Discussion-2: Working with Various Bag Implementations Lists

Discussion Topic:

This week we will further understand how to use arrays to implement an ADT bag in Java. Define array and give a simple code example demonstrating your understanding. How can you determine if you would need to utilize a fixed-size array or a dynamically-resizable array?

In your answer, specifically think of and give a real-life scenario where:

The array data structure can be used
The array list data structure can be used

My Post:

Hello class,

The ADT (Abstract Data Type) bag can be implemented using different data structures, including array-based implementations, linked lists, and more advanced data structures. This discussion explores, in Java, the array-based implementation of the ADT bag, which can be categorized into two approaches: fixed-size arrays and dynamic arrays.

Fixed-Size Array Bag Implementation Approach

In Java, this implementation utilizes a basic array, which is a fundamental data structure that stores a fixed number of elements of the same type (Java Doc., n.d.a). Each element in the array can be accessed using the array index functionality. Moreover, the size of the array is fixed and is set at the time of the array creation, meaning that once the array is full, no new element can be added to it.

Here is the implementation of Lady Gaga's songs list using an array:

```java
public class LadyGagaSongs {
    public static void main(String[] args) {
        // Declare an array of Lady Gaga songs
        String[] ladyGagaSongs = new String[5];

        // Initialize the array with song titles
        ladyGagaSongs[0] = "Bad Romance";
        ladyGagaSongs[1] = "Poker Face";
        ladyGagaSongs[2] = "Shallow";
        ladyGagaSongs[3] = "Just Dance";
        ladyGagaSongs[4] = "Alejandro";

        // Accessing an element in the array
        String firstSong = ladyGagaSongs[0]; // " Bad Romance"


        // Display the first song
        System.out.println("First song: " + firstSong);
    }
}
```

A fixed-size array bag implementation is best suited for accessing and setting elements. These operations have a constant-time complexity of O(1), see Table 1. However, adding or removing elements is inefficient because these operations often require shifting elements within the array, which runs in linear time O(n) in the worst case. Additionally, the bag has a set size due to the fixed array size making the fixed-size array bag implementation best suited for applications where the maximum number of items in the bag is known in advance and will not change. Nonetheless, by definition, an ADT bag supports dynamic sizing; thus if using a fixed-size array to implement an ADT bag a new larger array needs to be generated if the size of the initial array is reached see the code for Fixed-Size Array Bag Implementation, see the Java Code Fixed-Size Array Bag Implementation.

Table 1

Fixed-Array Bag Time Complexity

Method	Running time
`size()`	$O(1)$
`isEmpty()`	$O(1)$
`get(`i`)`	$O(1)$
`set(`i, e`)`	$O(1)$
`add(`i, e`)`	$O(n)$
`remove(`i`)`	$O(n)$

Note: From 7.2 Array lists, Data Structures and Abstractions with Java (5th Edition) by Carrano and Henry (2018).

Java Code Fixed-Size Array Bag Implementation

```
public class MySongBag<T> {
        private T[] bag;
        private int numberOfEntries;
        private static final int DEFAULT_CAPACITY = 10;

        // Constructor initializes the bag with default capacity
        @SuppressWarnings("unchecked")
        public MySongBag() {
                bag = (T[]) new Object[DEFAULT_CAPACITY];
                numberOfEntries = 0;
        }

        // Returns the current size of the bag
        public int getCurrentSize() {
                return numberOfEntries;
        }

        // Checks if the bag is empty
        public boolean isEmpty() {
                return numberOfEntries == 0;
        }
```

```
	// Adds a new entry to the bag
	public boolean add(T newEntry) {
		if (isArrayFull()) {
			resize(2 * bag.length); // Double the capacity if the array is full
		}
		bag[numberOfEntries] = newEntry;
		numberOfEntries++;
		return true;
	}

	// Removes and returns the last entry from the bag
	public T remove() {
		if (isEmpty()) {
			return null; // Nothing to remove
		} else {
			T result = bag[numberOfEntries - 1];
			bag[numberOfEntries - 1] = null;
			numberOfEntries--;
			return result;
		}
	}

	// Removes a specific entry from the bag
	public boolean remove(T anEntry) {
		int index = getIndexOf(anEntry);
		if (index >= 0) {
			bag[index] = bag[numberOfEntries - 1];
			bag[numberOfEntries - 1] = null;
			numberOfEntries--;
			return true;
		}
		return false;
	}

	// Clears all entries from the bag
	public void clear() {
		while (!isEmpty()) {
			remove();
		}
	}

	// Gets the frequency of a specific entry in the bag
	public int getFrequencyOf(T anEntry) {
		int frequency = 0;
		for (int i = 0; i < numberOfEntries; i++) {
			if (anEntry.equals(bag[i])) {
				frequency++;
			}
		}
		return frequency;
	}

	// Checks if the bag contains a specific entry
	public boolean contains(T anEntry) {
		return getIndexOf(anEntry) >= 0;
	}

	// Returns an array of all entries in the bag
	@SuppressWarnings("unchecked")
	public T[] toArray(T[] a) {
		if (a.length < numberOfEntries) {
			// If the array is too small, create a new array of the same runtime type and
			// size
			return (T[]) java.util.Arrays.copyOf(bag, numberOfEntries, a.getClass());
		}
		System.arraycopy(bag, 0, a, 0, numberOfEntries);
		if (a.length > numberOfEntries) {
			a[numberOfEntries] = null;
```

```
                }
                return a;
        }

        // ------ Private Methods -----

        // Checks if the array is full
        private boolean isArrayFull() {
                return numberOfEntries >= bag.length;
        }

        // Resizes the array
        @SuppressWarnings("unchecked")
        private void resize(int newCapacity) {
                T[] newBag = (T[]) new Object[newCapacity];
                System.arraycopy(bag, 0, newBag, 0, numberOfEntries);
                bag = newBag;
        }

        // Find the index of a specific entry
        private int getIndexOf(T anEntry) {
                int where = -1;
                boolean found = false;
                int index = 0;
                while (!found && (index < numberOfEntries)) {
                        if (anEntry.equals(bag[index])) {
                                found = true;
                                where = index;
                        }
                        index++;
                }
                return where;
        }
}
```

Dynamic Array Bag Implementation Approach

In Java, this implementation uses ArrayList, a resizable implementation of the List interface that provides all the methods related to list operations (JavaDoc, n.d.). Both arrays and ArrayLists use indexes to access data and share the same time complexities for these operations, see Table 1; however, the key difference is that ArrayLists can dynamically resize, while arrays have a fixed size. Additionally, ArrayLists have slightly higher memory usage due to their overhead functionality than arrays.

Here is the implementation of Lady Gaga's songs list using an ArrayList:

```
import java.util.ArrayList;
import java.util.List;

public class LadyGagaSongs {
    public static void main(String[] args) {
        // Declare an ArrayList of Lady Gaga songs
        List<String> ladyGagaSongs = new ArrayList<>();

        // Initialize the ArrayList with song titles
        ladyGagaSongs.add("Bad Romance");
        ladyGagaSongs.add("Poker Face");
        ladyGagaSongs.add("Shallow");
        ladyGagaSongs.add("Just Dance");
        ladyGagaSongs.add("Alejandro");

        // Accessing an element in the ArrayList
        String lastSong = ladyGagaSongs.get(ladyGagaSongs.size() - 1); // "Alejandro"

        // Display the first song
```

```
        System.out.println("Last song: " + lastSong);
    }
}
```

An ArrayList-based bag implementation is well-suited for accessing and setting elements, both of which have a constant-time complexity of O(1), as the fixed-size array ad shown in Table 1. In the context of the amount of code line, ArrayList are a little better at removing elements than arrays due to its dynamic resizing capability, though these operations can sometimes require shifting elements within the list, resulting in a worst-case time complexity of O(n). The ArrayList automatically adjusts its size as needed, making it ideal for bag implementation applications where the number of items in the bag can change often, see the see the Java Code ArrayList Bag Implementation.

Java Code ArrayList Bag Implementation

```
import java.util.ArrayList;
import java.util.List;

public class MySongBag<T> {
        private List<T> bag;

        // Constructor to initialize the bag
        public MySongBag() {
                bag = new ArrayList<>();
        }

        // Returns the current size of the bag
        public int getCurrentSize() {
                return bag.size();
        }

        // Checks if the bag is empty
        public boolean isEmpty() {
                return bag.isEmpty();
        }

        // Adds a new entry to the bag at a specific index
        public void add(int index, T newEntry) {
                if (index >= 0 && index <= bag.size()) {
                        bag.add(index, newEntry);
                } else {
                        throw new IndexOutOfBoundsException("Invalid index: " + index);
                }
        }

        // Sets an entry at a specific index
        public void set(int index, T newEntry) {
                if (index >= 0 && index < bag.size()) {
                        bag.set(index, newEntry);
                } else {
                        throw new IndexOutOfBoundsException("Invalid index: " + index);
                }
        }

        // Removes an entry at a specific index
        public T remove(int index) {
                if (index >= 0 && index < bag.size()) {
                        return bag.remove(index);
                } else {
                        throw new IndexOutOfBoundsException("Invalid index: " + index);
                }
        }

        // Removes a specific entry from the bag
```

```
        public boolean remove(T anEntry) {
                return bag.remove(anEntry);
        }

        // Clears all entries from the bag
        public void clear() {
                bag.clear();
        }

        // Gets the frequency of a specific entry in the bag
        public int getFrequencyOf(T anEntry) {
                int frequency = 0;
                for (T entry : bag) {
                        if (anEntry.equals(entry)) {
                                frequency++;
                        }
                }
                return frequency;
        }

        // Checks if the bag contains a specific entry
        public boolean contains(T anEntry) {
                return bag.contains(anEntry);
        }

        // Returns an array of all entries in the bag
        public T[] toArray(T[] a) {
                return bag.toArray(a);
        }

        // Displays all songs in the bag
        public void displaySongs() {
                System.out.println("Songs in the bag:");
                for (T song : bag) {
                        System.out.println(song);
                }
        }
}
```

In summary, both ADT bag implementation approaches have the same time complexity. However, fixed-size arrays need dynamic sizing to be implemented adding complexity to the implementation. Nonetheless, the fixed-size array is better for bag applications where the size of the bag is best suited for applications where the maximum number of items in the bag is known in advance and will not change, offering less memory overhead than an ArrayList. On the other hand, ArrayLists are better suited for applications where the number of items in the bag can change frequently, providing less implementation complexity despite slightly higher memory usage.

-Alex

References:

Carrano, F. M., & Henry, T. M. (2018, January 31). *Data structures and abstractions with Java (5th Edition). Pearson.*

Java Doc. (n.d.a). Arrays. *The Java Tutorials*. Oracle. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html

Java Doc. (n.d.b). Class ArrayList<E>. *API*. Oracle. https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html