Programming Assignment 5 Checklist: Kd-Trees

Frequently Asked Questions

I'm nervous about writing recursive search tree code. How do I even start on KdTreeST.java? You can use BST.java as a guide. The trickiest part is understanding how the `put()` method works. You do not need to include code that involves storing the subtree sizes (since this is used only for ordered symbol table operations).

What makes KdTree a hard assignment? How do I make the best use of my time? Debugging performance errors is very hard on this assignment. It is very important that you understand and implement the crucial optimizations listed in the assignment text, namely:

- *Range search pruning.* Do not search a subtree whose corresponding rectangle does not intersect the query rectangle.
- *Nearest neighbor recursive call ordering.* When there are two subtrees to explore, choose first the subtree that is on the same side of the splitting line as the query point. This rule implies that if one of the two subtrees contains the query point, you will consider that subtree first.
- *Nearest neighbor pruning.* Do not search a subtree if no point (that could possibly be) in its corresponding rectangle could be closer to the query point than the best candidate point found so far. Nearest neighbor pruning is most effective when you properly order the recursive calls because a good candidate point enables you to do more pruning.

Do *not* start range search or nearest neighbor until you understand these rules.

Is a point on the boundary of a rectangle considered inside it? Do two rectangle intersect if they have just one point in common? Yes and yes (which is consistent with the implementation of RectHV.java).

Can I use the `distanceTo()` method in `Point2D` and `RectHV`? No, you may use only the subset of the methods listed. You should be able to accomplish the same result (more efficiently) with `distanceSquaredTo()`.

Can I use the `X_ORDER()` and `Y_ORDER()` comparators in `Point2D`? No, you may use only the subset of the methods listed. You should be able to accomplish the same result by calling the methods `x()` and `y()`.

What should I do if a point has the same *x*-coordinate as the point in a node when inserting or searching in a 2d-tree? Go to the right subtree as specified in the assignment under *Search and insert*.

What should I do if a point is inserted twice in the data structure? The data structure represents a *symbol table*, so you should replace the old value with the new value.

What should `points()` return if there are no points in the data structure? What should `range()` return if there are no points in the range? An `Iterable<Point2D>` object with zero points.

In which order should the `points()` method in `PointST` return the points? The assignment does not specify the order, so any order is fine.

What should `nearest()` return if there are two (or more) nearest points? The assignment does not specify, so any nearest point is fine.

How much memory does a `Point2D` object use? For simplicity, assume that each `Point2D` object uses 32 bytes—in reality, it uses a bit more because of the `Comparator` instance variables.

How much memory does a `RectHV` object use? You should look at the code and analyze its memory usage.

I run out of memory when running some of the large sample files. What should I do? Be sure to ask Java for additional memory, e.g., `java-algs4 -Xmx1600m RangeSearchVisualizer input1M.txt`.

I get the checkstyle warning "More than 7 parameters". Do I need to fix it? Ordinarily, it is bad style to have functions with too many parameter variables. If breaking this rule when implementing the `put()` method leads to clearer code, you can ignore the warning.

I get the checkstyle warning "Assignment of parameter 'champion' is not allowed." Do I need to fix it? Many programmers consider it bad style to change the value of a parameter variable. If breaking this rule when implementing the `nearest()` method leads to clearer code, you can ignore the warning.

Testing

Testing. A good way to test `KdTreeST` is to perform the same sequence of operations on both the `PointST` and `KdTreeST` data types and identify any discrepancies. Indeed, this is how most of the autograder test are performed. They key is to implement a reference solution in which you have confidence. The brute–force implementation `PointST` can serve this purpose in your testing.

- The sample client RangeSearchVisualizer.java reads a set of points from a file (given as a command–line argument) and draws them to standard drawing. It also highlights the points inside the rectangle that the user selects by dragging the mouse. Specifically, it colors red the points returned by the method `range()` in `PointST` and blue the points returned by the method `range()` in `KdTreeST`.
- The sample client NearestNeighborVisualizer.java reads a set of points from a file (given as a command–line argument) and draws them to standard drawing. It also highlights the point closest to the mouse. Specifically, it colors red the point returned by the method `nearest()` in `PointST` and blue the point returned by the method `nearest()` in `KdTreeST`.

Warning: both of these clients will be slow for large inputs because (i) the methods in the brute–force implementation are slow and (ii) drawing all the points is slow.

Sample input files. The directory kdtree contains some sample input files in the specified format.

Possible Progress Steps

These are purely suggestions for how you might make progress on KdTreeST.java. You do not have to follow these steps.

1. **Complete the KdTree worksheet.** Here is a set of practice problems for the core kd-tree methods. Here are the answers.

2. **Node data type.** There are several reasonable ways to represent a node in a 2d-tree. One approach is to include the point, a link to the left/bottom subtree, a link to the right/top subtree, and an axis-aligned rectangle corresponding to the node.

```
private class Node {
   private Point2D p;      // the point
   private Value value;    // the symbol table maps the point to this value
   private RectHV rect;    // the axis-aligned rectangle corresponding to this node
   private Node lb;        // the left/bottom subtree
   private Node rt;        // the right/top subtree
}
```

   Since we don't need to implement the *rank* and *select* operations, there is no need to store the subtree size.

3. **Writing KdTreeST.**

   - Write `isEmpty()` and `size()`. These should be very easy.

   - Write a simplified version of `put()` which does everything except set up the `RectHV` for each node. Write the `get()` and `contains()` method, and use these to test that `put()` was implemented properly. Note that `put()` and `get()` are best implemented by using private helper methods analogous to those found on page 399 of the book or by looking at BST.java. We recommend using the orientation (vertical or horizontal) as an argument to these helper methods.

     A common error is to not rely on your base case (or cases). For example, compare the following two `get()` methods for searching in a BST:

```
private Value get(Node x, Key key) {
  if (x == null) return null;

  int cmp = key.compareTo(x.key);
  if      (cmp < 0) return get(x.left, key);
  else if (cmp > 0) return get(x.right, key);
  else              return x.value
}

private Value overlyComplicatedGet(Node x, Key key) {
  if (x == null) return null;

  int cmp = key.compareTo(x.key);
  if      (cmp < 0) {
     if (x.left == null)  return null;
     else                 return overlyComplicatedGet(x.left, key);
  }
  else if (cmp > 0) {
     if (x.right == null) return null;
     else                 return overlyComplicatedGet(x.right, key);
  }
  else                    return x.value
}
```

     In the latter method, extraneous null checks are made that would otherwise be caught by the base case. Trust in the base case. Your method may have additional

base cases, and code like this becomes harder and harder to read and debug.

- Implement the `points()` method. Use this to check the structure of your kd-tree.
- Add code to `put()` which sets up the `RectHV` for each `Node`.
- Write the `range()` method. Test your implementation using RangeSearchVisualizer.java, which is described in the testing section.
- Write the `nearest()` method. Test your implementation using NearestNeighborVisualizer.java, which is described in the testing section.

A video, worksheet,and coding tips are provided for those wishing additional assistance. Be forewarned that videos were made in early 2014 and is somewhat out of date. For example the API has changed.

Optimizations

These are many ways to improve performance of your 2d-tree. Here are some ideas.

- **Squared distances.** Whenever you need to compare two Euclidean distances, it is often more efficient to compare the squares of the two distances to avoid the expensive operation of taking square roots. Everyone must implement this optimization because it is easy to do; it is likely a bottleneck; and you are permitted to call `distanceSquaredTo()` but not `distanceTo()`.
- **Range search.** Instead of checking whether the query rectangle intersects the rectangle corresponding to a node, it suffices to check only whether the query rectangle intersects the splitting line segment: if it does, then recursively search both subtrees; otherwise, recursively search the one subtree where points intersecting the query rectangle could be.
- **Save memory.** You are not required to explicitly store a `RectHV` in each 2d-tree node (though it is probably wise in your first version).