Programming Assignment #3 CS671

1st submission due 7/2/2021
2nd submission due 7/9/2021

This assignment's focus is on recursion on non linear structures (like trees) and higher-order functions. The objective is to implement a spatial data structure known as quadtree and to make it support a few higher-order functions.

Quadtrees

A quadtree is a hierarchical data structure that splits a two-dimensional space into four quadrants (top-left, top-right, bottom-left and bottom-right) and recursively splits each subspace in the same manner. (3D spaces use *octrees* instead.) The idea is to speed up searches by relying on the spatial properties of a target in order to ignore quadrants that cannot possibly contain it.

The quadtree to implement in this assignment is based on the following ideas:

- Each node in the tree represents a *space*. The root of the tree represents the entire space.
- There are two types of nodes:
  - *internal nodes*, with exactly 4 children. Each child represents a quadrant of the space;
  - *leaves*, with no children. Leaves contain the actual data, i.e., a collection of points in the space.
- The structure of a tree is based on a *threshold* or *bucket size*. A leaf cannot contain more elements than the threshold. If a space is to contain more points, it has to be represented by an internal node. A tree in which:
  - leaves have at most the threshold number of elements, and
  - internal nodes are used only for subtrees with more than the threshold

  are said to be in *canonical form*. Quadtrees should always be left in canonical form after an operation is completed.

Figure 1 shows a quadtree of threshold 2 with 6 points *A*, *B*, *C*, *D*, *E* and *F*. The space represented by that tree is the rectangle (0, 0, 15, 10), that is a rectangle that can contain points (*x, y*) where $0 \le x \le 15$ and $0 \le y \le 10$. Because the tree contains more points than the threshold, it cannot be implemented as a single leaf node. Instead, its space is split into four quadrants and the root of the tree has four children **TL**, **TR**, **BL** and **BR**, which are themselves quadtrees. Tree **TL** represents the space (0, 6, 7, 10), tree **TR** represents the space (8, 6, 15, 10), etc.

Tree **TL** is empty (no points) and is implemented as a leaf node with an empty collection. Trees **BL** and **TR** contain one and two points, respectively, and are also represented as leaves. Tree **BR**, on the other hand, contains three points, which is more than the threshold. It is therefore implemented as an internal node with four children. The top-left quadrant of **BR**, denoted as **BR**/**TL**, represents the space (8, 3, 11, 5) and is implemented as a leaf node with a collection of two elements: *D* and *E*. Similarly, **BR**/**BL** is a leaf that contains point *F* and **BR**/**TR** and **BR**/**BR** are both empty leaves.

Spatial queries can benefit from the structure of a quadtree. For example, consider the problem of finding the point closest to coordinates (5, 4) (this is akin to a finding the closest gas station in a map software). A search can be initiated in the quadrant that contains (5, 4). This quadrant is a leaf and its closest point to the target coordinates is *C* at (6, 4), which is at distance 1 from the target. At this point, we know that any closer point, if any, *must* be in the gray area surrounding the target. Since this area

does not overlap with the spaces represented by the other three trees (**TL**, **TR** and **BR**), it is not necessary to search them and *C* can be immediately returned as the closest point. (Sample test "`getNearest (1)`" implements this scenario and checks that points *A*, *B*, *D*, *E* and *F* were not used in the search.)

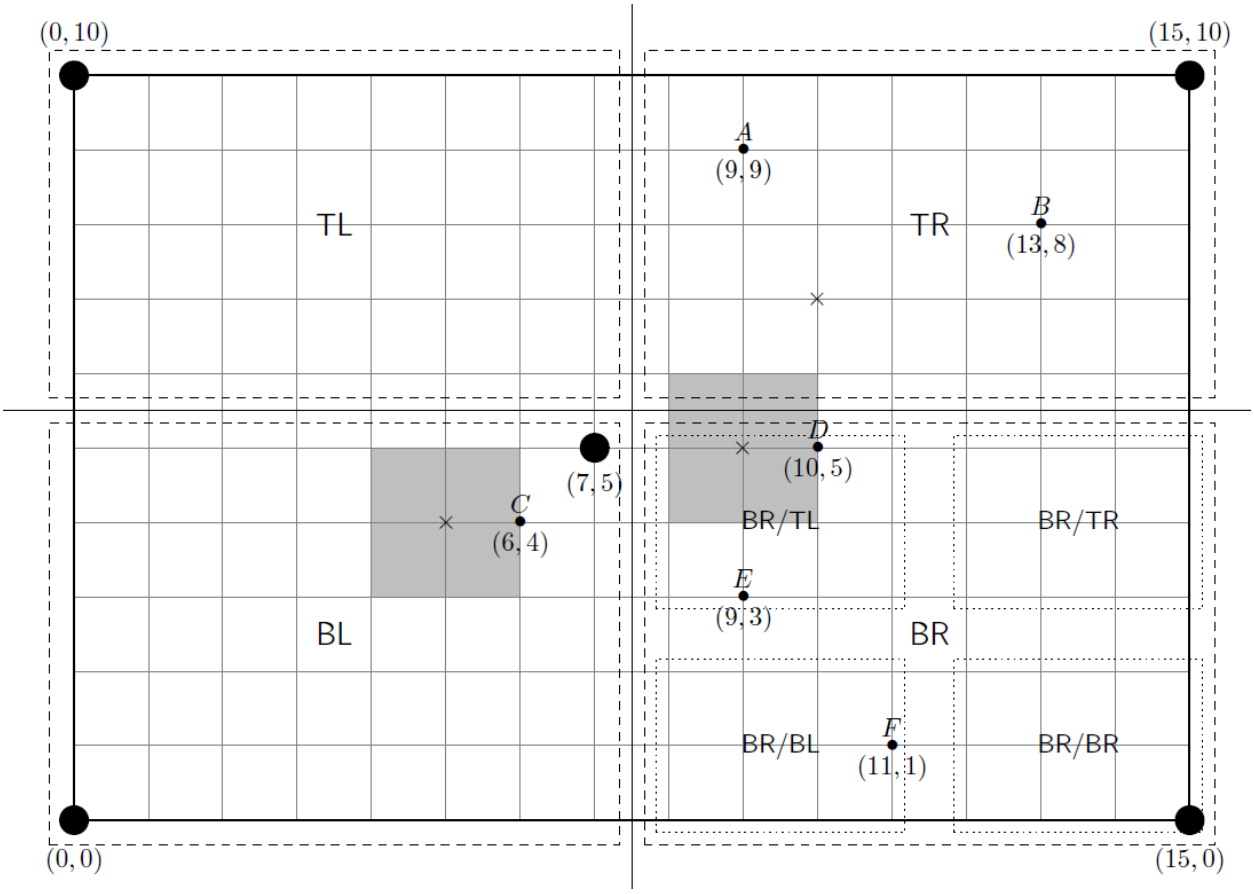

Figure 1: A quadtree with 6 points and a threshold of 2.

Note, however, that the closest point to a target need not be in the same quadrant as the target. To illustrate this scenario, consider the problem of finding the point closest to coordinates (10, 7). A search is initiated in the **TR** quadrant and returns *A* as the closest point, with a distance of $\sqrt{5} \approx 2.24$. Point *D*, however, is closer at a distance of 2. The search continues because the acceptable area around the target (not represented in the figure) overlaps with the other children, which are then explored until *D* is found. (This scenario is implemented as test "`getNearest (2)`".)

As a third example, consider a search with target (9, 5). The initial search takes place in quadrant **BR**/**TL** and finds *D*, which is indeed the closest point. The area of possible closer points, however, still overlaps with quadrant **TR**, which needs to be explored as well, although this second search will not produce a better candidate than *D*. Even in this case, several trees (**TL**, **BL**, **BR**/**TR**, **BR**/**BL**, **BR**/**BR**) can safely be ignored during the process and points *C* and *F* are not considered at all. (This scenario is implemented as test "`getNearest (3)`".)

To summarize, the search algorithm in a quadtree works as follows:

- Define a space *S* for the search to take place in. Initially, *S* is the entire space represented by the tree.

- On a leaf node, simply enumerate all the candidates (at most as many as the threshold).
- On an internal node:
    - Initiate a search in the quadrant that contains the target (or the quadrant closest to the target if the target is not in the space covered by the tree).
    - Use the results of this search to try to reduce space *S*. It is possible that *S* stays the same but, as seen in the examples above, *S* can also become much smaller.
    - Begin a search in another quadrant *that overlaps with S*. Quadrants that don't overlap with *S* can be ignored. Use the results from this search to possibly further reduce *S* before exploring other quadrants. There lies the power of the quadtree (as opposed to a non-spatial data structure).
    - return the best result of all searches (at most four searches take place in an inside node).

The algorithm is recursive, but in many cases, does not need to explore all the children of a node.

The `QuadTree` Class

In this assignment, quadtrees are implemented as instances of the `QuadTree` class. Those instances are immutable. The class is parameterized by two types: `Loc`, which represents locations, used as keys, and `A`, which represents the tree's content. Type `Loc` is required to be a subtype of Location, a generic type that represents 2D coordinates. As a special case, if type `A` is itself a subtype of `Location`—for instance, `A` is a point—values can be used directly as keys and the tree represents a set. The companion object of the `QuadTree` class offers methods to create trees for either scenario.

Internally, a quadtree is implemented in terms of two classes: `QuadNode` (an internal node) and `QuadLeaf` (a leaf). Case classes are used so quadtrees can be processed through pattern-matching. Both classes keep track of the threshold and the space covered by a tree or subtree. The root's space is the space for the entire tree. Other nodes have smaller spaces that represent quadrants and sub-quadrants in the area of interest. The threshold value is the same for all the nodes (i.e., every tree has a given threshold but different trees can have different thresholds).

In addition to a threshold and a space, `QuadLeaf` instances contain a collection of mappings `(Loc -> A)`, which are stored as a regular immutable map. The size of this map is bounded by the threshold. `QuadNode` instances, on the other hand, contain references to four children of type `QuadTree`, named `tl`, `tr`, `bl` and `br`. These children can be instances of `QuadNode` or `QuadLeaf`.

The QuadTree class implements the following public methods:

- `space`: the space (whole area) represented by a tree,
- `size`: the number of elements in a tree.
- `isEmpty`: true iff a tree contains no element.
- `get(Loc)`: retrieves an element by location. For an internal node, the algorithm is similar to that of a binary search tree (except that the tree is quaternary): find the quadrant that might contain the target (if none, the target is outside the tree space and cannot possibly be found) and recursively search in it (only one quadrant at the most is searched at each level). For a leaf, it is a lookup in the leaf's map.
- `toMap`: can be used to dump the contents of tree into a (regular) map. Note that this is just a way to extract data from a tree, not to (inefficiently) implement the search operations.

- `add`: adds a value to a tree. Locations that are outside the global space of the tree cannot be added and are rejected with an exception. Because the tree is immutable, this produces a new tree. The algorithm works by adding an element to the correct leaf, i.e., the leaf whose space contains the coordinates to add. However, if this leaf already contains a number of values equal to the threshold, a new internal node needs to be created to replace the leaf node, whose values are split among the four new quadrants.
- `remove`: removes a value from a tree. The tree is left unchanged if it does not contain the value (including values outside the tree space). Otherwise, a new tree is created, since trees are immutable. As in the case of method `add`, there is a special situation when an internal node is left with a number of elements that is equal to the threshold: In that case, the internal node should be replaced with a leaf node.
- `clearSpace`: removes all the elements contained in a space. Because of the structure of a quadtree, subtrees that do not intersect the given space are unchanged and do not need to be traversed. As in the case of method `remove`, internal nodes left with as many or fewer elements than the threshold need to be replaced with leaves.
- `get(Space)`: retrieves all the elements contained in a given space. Because of the structure of a quadtree, subtrees that do not intersect the given space do not need to be traversed.
- `getWithin`: retrieves all the elements contained in a circle. Because of the structure of a quadtree, subtrees that do not intersect the bounding box of the circle do not need to be traversed.
- `getNearest`: finds the element closest to a target. An optional filter can be used to require this element to satisfy a given property.[1] The target may or may not belong to the global space of the tree. This method throws a `NoSuchElementException` if no element in the tree satisfies the condition.
  Note that in order to implement `getNearest` efficiently (i.e., without exploring the entire tree), a space *S* needs to be passed to a search, but the `getNearest` method has no such argument. A common approach to deal with this situation is to introduce a non-public method (say, `getNearest0`) that takes the *S* argument, and then implement `getNearest` in terms of `getNearest0` (`getNearest` calls getNearest0 with the entire space initially).
- `map(f)`: a higher-order function that applies `f` to all the elements of a tree and returns a tree of the results. Both trees have the same structure and contain the same locations. In other words, function `f` can modify an element (e.g., add a color to a point), but cannot move it (the element's location remains the same).[2]
- `filter(f)`: a higher-order function that applies `f` to all the elements of a tree and returns a tree containing only the elements for which `f` returned true. The locations of these elements are unchanged. Like methods `remove` and `clearSpace`, this method produces trees in canonical form.

[1] `tree.getNearest(loc,Some(f))` is functionally equivalent to `tree.filter(f).getNearest(loc,None)` but is more efficient because it does not need to build a new filtered tree.

[2] This function is intended to be used on trees created as maps from `Location` to `A`. On a tree created as a set of `A`, it has the strange effect of replacing a mapping `(x -> x)` into a mapping `(x -> f(x))`, thus turning the set into a more general map.

- `foreach(f)`: a higher-order function that applies `f` to all the elements of a tree. The function is applied for the sake of side effects. The order in which the function is applied is unspecified. No value is returned.

The `Space` Class

This class implements the notion of "space" and offers methods that can be used when implementing class `QuadTree`:

- `contains`: tests whether a location is within the space. The boundaries of a space are considered within that space (i.e., `Space(0,0,1,1)` contains (0, 0) and (1, 1)).
- `center`: the center of a space. Because spaces are specified using integers, this point cannot always be exactly in the center of the space. Non-integers numbers should be rounded down, placing this point left and below of the exact center.
- `intersects`: whether a space overlaps with another space in at least one point.
- `split`: splits a space into four non-overlapping quadrants. The center of the space is the top-right corner of the bottom-left quadrant.
- `Space.bounds`: calculates the bounding box of a collection of points (i.e., the smallest space that contains all the points).

The `CityApp` Application

The `CityApp` class uses `QuadTree` in a simple application. The application uses the latitude, longitude, population and elevation of 16196 US cities to build a location map and uses the map for a few simple queries: closest city to Kingsbury Hall (Durham), closest city to Kingsbury Hall with a population of 15000 or more (Dover), closest Massachusetts city to Kingsbury Hall (Amesbury), closest US city to the Eiffel Tower (Van Buren, ME) and a list of cities within 10km of Kingsbury Hall (Barrington, Dover, Durham, Lee, Madbury, Newmarket and Stratham Station).

```
> runMain city.CityApp 128
map constructed (16196 elements)
City(Durham,NH,10345,Some(15),43.13397,-70.92645)
City(Van Buren,ME,1937,Some(147),47.15727,-67.9353)
City(Dover,NH,29987,Some(19),43.19786,-70.87367)
City(Amesbury,MA,18313,Some(17),42.85842,-70.93005)
List(Barrington, Dover, Durham, Lee, Madbury, Newmarket, Stratham Station)
```

Figure 2: `CityApp` run.

For this application, the *x* and *y* coordinates of points are longitudes and latitudes on a sphere. This makes the implementation of distances and bounding boxes more complicated (methods `distanceTo` and `around` in `Location`), but the principle remains the same and a quadtree structure can still be used. The application takes a threshold value as a command-line argument (see Figure 2.)

Notes

- There are many variants of quadtrees as well as more efficient spatial data structures. This assignment uses a fairly straightforward implementation for the sake of clarity. A real library could use fancier code.
- The implementation of `EuclideanLocation` and `LatLonLocation` are given and need no modification.

- The implementation of trait `Space` cannot be modified. Instances are created through the companion object, which needs to be implemented. This will require a class that extends trait `Space`. Using a case class helps deal with equality comparison. This class should not be public and should be implemented first and thoroughly tested, making it easier to implement `QuadTree` without having to solve bugs related to `contains` or `split` on spaces.
- In languages like Java and Scala, parametrized (generic) data structures involve tricky type problems. For instance, if `ColoredPoint` is a subtype of `Point`, `Map[Loc,ColoredPoint]` is a subtype of `Map[Loc,Point]` in Scala but `Map<Loc,ColoredPoint>` is *not* a subtype of `Map<Loc,Point>` in Java. Because these issues have not been discussed in class yet, `LocationMap[Loc,ColoredPoint]` has not been made a subtype of `LocationMap[Loc,Point]` even though it should be.
- By the same token, scary-looking syntax like `Loc <: Location[Loc]` has not been covered yet. It shouldn't get in the way and can be though of as simply a type `Loc` with methods `x`, `y`, `Coordinates`, `distanceTo` and `around`.
- In a real library, classes `QuadNode` and `QuadLeaf` would be private, as they expose too much of the underlying implementation and allow users to build trees in non-canonical form. They are left public for the purpose of testing and grading.
- The algorithm for `QuadTree.getNearest` specifies that the search should start with the quadrant that contains the target (or the closest quadrant if the target is outside the space). Some grading tests will fail if the search starts with another quadrant (see the discussion of the `getNearest` sample tests above). The order in which the 3 remaining quadrants are processed is not specified.
- If you see the documentation generated by Scaladoc, it shows some methods of class `QuadTree` to be abstract and others to be concrete. This is specific to the reference implementation. Other implementations could have more or fewer concrete methods in `QuadTree`. The starting code has all the methods abstract in `QuadTree`.
- Helper methods like `getNearest0` should not be public. They can be made `private[quadtree]` to be visible only with the `quadtree` package (same as not using any modifier in Java).
- The order is which the higher-order functions of `QuadTree` apply their parameter `f` is not specified (and only matters if `f` has side effects).
- The `CityApp` object is already implemented and does not require any modification. The `City` class and object are also given. The code to be written is in the `CityApp` class.