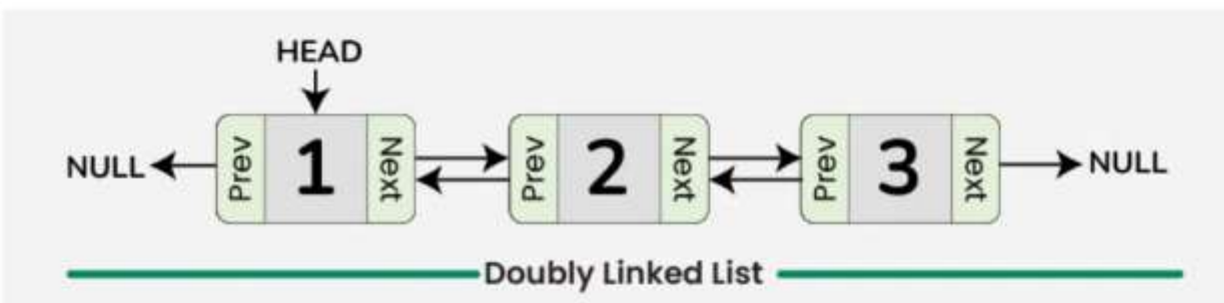
A **doubly linked list** is a more complex data structure than a singly linked list, but it offers several advantages. The main advantage of a doubly linked list is that it allows for efficient traversal of the list in both directions. This is because each node in the list contains a pointer to the previous node and a pointer to the next node. This allows for quick and easy insertion and deletion of nodes from the list, as well as efficient traversal of the list in both directions.

What is a Doubly Linked List?

A **doubly linked list** is a data structure that consists of a set of nodes, each of which contains a **value** and **two pointers**, one pointing to the **previous node** in the list and one pointing to the **next node** in the list. This allows for efficient traversal of the list in **both directions**, making it suitable for applications where frequent **insertions** and **deletions** are required.

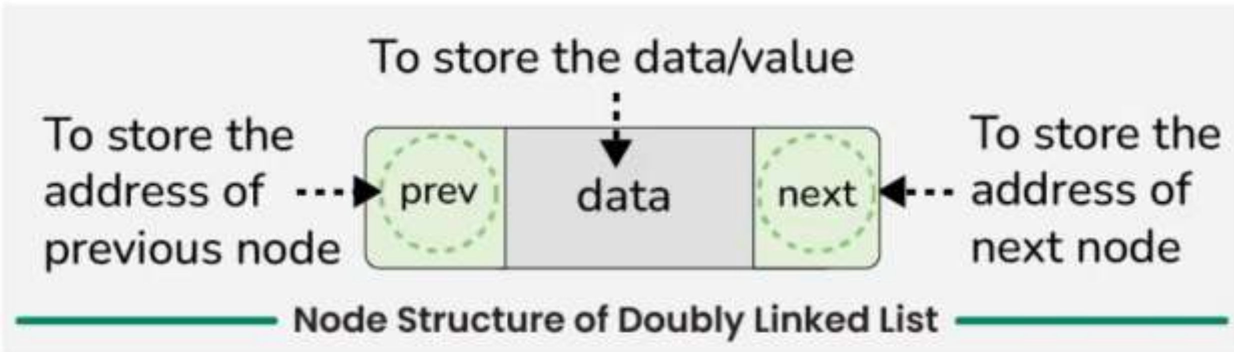


Doubly Linked List

Representation of Doubly Linked List in Data Structure

In a data structure, a doubly linked list is represented using nodes that have three fields:

1. Data
2. A pointer to the next node (**next**)
3. A pointer to the previous node (**prev**)



Node Structure of Doubly Linked List

Node Definition

Here is how a node in a Doubly Linked List is typically represented:

Recommended Problem

[Doubly Linked List Traversal]

```
struct Node {

    // To store the Value or data.
    int data;

    // Pointer to point the Previous Element
    Node* prev;

    // Pointer to point the Next Element
    Node* next;

    // Constructor
    Node(int d) {
       data = d;
       prev = next = nullptr;
    }
};
```

Each node in a **Doubly Linked List** contains the **data** it holds, a pointer to the **next** node in the list, and a pointer to the **previous** node in the list. By linking these nodes together through the **next** and **prev** pointers, we can traverse the list in both directions (forward and backward), which is a key feature of a Doubly Linked List.

Operations on Doubly Linked List

- **Traversal in Doubly Linked List**
- **Searching in Doubly Linked List**
- **Finding Length of Doubly Linked List**
- **Insertion in Doubly Linked List**:
  - Insertion at the beginning of Doubly Linked List
  - Insertion at the end of the Doubly Linked List
  - Insertion at a specific position in Doubly Linked List
- **Deletion in Doubly Linked List**:
  - Deletion of a node at the beginning of Doubly Linked List
  - Deletion of a node at the end of Doubly Linked List
  - Deletion of a node at a specific position in Doubly Linked List

Let’s go through each of the operations mentioned above, one by one.

Traversal in Doubly Linked List

To Traverse the doubly list, we can use the following steps:

a. Forward Traversal:

- Initialize a pointer to the head of the linked list.
- While the pointer is not null:
  - Visit the data at the current node.
  - Move the pointer to the next node.

b. Backward Traversal:

- Initialize a pointer to the tail of the linked list.
- While the pointer is not null:
  - Visit the data at the current node.

- Move the pointer to the previous node.

Below are the implementation of the above approach:

C++CJavaPythonC#JavaScript

```
#include <iostream>
using namespace std;

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

    // Constructor to initialize Node with data
    Node(int data) : data(data), next(nullptr),
        prev(nullptr) {}
};

// Function to traverse the doubly linked list
// in forward direction
void forwardTraversal(Node* head) {

    // Start traversal from the head of the list
    Node* curr = head;

    // Continue until current node is not null
    // (end of list)
    while (curr != nullptr) {

        // Output data of the current node
        cout << curr->data << " ";

        // Move to the next node
        curr = curr->next;
```

```
    }

    // Print newline after traversal
    cout << endl;
}

// Function to traverse the doubly linked list
// in backward direction
void backwardTraversal(Node* tail) {

    // Start traversal from the tail of the list
    Node* curr = tail;

    // Continue until current node is not null
    // (end of list)
    while (curr != nullptr) {

        // Output data of the current node
        cout << curr->data << " ";

        // Move to the previous node
        curr = curr->prev;
    }

    // Print newline after traversal
    cout << endl;
}

int main() {

    // Sample usage of the doubly linked list and
    // traversal functions
    Node* head = new Node(1);
    Node* second = new Node(2);
```

```
    Node* third = new Node(3);

    head->next = second;
    second->prev = head;
    second->next = third;
    third->prev = second;

    cout << "Forward Traversal:" << endl;
    forwardTraversal(head);

    cout << "Backward Traversal:" << endl;
    backwardTraversal(third);

    return 0;
}
```

Output

```
Forward Traversal:

1 2 3

Backward Traversal:

3 2 1
```

Finding Length of Doubly Linked List

To find the length of doubly list, we can use the following steps:

- Start at the head of the list.
- Traverse through the list, counting each node visited.
- Return the total count of nodes as the length of the list.

Below are the implementation of the above approach:

C++CJavaPythonC#JavaScript

```
#include <iostream>
```

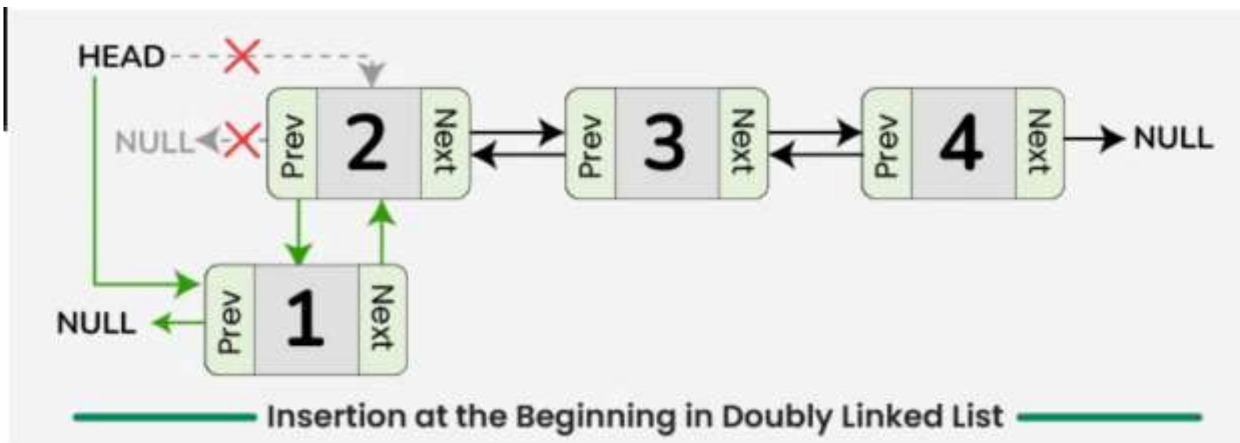
```
using namespace std;

// Node structure for doubly linked list
struct Node {
    int data;
    Node * prev;
    Node * next;

    Node(int val) {
        data = val;
        prev = next = nullptr;
    }
};

// Function to find the length of a doubly
//linked list
int findLength(Node * head) {
    int count = 0;
    for (Node * cur = head; cur != nullptr; cur = cur -> next)
        count++;
    return count;
}

int main() {

    // Create a DLL with 3 nodes
    Node * head = new Node(1);
    Node * second = new Node(2);
    Node * third = new Node(3);
    head -> next = second;
    second -> prev = head;
    second -> next = third;
    third -> prev = second;
```

```
    cout << "Length of the doubly linked list: " <<
        findLength(head) << endl;

    return 0;
}
```

Output

```
Length of the doubly linked list: 3
```

Insertion at the Beginning in Doubly Linked List



Insertion at the Beginning in Doubly Linked List

To insert a new node at the beginning of the doubly list, we can use the following steps:

- 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 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.

Below are the implementation of the above approach:

C++CJavaPythonC#JavaScript

```
// C++ Program to insert a new node at the
// beginning of doubly linked list

#include <iostream>
using namespace std;

// Node structure for the doubly linked list
struct Node {
    int data;
    Node* prev;
    Node* next;

    Node(int d) {
      data = d;
      prev = next = NULL;
    }
};

// Insert a node at the beginning
Node* insertBegin(Node* head, int data) {

    // Create a new node
    Node* new_node = new Node(data);

    // Make next of it as head
    new_node->next = head;

    // Set previous of head as new node
    if (head != NULL) {
        head->prev = new_node;
    }
```

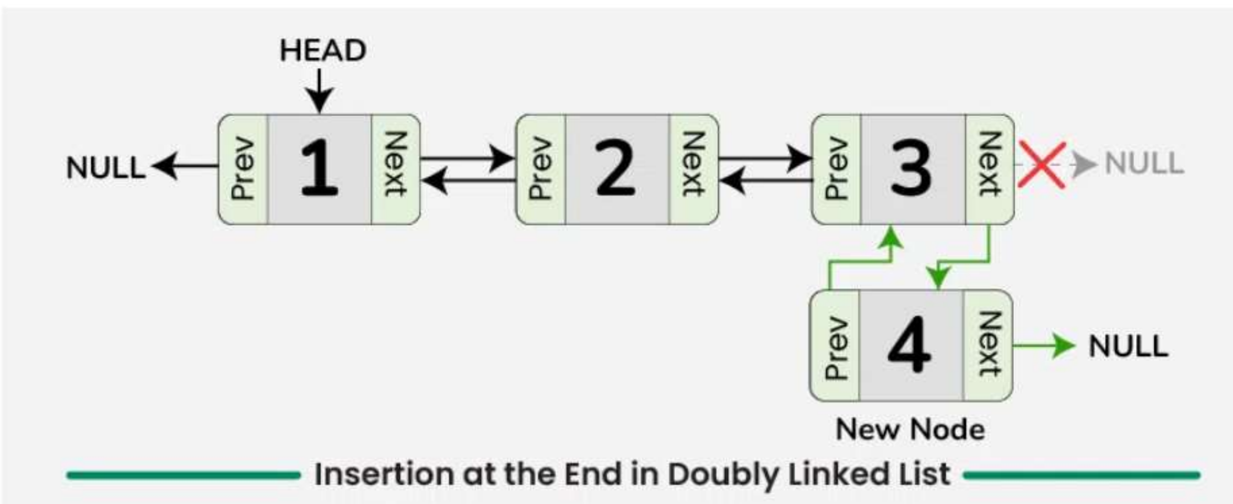
```
    // Return new node as new head
    return new_node;
}

void printList(Node* head) {
    Node* curr = head;
    while (curr != NULL) {
        cout << curr->data << " ";
        curr = curr->next;
    }
        cout << "\n";
}

int main() {

    // Create a hardcoded linked list:
        // 2 <-> 3 <-> 4
    Node* head = new Node(2);
    Node* temp1 = new Node(3);
    Node* temp2 = new Node(4);
    head->next = temp1;
    temp1->prev = head;
    temp1->next = temp2;
    temp2->prev = temp1;

        // Print the original list
    cout << "Original Linked List: ";
    printList(head);

    // Insert a new node at the front of the list
    head = insertBegin(head, 1);

    // Print the updated list
```

```
        cout << "After inserting Node 1 at the front: ";
    printList(head);

    return 0;
}
```

Output

```
Original Linked List: 2 3 4

After inserting Node 1 at the front: 1 2 3 4
```

Insertion at the End of Doubly Linked List



Insertion at the End in the Doubly Linked List

To insert a new node at the end of the doubly linked list, we can use the following steps:

- Allocate memory for a new node and assign the provided value to its data field.
- Initialize the next pointer of the new node to nullptr.
- If the list is empty:
    - Set the previous pointer of the new node to nullptr.
    - Update the head pointer to point to the new node.

- If the list is not empty:
  - Traverse the list starting from the head to reach the last node.
  - Set the next pointer of the last node to point to the new node.
  - Set the previous pointer of the new node to point to the last node.

Below are the implementation of the above approach:

C++CJavaPythonC#JavaScript

```
// C++ Program to insert a node at the end of
//doubly linked list

#include <bits/stdc++.h>
using namespace std;

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

    Node(int new_data) {
        data = new_data;
        next = prev = nullptr;
    }
};

// Function to insert a new node at the end of
//doubly linked list
Node *insertEnd(Node *head, int new_data) {

    // Create a new node
    Node *new_node = new Node(new_data);

    // If the linked list is empty, set the new
        //node as the head of linked list
```

```
    if (head == NULL) {
        head = new_node;
    }
    else {
          Node *curr = head;
        while (curr->next != NULL) {
            curr = curr->next;
        }

        // Set the next of last node to new node
        curr->next = new_node;

        // Set prev of new node to last node
        new_node->prev = curr;
    }

    // Return the head of the doubly linked list
    return head;
}

void printList(Node *head) {
    Node *curr = head;
    while (curr != NULL) {
        cout << curr->data << " ";
        curr = curr->next;
    }
    cout << endl;
}

int main() {

    // Create a harcoded doubly linked list:
    // 1 <-> 2 <-> 3
    Node *head = new Node(1);
```

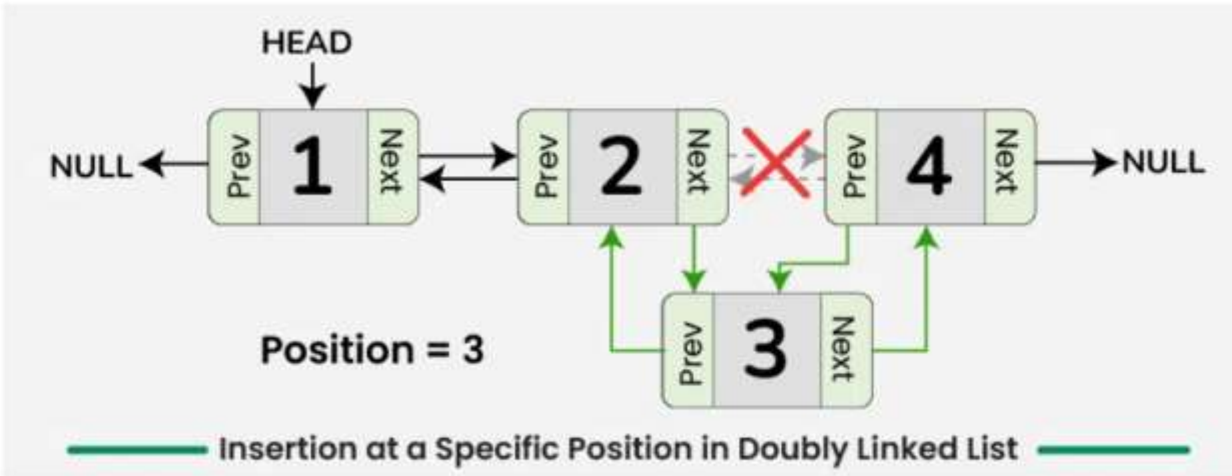
```
    head->next = new Node(2);
    head->next->prev = head;
    head->next->next = new Node(3);
    head->next->next->prev = head->next;

    // Print the original list
    cout << "Original Linked List: ";
    printList(head);

    // Insert a new node with data 4 at the end
    cout << "Inserting Node with data 4 at the end: ";
    int data = 4;
    head = insertEnd(head, data);

    // Print the updated list
    printList(head);

    return 0;
}
```

Output

```
Original Linked List: 1 2 3

Inserting Node with data 4 at the end: 1 2 3 4
```

Insertion at a Specific Position in Doubly Linked List

To insert a node at a specific Position in doubly linked list, we can use the following steps:



Insertion at a Specific Position in Doubly Linked List

To insert a new node at a specific position,

- 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.**

Below is the implementation of the above approach:

C++CJavaPythonC#JavaScript

```
// C++ Program to insert a node at a given position


#include <bits/stdc++.h>
using namespace std;


struct Node {
```

```
    int data;
    Node *next, *prev;

    Node(int new_data) {
        data = new_data;
        next = prev = nullptr;
    }
};

// Function to insert a new node at a given position
Node *insertAtPosition(Node *head, int pos, int new_data) {

    // Create a new node
    Node *new_node = new Node(new_data);

    // Insertion at the beginning
    if (pos == 1) {
        new_node->next = head;

        // If the linked list is not empty, set the prev
        //of head to new node
        if (head != NULL)
            head->prev = new_node;

        // Set the new node as the head of linked list
        head = new_node;
        return head;
    }

    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;
```

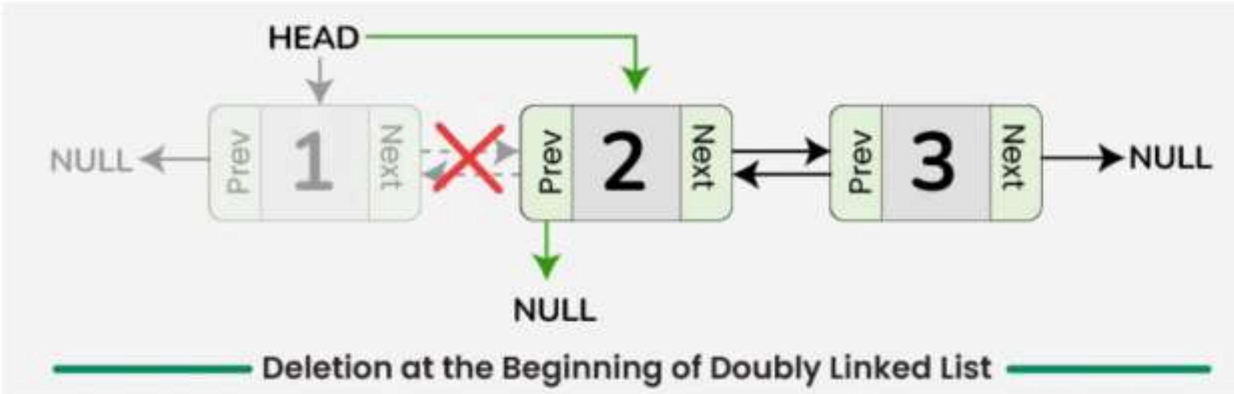
```
    }

    // If the position is out of bounds
    if (curr == NULL) {
        cout << "Position is out of bounds." << endl;
        delete new_node;
        return head;
    }

    // Set the prev of new node to curr
    new_node->prev = curr;

    // Set the new of new node to next of curr
    new_node->next = curr->next;

    // Update the next of current node to new node
    curr->next = new_node;

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

    // Return the head of the doubly linked list
    return head;
}

void printList(Node *head) {
    Node *curr = head;
    while (curr != NULL) {
        cout << curr->data << " ";
        curr = curr->next;
    }
    cout << endl;
```

```
}

int main() {

    // Create a harcoded doubly linked list:
    // 1 <-> 2 <-> 4
    Node *head = new Node(1);
    head->next = new Node(2);
    head->next->prev = head;
    head->next->next = new Node(4);
    head->next->next->prev = head->next;

    // Print the original list
    cout << "Original Linked List: ";
    printList(head);

    // Insert new node with data 3 at position 3
    cout << "Inserting Node with data 3 at position 3: ";
    int data = 3;
    int pos = 3;
    head = insertAtPosition(head, pos, data);

    // Print the updated list
    printList(head);

    return 0;
}
```

Output

```
Original Linked List: 1 2 4

Inserting Node with data 3 at position 3: 1 2 3 4
```

Deletion at the Beginning of Doubly Linked List



Deletion at the Beginning of Doubly Linked List

To delete a node at the beginning in doubly linked list, we can use the following steps:

- 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**.

Below is the implementation of the above approach:

C++CJavaPythonC#JavaScript

```
// C++ Program to delete a node from the
// beginning of Doubly Linked List

#include <bits/stdc++.h>
using namespace std;

struct Node{
    int data;
    Node *prev;
    Node *next;
    Node(int d) {
        data = d;
        prev = next = nullptr;
```

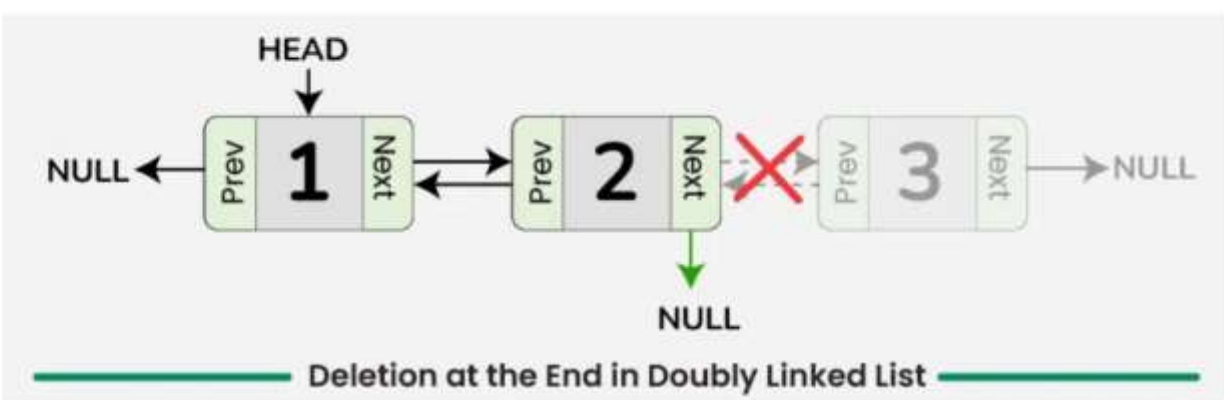
```
    }
};

// Deletes the first node (head) of the list
// and returns the second node as new head
Node *delHead(Node *head) {

    // If empty, return
    if (head == nullptr)
        return nullptr;

    // Store in temp for deletion later
    Node *temp = head;

    // Move head to the next node
    head = head->next;

    // Set prev of the new head
    if (head != nullptr)
        head->prev = nullptr;

    // Free memory and return new head
    delete temp;
    return head;
}

void printList(Node *head) {
    for (Node *curr = head; curr != nullptr; curr = curr->next)
        cout << curr->data << " ";
    cout << endl;
}

int main() {
```

```
    // Create a hardcoded doubly linked list:
    // 1 <-> 2 <-> 3
    struct Node *head = new Node(1);
    head->next = new Node(2);
    head->next->prev = head;
    head->next->next = new Node(3);
    head->next->next->prev = head->next;

    printf("Original Linked List: ");
    printList(head);

    printf("After Deletion at the beginning: ");
    head = delHead(head);

    printList(head);

    return 0;
}
```

Output

```
Original Linked List: 1 2 3

After Deletion at the beginning: 2 3
```

[Deletion at the End of Doubly Linked List]



Deletion at the End in Doubly Linked List

To delete a node at the end in doubly linked list, we can use the following steps:

- 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.

Below is the implementation of the above approach:

C++CJavaPythonC#JavaScript

```
// C++ Program to delete a node from the end of
//Doubly Linked List

#include <bits/stdc++.h>
using namespace std;

struct Node {
    int data;
    Node *prev;
    Node *next;
    Node(int d) {
        data = d;
        prev = NULL;
        next = NULL;
    }
};

// Function to delete the last node of the doubly
// linked list
Node *delLast(Node *head) {
```

```
    // Corner cases
    if (head == NULL)
        return NULL;
    if (head->next == NULL) {
        delete head;
        return NULL;
    }

    // Traverse to the last node
    Node *curr = head;
    while (curr->next != NULL)
        curr = curr->next;

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

        // Delete the last node
    delete curr;

    // Return the updated head
    return head;
}

void printList(Node *head) {
    Node *curr = head;
    while (curr != NULL) {
        cout << curr->data << " ";
        curr = curr->next;
    }
    cout << endl;
}

int main() {
```

```
    // Create a hardcoded doubly linked list:
    // 1 <-> 2 <-> 3
    struct Node *head = new Node(1);
    head->next = new Node(2);
    head->next->prev = head;
    head->next->next = new Node(3);
    head->next->next->prev = head->next;

    printf("Original Linked List: ");
    printList(head);

    printf("After Deletion at the end: ");
    head = delLast(head);

    printList(head);

    return 0;
}
```

Output

```
Original Linked List: 1 2 3

After Deletion at the end: 1 2
```

Deletion at a Specific Position in Doubly Linked List

Deletion at a Specific Position in Doubly Linked List

To delete a node at a specific position in doubly linked list, we can use the following steps:

- 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.

Below is the implementation of the above approach:

C++CJavaPythonC#JavaScript

```
// C++ Program to delete node at a specific position
// in Doubly Linked List

#include <iostream>

using namespace std;

struct Node {
    int data;
```

```
    Node * prev;
    Node * next;
    Node(int d) {
        data = d;
        prev = next = NULL;
    }
};

// Function to delete a node at a specific position
// in the doubly linked list
Node * delPos(Node * head, int pos) {

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

    Node * curr = head;

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

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

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

    // Update the next node's prev pointer
    if (curr -> next)
        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
    delete curr;
    return head;
}

// Function to print the doubly linked list
void printList(Node * head) {
    Node * curr = head;
    while (curr != nullptr) {
        cout << curr -> data << " ";
        curr = curr -> next;
    }
    cout << endl;
}

int main() {

    // Create a hardcoded doubly linked list:
    // 1 <-> 2 <-> 3
    struct Node * head = new Node(1);
    head -> next = new Node(2);
    head -> next -> prev = head;
    head -> next -> next = new Node(3);
    head -> next -> next -> prev = head -> next;

    cout << "Original Linked List: ";
    printList(head);

    cout << "After Deletion at the position 2: ";
```

```
    head = delPos(head, 2);

    printList(head);

    return 0;
}
```

Output

```
Original Linked List: 1 2 3

After Deletion at the position 2: 1 3
```

Advantages of Doubly Linked List

- **Efficient traversal in both directions:** Doubly linked lists allow for efficient traversal of the list in both directions, making it suitable for applications where frequent insertions and deletions are required.
- **Easy insertion and deletion of nodes:** The presence of pointers to both the previous and next nodes makes it easy to insert or delete nodes from the list, without having to traverse the entire list.
- **Can be used to implement a stack or queue:** Doubly linked lists can be used to implement both stacks and queues, which are common data structures used in programming.

Disadvantages of Doubly Linked List

- **More complex than singly linked lists:** Doubly linked lists are more complex than singly linked lists, as they require additional pointers for each node.
- **More memory overhead:** Doubly linked lists require more memory overhead than singly linked lists, as each node stores two pointers instead of one.

Applications of Doubly Linked List

- Implementation of undo and redo functionality in text editors.
- Cache implementation where quick insertion and deletion of elements are required.
- Browser history management to navigate back and forth between visited pages.
- Music player applications to manage playlists and navigate through songs efficiently.
- Implementing data structures like Deque (double-ended queue) for efficient insertion and deletion at both ends.