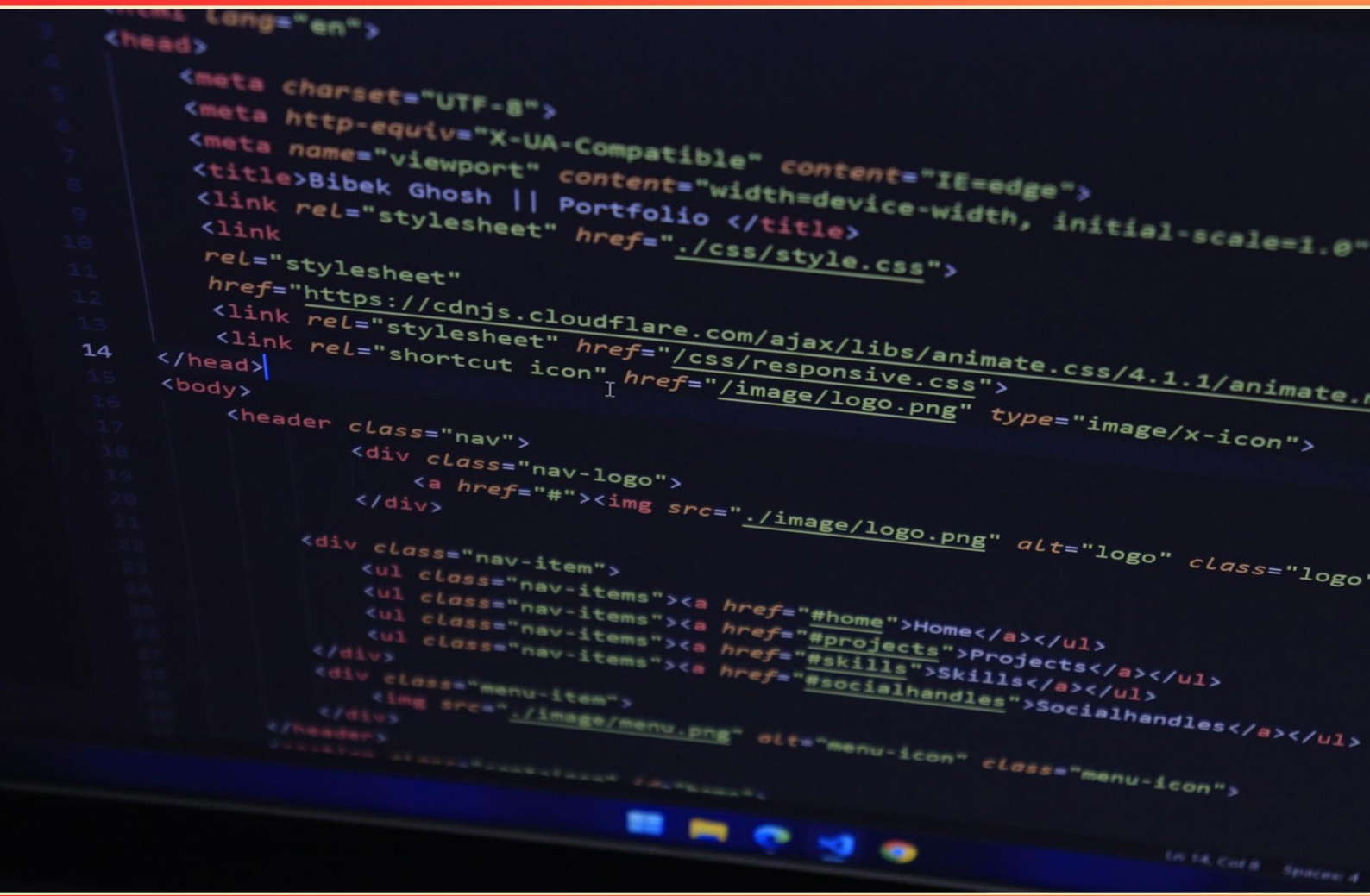
LINKED LISTS STRUCTURES, OPERATIONS, AND TYPES IN DATA STRUCTURES PRACTICE TEST (SAMPLE)

STUDY GUIDE



SAMPLE STUDY GUIDE WITH LIMITED QUESTIONS



VISIT US ONLINE FOR THE FULL VERSION STUDY GUIDE

https://linkedlistsstructuresopstypes.examzify.com

Copyright © 2026 by Examzify - A Kaluba Technologies Inc. product.

ALL RIGHTS RESERVED.

No part of this book may be reproduced or transferred in any form or by any means, graphic, electronic, or mechanical, including photocopying, recording, web distribution, taping, or by any information storage retrieval system, without the written permission of the author.

Notice: Examzify makes every reasonable effort to obtain accurate, complete, and timely information about this product from reliable sources.

SAMPLE

Table of Contents

Copyright 1

Table of Contents 2

Introduction 3

How to Use This Guide 4

Questions 5

Answers 8

Explanations 10

Next Steps 16

SAMPLE

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.

SAMPLE

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!

SAMPLE

Questions

SAMPLE

1. How would you implement a function to remove the last node from a singly linked list with a tail pointer?

A. Even with a tail pointer, removing the last node typically requires locating the second-last node to update its next to NULL and updating tail.

B. Tail pointer allows O(1) removal of last node by updating tail to NULL.

C. You can just set tail.next = NULL.

D. You must traverse from tail backwards.

2. How would you implement a memory-efficient queue using a singly linked list with head and tail pointers, and what are the time complexities of enqueue and dequeue?

A. Maintain head and tail; enqueue O(1), dequeue O(1)

B. Use only a head pointer; both operations O(n)

C. Use a circular buffer; both O(1) average

D. Rebuild the list on every operation; O(n)

3. Which operation on a singly linked list is guaranteed to be O(1) only when a tail pointer is maintained?

A. Inserting at the tail.

B. Inserting at the head.

C. Deleting the head.

D. Traversing to the end to compute length.

4. To move from a node toward the previous node in a doubly linked list, which pointer do you follow?

A. Next.

B. Prev.

C. Parent.

D. Child.

5. In a doubly linked list, what pointers exist in each node?

A. Next and Prev.

B. Left and Right.

C. Parent and Child.

D. Top and Bottom.

SAMPLE

6. Compared to an O(1) space palindrome check, what is the space complexity of the stack-based approach?

A. O(1) space

B. O(n log n) space

C. O(n) space

D. O(n^2) space

7. How can you safely traverse and print a circular linked list without risking an infinite loop?

A. Traverse until you reach a null pointer

B. Start at head and stop when you return to head after visiting at least once

C. Use a visited flag per node

D. Recursively traverse until stack overflow

8. Explain the recursive method to reverse a singly linked list and the base/recursive steps.

A. Sort the list first then reverse.

B. Base: head is NULL or head.next is NULL; return head. Recursive: reverse head.next to get rest; then set head.next.next = head; set head.next = NULL; return rest.

C. Base: head is non-empty; swap values.

D. Base: use a loop to reverse pointers without recursion.

9. What is a key property of circular linked lists that differentiates them from linear linked lists?

A. There is always a sentinel node at end

B. There is no null terminator; traversal wraps around

C. Nodes store a hash value to signal end

D. Traversal ends at a designated 'end' node

10. What is the trade-off of using a dummy node versus not using one in insertion at the head?

A. A dummy node increases complexity and has no benefits.

B. A dummy node simplifies edge cases and head insertions; without one you must treat head specially, but you save a node.

C. Using a dummy node makes insertion slower by a constant factor but no other effect.

D. Dummy node guarantees O(1) time but requires dynamic memory management.

SAMPLE

Answers

SAMPLE

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

SAMPLE

Explanations

SAMPLE

1. How would you implement a function to remove the last node from a singly linked list with a tail pointer?

A. Even with a tail pointer, removing the last node typically requires locating the second-last node to update its next to NULL and updating tail.

B. Tail pointer allows O(1) removal of last node by updating tail to NULL.

C. You can just set tail.next = NULL.

D. You must traverse from tail backwards.

Removing the last node in a singly linked list with a tail pointer hinges on not having a direct pointer to the node before the last. In a singly list, each node points forward, and the tail points to the final node whose next is NULL. To remove that last node, you need to detach it by changing the next field of the node that comes just before it to NULL, and then update the tail to that preceding node. Since there's no backward link, you typically start at the head and walk forward until you find the node whose next is the tail. If the list has only one node, you simply set both head and tail to NULL. This operation is O(n) in the general case because of the traversal, unless you maintain an extra pointer to the previous node or convert to a doubly linked list. So the correct approach is to locate the second-last node, set its next to NULL, and move the tail to that node.

2. How would you implement a memory-efficient queue using a singly linked list with head and tail pointers, and what are the time complexities of enqueue and dequeue?

A. Maintain head and tail; enqueue O(1), dequeue O(1)

B. Use only a head pointer; both operations O(n)

C. Use a circular buffer; both O(1) average

D. Rebuild the list on every operation; O(n)

When you aim for minimal extra state in a queue built from a singly linked list, you would avoid keeping extra pointers beyond what's strictly needed. Rebuilding the list on every operation achieves that: you don't store a persistent tail pointer or other helper data, so the memory footprint stays small. The trade-off is time. To perform each enqueue or dequeue, you must reconstruct or relink the entire list, so every operation touches all elements. That makes both enqueue and dequeue run in linear time, O(n). In this light, the described approach aligns with memory efficiency at the cost of slower operations. If you did keep a head and tail pointer, you'd get O(1) for both operations, but you'd be using a tiny amount more space for that extra pointer.

SAMPLE

3. Which operation on a singly linked list is guaranteed to be O(1) only when a tail pointer is maintained?

A. Inserting at the tail.

B. Inserting at the head.

C. Deleting the head.

D. Traversing to the end to compute length.

Maintaining a tail pointer changes how you access the end of a list. In a singly linked list, each node points to the next one, so to append a new node at the tail without a tail pointer you must start at the head and walk through every node until you reach the last one. That traversal takes linear time, O(n). But if you keep a pointer directly to the last node, you can link the new node after the current tail and then update the tail pointer, all in constant time, O(1). Other operations don't depend on the tail pointer in the same way: inserting at the head is O(1) because you just create the new node and re-point the head. Deleting the head is also O(1) since you move the head pointer to the next node. Traversing to the end to compute length is O(n) because you must visit every node, unless you maintain an additional length counter. So, inserting at the tail is the operation that becomes guaranteed O(1) specifically with a tail pointer.

4. To move from a node toward the previous node in a doubly linked list, which pointer do you follow?

A. Next.

B. Prev.

C. Parent.

D. Child.

Navigation in a doubly linked list relies on two pointers: one to the next node and one to the previous node. To move toward the node that comes before the current one, you follow the previous pointer. This pointer directly links each node to its predecessor, so stepping back in the list is accomplished by jumping via that reference. The next pointer, in contrast, moves you forward to the following node. Terms like parent and child belong to tree structures, not doubly linked lists, so they aren't used here. If you're at the head, the previous pointer is typically null, and if you're at the tail, the next pointer is typically null.

5. In a doubly linked list, what pointers exist in each node?

A. Next and Prev.

B. Left and Right.

C. Parent and Child.

D. Top and Bottom.

In a doubly linked list each node stores two pointers: one to the next node and one to the previous node. These two links, typically called next and prev, allow you to traverse the list in either direction. The ability to move backward as well as forward is what distinguishes a doubly linked list from a singly linked list, which only has a next pointer. At the ends, the head's previous pointer and the tail's next pointer are usually null, marking the boundaries. Other names like left/right, parent/child, or top/bottom aren't used to describe the standard structure of a doubly linked list.

SAMPLE

6. Compared to an O(1) space palindrome check, what is the space complexity of the stack-based approach?

A. O(1) space

B. O(n log n) space

C. O(n) space

D. O(n^2) space

The main idea here is how much extra storage a stack-based palindrome check needs as the input grows. With a stack-based approach you push characters onto the stack so you can pop them in reverse order to compare with the second half of the string. For a string of length n, you end up storing about n characters (or n/2 if you stop at the middle, but that's still proportional to n). So the extra space grows linearly with the input, giving O(n) space. In contrast, the constant-space version uses only a couple of indices or pointers and a few variables, hence O(1) space. The other options imply sublinear or superlinear growth that doesn't match how a stack holds elements, so they aren't correct.

7. How can you safely traverse and print a circular linked list without risking an infinite loop?

A. Traverse until you reach a null pointer

B. Start at head and stop when you return to head after visiting at least once

C. Use a visited flag per node

D. Recursively traverse until stack overflow

In a circular linked list there is no null terminator to signal the end, so you must detect when you've completed one full cycle. The reliable way is to start at the head, print it, and then move forward until you return to the head after visiting at least once. This ensures you traverse every node exactly one time and stop, since the traversal naturally cycles back to the starting node. Handle the empty-list case first: if the head is null, print nothing. This method avoids infinite looping because the stop condition is returning to the starting node, which only happens after exactly one full loop. Relying on a null pointer would fail here because there isn't one in a circular list. Using a per-node visited flag is possible but introduces extra state to manage and isn't as clean. Recursively traversing would also risk infinite recursion and stack overflow due to the cycle.

SAMPLE

8. Explain the recursive method to reverse a singly linked list and the base/recursive steps.

A. Sort the list first then reverse.

B. Base: head is NULL or head.next is NULL; return head. Recursive: reverse head.next to get rest; then set head.next.next = head; set head.next = NULL; return rest.

C. Base: head is non-empty; swap values.

D. Base: use a loop to reverse pointers without recursion.

The idea being tested is reversing a singly linked list by changing the links through recursion, not just moving values. The correct approach uses a base case that detects when you've reached the end (or an empty list) and then rewires pointers on the way back up the call stack. Base case: if the list is empty or has a single node, you can return that node as the new head, since nothing else needs to be done. Recursive step: reverse the rest of the list starting at the next node (let rest = reverse(head.next)). After this call, rest is the head of the already-reversed portion. To put the current node at the end of that reversed portion, set head.next.next = head. This links the next node back to the current node, effectively putting head after the reversed tail. Then detach the current node from the rest by setting head.next = NULL. Finally, return rest as the new head of the reversed list. For example, turning 1->2->3 into 3->2->1 works by first reversing 2->3 to 3->2, then linking 1 after 2 and setting 1's next to NULL, yielding 3->2->1. Swapping values, while it may appear to "reverse" the sequence of data, does not rearrange the actual node links, so it isn't the same recursive pointer reversal. A loop-based approach is iterative, not recursive.

9. What is a key property of circular linked lists that differentiates them from linear linked lists?

A. There is always a sentinel node at end

B. There is no null terminator; traversal wraps around

C. Nodes store a hash value to signal end

D. Traversal ends at a designated 'end' node

The main idea is that circular linked lists form a cycle, so there is no null terminator to signal the end. In a circular list, the next pointer of the last node points back to the first node, which means traversal can keep going indefinitely unless you stop it intentionally—typically by stopping when you return to your starting node or after counting a known number of nodes. This wrap-around behavior is what sets circular lists apart from linear ones, which end when their next pointer is null. That's why this option is the best: it captures the absence of a terminating null and the wrap-around traversal that defines circular lists. A sentinel node at the end isn't required by the concept, and storing a hash value or ending at a designated end node would imply a non-circular structure, which doesn't align with how circular lists operate.

SAMPLE

10. What is the trade-off of using a dummy node versus not using one in insertion at the head?

A. A dummy node increases complexity and has no benefits.

B. A dummy node simplifies edge cases and head insertions; without one you must treat head specially, but you save a node.

C. Using a dummy node makes insertion slower by a constant factor but no other effect.

D. Dummy node guarantees O(1) time but requires dynamic memory management.

Using a dummy (sentinel) node as a fixed anchor makes insertion at the head simpler and less error-prone. With a dummy in place, you always insert after the dummy, so you don't need a special case for when the list is empty or when the new element becomes the first real node. This unifies the insertion logic across all positions and reduces the risk of mistakes when updating the head pointer. The trade-off is memory and a tiny constant-factor cost. You pay for one extra node that doesn't hold meaningful data, and every operation must traverse or reference the dummy as part of the path, which adds a small overhead. Without a dummy, you save that node and avoid that extra indirection, but you must explicitly handle the head as a special case—checking for an empty list and updating the head pointer manually—which increases code complexity and the chance of subtle bugs.

SAMPLE

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!

SAMPLE