CSC 212 Programming Assignment
Efficient Storage of Vehicle Locations for Vehicle Hiring Applications
Deadline:14/12/2021

College of Computer and Information Sciences
King Saud University

Fall 2021

1 Introduction

Vehicle hiring companies such as Uber, Careem, and Jeeny rely on fast localization of available vehicles near potential customers. Such efficient localization requires specialized data structures, and the goal of this project is to implement and use one such data structure.

2 Storing vehicles locations

Vehicles and customers are localized using GPS devices available in mobile phones, and these devices determine the longitude and latitude coordinates as floating-point numbers. To simplify the task, however, we will divide the geographical area under consideration into small cells and use integers to identify each location on the map (see Figure 1).

Figure 1: The city is divided into small cells so that a location is identified by two integer coordinates x and y. Note that a single cell may contain several vehicles and customers.

When a customer request a ride, only vehicles within a certain range of this customer will be contacted (see Figure 2).

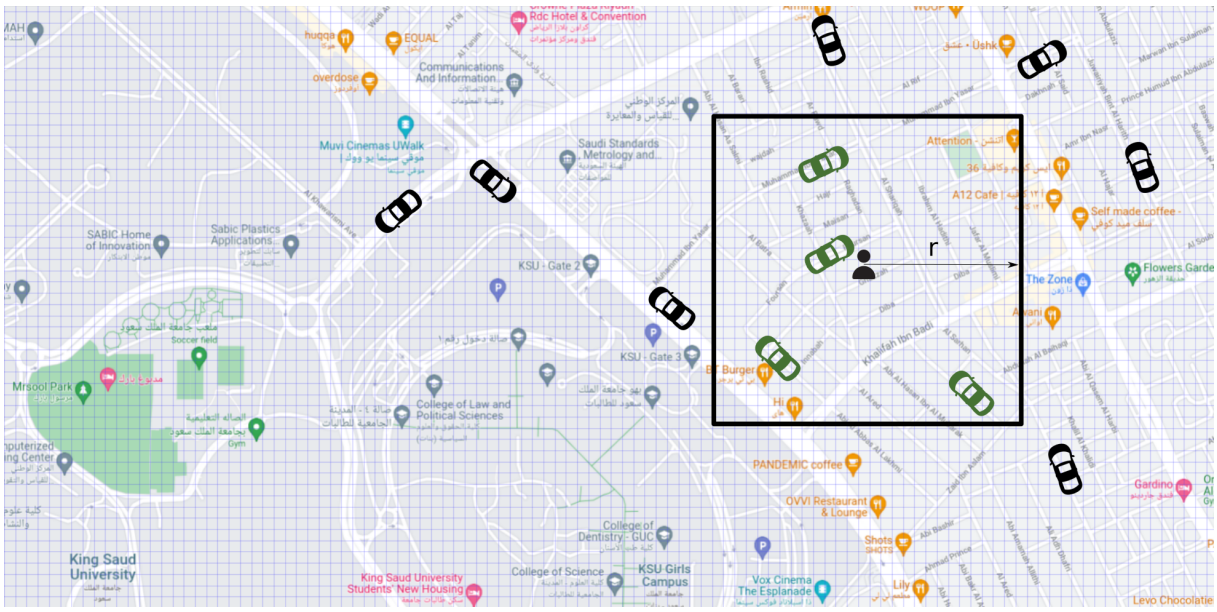

Figure 2: Only vehicles lying within a square of side $2r$ centered at the customer are contacted (shown in green).

To store locations for efficient search, we use a tree structure similar to a BST. In addition to data, each node of this tree contains a location (u, v) and has up to four children (see Figure 3):

- The nodes in the sub-tree rooted at Child 1 (shown in red in Figure 3) contain locations (x, y) satisfying: $x < u$ and $y \leq v$.
- The nodes in the sub-tree rooted at Child 2 (shown in green in Figure 3) contain locations (x, y) satisfying: $x \leq u$ and $y > v$.
- The nodes in the sub-tree rooted at Child 3 (shown in blue in Figure 3) contain locations (x, y) satisfying: $x > u$ and $y \geq v$.
- The nodes in the sub-tree rooted at Child 4 (shown in yellow in Figure 3) contain locations (x, y) satisfying: $x \geq u$ and $y < v$.

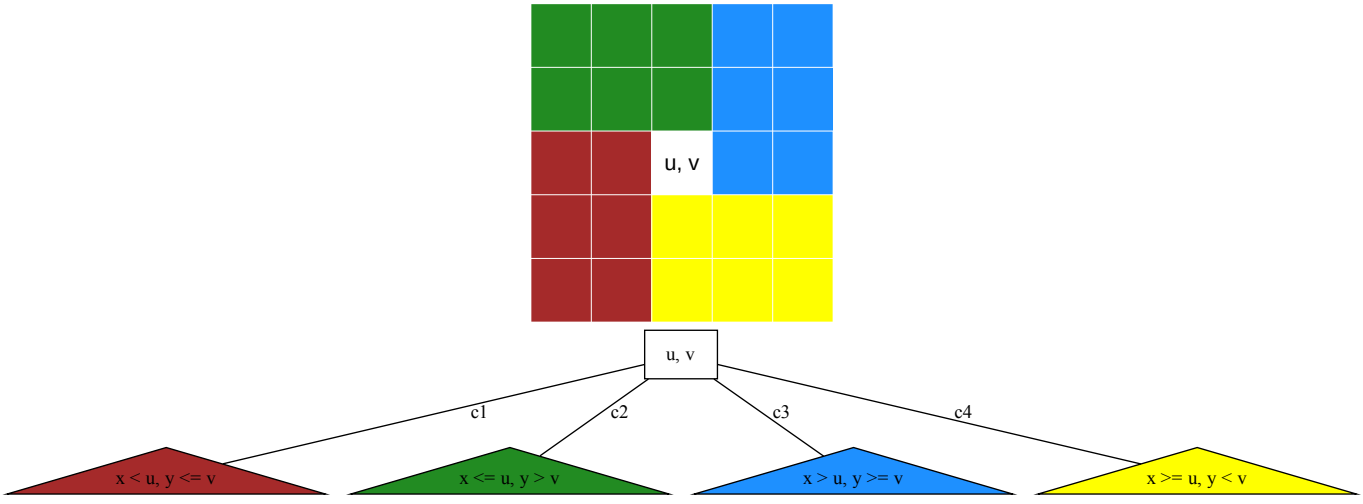

Figure 3: Data structure used to store locations.

Example 1. *If we insert the following data at the specified locations in an empty tree:*
$F \rightarrow (4, 7)$
$V \rightarrow (5, 7)$
$K \rightarrow (6, 1)$
$D \rightarrow (4, 3)$
$O \rightarrow (4, 8)$
$U \rightarrow (8, 4)$

V → (8, 2)
Y → (2, 2)
S → (6, 1)
B → (6, 3)
then, we obtain the tree shown in Figure 4:

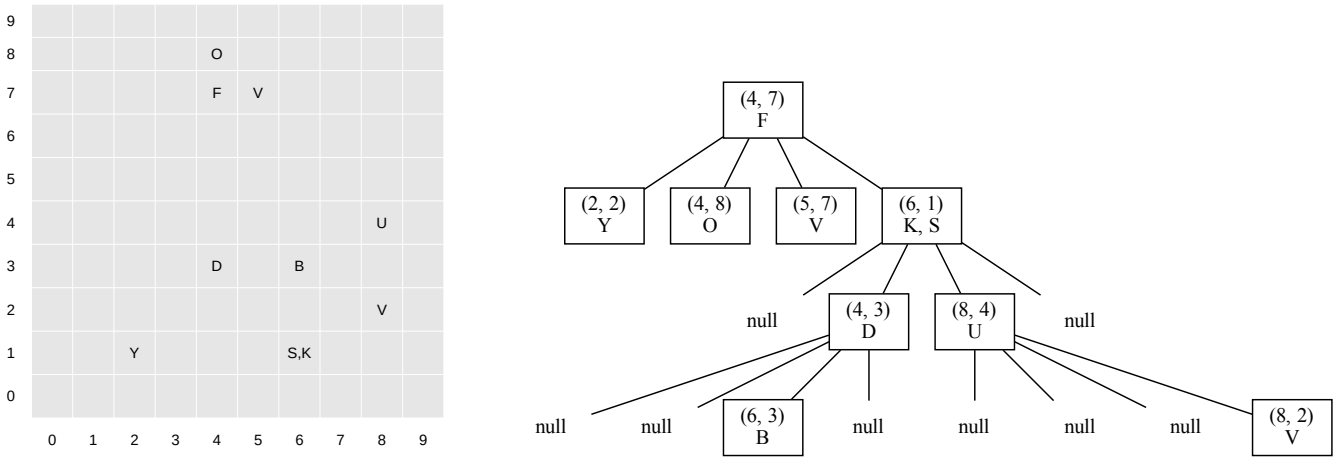

Figure 4: Example tree.

3 Requirements

The requirements for this assignment are incremental, that is, you will get partial points for implementing each class. Note however that:

- Your code must be **compatible with Java 1.7**. In Eclipse, go to: Project > Properties > Java Compiler, and set Compiler compliance level to 1.7.
- If your code contains compilation errors, you will get 0 even if some classes are implemented correctly. Thus you should upload all classes even those you did not implement (classes that are not fully implemented should have default implementations of all methods specified in the interface).
- If your code contains too many infinite loops and exceeds the time limit set for the tests, it will be interrupted by the system and assigned the grade 0.

The code for this assignments consists of the following classes (details are included below):

- Two basic classes `Location` and `Pair` used to represent locations and pairs of elements respectively:
- The class `VehicleHiringManager`, which stores the locations of vehicles and allows to add, remove and move vehicles to a new position. Vehicles are identified by a key of type String. An important functionality of this class is finding all vehicles within a square centered at the customer's location.
- The class `LinkedList` used to store data when needed.
- The class `BST` used to map vehicle IDs to their locations.
- The class `TreeLocator`, which is used to store the vehicles available at every location using the method explained in Section 2.
- The class `TreeLocatorMap` which is used to store the locations of vehicles by combining `BST` and `TreeLocator` to guarantee that vehicle IDs are unique (unlike the class `TreeLocator`, which only guarantees the uniqueness of the locations but not the data).

3.1 Class `Location` (Given)

```
// Represents a 2D location.
public class Location {
  public int x;
  public int y;

  public Location(int x, int y) {
```

```
    this.x = x;
    this.y = y;
  }

  @Override
  public String toString() {
    return "(" + x + "," + y + ")";
  }
}
```

3.2 Class `Pair` (Given)

```
// Stores a pair of elements.
public class Pair<U, V> {
  public U first;
  public V second;

  public Pair(U first, V second) {
    this.first = first;
    this.second = second;
  }

  @Override
  public String toString() {
    return "(" + first + ",␣" + second + ")";
  }
}
```

3.3 Class `LinkedList` (3pt)

Implement the class `LinkedList` that has a no-argument constructor and implements the following interface:

```
public interface List<T> {

  public boolean empty();

  public boolean full();

  public void findFirst();

  public void findNext();

  public boolean last();

  public T retrieve();

  public void update(T e);

  public void insert(T e);

  public void remove();

}
```

3.4 Class `BST` (5pt)

Implement the class `BST` that has a no-argument constructor and implements the following interface:

```
public interface Map<K extends Comparable<K>, T> {

  // Returns true if the tree is empty. Must be O(1).
  boolean empty();

  // Returns true if the tree is full. Must be O(1).
  boolean full();

  // Returns the data of the current element
  T retrieve();

  // Updates the data of current element.
  void update(T e);

  // Makes the element with key k the current element if it exists, and if k does
  // not exist, the current is unchanged. The first element of the returned pair
  // indicates whether k exists, and the second is the number of key comparisons
  // made.
  Pair<Boolean, Integer> find(K key);
```

```
  // Inserts a new element if it does not exist and makes it the current element.
  // If the k already exists, the current does not change. The first element of
  // the returned pair indicates whether k was inserted, and the second is the
  // number of key comparisons made.
  Pair<Boolean, Integer> insert(K key, T data);

  // Removes the element with key k if it exists. The position of current is
  // unspecified after calling this method.The first element of the returned pair
  // indicates whether k was removed, and the second is the number of key
  // comparisons made.
  Pair<Boolean, Integer> remove(K key);

  // Returns all keys of the map as a list sorted in increasing order.
  List<K> getAll();
}
```

3.5 Class `TreeLocator` (7pt)

Implement the class `TreeLocator` that has a no-argument constructor and implements the following interface using the storage scheme explained in Section 2:

```
public interface Locator<T> {

  // Inserts e at location loc and returns the number of comparisons made when
  // searching for loc.
  int add(T e, Location loc);

  // The first element of the returned pair is a list containing all elements
  // located at loc. If loc does not exist or has no elements, the returned list
  // is empty. The second element of the pair is the number of comparisons made
  // when searching for loc.
  Pair<List<T>, Integer> get(Location loc);

  // Removes all occurrences of element e from location loc. The first element
  // of the returned pair is true if e is removed and false if loc does not exist
  // or e does not exist in loc. The second element of the pair is the number of
  // comparisons made when searching for loc.
  Pair<Boolean, Integer> remove(T e, Location loc);

  // Returns all locations and the elements they contain.
  List<Pair<Location, List<T>>> getAll();

  // The first element of the returned pair is a list of all locations and their
  // data if they are located within the rectangle specified by its lower left and
  // upper right corners (inclusive of the boundaries). The second element of the
  // pair is the number of comparisons made.
  Pair<List<Pair<Location, List<T>>>, Integer> inRange(Location lowerLeft, Location upperRight);
}
```

Remark 1. *The number of comparisons in the method* `inRange` *must be minimal.*

3.6 Class `TreeLocatorMap` (5pt)

Implement the class `TreeLocatorMap` that has a no-argument constructor and implements the following interface by using the two classes `BST` and `TreeLocator`:

```
public interface LocatorMap<K extends Comparable<K>> {

  // Returns a map with the location of every key.
  Map<K, Location> getMap();

  // Returns a locator that contains all keys.
  Locator<K> getLocator();

  // Inserts the key k at location loc if it does not exist. The first element of
  // the returned pair indicates whether k was inserted, and the second is the
  // number of key comparisons made.
  Pair<Boolean, Integer> add(K k, Location loc);

  // Moves the key k to location loc if k exists. The first element of
  // the returned pair indicates whether k exists, and the second is the
  // number of key comparisons made.
  Pair<Boolean, Integer> move(K k, Location loc);
```

```
    // The first element of the returned pair contains the location of key k if it
    // exists, null otherwise. The second element is the number of key comparisons
    // required to find k.
    Pair<Location, Integer> getLoc(K k);

    // Removes the element with key k if it exists. .The first element of the
    // returned pair indicates whether k was removed, and the second is the number
    // of key comparisons required to find k.
    Pair<Boolean, Integer> remove(K k);

    // Returns all keys in the map sorted in increasing order.
    List<K> getAll();

    // The first element of the returned pair is a list of all keys located within
    // the rectangle specified by its lower left and upper right corners (inclusive
    // of the boundaries). The second element of the pair is the number of
    // comparisons made.
    Pair<List<K>, Integer> getInRange(Location lowerLeft, Location upperRight);

}
```

3.7 Class `VehicleHiringManager` (3pt)

Implement the class `VehicleHiringManager` by using the class `TreeLocatorMap`:

```
public class VehicleHiringManager {

  public VehicleHiringManager() {
  }

  // Returns the locator map.
  public LocatorMap<String> getLocatorMap() {
    return null;
  }

  // Sets the locator map.
  public void setLocatorMap(LocatorMap<String> locatorMap) {
  }

  // Inserts the vehicle id at location loc if it does not exist and returns true.
  // If id already exists, the method returns false.
  public boolean addVehicle(String id, Location loc) {
    return false;
  }

  // Moves the vehicle id to location loc if id exists and returns true. If id not
  // exist, the method returns false.
  public boolean moveVehicle(String id, Location loc) {
    return false;
  }

  // Removes the vehicle id if it exists and returns true. If id does not exist,
  // the method returns false.
  public boolean removeVehicle(String id) {
    return false;
  }

  // Returns the location of vehicle id if it exists, null otherwise.
  public Location getVehicleLoc(String id) {
    return null;
  }

  // Returns all vehicles located within a square of side 2*r centered at loc
  // (inclusive of the boundaries).
  public List<String> getVehiclesInRange(Location loc, int r) {
    return null;
  }
}
```

Remark 2. *In the implementation of the class* `VehicleHiringManager`, ***do not call*** *the method* `getAll` *of the interface* `LocatorMap`.

4 Deliverables and rules

You must deliver:

1. Source code submission to Web-CAT.

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. This assignment is an individual assignment. Sharing code with other students will result in harsh penalties.
3. 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.
4. The submitted software will be evaluated automatically using Web-Cat.
5. All submitted code will be automatically checked for similarity, and if plagiarism is confirmed penalties will apply.
6. 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.