Unit-3: LINKED LISTS

Introduction:

Linked lists were developed in 1955-1956, by Allen Newell, Cliff Shaw and Herbert A. Simon at RAND Corporation and Carnegie Mellon University as the primary data structure for their Information Processing Language (IPL).

A linked list is a linear collection of data elements in which each element points to the next element.

A linked list is a linear data structure consisting of a collection of nodes which together represent a sequence.

A **Linked List** is a **dynamic data structure** consisting of a sequence of elements, called **nodes**, where each node contains two parts: 1. **Data field** – stores actual data.

2. **Pointer field (link/next)** – stores the address of the next node.

A linked list is a sequence of nodes that contain two fields - **data** and **a link (**a **pointer to the next node)** to the next node. The last node is linked to a terminator used to indicate the end of the list more often denoted by NULL.

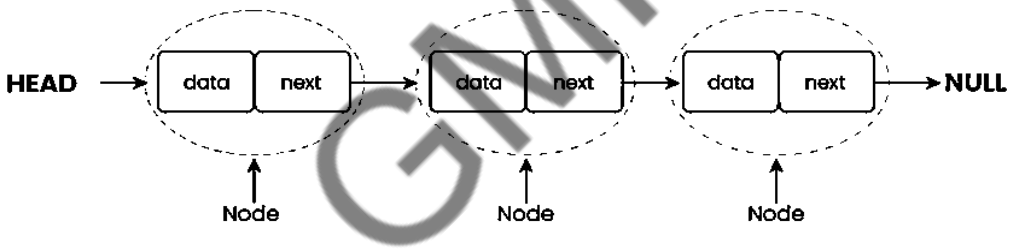

Figure 3.1: Structure of Linked list

Example : Linked list containing integer data

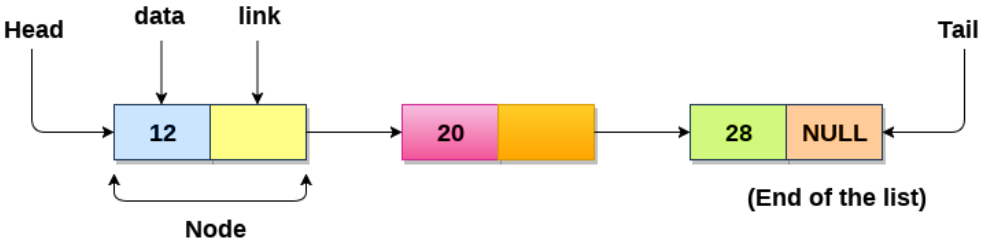

Figure 3.2: Linked list containing integer data

Linked lists do not **require contiguous memory allocation**, can **grow or shrink dynamically during program execution,** allow **efficient insertion and deletion operations**.

Linked List Key Characteristics are:

- ✓ Linear data structure (elements are arranged sequentially).
- ✓ Access to elements is sequential, not random.
- ✓ Memory is allocated dynamically using pointers.
- ✓ Each node is connected by links (addresses).

Applications of Linked list:

The applications of the Linked list are given as follows -

- With the help of a linked list, the polynomials can be represented as well as we can perform the operations on the polynomial.
- A linked list can be used to represent the sparse matrix.
- The various operations like student's details, employee's details, or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
- Using linked list, we can implement stack, queue, tree, and other various data structures.
- The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.
- A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.
- A linked list can be used in undo functionality of software's

Major differences between array and linked-list are listed below:

Representation of Linked List:

Each node of the linked list is represented as follows.

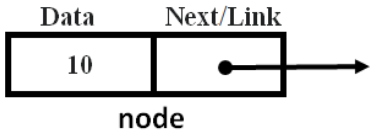

Figure 3.3: node representation

Each node consists:

- A **data** item
- An **address** of another node

Representation of a Node in C is given below. Each node is dynamically allocated using the **malloc()** function and linked to form a list. Both the data item and the next node reference are wrapped in a structure as:

```
struct node
{
  int data;
  struct node *next;
};
```

Each struct node has a data item and a pointer to another struct node.

For Doubly Linked Lists, an additional pointer `prev` is used:

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


3.2 Operations of Linked Lists

The basic operations in a linked list are:

1. **Creation:** Creating a singly linked list process starts with creating a new node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the dynamic memory allocation malloc() function.
2. **Insertion:** Adding 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
3. **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.

4. **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.

3.3 Types of linked lists

There are various types of linked list:

1. **Singly Linked Lists:** Each node has a data field and a pointer to the next node.

   Head → [10|*] → [20|*] → [30|NULL]

   - Traversal is one-way (forward) only.
   - The last node's `next` pointer is NULL.
2. **Doubly Linked Lists:** Each node contains two pointers — one pointing to the next node and another to the previous node.

   NULL ← [*|10|*] ↔ [*|20|*] ↔ [*|30|*] → NULL

   - Allows **bidirectional traversal**.
   - Makes deletion and insertion easier but requires extra memory.
3. **Circular Linked Lists:** The last node points to the first node, making a circular chain.

   Head → [10|*] → [20|*] → [30|*] ↺ (back to Head)

   - No NULL pointer exists.
   - Can start traversal from any node.
   - Useful in applications requiring circular traversal (like CPU scheduling).
4. **Circular Doubly Linked List:** The last node points to the first node, making a circular chain. Each node contains two pointers — one pointing to the next node and another to the previous node.

Basic Operations on Linked Lists

A. Traversal: Visiting each node in sequence to access data.

Algorithm:

Step 1: Set ptr = head

Step 2: While ptr != NULL

Display ptr->data

ptr = ptr->next

Time Complexity:O(n)

B. Insertion: Adding a new node to the linked list.

(i) At Beginning

Create a new node.

Make its `next` pointer point to current `head`.

Update `head` to new node.

(ii) At End

Traverse to the last node.

Set last node's `next` to new node.

Set new node's `next` = NULL.

(iii) At Given Position

Traverse to the node **before** the desired position.

Adjust pointers to insert new node.

Example Illustration:

Before: [10|*] → [20|*] → [30|NULL]

Insert 15 after 10:

After: [10|*] → [15|*] → [20|*] → [30|NULL]

C. Deletion: Removing a node from the list.

(i) From Beginning

Store address of head node.

Move head to next node.

Free deleted node's memory.

(ii) From End

Traverse till second-last node.

Set its `next` to NULL.

Free last node.

(iii) Specific Node

Find the node preceding the target node.

Adjust its `next` pointer.

Free the target node.

D. Searching: Traverse each node until data matches the given key.

Algorithm:

```
ptr = head
while(ptr != NULL)
{
   if(ptr->data == key)
      return FOUND
   ptr = ptr->next
}
return NOT FOUND
```


E. Reversal: Changing direction of all links.

Before: [10|*] → [20|*] → [30|NULL]

After: [30|*] → [20|*] → [10|NULL]

Real-Life Applications

Type	Example Applications
Singly Linked List	Polynomial manipulation, stack implementation
Circular Linked List	CPU scheduling, multiplayer games, buffering
Doubly Linked List	Navigation menus, undo-redo, memory allocation

Applications of Linear Linked List-Polynomial Representation

A polynomial like $P(x) = 5x^3 + 4x^2 + 2x + 1$ can be represented using linked lists where each node stores, Coefficient, Exponent, Pointer to next term.

Structure:

```
struct PolyNode {
    int coeff;
    int exp;
    struct PolyNode *next;
};
```

Representation:

Head → [5,3] → [4,2] → [2,1] → [1,0] → NULL

A. Polynomial Addition

Polynomial Addition Using Linked List

Algorithm:

1. Create linked lists for P(x) and Q(x).
2. Compare exponents:
   If equal → Add coefficients.
   If greater → Copy higher exponent term.
3. Insert result term into new list.
4. Continue until both lists are finished.

Example:

$P(x) = 5x^3 + 4x^2 + 2$

$Q(x) = 3x^3 + x + 6$

Result:

$R(x) = (5+3)x^3 + 4x^2 + (1)x + (2+6)$

$R(x) = 8x^3 + 4x^2 + x + 8$

A polynomial such as $P(x) = 5x^3 + 4x^2 + 2x + 1$ can be represented as a **linked list** where each node contains 3 fields **coeff** (Coefficient of the term), **exp** (Exponent (power) of x), **next** (Pointer to the next term)

Node for polynomial expression can be represented as :

```
struct PolyNode {
   int coeff;
   int exp;
   struct PolyNode *next;
};
```


Representation of P(x): |5,3| → |4,2| → |2,1| → |1,0| → NULL

Example: $P_1(x) = 5x^3 + 4x^2 + 2$

$P_2(x) = 3x^3 + x + 6$

Addition Result: $R(x) = 8x^3 + 4x^2 + x + 8$

Algorithm Polynomial Addition Using Linked List

Algorithm **POLY_ADD**(P1, P2)

//Input: Two linked lists `P1` and `P2` representing two polynomials.

//Output: A linked list `P3` representing the sum of the two polynomials.

Steps:

1. Initialize pointers

p1 = head of first polynomial (P1)

p2 = head of second polynomial (P2)

p3 = NULL (resultant polynomial)

2. Repeat while both p1 and p2 are not NULL

    1. If `p1->exp == p2->exp`

        Create a new node.

        Set `newNode->coeff = p1->coeff + p2->coeff`

        Set `newNode->exp = p1->exp`

        Insert `newNode` at the end of `p3`.

        Move both pointers:

            p1 = p1->next

            p2 = p2->next

    2. Else if `p1->exp > p2->exp`

        Copy term from `p1` to `p3`.

        Move `p1 = p1->next`.

    3. Else if `p1->exp < p2->exp`

        Copy term from `p2` to `p3`.

        Move `p2 = p2->next`.

3. If terms remain in P1

    Copy all remaining terms of `p1` to `p3`.

4. If terms remain in P2

    Copy all remaining terms of `p2` to `p3`.

5. Return `p3` (head of resultant list).

Example: P1: $5x^3 + 4x^2 + 2$ P2: $3x^3 + x + 6$

P1: [5,3] → [4,2] → [2,0]

P2: [3,3] → [1,1] → [6,0]

R: [8,3] → [4,2] → [1,1] → [8,0] → NULL

Result Polynomial: $R(x) = 8x^3 + 4x^2 + x + 8$

Step	Comparison	Action	Resultant Term
1	3 = 3	Add (5+3)x^3	$8x^3$
2	2 > 1	Copy $4x^2$	$4x^2$
3	1 < 2	Copy x	x
4	P1: constant term (2), P2: constant (6)	Add (2+6)	8

B. Polynomial Subtraction

Polynomial Subtraction Using Linked List

Algorithm:

1. Traverse both lists.

2. Subtract coefficients when powers are equal.

3. Copy remaining terms as they are.

Example:

$P(x) = 5x^3 + 4x^2 + 2$

$Q(x) = 3x^3 + x + 6$

Result:

$R(x) = (5-3)x^3 + 4x^2 - x - 4$

$R(x) = 2x^3 + 4x^2 - x - 4$

A polynomial such as $P(x) = 5x^3 + 4x^2 + 2x + 1$ can be represented as a linked list where each node contains 3 fields **coeff** (Coefficient of the term), **exp** (Exponent (power) of x), **next** (Pointer to the next term). Each term (node) represents one power of x, and the nodes are linked in descending order of exponents.

Node for polynomial expression can be represented as :

```
struct PolyNode {
    int coeff;
```

```
    int exp;
    struct PolyNode *next;
};
```

Structure of Node (C-style):

```
struct PolyNode {
    int coeff;
    int exp;
    struct PolyNode *next;
};
```

Example: $P(x) = 5x^3 + 4x^2 + 2$

Linked List: $[5,3] \rightarrow [4,2] \rightarrow [2,0] \rightarrow NULL$

$R(x) = P1(x) - P2(x)$ where both P1 and P2 are represented as linked lists.

Algorithm **POLY_SUBTRACT**(P1, P2)

//Input: Two linked lists representing polynomials `P1` and `P2`, where each node contains a coefficient and an exponent.

//Output:A new linked list `P3` representing the **difference** of the two polynomials.

1. Initialize pointers

    p1 = head of first polynomial (P1)

    p2 = head of second polynomial (P2)

    p3 = NULL // Resultant polynomial

2. While both lists are not NULL

    1. If `p1->exp == p2->exp`

        Create a new node.

        Set `newNode->coeff = p1->coeff - p2->coeff`

        Set `newNode->exp = p1->exp`

Insert `newNode` at end of `p3`.

Move both pointers:

p1 = p1->next

p2 = p2->next

2. Else if `p1->exp > p2->exp`

Copy term from `p1` to result `p3`.

Move `p1 = p1->next`.

3. Else (`p1->exp < p2->exp`)

Copy term from `p2`, but **negate its coefficient** (since subtracting).

Move `p2 = p2->next`.

3. If any terms remain in P1

Copy them directly to `p3`.

4. If any terms remain in P2

Copy them to `p3`, but **negate each coefficient**.

5. Return head of resultant polynomial (`p3`).

Example

$P1(x) = 5x^3 + 4x^2 + 2$

$P2(x) = 3x^3 + x + 6$

Step	Comparison	Action	Resultant Term
1	3 = 3	$(5-3)x^3$	$2x^3$
2	2 > 1	Copy $4x^2$	$4x^2$
3	1 < 2	Subtract $(0-1)x^1$	$-x$
4	Remaining terms: 2 − 6	$(2-6)x^0$	-4

Result Polynomial: $R(x) = 2x^3 + 4x^2 - x - 4$

Linked List Representation: $|2,3| \rightarrow |4,2| \rightarrow |-1,1| \rightarrow |-4,0| \rightarrow$ NULL

P1: [5,3] → [4,2] → [2,0]

P2: [3,3] → [1,1] → [6,0]

R: [2,3] → [4,2] → [−1,1] → [−4,0]

Notes for Implementation:

Always store terms in **descending order** of exponents.

When coefficients become zero after subtraction, the term may be **skipped** to simplify the result.

Use a helper function `InsertTerm()` to append nodes to the result list.