CMSC 420: KD Trees

Justin Wyss-Gallifent

October 19, 2023

1 Introduction

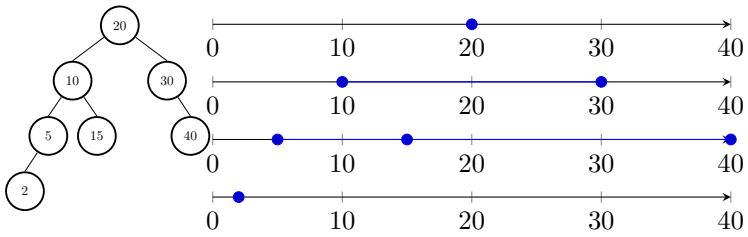
All of the data we've stored so far has been premised on 1-dimensional keys. Essentially we've been using integers but any set which can be ordered would work just as well.

Now we'd like to investigate how we might construct a tree (or tree-like structure) to store keys which are in n-dimensional space. For example if the keys were things like $(40, 32)$ or $(81, 3)$ in the 2-dimensional case.

2 Initial Ideas

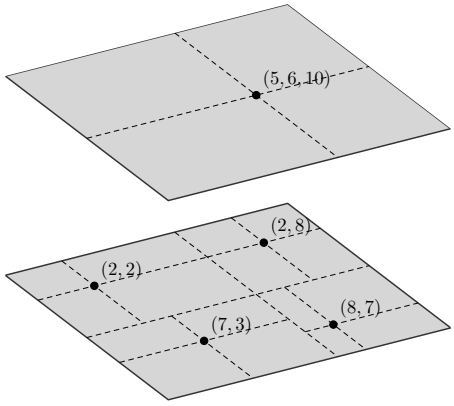
2.1 More Dimension Solution

A standard binary tree can be thought of as a layered collection of x-axis whereby each x-axis corresponds to a layer of the tree and each set of points is divided by the points above it:

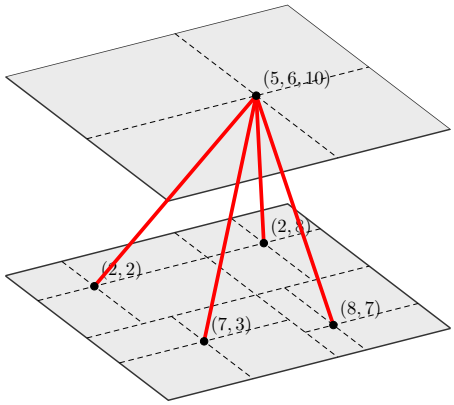


In light of this we could store a collection of points using layers of planes. For example if the root node contains $(5, 6)$ and the second level contains the points $(2, 2)$, $(7, 3)$, $(2, 8)$, and $(8, 7)$ we could draw the following. The dashed lines are just there to show how the points relate.

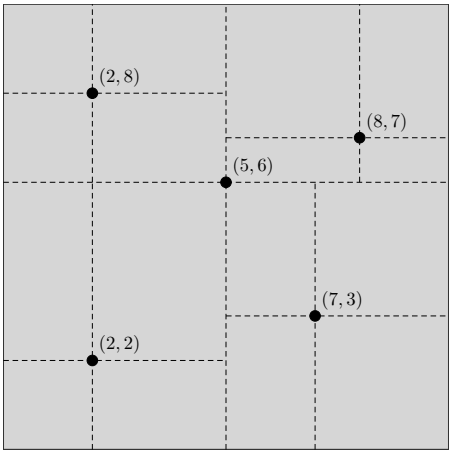
Note that the points are placed in the four quadrants relative to the root depending on how their x- and y-values relate to the parent node.



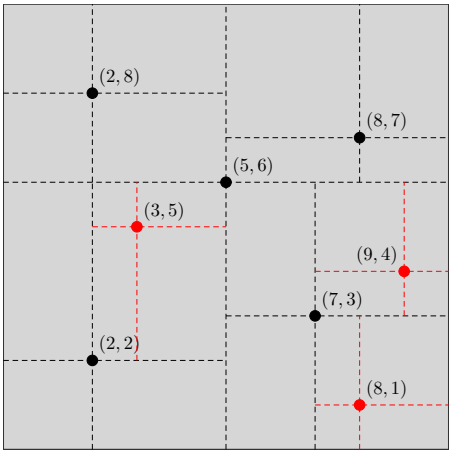
Then as a tree this would be the following, where the red lines are the ones connecting the root node to its four children and the planes and lines are just left for visual reference but of course are not part of the tree:



These stacked planes can also be simply drawn on top of each other. Here we have drawn dashed lines to show how each point has four related quadrants. Each point may have zero or one child in each quadrant, in our case we have exactly one in each.

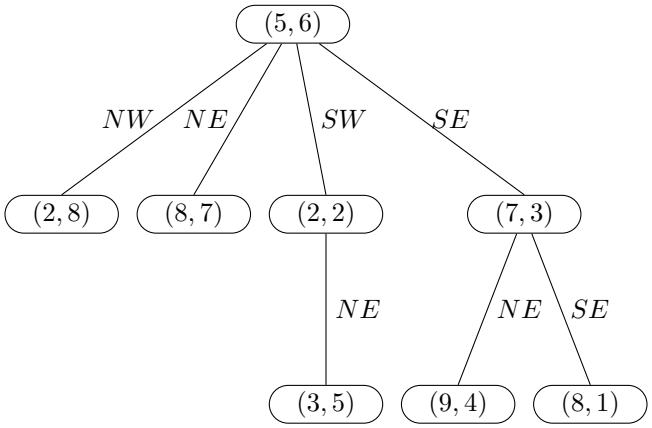


Then if we wish to add further children we just add them in the sub-rectangles, for example let's give $(7, 3)$ two children $(8, 1)$ and $(9, 4)$, and $(2, 2)$ one child $(3, 5)$:



2.2 Multi-Child Solution

The above picture may be pretty but it does little to instruct us on how to code it. One solution might be for each node to have up to four children, one for each combination of how their x- and y-values relate. For example a north-east (NE) child would have larger x- and y-values whereas a SE child would have smaller x- and y-values. For example the above picture would then become:



This isn't a bad approach but we can easily see that it will become somewhat intractible if we add more dimensions. Moreover it's not clear how we might implement some of our dictionary operations.

3 Standard Approach

3.1 Alternating Split Solution

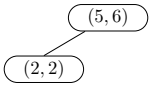
Let's head back to binary trees! We like binary trees because they're easy to comprehend. But how can we store this sort of data in a binary tree?

In the 2D case, what we'll do is this: When we go from the root level to the next level we will go left or right depending on the x-value. For the next level we will go left or right depending on the y-value. Then we switch back to x, then y, and so on.

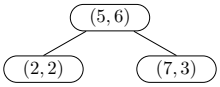
Let's insert all of the above points into such a tree in the order given here:

$$(5, 6),\ (2, 2),\ (7, 3),\ (2, 8),\ (8, 7),\ (8, 1),\ (9, 4),\ (3, 5)$$

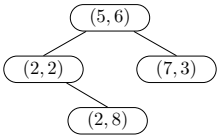
The root node is $(5, 6)$. This root level splits by x-value so when we insert $(2, 2)$ we go left:



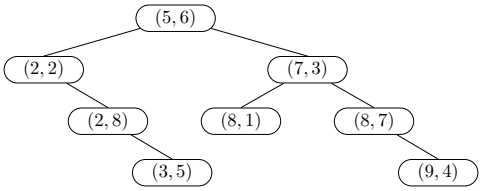
When we insert $(7, 3)$ we first ask how its x-value compares to $(5, 6)$. It's larger, so we go right and place it there:



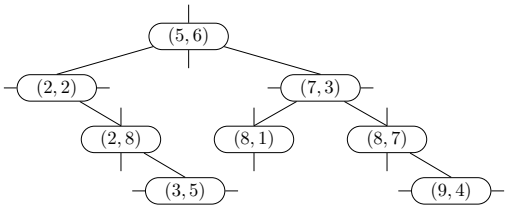
Now when we insert $(2, 8)$ we first ask how its x-value compares to $(5, 6)$. It's smaller so we go left and reach $(2, 2)$. Then we ask how its y-value compares to $(2, 2)$. It's larger so we go right and place it there:



If we continue this process we get the final tree:



Traditionally in order to keep track of the splits we decorate the nodes. We put a vertical decoration to indicate that we're splitting by x-coordinate and a horizontal decoration to indicate that we're splitting by y-coordinate:



3.2 More than 2 Dimensions

One advantage to this approach is that it's easy to generalize to more dimensions. For example when managing points in 3D we can rotate our splitting first by x, then y, then z, then x, then y, and so on.

3.3 Duplicate Coordinates

Typically when keys consist of a single value they are unique, and in this case while the points will be unique, specific coordinates may not be. For example we must allow both $(20, 30)$ and $(20, 40)$ in the 2D-case.

By convention we will say that if we are traveling down the tree to insert a node and we are at a node where we split by the α-coordinate and the node we wish to insert has the same coordinate value then we go right.

Note 3.3.1. α in this case represents any coordinate, think of it as a coordinate variable.

For example if we are inserting $(20, 30)$ and we encounter the node $(20, 40)$ where the split is by the x-coordinate then we go right.

We could just have well decided to go left, with some appropriate changes later, but the point is that we must make a choice so that we know how to handle search, insert, and as we'll see, delete.

To simplify later discussion we'll use the phrase *identical coordinates split right*.

4 Search

4.1 Search Algorithm

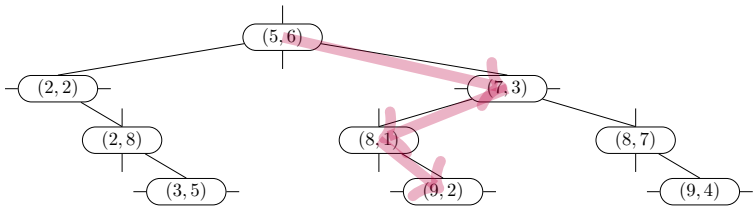
Search in a kd-tree works exactly like a binary tree with the adjustment that we follow the branch according to the splitting coordinate for each node.

5 Insert

5.1 Insertion Algorithm

Insertion in a kd-tree works exactly like a binary tree with the adjustment that we follow the branch according to the splitting coordinate for each node and we place the node as a leaf in the correct position with regards to the correct splitting coordinate at the end.

Example 5.1. For example the insertion of $(9, 2)$ into our kd-tree from earlier would look proceed as shown. We start at the root and compare x-coordinates and go right. At $(7, 3)$ we compare y-coordinates and go left. At $(8, 1)$ we compare x-coordinates and insert right. Notice that the inserted node has its split direction set appropriately for the level it is on.



6 Delete

6.1 Finding Replacements

With BSTs we regularly did replacements using a node's inorder successor. This successor was easy to find - go right (if possible) then left as far as possible. However this algorithm is premised on the fact that we are always splitting by the single key and with a kd-tree this is no longer the case.

Let's abstract the procedure and observe that when we are looking for a node's inorder successor we are looking for the smallest key in the right subtree.

So let's imagine we simply have a kd-tree and wish to find a node with smallest α-coordinate. Here α could be x, or y, or whatever.

Note 6.1.1. The following can be modified to find a node with largest α-coordinate, thereby emulating finding a node's inorder predecessor, but this should not be taken lightly and is commented on later.

The following procedure is recursive:

- If we are at a node which splits by α then the target is in the left subtree. Recall that equal coordinates go right so the left subtree consists only of points with α-coordinate strictly less than the node's α-coordinate. If that left subtree is NULL then just return the current node. Otherwise recurse to the root of the left subtree.
- If we are at a node which splits by something other than α then the target could be in either subtree or the node itself. We recurse to both subtrees, take the multiple results as well as the node itself and pick the one with minimum α-coordinate. If there are several then we can pick any of them.

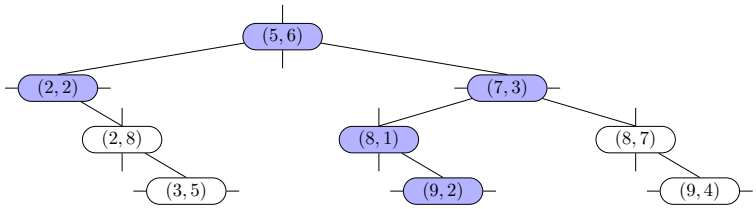
Example 6.1. Suppose in the most recent tree we wish to find the node with minimum y-coordinate. We start at $(5, 6)$ but since it splits by x-coordinate we have to check both subtrees.

We recurse to the subtree rooted at $(2,2)$, note that $(2,2)$ splits by y-coordinate and so we need to branch left. We cannot, however, so we simply return $(2,2)$.

We recurse to the subtree rooted at $(7,3)$, note that $(7,3)$ splits by y-coordinate and so we need to branch left. We branch left to $(8,1)$ but since it splits by x-coordinate we have to check both subtrees but there is only one, and that returns $(9,2)$.

So the $(8,1)$ subtree returns the minimum of $(8,1)$ and $(9,2)$, where minimum means minimum y-coordinate, so that's $(8,1)$. The $(7,3)$ subtree returns this same value. The $(5,6)$ subtree returns the minimum of $(2,2)$, $(8,1)$ and $(5,6)$, so that's $(8,1)$.

This can be illustrated in the following diagram where we shade the nodes that we actually have to analyze. The recursive procedure takes the minimum up the tree:



6.2 Deletion Algorithm

In the 1-D case the process was simple - go right then as far left as possible to find the inorder successor, replace the deleted node with that inorder successor, then either splice the tree (if the inorder successor was not a leaf) or just chop off the old inorder successor node (if it was).

This won't work here because splicing the tree will result in the lower part of the splice having all its splitting directions messed up. So what can we do?

Note 6.2.1. Before proceeding we should comment on something immediately. The fact that we decided that equal coordinates go right forces us to use the inorder successor and not the inorder predecessor. The reason for this will be clarified a bit later, however it is worth noting that if we had decided that equal coordinates go left then we would be forced to use the inorder predecessor.

Here we'll use subscripts to denote coordinates, so u_α will mean the α coordinate of the point/node u. eg. $(17, 42, 100)_x = 17$ and so on.

Our deletion will work as follows; this process is recursive, assume the node to be deleted is u:

(a) If u is a leaf: Just delete it and we are done.

(b) If u has a right subtree: Let α be the coordinate that u splits on. Find a replacement node r in $u.right$ for which r_α is minimal. We copy r's point

to u (overwriting u's point). We then recursively call delete on the old r.

(c) If u does not have a right subtree then it has a left subtree: Let α be the coordinate that u splits on. Find a replacement node r in $u.left$ for which r_α is minimal. We copy r's point to u (overwriting u's point) and then we move u's left subtree to become u's right subtree. We then recursively call delete on the old r.

A few notes of importance:

Note 6.2.2. Observe that in (b) since r has minimal α-coordinate of all nodes in $u.right$ when we copy r to u there are no problems with how it splits with regards to its descendents since the entries in $u.left$ all have α-coordinate less than r_α and the entries in $u.right$ all have α-coordinate greater than or equal to r_α.

Note 6.2.3. Let us pay a bit more attention to (c) to make sure we understand why it works.

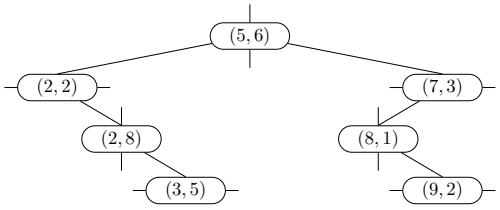
All the other nodes in $u.left$ have α-coordinate greater than or equal to r_α and so when $u.left$ is moved over to $u.right$ all the nodes in $u.right$ are correctly positioned with respect to r_α.

Moreover since this is simply a horizontal movement of the subtree there are no issues with levels being out of sync with regards to their own splitting coordinates, meaning the descendents are all fine.

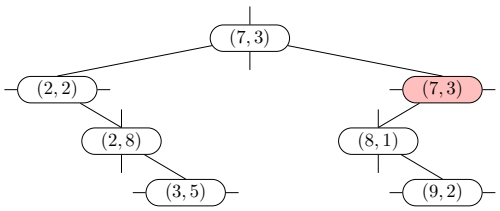
Note 6.2.4. Moreover with regards to ancestors observe that for any coordinate β if u has an ancestor which splits on β then r was already positioned correctly with regards to that ancestor (since it's lower on the tree than that ancestor) and this will not change when we copy it.

Note 6.2.5. It's tempting to think that instead of (c) we could just find the inorder predecessor in the left subtree. However returning to our comment earlier this can mess up our insistance that identical coordinates branch right. You encouraged to try to draw an example.

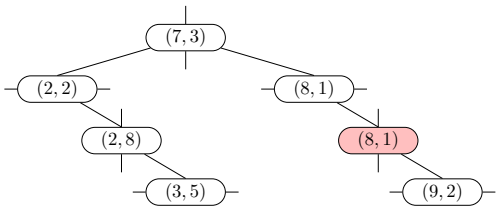
Example 6.2. Let's delete $(5, 6)$ from this tree:



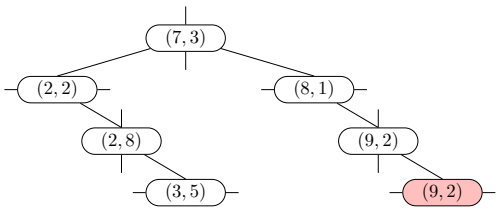
Since $(5, 6)$ splits on the x-coordinate and has a right subtree we pick the node in the right subtree with minimum x-coordinate, this is $(7, 3)$. We replace. There are temporarily two copies of $(7, 3)$, the red one below is next in line for replacement.



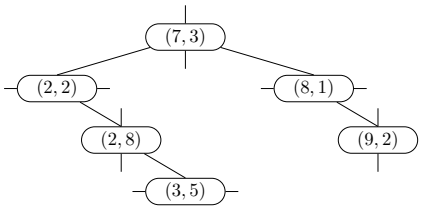
Since the red $(7, 3)$ splits on the y-coordinate but has no right subtree we pick the node in the left subtree with minimum y-coordinate, this is $(8, 1)$. We replace and then move the old left subtree to the right: There are now temporarily two copies of $(8, 1)$, the red one below is next in line for replacement.



Since the red $(8, 1)$ splits on the x-coordinate and has a right subtree we pick the node in the right subtree with minimum x-coordinate, this is $(9, 2)$. We replace. There are temporarily two copies of $(9, 2)$, the red one below is next in line for replacement.



Since the red $(9, 2)$ is a leaf we chop it off and we are done:



7 Modifications

There are several ways that one could modify a kd-tree. Some examples are:

1. Trying to keep the tree balanced. This is tricky because we cannot use rotations since they ruin our splitting according to level. Theoretically it's possible to take a scapegoat tree approach and rebuild subtrees.

2. Much like a B+ tree it not uncommon to modify a kd-tree so that the data is all in the leaf nodes and the internal nodes are used simply as guideposts to lead us to the data. The crucial benefit to this is that it makes it easier to insert and delete keys since we don't have to go through the convoluted deletion process.
3. As a modifcation of the above we can permit a leaf node to contain several points, generally a small number, stored in some other method (a list, a linked list, etc.) which we simply work with in a more traditional manner.

8 Time Complexity

8.1 Height

In the worst-case the tree would essentally be a linked list and the height would be $\mathcal{O}(n)$.

In the best case it would be well-balanced and the height would be $\mathcal{O}(\lg n)$.

The average case is a little tricker. Recalling when studying unbalanced binary trees that if we systematically choose an inorder success when hunting for replacements that the height turns out to be $\mathcal{O}(\sqrt{n})$. In the KD-tree case the fact that we have equal coordinates going right (which leads to additional bias in deletion) suggests the same outcome.

8.2 Search and Insert

Search and insert are dependent on the height of the tree so the worst, best, and average cases are identical.

8.3 Delete

Delete is a little tricker because of our necessity to search subtrees for replacement nodes.

It's certainly the case that each search is $\mathcal{O}(n)$ and there are certainly not more than $n/$ of them, hence worse-case $\mathcal{O}(n^2)$, but this can certainly be brought down through more careful analysis.