Project: Implementing a KdTree for Efficient 2D Searches

Overview

You are to implement a mutable data structure called a KdTree (or 2d–tree) that efficiently stores a collection of 2D points located in the unit square ([0, 1] times [0, 1]). The KdTree supports efficient range searches (to find all points within a given rectangle) and nearest-neighbor searches (to find the point closest to a query point). You will use two provided geometric primitives:

- **Point2D**: Represents a point in the 2D plane with x and y coordinates.
- **RectHV**: Represents an axis-aligned rectangle (defined by its minimum and maximum x- and y-coordinates).

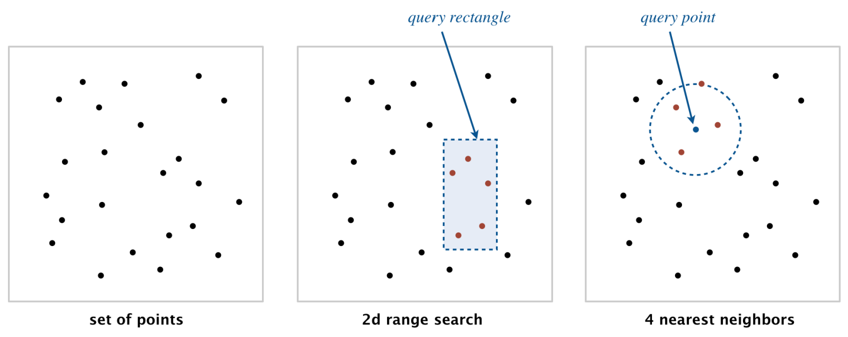
Your task is to implement the KdTree functionality in a language-agnostic manner so that the solution could be written in Java, Python, or any other language.

Geometric Primitives (Provided)

1. **Point2D**
   - Represents a point with x and y coordinates.
   - Provides methods to access coordinates, compute distances (Euclidean and squared), and perform equality comparisons.
2. **RectHV**
   - Represents an axis-aligned rectangle defined by minimum and maximum x- and y-coordinates.
   - Provides methods to:
     - Check if a point lies inside the rectangle.
     - Determine if two rectangles intersect.
     - Compute the distance (and squared distance) from a point to the rectangle.

Working Idea

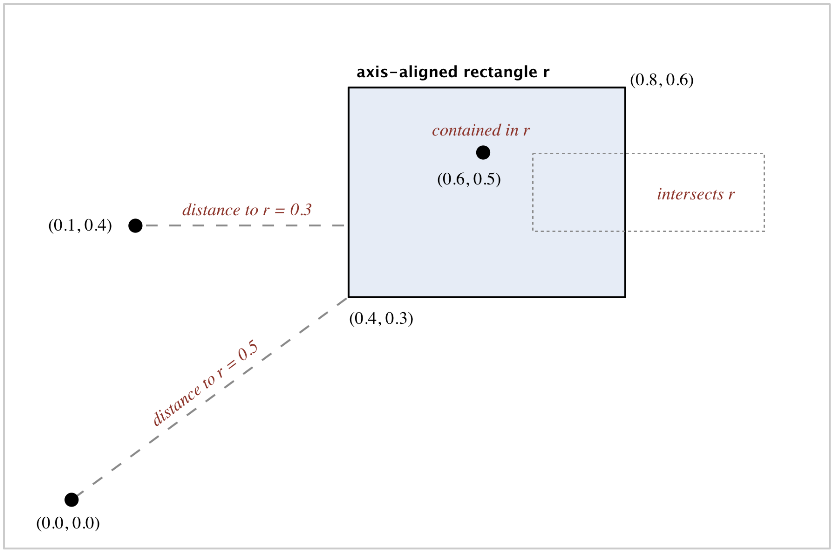
1. **Figure 1: Range Search and Nearest Neighbors**



   This figure (left to right) illustrates:

- A set of points in the 2D plane (all black dots).
- A rectangular query (dashed rectangle) used for **range search**, highlighting the points (in red) that lie inside the query rectangle.
- A circular region around a query point (dashed circle) illustrating the **nearest neighbors** (in red) of that query point (here, the 4 closest points are highlighted).

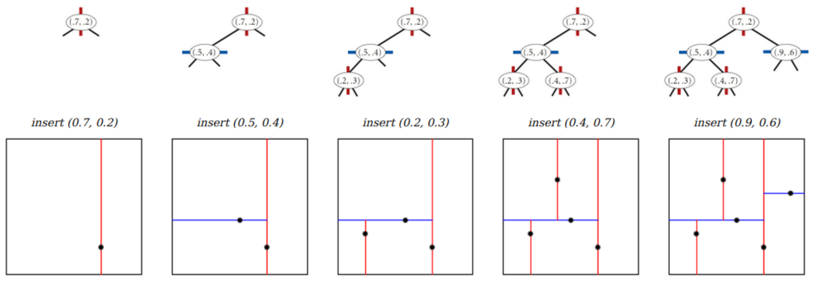
2. **Figure 2: Axis-Aligned Rectangle Operations**



This figure shows:

- A rectangle `r` defined by its minimum and maximum x- and y-coordinates, e.g. ((0.4, 0.3)) to ((0.8, 0.6)).
- How to check if a point is **contained** in `r` (e.g., the point ((0.6, 0.5))).
- How to determine if a rectangle **intersects** another rectangle (shown in dotted lines).
- How to compute the **distance** from a point to a rectangle (e.g., distance from ((0.1, 0.4)) to `r` is (0.3), and from ((0.0, 0.0)) to `r` is (0.5)).

3. **Figure 3: Kd-Tree Insertion Steps**



This figure shows an example of constructing a kd-tree step by step:

- Each node in the tree alternates between splitting on the x-coordinate (red vertical lines) and the y-coordinate (blue horizontal lines).
- In the first insertion (inserting ((0.7, 0.2))), the root splits the entire unit square vertically at (x = 0.7).
- In the second insertion (((0.5, 0.4))), the next level splits horizontally at (y = 0.4), and so on.
- The diagram at each step shows both the tree structure (top) and how the plane is partitioned (bottom).

KdTree Requirements

Implement the following methods for the KdTree. Each method's functionality is described in plain English:

1. **Constructor: `KdTree()`**
    - **Purpose:** Create an empty kd–tree.
    - **Behavior:** Initialize an empty data structure with no points.
2. **`isEmpty()` Method**
    - **Purpose:** Check if the kd–tree has no points.
    - **Returns:** A boolean value — `true` if the tree is empty, `false` otherwise.
3. **`size()` Method**
    - **Purpose:** Report the number of points currently stored in the kd–tree.
    - **Returns:** An integer representing the total count of points.
4. **`insert(Point2D p)` Method**
    - **Purpose:** Add a new point (p) into the kd–tree.
    - **Behavior:**
        - If the tree does not already contain (p), insert it.
        - Use alternating comparisons at each level:
            * At the root, compare by the x-coordinate (vertical split).
            * At the next level, compare by the y-coordinate (horizontal split).
            * Continue alternating at each subsequent level.
        - Define the region (rectangle) for each node based on the splitting line.
    - **Edge Cases:**
        - If (p) is `null` (or an equivalent), throw an exception or error.
5. **`contains(Point2D p)` Method**
    - **Purpose:** Determine whether a given point (p) is already in the kd–tree.
    - **Returns:** A boolean value — `true` if the point exists in the tree, `false` otherwise.
    - **Behavior:**
        - Traverse the tree by comparing coordinates (alternating between x and y) until either the point is found or a leaf is reached.

- **Edge Cases:**
    - If (p) is `null`, throw an exception or error.

6. **`range(RectHV rect)` Method**
    - **Purpose:** Find and return all points that lie within a given query rectangle.
    - **Returns:** A collection (e.g., list, iterable) of all points inside (or on the boundary of) the rectangle.
    - **Behavior:**
        - Recursively search the kd–tree.
        - Use the rectangle associated with each node to decide whether to search its subtree:
            * If the query rectangle does not intersect the node's rectangle, prune that subtree.
            * Otherwise, if the node's point is inside the query rectangle, include it in the result.
    - **Edge Cases:**
        - If the query rectangle is `null`, throw an exception or error.
7. **`nearest(Point2D p)` Method**
    - **Purpose:** Find the point in the kd–tree that is closest to the given query point (p).
    - **Returns:** The nearest point to (p) or `null` if the kd–tree is empty.
    - **Behavior:**
        - Recursively search the tree and keep track of the current best (closest) point.
        - Use the distance from the query point to the rectangle of each node as a pruning rule:
            * If the node's rectangle is farther than the best distance found so far, do not search that subtree.
        - Always search the subtree that is on the same side of the splitting line as the query point first to potentially update the best distance.
    - **Edge Cases:**
        - If (p) is `null`, throw an exception or error.

Performance Considerations

- **Insertion and Search:** The kd–tree should support efficient insertion and search operations, ideally in logarithmic time on average.
- **Range and Nearest Queries:** The recursive search should utilize pruning to avoid examining nodes that cannot contain a valid candidate, thus speeding up the queries.

Testing Guidelines

Your implementation should include tests (or a main routine) that demonstrate the following: - **Insertion:** Read a set of points (for example, from user input or a file) and insert them into the kd–tree. - **Size:** Verify the total number of points stored. - **Range Query:** Execute a range query on a given rectangle and print or otherwise output all points found inside the rectangle. - **Nearest Neighbor:** Execute a nearest-neighbor query for a given point and output the closest point found.

Submission

Submit your source code files for the KdTree implementation along with any necessary documentation. The provided geometric primitives (Point2D and RectHV) are assumed to be available, so you do not need to implement them. Ensure your code is well-documented and each method is clearly commented with its purpose and behavior.