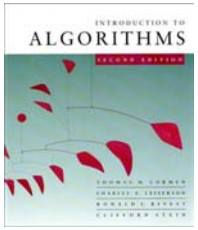
CS 5633 -- Spring 2008



Range Trees

Carola Wenk

Slides courtesy of Charles Leiserson with small changes by Carola Wenk

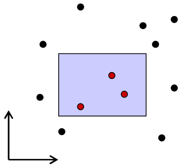
Orthogonal range searching

Input: n points in d dimensions

- E.g., representing a database of n records each with d numeric fields

Query: Axis-aligned ***box*** (in 2D, a rectangle)

- Report on the points inside the box:
  - Are there any points?
  - How many are there?
  - List the points.



Orthogonal range searching

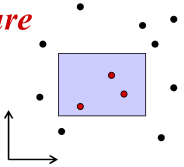
Input: n points in d dimensions

Query: Axis-aligned ***box*** (in 2D, a rectangle)

- Report on the points inside the box

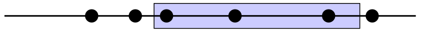
Goal: Preprocess points into a data structure to support fast queries

- Primary goal: ***Static data structure***
- In 1D, we will also obtain a dynamic data structure supporting insert and delete



1D range searching

In 1D, the query is an interval:



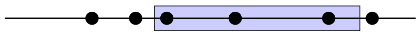
First solution:

- Sort the points and store them in an array
  - Solve query by binary search on endpoints.
  - Obtain a static structure that can list k answers in a query in $O(k + \log n)$ time.

Goal: Obtain a dynamic structure that can list k answers in a query in $O(k + \log n)$ time.

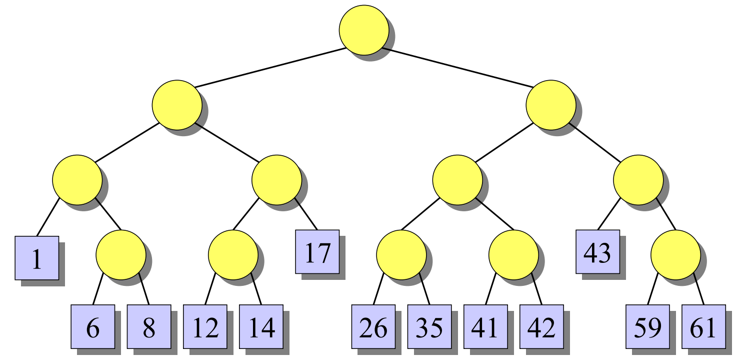
1D range searching

In 1D, the query is an interval:



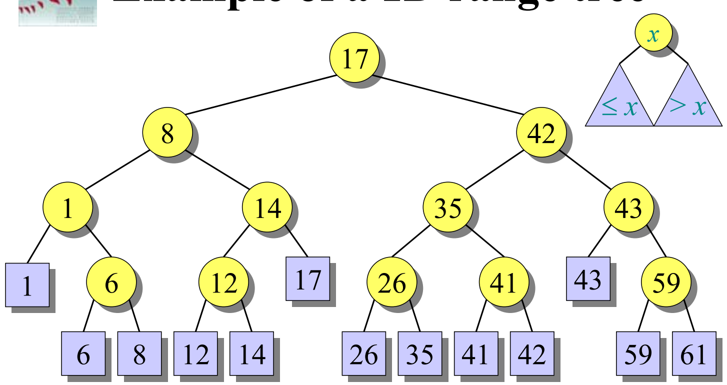
New solution that extends to higher dimensions:
- Balanced binary search tree
  - New organization principle: Store points in the *leaves* of the tree.
  - Internal nodes store copies of the leaves to satisfy binary search property:
    - Node x stores in *key*[x] the maximum key of any leaf in the left subtree of x.

Example of a 1D range tree



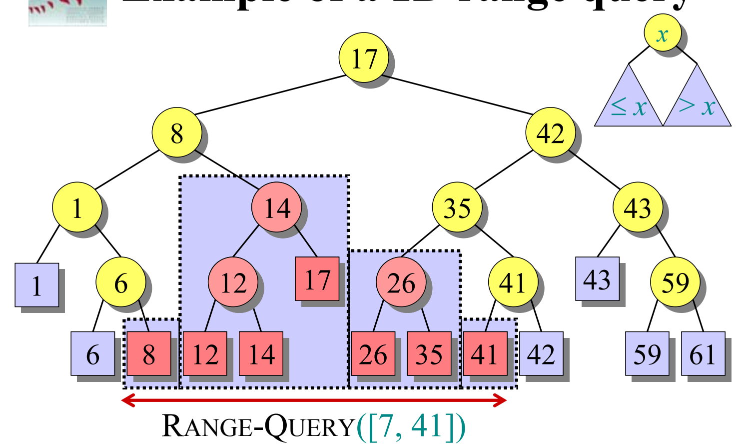
key[x] is the maximum key of any leaf in the left subtree of x.

Example of a 1D range tree

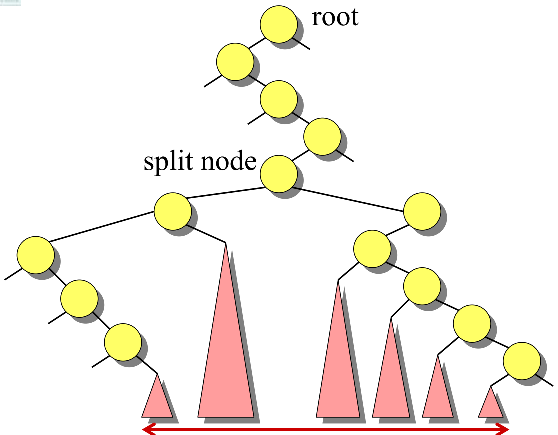


key[x] is the maximum key of any leaf in the left subtree of x.

Example of a 1D range query

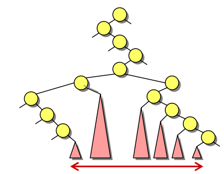


General 1D range query



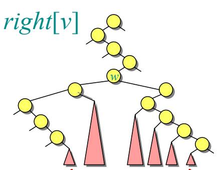
Pseudocode, part 1: Find the split node

1D-RANGE-QUERY$(T, [x_1, x_2])$
- $w \leftarrow$ root$[T]$
- **while** w is not a leaf and $(x_2 \leq key[w]$ or $key[w] < x_1)$
  - **do if** $x_2 \leq key[w]$
    - **then** $w \leftarrow left[w]$
    - **else** $w \leftarrow right[w]$
- // w is now the split node
- [*traverse left and right from w and report relevant subtrees*]



Pseudocode, part 2: Traverse left and right from split node

1D-RANGE-QUERY$(T, [x_1, x_2])$
- [*find the split node*]
- // w is now the split node
- **if** w is a leaf
  - **then** output the leaf w if $x_1 \leq key[w] \leq x_2$
  - **else** $v \leftarrow left[w]$ // Left traversal
    - **while** v is not a leaf
      - **do if** $x_1 \leq key[v]$
        - **then** output the subtree rooted at $right[v]$
          - $v \leftarrow left[v]$
        - **else** $v \leftarrow right[v]$
    - output the leaf v if $x_1 \leq key[v] \leq x_2$
    - [*symmetrically for right traversal*]



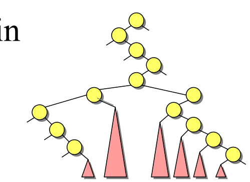
Analysis of 1D-RANGE-QUERY

Query time: Answer to range query represented by $O(\log n)$ subtrees found in $O(\log n)$ time.

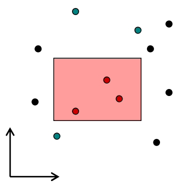
Thus:
- Can test for points in interval in $O(\log n)$ time.
- Can report all k points in interval in $O(k + \log n)$ time.
- Can count points in interval in $O(\log n)$ time (exercise)

Space: $O(n)$

Preprocessing time: $O(n \log n)$



2D range trees

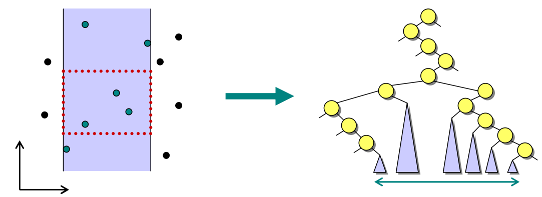


2D range trees

Store a ***primary*** 1D range tree for all the points based on x-coordinate.

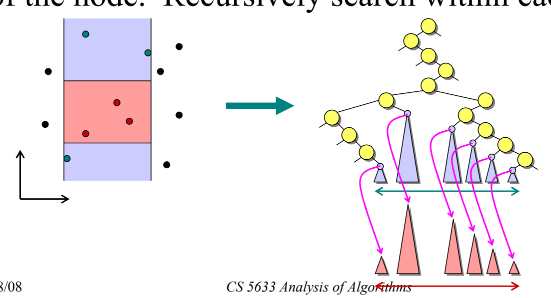
Thus in $O(\log n)$ time we can find $O(\log n)$ subtrees representing the points with proper x-coordinate.

How to restrict to points with proper y-coordinate?



2D range trees

Idea: In primary 1D range tree of x-coordinate, every node stores a ***secondary*** 1D range tree based on y-coordinate for all points in the subtree of the node. Recursively search within each.



Analysis of 2D range trees

Query time: In $O(\log^2 n) = O((\log n)^2)$ time, we can represent answer to range query by $O(\log^2 n)$ subtrees. Total cost for reporting k points: $O(k + (\log n)^2)$.

Space: The secondary trees at each level of the primary tree together store a copy of the points. Also, each point is present in each secondary tree along the path from the leaf to the root. Either way, we obtain that the space is $O(n \log n)$.

Preprocessing time: $O(n \log n)$

d-dimensional range trees

Each node of the secondary y-structure stores a tertiary z-structure representing the points in the subtree rooted at the node, etc.

Query time: $O(k + \log^d n)$ to report k points.
Space: $O(n \log^{d-1} n)$
Preprocessing time: $O(n \log^{d-1} n)$

Best data structure to date:
Query time: $O(k + \log^{d-1} n)$ to report k points.
Space: $O(n (\log n / \log \log n)^{d-1})$
Preprocessing time: $O(n \log^{d-1} n)$