420-301-VA: Programming Patterns

Lab 4: Implementing Data Structures: List, Stack, Queue

Computer Science & Technology
Vanier College

Introduction

In the previous lab, we explored how to use the high-performance data structures provided by the Java Collections Framework. Now, we will peel back the layers of abstraction and investigate how these fundamental structures are built. This lab is about getting into the internal details of these collections, implementing the logic for different methods, and understanding the trade-offs between different implementation strategies, such as using an array versus a linked structure. By building these data structures from the ground up, you will gain a deeper appreciation for their design and performance characteristics.

Objectives

This lab will guide you through the implementation of core data structures. By the end of this lab, you should be comfortable with:

- Designing common operations for lists within a Java interface.
- Implementing a dynamic list using a backing array (`MyArrayList`).
- Understanding the mechanism of dynamic array resizing to accommodate a growing number of elements.
- Implementing a dynamic list using a linked structure of nodes (`MyLinkedList`).
- Implementing ‘Stack‘ and ‘Queue‘ classes using composition, leveraging existing list implementations.
- Appreciating the design principle of favoring composition over inheritance for building new data structures from existing ones.

Instructions

- Please try out the examples first to deepen your understanding.
- For each exercise, create the required classes and a main method to test your implementation. You may need to use the test code provided via URLs in the exercise descriptions.
- Copy-paste your formatted code for each exercise into a separate document, followed by screenshots of the console outputs to validate that your code worked as expected.
- You can work with your peers, but each person must write, execute, and document their own work.

1 Part 1: Implementing an `ArrayList`

1.1 Concept Review: The Array-Backed List

An `ArrayList` implements a dynamic list using a fixed-size array. But how can a fixed-size structure be dynamic? The trick is to create a new, larger array and copy the contents of the old array over whenever the current array runs out of capacity.

We will start with a common interface, `MyList`, which will define the standard operations for our list implementations.

```java
public interface MyList<E> extends java.util.Collection<E> {
  /** Add a new element at the specified index in this list */
  public void add(int index, E e);

  /** Return the element from this list at the specified index */
  public E get(int index);

  /** Return the index of the first matching element in this list.
   * Return -1 if no match. */
  public int indexOf(Object e);

  /** Remove the element at the specified position in this list.
   * Shift any subsequent elements to the left.
   * Return the element that was removed from the list. */
  public E remove(int index);

  // ... other methods
}
```

Listing 1: The MyList Interface

When adding an element at a specific index, all subsequent elements must be shifted to the right to make space. Similarly, removing an element requires shifting subsequent elements to the left.

```java
@Override /** Add a new element at the specified index */
public void add(int index, E e) {
  // Ensure the index is in the right range
  if (index < 0 || index > size)
    throw new IndexOutOfBoundsException
      ("Index: " + index + ", Size: " + size);

  ensureCapacity();

  // Move the elements to the right after the specified index
  for (int i = size - 1; i >= index; i--)
    data[i + 1] = data[i];

  // Insert new element to data[index]
  data[index] = e;

  // Increase size by 1
  size++;
}

/** Create a new larger array, double the current size + 1 */
private void ensureCapacity() {
  if (size >= data.length) {
    E[] newData = (E[])(new Object[size * 2 + 1]);
    System.arraycopy(data, 0, newData, 0, size);
    data = newData;
  }
}
```

Listing 2: Adding an element and ensuring capacity in MyArrayList

Exercise 1: Implement Set Operations in `MyList`

The provided `MyList` interface omits the implementation of several methods inherited from the `Collection` interface, such as `addAll`, `removeAll`, and `retainAll`.

Implement these methods as `default` methods directly within the `MyList.java` interface file. Test your implementations using the provided test code.

1. Download the starting `MyList.java` and `MyArrayList.java` files.
2. Implement the following `default` methods in `MyList.java`:
   - `public default boolean containsAll(Collection<?> c)`
   - `public default boolean addAll(Collection<? extends E> c)`
   - `public default boolean removeAll(Collection<?> c)`
   - `public default boolean retainAll(Collection<?> c)`
   - `public default Object[] toArray()`
   - `public default <T> T[] toArray(T[] a)`
3. Use the test program from Exercise24_01.txt to validate your implementation.

2 Part 2: Implementing a `LinkedList`

2.1 Concept Review: The Node-Based List

A `LinkedList` offers a different approach. Instead of a contiguous block of memory like an array, it uses a "linked structure" composed of individual `Node` objects. Each node contains an element and a reference (or "pointer") to the next node in the sequence.

```java
private static class Node<E> {
  E element;
  Node<E> next;

  public Node(E element) {
    this.element = element;
  }
}
```

Listing 3: A simple Node class

Operations like adding to the beginning (`addFirst`) are very efficient because they only require updating a few references, without any need to shift elements.

```java
/** Add an element to the beginning of the list */
public void addFirst(E e) {
  Node<E> newNode = new Node<>(e); // Create a new node
  newNode.next = head; // link the new node with the head
  head = newNode; // head points to the new node
  size++; // Increase list size

  if (tail == null) // the new node is the only node in list
    tail = head;
}
```

Listing 4: Adding an element to the front of a MyLinkedList

Exercise 2: Completing the `MyLinkedList` Implementation

The provided `MyLinkedList` implementation omits several key methods for accessing and modifying elements. Your task is to implement them.

1. Download the starting `MyLinkedList.java` file. You will also need your completed `MyList.java` from the previous exercise.

2. Implement the following methods inside the `MyLinkedList` class:

   - `public boolean contains(Object e)`
   - `public E get(int index)`
   - `public int indexOf(Object e)`
   - `public int lastIndexOf(E e)`
   - `public E set(int index, E e)`

3. Create a main method or use the test code at Exercise24_02.txt to validate your work. For methods like `get` and `indexOf`, remember that you must traverse the list from the `head` node.

3 Part 3: Implementing a `Stack` (LIFO)

3.1 Concept Review: Building with Composition

A stack is a Last-In, First-Out (LIFO) data structure. While one could implement a stack by extending `ArrayList`, a better design approach is to use **composition**. With composition, our `MyStack` class will contain an instance of an `ArrayList` as a private field and delegate the work to it. This is preferable because it prevents inheriting unnecessary methods (like `get(index)`) from the list, creating a cleaner and more robust API for our stack.

```java
public class MyStack<E> {
  private java.util.ArrayList<E> list = new java.util.ArrayList<>();

  public void push(E o) {
    list.add(o);
  }

  public E pop() {
    E o = list.get(getSize() - 1);
    list.remove(getSize() - 1);
    return o;
  }

  public E peek() {
    return list.get(getSize() - 1);
  }

  public int getSize() {
    return list.size();
  }

  public boolean isEmpty() {
    return list.isEmpty();
  }

  // ... other methods
}
```

Listing 5: MyStack using an ArrayList for storage

Exercise 3: Implementing a Stack with a LinkedList

In the concept review, we saw a `MyStack` class implemented using an `ArrayList`. While this is efficient, a stack can also be implemented using other list structures. In this exercise, you will implement a stack using a `java.util.LinkedList` via composition. This approach is useful in scenarios where operations at both ends of a list are common, and it demonstrates the flexibility of the composition pattern.

1. Create a new class named `LinkedStack<E>`.

2. Use composition by creating a private `java.util.LinkedList<E>` data field to store the elements.

3. Implement the standard public stack methods by delegating calls to the internal linked list:

   - `public void push(E e)`: This should add an element to the end of the linked list.
   - `public E pop()`: This should remove and return the element from the end of the linked list.
   - `public E peek()`: This should return the element from the end of the list without removing it.
   - `public int getSize()`: Should return the number of elements in the stack.
   - `public boolean isEmpty()`: Should check if the stack is empty.

4. Use the test code from Exercise24_Stack.txt to create a main method and validate your implementation.

4 Part 4: Implementing a `Queue` (FIFO)

4.1 Concept Review: Composition with LinkedList

A queue is a First-In, First-Out (FIFO) data structure, much like a waiting line. Since elements are added to one end (the tail) and removed from the other (the head), a `LinkedList` is a more efficient underlying structure than an `ArrayList`. Removing from the front of an `ArrayList` is an $O(n)$ operation, whereas for a `LinkedList` it is $O(1)$.

```java
public class MyQueue<E> {
  private java.util.LinkedList<E> list = new java.util.LinkedList
     <>();

  public void enqueue(E e) {
    list.addLast(e);
  }

  public E dequeue() {
    return list.removeFirst();
  }

  public int getSize() {
    return list.size();
  }

  // ... other methods
}
```

Listing 6: MyQueue using a LinkedList for storage

Exercise 4: Implementing a Deque

A Deque, or "double-ended queue", is a generalization of a queue that allows adding and removing elements from both the front and the back.

1. Create a new class named `MyDeque<E>`.
2. Use composition, containing a private `java.util.LinkedList<E>` to store the elements.
3. Implement the following public methods by delegating the calls to the internal linked list:
   - `void addFirst(E e)`
   - `void addLast(E e)`
   - `E removeFirst()`
   - `E removeLast()`
   - `E peekFirst()`
   - `E peekLast()`
   - `int getSize()`
   - `boolean isEmpty()`

4. Create a main method to test your `MyDeque` implementation by adding and removing several elements from both ends and printing the result.

References

Core Textbook & Code Implementations

- Liang, Y. Daniel: Introduction to Java Programming and Data Structures, 12th Edition: The official textbook chapter that these implementations are based on[cite: 1]. It provides detailed explanations and diagrams for each data structure.
- Liang, Y. Daniel: MyList.java: The source code for the custom `MyList` interface that defines the common operations for list implementations.
- Liang, Y. Daniel: MyArrayList.java: The source code for the array-based list implementation, demonstrating dynamic array resizing and element shifting.
- Liang, Y. Daniel: MyLinkedList.java: The source code for the singly linked-list implementation, showing node-based data management.

Stack and Queue Implementations

- Liang, Y. Daniel: MyStack.java: The source code for a generic stack class implemented using composition (containing an `ArrayList`).
- Liang, Y. Daniel: MyQueue.java: The source code for a generic queue class implemented using composition (containing a `LinkedList`).

Test Programs

- Liang, Y. Daniel: TestMyArrayList.java: A test program for validating the functionality of the `MyArrayList` implementation.
- Liang, Y. Daniel: TestMyLinkedList.java: A test program for validating the functionality of the `MyLinkedList` implementation.
- Liang, Y. Daniel: Exercise24_01.txt: Test code for verifying the implementation of set operations in the `MyList` interface.