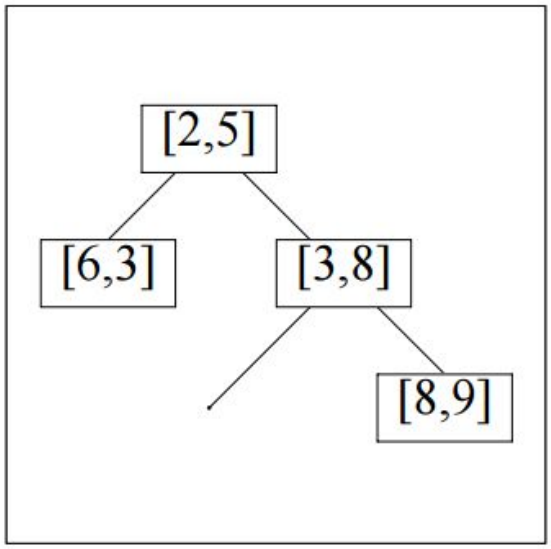
Vo Tran Thanh Luong
1551020

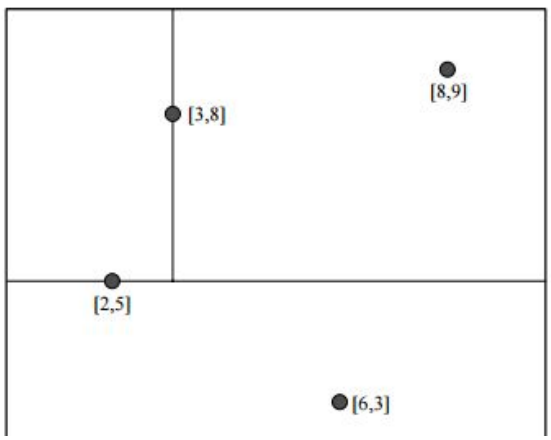
KD- TREE

I. About KD-tree as a data structure :



A kd-tree is a data structure for storing a nite set of points from a k-dimensional space.
Actually , a kd-tree is also a form of binary tree. K-d trees are a useful data structure for several applications, such as searches involving a multidimensional search key (e.g. range searches and nearest neighbor searches). However, a kd-tree cannot be used to store collections of other data types, such as strings

On the left picture, we can see an example 2-d tree of four elements.

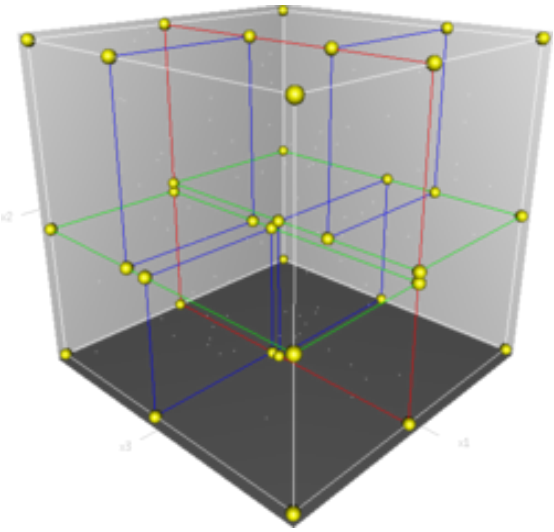


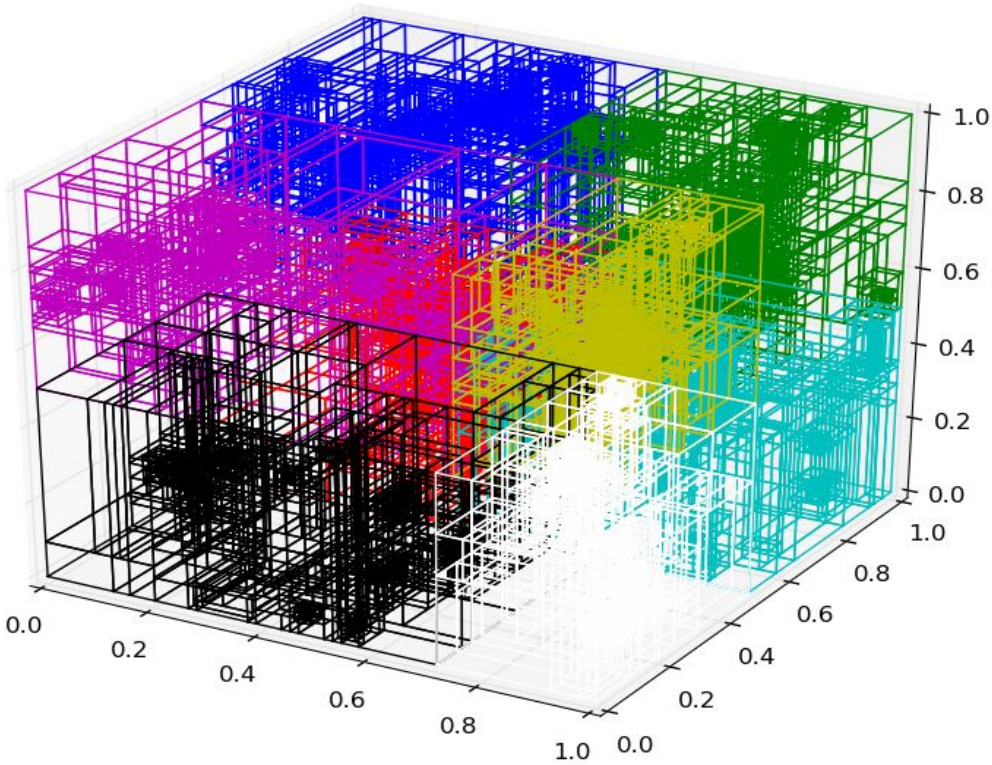
Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces. Points to the left of this hyperplane are represented by the left subtree of that node and points right of the hyperplane are represented by the right subtree.

Vo Tran Thanh Luong
1551020

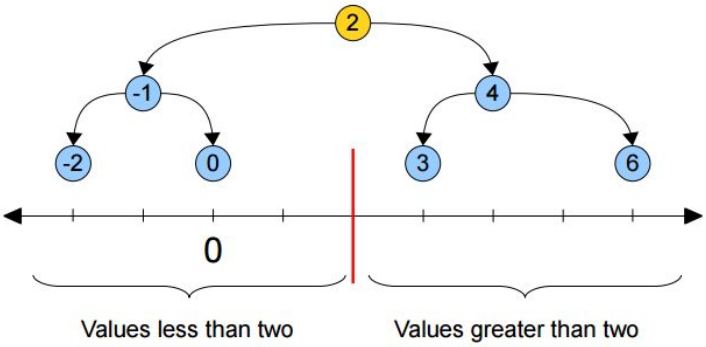
A very important note here is that all of the data stored in a kd-tree must have the same dimension.

The kd-tree has a geometric meaning behind its structure. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces. Points to the left of this hyperplane are represented by the left subtree of that node and points right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the k-dimensions, with the hyperplane perpendicular to that dimension's axis. So, for example, if for a particular split the "x" axis is chosen, all points in the subtree with a smaller "x" value than the node will appear in the left subtree and all points with larger "x" value will be in the right subtree. In such a case, the hyperplane would be set by the x-value of the point, and its normal would be the unit x-axis.

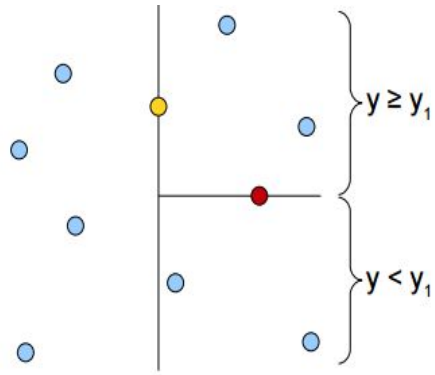


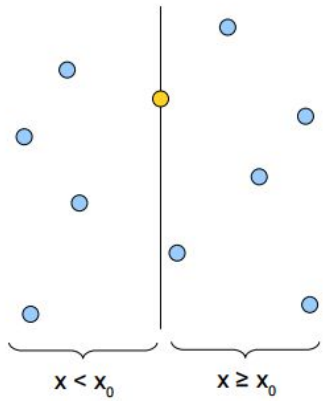


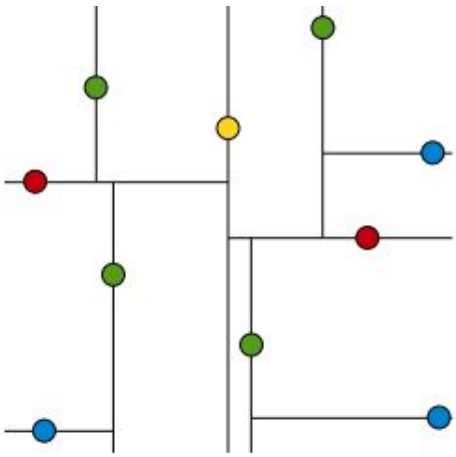
Vo Tran Thanh Luong
1551020



Why data is stored in the K-d tree in this way? Why do we have to mind its geometrical meaning? The answer is because normal binary search tree is restricted to 1 dimension. For example, if we want to abstract the data of geometric points into a tree, this is what we usually do : we take the middle point out then add points by comparing to the previous one and recursively to the root. However, when we scale up data to a higher level, it is hard to compare data in such way. So we think of a special way that the k-d tree is using now

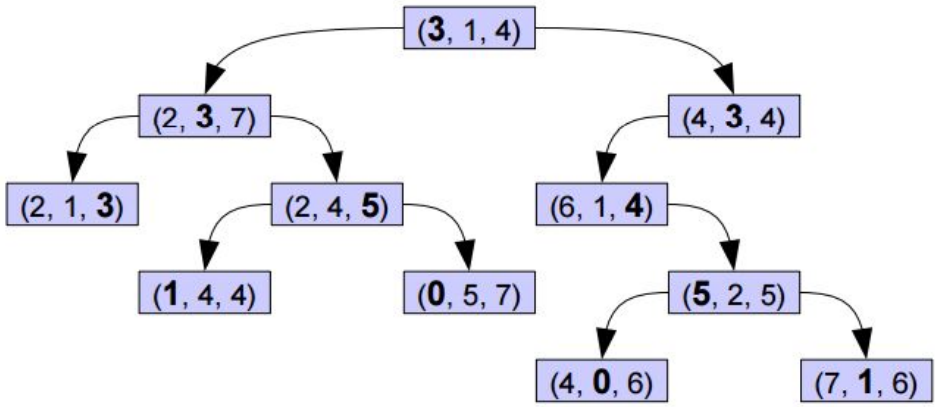






II. How to search and insert values

This is another example of the kd tree :



As we can see, inside every node of the KD-tree, there is 1 bolded element. The guideline for bolding these elements is that at level n , the element (n%3) inside every node must be bolded. Naturally, the question here is : why do we make them bold? Kd-tree, as mentioned above, works a lot like a binary search tree, but it discriminates only along the bolded component. For example, the first component of every node in the left subtree is less than the first component of the root of the tree, while the first component of every node in the right subtree has a first component at least as a large as the root node's. Similarly, consider the kd-tree's left subtree. The root of this tree has the value (2, 3, 7), with the three in bold. If you look at all the nodes in its left subtree, you'll notice that the second component has a value strictly less than three. Similarly, in the right subtree the second component of each node is at least three. This trend continues throughout the tree.

This make querying and searching for data more efficient.

For example, Given a point A, start at the root of the tree. If the root node is A, return the root node. If the first element of A is strictly less than the first element of the root, then look for A in the left subtree, this time comparing the second component of P. Otherwise, then the first component of P is at least as large as the first element of the root node, and we descend into the right subtree and next time compare the second element of P. We continue this process until we fall off the tree or find the node in question.

Inserting is just about the same as searching. It works like inserting into a normal binary search tree but we just consider, except that when move down to each level to find space to insert, we only look at one certain part of the tree.

Adding points can cause the tree to become unbalanced, leading to decreased tree performance. The rate of tree performance degradation is dependent upon the spatial distribution of tree points being added, and the number of points added in relation to the tree size. If a tree becomes too unbalanced, it may need to be re-balanced to restore the performance of queries that rely on the tree balancing, such as nearest **neighbour searching.**

Let’s talk about **neighbour searching**. The nearest neighbour search algorithm aims to find the point in the tree that is nearest to a given input point. This search can be done efficiently by using the tree properties to quickly eliminate large portions of the search space. Searching for a nearest neighbour in a k-d tree proceeds as follows:

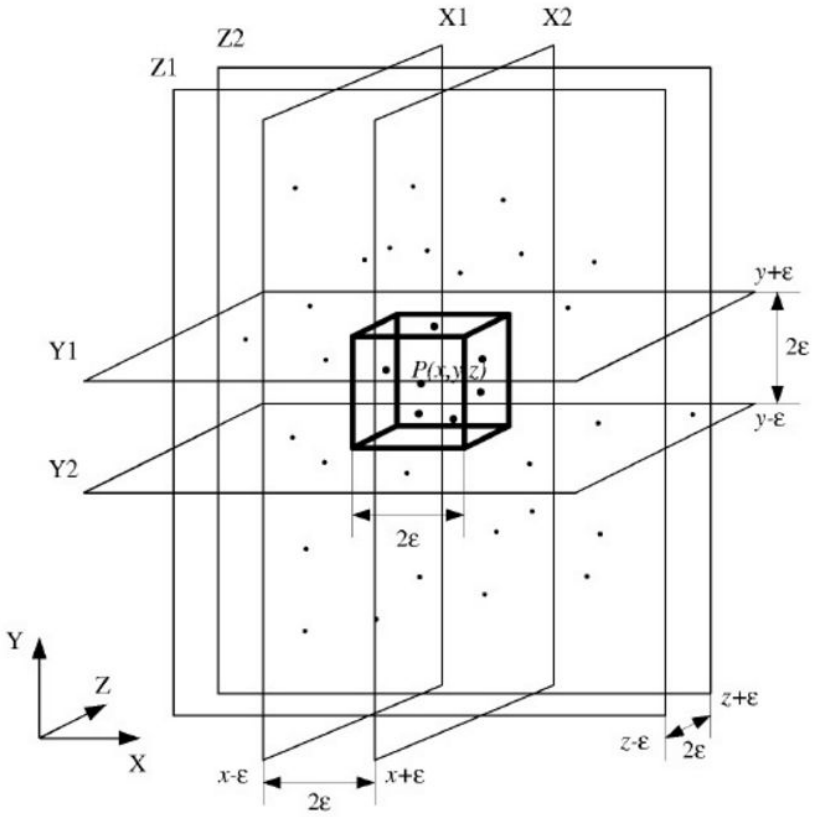
1. Starting with the root node, the algorithm moves down the tree recursively, in the same way that it would if the search point were being inserted (i.e. it goes left or right depending on whether the point is lesser than or greater than the current node in the split dimension).
2. Once the algorithm reaches a leaf node, it saves that node point as the "current best"

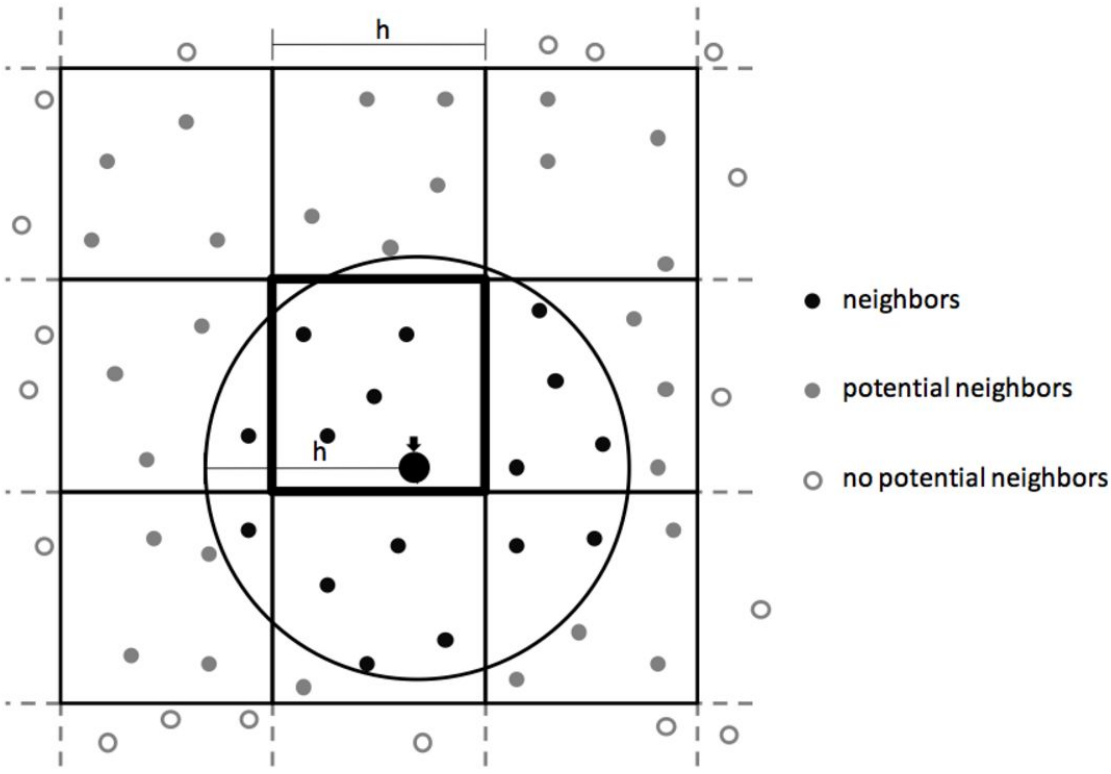
3. The algorithm unwinds the recursion of the tree, performing the following steps at each node:

- If the current node is closer than the current best, then it becomes the current best.
- The algorithm checks whether there could be any points on the other side of the splitting plane that are closer to the search point than the current best. In concept, this is done by intersecting the splitting hyperplane with a hypersphere around the search point that has a radius equal to the current nearest distance. Since the hyperplanes are all axis-aligned this is implemented as a simple comparison to see whether the distance between the splitting coordinate of the search point and current node is lesser than the distance (overall coordinates) from the search point to the current best. If the hypersphere crosses the plane, there could be nearer points on the other side of the plane, so the algorithm must move down the other branch of the tree from the current node looking for closer points, following the same recursive process as the entire search. If the hypersphere doesn't intersect the splitting plane, then the algorithm continues walking up the tree, and the entire branch on the other side of that node is eliminated.

4. When the algorithm finishes this process for the root node, then the search is complete.

Below is a visualizations of neighbor search :


X1
X2
Z2
Z1
y+ε
2ε
Y1
P(x,y,z)
y-ε
Y2
2ε
Y
Z
X
z+ε
z-ε
2ε
x-ε
x+ε
2ε


h
h
neighbors
potential neighbors
no potential neighbors

Vo Tran Thanh Luong
1551020

III. Real world application:

You can build a **MAP** that helps people get to a nearest facility using 2-D tree with Neighbour searching. All you have to do is construct a 2 dimensional k-d tree from the locations of all the facilities in your city, and then query the k-d tree to find the a certain nearest facility to any given location in the city.
KD-tree is widely used in **image rendering** and **data abstraction**. **CAD, games, movies, virtual reality, databases, GIS,** … all use KD-tree.
Biology uses K-d tree to store data of gene and DNA.
Astronomy with K-d tree and its algorithm can been used to solve a near neighbor problem for cross-identification of huge catalogs and realize the classification of astronomical objects.
Physics uses K-d tree in many ways, such as simulate light transport through spray generated by a wave.
Even **chemistry** can apply K-d tree to store data of atoms.