MODULE-3

LINKED LISTS

Disadvantages of Arrays

Although arrays are one of the simplest and most efficient data structures for storing elements of the same data type, they have several limitations. These limitations make arrays less suitable for applications where the amount of data changes frequently or where insertions and deletions occur often. The major drawbacks of arrays are explained below.

1. Fixed Size: The size of an array is fixed when it is created. It cannot be increased or decreased later.

Example: **int arr[5]**

This array can store only **5 elements**. If you need to store a 6th element, you must create a new array.

2. Memory Waste: If you allocate a large array but use only a few elements, the remaining memory is wasted.

Example: int arr[100]

If you store only **10 elements**, the other **90 locations remain unused**, wasting memory.

3. Costly Insertion and Deletion: Inserting or deleting an element in the middle of an array requires shifting the remaining elements, which takes extra time.

Example: int arr[5] = {10, 20, 30, 40, 50};

If you insert **25** after **20**, the elements **30, 40, and 50** must all be shifted one position to the right.

Result: Before: 10 20 30 40 50

After : 10 20 25 30 40 50

4. Overflow Risk: Trying to store more elements than the array's capacity can cause errors or undefined behaviour.

Example: int arr[3] = {10, 20, 30};

arr[3] = 40; // Invalid

The array has space for only **3 elements (indices 0, 1, 2)**. Storing a fourth element is invalid.

5. Costly Resizing: If the array becomes full, you cannot simply increase its size. You must create a new larger array and copy all the existing elements.

Example: int arr1[3] = {10, 20, 30};

int arr2[5];

To add more elements, copy all values from arr1 to arr2, then continue inserting new elements.

LINKED LISTS

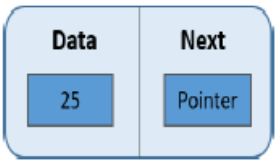
Linked List is the solution to the major drawbacks of arrays. It provides dynamic memory allocation, efficient insertion and deletion, reduced memory wastage, and flexible storage, making it suitable for applications where the amount of data changes frequently. Generally, arrays allocate the memory for all its elements in one block whereas linked lists use an entirely different strategy. Linked lists allocate memory for each element separately and only when necessary.

DEF: A linked list is a dynamic linear data structure used to store a collection of elements called nodes. Unlike an array, the elements in a linked list are not stored in contiguous (continuous) memory locations. Instead, each node is stored anywhere in memory and is connected to the next node through a pointer (link).

— In a linked list, every node contains a pointer to another node which is of same type, called as "Self-referential data type", and structure is self-referential structure.

— Linked lists are used in many real-world applications to store and manage sequential data dynamically. Each item (such as a patient, song, passenger, or bus stop etc….) is represented as a node that contains data and a link to the next item, making insertion and deletion operations efficient without shifting other elements.

Structure of a Node



Each node contains two parts:

Data – Stores the actual value or information.

Next Pointer (Link) – Stores the memory address of the next node.

Types of Linked Lists

A linked list is classified into different types based on how the nodes are connected and traversed.

1. Singly Linked List (SLL): Navigation is forward only
2. Doubly Linked List (DLL): Forward and Backward navigation is possible

3. Circular Linked List (CLL): Last element is linked to the first element

Singly Linked List (SLL)

A Singly Linked List is the simplest type of linked list. Each node contains two fields: a data field and a next pointer. The next pointer stores the address of the next node in the list. Traversal is possible only in the forward direction, starting from the head node and ending at the last node, whose next pointer contains NULL.

The graphical representation of a node in a single linked list is as follows...



The first node is called the head, which acts as the entry point to the linked list. The last node points to NULL, indicating the end of the list.

Structure of a node in C:

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

Int data refers to the data part of a node and Struct node *next is a pointer refers the next node in the list.

Main Operations of Single Linked List

1. Creation
2. Insertion (Beginning, End, Specific Position)
3. Deletion (Beginning, End, Specific Position)
4. Traversal (Display)
5. Searching

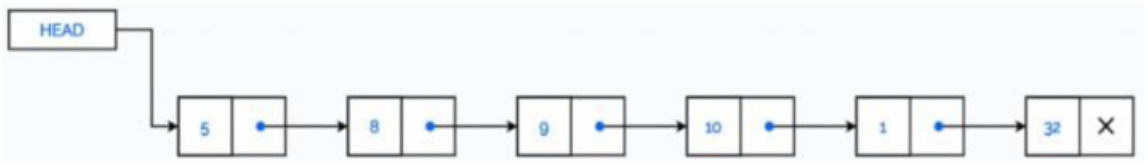
❖ Creation Operation

The creation operation in a singly linked list is the process of creating new nodes dynamically and linking them together to form a linked list. Each new node contains a data field and a pointer (next) that stores the address of the next node. The first node is pointed to by the head pointer.

Algorithm:

1. Start.
2. Create a new node using dynamic memory allocation.
3. Enter the data.
4. Store the data in the new node.
5. Set newnode->next = NULL.
6. If head == NULL, then
    - Set head = newnode.
7. Otherwise,
    - Set temp = head.
    - Move temp to the last node (while temp->next != NULL).
    - Set temp->next = newnode.
8. Repeat Steps 2–7 until all nodes are created.
9. Stop.

Graphical Representation of Singly Linked List Creation



<u>Real-World Example</u>:

Application: Student Registration System: In a Student Registration System, whenever a new student enrols in a college, the student's record is created and added to the end of the registration list. Similarly, in a singly linked list, each new node is created dynamically and linked to the end of the existing list.

Initially: NULL

After Registering Student 1: Aisha → NULL

After Registering Student 2: Aisha → Rahul → NULL

After Registering Student 3: Aisha → Rahul → Priya → NULL

Each newly registered student is added as a new node at the end of the linked list.

Time Complexity

- **Best Case: O(1)** (when creating the first node or when a tail pointer is maintained)
- **Worst Case: O(n)** (when inserting each new node at the end without maintaining a tail pointer)

Reason: During the creation of a linked list, each new node is inserted at the end. If the program does **not** maintain a **tail pointer**, it must traverse the entire linked list to reach the last node before linking the new node. Therefore, each insertion at the end takes **O(n)** time in the worst case.

Note: If a **tail pointer** is maintained, each new node can be added directly at the end without traversal, making each insertion **O(1)**.

❖ Insertion Operations

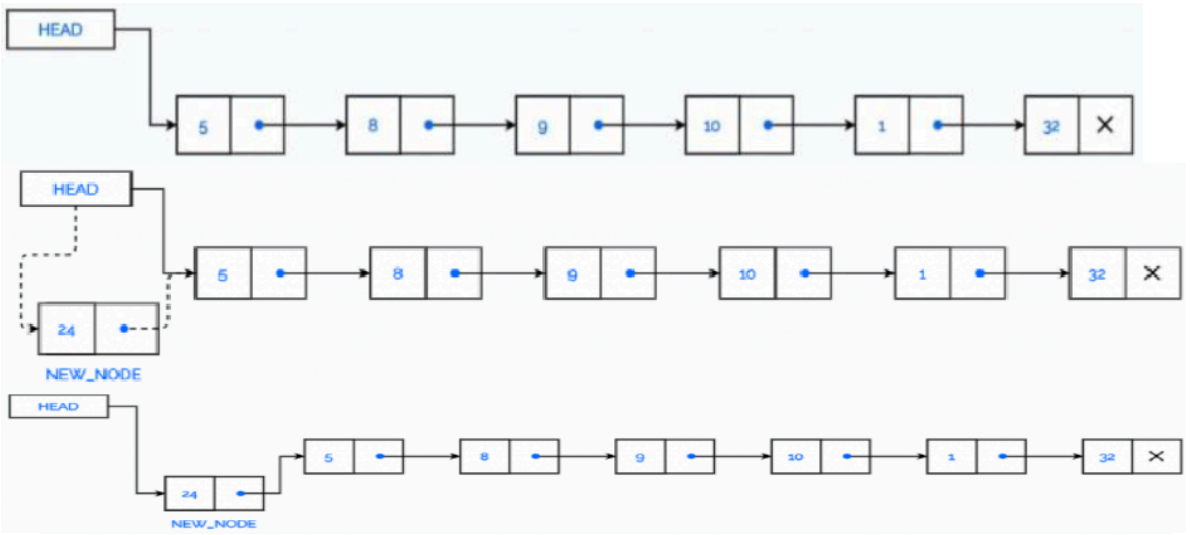
Insertion in a Singly Linked List is the operation of adding a new node into the list. The new node can be inserted at the beginning, end, or any specified position by updating the next pointers appropriately while maintaining the link between nodes.

Insertion at Beginning: Insertion at the beginning of a singly linked list is the process of creating a new node, storing the data, making the new node point to the current first node, and updating the head pointer to the new node. This operation requires O(1) time because it does not involve traversing the list.

Algorithm: InsertAtBeginning ()

1. Start.
2. Create a new node.
3. Enter the data into the new node.
4. Set newnode->next = head.
5. Set head = newnode.
6. Stop.

Graphical representation of Insertion at Beginning



Real-World Example: Emergency Patient Queue (Insertion at the Beginning)

Application: Hospital Emergency Management System

In a hospital emergency management system, patients are normally treated in the order they arrive. However, if a critically ill patient arrives, they are given the highest priority and moved to the front of the queue. This is similar to insertion at the beginning in a singly linked list

Emergency Patient → Patient A → Patient B → Patient C → NULL

Time Complexity

- **Best Case: O(1)**
- **Worst Case: O(1)**

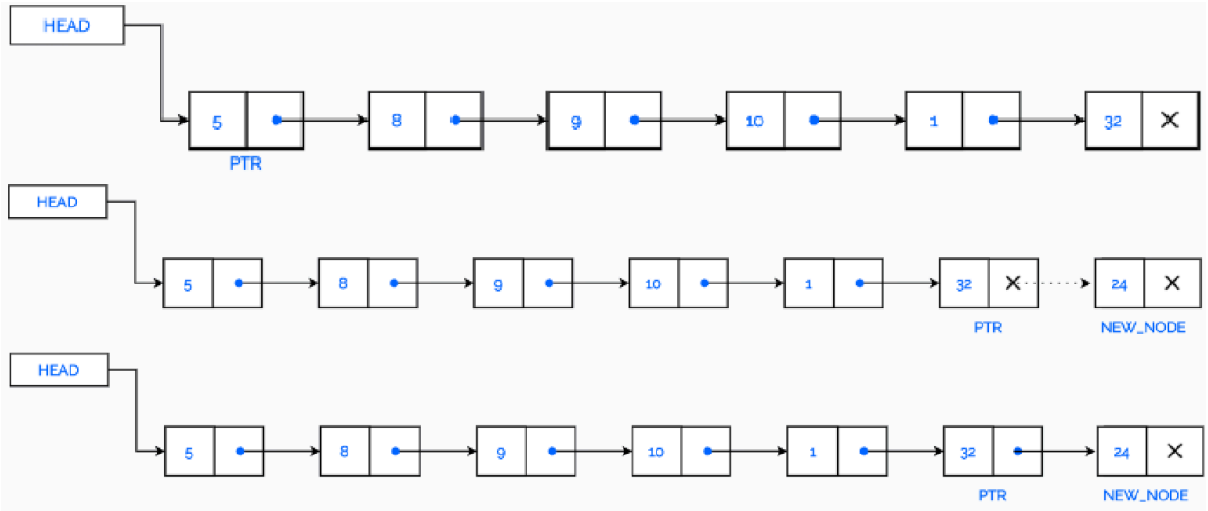
Insertion at End: Insertion at the end of a singly linked list is the process of creating a new node, storing the data, setting its next pointer to NULL, traversing the list to the last node, and updating the last node's next pointer to the new node. If the list is empty, the new node becomes the head of the list. This operation has a time complexity of O(n) because it requires traversal to reach the last node.

Algorithm: InsertAtEnd ()

1. Start.
2. Create a new node.
3. Enter the data into the new node.
4. Set newnode->next = NULL.

5. If head == NULL, then
    - Set head = newnode.
6. Otherwise,
    - Set temp = head.
    - Traverse the list until temp->next == NULL.
    - Set temp->next = newnode.
7. Stop.

Graphical representation of Insertion at End



<u>Real-World Example</u>: Train Waiting List (Insertion at End)

Application: Railway Reservation System: In a railway reservation system, passengers are added to the waiting list in the order they book their tickets. Every new passenger is placed at the end of the waiting list. This is similar to **insertion at the end** in a singly linked list.

Passenger A → Passenger B → Passenger C → **New Passenger** → NULL

Time Complexity

- **Best Case: O(1)** (when the list is empty)
- **Worst Case: O(n)** (when the list contains nodes and traversal to the last node is required)

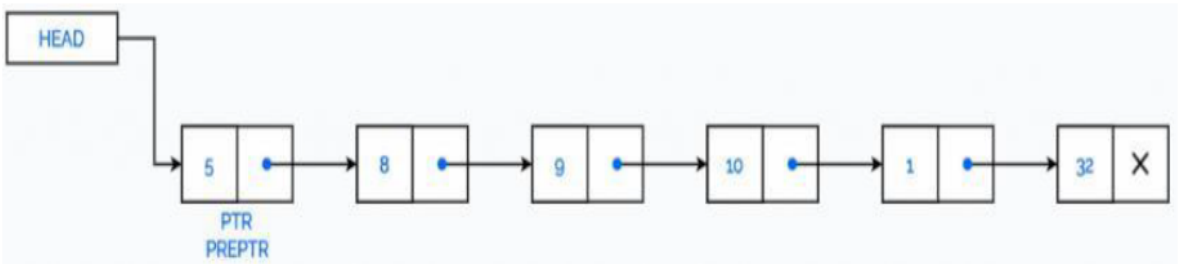
Insertion at a specific position: Insertion at a specific position in a singly linked list is the process of inserting a new node at a specified location by first traversing to the node preceding the desired position.

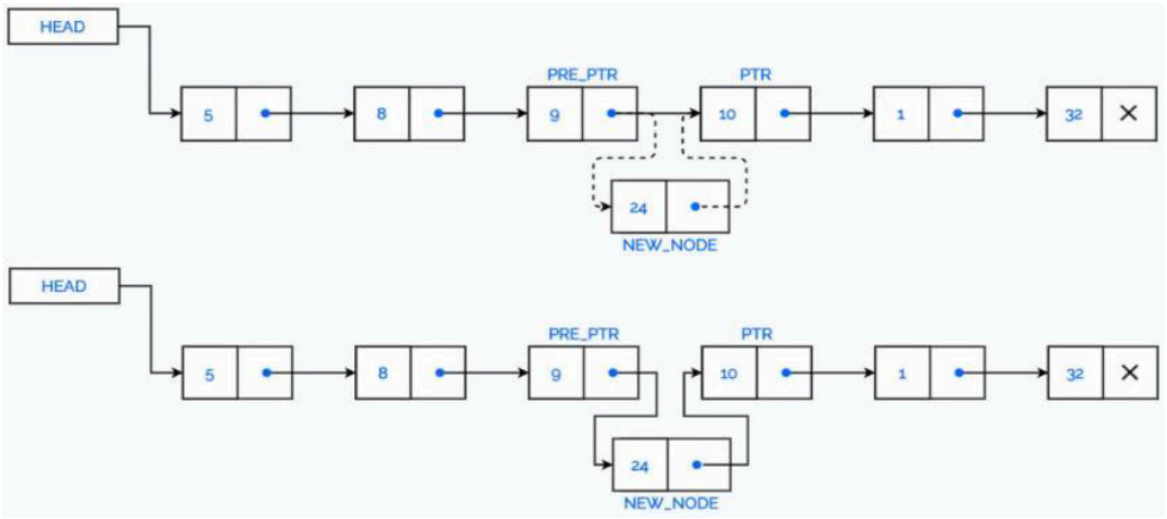
The new node is linked by setting newnode->next = temp->next and then updating temp->next = newnode. This preserves the existing links and inserts the new node at the required position. The time complexity is **O(n)** because traversal may be required to reach the specified position.

Algorithm: InsertAtPosition(pos, data)

1. Start.
2. Create a new node.
3. Enter the data into the new node.
4. If pos == 1, then
    - Set newnode->next = head.
    - Set head = newnode.
5. Otherwise,
    - Set temp = head.
    - Move temp to the node just before the required position using a loop.
    - Set newnode->next = temp->next.
    - Set temp->next = newnode.
6. Stop.

Graphical representation of Insertion at Specific Position





Real-World Example: Bus Route (InsertAtPosition)

Application: Public Transportation Management System:In a public transportation system, if a new bus stop needs to be introduced between two existing stops, it is inserted at the required position without changing the order of the remaining stops. This is similar to **insertion at a specific position** in a singly linked list.

Stop A → Stop B → Stop C → Stop D → **New Stop** →Stop E → NULL

Time Complexity

- **Best Case: O(1) (when inserting at the beginning)**
- **Worst Case: O(n) (when traversing to the required position)**

❖ Deletion operations

Deletion is the process of removing an existing node from a singly linked list. After deleting a node, the links are updated so that the remaining nodes remain connected. The memory occupied by the deleted node is then released.

A node can be deleted in three ways:

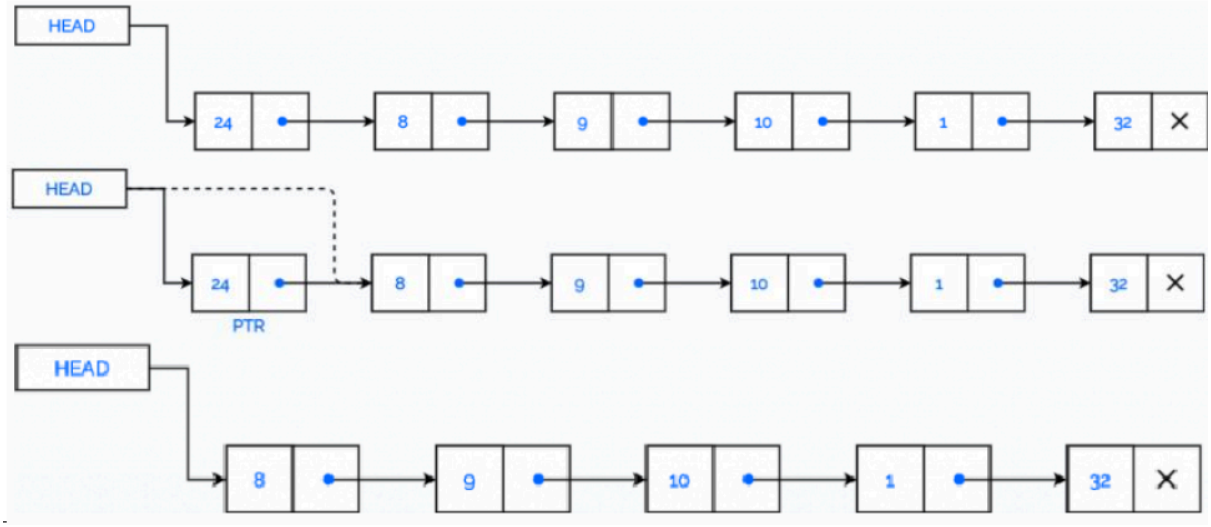
1. **Deletion at the Beginning**
2. **Deletion at the End**
3. **Deletion at the Specific Position**

Deletion at the Beginning: Deletion at the beginning is the operation of removing the **first node** from a singly linked list. After deleting the first node, the **head** pointer is updated to point to the second node. The memory occupied by the deleted node is then released.

Algorithm: DeleteAtBeginning ()

1. **Start.**
2. Check whether the linked list is empty.
3. If head == NULL, display **"List is Empty"** and **Stop**.
4. Store the address of the first node in a temporary pointer.
    - temp = head
5. Move the **head** pointer to the next node.
    - head = head->next
6. Delete (free) the first node.
    - free(temp)
7. Display **"Node Deleted Successfully"** (optional).
8. **Stop.**

Graphical representation of Deletion at the Beginning



Real-World Example: Hospital Patient Queue (Deletion at the Beginning)

Application: Hospital Emergency Management System: In a hospital emergency management system, patients are treated one by one. After the first patient receives treatment, their record is removed from the front of the queue. This is similar to **deletion at the beginning** in a singly linked list.

Patient A → Patient B → Patient C → NULL

After deletion: Patient B → Patient C → NULL

Time Complexity

- **Best Case: O(1)**
- **Worst Case: O(1)**

Reason: The deletion operation updates only the **head** pointer and deletes the first node without traversing the list. Therefore, it always takes constant time.

Deletion at the End: Deletion at the end is the operation of removing the last node from a singly linked list. To delete the last node, the list is traversed until the second-last node is reached. The second-last node's next pointer is then updated to NULL, and the memory occupied by the last node is released.

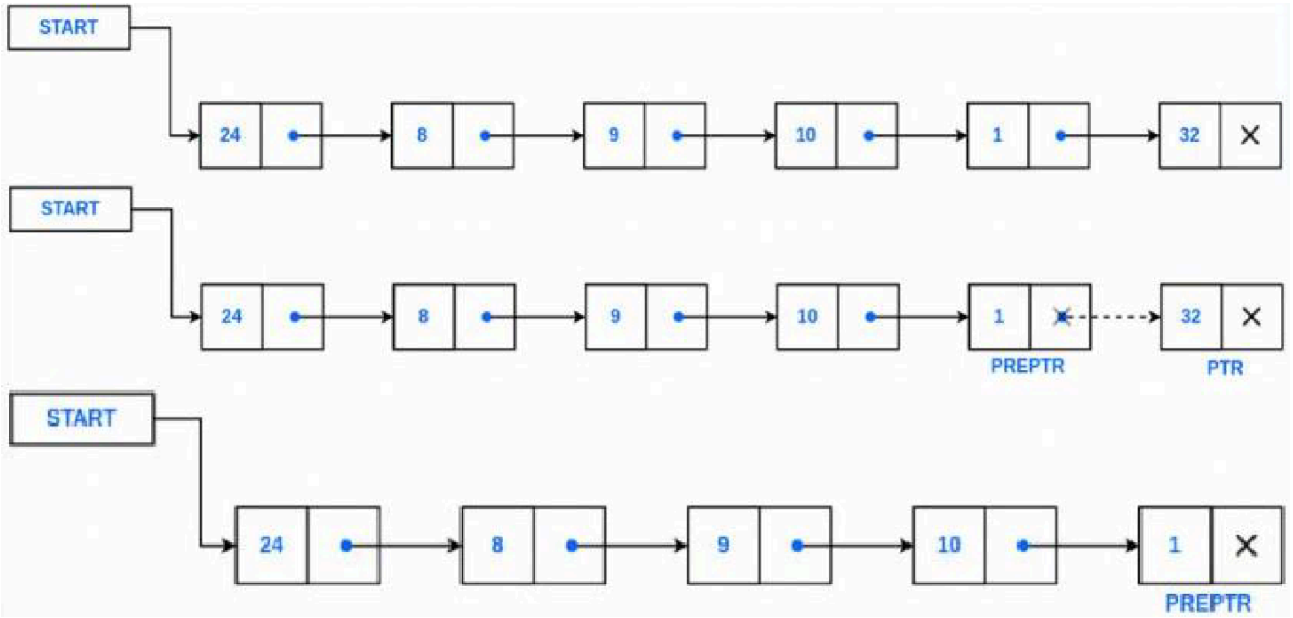
Algorithm: Deletion AtEnd ()

1. **Start.**
2. Check whether the linked list is empty.
3. If head == NULL, display **"List is Empty"** and Stop.
4. If the list contains only one node (head->next == NULL):

- Delete the node.
- Set head = NULL.
- Stop.

5. Create a temporary pointer and move it to the head.
   - temp = head
6. Traverse the list until the second-last node is reached.
   - while(temp->next->next != NULL)
   - temp = temp->next
7. Store the last node in another pointer.
   - last = temp->next
8. Remove the last node by updating the next pointer.
   - temp->next = NULL
9. Delete (free) the last node.
   - free(last)
10. Display **"Node Deleted Successfully"** (optional).
11. **Stop.**

Graphical representation of Deletion at the End



Real-World Example: Shopping Cart (Deletion at the End)

Application: E-Commerce Application (Amazon, Flipkart): In an online shopping application, when a customer removes the **last item** added to the shopping cart, that item is deleted from the end of the list. This is similar to **deletion at the end** in a singly linked list.

Laptop → Mouse → Headphones → NULL

After Deletion: Laptop → Mouse → NULL

Time Complexity

- **Best Case: O(n)**
- **Worst Case: O(n)**

Reason: The list must be traversed to reach the second-last node before deleting the last node.

Deletion at a Specific Position

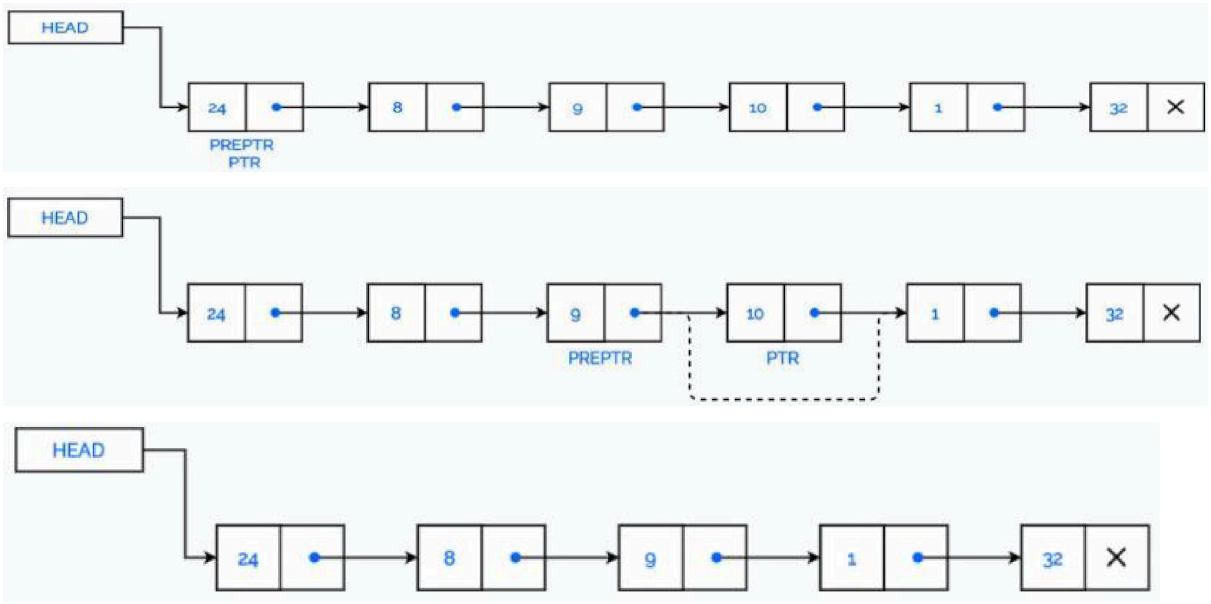
Deletion at a specific position in a singly linked list is the process of removing a node from a desired position by traversing to the node immediately before it, updating its next pointer to bypass the node being deleted, and freeing the memory occupied by the deleted node. This operation has a time complexity of **O(n)** because traversal is required to reach the specified position.

Algorithm: DeleteAtPosition(pos)

1. **Start.**
2. Check whether the linked list is empty.
3. If head == NULL, display **"List is Empty"** and stop.
4. If pos == 1, then:
    - Set temp = head.
    - Set head = head->next.
    - Free temp.
    - Stop.
5. Set temp = head.
6. Traverse the list until the node before the required position.
7. Set del = temp->next.
8. Set temp->next = del->next.
9. Free del.
10. Display **"Node Deleted Successfully"** (optional).

11. **Stop.**

Graphical representation of Deletion at a Specific Position



Real-World Example: Removing a Student from an Attendance List (Deletion at Specific Position)

Application: Student Attendance Management System: In a Student Attendance Management System, if a student withdraws from the course or is removed from the class, the student's record is deleted from the specified position in the attendance list. This is similar to deletion at a specific position in a singly linked list.

Before Deletion (Delete Student at Position 3): Aisha → Rahul → Priya → David → NULL

After Deletion: Aisha → Rahul → David → NULL

Here, the student **Priya** is removed from the third position, and the link is updated so that **Rahul** points directly to **David**.

Time Complexity

- **Best Case: O(1)** (when deleting the first node)
- **Worst Case: O(n)**

Reason: The linked list must be traversed to reach the node just before the specified position. After reaching that node, the links are updated and the target node is deleted. Therefore, the overall time complexity is **O(n)**.

❖ Searching Operation

Searching is the operation of finding whether a particular element (key) is present in a singly linked list. The search starts from the **head** node and proceeds sequentially through each node until the desired element is found or the end of the list (**NULL**) is reached.

Algorithm: SearchElement(key)

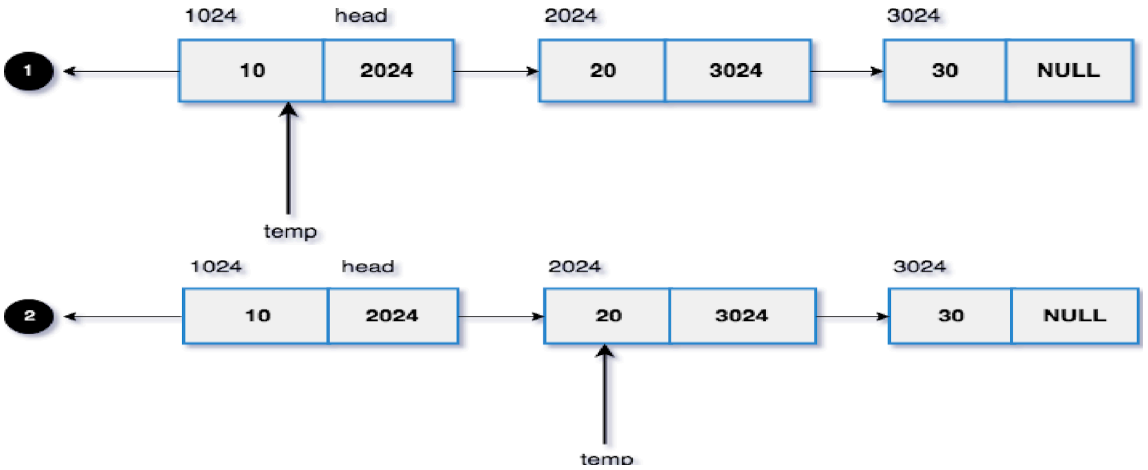
1. Start.
2. Check whether the linked list is empty.
3. If head == NULL, display **"List is Empty"** and stop.
4. Set temp = head.
5. Repeat until temp == NULL:
    - If temp->data == key
        - Display **"Element Found"**.
        - Stop.
    - Otherwise, move to the next node.
        - temp = temp->next
6. Display **"Element Not Found"**.
7. Stop.

Graphical representation of Deletion at a Specific Position

Return 1 on search found: -

Linked List: 10 20 30 NULL

🎬 Key: 20

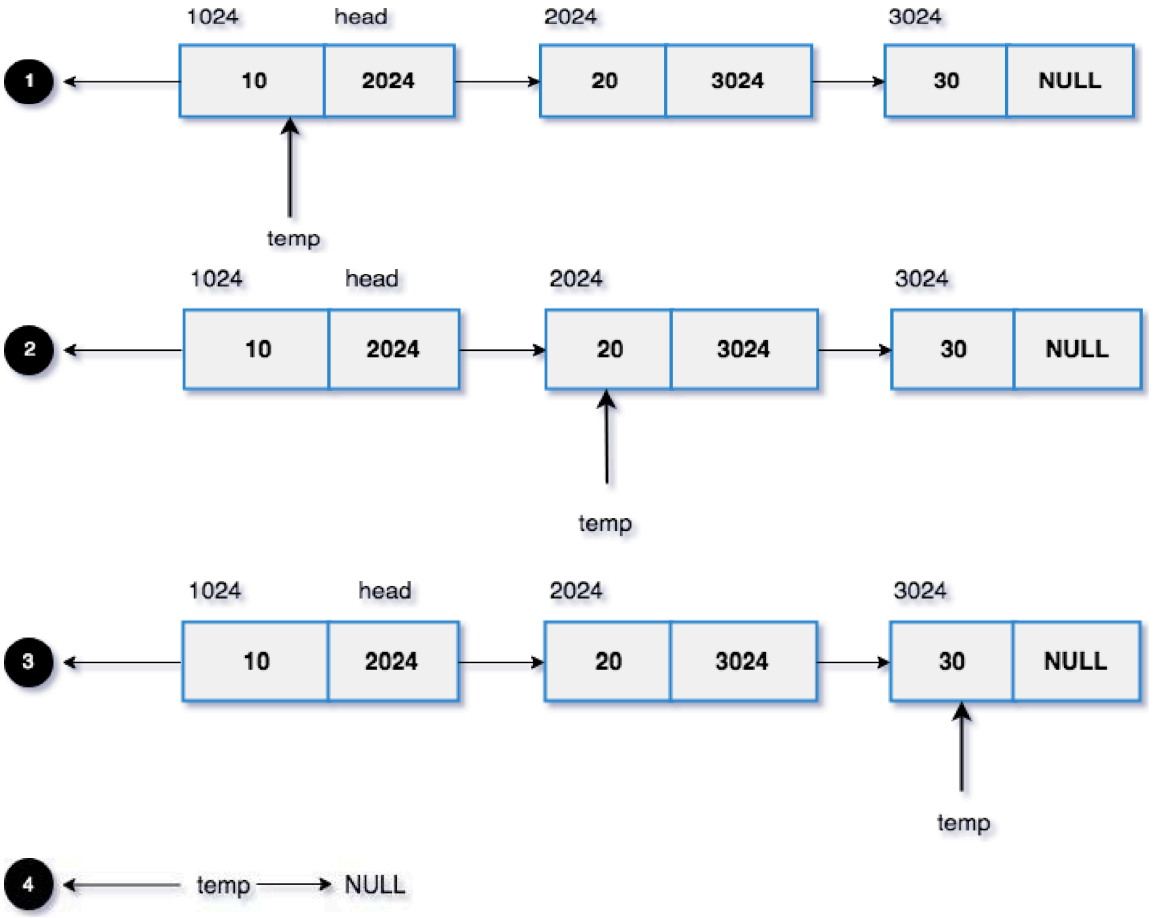


- temp holding the address of the head node. temp->data = 10. key = 20. temp->data != key, so move the temp variable to the next node.
- Now, temp->data = 20. key = 20. temp->key == key. "Seach Found".

Return -1 on search not found:-

Linked List: 10 20 30 NULL

🎬 Key: 100



- temp->data = 10. key = 100. temp->data != key. Hence move the temp variable to the next node.
- temp->data = 20. key = 100. temp->data != key. Hence move the temp variable to the next node.
- temp->data = 30. key = 100. temp->data != key. Hence move the temp variable to the next node which is NULL.
- Finally, the program execution will come out of the loop. So, it will return -1.
- "Search Not Found".

Real-World Example

Application: Contact Management Application (Mobile Phone Contacts): In a Contact Management Application, when a user searches for a contact by name or phone number, the application checks each contact one by one until the required contact is found. This is similar to the searching operation in a singly linked list.

Contact List: Aisha → Rahul → Priya → David → NULL

Suppose the user searches for **"Priya"**.

Searching Process: Aisha ✗ → Rahul ✗ → Priya ✓

The application compares each contact sequentially. When **Priya** is found, the search stops, and the contact details are displayed.

Time Complexity

- **Best Case: O(1)**
- **Worst Case: O(n)**

Reason: In a singly linked list, searching starts from the **head** node and compares each node one by one. If the required element is present in the **first node**, the search completes immediately (**O(1)**). If the element is in the **last node** or is **not present**, every node must be visited before the search ends, resulting in a time complexity of **O(n)**.

❖ Traversal Operation

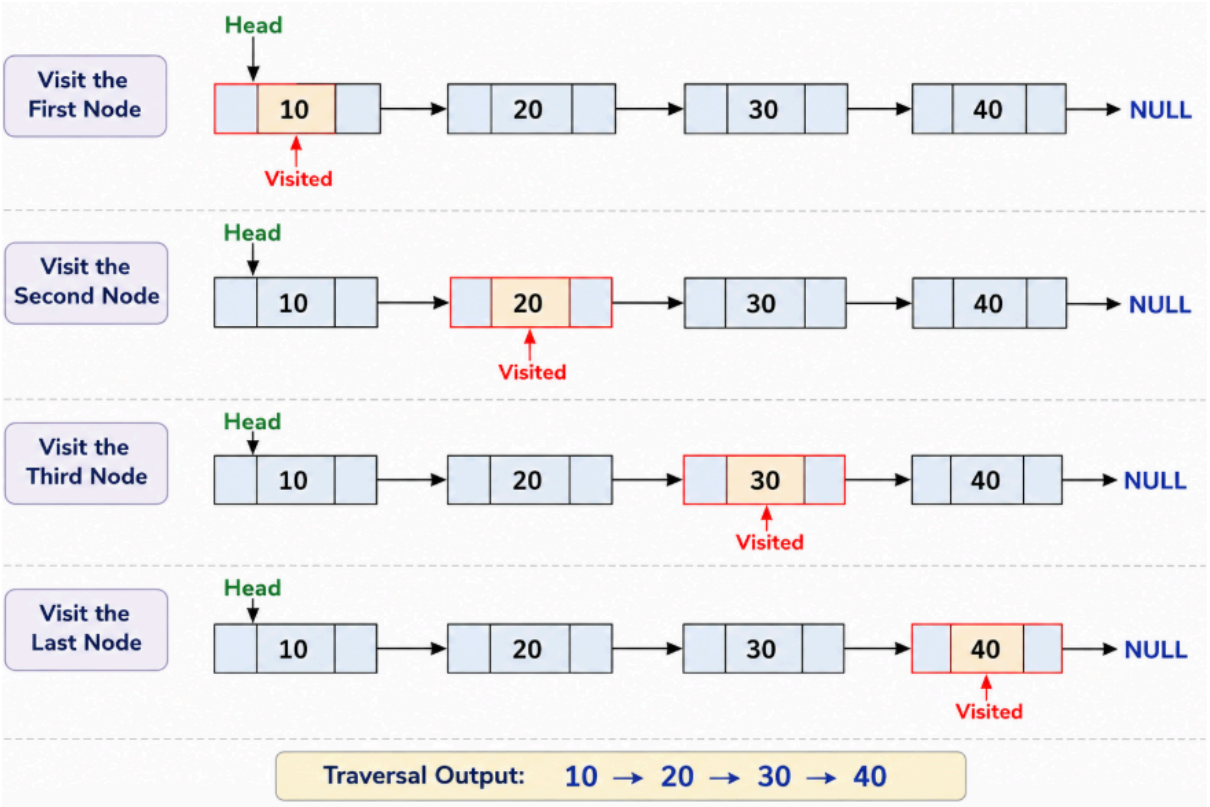
Traversal is the process of **visiting each node of a singly linked list exactly once** to access, display, or process the data stored in every node. Traversal starts from the **head** node and continues by following the next pointer until the last node (NULL) is reached.

Algorithm: TraverseSLL()

1. Start.
2. Check whether the linked list is empty.
3. If head == NULL, display **"List is Empty"** and stop.
4. Set temp = head.

5. Repeat until temp == NULL:
    - Display temp->data.
    - Set temp = temp->next.
6. Stop.

Graphical representation



<u>Real-World Example</u>: Application: Music Playlist Application

In a Music Playlist Application (such as Spotify or YouTube Music), songs are played one after another in the order they are stored in the playlist. The application visits each song sequentially until the playlist ends. This is similar to the traversal operation in a singly linked list.

Playlist: Song 1 → Song 2 → Song 3 → Song 4 → NULL

Traversal Order: Song 1 → Song 2 → Song 3 → Song 4

Each song is accessed one by one until the end of the playlist is reached.

Time Complexity

- **Best Case: O(n)**
- **Worst Case: O(n)**

Reason: During traversal, every node in the linked list must be visited exactly once to access or display its data. Therefore, the time required is proportional to the number of nodes in the list.

Advantages

- Dynamic memory allocation.
- Efficient insertion and deletion.
- No need for contiguous memory allocation.
- Better memory utilization.
- Flexible size.
- Easy implementation of dynamic data structures such as stacks and queues.
- Easy to expand or shrink during execution.

Disadvantages

- Sequential access only.
- Extra memory is required for storing pointers.
- Searching is slower (**O(n)**).
- Cannot traverse backward.
- Pointer manipulation increases implementation complexity.
- No direct/random access to elements.
- Lower cache performance compared to arrays.

Applications of Singly Linked List: -

- **Implementing stacks and queues:** Singly linked lists can be used to implement stacks and queues. In a stack, elements are added and removed from one end of the list, while in a queue, elements are added at one end and removed from the other end of the list.
- **Navigation in web browsers:** Singly linked lists can be used to store the browsing history in web browsers. Each URL visited is stored as a node in the list, with the next pointer pointing to the next URL visited.
- **Implementing symbol tables:** Symbol tables are data structures used to store key-value pairs. Singly-linked lists can be used to implement symbol tables, with each node storing a key-value pair.
- **Hash table chaining:** Hash table chaining is a technique used to handle collisions in hash tables. It involves using a singly linked list to store all the keys that hash to the same index in the table.
- **Music Player**: Songs in the music player are linked to the previous and next songs. So you can play songs either from starting or ending of the list.

- **Image Processing:** Linked lists can be used to represent images, where each pixel is represented as a node in the list.
- **Polynomial Representation:** Polynomials can be represented using linked lists, where each term in the polynomial is represented as a node in the list.

Doubly Linked List

A **Doubly Linked List (DLL)** is a type of linked list in which **each node contains two pointers**: one pointing to the **previous node** and another pointing to the **next node**. Unlike a singly linked list, a doubly linked list allows **bidirectional traversal**, enabling navigation in both the **forward** and **backward** directions.

Because each node maintains links to both its predecessor and successor, insertion and deletion operations become more efficient when the location of the node is known.

Structure of a Doubly Linked List Node

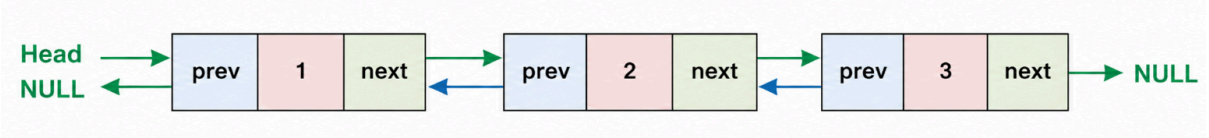
Each node in a doubly linked list consists of **three components**:

- **prev** – Stores the address of the **previous node**.
- **data** – Stores the actual data or information.
- **next** – Stores the address of the **next node**.

Node Structure

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

Representation of a Doubly Linked List



🎬 Head points to the first node.

🎬 Prev stores the address of the previous node.

🎬 Data stores the actual information.

🎬 Next stores the address of the next node.

🎬 The **first node's prev pointer is NULL** because there is no previous node.

🎬 The **last node's next pointer is NULL** because there is no next node.

🎬 The **double arrows (↔)** indicate that traversal is possible in both **forward** and **backward** directions.

Main Operations of Doubly Linked List

The primary operations of a Doubly Linked List are:

1. Creation
2. Insertion (Beginning, End, Specific Position)
3. Deletion (Beginning, End, Specific Position)
4. Traversal (Display)
5. Searching
6. Updating

❖ Creation

The **creation operation** in a **Doubly Linked List** is the process of **creating new nodes dynamically and linking them together** using both the **previous (prev)** and **next (next)** pointers. Each newly created node stores data, the address of the previous node, and the address of the next node. The **head** pointer stores the address of the first node, while the **last node's next pointer** is set to **NULL**.

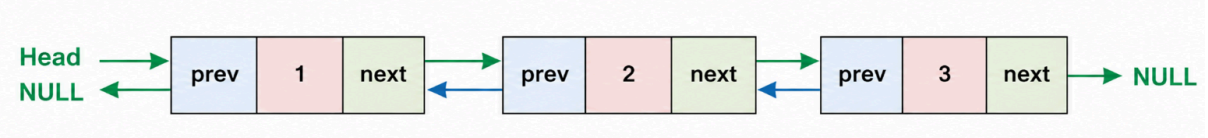
Algorithm: CreateDLL ()

1. **Start.**
2. Create a new node using dynamic memory allocation.
3. Enter the data into the new node.
4. Set newnode->next = NULL.
5. Set newnode->prev = NULL.
6. If head == NULL, then
    - Set head = newnode.
7. Otherwise,
    - Set temp = head.

- Traverse to the last node (while temp->next != NULL).
- Set temp->next = newnode.
- Set newnode->prev = temp.

8. Repeat Steps 2–7 until all nodes are created.
9. **Stop.**

Graphical Representation



Real-World Example:

Application: Music Playlist Application

In a **Music Playlist Application** (such as Spotify or Apple Music), whenever a user adds a new song to a playlist, a new node is created and added to the end of the doubly linked list. Each song maintains links to both the **previous** and **next** songs, allowing users to move to the **previous** or **next** song easily.

Initially: NULL

After Adding Song 1: NULL ← Song 1 → NULL

After Adding Song 2: NULL ← Song 1 ⇄ Song 2 → NULL

After Adding Song 3: NULL ← Song 1 ⇄ Song 2 ⇄ Song 3 → NULL

Time Complexity

- **Best Case: O(1)** (when creating the first node or when a tail pointer is maintained)
- **Worst Case: O(n)** (when inserting each new node at the end without maintaining a tail pointer)

Reason: During the creation of a doubly linked list, each new node is usually inserted at the end. If a **tail pointer** is not maintained, the list must be traversed to reach the last node before linking the new node. Therefore, the worst-case time complexity is **O(n)**.

❖ Insertion

Insertion in a Doubly Linked List is the process of adding a new node at the beginning, end, or a specified position by updating both the **previous (prev)** and **next (next)** pointers. This ensures that the list remains properly linked in both forward and backward directions.

A new node can be inserted in three ways:

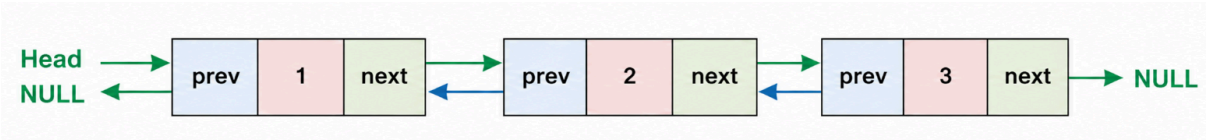
1. **Insertion at the Beginning**
2. **Insertion at the End**
3. **Insertion at a Specific Position**

Insertion at the Beginning: Insertion at the beginning is the operation of adding a new node as the **first node** of the doubly linked list. The new node becomes the **head** of the list. Both the **prev** and **next** pointers are updated accordingly.

Algorithm: InsertAtBeginning ()

1. Start.
2. Create a new node.
3. Enter the data.
4. Set newnode->prev = NULL.
5. Set newnode->next = head.
6. If head != NULL, set head->prev = newnode.
7. Set head = newnode.
8. Stop.

Graphical Representation

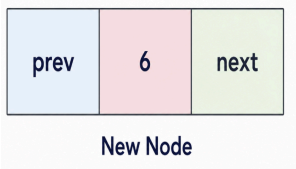


Let's add a node with value 6 at the beginning of the doubly linked list we made above.

1. Create a new node

· allocate memory for newNode

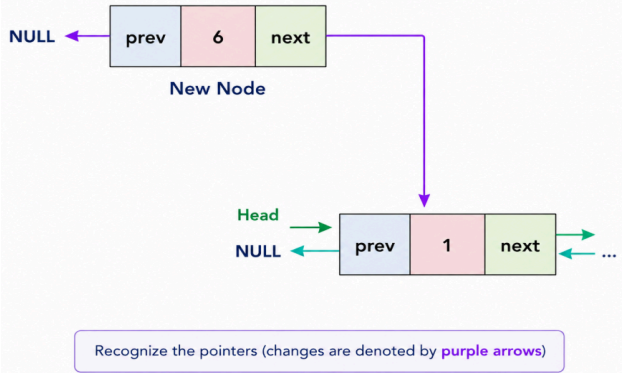
· assign the data to newNode.



2. Set prev and next pointers of new node

· point next of newNode to the first node of the doubly linked list

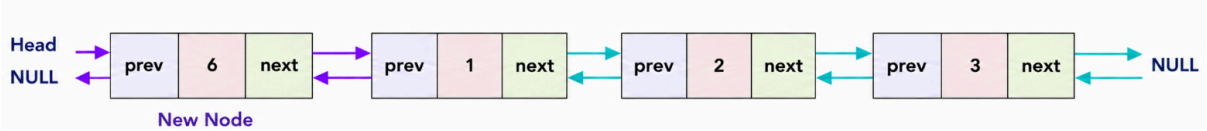
· point prev to null



3. Make new node as head node

· Point prev of the first node to newNode (now the previous head is the second node)

· Point head to newNode



<u>Real-World Example</u>:

Application: Browser History

When a new web page is opened at the beginning of the browser history, it becomes the first page while maintaining links to the next page.

Time Complexity

- **Best Case:** O(1)
- **Worst Case:** O(1)

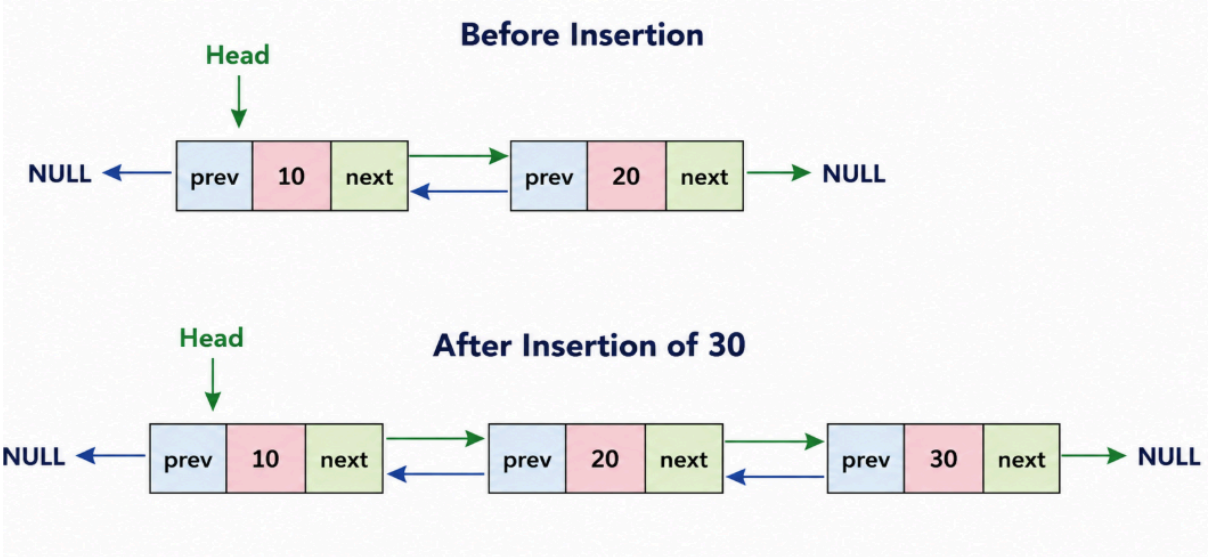
Reason: No traversal is required. Only pointer updates are performed.

<u>Insertion at the End:</u> **Insertion at the end** is the process of adding a new node as the **last node** of the doubly linked list. The new node is connected to the current last node using both **prev** and **next** pointers.

Algorithm: InsertAtEnd ()

1. Start.
2. Create a new node.
3. Enter the data.
4. Set newnode->next = NULL.
5. If head == NULL
    - Set newnode->prev = NULL.
    - Set head = newnode.
6. Otherwise
    - Traverse to the last node.
    - Set last->next = newnode.
    - Set newnode->prev = last.
7. Stop.

Graphical Representation



<u>Real-World Example</u>:

Application: Music Playlist

When a new song is added to the end of a playlist, it becomes the last song while maintaining links to both the previous and next songs.

Time Complexity

- **Best Case:** O(1) (if the list is empty or a tail pointer is maintained)
- **Worst Case:** O(n)

Reason: Without a tail pointer, the list must be traversed to reach the last node.

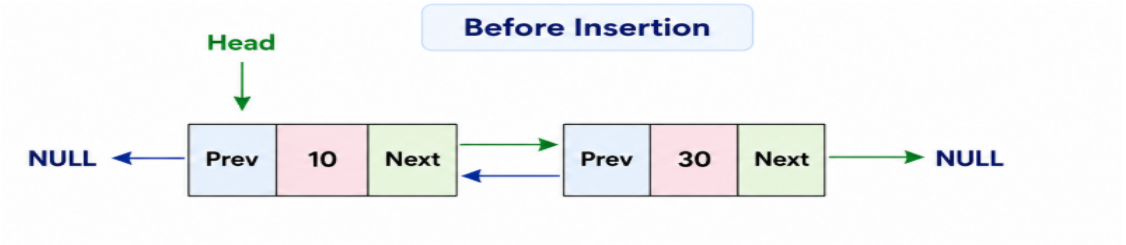
Insertion at a Specific Position

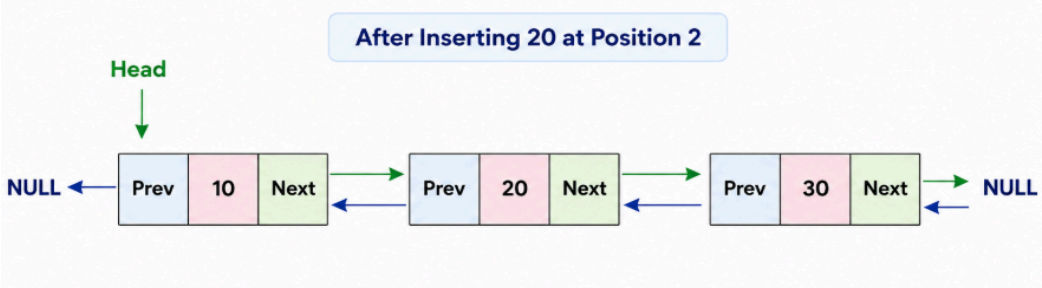
Insertion at a specific position is the process of inserting a new node at a desired position in a doubly linked list. The new node is placed between two existing nodes by updating both the **previous (prev)** and **next (next)** pointers of the adjacent nodes, thereby maintaining the bidirectional links.

Algorithm: InsertAtPosition ()

1. Start.
2. Create a new node.
3. Enter the data.
4. If the linked list is empty, set:
    - newnode->prev = NULL
    - newnode->next = NULL
    - head = newnode
5. Otherwise,
    - Traverse the list to reach the node just before the specified position.
    - Set newnode->next = temp->next.
    - Set newnode->prev = temp.
    - If temp->next != NULL
        - Set temp->next->prev = newnode.
    - Set temp->next = newnode.
6. Stop.

Graphical Representation





❖ Deletion

Deletion in a Doubly Linked List is the process of removing an existing node from the linked list by updating both the previous (prev) and next (next) pointers of the adjacent nodes. This ensures that the remaining nodes remain properly connected in both the forward and backward directions. After updating the links, the memory occupied by the deleted node is released.

A node can be deleted in three ways:

1. Deletion at the Beginning
2. Deletion at the End
3. Deletion at a Specific Position

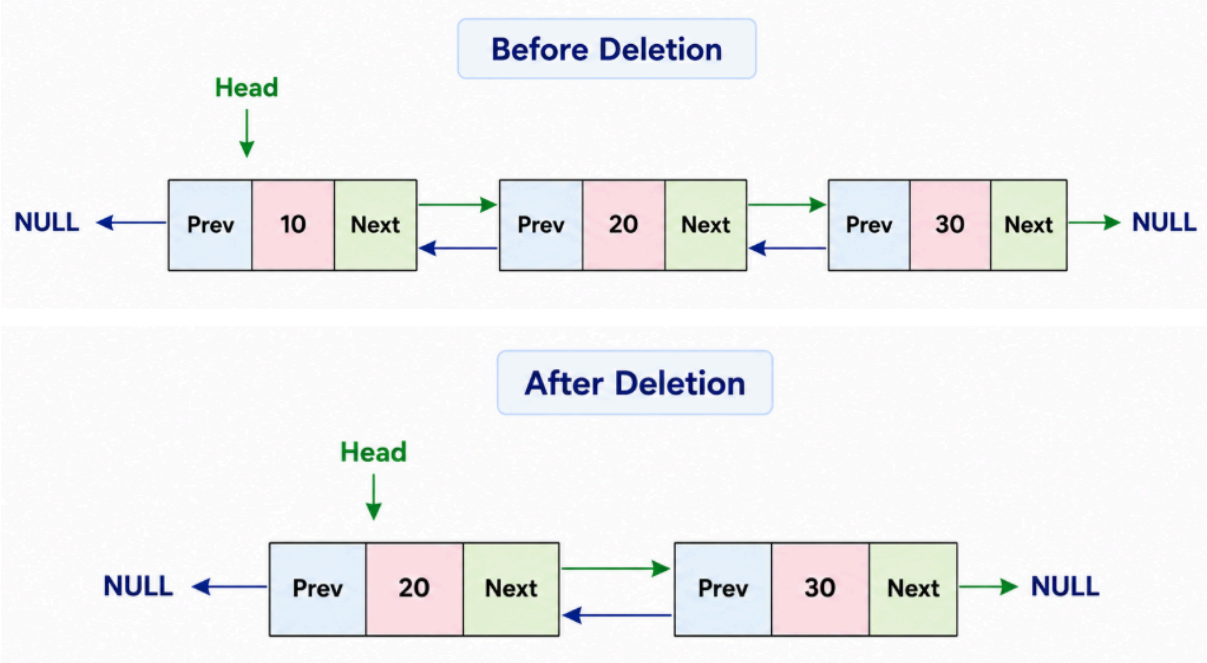
Deletion at the Beginning: Deletion at the beginning is the process of removing the first node from a doubly linked list. After deleting the first node, the head pointer is updated to point to the second node, and the previous (prev) pointer of the new first node is set to NULL. Finally, the memory occupied by the deleted node is released.

Algorithm: DeleteAtBeginning ()

1. **Start.**
2. Check whether the linked list is empty.
3. If head == NULL, display **"List is Empty"** and stop.
4. Store the first node in a temporary pointer (temp = head).
5. Set head = head->next.
6. If head != NULL, set head->prev = NULL.

7. Delete (free) the node pointed to by temp.
8. **Stop.**

Graphical Representation



Real-World Example

Application: Browser History

When the oldest web page is removed from the browser history, the second page becomes the first page in the history. The new first page has no previous page, so its prev pointer becomes NULL.

Time Complexity

- **Best Case: O(1)**
- **Worst Case: O(1)**

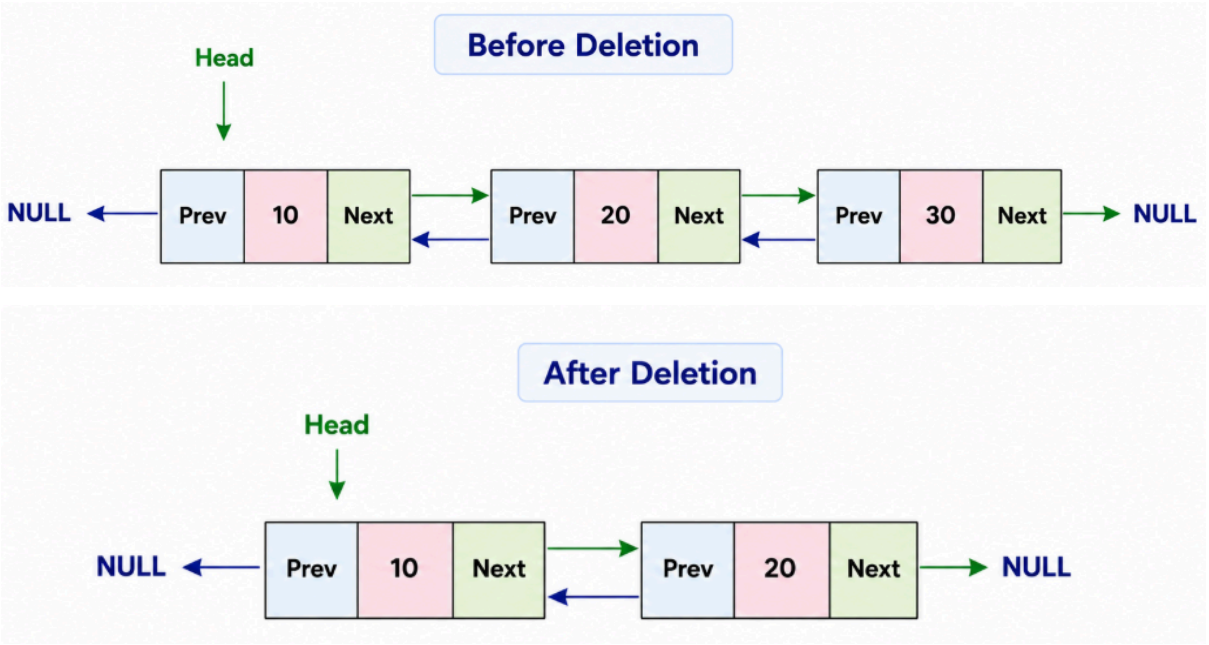
Reason: Only the first node is removed, and a few pointer updates are required. No traversal is needed.

Deletion at the End: Deletion at the end is the process of removing the last node from a doubly linked list. The next (next) pointer of the second-last node is updated to NULL, and the last node is deleted.

Algorithm: DeleteAtEnd()

1. **Start.**
2. Check whether the linked list is empty.
3. If head == NULL, display **"List is Empty"** and stop.
4. Traverse the list to reach the last node.
5. Set temp = last node.
6. Set (temp->prev)->next = NULL.
7. Delete (free) temp.
8. **Stop.**

Graphical Representation



Real-World Example

Application: Music Playlist

When the last song is removed from a music playlist, the second-last song becomes the last song. Its next pointer is updated to NULL.

Time Complexity

- **Best Case: O(1) (when a tail pointer is maintained)**
- **Worst Case: O(n)**

Reason: Without a tail pointer, the list must be traversed to reach the last node before deleting it.

Deletion at a Specific Position

Deletion at a specific position is the process of removing a node from a doubly linked list at a given position. After deleting the node, the next pointer of the previous node and the previous pointer of the next node are updated so that the remaining nodes remain properly connected.

Algorithm: DeleteAtPosition(pos)

1. Start.
2. Check whether the linked list is empty.
3. If head == NULL, display "List is Empty" and stop.
4. If pos == 1, perform DeleteAtBeginning() and stop.
5. Traverse the list until the node at the specified position is reached.
6. If the position is invalid or the node is not found, display "Invalid Position" and stop.
7. Let temp be the node at the specified position.
8. Update the next pointer of the previous node:

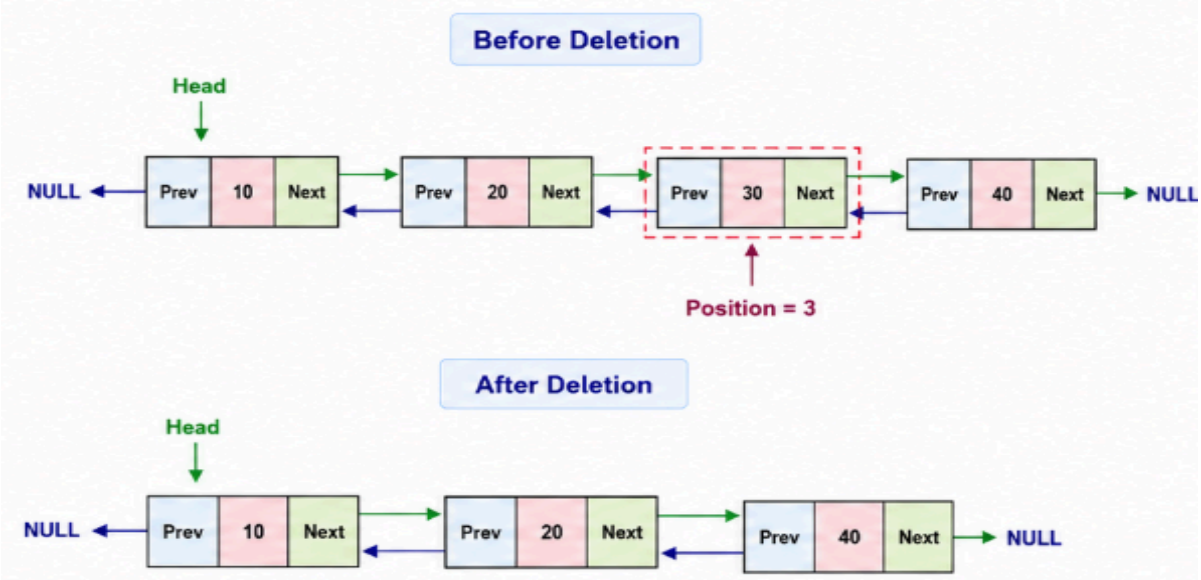
temp->prev->next = temp->next;

9. If temp->next != NULL, update the previous pointer of the next node:

temp->next->prev = temp->prev;

10. Delete (free) temp.
11. Stop.

Graphical Representation



Explanation:

- Node 30 at Position 3 is deleted.
- The next pointer of node 20 is updated to point to node 40.
- The previous pointer of node 40 is updated to point to node 20.

Real-World Example

Application: Browser History

When a webpage is removed from the middle of the browser history (for example, the 3rd visited page), the previous webpage is directly connected to the next webpage. This ensures that the browsing history remains continuous without the deleted page.

Example:

Before deletion:

Google ⇄ Gmail ⇄ YouTube ⇄ ChatGPT

Delete YouTube (3rd page):

Google ⇄ Gmail ⇄ ChatGPT

The Gmail page now directly links to ChatGPT.

Time Complexity

- Best Case: O(1) (when the node is already known or deleting the first node)
- Worst Case: O(n)

Reason

Without a direct reference to the required node, the linked list must be traversed from the beginning to locate the specified position before deletion. Therefore, the deletion operation takes O(n) time in the worst case.

❖ Searching

Searching in a doubly linked list is the process of finding a node containing a specified key (data value). The list is traversed from the head node (or tail node, if available), and each node's data is compared with the search key until the required node is found or the end of the list is reached.

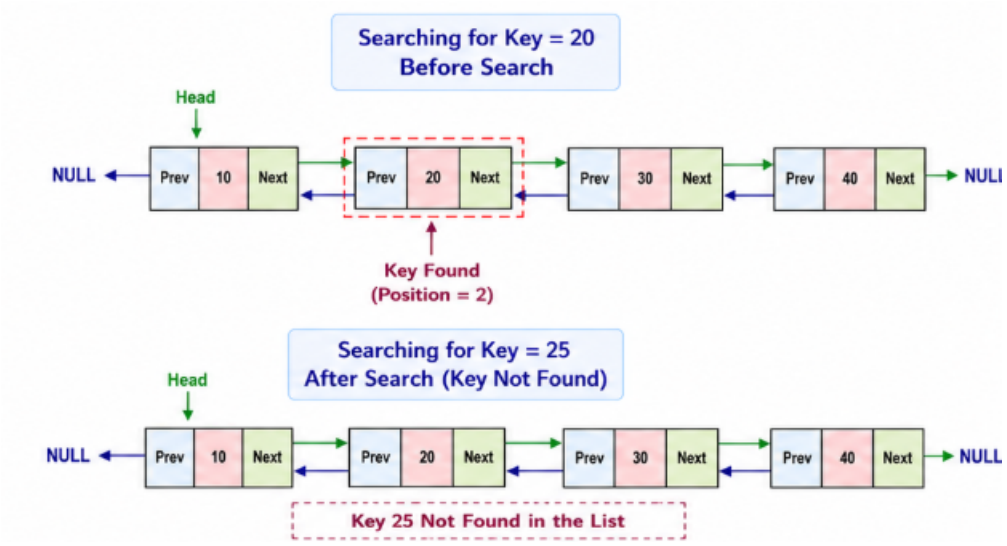
Algorithm: Search(key)

1. Start.
2. Check whether the linked list is empty.
3. If head == NULL, display "List is Empty" and stop.
4. Set temp = head.
5. Repeat until temp == NULL:
    - Compare temp->data with key.
    - If temp->data == key:
        - Display "Key Found".
        - Display the position of the node.
        - Stop.
    - Otherwise, move to the next node:

temp = temp->next;

6. If temp == NULL, display "Key Not Found".
7. Stop.

Graphical Representation



Explanation:

- If the key is present, the search stops immediately and returns the node's position.
- If the key is absent, the entire list is traversed until NULL is reached.

Real-World Example

Application: Contact List

Searching for a person's name in a mobile phone contact list is similar to searching in a doubly linked list. The contacts are checked one by one until the required contact is found.

Example:

Contact List:

Aisha ⇄ Basha ⇄ Chandra ⇄ Divya

Search for Chandra:

Aisha → Not Match

Basha → Not Match

Chandra → Match Found

The search stops as soon as Chandra is found.

If you search for Ravi, the entire list is traversed and the message "Contact Not Found" is displayed.

Time Complexity

- Best Case: O(1)
- Average Case: O(n/2) ≈ O(n)
- Worst Case: O(n)

Reason

- Best Case (O(1)): The required key is found at the first node (head).
- Average Case (O(n)): On average, half of the nodes are traversed before finding the key.
- Worst Case (O(n)): The key is present in the last node or is not present in the list, requiring traversal of all nodes.

❖ Traversal of Doubly Linked Lists

Traversal is the process of visiting each node in a doubly linked list exactly once to access or display the data stored in each node. Since a doubly linked list contains both next and previous pointers, traversal can be performed in two ways:

- Forward Traversal (from the first node to the last node)

- Backward Traversal (from the last node to the first node)

    - **Forward Traversal (From the First Node to the Last Node):**Forward traversal starts from the **head** (first node) and moves towards the **last node** by following the **next** pointer of each node until **NULL** is reached.

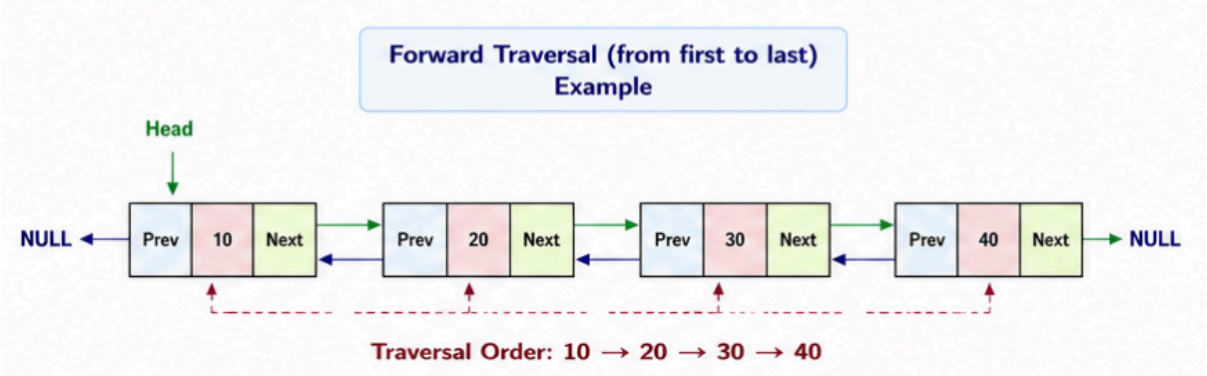
Algorithm: TraverseForward()

1. Start.
2. Check whether the linked list is empty.
3. If head == NULL, display "List is Empty" and stop.
4. Set temp = head.
5. Repeat until temp == NULL:
    - Display (visit) temp->data.
    - Move to the next node:

temp = temp->next;

6. Stop.

Graphical Representation



- **Backward Traversal (From the Last Node to the First Node)**

Backward traversal starts from the **last node (tail)** and moves towards the **first node** by following the **previous (prev)** pointer until **NULL** is reached.

Algorithm: TraverseBackward()

1. **Start.**
2. Check whether the linked list is empty.
3. If **head == NULL**, display **"List is Empty"** and stop.
4. Set **temp = head**.
5. Traverse to the last node:
6. while(temp->next != NULL)

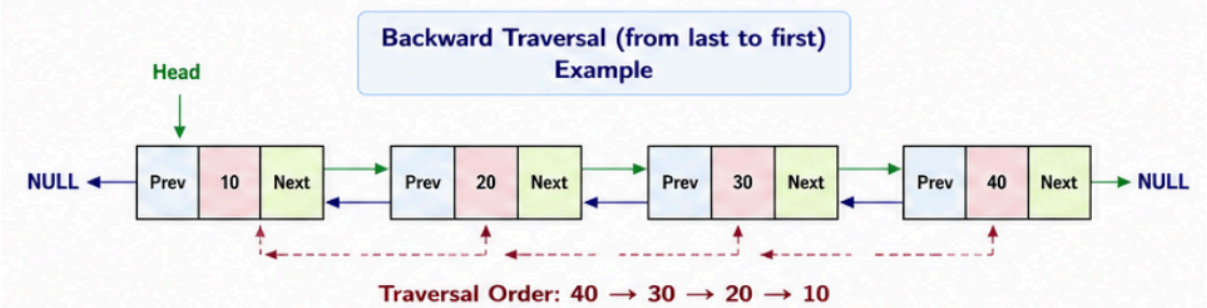
temp = temp->next;

7. Repeat until **temp == NULL**:
    - Display (visit) **temp->data**.
    - Move to the previous node:

temp = temp->prev;

8. **Stop.**

Graphical Representation



Real-World Example

Application: Music Playlist

A music player uses a doubly linked list to navigate songs.

- Forward Traversal: Pressing the Next button plays songs from the first song to the last song.
- Backward Traversal: Pressing the Previous button plays songs from the last song back to the first song.

Time Complexity

Operation	Best Case	Average Case	Worst Case	Reason
Forward Traversal	O(n)	O(n)	O(n)	Every node is visited exactly once.
Backward Traversal	O(n)	O(n)	O(n)	Every node is visited exactly once.

❖ Updating

Updating in a doubly linked list is the process of modifying the **data (value)** stored in an existing node without changing the structure of the list. The **previous (prev)** and **next** pointers remain unchanged; only the data field of the specified node is updated.

- Updating modifies **only the data** stored in a node.
- The **structure of the doubly linked list remains unchanged**.
- No new node is inserted, and no existing node is deleted.
- The **prev** and **next** pointers are not modified.
- Updating is commonly used in applications such as **student records, employee databases, bank account information, inventory management, and contact lists**.

Algorithm: Update(key, newData)

1. **Start.**
2. Check whether the linked list is empty.
3. If **head == NULL**, display **"List is Empty"** and stop.
4. Set **temp = head**.
5. Repeat until **temp == NULL**:
    - If **temp->data == key**:
        - Update the data:

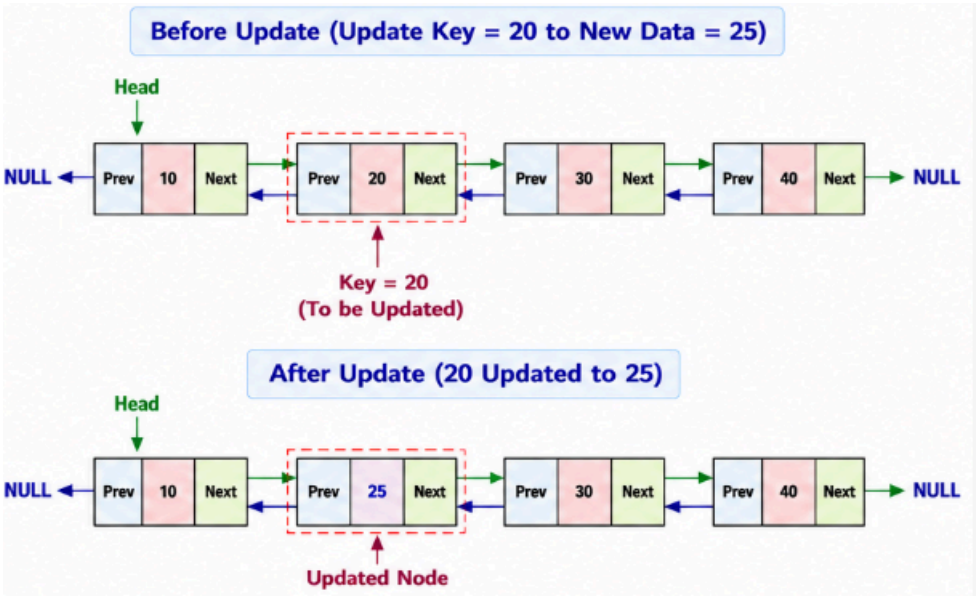
temp->data = newData;

        - Display **"Data Updated Successfully"**.
        - Stop.
    - Otherwise, move to the next node:

temp = temp->next;

6. If **temp == NULL**, display **"Key Not Found"**.
7. **Stop.**

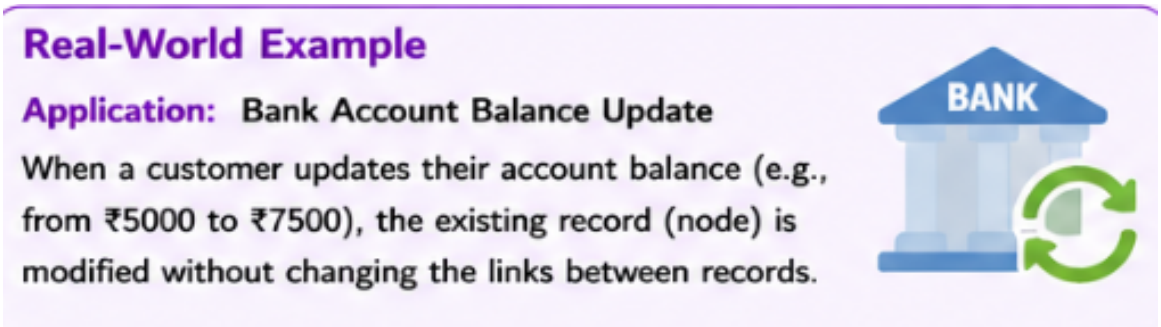
Graphical Representation



Explanation:

- The node containing 20 is found.
- Its value is changed to 25.
- The prev and next pointers remain unchanged.
- Only the data field of the node is modified.

Real-World Example



Time Complexity

Case	Complexity	Reason
Best Case	O(1)	When the key is found at the first node (head).
Worst Case	O(n)	When the key is found at the last node or not found in the list (traversing all nodes).
Average Case	O(n/2) ≈ O(n)	On average, half of the nodes are traversed.

Advantages of Doubly Linked Lists

1. Bidirectional Traversal

A doubly linked list allows traversal in both forward (first to last) and backward (last to first) directions, making navigation more flexible.

2. Easy Insertion

Insertion of a new node before or after a given node is easier because the previous node can be accessed directly using the prev pointer.

3. Easy Deletion

Deletion of a node is more efficient since both the previous and next nodes are directly accessible. There is no need to search for the previous node.

4. Efficient Backward Navigation

The prev pointer allows direct movement to the previous node without restarting the traversal from the head.

5. Efficient Update of Links

During insertion and deletion, updating the links is simpler because both prev and next pointers are available**.**

6. Suitable for Complex Applications

Doubly linked lists are ideal for applications that require movement in both directions, such as:

- Browser history
- Music playlists
- Undo/Redo operations
- Deques (Double-Ended Queues)

7. Supports Advanced Data Structures

DLLs are widely used in implementing advanced data structures such as:

- LRU (Least Recently Used) Cache
- Navigation systems
- Palindrome checking
- Text editors

8. Dynamic Memory Allocation

Memory is allocated dynamically, so the list can grow or shrink during program execution without requiring a fixed size.

Disadvantages of Doubly Linked Lists

1. More Memory Usage

Each node stores an additional prev pointer, so DLL requires more memory than a singly linked list.

2. More Complex Implementation

Insertion, deletion, and pointer management are more complicated because both prev and next pointers must be maintained.

3. More Pointer Updates

Every insertion and deletion operation requires updating two pointers, increasing the possibility of programming errors.

4. Slower Execution

Operations may take slightly more time than singly linked lists because two pointers need to be updated instead of one.

5. Higher Space Overhead

The extra pointer in every node increases the overall memory consumption, especially for large lists.

6. Difficult Debugging

Incorrect updating of either the prev or next pointer can corrupt the list, making debugging more difficult.

7. Not Suitable for Simple Applications

If only forward traversal is required, a singly linked list is simpler and more memory-efficient than a doubly linked list.

8. Risk of List Corruption

Improper maintenance of pointers during insertion or deletion can break the links between nodes, making the list inaccessible.

❖ Circular Linked List

A Circular Linked List (CLL) is a type of linked list in which the last node points back to the first node (head) instead of pointing to NULL. This creates a circular structure, allowing continuous traversal of the list. Unlike a singly linked list, there is no end node because the last node is connected to the first node.

Since the nodes are connected in a circular manner, traversal can start from any node and continue until the starting node is reached again. Circular linked lists are commonly used in applications that require repeated processing of data in a cyclic order, such as CPU scheduling, round-robin scheduling, music playlists, and multiplayer games.

Structure of a Circular Linked List Node

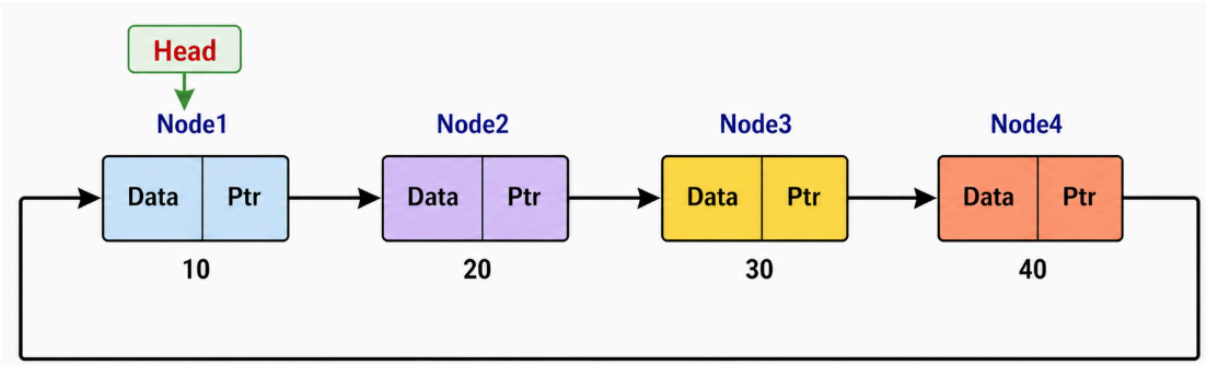
Each node in a circular linked list consists of two components:

- data – Stores the actual data or information.
- next Ptr– Stores the address of the next node.

Node Structure

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

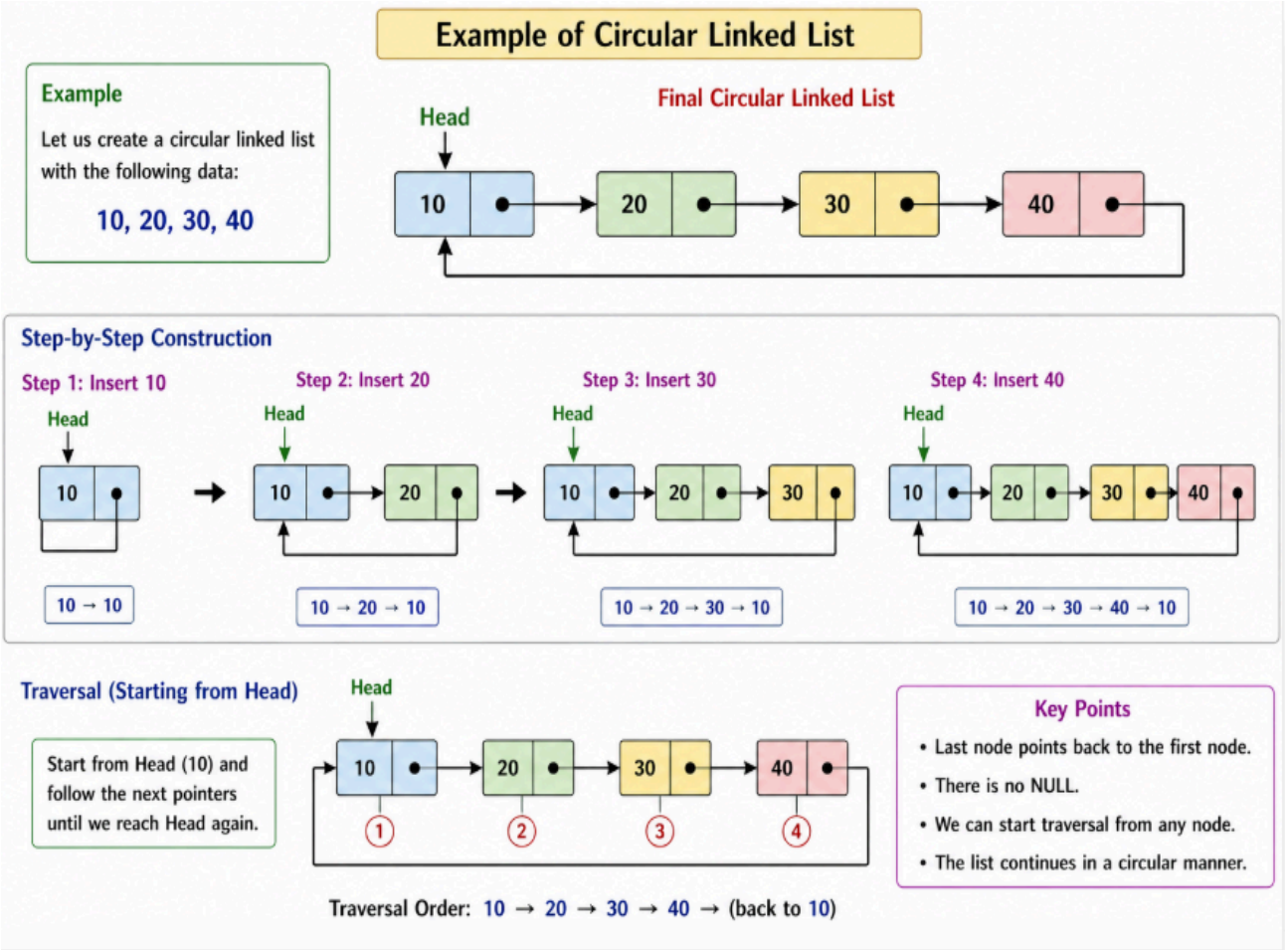
Representation of a Circular Linked List



- Head points to the first node.
- Data stores the actual information.
- Next stores the address of the next node.
- The last node's next pointer points back to the head node instead of NULL.
- The circular connection allows continuous traversal of the linked list.
- Traversal can begin from any node and continues until the starting node is reached again.

Algorithm: CreateCircularLinkedList()

1. **Start.**
2. Create the first node.
3. Read the data and store it in the first node.
4. Set **head = first node**.
5. Set **head->next = head** (the first node points to itself).
6. Repeat for each remaining node:
    - Create a new node.
    - Read the data and store it in the new node.
    - Set **newNode->next = head**.
    - Set **lastNode->next = newNode**.
    - Update **lastNode = newNode**.
7. **Stop.**



Main Operations of Circular Linked List

The primary operations of a Circular Linked List are:

1. Creation
2. Insertion
    - At the Beginning
    - At the End
    - At a Specific Position
3. Deletion
    - At the Beginning
    - At the End
    - At a Specific Position
4. Traversal (Display)
5. Searching
6. Updating

Applications of Circular Linked List

- CPU Round Robin Scheduling
- Circular Queue Implementation
- Music Playlist (Repeat Mode)
- Multiplayer Game Turn Management
- Traffic Light Control Systems
- Operating System Process Scheduling
- Token Ring Networks

Advantages of Circular Linked List

- Supports continuous traversal without reaching NULL.
- Efficient implementation of cyclic processes.
- Easy insertion and deletion at the beginning and end (with a tail pointer).

- Suitable for applications requiring repeated processing.
- Better memory utilization than arrays due to dynamic allocation.

Disadvantages of Circular Linked List

- More difficult to implement than a simple linked list.
- Traversal requires careful stopping conditions to avoid infinite loops.
- Debugging is comparatively difficult.
- Searching still requires O(n) time in the worst case.
- Pointer manipulation is more complex than arrays.

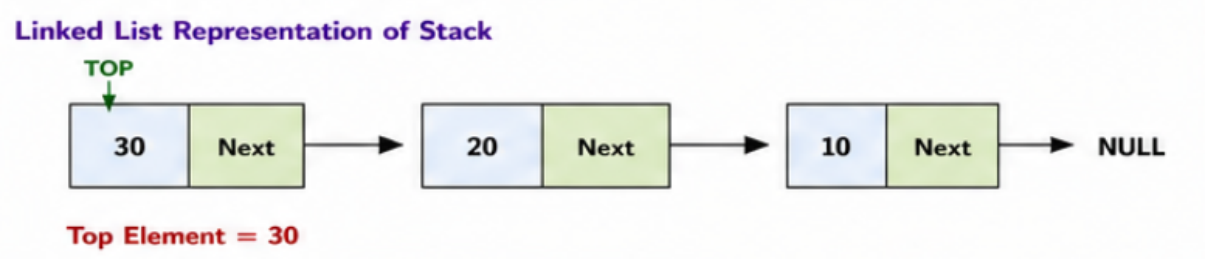
Stack and Queue Representation Using Linked List

A **Stack** and a **Queue** are linear data structures that can be efficiently implemented using a linked list.

- A **Stack** follows the **LIFO (Last In, First Out)** principle, where insertion and deletion are performed at the **TOP** of the list.
- A **Queue** follows the **FIFO (First In, First Out)** principle, where insertion is performed at the **REAR** and deletion is performed at the **FRONT** of the list.

❖ **Stack Using Linked List:** A Stack is a linear data structure in which insertion (PUSH) and deletion (POP) are performed at the same end called the TOP. It follows the Last In, First Out (LIFO) principle.

Graphical representation



Stack Operations

- **PUSH(x):** Insert a new node at the beginning (TOP).
- **POP():** Delete the node from the beginning (TOP).

Algorithm: PUSH(x)

1. Start.
2. Create a new node with data = x.
3. Set newNode->next = top.
4. Set top = newNode.
5. Stop.

Algorithm: POP()

1. Start.
2. Check whether the stack is empty.
3. If top == NULL, display "Stack is Empty" and stop.
4. Set temp = top.
5. Set top = top->next.
6. Delete (free) temp.
7. Stop.

Example of Stack Operations

Operation	Stack (Top → Bottom)	Top
Initial	Empty	NULL
PUSH(10)	10	10
PUSH(20)	20 → 10	20
PUSH(30)	30 → 20 → 10	30
POP()	20 → 10	20
POP()	10	10
POP()	Empty	NULL

Real-World Example

Application: Function Call Management

A stack is used to manage function calls in programming. The most recently called function is executed first, and once it finishes, control returns to the previous function.

Other applications include:



- Expression evaluation
- Undo operations in text editors
- Browser backtracking
- Recursive function calls

Time Complexity

Operation	Best Case	Worst Case
PUSH(x)	O(1)	O(1)
POP()	O(1)	O(1)

Reason: Both PUSH and POP are performed at the top of the linked list without traversal.

Complete C program

This program implements all four operations using a linked list.

```
#include <stdio.h>
#include <stdlib.h>
 struct Node {
    int data;
    struct Node *next;
};
struct Node *top = NULL;
 // Push operation
void push(int value) {
    struct Node *newNode;
    newNode = (struct Node *)malloc(sizeof(struct Node));
```

```
    if (newNode == NULL) {
        printf("Stack Overflow: Memory not available\n");
        return;
    }
    newNode->data = value;
    newNode->next = top;
    top = newNode;
  printf("%d pushed into stack\n", value);
}

// Pop operation
void pop() {
    struct Node *temp;
 if (top == NULL) {
printf("Stack Underflow: Stack is empty\n");
        return;
    }

    temp = top;
    printf("Popped element: %d\n", top->data);
    top = top->next;
    free(temp);
}

// Peek operation
void peek() {
    if (top == NULL) {
        printf("Stack is empty\n");
    } else {
```

```
        printf("Top element: %d\n", top->data);
    }
}

// Display operation
void display() {
    struct Node *temp = top;

    if (top == NULL) {
        printf("Stack is empty\n");
        return;
    }

    printf("Stack elements: ");
while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }

    printf("NULL\n");
}

int main() {
    int choice, value;

    while (1) {
        printf("\n--- STACK MENU ---\n");
        printf("1. Push\n");
        printf("2. Pop\n");
```

```
printf("3. Peek\n");
printf("4. Display\n");
printf("5. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
    case 1:
        printf("Enter value: ");
        scanf("%d", &value);
        push(value);
        break;


  case 2:
        pop();
        break;

    case 3:
        peek();
        break;

    case 4:
        display();
        break;

    case 5:
        while (top != NULL) {
            struct Node *temp = top;
```

```
            top = top->next;
            free(temp);
        }
        return 0;

    default:
        printf("Invalid choice\n");
    }
  }
}
```

Sample output

```
--- STACK MENU ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value: 10
10 pushed into stack

--- STACK MENU ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value: 20
```

20 pushed into stack

--- STACK MENU ---

1. Push

2. Pop

3. Peek

4. Display

5. Exit

Enter your choice: 1

Enter value: 30

30 pushed into stack

--- STACK MENU ---

1. Push

2. Pop

3. Peek

4. Display

5. Exit

Enter your choice: 4

Stack elements: 30 -> 20 -> 10 -> NULL

--- STACK MENU ---

1. Push

2. Pop

3. Peek

4. Display

5. Exit

Enter your choice: 3

Top element: 30

--- STACK MENU ---

1. Push

2. Pop

3. Peek

4. Display

5. Exit

Enter your choice: 2

Popped element: 30

--- STACK MENU ---

1. Push

2. Pop

3. Peek

4. Display

5. Exit

Enter your choice: 4

Stack elements: 20 -> 10 -> NULL

--- STACK MENU ---

1. Push

2. Pop

3. Peek

4. Display

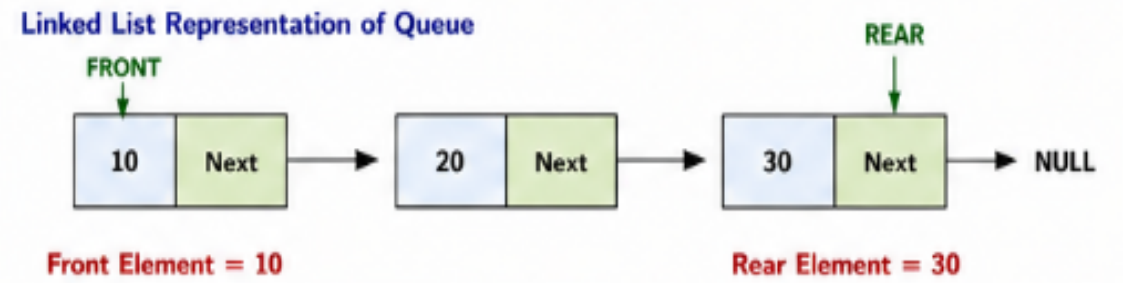
5. Exit

Enter your choice: 5

❖ Queue Using Linked List

A **Queue** is a linear data structure in which insertion (**ENQUEUE**) is performed at the **REAR**, and deletion (**DEQUEUE**) is performed at the **FRONT**. It follows the **First In, First Out (FIFO)** principle.

Linked List Representation of Queue



Queue Operations

- **ENQUEUE(x):** Insert a new node at the rear.
- **DEQUEUE():** Delete the node from the front.

Algorithm: ENQUEUE(x)

1. **Start.**
2. Create a new node with data = **x**.
3. Set **newNode->next = NULL**.
4. If **rear == NULL**:
    - Set **front = rear = newNode**.
5. Otherwise:
    - Set **rear->next = newNode**.
    - Set **rear = newNode**.
6. **Stop.**

Algorithm: DEQUEUE()

1. **Start.**
2. Check whether the queue is empty.
3. If **front == NULL**, display **"Queue is Empty"** and stop.
4. Set **temp = front**.
5. Set **front = front->next**.

6. If **front == NULL**, set **rear = NULL**.
7. Delete (free) **temp**.
8. **Stop.**

Example of Queue Operations

Operation	Queue (Front → Rear)	Front	Rear
Initial	Empty	NULL	NULL
ENQUEUE(10)	10	10	10
ENQUEUE(20)	10 → 20	10	20
ENQUEUE(30)	10 → 20 → 30	10	30
DEQUEUE()	20 → 30	20	30
DEQUEUE()	30	30	30
DEQUEUE()	Empty	NULL	NULL

Real-World Example



Application: Waiting Queue

A queue is commonly used where the first person or task that arrives is served first.

Examples include:

- CPU Scheduling
- Printer Queue
- Ticket Reservation System
- Breadth First Search (BFS)
- Customer waiting lines

Time Complexity

Operation	Best Case	Worst Case
ENQUEUE(x)	O(1)	O(1)
DEQUEUE()	O(1)	O(1)

- **Reason:** Both insertion at the rear and deletion from the front are performed in constant time using front and rear pointers.

Applications of Linked List-Polynomial Representation

Polynomial representation using a linked list stores each polynomial term as a separate node containing the **coefficient**, **exponent**, and **next pointer**. Unlike arrays, linked lists store only the non-zero terms, making polynomial representation memory-efficient. It is widely used in mathematical computations where polynomial operations such as addition, subtraction, multiplication, and evaluation are performed efficiently.

Structure of a Node



Each node consists of the following fields:

- **Coefficient (Coeff):** Represents the coefficient of the term.
- **Exponent (Exp):** Represents the power of the variable.
- **Next Pointer:** Points to the next polynomial term.

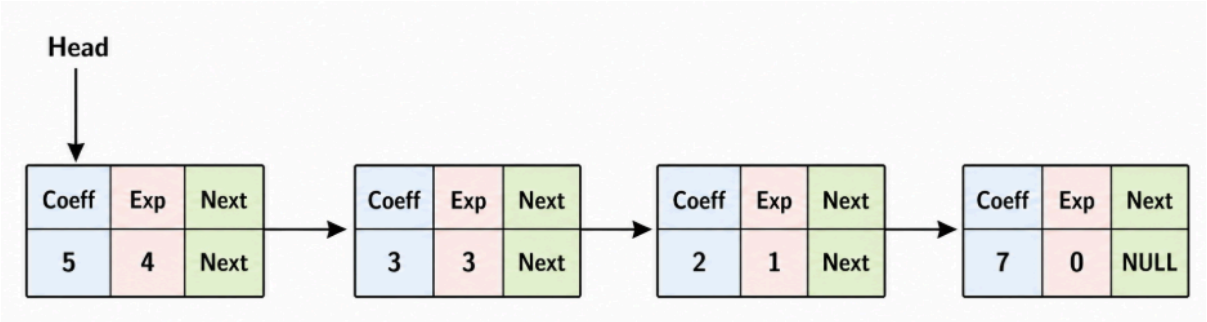
Algorithm: CreatePolynomial()

1. **Start.**
2. Read the number of terms (**n**).
3. Repeat **n** times:
    - Read the coefficient and exponent.
    - Create a new node.
    - Store the coefficient and exponent.
    - Insert the node at the end of the linked list.
4. Display the polynomial.
5. **Stop.**

Graphical Representation

Polynomial:

$$P(x) = 5x^4 + 3x^3 + 2x + 7$$



Real-World Example

Application: Mathematical Software

Mathematical software such as MATLAB, Mathematica, and symbolic algebra systems use linked lists to represent polynomial expressions efficiently.

For example:

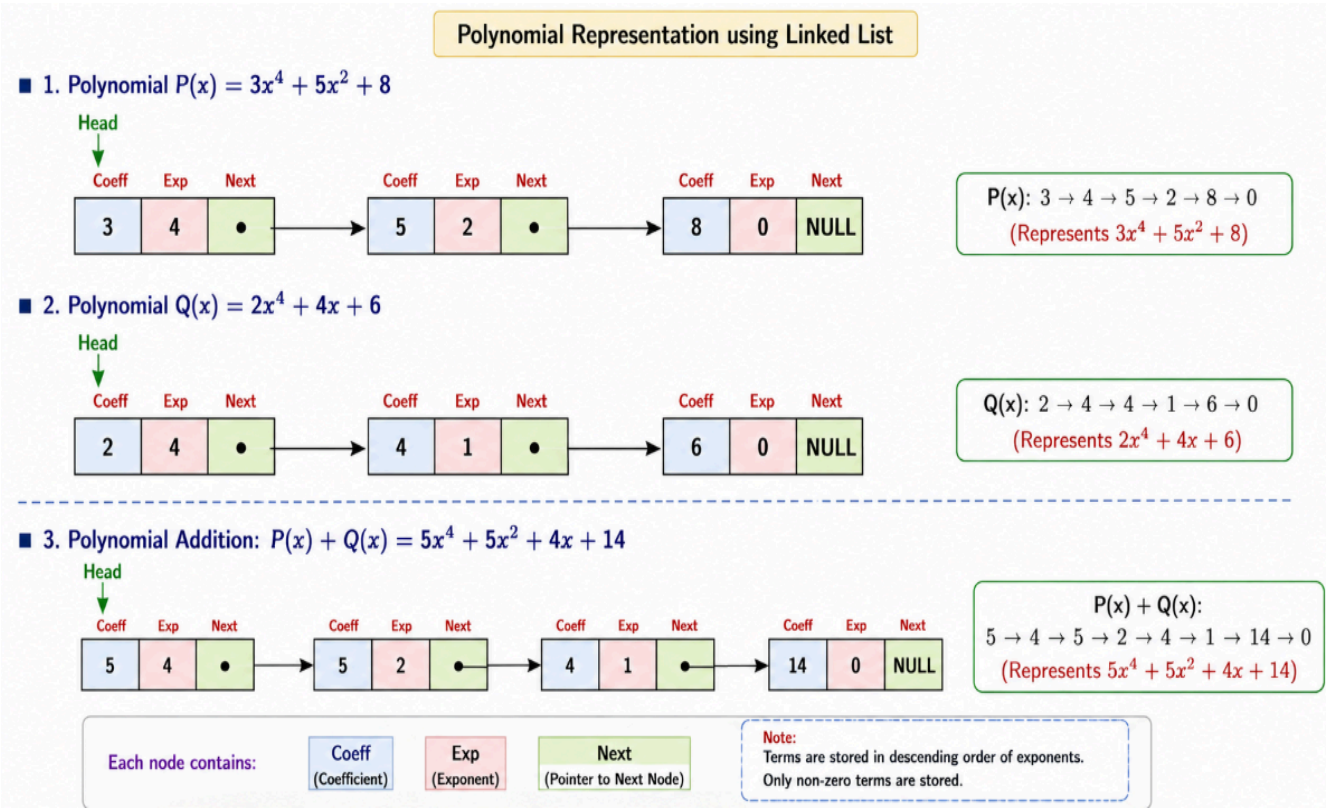
$$P(x) = 3x^4 + 5x^2 + 8$$

$$Q(x) = 2x^4 + 4x + 6$$

During polynomial addition:

$$P(x) + Q(x) = 5x^4 + 5x^2 + 4x + 14$$

The linked list makes it easy to compare exponents and combine like terms.



[P(x)=5x^3+4x^2+2x+7]

[Q(x)=3x^3+2x^2+6x+1]

Compare the exponents and add coefficients:

Terms	Calculation	Result
(x^3) terms	(5+3)	(8x^3)
(x^2) terms	(4+2)	(6x^2)
(x) terms	(2+6)	(8x)
Constant terms	(7+1)	(8)

Final output:

$$8x^3 + 6x^2 + 8x + 8$$

Advantages of Polynomial Representation Using Linked List

1. **Memory Efficient**
    - Stores only non-zero terms.
2. **Dynamic Size**
    - No fixed memory allocation is required.
3. **Easy Polynomial Operations**
    - Supports efficient addition, subtraction, and multiplication.
4. **Easy Insertion and Deletion**
    - Polynomial terms can be inserted or removed without shifting elements.
5. **Suitable for Sparse Polynomials**
    - Efficiently represents polynomials containing only a few non-zero terms.

Time Complexity

Operation	Time Complexity
Create Polynomial	O(n)
Display Polynomial	O(n)
Search a Term	O(n)
Polynomial Addition	O (n + m)
Polynomial Multiplication	O (n × m)

```
#include <stdio.h>
#include<stdlib.h>
struct node {
   int coff;
   int exp;
   struct node *next;
};
void readpoly(struct node **head){
int n,i;
struct node *temp,*cur;
printf("number of terms");
scanf("%d", &n);
for(i=0;i<n;i++){
temp=(struct node *)malloc(sizeof(struct node));
scanf("%d%d",&temp->coff,&temp->exp);
temp->next=NULL;
if (*head==NULL)
*head=cur=temp;
else{
```

```
cur->next=temp;
cur=temp;
}
}
}
struct node* addpoly(struct node *x,struct node *y){
   struct node *temp1,*temp2,*temp3,*head=NULL,*cur;
   temp1=x;
   temp2=y;
   while(temp1!=NULL && temp2!=NULL){
      temp3=(struct node *)malloc(sizeof(struct node));
      if(temp1->exp>temp2->exp){
         temp3->coff=temp1->coff;
         temp3->exp=temp1->exp;
         temp1=temp1->next;
      }
      else if(temp1->exp<temp2->exp){
         temp3->coff=temp2->coff;
         temp3->exp=temp2->exp;
         temp2=temp2->next;
      }
      else{
         temp3->coff=temp1->coff+temp2->coff;
         temp3->exp=temp1->exp;
         temp1=temp1->next;
         temp2=temp2->next;
      }
      temp3->next=NULL;
      if(head==NULL){
         head=cur=temp3;
      }
      else{
         cur->next=temp3;
         cur=temp3;
      }
   }
   while(temp1!=NULL){
      temp3=(struct node *)malloc(sizeof(struct node));
      temp3->coff=temp1->coff;
      temp3->exp=temp1->exp;
      temp3->next=NULL;
      cur->next=temp3;
      cur=temp3;
```

```
        temp1=temp1->next;
    }
    while(temp2!=NULL){
        temp3=(struct node *)malloc(sizeof(struct node));
        temp3->coff=temp2->coff;
        temp3->exp=temp2->exp;
        temp3->next=NULL;
        cur->next=temp3;
        cur=temp3;
        temp2=temp2->next;
    }
    return head;
    }
    void display(struct node *temp){
        for(;temp->next!=NULL;temp=temp->next){
            printf("%dx^%d+",temp->coff,temp->exp);
        }
        printf("%dx^%d",temp->coff,temp->exp);
    }

int main(void) {
    struct node *p = NULL;
    struct node *q = NULL;
    struct node *r = NULL;

    printf("enter polynomial p(x):\n");
    readpoly(&p);

    printf("\nenter polynomial q(x):\n");
    readpoly(&q);

    printf("\np(x) = ");
    display(p);

    printf("\nq(x) = ");
    display(q);

    r = addpoly(p, q);

    printf("\n\nresult p(x) + q(x) = ");
    display(r);

    printf("\n");
```

```
    return 0;
}
```

P(x) = 5x^3 + 4x^2 + 2x + 7
Q(x) = 3x^3 + 2x^2 + 6x + 1

Result P(x) + Q(x) = 8x^3 + 6x^2 + 8x + 8