Comparative Analysis of Quadtrees and kd-Trees

Andres Chavez Armijos

John Neal

1 Introduction and Problem Description

The goal of our project was originally to compare four spatial data structures and two linear data structures as an introduction to each of these data structures. This would have involved a comparison of the amortized analysis of the most basic methods of each of the data structures (find, insert, delete) with the possibility of implementing methods that are significant to each (such as nearest-neighbor query for kd-trees, quad trees, and octrees on randomly generated datasets.

Due to poor planning and time constraints, we ended up only successfully implementing insert and find for both data structures and comparing runtimes. For quad trees a successful nearest neighbor query was run. This report will present the combined results and methods of the spatial data structures implementations. The results of the linear data structures will be reported by Krishna Palle in a separate report.

2 KD-Tree

2.1 Implementation Details

The kd-tree was implemented as a C++ class:

kd_tree.h

```
class kd_tree{
public:
  // constructs empty kd tree
  kd_tree();

  // constructs a one node empty kd tree
  // kd_tree(int* array, int dim);

  // constructs a no node kd_tree with d = dim;
  kd_tree(int dim);
  void print_node(Node* a);
  void insert_node(int* array);
```

```
    void delete_node(int* array);
    Node* find_node(int* array);
private:
public:
    // dimension of the coordinates of each node
    int d;
    Node *root; };
```

The multiple kd_tree constructors were simply for convenience. The constructor that was used in implementation was *kd_tree*(int dim) and then consecutive insertions of nodes would build the tree.

The *find*, *delete*, and *insert* methods all require as input a pointer to the coordinate array of the point to be inserted.

Of course, the implementation of the kd_tree requires a node pointer *root. Node was implemented as a struct:

```
struct Node {

    // "Feature Vector"
    int *coordinates;

    // Left Child
    Node *lo_child;

    // Right Child
    Node *hi_child;

    // DISC
    int DISC;
};

int next_disc(int disc, int d);

Node* insert_or_find_node(Node* Q, int* point, int d, bool insert, int disc);

void find_min_key_in_subtree(Node* Q, Node* Parent_Q, Node* best_min_node, Node*
best_min_node_parent, bool* is_hi, int j_disc);
```

```
Node* find_max_key_in_subtree(Node* Q, Node* Parent_Q, Node* best_max_node, Node*
best_max_node_parent, bool* is_hi, int j_disc);

Node* remove_node(Node* P);
```

The node struct is straightforward. There is a point to the coordinate array (the keys of the node), a pointer to the hi and lo subtrees, and the DISC (discriminator). The discriminator is such that:

Parent->coordinates[DISC] < hi_child->coordinates[DISC]

Parent->coordinates[DISC] $\geq$ lo_child->coordinates[DISC]

Since **insert_node** is basically **find_node** where the node is not found and a node points to a new child node (the inserted node) these two functions were combined. **next_disc** computes the discriminator of a child node, which is used in both **insert_node** and **delete_node**. **delete_node** calls **remove_node** to find the node to be deleted and recursively find a replacement node, delete the replacement node, and point the parent node to the replacement node. In order to do this one must find the minimum or maximum key in the subtree of the node to be deleted, (right and left subtree respectively). For this reason, **find_min_key_in_subtree** and **find_max_key_in_subtree** are called in **remove_node.**

The program time_insert_and_find.cpp produces a .txt file with two columns when the Makefile is run. Each .txt file corresponds to the result of computing build times and random find node times for each size of random datasets and randomly permuted datasets. For further clarification:

3 Quadtree

The main idea behind quadtrees was to generate an efficient storage methodology for 2-dimensional data sets that would facilitate queries regarding the retrieval of specific spatial information. Thus, the classification given to a quadtree is of a spatial data structure. In general, a quadtree can be considered a hierarchical data structure since it is based on the idea of decomposing the spatial data into subregions in a recursive manner (similar to divide and conquer

methods) (Sahni, 2004). As its name suggest, quadtrees are based on the principle of hierarchically decompose the regions of interest into four subregions using a discretization method, which in comparison to KD-trees, generates subregions of equal size or ranges in depending of the type of quadtree used and the type of data that it represents.

Depending on the application, one can fix the resolution of the spatial decomposition beforehand or it can be governed by the properties of the input data such as pixel classification, areas, or boundaries (Samet, 1990). Thus, one can subclassify the quadtree given the data representation that we want to work with. For this project we focused on only 2 of these representations, Point-Region Quadtree and Matrix Region Quadtrees. However, for the sake of the conceptualization of the problem, we start by introducing point quadtrees.

3.1 Point Quadtree

A point based quadtree is a multidimensional generalization of a well-known binary search tree with the difference that each node contains four different child nodes if they are not leaf nodes. The main characteristic of a point-based quadtree is that as its name suggests, each 2D data point is represented as a node in the tree and its child nodes represent data that is located on a sub quadrant with respect to the parent node (Samet, 1984). These sub quadrants are Northeast (NE), Northwest (NW), Southeast (SE), and Southwest (SW). It is important to remark that for the Point-based quadtrees, the center of each sub quadrant is the father node. Thus, the leaf nodes are empty and the resulting sub quadrants are not equally spaced. This can be visualized in Figure 1. The main issue with this kind of quadtree is that the deletion of nodes can be computationally expensive since every data point node contains a child data point in a hierarchical matter. Thus, the deletion of a node requires a replacement which is translated as reinsertion of nodes. The selection of a replacement candidate is not trivial, and its algorithm is not unified. However, the task is easier when the data is uniformly distributed.

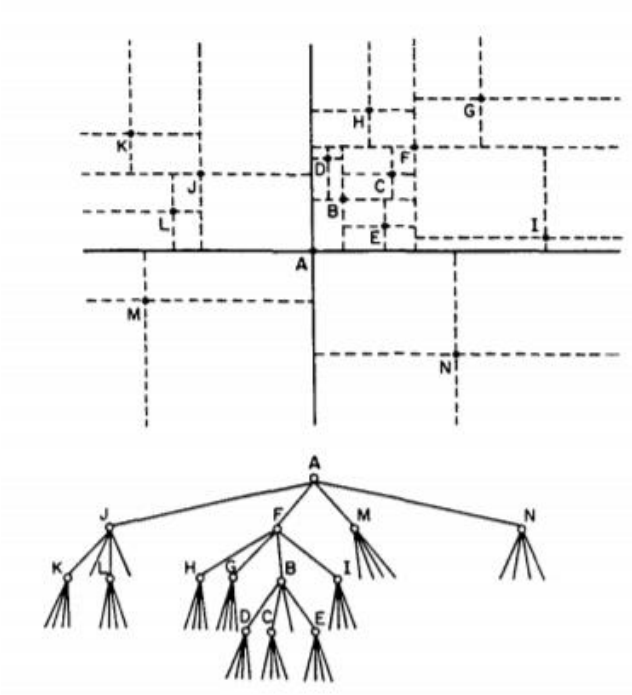


Figure 1 Point-Based Quadtree representation (Samet, 1990)

3.2 Region-Based Quadtrees

Similar to the point quadtrees, one can represent 2-D data points in the exact same way as for the point quadtrees. However, for the region quadtrees, the implementation can be compared to the implementation of a KD-tree, since the KD-tree can be adapted to handle data so the output generates a quadtree. The main difference between a point quadtree and a Point-Region quadtree is that the data nodes are all located on the leaves. Each leaf node is referenced given a subregion. For point-region quadtrees, when a subdivision occurs, every subregion is of the same size. This can be visualized in Figure 2, where each subregion has the same area and dimensions as of its brother subregion.

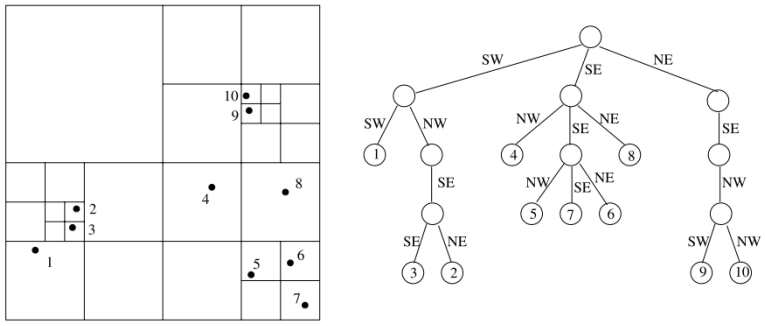


Figure 2 Point Region Quadtree representation (Sahni, 2004)

Allowing the data to be contained only in the leaf nodes simplifies the deletion time since in this case, it is only necessary to delete a leaf node that does not contain any further sub hierarchical structures connected to itself. However, the insertion time increases in comparison to the point quadtree implementation since every parent node needs to be reinserted for every time that a new insertion requires a region subdivision.

3.3 Implementation

Like the KD-tree case, the quadtree was implemented as a c++ class with its members shown as follows:

```
#ifndef _QUADTREE_H
#define _QUADTREE_H

#include <stdio.h>
#include "stack.h"
#include <opencv2/core.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/highgui.hpp>



/* Quadtrees Structure definition */
class quadtree
{
protected:
    // Boundary details
```

```
    Point top_left;
    Point bot_right;
    // Set children
    quadtree *child[4];
    quadtree *Parent;
    // Points
    Stack *pointArray;
    // Tree Metadata
    int capacity;   //Maximum number of nodes per tree
    int size;       // Current size of tree
    // Private function members
    // create additional four children by dividing region 4 times
    void __subdivideQuadPR__();
    void __subdivideQuadMXR__();
    // Check if pt data is within boundary
    bool __inBoundary__(Point Pt);
    // Check if bounding boxes are intersected
    bool __instersectsBoundary__(Point top_left2, Point bot_right2);
    // Query points within region
    void __query__(Point topLeft, Point botRight, Stack* X);
    int __distance__(Point P1, Point P2);

public:
    quadtree(Point topLeft, Point botRight, int capacity);
    quadtree( Point topLeft, Point botRight, int capacity, quadtree* Father);
    ~quadtree();
    /* ------Util Functions-----*/
    // insert a node onto tree
    int insert_nodeMXR(Node *Pt); //********* Could be node pointer
    int insert_nodePR(Node *Pt); //********* Could be node pointer
    // Delete Node
    void delete_node(Node *Pt);
    // Print Quadtree
    void printCapacities(int depth = 0);
    void printTree(int depth = 0);

    /* ----- Return Functions -----*/
    // Lookup for pt node
    Node searchPt(Point Pt);
    // Extract capacity
    int getCapacity() const;
    // Extract current Size
    int getSize() const;
    // Print Values within Region
    void inRegion(Point topLeft, Point botRight);
```

```
    // Nearest neighbour
    Node nearestNeighbour(Point Pt);
    // Look for smallest quadrant containing point Pt
    quadtree* __searchQuadrant__(Point Pt);
    //  Nearest Points
    void __searchNearestPts__(Point Pt, Stack *S);
    void NearestPt(Point Pt, Stack *S);
    // Draw function
    void show(cv::Mat *window);
    void drawInRegion(cv::Mat *img, Point topLeft, Point botRight);
};

#endif /*_QUADTREE_H*/
```

As it can be seen, the code includes a function to allow the subregions to be drawn and includes a function that allows one to generate a query given a user defined area. This query would return all the points contained within the specified area. Additionally, every node contained within a Quadtree has a structure member that allows the user to store additional data such as pixel intensity or a name.

Since the implementation taken for the quadtree is not only for a single point representation, a stack structure was implemented to allow the storage, retrieval, and generation of arrays more efficiently. This is shown below.

```
#ifndef _STACK_H
#define _STACK_H

/* Node data structure for 2D representation */
typedef struct Point
{
    int x;
    int y;
} Point;
Point * createPoint(int x, int y);
// Check if two points are the same
bool isPointEqual(Point Pt1, Point Pt2);

// Node Data
typedef struct Node
{
```

```
    Point coordinates;
    int data;
} Node;
Node * createNode(Point point, int data);


typedef struct Stack
{
        int capacity;
        int size;
        int top;
        //int rear;
        Node *elements;
}Stack;

Stack * createStack(int maxElements);
void Push(Stack *S,Node element);
Node Pop(Stack *S);
Node Show(Stack *S, int Pos);


#endif /*_STACK_H*/
```

A visual representation of a point region quadtree and a point quadtree was implemented and its shown in Figure 3. Additionally, one can define the capacity of each node so more than 1 point can be contained within a region. A sample output is shown below. As it can be seen from Figure 3, every subregion contains only one subregion. Also, on Figure 4, a implementation of a point quadtree is shown where every subregion appears to have more than just one point. However, this is because only leaf subdivisions are shown. For both cases a blue rectangle is shown to demonstrate the query capabilities on the specific area. The points gathered from the area are colored green.

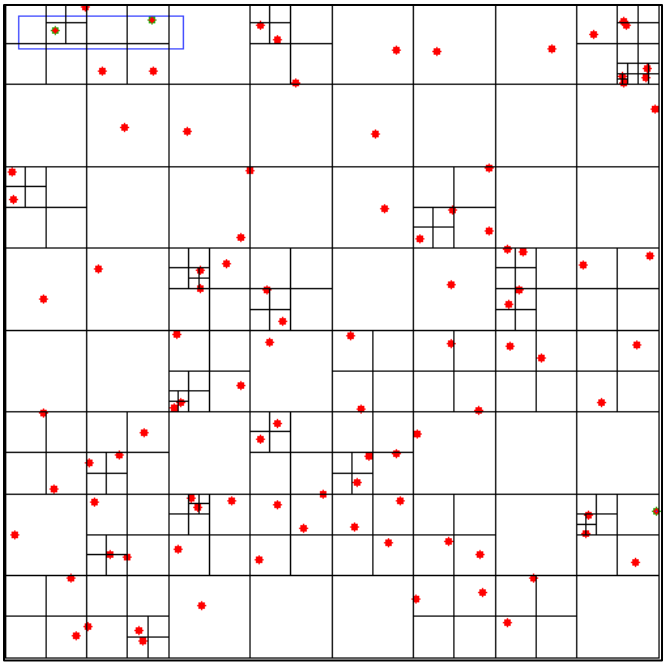


Figure 3 Point-Region Quadtree Sample Implementation

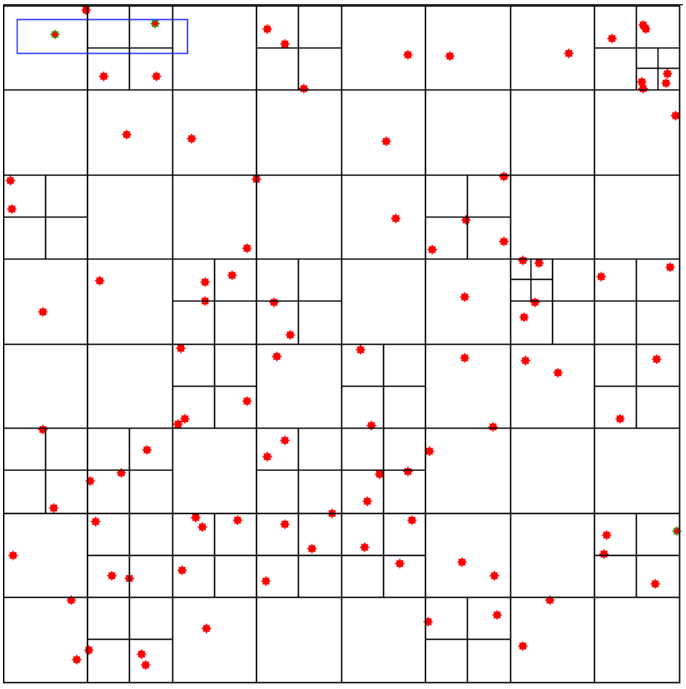


Figure 4 Point Quadtree Implementation Sample

4 Test Results

For KD-trees and Quadtrees the same 2-dimensional data sets were generated and also permuted in accordance with Table 1. Every data set was used as an input to the Quadtrees and KD-trees and the time taken to build for each case was recorded for both spatial trees. Additionally, the find algorithm for the two cases was run for every test case.

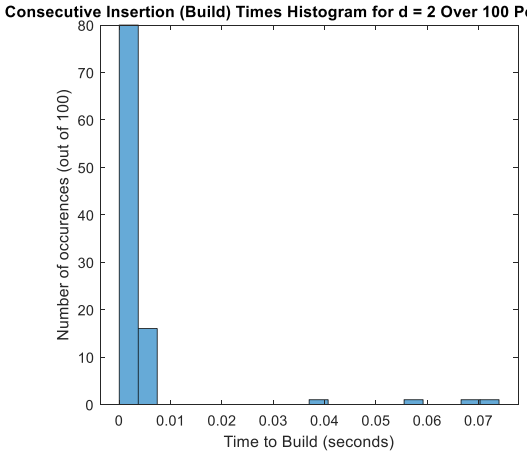
Table 1 Test Case

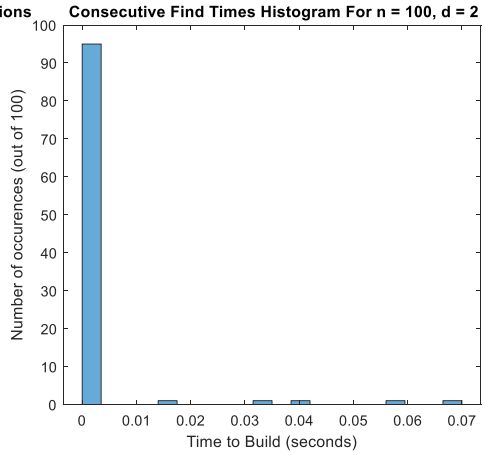
	Random Datasets				**Random Datasets Permuted**			
n (number of rows)	100	1000	100K	1M	100	1000	100K	1M
Number of Datasets	100	100	10	10	100	100	10	10

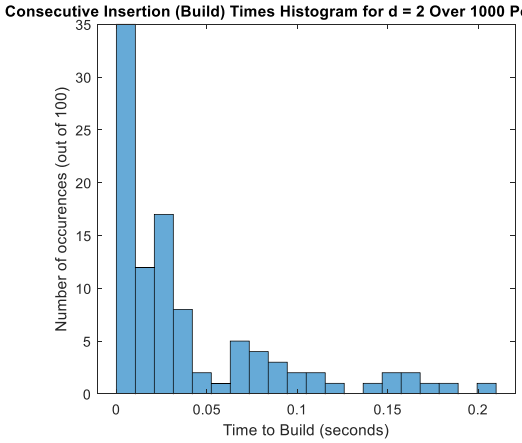
"Random datasets" corresponds to data in which the coordinate values are sampled from uniform[0, n*10]. "Random Datasets Permuted" permutes the 0^{th} dataset in each "Random Datasets" category a certain number of times. The hope was that calculating histograms of the permutations would test an operation across different shapes of the tree, while the simple "Random Datasets" would test the operation across randomly sampled trees.

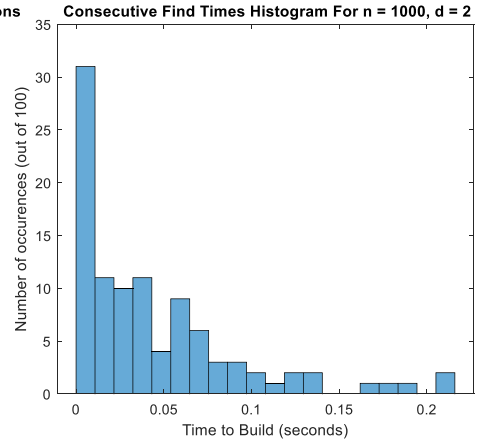
`These datasets were generated by random_dataset_permutations_generator.cpp and random_dataset_generator.cpp. It took a while to generate the random datasets, especially those of size 1M. For this reason, it is not advised that these are run, but the user is welcome. The output of each is stored in "random_dataset_permuted" and "random_datasets" with a subdirectory "n_*" where "*" corresponds to the size *n* of the dataset.

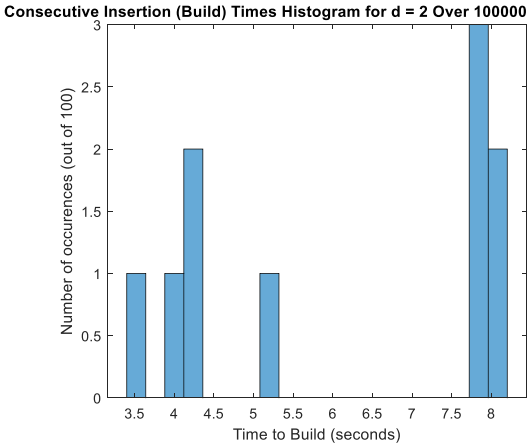
4.1 KD-tree Implementation Results

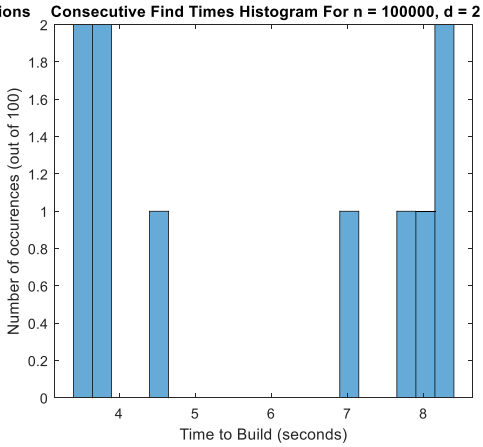












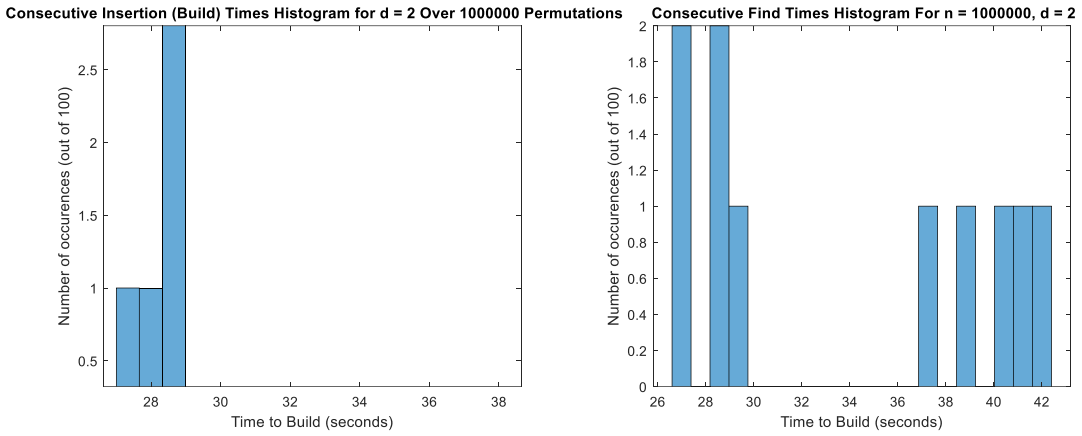


Figure 5 Histogram Results for KD-Tree

At first glance the Build and Find time plots for the n = 100 case seem uninteresting since binning isn't fine enough to get good enough resolution below .02 seconds. However, taken together with the plots for n = 1000, 100K, and 1M the results make sense. The cases where the time to build occurs well over .02s are probably cases where the tree is incredibly unbalanced, meaning that finding or inserting a node requires more time to travel to the leaves for insertion or find. For the cases where n > 100 the times tend to be more uniformly distributed probably because it is much easier to make a tree more unbalanced when it has fewer nodes then when it has many: as in, the ratio of unbalancing insertions to balanced insertions increases with the size of the tree.

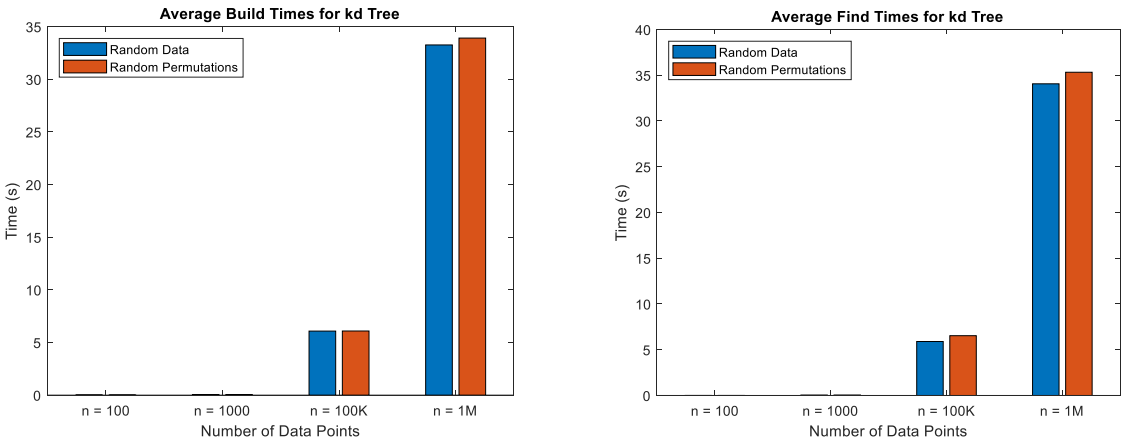
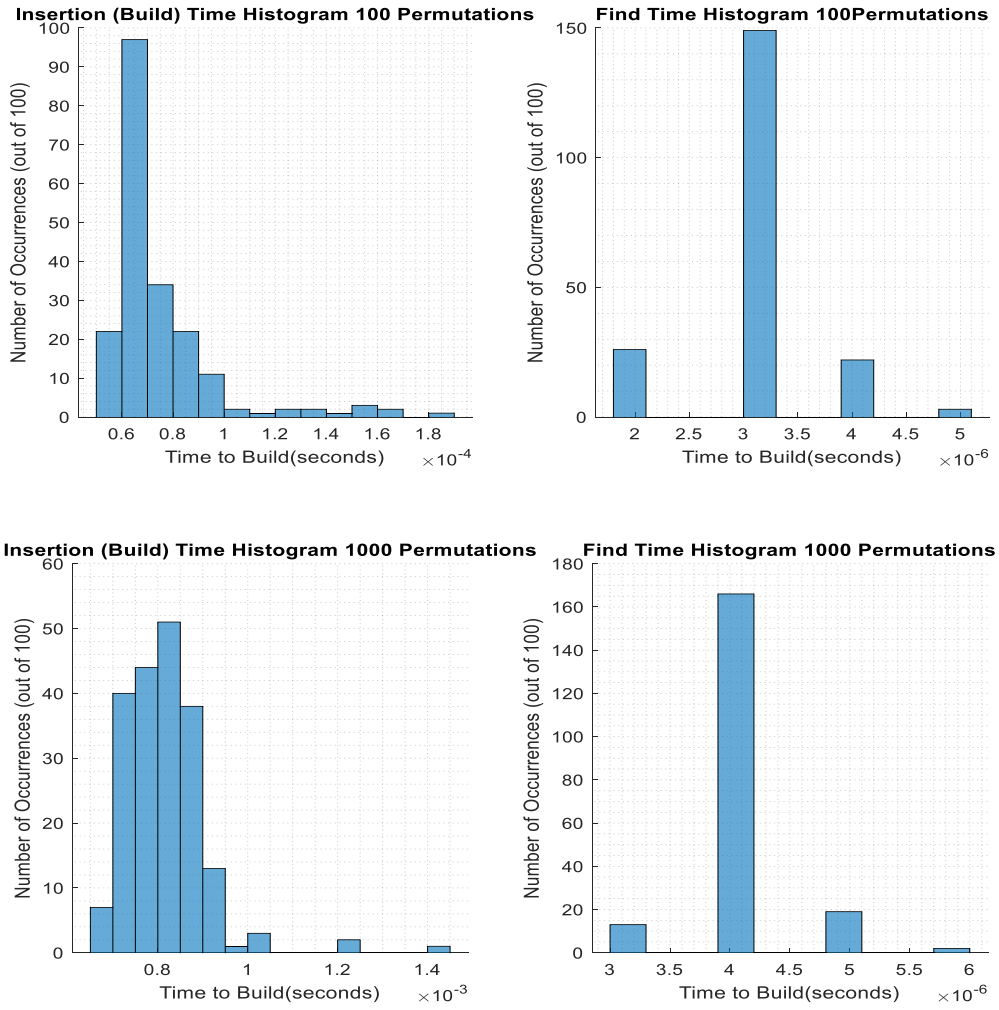


Figure 6 Average Build/Find Times for kd-tree

For the average Build/Find times, what's interesting to note is that the average time for Find for a randomly permuted dataset is greater than a simple randomly generated dataset. This is probably due to the fact that it is more likely when permuting the same tree that the tree will be more unbalanced then when simply generating random keys.

4.2 Quadtree Implementation Results



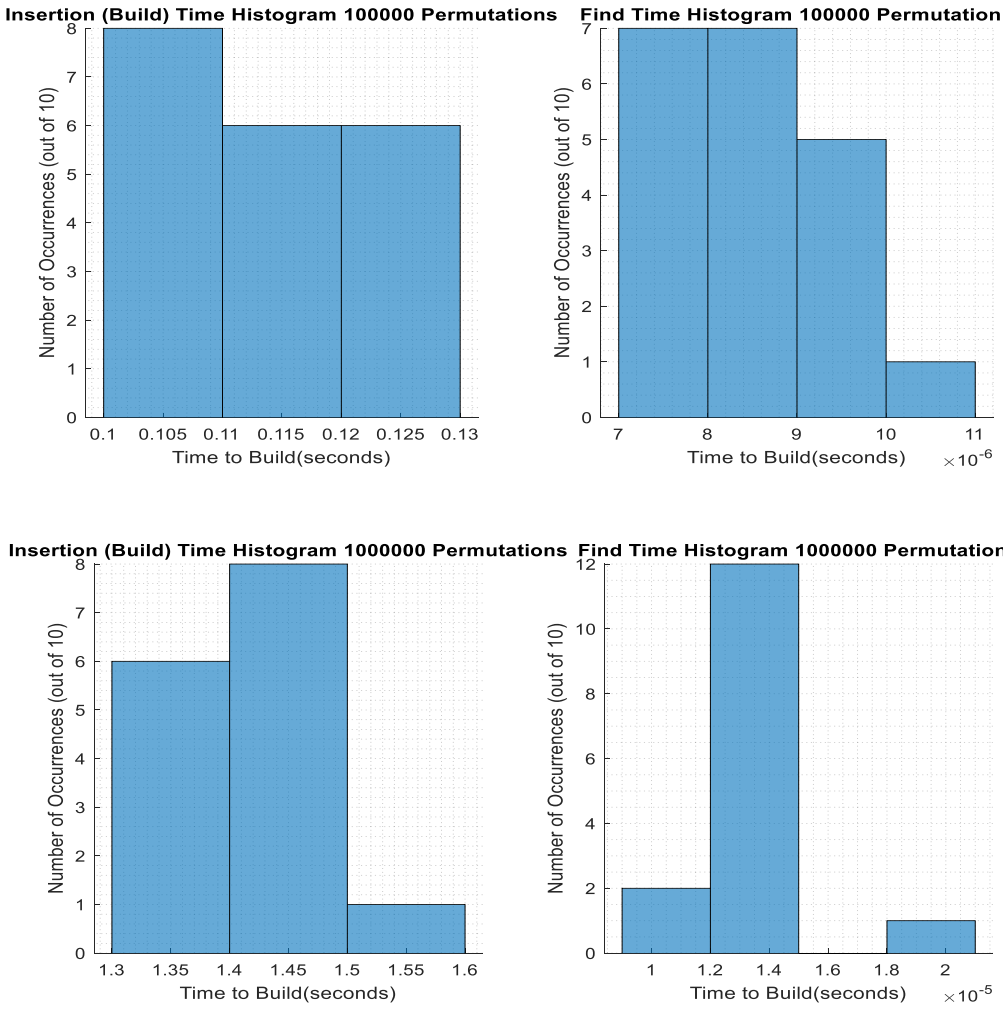


Figure 7 Quadtree Histogram Results

As it can be seen, for the insertion cases on the permuted data sets, the quadtree insertions are usually consistent. However, it can be appreciated that for some cases on the 1000 and 100 data set permutations, the quadtree insertion method presents a few outliers. This could be because of existence of repeated points that increase the computation time since it needs to do a check for every child node.

Similarly, the finding method time appears to be constant, or that most of the cases resulted in a constant finding time. For the outliers, the reason could be that since the points to be found were randomly generated, it could happen that the points were wither nonexistent on the data set. This is the reason that the 100 and 1000 data sets contain higher distances between bars since the probability of obtaining a nonexistent point is greater.

4.3 Average Time Comparison

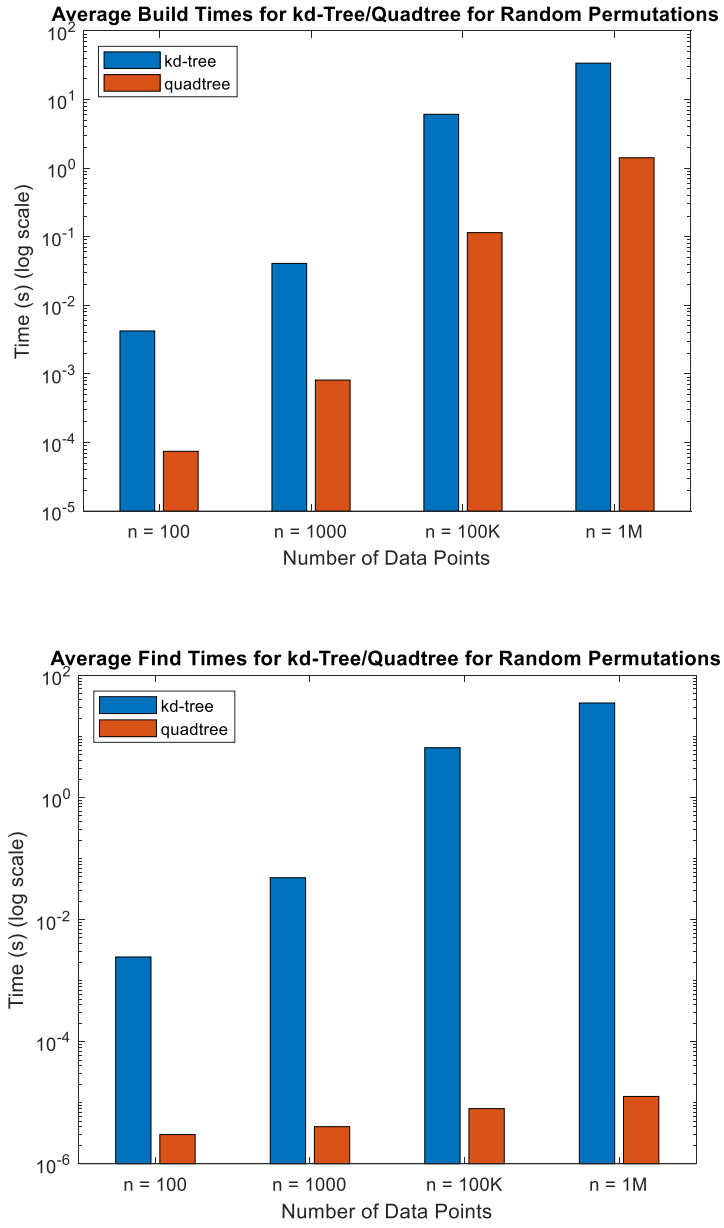


Figure 8 Average Times Comparison Between kd-Tree and Quadtree

The results confirm that quadtree's insertion and deletion time is faster than kd-trees. This behaves as expected since quadtrees have four equally spaced children rather than the two children from kd-trees. Thus, due to the dependency of the build time to the depth of the trees, it's obvious that the quadtree contains less levels in average than the quadtree. This shows that

the quadtree tends performs better due to the number of average recursions that the insertion and find methods need to incur.

5 Conclusion

A successful implementation of KD-trees and Quadtrees was performed. For each of the trees the methods for insert and find points within the data structures were implemented. Additionally, for quadtrees, a method for region query was implemented. For each of the trees, the same data set was built and analyzed using several permutations of the same data set for different data set sizes. At the end a comparison graph plot was generated in which it is shown that for general insertion and query of datasets. It is concluded that quadtrees have a better performance in terms of insertion and query due to the average depth of each of the data structures. However, in future work, a more in depth analysis of the space complexity and average depth of each tree would be required to validate our assumptions.

6 Division of Labor

Andres Chavez Armijos – Quadtree Implementation

John Neal – kd-Tree Implementation

7 References

Sahni, S. (2004). *Handbook of Data Structures and Applications*. *Handbook of Data Structures and Applications*. Chapman & Hall/CRC. https://doi.org/10.1201/9781420035179

Samet, H. (1984). The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys (CSUR)*, *16*(2), 187–260. https://doi.org/10.1145/356924.356930

Samet, H. (1990). *The design and analysis of spatial data structures*. *The design and analysis of spatial data structures*. ADDISON-WESLEY Publishing Company. https://doi.org/10.1016/0924-2716(91)90007-i

Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. Commun. ACM 18, 9 (September 1975), 509-517. DOI=http://dx.doi.org/10.1145/361002.361007