Data Structures

K-d Tree

CS 225
Brad Solomon & G Carl Evans

September 29, 2023

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

MP_Lists Plagiarism Report

Significant increase in plagiarism

Still processing all the FAIR cases

Remember course policies!

MP_Mosaic Extra Credit Extension

Todays lecture will 'review' several key concepts

Concepts may be new to some, extra credit is extended

Extra credit deadline: Wednesday

Learning Objectives

Discuss (one) extension beyond BST

Introduce lambda functions in C++

Finish AVL proof and introduce B-Trees

Summary of Balanced BST

AVL Trees

- Max height: ???? * lg(n)
- Rotations:

  Zero rotations on find

  One rotation on insert

  O(**h**) == O(**lg(n)**) rotations on remove

Range-based Searches

Balanced BSTs are useful structures for range-based and nearest-neighbor searches.

Q: Consider points in 1D: $\mathbf{p = \{p_1, p_2, \ldots, p_n\}}$.

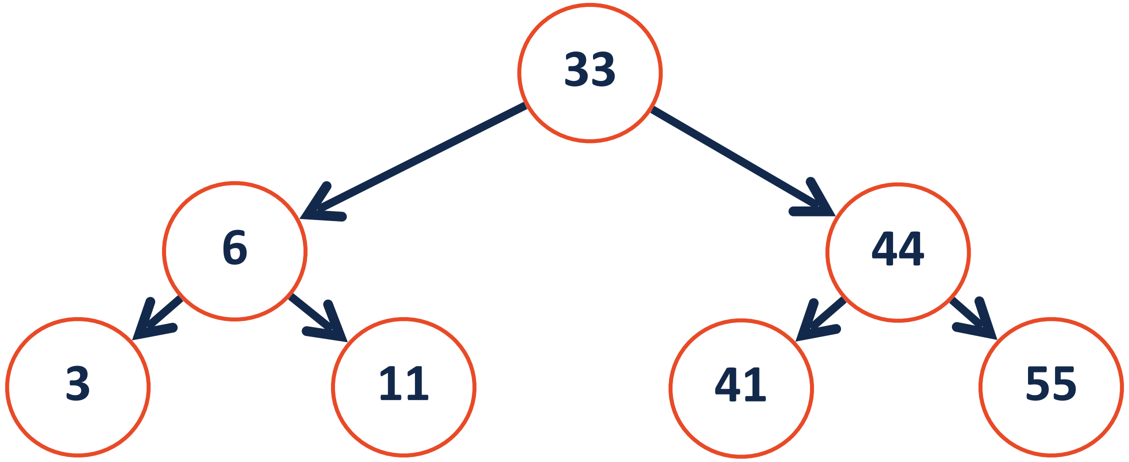
…what points fall in [11, 42]?

Ex:

Range-based Searches

Q: Consider points in 1D: $\mathbf{p} = \{p_1, p_2, \ldots, p_n\}$.

…what points fall in [11, 42]?



Red-Black Trees in C++

C++ provides us a balanced BST as part of the standard library:

```
std::map<K, V> map;

V & std::map<K, V>::operator[]( const K & )

std::map<K, V>::erase( const K & )
```

Red-Black Trees in C++

C++ provides us a balanced BST as part of the standard library:

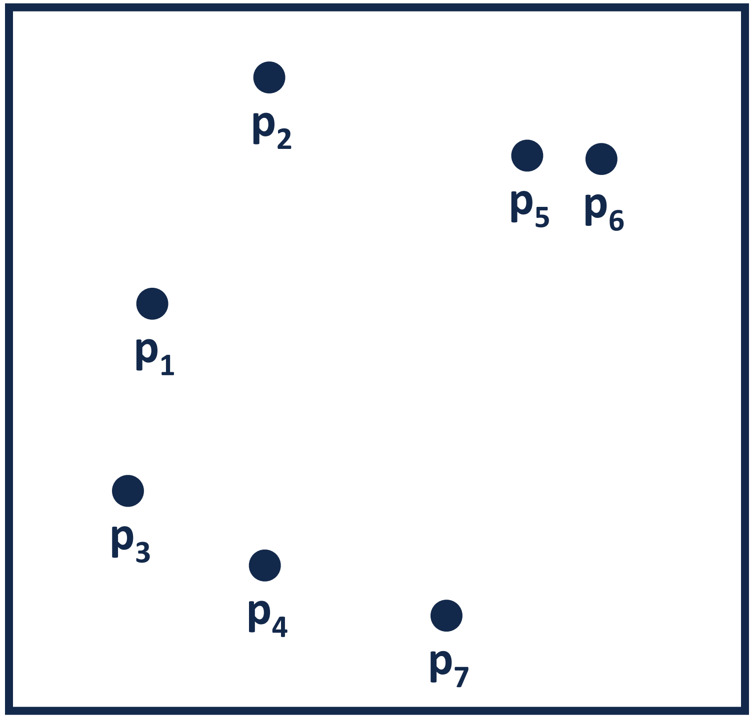
```
iterator std::map<K, V>::lower_bound( const K & );

iterator std::map<K, V>::upper_bound( const K & );
```

Range-based Searches

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

Q: What points are in the rectangle:
[(x_1, y_1), (x_2, y_2)]?

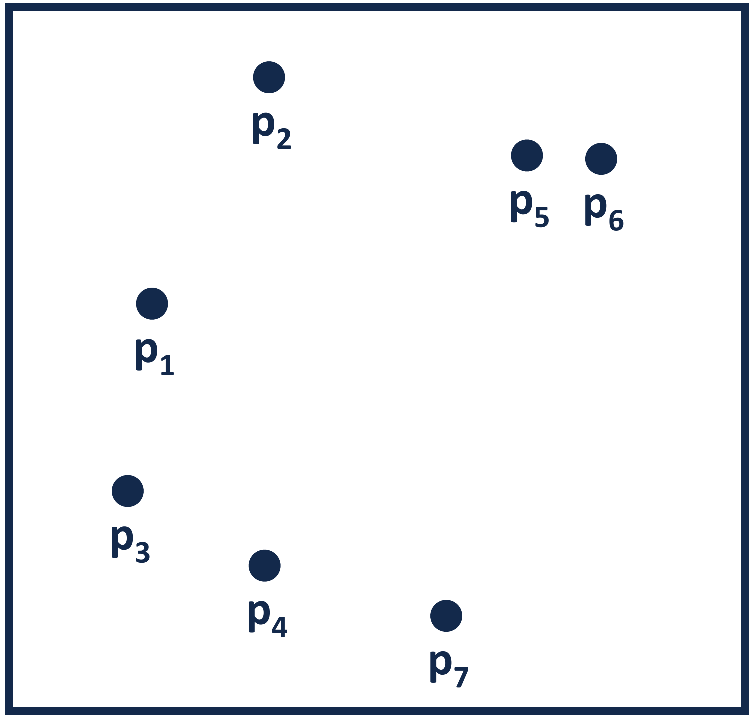
Q: What is the nearest point to **(x_1, y_1)**?



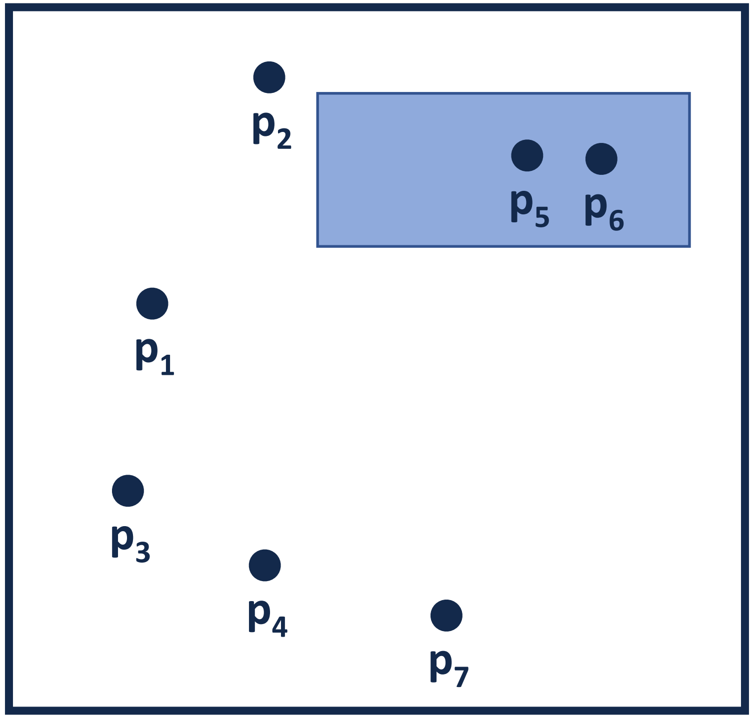
Range-based Searches

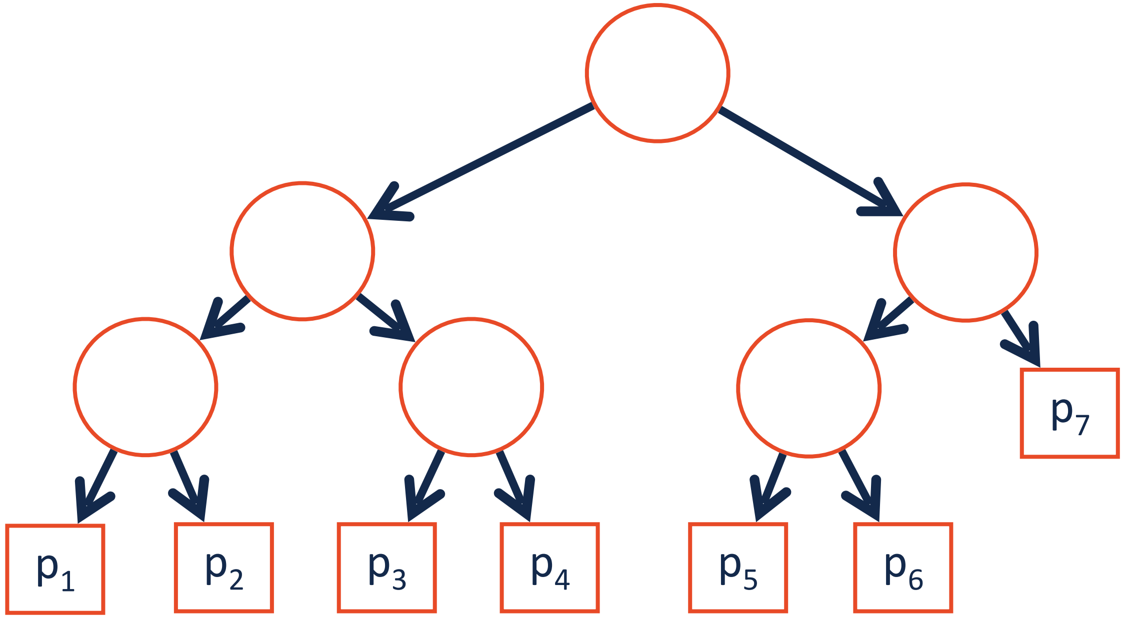
Consider points in 2D: $\mathbf{p} = \{\mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_n}\}$.

Tree construction:

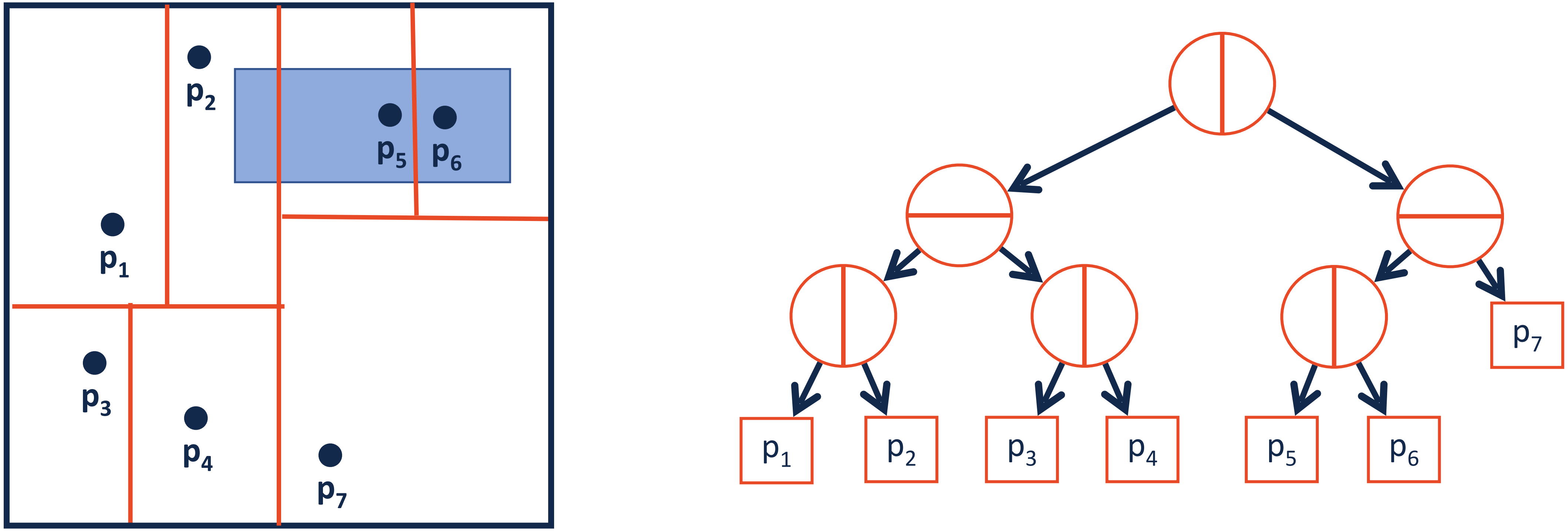


Range-based Searches





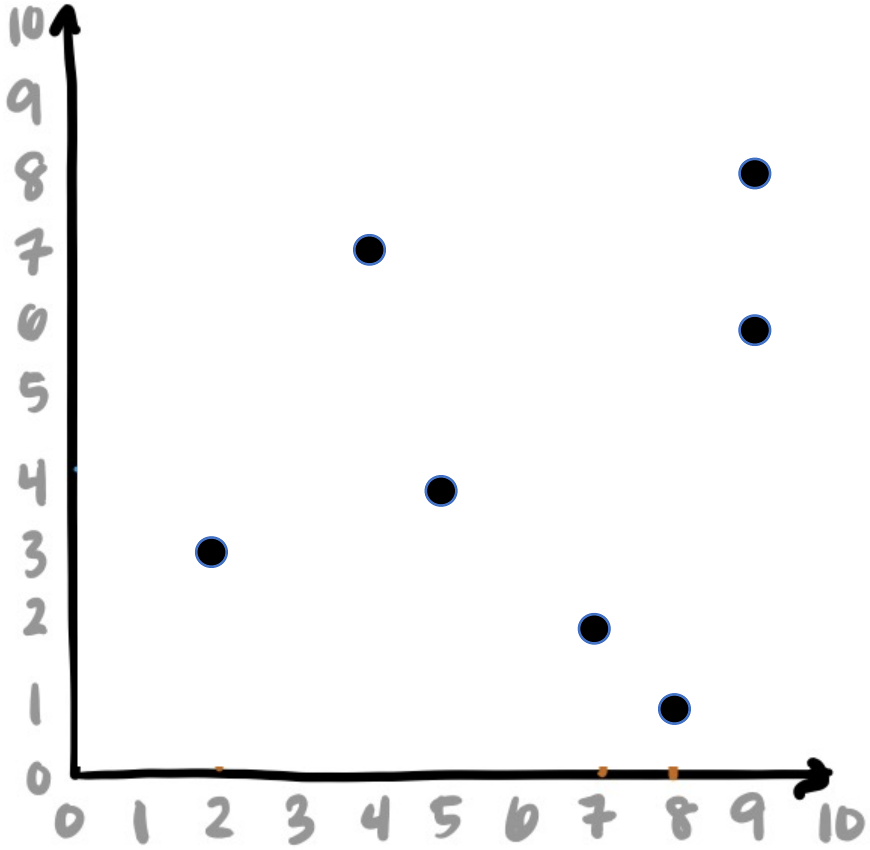
Range-based Searches



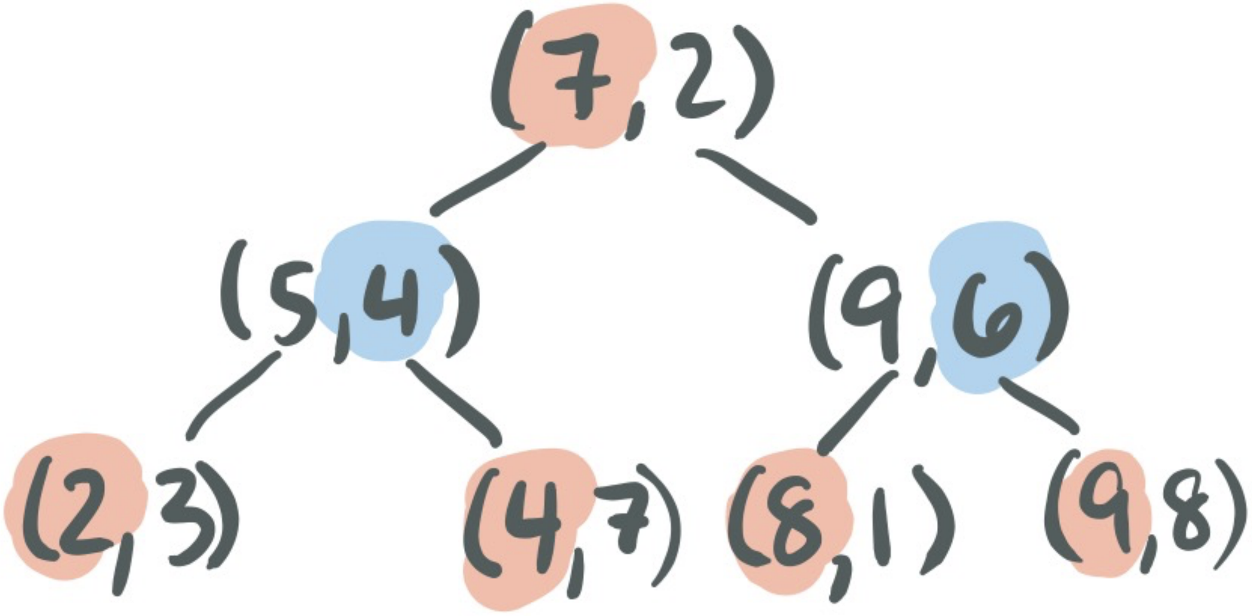
Nearest Neighbor: k-d tree

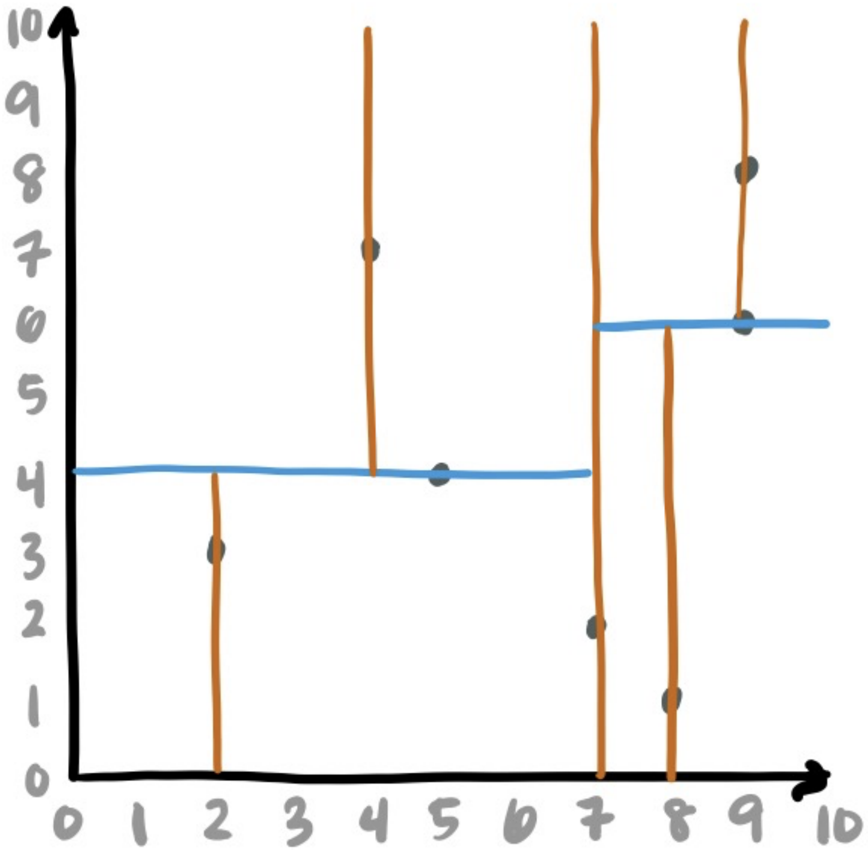
A **k-d tree** is similar but splits on points:

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



Nearest Neighbor: k-d tree





Nearest Neighbor: k-d tree

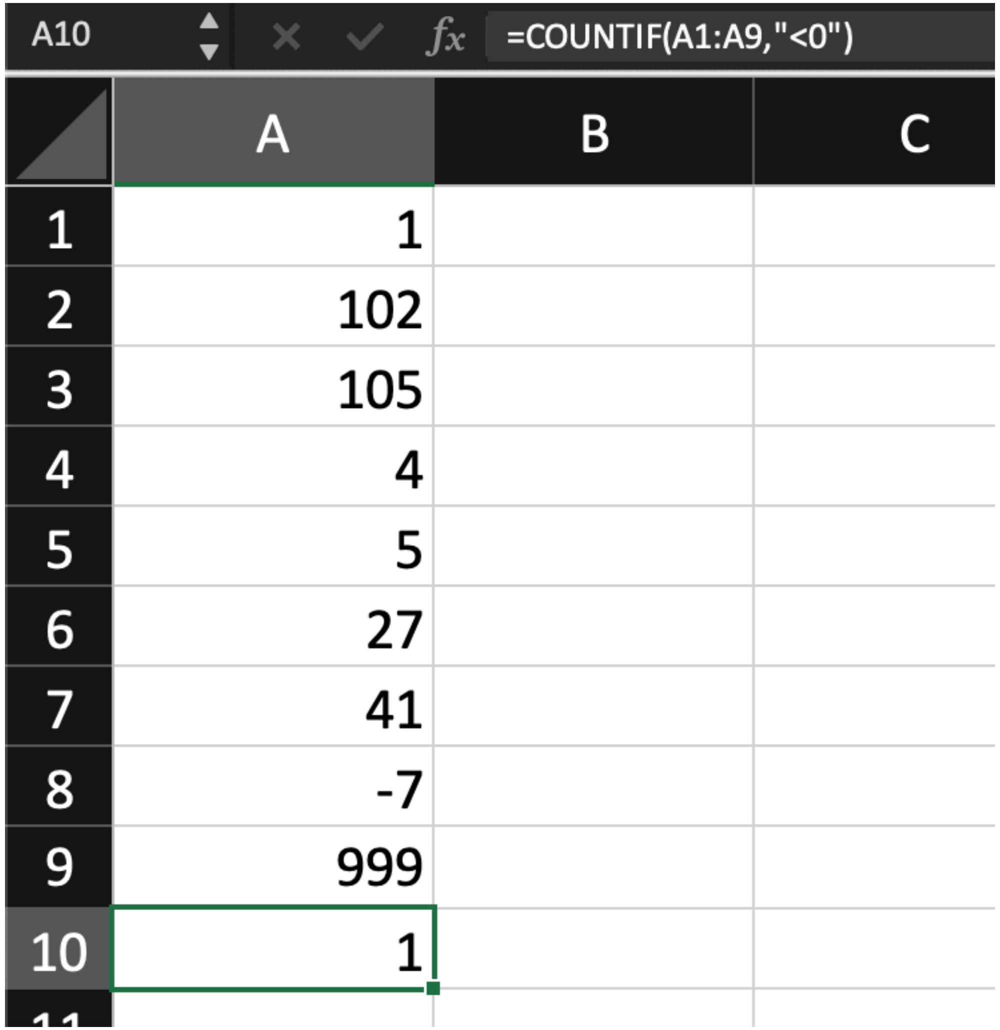
This construction seems easy conceptually but…

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

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

Functions as arguments

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



=COUNTIF(A1:A9,"<0")

	A	B	C
1	1		
2	102		
3	105		
4	4		
5	5		
6	27		
7	41		
8	-7		
9	999		
10	1		

Functions as arguments

Countif.hpp

```
template <typename Iter, typename Pred>
int Countif(Iter begin, Iter end, Pred pred) {
  int count = 0;
  auto cur = begin;

  while(cur != end) {
    if(pred(*cur))
      ++count;
    ++cur;
  }

  return count;
}
```

Lambda Functions in C++

main.cpp

```
bool isNegative(int num) { return (num < 0); }

class IsNegative {
public:
    bool operator() (int num) { return (num < 0); }
};

int main() {
  std::vector<int> numbers = {1, 102, 105, 4, 5, 27, 41, -7, 999};

  auto isnegl = [](int num) { return (num < 0); };
  auto isnegfp = isNegative;
  auto isnegfunctor = IsNegative();

  cout << "There are " << Countif(numbers.begin(), numbers.end(), _______)
    << " negative numbers" << std::endl;

```

Lambda Functions in C++

[ ]( ){ }

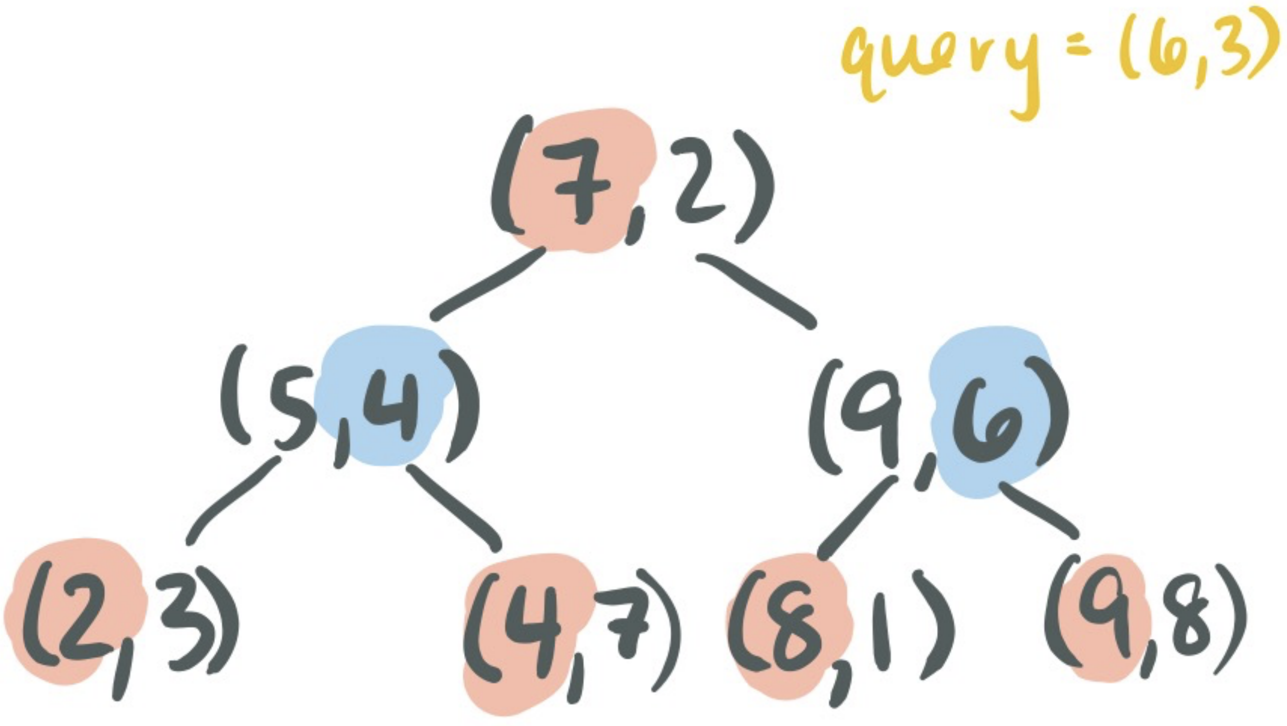
Lambda Functions in C++

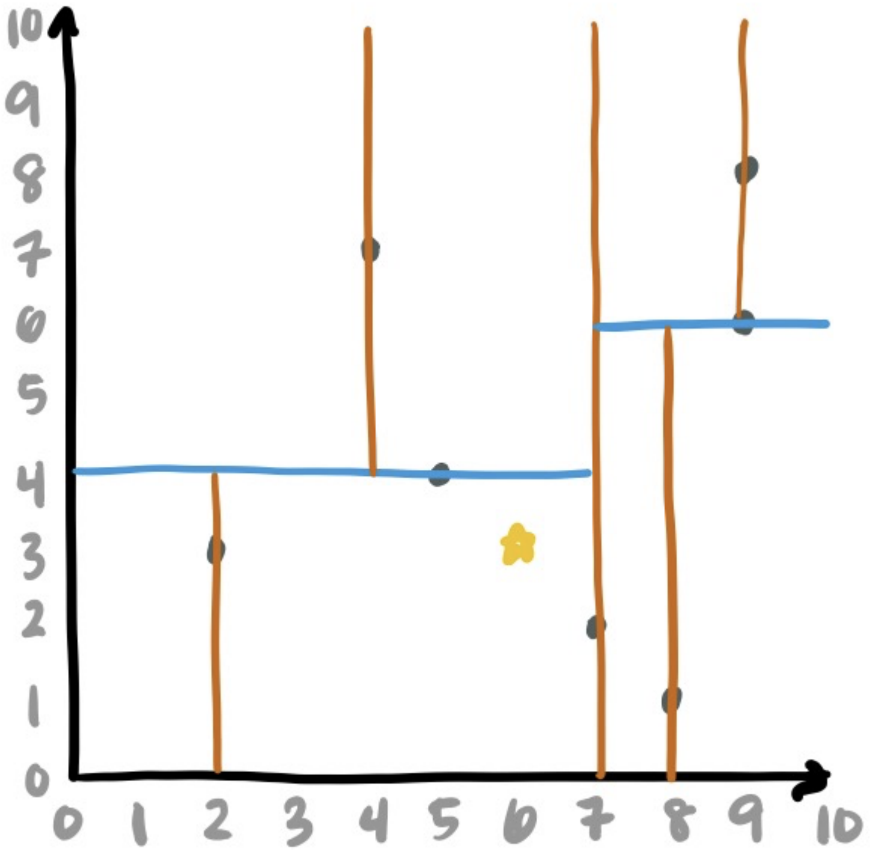
main.cpp

```
  int big;


  std::cout << "How big is big? ";
  std::cin >> big;

  auto isbig = [big](int num) { return (num >= big); };

  std::cout << "There are " << Countif(numbers.begin(), numbers.end(), isbig)
    << " big numbers" << std::endl;
}
```

Nearest Neighbor: k-d tree

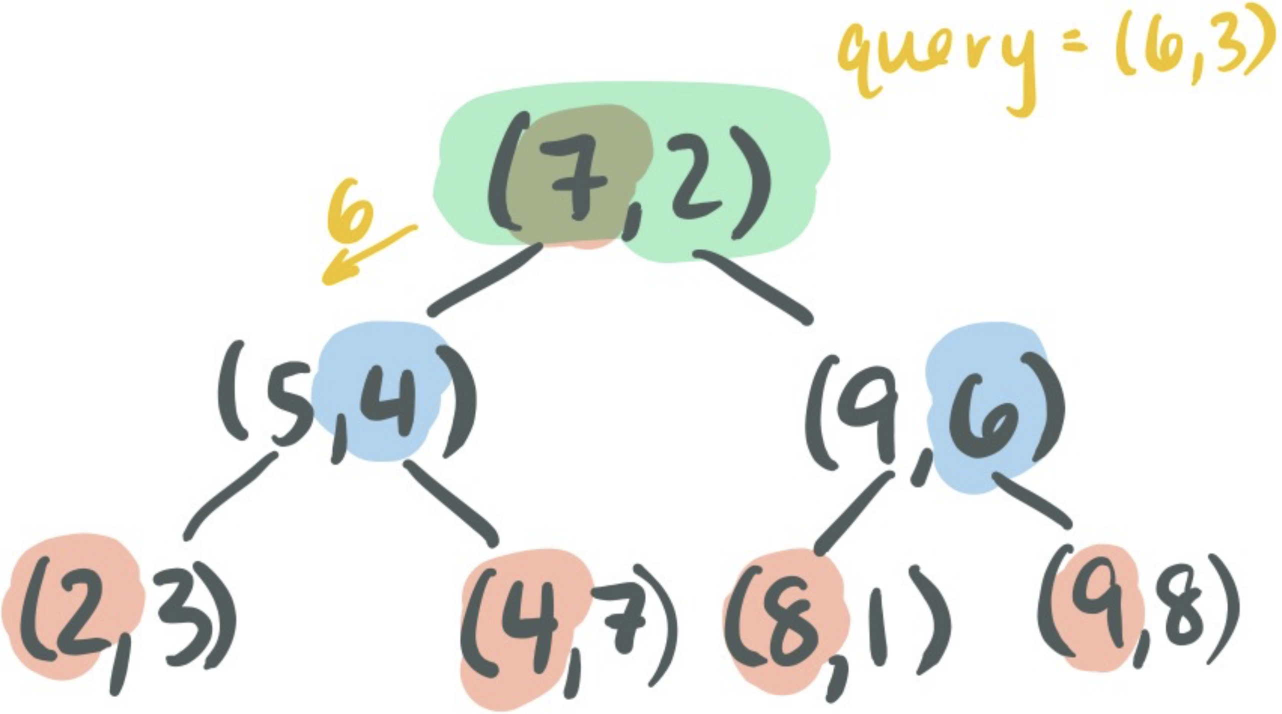
When querying a k-d tree, it acts like a BST* at first…

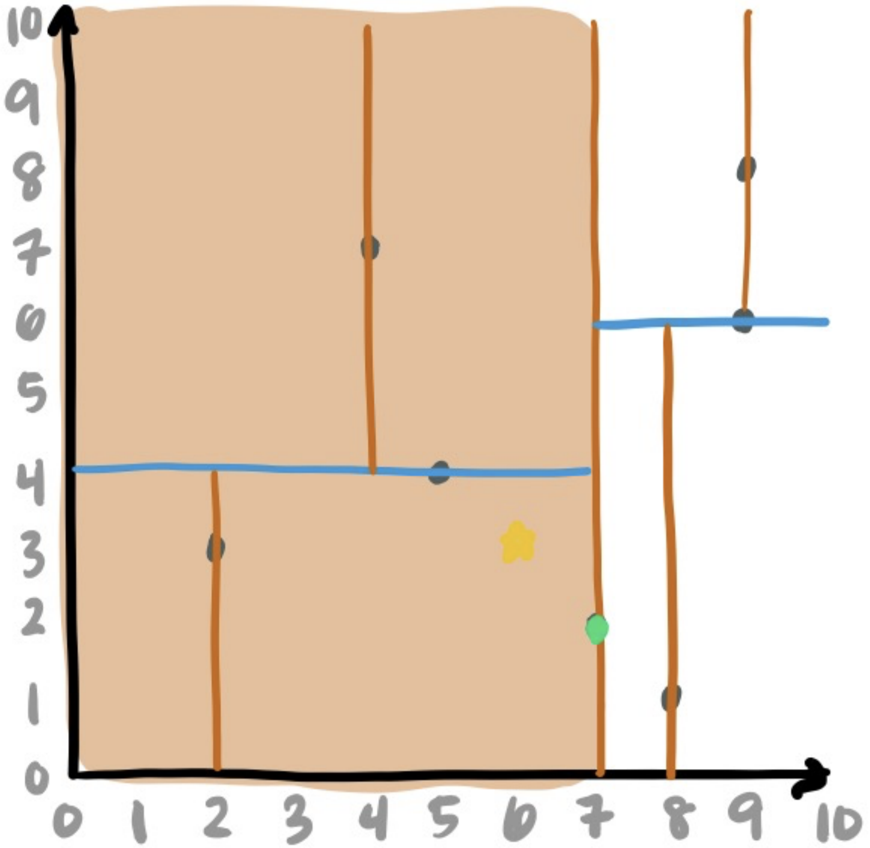




Nearest Neighbor: k-d tree

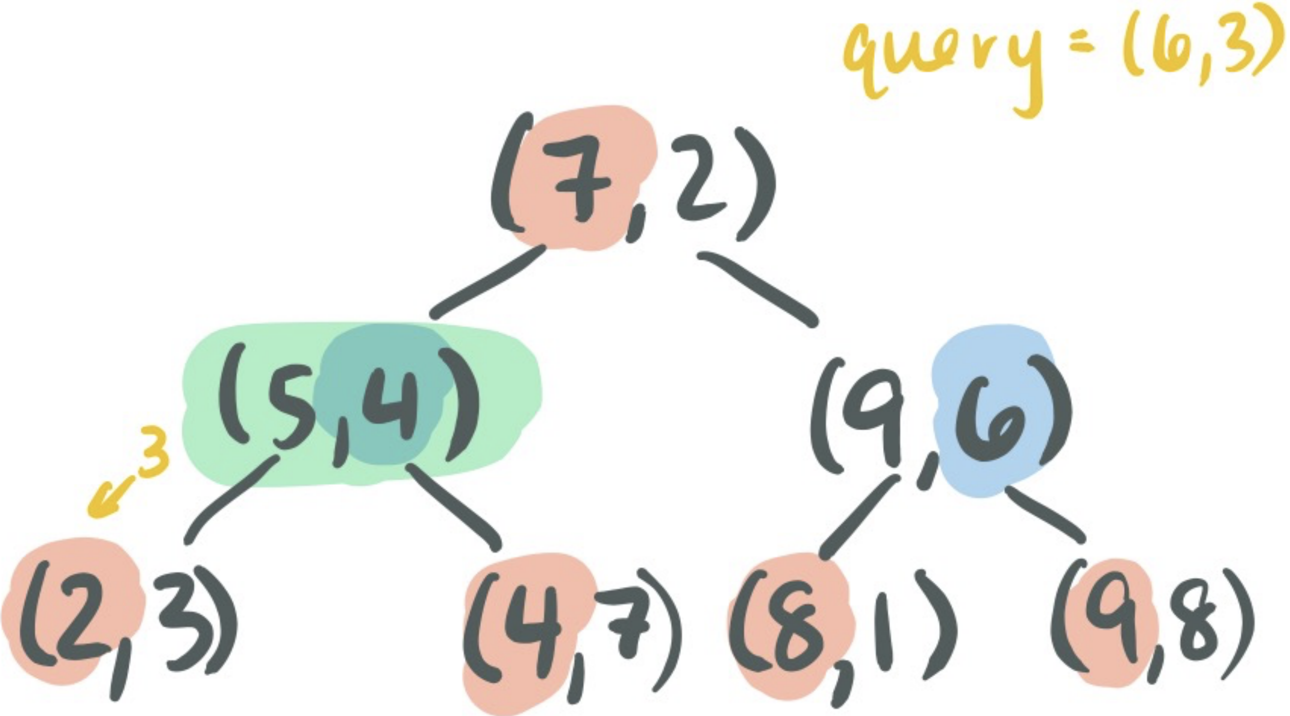
When querying a k-d tree, it acts like a BST* at first…

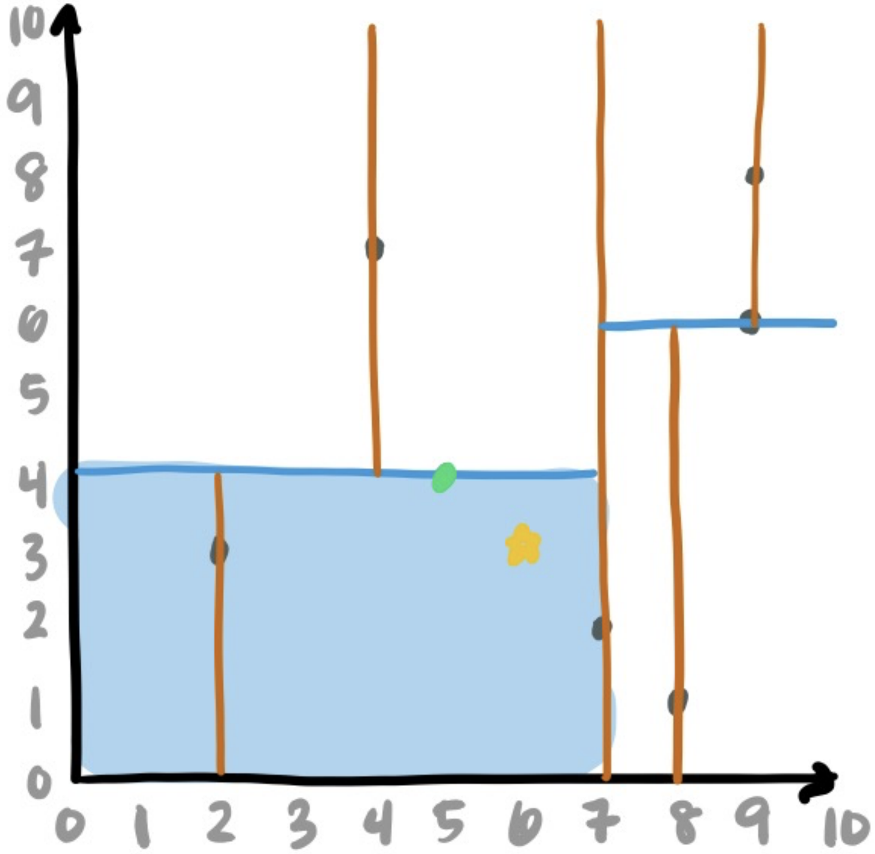




Nearest Neighbor: k-d tree

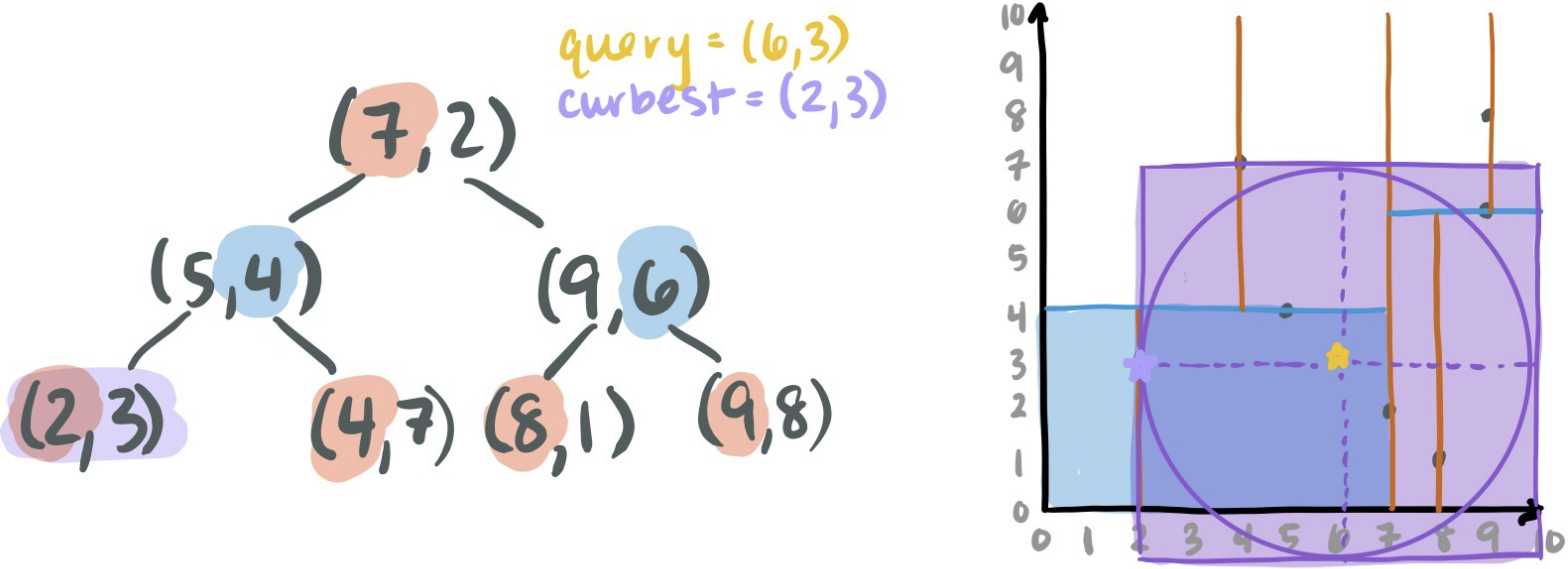
When querying a k-d tree, it acts like a BST* at first…





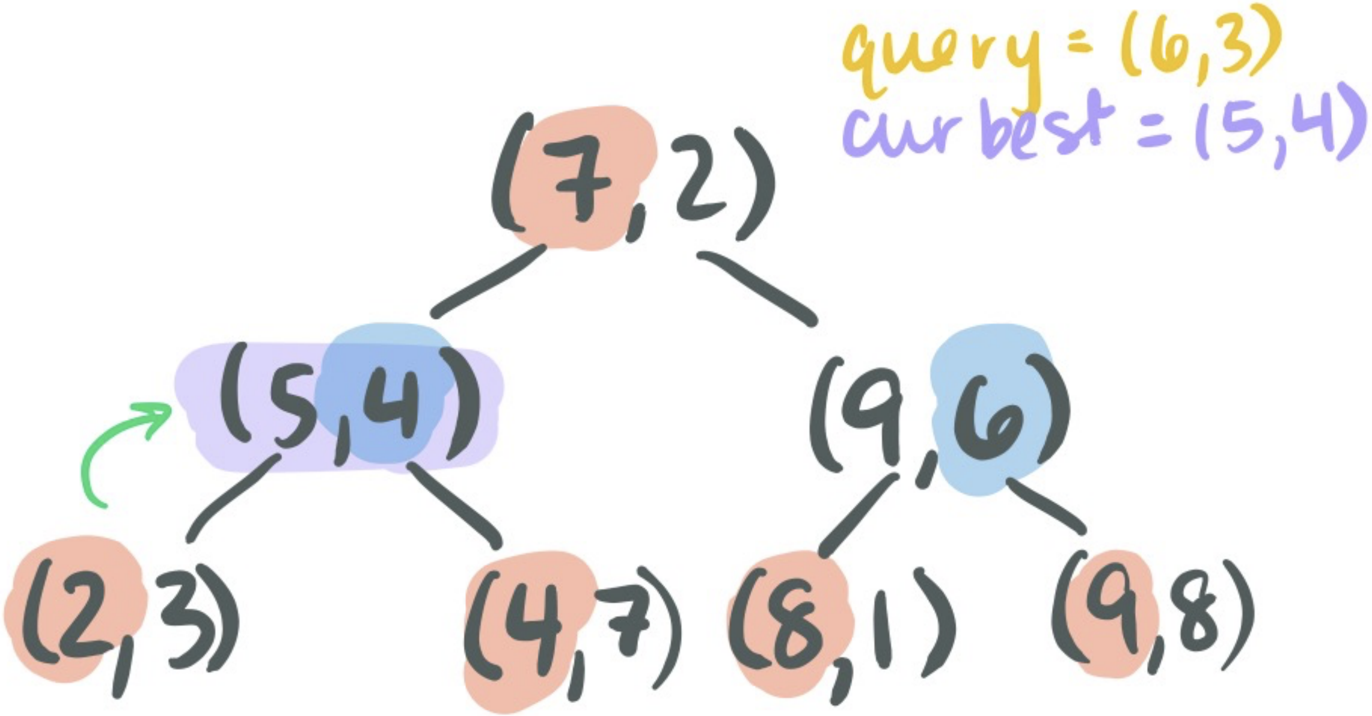
Nearest Neighbor: k-d tree

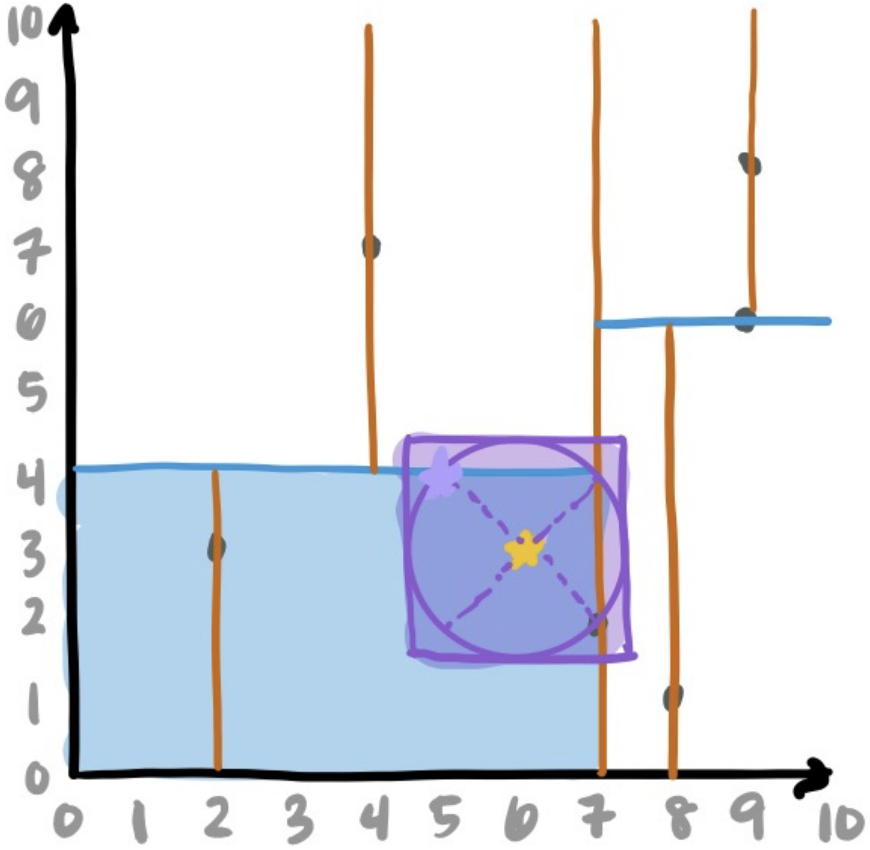
When querying a k-d tree, it acts like a BST* at first…



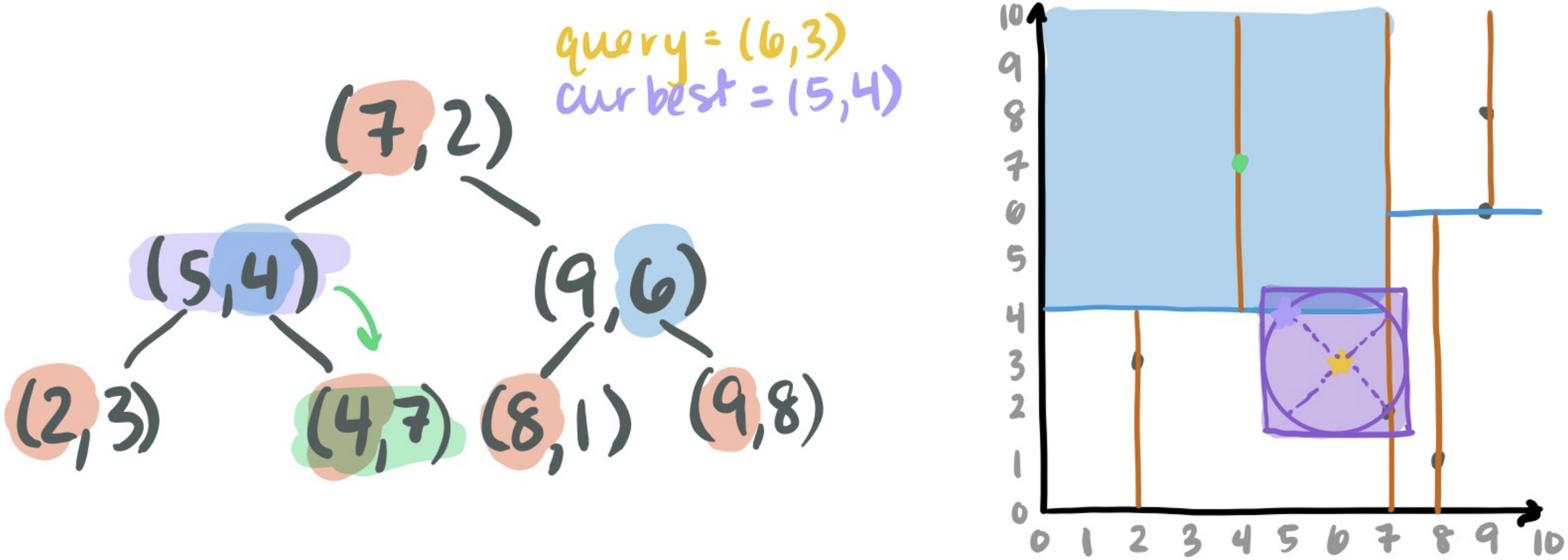
Nearest Neighbor: k-d tree

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



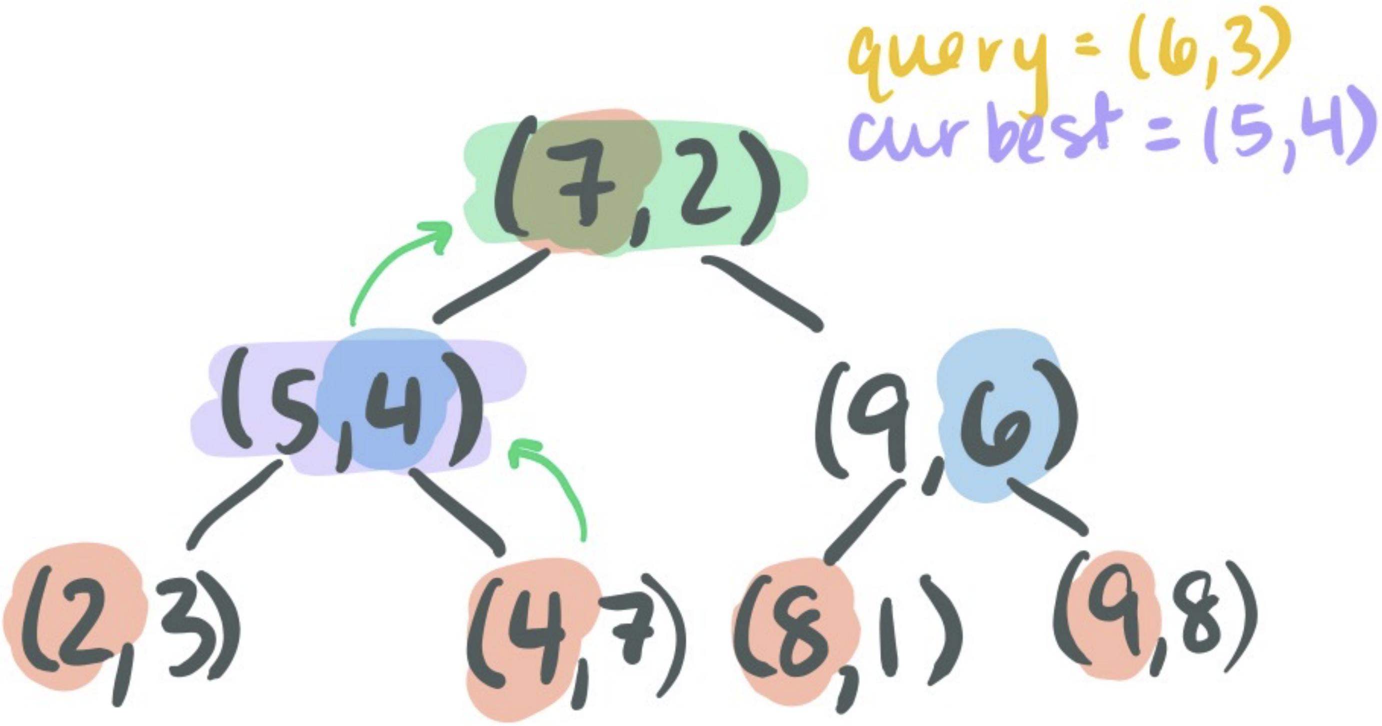


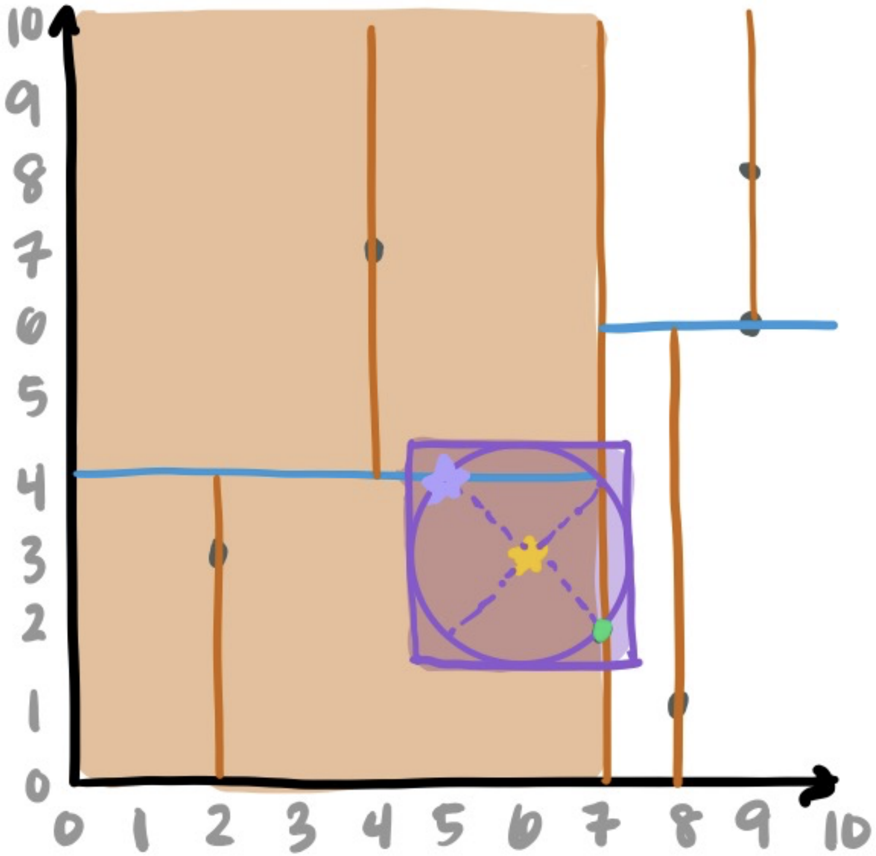
Nearest Neighbor: k-d tree



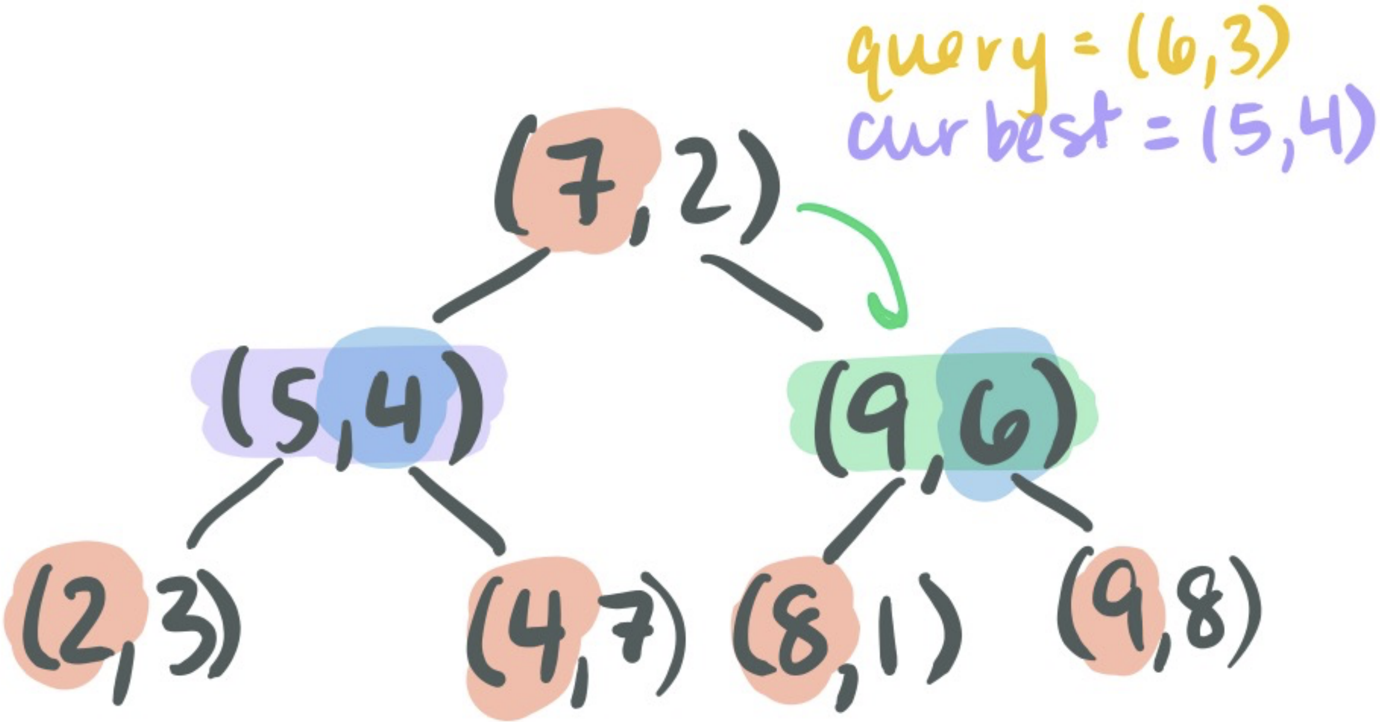
Nearest Neighbor: k-d tree

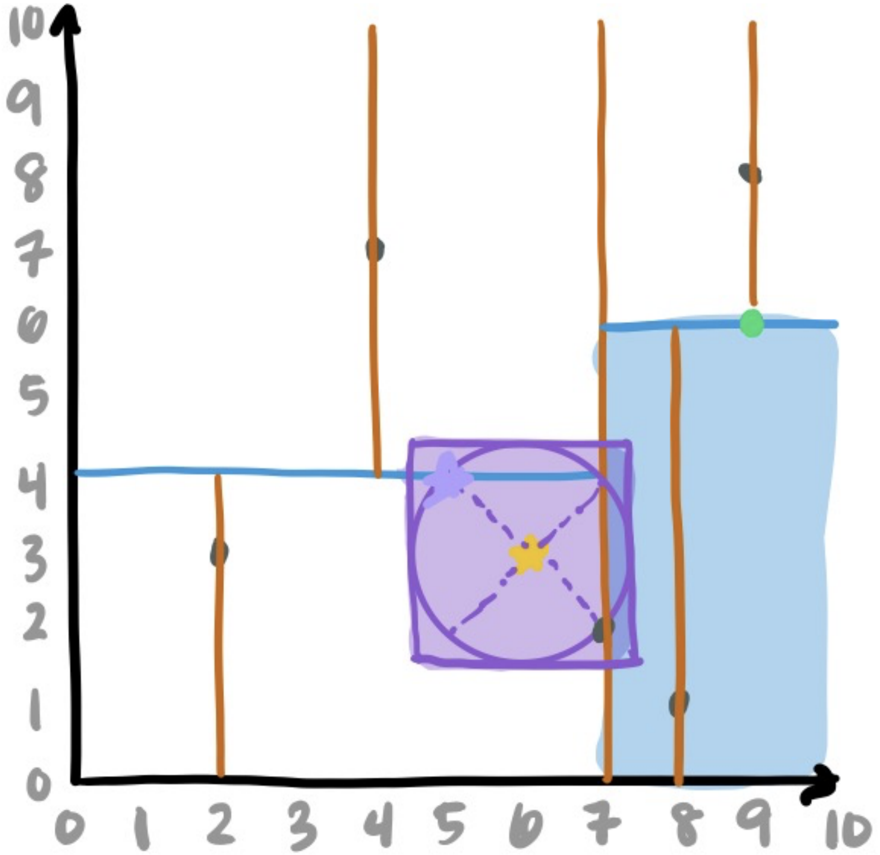
On ties, use `smallerDimVal` to determine which point remains `curBest`



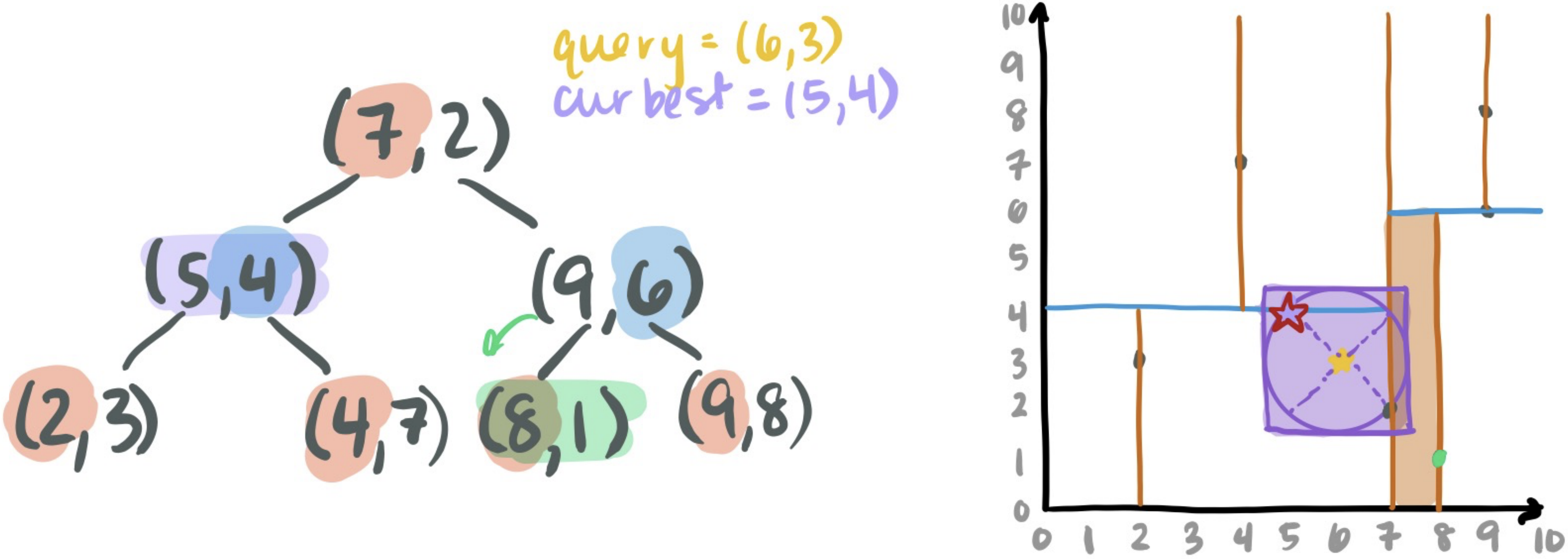


Nearest Neighbor: k-d tree





Nearest Neighbor: k-d tree



query = (6,3)

cur best = (5,4)

BEST: (5,4)

Nearest Neighbor: k-d tree

Final tips:

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

The suggestions in the writeup should be followed carefully

Plan of Action

Since our goal is to find the lower bound on **n** given **h**, we can begin by defining a function given **h** which describes the smallest number of nodes in an AVL tree of height **h**:

$N(h)$ = minimum number of nodes in an AVL tree of height h

Simplify the Recurrence

$$N(h) = 1 + N(h-1) + N(h-2)$$

State a Theorem

Theorem: An AVL tree of height h has at least ___________.

Proof by Induction:

I. Consider an AVL tree and let **h** denote its height.

II. Base Case: ________________

An AVL tree of height ____ has at least ____ nodes.

Prove a Theorem

III. Base Case: ________________

An AVL tree of height ____ has at least ____ nodes.

Prove a Theorem

IV. Induction Case: ________________

Assume for all heights $i < h$, $N(i) \geq 2^{i/2}$. Prove that $N(h) \geq 2^{h/2}$

Prove a Theorem

V. Using a proof by induction, we have shown that:

…and inverting:

AVL Runtime Proof

An upper-bound on the height of an AVL tree is **O(lg(n))**:

N(h) := Minimum # of nodes in an AVL tree of height h

$$N(h) = 1 + N(h-1) + N(h-2)$$

$$> 1 + 2^{(h-1)/2} + 2^{(h-2)/2}$$

$$> 2 \times 2^{(h-2)/2} = 2^{(h-2)/2+1} = 2^{h/2}$$

Theorem #1:

Every AVL tree of height h has at least $2^{h/2}$ nodes.

Summary of Balanced BST

AVL Trees

- Max height: 1.44 * lg(n)
- Rotations:

  Zero rotations on find
  One rotation on insert
  O(**h**) == O(**lg(n)**) rotations on remove

Red-Black Trees

- Max height: 2 * lg(n)
- Constant number of rotations on insert (max 2), remove (max 3).

Summary of Balanced BST

Pros:

- Running Time:
  - Improvement Over:
- Great for specific applications:

Summary of Balanced BST

Cons:

- Running Time:

- In-memory Requirement:

Next Week: 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)?