CMSC 420: Fall 2022

Programming Assignment 2: Extended kd-Trees

Handed out: Thu, Oct 27. Due: **Thu, Nov 10, 11:59pm**. (Submission via Gradescope.)

Overview: In this assignment we will implement a variant of the kd-tree data structure, which will call an *extended kd-tree* (or `XkdTree`) to store a set of points in 2-dimensional space. This data structure involves a number of practical extensions over the standard kd-tree covered in class. First, the tree will be extended, meaning that we distinguish between *internal nodes*, whose only function is to subdivide space, and *external nodes* (or *leaves*), where the points are actually stored. Second, we allow each external node to a small number of points, depending on a parameter called the *bucket size*.

Points and Rectangles: The objects to be stored in the trees are 2-dimensional points. To save you some effort, as part of the skeleton code, we will provide you with a class for 2-dimensional points, called `Point2D.java`. This class will provide you with some utility functions, such as accessing individual coordinates and computing distances. We will also provide you with a class for storing axis-aligned rectangles, called `Rectangle2D.java`. This also provides a number of useful functions, such as testing whether two rectangles are disjoint or whether one contains the other.

Each point to be stored in the data structure will have an associated value, called its *label*. In our case, the label is just a Java `String`. The resulting object is called a *labeled point*. Rather than impose a particular class structure, a labeled point is any class that supports a Java interface, which we call `LabeledPoint2D.java`. Here is the interface

```
public interface LabeledPoint2D {
    public double getX();          // get point's x-coordinate
    public double getY();          // get point's y-coordinate
    public double get(int i);      // get point's i-th coordinate (0=x, 1=y)
    public Point2D getPoint2D();   // get the point itself (without the label)
    public String getLabel();      // get the label (without the point)
}
```

As the implementer of the data structure, you do not need to worry about the actual objects being stored, as long as you access the object through the interface functions. Again, all of these will be provided to you in our skeleton code.

Extended kd-Tree: The `XkdTree` data structure will be templated with the labeled-point type. Since this is any object that implements the `LabeledPoint2D` interface, we will call it `LPoint`. Your class declaration (which we will provide you) looks like this:

```
public class XkdTree<LPoint extends LabeledPoint2D> { /* fill this in */ }
```

An *extended kd-tree* involves two modifications to the standard kd-tree as discussed in lecture (see Fig. 1). First, the tree is extended, meaning that there are two types of nodes, *internal nodes* and *external nodes*. Second, rather than holding a single point, each external node

stores a small set of points, called a *bucket*. The actual number ranges from 0 up to some maximum number, called the *bucket size*. The bucket size is specified by the user when the data structure is first constructed. You should think of it as a small positive integer, ranging from perhaps 1 up to 10.

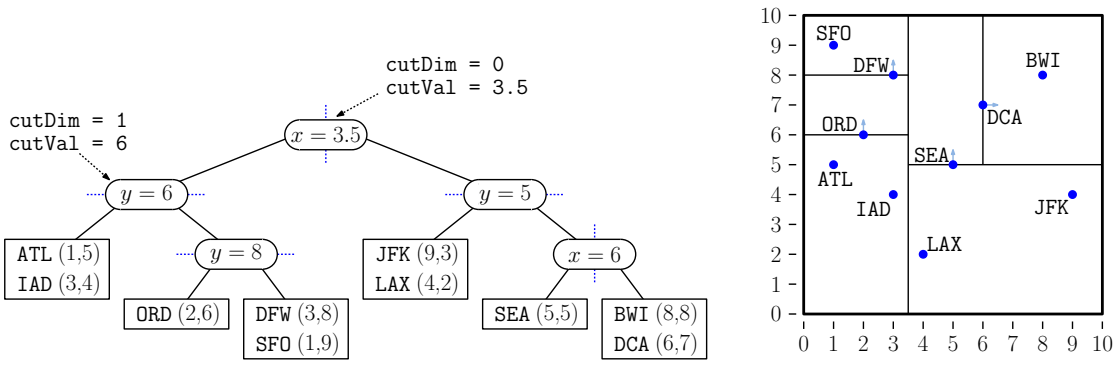

Figure 1: An example of an extended kd-tree with bucket size 2 and bounding box $[0, 10] \times [0, 10]$.

Internal Nodes: 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 used to determine which subtree the points belong to. 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 5, then a point $p = (p_x, p_y)$ will be put in the left subtree if $p_x < 5$ and in the right subtree if $p_x \geq 5$. Note that the cutting value does *not* need to be the coordinate of any point in the tree.

External Nodes: Each external node stores a list (e.g., a Java `ArrayList`) of the labeled points that lie within this node. The size of this list can vary between zero and the data structure's bucket size.

Note that there are no `null` pointers. Every internal node has two non-null children, which may be external nodes, and external nodes have no children by definition. What if the tree is empty? An empty tree is represented setting the `root` to point to a single external node whose bucket is empty.

Requirements: Your program will implement the following functions for the `XkdTree`. 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`.

`XkdTree(int bucketSize, Rectangle2D bbox)`: This constructs a new (empty) `XkdTree` with the given bucket size and bounding box.

`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, this would return 10. For efficiency, you should store this value in your structure, rather than compute it by traversing the tree.

`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`. (Note that the query is a standard 2-dimensional point, while the output is a labeled point.)

Unlike standard kd-trees, if `pt` lies directly on the cutting line (that is, `pt[cutDim] == cutVal`), this point might lie in either the left subtree or the right subtree. (In the standard kd-tree presented in class, it always lies in the right subtree.) Whenever you visit a node where this is the case, you will need to check *both* subtrees of an internal node. If it is found in either, then return the associated labeled point. If it is found in neither, return `null`.

`void insert(LPoint pt) throws Exception`: Inserts a single labeled point `pt` in the tree. You may assume that there are no duplicate points, but there may be duplicate coordinate values. Inserting a single point is functionally equivalent to performing a bulk insertion (see below) with a point set of size one.

`void bulkInsert(ArrayList<LPoint> pts) throws Exception`: Inserts a set of one or more labeled points, `pts`, in the tree. You may assume there are no duplicate points. If any point lies outside the bounding box, you should throw an `Exception` with the error message `"Attempt to insert a point outside bounding box"`.

The insertion is performed as follows. First, we determine the external nodes into which each of the points falls. (If the point lies on the splitting line of an internal node, insert it into the right subtree.) Next, consider each of the external node that has received at least one new point. We merge all the new points together with the existing points of this node's bucket. (The built-in Java function `addAll` is useful for doing this.) If the number of total number of points is not greater than the bucket size, we are done. Otherwise, we need to split the bucket, as described next.

Splitting (Big View): The splitting process works as follows. First, we compute the smallest bounding axis-aligned rectangle containing all the points. (The function `expand`, which is provided in `Rectangle2D` is useful for doing this.) If this rectangle's width is greater than or equal to its height, the cutting dimension is set to 0 (that is, x), and otherwise it is set to 1 (that is, y).

Next, compute the median coordinate along the cutting dimension and partition the set about this median element into two sets, call them `L` and `R`. We create a new internal node having this cutting dimension and set its cutting value to be the median coordinate. The elements of `L` are then recursively inserted into its left subtree and the elements of `R` are recursively inserted into the right subtree. Whenever the number of remaining points is less than or equal to the bucket size, we create a new external node and store the points there.

Splitting (Details): Care must be taken with the splitting process to avoid issues that arise when points have duplicate coordinates along the cutting dimension. If multiple points share the same coordinate value as the median, our usual convention would put all of these points into the right subtree. But this may result in a highly imbalanced tree (and even infinite looping if you are not careful). In order to ensure that the tree is balanced, we want every partition to be as balanced as possible.

Here is how to implement this. (For the sake of uniformity in grading, it is a *requirement* that you partition your points in this way.) Once the cutting dimension

has been determined, sort the points according to cutting dimension with ties broken by other coordinate. Suppose that there are n points, and let `points[n]` denote the sorted sequence of points (this is probably an `ArrayList`). Let $m = \lfloor n/2 \rfloor$. If n is odd, the cutting value is taken to be the unique median element, `points[m]`. If n is even, the cutting value is taken to be the mean of the lower and upper medians, that is, (`points[m - 1]` + `points[m]`)/2.

Next, define the left list `L` to be the sublist consisting of the first m elements. Let's use the notation `points[i,j]` to denote the subarray from `points[i]` to `points[j-1]`. We have `L = points[0,m]`. Define the right list `R` to be the sublist consisting of the last $n - m$ elements, that is, `R = points[m,n]`. (Note that Java provides a handy function, called `sublist`, which can be used to partition the list.)

Sorting Labeled Points: You might be wondering, "Do I need to write my own sorting function?" (Answer: No, you should *never* write your own sorting algorithm.) Java provides a flexible method sorting based on various criteria. There is a built-in sorting function, `Collections.sort`. To instruct it how to sort, you provide it a comparison function. (In the case of partitioning for insertion, this is either lexicographically by (x, y) or lexicographically by (y, x), depending on the cutting dimension).

In Java, this is done defining a class that implements the `Comparator` interface. Such a class defines a single function, called `compare`, which compares two objects of the desired type, and returns a negative, zero, or positive result depending on which argument is larger. In our case, the objects are labeled points, `LPoint`. Here is a brief example on how you might set this up. (A Google search for "`Java sorting with a Comparator`" will reveal more examples.)

```
private class ByXThenY implements Comparator<LPoint> {
    public int compare(LPoint pt1, LPoint pt2) {
        /* compare pt1 and pt2 lexicographically by x then y */
    }
}
private class ByYThenX implements Comparator<LPoint> {
    public int compare(LPoint pt1, LPoint pt2) {
        /* compare pt1 and pt2 lexicographically by y then x */
    }
}
```

`void delete(Point2D pt) throws Exception:` (There is no deletion function required for this assignment.)

`ArrayList<String> list():` This operation generates a right-to-left preorder traversal of the nodes in the tree. (Recall that `right-to-left` means that we visit the right subtree before the left.) There is one entry in the list for each node of the tree. The output is described below:

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

`"(x=" + cutVal + ")"` or `"(y=" + cutVal + ")"`

External nodes: This generates list of points in this bucket surrounding by square brackets, `"[" + ... + "]"`. The "`...`" is a list of the labeled points in the bucket,

each enclosed in curly braces `"{" + ... + "}"`. The points should be sorted by their string labels (another `Comparator`!). To output each labeled point, you can invoke the function `point.toString()`, which is defined in `Airport.java`.

For example, here is the result for the tree of Fig. 1. (Take note of spaces.)

```
(x=3.5)
(y=5.0)
(x=6.0)
[ {BWI: (8.0,8.0)} {DCA: (6.0,7.0)} ]
[ {SEA: (5.0,5.0)} ]
[ {JFK: (9.0,3.0)} {LAX: (4.0,2.0)} ]
(y=6.0)
(y=8.0)
[ {DFW: (3.0,8.0)} {SFO: (1.0,9.0)} ]
[ {ORD: (2.0,6.0)} ]
[ {ATL: (1.0,5.0)} {IAD: (3.0,4.0)} ]
```

Note that our autograder is sensitive to both case and whitespace. Our command-handler program will convert this into a formatted tree structure:

```
Tree structure:
  | | | [ {BWI: (8.0,8.0)} {DCA: (6.0,7.0)} ]
  | | (x=6.0)
  | | | [ {SEA: (5.0,5.0)} ]
  | (y=5.0)
  | | [ {JFK: (9.0,3.0)} {LAX: (4.0,2.0)} ]
  (x=3.5)
  | | | [ {DFW: (3.0,8.0)} {SFO: (1.0,9.0)} ]
  | | (y=8.0)
  | | | [ {ORD: (2.0,6.0)} ]
  | (y=6.0)
  | | [ {ATL: (1.0,5.0)} {IAD: (3.0,4.0)} ]
```

`LPoint nearestNeighbor(Point2D center)`: This function is given a query point (regular point, not a labeled point), and it computes the closest point. If the tree is empty, it returns `null`. Otherwise, it returns a reference to the closest `LPoint` in the kd-tree. For example, in Fig. 2, the nearest neighbor for `center` $= (6, 4)$ would return a reference to the point labeled `LAX`.

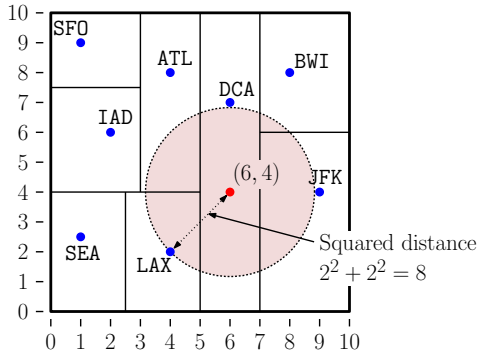

Figure 2: Nearest-neighbor query using squared distances.

Computing distances involves computing square roots, which is both unnecessary and introduces floating-point errors. Instead, you should compute squared Euclidean distances. (This does not change the identity of the closest point). 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`.

You should adapt the algorithm given in class to the context of extended trees. In particular, when visiting an internal node, you should first visit the subtree that is closer to the query point. Second, you should avoid visiting nodes that you can infer cannot contain the nearest neighbor. To do this, keep track of the closest point seen so far, and if node's cell is farther away from the query point than this, then you should avoid visiting the node. (Or if you visit it, you should discover this and return immediately.)

If there are multiple nearest neighbors of the query point, you may return any one of them. (We will engineer the test data so this never happens, so there should be no variance with our expected outputs.)

Java Node Structure: Because we have two types of nodes, we will need a more sophisticated node structure. We use good object-oriented principles by defining a parent node type, called `Node`. This is abstract, which means that we never create such a node. The actual nodes are called `InternalNode` and `ExternalNode`. (You may name them whatever you like.) Internal nodes store splitting information (cutting dimension and cutting value) and subtrees. External nodes stores the bucket, which might be stored as a Java `ArrayList`.

An example of how this might look is shown below. Since this is all internal, you are free to implement however you like. But, I would strongly encourage you to use object inheritance, since this is just good programming style.

```
public class XkdTree<LPoint extends LabeledPoint2D> {

    private abstract class Node { // generic node (purely abstract)
        abstract LPoint find(Point2D pt); // find helper - abstract
        // ... other helper functions omitted
    }

    private class InternalNode extends Node {
        int         cutDim;      // the cutting dimension (0 = x, 1 = y)
        double      cutVal;      // the cutting value
        Node        left, right; // children

        LPoint find(Point2D q) { /* find helper for internal nodes */ }
        // ... other helper functions omitted
    }

    private class ExternalNode extends Node {
        ArrayList<LPoint> points; // the bucket

        LPoint find(Point2D pt) { /* find helper for external nodes */ }
        // ... other helper functions omitted
    }
```

```
    // ... the rest of the class
}
```

You might observe that we don't need to store the node types. This is handled automatically by Java's inheritance mechanisms. For example, suppose that we want to invoke the `find` helper function on the root. Each node type defines its own helper. We then invoke `root.find(pt)`. If `root` is an internal node, this invokes the internal-node find helper, and otherwise it invokes the external-node find helper.

Skeleton Code: As usual, we will provide skeleton code on the class Projects Page. You will need to fill in the implementation of the `XkdTree.java`. We also provide some utility classes (e.g., `Point2D` and `Rectangle2D`). 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.

As with the previous assignment, the package "`cmsc420_f22`" is required for all your source files. As usual, we will provide a driver programs (tester and command-handler) for processing input and output. You should not modify the signature of the public functions, but you are free to set up the internal structure however you like.

Efficiency requirements: (10% of the final grade) For the sake of partitioning, your bulk insert function is allowed to sort the points being inserted at each node of the tree. (You are encouraged to think about more efficient methods. It is obviously inefficient to sort the points by x at one node, then resort them by y at its child, and then resort them again by x at its grandchild, but this is the sort of thing that the above construction algorithm will do.) As mentioned above, you should use Java's `Collection.sort` (or some other built-in) sorting function, as opposed to implementing your own bubble sort algorithm (ugh!) or the like.

You should have a variable that tracks the number of points in the tree in order to answer the `size` operation efficiently.

The nearest-neighbor algorithm should follow the structure given in the lecture notes. Minor variations are allowed, but your implementation should have the two major features as the one given in class. First, it should prioritize visiting on the side of the tree that is closer to the query point, and second, it should not visit subtrees that cannot possibly contain the nearest neighbor.

Style requirements: (5% of the final grade) Good style is not a major component of the grade, but you should demonstrate some effort here. Part of the grade is based on clean, elegant coding. There is no hard rules here, and we will not be picky. If we deduct points, it will because you used an excessively complicated structure to implement a relatively simple computation.

The other part is based on commenting. You should have a comment at the top of each file you submit. This identifies you as the author of the program and provides a short description of what the program does. For each function (other than the most trivial), you should also include a comment that describes what the function does, what its parameters are, and

what it returns. (If you would like to see an example, check out our *canonical solution* to Programming Assignment 1, on the class Project Page.)

Testing/Grading: As always, we will provide some sample test data and expected results along with the skeleton code.

As before, we will be using Gradescope's autograder for grading your submissions. You only need to submit your `XkdTree.java` file. If you created any additional files for utility objects, you will need to upload those as well.

Challenge Problem: (Remember, challenge problems count for extra-credit points, which are only taken into consideration after the final cutoffs have been set.)

Implement a deletion operation. This is a public function with the following function declaration:

```
public void delete(Point2D pt) throws Exception
```

If the point is not found in the tree, the deletion function throws an `Exception` with the error message `"Deletion of nonexistent point"`. Otherwise, it removes this point from the kd-tree. If the deletion results in an external node becoming empty, and this external node is not the root, then the tree should be restructured to eliminate this empty external node. This will induce other changes in the tree. Since an internal node cannot have a `null` child, removing an external node results in the removal of its parent. The external node's sibling will now be *promoted*, in the sense that it will become a child of its former grandparent. Part of the challenge is for you to figure out these changes.

Note that repeated deletions can cause the tree to become unbalanced, but there is no requirement that you do anything about rebalancing the tree. (When we study scapegoat trees, we'll see that there is an easy way to do this.)