MuSearch

Version 0.97

April 11th 2018
Group #20

SFWR 2XB3
Department of Computing and Software
McMaster University

Edward Liu
Connor Czarnuch
Bilal Jaffry
Stephanus Jonatan

Revision History

Report Version I

- Added contributions page and title page of documentation specification.

Report Version II

- UML Class diagrams generated and added to documentation specification.

Report Version III

- Executive Summary added with proper Table of Contents outlining documentation on project specification.

Report Version IV

- Each class description of the interface and semantics for each public method in each class.
- Uses relationship and traceback for requirements for each class interface.

Report Version V

- Internal review/evaluation of design document added to document specification.
- Report history page added to document changes in specification.

Team Members	Student Number	Roles and Responsibilities
Edward Liu	400079479	Back end, Sorting algorithm, Graphing algorithm
Bilal Jaffry	400067222	Front end, Searching algorithm, Project facilitator
Stephanus Jonatan	400072396	Front end, Sorting algorithm, Project log
Connor Czarnuch	400053071	Back end, Parsing dataset, Hashing algorithm

By virtue of submitting this document we electronically sign and date that the work being submitted by all the individuals in the group is their exclusive work as a group and we consent to make available the application developed through [CS] or [SE]-2XB3 project, the reports, presentations, and assignments (not including my name and student number) for future teaching purposes.

Contribution Outline

Name	Role(s)	Contributions	Comments
Bilal	Front end, Searching algorithm, Project facilitator	Quicksort for list Binary search changes	Made changes to increase performance of binary search
Stephanus	Front end, Sorting algorithm, Project log	Binary Search for list Quicksort changes	Made changes to mergesort to switch to quicksort
Connor	Back end, Parsing dataset, Hashing algorithm	Parsing dataset Implemented hash algorithm File input	Increased run time by increasing performance for reading.
Edward	Back end, Sorting algorithm, Graphing algorithm	User and Song ADT Implemented graph algorithm	Increased general run time by increasing performance for graphing.

Executive Summary

MuSearch is a software application designed for music recommendation based on a open data set of listeners top listened songs. This software uses collaborative filtering to generate a list of recommendations of songs, employing a method that is different from other music based software recommendation tools over the years such as Last.fm, Spotify etc. The software bases its recommendations on the frequency of which the songs other people who have listened to the most from the given song. The user inputs a song and/or given artist and the software would give you a list of songs to choose for the search based on the given input. This input parsed in is then searches through the database of over forty million nodes and edges of songs/artists to find the best recommendations for the user. The database, EchoNest is similar to the database that big companies - such as Spotify - use to recommend songs to their users.

Table of Content

Page Number

Description of Classes

Containers

Bag.java

- Used in Graph.java to act as a container to hold the adjacency list of the graph being constructed.

Queue.java

- Used in IdGetter.java to queue Song objects. In IdGetter.java, the method addSame() uses a queue to add all similar songs for the song name and artist passed into addSame.

Data Types

Recommendaton.java

- Abstract Data type to store the recommendation for the song/artists with number matches found for the given song.

Song.java

- A Song abstract data type used to represent a song with attributes such as a song ID (used to differentiate songs from other songs with similar names and/or artists), song name, artist, album, and the date it was released. All aforementioned attributes are included in the databases used.

Songs.java

- Stores a list of all songs from input file, contains an iterator for a list of songs. Run's the Sort.sort() method on the list of Songs from the input the file and consequently runs binary search to find the respective songID. Establishes the ability to obtain the song ID's from the song names, and vice versa.

Graph

Graph.java

- Used to graph songs with its neighbors as users who listen to these songs. The neighbors of a user are all the songs that a particular user listens to.

NeighborsDFS.java

- Since the neighbors of a song are users, and the neighbors of users are songs, NeighborsDFS was implemented to perform DFS on a song vertex, but stops after it reaches a depth of two, to find all the other songs that each user - connected to song A - listens to.

Helper

IdGetter.java

- After the user inputs a song and/or artist, there is a problem if the entered song and/or artist has similar song or artist names. To avoid this problem, IdGetter uses the method addSame to gather all similar songs and artists into a queue. The elements are then displayed to the user and asks for input on which song

the user wants recommendations for. After the selection, the method getId retrieves the song in the queue, and returns the song ID in order to perform DFS on it.

RecommendSongsDFS.java

- Implements a depth first search on the graph of song nodes using the NeighbourDFS and a Graph object of all the songs. The search returns a given number of recommendations from a Queue of songs list in the depth first search.

Prog

MuSearch.java

- Used to run the entire MuSearch java project and display any necessary information to the user.

Read

Read.java

- Used to parse through the databases (song_data.csv, train_triplets.txt)

Search

Search.java

- Implements a binary search to search through a sorted list of Song IDs.

Sort

Sort.java

- Implements quick sort algorithm to sort the song IDs after reading the databases.

Stopwatch

Stopwatch.java

- Used to time how long it takes to read through the data and construct the graph

SymbolTables

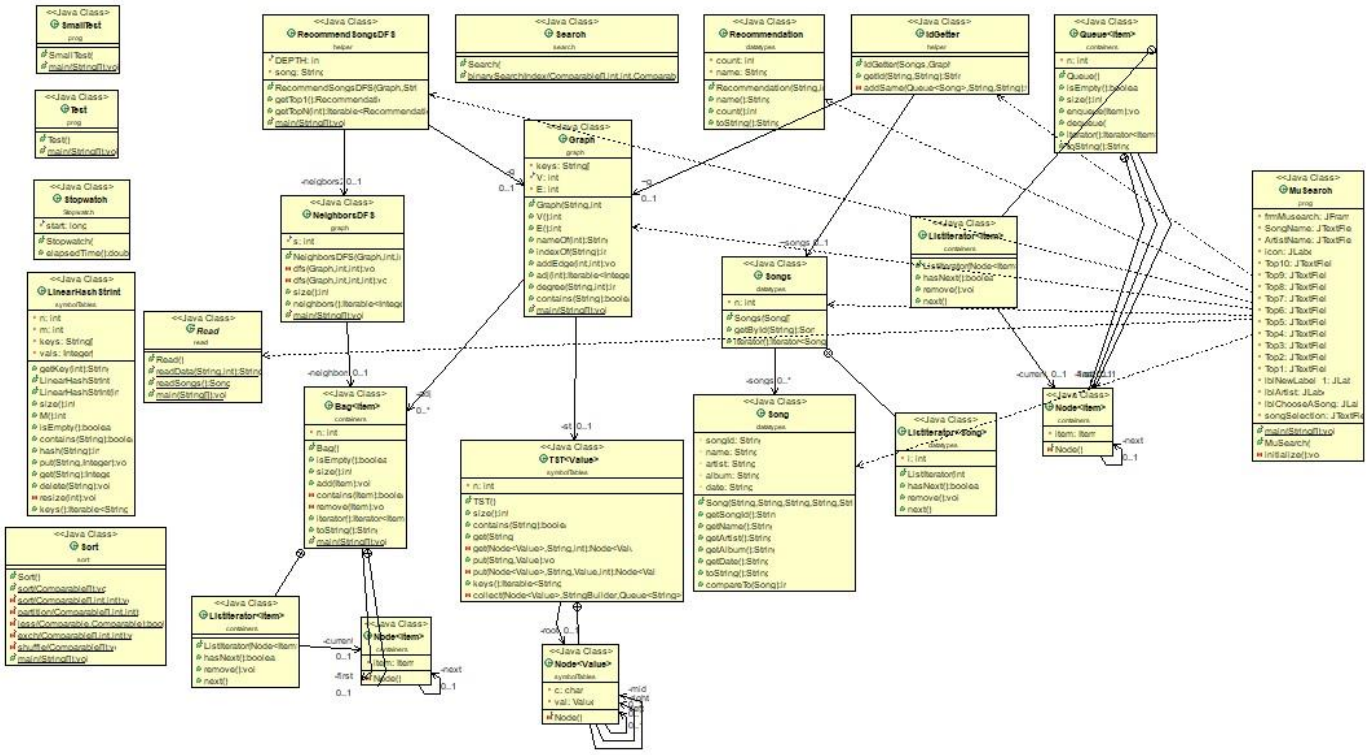
LinearHashStrInt.java

- Hashes song IDs into a symbol table

TST.java

- It provides a character based method to find strings in a symbol table based on the symbol table of a given prefix value and find all strings in the symbol table. This is done using a Ternary Search Trie.

MuSearch UML Diagram



Interface Specification of Methods

Bag

Variables:

Private Node<Item> first
- Beginning of a bag

Private int n
- Number of elements in a bag

Methods:

Public Bag()
- Constructor for a Bag object; initializes an empty bag

Public boolean isEmpty()
- Returns true if this bag is empty

Public int size()
- Returns the number of items in this bag; returns variable n

Public void add(Item item)
- Adds an item to the bag
- Parameter item is the item to add into a bag

Private boolean contains(Item item)
- Checks if an item is in the bag. Returns true if it is, else false.
- Parameter item is the item being checked for its existence in the bag.

Private void remove(Item item)
- Removes an item in a bag
- Parameter item is the item being removed from the bag

Private class ListIterator<Item> implements Iterator<Item>
- A subclass for bag to produce an iterator of object type Item.

Attributes

Private Node<Item> current
- Current is the last item added to the ListIterator object type

Methods:

Public ListIterator(Node<item> first)
- ListIterator constructor
- Parameter first is the first item inserted into the list

Public boolean hasNext()
- Returns true if there is an item in the list

Public void remove()
- Is not implemented since it is optional

Public Item next()
- Returns the next item in the list

Public String toString()
- Returns all the items in the list to a string

Queue

Variables:

private Node<Item> first;

- The beginning element of a queue

private Node<Item> last;

- The last element of a queue

private int n;

- The number of elements on a queue

Methods:

Private class Node<Item>

- A subclass of Queue used to represent an element or node in a queue

Attributes

Private Item item

- The item of the node

Private Node<Item> next;

- The next node after the current node

Public Queue

- The constructor for a Queue object
- Initializes variables first, last, and n to null, null, and 0

Public boolean isEmpty()

- Return true if the queue is empty

Public int size()

- Returns the size of the queue; returns variable n

Public void enqueue(Item item)

- Adds the item into the queue as the last element
- Parameter item is the item being added into the queue

Public void dequeue()

- Removes the first item in the queue

Public Iterator<Item> iterator()

- Returns a ListIterator<Item> with the attribute first as the current value

Private class ListIterator<Item> implements Iterator<Item>

- A subclass for bag to produce an iterator of object type Item.

  Attributes

    Private Node<Item> current
    - Current is the last item added to the ListIterator object type

  Methods:

    Public ListIterator(Node<item> first)
    - ListIterator constructor
    - Parameter first is the first item inserted into the list

    Public boolean hasNext()
    - Returns true if there is an item in the list

    Public void remove()
    - Is not implemented since it is optional

    Public Item next()
    - Returns the next item in the list

    Public String toString()
    - Returns all the items in the list to a string

IdGetter

Variables:

Songs songs
- Holds the songs sorted by their song IDs here

Graph g
- The graph constructed based off of the songs and users in the database

Methods:

Public IdGetter(Songs songs, Graph g)
- IdGetter constructor
- Initializes variable songs to parameter songs, and variable g to parameter g.

Public String getId(String songName, String artist)
- After the user inputs a song and/or artist, there is a problem if the entered song and/or artist has similar song or artist names. To avoid this problem, getId uses the method addSame to gather all similar songs and artists into a queue. getId then displays all the queue elements to the user and asks for input on which song the user wants recommendations for. After the selection, the method getId retrieves the song in the queue, and returns the song ID.
- Parameter songName is the song that getId searches for to get its song ID.
- Parameter artist is the artist of the song that getId searches for to get its song ID.

Private addSame(Queue<Song> q, String songName, String artist)
- gather all similar songs and artists into a queue
- Parameter q is the queue that will be updated to contain all the similar songs and artists

- Parameter songName is the song name that addSame will search for similar songs for
- Parameter artist is the artist that addSame will search for similar artists for

RecommendSongsDFS

Variables:

Private final int DEPTH
- The max depth that DFS travels to
- Set to 2

Private NeighborsDFS neighbors2
- Holds all the neighbors of neighbors of the source song

Private Graph g
- The graph that DFS is performed on

Private String song
- The source song that DFS is performed on

Methods:

Public RecommendSongsDFS(Graph g, String song)
- RecommendSongsDFS constructor
- Initializes variable g to parameter g
- Initializes variable song to parameter song
- Initializes neighbors2 to NeighborsDFS(g, g.indexOf(song), DEPTH)

Public Recommendation getTop1()
- Returns the top recommended song

public Iterable<Recommendation> getTopN(int N)
- Returns the top N recommended songs
- Parameter N is the number of recommended songs returned

MuSearch

Variables:

private JFrame frmMusearch;
- The Jframe

private JTextField SongName;
- The text field for inputting the song name before searching

private JTextField ArtistName;
- The text field for inputting the artist name before searching

private JLabel icon;
- MuSearch's icon

private JTextField Top10;
- Tenth highest recommended displayed in text field

private JTextField Top9;
- Ninth highest recommended displayed in text field

private JTextField Top8;
- Eighth highest recommended displayed in text field

private JTextField Top7;
- Seventh highest recommended displayed in text field

private JTextField Top6;
- Sixth highest recommended displayed in text field

private JTextField Top5;
- Fifth highest recommended displayed in text field

private JTextField Top4;
- Fourth highest recommended displayed in text field

private JTextField Top3;
- Third highest recommended displayed in text field

private JTextField Top2;
- Second highest recommended displayed in text field

private JTextField Top1;
- First highest recommended displayed in text field

private JLabel lblNewLabel_1;
- A JLabel to indicate which text field to input the song name

private JLabel lblArtist;
- A JLabel to indicate which text field to input the artist name

private JLabel lblChooseASong;
- A JLabel to draw attention to the text field songSelection

private JTextField songSelection;
- A text field to indicate to the user to look at the console after pressing the search button to select a song to find recommendations for

Methods:

Public static void main(String[] args)

- Used to run the GUI

Public MuSearch()

- Used to create the application

Private void initialize()

- Used to initialize the contents of the frame, variable frmMusearch

Recommendation

Variables:

Private String name

- Holds the string for name of recommendation.

Private int count

- Holds the integer for number of recommendations of specific name.

Methods:

Public Recommendation(String name, int count)

- Constructor for recommendation ADT. Sets state variables *int count*, *String name* to the parameters respectively.

Public String name()

- Accessor for the name in recommendation ADT.

Public int count()

- Accessor for the number of matches,count, for the give name in recommendation ADT.

Public String toString()

- Returns a string concatenated with a name and the count.

Song

Variables:

Private String songId, name, artist, album, date

- Hold the respective songId, name, artist, and date of release a specific song.

Methods:

Public Song(String songId, String name, String album, String artist, String date)

- Constructor for song abstract data type. Sets the state variable *String songId, name, album, artist* and *date* to the parameters respectively.

Public String getSongId()

- Accessor for the *songId* state variable in song ADT.

Public String getName()

- Accessor for the *name* state variable in song ADT.

Public String getArtist()

- Accessor for the *artist* state variable in song ADT.

Public String getAlbum()

- Accessor for the *album* state variable in song ADT.

Public String getDate()

- Accessor for the *date* state variable in song ADT.

Public String toString()

- Returns String representation of state variables respectively.

Songs

Variables:

Private Song[] songs

- Holds songs abstract types in an array.

Private Int n

- Contains the length of the array of songs.

Methods:

Public Songs(Song[] songs)

- Constructor for an array of songs. Sets the state variable *Song[] songs* to the array of song ADT's, and *int n* to the length of the *songs* array. The *Sort.sort(songs)* function is run on the array of songs for use from other Songs methods.

Public Song getById(String id)

- Accessor for the index at which a *String id* is located in the array of *songs* constructed in the constructor. *Search.binarySearchIndex()* is executed on the array of songs, and returns the respective song at the given index.

Public Iterator<Song> iterator()

- Creates a list iterator for a list of songs to be iterated through.

Private Class ListIterator<Song>

Variables:

Private Int i

- Holds the respective index of the iterator.

Methods:

Public ListIterator(int first)

- Constructs an iterator index starting location. Sets the state variable *int i* to the parameter *first.*

Public boolean hasNext()

- Checks to see if the iterator has reached the end of the array. Compares *int i* with *int n*, the size of the array.

Public Song next()

- Iterates to the index of the list of songs and returns the Song at the given index.

Read

Methods:

Public static String[] readData(String filename, int lines)

- Creates an array *s* of Strings double size of the lines, due to the size of the data base.
- Uses java internal methods BufferedReader and FileReader to read all the lines of the *filename* and split at the tab characters.

Public static Songs readSongs()

- Initializes the number of lines of the song data file, and the an array of Songs the proportionate to the number of lines in the song data file.
- Uses the java built in method BufferedReader and File Reader and splits at the "," to obtain the respective columns in the the data file.
- Returns all the songs from the data file with their respective information.

Search

Methods:

Public static int binarySearchIndex(Comparable[] arr, int lo, int hi, Comparable x)

- Executes a binarySearch on an array of items with their given range of *int lo* to *int hi* comparing with the given element being searched for *Comparable x*.
- Checks to see if *int hi* is greater than or equal to *int lo* and resets the mid if this is true. Recursively does this changing the lo and mid based on if the element att the middle index is greater than or less than the value,
- This search assumes the array is already sorted. Once the recursion is over returns the index.

TST

Variables:

Private int n
- Size of TST

Private node<Value> root
- Root of TST

Private static class Node<Value>
- Node of the TST
- Contains char c
- Nodes left, middle, right
- Value of string

Methods:

Public int size()
- Returns the size of the TST

Public boolean contains(String key)
- Returns true if the symbol table contains the key, false otherwise

Public value get(String key)
- Returns the value associated with the given string

Private Node<Value> get(Node<Value> x, String key, int d)
- Returns the subtree corresponding to the given key

Public void put(String key, Value val)
- Inserts a new key-value pair into the TST
- If the key already exists, the value is overwritten

Private Node<Value> put(Node<Value> x, String key, Value val, int d)
- Recursively creates a new Node and inserts it into the symbol table

Public Iterable<String> keys()
- Returns a queue of the strings that are in the TST, which is iterable

Private void collect(Node<Value> x, Stringbuilder prefix, Queue<String> queue)
- Collects the keys and reorganizes them based on the prefix

LinearHashStrInt

Variables:

Private int n
- Number of key-value pairs in the symbol table

Private int m
- Size of the linear probing table

Private String[] keys
- String of the keys in the table

Private Integer[] vals
- Integer values of the keys in the table

Methods:

Public string getkey(int i)
- Returns the key at index i

Public LinearHashStrInt()
- Initializes the symbol table at the default size of 4

Public LinearHashStrInt(int capacity)
- Initializes the symbol table at the user determined size

Public int size()
- Returns the number of keys in the table

Public int M()
- Returns the size of the linear probing table

Public boolean isEmpty()
- Returns true if the table is empty, false otherwise

Public boolean contains(String key)
- Returns true if the key is stored in the hash table

Public int hash(String key)
- Hashes the key that is given

Public void put(String key, Integer val)
- Inserts the specified key-value pair into the symbol table
- Overwrites the value if the key already exists

Public integer get(String key)

- Returns the value associated with the specified key

Public void delete(String key)
- Removes the specified key and its value from the symbol table

Private void resize(int capacity)
- Resizes the hash table if the table based on the new capacity by copying it to a new table of size capacity

Public Iterable<String> keys()
- Returns an iterable type Bag<String> by creating a new bag with the values of key in it

Sort

Methods:

Public static void sort(Comparable[] a)

- Shuffles the array to remove dependence on input
- Sorts the array using a quicksort algorithm

Private static void sort(Comparable[] a, int lo, int hi)

- Uses quicksort to sort the given array from indices lo to hi

Private static int partition(Comparable[] a, int lo, int hi)

- Partitions the array a between the indices lo and hi based on the partitioning element at a[lo]

Private static boolean less(Comparable v, Comparable w)

- Compares the two values and returns true if v is less than w

Private static void exch(Comparable[] a, int i, int j)

- Exchanges the two values in the array

Private static void shuffle(Comparable[] ar)

- Shuffles the values of the array

Graph:

Variables:

Private TST<Integer> st
- Ternary Search Tree of values

Private String[] keys
- Array of keys in the graph

Private final int V
- Number of vertices in the graph

Private int E
- Number of edges in the graph

Private Bag<Integer>[] adj
- Adjacency list containing the graph of nodes

Methods:

Public graph(String filename, int lines)
- Constructs the graph based on the data at the file specified and the number of lines that it contains

Public int V()
- Returns the number of vertices in the graph

Public int E()
- Returns the number of edges in the graph

Public String nameOf(int i)
- Returns the name of the key at index i in keys[]

Public int indexOf(String v)
- Returns the index of songs from the TST

Public void addEdge(int v, int w)
- Adds an edge based on the two vertices given

Public Iterable<Integer> adj(int v)
- Returns an iterable of the nodes connected to vertex v

Public int degree(String v)
- Returns the degree of the vertex of String v

Public boolean contains(String id)
- Returns true if the id is contained in the TST

NeighborsDFS

Variables:

Private final int s

- Song number

Private Bag<Integer> neighbors

- A Bag of the neighbors that this node has

Methods:

Public NeighborsDFS(Graph G, int s, int depth)

- Performs a DFS on the graph and finds neighbors of the node given

Private void dfs(Graph Gf, int v, int depth)

- Performs DFS on the given graph to a depth specified from the vertex specified

Private void dfs(Graph G, int v, int start, int stop)

- Performs DFS on the given graph from v for a number of elements start to stop

Public int size()

- Returns the number of neighbors that the node has

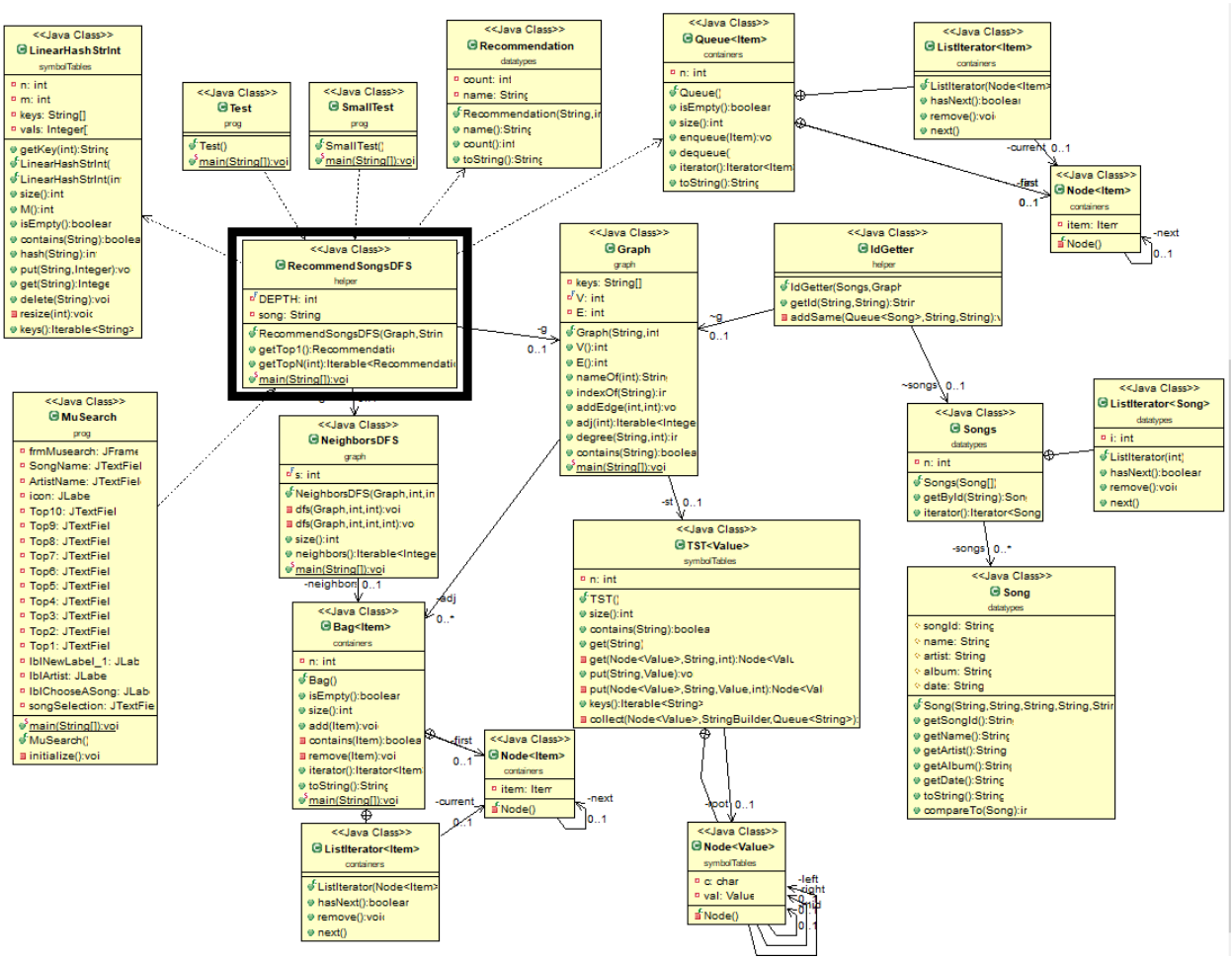
Public Iterable<Integer> neighbors()

- Returns the neighbors of the node

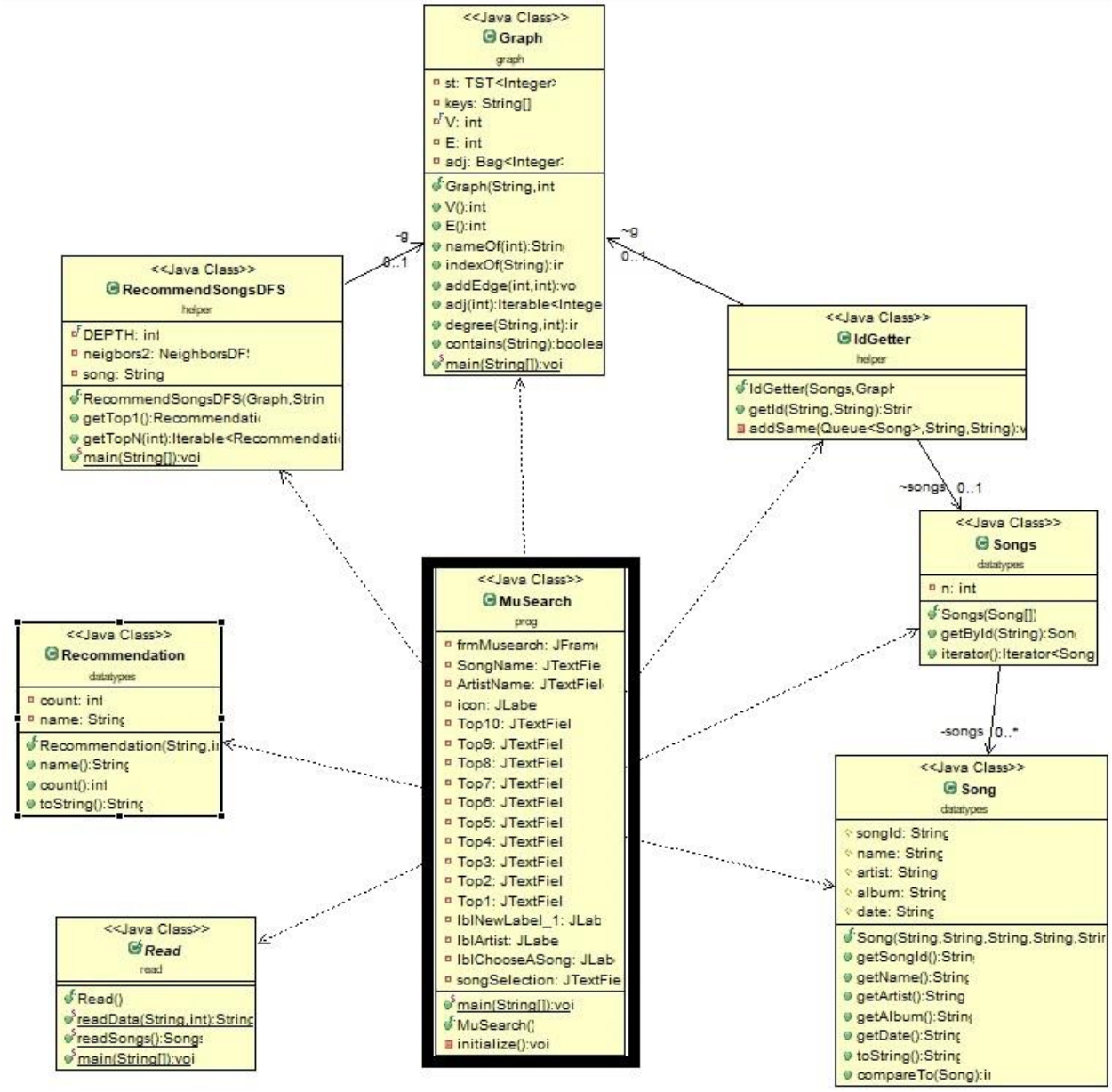
Uses Relations

Module	**Uses**
Recommendation	*N/A*
Song	*N/A*
Songs	*Search* *Sort*
Graph	*Stopwatch* *Bag* *TST*
NeighborsDFS	*Bag*
IdGetter	*Graph*
RecommendSongDFS	*Graph* *NeighborsDFS* *Queue* *Recommendation* *LinearHashStrInt*
Read	*Song* *Songs* *Stopwatch*
Search	*N/A*
Sort	*N/A*
Stopwatch	*N/A*
LinearHashStrInt	*Bag*
TST	*Queue*

RecommendedSongsDFS UML Diagram



MuSearch UML Diagram



Internal Review/Evaluation

Organization of the Code:

- In general, the code could be more organized in terms of methods. From the start of the project, it may have been beneficial to plan out and organize the modules and methods within the modules from the start as this would have made the end product cleaner and more readable.

Performance of Code:

- The functionality of the code was dependent on the time the software took to create a graph for the forty million nodes of artists/songs. Initial builds of the MuSearch showed an average run time of 22 seconds. This is in part to the reading and graph construction portion of the code. Later builds of the software including *version 0.97,* included faster graphing methods using ternary search trees and using the BufferedReader built in Eclipse method, from the previous built in Eclipse method Scanner. This resulted in a run time of 10 seconds.

Parsing:

- Originally, to read the file into the program we had used the java class Scanner. This proved unreliable as the data set was very large and scanner has a small buffer and took a long time to read the entire set. After some optimization, the Scanner had been switched to BufferedReader. BufferedReader has a much larger buffer of 8mb compared to 1mb. This increased the reading speed and ability to store the data quickly. After the data was stored, it can be hashed and inputted into the graph which will be discussed later.

Sorting/Searching:

- The algorithms utilized for searching and sorting using for the program were binary search and quick sort respectively. Binary search was used in for the finding song ID being searched in the GUI. Quick Sort was used to sort the song ID's in order so the binary sort would be able to carried on.

Graphing:

- The use of a TST in the graphing algorithm increased performance significantly as touched upon earlier. The process of finding neighbors by using a Depth First Search allowed the program to easily track the depth of the current node. Since the program's purpose was to find the neighbors of the neighbors, the depth would only need to reach 2.

- Hash tables were used to assist with the graphing. Each code representation of the song was lexicographically hashed and inputted into the hash table. This increased performance again, while only slightly increasing the memory required.

Information Hiding:

- The principle of information hiding is prevalent in our project. The private variables from the data types of Recommendation, Song, and Songs each are not directly accessible from any method. Instead, getters and mutators must be used to access or change the state of any data type variable.

GUI (Graphical User Interface):

- Since a GUI was not a required component of the project, we did not spend as much time on it as other components of the project. At the time of submission, the GUI is not fully working but there is a functional component.