CS 225

Data Structures

March 5 – AVL Applications

Wade Fagen-Ulmschneider

AVL Runtime Proof

On Friday, we proved an upper-bound on the height of an AVL tree is **2*lg(n)** or **O(lg(n))**.

Summary of Balanced BST

AVL Trees

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

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, remove, and find

Why AVL?

Summary of Balanced BST

Pros:

- Running Time:

  - Improvement Over:

- Great for specific applications:

Summary of Balanced BST

Cons:

- Running Time:

- In-memory Requirement:

Red-Black Trees in C++

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

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

Red-Black Trees in C++

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

Red-Black Trees in C++

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

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

Red-Black Trees in C++

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

CS 225 -- Course Update

This weekend, the following grades were updated:

- mp1
- mp2*
- mp3*
- lab_inheritance
- lab_quacks
- lab_trees

Iterators

Why do we care?

```
DFS dfs(...);
for ( ImageTraversal::Iterator it = dfs.begin(); it != dfs.end(); ++it ) {
  std::cout << (*it) << std::endl;
}
```

Iterators

Why do we care?

```
DFS dfs(...);
for ( ImageTraversal::Iterator it = dfs.begin(); it != dfs.end(); ++it ) {
  std::cout << (*it) << std::endl;
}
```

```
DFS dfs(...);
for ( const Point & p : dfs ) {
  std::cout << p << std::endl;
}
```

Iterators

Why do we care?

```
DFS dfs(...);
for ( ImageTraversal::Iterator it = dfs.begin(); it != dfs.end(); ++it ) {
  std::cout << (*it) << std::endl;
}
```

```
DFS dfs(...);
for ( const Point & p : dfs ) {
  std::cout << p << std::endl;
}
```

```
ImageTraversal & traversal = /* ... */;
for ( const Point & p : traversal ) {
  std::cout << p << std::endl;
}
```

Iterators

```
ImageTraversal *traversal = /* ... */;
for ( const Point & p : traversal ) {
  std::cout << p << std::endl;
}
```

Every Data Structure So Far

	Unsorted Array	Sorted Array	Unsorted List	Sorted List	Binary Tree	BST	AVL
Find							
Insert							
Remove							
Traverse							

Range-based Searches

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

…what points fall in [11, 42]?

Tree construction:

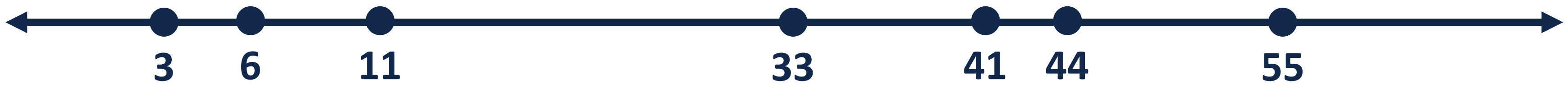
Range-based Searches

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

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

…what points fall in [11, 42]?

Ex:

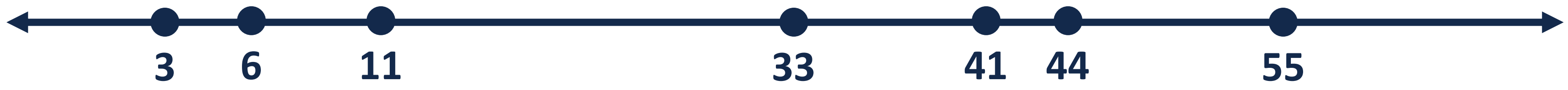


Range-based Searches

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

…what points fall in [11, 42]?

Ex:



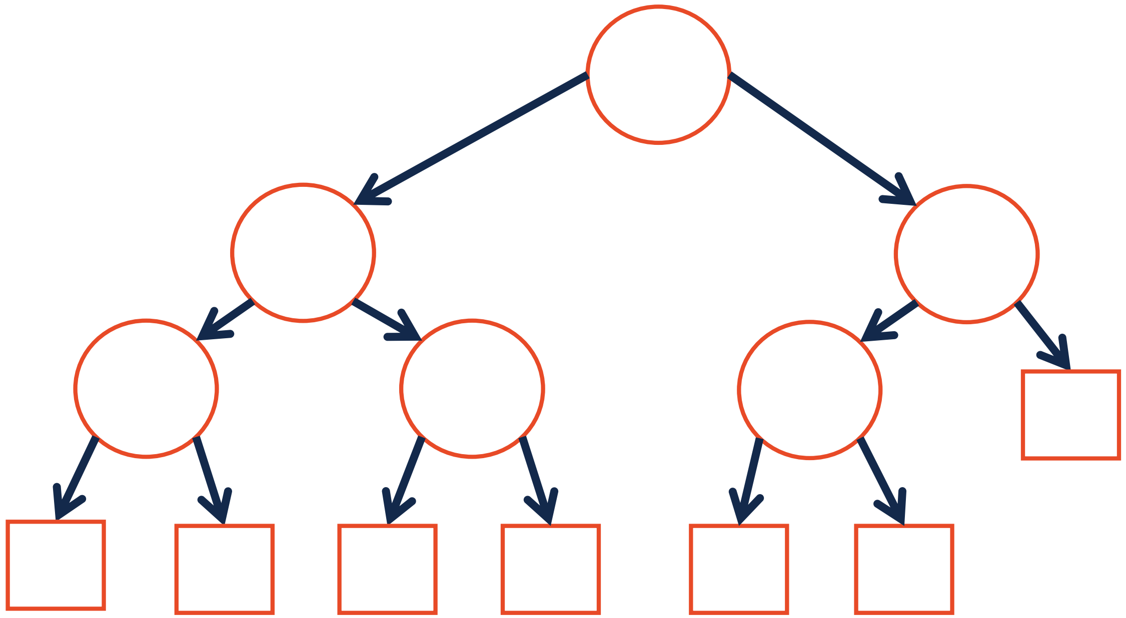
Range-based Searches

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

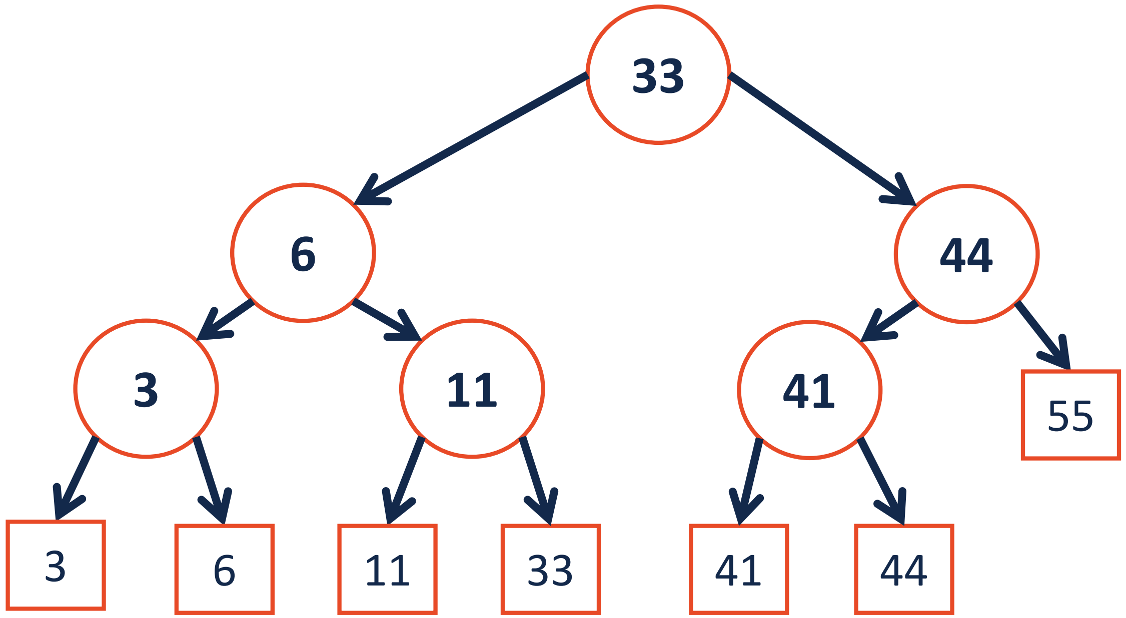
…what points fall in [11, 42]?

Tree construction:

Range-based Searches

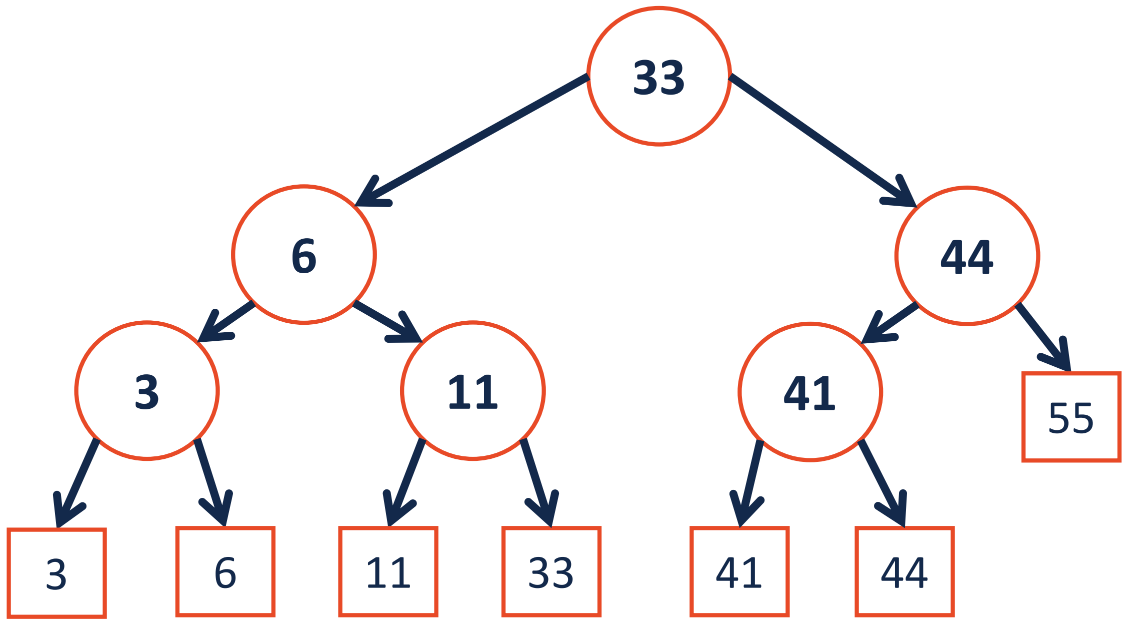


Range-based Searches

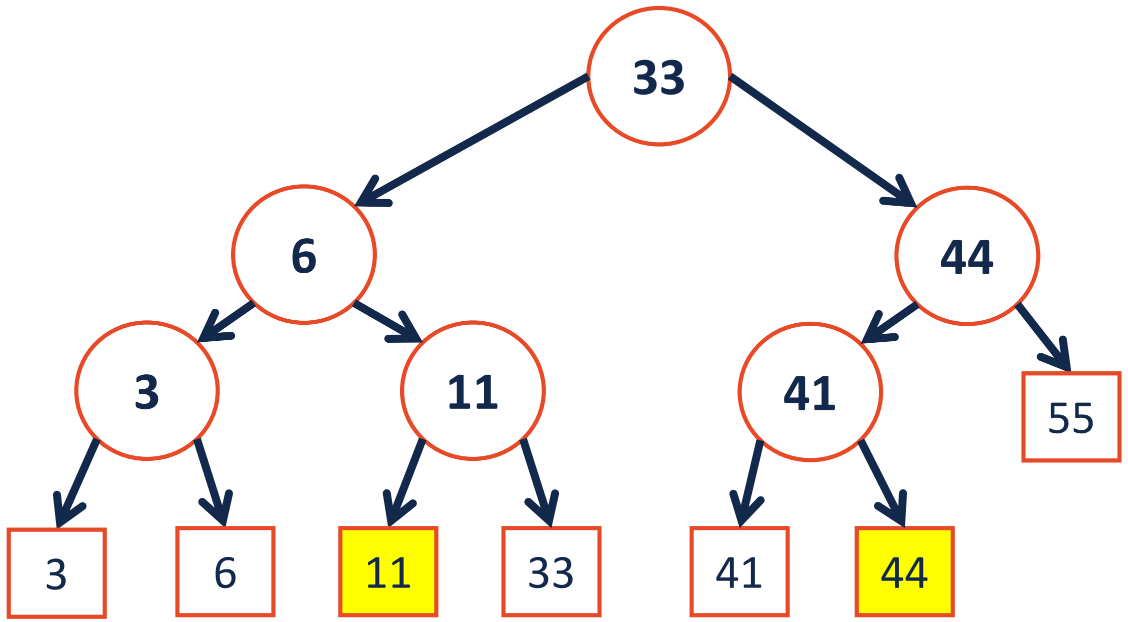


Range-based Searches

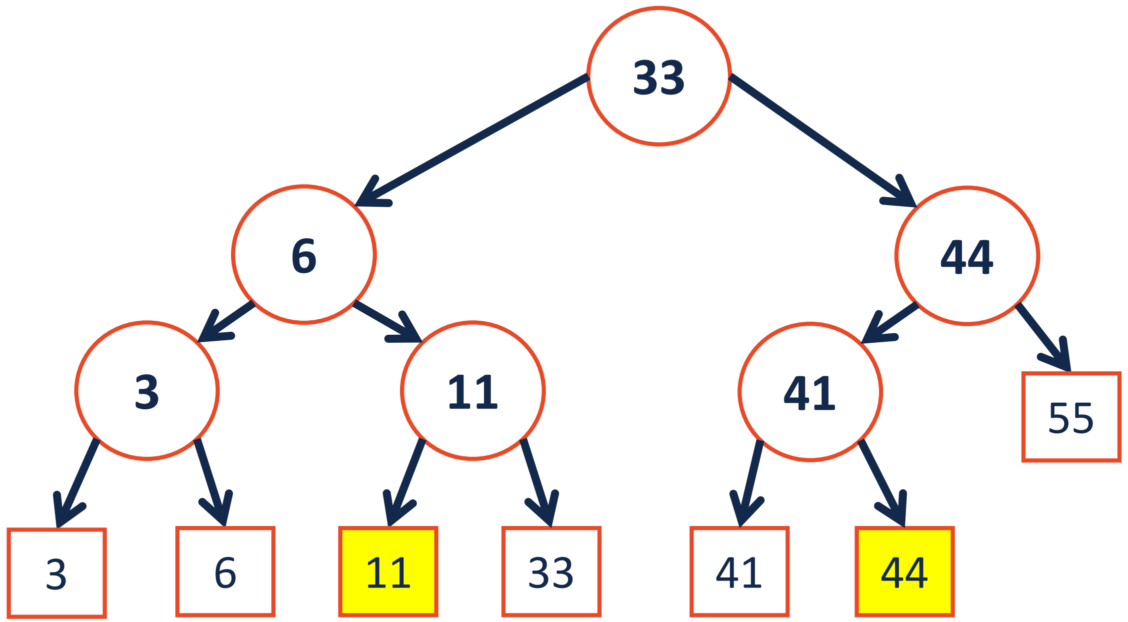
Q: Consider points in 1D: $\mathbf{p = \{p_1, p_2, \ldots, p_n\}}$.
…what points fall in [11, 42]?



Range-based Searches



Running Time

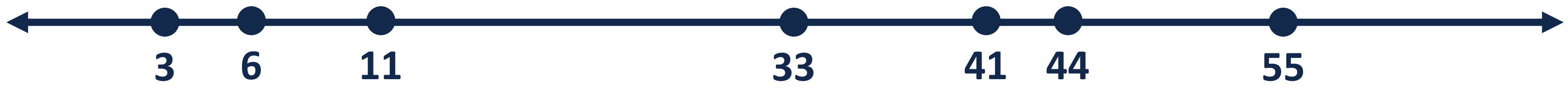


Range-based Searches

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

…what points fall in [11, 42]?

Ex:



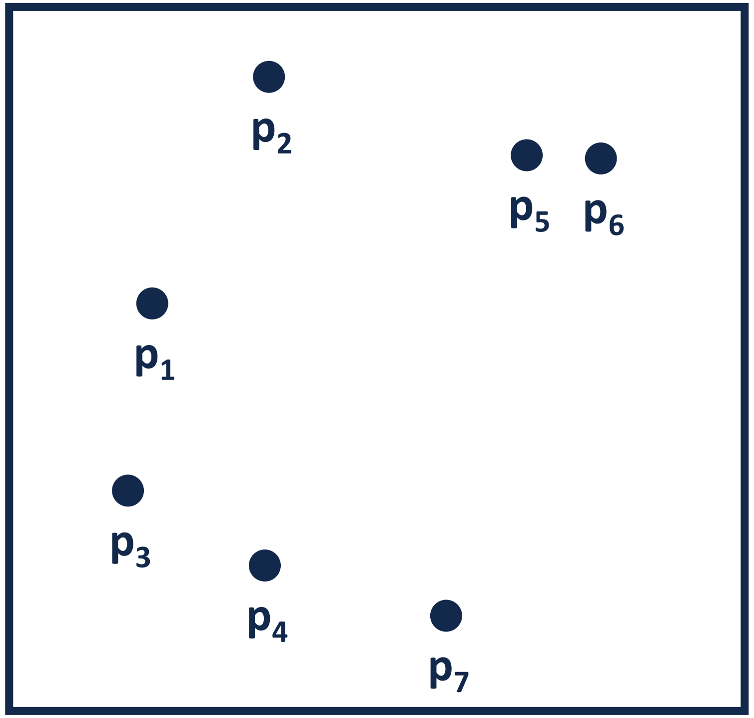
Range-based Searches

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

Q: What points are in the rectangle:

$[\ (x_1, y_1),\ (x_2, y_2)\ ]$?

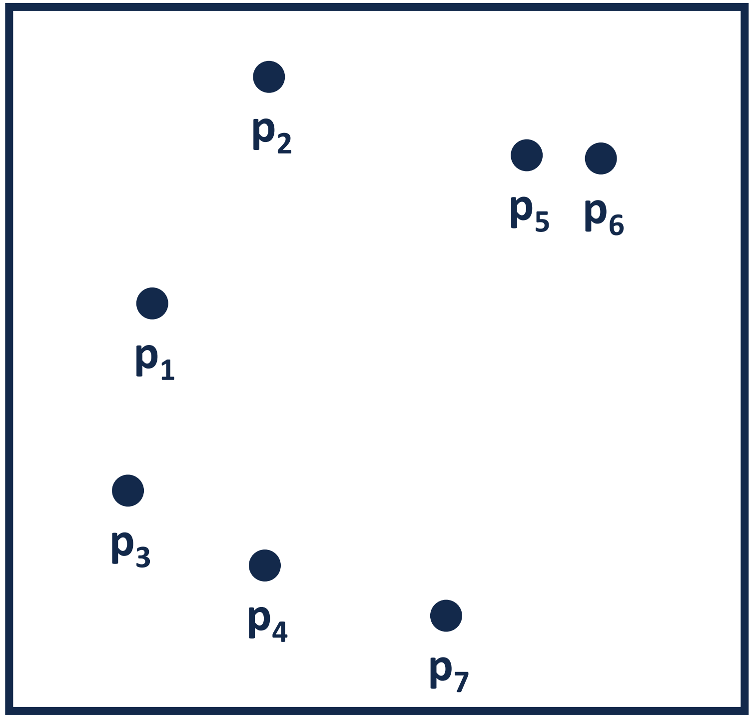
Q: What is the nearest point to (x_1, y_1)?



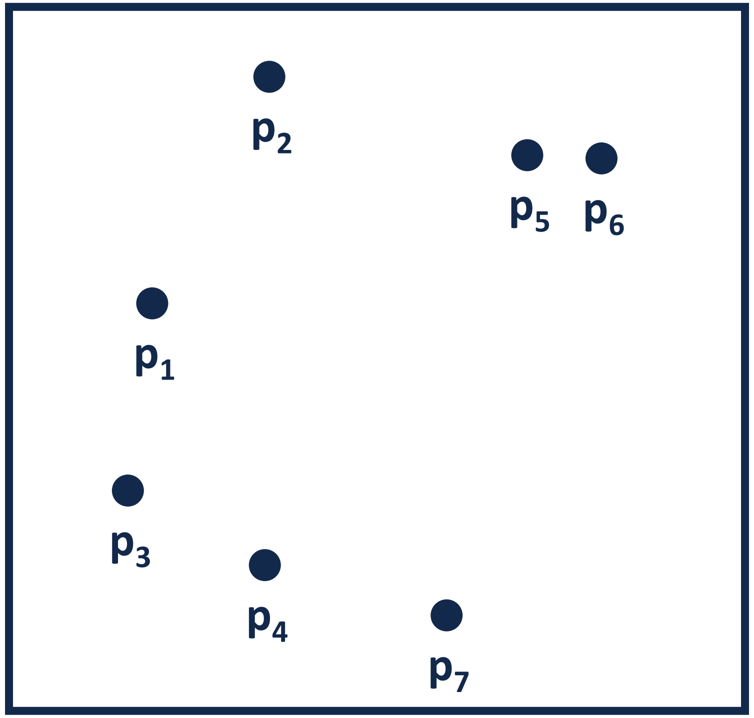
Range-based Searches

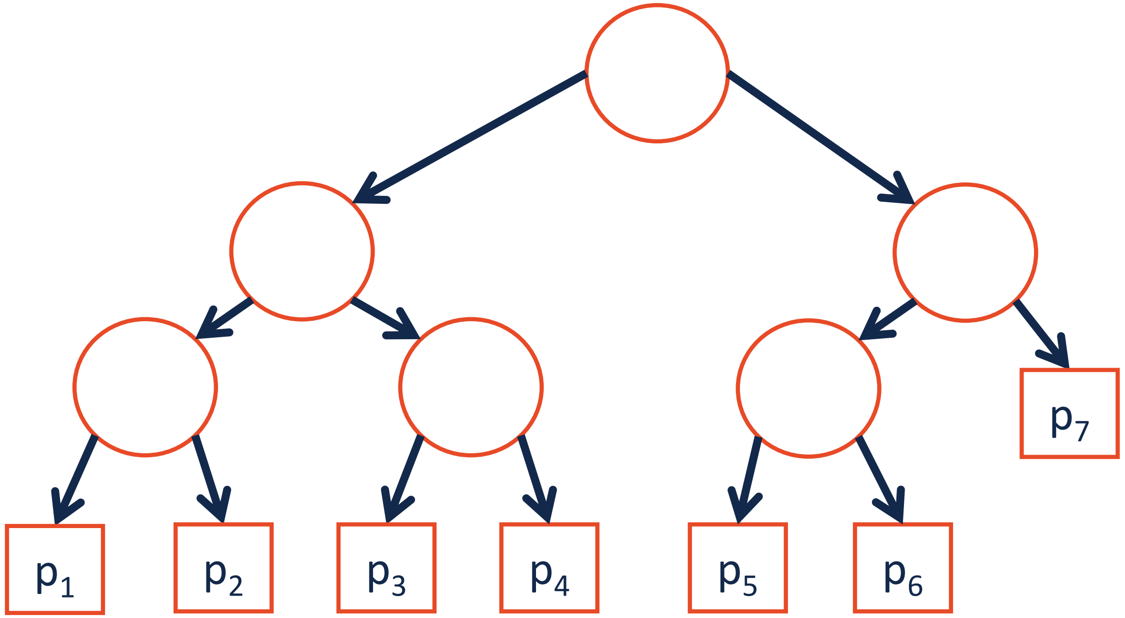
Consider points in 2D: $\mathbf{p} = \{\mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_n}\}$.

Tree construction:

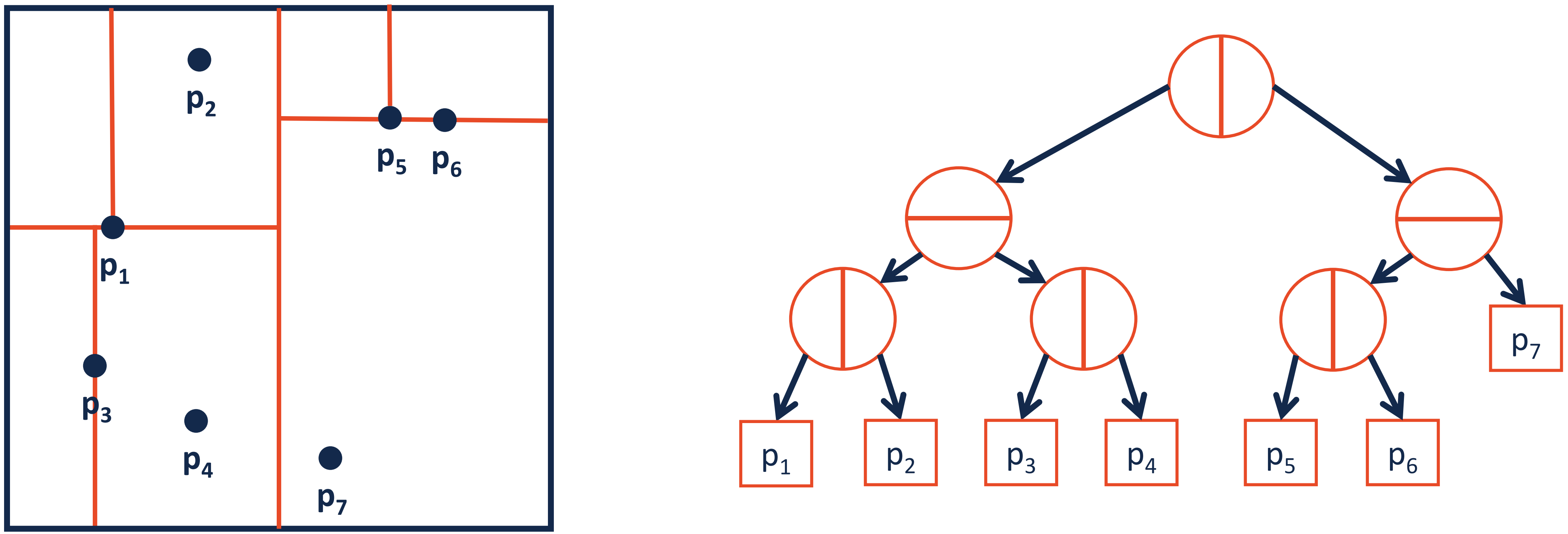


Range-based Searches





kD-Trees



kD-Trees