Interview Questions and Answers for the Generalised Data Structures and Algorithms Library Project

Section 1: Core Data Structures (Basic to Intermediate)

Q1. What is the primary purpose of the QueueX and StackX classes in this project?

Answer: `QueueX` implements a **First-In, First-Out (FIFO) data structure, primarily used for managing resources or tasks in a sequence. `StackX` implements a **Last-In, First-Out (LIFO)** data structure, typically used for function call management (recursion) and expression evaluation.**

Q2. How is the Queue (QueueX) implemented in the code, and what is the time complexity of the enqueue() operation?

Answer: The Queue is implemented using a **Singly Linked List (SLL). The `enqueue()` operation involves traversing the list to find the last node, making its time complexity **O(n)** (where n is the number of elements), which is inefficient for a queue.**

Q3. In StackX, where are new elements inserted (push), and what is the time complexity for push()?

Answer: New elements are inserted at the **beginning of the linked list (`first` node). This is a constant-time operation, so its complexity is **O(1)**.**

Q4. What is the fundamental difference between a SinglyLL and a DoublyLL implementation?

Answer: **SinglyLL nodes only have a `next` pointer, allowing traversal in one direction. **DoublyLL** nodes have both a `next` and a `prev` pointer, allowing for **bidirectional traversal** and **O(1)** complexity for `deleteLast()` (which is O(n) in SLL).**

Q5. Explain the concept of a Circular Linked List as implemented in SinglyCLL and DoublyCLL.

Answer: In a Circular Linked List, the **last node's `next` pointer points back to the **first** node, forming a closed loop. This is useful for circular buffers or round-robin scheduling.**

Q6. Why are the Linked List classes constrained with <T extends Number>?

Answer: The constraint is necessary because these lists contain "Extra Logic" methods like `displayPerfect()`, `displayPrime()`, `sumOfDigits()`, and `reverseDigits()`, which require calling **intValue() on the generic data type T.**

Q7. What problem could arise if you try to dequeue() an element from an empty QueueX? How does your code handle this?

Answer: It would result in a **Null Pointer Exception (NPE) if not handled. The code prevents this by checking `if (first == null)` and then throwing a **NoSuchElementException**.**

Section 2: Non-Linear Structures and Algorithms (Intermediate)

Q8. What is a Binary Search Tree (BST), and why is the BST class constrained with <T extends Comparable<T>>?

Answer: A BST is a binary tree where the value of every node in the left subtree is less than the node's value, and the value of every node in the right subtree is greater. The constraint is vital because BST operations like `insert` and `search` require the elements to be **comparable using the `compareTo()` method.**

Q9. Name and describe the three traversal methods implemented in the BST class.

Answer: **Inorder: (Left → Root → Right) - Visits nodes in **sorted order**.
Preorder: (Root → Left → Right) - Useful for **copying** or recreating the tree structure.
Postorder: (Left → Right → Root) - Useful for **deleting** the tree (root is deleted last).**

Q10. What is the worst-case time complexity for insert() and search() operations in the implemented BST?

Answer: The worst-case occurs when the data is inserted in sorted (or reverse sorted) order, creating a **skewed tree that resembles a linked list. In this case, the complexity degrades to **O(n)**.**

Q11. Describe the working principle of Bubble Sort (Sorting.bubbleSort).

Answer: Bubble Sort repeatedly steps through the list, compares adjacent elements, and **swaps them if they are in the wrong order. It continues until no swaps are needed in a pass, effectively "bubbling" the largest elements to the end.**

Q12. Describe the working principle of Selection Sort (Sorting.selectionSort).

Answer: Selection Sort divides the input list into two parts: the sorted and the unsorted. It **finds the minimum element from the unsorted part and **swaps** it with the element at the beginning of the unsorted part.**

Q13. What is the time complexity of Bubble Sort and Selection Sort, and why are they considered inefficient for large datasets?

Answer: Both have a worst-case and average-case time complexity of **O(n^2). They are inefficient because the number of comparisons and swaps grows quadratically with the size of the input n.**

Q14. When should Linear Search be preferred over Binary Search?

Answer: Linear Search is preferred when the array is **unsorted, or when the array is relatively **small**. Binary Search requires the array to be sorted, and the cost of sorting an unsorted array often outweighs the benefit of O(log n) search time.**

Q15. What is the prerequisite for running Searching.binarySearch() effectively, and what is its time complexity?

Answer: The input array **must be sorted. Its time complexity is **O(log n)** because it repeatedly divides the search interval in half.**

Section 3: Design, Code Quality, and Advanced Concepts

Q16. The enqueue() in QueueX is O(n). How could you modify the implementation to achieve the optimal O(1) complexity?

Answer: By tracking both the **first and the **last** nodes (like the Doubly Linked Lists do). `enqueue` would then directly use the `last` pointer to append the new node, making it O(1).**

Q17. Explain the logic implemented in the isPerfect() helper method.

Answer: A **Perfect Number is a positive integer that is equal to the sum of its proper positive divisors (divisors excluding the number itself). The method calculates the sum of all divisors from 1 up to no/2 and returns true if this sum equals no.**

Q18. Why does the isPrime() helper method only iterate up to no/2?

Answer: If a number no has a divisor greater than no/2, it must also have a divisor less than 2, which is not possible for an integer. The check up to no/2 is sufficient, although iterating only up to √no is a more optimal O(√n) approach.

Q19. Analyze the complexity of the sumOfDigits() method in the Linked Lists.

Answer: If N is the number of nodes and D is the maximum number of digits in any element, the complexity is **O(N · D). Since D is proportional to log10(Value), the complexity can also be expressed as O(N · log(Max Value)).**

Q20. What is the purpose of the inner class Node in all your data structures?

Answer: The inner class **Node encapsulates the data and the references (next/prev) that define the structure. Making it a private inner class ensures it is only used internally by the data structure class, promoting **encapsulation** and cleaner object-oriented design.**

Q21. In binarySearch(), why is the middle index calculated as mid = left + (right − left) / 2 instead of mid = (left + right) / 2?

Answer: The expression mid = (left + right) / 2 can cause **integer overflow if both left and right are large positive integers (close to Integer.MAX_VALUE). The form mid = left + (right − left) / 2 mitigates this risk by keeping the intermediate calculation smaller.**

Q22. Why do SinglyCLL and DoublyCLL not have explicit deleteFirst() and deleteLast() methods in the provided code (unlike the linear lists)?

Answer: While they can be implemented, deletion in circular lists is more complex, especially for deleteLast(), as it requires locating the node *before* the last node to update its next pointer, which can be an O(n) operation in a Singly CLL.

Q23. The reverseDigits() method modifies the list. What is the type safety consideration when doing temp.data = (T) Integer.valueOf(rev)?

Answer: This involves an **unchecked cast. Since the list is constrained to T extends Number and the operation converts the data to an Integer, the cast is generally safe in this context, but in broader generic programming, unchecked casts can lead to runtime errors.**

Q24. How would you enhance the Sorting class to handle generic types instead of just int[]?

Answer: You would need to: 1. Make the methods generic: `public static <T extends Comparable<T>> void bubbleSort(T[] arr)`. 2. Use arr[j].compareTo(arr[j+1]) > 0 instead of arr[j] > arr[j+1] for comparisons.

Q25. Discuss the memory usage difference between a node in SinglyLL vs. a node in DoublyCLL.

Answer: A node in DoublyCLL uses **one extra reference (pointer) (prev) compared to a node in SinglyLL. This overhead means DoublyCLL uses more memory per node but gains flexibility and performance improvements in certain operations (like O(1) deleteLast).**

Q26. What is the significance of the insertRec method being private in the BST class?

Answer: The insertRec method is a **recursive helper function that handles the actual node insertion logic. It is made private because users of the BST class should only call the public, simpler insert(T value) method, which starts the recursion from the root. This adheres to the principle of **information hiding**.**

Q27. How does the DoublyCLL.insertFirst() implementation maintain the circular property?

Answer: After inserting the new node and updating first, it ensures the links between first and last are correct: last.next = first and first.prev = last. This maintains the two-way circularity.

Q28. If you were to implement a delete() method in BST, what would be the three main cases you would need to handle?

Answer: 1. **Node to be deleted is a leaf (no children).**
2. **Node has one child.**
3. **Node has two children (requires finding the Inorder Successor or Predecessor to replace the deleted node).**

Q29. Why is the Generalised_Data_Structures_library class defined as public while all other classes are package-private (default access)?

Answer: Only the class containing the **main method (the entry point for execution) is typically required to be public to match the file name. The other classes are designed as internal components of the library and don't need to be accessible outside the package.**

Q30. If you had to use this library for a new project, which data structure would you choose for implementing a simple, fixed-size cache (where the oldest item is removed when full)?

Answer: I would choose a **Doubly Linked List (DoublyLL) combined with a **HashMap**. The DoublyLL maintains the insertion/access order (oldest at first, newest at last), while the HashMap provides O(1) lookups to quickly find and move an existing item to the last position (LRU cache principle).**