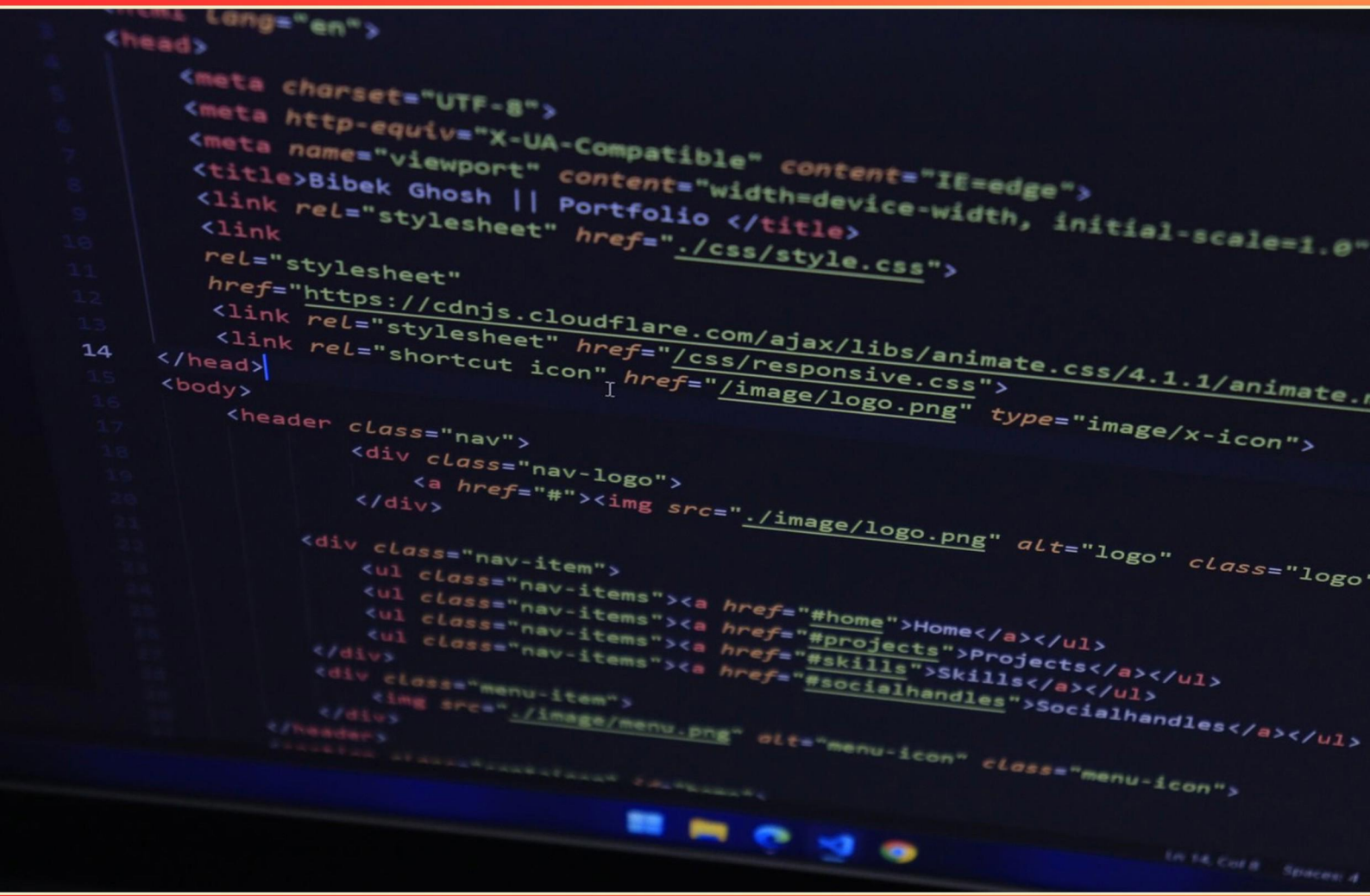
LINKED LISTS STRUCTURES, OPERATIONS, AND TYPES IN DATA STRUCTURES PRACTICE TEST (SAMPLE)

STUDY GUIDE



SAMPLE STUDY GUIDE WITH LIMITED QUESTIONS

VISIT US ONLINE FOR THE FULL VERSION STUDY GUIDE

https://linkedlistsstructuresopstypes.examzify.com

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. What is the effect of maintaining a tail pointer on the amortized cost of append operations when the list is used as a queue?

A. Enables O(1) amortized append by avoiding traversal to the end

B. Makes append slower due to extra pointer updates

C. Has no effect on cost

D. Increases space consumption without benefit

2. 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.

3. How do you remove duplicates from a sorted singly linked list?

A. Traverse with a pointer; if current value equals next value, bypass the duplicate by linking current.next to current.next.next; O(n).

B. Use a hash set to track seen values.

C. Sort the list, then remove duplicates.

D. Always delete the current node after checking next.

4. What is the purpose of the copy constructor in a linked list?

A. To perform a shallow copy

B. To create a new list by deep-copying another list

C. To delete all nodes in the original

D. To reserve memory for future nodes

5. What is the expected time complexity to search for an element in a skip list compared to a standard singly linked list?

A. Skip list search is O(n) on average; standard singly linked list search is O(n).

B. Skip list search is O(1); singly linked list is O(n).

C. Skip list search is O(log n) on average; standard singly linked list search is O(n).

D. Skip list search is O(log n) worst-case; singly linked list is O(1).

6. How do you remove duplicates from an unsorted singly linked list without extra storage?

A. Use a hash set to track seen values.

B. Sort the list to bring duplicates together.

C. Use two nested loops to compare each node with the rest; O(n^2) time, O(1) space.

D. Slide a window pointer and delete duplicates.

7. 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

8. What is the standard approach to inserting at the tail when no tail pointer exists?

A. Traverse to the last node, set last.next = new; new.next = null.

B. Insert at the head by using head pointer.

C. Set head.next = new; new.next = null.

D. Create a new list with the node as the only element.

9. If you are given only a reference to a node (not the head) and you know it is not the tail, how can you delete that node?

A. Delete by removing the current node from the list using a head pointer.

B. Copy the data from the next node into the current node, set current.next = current.next.next; delete the next node; O(1)

C. Replace the current node with a new node carrying the same data.

D. Set current = current.next; deallocate the old node.

10. What does the clear() method do in the LList class?

A. It clears the list but preserves the allocated memory.

B. It removes all nodes and resets the list.

C. It only empties the first node.

D. It performs a shallow clear without deallocating nodes.

Answers

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

Explanations

1. What is the effect of maintaining a tail pointer on the amortized cost of append operations when the list is used as a queue?

A. Enables O(1) amortized append by avoiding traversal to the end

B. Makes append slower due to extra pointer updates

C. Has no effect on cost

D. Increases space consumption without benefit

Keeping a tail pointer lets you add at the end in constant time, which makes each append operation O(1) and thus O(1) amortized when the list is used as a queue. With a tail pointer you directly link the new node after the current tail and update the tail reference; you don't need to walk from the head to the last node. Without a tail pointer, every append would require traversing to the end, making the cost proportional to the list length and hurting the amortized cost over a sequence of operations. The tail pointer adds a tiny constant-space overhead (one extra reference) but provides a real time gain. So maintaining the tail pointer enables O(1) amortized append by avoiding traversal.

2. 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.

3. How do you remove duplicates from a sorted singly linked list?

A. Traverse with a pointer; if current value equals next value, bypass the duplicate by linking current.next to current.next.next; O(n).

B. Use a hash set to track seen values.

C. Sort the list, then remove duplicates.

D. Always delete the current node after checking next.

In a sorted singly linked list, duplicates appear next to each other, so you can remove them in one pass by adjusting next pointers as you go. Use a pointer to the current node and check the next node: if the current value equals the next value, skip the next node by setting current.next to current.next.next. Do not advance when you remove a node; after skipping, check again since another duplicate might follow. If they're different, simply move to the next node. This approach keeps the first occurrence of each value and removes all duplicates in a single traversal. It runs in O(n) time because each node is visited a constant number of times, and uses O(1) extra space since only a couple of pointers are needed. Using a hash set would add extra space without benefit here, sorting again is unnecessary because the list is already sorted, and deleting the current node every time would discard non-duplicate nodes and complicate maintenance of the list structure.

4. What is the purpose of the copy constructor in a linked list?

A. To perform a shallow copy

B. To create a new list by deep-copying another list

C. To delete all nodes in the original

D. To reserve memory for future nodes

The copy constructor for a linked list is used to produce an entirely new list that is an independent duplicate of an existing one. It does a deep copy by allocating new nodes and copying each node's data, then linking those new nodes in the same order as the source. This yields two separate lists: changes to one won't affect the other, and destroying one won't invalidate the other. If the source is empty, the new list is empty; otherwise, you clone every node as you traverse. A shallow copy would only copy pointers to the existing nodes, causing both lists to share the same nodes and leading to problems when either list is modified or destroyed. Deleting all nodes in the original is a cleanup task, not part of copying, and reserving memory for future nodes isn't the responsibility of the copy constructor.

5. What is the expected time complexity to search for an element in a skip list compared to a standard singly linked list?

A. Skip list search is O(n) on average; standard singly linked list search is O(n).

B. Skip list search is O(1); singly linked list is O(n).

C. Skip list search is O(log n) on average; standard singly linked list search is O(n).

D. Skip list search is O(log n) worst-case; singly linked list is O(1).

Skip lists get their speed by using several levels of forward pointers, so you can skip over many elements at once. As you search, you move forward on higher levels and drop down as needed, and the number of steps you take grows with the height of the structure. Because the height grows only about as log n, the average case for searching is O(log n). A standard singly linked list has no higher levels to skip, so you must check elements one by one from the start, giving O(n) time on average. So the best description is that a skip list search is O(log n) on average while a singly linked list search is O(n). (Note: the skip list can be O(n) in the worst case, but the average case is O(log n).)

6. How do you remove duplicates from an unsorted singly linked list without extra storage?

A. Use a hash set to track seen values.

B. Sort the list to bring duplicates together.

C. Use two nested loops to compare each node with the rest; O(n^2) time, O(1) space.

D. Slide a window pointer and delete duplicates.

To remove duplicates without extra storage on an unsorted singly linked list, the in-place approach compares each node with all nodes that follow it and removes any duplicates as it goes. Start with the first node as the "current." For each current node, scan the remainder of the list with a secondary pointer that trails behind a candidate duplicate. If you find a node with the same value as current, bypass it by adjusting the next pointer (and free the removed node, if you're managing memory). Move the trailing pointer forward when there's no match. Then advance current to the next node and repeat until you reach the end. This method uses only a fixed set of pointers, so the extra space is O(1). However, because you may traverse the remaining list for every node, the time complexity is O(n^2) in the worst case. Using a hash set would need extra storage, which isn't allowed here, and while sorting the list first could group duplicates together, the straightforward no-storage solution typically taught is the in-place, two-nested-loop approach with quadratic time.

7. 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.

8. What is the standard approach to inserting at the tail when no tail pointer exists?

A. Traverse to the last node, set last.next = new; new.next = null.

B. Insert at the head by using head pointer.

C. Set head.next = new; new.next = null.

D. Create a new list with the node as the only element.

When you only have a head pointer and no tail pointer, adding to the end means you must locate the current last node by walking from the head until you reach a node whose next is null. Once you find that last node, you attach the new node by setting last.next to the new node and then set the new node's next to null. This preserves the existing list and places the new element at the tail. Because you have to traverse from the head, this operation runs in linear time with respect to the list length. The other approaches don't correctly perform tail insertion in this scenario: inserting at the head moves the front of the list to the new node rather than extending the end; setting the next pointer of the head directly can break the list structure or misplace the new node, especially if the list has multiple elements or is empty; creating a new list with only the new node discards all existing nodes.

9. If you are given only a reference to a node (not the head) and you know it is not the tail, how can you delete that node?

A. Delete by removing the current node from the list using a head pointer.

B. Copy the data from the next node into the current node, set current.next = current.next.next; delete the next node; O(1)

C. Replace the current node with a new node carrying the same data.

D. Set current = current.next; deallocate the old node.

When you can only reference the node to delete and you know it isn't the tail, you don't have access to the previous node to relink the list. The trick is to make the current node take on the identity of its next node and then remove that next node. Specifically, you copy the data from the next node into the current node, then bypass the next node by linking current.next to current.next.next, and finally delete the next node. This achieves deletion in constant time without requiring a head pointer, because you're effectively removing the next node and leaving the current node with the next node's value. This approach works only if there is a next node to copy from; if the node were the tail, there is no next node to copy, and you would need access to the previous node or the head to perform deletion.

10. What does the clear() method do in the LList class?

A. It clears the list but preserves the allocated memory.

B. It removes all nodes and resets the list.

C. It only empties the first node.

D. It performs a shallow clear without deallocating nodes.

The method tested is about fully cleaning up and resetting the list. In a linked list, clear should remove every node by deallocating them, then reset the list's internal state so it behaves like a brand-new empty list. Practically, that means deleting all nodes, setting the head (and usually the tail) to null, and resetting the size counter to zero. This makes the list truly empty and ready for fresh inserts. It isn't about just removing the first node, nor about a shallow reset that leaves nodes allocated or memory leaks behind. The goal is to reclaim memory and leave no remaining elements or stale references.

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!