**RECAP QUIZ**

Best case time complexity of different data structures for different operations

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Data Structure** | **Access** | **Search** | **Insertion** | **Deletion** |
| Array | O(1) | O(1) | O(1) | O(1) |
| Stack | O(1) | O(1) | O(1) | O(1) |
| Queue | O(1) | O(1) | O(1) | O(1) |
| Singly Linked List | O(1) | O(1) | O(1) | O(1) |
| Doubly Linked List | O(1) | O(1) | O(1) | O(1) |
| Hash Table | O(1) | O(1) | O(1) | O(1) |
| Binary Search Tree | O(log n) | O(log n) | O(log n) | O(log n) |
| AVL Tree | O(log n) | O(log n) | O(log n) | O(log n) |
| B Tree | O(log n) | O(log n) | O(log n) | O(log n) |
| Red Black Tree | O(log n) | O(log n) | O(log n) | O(log n) |

Worst Case time complexity of different data structures for different operations

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Data Structure** | **Access** | **Search** | **Insertion** | **Deletion** |
| Array | O(1) | O(N) | O(N) | O(N) |
| Stack | O(N) | O(N) | O(1) | O(1) |
| Queue | O(N) | O(N) | O(1) | O(1) |
| Singly Linked List | O(N) | O(N) | O(N) | O(N) |
| Doubly Linked List | O(N) | O(N) | O(1) | O(1) |
| Hash Table | O(N) | O(N) | O(N) | O(N) |
| Binary Search Tree | O(N) | O(N) | O(N) | O(N) |
| AVL Tree | O(log N) | O(log N) | O(log N) | O(log N) |
| Binary Tree | O(N) | O(N) | O(N) | O(N) |
| Red Black Tree | O(log N) | O(log N) | O(log N) | O(log N) |

The average time complexity of different data structures for different operations

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Data Structure** | **Access** | **Search** | **Insertion** | **Deletion** |
| Array | O(1) | O(N) | O(N) | O(N) |
| Stack | O(N) | O(N) | O(1) | O(1) |
| Queue | O(N) | O(N) | O(1) | O(1) |
| Singly Linked List | O(N) | O(N) | O(1) | O(1) |
| Doubly Linked List | O(N) | O(N) | O(1) | O(1) |
| Hash Table | O(1) | O(1) | O(1) | O(1) |
| Binary Search Tree | O(log N) | O(log N) | O(log N) | O(log N) |
| AVL Tree | O(log N) | O(log N) | O(log N) | O(log N) |
| B Tree | O(log N) | O(log N) | O(log N) | O(log N) |
| Red Black Tree | O(log N) | O(log N) | O(log N) | O(log N) |

1. **Analysis of Recursive Programs**

1. **What is a recursive function?**
   * A function that calls itself.
2. **How does a base case help in recursion?**
   * It prevents infinite recursion by providing a stopping condition.
3. **Describe a scenario where recursion is preferred over iteration.**
   * Tree traversal (e.g., depth-first search).
4. **What is tail recursion?**
   * A recursion where the recursive call is the last operation.
5. **Why is recursion used in algorithms like depth-first search?**
   * It naturally mirrors the stack-based exploration of paths.
6. **Explain the difference between direct and indirect recursion.**
   * Direct: A function calls itself; Indirect: A function calls another that eventually calls the original.
7. **Write a recursive algorithm for finding the factorial of a number.**
   * factorial(n) = n \* factorial(n-1), with base case factorial(0) = 1.
8. **Analyze the time complexity of the recursive Fibonacci algorithm.**
   * Exponential, O(2^n).
9. **Compare the time and space complexities of recursion versus iteration for solving the same problem.**
   * Recursion may use more space (stack) but time complexity depends on the algorithm.
10. **What is memoization and how does it improve recursive programs?**

* Storing results of expensive function calls to avoid redundant computation.

1. **Describe the impact of stack size limits on recursive algorithms.**

* May cause stack overflow for deep recursion.

1. **Modify a recursive function into an iterative one without changing its logic.**

* Use an explicit stack or loop to simulate recursion.

1. **Discuss how recursion can lead to exponential time complexity in some algorithms.**

* Recomputing overlapping subproblems (e.g., Fibonacci).

**2 Solving Recurrence Equations**

1. **What is a recurrence relation?**
   * An equation that defines a function in terms of its smaller inputs.
2. **Give an example of a simple recurrence equation.**
   * T(n) = 2T(n/2) + n.
3. **How is the divide-and-conquer approach related to recurrence equations?**
   * It breaks problems into subproblems, modeled by recurrences.
4. **Define the base case for a recurrence relation.**
   * The smallest input where the recurrence stops, e.g., T(1).
5. **What is the master theorem used for?**
   * Solving recurrences in divide-and-conquer algorithms.
6. **Solve the recurrence: T(n) = T(n/2) + O(1).**
   * O(log n).
7. **Solve the recurrence: T(n) = 2T(n/2) + O(n).**
   * O(n log n).
8. **Analyze the recurrence relation for merge sort.**
   * T(n) = 2T(n/2) + O(n), solution: O(n log n).
9. **Use the substitution method to solve T(n) = 3T(n/4) + O(n).**
   * O(n log n).
10. **Apply the master theorem to solve a recurrence of your choice.**
    * Example: T(n) = 2T(n/2) + O(n), solution: O(n log n).
11. **Compare different methods for solving recurrences.**
    * Substitution: guess and verify; Recursion tree: visual breakdown; Master theorem: direct formula.
12. **Solve a recurrence relation with a logarithmic term, e.g., T(n) = 2T(n/2) + O(log n).**
    * O(n log n).
13. **Devise a recurrence relation for an algorithm and solve it.**
    * Example: Binary search, T(n) = T(n/2) + O(1), solution: O(log n).

3. **Implementation of Lists**

1. **What is a linked list?**
   * A data structure of nodes, each pointing to the next.
2. **Describe the difference between a singly linked list and a doubly linked list.**
   * Singly: points to next node; Doubly: points to both next and previous nodes.
3. **How do you insert an element at the beginning of a linked list?**
   * Update the head to the new node and point it to the old head.
4. **How do you traverse a linked list?**
   * Start from the head and follow pointers to the next node.
5. **What is the time complexity of searching for an element in a linked list?**
   * O(n).
6. **Implement a function to delete a node from a singly linked list.**
   * Update the previous node's pointer to skip the target node.
7. **Write a function to reverse a linked list.**
   * Iteratively update each node’s pointer to the previous node.
8. **Explain how circular linked lists differ from regular linked lists.**
   * Last node points back to the head instead of null.
9. **Compare the space and time efficiencies of arrays and linked lists for insertion operations.**
   * Arrays: O(n) time, O(1) space; Linked lists: O(1) time, O(n) space.
10. **Modify a linked list so that it has a randomized access feature like arrays.**
    * Use a hash table or array-like structure for indexing.
11. **Design an algorithm to detect a cycle in a linked list.**
    * Floyd's Cycle Detection (slow and fast pointers).
12. **Compare the memory usage of a doubly linked list with a singly linked list.**
    * Doubly uses more memory due to extra pointers.
13. **Implement a custom linked list class that supports efficient insertion and deletion.**
    * Create a class with node and pointer manipulation functions for O(1) insertions and deletions.

4. **Implementation of Stacks**

1. **What is a stack data structure?**
   * A linear data structure following LIFO (Last-In-First-Out).
2. **List two real-world examples of a stack.**
   * Undo feature in text editors, browser back button.
3. **What are the main operations of a stack?**
   * Push, Pop, Peek (or Top).
4. **Describe how a stack follows the Last-In-First-Out (LIFO) principle.**
   * The last element pushed is the first one popped.
5. **How is a stack implemented using arrays?**
   * Use an array with a pointer/index to track the top element.
6. **Write a program to check if a string has balanced parentheses using a stack.**
   * Push opening parentheses, pop when a matching closing one is found.
7. **Implement a stack using linked lists.**
   * Each node points to the next; the top node is pushed and popped.
8. **Explain how recursion internally uses a stack.**
   * Function calls are stored on the call stack, with the last called function returning first.
9. **Explain how a stack can be used to evaluate a postfix expression.**
   * Push operands onto the stack, pop and evaluate when an operator is encountered.
10. **Discuss how a stack is used in backtracking algorithms.**
    * Stores states or choices, and backtracks by popping the last choice.
11. **Compare the time and space complexities of a stack implemented using an array versus a linked list.**
    * Array: O(1) time, fixed space; Linked list: O(1) time, dynamic space.
12. **Design a stack with a function to get the minimum element in constant time.**
    * Use an auxiliary stack to track the minimum element.
13. **Create a modified stack class that supports both undo and redo operations.**
    * Use two stacks: one for undo and one for redo, managing state transitions.

**5. Implementation of Queues**

1. **What is a queue data structure?**
   * A linear data structure following FIFO (First-In-First-Out).
2. **How does a queue follow the First-In-First-Out (FIFO) principle?**
   * The first element added is the first one removed.
3. **What are the two main operations of a queue?**
   * Enqueue (insert), Dequeue (remove).
4. **Give two real-world examples of a queue.**
   * Waiting in line, print job queue.
5. **How is a queue implemented using arrays?**
   * Use an array with two pointers (front and rear) to manage insertion and removal.
6. **Implement a circular queue.**
   * Use a fixed-size array with wraparound behavior via modulo arithmetic.
7. **Write a program to implement a queue using two stacks.**
   * Use one stack for enqueuing and another for dequeuing.
8. **Describe the difference between a queue and a deque.**
   * Queue: insertion at rear, removal from front; Deque: insertion and removal from both ends.
9. **Discuss the use of queues in breadth-first search (BFS) algorithms.**
   * Store nodes to explore, processing each node's neighbors level by level.
10. **Implement a priority queue and explain how it differs from a regular queue.**
    * Each element has a priority; dequeue removes the highest priority element.
11. **Compare time complexities of different queue implementations (array, linked list, and circular queue).**
    * Array: O(1) enqueue, dequeue (fixed size); Linked list: O(1) for both; Circular queue: O(1) with fixed memory.
12. **Explain how queues can be used to simulate real-world processes like ticketing systems.**
    * Manage sequential service by processing requests in the order they arrive.
13. **Design an algorithm that utilizes a queue to reverse the order of elements in a linked list.**
    * Enqueue all elements, then dequeue back into the list.

6. **Implementation of Trees**

1. **What is a binary tree?**
   * A tree where each node has at most two children.
2. **Define a binary search tree (BST).**
   * A binary tree where the left child is smaller, and the right child is larger than the parent.
3. **How do you insert a node in a binary search tree?**
   * Recursively compare and place the node in the correct left or right subtree.
4. **What is an inorder traversal?**
   * Visit the left subtree, the node, and then the right subtree.
5. **How do you delete a node from a binary search tree?**
   * Replace with in-order successor/predecessor if the node has two children.
6. **Implement a function to find the height of a binary tree.**
   * Recursively find the maximum depth of the left and right subtrees.
7. **Write a program to check if a binary tree is balanced.**
   * Check if the height difference between left and right subtrees is at most 1.
8. **Explain the difference between preorder and postorder traversals.**
   * Preorder: visit node, left, right; Postorder: left, right, node.
9. **Implement an algorithm to convert a binary tree to a doubly linked list.**
   * Perform in-order traversal while adjusting pointers to form a doubly linked list.
10. **Compare AVL trees and red-black trees in terms of balancing operations.**
    * AVL: stricter balancing, more rotations; Red-black: less strict, fewer rotations.
11. **Design an efficient algorithm to find the lowest common ancestor of two nodes in a binary tree.**
    * Recursively find the split point where one node is in the left subtree and the other in the right.
12. **Discuss the trade-offs between depth-first and breadth-first tree traversals.**
    * Depth-first uses less memory but can get stuck; Breadth-first uses more memory but finds the shortest path.
13. **Implement a function that performs a level-order traversal of a binary tree using a queue.**
    * Use a queue to enqueue nodes level by level, starting from the root.

7. **Sorting Algorithms (Bubble, Selection, Insertion)**

1. **What is bubble sort?**
   * A sorting algorithm that repeatedly swaps adjacent elements if they are in the wrong order.
2. **How does selection sort work?**
   * Selects the smallest (or largest) element and swaps it with the first unsorted element.
3. **Describe the insertion sort algorithm.**
   * Builds a sorted array one element at a time by inserting elements into their correct position.
4. **What is the time complexity of bubble sort in the worst case?**
   * O(n²).
5. **Why is insertion sort more efficient on small datasets?**
   * Fewer comparisons and shifts for small or nearly sorted datasets.
6. **Implement the selection sort algorithm.**
   * Repeatedly find the minimum element from the unsorted part and swap it with the first unsorted element.
7. **Compare the performance of bubble sort and selection sort for sorted input.**
   * Bubble sort: O(n); Selection sort: O(n²), no performance improvement for sorted input.
8. **Write a program to sort an array using insertion sort.**
   * Traverse the array, shifting elements to insert each new element into its correct position.
9. **Compare the stability of bubble, selection, and insertion sort.**
   * Bubble and insertion are stable; selection is not stable.
10. **Modify the bubble sort algorithm to detect early termination for nearly sorted arrays.**
    * Add a flag to stop the algorithm if no swaps occur in a pass.
11. **Design an optimized version of insertion sort for partially sorted arrays.**
    * Use binary search to find the insertion point, reducing comparisons.
12. **Discuss the space complexity of bubble, selection, and insertion sort and suggest improvements.**
    * All have O(1) space complexity; improvements come from hybrid algorithms.
13. **Implement a hybrid sorting algorithm combining insertion sort with another efficient sort.**
    * Use insertion sort for small subarrays and merge/quick sort for larger ones.

8. **Quick Sort**

1. **What is the basic principle behind quicksort?**
   * Divide-and-conquer using a pivot to partition the array.
2. **Describe the partitioning step in quicksort.**
   * Reorder the array so that elements smaller than the pivot come before it, and larger elements after.
3. **What is the best-case time complexity of quicksort?**
   * O(n log n).
4. **How does the choice of pivot affect quicksort performance?**
   * A poor pivot choice (e.g., smallest/largest element) leads to unbalanced partitions and worse performance.
5. **What is the average-case time complexity of quicksort?**
   * O(n log n).
6. **Implement the quicksort algorithm.**
   * Recursively partition the array around a pivot and sort the subarrays.
7. **Explain the worst-case scenario for quicksort and how to avoid it.**
   * Worst-case: O(n²) occurs when the pivot consistently produces unbalanced partitions; avoid with randomized or median-of-three pivot selection.
8. **Write a program to sort an array using quicksort with randomized pivot selection.**
   * Select a random element as the pivot before partitioning.
9. **Discuss how quicksort can be optimized for small datasets.**
   * Switch to insertion sort for small subarrays (e.g., size ≤ 10).
10. **Compare the performance of quicksort and mergesort for large datasets.**
    * Quicksort: faster on average, O(n log n) in-place; Mergesort: O(n log n), stable but requires extra space.
11. **Design an in-place quicksort algorithm.**
    * Use the same array for partitioning without additional memory allocation.
12. **Explain the use of tail call elimination to reduce space complexity in quicksort.**
    * Recur on the smaller partition first and convert the larger recursive call into an iterative loop.
13. **Analyze the performance of quicksort for different types of input data (random, sorted, reverse-sorted).**
    * Random: O(n log n); Sorted/Reverse-sorted: O(n²) if no pivot optimization (e.g., randomized or median-of-three).

9. **Merge Sort**

1. **What is merge sort?**
   * A divide-and-conquer sorting algorithm that splits the array and merges sorted subarrays.
2. **Describe the process of merging in merge sort.**
   * Compare the smallest elements of the subarrays and combine them into a sorted array.
3. **What is the time complexity of merge sort in all cases?**
   * O(n log n) in best, average, and worst cases.
4. **Why is merge sort a stable sorting algorithm?**
   * It maintains the relative order of equal elements during merging.
5. **How does merge sort differ from quicksort?**
   * Merge sort is stable and always O(n log n); quicksort is generally faster but can be O(n²) in worst cases.
6. **Implement the merge sort algorithm.**
   * Recursively divide the array and merge sorted halves.
7. **Write a program to count inversions in an array using merge sort.**
   * Count inversions during the merge step by comparing elements from left and right subarrays.
8. **Compare the space complexity of merge sort and quicksort.**
   * Merge sort: O(n) due to auxiliary space; Quicksort: O(log n) for recursive stack space.
9. **Modify merge sort to sort linked lists instead of arrays.**
   * Use pointers to split the linked list and merge sorted nodes without additional space.
10. **Design a parallel version of merge sort.**
    * Divide the array into segments, sort each in parallel, and then merge results.
11. **Explain how merge sort is used in external sorting for large datasets.**
    * Efficiently merges sorted chunks of data from external storage to minimize disk I/O.
12. **Analyze the time complexity of merge sort when applied to datasets stored on disk.**
    * Generally O(n log n), but I/O operations can affect performance depending on the block size.
13. **Compare merge sort and heap sort for sorting large datasets.**
    * Merge sort: stable, good for external sorting; Heap sort: in-place, not stable, generally slower due to heapification.

10. **Heap Sort**

1. **What is a heap?**
   * A complete binary tree that satisfies the heap property (max-heap: parent ≥ children; min-heap: parent ≤ children).
2. **How does heap sort work?**
   * Build a max heap from the array, then repeatedly swap the root with the last element and heapify the reduced heap.
3. **What is the time complexity of heap sort?**
   * O(n log n) for all cases (best, average, worst).
4. **Describe the process of heapifying an array.**
   * Rearrange elements to satisfy the heap property starting from the last non-leaf node up to the root.
5. **How is heap sort different from quicksort?**
   * Heap sort is not recursive and always O(n log n); quicksort is typically faster but can be O(n²) in worst cases.
6. **Implement the heap sort algorithm.**
   * Build a max heap, then repeatedly extract the maximum and heapify.
7. **Write a function to convert an unsorted array into a max heap.**
   * Apply the heapify process from the last non-leaf node to the root.
8. **Compare the performance of heap sort and merge sort.**
   * Heap sort: O(n log n) in-place, not stable; Merge sort: O(n log n), stable but requires extra space.
9. **Design an algorithm to maintain a dynamically changing heap (insertions and deletions).**
   * Use insert (bubble up) for insertions and delete (bubble down) for deletions.
10. **Explain how heaps are used in priority queues and heap sort.**
    * Priority queues use heaps to efficiently manage elements with varying priorities; Heap sort leverages heaps to sort elements.
11. **Analyze the worst-case and average-case performance of heap sort.**
    * Both are O(n log n) due to the need to maintain the heap property during sorting.
12. **Compare the space complexity of heap sort with other sorting algorithms.**
    * Heap sort: O(1) (in-place); Merge sort: O(n); Quick sort: O(log n).
13. **Implement a hybrid sorting algorithm combining heap sort and insertion sort for optimized performance.**
    * Use heap sort for large datasets and switch to insertion sort for small subarrays (e.g., size ≤ 10).