**Question (One) 1**

For this problem, I used Floyd's Tortoise and Hare algorithm, also known as the "two pointers" approach. This algorithm involves two pointers, called the slow pointer and the fast pointer, starting from the head of the linked list.

The slow pointer moves one step at a time, while the fast pointer moves two steps at a time. If there is a cycle in the linked list, eventually, the fast pointer will catch up to the slow pointer. If there is no cycle, the fast pointer will reach the end of the list (i.e., become nullptr).

Here's the C++ code implementing this algorithm:

#include <iostream>

struct ListNode {

int val;

ListNode\* next;

ListNode(int x) : val(x), next(nullptr) {}

};

bool hasCycle(ListNode\* head) {

if (head == nullptr || head->next == nullptr) {

return false;

}

ListNode\* slow = head;

ListNode\* fast = head->next;

while (slow != fast) {

if (fast == nullptr || fast->next == nullptr) {

return false; // No cycle found

}

slow = slow->next;

fast = fast->next->next;

}

return true; // Cycle found

}

// Example usage

int main() {

// Example 1

ListNode\* head1 = new ListNode(3);

head1->next = new ListNode(2);

head1->next->next = new ListNode(0);

head1->next->next->next = new ListNode(-4);

head1->next->next->next->next = head1->next;

std::cout << "Example 1: " << std::boolalpha << hasCycle(head1) << std::endl;

// Example 2

ListNode\* head2 = new ListNode(1);

head2->next = new ListNode(2);

head2->next->next = head2;

std::cout << "Example 2: " << std::boolalpha << hasCycle(head2) << std::endl;

// Example 3

ListNode\* head3 = new ListNode(1);

std::cout << "Example 3: " << std::boolalpha << hasCycle(head3) << std::endl;

return 0;

}

**QUESTION TWO (2):**

For this problem, I used the Floyd's Tortoise and Hare algorithm, also known as the "two pointers" approach. This algorithm involves two pointers, called the slow pointer and the fast pointer, starting from the head of the linked list.

The slow pointer moves one step at a time, while the fast pointer moves two steps at a time. If there is a cycle in the linked list, eventually, the fast pointer will catch up to the slow pointer within the cycle.

Once they meet, we reset one of the pointers to the head of the linked list, and then move both pointers at the same pace (one step at a time) until they meet again. The point where they meet is the starting point of the cycle.

Here's the C++ code implementing this algorithm:

#include <iostream>

struct ListNode {

int val;

ListNode\* next;

ListNode(int x) : val(x), next(nullptr) {}

};

ListNode\* detectCycle(ListNode\* head) {

if (head == nullptr || head->next == nullptr) {

return nullptr;

}

ListNode\* slow = head;

ListNode\* fast = head;

// Phase 1: Detect cycle

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

slow = slow->next;

fast = fast->next->next;

if (slow == fast) {

break; // Cycle detected

}

}

// If no cycle found

if (fast == nullptr || fast->next == nullptr) {

return nullptr;

}

// Phase 2: Find starting point of the cycle

slow = head;

while (slow != fast) {

slow = slow->next;

fast = fast->next;

}

return slow; // Return the starting point of the cycle

}

// Example usage

int main() {

// Example 1

ListNode\* head1 = new ListNode(3);

head1->next = new ListNode(2);

head1->next->next = new ListNode(0);

head1->next->next->next = new ListNode(-4);

head1->next->next->next->next = head1->next;

ListNode\* cycleStart1 = detectCycle(head1);

std::cout << "Example 1: " << cycleStart1->val << std::endl;

// Example 2

ListNode\* head2 = new ListNode(1);

head2->next = new ListNode(2);

head2->next->next = head2;

ListNode\* cycleStart2 = detectCycle(head2);

std::cout << "Example 2: " << cycleStart2->val << std::endl;

// Example 3

ListNode\* head3 = new ListNode(1);

ListNode\* cycleStart3 = detectCycle(head3);

if (cycleStart3 != nullptr) {

std::cout << "Example 3: " << cycleStart3->val << std::endl;

} else {

std::cout << "Example 3: No cycle" << std::endl;

}

return 0;

}

**QUESTION THREE (3):**

ListNode\* reverseLinkedList(ListNode\* head) {

ListNode\* prev = nullptr;

ListNode\* curr = head;

ListNode\* next = nullptr;

while (curr != nullptr) {

// Store the next node

next = curr->next;

// Reverse the link

curr->next = prev;

// Move pointers one step forward

prev = curr;

curr = next;

}

// Update the head of the reversed list

head = prev;

return head;

}