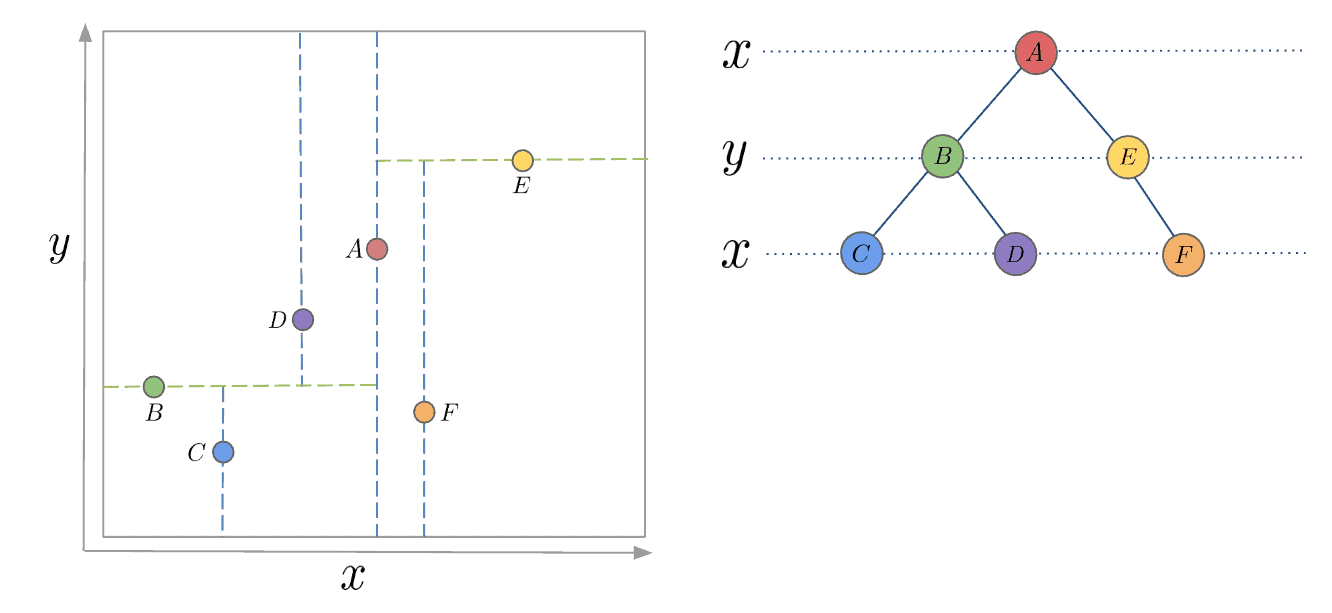
K-D Trees

**What Are K-D Trees?**

A K-D Tree is a binary tree in which each node represents a k-dimensional point**. Every non-leaf node in the tree acts as a hyperplane, dividing the space into two partitions.** This hyperplane is perpendicular to the chosen axis, which is associated with one of the K dimensions.

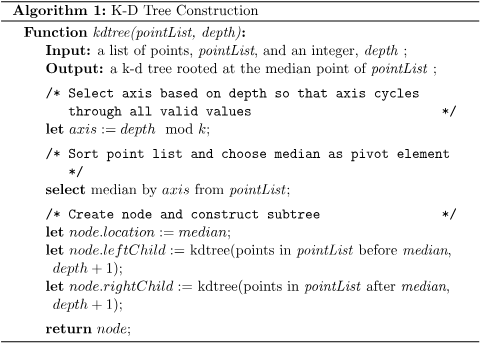
**There are different strategies for choosing an axis when dividing, but the most common one would be to cycle through each of the K dimensions repeatedly and select a midpoint along it to divide the space.** For instance, in the case of 2-dimensional points with  and  axes, we first split along the -axis, then the -axis, and then the -axis again, continuing in this manner until all points are accounted for:



**How to Build a K-D Tree?**

The construction of a K-D Tree involves recursively partitioning the points in the space, forming a binary tree. The process starts by selecting an axis. We then choose the middle(median) point along the axis and split the space, so that the remaining points lie into two subsets based on their position relative to the splitting hyperplane.

The left child of the root node is then created with the points in the subset that lie to the left of the hyperplane, while the right child is created with the points that lie to the right. This process is repeated recursively for each child node, selecting a new axis and splitting the points based on their position relative to the new hyperplane.



If the above algorithm is executed correctly, the resulting tree will be balanced, with each leaf node being approximately equidistant from the root. To achieve this, it’s essential to select the median point every time. It is also worth noting that finding the median can add some complexity, as it requires the usage of another algorithm. If the median point is not selected, there is no guarantee that the tree will be balanced.

One approach to finding the median is to use a sorting algorithm, sort the points along the selected axis and take the middle point. To address the added complexity, we can sort a fixed number of randomly selected points and use the median of those points as the splitting plane. This alternative practice can be less computationally expensive than sorting the entire array of points.

## **Inserting and Removing Nodes**

### Inserting and removing nodes are essential operations in maintaining a K-D tree’s structure and performance. However, due to the tree’s special characteristics, they must be implemented with care.

### **Insertion**

To add a new point to a K-D tree, we can follow the same process as when adding an element to any other search tree. Start by traversing the tree from the root and move to either the left or right child depending on the location of the point being inserted in relation to the splitting plane. Once we reach the node under which the child should be located, add the new point as either the left or right child of the leaf node, depending on which side of the node’s splitting plane contains the new node.

However, this method of adding points can cause the tree to become unbalanced, reducing its performance. The degree of degradation in the tree’s performance depends on the spatial distribution of the points being added and the number of points added to the tree’s size. If the tree becomes too unbalanced, it may be necessary to re-balance the tree to restore its performance for queries that rely on tree balancing, such as nearest neighbor searching.

### Deletion

When deleting a node from a K-D tree, we first need to locate the node to be deleted, just like deleting from a [Binary Search Tree](https://www.baeldung.com/cs/binary-search-trees). If the node has no children, it can be removed from the tree without affecting its structure. If the node has a descendant, we need to find the right descendant of the node that can replace it without violating the K-D tree property.

In a Binary Search Tree, we replace the node to be deleted with its existing child if one of the children is not present. However, in a K-D tree, replacing the node with its child would break the K-D tree property since the child’s dimension would be different from the node’s axis. For instance, if a node divides by x-axis values, then its children divide by the y-axis. Hence, we can’t just replace the node with its child.

o find a suitable replacement, we consider the following scenarios:

1. If the node to be deleted has children to the right, we must find the node with the minimum value along the targeted axis from the right subtree.
2. Otherwise, we need to find the point with the maximum value from the subtree rooted at the left child.

Once we’ve found the replacement node, we use it to replace the node to be deleted and then delete it recursively from the subtree it was selected from.

## **Nearest Neighbour Search**

K-D trees are widely used for nearest-neighbor searches, where the objective is to find the point in the tree that is closest to a given query point.