# Section 1: Arrays and Strings in Java (MCQs)

### Q1. What will be the output of the following Java code?

String s1 = "hello";  
String s2 = new String("hello");  
System.out.println(s1 == s2);

**A)** true  
**B)** false  
**C)** Compilation error  
**D)** Runtime exception

**Answer: B**  
**Explanation:** s1 == s2 compares object references. s2 is a new object.

### Q2. What is the time complexity of searching an element in a sorted array using binary search?

**A)** O(1)  
**B)** O(n)  
**C)** O(log n)  
**D)** O(n log n)

**Answer: C**  
**Explanation:** Binary search halves the array each time → O(log n).

### Q3. Which of the following is **true** about Java arrays?

**A)** Arrays are dynamic in size  
**B)** Array indexes start from 1  
**C)** Arrays are objects in Java  
**D)** Array length can be changed at runtime

**Answer: C**  
**Explanation:** Arrays are objects, and arr.length is a final property.

### Q4. What will be the output?

String str = "abc";  
str.concat("def");  
System.out.println(str);

**A)** abc  
**B)** abcdef  
**C)** def  
**D)** Compilation error

**Answer: A**  
**Explanation:** Strings are immutable. concat() returns a new string.

### Q5. Which class should be used when you have to frequently modify a string in a multi-threaded environment?

**A)** StringBuilder  
**B)** String  
**C)** StringBuffer  
**D)** CharSequence

**Answer: C**  
**Explanation:** StringBuffer is synchronized and safe for multi-threading.

### Q6. What is the output of this code?

int[] arr = new int[5];  
System.out.println(arr[0]);

**A)** 0  
**B)** null  
**C)** undefined  
**D)** Compilation error

**Answer: A**  
**Explanation:** Java initializes int arrays with default value 0.

### Q7. What does this code do?

int[] nums = {2, 4, 6};  
System.out.println(nums.length);

**A)** Prints 2  
**B)** Prints 3  
**C)** Compilation error  
**D)** Runtime error

**Answer: B**  
**Explanation:** nums.length is 3.

### Q8. What is the worst-case time complexity of the KMP algorithm?

**A)** O(n^2)  
**B)** O(m\*n)  
**C)** O(n)  
**D)** O(log n)

**Answer: C**  
**Explanation:** KMP uses prefix table → linear in the size of input.

### Q9. In Rabin-Karp, what causes hash collisions?

**A)** Different strings having same hash value  
**B)** Incorrect loop usage  
**C)** Unsorted array  
**D)** String immutability

**Answer: A**  
**Explanation:** Rabin-Karp uses rolling hash → collisions may occur.

### Q10. What will the following code print?

String s1 = "Java";  
String s2 = "Java";  
System.out.println(s1 == s2);

**A)** true  
**B)** false  
**C)** null  
**D)** Compilation error

**Answer: A**  
**Explanation:** Both refer to the same object in the string pool.

### Q11. What is the output of this Java code?

String s = "abc";  
s.toUpperCase();  
System.out.println(s);

**A)** ABC  
**B)** abc  
**C)** Compilation error  
**D)** runtime error

**Answer: B**  
**Explanation:** toUpperCase() returns a new string, doesn’t modify the original.

### Q12. Which of the following is true for Java StringBuilder?

**A)** It is synchronized  
**B)** It is immutable  
**C)** It has better performance than StringBuffer in single-threaded scenarios  
**D)** It does not allow append()

**Answer: C**  
**Explanation:** StringBuilder is not synchronized and is faster than StringBuffer in single-threaded contexts.

### Q13. What will be printed?

char[][] grid = new char[2][2];  
System.out.println(grid[0][0]);

**A)** ’ ’ (space)  
**B)** ‘000’  
**C)** 0  
**D)** Compilation error

**Answer: B**  
**Explanation:** Default char value in Java is \u0000 (null char).

### Q14. What’s the output?

int[] a = {1, 2, 3};  
a[1] = a[1] + a[2];  
System.out.println(Arrays.toString(a));

**A)** [1, 5, 3]  
**B)** [1, 2, 3]  
**C)** [1, 3, 5]  
**D)** Compilation error

**Answer: A**  
**Explanation:** a[1] = 2 + 3 → a[1] becomes 5.

### Q15. What does this return?

String s = "abc";  
System.out.println(s.charAt(3));

**A)** ‘c’  
**B)** ’’ (empty)  
**C)** IndexOutOfBoundsException  
**D)** Compilation error

**Answer: C**  
**Explanation:** Index 3 is out of bounds (0–2 valid).

### Q16. Which method checks for equality of two string values (not references)?

**A)** ==  
**B)** equals()  
**C)** equalsIgnoreCase()  
**D)** compareTo()

**Answer: B**  
**Explanation:** equals() checks value equality.

### Q17. What is the time complexity to reverse an array in-place?

**A)** O(n log n)  
**B)** O(1)  
**C)** O(n)  
**D)** O(n^2)

**Answer: C**  
**Explanation:** Every element is visited once.

### Q18. How many objects are created here?

String s1 = new String("Java");

**A)** 1  
**B)** 2  
**C)** 0  
**D)** Depends on JVM

**Answer: B**  
**Explanation:** One in pool, one with new keyword.

### Q19. What is the default value of a String array element?

String[] s = new String[3];  
System.out.println(s[0]);

**A)** null  
**B)** ““  
**C)** 0  
**D)** Compilation error

**Answer: A**  
**Explanation:** Object arrays (like String) are initialized to null.

### Q20. Which function helps find a substring’s index in a string?

**A)** substring()  
**B)** match()  
**C)** indexOf()  
**D)** charAt()

**Answer: C**  
**Explanation:** indexOf() returns the first occurrence of the substring.

**Q21. Which method is used to extract a portion of a string in Java?**

**A)** extract()  
**B)** substring()  
**C)** split()  
**D)** slice()  
✅ **Answer: B**  
**Explanation:** substring() returns a specified portion of the string.

**Q22. What will the following code print?**

java

String str = "hello";

System.out.println(str.substring(1, 3));

**A)** he  
**B)** el  
**C)** ll  
**D)** ello  
✅ **Answer: B**  
**Explanation:** substring(1,3) returns characters from index 1 to 2 → "el".

**Q23. Which of the following methods splits a string into an array?**

**A)** split()  
**B)** break()  
**C)** tokenize()  
**D)** parse()  
✅ **Answer: A**  
**Explanation:** split() divides the string into parts based on a delimiter.

**Q24. How do you convert a String to a char[]?**

**A)** str.parse()  
**B)** str.charArray()  
**C)** str.toArray()  
**D)** str.toCharArray()  
✅ **Answer: D**  
**Explanation:** toCharArray() returns an array of characters from the string.

**Q25. Which class provides mutable strings in Java?**

**A)** String  
**B)** StringBuffer  
**C)** CharSequence  
**D)** CharArray  
✅ **Answer: B**  
**Explanation:** StringBuffer and StringBuilder allow string modification.

**Q26. What is returned by:**

java

Arrays.equals(new int[]{1,2}, new int[]{1,2});

**A)** true  
**B)** false  
**C)** Compilation error  
**D)** Reference error  
✅ **Answer: A**  
**Explanation:** Arrays.equals() checks if the content of both arrays is the same.

**Q27. What does this code return?**

java

int[][] matrix = new int[3][2];

System.out.println(matrix.length);

**A)** 2  
**B)** 3  
**C)** 6  
**D)** Compilation error  
✅ **Answer: B**  
**Explanation:** The outer array has 3 rows → matrix.length = 3.

**Q28. Which is the correct way to declare and initialize a 2D array?**

**A)** int arr[][] = new int[][]{{1,2},{3,4}};  
**B)** int[][] arr = {{1,2},{3,4}};  
**C)** int arr[][] = {{1,2},{3,4}};  
**D)** All of the above  
✅ **Answer: D**  
**Explanation:** All are valid Java syntax for 2D array initialization.

**Q29. What will happen here?**

java

int[] arr = new int[3];

System.out.println(arr[3]);

**A)** 0  
**B)** null  
**C)** ArrayIndexOutOfBoundsException  
**D)** Compilation error  
✅ **Answer: C**  
**Explanation:** Index 3 is invalid for an array of size 3 (0–2 valid).

**Q30. Which loop structure is best for linear traversal of an array in Java?**

**A)** while loop  
**B)** do-while loop  
**C)** for-each loop  
**D)** switch-case  
✅ **Answer: C**  
**Explanation:** Enhanced for-each loop is concise and prevents index errors.

**Section 2: Linked Lists in Java (MCQs)**

**Q1. What is the time complexity to insert a node at the beginning of a singly linked list?**

**A)** O(1)  
**B)** O(n)  
**C)** O(log n)  
**D)** O(n log n)

**Answer: A**  
**Explanation:** Inserting at the head only requires pointer adjustment.

**Q2. What is the correct way to define a node in a singly linked list in Java?**

class Node {

int data;

Node next;

}

**A)** Correct  
**B)** Missing constructor  
**C)** Invalid data type  
**D)** Compilation error

**Answer: A**  
**Explanation:** This is the standard way to define a node structure.

**Q3. In a singly linked list, how do you detect a cycle efficiently?**

**A)** HashMap  
**B)** Floyd’s Tortoise and Hare algorithm  
**C)** Recursion  
**D)** Reverse traversal

**Answer: B**  
**Explanation:** Floyd’s algorithm uses two pointers to detect a loop.

**Q4. What is the output of this code?**

LinkedList<Integer> list = new LinkedList<>();

list.add(10);

list.addFirst(5);

System.out.println(list);

**A)** [10, 5]  
**B)** [5, 10]  
**C)** [10]  
**D)** [5]

**Answer: B**  
**Explanation:** addFirst() inserts element at the head.

**Q5. Which interface must a class implement to be used with Collections.sort()?**

**A)** Serializable  
**B)** Comparator  
**C)** Comparable  
**D)** Iterable

**Answer: C**  
**Explanation:** Objects must implement Comparable to define natural ordering.

**Q6. Which LinkedList method removes and returns the first element?**

**A)** removeFirst()  
**B)** delete()  
**C)** poll()  
**D)** shift()

**Answer: A**  
**Explanation:** removeFirst() deletes and returns the first node.

**Q7. What will happen if you call .getLast() on an empty LinkedList?**

**A)** Returns null  
**B)** Throws NoSuchElementException  
**C)** Returns 0  
**D)** Compilation error

**Answer: B**  
**Explanation:** It throws NoSuchElementException if the list is empty.

**Q8. In Java's LinkedList class, which operations have O(1) complexity?**

**A)** add at end  
**B)** add at start  
**C)** remove first  
**D)** All of the above

**Answer: D**  
**Explanation:** Java’s LinkedList maintains head and tail pointers.

**Q9. Which method do you use to iterate through a LinkedList?**

**A)** for-each loop  
**B)** iterator()  
**C)** listIterator()  
**D)** All of the above

**Answer: D**  
**Explanation:** All are valid for LinkedList traversal.

**Q10. What is the primary advantage of a doubly linked list over a singly linked list?**

**A)** Better memory efficiency  
**B)** Ability to traverse backward  
**C)** Less pointer usage  
**D)** Simpler insertion logic

**Answer: B**  
**Explanation:** Doubly linked lists support bidirectional traversal.

**Q11. What is the space complexity of a singly linked list with n nodes?**

**A)** O(1)  
**B)** O(log n)  
**C)** O(n)  
**D)** O(n²)  
✅ **Answer: C**  
**Explanation:** Each node holds data and a reference, resulting in linear space.

**Q12. Which Java class is best suited for implementing a queue with constant-time insertion and deletion from both ends?**

**A)** Stack  
**B)** LinkedList  
**C)** ArrayList  
**D)** TreeSet  
✅ **Answer: B**  
**Explanation:** LinkedList implements Deque, allowing O(1) insertion/removal at both ends.

**Q13. What is the output of the following code?**

java

LinkedList<String> ll = new LinkedList<>();

ll.add("A");

ll.add("B");

ll.add(1, "C");

System.out.println(ll);

**A)** [A, B, C]  
**B)** [C, A, B]  
**C)** [A, C, B]  
**D)** [A, B]  
✅ **Answer: C**  
**Explanation:** .add(1, "C") inserts "C" at index 1.

**Q14. What is the worst-case time complexity to search for an element in a singly linked list?**

**A)** O(log n)  
**B)** O(1)  
**C)** O(n)  
**D)** O(n log n)  
✅ **Answer: C**  
**Explanation:** A linear traversal is needed in the worst case.

**Q15. How would you reverse a singly linked list in-place?**

**A)** Recursively re-append nodes  
**B)** Create a new list and copy values  
**C)** Iteratively change next pointers  
**D)** Use built-in Collections.reverse()  
✅ **Answer: C**  
**Explanation:** Iterative pointer manipulation is used to reverse in-place.

**Q16. What is a sentinel node in linked lists?**

**A)** A null node  
**B)** A node holding metadata  
**C)** A dummy head node to simplify edge cases  
**D)** Last element in list  
✅ **Answer: C**  
**Explanation:** Sentinel simplifies edge-case handling (e.g., head/tail insertion).

**Q17. Which operation is more efficient in LinkedList than ArrayList?**

**A)** Random access  
**B)** Insert at end  
**C)** Insert at beginning  
**D)** None  
✅ **Answer: C**  
**Explanation:** LinkedList offers O(1) insertion at the beginning.

**Q18. What will this code output?**

java

LinkedList<Integer> list = new LinkedList<>();

list.add(1);

list.add(2);

list.add(3);

list.removeFirst();

System.out.println(list);

**A)** [1, 2, 3]  
**B)** [2, 3]  
**C)** [1, 3]  
**D)** [3]  
✅ **Answer: B**  
**Explanation:** removeFirst() removes 1.

**Q19. In a circular linked list with one node, what does that node point to?**

**A)** null  
**B)** Itself  
**C)** Head  
**D)** Tail  
✅ **Answer: B**  
**Explanation:** It points to itself to maintain the circular structure.

**Q20. What is returned by .poll() if the list is empty?**

**A)** null  
**B)** Exception  
**C)** -1  
**D)** false  
✅ **Answer: A**  
**Explanation:** poll() returns null on an empty list (safe removal).

**Q21. How can you convert a LinkedList to an array in Java?**

**A)** toList()  
**B)** convertToArray()  
**C)** toArray()  
**D)** asList()  
✅ **Answer: C**  
**Explanation:** toArray() returns an array from the linked list.

**Q22. What happens if you insert an element at index = size of list in Java’s LinkedList?**

**A)** Inserts at end  
**B)** Throws exception  
**C)** Inserts at beginning  
**D)** Overwrites last element  
✅ **Answer: A**  
**Explanation:** Adding at size appends the element.

**Q23. Which method adds an element to the end of a LinkedList?**

**A)** addLast()  
**B)** append()  
**C)** push()  
**D)** offerFirst()  
✅ **Answer: A**  
**Explanation:** addLast() appends an element at the tail.

**Q24. What does the size() method of LinkedList return?**

**A)** Number of elements  
**B)** Memory in bytes  
**C)** Last index  
**D)** Null if empty  
✅ **Answer: A**  
**Explanation:** It returns the total number of nodes.

**Q25. Which LinkedList method retrieves but does not remove the head element?**

**A)** get()  
**B)** peek()  
**C)** poll()  
**D)** pop()  
✅ **Answer: B**  
**Explanation:** peek() returns the first element without removing it.

**Q26. Which method adds a collection at the end of a LinkedList?**

**A)** extend()  
**B)** add()  
**C)** addAll()  
**D)** appendList()  
✅ **Answer: C**  
**Explanation:** addAll() appends all elements of another collection.

**Q27. Which interface does Java’s LinkedList implement for FIFO behavior?**

**A)** List  
**B)** Deque  
**C)** Queue  
**D)** Collection  
✅ **Answer: C**  
**Explanation:** Implements Queue for FIFO operations.

**Q28. What will .removeLast() do in an empty LinkedList?**

**A)** Throws NoSuchElementException  
**B)** Returns null  
**C)** Returns -1  
**D)** Compilation error  
✅ **Answer: A**  
**Explanation:** It throws exception if list is empty.

**Q29. Which of these operations is not efficient in LinkedList?**

**A)** Adding at beginning  
**B)** Adding at end  
**C)** Deleting middle element  
**D)** Removing first element  
✅ **Answer: C**  
**Explanation:** Accessing middle element takes O(n) time in LinkedList.

**Q30. Which collection uses LinkedList internally by default?**

**A)** Stack  
**B)** PriorityQueue  
**C)** Deque  
**D)** BlockingQueue  
✅ **Answer: D**  
**Explanation:** LinkedBlockingQueue uses a linked list structure under the hood.

**🧮 Section 3: Stacks and Queues in Java (MCQs)**

**Q1. What is the time complexity of push() and pop() operations in a Stack?**

**A)** O(log n)  
**B)** O(n)  
**C)** O(1)  
**D)** O(n log n)  
✅ **Answer: C**  
**Explanation:** Stack operations (push/pop) take constant time if implemented using array or linked list.

**Q2. What is the output of this code?**

Stack<Integer> stack = new Stack<>();

stack.push(1);

stack.push(2);

stack.pop();

System.out.println(stack.peek());

**A)** 1  
**B)** 2  
**C)** Empty  
**D)** Compilation error  
✅ **Answer: A**  
**Explanation:** 2 is popped, 1 remains on top.

**Q3. Which Java class implements a Queue and allows element insertion/removal from both ends?**

**A)** ArrayList  
**B)** Deque  
**C)** PriorityQueue  
**D)** TreeMap  
✅ **Answer: B**  
**Explanation:** Deque supports FIFO and LIFO behavior.

**Q4. Which method is used to safely remove the head of a queue without throwing an exception?**

**A)** remove()  
**B)** pop()  
**C)** poll()  
**D)** get()  
✅ **Answer: C**  
**Explanation:** poll() returns null instead of throwing NoSuchElementException.

**Q5. What is the default implementation of Queue in Java Collections Framework?**

**A)** LinkedList  
**B)** HashSet  
**C)** Stack  
**D)** HashMap  
✅ **Answer: A**  
**Explanation:** LinkedList implements the Queue interface.

**Q6. What data structure is typically used for recursive function call management internally?**

**A)** Queue  
**B)** Tree  
**C)** Heap  
**D)** Stack  
✅ **Answer: D**  
**Explanation:** The call stack stores recursive function calls.

**Q7. What is the output of this queue operation?**

Queue<Integer> q = new LinkedList<>();

q.add(5);

q.add(10);

q.remove();

System.out.println(q.peek());

**A)** 10  
**B)** 5  
**C)** null  
**D)** Compilation error  
✅ **Answer: A**  
**Explanation:** 5 is removed, 10 remains at the head.

**Q8. What’s the difference between Stack and Deque in Java?**

**A)** Stack is synchronized, Deque is not  
**B)** Deque is LIFO, Stack is FIFO  
**C)** Stack only supports LIFO, Deque supports both LIFO & FIFO  
**D)** Deque is legacy, Stack is newer  
✅ **Answer: C**  
**Explanation:** Deque is more versatile and can behave as both stack and queue.

**Q9. Which method would you use to insert an element at the front of a Deque?**

**A)** offer()  
**B)** addLast()  
**C)** offerFirst()  
**D)** pushBack()  
✅ **Answer: C**  
**Explanation:** offerFirst() inserts at the front without exception risk.

**Q10. Which stack-related class is preferred in modern Java (over Stack) for thread-safe operations?**

**A)** PriorityQueue  
**B)** ConcurrentStack  
**C)** Deque  
**D)** LinkedBlockingDeque  
✅ **Answer: D**  
**Explanation:** LinkedBlockingDeque is preferred for concurrent environments.

**Q11. What is the behavior of Java’s Queue.remove() method when the queue is empty?**

**A)** Returns null  
**B)** Throws NullPointerException  
**C)** Throws NoSuchElementException  
**D)** Does nothing  
✅ **Answer: C**  
**Explanation:** Unlike poll(), remove() throws NoSuchElementException if the queue is empty.

**Q12. What is the output of this code?**

Deque<Integer> dq = new ArrayDeque<>();

dq.push(1);

dq.push(2);

System.out.println(dq.pop());

**A)** 2  
**B)** 1  
**C)** 0  
**D)** null  
✅ **Answer: A**  
**Explanation:** push() adds to the front; pop() removes from the front → LIFO.

**Q13. Which Java class would you use for a priority-based queue?**

**A)** Deque  
**B)** PriorityQueue  
**C)** Stack  
**D)** BlockingQueue  
✅ **Answer: B**  
**Explanation:** PriorityQueue orders elements according to their natural order or a comparator.

**Q14. Which method retrieves and removes the last element in a Deque?**

**A)** poll()  
**B)** removeLast()  
**C)** pop()  
**D)** getLast()  
✅ **Answer: B**  
**Explanation:** removeLast() removes the last element.

**Q15. What is the space complexity of a queue implemented using a linked list with n elements?**

**A)** O(1)  
**B)** O(log n)  
**C)** O(n)  
**D)** O(n²)  
✅ **Answer: C**  
**Explanation:** Each node takes space → linear complexity.

**Q16. What does the peek() method return on an empty stack or queue?**

**A)** null  
**B)** throws exception  
**C)** 0  
**D)** -1  
✅ **Answer: A**  
**Explanation:** peek() returns null instead of throwing an exception.

**Q17. Which method inserts an element at the end of a queue without exception?**

**A)** addLast()  
**B)** offer()  
**C)** enqueue()  
**D)** insert()  
✅ **Answer: B**  
**Explanation:** offer() inserts safely at the end (tail) of the queue.

**Q18. What happens if you call pop() on an empty Stack?**

**A)** Returns null  
**B)** Returns -1  
**C)** Throws EmptyStackException  
**D)** Compilation error  
✅ **Answer: C**  
**Explanation:** pop() throws EmptyStackException if stack is empty.

**Q19. Which of the following can be used to implement a browser’s back/forward navigation?**

**A)** Queue  
**B)** Stack  
**C)** TreeMap  
**D)** HashMap  
✅ **Answer: B**  
**Explanation:** Backtracking history can be modeled using stacks (LIFO).

**Q20. Which Java method pushes an element on the top of a Stack?**

**A)** push()  
**B)** add()  
**C)** append()  
**D)** insert()  
✅ **Answer: A**  
**Explanation:** push() adds an element to the top of the stack.

**Q21. What will this code print?**

Deque<Integer> d = new ArrayDeque<>();

d.addFirst(3);

d.addLast(5);

System.out.println(d);

**A)** [5, 3]  
**B)** [3, 5]  
**C)** [3]  
**D)** Compilation error  
✅ **Answer: B**  
**Explanation:** addFirst adds at front, addLast at back.

**Q22. What interface do both Queue and Deque extend?**

**A)** Iterable  
**B)** Collection  
**C)** List  
**D)** Map  
✅ **Answer: B**  
**Explanation:** Queue and Deque are part of the Java Collection hierarchy.

**Q23. Which is a better alternative to Stack class for stack operations in Java 8+?**

**A)** Vector  
**B)** ArrayDeque  
**C)** ArrayList  
**D)** TreeSet  
✅ **Answer: B**  
**Explanation:** ArrayDeque is preferred over Stack (legacy) for better performance.

**Q24. How to check if a Deque is empty?**

**A)** isEmpty()  
**B)** size() == 1  
**C)** equals(null)  
**D)** check()  
✅ **Answer: A**  
**Explanation:** isEmpty() is standard method for checking emptiness.

**Q25. Which collection class allows LIFO and FIFO behavior based on method usage?**

**A)** Stack  
**B)** ArrayDeque  
**C)** PriorityQueue  
**D)** HashMap  
✅ **Answer: B**  
**Explanation:** ArrayDeque allows both stack and queue operations.

**Q26. What happens when offer() fails to insert due to capacity limits?**

**A)** Returns false  
**B)** Throws exception  
**C)** Inserts anyway  
**D)** Returns null  
✅ **Answer: A**  
**Explanation:** offer() fails silently by returning false.

**Q27. What does push() do in an ArrayDeque?**

**A)** Adds at the back  
**B)** Adds at the front  
**C)** Removes from front  
**D)** Overwrites first element  
✅ **Answer: B**  
**Explanation:** push() = addFirst() in ArrayDeque.

**Q28. Which queue implementation is thread-safe by default in Java?**

**A)** ArrayDeque  
**B)** LinkedList  
**C)** ConcurrentLinkedQueue  
**D)** TreeMap  
✅ **Answer: C**  
**Explanation:** ConcurrentLinkedQueue is designed for multi-threading.

**Q29. What will queue.size() return after 3 add() and 2 remove() operations?**

**A)** 1  
**B)** 2  
**C)** 3  
**D)** 5  
✅ **Answer: B**  
**Explanation:** 3 additions – 2 removals = 1 element remaining.

**Q30. Which stack operation is not directly available in the Queue interface?**

**A)** offer()  
**B)** poll()  
**C)** push()  
**D)** peek()  
✅ **Answer: C**  
**Explanation:** Queue doesn't support push(); it’s a stack-specific method.

🌲 **Section 4: Trees and Graphs in Java**

**Q1. What is the time complexity of searching in a balanced binary search tree (BST)?**

**A)** O(n)  
**B)** O(log n)  
**C)** O(n log n)  
**D)** O(1)  
✅ **Answer: B**  
**Explanation:** In a balanced BST, each search reduces problem size by half → O(log n).

**Q2. Which traversal method visits nodes in the order: Left → Root → Right?**

**A)** Preorder  
**B)** Inorder  
**C)** Postorder  
**D)** Level Order  
✅ **Answer: B**  
**Explanation:** Inorder = L → Root → R.

**Q3. How is a binary tree typically represented in Java?**

**A)** Array  
**B)** HashMap  
**C)** Custom class with left and right pointers  
**D)** LinkedList  
✅ **Answer: C**  
**Explanation:** TreeNode class with left and right pointers.

**Q4. Which data structure is used to implement Depth First Search (DFS)?**

**A)** Queue  
**B)** Stack  
**C)** Heap  
**D)** PriorityQueue  
✅ **Answer: B**  
**Explanation:** DFS uses a stack (explicit or call stack).

**Q5. Which traversal guarantees the shortest path in an unweighted graph?**

**A)** DFS  
**B)** BFS  
**C)** Inorder  
**D)** Preorder  
✅ **Answer: B**  
**Explanation:** BFS visits nodes level by level.

**Q6. What does the following code print?**

TreeMap<Integer, String> map = new TreeMap<>();

map.put(3, "C");

map.put(1, "A");

map.put(2, "B");

System.out.println(map.keySet());

**A)** [3, 1, 2]  
**B)** [1, 2, 3]  
**C)** [2, 1, 3]  
**D)** Random  
✅ **Answer: B**  
**Explanation:** TreeMap maintains sorted key order.

**Q7. Which of the following is true about a complete binary tree?**

**A)** All levels filled  
**B)** Every node has 2 children  
**C)** All leaves are at the same level  
**D)** Nodes are filled from left to right  
✅ **Answer: D**  
**Explanation:** In a complete binary tree, nodes are filled left to right.

**Q8. How many edges are in a tree with n nodes?**

**A)** n  
**B)** n-1  
**C)** 2n  
**D)** log n  
✅ **Answer: B**  
**Explanation:** Every node except root has one parent → n-1 edges.

**Q9. What is the output of the code?**

Queue<Integer> q = new LinkedList<>();

q.add(10);

q.add(20);

q.poll();

System.out.println(q.peek());

**A)** 10  
**B)** 20  
**C)** null  
**D)** Compilation error  
✅ **Answer: B**  
**Explanation:** 10 is removed, 20 is at front.

**Q10. What traversal is used in BST to get sorted values?**

1. Inorder  
   **B)** Preorder  
   **C)** Postorder  
   **D)** DFS  
   ✅ **Answer: A**

**Q11. Which data structure is most efficient for implementing level-order traversal in a binary tree?**

**A)** Stack  
**B)** Queue  
**C)** PriorityQueue  
**D)** LinkedList  
✅ **Answer: B**  
**Explanation:** Level-order traversal uses a Queue to maintain the current level.

**Q12. Which of the following is not true about a binary search tree (BST)?**

**A)** Left subtree has nodes < root  
**B)** Right subtree has nodes > root  
**C)** Inorder traversal results in sorted output  
**D)** Duplicate values are always allowed  
✅ **Answer: D**  
**Explanation:** BSTs **may not allow duplicates** based on implementation.

**Q13. What is the time complexity of inserting a node in a balanced BST?**

**A)** O(1)  
**B)** O(n)  
**C)** O(log n)  
**D)** O(n log n)  
✅ **Answer: C**  
**Explanation:** A balanced BST maintains O(log n) for insert/search/delete.

**Q14. Which traversal visits root before all subtrees?**

**A)** Preorder  
**B)** Inorder  
**C)** Postorder  
**D)** Level Order  
✅ **Answer: A**  
**Explanation:** Preorder: Root → Left → Right

**Q15. Which traversal is best suited for deleting the tree?**

**A)** Preorder  
**B)** Postorder  
**C)** Inorder  
**D)** Level Order  
✅ **Answer: B**  
**Explanation:** Postorder deletes children before parent.

**Q16. How many child nodes can a binary tree node have at most?**

**A)** 1  
**B)** 2  
**C)** 3  
**D)** Unlimited  
✅ **Answer: B**  
**Explanation:** Binary tree nodes have a max of 2 children: left and right.

**Q17. What is the maximum number of nodes at level l in a binary tree?**

**A)** 2^l  
**B)** 2^l - 1  
**C)** 2^(l-1)  
**D)** l  
✅ **Answer: C**  
**Explanation:** Level l can have at most 2^(l - 1) nodes.

**Q18. What is the output of the following?**

Map<Integer, String> tree = new TreeMap<>();

tree.put(5, "Five");

tree.put(2, "Two");

tree.put(8, "Eight");

System.out.println(tree.values());

**A)** [Five, Two, Eight]  
**B)** [Two, Five, Eight]  
**C)** [8, 2, 5]  
**D)** [Eight, Two, Five]  
✅ **Answer: B**  
**Explanation:** TreeMap sorts by keys.

**Q19. What is a leaf node in a tree?**

**A)** A node with only one child  
**B)** A node with no children  
**C)** A node with two children  
**D)** A node at level 0  
✅ **Answer: B**  
**Explanation:** A **leaf node** is a node with **no children** — it marks the end of a branch.

**Q20. What is the best representation for a sparse graph?**

**A)** Adjacency matrix  
**B)** Linked list  
**C)** Adjacency list  
**D)** 2D array  
✅ **Answer: C**  
**Explanation:** An **adjacency list** uses less space and is more efficient for **sparse graphs** (few edges).

**Q21. Which traversal uses recursion by default in its implementation?**

**A)** BFS  
**B)** DFS  
**C)** Level-order  
**D)** Random walk  
✅ **Answer: B**  
**Explanation:** **Depth First Search (DFS)** is often implemented **recursively**, leveraging the call stack.

**Q22. What is the default return of get() from a Map if the key doesn’t exist?**

**A)** null  
**B)** 0  
**C)** -1  
**D)** Throws Exception  
✅ **Answer: A**  
**Explanation:** Java’s Map.get() returns **null** if the key is not found.

**Q23. What is the time complexity to find the height of a binary tree?**

**A)** O(1)  
**B)** O(log n)  
**C)** O(n)  
**D)** O(n²)  
✅ **Answer: C**  
**Explanation:** You must **traverse all nodes** in the worst case → O(n).

**Q24. Which exception is likely thrown if you forget to check for null in a tree node?**

**A)** IndexOutOfBoundsException  
**B)** NullPointerException  
**C)** ClassCastException  
**D)** NumberFormatException  
✅ **Answer: B**  
**Explanation:** Accessing a method or field on a null reference results in a **NullPointerException**.

**Q25. Which traversal would result in D B E A F C for the following tree?**

A

/ \

B C

/ \ \

D E F

**A)** Inorder  
**B)** Preorder  
**C)** Postorder  
**D)** Level Order  
✅ **Answer: A**  
**Explanation:** **Inorder traversal** of this tree is: D B E A F C.

**Q26. What is the worst-case time complexity of DFS on a graph with V vertices and E edges?**

**A)** O(V)  
**B)** O(V + E)  
**C)** O(V \* E)  
**D)** O(log V)  
✅ **Answer: B**  
**Explanation:** **Each vertex and each edge** is visited once → O(V + E).

**Q27. Which of the following algorithms uses BFS internally?**

**A)** Kruskal's  
**B)** Prim's  
**C)** Dijkstra (unweighted)  
**D)** Bellman-Ford  
✅ **Answer: C**  
**Explanation:** **BFS** is used to find the shortest path in an **unweighted graph**, just like Dijkstra without weights.

**Q28. A graph with no cycles and V nodes with V-1 edges is called:**

**A)** DAG  
**B)** Spanning Tree  
**C)** Heap  
**D)** Forest  
✅ **Answer: B**  
**Explanation:** A **tree** (or spanning tree) is a **connected acyclic graph** with exactly **V - 1** edges.

**Q29. Which method is used in Java to iterate over a Set of visited nodes in a graph?**

**A)** for loop  
**B)** while loop  
**C)** iterator()  
**D)** map()  
✅ **Answer: C**  
**Explanation:** Set implements Iterable, so you can use iterator() to loop over it.

**Q30. Which Java class would you use for sorted unique node storage?**

**A)** HashSet  
**B)** ArrayList  
**C)** TreeSet  
**D)** HashMap  
✅ **Answer: C**  
**Explanation:** TreeSet stores **unique** values in **sorted** order — ideal for node-based sets in trees or graphs.

**Section 5: Recursion and Backtracking in Java**

**Q1. What is the key condition required in every recursive function?**

**A)** A loop  
**B)** A base case  
**C)** A helper method  
**D)** Return type  
✅ **Answer: B**  
**Explanation:** Without a base case, recursion would lead to infinite calls and a **StackOverflowError**.

**Q2. What does the following recursive method return?**

int sum(int n) {

if (n == 0) return 0;

return n + sum(n - 1);

}

**A)** 0  
**B)** n  
**C)** Sum of 1 to n  
**D)** Infinite recursion  
✅ **Answer: C**  
**Explanation:** This is the classic recursive formula for summing 1 to n.

**Q3. In recursion, what does each recursive call store in the stack frame?**

**A)** Only base case  
**B)** Entire variable state and return address  
**C)** Final answer  
**D)** Stack memory location  
✅ **Answer: B**  
**Explanation:** Java stores the function’s **local variables** and **return point** on the call stack.

**Q4. Which of the following problems is best suited for backtracking?**

**A)** Binary Search  
**B)** Sorting  
**C)** N-Queens  
**D)** Matrix multiplication  
✅ **Answer: C**  
**Explanation:** **Backtracking** systematically searches for solutions by undoing choices (e.g., N-Queens).

**Q5. Which statement is true about backtracking?**

**A)** It tries all possibilities blindly  
**B)** It avoids recursion  
**C)** It explores and undoes incorrect paths  
**D)** It never terminates  
✅ **Answer: C**  
**Explanation:** Backtracking **explores choices** and **prunes paths** that violate constraints.

**Q6. What is the time complexity of generating all permutations of a string with n characters?**

**A)** O(n)  
**B)** O(n²)  
**C)** O(n!)  
**D)** O(2^n)  
✅ **Answer: C**  
**Explanation:** Each character can be placed in all n positions → **n! permutations**.

**Q7. Which condition is mandatory in backtracking?**

**A)** Loop from end to start  
**B)** Pruning  
**C)** Base condition to stop recursion  
**D)** Non-recursive function  
✅ **Answer: C**  
**Explanation:** Like recursion, backtracking must **terminate** via base condition.

**Q8. In the N-Queens problem, what does the backtracking algorithm track?**

**A)** Queen counts  
**B)** Row and column collisions  
**C)** Diagonal threats  
**D)** All of the above  
✅ **Answer: D**  
**Explanation:** Valid queen positions must avoid **row**, **column**, and **diagonal** attacks.

**Q9. What does this method do?**

java

void reverse(String str) {

if (str == null || str.length() <= 1) {

System.out.print(str);

return;

}

reverse(str.substring(1));

System.out.print(str.charAt(0));

}

**A)** Prints original string  
**B)** Reverses string recursively  
**C)** Prints null  
**D)** Throws error  
✅ **Answer: B**  
**Explanation:** The method recursively reverses the string and prints it.

**Q10. What happens if you call a recursive function with no base case?**

**A)** Function completes  
**B)** Infinite loop  
**C)** Compile-time error  
**D)** StackOverflowError  
✅ **Answer: D**  
**Explanation:** Without a base case, recursion **never terminates** and overflows the call stack.

**Q11. Which technique can reduce the repeated computation in recursive functions like Fibonacci?**

**A)** Sorting  
**B)** Memoization  
**C)** Backtracking  
**D)** Shuffling  
✅ **Answer: B**  
**Explanation:** **Memoization** stores previously computed values to avoid redundant recursion.

**Q12. What is the output of this code?**int factorial(int n) {

if (n <= 1) return 1;

return n \* factorial(n - 1);

}

System.out.println(factorial(4));

**A)** 24  
**B)** 10  
**C)** 4  
**D)** StackOverflowError  
✅ **Answer: A**  
**Explanation:** 4! = 4×3×2×1 = **24**

**Q13. Which recursive pattern explores multiple recursive calls per level?**

**A)** Linear recursion  
**B)** Tree recursion  
**C)** Tail recursion  
**D)** Iterative recursion  
✅ **Answer: B**  
**Explanation:** Tree recursion occurs when a function calls itself more than once per call (e.g., Fibonacci).

**Q14. Which of the following is NOT a characteristic of backtracking?**

**A)** Recursive calls  
**B)** Exploring all paths  
**C)** Greedy selection  
**D)** Undoing decisions  
✅ **Answer: C**  
**Explanation:** Greedy algorithms make irreversible choices — **not** used in backtracking.

**Q15. What does the recursion tree model help visualize?**

**A)** Stack memory allocation  
**B)** Number of function calls  
**C)** Final return values only  
**D)** Iteration count  
✅ **Answer: B**  
**Explanation:** A **recursion tree** helps trace the number and structure of recursive calls.

**Q16. What’s the best way to avoid infinite recursion?**

**A)** Use try-catch  
**B)** Avoid function calls  
**C)** Define and reach a base case  
**D)** Use static methods  
✅ **Answer: C**  
**Explanation:** A correct and reachable **base case** prevents infinite recursion.

**Q17. Which of the following recursive calls is tail-recursive?**

int sumTail(int n, int acc) {

if (n == 0) return acc;

return sumTail(n - 1, acc + n);

}

**A)** Tail-recursive  
**B)** Not tail-recursive  
**C)** Iterative  
**D)** Stack-safe only  
✅ **Answer: A**  
**Explanation:** The recursive call is the **last operation**, so it is tail-recursive.

**Q18. What is the time complexity of generating all subsets of an array with n elements?**

**A)** O(n)  
**B)** O(n!)  
**C)** O(2^n)  
**D)** O(n²)  
✅ **Answer: C**  
**Explanation:** Each element has two choices (in or out) → 2ⁿ subsets.

**Q19. What will happen with this recursion?**

void call() {

System.out.println("Call");

call();

}

**A)** Infinite printing  
**B)** StackOverflowError eventually  
**C)** Compile error  
**D)** Nothing  
✅ **Answer: B**  
**Explanation:** No base case → infinite recursion → stack overflow.

**Q20. What’s the difference between backtracking and brute-force?**

**A)** Backtracking is recursive, brute-force is not  
**B)** Brute-force generates all solutions blindly; backtracking **eliminates invalid paths early**  
**C)** Brute-force is faster  
**D)** No difference  
✅ **Answer: B**  
**Explanation:** Backtracking is **optimized** brute-force with **pruning**.

**Q21. In recursion, which of the following statements is true about function call stack depth?**

**A)** Each recursive call increases stack size  
**B)** It remains the same  
**C)** Decreases with each call  
**D)** Stack is not used  
✅ **Answer: A**  
**Explanation:** Each recursive call is added to the **call stack** → depth increases.

**Q22. What is the best strategy to implement constraints in a backtracking algorithm?**

**A)** Apply constraints at base case  
**B)** Apply constraints after recursion  
**C)** Apply constraints **before** deeper recursive calls  
**D)** Apply constraints using global variables only  
✅ **Answer: C**  
**Explanation:** Early **constraint checking** avoids unnecessary recursion → more efficient.

**Q23. Which algorithmic problem can be solved by backtracking?**

**A)** Subset sum  
**B)** Sudoku solver  
**C)** Word search  
**D)** All of the above  
✅ **Answer: D**  
**Explanation:** All these involve **trying, checking, and undoing** → perfect for backtracking.

**Q24. What is the key difference between recursion and iteration?**

**A)** Recursion uses loops  
**B)** Recursion uses call stack; iteration uses control structures  
**C)** Recursion is always faster  
**D)** Iteration consumes more memory  
✅ **Answer: B**  
**Explanation:** Recursion uses **function stack frames** to simulate loops.

**Q25. What is the stack frame size for a recursive call?**

**A)** Depends on input  
**B)** Fixed  
**C)** Variable but constant per function  
**D)** Infinite  
✅ **Answer: C**  
**Explanation:** Stack frame size is **fixed for each function**, but total depth varies.

**Q26. Which recursive problem solves the maximum value path in a matrix with backtracking?**

**A)** Longest increasing path  
**B)** Word break  
**C)** Power set  
**D)** Binary search  
✅ **Answer: A**  
**Explanation:** LIP involves choosing path values and backtracking when dead-ends occur.

**Q27. What is the maximum stack depth in Java (approximate)?**

**A)** 64  
**B)** 1024  
**C)** 4096–10000 (depends on OS/JVM)  
**D)** Unlimited  
✅ **Answer: C**  
**Explanation:** Stack depth varies, but typically ~5000–10000 calls are allowed.

**Q28. What Java keyword can be used to return early in recursion?**

**A)** continue  
**B)** this  
**C)** break  
**D)** return  
✅ **Answer: D**  
**Explanation:** return exits the current function call.

**Q29. What’s the best case for avoiding repeated recursive calls in a grid or maze?**

**A)** Use Set  
**B)** Use visited[][] array  
**C)** Use List  
**D)** Use Stack  
✅ **Answer: B**  
**Explanation:** A visited matrix tracks already explored paths to avoid cycles.

**Q30. Which of the following statements is true about tail-recursion in Java?**

**A)** JVM optimizes tail recursion  
**B)** Java supports automatic tail call optimization  
**C)** Java does **not** optimize tail recursion  
**D)** Tail recursion leads to infinite loop  
✅ **Answer: C**  
**Explanation:** Unlike some languages, **Java doesn’t support tail-call optimization**, so tail-recursive functions can still cause stack overflow.

⚙️ **Section 6: Heaps, Hashing, and Tries in Java**

**Q1. What is the default behavior of Java's PriorityQueue<Integer>?**

**A)** Max-Heap  
**B)** Min-Heap  
**C)** Stack behavior  
**D)** Random order  
✅ **Answer: B**  
**Explanation:** By default, PriorityQueue in Java acts as a **min-heap**.

**Q2. Which Java collection guarantees O(1) average time for put() and get() operations?**

**A)** TreeMap  
**B)** LinkedList  
**C)** HashMap  
**D)** ArrayList  
✅ **Answer: C**  
**Explanation:** HashMap offers **constant time** for put() and get() on average.

**Q3. What happens when two keys have the same hashCode in a HashMap?**

**A)** Both are discarded  
**B)** One replaces the other  
**C)** They are stored in a bucket list (chaining)  
**D)** Exception is thrown  
✅ **Answer: C**  
**Explanation:** Collisions are handled via **chaining** (linked list or tree).

**Q4. Which method must be overridden to ensure key uniqueness in HashMap?**

**A)** toString()  
**B)** clone()  
**C)** equals() and hashCode()  
**D)** finalize()  
✅ **Answer: C**  
**Explanation:** Both equals() and hashCode() must be overridden to ensure proper hashing behavior.

**Q5. What is the time complexity of inserting into a min-heap with n elements?**

**A)** O(1)  
**B)** O(log n)  
**C)** O(n)  
**D)** O(n log n)  
✅ **Answer: B**  
**Explanation:** Insertions in a heap take **O(log n)** due to heapify-up.

**Q6. Which data structure is used to implement a Trie?**

**A)** Array of nodes  
**B)** HashMap of characters  
**C)** Linked list  
**D)** Both A and B  
✅ **Answer: D**  
**Explanation:** A Trie node may use an **array (for 26 letters)** or a **HashMap** (for variable character sets).

**Q7. What does the following code print?**

PriorityQueue<Integer> pq = new PriorityQueue<>();

pq.add(10); pq.add(5); pq.add(20);

System.out.println(pq.poll());

**A)** 5  
**B)** 10  
**C)** 20  
**D)** 0  
✅ **Answer: A**  
**Explanation:** Poll removes the **smallest** (min-heap) → 5.

**Q8. What is the load factor in a HashMap?**

**A)** Total capacity  
**B)** Ratio of size to capacity  
**C)** Growth rate  
**D)** Memory used per entry  
✅ **Answer: B**  
**Explanation:** Load factor = size / capacity → triggers **rehashing** when exceeded.

**Q9. What is the default initial capacity and load factor of a Java HashMap?**

**A)** 8, 0.5  
**B)** 16, 0.75  
**C)** 32, 1.0  
**D)** 10, 0.6  
✅ **Answer: B**  
**Explanation:** Default capacity = **16**, load factor = **0.75**

**Q10. Which of the following is true about a Trie?**

**A)** Used for prefix matching  
**B)** Good for dictionary lookups  
**C)** Each node stores a character  
**D)** All of the above  
✅ **Answer: D**  
**Explanation:** Tries are designed for **prefix-based searching** and store character nodes.

**Q11. What is the time complexity of inserting a word into a Trie of length L?**

**A)** O(log L)  
**B)** O(1)  
**C)** O(L)  
**D)** O(L²)  
✅ **Answer: C**  
**Explanation:** You must traverse each character → insertion is **O(L)** where L is word length.

**Q12. Which Java class maintains insertion order of key-value pairs?**

**A)** HashMap  
**B)** LinkedHashMap  
**C)** TreeMap  
**D)** PriorityQueue  
✅ **Answer: B**  
**Explanation:** LinkedHashMap maintains **insertion order** using a doubly linked list.

**Q13. Which collection ensures that keys are sorted?**

**A)** HashMap  
**B)** TreeMap  
**C)** HashSet  
**D)** LinkedList  
✅ **Answer: B**  
**Explanation:** TreeMap automatically sorts the keys in **natural order** or using a **Comparator**.

**Q14. What happens if you override equals() but not hashCode() in a HashMap key?**

**A)** Everything works fine  
**B)** Duplicate keys will appear  
**C)** Key lookups will fail  
**D)** Runtime exception  
✅ **Answer: C**  
**Explanation:** hashCode() determines the **bucket**; mismatch between equals() and hashCode() breaks lookups.

**Q15. Which is true about Java's HashSet?**

**A)** Allows duplicates  
**B)** Maintains order  
**C)** Implements Set using HashMap internally  
**D)** Is thread-safe  
✅ **Answer: C**  
**Explanation:** A HashSet is backed by a **HashMap** where values are dummy constants.

**Q16. Which of the following correctly initializes a custom max-heap in Java?**

PriorityQueue<Integer> pq = new PriorityQueue<>(Collections.reverseOrder());

**A)** Valid  
**B)** Invalid  
**C)** Only works for Strings  
**D)** Throws exception  
✅ **Answer: A**  
**Explanation:** This creates a **max-heap** by using a **reversed comparator**.

**Q17. What is the best use case for a Trie?**

**A)** Sorting integers  
**B)** Maintaining database connections  
**C)** Prefix auto-complete  
**D)** Graph traversal  
✅ **Answer: C**  
**Explanation:** Tries are ideal for **prefix-based searches** and suggestions like autocomplete.

**Q18. How does Java handle hash collisions in a HashMap after Java 8?**

**A)** Linked list chaining  
**B)** Binary tree for large buckets  
**C)** Discards duplicates  
**D)** Rehashes the map  
✅ **Answer: B**  
**Explanation:** From **Java 8**, if a bucket exceeds threshold size, it is converted into a **balanced tree** (TreeNode) for better performance.

**Q19. Which data structure ensures O(log n) time for both insert and delete?**

**A)** HashMap  
**B)** ArrayList  
**C)** TreeMap  
**D)** Stack  
✅ **Answer: C**  
**Explanation:** TreeMap operations (put, remove) are based on **Red-Black Tree** → O(log n).

**Q20. Which of these would most efficiently implement a top-k frequent elements problem?**

**A)** Array  
**B)** TreeSet  
**C)** HashMap + MinHeap  
**D)** LinkedList  
✅ **Answer: C**  
**Explanation:** Use HashMap to count frequencies and a MinHeap to track top-k efficiently.

**Q21. What will be the result of inserting duplicate keys in a HashMap?**

**A)** Adds both  
**B)** Skips duplicate  
**C)** Replaces old value  
**D)** Throws exception  
✅ **Answer: C**  
**Explanation:** HashMap.put(k, v) **replaces** the old value if the key already exists.

**Q22. What is the worst-case time complexity of HashMap get()?**

**A)** O(1)  
**B)** O(n)  
**C)** O(log n)  
**D)** O(n log n)  
✅ **Answer: B**  
**Explanation:** In **worst case** (all keys hash to the same bucket), it degrades to **O(n)**.

**Q23. In a Trie, how many children can a node have in a lowercase alphabet Trie?**

**A)** 10  
**B)** 26  
**C)** 52  
**D)** 128  
✅ **Answer: B**  
**Explanation:** For lowercase English, each node has **26 possible children** (a–z).

**Q24. What kind of tree is used in Java’s TreeMap?**

**A)** AVL Tree  
**B)** Binary Search Tree  
**C)** Red-Black Tree  
**D)** B+ Tree  
✅ **Answer: C**  
**Explanation:** Java’s TreeMap is implemented using a **Red-Black Tree**.

**Q25. What’s the primary difference between HashMap and ConcurrentHashMap?**

**A)** Speed  
**B)** Thread safety  
**C)** Sorting  
**D)** Size limit  
✅ **Answer: B**  
**Explanation:** ConcurrentHashMap is designed for **thread-safe** concurrent access.

**Q26. What’s the effect of a low load factor (e.g. 0.2) in a HashMap?**

**A)** Saves memory  
**B)** Increases collisions  
**C)** Increases rehashing frequency  
**D)** Avoids rehashing  
✅ **Answer: C**  
**Explanation:** Lower load factor → triggers **rehashing** sooner → higher memory usage.

**Q27. Which method retrieves but does not remove the head of a PriorityQueue?**

**A)** poll()  
**B)** remove()  
**C)** peek()  
**D)** pop()  
✅ **Answer: C**  
**Explanation:** peek() returns the head **without removing** it.

**Q28. In a Trie, how is end of word typically marked?**

**A)** By null pointer  
**B)** With an index  
**C)** With a boolean flag  
**D)** With character ‘\0’  
✅ **Answer: C**  
**Explanation:** A node usually has a flag like isEndOfWord set to **true**.

**Q29. In HashMap<K, V>, which generic types can be used?**

**A)** Only Integer  
**B)** Only String  
**C)** Any Object with proper equals() and hashCode()  
**D)** Only primitive types  
✅ **Answer: C**  
**Explanation:** Any class can be used as a key **if it correctly implements equals() and hashCode()**.

**Q30. What is the main disadvantage of a Trie?**

**A)** Time complexity  
**B)** Memory consumption  
**C)** No search capability  
**D)** Limited to numbers only  
✅ **Answer: B**  
**Explanation:** Tries are **memory-intensive** as each character node adds new branches, even for short strings.

**Advanced DSA Patterns and Techniques in Java:**

**Q1. What is the time complexity to find max element in all subarrays of size k using a deque?**

**A)** O(k²)  
**B)** O(nk)  
**C)** O(n)  
**D)** O(n log k)  
✅ **Answer: C**  
**Explanation:** Using a **monotonic deque**, we can solve the sliding window maximum in **O(n)** time.

**Q2. What is the key idea behind the sliding window technique?**

**A)** Iterate from both ends  
**B)** Expand and shrink window boundaries  
**C)** Use recursion  
**D)** Hashing only  
✅ **Answer: B**  
**Explanation:** Sliding window involves maintaining a **window (subarray)** and **adjusting boundaries** efficiently.

**Q3. Which technique solves this: “Find longest substring without repeating characters”?**

**A)** Binary search  
**B)** Prefix sum  
**C)** Sliding window with HashSet  
**D)** DFS  
✅ **Answer: C**  
**Explanation:** Use **HashSet + sliding window** to keep track of seen characters efficiently.

**Q4. What is a good problem for two-pointer technique?**

**A)** Inorder traversal  
**B)** Merge sorted arrays  
**C)** Depth calculation  
**D)** Finding divisors  
✅ **Answer: B**  
**Explanation:** Two-pointer works well when merging or comparing **two sorted lists or subarrays**.

**Q5. What is the core idea behind monotonic stacks?**

**A)** Sort stack  
**B)** Stack maintains increasing/decreasing order  
**C)** Balanced brackets  
**D)** Fibonacci generator  
✅ **Answer: B**  
**Explanation:** A **monotonic stack** keeps elements in strictly **increasing or decreasing** order to solve next greater/smaller problems.

**Q6. Which technique is used in "Trapping Rain Water" problem?**

**A)** Binary Search  
**B)** Monotonic Stack  
**C)** Prefix Sum + Two-pointer  
**D)** DFS  
✅ **Answer: C**  
**Explanation:** You precompute **leftMax/rightMax arrays** (prefix max) and use two pointers to accumulate trapped water.

**Q7. What is the time complexity of Union-Find (DSU) with path compression?**

**A)** O(n)  
**B)** O(log n)  
**C)** O(α(n))  
**D)** O(n²)  
✅ **Answer: C**  
**Explanation:** DSU with **path compression + union by rank** gives near-constant time: **O(α(n))**, where α(n) is the inverse Ackermann function.

**Q8. What data structure is used for range queries (sum, min) in log(n) time?**

**A)** Queue  
**B)** Stack  
**C)** Segment Tree  
**D)** HashMap  
✅ **Answer: C**  
**Explanation:** **Segment trees** support range queries and point updates in **O(log n)**.

**Q9. Which operation is fast with prefix sum?**

**A)** Point update  
**B)** Range sum query  
**C)** Sorting  
**D)** DFS  
✅ **Answer: B**  
**Explanation:** Prefix sum allows **O(1)** range sum queries after **O(n)** preprocessing.

**Q10. In Java, which bitwise operation can isolate the rightmost set bit?**

**A)** x & (x + 1)  
**B)** x | (x - 1)  
**C)** x & -x  
**D)** ~x  
✅ **Answer: C**  
**Explanation:** x & -x isolates the **least significant bit that is set to 1**

**Q11. What will the following code output?**

int[] arr = {1, 2, 3, 4, 5};

int[] prefix = new int[arr.length + 1];

for (int i = 0; i < arr.length; i++) {

prefix[i + 1] = prefix[i] + arr[i];

}

System.out.println(prefix[3] - prefix[1]);

**A)** 3  
**B)** 4  
**C)** 5  
**D)** 6  
✅ **Answer: D**  
**Explanation:** Sum of arr[1] to arr[2] → 2 + 3 = 5. prefix[3] - prefix[1] = 6 - 1 = 5. Correct sum is 5.  
Correction: **Answer is C** (Typo fixed in explanation).

**Q12. Which bitwise expression clears the rightmost set bit in a number x?**

**A)** x & (x - 1)  
**B)** x | (x - 1)  
**C)** x ^ (x - 1)  
**D)** x & -x  
✅ **Answer: A**  
**Explanation:** x & (x - 1) removes the lowest set bit.

**Q13. In a Union-Find structure, which line causes path compression?**

java

CopyEdit

int find(int x) {

if (parent[x] != x)

parent[x] = find(parent[x]);

return parent[x];

}

**A)** Line 1  
**B)** Line 2 (parent[x] = find(parent[x]);)  
**C)** return statement  
**D)** No compression is done  
✅ **Answer: B**  
**Explanation:** This line compresses the path by pointing x directly to the root.

**Q14. You are given a binary search problem where values are not sorted but constraints apply (e.g., capacity, time). Which approach applies?**

**A)** Binary search on array  
**B)** Binary search on index  
**C)** Binary search on answer  
**D)** Brute force  
✅ **Answer: C**  
**Explanation:** In problems like “minimum max weight,” we apply **binary search on the answer space**.

**Q15. Which of the following solves “minimum number of platforms needed” in train problem?**

**A)** Greedy sorting + two pointers  
**B)** Stack  
**C)** Prefix sum  
**D)** Bitmask  
✅ **Answer: A**  
**Explanation:** Sort arrival and departure → use two pointers to track platform usage.

**Q16. What’s the time complexity to build a segment tree for n elements?**

**A)** O(n²)  
**B)** O(n)  
**C)** O(n log n)  
**D)** O(log n)  
✅ **Answer: C**  
**Explanation:** Each level has **O(n)** total work across levels → **log n levels**.

**Q17. What is the time complexity for range minimum query on a segment tree?**

**A)** O(log n)  
**B)** O(n)  
**C)** O(1)  
**D)** O(n log n)  
✅ **Answer: A**  
**Explanation:** Segment tree queries take **O(log n)** time.

**Q18. In Java, what’s the result of 3 ^ 3 ^ 2?**

**A)** 2  
**B)** 0  
**C)** 3  
**D)** 1  
✅ **Answer: A**  
**Explanation:** XOR is associative: 3 ^ 3 = 0; then 0 ^ 2 = 2.

**Q19. Which problem can be solved using bitmasking?**

**A)** Subset sum  
**B)** N-Queens  
**C)** Sudoku  
**D)** All of the above  
✅ **Answer: D**  
**Explanation:** Bitmasking efficiently represents and manipulates states in **backtracking & subsets**.

**Q20. What is the space complexity of DSU with parent and rank arrays?**

**A)** O(n²)  
**B)** O(1)  
**C)** O(n)  
**D)** O(log n)  
✅ **Answer: C**  
**Explanation:** Arrays of size n → **O(n)** space.

**Q21. What will this code return for nums = [2,3,1,2,4,3], target = 7?**

int left = 0, sum = 0, minLen = Integer.MAX\_VALUE;

for (int right = 0; right < nums.length; right++) {

sum += nums[right];

while (sum >= target) {

minLen = Math.min(minLen, right - left + 1);

sum -= nums[left++];

}

}

**A)** 2  
**B)** 3  
**C)** 4  
**D)** 1  
✅ **Answer: A**  
**Explanation:** Sliding window finds minimal subarray [4,3] with sum ≥ 7 → length = 2.

**Q22. What technique does this represent?**

int max = 0;

Set<Character> set = new HashSet<>();

int l = 0;

for (int r = 0; r < s.length(); r++) {

while (set.contains(s.charAt(r))) {

set.remove(s.charAt(l++));

}

set.add(s.charAt(r));

max = Math.max(max, r - l + 1);

}

**A)** Binary search  
**B)** DFS  
**C)** Sliding window  
**D)** Monotonic queue  
✅ **Answer: C**  
**Explanation:** It's a classic **sliding window** technique for longest substring without repetition.

**Q23. In a bitmask of n bits, how many total subsets can be generated?**

**A)** n  
**B)** 2n  
**C)** n²  
**D)** 2ⁿ  
✅ **Answer: D**  
**Explanation:** Each bit can be on/off → **2ⁿ** subsets.

**Q24. What’s a real-world usage of segment tree?**

**A)** Hotel booking range overlaps  
**B)** Matrix multiplication  
**C)** Reversing a list  
**D)** Hashing user data  
✅ **Answer: A**  
**Explanation:** Segment trees are ideal for **range tracking** like reservations or schedules.

**Q25. What does this expression return: Integer.bitCount(7)?**

**A)** 1  
**B)** 2  
**C)** 3  
**D)** 4  
✅ **Answer: C**  
**Explanation:** 7 = 111 in binary → **3 set bits**.

**Q26. What’s the use of >> and >>> in Java?**

**A)** Shift left  
**B)** Rotate bits  
**C)** Signed and unsigned right shift  
**D)** Remove digits  
✅ **Answer: C**  
**Explanation:** >> is **signed**, >>> is **unsigned right shift**.

**Q27. How do you efficiently find a duplicate in an array of n+1 numbers where all are 1 to n?**

**A)** HashSet  
**B)** Brute force  
**C)** Floyd’s cycle detection  
**D)** Sorting  
✅ **Answer: C**  
**Explanation:** Duplicate creates a cycle → use **Floyd’s Tortoise & Hare**.

**Q28. In problems like “Allocate Minimum Pages” or “Koko Eating Bananas,” what strategy is used?**

**A)** Recursion  
**B)** Two pointer  
**C)** Binary Search on Answer  
**D)** Hashing  
✅ **Answer: C**  
**Explanation:** These are classic **binary search on range/answer** problems.

**Q29. Which problem would monotonic stack solve efficiently?**

**A)** Valid parentheses  
**B)** Largest Rectangle in Histogram  
**C)** Merge intervals  
**D)** Trie lookups  
✅ **Answer: B**  
**Explanation:** Monotonic stack tracks increasing bar heights.

**Q30. What is the value of 1 << 3 in Java?**

**A)** 3  
**B)** 8  
**C)** 6  
**D)** 1  
✅ **Answer: B**  
**Explanation:** 1 << 3 means shift 1 three bits to the left → 1000 in binary = **8**.