JAVA

4.Deque :

Key Features and Methods of Java Deque

- **Creation:**
  - Deque instances can be created using classes like ArrayDeque or LinkedList.

```
Deque<Type> dequeName = new ArrayDeque<>();
Deque<Type> dequeName = new LinkedList<>();
```

- **Adding Elements:**
  - **addFirst(element), offerFirst(element): Add elements to the front of the deque.**
  - **addLast(element), offerLast(element): Add elements to the end of the deque.**
- **Accessing Elements:**
  - **getFirst(), peekFirst(): Access the first element without removing it.**
  - **getLast(), peekLast(): Access the last element without removing it.**
- **Removing Elements:**
  - **removeFirst(), pollFirst(): Remove and return the first element.**
  - **removeLast(), pollLast(): Remove and return the last element.**
- **Other Useful Methods:**
  - **isEmpty(): Check if the deque is empty.**
  - **size(): Get the number of elements in the deque.**
  - **Iteration, conversion to array, and more.**

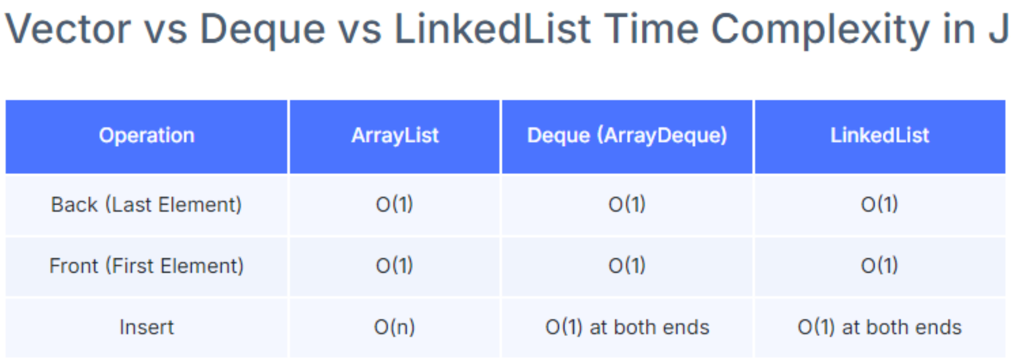
Example:

```
import java.util.Deque;
import java.util.ArrayDeque;
public class Main {
    public static void main(String[] args) {
        Deque<Integer> myDeque = new ArrayDeque<>();
        // Adding elements
        myDeque.addFirst(1);   // Add 1 to the front
        myDeque.addLast(2);    // Add 2 to the back
        myDeque.offerFirst(3); // Offer 3 to the front
        myDeque.offerLast(4);  // Offer 4 to the back
        // Accessing elements
        System.out.println("First Element: " + myDeque.getFirst());
        System.out.println("Last Element: " + myDeque.getLast());
        // Removing elements
        myDeque.removeFirst(); // Remove the first element
        myDeque.removeLast();  // Remove the last element
        // Iterating over Deque
        System.out.println("Elements in Deque:");
        for (int element : myDeque) {
            System.out.println(element);
        }
    }
}
```

- In this Java example, an ArrayDeque is used to demonstrate the basic operations of a deque, such as adding, accessing, and removing elements from both ends.
- The ArrayDeque class is a resizable-array implementation of the Deque interface, offering a convenient way to use double-ended queues in Java.

5.List in Java :

- In Java, the closest equivalent to the C++ List (a doubly-linked list) is the LinkedList class, which is part of the Java Collections Framework.
- LinkedList in Java allows for efficient insertion and deletion of elements anywhere within the list.



Vector vs Deque vs LinkedList Time Complexity in Java

Operation	ArrayList	Deque (ArrayDeque)	LinkedList
Back (Last Element)	O(1)	O(1)	O(1)
Front (First Element)	O(1)	O(1)	O(1)
Insert	O(n)	O(1) at both ends	O(1) at both ends

Basic Operations :

```
import java.util.LinkedList;
class Example {
    public static void main(String[] args) {
        LinkedList<Integer> list1 = new LinkedList<>();
        list1.addLast(1);  // Adds 1 at the back of the list
        list1.addLast(2);  // Adds 2 at the back of the list
        list1.addFirst(3);  // Adds 3 at the front of the list
        list1.addFirst(4);  // Adds 4 at the front of the list
        System.out.println(list1.get(0));
        list1.removeLast();  // Removes the last element of the list
        list1.removeFirst();  // Removes the first element of the list
        System.out.println("First Element: " + list1.getLast());
        System.out.println("Last Element: " + list1.getFirst());
    }
}
```

- **LinkedList<Integer> list1**: Declare a LinkedList named list1 to store integers.
- **list1.addLast(element)**: Adds element to the back of list1.
- **list1.addFirst(element)**: Adds element to the front of list1.
- **list1.get()**: Accessing elements by index is allowed in Java's LinkedList.
- **list1.removeLast()**: Remove the last element from list1.
- **list1.removeFirst()**: Remove the first element from list1.
- **list1.getLast()**: Get the last element of list1.
- **list1.getFirst()**: Get the first element of list1.

6.Stack :

A stack, follows Last-In, First-Out (LIFO) principle meaning that the last element added to the stack is the first one to be removed, is a linear data structure with fundamental operations including push, pop, and auxiliary functions like top, size, empty, and swap.

Real Life Use Case

- In the analogy of kitchen plates, pushing washes and adds a plate to the stack, while popping retrieves the top plate for use.
- "Empty" checks if plates remain, "top" shows the top plate, and "size" indicates the stack's plate count.
- For exchanging stacks, "swap" is employed.Java Stack Class
- In Java, the stack functionality can be achieved using the Stack class from the java.util package.
- A Stack in Java follows the Last-In, First-Out (LIFO) principle, where the last element added to the stack is the first one to be removed.
- This class provides methods for standard stack operations like push, pop, peek, and auxiliary functions like empty, search, and more.Key Features and Methods of Java Stack

- **Creation:**
  - A Stack can be created as an instance of the Stack class.

  ```
  Stack<Type> stackName = new Stack<>();
  ```

- **Adding Elements:**

   - **push(element)**: Pushes an element onto the top of the stack.

- **Removing Elements:**
   - **pop()**: Removes and returns the top element of the stack.
- **Accessing the Top Element:**
   - **peek()**: Looks at the top element of the stack without removing it.
- **Checking if Stack is Empty:**
   - **empty()**: Tests if the stack is empty.
- **Searching for Elements:**
   - **search(element)**: Returns the 1-based position of the element from the top of the stack.
- **Size of the Stack:**
   - While there's no direct **size()** method, you can use **stackName.size()** inherited from **Vector** class to get the stack size.

Example:

```
import java.util.Stack;
public class Main {
   public static void main(String[] args) {
      Stack<Integer> stk = new Stack<>();
      // Push elements onto the stack
      stk.push(10);
      stk.push(20);
      stk.push(30);
      // Display the top element
      System.out.println("Top element: " + stk.peek()); // 30
      // Pop an element from the stack
      stk.pop();
      // Check if the stack is empty
      if (stk.empty()) {
         System.out.println("Stack is empty.");
      } else {
         System.out.println("Stack is not empty."); // Not Empty
      }
      // Display the size of the stack
      System.out.println("Size of stack: " + stk.size()); // 2
      // Search for an element
      int position = stk.search(10);
      System.out.println("Position of 10: " + position); // 2 (1-based index)
```

```
    }
}
```

7.Queue

- In Java, the functionality similar to C++'s STL Queue is provided by the Queue interface in the java.util package.
- A Queue in Java follows the First-In-First-Out (FIFO) principle, where the first element added to the queue is the first to be removed.
- The Queue interface is typically implemented by classes like LinkedList, ArrayDeque, and PriorityQueue.

Key Features and Methods of Java Queue

- **Creation:**
  - A Queue instance is typically created as a LinkedList or ArrayDeque.

```
Queue<Type> queueName = new LinkedList<>()
Queue<Type> queueName = new ArrayDeque<>()
```

- **Adding Elements:**
  - **offer(element)**: Adds an element to the back of the queue. Preferred over add(element) as it returns false instead of throwing an exception for capacity-restricted queues.
  - **add(element)**: Also adds an element to the queue.
- **Removing Elements:**
  - **poll()**: Removes and returns the head of the queue, or returns Null if the queue is empty.
  - **remove()**: Similar to poll(), but throws an exception if the queue is empty.
- **Accessing Elements:**
  - **peek()**: Retrieves, but does not remove, the head of the queue, returning null if the queue is empty.
  - **element()**: Similar to **peek()**, but throws an exception if the queue is empty.
- **Other Useful Operations:**
  - **size()**: Returns the number of elements in the queue.
  - **isEmpty()**: Checks if the queue is empty.

Example:

```
import java.util.Queue;
 import java.util.LinkedList;
public class Main {
    public static void main(String[] args) {
        Queue<Integer> myQueue = new LinkedList<>();
        // Adding elements to the queue
        myQueue.offer(10); // {10}
        myQueue.offer(20); // {10, 20}
        myQueue.offer(30); // front -> {10, 20, 30} -> back
        // Accessing front and back elements
        System.out.println("Front element: " + myQueue.peek());  // 10
        // (Note: In Queue, back element access is not directly supported)
        // Removing the front element
        myQueue.poll();
        System.out.println("After polling, front element: " + myQueue.peek()); // 20
        // Queue size and empty check
        System.out.println("Queue size: " + myQueue.size()); // 2
       System.out.println("Is the queue empty? " + myQueue.isEmpty()); // false
    }
}}
```

In this Java example, a LinkedList is used to demonstrate basic queue operations like offering (adding), polling (removing), and peeking(accessing the front element).

- Unlike C++'s queue, Java's Queue interface does not directly support accessing the back element, and the implementation class (like LinkedList or ArrayDeque) determines specific behaviors and performance characteristics.

8. Priority Queue in Java :

- In Java, a priority queue is a container for elements that are arranged in a specific priority order.
- By default, the highest-priority element is at the front of the queue.
- Priority queues are often implemented as binary heaps, which offer efficient operations to maintain the highest-priority element at the top.

Max Heap or Max Priority Queue in Java

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());
        // Push elements into the max heap priority queue
        maxHeap.add(30); // maxHeap = {30}
        maxHeap.add(10); // maxHeap = {30, 10}
        maxHeap.add(50); // maxHeap = {50, 30, 10}
        maxHeap.add(20); // maxHeap = {50, 30, 20, 10}
        // Print the top element (element with the highest priority) without removing it
        System.out.println("Top element: " + maxHeap.peek()); // 50
        // Pop the top element
        maxHeap.poll();
        // Get the size of the max heap priority queue
        System.out.println("Size of the priority queue: " + maxHeap.size()); // 3
        // Check if the priority queue is empty
        if (maxHeap.isEmpty()) {
            System.out.println("The priority queue is empty.");
        } else {
            System.out.println("The priority queue is not empty."); // Not empty
        }
        // Create a second max heap priority queue
        PriorityQueue<Integer> maxHeap2 = new PriorityQueue<>(Collections.reverseOrder());
        // Swap the content of the first max heap priority queue with the second one
        maxHeap2.addAll(maxHeap);
        System.out.println("After swapping, the first priority queue size: " + maxHeap.size()); // 0
        System.out.println("After swapping, the second priority queue size: " + maxHeap2.size()); // 3
    }
}
```

- **PriorityQueue<Integer> maxHeap**: Declare a max-heap priority queue named maxheap for storing integers. It's initialized as a max-heap by using Collections.reverseOrder() as the comparator.
- **maxHeap.add()**: Push elements into the max-heap priority queue.
- **maxHeap.peek()**: Print the top element (with the highest priority) without removing it.
- **maxHeap.poll()**: Remove the top element from the max-heap priority queue.
- **maxHeap.size()**: Print the size of the max-heap priority queue.
- **maxHeap.isEmpty()**: Check if the max-heap priority queue is empty.
- **maxHeap2.addAll(maxHeap)**: Swap the content of the first max-heap priority queue with the second one by adding all elements from maxheap to maxHeap2.

MinHeap or Min Priority Queue in Java

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();
        // Push elements into the min heap priority queue
        minHeap.add(30); // minHeap = {30}
        minHeap.add(10); // minHeap = {10, 30}
        minHeap.add(50); // minHeap = {10, 30, 50}
        minHeap.add(20); // minHeap = {10, 20, 30, 50}
// Print the top element (element with the lowest priority) without removing it
        System.out.println("Top element: " + minHeap.peek()); // 10
        // Pop the top element
        minHeap.poll();
        // Get the size of the min heap priority queue
        System.out.println("Size of the priority queue: " + minHeap.size()); // 3
        // Check if the priority queue is empty
        if (minHeap.isEmpty()) {
          System.out.println("The priority queue is empty.");
        } else {
          System.out.println("The priority queue is not empty."); // Not empty
        }
        // Create a second min heap priority queue
        PriorityQueue<Integer> minHeap2 = new PriorityQueue<>();
        // Swap the content of the first min heap priority queue with the second one
        minHeap2.addAll(minHeap);
        System.out.println("After swapping, the first priority queue size: " + minHeap.size()); // 0
        System.out.println("After swapping, the second priority queue size: " + minHeap2.size()); // 3
    }
}
```

- **PriorityQueue<Integer> minHeap**: Declare a min-heap priority queue named minheap for storing integers.
- **minHeap.add()**: Push elements into the min-heap priority queue.
- **minHeap.peek()**: Print the top element (with the lowest priority) without removing it.
- **minHeap.poll()**: Remove the top element from the min-heap priority queue.
- **minHeap.size()**: Print the size of the min-heap priority queue.
- **minHeap.isEmpty()**: Check if the min-heap priority queue is empty.
- **minHeap2.addAll(minHeap)**: Swap the content of the first min-heap priority queue with the second one by adding all elements from minheap to minHeap2.

9. Set in Java

A Set is a container in Java that stores unique elements in a sorted order and does not allow duplicate values.

Set can be implemented using:

- LinkedHashSet
- TreeSet

Properties

- **Ordered:** Yes, maintains the order of elements as they were inserted.
- **Unique:** Yes, only allows unique elements without duplicates.

LinkedHashSet in Java

- It is a variant of the HashSet that combines hash table and linked list structures to maintain a doubly-linked list across its elements.
- This hybrid structure allows LinkedHashSet to order its elements based on the sequence of their insertion, thus preserving the insertion order.
- It inherits from the HashSet and implements the Set interface.

Basic Operations

- **Adding Elements:** Inserts elements into the set without duplicating values. If an attempt is made to add a duplicate, it is ignored.
- **Removing Elements:** Deletes a specified element from the set.
- **Iteration:** Provides an iterator that follows the insertion order of the elements.
- **Checking for Elements:** Checks whether a specific element exists in the set.
- **Size and Cleanup:** Returns the number of elements and clears all elements from the set.

Example Code

```
import java.util.LinkedHashSet;
import java.util.Iterator;
import java.util.Set;
class LinkedHashSetExample {
    public static void main(String[] args) {
        // Create a LinkedHashSet and insert elements
        Set<String> linkedHashSet = new LinkedHashSet<>();
        linkedHashSet.add("apple"); // {apple}
        linkedHashSet.add("banana"); // {apple, banana}
        linkedHashSet.add("cherry"); // {apple, banana, cherry}
        linkedHashSet.add("banana"); // {apple, banana, cherry} --> 'banana' doesn't get inserted again
        // Erase elements from the set
        linkedHashSet.remove("banana"); // Erase element 'banana'
        Iterator<String> iterator = linkedHashSet.iterator();
        while (iterator.hasNext()) {
            String element = iterator.next();
            if (element.equals("cherry")) {
                iterator.remove(); // Erase element 'cherry'
            }
        }
        linkedHashSet.addAll(Set.of("orange", "grape", "lemon"));
        boolean hasApple = linkedHashSet.contains("apple");
        boolean hasBanana = linkedHashSet.contains("banana");
        System.out.println("Contains apple: " + hasApple); // true
        System.out.println("Contains banana: " + hasBanana); // false
        //Size
        System.out.println("Size of LinkedHashSet after cleanup: " + linkedHashSet.size());
        // Cleanup: clear all elements
        linkedHashSet.clear();
        //Size
        System.out.println("Size of LinkedHashSet after cleanup: " + linkedHashSet.size());
    }
}
```

TreeSet in Java

- It is an implementation of the Set interface that uses a tree for storage.
- Elements in a TreeSet are sorted according to their natural ordering, or by a Comparator provided at set creation, ensuring that the set is always in ascending order.
- This makes TreeSet an excellent choice for storing ordered collections that must also be free of duplicates.Properties
- **Sorted:** Yes, elements are sorted automatically either by natural ordering or by a specified Comparator.
- **Unique:** Yes, like other sets, it allows only unique elements and no duplicates.

Basic Operations

- **Adding Elements:** Adds new elements to the set in sorted order. If a duplicate is added, it is not inserted.
- **Removing Elements:** Removes a specified element from the set.
- **Iteration:** Provides an iterator to traverse the set in ascending order of the elements.
- **Checking for Elements:** Verifies the presence of an element in the set.
- **Size and Cleanup:** Gives the number of elements in the set and clears all elements.

Example Code

```
import java.util.TreeSet;
import java.util.Iterator;
import java.util.Set;
class TreeSetExample {
    public static void main(String[] args) {
        // Create a TreeSet and insert elements
        Set<String> treeSet = new TreeSet<>();
        treeSet.add("cherry"); // Elements are sorted as they are added
        treeSet.add("banana");
        treeSet.add("apple");
        treeSet.add("banana"); // 'banana' doesn't get inserted again
        // Erase elements from the set
        treeSet.remove("banana"); // Erase element 'banana'
        Iterator<String> iterator = treeSet.iterator();
        while (iterator.hasNext()) {
            String element = iterator.next();
            if (element.equals("apple")) {
                iterator.remove(); // Erase element 'apple'
            }
        }
        treeSet.addAll(Set.of("fig", "elderberry", "date"));
        boolean hasCherry = treeSet.contains("cherry");
        boolean hasBanana = treeSet.contains("banana");
        System.out.println("Contains cherry: " + hasCherry); // true
        System.out.println("Contains banana: " + hasBanana); // false
        // Size of set
        System.out.println("Size of TreeSet after cleanup: " + treeSet.size());
        //Clearing the set
        treeSet.clear();
        // Size of set
        System.out.println("Size of TreeSet after cleanup: " + treeSet.size());
    }
}
```

10.Multi-Set in Java

- A Multi-Set in Java, also known as a Bag, is a container that allows you to store multiple values of the same type in a sorted order.
- Unlike a Set, it can contain duplicate elements.

Properties

- **Sorted:** Yes, the elements are stored in sorted order.
- **Unique:** No, it allows duplicate elements.

Basic Operations

```
import java.util.TreeMap;
import java.util.Map;
import java.util.Set;
class Multiset<E> {
    private Map<E, Integer> map = new TreeMap<>();
    // Add an element to the multiset
    public void add(E element) {
        map.put(element, map.getOrDefault(element, 0) + 1);
    }

    // Add multiple instances of an element to the multiset
    public void add(E element, int occurrences) {
        if (occurrences < 0) throw new IllegalArgumentException("Occurrences cannot be negative.");
        map.put(element, map.getOrDefault(element, 0) + occurrences);
    }
    // Remove one occurrence of an element
    public void remove(E element) {
        map.computeIfPresent(element, (key, count) -> count > 1 ? count - 1 : null);
    }
    // Remove multiple occurrences of an element
    public void remove(E element, int occurrences) {
        if (occurrences < 0) throw new IllegalArgumentException("Occurrences cannot be negative.");
        map.computeIfPresent(element, (key, count) -> count > occurrences ? count - occurrences :
null);
    }
    // Get the count of an element in the multiset
    public int count(E element) {
        return map.getOrDefault(element, 0);
    }
    // Get the set of elements
    public Set<E> elementSet() {
        return map.keySet();
    }
    // Get the total size of the multiset
    public int size() {
        return map.values().stream().mapToInt(Integer::intValue).sum();
    }
    // Check if the multiset is empty
    public boolean isEmpty() {
        return map.isEmpty();
    }
}

public class Main {
    public static void main(String[] args) {
        Multiset<String> myMultiset = new Multiset<>();

        // Adding elements
        myMultiset.add("banana");
        myMultiset.add("apple");
        myMultiset.add("apple");
        myMultiset.add("banana", 2);
        myMultiset.add("apple", 3);  // Adds three more apples
        myMultiset.add("banana", 3);

        // Removing elements
        myMultiset.remove("apple");  // Removes one apple

        // Checking counts
        System.out.println("Count of apples: " + myMultiset.count("apple")); // 4
        System.out.println("Count of bananas: " + myMultiset.count("banana")); // 1

        // Iterating over multiset
        System.out.println("Elements in the multiset:");
        myMultiset.elementSet().forEach(element ->
          System.out.println(element + " x " + myMultiset.count(element)));

        // Checking if multiset is empty
        System.out.println("\nIs multiset empty? " + myMultiset.isEmpty());  // Output: false

        // Total size of the multiset
        System.out.println("Total size of multiset: " + myMultiset.size());  // Output: 3
    }
}
```

- **Multiset<E>**: A custom class to create a Multi-Set using a TreeMap that tracks element counts.
- **multiset.add()**: Insert elements into the multisetusing the addmethod.
- **multiset.remove()**: Erase elements from the multiset using the remove method.
- **Iterator**: No direct iterator; use elementSet() to get a set of unique elements for iteration.
- **multiset.count()**: Count the occurrences of a specific element in the multiset.
- **multiset.size()**: Returns the size.

11.Unordered Set in Java :

- An Unordered Set in Java is represented by the HashSet class from the Java Collections Framework.
- It is a container that stores a collection of unique elements without maintaining a specific order among the elements.

Properties

- **Sorted:** No, the elements are stored in a random order.
- **Unique:** It allows only unique elements.

Basic Operations :

```
import java.util.HashSet;
import java.util.Iterator;
public class Main {
    public static void main(String[] args) {
        // Creating an unordered set using HashSet
        HashSet<Integer> mySet = new HashSet<>();
        // Insert operation
        mySet.add(3);
        mySet.add(4);
        mySet.add(2);
        mySet.add(0);
        mySet.add(3);
        mySet.add(2);
        mySet.add(3);
        // mySet = [0, 2, 3, 4] -> random order and just an example
        // Erase operation
        mySet.remove(2);
        // Find operation
        if (mySet.contains(3)) {
            System.out.println("Found 3"); // Found 3
        } else {
            System.out.println("Not found");
        }
        // Count operation
        int count = 0;
        for (int element : mySet) {
            if (element == 0) {
                count++;
            }
        }
        System.out.println("Count of 0: " + count); // 1
    }
}
```

- **HashSet<Integer> mySet**: Declare a HashSet for storing integers.
- **mySet.add()**: Insert elements into the mySet using the add method.
- **mySet.remove()**: Remove elements from the mySet using the remove method.
- **mySet.contains()**: Use the contains method to check if a specific value exists in the mySet.
- **Iteration**: You can iterate through the elements of the mySet using a for-each loop or an iterator. The order of iteration is random.

Complexity Analysis

Operation	Set	Multi-Set	Unordered Set
insert()	O(log n)	O(log n)	O(1), worst case - O(n)
erase()	O(log n)	O(log n)	O(1), worst case - O(n)
find()	O(log n)	O(log n)	O(1), worst case - O(n)
count()	O(log n)	O(log n)	O(1), worst case - O(n)

In Java, the `HashSet` provides constant-time average complexity for basic operations, but worst-case complexity for certain operations can be O(n) when dealing with hash collisions.