UNIT 3
LINKED LIST

What is a Linked List?

A linked list is a linear data structure that consists of a series of nodes connected by pointers. Each node contains data and a reference to the next node in the list. Unlike arrays, linked lists allow for efficient insertion or removal of elements from any position in the list, as the nodes are not stored contiguously in memory.

Representation of a Linked list

Linked list can be represented as the connection of nodes in which each node points to the next node of the list.

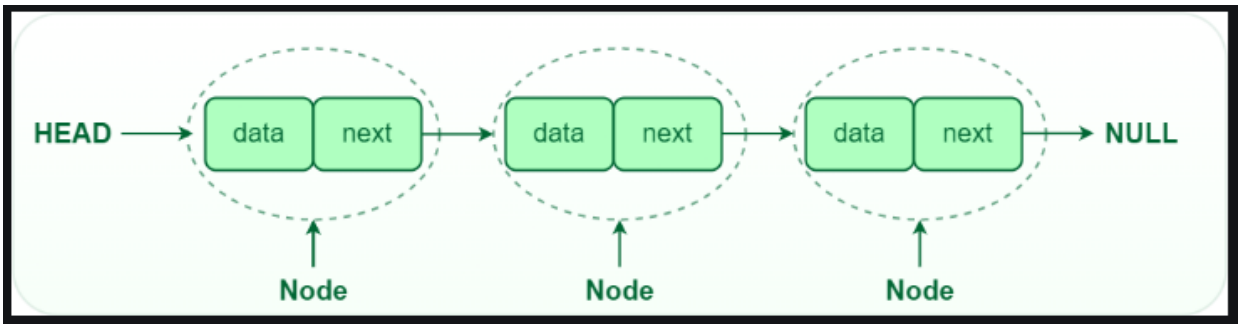
Node Structure: A node in a linked list typically consists of two components:

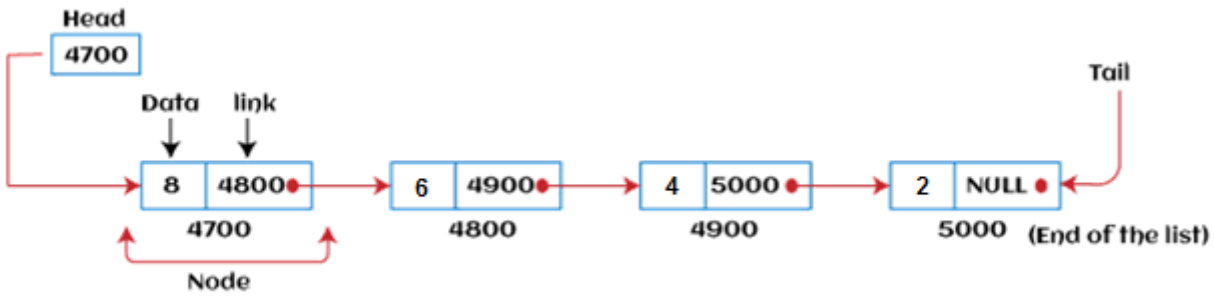
Data: It holds the actual value or data associated with the node.

Next Pointer: It stores the memory address (reference) of the next node in the sequence.

Head and Tail: The linked list is accessed through the head node, which points to the first node in the list. The last node in the list points to NULL or nullptr, indicating the end of the list. This node is known as the tail node.

The representation of the linked list is shown below -





Types of linked lists:

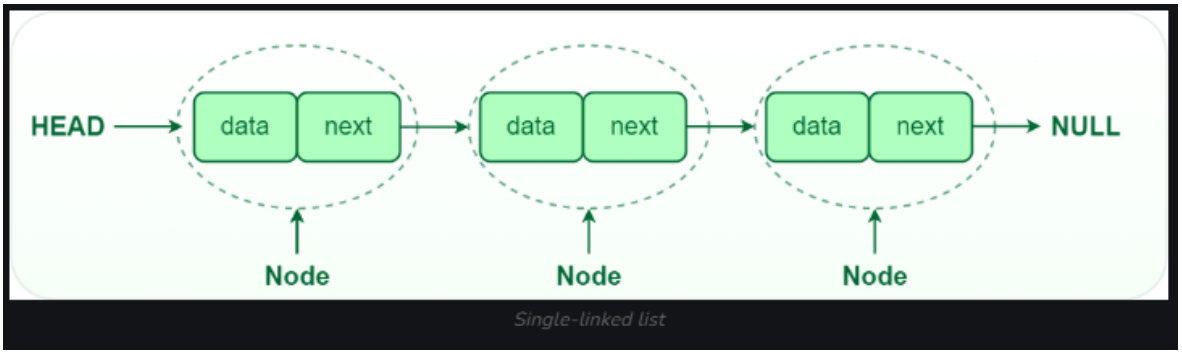
There are mainly three types of linked lists:

1) Single-linked list
2) Double linked list

3) Circular linked list

1. Single-linked list:

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

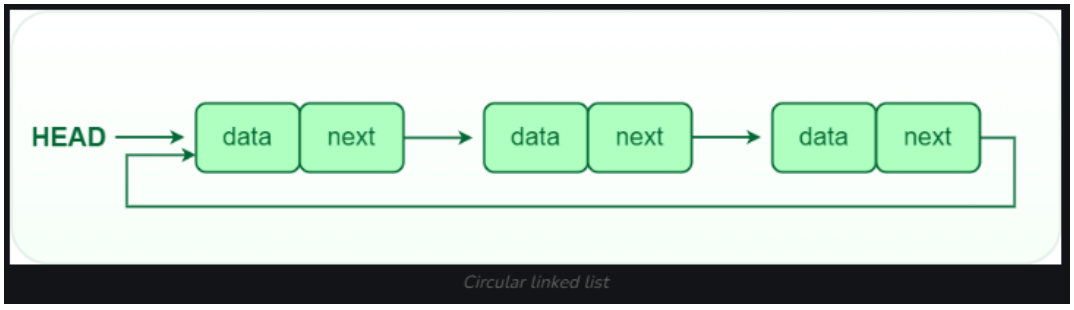


2. Double-linked list:

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

3. Circular linked list:

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



How Can We Declare a Linked List?

To declare a linked list in C, you need to define a node structure and then create functions to manipulate the list. Here's a step-by-step guide:

1. Define a Node Structure

Each node in a linked list contains data and a pointer to the next node.

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

Operations on Linked Lists

- **Insertion:** Adding a new node to a linked list involves adjusting the pointers of the existing nodes to maintain the proper sequence. Insertion can be performed at the beginning, end, or any position within the list

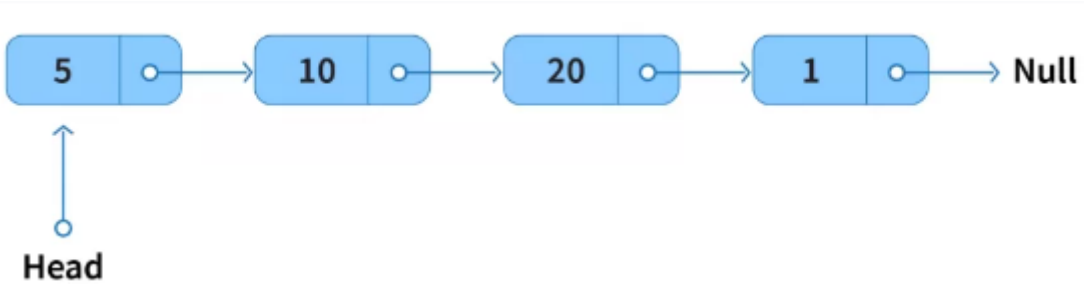
- **Deletion:** Removing a node from a linked list requires adjusting the pointers of the neighboring nodes to bridge the gap left by the deleted node. Deletion can be performed at the beginning, end, or any position within the list.

- **Searching:** Searching for a specific value in a linked list involves traversing the list from the head node until the value is found or the end of the list is reached.

- **Traversal:** We can traverse the entire linked list starting from the head node. If there are n nodes then the time complexity for traversal becomes O(n) as we hop through each and every node.

Single Linked List:

- Singly linked lists in C are the simplest type of linked list.
- Each node in a singly linked list contains a **data field** and a **pointer** to the next node in the list.
- The last node in the list points to null, indicating the end of the list.



Operations on Singly Linked List

1. Insertion

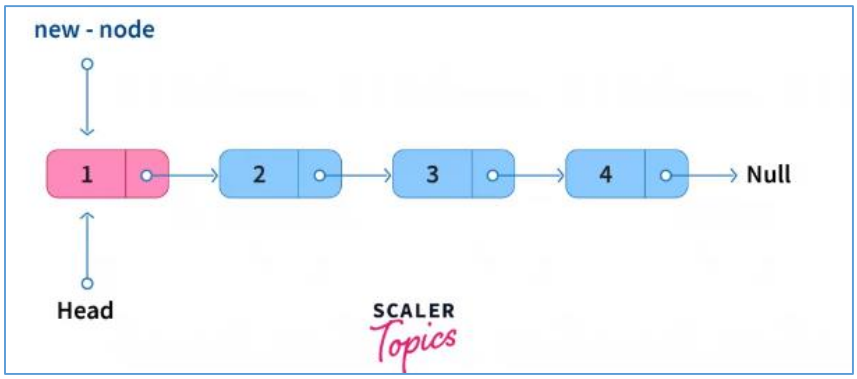
Insertion in a singly linked list can be performed in the following ways,

- **Insertion at the start**

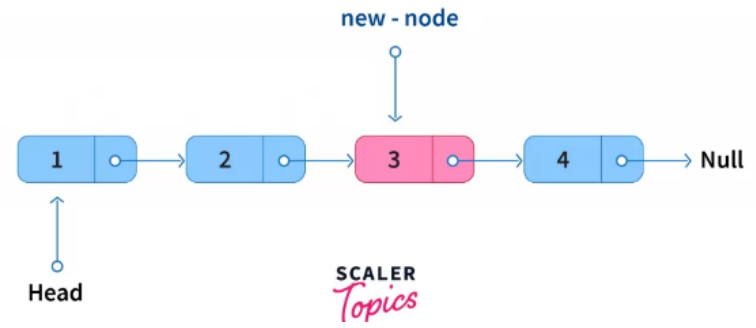
  Insertion of a new node at the start of a singly linked list is carried out in the following manner.

  - Make the new node point to HEAD.
  - Make the HEAD point to the new node.

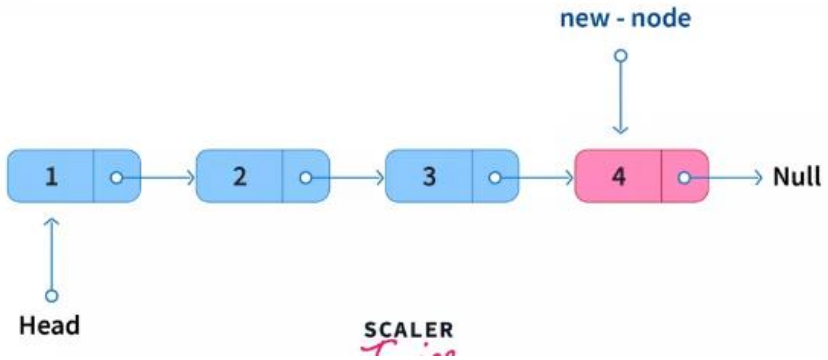
Inserting a new node at the start is an O(1) operation.



- **Insertion after some Node** Insertion of a new node after some node in a singly linked list is carried out in the following manner,
  - Reach the desired node after which the new node is to be inserted.
  - Make the new node point to the next element of the current node.
  - Make the current node point to the new node. Inserting a new node after some node is an O(N) operation.



- **Insertion at the end** Insertion of a new node at the end of a singly linked list isperformed in te followin way,
  - Taverse the list from start and reach the last node.
  - Make the last node point to the new node.
  - Make the new node point to null, marking the end of the list. Inserting a new node at the end is an O(N) operation.



2. Deletion

Deletion in a singly linked list can be performed in the following ways,

- **Deletion at the start**
  The first node of the singly linked list can be deleted as follows,
  - Make the HEAD point to its next element.

Deleting the first node of a singly linked list is an O(1) operation.

- **Deletion at the middle**

The deletion after a specific node can be formed in the following way,

- Reach the desired node after which the node is to be deleted.
- Make the current node point to the next of next element.

Deleting a node after a specific node is an O(N) operation.

- **Deletion at last**

The deletion of the last node is performed in the following manner,

- Reach the second last node of th singly linke list.
- Make the second last node point null.

Deleting the last node is an O(N) operation.

3. Display

To display the entire singly linked list, we need to traverse it from first to last.

In contrast to arrays, linked list nodes cannot be accessed randomly. Hence to reach the n-th element, we are bound to traverse through all (n-1) elements.

Since the entire linked list is traversed, the operation costs us O(N) time complexity. The following JAVA snippet shows how the entire singly linked list can be displayed.

4. Search

To search an element in the singly linked list, we need to traverse the linked list right from the start.

At each node, we perform a lookup to determine if the target has been found, if yes, then we return the target node else we move to the next element.

In the worst case, we could end up visiting all the nodes in the list and hence searching an element in the singly linked list cost us O(N) operational time.

Program:

```
#include <stdio.h>
#include <stdlib.h>

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

// Function to insert a node at the end
void insertAtEnd(struct Node** head, int data)
{
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   if (!newNode)
   {
      printf("Memory allocation error\n");
      exit(1);
   }
   newNode->data = data;
   newNode->next = NULL;

   if (*head == NULL)
   {
      *head = newNode;
      return;
   }

   struct Node* temp = *head;
   while (temp->next != NULL)
   {
      temp = temp->next;
```

```
    }
    temp->next = newNode;
}

// Function to insert a node at the front
void insertAtFront(struct Node** head, int data)
{
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode)
    {
        printf("Memory allocation error\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = *head;
    *head = newNode;
}

// Function to insert a node at a given position
void insertAtPosition(struct Node** head, int data, int position)
{
    if (position < 1)
    {
        printf("Invalid position\n");
        return;
    }

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode)
    {
        printf("Memory allocation error\n");
        exit(1);
    }
    newNode->data = data;

    if (position == 1)
    {
        newNode->next = *head;
        *head = newNode;
        return;
    }

    struct Node* temp = *head;
    for (int i = 1; i < position - 1 && temp != NULL; i++)
```

```
    {
        temp = temp->next;
    }

    if (temp == NULL)
    {
        printf("Position out of range\n");
        free(newNode);
    }
    else
    {
        newNode->next = temp->next;
        temp->next = newNode;
    }
}

// Function to delete a node from the front
void deleteFromFront(struct Node** head)
{
    if (*head == NULL)
    {
        printf("List is empty\n");
        return;
    }

    struct Node* temp = *head;
    *head = temp->next;
    free(temp);
}

// Function to delete a node from the end
void deleteFromEnd(struct Node** head)
{
    if (*head == NULL)
    {
        printf("List is empty\n");
        return;
    }

    if ((*head)->next == NULL)
    {
        free(*head);
        *head = NULL;
        return;
```

```
    }

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

    free(temp->next);
    temp->next = NULL;
}

// Function to delete a node from a given position
void deleteAtPosition(struct Node** head, int position)
{
    if (*head == NULL)
    {
        printf("List is empty\n");
        return;
    }

    if (position < 1)
    {
        printf("Invalid position\n");
        return;
    }

    struct Node* temp = *head;

    if (position == 1)
    {
        *head = temp->next;
        free(temp);
        return;
    }

    struct Node* prev = NULL;
    for (int i = 1; i < position && temp != NULL; i++)
    {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL)
```

```
    {
        printf("Position out of range\n");
    }
    else
    {
        prev->next = temp->next;
        free(temp);
    }
}

// Function to search for a node in the list
int search(struct Node* head, int data)
{
    struct Node* temp = head;
    int position = 1;

    while (temp != NULL)
    {
        if (temp->data == data)
        {
            return position;
        }
        temp = temp->next;
        position++;
    }
    return -1; // Data not found
}

// Function to display the list
void displayList(struct Node* head)
{
    struct Node* temp = head;
    while (temp != NULL)
    {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main()
{
    struct Node* head = NULL;
    int choice, data, position;
```

```
while (1)
{
    printf("\nMenu:\n");
    printf("1. Insert at End\n");
    printf("2. Insert at Front\n");
    printf("3. Insert at Position\n");
    printf("4. Delete from Front\n");
    printf("5. Delete from End\n");
    printf("6. Delete from Position\n");
    printf("7. Display List\n");
    printf("8. Search for an Element\n");
    printf("9. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice)
    {
        case 1:
            printf("Enter the data to insert at end: ");
            scanf("%d", &data);
            insertAtEnd(&head, data);
            break;
        case 2:
            printf("Enter the data to insert at front: ");
            scanf("%d", &data);
            insertAtFront(&head, data);
            break;
        case 3:
            printf("Enter the data to insert: ");
            scanf("%d", &data);
            printf("Enter the position: ");
            scanf("%d", &position);
            insertAtPosition(&head, data, position);
            break;
        case 4:
            deleteFromFront(&head);
            break;
        case 5:
            deleteFromEnd(&head);
            break;
        case 6:
            printf("Enter the position to delete: ");
            scanf("%d", &position);
```

```
            deleteAtPosition(&head, position);
            break;
        case 7:
            displayList(head);
            break;
        case 8:
            printf("Enter the element to search for: ");
            scanf("%d", &data);
            position = search(head, data);
            if (position != -1)
            {
                printf("Element %d found at position %d\n", data, position);
            }
            else
            {
                printf("Element %d not found in the list\n", data);
            }
            break;
        case 9:
            printf("Exiting...\n");
            exit(0);
        default:
            printf("Invalid choice! Please try again.\n");
    }
  }

  return 0;
}
```

---

2. Doubly Linked Lists.

- Doubly linked lists in C are more complex to implement than singly linked lists, but they are more efficient for certain operations, such as insertion, deletion, and traversal in both directions.
- Each node in a doubly linked list contains a data field, a pointer to the next node in the list, and a pointer to the previous node in the list.
- Example: undo/redo history, doubly linked list implementation of a binary tree

A doubly linked list is a type of linked list in which each node consists of 3 components:

`*prev` - address of the previous node
`data` - data item
`*next` - address of next node

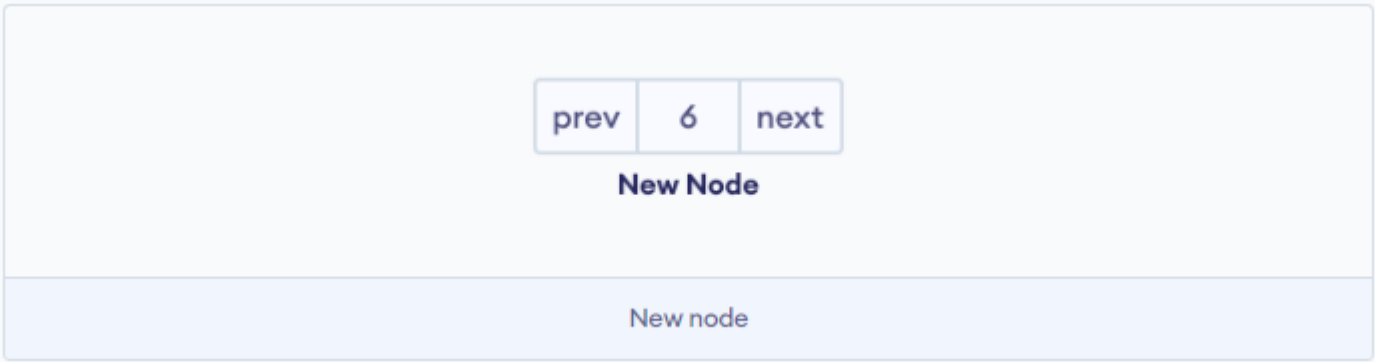
Insertion Operation In Doubly Linked List

1. Insertion at the Beginning

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`.

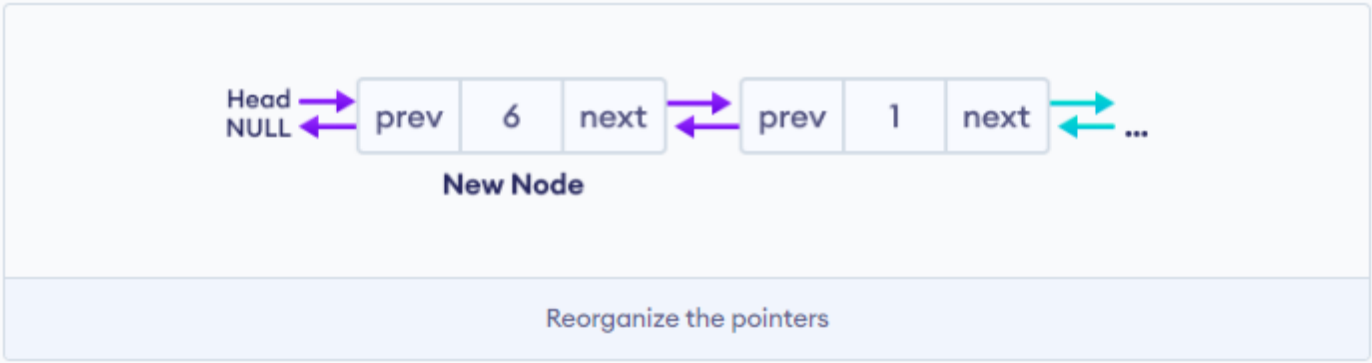


2. Set prev and next pointers of new node

- 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`

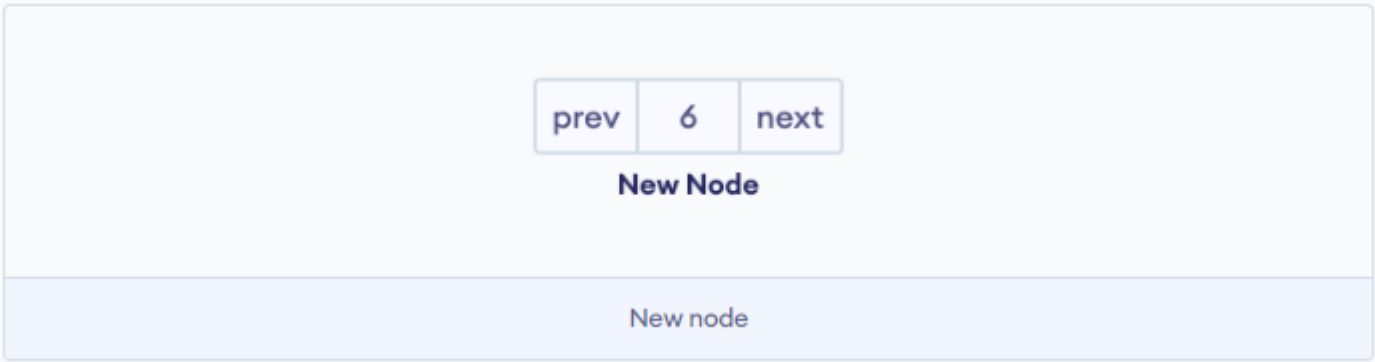


2. Insertion in between two nodes

Let's add a node with value 6 after node with value 1 in the doubly linked list.

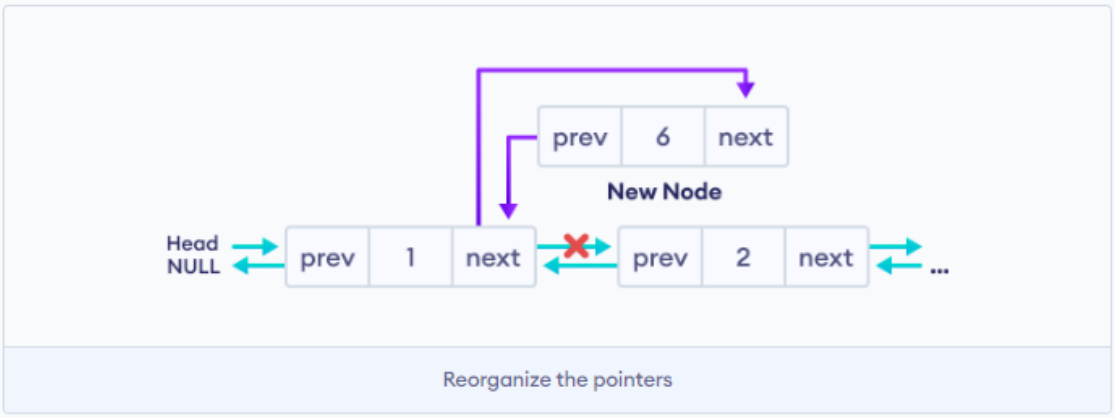
1. Create a new node

- allocate memory for `newNode`
- assign the data to `newNode`.



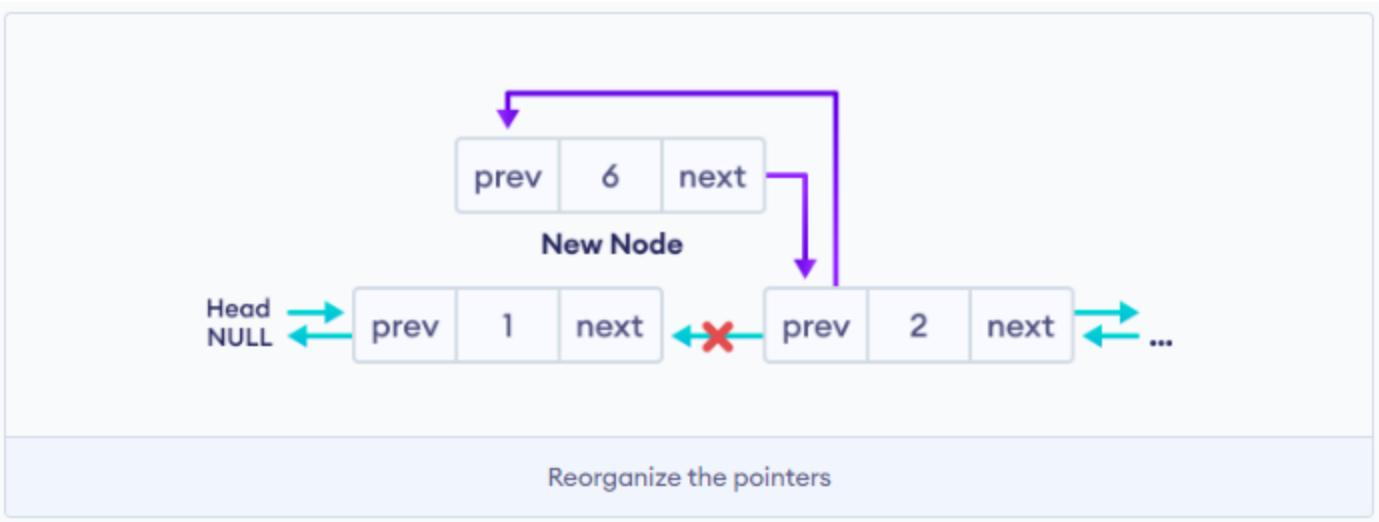
2. Set the next pointer of new node and previous node

- assign the value of `next` from previous node to the `next` of `newNode`
- assign the address of `newNode` to the `next` of previous node

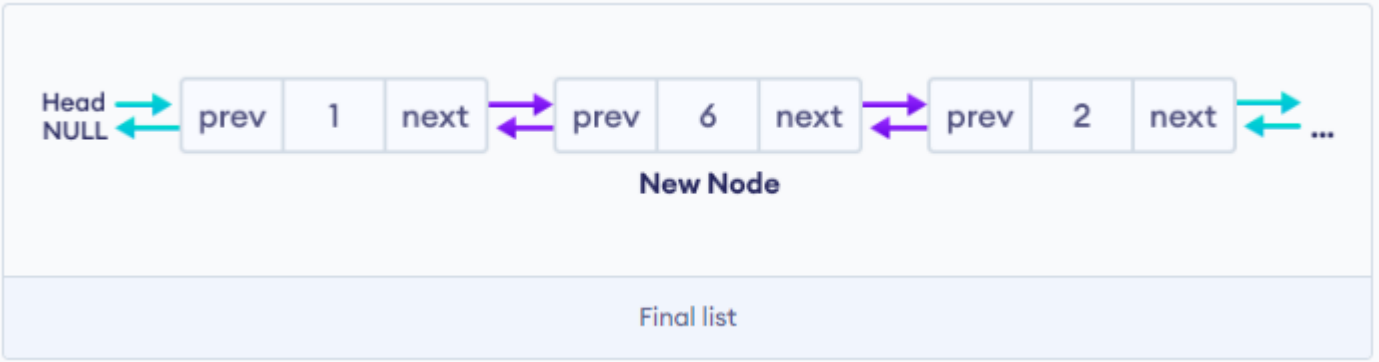


3. Set the prev pointer of new node and the next node

- assign the value of `prev` of next node to the `prev` of `newNode`
- assign the address of `newNode` to the `prev` of next node



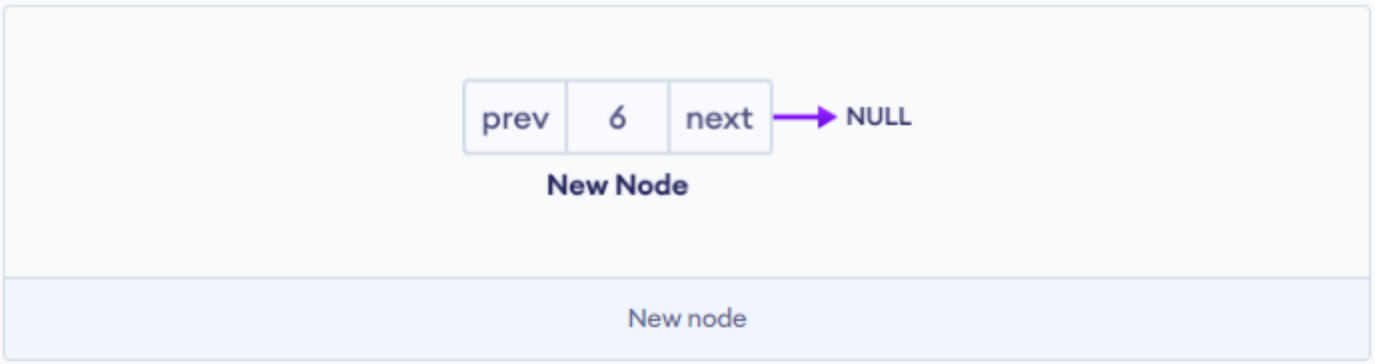
The final doubly linked list is after this insertion is:



3. Insertion at the End

Let's add a node with value 6 at the end of the doubly linked list.

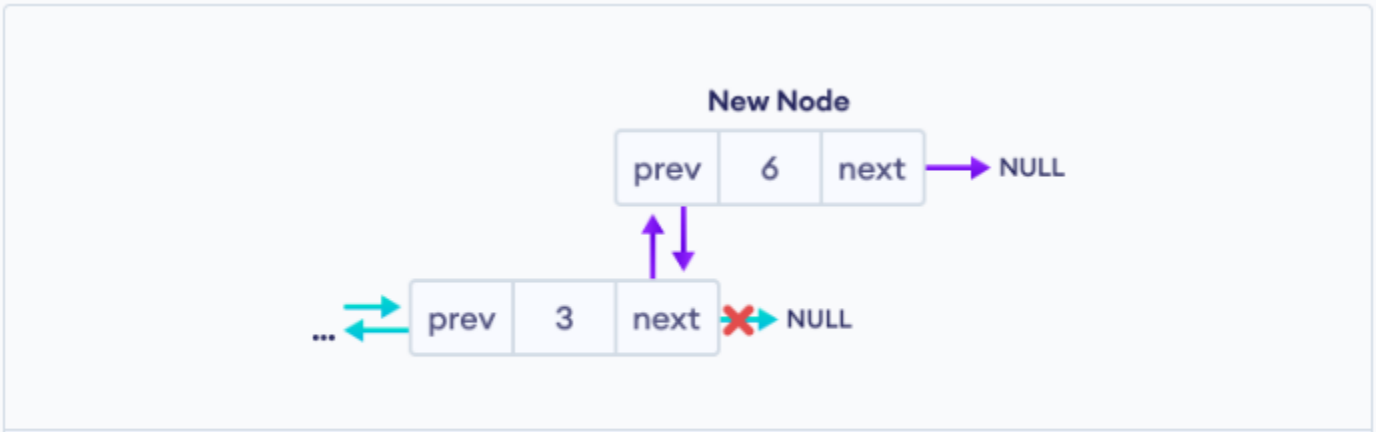
1. Create a new node



New node

2. Set prev and next pointers of new node and the previous node

If the linked list is empty, make the `newNode` as the head node. Otherwise, traverse to the end of the doubly linked list and

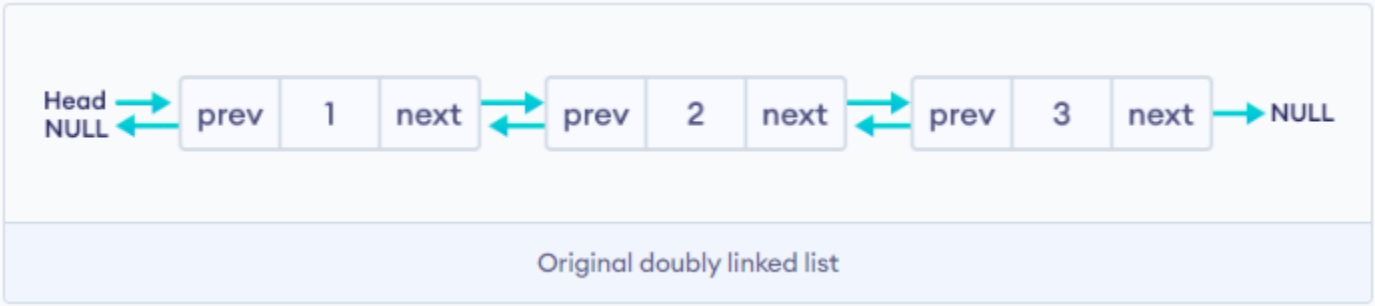


Reorganize the pointers

Deletion from a Doubly Linked List

Similar to insertion, we can also delete a node from **3** different positions of a doubly linked list.

Suppose we have a double-linked list with elements **1**, **2**, and **3**.

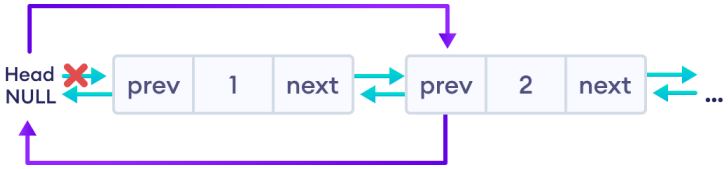


Original doubly linked list

1. Delete the First Node of Doubly Linked List

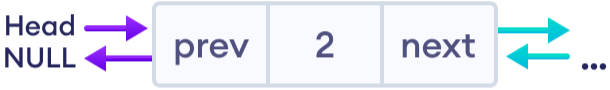
If the node to be deleted (i.e. `del_node`) is at the beginning

Reset value node after the del_node (i.e. node two)



Reorganize the pointers

Finally, free the memory of `del_node`. And, the linked will look like this



Free the space of the first node

Final list

2. Deletion of the Inner Node

If `del_node` is an inner node (second node), we must have to reset the value of `next` and `prev` of the nodes before and after the `del_node`.

For the node before the del_node (i.e. first node)

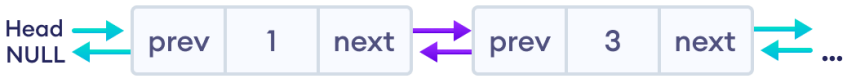
Assign the value of `next` of `del_node` to the `next` of the `first` node.

For the node after the del_node (i.e. third node)

Assign the value of `prev` of `del_node` to the `prev` of the `third` node.

Reorganize the pointers

Finally, we will free the memory of `del_node`. And, the final doubly linked list looks like this.

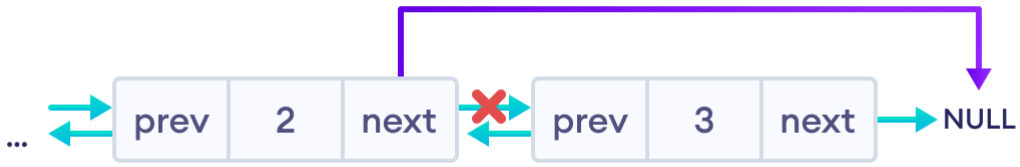


Final list

3. Delete the Last Node of Doubly Linked List

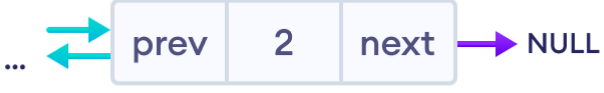
In this case, we are deleting the last node with value **3** of the doubly linked list.

Here, we can simply delete the `del_node` and make the `next` of node before `del_node` point to `NULL`.



Reorganize the pointers

The final doubly linked list looks like this.



Final list

Program:

```
#include <stdio.h>
#include <stdlib.h>

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

// Function to insert a node at the end
void insertAtEnd(struct Node** head, int data)
{
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   if (!newNode)
   {
      printf("Memory allocation error\n");
      exit(1);
   }
   newNode->data = data;
   newNode->next = NULL;
   newNode->prev = NULL;

   if (*head == NULL)
   {
      *head = newNode;
      return;
   }

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

// Function to insert a node at the front
void insertAtFront(struct Node** head, int data)
{
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
    if (!newNode)
    {
        printf("Memory allocation error\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = *head;
    newNode->prev = NULL;

    if (*head != NULL)
    {
        (*head)->prev = newNode;
    }
    *head = newNode;
}

// Function to insert a node at a given position
void insertAtPosition(struct Node** head, int data, int position)
{
    if (position < 1)
    {
        printf("Invalid position\n");
        return;
    }

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode)
    {
        printf("Memory allocation error\n");
        exit(1);
    }
    newNode->data = data;

    if (position == 1)
    {
        newNode->next = *head;
        newNode->prev = NULL;
        if (*head != NULL)
        {
            (*head)->prev = newNode;
        }
        *head = newNode;
        return;
    }
```

```
    struct Node* temp = *head;
    for (int i = 1; i < position - 1 && temp != NULL; i++)
    {
        temp = temp->next;
    }

    if (temp == NULL)
    {
        printf("Position out of range\n");
        free(newNode);
    }
    else
    {
        newNode->next = temp->next;
        newNode->prev = temp;
        if (temp->next != NULL)
        {
            temp->next->prev = newNode;
        }
        temp->next = newNode;
    }
}

// Function to delete a node from the front
void deleteFromFront(struct Node** head)
{
    if (*head == NULL)
    {
        printf("List is empty\n");
        return;
    }

    struct Node* temp = *head;
    *head = temp->next;

    if (*head != NULL)
    {
        (*head)->prev = NULL;
    }
    free(temp);
}

// Function to delete a node from the end
```

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

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

    if (temp->prev != NULL)
    {
        temp->prev->next = NULL;
    }
    else
    {
        *head = NULL;
    }
    free(temp);
}

// Function to delete a node from a given position
void deleteAtPosition(struct Node** head, int position)
{
    if (*head == NULL)
    {
        printf("List is empty\n");
        return;
    }

    if (position < 1)
    {
        printf("Invalid position\n");
        return;
    }

    struct Node* temp = *head;

    if (position == 1)
    {
```

```
        *head = temp->next;
        if (*head != NULL)
        {
            (*head)->prev = NULL;
        }
        free(temp);
        return;
    }

    for (int i = 1; i < position && temp != NULL; i++)
    {
        temp = temp->next;
    }

    if (temp == NULL)
    {
        printf("Position out of range\n");
    }
    else
    {
        if (temp->prev != NULL)
        {
            temp->prev->next = temp->next;
        }
        if (temp->next != NULL)
        {
            temp->next->prev = temp->prev;
        }
        free(temp);
    }
}

// Function to search for a node in the list
int search(struct Node* head, int data)
{
    struct Node* temp = head;
    int position = 1;

    while (temp != NULL)
    {
        if (temp->data == data)
        {
            return position;
        }
```

```
        temp = temp->next;
        position++;
    }
    return -1; // Data not found
}

// Function to display the list
void displayList(struct Node* head)
{
    struct Node* temp = head;
    while (temp != NULL)
    {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main()
{
    struct Node* head = NULL;
    int choice, data, position;

    while (1)
    {
        printf("\nMenu:\n");
        printf("1. Insert at End\n");
        printf("2. Insert at Front\n");
        printf("3. Insert at Position\n");
        printf("4. Delete from Front\n");
        printf("5. Delete from End\n");
        printf("6. Delete from Position\n");
        printf("7. Display List\n");
        printf("8. Search for an Element\n");
        printf("9. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
            case 1:
                printf("Enter the data to insert at end: ");
                scanf("%d", &data);
                insertAtEnd(&head, data);
```

```
            break;
        case 2:
            printf("Enter the data to insert at front: ");
            scanf("%d", &data);
            insertAtFront(&head, data);
            break;
        case 3:
            printf("Enter the data to insert: ");
            scanf("%d", &data);
            printf("Enter the position: ");
            scanf("%d", &position);
            insertAtPosition(&head, data, position);
            break;
        case 4:
            deleteFromFront(&head);
            break;
        case 5:
            deleteFromEnd(&head);
            break;
        case 6:
            printf("Enter the position to delete: ");
            scanf("%d", &position);
            deleteAtPosition(&head, position);
            break;
        case 7:
            displayList(head);
            break;
        case 8:
            printf("Enter the element to search for: ");
            scanf("%d", &data);
            position = search(head, data);
            if (position != -1)
            {
                printf("Element %d found at position %d\n", data, position);
            }
            else
            {
                printf("Element %d not found in the list\n", data);
            }
            break;
        case 9:
            printf("Exiting...\n");
            exit(0);
        default:
```

```
            printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

---

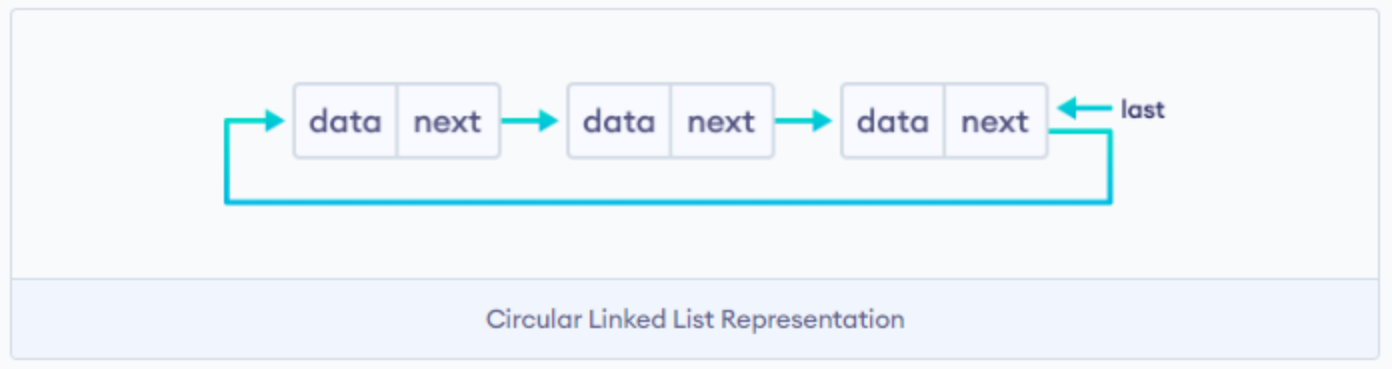
3. Circular Linked Lists

- Circular linked lists are the most complex type of linked list to implement, but they can be very efficient for certain operations, such as traversal and queueing.
- In a circular linked list, the last node in the list points back to the first node in the list, forming a loop.
- Example: Circular buffer, circular queue, circular linked list implementation of a hash table.

There are basically two types of circular linked list:

1. Circular Singly Linked List

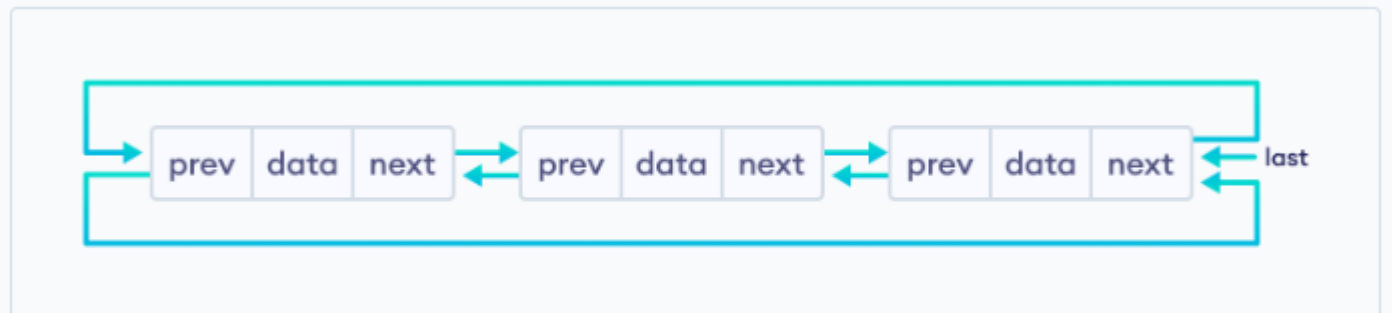
Here, the address of the last node consists of the address of the first node.



Circular Linked List Representation

2. Circular Doubly Linked List

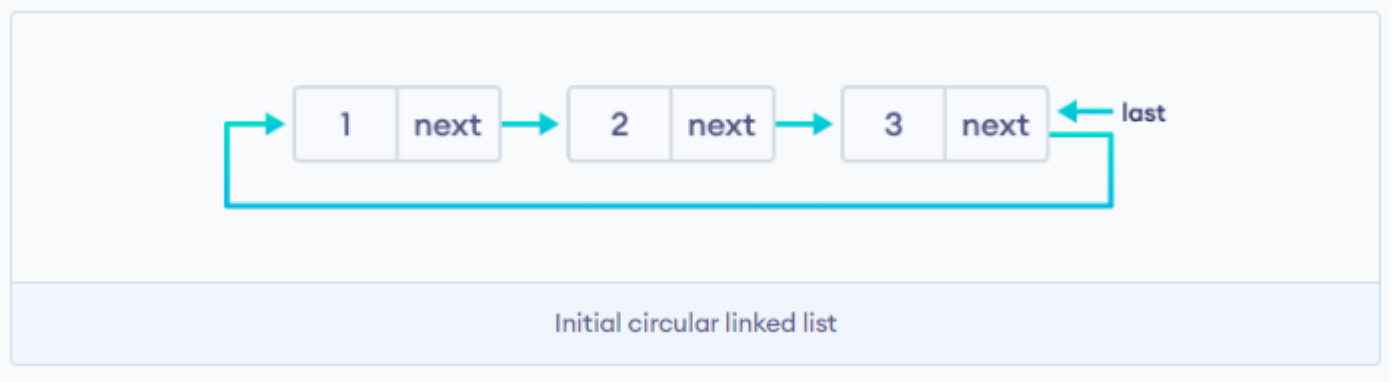
Here, in addition to the last node storing the address of the first node, the first node will also store the address of the last node.



Circular Doubly Linked List Representation

Representation of Circular Linked List

Let's see how we can represent a circular linked list on an algorithm/code. Suppose we have a linked list:



Initial circular linked list

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

Insertion on a Circular Linked List

We can insert elements at 3 different positions of a circular linked list:

1. Insertion at the beginning
2. Insertion in-between nodes
3. Insertion at the end

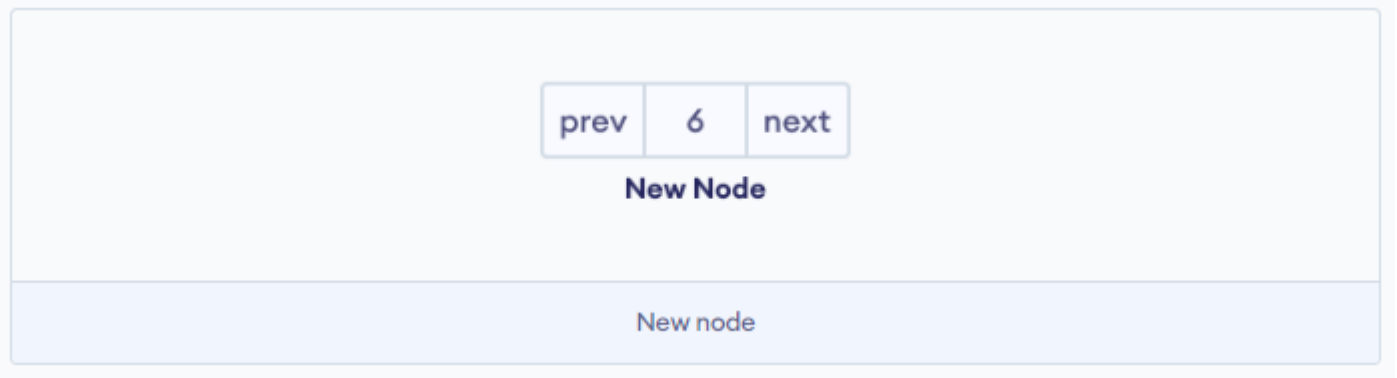
Suppose we have a circular linked list with elements 1, 2, and 3.

Initial circular linked list

Let's add a node with value 6 at different positions of the circular linked list we made above.

The first step is to create a new node.

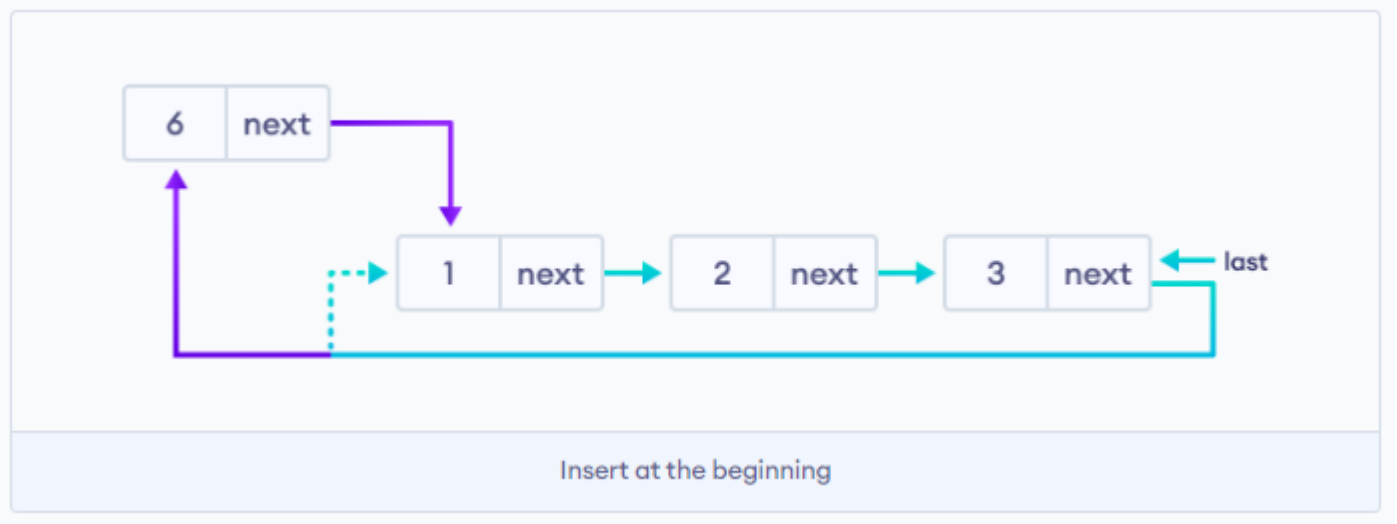
- allocate memory for `newNode`
- assign the data to `newNode`



New node

1. Insertion at the Beginning

- store the address of the current first node in the `newNode` (i.e. pointing the `newNode` to the current first node)
- point the last node to `newNode` (i.e making `newNode` as head)

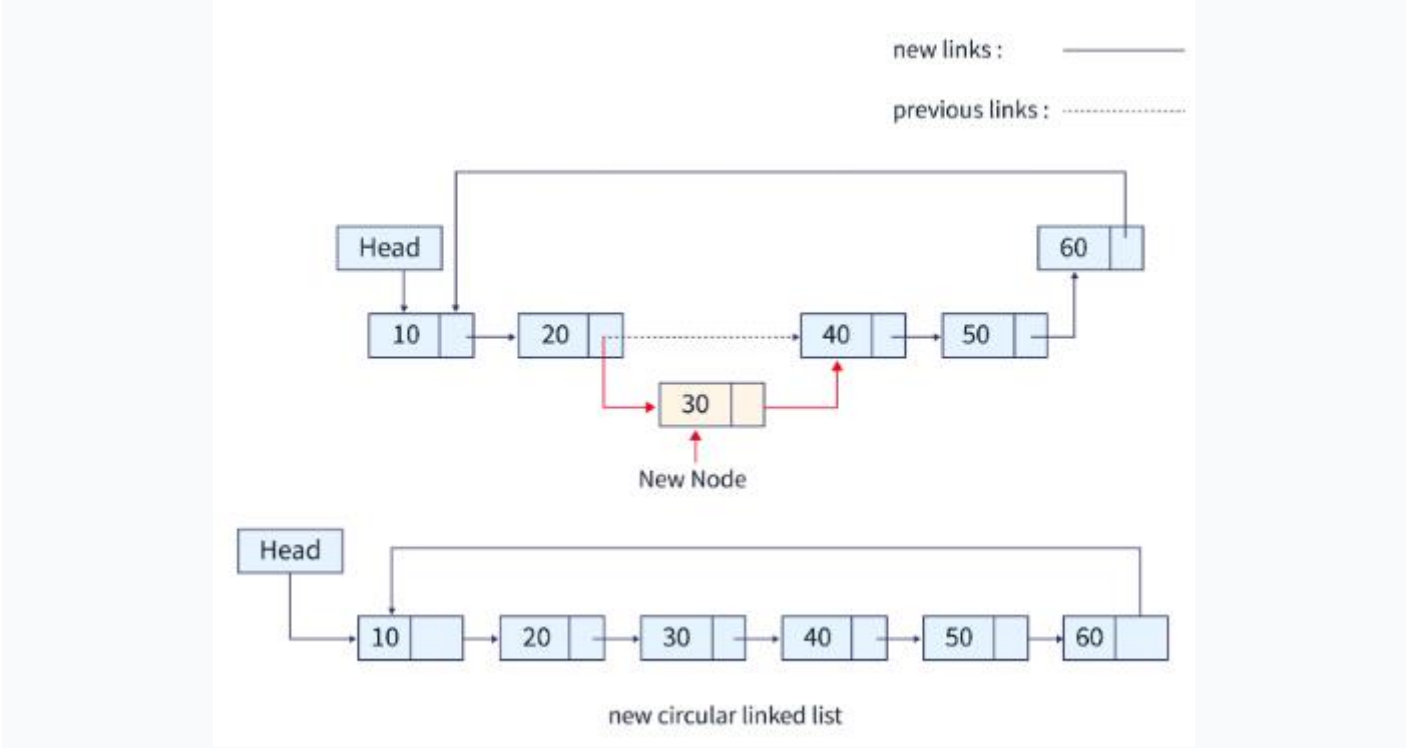


Insert at the beginning

2. Insertion of a Node After a Specific Element

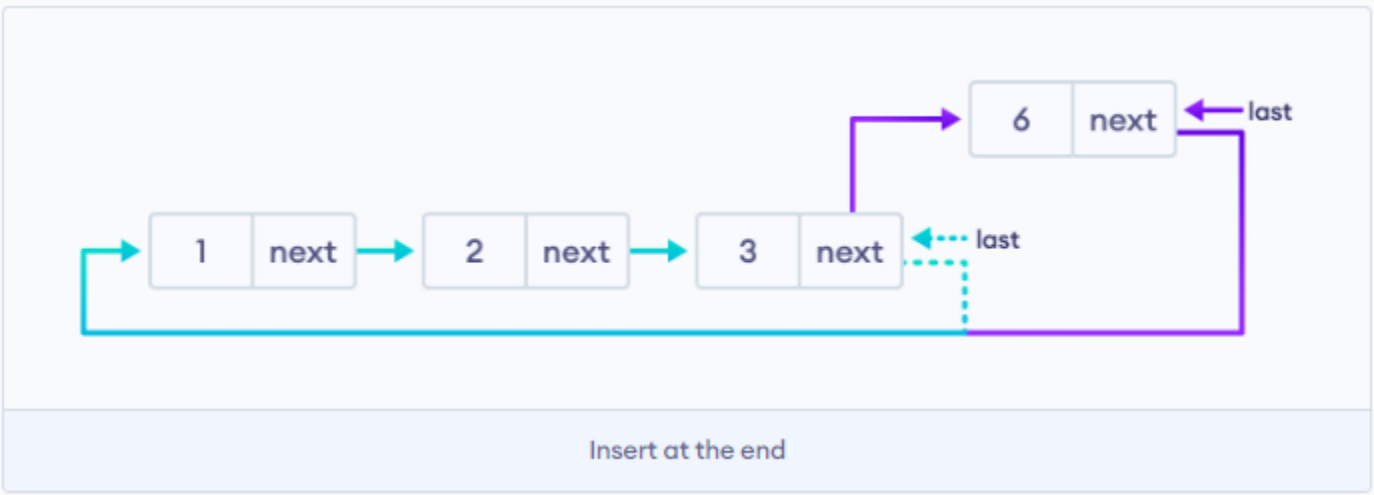
We can insert a node after a specific element in the circular linked list using the below procedure:

1. Create a new node and assign it to a new node variable.
2. Assign the head node address in a new temp variable, and traverse the list till the next node contains the specified value.
3. Assign the next node's address in the new node variable.
4. Update the current node's next block address to point to the new node.



3. Insertion at the end

- store the address of the head node to `next` of newNode (making `newNode` the last node)
- point the current last node to `newNode`
- make `newNode` as the last node



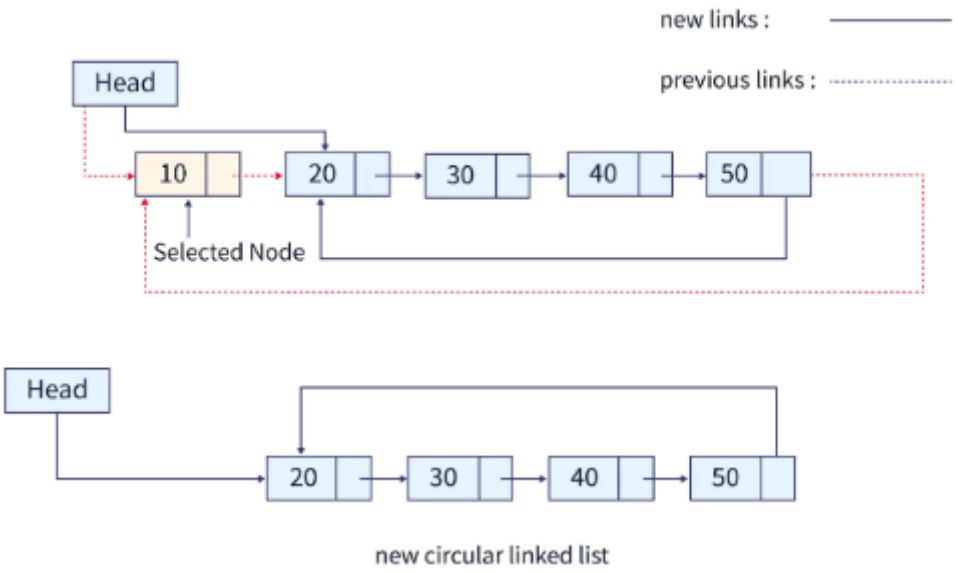
Deleting the First Node

We can delete the first node of a circular linked list in C using the below procedure:

1. If the list has only one node, then delete the head node by using the free() function directly.
2. Create two temporary variables and store the head address in them.

3. Traverse the list to the last node and assign the head's next node address to the last node.
4. Update the head pointer to point to the next node.
5. free the temporary variable which still points to the first node.

Diagrammatic Representation:

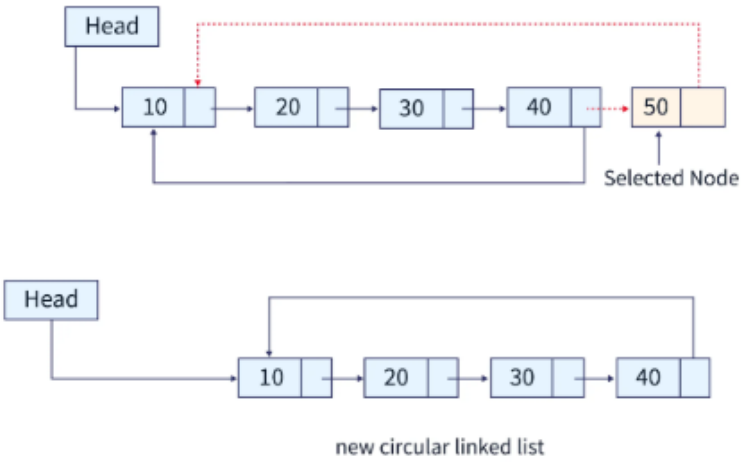


Deleting the Last Node

We can delete the last node of a circular linked list using the below procedure:

1. If the list has only one node, then delete the head node by using the free() function directly.
2. Create two temporary variables and store the head address in one of them.
3. Traverse the list till the second last node and assign the last node address in a temporary node variable.
4. Update the second last node pointer to point to the head node.
5. free the temporary variable which still points to the last node.

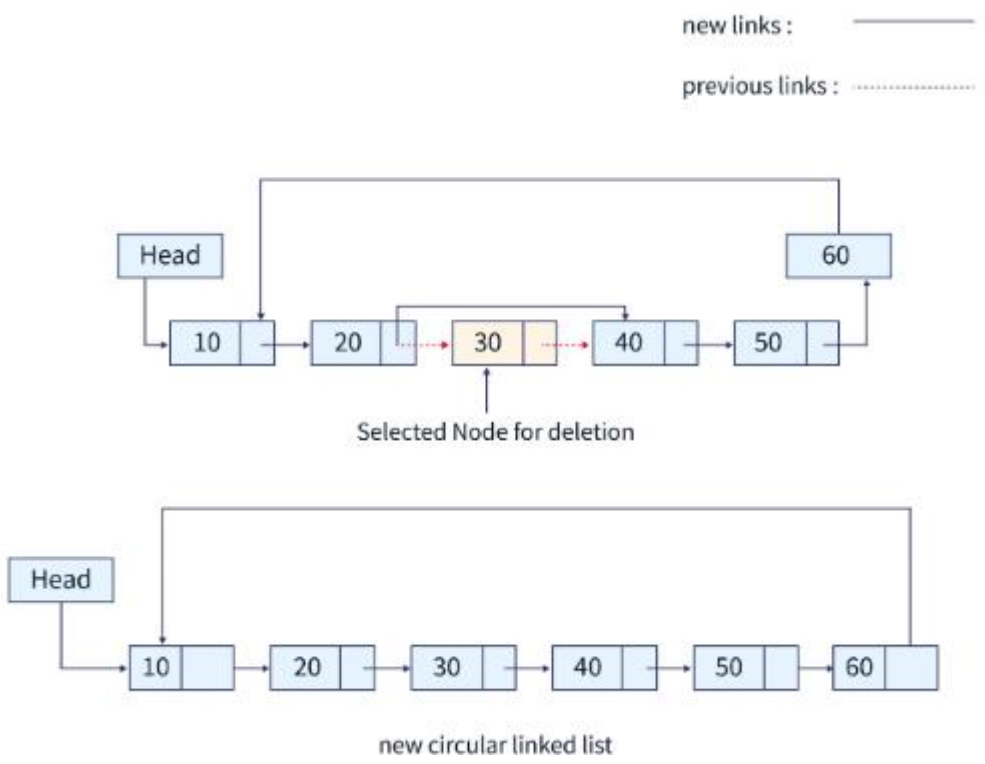
Diagrammatic Representation:



Deleting a Node at a Given Point

We can delete the last node of a circular linked list using the below procedure:

1. Check if the position is first, then apply the algorithm for deleting the first node from a circular linked list (mentioned above).
2. Create two temporary variables and store the head address in one of them.
3. Traverse the list till the previous node to the node to be deleted and assign the position's next node address in a temporary node variable.
4. Update the previous node pointer to point to the current's next node.
5. free the temporary variable which still points to the position node.



Program:

```
#include <stdio.h>
#include <stdlib.h>

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

// Function to insert a node at the end
void insertAtEnd(struct Node** head, int data)
{
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   if (!newNode)
   {
      printf("Memory allocation error\n");
```

```
        exit(1);
    }
    newNode->data = data;
    newNode->next = newNode; // Points to itself initially

    if (*head == NULL)
    {
        *head = newNode;
    }
    else
    {
        struct Node* temp = *head;
        while (temp->next != *head)
        {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = *head;
    }
}

// Function to insert a node at the front
void insertAtFront(struct Node** head, int data)
{
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode)
    {
        printf("Memory allocation error\n");
        exit(1);
    }
    newNode->data = data;

    if (*head == NULL)
    {
        newNode->next = newNode;
        *head = newNode;
    }
    else
    {
        struct Node* temp = *head;
        while (temp->next != *head)
        {
            temp = temp->next;
        }
```

```
        newNode->next = *head;
        temp->next = newNode;
        *head = newNode;
    }
}

// Function to insert a node at a given position
void insertAtPosition(struct Node** head, int data, int position)
{
    if (position < 1)
    {
        printf("Invalid position\n");
        return;
    }

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode)
    {
        printf("Memory allocation error\n");
        exit(1);
    }
    newNode->data = data;

    if (position == 1)
    {
        if (*head == NULL)
        {
            newNode->next = newNode;
            *head = newNode;
        }
        else
        {
            struct Node* temp = *head;
            while (temp->next != *head)
            {
                temp = temp->next;
            }
            newNode->next = *head;
            temp->next = newNode;
            *head = newNode;
        }
    }
    else
    {
```

```
        struct Node* temp = *head;
        for (int i = 1; i < position - 1 && temp->next != *head; i++)
        {
            temp = temp->next;
        }

        if (temp->next == *head && position != 2)
        {
            printf("Position out of range\n");
            free(newNode);
        }
        else
        {
            newNode->next = temp->next;
            temp->next = newNode;
        }
    }
}

// Function to delete a node from the front
void deleteFromFront(struct Node** head)
{
    if (*head == NULL)
    {
        printf("List is empty\n");
        return;
    }

    if ((*head)->next == *head)
    {
        free(*head);
        *head = NULL;
    }
    else
    {
        struct Node* temp = *head;
        while (temp->next != *head)
        {
            temp = temp->next;
        }
        struct Node* toDelete = *head;
        temp->next = (*head)->next;
        *head = (*head)->next;
        free(toDelete);
```

```
    }
}

// Function to delete a node from the end
void deleteFromEnd(struct Node** head)
{
    if (*head == NULL)
    {
        printf("List is empty\n");
        return;
    }

    if ((*head)->next == *head)
    {
        free(*head);
        *head = NULL;
    }
    else
    {
        struct Node* temp = *head;
        while (temp->next->next != *head)
        {
            temp = temp->next;
        }
        free(temp->next);
        temp->next = *head;
    }
}

// Function to delete a node from a given position
void deleteAtPosition(struct Node** head, int position)
{
    if (*head == NULL)
    {
        printf("List is empty\n");
        return;
    }

    if (position < 1)
    {
        printf("Invalid position\n");
        return;
    }
```

```
    if (position == 1)
    {
        deleteFromFront(head);
        return;
    }

    struct Node* temp = *head;
    struct Node* prev = NULL;

    for (int i = 1; i < position && temp->next != *head; i++)
    {
        prev = temp;
        temp = temp->next;
    }

    if (temp->next == *head && position != 2)
    {
        printf("Position out of range\n");
    }
    else
    {
        prev->next = temp->next;
        free(temp);
    }
}

// Function to search for a node in the list
int search(struct Node* head, int data)
{
    struct Node* temp = head;
    int position = 1;

    if (head != NULL)
    {
        do
        {
            if (temp->data == data)
            {
                return position;
            }
            temp = temp->next;
            position++;
        } while (temp != head);
    }
```

```
    return -1; // Data not found
}

// Function to display the list
void displayList(struct Node* head)
{
    if (head == NULL)
    {
        printf("List is empty\n");
        return;
    }

    struct Node* temp = head;
    do
    {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != head);
    printf("(head)\n");
}

int main()
{
    struct Node* head = NULL;
    int choice, data, position;

    while (1)
    {
        printf("\nMenu:\n");
        printf("1. Insert at End\n");
        printf("2. Insert at Front\n");
        printf("3. Insert at Position\n");
        printf("4. Delete from Front\n");
        printf("5. Delete from End\n");
        printf("6. Delete from Position\n");
        printf("7. Display List\n");
        printf("8. Search for an Element\n");
        printf("9. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
            case 1:
```

```
            printf("Enter the data to insert at end: ");
            scanf("%d", &data);
            insertAtEnd(&head, data);
            break;
        case 2:
            printf("Enter the data to insert at front: ");
            scanf("%d", &data);
            insertAtFront(&head, data);
            break;
        case 3:
            printf("Enter the data to insert: ");
            scanf("%d", &data);
            printf("Enter the position: ");
            scanf("%d", &position);
            insertAtPosition(&head, data, position);
            break;
        case 4:
            deleteFromFront(&head);
            break;
        case 5:
            deleteFromEnd(&head);
            break;
        case 6:
            printf("Enter the position to delete: ");
            scanf("%d", &position);
            deleteAtPosition(&head, position);
            break;
        case 7:
            displayList(head);
            break;
        case 8:
            printf("Enter the element to search for: ");
            scanf("%d", &data);
            position = search(head, data);
            if (position != -1)
            {
                printf("Element %d found at position %d\n", data, position);
            }
            else
            {
                printf("Element %d not found in the list\n", data);
            }
            break;
        case 9:
```

```
            printf("Exiting...\n");
            exit(0);
        default:
            printf("Invalid choice! Please try again.\n");
      }
   }

   return 0;
}
```

Implement a stack using singly linked list

To implement a stack using the singly linked list concept, all the singly linked list operations should be performed based on Stack operations LIFO(last in first out) and with the help of that knowledge, we are going to implement a stack using a singly linked list.

So we need to follow a simple rule in the implementation of a stack which is last in first out and all the operations can be performed with the help of a top variable.

Stack Operations:

push(): Insert a new element into the stack i.e just insert a new element at the beginning of the linked list.
pop(): Return the top element of the Stack i.e simply delete the first element from the linked list.
peek(): Return the top element.
display(): Print all elements in Stack

Push Operation:

- Initialise a node
- Update the value of that node by data i.e. node->data = data
- Now link this node to the top of the linked list
- And update top pointer to the current node

Pop Operation:

- First Check whether there is any node present in the linked list or not, if not then return
- Otherwise make pointer let say temp to the top node and move forward the top node by 1 step
- Now free this temp node

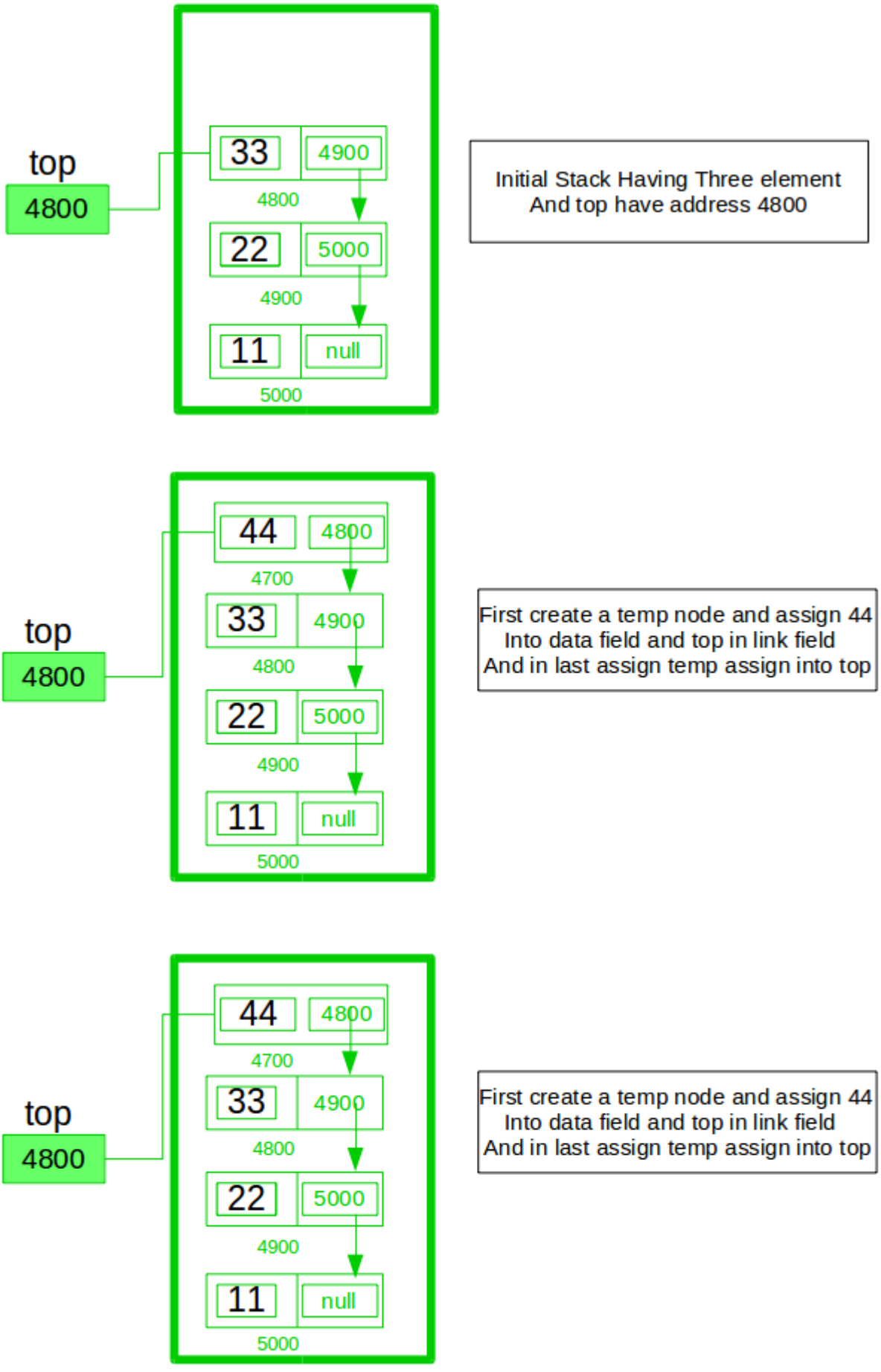
Peek Operation:

- Check if there is any node present or not, if not then return.
- Otherwise return the value of top node of the linked list

Display Operation:

- Take a temp node and initialize it with top pointer
- Now start traversing temp till it encounters NULL

- Simultaneously print the value of the temp node



Program:

```
#include <stdio.h>
#include <stdlib.h>

// Define the node structure
struct Node
{
   int data;
   struct Node* next;
};

void push(struct Node** top, int data)
{
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   if (!newNode)
   {
      printf("Memory allocation error\n");
      exit(1);
   }
   newNode->data = data;
   newNode->next = *top;  // Point the new node to the current top
   *top = newNode;  // Update the top to the new node
}

// Function to pop an element from the stack
int pop(struct Node** top)
{
   if (*top == NULL)
   {
      printf("Stack Underflow\n");
      return -1;  // Return -1 if stack is empty
   }
   struct Node* temp = *top;
   int data = temp->data;
   *top = (*top)->next;  // Update top to the next node
   free(temp);  // Free the memory of the popped node
   return data;
}

// Function to peek at the top element of the stack
int peek(struct Node* top)
{
   if (top == NULL)
   {
```

```
        printf("Stack is empty\n");
        return -1;
    }
    return top->data;  // Return the data of the top node
}

// Function to check if the stack is empty
int isEmpty(struct Node* top)
{
    return top == NULL;  // Return 1 if empty, 0 otherwise
}

// Function to display the stack
void displayStack(struct Node* top)
{
    if (top == NULL)
    {
        printf("Stack is empty\n");
        return;
    }
    struct Node* temp = top;
    while (temp != NULL)
    {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL (top)\n");
}

int main()
{
    struct Node* top = NULL;  // Initialize stack as empty

    // Push elements onto the stack
    push(&top, 10);
    push(&top, 20);
    push(&top, 30);
    push(&top, 40);

    printf("Stack: ");
    displayStack(top);  // Display the current stack

    printf("Top element is %d\n", peek(top));  // Display the top element
```

```
    printf("Popped element is %d\n", pop(&top));  // Pop the top element
    printf("Stack after pop: ");
    displayStack(top);  // Display the stack after pop

    return 0;
}
```