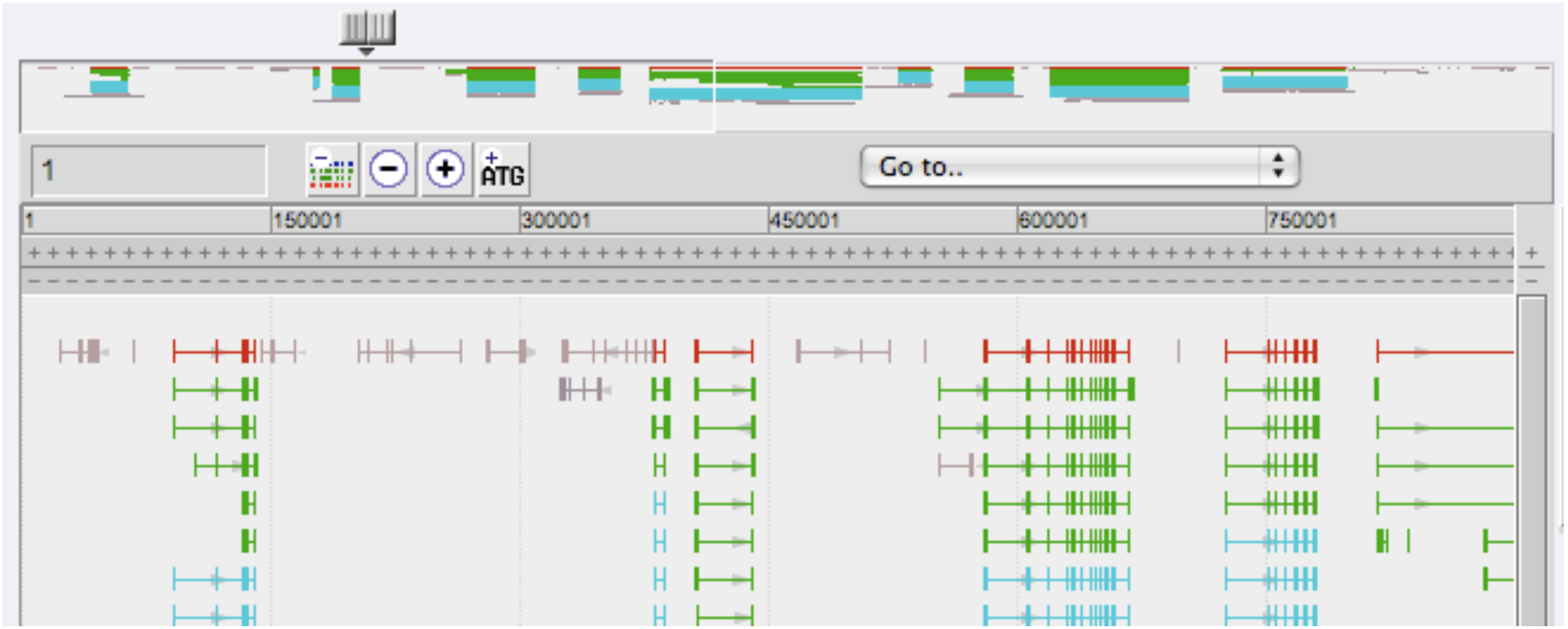
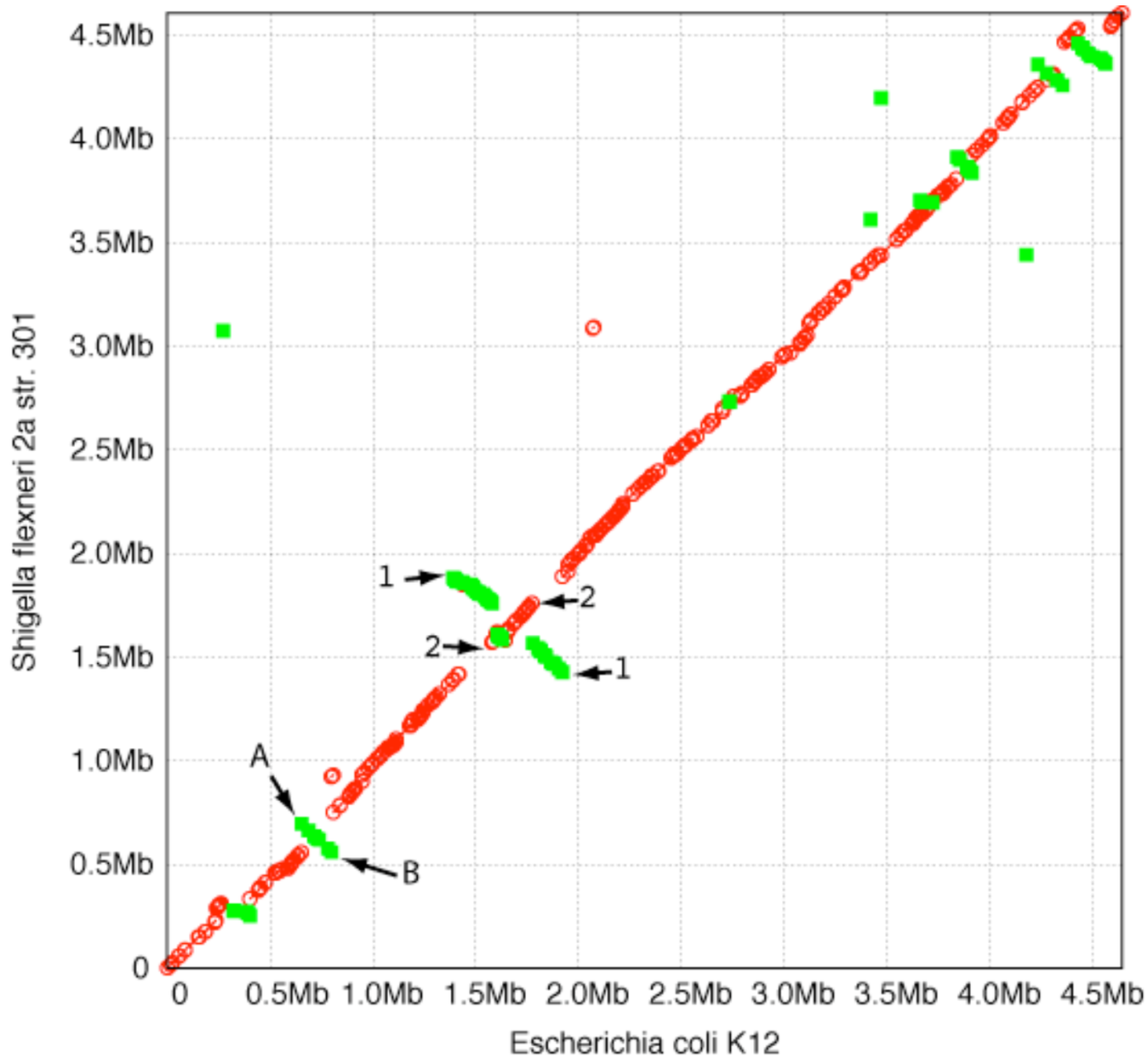
Project Part 2a: DS for Genome Browser



- Biologists want to be able to browse and search all the features of the genome
- We're considering only genes, but there are lots more: implementation is similar
- Examples:
  - ENCODE region browser
  - Bacterial Browser
  - USCD Genome Browser

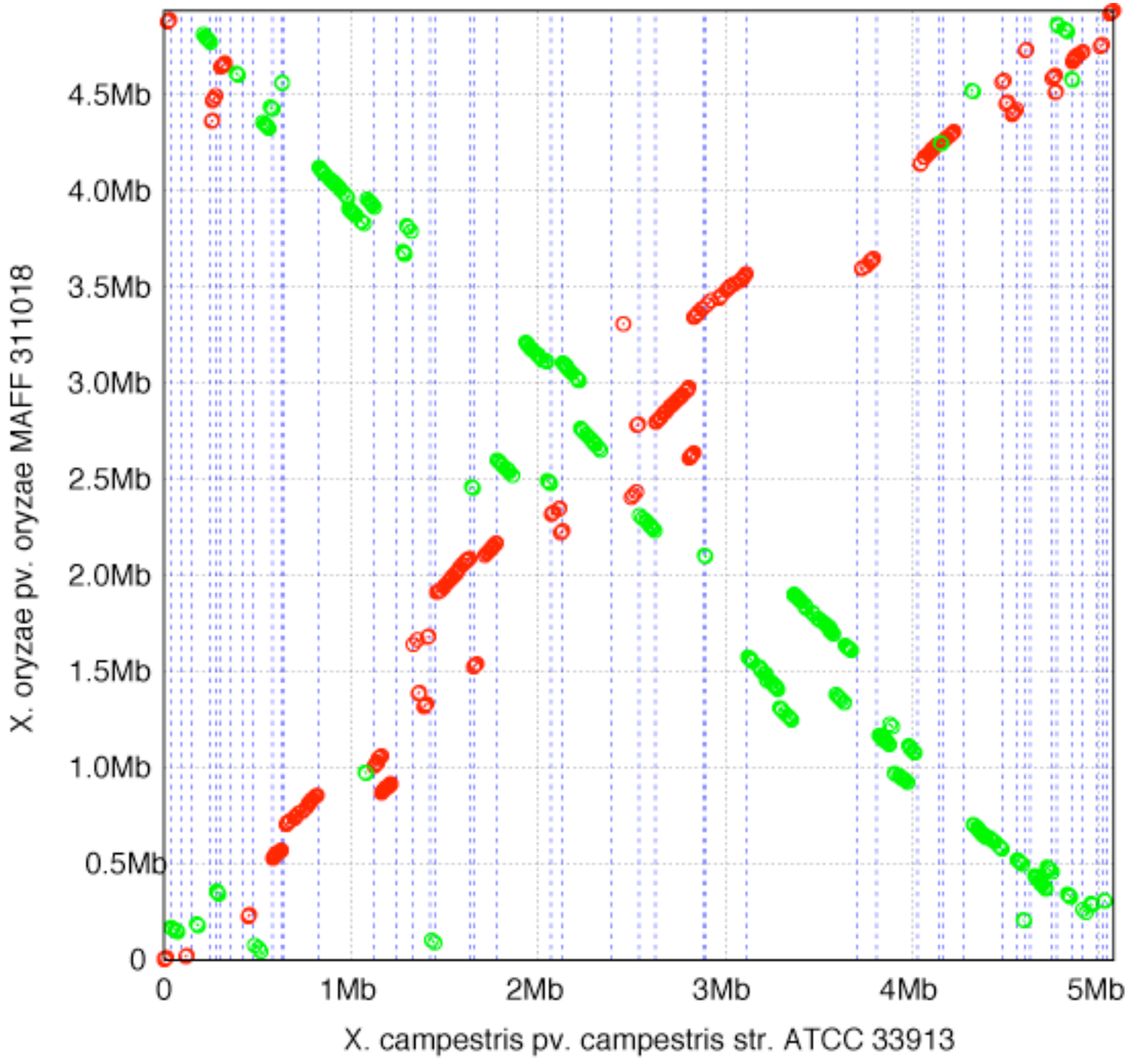
Project Part 2b: DS for comparing genomes

- Overtime, genes can move in genome
- MUMMER is a tool developed here to compare two genomes:
  - Places a dot every place a sequence in 1 genome is found in the other genome
  - Uses suffix trees (which we'll talk about soon)
- Project assumes you're given the mapping between places (genes) on the genome & you have to answer region queries



MUMMER: another example

- Genomes more divergent (more shuffling)
- Xanthomonas
  - Bacteria
  - Common plant pathogen



Range Trees

1-Dimensional Range Trees

- Suppose you have "points" in 1-dimension (aka numbers)
- Want to answer range queries: "Return all keys between x_1 and x_2."
- How could you solve this?

Balanced Binary Search Tree

Range Queries on Binary Search Trees

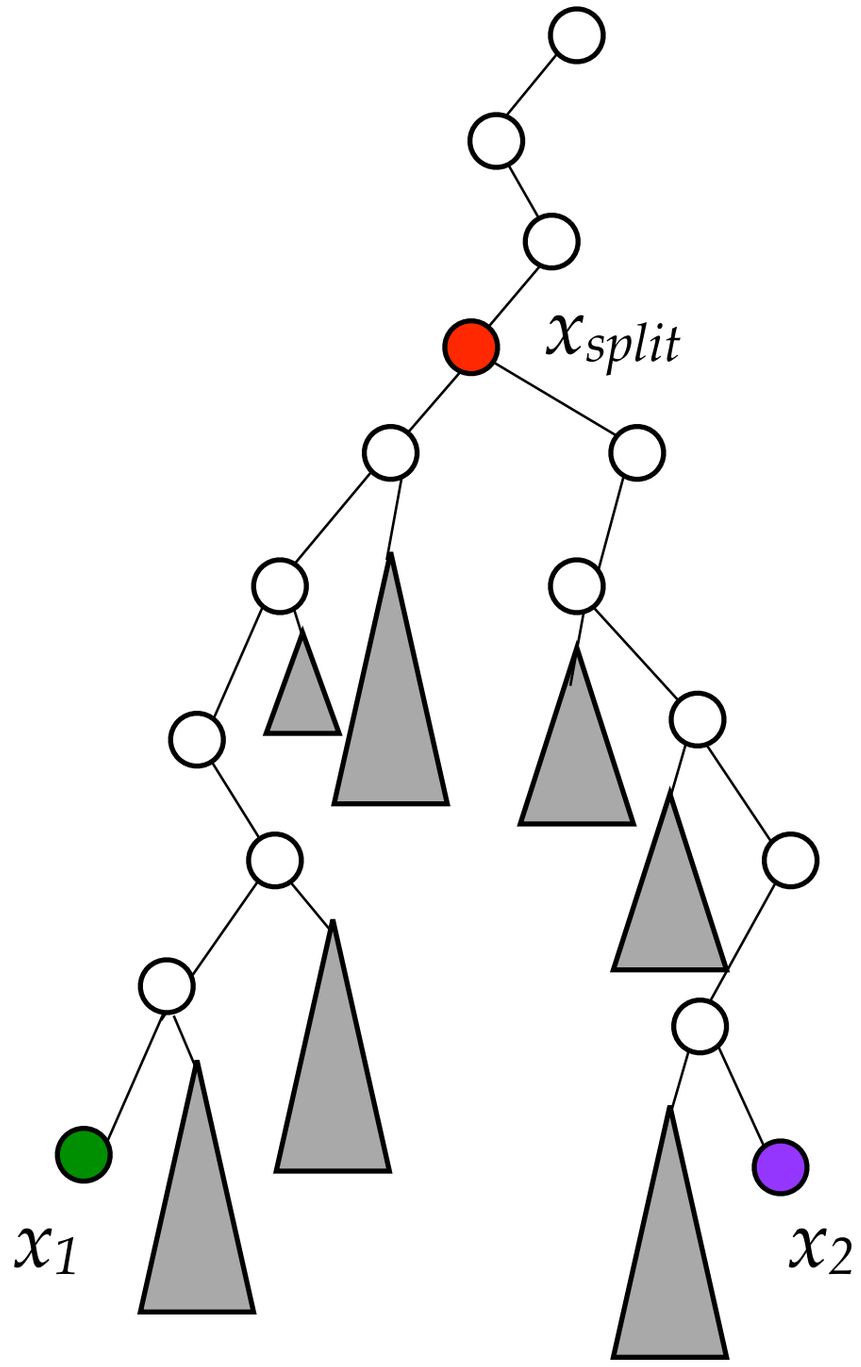
Assume all data are in the leaves

Search for x_1 and x_2

Let x_{split} be the node were the search paths diverge

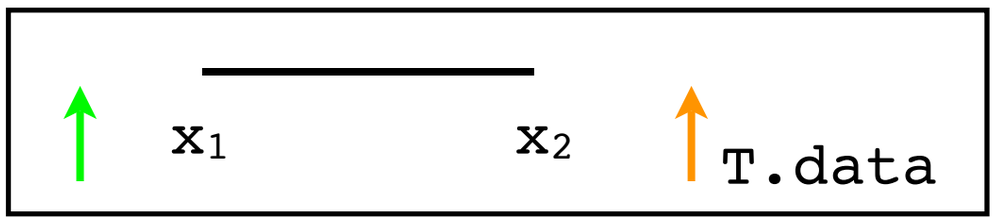
Output leaves in the right subtrees of nodes on the path from x_{split} to x_1

Output leaves in the left subtrees of nodes on the path from x_{split} to x_2



```
OneDRange(T, x1, x2):
  // walk until we find x_split
  while not isLeaf(T) and (x2 ≤ T.data or x1 > T.data):
     if x2 ≤ T.data:
       T = T.left
     else:
       T = T.right
  if isLeaf(T):
    if x1 ≤ T.data ≤ x2: output(T.data)
  else:
    v = T
    // walk down from x_split to x1
10: while not isLeaf(v):
      if x1 ≤ v.data:
        output_subtree(v.right)
        v = v.left
      else:
15:     v = v.right

    // repeat lines 10-15,
    // except walk down the path to x2.
    // ... code not shown ...
```



1-D Query Time

- O(k + log n), where k is the number of points output.
  - Tree is balanced, so depth is O(log n)
  - Length of paths to x1 and x2 are O(log n)
  - Therefore visit O(log n) nodes to find the roots of subtrees to output
  - Traversing the subtrees is linear, O(k), in the number of items output.

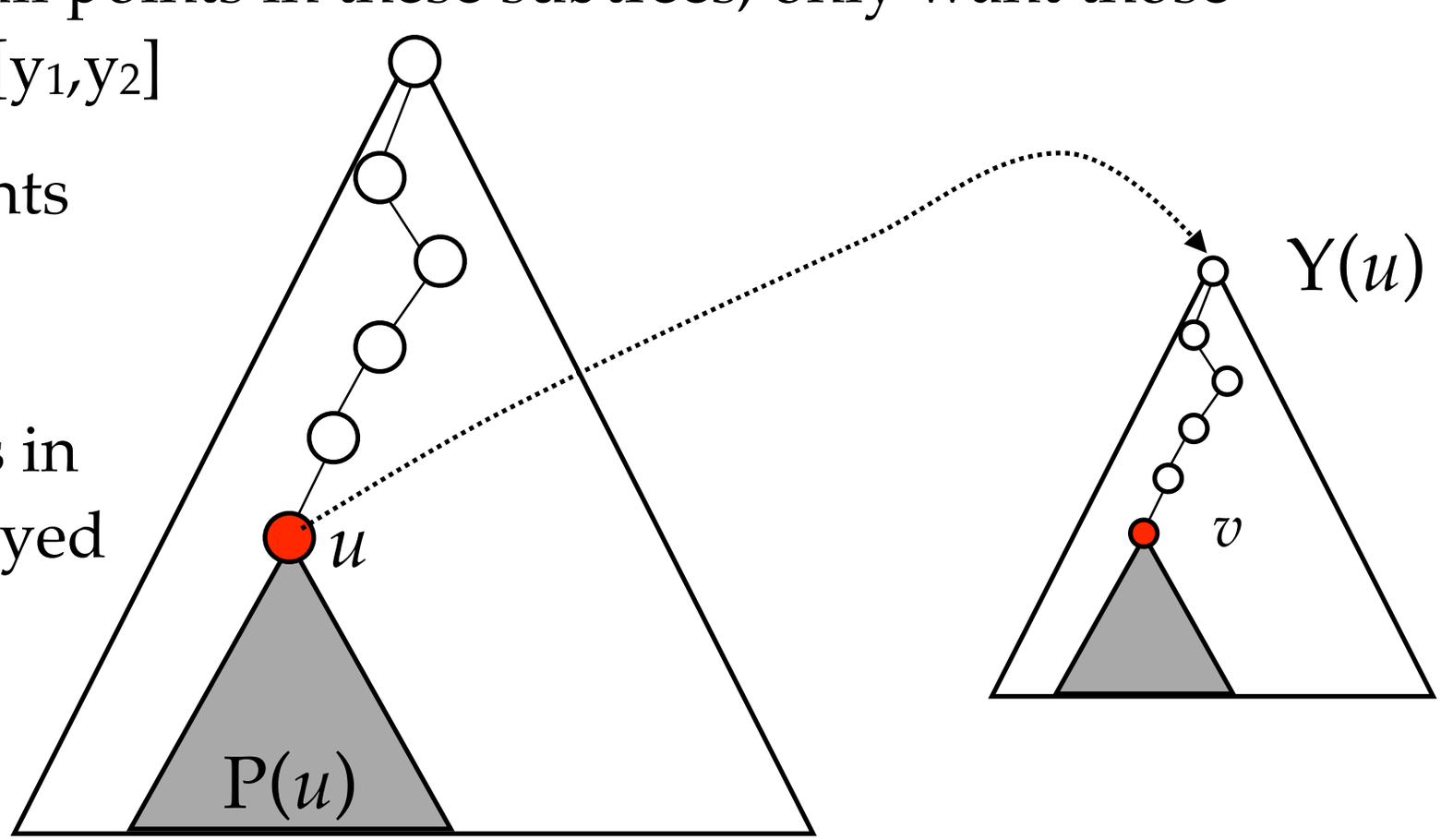
How would you generalize to 2d?

2d Range Trees

- Treat range query as 2 nested one-dimensional queries:
  - $[x_1,x_2]$ by $[y_1,y_2]$
  - First ask for the points with x-coordinates in the given range $[x_1,x_2]$ => a set of subtrees △
  - Instead of all points in these subtrees, only want those that fall in $[y_1,y_2]$

P(u) is the set of points under *u*

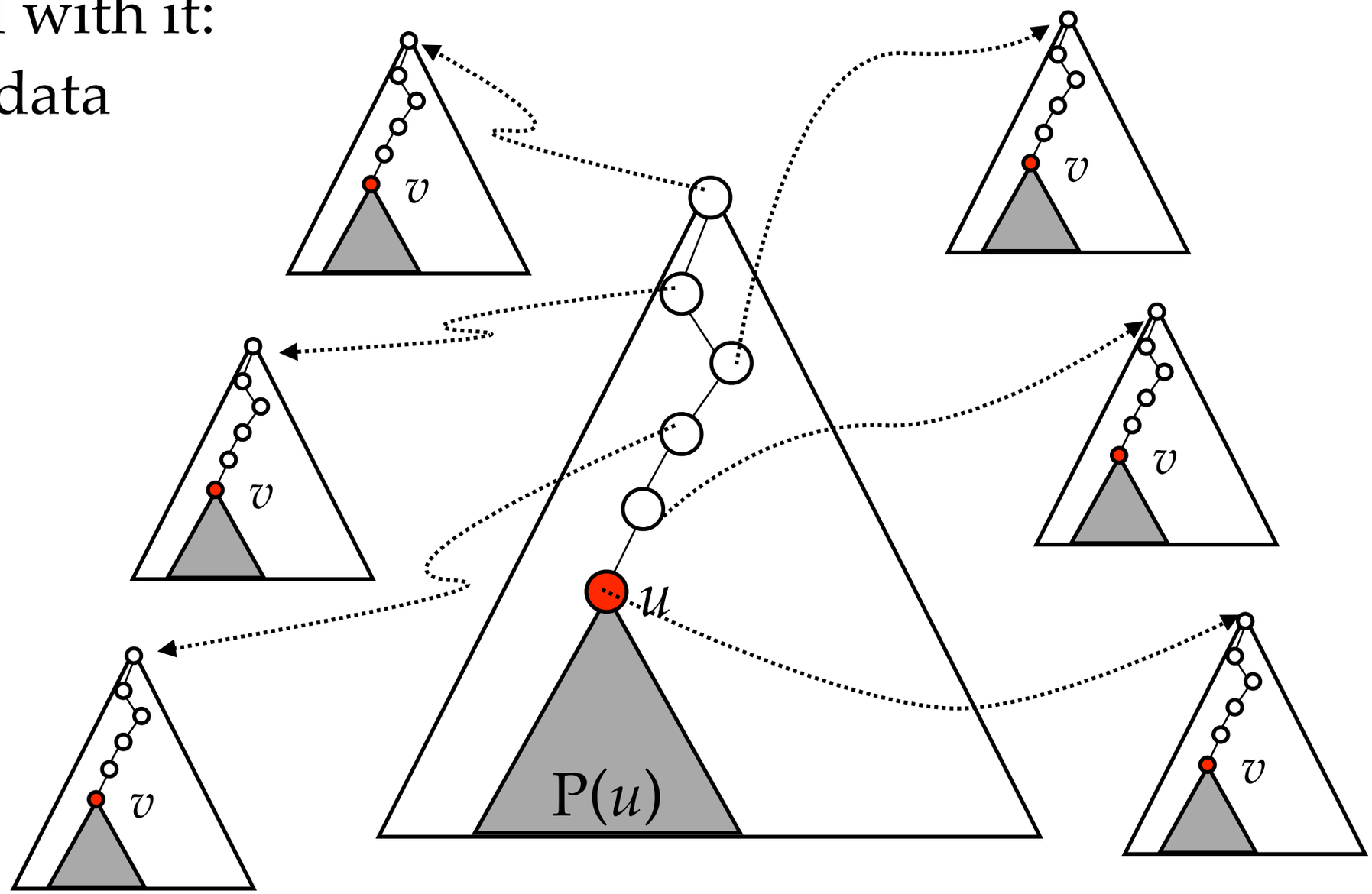
We store *those* points in another tree Y(u), keyed by the y-dimension



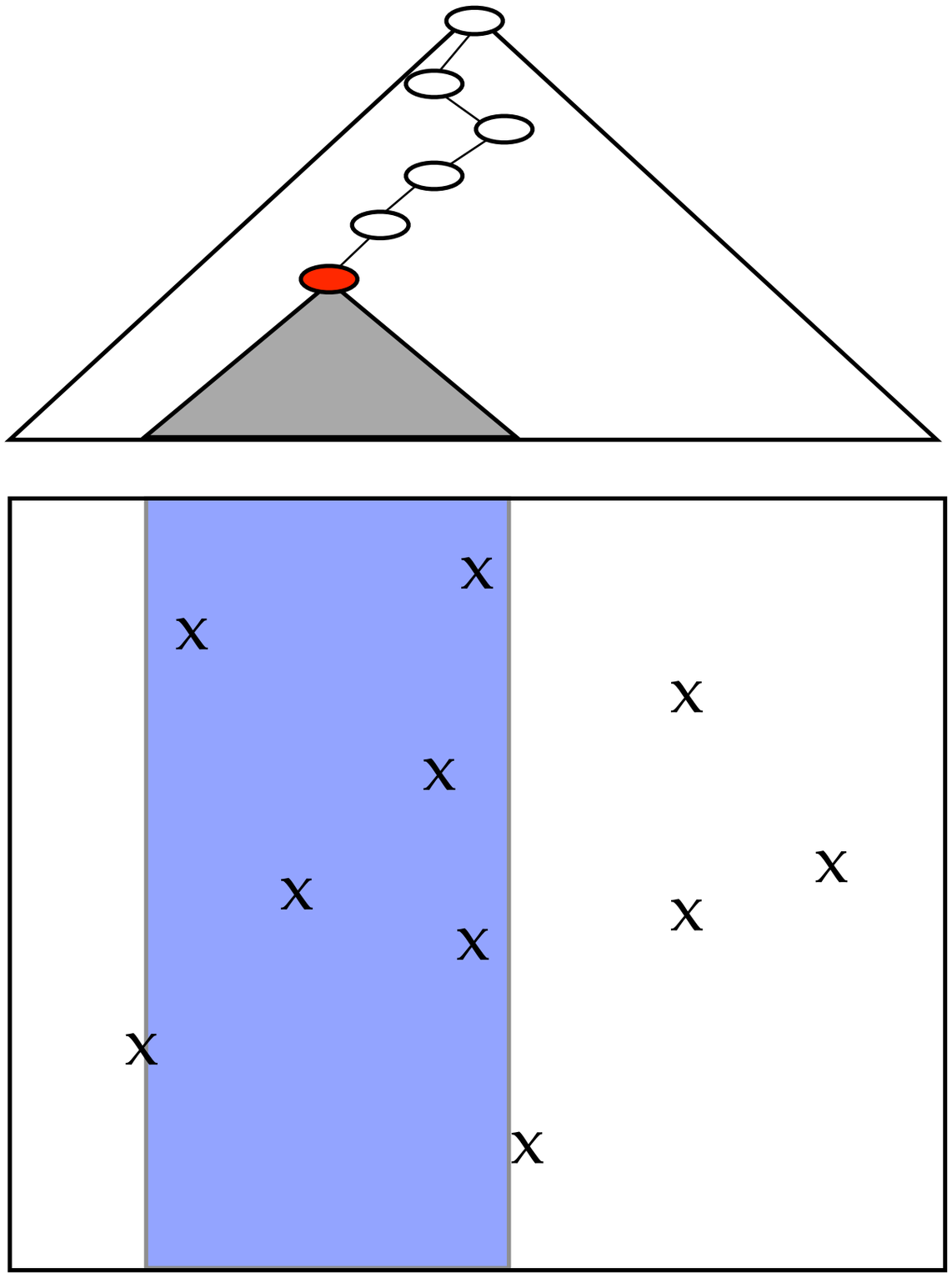
2-D Range Trees, Cont.

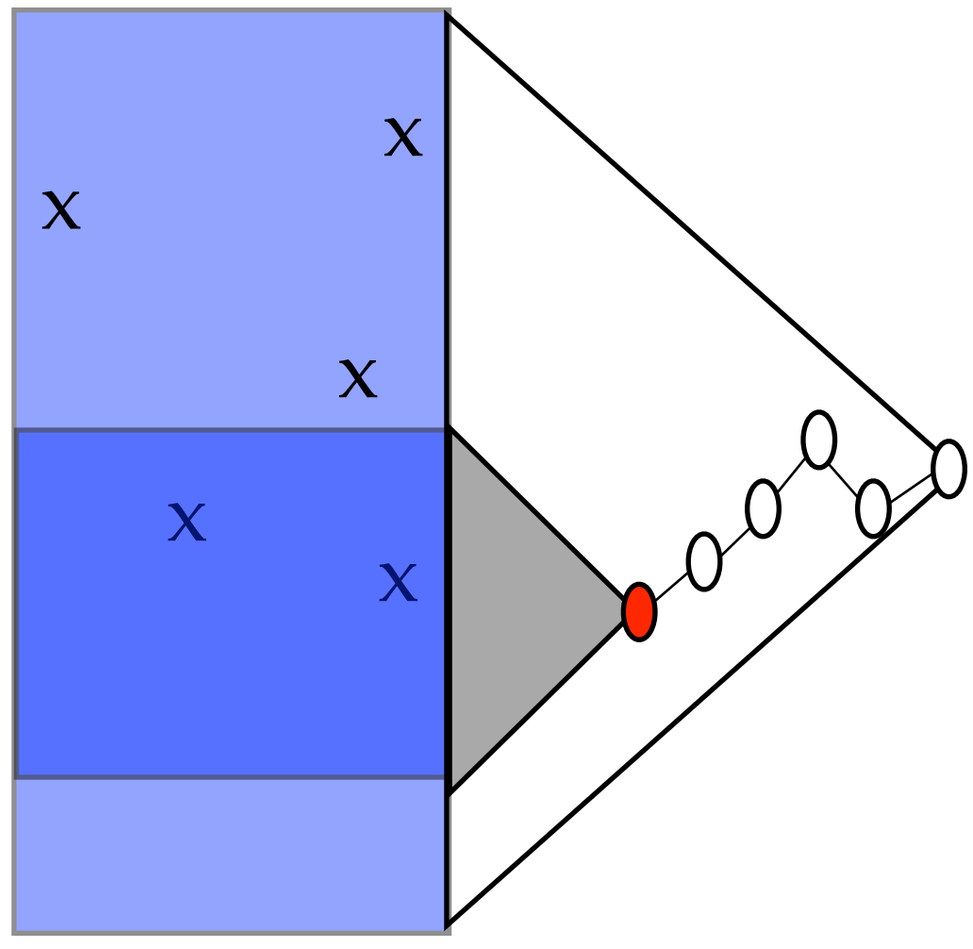
Every node has a tree associated with it:

multilevel data structure



Range Trees, continued.





2d-range tree space requirements

- Sum of the sizes of Y(u) for u at a given depth is O(n)
  - Each point stored in the Y(u) tree for at most one node at a given depth

- Since main tree is balanced, has O(log n) depth

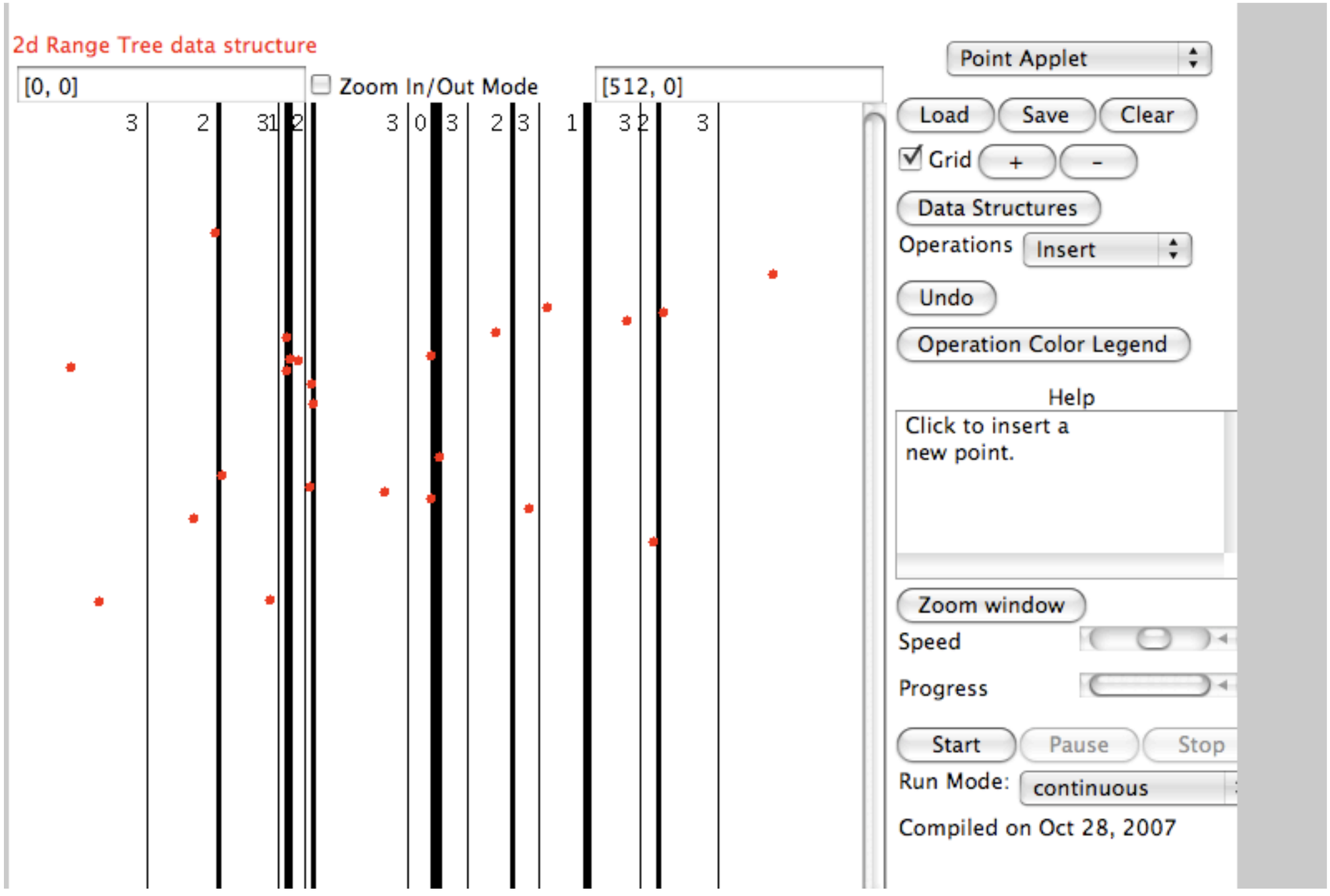
- Meaning total space requirement is O(n log n)

2d Range Tree Range Searches

1. First find trees that match the x-constraint;
2. Then output points in those subtrees that match the y-constraint (by 1-d range searching the associated Y(u) trees)

- Step 1 will return at most O(log n) subtrees to process.
- Step 2 will thus perform the following O(log n) times:
  - Range search the Y(u) tree. This takes $O(\log n + k_u)$, where k_u is the number of points output for that Y(u) tree.
- Total time is $\sum_u O(\log n + k_u)$ where u ranges over O(log n) nodes. Thus the total time is $O(\log^2 n + k)$.

2d Range Tree Demo



kd-tree vs. Range Tree

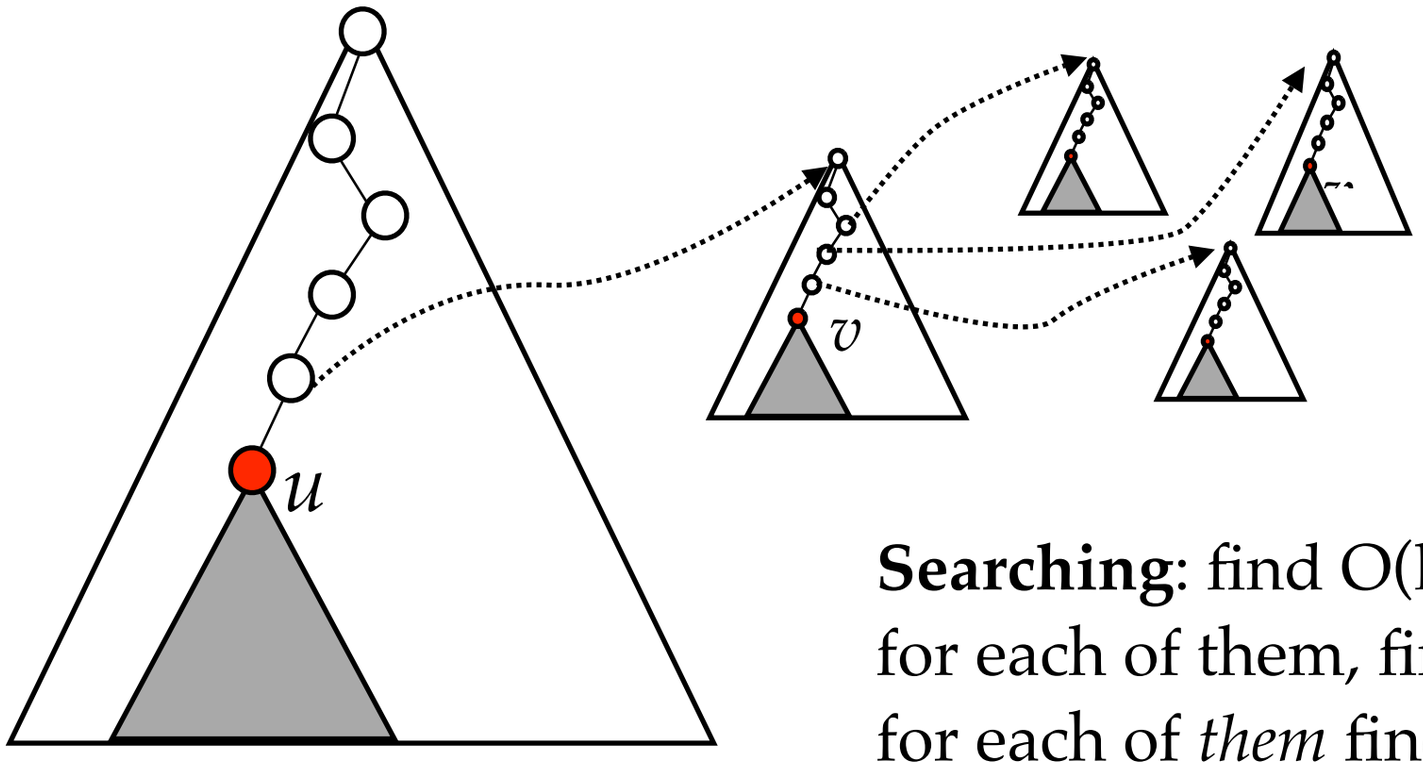
- 2d kd-tree:
  - Space = O(n)
  - Range Query Time = $O(k + \sqrt{n})$
  - Inserts O(log n)

- 2d Range Tree:
  - Space = O(n log n)
  - Range Query Time = $O(k+ \log^2 n)$
  - Inserts $O(\log^2 n)$

How would you extend this to > 2 dimensions?

Range Trees for d > 2

- Now, your associated trees Y(u) themselves have associated trees Z(v) and so on:



Searching: find O(log n) nodes in first tree for each of them, find another O(log n) sets for each of *them* find another log n sets

Leads to $O(k+ \log^d n)$ search time
Space: $O(n \log^{d-1} n)$ space

Fractional Cascading Speed-up: Idea

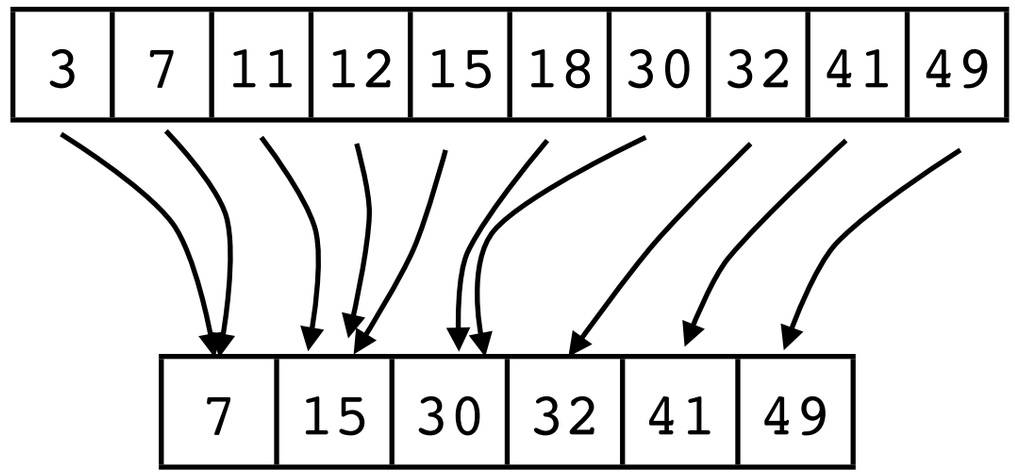
- Suppose you had two sorted arrays A_1 A_2
  - Elements in A_2 are subset of those in A_1
  - Want to range search in both arrays with the same range: $[x_1,x_2]$

- Simple:
  - Binary Search to find x_1 in both A_1 and A_2
  - Walk along array until you pass x_2

- O(log n) time for each Binary Search,
  - have to do it twice though

Can do better:

- Since A_2 subset of A_1:
  - Keep pointer at each element u of A_1 pointing to the smallest element of A_2 that is $\geq u$.

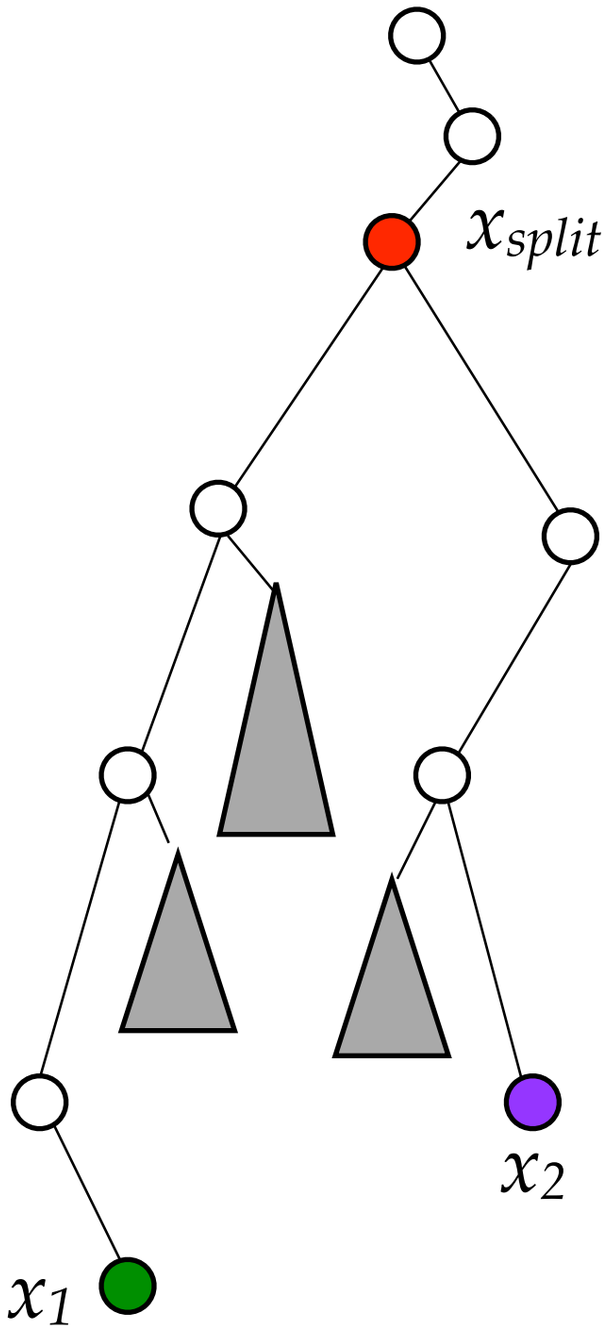


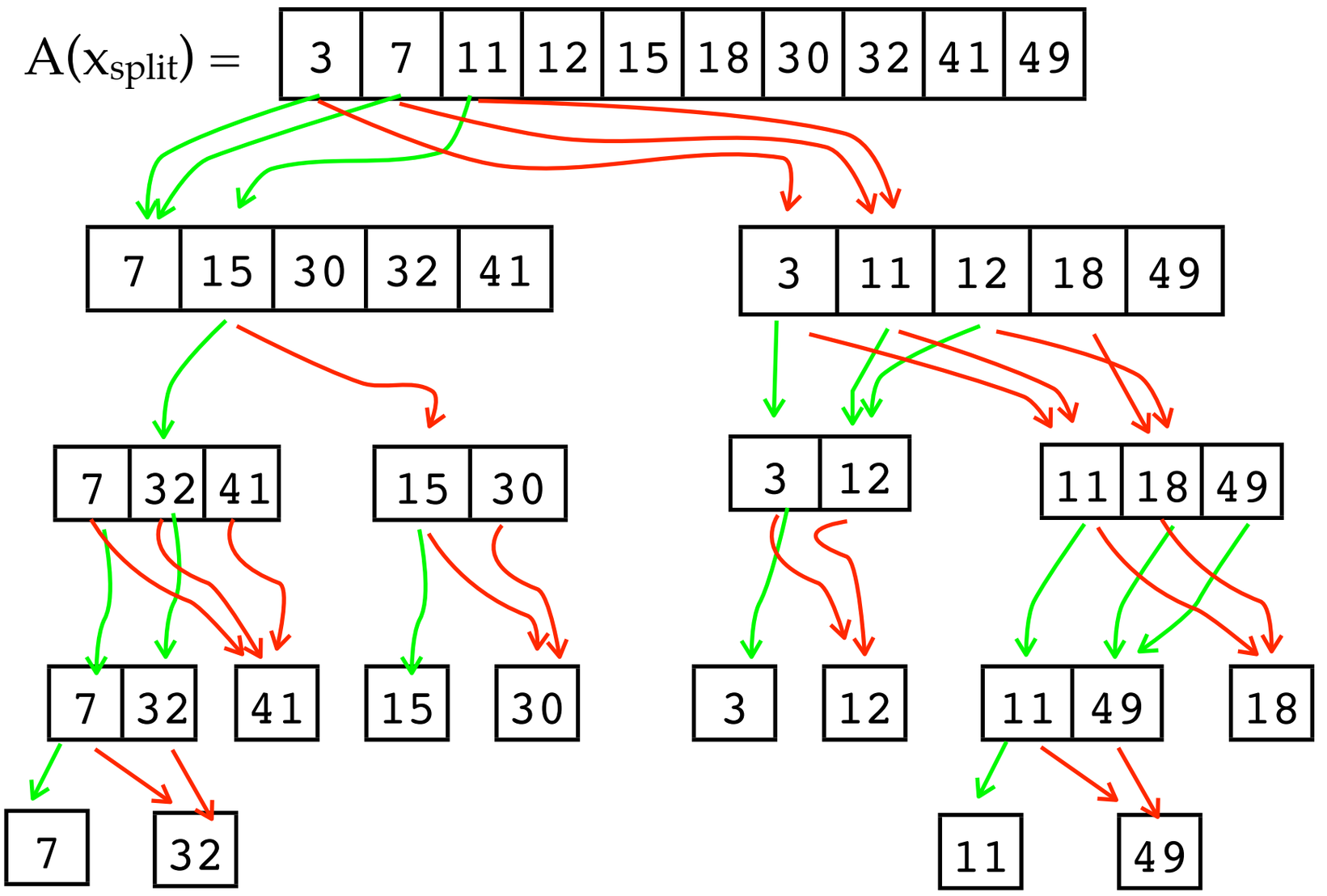
  - After Binary Search in A_1 , use pointer to find where to start in A_2

- Can do similar in Range Trees to eliminate an O(log n) factor (see next slides)

Fractional Cascading in Range Trees

Instead of an aux. tree, we store an array, sorted by Y-coord. At x_{split}, we do a binary search for y_1. As we continue to search for x_1 and x_2, we also use pointers to keep track of the result of a binary search for y_1 in each of the arrays along the path.





(Only subset of pointers are shown)

Fractional Cascading Search

- RangeQuery([x1,x2] by [y1,y2]):
  - Search for x_{split}
  - Use binary search to find the first point in $A(x_{split})$ that is larger that y_1.
  - Continue searching for x_1 and x_2, following the now diverged paths
  - Let u_1--u_2--u_3--u_k be the path to x_1. While following this path, use the "cascading" pointers to find the first point in each $A(u_i)$ that is larger than y_1. [similarly with the path v_1--v_2--v_m to x_2]
  - If a child of u_i or v_i is the root of a subtree to output, then use a cascading pointer to find the first point larger than y_1, output all points until you pass y_2.

Fractional Cascading: Runtime

- Instead of $O(\log n)$ binary searches, you perform just one

- Therefore, $O(\log^2 n)$ becomes $O(\log n)$

- 2d-rectangle range queries in $O(\log n + k)$ time

- In d dimensions: $O(\log^{d-1} n + k)$