CMSC 420: Spring 2023

## Programming Assignment 3: Clustering and the Farthest-First Traversal

Handed out: Tue, Apr 25. Due **Wed, May 17, 11:59pm**.

**Overview:** In this assignment we will extend our kd-tree and heap data structures from the earlier assignments to an application in data science and machine learning. The basic assignment (which is worth 80% of the grade, excluding the style/efficiency points) involves utilities for maintaining a clustering of a set of points, and the full assignment involves an interesting algorithm called the *farthest-first traversal*. The basic assignment largely involves modifications to the `SMkdTree` data structure, and the full assignment involves both both the `SMkdTree` and the `WtLeftHeap` structure. (We will make versions of our solutions to Programming Assignments 1 and 2 available to you. You may either use yours or modify ours without penalty.)

**Center-Based Clustering:** Suppose that we are given a large set of points, called *sites*, which we wish cluster. Each cluster is modeled by a point called the *cluster center*. Each site is assigned to a cluster based on its closest cluster center. For example, in Fig. 1(a), we show a set of sites (black) and cluster centers (blue), and in Fig. 1(b) we illustrate the assignment of sites to centers, where the sites in each colored region are assigned to the cluster center at the middle of the region.

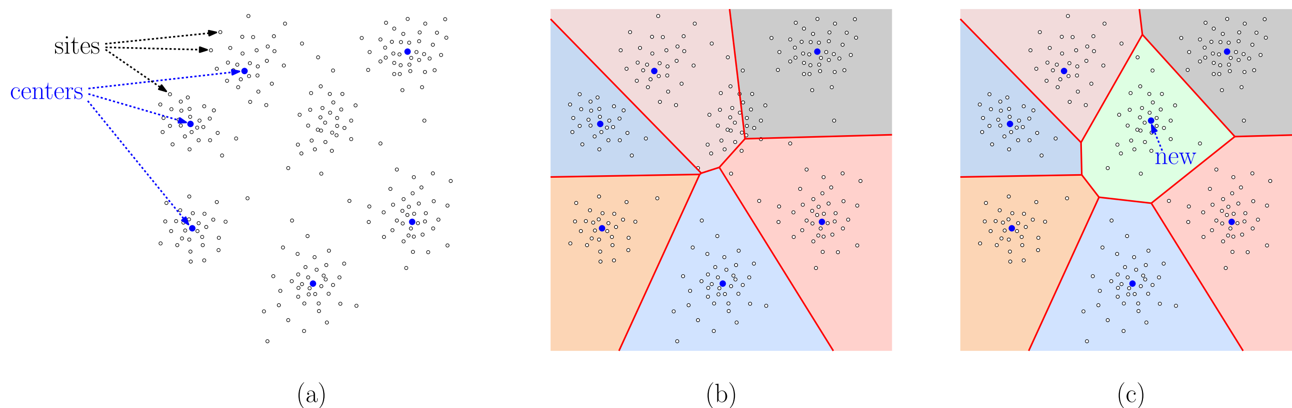


Figure 1: Center-based clustering. Each site is assigned to the cluster center that is closest to it.

In many applications, the centers are computed through an incremental process, where the centers are added one by one, based on where the need is greatest. Think of sites as phones and the cluster centers cell phone towers. Whenever a new tower is opened, many sites change their cluster membership (as is the case for the sites in the green region of Fig. 1(c)).

In many applications, cluster centers must be chosen from among the sites. We will not assume this, and in our tests, the cluster centers will always be distinct from the sites.

**Basic Assignment: Cluster Assignment:** In this part of the assignment, you will both modify the `SMkdTree` and implement new data structure, called `ClusterAssignment`, which maintains two point sets in $\mathbb{R}^2$, a set of sites and a set of cluster centers. It supports the insertion

and deletion of sites and the insertion of cluster centers. The objective is to efficiently maintain an *assignment* of each site to its closest center.

The sites are stored in the kd-tree in the same manner as in Programming Assignment 2, but we will augment the kd-tree to efficiently process the addition of new cluster centers. Recall that each node of the tree is associated with a rectangular cell, call it `cell`. In addition, for each node we maintain a set of center points, called *contender*, which might be the closest center to some site within this cell. How do we determine which centers are contenders for a given node?

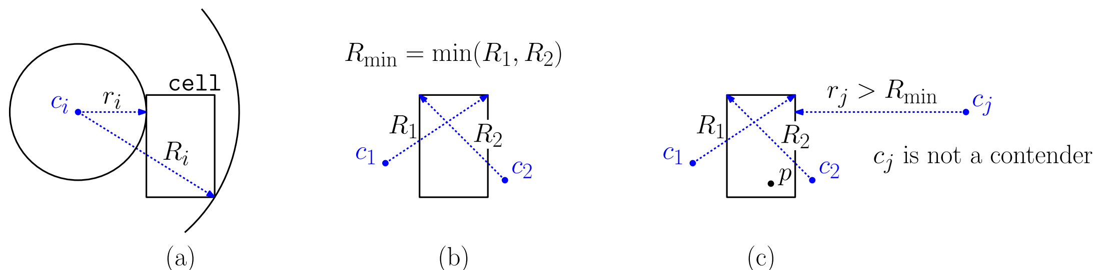


Figure 2: Contenders for the closest center.

Suppose that we have a set of existing contenders for some cell, $\{c_1, \ldots, c_k\}$. For each of these cluster centers $c_i$, we compute its closest squared distance to the cell, $r_i$, and its largest squared distance to the cell $R_i$ (see Fig. 2(a)). For any site $p$ in the cell, we know that $r_i \leq \text{dist}^2(p, c_i) \leq R_i$. Let $R_{\min} = \min_{1 \leq i \leq k} R_i$ be the minimum among the $R_i$ values (see Fig. 2(b)). If for any center $c_j$, we have $r_j > R_{\min}$, we can infer that $c_j$ cannot be a contender. Why? Suppose that $c_1$ is the contender with the smallest $R$ value, so that $R_{\min} = R_1$. The closest that any site $p$ of the cell could be to $c_j$ is $r_j$. The farthest it could be from $c_1$ is $R_1$. Therefore, we have

$$\text{dist}^2(p, c_j) \;\geq\; r_j \;>\; R_{\min} \;=\; R_1 \;\geq\; \text{dist}^2(p, c_1),$$

which implies that $p$ is closer to $c_1$ than $c_j$, no matter which point $p$ is selected within the cell. (Note that we cannot infer that $c_1$ is $p$'s closest center, only that $c_j$ *not* its closest center).

**Augmenting the kd-tree:** We will augment our kd-tree as follows. For every node (both internal and external) we will maintain a list of closest-center contenders as described above. (Because we will need to both insert and delete entries, it is recommended that you implement this as a Java `LinkedList`, rather than an `ArrayList`.) For example, this could be made a member of your general kd-tree node, so that both internal and external nodes will inherit it.

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

    private abstract class Node {
        LinkedList<LPoint> contenders; // list of this node's contenders
        // ... other Node stuff
    }

    class InternalNode extends Node { /* ... */ }
```

```
        class ExternalNode extends Node { /* ... */ }
        // ... and so on
    }
```

Every node will have at least one contender. To guarantee this, even when the tree is empty, the tree constructor will be given an inition starting center, `startCenter`. Initially, the tree consists of an empty leaf node whose cell is `rootCell`, and whose associated contender list contains a single entry `startCenter`.

**Adding a Center:** A new center, called `center`, is added as follows. Observe that if a center is a contender for some node, it is automatically a contender for all its ancestors. (We'll leave this as an exercise for you.) This means that we can employ a top-down process for filtering this center into all the nodes for which it is a contender.

The process works as follows. We create a recursive helper to do this, which will be a member function of the internal and external node classes. The initial call is at the root node. We pass in the new `center` and the root cell. In general, let `u` denote the current node being visited in the recursive process.

- Iterate through the points of `u.contenders` and for each, compute its $R_i$ value,[1] that is, the largest squared distance from this center to the current node's cell. Save the minimum value among all of these, including the $R$ value for `center` itself, as $R_{\min}$.
- Iterate through the elements of `u.contenders` a second time. For each compute its $r_i$ value,[2] that is, the closest squared distance from this center to the current node's cell. If for any contender, $r_i$ is strictly greater than $R_{\min}$, we remove the center corresponding to $r_i$ from the contender's list.
- Letting $r_{\text{new}}$ denote the $r_i$ value for the new center, if $r_{\text{new}} \leq R_{\min}$, add `center` to `u.contenders`. Further, if this is an internal node, apply the function recursively to add `center` to its left and right subtrees (using `leftPart` and `rightPart` to determine their associated cells). On the other hand, if $r_{\text{new}} > R_{\min}$, we are done and simply return.

The above process computes the $R_i$ and $r_i$ values each time we visit the node. For greater efficiency, it would be a good idea to compute these values once, and save them in the cell. We can also compute $R_{\min}$ and update it incrementally. (This does not affect the asymptotic time, however, because we will still need to traverse the list to determine which contenders to delete.)

You might wonder why we need to maintain a full list of contenders for each external node. Why not just compute the *exact* closest center to the site stored in this node. The reason is for the sake of future insertions and rebuildings, which we discuss next.

**Deleting a Center:** For this assignment, we *do not* delete centers.

**Inserting/Deleting Sites and Rebuilding:** Inserting and deleting sites is essentially the same as in Programming Assignment 2, but some changes are needed to deal with the contenders.

[1] To help you, we have added a function to the `Rectangle2D` class, `maxDistanceSq`, which compute the maximum squared distance between a point and a rectangle.

[2] You can use the function `distanceSq` in the `Rectangle2D` class.

Because contenders depend only on the cell, not the points within the cell, the insertion and deletion of sites does not affect the node contenders.

However, rebuilding a subtree will generate new nodes with new cells, and hence with new contenders. The approach is quite simple, however. Whenever we rebuild the subtree rooted at some node `u`, we save a reference to its contender list, `u.contenders`. We then employ the rebuilding process from Programming Assignment 2. Finally, we iterate through the saved contender list, and add each of these centers one by one. (There may be more efficient ways to process them as a batch, but this is sufficient for full credit.)

**Basic-Assignment Requirements:** The basic part of the assignment will involve creating a new class for storing sites, centers, and assignments, called `ClusterAssignment`. To implement this, it will also be necessary to make modifications to your `SMkdTree`.

Let's first describe this new class, called `ClusterAssignment`. Its job is to maintain a dynamic collection of sites and cluster centers and to keep track of the assignment of each site to its closest cluster center. It supports inserting and deleting sites and adding new centers. It has the same generic structure as the `SMkdTree`:

```
public class ClusterAssignment<LPoint extends LabeledPoint2D> { /* ... */ }
```

At a minimum, the private data for the class consists of a kd-tree for storing the sites and a list for storing the centers. (You will probably need to add additional data items as well.)

```
private SMkdTree<LPoint> kdTree; // storage for the sites
private ArrayList<LPoint> centers; // storage for the centers
```

The `ClusterAssignment` class supports the following public functions:

`ClusterAssignment(int rebuildOffset, Rectangle2D bbox, LPoint startCenter)`: The constructor creates a new structure with an empty kd-tree, which will eventually hold the sites and it initializes the cluster center set to consist of the single cluster point `startCenter`.

`void addSite(LPoint site)`: Inserts a new site in the structure. This will involve inserting the site in the kd-tree, which may throw an exception if the site is a duplicate of an existing site. (If any new nodes are created, their contender lists must be created as well.)

`void deleteSite(LPoint site)`: Deletes an existing site from the structure. This will involve deleting the site from the kd-tree, which may throw an exception if the site does not exist. (If this induces a rebuild, the contender lists for all the nodes of the tree will need to be generated.)

`void addCenter(LPoint center)`: This adds a new center point. The point is added to the list of centers and it is processed through the kd-tree, adding it to all the nodes for which this point is a contender.

`int sitesSize()`: Returns the number of sites in the structure. (This just invokes `size()` on the kd-tree.)

`int centersSize()`: Returns the number of centers in the structure. (This just invokes `size()` on the list of center points.)

`void clear()`: Removes all the sites by clearing the kd-tree. It also removes all the centers, except `startCenter`, given in the constructor.

`ArrayList<String> listKdWithCenters()`: This just invokes `listWithCenters()` (described below) on the kd-tree.

`ArrayList<String> listCenters()`: This produces an `ArrayList` of strings one per center, where each string contains the label and coordinates of the center. The list should be sorted in alphabetic order. An example is shown below, left.

```
listCenters():            listAssignments():
------------------------- ------------------------------
BWI: (80.0,80.0)          [DCA->BWI] distSq = 181.0
ORD: (20.0,60.0)          [ATL->ORD] distSq = 200.0
PVD: (90.0,20.0)          [JFK->PVD] distSq = 400.0
...                       ...
```

`ArrayList<String> listAssignments()`: This produces an `ArrayList` of strings, one per site. Each string contains the site, the associated closest center, and the squared distance between the site and center. The list should be sorted in increasing order of squared distance. If there are duplicates, ties should be broken lexicographically by the $(x, y)$ coordinates of the site. (An example is shown above, right. The first line means that the site `DCA`'s closest center is `BWI`, and the squared distance between these two points is 181.)

We can compute this as follows. First, we perform a traversal of the kd-tree. (Pick any traversal order you like.) On arriving at a non-empty external node `u`, we access its list of contenders `u.contenders`, and select the one that minimizes the squared distance to `u.point`. If there are ties for the closest, take the contender that has the lexicographically minimum $(x, y)$ coordinates. Create a triple consisting of the site, the closest center, and the squared distance, and add it to the list of assignments. On returning, sort this list by distances, breaking ties as described above.

How to implement these triples? A good way to do this is to implement a new class, call it `AssignedPair`. (This could be a standalone public class in `AssignedPair.java` or an inner class within `ClusterAssignment`, its up to you.) Each instance of this class contains a site (`LPoint`), a center (`LPoint`), and the squared distance between them (`double`). This class should "`implement Comparable<AssignedPair>`". This means that it must implement a public function "`int compareTo(AssignedPair o)`", which compares this assigned pair to another one. This compare function should be based primarily on the squared distance, but if two distances are equal, it breaks the tie based on the lexicographical order of the sites. (This always succeeds, because each site appears only one assignment, and no two sites have the same coordinates.) Because this class implements the `Comparable` interface, you can feed it into `Collections.sort` to obtain the sorted list.

Next, let us present the possible modifications to the kd-tree. You will need to make changes to handle the processing of center points. *The following are suggestions.* Since these functions are just going to be used internally by your `ClusterAssignment` and `FarthestFirst` classes, you may add/remove/modify these however you see fit.

**SMkdTree(int rebuildOffset, Rectangle2D rootCell, LPoint startCenter):** The constructor has been modified to include a new argument, `startCenter`. As described earlier, this is the initial center. (This is mainly a convenience, which allows you to assume that the contender lists are always non-empty.) The initial structure consists of a single empty external node whose contender list contains a single entry `startCenter`.

**void addCenter(LPoint center):** Adds `center` as a new center and updates all affected contender lists appropriately. You may assume that `center` is distinct from any of the sites that are currently in the tree (in both its label and coordinates). Note however, that it is possible to delete a site, and add it later as a center. (My version of this function returned an array-list of sites whose center had changed, but you are free to do whatever you like.)

**void clear():** Clears the kd-tree, removing all the sites. It also removes all the centers, except `startCenter`, given in the constructor.

**ArrayList<String> listWithCenters():** This is identical to the standard `list` function, but for each node we also include a list of the contenders (just the labels) in curly braces. For the sake of consistency, order the list alphabetically by the label. (The most efficient way to do this is to maintain the contenders lists sorted by label, but you will get full credit if you just invoke `Collections.sort` to sort them each time you perform this operation.) We show an example below, where we have added centers `BWI`, `ORD`, and `PVD`.

```
list():                    listWithCenters():
-------------------------- -------------------------------
(x=50.0) 6:4               (x=50.0) 6:4 => {BWI ORD PVD}
(y=50.0) 3:1               (y=50.0) 3:1 => {BWI ORD PVD}
(x=70.0) 2:0               (x=70.0) 2:0 => {BWI PVD}
[DCA: (70.0,71.0)]         [DCA: (70.0,71.0)] => {PVD}
[SEA: (50.0,51.0)]         [SEA: (50.0,51.0)] => {ORD PVD}
...                        ...
```

When we are working with much larger examples, the contender lists can be quite long. If there are 11 or more contenders, just list the first 10 in alphabetical order and add "`...`" at the end, for example.

```
(y=0.0) 30:0 => {ABJ ABV AKR AZR BHX BLI BNI BOY BSK BYK...}
```

**Full Assignment: Farthest-First Traversal:** Modern data sets are massive. An important step in analyzing such large sets involves *sketching*, where the original data set is represented by a much smaller subset. The most common approach to sketching is through random sampling, but there are times when random sampling is not the best option.

Consider, for example, an application where the point set consists of all the cell towers in the USA, and the question under consideration is "What is the farthest distance from any phone in the USA to the nearest cell tower?". A random sample of cell towers would heavily sample urban regions where there are lots of phones and lots of towers, but it would miss large rural regions where there are few (see Fig. 3(a)). For this particular query, a better approach would be to distribute the samples based instead on distance. In particular, we want our sample to be as close as possible to every cell phone user in the USA (see Fig. 3(b)).

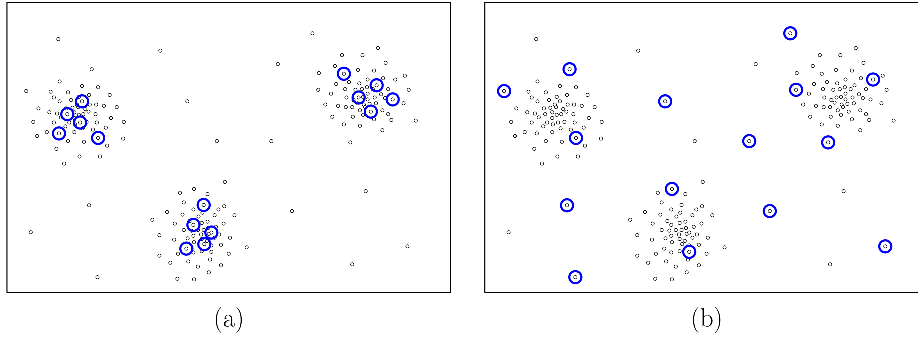


Figure 3: Sketching via (a) random sampling versus (b) distance-based sampling.

A common way to produce such a sketch is based on a concept called the *farthest-first traversal*. Given a set $S$ of $n$ points such a traversal visits every point of the set exactly once. The initial point of the traversal, call it $c_0 \in S$, can be chosen arbitrarily. The next point is the point of $S$ that is farthest from $c_0$. The next is the point that maximizes the distance to the closer of $c_0$ and $c_1$. Generally, suppose that we have already chosen some subset $C_k = \{c_0, \ldots, c_{k-1}\}$. The next point of the traversal is the point $s \in S \setminus C_k$ that is a far away as possible from any of the points chosen so far, or more formally

$$c_k \;=\; \mathrm{argmax}_{s \in S} \left\{ \min_{0 \leq i < k} \mathrm{dist}(s, c_i) \right\},$$

where "argmax" means point that achieves the value of the stuff inside the curly braces. The final sequence $\{c_0, \ldots, c_{n-1}\}$ is called the *farthest-firt traversal* of $S$ (given the starting point $c_0$). Intuitively, the farthest-first traversal attempts to select the points so that in any initial subsequence, the points are spread out as much as possible. (It does not succeed in this formally because this is an NP-hard problem, but it is a provably good approximation. This is related to a famous approximation algorithm for $k$-center clustering, called *Gonzalez's algorithm*.)

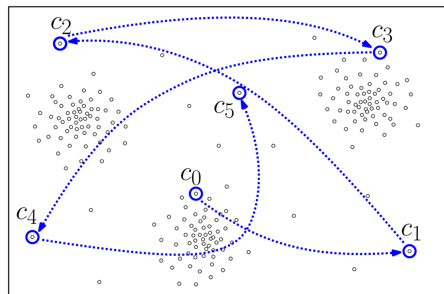


Figure 4: The first six points in a farthest-first traversal starting at $c_0$.

What does this have to do with distance-based sketching? Well, a provably good way to obtain a sketch of any desired size $m$ is to compute the entire farthest-first traversal, and then select the first $m$ points of the traversal. The purpose of Part 2 of the assignment is to implement an efficient algorithm for computing the farthest-first traversal.

**Full Assignment: Farthest-First Traversal:** For the full assignment, you will extend the basic part by implementing a class `FarthestFirst`, which implements an efficient algorithm for the farthest-first traversal of a point set. This will involve a spatial index (the `SMkdTree`) and a priority queue (the `WtLeftHeap` from Programming Assignment 1). It will also make use of much of the functionality of the `ClusterAssignment` class.

The algorithm is given a point set $S$, which we call the *sites*, and an starting point $c_0$, which we will call `startCenter`. The sites are inserted into a kd-tree, and $c_0$ is the initial cluster center for all the points. The algorithm proceeds by selecting one of the sites, removing it from the kd-tree, adding it to the existing set of cluster centers (the points in the traversal), and updating the closest cluster center for all the sites. Thus, at any time in the process, the cluster centers consist of the points that have been added to the traversal, and the sites consist of the points that are waiting to be added to the traversal.

We have almost everything we need to implement this algorithm efficiently. The insertion and deletion of sites is handled by the kd-tree. Centers can be inserted by invoking the `addCenter` function of the `ClusterAssignment` structure. This in turn causes the kd-tree to update the closest center contenders for each node.

There is only one operation remaining, that of finding the site that has the largest distance to its closest cluster center. It would too inefficient to generate all the site-center assignments and select the one with the largest distance. Instead, we will use our `WtLeftHeap` data structure to store all the site-center assignments, sorted by the squared distance between the site and its associated center.

Recall that when discussing `listAssignments()` above, we suggested defining an inner class `AssignedPair` within `ClusterAssignment` for storing site-center-distance triples. We can create an instance of `WtLeftHeap`, where the keys are squared distances (`double`) and the values are site-center pairs (`AssignedPair`).[3] The heap allows us to efficiently extract the assigned pair with the maximum distance. When we add the new center point to the traversal, a number of sites may discover that they are closer to the newly inserted center. To handle this, we need to perform an `updateKey` operation on all of these sites. This means that we need make use of locators to identify where each assigned pair appears in the heap. We need a place to store these locators. Our solution will be to use a Java `HashMap` to map site labels to heap locators.

We can describe a single step of the farthest-first traversal. Initially, we have the starting center $c_0$ and a collection of points (sites) stored in a kd-tree. (The point $c_0$ itself is *not* stored in the kd-tree.) In general, we will have computed the points in a partial traversal $C_k = \{c_0, c_1, \ldots, c_{k-1}\}$. Call these points the *centers*. The remaining points of $S$, called *sites*, are stored in the kd-tree. Every site is assigned to its closest center, and each of these site-center pairs has an associated `AssignedPair` for it. These pairs are stored in a max heap, sorted by the squared distance between the site and its center. Finally, every entry in the

[3] You might wonder why we chose to use the squared distance as the key, rather than just using the comparator within the `AssignedPair` class. The latter is really the better approach, because it provides us with automatic tie-breaking when squared distances are equal. However, there are subtle programming issues that can arise when the key and value are the same. We have decided to avoid these issues by keeping keys and values distinct. We will avoid distance ties in our test data.

heap has a locator, and these locators are stored in the Java `HashMap`, where the key is the site's label. We proceed as follows.

- If there are no more remaining sites (the kd-tree is empty), we are done, and the traversal is complete.
- Otherwise, extract the site-center pair from the heap with the maximum squared distance. Let $s$ denote the associated site.
- Delete $s$ from the kd-tree. This former site now becomes a new center, called $c_k$.
- Add $c_k$ to the end of the traversal.
- Update kd-tree contenders appropriately for the addition of $c_k$. Let $U = \{s_1, \ldots, s_m\}$ denote sites of the kd-tree whose closest center has changed to $c_k$ as a result. For each element $s_i$ in $U$, do the following:
  - Use $s_i$'s code to access its entry in the hash-map to obtain its heap locator.
  - Using this locator, update the key of $s_i$'s heap entry to the squared distance between $s_i$ and $c_k$.

**Full-Assignment Requirements:** For the full assignment, we will create a new class, called `FarthestFirst`. You can start by making a copy of `ClusterAssignment` and make modifications to it. This class has the same generic structure as the `SMkdTree`:

```
public class FarthestFirst<LPoint extends LabeledPoint2D> { /* ... */ }
```

At a minimum, the private data for the class consists of a kd-tree for storing the sites and a list for storing the centers. (You will probably need to add additional data items as well.) As mentioned earlier, the class `AssignedPair` can be either be an inner class or a standalone class. It's up to you.

```
public class FarthestFirst<LPoint extends LabeledPoint2D> {
    private class AssignedPair implements Comparable<AssignedPair> {
        private LPoint site; // the site
        private LPoint center; // its assigned center
        private double distanceSq; // squared distance to the assigned center
        // ... addtional data as needed
    }

    private LPoint startCenter; // the initial center
    private WtLeftHeap<Double, AssignedPair> heap; // heap for assigned pairs
    private HashMap<String, ...> map; // a map used for saving heap locators
    private SMkdTree<LPoint> kdTree; // kd-tree for sites
    private ArrayList<LPoint> traversal; // the traversal
```

The role played by the set of centers in the basic assignment is taken by the `traversal` object. Here are the following public functions of `FarthestFirst`:

`public FarthestFirst(int rebuildOffset, Rectangle2D bbox, LPoint startCenter):` The constructor creates a new structure with an empty kd-tree, which will eventually hold the sites and it initializes the traversal to consist of the single cluster point `startCenter`.

`void addSite(LPoint site):` Inserts a new site in the structure. This will involve inserting the site in the kd-tree, which may throw an exception if the site is a duplicate of an existing site. (If any new nodes are created, their contender lists must be created as well.) Let `center` denote its closest center point. We create a new `AssignedPair` consisting of (`site`, `center`) which we add to the heap, where the key is the squared distance between them. We save the locator in the map.

`LPoint extractNext():` Performs one step of the farthest-first traversal (described in the bulleted list above). If there are no remaining sites (the kd-tree is empty), it returns `null`. Otherwise, it performs the various described actions (deletion from the kd-tree, adding to the traversal, updating the kd-tree contenders and the updating the keys of affected sites). Finally, it returns the point that has just been added to the traversal.

`int sitesSize():` Same as for `ClusterAssignment`.

`int traversalSize():` (Analogous to `centersSize` for `ClusterAssignment`.) Returns the number of centers in the traversal. (This just invokes `size()` on the traversal list.)

`void clear():` Clears everything as in `ClusterAssignment` and also clears the hash map, the heap, and the traversal.

`ArrayList<String> listKdWithCenters():` Same as for `ClusterAssignment`.

`ArrayList<LPoint> getTraversal():` This simply returns the `ArrayList` containing the points of the traversal.

`ArrayList<String> listCenters():` Same as for `ClusterAssignment`, except using the traversal rather than the list of centers.

`ArrayList<String> listAssignments():` Same as for `ClusterAssignment`.

**Skeleton Code:** As usual, we will provide skeleton code on the class Projects Page. You will need to fill in the implementations of the `SMkdTree.java`, `WtLeftHeap.java`, `ClusterAssignment` and (for the full assignment) `FarthestFirst`.

As in Programming Assignment 2, we will provide utilities (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, you may want to create additional classes, like `AssignedPair.java`, or if you want 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_s23`" 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) Your functions for updating contender lists should operate efficiently. Imagine that your kd-tree has millions of points and your contender lists can contain thousands of entries. Java provides built-in functions to perform operations on basic data structures (array-lists, linked-lists, hash-maps, etc.) but these are not all equally efficient. Based on your knowledge of data structures, you should be able to determine how efficient these functions are and structure your program to apply them appropriately.

Also, your program should avoid visiting nodes that cannot possibly be affected by the operation. For example, if you are inserting a new center, you should only visit nodes that are close enough to the newly added center that they might reasonably need this center as a contender. We are not going to apply a rigid policy in grading, but we will deduct points if your algorithm visits nodes that are clearly irrelevant to the operation.

**Style requirements:** (5% of the final grade) The style requirements are essentially the same as for the previous assignment. Your code should be relatively clean, and comments should be provided to explain important functions. You may refer to our canonical solutions for guidance (but you do not need to be quite as excessive in commenting as I am).

**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. Unless you created your own additional files, you should only submit only the four files mentioned above (`SMkdTree.java`, `WtLeftHeap.java`, `ClusterAssignment.java` and (for the full assignment) `FarthestFirst.java`). If you created any additional files for utility objects, you will need to upload those as well.

**Challenge Problem:** For the class `ClusterAssignment`, implement a new function,

```
void deleteCenter(LPoint center) throws Exception
```

that deletes the given center point. If this point does not exist among the current set of centers, an `Exception` is thrown with the message `"Deletion of nonexistent center"`. This removes `center` from all the contender lists in the kd-tree, and updates these lists. Note that deleting a center can increase the $R_{\min}$ value for a node, and this can result in "resurrecting" contenders that were filtered out by `center` earlier in the process.

At the top of your submission, add a comment indicating that you attempted this operation, and present a clear explanation (in plain English) of how you implemented this function.

**Hint:** For full credit, this should be implemented in an efficient manner. There is no clear optimum strategy for doing this, and any solution is likely to involve tradeoffs. There are some pitfalls you should avoid.

- First, deleting a center in one region of the kd-tree should not require visiting all the nodes of the kd-tree. Ideally, your code should only need to visit the nodes where `center` was among its list of contenders. (Or, if it does visit a node that does not contain `center` among its contenders, it should discover this and not recurse on its children.)
- Second, as mentioned above, deleting `center` from the contender list of some node may result in a need to resurrect contenders that were filtered out earlier. It would be inefficient to consider all possible centers as candidates for resurrection. (As an example, if you remove a cell phone tower in Maryland, you should not need to consider cell phone towers in California as candidates for replacing it.) Consider more efficient ways to identify a smaller subset of local contenders.

This is an open-ended design problem, and there really is no single correct answer. Explain in your leading comment what strategy you decided upon.