Linked list Deletion

📘 Linked List Deletion (Complete Detailed Notes)

🧩 What is Deletion in a Linked List?

Deletion in a linked list means removing a node from the list and ensuring that the **links between the remaining nodes** are maintained properly.

Unlike arrays (where deletion shifts elements), in a linked list we just **rearrange pointers** and **free memory** of the removed node.

📂 Node Structure

In C:

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

In C++:

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

⚙️ Types of Deletion in Linked List

1. **Deletion at Beginning**

2. **Deletion at End**
3. **Deletion at a Given Position (or by Value)**

🧩 1️⃣ Deletion at the Beginning

📘 Concept

We remove the first node (head) and make the next node the new head.

Before Deletion:

```
head → [10 | next] → [20 | next] → [30 | next] → NULL
```

After deleting first node:

```
head → [20 | next] → [30 | next] → NULL
```

🧱 Algorithm

1. Check if list is empty → if yes, print "List is empty".
2. Store current head in a temporary pointer.
3. Move head to `head→next`.
4. Free the temporary pointer (in C) / delete (in C++).

💻 C Code

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

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

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

```
        return;
    }

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

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

int main() {
    struct Node* head = NULL;

    // Create list manually
    struct Node* first = (struct Node*)malloc(sizeof(struct Node));
    struct Node* second = (struct Node*)malloc(sizeof(struct Node));
    struct Node* third = (struct Node*)malloc(sizeof(struct Node));

    first->data = 10; first->next = second;
    second->data = 20; second->next = third;
    third->data = 30; third->next = NULL;
    head = first;

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

    deleteAtBeginning(&head);

    printf("After Deletion at Beginning: ");
    printList(head);

    return 0;
}
```

💻 C++ Code

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

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

void deleteAtBeginning(Node*& head) {
    if (head == nullptr) {
        cout << "List is empty." << endl;
        return;
    }

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

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

int main() {
    Node* head = new Node{10, new Node{20, new Node{30, nullptr}}};

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

    deleteAtBeginning(head);
```

```
    cout << "After Deletion at Beginning: ";
    printList(head);

    return 0;
}
```

🧩2️⃣ Deletion at the End

📘 Concept

We traverse the list to the **second last node**, then make its `next` pointer `NULL`, and delete the last node.

Before:

```
10 → 20 → 30 → NULL
```

After Deletion:

```
10 → 20 → NULL
```

🧱 Algorithm

1. If list is empty → print “List is empty”.
2. If list has only one node → delete it and set head = NULL.
3. Traverse until the **second last node**.
4. Delete the **last node** and make `second_last->next = NULL`.

💻 C Code

```
void deleteAtEnd(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;
}
```

💻 C++ Code

```
void deleteAtEnd(Node*& head) {
    if (head == nullptr) {
        cout << "List is empty." << endl;
        return;
    }

    if (head->next == nullptr) {
        delete head;
        head = nullptr;
        return;
    }

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

    delete temp->next;
    temp->next = nullptr;
}
```

🧩3️⃣ Deletion at a Given Position

📘 Concept

We remove a node at a particular **position** (1-based index).

Example:

```
10 → 20 → 30 → 40 → NULL
Delete position 3
After: 10 → 20 → 40 → NULL
```

🧱 Algorithm

1. If list is empty → print "List is empty".
2. If position = 1 → call `deleteAtBeginning`.
3. Traverse to the (pos - 1)th node.
4. Change links:
   - `temp->next = temp->next->next`
5. Delete the node to be removed.

💻 C Code

```
void deleteAtPosition(struct Node** head, int position) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = *head;

    // Case 1: Delete first node
    if (position == 1) {
        *head = temp->next;
        free(temp);
        return;
    }
```

```
    // Traverse to (pos-1)th node
    for (int i = 1; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
    }

    if (temp == NULL || temp->next == NULL) {
        printf("Position out of range.\n");
        return;
    }

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

💻 C++ Code

```
void deleteAtPosition(Node*& head, int position) {
    if (head == nullptr) {
        cout << "List is empty." << endl;
        return;
    }

    Node* temp = head;

    // Case 1: Delete first node
    if (position == 1) {
        head = temp->next;
        delete temp;
        return;
    }

    // Traverse to (pos-1)th node
    for (int i = 1; i < position - 1 && temp != nullptr; i++) {
        temp = temp->next;
    }

    if (temp == nullptr || temp->next == nullptr) {
```

```
        cout << "Position out of range." << endl;
        return;
    }

    Node* nodeToDelete = temp->next;
    temp->next = nodeToDelete->next;
    delete nodeToDelete;
}
```

🧮 Time & Space Complexity Analysis

Operation	Time Complexity	Space Complexity	Reason
Deletion at Beginning	O(1)	O(1)	Only head change
Deletion at End	O(n)	O(1)	Traverse till second last node
Deletion at Position	O(n)	O(1)	Traverse till given position

🧠 Why `*pointer` in C but not in C++?

- In **C**, arguments are **passed by value**, so changes to `head` inside a function don't affect the original variable.

  → We pass `&head` → function uses `struct Node **head`.

- In **C++**, we can use **reference (`&`)**.

  → `Node*& head` directly modifies the original head pointer — no need for double pointer.

✅ Final Summary

Operation	C Function	C++ Function	Time Complexity	Description
Delete at Beginning	`deleteAtBeginning(struct Node** head)`	`deleteAtBeginning(Node*& head)`	O(1)	Removes first node
Delete at End	`deleteAtEnd(struct Node** head)`	`deleteAtEnd(Node*& head)`	O(n)	Removes last node

Operation	C Function	C++ Function	Time Complexity	Description
Delete at Position	`deleteAtPosition(struct Node** head, int pos)`	`deleteAtPosition(Node*& head, int pos)`	O(n)	Removes node at given position

📖 Key Points

- Always **free() / delete** the removed node to avoid memory leaks.
- Check for **NULL conditions** before accessing `next`.
- Handle **special cases** like empty list or single-node list carefully.
- Deletion does **not require additional space**, so space complexity is always **O(1)**.

Would you like me to **combine both "Insertion" and "Deletion" documents** into a **single, professionally formatted PDF** (with code blocks, diagrams, and headings for easy revision)?