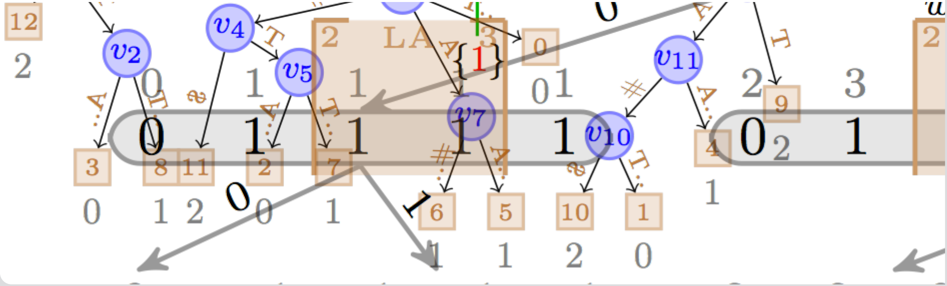
Karlsruhe Institute of Technology

Advanced Data Structures

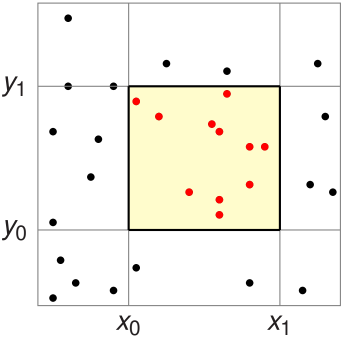
Simon Gog – *gog@kit.edu*

Institute of Theoretical Informatics - Algorithmics

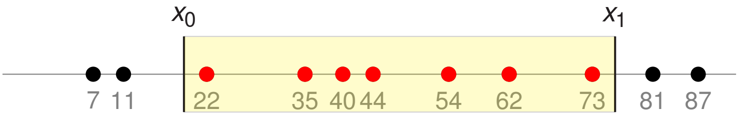


Orthogonal range searching

Classical OLAP queries: „Find all users aged between 30 and 35 which are connected to at least 100 and at most 200 other users"



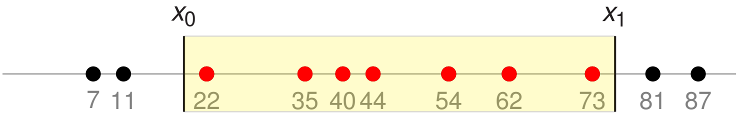
Orthogonal range searching – 1D



One dimensional case ($d = 1$). Example ($x_0 = 19$, $x_1 = 76$):

- count(x_0, x_1)=8
- report(x_0, x_1)=$\{22, 35, 40, 44, 54, 62, 73\}$

Orthogonal range searching – 1D



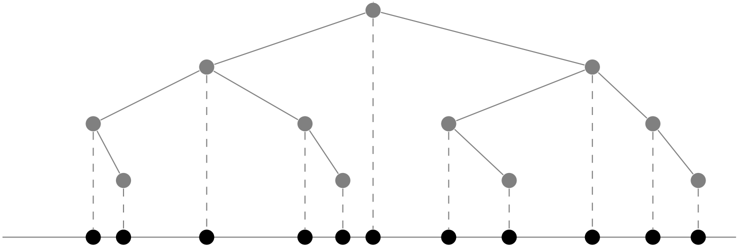
Simple solution

- Sort points according to x-coordinates ($O(n \log n)$) and store them in array A
- Calculate successor x_0' of x_0 and predecessor x_1' of x_1
- Let i' (j') be the index of x_0' (x_1') in A
- Method *count* returns $k = j' - i' + 1$ (in $O(\log n)$ time)
- Method *report* returns subarray $A[i', j']$ (in $O(\log n + k)$ time)

Orthogonal range searching – 1D

Karlsruhe Institute of Technology

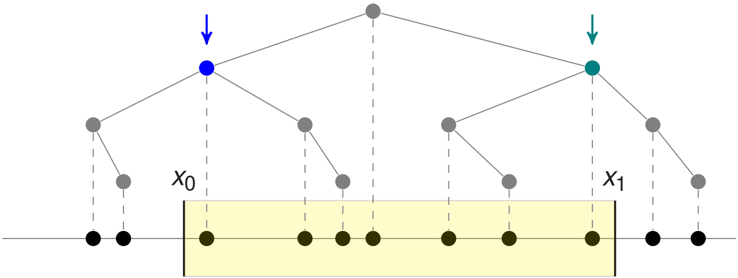
Alternative solution: balanced binary search trees



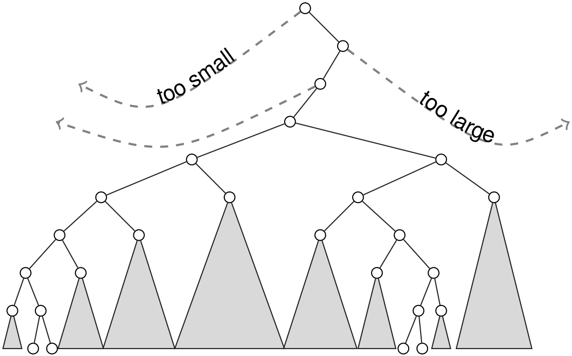
- Find point in middle, split set and recurse on both half (pick left point if set size is even)
- Depth is $\log n$, construction time is bounded by sorting ($O(n \log n)$)

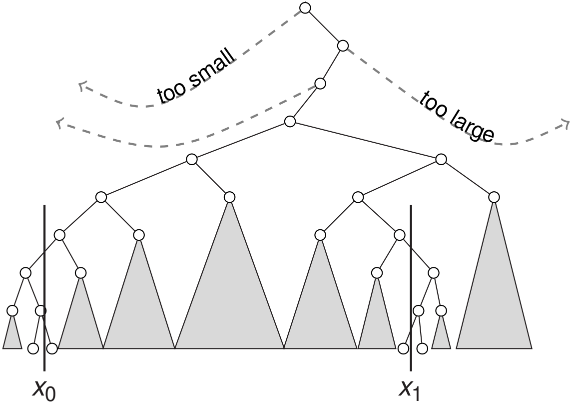
Orthogonal range searching – 1D

Alternative solution: balanced binary search trees

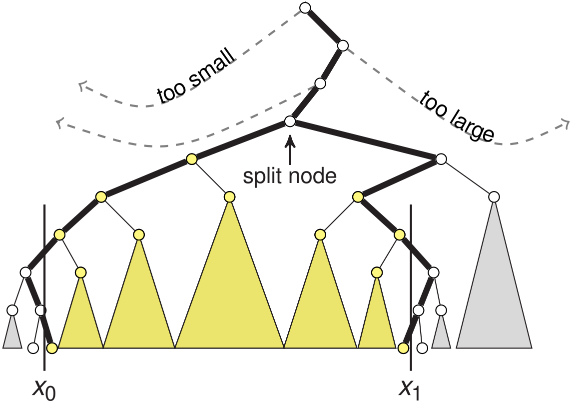


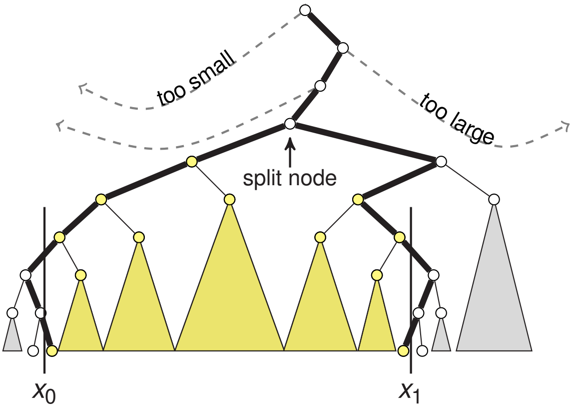
- Find successor (predecessor) of x_0 (x_1) again


too small
too large



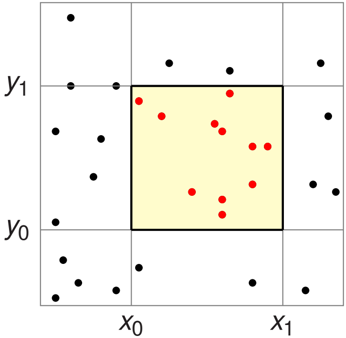
too small
too large
x_0
x_1


too small
too large
split node
x_0
x_1



- Subtrees of off-path edges are either included or excluded form result
- Result can be implicitly represented using included off-path subtrees (there are at most $O(\log n)$ of them)

Orthogonal range searching – 2D



Two dimensional case ($d = 2$). Example ($x_0 = 12$, $x_1 = 32$, $y_0 = 10, y_1 = 29$):

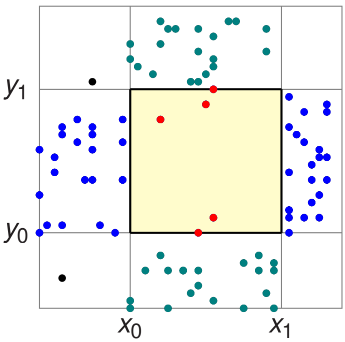
- $count(x_0, x_1, y_0, y_1) = 11$
- $report(x_0, x_1, y_0, y_1) = \ (19, 40), \ (23, 39), \ (22, 49), \ \cdots$

Orthogonal range searching – 2D

First attempt of a solution

- Find all points with $x_0 \leq x \leq x_1$
- Find all points with $y_0 \leq y \leq y_1$
- Intersect the two resulting list
- Time complexity for this approach:
  $O(\log n + k_x) + O(\log n + k_y) + O(k_x + k_y) = O(\log n + k_x + k_y)$,
  where k_x and k_y is the length of the two lists
- Well, there are cases...

Orthogonal range searching – 2D



$k_x = 45$

$k_y = 49$

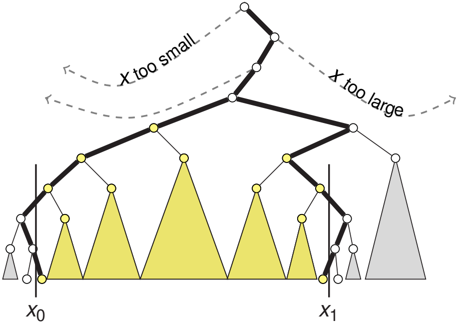
$k = 5$

$n = 91$

Orthogonal range search – 2D

Second attempt

- Build a balanced binary tree using the x coordinates
- Calculate the $O(\log n)$ subtrees which contain all points with $x_0 \leq x \leq x_1$
- Idea: Filter these subtrees by y-coordinate



Orthogonal range searching – 2D

How to filter by y-coordinate

- For each node v in the tree build a 1D range searching structure on the y-coordinates of all points in v's subtree
- This can be done during the preprocessing
- How does the query process change?
  - Determine paths to successor and predecessor of x_0 and x_1
  - Determine the root nodes of the $O(\log n)$ included off-path subtrees
  - For each such root node v_i retrieve all points which are in $[y_0, y_1]$ in $O(\log n + k_i)$ time, where k_i is the number of matching points

Total time complexity: $O(\log^2 n + k)$

- At most $O(\log n)$ subtrees for x
- Retrieval time for each subtree $O(\log n + k_i)$
- Points from two different subtrees are *distinct*

Orthogonal range searching – 2D

How to filter by y-coordinate

- For each node v in the tree build a 1D range searching structure on the y-coordinates of all points in v's subtree
- This can be done during the preprocessing
- How does the query process change?
  - Determine paths to successor and predecessor of x_0 and x_1
  - Determine the root nodes of the $O(\log n)$ included off-path subtrees
  - For each such root node v_i retrieve all points which are in $[y_0, y_1]$ in $O(\log n + k_i)$ time, where k_i is the number of matching points

Total time complexity: $O(\log^2 n + k)$

- At most $O(\log n)$ subtrees for x
- Retrieval time for each subtree $O(\log n + k_i)$
- Points from two different subtrees are *distinct*

Orthogonal range searching – 2D

How much space is used?

- Tree high is $O(\log n)$
- On each level ℓ each point is represented in only one node
- $\sum_{i=0}^{c_1 \log n} c_2 n = O(n \log n)$ words

How long does the preprocessing take?

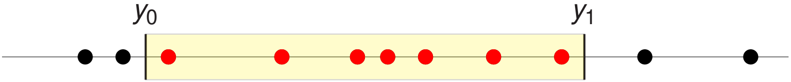
- Problem: points have to be sorted according to y-coordinate in each node
- Solution: bottom-up construction
  - Start at the leaves
  - Merge the (already sorted) lists of the two children of a node
  - I.e. $O(n \log n)$ construction time

Orthogonal range searching – 2D

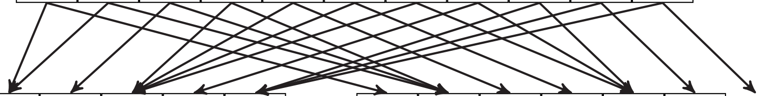
2d-range searching in $O(\log n + k)$ time

- Idea: Avoid expensive calcuation of successor/predecessor in all $O(\log n)$ 1d-range structures for y-coordinates
- Determine successor/predecessor in root node and map result into child nodes
- Technique known as *fractional cascading*
- More detailed: For each node v and entry of the y-range searching structure, store a pointer the corresponding successor in v's left and right child

Orthogonal range searching – 2D



6	11	17	32	42	46	51	60	69	80	94



11	17	51	60	94

6	32	42	46	69	80