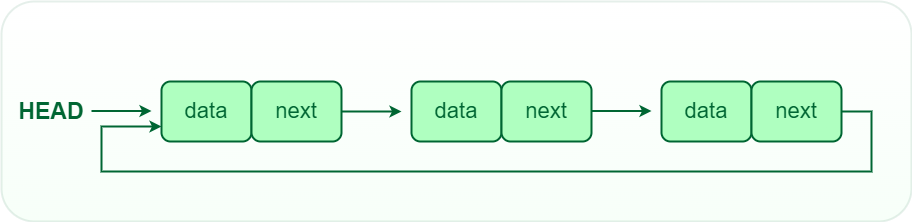
Step 8: preptr -> next = NULL

Step 9: FREE ptr

Step 10: END

**Circular Linked List**

 In a circular linked list, the last node points back to the head node, creating a circular structure. It can be either singly or doubly linked.



PSEUDO CODE

1. Inserting at front

Step 1: BEGIN

Step 2: Check overflow condition and create newnode

Step 3: newnode -> data = num

Step 4: newnode -> next = head

Step 5: ptr = head

Step 6: REPEAT Step 7

Step 7: ptr = ptr -> next

Step 8: UNTILL ptr -> next != head

Step 9: ptr -> next = newnode

Step 10: head = newnode

Step 11: END

1. Inserting at end

Step 1: BEGIN

Step 2: check overflow condition and create newnode

Step 3: newnode -> data = num

Step 4: newnode -> next = head

Step 5: ptr -> head

Step 6: REPEAT Step 7

Step 7: ptr = ptr -> next

Step 8: UNTILL ptr -> next != head

Step 9: ptr -> next = newnode

Step 10: END

1. Delete first node

Step 1: BEGIN

Step 2: Check underflow condition

Step 3: ptr = head

Step 4: REPEAT Step 5

Step 5: ptr = ptr -> next

Step 6: UNTILL ptr -> next != head

Step 7: ptr -> next = head -> next

Step 8: FREE head

Step 9: head = ptr -> next

Step 10: END

1. Delete last node

Step 1: BEGIN

Step 2: check underflow condition

Step 3: ptr = head

Step 4: REPEAT Step 5, 6

Step 5: preptr = ptr

Step 6: ptr = ptr -> next

Step 7: UNTILL ptr -> next != head

Step 8: preptr -> next = head

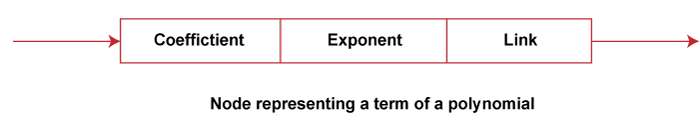
Step 9: FREE ptr

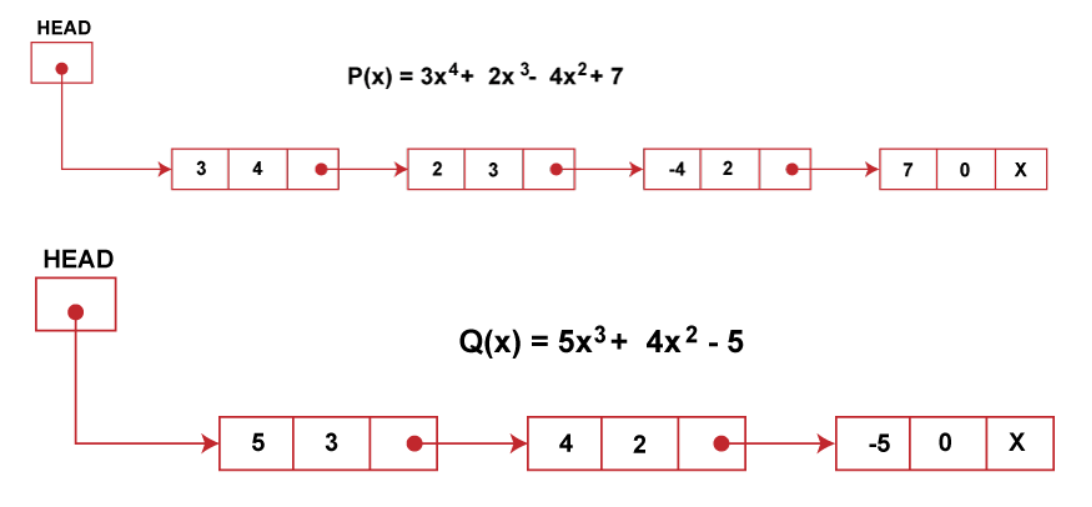
Step 10: END

**Linked List representation of Polynomials**

Here each node represents a term of the polynomials, containing three parts: -

1. **First part** - Value / coefficient of the term
2. **Second part** - Value of exponent
3. **Third part** - Link that points to the next term of the polynomial





**Array**

