Insertion into Linked List:

Inserting a node into a linked list involves adding a new node with some data at a specified position within the list.

1. **Insertion at the Beginning (Head)**:
2. **Insertion in the Middle**:
3. **Insertion at the End**:

Let's break down each type of insertion in a linked list with explanations and examples.

1. **Insertion at the Beginning (Head)**:
    - **Algorithm**:
        1. Create a new node with the desired data value.
        2. Make the new node's **next** point to the current head of the list.
        3. Update the head pointer to point to the new node.
    - **Example**:
        - Suppose you have a linked list like this: **2 -> 3 -> 4 -> nullptr**
        - You want to insert a new node with the value **1** at the beginning.
        - After the insertion, the list will look like this: **1 -> 2 -> 3 -> 4 -> nullptr**
2. **Insertion in the Middle**:
    - **Algorithm**:
        0. Create a new node with the desired data value.
        1. Traverse the list to find the node after which you want to insert the new node.
        2. Update the new node's **next** to point to the node that comes after it.
        3. Update the previous node's **next** to point to the new node.
    - **Example**:
        - Suppose you have a linked list like this: **1 -> 2 -> 4 -> nullptr**
        - You want to insert a new node with the value **3** after the node with the value **2**.
        - After the insertion, the list will look like this: **1 -> 2 -> 3 -> 4 -> nullptr**
3. **Insertion at the End**:

- **Algorithm**:
    0. Create a new node with the desired data value.
    1. Traverse the list to find the last node (the one whose **next** points to **nullptr**).
    2. Update the last node's **next** to point to the new node.
- **Example**:
    - Suppose you have a linked list like this: **1 -> 2 -> nullptr**
    - You want to insert a new node with the value **3** at the end.
    - After the insertion, the list will look like this: **1 -> 2 -> 3 -> nullptr**

CPP program to demonstrate insertion operation at linked list from above algorithm

```
#include <iostream>

class Node {
public:
    int data;
    Node* next;

    Node(int value) {
        data = value;
        next = nullptr;
    }
};

class LinkedList {
public:
    Node* head;

    LinkedList() {
        head = nullptr;
    }

    // Function to insert a node at the beginning of the linked list
    void insertAtBeginning(int value) {
        Node* newNode = new Node(value);
        newNode->next = head;
        head = newNode;
    }

    // Function to insert a node in the middle of the linked list
    void insertInMiddle(int value, int position) {
        Node* newNode = new Node(value);

        if (position == 0 || head == nullptr) {
            newNode->next = head;
            head = newNode;
```

```
        } else {
            Node* current = head;
            int currentPosition = 0;

            while (currentPosition < position - 1 && current->next !=
nullptr) {
                current = current->next;
                currentPosition++;
            }

            newNode->next = current->next;
            current->next = newNode;
        }
    }

    // Function to insert a node at the end of the linked list
    void insertAtEnd(int value) {
        Node* newNode = new Node(value);

        if (head == nullptr) {
            head = newNode;
        } else {
            Node* current = head;

            while (current->next != nullptr) {
                current = current->next;
            }

            current->next = newNode;
        }
    }

    // Function to display the linked list
    void display() {
        Node* current = head;

        while (current != nullptr) {
            std::cout << current->data << " -> ";
            current = current->next;
        }

        cout << "nullptr" << endl;
    }
};

int main() {
    LinkedList list;

    // Insert at the beginning
    list.insertAtBeginning(1);
    list.insertAtBeginning(0);

    // Insert in the middle
    list.insertInMiddle(2, 1);

    // Insert at the end
    list.insertAtEnd(3);
```

```
    // Display the linked list
    list.display();

    return 0;
}
```

Output:

```
0 -> 2 -> 1 -> 3 -> nullptr
```

Deletion from Linked List:

Let's go through the algorithms for three common scenarios:

1. **Deletion at the beginning,**
2. **Deletion at the end of the linked list, and**
3. **Deletion after a specified node.**

1. **Deletion at the Beginning of the Linked List:**

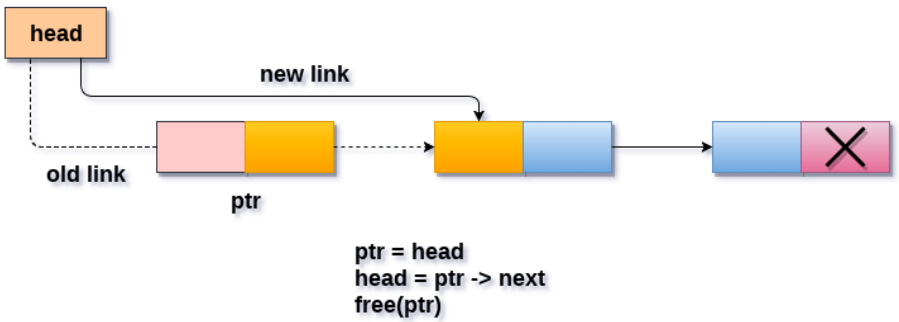
Algorithm:

1. Set the head of the linked list to the next node.
2. Free the memory of the old head node if necessary.

Example: Initial Linked List: **1 -> 2 -> 3 -> 4**

After deletion at the beginning:

2 -> 3 -> 4



Deleting a node from the beginning

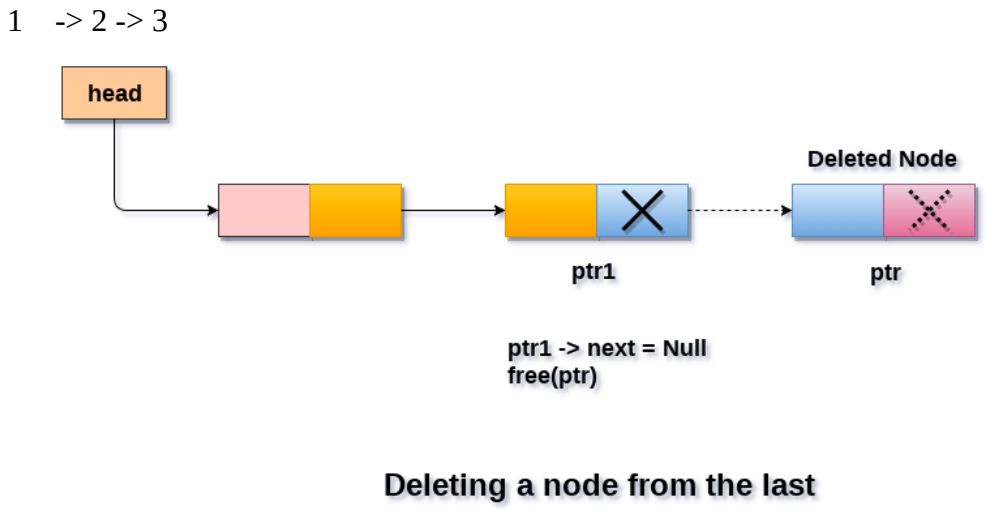
2. **Deletion at the End of the Linked List:**

Algorithm:

1. Traverse the linked list until you reach the second-to-last node.
2. Update the **next** pointer of the second-to-last node to **null**.
3. Free the memory of the last node if necessary.

Example: Initial Linked List: **1 -> 2 -> 3 -> 4**

After deletion at the end



3. **Deletion After a Specified Node:**

Algorithm:

1. Traverse the linked list until you find the specified node.
2. Update the **next** pointer of the specified node to skip the next node.
3. Free the memory of the deleted node if necessary.

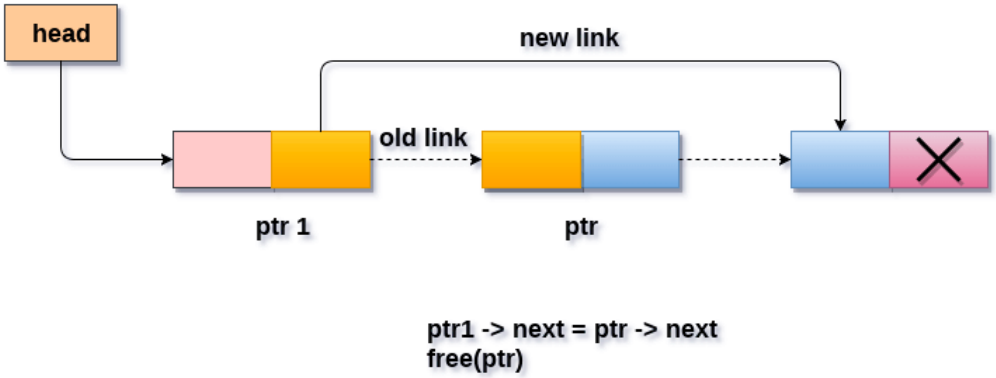
Example: Initial Linked List: **1 -> 2 -> 3 -> 4**

If you want to delete after node with value **2**:

1 -> 2 -> 4

If you want to delete after node with value **3**:

1 -> 2 -> 3



Deletion a node from specified position

These are the basic algorithms for performing these types of deletions in a linked list. Be sure to handle edge cases, such as checking if the list is empty or if the specified node exists before performing deletions, below are C++ programs for each of the three common scenarios: deletion at the beginning, deletion at the end of the linked list, and deletion after a specified node.

C++ Program: Here's a C++ program that implements the three deletion operations on a singly linked list:

```
#include <iostream>

// Define a Node structure for the linked list
struct Node {
    int data;
    Node* next;

    Node(int val) : data(val), next(nullptr) {}
};

class LinkedList {
private:
    Node* head;
```

```
public:
    LinkedList() : head(nullptr) {}

    // Function to insert a node at the beginning of the linked list
    void insertAtBeginning(int val) {
        Node* newNode = new Node(val);
        newNode->next = head;
        head = newNode;
    }

    // Function to delete a node at the beginning of the linked list
    void deleteAtBeginning() {
        if (head == nullptr) {
            std::cout << "The list is empty. Cannot delete at the
beginning.\n";
            return;
        }

        Node* temp = head;
        head = head->next;
        delete temp;
    }

    // Function to delete a node at the end of the linked list
    void deleteAtEnd() {
        if (head == nullptr) {
            std::cout << "The list is empty. Cannot delete at the end.\n";
            return;
        }

        if (head->next == nullptr) {
            // If there is only one node, delete it and set head to nullptr
            delete head;
            head = nullptr;
            return;
        }

        Node* prev = nullptr;
        Node* curr = head;

        while (curr->next != nullptr) {
            prev = curr;
            curr = curr->next;
        }

        prev->next = nullptr;
        delete curr;
    }

    // Function to delete a node after a specified node with value 'key'
    void deleteAfter(int key) {
        if (head == nullptr) {
            std::cout << "The list is empty. Cannot delete after a specified
node.\n";
            return;
        }
```

```
        Node* curr = head;

        while (curr != nullptr && curr->data != key) {
            curr = curr->next;
        }

        if (curr == nullptr || curr->next == nullptr) {
            std::cout << "Node with value " << key << " not found or it is
the last node.\n";
            return;
        }

        Node* temp = curr->next;
        curr->next = temp->next;
        delete temp;
    }

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

int main() {
    LinkedList list;

    // Insert elements at the beginning
    list.insertAtBeginning(4);
    list.insertAtBeginning(3);
    list.insertAtBeginning(2);
    list.insertAtBeginning(1);

    std::cout << "Initial Linked List: ";
    list.printList();

    // Delete at the beginning
    list.deleteAtBeginning();
    std::cout << "After deletion at the beginning: ";
    list.printList();

    // Delete at the end
    list.deleteAtEnd();
    std::cout << "After deletion at the end: ";
    list.printList();

    // Delete after a specified node (e.g., value 2)
    list.deleteAfter(2);
    std::cout << "After deletion after node with value 2: ";
    list.printList();

    // Delete after a specified node (e.g., value 3)
    list.deleteAfter(3);
```

```
    std::cout << "After deletion after node with value 3: ";
    list.printList();

    return 0;
}
```

```
Initial Linked List: 1 -> 2 -> 3 -> 4 -> nullptr
After deletion at the beginning: 2 -> 3 -> 4 -> nullptr
After deletion at the end: 2 -> 3 -> nullptr
After deletion after node with value 2: 2 -> nullptr
Node with value 3 not found or it is the last node.
After deletion after node with value 3: 2 -> nullptr
```

Two way Linked List

A two-way linked list, also known as a doubly linked list, is a data structure used in computer programming to organize and store a collection of elements called nodes. Each node in a doubly linked list contains two references or pointers: one to the next node in the list (forward pointer) and one to the previous node in the list (backward pointer). This structure allows for efficient traversal in both directions, forward and backward.

Studying a doubly linked list (or two-way linked list) is important because it offers advantages and features that are not available in a singly linked list. Here are some reasons why studying doubly linked lists is valuable when compared to single linked lists:

1. **Bidirectional Traversal:** Doubly linked lists allow you to traverse the list in both forward and backward directions. In contrast, singly linked lists only support forward traversal. This bidirectional traversal can be essential in scenarios where you need to navigate data in reverse order efficiently.

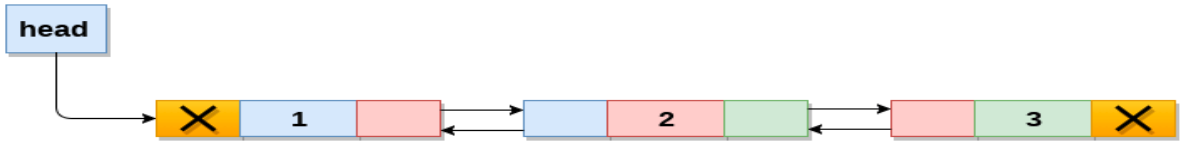
2. **Efficient Deletion:** Deleting a node from a doubly linked list is more efficient than in a singly linked list, especially when you need to remove a node from the middle of the list. With a doubly linked list, you can access the previous node directly, making deletion faster and simpler.

3. **Insertions and Deletions at Both Ends:** Doubly linked lists make it easy to insert and delete nodes at both the beginning and end of the list. This can be useful in implementing various data structures and algorithms efficiently.
4. **Implementation of Advanced Data Structures:** Doubly linked lists are fundamental in the implementation of more complex data structures like stacks, queues, and doubly ended queues (dequeues). These data structures often require the ability to manipulate elements at both ends of the list, which is conveniently supported by doubly linked lists.
5. **Backtracking and Undo Operations:** In applications like text editors, where you may need to implement features like undo and redo, doubly linked lists are helpful. You can maintain a history of states and easily backtrack or redo operations by traversing the list in both directions.
6. **Memory Overhead:** While doubly linked lists require more memory due to the additional backward pointers, this trade-off can be acceptable in situations where efficient backward traversal or deletion operations are essential. In contrast, singly linked lists have lower memory overhead but lack the advantages mentioned above.
7. **Learning and Problem Solving:** Studying doubly linked lists helps improve problem-solving skills and understanding of data structures. It expands your knowledge beyond the basics of singly linked lists and prepares you for more advanced data structures and algorithms.

Let's explain a doubly linked list with a simple example and provide a C++ program to demonstrate its implementation.

Example:

Imagine you want to create a doubly linked list to store a list of integers. Each node will contain a integer and two pointers: one pointing to the next node and one pointing to the previous node. This structure will allow you to easily traverse the list both forwards and backwards.



Doubly Linked List

C++ Program:

Here's a basic C++ program that defines a doubly linked list for names:

```
#include <iostream>
#include <string>

using namespace std;

// Define a Node structure for the doubly linked list
struct Node {
    string name;
    Node* next;
    Node* prev;

    Node(const string& n) : name(n), next(nullptr), prev(nullptr) {}
};

class DoublyLinkedList {
private:
    Node* head;
    Node* tail;

public:
    DoublyLinkedList() : head(nullptr), tail(nullptr) {}

    // Function to add a new node at the end of the list
    void append(const string& name) {
        Node* newNode = new Node(name);
        if (!head) {
            head = newNode;
            tail = newNode;
```

```
        } else {
            newNode->prev = tail;
            tail->next = newNode;
            tail = newNode;
        }
    }

    // Function to display the list forwards
    void displayForward() {
        Node* current = head;
        while (current) {
            cout << current->name << " ";
            current = current->next;
        }
        cout << endl;
    }

    // Function to display the list backwards
    void displayBackward() {
        Node* current = tail;
        while (current) {
            cout << current->name << " ";
            current = current->prev;
        }
        cout << endl;
    }
};

int main() {
    DoublyLinkedList namesList;

    namesList.append("1");
    namesList.append("2");
    namesList.append("3");
    namesList.append("4");

    cout << " Integers Forward: ";
    namesList.displayForward();
```

```
    cout << " Integers Backward: ";
    namesList.displayBackward();

    return 0;
}
```

Output:

```
Integers Forward: 1 2 3 4
Integers Backward: 4 3 2 1
```

Code Explanation:

The provided C++ code demonstrates a simple example of a doubly linked list, a data structure used to store and manage a list of items efficiently. Here's a simplified explanation:

1. We have a list of names: "1," "Bob," "Charlie," and "David."
2. The program defines a structure called **Node** to represent each item in the list. Each **Node** contains the name, a pointer to the next item (forward), and a pointer to the previous item (backward).
3. We create a **DoublyLinkedList** class to manage the list. It has functions to add names to the end of the list and display the list in both forward and backward orders.
4. In the **main** function:
    - We create an empty doubly linked list.
    - Add the names to the list one by one (e.g., "Alice," "Bob").
    - Then, we display the list forwards ("1 2 3 4 ") and backward ("4 3 2 1").

So, this code demonstrates how a doubly linked list allows you to store and traverse a list of items in both directions efficiently.