Binary Tree to CDLL in C++

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

class Node {
public:
    int data;
    Node* left;
    Node* right;

    Node(int data) {
        this->data = data;
        this->left = nullptr;
        this->right = nullptr;
    }
};

class BinartTree2CDLL {
public:
    // Function to concatenate two circular doubly linked lists
    Node* concatenate(Node* H1, Node* H2) {
        if (H1 == nullptr) return H2;
        if (H2 == nullptr) return H1;

        Node* T1 = H1->left;
        Node* T2 = H2->left;

        T1->right = H2;
        H2->left = T1;

        T2->right = H1;
        H1->left = T2;

        return H1;
    }

    // Function to convert binary tree into circular doubly linked list
    Node* bTreeToClist(Node* root) {
        if (root == nullptr) return nullptr;

        Node* l = bTreeToClist(root->left);
        Node* r = bTreeToClist(root->right);

        root->left = root->right = root;

        Node* result = concatenate(concatenate(l, root), r);

        return result;
    }

    // Function to print the circular doubly linked list
    void printCList(Node* head) {
        if (head == nullptr) return;

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

Your code to convert a **Binary Tree to a Circular Doubly Linked List (CDLL)** is **elegant and correct**. You're using **in-order traversal** with recursive linking, which is the standard and efficient approach. Let's break it down with a **dry run + visual table** using the tree:

```
      1
     / \
    2   3
   / \   \
  4   5   6
```

🔁 Step-by-Step Dry Run (In-order traversal)

Traversal order: **4 → 2 → 5 → 1 → 3 → 6**

Call Stack Depth	Node Visited	Left CDLL	Right CDLL	Resulting CDLL
1	4	null	null	4
1	5	null	null	5
2	2	4	5	4 ⇄ 2 ⇄ 5
1	6	null	null	6
2	3	null	6	3 ⇄ 6
3 (root)	1	4 ⇄ 2 ⇄ 5	3 ⇄ 6	4 ⇄ 2 ⇄ 5 ⇄ 1 ⇄ 3 ⇄ 6

- ⇄ means CDLL bidirectional links.
- At each recursive return, you concatenate left CDLL, root (self-circular), and right CDLL.

✅ Output

Circular Doubly Linked List:
4 2 5 1 3 6

```
        cout << endl;
    }
};

// Main method to test the bTreeToClist function
int main() {
    BinartTree2CDLL solution;

    // Creating a sample binary tree:
    //      1
    //     / \
    //    2   3
    //   / \   \
    //  4   5   6
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    root->right->right = new Node(6);

    Node* head = solution.bTreeToClist(root);

    cout << "Circular Doubly Linked List:" << endl;
    solution.printCList(head);

    // Clean up memory
    // In a real-world scenario, you would implement a function to delete the tree nodes.
    // For brevity, memory cleanup is not shown in this example.

    return 0;
}
```

Output:-

Circular Doubly Linked List:
4 2 5 1 3 6

Binary Tree to LL in C++

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

// TreeNode class definition
class TreeNode {
public:
    int val;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int x) {
        val = x;
        left = nullptr;
        right = nullptr;
    }
};

class FlattenBinaryTreeToLinkedList {
public:
    void flatten(TreeNode* root) {
        flattenHelper(root);
    }

private:
    TreeNode* flattenHelper(TreeNode* root) {
        if (root == nullptr) return nullptr;

        TreeNode* leftTail = flattenHelper(root->left);
        TreeNode* rightTail = flattenHelper(root->right);

        if (leftTail != nullptr) {
            leftTail->right = root->right;  // Connect the end of the left subtree to the start of the right subtree
            root->right = root->left;       // Move the left subtree to the right
            root->left = nullptr;           // Nullify the left pointer
        }

        // Return the last node in the flattened tree
        if (rightTail != nullptr) {
            return rightTail;
        } else if (leftTail != nullptr) {
            return leftTail;
        } else {
            return root;
        }
    }

public:
    // Utility function to print the flattened tree
    void printFlattenedTree(TreeNode* root) {
        while (root != nullptr) {
            cout << root->val << " ";
            root = root->right;
        }
        cout << endl;
    }
```

Absolutely! Let's dry run your `flatten` function **with a step-by-step table**, using this binary tree:

```
        1
       / \
      2   5
     / \   \
    3   4   6
```

The goal is to flatten this tree into a **linked list using right pointers** in **pre-order traversal**:

```
1 -> 2 -> 3 -> 4 -> 5 -> 6
```

🔁 Dry Run Table:

Step	Node Visited	Left Subtree Tail	Right Subtree Tail	Action Taken	Resulting Right Chain (Partial)
1	3	`nullptr`	`nullptr`	Leaf node → return `3`	3
2	4	`nullptr`	`nullptr`	Leaf node → return `4`	4
3	2	3	4	Move left to right: 2->left becomes nullptr, 2->right = 3, 3->right = 4	2 → 3 → 4
4	6	`nullptr`	`nullptr`	Leaf node → return `6`	6
5	5	`nullptr`	6	No left to move → do nothing, return 6	5 → 6
6	1	4 (tail of 2)	6 (tail of 5)	Move left to right: 1->right = 2, 4->right = 5 (attach 5 to end	1 → 2 → 3 → 4 → 5 → 6

```
    // Function to delete a binary tree to free memory
    void deleteTree(TreeNode* root) {
        if (root == nullptr) return;
        deleteTree(root->left);
        deleteTree(root->right);
        delete root;
    }
};

int main() {
    FlattenBinaryTreeToLinkedList solution;

    // Creating a sample binary tree:
    //     1
    //    / \
    //   2   5
    //  / \   \
    // 3   4   6
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(5);
    root->left->left = new TreeNode(3);
    root->left->right = new TreeNode(4);
    root->right->right = new TreeNode(6);

    cout << "Original Tree:" << endl;
    solution.printFlattenedTree(root); // This will just print the root node, as the tree is not flattened yet

    solution.flatten(root);

    cout << "Flattened Tree:" << endl;
    solution.printFlattenedTree(root);

    // Clean up memory
    solution.deleteTree(root);

    return 0;
}
```

				of 2 chain)	

🔚 Final Result:

The flattened tree is:

`1 → 2 → 3 → 4 → 5 → 6 → nullptr`

Output:-

`1 → 2 → 3 → 4 → 5 → 6 → nullptr`

CopyListwithRandomPointers in C++

```
#include <iostream>
#include <unordered_map>

// Definition for a Node.
struct Node {
    int val;
    Node* next;
    Node* random;

    Node(int _val) {
        val = _val;
        next = nullptr;
        random = nullptr;
    }
};

Node* copyRandomList(Node* head) {
    if (head == nullptr) return nullptr;

    std::unordered_map<Node*, Node*> map;
    Node* curr = head;

    // First pass: create all nodes and store them in the
map.
    while (curr != nullptr) {
        map[curr] = new Node(curr->val);
        curr = curr->next;
    }

    // Second pass: assign next and random pointers.
    curr = head;
    while (curr != nullptr) {
        map[curr]->next = map[curr->next];
        map[curr]->random = map[curr->random];
        curr = curr->next;
    }

    return map[head];
}

void printList(Node* head) {
    while (head != nullptr) {
        std::cout << "Node(" << head->val << ")";
        if (head->random != nullptr) {
            std::cout << " [Random(" << head->random-
>val << ")]";
        }
        std::cout << " -> ";
        head = head->next;
    }
    std::cout << "null" << std::endl;
}

int main() {
    Node* head = new Node(1);
    head->next = new Node(2);
    head->next->next = new Node(3);
    head->random = head->next->next;
    head->next->random = head;

    Node* result = copyRandomList(head);
```

Goal: Deep copy a linked list where each node has next and random pointers.

Given input:

```
1 -> 2 -> 3
|    |
v    v
3    1
```

📌 **Step-by-Step Dry Run Table**

Step	Operation	Affected Node	Explanation
First Pass	map[1] = new Node(1)	Node 1	Creates a copy of node 1
	map[2] = new Node(2)	Node 2	Creates a copy of node 2
	map[3] = new Node(3)	Node 3	Creates a copy of node 3
Second Pass	map[1]->next = map[2]	Node 1 copy	Sets next of copied 1 to copied 2
	map[1]->random = map[3]	Node 1 copy	Sets random of copied 1 to copied 3 (like original)
	map[2]->next = map[3]	Node 2 copy	Sets next of copied 2 to copied 3
	map[2]->random = map[1]	Node 2 copy	Sets random of copied 2 to copied 1
	map[3]->next = map[nullptr] = null	Node 3 copy	Last node, next is null
	map[3]->random = map[nullptr]	Node 3 copy	random was not set originally, stays null

✅ **Final Output:**

Copied list:

1 [Random(3)] -> 2 [Random(1)] -> 3 -> null

```
    printList(result);

    // Free the allocated memory
    Node* curr = result;
    while (curr != nullptr) {
        Node* temp = curr;
        curr = curr->next;
        delete temp;
    }

    return 0;
}
```

Output:-
0

Cycle in C++

```cpp
#include <iostream>

using namespace std;

// Definition of a Node in the linked list
struct Node {
    int val;
    Node* next;
     Node(int x) {
        val = x;        // Assigns the parameter x to the member variable val
        next = nullptr; // Initializes the next pointer to nullptr
    }
};

// Function to detect if there is a cycle in the linked list
bool hasCycle(Node* head) {
    if (head == nullptr || head->next == nullptr) {
        return false;
    }

    Node* slow = head;
    Node* fast = head;

    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;

        if (slow == fast) {
            return true; // Cycle detected
        }
    }

    return false; // No cycle found
}

int main() {
    // Creating a linked list: 1 -> 2 -> 3 -> 4 -> 5
    Node* head = new Node(1);
    head->next = new Node(2);
    head->next->next = new Node(3);
    head->next->next->next = new Node(4);
    head->next->next->next->next = new Node(5);

    // Creating a cycle by pointing the next of last node to the node with value 3 (index 2)
    Node* tail = head;
    while (tail->next != nullptr) {
        tail = tail->next;
    }
    Node* cycleNode = head->next->next; // Node with value 3
    tail->next = cycleNode;

    // Check if the cycle is present
    cout << (hasCycle(head) ? "Cycle is present" : "No cycle") << endl;

    return 0;
```

Core Logic Recap

Floyd's algorithm uses:

- slow: moves 1 step at a time.
- fast: moves 2 steps at a time.

If there's a cycle, slow and fast will eventually meet inside the loop.

🧪 Dry Run

Linked List:

```
1 -> 2 -> 3 -> 4 -> 5
          ^         |
          |_________|
```

Cycle: 5 -> 3 creates a loop back to node with value 3.

📜 Dry Run Table

Iteration	slow value	fast value	Notes
1	2	3	both moved: slow+1, fast+2
2	3	5	fast jumps into cycle
3	4	4	slow == fast → cycle found

🧠 Output:

Cycle is present

}	
Output:- Cycle is present	

MergeSort in C++

```cpp
#include <iostream>

using namespace std;

// Definition for a singly-linked list node
struct ListNode {
    int data;
    ListNode* next;

    ListNode(int x) {
        data = x;
        next = nullptr;
    }
};

// Function to merge two sorted linked lists
ListNode* merge(ListNode* h1, ListNode* h2) {
    if (h1 == nullptr) return h2;
    if (h2 == nullptr) return h1;

    ListNode* ans = nullptr;
    ListNode* t = nullptr;

    if (h1->data < h2->data) {
        ans = h1;
        t = h1;
        h1 = h1->next;
    } else {
        ans = h2;
        t = h2;
        h2 = h2->next;
    }

    while (h1 != nullptr && h2 != nullptr) {
        if (h1->data < h2->data) {
            t->next = h1;
            t = t->next;
            h1 = h1->next;
        } else {
            t->next = h2;
            t = t->next;
            h2 = h2->next;
        }
    }

    if (h1 != nullptr) t->next = h1;
    if (h2 != nullptr) t->next = h2;

    return ans;
}

// Function to find the middle of the linked list
ListNode* mid(ListNode* h) {
    ListNode* slow = h;
    ListNode* fast = h;

    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
    }
```

Dry Run — Function Calls Breakdown:

1. Initial Call:

`mergeSort(4 -> 2 -> 1 -> 3)`

Midpoint = 1 (list breaks into):

- h1 = 4 -> 2
- h2 = 1 -> 3

2. Recursive Breakdown:

Level	Call	Mid Node	Left Part	Right Part
1	mergeSort(4->2->1->3)	1	4->2	1->3
2	mergeSort(4->2)	2	4	2
2	mergeSort(1->3)	3	1	3

3. Merge Steps (Bottom-Up):

Step	Merge Call	Output
1	merge(4, 2)	2 -> 4
2	merge(1, 3)	1 -> 3
3	merge(2->4, 1->3)	1 -> 2 -> 3 -> 4

✔ Final Output:

```
Sorted Linked List: 1 -> 2 -> 3 -> 4
```

```
    return slow;
}

// Function to perform merge sort on the linked list
ListNode* mergeSort(ListNode* h1) {
    if (h1 == nullptr || h1->next == nullptr) return h1;

    ListNode* m = mid(h1);
    ListNode* h2 = m->next;
    m->next = nullptr;

    ListNode* t1 = mergeSort(h1);
    ListNode* t2 = mergeSort(h2);
    ListNode* t3 = merge(t1, t2);

    return t3;
}

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

int main() {
    // Creating an example linked list: 4 -> 2 -> 1 -> 3
    ListNode* head = new ListNode(4);
    head->next = new ListNode(2);
    head->next->next = new ListNode(1);
    head->next->next->next = new ListNode(3);

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

    head = mergeSort(head);

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

    // Clean up allocated memory
    ListNode* current = head;
    while (current != nullptr) {
        ListNode* next = current->next;
        delete current;
        current = next;
    }

    return 0;
}
```

Output:-
0

OddEven in C++

```cpp
#include <iostream>

using namespace std;

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

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

// LinkedList class definition
class LinkedList {
public:
    Node* head;
    Node* tail;
    int size;

    LinkedList() {
        head = nullptr;
        tail = nullptr;
        size = 0;
    }

    // Method to add a node at the end of the list
    void addLast(int val) {
        Node* newNode = new Node(val);
        if (size == 0) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            tail = newNode;
        }
        size++;
    }

    // Method to display the elements of the list
    void display() {
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }

    // Method to remove the first node from the list
    void removeFirst() {
        if (size == 0) {
            cout << "List is empty" << endl;
        } else if (size == 1) {
            head = tail = nullptr;
            size = 0;
        } else {
            head = head->next;
            size--;
```

Initial List:

Original List: 2 -> 8 -> 9 -> 1 -> 5 -> 4 -> 3

🔄 Dry Run Table for oddEven() Method

We'll track how elements are moved to either the **odd** or **even** list.

Step	Current Node (val)	Is Even?	Action	Odd List	Even List
1	2	✅ Yes	Add to Even		2
2	8	✅ Yes	Add to Even		2 -> 8
3	9	❌ No	Add to Odd	9	2 -> 8
4	1	❌ No	Add to Odd	9 -> 1	2 -> 8
5	5	❌ No	Add to Odd	9 -> 1 -> 5	2 -> 8
6	4	✅ Yes	Add to Even	9 -> 1 -> 5	2 -> 8 -> 4
7	3	❌ No	Add to Odd	9 -> 1 -> 5 -> 3	2 -> 8 -> 4

🧩 Reconnecting Lists

- Since **both odd and even lists exist**, we connect:
  - odd.tail->next = even.head
  - New head = odd.head
  - New tail = even.tail
  - New size = odd.size + even.size = 4 + 3 = 7

🟢 Result after oddEven():

List after Odd-Even Segregation: 9 -> 1 -> 5 -> 3 -> 2 -> 8 -> 4

➕ Add 10 at beginning, 100 at end:

- After addFirst(10): 10 -> 9 -> 1 -> 5 -> 3 -> 2 -> 8 -> 4
- After addLast(100): 10 -> 9 -> 1 -> 5 -> 3 -> 2 -> 8 -> 4 -> 100

✅ Final Output:

```
        }
    }

    // Method to get the data of the first node
    int getFirst() {
        if (size == 0) {
            cout << "List is empty" << endl;
            return -1;
        } else {
            return head->data;
        }
    }

    // Method to add a node at the beginning of the list
    void addFirst(int val) {
        Node* newNode = new Node(val);
        newNode->next = head;
        head = newNode;

        if (size == 0) {
            tail = newNode;
        }

        size++;
    }

    // Method to segregate odd and even nodes in the
list
    void oddEven() {
        LinkedList odd;
        LinkedList even;

        while (size > 0) {
            int val = getFirst();
            removeFirst();

            if (val % 2 == 0) {
                even.addLast(val);
            } else {
                odd.addLast(val);
            }
        }

        if (odd.size > 0 && even.size > 0) {
            odd.tail->next = even.head;
            head = odd.head;
            tail = even.tail;
            size = odd.size + even.size;
        } else if (odd.size > 0) {
            head = odd.head;
            tail = odd.tail;
            size = odd.size;
        } else if (even.size > 0) {
            head = even.head;
            tail = even.tail;
            size = even.size;
        }
    }
};

int main() {
    // Initialize LinkedList
```

List after adding 10 at the beginning and 100 at the end: 10 -> 9 -> 1 -> 5 -> 3 -> 2 -> 8 -> 4 -> 100

LinkedList l1;
 	
// Add elements to the LinkedList
l1.addLast(2);
l1.addLast(8);
l1.addLast(9);
l1.addLast(1);
l1.addLast(5);
l1.addLast(4);
l1.addLast(3);
 	
// Display original list
cout << "Original List: ";
l1.display();
 	
// Perform odd-even segregation
l1.oddEven();
 	
// Display list after odd-even segregation
cout << "List after Odd-Even Segregation: ";
l1.display();
 	
// Add elements at the beginning and end
int a = 10;
int b = 100;
l1.addFirst(a);
l1.addLast(b);
 	
// Display list after adding elements
cout << "List after adding " << a << " at the beginning and " << b << " at the end: ";
l1.display();
 	
return 0;
}	

Output:- List after adding 10 at the beginning and 100 at the end: 10 -> 9 -> 1 -> 5 -> 3 -> 2 -> 8 -> 4 -> 100	

Palindrome in C++

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

// Node class for the linked list
class Node {
public:
    int val;
    Node* next;

    Node(int val) {
        this->val = val;
        this->next = nullptr;
    }
};

// Function to find the middle node of the linked list
Node* midNode(Node* head) {
    if (head == nullptr || head->next == nullptr) return head;

    Node* slow = head;
    Node* fast = head;

    while (fast->next != nullptr && fast->next->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
    }

    return slow;
}

// Function to reverse a linked list
Node* reverseOfLL(Node* head) {
    if (head == nullptr || head->next == nullptr) return head;

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

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

    return prev;
}

// Function to check if a linked list is a palindrome
bool isPalindrome(Node* head) {
    if (head == nullptr || head->next == nullptr) return true;

    // Find the middle of the linked list
    Node* mid = midNode(head);

    // Reverse the second half of the list
    Node* nHead = mid->next;
```

Step-by-Step Dry Run Table

Step	Operation	Pointer/Variable	Value(s)
1	Find mid	`slow`, `fast`	Mid = 3 (slow stops here)
2	Reverse 2nd half	From node `2 -> 1`	Reversed to `1 -> 2`
3	Compare halves	`1-2-3` vs `1-2`	Matches fully
4	Restore 2nd half	Reverse back `1->2`	Back to `2->1`
5	Result		✅ `true` (Palindrome)

🧠 Output

```
true
```

<pre>   mid->next = nullptr; // Split the list into two halves
    nHead = reverseOfLL(nHead);
 	
    // Compare the two halves
    Node* c1 = head;
    Node* c2 = nHead;
 	
    bool res = true;
    while (c2 != nullptr) { // Only need to compare until
c2 ends
        if (c1->val != c2->val) {
            res = false;
            break;
        }
        c1 = c1->next;
        c2 = c2->next;
    }
 	
    // Restore the original list
    nHead = reverseOfLL(nHead);
    mid->next = nHead;
 	
    return res;
}
 	
// Function to create a linked list from an array of
integers
Node* createList(int values[], int n) {
    Node* dummy = new Node(-1);
    Node* prev = dummy;
    for (int i = 0; i < n; ++i) {
        prev->next = new Node(values[i]);
        prev = prev->next;
    }
    return dummy->next;
}
 	
int main() {
    // Hardcoding the linked list: 1 -> 2 -> 3 -> 2 -> 1
    int arr[] = {1, 2, 3, 2, 1};
    int n = sizeof(arr) / sizeof(arr[0]);
    Node* head = createList(arr, n);
 	
    // Checking if the linked list is a palindrome
    cout << boolalpha << isPalindrome(head) <<
endl; // should print true
 	
    return 0;
}</pre>	

Output:- true	

Reverse a LL in C++	

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

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

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

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

// Function to reverse the linked list recursively
Node* reverse(Node* head) {
    if (head == nullptr || head->next == nullptr) {
        return head;
    }
    Node* smallAns = reverse(head->next);
    head->next->next = head;
    head->next = nullptr;
    return smallAns;
}

// Function to reverse the linked list iteratively
Node* reverseI(Node* head) {
    if (head == nullptr || head->next == nullptr) {
        return head;
    }
    Node* prev = nullptr;
    Node* curr = head;
    Node* next = nullptr;
    while (curr != nullptr) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}

int main() {
    // Creating the linked list
    Node* one = new Node(1);
    Node* two = new Node(2);
    Node* three = new Node(3);
    Node* four = new Node(4);
```

Dry Run Table (Step-by-step Iteration)

Iteration	curr->data	next->data	prev->data	What Happens	List State
0	1	2	nullptr	Reverse 1->nullptr, move prev = 1, curr = 2	1
1	2	3	1	Reverse 2->1, move prev = 2, curr = 3	2 -> 1
2	3	4	2	Reverse 3->2, move prev = 3, curr = 4	3 -> 2 -> 1
3	4	5	3	Reverse 4->3, move prev = 4, curr = 5	4 -> 3 -> 2 -> 1
4	5	6	4	Reverse 5->4, move prev = 5, curr = 6	5 -> 4 -> 3 -> 2 -> 1
5	6	7	5	Reverse 6->5, move prev = 6, curr = 7	6 -> 5 -> 4 -> 3 -> 2 -> 1
6	7	nullptr	6	Reverse 7->6, move prev = 7, curr = nullptr	7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1

✓ Final Pointers:

- curr == nullptr → end of list
- prev == 7 → head of reversed list
- So, the function returns prev as the new head.

✓ Final Output:

List after iterative reversal: 7->6->5->4->3->2->1

```
    Node* five = new Node(5);
    Node* six = new Node(6);
    Node* seven = new Node(7);
    one->next = two;
    two->next = three;
    three->next = four;
    four->next = five;
    five->next = six;
    six->next = seven;

    // Displaying the original list
    cout << "Original List: ";
    display(one);

    // Reversing the list recursively
    cout << "List after recursive reversal: ";
    Node* revRec = reverse(one);
    display(revRec);

    // Reversing the list iteratively
    cout << "List after iterative reversal: ";
    Node* revIter = reverseI(revRec);
    display(revIter);

    // Deallocating memory
    delete revIter;

    return 0;
}
```

Output:-

List after iterative reversal: 7->6->5->4->3->2->1

Rotate list by k C++

```
#include <iostream>

struct Node {
    int val;
    Node* next;
    Node(int x) {
        val = x;
        next = nullptr;
    }
};

Node* rotateRight(Node* head, int k) {
    if (head == nullptr || k == 0) return head;

    int length = 1;
    Node* tail = head;
    while (tail->next != nullptr) {
        tail = tail->next;
        length++;
    }

    k = k % length;
    if (k == 0) return head;

    Node* newTail = head;
    for (int i = 0; i < length - k - 1; i++) {
        newTail = newTail->next;
    }

    Node* newHead = newTail->next;
    newTail->next = nullptr;
    tail->next = head;

    return newHead;
}

void printList(Node* head) {
    while (head != nullptr) {
        std::cout << head->val << " -> ";
        head = head->next;
    }
    std::cout << "null" << std::endl;
}

int main() {
    Node* head = new Node(1);
    head->next = new Node(2);
    head->next->next = new Node(3);
    head->next->next->next = new Node(4);
    head->next->next->next->next = new Node(5);

    Node* result = rotateRight(head, 2);
    printList(result);

    // Free the allocated memory
    Node* curr = result;
    while (curr != nullptr) {
        Node* temp = curr;
        curr = curr->next;
        delete temp;
    }
```

Problem Summary:

Rotate a singly linked list **to the right** by k places.

📋 **Input:**

Linked List:

rust
CopyEdit
1 -> 2 -> 3 -> 4 -> 5

Rotate by k = 2

🔁 **Dry Run Steps:**

Step	Explanation	State
1	Initial list	1 -> 2 -> 3 -> 4 -> 5 -> null
2	Traverse list to find length and tail	length = 5, tail = 5
3	Normalize k: k = k % length = 2 % 5 = 2	Effective rotation is 2 places
4	Move to new tail: length - k - 1 = 5 - 2 - 1 = 2	Move 2 steps from head: node with value 3 is new tail
5	newTail = 3, newHead = 4, break link	newTail->next = nullptr, tail->next = head
6	New list after rotation	4 -> 5 -> 1 -> 2 -> 3 -> null

🧠 **Final State:**

- **Old Tail**: Node with value 5
- **Old Head**: Node with value 1
- **New Head**: Node with value 4
- **New Tail**: Node with value 3

✅ **Output:**

4 -> 5 -> 1 -> 2 -> 3 -> null

return 0; }	

Output:-

4 -> 5 -> 1 -> 2 -> 3 -> null

Swap nods in pairs in C++

```cpp
#include <iostream>

struct Node {
    int val;
    Node* next;
    Node(int x) {
        val = x;
        next = nullptr;
    }
};

class SwapNodesInPairs {
public:
    Node* swapPairs(Node* head) {
        Node dummy(0);
        dummy.next = head;
        Node* current = &dummy;

        while (current->next != nullptr && current->next->next != nullptr) {
            Node* first = current->next;
            Node* second = current->next->next;

            first->next = second->next;
            second->next = first;
            current->next = second;

            current = first;
        }

        return dummy.next;
    }

    static void printList(Node* head) {
        while (head != nullptr) {
            std::cout << head->val << " -> ";
            head = head->next;
        }
        std::cout << "null" << std::endl;
    }
};

int main() {
    SwapNodesInPairs solution;

    Node* head = new Node(1);
    head->next = new Node(2);
    head->next->next = new Node(3);
    head->next->next->next = new Node(4);

    Node* result = solution.swapPairs(head);
    SwapNodesInPairs::printList(result);

    // Free the allocated memory
    Node* curr = result;
    while (curr != nullptr) {
        Node* temp = curr;
        curr = curr->next;
        delete temp;
    }
```

for input:

1 -> 2 -> 3 -> 4

The goal is to swap every two adjacent nodes. So, the expected output is:

2 -> 1 -> 4 -> 3

🧠 Key Pointers:

- dummy is a placeholder node that simplifies head manipulation.
- current starts at dummy.
- first and second are the two nodes to be swapped.
- The loop continues as long as there are at least 2 nodes ahead of current.

🔁 Dry Run Table:

Iteration	current Points To	first	second	Operation	List After Swap
1	dummy (0) → 1	1	2	Swap 1 and 2	2 → 1 → 3 → 4
				first->next = 3	
				second->next = 1, current->next = 2	
				current = first → moves to node 1	
2	current → 1	3	4	Swap 3 and 4	2 → 1 → 4 → 3
				first->next = nullptr	
				second->next = 3, current->next = 4	
				current = first → moves to node 3	

✅ Final Output:

<table>
<tr><td>

```
    return 0;
}
```

</td><td>2 -> 1 -> 4 -> 3 -> null</td></tr>
<tr><td colspan="2">Output:-
2 -> 1 -> 4 -> 3 -> null</td></tr>
</table>