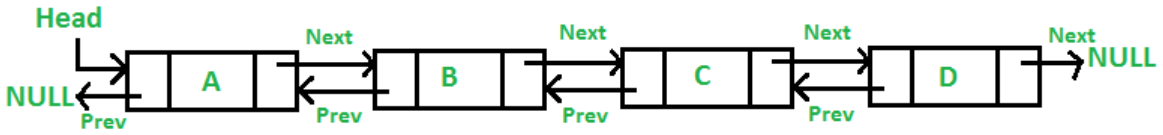
Doubly Linked List in C

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.
In this article, we will learn about the doubly linked list implementation in C. We will also look at the working of the doubly linked list and the basic operations that can be performed using the doubly linked list in C.

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:

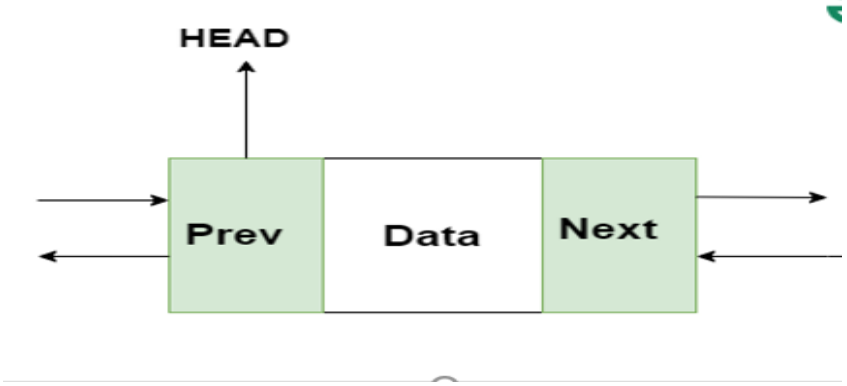
1. **Data:** data is the actual information stored in the node.
2. **Next:** next is a pointer that links to the next node in the list.
3. **Prev:** previous **is** a pointer that links to the previous node in the list.



Doubly Linked List

Implementation of Doubly Linked List in C

To implement a doubly linked list in C first, we need to define a node that has three parts: data, a pointer to the next node, and a pointer to the previous node, and then create a new node.
We can create a node using the structure which allows us to combine different data types into single type.



Define a Node in Doubly Linked List

Node-Doubly Linked List

To define a structure of a node in doubly linked list use the below format:

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

We can then dynamically create the node using malloc() and assign the values to the next, prev and data fields. It is generally preferred to create a function that creates a node, assign the data field and return the pointer to that node.

Basic Operations on C Doubly Linked List

We can perform the following basic operations in a doubly-linked list:

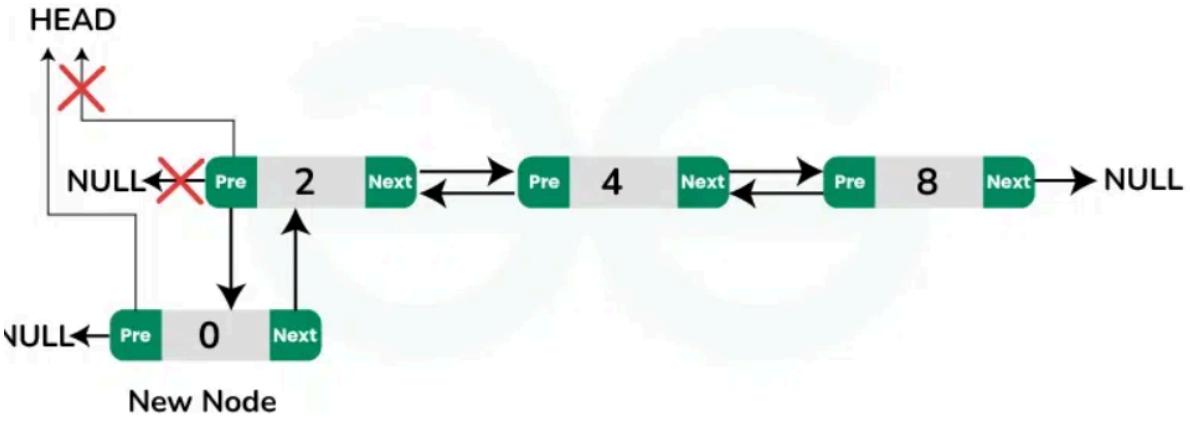
- Insertion
- Deletion
- Traversal

1. Insertion in Doubly Linked List in C

Inserting a new node in a doubly linked list can be done in a similar way like inserting a new node in a singly linked list but we have to maintain the link of previous node also. We have three scenarios while inserting a node in a doubly linked list:

- Insertion at the beginning of the list.
- Insertion at the end of the list.
- Insertion in the middle of the list.

a. Insertion at the Beginning of the Doubly Linked List



Insertion at the Beginning in Doubly Linked List

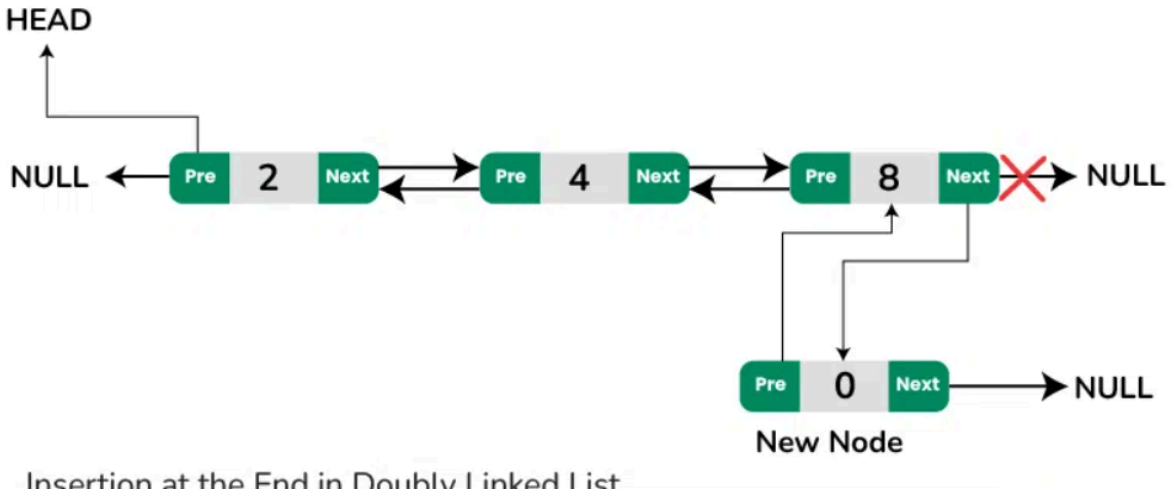
To insert a new node at the beginning of the doubly linked list follow the below approach:

Approach:

- *Create a newNode with **data** field assigned the given value.*
- *If the list is empty:*
    - *Set newNode->next to NULL.*
    - *Set newNode->prev to NULL.*
    - *Update the head pointer to point to newNode.*
- *If the list is not empty:*
    - *Set newNode->next to head.*
    - *Set newNode->prev to NULL.*
    - *Set head->prev to newNode.*
    - *Update the head pointer to point to newNode.*

Time Complexity: O(1), as insertion is done at front so we are not traversing through the list.
Space Complexity: O(1)

b. Insertion at the End of the Doubly Linked List



Insertion at the End in Doubly Linked List

Insertion at the end in doubly linked list

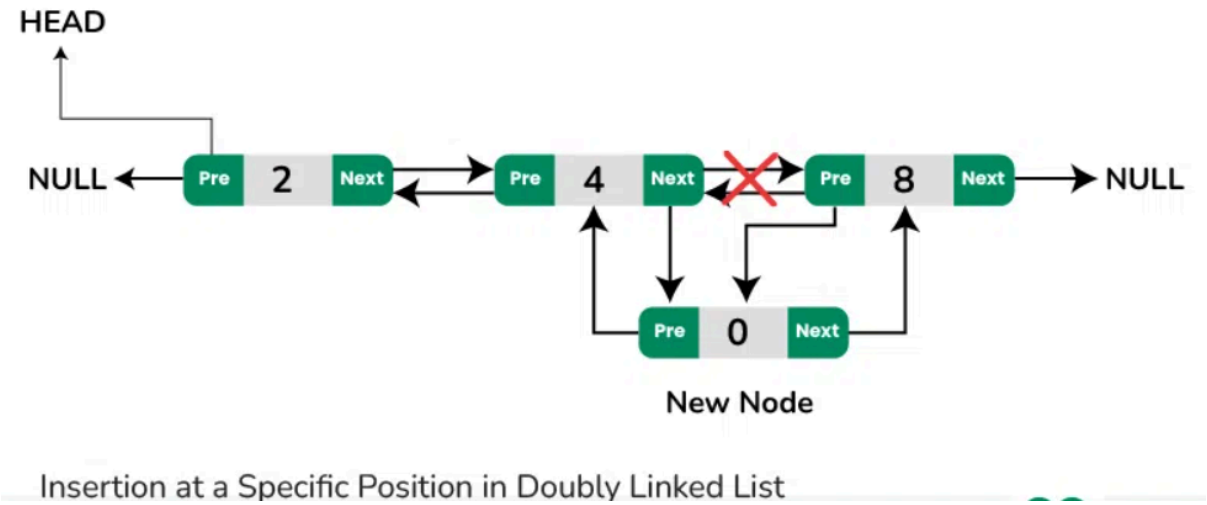
To insert a new node at the end of the doubly linked list, follow the below approach:

Approach:

- *Create a newNode with **data** field assigned the given value.*
- *If the list is empty:*
    - *Set newNode->next to NULL.*
    - *Set newNode->prev to NULL.*
    - *Update the head pointer to point to newNode.*
- *If the list is not empty:*
    - *Traverse the list to find the last node (where last->next == NULL).*
    - *Set last->next to newNode.*
    - *Set newNode->prev to last.*
    - *Set newNode->next to NULL.*

Time Complexity: O(n), as insertion is done at the end, so we need to traverse through the list.
Space Complexty: O(1)

c. Insertion in the Middle of the Doubly Linked List



Insertion at a Specific Position in Doubly Linked List

Insertion in the middle of doubly linked list

For inserting a node at a specific position in a doubly linked list follow the below approach:

Approach:

- *Create a new Node.*
- *Find the size of the list to ensure the position is within valid bounds.*
- *If the position is 0 (or 1 depending on your indexing):*
  - *Use the insertion at the beginning approach.*
- *If the position is equal to the size of the list:*
  - *Use the insertion at the end approach.*
- *If the position is valid and not at the boundaries:*
  - *Traverse the list to find the node immediately before the desired insertion point (prevNode).*
  - *Set newNode->next to prevNode->next.*
  - *Set newNode->prev to prevNode.*
  - *If prevNode->next is not NULL, set prevNode->next->prev to newNode.*
  - *Set prevNode->next to newNode.*

Time Complexity: O(n), as insertion is done in between, so we need to traverse through the list until we reach the desired position.
Space Complexity: O(1)

2. Deletion in a Doubly Linked List in C

Just like insertion, we have three scenarios while deleting a node in a doubly linked list:

- Deletion from the beginning of the list.
- Deletion from the end of the list.
- Deletion at specific position of the list.

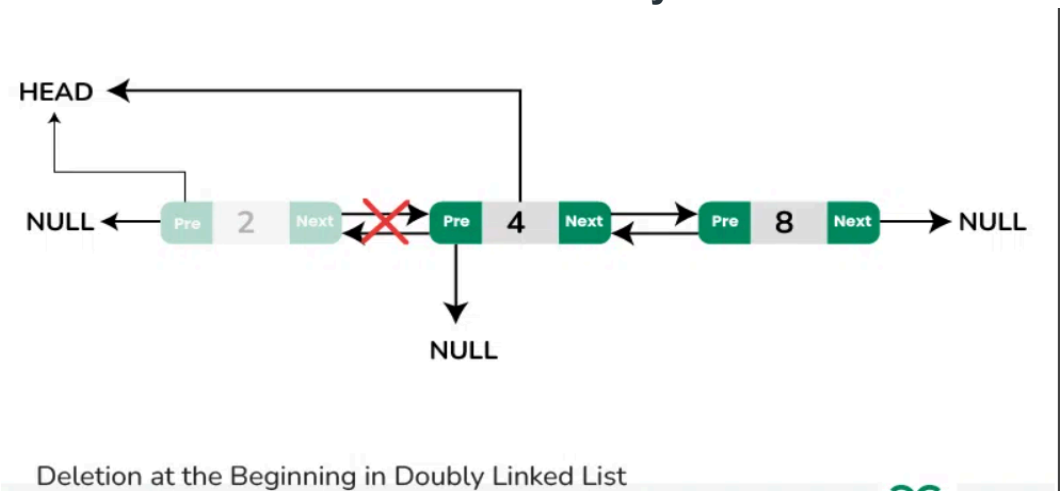
a. Deletion at the Beginning of the Doubly Linked List

To delete a node at the beginning of the doubly linked list, follow the below approach:
Approach:

- ***Check if the List is Empty****: If true, there's nothing to delete.*
- ***Check if the List Contains Only One Node****:*
    - *Set head to NULL.*
    - *Free the memory of the node.*
- ***If the List Contains More Than One Node****:*
    - *Update head to head->next.*
    - *Set the prev of the new head to NULL.*
    - *Free the memory of the old head.*

Time Complexity: O(1), as deletion is done at front so we are not traversing through the list.

b. Deletion at the End of the Doubly Linked List



Deletion at the Beginning in Doubly Linked List

To delete a node at the end of the doubly linked list, follow the below approach:
Approach:

- *Check if the List is Empty: If true, there's nothing to delete.*
- *If the List is Not Empty:*
    - *Traverse to the last node using a loop.*
    - *Check if the last node is the only node (head->next == NULL):*
        - *Update head to NULL.*
    - *If more than one node:*
        - *Set the next of the second last node (last->prev) to NULL.*
    - *Free the memory of the last node.*

Time Complexity: O(n), as deletion is done at the end, so we need to traverse through the list. If a tail pointer is maintained, this operation can be optimized to O(1) by directly accessing the last node.

For deleting a node at a specific position in a doubly linked list, follow the below approach:

Approach:

- ***Check Position Validity****:*
    - *If position is 0 (or 1 depending on indexing), use deletion at the beginning.*
    - *If position is the size of the list, use deletion at the end.*
    - *Otherwise, proceed with the middle deletion.*
- ***Traverse to the Specified Position****:*
    - *Use a loop to reach the node (delNode) at the desired position.*
    - *If delNode is the first node, adjust head.*
    - *If not, adjust delNode->prev->next and delNode->next->prev if delNode->next is not NULL.*
- ***Free the Memory****:*
    - *Release the node to be deleted.*

Time Complexity: O(n), as deletion is done in between, as we may need to traverse up to n elements to find the correct position.

3. Traversal in a Doubly Linked List in C

Traversal in a doubly linked list means visiting each node of the doubly linked list to perform some kind of operation like displaying the data stored in that node. Unlike singly linked list we can traverse in doubly linked list in both the directions.

- **Forward Traversal:** From head node to tail node
- **Reverse Traversal:** From tail node to head node

a. Forward Traversal in Doubly Linked List

For traversing in a forward direction in a doubly linked list, follow the below approach:

Approach:

- *Create a temporary pointer ptr and copy the head pointer into it.*
- *Traverse through the list using a while loop.*
- *Keep moving the value of temp pointer variable ptr to ptr->next until the last node whose next part contains null is found.*
- *During each iteration of the loop, perform the desired operation.*

Time Complexity: O(n), as we need to traverse through the whole list containing n number of nodes.

b. Reverse Traversal in Doubly Linked List

For traversing in a reverse direction in a doubly linked list, follow the below approach:

- *Create a temporary pointer ptr and copy the tail pointer into it.*
- *Traverse through the list using a while loop.*
- *Keep moving the value of temp pointer variable ptr to ptr->prev until the first node whose prev part contains null is found.*
- *During each iteration of the loop, perform the desired operation .*

Time Complexity: O(n), as here also we need to traverse through the whole list containing n number of nodes.

C Program to Implement Doubly Linked List

The below program demonstrates all the major operations of a doubly linked list: insertion (at the beginning, at the end, and at a specific position), deletion (at the beginning, at the end, and at a specific position), and traversal (forward and reverse).

```
// 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");
}

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

```
After Insertions:

Forward List: 5 15 10 20

Reverse List: 20 10 15 5

After Deletions:

Forward List: 15
```

Advantages of Doubly Linked List

- Unlike singly linked list, we can traverse in both the direction forward and reverse using doubly linked list which makes operations like reversing and searching from end easier and more efiicient.

- It is fast and easier to delete a node in doubly linked list, if the node to be deleted is given then we can search the previous node quickly and update it's links.
- Inserting a new node is also easier in doubly linked list, as we can simply insert a new node before a given node by updating the links.
- In a doubly linked list, we need not to keep track of the previous node during traversal that is very useful in data structures like the binary tree where we need to keep track of the parent node.

Disadvantages of Doubly Linked List

- Although doubly linked lists provide more flexibility and efficiency in certain operations compared to singly linked lists, they also consume more memory as they need to store an extra pointer for each node.
- The insertion and deletion code in a doubly linked list is more complex as we need to maintain the previous links too.
- For insertion and deletion at a specific position in a doubly linked list we need to traverse from a head node to the specified node that can be time-consuming in case of larger lists.
- For each node we have to maintain two pointers in a doubly linked list that leads to the overhead of maintaining and updating these pointers during insertions and deletions.