ALGORITHM AND DATA STRUCTURES
ASSIGNMENT 2 – LINKEDLIST

By:
I Kadek Mahesa Permana Putra
F1D02410052

FACULTY OF ENGINEERING
DEPARTEMENT OF INFORMATICS ENGINEERING
UNIVERSITY OF MATARAM
2025

A. DIRECTORY

Image 1. Project Directory

This is the Project's Directory, this image show the hierarcy of the files used for this program, the **Main** folder contains the main program which is **Spotify.java**, and then the other directory such as **src/Data** used to store the abstract data type used for this program. **src/List** for the Linkedlist library and lastly **src/Nodes** stores the different types of Nodes (Singly, Doubly, and also the parent class Node).

B. SOURCE CODE

NODES

```java
package Nodes;

public abstract class Node {
   protected Object data;

   public Node(Object var1) {
      this.data = var1;
   }

   public Object getData() {
      return this.data;
   }

   public void setData(Object var1) {
      this.data = var1;
   }

   public abstract String toString();
}
```

This is the parent class of Node, called Node.java. this class is abstract so that the child of this class should implements the abstract method inside this class, such as **toString()** method, but originally this method is from super class called **Object** (pre-built in the java programming language), so the child of this class should use the **@Override** notation to differentiate the method.

```java
package Nodes;

public class SinglyNode extends Node {
    public SinglyNode next;

    public SinglyNode(Object data){
        super(data);
        this.next = null;
    }

    public SinglyNode getNext() { return next; }
    public void setNext(SinglyNode next) { this.next = next; }

    @Override
    public String toString() { return data.toString(); }
}
```

The class provides getter and setter methods (**getNext()** and s**etNext()**) to safely access and modify the next reference. The **toString()** method is overridden to return the string representation of the node's data, which makes printing nodes easier when displaying the list.

```java
package Nodes;

public class DoublyNode extends Node {
    public DoublyNode next;
    public DoublyNode prev;

    public DoublyNode(Object data){
        super(data);
        this.next = null;
        this.prev = null;
    }

    public DoublyNode getNext() { return next; }
    public DoublyNode getPrev() { return prev; }
    public void setNext(DoublyNode next) { this.next = next; }
    public void setPrev(DoublyNode prev) { this.prev = prev; }

    @Override
    public String toString() { return data.toString(); }
}
```

Basically in this class (DoublyNode and SinglyNode) there are no big different between them, except the point to the next and previous Node for DoublyNode (which in SinglyNode only next).

LINKEDLISTS

```java
package List;

public abstract class LinkedList {
    public int size;
    public LinkedList() { this.size = 0; }

    public int getSize() { return size; }
    public boolean isEmpty() { return size == 0; }

    // Abstract methods that must be implemented
    public abstract void insertFirst(Object data);
    public abstract void insertLast(Object data);
    public abstract void insertAt(int index, Object data);
    public abstract Object deleteFirst();
    public abstract Object deleteLast();
    public abstract Object deleteAt(int index);
    public abstract Object get(int index);
```

```
    public abstract int search(Object data);
    public abstract void display();
    public abstract void clear();
}
```

Image above shows the parent class called Linkedlist, there are so many method to implements, which this method's parameter is Object class to store multiple data types (any kind)

```
package List;
import Nodes.SinglyNode;

public class SinglyLinkedList extends LinkedList {
    public SinglyNode head;

    public SinglyLinkedList() {
        super();
        this.head = null;
    }

    @Override
    public void insertFirst(Object data) {
        SinglyNode newNode = new SinglyNode(data);
        newNode.setNext(head);
        head = newNode;
        size++;
    }

    @Override
    public void insertLast(Object data) {
        SinglyNode newNode = new SinglyNode(data);
        if (head == null) {
            head = newNode;
        } else {
            SinglyNode current = head;
            while (current.getNext() != null) {
                current = current.getNext();
            }
            current.setNext(newNode);
        }
        size++;
    }

    @Override
    public void insertAt(int index, Object data) {
        if (index < 0 || index > size) {
            throw new IndexOutOfBoundsException("Index out of bounds");
        }
```

```
        if (index == 0) {
            insertFirst(data);
            return;
        }

        SinglyNode newNode = new SinglyNode(data);
        SinglyNode current = head;
        for (int i = 0; i < index - 1; i++) {
            current = current.getNext();
        }
        newNode.setNext(current.getNext());
        current.setNext(newNode);
        size++;
    }

    @Override
    public Object deleteFirst() {
        if (head == null) {
            return null;
        }
        Object data = head.getData();
        head = head.getNext();
        size--;
        return data;
    }

    @Override
    public Object deleteLast() {
        if (head == null) {
            return null;
        }

        if (head.getNext() == null) {
            Object data = head.getData();
            head = null;
            size--;
            return data;
        }

        SinglyNode current = head;
        while (current.getNext().getNext() != null) {
            current = current.getNext();
        }
        Object data = current.getNext().getData();
        current.setNext(null);
        size--;
        return data;
    }
```

```java
@Override
public Object deleteAt(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException("Index out of bounds");
    }

    if (index == 0) {
        return deleteFirst();
    }

    SinglyNode current = head;
    for (int i = 0; i < index - 1; i++) {
        current = current.getNext();
    }
    Object data = current.getNext().getData();
    current.setNext(current.getNext().getNext());
    size--;
    return data;
}

@Override
public Object get(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException("Index out of bounds");
    }

    SinglyNode current = head;
    for (int i = 0; i < index; i++) {
        current = current.getNext();
    }
    return current.getData();
}

@Override
public int search(Object data) {
    SinglyNode current = head;
    int index = 0;
    while (current != null) {
        if (current.getData().equals(data)) {
            return index;
        }
        current = current.getNext();
        index++;
    }
    return -1;
}
```

```
    @Override
    public void display() {
        if (head == null) {
            System.out.println("List is empty");
            return;
        }

        SinglyNode current = head;
        System.out.print("Singly LinkedList: ");
        while (current != null) {
            System.out.print(current.getData());
            if (current.getNext() != null) {
                System.out.print(" -> ");
            }
            current = current.getNext();
        }
        System.out.println(" -> null");
    }

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

The code above is an implementation of a Singly Linked List in Java, which extends the **LinkedList** class. A Singly Linked List is a linear data structure where each node stores data and a reference to the next node (next), but unlike a doubly linked list, it does not keep a reference to the previous node. This means traversal can only be done in one direction, from the head toward the end of the list.

```
package List;
import Nodes.DoublyNode;

public class DoublyLinkedList extends LinkedList {
    public DoublyNode head;
    public DoublyNode tail;

    public DoublyLinkedList(){
        super();
        this.head = null;
        this.tail = null;
    }

    @Override
    public void insertFirst(Object data){
        DoublyNode newNode = new DoublyNode(data);
```

```
        if(head == null){
            head = tail = newNode;
        } else {
            newNode.setNext(head);
            head.setPrev(newNode);
            head = newNode;
        }
        size++;
    }

    @Override
    public void insertLast(Object data){
        DoublyNode newNode = new DoublyNode(data);
        if(tail == null){
            head = tail =  newNode;
        } else {
            tail.setNext(newNode);
            newNode.setPrev(tail);
            tail = newNode;
        }
        size++;
    }

    @Override
    public void insertAt(int index, Object data){
        if(index < 0 || index > size){
            throw new IndexOutOfBoundsException("Index out of bounds");
        }

        if(index == 0) { insertFirst(data); return; }
        if(index == size) { insertLast(data); return; }

        DoublyNode newNode = new DoublyNode(data);
        DoublyNode current = head;
        for(int i = 0; i < index; i++){
            current = current.getNext();
        }

        newNode.setPrev(current.getPrev());
        newNode.setNext(current);
        current.getPrev().setNext(newNode);
        current.setPrev(newNode);
        size++;
    }

    @Override
    public Object deleteFirst() {
        if (head == null) { return null; }
```

```
        Object data = head.getData();
        if (head == tail) {
            head = tail = null;
        } else {
            head = head.getNext();
            head.setPrev(null);
        }
        size--;
        return data;
    }

    @Override
    public Object deleteLast() {
        if (tail == null) { return null; }

        Object data = tail.getData();
        if (head == tail) {
            head = tail = null;
        } else {
            tail = tail.getPrev();
            tail.setNext(null);
        }
        size--;
        return data;
    }

    @Override
    public Object deleteAt(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException("Index out of bounds");
        }

        if (index == 0) { return deleteFirst(); }

        if (index == size - 1) { return deleteLast(); }

        DoublyNode current = head;
        for (int i = 0; i < index; i++) {
            current = current.getNext();
        }

        Object data = current.getData();
        current.getPrev().setNext(current.getNext());
        current.getNext().setPrev(current.getPrev());
        size--;
        return data;
    }
```

```java
@Override
public Object get(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException("Index out of bounds");
    }

    DoublyNode current;
    if (index < size / 2) {
        current = head;
        for (int i = 0; i < index; i++) {
            current = current.getNext();
        }
    } else {
        current = tail;
        for (int i = size - 1; i > index; i--) {
            current = current.getPrev();
        }
    }
    return current.getData();
}

@Override
public int search(Object data) {
    DoublyNode current = head;
    int index = 0;
    while (current != null) {
        if (current.getData().equals(data)) {
            return index;
        }
        current = current.getNext();
        index++;
    }
    return -1;
}

@Override
public void display() {
    if (head == null) {
        System.out.println("List is empty");
        return;
    }

    DoublyNode current = head;
    System.out.print("Doubly LinkedList: null <- ");
    while (current != null) {
        System.out.print(current.getData());
        if (current.getNext() != null) {
```

```
                System.out.print(" <-> ");
            }
            current = current.getNext();
        }
        System.out.println(" -> null");
    }

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

    //Exclusif method for this class
    public void displayReverse() {
        if (tail == null) {
            System.out.println("List is empty");
            return;
        }

        DoublyNode current = tail;
        System.out.print("Doubly LinkedList (Reverse): null <- ");
        while (current != null) {
            System.out.print(current.getData());
            if (current.getPrev() != null) {
                System.out.print(" <-> ");
            }
            current = current.getPrev();
        }
        System.out.println(" -> null");
    }
}
```

The code above is an implementation of the Doubly Linked List data structure in Java, extending a **LinkedList** class. A Doubly Linked List is a type of linked list where each node stores two references, one pointing to the next node (next) and another pointing to the previous node (prev). This allows traversal in both directions, from the head to the tail and vice versa. The **DoublyLinkedList** class uses **DoublyNode** objects as nodes to hold data, along with head (the first element) and tail (the last element) references. It provides a complete set of methods to manipulate the list. Overall, this code demonstrates how a Doubly Linked List works efficiently for insertion, deletion, and bidirectional traversal. It also shows proper index validation to prevent errors when accessing out-of-range positions.

```
package List;
import Nodes.SinglyNode;
```

```java
public class CircularLinkedList extends LinkedList {
    public SinglyNode tail;

    public CircularLinkedList() {
        super();
        this.tail = null;
    }

    @Override
    public void insertFirst(Object data) {
        SinglyNode newNode = new SinglyNode(data);
        if (tail == null) {
            tail = newNode;
            newNode.setNext(newNode);
        } else {
            newNode.setNext(tail.getNext());
            tail.setNext(newNode);
        }
        size++;
    }

    @Override
    public void insertLast(Object data) {
        SinglyNode newNode = new SinglyNode(data);
        if (tail == null) {
            tail = newNode;
            newNode.setNext(newNode);
        } else {
            newNode.setNext(tail.getNext());
            tail.setNext(newNode);
            tail = newNode;
        }
        size++;
    }

    @Override
    public void insertAt(int index, Object data) {
        if (index < 0 || index > size) {
            throw new IndexOutOfBoundsException("Index out of bounds");
        }

        if (index == 0) {
            insertFirst(data);
            return;
        }

        if (index == size) {
            insertLast(data);
```

```
            return;
        }

        SinglyNode newNode = new SinglyNode(data);
        SinglyNode current = tail.getNext();
        for (int i = 0; i < index - 1; i++) {
            current = current.getNext();
        }
        newNode.setNext(current.getNext());
        current.setNext(newNode);
        size++;
    }

    @Override
    public Object deleteFirst() {
        if (tail == null) {
            return null;
        }

        Object data;
        if (tail.getNext() == tail) {
            data = tail.getData();
            tail = null;
        } else {
            SinglyNode head = tail.getNext();
            data = head.getData();
            tail.setNext(head.getNext());
        }
        size--;
        return data;
    }

    @Override
    public Object deleteLast() {
        if (tail == null) {
            return null;
        }

        Object data = tail.getData();
        if (tail.getNext() == tail) {
            tail = null;
        } else {
            SinglyNode current = tail.getNext();
            while (current.getNext() != tail) {
                current = current.getNext();
            }
            current.setNext(tail.getNext());
            tail = current;
```

```
        }
        size--;
        return data;
    }

    @Override
    public Object deleteAt(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException("Index out of bounds");
        }

        if (index == 0) {
            return deleteFirst();
        }

        if (index == size - 1) {
            return deleteLast();
        }

        SinglyNode current = tail.getNext();
        for (int i = 0; i < index - 1; i++) {
            current = current.getNext();
        }
        Object data = current.getNext().getData();
        current.setNext(current.getNext().getNext());
        size--;
        return data;
    }

    @Override
    public Object get(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException("Index out of bounds");
        }

        SinglyNode current = tail.getNext();
        for (int i = 0; i < index; i++) {
            current = current.getNext();
        }
        return current.getData();
    }

    @Override
    public int search(Object data) {
        if (tail == null) {
            return -1;
        }
```

```
        SinglyNode current = tail.getNext();
        int index = 0;
        do {
            if (current.getData().equals(data)) {
                return index;
            }
            current = current.getNext();
            index++;
        } while (current != tail.getNext());
        return -1;
    }

    @Override
    public void display() {
        if (tail == null) {
            System.out.println("List is empty");
            return;
        }

        SinglyNode current = tail.getNext();
        System.out.print("Circular LinkedList: ");
        do {
            System.out.print(current.getData());
            current = current.getNext();
            if (current != tail.getNext()) {
                System.out.print(" -> ");
            }
        } while (current != tail.getNext());
        System.out.println(" -> (back to first)");
    }

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

The code above is an implementation of a Circular Singly Linked List in Java, which extends the **LinkedList** class. Unlike a regular singly linked list, the circular version connects the last node back to the first node, creating a closed loop. Instead of maintaining a head, this implementation uses a tail reference, where **tail.getNext()** always points to the first node in the list. This design allows traversal to wrap around seamlessly without reaching a null reference.

DATA (SONG CLASS)

```
package Data;
```

```java
public abstract class Song {
    public String title;
    public String artist;
    public String album;
    public int duration;
    public String genre;

    public Song(String title, String artist, String album, int duration,
String genre) {
        this.title = title;
        this.artist = artist;
        this.album = album;
        this.duration = duration;
        this.genre = genre;
    }

    public abstract void play();
    public abstract void pause();
    public abstract void stop();

    public String getTitle() { return title; }
    public String getArtist() { return artist; }
    public String getAlbum() { return album; }
    public int getDuration() { return duration; }
    public String getGenre() { return genre; }

    public void setTitle(String title) { this.title = title; }
    public void setArtist(String artist) { this.artist = artist; }
    public void setAlbum(String album) { this.album = album; }
    public void setDuration(int duration) { this.duration = duration; }
    public void setGenre(String genre) { this.genre = genre; }

    public String getFormattedDuration() {
        int minutes = duration / 60;
        int seconds = duration % 60;
        return String.format("%d:%02d", minutes, seconds);
    }

    @Override
    public String toString() {
        return String.format("%s - %s | %s | %s | %s",
                             title, artist, album, getFormattedDuration(),
genre);
    }
}
```

This code defines an abstract class Song that serves as a blueprint for representing a music track in a program. Because it is declared as abstract, it cannot be instantiated directly,

instead, other classes must extend it and provide concrete implementations of its abstract methods. In summary, the Song class defines a **general framework for handling song data and playback behaviors**. It separates *what* every song should have and do (attributes and abstract methods) from *how* those actions are carried out, leaving flexibility for different playback systems.

```
package Data;

public class SpotifySong extends Song {
    public boolean isPlaying;

    public SpotifySong(String title, String artist, String album, int
duration, String genre) {
        super(title, artist, album, duration, genre);
        this.isPlaying = false;
    }

    @Override
    public void play() {
        isPlaying = true;
        System.out.println("🎵 Playing: " + title + " - " + artist);
    }

    @Override
    public void pause() {
        isPlaying = false;
        System.out.println("⏸  Paused: " + title + " - " + artist);
    }

    @Override
    public void stop() {
        isPlaying = false;
        System.out.println("⏹  Stopped: " + title + " - " + artist);
    }

    public boolean isPlaying() { return isPlaying; }

}
```

This code defines the **SpotifySong** class, which extends the abstract Song class and provides concrete implementations for its abstract methods (play, pause, and stop). It simulates the behavior of a song object within a music player environment.

MAIN CLASS

```
package Main;

import Data.SpotifySong;
import List.*;
import java.util.Scanner;

public class Spotify {
    public static void main(String[] args) {
        DoublyLinkedList mainPlaylist = new DoublyLinkedList();
        SinglyLinkedList recentlyPlayed = new SinglyLinkedList();
        CircularLinkedList loopPlaylist = new CircularLinkedList();

        // Sample for demonstration
        SpotifySong sample1 = new SpotifySong("Cruel Angel's Thesis", "Yoko
Takahashi", "Neon Genesis Evangelion", 240, "Anime");
        mainPlaylist.insertLast(sample1);
        loopPlaylist.insertLast(sample1);
        SpotifySong sample2 = new SpotifySong("Tank!", "SEATBELTS", "Cowboy
Bebop", 180, "Anime");
        mainPlaylist.insertLast(sample2);
        loopPlaylist.insertLast(sample2);
        SpotifySong sample3 = new SpotifySong("Blue Bird", "Ikimonogakari",
"Naruto Shippuden", 250, "Anime");
        mainPlaylist.insertLast(sample3);
        loopPlaylist.insertLast(sample3);
        SpotifySong sample4 = new SpotifySong("Butterfly", "Kouji Wada",
"Digimon Adventure", 220, "Anime");
        mainPlaylist.insertLast(sample4);
        loopPlaylist.insertLast(sample4);
        SpotifySong sample5 = new SpotifySong("Unravel", "TK from Ling
Tosite Sigure", "Tokyo Ghoul", 230, "Anime");
        mainPlaylist.insertLast(sample5);
        loopPlaylist.insertLast(sample5);

        Scanner scanner = new Scanner(System.in);
        boolean running = true;

        while (running) {
            System.out.println("\n" + "=".repeat(40));
            System.out.println("         Spotify Program Menu");
            System.out.println("=".repeat(40));
            System.out.println("1. Add Song");
            System.out.println("2. View Playlists");
            System.out.println("3. Play Music");
            System.out.println("4. Exit");
            System.out.println("-".repeat(40));
            System.out.print("Choose an option: ");
```

```
            int choice = scanner.nextInt();
            scanner.nextLine(); // consume newline

            switch (choice) {
                case 1:
                    addSongMenu(scanner, mainPlaylist, recentlyPlayed,
loopPlaylist);
                    break;
                case 2:
                    viewPlaylistsMenu(scanner, mainPlaylist, recentlyPlayed,
loopPlaylist);
                    break;
                case 3:
                    playMusicMenu(scanner, mainPlaylist, recentlyPlayed,
loopPlaylist);
                    break;
                case 4:
                    running = false;
                    System.out.println("Exiting Spotify. Goodbye!");
                    break;
                default:
                    System.out.println("Invalid option. Please try again.");
            }
        }
        scanner.close();
    }

    // Helper method to create a new song
    private static SpotifySong createSong(Scanner scanner) {
        System.out.println("\n" + "-".repeat(30));
        System.out.println("Enter Song Details:");
        System.out.println("-".repeat(30));
        System.out.print("Title: ");
        String title = scanner.nextLine();
        System.out.print("Artist: ");
        String artist = scanner.nextLine();
        System.out.print("Album: ");
        String album = scanner.nextLine();
        System.out.print("Duration (seconds): ");
        int duration = scanner.nextInt();
        scanner.nextLine();
        System.out.print("Genre: ");
        String genre = scanner.nextLine();
        System.out.println("-".repeat(30));
        return new SpotifySong(title, artist, album, duration, genre);
    }
```

```
    private static void playSong(DoublyLinkedList playlist, Scanner scanner,
SinglyLinkedList recentlyPlayed) {
        System.out.print("Enter index to play: ");
        int index = scanner.nextInt();
        scanner.nextLine();
        try {
            SpotifySong song = (SpotifySong) playlist.get(index);
            song.play();
            recentlyPlayed.insertLast(song);
        } catch (IndexOutOfBoundsException e) {
            System.out.println("Invalid index.");
        }
    }

    private static void pauseSong(DoublyLinkedList playlist, Scanner
scanner) {
        System.out.print("Enter index to pause: ");
        int index = scanner.nextInt();
        scanner.nextLine();
        try {
            SpotifySong song = (SpotifySong) playlist.get(index);
            song.pause();
        } catch (IndexOutOfBoundsException e) {
            System.out.println("Invalid index.");
        }
    }

    private static void stopSong(DoublyLinkedList playlist, Scanner scanner)
{
        System.out.print("Enter index to stop: ");
        int index = scanner.nextInt();
        scanner.nextLine();
        try {
            SpotifySong song = (SpotifySong) playlist.get(index);
            song.stop();
        } catch (IndexOutOfBoundsException e) {
            System.out.println("Invalid index.");
        }
    }

    private static void deleteSong(DoublyLinkedList playlist, Scanner
scanner) {
        System.out.print("Enter index to delete: ");
        int index = scanner.nextInt();
        scanner.nextLine();
        try {
            playlist.deleteAt(index);
            System.out.println("Song deleted.");
```

```
        } catch (IndexOutOfBoundsException e) {
            System.out.println("Invalid index.");
        }
    }

    private static void loopPlay(CircularLinkedList playlist) {
        if (playlist.size == 0) {
            System.out.println("Loop Playlist is empty.");
            return;
        }
        System.out.println("Looping through Loop Playlist:");
        for (int i = 0; i < playlist.size; i++) {
            SpotifySong song = (SpotifySong) playlist.get(i);
            song.play();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        System.out.println("Loop finished.");
    }

    private static void addSongMenu(Scanner scanner, DoublyLinkedList
mainPlaylist, SinglyLinkedList recentlyPlayed, CircularLinkedList
loopPlaylist) {
        System.out.println("\n" + "-".repeat(30));
        System.out.println("Add Song to Playlist");
        System.out.println("-".repeat(30));
        System.out.println("1. Main Playlist (Doubly Linked List)");
        System.out.println("2. Recently Played (Singly Linked List)");
        System.out.println("3. Loop Playlist (Circular Linked List)");
        System.out.println("4. Back to Main Menu");
        System.out.print("Choose a playlist: ");
        int choice = scanner.nextInt();
        scanner.nextLine();
        switch (choice) {
            case 1:
                mainPlaylist.insertLast(createSong(scanner));
                System.out.println("Song added to Main Playlist.");
                break;
            case 2:
                recentlyPlayed.insertLast(createSong(scanner));
                System.out.println("Song added to Recently Played.");
                break;
            case 3:
                loopPlaylist.insertLast(createSong(scanner));
                System.out.println("Song added to Loop Playlist.");
```

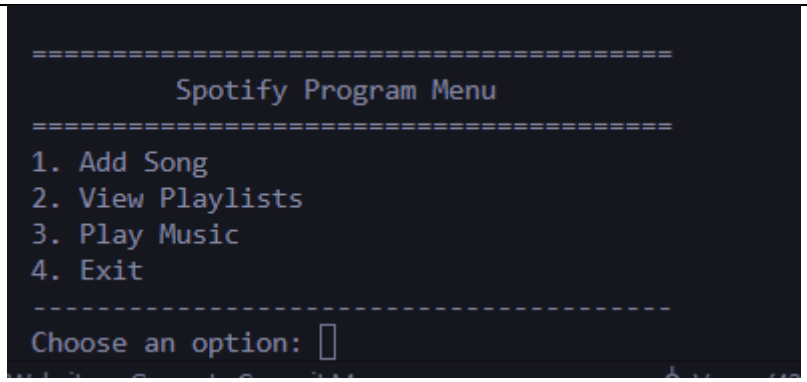
```
                break;
            case 4:
                break;
            default:
                System.out.println("Invalid option. Returning to main
menu.");
        }
    }

    private static void describeLoopPlaylist(CircularLinkedList playlist) {
        if (playlist.size == 0) {
            System.out.println("Loop Playlist is empty.");
            return;
        }
        System.out.println("Loop Playlist:");
        for (int i = 0; i < playlist.size; i++) {
            SpotifySong song = (SpotifySong) playlist.get(i);
            System.out.println((i + 1) + ". " + song.toString());
        }
    }

    private static void viewPlaylistsMenu(Scanner scanner, DoublyLinkedList
mainPlaylist, SinglyLinkedList recentlyPlayed, CircularLinkedList
loopPlaylist) {
        System.out.println("\n" + "-".repeat(30));
        System.out.println("View Playlists");
        System.out.println("-".repeat(30));
        System.out.println("1. Main Playlist");
        System.out.println("2. Recently Played");
        System.out.println("3. Loop Playlist");
        System.out.println("4. Back to Main Menu");
        System.out.print("Choose a playlist to view: ");
        int choice = scanner.nextInt();
        scanner.nextLine();
        switch (choice) {
            case 1: mainPlaylist.display();
                break;
            case 2: recentlyPlayed.display();
                break;
            case 3: describeLoopPlaylist(loopPlaylist);
                break;
            case 4:
                break;
            default:
                System.out.println("Invalid option. Returning to main
menu.");
        }
    }
```

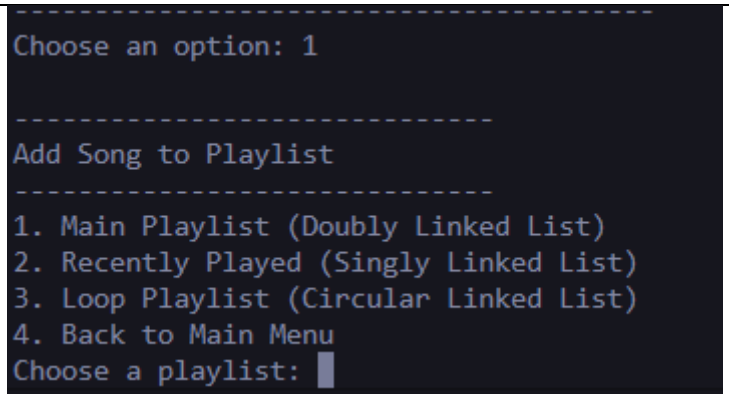
```
    private static void playMusicMenu(Scanner scanner, DoublyLinkedList
mainPlaylist, SinglyLinkedList recentlyPlayed, CircularLinkedList
loopPlaylist) {
        System.out.println("\n" + "-".repeat(30));
        System.out.println("Play Music");
        System.out.println("-".repeat(30));
        System.out.println("1. Play song from Main Playlist");
        System.out.println("2. Pause song from Main Playlist");
        System.out.println("3. Stop song from Main Playlist");
        System.out.println("4. Delete song from Main Playlist");
        System.out.println("5. Loop play from Loop Playlist");
        System.out.println("6. Back to Main Menu");
        System.out.print("Choose an option: ");
        int choice = scanner.nextInt();
        scanner.nextLine();
        switch (choice) {
            case 1: playSong(mainPlaylist, scanner, recentlyPlayed);
                break;
            case 2: pauseSong(mainPlaylist, scanner);
                break;
            case 3: stopSong(mainPlaylist, scanner);
                break;
            case 4: deleteSong(mainPlaylist, scanner);
                break;
            case 5: loopPlay(loopPlaylist);
                break;
            case 6:
                break;
            default:
                System.out.println("Invalid option. Returning to main
menu.");
        }
    }
}
```

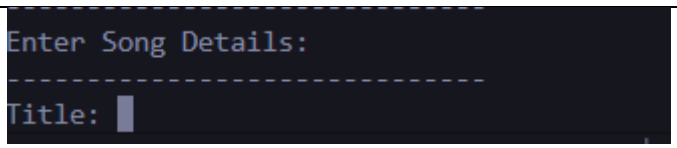
Three types of playlists are created, a **DoublyLinkedList** for the main playlist, a **SinglyLinkedList** for recently played songs, and a **CircularLinkedList** for the loop playlist. These data structures were chosen because they reflect different playlist behaviors, the main playlist allows flexible navigation and deletion, the recently played list simply tracks what has been listened to, and the loop playlist ensures continuous playback. At the start of the program, a few sample *SpotifySong* objects are created and inserted into both the main playlist and the loop playlist. This ensures that the application has initial data for demonstration, so the user can immediately interact with the menu without needing to add songs first. The application then enters a loop where it continuously displays a menu with four options, *adding* a song,

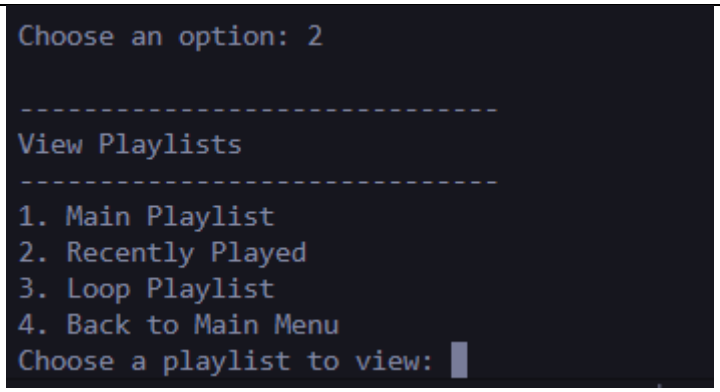
viewing playlists, *playing* music, or *exiting* the program. The user interacts with the menu through console input via a **Scanner**. If the user chooses to add a song, they are prompted to input song details such as title, artist, album, duration, and genre, after which a new *SpotifySong* object is created and added to the chosen playlist. Viewing playlists lets the user inspect the contents of the main, recently played, or loop playlist, with each being displayed differently depending on its type. The play music menu allows the user to play, pause, or stop songs from the main playlist by specifying the song's index. When a song is played, it is also added to the recently played list for tracking. Additionally, users can delete songs from the main playlist, or trigger the loop playlist to simulate playing each song in order with a short pause between tracks. The program remains active until the user selects the exit option, after which it terminates gracefully with a closing message.

C. RESULT

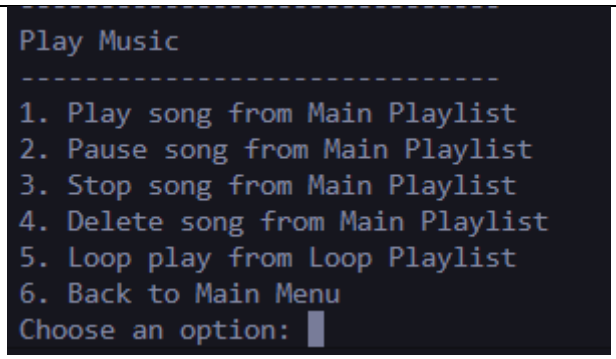


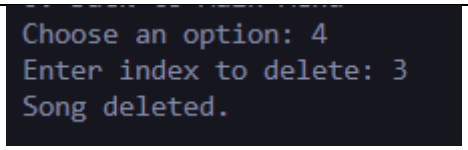





Choose an option: 2
View Playlists
1. Main Playlist
2. Recently Played
3. Loop Playlist
4. Back to Main Menu
Choose a playlist to view:


Choose a playlist to view: 3
Loop Playlist:
1. Cruel Angel's Thesis - Yoko Takahashi | Neon Genesis Evangelion | 4:00 | Anime
2. Tank! - SEATBELTS | Cowboy Bebop | 3:00 | Anime
3. Blue Bird - Ikimonogakari | Naruto Shippuden | 4:10 | Anime
4. Butterfly - Kouji Wada | Digimon Adventure | 3:40 | Anime
5. Unravel - TK from Ling Tosite Sigure | Tokyo Ghoul | 3:50 | Anime
Spotify Program Menu
1. Add Song
2. View Playlists
3. Play Music
4. Exit
Choose an option:


Play Music
1. Play song from Main Playlist
2. Pause song from Main Playlist
3. Stop song from Main Playlist
4. Delete song from Main Playlist
5. Loop play from Loop Playlist
6. Back to Main Menu
Choose an option:


Choose an option: 4
Enter index to delete: 3
Song deleted.

Overall, this program demonstrates the practical implementation of all major variations of linked lists namely singly linked lists, doubly linked lists, and circular linked lists within the context of a music playlist system. Each type of list is applied to a specific use case the singly linked list manages the “recently played” songs, the doubly linked list acts as the main playlist with flexible insertion and deletion in both directions, and the circular linked list handles continuous loop playback. Across these structures, the program makes use of core linked list algorithms such as insertion at the beginning, end, or a specific index; deletion of nodes at various positions; searching for elements; retrieving items by index; traversing and displaying nodes; and clearing entire lists. This not only showcases how fundamental linked list operations can be implemented in Java, but also illustrates how data structures can be mapped to real-world applications like music management systems.