Computer Science Extended Essay

Analyzing the efficiency of different multi-dimensional data structures

How does the efficiency of Quadtrees compare to that of a K-dimensional Tree in two-dimensional space?"

Topic: Computer Science

Candidate Number: lfg856

Session: May 2025

Page Count: 52

Word Count: 3868

Table of Contents

1. Introduction:

This paper will focus on the representation of trees in two-dimensional space, specifically Quadtrees and K-dimensional trees. Though a seemingly simplistic way to model data, they have many practical applications ranging from medical applications to mapping software. With an increasingly data-driven world, there has been an increase in research in both k-d trees and quadtrees due to their ability to model multi-dimensional data. A successful tree will enable faster computations in areas such as Artificial Intelligence as well as path-generation softwares used in robotics.

The key to creating an efficient tree is to find the balance between a high constant time factor and a low variable-dependent one. Maintaining this balance ensures the least amount of time is spent. Thus, given a set of data, the experimental and theoretical time complexity of operations of both data structures will be compared to evaluate the efficiency of each. Time complexity in this context is defined as "a description of how much computer time is required to run an algorithm" (Eldridge). The operations that will be used to evaluate each data structure are as follows:

1. Insertion of A. The insertion of a node into a tree.
2. Deletion of A. The deletion of a node from a tree. If this node doesn't exist, nothing will be changed
3. Nearest Neighbor of A. The point in the dataset that is the closest to A. For this paper, the "closeness" factor will be determined by the Euclidean distance, defined as follows:

$$f(X,Y) = \sqrt{\sum_{i=1}^{n}(X_i - Y_i)^2}$$

4. Range Queries of $\{B_1, B_2, B_3, \dots, B_n\}$, where $B_i = \{L, U\}$. The number of points contained in the hyperspace where the bounds of it in dimension i are given by B_i

This provides the opportunity to investigate the research question, "How does the efficiency of Quadtrees compare to that of a K-dimensional Tree in two-dimensional space?"

2. Theory:

2.1 Binary Search Tree:

A binary search tree (BST) is a data structure that serves as the foundation to both the K-d tree and Quadtree. Rather than searching through multi-dimensional data, the case with K-d trees and Quadtrees, a BST only searches in a single dimension. As with any tree, a BST is composed of both nodes and edges; however, each node can have at most two children. The unique property of a BST is that it constructs a sorted tree that enables a fast traversal. This is done through making every node of the left subtree smaller than a node, and every node of the right subtree greater than a node. The BST operations important in understanding k-d trees and Quadtrees are insertion, deletion, and rebalancing. While some implementations consider duplicates to be different nodes, this paper will consider duplicates to be the same node.

2.1.1 Binary Search Tree Insertion:

To insert a new node, the tree must be traversed. At every node, the node to be added is compared to the existing node, with the left being smaller, and the right being larger. This algorithm is summarized below:

1. If the tree is empty, the root node is set to A
2. If A has a value less than the current node, the insertion process will continue in the left subtree
3. If A has a value greater than the current node, the insertion process will continue in the right subtree
4. If the current node is empty, the node is set to A

This enables a tree to be constructed where all the nodes in the left subtree are less than the node and all nodes in the right subtree are greater than the node.

2.1.2 Binary Search Tree Deletion:

To delete a node, the same traversal used in insertion is used. Assume that node A is to be deleted. When the traversal reaches A we have four cases:

1. A is a leaf
2. A has no left subtree
3. A has no right subtree
4. A has both left and right subtrees

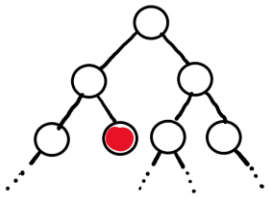

Figure 1: An example of case 1, with the shaded node to be deleted

For case 1, all that is necessary is for A to be deleted. For the rest of the cases, assume that deleted node is at depth d.

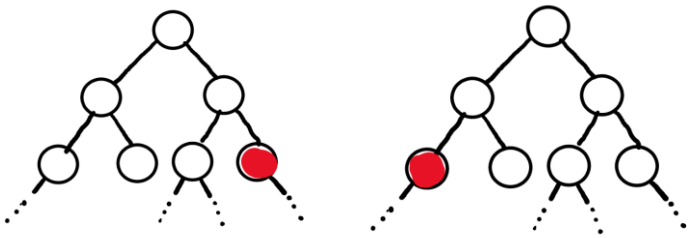

Figure 2: An example of case 2 on the left and case 3 on the right, with the shaded node to be deleted

For case 2, we consider the right subtree. In order for the BST to maintain its properties, everything right of the node replacing A must be greater than it. Therefore, we find the smallest value B in the right subtree of A, because everything in this subtree is greater than it. Then, this deletion algorithm is recursively called to delete B from the right subtree of A. Case 3 follows similar logic but instead finds the maximum of the left subtree of A.

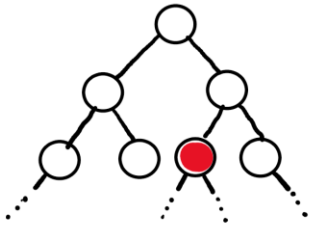

Figure 3: An example of case 4, with the shaded node to be deleted

For case 4, both methods for approaching case 2 and 3 may be used to delete A. In this paper, the method of finding the maximum in the left subtree is used if A has both left and right subtrees.

2.1.3 Binary Search Tree Rebalancing:

When constructing a BST, it may be the case that an unbalanced tree is constructed, making the time complexity of these operations $O(n)$ instead of $O(\log n)$. For example, consider constructing a tree in the following order: $\{1, 2, 3, 4, 5, 6\}$. This builds the following BST:

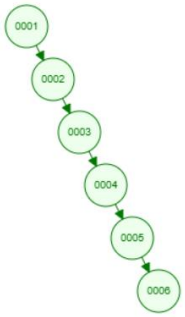

Figure 4: A BST of the set $\{1, 2, 3, 4, 5, 6\}$ ('Binary Search Tree Visualization')

In order to preserve a balanced tree, different BSTs have been implemented, such as the Red-Black Tree or AVL Tree, that implicitly reorder the nodes to preserve a balanced tree, while not costing extra time or space complexity. The algorithms used in these data structures will not be covered as they are outside the scope of this paper, but the idea of rebalancing will be used later.

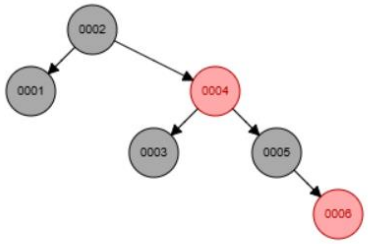

Figure 5: The balanced tree of the $\{1, 2, 3, 4, 5, 6\}$ performed by a Red-Black Tree ('Red/Black Tree Visualization')

2.2 Quadtree:

A quadtree is a data structure that represents 2-dimensional data by creating square regions to successfully identify regions with points inside of them. Geometrically, this will be a square region, that continually divides the original square region into four new squares located inside of it. For convenience, this paper will describe these regions as NW, NE, SE, and SW, representing the top left, top right, bottom right, and bottom left corner.

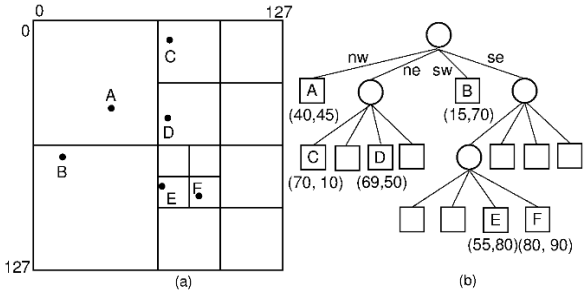

Figure 6: Two representations of a Quadtree. The left is the physical graphical representation, whereas the right is the tree representation ('15.3. The PR Quadtree')

This tree will be built on nodes that represent the square region rather than individual nodes. Each square region has a certain capacity of points it can contain before it subdivides into four more rectangles. The capacity that will be used is four. Additionally, this node will have four children: NW, NE, SE, and SW. Furthermore, the square is represented inside the class to show the overall region that the node represents.

Note that to traverse this tree to find a point, we have to search until a leaf is reached because through the insertion discussed below, all the points are propagated into the smaller division of squares in the region as shown in Figure 6.

2.2.1 Quadtree Insertion:

To insert node A into a quadtree we have to find the square region, R, where A is located. If the addition to R exceeds the capacity of four in the region we further divide R into four regions and redistribute the points in R. To do this, we call the insertion function again to add A into one of the four subregions. If the addition is below the capacity, we simply add the point to the region. These two cases can be expressed below:

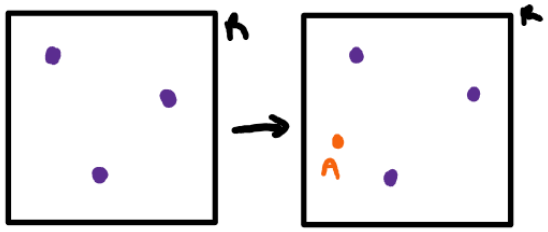

Figure 7: The case where A is added into region R, but it doesn't exceed the capacity. No subdivision is called

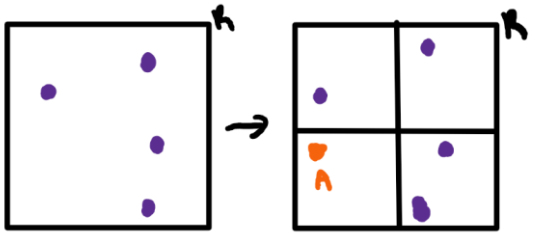

Figure 8: The case where A is added into region R, but it does exceed capacity. One subdivision is called

2.2.2 Quadtree Deletion:

The deletion of node A into a quadtree again requires us to find the square region, R, where A is located. When this region R is found, A is deleted from here. After the removal we have three cases:

1. The capacity of the parent of R is equal to the bounds after the deletion. If this is true, we can regroup the NW, NE, SE, SW region into one region.
2. The capacity of the parent of R is over the bounds after the deletion.
3. The capacity of the parent of R is smaller than the bounds after the deletion

The three cases are described below:

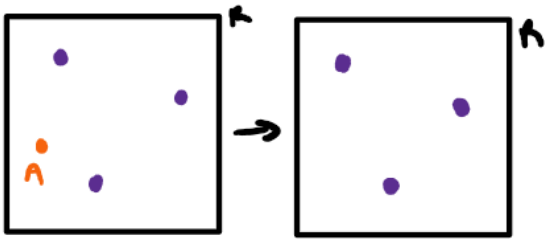

Figure 9: An example of case 3. This is already an indivisible region, so removing a point does not damage the structure. Similar logic applies to case 2.

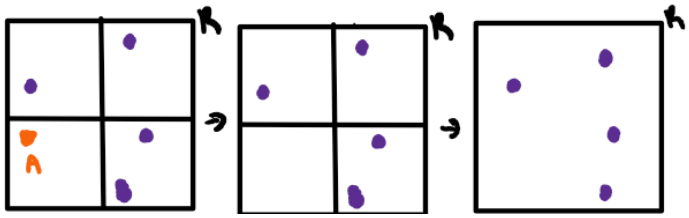

Figure 10: An example of case 1. After the removal of A, the capacity of the parent of R becomes 4 instead of 5. Thus, the subdivisions are removed, as the capacity is not exceeded

2.2.3 Quadtree Rebalancing:

No implicit way exists to rebalance a quadtree, so an external method is used. The rebalance algorithm will be called every 10 change operations (addition or deletion) to ensure a balanced tree. To rebalance a quadtree, a median is used to separate the data into two halves. The tree is then rebuilt from this data point serving as the node.

The median algorithm being used is a modified version of the median-of-median algorithm published by Blum et al, which finds the median in linear time. This will be done by randomly choosing a pivot and maintaining a list of points greater than (highPoints), less than (lowPoints), or equal to (equalPoints) the chosen pivot. The size of these arrays will then be evaluated to find the k index. There are three cases where index k can reside:

1. k is located inside lowPoints
2. k is located inside equalPoints
3. k is located inside highPoints

For case 1, the function is recursed to find index k in lowPoints. For case 2, the pivot is outputted because the median range is all the same. For case 3, the function is recursed to find index k - size(lowPoints)-size(highPoints) in highPoints.

The average time complexity of this algorithm is analyzed below (Moore). On average, the pivot will split the list into two pieces that are approximately the same length. Set the total amount of points to be n. Therefore, the average time-complexity can be derived from the following:

$$C = n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots = 2n = O(n)$$

The sum of this series is derived from the sum of an infinite sequence. The median-in-median algorithm used by Blum et Al creates a deterministic $O(n)$ time complexity by splitting the list in chunks of five elements and sorts the chunks as a sub-problem. This method isn't used for simplicity purposes.

2.2.4 Quadtree Nearest Neighbor Search:

Consider the geometrical interpretation of a quadtree, and that we are trying to find the nearest neighbor to node A. We traverse the tree until we find the indivisible region containing A. When we are at the indivisible region, every node in this region is manually checked. However, this does not account for all shortest distances as the nearest neighbor may not be in the current region. For example, consider the following example,

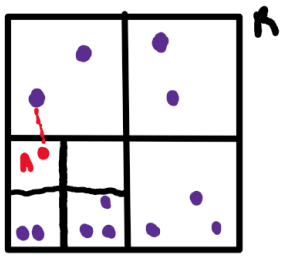

Figure 11: An example when the nearest neighbor of *A* is not in its indivisible region, but rather outside of it. The nearest neighbor is shown by the red line

2.2.5 Quadtree Range Query:

To find the range query, we again divide the searchable region until the current region is inside of the desired bound. If it is not completely contained inside the bounds, we continue the range query onto the subdivided regions. This propagation stops when we reach an indivisible region. Here we manually check all nodes in the region. Consider the following example,

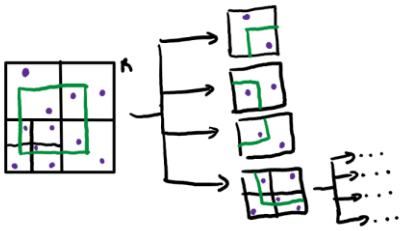

Figure 12: An example of the recursive calls in finding a range query. The green box is the range query we seek to obtain. After the first query is called, the NE, SE, SW, NW are checked if the range lies in one of the regions.

2.3 K-Dimensional Tree

A k-dimensional tree is a space partitioning data structure meant to organize points in k-dimensional space. Similar to a BST, each node is split based on the dimension correlated with its depth. Geometrically, split can be thought of as a hyperplane that divides the remaining set of points into two spaces (left and right subtrees). If the depth exceeds the number of dimensions, the place compared is the depth in the modulo of the number of dimensions. This will be explained in detail later.

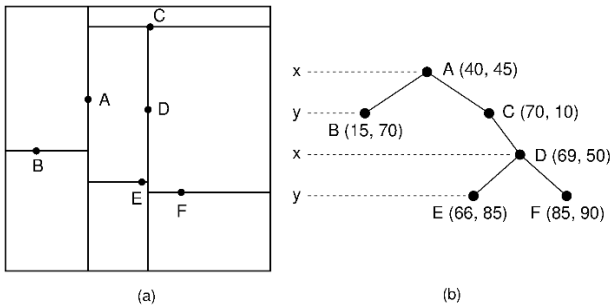

Figure 13: Two representations of a k-d tree. The left is the physical graphical representation, whereas the right is the tree representation ('15.4. KD Trees')

Each node, *A*, of a k-d tree stores the multi-dimensional value, left and right child, and the number of nodes where *A* is an ancestor.

2.3.1 K-d Trees Insertion:

Similar to a BST, we first check if the current node, node, is empty. If it is, that means that we found where to place it. Otherwise, we continue traversing along the tree. curDim is set to depth % KDNode.DIMENSIONS to account for if the depth exceeds the number of dimensions considered. A k-d is traversed the exact same way as a BST – an if statement is used to check the magnitude of the numbers.

2.3.2 K-d Tree Deletion:

Again, the deletion algorithm for a k-d follows similar logic to a BST. However, as stated above the only exception is the consideration of extra dimensions. Let the deleted node be at depth d. In the left and subtree tree following the deletion, only the parameter in dimension d is analyzed.

Additionally, we also have to consider the following edge-cases:

1. Desired node for deletion is not in the tree
2. Desired node for deletion is the root node in a tree where the root node is the only node

Case 1 is checked by traversing the entire tree and verifying the desired node contained in it. Case 2 is accounted for by checking the number of nodes left in the tree for deletion. These are considered in the following code-snippet.

2.3.3 K-d Tree Rebalancing:

While there exists no implicit way to rebalance a k-d tree, the rebalancing algorithm will run periodically every ten operations. Similar to quadtrees, only operations that will count are the insertion of a node and the deletion of a node.

Let the current dimension being evaluated by d. To form a balanced tree, we want half of the nodes to be on the left and half of them to be on the right. Therefore, we find the median m of these nodes in the dimension d. We then recursively generate the left and right subtree by performing the same median finding algorithm for the points less than m in dimension d and those greater than m in dimension d.

2.3.4 K-d Tree Nearest Neighbor Search:

Consider the geometric interpretation of a k-d tree in Figure 13. We first consider the case where A is located inside a bounded box, R. This boundary can be found by traversing the tree until a leaf is reached. While traversing this, we maintain the smallest path to any point in the boundary box. However, we also have to consider the points outside the boundary boxes. For example, in the case below, the nearest neighbor lies outside R.

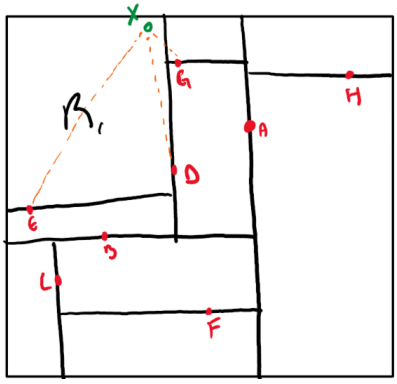

Figure 14: Consider the example where we are trying to find the nearest neighbors of X. Despite being in region R_1, G (which lies outside R_1) is the closest neighbor

These cases will be considered by returning from leaf to root after we finish our initial traversal. When this second traversal happens, we go into a new subtree if the distance to the border is less than the smallest distance. This is

because it may be possible for a point not in R to be closer to A than any point in R as shown in Figure 14. To separate between the first and second traversal in the code, branchOne and branchTwo are defined. branchTwo is only considered if its distance to the border is less than the minimum distance as mentioned before.

2.3.5 K-d Tree Range Query:

To find the range query in a k-d tree, we will propagate the query into applicable regions. This will ensure each region is accounted for while pruning the total search space. Consider the geometrical interpretation of the k-d tree again. We have three cases where our node lies in respect to the query region:

1. The node lies inside the query space
2. The node is less than the query space
3. The node is greater than the query space

Figure 15: Consider the above diagram in dimension i. B_{i_l} and B_{i_u} denotes the lower and upper bound, respectively. Case 1 looks into B; Case 2 looks into A; Case 3 looks into C.

For case 1, the query space is divided by the node to form two new query spaces, one on the left and on the right. Thus, both children need to be considered.

For case 2, the left child does not need to be considered as we know it is less than the current node, so it can't lie in the subspace. Therefore, only the right child is considered. Similarly, we use the same logic for case 3, but instead the right child is not checked, and the left child is

The total amount of nodes residing inside the children are than added to the total of the parent node. Finally, we check the parent node and see if it is contained inside the boundary in constant time. The final sum is outputted as the answer to our range query.

2.4 Complexity of K-d trees and Quadtrees:

For the following it is assumed a well-balanced tree structure is used. If not, the operations will all have a worst case of $O(n)$.

2.4.1 Addition and Deletion:

Both structures follow the same algorithm as BST for addition and deletion. Thus, the complexity for both is an average of $O(\log n)$. While there are checks for both data structures, this is a constant time check, therefore disregarded.

2.4.2 Nearest Neighbor:

Both structures take $O(\log n)$ to find the root node. However, as we recurse up the tree, we may have to go into other nodes. Therefore, the complexity may vary significantly based on the initial bifurcation of the set of points.

2.4.3 Range Query:

The main cost lies in how many cells a boundary intersects, as we can determine the range query of a cell lying completely in or outside the boundary in $O(1)$. It is accepted that the number of average cells intersected is:

$$O\left(n^{1-\frac{1}{k}}\right) = O(\sqrt{n})$$

k denotes the number of dimensions, which in this case is 2 (Aboulnaga, Chanzy et al.)

3. Hypothesis:

While the time complexity of both algorithms is identical, I hypothesize that the time and space complexity of a k-d tree will have a faster experimental time complexity than a quadtree. This is because the nodes in the tree of a k-d tree physically represent nodes, whereas the nodes in a k-d tree represent square regions that only contain the points in their leaves. Not only will this increase the space needed to store the values, but also the time needed to travel through this extended space. Additionally, the extra space needed will become extra time for rebalancing algorithms in a quadtree compared to a k-d tree.

I believe that by plotting the time elapsed against the total operations will result in a logarithmic graph for every operation other than range queries which will have a square root graph. This is because assuming the average is a balanced tree, each operation should have a max of $O(\log n)$.

4. Methodology:

4.1 Independent Variables:

These variables will refer to those that I changed in the experiment too better evaluate the question. I will be changing the number of operations performed (addition, deletion, range queries, and nearest neighbors) on both data structures. These operations will be performed in order of addition, deletion, range queries, and nearest neighbor. The input will be randomized by using Java's Random Class but will stay consistent across both data structures. Additionally, I will be testing the rebalancing feature of each data structure, by toggling them on and off, to analyze the difference with and without them.

4.2 Dependent Variables:

The only variable that will be measured will be the time taken to run these operations. This will be measured with Java's `nanoTime()` method, the most precise measurement feature in Java's System library.

4.3 Controlled Variables:

Table 1: Controlled Variables

Variable	Description and Specifications (if applicable)	Effect if Not Controlled
Computer and Operating System	This experiment will be run on a Dell Inspiron 16 with a Windows 11 Operating System	A different environment may lead to different processes being run. This may cause minute differences in times as different environments may use different methods to run the same procedure.
Software Environment	Java Version: 21.0.3 Integrated Development Environment: Visual Studio Code Java Development Kit: OpenJDK 64-bit Server VM	
Input Data	The same input data will be fed into both data structures. This input data is generated through Java's Random class.	For consistent readings and for the data structures to be evaluated, they must be considered on the same testing basis.
Algorithm and Functions Called	The same algorithms and functions located in Appendix A will be run	Different classes will cause different runtimes, even if the

		theoretical time complexity of each is the same.

4.4 Test Function:

For organizational purposes and consistent data a separate generator and driver class is used to run the code. The generator class `RandomGenerator` creates loads the desired operations into an output file. This file is then read by the driver class `Test`. Each operation is timed with Java's `nanoTime()` and added to a totaled time across all operations.

4.5 Procedure:

The steps of this experiment are as follows:

1. Build the code in the `RandomGenerator` class to generate the `data.out` file.
2. In the `KDTrees` and `QuadTrees` class change CHECKPOINT to 10 or 100 or comment out the `checkOp()` method inside the add and delete node in both classes to evaluate their performance
3. Uncomment the time methods in the add method
4. Run the Test class and read the results in from the generated output file
5. Comment the time methods in the add method
6. Repeat steps 3 through 5 for the deletion, nearest neighbor, and range query method.
7. Repeat steps 2 through 6 for Trial 2 and 3

5. Data Processing:

The data considered will only be the average time across all the trials conducted. In the following tables and graphs, QuadTime 1, 2, 3 will refer to quadtrees that are balanced every 10, 100, and never operations, respectively. Similarly, K-D Tree 1, 2, 3 will refer to k-d trees that are balanced every 10, 100, and never operations. The different balances are not considered for range and nearest neighbor queries, because both don't change the structure of the existing graph

5.1 Graph of Time Against Operation Count:

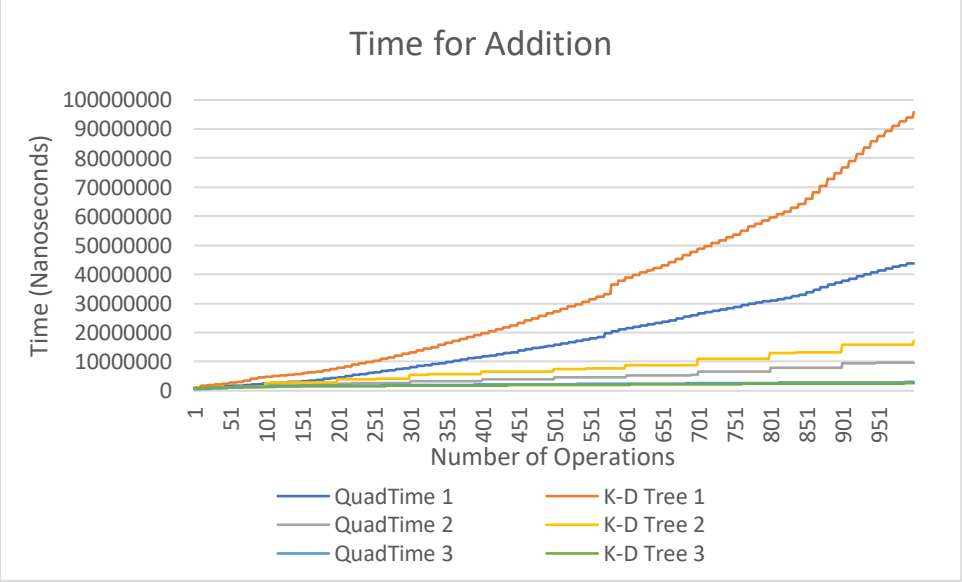

Figure 16: Graph of insertion time (y) against operation count (x)

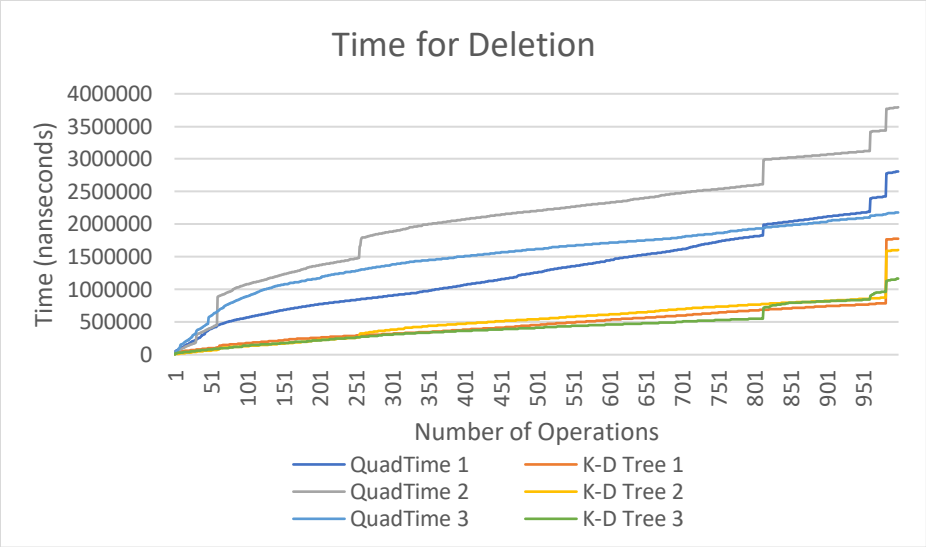

Figure 17: Graph of deletion time (y) against operation count (x)

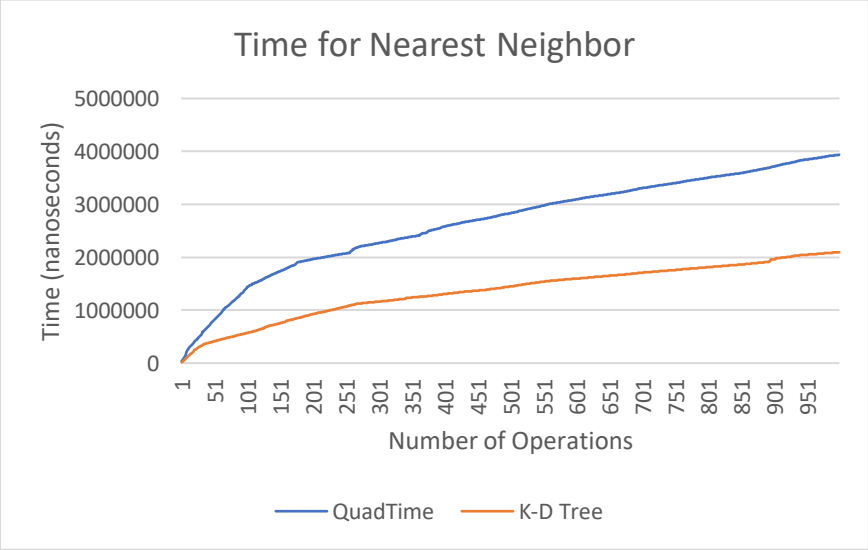

Figure 18: Graph of nearest neighbor search time (y) against operation count (x)

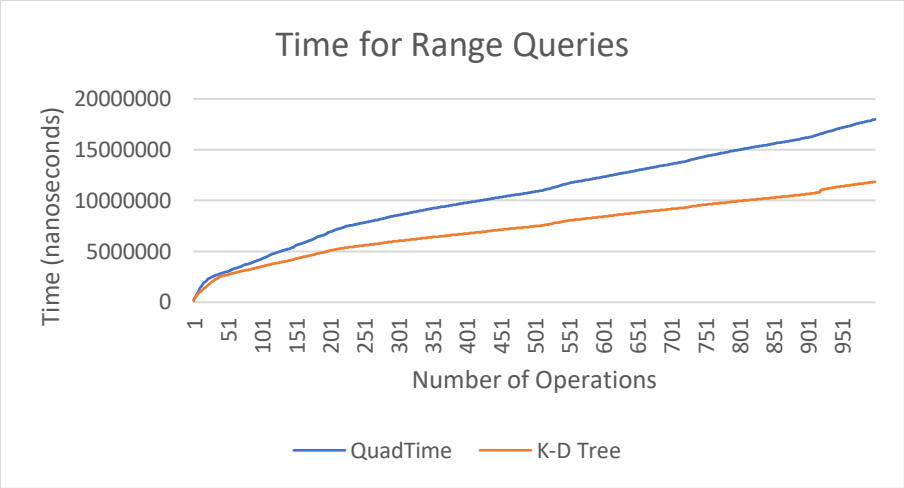

Figure 19: Graph of range query time (y) against operation count (x)

5.2 Statistical Analysis:

To find the correlation between the aforementioned graphs and a logarithmic curve, regression was used to determine the best fitting logarithmic and square root curve. Following this, the coefficient of determination is calculated. This processing is all done on Microsoft Excel.

Table 2: Coefficient of determination (r^2) of each operation with the respective data structure

Operation	QuadTime 1	KD-Tree 1	QuadTime 2	K-D Tree 2	QuadTime 3	K-D Tree 3
Addition (logarithmic)	0.97725	0.94413	0.96110	0.96873	0.92885	0.97696
Deletion (logarithmic)	0.97459	0.78991	0.91815	0.90093	0.88959	0.91603
Nearest Neighbor (logarithmic)					0.92986	0.94779
Range Query (square root)					0.92233	0.90984

6. Results:

From 5.2, it is evident that both data structures follow logarithmic time complexities and square root complexity as predicted in the hypothesis based on the high coefficients of determination. Note that Figure 16 may not look like a logarithmic curve, but this is because of the rate of y increasing at a faster rate than y that distorts the graph.

Additionally, for nearly every operation the k-d tree performed better than the quadtree with the same rebalancing rate. This was however not the case with the addition operation. I believe the cause of the rebalancing operation: the time taken to rebalance exceeds the time taken to add a node. Because the rebalancing takes $O(n \log n)$, whereas adding in an unbalanced tree takes $O(n)$, a minimum of $\log n$ adding operations could be done in the span it takes to rebalance. Since the input is uniformly random, the case of continually taking $O(n)$ to add a node is extremely unlikely.

The issue of rebalancing is again demonstrated in both Figure 16 and 17. The graph maintains a smooth linear form between any two rebalances. However, when a rebalance occurs, the time exhibits a jump-like behavior. This demonstrates that the time needed to rebalance is not in proportion to operations. If it was in proportion, these graphs would contain smoother features that make the impact of rebalancing less visible. Thus, even though the

rebalancing algorithm may result in a more efficient tree, the rebalancing time inhibits better performance. This refutes part of my hypothesis, where I thought any rebalancing efforts would result in optimized behavior.

The difference in performance between the two is best shown in Figure 18 and 19, where the time taken by the quadtree is nearly double that of the k-d tree.

7. Limitations:

The first limitation to this study is the balancing that is used. This paper rebalances trees after a constant number of operations. However, this overlooks the possibility that the tree doesn't need to rebalance. Past research has shown that there exists cases were unbalanced trees sometimes perform better than balanced trees, such as fewer nodes or randomly generated sequences (Pfaff, Hofri and Shachnai). Thus, it may not be the case that the operations are slower after the rebalance, but rather the rebalance is slowing the operations. Additionally, it may be the rebalancing algorithm that is being used that slows down the process because both the k-d tree and quadtree are completely reconstructed after the rebalancing algorithm is called.

Furthermore, another limitation is the total number of operations being tested. This may impact the data collected as the long-term impact of using an unbalanced tree and balanced tree are unable to be seen. I believe that for an increasing amount of operations, a balanced tree will be more efficient than an unbalanced one because of the total depth that will occur in an unbalanced tree. The number of operations was not increased because the time may not have been calculated correctly. As a double was used to store the time and nanoseconds being a immensely small, the variable of time may have overflowed, which would have destroyed the validity of any such data.

8. Conclusion:

This experiment aimed to analyze the efficiency between k-d trees and quadtrees in two-dimensional space by recording the total time taken by both to complete the same set of operations. As expected from the theory explained in Section 2, every operation had a strong correlation to its logarithmic regression and square root regraph as shown in Table 2. This demonstrated that every operation had a logarithm factor associated with it. Additionally, through these graphs, it was made apparent that k-d trees perform far better than quadtrees for the given set of operations. Furthermore, the impact of balancing each data structure was looked at by setting the rebalancing algorithm to run every given number of operations. The test input was generated randomly across three trials to ensure the fairness of each input.

From the results found, I conclude that for randomized data where the total amount of operations is less than 1000, the unbalanced k-tree is more optimal than the balanced k-d tree, and balanced/unbalanced quadtree. However, for cases where the amount of operations exceeds 1000, this may not be the case as the depth of the unbalanced tree becomes too large.

A future extension of this experiment may consider using other rebalancing techniques that don't rely on a set number of operations to rebalance, such as lazy propagation. By doing so, the balancing limitation of this experiment may be removed. Furthermore, extending this experiment into further dimensions may be of practical usage to determine the best way for multi-dimensional data to be stored in the future.

Works Cited:

"15.3. The PR Quadtree — CS3 Data Structures & Algorithms." *Opendsa-Server.cs.vt.edu*, opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/PRquadtree.html.

"15.4. KD Trees — CS3 Data Structures & Algorithms." *Opendsa-Server.cs.vt.edu*, opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/KDtree.html.

"Binary Search Tree Visualization." *Www.cs.usfca.edu*, www.cs.usfca.edu/~galles/visualization/BST.html

Blum, Manuel, et al. Time Bounds for Selection. Vol. 7, no. 4, 1 Aug. 1973, pp. 448–461, https://doi.org/10.1016/s0022-0000(73)80033-9.

Eldridge, Stephen. "Time Complexity | Definition, Examples, & Facts | Britannica." *Www.britannica.com*, 7 Mar. 2023, www.britannica.com/science/time-complexity.

Hofri, Micha, and Hadas Shachnai. "Efficient Reorganization of Binary Search Trees." *Algorithmica*, vol. 31, no. 3, Nov. 2001, pp. 378–402, https://doi.org/10.1007/s00453-001-0057-z.

Moore, Karleigh, and Geoff Pilling. "Median-Finding Algorithm | Brilliant Math & Science Wiki." Brilliant.org, brilliant.org/wiki/median-finding-algorithm/.

Pfaff, Ben. "Performance Analysis of BSTs in System Software." *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, no. 1, 1 June 2004, p. 410, https://doi.org/10.1145/1012888.1005742.

"Red/Black Tree Visualization." *Www.cs.usfca.edu*, www.cs.usfca.edu/~galles/visualization/RedBlack.html.

Appendices:

Appendix A: Tree and Node Code:

A.1: KDTrees.java

```
package Structures.KDTree;

import java.util.*;
import java.io.*;

public class KDTrees{
    public static final int CHECKPOINT = 100;

    private KDNode root;
    private int total = 0;
    private int op = 0;

    //KD TREE OPERATIONS
    public void insertNode(double[] vals){
        if(findNode(vals)){ //IF IT ALREADY EXISTS YOU DON'T NEED TO ADD IT
            return;
        }
        KDNode node = insertNode(root, vals, 0);
        if(root == null){
            root = node;
        }
        op++;
        total++;

        checkOp();
    }
    public void deleteNode(double[] vals){
        if(!findNode(vals)){
            return;
```

```
    }
    deleteNode(root, vals, 0);
    if(total == 1){
        root = null;
    }
    total--;
    op++;

    checkOp();
}
public boolean findNode(double[] vals){
    return findNode(root, vals, 0);
}
public double[] nearestNeighbor(double[] vals){
    return nearestNeighbor(root, vals, null, Double.MAX_VALUE, 0).getVal();
}
public int rangeQuery(double[][] vals){
    double[][] bounds = new double[KDNode.DIMENSIONS][2];

    for (int i = 0; i < bounds.length; i++) {
        bounds[i][0] = findMin(root, 0, i)[i];
        bounds[i][1] = findMax(root, 0, i)[i];
    }

    return rangeQuery(root, vals, bounds, 0);
}

//NODE INSERTION
private KDNode insertNode(KDNode node, double[] vals, int depth){
    if(node == null){
        return new KDNode(vals);
    }
```

```
    int curDim = depth % KDNode.DIMENSIONS;
    node.setAmt(node.getAmt() + 1);

    if(vals[curDim] <= node.getVal()[curDim]){
        node.setLeft(insertNode(node.getLeft(), vals, depth + 1));
    }else{
        node.setRight(insertNode(node.getRight(), vals, depth + 1));
    }

    return node;
}

//NODE DELETION
private KDNode deleteNode(KDNode node, double[] vals, int depth){
    if(node == null){
        return null;
    }

    int curDim = depth % KDNode.DIMENSIONS;

    if(isEquals(node.getVal(), vals)){
        if(node.getLeft() == null && node.getRight() == null){
            return null;
        }
        if(node.getLeft() == null){
            double[] min = findMin(node.getRight(), depth + 1, curDim);
            node.setVals(min);

            node.setRight(deleteNode(node.getRight(), min, depth + 1));
        }else{
            double[] max = findMax(node.getLeft(), depth + 1, curDim);
```

```
            node.setVals(max);


            node.setLeft(deleteNode(node.getLeft(), max, depth + 1));
        }
    }else{
        if(vals[curDim] <= node.getVal()[curDim]){
            deleteNode(node.getLeft(), vals, depth + 1);
        }else{
            deleteNode(node.getRight(), vals, depth + 1);
        }
    }
    node.setAmt(((node.getLeft() == null) ? 0 : node.getLeft().getAmt()) + ((node.getRight() == null) ? 0 :
node.getRight().getAmt()) + 1);
    return node;
  }


  //NODE DETECTION
  private boolean findNode(KDNode node, double[] vals, int depth){
    if(node == null){
        return false;
    }
    if(isEquals(node.getVal(), vals)){
        return true;
    }


    int curDim = depth % KDNode.DIMENSIONS;


    if(vals[curDim] <= node.getVal()[curDim]){
        return findNode(node.getLeft(), vals, depth + 1);
    }else{
        return findNode(node.getRight(), vals, depth + 1);
    }
```

```
    }

    //NEAREST NEIGHBOR
    private KDNode nearestNeighbor(KDNode node, double[] vals, KDNode bestNode, double bestDist, int
depth){
        if(node == null){
            return bestNode;
        }

        double curDist = eucDistance(node.getVal(), vals);

        if(bestNode == null || curDist < bestDist){
            bestNode = node;
            bestDist = curDist;
        }

        int curDim = depth % KDNode.DIMENSIONS;
        KDNode branchOne = null;
        KDNode branchTwo = null;

        if(vals[curDim] <= node.getVal()[curDim]){
            branchOne = node.getLeft();
            branchTwo = node.getRight();
        }else{
            branchOne = node.getRight();
            branchTwo = node.getLeft();
        }

        bestNode = nearestNeighbor(branchOne, vals, bestNode, bestDist, depth + 1);
        bestDist = eucDistance(vals, bestNode.getVal());

        if(Math.abs(node.getVal()[curDim] - vals[curDim]) < bestDist){
```

```
            bestNode = nearestNeighbor(branchTwo, vals, bestNode, bestDist, depth + 1);
            bestDist = eucDistance(vals, bestNode.getVal());
        }
        return bestNode;
    }

    //RANGE QUERIES
    private int rangeQuery(KDNode node, double[][] vals, double[][] bounds, int depth){
        if(node == null){
            return 0;
        }

        int count = 0;
        int curDim = depth % KDNode.DIMENSIONS;

        if(contains(bounds, vals)){
            return node.getAmt();
        }

        //CHECK YOUR LEFT
        if(vals[curDim][0] <= node.getVal()[curDim]){
            double original = bounds[curDim][1];
            bounds[curDim][1] = node.getVal()[curDim];
            count += rangeQuery(node.getLeft(), vals, bounds, depth + 1);
            bounds[curDim][1] = original;
        }

        //CHECK YOUR RIGHT
        if(node.getVal()[curDim] <= vals[curDim][1]){
            double original = bounds[curDim][0];
            bounds[curDim][0] = node.getVal()[curDim] + 1;
            count += rangeQuery(node.getRight(), vals, bounds, depth + 1);
```

```
            bounds[curDim][0] = original;
        }

        //CHECK YOUR NODE
        if(contains(node.getVal(), vals)){
            count++;
        }
        return count;
    }

    //HELPER FUNCTIONS
    private boolean isEquals(double[] a, double[] b){
        assert a.length == b.length;

        for (int i = 0; i < a.length; i++) {
            if(a[i] != b[i]){
                return false;
            }
        }
        return true;
    }
    public double eucDistance(double[] a, double[] b){
        assert a.length == b.length;

        double dist = 0;
        for (int i = 0; i < a.length; i++) {
            dist += (a[i] - b[i]) * (a[i] - b[i]);
        }
        dist = Math.sqrt(dist);
        return dist;
    }
    private List<double[]> collectPoints(){
```

```
    ArrayList<double[]> cur = new ArrayList<>();
    Stack<KDNode> dfs = new Stack<>();
        dfs.add(root);

    while(!dfs.isEmpty()){
        KDNode x = dfs.pop();

        if(x == null){
            continue;
        }

        cur.add(x.getVal());
        dfs.add(x.getRight());
        dfs.add(x.getLeft());
    }

    return cur;
}
private KDNode rebalanceTree(List<double[]> points, int depth){
    if(points.isEmpty()){
        return null;
    }

    int medianIdx = points.size()/2;

    int curDim = depth % KDNode.DIMENSIONS;
    double[] medianPoint = quickSelect(points, medianIdx, curDim);
    KDNode median = new KDNode(medianPoint);

    List<double[]> leftPoints = new ArrayList<>();
    List<double[]> rightPoints = new ArrayList<>();
```

```
    for(double[] e : points){
        if(isEquals(e, median.getVal())){
            continue;
        }
        if(e[curDim] <= median.getVal()[curDim]){
            leftPoints.add(e);
        }else{
            rightPoints.add(e);
        }
    }

    KDNode left = rebalanceTree(leftPoints, depth + 1);
    KDNode right = rebalanceTree(rightPoints, depth + 1);

    median.setLeft(left);
    median.setRight(right);
    median.setAmt(leftPoints.size() + rightPoints.size() + 1);

    return median;
}
private double[] quickSelect(List<double[]> points, int k, int curDim){
    if(points.size() == 1){
        return points.get(0);
    }

    double[] pivot = points.get(points.size()/2);
    List<double[]> lowPoints = new ArrayList<>();
    List<double[]> highPoints = new ArrayList<>();
    List<double[]> equalPoints = new ArrayList<>();

    for(double[] e : points){
        if(e[curDim] < pivot[curDim]){
```

```
            lowPoints.add(e);
        }else if(e[curDim] == pivot[curDim]){
            equalPoints.add(e);
        }else{
            highPoints.add(e);
        }
    }
    if(k < lowPoints.size()){
        return quickSelect(lowPoints, k, curDim);
    }else if(k < lowPoints.size() + equalPoints.size()){
        return pivot;
    }else{
        return quickSelect(highPoints, k - lowPoints.size() - equalPoints.size(), curDim);
    }
}
private double[] findMax(KDNode node, int depth, int axis){
    int curDim = depth % KDNode.DIMENSIONS;

    if(node == null){
        return null;
    }
    if(curDim == axis){
        if(node.getRight() == null){
            return node.getVal();
        }else{
            return findMax(node.getRight(), depth + 1, axis);
        }
    }else{
        double[] leftMax = findMax(node.getLeft(), depth + 1, axis);
        double[] rightMax = findMax(node.getRight(), depth + 1, axis);
        double[] curMax = node.getVal();
```

```
            return maxPoint(maxPoint(leftMax, rightMax, axis), curMax, axis);
        }
    }
    private double[] findMin(KDNode node, int depth, int axis){
        int curDim = depth % KDNode.DIMENSIONS;

        if(node == null){
            return null;
        }
        if(curDim == axis){
            if(node.getLeft() == null){
                return node.getVal();
            }else{
                return findMin(node.getLeft(), depth + 1, axis);
            }
        }else{
            double[] leftMin = findMin(node.getLeft(), depth + 1, axis);
            double[] rightMin = findMin(node.getRight(), depth + 1, axis);
            double[] curMin = node.getVal();

            return minPoint(minPoint(leftMin, rightMin, axis), curMin, axis);
        }
    }
    private double[] maxPoint(double[] a, double[] b, int axis){
        if(a == null){
            return b;
        }else if(b == null){
            return a;
        }
        if(a[axis] > b[axis]){
            return a;
        }else{
```

```
            return b;
        }
    }
    private double[] minPoint(double[] a, double[] b, int axis){
        if(a == null){
            return b;
        }else if(b == null){
            return a;
        }
        if(a[axis] < b[axis]){
            return a;
        }else{
            return b;
        }
    }
    private void checkOp(){
        if(op % CHECKPOINT == 0){
            List<double[]> points = collectPoints();
            root = rebalanceTree(points, 0);
        }
    }
    private boolean contains(double[][] a, double[][] b){ //IF A FOLLOWS ALL THE VALUES OF B
        for (int i = 0; i < a.length; i++) {
            if(!(b[i][0] <= a[i][0] && a[i][1] <= b[i][1])){
                return false;
            }
        }
        return true;
    }
    private boolean contains(double[] a, double[][] b){
        for (int i = 0; i < b.length; i++) {
            if(!(b[i][0] <= a[i] && a[i] <= b[i][1])){
```

```
            return false;
        }
    }
    return true;
}
public void getTree(){
    Stack<KDNode> cur = new Stack<>();
        cur.add(root);
    while(!cur.isEmpty()){
        KDNode node = cur.pop();
        if(node == null){
            System.out.println(node);
            continue;
        }

        System.out.println(Arrays.toString(node.getVal()) + " " + node.getAmt());
        cur.add(node.getRight());
        cur.add(node.getLeft());
    }
}
}
```

A.2: KDNode.java

```
package Structures.KDTree;


public class KDNode{
    public static final int DIMENSIONS = 2;


    private KDNode left, right;
    private double[] val;
    private int amt;


    //INIT
```

```
public KDNode(double[] val_){
    val = new double[DIMENSIONS];
    for (int i = 0; i < DIMENSIONS; i++) {
        val[i] = val_[i];
    }

    left = null;
    right = null;
    amt = 1;
}

//SETTER FUNCTIONS
public void setLeft(KDNode node){
    left = node;
}
public void setRight(KDNode node){
    right = node;
}
public void setVals(double[] val_){
    for (int i = 0; i < val_.length; i++) {
        val[i] = val_[i];
    }
}
public void setAmt(int val_){
    amt = val_;
}

//GETTER FUNCTIONS
public double[] getVal(){
    return val;
}
public KDNode getLeft(){
```

```
        return left;
    }
    public KDNode getRight(){
        return right;
    }
    public int getAmt(){
        return amt;
    }
}
```

A.3: QuadTrees.java

```
package Structures.Quadtree;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Stack;

public class QuadTrees {
    public static final int CHECKPOINT = 100;

    public static final double MAX_BOUND_X = 2000;
    public static final double MAX_BOUND_Y = 2000;

    //ADDED FOR CONVIENCE: USED TO DISTINGUISH BETWEEN THE DIFFERENT SIDES OF THE RECTANGLE FOR THE SUBDIVISION
    private static final int[] regionXIdx = {0, 2, 2, 0};
    private static final int[] regionYIdx = {1, 1, 3, 3};

    public QuadNode root;
    public int op = 0;

    //QUADTREE OPERATIONS
    public void insertNode(double[] vals){
```

```
    if(root == null){
        root = new QuadNode(vals, MAX_BOUND_X, MAX_BOUND_Y);
        root.addVal(vals);
        return;
    }

    insertNode(root, vals);
    op++;

    checkOp();
}
public void deleteNode(double[] vals){
    if(root == null || !contains(root, vals)){
        return;
    }

    deleteNode(root, null, vals);
    op++;

    checkOp();
}
public double[] nearestNeighbor(double[] vals){
    return nearestNeighbor(root, vals, null, Double.MAX_VALUE);
}
public int rangeQuery(double[][] bounds){
    return rangeQuery(root, bounds);
}

//QUADTREE INSERTION
private void insertNode(QuadNode node, double[] vals){
    if(!inBounds(node.getRect(), vals)){
        return;
```

```
    }
    node.setAmt(node.getAmt() + 1);

    if(node.getVal().size() < QuadNode.CAPACITY){
        node.addVal(vals);
        return;
    }

    double[] mids = getMid(node);
    double[] rect = node.getRect();

    if(!node.getDivide()){
        node.setDivideTrue();
        for (double[] e : node.getVal()) {
            add(mids[0], mids[1], e, rect, node);
        }
    }
    add(mids[0], mids[1], vals, rect, node);
}

//QUADTREE DELETION
private QuadNode deleteNode(QuadNode node, QuadNode par, double[] vals){
    if(node == null || node.getAmt() == 0){
        return null;
    }

    int contain = contains(node.getVal(), vals);

    if(contain != -1 && !node.getDivide()){
        node.getVal().remove(contain);
        if(par != null){
            int sum = 0;
```

```
            for(QuadNode e : par.getChildren()){
                if(e == null){
                    continue;
                }
                sum += e.getVal().size();
            }
            if(sum <= QuadNode.CAPACITY){
                par.setDivideFalse();
                par.clearVal();
                for(QuadNode e1 : par.getChildren()){
                    if(e1 == null){
                        continue;
                    }
                    for(double[] e2 : e1.getVal()){
                        par.addVal(e2);
                    }
                    e1.clearVal();
                }
            }
        }
    }else{
        if(contain != -1){
            node.getVal().remove(contain);
        }

        double[] mids = getMid(node);
        int idx = findRegion(vals, mids[0], mids[1]);

        deleteNode(node.getChildren()[idx], node, vals);
    }
    int sum = 0;
    for(QuadNode e : node.getChildren()){
```

```
        if(e == null){
            continue;
        }
        if(e.getVal().size() > 0){
            sum += e.getAmt();
        }
    }
    node.setAmt(sum);
    return node;
}

//QUADTREE NEARESTNEIGHBOR
private double[] nearestNeighbor(QuadNode node, double[] vals, double[] bestNode, double bestDist){
    if(node == null || node.getAmt() == 0){
        return bestNode;
    }

    double[] curBest = minEucDist(node.getVal(), vals);
    double curDist = eucDist(curBest, vals);

    if(bestNode == null || curDist < bestDist){
        bestNode = curBest;
        bestDist = curDist;
    }
    // System.out.println("==" + Arrays.toString(bestNode) +" "  +bestDist);

    if(!node.getDivide()){
        for (int i = 0; i < node.getVal().size(); i++) {
            curDist = eucDist(node.getVal().get(i), vals);
            if(curDist < bestDist){
                bestNode = node.getVal().get(i);
                bestDist = curDist;
```

```
        }
    }

    return bestNode;
}

double[] rect = node.getRect();
double xMid = (rect[0] + rect[2])/2; double yMid = (rect[1] + rect[3])/2;
int idx = findRegion(vals, xMid, yMid);

bestNode = nearestNeighbor(node.getChildren()[idx], vals, bestNode, bestDist);
bestDist = eucDist(bestNode, vals);

if(idx == 0){
    if(Math.abs(xMid - vals[0]) < bestDist){
        bestNode = nearestNeighbor(node.getChildren()[1], vals, bestNode, bestDist);
        bestDist = eucDist(bestNode, vals);
    }
    if(Math.abs(yMid - vals[1]) < bestDist){
        bestNode = nearestNeighbor(node.getChildren()[3], vals, bestNode, bestDist);
        bestDist = eucDist(bestNode, vals);
    }
    if(Math.abs(yMid - vals[1]) < bestDist || Math.abs(xMid - vals[0]) < bestDist){
        bestNode = nearestNeighbor(node.getChildren()[2], vals, bestNode, bestDist);
        bestDist = eucDist(bestNode, vals);
    }
}else if(idx == 1){
    if(Math.abs(xMid - vals[0]) < bestDist){
        bestNode = nearestNeighbor(node.getChildren()[0], vals, bestNode, bestDist);
        bestDist = eucDist(bestNode, vals);
    }
    if(Math.abs(yMid - vals[1]) < bestDist){
```

```
            bestNode = nearestNeighbor(node.getChildren()[2], vals, bestNode, bestDist);
            bestDist = eucDist(bestNode, vals);
        }
        if(Math.abs(yMid - vals[1]) < bestDist || Math.abs(xMid - vals[0]) < bestDist){
            bestNode = nearestNeighbor(node.getChildren()[3], vals, bestNode, bestDist);
            bestDist = eucDist(bestNode, vals);
        }
    }else if(idx == 2){
        if(Math.abs(xMid - vals[0]) < bestDist){
            bestNode = nearestNeighbor(node.getChildren()[3], vals, bestNode, bestDist);
            bestDist = eucDist(bestNode, vals);
        }
        if(Math.abs(yMid - vals[1]) < bestDist){
            bestNode = nearestNeighbor(node.getChildren()[1], vals, bestNode, bestDist);
            bestDist = eucDist(bestNode, vals);
        }
        if(Math.abs(yMid - vals[1]) < bestDist || Math.abs(xMid - vals[0]) < bestDist){
            bestNode = nearestNeighbor(node.getChildren()[0], vals, bestNode, bestDist);
            bestDist = eucDist(bestNode, vals);
        }
    }else{
        if(Math.abs(xMid - vals[0]) < bestDist){
            bestNode = nearestNeighbor(node.getChildren()[2], vals, bestNode, bestDist);
            bestDist = eucDist(bestNode, vals);
        }
        if(Math.abs(yMid - vals[1]) < bestDist){
            bestNode = nearestNeighbor(node.getChildren()[0], vals, bestNode, bestDist);
            bestDist = eucDist(bestNode, vals);
        }
        if(Math.abs(yMid - vals[1]) < bestDist || Math.abs(xMid - vals[0]) < bestDist){
            bestNode = nearestNeighbor(node.getChildren()[1], vals, bestNode, bestDist);
            bestDist = eucDist(bestNode, vals);
```

```
        }
    }
    return bestNode;
}

//QUADTREE RANGE QUERY
private int rangeQuery(QuadNode node, double[][] bounds){
    int count = 0;
    double[] rect = node.getRect();

    if(in(new double[][]{{rect[0], rect[2]}, {rect[3], rect[1]}}, bounds)){
        // System.out.println(Arrays.toString(rect) + " " + node.getAmt());
        return node.getAmt();
    }

    if(!node.getDivide()){
        for(double[] e : node.getVal()){
            boolean inside = true;
            for (int i = 0; i < e.length; i++) {
                if(!(bounds[i][0] <= e[i] && e[i] <= bounds[i][1])){
                    inside = false;
                }
            }
            if(inside){
                count++;
            }
        }
        return count;
    }

    for(QuadNode e : node.getChildren()){
        if(e == null){
```

```
            continue;
        }
        double[] childRect = e.getRect();
        if(intersect(new double[][]{{childRect[0], childRect[2]}, {childRect[3], childRect[1]}}, bounds)){
            count += rangeQuery(e, bounds);
        }
    }

    return count;
}

//HELPER FUNCTIONS
//CHECKS TO SEE IF VALS LIES INSIDEOF RECT
private boolean inBounds(double[] rect, double[] vals){
    return rect[0] <= vals[0] && vals[0] <= rect[2] && rect[3] <= vals[1] && vals[1] <= rect[1];
}

//CHECKS THE REGION AND WHETHER YOU CAN ADD INTO THE REGION
private void add(double xMid, double yMid, double[] vals, double[] rect, QuadNode node){
    int idx = findRegion(vals, xMid, yMid);

    QuadNode next = node.getChildren()[idx];
    if(next == null){
        double[] center = new double[]{(rect[regionXIdx[idx]] + xMid)/2, (rect[regionYIdx[idx]] + yMid)/2};
        node.setChildren(new QuadNode(center, node.getLength()/2, node.getHeight()/2), idx);
    }
    insertNode(node.getChildren()[idx], vals);
}

//FIND THE REGION WHERE VAL IS LOCATED GIVEN THE MIDDLE X AND MIDDLE Y
private int findRegion(double[] vals, double xMid, double yMid){
    boolean left = vals[0] <= xMid;
```

```
    boolean top = vals[1] >= yMid;

    //CHECK EACH REGION
    if(left && top){
        return 0;
    }else if(!left && top){
        return 1;
    }else if(!left && !top){
        return 2;
    }else{
        return 3;
    }
}

//USED TO CHECK IF VALS IS CONTAINED INSIDE ARR
private int contains(ArrayList<double[]> arr, double[] vals){
    for (int i = 0; i < arr.size(); i++) {
        if(arr.get(i)[0] == vals[0] && arr.get(i)[1] == vals[1]){
            return i;
        }
    }
    return -1;
}

//USED TO CHECK IF VALS IS IS THE NODE REGION
public boolean contains(QuadNode node, double[] vals){
    if(node == null){
        return false;
    }
    if(contains(node.getVal(), vals) != -1){
        return true;
    }else{
```

```
        double[] mids = getMid(node);
        int idx = findRegion(vals, mids[0], mids[1]);

        return contains(node.getChildren()[idx], vals);
    }
}

//USED TO CHECK IF ARR1 IS CONTAINED INSIDE OF ARR2
private boolean in(double[][] arr1, double[][] arr2){
    for (int i = 0; i < arr1.length; i++) {
        if(!(arr2[i][0] <= arr1[i][0] && arr1[i][1] <= arr2[i][1])){
            return false;
        }
    }
    return true;
}

//USED TO CHECK IF ARR1 INTERSECTS WITH ARR2 (SAME THING VICE VERSA)
private boolean intersect(double[][] arr1, double[][] arr2){
    for (int i = 0; i < arr1.length; i++) {
        if(Math.max(arr1[i][0], arr2[i][0]) > Math.min(arr1[i][1], arr2[i][1])){
            return false;
        }
    }
    return true;
}

//USED TO FIND THE X-MID AND Y-MID OF A NODE REGION
private double[] getMid(QuadNode node){
    double[] rect = node.getRect();
    double xMid = (rect[0] + rect[2])/2;
    double yMid = (rect[1] + rect[3])/2;
```

```
    return new double[]{xMid, yMid};
}

//FINDS THE MINIMUM DISTANCE INSIDE A REGION TO A POINT
private double[] minEucDist(ArrayList<double[]> points, double[] val){
    double minDist = Double.MAX_VALUE;
    int idx = 0;
    for(int i = 0; i < points.size(); i++){
        double curDistt = eucDist(points.get(i), val);
        if(curDistt < minDist){
            minDist = curDistt;
            idx = i;
        }
    }
    return points.get(idx);
}

//FINDS THE EUCLIDEAN DISTANCE BETWEEN TWO POINTS
public double eucDist(double[] p1, double[] p2){
    double dist = 0;
    for (int i = 0; i < p1.length; i++) {
        dist += (p1[i] - p2[i]) * (p1[i] - p2[i]);
    }
    dist = Math.sqrt(dist);
    return dist;
}

//REBUILD OPERATION
private void checkOp(){
    if(op % CHECKPOINT == 0){
        ArrayList<double[]> points = collectPoints();
        rebalanceTree(points);
```

```
    }
}

//RETURNING ALL THE POINTS INSIDE THE QUADTREE
private ArrayList<double[]> collectPoints(){
    ArrayList<double[]> totalPoints = new ArrayList<>();

    Stack<QuadNode> cur = new Stack<>();
    cur.add(root);

    while(!cur.isEmpty()){
        QuadNode x = cur.pop();

        if(!x.getDivide()){
            for(double[] e : x.getVal()){
                totalPoints.add(e);
            }
        }else{
            for(QuadNode e : x.getChildren()){
                if(e == null){
                    continue;
                }
                cur.add(e);
            }
        }
    }
    return totalPoints;
}
private QuadNode rebalanceTree(ArrayList<double[]> points){
    double[] med = quickSelect(points, points.size()/2, 0);
    root = new QuadNode(med, MAX_BOUND_X, MAX_BOUND_Y);
```

```
    for(double[] e : points){
        insertNode(root, e);
    }
    return root;
}
private double[] quickSelect(ArrayList<double[]> points, int k, int idx){
    if(points.size() == 1){
        return points.get(0);
    }

    double[] pivot = points.get(points.size()/2);

    ArrayList<double[]> lowPoints = new ArrayList<>();
    ArrayList<double[]> highPoints = new ArrayList<>();
    ArrayList<double[]> equalPoints = new ArrayList<>();

    for(double[] e : points){
        if(e[idx] < pivot[idx]){
            lowPoints.add(e);
        }else if(e[idx] == pivot[idx]){
            equalPoints.add(e);
        }else{
            highPoints.add(e);
        }
    }
    if(k < lowPoints.size()){
        return quickSelect(lowPoints, k, idx);
    }else if(k < lowPoints.size() + equalPoints.size()){
        return pivot;
    }else{
        return quickSelect(highPoints, k - lowPoints.size() - equalPoints.size(), idx);
    }
```

```
    }
}
```

A.4: QuadNode.java

```
package Structures.Quadtree;

import java.util.ArrayList;

public class QuadNode {
    public static final int DIMENSIONS = 2;
    public static final int CAPACITY = 4;

    private ArrayList<double[]> val = new ArrayList<>();
    private double[] rect;
    private double length, height;
    private boolean divide;
    private QuadNode[] children;
    private int amt;

    public QuadNode(double[] center_, double length_, double height_){
        rect = new double[]{
            center_[0] - length_/2,
            center_[1] + height_/2,
            center_[0] + length_/2,
            center_[1] - height_/2};

        length = length_;
        height = height_;

        divide = false;

        children = new QuadNode[4];
```

```
    // amt = 1;
}

public void setChildren(QuadNode node, int idx){
    children[idx] = node;
}
public void addVal(double[] val_){
    val.add(val_);
}
public void setDivideTrue(){
    divide = true;
}
public void setDivideFalse(){
    divide = true;
}
public void clearVal(){
    val = new ArrayList<>();
}
public void setAmt(int x){
    amt = x;
}

public ArrayList<double[]> getVal(){
    return val;
}
public double[] getRect(){
    return rect;
}
public QuadNode[] getChildren(){
    return children;
}
public double getLength(){
```

```
        return length;
    }
    public double getHeight(){
        return height;
    }
    public boolean getDivide(){
        return divide;
    }
    public int getAmt(){
        return amt;
    }
}
```

Appendix B: Driver and Random Generator:

B.1 Test.java

```
import Structures.KDTree.KDNode;
import Structures.KDTree.KDTrees;
import Structures.Quadtree.QuadNode;
import Structures.Quadtree.QuadTrees;

import java.util.*;
import java.io.*;

public class Test {
    public static void main(String[] args) throws IOException{
        BufferedReader r = new BufferedReader(new FileReader("data.out"));
        PrintWriter pw = new PrintWriter(new BufferedWriter(new FileWriter("output.out")));

        QuadTrees quadTree = new QuadTrees();
        KDTrees kdTree = new KDTrees();

        double quadTreeTime = 0;
        double kdTreeTime = 0;
```

```
for (int i = 0; i < RandomGenerator.ADDOPERATIONS; i++) {
    StringTokenizer st = new StringTokenizer(r.readLine());
    double[] points = new double[]{Integer.parseInt(st.nextToken()), Integer.parseInt(st.nextToken())};

    double curTime = System.nanoTime();
    quadTree.insertNode(points);
    quadTreeTime += System.nanoTime() - curTime;

    curTime = System.nanoTime();
    kdTree.insertNode(points);
    kdTreeTime += System.nanoTime() - curTime;

    pw.println((i + 1) + " " + quadTreeTime + " " + kdTreeTime);
}

for (int i = 0; i < RandomGenerator.DELETIONOPERATIONS; i++) {
    StringTokenizer st = new StringTokenizer(r.readLine());
    double[] points = new double[]{Integer.parseInt(st.nextToken()), Integer.parseInt(st.nextToken())};

    double curTime = System.nanoTime();
    quadTree.deleteNode(points);
    quadTreeTime += System.nanoTime() - curTime;

    curTime = System.nanoTime();
    kdTree.deleteNode(points);
    kdTreeTime += System.nanoTime() - curTime;

    pw.println((i + 1) + " " + quadTreeTime + " " + kdTreeTime);
}

for (int i = 0; i < RandomGenerator.NNADDOPERATIONS; i++) {
```

```
            StringTokenizer st = new StringTokenizer(r.readLine());
            double[] points = new double[]{Integer.parseInt(st.nextToken()), Integer.parseInt(st.nextToken())};

            double curTime = System.nanoTime();
            quadTree.nearestNeighbor(points);
            quadTreeTime += System.nanoTime() - curTime;

            curTime = System.nanoTime();
            kdTree.nearestNeighbor(points);
            kdTreeTime += System.nanoTime() - curTime;

            pw.println((i + 1) + " " + quadTreeTime + " " + kdTreeTime);
        }

        for (int i = 0; i < RandomGenerator.RNGQOPERATIONS; i++) {
            StringTokenizer st = new StringTokenizer(r.readLine());
            double[][] query = new double[][]{{Integer.parseInt(st.nextToken()), Integer.parseInt(st.nextToken())},
{Integer.parseInt(st.nextToken()), Integer.parseInt(st.nextToken())}};

            double curTime = System.nanoTime();
            quadTree.rangeQuery(query);
            quadTreeTime += System.nanoTime() - curTime;

            curTime = System.nanoTime();
            kdTree.rangeQuery(query);
            kdTreeTime += System.nanoTime() - curTime;

            pw.println((i + 1) + " " + quadTreeTime + " " + kdTreeTime);
        }

        pw.close();
    }
```

```
}
```

B.2 RandomGenerator.java

```
import java.util.*;


import Structures.KDTree.KDNode;


import java.io.*;


public class RandomGenerator {
    public static final int MAX = 1000;
    public static final int ADDOPERATIONS = 1000;
    public static final int DELETIONOPERATIONS = 1000;
    public static final int NNADDOPERATIONS = 1000;
    public static final int RNGQOPERATIONS = 1000;


    public static void main(String[] args) throws IOException{
        PrintWriter pw = new PrintWriter(new BufferedWriter(new FileWriter("data.out")));


        Random gen = new Random();


        StringBuilder addition = new StringBuilder();
        StringBuilder deletion  = new StringBuilder();
        StringBuilder nearestNeighbor = new StringBuilder();
        StringBuilder rangeQueries = new StringBuilder();


        for (int i = 0; i < ADDOPERATIONS; i++) {
            addition.append(-MAX/2 + gen.nextInt(MAX + 1)).append(" ").append(-MAX/2 + gen.nextInt(MAX +
1)).append('\n');
        }
        for (int i = 0; i < DELETIONOPERATIONS; i++) {
            deletion.append(-MAX/2 + gen.nextInt(MAX + 1)).append(" ").append(-MAX/2 + gen.nextInt(MAX +
1)).append(" ==").append('\n');
```

```
        }
        for (int i = 0; i < NNADDOPERATIONS; i++) {
            nearestNeighbor.append(-MAX/2 + gen.nextInt(MAX + 1)).append(" ").append(-MAX/2 +
gen.nextInt(MAX + 1)).append('\n');
        }
        for (int i = 0; i < RNGQOPERATIONS; i++) {
            for (int j = 0; j < KDNode.DIMENSIONS; j++) {
                int a = -MAX/2 + gen.nextInt(MAX + 1); int b = -MAX/2 + gen.nextInt(MAX + 1);
                rangeQueries.append(Math.min(a, b)).append(" ").append(Math.max(a, b)).append(" ");
            }
            rangeQueries.deleteCharAt(rangeQueries.length() - 1);
            rangeQueries.append('\n');
        }
        addition.deleteCharAt(addition.length() - 1);
        deletion.deleteCharAt(deletion.length() - 1);
        nearestNeighbor.deleteCharAt(nearestNeighbor.length() - 1);
        rangeQueries.deleteCharAt(rangeQueries.length() - 1);

        pw.println(addition);
        pw.println(deletion);
        pw.println(nearestNeighbor);
        pw.println(rangeQueries);

        pw.close();
    }
}
```