AN ALGORITHMIC PERSPECTIVE

Understanding K-Dimensional Trees

An In-Depth Analysis Of The Space-Partitioning Data Structure

Avinash Ravishankar Nov 5, 2020 · 6 min read ★

Photo by Faye Cornish on Unsplash

By now you'd have come across many articles on the *K*-Nearest Neighbor algorithm. Be it a basic image classifier, or a simple movie recommender app for home entertainment, or even, looking up the nearest gas station on your phone, you'll find the *k*-NN algorithm in action. It is the simplest algorithmic tool that works well for classification, regression, and mapping applications.

Like any other algorithm, the nearest neighbor also has a brute-force method that isn't practical for any real application. The alternative is to use efficient data structures like the *k-dimensional tree* or the *ball-tree* as a foundation for the search algorithm.

In this post, we'll focus on learning the *k-dimensional tree* along with basic operations to manipulate its data.

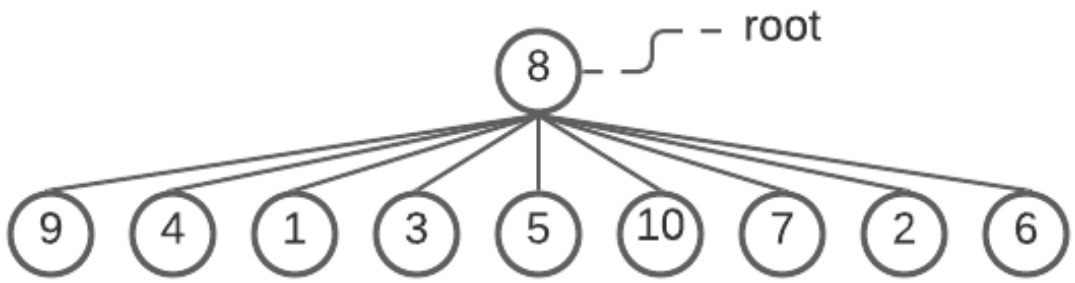
Before we dive in, let's do a quick recap of **trees** and some of its common variants.

Trees: A Quick Recap

Formally, a **tree** is defined as a finite set of one or more nodes such that:

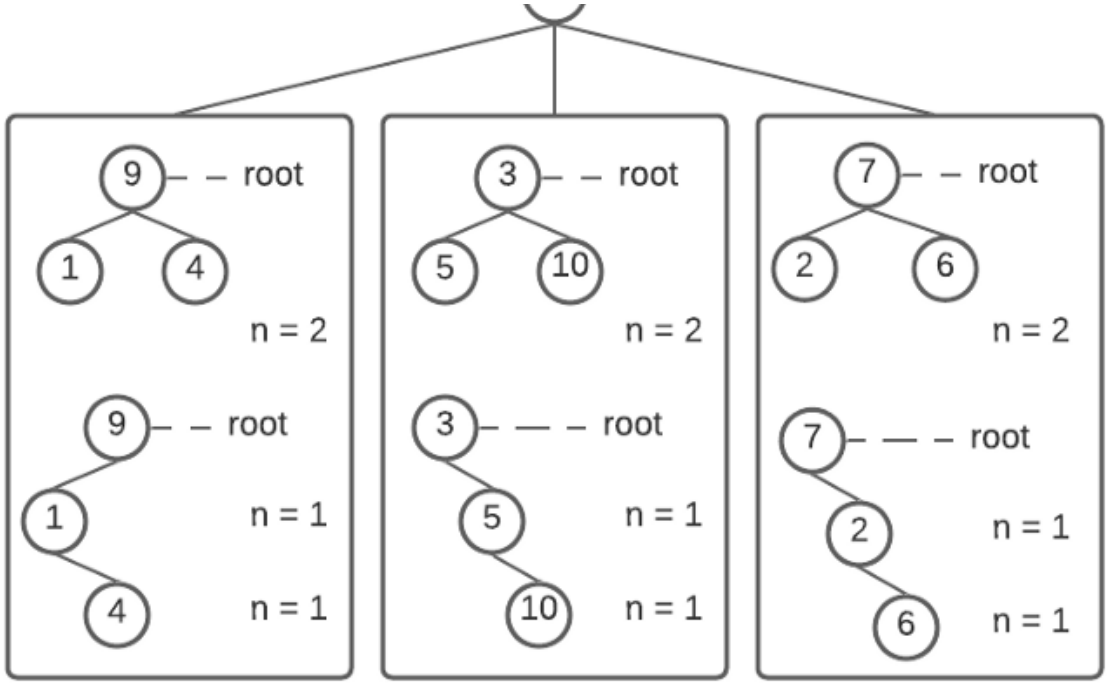
- There is a designated node called the *root*.
- The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots, T_n$ where each of these sets is a tree and called *sub-trees* of the root. [1]

For example, say, you have a set $A = \{8, 9, 4, 1, 3, 5, 10, 7, 2, 6\}$. If you choose the root to be 8 and the number of partitions $n = 9$, then, by definition, the remaining nodes in the set form independent sub-trees.



A tree with root=8 and nine sub-trees . ~Image by author

If you choose $n = 3$ as the number of partitions, you'll end up with three disjoint subsets: $B = \{9, 4, 1\}$, $\Gamma = \{3, 5, 10\}$, and $\Delta = \{7, 2, 6\}$. Once again, each subset has a node as its *root*.



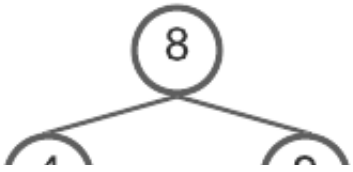
Top-half: With n = 2, each root has two children that are themselves **root** nodes. Bottom-half: With n = 1, each root has one child that intern has a **root** and a child node. ~Image by author

The root nodes **9**, **3**, **7** from the sub-trees form equal children of node **8**. Each sub-tree can be further partitioned into 1 or 2 sub-trees. The above figure shows this in detail.

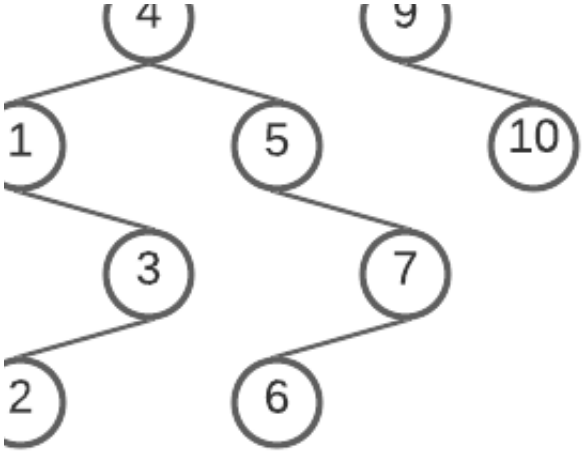
Binary Trees

A binary tree is an important tree structure you come across quite often. It is defined as a finite set of nodes that is empty or consists of a root and the left and the right sub-trees. 1

Binary Search Trees



An important use-case of the binary tree is the *Binary Search Tree* (BST).



earch Tree created from the set A. ~Image by author

Take a look at this tree for example. Any given node in the tree has a value that is greater than its left sub-tree but less than its right sub-tree. This unique property allows us to search, insert, or delete a node in O(log(n)) time.

At a high level, a kd-tree is a generalization of a BST that stores k-dimensional points. 2

With the prerequisites done, let's focus on the concept of the kd-tree.

KD-Trees: The Concept

Developed by Jon Bentley, the kd-tree is a variation of the BST. The difference is that each level of the tree branches based on a particular dimension associated with that level. 3

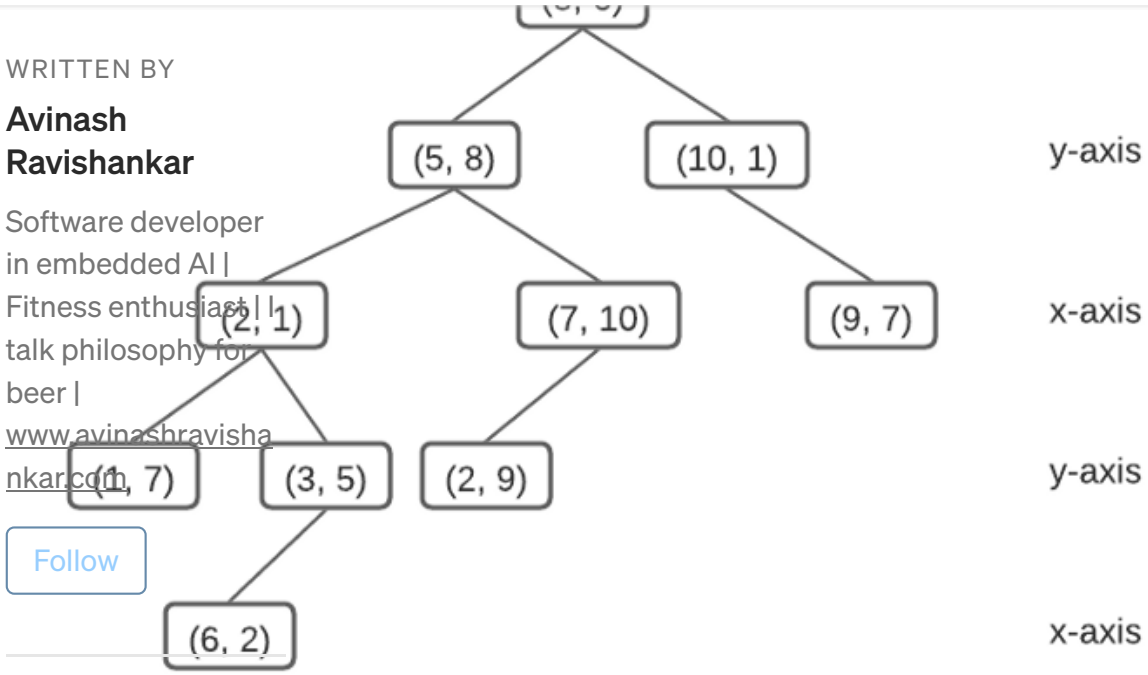
In other words, for a kd-tree *with n* levels, each level i, $\forall\, i = \{1, \ldots, n\}$, has a particular dimension $d = i \bmod k$ assigned for comparison, called the *splitting dimension*. As you traverse the tree, you compare nodes at the splitting dimension of the given

level. If it compares less, you branch left. Otherwise, you branch right. This preserves the structure of the BST irrespective of the dimensions.

Let's try out an example. Consider the set E in an arbitrary 2d-space.

E = {(8, 6), (10, 1), (5, 8), (9, 7), (2, 1), (3, 5), (1, 7), (7, 10), (2, 9), (6, 2)}

WRITTEN BY
Avinash Ravishankar
Software developer in embedded AI | Fitness enthusiast | | talk philosophy for beer | www.avinashravishankar.com
Follow

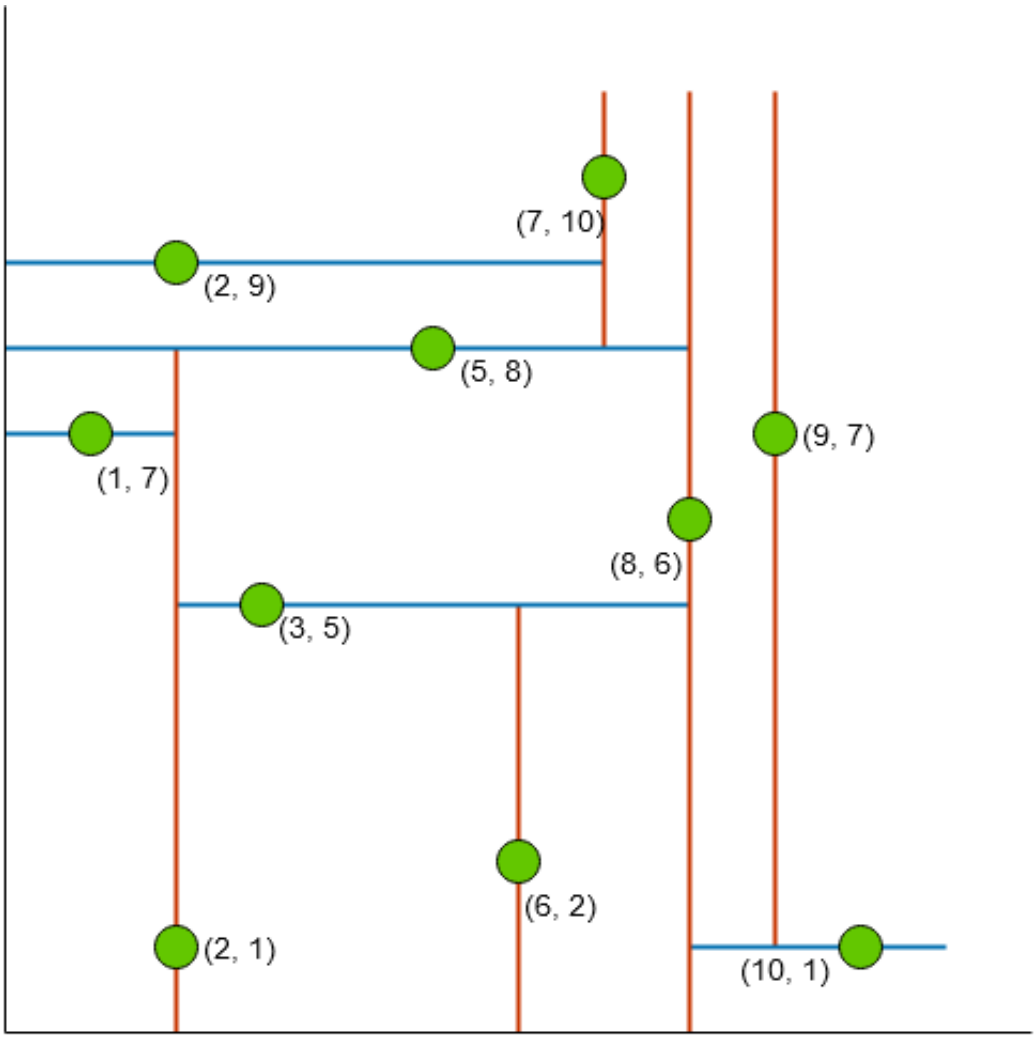


A 2-dimensional tree constructed from E. ~Image by author

With ***x*** as the splitting dimension for level 1, the tree compares nodes (**5**, 8) and (**10**, 1) with (**8**, 6) at the *x*-axis. As (5, 8) compares less, it branches to the left, whereas, (10, 1) branches

to the right. On level 2, the assigned splitting dimension is ***y***. The tree compares nodes (2, **1**) and (7, **10**) with (5, **8**) at the *y*-axis. As (2, 1) compares less, it branches to the left, whereas, (7, 10) branches to the right.

It is easy to visualize these points in a 2d-space. The figure below shows red lines partitioning the space in ***x*** and blue lines in ***y*** dimensions.



Set E visualized in 2d-space. ~Image by author

As you can see, the lines partition the 2d-space. Hence, called a

space-partitioning algorithm.

The 2-dimensional tree makes it easier to visualize and understand the fundamental concept. Yet, the data structure applies to any number of dimensions.

In this section, we'll focus on three fundamental operations on a kd-tree.

KD-Trees: The Algorithms

For the sake of clarity, let's introduce a `Node` class to store our kd-points.

A Node class to store kd-points

Insert

Inserting a node into a kd-tree is like insertion in a BST. The algorithm traverses the tree until it finds the correct position, a `None` object. As we traverse the tree, we loop through the splitting dimensions to ensure comparisons at the correct dimension for the given level.

Insert a node.

Find Minimum

To find the least node for a given dimension, you traverse its left sub-tree if the splitting dimension matches the given. If not, you traverse both its left and right sub-trees, as the minimum could be in either of them.

Find the least node for a given dimension.

Delete

To delete a node from a kd-tree, you traverse the tree until you find the required node. If it turns out to be a leaf node, delete it and return a `None` object.

If the node to delete has the right sub-tree, then,

- Find the node with the least value for the given splitting dimension.
- Replace the node with this least node (This duplicates the least node).
- Traverse the right sub-tree to find the duplicate node, i.e., min-node, and delete it.

If the node has a sub-tree to it's left but a `None` to it's right, we cannot just find the largest node, replace it, and delete its duplicate. There is no guarantee that the left sub-tree contains a unique max-node and could violate the property of *coordinate invariance*. To verify, try adding the node (7, 6) to the 2d-tree

above. The left-sub-tree of the root node will then have two 7s in the x-axis.

To get around this,

- Repeat the steps for the right sub-tree on the left.
- Swap the left and the right sub-trees of the replaced node.

Delete a node.

Conclusion

To sum up, that was a lot of recursion for one blog post! I hope it doesn't discourage you from learning the kd-tree. Let me know in the comments if this post was resourceful to you.

The complete source code in python is found here.

References

[1] E. Horowitz, S. Sahni, Fundamentals of Data Structures, Computer Science Press

[2] Assignment handout on kd-trees, Stanford

[3] OpenDSA, CS3 Data Structures and Algorithms: kd-trees

[4] C. Kingsford, Lecture notes on kd-trees, CMU

Thanks to Linda Chen.

Algorithms K Nearest Neighbors Nearest Neighbor Search Machine Learning

Kd Tree