CSC 212 Programming Assignment # 4
Managing Location-stamped Notifications.
Due date: 4/12/2018

Marks: 8 + 2 (bonus). In case of plagiarism -10

Guidelines:	This is an **individual** assignment. The assignment must be submitted to **Web-CAT**

Tired of forgetting to perform your daily tasks, you decide it is time to use technology to help you remember. Instead of using a simple to-do list, however, you want your phone to do the job for you and remind you of your tasks whenever your are near the place where they should be done. For instance, you want to be reminded of submitting the PA whenever your are in college, or buy a new notebook when you visit a specific bookstore. Because you can never predict the exact position where you will be, you want the notification to be sent to you whenever your are in the area surrounding the location as shown in Figure 1.

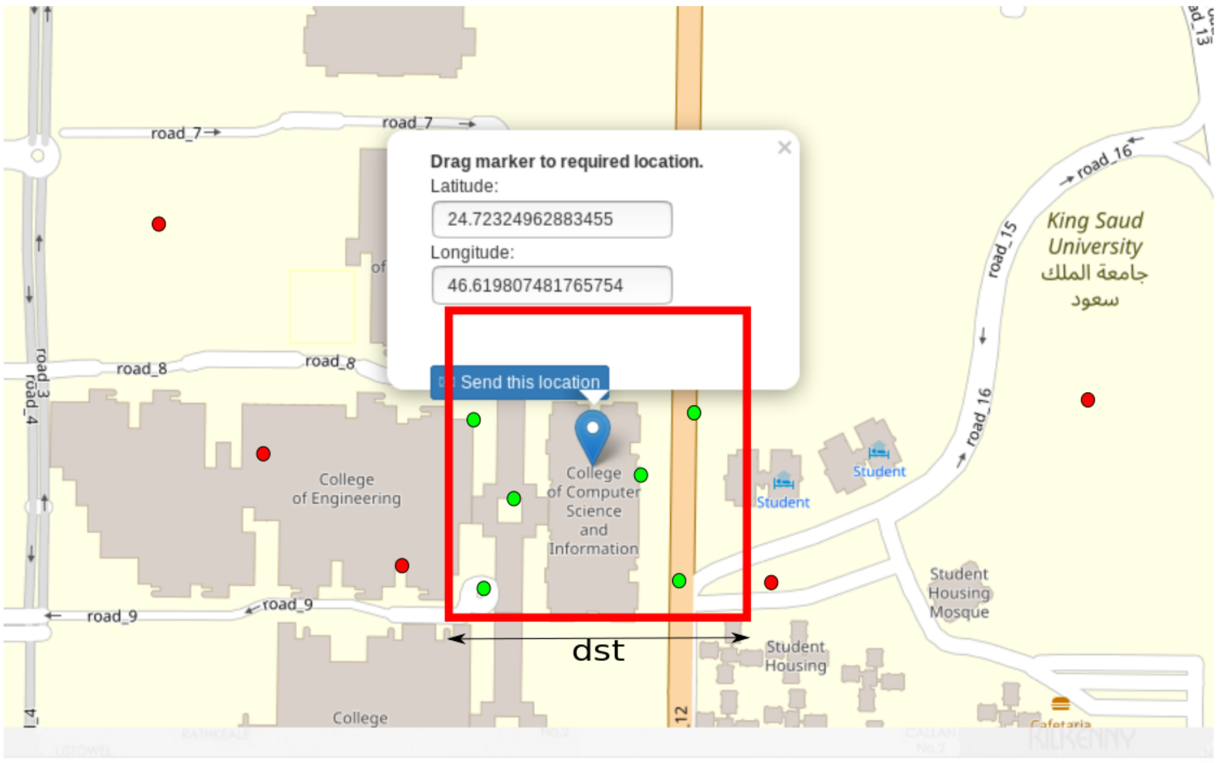

Figure 1: You want the notification to be sent whenever its location is within a square (shown in red) of side dst centered at your location. Note that the square is aligned with the latitude and longitude lines. Notifications inside the square are shown in green, those outsider are shown in red.

Most mobile phones have GPS units that give the phone location to a good precision. The location given is specified by two angles: latitude and longitude as shown in Figure 2.

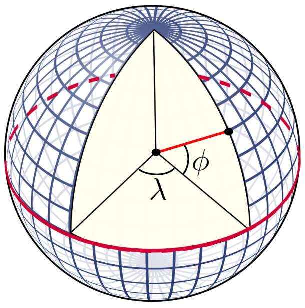

Figure 2: Latitude (ϕ) and longitude (λ) [Credit: Wikipedia].

The class GPS (given to you with this assignment) allows to convert a distance on Earth surface to angles and compute the distance between two points given by latitude and longitude. You may use it if necessary.

Your goal in this assignment is to write the core part of this application, which stores and manages notifications[1]. To Achieve this, we proceed in three steps:

1. Write a binary search tree implementation of a map, which has both key and data generic. The data structure also supports range search (finding all keys in a given interval).
2. Choose a representation and storage scheme for notifications.
3. Write a class `LocNotManager` that manipulates the notifications hence represented.

These three steps are detailed in what follows.

1 Implementing a map

In this step, you have to write a BST implementation of the interface `Map` below. Note that both key and data are generic in this interface. In addition to the operations seen in class, there is an important new method that you have to implement, which is `getRange(K k1, K k2)`. This method performs a **range search**, that is, it returns all elements having a key `k` such that: $k_1 \leq k \leq k_2$.

```
public interface Map<K extends Comparable<K>, T> {
  // Return true if the tree is empty. Must be O(1).
  boolean empty();
  // Return true if the tree is full. Must be O(1).
  boolean full();
  // Removes all elements in the map.
  void clear();
  // Return the data of the current element
  T retrieve();
  // Update the data of current element.
  void update(T e);
  // Search for element with key k and make it the current element if it
     exists. If the element does not exist the current is unchanged and
     false is returned. This method must be O(log(n)) in average.
  boolean find(K key);
```

[1]There will be no use of a mobile phone or GPS unit. However, it is possible to use the code you will write in this assignment as a part of a real application.

```
  // Return the number of keys one needs to compare to in order to find
     key.
  int nbKeyComp(K key);
  // Insert a new element if does not exist and return true. The current
     points to the new element. If the element already exists, current
     does not change and false is returned. This method must be O(log(n))
      in average.
  boolean insert(K key, T data);
  // Remove the element with key k if it exists and return true. If the
     element does not exist false is returned (the position of current is
      unspecified after calling this method). This method must be O(log(n
     )) in average.
  boolean remove(K key);
  // Return all data in the map in increasing order of the keys.
  List<Pair<K, T>> getAll();
  // Return all elements of the map with key k such that k1 <= k <= k2 in
      increasing order of the keys.
  List<Pair<K, T>> getRange(K k1, K k2);
  // Return the number of keys one needs to compare to in order to find
     all keys in the range [k1, k2].
  int nbKeyComp(K k1, K k2);
}
```

2 Representing and storing notifications

Notifications are represented by the class `LocNot` (shown below and given to you with the assignment), which contains the following information:

- **double** `lat, lng`: Latitude and longitude coordinates in degrees.
- **int** `maxNbRepeats`: Some tasks are repetitive (buying milk, for example) whereas other are done once (buying a laptop). The member `maxNbRepeats` specifies the maximum number of times the task should be repeated. Notifications are sent until the tasks is performed `maxNbRepeats` times. If `maxNbRepeats == 0`, the notification is sent every time the phone is near the task location. When the number of repeats is less than the maximum (or `maxNbRepeats == 0`), the notification is said to be **active**, otherwise it is **inactive**.
- **int** `nbRepeats`: Actual number of times the task was repeated. This value is incremented by calling the method `perform`.
- `String text`: Text of the note. The text does not contain any tabulations, return to line or punctuation marks.

```
public class LocNot {
  private double lat, lng; // Latitude and longitude coordinates in
     degrees
  private int maxNbRepeats; // Maximum number of times the associated
     task should be repeated.
  private int nbRepeats; // Actual number of times the task has been
     repeated.
  private String text; // Text of the note
  public double getLng() {
    return lng;
  }
  public double getLat() {
    return lat;
  }
  public int getMaxNbRepeats() {
    return maxNbRepeats;
  }
  public int getNbRepeats() {
    return nbRepeats;
  }
  public String getText() {
    return text;
  }
  public LocNot(String text, double lat, double lng, int maxNbRepeats,
      int nbRepeats) {
    this.text = text;
    this.lat = lat;
    this.lng = lng;
    this.maxNbRepeats = maxNbRepeats;
    this.nbRepeats = nbRepeats;
  }
  // Check if the notification is active.
  public boolean isActive() {
    return maxNbRepeats == 0 || nbRepeats < maxNbRepeats;
  }
  // Perform the task. This has effect only if the notification is active
      , in which case the present method increases the number of repeats
      and returns true. If the notification is inactive, the method has no
       effect and returns false.
  public boolean perform() {
    if (maxNbRepeats == 0 || nbRepeats < maxNbRepeats) {
      nbRepeats++;
      return true;
    } else {
      return false;
    }
  }
}
```

Notifications can be stored in a text file having the following format [2]:

```
24.75365        46.62900        0       0       Buy water
24.72294        46.61978        5       0       Submit PA male
24.72294        46.61838        5       0       Submit HW male
24.72328        46.63640        5       4       Submit PA female
24.72328        46.63540        5       4       Submit HW female
24.72062        46.62248        1       0       Ask for room reparations
24.75213        46.62443        1       0       Buy notebook
24.74203        46.65809        0       0       Buy milk
24.71144        46.62124        1       0       Make doctor appointment
```

There is exactly one notification per line. The fields are in the order specified above and are separated with a tabulation `'\t'`.

To manage notifications, we store them in a **map of maps of** `LocNot` (see Figure 3). The key of the first map is the latitude coordinate, and its data is a map that contains all notifications located at that specific latitude. The key of the second map is longitude and its data is of class `LocNot`.

[2]The coordinates contained in this file are real places: Carrefour Riyadh Park (24.75365, 46.62900), KSU-CCIS (male): (24.72294, 46.61978), KSU-CCIS (female): (24.72328, 46.63640), KSU-Student housing administration (male): (24.72062, 46.62248), Jarir Bookstore Exit 2: (24.75213, 46.62443), Hyper Panda Riyadh Galley: (24.74203, 46.65809), KSU-Hospital: (24.71144, 46.62124).

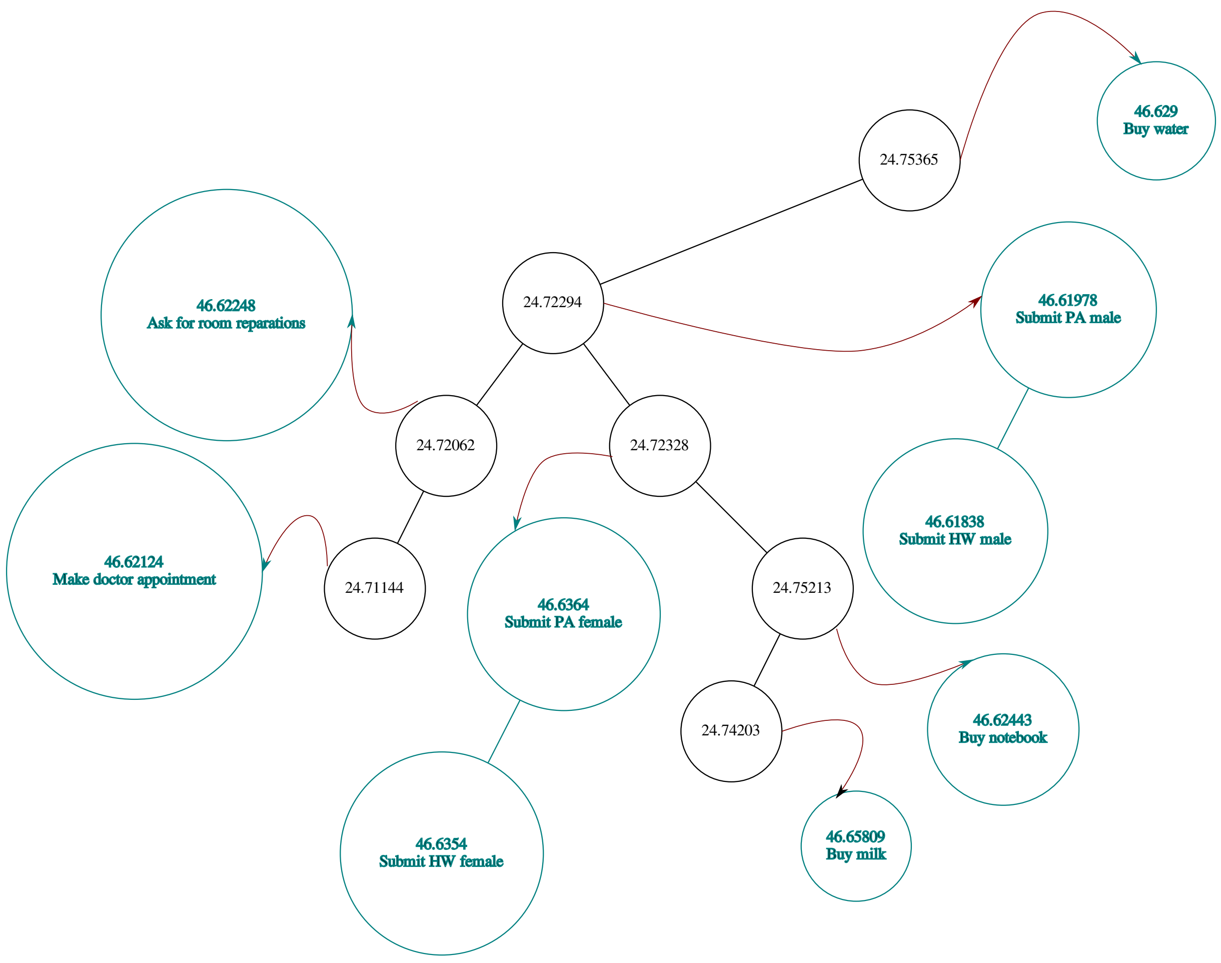

Figure 3: A map of maps used to store notifications. The primary map is indexed by latitude and contains the secondary maps as data, whereas the secondary maps are indexed by longitude and contain the notifications objects as data.

3 Managing notifications

All operations on the notifications map are static methods of the class `LocNotManager`. These include operations of adding and removing notifications, finding notifications that are active at a certain location, and removing notifications that contain a certain word.

The **most important method** of the class `LocNotManager` is `getNotsAt`, which returns all notifications that are located within a square of side `dst` centered at the specified location.

Hint: use range search.

```
public class LocNotManager {
  // Load notifications from file. Assume format is correct. The
     notifications are indexed by latitude then by longitude.
  public static Map<Double, Map<Double, LocNot>> load(String fileName) {
    return null;
  }
  // Save notifications to file.
  public static void save(String fileName, Map<Double, Map<Double, LocNot
     >> nots) {
  }
  // Return all notifications sorted first by latitude then by longitude.
  public static List<LocNot> getAllNots(Map<Double, Map<Double, LocNot>>
     nots) {
    return null;
  }
  // Add a notification. Returns true if insert took place, false
     otherwise.
  public static boolean addNot(Map<Double, Map<Double, LocNot>> nots,
     LocNot not) {
    return false;
  }
  // Delete the notification at (lat, lng). Returns true if delete took
     place, false otherwise.
  public static boolean delNot(Map<Double, Map<Double, LocNot>> nots,
     double lat, double lng) {
    return false;
  }
  // Return the list of notifications within a square of side dst (in
     meters) centered at the position (lat, lng) (it does not matter if
     the notification is active or not). Do not call Map.getAll().
  public static List<LocNot> getNotsAt(Map<Double, Map<Double, LocNot>>
     nots, double lat, double lng, double dst) {
    return null;
  }
  // Return the list of active notifications within a square of side dst
     (in meters) centered at the position (lat, lng).  Do not call Map.
     getAll().
  public static List<LocNot> getActiveNotsAt(Map<Double, Map<Double,
     LocNot>> nots, double lat, double lng, double dst) {
    return null;
  }
  // Perform task of any active notification within a square of side dst
     (in meters) centered at the position (lat, lng) (call method perform
     ).  Do not call Map.getAll().
  public static void perform(Map<Double, Map<Double, LocNot>> nots,
     double lat, double lng, double dst) {
  }
  // Return a map that maps every word to the list of notifications in
     which it appears. The list must have no duplicates.
  public static Map<String, List<LocNot>> index(Map<Double, Map<Double,
     LocNot>> nots) {
    return null;
  }
  // Delete all notifications containing the word w.
  public static void delNots(Map<Double, Map<Double, LocNot>> nots,
     String w) {
  }
}
```

4 Deliverable and rules

You must deliver:

1. Source code submission to Web-CAT. You have to upload the following classed in a zipped file:

   - `BST.java` (must include the class BSTNode)
   - `LocNotManager.java`

   You **must not upload** the interfaces `Map` and `List`, the classes `LinkedList`, `Pair`, `GPS`, `Main`, and the data files.

The submission **deadline** is: **4/12/2018**.

You have to read and follow the following rules:

1. The specification given in the assignment (**class and interface names, and method signatures**) must not be modified. Any change to the specification results in compilation errors and consequently the mark zero.
2. All data structures used in this assignment **must be implemented** by the student. The use of Java collections or any other data structures library is strictly forbidden.
3. This assignment is an individual assignment. Sharing code with other students will result in harsh penalties.
4. Posting the code of the assignment or a link to it on public servers, social platforms or any communication media including but not limited to Facebook, Twitter or WhatsApp will result in disciplinary measures against any involved parties.
5. The submitted software will be evaluated automatically using Web-Cat.
6. All submitted code will be automatically checked for similarity, and if plagiarism is confirmed penalties will apply.
7. You may be selected for discussing your code with an examiner at the discretion of the teaching team. If the examiner concludes plagiarism has taken place, penalties will apply.