RANGE TREES AND PRIORITY SEARCH TREES

Hanan Samet

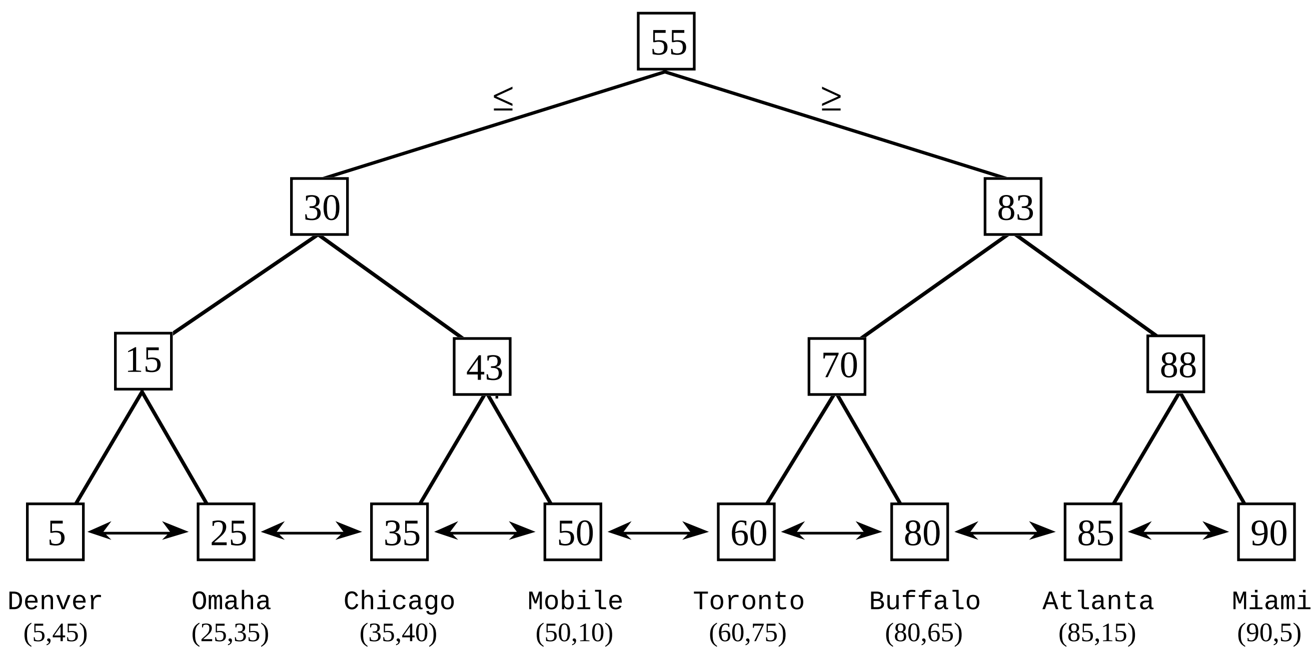
Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
e-mail: hjs@umiacs.umd.edu

RANGE TREES

- Balanced binary search tree
- All data stored in the leaf nodes
- Leaf nodes linked in sorted order by a doubly-linked list
- Searching for [B : E]

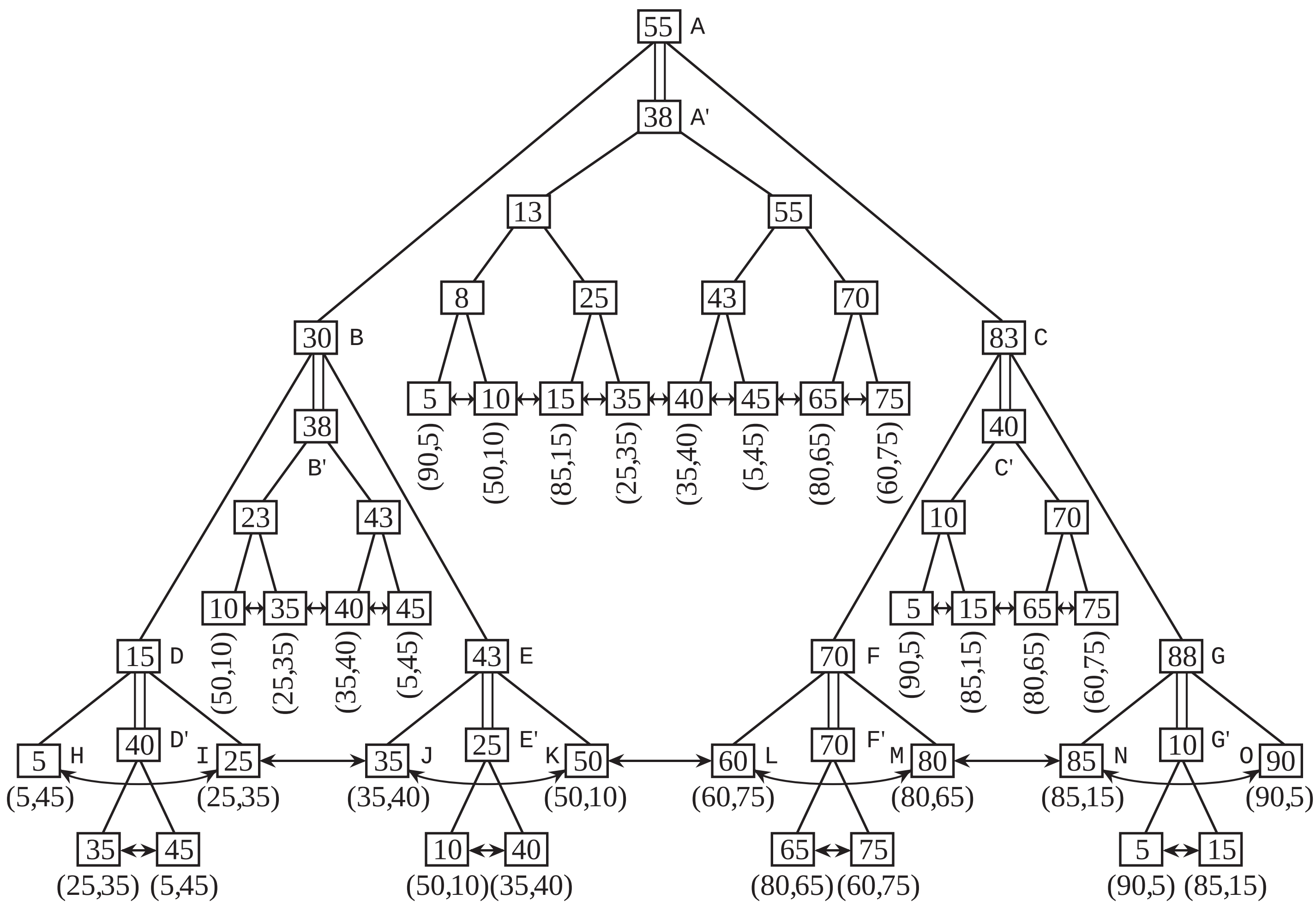
  1. find node with smallest value ≥B or largest ≤B
  2. follow links until reach node with value >E

- $O(\log_2 N + F)$ time to search, $O(N \cdot \log_2 N)$ to build, and $O(N)$ space for N points and F answers
- Ex: sort points in 2-d on their x coordinate value



2-D RANGE TREES

- Binary tree of binary trees
- Sort all points along one dimension (say *x*) and store them in the leaf nodes of a balanced binary tree such as a range tree (single line)
- Each nonleaf node contains a 1-d range tree of the points in its subtrees sorted along *y* (double lines)
- Ex:

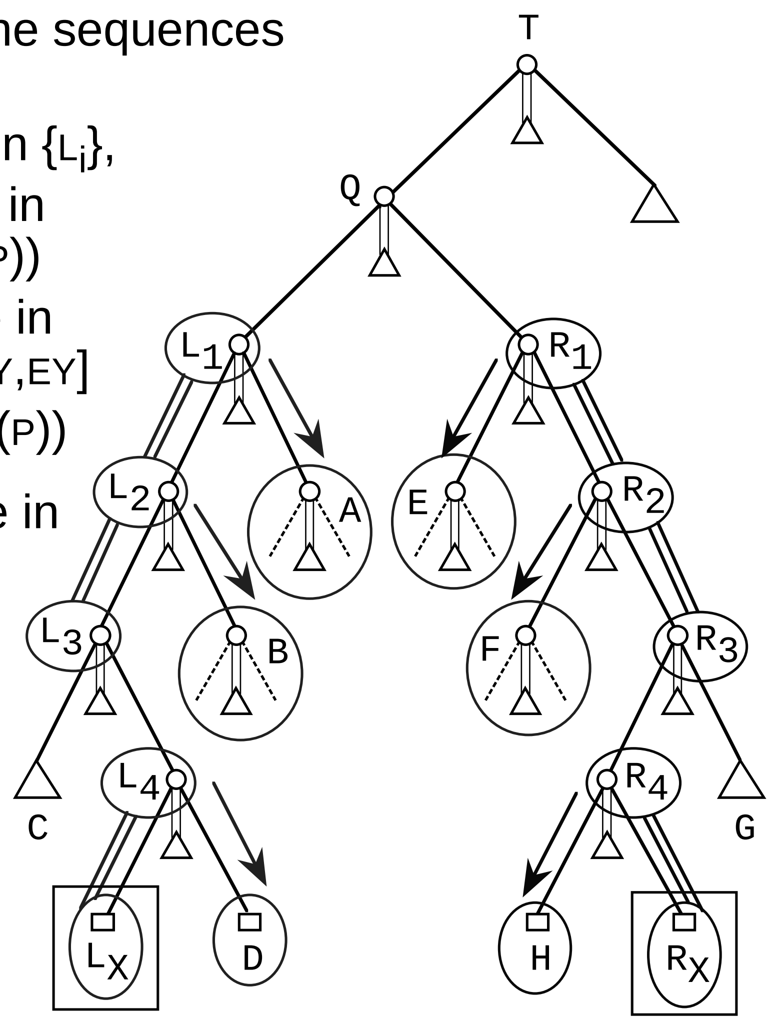


- Actually, don't need the 1-d range tree in *y* at the root and at the sons of the root

SEARCHING 2-D RANGE TREES ([BX:EX],[BY:EY])

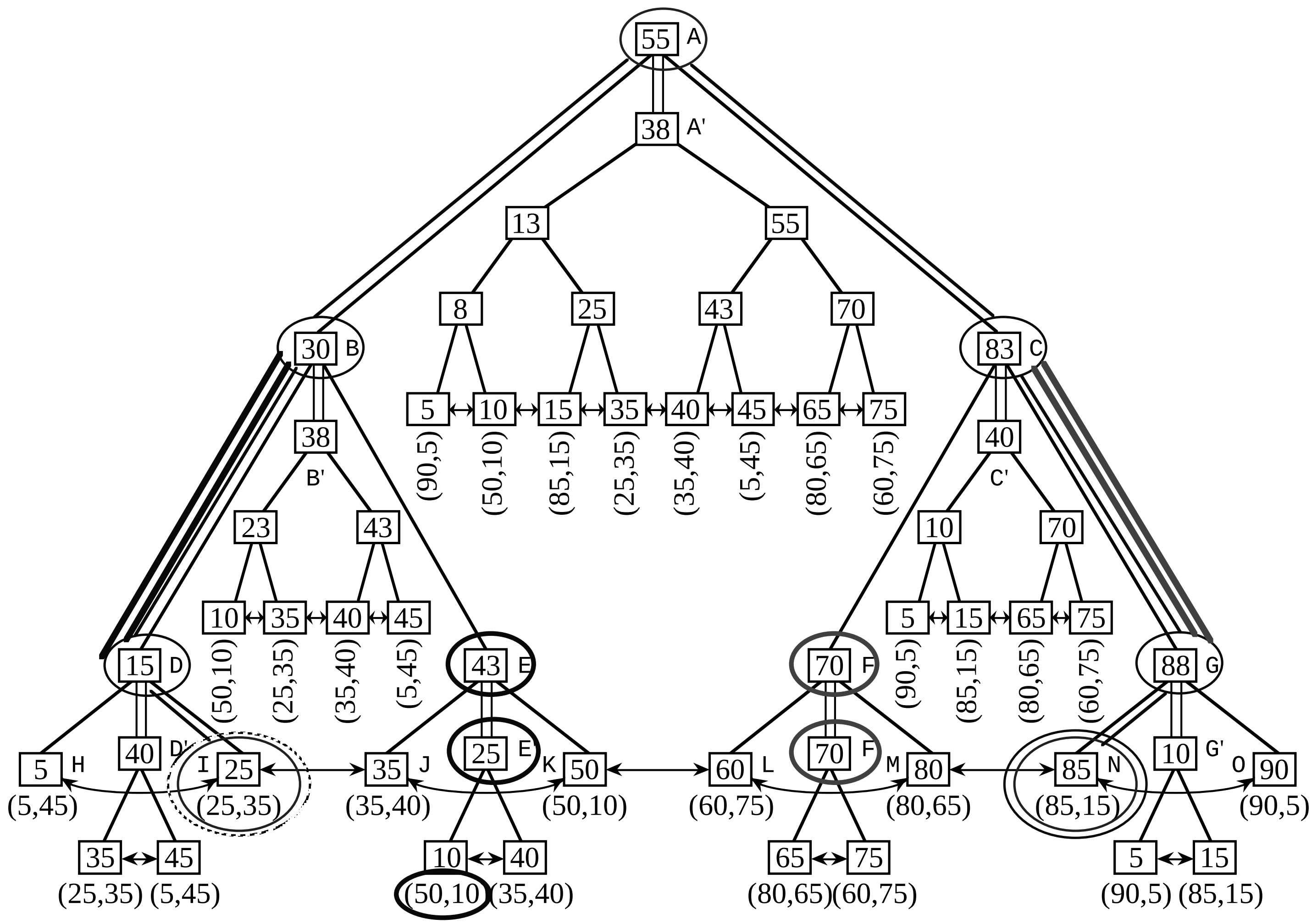
1. Search tree T for nodes BX and EX
   - find node LX with a minimum value ≥ BX
   - find node RX with a maximum value ≤ EX
2. Find their nearest common ancestor Q
3. Compute $\{L_i\}$ and $\{R_i\}$, the sequences of nodes forming the paths from Q to LX and RX, respectively (including LX and RX but excluding Q)
   - LEFT(P) and RIGHT(P) are sons of P
   - MIDRANGE(P) discriminates on *x* coordinate value
   - RANGE_TREE(P) denotes the 1-d range tree stored at P
4. For each element in the sequences $\{L_i\}$ and $\{R_i\}$ do
   - if P and LEFT(P) are in $\{L_i\}$, then look for [BY,EY] in RANGE_TREE(RIGHT(P))
   - if P and RIGHT(P) are in $\{R_i\}$, then look for [BY,EY] in RANGE_TREE(LEFT(P))
5. Check if LX and RX are in ([BX:EX],[BY:EY])

- Total $O(\log_2^2 N + F)$ time to search and $O(N \cdot \log_2 N)$ space and time to build for N points and F answers



EXAMPLE OF SEARCH IN A 2-D RANGE TREE

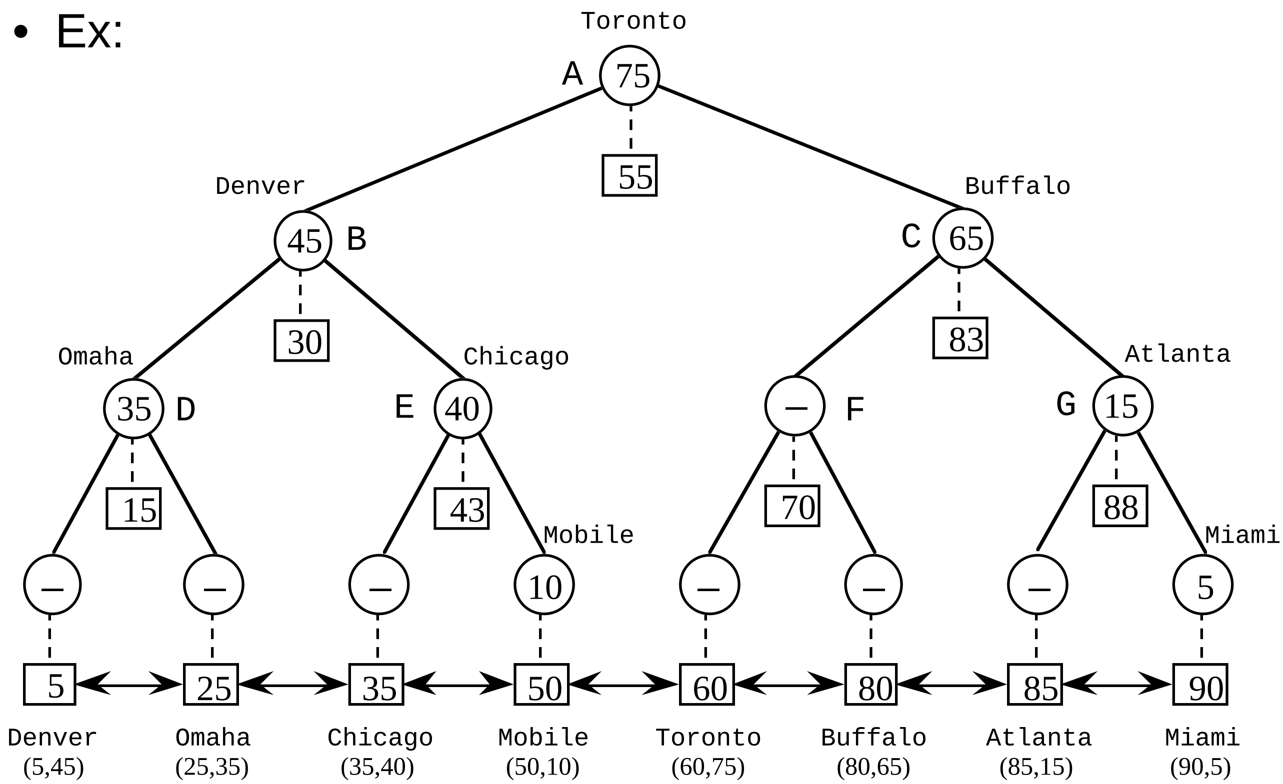
- Find all points in ([25:85],[8:16])



1. Find nearest common ancestor — i.e., A
2. Find paths to LX=25 and RX=85
3. Look in subtrees
   - B and B’s left son D are in path, so search range tree of B’s right son E and report (50,10)
   - C and C’s right son G are in path, so search range tree of C’s left son F and report none
4. Check boundaries of *x* range (i.e., (25,35) and (85,15)) and report (85,15)

PRIORITY SEARCH TREES

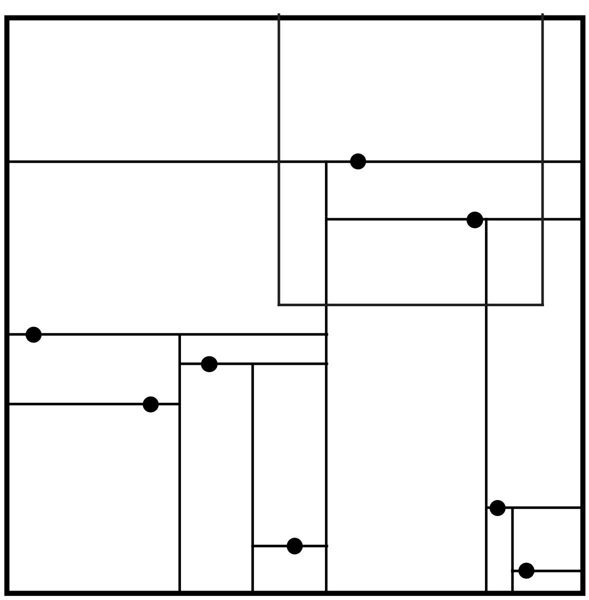
- Sort all points by their *x* coordinate value and store them in the leaf nodes of a balanced binary tree (i.e., a range tree)
- Starting at the root, each node contains the point in its subtree with the maximum value for its *y* coordinate that has not been stored at a shallower depth in the tree; if no such node exists, then node is empty
- $O(N)$ space and $O(N \cdot \log_2 N)$ time to build for *N* points
- Result: range tree in *x* and heap (i.e., priority queue) in *y*
- Ex:



- Good for semi-infinite ranges — i.e., ([BX:EX],[BY:∞])
- Can only perform a 2-d range query if find ([BX:EX],[BY:∞]) and discard all points (*x*,*y*) such that *y* > EY
- No need to link leaf nodes unless search for all points in range of *x* coordinate values

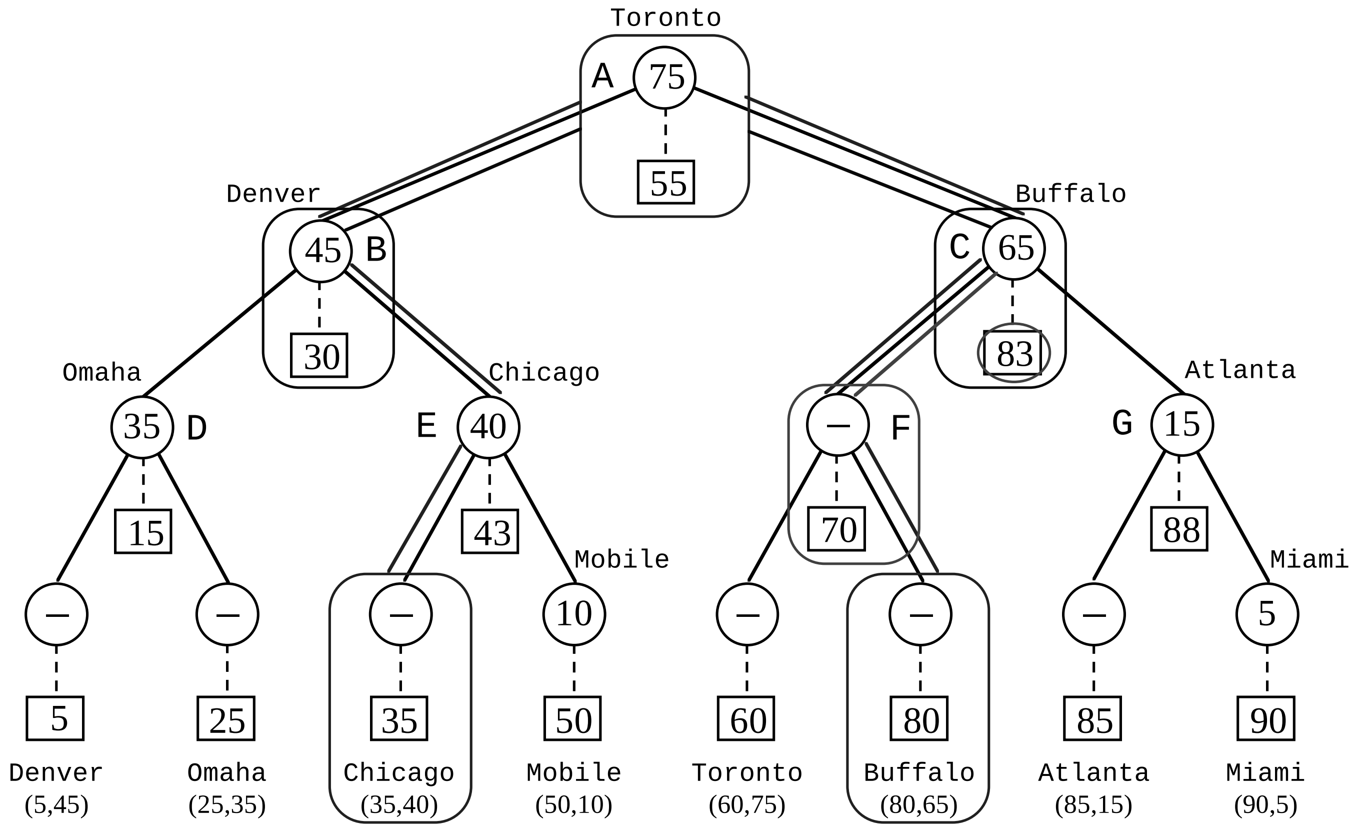
SEMI-INFINITE RANGE QUERY ON A PRIORITY SEARCH TREE ([BX:EX],[BY:∞])

- Procedure
  1. Descend tree looking for the nearest common ancestor of BX and EX — i.e., Q
     - associated with each examined node T is a point P
     - exit if P does not exist as all points in the subtrees have been examined and/or reported
     - exit if P_y < BY as P is point with maximum *y* coordinate value in T
     - otherwise, output P if P_x is in [BX:EX]
  2. Once Q has been found, process left and right subtrees applying the tests above to their root nodes T
     - T in left (right) subtree of Q:
       a. check if BX (EX) in LEFT(T) (RIGHT(T))
       b. yes: all points in RIGHT(T) (LEFT(T)) are in *x* range
          - check if in *y* range
          - recursively apply to LEFT(T) (RIGHT(T))
       c. no: recursively apply to RIGHT(T) (LEFT(T))
- $O(\log_2 N + F)$ time to search for *N* points and *F* answers
- Ex: Find all points in ([35:80],[50:∞])



EXAMPLE OF A SEARCH IN A PRIORITY SEARCH TREE

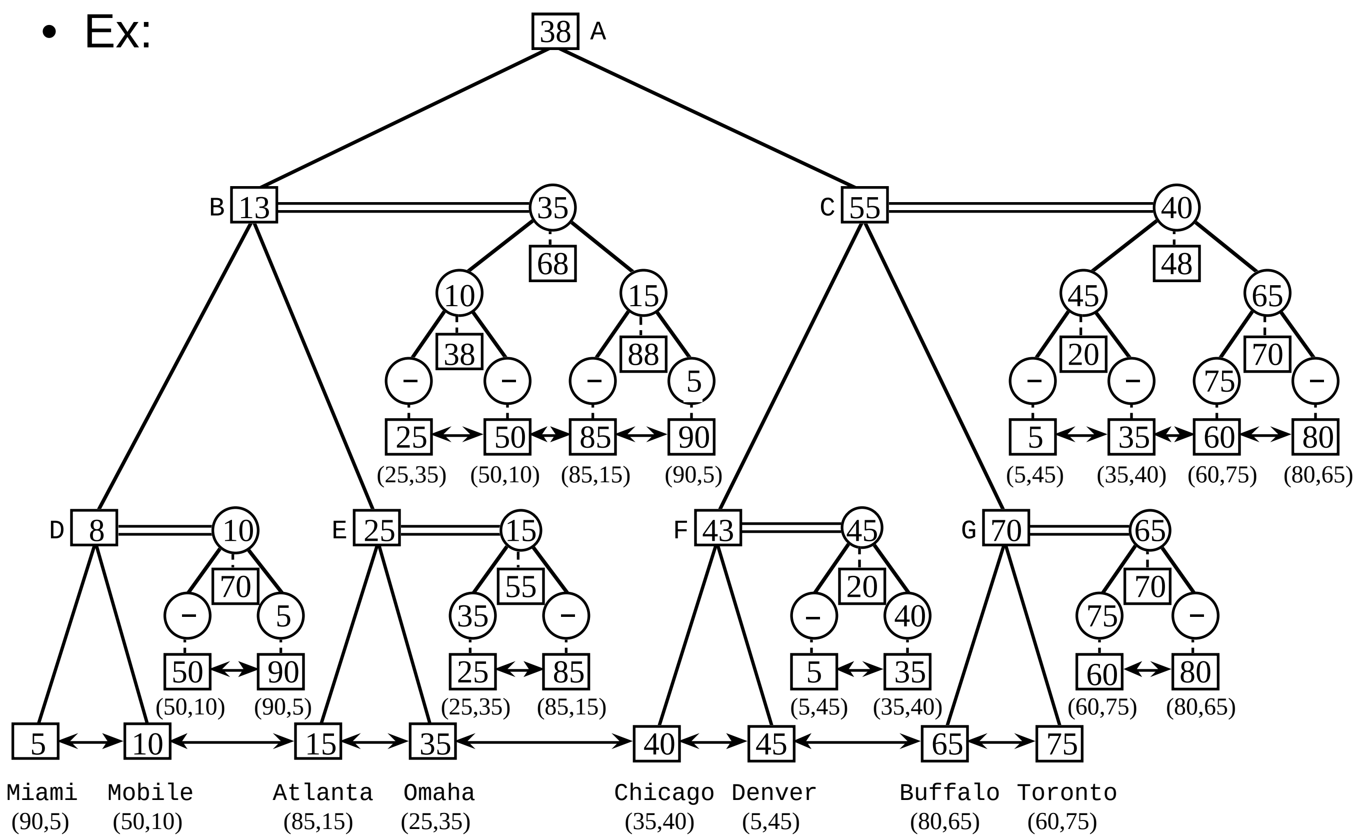
- Find all points in ([35:80],[50:∞])



1. Find nearest common ancestor — i.e., A
   - output Toronto (60,75) since 60 is in [35:80] and 75≥50
2. Process left subtree of A (i.e., B)
   - cease processing as 45<50
3. Process right subtree of A (i.e., C)
   - output (80,65) as 65 ≥ 50 and 80 is in [35:80]
4. Examine midrange value of C which is 83 and descend left subtree of C (i.e., F)
   - cease processing since no point is associated with F meaning all nodes in the subtree have been examined

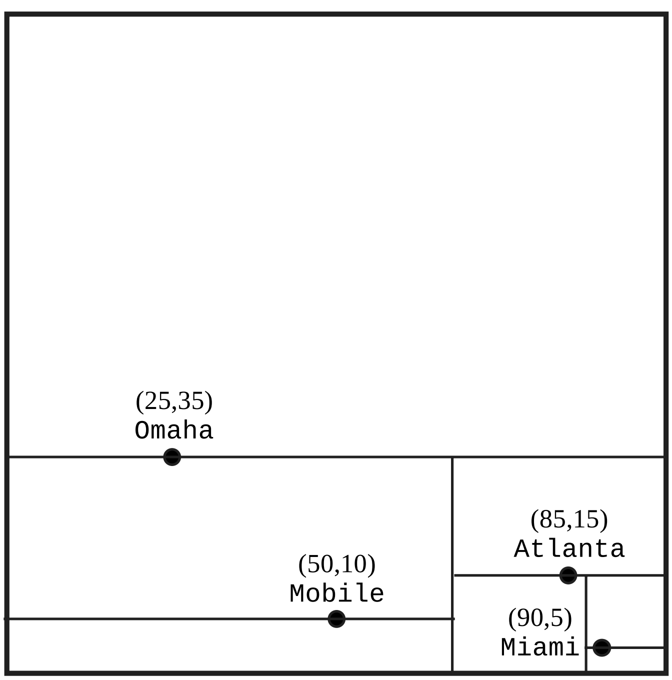
RANGE PRIORITY TREES

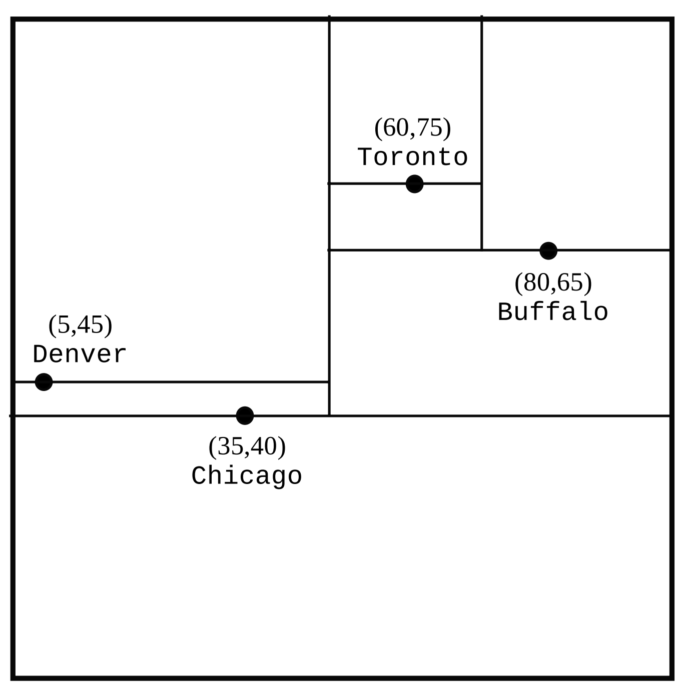
- Variation on priority search tree
- Inverse priority search tree: heap node stores point with minimum *y* coordinate value that has not been stored in a shallower depth in the tree (instead of maximum)
- Structure
  1. sort all points by their *y* coordinate value and store in leaf of a balanced binary tree such as range tree (single lines)
     - no need to link leaf nodes unless search for all points in range of *x* coordinate values
  2. nonleaf node left sons of their father contains a priority search tree of points in subtree (double lines)
  3. nonleaf node right sons of their father contains an inverse priority search tree of points in subtree (double lines)
- $O(N \cdot \log_2 N)$ space and time to build for N points
- Ex:



SEARCHING A RANGE PRIORITY TREE ([BX:EX],[BY:EY])

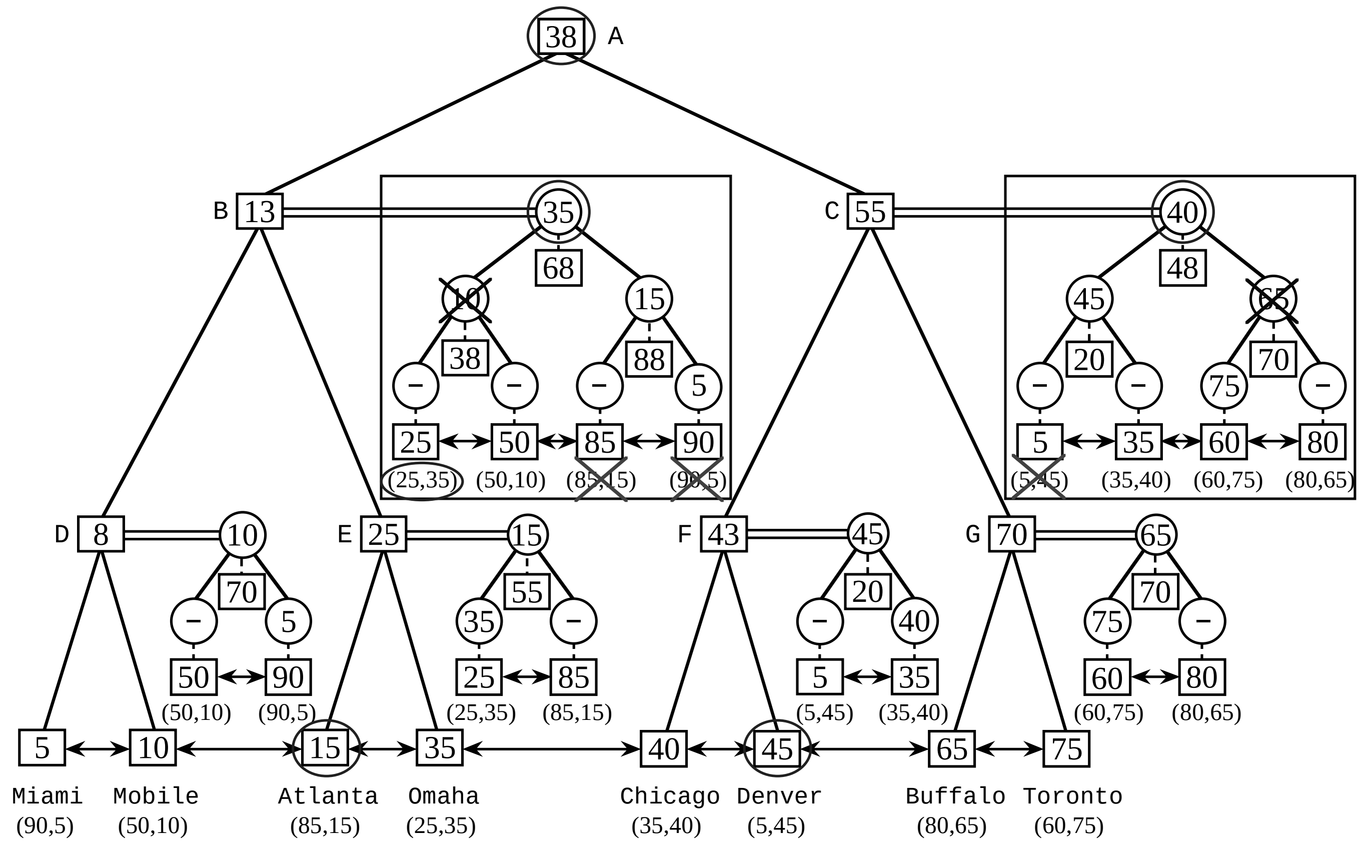
- Procedure
  1. find nearest common ancestor of BY and EY — i.e., Q
  2. all points in LEFT(Q) have *y* coordinate values <EY
     - want to retrieve just the ones ≥BY
     - find them with ([BX:EX],[BY:∞]) on priority tree of LEFT(Q)
     - priority tree is good for retrieving all points with a specific lower bound as it stores an upper bound and hence irrelevant values can be easily pruned
  3. all points in RIGHT(Q) have *y* coordinate values >BY
     - want to retrieve just the ones ≤EY
     - find them with ([BX:EX],[–∞:EY]) on the inverse priority tree of RIGHT(Q)
     - inverse priority tree is good for retrieving all points with a specific upper bound as it stores a lower bound and hence irrelevant values can be easily pruned
- $O(\log_2 N + F)$ time to search for *N* points and *F* answers





EXAMPLE OF A SEARCH IN A RANGE PRIORITY TREE

- Find all points in ([25:60],[15:45])



1. Find nearest common ancestor of 15 and 45 — i.e., A
2. Search for ([25:60],[15:∞]) in priority tree hanging from left son of A — i.e., B (all with *y*≤45 since a range tree in *y* and in left subtree of a node with *y* midrange value of 38)
   - output (25,35) as in range
   - reject left subtree as 10 < lower limit of *y* range
   - reject items in right subtree as out of *x* range
3. Search for ([25:60],[–∞:45]) in inverse priority tree hanging from right son of A — i.e., C (all with *y*≥15 since in right subtree of a node with *y* midrange value of 38)
   - output (35,40) as in range
   - reject unreported items in left subtree as out of *x* range
   - reject right subtree as 65 > upper limit of *y* range