Documentation: 2D Range Tree Project (Static & Dynamic)

Parinay Dewangan (22B1286) | Kaushal Malpure (22B1276)

27/04/2025

1 Part 1: Static 2D Range Tree

1.1 Implementation Details

This part implements a **Static 2D Range Tree** to efficiently perform orthogonal range queries on a set of 2D points. The tree is built once and not modified after construction.

Tree Construction

- Points are recursively divided by their x-coordinate to build a binary tree.
- Each node stores:
  - A list of points sorted by x-coordinate.
  - A list of points sorted by y-coordinate for binary search filtering.
  - A midpoint x_{mid} to split into subtrees.

Query Processing

- Queries use an (x_1, x_2, y_1, y_2) rectangle.
- Nodes fully within x range use binary search on the sorted y-list.
- Partial overlaps trigger recursion into child nodes.

1.2 Techniques Used

- **Divide and Conquer** – Used for building the tree.
- **Binary Search** – Applied to sorted y-coordinate lists.
- **Recursive Tree Traversal** – For both construction and querying.

1.3 Data Structures Used

Structure	Purpose	Reason
List of Tuples	To represent 2D points	Efficient and lightweight storage for point data.
Class-based Tree Node	Tree construction	Enables recursive binary tree formation.
Sorted Lists	For fast y-range filtering	Enables binary search within y-dimension.

Table 1: Data structures for Static Range Tree

1.4 Pseudocode

Build Tree

```
function buildTree(points):
    if len(points) == 1:
        return LeafNode(points)
    sort points by x
    mid = len(points) // 2
    left = buildTree(points[:mid])
    right = buildTree(points[mid:])
    return Node(points, left, right)
```

Query

```
function query(node, x1, x2, y1, y2):
    if node is null:
        return []
    if node.x_range inside [x1, x2]:
        return y_filter(node.y_sorted, y1, y2)
    result = []
    if x1 <= node.x_mid:
        result += query(node.left, x1, x2, y1, y2)
    if x2 >= node.x_mid:
        result += query(node.right, x1, x2, y1, y2)
    return result
```

1.5 Time Complexity Analysis

1.5.1 Build Time

- Sorting points by x-coordinate: $O(n \log n)$.

- At each node, storing y-sorted list across the tree costs $O(n \log n)$.
- **Total Build Time:** $O(n \log n)$.

1.5.2 Query Time

- Tree traversal requires $O(\log n)$ levels.
- Binary search on y-sorted lists at each relevant node costs $O(\log n)$.
- **Total Query Time:** $O(\log^2 n + k)$, where k is the number of points reported.

1.5.3 Space Complexity

- Tree nodes store points and y-sorted lists.
- **Total Space:** $O(n \log n)$.

1.6 Tricky Parts

- Differentiating between full and partial x-range overlap for efficient pruning.
- Manually implementing lower/upper bound binary search functions.

1.7 Challenges and Solutions

- **Query Efficiency** – Precomputed y-sorted lists reduce query time.
- **Edge Cases** – Handled gracefully via recursion base case.

2 Part 2: Dynamic 2D Range Tree

2.1 Implementation Details

This part implements a **Dynamic 2D Range Tree** using a segment tree structure that supports:

- Point insertions and deletions.
- Live range queries with update capability.

Structure

- A segment tree is built over x-coordinate ranges.
- Each node stores:
  - Its x-range.
  - A sorted list of y-values.
  - The actual points in its range.

Operations

- **Insertion**: Traverses down the segment tree and inserts the point in all overlapping nodes, maintaining sorted y-lists.
- **Deletion**: Searches and removes the point from relevant nodes and y-lists.
- **Query**: Recursively collects points in the query rectangle by filtering y-values in each node using binary search.

2.2 Techniques Used

- **Segment Tree over x-axis** – Allows efficient range updates and queries.
- **Binary Search** – Used to maintain and query sorted y-lists.
- **Double Recursion** – One for x-tree navigation, another for point/y-value filtering.

2.3 Data Structures Used

Structure	Purpose	Reason
Segment Tree Nodes	Index x-axis ranges	Enable log-time updates and queries.
Sorted y-lists	Fast range filtering	Necessary for efficient 2D filtering.
Point Storage (List)	Actual point data	Needed for accurate query result generation.

Table 2: Data structures for Dynamic Range Tree

2.4 Pseudocode

Build Segment Tree

```
function buildSegmentTree(l, r):
    if l == r:
        return new SegmentTreeNode(l, r)
    mid = (l + r) // 2
    left = buildSegmentTree(l, mid)
    right = buildSegmentTree(mid + 1, r)
    return new SegmentTreeNode(l, r, left, right)
```

Insert Point

```
function insert(node, x, y):
    if x not in node.range:
        return
    insert y into node.ys (maintain sorted order)
    insert (x, y) into node.points
    insert(node.left, x, y)
    insert(node.right, x, y)
```

Query

```
function query(node, x1, x2, y1, y2, result):
    if node.range is outside [x1, x2]:
        return
    if node.range inside [x1, x2]:
        for y in y_list in [y1, y2]:
            for (px, py) in node.points:
                if py == y and in rectangle:
                    result.append((px, py))
        return
    query(node.left, x1, x2, y1, y2, result)
    query(node.right, x1, x2, y1, y2, result)
```

2.5 Time Complexity Analysis

2.5.1 Build Time

- Building the segment tree structure: $O(x_{max})$ nodes (typically fixed small range).

2.5.2 Insertion Time

- Traverse $O(\log x_{max})$ segment tree nodes.
- Insert y into sorted list: $O(\log n)$ per node.
- **Total Insertion Time:** $O(\log x_{max} \times \log n)$.

2.5.3 Deletion Time

- Traverse $O(\log x_{max})$ nodes.
- Remove y from sorted list: $O(\log n)$.
- **Total Deletion Time:** $O(\log x_{max} \times \log n)$.

2.5.4 Query Time

- Traverse $O(\log x_{max})$ nodes.
- Binary search for y-values: $O(\log n)$ at each node.
- **Total Query Time:** $O(\log x_{max} \times \log n + k)$.

2.5.5 Space Complexity

- Each point is stored in $O(\log x_{max})$ nodes.
- **Total Space:** $O(n \log x_{max})$.

2.6 Tricky Parts

- Keeping y-lists sorted during insertion and deletion with custom binary insertion/removal logic.
- Ensuring query correctness after dynamic changes to the data.
- Avoiding duplication or omission in multi-level segment tree traversal.

2.7 Challenges and Solutions

- **Dynamic Updates** – Handled with recursive tree traversal and updates at each node.
- **Efficient Deletion** – Manual binary search logic ensures sorted array integrity.
- **UI Integration** – A Tkinter-based interface was added for interactive testing and visualization (not the focus of algorithmic explanation).

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*. MIT Press, 3rd Edition, 2009.

[2] Wikipedia contributors. "Range tree." *Wikipedia, The Free Encyclopedia*, `https://en.wikipedia.org/wiki/Range_tree`

[3] CP Algorithms. "Segment Tree." `https://cp-algorithms.com/data_structures/segment_tree.html`

[4] GeeksForGeeks. "2D Range Queries using Segment Tree." `https://www.geeksforgeeks.org/2d-range-minimum-query-rmq-using-segment-tree/`

[5] Python Docs. "Tkinter — Python interface to Tcl/Tk." `https://docs.python.org/3/library/tkinter.html`