Motivation

We are passionate and motivated to study about space partitioning because of the variety of applications it has in our daily lives. Space partitioning algorithms solve the nearest neighbor search and range search in computational geography. One example of nearest neighbor search is calling for an Uber to pick people up. When a person confirms their request for an Uber, the algorithm takes note of the person's location and attempts to find an Uber driver closest to their location and pair that driver up with the person. The usage of this algorithm is crucial: if the user does not get paired up with the drivers closest to them, they will have to wait for quite a while, which may lead them to canceling their request and using another platform.

Furthermore, range search is applicable to various fields as well, from computer-aided designs (CAD), navigation systems, and databases. One example that is very applicable to our daily lives is the usage of Google Maps. When we search for the closest coffee shops within a couple miles of our location, Google Map's algorithm takes note of our current location and then uses GPS to find the nearest coffee shops.

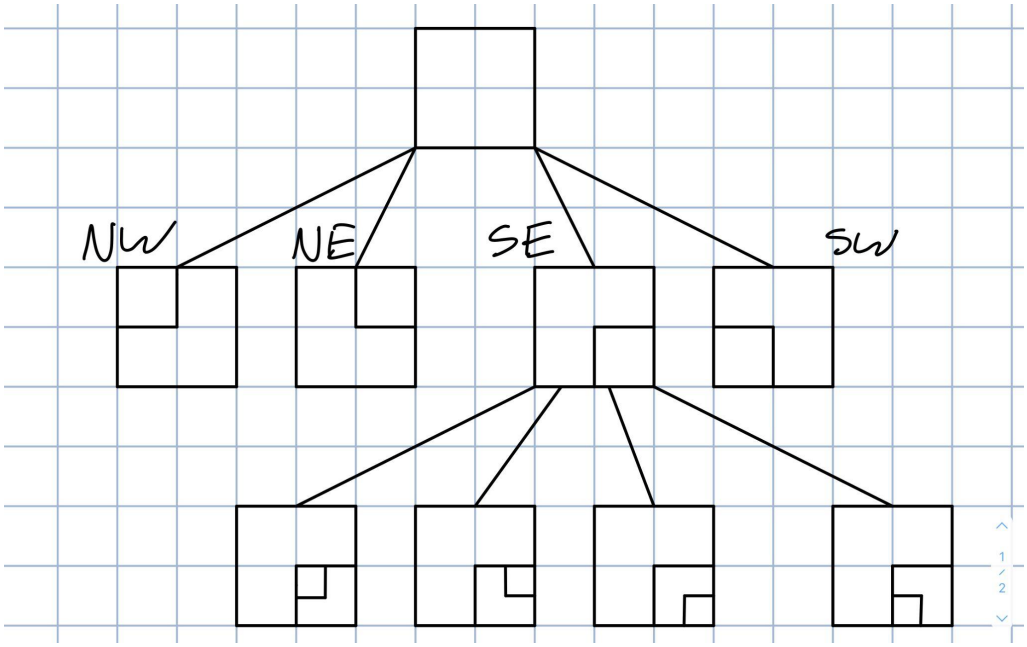
The problem and its solutions expand on concepts from this course such as binary search trees, 2-3 trees, and red-black trees. Binary search trees are optimal for storing and organizing one-dimensional data such as numbers, but they can iterate over all objects which takes more time. 2-3 trees are used for similar purposes to binary search trees, but are most optimal for larger blocks of data due to its self balancing feature, so it is commonly used for database and file systems. Red-black trees also have a self balancing feature that optimizes the topology of the tree, and are used for worst-case time complexity guarantees for insert, delete, and search functions. Although each of these types of trees are highly applicable and useful for storing data in one dimension, none of them are ideal for storing multidimensional data: for example, typically for two-dimensional data, you build a tree based on only the x-coordinates or only the y-coordinates of the data, not both, which is sufficient enough for some queries. However, range-finding operations concerned with the opposite coordinate that our tree is built upon will fail, and the runtime will be considerably slow since we will have to traverse over every node in the tree.

When we want to perform operations on objects in a multidimensional space, we would like to store data in a similar format to these types of trees but with improved efficiency and expanded features. To solve this issue, we can generalize the idea of these trees in the form of quadtrees, k-d trees, or uniform partitioning to apply to multidimensional data.

Design and Analysis

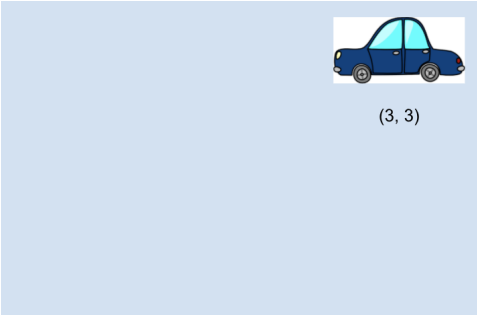
Quadtrees

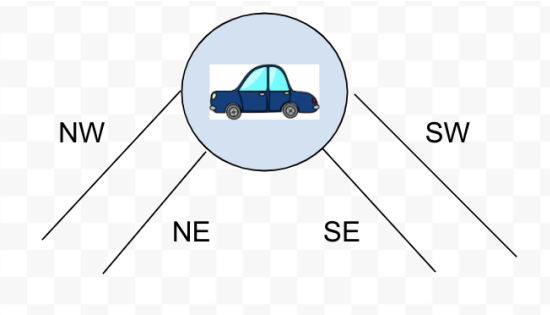
A quadtree is a type of tree in which each non-leaf node has exactly four children. This is generally used for partitioning two-dimensional space, in which case the four children nodes represent the partitions to the northwest, northeast, southwest, and southeast directions. It is mainly used for image compression, in which each node has the average color value of their children. It is also used to find nodes in a 2D space, such as finding the closest point to some coordinate.
Below, we have provided an illustration to show the relationship between the parent node and its children:



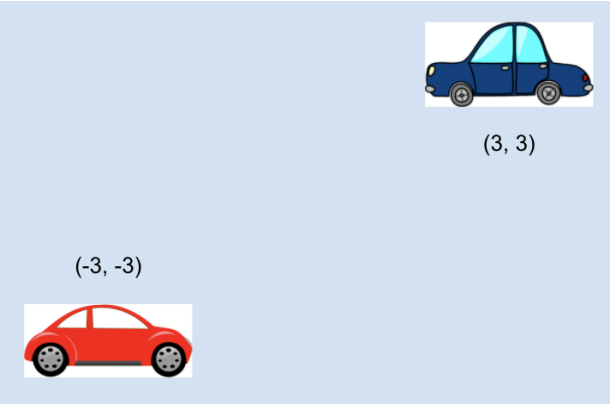
Quadtree Insertion

We will set the starting node as a blue car with coordinate (3, 3). The blue car node will have four edges extending out: northwest (NW), northeast (NE), southeast (SE), and southwest (SW).

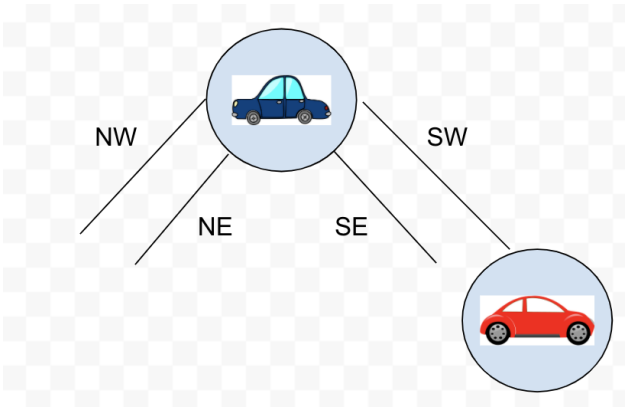




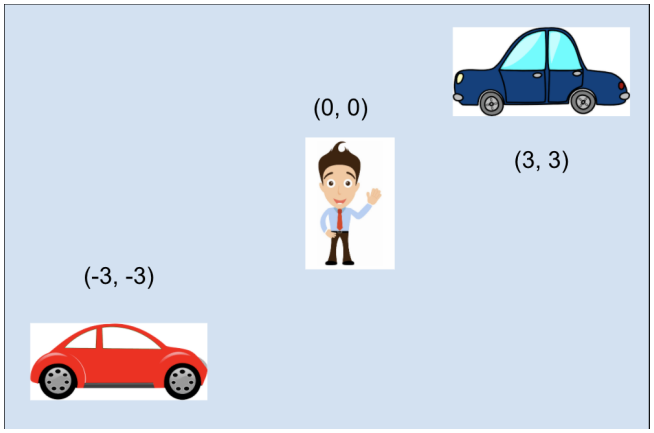
Next, we can add another red car to the 2D plane with coordinate (-3, -3).



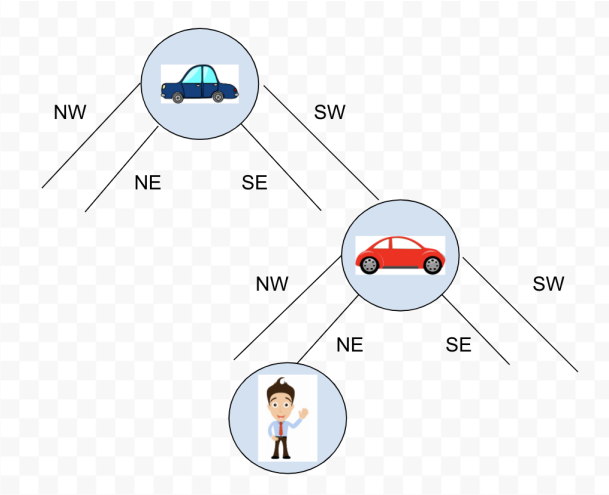
We can see that from the location of the blue car, the red car is positioned in the southwest direction. From this, we can add the red car as the southwest child of the blue car in the quadtree.



Lastly, if we wanted to add a person with coordinates (0, 0) on the plane, it would look like this.

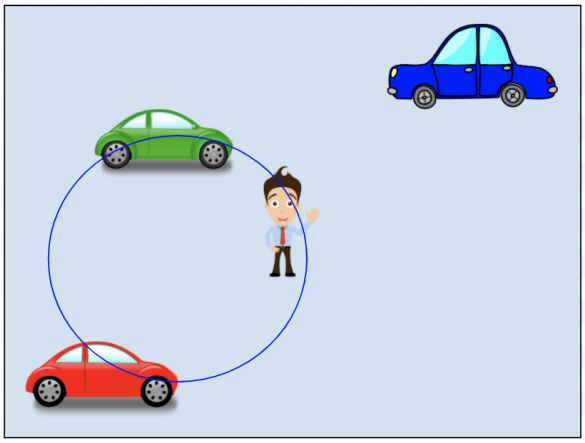


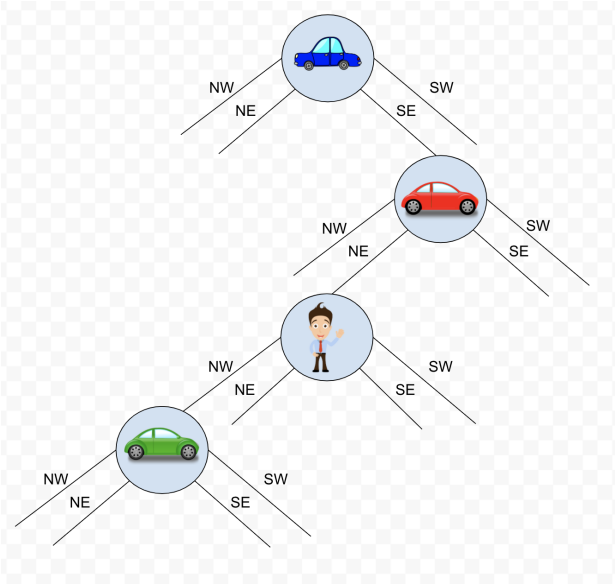
The person is located in the northeast direction of the red car. This would make the person node become the NE child of the red car. We can also see that the person is to the southwest of the blue car, which is accurately depicted in the quadtree, as the person node will be found in the southwest edge of the blue car. This is how a quadtree is constructed.



Nearest Neighbor Search

To understand the nearest neighbor search, we can look at the following question: given the left image below, which car is closest to the person pictured?





Note: the image and tree diagram are similar to the ones in the quadtree construction, except for the addition of the green car positioned at the NW direction relative to the person.

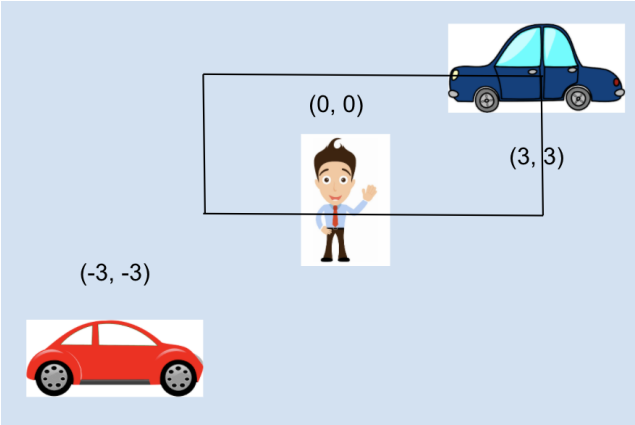
When implementing the nearest neighbor search with quadtrees, we create a circle around the object that we would find the neighbors for. Then, we look at the root of the tree and work our way down, adding any objects that are in the region of the circle to our answer.

In the example above, we start with the blue car since it is at the root. Since the blue car is not in the region of the circle, it is not added to our answer. Then, go in the SW direction to the red car since it is next in the tree. The red car is in the region of the circle, so we add the red car to our answer, and go to the NE edge where we find the person. Since he is the object we are performing the search on, we don't add him to our answer. Lastly, we go to the NW edge and find the green car, which is in the region of the circle. We add the green car to our list of answers.

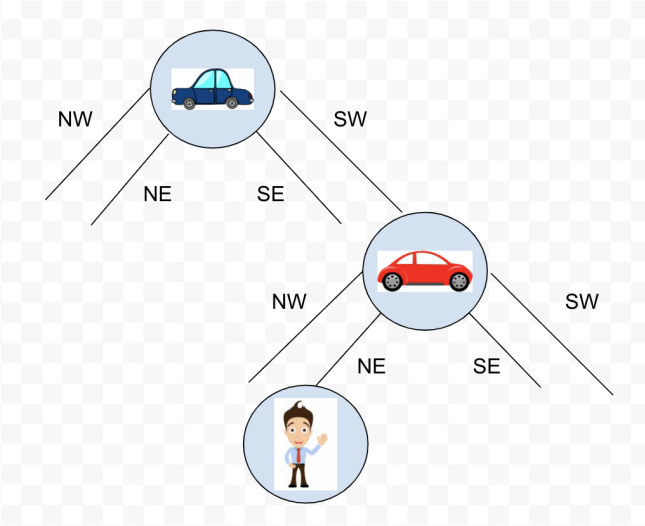
Therefore, the final answer to the question is the red car and the green car.

Range Search

Next, we can examine how the nearest neighbor search can be implemented with quadtrees. The overall idea of performing a range search is that we do not explore subspaces that do not cross the query's shape (rectangle, square, circle, etc.).
Using the visualization to the right, how can we answer the question of: what objects are present in the black rectangle?



We can reference the quadtree we previously created in quadtree construction, as shown to the right. To answer this question, we need to begin by looking at the root node, which is the blue car. We can see that the blue car is in the rectangle, so we take note of that to include it in the results. We then examine the four subregions of the blue car that intersect with the rectangle. As we can see, the subspaces that might have good points are in the southwest direction. So, we prune (eliminate) the other edges and focus on the southwest edge.



Next, we look at the red car node. It is not in the box, so we don't add it to our results. We look at the four subregions of the red car, and can see that the box is located in the northeast path from the red car, so we prune the other edges and only explore the northeast node. The person node is on the northeast edge and is in the box, so we add it to our results.

As a result, the answer to our question would be the blue car and the male cartoon. This is how the range search algorithm works on quadtrees.

Runtime Analysis

Insertion: $\Theta(\log N)$

To insert a new node in the tree, its key is first compared to the root. For example, if it's located southwest of the root, it is then compared with the key of the root's southwest child, but if it's located northeast of the root, it is then compared with the key of the root's northeast child. This continues until the node is compared to a leaf node and inserted accordingly, but average case is $\Theta(\log N)$ because its runtime is proportional to the height of the tree, which is $\log N$.

Search: $\Theta(\log N)$

The search function for a quadtree is used to find the location of a node in a given quad. When searching for a node, we can prune all quadrants that we know do not contain the node we are searching for. This means that we only actually need to traverse through one node per level of the tree, or the tree's height, $\log N$.

Removal: $\Theta(\log N)$

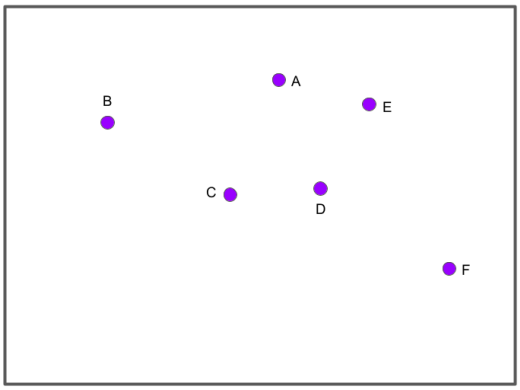
In the removal function, we need to find the successor of the node to be deleted, then replace the node with that successor. This is proportional to the height of the tree, resulting in a $\log N$ runtime.

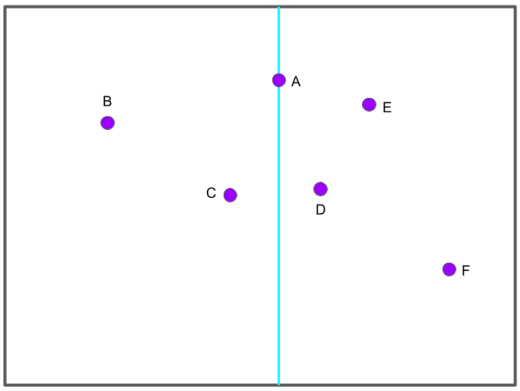
Connection to Class Concepts

Quadtrees are an adaptation of binary search trees in which each parent node has four children instead of two. While a binary tree divides a one-dimensional space into two parts (the left and right nodes), a quadtree divides the space into four quadrants (the northwest, northeast, southeast, and southwest nodes) to accommodate for two-dimensional data. Additionally, their insertion, search, and removal functions all share the same time complexities since the trees share the same height properties.

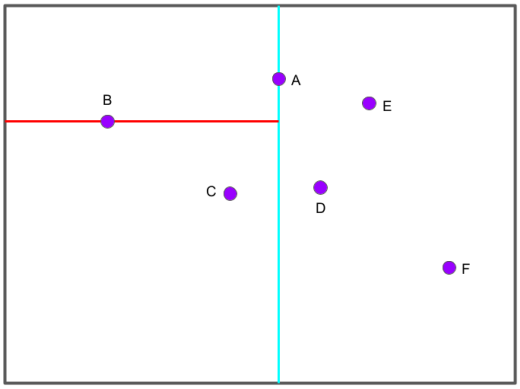
K-D Trees

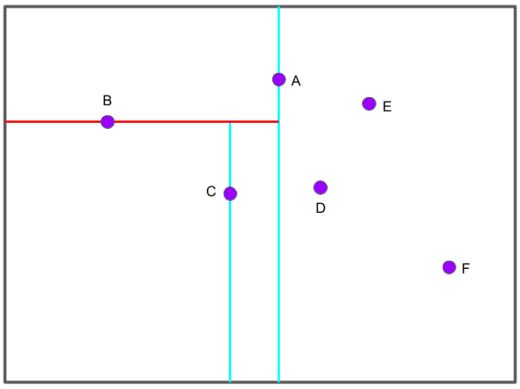
K-d trees (short for k-dimensional trees) are binary trees in which each leaf node is a point in k-dimensional space. Each non-leaf node divides the k-dimensional space it resides in into two parts along one dimension, and each successive division cycles through the various dimensions. One way to construct a balanced k-d tree is to always choose the point with the median coordinate in the dimension currently being used for splitting. This assumes that all points are known at the start, rather than being inserted one at a time. An example with diagrams is shown below:
We find the median point (sorted by x-coordinate), which is point A, and split.

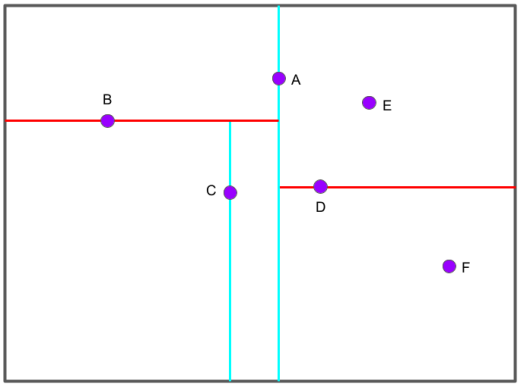


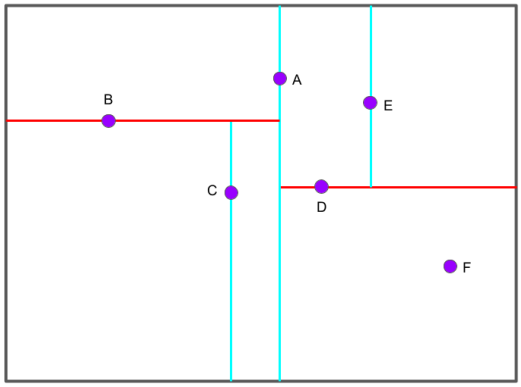


We repeat, this time taking the median by y-coordinate, giving us point B, and then point C is the only point remaining in that space so we choose it.

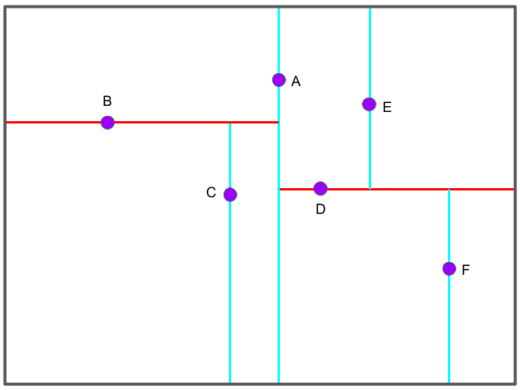


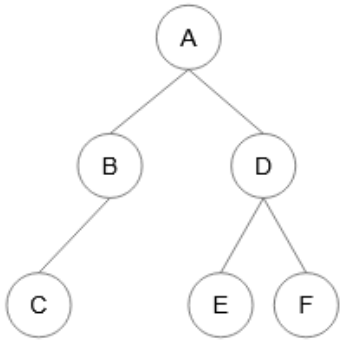






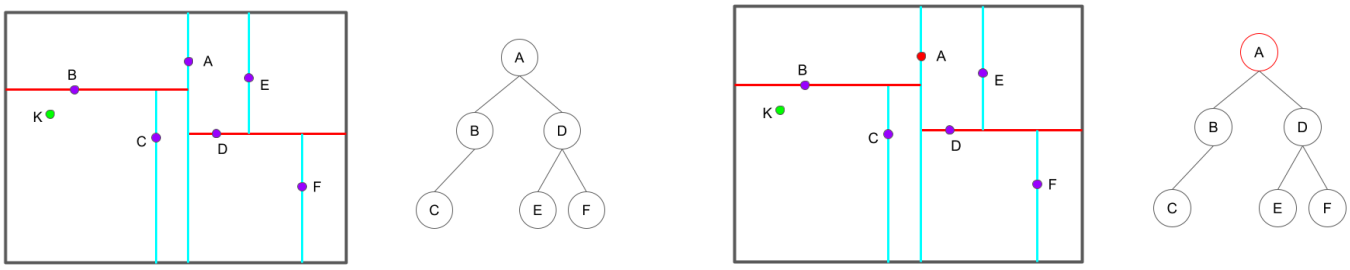
Moving on to the right side, we find the median by y-coordinate again giving us point D, which we split horizontally. Searching the top space, we find E and split. Finally, we search the bottom, find point F, and split there. There are no more points, so the tree is complete. On the right we have the tree's binary tree representation.



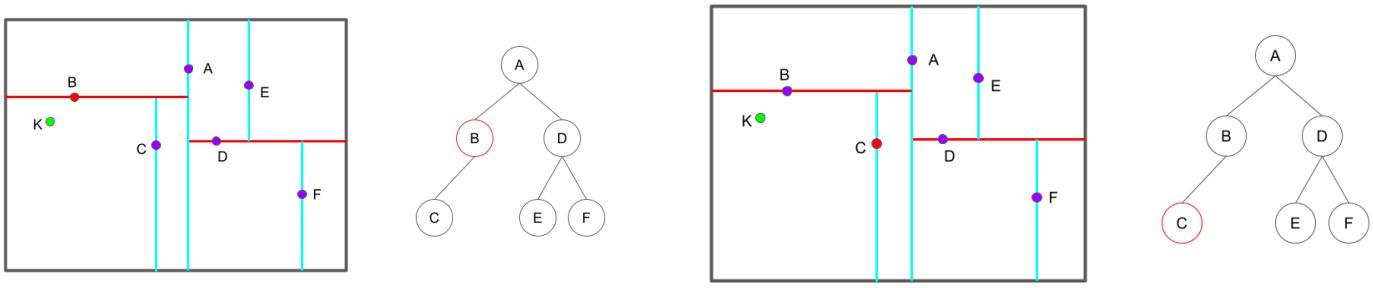


Nearest Neighbor Search

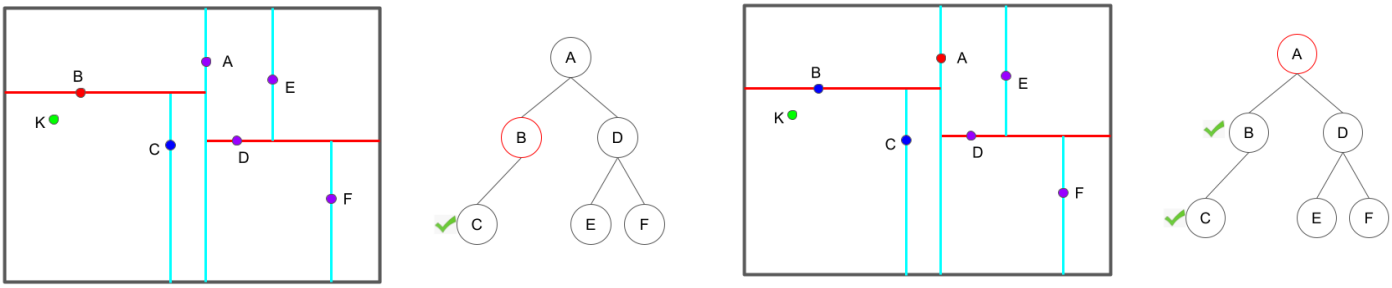
Like other space partitioning algorithms, k-d trees optimize nearest neighbor search by pruning possibilities. In a k-d tree, this occurs by removing possibilities in a space farther from the chosen point than the current nearest point. The algorithm also optimizes by always first trying the side closer to the chosen point. An example can be seen below, with the point being searched for labeled K:



Starting at the root of the binary tree, we label A as the current closest point, and then look at the children. B is closer than D, so we follow that branch.

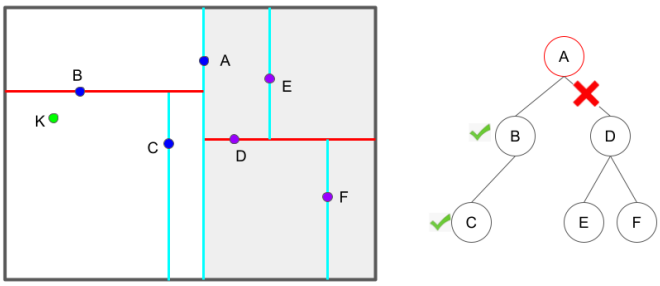


Here we see that B is closer than A, so we update our closest point to be B and continue down the tree. B only has one child, C, so we go there. We see that C is farther from K than B, so we don't update the closest point, which is still B.



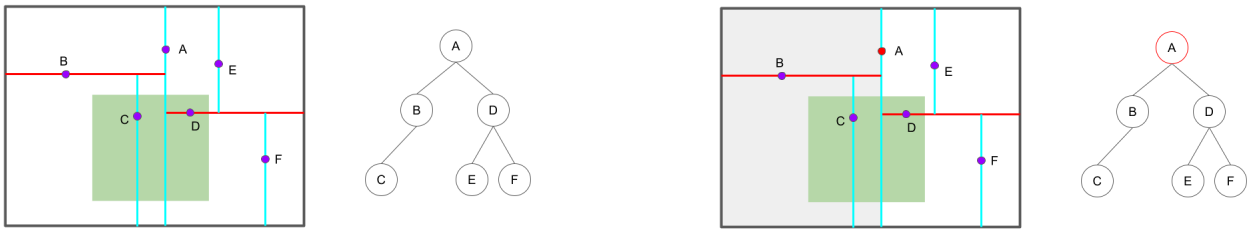
C has no children, so we can start climbing back up the tree. Since B doesn't have any other children, we go back to A. We check the distance from K to the line that A is sitting on and see that it is less than the distance from K to the current best point (B), so we

know that no points to the right of A can possibly be closer to K than B, so we can prune the right side of the tree, as shown below.

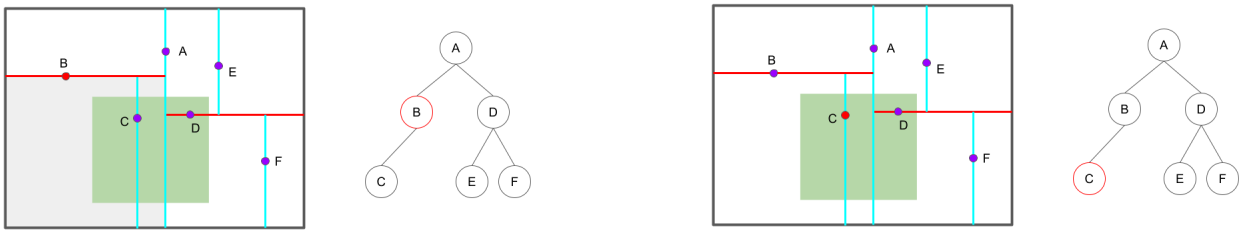


Now because we have finished checking both children of A, we are done checking A, and because A is the root of the tree, we are done with the nearest neighbor search and know that the closest point in the tree to K is B.

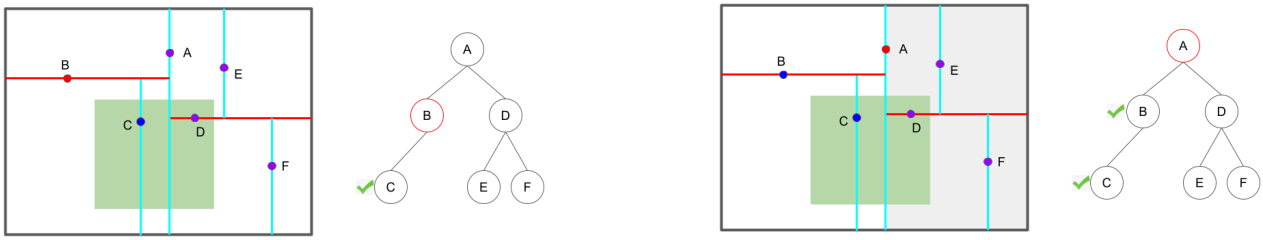
Range Search



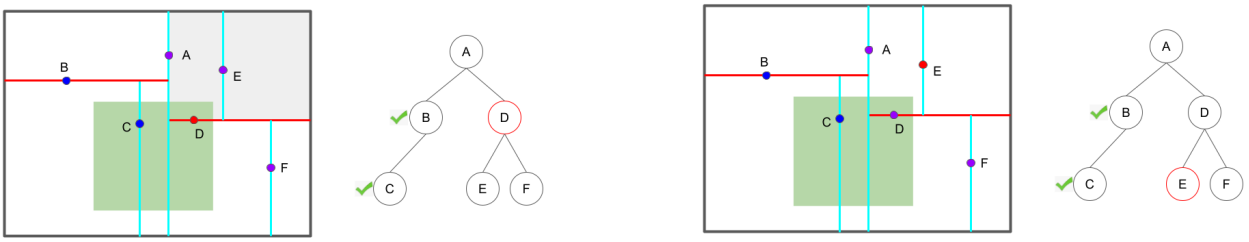
Suppose we again have this k-d tree, and we want to find all points which lie in the green rectangle. We begin our search at the root (A), as with nearest neighbor search. First, we check if A lies in the green rectangle. It does not, so we move on to its children. We check point B, noting that it is on A's left side. Thus, we examine the entire left side and check if it overlaps with the green rectangle. It does, so we continue to point B.
B is not in the green rectangle. However, it has children, so we examine point C. C is below B, so we check the region bounded by the line B lies on to see if it overlaps with the green rectangle. It does, so we move on to point C.



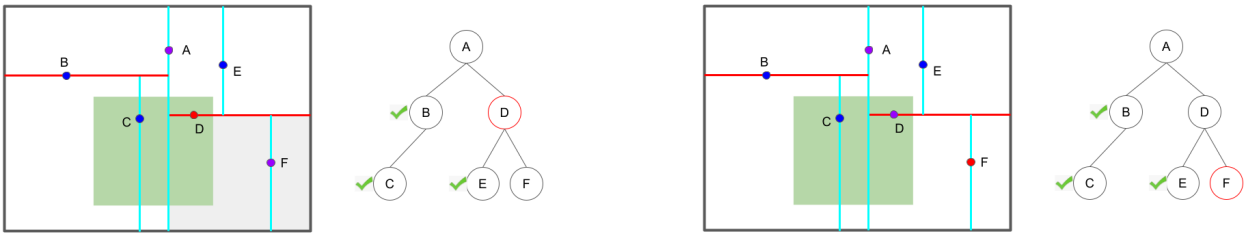
Point C lies in our desired region, so we add it to the list. It is a leaf node, so we begin climbing back up the tree to point B. B has no other children, so we continue climbing back to A. Having checked A's left side, we move on to the right. The right side overlaps with the green rectangle, so we start checking point D.



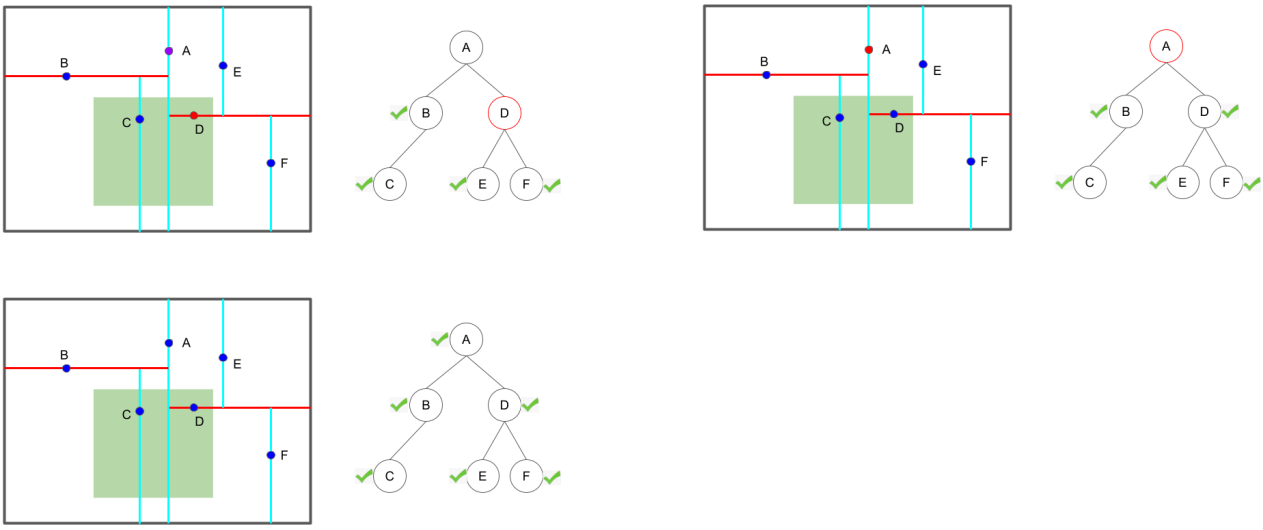
We first note that D is in the green rectangle, so we add it to the list, which now contains both C and D. Next, we examine D's children. E is above D, and the area above D intersects the green rectangle, so we check point E.



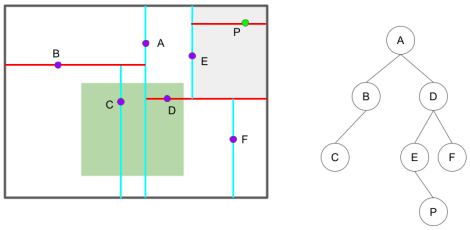
E is not in the green rectangle, and it has no children, so we return to point D. From here we look at F. F is below D, and the area below D intersects the green rectangle, so we check F.



F is outside the green rectangle, and has no children, so we can climb back up the tree. Now both of D's children have been checked, so we continue back up to A. Both of A's children have been checked, and A is the root node, so we are done.



One final note: this example didn't include pruning. What does that look like? Suppose we add another point P, as shown below:



As we traverse the tree, we will eventually reach point E, as shown before. Now we note that P is to the right of E. We then examine the bounds of the rectangle bounded by E (colored in grey here) and find that it does not intersect the green rectangle. Thus, we do not have to examine the subtree rooted at P at all, no matter how many nodes it might contain. This is how a k-d tree can optimize range search. Note however, that there is no guarantee that a range search will not have to check every node in the tree. The same is true for the worst case in nearest neighbor search.

Runtime Analysis

Insertion: $\Theta(\log N)$

To insert a new point into a k-d tree, we will start at the root node and traverse through either the left or right child depending on the node being inserted. We will continue to traverse the left or right child of each node visited until we get to the node under which the new node should be located, then add it either to that node's left or right. In this function we will on average traverse through one node on each level of the tree, or the tree's height, $\log N$.

Nearest Neighbor Search: $\Theta(\log N)$

In the nearest neighbor search, we start from the root node of the tree and traverse downwards in the same way as insertion. Once it reaches a leaf node, it checks to see if that distance is shorter than the current distance saved, and if so, saves it as the current best distance. It does this recursively until all possible nodes are visited. The number of nodes visited will be proportional to the tree's height, $\log N$.

Removal: $\Theta(\log N)$

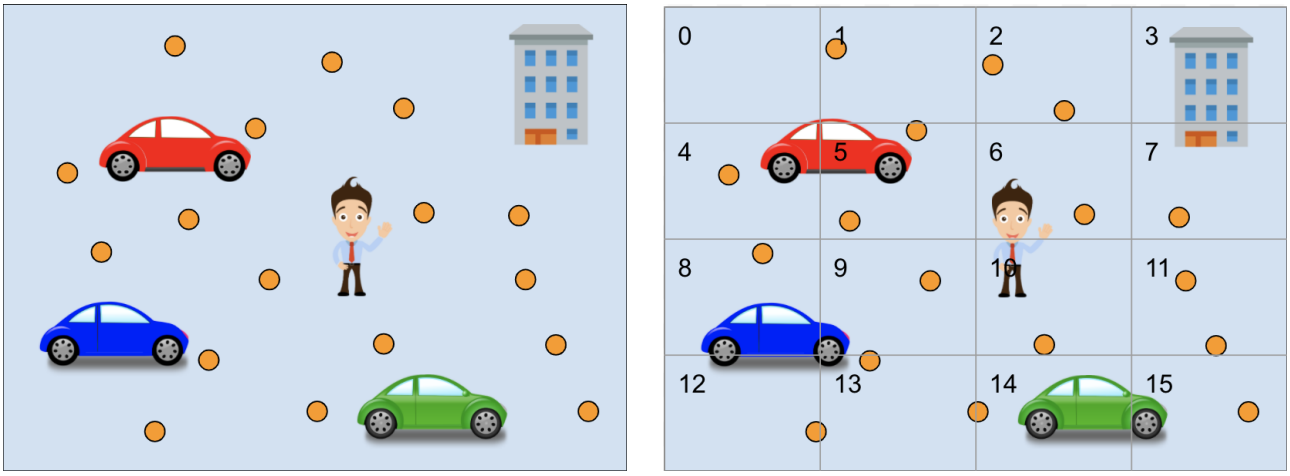
In a k-d tree removal algorithm, we will recursively traverse through the tree and search for the node to be removed. It will follow a single path down the tree and does not visit a single node twice as it traverses, hence it essentially searches the tree's height, $\log N$.

Connection to Class Concepts

K-d trees are another adaptation to binary search trees in which every leaf node is a k-dimensional point. It builds on binary search trees in that it has k data fields instead of one. While binary trees are incredibly useful for storing one-dimensional data such as numbers for quick look-up, addition, and removal, k-d trees can store multidimensional data which gives it the ability to solve complex geometric problems in statistics and data analysis. Every leaf node in a k-d tree can be thought of as a median generated from splitting a hyperplane into two parts.

Uniform Partitioning

Lastly, we have the uniform partitioning algorithm. Unlike the other algorithms that we have discussed so far, it is not based on a tree. The space is partitioned into uniform (same size and shape) rectangular buckets, also known as bins. It is quite similar to hashing in that it allows efficient region queries. We have provided an example visualization below. The left is the original image and the right is the partitioned version of the image, divided into a 4x4 grid with bins numbered from 0 to 15.



The goal is to identify the buckets that need to be iterated over in order to find all the points in a given range. This will prune the areas that we search for the points within the plane instead of having to look through the entire plane, resulting in a faster runtime.

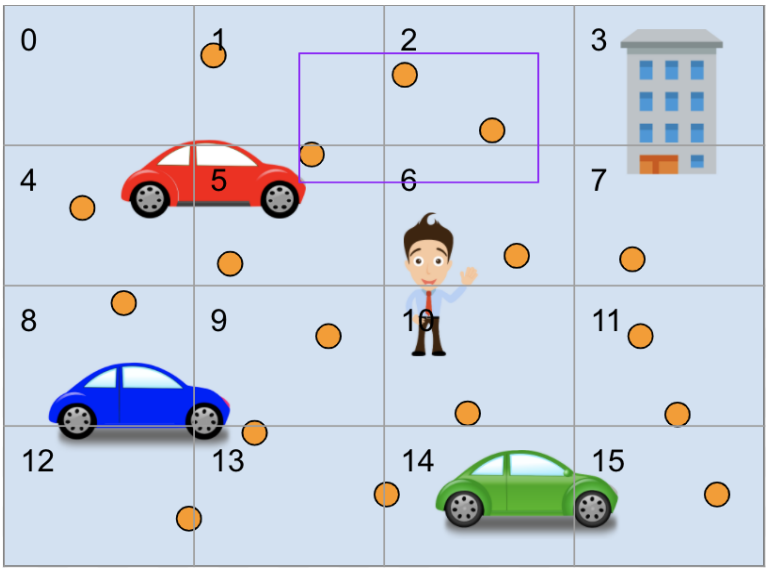
Nearest Neighbor Search

To understand how nearest neighbor search works in uniform partitioning, we can examine the following question: how many points do we need to examine to find the point nearest to the blue car with?

First, we need to find the bin numbers that the blue car is located in. From the partitioned picture above, we can see that the blue car is located in bin numbers 8, 9, 12, and 13, so we can focus on those specific bins to figure out which points are closest to the blue car. There is one point in each of those four bins, so we would need to examine those four points in order to find the nearest point to the blue car.

Range Search

Consider the visualization below. It is the same as the visualization above, but with a purple rectangle. Range search in uniform partitioning can answer the question of: which bins do we need to iterate over to find all points in the purple rectangle region?



To answer this question, we would look at what bins are in the purple rectangle region. Based on the image above, we can see that the purple rectangle lies in bins 1, 2, 5, and 6, so we only need to examine the points that lie in those bins! If we were implementing a list, we would add all the points in those particular bins to the resulting list.

A common follow-up question is: why are we looking at all the points in those selected bins when some of them aren't close to the purple rectangle, such as the one directly to the right of the male cartoon? Well, we can divide those bins into smaller pieces to further refine the area of search.

Uniform partitioning simplifies the area in which we search in because we only had to examine a couple of bins instead of all the bins from 0 to 15.

Runtime Analysis

Search: $\Theta(N)$

Although uniform partitioning divides our image space into x number of buckets, the runtime is still $\Theta(N)$. This is because on average, the runtime will be x times faster than without the uniform partitioning since we no longer need to visit every point, but N/x still simplifies to a runtime of $\Theta(N)$.

Connection to Class Concepts

Uniform partitioning can be a great alternative to separate chaining hash tables when dealing with spatial data. In separate chaining hash tables, a hash function is used to compute the corresponding array to put each value in to determine its index. They offer many benefits, including the ability to check if a key is already stored or adding a new one. However, the main issue with them is that each item ends up in a random bucket, which can cause the lookup to be unnecessarily grueling. We can expand the idea of hash tables to uniform partitioning, in which the bucket numbers depend only on position, rather than the items' hash codes. This method improves our runtime, since we only need to visit buckets that may contain a certain item rather than iterating through every other item in the array before it as well.

Two Approaches to K-D Tree Construction

Using merge sort as the median-finding algorithm for a balanced k-d tree construction results in an overall construction runtime in $\Theta(N\log N)$. Performing merge sort on a set of data one time results in a runtime of $\Theta(N\log N)$, because we will need to visit N items to sort, and logN is the height of a tree. The extra logN in the k-d tree construction comes from calling the merge sort over and over again recursively to recalculate the new median each time to add to the tree to make it as bushy as possible, thus making the runtime $\Theta(N\log N)$.

A more efficient approach that would reduce the overall runtime to $\Theta(N\log N)$ would be to sort the data before adding it to the tree, using the median as the overall root of the tree. This eliminates a $\log N$ from the original runtime, since by sorting the data first we no longer need the recursive call for the tree construction.

Affordance Analysis

In CSE 373, we learned to analyze the broader societal impacts of choosing and implementing specific data structures. It is important to consider when to apply a particular space partitioning algorithm and discuss the implications of choosing each one.

Quadtrees are often used to partition a two-dimensional space by recursively subdividing it into four quadrants. Each leaf node contains spatial information or data that can be used in a variety of applications. They are highly useful for image compression, where each node contains the average color of each of its children. The further down the tree you traverse, the more detailed the image. Images in large quantities can take up a lot of memory and processing power to keep them at their full quality.

To solve this issue, we can use image compression to reduce the size of the image and the pixels and memory being used. Quadtrees are a great data structure to use in image compression since they can ensure that we keep the original quality of the image and use machine learning for edge detection. In image compression algorithms, quadtrees recursively divide the image into four quadrants with each holding the average RGB color and the error determining that color for its subspaces. The tree starts with a single node of the average color computed from the average RGB value of the entire image. To reduce the error of this one average color, the tree splits the root node into four quadrants, each with their own average color value from just that quadrant.

As this process continues, we will get an image that is closer to the original, but some quadrants may seem out of place due to the compression of the image. This process gives us a tree that still uses larger amounts of memory for the detailed areas of the image, and less for the areas that are less detailed. Since quadtrees focus more on the detailed parts of an image and less on the non-detailed parts, this improves edge detection of images since to find an edge we only need to visit the leaf nodes and their parents, rather than the whole tree. A disadvantage of quadtrees is that they are not great for shape analysis and pattern recognition. For example, it is nearly impossible to compare two images that differ only in rotation or translation, since the quadtree represents each image in completely different ways.

Although quadtrees have very efficient runtimes in nearest neighbor search and range search, it is not ideal for data in more than two dimensions. K-d trees are particularly useful for working with larger dimensions of data, such as color reduction. In the color reduction, we can specify x number of colors to represent a complete color image. Then, we represent colors in their Red Green Blue (RGB) values. We can create a three-dimensional k-d tree to partition the space that has the complete colors of the

image. We would continue to construct the k-d tree until the leaf node is equivalent to the x number of colors we want to represent. Lastly, we would compute the average RGB value of the x number of partitions to find a x number color palette that represents the full color image.

That being said, one of the limitations of k-d trees is that it is not easy to implement due to the structuring of the tree. Additionally, as the number of dimensions increases, the efficiency of using k-d tree's nearest neighbor search can degrade significantly. In higher dimensions, the points are farther away from each other, so when we attempt to do a query on the nearest neighbor to a particular point, the nearest point won't be close by. The search radius to the nearest neighbor will be so far that it intersects a lot of other partitions, so we end up having to search through many partitions, which is inefficient.

Uniform partitioning solves the issue of hash tables being an unordered data structure by assigning buckets based on position. This algorithm is also considerably easier to implement compared to quadtrees and k-d trees. It is important to note that in some higher-performance applications, uniform partitioning may not perform as well as the other two algorithms, so we need to consider the context in which we are using these algorithms.

References

Banik, A. [NPTEL - Special Lecture Series]. (2019, October 14). *Range Searching (KD Tree)* [Video]. YouTube. https://www.youtube.com/watch?v=WJOMlD780sg&t=175s

Hug, J. (2019). *Hug61B.* GitBook. https://joshhug.gitbooks.io/hug61b/content/

Hug, J. [Josh Hug]. (2020, October 8). *61B 2020 Lecture 19 - Multidimensional Data* [Video Playlist]. YouTube. https://www.youtube.com/playlist?list=PL8FaHk7qbOD4F7nPFfgD0dGdLos1uhUPg

Gupta, A. (2020, November 3). *K Dimensional Tree Set 1 | Search and Insert.* GeeksforGeeks. https://www.geeksforgeeks.org/k-dimensional-tree/

Gupta, A. (2020, November 3). *K Dimensional Tree Set 3 | Delete.* GeeksforGeeks. https://www.geeksforgeeks.org/k-dimensional-tree-set-3-delete/

Kakde, H. M. (2005, August 25). *Range Searching Using Kd Tree.* *http://www.cs.utah.edu/~lifeifei/cis5930/kdtree.pdf*

Kamath, A. (2018, February 7). *Quad Tree.* GeeksforGeeks. https://www.geeksforgeeks.org/quad-tree/

K-d tree. (2021, March 6). In *Wikipedia.* https://en.wikipedia.org/wiki/K-d_tree#Removing_elements

Quadtree. (2021, February 13). In *Wikipedia*. https://en.wikipedia.org/wiki/Quadtree

Space Partitioning. (2021, March 10). In *Wikipedia*. https://en.wikipedia.org/wiki/Space_partitioning

York, T. (2020, May 12). *Quadtrees for Image Compression.* Medium. https://medium.com/@tannerwyork/quadtrees-for-image-processing-302536c95c00