COMP4905

SQL DATA STRUCTURES

By Joseph VanRompaey

Supervised by Prof. Louis D. Nel School of Computer Science

Carleton University

April 13, 2016

Abstract

SQLite is a database that gives the option of having a database in main memory as opposed to being on a hard disk. This can provide faster access to the database. Using this feature of SQLite the goal of this project is to develop and show how build data structures in a database. To do this tutorials are used to build up the data structures and show how we can maintain and present data.

Table of Contents

SQL Data Structures

When working with large data sets a database is essential. In analyzing data in a database, using a data structure to model the data would be useful. Normally students are taught data structures on small data sets that can easily fit into the main memory of a computer. Whereas databases are normally large and kept on disk. SQLite changes this approach and allows for a database to be kept in main memory.

This opens up new opportunities for looking at large data sets. The data can be accessed faster in memory than by using the disk. The data can also be incorporated into data structures, allowing the data to be modeled in familiar ways and algorithms previously developed can be brought in and run faster than if the database had been on the disk.

The goal of this project is to develop these data structure for SQL and incorporate them into a series of tutorials. These tutorials are able to be taught alongside a data structures course where that students will have minimal exposure to SQL and databases.

This guide has five tutorials on simple data structures covers how they are implemented and how to use them practically in a real data set. For this I have used the OC Transpo bus schedule data for winter 2016. As for the data structures we are going to look at we have an Arraylist, single linked list, doubly linked list where we will also cover stacks and queues, a graph and a binary search tree.

To represent data structures we are going to need establish a framework for managing the data structure in the database and operating on the data structure in Java. The data will already exist in the database, but a Java class will be need for representing the data in Java.

In the bus database there are four tables: ROUTES, STOPS, TRIPS and STOP_TIMES. We will need to make classes to represent these tables in Java. Here is the one for the route:

```
public class Route {
    String route_id,route_short_name,route_long_name,route_desc,route_type,route_url;
    public Route(String [] r){
        if (r.length==6){
            route_id = r[0];
            route_short_name = r[1];
            route_long_name = r[2];
            route_desc=r[3];
            route_type=r[4];
            route_url=r[5];
        }
    }
    public String toString(){
        return ""+route_id+" "+route_short_name+" "+route_long_name+" "+route_desc+" "+route_type+" "+route_url;
    }
}
```

Since ROUTES is the smallest table with only about 150 entries we will be able to see how we can manipulate the list without being overwhelmed.

In the database we will also need to keep track of the data structures that we have created. To do this an additional table will be used for each data structure we instantiate. These tables of data structures will have an index of tables for each type of data structure along

with the name of class of the table. The naming of tables in the database will be the data structure followed by the class name of the data followed by a table number for that class, for example ArrayListRoute0 will be the first arraylist of the route class.

ArrayLists

In this tutorial we will develop an arraylist as our first data structure with SQL. An arraylist is data structure that uses an array to store data, but has methods to automatically take care of operations, such as expanding the array when the array is full. For a database data is stored in a table, rows represent objects and columns represent the fields of the objects. If we want to express an arraylist in a table, the rows can be thought of as entries in an array of objects. But there are some features that an array has that become constraints when transforming an array into a table. Direct access of the array's entries is a key feature of the arraylist but tables do not have this feature, to recover this we will need to do some extra work to retain this function.

Constructor

Our table representation of an arraylist will consist of two columns, "eid" the index of the elements and "fid" the reference to the key of the table we wish to create the data structure for. We will start by looking at the SQLArrayList up to its constructor:

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

public class SQLArrayList<T> {
    int numberOfElements;
    int tableNumber;
    Connection theDatabase;
    String tableName;
    String colName;

    Statement s;
    Class<T> c;


    public SQLArrayList(Class<T> aClass, Connection aDatabase, String _tableName, String _colName)throws SQLException{
        numberOfElements = 0;
        c = aClass;
        theDatabase = aDatabase;
        tableName = _tableName;
        colName = _colName;
        s = theDatabase.createStatement();
        s.execute("CREATE TABLE IF NOT EXISTS ARRAYLISTS(lID INTEGER PRIMARY KEY AUTOINCREMENT, ArrayListName TEXT NOT NULL);");
        s.execute("INSERT INTO ARRAYLISTS (ArrayListName) VALUES ('"+c.getName()+"');");
        ResultSet a= s.executeQuery("SELECT COUNT (*) from ARRAYLISTS;");
        tableNumber = a.getRow();
        s.execute("CREATE TABLE IF NOT EXISTS ArrayList"+c.getName()+""+tableNumber+" (eID INTEGER NOT NULL, fID INTEGER, "
                + "PRIMARY KEY(eID), FOREIGN KEY(fID) REFERENCES "+tableName+"("+colName+"));");
        s.execute("Create Index AL"+c.getName()+""+tableNumber+" on  ArrayList"+c.getName()+""+tableNumber+"(fid);");
    }
```

First we have the imports for the class, we are going to need several classes for accessing the database. Connection is used to connect to the database, Statement is used to send messages to the database, ResultSet is used to store the results from queries on the database, and SQLException is used to handle problems that might arise accessing the database.

Since the data structure is going to be generic to allow different types of objects to use it, the class declaration SQLArrayList<T> will use T to specify the class being used. Next we have the variables used by the SQLArrayList, "DataBase theConnection" is the connection to the database, "Statement s" is the statement used to access the database, "int tableNumber" is

the number of the table in the database, “Class<T> c” is the class of the arraylist, “String tableName” and “String colName” refer to the table and column being used in the database. “numberOfElements” is the current count of elements in the list, although this number could be found by counting the rows in the table, storing this locally will save many calls to the database as the size of the list will be used frequently.

Next is the constructor itself, first we initialize all of the variables and we go on to check if the arraylist table has been created and add our new arraylist to the table once the table has been created. Finally we create the table for the arraylist, the table will have the “fid” reference the the table and column supplied to the constructor and establish foreign key relationship.

Insertion

Insertion into a normal arraylist can be done in effectively constant time, unless the array is at capacity, at which point the array needs to be expanded. Typically when an array is expanded the array’s size is doubled and then copied from its old location in memory to the new expanded one. All of this is done in linear time.

Inserting into a row into a table never has to worry about expanding so inserting into a table can be done in logarithmic time since the database uses a B-tree to store data.

```
public void add(int e){
    try {
        s.execute("INSERT INTO ArrayList"+c.getName()+""+tableNumber+" values ("+numberOfElements+", "+e+");");
        numberOfElements++;
    } catch (SQLException e1) {
        e1.printStackTrace();
    }

}
```

Here the insert places the index of the row, e, we wish to add to the SQLArrayList. The add(int e) function will insert the index to the end of the table and give it the index of the current size of the list.

If we wish to insert the element at a particular point a little more work is required to maintain the index we use to make the table an arraylist.

```
public void add(int i, int e){
    try {
        for(int j = numberOfElements; j>i; j--){
            s.addBatch("UPDATE ArrayList"+c.getName()+""+tableNumber+" SET eID = "+(j)+" WHERE eID = "+(j-1)+";");
        }
        s.addBatch("INSERT INTO ArrayList"+c.getName()+""+tableNumber+" values ("+i+", "+e+");");
        s.executeBatch();
        numberOfElements++;
    } catch (SQLException e1) {
        e1.printStackTrace();
    }
}
```

To insert at a particular spot we first need to make room for the element. In a normal arraylist implementation the elements are copied over to next entry in the array to make room for the new element, this is done in linear time.

However with the SQLArrayList we have to use the database's UPDATE instruction a linear number of times, but the UPDATE has to look at all of the entries if the variable the WHERE instruction has is not indexed. That would take linear time as well leaving us with quadratic time. But since we made eID the primary key SQLite will automatically create an index for us and reduce the update down to logarithmic time and bring the shift back down to nlogn time. Since the the final insert of the element is done in logarithmic time we end up with nlogn time.

Add All

Typically when using a database, we will want to work with large sets of data. To add data to our SQLArrayList we are going to need a couple of addAll methods to take in a data set. First let's look adding a set to an empty list or appending a set onto the end of an existing list:

```
public void addAll(Collection<Integer> e){
    try {
        for(Integer eh :e){
            s.addBatch("INSERT INTO ArrayList"+c.getName()+""+tableNumber+" values ("+numberOfElements+", "+eh.intValue()+");");
            numberOfElements++;
        }
        s.executeBatch();
    } catch (SQLException e1) {
        e1.printStackTrace();
    }
}
```

For addAll the Collection<Integer> e will be a set of keys from the database as the result of a query such as this:

ResultSet r = stat.executeQuery("select eID from routes;");

In the bus database all of the primary keys are in the eID column, this query will search the database and return all of the keys from the route table in the bus database. From this ResultSet we can add the indexes to a Java collection and pass all of the keys to the addAll in the SQLArrayList class.

From here the method inserts the keys just like before with the regular insert. Since we will be doing several operations in one method it becomes more efficient to do all of the operations in a batch.

If we do not want to insert at the end of the list, we can insert at a particular index:

```
public void addAll(int index, Collection<Integer> e){
    if(index<0||index>numberOfElements) throw new IndexOutOfBoundsException();
    try {
        for(int j = numberOfElements+e.size(); j>index; j--){
            s.addBatch("UPDATE ArrayList"+c.getName()+""+tableNumber+" SET eID = "+(j)+" WHERE eID = "+(j-e.size())+";");
        }
        for(Integer eh :e){
            s.addBatch("INSERT INTO ArrayList"+c.getName()+""+tableNumber+" values ("+index+", "+eh.intValue()+");");
            index++;
        }
        s.executeBatch();
        numberOfElements+=e.size();
    } catch (SQLException e1) {
        e1.printStackTrace();
    }
}
```

Here we combine approaches used previously in addAll and the at an index from before. First we need to shift down the existing elements by the size of the incoming set then insert the new elements. The run time of this maintains a nlogn runtime by shifting down the whole set at once instead of one at a time if we had inserted the elements one by one which would have had a quadratic run time.

Get, Set and Size

In a normal arraylist getting an element is done in constant time, but to find an element in a table the best we can do is logarithmic time. We can query the database for the index we want and retrieve the key of the original table.

```
public int get(int index){
    try {
        ResultSet a = s.executeQuery("SELECT fID FROM ArrayList"+c.getName()+""+tableNumber+" WHERE eID = "+index+";");
        int q = a.getInt(1);
        return q;
    } catch (SQLException e) {

        e.printStackTrace();
    }
    return -1;
}
```

For set a normal arraylist will set an element in constant time and but for the SQLArrayList the set will take logarithmic time to find the element and constant time to update it.

```
public int set(int i, int e){
    if(i < 0 || i> numberOfElements-1) throw new IndexOutOfBoundsException();
    try {
        ResultSet a = s.executeQuery("SELECT fID FROM ArrayList"+c.getName()+""+tableNumber+" WHERE eID = "+i+";");
        int res = a.getInt(1);
        s.execute("UPDATE ArrayList"+c.getName()+""+tableNumber+" SET fID = "+e+" WHERE eID = "+i+";");
        return res;
    } catch (SQLException e1) {
        e1.printStackTrace();
    }
    return -1;
}
```

The size of the list could be found with a call to the database that would count the number of rows in the table, since this number is constantly being used by the SQLArrayList, storing the number locally and returning the local variable is more efficient.

```
public int size() {
    return numberOfElements;
}
```

Index Of

If we want to know if the index a certain row is in the SQLArrayList we can do a search for the element in linear time by searching every element in the table. But since we are only creating two columns we can create an index of the fid column in addition to the one automatically created by SQLite for the eid column. To create an Index we can add

s.execute("Create Index AL"+c.getName()+""+tableNumber+" on ArrayList"+c.getName()+""+tableNumber+"");

at some point and SQL will create an index for the table, which allow us to search the table in logarithmic time. But indexing comes at a cost, every update and insert that modifies the table also has to modify the index, creating the index only once all the data has been entered will save some work for the database.

As for the IndexOf functions we often want the first instance or last instance in a data set. For these functions we can search the fid for the row we want and sort by ascending or descending order to get the first and last instance respectively.

```
public int indexOf(int e) throws SQLException{
    ResultSet a = s.executeQuery("SELECT eID FROM ArrayList"+c.getName()+""+tableNumber+" WHERE fID = "+e+" ORDER BY eID ASC;");
    int res = a.getInt(1);
    return res;
}

public int lastIndexOf(int e) throws SQLException{
    ResultSet a = s.executeQuery("SELECT eID FROM ArrayList"+c.getName()+""+tableNumber+" WHERE fID = "+e+" ORDER BY eID DESC;");
    int res = a.getInt(1);
    return res;
}
```

Remove

To remove an element from an arraylist the requested element is removed and then the other elements are shifted down in nlogn time. If we remove the last element this will give us logarithmic time since there is no shifting required. For the SQLArrayList deleting an element first finds the row number to be deleted, retrieves its value, deletes the row from the table and updates the index of the rows that exist afterwards.

```
public int remove(int index) throws SQLException{
    if(index<0||index>numberOfElements-1)throw new IndexOutOfBoundsException();
    ResultSet a = s.executeQuery("SELECT fID FROM ArrayList"+c.getName()+""+tableNumber+" WHERE eID = "+index+";");
    int res = a.getInt(1);
    s.execute("DELETE FROM ArrayList"+c.getName()+""+tableNumber+" WHERE eID = "+index+";");
    for(int i = index; i<numberOfElements; i++){
        s.addBatch("UPDATE ArrayList"+c.getName()+""+tableNumber+" SET eID = "+(i)+" WHERE eID = "+(i+1)+";");
    }
    s.executeBatch();
    numberOfElements--;
    return res;
}
```

Working with large sets we will also want a way to work with sublists. To retrieve a sublist we can have the database perform a range query and return the sublist with only one call to the database.

```
public List<Integer> subList(int fromIndex, int toIndex) throws SQLException{
    if(fromIndex>toIndex||fromIndex<0||toIndex>numberOfElements)throw new IndexOutOfBoundsException();
    ResultSet a = s.executeQuery("SELECT fID FROM ArrayList"+c.getName()+""+tableNumber+" WHERE eID BETWEEN "+fromIndex+" AND "+toIndex+";");
    List<Integer> q = new ArrayList<Integer>();
    while(!a.isAfterLast())
    {
        a.next();
        q.add(a.getInt(1));
    }
    return q;
}
```

This will return the list, but if we want to remove a sublist of elements we can do that as well

```
public void removeRange(int a , int b) throws SQLException{
    if(a>b||a<0||b>numberOfElements)throw new IndexOutOfBoundsException();
    s.addBatch("DELETE FROM ArrayList"+c.getName()+""+tableNumber+" WHERE eID BETWEEN "+a+" AND "+b+";");
    for(int i = b; i<numberOfElements; i++){
        s.addBatch("UPDATE ArrayList"+c.getName()+""+tableNumber+" SET eID = "+(i-(b-a)-1)+" WHERE eID = "+(i)+";");
    }
    s.executeBatch();
}
```

If we wish to delete all of the elements in the list we can clear the list:

```
public void clear() throws SQLException{
    s.execute("DELETE FROM ArrayList"+c.getName()+""+tableNumber+";");
}
```

Printing

Lastly we will want to print the list to see what elements are inside. This will print the element's index, the key into the referenced table and one of the columns from the referenced table to give a label to the row.

```
String listPrint(String label) throws SQLException{
    String trt="";
    ResultSet r = s.executeQuery("select eid, fid, "+label+" from ArrayList"+c.getName()+""+tableNumber+" join "+tableName+" on fid="+colName+";");
    while(r.next()){
        trt+= r.getString(1)+" | ";
        trt+= r.getString(2)+" | ";
        trt+= r.getString(3);
        trt+="\n";
    }
    return trt;

}
```

Tutorial

To try out these functions we will create a SQLArrayList of bus routes from the bus database.

First we will create a list of existing busses:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.LinkedList;

public class BusTutorial {
    public static void main(String[] args) throws Exception {
        try {
            Class.forName("org.sqlite.JDBC");
            Connection database = DriverManager.getConnection("jdbc:sqlite:");
            Statement stat = database.createStatement();
            stat.executeUpdate("restore from OCBus.db");
            ResultSet r = stat.executeQuery("select rID from routes;");
            SQLArrayList<Route> routes = new SQLArrayList<Route>(Route.class, database,"routes", "rID");
            LinkedList<Integer> routeNumbers = new LinkedList<Integer>();
            while(r.next()){
                routeNumbers.add(r.getInt(1));
            }
            routes.addAll(routeNumbers);
            stat.executeUpdate("backup to bus.db");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }

    }
}
```

Here we start by creating an in memory database. The line:

Connection database = DriverManager.getConnection("jdbc:sqlite:");

creates a database in memory since we do not specifying a database like we would if we were to connect to the database on disk like this:

Connection database = DriverManager.getConnection("jdbc:sqlite:OCBus.db");

Since this creates an empty database we need to load our data off of the disk with the line:

stat.executeUpdate("restore from OCBus.db");

The SQLArrayList for the routes is created with the line:

SQLArrayList<Route> routes = new SQLArrayList<Route>(Route.class, database,"routes", "rID");

In the Routes table in the bus database the primary key is the rID column. We pass this column name to the SQLArrayList reference it for the elements.

Once the database is loaded we can find the routes we want to add to the SQLArrayList.

Since the ResultSet can not return the column as list, we add the key to a list that we pass to through to the SQLArrayList with addAll.

Look at last line before the catch blocks:

```
stat.executeUpdate("backup to bus.db");
```

This backs up the database to a file if we want to use it again in the future. To overwrite the existing database we use the same name as the one we loaded or we can specify a new file if we wish to preserve the original database.

Let's print out the list along with bus's route number, which is found in the column by adding:

```
System.out.println(routes.listPrint("route_id, route_short_name"));
```

Before the statement to backup the database and run the program.

Here can see the first column is the element's index, followed by the foreign key to the routes table followed by our label which here is the route numbers.

Next we will try making some bus routes. Since there's no routes in the 300 range lets add a new route. First we need to insert the new bus into the the data set:

```
stat.execute("INSERT INTO ROUTES VALUES(154, '301-224',301,'null','null',3,'null');");
```

Now let's add our new route to the SQLArrayList:

```
r= stat.executeQuery("SELECT rID from ROUTES WHERE route_short_name = 301");
routes.add(r.getInt(1));
```

Create a few more routes and add them together.

If we print the list again we will see that we have added the elements to the end of the list but the rest if the list is in numerical order. So let's remove the elements we just added and reinsert them in the correct spot:

```
routes.remove(153);
routes.add(124, 154);
```

If we want to create a java object of a route we will need to get the row from the database and pass the results to the route's constructor:

```
routes.addAll(routeNumbers);
stat.execute("INSERT INTO ROUTES VALUES(154, '301-224',301,'null','null',3,'null');");
r= stat.executeQuery("SELECT rID from ROUTES WHERE route_short_name = 301");
routes.add(r.getInt(1));
routes.remove(153);
routes.add(124, 154);
System.out.println(routes.listPrint("route_id, route_short_name"));
int q = routes.get(47);
r = stat.executeQuery("Select * from routes where rID = "+q+";");
Route nintyfive = new Route(new String[]{r.getString(1),r.getString(2),r.getString(3),r.getString(4),r.getString(5),r.getString(6)});
System.out.println("The route nintyfive "+nintyfive);
stat.executeUpdate("backup to bus.db");
```

Although we do not have anything in the route class other than the print function right now, we will build on it in the future as we add stops to our routes and the times the busses stops at the stops.

LinkedList

The linked list is the next data structure we are going to look at. The SQLLinkedList will improve on our design for the SQLArrayList by allowing us to perform a couple operations more efficiently but not all of them. The Linked List works but having a pointer from the current element to the element that comes after. To store this in the database we are going to use a similar structure as the SQLArrayList but we are going to add an additional column to store the element that comes next in the list.

Constructor

The table representation of the linked list will be composed of three columns, “eid” the index of the elements in the list, “fid” the reference to the key of the other table and “nid” the column with next element in the list, which will be an “eid” in the SQLLinkedList table. Here is code up to the constructor

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Collection;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class SQLLinkedList<T> {
    int numberOfElements;
    int elementCounter;
    int tableNumber;
    String colName;
    String tableName;
    Connection theDatabase;
    Statement s;
    Class<T> c;
    int head;
    int tail;
    public SQLLinkedList(Class<T> aClass, Connection aDatabase, String _tableName, String _colName) throws SQLException{
        numberOfElements = 0;
        c = aClass;
        theDatabase = aDatabase;
        colName = _colName;
        tableName = _tableName;
        tail = 0;
        head =0;
        elementCounter = 0;
        s = theDatabase.createStatement();
        s.execute("CREATE TABLE IF NOT EXISTS LINKEDLISTS(lID INTEGER PRIMARY KEY AUTOINCREMENT, LinkedListName TEXT NOT NULL);");
        s.execute("INSERT INTO LINKEDLISTS (LinkedListName) VALUES ('"+c.getName()+"');");
        ResultSet a= s.executeQuery("SELECT COUNT (*) from LINKEDLISTS;");
        tableNumber = a.getRow();
        s.execute("CREATE TABLE IF NOT EXISTS LinkedList"+c.getName()+""+tableNumber+" (eID INTEGER NOT NULL, fID INTEGER, "
                + "nID INTEGER, FOREIGN KEY(fID) REFERENCES "+tableName+"("+colName+"));");
    }
```

Like before we have the various classes we require for the SQLLinkedList, they are mostly the same as SQLArrayList. There’s a couple of new ones but their use will be explained in context.

The big change here from the SQLArrayList’s constructor is the addition of the head and tail, we will use them to keep track of the first and last element respectively. We have also changed the table we are using for this so we change the create table as well, adding our new column for the next element.

Insertion

The insertion will be a bit different than the arraylist, in a linked list we only need to modify one element's next variable to add a new one. This can be done in logarithmic time since the update and and insert are logarithmic time but since we are going to store the tail of the list as well as the head we will save us the trouble of finding the end of the list which would take nlogn time.

```
boolean add(int e)throws SQLException{
    if(numberOfElements == 0){
        s.execute("INSERT INTO LinkedList"+c.getName()+""+tableNumber+" VALUES ("+elementCounter+", "+e+", null);");
        head = elementCounter;
        tail = elementCounter;
        elementCounter++;
        numberOfElements++;
        return true;
    }
    s.addBatch("INSERT INTO LinkedList"+c.getName()+""+tableNumber+" VALUES ("+elementCounter+", "+e+", null);");
    s.addBatch("UPDATE LinkedList"+c.getName()+""+tableNumber+" SET nID = "+elementCounter+" WHERE eID = "+tail+";");
    s.executeBatch();
    tail = elementCounter;
    elementCounter++;
    numberOfElements++;
    return true;
}
```

The insert function checks if the list is empty, if the list is we add the first element to the table, otherwise we add the the new element and update the tail to point to the new element. At the end of the function we set the tail to the elementCounter, which will be equal to the eid of the element we just added. Finally we increase the elementCounter and the number of elements in the list. It is important to note that the elementCounter is used to keep track of the elements we insert to ensure the new items we insert have unique eid values.

```
void add(int i, int e) throws SQLException{
    if(i<0||i>numberOfElements-1) throw new IndexOutOfBoundsException();
    if(i==0){
        addFirst(e);
    }
    else if(i==(numberOfElements-1)) {
        add(e);
    }
    else{
        int j=0;
        int curr =head;
        s.addBatch("UPDATE LinkedList"+c.getName()+""+tableNumber+" set nID = "+elementCounter+" WHERE eID="+tail+";");
        ResultSet r;
        while(j!=(i-1)){
            r = s.executeQuery("SELECT nID from LinkedList"+c.getName()+""+tableNumber+" WHERE eID = "+curr+";");
            curr = r.getInt(1);
            j++;
        }
        r = s.executeQuery("SELECT nID from LinkedList"+c.getName()+""+tableNumber+" WHERE eID = "+curr+";");
        j= r.getInt(1);
        if(curr==tail){
            s.addBatch("INSERT INTO LinkedList"+c.getName()+""+tableNumber+" VALUES ("+elementCounter+", "+e+", null);");
            s.addBatch("UPDATE LinkedList"+c.getName()+""+tableNumber+" set nID ="+elementCounter+" WHERE eID = "+curr+";");
            tail = elementCounter;
        }else{
        s.addBatch("INSERT INTO LinkedList"+c.getName()+""+tableNumber+" VALUES ("+elementCounter+", "+e+", "+j+");");
        s.addBatch("UPDATE LinkedList"+c.getName()+""+tableNumber+" set nID ="+elementCounter+" WHERE eID = "+curr+";");
        }
        s.executeBatch();
        elementCounter++;
        numberOfElements++;
    }
}
```

To insert at a particular index is a little bit harder since we have to find where in the list our ith element is. To do this we need to go through each item in the list to count where the list until we reach the ith entry and then insert at that location. The while loop counts how many

entries we have checked asking the database for the next id in the list. Once we have the right spot to insert the element and update the previous one.

Add all

For the add all functions we can add the elements we wish to add in a batch and insert them at the end of the list.

```
boolean addAll(Collection<Integer> e) throws SQLException{
    if(e.isEmpty()){
        return false;
    }
    Iterator<Integer> k=e.iterator();
    if(numberOfElements == 0){
        add(k.next());
        numberOfElements-=1;
    }
    s.addBatch("UPDATE LinkedList"+c.getName()+""+tableNumber+" set nID = "+elementCounter+" WHERE eID="+tail+";");
    int next;
    while(k.hasNext()){
        next = k.next();
        s.addBatch("INSERT INTO LinkedList"+c.getName()+""+tableNumber+" VALUES ("+elementCounter+", "+next+", "+(++elementCounter)+");");
    }
    s.addBatch("UPDATE LinkedList"+c.getName()+""+tableNumber+" set nID = null WHERE nID="+(elementCounter)+";");
    s.executeBatch();
    tail=elementCounter-1;
    numberOfElements+=e.size();
    return true;
}
```

For the addAll at a certain index function the task of finding the ith element has been abstracted to its own function since find the ith element is used in several different functions.

```
boolean addAll(int i, Collection<Integer> e) throws SQLException{
    if(i<0||i>numberOfElements) throw new IndexOutOfBoundsException();
    if(e.isEmpty()){
        return false;
    }
    if(i==numberOfElements){
        return addAll(e);
    }
    Iterator<Integer> k=e.iterator();
    if(elementCounter==0){
        add(k.next());
        numberOfElements-=1;
    }
    int curr = findI(i);

    if(i==0){
        head = elementCounter;
    }
    int next;
    while(k.hasNext()){
        next = k.next();
        s.addBatch("INSERT INTO LinkedList"+c.getName()+""+tableNumber+" VALUES ("+elementCounter+", "+next+", "+(++elementCounter)+");");
    }
    s.addBatch("UPDATE LinkedList"+c.getName()+""+tableNumber+" set nID = "+curr+" WHERE nID="+(elementCounter)+";");
    s.executeBatch();
    numberOfElements+=e.size();
    return true;
}
```

The add all here works just like the add where we insert all of the elements and then we update the last element we inserted. The add all here is nlogn which again and will save us some work over inserting one at a time since we modify the database in one batch. We also have functions for addFirst and add last:

```
void addFirst(int e) throws SQLException{
    if(numberOfElements==0) {
        add(e);
    }
    s.execute("INSERT INTO LinkedList"+c.getName()+""+tableNumber+" VALUES ("+elementCounter+", "+e+", "+head+");");
    head = elementCounter;
    elementCounter++;
    numberOfElements++;
}

void addLast(int e) throws SQLException{
    add(e);
}
```

Get Set and Size

In linked list getting an element takes linear time since we have to count items until we have the right one. For us it will take nlogn time to find the element in the database since a search is done every time and the search will take logarithmic time.

```
int get(int i) throws SQLException{
    if(i<0||i>numberOfElements) throw new IndexOutOfBoundsException();
    int j=0;
    ResultSet r;
    int curr =head;
    while(j!=i){
        r = s.executeQuery("SELECT nID from LinkedList"+c.getName()+""+tableNumber+" WHERE eID = "+curr+";");
        curr = r.getInt(1);
        j++;
    }
    r = s.executeQuery("SELECT fID from LinkedList"+c.getName()+""+tableNumber+" WHERE eID = "+curr+";");
    return r.getInt(1);
}
```

For the linked list we added a couple more get methods Get First and get last to grab the tail item of the list

```
int getFirst() throws SQLException{
    ResultSet r = s.executeQuery("SELECT fID from LinkedList"+c.getName()+""+tableNumber+" WHERE eID = "+head+";");
    return r.getInt(1);
}

int getLast() throws SQLException{
    ResultSet r = s.executeQuery("SELECT fID from LinkedList"+c.getName()+""+tableNumber+" WHERE eID = "+tail+";");
    return r.getInt(1);
}
```

With the set operation we have the same issue of trying to get the ith element and have to search through the table to find the right element.

For size we just return the number of elements in the list.

```
int size(){
    return numberOfElements;
}
```

Index Of

Finding the index of an item in a linked list will require searching the the list to find the object. This is going to take nlogn time. To do this, the code for finding the the index used before has been modified to return once the item has been found.

```
int indexOf(int o) throws SQLException{
    int j=-1;
    ResultSet r;
    int curr =head;
    do{
        j++;
        r = s.executeQuery("SELECT nID, fID from LinkedList"+c.getName()+""+tableNumber+" WHERE eID = "+curr+";");

        if(o==r.getInt(2)){
            return j;
        }
        curr = r.getInt(1);
    }
    while(curr!=tail);
    return -1;
}

int lastIndexOf(int o) throws SQLException{
    int j=-1;
    ResultSet r;
    int loc = -1;
    int curr =head;
    do{
        r = s.executeQuery("SELECT nID, fID from LinkedList"+c.getName()+""+tableNumber+" WHERE eID = "+curr+";");
        if(o==r.getInt(2)){
            loc = j+1;
        }
        curr = r.getInt(1);
        j++;
    }
    while(curr!=tail);
    return loc;
}
```

Finding the last index works the same way but has to check to whole list to make sure it finds the last index of the item.

Remove

The remove function removes the first element of the linked list and updates the head of the list to the next element. This is done in logarithmic since we have to find and delete the row from the table and both operations take logarithmic time.

```
int remove() throws SQLException{
    if(numberOfElements ==0)throw new NoSuchElementException();
    ResultSet r = s.executeQuery("SELECT nID, fID from LinkedList"+c.getName()+""+tableNumber+" WHERE eID = "+head+";");
    int b = head;
    head = r.getInt(1);
    numberOfElements--;
    int a =r.getInt(2);
    s.execute("DELETE from LinkedList"+c.getName()+""+tableNumber+" WHERE eID = "+b+";");
    return a;
}
```

The remove at function takes nlogn time since we have to find the element we want to delete before deleting it. Since we are removing from the middle of the list we also need to update the next element column of the preceding element to maintain the linked list's structure.

```
int remove(int i) throws SQLException{
    if(i<0||i>numberOfElements) throw new IndexOutOfBoundsException();
    int j=0;
    ResultSet r;
    int curr =head;
    int p = -1;
    while(j!=(i-1)){
        r = s.executeQuery("SELECT nID from LinkedList"+c.getName()+""+tableNumber+" WHERE eID = "+curr+";");
        p = curr;
        curr = r.getInt(1);
        j++;
    }
    r =s.executeQuery("SELECT nid, fID from LinkedList"+c.getName()+""+tableNumber+" WHERE eID = "+curr+";");
    int o = r.getInt(2);
    s.execute("UPDATE LinkedList"+c.getName()+""+tableNumber+" SET nID ="+r.getInt(1)+" WHERE eID ="+p+";");
    s.execute("DELETE from LinkedList"+c.getName()+""+tableNumber+" WHERE eID = "+curr+";");
    numberOfElements--;
    return o;
}
```

Included here is also a removeFirst function and a removeLast function. The removeFirst function just calls remove but the removeLast function does a bit more work. First we find the element we find the tail and save the fid, then we delete the row from the table, then find and update the new tail.

```
int removeFirst() throws SQLException{
    return remove();
}

int removeLast() throws SQLException{
    if(numberOfElements ==0)throw new NoSuchElementException();
    ResultSet r = s.executeQuery("SELECT fID from LinkedList"+c.getName()+""+tableNumber+" WHERE eID = "+tail+";");
    int a = r.getInt(1);
    s.execute("DELETE from LinkedList"+c.getName()+""+tableNumber+" WHERE eID = "+tail+";");
    r= s.executeQuery("SELECT eID from LinkedList"+c.getName()+""+tableNumber+" WHERE nID = "+tail+";");
    tail = r.getInt(1);
    s.execute("UPDATE LinkedList"+c.getName()+""+tableNumber+" set nID = null where eID ="+tail+";");
    numberOfElements--;
    return a;
}
```

Also included is a removeObject where you can pass in a number in the fid column and delete the first instance of it from the list. It works the same way as the remove at function but it just searching for fid instead of eid.

```
boolean removeObject(int o) throws SQLException
{
    ResultSet r;
    int curr =head;
    do{
        r = s.executeQuery("SELECT nID, fID from LinkedList"+c.getName()+""+tableNumber+" WHERE eID = "+curr+";");
        if(o==r.getInt(2)){
            s.execute("DELETE from LinkedList"+c.getName()+""+tableNumber+" WHERE eID = "+curr+";");
            s.execute("UPDATE LinkedList"+c.getName()+""+tableNumber+" SET nID ="+r.getInt(1)+" WHERE nID ="+curr+";");
            numberOfElements--;
            return true;
        }
        curr = r.getInt(1);
    }
    while(curr!=tail);
    return false;
}
```

Tutorial

In this tutorial we are going to use the SQLLinkedList to hold the stops for a bus trip. For this we are going to find a particular bus trip and then find the stops the bus stops at. First we're going to find a particular trip from the database. The bus trips are in the trips table, but to find out what stops are in the trip we will need to get the data from the stop times table. So we will start by searching for trips for a particular bus in the database we can search:

*select * from trips where route_id like "7-%";*

This will search for bus trips of the number 7 bus. The data covers the OC Transpo service for a few months in the winter of 2016, there are about a thousand trips on the schedule for the bus we only need one right now. The trips table distinguishes different schedules depending on the day of the week of for a holiday. For our purposes we are going to use a weekday bus, the trip numbered 39373164 leaves St Laurent station around 9am and arrives at Carleton station about an hour later. Since we want to find the stops on this trip we can join the stops table with the stop times table joining on the stops where our trip stops.

*select * from stop_times join stops on stop_times.stop_id=stops.stop_id where trip_id like "%39373164%";*

This returns a table of the the stops in our trip from which we can make our list of stops. First let's create a SQLLinkedList for our stops:

SQLLinkedList<Stop> BusTripStops = new SQLLinkedList<Stop>(Stop.class, database,"stops", "sid");

Now we can fill the list from the table we searched:

```
ResultSet r = stat.executeQuery("select sid from stop_times join stops on stop_times.stop_id=stops.stop_id where trip_id like \"%39373164%\";");
SQLLinkedList<Stop> BusTripStops = new SQLLinkedList<Stop>(Stop.class, database,"stops", "sid");
LinkedList<Integer> stopTimes = new LinkedList<Integer>();
while(r.next()){
    stopTimes.add(r.getInt(1));
}
BusTripStops.addAll(stopTimes);
```

If we wish to change this list we can add and remove stops to the SQLLinkedList but this will only change the list and not the data in the database. But let's manipulate the SQLLinkedList anyway to let's change one of the stops on the list:

BusTripStops.set(34, 581);

This will detour the bus to the next stop down the road. If we want to know where a stop is in the list we can find it by using the indexof function:

System.out.println("BANK / SOMERSET is stop "+BusTripStops.indexOf(766));

Or something more useful would be trying to find how many stops are between two stops. Such as:

System.out.println("BANK / SOMERSET is "+(BusTripStops.indexOf(766)-BusTripStops.indexOf(584))+" stops from RIDEAU / CHAPEL");

If we were monitoring this bus as it is travelling around the city we might want to get stops as we travel. The stops are stored sequentially in our list so we can remove the first element from the list once we have passed the stop.

```
BusTripStops.removeFirst();
```

We can call this every time we reach a stop and when we reach the end of the trip our list will be empty.

Double Linked List

The double linked list is the next structure we are going to implement. The idea is going to be similar to the linked list where in the table will have a column for the next entry but the double linked list will also have another column with the previous element in the list.
The double linked list, SQLDoubleLL, will also have the functions for a queue and a stack should we want to use them. For the most part the implementation of the double linked list will be similar to the single linked list with an extra update to update both previous and next statements. More functions have been added to try and match the normal LinkedList class used in Java.

Constructor

The table representation of the double linked list will have four columns, “eid” the element’s id in the list, “fid” the reference to the other table, “nid” which has the eid of the next entry, and “pid” the eid of the previous entry.

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Collection;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class SQLDoubleLL <T> {
    int numberOfElements;
    int elementCounter;
    int tableNumber;
    String tableName;
    String colName;
    Connection theDatabase;
    Statement s;
    Class<T> c;
    int head;
    int tail;
    public SQLDoubleLL(Class<T> aClass, Connection aDatabase, String _tableName, String _colName) throws SQLException{
        numberOfElements = 0;
        elementCounter = 0;
        colName = _colName;
        tableName = _tableName;
        c = aClass;
        theDatabase = aDatabase;
        tail = 0;
        head =0;
        s = theDatabase.createStatement();
        s.execute("CREATE TABLE IF NOT EXISTS DOUBLELL(lID INTEGER PRIMARY KEY AUTOINCREMENT, DoubleLLName TEXT NOT NULL);");
        s.execute("INSERT INTO DOUBLELL (DoubleLLName) VALUES ('"+c.getName()+"');");
        ResultSet a= s.executeQuery("SELECT COUNT (*) from DOUBLELL;");
        tableNumber = a.getRow();
        s.execute("CREATE TABLE IF NOT EXISTS DoubleLL"+c.getName()+""+tableNumber+" (eID INTEGER NOT NULL, fID INTEGER, "
                + "nID INTEGER, pID INTEGER, FOREIGN KEY(fID) REFERENCES "+tableName+"("+colName+"));");
    }
```

The constructor is the as the SQLLinkedList just with the extra column for the previous entry. Just like before we have a head and a tail to our list as well as an element counter to keep track of the elements as their added and a separate integer numberOfElements to maintain the size of the list.

Find I

In the SQLLinkedList there were several instances where we wanted to the element at a particular point in the list. To do this we were searching in the function and changing the implementation each time. But it might be better to use one function that finds the element,

so for the SQLDoubleLL we will use the function *findI(int i)* to find what item is at position i in the list.

```
private int findI(int i) throws SQLException{
    if(i<0||i>numberOfElements)throw new IndexOutOfBoundsException();
    if(i==0){
        return head;
    }
    int j;
    int curr =head;
    ResultSet r;
    for(j=0; j<i; j++){
        r = s.executeQuery("SELECT nID from DoubleLL"+c.getName()+""+tableNumber+" WHERE eID = "+curr+";");
        curr = r.getInt(1);
    }
    return curr;
}
```

The code here goes through the list in nlogn time, checking if the element is the ith one, and returning the eid of the element.

Insertion

The code for the inserts does not change too much since before we were keeping track of the last element using tail in the linked list. The regular add function will add the new item to the list:

```
boolean add(int e) throws SQLException{
    if(numberOfElements == 0){
        s.execute("INSERT INTO DoubleLL"+c.getName()+""+tableNumber+" VALUES ("+elementCounter+", "+e+", null, null);");
        head = elementCounter;
        tail = elementCounter;
        elementCounter++;
        numberOfElements++;
        return true;
    }
    s.addBatch("INSERT INTO DoubleLL"+c.getName()+""+tableNumber+" VALUES ("+elementCounter+", "+e+", null, "+tail+");");
    s.addBatch("UPDATE DoubleLL"+c.getName()+""+tableNumber+" SET nID = "+elementCounter+" WHERE eID = "+tail+";");
    s.executeBatch();
    tail = elementCounter;
    elementCounter++;
    numberOfElements++;
    return true;
}
```

In a normal double linked list we could do this insertion in constant time, but to do this with the database we need to perform and update and an insert which will take logarithmic time for each.

```
void add(int i, int e) throws SQLException{
    if(i<0||i>numberOfElements-1)throw new IndexOutOfBoundsException();
    if(i==0){
        addFirst(e);
    }
    else if(i==numberOfElements-1){
        add(e);
    }
    else{
        int eid = findI(i);
        ResultSet r = s.executeQuery("select pID from DoubleLL"+c.getName()+""+tableNumber+" where eID = "+eid+";");
        Statement s2 = theDatabase.createStatement();
        s2.addBatch("INSERT INTO DoubleLL"+c.getName()+""+tableNumber+" values ("+elementCounter+", "+e+", "+eid+", "+r.getInt(1)+");");
        s2.addBatch("UPDATE DoubleLL"+c.getName()+""+tableNumber+" set nID = "+elementCounter+" where eID ="+r.getInt(1)+";");
        s2.addBatch("UPDATE DoubleLL"+c.getName()+""+tableNumber+" set pID = "+elementCounter+" where eID ="+eid+";");
        s2.executeBatch();
        numberOfElements++;
        elementCounter++;
    }
}
```

The add at function runs in nlogn time since we have to search through the list, not knowing the order the elements are in, we have to search them to find the right one. The other operations update and insert will take logarithmic time and will typically be dwarfed by the runtime of the findI function.

```
void addFirst(int e) throws SQLException{
    if(numberOfElements == 0){
        add(e);
    }
    s.addBatch("INSERT INTO DoubleLL"+c.getName()+""+tableNumber+" VALUES ("+elementCounter+", "+e+", "+head+", null);");
    s.addBatch("UPDATE DoubleLL"+c.getName()+""+tableNumber+" SET pID = "+elementCounter+" where eID="+head+";");
    s.executeBatch();
    head = elementCounter;
    elementCounter++;
    numberOfElements++;
}

void addLast(int e) throws SQLException{
    add(e);
}
```

Adding to either end of the list is usually done in constant time for the SQLDoubleLL we will be doing this in logarithmic time to insert and update the list. The addLast just calls the add function since that will add the element to the end.

Add All

The addAll Function adds items to the SQLDoubleLL in nlogn time tossing all of the inserts into an SQL batch and then updates the tail and size of the list. If the collection e is empty the function will false and if the SQLDoubleLL is not empty they will be added to the end of the list:

```
boolean addAll(Collection<Integer> e) throws SQLException{
    if(e.isEmpty()){return false;}
    if(numberOfElements == 0){
        head = elementCounter;
        Iterator<Integer> k = e.iterator();
        s.addBatch("INSERT INTO DoubleLL"+c.getName()+""+tableNumber+" VALUES ("+elementCounter+", "+k.next()+", "+(++elementCounter)+", null);");
        while(k.hasNext()){
            s.addBatch("INSERT INTO DoubleLL"+c.getName()+""+tableNumber+" VALUES ("+elementCounter+", "+k.next()+", "+(++elementCounter)+", "+(elementCounter-2)+");");
        }
        s.addBatch("UPDATE DoubleLL"+c.getName()+""+tableNumber+" SET nID = null WHERE eID = "+(elementCounter-1)+";");
        s.executeBatch();
        tail = elementCounter-1;
        numberOfElements+=e.size();
        return true;
    }
    Iterator<Integer> k = e.iterator();
    s.addBatch("UPDATE DoubleLL"+c.getName()+""+tableNumber+" SET nID = "+elementCounter+" WHERE eID = "+tail+";");
    while(k.hasNext()){
        s.addBatch("INSERT INTO DoubleLL"+c.getName()+""+tableNumber+" VALUES ("+elementCounter+", "+k.next()+", "+(elementCounter+1)+", "+(elementCounter-1)+");");
        elementCounter++;
    }
    s.addBatch("UPDATE DoubleLL"+c.getName()+""+tableNumber+" SET nID = null WHERE eID = "+(elementCounter-1)+";");
    s.executeBatch();
    tail = elementCounter-1;
    numberOfElements+=e.size();
    return true;
}
```

The add all at function also add the items in nlogn time. If the index i passed to the function is either negative or greater than the number of elements in the list the function will throw an IndexOutOfBoundsException. If the index we wish to add at is at the end of the list the regular add all function is called to add the items. The new items are inserted into a batch and at the end the list is updated depending on if the added list is in the front or middle of the old list and change the values before and after the new entries.

```
boolean addAll(int i, Collection<Integer> e) throws SQLException{
    if(i<0||i>numberOfElements)throw new IndexOutOfBoundsException();
    if(e.isEmpty()){
        return false;
    }
    if(i==numberOfElements-1){
        return addAll(e);
    }
    int eid = findI(i);
    ResultSet r = s.executeQuery("select pID, nid from DoubleLL"+c.getName()+""+tableNumber+" where eID = "+eid+";");
    Iterator<Integer> k = e.iterator();
    int p = r.getInt(1);
    do{
        s.addBatch("INSERT INTO DoubleLL"+c.getName()+""+tableNumber+" values ("+elementCounter+", "+k.next()+", "+(elementCounter+1)+", "+p+");");
        p = elementCounter;
        elementCounter++;
    }while(k.hasNext());
    if(i==0){
        s.addBatch("UPDATE DoubleLL"+c.getName()+""+tableNumber+" SET pID = "+(elementCounter-1)+" where eID = "+head+";");
        head = elementCounter-e.size();
        s.addBatch("UPDATE DoubleLL"+c.getName()+""+tableNumber+" SET pID = null where eID = "+head+";");
    }else{
        s.addBatch("UPDATE DoubleLL"+c.getName()+""+tableNumber+" SET pID = "+(elementCounter-1)+" where eID = "+eid+";");
        s.addBatch("UPDATE DoubleLL"+c.getName()+""+tableNumber+" SET nID = "+(elementCounter-e.size())+" where nID = "+eid+";");
    }
    s.addBatch("UPDATE DoubleLL"+c.getName()+""+tableNumber+" SET nID = "+eid+" where eID = "+(elementCounter-1)+";");
    s.executeBatch();
    numberOfElements+=e.size();
    return true;
}
```

Get Set and Size

The get, set, and size methods for the SQLDoubleLL are mostly the same as the SQLinkedList. The get function now just calls findI instead of searching through the list.

```
int get(int i) throws SQLException{
    if (i<0||i>numberOfElements)throw new IndexOutOfBoundsException();
    int eid = findI(i);
    ResultSet r = s.executeQuery("select fID from DoubleLL"+c.getName()+""+tableNumber+" where eID = "+eid+";");
    return r.getInt(1);
}

int getFirst() throws SQLException{
    ResultSet r = s.executeQuery("select fid from DoubleLL"+c.getName()+""+tableNumber+" where eID = "+head+";");
    return r.getInt(1);
}

int getLast() throws SQLException{
    ResultSet r = s.executeQuery("select fid from DoubleLL"+c.getName()+""+tableNumber+" where eID = "+tail+";");
    return r.getInt(1);
}
```

The set funct now also uses the findI method to find where elements are in the list:

```
int set(int i, int e) throws SQLException{
    int eid = findI(i);
    ResultSet r = s.executeQuery("select fid from DoubleLL"+c.getName()+""+tableNumber+" where eID = "+eid+";");
    int o = r.getInt(1);
    s.execute("UPDATE DoubleLL"+c.getName()+""+tableNumber+" SET fID = "+e+" where eID = "+eid+";");
    return o;
}
```

The size has not changed and still return the number of elements:

```
int size(){
    return numberOfElements;
}

boolean isEmpty(){
    return (numberOfElements==0);
}
```

IndexOf and Contains

The IndexOf Function works the same as before:

```
int indexOf(int o) throws SQLException{
    int j=0;
    ResultSet r;
    int curr =head;
    do{
        r = s.executeQuery("SELECT nID, fID from DoubleLL"+c.getName()+""+tableNumber+" WHERE eID = "+curr+";");
        if(r.next()){
            if(o==r.getInt(2)){
                return j;
            }
            curr = r.getInt(1);
            j++;
        }
        else{
            return -1;
        }
    }while(curr!=tail);
    return -1;
}
```

The lastIndexOf function can be rewritten now that we can search through the list backwards we can search from the end of the list:

```
int lastIndexOf(int o) throws SQLException{
    int j=numberOfElements-1;
    ResultSet r;
    int curr =tail;
    do{
        r = s.executeQuery("SELECT pID, fID from DoubleLL"+c.getName()+""+tableNumber+" WHERE eID = "+curr+";");
        if(r.next()){
            if(o==r.getInt(2)){
                return j;
            }
            curr = r.getInt(1);
            j--;
        }
        else{
            return -1;
        }
    }while(curr!=head);
    return -1;
}
```

For the SQLDoubleLL we are going to add a contains functions to check if an item is in our list. Normally in a Linked LIst something like this would take linear time potentially checking every element but for this we can check in in logarithmic time if we have an index on the fid column.

```
boolean contains(int o) throws SQLException{
    ResultSet r= s.executeQuery("SELECT * from DoubleLL"+c.getName()+""+tableNumber+" WHERE fID = "+o+";");
    if(!r.next()){
        return false;
    }
    return true;
}
```

Remove

The remove works just like before removing the last element first element from the list updating the list's tails and returning the element.

```
int remove() throws SQLException{
    if(numberOfElements ==0)throw new NoSuchElementException();
    ResultSet r = s.executeQuery("SELECT pID, fID from DoubleLL"+c.getName()+""+tableNumber+" WHERE eID = "+tail+";");
    int a = r.getInt(1);
    int b = r.getInt(2);
    s.execute("DELETE from DoubleLL"+c.getName()+""+tableNumber+" WHERE eID = "+tail+";");
    s.execute("UPDATE DoubleLL"+c.getName()+""+tableNumber+" set nID = null where eID = "+a+";");
    tail = a;
    numberOfElements--;
    return b;
}
```

The remove at function now uses the findI method to find the element to be delete, then updates the the list and returns the the key of the element being deleted.

```
int remove(int i) throws SQLException{
    if(numberOfElements ==0)throw new NoSuchElementException();
    if(i<0||i>numberOfElements-1)throw new IndexOutOfBoundsException();
    int eid = findI(i);
    ResultSet r = s.executeQuery("SELECT nID, pID, fID from DoubleLL"+c.getName()+""+tableNumber+" WHERE eID = "+eid+";");
    int a = r.getInt(1);
    int b = r.getInt(2);
    int d = r.getInt(3);
    s.addBatch("UPDATE DoubleLL"+c.getName()+""+tableNumber+" set nID = "+a+" where nID = "+eid+";");
    s.addBatch("UPDATE DoubleLL"+c.getName()+""+tableNumber+" set pID = "+b+" where pID = "+eid+";");
    s.addBatch("DELETE from DoubleLL"+c.getName()+""+tableNumber+" WHERE eID = "+eid+";");
    s.executeBatch();
    numberOfElements--;
    return d;
}
```

The remove object function goes through the list checking if an element is equal to the one to be deleted and then deletes it. This will only remove the first instance in the list

```
boolean removeObject(int o) throws SQLException
{
    ResultSet r;
    int curr =head;
    do{
        r = s.executeQuery("SELECT nID, pID, fID  from DoubleLL"+c.getName()+""+tableNumber+" WHERE eID = "+curr+";");
        if(o==r.getInt(3)){
            s.execute("DELETE from DoubleLL"+c.getName()+""+tableNumber+" WHERE eID = "+curr+";");
            s.execute("UPDATE DoubleLL"+c.getName()+""+tableNumber+" set nID = "+r.getInt(1)+" where nID = "+curr+";");
            s.execute("UPDATE DoubleLL"+c.getName()+""+tableNumber+" set pID = "+r.getInt(2)+" where pID = "+curr+";");
            numberOfElements--;
            return true;
        }
        curr = r.getInt(1);

    }while(curr!=tail);
    return false;
}
```

There is also a function called removeFirstOccurrence which just calls the remove object function. This might seem a bit redundant but we want to mimic the behaviour of Java's LinkedList function so that if we wanted to supplement our SQLDoubleLL the expected behaviour should be the same.

```
boolean removeFirstOccurrence(int o) throws SQLException{
    return removeObject(o);

}

boolean removeLastOccurrence(int o) throws SQLException{
    ResultSet r;
    int curr =tail;
    do{
        r = s.executeQuery("SELECT nID, pID, fID  from DoubleLL"+c.getName()+""+tableNumber+" WHERE eID = "+curr+";");
        if(o==r.getInt(3)){
            s.execute("DELETE from DoubleLL"+c.getName()+""+tableNumber+" WHERE eID = "+curr+";");
            s.execute("UPDATE DoubleLL"+c.getName()+""+tableNumber+" set nID = "+r.getInt(1)+" where nID = "+curr+";");
            s.execute("UPDATE DoubleLL"+c.getName()+""+tableNumber+" set pID = "+r.getInt(2)+" where pID = "+curr+";");
            numberOfElements--;
            return true;
        }
        curr = r.getInt(1);
    }while(curr!=head);
    return false;
}
```

There is also a remove last occurrence which will search the list backwards looking for the last instance of the key and return false if the key is not there.

```
int removeFirst() throws SQLException{
    if(numberOfElements ==0)throw new NoSuchElementException();
    ResultSet r = s.executeQuery("SELECT nID, fID from DoubleLL"+c.getName()+""+tableNumber+" WHERE eID = "+head+";");
    int a = r.getInt(1);
    int b = r.getInt(2);
    s.addBatch("UPDATE DoubleLL"+c.getName()+""+tableNumber+" set pID = null where eID = "+a+";");
    s.addBatch("DELETE from DoubleLL"+c.getName()+""+tableNumber+" WHERE eID = "+head+";");

    s.executeBatch();
    head = a;
    numberOfElements--;
    return b;

}

int removeLast() throws SQLException{
    return remove();
}
```

The remove first works just like the remove did for the SQLLinkedList updating the head and returning the element. The remove last just calls the remove function.

```
void clear() throws SQLException{
    s.execute("DELETE FROM DoubleLL"+c.getName()+""+tableNumber+";");
    numberOfElements = 0;
    elementCounter = 0;
}
```

The clear function allows for the deletion of all of the elements in the list leaving empty list.

Queue and Stack

Java's LinkedList also contains the functions to allow it to act as a stack and a queue. Since one of our goals is to mimic the functions used by Java's LinkedList we are going to do the same. So for the queue we have:

```
boolean offer(int e) throws SQLException{
    addLast(e);
    return true;
}

boolean offerFirst(int e) throws SQLException{
    addFirst(e);
    return true;
}

boolean offerLast(int e) throws SQLException{
    addLast(e);
    return true;
}

int peek() throws SQLException{
    return getFirst();
}

int peekFirst() throws SQLException{
    return getFirst();
}

int peekLast() throws SQLException{
    return getLast();
}

int poll() throws SQLException{
    return removeFirst();
}

int pollFirst() throws SQLException{
    return removeFirst();
}

int pollLast() throws SQLException{
    return removeLast();
}
```

To add to the queue we have the offer, offerFirst and offerLast functions. PeekFirst and Peek just get the first element from the queue without removing and PeekLast returns the last element. Poll and PollFirst will remove and return the first element in the queue and PollLast will call removeLast and return the element.

```
int pop() throws SQLException{
    return removeLast();
}

void push(int e) throws SQLException{
    add(e);
}
```

The push function will add to the end of the list. The pop function will return and remove the last item in the list.

Tutorial

For this tutorial we are going to start by repeating what we did last time with the SQLLinkedList but with SQLDoubleLL instead. The results should be the same as before we have a list of bus stops that one bus makes. Now use the stack and queue features of the SQLDoubleLL.

First we will use a queue to find out how far away a stop is. To fill the queue we will get the stops for another route:

```
ResultSet r = stat.executeQuery("select sid from stop_times join stops on stop_times.stop_id=stops.stop_id where trip_id like \"%39413992%\";");
```

Then we will fill the queue using offer:

```
SQLDoubleLL<Stop> BusTripStops = new SQLDoubleLL<Stop>(Stop.class, database,"stops", "sid");
while(r.next()){
    BusTripStops.offer(r.getInt(1));
}
```

Once we have filled the queue we want to access the elements in the queue:

```
while(!BusTripStops.isEmpty()){
    System.out.println("SOUTH KEYS 2A is "+(BusTripStops.indexOf(3162))+" stops away");
    r =stat.executeQuery("Select stop_name from stops where sid ="+BusTripStops.poll()+";");
    System.out.println("The current stop is: "+r.getString(1));
    System.out.println();
}
```

This will print out the current stop along with how many there are to go until the desired stop.

Next we will use the same SQLDoubleLL to hold the route as a stack, but first we can use the clear function to reset the list.

BusTripStops.clear();

Now let's fill the stack:

```
r = stat.executeQuery("select sid from stop_times join stops on stop_times.stop_id=stops.stop_id where trip_id like \"%39414414%\" order by stop_times.stop_sequence desc;");
while(r.next()){
    BusTripStops.push(r.getInt(1));
}
```

And we can pop the elements off of the stack:

```
while(!BusTripStops.isEmpty()){

    r =stat.executeQuery("Select stop_name from stops where sid ="+BusTripStops.pop()+";");
    System.out.println("The current stop is: "+r.getString(1));
}
```

Graph

To store a graph we are going to use two tables, one for the vertices and one for the edges. The edge table is going to have four columns, the “eid” column, the “startId” column for where the edge starts, and the “endId” for where the edge ends and “weight” column to give the edge a weight value. The values in the startid and endid columns reference the vertices table.

Constructor

The constructor will work just like the previous ones but we will need two tables this time. We will also need counters for edges and vertices.

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.LinkedList;

public class SQLGraph<T> {
    int numberOfVertices;
    int numberOfEdges;
    int vertexCounter;
    int edgeCounter;
    int tableNumber;
    String colName;
    String tableName;
    Connection theDatabase;
    Statement s;
    Class<T> c;

    public SQLGraph(Class<T> vClass, Connection aDatabase, String _tableName, String _colName) throws SQLException{
        numberOfVertices = 0;
        numberOfEdges = 0;
        vertexCounter = 0;
        edgeCounter = 0;
        c = vClass;
        theDatabase = aDatabase;
        colName = _colName;
        tableName = _tableName;
        s = theDatabase.createStatement();
        s.execute("CREATE TABLE IF NOT EXISTS GRAPHS(lID INTEGER PRIMARY KEY AUTOINCREMENT, GraphName TEXT NOT NULL);");
        s.execute("INSERT INTO GRAPHS (GraphName) VALUES ('"+c.getName()+"');");
        ResultSet a= s.executeQuery("SELECT COUNT (*) from GRAPHS;");
        tableNumber = a.getRow();
        s.execute("CREATE TABLE IF NOT EXISTS Graph"+c.getName()+""+tableNumber+"Vertices (vID INTEGER NOT NULL, fID INTEGER, "
                + "FOREIGN KEY(fID) REFERENCES "+tableName+"("+colName+"));");
        s.execute("CREATE TABLE IF NOT EXISTS Graph"+c.getName()+""+tableNumber+"Edges (eID INTEGER NOT NULL, startID INTEGER, "
                + "endID INTEGER, weight Integer, FOREIGN KEY(startID) REFERENCES Graph"+c.getName()+""+tableNumber+"Vertices(vID), "
                        + "FOREIGN KEY(endID) REFERENCES Graph"+c.getName()+""+tableNumber+"Vertices(vID));");
    }
```

One thing we have not considered yet is how to reload the data structure from the database for that we are going to add a new constructor to reload the graph

```
public SQLGraph(Class<T> vClass, Connection aDatabase, String _tableName, String _colName, int _tableNumber) throws SQLException{
    c = vClass;
    theDatabase = aDatabase;
    colName = _colName;
    tableName = _tableName;
    s = theDatabase.createStatement();
    ResultSet r = s.executeQuery("select count(*), max(vid) from Graph"+c.getName()+""+tableNumber+"Vertices;");
    numberOfVertices = r.getInt(1);
    vertexCounter = (r.getInt(2))+1;
    r = s.executeQuery("select count(*), max(eid) from Graph"+c.getName()+""+tableNumber+"Edges;");
    numberOfEdges = r.getInt(1);
    edgeCounter = (r.getInt(2))+1;
}
```

Here the graph reloads based on the table number used before when making the graph. The values are restored and we are reconnected to the tables in the database.

Insertion

For insertion we have two types insertion of edges and insertion of vertices.
Since we are now working with a sets of vertices and edges we do not want duplicates. Now when we add to our graph , were going to check if there are duplicates and add new edges and vertices only if they are not already present But sometimes it might be useful to count the duplicates we encounter in a graph so there is an option to count duplicates and have them use the count as the weight of the edge.

For the vertex insert we just have the search to check if the vertex is already in the vertex list and the the insertion if it is not giving a logarithmic time for the operation.

```
boolean addVertex(int v) throws SQLException{
    ResultSet r= s.executeQuery("SELECT * from Graph"+c.getName()+""+tableNumber+"Vertices WHERE fID = "+v+";");
    if(!r.next()){
        s.execute("INSERT INTO Graph"+c.getName()+""+tableNumber+"Vertices (vID, fID) values ("+(++vertexCounter)+", "+v+");");
        numberOfVertices++;
        return true;
    }
    return false;
}
```

Like before we may also want to add a collection of vertices for that there we have a add vertex function for a list. This can be done in nlogn time searching for each vertex and then inserting into the set:

```
boolean addAllVertext(Collection<Integer> v)throws SQLException{
    for(int q:v){
        ResultSet r= s.executeQuery("SELECT * from Graph"+c.getName()+""+tableNumber+"Vertices WHERE fID = "+q+";");
        if(!r.next()){
            s.execute("INSERT INTO Graph"+c.getName()+""+tableNumber+"Vertices (vID, fID) values ("+(++vertexCounter)+", "+q+");");
        }
    }
    numberOfVertices+=v.size();
    return true;
}
```

And for adding edges take nolgn time for searching and inserting but were also going to have the option to count duplicate edges, update the existing edge's weight instead of inserting a new vertex:
//code for adding edges

```
boolean addEdge(int a, int b, boolean Dubs) throws SQLException{
    ResultSet r= s.executeQuery("SELECT * from Graph"+c.getName()+""+tableNumber+"Edges WHERE startID = "+a+" AND endID = "+b+";");
    if(!r.next()){
        s.execute("INSERT INTO Graph"+c.getName()+""+tableNumber+"Edges (eID, startID, endID, weight) values ("+(++edgeCounter)+", "+a+", "+b+", 1);");
        numberOfEdges++;
        return true;
    }
    if(Dubs){
    s.execute("UPDATE Graph"+c.getName()+""+tableNumber+"Edges SET weight = "+(r.getInt(4)+1)+" WHERE startID = "+a+" AND endID = "+b+";");
    }
    return false;
}
```

If instead we want to give our edges certain values we can do that as well:

```
boolean addEdge(int a, int b, int w, boolean Dubs) throws SQLException{
    ResultSet r= s.executeQuery("SELECT * from Graph"+c.getName()+""+tableNumber+"Edges WHERE startID = "+a+" AND endID = "+b+";");
    if(!r.next()){
        s.execute("INSERT INTO Graph"+c.getName()+""+tableNumber+"Edges (eID, startID, endID, weight) values ("+(++edgeCounter)+", "+a+", "+b+", "+w+");");
        numberOfEdges++;
        return true;
    }
    if(Dubs){
    s.execute("UPDATE Graph"+c.getName()+""+tableNumber+"Edges SET weight = "+(r.getInt(4)+1)+" WHERE startID = "+a+" AND endID = "+b+";");
    }
    return false;
}
```

This will also take nlogn time, but if we have a duplicate edge we will just not add it to the list. We also have the option add the edges from a pair of collections:

```
boolean addAllEdge(Collection<Integer>a, Collection<Integer>b, boolean Dubs) throws SQLException{
    if(a.size()!=b.size()){
        return false;
    }
    Iterator<Integer> A = a.iterator();
    Iterator<Integer> B = b.iterator();
    ResultSet r;
    LinkedList<String>sqlStrings = new LinkedList<String>();
    while(A.hasNext()){
        int q = A.next();
        int z = B.next();
        sqlStrings.add("Create temp table IF NOT EXISTS hgf (eID INTEGER NOT NULL, startID INTEGER, endID INTEGER, weight Integer);");
        r= s.executeQuery("SELECT * from Graph"+c.getName()+""+tableNumber+"Edges WHERE startID = "+q+" AND endID = "+z+";");
        if(r.next()&&Dubs){
            sqlStrings.add("UPDATE Graph"+c.getName()+""+tableNumber+"Edges SET weight = "+(r.getInt(4)+1)+" WHERE startID = "+q+" AND endID = "+z+";");
        }else{

            sqlStrings.add("INSERT INTO hgf (eID, startID, endID, weight) values ("+(++edgeCounter)+", "+q+", "+z+", "+1+");");
            numberOfEdges++;
        }
    }
    for(String q :sqlStrings){
        s.addBatch(q);
    }
    s.executeBatch();
    if(Dubs){
        s.execute("Insert into Graph"+c.getName()+""+tableNumber+"Edges select eid, startid, endid, count(*) as weight from hgf group by startid, endid;");
    }
    else{
        s.execute("Insert into Graph"+c.getName()+""+tableNumber+"Edges select eid, startid, endid, weight as weight from hgf group by startid, endid;");
    }
    return true;
}
```

If we have a third collection of values to use for weight we can load that as well:

```
boolean addAllEdge(Collection<Integer>a, Collection<Integer>b, Collection<Integer>w) throws SQLException{
    if(a.size()==b.size()&&a.size()==w.size()){

        Iterator<Integer> A = a.iterator();
        Iterator<Integer> B = b.iterator();
        Iterator<Integer> W = w.iterator();
        ResultSet r;
        LinkedList<String> sqlStrings = new LinkedList<String>();
        sqlStrings.add("Create table IF NOT EXISTS ZX (eID INTEGER NOT NULL, startID INTEGER, endID INTEGER, weight Integer;");
        while(A.hasNext()){
            int z = A.next();
            int q = B.next();
            int x = W.next();
            r= s.executeQuery("SELECT * from Graph"+c.getName()+""+tableNumber+"Edges WHERE startID = "+q+" AND endID = "+z+";");
            if(r.next()){
            }else{
                sqlStrings.add("INSERT INTO ZX (eID, startID, endID, weight) values ("+(++edgeCounter)+", "+q+", "+z+", "+x+");");
                numberOfEdges++;
            }
        }
        for(String q : sqlStrings){
            s.addBatch(q);
        }
        s.executeBatch();
        return true;
    }
    return false;
}
```

Remove

Removing a vertex or an edge works just that same as it did for the list functions, but if we delete a vertex will also need to delete the edges that connected to it. All of these operations will be done in logarithmic time.

```
boolean removeVertex(int v) throws SQLException{
    ResultSet r= s.executeQuery("SELECT * from Graph"+c.getName()+""+tableNumber+"Vertices WHERE fID = "+v+";");
    if(!r.next()){
        return false;
    }
    s.addBatch("DELETE FROM Graph"+c.getName()+""+tableNumber+"Edges Where startID = "+v+" OR endID = "+v+";");
    s.addBatch("DELETE FROM Graph"+c.getName()+""+tableNumber+"Vertices Where fID = "+v+";");
    s.executeBatch();
    r = s.executeQuery("Select count(*) from Graph"+c.getName()+""+tableNumber+"Edges;");
    numberOfEdges = r.getInt(1);
    numberOfVertices--;
    return true;
}
```

Deleting an edge only requires deleting the edge itself:

```
boolean removeEdge(int a, int b) throws SQLException{
    ResultSet r= s.executeQuery("SELECT * from Graph"+c.getName()+""+tableNumber+"Edges WHERE startID = "+a+" AND endID = "+b+";");
    if(!r.next()){
        return false;
    }
    s.execute("DELETE FROM Graph"+c.getName()+""+tableNumber+"Edges where startID = "+a+" AND endID = "+b+";");
    numberOfEdges--;
    return true;
}
```

Graph Functions

Since the graph is a bit different than a list we have some operations that we will want to do such finding out if an edge exist between two vertices or checking out how many edges live a particular vertex.

If we want to check if two vertices are connected we can do this by checking if there is an edge from a to b or from b to a. This can be done in linear time or logarithmic if we have indices for the startID and endID:

//code for boolean adjacent

```
boolean adjacent(int a, int b) throws SQLException{
    ResultSet r= s.executeQuery("SELECT * from Graph"+c.getName()+""+tableNumber+"Edges WHERE startID = "+a+" AND endID = "+b+";");
    if(!r.next()){
        return false;
    }
    r= s.executeQuery("SELECT * from Graph"+c.getName()+""+tableNumber+"Edges WHERE startID = "+b+" AND endID = "+a+";");
    if(!r.next()){
        return false;
    }
    return true;
}
```

We also may want to find what vertices are adjacent to a vertex this is done by returning a list of the end vertices of edges that have the vertex in question as a start vertex. Again this will be linear if we do not have the startid indexed:

```
ArrayList<Integer> adjacent(int v) throws SQLException{
    ArrayList<Integer> n = new ArrayList<Integer>();
    ResultSet r = s.executeQuery("Select startID from Graph"+c.getName()+""+tableNumber+"Edges where endID = "+v+";");
    while(r.next()){
        n.add(r.getInt(1));
    }
    r = s.executeQuery("Select endID from Graph"+c.getName()+""+tableNumber+"Edges where startID = "+v+";");
    while(r.next()){
        n.add(r.getInt(1));
    }
    return n;
}
```

There are also functions for getting lists of vertices that have a vertex as a source, destination or either:

```
ArrayList<Integer> incedentEdges(int v) throws SQLException{
    ArrayList<Integer>n = new ArrayList<Integer>();
    ResultSet r = s.executeQuery("Select eID from Graph"+c.getName()+""+tableNumber+"Edges where startID ="+v+" OR endID = "+v+";");
    while(r.next()){
        n.add(r.getInt(1));
    }

    return n;
}

ArrayList<Integer> outEdges(int v) throws SQLException{
    ArrayList<Integer> n = new ArrayList<Integer>();
    ResultSet r = s.executeQuery("Select eID from Graph"+c.getName()+""+tableNumber+"Edges where startID = "+v+";");
    while(r.next()){
        n.add(r.getInt(1));
    }
    return n;
}

ArrayList<Integer> inEdges(int v) throws SQLException{
    ArrayList<Integer> n = new ArrayList<Integer>();
    ResultSet r = s.executeQuery("Select eID from Graph"+c.getName()+""+tableNumber+"Edges where endID = "+v+";");
    while(r.next()){
        n.add(r.getInt(1));
    }
    return n;
}
```

Printing

For printing we are going to want to print both the vertices as well as the edges:

```
String listPrint(String label) throws SQLException{
    String trt="";
    ResultSet r = s.executeQuery("select vid, fid, "+label+" from Graph"+c.getName()+""+tableNumber+"Vertices join "+tableName+" on fid="+colName+";");
    while(r.next()){
        trt+= r.getString(1)+" | ";
        trt+= r.getString(2)+" | ";
        trt+= r.getString(3);
        trt+="\n";
    }
    r = s.executeQuery("select eid, startid, endid, weight from Graph"+c.getName()+""+tableNumber+"Edges;");
    while(r.next()){
        trt+= r.getString(1)+" | ";
        trt+= r.getString(2)+" | ";
        trt+= r.getString(3)+" | ";
        trt+= r.getString(4);
        trt+="\n";
    }
    return trt;

}
```

Tutorial

In this tutorial we are going to uses the bus database to make a graph to plot the location of bus stops and bus paths around the city, using the stops as vertices and the the connections between stops as edges.

For this we are going to do this tutorial in two parts, first we are going generate the data for the edges since it is not explicitly stated in the data, then we are going to use the graph the we've generated to display the stops and their connections.

Were going to start like we have before connecting to the database search the database for the data we want and store it in the graph. First we will find the vertices:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.LinkedList;

public class BusTutorialGraphA {
    public static void main(String[] args) throws Exception {
        try {
            Class.forName("org.sqlite.JDBC");
            Connection database = DriverManager.getConnection("jdbc:sqlite:");
            Statement stat = database.createStatement();
            stat.executeUpdate("restore from OCBus.db");

            LinkedList<Integer> stopList = new LinkedList<Integer>();
            ResultSet r =  stat.executeQuery("Select sid from stops;");
            SQLGraph<Stop> MyMap = new SQLGraph<Stop>(Stop.class,database, "stops", "sid");
            while(r.next()){
                 stopList.add(r.getInt(1));
            }
            MyMap.addAllVertext(stopList);
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }
}
```

Since we are using stops as our vertices we use this as the class for the SQLGraph.

Once we have the vertices done we can make the edges. This is going to require a bit more work and knowledge about the data we are using. What we want is the connections between bus stops that are taken by the busses. So we are going to need the order in which busses stop at the stops. Luckily this data is in the database in the stop_times table as stop_sequence as well as the stop_id's for the stops. Joining the stop_times table with the stops table we can get a list of the stops for each trip and the sort them in order in which the stop:

```
ResultSet g = stat.executeQuery("select stop_times.trip_id, stop_times.stop_sequence, stops.sid from stop_times "
        + "join stops on stop_times.stop_id = stops.stop_id "
        + "order by stop_times.trip_id asc, stop_times.stop_sequence asc;");
```

Now to generate the edges were going to find two sequential stops and put an edge between them. To do this we are going to make two lists and insert the vertices into both lists, then shifting one list at the end to make the edges. To know when we switch from one route another we check that the sequence number is not lower than the last time we added an edge:

```
int last = Integer.MAX_VALUE;
LinkedList<Integer> stopTimes = new LinkedList<Integer>();
LinkedList<Integer> stopTimes2 = new LinkedList<Integer>();
while(g.next()){
    if(last > g.getInt(2)){
        if(!stopTimes.isEmpty()){
        stopTimes.removeLast();
        }
        stopTimes.add(g.getInt(3));

    }
    else{
        stopTimes.add(g.getInt(3));
        stopTimes2.add(g.getInt(3));
    }
    last = g.getInt(2);
}
stopTimes.removeLast();
MyMap.addAllEdge(stopTimes, stopTimes2, true);
```

The last line here inserts the edges into the set and has the count duplicates marked true. Since we do not want edges to appear more than once, it might be useful to know how many trips use a particular part of road is used, so we count the number of times each edge appears in the data.

The last thing we are going to do is backup the database to a fresh file

stat.executeUpdate("backup to bus.db");

This will create a second database file with our graph in it. Generating the graph will take several minutes and we do not want to regenerate the graph every time we use it. We are going to do the generation as one program and then use the graph in another.
Run the generation program and we will start making the map program.

To make the map we will use JavaFX to draw dots and lines for our map. We are going to have a few additional requirements for our set up:

```
import java.util.ArrayList;
import javafx.application.Application;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import javafx.scene.shape.Line;

public class JavaFXSkeleton extends Application {
    @Override
    public void start(Stage primaryStage) {
        try {
            Group root = new Group();
            Connection database;
            Statement stat;
            Statement stat2;
            Class.forName("org.sqlite.JDBC");
            database = DriverManager.getConnection("jdbc:sqlite:");
            stat = database.createStatement();
            stat2 = database.createStatement();
            stat.executeUpdate("restore from bus.db");
            Scene scene = new Scene(root, 1000, 1000, Color.BLACK);
            primaryStage.setScene(scene);
            primaryStage.show();
        }
        catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

This will create a 1000x1000 pixel window where we are going to display our map. The vertices are going to be plotted with the longitude and latitude data provided in the stops table. But first we need to load the database from before:

```
stat.executeUpdate("restore from bus.db");
```

Next we going to need to create the Java representation of our graph:

```
SQLGraph<Stop> MyMap = new SQLGraph<Stop>(Stop.class,database, "stops", "sid", 0);
```

Next we are going to find the longitudinal and latitudinal ranges of our points. We will use this data to plot the stops on the map:

```
stat.executeUpdate("restore from bus.db");
Scene scene = new Scene(root, 1000, 1000, Color.BLACK);
SQLGraph<Stop> MyMap = new SQLGraph<Stop>(Stop.class,database, "stops", "sid", 0);
ResultSet r = stat.executeQuery("select min(stop_lat), max(stop_lat), min(stop_lon), max(stop_lon) from stops;");
minLong = r.getDouble(3);
minLat =r.getDouble(1);
maxLong = r.getDouble(4);
maxLat = r.getDouble(2);
```

Now take a look at the function mapPoint:

```
private int[] mapPoint(double lat, double lon){

    int Y = (int)(((-1)*Math.floor(((lat-minLat)/(maxLat-minLat))*1000))+1000);
    int X = (int)( Math.floor(((lon-minLong)/(maxLong-minLong))*1000));
    int[] xy = new int[]{X, Y};
    return xy;
}
```

We can plot the points on the screen using the relative position of the point with respect to the minimum and maximum points in each direction. A little more work is required to get the latitude into screen coordinates.

Now we are going to place the stops on the map:

```
r = stat.executeQuery("select stop_lat,stop_lon from stops;");
Group sDots = new Group();
int circleSize = 1;
while(r.next()){
    Circle c = new Circle(circleSize, Color.RED);
    int [] pos = mapPoint(r.getDouble(1), r.getDouble(2));
    c.setLayoutX(pos[0]);
    c.setLayoutY(pos[1]);

    sDots.getChildren().add(c);
}
```

This will retrieve all of the coordinates for the stops, find the location to put the circle on the screen and then add it to the group of stops.

Next we will add the edges from the graph

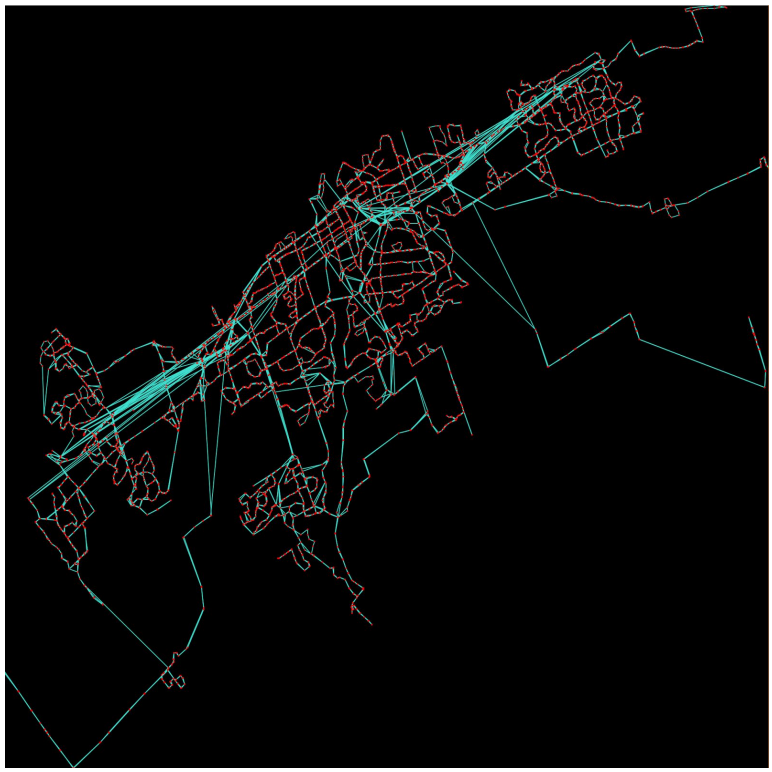
```
Group path = new Group();
ResultSet f = stat.executeQuery("select max(weight), min(weight) from "+MyMap.getName()+"Edges;");
maxWeight = f.getDouble(1);
minWeight = f.getDouble(2);
f = stat.executeQuery("select * from "+MyMap.getName()+"Edges;");
while(f.next()){

    r = stat2.executeQuery("select stop_lat,stop_lon from stops where sid = "+f.getInt(2)+";");
    int []xy1 = mapPoint(r.getDouble(1), r.getDouble(2));
    r = stat2.executeQuery("select stop_lat,stop_lon from stops where sid = "+f.getInt(3)+";");
    int []xy2 = mapPoint(r.getDouble(1), r.getDouble(2));
    Line l = new Line(xy1[0], xy1[1], xy2[0], xy2[1]);
    l.setStroke(Color.TURQUOISE);
    path.getChildren().add(l);

}
```

So here we grab the list of edges from the graph and make a line from the starting position to the end position.

Now if we add our stops and paths to the scene we can run the program and see a map of the stops and bus routes in Ottawa.



Now we are going to try out the adjacent function to see how far we can go if we only want to take a certain number of stops.

In our graph Elmvale mall is the stop with sid equal 457, to see where we can get to in eight stops we can run:

```
ArrayList<Integer> stopList  =MyMap.adjacent(457);
for(int i = 0; i< 7; i++){
    ArrayList<Integer> nList = new ArrayList<Integer>();
    for(int k :stopList){
        nList.addAll(MyMap.adjacent(k));
    }
    nList.removeAll(stopList);
    stopList.addAll(nList);
}
```

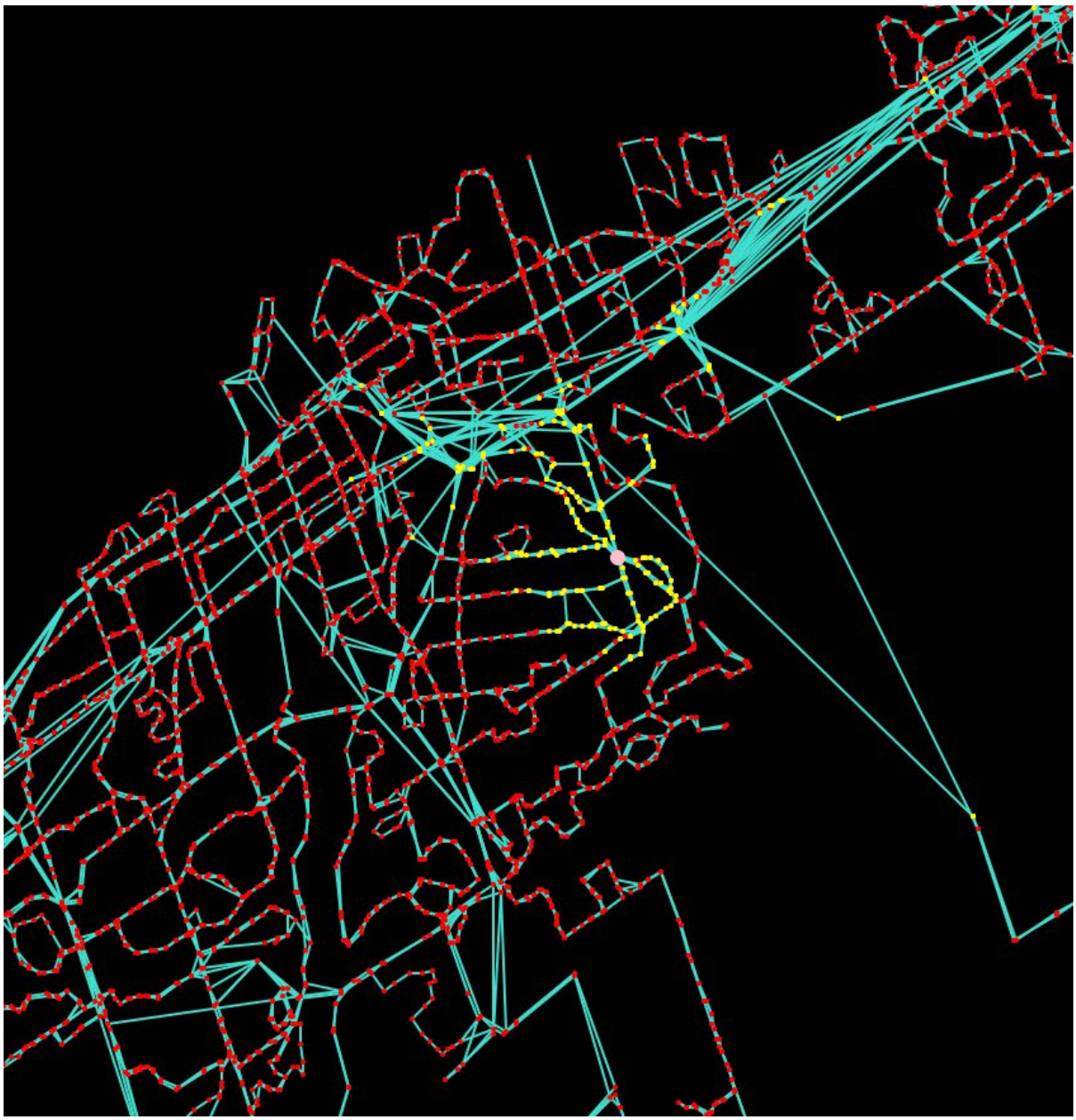
Now we can plot this new group of stops in another colour and see on the map where we can reach

```
  r = stat.executeQuery("select stop_lat,stop_lon from stops where sid = 457;");
  Circle c = new Circle(circleSize*3, Color.PINK);
  int [] pos = mapPoint(r.getDouble(1), r.getDouble(2));
  c.setLayoutX(pos[0]);
  c.setLayoutY(pos[1]);
  cDots.getChildren().add(c);
  root.getChildren().add(path);
  root.getChildren().add(sDots);
  root.getChildren().add(cDots);
  stat.executeUpdate("backup to bus.db");
  primaryStage.setScene(scene);
  primaryStage.show();
```

This gives us a cluster of yellow circles in the middle of the map. To help us see where the initial stop is, recolour it with another larger dot:

```
r = stat.executeQuery("select stop_lat,stop_lon from stops where sid = 457;");
Circle c = new Circle(circleSize*3, Color.PINK);
int [] pos = mapPoint(r.getDouble(1), r.getDouble(2));
c.setLayoutX(pos[0]);
c.setLayoutY(pos[1]);
cDots.getChildren().add(c);
```

The close up results of running the code:



Binary Search Tree

The binary search tree is the last data structure we will look at. The binary tree is going to be stored a bit like the linked lists except instead of having a previous and next element we are going to have a parent, left child and right child. These will have their own columns in the table representation of the binary search tree.

Constructor

The code here will be just the list classes we used before just with a different table setup:

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;

public class SQLBST <T>{
    int numberOfElements;
    int tableNumber;
    Connection theDatabase;
    String tableName;
    String colName;
    Statement s;
    Class<T> c;
    int root =0;
    public SQLBST(Class<T> aClass, Connection aDatabase, String _tableName, String _colName)throws SQLException{
        numberOfElements = 0;
        c = aClass;
        theDatabase = aDatabase;
        tableName = _tableName;
        colName = _colName;
        s = theDatabase.createStatement();
        s.execute("CREATE TABLE IF NOT EXISTS BST(lID INTEGER PRIMARY KEY AUTOINCREMENT, BSTName TEXT NOT NULL);");
        s.execute("INSERT INTO BST (BSTName) VALUES ('"+c.getName()+"');");
        ResultSet a= s.executeQuery("SELECT COUNT (*) from BST;");
        tableNumber = a.getRow();
        s.execute("CREATE TABLE IF NOT EXISTS BST"+c.getName()+""+tableNumber+" (fID INTEGER,parent INTEGER, "
                + "lChild INTEGER, rChild INTEGER, FOREIGN KEY(fID) REFERENCES "+tableName+"("+colName+"));");
    }
```

Insertion

For the insertion we are going to put the elements in one at a time to insert them properly into the tree. The tree is not self balancing so the shape of the tree will not be perfect. The cost of inserting an element will be doing several searches in logarithmic time, and we do this at most the height of the tree. Since the tree is not self balancing, the height of the tree is not logarithmic of the number of elements.

```
void insert(int q) throws SQLException{
    int pNode=0;
    int cNode=root;
    ResultSet r;
    if(root==0){
        root = q;
        s.execute("Insert into BST"+c.getName()+""+tableNumber+" values ("+q+", 0, 0, 0);");

    }
    else{
        while (cNode!=0){
            pNode = cNode;
            if(q==cNode){
                return;
            }
            else if(q<cNode){
                r = s.executeQuery("Select lChild from BST"+c.getName()+""+tableNumber+" where fid = "+cNode+";");
                cNode = r.getInt(1);
            }
            else{
                r = s.executeQuery("Select rChild from BST"+c.getName()+""+tableNumber+" where fid = "+cNode+";");
                cNode = r.getInt(1);
            }
        }
        if(q<pNode){
            s.addBatch("Insert into BST"+c.getName()+""+tableNumber+" values ("+q+", "+pNode+", 0, 0);");
            s.addBatch("UPDATE BST"+c.getName()+""+tableNumber+" set lChild = "+q+" where fid = "+pNode+";");
            s.executeBatch();
        }
        else{
            s.addBatch("Insert into BST"+c.getName()+""+tableNumber+" values ("+q+", "+pNode+", 0, 0);");
            s.addBatch("UPDATE BST"+c.getName()+""+tableNumber+" set rChild = "+q+" where fid = "+pNode+";");
            s.executeBatch();
        }
    }
    numberOfElements++;
}
```

Deletion

To delete from the tree the element is searched for then the smaller of its children replaces it in the tree to do that we are going to use another function transplant to move one branch of the tree to another part of the tree. Transplant has a logarithmic runtime since it will do several searches.

```
void transplant(int q, int w) throws SQLException{
    int cNode = w;
    if(root == q){
        root = w;
    }
    else{
        ResultSet r = s.executeQuery("select fid from BST"+c.getName()+""+tableNumber+" where lChild = "+q+";");
        if(r.next()){
            cNode = r.getInt(1);
            s.addBatch("UPDATE BST"+c.getName()+""+tableNumber+" set lChild = "+w+" where lChild = "+q+";");
        }
        else{
            r = s.executeQuery("select fid from BST"+c.getName()+""+tableNumber+" where rChild = "+q+";");
            cNode = r.getInt(1);
            s.addBatch("UPDATE BST"+c.getName()+""+tableNumber+" set rChild = "+w+" where rChild = "+q+";");
        }
        if(w!=0){
            s.addBatch("UPDATE BST"+c.getName()+""+tableNumber+" set parent = "+cNode+" where fid = "+w+";");
        }
        s.executeBatch();
    }

}
```

The deletion itself will also be in logarithmic time:

```
void delete(int q) throws SQLException{
    ResultSet r = s.executeQuery("select lChild, rChild from BST"+c.getName()+""+tableNumber+" where fid = "+q+";");
    int lChild=  r.getInt(1);
    int rChild = r.getInt(2);
    if(lChild == 0){
        transplant(q, rChild);
    }
    else if(rChild == 0){
        transplant(q, lChild);
    }
    else{
        int cNode = minimum(rChild);
        r = s.executeQuery("select parent, rChild from BST"+c.getName()+""+tableNumber+" where fid = "+cNode+";");
        if(r.getInt(1)==q){
            int f =  r.getInt(2);
            transplant(cNode, f);
            s.execute("UPDATE BST"+c.getName()+""+tableNumber+" set rChild = "+f+" where fid = "+cNode+";");
            s.execute("UPDATE BST"+c.getName()+""+tableNumber+" set parent = "+cNode+" where fid = "+f+";");
        }
        transplant(q, cNode);
        s.execute("UPDATE BST"+c.getName()+""+tableNumber+" set lChild = "+lChild+" where fid = "+cNode+";");
        s.execute("UPDATE BST"+c.getName()+""+tableNumber+" set parent = "+cNode+" where fid = "+lChild+";");
    }
    s.execute("DELETE from BST"+c.getName()+""+tableNumber+" where fid = "+q+";");
}
```

Binary Search Tree Functions

The tree walk function returns the elements of the tree in the order they would appear in a depth first search:

```
ArrayList<Integer> treeWalk() throws SQLException{
    ArrayList<Integer> trt = new ArrayList<Integer>();
    int u =root; int prev =0; int next;
    while(u!=0){
        ResultSet r = s.executeQuery("Select parent, lChild, rChild, fid from BST"+c.getName()+""+tableNumber+" where fid ="+u+";");
        int parent = r.getInt(1);
        int lChild = r.getInt(2);
        int rChild = r.getInt(3);
        int fid = r.getInt(4);
        if (prev == parent) {
            if (lChild != 0) next = lChild;
            else if (rChild != 0) next = rChild;
            else next = parent;
        }
        else if (prev == lChild) {
            if (rChild != 0) next = rChild;
            else next = parent;
        }
        else {
            next = parent;
            trt.add(fid);
        }
        prev = u;
        u = next;
    }
    return trt;

}
```

There is also a search function, it will return the element if the element is in the list and the closest if it is not.

```
int treeSearch(int k) throws SQLException{
    int cNode = root;
    ResultSet r = s.executeQuery("Select fid from BST"+c.getName()+""+tableNumber+" where fid = "+root+";");
    int fid = r.getInt(1);
    while(cNode!=0 && fid !=k){
        if(fid < k){
            r= s.executeQuery("Select lChild, fid from BST"+c.getName()+""+tableNumber+" where fid = "+cNode+";");
            cNode = r.getInt(1);
            fid = r.getInt(2);
        }
        else{
            r= s.executeQuery("Select rChild, fid from BST"+c.getName()+""+tableNumber+" where fid = "+cNode+";");
            cNode = r.getInt(1);
            fid = r.getInt(2);
        }
    }
    return cNode;
}
```

Finding the minimum and maximum is done by going down the leftmost or rightmost branches of the tree to find the proper element:

```
int minimum(int q) throws SQLException{
    int cNode =q;
    ResultSet r = s.executeQuery("Select lChild from BST"+c.getName()+""+tableNumber+" where fid ="+cNode+";");
    int lChild = r.getInt(1);
    while(lChild !=0){
        cNode = lChild;
        r= s.executeQuery("Select lChild from BST"+c.getName()+""+tableNumber+" where fid = "+cNode+";");
        lChild = r.getInt(1);
    }
    return cNode;
}

int maximum(int q) throws SQLException{
    int cNode =q;
    ResultSet r = s.executeQuery("Select rChild from BST"+c.getName()+""+tableNumber+" where fid ="+cNode+";");
    int rChild = r.getInt(1);
    while(rChild !=0){
        cNode = rChild;
        r= s.executeQuery("Select rChild from BST"+c.getName()+""+tableNumber+" where fid = "+cNode+";");
        rChild = r.getInt(1);
    }
    return cNode;
}
```

We can find out how many elements are in the list with the size function:

```
int size() throws SQLException{
    ResultSet r = s.executeQuery("SELECT COUNT (*) from BST"+c.getName()+""+tableNumber+";");
    return r.getInt(1);
}
```

The height the tree is found recursively, looking for the path with the longest distance from root to leaf:

```
int height(int q) throws SQLException{
    if (q==0){
        return -1;
    }
    ResultSet r = s.executeQuery("SELECT lChild, rChild from BST"+c.getName()+""+tableNumber+" where fid = "+q+";");
    int l = r.getInt(1);
    int w = r.getInt(2);
    return 1+ java.lang.Math.max(height(l),height(w));
}
```

Tutorial

In this tutorial we are going to create a binary search tree for the stops based the bus stop's stop code. The stop code is the number displayed at the bus stop. First we are going to create the binary search tree:

```
SQLBST<Stop> BusStops = new SQLBST<Stop>(Stop.class, database,"stops", "stop_code");
while(r.next()){
    BusStops.insert(r.getInt(1));
}
```

Next we can search for a stop:

```
System.out.println("Tree Search 8791: "+BusStops.treeSearch(8791));
```

We can find the height of the tree:

```
System.out.println("Tree height: "+BusStops.height(BusStops.root));
```

Find the minimum and maximum:

```
System.out.println("Tree minimum: "+BusStops.minimum(BusStops.root));
System.out.println("Tree maximum: "+BusStops.maximum(BusStops.root));
```

And we can delete a node from the tree and still have a tree:

```
BusStops.delete(4627);
System.out.println(BusStops.listPrint("stop_name"));
```

Conclusion

There are some advantages and disadvantages with this approach to making data structures. The most noticeable disadvantage is the performance of moving data back and forth into and out of the database. Every operation we do ends up getting a few logarithmic operations tacked on to get information where we want it. Another disadvantage is in the programming itself coding between two languages can make it difficult to find where the bugs are. A common problem caused by using the Java and SQL together is if an entry has a null value and we ask for an integer we receive back the number 0. This can cause problems if we are checking for indexes to other tables or need the value for a computation and might happen if we are missing data. Additionally since all of the operations use strings problems can arise from getting the escape characters right for both languages.

Some advantages are having the data structure in the database allows for the data to be restored without having to be regenerated as we saw with the bus map in the graph tutorial. By using the database we can uses indices to search through the data faster. The table representation for the graph can operate a bit more efficiently than adjacency list if we are searching for data in logarithmic time using indices versus iterating through and adjacency list. Since we never create new copies of the data and always refer to the existing data we can save memory.

Going forward there might be a nontraditional data structure, such as neural network, that might work with this approach to data in a database. Further developing the graph to have more algorithms available would make that structure more useful. Using better visualization methods could help better convey the data to the reader. The binary search tree has slow performance since there is currently no way to add a collection at once. This project was done using a relational database, further work could see how this approach could with an object database or a graph database.

The idea could also be improved if some more of the work was done in the server side of the database. For instance if the database knew we were making an arraylist it could shift the elements around without us having a large number of updates to the database.

References

Source for dataset:
http://data.ottawa.ca/en/dataset/oc-transpo-schedules

Morin, Pat, (2011) *Open Data Structures (in Java)*

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. (2009) *Introduction to Algorithms*, 3rd Edition, Massachusetts Institute of Technology

Goodrich, M.T., Tamassia, R (2006) *Data Structures and Algorithms in Java 4th Eddition* John Wiley & Sons, Inc.

"Are Tables General Purpose Structures." *Are Tables General Purpose Structures*. Web. 13 Apr. 2016. http://c2.com/cgi/wiki?AreTablesGeneralPurposeStructures