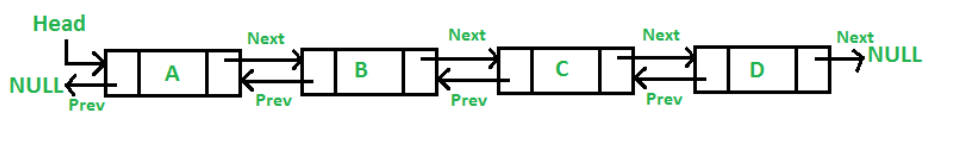
**Doubly Linked List in C**

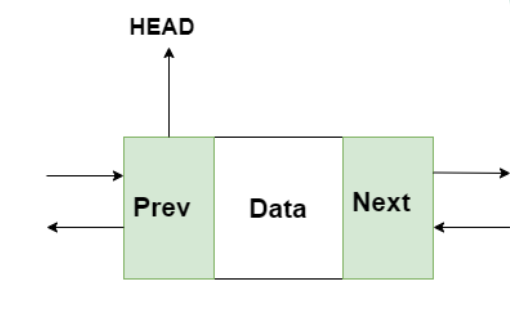
A **doubly linked list** is a type of linked list in which each node contains 3 parts, a data part and two addresses, one points to the previous node and one for the next node. It differs from the singly linked list as it has an extra pointer called previous that points to the previous node, allowing the traversal in both forward and backward directions.

**Doubly Linked List Representation in C**

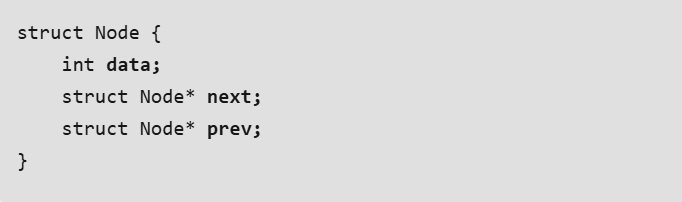
A doubly linked list is represented in C as the pointer to the head (i.e. first node) in the list. Each node in a doubly linked list contains three components:

1. **Data:**data is the actual information stored in the node.
2. **Next:** nextis apointer that links to the next node in the list.
3. **Prev:**previous**is**a pointer that links to the previous node in the list.





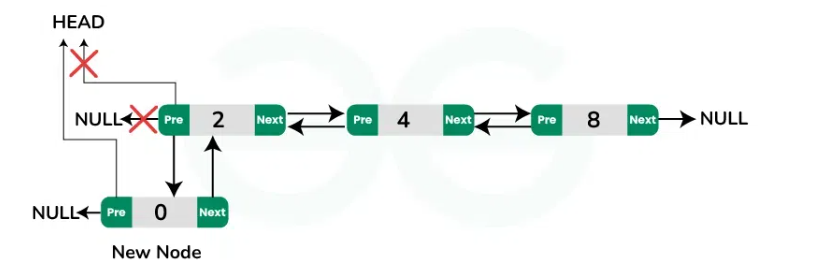
**The node structure can be defined as follows:**



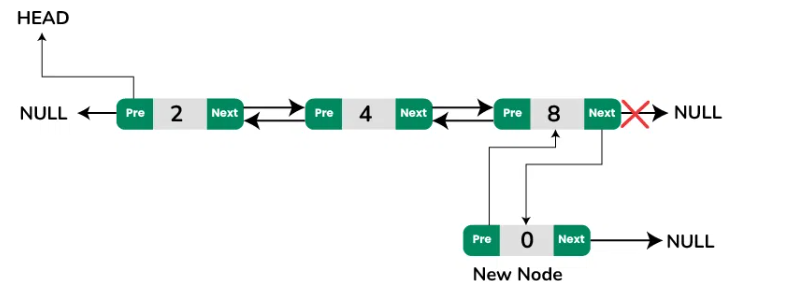
**Insertion and deletion operations in doubly linked list**

**Insertion**

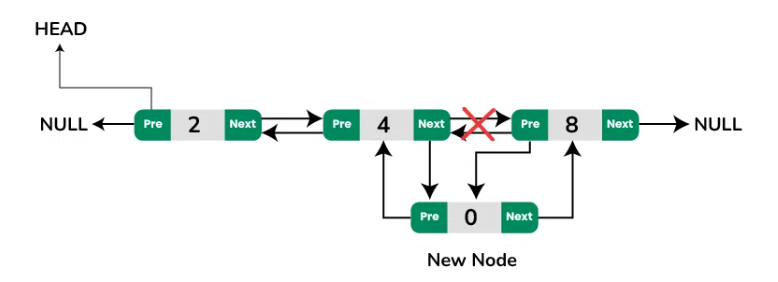
**Case 1: Insertion in the beginning**

****

**Case 2: Insertion at the end**

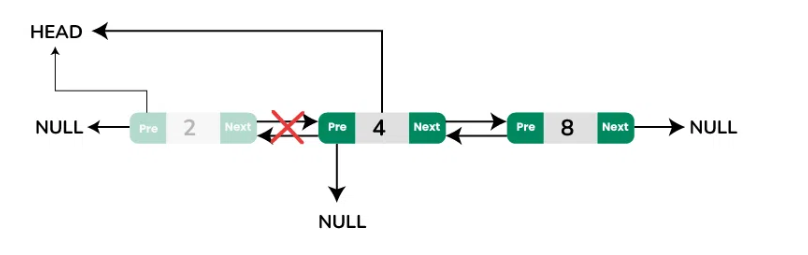
****

**Case 3: Insertion at a specific position**

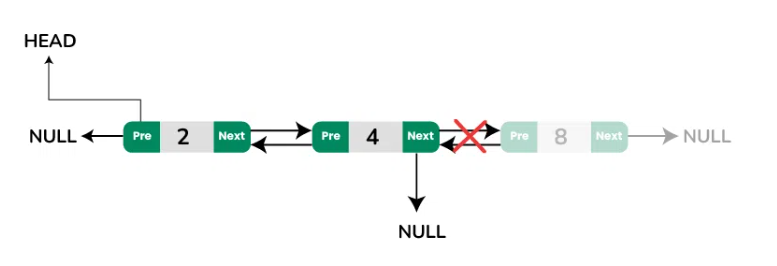
****

**Deletion**

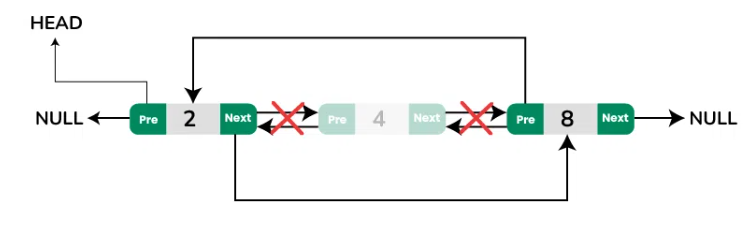
**Case 1: Deletion at the beginning**

****

**Case 2: Deletion at the end of the list**

****

**Case 3: Deletion at a specific position**

****

**C Program to implement doubly linked list**

// C Program to Implement Doubly Linked List

#include <stdio.h>

#include <stdlib.h>

// defining a node

typedef struct Node {

int data;

struct Node\* next;

struct Node\* prev;

} Node;

// Function to create a new node with given value as data

Node\* createNode(int data)

{

Node\* newNode = (Node\*)malloc(sizeof(Node));

newNode->data = data;

newNode->next = NULL;

newNode->prev = NULL;

return newNode;

}

// Function to insert a node at the beginning

void insertAtBeginning(Node\*\* head, int data)

{

// creating new node

Node\* newNode = createNode(data);

// check if DLL is empty

if (\*head == NULL) {

\*head = newNode;

return;

}

newNode->next = \*head;

(\*head)->prev = newNode;

\*head = newNode;

}

// Function to insert a node at the end

void insertAtEnd(Node\*\* head, int data)

{

// creating new node

Node\* newNode = createNode(data);

// check if DLL is empty

if (\*head == NULL) {

\*head = newNode;

return;

}

Node\* temp = \*head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

// Function to insert a node at a specified position

void insertAtPosition(Node\*\* head, int data, int position)

{

if (position < 1) {

printf("Position should be >= 1.\n");

return;

}

// if we are inserting at head

if (position == 1) {

insertAtBeginning(head, data);

return;

}

Node\* newNode = createNode(data);

Node\* temp = \*head;

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

temp = temp->next;

}

if (temp == NULL) {

printf(

"Position greater than the number of nodes.\n");

return;

}

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 beginning

void deleteAtBeginning(Node\*\* head)

{

// checking if the DLL is empty

if (\*head == NULL) {

printf("The list is already empty.\n");

return;

}

Node\* temp = \*head;

\*head = (\*head)->next;

if (\*head != NULL) {

(\*head)->prev = NULL;

}

free(temp);

}

// Function to delete a node from the end

void deleteAtEnd(Node\*\* head)

{

// checking if DLL is empty

if (\*head == NULL) {

printf("The list is already empty.\n");

return;

}

Node\* temp = \*head;

if (temp->next == NULL) {

\*head = NULL;

free(temp);

return;

}

while (temp->next != NULL) {

temp = temp->next;

}

temp->prev->next = NULL;

free(temp);

}

// Function to delete a node from a specified position

void deleteAtPosition(Node\*\* head, int position)

{

if (\*head == NULL) {

printf("The list is already empty.\n");

return;

}

Node\* temp = \*head;

if (position == 1) {

deleteAtBeginning(head);

return;

}

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

temp = temp->next;

}

if (temp == NULL) {

printf("Position is greater than the number of "

"nodes.\n");

return;

}

if (temp->next != NULL) {

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

}

if (temp->prev != NULL) {

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

}

free(temp);

}

// Function to print the list in forward direction

void printListForward(Node\* head)

{

Node\* temp = head;

printf("Forward List: ");

while (temp != NULL) {

printf("%d ", temp->data);

temp = temp->next;

}

printf("\n");

}

// Function to print the list in reverse direction

void printListReverse(Node\* head)

{

Node\* temp = head;

if (temp == NULL) {

printf("The list is empty.\n");

return;

}

// Move to the end of the list

while (temp->next != NULL) {

temp = temp->next;

}

// Traverse backwards

printf("Reverse List: ");

while (temp != NULL) {

printf("%d ", temp->data);

temp = temp->prev;

}

printf("\n");

}

//Main function

int main()

{

Node\* head = NULL;

// Demonstrating various operations

insertAtEnd(&head, 10);

insertAtEnd(&head, 20);

insertAtBeginning(&head, 5);

insertAtPosition(&head, 15, 2); // List: 5 15 10 20

printf("After Insertions:\n");

printListForward(head);

printListReverse(head);

deleteAtBeginning(&head); // List: 15 10 20

deleteAtEnd(&head); // List: 15 10

deleteAtPosition(&head, 2); // List: 15

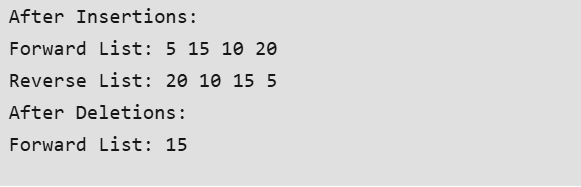
printf("After Deletions:\n");

printListForward(head);

return 0;

}

**OUTPUT**

****