1. Circularly Linked Lists

Concept (Explanation)

A **Circular Linked List (CLL)** is a type of linked list where the last node points back to the first node instead of pointing to NULL. This creates a loop or circle structure. It is useful in applications where data needs to be accessed repeatedly in cycles. Unlike linear linked lists, there is no clear "end" node. Traversal can start from any node and continue indefinitely. It is memory efficient for cyclic processes like CPU scheduling. However, care must be taken to avoid infinite loops during traversal.

Syntax (C++)

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

Node* head = NULL;
```

Example (Creation & Traversal)

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

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

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

   head->next = second;
   second->next = third;
   third->next = head; // Circular link

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

```
    return 0;
}
```

---

2. Reversing a Linked List

Concept (Explanation)

Reversing a linked list means changing the direction of links between nodes so that the last node becomes the first. In a singly linked list, each node points to the next node, so reversing requires adjusting pointers carefully. This operation is important in many algorithms and interview questions. It can be done iteratively or recursively. The process involves maintaining previous, current, and next pointers. After reversal, the head pointer must be updated.

Syntax (C++)

```
Node* reverse(Node* head);
```

Example (Iterative Reversal)

```
Node* reverse(Node* head) {
    Node* prev = NULL;
    Node* current = head;
    Node* next = NULL;

    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    return prev;
}
```

---

3. Recursion

Concept (Explanation)

Recursion is a technique where a function calls itself to solve a problem. It breaks a large problem into smaller subproblems of the same type. Every recursive function must have a **base case** to stop the recursion. Without a base case, the function will run infinitely. Recursion is commonly used in tree traversal, searching, and sorting algorithms. It simplifies code but may

use more memory due to function call stack. Understanding recursion is key for advanced algorithms.

Syntax (C++)

```
void functionName() {
   if (base_condition)
      return;
   functionName(); // recursive call
}
```

Example (Factorial)

```
int factorial(int n) {
   if (n == 0)
      return 1;
   return n * factorial(n - 1);
}
```

---

4. Linear Recursion

Concept (Explanation)

Linear recursion occurs when a function makes **only one recursive call** in each step. The function reduces the problem size gradually until it reaches the base case. It forms a straight chain of function calls. This is the simplest form of recursion. Examples include factorial, sum of numbers, and linked list traversal. It is easy to understand and implement. However, it may still consume stack memory for large inputs.

Syntax (C++)

```
void linearRec(int n) {
   if (n == 0) return;
   linearRec(n - 1);
}
```

Example (Sum of Natural Numbers)

```
int sum(int n) {
```

```
    if (n == 0)
        return 0;
    return n + sum(n - 1);
}
```

---

5. Binary Recursion

Concept (Explanation)

Binary recursion occurs when a function makes **two recursive calls** in each step. This creates a tree-like structure of function calls. It is commonly used in problems like Fibonacci and tree traversals. The number of function calls increases rapidly. This type of recursion is less efficient due to repeated calculations. However, it is useful for understanding recursion trees. Optimization techniques like memoization can improve performance.

Syntax (C++)

```
void binaryRec(int n) {
    if (n == 0) return;
    binaryRec(n - 1);
    binaryRec(n - 1);
}
```

Example (Fibonacci)

```
int fib(int n) {
    if (n <= 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}
```

---

6. Multiple Recursion

Concept (Explanation)

Multiple recursion is a general form where a function makes **more than one recursive call** (not limited to two). It is used in complex problems like backtracking and combinatorial algorithms. Examples include generating permutations or solving puzzles like Tower of Hanoi. It creates multiple branches in recursion tree. It can be computationally expensive. Proper base conditions are critical to avoid infinite recursion. It is powerful but must be used carefully.

Syntax (C++)

```
void multiRec(int n) {
   if (n == 0) return;
   multiRec(n - 1);
   multiRec(n - 2);
   multiRec(n - 3);
}
```

Example (Simple Multi-Call)

```
void printPattern(int n) {
   if (n <= 0) return;
   cout << n << " ";
   printPattern(n - 1);
   printPattern(n - 2);
}
```

---

7. Exercises

Exercise 1: Circular Linked List Traversal

Write a program to create a circular linked list and display all elements.

Exercise 2: Reverse Linked List

Write a function to reverse a singly linked list and print it.

Exercise 3: Recursive Factorial

Write a recursive function to calculate factorial of a number.

Exercise 4: Fibonacci Series

Print first `n` Fibonacci numbers using recursion.

Exercise 5: Sum using Recursion

Find sum of first `n` natural numbers using recursion.

Exercise 6: Challenge Problem

Reverse a linked list using **recursion instead of loop**.

Summary

- Circular Linked List → Last node connects to first
- Reversal → Change pointer direction
- Recursion → Function calls itself
- Linear → One call
- Binary → Two calls
- Multiple → More than two calls
- Important for **interviews + algorithm design**