Linked Lists Structures, Operations, and Types in Data Structures Practice Test (Sample)

Study Guide

Everything you need from our exam experts!

Table of Contents

Introduction

Preparing for a certification exam can feel overwhelming, but with the right tools, it becomes an opportunity to build confidence, sharpen your skills, and move one step closer to your goals. At Examzify, we believe that effective exam preparation isn't just about memorization, it's about understanding the material, identifying knowledge gaps, and building the test-taking strategies that lead to success.

This guide was designed to help you do exactly that.

Whether you're preparing for a licensing exam, professional certification, or entry-level qualification, this book offers structured practice to reinforce key concepts. You'll find a wide range of multiple-choice questions, each followed by clear explanations to help you understand not just the right answer, but why it's correct.

The content in this guide is based on real-world exam objectives and aligned with the types of questions and topics commonly found on official tests. It's ideal for learners who want to:

- **Practice answering questions under realistic conditions,**
- **Improve accuracy and speed,**
- **Review explanations to strengthen weak areas, and**
- **Approach the exam with greater confidence.**

We recommend using this book not as a stand-alone study tool, but alongside other resources like flashcards, textbooks, or hands-on training. For best results, we recommend working through each question, reflecting on the explanation provided, and revisiting the topics that challenge you most.

Remember: successful test preparation isn't about getting every question right the first time, it's about learning from your mistakes and improving over time. Stay focused, trust the process, and know that every page you turn brings you closer to success.

Let's begin.

How to Use This Guide

This guide is designed to help you study more effectively and approach your exam with confidence. Whether you're reviewing for the first time or doing a final refresh, here’s how to get the most out of your Examzify study guide:

1. Start with a Diagnostic Review

Skim through the questions to get a sense of what you know and what you need to focus on. Your goal is to identify knowledge gaps early.

2. Study in Short, Focused Sessions

Break your study time into manageable blocks (e.g. 30 – 45 minutes). Review a handful of questions, reflect on the explanations.

3. Learn from the Explanations

After answering a question, always read the explanation, even if you got it right. It reinforces key points, corrects misunderstandings, and teaches subtle distinctions between similar answers.

4. Track Your Progress

Use bookmarks or notes (if reading digitally) to mark difficult questions. Revisit these regularly and track improvements over time.

5. Simulate the Real Exam

Once you're comfortable, try taking a full set of questions without pausing. Set a timer and simulate test-day conditions to build confidence and time management skills.

6. Repeat and Review

Don’t just study once, repetition builds retention. Re-attempt questions after a few days and revisit explanations to reinforce learning. Pair this guide with other Examzify tools like flashcards, and digital practice tests to strengthen your preparation across formats.

There’s no single right way to study, but consistent, thoughtful effort always wins. Use this guide flexibly, adapt the tips above to fit your pace and learning style. You've got this!

Questions

1. How does a doubly linked list differ from a singly linked list in terms of per-node pointers and memory usage?

A. Doubly uses next and prev; higher per-node memory overhead but enables bidirectional traversal

B. Doubly uses two next pointers

C. Doubly uses only prev pointer

D. Singly uses two prev pointers

2. What are the time and space complexities of Floyd's cycle detection algorithm?

A. Time: O(n^2); Space: O(1)

B. Time: O(n); Space: O(n)

C. Time: O(n); Space: O(1)

D. Time: O(1); Space: O(1)

3. Which statement is true about a circular singly linked list?

A. The list has a NULL termination.

B. The last node's next pointer is NULL.

C. The list cannot be traversed backwards.

D. The last node points back to the head, forming a loop.

4. When reversing a block of k nodes in a singly linked list, what happens to the order of nodes within that block?

A. The order of nodes within the block remains the same.

B. The order of nodes inside the block is reversed.

C. The block is moved to the end of the list.

D. The direction of traversal is inverted globally.

5. Which approach is commonly used to merge two sorted singly linked lists into a single sorted list?

A. Attach the lists end-to-end without comparing values.

B. Attach the larger node first.

C. Use a dummy head and repeatedly attach the smaller node from either list.

D. First reverse both lists, then concatenate.

6. What is the role of the Node structure within the LList class?

A. It defines the structure of each node, including data and pointers to previous and next nodes

B. It stores the entire list as an array

C. It manages memory for the list

D. It tracks how many nodes are in the list

7. Which method finds the middle element of a singly linked list in one pass?

A. Slow and fast pointers: move slow by 1 and fast by 2; when fast reaches end, slow is middle; O(n)

B. First compute length, then move to length/2 from head.

C. Push nodes onto a stack until you reach the middle; pop to middle; O(n) extra space

D. Return the node after head.

8. Explain how to delete a node from a doubly linked list given a pointer to that node.

A. Relink neighbors: if node.prev != NULL, node.prev.next = node.next; if node.next != NULL, node.next.prev = node.prev; free node.

B. Simply free the node without relinking neighbors.

C. Set node.prev.next = NULL; node.next.prev = NULL; free node.

D. Update only node.next and leave node.prev intact.

9. How do you initialize a linked list?

A. Set the head and tail pointers to nullptr

B. Create a sentinel node

C. Set both head and tail pointers to a dummy node

D. Do nothing; initialization is automatic

10. What is the defining feature of a threaded linked list that enables traversal without recursion?

A. It uses pointers to in-order successor to move to the next node.

B. It stores all nodes in a separate array.

C. It uses backward pointers to previous node.

D. It clones the list and traverses the clone.

Answers

1. C
2. C
3. D
4. B
5. C
6. A
7. A
8. A
9. A
10. A

Explanations

1. How does a doubly linked list differ from a singly linked list in terms of per-node pointers and memory usage?

A. Doubly uses next and prev; higher per-node memory overhead but enables bidirectional traversal

B. Doubly uses two next pointers

C. Doubly uses only prev pointer

D. Singly uses two prev pointers

In a doubly linked list, each node stores two pointers: one to the next node and one to the previous node. This is why per-node memory usage is higher than in a singly linked list, where each node keeps only a single next pointer. The payoff is bidirectional traversal: you can move forward using the next pointer and backward using the prev pointer without starting from the head each time. The extra pointer is the source of the increased memory overhead, but it enables convenient backwards navigation, easier deletions, and simpler insertions before a given node. The option that mentions two next pointers would not provide backward navigation on its own, the idea of using only a previous pointer would eliminate forward traversal, and a singly linked list uses one pointer, not two prev pointers.

2. What are the time and space complexities of Floyd's cycle detection algorithm?

A. Time: O(n^2); Space: O(1)

B. Time: O(n); Space: O(n)

C. Time: O(n); Space: O(1)

D. Time: O(1); Space: O(1)

Two pointers move through the list at different speeds, the fast one advancing two steps for every step the slow one takes. This setup guarantees a meeting somewhere: if there's a cycle, they will eventually collide inside it, and if there isn't, the fast pointer will reach the end. The number of steps before they meet (or before the end is reached) scales linearly with the number of nodes in the list, so the time complexity is O(n). The algorithm uses only a fixed number of pointers, so the extra space required is constant, O(1). This makes it the best answer for time linear in n and space constant.

3. Which statement is true about a circular singly linked list?

A. The list has a NULL termination.

B. The last node's next pointer is NULL.

C. The list cannot be traversed backwards.

D. The last node points back to the head, forming a loop.

In a circular singly linked list, the defining trait is that the last node links back to the first node, creating a loop. This means there is no NULL termination like in a linear singly linked list; instead, following next pointers eventually brings you back to the head. That loop-back behavior is what makes the structure circular and distinguishes it from a normal list. So the statement that the last node points back to the head, forming a loop, captures this essential property. The other descriptions don't fit because a circular list doesn't end with NULL, so there isn't a NULL termination and the last node's next isn't NULL. Also, while you can't traverse backwards in a singly linked list, that's true for all singly linked lists and isn't what uniquely defines the circular form.

4. When reversing a block of k nodes in a singly linked list, what happens to the order of nodes within that block?

A. The order of nodes within the block remains the same.

B. The order of nodes inside the block is reversed.

C. The block is moved to the end of the list.

D. The direction of traversal is inverted globally.

Reversing a block of k consecutive nodes in a singly linked list rearranges the next pointers inside that block so the order of those k nodes becomes the opposite of what it was. The first node in the block ends up last, and the last node ends up first, while the connections to the rest of the list stay the same at the block's ends. For example, A -> B -> C -> D, reversing the first three nodes yields C -> B -> A -> D. The rest of the list is untouched except that it now connects to the new ends of the reversed block.

5. Which approach is commonly used to merge two sorted singly linked lists into a single sorted list?

A. Attach the lists end-to-end without comparing values.

B. Attach the larger node first.

C. Use a dummy head and repeatedly attach the smaller node from either list.

D. First reverse both lists, then concatenate.

Merging two sorted lists by always choosing the smaller next node preserves the overall order as you build the result. Using a dummy head makes this process clean and robust because you don't need special handling for the first node—you simply attach nodes to a tail as you go. Here's how it works: create a dummy node and a tail pointer that starts at the dummy. Have pointers to the current nodes of both lists. Compare their values; attach the smaller node to tail.next, and advance that list's pointer. Move the tail forward. Repeat until one list runs out, then link the remaining non-empty list to tail.next. Finally, return dummy.next as the head of the merged list. This approach is optimal for correctness and efficiency: it maintains sorted order by always picking the smaller current value, handles edge cases smoothly with the dummy head, runs in linear time proportional to the total number of nodes (O(n + m)), and uses constant extra space since it reuses existing nodes. Attaching without comparisons would break order, and reversing lists before concatenation would not yield a sorted result without extra steps.

6. What is the role of the Node structure within the LList class?

A. It defines the structure of each node, including data and pointers to previous and next nodes

B. It stores the entire list as an array

C. It manages memory for the list

D. It tracks how many nodes are in the list

In a linked list, the Node is the building block that defines what each element contains and how elements connect to one another. Each Node holds the actual data value and the references to its neighbors—previous and next in a doubly linked list (or just next in a singly linked list). The LList is built by linking these Nodes together through those pointers, so you can traverse from one node to the next, insert new nodes, or remove them by updating the relevant pointers. The list as a whole isn't stored in a single array; instead, it's a chain of Node objects connected by their links, with the list maintaining a reference to the starting node (and sometimes the ending node). Memory management and counting the number of nodes are handled by other parts of the LList implementation, not by the Node itself. So, the Node's role is to define the structure of each node, including data and pointers to and from neighboring nodes.

7. Which method finds the middle element of a singly linked list in one pass?

A. Slow and fast pointers: move slow by 1 and fast by 2; when fast reaches end, slow is middle; O(n)

B. First compute length, then move to length/2 from head.

C. Push nodes onto a stack until you reach the middle; pop to middle; O(n) extra space

D. Return the node after head.

Finding the middle in one pass uses the two-pointer idea: have a slow pointer move one node at a time and a fast pointer move two nodes at a time. As you traverse, the fast pointer quickly reaches the end while the slow pointer has advanced roughly half as many steps. When the fast pointer reaches the end, the slow pointer lands at the middle. This gives you the middle in a single traversal with O(n) time and only O(1) extra space, since you're just keeping two references. This works for lists of any length: for odd-length lists, the slow pointer ends up exactly at the middle element; for even-length lists, you end up at the lower of the two middle positions in the usual implementation, which is still a valid "middle" in many contexts. Edge cases like an empty list or a single node are naturally handled by the pointers reaching their end conditions. Other approaches require extra passes or extra storage: computing the length first is two passes; using a stack needs O(n) extra space; simply returning the node after head gives a position that isn't the middle in general.

8. Explain how to delete a node from a doubly linked list given a pointer to that node.

A. Relink neighbors: if node.prev != NULL, node.prev.next = node.next; if node.next != NULL, node.next.prev = node.prev; free node.

B. Simply free the node without relinking neighbors.

C. Set node.prev.next = NULL; node.next.prev = NULL; free node.

D. Update only node.next and leave node.prev intact.

In a doubly linked list, deleting a node must remove it from the chain without leaving the rest of the list broken. You do this by relinking its neighbors: connect the previous node's next pointer to the target's next node, and connect the next node's prev pointer to the target's previous node. If either neighbor is absent (the node is at the head or tail), you skip that side safely. After the links are updated, free the target node. This preserves the continuity of the list and prevents any dangling references. The method that relinks both sides before freeing is the right one because simply freeing the node leaves the neighboring nodes still pointing to a memory region that's no longer part of the list, which can crash or corrupt the list. Setting one side to NULL or attempting to NULL out pointers without connecting the two neighbors would either truncate the list or leave broken links. Freeing without relinking, or truncating only part of the links, breaks the integrity of the structure.

9. How do you initialize a linked list?

A. Set the head and tail pointers to nullptr

B. Create a sentinel node

C. Set both head and tail pointers to a dummy node

D. Do nothing; initialization is automatic

Initializing a linked list means establishing a well-defined empty state so the rest of the code can safely build from it. In a typical implementation that keeps both a head and a tail pointer, the cleanest way to represent an empty list is to set both pointers to null. This clearly signals that there are no nodes yet, and it makes future operations straightforward: when you insert the first node, you create it and update both head and tail to point to that node (or update head and then tail if you're maintaining it). With this setup, checks like "is the list empty?" are simply head == null, and you avoid dereferencing garbage pointers. Using a sentinel or dummy node is a valid alternative design, but it isn't the standard initialization that signals emptiness—those approaches start with a node already in the list and adjust how you interpret the head pointer. Similarly, doing nothing leaves pointers uninitialized in many languages, which is unsafe and leads to undefined behavior. So the most direct and robust way to start is to set head and tail to null to represent an empty list.

10. What is the defining feature of a threaded linked list that enables traversal without recursion?

A. It uses pointers to in-order successor to move to the next node.

B. It stores all nodes in a separate array.

C. It uses backward pointers to previous node.

D. It clones the list and traverses the clone.

Threaded linked lists embed a direct link to the next node in the traversal order, typically the in-order successor. This means every node carries a pointer to the node that should come next when traversing the structure. With these successor pointers, you can walk through the entire list by simply following each node's successor, in a simple loop, without using recursion or a stack. This threading eliminates the need to backtrack or keep additional state while traversing in the desired order. Storing all nodes in a separate array would be external to the list's structure, so it doesn't inherently enable non-recursive traversal. Backward pointers help with reverse traversal, not the smooth forward in-order traversal without extra mechanisms. Cloning the list and traversing the clone is unnecessary and doesn't reflect how the threading mechanism provides a built-in non-recursive path through the original structure.

Next Steps

Congratulations on reaching the final section of this guide. You've taken a meaningful step toward passing your certification exam and advancing your career.

As you continue preparing, remember that consistent practice, review, and self-reflection are key to success. Make time to revisit difficult topics, simulate exam conditions, and track your progress along the way.

If you need help, have suggestions, or want to share feedback, we'd love to hear from you. Reach out to our team at hello@examzify.com.

Or visit your dedicated course page for more study tools and resources:

https://linkedlistsstructuresopstypes.examzify.com

We wish you the very best on your exam journey. You've got this!