Exam Strategy

1. Draw diagrams for pointer manipulation.
2. Test edge cases immediately after coding.
3. Use helper functions (e.g., `reverse()` for palindrome check).
4. Comment steps to stay organized under time pressure.

---

1. Detecting and Removing Cycles

Key Idea:

- Use Floyd's Tortoise & Hare (fast/slow pointers). If they meet, there's a cycle.
- To remove: Reset slow to head, move both at same speed until their `next` pointers meet.

```
// Detect cycle
bool hasCycle(struct ListNode *head) {
    struct ListNode *slow = head, *fast = head;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) return true;
    }
    return false;
}

// Remove cycle (if exists)
void removeCycle(struct ListNode *head) {
    struct ListNode *slow = head, *fast = head;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
```

```
        if (slow == fast) break;
    }
    if (slow != fast) return; // No cycle
    slow = head;
    while (slow->next != fast->next) {
        slow = slow->next;
        fast = fast->next;
    }
    fast->next = NULL; // Break the cycle
}
```

Edge Cases:

- Empty list.
- Cycle at head.

Core Insight

- **Floyd's Algorithm (Tortoise & Hare)**:
  - **Why two pointers?** The fast pointer (`hare`) catches up to the slow (`tortoise`) at `O(n)` time if a cycle exists.
  - **Mathematical proof**: When they meet, the distance from the head to the cycle start equals the meeting point to the cycle start (modulo cycle length).

Key Steps

1. **Detection**: Move `slow` (1 step) and `fast` (2 steps) until they meet.
2. **Removal**: Reset `slow` to head, move both at 1 step. Their meeting `next` is the cycle start.

Why It Works

- **Distance analysis**: If the cycle has length `L`, the meeting point is `L - k` from the start (where `k` is the non-cycle part).

---

2. Merge Two Sorted Lists

Key Idea:

- Use a dummy node to build the merged list. Compare nodes and link the smaller one.

```
struct ListNode* mergeTwoLists(struct ListNode* l1, struct
ListNode* l2) {
    struct ListNode dummy;
    struct ListNode *tail = &dummy;
    dummy.next = NULL;
    while (l1 && l2) {
        if (l1->val <= l2->val) {
            tail->next = l1;
            l1 = l1->next;
        } else {
            tail->next = l2;
            l2 = l2->next;
        }
        tail = tail->next;
    }
    tail->next = l1 ? l1 : l2;
    return dummy.next;
}
```

Edge Cases:

- One list is empty.
- All nodes in one list are smaller than the other.

Core Insight

- Dummy Node Trick: Avoids edge cases (empty lists) and simplifies pointer updates.
- Greedy Approach: Always pick the smaller node, ensuring sorted order.

Key Steps

1. Compare heads of both lists.
2. Link the smaller node to the merged list.
3. Attach the remaining nodes of the non-empty list.

Why It Works

- **Invariant**: The merged list is always sorted after each step.

---

3. Find Middle Node

Key Idea:

- Fast pointer moves 2 steps, slow moves 1. When fast reaches end, slow is at middle.

```c
struct ListNode* middleNode(struct ListNode* head) {
    struct ListNode *slow = head, *fast = head;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}
```

Edge Cases:

- Even length: Returns the second middle.

Core Insight

- **Fast & Slow Pointers**:
  - Fast reaches the end when slow is at the middle.

- For even lengths, slow points to the second middle (consistent with standard definitions).

Why It Works

- Distance: Fast pointer travels twice as far as slow (`2d = d + n` ⇒ `d = n`).

---

4. Palindrome Check

Key Idea:

- Find middle, reverse the second half, compare with first half.

```
bool isPalindrome(struct ListNode* head) {
    // Find middle
    struct ListNode *slow = head, *fast = head;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
    }
    // Reverse second half
    struct ListNode *prev = NULL, *curr = slow;
    while (curr) {
        struct ListNode *next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    // Compare
    while (prev) {
        if (head->val != prev->val) return false;
        head = head->next;
        prev = prev->next;
    }
```

```
    return true;
}
```

Edge Cases:

- Odd/even length lists.
- Single-node list.

Core Insight

1. Middle Finding: Split the list into two halves.
2. Reversal: Reverse the second half to compare with the first.
3. Comparison: Traverse both halves simultaneously.

Key Nuance

- Odd-length lists: The middle node is ignored during comparison.

Why It Works

- Symmetry: A palindrome reads the same forwards and backwards.

---

5. Intersection Point of Two Lists

Key Idea:

- Traverse both lists to find lengths. Align their starting points, then traverse together.

```
struct ListNode *getIntersectionNode(struct ListNode
*headA, struct ListNode *headB) {
    // Calculate lengths
    int lenA = 0, lenB = 0;
    struct ListNode *a = headA, *b = headB;
    while (a) { lenA++; a = a->next; }
    while (b) { lenB++; b = b->next; }
```

```
    // Align pointers
    a = headA; b = headB;
    while (lenA > lenB) { a = a->next; lenA--; }
    while (lenB > lenA) { b = b->next; lenB--; }
    // Find intersection
    while (a != b) {
        a = a->next;
        b = b->next;
    }
    return a;
}
```

Edge Cases:

- No intersection.
- One list is a subset of the other.

Core Insight

- Alignment: Equalize the starting points by skipping the excess nodes in the longer list.
- Synchronized traversal: Both pointers will meet at the intersection (if it exists).

Why It Works

- Path equivalence: After alignment, both pointers traverse the same distance to the intersection.

---

6. Clone List with Random Pointers

Key Idea:

- Use a hash map to map original nodes to copies. Then link `next` and `random` pointers.

```
struct Node* copyRandomList(struct Node* head) {
    if (!head) return NULL;
    // Step 1: Create copies and map originals to copies
    struct Node *curr = head;
    while (curr) {
        struct Node *copy = malloc(sizeof(struct Node));
        copy->val = curr->val;
        copy->next = curr->next;
        curr->next = copy;
        curr = copy->next;
    }
    // Step 2: Assign random pointers
    curr = head;
    while (curr) {
        if (curr->random)
            curr->next->random = curr->random->next;
        curr = curr->next->next;
    }
    // Step 3: Separate original and copied lists
    struct Node *newHead = head->next;
    curr = head;
    while (curr) {
        struct Node *copy = curr->next;
        curr->next = copy->next;
        curr = curr->next;
        if (curr) copy->next = curr->next;
    }
    return newHead;
}
```

Edge Cases:

- `random` pointers pointing to `NULL`.
- Single-node list.

Core Insight

1. **Interleaving Copies**: Creates a mapping without extra space (`original->next = copy`).
2. **Random Pointer Assignment**: `copy->random = original->random->next`.
3. **List Separation**: Restore original list and extract the copy.

Why It Works

- **Implicit Mapping**: The interleaved structure preserves node relationships.