CPSC 4240/5240: Parallel Programming Techniques

Lecture 15: ParlayLib and kNN

Lecturer: Quanquan C. Liu

Spring 2026

Announcements

- Midterm 2: March 24
  - Covers **all lectures** up to Lecture 15 (today)
  - 2 sheets of notes (front and back)
- Homework 3 Posted
  - Due **Monday, March 30 11:59pm**
  - Late due date: **Thursday, April 2, 11:59pm**
- Final Project
  - Due **April 24**
  - 5-10 min final project presentation **April 21/23**

Final Project: Additional Details

- Final Project
  - Due **April 24**
  - 5-10 min final project presentation **April 21/23**
- Grade Breakdown for Project:
  - **Originality: 30%**
  - Execution: 30%
  - Written Report: 20%
  - Final Presentation: 20%

Final Project: Additional Details

- Final Project
  - Due **April 24**
  - 5-10 min final project presentation **April 21/23**
- Grade Breakdown for Project:
  - **Originality: 30% (for 524 students: half of this grade will consist of research potential)**
  - Execution: 30%
  - Written Report: 20%
  - Final Presentation: 20%

How to Use ParlayLib with OpenMP

Using ParlayLib with OpenMP

Only need to add the flag

-fopenmp

```
const size_t n = 100000000;
const unsigned seed = 42;

std::mt19937 gen(seed);
std::uniform_int_distribution<int> dist(0, 100000000);

// Use parlay sequence to create sequence of integers
parlay::sequence<int> data(n);

#pragma omp parallel for
for (size_t i = 0; i < n; i++) {
    data[i] = dist(gen);
}

parlay::internal::timer t("Time");

t.start();
merge_sort(data);
t.next("mergesort time");
```

ParlayLib Common Structures and Algorithms

Things like sequences and parallel_for

Containers

- **parlay::sequence<T>** is a flexible container that contain elements of type T
- All core primitives can operate on a sequence

```
parlay::sequence<double> seq(n);
```

```
parlay::sequence<std::string> words = { "banana",
"apple", "zebra", "apple", "mango", "pear" };
```

```
parlay::sequence<std::pair<int, float>> seq(n);
```

- **parlay::sequence<T>** can contain structs

Containers

- **parlay::sequence<T>** can contain structs

```
struct Person {
    std::string name;
    int age;
};

parlay::sequence<Person> people = {
    {"Alice", 30}, {"Bob", 25}, {"Eve", 42}, {"Charlie", 25}
};
```

Parallel_for

- Parlay has a parallel for loop

```
// Create a parlay sequence of n integers
parlay::sequence<int> data(n);

// Use parallel_for to update the sequence in parallel
parlay::parallel_for(0, n, [&](size_t i){
    data[i] = i;
});
```

Parallel_for

- Parlay has a parallel for loop

Define the index

```
// Create a parlay sequence of n integers
parlay::sequence<int> data(n);

// Use parallel_for to update the sequence in parallel
parlay::parallel_for(0, n, [&](size_t i){
    data[i] = i;
});
```

Parallel_for

- Parlay has a parallel for loop

Define the index; lambda function & captures by reference data (rather than a copy)

```
// Create a parlay sequence of n integers
parlay::sequence<int> data(n);

// Use parallel_for to update the sequence in parallel
parlay::parallel_for(0, n, [&](size_t i){
    data[i] = i;
});
```

Parallel_for

- Parlay has a parallel for loop

Start and end indices (as in a sequential for loop)

```
// Create a parlay sequence of n integers
parlay::sequence<int> data(n);

// Use parallel_for to update the sequence in parallel
parlay::parallel_for(0, n, [&](size_t i){
    data[i] = i;
});
```

Parallel Reduce

- Parlay has a **parlay::reduce** function that performs a reduction: sum, min, max, etc. over a range in a sequence in parallel
- Associative and commutative operations

```
parlay::sequence<int> data(n);

int total_sum = parlay::reduce(
    data,
    parlay::addm<int>()
);
```

Parallel Reduce

- Parlay has a **parlay::reduce** function that performs a reduction: sum, min, max, etc. over a range in a sequence in parallel
- Associative and commutative operations

```
parlay::sequence<int> data(n);

int total_sum = parlay::reduce(
    data,
    parlay::minm<int>()
);
```

Parallel Reduce

- Parlay has a **parlay::reduce** function that performs a reduction: sum, min, max, etc. over a range in a sequence in parallel
- Associative and commutative operations

```
parlay::sequence<int> data(n);

int total_sum = parlay::reduce(
    data,
    parlay::maxm<int>()
);
```

Parallel Map

- Parlay has a **parlay::map** function that **performs a unary function on each element of a sequence** and returns a new sequence with the transformed values

Parallel Map

- Parlay has a **parlay::map** function that **performs a unary function on each element of a sequence** and returns a new sequence with the transformed values

```
parlay::sequence<int> data = {1, 2, 3, 4, 5};

auto squares = parlay::map(data, [](int x){
    return x * x;
});
```

Parallel Map

- Parlay has a **parlay::map** function that **performs a unary function on each element of a sequence** and returns a new sequence with the transformed values

```
parlay::sequence<int> data = {1, 2, 3, 4, 5};

auto squares = parlay::map(data, [](int x){
    return x * x;
});
```

Takes a **lambda function**

Parallel Map

- Parlay has a **parlay::map** function that **performs a unary function on each element of a sequence** and returns a new sequence with the transformed values

```
parlay::sequence<int> data = {1, 2, 3, 4, 5};

auto squares = parlay::map(data, [](int x){
    return x * x;
});
```

Outputs a sequence

Parallel Map

- Parlay has a **parlay::map** function that **performs a unary function on each element of a sequence** and returns a new sequence with the transformed values

```
parlay::sequence<int> data = {1, 2, 3, 4, 5};

auto str_seq = parlay::map(data, [](int x){
        return std::to_string(x);
});
```

Parallel Map

Can use custom structs in the method

```
struct Person {
        std::string name;
        int age;
};

parlay::sequence<int> data = {1, 2, 3, 4, 5};

auto people = parlay::map(data, [&](int age) {
        Person p;
        p.name = "Person_" + std::to_string(age);
        p.age = age;
        return p;
});
```

Parallel Map

Note: this is known as **data-parallelism**

```
struct Person {
        std::string name;
        int age;
};

parlay::sequence<int> data = {1, 2, 3, 4, 5};

auto people = parlay::map(data, [&](int age) {
        Person p;
        p.name = "Person_" + std::to_string(age);
        p.age = age;
        return p;
});
```

Parallel Scan

- Prefix-sum implementation in **parlay::scan**
- **Function returns a pair:**
  - First element is a **sequence of first n-1 sums**
  - Next element is **"total" over all elements in the sequence**

```
parlay::sequence<int> data = {1, 2, 3, 4, 5};

auto [prefix_sums, total_sum] = parlay::scan(data, parlay::addm<int>());
```

Parallel Scan

- Prefix-sum implementation in **parlay::scan**
- **Function returns a pair:**
  - First element is a **sequence of first n-1 sums**
  - Next element is **"total" over all elements in the sequence**

```
parlay::sequence<int> data = {1, 2, 3, 4, 5};

auto [prefix_sums, total_sum] = parlay::scan(data, parlay::addm<int>());
```

Parallel Filter

- Create a new sequence containing elements for which a predicate is true

```
parlay::sequence<int> data = {1, 2, 3, 4, 5};

auto evens = parlay::filter(data, [](int x) {
        return (x % 2) == 0;
});
```

Parallel Filter

- Create a new sequence containing elements for which a predicate is true

```
parlay::sequence<std::string> words = {
      "apple", "hi", "computer", "at", "umbrella", "sea", "map"
 };

auto long_words = parlay::filter(words, [](const std::string& w) {
      return w.size() > 3;
});
```

Parallel Pack

- Similar to parallel filter operation that **takes a Boolean "flag" sequence** indicating which elements to keep

```
parlay::sequence<int> data = {1, 2, 3, 4, 5};

auto is_even = parlay::map(data, [](int x){
        return (x % 2) == 0;
});

auto evens = parlay::pack(data, is_even);
```

Problem Solving Examples

Now let’s take what we learned and do some interactive problem solving!

Motif Finding in DNA

- Suppose you have a long string of DNA, consisting of the letters A, G, T, C
- You want to find all occurrences of a motif e.g. “ACT” in this string
- How do you do this in parallel?

Example

Input: ATCG**ACT**CG**ACT**GC

Output (indices of the first letter): [**4**, **9**]

Motif Finding in DNA

- Suppose you have a long string of DNA, consisting of the letters A, G, T, C
- You want to find all occurrences of a motif e.g. “ACT” in this string
- How do you do this in parallel?

<u>Idea:</u>

Parallel for loop; check every location

Motif Finding in DNA

What is the work?

- Suppose you have a long string of DNA, consisting of the letters A, G, T, C
- You want to find all occurrences of a motif e.g. “ACT” in this string
- How do you do this in parallel?

Idea:

Parallel for loop; check every location

Motif Finding in DNA

What is the work?
$O(n \cdot k)$

- Suppose you have a long string of DNA, consisting of the letters A, G, T, C
- You want to find all occurrences of a motif **(of length k)** e.g. "ACT" in this string
- How do you do this in parallel?

Idea:

Parallel for loop; check every location

Motif Finding in DNA

What is the work?
$O(n \cdot k)$

- Suppose you have a long string of DNA, consisting of the letters A, G, T, C
- You want to find all occurrences of a motif **(of length k)** e.g. "ACT" in this string
- How do you do this in parallel?

Idea:

Parallel for loop; check every location

Let's implement using both OpenMP and Parlaylib

Motif Finding in DNA

What is the work?
$O(n \cdot k)$

- Many different answers!
- Use parlay::map to map each letter in the input sequence to its index
  - Call this the **index array**
- Use parlay::map for each letter in the input sequence
  - Return 1 if next $k-1$ letters and current letter matches motif
  - Return -1 if it does not
  - Call this the **Boolean array**
- Use parlay::pack using the index array and the Boolean array

Motif Finding in DNA

What is the work?
$O(n \cdot k)$

Span: $O(k + \log n)$

- Many different answers!
- Use parlay::map to map each letter in the input sequence to its index
  - Call this the **index array**
- Use parlay::map for each letter in the input sequence
  - Return 1 if next $k - 1$ letters and current letter matches motif
  - Return -1 if it does not
  - Call this the **Boolean array**
- Use parlay::pack using the index array and the Boolean array

Motif Finding in DNA

What is the work?
$O(n \cdot k)$

Span: $O(k + \log n)$

- How do we decrease the work for large k?

- Potential solution:
  - Sample a large enough number of indices to check
  - Get a high probability bound that you correctly found all of the motifs

Motif Finding in DNA

What is the work?

$$\boldsymbol{O(n \cdot \log n)}$$

Span: $\boldsymbol{O(k + \log n)}$

- How do we decrease the work for large k?

- Potential solution:
  - Sample a large enough number of indices to check
  - Get a high probability bound that you correctly found all of the motifs

Motif Finding in DNA

What is the work?

$$\boldsymbol{O(n \cdot \log n)}$$

Span: $\boldsymbol{O(k + \log n)}$

- How do we decrease the work for large k?

- Potential solution:
  - Sample a large enough number of indices to check
  - Get a high probability bound that you correctly found all of the motifs
- More advanced solutions:
  - Parallel KMP (Knuth-Morris-Pratt) and parallel string hashing solutions (Rabin-Karp rolling hash)

Parallel kNN (k-Nearest Neighbors Classification)

Parallel kNN

- Given:
  - A training set of N data points, each is associated with:
    - A feature vector $x_i \in R^d$
    - A class label $y_i \in \{1, 2, \ldots, C\}$
  - A test set M of query points $q_j \in R^d$ for which we want to predict a label
- Goal:
  - For each q_j find the k training points nearest q_j
  - Take the majority vote label as predicted label (tie-break arbitrarily)

Parallel kNN

- Solutions Ideas?
- Naïve:
  - Compute for each query, in parallel, distance to all training points
  - Sort the distances in parallel for each query
  - Take majority of the distances for each query
- Work and span?
  - Work is $O(M \cdot N)$, very large!
  - Span could be $O\left(\log\left(N + M\right)\right)$ if implemented correctly

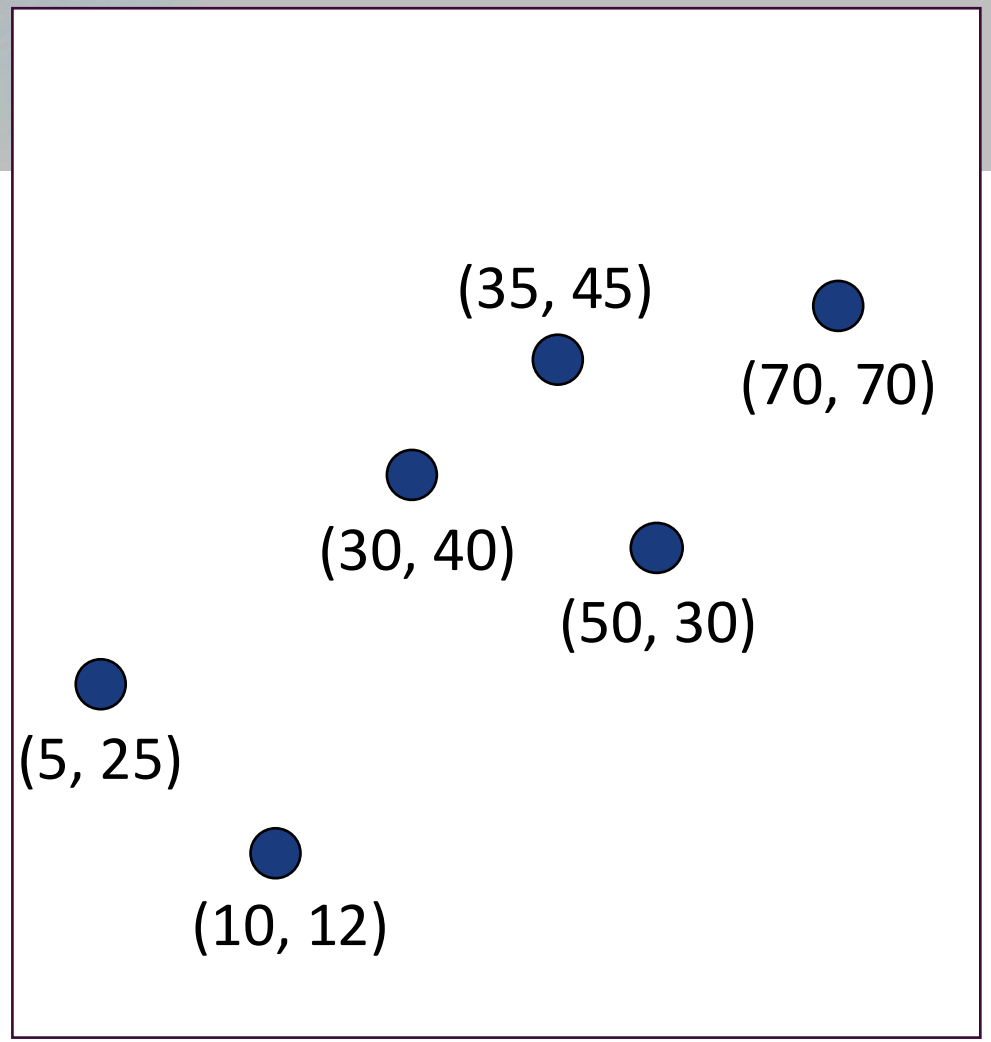
Parallel kNN: kd-tree

- Space partitioning data structure for organizing points in a d-dimensional space
- Every **leaf node** is a d-dimensional point
- Each **non-leaf node** is a splitting hyperplane along one of the dimensions
- Each level associated with a dimension
  - Choose a **pivot x** if value smaller than x, go to left subtree
  - If value greater than x, go to right subtree

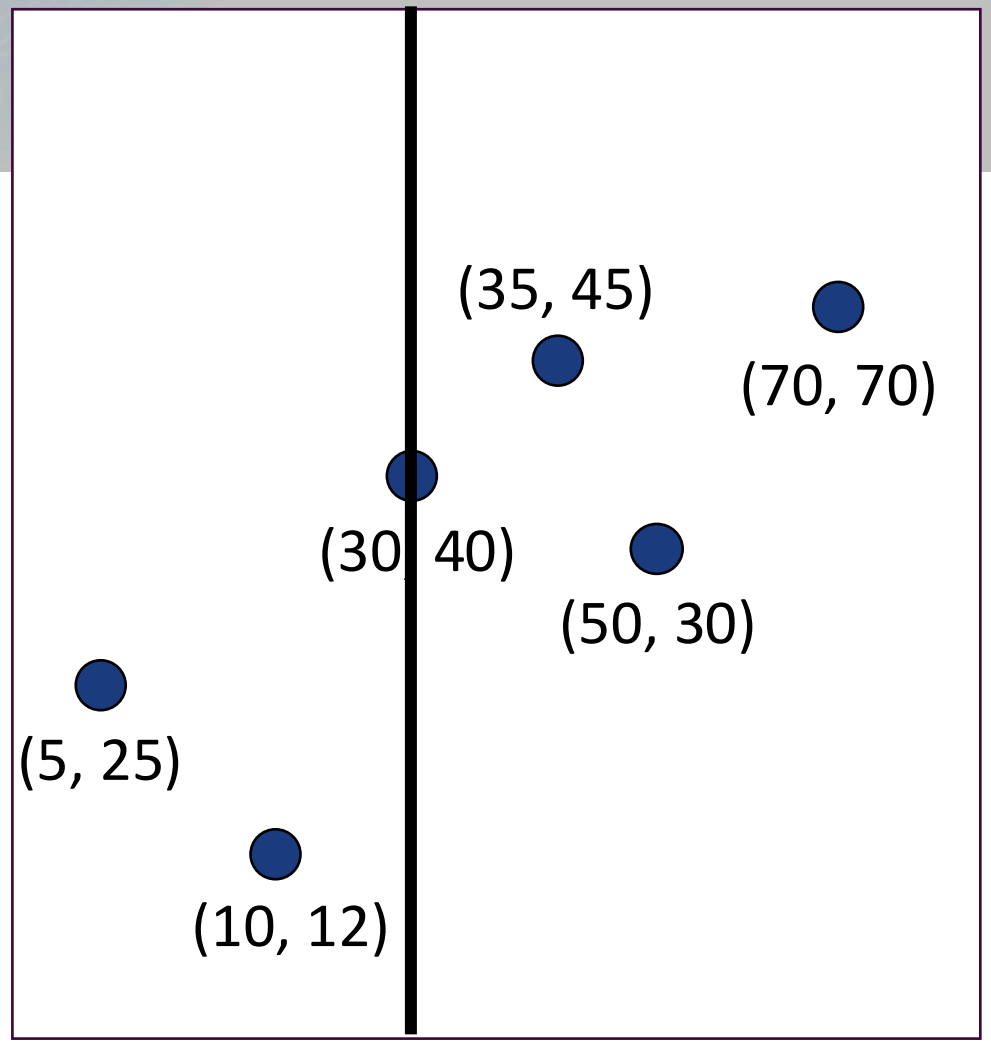
kd-tree Construction

- Given a set of points in d-dimensional space:
  - Cycle through the dimensions
    - E.g. in a 3-dimensional tree
      - First level uses pivot in x dimension
      - Second level uses pivot in y dimension
      - Third level uses pivot in z dimension
      - Fourth level uses pivot in x dimension…etc.
  - Pivot chosen as **median** of dimension of points in the subtree

2D-tree Example



2D-tree Example



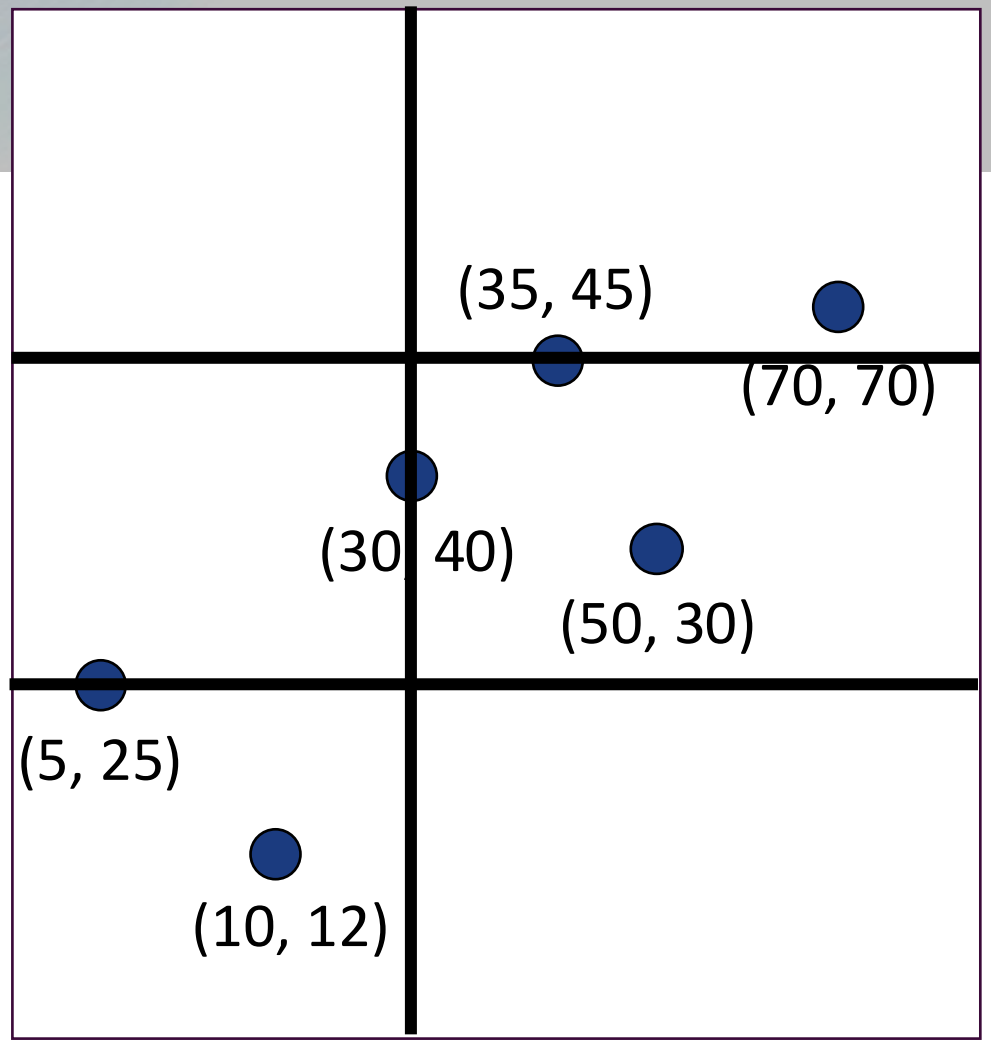
x-axis

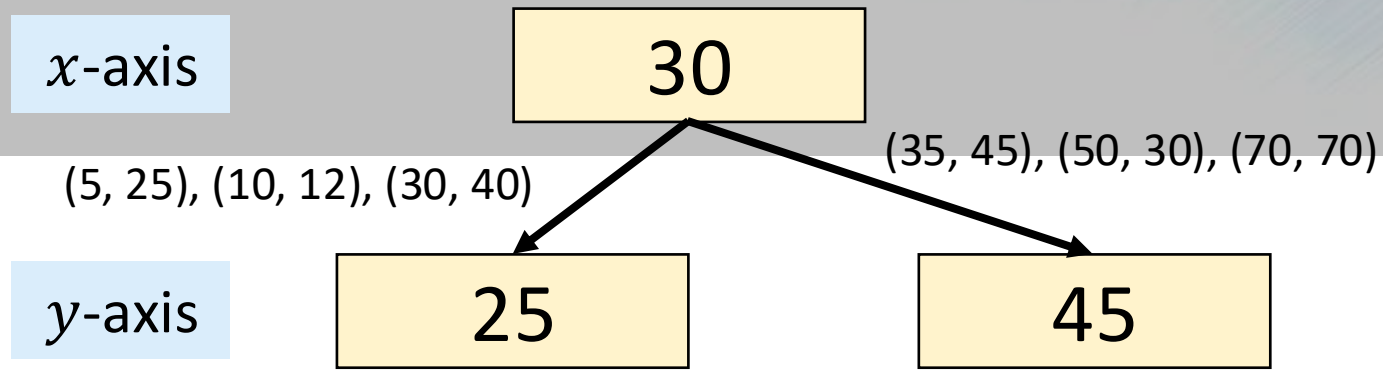
30

(5, 25), (10, 12), (30, 40)

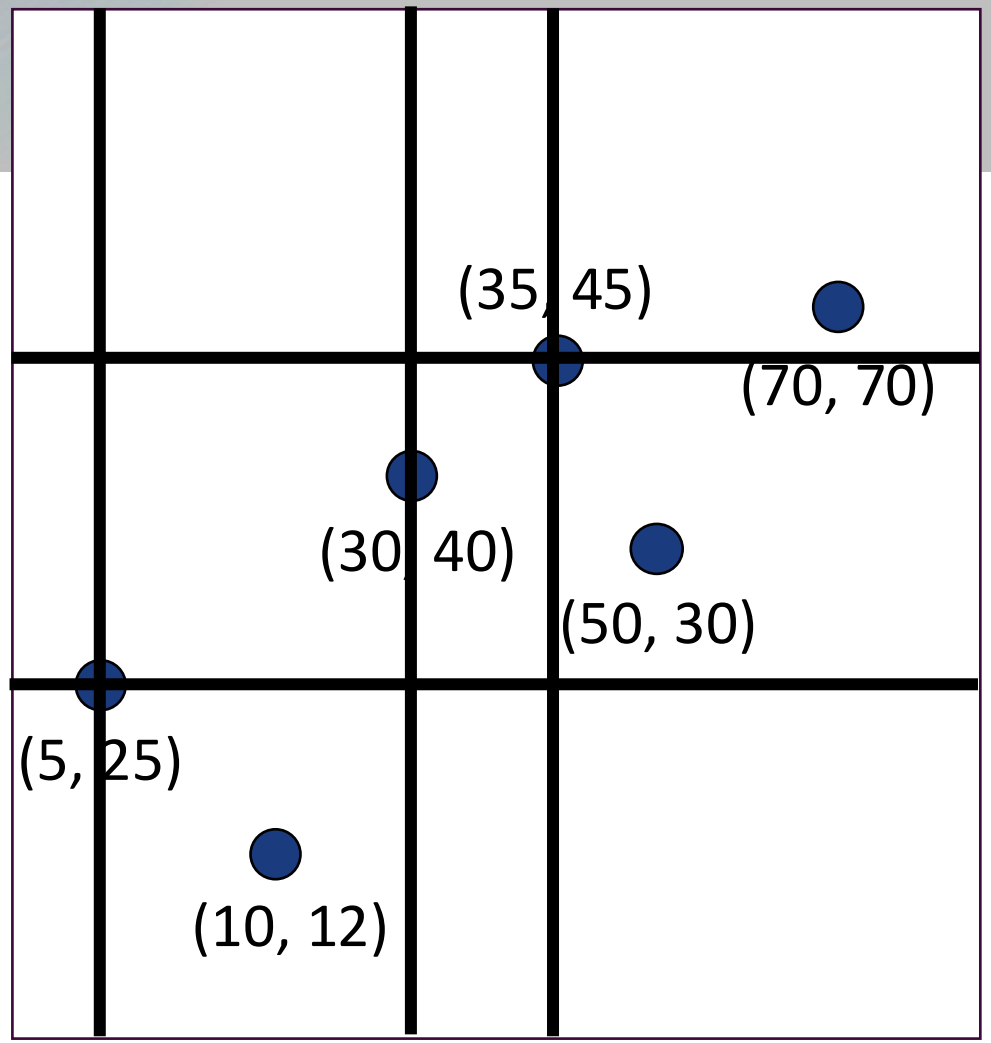
(35, 45), (50, 30), (70, 70)

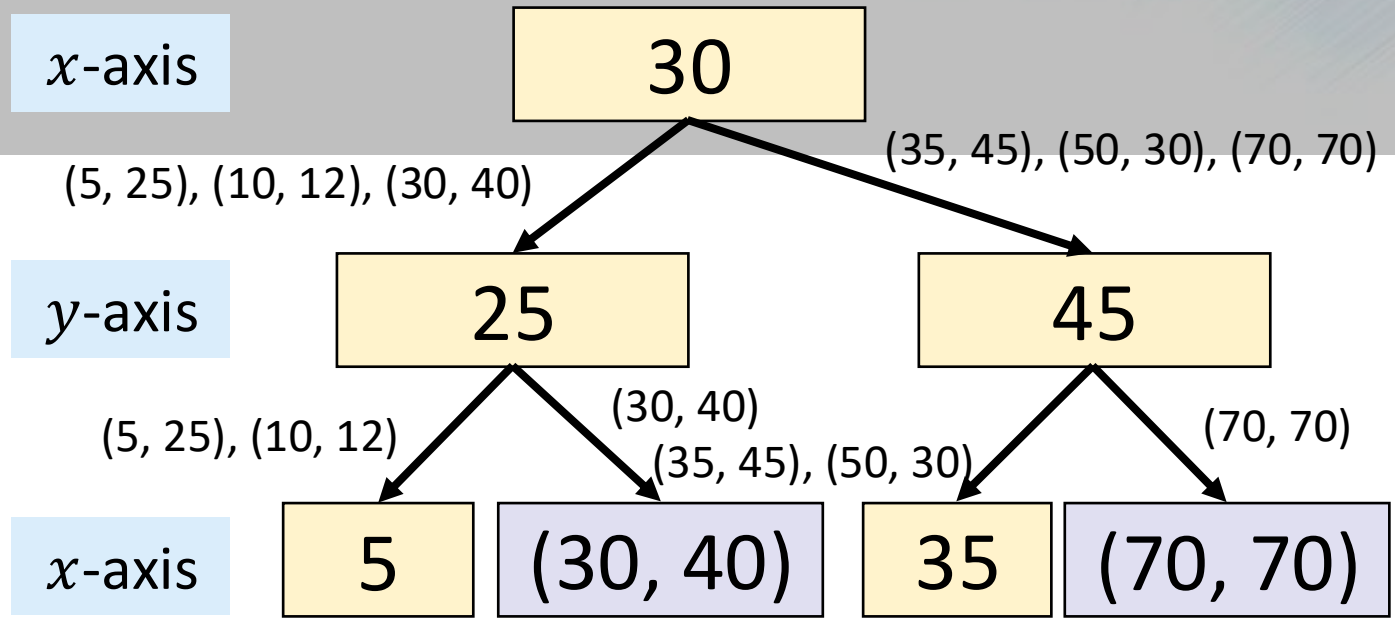
2D-tree Example



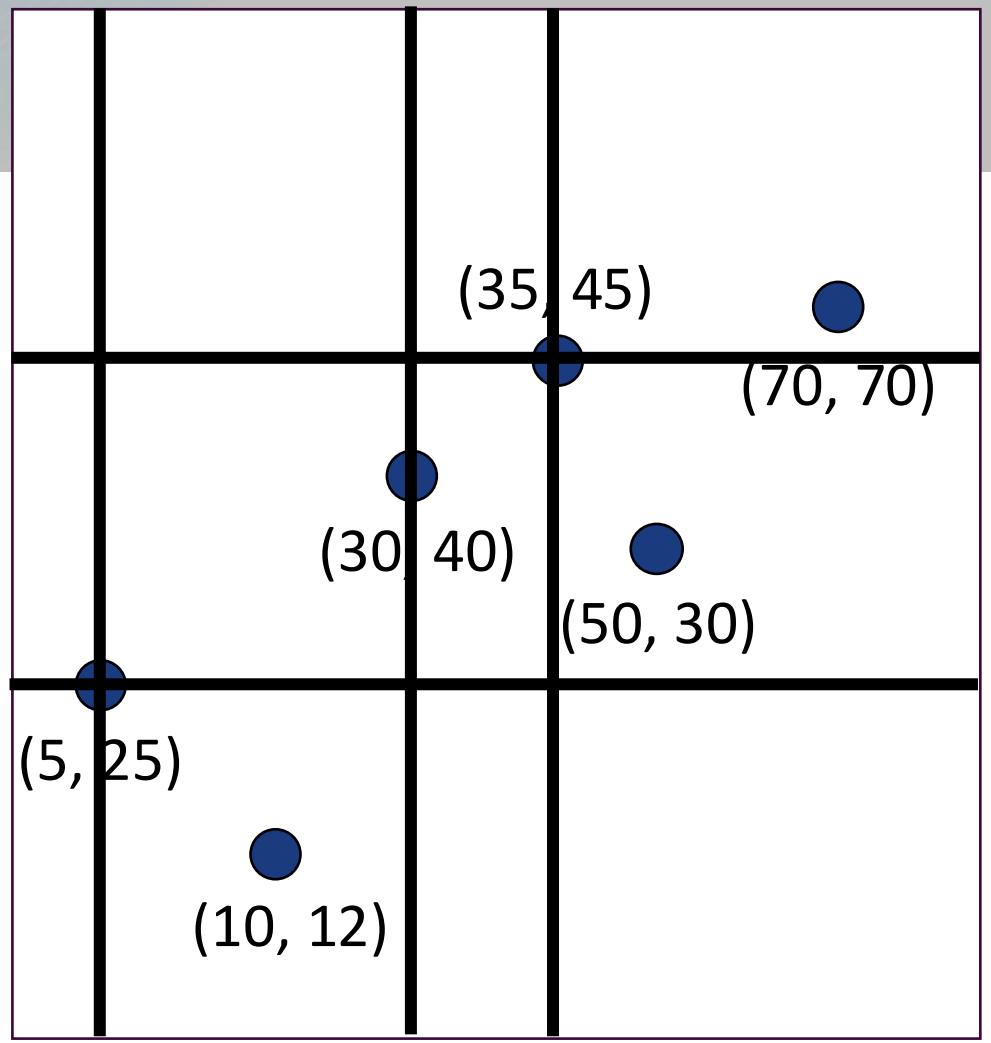


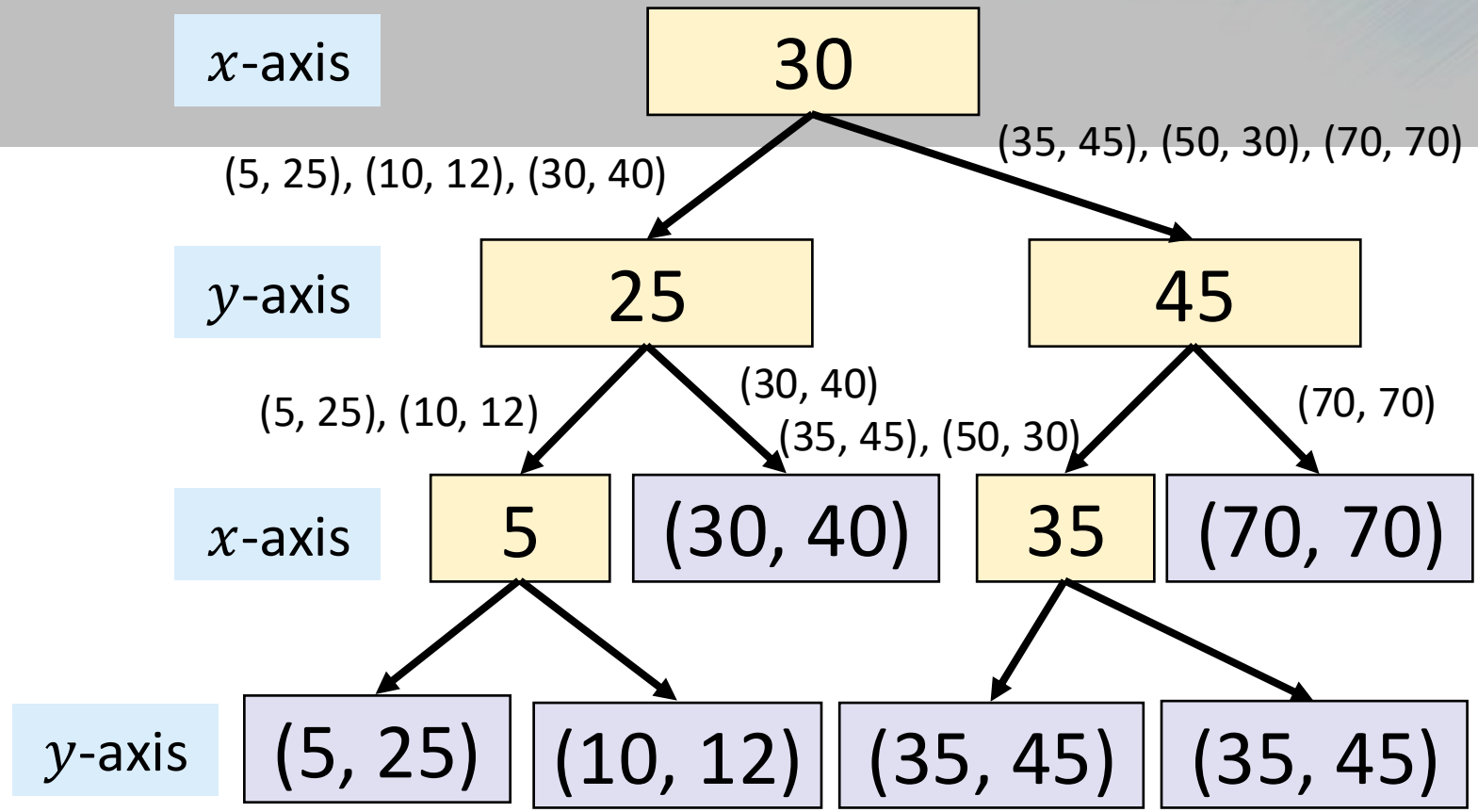
2D-tree Example





2D-tree Example



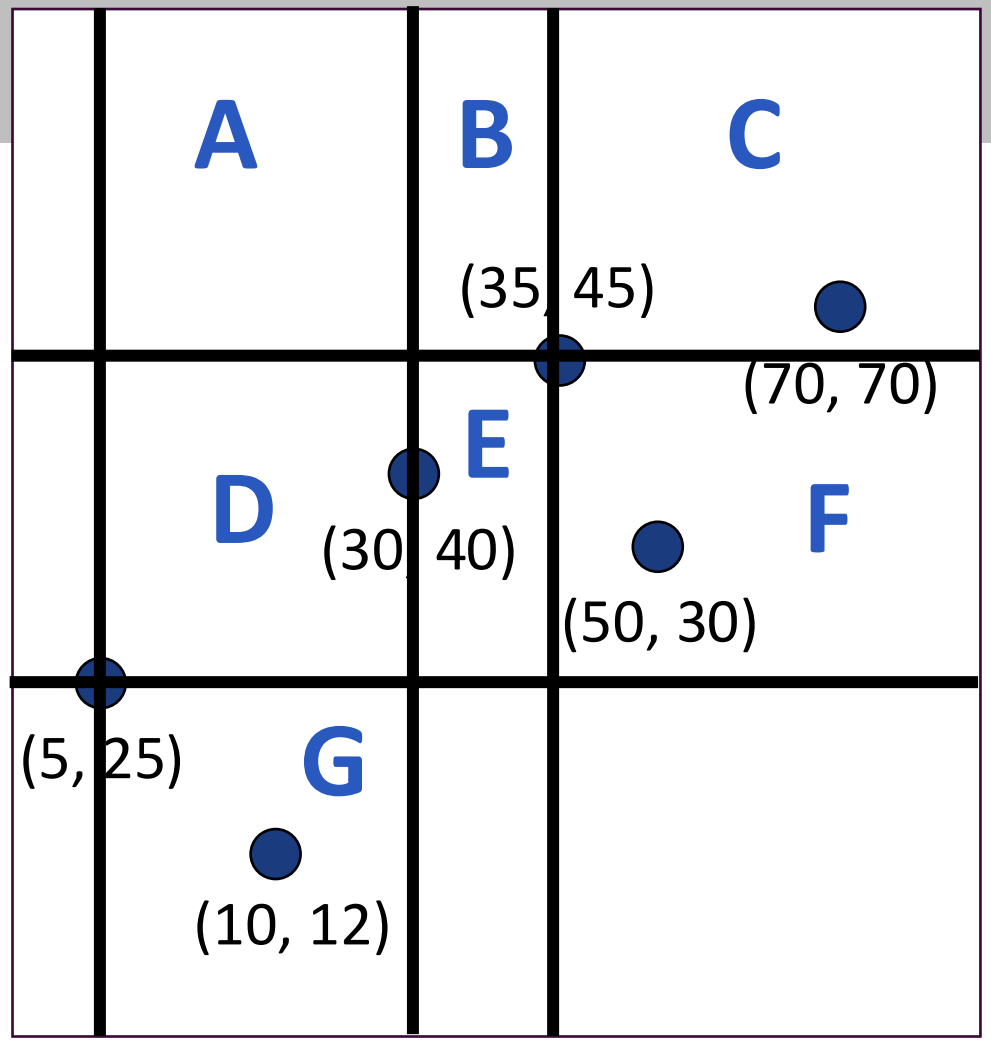


kd-Trees and Nearest Neighbor Search

- First, **search for the position of the query point** in your kd-tree
- **Go left or right** (just like a binary search):
  - If query's coordinate is less than or equal to node's coordinate, visit left child first
  - Otherwise, visit right child first
- Done in a **DFS fashion (recursively)!**
- Keep track of best neighbors found so far in max-heap
- Check the other side of split if necessarily using **bounding box**

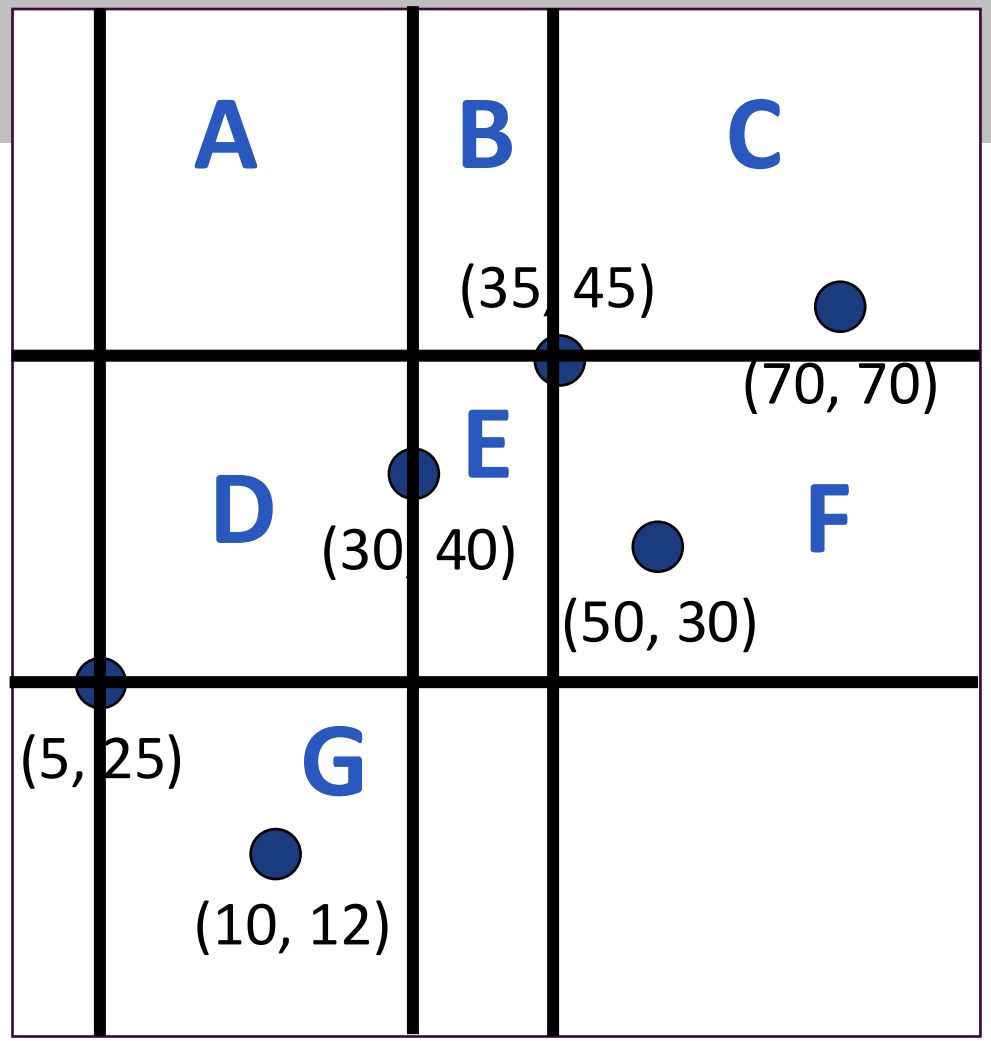
kd-Trees Bounding Box

Suppose you have query point that lies in box E, which bounding boxes could contain points close to it?



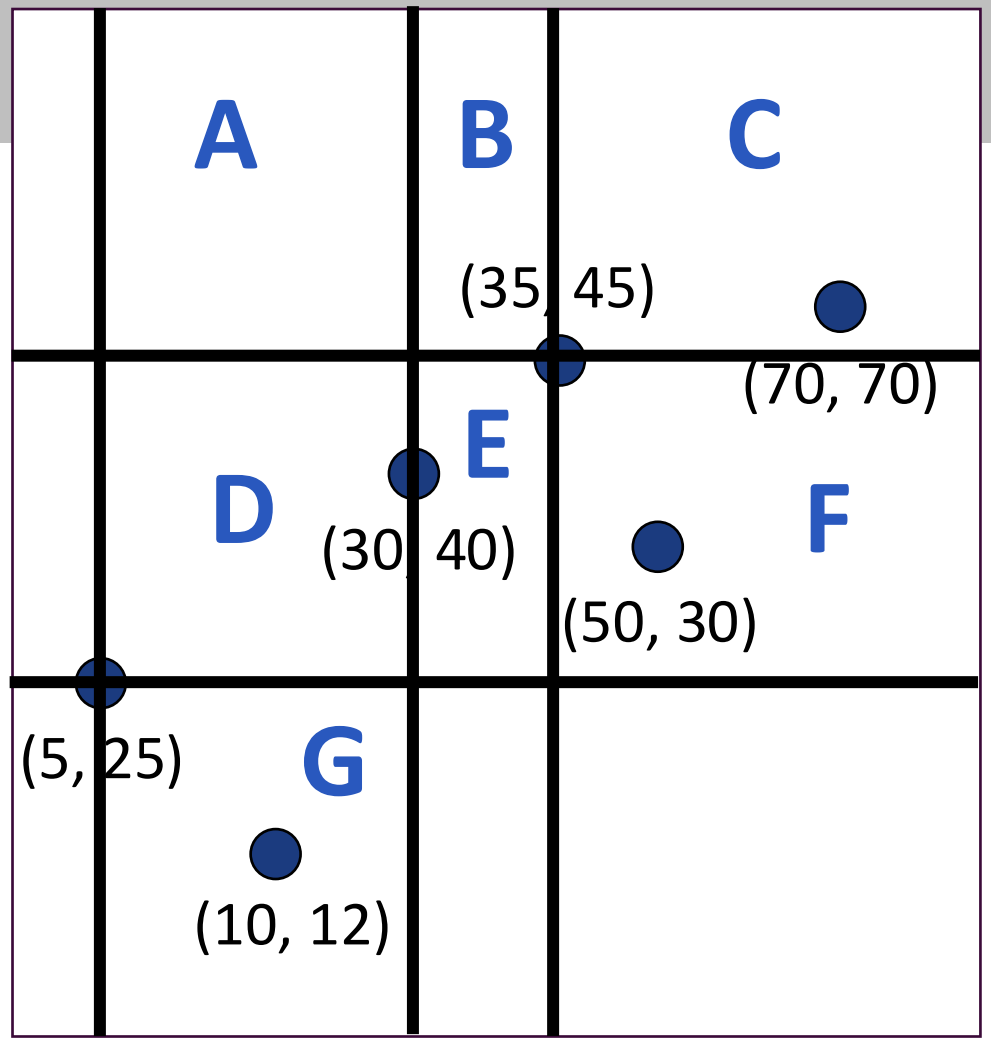
kd-Trees Bounding Box

In general, how do you check for which bounding boxes could contain a close enough point?



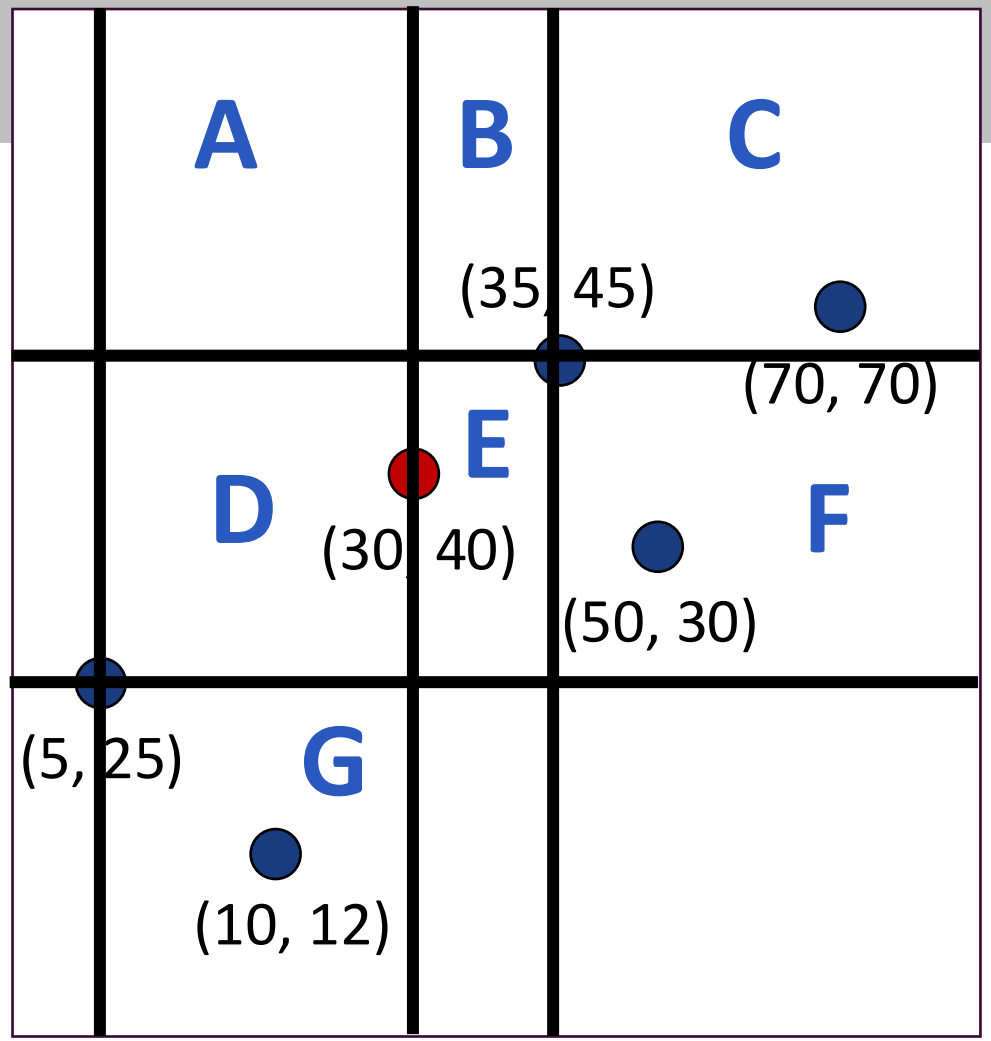
kd-Trees Bounding Box

Key here: search for the correct bounding boxes as you go down the recursion DFS tree



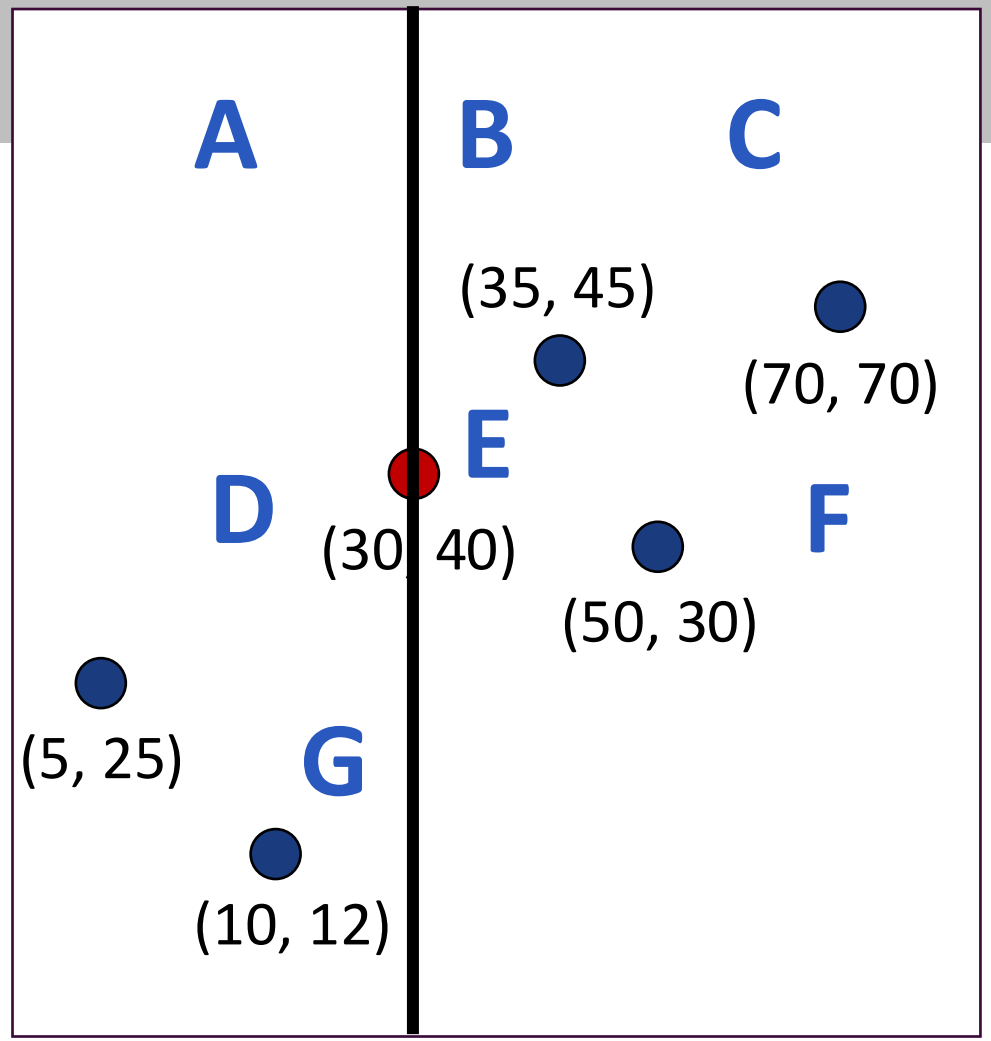
kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)



kd-Trees and 2-Nearest Neighbors Example

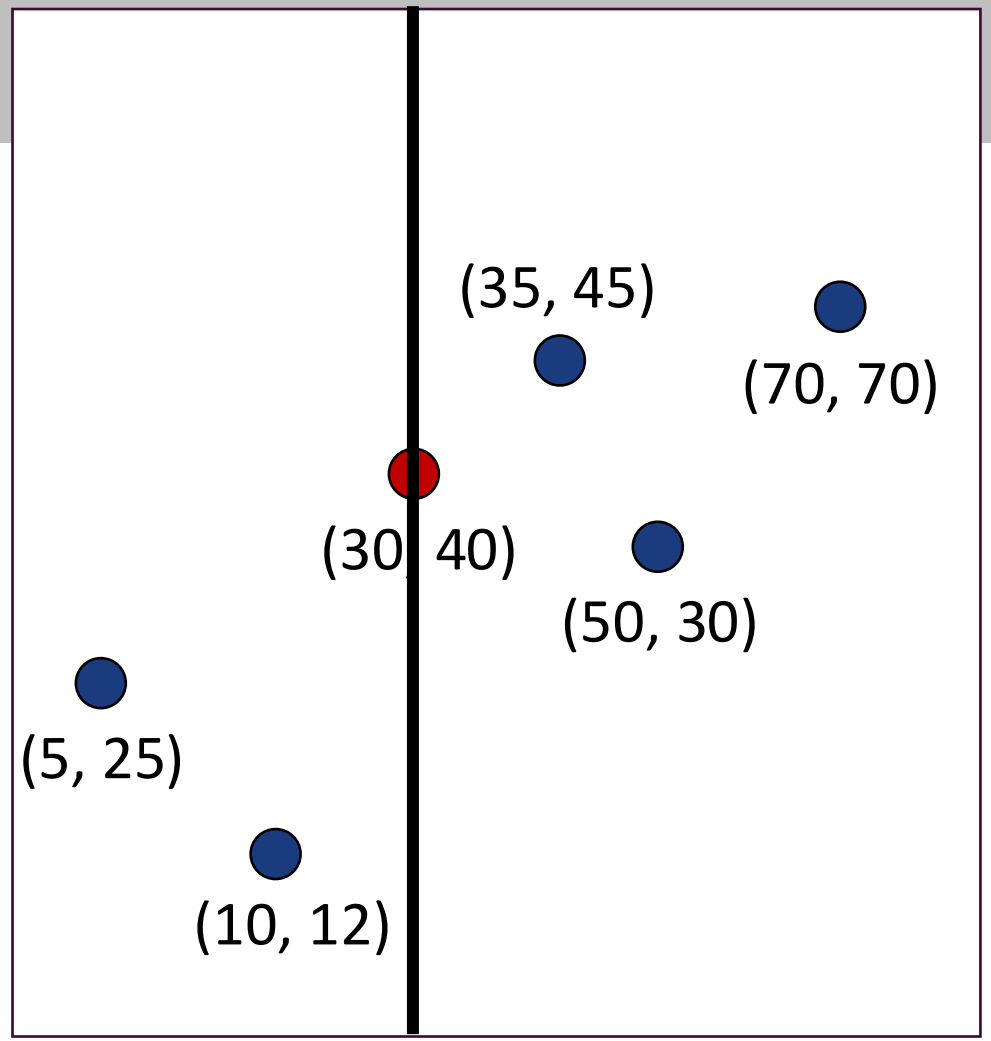
Example: (32, 29)



kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

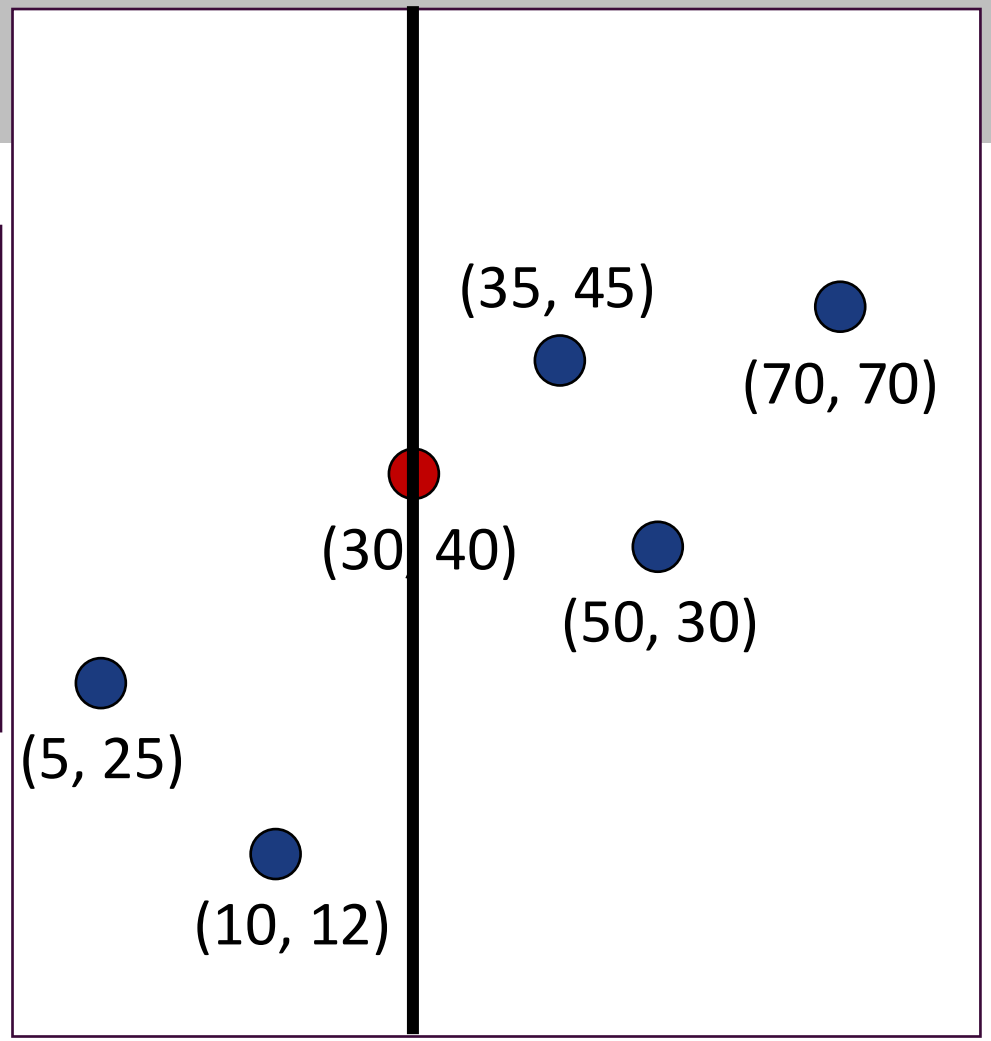
Traverse right, then left



kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Compute distance of (30, 40) to (32, 29)

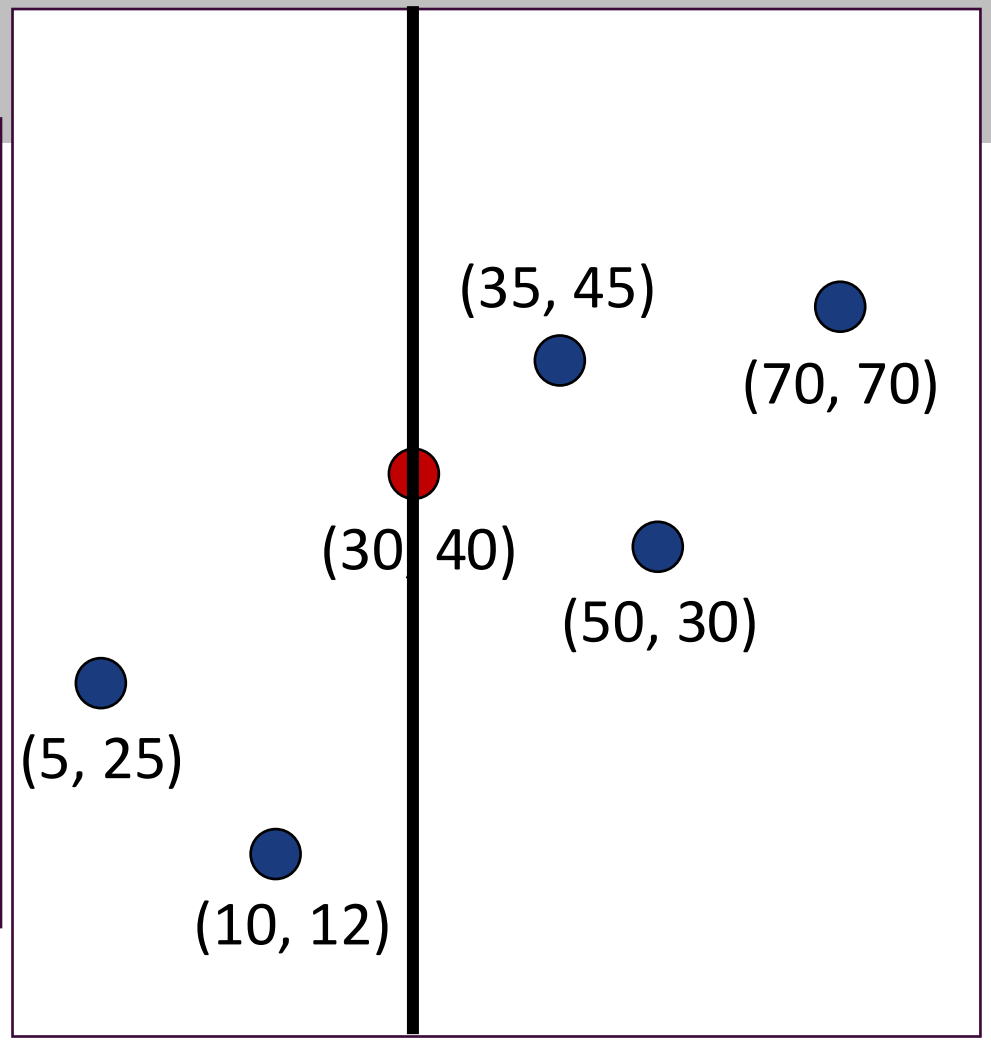


kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Compute distance of (30, 40) to (32, 29)

If smaller than max distance in max-heap, add to heap

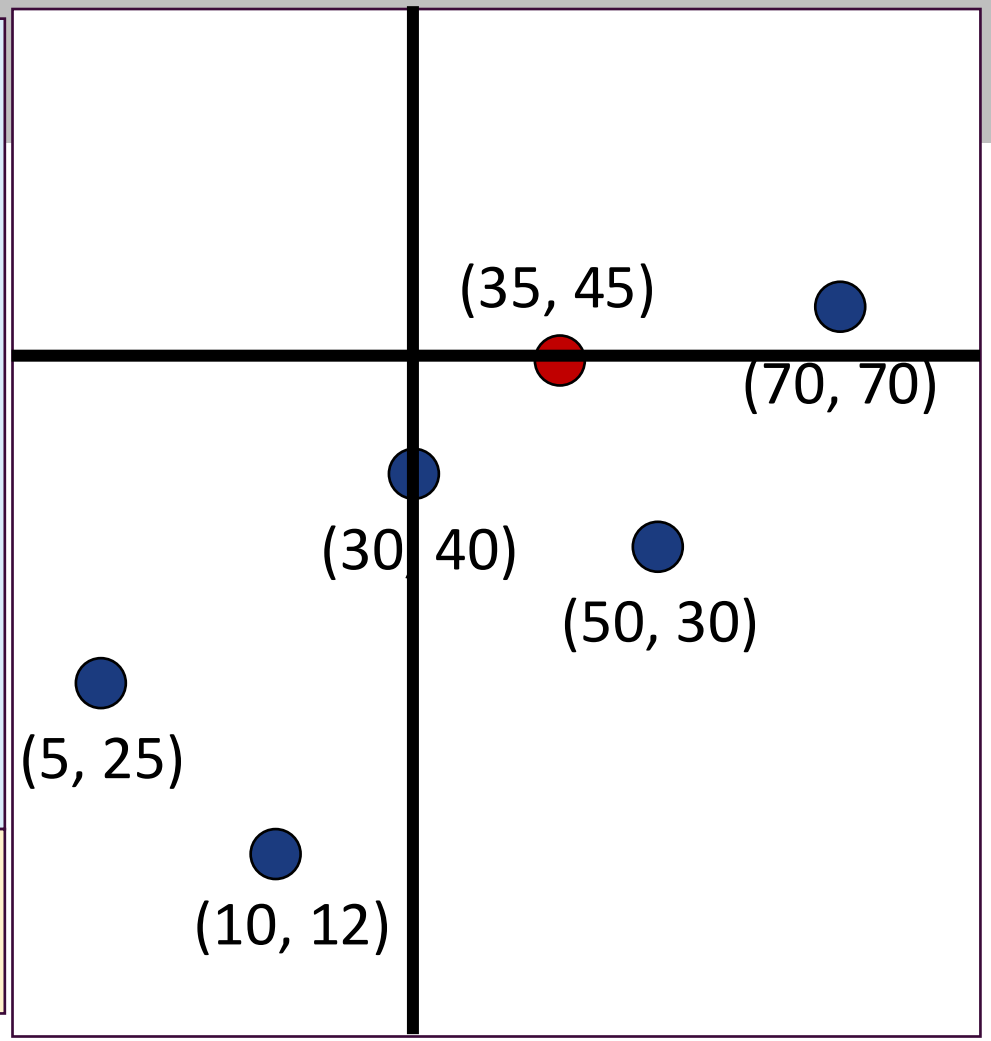


kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Traverse down first, then up

Heap: (32, 29)



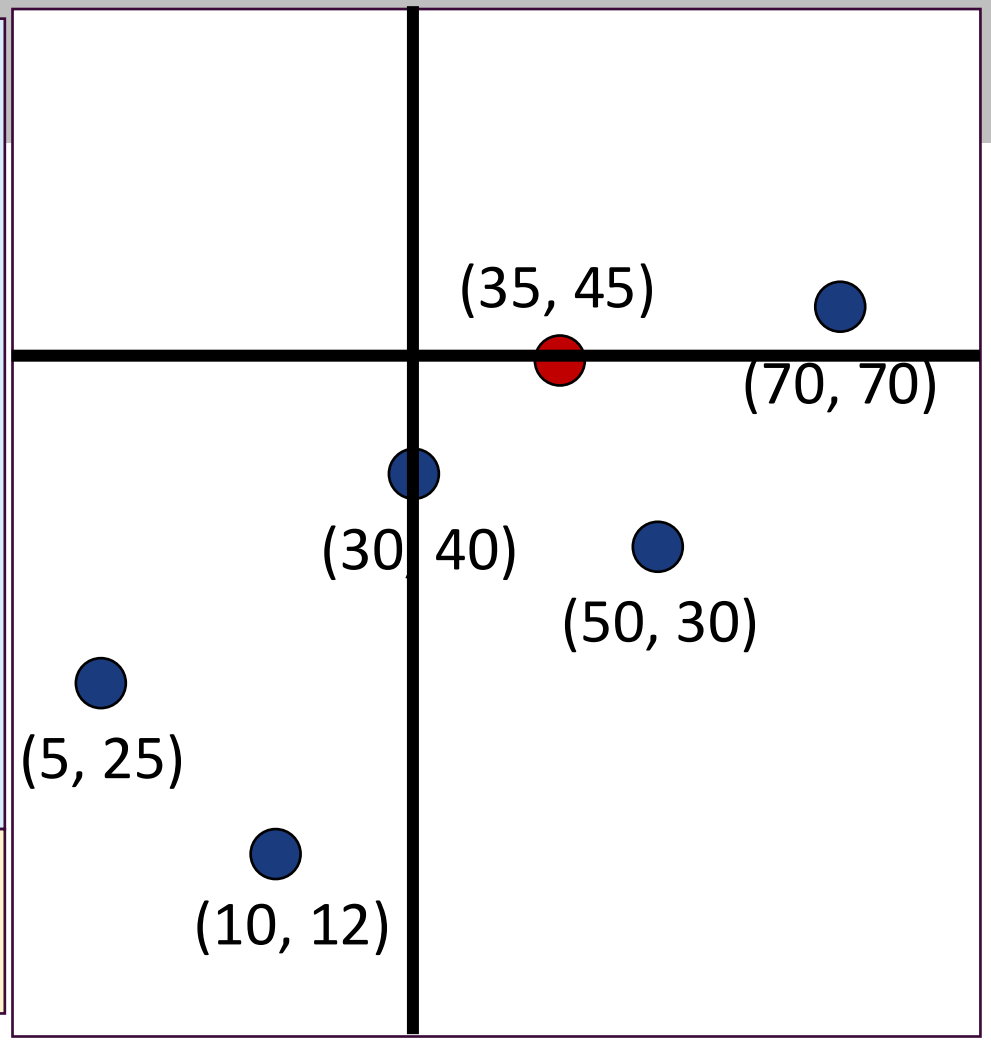
kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Traverse down first, then up

Compute distance of (35, 45) and add to heap if smaller than max

Heap: (30, 40); (35, 45)

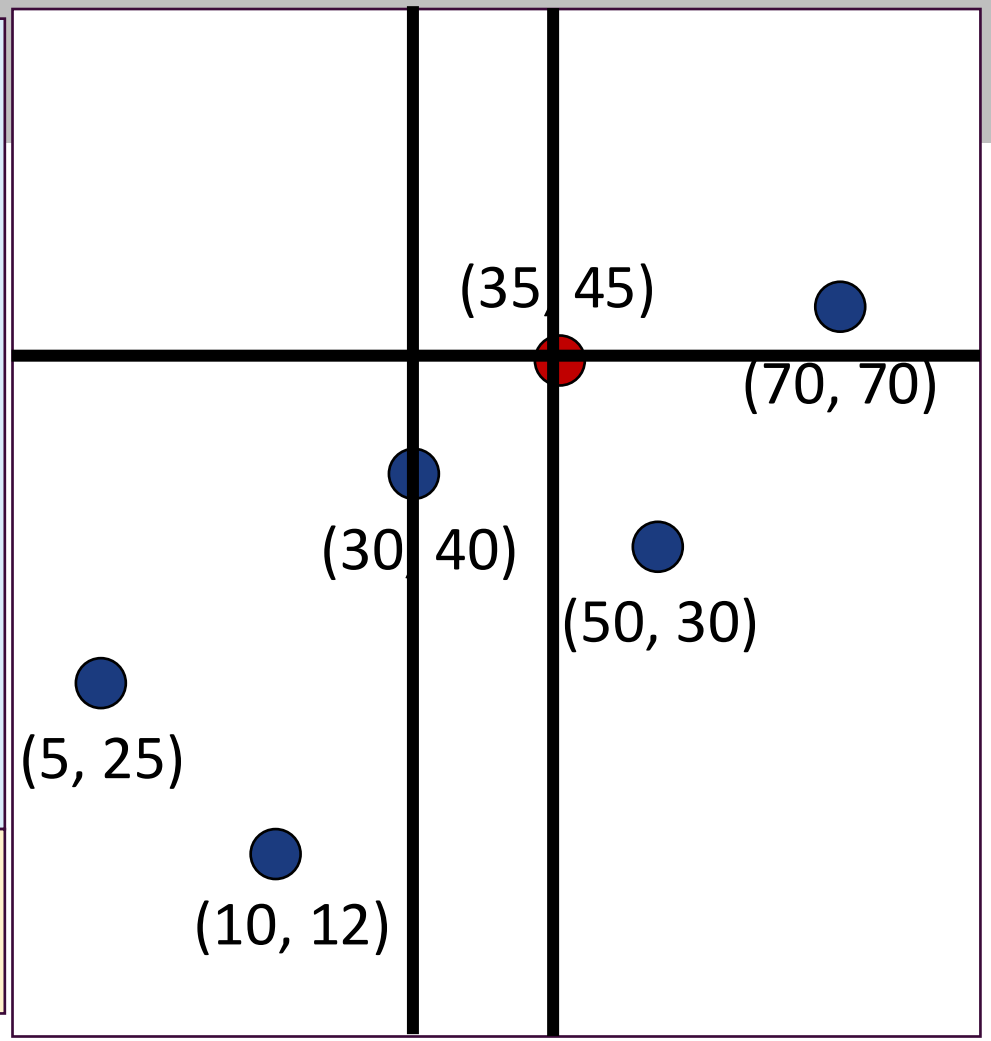


kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Traverse left first, then right

Heap: (30, 40); (35, 45)



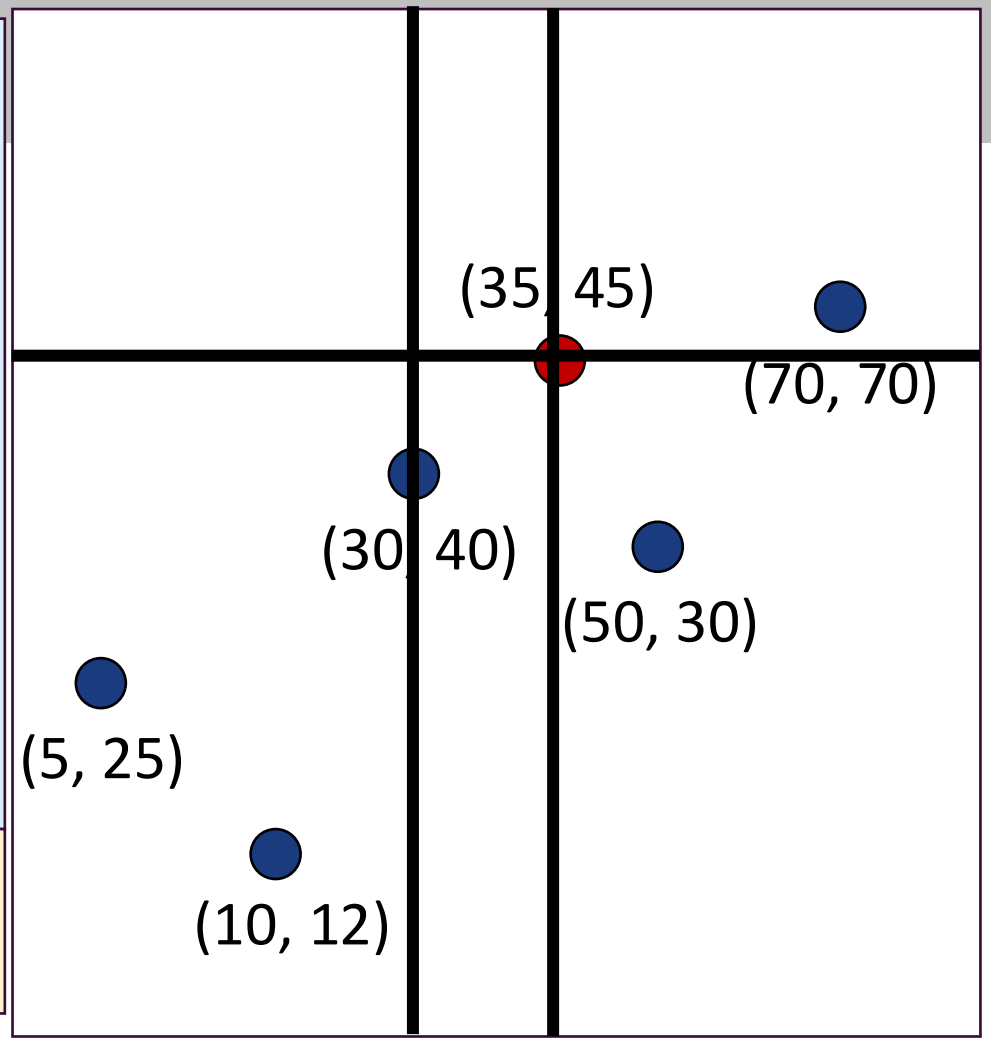
kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Traverse left first, then right

No more levels of recursion!

Heap: (30, 40); (35, 45)

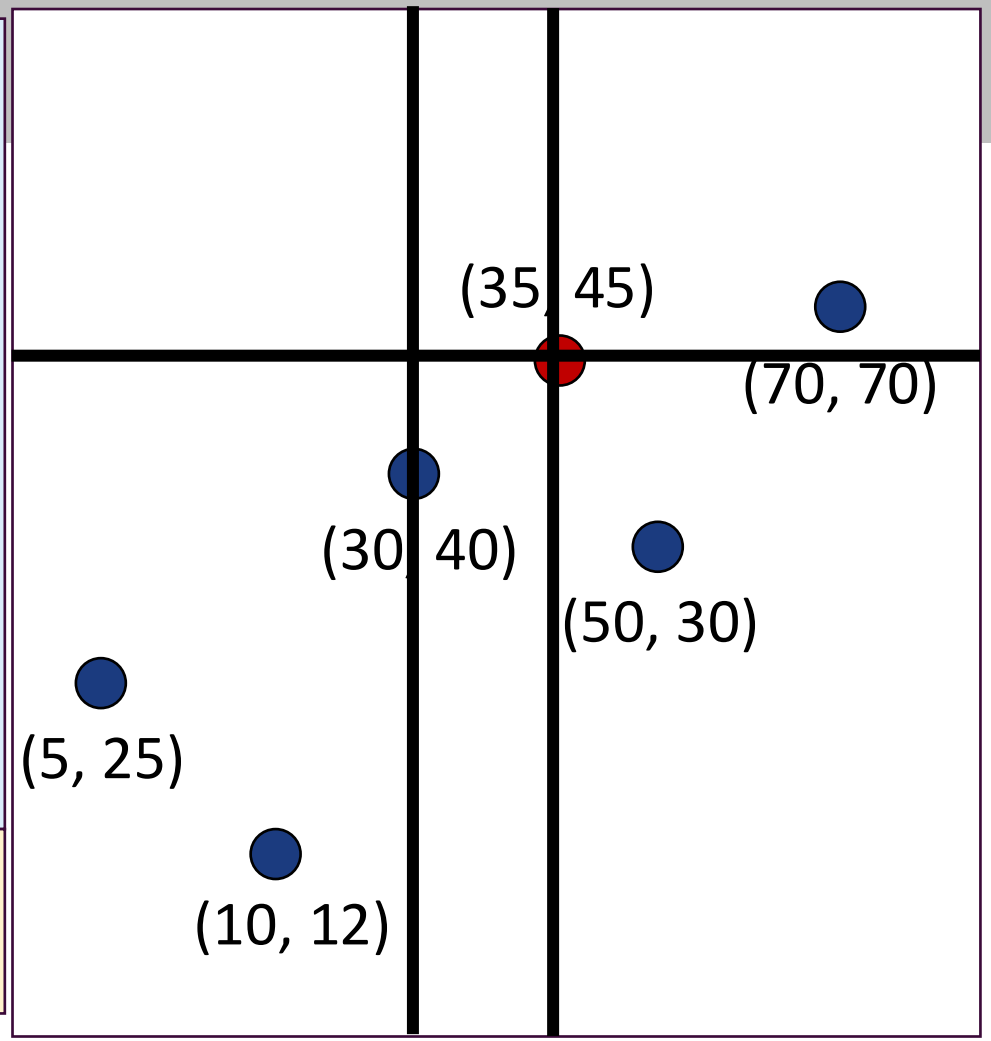


kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Go back up the DFS recursion tree

Heap: (30, 40); (35, 45)

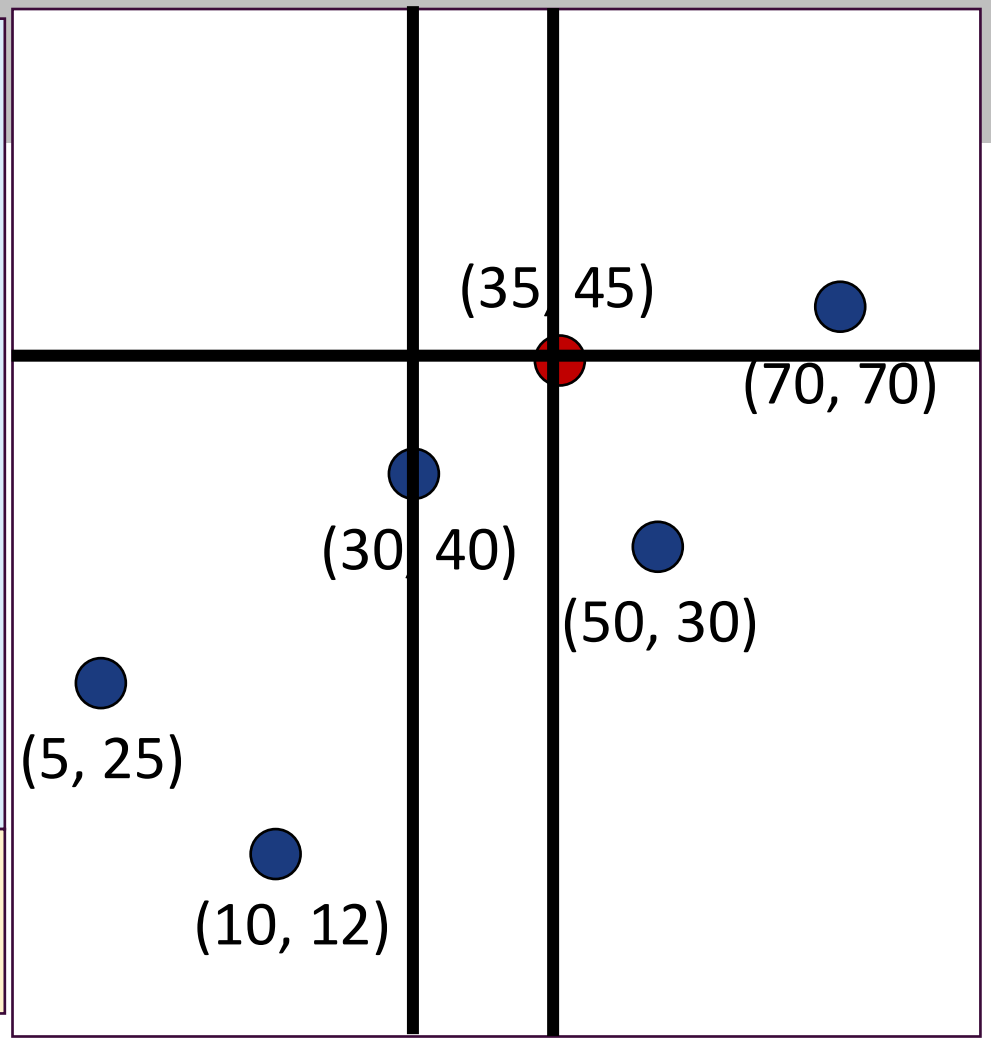


kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Traverse left first, then right

Heap: (30, 40); (35, 45)



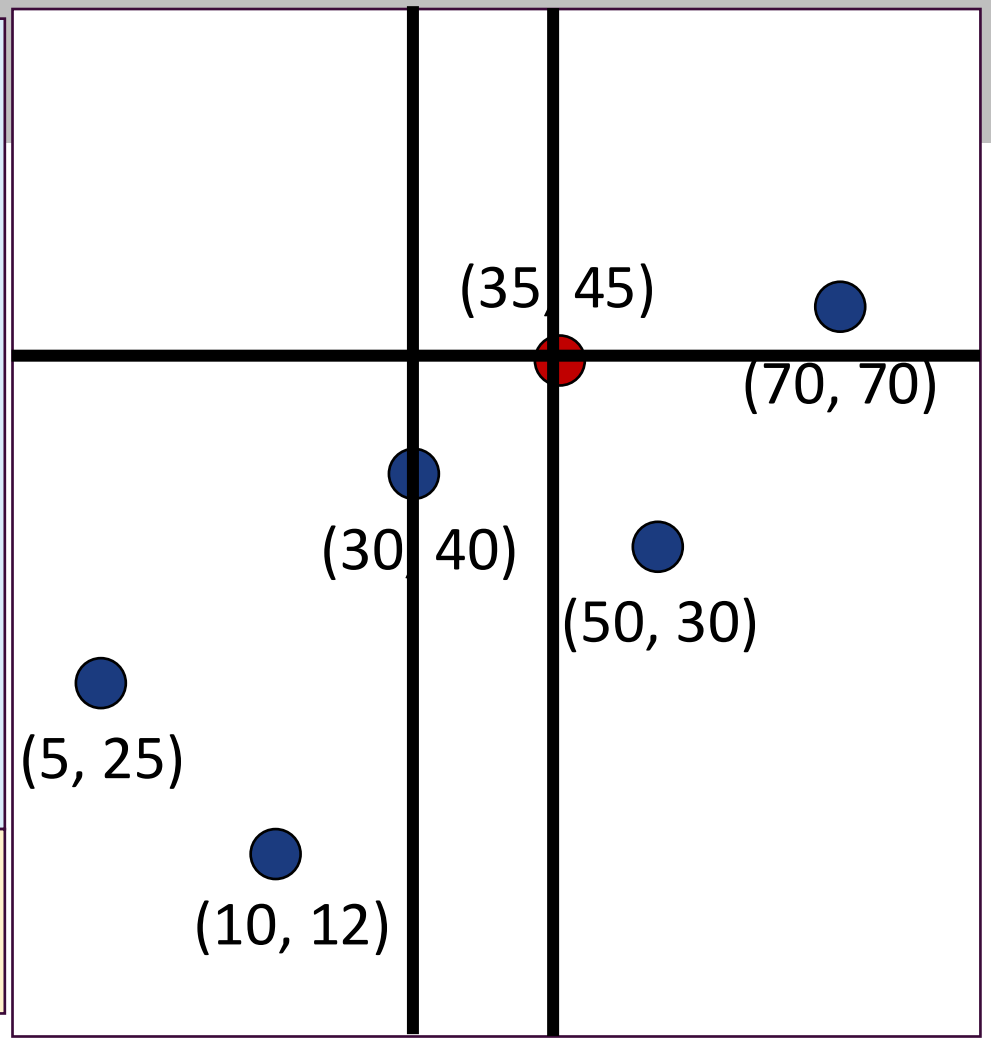
kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Traverse left first, then right

Find out if you need to traverse right

Heap: (30, 40); (35, 45)



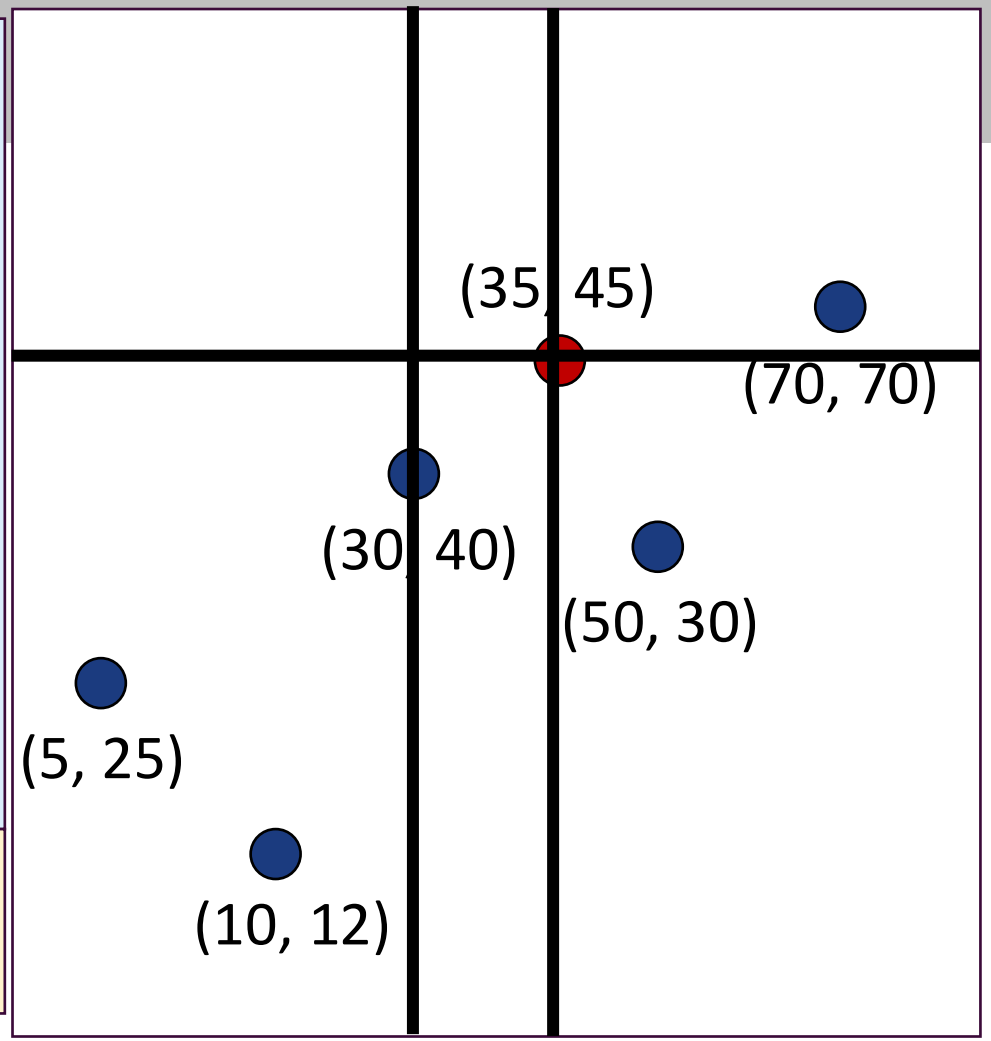
kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Traverse left first, then right

Find distance to bounding box!

Heap: (30, 40); (35, 45)



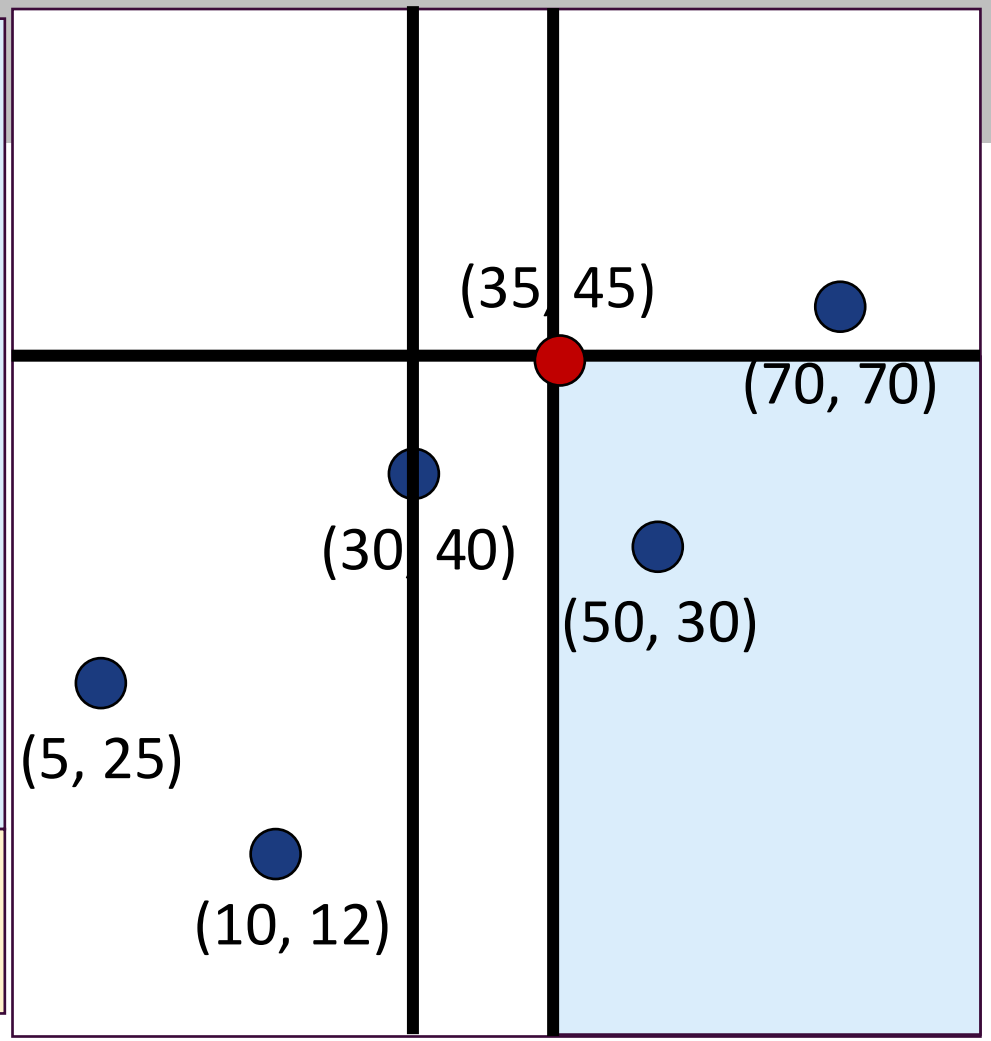
kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Traverse left first, then right

Find distance to bounding box! What is the distance here?

Heap: (30, 40); (35, 45)



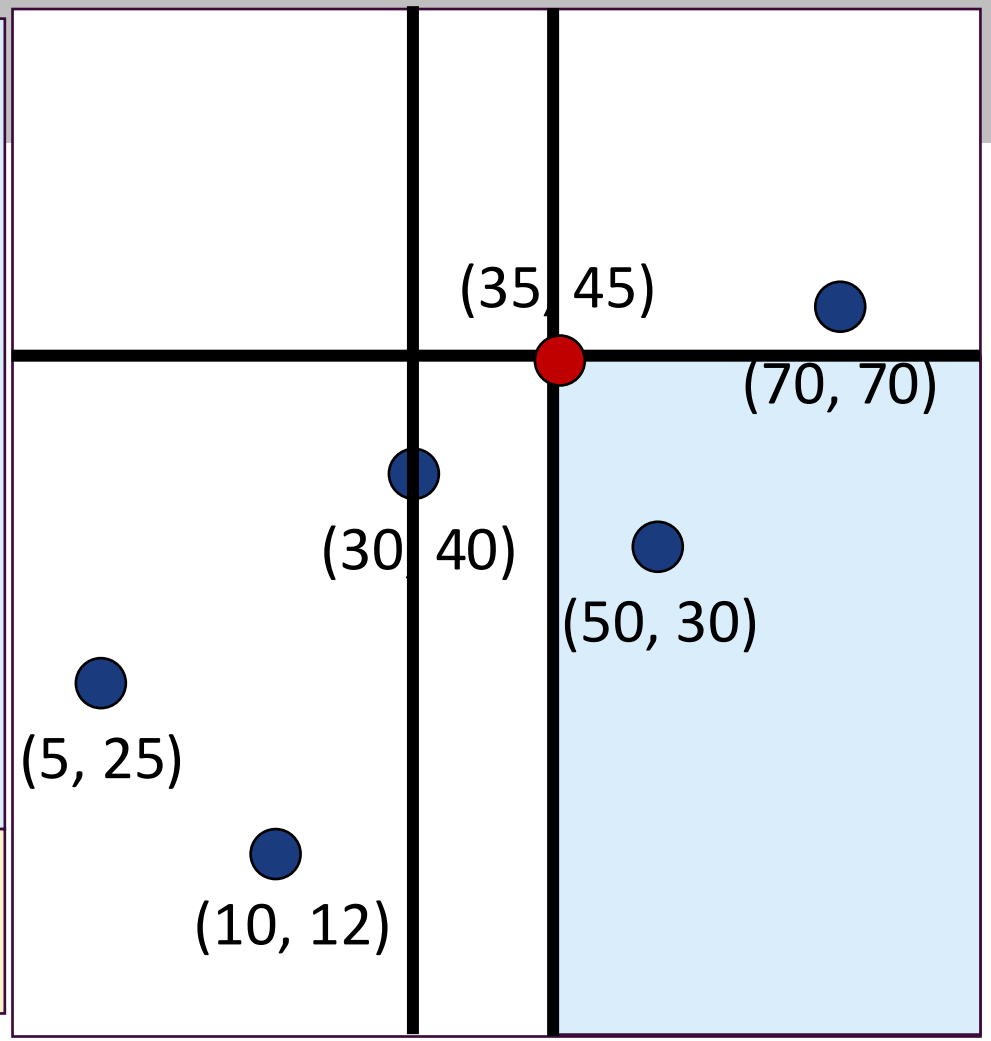
kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Traverse left first, then right

Find distance to bounding box! What is the distance here? 3! Smaller than max

Heap: (30, 40); (35, 45)

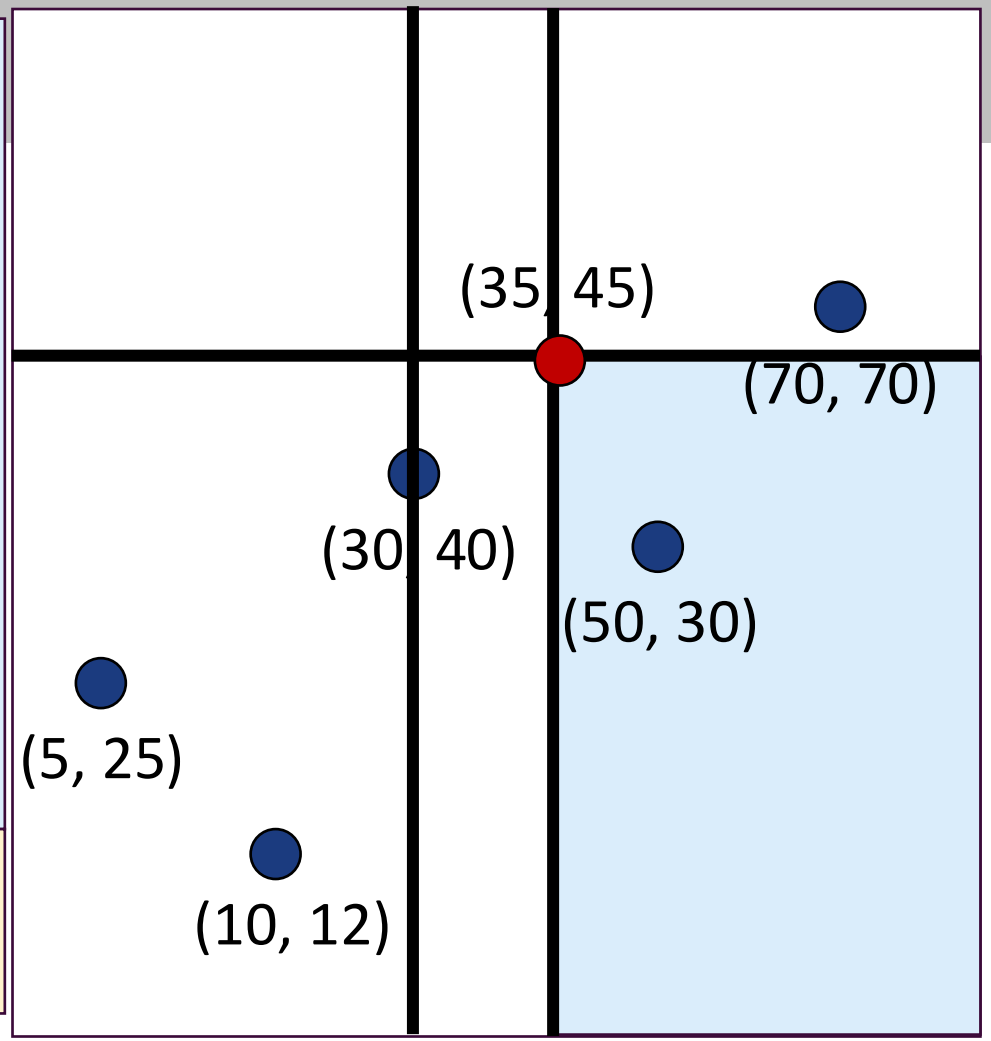


kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Traverse Right

Heap: (30, 40); (35, 45)

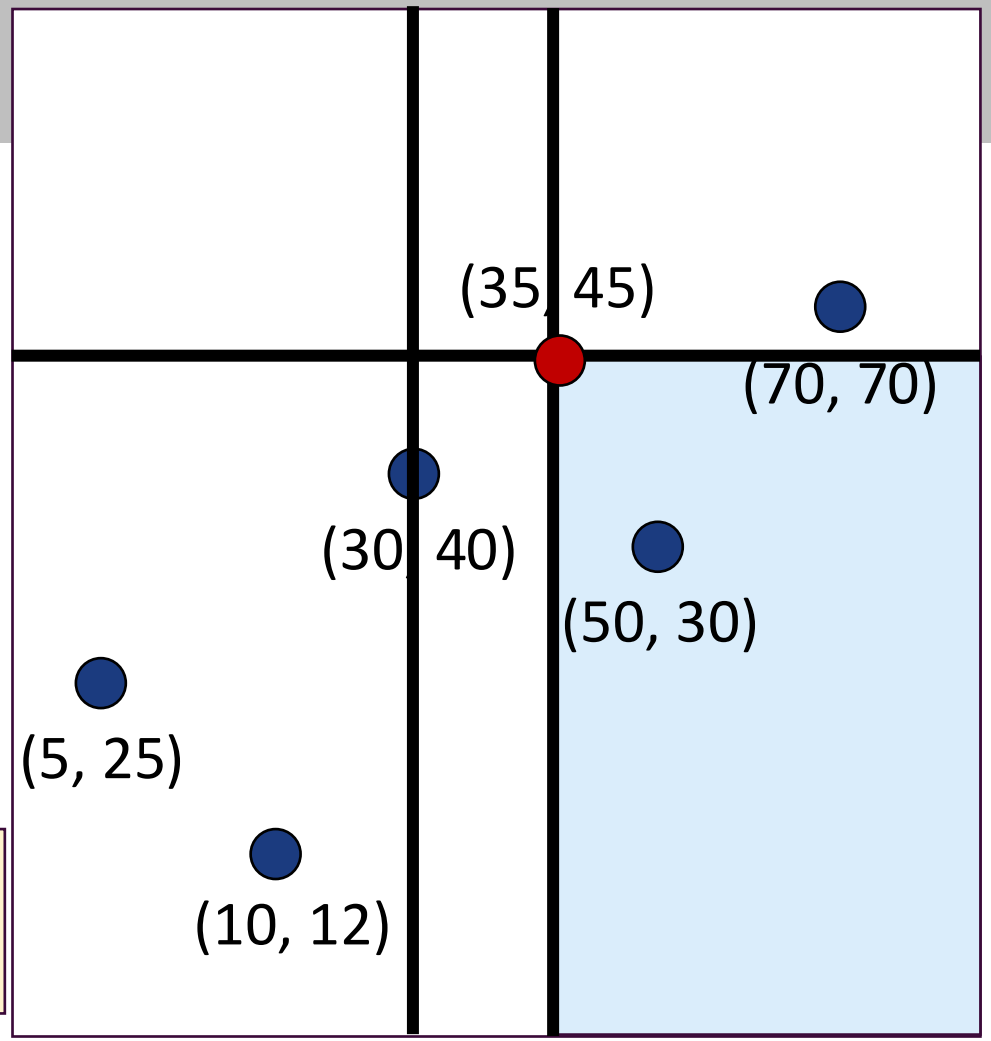


kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Traverse Right; check distance from (50, 30) to (32, 29)

Heap: (30, 40); (35, 45)

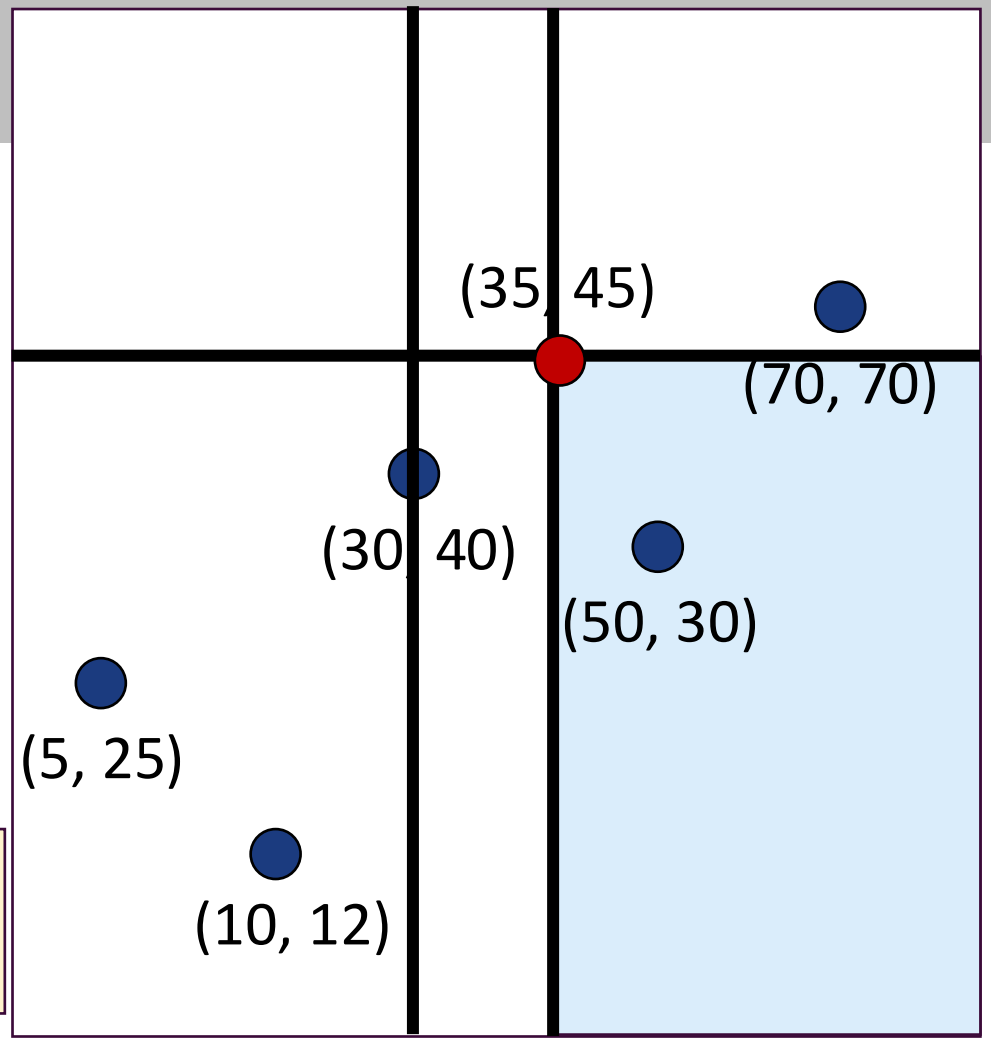


kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Go back up the DFS recursion tree

Heap: (30, 40); (35, 45)

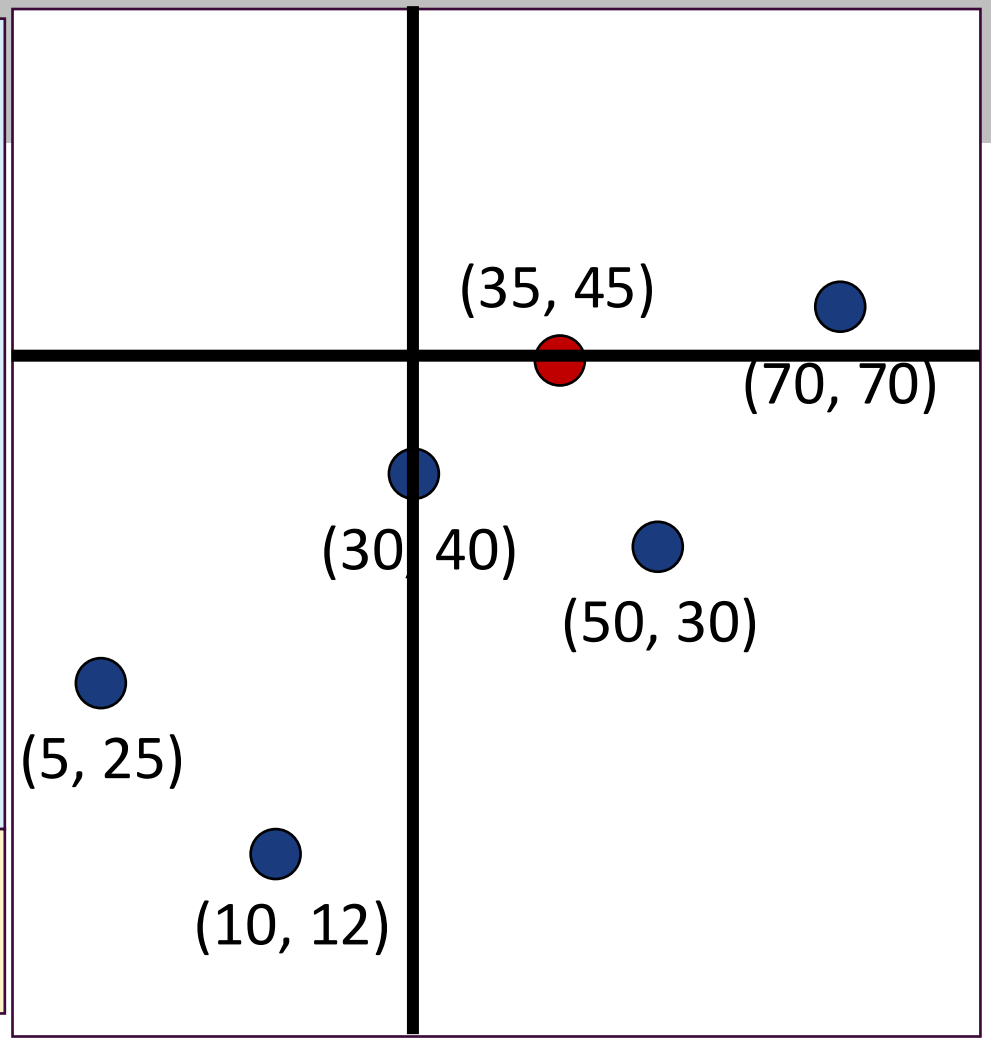


kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Traverse down first, then up

Heap: (30, 40); (35, 45)



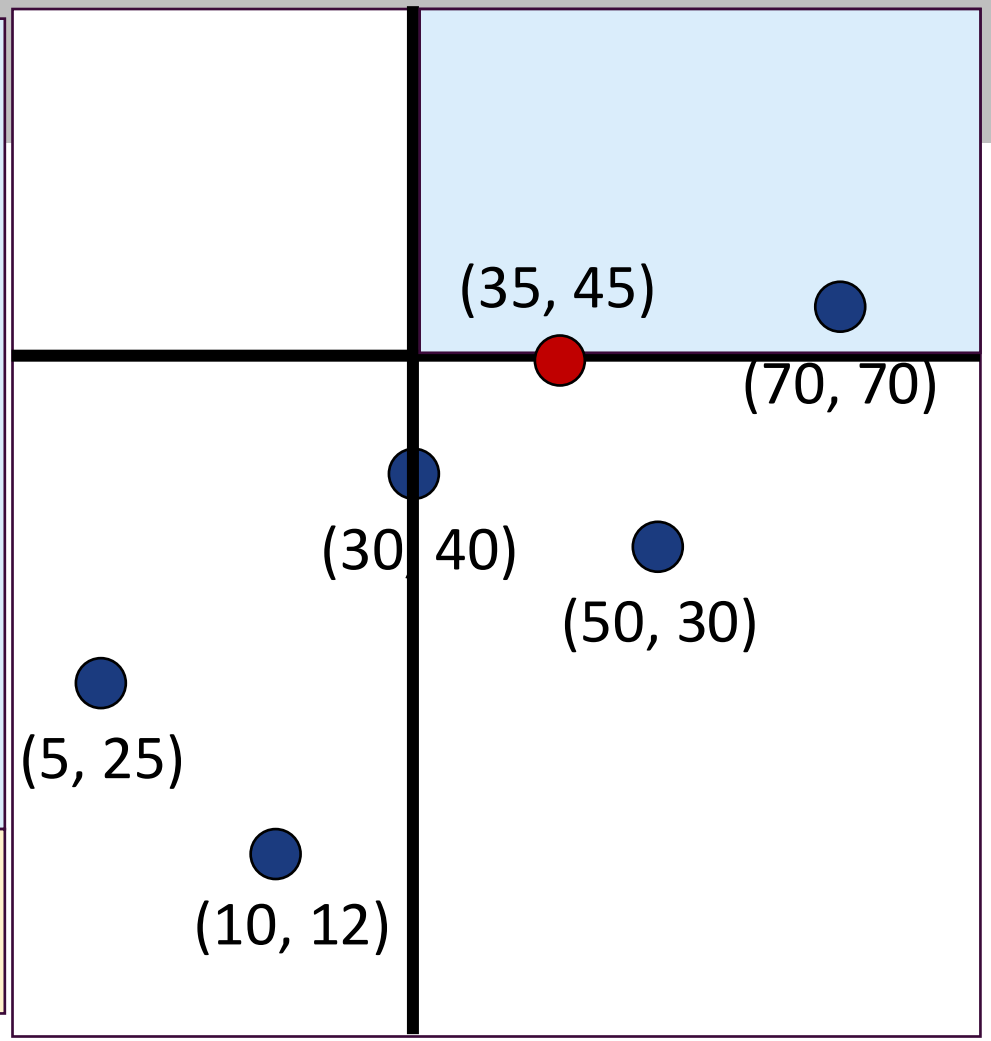
kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Traverse down first, then up

Distance to bounding box is 16, less than max (barely)!

Heap: (30, 40); (35, 45)



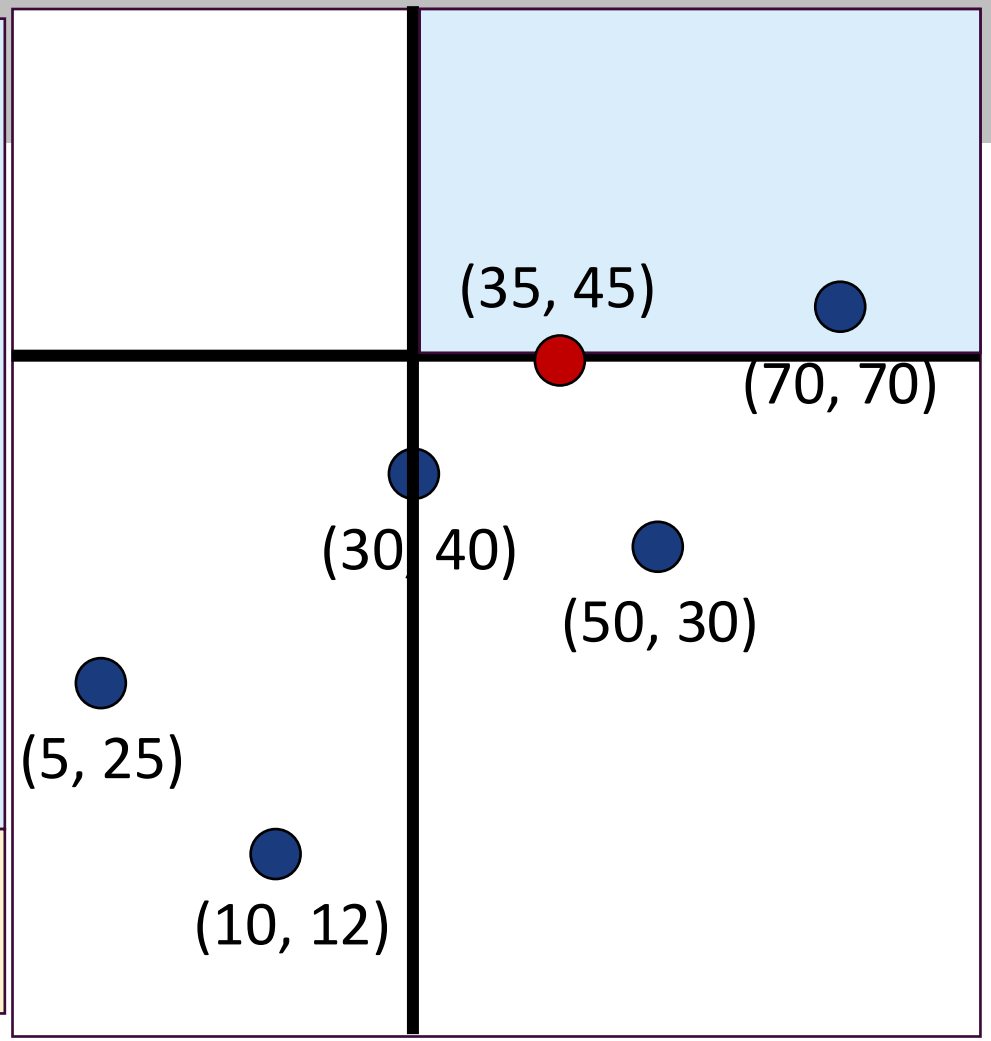
kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Traverse down first, then up

Recurse up; (70, 70) distance is much greater; do nothing

Heap: (30, 40); (35, 45)

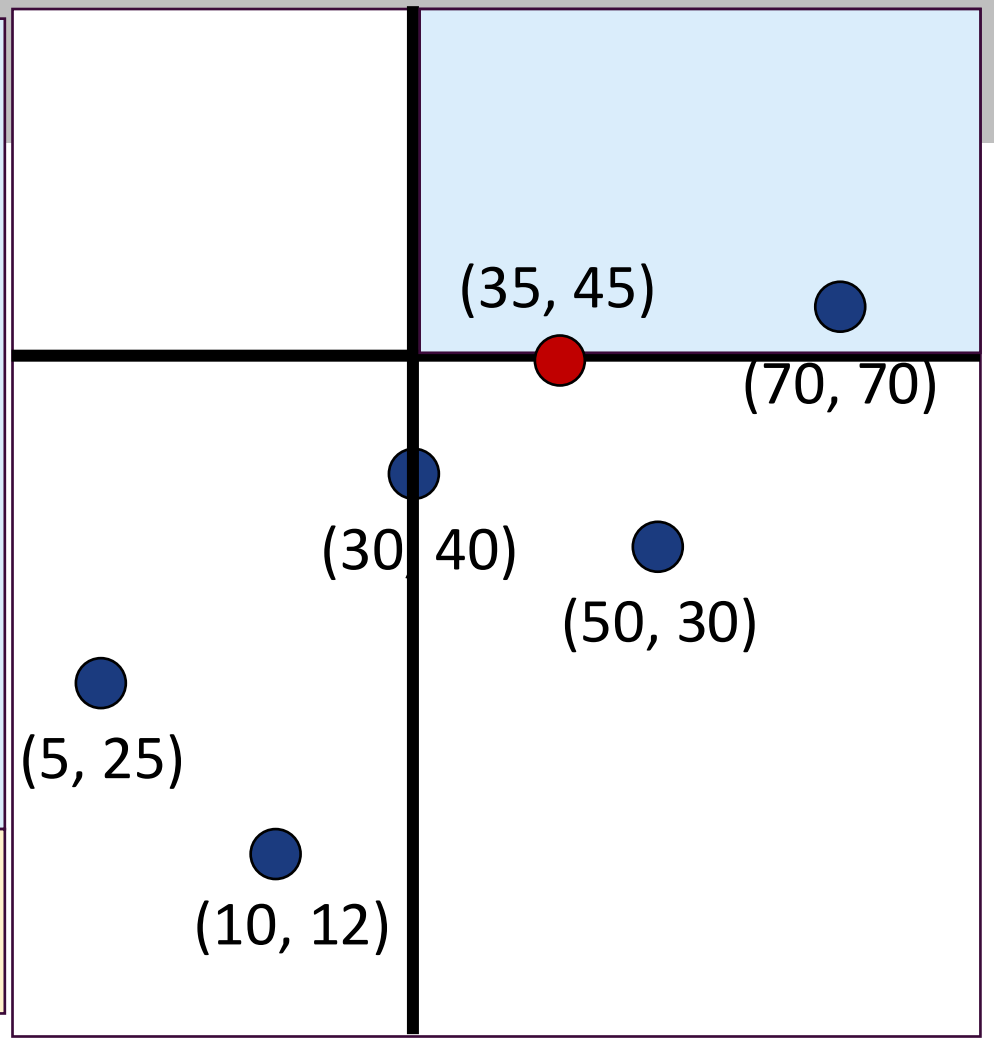


kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Go back up DFS recursion tree

Heap: (30, 40); (35, 45)

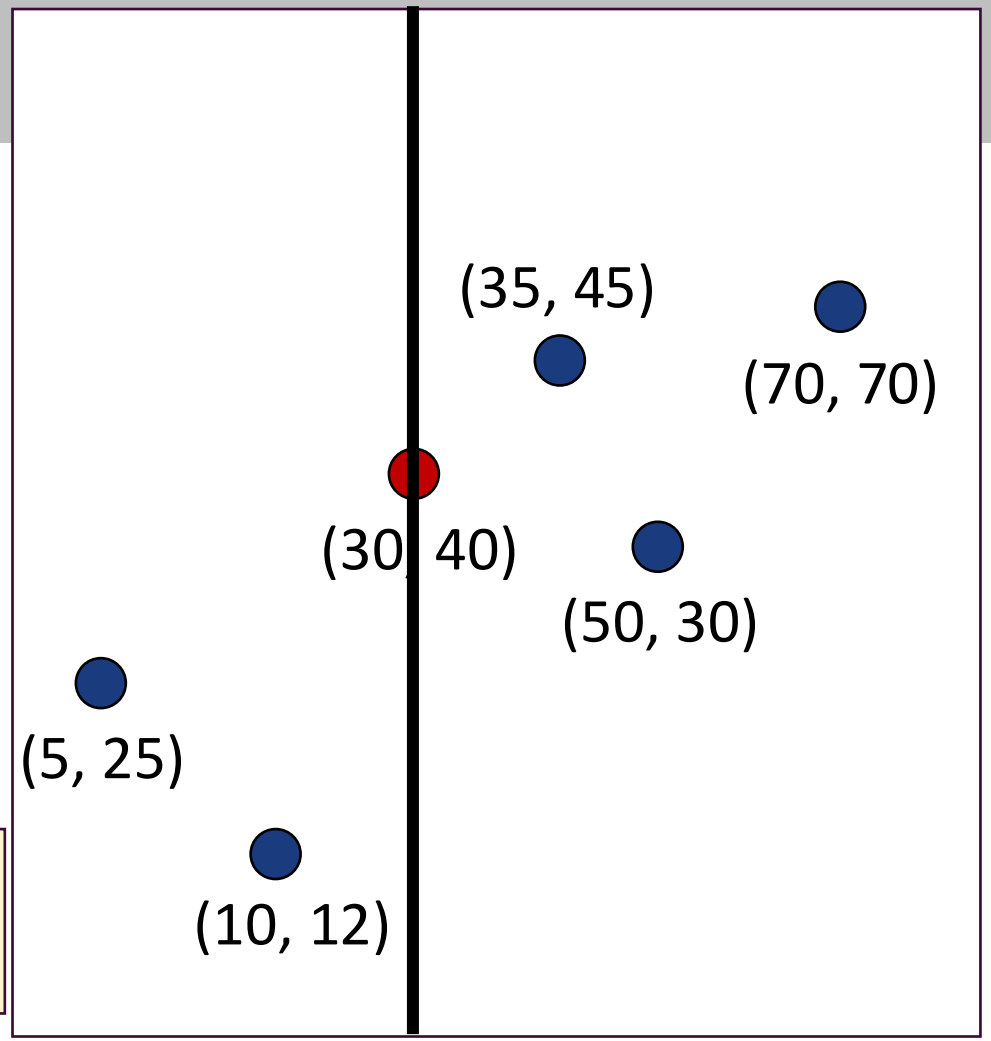


kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Traverse right, then left

Distance: 2!

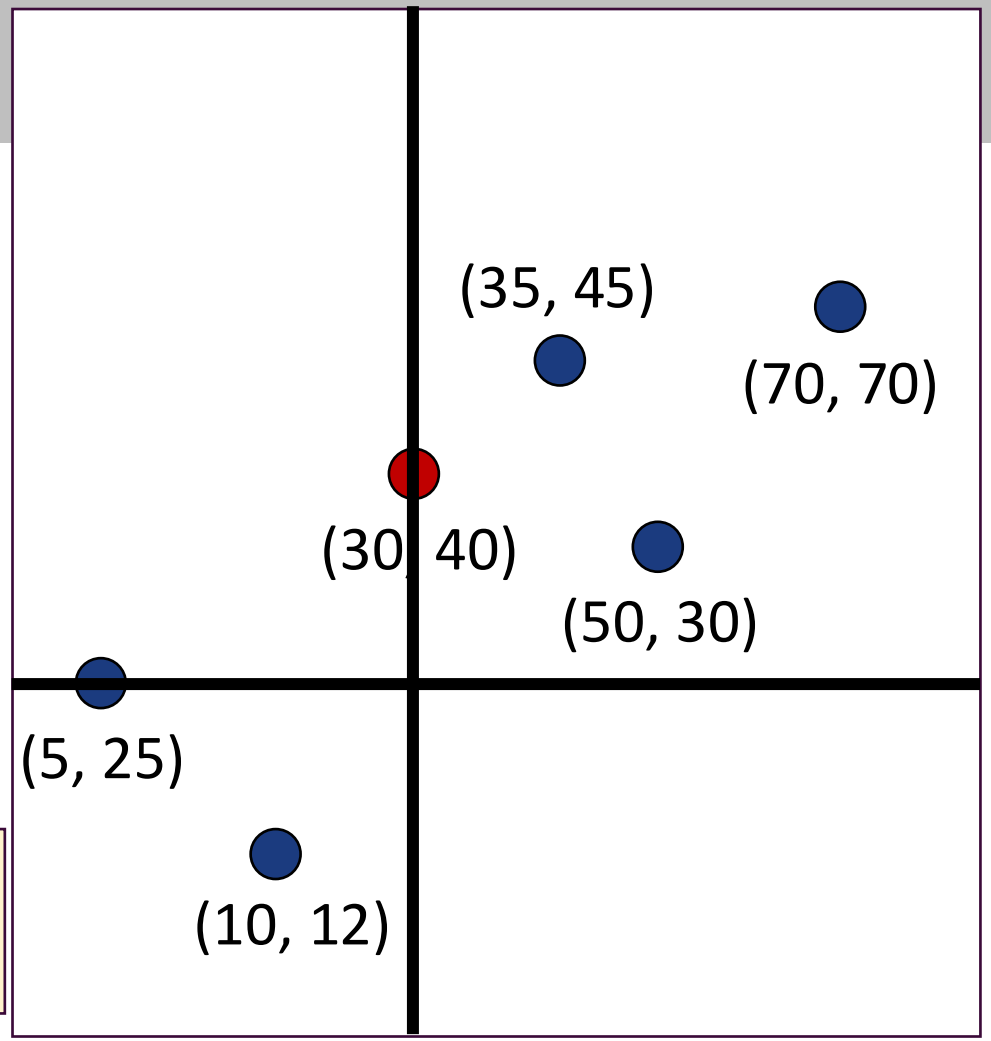


Heap: (30, 40); (35, 45)

kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Traverse up, then down



Heap: (30, 40); (35, 45)

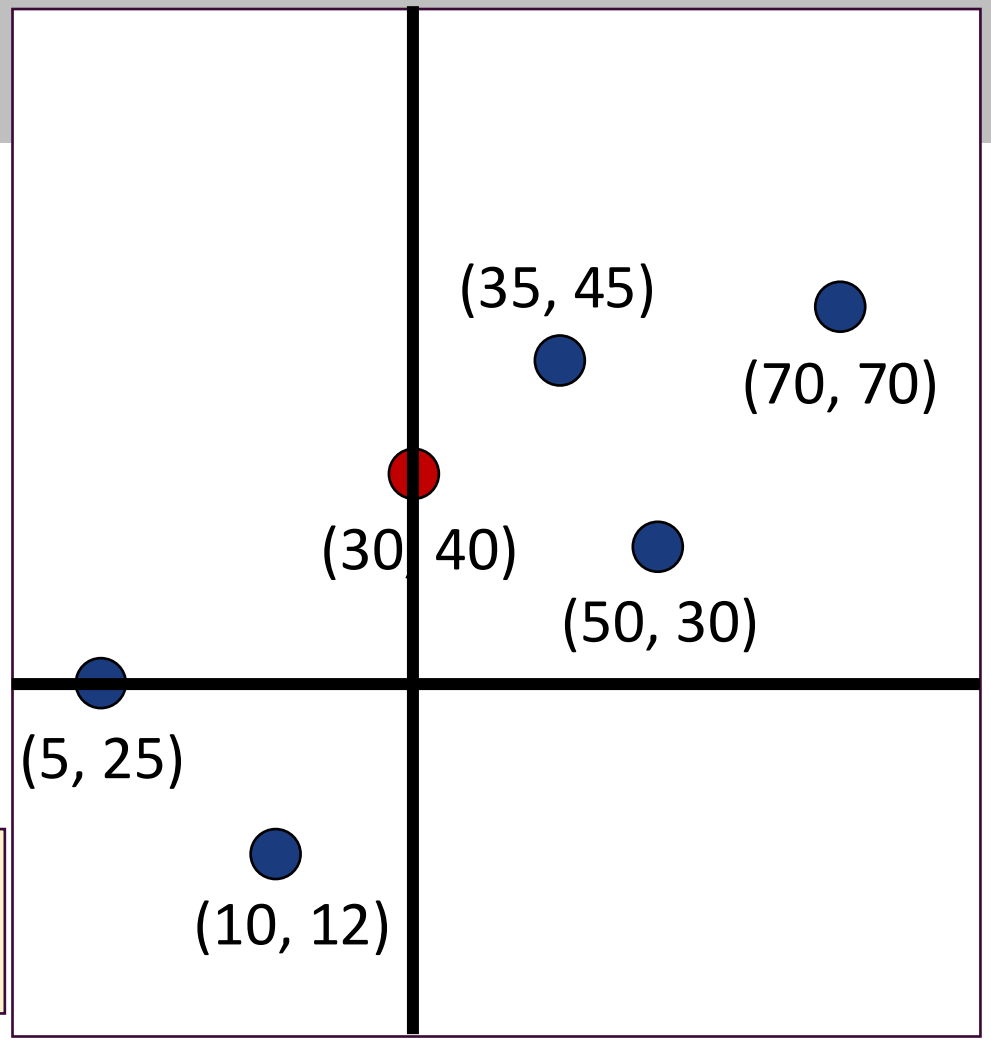
kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

Traverse up, then down

Check (5, 25) with the max distance; not greater

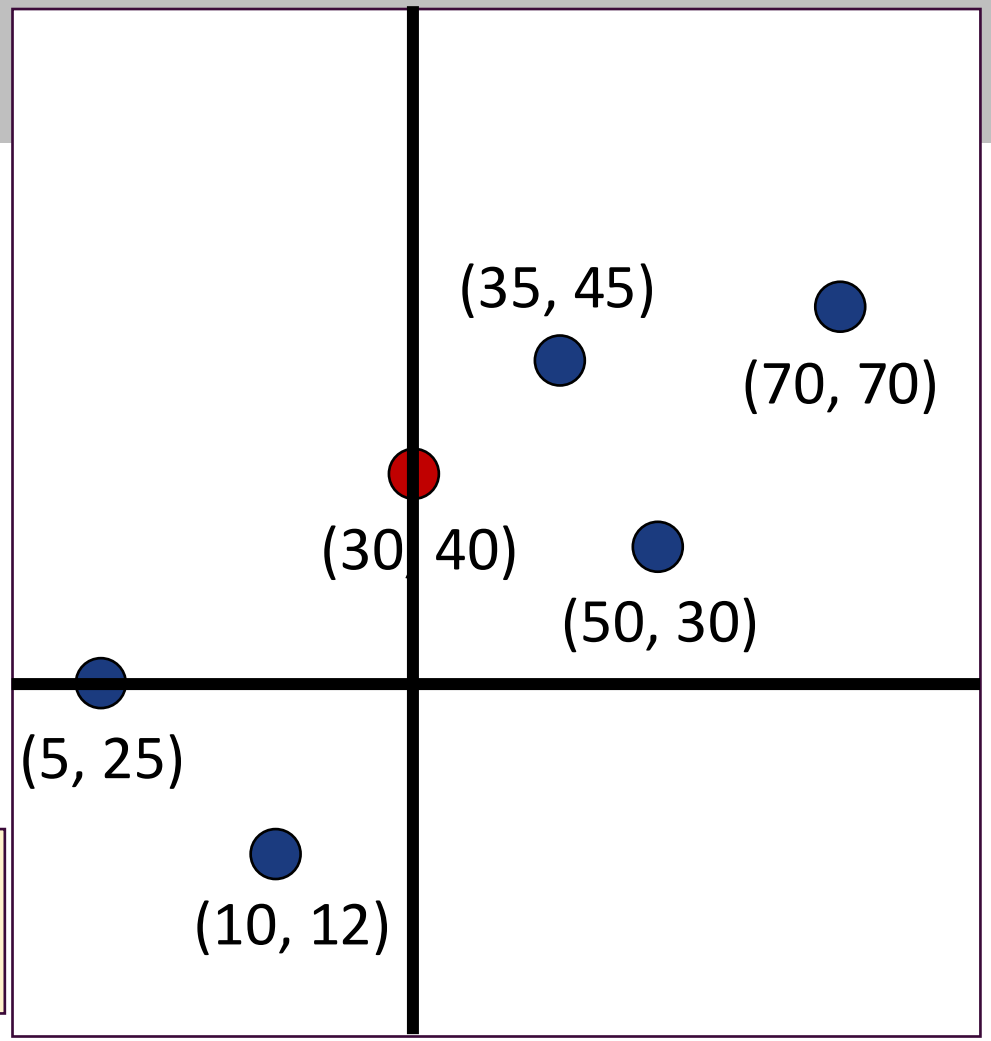
Heap: (30, 40); (35, 45)



kd-Trees and 2-Nearest Neighbors Example

Example: (32, 29)

And so on…

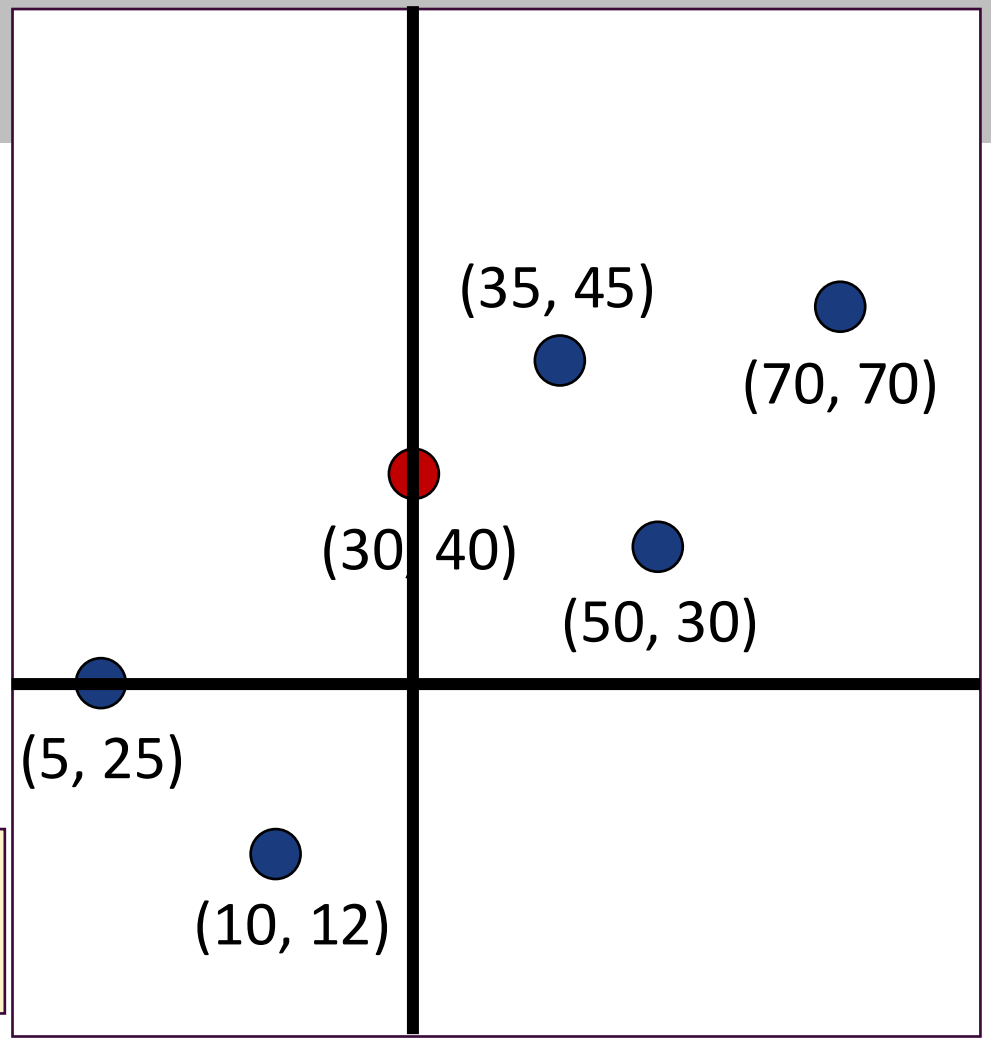


Heap: (30, 40); (35, 45)

kd-Trees and 2-Nearest Neighbors Example

Runtime guarantees?

Build? Query?

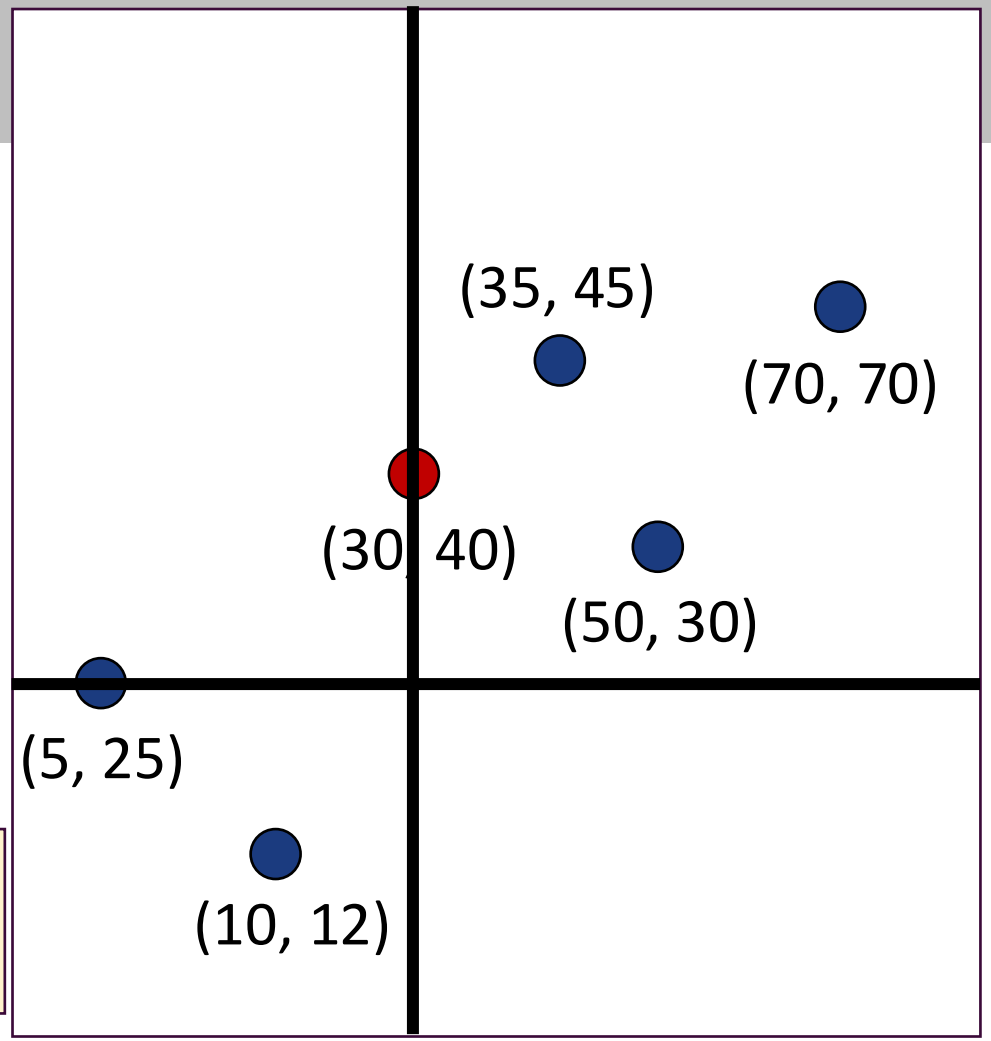


Heap: (30, 40); (35, 45)

kd-Trees and 2-Nearest Neighbors Example

Runtime guarantees?

Build? Suppose n points



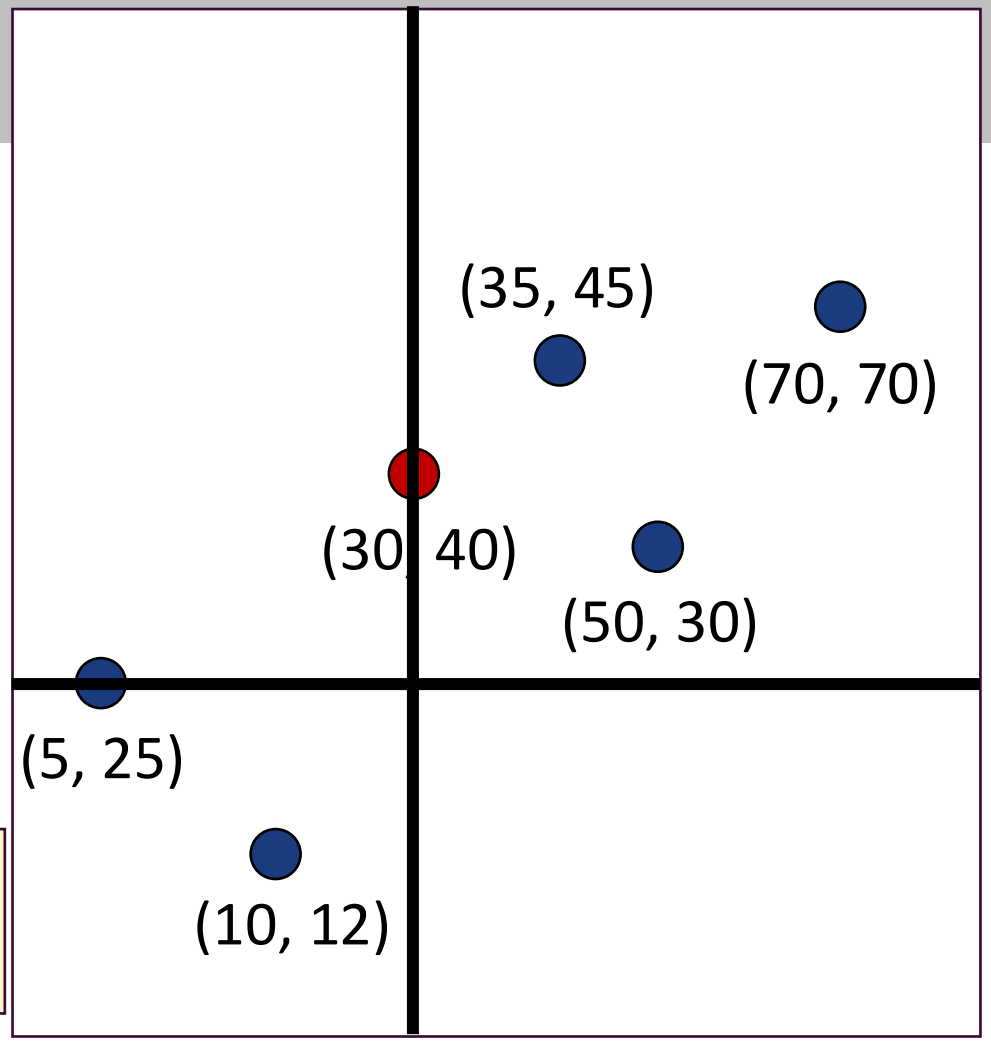
Heap: (30, 40); (35, 45)

kd-Trees and 2-Nearest Neighbors Example

Runtime guarantees?

Build? Suppose n points

$O\left(n \log^2(n)\right)$ **naively**

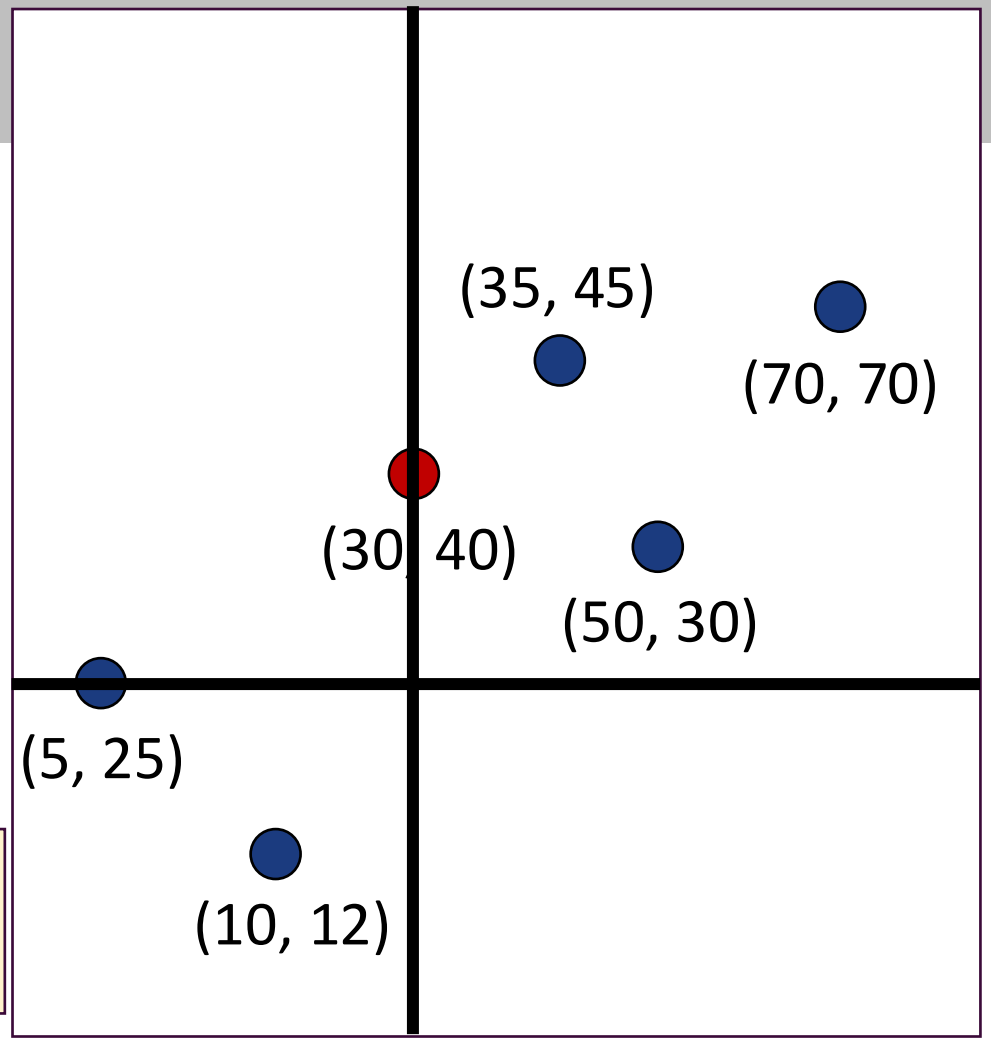


Heap: (30, 40); (35, 45)

kd-Trees and 2-Nearest Neighbors Example

Runtime guarantees?

Per Query Runtime?



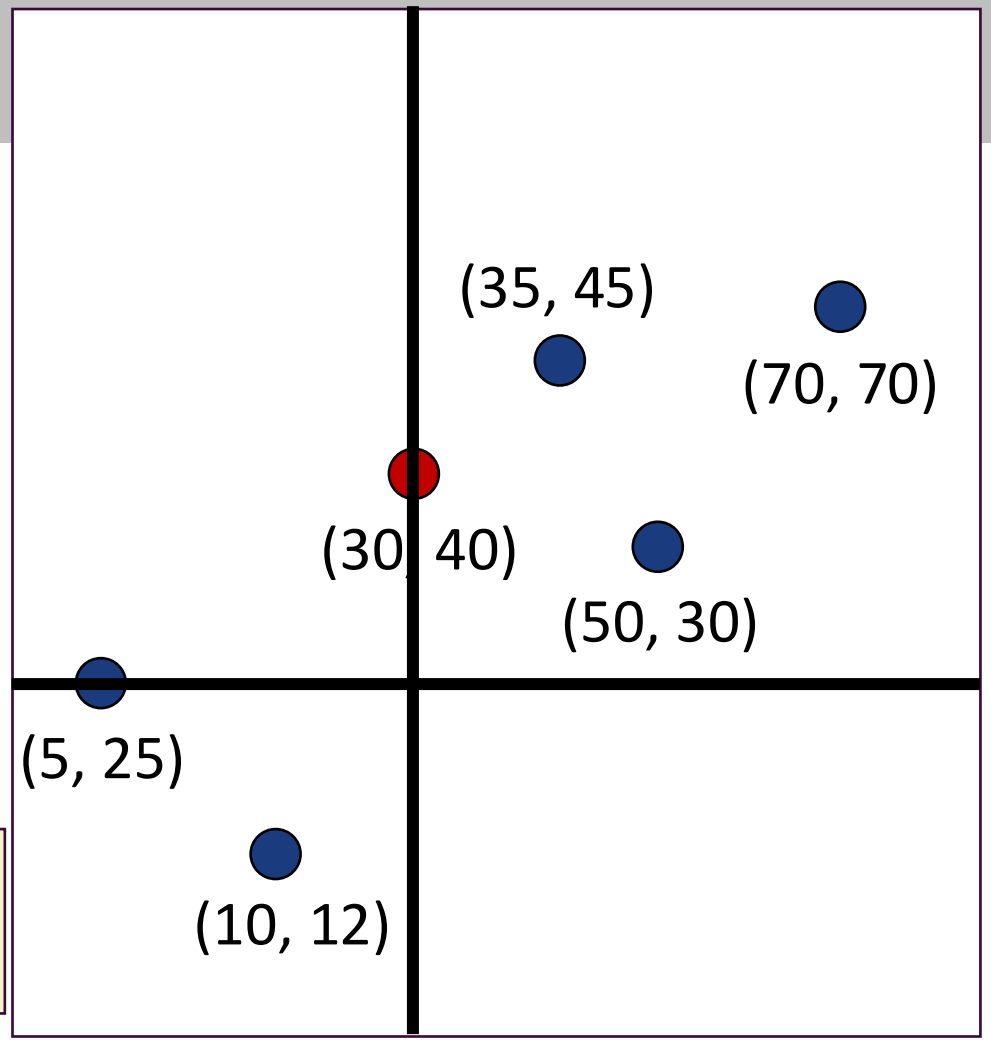
Heap: (30, 40); (35, 45)

kd-Trees and 2-Nearest Neighbors Example

Runtime guarantees?

Per Query Runtime?

$\boldsymbol{O(\log n + k \log k)}$ **best case**



Heap: (30, 40); (35, 45)

kd-Trees and 2-Nearest Neighbors Example

Runtime guarantees?

Per Query Runtime?

$O(\log n + k \log k)$ **best case**

$O(n \log k)$ **worst case**

Heap: (30, 40); (35, 45)