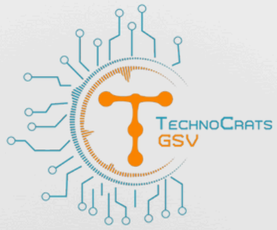


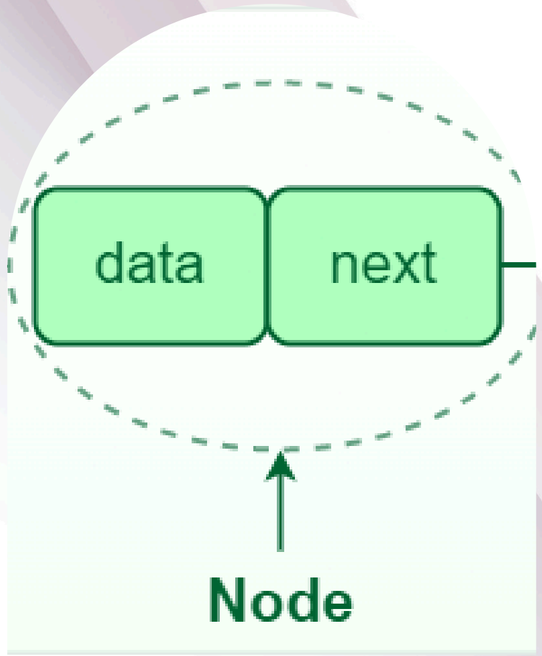

DSA Series Session- 4

# Linked List

Understanding the Basics of Dynamic Data Structures

Presented by: Shivam Gangwar

# What is a Linked List?




- A linear data structure made up of nodes.
- Unlike arrays, elements are not stored in contiguous memory.

Each node contains:

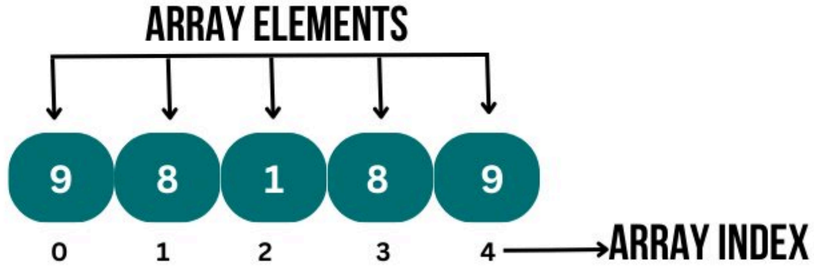
- Data
- Pointer to next node

- It allows efficient insertion and deletion compared to arrays.
- Linked lists are also used to build other structures like stacks, queues, and deques.

Real-Life Analogy

- Think of a linked list like a train:
- Each coach is a node.
- Connectors between coaches act like pointers.

# Comparison B/W Linked List & Array


## • Linked List

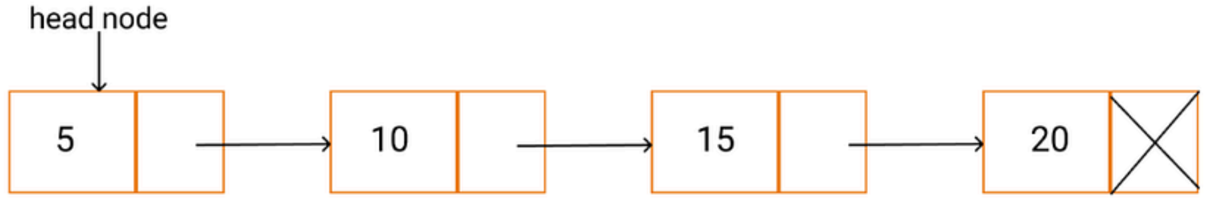
- **Data Structure**: Non-contiguous
- **Memory Allocation**: Typically allocated one by one to individual elements
- **Insertion/Deletion**: Efficient
- **Access**: Sequential

## • Array

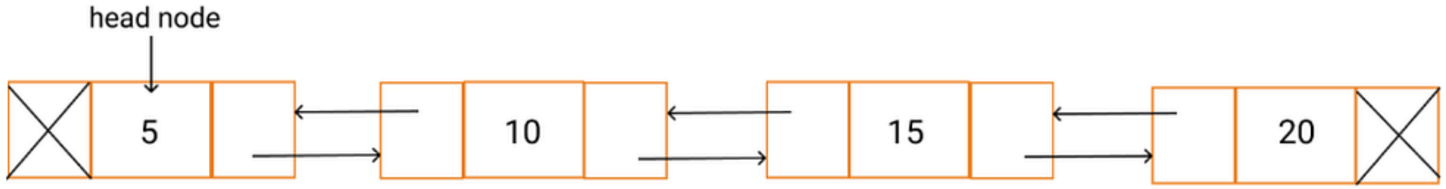
- **Data Structure:** Contiguous
- **Memory Allocation:** Typically allocated to the whole array
- **Insertion/Deletion: Inefficient**
- **Access:** Random

# Types of Linked Lists

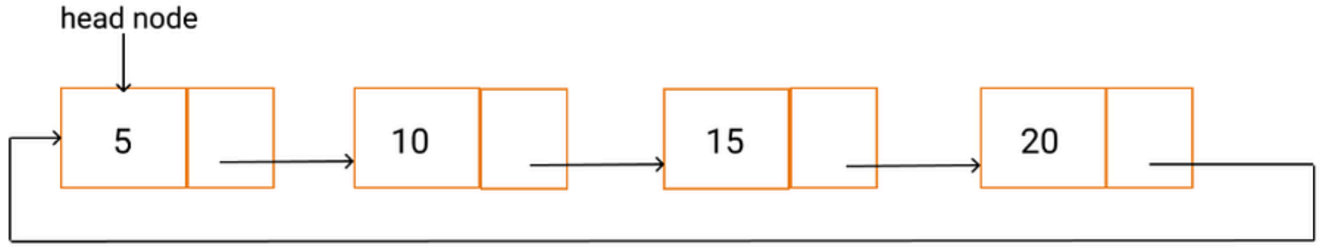
Singly Linked List



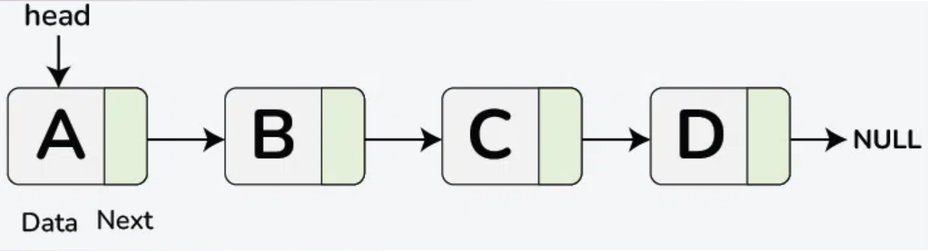

Doubly Linked List




Circular Linked List



# Singly Linked List




A singly linked list is a basic data structure made of nodes.

Each node has:

- Data
- A link to the next node

The last node points to null, showing the end of the list.

It allows fast insertion and deletion compared to arrays.

**In C Programming**

```
struct Node {
    int data;
    struct Node* next;
};
// Function to create a new Node
struct Node* newNode(int data) {
    struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = data;
    temp->next = NULL;
    return temp;
}
```

# Operations on Singly Linked List

## • Traversal of Singly Linked List

The process of traversing a singly linked list involves printing the value of each node and then going on to the next node and print that node's value also and so on, till we reach the last node in the singly linked list, whose next node points towards the null.

## • Searching in Singly Linked List

The idea is to traverse all the nodes of the linked list, starting from the head. While traversing, if we find a node whose value is equal to key then print "Yes", otherwise print "No".

## • Insertion in Singly Linked List

a. Insertion at the Beginning of Singly Linked List
b. Insertion at the End of Singly Linked List
c. Insertion at a Specific Position of the Singly Linked List

## • Deletion in Singly Linked List

a. Deletion at the Beginning of Singly Linked List
b. Deletion at the End of Singly Linked List
c. Deletion at a Specific Position of Singly Linked List

# Traversal of Singly Linked List

(Iterative Approach)
Step-by-Step Algorithm:

- We will initialize a temporary pointer to the head node of the singly linked list.
- After that, we will check if that pointer is null or not null, if it is null, then return.
- While the pointer is not null, we will access and print the data of the current node, then we move the pointer to next node.

Time Complexity: O(n), where n is the number of nodes in the linked list.
Space Complexity: O(1)

## Structure

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

## Traversal Function

```
void traverse(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d → ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}
```

# Searching in Singly Linked List

(Iterative Approach)
Step-by-Step Algorithm:

- Initialize a node pointer, curr = head.
- Do following while current is not NULL
  - If the current value (i.e., curr->key) is equal to the key being searched return true.
  - Otherwise, move to the next node (curr = curr->next).
- If the key is not found, return false

Time Complexity : O(N)
Space Complexity: O(1)

## Structure

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

## Search Function

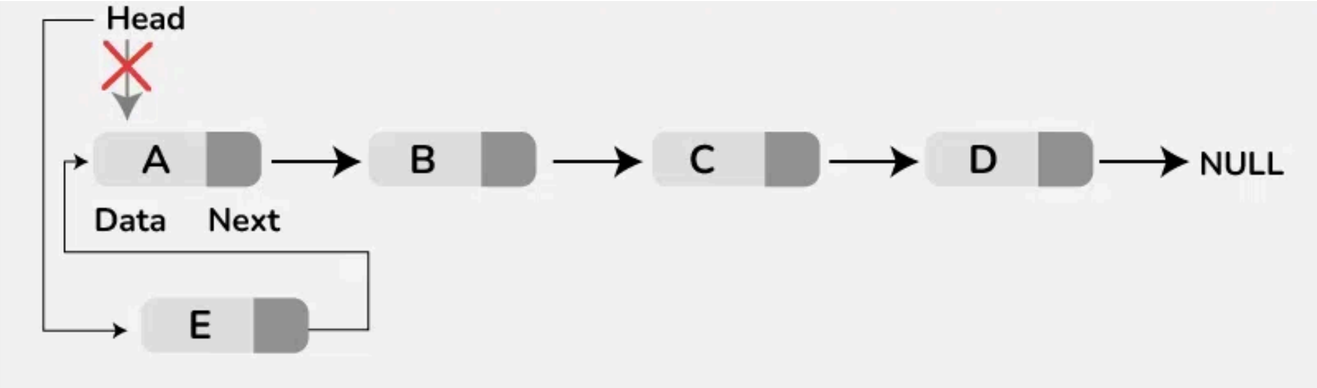
```
void search(struct Node* head, int key) {
    struct Node* current = head;
    while (current != NULL) {
        if (current->data == key) {
            printf("Element %d found.\n", key);
            return;
        }
        current = current->next;
    }
    printf("Element %d not found.\n", key);
}
```

# Insertion in Singly Linked List

## a. Insertion at the Beginning

Step-by-step approach:

- Create a new node with the given value.
- Set the next pointer of the new node to the current head.
- Move the head to point to the new node.
- Return the new head of the linked list.




## Structure

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

## Insertion Function

```
// Function to insert at the beginning
struct node *insertAtBeg(struct node *head, int data) {
  struct node *new_node = (struct node *)malloc(sizeof(struct node));
  if (new_node == NULL) {
    printf("Overflow\n");
    return NULL;
  }
  new_node->data = data;
  new_node->next = head;
  head = new_node;
  return head;
}
```

Time Complexity: O(1)

Space Complexity : O(1)

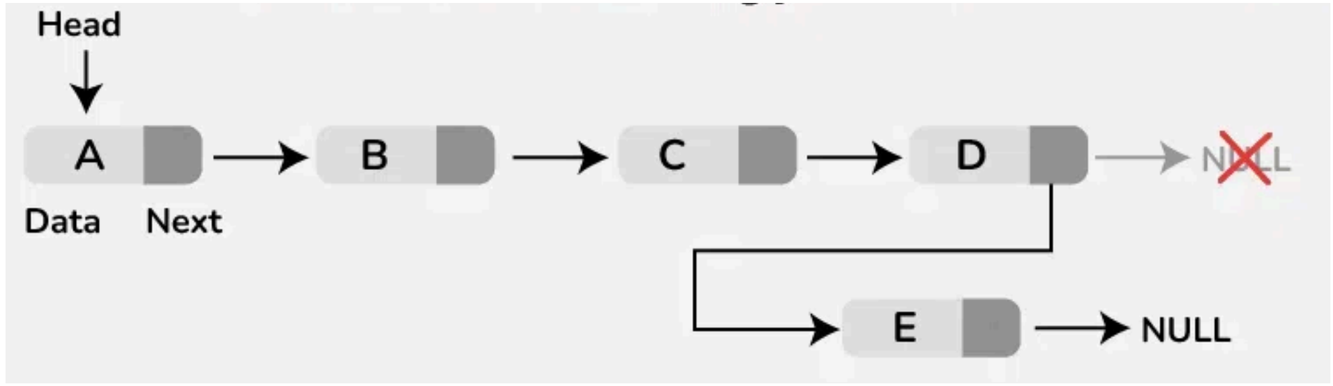
# Insertion in Singly Linked List

## b. Insertion at the End

Step-by-step approach:

- Create a new node with the given value.
- Check if the list is empty:
- If it is, make the new node the head and return.
- Traverse the list until the last node is reached.
- Link the new node to the current last node by setting the last node's next pointer to the new node.

Time Complexity: O(N) where N is the length of the linked list
Space Complexity : O(1)




## Structure

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

## Insertion Function

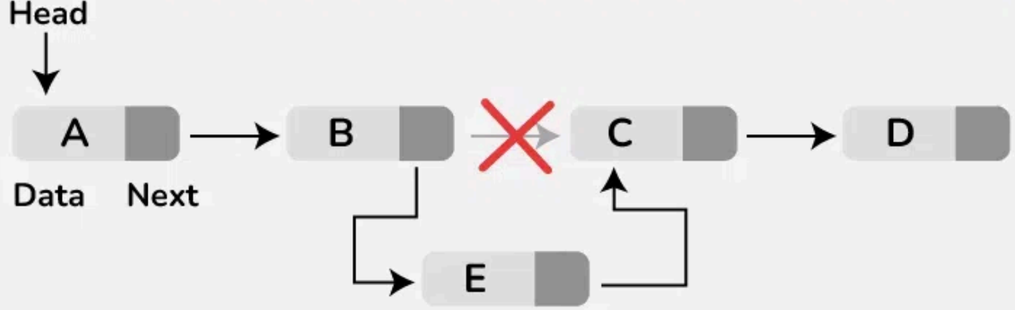
```
struct node *insertAtEnd(struct node *head, int data) {
  struct node *new_node = (struct node *)malloc(sizeof(struct node));
  if (new_node == NULL) {
    printf("Overflow\n");
    return NULL;
  }
  new_node->data = data;
  new_node->next = NULL;

  if (head == NULL) {
    return new_node; // If list is empty, new node becomes the head
  }

  struct node *temp = head;
  while (temp->next != NULL) {
    temp = temp->next;
  }

  temp->next = new_node;
  return head;
}
```

# Insertion in Singly Linked List

## c. Insertion at a Specific Position

Step-by-step approach:

- Create a new node and assign it a value.
- If inserting at the beginning (position = 1):
  - Point the new node's next to the current head.
  - Update the head to the new node.
  - Return (Insertion done).
- Otherwise, traverse the list:
  - Start from the head and move to the (position - 1)$^{th}$ node (just before the desired position).
  - If the position is beyond the list length, return an error or append at the end.
- Insert the new node:
  - Point the new node's next to the next node of the current position.
  - Update the previous node's next to the new node.
- Return the updated list.




## Insertion Function

```
struct Node* insertAtPosition(struct Node* head, int pos, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = newData;

    // Insertion at the beginning (position 1)
    if (pos == 1) {
        newNode->next = head;
        return newNode;
    }

    // Traverse to the (pos - 1)th node
    struct Node* temp = head;
    for (int i = 1; i < pos - 1 && temp!= NULL; i++) {
        temp= temp->next;
    }

    // If position is invalid
    if (temp== NULL) {
        printf("Position out of range.\n");
        free(newNode);
        return head;
    }

    // Insert the new node
    newNode->next = temp->next;
    temp->next = newNode;

    return head;
}
```

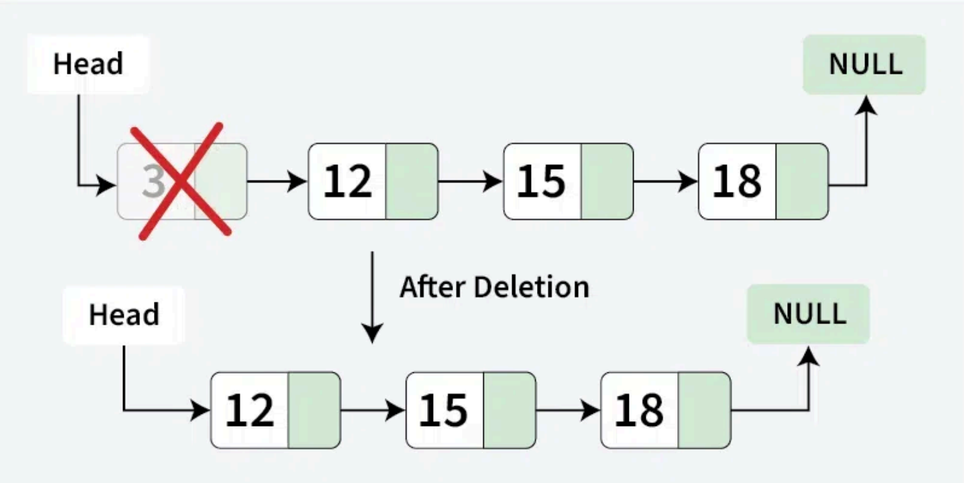
Time Complexity: O(n),
Space Complexity : O(1)

# Deletion in Singly Linked List

## a. Deletion at the Beginning

Steps-by-step approach:

- Check if the head is NULL.
  - If it is, return NULL (the list is empty).
- Store the current head node in a temporary variable temp.
- Move the head pointer to the next node.
- Delete the temporary node.
- Return the new head of the linked list.




## Deletion Function

```
struct Node* deleteAtBeginning(struct Node* head) {
    if (head == NULL) {
        printf("List is already empty.\n");
        return NULL;
    }

    struct Node* temp = head;
    head = head->next;
    free(temp);  // Free memory of old head

    return head;
}
```

Time Complexity: O(1)

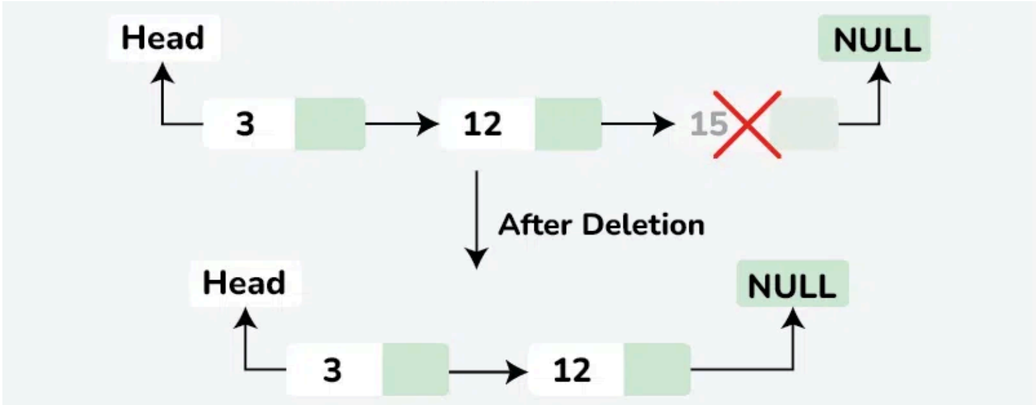
Space Complexity: O(1)

# Deletion in Singly Linked List

## b. Deletion at the End

Step-by-step approach:

- Check if the head is NULL.
  - If it is, return NULL (the list is empty).
- Check if the head's next is NULL (only one node in the list).
  - If true, delete the head and return NULL.
- Traverse the list to find the second last node (second_last).
- Delete the last node (the node after second_last).
- Set the next pointer of the second last node to NULL.
- Return the head of the linked list.




## Deletion Function

```
struct Node* deleteAtEnd(struct Node* head) {
    if (head == NULL) {
        printf("List is already empty.\n");
        return NULL;
    }

    // If only one node
    if (head->next == NULL) {
        free(head);
        return NULL;
    }

    // Traverse to the second last node
    struct Node* current = head;
    while (current->next->next != NULL) {
        current = current->next;
    }

    // Delete last node
    free(current->next);
    current->next = NULL;

    return head;
}
```

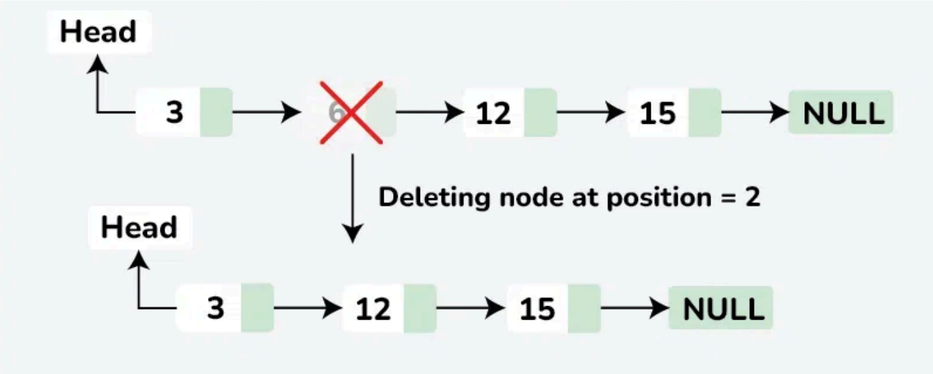
Time Complexity: O(n)
Space Complexity : O(1)

# Deletion in Singly Linked List

## c. Deletion at a Specific Position

Step-by-step approach:

- Check if the list is empty or the position is invalid, return if so.
- If the head needs to be deleted, update the head and delete the node.
- Traverse to the node before the position to be deleted.
- If the position is out of range, return.
- Store the node to be deleted.
- Update the links to bypass the node.
- Delete the stored node.




## Deletion Function

```
struct Node* deleteAtPosition(struct Node* head, int pos) {
    if (head == NULL) {
        printf("List is already empty.\n");
        return NULL;
    }

    // Delete first node
    if (pos == 1) {
        struct Node* temp = head;
        head = head->next;
        free(temp);
        return head;
    }

    // Traverse to (pos - 1)th node
    struct Node* current = head;
    for (int i = 1; i < pos - 1 && current != NULL; i++) {
        current = current->next;
    }

    // If position is invalid
    if (current == NULL || current->next == NULL) {
        printf("Position out of range.\n");
        return head;
    }

    // Delete node at position
    struct Node* temp = current->next;
    current->next = temp->next;
    free(temp);

    return head;
}
```

Time Complexity: O(n)

Space Complexity : O(1)

# Modify a Singly Linked List

Step-by-step approach:

- Start from the head of the list.
- Traverse to the required position (move current node to position).
- Check if the position is valid:
  - If the position is out of bounds, return an error.
- Update the node's data with the new value.
- Return the modified list.

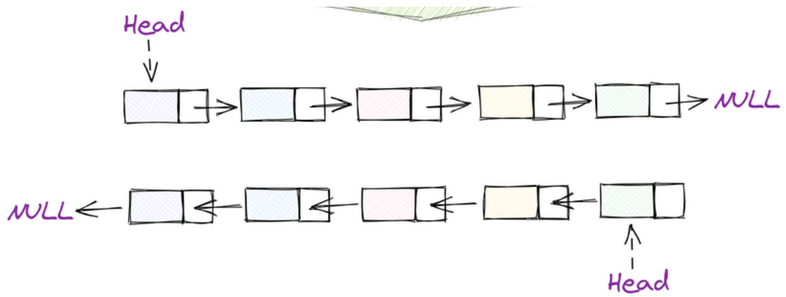
## Modify Function

```
void modifyNode(struct Node* head, int pos, int newData) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* current = head;

    // Traverse to the node at position
    for (int i = 1; i < pos && current != NULL; i++) {
        current = current->next;
    }

    // If position is invalid
    if (current == NULL) {
        printf("Position out of range.\n");
        return;
    }

    // Modify the data
    current->data = newData;
        printf("Node at position %d modified to %d.\n", pos,
newData);
}
```

# Reversing a Singly Linked List

Step-by-step approach:

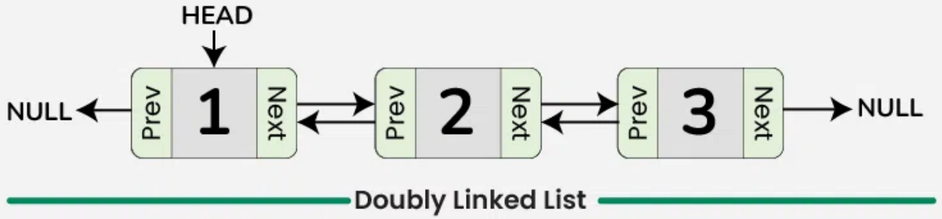
- Initialize three pointers:
  - prev = NULL (to track the previous node)
  - current = head (starting point)
  - next = NULL (to store the next node temporarily)
- Iterate through the list:
  - Store next = current->next (save next node).
  - Reverse the link: current->next = prev.
  - Move prev and current forward (prev = current, current = next).
- Update head to prev (new head is the last node).




## Reverse Function

```
struct Node* reverseList(struct Node* head) {
    struct Node* prev = NULL;
    struct Node* current = head;
    struct Node* next = NULL;

    while (current != NULL) {
        next = current->next;     // Store next node
        current->next = prev;     // Reverse current node's link
        prev = current;           // Move prev forward
        current = next;           // Move current forward
    }

    return prev;  // New head of the reversed list
}
```

# Doubly Linked List

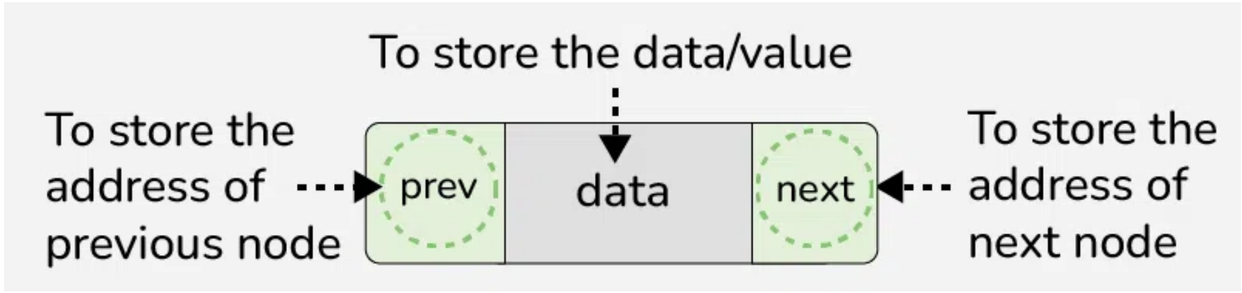



A doubly linked list is a data structure where each node has links to both the next and previous nodes.

Each node has:

- Data
- A link to the next node
- A link to the previous node

It allows traversal in both directions and easier insertion or deletion from any position.




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

// Function to create a new node
struct Node *createNode(int new_data) {
    struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = NULL;
    new_node->prev = NULL;
    return new_node;
}
```

# Operations on Doubly Linked List

## • Traversal of Doubly Linked List

Traversal in a Doubly Linked List involves visiting each node, processing its data, and moving to the next or previous node using the forward (next) and backward (prev) pointers.

## • Searching in Doubly Linked List

Traversing the list from either the head or tail, checking each node's data against the search value. If a match is found, the search is successful; otherwise, it continues until the end of the list is reached.

## • Insertion in Doubly Linked List

a. Insertion at the Beginning of Doubly Linked List
b. Insertion at the End of Doubly Linked List
c. Insertion at a Specific Position of the Doubly Linked List

## • Deletion in Doubly Linked List

a. Deletion at the Beginning of Doubly Linked List
b. Deletion at the End of Doubly Linked List
c. Deletion at a Specific Position of Doubly Linked List

# Traversal of Doubly Linked List

Step-by-Step Approach for Traversal:

1. Start from the head of the list.
2. Traverse forward:
   - Visit the current node and process its data.
   - Move to the next node using current = current->next.
   - Repeat the process until the end of the list (current == NULL).

Time Complexity: O(n), where n is the number of nodes in the linked
Space Complexity : O(1)

## Structure

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

## Function to Traverse Forward

```
void traverseForward(struct Node*
head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
}
```

# Searching in Doubly Linked List

Step 1: Start from the head node.
Step 2: Initialize a pointer current = head.
Step 3: While current is not NULL:

- a. Compare current->data with the target value.
- b. If they match, element is found → Exit.
- c. Else, move current = current->next.

Step 4: If end of the list is reached and no match, then element not found.

Time Complexity : O(N)
Space Complexity: O(1)

## Structure

```
struct Node {
    int data
    Node* next
    Node* prev
}
```

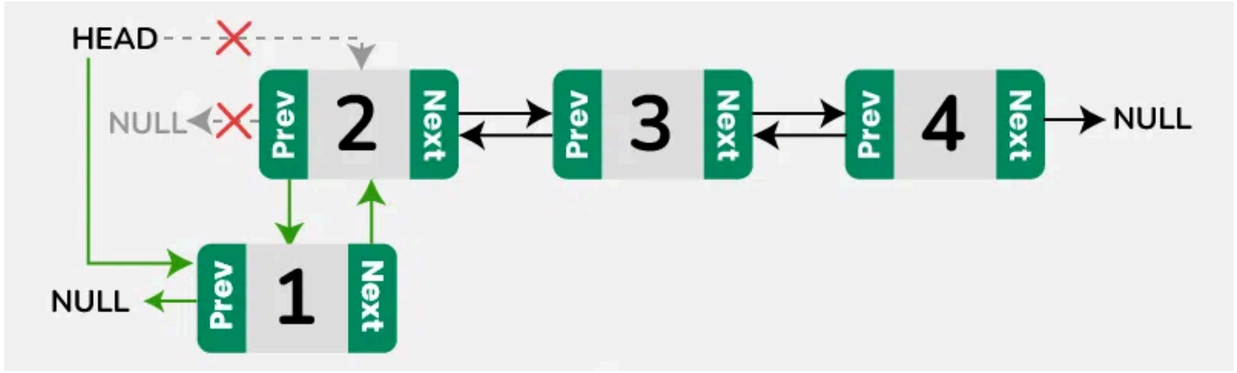
## Search Function Algo

```
Start from the head node
Set position = 1

While current node is not NULL
    If current node's data == key
            Print "Element found at position", position
        Exit
    Move to the next node
    Increase position by 1
End While

Print "Element not found"
```

# Insertion in Doubly Linked List

## a. Insertion at the Beginning

- Create a new node, say new_node with the given data and set its previous pointer to null, new_node->prev = NULL.
- Set the next pointer of new_node to the current head, new_node->next = head.
- If the linked list is not empty, update the previous pointer of the current head to new_node, head->prev = new_node.
- Return new_node as the head of the updated linked list.




## Structure

```
struct Node {
    int data
    Node* next
    Node* prev
}
```

## Insertion Function

```
struct node *insertAtBeg(struct node *head, int data)
{
  struct node *new_node = createNode(data);
  if (head == NULL)
  {
    return new_node;
  }
  new_node->next = head;
  head->prev = new_node;   // Set the prev pointer of the old head
  head = new_node;
  return head;
}
```

Time Complexity: O(1)

Space Complexity : O(1)

# Insertion in Doubly Linked List

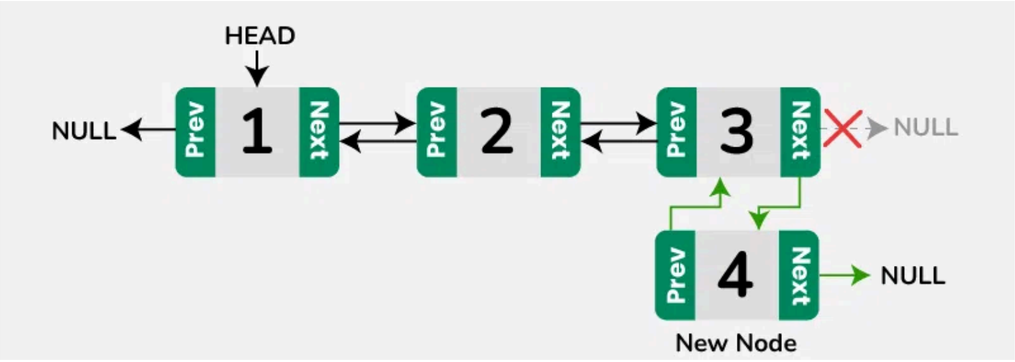
## b. Insertion at the End

Step-by-step approach:

- If the linked list is empty, we set the new node as the head of linked list and return it as the new head of the linked list.
- Otherwise, traverse the entire list until we reach the last node, say curr.

Then, set the last node's next to new node and new node's prev to last node, making the new node the last element in the list.

Time Complexity: O(N) where N is the length of the linked list
Space Complexity : O(1)



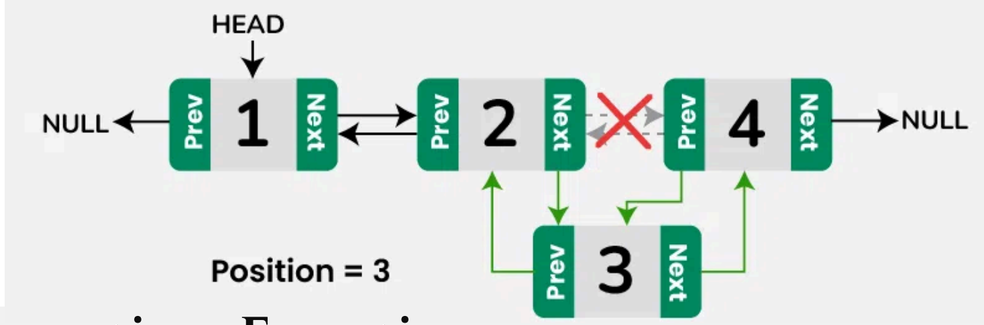

## Structure

```
struct Node {
    int data
    Node* next
    Node* prev
}
```

## Insertion Function

```
// Function to insert at the end of the doubly linked list
struct node *insertAtEnd(struct node *head, int data){
  struct node *new_node = createNode(data);
  if (head == NULL)
  {
    return new_node;
  }

  struct node *temp = head;
  while (temp->next != NULL)
  {
    temp = temp->next;
  }

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

  return head;
}
```

# Insertion in Doubly Linked List




## c. Insertion at a Specific Position

Step-by-step approach:

- If position = 1, create a new node and make it the head of the linked list and return it.
- Otherwise, traverse the list to reach the node at position – 1, say curr.
- If the position is valid, create a new node with given data, say new_node.
- Update the next pointer of new node to the next of current node and prev pointer of new node to current node, new_node->next = curr->next and new_node->prev = curr.
- Similarly, update next pointer of current node to the new node, curr->next = new_node.
- If the new node is not the last node, update prev pointer of new node's next to the new node, new_node->next->prev = new_node.

## Insertion Function

```
struct Node* insertAtPosition(struct Node *head, int pos, int new_data) {
 struct Node *new_node = createNode(new_data);
 // Insertion at the beginning
 if (pos == 1) {
  new_node->next = head;
  if (head != NULL) {
   head->prev = new_node;
  }
  head = new_node;
  return head;
 }

 struct Node *curr = head;
 // Traverse the list to find the node before the insertion point
 for (int i = 1; i < pos - 1 && curr != NULL; ++i) {
  curr = curr->next;
 }
 if (curr == NULL) {         // If the position is out of bounds
  printf("Position is out of bounds.\n");
  free(new_node);
  return head;
 }
 new_node->prev = curr;    // Set the prev of new node to curr
 new_node->next = curr->next; // Set the next of new node to next of curr
 curr->next = new_node;    // Update the next of current node to new node

 // If the new node is not the last node, update the prev of next node to new node
 if (new_node->next != NULL) {
  new_node->next->prev = new_node;
 }
 return head;
}
```

Time Complexity: O(N)

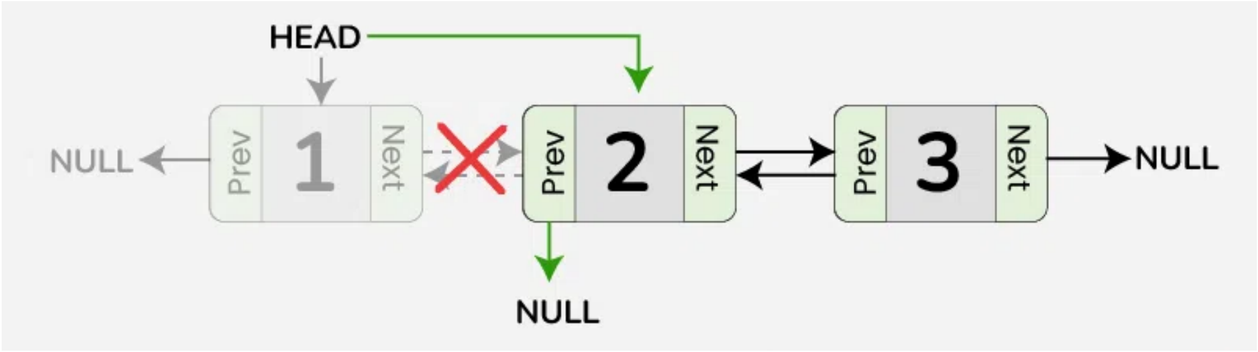
Space Complexity : O(1)

# Deletion in Doubly Linked List

## a. Deletion at the Beginning

Steps-by-step approach:

- Check if the list is empty, there is nothing to delete, return.
- Store the head pointer in a variable, say temp.
- Update the head of linked list to the node next to the current head, head = head->next.
- If the new head is not NULL, update the previous pointer of new head to NULL, head->prev = NULL.




## Deletion Function

```
struct node *DeletionAtBeginning(struct node *head){
  struct node *ptr = head;
  if (head == NULL)
  {
    printf("Underflow..\n");
    return NULL;
  }
  head = head->next;
  head->prev = NULL;
  free(ptr);
  return head;
}
```

Time Complexity: O(1)

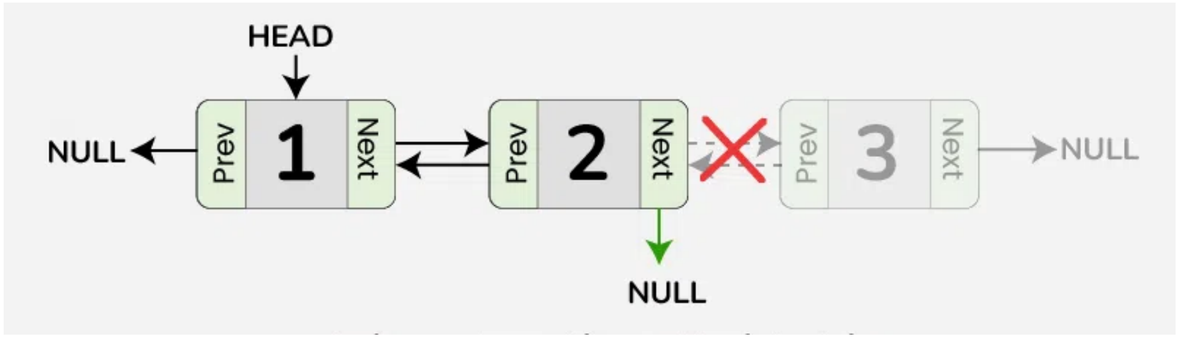
Space Complexity: O(1)

# Deletion in Doubly Linked List

## b. Deletion at the End

Step-by-step approach:

- Check if the doubly linked list is empty. If it is empty, then there is nothing to delete.
- If the list is not empty, then move to the last node of the doubly linked list, say curr.
- Update the second-to-last node's next pointer to NULL, curr->prev->next = NULL.
- Free the memory allocated for the node that was deleted




## Deletion Function

```
struct node *DeletionAtEnd(struct node* head){
  struct node* ptr = head;
  while(ptr->next!=NULL){
    ptr=ptr->next;
  }
  ptr->prev->next=NULL;
  free(ptr);
  return head;
}
```

Time Complexity: O(n)
Space Complexity : O(1)

# Deletion in Doubly Linked List

## c. Deletion at a Specific Position

Step-by-step approach:

- Traverse to the node at the specified position, say curr.
- If the position is valid, adjust the pointers to skip the node to be deleted.
- If curr is not the head of the linked list, update the next pointer of the node before curr to point to the node after curr, curr->prev->next = curr-next.
- If curr is not the last node of the linked list, update the previous pointer of the node after curr to the node before curr, curr->next->prev = curr->prev.
- Free the memory allocated for the deleted node.


## Deletion Function

```
struct Node* delPos(struct Node* head, int pos) {

    if (head == NULL)        // If the list is empty
        return head;

    struct Node* curr = head;

    // Traverse to the node at the given position
    for (int i = 1; curr != NULL && i < pos; ++i) {
        curr = curr->next;
    }

    // If the position is out of range
    if (curr == NULL)
        return head;

    // Update the previous node's next pointer
    if (curr->prev != NULL)
        curr->prev->next = curr->next;

    // Update the next node's prev pointer
    if (curr->next != NULL)
        curr->next->prev = curr->prev;

    // If the node to be deleted is the head node
    if (head == curr)
        head = curr->next;

    // Deallocate memory for the deleted node
    free(curr);
    return head;
}
```

Time Complexity: O(n)

Space Complexity : O(1)

# Introduction to Circular Linked List


A circular linked list is a data structure where the last node points back to the first node, forming a loop.

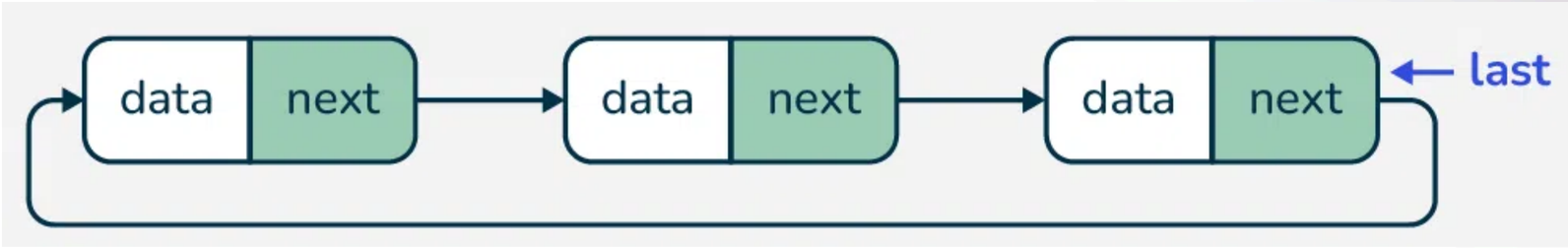
Unlike a regular list that ends with NULL, it has no end—so you can keep traversing it in a loop.

- It allows continuous traversal without stopping.
- It's useful for tasks like scheduling and playlists.

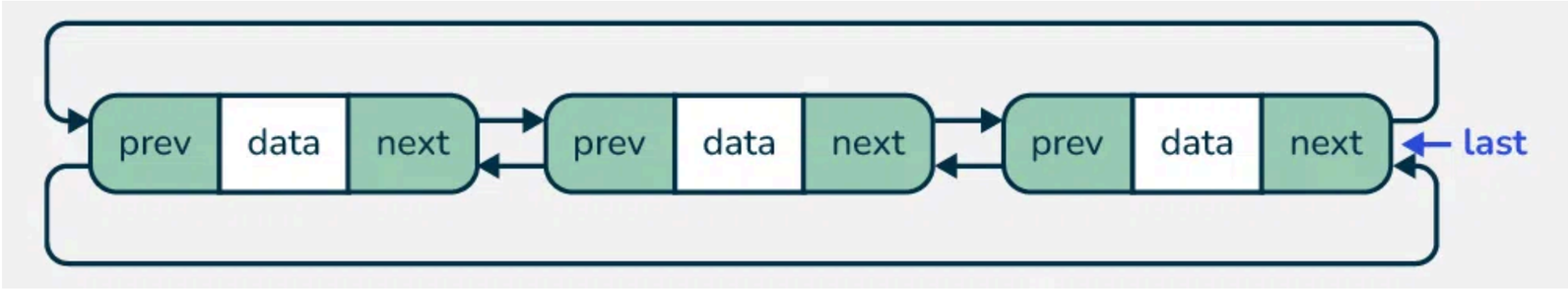
# Types of Circular Linked Lists

We can create a circular linked list from both Singly Linked List and Doubly Linked List.
So, circular linked lists are basically of two types:

1. Circular Singly Linked List




2. Circular Doubly Linked List



# Operations on the Circular Linked list

We can do some operations on the circular linked list similar to the singly and doubly linked list which are:

1. Searching

2. Insertion
   - Insertion at the beginning
   - Insertion at the end
   - Insertion at the given position

3. Deletion
   - Delete the first node
   - Delete the last node
   - Delete the node from any position

# Advantages & Disadvantages Circular Linked list

Advantages of Circular Linked Lists:

- No NULL at the end—easier and safer to traverse.
- Can start from any node and loop back to it.
- Ideal for circular queues and looping tasks.
- Though it lacks a direct "previous" link, the previous node can be found by traversal.

Disadvantages of Circular Linked Lists:

- More complex to implement than singly linked lists.
- Can cause infinite loops if not handled properly.
- Harder to debug due to the circular structure.

# Applications of Circular Linked Lists

Applications of Circular Linked Lists:

- Used in Round-Robin scheduling for time-sharing.
- In games, to switch turns between players in a loop.
- For buffers in streaming, where data flows continuously.
- In media players, to loop through playlists.
- In browsers, to manage history using the back button.

# Applications of Singly Linked List

Stacks and Queues

- Efficient for implementing stack (LIFO) and queue (FIFO) structures.

Polynomial Operations

- Represent and add polynomials using linked nodes.

Adjacency Lists in Graphs

- Used to store neighbors of a node.

Dynamic Memory Allocation

- Helps manage free memory blocks.

String Manipulation

- Useful in managing characters and substrings.

# Applications of Doubly Linked List

Navigation Systems

- Move forward and backward (e.g., browser history).

Music and Video Playlists

- Easily go to next or previous media file.

Undo/Redo Functionality

- In editors, go back or forward through changes.

MRU/LRU Cache Implementation

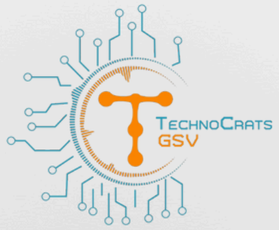
- Efficient access and update of recently used data.

Complex Data Structures

- Used as the base for Deques, Fibonacci Heaps, etc.

# Conclusion

A linked list is a way to store similar data items efficiently. It is the base for other structures like stacks and queues. With different types of linked lists, inserting and deleting nodes becomes easy. Also, it uses memory wisely, as its size can grow or shrink during program execution.




# Thank You For Your Attention