Karnataka PGCET MCA 2025 Question Paper with Solutions

Time Allowed :2 Hours 30 Minutes	Maximum Marks :100	Total Questions :75

General Instructions

Read the following instructions very carefully and strictly follow them:

1. The duration of the test is 150 minutes.
2. MCA PGCET Paper is divided into 2 parts- Part-A and Part-B
3. Each question will be asked in a multiple-choice (Objective) type format, wherein each question will have four different options.
4. In Part A, there will be 50 questions that will carry one mark each.
5. In Part B, there will be 25 questions that will carry two marks each. Hence, there will be 75 questions and the total mark is 100

1. Which of the following data structures is most suitable for implementing a priority queue?

(A) Array
(B) Stack
(C) Binary Heap
(D) Linked List

Correct Answer: (C) Binary Heap

Solution:

To determine the most suitable data structure for implementing a priority queue, we need to analyze how each data structure performs the necessary operations: insertion, deletion, and access to the highest (or lowest) priority element. A priority queue requires efficient access to the element with the highest (or lowest) priority, and its operations should be optimized accordingly.

Step 1: Review the Properties of a Priority Queue
A priority queue is an abstract data structure where each element has a priority level. The main operations we need to support are:
- **Insertion** of elements with a priority.
- **Accessing and removing** the element with the highest (or lowest) priority, which should be done efficiently.

Let's now examine the suitability of each data structure for these operations:

Option A: Array
While an array can be used to implement a priority queue, it is not the most efficient. Here's why:
- **Insertion** in an unsorted array can be done in constant time ($O(1)$), but finding the element with the highest or lowest priority takes $O(n)$ time, as we would need to search through the entire array.
- If the array is **sorted**, insertion becomes more costly, requiring $O(n)$ time for insertion to maintain the order. Removing the highest priority element, in this case, can be done in $O(1)$, but the overall time complexity for operations like insertion and deletion is less efficient than other alternatives.
Thus, although arrays can be used for priority queues, they do not offer the best performance in terms of time complexity.

Option B: Stack
A stack operates on the **Last In, First Out (LIFO)** principle, where the last inserted element is the first to be removed. This is fundamentally incompatible with the behavior of a priority queue:
- A stack does not maintain any ordering based on the priority of the elements. Therefore, it is impossible to ensure that the highest or lowest priority element is accessed first.
- The stack is not designed for accessing or removing elements based on priority, so it cannot be used to efficiently implement a priority queue.
For these reasons, a stack is not suitable for implementing a priority queue.

Option C: Binary Heap
A **binary heap** is a specialized binary tree that satisfies the heap property, making it the most suitable data structure for a priority queue. Here's why:
- A binary heap can be implemented as an array where each parent node has a higher priority than its children (max-heap) or a lower priority than its children (min-heap).
- **Insertion** takes $O(\log n)$ time because we need to maintain the heap property by "bubbling up" the newly inserted element.
- **Accessing and removing** the element with the highest (or lowest) priority also takes $O(\log n)$ time because we only need to swap the root with the last element and then "bubble down" to maintain the heap property.
- The binary heap structure ensures that the highest (or lowest) priority element is always at the root, making it ideal for priority queue operations.

Thus, a binary heap allows both insertion and removal of elements with a time complexity of $O(\log n)$, which is efficient for implementing a priority queue. Therefore, option C is the correct answer.

Option D: Linked List
A **linked list** can be used to implement a priority queue, but it is less efficient compared to a binary heap:
- If the linked list is **unsorted**, accessing and removing the element with the highest or lowest priority requires searching through the entire list, resulting in $O(n)$ time complexity.
- If the linked list is **sorted**, insertion requires finding the correct position to maintain the order, which takes $O(n)$ time. Removal of the highest or lowest priority element can be done

in constant time ($O(1)$), but the overall insertion time complexity makes the sorted linked list less efficient than a binary heap.

Therefore, while a linked list can be used for a priority queue, it is not as efficient as a binary heap, which has better time complexity for the critical operations.

Step 2: Why the Other Options Are Less Efficient
- **Option A (Array)**: An unsorted array requires linear time to access the highest or lowest priority element, and a sorted array has inefficient insertions. Both cases are not optimal for a priority queue.
- **Option B (Stack)**: A stack is inherently unsuitable for a priority queue because it does not provide any priority-based ordering.
- **Option D (Linked List)**: While a linked list can be used, it has a higher time complexity for insertion and access compared to a binary heap. Thus, a linked list is not as efficient as a binary heap for implementing a priority queue.

Step 3: Summary of Operations
- **Binary Heap**: Best choice. Efficient with $O(\log n)$ time complexity for insertion, deletion, and access to the highest or lowest priority element.
- **Array**: Suitable but inefficient due to $O(n)$ time for access and deletion in unsorted arrays.
- **Stack**: Not suitable as it doesn't maintain any ordering based on priority.
- **Linked List**: Less efficient than a binary heap due to $O(n)$ time for insertion and access in an unsorted list.

Quick Tip

When implementing a priority queue, consider the time complexity of key operations (insertion, deletion, and access). A binary heap offers the most efficient solution with $O(\log n)$ complexity for these operations, making it the preferred choice.

---

2. What is the time complexity of accessing an element in a hash table (assuming a good hash function)?
(A) O(1)
(B) O(log n)
(C) O(n)
(D) O(n^2)

Correct Answer: (A) O(1)

Solution:

To determine the time complexity of accessing an element in a hash table with a good hash function, we need to understand how hash tables work and the role of the hash function.

Step 1: Understand Hash Tables
A hash table is a data structure that stores key-value pairs and uses a hash function to map keys to specific indices in an array.
Accessing an element involves computing the hash value of the key and using it to locate the element directly.

Step 2: Role of a Good Hash Function
A good hash function minimizes collisions (where two keys hash to the same index) and distributes keys uniformly across the array.
Assuming a good hash function and no collisions, the element can be accessed directly using the computed index.

Step 3: Analyze Option A - O(1)
O(1) represents constant time complexity, meaning the access time does not depend on the number of elements (n).
With a good hash function and an efficient implementation (e.g., using open addressing or chaining with a small number of collisions), accessing an element is a constant-time operation.
Thus, option A is correct.

Step 4: Analyze Option B - O(log n)
O(log n) is typical for balanced binary search trees, where the height of the tree determines the number of comparisons.
Hash tables do not rely on tree structures for access with a good hash function, so this does not apply.
Thus, option B is incorrect.

Step 5: Analyze Option C - O(n)
O(n) occurs in linear search or when there are many collisions in a hash table, requiring a search through a linked list or array.
With a good hash function, collisions are minimized, and access remains constant, not linear.
Thus, option C is incorrect.

Step 6: Analyze Option D - O(n^2)
O(n^2) is associated with nested loops or inefficient algorithms like bubble sort.
This complexity is irrelevant to hash table access, even with poor performance.
Thus, option D is incorrect.

Step 7: Key Consideration
The assumption of a "good hash function" is critical. If the hash function is poor and causes many collisions, the worst-case time complexity could degrade to O(n).
However, the question specifies a good hash function, ensuring O(1) average-case performance.

Quick Tip

To understand hash table complexity:
- A good hash function ensures O(1) average time for access and insertion.
- Minimize collisions to maintain efficiency.
- Compare with other structures: Arrays (O(1) with index), Trees (O(log n)), Linear Search (O(n)).

---

3. Which of the following is a correct statement regarding object-oriented programming (OOP)?
(A) OOP is based on functions rather than data
(B) In OOP, objects can have both data and methods
(C) In OOP, the program is written as a sequence of instructions
(D) OOP does not support inheritance

Correct Answer: (B) In OOP, objects can have both data and methods

Solution:

To determine the correct statement about object-oriented programming (OOP), we need to understand its core principles.

Step 1: Understand OOP
OOP is a programming paradigm that uses objects, which are instances of classes.
It emphasizes data (attributes) and the methods (functions) that operate on that data, encapsulated together.
Key principles include encapsulation, inheritance, polymorphism, and abstraction.

Step 2: Analyze Option A - OOP is based on functions rather than data
OOP is centered around objects that combine data and methods, not just functions.
This is more characteristic of procedural programming, not OOP.
Thus, option A is incorrect.

Step 3: Analyze Option B - In OOP, objects can have both data and methods
In OOP, objects are instances of classes that contain both data (e.g., variables or attributes) and methods (e.g., functions or behaviors).
This reflects the principle of encapsulation, a core feature of OOP.
Thus, option B is correct.

Step 4: Analyze Option C - In OOP, the program is written as a sequence of instructions
Writing a program as a sequence of instructions is typical of procedural programming, not OOP.
OOP focuses on objects and their interactions rather than a linear sequence.

Thus, option C is incorrect.

Step 5: Analyze Option D - OOP does not support inheritance
Inheritance is a fundamental OOP principle, allowing a class to inherit properties and methods from another class.
OOP does support inheritance, making this statement false.
Thus, option D is incorrect.

Step 6: Conclusion
The correct statement aligns with OOP's definition, where objects encapsulate both data and methods.

Quick Tip

To understand OOP:
- Focus on objects combining data and methods.
- Learn key principles: Encapsulation, Inheritance, Polymorphism, Abstraction.
- Compare with procedural programming, which uses sequences of instructions.

---

4. Which of the following sorting algorithms has the worst-case time complexity of $O(n^2)$?
(A) Merge Sort
(B) Quick Sort
(C) Bubble Sort
(D) Heap Sort

Correct Answer: (C) Bubble Sort

Solution:

To determine which sorting algorithm has a worst-case time complexity of $O(n^2)$, we need to evaluate the time complexity of each algorithm in its worst-case scenario.

Step 1: Understand Time Complexity
The worst-case time complexity indicates the maximum time an algorithm takes for any input of size n.
$O(n^24)$ means the time grows quadratically with the input size, typically seen in algorithms with nested loops.

Step 2: Analyze Option A - Merge Sort
Merge Sort uses a divide-and-conquer approach, splitting the array into halves and merging them.
Its worst-case time complexity is O(n log n) due to the merging process.

Thus, option A is incorrect.

Step 3: Analyze Option B - Quick Sort
Quick Sort also uses a divide-and-conquer strategy, partitioning the array around a pivot.
Its average-case time complexity is O(n log n), but the worst case (e.g., already sorted array with a bad pivot) is $O(n^2)$.
However, the question asks for the algorithm where $O(n^2)$ is the standard worst-case, not an exceptional case.
Thus, option B is not the best answer here.

Step 4: Analyze Option C - Bubble Sort
Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order.
In the worst case (e.g., reverse-sorted array), it requires n-1 passes with up to n-1 comparisons each, leading to $O(n^2)$ time complexity.
This is the standard worst-case for Bubble Sort.
Thus, option C is correct.

Step 5: Analyze Option D - Heap Sort
Heap Sort builds a max-heap and repeatedly extracts the maximum element, with a time complexity of O(n log n) in all cases (best, average, and worst).
Thus, option D is incorrect.

Step 6: Why the Other Options Are Incorrect
- **Option A (Merge Sort)**: Always O(n log n), better than $O(n^2)$.
- **Option B (Quick Sort)**: Worst case is $O(n^2)$ but not its defining characteristic; it's typically optimized to avoid this.
- **Option D (Heap Sort)**: Consistently O(n log n), never $O(n^2)$.
Bubble Sort's $O(n^2)$ is its standard worst-case behavior.

Step 7: Key Insight
While Quick Sort can reach $O(n^2)$ in rare cases, Bubble Sort is inherently $O(n^2)$ in its worst case due to its simple, unoptimized comparison approach.

Quick Tip

To understand sorting complexities:
- Learn worst-case scenarios: Bubble Sort ($O(n^2)$), Merge/Heap Sort (O(n log n)).
- Quick Sort's $O(n^2)$ is avoidable with good pivot selection.
- Practice with small arrays to observe behavior.

---

5. Which of the following is true about the SQL JOIN operation?
(A) It combines rows from two or more tables based on a related column

(B) It only works on columns with unique values
(C) It is used for updating data in a table
(D) It can only be used to select data from one table at a time

Correct Answer: (A) It combines rows from two or more tables based on a related column

Solution:

To determine the correct statement about the SQL JOIN operation, we need to understand its purpose and functionality in database management.

Step 1: Understand the SQL JOIN Operation
The JOIN operation in SQL is used to combine rows from two or more tables based on a related column, typically a foreign key or a common field.
It allows retrieval of data from multiple tables in a single query, based on a condition.

Step 2: Analyze Option A - It combines rows from two or more tables based on a related column
This accurately describes the primary function of a JOIN operation, such as INNER JOIN, LEFT JOIN, RIGHT JOIN, or FULL JOIN, which link tables using a related column (e.g., a primary key in one table and a foreign key in another).
Thus, option A is correct.

Step 3: Analyze Option B - It only works on columns with unique values
A JOIN operation does not require columns to have unique values; it works with related columns regardless of uniqueness (e.g., non-unique foreign keys are common).
Uniqueness is relevant for primary keys, but not a requirement for JOINs.
Thus, option B is incorrect.

Step 4: Analyze Option C - It is used for updating data in a table
The UPDATE statement, not JOIN, is used to modify data in a table.
JOIN is used for querying and combining data, not for updates (though it can be part of an UPDATE query with a JOIN clause).
Thus, option C is incorrect.

Step 5: Analyze Option D - It can only be used to select data from one table at a time
JOIN is specifically designed to select data from multiple tables by combining rows based on related columns.
Limiting it to one table contradicts its purpose.
Thus, option D is incorrect.

Step 6: Conclusion
The core function of a SQL JOIN is to combine data from multiple tables, making option A the correct choice.

Quick Tip

To understand SQL JOIN:
- Use JOIN to combine tables based on related columns (e.g., ID fields).
- Learn types: INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN.
- Practice with simple queries to see row combinations.

---

6. Which of the following is NOT a characteristic of the C programming language?
(A) It supports low-level memory manipulation
(B) It is a high-level, structured language
(C) It allows direct access to hardware
(D) It does not support recursion

Correct Answer: (D) It does not support recursion

Solution:

To determine which statement is NOT a characteristic of the C programming language, we need to evaluate each option against the known features of C.

Step 1: Understand C Programming Language
C is a general-purpose programming language known for its flexibility, efficiency, and control over system resources.
It supports low-level operations, structured programming, and hardware interaction, and also includes support for recursion.

Step 2: Analyze Option A - It supports low-level memory manipulation
C provides pointers and manual memory management (e.g., malloc, free), which is a key feature for low-level memory control.
Thus, option A is correct.

Step 3: Analyze Option B - It is a high-level, structured language
C is considered a high-level language with structured programming features (e.g., functions, loops), though it retains low-level capabilities.
Thus, option B is correct.

Step 4: Analyze Option C - It allows direct access to hardware
C allows direct hardware manipulation through pointers and memory addresses, making it suitable for system programming.
Thus, option C is correct.

Step 5: Analyze Option D - It does not support recursion
C fully supports recursion, as functions can call themselves, and the language includes a call stack to manage recursive calls.

The statement "It does not support recursion" is false.
Thus, option D is incorrect and the correct answer.

Step 6: Conclusion
The only statement that does not describe C is that it lacks recursion support, making option D the answer.

Quick Tip

To understand C characteristics:
- Learn about pointers for memory manipulation.
- Recognize structured programming with functions.
- Note hardware access via memory addresses.
- Test recursion with simple factorial functions.

---

7. In the context of network protocols, what does the acronym "TCP" stand for?
(A) Transfer Control Protocol
(B) Transport Control Protocol
(C) Transmission Control Protocol
(D) Time Control Protocol

Correct Answer: (C) Transmission Control Protocol

Solution:

To determine the meaning of the acronym "TCP" in the context of network protocols, we need to identify the standard definition used in networking.

Step 1: Understand Network Protocols
TCP is a fundamental protocol in the Internet Protocol Suite, working with IP to ensure reliable data transmission.
It is part of the transport layer in the OSI model.

Step 2: Analyze Option A - Transfer Control Protocol
"Transfer Control Protocol" is not a recognized standard acronym in networking.
Thus, option A is incorrect.

Step 3: Analyze Option B - Transport Control Protocol
While "Transport Control Protocol" is sometimes used informally, the correct and widely accepted term is "Transmission Control Protocol."
Thus, option B is incorrect.

Step 4: Analyze Option C - Transmission Control Protocol
"TCP" stands for "Transmission Control Protocol," which ensures reliable, ordered, and error-checked delivery of data between applications.
This is the standard definition in networking.
Thus, option C is correct.

Step 5: Analyze Option D - Time Control Protocol
"Time Control Protocol" is not a known protocol in networking contexts.
Thus, option D is incorrect.

Step 6: Conclusion
The correct expansion of "TCP" in network protocols is "Transmission Control Protocol," making option C the answer.

Quick Tip

To understand network acronyms:
- Remember TCP/IP as the core Internet protocol suite.
- Learn TCP's role in reliable data transmission.
- Distinguish from UDP (User Datagram Protocol) for comparison.

---

8. Which of the following is the correct sequence of memory hierarchy from fastest to slowest?
(A) Registers > Cache > RAM > Disk
(B) Disk > RAM > Cache > Registers
(C) RAM > Registers > Disk > Cache
(D) Registers > Disk > RAM > Cache

Correct Answer: (A) Registers > Cache > RAM > Disk

Solution:

To determine the correct sequence of the memory hierarchy from fastest to slowest, we need to understand the relative speeds of different memory types in a computer system.

Step 1: Understand the Memory Hierarchy
The memory hierarchy is designed to balance speed and cost, with faster memory being smaller and more expensive, and slower memory being larger and cheaper.
The typical order from fastest to slowest is: Registers, Cache, RAM (Random Access Memory), and Disk.

Step 2: Analyze Option A - Registers > Cache > RAM > Disk
- Registers are the fastest memory, located inside the CPU for immediate data access.

- Cache is faster than RAM, serving as a buffer between the CPU and main memory.
- RAM is slower than Cache but faster than Disk, used for primary storage.
- Disk (e.g., HDD or SSD) is the slowest, used for long-term storage.
This sequence matches the standard memory hierarchy from fastest to slowest.
Thus, option A is correct.

Step 3: Analyze Option B - Disk > RAM > Cache > Registers
This reverses the hierarchy, placing the slowest memory (Disk) first and the fastest (Registers) last.
This is incorrect as it contradicts the known speed order.
Thus, option B is incorrect.

Step 4: Analyze Option C - RAM > Registers > Disk > Cache
This sequence is incorrect because Registers are faster than RAM, and Cache is faster than Disk.
The order does not align with the memory hierarchy.
Thus, option C is incorrect.

Step 5: Analyze Option D - Registers > Disk > RAM > Cache
This sequence places Disk before RAM and Cache, which is wrong since Disk is slower than both RAM and Cache.
Thus, option D is incorrect.

Step 6: Conclusion
The correct order, based on speed from fastest to slowest, is Registers > Cache > RAM > Disk, making option A the answer.

Quick Tip

To understand memory hierarchy:
- Remember: Registers (fastest), Cache, RAM, Disk (slowest).
- Speed decreases as capacity and cost per unit increase.
- Study CPU design to see how these levels interact.

---

9. Which of the following is a feature of the "Stack" data structure?
(A) Follows a First-In-First-Out (FIFO) order
(B) Allows elements to be inserted and removed at both ends
(C) Follows a Last-In-First-Out (LIFO) order
(D) Does not allow random access to elements

Correct Answer: (C) Follows a Last-In-First-Out (LIFO) order

Solution:

To determine the correct feature of the Stack data structure, we need to understand its defining characteristics.

Step 1: Understand the Stack Data Structure
A Stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle, meaning the last element added is the first to be removed.
Operations are typically limited to push (insert) and pop (remove) at one end, known as the top.

Step 2: Analyze Option A - Follows a First-In-First-Out (FIFO) order
FIFO is characteristic of a Queue, not a Stack, where the first element added is the first to be removed.
Thus, option A is incorrect.

Step 3: Analyze Option B - Allows elements to be inserted and removed at both ends
This describes a Deque (double-ended queue), not a Stack, which only allows operations at the top.
Thus, option B is incorrect.

Step 4: Analyze Option C - Follows a Last-In-First-Out (LIFO) order
LIFO is the defining feature of a Stack, where the most recently added element is removed first (e.g., like a stack of plates).
This aligns with the Stack's behavior.
Thus, option C is correct.

Step 5: Analyze Option D - Does not allow random access to elements
While true that Stacks do not support random access (elements can only be accessed via pop from the top), this is a limitation rather than a primary feature.
The question asks for a defining feature, making LIFO more appropriate.
Thus, option D is not the best answer but is a secondary characteristic.

Step 6: Conclusion
The primary feature of a Stack is its LIFO order, making option C the correct answer.

Quick Tip

To understand Stack:
- Remember LIFO: Last In, First Out.
- Compare with Queue (FIFO) and Deque (both ends).
- Practice with push/pop operations.

---

10. In object-oriented programming, which of the following is true about "Poly-

morphism"?
(A) It allows objects of different classes to be treated as objects of a common superclass
(B) It refers to the ability of a class to inherit properties from more than one class
(C) It refers to the ability to define a function with the same name but different signatures
(D) It allows data to be encapsulated in the class

Correct Answer: (A) It allows objects of different classes to be treated as objects of a common superclass

Solution:

To determine the correct definition of polymorphism in object-oriented programming, we need to evaluate each option against the concept's meaning.

Step 1: Understand Polymorphism
Polymorphism allows objects of different classes to be treated as objects of a common superclass, enabling a single interface to represent different underlying forms (e.g., method overriding or interfaces).

Step 2: Analyze Option A - It allows objects of different classes to be treated as objects of a common superclass
This is the core definition of polymorphism, where objects of derived classes can be handled as instances of a base class, often through method overriding.
Thus, option A is correct.

Step 3: Analyze Option B - It refers to the ability of a class to inherit properties from more than one class
This describes multiple inheritance, not polymorphism.
While related to OOP, it is a distinct concept.
Thus, option B is incorrect.

Step 4: Analyze Option C - It refers to the ability to define a function with the same name but different signatures
This describes function overloading, a form of compile-time polymorphism, but not the broader definition of runtime polymorphism (e.g., method overriding).
Thus, option C is partially correct but not the best answer.

Step 5: Analyze Option D - It allows data to be encapsulated in the class
This refers to encapsulation, another OOP principle, not polymorphism.
Thus, option D is incorrect.

Step 6: Conclusion
The most accurate and comprehensive definition of polymorphism is option A, focusing on the treatment of objects under a common superclass.

Quick Tip

To understand polymorphism:
- Focus on treating different classes as a common type.
- Practice with method overriding in inheritance.
- Distinguish from overloading (same name, different parameters).

---

11. Which of the following algorithms is based on the divide and conquer technique?
(A) Selection Sort
(B) Quick Sort
(C) Insertion Sort
(D) Bubble Sort

Correct Answer: (B) Quick Sort

Solution:

To determine which algorithm uses the divide and conquer technique, we need to understand the approach and evaluate each option.

Step 1: Understand Divide and Conquer
The divide and conquer technique involves dividing a problem into smaller subproblems, solving them independently, and combining the results.

Step 2: Analyze Option A - Selection Sort
Selection Sort repeatedly finds the minimum element and places it at the beginning, using a linear search.
It does not divide the problem into subproblems.
Thus, option A is incorrect.

Step 3: Analyze Option B - Quick Sort
Quick Sort selects a pivot, partitions the array around it, and recursively sorts the subarrays.
This clearly follows the divide and conquer strategy.
Thus, option B is correct.

Step 4: Analyze Option C - Insertion Sort
Insertion Sort builds the sorted array one element at a time by inserting each element into its correct position.
It does not use division into subproblems.
Thus, option C is incorrect.

Step 5: Analyze Option D - Bubble Sort
Bubble Sort compares adjacent elements and swaps them, repeating until sorted.
It does not involve dividing the problem.

Thus, option D is incorrect.

Step 6: Conclusion
Quick Sort is based on the divide and conquer technique, making option B the answer.

Quick Tip

To understand divide and conquer:
- Look for recursive division (e.g., Quick Sort, Merge Sort).
- Contrast with iterative methods (e.g., Bubble Sort).
- Practice tracing Quick Sort steps.

---

12. Which of the following sorting algorithms is stable?
(A) Quick Sort
(B) Merge Sort
(C) Heap Sort
(D) Selection Sort

Correct Answer: (B) Merge Sort

Solution:

To determine which sorting algorithm is stable, we need to understand the concept of stability in sorting and evaluate each option.

Step 1: Understand Stability in Sorting
A sorting algorithm is stable if it preserves the relative order of equal elements in the sorted output, i.e., if two elements have equal keys, their original order is maintained.

Step 2: Analyze Option A - Quick Sort
Quick Sort uses a pivot and partitioning, which can swap equal elements, potentially changing their relative order.
It is generally not stable unless modified (e.g., with extra space or indexing).
Thus, option A is incorrect.

Step 3: Analyze Option B - Merge Sort
Merge Sort divides the array into halves, sorts them, and merges them while preserving the order of equal elements during the merge process.
When implemented correctly (e.g., stable merge), it maintains the relative order of equal elements.
Thus, option B is correct.

Step 4: Analyze Option C - Heap Sort
Heap Sort builds a heap and repeatedly extracts the maximum element, which can reorder equal elements due to the heap structure.
It is not stable by default.
Thus, option C is incorrect.

Step 5: Analyze Option D - Selection Sort
Selection Sort finds the minimum element and places it at the beginning, which may swap equal elements and change their original order.
It is not stable.
Thus, option D is incorrect.

Step 6: Conclusion
Among the given options, only Merge Sort is inherently stable when implemented with a stable merge procedure, making option B the answer.

Quick Tip

To understand stability:
- A stable sort keeps equal elements in original order.
- Merge Sort is stable; Quick Sort and Heap Sort are not.
- Test with arrays having duplicate keys to observe behavior.

---

13. What is the worst-case time complexity of the Binary Search algorithm?
(A) O(1)
(B) O(n)
(C) O(log n)
(D) O(n^2)

Correct Answer: (C) O(log n)

Solution:

To determine the worst-case time complexity of the Binary Search algorithm, we need to understand how it operates and analyze its performance.

Step 1: Understand Binary Search
Binary Search is an efficient algorithm for finding an element in a sorted array by repeatedly dividing the search interval in half.
It requires the array to be sorted and compares the target value with the middle element, adjusting the search range accordingly.

Step 2: Analyze Option A - O(1)
O(1) represents constant time, which applies to direct access (e.g., array indexing).
Binary Search does not guarantee constant time as it depends on the array size.
Thus, option A is incorrect.

Step 3: Analyze Option B - O(n)
O(n) is typical for linear search, which checks each element sequentially.
Binary Search eliminates half the elements each step, making it more efficient than O(n).
Thus, option B is incorrect.

Step 4: Analyze Option C - O(log n)
In the worst case (e.g., the target is not present or at the last comparison), Binary Search reduces the search space by half with each step.
The number of steps is logarithmic in the size of the array (n), leading to O(log n) complexity.
Thus, option C is correct.

Step 5: Analyze Option D - O(n^2)
O(n^2) is associated with algorithms like Bubble Sort, involving nested loops.
Binary Search does not use nested iterations, making this complexity irrelevant.
Thus, option D is incorrect.

Step 6: Conclusion
The worst-case time complexity of Binary Search is O(log n) due to its halving strategy, making option C the answer.

Quick Tip

To understand Binary Search:
- Requires a sorted array.
- Halves the search space each step (log n).
- Contrast with linear search (O(n)).
- Practice with small sorted arrays.

---

14. Which of the following is a characteristic of the "Queue" data structure?
(A) Follows a Last-In-First-Out (LIFO) order
(B) Allows insertion at both ends
(C) Follows a First-In-First-Out (FIFO) order
(D) Does not allow deletion of elements

Correct Answer: (C) Follows a First-In-First-Out (FIFO) order

Solution:

To determine the correct characteristic of the Queue data structure, we need to understand its defining properties.

Step 1: Understand the Queue Data Structure
A Queue is a linear data structure that follows the First-In-First-Out (FIFO) principle, meaning the first element added is the first to be removed.
Elements are typically inserted at the rear (enqueue) and removed from the front (dequeue).

Step 2: Analyze Option A - Follows a Last-In-First-Out (LIFO) order
LIFO is characteristic of a Stack, not a Queue, where the last element added is removed first.
Thus, option A is incorrect.

Step 3: Analyze Option B - Allows insertion at both ends
This describes a Deque (double-ended queue), not a standard Queue, which allows insertion only at the rear.
Thus, option B is incorrect.

Step 4: Analyze Option C - Follows a First-In-First-Out (FIFO) order
FIFO is the defining feature of a Queue, where elements are processed in the order they are added (e.g., like a line of people).
This aligns with the Queue's behavior.
Thus, option C is correct.

Step 5: Analyze Option D - Does not allow deletion of elements
Queues allow deletion (dequeue) from the front, making this statement false.
Thus, option D is incorrect.

Step 6: Conclusion
The primary characteristic of a Queue is its FIFO order, making option C the correct answer.

Quick Tip

To understand Queue:
- Remember FIFO: First In, First Out.
- Compare with Stack (LIFO) and Deque (both ends).
- Practice with enqueue/dequeue operations.
