Data Structure

Chapter 5: Linked List

Exam-Oriented Problems and Solutions — Set 2

This second set focuses on the two-pointer (slow/fast) technique, cycle detection, the nth-from-end problem, doubly linked list operations, and other frequently examined variations on the Linked List chapter. Each problem is followed by a complete, step-by-step solution.

Section A: Conceptual and Numerical Problems

Problem 1. You are given a pointer to a node X in a singly linked list, but not a pointer to the head, and X is known not to be the last node. Explain how X can still be effectively 'deleted' from the list in O(1) time. *[5 marks]*

Solution:

Normally, deleting a node requires access to its predecessor, which in turn requires traversing from the head — an O(n) operation. However, when only the node itself is given (and it is guaranteed not to be the last node), a different trick is used: instead of removing X itself, X's data is overwritten with the data of the next node, and then the next node is unlinked and freed.

1. Copy the data field: X->data = X->next->data.
2. Save a temporary pointer to the node after X: temp = X->next.
3. Relink: X->next = X->next->next (i.e., X->next = temp->next).
4. Free the temporary node: free(temp).

After this, the node originally called X now holds the value that used to belong to the next node, and the actual next node has been removed from the chain — so, from an external viewpoint, X's original value has effectively disappeared from the list, in O(1) time and without touching the head pointer. This trick fails only when X is the last node, since there is no next node whose data could be copied.

Problem 2. A singly linked list has 9 nodes. Using the slow/fast pointer technique (slow moves 1 step, fast moves 2 steps), at which node does slow point when fast reaches the end? Number the nodes 1 to 9. *[4 marks]*

Solution:

For a list of n nodes, the slow/fast technique (with fast starting one step ahead, or both starting together depending on convention) lands slow on the middle node. Using the common convention where both start at node 1:

```
Step 0:  slow=1, fast=1
Step 1:  slow=2, fast=3
Step 2:  slow=3, fast=5
Step 3:  slow=4, fast=7
Step 4:  slow=5, fast=9   (fast->next is NULL, loop stops)
```

Fast has reached the last node (node 9) after 4 steps, and at that point slow is at node 5 — the exact middle of a 9-node list, confirming the standard result that this technique locates the middle in a single pass using O(1) extra space.

Problem 3. Each node of a singly linked list storing an int requires 4 bytes for data and 8 bytes for the next pointer (on a 64-bit system). An equivalent array of int also uses 4 bytes per element but no pointer overhead. For a list/array of 1000 elements, how much extra memory does the linked list consume compared to the array, purely due to pointer overhead? *[3 marks]*

Solution:

Each linked list node uses 4 + 8 = 12 bytes, of which 8 bytes is pure pointer overhead not present in the array representation. For 1000 nodes, the total pointer overhead is 1000 × 8 = 8000 bytes (approximately 7.8 KB).

This is the memory cost paid for the flexibility of dynamic sizing and O(1) insertion/deletion: the array stores 1000 × 4 = 4000 bytes with zero overhead, while the linked list stores 1000 × 12 = 12000 bytes, i.e. exactly 8000 bytes (double the useful data size) more than the array, purely in link fields.

Problem 4. A doubly linked list has n nodes. Write down, in order, the pointer updates required to insert a new node newNode between two existing adjacent nodes A and B (so the list changes from ... -> A -> B -> ... to ... -> A -> newNode -> B -> ...). *[5 marks]*

Solution:

1. newNode->next = B (link the new node forward to B)
2. newNode->prev = A (link the new node backward to A)
3. A->next = newNode (redirect A's forward link to the new node)
4. B->prev = newNode (redirect B's backward link to the new node)

The order matters: newNode's own next and prev must be set first, while A and B are still directly linked to each other, so that no address is lost. Only after newNode correctly points to both neighbours are A and B's links redirected to point to newNode instead of to each other. A doubly linked list requires all four of these updates (compared to just two in a singly linked list insertion) because every node's backward link must also stay consistent.

Section B: Algorithm and Code-Based Problems

Problem 5. Write a function using the slow/fast pointer technique to find the middle node of a singly linked list in a single pass. *[6 marks]*

Solution:

```
struct Node* findMiddle(struct Node *head) {
    struct Node *slow = head, *fast = head;
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;   // middle node (second middle, if length is even)
}
```

Two pointers start at the head. On every iteration, slow advances by one node while fast advances by two nodes. Since fast covers ground twice as fast as slow, by the time fast reaches the end of the list (fast becomes NULL or fast->next becomes NULL), slow has covered exactly half the distance and is therefore at the middle node. This runs in a single O(n) pass using only O(1) extra space, avoiding the need to first count the nodes and then traverse again to the n/2-th node.

Problem 6. Write a function to find the n-th node from the end of a singly linked list in a single pass, without knowing the length of the list in advance. *[6 marks]*

Solution:

```
struct Node* nthFromEnd(struct Node *head, int n) {
    struct Node *first = head, *second = head;

    for (int i = 0; i < n; i++) {
        if (first == NULL) return NULL;   // n is larger than list length
        first = first->next;
    }

    while (first != NULL) {
        first = first->next;
        second = second->next;
    }
    return second;   // n-th node from the end
}
```

The pointer first is advanced n steps ahead of second. From that point on, both pointers move together, one step at a time. Since first is always exactly n nodes ahead of second, when first reaches the end of the list (NULL), second is exactly n nodes behind the end, i.e. at the n-th node from the end. This solves the problem in one pass, O(n) time and O(1) space, instead of the naive two-pass approach of first counting the total length.

Problem 7. Write a function to detect whether a singly linked list contains a cycle (loop), using Floyd's Cycle Detection Algorithm. *[6 marks]*

Solution:

```
int hasCycle(struct Node *head) {
    struct Node *slow = head, *fast = head;
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast)
```

```
            return 1;   // cycle detected
    }
    return 0;   // fast reached NULL: no cycle
}
```

This is also known as the tortoise-and-hare algorithm. Slow moves one step and fast moves two steps per iteration. If the list has no cycle, fast will reach NULL and the loop terminates normally. If the list does contain a cycle, fast eventually enters the loop and, because it gains on slow by one extra node per iteration inside a finite loop, it is guaranteed to eventually land exactly on slow, at which point slow == fast confirms a cycle. This runs in O(n) time and O(1) extra space, unlike a hash-set-based approach which would need O(n) extra space.

Problem 8. Write a function to remove duplicate values from an unsorted singly linked list, using a hash set to achieve O(n) time. *[6 marks]*

Solution:

```
void removeDuplicates(struct Node *head) {
    if (head == NULL) return;
    HashSet seen;
    insert(seen, head->data);
    struct Node *curr = head;
    while (curr->next != NULL) {
        if (contains(seen, curr->next->data)) {
            struct Node *dup = curr->next;
            curr->next = curr->next->next;
            free(dup);
        } else {
            insert(seen, curr->next->data);
            curr = curr->next;
        }
    }
}
```

A hash set records every value already seen. The list is scanned once with a pointer curr: for each following node, if its value is already present in the set, that node is unlinked (its predecessor's next is redirected past it) and freed; otherwise its value is added to the set and curr moves forward. Because hash set lookups and insertions are O(1) on average, the whole list is processed in O(n) time using O(n) extra space for the set. If extra space must be avoided (O(1) space), an $O(n^2)$ nested-loop approach comparing every node with every later node can be used instead.

Section C: Trace-Based Problems

Problem 9. A singly linked list is 1 -> 2 -> 3 -> 4 -> 5 -> 3 (i.e., node 5 points back to node 3, forming a cycle). Trace Floyd's algorithm (slow moves 1 step, fast moves 2 steps) and show at which node slow and fast meet. *[6 marks]*

Solution:

```
List (by position): 1 -> 2 -> 3 -> 4 -> 5 -> (back to 3)

Start:      slow=1, fast=1
Step 1:     slow=2, fast=3
Step 2:     slow=3, fast=5
Step 3:     slow=4, fast=4   (fast: 5 -> 3 -> 4)
Step 4:     slow=5, fast=3   (fast: 4 -> 5 -> 3)
Step 5:     slow=3, fast=2   (fast: 3 -> 4 -> 5)
Step 6:     slow=4, fast=4   (fast: 5 -> 3 -> 4)   -> slow == fast

Meeting point: node with value 4
```

Since fast moves twice as fast as slow, once both pointers are inside the cyclic portion of the list, fast gains one extra position on slow every step; within a finite loop it must eventually catch up exactly, and the two pointers meet at some node inside the cycle (here, node 4) — this meeting confirms a cycle exists, even though it does not directly give the starting node of the cycle (a further step, restarting one pointer from the head, is used for that).

Problem 10. Trace the nthFromEnd algorithm to find the 2nd node from the end of the list 10 -> 20 -> 30 -> 40 -> 50 -> NULL. *[5 marks]*

Solution:

```
List: 10 -> 20 -> 30 -> 40 -> 50 -> NULL,  n = 2

Advance 'first' by n=2 steps:
   first: 10 -> 20 -> 30     (first now at 30, second still at 10)

Move both together until first reaches NULL:
   first=30, second=10  ->  first=40, second=20
   first=40, second=20  ->  first=50, second=30
   first=50, second=30  ->  first=NULL, second=40

first is now NULL, so second (= 40) is the 2nd node from the end.
```

This matches the expected answer directly: in the 5-node list 10,20,30,40,50, the 2nd node from the end is indeed 40, confirming the two-pointer gap technique without ever explicitly counting the list length.

Problem 11. A doubly linked list is NULL <- 10 <-> 20 <-> 30 -> NULL. Trace the pointer changes to delete the middle node (20). *[5 marks]*

Solution:

```
Before:   NULL <- 10 <-> 20 <-> 30 -> NULL

Let target = node 20.  prevNode = 10,  nextNode = 30.
Step 1: prevNode->next = nextNode     (10's next now points to 30)
Step 2: nextNode->prev = prevNode     (30's prev now points to 10)
```

```
Step 3: free(target)                (node 20 is removed)

After:    NULL <- 10 <-> 30 -> NULL
```

Unlike a singly linked list, deleting a node from a doubly linked list requires updating links on both sides: the previous node's next pointer and the next node's prev pointer must both be redirected around the deleted node, so that the list remains correctly traversable in both the forward and backward directions after the deletion.

Problem 12. An array A = {4, 8, 15, 16, 23} is to be converted into a singly linked list by inserting each element at the end, in order. Trace the head pointer and the list after each insertion. *[5 marks]*

Solution:

```
Start:                head = NULL
Insert 4:              head -> 4 -> NULL
Insert 8:              head -> 4 -> 8 -> NULL
Insert 15:             head -> 4 -> 8 -> 15 -> NULL
Insert 16:             head -> 4 -> 8 -> 15 -> 16 -> NULL
Insert 23:             head -> 4 -> 8 -> 15 -> 16 -> 23 -> NULL
```

The head pointer itself only changes once, when the very first element (4) is inserted into the previously empty list; for every subsequent element, the head stays fixed and only the next pointer of the current last node is updated to include the new node, which is why repeated end-insertion without a tail pointer still costs O(n) per insertion (to find the current last node) even though the head is untouched.

Problem 13. Two singly linked lists intersect and merge into one tail: List 1: 1 -> 2 -> 3 -> 7 -> 8 -> NULL and List 2: 4 -> 5 -> 3 -> 7 -> 8 -> NULL (they share nodes 3, 7, 8). List 1 has length 5 and List 2 has length 5. Describe how the intersection node can be found in O(m + n) time and O(1) extra space, and identify the intersection node. *[6 marks]*

Solution:

5. Find the lengths of both lists: length1 = 5, length2 = 5.
6. Compute the difference d = |length1 - length2| = 0 in this case.
7. Advance the pointer of the longer list by d steps (here, d = 0, so no advance is needed since both lists are equal length).
8. Move both pointers forward together, one step at a time, comparing node addresses (not values) at each step, until the two pointers point to the same node.

```
p1: 1 -> 2 -> 3 -> 7 -> 8      p2: 4 -> 5 -> 3 -> 7 -> 8
Compare: 1 vs 4 (no) -> 2 vs 5 (no) -> 3 vs 3 (match!)
Intersection node: the node containing 3
```

Equalizing the starting position using the length difference ensures that once both pointers have the same number of nodes remaining to the end, they will hit the shared tail at exactly the same node if the lists do intersect. Since only two traversals of length at most max(m, n) are made (one to find lengths, one to find the intersection), and only a constant number of extra pointers are used, this runs in O(m + n) time and O(1) extra space — better than a hash-set approach, which would need O(m) extra space.