Unit 1 Short Answers :

1. **What are Generics? (L1):** Generics in Java allow you to create classes, interfaces, and methods that operate on different data types without sacrificing type safety. They provide a way to parameterize types, allowing you to write code that can be reused with different data types.

2. **Benefits of Generics (L2):** Generics offer several benefits:

   - **Type Safety:** Generics help catch type-related errors at compile time, reducing runtime errors.

   - **Code Reusability:** You can write classes and methods that work with a variety of data types.

   - **Elimination of Casts:** With generics, you avoid the need for casting objects, improving code readability and performance.

   - **Compile-Time Checks:** Generics enable the compiler to perform type checks, providing early feedback on errors.

3. **What are Type Parameters? (L1):** Type parameters are placeholders for actual data types that you can use when defining generic classes, methods, or interfaces. They allow you to create generic code that can work with different data types.

4. **Demonstrate Restriction on Generics (L3):** Generics cannot be used with primitive types directly. For example, you cannot create a generic array of primitive types like `List<int>`. Instead, you need to use the corresponding wrapper classes, such as `List<Integer>`.

5. **Type Inference in Generics (L3):** Type inference allows the compiler to automatically determine the type argument based on the context in which a generic method or constructor is invoked. For example, when calling a generic method like `Collections.emptyList()`, the compiler infers the type argument based on the expected return type.

6. **Type Erasure in Generics (L3):** The compiler uses type erasure to replace generic type information with type bounds or Object types during compilation. This allows Java's generics to provide type safety at compile time while avoiding additional runtime overhead.

7. **Syntax of Generic Class (L1):**

```java
class ClassName<T> {
```

```
    // Class definition with type parameter T
}
```

8. **Syntax of Generic Methods (L1):**

```java
class ClassName {
    <T> ReturnType methodName(T parameter) {
        // Method implementation with type parameter T
    }
}
```

9. **Syntax of Generic Interfaces (L1):**

```java
interface InterfaceName<T> {
    // Interface definition with type parameter T
}
```

10. **Comparison of Raw Types with Generic Types (L6):** Raw types (non-generic) do not provide type safety and may lead to runtime errors. Generic types offer better type safety and allow the compiler to catch errors at compile time. For example, using a raw type like `List` instead of a generic type like `List<Integer>` can result in unchecked warnings and potential errors.

11. **Applications of Wildcard Arguments (L2):** Wildcards (`?`) in generics are used to provide flexibility when working with unknown types. They are commonly used in scenarios like:

- Accepting a collection of unknown types in a method.
- Specifying method parameters that can handle different generic types.
- Defining methods that can operate on collections of different generic types.

12. **Define Bounded Types (L1):** Bounded types in generics allow you to restrict the types that can be used as type arguments. You can specify upper or lower bounds to ensure that the type parameter is a subclass or superclass of a certain type. For example, `<? extends Number>` represents a bounded type that includes subclasses of `Number`, while `<? super T>` represents a bounded type that includes superclasses of `T`.

Unit 2 Short Answers :

1. **Java Collections Framework and Benefits (L3):** The Java Collections Framework is a set of interfaces, classes, and algorithms provided by Java to represent and manipulate collections of objects. It provides a unified and standardized way to work with collections, offering benefits such as:

- Reusability: Pre-defined data structures and algorithms for common collection operations.
- Type Safety: Strongly typed collections help catch type-related errors at compile time.
- Efficiency: Optimized implementations of data structures for efficient storage and access.
- Interoperability: Collections can be easily interchanged and used across different APIs.
- Scalability: Allows handling large amounts of data effectively.
- Standardization: Common interfaces and conventions for consistent programming.
- Extensibility: Custom implementations can be created to suit specific needs.

2. **Using the List Interface (L3):** The List interface in Java Collections Framework represents an ordered collection of elements, allowing duplicates and preserving insertion order. You can use the List interface as follows:

- Create an instance of a class that implements the List interface, such as `ArrayList`, `LinkedList`, or `Vector`.
- Add elements to the list using the `add` method.
- Access elements using their index with the `get` method.
- Iterate over elements using loops or iterators.
- Perform various operations like removing elements, searching, sorting, etc.

3. **Main Idea of Collection Set (L2):** The main idea of a collection Set is to represent a group of distinct elements with no duplicates. Set interfaces like `Set` and `SortedSet` provide methods for adding, removing, checking existence, and performing set operations like union, intersection, and difference.

4. **Main Idea of Iterator Interface (L2):** The main idea of the Iterator interface is to provide a way to traverse elements of a collection sequentially without exposing the underlying details of the collection's implementation. It offers methods like `hasNext` to check if there are more elements and `next` to retrieve the next element.

5. **Classification of Interfaces in Java Collections Framework (L4):** The interfaces in Java Collections Framework are classified into three main categories:

- Collection Interfaces: Represent various types of collections, such as lists, sets, and queues. Examples: `List`, `Set`, `Queue`.

- Map Interfaces: Represent key-value pair associations. Examples: `Map`, `SortedMap`.

- Utility Interfaces: Provide utility methods for various collection operations. Examples: `Iterable`, `Comparator`.

6. **Difference Between HashMap and Hashtable (L2):**

- HashMap is not synchronized and not thread-safe, while Hashtable is synchronized and thread-safe.

- HashMap allows null keys and values, while Hashtable does not allow null keys or values.

- HashMap is generally faster than Hashtable due to lack of synchronization.

- HashMap is part of the Java Collections Framework, while Hashtable is a legacy class.

7. **Similarities and Differences Between ArrayList and Vector (L2):**

- Similarities: Both implement the List interface, allow duplicates, and maintain insertion order.

- Differences:

  - Vector is synchronized and thread-safe, while ArrayList is not.

  - Vector is slower due to synchronization, while ArrayList is faster.

  - ArrayList is part of the Java Collections Framework, while Vector is a legacy class.

8. **Reason Collection Interface Doesn't Extend Cloneable and Serializable (L4):** The Collection interface does not extend `Cloneable` and `Serializable` interfaces because not all collections need to support cloning and serialization. Adding these interfaces would impose unnecessary requirements on all implementing classes.

9. **Reason Map Interface Doesn't Extend Collection Interface (L4):** The Map interface does not extend the Collection interface because maps represent associations between

keys and values, while collections represent groups of elements. The semantics and operations of maps and collections are significantly different.

10. **Difference Between Iterator and ListIterator (L2):**

- Iterator: Used to traverse elements in a forward direction for any collection. It has limited functionality compared to ListIterator.

- ListIterator: Extends Iterator and is specific to lists. It allows bidirectional traversal, element modification, and more operations.

11. **Basic Idea of Map Interface (L2):** The Map interface represents a collection of key-value pairs where each key is associated with a unique value. It allows efficient retrieval, insertion, and removal of values based on their keys.

12. **Difference Between HashSet and LinkedHashSet (L2):**

- HashSet: Does not maintain any specific order of elements. Provides constant-time average for basic operations.

- LinkedHashSet: Maintains insertion order of elements. Offers better performance than TreeSet for most operations.

13. **Difference Between HashMap and LinkedHashMap (L2):**

- HashMap: Does not guarantee order of keys or values. Provides constant-time average for basic operations.

- LinkedHashMap: Maintains insertion order of keys and values. Offers better performance than TreeMap for most operations.

14. **Difference Between ArrayList and LinkedList (L2):**

- ArrayList: Backed by an array, allows fast random access and efficient traversal. Good for read-heavy operations.

- LinkedList: Implements a doubly-linked list, supports fast insertions and deletions. Good for add-remove operations.

15. **Difference Between HashMap and Hashtable (L2):**

- HashMap: Not synchronized, not thread-safe, allows null keys and values, part of the Java Collections Framework.

- Hashtable: Synchronized, thread-safe, does not allow null keys or values, a legacy class.

Unit 3 Short Questions :

1. **Define Dictionary (L1):** A dictionary, in the context of data structures, refers to a collection that stores key-value pairs. Each key is associated with a specific value, and dictionaries allow efficient retrieval, insertion, and deletion of values based on their corresponding keys.

2. **Operations on a Dictionary (L1):** The main operations on a dictionary include:

   - Insertion: Adding a new key-value pair to the dictionary.
   - Retrieval: Accessing the value associated with a specific key.
   - Deletion: Removing a key-value pair from the dictionary.
   - Update: Modifying the value associated with an existing key.

3. **When Does a Collision Occur? (L1):** A collision occurs in a hash-based data structure, such as a hash table, when two or more distinct keys produce the same hash code. This happens because hash functions map a potentially infinite set of keys to a finite set of hash codes, leading to the possibility of different keys having the same hash code.

4. **Load Factor (L2):** The load factor of a hash table is the ratio of the number of elements stored in the table to the total number of buckets (slots) available. A higher load factor can lead to more collisions, impacting performance, while a lower load factor requires more memory. Balancing the load factor helps maintain efficient hash table operations.

5. **Rehashing (L2):** Rehashing is the process of resizing a hash table by creating a new, larger array and reinserting the existing elements into the new array. It is typically performed when the load factor exceeds a certain threshold to avoid excessive collisions and maintain good hash table performance.

6. **Main Idea of Double Hashing (L2):** Double hashing is a collision resolution technique used in hash tables. It involves applying a second hash function to calculate the number of steps to probe for an available slot when a collision occurs. The goal is to find an alternative slot in a systematic manner to avoid clustering and improve insertion efficiency.

7. **Extendible Hashing (L1):** Extendible hashing is a dynamic hashing technique where the hash function is updated as the number of keys increases. It uses a directory structure to map keys to buckets, and when a collision occurs, the directory is expanded to

accommodate more buckets. This approach reduces collisions and provides efficient resizing.

8. **Drawbacks of Open Addressing in a Hash Table (L4):** Open addressing can have several drawbacks, including:

- Increased Clustering: Collisions can lead to clustering, causing subsequent insertions to encounter longer probe sequences and decreased performance.

- Deletion Challenges: Deleting elements requires careful handling to mark deleted slots, leading to complexities and potential inefficiencies.

- Load Factor Sensitivity: Open addressing's performance is highly sensitive to the load factor, which can affect its efficiency.

- Resizing Complexity: Resizing open-addressed tables requires rehashing and reinsertion, which can be computationally expensive.

9. **Advantages of Chained Hash Table over Open Addressing (L2):** Chained hash tables offer advantages such as:

- Reduced Clustering: Collisions are managed by separate linked lists, reducing clustering effects.

- Simpler Deletion: Deleting elements from a linked list is easier and does not require special handling.

- Load Factor Flexibility: Chained hash tables are less sensitive to load factor changes.

- Dynamic Resizing: Resizing involves only extending the linked lists, making resizing less complex.

10. **Define Binary Search Tree (L1):** A binary search tree (BST) is a binary tree in which each node has at most two children, typically referred to as the left child and the right child. The key value of each node is greater than all keys in its left subtree and less than all keys in its right subtree.

11. **Degenerated Tree (L1):** A degenerated tree, also known as a pathological tree, is a binary tree in which each parent node has only one child. This type of tree is essentially a linked list and can result from inserting elements in sorted order into a binary search tree.

12. **Strictly Binary Tree (L1):** A strictly binary tree is a binary tree in which each parent node has exactly two children. There are no nodes with only one child.

13. **Difference Between Fully, Strictly, Complete Binary Tree (L2):**

- Fully Binary Tree: Every node in the tree has either zero or two children.

- Strictly Binary Tree: Every node has exactly two children.

- Complete Binary Tree: All levels of the tree are fully filled, except possibly the last level, which is filled from left to right.

Unit -IV short questions :

1. **What is an AVL Tree? (L1):** An AVL tree (Adelson-Velsky and Landis tree) is a self-balancing binary search tree where the height of the two child subtrees of any node differs by at most one. This balancing property ensures that the tree remains relatively balanced, leading to efficient search, insertion, and deletion operations.

2. **List the Rotations in AVL Tree (L1):** In an AVL tree, rotations are used to maintain the balance factor of nodes during insertion or deletion. There are four types of rotations:

- Left-Left (LL) rotation

- Left-Right (LR) rotation

- Right-Right (RR) rotation

- Right-Left (RL) rotation

3. **What are Balanced Trees? (L1):** Balanced trees are binary search trees in which the height of the subtrees of any node is approximately the same. This balance ensures that the tree's height remains logarithmic, leading to efficient search, insertion, and deletion operations. AVL trees and red-black trees are examples of balanced trees.

4. **Effect of Balance Factor in AVL Tree (L2):** The balance factor of a node in an AVL tree is the difference between the heights of its left and right subtrees. The balance factor indicates whether a node's subtree is left-heavy, right-heavy, or balanced. An AVL tree maintains the balance factor close to -1, 0, or 1. When performing insertions or deletions, if a node's balance factor becomes -2 or 2, rotations are performed to restore balance and maintain the AVL tree properties.

5. **Apply RR Rotation in Insertion (L3):** The RR rotation (Right-Right rotation) is used to balance an AVL tree after performing a right insertion into the right subtree of the right child of a node. It involves a single rotation to the left. The steps are:

1. Perform a left rotation between the parent node and its right child.

2. Update the heights of the rotated nodes.

6. **Apply LL Rotation in Insertion (L3):** The LL rotation (Left-Left rotation) is used to balance an AVL tree after performing a left insertion into the left subtree of the left child of a node. It involves a single rotation to the right. The steps are:

   1. Perform a right rotation between the parent node and its left child.

   2. Update the heights of the rotated nodes.

7. **Apply RL Rotation in Insertion (L3):** The RL rotation (Right-Left rotation) is used to balance an AVL tree after performing a left insertion into the right subtree of the left child of a node. It involves two rotations: first a right rotation, followed by a left rotation. The steps are:

   1. Perform a right rotation between the left child and its right child.

   2. Perform a left rotation between the parent node and the new left child.

   3. Update the heights of the rotated nodes.

8. **Define Red-Black Search Tree (L1):** A red-black tree is a type of self-balancing binary search tree in which each node has an extra attribute, a color (red or black), to ensure that the tree remains approximately balanced during insertions and deletions. Red-black trees satisfy specific properties to maintain balanced properties.

9. **Special Property of Red-Black Trees (L1):** The special property of red-black trees is that they maintain balance by adhering to five properties:

   - Every node is either red or black.

   - The root node is black.

   - Every leaf (NIL) is black.

   - If a red node has children, they are black.

   - Every simple path from a node to a descendant leaf contains the same number of black nodes. This ensures that the tree remains balanced.

10. **What is an M-Way Search Tree? (L1):** An m-way search tree is a generalization of a binary search tree where each internal node can have up to m - 1 keys and m children. It allows for more than two branches at each node, providing a way to manage data in a multiway fashion.

11. **Define a B-Tree (L1):** A B-tree is a self-balancing, multiway search tree designed to maintain large datasets efficiently on disk or in other storage systems. It is characterized by a balance factor (often denoted as B) that determines the maximum number of children each internal node can have.

12. **Applications of a B-Tree (L3):** B-trees are used in various applications, including:

- Database management systems for indexing and efficient data retrieval.
- File systems to manage large files and directories efficiently.
- Storage systems to organize data blocks for efficient disk I/O.
- Network routers to store routing information and optimize packet forwarding.

13. **Applications of a Red-Black Tree (L3):** Red-black trees are used in a variety of applications, including:

- Implementing associative containers like sets and maps in standard libraries.
- Symbol table implementations in compilers, interpreters, and language processors.
- Memory allocators and garbage collectors in programming languages.
- File systems to manage directory structures and improve disk access efficiency.

Unit 5 Short Questions:

1. **Define a Circular Queue (L1):** A circular queue is a linear data structure that follows the First-In-First-Out (FIFO) principle, similar to a regular queue. However, in a circular queue, the last element is connected to the first element to form a circular arrangement. This enables efficient utilization of memory and allows continuous insertion and deletion operations without shifting elements.

2. **Define a Priority Queue (L1):** A priority queue is a data structure that stores elements along with their associated priorities. It allows efficient retrieval of the element with the highest (or lowest) priority. Elements are retrieved based on their priority rather than their order of insertion.

3. **Define a Dequeue (L1):** A dequeue, also known as a double-ended queue, is a linear data structure that allows insertion and deletion of elements from both ends. It supports operations like adding elements at the front or rear, removing elements from the front or rear, and peeking at the front or rear element.

4. **Define a Heap (L1):** A heap is a specialized binary tree-based data structure that satisfies the heap property. In a min-heap, the parent node's value is less than or equal to the values of its children. In a max-heap, the parent node's value is greater than or equal to the values of its children. Heaps are commonly used in priority queues and sorting algorithms.

5. **Apply KMP Failure Function on a Pattern (L3):** The Knuth-Morris-Pratt (KMP) algorithm's failure function is used to determine the length of the longest proper prefix that is also a proper suffix of a substring. It helps in efficient string matching. To apply the KMP failure function on a pattern, you preprocess the pattern to construct the failure function array.

6. **Define a Trie (L1):** A trie (pronounced "try") is a tree-like data structure used to store a dynamic set of strings, where each node represents a character and the path from the root to a node forms a string. Tries are commonly used for fast string retrieval, prefix matching, and autocomplete functionalities.

7. **Example of a Standard Trie (L2):** A standard trie for the words "cat," "dog," "car," and "cart" would have nodes for each character in these words, forming a tree structure.

8. **Define a Compressed Trie (L1):** A compressed trie, also known as a radix tree or compact prefix tree, is an optimized version of a standard trie that compresses multiple nodes with a single child into a single node. This reduces memory consumption and speeds up traversal.

9. **Represent the String "banana" Using Suffix Trie:** A suffix trie for the string "banana" would have nodes for each suffix of the string: "banana," "anana," "nana," "ana," "na," and "a."