Discussion-1: Introduction to the Abstract Data Types and Bags

Discussion Topic:

This week's discussion has provided an introduction to working with ADT's and bags in Java. What are some of the benefits of using an ADT bag in Java? Provide an example that illustrates when using a bag would be useful.
In your answer, specifically think of and give a real-life scenario where:

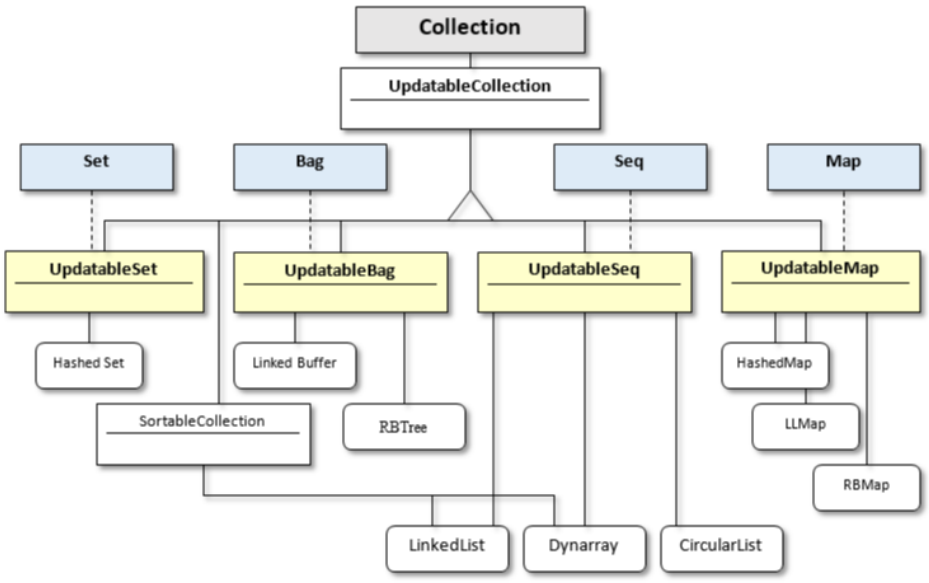
- The Bag ADT can be useful
- Using Bag ADT is better than the alternatives

My Post:

Hello class,

Abstract Data Type (ADT) can be defined as a "specification that describes a data set and the operations on that data" (Carrano & Henry, 2018). Java being an object-oriented language uses extensively ADTs in data structures, such as bags, queues, and deques.
The Bag type, in Java, is a fundamental data type belonging to the Collection class, see Figure 1.

Figure 1
Updatable Collections



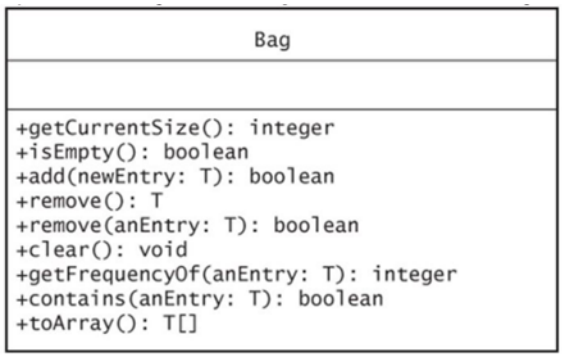
Note: A Linked Buffer is an implementation of UpdatableBag consisting of any number of buffers holding elements, arranged in a list. Each buffer holds an array of elements. A RBTree is an implementation of UpdatableBag, ElementSortedCollection implementing red-black trees, a form of balanced sorted binary search tree. *From* Overview of the Collections Package by Lee (n. d.).

The Bag's properties can be defined as follows:

- Duplicates - Allow duplicates.
- Unordered – The Bag elements are not ordered.
- Dynamic Sizing – The size can be changed dynamically.
- Iterable – In Java, Bags can be Iterabled.
- No Indexing - Bag elements are not indexed.
- Efficient Add Operation - Adding an element to a bag is typically an O(1) operation.
- Supports Collection Operations. See Figure 2.
- Generic Types - Bags can hold elements of any type.
- No key-value Pairs - bags store elements without associated keys.
- Mutable - The contents of a bag can be changed after creation, it can be resized dynamically.
- Supports Null Elements - bags may allow null elements.

Figure 2

Bag Class UML



From Data Structures and Abstractions with Java (5th Edition) by Carrano and Henry (2018).

In the context of the Java programming languages, Stack, Queue, LinkedList, Set, and Bag can be described as follows:

- Stack is ADT of the collection of elements with specific remove order = LIFO (last-in-first-out), allowing duplicates.
- Queue is ADT of the collection of elements with specific remove order = FIFO (first-in-first-out), allowing duplicates.
- LinkedList is an implementation of a list.
- Set is ADT of the collection of elements which disallows duplicates.
- Bag is ADT of the collection of elements which allows duplicates.

In other words, anything that holds an element is a Collection, and any collection that allows duplicates is a Bag, otherwise it is a Set. (Matoni, 2017)

The main benefit of the Bag is that it allows duplicates and iteration, it is also mutable allowing dynamic sizing and modification of its elements. In other words, "if duplicates are not desired, you would use Set instead of Bag. Moreover, if you care about remove order you would pick Stack or Queue which are

basically Bags with specific remove order. You can think of Bag as super-type of the Stack and Queue which extends its api by specific operations" (Matoni, 2017).

The program below is an example of a simple implementation of the Bag ADT approach to a grocery inventory.

The Bag class implements the Bag ADT:

```
package BagProgram;

import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * A simple implementation of a Bag ADT (Abstract Data Type).
 * It implements a linked structure systems, a chain data structure.
 * [item | next] -> [item | next] -> [item | next] -> null.
 *
 * @param <T> the type of elements held in this bag
 *
 * @author Alejandro Ricciardi
 * @date 08/12/2024
 */
public class Bag<T> implements Iterable<T> {
    private Node headNode; // The first node in the bag list
    private int size;      // Number of node in the bag

    /**
     * Private inner class that creates a node in the linked structure.
     * Each node contains an item and a reference to the next node.
     * [item | next] -> [item | next] -> [item | next] -> null
     *
     */
    private class Node {
        T item;     // The item stored in the node
        Node next;  // Reference to the next node
    }

    /**
     * Constructs an empty bag.
     */
    public Bag() {
        headNode = null;  // Initialize the first (head) node in the bag to null (empty bag)
        size = 0;         // Initialize the size of the bag to 0
    }

    /**
     * Adds the specified item to this bag.
     * This operation runs in constant time O(1).
     *
     * @param item the item to add to the bag
     */
    public void add(T item) {
        Node nextNode = headNode;     // Save the current head node becoming the next node in
the added node
        headNode = new Node();        // Create a new node and make it the head node
        headNode.item = item;         // Set the item of the new node
        headNode.next = nextNode;     // Link the new node to the old head node
        size++;                       // Increment the size of the bag
    }
```

```java
/**
 * Removes one item from the bag if it exists.
 * This operation runs in linear time O(n) in the worst case.
 *
 * @param item the item to remove
 * @return true if the item was found and removed, false otherwise
 */
public boolean remove(T item) {
    if (headNode == null) return false;  // If the bag is empty, return false

    if (headNode.item.equals(item)) {    // If the item is in the first node
        headNode = headNode.next;        // Make the next node the new head node
        size--;                          // Decrement the size
        return true;
    }

    Node current = headNode;
    while (current.next != null) {             // Iterates the linked structure
        if (current.next.item.equals(item)) {  // If the next node contains the item
            current.next = current.next.next;  // Remove the next node from the chain
            size--;                            // Decrement the size
            return true;
        }
        current = current.next;                // Move to the next node
    }
    return false;  // Item not found in the bag
}

/**
 * Returns the number of items in this bag.
 *
 * @return the number of items in this bag
 */
public int size() {
    return size;
}

/**
 * Checks if this bag is empty.
 *
 * @return true if this bag contains no items, false otherwise
 */
public boolean isEmpty() {
    return size == 0;
}

/**
 * Returns an iterator that iterates over the items in this bag.
 *
 * @return an iterator that iterates over the items in this bag
 */
public Iterator<T> iterator() {
    return new BagIterator();
}

/**
 * Private inner class that implements the Iterator interface.
 */
private class BagIterator implements Iterator<T> {
    private Node current = headNode;  // Start from the first (head) node
```

```
        /**
         * Checks if there are more elements to iterate over.
         *
         * @return true if there are more elements, false otherwise
         */
        public boolean hasNext() {
            return current != null;
        }

        /**
         * Returns the next element in the iteration.
         *
         * @return the next element in the iteration
         * @throws NoSuchElementException if there are no more elements
         */
        public T next() {
            if (!hasNext()) throw new NoSuchElementException();
            T item = current.item;   // Get the item from the current node
            current = current.next;  // Move to the next node
            return item;
        }
    }
}
```

The GroceryInventory implements a simple grocery store inventory management system using the Bag ADT approach:

```
/**
 * A simple grocery store inventory management system using a Bag ADT.
 *
 * @author Alejandro Ricciardi
 * @date 08/12/2024
 */
public class GroceryInventory {
    // The Bag ADT used to store items
    private Bag<String> inventory;

    /**
     * Constructs a new GrocoryInventory
     */
    public GroceryInventory() {
        inventory = new Bag<>();
    }

    /**
     * Adds items to the inventory.
     *
     * @param item The name of the item to add.
     * @param quantity The number of items to add.
     */
    public void addShipment(String item, int quantity) {
        // Add the item to the bag 'quantity' times
        for (int i = 0; i < quantity; i++) {
            inventory.add(item);
        }
    }

    /**
     * Removes item from the inventory.
     *
```

```
     * @param item The name of the item to remove.
     * @return true if the item was successfully removed, false otherwise.
     */
    public boolean sellItem(String item) {
        return inventory.remove(item);
    }

    /**
     * Counts the number of a specific item in the inventory (Duplicates).
     *
     * @param item The name of the item to be counted.
     * @return The number of this item in the inventory.
     */
    public int getItemCount(String item) {
        int count = 0;
        // Iterate through the bag and count the number of the given item in it
        for (String s : inventory) {
            if (s.equals(item)) {
                count++;
            }
        }
        return count;
    }

    /**
     * Main method.
     */
    public static void main(String[] args) {
        GroceryInventory store = new GroceryInventory();

        // Add inventory
        store.addShipment("Apple", 50);
        store.addShipment("Banana", 30);
        store.addShipment("Orange", 40);

        // Initial apple count
        System.out.println("Apples in stock: " + store.getItemCount("Apple"));

        // Selling apples
        store.sellItem("Apple");
        store.sellItem("Apple");

        // Apple count after selling
        System.out.println("Apples after selling 2: " + store.getItemCount("Apple"));
    }
}
```

The Bag ADT is useful in this example because it is iterable, it can store multiple instances of the same object (e.g., duplicates like apples), and creates an abstraction that simplifies operations on the item's data and does not require the items to be ordered. Unlike arrays, Bags do not require a fixed size and can dynamically resize as needed. Additionally, they are better than Lists because they do not require the items to be ordered (e.g. apples can be stored after or before bananas). Furthermore, they are better than Sets because they allow duplicates (e.g. you can have more than one apple in the store).
In other words, is useful and better in this example because:

- You can easily add multiple apples or bananas (addShipment).
- You can remove a single item when it's sold (sellItem).
- You can count how many of each item you have (getItemCount)

-Alex

References:

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

Lee, D. (n. d.). Overview of the collections package [Doug Lea's home page]. Department of Computer Science, State University Of New York. https://gee.cs.oswego.edu/dl/classes/collections/

Matoni (2017, April 15). Reasons for using a Bag in Java [Post]. Stackoverflow. https://stackoverflow.com/questions/43428114/reasons-for-using-a-bag-in-java