# CS603 Project

Hanvitha(210050047)
Yashaswi(210050127)

## Part A:

### Problem Specification

**Input:** A static set of $n$ points in $\mathbb{R}^2$, followed by a sequence of axis-aligned rectangular range queries.

**Output:** For each query rectangle, report all points from $P$ that lie inside it.

**Time Complexity:**

- Preprocessing time: $O(n \log n)$
- Query time: $O(\sqrt{n} + k)$, where $k$ is the number of reported points.

### Approach

#### 1. Tree Node Structure

Each node in the KD-Tree is represented by the `KDNode` class. A node can be either:

- An internal node: stores the splitting axis and the split value, and has pointers to its left and right subtrees.
- A leaf node: stores a single point from the input set.

#### 2. Tree Construction Using Median and Sorted Lists

The tree is constructed recursively using two sorted lists:

- $px$: points sorted by x-coordinate.
- $py$: points sorted by y-coordinate.

At each recursive level:

1. Determine the axis of division based on the depth:
    - If depth is even, split along the **x-axis**.
    - If depth is odd, split along the **y-axis**.

2. Choose the median point from the sorted list corresponding to the splitting axis:
   - Use the midpoint of $px$ or $py$, depending on the axis.
   - This ensures the splitting step is done in $O(1)$ time.
3. Partition both $px$ and $py$ based on the split coordinate to form:
   - A left subset containing all points strictly less than the median on the current axis.
   - A right subset containing all points greater than or equal to the median (including the median itself).
4. During this partitioning step, **the original relative order of points in the sorted lists is preserved**, which ensures stability and supports the recurrence structure needed for efficient construction.
5. The median point defines a hyperplane and is always placed in the **right subtree**.
   - If the split is along the x-axis (vertical), then points with $x \geq \text{median}_x$ go to the right.
   - If the split is along the y-axis (horizontal), then points with $y \geq \text{median}_y$ go to the right.
6. Update the region (bounding box) for left and right subtrees based on the split.
7. Recursively construct the left and right subtrees using the corresponding point subsets.

This method ensures both efficient median computation and spatial division, allowing construction of the KD-tree in $O(n \log n)$ time using $O(n)$ space.

### 3. Range Search

To process axis-aligned rectangular range queries:

- If a subtree's region is completely inside the query rectangle, all its points are reported immediately.
- If a subtree's region intersects the query rectangle, its children are explored recursively.
- If a subtree's region lies completely outside the query rectangle, it is skipped.

## User Interaction

- The user is prompted to input the number of 2D points, followed by coordinates for each point.
- Then, the user can input multiple rectangular queries.
- Each query is specified by the coordinates `xmin xmax ymin ymax`.
- The query returns all points that lie within the rectangle.

### Performance Analysis

- Construction time: $O(n \log n)$, where $n$ is the number of points.
- Range search time: $O(\sqrt{n} + k)$, where $k$ is the number of points in the query result.

### Function-Wise Implementation Details

**Overview:** We implemented the KD-tree with points stored at the leaves. Each internal node splits space along an axis-aligned hyperplane. The following functions were used:

- `class KDNode`: Defines a node of the KD-tree. It stores the splitting axis, split value, left and right children, a point (for leaf nodes), and the bounding rectangular region.
- `build_kd_tree(px, py, depth=0, region=...)`: Recursively builds the KD-tree by alternating between the $x$ and $y$ axes at each level. Points are partitioned based on the median of the corresponding axis. Leaf nodes contain exactly one point.
- `in_rectangle(point, rect)`: Checks whether a given point lies inside a given query rectangle.
- `region_inside(region, rect)`: Checks if a node's region is completely contained inside a query rectangle.
- `region_overlap(region, rect)`: Checks whether a node's region overlaps with a query rectangle.
- `report_subtree(node, result)`: Collects all points from a subtree and appends them to the result list.
- `search_kd_tree(node, rect, result)`: Performs a range query by recursively exploring only necessary branches based on region overlap or containment.
- `print_kd_tree(node, depth=0)`: Prints the KD-tree structure for visualization and debugging.

**Note:** The KD-tree was built using two sorted lists to ensure $O(n \log n)$ construction time, and region boundaries were maintained explicitly at each node for efficient querying.

## Part B:

### Problem Specification

Extend your implementation to handle dynamic updates. At each step, a point may be inserted or deleted.
**Operations**: Insert a point, delete a point, and query with an axis-aligned rectangle.
**Output**: For each query, report all points in the current set that lie inside the query rectangle.

## Insertion Algorithm

Insertion in a kd-tree works exactly like a binary tree with the adjustment that we follow the branch according to the splitting coordinate for each node and we place the node as a leaf in the correct position with regards to the correct splitting coordinate at the end.

### Example

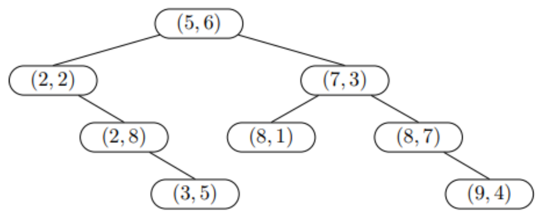


Figure 1: Given a kdtree need to insert point (9,2)

We start at the root and compare xcoordinates and go right. At (7, 3) we compare y-coordinates and go left. At (8, 1) we compare x-coordinates and insert right. Notice that the inserted node has its split direction set appropriately for the level it is on. **Insertion**

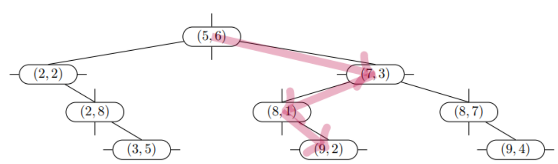


Figure 2: kdtree after insertion

**Complexity:** The time complexity of insertion depends on the height of the KD-Tree. In the best case, when the tree is balanced, it is $O(\log n)$. In the average case, due to partial imbalance, it is $O(\sqrt{n})$, and in the worst case, when the tree becomes a linked list, it is $O(n)$.

## Deletion Algorithm

Our deletion procedure is recursive. Assume the node to be deleted is $u$:

- **If $u$ is a leaf:** Just delete it, and we are done.
- **If $u$ has a right subtree:** Let $\alpha$ be the coordinate that $u$ splits on. Find a replacement node $r$ in $u$.`right` for which $r_\alpha$ is minimal. Copy $r$'s point to $u$ (overwriting $u$'s point). Then recursively call delete on the old $r$. Since $r$ has the minimal $\alpha$-coordinate in $u$.`right`, all nodes in $u$.`left` have $\alpha$-coordinate less than $r_\alpha$ and all nodes in $u$.`right` have $\alpha$-coordinate greater than or equal to $r_\alpha$, so splitting remains valid.

- **If $u$ does not have a right subtree but has a left subtree:** Let $\alpha$ be the coordinate that $u$ splits on. Find a replacement node $r$ in $u$.`left` for which $r_\alpha$ is minimal. Copy $r$'s point to $u$ (overwriting $u$'s point). Move $u$.`left` to become $u$.`right`. Then recursively call delete on the old $r$. All nodes in $u$.`left` have $\alpha$-coordinate greater than or equal to $r_\alpha$, so when moved to $u$.`right` they remain correctly positioned. Since this is just a horizontal subtree move, the splitting coordinates in descendants remain consistent.

- With respect to ancestors: for any coordinate $\beta$, if $u$ has an ancestor splitting on $\beta$, then $r$ was already correctly positioned with respect to that ancestor (as $r$ is below it in the tree). Copying $r$ to $u$ preserves this relationship.

- **Deletion Complexity:** Deletion involves finding a replacement point, which can require searching subtrees using `find_min`, potentially costing up to $O(n)$ per operation. Thus, the best case is $O(\log n)$, the average case is $O(n)$, and the worst-case time complexity can go up to $O(n^2)$ due to repeated deep replacements.

## Usage Interface

The program provides a command-line interface where users can:

- Build the tree from a set of points.
- Insert new points.
- Delete existing points.
- Search for points within a rectangular region.

## Function-wise Implementation Details

- `KDNode`
  Defines a node in the KD-Tree, storing the point, splitting axis(0 for x, 1 for y), region, and pointers to left and right children.

- `build_kd_tree(px, py, depth, region)`
  Recursively builds a balanced KD-Tree by choosing the median point along alternating axes and partitioning the space accordingly.

- `in_rectangle(point, rect)`
  Checks whether a given point lies inside a specified rectangular region.

- `region_inside(region, rect)`
  Determines if a node's region is completely contained within a query rectangle.

- `region_overlap(region, rect)`
  Determines if a node's region overlaps partially with a query rectangle.

- `report_subtree(node, result)`
  Traverses the subtree rooted at a node and collects all points into a list.

- `search_kd_tree(node, rect, result)`
  Searches for all points within a given rectangle using pruning based on region containment or overlap.
- `insert_kd_tree(node, point, depth, region)`
  Inserts a new point into the KD-Tree, maintaining correct region boundaries and alternating axes at each depth.
- `delete_kd_tree(node, point, depth)`
  Deletes a point from the KD-Tree by replacing it with a minimum point from the subtree along the splitting axis.
- `find_min(node, axis, depth)`
  Finds the node with the minimum coordinate value along a specified axis within a subtree.
- `print_kd_tree(node, depth)`
  Prints the structure of the KD-Tree in a readable format showing depth, axis, point, and region.