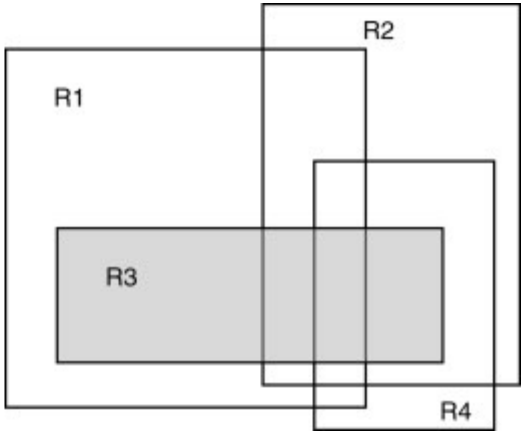
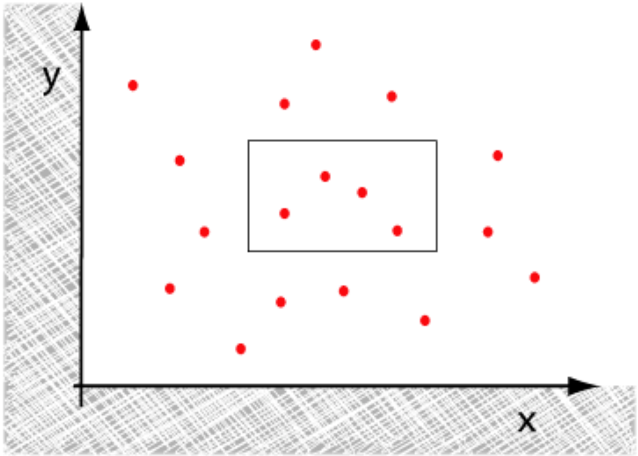
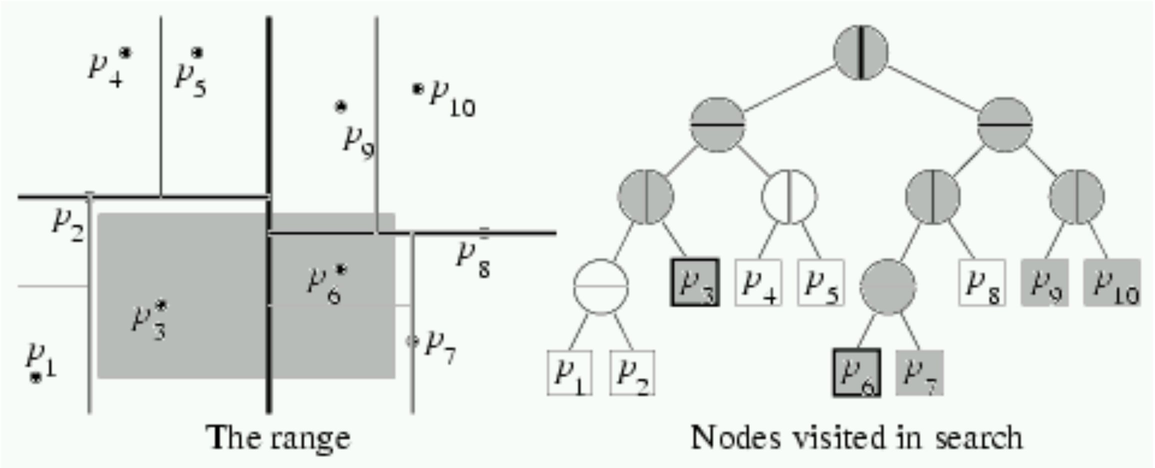


www.sciencedirect.com



www.sable.mcgill.ca

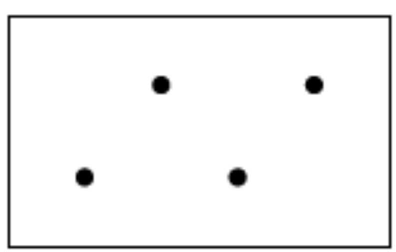
Orthogonal searching



www.cs.wustl.edu

Problem definition

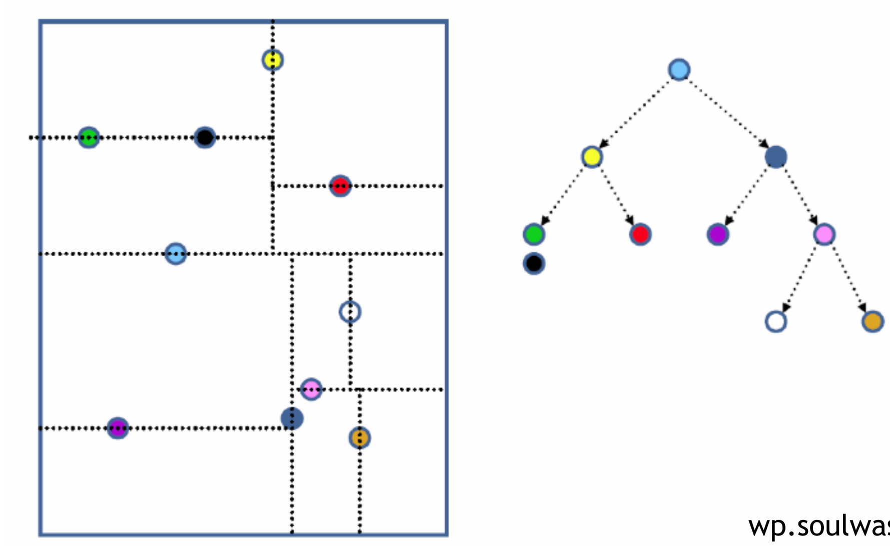
- Lets consider a finite set of points *P*. The goal is to find a structure enabling efficient search for points in a given range.
- E.g., in 2D rectangle:



$[x_1, x'_1], [x_2, x'_2]$

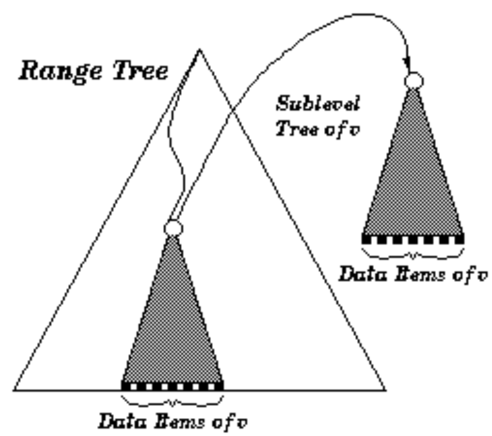
Solution

- k-D trees



wp.soulwasted.net

- Range trees



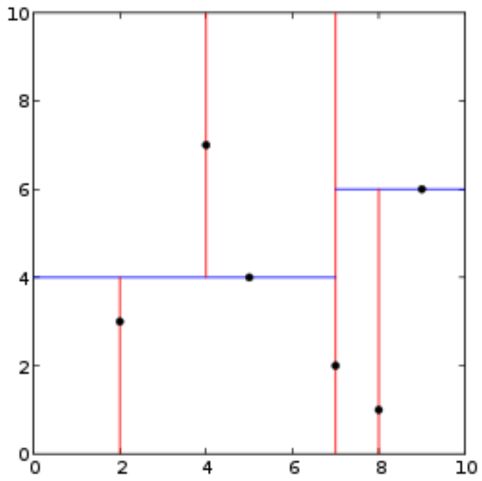
graphics.stanford.edu

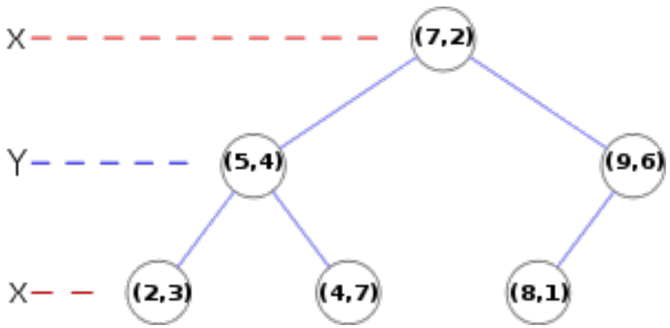
k-D trees

- Usage – GIS, computer graphics, databases
- By dividing the space we create a binary tree. Its inner nodes contain the dividing axis and two pointers, leaves contain the data
- Disadvantages
  - Sensitive to the order of points entering the structure

k-D trees

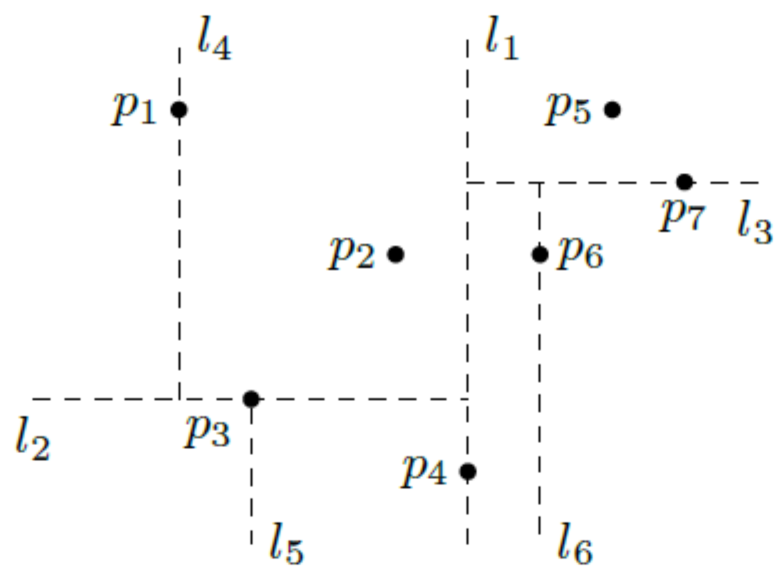
- Initial requirement: any two points from *P* don't have the same x or y axis (this requirement can be later removed)
- We build the tree by alternating the division by x and y axis





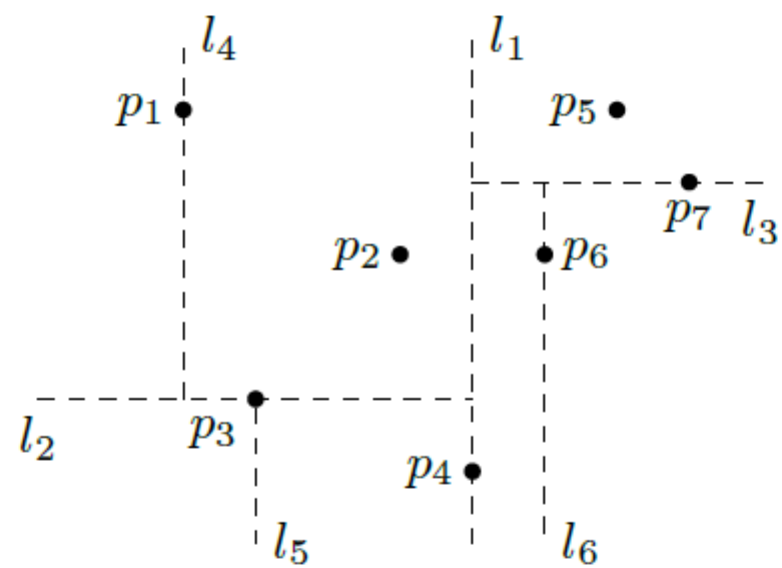
k-D trees

- Line l_1 intersects with p_4, which lies in the center of the set of points sorted according to x axis
- This divides the spac to two half-planes, in each of them we divide according to axis using the same criterion

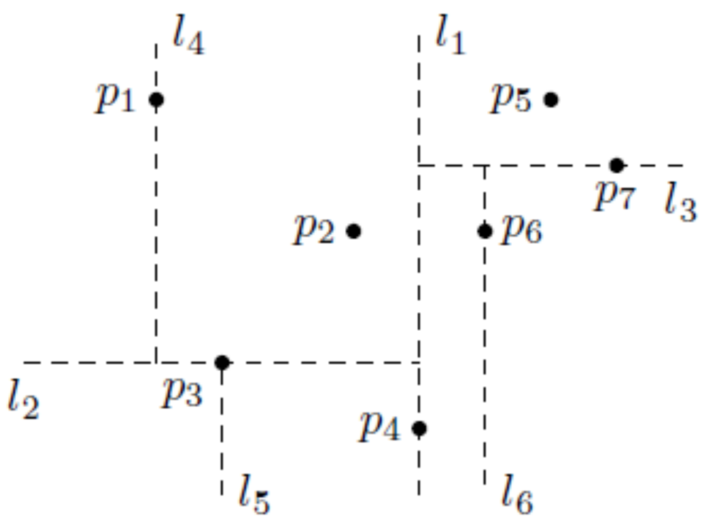


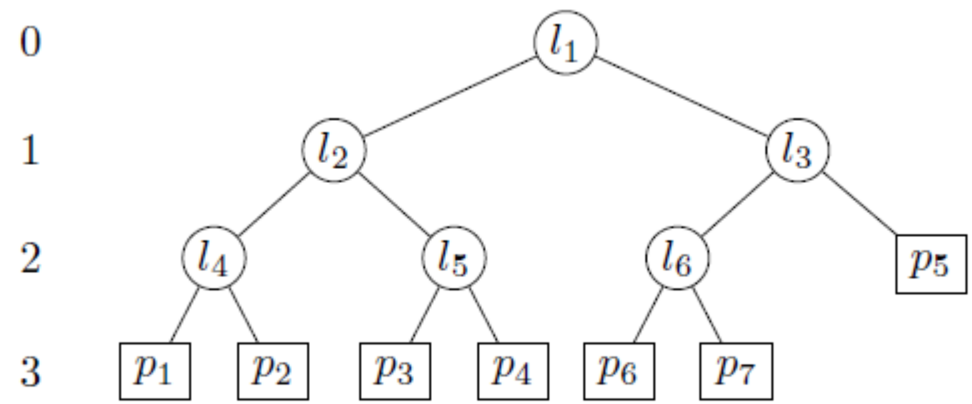
k-D trees

- Lines l_2, l_3 intersect points lying in the middle of “their” half-planes (according to y axis)
- We divide recursively until the half-planes contain more points
  or until we reach
  a given number of
  iteration (depth of
  the tree)

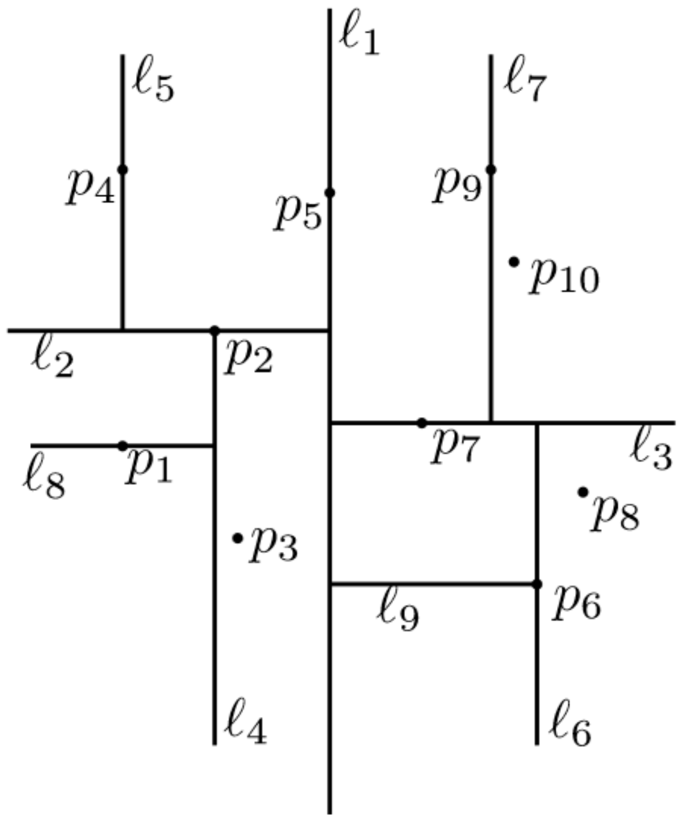


k-D trees

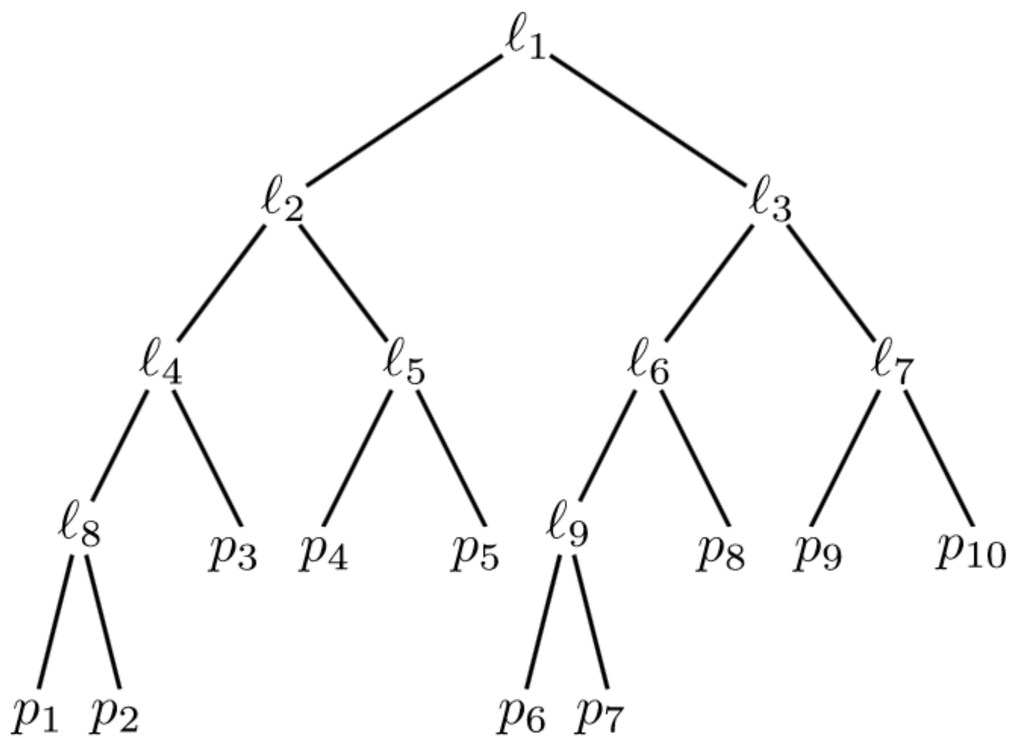

l_4
l_1
p_1
p_5
p_7
l_3
p_2
p_6
l_2
p_3
p_4
l_5
l_6


0
1
2
3
l_1
l_2
l_3
l_4
l_5
l_6
p_5
p_1
p_2
p_3
p_4
p_6
p_7

k-D trees



ℓ_1
ℓ_5
ℓ_7
p_4
p_5
p_9
p_{10}
ℓ_2
p_2
ℓ_8
p_1
p_7
ℓ_3
p_8
p_3
ℓ_9
p_6
ℓ_4
ℓ_6



ℓ_1
ℓ_2
ℓ_3
ℓ_4
ℓ_5
ℓ_6
ℓ_7
ℓ_8
p_3
p_4
p_5
ℓ_9
p_8
p_9
p_{10}
p_1
p_2
p_6
p_7

Pseudocode

Algorithm $\textsc{BuildKdTree}(P, \mathit{depth})$

if P contains only one point
    then return a leaf storing this point
    else if depth is even
        then Split P with a vertical line ℓ through the median x-coordinate into P_1 (left of or on ℓ) and P_2 (right of ℓ)
        else Split P with a horizontal line ℓ through the median y-coordinate into P_1 (below or on ℓ) and P_2 (above ℓ)
        $\nu_{\text{left}} \leftarrow \textsc{BuildKdTree}(P_1, \mathit{depth}+1)$
        $\nu_{\text{right}} \leftarrow \textsc{BuildKdTree}(P_2, \mathit{depth}+1)$
        Create a node ν storing ℓ, make ν_{left} the left child of ν, and make ν_{right} the right child of ν.
        return ν

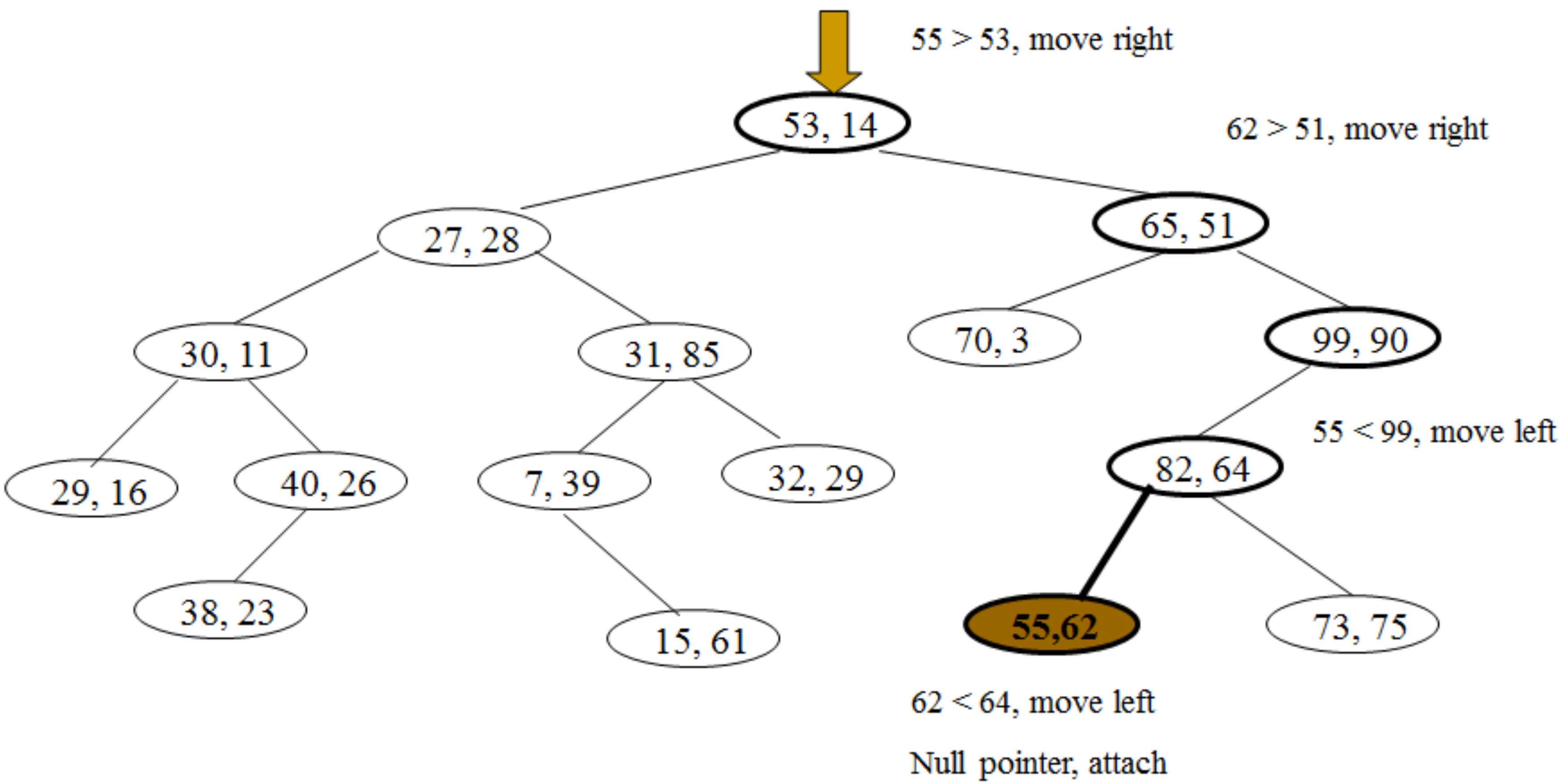
k-D trees

- Inserting to k-D tree:

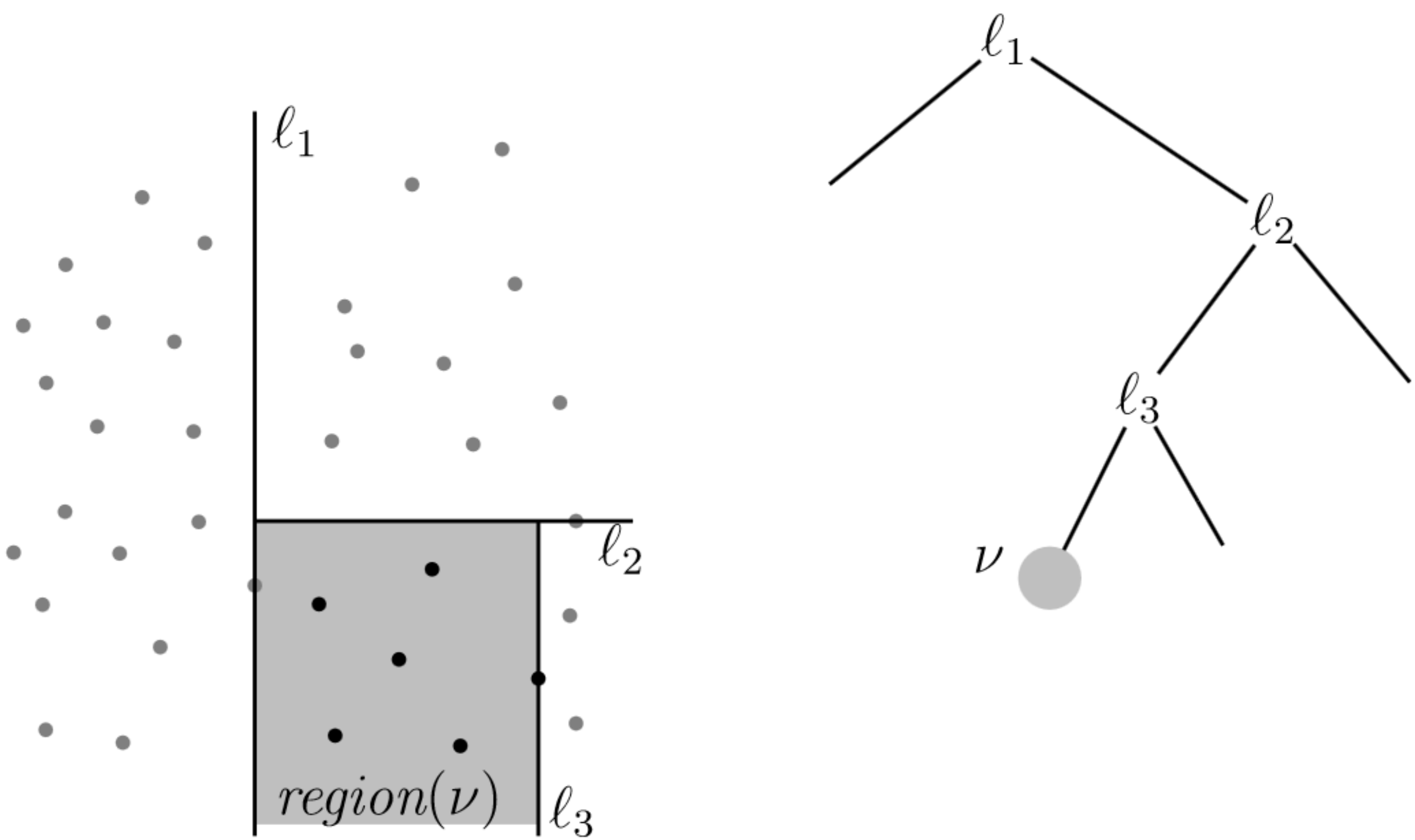
```
public void insert(Vector <T> x)
{
        root = insert( x, root, 0);
}

// this code is specific for 2-D trees
private KdNode<T> insert(Vector <T> x, KdNode<T> t, int level)
{
        if (t == null)
           t = new KdNode(x);

        int compareResult = x.get(level).compareTo(t.data.get(level));
        if (compareResult < 0)
           t.left = insert(x, t.left, 1 - level);
        else if( compareResult > 0)
           t.right = insert(x, t.right, 1 - level);
        else
                 ;  // do nothing if equal

        return t;
}
```

k-D trees

- Inserting node (55, 62)



Region



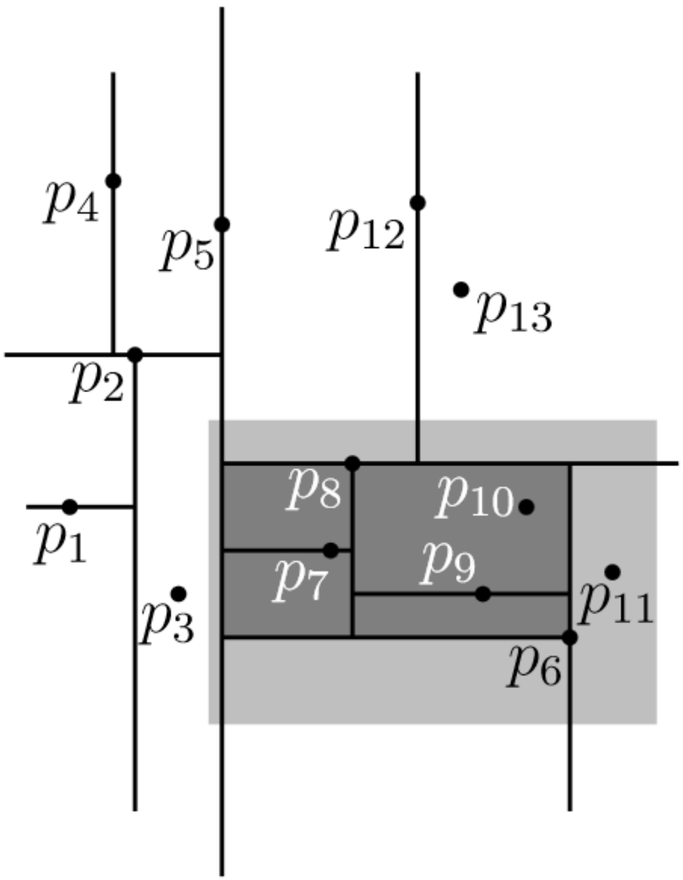
k-D trees

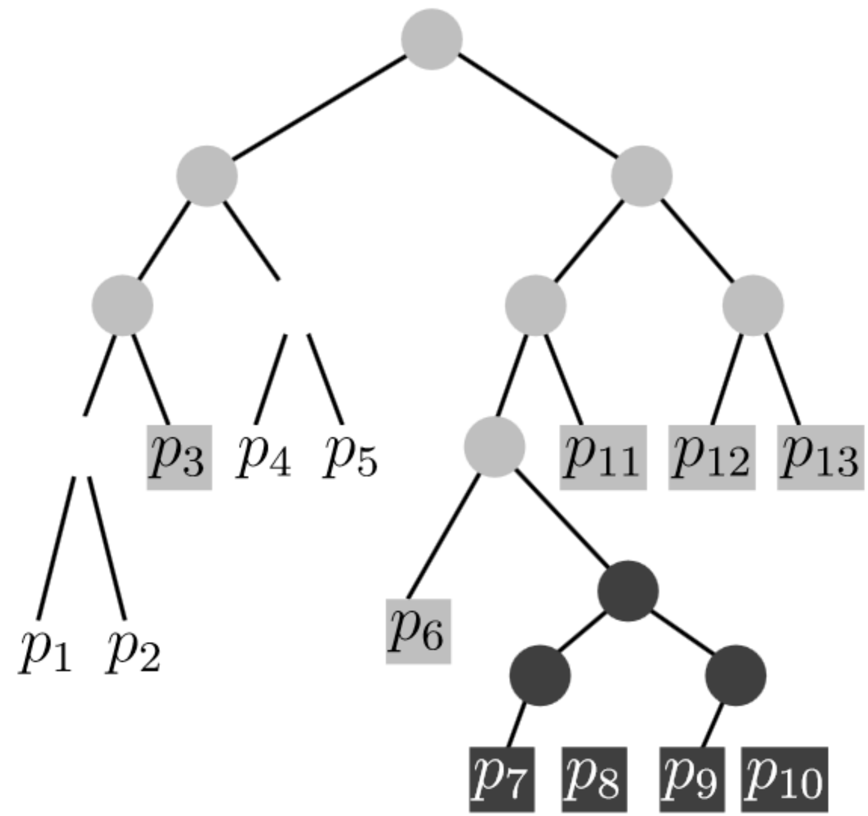
- Searching for a given range:

```
/**
 * Print items satisfying
 * lowRange.get(0) <= x.get(0) <= highRange.get(0)
 * and
 * lowRange.get(1) <= x.get(1) <= highRange.get(1)
 */
public void printRange(Vector <T> lowRange,
                                Vector <T>highRange)
{
    printRange(lowRange, highRange, root, 0);
}
```

```
private void
printRange(Vector <T> low,Vector <T> high,
                                        KdNode<T> t, int level)
{
   if (t != null)
   {
      if ((low.get(0).compareTo(t.data.get(0)) <= 0  &&
               t.data.get(0).compareTo(high.get(0)) <=0)
          &&(low.get(1).compareTo(t.data.get(1)) <= 0 &&
        t.data.get(1).compareTo(high.get(1)) <= 0))
        System.out.println("(" + t.data.get(0) + "," +
                                         t.data.get(1) + ")");
      if (low.get(level).compareTo(t.data.get(level)) <= 0)
               printRange(low, high, t.left, 1 - level);
      if (high.get(level).compareTo(t.data.get(level)) >= 0)
                printRange(low, high, t.right, 1 - level);
   }
}
```

Range search





k-D trees

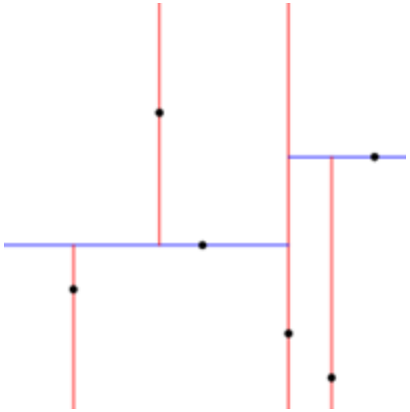
- Complexity:
  - Building k-D tree
    - *O(n log n)*
    - Memory complexity *O(n)*
  - Search
    - $O(n^{1-1/d} + k)$, where *d* is dimension, *k* is the number of nodes in a given query range [x,x’] x [y, y’]

k-D trees

- Removing node from k-D tree
  - Efficient solution doesn’t exist, a node is marked as deleted
- Balancing k-D tree
  - Any known strategy ensuring the balance of 2-D tree
  - Can be reached by repeated balancing the tree

Assignment

- Implement k-D tree to the basic framework and visualize the dividing lines



Implementation

- **KdNode**:
  - int k = 2; // dimensionality
  - int depth = 0; // current depth
  - Point id = null; // point representation
  - KdNode parent = null; // pointer to parent node
  - KdNode lesser = null; // pointer to left child
  - KdNode greater = null; // pointer to right child

Implementation

- **Point**
  - double x;
  - double y;
- **Store the results, e.g., to:**
  - TreeSet<KdNode> results;

- The comparator of points should be implemented using the Euclidean distance