CMSC 420: Spring 2021

Programming Assignment 2: Wrapped k-d Trees

Handed out: Tue, Apr 6. Due: **Wed, Apr 21, 11pm**. (Submission via Gradescope.)

Overview: In this assignment you will implement a variant of the kd-tree data structure, called a *wrapped kd-tree* (or `WKDTree`) to store a set of points in 2-dimensional space. This data structure will support insertion, deletion, and a number of geometric queries, some of which will be used later in Part 3 of the programming assignment.

The data structure will be templated with the point type, which is any class that implements the Java interface `LabeledPoint2D`, as described in the file `LabeledPoint2D.java` from the provided skeleton code. A labeled point is a 2-dimensional point (`Point2D` from the skeleton code) that supports an additional function `getLabel()`. This returns a string associated with the point.

In our case, the points will be the airports from our earlier projects, and the labels will be the 3-letter airport codes. The associated point (represented as a `Point2D`) can be extracted using the function `getPoint2D()`. The individual coordinates (which are `floats`) can be extracted directly using the functions `getX()` and `getY()`, or `get(i)`, where $i = 0$ for x and $i = 1$ for y.

Your wrapped kd-tree will be templated with one type, which we will call `LPoint` (for "labeled point"). For example, your file `WKDTree` will contain the following public class:

```
public class WKDTree<LPoint extends LabeledPoint2D> { ... }
```

Wrapped kd-Tree: Recall that a kd-tree is a data structure based on a hierarchical decomposition of space, using axis-orthogonal splits. A *wrapped kd-tree* involves two modifications to the standard kd-tree (see Fig. 1).

Extension: As with the previous assignment, our tree will be an extended binary tree involving *internal nodes* and *external nodes*. Each external node stores a single point.

Each internal node stores the splitting information, consisting of a *cutting dimension* and a *cutting value*. The cutting dimension (or `cutDim`) indicates which axis (0 for x and 1 for y) is to be split, and the cutting value (or `cutVal`) indicates where the cut occurs along this axis (see Fig. 1). For example, if the cutting dimension is 0 (for x) and the cutting value is z, then a point $p = (p_x, p_y)$ will be put in the left subtree if $p_x < z$ and in the right subtree if $p_x \geq z$. Note that the cutting value does *not* need to be the coordinate of any point in the tree.[1]

Wrapping: In normal kd-trees, each node is associated with an axis-aligned rectangular cell, which is based on the splits that have been made by this node and its ancestors. In a wrapped kd-tree each internal node explicitly stores a *wrapper*, which is defined to be a

[1]In general, this is a useful feature because we can select splitting lines to optimize query processing. Although we will not take advantage of this fact in this project, this can be important in applications in high dimensional spaces as occurs in machine-learning applications.

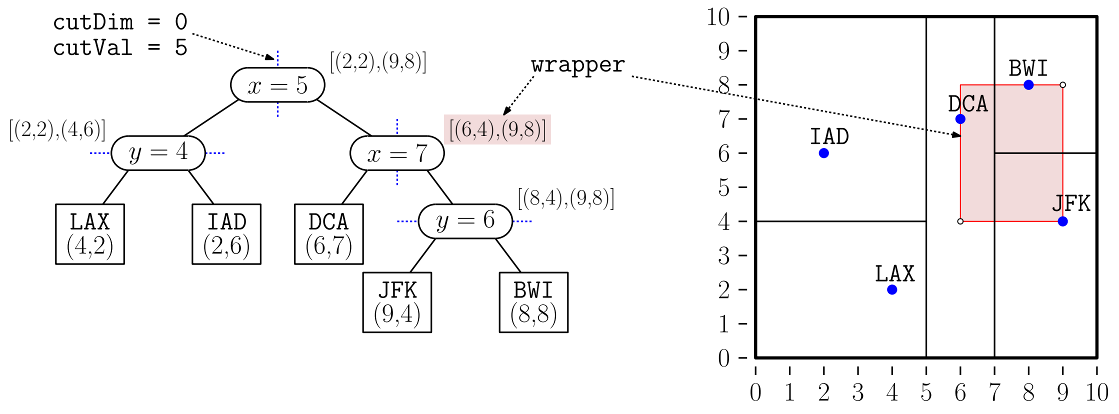

Figure 1: A wrapped kd-tree and the associated spatial subdivision.

minimum axis-aligned bounding box for the points in the subtree associated with this node. (In Fig. 1, the wrapper for the internal node "$x = 7$" is highlighted. A wrapper can be represented as any axis-parallel rectangle, say its lower-left and upper-right corner points.) We will provide a class `Rectangle2D` in the skeleton code, and a node's `wrapper` is of this type.

The use of wrappers modestly increases the storage requirements of the data structure, but query processing can be much faster because a wrapper can be significantly smaller than the associated cell. This means that query processing can do a better job of filtering out subtrees that cannot contribute to the search result. This feature becomes more significant as the dimension of the space increases. As points are inserted and deleted from the tree, the wrappers associated with nodes of the tree need to be updated accordingly.

Wrappers are only computed for internal nodes. (Each external node has an "implicit" wrapper consisting of the trivial rectangle that contains the associated point.) We can define a node's wrapper recursively as the smallest axis-aligned rectangle that contains the wrappers of the node's left and right children. (The class `Rectangle2D` provides a function `union` to perform this operation.)

Specifications on how to implement will be given in the Supplemental Lecture on Wrapped kd-Trees, which will be posted on the class's Projects Page.

Requirements: Your program will implement the following functions for the `WKDTree`. While you can implement the data structure internally however you like (subject to the style and efficiency requirements given below), the following function signatures should not be altered. Recall that `Point2D` is a 2-dimensional point, and an `LPoint` is any object that implements `LabeledPoint2D`.

In addition to the wrapped kd-tree, you will also need to provide a working version of the `AAXTree` from the previous assignment. We will use the `AAXTree` data structure as a *locator*. Whenever we insert a labeled point (e.g., airport "`DCA`" at coordinates $(6, 7)$), we will insert the key-value pair $\langle$`DCA`$, (6, 7)\rangle$ in the `AAXTree`. This way, if we need to determine the coordinates of any airport, we can look it up using its three-letter code. You do not need to do anything other than provide the file, however. Our command handler will automatically insert each airport into both data structures.

`LPoint find(Point2D pt):` Determines whether a point coordinates `pt` occurs within the tree, and if so, it returns the associated `LPoint`. Otherwise, it returns `null`.

`void insert(LPoint pt) throws Exception:` Inserts point `pt` in the tree. It throws an `Exception` with the message `"Insertion of point with duplicate coordinates"` if there is a point in the dictionary with the same coordinates. The insertion process is described in the supplementary lecture. (If there is a tie for the choice of the cutting dimension, x is preferred over y.)

`void delete(Point2D pt) throws Exception:` Deletes the entry whose coordinates match those of `pt`. If there is no such point, it throws an `Exception` with the error message `"Deletion of nonexistent point"`. The deletion process is described in the supplementary lecture.

Note that the above exception will *usually not* appear in your output. The reason is that the default deletion command is given an airport code (e.g., "`delete:DCA`"), and our command handler checks first whether this code exists within your `AAXTree`. If so, it invokes the appropriate deletion command (e.g., "`delete(Point2D(6,7))`"). There is a variant form of the delete command (`delete-point`) which calls your `WKDTree` delete function directly, but we use it sparingly since it causes the `AAXTree` and `WKDTree` to go out of sync with each other.

`ArrayList<String> getPreorderList():` This operation generates a preorder enumeration of the nodes in the tree. This is represented as a Java `ArrayList` of type `String`, with one entry per node. The output for internal and external nodes has the following form (which must be matched exactly):

- Internal nodes: Depending on whether the cutting dimension is x or y, this generates either:

```
"(x=" + cutVal + "):␣" + wrapper.toString()
"(y=" + cutVal + "):␣" + wrapper.toString()
```

  where "␣" is a space character. The function `wrapper.toString()` is defined in `Rectangle2D.java`, which is part of the skeleton code. It outputs the lower-left and upper-right corners of the rectangle.

- External node: Letting `point` denote the labeled point stored in this node, this generates `"[" + point.toString() + "]"`. The function `point.toString()` is defined in `Airport.java`.

For example, here is the result for the tree of Fig. 1.

```
(x=5.0): [(2.0,2.0),(9.0,8.0)]
(y=4.0): [(2.0,2.0),(4.0,6.0)]
[LAX: (4.0,2.0)]
[IAD: (2.0,6.0)]
(x=7.0): [(6.0,4.0),(9.0,8.0)]
[DCA: (6.0,7.0)]
(y=6.0): [(8.0,4.0),(9.0,8.0)]
[JFK: (9.0,4.0)]
[BWI: (8.0,8.0)]
```

Note that our autograder is sensitive to both case and whitespace.

`void clear()`: This removes all the entries of the tree.

`int size()`: Returns the number of points in the tree. For example, for the tree of Fig. 1(a), this would return 5.

`LPoint getMinX()`: This returns a reference to the labeled point that is associated with the smallest x-coordinate in the tree. If two or more points have the same minimum x-coordinate, then the one with the smallest y-coordinate is returned. If the dictionary is empty, this returns `null`.

Analogously, there are functions `getMaxX()`, `getMinY()`, and `getMaxY()`, which perform the analogous operations. For the max versions, if there are ties for the principal coordinate, return the point with the largest "other" coordinate.

For example for the tree shown in Fig. 2(a), the calls `getMinX()` `getMaxX()`, `getMinY()`, and `getMaxY()` would return the labeled points "`SEA`", "`JFK`", "`LAX`", and "`SFO`" (respectively).

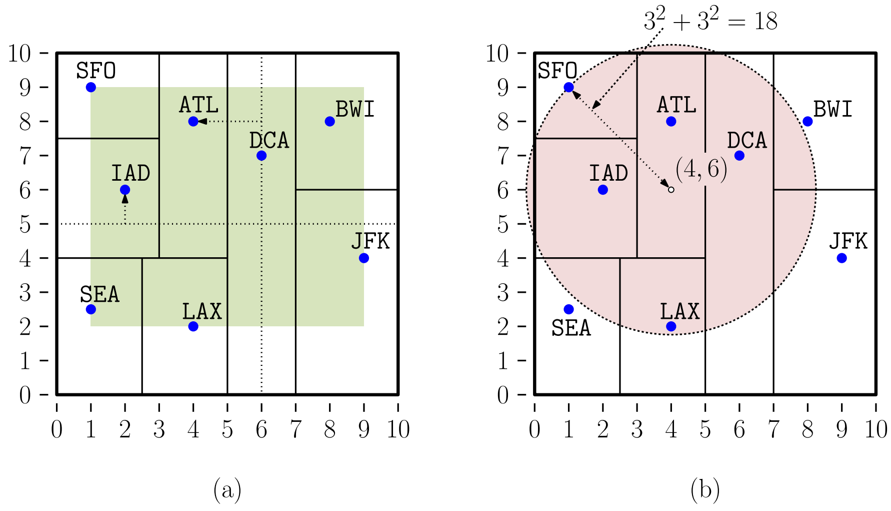

Figure 2: Queries on a wrapped kd-tree.

`LPoint findSmallerX(float x)`: Among all the points whose x-coordinates are *strictly smaller* than x, this returns a reference to the labeled point having the largest x-coordinate. (Note that there need not be a point having the coordinate x.) If the tree is empty or if there is no point whose x-coordinate is smaller than x, this returns `null`. If there are ties for the largest x-coordinate, return the point with the largest y-coordinate.

Analogously, there are functions `findLargerX()`, `findSmallerY()`, and `findLargerY()`. For the larger versions, if there are ties for the point with the smallest principal coordinate, return the point with the smallest "other" coordinate.

For example, for the tree of Fig. 2(a), `findSmallerX(6)` would return the labeled point "`ATL`", and `findLargerY(5)` would return the labeled point "`IAD`".

`ArrayList<LPoint> circularRange(Point2D center, float sqRadius)`: This function is given a circular disk, expressed as a center point `center` and a squared radius of `sqRadius`. (So, to represent a disk of radius 5, we would set `sqRadius` to $5^2 = 25$.) This function returns an array-list containing all the points whose squared distance from the center point is at most `sqRadius`. (Thus if a point lies on the boundary of the disk, it will be included.) If there are no points in the disk, it should return an empty array-list (not `null`). The order of elements in the list does not matter (because we will sort it before outputting), but there should be no duplicates in the list.

To assist you, the `Point2D` and `Rectangle2D` classes both provide a utility function `distanceSq(Point2D pt)`, which computes the squared distance from the current object to point `pt`.

For example, in the tree of Fig. 2(b), `circularRange(Point2D(4,6), 18)` would return an `ArrayList` with the labeled points "`ATL`", "`DCA`", "`IAD`", "`LAX`", and "`SFO`" (in any order).

Why squared radius? The advantage of using squared distances over standard Euclidean distances is that we can avoid invoking the square-root function required by the standard distance (from the Pythagorean Theorem). Since our input points have integer coordinates, squared distances can be computed exactly as integers, which avoids floating-point round-off errors due to limited precision. (For example, the squared distance between $(4,6)$ and $(1,9)$ in Fig. 2(b) is 18, whereas the Euclidean distance is $4.242640687119285\ldots$.)

Skeleton Code: As usual, we will provide skeleton code on the class Projects Page. You should replace the `AAXTree.java` file with your own, and you should add the implementation of the above functions to `WKDTree.java`. You should not modify any of the other files, but you can add new files of your own. For example, if you wanted to add additional functions to any of the classes, such as `Point2D` or `Rectangle2D`, it would be preferable to create an entirely new class (e.g., `MyRect2D`), which you will upload with your submission.

You must use the package "`cmsc420_s21`" for all your source files. (This is required for the autgrader to work.) As usual, we will provide a driver program (`Tester.java` and `CommandHandler.java`) that will input a set of commands. Here is a portion of the class's public interface (and of course, you will add all the private data and helper functions). You should not modify the signature of the public functions, but you are free to set up the internal structure however you like.

```
package cmsc420_s21;

public class WKDTree<LPoint extends LabeledPoint2D> {

    public WKDTree() { ... } // you fill these in
    public LPoint find(Point2D pt) { ... }
    public void insert(LPoint pt) throws Exception { ... }
    public void delete(Point2D pt) throws Exception { ... }
    // ... and so on
}
```

Efficiency requirements: Unlike the AA-Tree, there are no worse-case guarantees on the running times of the above functions. Nonetheless, you should make a reasonable effort to implement your functions in an efficient manner. (See the supplemental lecture notes for suggestions.) In particular, if it can be inferred (e.g., from the wrapper) that a node's subtree cannot possibly contribute to the query result, then the processing should immediately return, without visiting its children. Also, if there is one subtree that is obviously better to visit first than the other, your code should take advantage of this. Up to 10% of the final score will be based on this manual inspection of your code.

Testing/Grading: As before, we will be using Gradescope's autograder and JUnit for testing and grading your submissions. Because you will be submitting multiple files for this part, you should produce a zip file with the two principal files (`AAXTree.java` and `WKDTree.java`). You may include other files, but note that the files given in the skeleton code (e.g., `Point2D.java`, `Rectangle2D.java`, and so on) will be overwritten by the autograder. So, there is nothing to be gained by modifying these files.

As always, we will provide some sample test data and expected results along with the skeleton code. Note that some portion (up to 20%) of the final grade will be based on hidden tests.