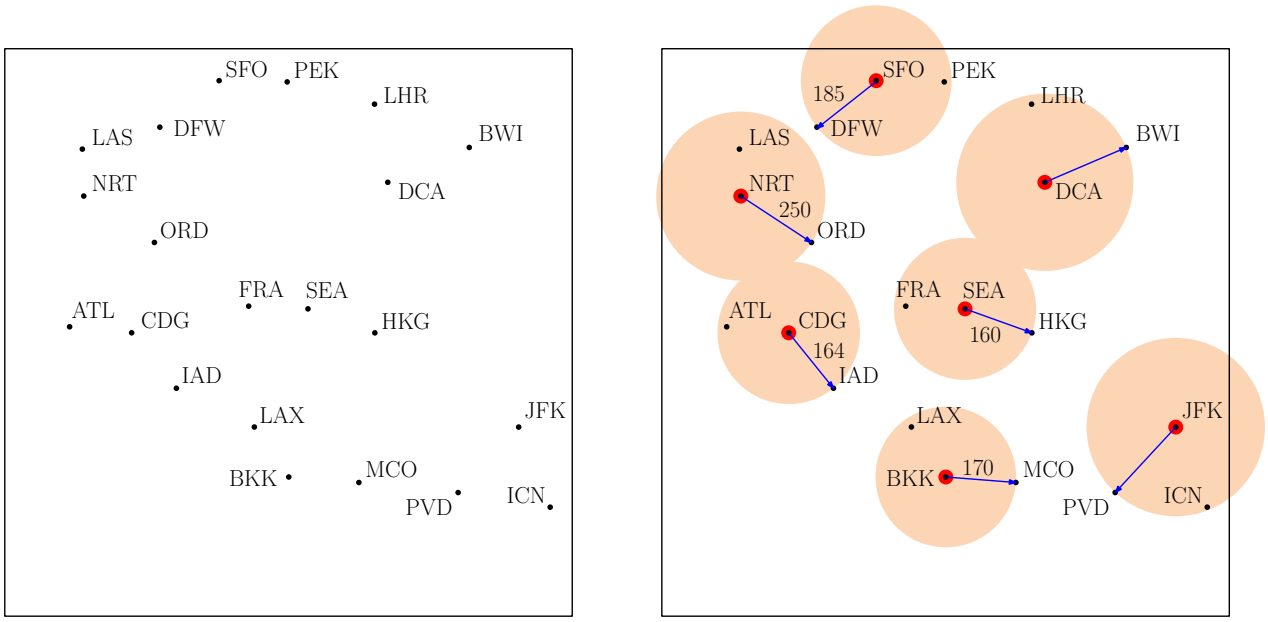
CMSC 420: Fall 2022

Programming Assignment 3: Capacitated Facility Location (Updated 12/05)

Handed out: Thu, Dec 1. Due **Mon, Dec 12, 11:59pm** through Gradescope.

Background: In this assignment, we will ask you to combine the various data structures we have implemented this semester to solve an important optimization problem from the field of operations research. This falls within a broad category of optimization problems known as *facility location*. To motivate the problem, imagine that you are in charge of deciding where to put a set of *service centers* for an organization. For example, this might be the locations of cell towers in city, the locations of retail outlets like coffee shops, or the locations of neighborhood facilities like post offices and schools. You want to distribute your service centers close to where your customers/clients are located, but there is a maximum limit, or *capacity*, to the number of clients a given center can serve. It is customary to refer to the client locations as *demand points*, since we think of each of them as demanding service which we need to provide. We don't want our customers to travel too far, so we wish to keep the travel distance from each demand point to its assigned service center to be small.



(a) (b)

Figure 1: (a) A set of 21 demand points and (b) a possible solution to the discrete k-capacitated facility location problem, for $k = 3$. (This test case can be found in `test04-input.txt`)

Discrete Capacitated Facility Location: This problem can be modeled mathematically as follows. Let $P = \{p_1, \ldots, p_n\}$ denote the set of *demand points* in $\mathbb{R}^2$ (see Fig. 1(a)), and the integer k denote the maximum capacity of any center. The objective is to determine the

locations of a set of *service centers* $C = \{c_1, c_2, \ldots\}$, and an assignment of each demand point to a one of these centers, so that each center is assigned to serve at most k demand points (see Fig. 1(b)). For each center c_j, define *service radius* r_j to be the maximum distance to any of the demand points assigned to it. Define the *cost* of the solution, denoted $\text{cost}(C) = \sum_j r_j$. The objective is to compute a set of centers satisfying the capacity constraints and minimizing the cost. This is called the *discrete k-capacitated facility location problem.*

Unfortunately, there is a trivial solution to the problem, namely to make *every* point of P a service center. To prevent this, let us add the additional constraint that every service center must be assigned to *exactly* k demand points (including itself). As a consequence, we will need to add the condition that the capacity k evenly divides the total number of points.

Greedy Heuristic: This problem is NP-hard, so we are not aiming to find an optimal solution. Instead, we will implement a common heuristic solution, called the *greedy solution.* This is computed as follows:

- Initialize the set of service centers to the empty set, that is, $C \leftarrow \emptyset$.
- While $P \neq \emptyset$, repeat the following steps:
  - For each $p_i \in P$ compute its k-nearest neighbors (including itself) from P. Let r_i denote the distance from p_i to its kth nearest neighbor, called its *radius.*
  - Find the point p_i that has the smallest radius value r_i. Add p_i to C. Remove p_i and its $k-1$ other nearest neighbors from P.
- Return C and the associated assignments as the final result.

Implementing the Greedy Heuristic: Unfortunately, a direct implementation of the above algorithm will not be very efficient. First off, computing nearest neighbors without the aid of a data structure is very slow. Also, with each iteration, we delete k points from P, which will affect the nearest-neighbor radii of the other points.

Instead, we will use the data structures we have implemented this semester to help us out. First, we will store the points in a spatial index (the extended kd-tree from Project 2) so that k-nearest-neighbor queries can be answered efficiently. Second, we will store service centers c_j and the associated radius values r_j in a priority queue (the leftist heap from Project 1). To extract the next center, we extract the minimum key from the priority queue.

We still have the problem that whenever we delete a point from P, it might affect the result of an earlier k-nearest-neighbor queries. To handle this, we will store not only the service center and radius, but all c_j's k-nearest-neighbors in the priority queue. When we extract a candidate cluster center, we will search the tree to see that all the points assigned to it still exist. If so, this cluster of points is valid, and add c_j to C and delete all the points in its cluster. If not, we recompute c_j's k-nearest neighbors, and recompute its new radius r_j, and put this entry back in the priority queue. The algorithm is outlined below. (Also see the description of `extractCluster` below.)

- Initialize the set of service centers to the empty set, that is, $C \leftarrow \emptyset$. Create a new (extended) kd-tree and add all the points of P in bulk to this tree. Create an new (leftist) priority queue.

- For each point $p_i \in P$, use the kd-tree to compute its k-nearest neighbors (including itself). Let L_i be a list storing these k points, and let r_i denote the distance to the farthest of these points. Store the key-value pair (r_i, L_i) in the priority queue.
- While $P \neq \emptyset$, repeat the following steps:
  - Use the extract-min operation to remove the pair (r_i, L_i) from the queue having the smallest radius. Let c_i denote the associated cluster center. (Note: In our implementation, we will assume that the elements of L_i are listed in increasing order of distance from the center point. This means that c_i will always be the first element in the list.)
  - For each $q_i \in L_i$, perform a find operation on the kd-tree to see whether q_i is still in the tree. If not, break out of the loop in failure.
  - If we exited the loop with success, delete all the points of L_i from the kd-tree. Add c_i to our list of service centers and the points of L_i (including c_i itself) are the demand points assigned to it.
  - If we exited the loop with failure, there are two cases. If c_i is still in the kd-tree, we perform a k-nearest neighbor search to compute its new list L_i' of nearest neighbors. Let r_i' be the distance to the farthest point in the new list. (Note: Since we assume that L_i' is sorted by distance, r_i' will be the distance from c_i to the last point in the list.) Insert the pair (r_i', L_i') into the priority queue. Otherwise, (if c_i is not in the kd-tree) do nothing. In either case, we stay in the loop until we find a valid cluster.
- Return C and the associated assignments as the final result.

In order to implement the above algorithm, you will need to make two enhancements to the kd-tree from Project 2. First, you will need to implement a delete operation, and second you will need to implement a k-nearest-neighbor operation. We will discuss these in further detail below.

Deletion and k-Nearest Neighbors

What are the main changes to your kd-tree implementation from Project 2? These are the operations of deletion and k-nearest neighbors (k-NN)queries. Along the way we will implement a useful utility data structure.

`void delete(Point2D pt) throws Exception`: This deletes the point `pt` from the tree. It begins by searching for the point. If it is not found, it throws an `Exception` with the error message `"Deletion of nonexistent point"`. Otherwise, it removes this point from the external-node containing it. (Note that this is very different from the deletion operation for standard kd-trees. We do not need to find a replacement node, because all deletions occur from the leaf level.)

If the deletion results in an empty bucket, we may need to take additional action. If the external node is the root of the tree, we just leave the external node empty. (This only happens when we delete the last point, so the tree is now empty.) Otherwise, the tree needs to be modified to avoid having an empty bucket. Let `q` be the external node where

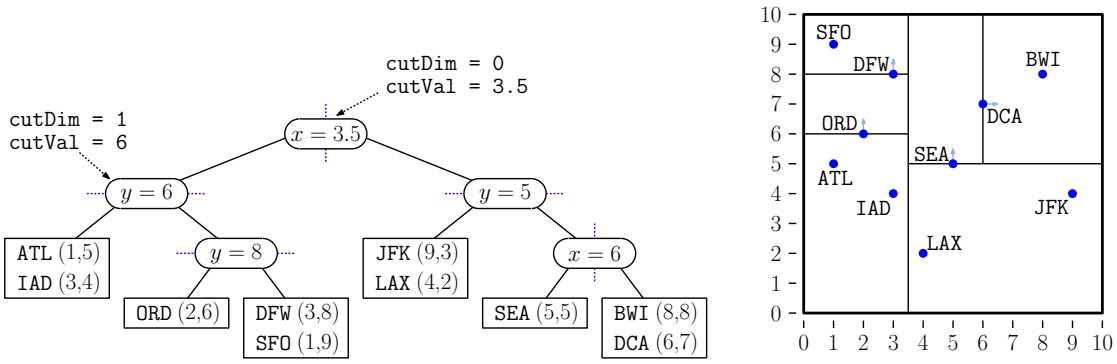


Figure 2: An example of an extended kd-tree with bucket size 2 and bounding box $[0, 10] \times [0, 10]$.

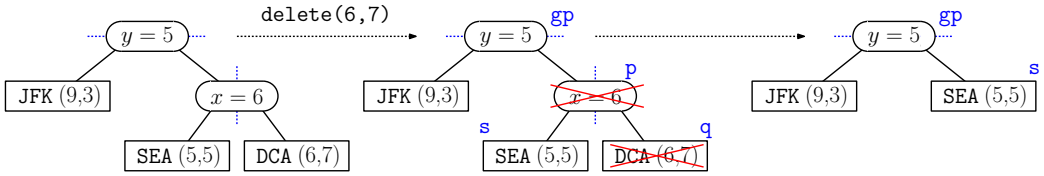


Figure 3: Deleting $(6, 7)$ results in an empty bucket. We remove this external node `q` and its parent `p`, replacing them with the sibling node `s`.

the deletion took place, let `p` be its parent, let `gp` be `q`'s grandparent, and let `s` be `q`'s sibling (see Fig. 3). We unlink both `q` and `p` from the tree, and `s` replaces `p` as the child of `gp`. Note that if `p` is the root of the tree, `s` becomes the new root node. In Fig. 4 we present an example of a series of deletions resulting eventually in an empty tree.

`ArrayList<LPoint> kNearestNeighbor(Point2D q, int k)`: Computes the k points that are closest to the query point `q`. The result is a Java `ArrayList` of `LPoint`. If the tree is empty, it returns an empty `ArrayList`. If k exceeds the number of points in the tree, the list contains only as many points as there are in the tree. The list should be sorted in increasing order of (squared) distance from the query point (see Fig. 5).

How to implement `kNearestNeighbor`? The k-NN algorithm is quite similar to that of the single nearest-neighbor. In the that algorithm, we maintained a variable `best`, which stored the closest point seen so far in the search. To generalize this for k-nearest neighbors, we will maintain the k closest points seen so far. To do this, you will implement a useful utlity data structure, called `MinK`. Each time we encounter a new point, we add it to the `MinK` structure, but no matter how many entries are inserted, this structure *only retains the smallest k items.* In our case, elements in `MinK` will be sorted by their squared distance from the query point, so we only save the k closest points.

Given this data structure, we can answer k-nearest neighbor queries as follows. First, we get the kth (that is, maximum) entry from the `MinK` structure (using an operation `getKth` described below). If the cell is farther away relative to `q`, we know that nothing in this subtree can provide a better point than one of the closest k, so we skip this node. Otherwise, we add a key-value pair consisting of this node's point and its distance from `q` to the `MinK` structure.

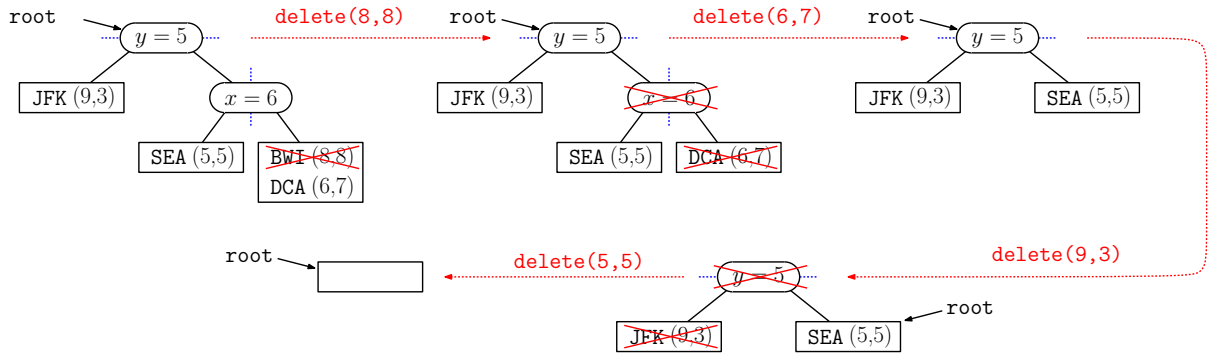


Figure 4: The operation `delete(8,8)` removes `BWI` from its bucket. Next, `delete(6,7)` removes `DCA` from its bucket, resulting in an empty bucket, causing `SEA` to replace its parent. Similarly, `delete(9,3)` removes both `JFK` and its parent, making `SEA` the new root. Finally, deleting `(5,5)` results in an empty tree with a single empty external node.

(Remember, this only saves the k closest points.) We then recursively visit the two children, giving priority to the one that is closer to `q`. The code block below shows how to do this for a standard kd-tree. Note that, unlike the code given in class, there is no need to return `minK`, because it is passed as a reference parameter. You will need to convert this to work with your extended kd-tree.

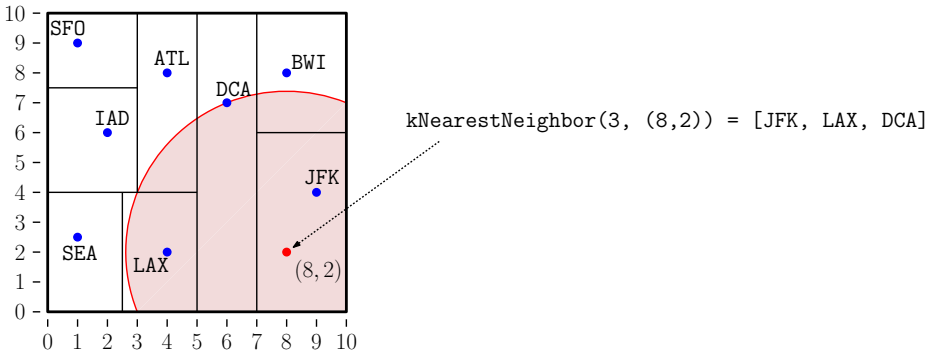


Figure 5: k-NN query for $k = 3$ and $q = (8, 2)$ returns the sorted point list `[JFK, LAX, DCA]`.

The `MinK` data structure: All that remains is to explain how `MinK` works. It is defined generically, based on the key and value pairs it stores. Here is the declaration:

```
public class MinK<Key extends Comparable<Key>, Value> { ... }
```

When we use it for k-NN queries, the keys will be squared distances to the query point (of type `Double`) and the values will be points of type `LPoint`. It has the following public members:

`MinK(int k, Key maxKey)`: This is the constructor, which is given the number of items `k` to maintain and the maximum possible key value. For our puroposes, the value `k` will just be the value k in the k-NN query, and since we are sorting by squared distances,

Helper for k-NN search in a standard kd-tree

```
void kNNHelper(Point2D q, KDNode p, Rect2D cell, MinK minK) {
    if (p == null) return                              // fell out of tree?
    if (cell.distTo(q) > minK.getKth()) return         // cell is too far away?
    minK.add(p.point.distTo(q), p.point)               // add this point
    int cd = p.cutDim                                  // cutting dimension
    Rectangle leftCell = cell.leftPart(cd, p.point)    // get child cells
    Rectangle rightCell = cell.rightPart(cd, p.point)

    if (q[cd] < p.point[cd]) {                         // q is closer to left?
        kNNHelper(q, p.left,  leftCell,  minK)
        kNNHelper(q, p.right, rightCell, minK)
    } else {                                           // q is closer to right?
        kNNHelper(q, p.right, rightCell, minK)
        kNNHelper(q, p.left,  leftCell,  minK)
    }
}
```

the value of `maxKey` can be set to `Double.POSITIVE_INFINITY`. (Save this value, since it will be needed for `getKth` below.)

`int size()`: This returns the current number of elements in the structure. Initially the size is zero, and as elements are added the size increases up to a maximum of k.

`void clear()`: This removes all entries, reseting the structure to its initial empty state.

`Key getKth()`: If the structure has k elements, this returns the maximum key value among these elements. Otherwise, it returns `maxKey` value given in the constructor. (Why `maxKey` and not `null`? The reason is that this is what works most conveniently for the k-NN helper. If we have not yet seen k points, we will never decline the opportunity to visit a node.)

`ArrayList<Value> list()`: Create a `ArrayList` of the values in the structure, sorted in increasing order by their key values. (Note: We will not modify the contents of the values stored in the returned list, so it is not necessary to perform a "deep copy" of the values. You can just copy the references into the resulting `ArrayList`.)

`void add(Key x, Value v)`: This adds the given key-value pair to the current set. If the structure has fewer than k element, this entry is added, increasing the size by one. If the structure has k elements and x is greater than or equal to the largest, the operation is ignored. If the structure has k elements and x is less than the largest, the pair (x, v) is added, and the previous largest is removed. Thus, the size of the structure remains k.

An example is shown in Fig. 6.

How to implement `MinK`? You have two choices on how to implement the `MinK` structure:

Simple and Slow: (Efficiency penalty of 10 points.) Just store the key-value pairs in an array (or `ArrayList`) sorted by key values (similar to Fig. 6). When a new entry is added, simulate one step of insertion-sort, by sliding all the larger elements down one position until finding the slot where the new item fits. If there were already k elements, the largest element just falls off the end of the array. (This takes $O(k)$ time per insertion.)

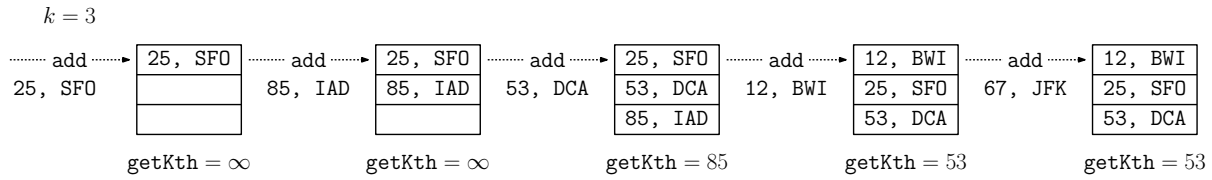


Figure 6: Example of `MinK` operations for $k = 3$ and `maxKey` $= \infty$.

Smart and Speedy: The second method is more efficient, and you will get full credit. Maintain a standard binary heap, as used in heap-sort. (Recall Lecture 5 on Heaps). It will be max-heap ordered with the largest element stored at the root. When a new entry is added, there are two cases. First, if there are currently fewer than k elements in the heap, add this element to the next available position in the heap array and sift it up to its proper location. Otherwise, if there are k elements in the heap, compare the newly added element to the root. (The root has the largest entry.) If it is greater than or equal, ignore the insertion. If it is less than the root's key, replace the root with the new item, and sift it down to its proper location. (This takes $O(\log k)$ time per insertion.) Note that the operation `list` will need to copy the contents of structure to a new `ArrayList` and then sort them by key. If you like, you may do this by implementing heap-sort. But it is fine for full credit to simply invoke `Collections.sort`.)

Back to Facility Location

In order to test your facility location algorithm, you will implement a class called `KCapFL`, for *k-capacitated facility locator*. It will implement the greedy facility location algorithm, and we will add a few additional functions for grading purposes to allow us to inspect how your function works. This class is parameterized by the point type, which like our kd-tree, will just be labeled points. The declaration is `public class KCapFL<LPoint extends LabeledPoint2D>`. It will store the following pieces of private data:

`int capacity`: This is the maximum capacity of any service center, which we have called k up to now

`XkdTree<LPoint> kdTree`: A kd-tree for storing the points

`LeftistHeap<Double, ArrayList<LPoint>> heap`: A leftist heap for storing key-value pairs. Each pair is of the form (r_i, L_i), where r_i is the squared radius of this cluster of points and L_i is the set of points in the cluster. The heap ordered by the (squared) service radii, that is, the squared distance to the kth nearest neighbor. (As with the programming assignment, this is min-heap ordered, with the smallest radius at the root.)

It supports the following public functions:

`KCapFL(int capacity, int bucketSize, Rectangle2D bbox)`: This is the constructor. It sets the capacity, creates a new kd-tree with the given bucket size and bounding box, and creates a new leftist heap.

`void clear()`: This clears the data structure by clearing both the kd-tree and the leftist heap.

`void build(ArrayList<LPoint> pts) throws Exception`: This initializes the structure. First, it checks whether the number of points is strictly greater than zero and is evenly divisible by the capacity. If not, it throws an `Exception` with the error message `"Invalid point set size"`. Otherwise, it performs a bulk-insertion of the points into your kd-tree. Finally, it creates the initial radii. Do do this, it enumerates all the points. For each point p_i it computes its k-nearest neighbors using the kd-tree. et L_i be the `ArrayList` of labeled points returned by the k-nearest neighbor procedure. Let c_i be the center point (the first point in L_i). Let r_i be the squared distance to the farthest point (the kth point of L_i). Insert the key-value pair (r_i, L_i) in your leftist heap. (There is not need to perform a deep copy. Just copy the reference to L_i.)

`ArrayList<LPoint> extractCluster()`: **(Updated 12/05)** This performs one step of the greedy algorithm. First, if the kd-tree is empty, return `null` as a signal that there are no more clusters. Otherwise, repeat the following steps until we are successful in finding a cluster. Extract the next cluster (r_i, L_i) from your priority queue (the leftist heap). Using the kd-tree `find` operation, check whether every point of L_i is still in the kd-tree. If so, we have successfully found a cluster, and otherwise we haven't. Here is how to process each of these cases:

Success: Delete all the points of L_i from the kd-tree, and return L_i as the answer.

Failure: Let c_i be the first point of L_i (the service center). If c_i is still in the kd-tree, compute a new list L_i' of its k-nearest neighbors, let r_i' be its new radius, and add the pair (r_i', L_i') back into the leftist heap. (Note that c_i will still be the first element of L_i', so we have effectively replaced an old damaged cluster for c_i with a new one.) Otherwise, (c_i is not in the kd-tree) do nothing. In either case, continue extracting clusters from the priority queue until we succeed.

Note that the leftist heap will throw an `Exception` if you attempt to extract when there are no more elements left. In theory, this should not happen, since if you still have points in your kd-tree, you should still have clusters in your leftist heap. Nonetheless, you will need to create a `try-catch` block to keep the compiler happy, but if you ever reach the `catch` section, there is something wrong in your program.

`ArrayList<String> listKdTree()`: This just invokes the `list` operation on your kd-tree tree (for debugging).

`ArrayList<String> listHeap()`: This just invokes the `list` operation on your leftist heap tree (for debugging).

More Information

Skeleton Code: As usual, we will provide skeleton code on the class Projects Page. You will need to fill in the implementation of the `KCapFL.java`, `XkdTree.java`, `LeftistHeap.java`, and `MinK.java`. As before, we will provide `Point2D`, `Rectangle2D`), and so on. We will also provide the testing programs, `Part3Tester.java` and `Part3CommandHandler.java`. As with the previous assignment, the package "`cmsc420_f22`" is required for all your source files.

What if I cannot finish? Give priority to working on point deletion and k-nearest neighbors. If you do just those, you will get roughly 50% credit. The remainder will be for the `KCapFL` functionality and the usual efficiency/style points. But, don't be intimidated by the lengthy description. The lengthy description above is almost a line-for-line presentation of the `KCapFL` class. My class was only about 1/5 as long as my leftist heap and kd-tree implementations.

What if my kd-tree/leftist heap didn't work? We'll make minimal versions of these programs available to you. You will still need to implement deletion and k-nearest neighbors.

Efficiency requirements: (20% of the final grade) 10% for efficiently implementing k-NN and 10% for efficiently implementing `MinK`.

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 need to submit `KCapFL.java`, `XkdTree.java`, `LeftistHeap.java`, and `MinK.java` files. If you created any additional files for utility objects, you will need to upload those as well.