Orthogonal Range Searching

Partha Bhowmick
https://cse.iitkgp.ac.in/~pb

1 Introduction

Orthogonal Range Searching is a fundamental problem in computational geometry with widespread applications in various fields, including databases, artificial intelligence, and computer vision. It enables efficient querying of multidimensional data by allowing retrieval of all points within a given range along each dimension. This capability is crucial for handling large-scale datasets where traditional linear searches become computationally expensive.

In databases, orthogonal range searching facilitates efficient range queries on indexed data, significantly improving query performance in applications such as geographic information systems (GIS), financial databases, and data warehousing. It ensures quick retrieval of records that satisfy specific conditions without scanning the entire dataset (e.g., finding all employees with ages in a given range and salaries in a given range).

In artificial intelligence and machine learning, it is employed for efficient nearest neighbor searches, feature selection, and spatial partitioning. Many machine learning algorithms, such as k-nearest neighbors (k-NN) and clustering techniques, rely on range searching to filter relevant data points effectively. It also plays a key role in decision tree-based models, including random forests, where efficient region-based queries enhance performance.

In computer vision, orthogonal range searching aids in object detection, image retrieval, and spatial indexing. By quickly locating pixels, feature points, or objects within specified regions, it enhances the efficiency of real-time vision-based applications.

The efficiency of orthogonal range searching is achieved through specialized data structures such as kd-trees, range trees, fractional cascading, interval trees, and segment trees, which provide logarithmic query times. These structures are instrumental in optimizing search operations in high-dimensional spaces, making them invaluable in large-scale applications where rapid data retrieval is essential.

2 1D Range Tree

Let $P := \{p_1, \ldots, p_n\}$ be a set of n real-valued elements, which can be viewed as n points on the x-axis. Given a query interval $[a, b]$, **1D range searching** aims to find all elements in $P \cap [a, b]$. A sorted linear array allows efficient querying, but this approach neither generalizes well to higher dimensions nor supports efficient updates to P.

A more generalized way is to use a **1D range tree** T defined as follows.

1. T is a height-balanced binary search tree.
2. $T(\text{left}(v)) = \{x \in P : x \leq x_v\}$, $T(\text{right}(v)) = \{x \in P : x > x_v\}$.
3. The leaves store all the elements of P, and each non-leaf node v stores the element of the rightmost node of $T(\text{left}(v))$. The non-leaf nodes are used to split the search to make it efficient.

Here, root(T) = root of T, $T(v)$ = subtree rooted at a node v, left(v) = left child of v, right(v) = right child of v, x_v = element stored at v.

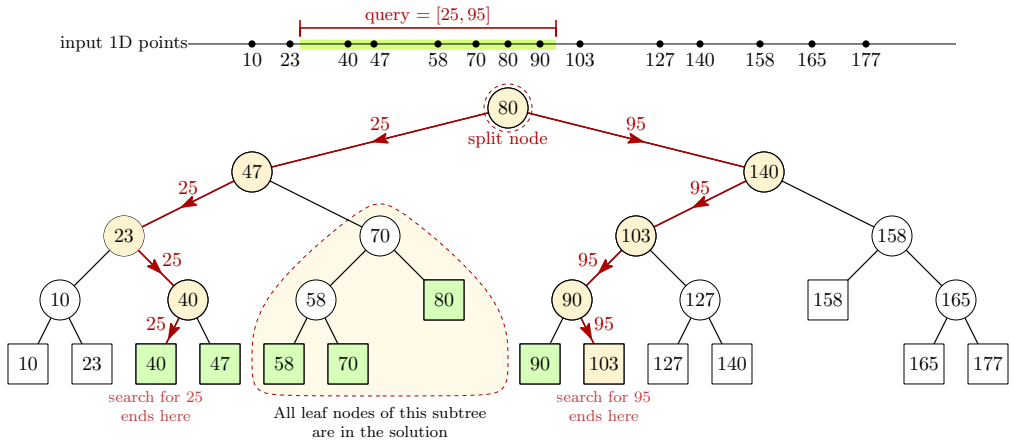

Figure 1: 1D range query—an example. In this example, the root is the split node. Every non-leaf node contains the same key as the rightmost node of its left subtree.

Algorithm 1: Find Split Node

Input: A 1D range tree T and a query interval $[a, b]$.
Output: The node v where the paths to a and b split, or the leaf where both paths end.

```
v ← root(T)
while v ≠ leaf and (b ≤ x_v or a > x_v) do
    if b ≤ x_v then
        v ← left(v)
    else
        v ← right(v)
return v
```

Algorithm 2: 1D Range Query

Input: A 1D range tree T and a query interval $[a, b]$.
Output: All points stored in T that lie in $[a, b]$.

```
v_split ← FindSplitNode(T, a, b)
if v_split is a leaf then
    Check if the point stored at v_split must be reported.
else
    // Follow the path to a and report points in subtrees right of the path.
    v ← left(v_split)
    while v is not a leaf do
        if a ≤ a_v then
            ReportSubtree(right(v))
            v ← left(v)
        else
            v ← right(v)
    Check if the point stored at the leaf v must be reported.
    // Similarly, follow the path to b, report points in subtrees left of the path,
       and check the leaf where the path ends.
```

As described in Algorithm 1 and Algorithm 2, and illustrated in Figure 1, the query with $[a, b]$ on T proceeds as follows.

We search for a and b in T. Let μ_a and μ_b be the leaves where these searches terminate. By the properties of the tree, the solution points are those stored in the leaves between μ_a and μ_b, including, potentially, the points at μ_a and μ_b. To locate these nodes, we first identify the split node v_{split}, where the paths to a and b diverge.

Starting from v_{split}, we traverse the search path of a. At each node v where the path moves left, we report all leaves in $T(\text{right}(v))$, as this subtree lies between the search paths. Similarly, we traverse the search path of b and report the leaves in the left subtree of nodes where the path moves right. This process utilizes the ReportSubtree procedure. Finally, we check if the points stored at μ_a and μ_b lie within $[a, b]$ and report them accordingly.

Construction time and space complexities: Since the number of non-leaf nodes in T is less than its number of leaves,* T requires $O(n)$ storage and has an $O(n \log n)$ construction time. Let k be the total number of reported points. The procedure ReportSubtree traverses the subtree rooted at a given node and reports the points stored at its leaves, taking $O(k)$ time in total. The remaining visited nodes lie along the search path of a or b. As T is balanced, these paths have length $O(\log n)$. Consequently, the query range is processed in $O(k + \log n)$ time.

3 kd-Tree

kd-tree is a height-balanced search tree meant for higher-dimensional search. It stores a set P of d-dimensional points on which a d-dimensional query can be applied. It's not a dynamic tree but used for efficient queries. Let's see how it is constructed and used for $d = 2$ (i.e., in 2D). The input set is $P = \{p_1, \ldots, p_n\}$, where each p_i has coordinates (x_i, y_i). Also assume for simplicity that all points have distinct x-coordinates and distinct y-coordinates.

3.1 Construction of kd-tree

At depth 0 (root), the set P is vertically partitioned into (P_1, P_2) with $|P_1| = \lceil n/2 \rceil$ and $|P_2| = \lfloor n/2 \rfloor$. At depth 1, each of P_1 and P_2 is horizontally partitioned into two nearly equal parts using their respective y-medians. Denote these partitions as (P_{11}, P_{12}) for P_1 and (P_{21}, P_{22}) for P_2. At depth 2, each of these four subsets is further split using their respective x-medians. This process continues in a cyclic manner: even-depth nodes are split using vertical lines, while odd-depth nodes are split using horizontal lines, ensuring balanced partitions at each step.

Each node stores the corresponding splitting line. For example, the root node stores the median x of P, left(root)(T) stores the median y of P_1, and right(root)(T) stores the median y of P_2, and so on. See Algorithm 3 and Figure 2.

*Prove it.

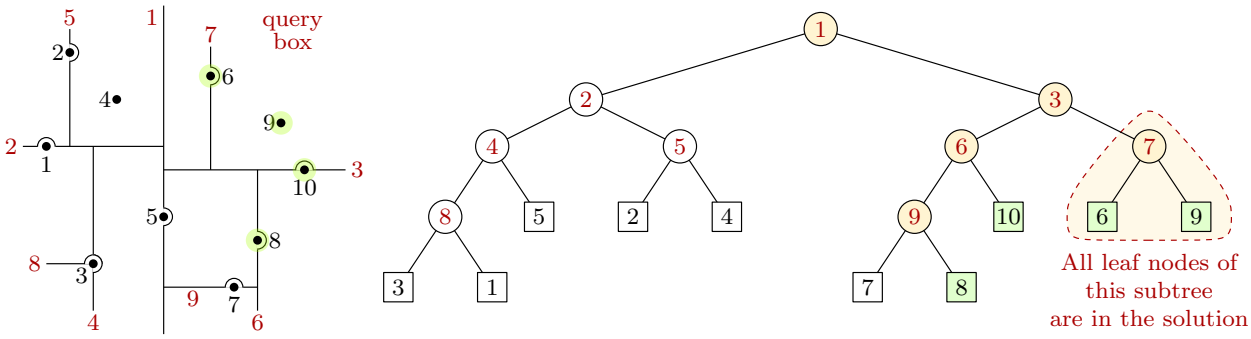

Figure 2: 2D range query in a kd-tree—an example. The labels for splitting lines are colored red and the point IDs are colored black. The leaf nodes in the solution are colored green.

Algorithm 3: BuildKDTree(P, *depth*)

Input: A set of points P and the current depth *depth*.
Output: The root of a kd-tree storing P.

if P *contains only one point* **then**
  return a leaf storing this point
else if *depth is even* **then**
  Partition P into (P_1, P_2) by a vertical line ℓ s.t. $|P_1| = \lceil n/2 \rceil$ and $|P_2| = \lfloor n/2 \rfloor$
else
  Partition P into (P_1, P_2) by a horizontal line ℓ s.t. $|P_1| = \lceil n/2 \rceil$ and $|P_2| = \lfloor n/2 \rfloor$
$v_{\text{left}} \leftarrow$ BuildKDTree$(P_1, \mathit{depth} + 1)$
$v_{\text{right}} \leftarrow$ BuildKDTree$(P_2, \mathit{depth} + 1)$
Create a node v storing ℓ, make v_{left} the left child of v, and make v_{right} the right child of v.
return v

Storage: Each leaf of the kd-tree stores a distinct point of P, resulting in n leaves. As a balanced binary tree, it has $O(n)$ internal nodes. Since each node requires $O(1)$ storage, the total storage requirement is $O(n)$.

Construction time: The most expensive step in each recursive call is determining the splitting line. This requires finding the median x- or y-coordinate, depending on whether the depth is even or odd. Although median finding can be done in linear time, linear-time median-finding algorithms are quite complex. The hidden cost is also large and we have to execute the median finding during every partition. An alternative and more efficient approach is to presort the set of points based on both x- and y-coordinates. The set P is then passed to the procedure in the form of two sorted lists—one sorted by x-coordinate and the other by y-coordinate. Given these sorted lists, the median-based partitioning takes linear time. Hence, the recurrence for the construction time $T(n)$ is:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1, \\ 2T(\lceil n/2 \rceil) + O(n) & \text{if } n > 1. \end{cases}$$

which solves to $T(n) = O(n \log n)$. This time complexity subsumes the time spent presorting the points by x- and y-coordinates.

Algorithm 4: SEARCHKDTREE(r, Q)

Input: The root node r of a kd-tree or its subtree, and a query range Q.
Output: All points in $T(r)$ that lie in Q.

```
1  if r = leaf then
2      Report the point stored at r if it lies in Q.
3  else
4      if R(left(r)) ⊆ Q then
5          ReportSubtree(left(r))
6      else if R(left(r)) ∩ Q ≠ ∅ then
7          SearchKDTree(left(r), Q)
8      if R(right(r)) ⊆ Q then
9          ReportSubtree(right(r))
10     else if R(right(r)) ∩ Q ≠ ∅ then
11         SearchKDTree(right(r), Q)
```

3.2 Query algorithm for kd-tree

Let r denote the root of T. The splitting line $\ell(r)$ at r partitions the entire plane into two half-planes—the left half-plane contains points stored in $T(\text{left}(r))$, while the right half-plane contains points stored in $T(\text{right}(r))$. In general, the region $R(v)$ associated with a node v is a rectangle, which may be unbounded on one or more sides. $R(v)$ is determined by the splitting lines stored at the ancestors of v. A point is stored in $T(v)$ if and only if it lies in $R(v)$.

The query algorithm is recursive. For a range query Q, visit only the nodes v of T with $R(v) \cap Q \neq \varnothing$. If $R(v) \subseteq Q$, immediately report all the points of $T(v)$ from its leaves, using the procedure REPORTSUBTREE(v). When the traversal reaches a leaf v, check whether the point stored at v lies inside Q and report it if so.

The main test is whether $R(v) \cap Q \neq \varnothing$ at a node v. For this, we can compute $R(v)$ for all nodes v during the preprocessing phase and store it. Alternatively, we can maintain the current region through the recursive calls using the lines stored in the internal nodes. For instance, the region corresponding to the left child of a node v at even depth can be computed from $R(v)$ as follows:

$$R(\text{left}(v)) = R(v) \cap \ell(v)_{\text{left}},$$

where $\ell(v)$ is the splitting line stored at v, and $\ell(v)_{\text{left}}$ is the half-plane to the left of and including $\ell(v)$.

Query time complexity: In Line 5 and Line 9 of SEARCHKDTREE, all leaf nodes are reported in time linear to the number of points stored in the subtrees $T(v)$, as the corresponding regions $R(v)$ are fully contained in Q.

Next, consider Line 7 and Line 11, which handle cases where the visited nodes correspond to regions partially intersecting Q. The number of such nodes matches the number of regions that partially intersect Q. This, in turn, asymptotically matches the number of regions intersected by any of Q's four sides. W.l.o.g., let's consider the left vertical side of Q, and let $f(n)$ be the maximum number of regions intersected by ℓ, the line containing that vertical side. Assume that n, the number of points in P, is a power of 2, i.e., $n = 2^t$ for some $t \geq 0$. You can verify later that the asymptotic value of $f(n)$ remains the same with this assumption. The derivation for $f(n)$ proceeds as follows.

Let r denote the root of T. The line ℓ intersects intersects $R(r)$ first (depth 0), and then at depth 1, intersects either $R(\text{left}(r))$ or $R(\text{right}(r))$ but not both. For example, in Figure 2, the left side of Q has intersected $R(\text{right}(r))$. At depth 2, ℓ intersects exactly two of the four regions corresponding to the four grandchildren of r. For example, in Figure 2, ℓ has intersected $R(\text{left}(\text{right}(r)))$ and $R(\text{right}(\text{right}(r)))$, and

but not $R(\text{left}(\text{left}(r)))$ or $R(\text{right}(\text{left}(r)))$. This observation leads to the following recurrence:

$$f(n) = \begin{cases} 2 & \text{if } n = 1, 2 \\ 2f(n/4) + 2 & \text{if } n = 2^2, 2^3, \ldots. \end{cases}$$

Applying the iterative method,[†] we get:

$$\begin{aligned} f(n) = f(2^t) &= 2 + 2f(2^{t-2}) \\ &= 2 + 2(2 + 2f(2^{t-2\times 2})) \\ &= 2 + 2^2 + 2^2 f(2^{t-2\times 2}) \\ &= 2 + 2^2 + 2^3 + 2^3 f(2^{t-2\times 3}) \\ &= 2 + 2^2 + 2^3 + \cdots + 2^s + 2^s f(2^{t-2s}), \text{ where, } t - 2s = 0 \text{ or } 1 \end{aligned}$$

$$\begin{aligned} t \text{ is even, i.e., } t - 2s = 0 \implies f(n) &= 2 + 2^2 + 2^3 + \cdots + 2^s + 2^s f(1) = 2(2^{s+1} - 1) \\ &= 4 \times 2^{t/2} - 2 = 4\sqrt{n} - 2 = O(\sqrt{n}). \\ t \text{ is odd, i.e., } t - 2s = 1 \implies f(n) &= 2 + 2^2 + 2^3 + \cdots + 2^s + 2^s f(2) = 2(2^{s+1} - 1) \\ &= 2 \cdot 2^{(t+1)/2} - 2 = 2\sqrt{2n} - 2 = O(\sqrt{n}). \end{aligned}$$

So, the total number of nodes explored in T is bounded by $O(\sqrt{n})$, and we have the main result:

> A rectangular query on a kd-tree takes $O(\sqrt{n} + k)$ time, where $k = \#$reported points.
> For a d-dimensional kd-tree, it can be shown that the query time is bounded by $O(n^{1-1/d} + k)$.

4 2D Range Tree

Similar to a 2-dimensional kd-tree, we are given a set P of n points on the xy-plane. For searching in a kd-trees, we need $O(\sqrt{n}+k)$ time, which remains high when the number of reported points k is relatively small. To address this limitation, the **range tree** is preferred, as it achieves a better query time of $O(\log^2 n + k)$. However, this improvement comes at the cost of an increased $O(n \log n)$ space complexity instead of $O(n)$ for kd-trees.

4.1 Construction algorithm

For a rectangular range query $Q = [a_x, b_x] \times [a_y, b_y]$, we simplify it into two 1D range queries: $Q_x = [a_x, b_x]$ and $Q_y = [a_y, b_y]$. We first perform the query Q_x on the **primary tree** T_x, which is a balanced binary search tree. The construction of T_x is similar to that of a 1D range tree, as it stores all points of P ordered by their x-coordinates. Additionally, from each leaf or non-leaf node v, there is a pointer to an exclusive **secondary tree** $T_y(v)$ corresponding to v. The tree $T_y(v)$ is also a 1D range tree, but it stores only the points in $T_x(v)$, ordered by their y-coordinates. $T_y(v)$'s are for querying with Q_y, as explained in §4.2.

Space complexity: The tree T_x itself takes $O(n)$ space. For each depth $i(\geq 0)$ of T_x, there are 2^i secondary trees, and each point $p \in P$ is contained in exactly one of them. Since each secondary tree is a 1D range tree, its size is linear in the number of points it contains. Thus, the total space complexity for all the secondary trees at a given depth i is $O(n)$. As the depth of T_x is bounded by $O(\log n)$, the total space required for all the secondary trees is $O(n \log n)$.

[†]We can alternatively use the Master Theorem. Recall that for $f(n) = af(n/b) + g(n)$ with $g(n) = \Omega(n^d)$, where $d < \log_b a$, the solution is $\Theta(n^{\log_b a})$. Here, $g(n) = 2 = \Theta(n^0)$, and since $0 < \frac{1}{2}$, we get $f(n) = \Theta(n^{\log_4 2}) = \Theta(\sqrt{n})$.

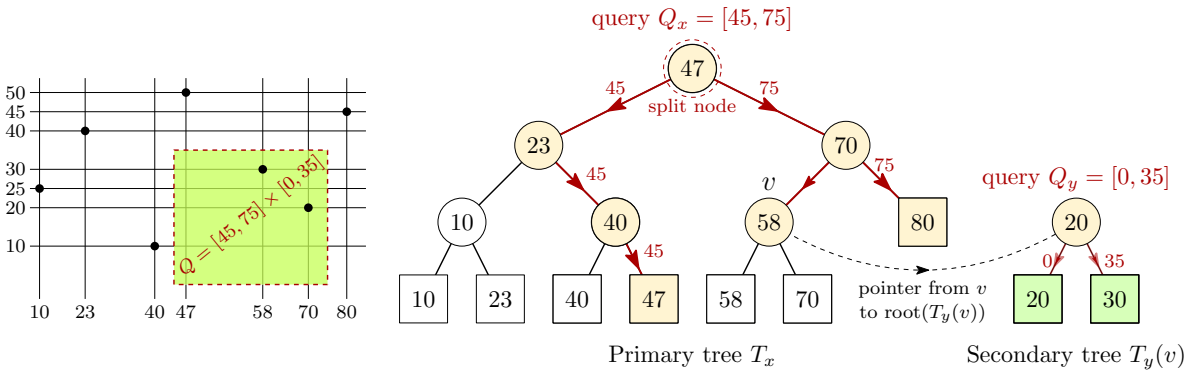

Figure 3: Query in a 2D range tree—an example.

Time complexity: To achieve optimal construction time, we first presort the points in P—once by x-coordinates to store in the set P_x, and once by y-coordinates to store in the set P_y. With these presorted sets, T_x can be built in $O(n)$ time, and building each secondary tree $T_y(v)$ will take time linear in the number of points it contains. As explained in the space complexity analysis, the points in P are partitioned among the secondary trees at each depth i. Therefore, the total time complexity to build all the secondary trees at depth i is $O(n)$. Since the depth of T_x is $O(\log n)$, the total time to build all the secondary trees is $O(n \log n)$.

4.2 Query algorithm

During the query with Q_x, we locate the split node in T_x where the search paths bifurcate for a_x and b_x. Let these bifurcated paths be π_L and π_R, respectively, and let V_x represent the nodes in $\pi_L \cup \pi_R$. The solution points can only be found in the subtrees of the nodes in V_x. Therefore, for each node $v \in V_x$, the corresponding secondary tree $T_y(v)$ is searched using the query Q_y to find the solution points in $T_y(v)$.

Time complexity: The solution points of $T_y(v)$ are obtained in $O(\log n + k_v)$ time, where $k_v = \#$solution points in $T_y(v)$. Suppose $|P \cap Q| = k$. As there are $O(\log n)$ nodes in π_L and π_R, we have $|V_x| = O(\log n)$, which yields the total query-time complexity:

$$\sum_{v \in V_x} O(\log n + k_v) = O(\log n) \times O(\log n) + \sum_{v \in V_x} O(k_v) = O(\log n \times \log n) + O(k) = O(\log^2 n + k).$$

Let $k = \#$solution points.

2D range query:
Total space $= O(n \log n)$ for primary tree plus all secondary trees.
Construction time $= O(n \log n)$.
Query time $= O(\log^2 n + k)$.

$d(\geq 2)$-dimensional range query:
Total space $= O(n \log^{d-1} n)$ for primary tree plus all secondary and tertiary trees.
Construction time $= O(n \log^{d-1} n)$.
Query time $= O(\log^d n + k)$. [Can be improved to $O(\log^{d-1} n + k)$ by *Fractional Cascading*]