420-301-VA: Programming Patterns
Assignment 1: Implementing Doubly Linked Lists
Fabro, Cyril Efren (6243515)

Part 1: Implement a Doubly Linked List

DoublyLinkedList.java

```
import java.util.Iterator;
import java.util.ListIterator;

import org.w3c.dom.Node;

public class DoublyLinkedList<E> implements MyList<E> {
    protected Node<E> head, tail;
    protected int size = 0;

    @Override
    public int size() {
        return size;
    }

    @Override
    public boolean contains(Object o) {
        Node<E> current = head;
        while(current != null){
            if(current.element == o){
                return true;
            }
            current = current.next;
        }
        return false;
    }

    @Override
    public void clear() {
        size = 0;
        head = tail = null;
    }

    @Override
    public void add(int index, E e) {
        Node<E> current = head;
        if(index == 0){
            addFirst(e);
        }
        else if(current == tail){
```

```
            addLast(e);
        }
        else{
            for(int i = 1; i < index; i++){
                current = current.next;
            }
            Node<E> newNode = new Node<>(e);
            if(current == null){
                return;
            }
            else{
                newNode.next = current.next;
                newNode.prev = current;
                if(current.next != null){
                    current.next.prev = newNode;
                }
                current.next = newNode;
                size++;
            }
        }
    }

    public void addFirst(E e){
        Node<E> newNode = new Node<>(e);
        newNode.next = head;
        newNode.prev = null;
        if(head != null){
            head.prev = newNode;
        }
        head = newNode;
        if(tail == null){
            tail = newNode;
        }
        size++;
    }

    public void addLast(E e){
        Node<E> newNode = new Node<>(e);
        newNode.next = null;
        newNode.prev = tail;
        if (tail != null) {
            tail.next = newNode;
        }
        tail = newNode;
        if (head == null) {
```

```
            head = newNode;
        }
        size++;
    }

    @Override
    public E get(int index) {
        Node<E> current = head;
        int count = 0;
        while(current != null){
            if(count == index){
                return current.element;
            }
            count++;
            current = current.next;
        }
        return null;
    }

    @Override
    public int indexOf(Object e) {
        Node<E> current = head;
        int index = 0;
        while(current != null){
            if(current.element.equals(e)){
                return index;
            }
            index++;
            current = current.next;
        }
        return -1;
    }

    @Override
    public int lastIndexOf(E e) {
        Node<E> current = tail;
        int count = this.size - 1;
        while(current != null){ //traverse backwards
            if(current.element == e){
                return count;
            }
            count--;
            current = current.prev;
        }
        return -1;//if not found
```

```
    }

    @Override
    public E remove(int index) {
        if(index < 0 || index >= size){
            return null;
        }
        else if(index == 0){
            return removeFirst();
        }
        else if(index == size - 1){
            return removeLast();
        }
        else{
            Node<E> previous = head;//node before the one to be removed
            for(int i = 1; i < index; i++){//traverse to the node before the one
to be removed
                previous = previous.next;
            }
            if (previous == null || previous.next == null) {
                return null;
            }
            Node<E> current = previous.next;
            Node<E> nextNode = current.next;

            previous.next = nextNode;
            if(nextNode != null){
                nextNode.prev = previous;
            }
            current.next = current.prev = null; //help garbage collection

            size--;
            return current.element;

        }
    }

    public E removeFirst(){
        if(size == 0){
            return null;
        }
        else{
            E temp = head.element;
            head = head.next;
            size--;
```

```
            if(head == null){
                tail = null;
            }
            size--;
            return temp;
        }
    }

    public E removeLast(){
        if(size == 0){
            return null;
        }
        else if(size == 1){
            head = null;
            tail = null;
            return null;
        }
        else{
            Node<E> temp = tail;
            tail = tail.prev;
            if(tail != null)
                tail.next = null;
                size--;
            return temp.element;
        }
    }

    @Override
    public E set(int index, E e) {
        if(index < 0 || index >= size){
            return null;
        }
        else{
            Node<E> current = head;
            for(int i = 0; i < index; i++){
                current = current.next;
            }
            E temp = current.element;
            current.element = e;
            return temp;
        }
    }

    private class Node<E> {
        E element;
```

```
        Node<E> next;
        Node<E> prev;

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

    @Override
    public java.util.Iterator<E> iterator() {
        return new LinkedListIterator();
    }

    private class LinkedListIterator implements java.util.ListIterator<E> {
        private Node<E> current;
        private int nextIndex;

        public LinkedListIterator() {
            current = head;
            nextIndex = 0;
        }

        public LinkedListIterator(int index) {
            if (index < 0 || index >= size)
                throw new IndexOutOfBoundsException("Index: " + index + ", Size:
"
                + size);
            current = head;
            nextIndex = 0;
            for (int nextIndex = 0; nextIndex < index; nextIndex++)
                if(current != null)
                    current = current.next;
        }

        public void setLast() {
            current = tail;
        }

        @Override
        public boolean hasNext() {
            return (current != null);
        }

        @Override
        public E next() {
```

```
            E e = current.element;
            current = current.next;
            nextIndex++;
            return e;
        }

        @Override
        public void remove() {
            Node<E> prevNode = current.prev;
            Node<E> nextNode = current.next;
            if(prevNode == null){ //we are at the head
                head = nextNode;
            }
            else{
                prevNode.next = nextNode;
            }

            if(nextNode == null){
                tail = prevNode;
            }
            else{
                nextNode.prev = prevNode;
            }
            size--;
        }

        @Override
        public void add(E e) {
            Node<E> newNode = new Node<>(e);
            if(head == null){
                head = tail = newNode;
            }
            else{
                newNode.prev = tail;
                tail.next = newNode;
                tail = newNode;
            }
            size++;
        }

        @Override
        public boolean hasPrevious() {
            return current != null;
        }
```

```
        @Override
        public int nextIndex() {
            return nextIndex;
        }

        @Override
        public E previous() {
            E e = current.element;
            current = current.prev;
            // nextIndex--;
            return e;
        }

        @Override
        public int previousIndex() {
            return nextIndex - 1;
        }

        @Override
        public void set(E e) {
            current.element = e; // TODO Auto-generated method stub
        }
    }

    public ListIterator<E> listIterator() {
        return new LinkedListIterator();
    }

    public ListIterator<E> listIterator(int index) {
        return new LinkedListIterator(index);
    }

}
```

Main.java

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        new Main();
    }

    public Main() {
        DoublyLinkedList<Double> list = new DoublyLinkedList<>();
```

```
        System.out.print("Enter five numbers: ");
        Scanner input = new Scanner(System.in);
        double[] v = new double[5];
        for (int i = 0; i < 5; i++)
        v[i] = input.nextDouble();

        list.add(v[1]);
        list.add(v[2]);
        list.add(v[3]);
        list.add(v[4]);
        list.add(0, v[0]);
        list.add(2, 10.55);
        list.remove(3);

        java.util.ListIterator<Double> iterator1 = list.listIterator();
        System.out.print("The list in forward order: ");
        while (iterator1.hasNext())
        System.out.print(iterator1.next() + " ");

        java.util.ListIterator<Double> iterator2 = list.listIterator(list.size()
- 1);
        System.out.print("\nThe list in backward order: ");
        while (iterator2.hasPrevious())
        System.out.print(iterator2.previous() + " ");
    }
}
```

Part 2: Complexity Analysis

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

N: the n in this code dictates the maximum result that the operation “I * 2” can produce.

Since there is only one operation in this code, we can assume that it has a constant time of 1 (1 * c = c, but this is negligible)

The loop doubles I after each iteration (log).

420-301-VA: Programming Patterns
Assignment 1: Implementing Doubly Linked Lists
Fabro, Cyril Efren (6243515)

The time complexity of this code is O(log n)

```
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 ;
}
```

(since there is no n in this code, I will assume that n is array.length)

N represents the length of the array

This code is a nested loop, as it uses two for loops to iterate through n twice (n^2)

There is an if statement, but however we will pick the worst case time complexity

The time complexity of this code is O(n^2)

Singly vs. Doubly Linked List – Insertion

Singly:

- Start at the head of the list
- Traverse through the list index times until you reach the index
- Add the new node at that position

Doubly:

- Depending on the index, you can start at either the head or the tail; if index is closer to head, traverse forward. If index is closer to tail, traverse backwards
- Add the new node at that position by linking it to the two nodes beside it

420-301-VA: Programming Patterns
Assignment 1: Implementing Doubly Linked Lists
Fabro, Cyril Efren (6243515)

Advantage of previous pointers

In the doubly linked list, since you can traverse forwards and backwards in the list, you can reduce the amount of steps needed to reach the index

Singly vs Doubly linked list – Removal

Singly:

- Traverse through the list until your reach the end
- Change the tail to the previous node

Doubly:

- Since we can already access the tail of the list, we just need to reference the tail to the previous node

Difference

Because the singly linked list needs to traverse through the whole list to reach the tail, it will take O(N) to accomplish this task, whereas the doubly linked list only needs to do one operation therefore making O(1)

Part 3: Empirical Performance Comparison

PerformanceTest.java

```
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 <
>() ;
        // --- Adding at the Beginning and Removing at the End ---
        MyLinkedList<Integer> singleListForExtra = new MyLinkedList<>();
        DoublyLinkedList<Integer> doubleListForExtra = 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 .
        timeOperation("Singly: Adding at the Beginning (" + N + " times)", () ->
{
```

420-301-VA: Programming Patterns
Assignment 1: Implementing Doubly Linked Lists
Fabro, Cyril Efren (6243515)

```
            // add to the front of the list for N times
            for(int i = 0; i < N; i++){
                singleListForExtra.addFirst(i);
            }
        });

        timeOperation("Doubly: Adding at the Beginning (" + N + " times)", () ->
{
            // add to the front of the list for N times
            for(int i = 0; i < N; i++){
                doubleListForExtra.addFirst(i);
            }
        });

        timeOperation("Singly: Removing from the End (" + N + " times)", () -> {
            for(int i = N; i >= 0; i--){
                singleListForExtra.removeLast();
            }
        });

        timeOperation("Doubly: Removing from the End (" + N + " times)", () -> {
            for(int i = N; i >= 0; i--){
                doubleListForExtra.removeLast();
            }
        });
    }
}
```

Conducting the following tests

Operation	MyLinkedList (ms)	DoublyLinkedList (ms)
Add at Random Index (M times)	71 ms	74 ms
Add at Beginning (N times)	1 ms	2 ms
Remove from End (N times)	1749 ms	0 ms

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.

The most significant performance difference is removing from the end. This likely do to the fact the singly linked list needs to traverse through the whole list every time it wants to reach the end of the list, where as the doubly linked list can automatically go to the end of the list to remove the element