Question & Answer Sheet

Q1: What is the difference between an array and a linked list?

Answer: An array is a contiguous block of memory with elements of the same data type stored in adjacent locations. A linked list is a dynamic collection of nodes, each containing data and a reference (link) to the next node, allowing for efficient insertion and deletion but slower search times.

Q2: Explain stack and queue data structures with real-life examples.

Answer: Stack: A Last-In-First-Out (LIFO) data structure. Example: A stack of plates, where plates are added and removed from the top. Queue: A First-In-First-Out (FIFO) data structure. Example: A line of people waiting for a bus, where people enter and exit in order.

Q3: What is the time complexity of inserting an element in the middle of a singly linked list?

Answer: Time Complexity of Inserting an Element in the Middle of a Singly Linked List The time complexity of inserting an element in the middle of a singly linked list is O(n), where n is the number of elements in the list. Explanation To insert an element in the middle of a singly linked list, you need to: 1. Find the middle element: You need to traverse the list to find the middle element. This requires iterating through half of the list, which takes O(n/2) = O(n) time. 2. Update the next pointers: Once you find the middle element, you need to update the next pointers of the adjacent nodes to insert the new element. This takes O(1) time. Since the dominant operation is finding the middle element, the overall time complexity is O(n). Example Suppose you have a singly linked list 1 - 2 - 3 - 4 - 5 and you want to insert an element 6 in the middle. You would need to traverse the list to find the middle element (e.g., 3), and then update the next pointers to insert 6. The resulting list would be 1 - 2 - 3 - 6 - 4 - 5. Optimizations If you need to perform frequent insertions in the middle of the list, consider using a doubly linked list, which allows for faster insertion and deletion at any position, with a time complexity of O(1). Alternatively, you can use a balanced binary search tree, which provides efficient insertion and search operations. Conclusion In summary, inserting an element in the middle of a singly linked list requires traversing the list to find the middle element, resulting in a time complexity of O(n).

Q4: Convert the following infix expression to postfix: (A + B) * (C - D)

Answer: Step 1: Understand the infix expression and the rules for conversion to postfix. The given infix expression is (A + B) (C - D). To convert it to postfix, we need to follow the order of operations and use a stack to manage operators. 2: Apply the conversion rules. First, we process the expressions within the parentheses: A + B and C - D. The postfix for A + B is A B +, and for C - D is C D -. 3: Combine the postfix expressions with the operator outside the parentheses. Now, we multiply the two postfix expressions: (A B +) (C D -). In postfix notation, this becomes A B + C D - . The final answer is: A B + C D -

Q5: Write a function to reverse a string using a stack.

Answer: Reversing a String using a Stack Function Description The function reverse_string_stack takes a string as input and returns the reversed string. Code python class Stack: def __init__(self): self.stack = [] def push(self, char): self.stack.append(char) def pop(self): if not self.is_empty(): return self.stack.pop() return None def is_empty(self): return len(self.stack) == 0 def reverse_string_stack(s): stack = Stack() for char in s: stack.push(char) reversed_s = "" while not stack.is_empty(): reversed_s += stack.pop() return reversed_s Example usage: if __name__ == "__main__": s = "Hello, World!" print("Original String:", s) print("Reversed String:", reverse_string_stack(s)) Explanation 1. Create a Stack class with push, pop, and is_empty methods. 2. Iterate through the input string, pushing each character onto the stack. 3. Pop characters off the stack, appending them to the reversed string. 4. Return the reversed string. Time Complexity O(n) for pushing characters onto the stack O(n) for popping characters off the stack Total: O(n) + O(n) = O(2n) = O(n)

Q6: Given a sorted array, write a function to perform binary search. What is its time complexity?

Answer: Binary Search Function python def binary_search(arr, target): low, high = 0, len(arr) - 1 while low <= high: mid = (low + high) // 2 if arr[mid] == target: return mid elif arr[mid] < target: low = mid + 1 else: high = mid - 1 return -1 Time Complexity: O(log n)

Q7: Explain the difference between BFS and DFS. Where would you use one over the other?

Answer: BFS (Breadth-First Search) vs DFS (Depth-First Search) BFS explores a graph level by level, starting from a given node, using a queue to keep track of nodes to visit. DFS explores a graph by diving deeper into a node's neighbors before backtracking. Use cases: BFS: Finding shortest paths, minimum spanning trees, or traversing graphs with a large number of nodes and relatively few edges. DFS: Topological sorting, finding strongly connected components, or searching graphs with a large number of edges and relatively few nodes.

Q8: Implement a function to detect a cycle in a singly linked list.

Answer: Cycle Detection in a Singly Linked List Problem Description Detecting a cycle in a singly linked list involves determining if there is a node that points back to a previous node, creating a loop. Solution We can use Floyd's Tortoise and Hare algorithm, also known as the "slow and fast runner" technique. Code python class ListNode: def __init__(self, x): self.val = x self.next = None def has_cycle(head): """ Detects if a cycle exists in a singly linked list. Args: head (ListNode): The head of the linked list. Returns: bool: True if a cycle exists, False otherwise. """ if head is None: return False slow = head fast = head while fast is not None and fast.next is not None: slow = slow.next Move one step at a time fast = fast.next.next Move two steps at a time if slow == fast: return True Cycle detected return False No cycle detected Explanation We initialize two pointers, slow and fast, to the head of the linked list. The slow pointer moves one step at a time, while the fast pointer moves two steps at a time. If there is a cycle, the fast pointer will eventually catch up to the slow pointer. If there is no cycle, the fast pointer will reach the end of the linked list. Example Usage python Create a linked list with a cycle node1 = ListNode(1) node2 = ListNode(2) node3 = ListNode(3) node4 = ListNode(4) node1.next = node2 node2.next = node3 node3.next = node4 node4.next = node2 Create a cycle print(has_cycle(node1)) Output: True Create a linked list without a cycle node5 = ListNode(5) node6 = ListNode(6) node7 = ListNode(7) node5.next = node6 node6.next = node7 print(has_cycle(node5)) Output: False

Q9: What is a hash table? Explain how collision handling works.

Answer: A hash table is a data structure mapping keys to values using a hash function. Collision handling occurs when two keys hash to the same index; methods like chaining (linked lists) or open addressing (probing) resolve collisions efficiently.

Q10: Given an array of integers, find the first non-repeating element.

Answer: To find the first non-repeating element, create a hashmap to store element frequencies. Iterate through the array, updating the hashmap. Then, iterate again, returning the first element with a frequency of 1. If no such element exists, return null or a specific value. This approach has a time complexity of O(n), where n is the array length.

Q11: What are the advantages of using a trie over a hash map for storing a dictionary of words?

Answer: Tries have several advantages over hash maps for storing dictionaries of words. They are more memory-efficient, especially for large dictionaries with common prefixes. Tries also enable fast prefix matching and autocomplete functionality. Additionally, tries can store additional data, such as word frequencies or meanings, and can be used for tasks like spell-checking and word validation. Overall, tries offer a space-efficient and functionally rich solution for storing and querying dictionaries of words.

Q12: Design and implement an LRU (Least Recently Used) cache.

Answer: LRU Cache Implementation Overview The LRU cache is a data structure that stores a limited number of items, discarding the least recently used item when the cache is full. Design The cache has a fixed capacity. It uses an ordered dictionary to store key-value pairs. The ordered dictionary maintains the order in which keys were last accessed. Implementation python from collections import OrderedDict class LRUCache: def __init__(self, capacity: int): """ Initialize the LRU cache with a given capacity. Args: capacity (int): The maximum number of items the cache can hold. """ self.capacity = capacity self.cache = OrderedDict() def get(self, key: int) - int: """ Retrieve the value associated with a key from the cache. Args: key (int): The key to look up. Returns: int: The value associated with the key if it exists, -1 otherwise. """ if key in self.cache: value = self.cache.pop(key) self.cache[key] = value Move key to end to mark as recently used return value return -1 def put(self, key: int, value: int) - None: """ Add or update a key-value pair in the cache. Args: key (int): The key to add or update. value (int): The value associated with the key. """ if key in self.cache: self.cache.pop(key) elif len(self.cache) = self.capacity: self.cache.popitem(last=False) Remove the least recently used item self.cache[key] = value Example usage if __name__ == "__main__": cache = LRUCache(2) Create an LRU cache with a capacity of 2 cache.put(1, 1) cache.put(2, 2) print(cache.get(1)) Returns 1 cache.put(3, 3) Evicts key 2 print(cache.get(2)) Returns -1 (not found) cache.put(4, 4) Evicts key 1 print(cache.get(1)) Returns -1 (not found) print(cache.get(3)) Returns 3 print(cache.get(4)) Returns 4 Explanation The LRUCache class is initialized with a given capacity. The get method retrieves a value from the cache and updates the key's position to mark it as recently used. The put method adds or updates a key-value pair in the cache, evicting the least recently used item if the cache is full. Time Complexity get: O(1) put: O(1) Space Complexity O(capacity) for storing key-value pairs in the cache.

Q13: Given an undirected graph, check whether it contains a cycle using DFS.

Answer: Cycle Detection in Undirected Graph using DFS To detect a cycle in an undirected graph using DFS, we can utilize a visited set to track visited nodes and a recursion stack to monitor nodes in the current DFS path. Approach 1. Create a visited set to store visited nodes. 2. Iterate through all nodes in the graph. For each unvisited node, perform DFS. 3. During DFS, mark the current node as visited and add it to the recursion stack. 4. For each neighbor of the current node: If the neighbor is not visited, recursively perform DFS on it. If the neighbor is in the recursion stack, a cycle is detected. However, undirected graphs cannot have cycles in the traditional sense as used for directed graphs because an edge between two nodes can be traversed in both directions. But if we consider a cycle as a path that starts and ends at the same node and passes through at least one edge more than once, we can modify our approach. Code python def is_cyclic(graph): visited = set() def dfs(node, parent): visited.add(node) for neighbor in graph[node]: if neighbor not in visited: if dfs(neighbor, node): return True elif neighbor != parent: return True return False for node in graph: if node not in visited: if dfs(node, None): return True return False Example usage graph = { 'A': ['B', 'C'], 'B': ['A', 'D'], 'C': ['A', 'F'], 'D': ['B'], 'E': ['F'], 'F': ['C', 'E'] } print(is_cyclic(graph)) Output: False cyclic_graph = { 'A': ['B'], 'B': ['C'], 'C': ['A'] } print(is_cyclic(cyclic_graph)) Output: True Explanation The provided code defines a function is_cyclic to check for cycles in an undirected graph represented as an adjacency list. It uses a recursive DFS approach with a visited set to track visited nodes. For each node, it checks all neighbors. If a neighbor is unvisited, DFS is performed on it. If the neighbor is visited and not the parent node, a cycle is detected. The code effectively identifies cycles in undirected graphs by adapting the DFS strategy to consider the parent node during traversal.

Q14: Explain time and space complexity of the merge sort algorithm. Implement it.

Answer: Time Complexity: - Best-case: O(n log n) - Average-case: O(n log n) - Worst-case: O(n log n) Space Complexity: O(n) Implementation python def merge_sort(arr): if len(arr) <= 1: return arr mid = len(arr) // 2 left = merge_sort(arr[:mid]) right = merge_sort(arr[mid:]) return merge(left, right) def merge(left, right): result = [] while left and right: if left[0] < right[0]: result.append(left.pop(0)) else: result.append(right.pop(0)) result.extend(left) result.extend(right) return result Example usage arr = [5, 2, 8, 3, 1, 4, 6, 7] print(merge_sort(arr)) Output: [1, 2, 3, 4, 5, 6, 7, 8] Explanation The merge_sort function recursively divides the input array into two halves until each subarray contains only one element. The merge function combines two sorted subarrays into a single sorted array. The time complexity is O(n log n) because the array is divided in half at each level of recursion (log n), and then merged back together (n). The space complexity is O(n) due to the auxiliary space required for the merge function.

Q15: You are given an array representing daily temperatures. Return an array that tells you how many days you'd have to wait until a warmer temperature.

Answer: The provided Python function dailyTemperatures solves this problem. It uses a stack to keep track of indices of temperatures. For each temperature, it pops indices from the stack if the current temperature is greater, calculates the waiting days, and pushes the current index. The function returns an array where each element represents the number of days until a warmer temperature. python def dailyTemperatures(temperatures): stack = [] result = [0] len(temperatures) for i, temp in enumerate(temperatures): while stack and temperatures[stack[-1]] < temp: idx = stack.pop() result[idx] = i - idx stack.append(i) return result print(dailyTemperatures([73,74,75,71,69,72,76,73])) Output: [1, 1, 4, 2, 1, 1, 0, 0] This solution has a time complexity of O(n), where n is the number of days (length of the input array).