Data Structure

Chapter 5: Linked List

Exam-Oriented Problems and Solutions — Set 5

This fifth set covers reversing a list in groups of k, finding the length of a loop, set operations (union and intersection) on linked lists, segregating even and odd valued nodes, and splitting a circular linked list into two halves. Each problem is followed by a complete, step-by-step solution.

Section A: Conceptual and Numerical Problems

Problem 1. Explain what a sentinel (dummy) head node is, and how using one simplifies insertion and deletion algorithms for a singly linked list. *[4 marks]*

Solution:

A sentinel or dummy node is an extra, permanently present node placed before the actual first data node of the list. The external head/reference pointer always points to this dummy node instead of directly to the first real node, and the dummy node's own data field is never used.

Without a dummy node, inserting or deleting the very first real node is a special case, because it requires updating the external head pointer itself rather than another node's next field, which typically needs an if-condition in the algorithm (as seen in ordinary beginning-insertion and beginning-deletion). With a dummy node in place, the 'first real node' always has a predecessor (the dummy node itself), so insertion and deletion at the front can be handled by exactly the same code path as insertion and deletion anywhere else in the list — dummy->next is simply treated like any other node's next field. This removes the need for a separate empty-list or first-node special case throughout the insertion, deletion and even reversal algorithms.

Problem 2. Two linked lists represent sets with m and n elements respectively. Compare the worst-case time complexity of computing their union using (a) a naive nested-loop approach that checks each element of the second list against the entire first list, versus (b) a hashing-based approach. *[4 marks]*

Solution:

(a) The naive nested-loop approach builds the result from the first list, then for every one of the n elements of the second list, scans the (up to m) elements already in the result to check whether it is a duplicate before adding it. In the worst case this is O(m × n), since each of the n checks can take O(m) time.

(b) The hashing-based approach inserts every element of both lists into a hash set, relying on O(1) average-case insertion and lookup; only elements not already present are added to the result. This reduces the worst-case time to O(m + n), at the cost of O(m + n) extra space for the hash set — a classic time–space trade-off that is preferred whenever m and n are large.

Problem 3. A circular linked list contains n nodes. If it is split into two halves using the slow/fast pointer technique, how many nodes does each half receive when n is odd versus when n is even? *[3 marks]*

Solution:

When n is even, the split is exact: each half receives n/2 nodes.

When n is odd, an exact 50-50 split is not possible, so by convention the first half (the one containing the original head) receives one extra node: the first half gets ⌈n/2⌉ = (n+1)/2 nodes, and the second half gets ⌊n/2⌋ = (n-1)/2 nodes. This is why the splitting algorithm includes an extra adjustment step for the fast pointer only

when the total number of nodes is even, so that the boundary between the two halves is placed correctly in both cases.

Problem 4. When segregating a linked list so that all even-valued nodes appear before all odd-valued nodes, why is it important that the relative order among the even nodes (and separately among the odd nodes) is preserved? What term describes this property? *[3 marks]*

Solution:

This property is called stability. It matters because the nodes carry no information about their original position once they are moved into separate even and odd sub-chains; if the algorithm did not preserve the order in which values of the same category originally appeared, information about the original sequence would be permanently lost, which could break any later processing that depends on that original relative ordering (for example, if the list represented a sequence of timestamped events, shuffling the even-valued events among themselves would lose their chronological order).

The segregation algorithm achieves stability simply by appending each node to the end of its respective (even or odd) sub-chain as it is encountered during a single left-to-right traversal, rather than, say, repeatedly searching for and moving nodes, which could easily disturb the original order.

Section B: Algorithm and Code-Based Problems

Problem 5. Write a recursive function to reverse a singly linked list in groups of k nodes (if fewer than k nodes remain at the end, reverse that remaining group as well). *[8 marks]*

Solution:

```
struct Node* reverseKGroup(struct Node *head, int k) {
    struct Node *curr = head, *prev = NULL, *next = NULL;
    int count = 0;

    // reverse the first k nodes (or fewer, if the list ends early)
    while (curr != NULL && count < k) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
        count++;
    }

    // 'head' is now the tail of this reversed group;
    // link it to the (recursively reversed) rest of the list
    if (next != NULL) {
        head->next = reverseKGroup(next, k);
    }
    return prev;   // prev is the new head of this group
}
```

The function reverses exactly one group of up to k nodes using the standard iterative reversal technique (prev/curr/next), stopping either after k nodes or when the list runs out. The original head of this group ends up as its last node after reversal, so its next pointer is used to attach the result of recursively reversing everything after this group. The recursion naturally handles a final partial group of fewer than k nodes, since the while loop simply stops early when curr becomes NULL. This takes O(n) time overall, since every node is visited a constant number of times, and O(n/k) auxiliary space due to the recursion depth.

Problem 6. Given that a singly linked list contains a cycle and a node where the slow and fast pointers met (via Floyd's algorithm) is known, write a function to compute the number of nodes in the cycle (the loop length). *[5 marks]*

Solution:

```
int loopLength(struct Node *meetingPoint) {
    struct Node *curr = meetingPoint->next;
    int length = 1;
    while (curr != meetingPoint) {
        curr = curr->next;
        length++;
    }
    return length;
}
```

Since the meeting point returned by Floyd's algorithm is guaranteed to lie inside the cycle, simply following next pointers starting from it and counting steps until arriving back at the same node gives exactly the number of distinct nodes in the cycle — because the cycle, by definition, loops back on itself after that many steps. This runs in O(c) time, where c is the length of the cycle, and O(1) extra space.

Problem 7. Two singly linked lists represent sets of distinct integers. Write functions to compute their union and their intersection as new linked lists (an element should appear only once in each result, even if repeated across inputs). *[8 marks]*

Solution:

```
// isPresent and insertUnique/insertEnd are simple O(k) helper traversals

struct Node* unionOfLists(struct Node *a, struct Node *b) {
    struct Node *result = NULL;
    for (struct Node *t = a; t != NULL; t = t->next)
        if (!isPresent(result, t->data))
            result = insertEnd(result, t->data);
    for (struct Node *t = b; t != NULL; t = t->next)
        if (!isPresent(result, t->data))
            result = insertEnd(result, t->data);
    return result;
}

struct Node* intersectionOfLists(struct Node *a, struct Node *b) {
    struct Node *result = NULL;
    for (struct Node *t = a; t != NULL; t = t->next)
        if (isPresent(b, t->data) && !isPresent(result, t->data))
            result = insertEnd(result, t->data);
    return result;
}
```

For the union, every element of both lists is added to the result list, but only if it is not already present in the result, which avoids duplicates. For the intersection, only elements of the first list that also appear somewhere in the second list are added, again guarding against duplicates in the result. Using plain list-traversal helpers for isPresent, this takes O(m × n) time in the worst case; replacing isPresent with a hash-set lookup reduces this to O(m + n) time at the cost of O(m + n) extra space, as discussed in Problem 2 of Section A.

Problem 8. Write a function to rearrange a singly linked list so that all even-valued nodes appear before all odd-valued nodes, preserving the relative order within each group, using only pointer rearrangement (no new nodes). *[7 marks]*

Solution:

```
struct Node* segregateEvenOdd(struct Node *head) {
    struct Node *evenStart = NULL, *evenEnd = NULL;
    struct Node *oddStart = NULL, *oddEnd = NULL;
    struct Node *curr = head;

    while (curr != NULL) {
        if (curr->data % 2 == 0) {
            if (evenStart == NULL) evenStart = evenEnd = curr;
            else { evenEnd->next = curr; evenEnd = curr; }
        } else {
            if (oddStart == NULL) oddStart = oddEnd = curr;
            else { oddEnd->next = curr; oddEnd = curr; }
        }
        curr = curr->next;
    }

    if (oddStart == NULL) { evenEnd->next = NULL; return evenStart; }
    if (evenStart == NULL) return oddStart;

    evenEnd->next = oddStart;   // join even chain to odd chain
    oddEnd->next = NULL;        // terminate the combined list
```

```
    return evenStart;
}
```

A single traversal splits the original nodes into two separate chains — one collecting even-valued nodes and one collecting odd-valued nodes — by appending each node to the tail of its matching chain as it is visited, which keeps each chain in the same relative order as the original list. After the traversal, the two chains are joined by pointing the last even node's next field to the first odd node, and the last odd node's next field is set to NULL to properly terminate the list. Two special cases (no odd nodes at all, or no even nodes at all) are handled separately to avoid dereferencing a NULL chain. This runs in O(n) time and O(1) extra space, since only existing nodes are relinked.

Problem 9. Write a function to split a circular singly linked list into two separate circular linked lists of (as close to) equal size, given only a pointer to the head. *[7 marks]*

Solution:

```
void splitCircularList(struct Node *head, struct Node **head1, struct Node **head2) {
    if (head == NULL) { *head1 = NULL; *head2 = NULL; return; }

    struct Node *slow = head, *fast = head;
    while (fast->next != head && fast->next->next != head) {
        slow = slow->next;
        fast = fast->next->next;
    }
    if (fast->next->next == head)   // even number of nodes: advance fast once more
        fast = fast->next;

    *head1 = head;
    *head2 = slow->next;

    slow->next = *head1;   // close the first half into its own circle
    fast->next = *head2;   // close the second half into its own circle
}
```

The slow/fast pointers advance through the circular list (checking against head, since there is no NULL terminator) until fast is near the end, at which point slow sits at the boundary between the two halves. For an odd total number of nodes, this naturally gives the first half one extra node; for an even total, one extra step for fast is needed so that the split falls exactly in the middle, which the explicit check handles. Finally, the two halves are each closed into their own valid circular list by redirecting the last node of each half to point back to that half's own starting node. This runs in O(n) time and O(1) extra space.

Section C: Trace-Based Problems

Problem 10. Trace the reverseKGroup algorithm on the list 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> NULL with k = 3. *[6 marks]*

Solution:

```
Original list:   1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> NULL

Group 1 (nodes 1,2,3) reversed:   3 -> 2 -> 1
   node 1 (old head of this group) will link to the result of the next group

Group 2 (nodes 4,5,6) reversed:   6 -> 5 -> 4
   node 4 (old head of this group) will link to the result of the next group

Group 3 (node 7 only – fewer than k):   7   (a single node, trivially 'reversed')

Linking back up: node 4 -> 7,   node 1 -> (6 -> 5 -> 4 -> 7)

Final list:   3 -> 2 -> 1 -> 6 -> 5 -> 4 -> 7 -> NULL
```

Each group of 3 is internally reversed independently, and the groups themselves remain in their original left-to-right order, with the last node of each reversed group (the original first node of that group) linking forward to the head of the next reversed group — including the final, undersized group of just one node.

Problem 11. A circular linked list is 3 -> 4 -> 5 -> (back to 3), reached via a longer list 1 -> 2 -> 3 -> 4 -> 5 -> (back to 3), where Floyd's algorithm reported the meeting point as node 4. Trace the loopLength algorithm to find the number of nodes in the cycle. *[5 marks]*

Solution:

```
meetingPoint = node 4
curr = meetingPoint->next = 5,   length = 1

curr(5) != meetingPoint(4)  ->  curr = 3,  length = 2
curr(3) != meetingPoint(4)  ->  curr = 4,  length = 3
curr(4) == meetingPoint(4)  ->  stop

Loop length = 3
```

This correctly counts the three nodes that form the cycle (3, 4 and 5), confirming that starting the count from the meeting point and walking forward until returning to it visits every node of the cycle exactly once.

Problem 12. List A represents the set {10, 15, 4, 20} and List B represents the set {8, 4, 2, 10}. Trace the computation of their union and intersection. *[6 marks]*

Solution:

```
List A: 10 -> 15 -> 4 -> 20 -> NULL
List B: 8 -> 4 -> 2 -> 10 -> NULL

Union (add all of A, then add B's elements not already present):
   From A: 10, 15, 4, 20   (all added, result so far: 10,15,4,20)
   From B: 8 (new, add) , 4 (already present, skip),
           2 (new, add), 10 (already present, skip)
   Union result:  10 -> 15 -> 4 -> 20 -> 8 -> 2 -> NULL
```

```
Intersection (elements of A that also appear in B):
   10 -> present in B? yes -> add
   15 -> present in B? no  -> skip
   4  -> present in B? yes -> add
   20 -> present in B? no  -> skip
   Intersection result:  10 -> 4 -> NULL
```

The union correctly contains every distinct value from both sets exactly once (six values total, since 4 and 10 were common to both lists and are not duplicated), and the intersection correctly contains only the two values, 10 and 4, that were present in both original lists.

Problem 13. Trace the segregateEvenOdd algorithm on the list 17 -> 15 -> 8 -> 12 -> 10 -> 5 -> 4 -> NULL. *[5 marks]*

Solution:

```
Original list: 17 -> 15 -> 8 -> 12 -> 10 -> 5 -> 4 -> NULL

Traverse and classify each node:
   17 (odd)   -> odd chain:  17
   15 (odd)   -> odd chain:  17 -> 15
   8  (even)  -> even chain: 8
   12 (even)  -> even chain: 8 -> 12
   10 (even)  -> even chain: 8 -> 12 -> 10
   5  (odd)   -> odd chain:  17 -> 15 -> 5
   4  (even)  -> even chain: 8 -> 12 -> 10 -> 4

Join even chain's tail (4) to odd chain's head (17); terminate at odd chain's tail (5):

Final list:  8 -> 12 -> 10 -> 4 -> 17 -> 15 -> 5 -> NULL
```

All four even-valued nodes (8, 12, 10, 4) appear first, in exactly the same relative order as in the original list, followed by all three odd-valued nodes (17, 15, 5), also in their original relative order — demonstrating the stability of the segregation.