Data Structures

KD Tree (Nearest Neighbor) 2

CS 225

October 4, 2024

Brad Solomon

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

~/illinois wcs PRESENTS

CODE ADA 2024: CODE TO CURE

A hackathon for female and non-binary participants

Are you passionate about leveraging computer science to create meaningful impact? Harness your creativity and skills to develop solutions for the modern healthcare industry!

No experience needed!

OCTOBER 19TH–20TH

go.illinoiswcs.org/code-ada-24

karenyg2@illinois.edu && an77@illinois.edu

PARTICIPANT SIGN UP

go.illinoiswcs.org/code-ada-24

PROJECT MANAGER SIGN UP

go.illinoiswcs.org/code-ada-pm-24

A brief reminder of academic integrity

Learning Objectives

Review KD Tree Construction

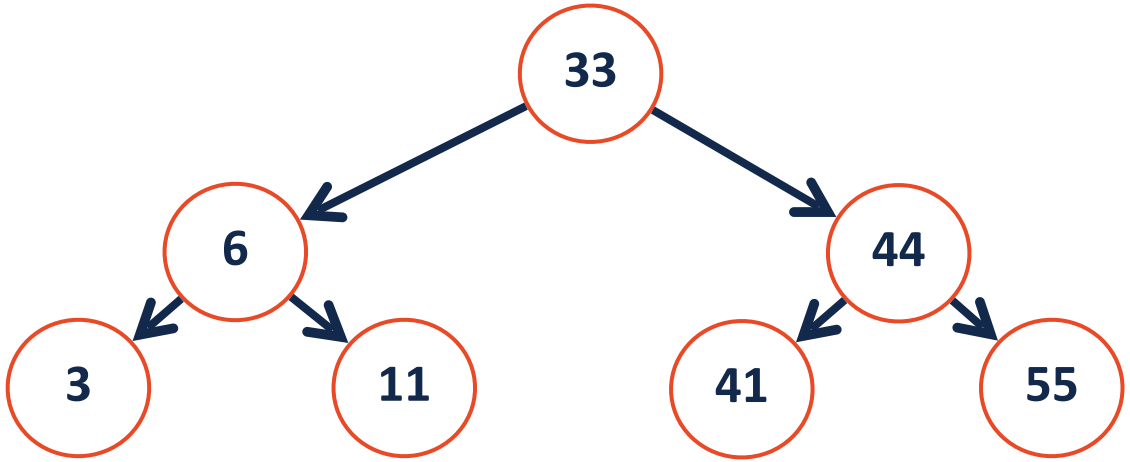
Explore KD Tree Search

Go over C++ concepts for mp_mosaics

Range-based Searches

Consider a collection of points on a 1D line: $\mathbf{p = \{p_1, p_2, \ldots, p_n\}}$

If I want to find all values between [A, B], how could I implement this?



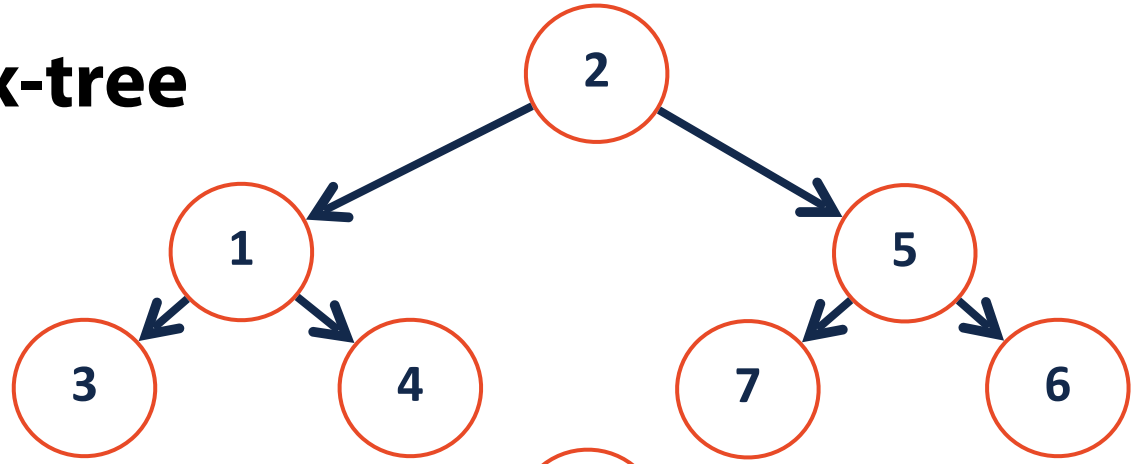

```
1
2 for(auto it = myMap.lower_bound(A); it != myMap.upper_bound(B); ++it){
3
4 // Do Stuff
5 }
```

Range-based Searches

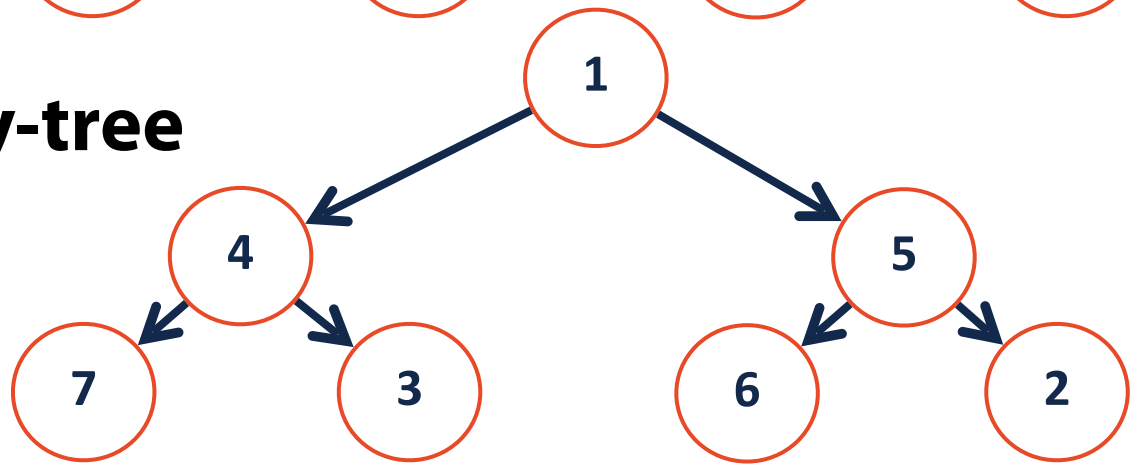
Consider points in 2D: $\mathbf{p = \{p_1, p_2, \ldots, p_n\}}$

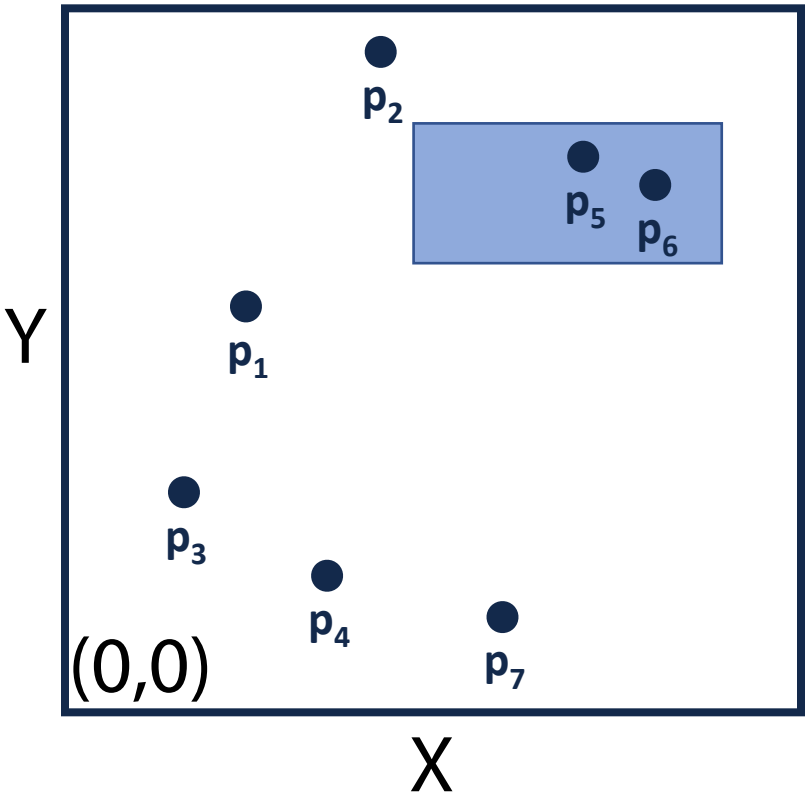
What points in rectangle [$\mathbf{(x_1, y_1)}$, $\mathbf{(x_2, y_2)}$]?

x-tree



y-tree

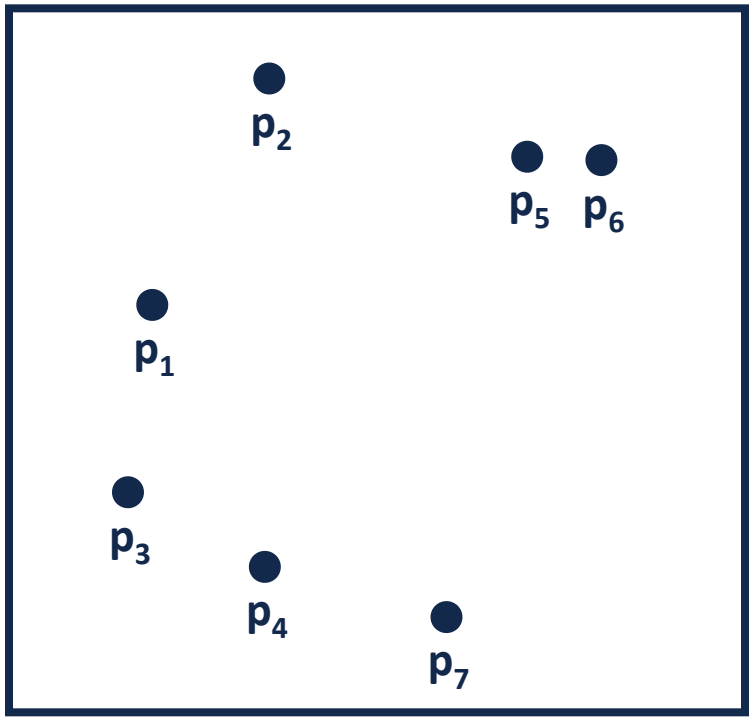




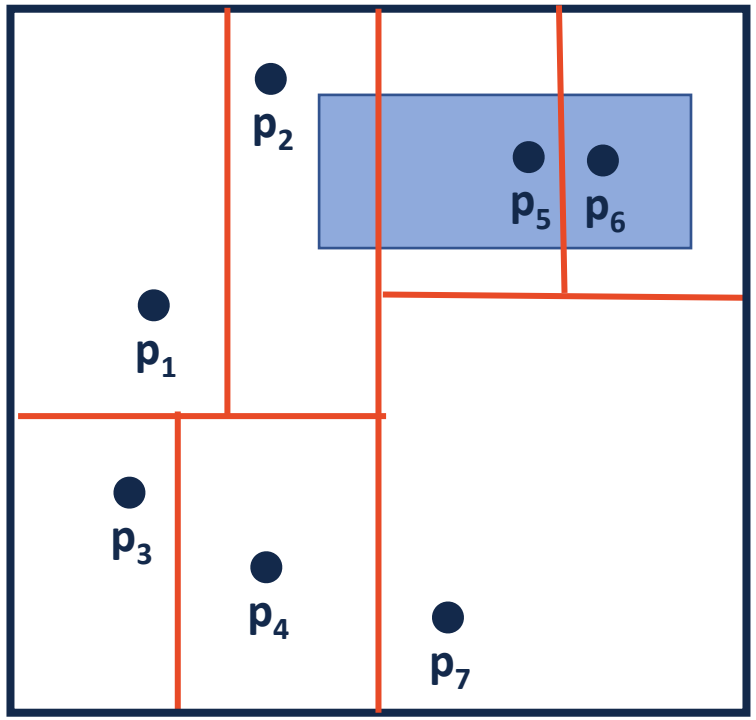
Range-based Searches

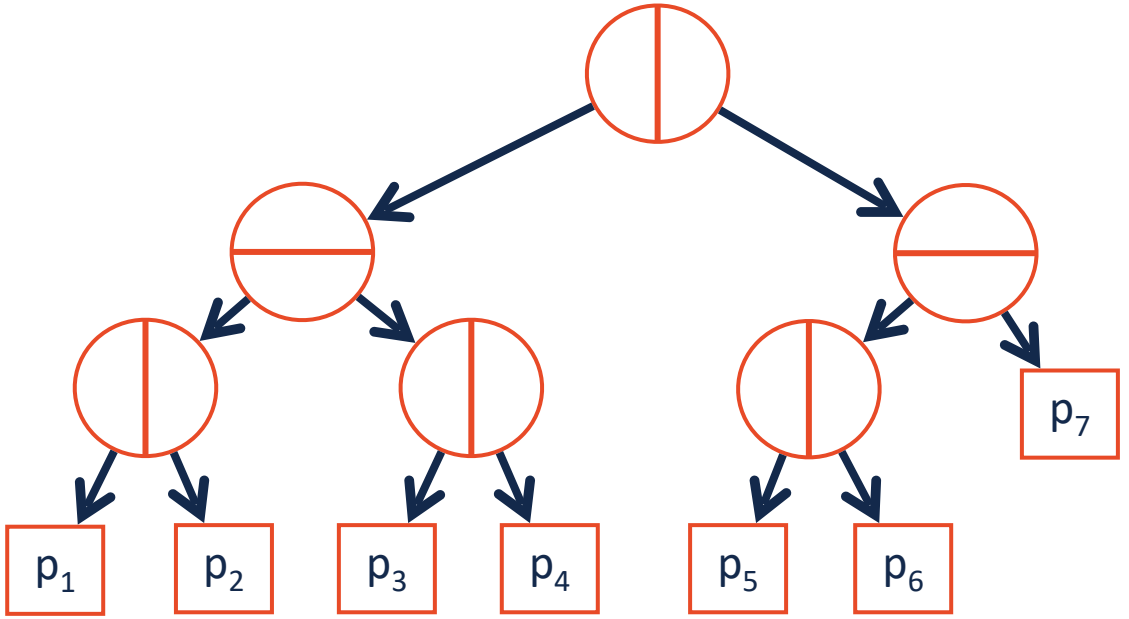
Consider points in 2D: $\mathbf{p = \{p_1, p_2, \ldots, p_n\}}$

What is nearest point to $\mathbf{(x_1, y_1)}$?



Range-based Searches

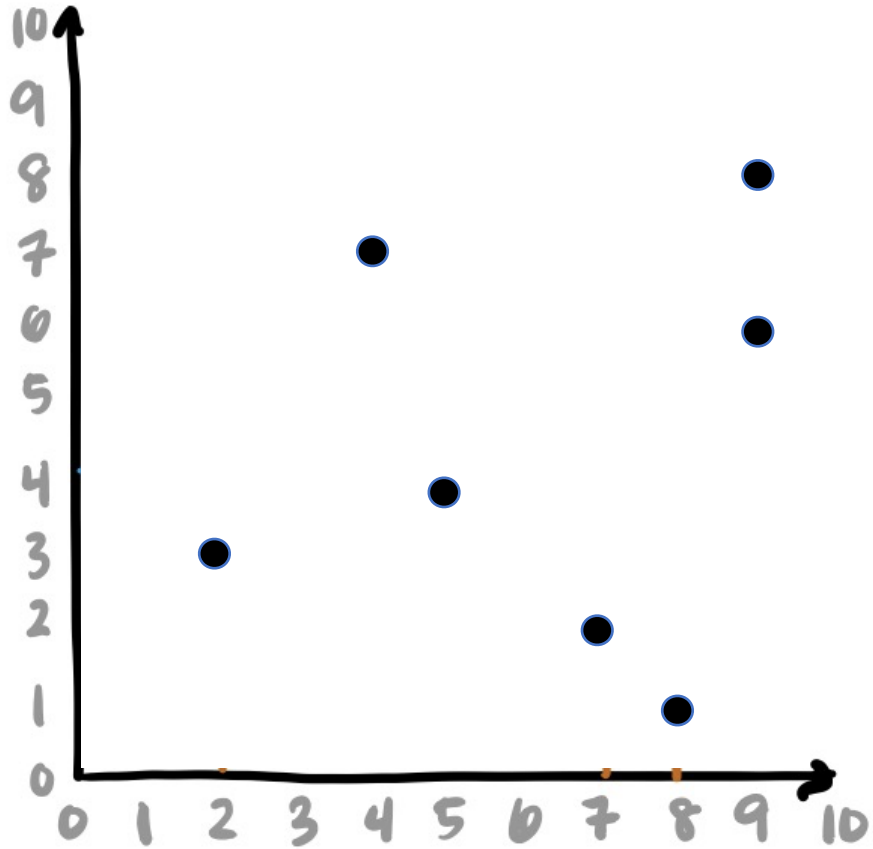




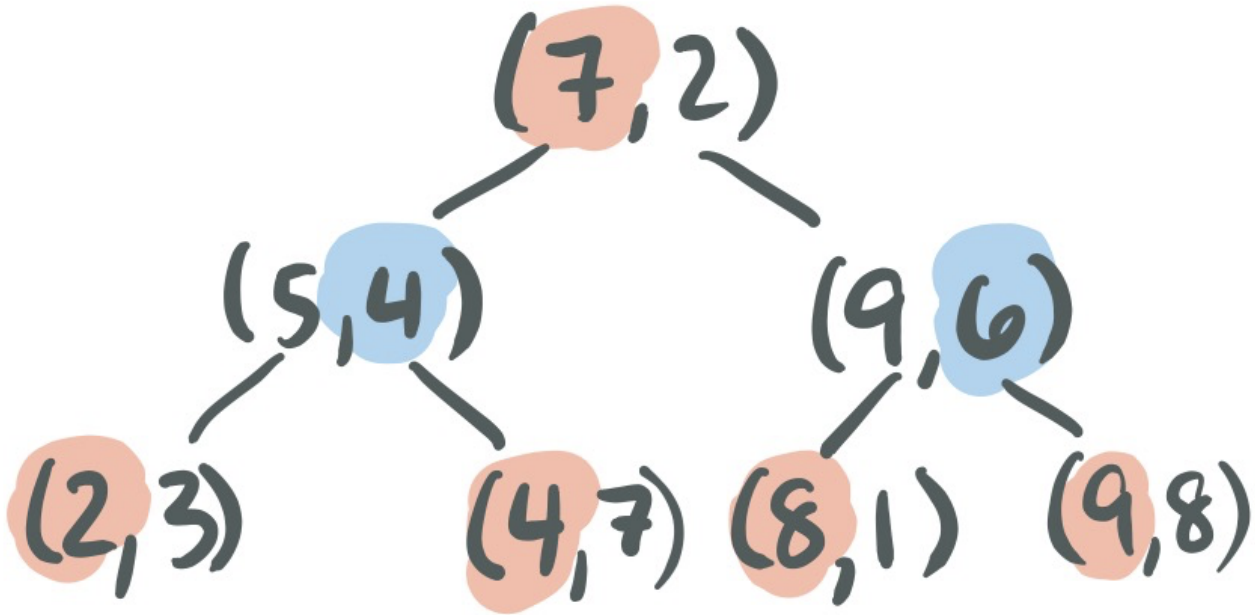
Nearest Neighbor: k-d tree

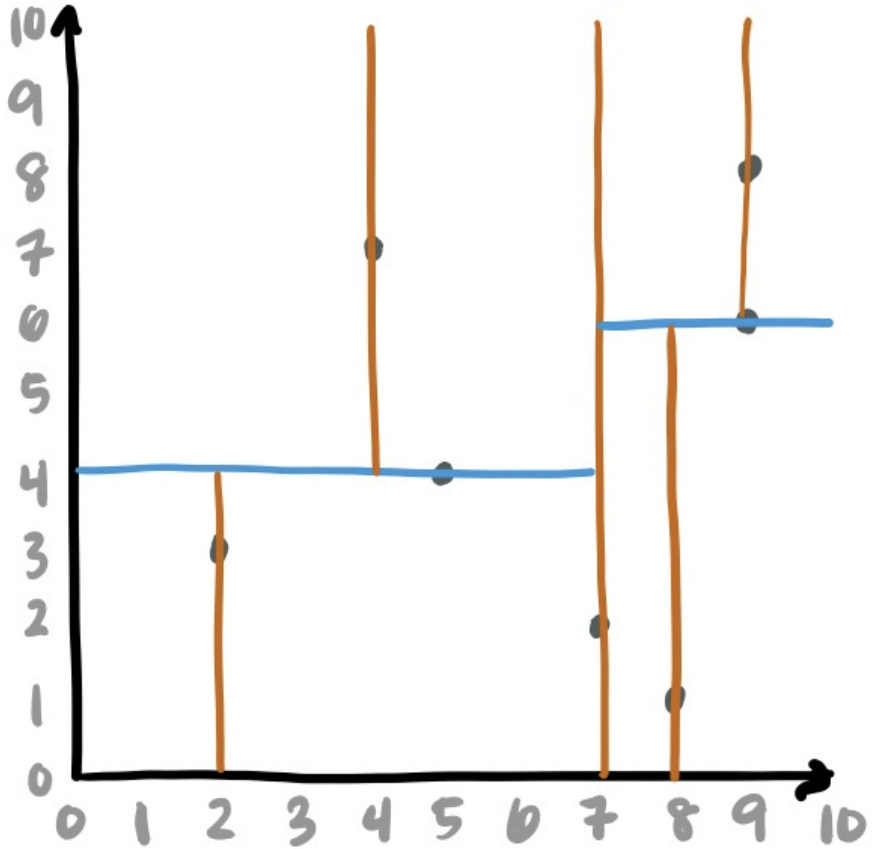
A **k-d tree** is similar but splits on points:

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



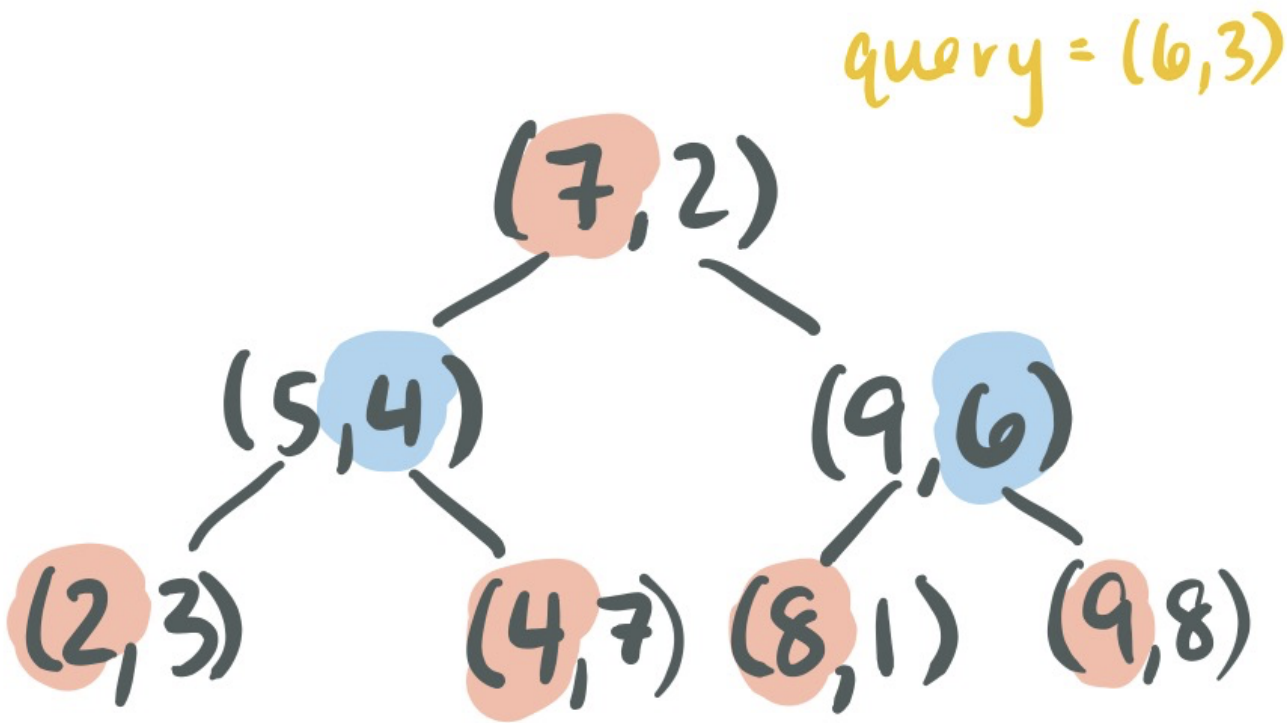
Nearest Neighbor: k-d tree

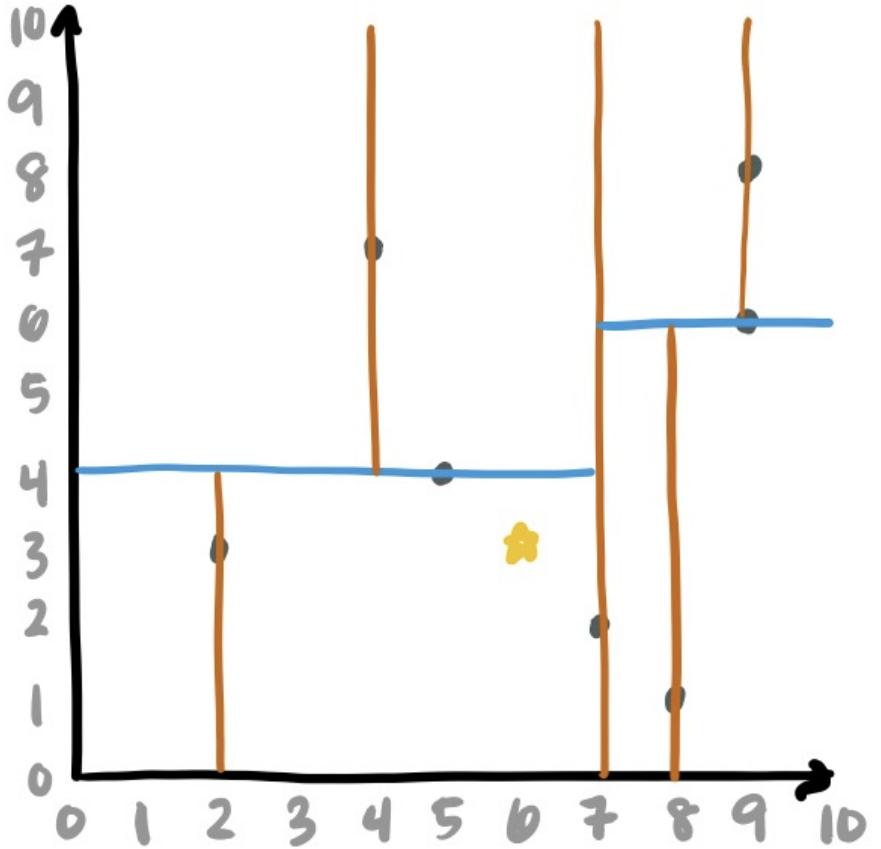




Nearest Neighbor: k-d tree

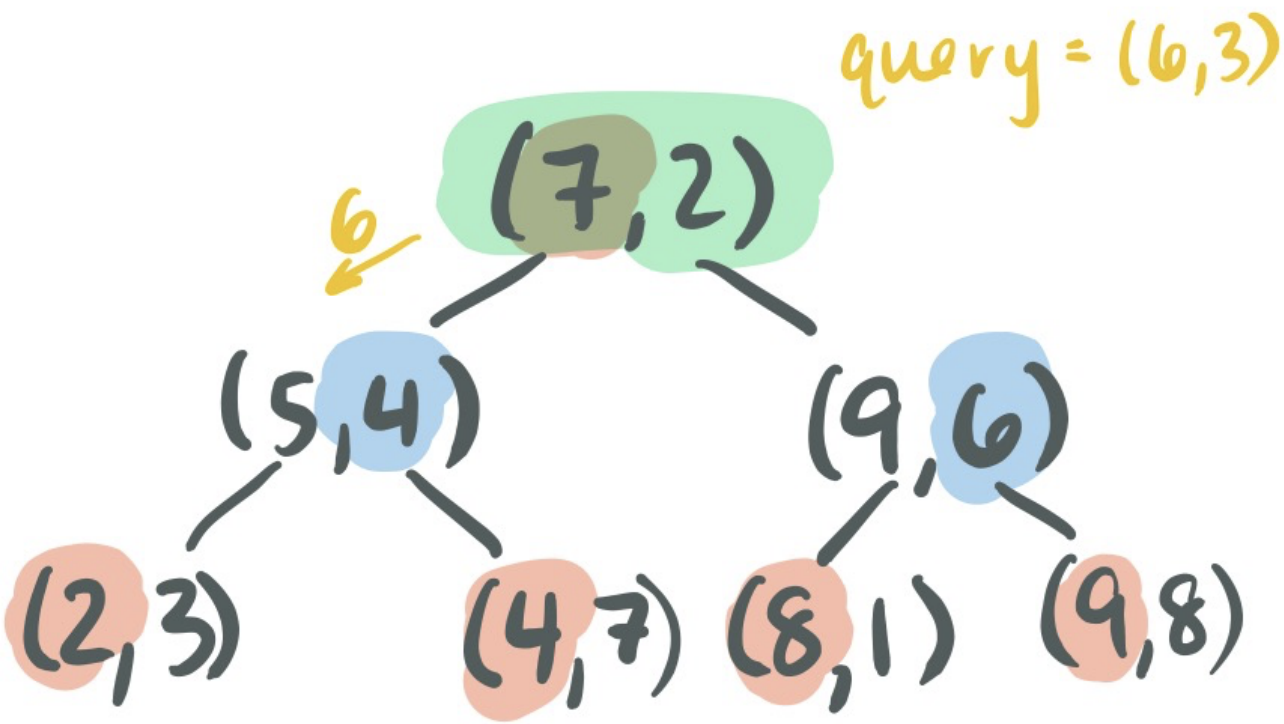
Search by comparing query and node in single dimension

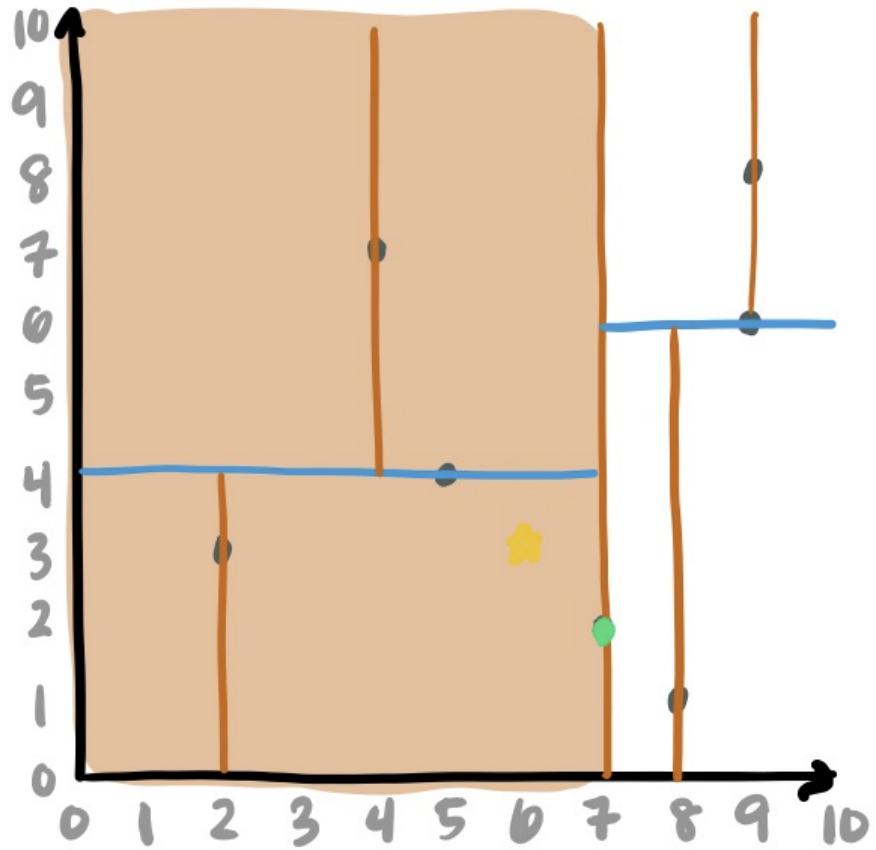




Nearest Neighbor: k-d tree

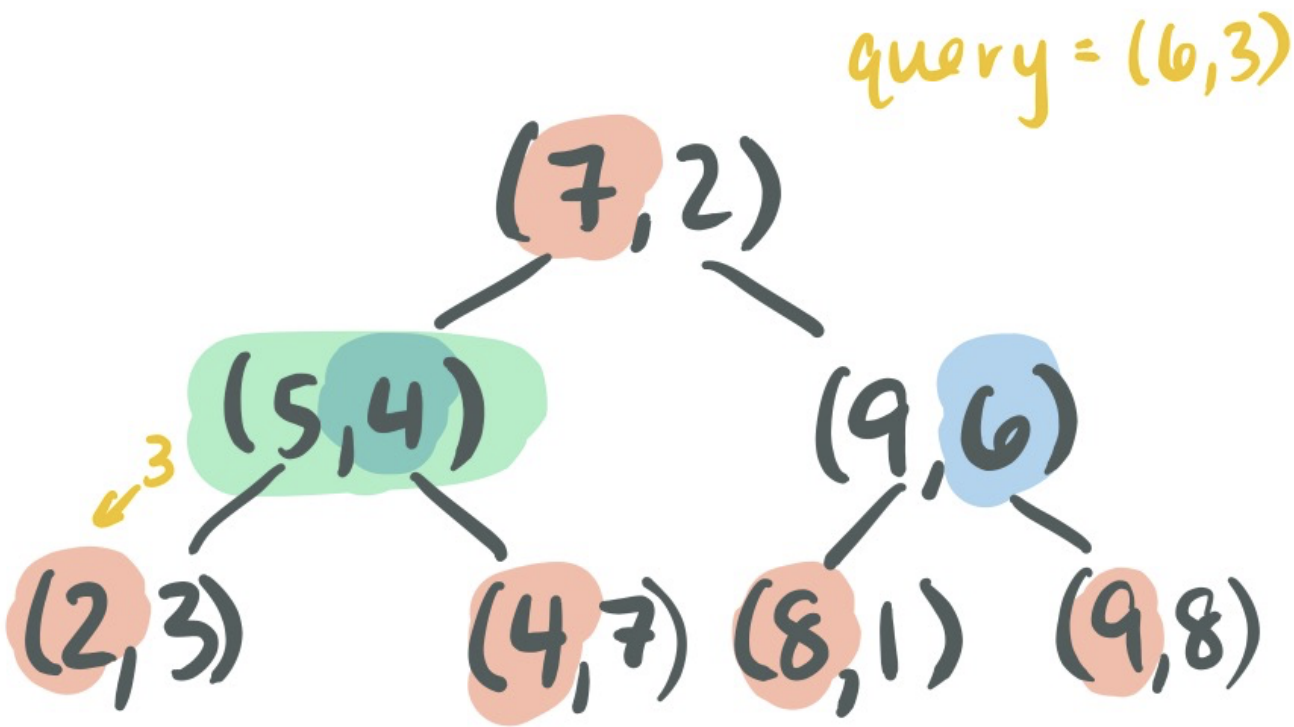
Search by comparing query and node in single dimension

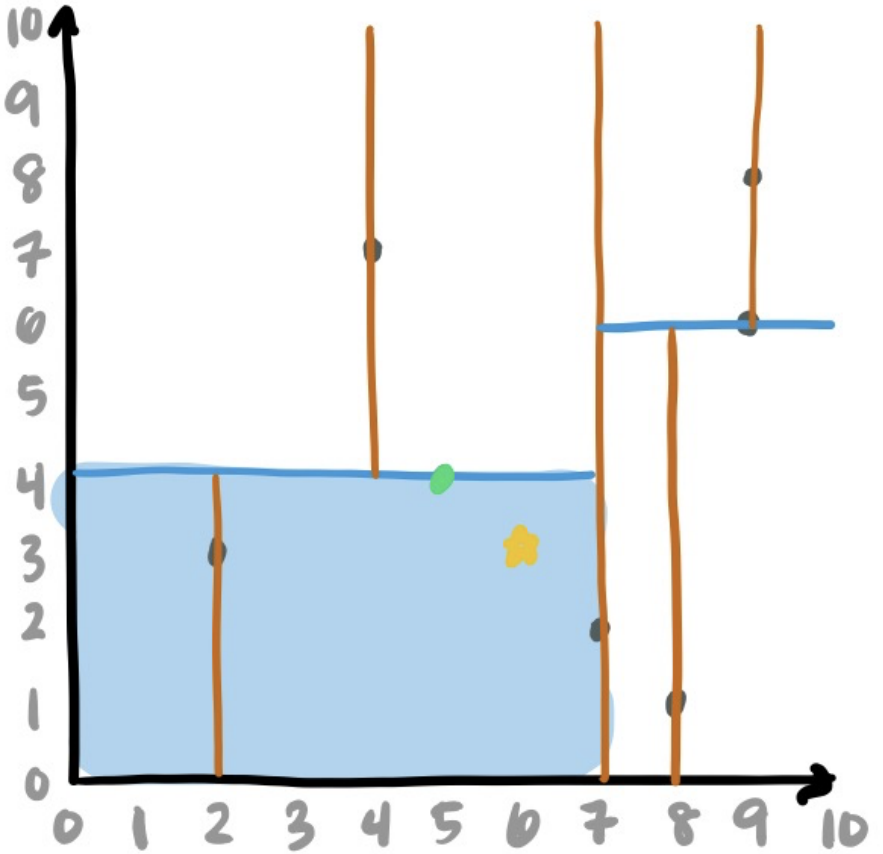




Nearest Neighbor: k-d tree

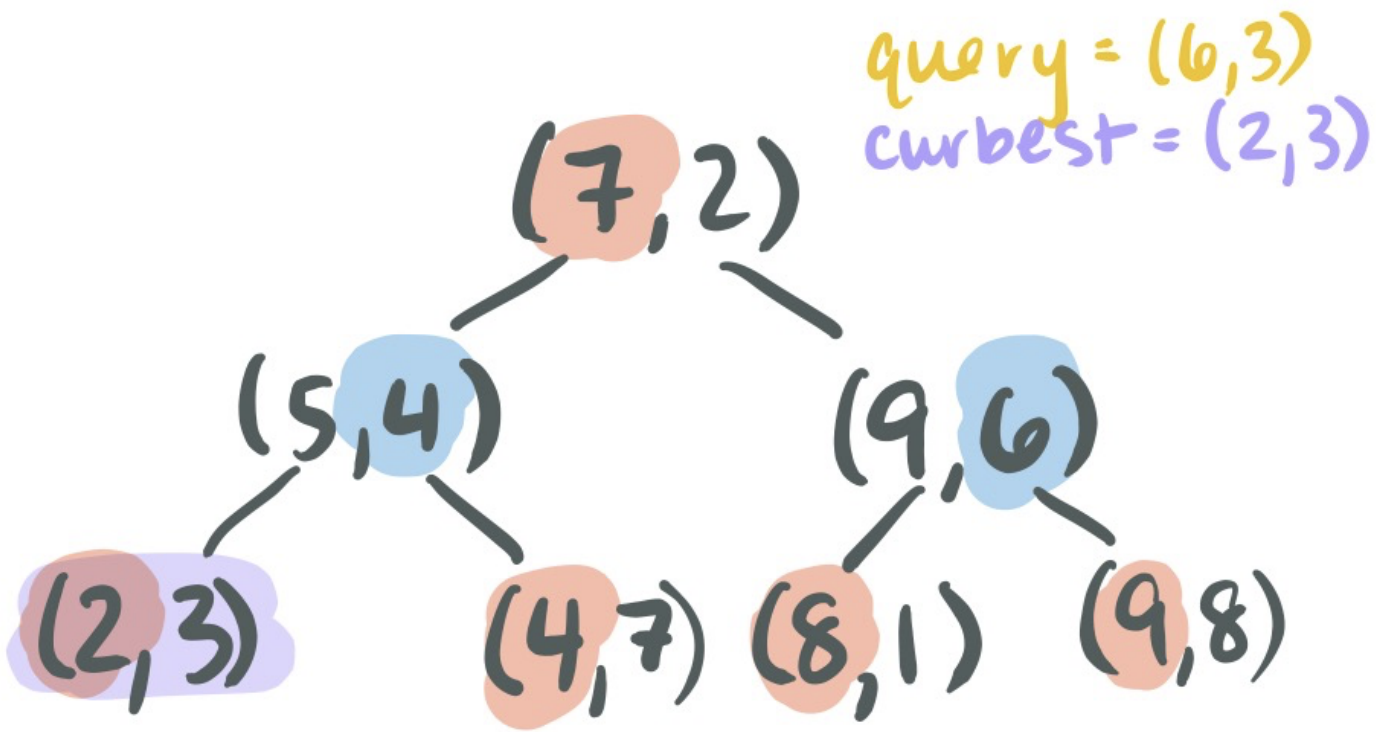
Search by comparing query and node in single **alternating** dimension

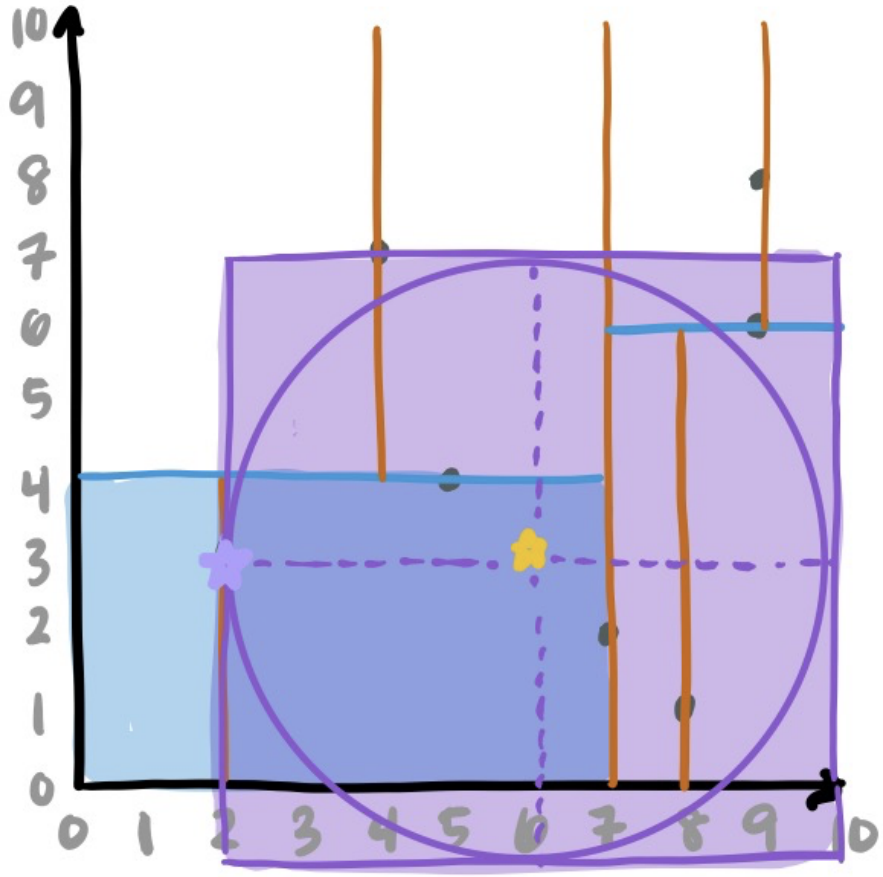




Nearest Neighbor: k-d tree

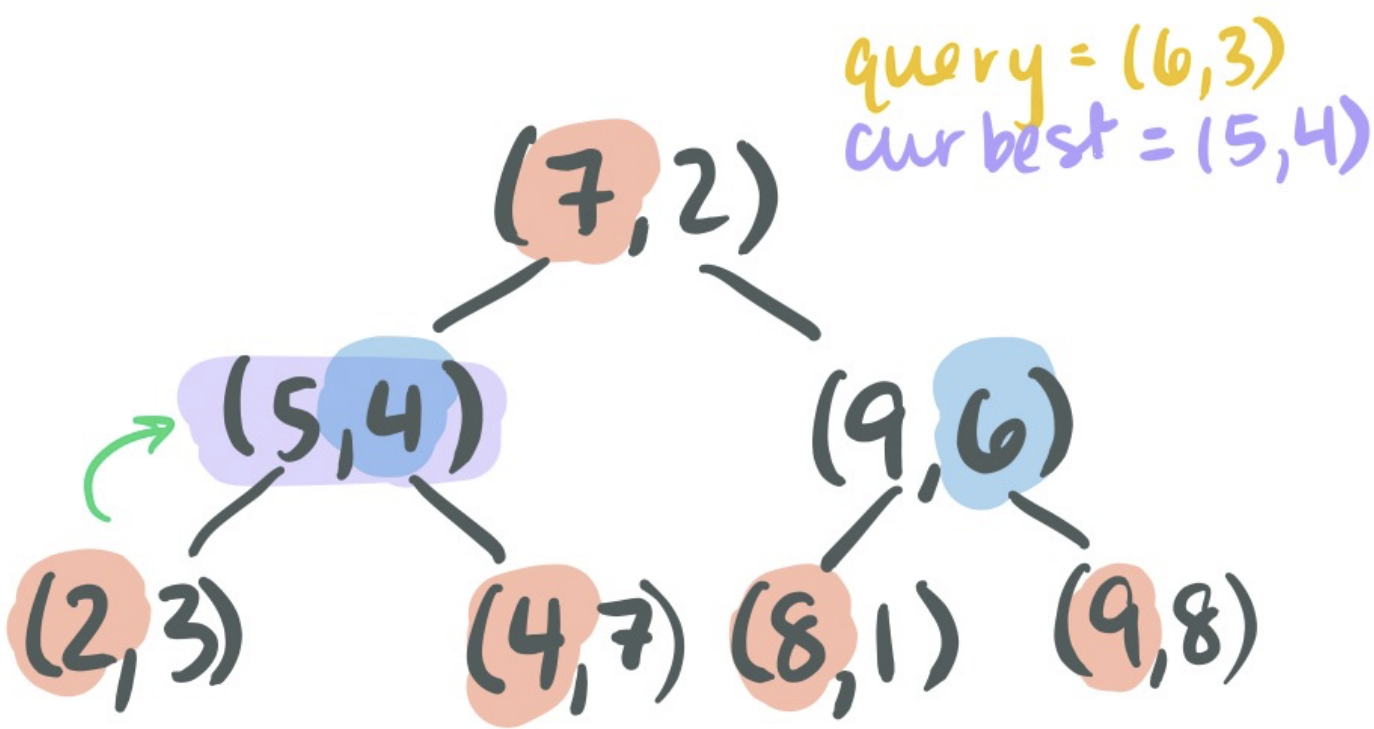
Nearest neighbor requires **backtracking**

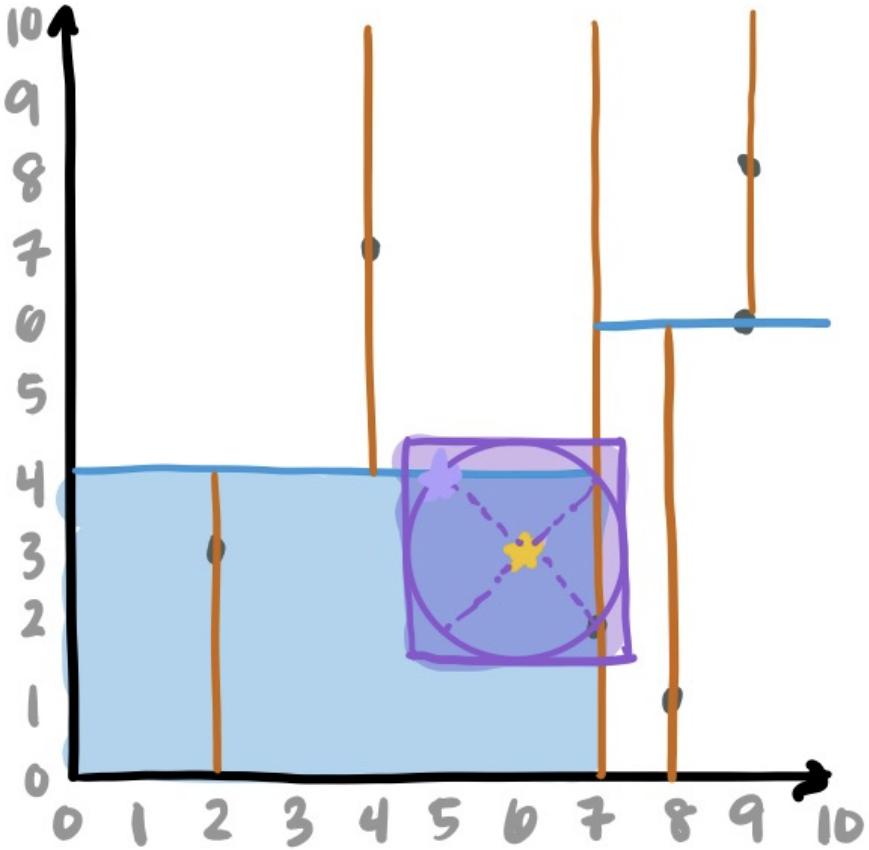




Nearest Neighbor: k-d tree

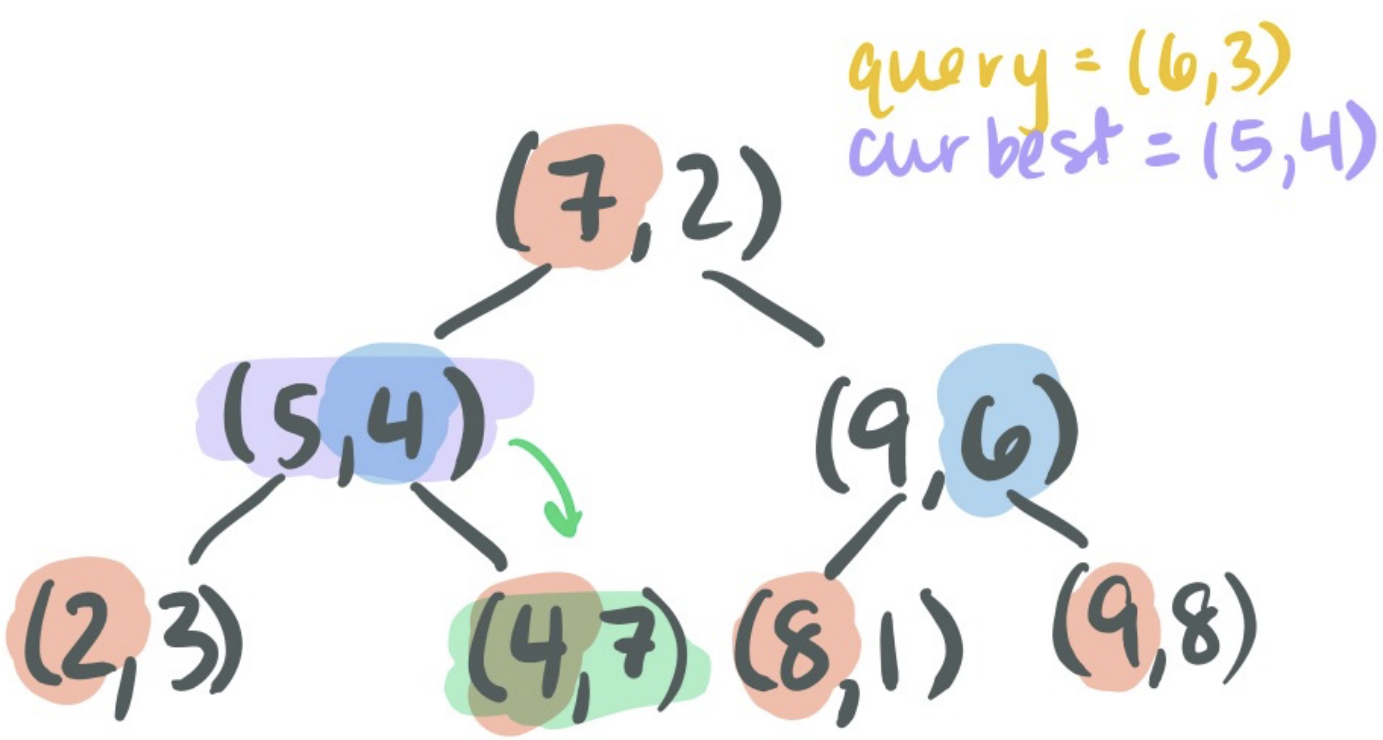
Backtracking: start recursing backwards -- store "best" possibility as you trace back

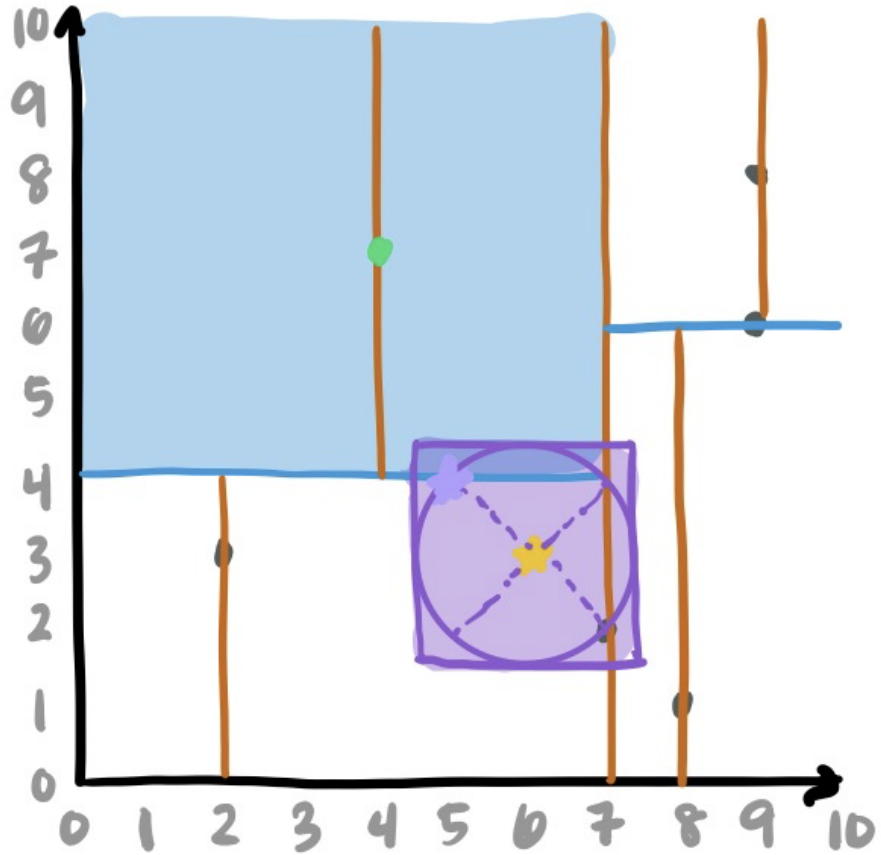




Nearest Neighbor: k-d tree

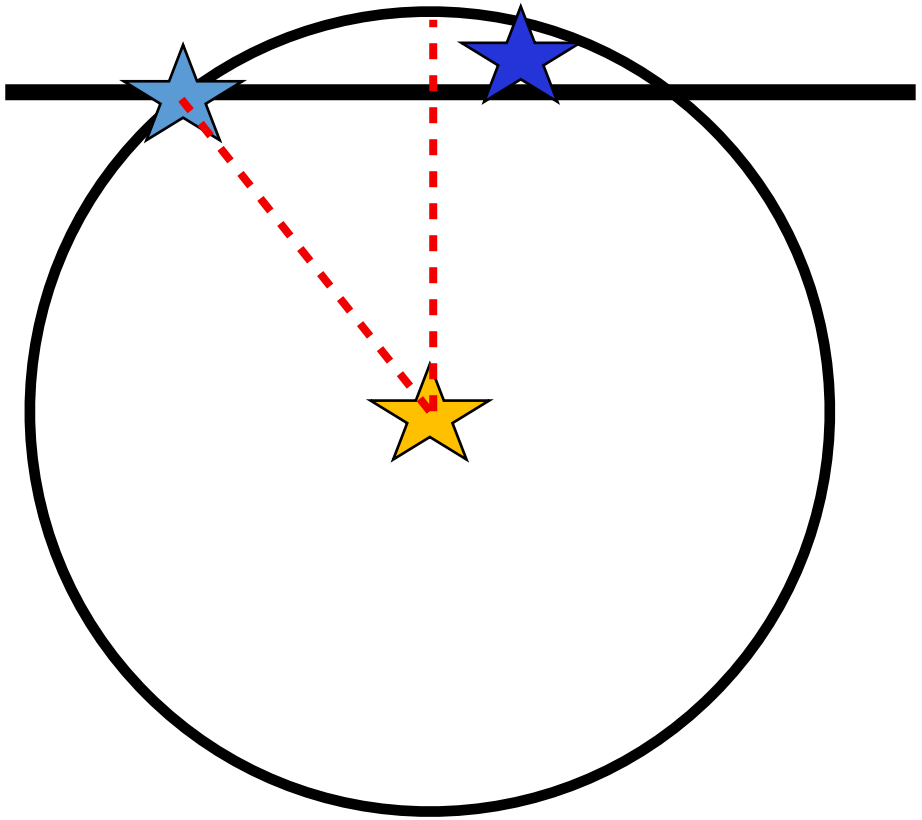
May have to recursively check other branches of tree — **why?**

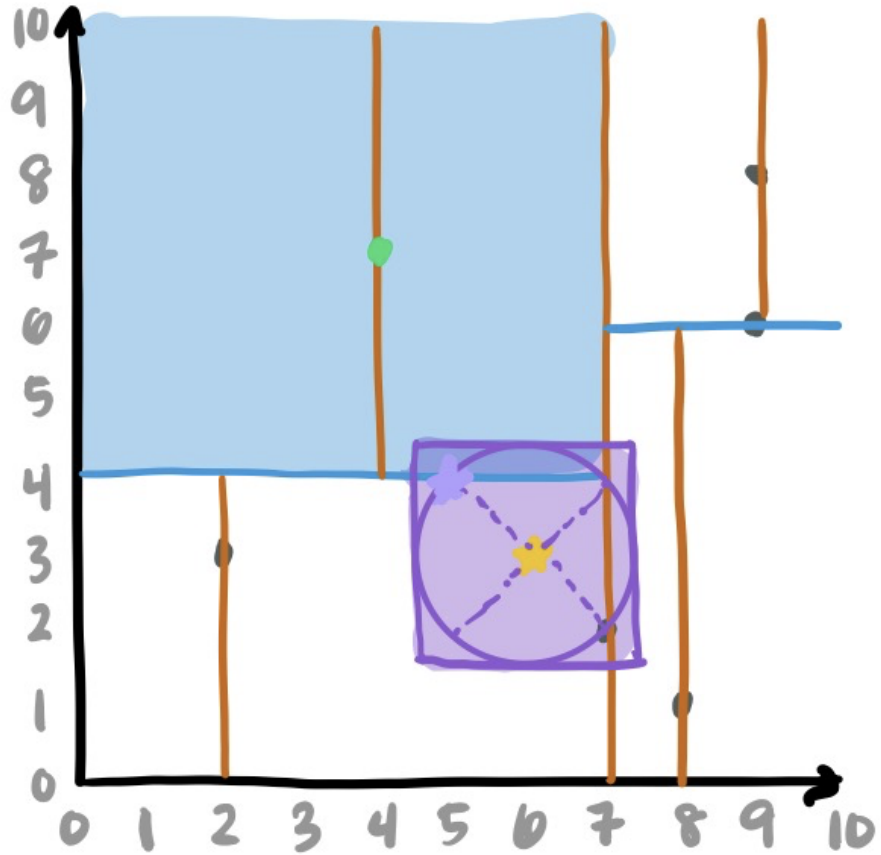




Nearest Neighbor: k-d tree

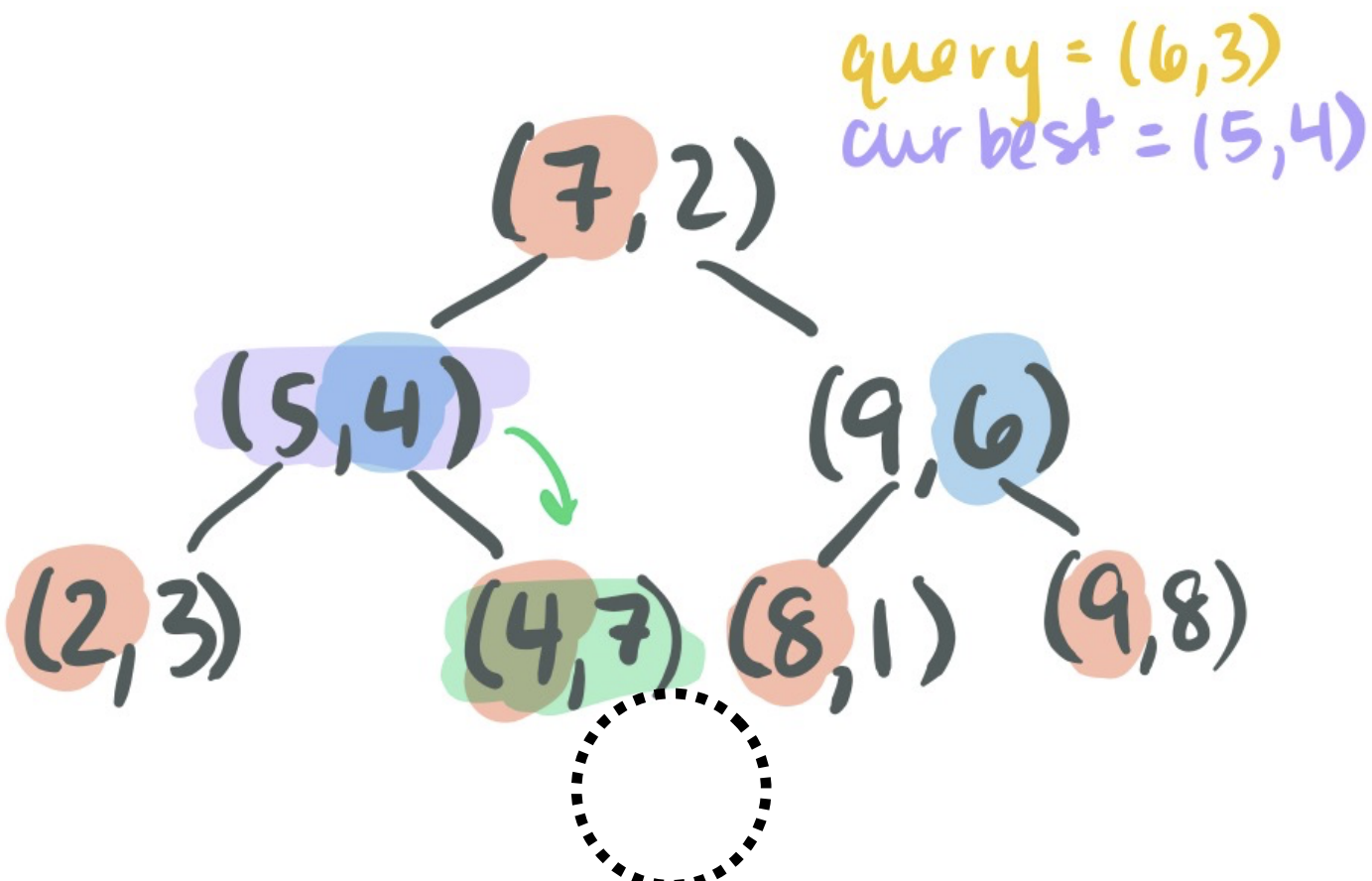
May have to recursively check other branches of tree — **why?**

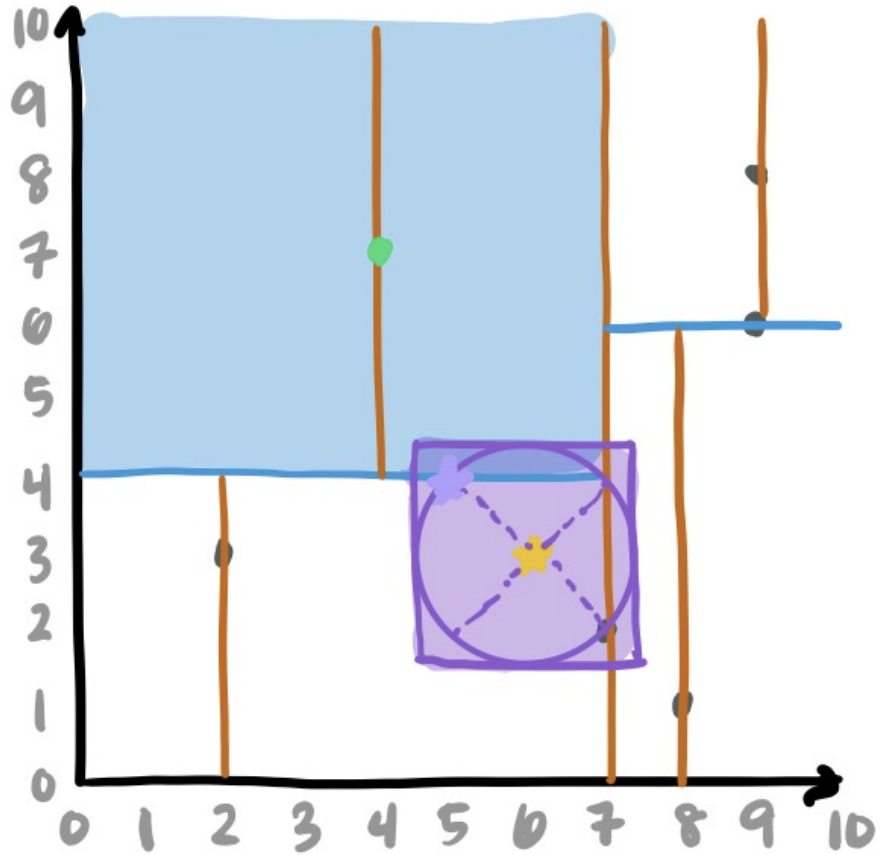




Nearest Neighbor: k-d tree

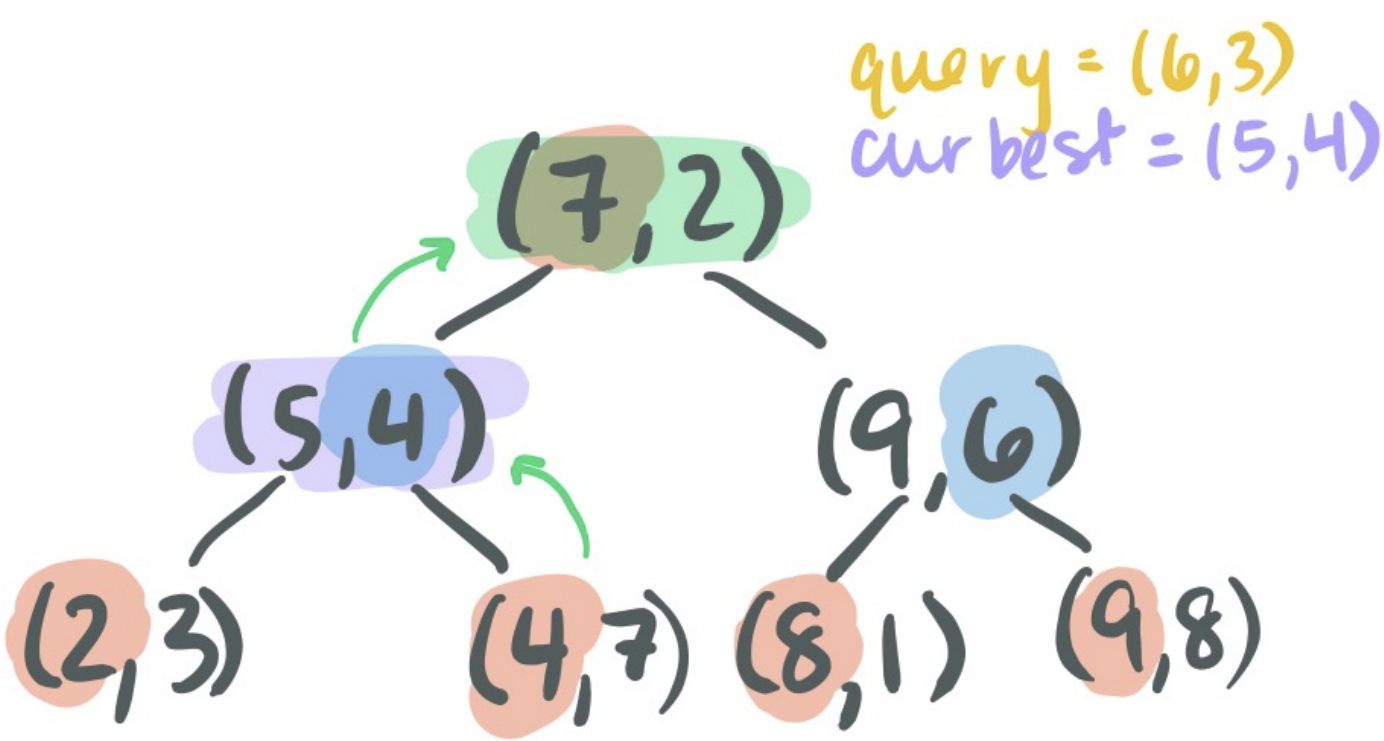
In this instance, there is no right child of (4, 7) so we continue…

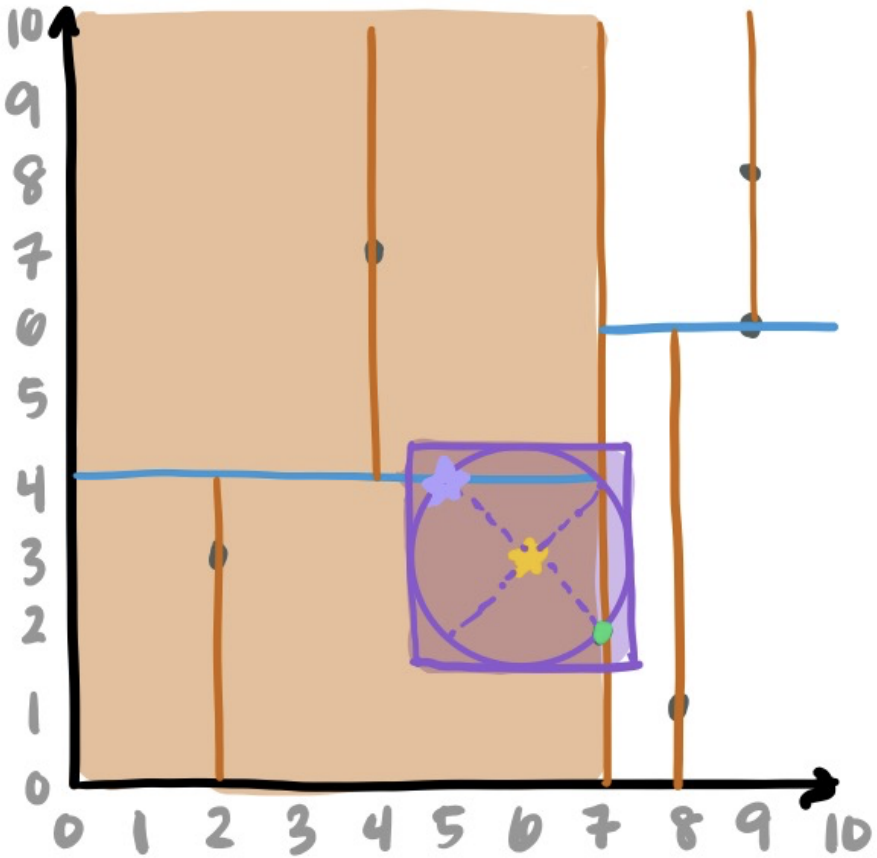




Nearest Neighbor: k-d tree

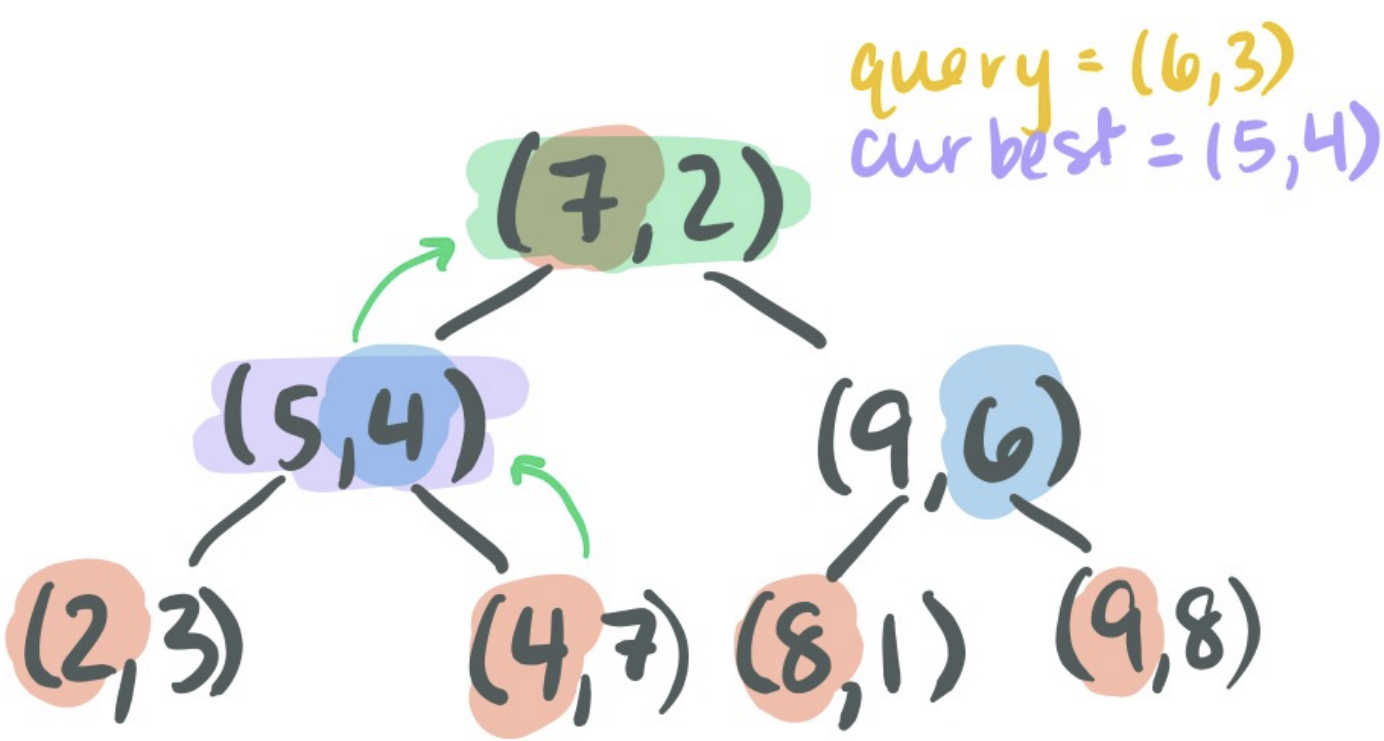
Tie breaking is described in doxygen (`smallerDimVal`)

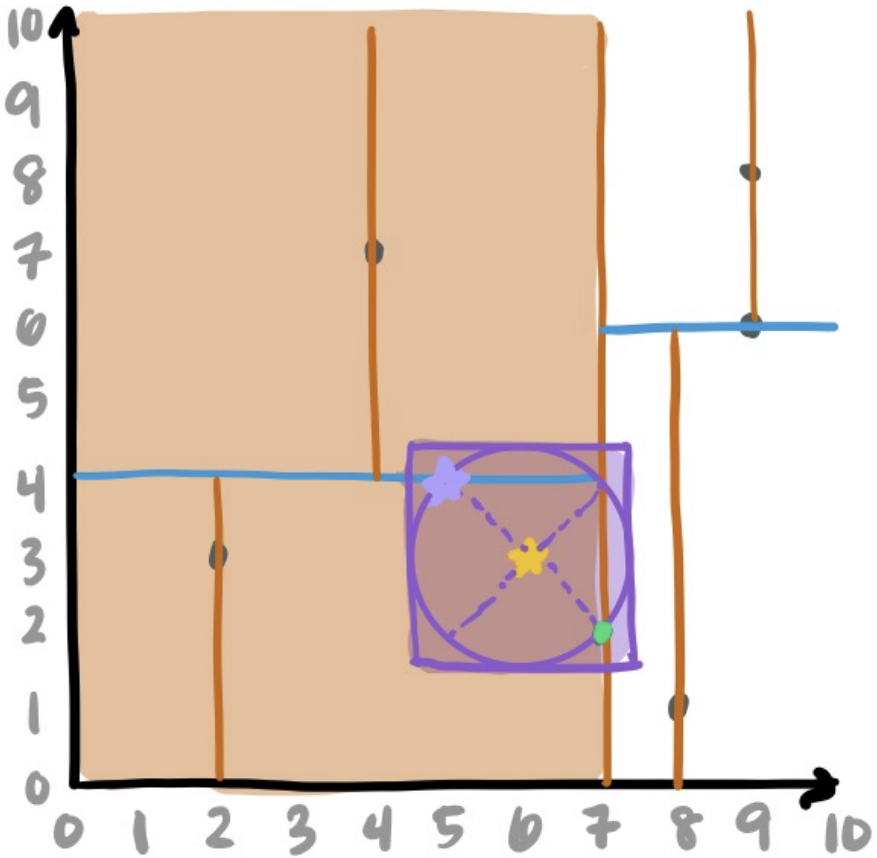




Nearest Neighbor: k-d tree

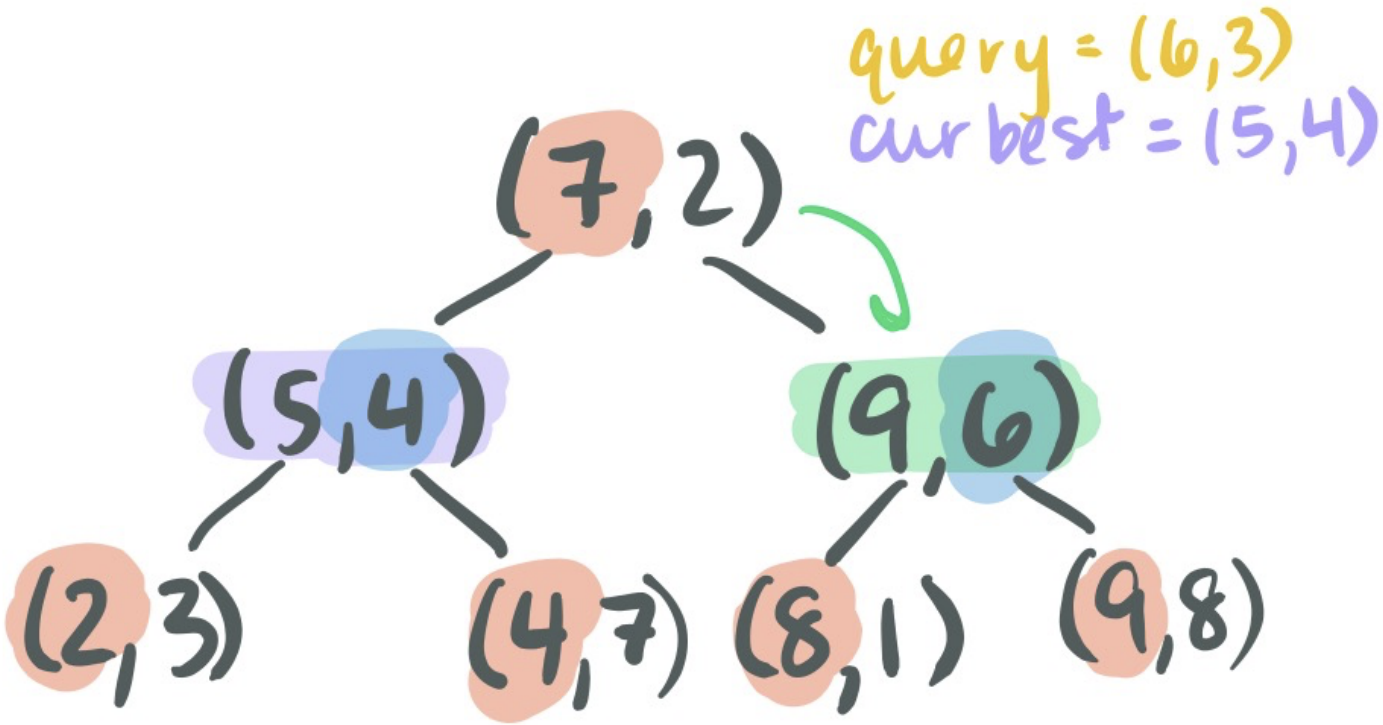
We've hit root and have a 'best' match — **are we done?**

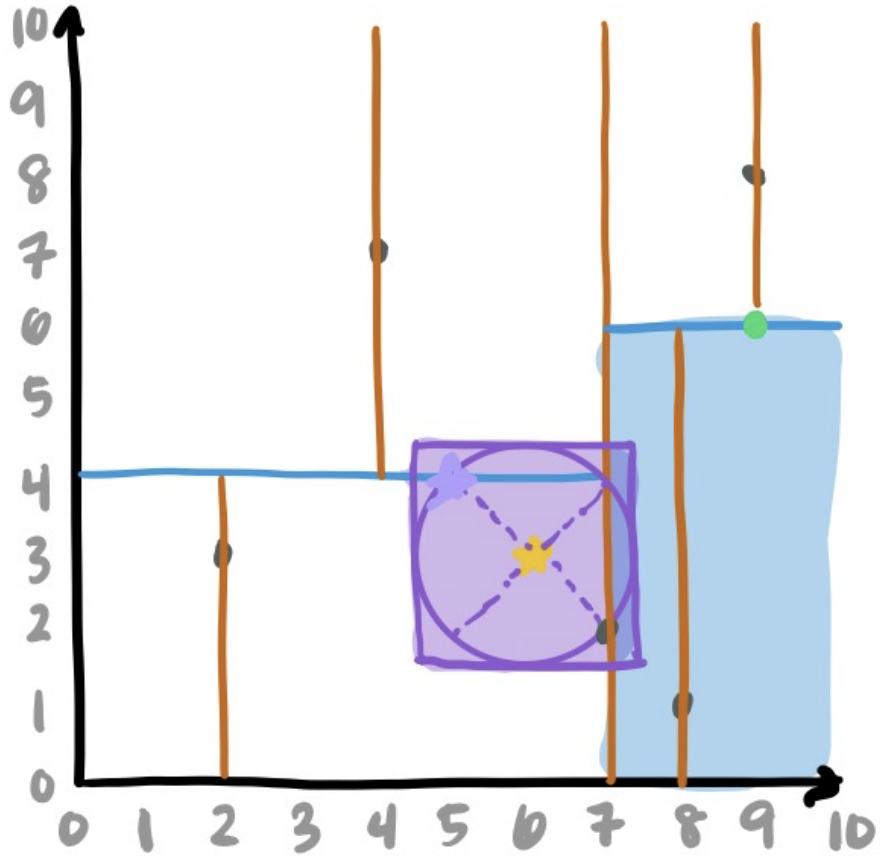




Nearest Neighbor: k-d tree

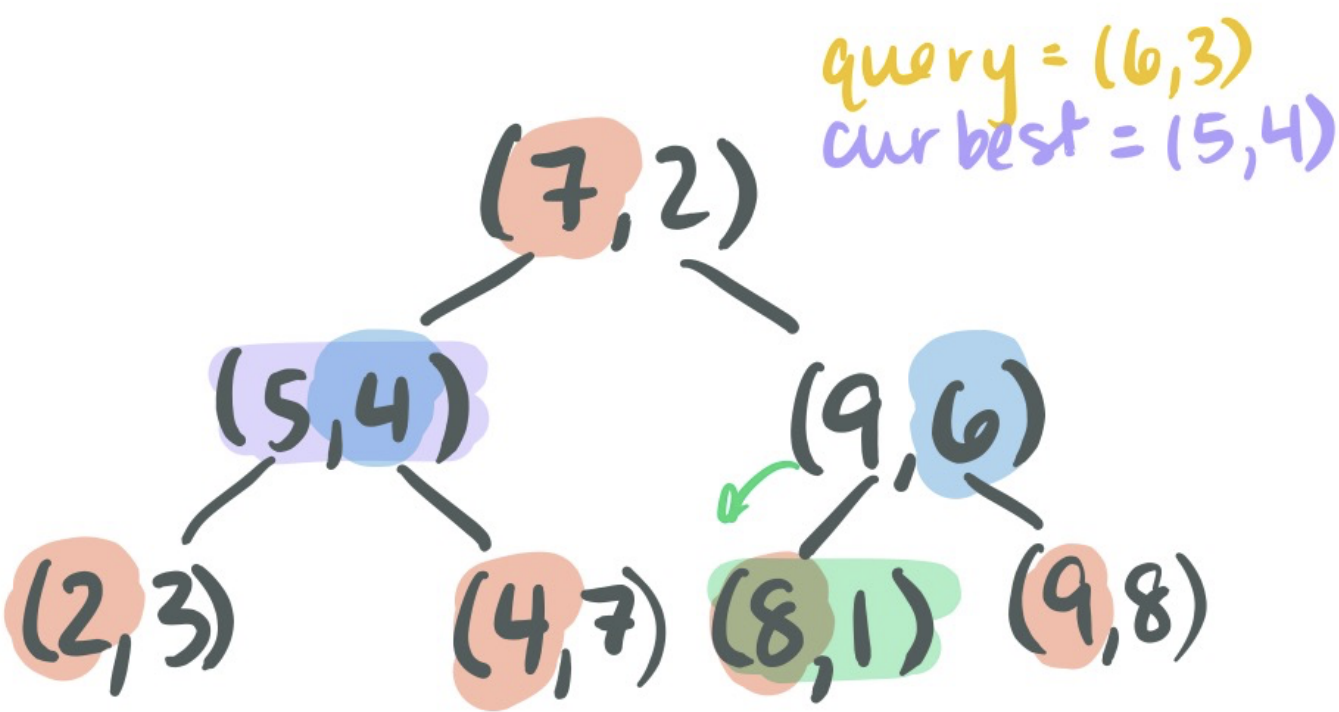
Why do we need to explore the right subtree?

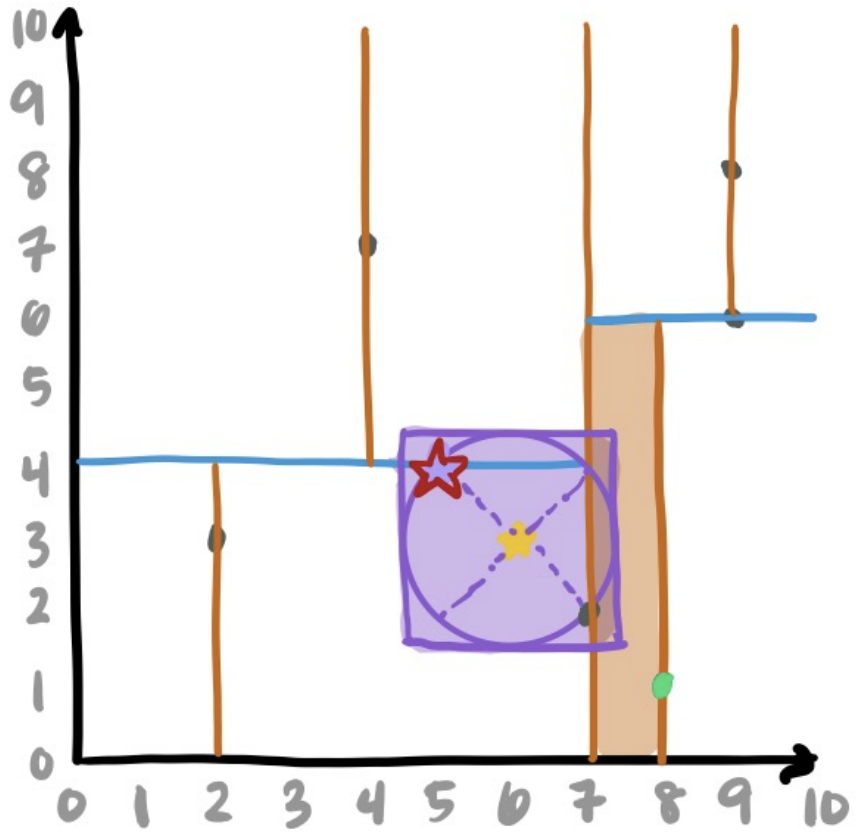




Nearest Neighbor: k-d tree

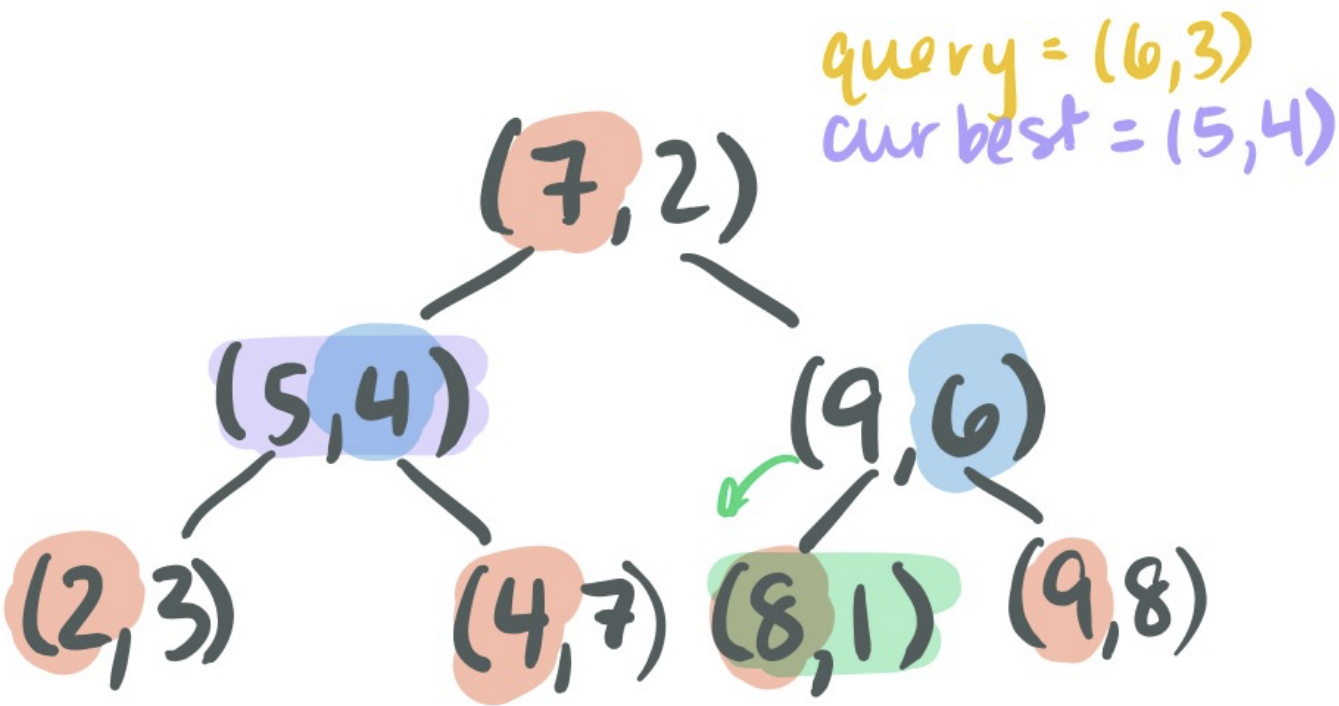
If there was a left child of (8,1), it could have been a better match!

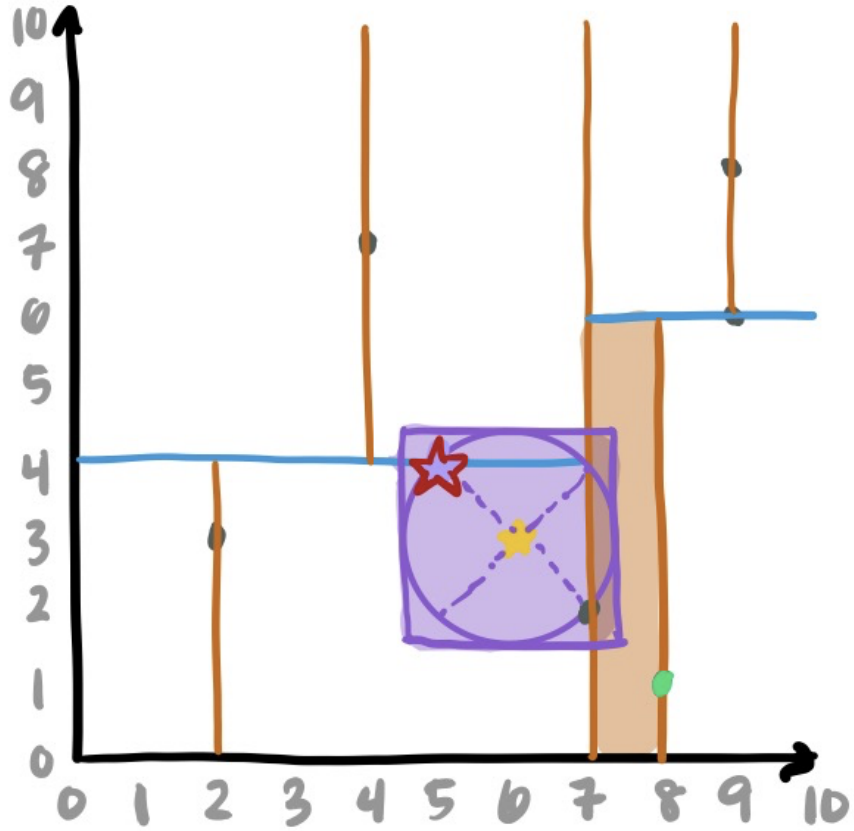




Nearest Neighbor: k-d tree

Having exhaustively explored for better matches, we are done!





BEST: (5,4)

Tips and Tricks for MP_Mosaics

1. Review, understand, and use **quickselect**

```
1  template <typename RandIter, typename Comparator>
2  void select(RandIter start, RandIter end, RandIter k, Comparator cmp)
3  {
4      /**
5       * @todo Implement this function!
6       */
7
8    }
9
```

2. Review, understand, and use **lambda functions**

Understanding ‘randIter’

An iterator is a container giving access in different ways:

Forward

Bidirectional

Random Access

Implementing quickselect with RandIter

Random Access Iterator lets you:

Swap items using std::swap()

```
1  template <typename RandIter, typename Comparator>
2  void BlackBox(RandIter A, RandIter B)
3  {
4
5     std::swap(*A, *B);
6
7
8  }
9
```

Hint: Look at pseudo-code for quickselect!

Implementing quickselect with RandIter

Random Access Iterator lets you:

Access container indices using math operations

```
randIter A;

auto nth = *(A + n);
```

Get distance between two iterators

```
randIter A, B;

 A < B;          // True if A is earlier in container than B

 A - B;          // The distance between A and B
```

Implementing quickselect with RandIter

Random Access Iterator lets you:

Do most things you'd expect an array to be able to do!

The power of the **Interface!**

https://en.cppreference.com/w/cpp/iterator/random_access_iterator

Tips and Tricks for MP_Mosaics

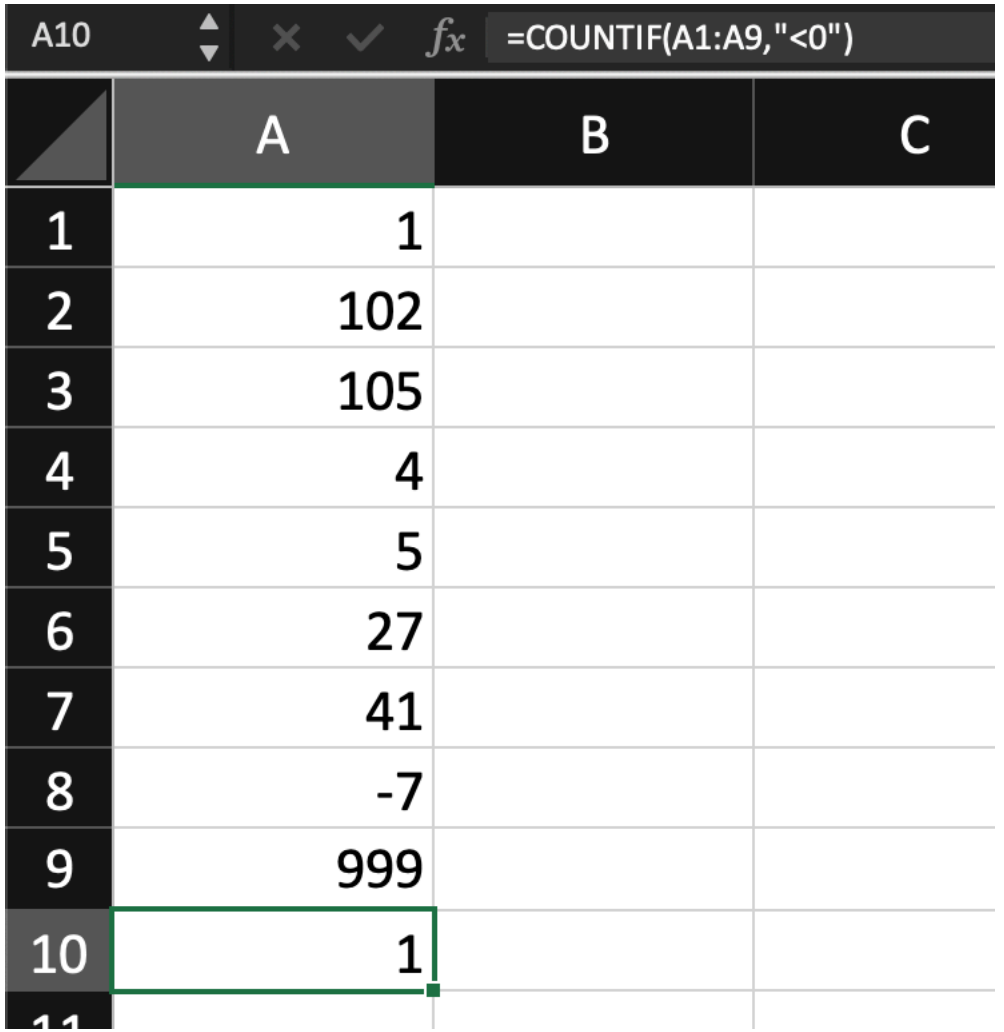
1. Review, understand, and use **quickselect**

```
1  template <typename RandIter, typename Comparator>
2  void select(RandIter start, RandIter end, RandIter k, Comparator cmp)
3  {
4      /**
5       * @todo Implement this function!
6       */
7
8    }
9
```

2. Review, understand, and use **lambda functions**

Functions as arguments

Consider the function from Excel
COUNTIF(*range*, *criteria*)



Functions as arguments

Countif.hpp

```
10 template <typename Iter, typename Pred>
11 int Countif(Iter begin, Iter end, Pred pred) {
12   int count = 0;
13   auto cur = begin;
14
15   while(cur != end) {
16     if(pred(*cur))
17       ++count;
18     ++cur;
19   }
20
21   return count;
22 }
```

Lambda Functions in C++

Here are several ways to write a function as an object

main.cpp

```
1  bool isNegative(int num) { return (num < 0); }
2
3  class IsNegative {
4  public:
5      bool operator() (int num) { return (num < 0); }
6  };
7
8  int main() {
9    std::vector<int> numbers = {1, 102, 105, 4, 5, 27, 41, -7, 999};
10
11   auto isnegl = [](int num) { return (num < 0); };
12   auto isnegfp = isNegative;
13   auto isnegfunctor = IsNegative();
14
15   cout << "There are " << Countif(numbers.begin(), numbers.end(), _______)
16     << " negative numbers" << std::endl;
17
```

Lambda Functions in C++

[Capture](Arg List){ Function Body}

Lambda Functions in C++

[Capture](Arg List){ Function Body}

Capture: Takes the value of object based on when the lambda was defined, NOT the current value of the object!

Arg List: Standard way of inputing into a function

Function Body: Code can use both capture vars and arg vars

Lambda Functions in C++

main.cpp

```
29    int big;
30    std::cout << "How big is big? ";
31    std::cin >> big;
32
33    auto isbig = [big](int num) { return (num >= big); };
34
35
36
37    std::cout << "There are " << Countif(numbers.begin(), numbers.end(), isbig)
38      << " big numbers" << std::endl;
}
```

Lambda Functions in C++

main.cpp

```
29    int big;
30    std::cout << "How big is big? ";
31    std::cin >> big;
32
33    auto isbig = [big](int num) { return (num >= big); };
34
35
36
37    std::cout << "There are " << Countif(numbers.begin(), numbers.end(), isbig)
38      << " big numbers" << std::endl;
  }
```

Useful for mp_mosaics!

KD-Tree will split points in one dimension

When comparing, we need to remember what dimension we are in!

Tips and Tricks for MP_Mosaics

Final tips:

The mp_mosaic writeup is long. **READ IT**

The suggestions in the writeup should be followed carefully

Summary of Balanced BST

Pros:

O(log N) for insert, find, remove

Optimal range queries in 1D

Cons:

O(log N) isn't that great

Large in-memory requirement

Considering hardware limitations

Can we always fit our data in main memory?

Where else can we keep our data?

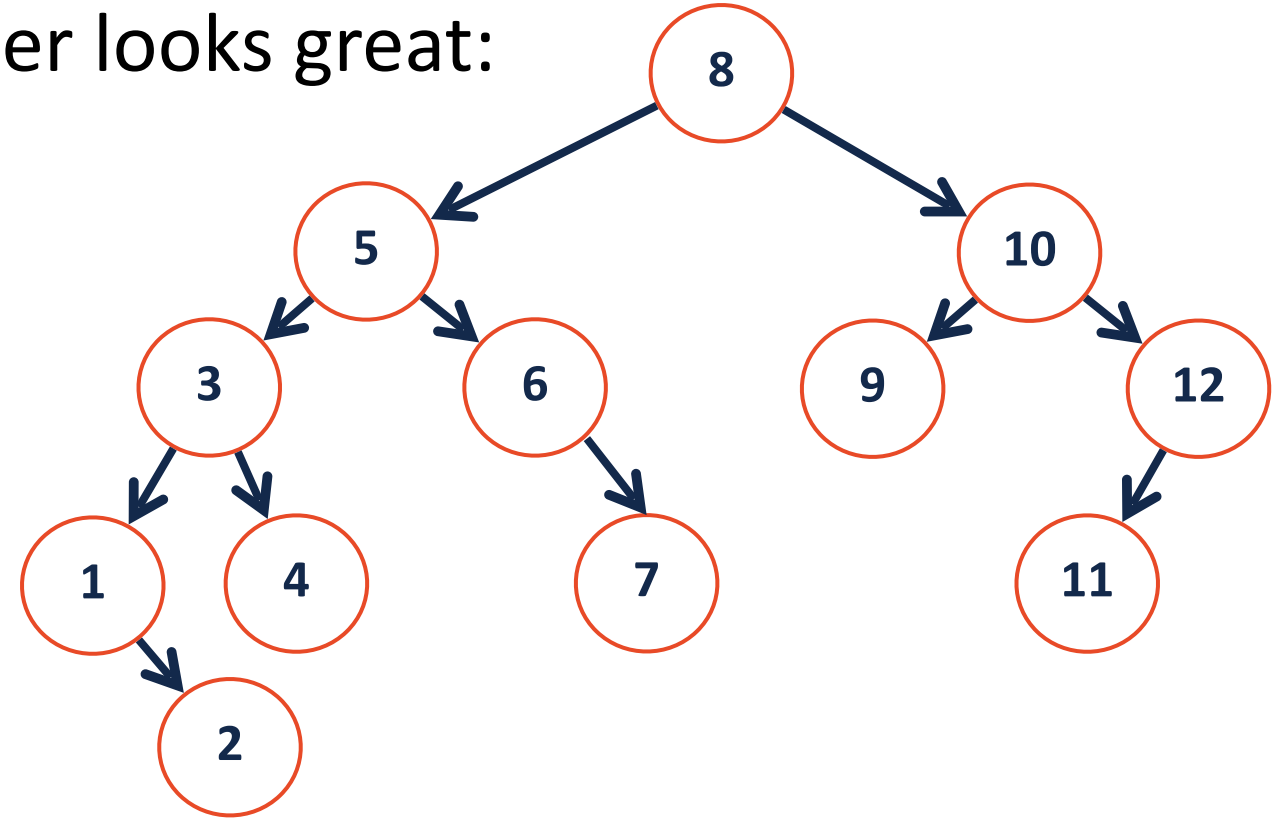
Does this match our assumption that all memory lookups are O(1)?

B-Tree Motivation

In Big-O we have assumed uniform time for all operations, but this isn't always true.

However, seeking data from the cloud may take 40ms+.

…an O(lg(n)) AVL tree no longer looks great:



BTree Design Motivations

When large seek times become an issue, we address this by:

BTree Design Motivations

When large seek times become an issue, we address this by:

1) Keep the number of seeks low

BTree Design Motivations

When large seek times become an issue, we address this by:

2) When possible keep data stored locally

BTree Design Motivations

When large seek times become an issue, we address this by:

3) Make sure the data we look up is relevant!

BTree Design Motivations

When large seek times become an issue, we address this by:

1) Keep the number of seeks low

2) When possible keep data stored locally

3) Make sure the data we look up is relevant!