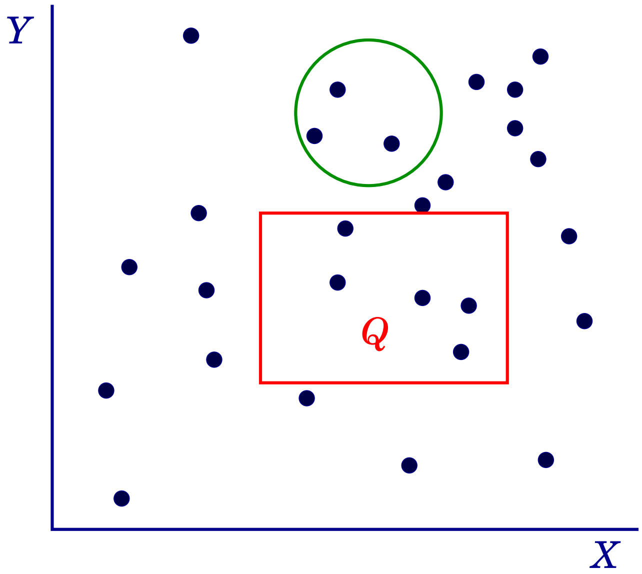
Range Searching

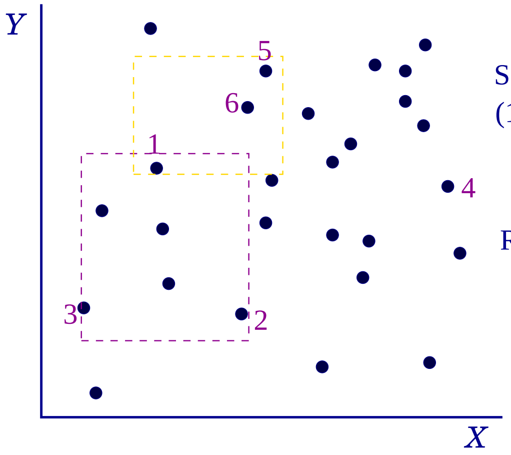
- Data structure for a set of objects (points, rectangles, polygons) for efficient range queries.



- Depends on type of objects and queries. Consider basic data structures with broad applicability.
- Time-Space tradeoff: the more we preprocess and store, the faster we can solve a query.
- Consider data structures with (nearly) linear space.

Orthogonal Range Searching

- **Fix a n-point set P. It has 2^n subsets. How many are possible answers to geometric range queries?**



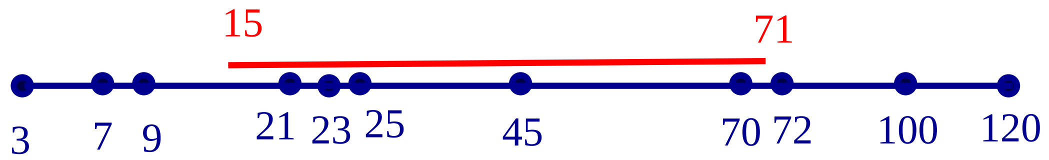
Some impossible rectangular ranges (1,2,3), (1,4), (2,5,6).

Range (1,5,6) is possible.

- **Efficiency comes from the fact that only a small fraction of subsets can be formed.**
- **Orthogonal range searching deals with point sets and axis-aligned rectangle queries.**
- **These generalize 1-dimensional sorting and searching, and the data structures are based on compositions of 1-dim structures.**

1-Dimensional Search

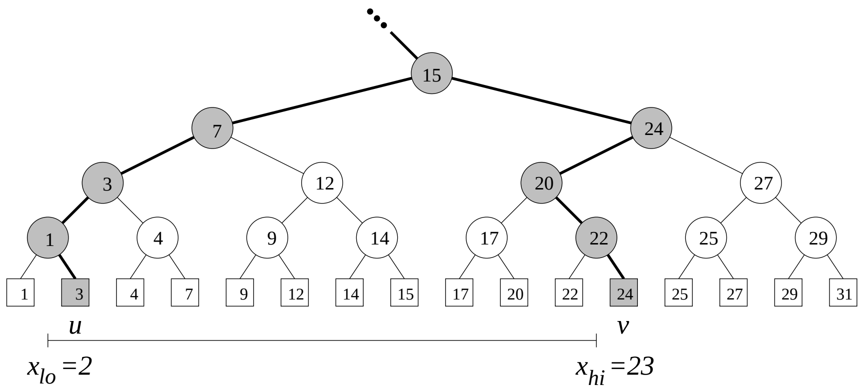
- **Points in 1D $P = \{p_1, p_2, \ldots, p_n\}$.**
- **Queries are intervals.**



- **If the range contains k points, we want to solve the problem in $O(\log n + k)$ time.**

- **Does hashing work? Why not?**
- **A sorted array achieves this bound. But it doesn't extend to higher dimensions.**
- **Instead, we use a balanced binary tree.**

Tree Search



- **Build a balanced binary tree on the sorted list of points (keys).**

- **Leaves correspond to points; internal nodes are branching nodes.**

- **Given an interval $[x_{lo}, x_{hi}]$, search down the tree for x_{lo} and x_{hi}.**

- **All leaves between the two form the answer.**

- **Tree searches takes $2 \log n$, and reporting the points in the answer set takes $O(k)$ time; assume leaves are linked together.**

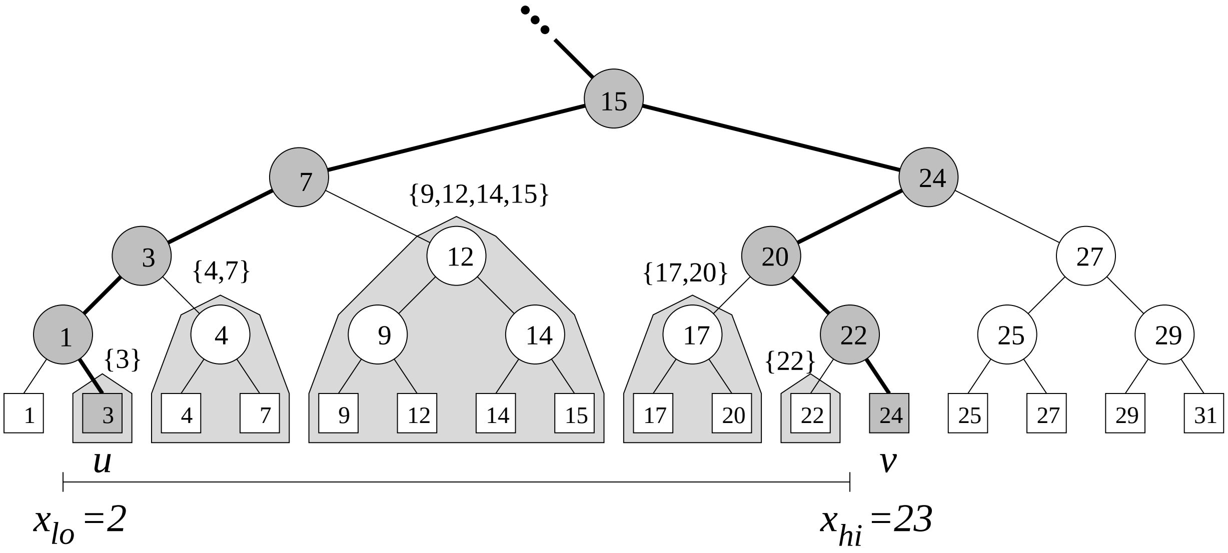
Canonical Subsets

- $S_1, S_2, \ldots, S_k$ **are canonical subsets,** $S_i \subseteq P$**, if the answer to any range query can be written as the disjoint union of some** S_i**'s.**

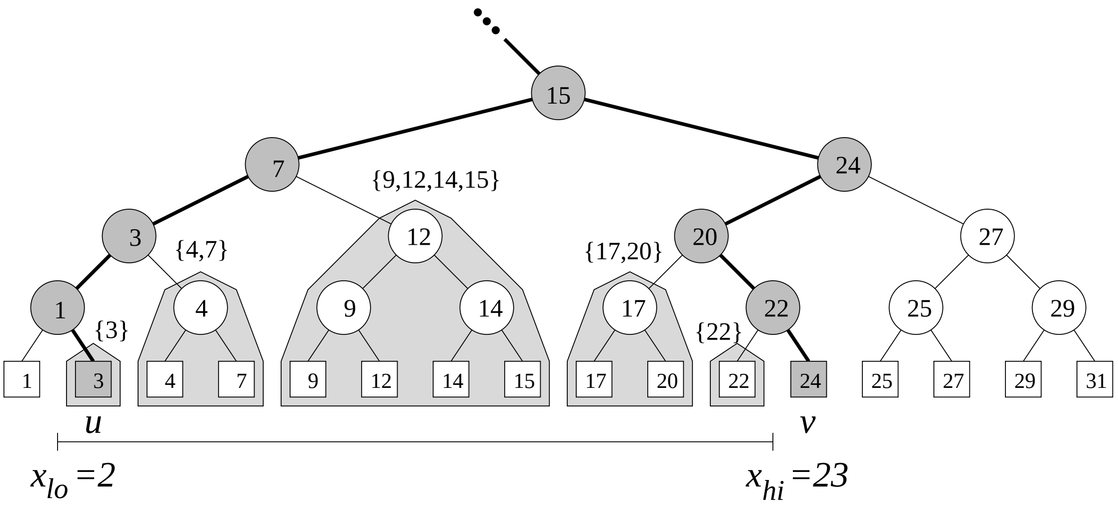
- **The canonical subsets may overlap.**

- **Key is to determine correct** S_i**'s, and given a query, efficiently determine the appropriate ones to use.**

- **In 1D, a canonical subset for each node of the tree:** S_v **is the set of points at the leaves of the subtree with root** v**.**



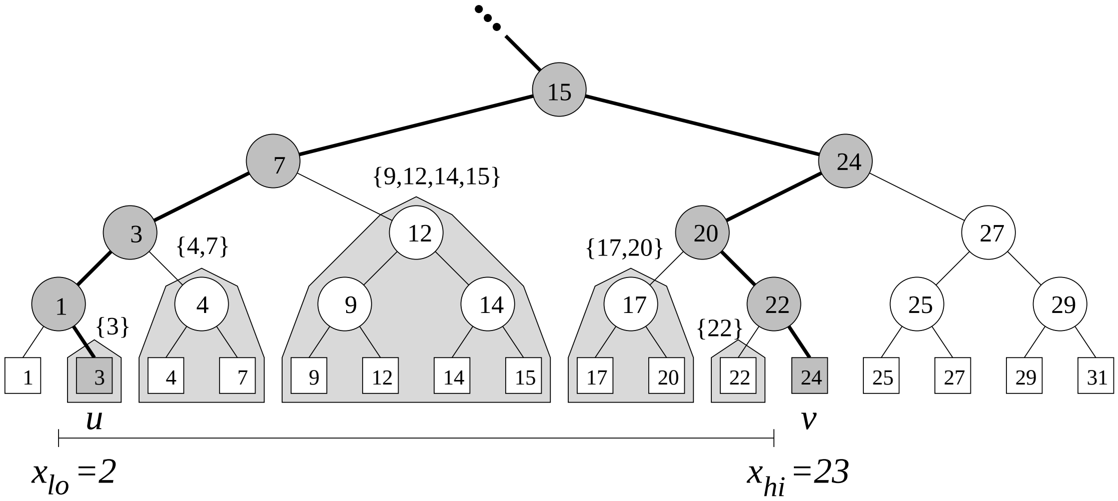
1D Range Query



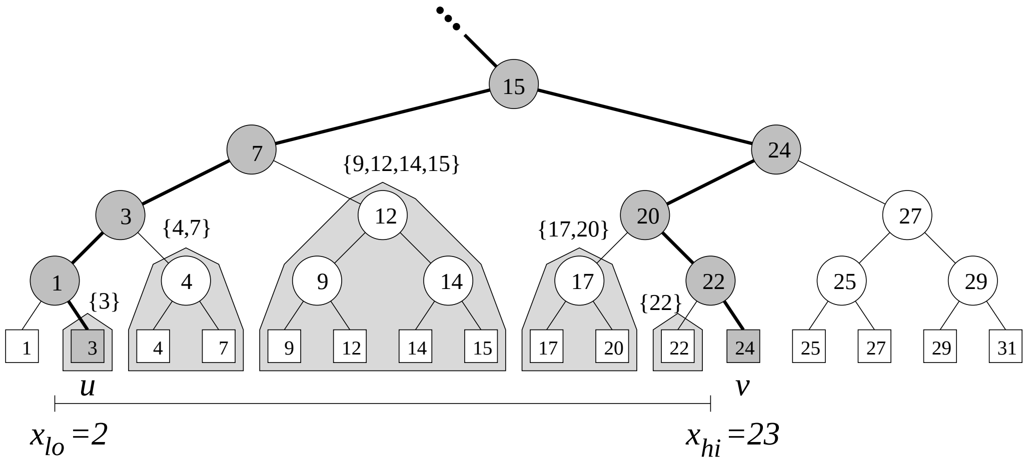
- **Given query $[x_{lo}, x_{hi}]$, search down the tree for leftmost leaf $u \geq x_{lo}$, and leftmost leaf $v \geq x_{hi}$.**
- **All leaves between u and v are in the range.**
- **If $u = x_{lo}$ or $v = x_{hi}$, include that leaf's canonical set (singleton) into the range.**
- **The remainder range determined by maximal subtree lying in the range $[u, v)$.**

Query Processing

- Let z be the last node common to search paths from root to u, v.
- Follow the left path from z to u. When path goes left, add the canonical subset of right child. (Nodes 7, 3, 1 in Fig.)
- Follow the right path from z to v. When path goes right, add the canonical subset of left child. (Nodes 20, 22 in Fig.)

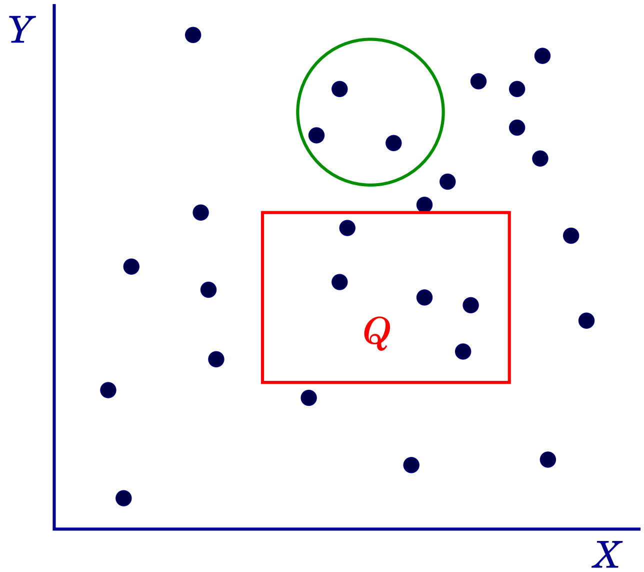


Analysis



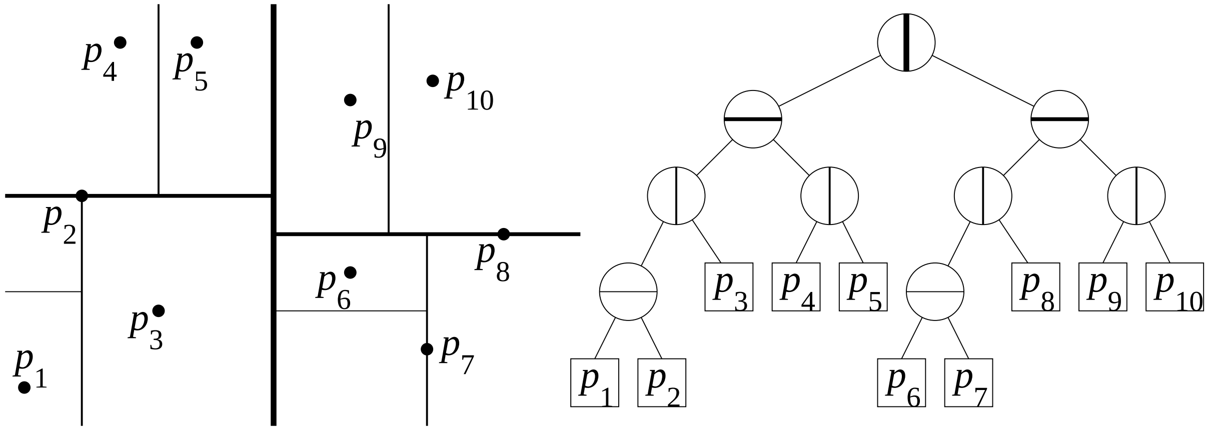
- **Since search paths have $O(\log n)$ nodes, there are $O(\log n)$ canonical subsets, which are found in $O(\log n)$ time.**
- **To list the sets, traverse those subtrees in linear time, for additional $O(k)$ time.**
- **If only count is needed, storing sizes of canonical sets at nodes suffices.**
- **Data structure uses $O(n)$ space, and answers range queries in $O(\log n)$ time.**

Multi-Dimensional Data



- **Range searching in higher dimensions?**
- ***kD*-trees [Jon Bentley 1975]. Stands for *k*-dimensional trees.**
- **Simple, general, and arbitrary dimensional. Asymptotic search complexity not very good.**
- **Extends 1D tree, but alternates using *x*-*y*-coordinates to split. In *k*-dimensions, cycle through the dimensions.**

kD-Trees

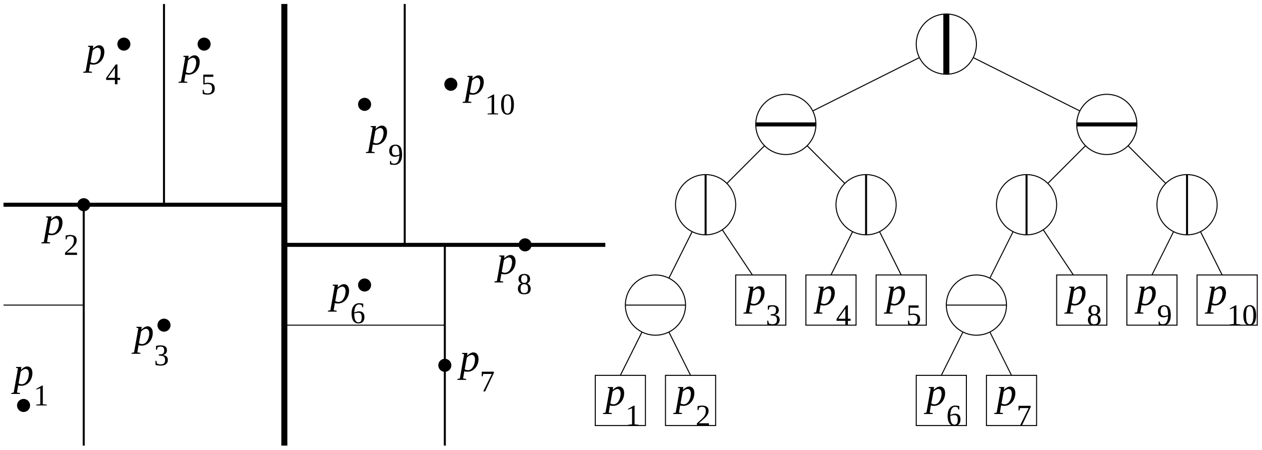


Subdivision

Tree structure

- **A binary tree. Each node has two values: split dimension, and split value.**
- **If split along x, at coordinate s, then left child has points with x-coordinate $\leq s$; right child has remaining points. Same for y.**
- **When $O(1)$ points remain, put them in a leaf node.**
- **Data points at leaves only; internal nodes for branching and splitting.**

Splitting



Subdivision

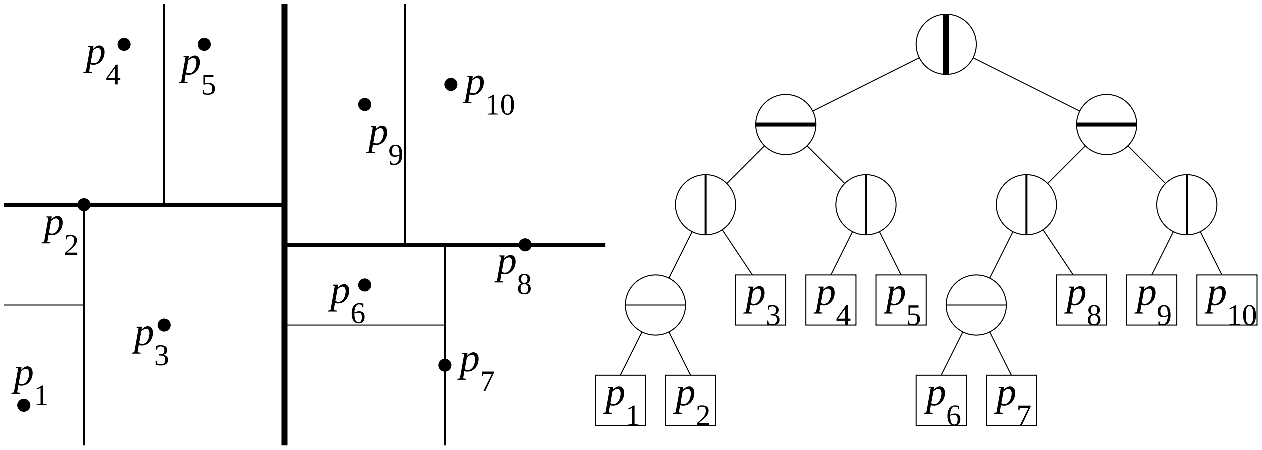
Tree structure

- To get balanced trees, use the median coordinate for splitting—median itself can be put in either half.

- With median splitting, the height of the tree guaranteed to be $O(\log n)$.

- Either cycle through the splitting dimensions, or make data-dependent choices. E.g. select dimension with max spread.

Space Partitioning View

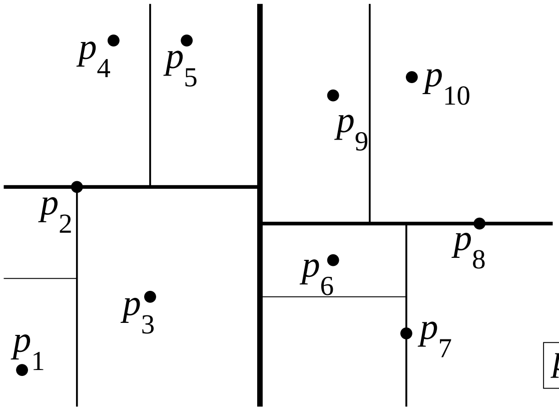


Subdivision

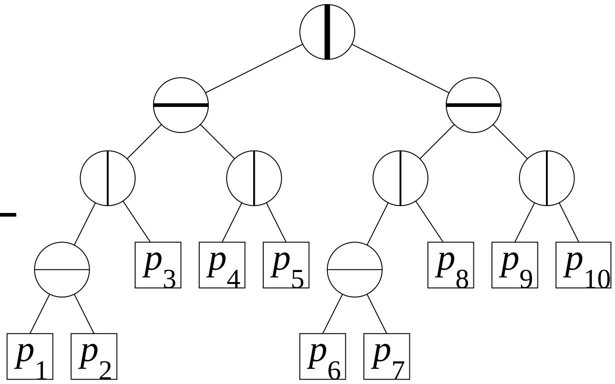
Tree structure

- kD-tree induces a space subdivision—each node introduces a x- or y-aligned cut.
- Points lying on two sides of the cut are passed to two children nodes.
- The subdivision consists of rectangular regions, called cells (possibly unbounded).
- Root corresponds to entire space; each child inherits one of the halfspaces, so on.
- Leaves correspond to the terminal cells.
- Special case of a general partition BSP.

Construction



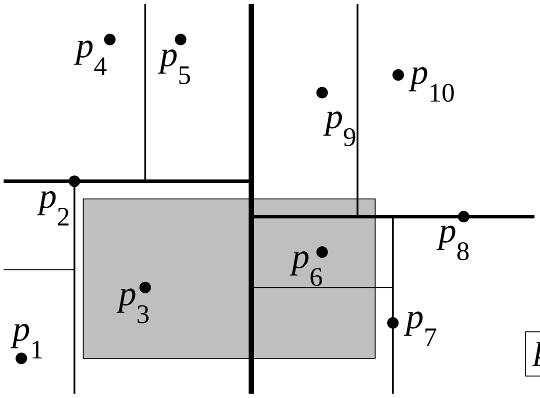
Subdivision



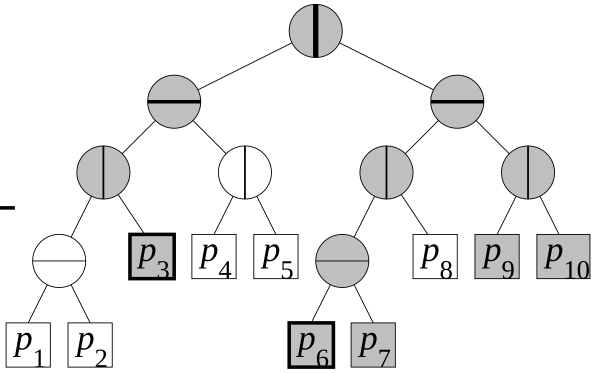
Tree structure

- **Can be built in $O(n \log n)$ time recursively.**
- **Presort points by x and y-coordinates, and cross-link these two sorted lists.**
- **Find the x-median, say, by scanning the x list. Split the list into two. Use the cross-links to split the y-list in $O(n)$ time.**
- **Now two subproblems, each of size $n/2$, and with their own sorted lists. Recurse.**
- **Recurrence $T(n) = 2T(n/2) + n$, which solves to $T(n) = O(n \log n)$.**

Searching kD-Trees



The range



Nodes visited in search

- **Suppose query rectangle is R. Start at root node.**

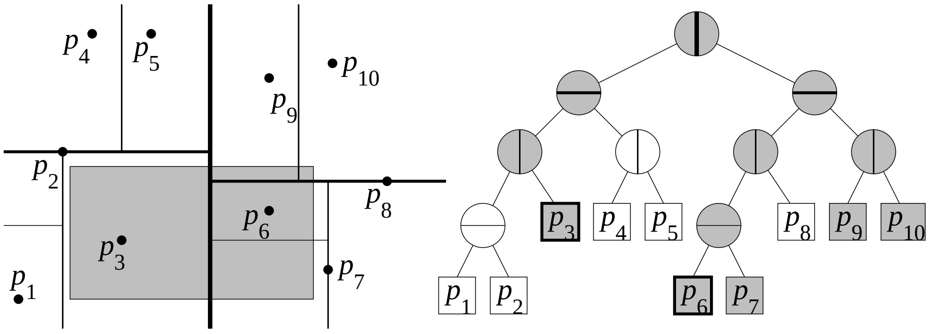
- **Suppose current splitting line is vertical (analogous for horizontal). Let v, w be left and right children nodes.**

- **If v a leaf, report $cell(v) \cap R$;**
  if $cell(v) \subseteq R$, report all points of cell(v);
  if $cell(v) \cap R = \emptyset$, skip;
  otherwise, search subtree of v recursively.

- **Do the same for w.**

- **Procedure obviously correct. What is the time complexity?**

Search Complexity

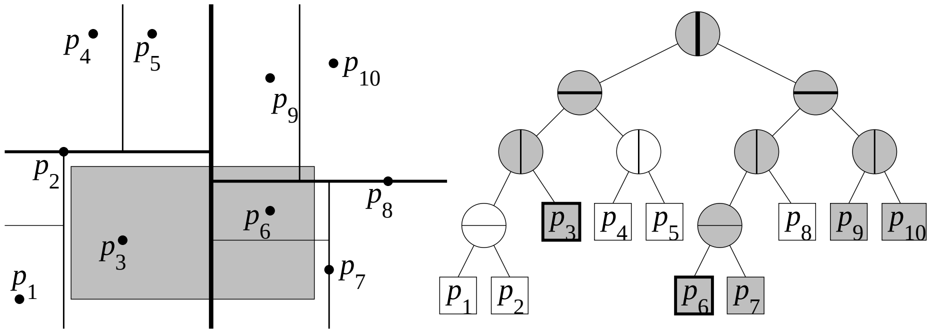


The range

Nodes visited in search

- **When** $cell(v) \subseteq R$**, complexity is linear in output size.**
- **It suffices to bound the number of nodes** v **visited for which the boundaries of** $cell(v)$ **and** R **intersect.**
- **If** $cell(v)$ **outside** R**, we don't search it; if** $cell(v)$ **inside** R**, we enumerate all points in region of** v**; a recursive call is made only if** $cell(v)$ **partially overlaps** R**; the** kD**-tree height is** $O(\log n)$**.**
- **Let** ℓ **be the line defining one side of** R**.**
- **We prove a bound on the number of cells that intersect** ℓ**; this is more than what is needed; multiply by 4 for total bound.**

Search Complexity



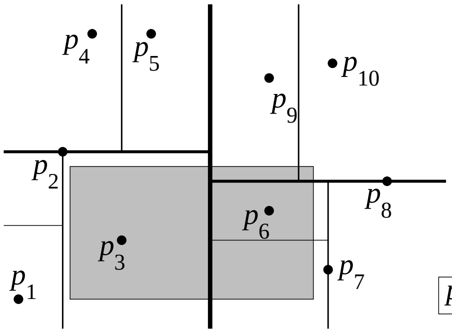
The range

Nodes visited in search

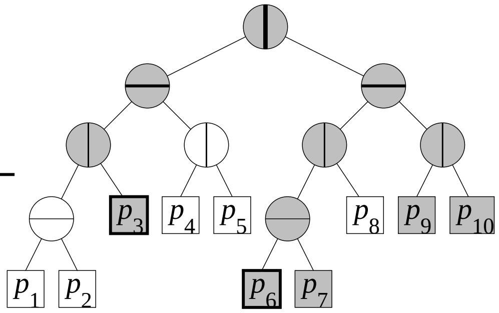
- How many cells can a line intersect?
- Since splitting dimensions alternate, the key idea is to consider two levels of the tree at a time.
- Suppose the first cut is vertical, and second horizontal. We have 4 cells, each with $n/4$ points.
- A line intersects exactly two cells; the others cells will be either outside or entirely inside R.
- The recurrence is

$$Q(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2Q(n/4) + 2 & \text{otherwise.} \end{cases}$$

Search Complexity



The range



Nodes visited in search

- **The recurrence $Q(n) = 2Q(n/4) + 2$ solves to**
$$Q(n) = O(\sqrt{n})$$
- **kD-Tree is an $O(n)$ space data structure that solves 2D range query in worst-case time $O(\sqrt{n} + m)$, where m is the output size.**

d-Dim Search Complexity

- What's the complexity in higher dimensions?
- Try 3D, and then generalize.
- The recurrence is

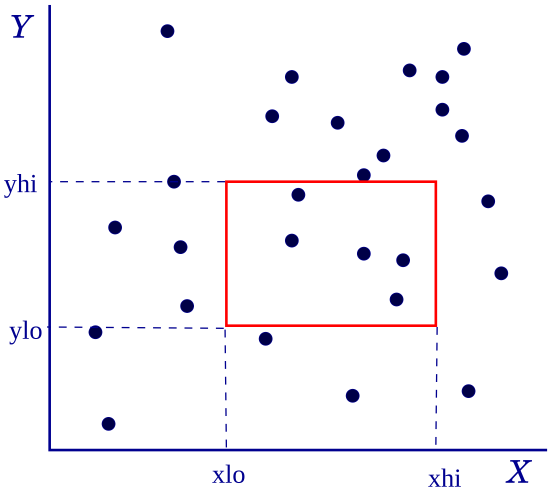
$$Q(n) = 2^{d-1} Q(n/2^d) + 1$$

- It solves to

$$Q(n) = O(n^{1-1/d})$$

- kD-Tree is an $O(dn)$ space data structure that solves d-dim range query in worst-case time $O(n^{1-1/d} + m)$, where m is the output size.

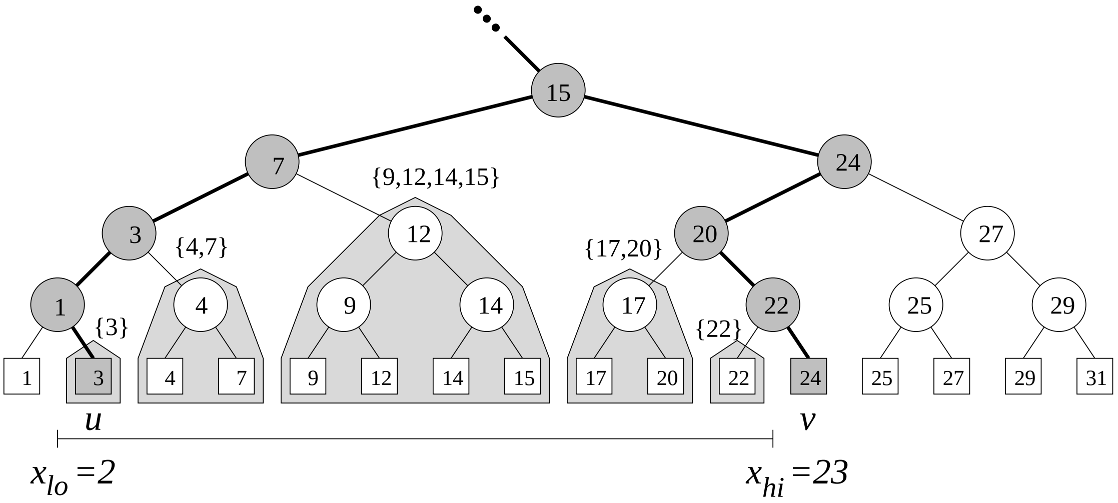
Orthogonal Range Trees



- **Generalize 1D search trees to dimension d.**
- **Each search recursively decomposes into multiple lower dimensional searches.**
- **Search complexity is $O((\log n)^d + k)$, where k is the answer size.**
- **Space & time complexity $O(n(\log n)^{d-1})$.**
- **Fractional cascading eliminates one $\log n$ factor from search time.**
- **We focus on $2D$, but ideas readily extend.**

2D Range Trees

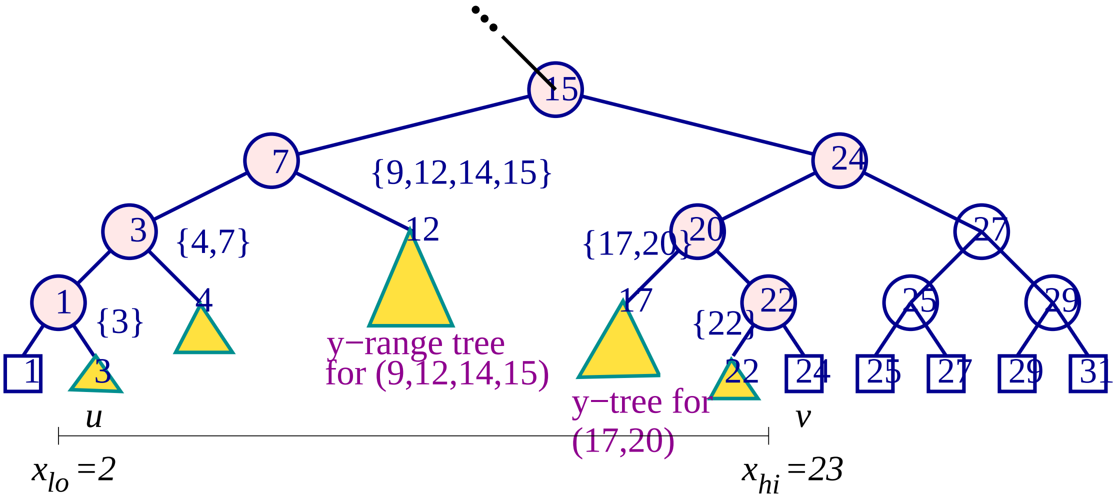
- Suppose $P = \{p_1, p_2, \ldots, p_n\}$ set of points in the plane.
- The generic query is $R = [x_{lo}, x_{hi}] \times [y_{lo}, y_{hi}]$.
- We first ignore the y-coordinates, and build a 1D x-range tree on P.



- The set of points that fall in $[x_{lo}, x_{hi}]$ belong to $O(\log n)$ canonical sets.
- This is a superset of the final answer. It can be significantly bigger than $|R \cap P|$, so we can't afford to look at each point in these canonical sets.

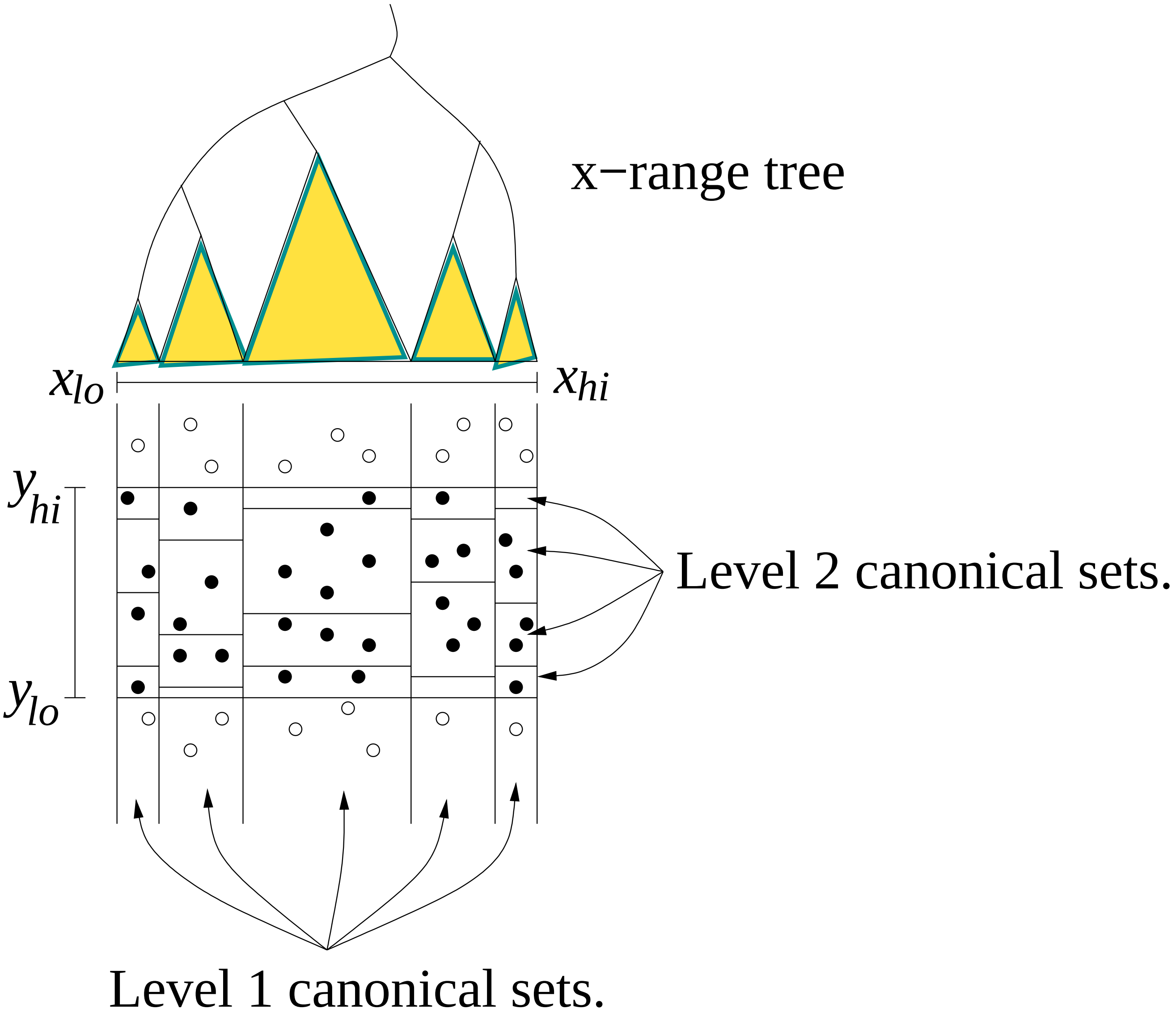
Level 2 Trees

- Key idea is to collect points of each canonical set, and build a y-range tree on them.
- E.g., the canonical set $\{9, 12, 14, 15\}$ is organized into a 1D range tree using those points' y-coordinates.



- We search each of the $O(\log n)$ canonical sets that include points for x-range $[x_{lo}, x_{hi}]$ using their y-range trees for range $[y_{lo}, y_{hi}]$.
- The y-range searches list out the points in $R \cap P$. (No duplicates.)

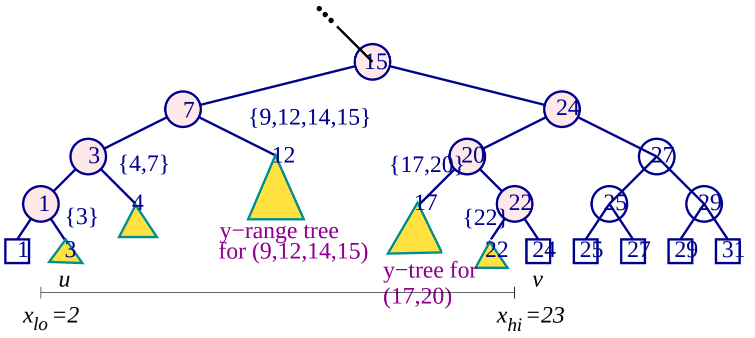
Canonical Sets



Analysis

- **Time complexity for 2D is $O((\log n)^2)$.**

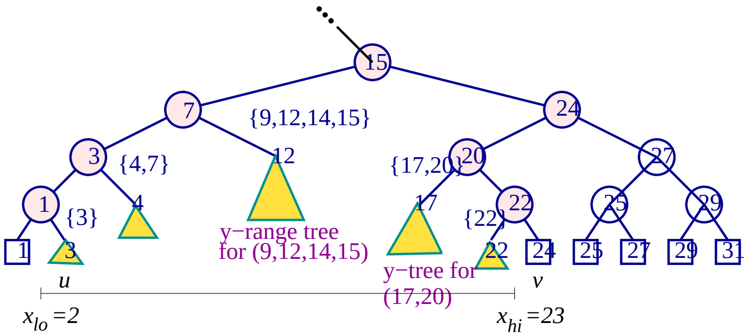
1. **$O(\log n)$ canonical sets for x-range.**
2. **Each set's y-range query takes $O(\log n)$ time.**



- **Space complexity is $O(n \log n)$.**

1. **What is the total size of all canonical sets in x-tree?**
2. **Number of nodes $\equiv$ number of leaves.**
3. **One set of size n. Two of size $n/2$, etc.**
4. **Total is $O(n \log n)$.**
5. **Each canonical set of size m requires $O(m)$ space for the y-range tree.**
6. **So, overall space is $O(n \log n)$.**

Construction



- **The x-tree can be built in $O(n \log n)$ time.**
- **Naively, since total size of all y-trees is $O(n \log n)$, it will take $O(n(\log n)^2)$ time to build them.**
- **By building them bottom-up, we can avoid sorting cost at each node.**
- **Once y-trees for the children nodes are built, we can merge their y-lists to get the parent's y-list in linear time.**
- **The cost of building the 1D range tree is linear after sorting.**
- **Thus, total time is linear in $O(n \log n)$, the total sizes of all y-tree.s**

d-Dim Range Trees

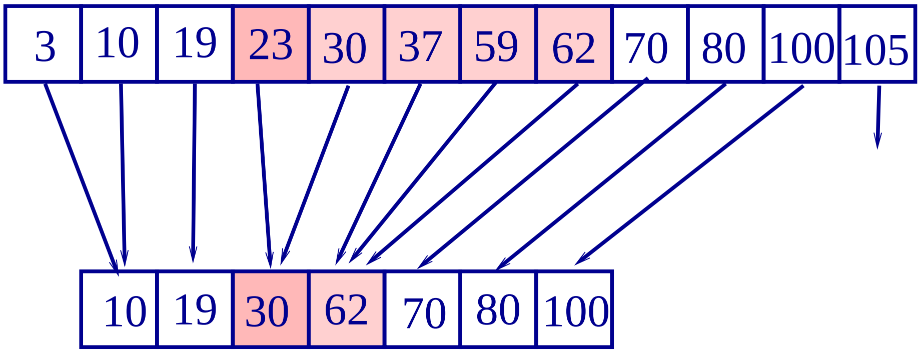
- **The multi-level range tree idea extends naturally to any dimension d.**
- **Build the x-tree on first coordinate.**
- **At each node v of this tree, build the $(d-1)$-dimensional range tree for canonical set of v on the remaining $d-1$ dimensions.**
- **Search complexity grows by one $\log n$ factor for each dimension—each dimensional increases the number of canonical sets by $\log n$ factor.**
- **So, search cost is $O((\log n)^d)$.**
- **Space and time complexity is $O(n(\log n)^{d-1})$.**

Fractional Cascading

- **A technique that improves the range tree search time by log factor. 2D search can be done in $O(\log n)$ time.**
- **Basic idea: Range tree first finds the set of points lying in $[x_{lo}, x_{hi}]$ as union of $O(\log n)$ canonical sets.**
- **Next, each canonical set is searched using the y-tree for range $[y_{lo}, y_{hi}]$. We locate y_{lo}; then read off points until y_{hi} reached.**
- **Since each set is searched for the same key, y_{lo}, we can improve the search to $O(1)$ per set.**
- **In effect, we do the first search in $O(\log n)$ time, but then use that information to search other structures more efficiently.**
- **The key is to place smart hooks linking the search structures for the canonical sets.**

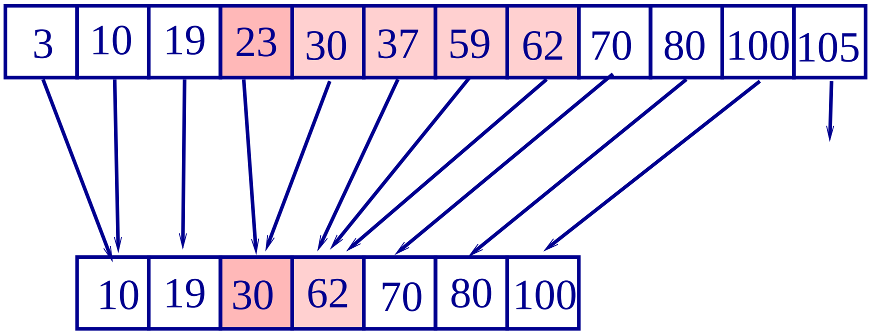
Basic Idea

- To understand the basic idea, consider a simple example.
- We have two sets of numbers, A_1, A_2, both sorted.
- Given a range $[x, x']$, want to report all keys in A_1, A_2 that lie in the range.
- Straightforward method takes $2 \log n + k$, if k is the answer size; separate binary searches in A_1, A_2 to locate x.
- For example, range $[20, 65]$.



Fractional Cascading Idea

- **Suppose $A_2 \subset A_1$. Add pointers from A_1 to A_2.**
- **If $A_1[i] = y_i$, store ptr to entry in A_2 with smallest key $\geq y_i$. (Nil if undefined.)**



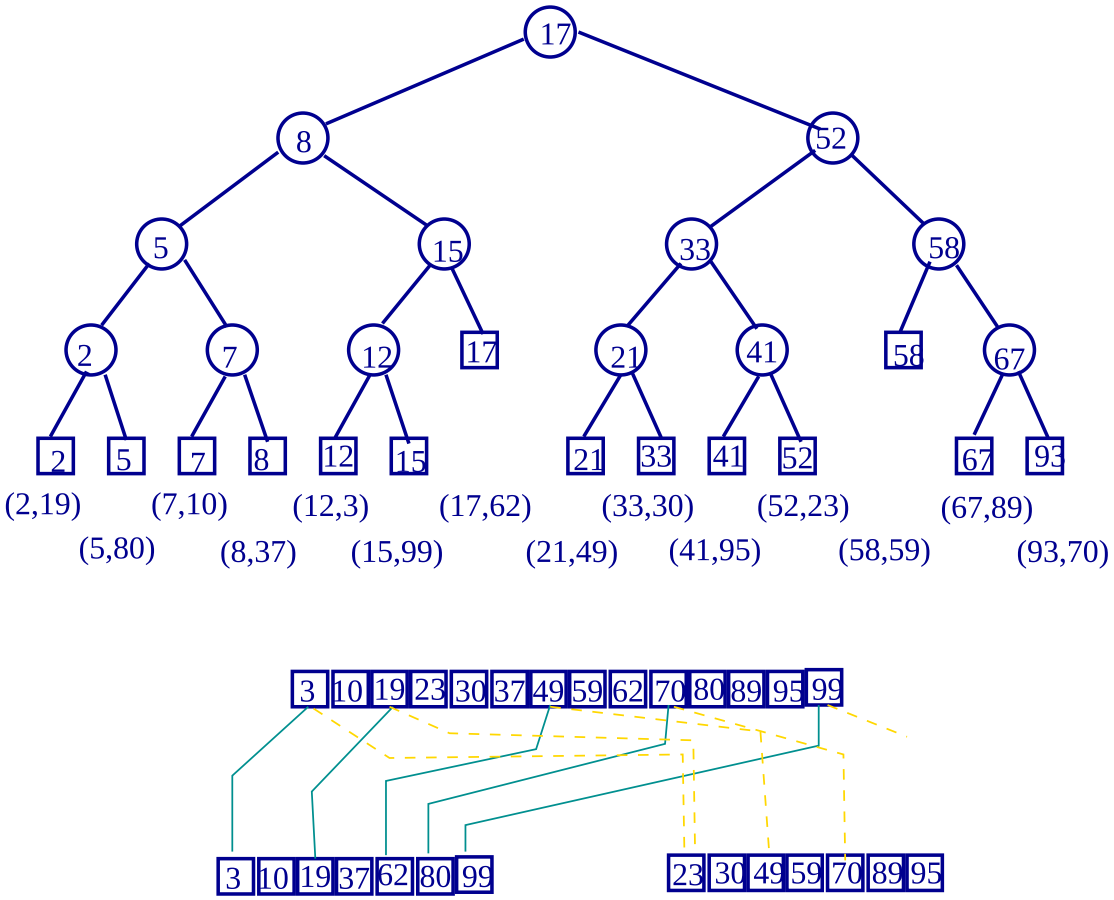
- **Suppose we want keys in range $[y, y']$.**
- **Search A_1 for y, and walk until past y'. Time $O(\log n + k_1)$.**
- **If A_1 search for y ended at $A_1[i]$, use its pointer to start search in A_2. This takes $O(1 + k_2)$ time.**
- **Example $[20, 65]$.**

FC in Range Trees

- **Key observation: canonical subsets $S(\ell(v))$ and $S(r(v))$ are subsets of $S(v)$.**
- **The x-tree is same as before. But instead of building y-trees for canonical subsets, we store them as sorted arrays, by y-coordinate.**
- **Each entry in $A(v)$ stores two pointers, into arrays $A(\ell(v))$ and $A(r(v))$.**
- **If $A(v)[i]$ stores point p, then ptr into $A(\ell(v))$ is to entry with smallest y-coordinate $\geq y(p)$. Same for $(r(v))$.**

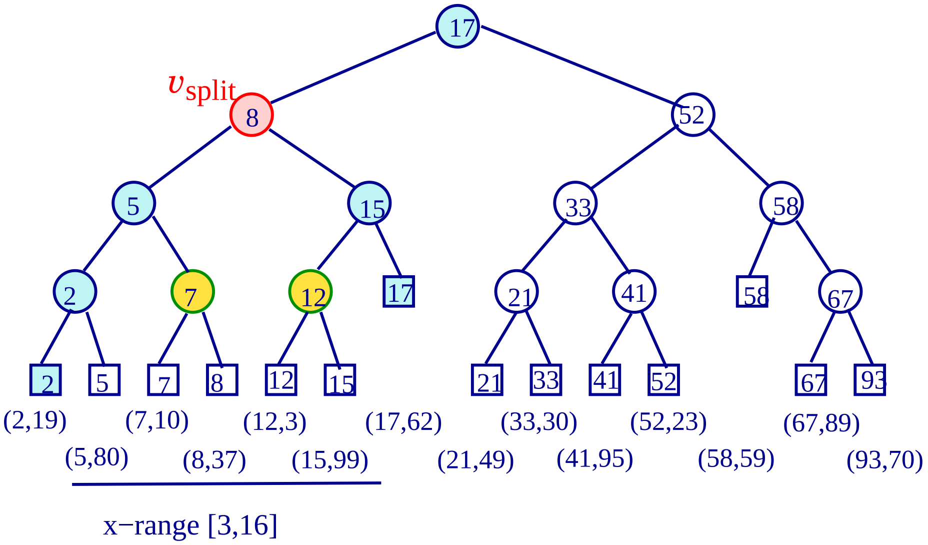
Range Tree FC

- **Only some pointers shown to avoid clutter.**



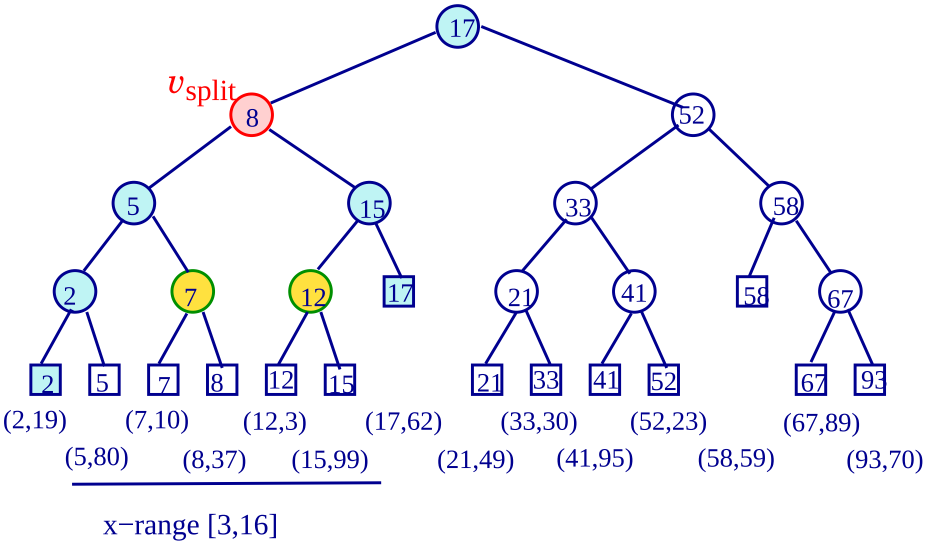
FC Search

- **Consider range $R = [x, x'] \times [y, y']$.**
- **Search for x, x' in the main x-tree.**
- **Let v_{split} be the node where the two search paths diverge.**
- **The $O(\log n)$ canonical subsets correspond to nodes that lie below v_{split}, and are the right (left) child of a node on search path to x (resp. x') where the path goes left (resp. right).**



FC Search

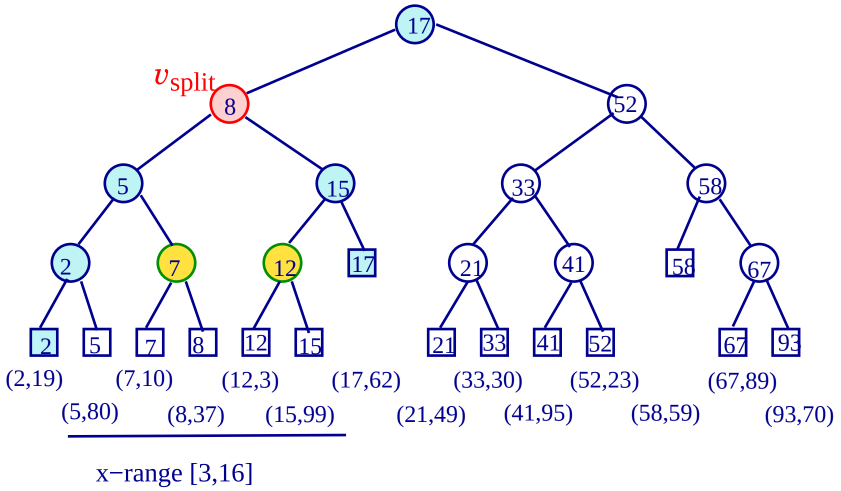
- At v_{split}, do binary search to locate y in $A(v_{split})$. $O(\log n)$ time.
- As we search down the x-tree for x, x', keep track of the entries in the associated arrays for smallest keys $\geq y$, at $O(1)$ cost per node.



- Let $A(v)$ be one of the $O(\log n)$ canonical nodes that is to be searched for $[y, y']$ range.
- We just need to find the smallest entry in $A(v) \geq y$.

FC Search

- **We can find this in $O(1)$ time because $parent(v)$ is on the search path, and we know smallest entry $\geq y$ in $A(parent(v))$, and have a pointer from that to v's array.**



- **So we can output all points in $A(v)$ that lie in range $[y, y']$ in time $O(1 + k_v)$, where k_v is the answer size.**
- **For 2D range search, the final time complexity is $O(\log n + k)$, and space $O(n \log n)$.**
- **d-dim range search takes $O((\log n)^{d-1} + k)$ time with fractional cascading.**