Algorithms and Data Structures for Data Science

Nearest Neighbor Search

CS 277

March 29, 2023

Brad Solomon

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Learning Objectives

Review BST implementations

Discuss applications of BSTs

Introduce nearest neighbor search using images

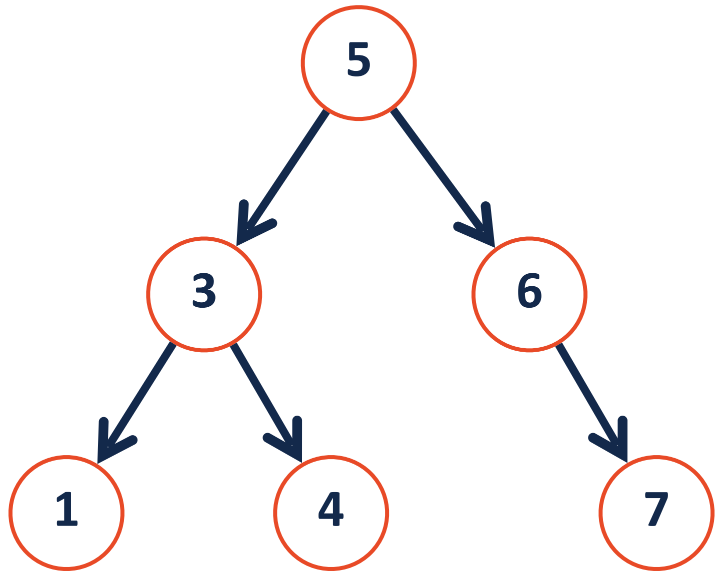
An overview of the KD-Tree **(You will not be implementing!)**

An overview of the Huffman tree

BST Find

find(4)

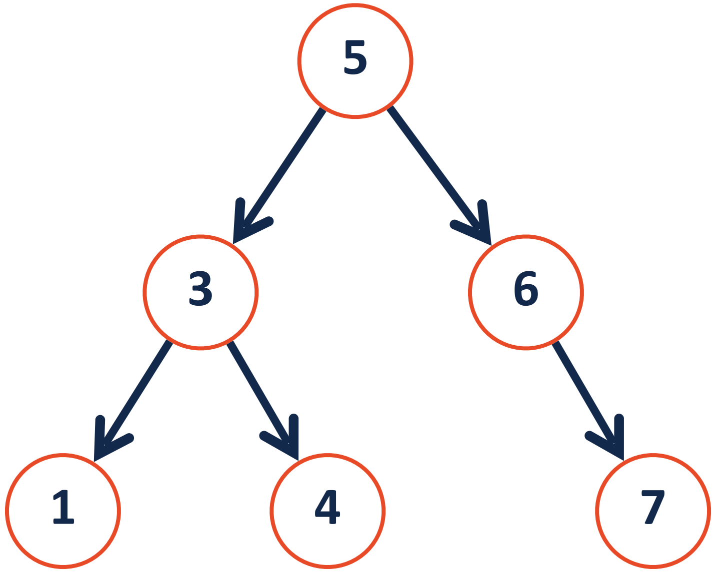
```
def find_helper(node,key):
    if not node:
        return None

    if node.key == key:
        return node

    if node.key > key:
        return find_helper(node.left, key)

    if node.key < key:
        return find_helper(node.right, key)
```



BST Find

find(4)

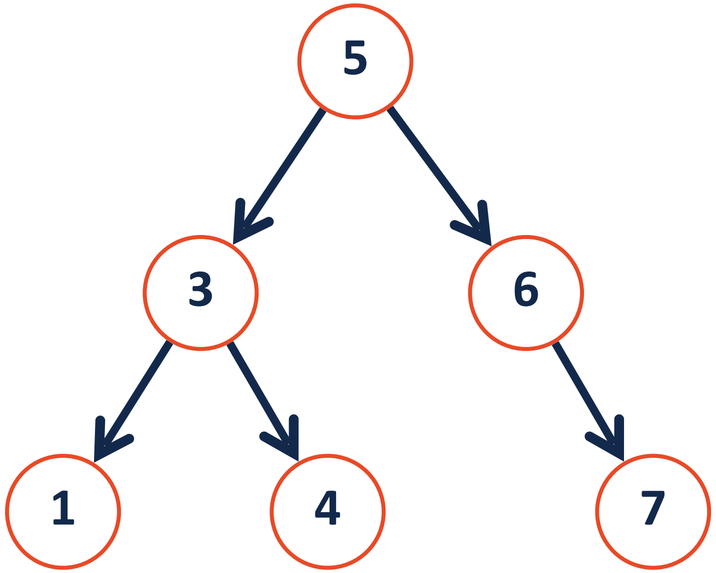
```
def find_helper(node,key):
    if not node:
        return None

    if node.key == key:
        return node

    if node.key > key:
        find_helper(node.left, key)

    if node.key < key:
        find_helper(node.right, key)
```



BST Insert

insert(5)

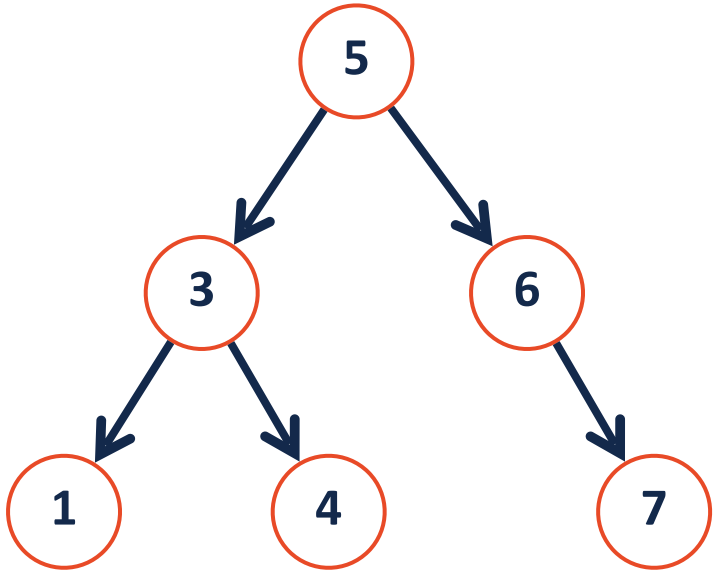
```
def insert_helper(node, key, value):
    if node == None:
        return bstNode(key, value)

    if node.key > key:
        node.left = insert_helper(node.left, key, value)
    if node.key < key:
        node.right = insert_helper(node.right, key, value)
    return node
```



BST Insert

insert(5)

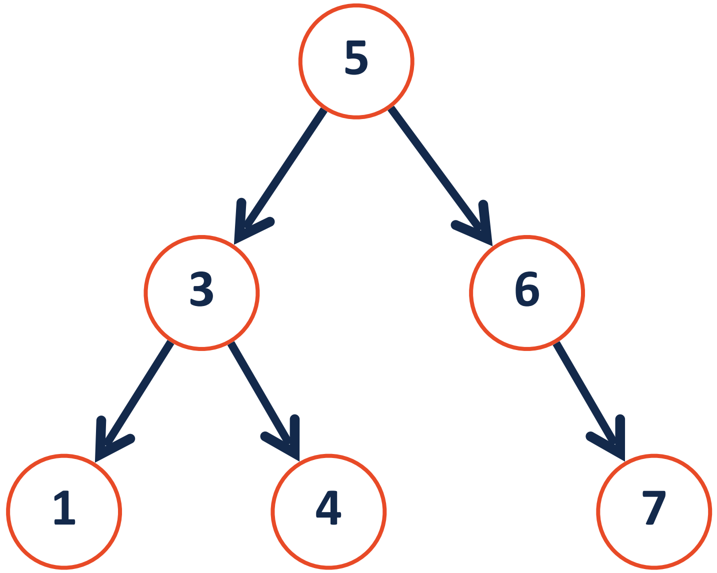
```
def insert_helper(node, key, value):
    if node == None:
        return bstNode(key, value)

    if node.key > key:
        insert_helper(node.left, key, value)
    if node.key < key:
        insert_helper(node.right, key, value)
    return node
```



BST Insert

insert(5)

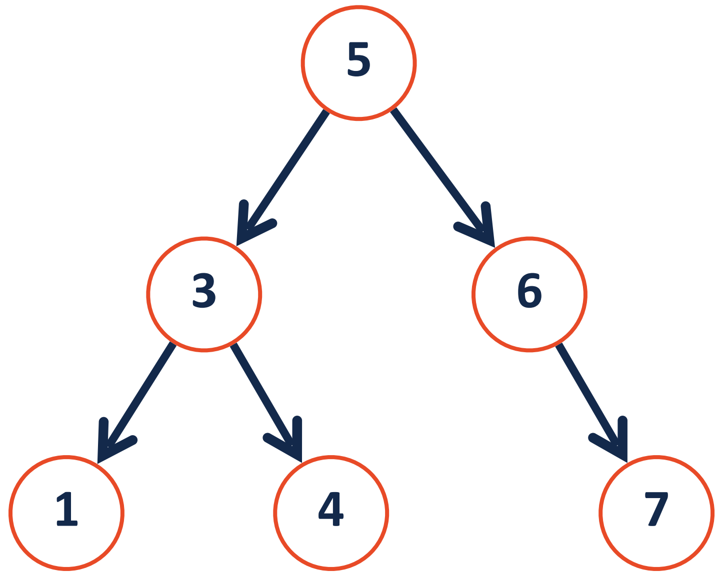
```
def insert_helper(node, key, value):
    if node == None:
        return bstNode(key, value)

    if node.key > key:
        node.left = insert_helper(node.left, key, value)
    if node.key < key:
        node.right = insert_helper(node.right, key, value)
```



BST Remove

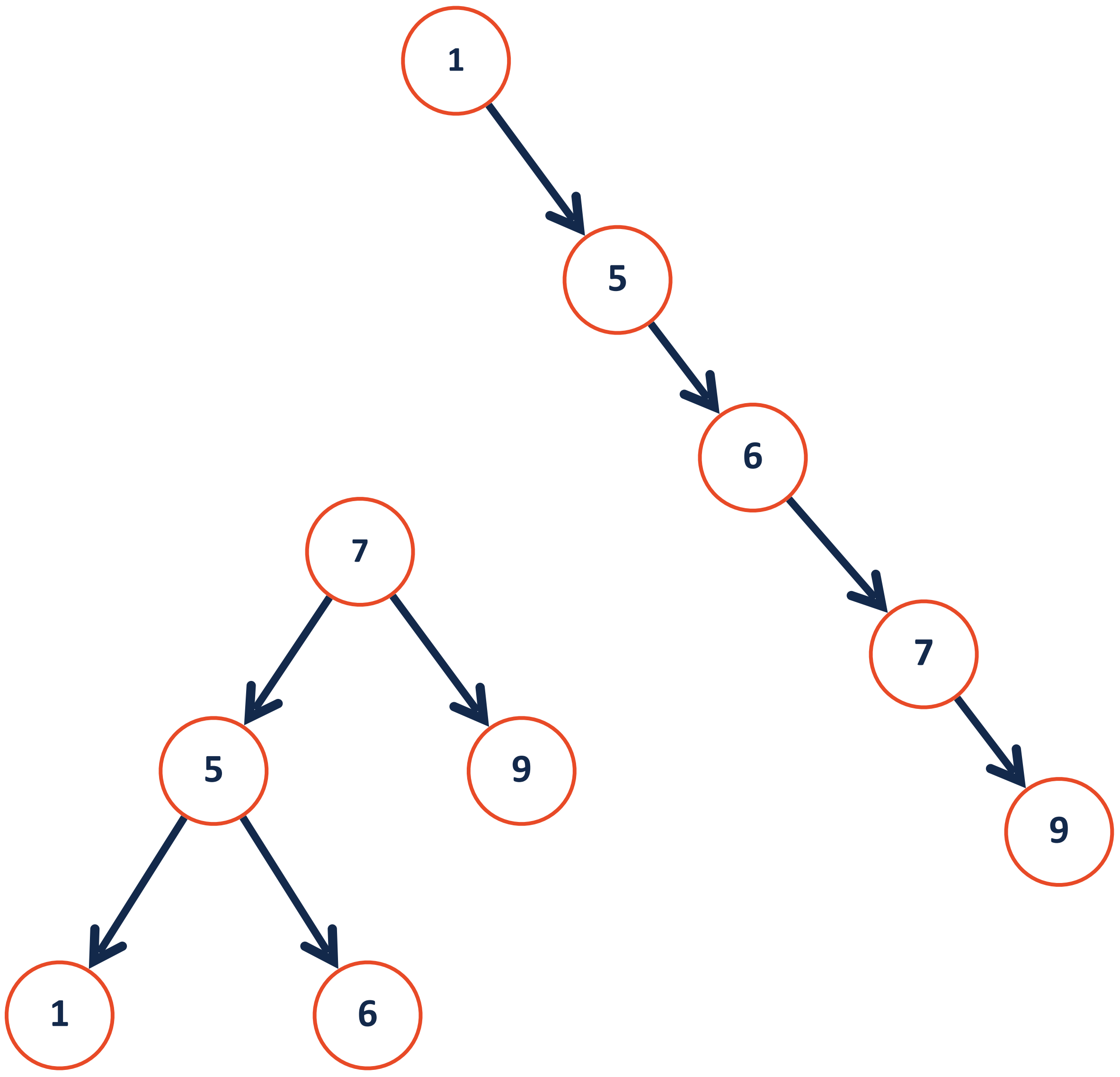
remove(3)

```
def remove_helper(node, key):
    if node == None:
        return None

    if node.key > key:
        node.left = remove_helper(node.left, key)
    if node.key < key:
        node.right = remove_helper(node.right, key)
    if node.key == key:
        if node.left == None and node.right == None:
            return None
        elif node.left == None:
            return node.right
        elif node.right == None:
            return node.left

        iop = findIOP(node)
        node.key = iop.key
        node.val = iop.val
        node.left = remove_helper(node.left, iop.key)

    return node

```



BST Analysis – Running Time

	BST Worst Case
find	O(h)
insert	O(h)
delete	O(h)
traverse	O(n)



When would we use a tree?

Pretend for a moment that we always have an optimal BST.

What is the running time of **find**?

What is the running time of **insert**?

What is the running time of **remove**?

Is there a data structure with a *better* running time for all of these?

Advantages of trees

The running time for a balanced tree is *always* **O(log n)**

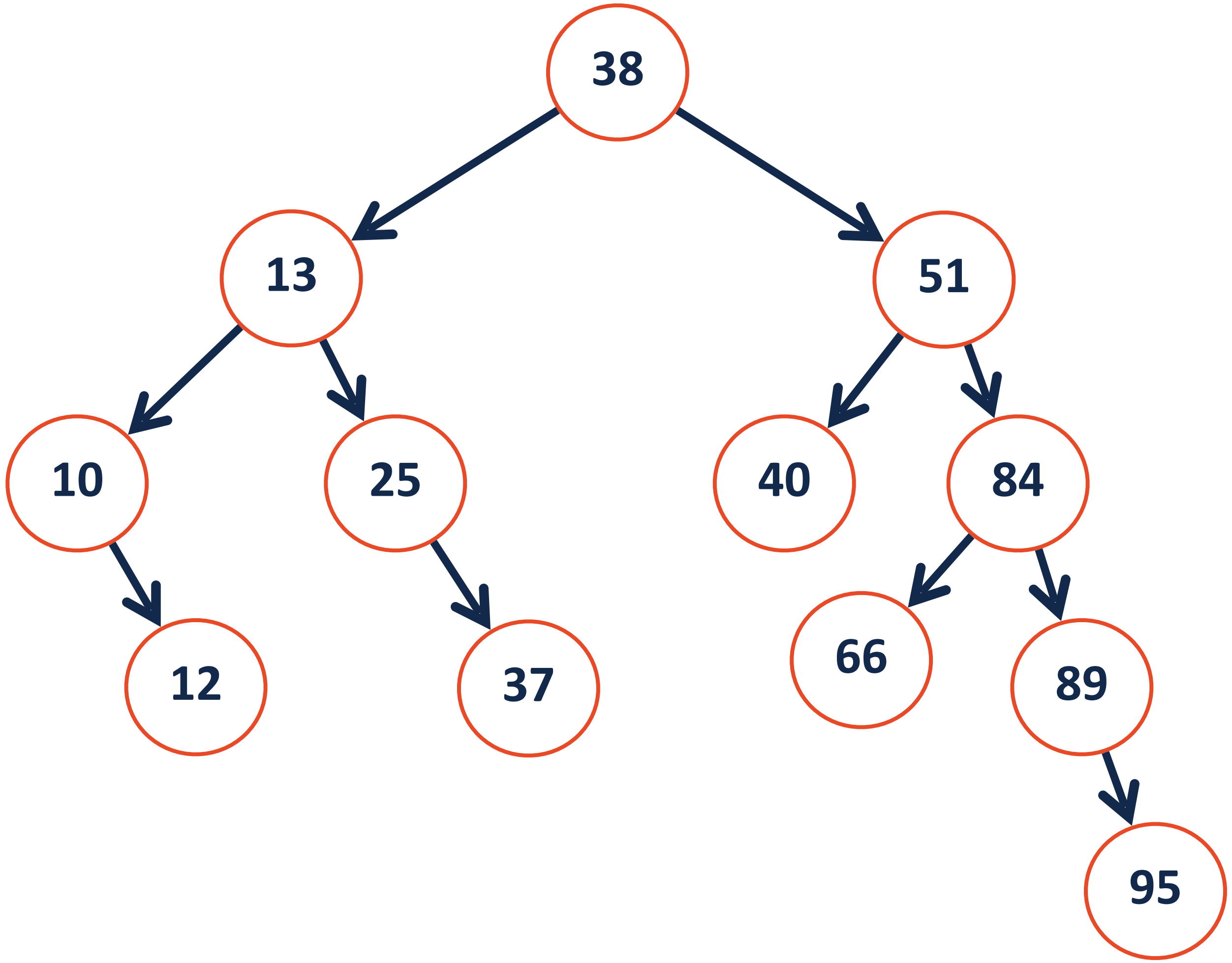
The structure of a tree can have underlying meaning

Ex: Huffman Trees for Huffman encoding

Trees can be used to **find the nearest neighbor**

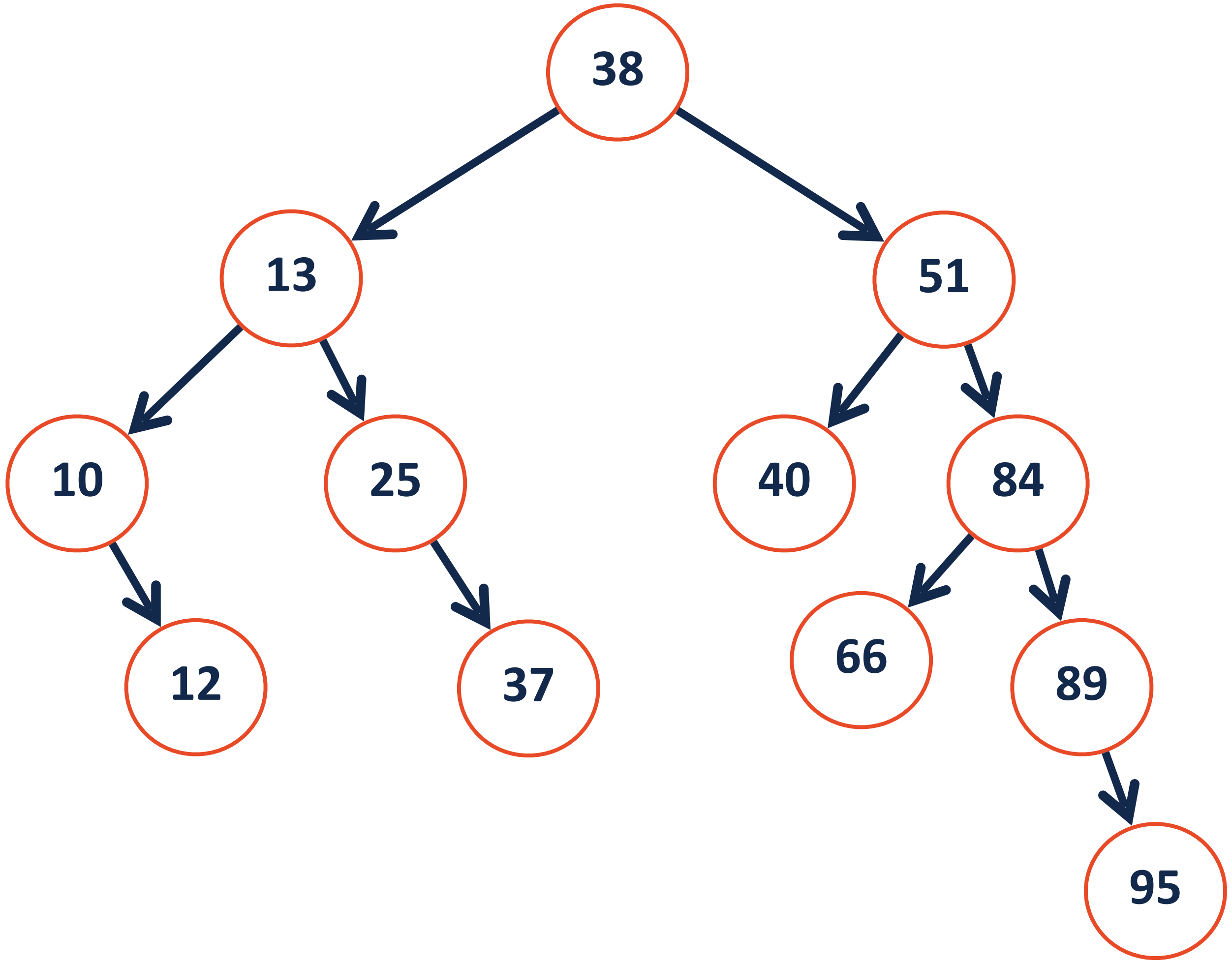
Nearest Neighbor Find

FNN(70)



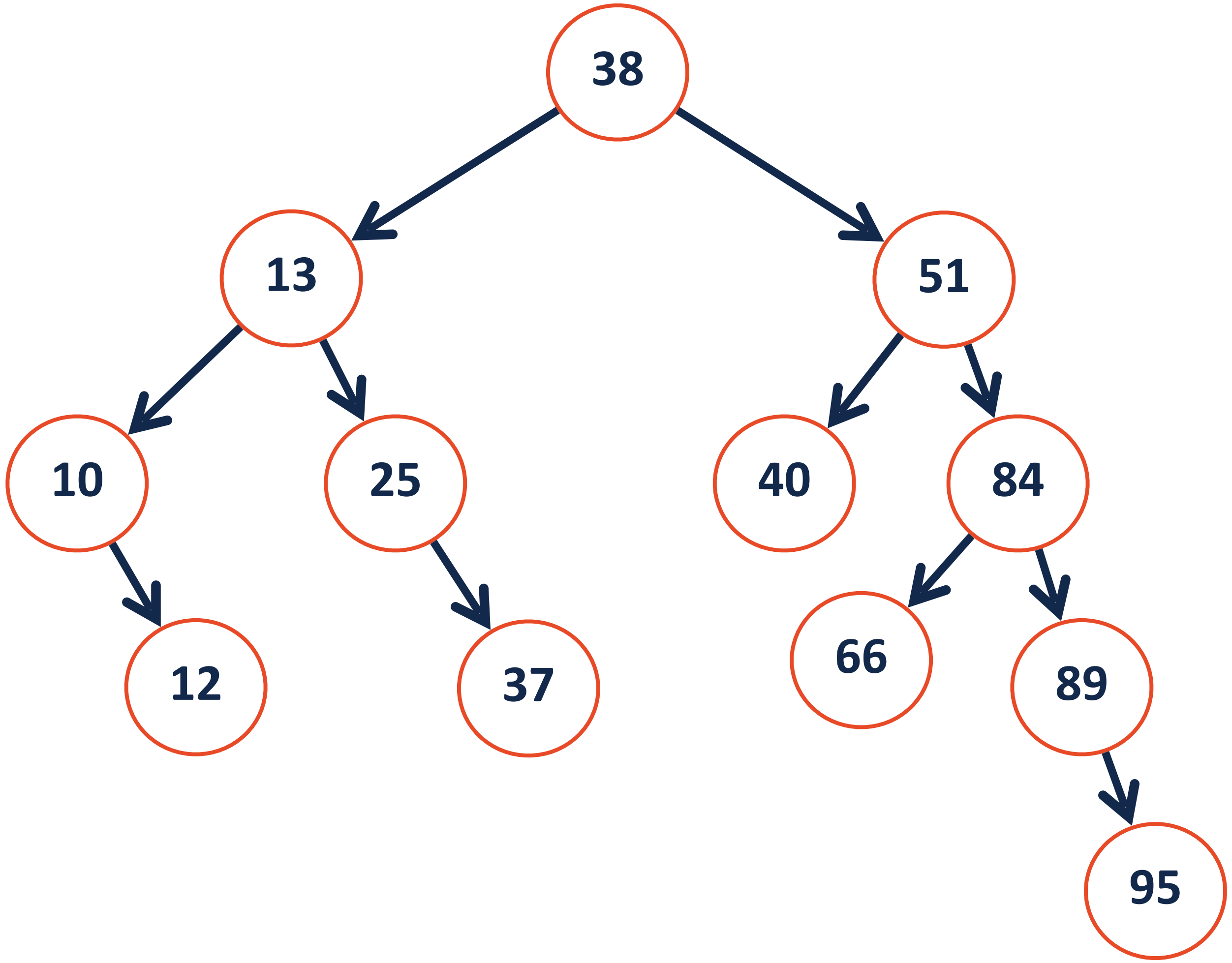
Nearest Neighbor Find

`FNN(27)`



Nearest Neighbor Find

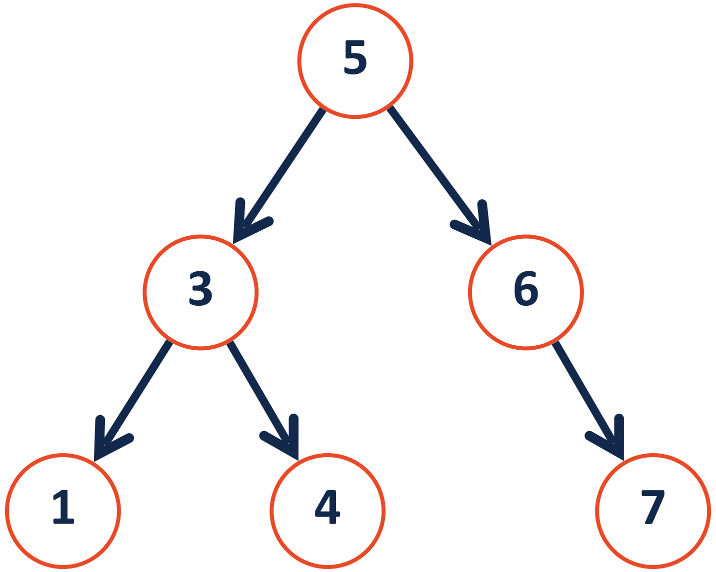
`FNN(14)`



Nearest Neighbor Find

FNN(4)

```
def fnn_helper(node,key):
    if not node:
        return None

    if node.key == key:
        return node

    if node.key > key:
        temp = fnn_helper(node.left, key)

    if node.key < key:
        temp = fnn_helper(node.right, key)

    # Nearest neighbor is either node.val (curr node)
    # OR the nearest neighbor found in the subtree
```



Real World Use Case: Nearest neighbor search

Given the collection above, what is the closest match to the color below?

Real World Use Case: Nearest neighbor search

(255, 0, 0) (255, 255, 0) (0, 255, 0) (0, 255, 255) (0,0,255) (255,0,255)

Given the collection above, what is the closest match to the color below?

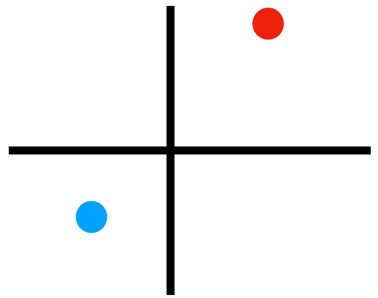
(90, 75, 50)

Euclidean Distance

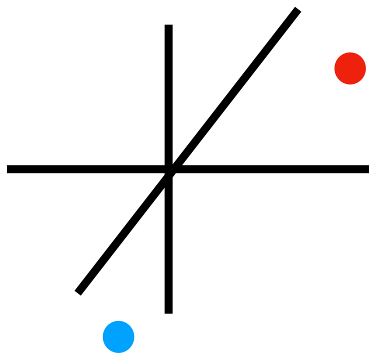
The distance between two points is the length of a line between them

1D: $d(p, q) = \sqrt{(p - q)^2}$

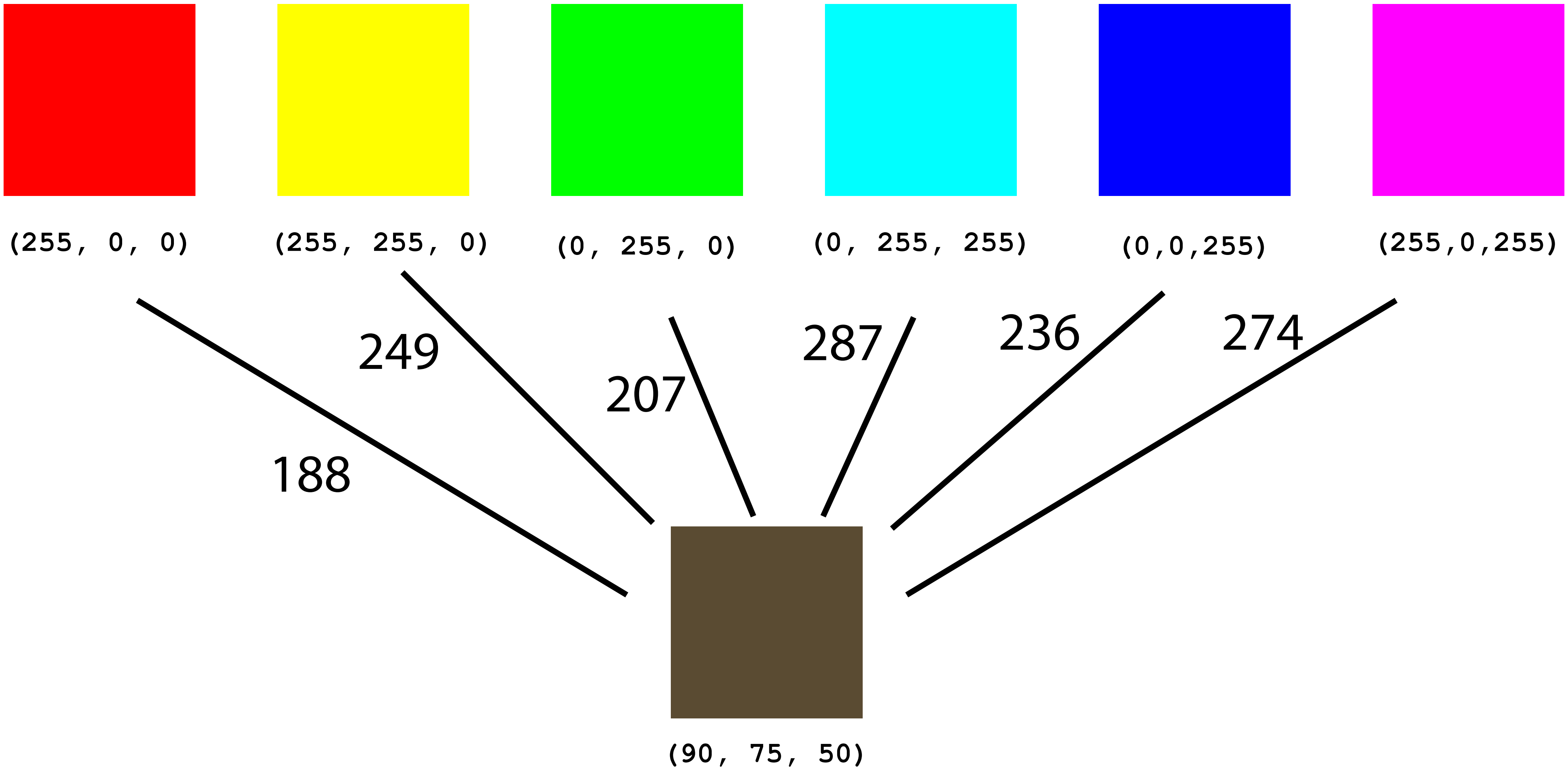
2D: $d(p, q) = \sqrt{(p_0 - q_0)^2 + (p_1 - q_1)^2}$



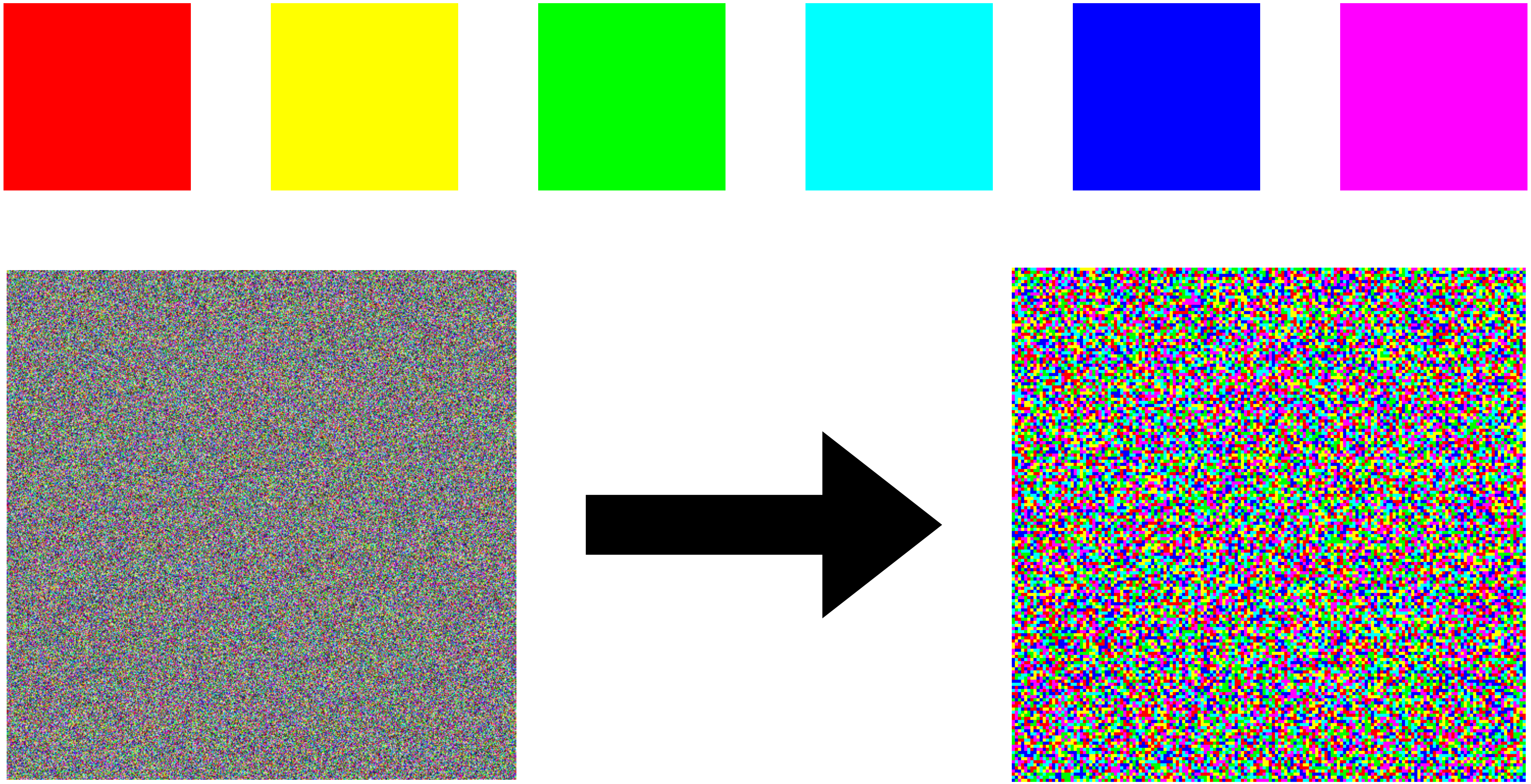
3D: $d(p, q) = \sqrt{(p_0 - q_0)^2 + (p_1 - q_1)^2 + + (p_2 - q_2)^2}$



Real World Use Case: Nearest neighbor search



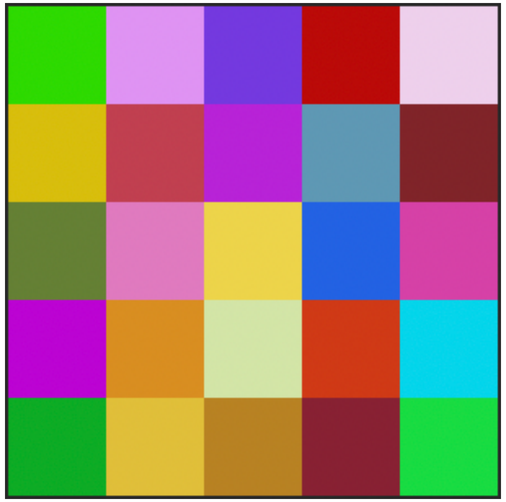
Real World Use Case: Nearest neighbor search



Real World Use Case: Nearest neighbor search

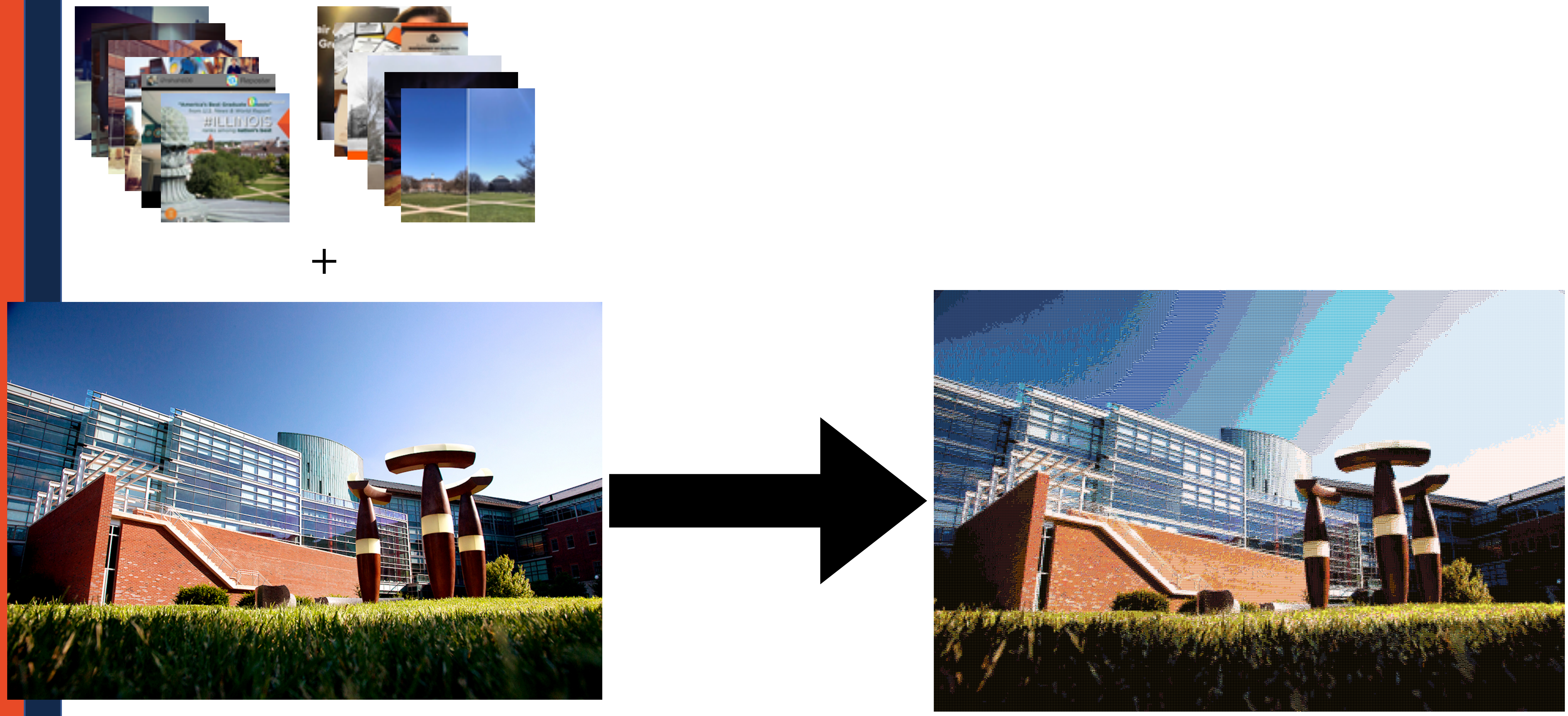
(255, 0, 0) (255, 255, 0) (0, 255, 0) (0, 255, 255) (0,0,255) (255,0,255)

We can reduce the total number of calculations by **averaging colors**

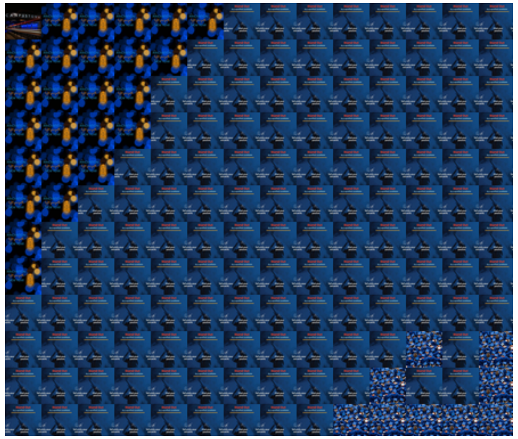


```
[[ 45 218   0],[223 147 243],[116  57 223],[187   9   9],[238 208 236]]
[[216 190  15],[193  64  80],[184  35 215],[ 95 152 180],[128  36  41]]
[[101 128  53],[224 122 191],[237 212  74],[ 35  98 227],[214  66 167]]
[[188   3 211],[217 142  33],[210 229 167],[208  57  22],[  3 213 235]]
[[ 11 172  37],[225 191  57],[184 130  34],[136  33  51],[ 26 220  67]]
```

Real World Use Case: Nearest neighbor search



Real World Use Case: Nearest neighbor search



+

Naive Nearest Neighbor Search

1. Create a method of getting the Euclidean distance between points

```
exactColorDist(c1, c2)
```

2. Create a method of getting the average color for a subset of the image

```
getAverageColor(numArray, rstart=0, cstart=0, rlen=None, clen=None)
```

3. For each sub-image of a large image, get the closest matching tile

```
getClosestColor(inlist, query)
```

Naive Nearest Neighbor Search

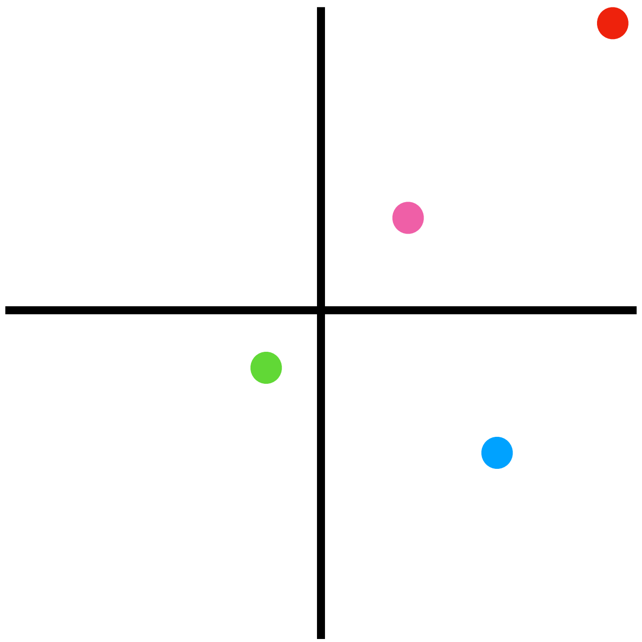
Pros:

Cons:

BST Nearest Neighbor Search

Rather than compare every sub-image to every tile, we want to build a BST!

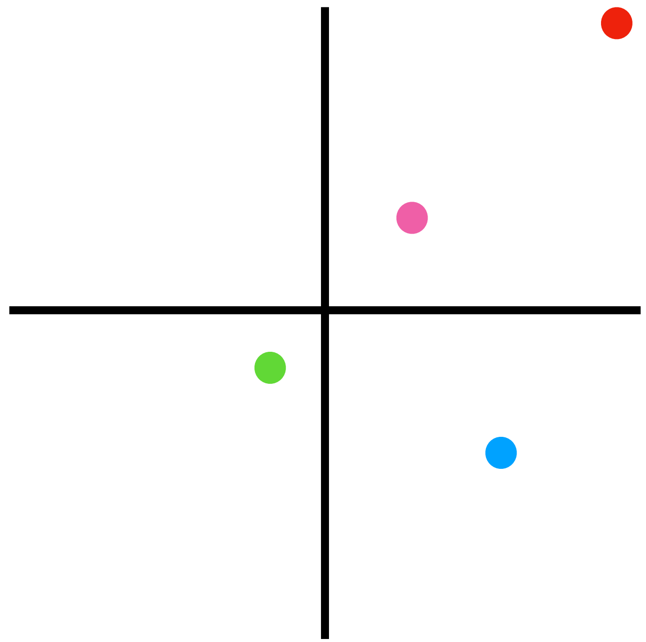
… what is the smallest point in a 2D plane?



BST Nearest Neighbor Search

Rather than compare every sub-image to every tile, we want to build a BST!

… what is the smallest point in a 2D plane?



Problem: There's no obvious order in multi-dimensional space!

We can *reduce the dimensions* to create an arbitrary order, but lose precision

BST Nearest Neighbor Search

Because our input set is ***colors***, there is at least one dimensional reduction

BST Nearest Neighbor Search

Because our input set is ***colors***, there is at least one dimensional reduction

Instead of a 3D RGB value, we can store a 1D ***luminance*** value:

$$L == 0.299R + 0.587G + 0.114B$$

BST Nearest Neighbor Search

1. Create a method of comparing 1D ‘sizes’ of 3D objects

```
getLum(c1)
```

2. Build a luminance BST that stores 3D objects based on their 1D size

```
lum_tree_insert(root, key, value)
```

3. Implement a nearest neighbor search on the luminance BST

```
lum_tree_find(root, key)
```

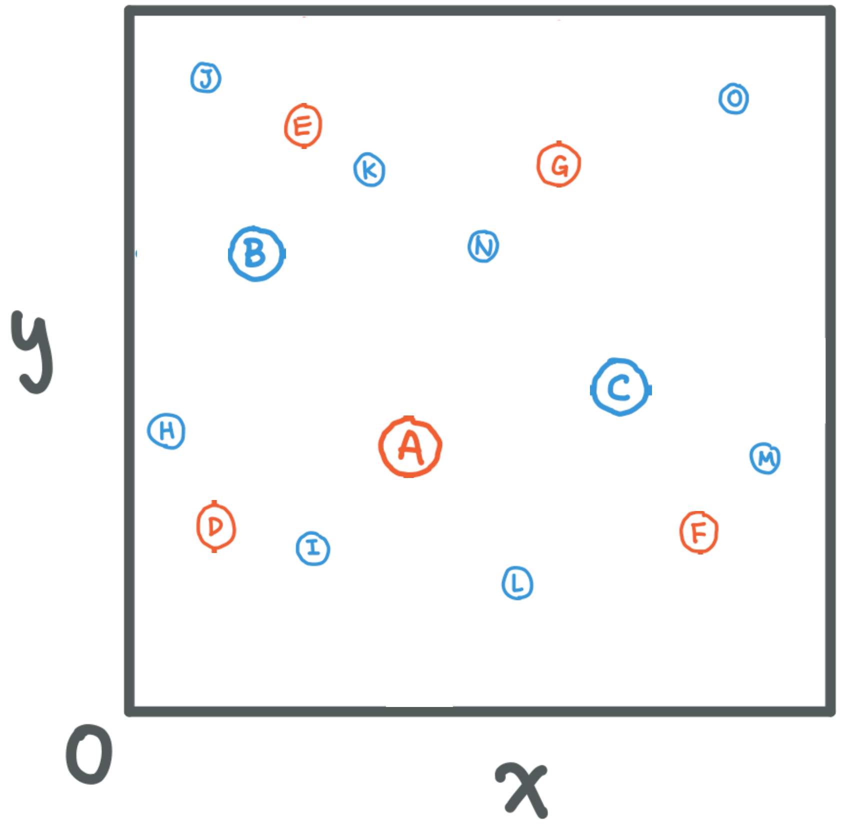
Luminance Nearest Neighbor Search

Pros:

Cons:

The k-dimension tree (KD-tree)

Imagine we have a set of two dimensional points…

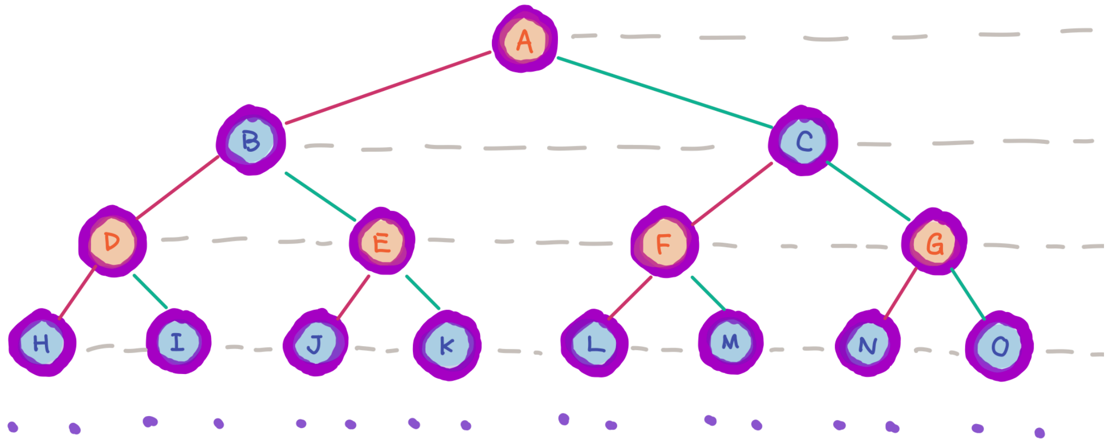


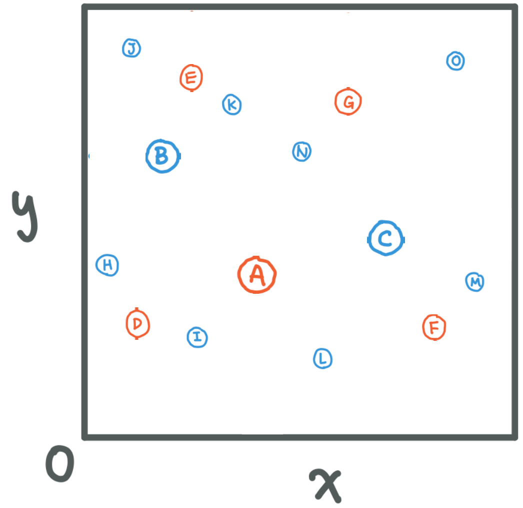
Images made by Jenny Chen (former CS 225 CA)

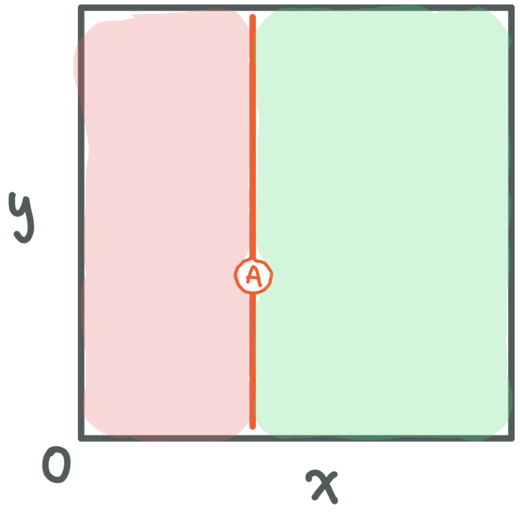
The k-dimension tree (KD-tree)

We can build a k-dimension BST by comparing one dimension at a time

Depth: Split Dimension:







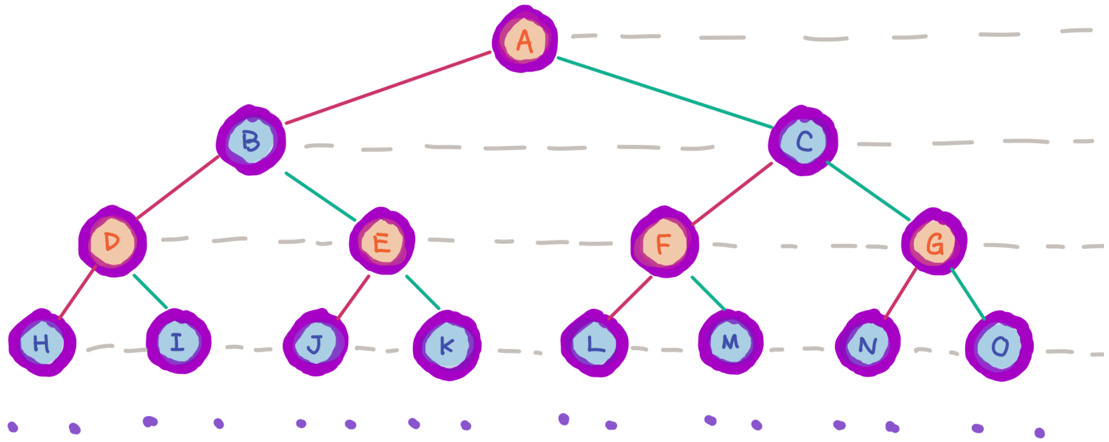
Images made by Jenny Chen (former CS 225 CA)

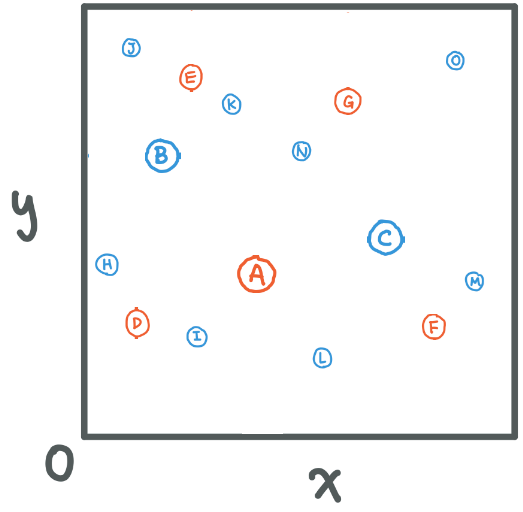
The k-dimension tree (KD-tree)

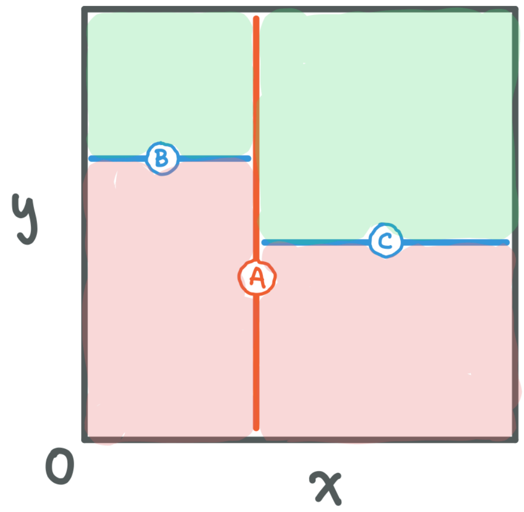
We can build a k-dimension BST by comparing one dimension at a time

Depth:

Split Dimension:







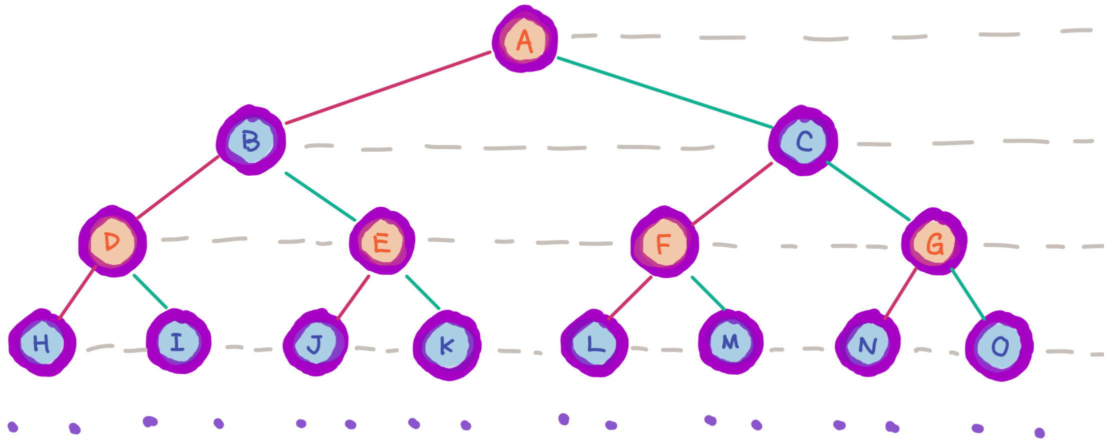
Images made by Jenny Chen (former CS 225 CA)

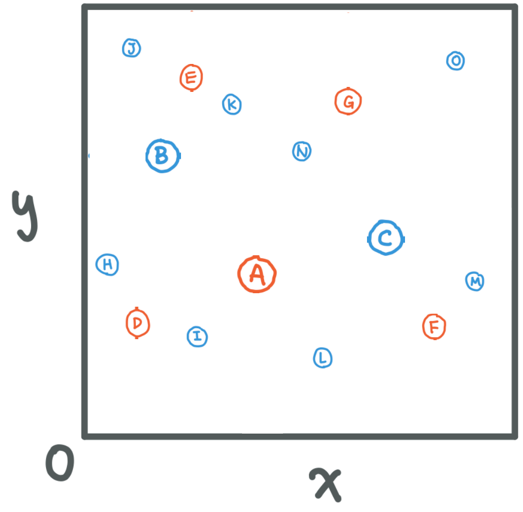
The k-dimension tree (KD-tree)

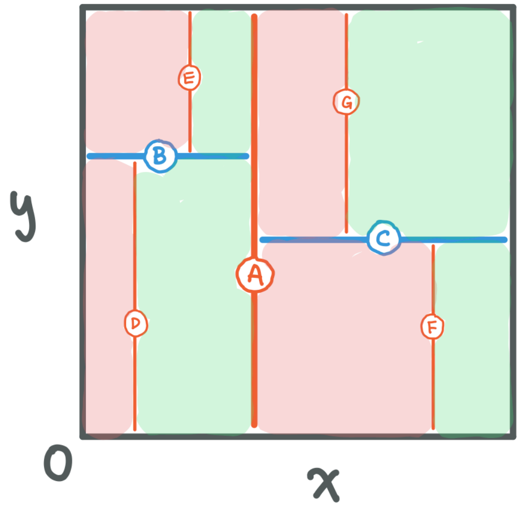
We can build a k-dimension BST by comparing one dimension at a time

Depth:

Split Dimension:





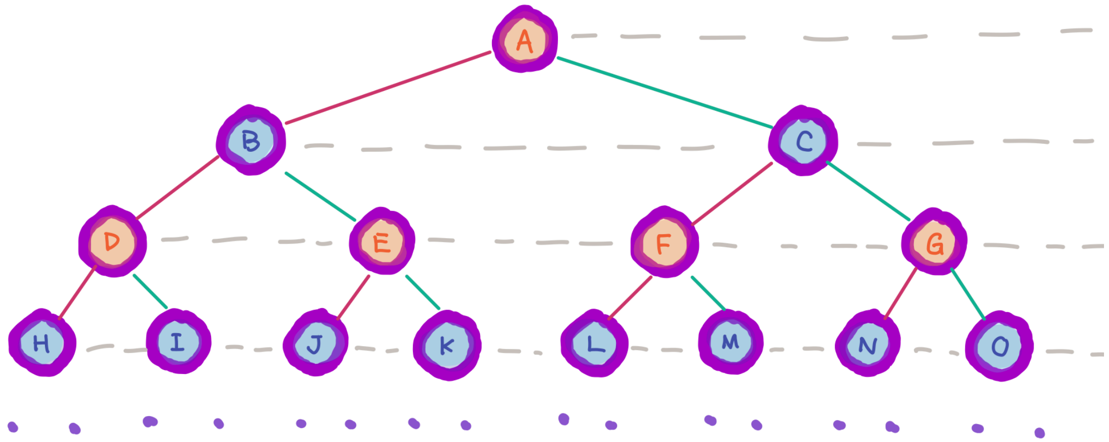


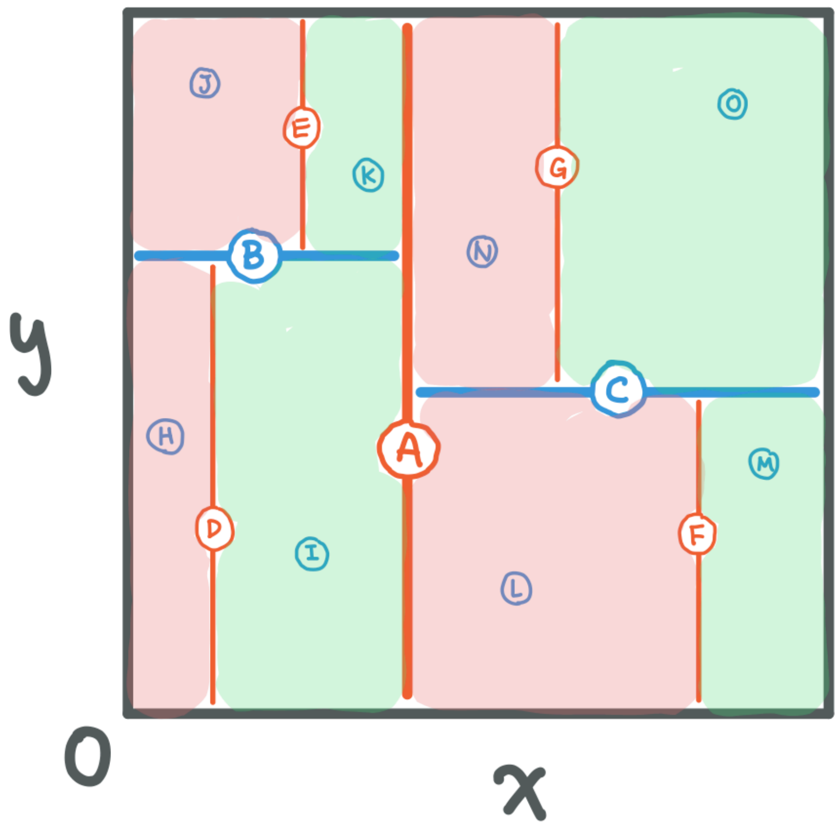
Images made by Jenny Chen (former CS 225 CA)

The k-dimension tree (KD-tree)

We can build a k-dimension BST by comparing one dimension at a time

Depth: Split Dimension:

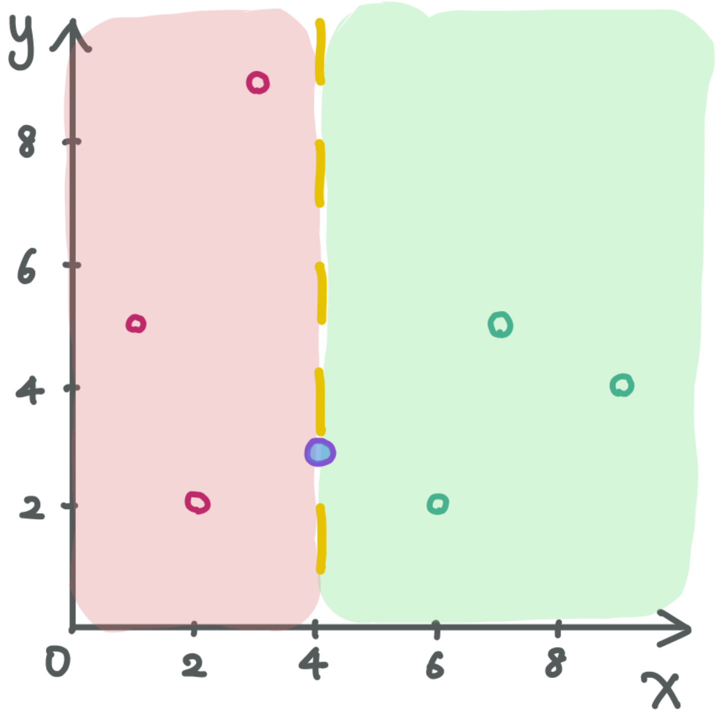


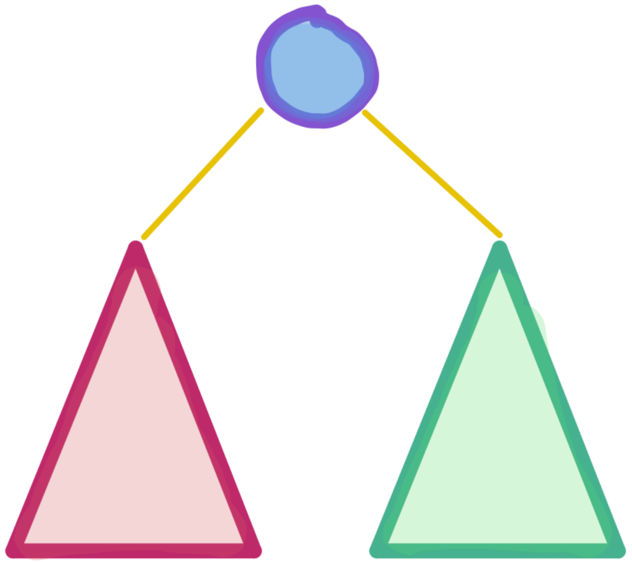


Images made by Jenny Chen (former CS 225 CA)

The k-dimension tree (KD-tree)

At every level, we essentially partition our tree in half:





KD-Tree Construction Example

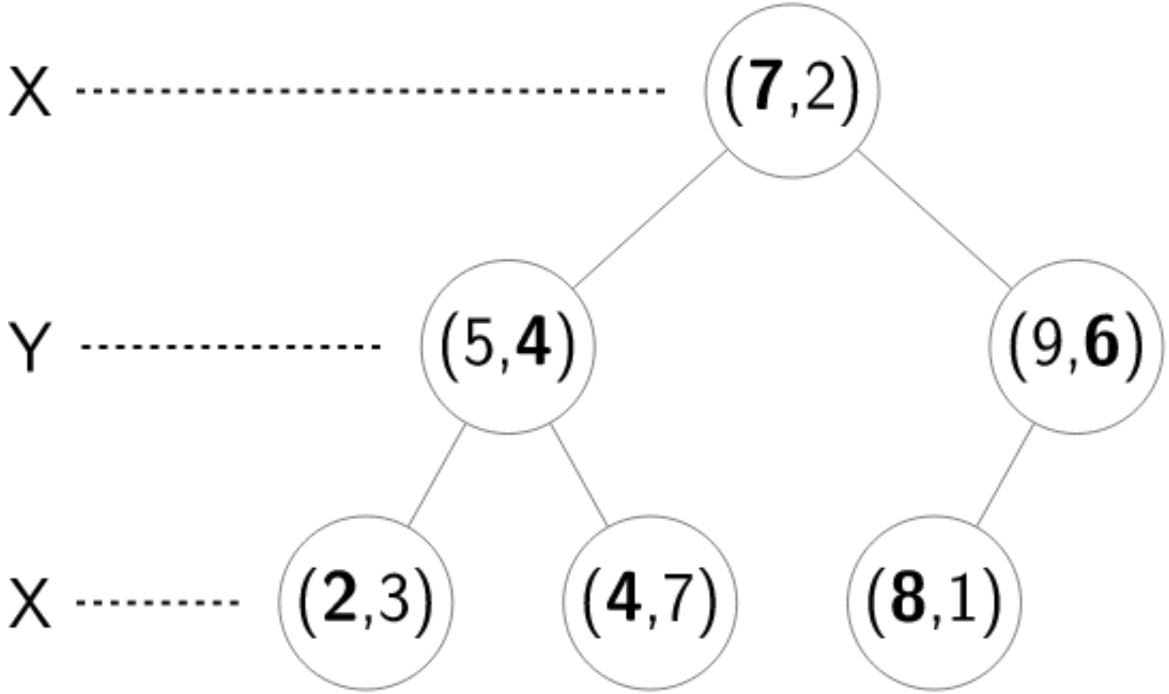
Imagine I wanted to build a KD-tree with the following points in order:

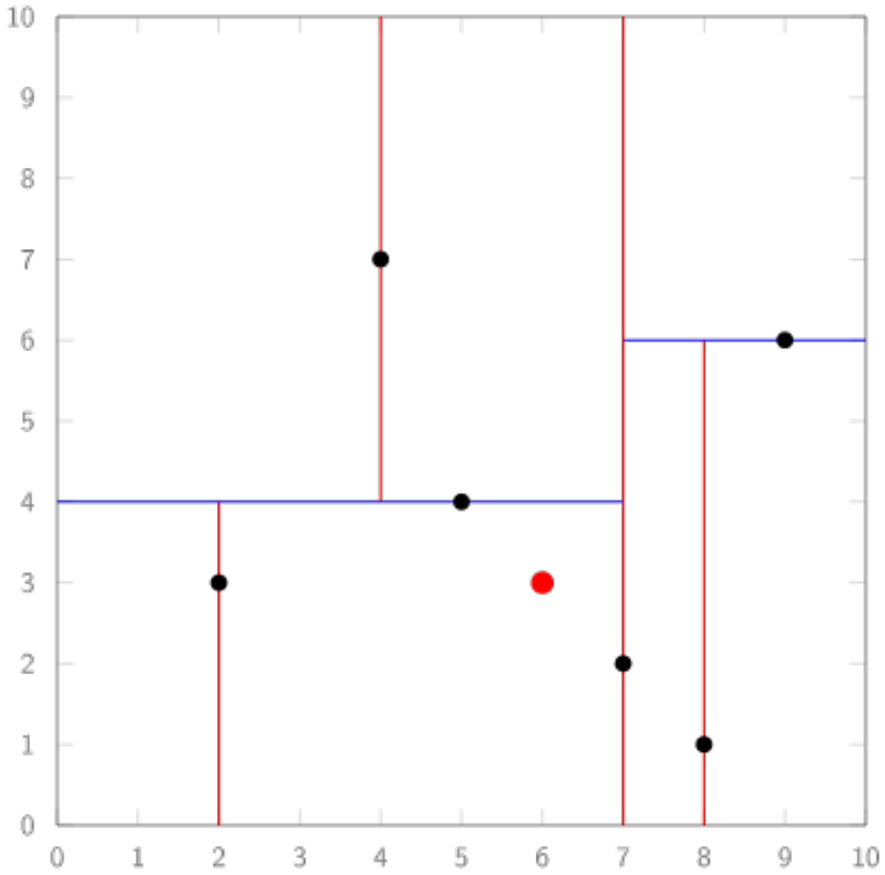
(7, 2), (5, 4), (9, 6), (2, 3), (8, 1), (4, 7)

KD-tree Search

`search((6, 3))`

At each level we want to compare *only* the relevant search dimension

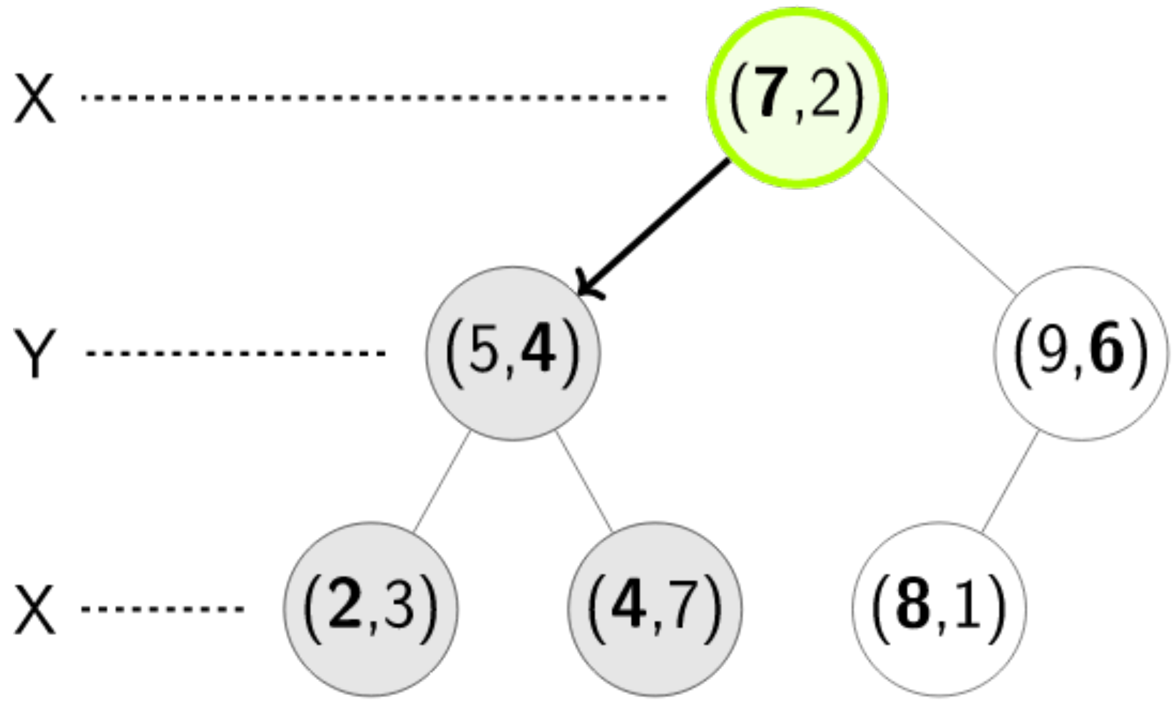


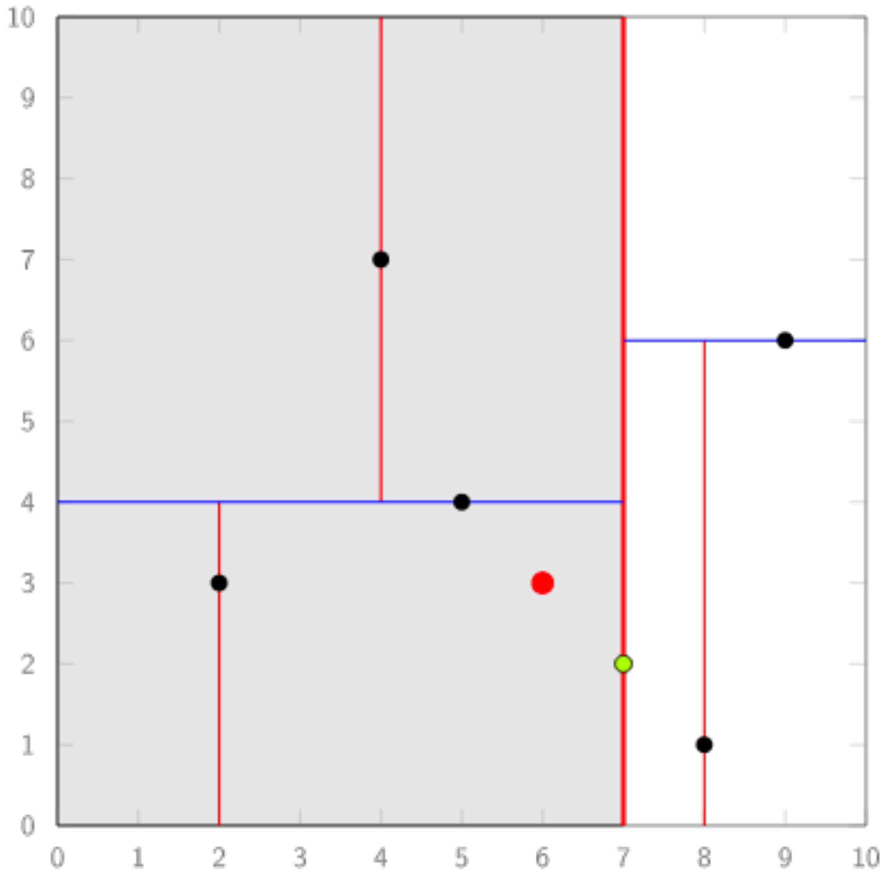


KD-tree Search

`search((6, 3))`

At each level we want to compare *only* the relevant search dimension

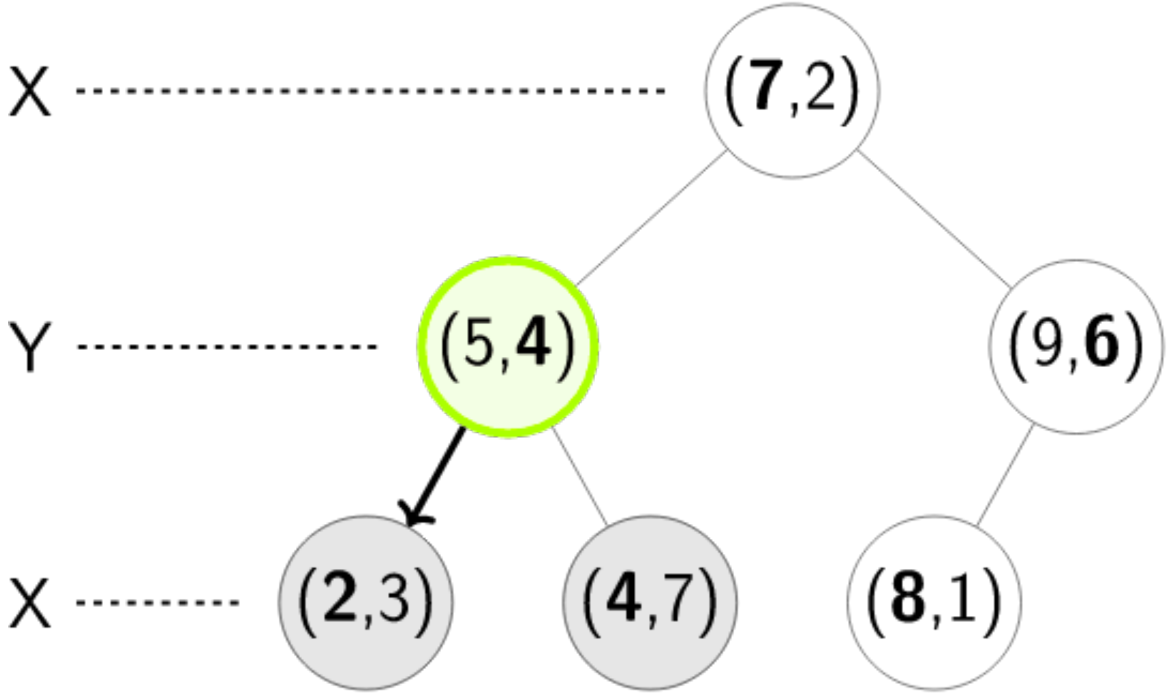


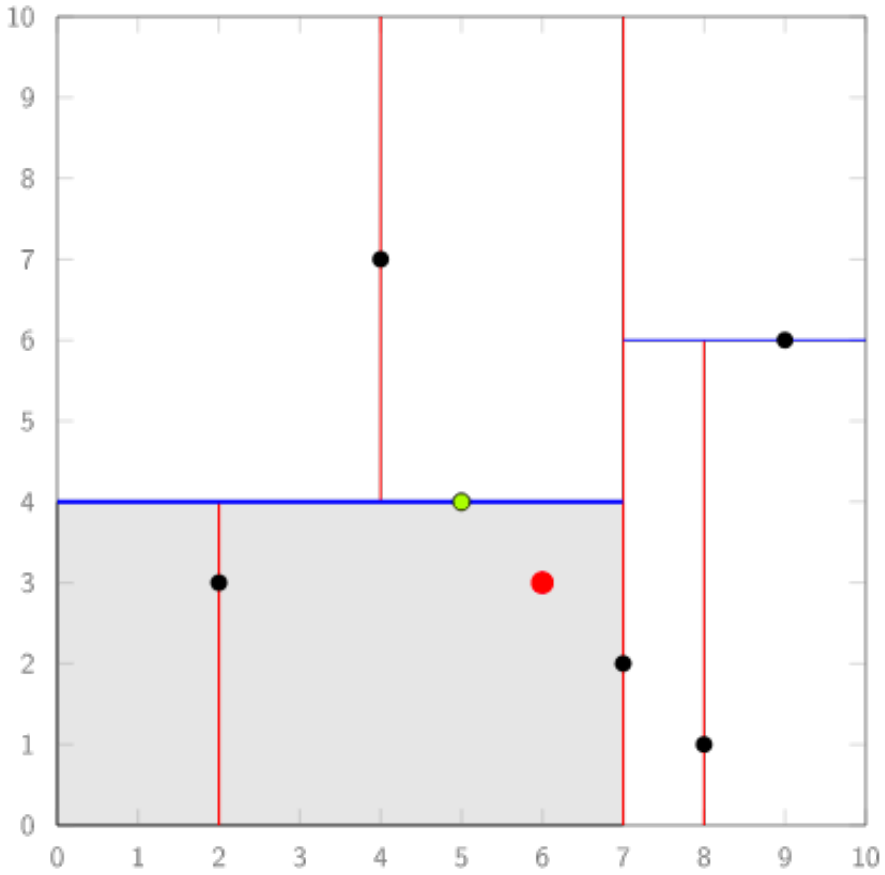


KD-tree Search

`search((6, 3))`

At each level we want to compare *only* the relevant search dimension

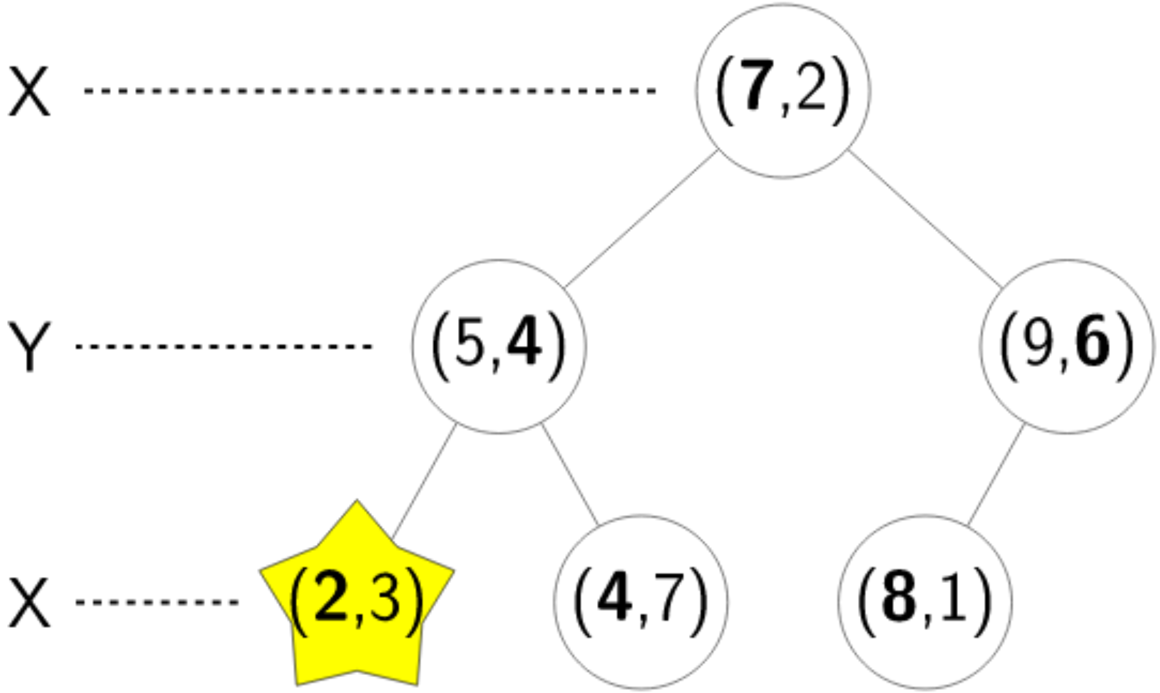


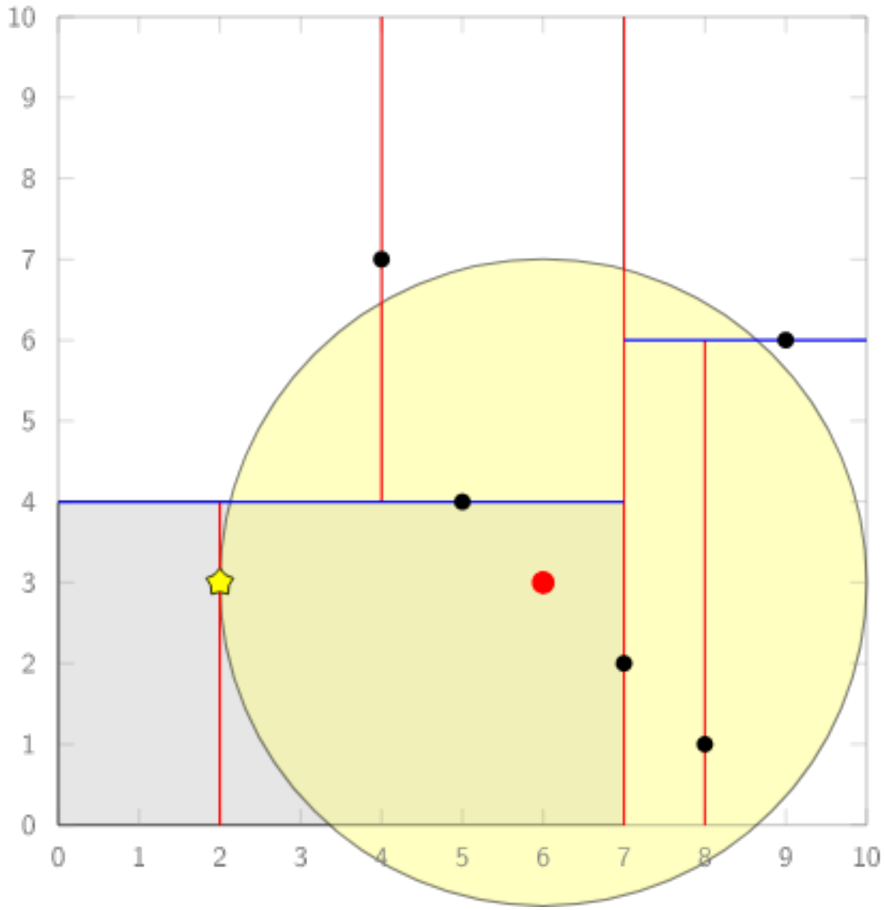


KD-tree Search

`search((6, 3))`

The leaf we find only tells us the ***maximum radius*** we need to search

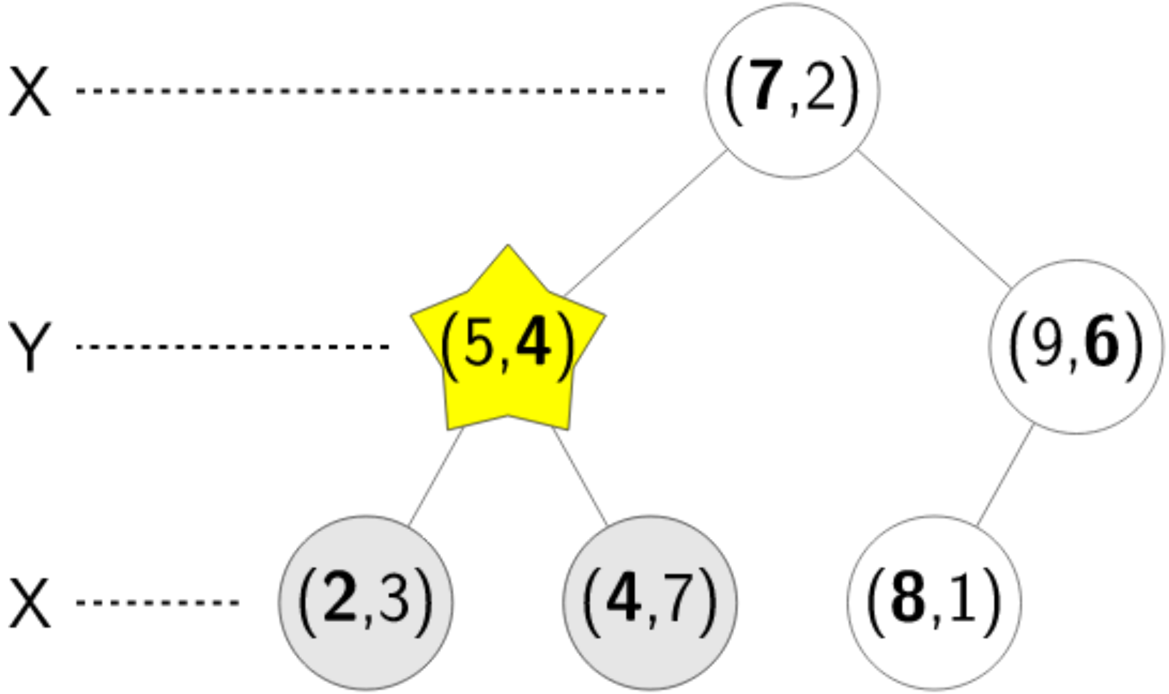


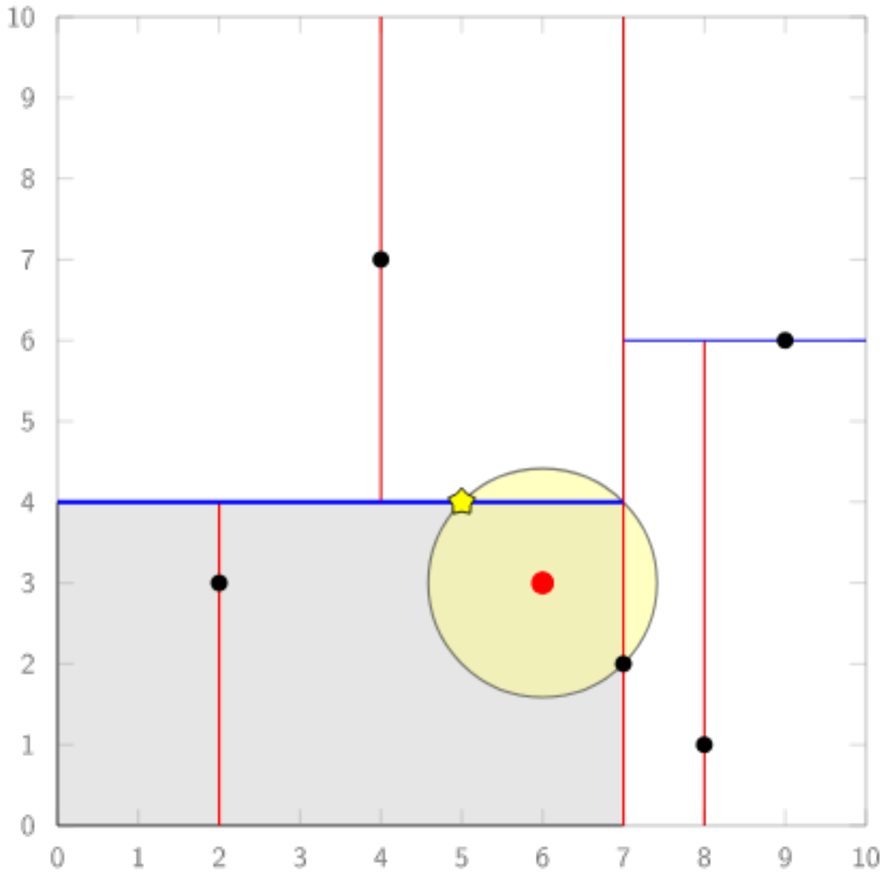


KD-tree Search

`search((6, 3))`

As we go back up the tree, we decide if our point is closer to our query

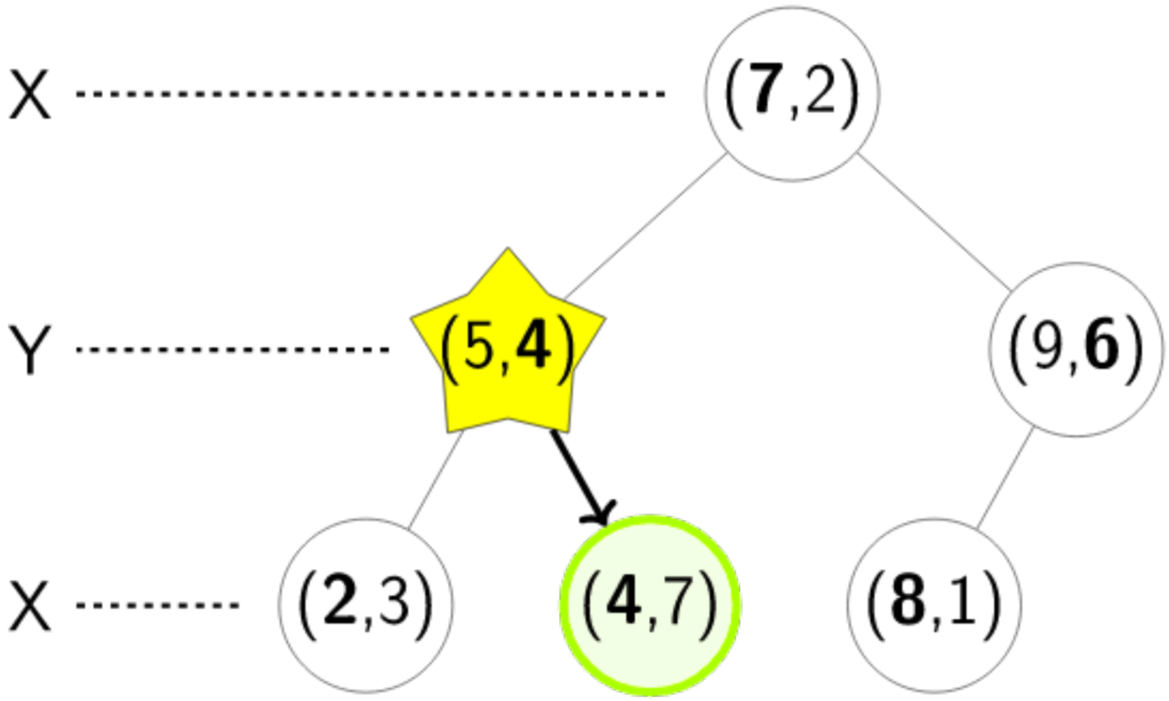


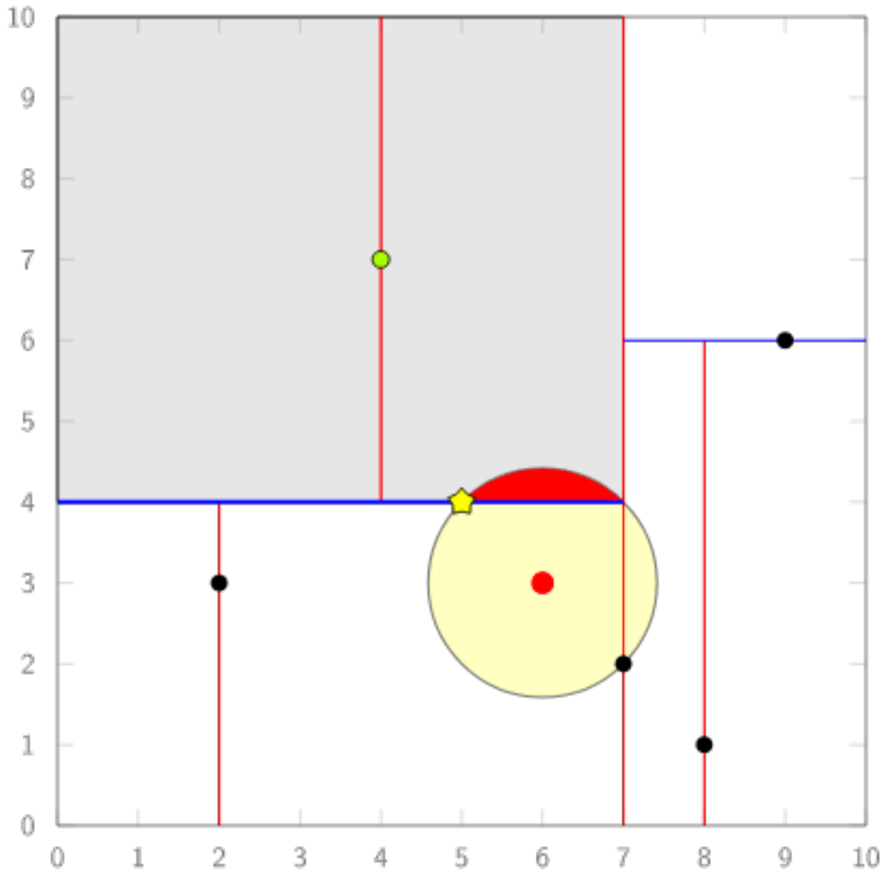


KD-tree Search

`search((6, 3))`

Since our splitting point was within our search radius, we also have to check if there's a closer point in the other subtree

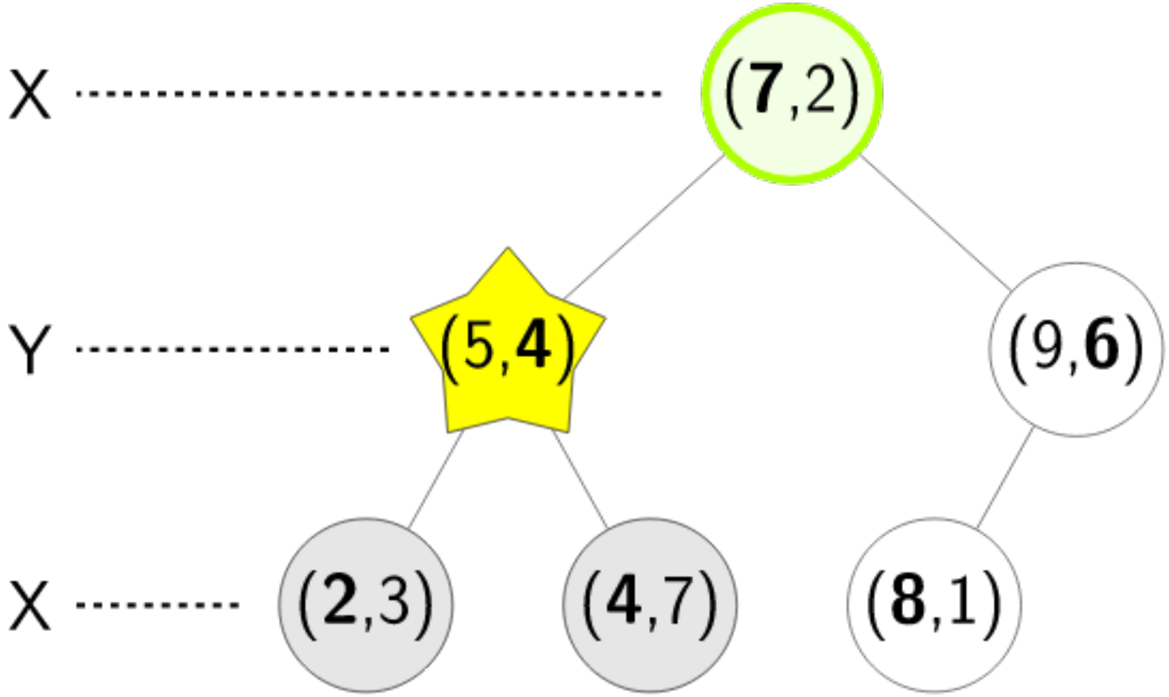


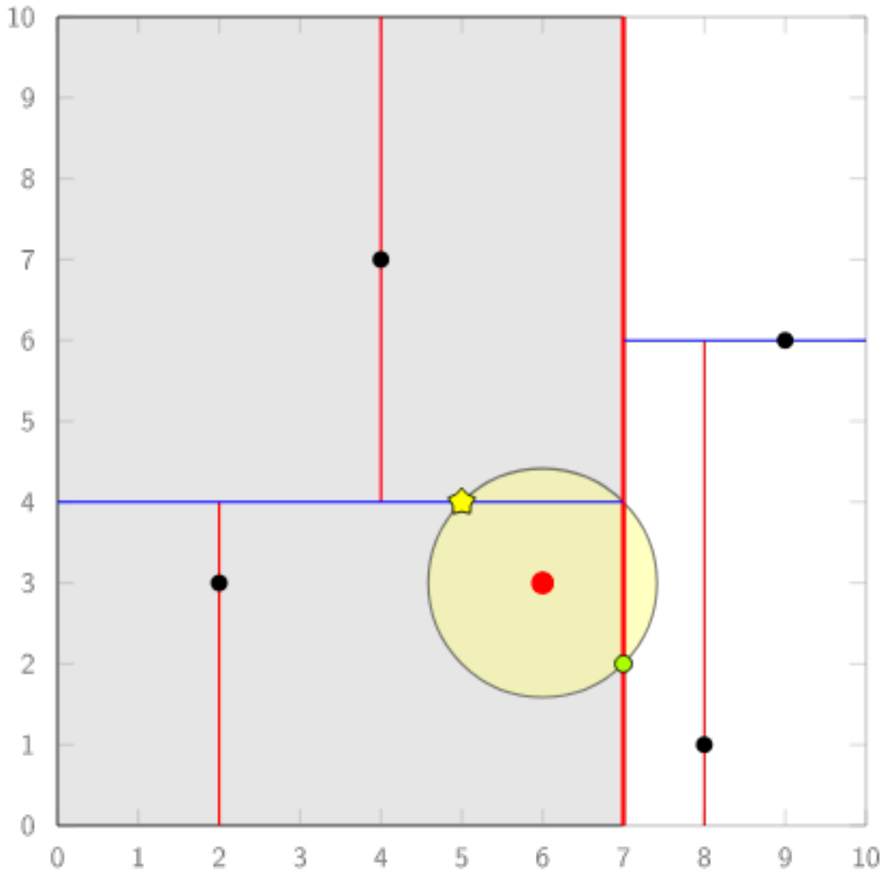


KD-tree Search

`search((6, 3))`

We repeat this process all the way up to the root…

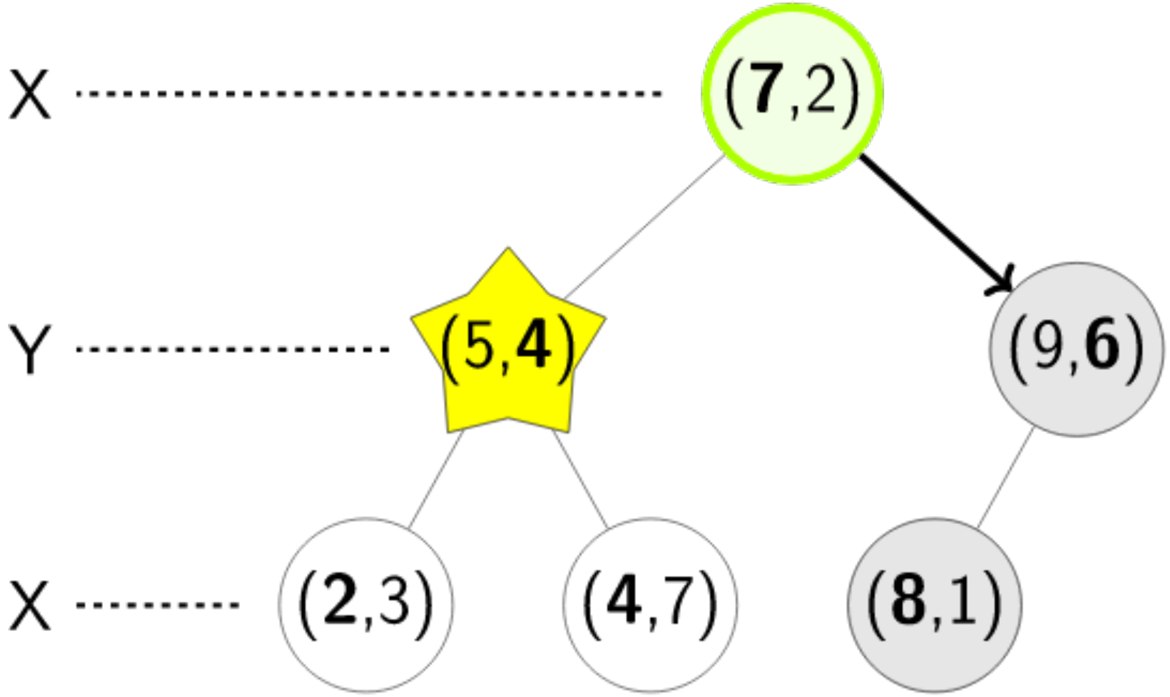


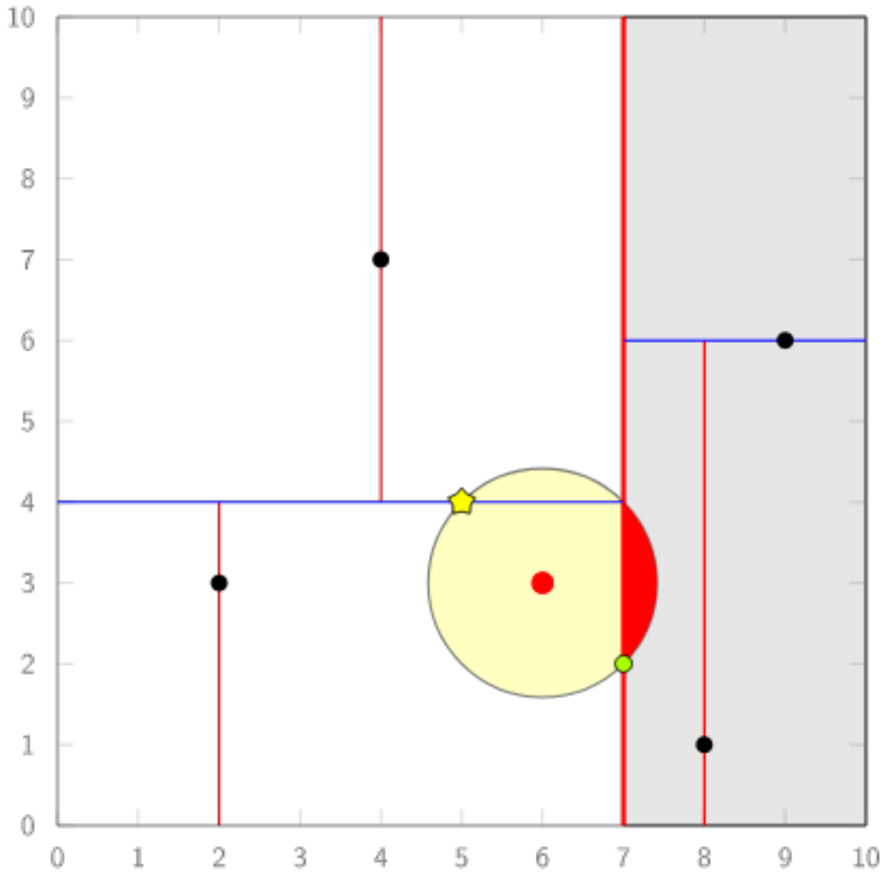


KD-tree Search

`search((6, 3))`

… and in the other subtree if the root was in our radius

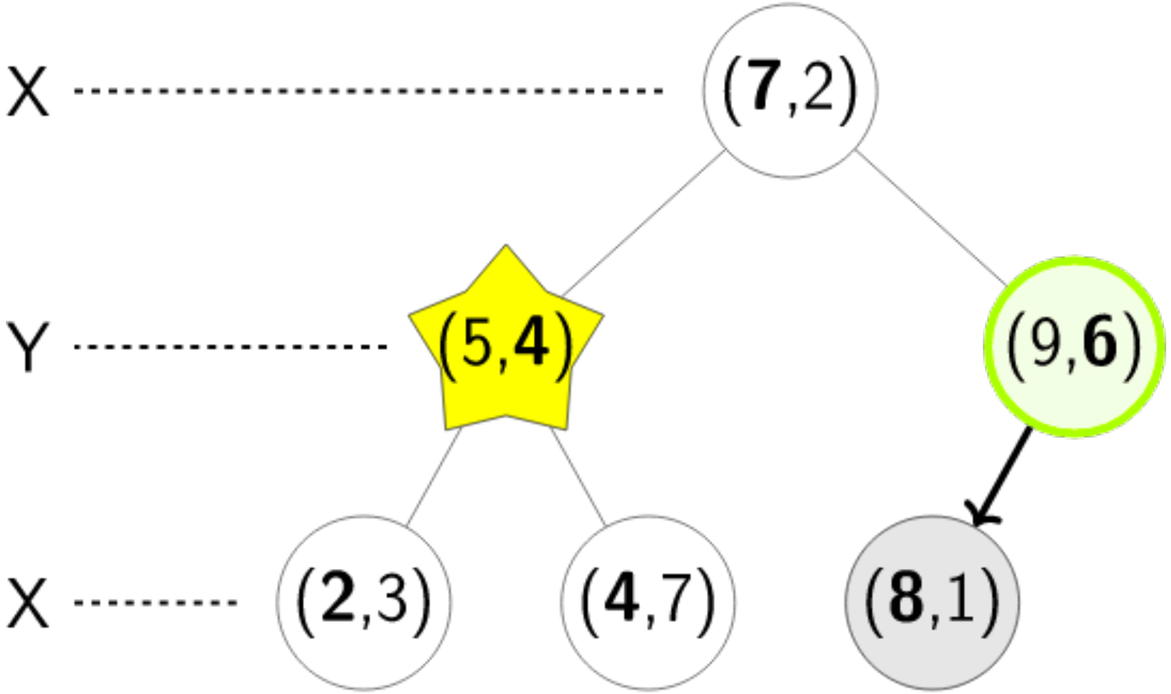


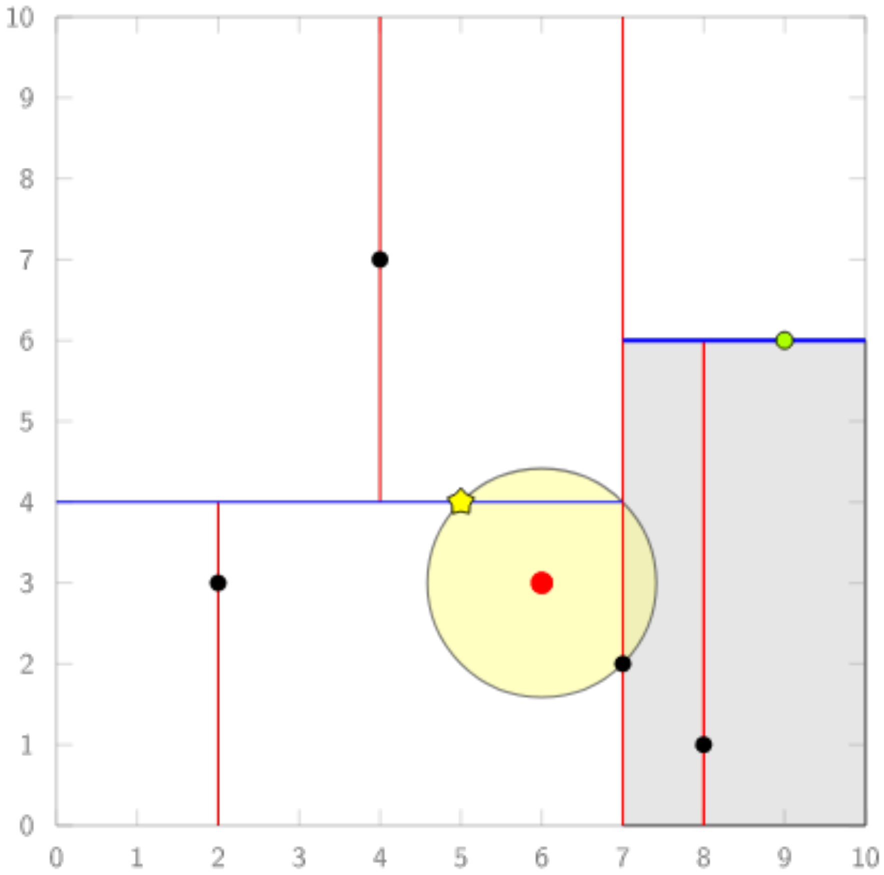


KD-tree Search

`search((6, 3))`

… and in the other subtree if the root was in our radius

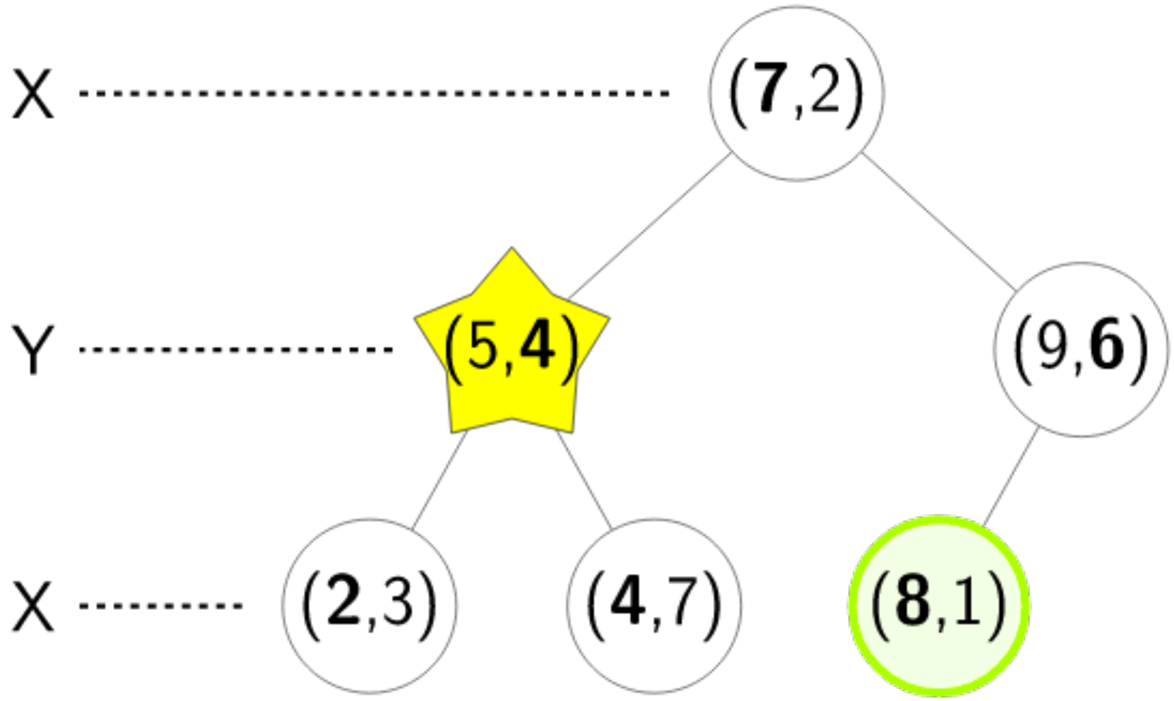


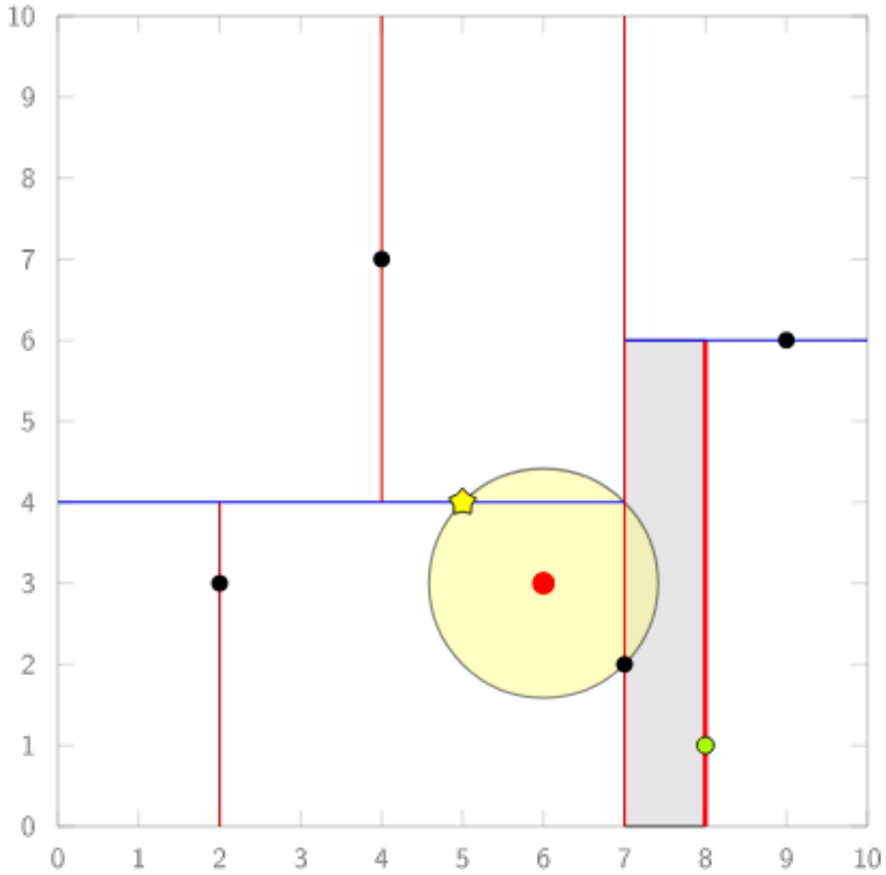


KD-tree Search

`search((6, 3))`

… and in the other subtree if the root was in our radius

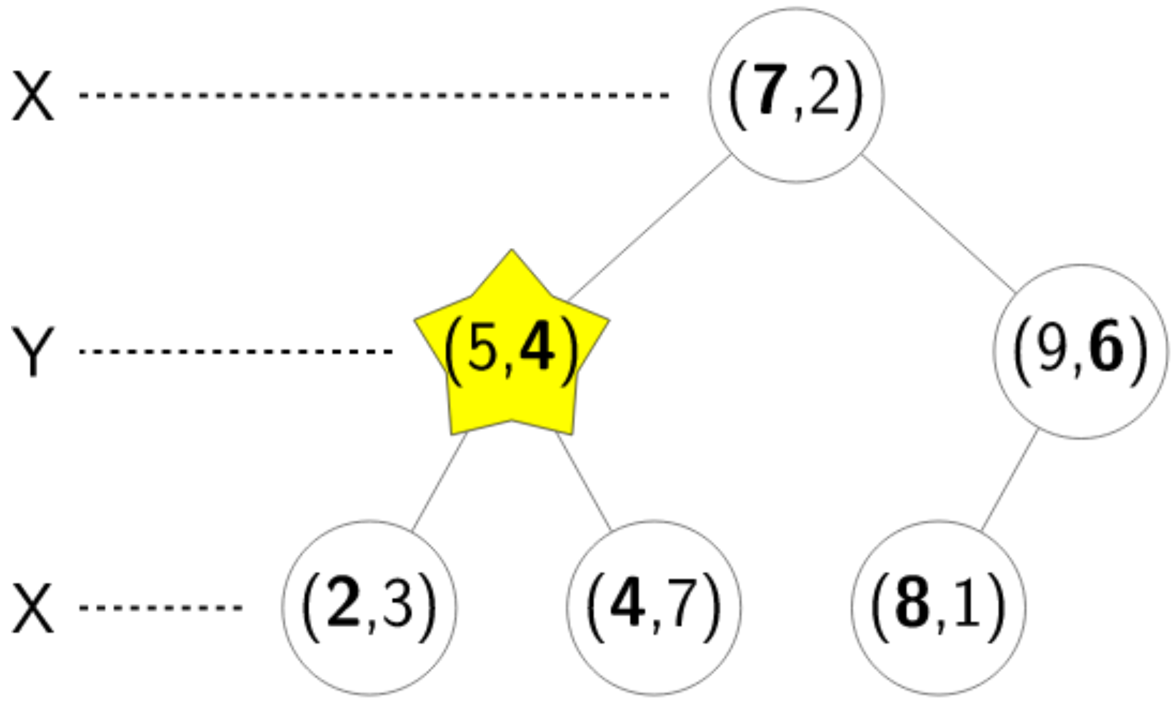


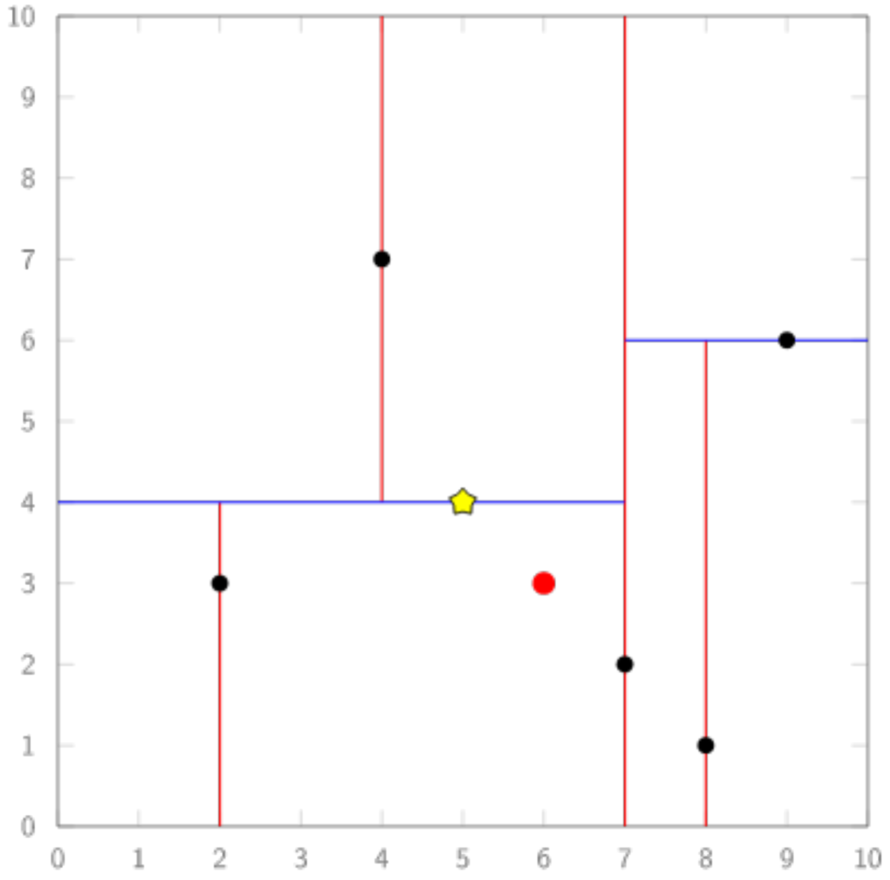


KD-tree Search

`search((6, 3))`

This strategy is worst case O(n) but on average O(log n).





KD-Tree Nearest Neighbor Search

Pros:

Cons:

KD-Tree in CS 277

You do not need to know how to implement a KD-tree

```
from scipy.spatial import KDTree

l = [(255, 0, 0), (255, 255, 0), (0, 255, 0), (0,
255, 255), (0,0,255), (255,0,255)]

kdt = KDTree(l)
```

You should know that it is the optimal solution to 100% accuracy NNS

You should understand conceptually how a KD-tree is built

Friday: A different tree application

Next Week: Addressing the ‘height’ problem

	BST Worst Case
find	O(h)
insert	O(h)
delete	O(h)
traverse	O(n)