CS61B - PROJECT REVIEW

Author: Kefan Li

I. NBody(Project 0)

The goal of this project is to give you a crash course in Java. Your goal for this project is to write a program simulating the motion of `N` objects in a plane, accounting for the gravitational forces mutually affecting each object as demonstrated by Sir Isaac Newton's Law of Universal Gravitation. Please refer Proj0 for more details about this project.

My solution is to create `Planet` class which contains the basic rules of planets such as moving and affecting each others through gravitation. Then I created `NBody` class to simulate the movement of planets in the space.

For the specific codes for Proj0, please refer Proj0 Codes

We could practice pushing and pulling files in Git and compiling Java file through terminal.

II. Data Structures - LinkedListDeque & ArrayDeque(Project 1a)

In project 1A, we will build implementations of a "Double Ended Queue" using both lists and arrays. The origin tasks could be found at Proj 1a.

1. LinkedListDeque

The LinkedListDeque is quite easy to finish. The main fundation of this data structure is DLList.

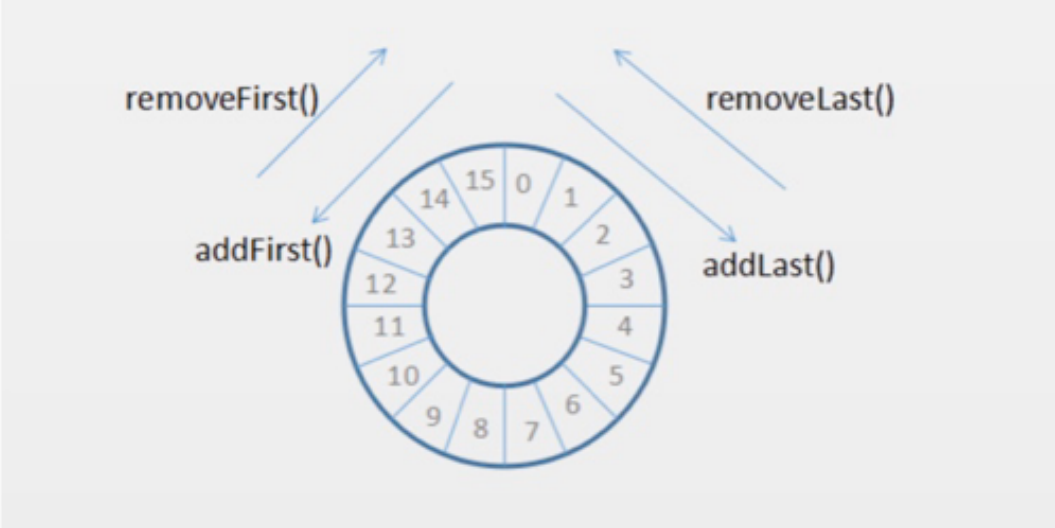
The only thing you should pay more attention is changing previous `IntNode.next` and later `IntNode.prev`.

For all codes, please refer LinkedListDeque

2. ArrayDeque

Since the users can never access the real array, so it is crucial for us to create an array that includes the items in `items` with real order(from first to last) to exclude the circle storage impaction. This step is very tricky and could be the most important part of Proj1a.在每次扩大或者缩小数组的时候，我们必须要得到一个按照真实顺序(从first到last)的数组，从而设定扩大/缩小后的数组的`nextFirst = items.lengh - 1`, `nextLast = size + 1`并且把新的数组的地址赋值给`items`，从而达到放大数组或者缩小数组的目的，并且可以完成`get()`和`print()`

The picture of a basic ArrayDeque is listed below.



Now I will show you guys key codes in that project.

A. Category Discussion(Not Recommended)

There are two key sections. The first one is `sizeIncrease()`

```
private void sizeIncrease() {
        int refactor = 2;
        T[] a = (T[]) new Object[refactor * items.length];

        if (nextFirst > 0) {
            System.arraycopy(items, nextFirst + 1, a, 0, items.length - nextFirst - 1);
            System.arraycopy(items, 0, a, items.length - nextFirst - 1, nextLast);
        }

        if (nextFirst == 0) {
            System.arraycopy(items, 1, a, 0, nextLast - 1);
        }

        if (nextLast == 1) {
            System.arraycopy(items, nextFirst + 1, a, 0, items.length - nextFirst - 1);
            System.arraycopy(items, 0, a, size - 1, 1);
        }
        items = a;
        nextFirst = items.length - 1;
        nextLast = size;
    }
```

The second one is `sizeDecrease()`

```
    private void sizeDecrease() {
        int refactor = 2;
        T[] a = (T[]) new Object[items.length / refactor];

        if (nextFirst < items.length - 1 && nextLast > 0 && items[0] != null) {
```

```
            System.arraycopy(items, nextFirst + 1, a, 0, items.length - nextFirst - 1);
            System.arraycopy(items, 0, a, items.length - nextFirst - 1, nextLast);
        }

        if (nextFirst == items.length - 1) {
            System.arraycopy(items, 0, a, 0, size);
        }

        if (nextLast == 0) {
            System.arraycopy(items, nextFirst + 1, a, 0, size);
        }

        if (nextFirst != items.length - 1 && nextLast <= items.length - 1
                && items[items.length - 1] == null) {
            System.arraycopy(items, nextFirst + 1, a, 0, size);
        }

        items = a;
        nextFirst = items.length - 1;
        nextLast = size;
    }
```

The category discussion is very tricky to consider all the circumstances, it could be time consuming and make you crazy. Therefore, it's better for us to find some invariants in whole program.

For all codes, please refer ArrayDeque

B. Using Invariants and Private Helper Method

There are two key sections. The first one is `itemsInRealOrder(int num)` helper method, which could get an array that contains all item in items with real order.

```
    private T[] itemsInRealOrder(int num) {
        T[] a = (T[]) new Object[num];

        int indexIna = 0;
        int iterateNumber = 0;
        int indexInItems = nextFirst + 1;

        while (iterateNumber < size) {
            indexInItems = indexInItems % items.length;
            a[indexIna] = items[indexInItems];
            indexInItems += 1;
            indexIna += 1;
            iterateNumber += 1;
        }

        return a;
    }
```

The second one is `sizeIncrease() sizeDecrease() get(index) printDeque()`

Their core components are based on `itemsInRealOrder(int num)`.I will only show `sizeIncrease()`

```
    private void sizeIncrease() {
        int refactor = 2;
        T[] a = itemsInRealOrder(refactor * items.length);
        items = a;
        nextFirst = items.length - 1;
        nextLast = size;
    }
```

This solution is tidy and elegant. I personally recommend that. For all codes, please refer ArrayDeque

III. Applying and Testing Data Structures(Project 1B)

The main purpose of this project is to use your deque from project 1A to solve a real world problem. Along the way, you are required to write tests of your application to convince yourself that everything works correctly.

Please refer proj1b official guide for more details.

Task 1: Deque Interface

Create an interface in a new file named `Deque.java` that contains all of the methods that appear in *both* `ArrayDeque` and `LinkedListDeque`. This process creates a super class for `ArrayDeque` and `LinkedListDeque`. Please note that IntelliJ will assume you want a class, so make sure to replace the `class` keyword with `interface`.

After creating this interface and adding all the methods, modify your `LinkedListDeque` and/or `ArrayDeque` so that they implement the `Deque` interface by adding `implements Deque<T>` to the line declaring the existence of the class.

Task 2: wordToDeque

Create a new file called `Palindrome.java`, we will use that file to detect whether a String is Palindrome.

Now we create a method `public Deque<Character> wordToDeque(String word)`. The utility of this method is to switch a `String Word` to be stored in a `Deque`.

The codes of wordToDeque is listed below.

```
    public Deque<Character> wordToDeque(String word) {
        char[] wordInChar = word.toCharArray();
        Deque<Character> D = new ArrayDeque<Character>();
        for (int i = 0; i < word.length(); i++) {
            D.addLast(wordInChar[i]);
        }
        return D;
    }
```

The method is quite starigtforward. The only crucial point that I would like to mention is

```
Deque<Character> D = new ArrayDeque<Character>();
```

the type of variable D is Deque, but it store ArrayDeque object because Deque is the superclass of ArrayDeque.

Task 3: isPalindrome

In this section, you will need to create a method `public boolean isPalindrome(String word)`.

The `isPalindrome` method should return `true` if the given word is a palindrome, and `false` otherwise. A palindrome is defined as a word that is the same whether it is read forwards or backwards. For example “a”, “racecar”, and “noon” are all palindromes. “horse”, “rancor”, and “aaaaab” are not palindromes. *Any word of length 1 or 0 is a palindrome. *Palindrome is case sensitive which means 'A' and 'a' should not be considered as equal.*

You could write a test for `isPalindrome` first then finishing `isPalindrome` to pass the test, which is called **TDD** learned in 3.1. However, you may choose others development operations if you want.

I choose recursion to solve this problem and my code is quite elegant.

```
/**
     * Find whether the given word is palindrome
     *
     * @param word
     * @return
     */
    public boolean isPalindrome(String word) {
        boolean A = isPalindrome(wordToDeque(word));
        return A;
    }

    /**
     * Helper method for isPalindrome
     *
     * @param D
     * @return
     */
    private boolean isPalindrome(Deque<Character> D) {
```

```
        if (D.size() == 1 || D.size() == 0) {
            return true;
        } else {
            if (D.removeFirst() == D.removeLast()) {
                isPalindrome(D);
            } else {
                return false;
            }
        }
        return true;
```

Task 4: Generalized Palindrome and OffByOne

In this task, your ultimate goal is to add a third public method to your `Palindrome` class with the following signature: `public boolean isPalindrome(String word, CharacterComparator cc)`

The method will return `true` if the word is a palindrome according to the character comparison test provided by the `CharacterComparator` passed in as argument `cc`.

In order to finish `public boolean isPalindrome(String word, CharacterComparator cc)`, we need first to create a class called `OffByOne.java`, which should implement `CharacterComparator` such that `equalChars` returns `true` for characters that are different by exactly one.

For instance, this program should all return true.

```
OffByOne obo = new OffByOne();
obo.equalChars('a', 'b');
obo.equalChars('r', 'q');
```

and this program should all return false.

```
obo.equalChars('a', 'e');
obo.equalChars('z', 'a');
obo.equalChars('a', 'a');
```

The key point to write this method is to calculate the difference between two chars, simply compute their difference in java. For example `int diff = 'd' - 'a';` would return `diff` as `-3`.

My solution is:

```
public class OffByOne implements CharacterComparator {
    @Override
    public boolean equalChars(char x, char y) {
        return x - y == 1 || x - y == -1;
    }
}
```

Task 5: OffByN

You will implement a class `OffByN`, which should implement the `CharacterComparator` interface, as well as a single argument constructor which takes an integer.

The callable methods and constructors will be:

- `OffByN(int N)`
- `equalChars(char x, char y)`

The `OffByN` constructor should return an object whose `equalChars` method returns `true` for characters that are off by `N`. For example the call to equal chars below should return `true`, since “a” and “f” are off by 5 letters, but the second call would return `false` since “f” and “h” are off by 4 letters.

```
OffByN offBy5 = new OffByN(5);
offBy5.equalChars('a', 'f');  // true
offBy5.equalChars('f', 'a');  // true
offBy5.equalChars('f', 'h');  // false
```

My solution is:

```
public class OffByN implements CharacterComparator {
    private int N;

    public OffByN(int N) {
        this.N = N;
    }

    @Override
    public boolean equalChars(char x, char y) {
        return x - y == N || x - y == -N;
    }
}
```

Now the proj1b is ended and you could refer proj1b for all files.

Concepts About Abstract Class

So basically the interface, `CharacterComparator` looks like:

```
/**
 * This interface defines a method for determining equality of characters.
 */
public interface CharacterComparator {
    /**
     * Returns true if characters are equal by the rules of the implementing class.
     */
    boolean equalChars(char x, char y);
}
```

This interface does not have a constructor. Someone who has prior C++ experience may notice that, this is what we called *abstract class*. A class that implement an *abstract class* can have its own constructor even if the *abstract class* does not have one.

IV. Autograding(Proj1 Gold)

In this project, you will build a rudimentary autograder for project 1A.

The key point of this part is the `message` parameter to `assertEquals` contains a list of operations that cause the `StudentArrayDeque` to output the wrong answer. The remaining part is starigtforward, just following the official guide. By the way, I use `StdRandom.uniform()` to generate a random number then using this number to decide which method in `Deque<T>` should be called and what argument should be used to fulfill `addFirst()` or `addLast()`.

The string message provided to assertEquals must be a series of method calls, where the last call in the sequence yields an incorrect return value. For example, if adding 5 to the front, then 3 to the front, then removing from the front yields an incorrect value, then the String message passed to assertEquals should be **exactly** the following, with newlines in between each command.

```
addFirst(5)
addFirst(3)
removeFirst()
```

Please refer GitHub for solution MySolution.

V. Bear Maps(Proj 3)

The original instructions and tips could be found at Proj3.

Your job for this project is to implement the back end of a web server. To use your program, a user will open an html file in their web browser that displays a map of the city of Berkeley, and the interface will support scrolling, zooming, and route finding (similar to Google Maps). We've provided all of the "front end" code. Your code will be the "back end" which does all the hard work of figuring out what data to display in the web browser.

At its heart, the project works like this: The user's web browser provides a URL to your Java program, and your Java program will take this URL and generate the appropriate output, which will displayed in the browser.

The `GraphDB` class will read in the Open Street Map dataset and store it as a graph. Each node in the graph will represent a single intersection, and each edge will represent a road. How you store your graph is up to you. This will be the strangest part of the project, since it involves using complex real world libraries to process complex real world data.

The `Router` class will take as input a `GraphDB`, a starting latitude and longitude, and a destination latitude and longitude, and it will produce a list of nodes (i.e. intersections) that you get from the start point to the end point. This part will be similar to the PuzzleSolver assignment, since you'll be implementing A* again, but now with an explicit graph object (that you build in `GraphDB`). As an additional feature, you will be taking that list to generate a sequence of driving instructions that the server will then be able display.

1. Map Rastering

> Rastering is the job of converting information into a pixel-by-pixel image. In the `Rasterer` class you will take a user's desired viewing rectangle and generate an image for them.
>
> The `Rasterer` class will take as input an upper left latitude and longitude, a lower right latitude and longitude, a window width, and a window height. Using these six numbers, it will produce a 2D array of filenames corresponding to the files to be rendered. Historically, this has been the hardest task to fully comprehend and most time consuming part of the project.
>
> The user's desired input will be provided to you as a `Map<String, Double> params`, and the main goal of your rastering code will be to create a `String[][]` that corresponds to the files that should be displayed in response to this query.
>
> The first tough part of this section is to fully understand how to rigorously determine which type of images which has different accuracy to use. We will define the **longitudinal distance per pixel (LonDPP)** as follows: Given a query box or image, the LonDPP of that box or image is: LonDPP=lower right longitude−upper left longitude / width of the image (or box) in pixels.
>
> The images that you return as a `String[][]` when rastering must be those that:
>
> - Include any region of the query box.
> - Have the greatest LonDPP that is less than or equal to the LonDPP of the query box (as zoomed out as possible). If the requested LonDPP is less than what is available in the data files, you should use the lowest LonDPP available instead (i.e. depth 7 images).

The following image could show the different types of relationships in raster.

The implementation of Rastering is:

```
/**
 * This class provides all code necessary to take a query box and produce
 * a query result. The getMapRaster method must return a Map containing all
 * seven of the required fields, otherwise the front end code will probably
 * not draw the output correctly.
 */
```

```
public class Rasterer {

    private static final double LONDPP = 0.00034332275390625;

    private class QueryBox {
        private double ullon, ullat, lrlon, lrlat;
        private double w, h;

        QueryBox(double ullon, double ullat, double lrlon, double lrlat,
                         double w, double h) {
            this.ullon = ullon;
            this.ullat = ullat;
            this.lrlon = lrlon;
            this.lrlat = lrlat;
            this.w = w;
            this.h = h;
        }
    }

    private class Area {
        private double ullon, ullat, lrlon, lrlat;

        Area(double ullon, double ullat, double lrlon, double lrlat) {
            this.ullon = ullon;
            this.ullat = ullat;
            this.lrlon = lrlon;
            this.lrlat = lrlat;
        }
    }

    private String[][] fileNames;
    private double rasterUllon, rasterUllat, rasterLrlon, rasterLrlat;
    private int ulx, uly, lrx, lry;
    private int depth;

    public Rasterer() {
        // YOUR CODE HERE
    }

    private void setDepth(QueryBox qb) {
        double dpp = (qb.lrlon - qb.ullon) / qb.w;
        for (int i = 0; i <= 7; i++) {
            if (calDepthDpp(i) <= dpp) {
                this.depth = i;
                return;
            }
        }
        this.depth = 7;
    }
```

```
    private double calDepthDpp(int dep) {
        return LONDPP / pow2(dep);
    }

    private int pow2(int n) {
        int res = 1;
        while (n-- > 0) {
            res = res * 2;
        }
        return res;
    }

    private void solve(QueryBox qb) {
        setDepth(qb);
        boolean findUl = false;
        boolean findLr = false;
        for (int i = 0; i < pow2(depth); i++) {
            for (int j = 0; j < pow2(depth); j++) {
                Area area = getArea(depth, i, j);
                if (area.lrlon > qb.ullon && area.lrlat < qb.ullat) {
                    this.rasterUllon = area.ullon;
                    this.rasterUllat = area.ullat;
                    this.ulx = i;
                    this.uly = j;
                    findUl = true;
                    break;
                }
            }
            if (findUl) {
                break;
            }
        }

        for (int i = pow2(depth) - 1; i >= 0; i--) {
            for (int j = pow2(depth) - 1; j >= 0; j--) {
                Area area = getArea(depth, i, j);
                if (area.ullon < qb.lrlon && area.ullat > qb.lrlat) {
                    this.rasterLrlon = area.lrlon;
                    this.rasterLrlat = area.lrlat;
                    this.lrx = i;
                    this.lry = j;
                    findLr = true;
                    break;
                }
            }
            if (findLr) {
                break;
            }
```

```
        }
        getFileNames();
    }

    private void getFileNames() {
        fileNames = new String[lry - uly + 1][lrx - ulx + 1];
        for (int i = 0; i <= lry - uly; i++) {
            for (int j = 0; j <= lrx - ulx; j++) {
                fileNames[i][j] = "d" + depth + "_x" + (ulx + j)
                        + "_y" + (uly + i) + ".png";
            }
        }
    }

    private Area getArea(int dep, int x, int y) {
        double lonDelta = (MapServer.ROOT_LRLON - MapServer.ROOT_ULLON) / pow2(dep);
        double latDelta = (MapServer.ROOT_ULLAT - MapServer.ROOT_LRLAT) / pow2(dep);
        double ullon = MapServer.ROOT_ULLON + lonDelta * x;
        double ullat = MapServer.ROOT_ULLAT - latDelta * y;
        double lrlon = ullon + lonDelta;
        double lrlat = ullat - latDelta;
        return new Area(ullon, ullat, lrlon, lrlat);
    }


    /**
     * Takes a user query and finds the grid of images that best matches the query.
These
     * images will be combined into one big image (rastered) by the front end. <br>
     * <p>
     * The grid of images must obey the following properties, where image in the
     * grid is referred to as a "tile".
     * <ul>
     *     <li>The tiles collected must cover the most longitudinal distance per pixel
     *     (LonDPP) possible, while still covering less than or equal to the amount of
     *     longitudinal distance per pixel in the query box for the user viewport size.
</li>
     *     <li>Contains all tiles that intersect the query bounding box that fulfill
the
     *     above condition.</li>
     *     <li>The tiles must be arranged in-order to reconstruct the full image.</li>
     * </ul>
     *
     * @param params Map of the HTTP GET request's query parameters - the query box and
     *               the user viewport width and height.
     * @return A map of results for the front end as specified: <br>
     * "render_grid"   : String[][], the files to display. <br>
     * "raster_ul_lon" : Number, the bounding upper left longitude of the rastered
image. <br>
```

```
     * "raster_ul_lat" : Number, the bounding upper left latitude of the rastered
image. <br>
     * "raster_lr_lon" : Number, the bounding lower right longitude of the rastered
image. <br>
     * "raster_lr_lat" : Number, the bounding lower right latitude of the rastered
image. <br>
     * "depth"         : Number, the depth of the nodes of the rastered image <br>
     * "query_success" : Boolean, whether the query was able to successfully complete;
don't
     * forget to set this to true on success! <br>
     */
    public Map<String, Object> getMapRaster(Map<String, Double> params) {
        System.out.println(params);
        Map<String, Object> results = new HashMap<>();
        double ullat = params.get("ullat");
        double ullon = params.get("ullon");
        double lrlat = params.get("lrlat");
        double lrlon = params.get("lrlon");
        double w = params.get("w");
        double h = params.get("h");
        if (lrlon < MapServer.ROOT_ULLON && lrlat > MapServer.ROOT_ULLAT
                && ullon > MapServer.ROOT_LRLON && ullat < MapServer.ROOT_LRLAT) {
            results.put("query_success", false);
            return results;
        }
        if (ullon > lrlon && ullat < lrlat) {
            results.put("query_success", false);
            return results;
        }
        QueryBox queryBox = new QueryBox(ullon, ullat, lrlon, lrlat, w, h);
        solve(queryBox);
        results.put("render_grid", fileNames);
        results.put("raster_ul_lon", rasterUllon);
        results.put("raster_ul_lat", rasterUllat);
        results.put("raster_lr_lon", rasterLrlon);
        results.put("raster_lr_lat", rasterLrlat);
        results.put("depth", depth);
        results.put("query_success", true);
        System.out.println(results);
        return results;
    }
}
```

Please be noted that we define two private nested class `Area` and `QueryBox` to accelerate to coding process.

2. Routing & Location Data

> In this part of the project, you'll use a real world dataset combined with an industrial strength dataset parser to construct a graph. This is similar to tasks you'll find yourself doing in the real world, where you are given a specific tool and a dataset to use, and you have to figure out how they go together. It'll feel shaky at first, but once you understand what's going on it won't be so bad.
>
> Routing and location data is provided to you in the `berkeley.osm` file. This is a subset of the full planet's routing and location data, pulled from here. The data is presented in the OSM XML file format.
>
> The most difficult part of this section is to fully understand the XML file. XML is a markup language for encoding data in a document. Open up the `berkeley.osm` file for an example of how it looks. Each element looks like an HTML tag, but for the OSM XML format, the content enclosed is (optionally), more elements. Each element has attributes, which give information about that element, and sub-elements, which can give additional information and whose name tell you what kind of information is given.

> The first step of this part of the project is to build a graph representation of the contents of `berkeley.osm`. We have chosen to use a SAX parser, which is an "event-driven online algorithm for parsing XML documents". It works by iterating through the elements of the XML file. At the beginning and end of each element, it calls the `startElement` and `endElement` callback methods with the appropriate parameters.
>
> Your job will be to override the `startElement` and `endElement` methods so that when the SAX parser has completed, you have built a graph. Understanding how the SAX parser works is going to be tricky.

We will self design and develop a Graph data structure that stores the information inside our data sets.

The implementation of `GraphDB` is:

```
/**
 * Graph for storing all of the intersection (vertex) and road (edge) information.
 * Uses your GraphBuildingHandler to convert the XML files into a graph. Your
 * code must include the vertices, adjacent, distance, closest, lat, and lon
 * methods. You'll also need to include instance variables and methods for
 * modifying the graph (e.g. addNode and addEdge).
 *
 * @author Alan Yao, Josh Hug
 */
public class GraphDB {
    /**
     * Your instance variables for storing the graph. You should consider
     * creating helper classes, e.g. Node, Edge, etc.
     */
    static class Node {
        private String id;
```

```
        private double lat, lon;
        private List<String> neighbors;
        private List<Edge> edges;
        private String location;

        Node(String id, double lat, double lon) {
            this.id = id;
            this.lat = lat;
            this.lon = lon;
            neighbors = new ArrayList<>();
            edges = new ArrayList<>();
        }

        public void setLocation(String location) {
            this.location = location;
        }

        public void setNeighbors(String s) {
            neighbors.add(s);
        }

        public void setEdge(Edge e) {
            edges.add(e);
        }
    }

    public static class Edge {
        private String id;
        private String name;
        private String speed;

        public Edge(String id, String name) {
            this.id = id;
            this.name = name;
        }

        public void setSpeed(String speed) {
            this.speed = speed;
        }
    }

    private Map<String, Node> nodes = new HashMap<>();
    private Map<String, Edge> edges = new HashMap<>();

    /**
     * Example constructor shows how to create and start an XML parser.
     * You do not need to modify this constructor, but you're welcome to do so.
     *
     * @param dbPath Path to the XML file to be parsed.
```

```
     */
    public GraphDB(String dbPath) {
        try {
            File inputFile = new File(dbPath);
            FileInputStream inputStream = new FileInputStream(inputFile);
            // GZIPInputStream stream = new GZIPInputStream(inputStream);

            SAXParserFactory factory = SAXParserFactory.newInstance();
            SAXParser saxParser = factory.newSAXParser();
            GraphBuildingHandler gbh = new GraphBuildingHandler(this);
            saxParser.parse(inputStream, gbh);
        } catch (ParserConfigurationException | SAXException | IOException e) {
            e.printStackTrace();
        }
        clean();
    }

    public void addNode(String id, Node n) {
        nodes.put(id, n);
    }

    private void deleteNode(String id) {
        nodes.remove(id);
    }

    public void addEdge(String eid, Edge e) {
        edges.put(eid, e);
    }

    public Node getNode(String id) {
        return nodes.get(id);
    }

    /**
     * Helper to process strings into their "cleaned" form, ignoring punctuation and
capitalization.
     *
     * @param s Input string.
     * @return Cleaned string.
     */
    static String cleanString(String s) {
        return s.replaceAll("[^a-zA-Z ]", "").toLowerCase();
    }

    /**
     * Remove nodes with no connections from the graph.
     * While this does not guarantee that any two nodes in the remaining graph are
connected,
     * we can reasonably assume this since typically roads are connected.
```

```
     */
    private void clean() {
        ArrayList<String> alone = new ArrayList<>();
        for (String s : nodes.keySet()) {
            if (nodes.get(s).neighbors.isEmpty()) {
                alone.add(s);
            }
        }
        for (String s : alone) {
            deleteNode(s);
        }
    }

    /**
     * Returns an iterable of all vertex IDs in the graph.
     *
     * @return An iterable of id's of all vertices in the graph.
     */
    Iterable<Long> vertices() {
        //YOUR CODE HERE, this currently returns only an empty list.
        ArrayList<Long> vertices = new ArrayList();
        for (String s : nodes.keySet()) {
            vertices.add(Long.parseLong(s));
        }
        return vertices;
    }

    /**
     * Returns ids of all vertices adjacent to v.
     *
     * @param v The id of the vertex we are looking adjacent to.
     * @return An iterable of the ids of the neighbors of v.
     */
    Iterable<Long> adjacent(long v) {
        ArrayList<Long> adj = new ArrayList<>();
        for (String s : getNode(Long.toString(v)).neighbors) {
            adj.add(Long.parseLong(s));
        }
        return adj;
    }

    /**
     * Returns the great-circle distance between vertices v and w in miles.
     * Assumes the lon/lat methods are implemented properly.
     * <a href="https://www.movable-type.co.uk/scripts/latlong.html">Source</a>.
     *
     * @param v The id of the first vertex.
     * @param w The id of the second vertex.
     * @return The great-circle distance between the two locations from the graph.
```

```
     */
    double distance(long v, long w) {
        return distance(lon(v), lat(v), lon(w), lat(w));
    }

    static double distance(double lonV, double latV, double lonW, double latW) {
        double phi1 = Math.toRadians(latV);
        double phi2 = Math.toRadians(latW);
        double dphi = Math.toRadians(latW - latV);
        double dlambda = Math.toRadians(lonW - lonV);

        double a = Math.sin(dphi / 2.0) * Math.sin(dphi / 2.0);
        a += Math.cos(phi1) * Math.cos(phi2) * Math.sin(dlambda / 2.0) *
Math.sin(dlambda / 2.0);
        double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
        return 3963 * c;
    }

    /**
     * Returns the initial bearing (angle) between vertices v and w in degrees.
     * The initial bearing is the angle that, if followed in a straight line
     * along a great-circle arc from the starting point, would take you to the
     * end point.
     * Assumes the lon/lat methods are implemented properly.
     * <a href="https://www.movable-type.co.uk/scripts/latlong.html">Source</a>.
     *
     * @param v The id of the first vertex.
     * @param w The id of the second vertex.
     * @return The initial bearing between the vertices.
     */
    double bearing(long v, long w) {
        return bearing(lon(v), lat(v), lon(w), lat(w));
    }

    static double bearing(double lonV, double latV, double lonW, double latW) {
        double phi1 = Math.toRadians(latV);
        double phi2 = Math.toRadians(latW);
        double lambda1 = Math.toRadians(lonV);
        double lambda2 = Math.toRadians(lonW);

        double y = Math.sin(lambda2 - lambda1) * Math.cos(phi2);
        double x = Math.cos(phi1) * Math.sin(phi2);
        x -= Math.sin(phi1) * Math.cos(phi2) * Math.cos(lambda2 - lambda1);
        return Math.toDegrees(Math.atan2(y, x));
    }

    /**
     * Returns the vertex closest to the given longitude and latitude.
     *
```

```
     * @param lon The target longitude.
     * @param lat The target latitude.
     * @return The id of the node in the graph closest to the target.
     */
    long closest(double lon, double lat) {
        String minId = null;
        double minDis = distance(MapServer.ROOT_ULLON, MapServer.ROOT_ULLAT,
                MapServer.ROOT_LRLON, MapServer.ROOT_LRLAT) + 10086;
        for (String id : nodes.keySet()) {
            Node n = getNode(id);
            double nlon = n.lon;
            double nlat = n.lat;
            if (distance(lon, lat, nlon, nlat) < minDis) {
                minDis = distance(lon, lat, nlon, nlat);
                minId = id;
            }
        }
        return Long.parseLong(minId);
    }

    /**
     * Gets the longitude of a vertex.
     *
     * @param v The id of the vertex.
     * @return The longitude of the vertex.
     */
    double lon(long v) {
        return getNode(Long.toString(v)).lon;
    }

    /**
     * Gets the latitude of a vertex.
     *
     * @param v The id of the vertex.
     * @return The latitude of the vertex.
     */
    double lat(long v) {
        return getNode(Long.toString(v)).lat;
    }
}
```

Please be noted that we define a static nested class `Node` to store the information of each point(e.g.store, hospital etc.) and

```
    private Map<String, Node> nodes = new HashMap<>();
    private Map<String, Edge> edges = new HashMap<>();
```

to store the information of GraphDB.

The `GraphBuildingHandler` class will help us to read the OSM XML files so that we could build our `` `GraphDB `` instance.

```
/**
 * Parses OSM XML files using an XML SAX parser. Used to construct the graph of roads
for
 * pathfinding, under some constraints.
 * See OSM documentation on
 * <a href="http://wiki.openstreetmap.org/wiki/Key:highway">the highway tag</a>,
 * <a href="http://wiki.openstreetmap.org/wiki/Way">the way XML element</a>,
 * <a href="http://wiki.openstreetmap.org/wiki/Node">the node XML element</a>,
 * and the java
 * <a href="https://docs.oracle.com/javase/tutorial/jaxp/sax/parsing.html">SAX parser
tutorial</a>.
 * <p>
 * You may find the CSCourseGraphDB and CSCourseGraphDBHandler examples useful.
 * <p>
 * The idea here is that some external library is going to walk through the XML
 * file, and your override method tells Java what to do every time it gets to the next
 * element in the file. This is a very common but strange-when-you-first-see it
pattern.
 * It is similar to the Visitor pattern we discussed for graphs.
 *
 * @author Alan Yao, Maurice Lee
 */
public class GraphBuildingHandler extends DefaultHandler {
    /**
     * Only allow for non-service roads; this prevents going on pedestrian streets as
much as
     * possible. Note that in Berkeley, many of the campus roads are tagged as motor
vehicle
     * roads, but in practice we walk all over them with such impunity that we forget
cars can
     * actually drive on them.
     */
    private static final Set<String> ALLOWED_HIGHWAY_TYPES = new HashSet<>
(Arrays.asList
            ("motorway", "trunk", "primary", "secondary", "tertiary", "unclassified",
                    "residential", "living_street", "motorway_link", "trunk_link",
"primary_link",
                    "secondary_link", "tertiary_link"));
    private String activeState = "";
    private final GraphDB g;
    private GraphDB.Node lastNode;
    private List<String> connections;
    private String wayId;
    private String currentSpeed;
    private String currentName;
    private boolean valid;
```

```
    /**
     * Create a new GraphBuildingHandler.
     *
     * @param g The graph to populate with the XML data.
     */
    public GraphBuildingHandler(GraphDB g) {
        this.g = g;
        this.lastNode = null;
        connections = new ArrayList<>();
        valid = false;
    }

    /**
     * Called at the beginning of an element. Typically, you will want to handle each
element in
     * here, and you may want to track the parent element.
     *
     * @param uri        The Namespace URI, or the empty string if the element has no
     *                   Namespace URI or
     *                   if Namespace processing is not being performed.
     * @param localName  The local name (without prefix), or the empty string if
Namespace
     *                   processing is not being performed.
     * @param qName      The qualified name (with prefix), or the empty string if
qualified names
     *                   are not available. This tells us which element we're looking
at.
     * @param attributes The attributes attached to the element. If there are no
attributes, it
     *                   shall be an empty Attributes object.
     * @throws SAXException Any SAX exception, possibly wrapping another exception.
     * @see Attributes
     */
    @Override
    public void startElement(String uri, String localName, String qName, Attributes
attributes)
            throws SAXException {
        /* Some example code on how you might begin to parse XML files. */
        if (qName.equals("node")) {
            /* We encountered a new <node...> tag. */
            activeState = "node";
//            System.out.println("Node id: " + attributes.getValue("id"));
//            System.out.println("Node lon: " + attributes.getValue("lon"));
//            System.out.println("Node lat: " + attributes.getValue("lat"));

            /* TODO Use the above information to save a "node" to somewhere. */
            /* Hint: A graph-like structure would be nice. */
            String id = attributes.getValue("id");
```

```
            double lat = Double.parseDouble(attributes.getValue("lat"));
            double lon = Double.parseDouble(attributes.getValue("lon"));
            GraphDB.Node n = new GraphDB.Node(id, lat, lon);
            g.addNode(id, n);
            lastNode = n;
        } else if (qName.equals("way")) {
            /* We encountered a new <way...> tag. */
            activeState = "way";
            lastNode = null;
            wayId = attributes.getValue("id");
//            System.out.println("Beginning a way...");
        } else if (activeState.equals("way") && qName.equals("nd")) {
            /* While looking at a way, we found a <nd...> tag. */
            //System.out.println("Id of a node in this way: " +
attributes.getValue("ref"));

            /* TODO Use the above id to make "possible" connections between the nodes
in this way */
            /* Hint1: It would be useful to remember what was the last node in this
way. */
            /* Hint2: Not all ways are valid. So, directly connecting the nodes here
would be
            cumbersome since you might have to remove the connections if you later see
a tag that
            makes this way invalid. Instead, think of keeping a list of possible
connections and
            remember whether this way is valid or not. */
            connections.add(attributes.getValue("ref"));
        } else if (activeState.equals("way") && qName.equals("tag")) {
            /* While looking at a way, we found a <tag...> tag. */
            String k = attributes.getValue("k");
            String v = attributes.getValue("v");
            if (k.equals("maxspeed")) {
                //System.out.println("Max Speed: " + v);
                /* TODO set the max speed of the "current way" here. */
                currentSpeed = v;
            } else if (k.equals("highway")) {
                //System.out.println("Highway type: " + v);
                /* TODO Figure out whether this way and its connections are valid. */
                /* Hint: Setting a "flag" is good enough! */
                if (ALLOWED_HIGHWAY_TYPES.contains(v)) {
                    valid = true;
                } else {
                    valid = false;
                }
            } else if (k.equals("name")) {
                //System.out.println("Way Name: " + v);
                currentName = v;
            }
```

```
//            System.out.println("Tag with k=" + k + ", v=" + v + ".");
        } else if (activeState.equals("node") && qName.equals("tag") &&
attributes.getValue("k")
                .equals("name")) {
            /* While looking at a node, we found a <tag...> with k="name". */
            /* TODO Create a location. */
            /* Hint: Since we found this <tag...> INSIDE a node, we should probably
remember which
            node this tag belongs to. Remember XML is parsed top-to-bottom, so probably
it's the
            last node that you looked at (check the first if-case). */
//            System.out.println("Node's name: " + attributes.getValue("v"));
            lastNode.setLocation(attributes.getValue("v"));
        }
    }

    /**
     * Receive notification of the end of an element. You may want to take specific
terminating
     * actions here, like finalizing vertices or edges found.
     *
     * @param uri       The Namespace URI, or the empty string if the element has no
     *                  Namespace URI or if Namespace processing is not being
performed.
     * @param localName The local name (without prefix), or the empty string if
Namespace
     *                  processing is not being performed.
     * @param qName     The qualified name (with prefix), or the empty string if
qualified names are
     *                  not available.
     * @throws SAXException Any SAX exception, possibly wrapping another exception.
     */
    @Override
    public void endElement(String uri, String localName, String qName) throws
SAXException {
        if (qName.equals("way")) {
            /* We are done looking at a way. (We finished looking at the nodes, speeds,
etc...)*/
            /* Hint1: If you have stored the possible connections for this way, here's
your
            chance to actually connect the nodes together if the way is valid. */
//            System.out.println("Finishing a way...");
            if (valid) {
                GraphDB.Edge e = new GraphDB.Edge(wayId, currentName);
                if (currentSpeed != null) {
                    e.setSpeed(currentSpeed);
                }
                g.addEdge(wayId, e);
                for (String s : connections) {
```

```
                    g.getNode(s).setEdge(e);
                }
                for (int i = 0; i < connections.size() - 1; i++) {
                    String id1 = connections.get(i);
                    String id2 = connections.get(i + 1);
                    GraphDB.Node node1 = g.getNode(id1);
                    GraphDB.Node node2 = g.getNode(id2);
                    node1.setNeighbors(id2);
                    node2.setNeighbors(id1);
                }
            }
            valid = false;
            currentName = null;
            currentSpeed = null;
            connections.clear();
            wayId = null;
        }
    }

}
```

There is no denying that you will feel infrustrate when you are handling it first time. Trying to refer the examples you will find that is so helpful.

3. Route Search

> The `/route` endpoint (kinda like a method in web programming) receives four values for input: the start point's longitude and latitude, and the end point's longitude and latitude. Implement `shortestPath` in your `Router` class so that it satisfies the requirements in the Javadoc.
>
> Your route should be the shortest path that starts from the closest connected node to the start point and ends at the closest connected node to the endpoint. Distance between two nodes is defined as the great-circle distance between their two points (lon1, lat1) and (lon2, lat2). The length of a path is the sum of the distances between the ordered nodes on the path. We do not take into account driving time (speed limits).
>
> Your routing algorithm should take into account the fact that latitude and longitude are in slightly different scales (at our latitude, 1 degree of latitude is ~364,000 feet and 1 degree of longitude is ~288,000 feet), and should also take into account that as you move north or south, these two scales change slightly. We've already created a `distance` method for you that you can use that automatically computes the distance. You should not try to write your own distance method that does something like `sqrt(londiff^2 + latdiff^2)`.

This part is quite straightforward since we will implement A* algorithm for solution.

```
/**
 * This class provides a shortestPath method for finding routes between two points
 * on the map. Start by using Dijkstra's, and if your code isn't fast enough for your
 * satisfaction (or the autograder), upgrade your implementation by switching it to A*.
```

```
 * Your code will probably not be fast enough to pass the autograder unless you use A*.
 * The difference between A* and Dijkstra's is only a couple of lines of code, and
boils
 * down to the priority you use to order your vertices.
 */
public class Router {
    private static class SearchNode implements Comparable<SearchNode> {
        private long id;
        private double distToSt;
        private double distToDest;

        SearchNode(long id, double distToSt, double distToDest) {
            this.id = id;
            this.distToSt = distToSt;
            this.distToDest = distToDest;
        }

        @Override
        public int compareTo(SearchNode s) {
            if (this.distToSt + this.distToDest < s.distToSt + s.distToDest) {
                return -1;
            } else if (this.distToSt + this.distToDest > s.distToSt + s.distToDest) {
                return 1;
            } else {
                return 0;
            }
        }
    }

    /**
     * Return a List of longs representing the shortest path from the node
     * closest to a start location and the node closest to the destination
     * location.
     *
     * @param g       The graph to use.
     * @param stlon   The longitude of the start location.
     * @param stlat   The latitude of the start location.
     * @param destlon The longitude of the destination location.
     * @param destlat The latitude of the destination location.
     * @return A list of node id's in the order visited on the shortest path.
     */
    public static List<Long> shortestPath(GraphDB g, double stlon, double stlat,
                                          double destlon, double destlat) {
        long stId = g.closest(stlon, stlat);
        long destId = g.closest(destlon, destlat);
        Map<Long, Long> edgeTo = new HashMap<>();
        Map<Long, Double> distTo = new HashMap<>();
        distTo.put(stId, 0.0);
        edgeTo.put(stId, stId);
```

```
        PriorityQueue<SearchNode> fringe = new PriorityQueue<>();
        SearchNode stNode = new SearchNode(stId, 0.0, g.distance(stId, destId));
        fringe.add(stNode);
        List<Long> preIds = new ArrayList<>();

        while (!fringe.isEmpty()) {
            SearchNode node = fringe.poll();
            preIds.add(node.id);
            if (node.id == destId) {
                break;
            }
            double distToSt = distTo.get(node.id);
            for (Long id : g.adjacent(node.id)) {
                if (preIds.contains(id)) {
                    continue;
                }
                double between = g.distance(node.id, id);
                if (distTo.get(id) == null || distTo.get(id) > distToSt + between) {
                    distTo.put(id, distToSt + between);
                    edgeTo.put(id, node.id);
                    if (fringe.contains(id)) {
                        fringe.remove(id);
                    }
                    fringe.add(new SearchNode(id, distTo.get(id), g.distance(id,
destId)));
                }
            }
        }

        List<Long> paths = new ArrayList<>();
        paths.add(destId);
        long id = destId;
        while (edgeTo.get(id) != stId) {
            paths.add(edgeTo.get(id));
            id = edgeTo.get(id);
        }
        paths.add(stId);
        Collections.reverse(paths);
        return paths; // FIXME
    }

    /**
     * Create the list of directions corresponding to a route on the graph.
     *
     * @param g     The graph to use.
     * @param route The route to translate into directions. Each element
     *              corresponds to a node from the graph in the route.
     * @return A list of NavigatiionDirection objects corresponding to the input
     * route.
```

```
     */
    public static List<NavigationDirection> routeDirections(GraphDB g, List<Long>
route) {
        for (int i = 0; i < route.size(); i++) {
            GraphDB.Node firstNode = g.getNode(Long.toString(route.get(i)));
            GraphDB.Node secondNode = g.getNode(Long.toString(route.get(i + 1)));

        }
        return null; // FIXME
    }
}
```

We got some ideas from HW4 that we create private staic nested class `SerachNode` and add or remove it from PriorityQueue to fulfill A* algorithm.

After fully understanding the A* algorithm, I know that the former

```
if (fringe.contains(id)) {
    fringe.remove(id);
}
```

statement is wrong since there are no id type exists in priority queue. Also, there are some bugs in the former version, whether the tests in the AutoGrader could pass is depending on luck instead of code itself.

In the newest version, I choose to use the `preNode` to store the paths from start postion the the end position and I delete the `preIds` variable since it may cause some errors. What's more, I delete `edgeTo` which is useless.

```
 /**
     * Return a List of longs representing the shortest path from the node
     * closest to a start location and the node closest to the destination
     * location.
     *
     * @param g       The graph to use.
     * @param stlon   The longitude of the start location.
     * @param stlat   The latitude of the start location.
     * @param destlon The longitude of the destination location.
     * @param destlat The latitude of the destination location.
     * @return A list of node id's in the order visited on the shortest path.
     */
    public static List<Long> shortestPath(GraphDB g, double stlon, double stlat,
                                          double destlon, double destlat) {
        long stId = g.closest(stlon, stlat);
        long destId = g.closest(destlon, destlat);
        SearchNode lastNode = null;
        Map<Long, Double> distTo = new HashMap<>();
        distTo.put(stId, 0.0);
        PriorityQueue<SearchNode> fringe = new PriorityQueue<>();
        SearchNode stNode = new SearchNode(stId, 0.0, g.distance(stId, destId), null);
```

```
        fringe.add(stNode);

        while (!fringe.isEmpty()) {
            SearchNode node = fringe.poll();
            if (node.id == destId) {
                lastNode = node;
                break;
            }
            double distToSt = distTo.get(node.id);
            for (Long id : g.adjacent(node.id)) {
                double between = g.distance(node.id, id);
                if (distTo.get(id) == null || distTo.get(id) > distToSt + between) {
                    distTo.put(id, distToSt + between);
                    fringe.add(new SearchNode(id, distTo.get(id), g.distance(id,
destId), node));
                }
            }
        }

        List<Long> paths = new ArrayList<>();
        SearchNode tmp = lastNode;
        while (tmp != null) {
            paths.add(tmp.id);
            tmp = tmp.preNode;
        }
         Collections.reverse(paths);
        return paths; // FIXME
    }
```