COSC 520 Assignment 2 - Advanced Data Structures: Spatial Partitioning Trees

ZHENBANG HE, Uniersity of British Columbia, Canada

In this report, we introduce a series of spatial partition tree structures, progressing from the simpler QuadTree to the more flexible KDTree and finally to the more compact and computation-efficient Implicit KDTree. We explain their core ideas, construction strategies, and typical use cases. We implemented Python versions of all three structures in 2D, along with a data generation script. We evaluated these structures on generated datasets and compared their building time, query performance, and memory consumption. The results show that QuadTree offers intuitive spatial subdivision but suffers from uneven partitioning on irregular data distributions. KDTree provides more balanced splits and achieves higher efficiency for nearest neighbor search, while Implicit KDTree further reduces memory usage and improves traversal speed due to its compact representation. Through these experiments, we gained deeper insight into how different spatial partition trees as well as the design choices within each affect practical performance in spatial query workloads. The code and dataset are available at https://github.com/m-iDev-0792/COSC520-Assignments/tree/main/Assignment2.

1 Introduction

In many computer science applications, we often need to find nearby data points in a multidimensional space. This kind of problem appears in areas such as computer graphics, machine learning, and data mining. For example, when we need to locate similar images in a feature space, detect nearby objects, or search for the closest location on a map, we rely on efficient mechanisms for performing spatial queries. A simple brute-force method checks all points one by one, but this becomes prohibitively slow as the dataset grows. To address this challenge, various spatial data structures have been developed to accelerate search by organizing the data according to spatial relationships. Among these structures, spatial partition trees form a fundamental family of solutions. At the simpler end of the spectrum, the QuadTree recursively subdivides 2D space into four quadrants, offering an intuitive and uniform hierarchical representation that works well for many grid-like or image-based applications. Moving to more flexible methods, the KDTree (k-dimensional tree) partitions space by splitting along selected dimensions at each node, producing a balanced binary structure that adapts more effectively to data distribution and supports efficient nearest neighbor queries. More advanced still, the Implicit KDTree adopts a compact array-based representation that eliminates explicit pointers, improves cache locality, and reduces memory overhead while preserving the KDTree's hierarchical structure.

In this report, we examine these three spatial partition tree structures to better understand their design principles and performance trade-offs. We implement 2D versions of QuadTree, KDTree, and Implicit KDTree in Python and explore multiple KDTree variants with different strategies for split-dimension selection and approximate median sampling. Through systematic experiments on generated datasets, we compare their building time, query efficiency, and memory usage. This analysis allows us to evaluate how different spatial partition strategies behave in practice and to gain deeper insight into the considerations behind designing efficient spatial data structures.

2 Quad Tree Algorithm

2.1 Principles and Brief Introduction

QuadTree is a classic spatial partition structure designed specifically for organizing points or objects in a two-dimensional space. It was first proposed by Finkel and Bentley in 1974, one year before

Author's Contact Information: Zhenbang He, zbhe96@student.ubc.ca, Uniersity of British Columbia, Kelowna, British Columbia, Canada.

the introduction of KDTree. The essential idea of a QuadTree is to recursively divide the 2D plane into four equal-sized quadrants. Each internal node corresponds to a region of space and has up to four children, each representing one of the four subdivided areas.

Before efficient spatial trees were developed, searching for neighboring points or identifying which region a point belongs to required scanning the entire dataset. QuadTree introduced a hierarchical subdivision strategy that can adapt to local densities. Its simplicity and intuitive geometric interpretation make it a natural choice for many 2D applications.

2.2 Algorithm and Construction

A QuadTree organizes space by repeatedly subdividing square regions into four quadrants: northwest, northeast, southwest, and southeast. A region is subdivided only when it contains more points than a predefined threshold, or when finer granularity is required for accuracy.

2.2.1 Construction Process.

(1) **Define the bounding region.** The root node corresponds to a square region that contains all the input points. If the data does not initially fit into a square, we can expand the region to the smallest enclosing square.
(2) **Insert points into the tree.** For each point, we determine which quadrant it belongs to and insert it recursively.
(3) **Subdivide when necessary.** When a node contains more points than a specified limit (commonly one point per leaf), the region is split into four equal quadrants. All points in that region are then redistributed into the appropriate subregions.
(4) **Recursively build subtrees.** Subdivision continues until all points are placed in leaf nodes that no longer require splitting.

2.2.2 Query Process. To perform range search or nearest neighbor search in a QuadTree:

(1) Start from the root and check whether the query region intersects with the node's region.
(2) If they do not intersect, the entire subtree can be skipped.
(3) If they intersect, recursively explore the child nodes that overlap with the query region.
(4) For nearest neighbor search, we record the best point found and prune subtrees whose regions lie farther away than the current best distance.

2.2.3 Time Complexity. The time complexity of QuadTree depends heavily on the point distribution.

- Building complexity is typically $O(N \log N)$ for uniformly distributed data, but can degrade to $O(N^2)$ in the worst case when points cluster into very small regions.
- Query performance is approximately $O(\log N)$ on average for balanced QuadTrees, but may become slower for highly skewed datasets.

2.2.4 Space Complexity. QuadTrees usually require more memory than KD Trees because each internal node has four children instead of two. In the average case, the complexity is $O(N)$, but deep subdivision can lead to additional overhead. Nevertheless, for many 2D applications, the memory usage remains manageable and the intuitive structure is appealing for visualization and spatial indexing.

3 KD Tree Algorithm

3.1 Principles and Brief Introduction

KDTree, or k-dimensional tree, is a kind of data structure used to organize points in a k-dimensional space. It was first introduced by Jon Louis Bentley in 1975. The main idea behind KDTree is to

divide the space step by step into smaller regions, so that we can search for data points more efficiently. Bentley's original purpose was to create a simple and effective structure for handling multidimensional search problems, such as finding points within a region or finding the nearest neighbor of a given query point. KDTree is widely used in computer graphics, machine learning (for example in k-nearest neighbor algorithms), robotics (for path planning), and many other fields.

3.2 Algorithm and Construction

A KDTree is a binary tree where each node represents a region of the space. Each node stores a point and a splitting dimension. The space is divided by a hyperplane that is perpendicular to the chosen dimension. The left subtree contains all points that are smaller than the node's value on that dimension, and the right subtree contains all points that are greater.

3.2.1 Construction Process.

(1) **Choose a dimension to split.** There are different strategies to decide which dimension to use. The simplest method is to alternate the dimension at each level (for example, x, then y, then x again in 2D). Another method is to choose the dimension with the highest variance, because that usually provides a better balance and reduces the depth of the tree.
(2) **Find a splitting point.** Usually we select the median value of the chosen dimension, so that the tree remains roughly balanced. This helps keep the search efficient.
(3) **Recursively build subtrees.** The points on the left side of the split go into the left subtree, and the points on the right go into the right subtree. The recursion stops when there are no more points to split.

3.2.2 Query Process. To find the nearest neighbor of a given point, KDTree uses a recursive search.

(1) We start from the root and move down the tree by comparing the query point with the node's splitting dimension, just like in binary search.
(2) When we reach a leaf node, we record the best (closest) point found so far.
(3) Then we go back up the tree and check whether the other side of the split might contain a closer point. If the distance between the query point and the splitting plane is smaller than the current best distance, we need to explore that other branch.
(4) This process is called pruning, because we skip the branches that are impossible to contain a better answer. With pruning, the search becomes much faster than checking all points.

3.2.3 Time Complexity. The time complexity of KDTree depends on the data distribution and the dimension.

- For building the tree, if we use an efficient algorithm to find the median at each step, the time complexity is about $O(N \cdot logN)$, where N is the number of points.
- For searching (nearest neighbor query), the average time complexity is about $O(logN)$, which is much faster than the linear search $O(N)$. However, in the worst case (for example, when the data is very high-dimensional or unevenly distributed), the search may degrade close to $O(N)$.

In practice, KDTree performs very well for low and medium dimensions (usually fewer than 20). But when the dimension is too high, the performance advantage becomes smaller because of the "curse of dimensionality."

3.2.4 Space Complexity. The space complexity of KDTree is mainly determined by the number of nodes and the extra information stored in each node. For each point, we store the coordinates, the splitting dimension, and links to its child nodes. Therefore, the space complexity is roughly $O(N)$, which is quite efficient. However, if we want to store additional data for each node (for example,

bounding boxes, precomputed distances, or cached values), the memory usage can increase. In most simple implementations, KDTree remains very lightweight. For large datasets, memory usage is still much smaller than storing all pairwise distances, which makes KDTree a practical choice for nearest neighbor search.

4 Implicit KD Tree Algorithm

4.1 Principles and Brief Introduction

Implicit KDTree is a variant of the KDTree that stores the tree nodes in an array rather than using dynamically allocated node objects with explicit pointers. This design is inspired by the heap representation of binary trees, where the left and right children of index i are located at indices $2i + 1$ and $2i + 2$, respectively. By eliminating pointers and using a compact memory layout, the Implicit KDTree achieves better cache efficiency and reduces memory overhead.

This structure is particularly useful when the dataset is static and the tree does not need to be updated dynamically. It has gained attention in high-performance computing, real-time rendering, and large-scale nearest neighbor search tasks where memory footprint and cache behavior are critical.

4.2 Algorithm and Construction

Similar to a regular KDTree, the Implicit KDTree recursively partitions the space using splitting dimensions and median values. However, instead of allocating nodes dynamically, all nodes are stored sequentially in an array.

4.2.1 Construction Process.

(1) **Determine splitting dimension.** As with standard KDTree, we may alternate dimensions or select the one with largest variance.
(2) **Find the median point.** The median along the chosen dimension becomes the root of the current subtree. Instead of creating a tree node, we store the point's coordinates directly in the appropriate array index.
(3) **Recursively build left and right subtrees.** The points smaller than the median go into the left subtree (index $2i + 1$) and the larger ones go into the right subtree (index $2i + 2$). This recursion continues until each subtree is processed.
(4) **Terminate on empty regions.** If a region has no points left, the corresponding index in the array remains empty or is marked with a placeholder.

4.2.2 Query Process. Nearest neighbor search in an Implicit KDTree follows the same logic as in a standard KDTree:

(1) Traverse the array as if navigating a binary tree, comparing the query point with the splitting dimension of each node.
(2) Maintain and update the current best point.
(3) Prune branches that do not need further exploration.
(4) Because nodes stored consecutively often reside close in memory, traversal can benefit from CPU cache locality, resulting in faster query performance.

4.2.3 Time Complexity. The time complexity remains similar to the standard KDTree:

- Building the tree requires approximately $O(N \log N)$ with efficient median selection.
- Query complexity averages around $O(\log N)$, though it may degrade to $O(N)$ in worst-case situations.

However, the practical running time is often smaller due to improved memory behavior.

4.2.4 Space Complexity. Implicit KDTree significantly reduces memory overhead:

- Instead of storing pointers and node structures, all data is kept in a flat array.
- Space complexity is still $O(N)$, but with a much smaller constant factor compared to the explicit KDTree.
- The trade-off is that the array may contain unused positions, especially when the tree is not perfectly balanced, which may cause slight overhead.

For static datasets, the memory efficiency and cache-friendly access pattern make Implicit KDTree an attractive option for large-scale nearest neighbor applications.

5 Implementation

In this section, we describe the implementation of our spatial indexing system and its supporting scripts. The full system comprises five major components: the standard KD-Tree module (`kd_tree_2d.py`), the Implicit KD-Tree module (`implicit_kd_tree_2d.py`), the Quad-Tree module (`quadtree_2d.py`), the dataset generation module (`dataset_gen.py`), and the dataset loading module (`dataset_load.py`).

5.1 Dataset Generation

The dataset generation process is implemented in `dataset_gen.py`. The script can generate any number of random 2D points within a user-defined rectangular area. Points are uniformly sampled using NumPy's random number generator, and the results are written to plain text files in chunks. Each line in a file represents a single data point in the format "x y". The generation process supports parallelism through Python's `multiprocessing` library, allowing large datasets (for example, tens of millions of points) to be created efficiently on multi-core systems.

Several parameters can be configured through command-line arguments, including the number of points, the output directory, the number of workers, and the floating-point precision. The use of fixed random seeds ensures reproducibility across runs. This dataset generator was used to create various test sets for benchmarking different KDTree configurations.

5.2 Dataset Loading

To read the generated datasets, we implemented `dataset_load.py`, which provides a convenient interface for streaming or loading 2D points from text files. The function `load_points()` can load a specified number of points or all points from a given folder. It also supports shuffling and conversion to NumPy arrays for efficient numerical processing. Files are read in buffered mode to improve I/O performance, and both comma- and space-separated formats are supported. This flexibility allows the system to easily handle large datasets without loading them entirely into memory at once.

5.3 Nearest Neighbor Query

For nearest neighbor queries, the algorithm performs a recursive search starting from the root. At each node, it compares the target point to the splitting hyperplane and explores the most promising branch first. Then, it decides whether to explore the other branch based on the current best distance and the bounding box of the opposite subtree. This pruning strategy significantly reduces the search space, especially for balanced trees.

5.4 Visualization Module

To better understand the behavior of KD-Tree queries, a visualization module was implemented (see `visualize_kd_tree_2d.py`). This tool uses `matplotlib` and `animation` to dynamically illustrate the step-by-step process of nearest neighbor search.

The visualization highlights different regions of the search space as the algorithm progresses. The light blue areas (Figure. 1 (a)) represent regions that are being visited, while the light gray regions (Figure. 1 (b)) indicate pruned subtrees that are skipped due to bounding-box pruning. The query point is shown as a red dot, and the current best nearest neighbor is displayed as a green star. The animation shows how the KD-Tree recursively explores and prunes regions, helping users intuitively understand how pruning improves efficiency.

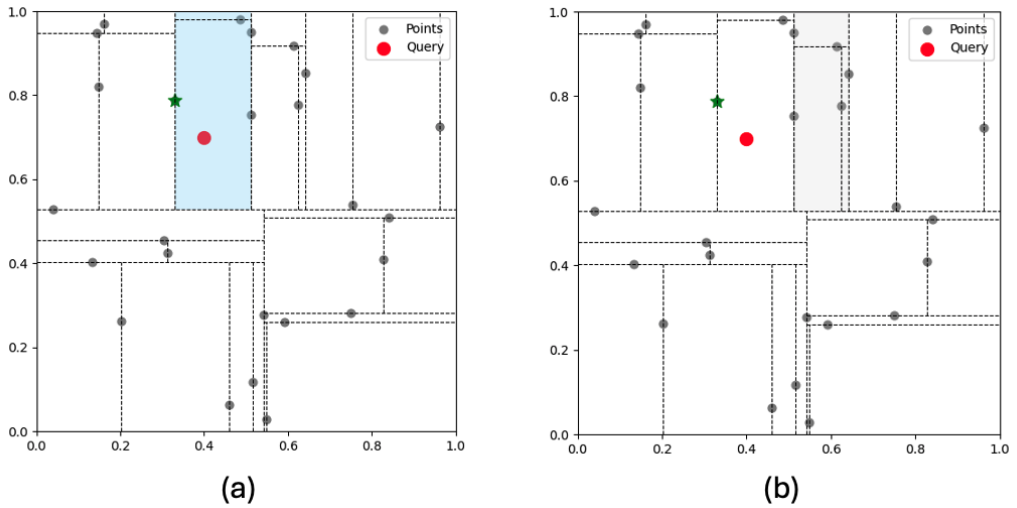

Fig. 1. Visualization of the KD-tree searching process: (a) The light blue region indicates the area currently being visited. (b) The light gray region indicates the area skipped due to pruning.

6 Experiments

6.1 Experimental Setup

To evaluate the performance of the three spatial partition tree structures (Quad Tree, KD Tree, and Implicit KD Tree), we designed a unified set of benchmark experiments using the same data generation procedure and evaluation pipeline. The objective of these experiments is to measure how different tree construction strategies and structural properties influence nearest neighbor search performance, memory characteristics, and preprocessing costs across varying dataset sizes.

The experiments were conducted using randomly generated 2D point datasets. Approximately 20% of all points were used as query points, while the remaining 80% were used to construct the spatial data structures. All experiments were performed on a Mac Mini 2024 desktop equipped with an Apple M4 Pro CPU and 64GB RAM, ensuring consistent and reproducible test conditions.

6.2 Compared Methods

We compared the following nearest neighbor search strategies implemented in our framework:

(1) **Linear Search (Brute Force):** This method exhaustively checks all points to find the nearest neighbor. While computationally expensive, it provides a correctness reference and an absolute lower bound for search accuracy.
(2) **Quad Tree:** The Quad Tree recursively subdivides the 2D space into four quadrants. For each node, subdivision occurs when the number of points exceeds a specified threshold, resulting in adaptive partitioning.
(3) **KD Tree:** This implementation builds a balanced binary tree by selecting median-based splits at each recursive step.
(4) **Implicit KD Tree:** This variant stores the KD Tree nodes in a flat array using a heap-like indexing scheme. It eliminates pointer overhead and improves cache utilization. The same splitting strategies as the explicit KD Tree are applied, but the storage layout differs significantly.

For KD Tree and Implicit KD Tree, we tested two axis selection strategies:

- **Variance-based:** The split is chosen along the axis with the highest variance to ensure more balanced partitions.
- **Interleaving-based:** The split alternates deterministically between x and y axes, reducing overhead at the cost of potentially less spatially informed splits.

For Quad Tree, construction does not rely on axis selection or median values; therefore, its behavior differs structurally. However, for consistency, we compare its performance against all KD Tree configurations.

By combining all factors, the complete set of evaluated configurations is:

Linear Search, Quad Tree, KDTree (V), KDTree (I), Implicit KDTree (V), Implicit KDTree (I)

where V = Variance, I = Interleave. These configurations allow us to study how differences in spatial partitioning strategies (uniform subdivision vs. adaptive binary splits), storage layouts (pointer-based vs. array-based), and construction heuristics impact the overall efficiency of spatial querying workloads.

6.3 Measured Metrics

For each configuration, we measured three main performance metrics:

- **Build Time (s):** The time required to construct the data structure, including median calculation and recursive partitioning.
- **Query Time (s):** The total time for performing nearest neighbor queries for all sampled query points.
- **Peak Memory Usage (MB):** The maximum memory consumption recorded during the process using the `tracemalloc` module.

All measurements were averaged over a single run per dataset size, as the random sampling and approximate median selection already introduce variability. We also ensured reproducibility by fixing the random seed when applicable.

6.4 Data Scale and Repetitions

The benchmarks were conducted across different dataset sizes ranging from 10,000 to 10 million points. Specifically, the tested scales were:

$$N = \{10^4, 5 \times 10^4, 10^5, 2 \times 10^5, 5 \times 10^5, 10^6, 2 \times 10^6, 5 \times 10^6, 10^7\}.$$

For each scale, the query set size was set to **5000**. This large range allows us to observe both small-scale and large-scale performance trends and to examine how different methods scale with increasing data.

6.5 Results and Discussion

The experiment results are summarized in Table 1, and the visualized metrics are shown in Figure 2. The comparison includes build time, query time, and memory usage across Brutal Force, Quad Tree, KD Tree, and Implicit KD Tree under multiple dataset sizes.

6.5.1 Build Time. The linear search baseline requires no preprocessing. Among spatial trees, **Quad Tree demonstrates the fastest construction time for small to medium datasets** (below 10^5 points), owing to its lightweight subdivision rule. Its construction cost increases moderately with data scale.

KD Tree constructions vary significantly by configuration. **Interleave-based KD Tree** builds faster than its variance-based counterpart because it avoids variance computation. **Variance-based KD Tree with exact median** is the slowest configuration because median selection and dimension evaluation are expensive at large scales.

Implicit KD Tree shows the same trend: the variance-based version has the highest construction overhead, while the interleaving strategy reduces its build time. The results indicate that **Implicit KD Tree has the highest construction cost overall**, especially for large datasets ($> 10^6$ points), mainly due to array-index organization.

6.5.2 Query Time. Query performance shows clear differences across methods. As expected, linear search scales poorly and becomes impractical beyond 10^5 points.

Quad Tree provides faster querying than brute force, but its performance degrades noticeably as the point distribution becomes non-uniform. In the benchmark, Quad Tree's query time is consistently higher than all KD Tree variants.

Variance-based KD Tree achieves the best query performance among all tested methods. For $N = 10\,000$, its average nearest-neighbor query time is around 0.04 s, significantly outperforming Quad Tree (0.65 s) and linear search (0.14 s). Interleave-based KD Tree is slightly slower but remains highly efficient.

Implicit KD Tree achieves **stable but not minimal** query time. Its contiguous array layout provides good cache locality, but the lack of pointer-based pruning optimization leads to performance slightly slower than explicit KD Trees. Query time remains around 0.10–0.15 s for moderately sized datasets.

6.5.3 Memory Usage. Memory usage grows approximately linearly with dataset size. **Quad Tree consistently exhibits the highest memory footprint**, due to expanded node structures and up to four child pointers per internal node.

KD Tree uses less memory than Quad Tree, and interleave-based KD Trees tend to be slightly more compact than variance-based variants because fewer intermediate computations are stored.

Implicit KD Tree demonstrates the lowest memory consumption across all datasets. Its flat array eliminates pointer overhead and reduces fragmentation, achieving less than half the memory footprint of explicit KD Trees for large datasets. This difference becomes increasingly significant as N exceeds one million.

Overall Observation. Based on the experimental results:

- Quad Tree offers very fast construction and reasonable performance for uniform or low-resolution workloads but falls behind KD Trees in query efficiency.

- KD Tree (especially variance-based) provides the strongest nearest-neighbor query acceleration across all dataset sizes.
- Interleave-based KD Tree offers an appealing balance between construction speed and search efficiency.
- Implicit KD Tree delivers the best memory efficiency and stable query time, but requires the highest build time when using variance-based splitting.

Figure 2 visualizes the overall performance landscape. **Variance-based KD Tree is the most effective structure for pure query speed**, while **Implicit KD Tree is the best choice when memory is constrained**. Quad Tree remains useful as a lightweight and robust baseline for 2D problems, particularly when construction time dominates.

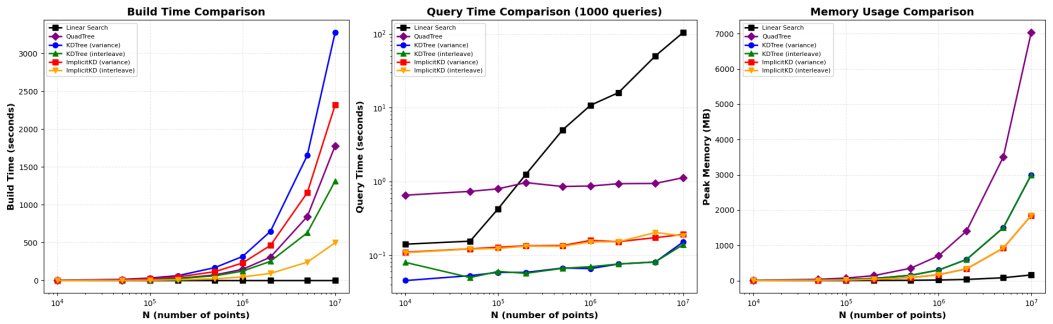

Fig. 2. The build time, query time and memory usage under different spatial partition configurations.

Across all methods, spatial partition trees significantly outperform brute-force search beyond moderate dataset sizes. The choice of structure depends on the available memory budget, construction time constraints, and desired query performance.

6.6 Discussion

Beyond the raw performance metrics, the benchmark results offer useful insights into when each spatial partition structure is most suitable. Quad Tree, with its extremely fast construction and simple geometric subdivision, is well suited for applications that require rapid preprocessing or frequent tree rebuilding, such as real-time simulations, game engines, particle systems, and approximate spatial lookups where query optimality is not critical. Its performance, however, is sensitive to non-uniform distributions, making it less ideal for datasets with strong clustering.

KD Tree demonstrates the most reliable and scalable nearest-neighbor performance, particularly under variance-based splitting. This makes it appropriate for workloads dominated by repeated or large-volume queries, where preprocessing cost is amortized, such as recommendation systems, geographic search, clustering algorithms, or machine learning pipelines that require nearest-neighbor operations. The interleave-based version is advantageous when construction cost must be kept low, such as in scenarios involving frequent incremental rebuilds.

Implicit KD Tree is especially suitable for memory-limited environments or systems where cache efficiency is essential, including embedded devices, mobile platforms, or large in-memory analytics systems. Although its construction cost is higher, its compact array-based representation ensures predictable memory behavior and stable query performance. It also fits static-data applications where the structure is built once and queried extensively.

7 Conclusion

This report compared three 2D spatial partition structures, Quad Tree, KD Tree, and Implicit KD Tree, through unified implementations and benchmark experiments. All spatial trees significantly accelerate nearest-neighbor search compared with brute-force scanning, but they differ in construction cost, query efficiency, and memory usage.

Quad Tree excels in situations demanding rapid construction, KD Tree provides the strongest overall query performance, and Implicit KD Tree offers superior memory efficiency with competitive search speed. These findings highlight that no single structure is universally optimal; instead, each represents a different point in the trade-off space between preprocessing time, query latency, and memory footprint. The appropriate structure should be selected according to the dataset characteristics and the performance requirements of the application.

Acknowledgments

The report referred some formulas and algorithms introduced in following links [3–6, 8, 9].Some parts of the code in this assignment were generated with the assistance of the Cursor Editor, a development environment that integrates multiple large language model (LLM) agents. The underlying backends of Cursor may include models such as ChatGPT [7], Claude [1], and others [2]. I hereby acknowledge their contribution. However, since I primarily used the Cursor Editor in a copilot-style fashion, the generated code largely extended my existing code structure and prior snippets. Therefore, the final implementation of this assignment should not be considered as being fully produced by LLMs.

References

[1] Anthropic. 2023. Claude AI. https://www.anthropic.com/claude. Accessed: 2025-10-03.
[2] Cursor. 2023. Cursor: The AI Code Editor. https://cursor.sh. Accessed: 2025-10-03.
[3] GeeksforGeeks. 2023. Ball Tree and KD Tree Algorithms. https://www.geeksforgeeks.org/machine-learning/ball-tree-and-kd-tree-algorithms/ Accessed: 2025-10-25.
[4] GeeksforGeeks. 2023. Search and Insertion in K-Dimensional Tree. https://www.geeksforgeeks.org/dsa/search-and-insertion-in-k-dimensional-tree/ Accessed: 2025-10-25.
[5] GeeksforGeeks. 2025. *Quad Tree.* https://www.geeksforgeeks.org/dsa/quad-tree/ Accessed: 2025-11-07.
[6] Carl Kingsford. n.d.. KD-Trees. https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/kdtrees.pdf Lecture notes, Carnegie Mellon University.
[7] OpenAI. 2022. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/chatgpt. Accessed: 2025-10-03.
[8] Katyayani Vemula. 2023. What is a K-dimensional Tree? https://medium.com/@katyayanivemula90/what-is-a-k-dimensional-tree-8265cc737d77 Medium article.
[9] Wikipedia contributors. 2025. *Implicit k-d tree.* https://en.wikipedia.org/wiki/Implicit_k-d_tree Wikipedia, The Free Encyclopedia; Accessed: 2025-11-07.

Table 1. Benchmark results for spatial tree construction and query performance

N	tree_type	method	build_time_s	query_time_s	avg_query_ms	peak_MB
10000	Linear		0.000779	0.141180	0.141180	0.160440
10000	QuadTree		1.129639	0.650685	0.650685	7.030941
10000	KDTree	variance	3.129203	0.045349	0.045349	3.021854
10000	KDTree	interleave	1.186913	0.079982	0.079982	3.009102
10000	ImplicitKDTree	variance	2.399890	0.109997	0.109997	1.841264
10000	ImplicitKDTree	interleave	1.286256	0.087203	0.087203	1.841264
50000	Linear		0.004835	0.789838	0.789838	0.802200
50000	QuadTree		6.275936	2.082045	2.082045	23.029917
50000	KDTree	variance	16.536773	0.288111	0.288111	15.015944
50000	KDTree	interleave	6.411401	0.242659	0.242659	15.015944
50000	ImplicitKDTree	variance	16.137145	0.249162	0.249162	15.015944
50000	ImplicitKDTree	interleave	6.318695	0.240680	0.240680	14.580000
100000	Linear		0.012003	1.995282	1.995282	1.600880
100000	QuadTree		13.942407	4.353590	4.353590	37.717263
100000	KDTree	variance	32.018244	0.260243	0.260243	30.050000
100000	KDTree	interleave	12.514678	0.251047	0.251047	30.049000
100000	ImplicitKDTree	variance	32.972808	0.274901	0.274901	29.982000
100000	ImplicitKDTree	interleave	12.657861	0.268650	0.268650	29.854000
200000	Linear		0.008340	5.894198	5.894198	3.804000
200000	QuadTree		27.998300	9.390100	9.390100	52.997000
200000	KDTree	variance	70.110231	0.310077	0.310077	60.224000
200000	KDTree	interleave	27.575672	0.278487	0.278487	60.224000
200000	ImplicitKDTree	variance	71.131508	0.287372	0.287372	58.885000
200000	ImplicitKDTree	interleave	29.089955	0.286022	0.286022	58.885000
500000	Linear		0.027127	22.412923	22.412923	8.001000
500000	QuadTree		107.630000	22.860000	22.860000	116.004000
500000	KDTree	variance	174.571764	0.296809	0.296809	149.170000
500000	KDTree	interleave	66.757586	0.307922	0.307922	149.170000
500000	ImplicitKDTree	variance	176.631111	0.343693	0.343693	148.728000
500000	ImplicitKDTree	interleave	68.032571	0.330661	0.330661	148.716000
1000000	Linear		0.091671	51.443074	51.443074	16.000000
1000000	QuadTree		361.198173	33.066242	33.066242	298.463000
1000000	KDTree	variance	135.058182	0.336642	0.336642	298.463000
1000000	KDTree	interleave	343.163051	0.300612	0.300612	298.463000
1000000	ImplicitKDTree	variance	132.422266	0.305877	0.305877	301.499000
1000000	ImplicitKDTree	interleave	0.005522	88.423747	88.423747	302.042000
2000000	QuadTree		672.749702	0.352914	0.352914	597.049000
2000000	KDTree	variance	258.692022	0.354970	0.354970	593.090000
2000000	KDTree	interleave	677.690723	0.328770	0.328770	593.090000
2000000	ImplicitKDTree	variance	272.543088	0.351209	0.351209	590.022000
2000000	ImplicitKDTree	interleave	0.033667	25.100756	25.100756	589.551000
5000000	Linear		0.091994	0.000000	0.000000	158.000000
5000000	QuadTree		1683.264672	0.369458	0.369458	1499.551000
5000000	KDTree	variance	659.355366	0.347988	0.347988	1472.426000
5000000	KDTree	interleave	1680.973910	0.368688	0.368688	1472.426000
5000000	ImplicitKDTree	variance	679.057911	0.364636	0.364636	1472.426000
5000000	ImplicitKDTree	interleave	0.034018	528.382304	528.382304	1600.000000
10000000	KDTree	variance	3322.512588	0.382914	0.382914	2969.227000
10000000	KDTree	interleave	1312.355432	0.641986	0.641986	2969.227000
10000000	ImplicitKDTree	variance	3359.254931	0.497951	0.497951	2969.227000
10000000	ImplicitKDTree	interleave	1360.382877	0.442839	0.442839	3052.927000