1. **Creation of Node**
2. **Traversal (Print list)**
3. **Insertion**
    - At Beginning
    - At End
    - At Given Position
4. **Searching**
5. **Deletion**
    - From Beginning
    - From End
    - From Given Position

-

📘 Singly Linked List Operations

🔹 1. Creating a Node

A singly linked list node contains:

- **data** (value)
- **next** (pointer to next node)

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

Creating the first node:

```
Node* head = new Node();
head->data = 10;
head->next = NULL;
```

🔹 2. Traversal (Printing the List)

👉 Traverse from `head` to `NULL`, printing data.

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

🔹 3. Insertion Operations

(a) Insert at Beginning

- Create new node.
- Point its `next` to current head.
- Update head to new node.

```
void insertAtBeginning(Node*& head, int newData) {
    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = head;
    head = newNode;
}
```

(b) Insert at End

- Create new node with `next = NULL`.
- Traverse to last node.
- Update last node's `next` to new node.

```
void insertAtEnd(Node*& head, int newData) {
    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = NULL;

    if (head == NULL) {
        head = newNode;
        return;
    }

    Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}
```

(c) Insert at Given Position

- Traverse to `(pos - 1)`th node.
- Link new node in between.

```
void insertAtPosition(Node*& head, int newData, int position) {
    Node* newNode = new Node();
    newNode->data = newData;

    if (position == 1) {  // insert at beginning
        newNode->next = head;
        head = newNode;
        return;
    }

    Node* temp = head;
    for (int i = 1; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
    }

    if (temp == NULL) {
        cout << "Position out of range!" << endl;
        delete newNode;
        return;
    }

    newNode->next = temp->next;
    temp->next = newNode;
}
```

🔹 4. Searching

👉 Traverse list, compare each node with `key`.

```
bool search(Node* head, int key) {
    Node* temp = head;
    while (temp != NULL) {
        if (temp->data == key) return true;
        temp = temp->next;
    }
    return false;
}
```

🔹 5. Deletion Operations

(a) Delete from Beginning

- Move `head` to `head->next`.
- Free old head.

```
void deleteFromBeginning(Node*& head) {
    if (head == NULL) return;
    Node* temp = head;
    head = head->next;
    delete temp;
}
```

(b) Delete from End

- Traverse to second-last node.
- Set its `next = NULL`.
- Free last node.

```
void deleteFromEnd(Node*& head) {
    if (head == NULL) return;
    if (head->next == NULL) { // only one node
        delete head;
        head = NULL;
        return;
    }

    Node* temp = head;
    while (temp->next->next != NULL) {
        temp = temp->next;
    }
    delete temp->next;
    temp->next = NULL;
}
```

(c) Delete from Given Position

- Traverse to `(pos - 1)`th node.
- Skip the target node and free memory.

```
void deleteFromPosition(Node*& head, int position) {
    if (head == NULL) return;

    if (position == 1) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return;
    }

    Node* temp = head;
    for (int i = 1; i < position - 1 && temp->next != NULL; i++) {
        temp = temp->next;
    }

    if (temp->next == NULL) {
        cout << "Position out of range!" << endl;
        return;
    }

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

✅ Example Usage

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

    // Insert elements
    insertAtBeginning(head, 30);
    insertAtBeginning(head, 20);
    insertAtEnd(head, 40);
    insertAtPosition(head, 25, 2);

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

    // Search
    cout << (search(head, 25) ? "Found" : "Not Found") << endl;

    // Delete operations
    deleteFromBeginning(head);
    deleteFromEnd(head);
    deleteFromPosition(head, 2);

    cout << "After Deletions: ";
    printList(head);

    return 0;
}
```

🔎 Sample Output

```
Linked List: 20 -> 25 -> 30 -> 40 -> NULL
Found
After Deletions: 25 -> NULL
```

✅ Full C++ Code for Singly Linked List Operations

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

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

// Print the linked list
void printList(Node* head) {
    Node* temp = head;
    while (temp != NULL) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
    cout << "NULL" << endl;
}

// Insert at beginning
void insertAtBeginning(Node*& head, int newData) {
    Node* newNode = new Node();
    newNode->data = newData;
    newNode->next = head;
    head = newNode;
}

// Insert at end
void insertAtEnd(Node*& head, int newData) {
    Node* newNode = new Node();
    newNode->data = newData;
```

```
    newNode->next = NULL;

    if (head == NULL) {
        head = newNode;
        return;
    }

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

// Insert at a given position
void insertAtPosition(Node*& head, int newData, int position) {
    Node* newNode = new Node();
    newNode->data = newData;

    if (position == 1) {
        newNode->next = head;
        head = newNode;
        return;
    }

    Node* temp = head;
    for (int i = 1; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
    }

    if (temp == NULL) {
        cout << "Position out of range!" << endl;
        delete newNode;
        return;
    }

    newNode->next = temp->next;
    temp->next = newNode;
```

```
}

// Search in linked list
bool search(Node* head, int key) {
    Node* temp = head;
    while (temp != NULL) {
        if (temp->data == key) return true;
        temp = temp->next;
    }
    return false;
}

// Delete from beginning
void deleteFromBeginning(Node*& head) {
    if (head == NULL) return;
    Node* temp = head;
    head = head->next;
    delete temp;
}

// Delete from end
void deleteFromEnd(Node*& head) {
    if (head == NULL) return;
    if (head->next == NULL) {
        delete head;
        head = NULL;
        return;
    }

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

// Delete from a given position
```

```
void deleteFromPosition(Node*& head, int position) {
    if (head == NULL) return;

    if (position == 1) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return;
    }

    Node* temp = head;
    for (int i = 1; i < position - 1 && temp->next != NULL; i++) {
        temp = temp->next;
    }

    if (temp->next == NULL) {
        cout << "Position out of range!" << endl;
        return;
    }

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

int main() {
    Node* head = NULL;

    cout << "Inserting at beginning..." << endl;
    insertAtBeginning(head, 30);
    insertAtBeginning(head, 20);
    insertAtBeginning(head, 10);
    printList(head);

    cout << "\nInserting at end..." << endl;
    insertAtEnd(head, 40);
    insertAtEnd(head, 50);
    printList(head);
```

```
    cout << "\nInserting at position 3..." << endl;
    insertAtPosition(head, 25, 3);
    printList(head);

    cout << "\nSearching 25..." << endl;
    cout << (search(head, 25) ? "Found" : "Not Found") << endl;

    cout << "\nDeleting from beginning..." << endl;
    deleteFromBeginning(head);
    printList(head);

    cout << "\nDeleting from end..." << endl;
    deleteFromEnd(head);
    printList(head);

    cout << "\nDeleting from position 2..." << endl;
    deleteFromPosition(head, 2);
    printList(head);

    return 0;
}
```

🔎 Sample Output

```
Inserting at beginning...
10 -> 20 -> 30 -> NULL

Inserting at end...
10 -> 20 -> 30 -> 40 -> 50 -> NULL

Inserting at position 3...
10 -> 20 -> 25 -> 30 -> 40 -> 50 -> NULL

Searching 25...
Found
```

```
Deleting from beginning...
20 -> 25 -> 30 -> 40 -> 50 -> NULL

Deleting from end...
20 -> 25 -> 30 -> 40 -> NULL

Deleting from position 2...
20 -> 30 -> 40 -> NULL
```

📖 Summary of Operations

Operation	Time Complexity	Space
Insert at Beginning	O(1)	O(1)
Insert at End	O(n)	O(1)
Insert at Position	O(n)	O(1)
Search	O(n)	O(1)
Delete from Beginning	O(1)	O(1)
Delete from End	O(n)	O(1)
Delete from Position	O(n)	O(1)
