kd-Trees

## Invented in 1970s by Jon Bentley

* Name originally meant “3d-trees, 4d-trees, etc” where k was the # of dimensions

## Now, people say “kd-tree of dimension d”

* Idea: Each level of the tree compares against 1 dimension.
* Let’s us have only **two children** at each node (instead of 2*d*)

## Each level has a “cutting dimension”

x

y

x

x

* Cycle through the dimensions as you walk down the tree.

## Each node contains a point P = (x,y)

* To find (x’,y’) you only y

## compare coordinate from the

cutting dimension

- e.g. if cutting dimension is x, then you ask: is x’ < x?

# kd-tree example

## insert: (30,40), (5,25), (10,12), (70,70), (50,30), (35,45)

x

30,40

5,25

70,70

10,12

50,30

35,45



(70,70)

(35,45)

0,40)

(5,25)

(50,30)

(10,12)

(3

## y x y

insert(Point x, KDNode t, int cd) {

**if** t == **null**

t = **new** KDNode(x)

**else if** (x == t.data)

// error! duplicate

**else if** (x[cd] < t.data[cd])

t.left = insert(x, t.left, (cd+1) % DIM)

##### else

t.right = insert(x, t.right, (cd+1) % DIM)

##### return t

}

## FindMin(d): find the point with the smallest value in the dth dimension.

* Recursively traverse the tree

## If cutdim(current\_node) = d, then the minimum can’t be in the right subtree, so recurse on just the left subtree

* + if no left subtree, then current node is the min for tree rooted at this node.

## If cutdim(current\_node) ≠ d, then minimum could be in *either* subtree, so recurse on both subtrees.

* + (unlike in 1-d structures, often have to explore several paths down the tree)

## FindMin(x-dimension):

x

51,75

25,40

70,70

10,30

35,90

55,1

60,80

1,10

50,50

|  |  |  |
| --- | --- | --- |
|  | |  |
| (35,90) | | (60,80) |
|  | | (51,75) |
|  | | (70,70) |
| (50,50) | |  |
| (25,40) | |  |
|  | (10,30) |  |
| (1, | 10) |  |
|  |  | (55,1) |



## y x y

FindMin(y-dimension):

## x

51,75

25,40

70,70

10,30

35,90

5555,1,1

60,80

11,1,100

50,50

|  |  |  |
| --- | --- | --- |
|  | |  |
| (35,90) | | (60,80) |
|  | | (51,75) |
|  | | (70,70) |
| (50,50) | |  |
| (25,40) | |  |
|  | (10,30) |  |
| (1, | 10) |  |
|  |  | (55,1) |



y x y

## FindMin(y-dimension): space searched

x

51,75

25,40

70,70

10,30

35,90

55,1

60,80

1,10

50,50

|  |  |  |
| --- | --- | --- |
|  | |  |
| (35,90) | | (60,80) |
|  | | (51,75) |
|  | | (70,70) |
| (50,50) | |  |
| (25,40) | |  |
|  | (10,30) |  |
| (1, | 10) |  |
|  |  | (55,1) |



## y x y

Point findmin(Node T, **int** dim, **int** cd):

// empty tree

**if** T == **NULL**: **return NULL**

// T splits on the dimension we’re searching

// => only visit left subtree

**if** cd == dim:

**if** t.left == **NULL**: **return** t.data

**else return** findmin(T.left, dim, (cd+1)%DIM)

// T splits on a different dimension

// => have to search both subtrees

##### else:

**return** minimum(

findmin(T.left, dim, (cd+1)%DIM), findmin(T.right, dim, (cd+1)%DIM) T.data

)

Want to delete node A.



**A**

Q

cd

**B**

P

Assume cutting dimension of A is cd

#### In BST, we’d

*findmin*(A.right). cd

#### Here, we have to

*findmin*(A.right, cd)

#### Everything in Q has cd-coord < B, and everything in P has cd-

coord ≥ B

#### What is right subtree is empty?



* Possible idea: Find the *max*

#### in the left subtree?

- Why might this not work?

#### Suppose I findmax(T.left) and get point (a,b):

x **(x,y)**

#### It’s possible that T.left contains *another* point with x = a.

Now, our equal coordinate invariant is violated!

**(a,c)**

Q

cd **(a,b)**

#### Swap the subtrees of node to be deleted



* *B = find****min***(T.left)

#### Replace deleted node by B

x **(x,y)**

#### Now, if there is another point with x=a, it appears in the right subtree, where it should

Q

cd **(a,b)**

### (a,c)

Point delete(Point x, Node T, **int** cd): **if** T == **NULL**: **error** point not found! next\_cd = (cd+1)%DIM

// This is the point to delete:

**if** x = T.data:

// use min(cd) from right subtree:

**if** t.right != NULL:

t.data = findmin(T.right, cd, next\_cd) t.right = delete(t.data, t.right, next\_cd)

// swap subtrees and use min(cd) from new right:

**else if** T.left != NULL:

t.data = findmin(T.left, cd, next\_cd) t.right = delete(t.data, t.left, next\_cd)

##### else

t = null // we’re a leaf: just remove

// this is **not** the point, so search for it:

**else if** x[cd] < t.data[cd]:

t.left = delete(x, t.left, next\_cd)

##### else

t.right = delete(x, t.right, next\_cd)

##### return t



* Nearest Neighbor Queries are very common: given a point Q find the point P in the data set that is closest to Q.
* Doesn’t work: find cell that would contain Q and return the point it contains.

- Reason: the nearest point to P in space may be far from P in the tree:

- E.g. NN(52,52):

|  |  |  |
| --- | --- | --- |
|  | |  |
| (35,90) | | (60,80) |
|  | | (51,75) |
|  | | (70,70) |
| (50,50) | |  |
| (25,40) | |  |
|  | (10,30) |  |
| (1 | ,10) |  |
|  |  | (55,1) |

51,75

25,40

70,70

10,30

35,90

55,1

60,80

1,10

50,50

* + Idea: traverse the whole tree, **BUT make two modifications to prune to search space:**

## Keep variable of closest point C found so far. Prune subtrees once their bounding boxes say that they can’t contain any point closer than C

1. Search the subtrees in order that maximizes the chance for pruning

Query Point Q



d

Bounding box

of subtree rooted at T

If d > dist(C, Q), then no point in BB(T) can be closer to Q than C.

Hence, no reason to search subtree rooted at T.

Update the best point so far, if T is better:

T

if dist(C, Q) > dist(T.data, Q), C := T.data

Recurse, but start with the subtree “closer” to Q:

First search the subtree that would contain Q if we were inserting Q below T.

# Nearest Neighbor, Code

best, best\_dist are global var (can also pass into function calls)

**def** NN(Point Q, kdTree T, **int** cd, Rect BB):

// if this bounding box is too far, do nothing

**if** T == NULL **or** distance(Q, BB) > best\_dist: **return**

// if this point is better than the best: dist = distance(Q, T.data)

|  |  |  |  |
| --- | --- | --- | --- |
| **if** | dist | < | best\_dist: |
|  | best | = | T.data |

best\_dist = dist

// visit subtrees is most promising order:

**if** Q[cd] < T.data[cd]:

NN(Q, T.left, next\_cd, BB.trimLeft(cd, t.data)) NN(Q, T.right, next\_cd, BB.trimRight(cd, t.data))

##### else:

NN(Q, T.right, next\_cd, BB.trimRight(cd, t.data)) NN(Q, T.left, next\_cd, BB.trimLeft(cd, t.data))

Following Dave Mount’s Notes (page 77)

# Nearest Neighbor Facts

## Might have to search close to the whole tree in the worst case. [O(n)]

* In practice, runtime is closer to:
  + O(2d + log n)
  + log n to find cells “near” the query point
  + 2d to search around cells in that neighborhood

## Three important concepts that reoccur in range / nearest neighbor searching:

* + *storing partial results*: keep best so far, and update

#### *pruning*: reduce search space by eliminating irrelevant trees.

* + *traversal order*: visit the most promising subtree first.