420-301-VA: Programming Patterns

Assignment 1: Implementing Doubly Linked Lists

Computer Science & Technology
Vanier College

Introduction

Linked lists are one of the most fundamental data structures, forming the basis for many other complex structures and algorithms. The standard singly linked list allows for efficient insertions and deletions but is limited to forward traversal. An enhancement to this is the doubly linked list, which includes a `previous` pointer in each node, enabling bidirectional traversal.

This assignment focuses on a deep dive into these two structures. You will implement a doubly linked list from scratch and then perform a detailed theoretical and empirical comparison against its singly linked counterpart to understand the practical trade-offs of this design enhancement.

Instructions

- Please read each part carefully and ensure you understand the requirements before starting.
- For the implementation part, create the required Java classes and a main method (or a separate test class) to test your implementation thoroughly.
- Your code should be well-commented and formatted.
- For analysis questions, provide clear and concise justifications for your answers.
- This is an individual assignment. You may discuss concepts with peers, but the work you submit must be your own.
- The final submission should contain a PDF with ALL the code that you added, your calculations, your comments, your answers, and any output screenshots. **In addition, you need to submit a zip file with all your code that can be run from main()**. If the code is incomplete or does not run, you will be penalized.

Part 1: Implement a Doubly Linked List (25 points)

The `MyLinkedList` from the lab is a singly linked list. Your first task is to build a doubly linked list, which provides more flexibility by allowing traversal in both directions.

1. Modify the `Node` class to include a reference to the previous node:

```
public class Node<E> {
  E element;
  Node<E> next;
  Node<E> previous;

  public Node(E e) {
    element = e;
  }
}
```

2. Create a new class named `DoublyLinkedList<E>` that implements the `MyList<E>` interface using this new doubly-linked `Node`. Your class must keep track of both `head` and `tail` pointers to manage the list efficiently.

3. Implement all the necessary methods from the `MyList` interface. Pay close attention to updating both the `next` and `previous` pointers correctly for all operations. You must implement at least the following methods:

   - `add(int index, E e)`
   - `addFirst(E e)` and `addLast(E e)`
   - `remove(int index)`
   - `removeFirst()` and `removeLast()`
   - `get(int index)`
   - `contains(Object e)`
   - `size()`
   - `clear()`

4. Use the test code from Exercise24_03.txt to thoroughly test your implementation.

Part 2: Complexity Analysis (15 points)

For each of the following, determine the time complexity in Big O notation. In each case, clearly state what the input size `n` represents and provide a brief justification for your answer.

1. **Code Snippet 1:**

```
void process(int n) {
    for (int i = 1; i < n; i = i * 2) {
        System.out.println("Processing...");
    }
}
```

2. **Code Snippet 2:**

```
boolean hasDuplicates(int[] array) {
    for (int i = 0; i < array.length; i++) {
        for (int j = i + 1; j < array.length; j++) {
            if (array[i] == array[j]) {
                return true;
            }
        }
    }
    return false;
}
```

3. **Singly vs. Doubly Linked List - Insertion:** Analyze the time complexity of the `add(int index, E e)` operation for both the singly-linked `MyLinkedList` and your `DoublyLinkedList`.

   - First, describe the steps required to perform this operation in each implementation.
   - Then, intuitively reason about the advantage the `previous` pointer gives the `DoublyLinkedList` for this operation. Can you optimize the traversal to the node at `index`? How does this potential optimization affect the worst-case time complexity?

4. **Singly vs. Doubly Linked List - Removal:** Compare the time complexity of the `removeLast()` operation for the singly-linked `MyLinkedList` and your doubly-linked `DoublyLinkedList`. Explain why they are the same or different by referring to the steps each implementation must take.

Part 3: Empirical Performance Comparison (10 points)

Theoretical complexity analysis is crucial, but empirical testing can validate our understanding and reveal real-world performance. In this bonus part, you will measure and compare the execution time of key operations on `MyLinkedList` and your `DoublyLinkedList`.

1. **Set up the Test Environment:** Use the starter code below to create a performance testing class. This code provides a utility method, `timeOperation`, which takes a description and a `Runnable` to time an operation.

```
import java.util.Random;

public class PerformanceTest {

    public static void timeOperation(String description,
       Runnable operation) {
        long startTime = System.nanoTime();
        operation.run();
        long endTime = System.nanoTime();
        long duration = (endTime - startTime) / 1000000; //
           milliseconds
        System.out.printf("%-50s: %d ms%n", description,
           duration);
    }

    public static void main(String[] args) {
        int N = 50000; // Number of elements for bulk
           operations
        int M = 10000; // Number of elements for random
           insertions

        System.out.println("--- Testing with N = " + N + "
           elements ---");

        // --- Add at Random Index ---
        Random rand = new Random();
        MyLinkedList<Integer> singleListForRandom = new
           MyLinkedList<>();
        DoublyLinkedList<Integer> doubleListForRandom = new
           DoublyLinkedList<>();

        timeOperation("Singly: Add at Random Index (" + M + "
           times)", () -> {
            for (int i = 0; i < M; i++) {
                // Add to a list that is growing
                int size = singleListForRandom.size();
                int index = size == 0 ? 0 : rand.nextInt(size);
                singleListForRandom.add(index, i);
```

```
            }
        });

        timeOperation("Doubly: Add at Random Index (" + M + "
            times)", () -> {
            for (int i = 0; i < M; i++) {
                int size = doubleListForRandom.size();
                int index = size == 0 ? 0 : rand.nextInt(size);
                doubleListForRandom.add(index, i);
            }
        });

        // TODO: Implement the other tests as described below.
    }
}
```

2. **Conduct the following tests:** For each test, create two new, empty lists and measure the time taken.
   - **Adding at the Beginning:** Add N elements to the **front** of each list using `addFirst()`.
   - **Removing from the End:** Pre-populate both lists with N elements. Then, remove all N elements one by one from the **end** of the list using `removeLast()`.
3. **Record and Analyze Results:** Run your tests with N = 50,000 and M = 10,000. Record your results (in milliseconds) in the table below.

Operation	**MyLinkedList (ms)**	**DoublyLinkedList (ms)**
Add at Random Index (M times)		
Add at Beginning (N times)		
Remove from End (N times)		

4. **Analysis Question:** For which operation(s) did you observe the most significant performance difference? Explain why this occurs by referring to the underlying implementation of each data structure and its theoretical time complexity for that operation.

Bonus: Functional Programming with a Doubly Linked List (10 points)

While linked lists are often managed with imperative loops, modern Java provides powerful functional constructs like Streams and lambdas that can offer a more declarative way to express operations. This bonus exercise challenges you to think functionally and apply these concepts to the `DoublyLinkedList` you built.

1. Stream-based Node Access

Your goal is to implement a method to get a node at a specific index without using a traditional `for` or `while` loop. Instead, you will use a stream to traverse the list. This technique can be used as a helper method for many common operations like `get(int index)` and `add(int index, E e)`.

1. Create a new public method `public Node<E> getNodeUsingStream(int index)`.
2. Inside this method, use `Stream.iterate` to generate a sequence of nodes starting from the `head`. The stream should advance from one node to the next using its `next` pointer.
3. Use stream operations like `limit()`, `skip()`, and `findFirst()` to locate the desired node at the given index.
4. Return the found node. Handle out-of-bounds indices appropriately (e.g., by returning null or throwing an exception).

2. Conditional Removal with a Predicate

Implement a method that removes elements based on a condition provided by a lambda expression. This is a powerful feature available in Java's standard collections.

1. Add the method `public void removeIf(java.util.function.Predicate<E> filter)` to your `DoublyLinkedList` class.
2. Iterate through your list from `head` to `tail`.
3. For each node, use the `test` method of the provided `filter` predicate to check if the node's element should be removed.
4. If an element needs to be removed, you must carefully update the `next` and `previous` pointers of the surrounding nodes to unlink the current node from the list. Pay close attention to edge cases, such as removing the head, the tail, or consecutive nodes.
5. Provide a test case in your `main` method that demonstrates this functionality, for example, removing all even numbers from a `DoublyLinkedList<Integer>`.