Intersection in C++	

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

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

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

// Intersection2LL class definition
class Intersection2LL {
public:
    Node* head1;
    Node* head2;

    int getCount(Node* node) {
        Node* current = node;
        int count = 0;

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

    int getNode() {
        int c1 = getCount(head1);
        int c2 = getCount(head2);
        int d;
        if (c1 > c2) {
            d = c1 - c2;
            return getIntesectionNode(d, head1, head2);
        } else {
            d = c2 - c1;
            return getIntesectionNode(d, head2, head1);
        }
    }

    int getIntesectionNode(int d, Node* node1, Node*
node2) {
        Node* current1 = node1;
        Node* current2 = node2;

        for (int i = 0; i < d; i++) {
            if (current1 == nullptr) {
                return -1;
            }
            current1 = current1->next;
        }

        while (current1 != nullptr && current2 !=
nullptr) {
            if (current1->data == current2->data) {
                return current1->data;
```

Final Linked Lists

List 1	List 2
3 → 6 → 9 → 15 → 30	10 → 15 → 30

- Intersection starts at **node 15** (shared memory).

🔄 Dry Run of getNode()

1. Count Nodes

Operation	Result
Count of List 1	5
Count of List 2	3
d = c1 - c2	2

2. Advance Longer List by d = 2 Nodes

After Skipping in List 1	Current Node 1	Current Node 2
Skip 1st → 3	6	
Skip 2nd → 6	9	

Now:

- current1 = 9
- current2 = 10

🔁 Start Comparing Nodes

Step	current1->data	current2->data	Same Node Address?	Action
1	9	10	✕	Move both forward
2	15	15	✓✓✓	**Return 15**

✓ Output

The node of intersection is 15

📜 Summary Table

Phase	Details
Total Nodes in List1	5
Total Nodes in List2	3
Difference d	2
First match by addr	Node with data 15
Final Answer	15

```
        }
        current1 = current1->next;
        current2 = current2->next;
    }

    return -1;
  }
};

int main() {
  // Creating an instance of Intersection2LL
  Intersection2LL list;

  // Creating first linked list
  list.head1 = new Node(3);
  list.head1->next = new Node(6);
  list.head1->next->next = new Node(9);
  list.head1->next->next->next = new Node(15);
  list.head1->next->next->next->next = new
Node(30);

  // Creating second linked list
  list.head2 = new Node(10);
  list.head2->next = new Node(15);
  list.head2->next->next = new Node(30);

  // Finding the intersection node
  cout << "The node of intersection is " <<
list.getNode() << endl;

  // Clean up memory
  delete list.head1->next->next->next->next;
  delete list.head1->next->next->next;
  delete list.head1->next->next;
  delete list.head1->next;
  delete list.head2->next->next;
  delete list.head2->next;
  delete list.head2;

  return 0;
}
```

The node of intersection is 15

K Reverse in C++

```cpp
#include <iostream>

using namespace std;

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

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

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

public:
    // Constructor
    LinkedList() {
        head = nullptr;
        tail = nullptr;
        size = 0;
    }

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

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

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

Initial Input:

List:
1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11
k = 3

🔁 kReverse Logic Dry Run:

We reverse **groups of 3 elements**. Let's track the changes in a **table** as each k-group is processed:

Group #	Extracted Nodes	Reversed Order	prev List After Merge
1	1 2 3	3 2 1	3 → 2 → 1
2	4 5 6	6 5 4	3 → 2 → 1 → 6 → 5 → 4
3	7 8 9	9 8 7	3 → 2 → 1 → 6 → 5 → 4 → 9 → 8 → 7
4	10 11	(unchanged)	... → 9 → 8 → 7 → 10 → 11

🔄 After kReverse:

List:
3 → 2 → 1 → 6 → 5 → 4 → 9 → 8 → 7 → 10 → 11

```
    }
    cout << endl;
}

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

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

// Method to reverse every k nodes in the list
void kReverse(int k) {
    LinkedList prev;

    while (size > 0) {
        LinkedList curr;

        if (size >= k) {
            for (int i = 0; i < k; i++) {
                int val = getFirst();
                removeFirst();
                curr.addFirst(val);
            }
        } else {
            int sz = size;
            for (int i = 0; i < sz; i++) {
                int val = getFirst();
                removeFirst();
                curr.addLast(val);
            }
        }

        if (prev.size == 0) {
            prev = curr;
        } else {
            tail->next = curr.head;
            tail = curr.tail;
            size += curr.size;
        }
    }

    head = prev.head;
    tail = prev.tail;
```

```
        size = prev.size;
    }

    // Destructor to free memory
    ~LinkedList() {
        Node* curr = head;
        while (curr != nullptr) {
            Node* temp = curr;
            curr = curr->next;
            delete temp;
        }
    }
};

// Main function to demonstrate LinkedList operations
int main() {
    LinkedList l1;

    l1.addLast(1);
    l1.addLast(2);
    l1.addLast(3);
    l1.addLast(4);
    l1.addLast(5);
    l1.addLast(6);
    l1.addLast(7);
    l1.addLast(8);
    l1.addLast(9);
    l1.addLast(10);
    l1.addLast(11);

    int k = 3;
    int a = 100;
    int b = 200;

    l1.display();          // Original list: 1 2 3 4 5 6 7 8 9
10 11
    l1.kReverse(k);        // Reverse every k nodes
    l1.display();          // After kReverse: 3 2 1 6 5 4 9 8
7 10 11
    l1.addFirst(a);        // Add element at the beginning:
100 3 2 1 6 5 4 9 8 7 10 11
    l1.addLast(b);         // Add element at the end: 100 3
2 1 6 5 4 9 8 7 10 11 200
    l1.display();          // Final list

    return 0;
}
```

1 2 3 4 5 6 7 8 9 10 11

Linked List (Add at index) in C++

```cpp
#include <iostream>

using namespace std;

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

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

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

public:
    // Constructor
    LinkedList() {
        head = nullptr;
        tail = nullptr;
        size = 0;
    }

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

    // Method to get the size of the list
    int getSize() {
        return 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
    void removeFirst() {
```

Dry Run Table

Step	Operation	List State	Output	Notes
1	addFirst(10)	10		Adds 10 at front
2	getFirst()	10	10	
3	addAt(0, 20)	20 → 10		Insert 20 at index 0
4	getFirst()	20 → 10	20	
5	getLast()	20 → 10	10	
6	display()	20 → 10	20 10	
7	getSize()	20 → 10	2	
8	addAt(2, 40)	20 → 10 → 40		Insert 40 at end
9	getLast()	20 → 10 → 40	40	
10	addAt(1, 50)	20 → 50 → 10 → 40		Insert 50 at index 1
11	addFirst(30)	30 → 20 → 50 → 10 → 40		Adds 30 at front
12	removeFirst()	20 → 50 → 10 → 40		Removes 30
13	getFirst()	20 → 50 → 10 → 40	20	
14	removeFirst()	50 → 10 → 40		Removes 20
15	removeFirst()	10 → 40		Removes 50
16	addAt(2, 60)	10 → 40 → 60		Adds 60 at index 2
17	display()	10 → 40 → 60	10 40 60	
18	getSize()	10 → 40 → 60	3	
19	removeFirst()	40 → 60		Removes 10

```
    if (size == 0) {
        cout << "List is empty" << endl;
    } else if (size == 1) {
        head = tail = nullptr;
        size = 0;
    } else {
        head = head->next;
        size--;
    }
}
int getFirst() {
    if (size == 0) {
        cout << "List is empty" << endl;
        return -1;
    } else {
        return head->data;
    }
}
int getLast() {
    if (size == 0) {
        cout << "List is empty" << endl;
        return -1;
    } else {
        return tail->data;
    }
}
int getAt(int idx) {
    if (size == 0) {
        cout << "List is empty" << endl;
        return -1;
    } else if (idx < 0 || idx >= size) {
        cout << "Invalid arguments" << endl;
        return -1;
    } else {
        Node* temp = head;
        for (int i = 0; i < idx; i++) {
            temp = temp->next;
        }
        return temp->data;
    }
}
// Method to add a node at the beginning of the list
void addFirst(int val) {
    Node* temp = new Node(val);
    temp->next = head;
    head = temp;
    if (size == 0) {
        tail = temp;
    }
    size++;
}

// Method to add a node at a specified index
void addAt(int idx, int val) {
    if (idx < 0 || idx > size) {
        cout << "Invalid arguments" << endl;
    } else if (idx == 0) {
        addFirst(val);
    } else if (idx == size) {
        addLast(val);
    } else {
        Node* node = new Node(val);
```

20	removeFirst()	60		Removes 40
21	getFirst()	60	60	

```
            Node* temp = head;
            for (int i = 0; i < idx - 1; i++) {
                temp = temp->next;
            }

            node->next = temp->next;
            temp->next = node;

            size++;
        }
    }
};

// Main function to demonstrate LinkedList operations
int main() {
    LinkedList list;

    // Hardcoded sequence of operations
    list.addFirst(10);
    cout << list.getFirst() << endl; // Should display: 10

    list.addAt(0, 20);
    cout << list.getFirst() << endl; // Should display: 20
    cout << list.getLast() << endl;  // Should display: 10

    list.display(); // Should display: 20 10

    cout << list.getSize() << endl; // Should display: 2

    list.addAt(2, 40);
    cout << list.getLast() << endl; // Should display: 40

    list.addAt(1, 50);
    list.addFirst(30);
    list.removeFirst();
    cout << list.getFirst() << endl; // Should display: 20

    list.removeFirst();
    list.removeFirst();
    list.addAt(2, 60);
    list.display(); // Should display: 50 10 60

    cout << list.getSize() << endl; // Should display: 3

    list.removeFirst();
    list.removeFirst();
    cout << list.getFirst() << endl; // Should display: 60

    return 0;
}
```

```
10
20
10
20 10
2
40
20
10 40 60
3
60
```

Merge in C++

```cpp
#include <iostream>

using namespace std;

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

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

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

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

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

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

    // Function to merge two sorted linked lists
    static Node* sortedMerge(Node* headA, Node*
headB) {
        Node* dummyNode = new Node(0);
        Node* tail = dummyNode;

        while (true) {
            if (headA == nullptr) {
```

What the Code Does

- Two sorted linked lists are created:
  - List 1: 5 -> 10 -> 15
  - List 2: 2 -> 3 -> 20
- The sortedMerge() function merges them into a single sorted list.
- Result is printed.

Initial Lists

List 1 (llist1)	List 2 (llist2)
5 → 10 → 15	2 → 3 → 20

Dry Run of sortedMerge()

Step	headA->data	headB->data	Chosen Node	Merged List So Far
1	5	2	2 (from B)	2
2	5	3	3 (from B)	2 → 3
3	5	20	5 (from A)	2 → 3 → 5
4	10	20	10 (from A)	2 → 3 → 5 → 10
5	15	20	15 (from A)	2 → 3 → 5 → 10 → 15
6	null	20	Append B	2 → 3 → 5 → 10 → 15 → 20

Final Output

2 3 5 10 15 20

Summary

Input List 1	Input List 2	Output (Merged Sorted List)
5 → 10 → 15	2 → 3 → 20	2 → 3 → 5 → 10 → 15 → 20

```
            tail->next = headB;
            break;
        }
        if (headB == nullptr) {
            tail->next = headA;
            break;
        }
        if (headA->data <= headB->data) {
            tail->next = headA;
            headA = headA->next;
        } else {
            tail->next = headB;
            headB = headB->next;
        }
        tail = tail->next;
    }

    return dummyNode->next;
  }
};

// Main function
int main() {
  LinkedList llist1;
  LinkedList llist2;

  // Adding elements to the first linked list
  llist1.addLast(5);
  llist1.addLast(10);
  llist1.addLast(15);

  // Adding elements to the second linked list
  llist2.addLast(2);
  llist2.addLast(3);
  llist2.addLast(20);

  // Merging the two sorted linked lists
  Node* mergedHead =
LinkedList::sortedMerge(llist1.head, llist2.head);

  // Printing the merged list
  Node* temp = mergedHead;
  while (temp != nullptr) {
    cout << temp->data << " ";
    temp = temp->next;
  }
  cout << endl;

  return 0;
}
```

```
2 3 5 10 15 20
```

Multiply LL in C++

```
#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;
    }
};
Node* reverse(Node*
head) {
    if (head == nullptr ||
head->next == nullptr)
return head;

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

    return prev;
}

// Function to add two
linked lists in place
void
addTwoLinkedList(Node*
head, Node* ansItr) {
    Node* c1 = head;
    Node* c2 = ansItr;

    int carry = 0;
    while (c1 != nullptr ||
carry != 0) {
        int sum = carry + (c1
!= nullptr ? c1->val : 0) +
(c2->next != nullptr ? c2-
>next->val : 0);
        int digit = sum % 10;
        carry = sum / 10;

        if (c2->next !=
nullptr) c2->next->val =
digit;
        else c2->next = new
Node(digit);

        if (c1 != nullptr) c1 =
c1->next;
        c2 = c2->next;
```

Given:

- **l1 = 2 -> 4 -> 3** (representing the number 342)
- **l2 = 5 -> 6 -> 4** (representing the number 465)

We are multiplying these two numbers, and as part of the algorithm, we reverse both linked lists, perform multiplication on each digit, and handle carries. Then, we add the intermediate results, ensuring proper shifting of digits.

Dry Run Table:

Step	l1 (reversed)	l2 (reversed)	Current digit of l2 (l2_itr->val)	Multiplication Result (prod)	Shift Applied	Interim Result
Initial	3 -> 4 -> 2	4 -> 6 -> 5	N/A	N/A	N/A	N/A
Reversed	2 -> 4 -> 3	5 -> 6 -> 4	N/A	N/A	N/A	N/A
Multiply l1 by 5 (1st digit of l2)	2 -> 4 -> 3	5	5 * 3 = 15, 5 * 4 = 20 + 1 (carry) = 21, 5 * 2 = 10 + 2 (carry) = 12	5 -> 1 -> 2	No Shift (first digit)	5 -> 1 -> 2
Add this result to the intermediate result (result = 5 -> 1 -> 2)	2 -> 4 -> 3	6 -> 5	N/A	N/A	N/A	5 -> 1 -> 2 (no change)
Multiply l1 by 6 (2nd digit of l2)	2 -> 4 -> 3	6	6 * 3 = 18, 6 * 4 = 24 + 1 (carry) = 25, 6 * 2 = 12 + 2 (carry) = 14	8 -> 5 -> 4	Shift by 1	8 -> 5 -> 4 -> 0 -> 0
Add this result to the intermediate result (add 8 -> 5 -> 4 -> 0 -> 0 to 5 -> 1 -> 2)	2 -> 4 -> 3	5	N/A	N/A	N/A	1 -> 5 -> 9 -> 0 -> 3 -> 0
Multiply l1 by 4 (3rd digit of l2)	2 -> 4 -> 3	4	4 * 3 = 12, 4 * 4 = 16 + 1 (carry) = 17, 4 * 2 = 8 + 1 (carry) = 9	2 -> 7 -> 9	Shift by 2	2 -> 7 -> 9 -> 0 -> 0 -> 0
Add this result to the intermediate result (add 2	2 -> 4 -> 3	4	N/A	N/A	N/A	1 -> 5 -> 9 -> 0 -> 3 -> 0 (final

```
    }
}

// Function to multiply a linked list with a single digit
Node* multiplyLLWithDigit(Node* head, int dig) {
    Node* dummy = new Node(-1);
    Node* ac = dummy;
    Node* curr = head;
    int carry = 0;
    while (curr != nullptr || carry != 0) {
        int sum = carry + (curr != nullptr ? curr->val * dig : 0);

        int digit = sum % 10;
        carry = sum / 10;

        ac->next = new Node(digit);

        if (curr != nullptr) curr = curr->next;
        ac = ac->next;
    }

    return dummy->next;
}

// Function to multiply two linked lists representing numbers
Node* multiplyTwoLL(Node* l1, Node* l2) {
    l1 = reverse(l1);
    l2 = reverse(l2);

    Node* l2_Itr = l2;
    Node* dummy = new Node(-1);
    Node* ansItr = dummy;

    while (l2_Itr != nullptr) {
        Node* prod = multiplyLLWithDigit(l1, l2_Itr->val);
        l2_Itr = l2_Itr->next;


addTwoLinkedList(prod, ansItr);
        ansItr = ansItr->next;
    }
```

-> 7 -> 9 -> 0 -> 0 -> 0 to 1 -> 5 -> 9 -> 0 -> 3 -> 0)						result)

Step-by-Step Process:

1. **Reversing the Lists**:
   - l1 = 2 -> 4 -> 3 becomes 3 -> 4 -> 2.
   - l2 = 5 -> 6 -> 4 becomes 4 -> 6 -> 5.
2. **Multiplying l1 by each digit of l2**:
   - **First, multiply l1 by 5**:
     - 5 * 3 = 15, carry = 1.
     - 5 * 4 = 20 + 1 (carry) = 21, carry = 2.
     - 5 * 2 = 10 + 2 (carry) = 12, carry = 1.
     - Result: 5 -> 1 -> 2.
   - **Second, multiply l1 by 6** (shifting by one place):
     - 6 * 3 = 18, carry = 1.
     - 6 * 4 = 24 + 1 (carry) = 25, carry = 2.
     - 6 * 2 = 12 + 2 (carry) = 14, carry = 1.
     - Result: 8 -> 5 -> 4 -> 0 -> 0.
   - **Third, multiply l1 by 4** (shifting by two places):
     - 4 * 3 = 12, carry = 1.
     - 4 * 4 = 16 + 1 (carry) = 17, carry = 1.
     - 4 * 2 = 8 + 1 (carry) = 9, carry = 0.
     - Result: 2 -> 7 -> 9 -> 0 -> 0.
3. **Adding the Intermediate Results**:
   - Add the first product 5 -> 1 -> 2 to the result.
   - Add the second product 8 -> 5 -> 4 -> 0 -> 0 to the result.
   - Add the third product 2 -> 7 -> 9 -> 0 -> 0 -> 0 to the result.
4. **Final Output**:
   - The result after adding all the intermediate products is 1 -> 5 -> 9 -> 0 -> 3 -> 0, which is the correct result for 342 * 465 = 159030.

Final Output:

159030

return reverse(dummy->next); } // Function to print the linked list void printList(Node* node) { while (node != nullptr) { cout << node->val << " "; node = node->next; } cout << endl; } // 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 lists // First list: 3 -> 4 -> 2 (represents the number 243) int arr1[] = {3, 4, 2}; int n1 = sizeof(arr1) / sizeof(arr1[0]); Node* head1 = createList(arr1, n1); // Second list: 4 -> 6 -> 5 (represents the number 564) int arr2[] = {4, 6, 5}; int n2 = sizeof(arr2) / sizeof(arr2[0]); Node* head2 = createList(arr2, n2); // Multiplying the two linked lists Node* ans = multiplyTwoLL(head1, head2); // Printing the result	

`` printList(ans); return 0; } ```	
1 5 9 0 3 0	

Pair Wise swap in C++

```
#include <iostream>

using namespace std;

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

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

// PairwiseSwapLL class definition
class PairwiseSwapLL {
public:
    Node* head;

    PairwiseSwapLL() {
        head = nullptr;
    }

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

    // Method to perform pairwise swapping of nodes
    Node* pairWiseSwap(Node* node) {
        if (node == nullptr || node->next == nullptr) {
            return node;
        }

        Node* remaining = node->next->next;
        Node* newHead = node->next;
        node->next->next = node;
        node->next = pairWiseSwap(remaining);
        return newHead;
    }
};

int main() {
    // Create an instance of PairwiseSwapLL
    PairwiseSwapLL list;

    // Construct the linked list: 1->2->3->4->5->6->7
    list.head = new Node(1);
    list.head->next = new Node(2);
    list.head->next->next = new Node(3);
    list.head->next->next->next = new Node(4);
    list.head->next->next->next->next = new Node(5);
    list.head->next->next->next->next->next = new
Node(6);
    list.head->next->next->next->next->next->next =
```

Dry Run Table

Input List: 1 → 2 → 3 → 4 → 5 → 6 → 7

Recursive Call	node	Swapped Pair	Remaining	Result after call
1	1	1 ↔ 2	3	2 → 1 → ?
2	3	3 ↔ 4	5	4 → 3 → ?
3	5	5 ↔ 6	7	6 → 5 → ?
4	7	no pair	nullptr	7

🔁 Backtracking:

- 4th call returns: 7
- 3rd call builds: 6 → 5 → 7
- 2nd call builds: 4 → 3 → 6 → 5 → 7
- 1st call builds: 2 → 1 → 4 → 3 → 6 → 5 → 7

✅ **Final Output:**

2 1 4 3 6 5 7

new Node(7); // Display the original list cout << "Linked list before calling pairwiseSwap() " << endl; list.printList(list.head); // Perform pairwise swapping list.head = list.pairWiseSwap(list.head); // Display the list after pairwise swapping cout << "Linked list after calling pairwiseSwap() " << endl; list.printList(list.head); // Clean up allocated memory Node* curr = list.head; Node* next = nullptr; while (curr != nullptr) { next = curr->next; delete curr; curr = next; } return 0; }	
Linked list before calling pairwiseSwap() 1 2 3 4 5 6 7 Linked list after calling pairwiseSwap() 2 1 4 3 6 5 7	

Palindrome in LL in C++

```
#include <iostream>
#include <stack>

using namespace std;

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

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

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

public:
    // Constructor
    LinkedList() {
        head = nullptr;
        tail = nullptr;
        size = 0;
    }

    // Method to add a node at the end of the list
    void addLast(int val) {
        Node* temp = new Node(val);
        if (size == 0) {
            head = tail = temp;
        } else {
            tail->next = temp;
            tail = temp;
        }
        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 check if the linked list is a palindrome
    bool isPalindrome() {
        Node* slow = head;
        stack<int> stack;

        // Push elements of the first half of the linked list
```

Dry Run for Your Example: **1 → 2 → 3 → 2 → 1**

Step	Stack Contents	slow points to	Comparison
Push	1, 2	3	-
Skip	(middle: 3)	2	-
Check	Top: 2 vs 2	2	✅
Check	Top: 1 vs 1	1	✅

✅ **Result: true**

Let me know if you'd like a version that modifies the list

onto the stack while (slow != nullptr) { stack.push(slow->data); slow = slow->next; } // Compare elements of the second half of the linked list with the stack slow = head; while (slow != nullptr) { int top = stack.top(); stack.pop(); if (slow->data != top) { return false; } slow = slow->next; } return true; } }; // Main function to demonstrate LinkedList operations int main() { // Create a linked list LinkedList list; // Add elements to the linked list list.addLast(1); list.addLast(2); list.addLast(3); list.addLast(2); list.addLast(1); // Check if the linked list is a palindrome cout << boolalpha << list.isPalindrome() << endl; // Output: true return 0; }	
true	

Remove duplicate in LL in C++

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

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

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

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

// Function to remove duplicates from the linked list
void deleteDups(Node* head) {
    if (head == nullptr || head->next == nullptr) return;

    Node* current = head;
    while (current != nullptr) {
        Node* runner = current;
        while (runner->next != nullptr) {
            if (runner->next->data == current->data) {
                runner->next = runner->next->next;
            } else {
                runner = runner->next;
            }
        }
        current = current->next;
    }
}

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

Creates a linked list: 1 -> 2 -> 2 -> 3 -> 4 -> 3 -> 5 -> null

Initial Linked List Creation

Node	Value	Next Points To
head	1	Node 2
head->next	2	Node 2
...	2	Node 3
...	3	Node 4
...	4	Node 3
...	3	Node 5
...	5	nullptr

🖨️ Initial Output from printList(head)

Original Linked List:
1 -> 2 -> 2 -> 3 -> 4 -> 3 -> 5 -> null

🔁 deleteDups(head) Dry Run

Loop Over current Node

current->data	Duplicate(s) Found and Removed	Resulting List
1	None	1 → 2 → 2 → 3 → 4 → 3 → 5
2	Second 2 removed	1 → 2 → 3 → 4 → 3 → 5
3	Second 3 removed	1 → 2 → 3 → 4 → 5
4	None	1 → 2 → 3 → 4 → 5
5	None	1 → 2 → 3 → 4 → 5

✅ Final Linked List After deleteDups(head)

Linked List after removing duplicates:

```
    // Print the original linked list
    cout << "Original Linked List:" << endl;
    printList(head);

    // Remove duplicates
    deleteDups(head);

    // Print the linked list after removing duplicates
    cout << "Linked List after removing duplicates:" <<
endl;
    printList(head);

    return 0;
}
```

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

```
Original Linked List:
1 -> 2 -> 2 -> 3 -> 4 -> 3 -> 5 -> null
Linked List after removing duplicates:
1 -> 2 -> 3 -> 4 -> 5 -> null
```

Reverse 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);
```

Recursive Reversal: reverse(Node* head)

🔍 **Dry Run (for list: 1 -> 2 -> 3)**

Step	**Call Stack (Function Call)**	**Action**	**Resulting Links**
1	reverse(1)	Calls reverse(2)	-
2	reverse(2)	Calls reverse(3)	-
3	reverse(3)	Base case hit, returns 3	-
4	Back to reverse(2)	3->next = 2, 2->next = nullptr	3 → 2
5	Back to reverse(1)	2->next = 1, 1->next = nullptr	3 → 2 → 1

✅ Final Result: 3 → 2 → 1

🔁 Iterative Reversal: reverseI(Node* head)

🔍 **Dry Run (on 3 → 2 → 1)**

curr	**prev**	**next**	**Action**	**New Links**
3	null	2	3->next = null	3
2	3	1	2->next = 3	2 → 3
1	2	null	1->next = 2	1 → 2 → 3

✅ Final Result: 1 → 2 → 3

<pre> 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; }</pre>	
Original List: 1->2->3->4->5->6->7 List after recursive reversal: 7->6->5->4->3->2->1 List after iterative reversal: 1->2->3->4->5->6->7	

Segregate Even Odd in C++

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

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

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

Node* segregateEvenOdd(Node* head) {
    if (head == nullptr || head->next == nullptr)
return head;

    Node* dummyEven = new Node(-1);
    Node* dummyOdd = new Node(-1);
    Node* evenTail = dummyEven;
    Node* oddTail = dummyOdd;
    Node* curr = head;

    while (curr != nullptr) {
        if (curr->val % 2 != 0) {
            oddTail->next = curr;
            oddTail = oddTail->next;
        } else {
            evenTail->next = curr;
            evenTail = evenTail->next;
        }

        curr = curr->next;
    }

    evenTail->next = dummyOdd->next;
    oddTail->next = nullptr;

    Node* result = dummyEven->next;
    delete dummyEven;
    delete dummyOdd;
    return result;
}

void push(Node*& head, int new_data) {
    Node* new_node = new Node(new_data);
    new_node->next = head;
    head = new_node;
}

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

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

What This Code Does

1. Builds a linked list: 6 -> 9 -> 10 -> 11
2. Separates **even** and **odd** numbers.
3. Appends odd list **after** the even list.
4. Prints the result: 6 -> 10 -> 9 -> 11

🧱 Linked List Construction (push)

push inserts at the head. So insertion order is:

Push Order	Value Inserted	List After Push
1	11	11
2	10	10 → 11
3	9	9 → 10 → 11
4	6	6 → 9 → 10 → 11

🔄 segregateEvenOdd(head) Dry Run

curr->val	Even/Odd	Action	Even List	Odd List
6	Even	Added to even list	6	-
9	Odd	Added to odd list	6	9
10	Even	Added to even list	6 → 10	9
11	Odd	Added to odd list	6 → 10	9 → 11

Then:

- evenTail->next = dummyOdd->next connects 6 → 10 → 9 → 11
- oddTail->next = nullptr ends the list

🖨️ Final Output from printList(head1)

6 10 9 11

📌 Summary

Before Segregation	After Segregation
6 → 9 → 10 → 11	6 → 10 → 9 → 11

<pre> push(head, 11); push(head, 10); push(head, 9); push(head, 6); Node* head1 = segregateEvenOdd(head); printList(head1); return 0; }</pre>	
6 10 9 11	

Sublist in C++

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

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

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

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

void sublists(Node* head) {
    Node* i = head;
    while (i != nullptr) {
        Node* j = i;
        while (j != nullptr) {
            cout << j->data << " -> ";
            j = j->next;
        }
        cout << "null" << endl;
        i = i->next;
    }
}

int main() {
    // Create a linked list with 5 hard-coded nodes
    Node* head = new Node(1);
    head->next = new Node(2);
    head->next->next = new Node(2);
    head->next->next->next = new Node(3);
    head->next->next->next->next = new Node(4);
    head->next->next->next->next->next = new
Node(3);
    head->next->next->next->next->next->next = new
Node(5);

    // Print the linked list
    printList(head);

    // Print all sublists
    sublists(head);

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

Linked List Creation

Step	Node Created	data	next Points To
1	head	1	Node with 2
2	head->next	2	Node with 2
3	...	2	Node with 3
4	...	3	Node with 4
5	...	4	Node with 3
6	...	3	Node with 5
7	...	5	nullptr

📤 printList(head) Output

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

🔁 sublists(head) Dry Run Table

Outer Loop (i->data)	Inner Loop Iteration (→ values printed)
1	1 -> 2 -> 2 -> 3 -> 4 -> 3 -> 5 -> null
2 (1st)	2 -> 2 -> 3 -> 4 -> 3 -> 5 -> null
2 (2nd)	2 -> 3 -> 4 -> 3 -> 5 -> null
3	3 -> 4 -> 3 -> 5 -> null
4	4 -> 3 -> 5 -> null
3 (last)	3 -> 5 -> null
5	5 -> null

🧹 Cleanup (Memory Deallocation)

Step	Node Deleted	data
1	head	1
2		2
3		2
4		3
5		4

```
    return 0;
}
```

Step	Node Deleted	data
6		3
7		5

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

Sumlist in C++

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

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

    // Default constructor
    Node() {
        data = 0;
        next = nullptr;
    }

    // Constructor with data parameter
    Node(int data) {
        this->data = data;
        next = nullptr;
    }

    void setNext(Node* next) {
        this->next = next;
    }
};

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

// Function to add two linked lists representing numbers
Node* add(Node* l1, Node* l2, int carry) {
    if (l1 == nullptr && l2 == nullptr && carry == 0) {
        return nullptr;
    }

    Node* result = new Node();
    int value = carry;
    if (l1 != nullptr) {
        value += l1->data;
    }
    if (l2 != nullptr) {
        value += l2->data;
    }
    result->data = value % 10;

    if (l1 != nullptr || l2 != nullptr) {
        Node* more = add(l1 == nullptr ? nullptr : l1->next, l2 == nullptr ? nullptr : l2->next, value >= 10 ? 1 : 0);
        result->setNext(more);
    }
    return result;
```

What the Code Does

- Adds two numbers represented by linked lists in **reverse order** (just like how we add numbers manually from right to left).
- Example:
    - List 1: 7 -> 1 -> 6 = 617
    - List 2: 5 -> 9 -> 2 = 295
    - Sum: **617 + 295 = 912**
    - Result list: 2 -> 1 -> 9

Input Linked Lists

List	Nodes	Represents
l1	7 → 1 → 6	617
l2	5 → 9 → 2	295

add(l1, l2, carry) Dry Run

Step	l1->data	l2->data	Carry In	Sum	Digit Stored	Carry Out	Notes
1	7	5	0	12	2	1	result->data = 2
2	1	9	1	11	1	1	result->next->data = 1
3	6	2	1	9	9	0	result->next->next->data = 9
4	null	null	0	-	-	-	Recursion stops

Result Linked List After Addition

2 -> 1 -> 9 -> null

```
}

int main() {
    // Creating two linked lists representing numbers
    Node* head1 = new Node(7);
    head1->next = new Node(1);
    head1->next->next = new Node(6);

    Node* head2 = new Node(5);
    head2->next = new Node(9);
    head2->next->next = new Node(2);

    // Adding the two linked lists
    Node* result = add(head1, head2, 0);

    // Printing the result linked list
    cout << "Result of addition:" << endl;
    printList(result);

    return 0;
}
```

```
Result of addition:
2 -> 1 -> 9 -> null
```