Ch. 13 CSE3110 - Iterative Algorithm 1: Reflection Log

Zephram Gilson

StackList (+ StackListDemo)

The StackList class in Java uses a linked list to implement a stack; it includes standard stack operations: push, pop, peek, and isEmpty.

First, I created the main `StackList` class. I also set a private field, `top`, which will keep track of the top element of the stack.

```
public class StackList {
    private Node top;
```

To manage the stack, I needed a `Node` class to represent each item in the stack. This inner class holds two fields:

- `data`: stores the actual value of the element.
- `next`: a reference to the next node in the stack.

```
public class StackList {
    private Node top;

    // Inner Node class to represent each element in the stack
    private class Node {
        Object data;
        Node next;

        Node(Object data) {
            this.data = data;
        }
    }
```

Next, I added a constructor for `StackList`. When the stack is created, it starts empty, so I set `top` to `null`.

```
// Constructor for an empty stack
public StackList() {
    top = null;
}
```

Next, I added the `push` method to a new item to the top of the stack. First, I created a new `Node` with the provided `data`. Then I set this new node's `next` reference to the current `top` node. Lastly, I updated `top` to refer to this new node.

```
// Push operation to add an element to the top of the stack
public void push(Object data) {
    Node newNode = new Node(data);
    newNode.next = top;
    top = newNode;
}
```

After adding the push method, I added the `pop` method, which removes and returns the top element from the stack. First, I checked if the stack is empty (if `top` is `null`). If so, I threw an `IllegalStateException`. If the stack is not empty, I stored the `data` from `top`, updated `top` to `top.next` (removing the top element), and returned the stored data.

```
// Pop operation to remove the top element from the stack
public Object pop() {
    if (isEmpty()) {
        throw new IllegalStateException("Stack is empty");
    }
    Object data = top.data;
    top = top.next;
    return data;
}
```

Next, I added the `peek` method that returns the top element without removing it. I implemented it similarly to `pop` but without modifying the `top`.

```
// Peek operation to view the top element without removing it
public Object peek() {
    if (isEmpty()) {
        throw new IllegalStateException("Stack is empty");
    }
    return top.data;
}
```

To check if the stack is empty, I implemented the `isEmpty` method that checks if the `top` is `null`.

```
// Check if the stack is empty
public boolean isEmpty() {
    return top == null;
}
```

Then, in my client code, I started by creating a `main` method to serve as the entry point of the program. This method demonstrates the functionality of the `StackList` class by performing several operations on a stack.

```
public class StackListDemo {
    public static void main(String[] args) {
        StackList stack = new StackList();
```

To start, I demonstrated the `push` method by pushing a few elements (`10`, `20`, and `30`) onto the stack. I added a `System.out.println` statement to indicate which elements were being added.

```
// Pushing elements onto the stack
System.out.println("Pushing elements: 10, 20, 30");
stack.push(10);
stack.push(20);
stack.push(30);
```

Then, to demonstrate the `peek` method, I added a `System.out.println` statement to indicate what element is at the top of the `StackList.`

```
// Peeking at the top element
System.out.println("Top element (peek): " + stack.peek());
```

Next, the `pop` method is demonstrated by removing elements from the stack in a loop until it's empty. To print each removed element, I used a `while` loop that continues as long as the stack isn't empty.

```
// Popping elements from the stack
System.out.println("\n" + "Popping elements:");
while (!stack.isEmpty()) {
    System.out.println(stack.pop());
}
```

To illustrate error handling, I attempted to `peek` at the top element of an empty stack. This throws an `IllegalStateException,` which I handled with a `try-catch` block. In the `catch` block, I printed an appropriate message to show that an exception was caught.

Similarly, I tested the `pop` method on an empty stack to demonstrate underflow handling. This also throws an `IllegalStateException`, which I caught and printed an error message for.

```
// Trying to peek or pop from an empty stack
try {
    System.out.println("\n" + "Attempting to peek on an empty stack:");
    System.out.println(stack.peek());
} catch (IllegalStateException e) {
    System.out.println("Caught exception: " + e.getMessage());
}

try {
    System.out.println("\n" + "Attempting to pop from an empty stack:");
    stack.pop();
} catch (IllegalStateException e) {
    System.out.println("Caught exception: " + e.getMessage());
}
```

I made no major changes to my code in terms of logic/functionality throughout my process.

ReverseList

The ReverseList program uses a stack to reverse a list of up to 10 integers entered by the user, displaying them in reverse order by leveraging the stack's Last-In-First-Out structure.

First, I needed a stack and user input handling, so I imported `java.util.Stack` and `java.util.Scanner`. Inside the main method, I initialized:

- `scanner`, an instance of `Scanner`, to capture input from the user.
- `stack`, an instance of `Stack<Integer>`, to store the integers entered by the user.

```
package mastery;

import java.util.Scanner;

public class ReverseList {
    public static void main(String[] args) {
        // Create a Scanner for user input and a Stack to store integers
        Scanner scanner = new Scanner(System.in);
        Stack<Integer> stack = new Stack<>();
```

Then, I added a simple user prompt:

```
System.out.println("Enter up to 10 numbers to reverse, or enter 999 to finish early:");
```

To collect numbers, I created a `while` loop with a maximum of 10 iterations (since the user can enter up to 10 numbers). I also added a counter variable, `count`, initialized to 0. Each time a number is entered, the counter increments by one to keep track of the number of inputs.

```
int count = 0;  // Counter to track the number of inputs
while (count < 10) {  // Limit input to 10 numbers
    System.out.print("Enter number " + (count + 1) + ": ");
    int input = scanner.nextInt();

    if (input == 999) {  // Check if the user wants to stop early
        break;
    }

    stack.push(input);  // Push the valid input onto the stack
    count++;  // Increment the count of numbers entered
}
```

For valid numbers (anything other than `999`), I used `stack.push(input)` to add each number to the stack. This way, every time the user enters a number, it gets added to the top of the stack, making it easy to reverse the order later.

Once the input loop ended (either because the user entered 10 numbers or stopped with 999), I moved on to display the reversed list.

Since a stack is a LIFO structure, popping elements one by one returns them in reverse order. In this loop, I used `stack.pop()` to remove the top item from the stack.
Then, each popped item is printed, which effectively displays the user's numbers in reverse order.

```java
// Display reversed numbers by popping from the stack
System.out.println("\nReversed numbers:");
while (!stack.isEmpty()) {
    System.out.print(stack.pop() + " ");  // Pop each element to reverse order
}
scanner.close();
```

I originally planned to approach this using the same methodology as reversing an array, however this would not work, because of the stack's LIFO structure. I can only interact with the top element. Therefore, popping them proved to be the most efficient solution.

I made no other major changes to my code in terms of logic/functionality throughout my process.

QueueList (+ QueueListDemo)

The `QueueList` program implements a queue using a linked list, supporting standard operations like `enqueue`, `dequeue`, `peek`, `isEmpty`, and `getSize`. The `QueueListDemo` class tests these functions by enqueuing and dequeuing elements, displaying the queue's state, and handling exceptions when attempting operations on an empty queue, demonstrating the functionality and advantages of a linked list-based queue.

To create a queue that stores `Object` data using a linked list, I started by defining the `QueueList` class. This class needed to handle the standard queue operations: `enqueue`, `dequeue`, `peek`, and `isEmpty`.

To manage the elements, I created two private fields, `front` and `rear`, which would keep track of the front and rear nodes of the queue. I also added an `int size` field to keep track of the number of elements currently in the queue.

```java
package mastery.QueueList;

public class QueueList {
    private Node front;
    private Node rear;
    private int size;
```

Inside `QueueList`, I created a private `Node` class to represent each element in the queue. Each `Node` object would have:

- A `data` field to store the actual value of the element.
- A `next` reference, which points to the next node in the queue.

```java
// Inner Node class
private class Node {
    Object data;
    Node next;

    Node(Object data) {
        this.data = data;
        this.next = null;
    }
}
```

When a QueueList instance is created, the queue should start empty. Therefore, in the QueueList constructor, I initialized front and rear to null and set size to 0.

```
// Constructor
public QueueList() {
    front = null;
    rear = null;
    size = 0;
}
```

Next, I implemented the enqueue method to add a new item to the rear of the queue.

```
// Enqueue operation - Adds an item to the rear of the queue
public void enqueue(Object item) {
    Node newNode = new Node(item);
    if (rear != null) {
        rear.next = newNode;
    }
    rear = newNode;
    if (front == null) {
        front = newNode;
    }
    size++;
}
```

First, I created a new Node with the provided data. Then, if rear was not null, it meant the queue already had elements, so I set rear.next to point to this new node.

Then, I updated rear to refer to the new node. If the queue was initially empty (front was null), I also set front to the new node. Finally, I incremented the size to reflect the new element.

After enqueue, I added the dequeue method, which removes and returns the element at the front of the queue.

```
// Dequeue operation - Removes and returns the item at the front of the queue
public Object dequeue() {
    if (isEmpty()) {
        throw new IllegalStateException("Queue is empty");
    }
    Object item = front.data;
    front = front.next;
    if (front == null) {
        rear = null;  // Queue is now empty
    }
    size--;
    return item;
}
```

First, I checked if the queue was empty (i.e., if front was null). If it was empty, I threw an IllegalStateException. Otherwise, I stored the data from front, updated front to front.next (effectively removing the front element), and decreased the `size`. If the queue became empty after the dequeue (i.e., `front` became `null`), I also set `rear` to `null`.

Next, I added the `peek` method, which returns the element at the front without removing it.

```
// Peek operation - Returns the item at the front of the queue without removing it
public Object peek() {
    if (isEmpty()) {
        throw new IllegalStateException("Queue is empty");
    }
    return front.data;
}
```

I implemented it similarly to `dequeue` but without modifying `front`. If the queue was empty, `peek` would throw an `IllegalStateException`.

To check if the queue was empty, I implemented `isEmpty`, which simply returns `true` if `front` is `null`. This method makes it easy to check the queue state before dequeue or peek operations.

```
    // isEmpty operation - Checks if the queue is empty
    public boolean isEmpty() {
        return front == null;
    }
```

Finally, I added the `getSize` method to return the current size of the queue. This can help clients monitor the number of elements in the queue at any time.

```
// Returns the current size of the queue
public int getSize() {
    return size;
}
```

To demonstrate and test the `QueueList` class, I wrote the `QueueListDemo` class, where I used each queue operation.

I first demonstrated the `enqueue` method by adding a few elements ("A," "B," "C," and "D") to the queue. Each time an element was added, I printed a message to indicate what was enqueued. This let me verify that the elements were added in the correct order and that the queue size was updated.

```java
public class QueueListDemo {
    public static void main(String[] args) {
        QueueList queue = new QueueList();

        System.out.println("Enqueuing elements: A, B, C, D");
        queue.enqueue("A");
        queue.enqueue("B");
        queue.enqueue("C");
        queue.enqueue("D");

        System.out.println("Current queue size: " + queue.getSize());  // Expected: 4
        System.out.println("Peek front element: " + queue.peek());      // Expected: A
```

I demonstrated `peek` by printing the current front element without removing it. This confirmed that `peek` accurately displayed the front element without altering the queue state.

I tested `dequeue` by removing elements from the queue in a loop. Using `isEmpty` as a condition for the loop, I dequeued each element one by one, printing each removed element and the queue's size after each removal. This confirmed that `dequeue` correctly removed the elements from the front in a FIFO (First In, First Out) order.

```java
System.out.println("\nDequeuing elements:");
while (!queue.isEmpty()) {
    System.out.println("Dequeued: " + queue.dequeue());
    System.out.println("Queue size after dequeue: " + queue.getSize());
    if (!queue.isEmpty()) {
        System.out.println("Peek next front element: " + queue.peek());
    }
}
```

I attempted to `dequeue` and `peek` on an empty queue. Both operations throw an `IllegalStateException` when the queue is empty, so I placed these calls inside `try-catch` blocks. In the `catch` blocks, I printed appropriate messages to confirm that the exceptions were caught correctly. This verified that the queue's boundary conditions and exception handling were implemented properly.

```java
// Test dequeue on empty queue
try {
    System.out.println("\nAttempting to dequeue from an empty queue...");
    queue.dequeue();
} catch (IllegalStateException e) {
    System.out.println("Caught exception: " + e.getMessage());
}

// Test peek on empty queue
try {
    System.out.println("\nAttempting to peek at an empty queue...");
    queue.peek();
} catch (IllegalStateException e) {
    System.out.println("Caught exception: " + e.getMessage());
}

System.out.println("Final queue size: " + queue.getSize());  // Expected: 0
```

I made no major changes to my code in terms of logic/functionality throughout my process.