Data Structures & Algorithms for Problem Solving (CS1.304)

Searching in Higher Dimensions

Avinash Sharma
Center for Visual Information Technology (CVIT),
IIIT Hyderabad

Motivation

- Consider a dataset of people storing their age and annual income.
- A subset of census data.
- Of this data, suppose we want the number of people with age between 30 to 40, and income more than 10 L INR.
- Similar queries
- Find the person(s) such that there are no persons with both a bigger age and bigger income.

S.No	Age	Income (in Lakhs INR)
1	21	2.3
2	25	4.5
3	56	5.7
.	.	
.	.	
.	.	
n	32	4.2

Motivation

- Two kinds of problems:
- Ask a one-time question on the entire data set or a subset of the dataset.
- Or, ask multiple questions on different subsets of the dataset.
  - Example: Find the person with the largest income in the age group of 25-35.
  - Find the person with the largest income in the age group of 55-65.

S.No	Age	Income (in Lakhs INR)
1	21	2.3
2	25	4.5
3	56	5.7
.	.	
.	.	
.	.	
n	32	4.2

Motivation

- Even for one-time questions, some problems require more sophisticated data structures.
- Examples:
  - Find a pair of people such that they have the closest difference between their age and their income.
- Often called as the *closest pair of points* queries.
- Another example:
  - Find all people who are at least 25 years and earn more than 15 L/yr.

Multi-Dimensional Data Sets

- Current data structures such as search trees can work only with 1-dimensional data.
- A big inherent problem with multi-dimensional data is that they are not comparable.
- In a 2-d setting, which is bigger? (10, 15) or (22, 8)?

- So need new data structures that can impose an order on multi-dimensional data.

Multi-Dimensional Data Sets

- All the previous problems on multi-dimensional data sets apply to the multiple query setting also.
- There are additional challenges.

- Typically, each query is on a subset of the points.
- How to identify these points quickly?
- Should not consider all points.

The Case of 1-Dimension

- Identifying the relevant points in a 1-dimensional setting is possible.
- Think of a search tree T, that can locate all the values between x and y (both inclusive).
- Can be done in time O(k+log n) where k is the number of such elements.

- How?

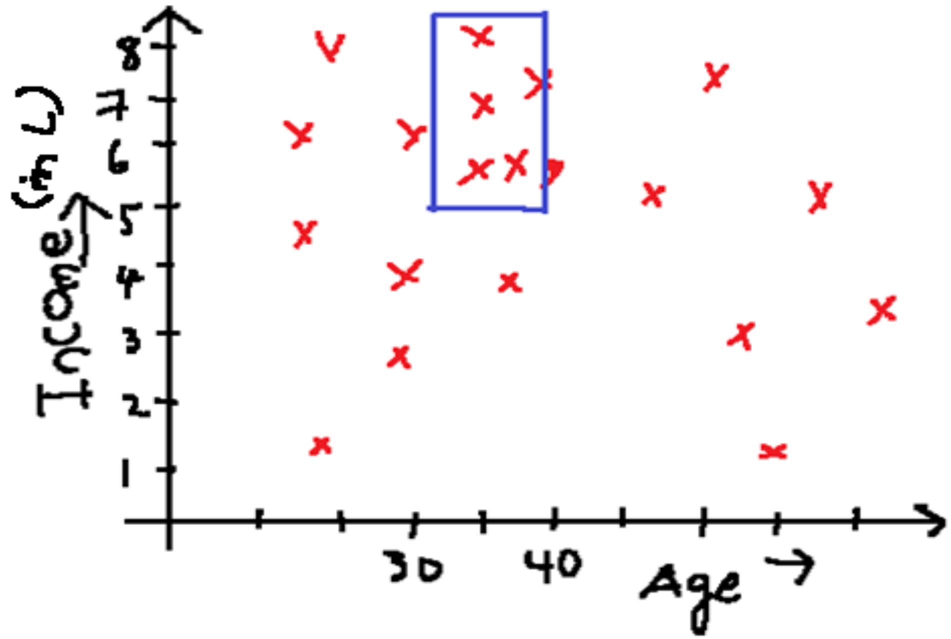
The Case of 1-Dimension

- Identifying the relevant points in a 1-dimensional setting is possible.
- Think of a search tree T, that can locate all the values between x and y (both inclusive).
- Can be done in time O(k+log n) where k is the number of such elements.
  - Need not filter the n points which takes O(n) time.
- How?

- Need a similar technique and a data structure for higher dimensional data sets.
- A solution that is faster than O(n).

Three Solutions

- We will study **three** different data structures in this context.
- We will also seek to solve a standard query:
- Given a rectangular region q = [a,b]x[c,d], find all the points of P that are inside q.
- This is called as a range query.



Some Solution Ideas

- In each case, suppose there are n points in a d-dimensional space.
- Can consider all these points and arrive at the result.
- Typically, however, the region of interest, or the query region, has far fewer points. Can find the result on these subset of points.
- But, identifying these points itself may take time.

A New Way for Data Structures

- Preprocess the points and create a suitable data structure.
- This data structure can then be used repeatedly for answering any query.
- Some parameters of efficiency:
  - Space used by the data structure
  - Time taken to build the data structure
  - Time to answer a query
- Query time is lower bounded by the size of the output. This is denoted k.
- So, we seek query time of O(k+polylog(n)) so that the result is practically fast.

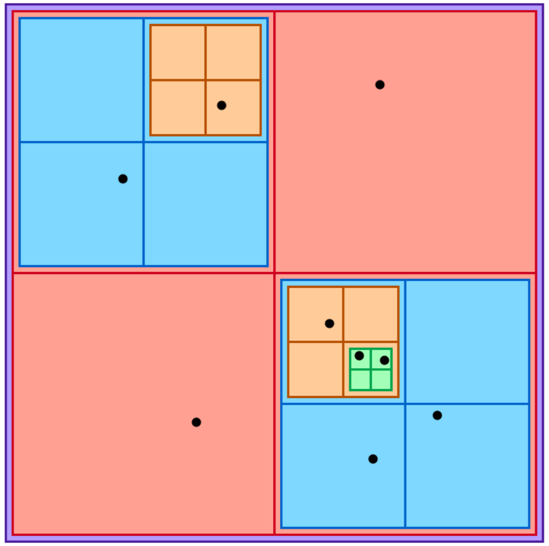
A New Way for Data Structures

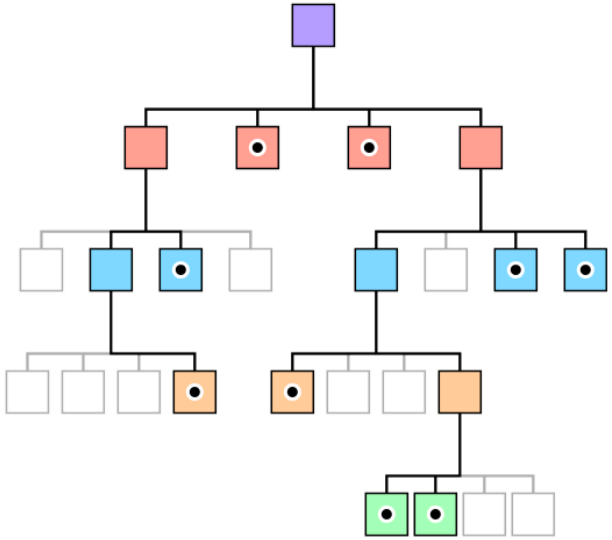
- Seemingly easy with one dimensional data.
- Build a balanced binary search tree of the n points.
- Exercise: Given two values x and y with $x < y$, can find the values in the input set that are between x and y.
- Can do this in time $O(k + \log n)$.
- Is it possible to do similar things in two and further dimensions?

First Solution – A Quad Tree

- We will first generalize a binary search tree to be useful for 2-d points.
- A node now has four children, labeled NE, NW, SE, and SW.
- At a node u, the points in the subtree rooted at the NE child have x- and y- coordinates larger than that of at u.

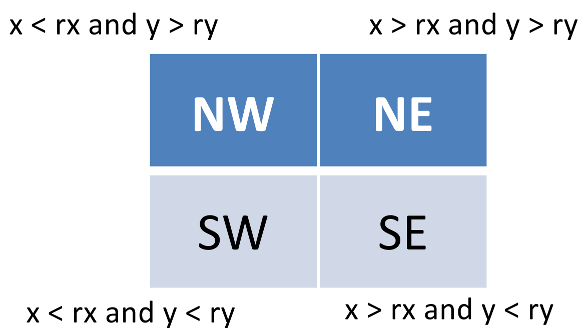
NW	**NE**
SW	SE



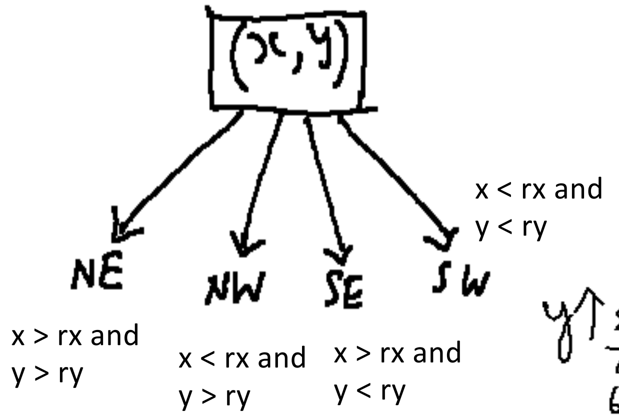


Quad Tree

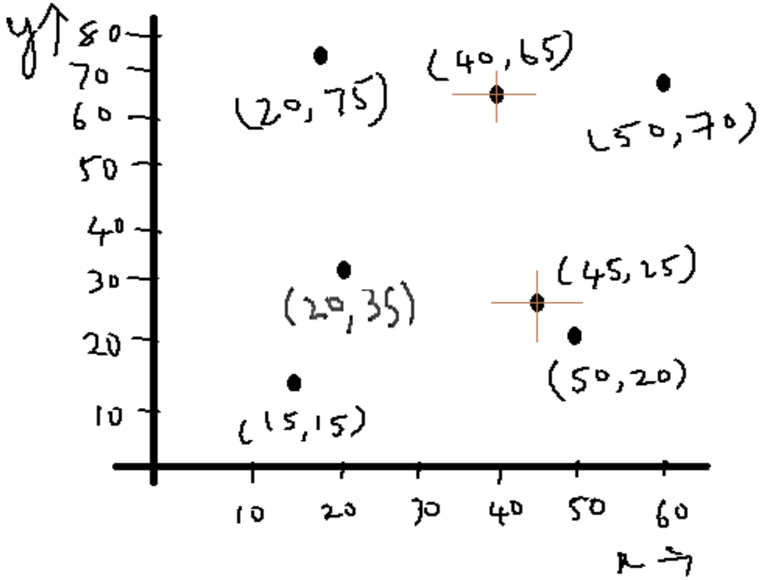
- Similar routines to that of a binary search tree.
- To Insert(p = (x,y)) into a quad tree Q, do the following:
  - Start from the root of the tree.
  - Let the point at the root be r = (rx, ry).
  - Four cases:
    - If x > rx and y > ry – Insert p in the NE child.
    - If x > rx and y < ry – Insert p in the SE child
    - If x < rx and y > ry – Insert p in the NW child.
    - If x < rx and y < ry – Insert p in the SW child.



Quad Tree



(x, y)
NE
NW
SE
SW
x > rx and
y > ry
x < rx and
y > ry
x > rx and
y < ry
x < rx and
y < ry



(20, 75)
(40, 65)
(50, 70)
(45, 25)
(20, 35)
(50, 20)
(15, 15)

Quad Tree

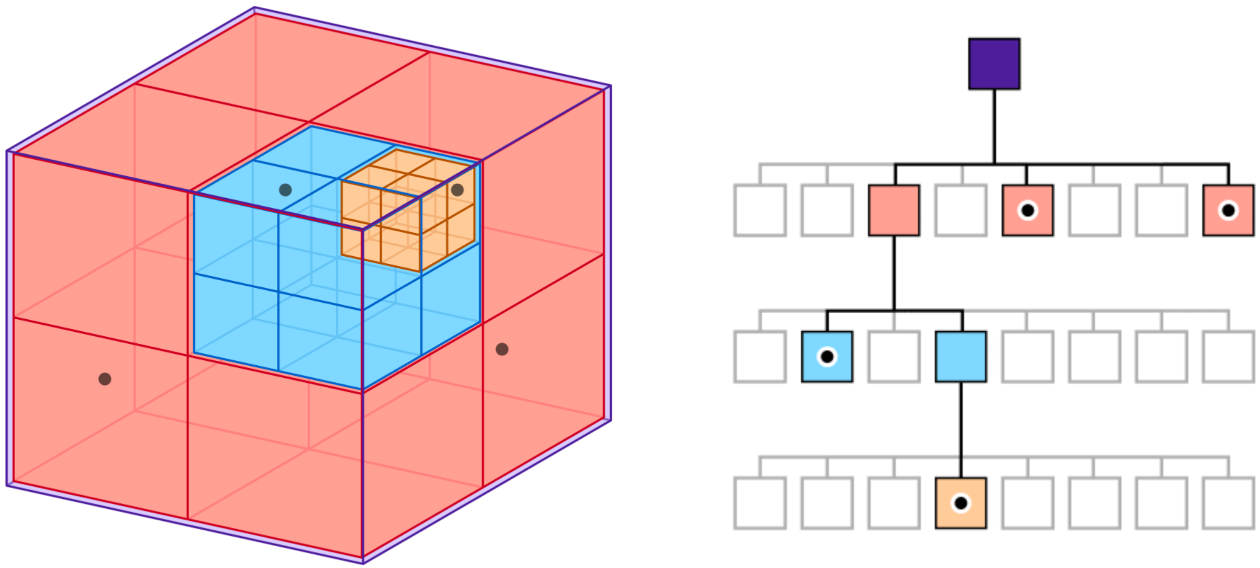
- Similar routines to that of a binary search tree.
- A Delete(p) routine can also be designed akin to the Delete routine of a binary search tree.

- A Find(p) routine is also similar. Start from the root, and depending on the x and y coordinates, search one of the four subtrees.

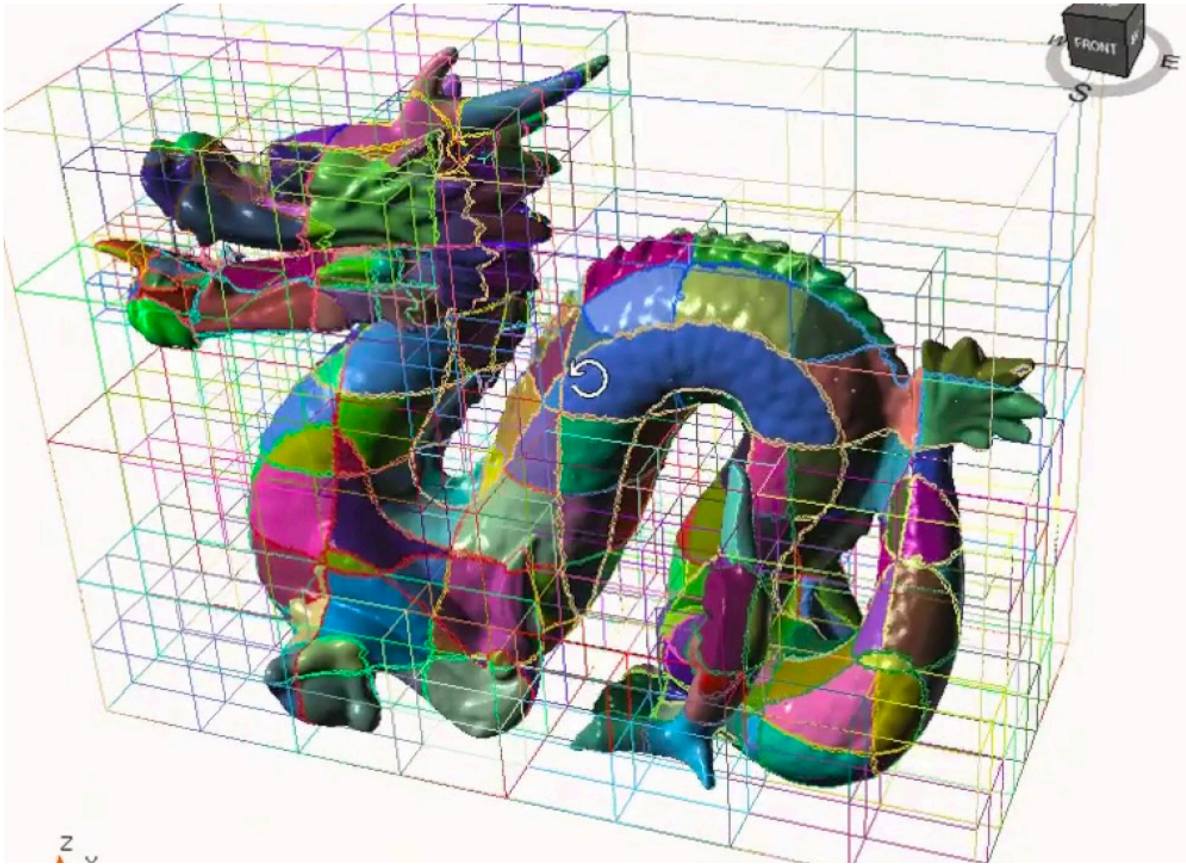
Quad Tree

- But there are several disadvantages:
- Height balance is difficult to achieve on insertions and deletions.
- Number of children grows exponentially with the number of dimensions.

Octree
for 3 dimensions



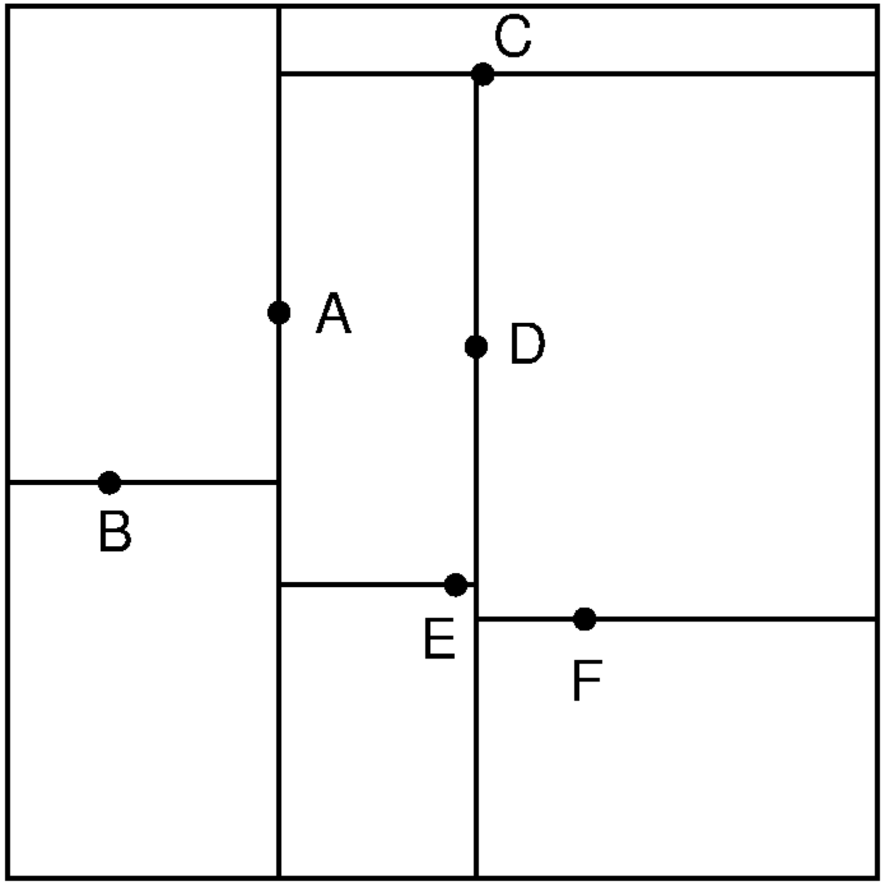
Octree in Computer Graphics

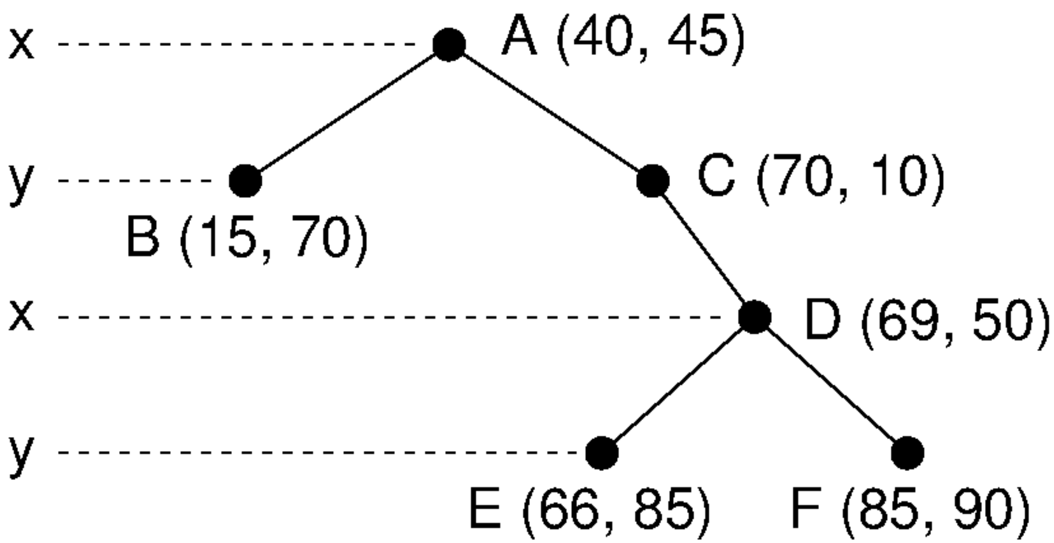


A Better Solution – A 2-d Tree

- Retain the flavour of a binary search tree.
  - Each node has at most 2 children.
- Recall that in a BST, values to the left (right) are smaller (larger) than the value at the root.
- In our 2-d tree, we position values so that each level alternates between x- and y-axes values.
- Points to the left (right) of the root node have their x-coordinate smaller (larger) than the x-coordinate of the root node.
- Points to the left (right) of the child of the root node have their y-coordinate smaller (larger) than that of the y-coordinate of the child node.

A Better Solution – A 2-d Tree





A Better Solution – A 2-d Tree

- Build Kd Tree for input : (30,40), (82,45), (8,60), (20,90), (10,10), (60,20), (62,70), (85,15)

A Better Solution – A 2-d Tree

- Inserting into the 2-d tree is straight forward.
- Proceed from the root of the tree, at each level comparing the appropriate coordinate value.
- For higher dimensions, cycle over all the d dimensions over d levels.

A Better Solution – A 2-d Tree

- If all the points are known a-priori, then can also build the tree as follows.
- Sort the points on their x-coordinates, pick the median, and make it the root.
- In each subtree, sort along the y-coordinates, and make the median as the root.

- Gets height balance.

A Better Solution – A 2-d Tree

- Some unexpected changes to other routines. Consider FindMin.
- Given a node u in a 2d-tree, and given a coordinate number c, find the point in the subtree rooted at u that has the smallest c-coordinate.
- WLOG, we will assume c refers to the x-coordinate.

A Better Solution – A 2-d Tree

- The solution can be developed as follows. Assume for a moment that the level to which node u belongs splits the x-axis.
- Then, the point with the smallest x-coordinate can only be in the left subtree of u.
- But, the left subtree of u splits the y-coordinate. So, we have to search both the left and the right subtrees of the left child of u.

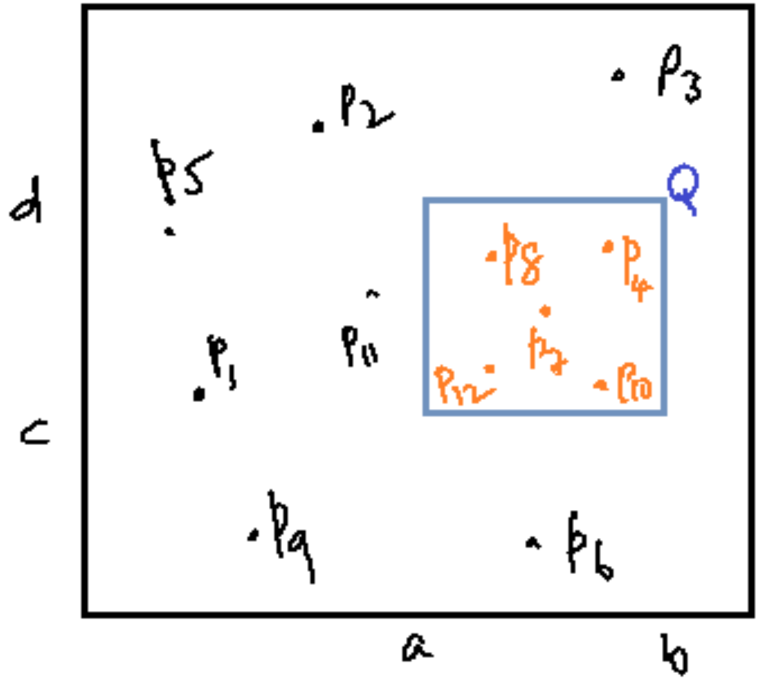
A Better Solution – A 2-d Tree

- Write pseudocode for FindMin(Node t, int c) .

```
Point FindMin(Node t, int c)
begin
        d = dimension that node t splits
        if (d == c) then
                return FindMin(t->left, c);
        else
                return min{t->data,
              FindMin(t->right, c),
              FindMin(t->left, c)
             }
end.
```

A Better Solution – A 2-d Tree

- To range searching: Given a query rectangle Q = [a,b] x [c,d], find the points that fall inside Q.
- Queries of this nature are called as orthogonal range queries.
- The following variant is also popular.
- Given a query rectangle Q = [a,b] x [c,d], count the number of points that fall inside Q.



A Better Solution – A 2-d Tree

- We will describe a recursive solution. Start from the root.
- Let the current node be u. Four cases:
  - If u is a null node, return.
  - If u is disjoint from Q, return.
  - If the cell corresponding to u is contained in Q, return the entire set of points in the leaves of the subtree at u.
  - Otherwise, we recurse on the two children of u after considering u itself.

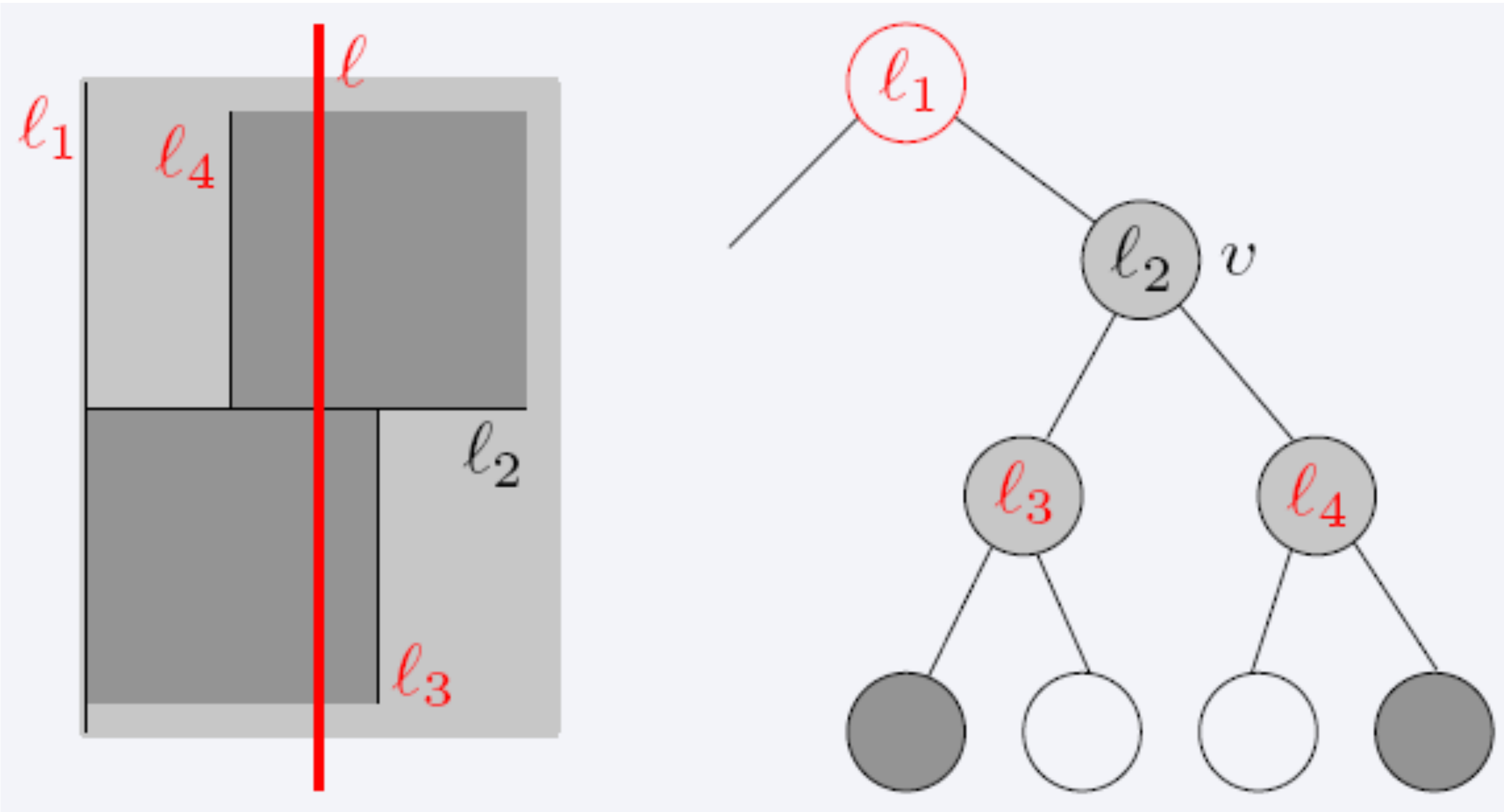
Analysis

- We claim that the above procedure visits O(sqrt{n}) nodes, assuming that the 2-d tree has a height of log n.
- Proof as follows.
  - Call a node u processed if the query algorithm visits both the children of the node. (In this case, the cell at u should overlap Q but not be completely contained in Q).
  - Every node can be associated with a cell.
  - We say that such a cell is stabbed by Q.

Analysis

- Consider a vertical line x = x0.
- We prove that the line x = x0 stabs no more than O(sqrt{n}) cells.
- Consider a node u which splits the x-axis. Then the line x=x0 stabs either the cell corresponding to the left child of u or the right child of u, but not both.
- Let the left child of u be ul. Then, ul splits the y-axis.
- However, the line x=x0 can stab the cells at both the children of ul.

Analysis



Analysis

- In summary, every two levels of the tree doubles the number of nodes stabbed.
- The recurrence relation that captures this phenomenon is T(n) = 2T(n/4) + O(1) with a solution of T(n) = O(sqrt{n}).

- Can use the above calculation four times over the four lines that make up Q.
- So, the total number of nodes stabbed by Q is O(sqrt{n}).

- Counting queries are also easy to support.
- Store the number of points inside each subtree. Add the required numbers.

Analysis

Master Theorem

The Master Theorem applies to recurrences of the following form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

There are 3 cases:

1. If $\boxed{f(n) = O(n^{\log_b a-\epsilon}) \text{ for some constant } \epsilon > 0\text{, then } T(n) = \Theta(n^{\log_b a}).}$
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with[1] $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$ with $\epsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$. Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

2-d Tree extended to k-dimensions

- The data structure is also known as the **kd-tree.**
- Slight *misnomer*, since k there refers to the dimension.
- What is the time complexity ?
- In practice, the query time is much smaller.
- So, the analysis may be rather pessimistic.

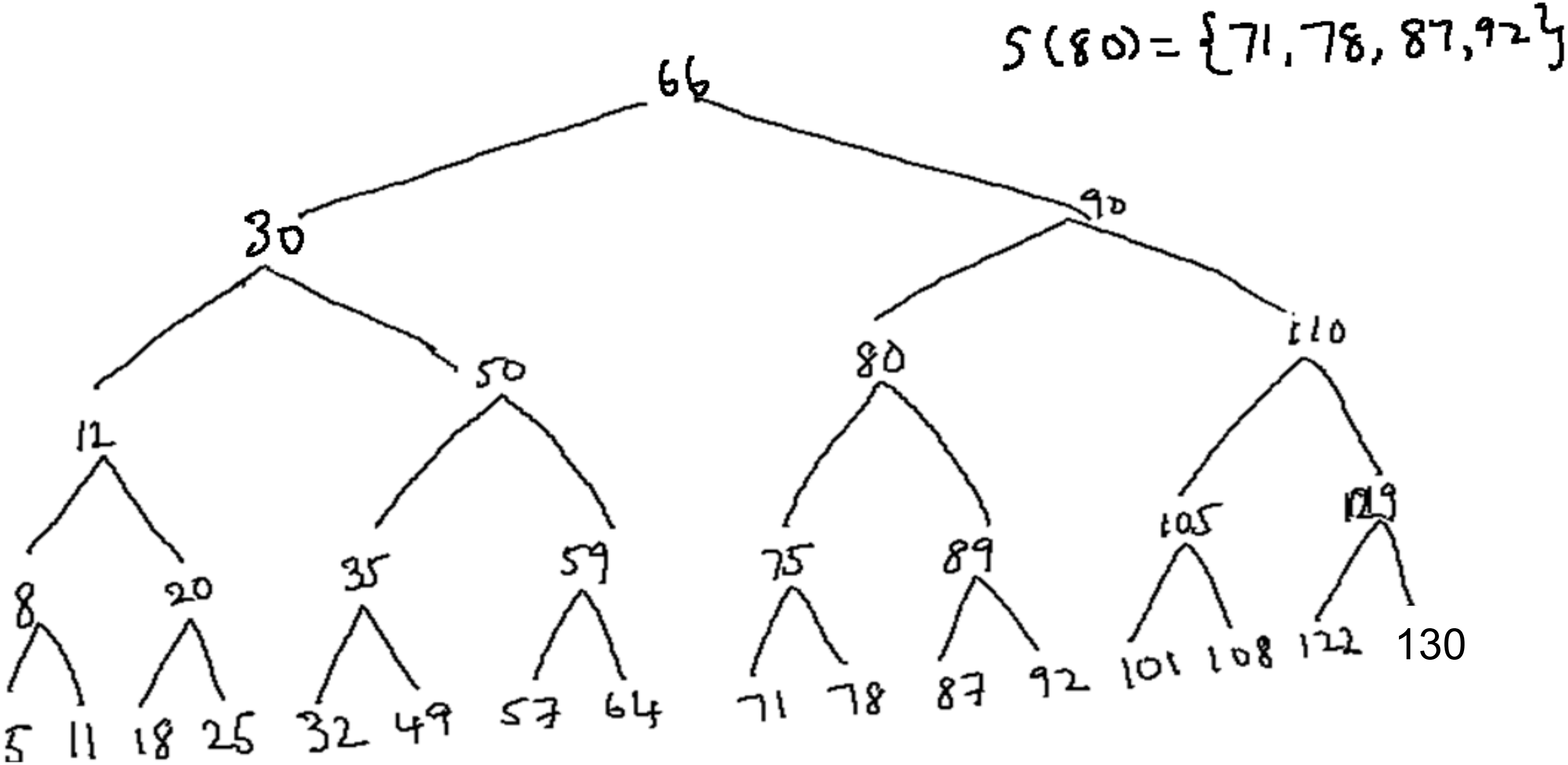
A Better Solution – Range Trees

- A query time of O(sqrt{n}) is quite high to bear in general.
- Range tree offers a better solution.
- We will study the 1-dimensional version first, and then extend the idea to higher dimensions.

1-d Range Trees

- Store all the data values in the leaves of the tree in increasing order.
- Internal nodes store only an index into the actual data values. (May not correspond to actual values).
- Rules similar to BST apply. If an internal nodes u stores a value x, then all the values in the left (right) subtree of u are smaller (larger) than x.
- With node u, can associate a set S(u) that contains values at the leaves in the subtree rooted at u.

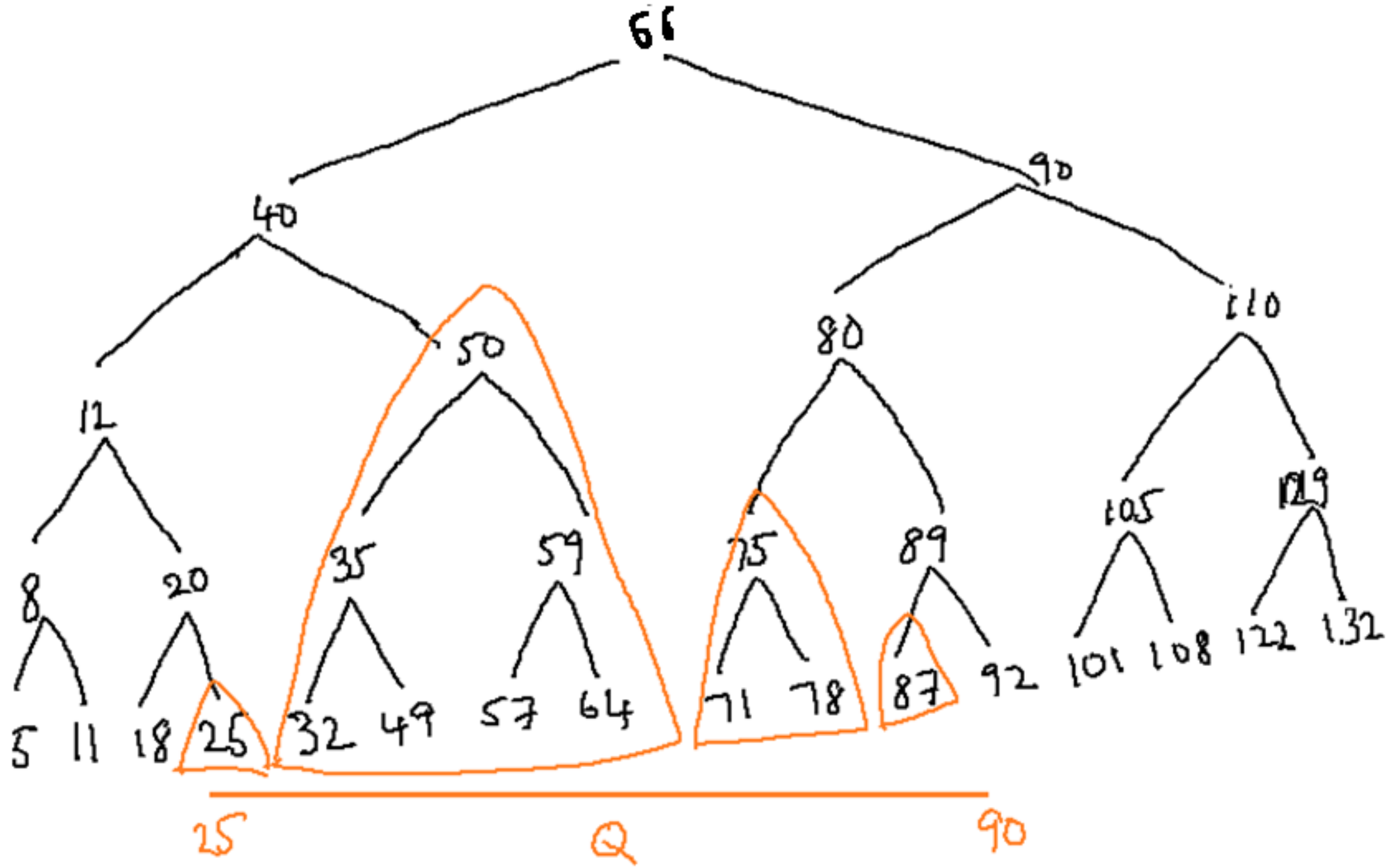
1-d Range Trees



1-d Range Trees

- Some notation.
- Let a query Q be [l, h].
- With respect to Q:
  - A node u is relevant if $S(u) \subseteq Q$.
  - A node u is canonical if u is relevant, but the parent of u is not relevant.
  - Call the subsets at canonical nodes as canonical subsets.
- Canonical nodes are therefore the roots of the maximal subtrees that are contained within Q.

1-d Range Trees



Some Observations

- Canonical subsets are non-overlapping and cover the interval Q.
- Canonical subsets can be identified in O(log n) time.
  - Recall that Q = [l, h].
  - Find the leftmost leaf node u whose value is at least l, and the rightmost leaf v whose value is at most h.
  - The path joining u and v has at most 2log n roots of nonoverlapping subtrees.
  - Taking the maximal roots from these 2log n roots gives us the canonical nodes and canonical subsets.

The Results in 1-d

- Putting together everything, we have:
- A set of n values can be preprocessed into a 1-d range tree T so that:
- T takes O(n) space,
  - T can be built in O(nlog n) time, and
  - Each reporting query can be answered in O(log n+ k) time.
  - Each counting query can be answered in O(log n) time.

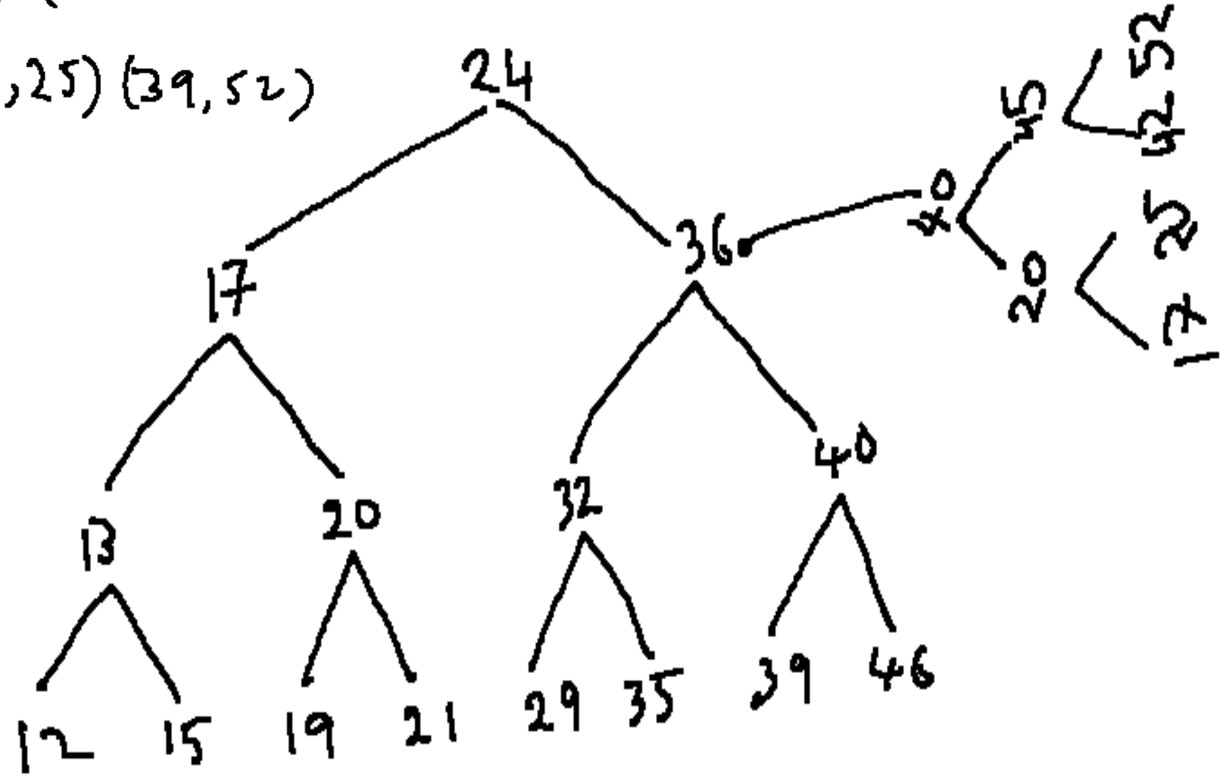
Extension to 2-d

- The 1-d range tree extends to 2-d as follows.
- Create a 1-d range tree for the x-coordinates of the points.
- For each node u, denote by S(u) the points in the subtree of u.
- Build another 1-d range tree for the points in S(u), using their y-coordinates. This is called as the *auxiliary tree* at u.

Extension to 2-d

(15,22) (35,17) (12,60) (19,37)

(21,16) (29,42) (46,25) (39,52)



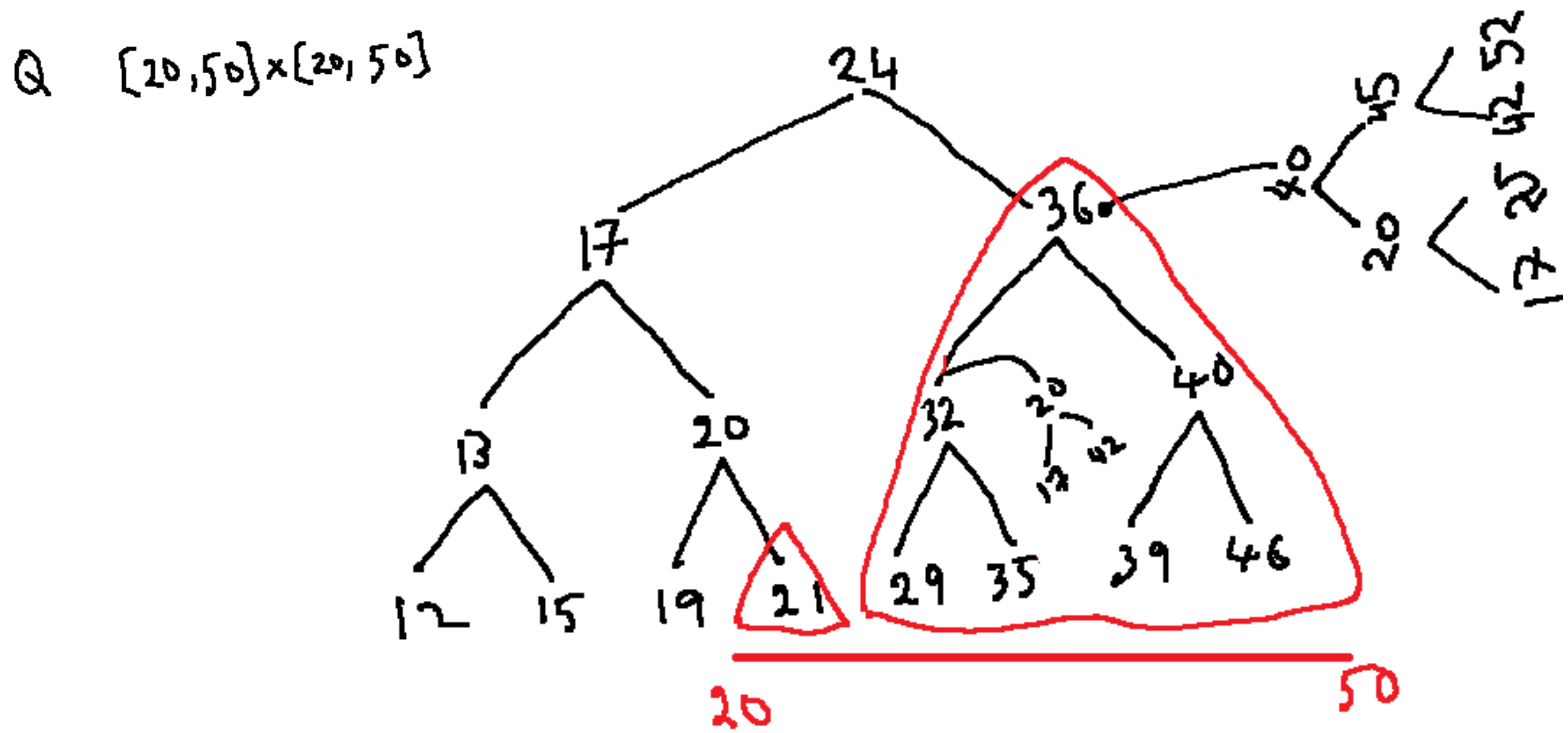
Extension to 2-d

- Some observations:
  - All the auxiliary trees require a total space of O(nlog n).
  - The overall space is also O(nlog n).
  - Can be constructed in O(nlog n) time.

Extension to 2-d

- A query Q = [a,b] x [c,d] can now be answered as follows.
- Identify the O(log n) canonical points of the 1-d range tree built on x-coordinates.
- All the points in the canonical subsets have their x-coordinates in Q but their y-coordinates may not be in Q.
- This is where the auxiliary trees help.
- Search the auxiliary trees of the canonical nodes to identify the required points.

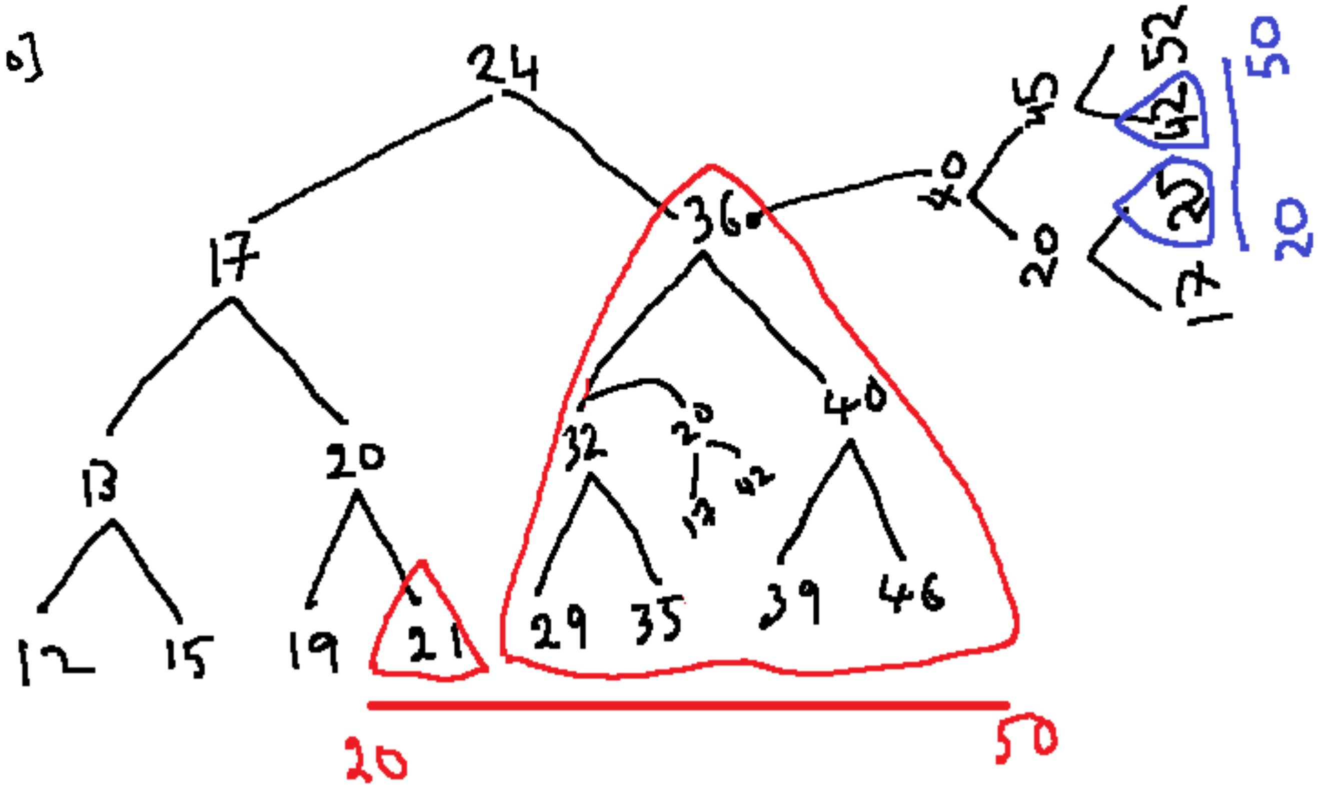
Extension to 2-d



Extension to 2-d

Q [20,50]x[20,50]

output (29,42)
(46,25)



Analysis

- There are at most O(log n) canonical nodes in the 1-d x-range tree.
- At each of these nodes, we perform a similar search in their auxiliary range trees.
- Each such search results in at most O($\log^2$ n) canonical nodes.
- So, the query time for reporting is O(k+$\log^2$ n), and for counting is O($\log^2$ n).
- In d-dimensions, the respective times are O(k+$\log^d$ n) and O($\log^d$ n).

Thank You