Today: Augmentation
- easy tree augmentation
- order-statistic trees
- finger search trees
- range trees

Idea: modify "off-the-shelf" data structure to store additional information → updates

Easy tree augmentation:
- goal: store \( f(\text{subtree rooted at } x) \) at each node \( x \) in \( x.f \)
- suppose \( x.f \) can be computed in \( O(1) \) time from \( x, \text{children, & children}.f \)
- if modify set \( S \) of nodes (data, children) then costs \( O(\#\text{ancestors of nodes in } S) \) to update \( f(x) \)'s (walk up from \( S \))
- so \( O(\log n) \) updates in
- AVL trees: e.g. rotate
  -> update \( y \) then \( x \)
- 2-3 trees: e.g. split
  -> update \( x \) & \( z \)
(→ also update up the tree)
Order-statistic trees: (from 6.006) (Abstract Data Type)
- **ADT/interface:**
  - \( \text{insert}(x) / \text{delete}(x) / \text{successor}(x) \)
  - \( \text{rank}(x) \): find \( x \)'s index in sorted order (= \# elements < \( x \) if all distinct)
  - \( \text{select}(i) \): find element of rank \( i \)

- **Idea:** use easy tree augmentation to store subtree size: \( f(\text{subtree}) = \# \text{nodes in it} \)
- Say, AVL trees \( \Rightarrow \) binary (2-3 trees also work)
- \( \text{rank}(x) \):
  - \( \text{rank} = x.\text{left}.\text{size} + 1 \)*
  - walk up to root from \( x \)
    - when go left (\( x \rightarrow x' \)):
      \( \text{rank} += x'.\text{left}.\text{size} + 1 \)
- \( \text{select}(i) \):
  - \( x = \text{root} \)
  - \( \text{rank} = x.\text{left}.\text{size} + 1 \)*
    - if \( i = \text{rank} \): return \( x \)
    - if \( i < \text{rank} \): \( x = x.\text{left} \)
    - if \( i > \text{rank} \): \( x = x.\text{right} \)
      \( i -= \text{rank} \)
  - repeat

- e.g., can't maintain rank of each node: \( \text{insert}(-\infty) \) would change all ranks
Finger search trees: [Brown & Tarjan 1980]
- goal: if already found y, search(x from y) should only take \(O(\lg |\text{rank}(x) - \text{rank}(y)|)\)
- idea: level-linked 2-3 trees
  - each node points to next & previous on same level
- maintain during split/merge:

- also maintain min & max of each subtree (via easy tree augmentation)
- search(x from y):
  - if y is in a leaf: \(v = y\)'s node
  - else: \(v = y\)'s previous child, max
  - if \(v\.min\.key \leq x \leq v\.max\.key\):
    search within v's subtree (as in regular 2-3 tree search)
  - if \(x < v\.key\): \(v = v\.level\.left\)
  - else if \(x > v\.key\): \(v = v\.level\.right\)
  - \(v = v\.parent\)
  - repeat
Analysis:
- start at leaf level (height $\emptyset$)
- each round, go up 1 level
  $\Rightarrow$ at step $i$, level link (height $i$) skips $\approx c_i$ nodes where $c \in [2,3]$
  $\Rightarrow$ if $|\text{rank}(x) - \text{rank}(y.\text{key})| = k$
  then reach $x$ in $O(lg k)$ steps
  (and downward search also $O(lg k)$)
Orthogonal range searching: preprocess $n$ points in $d$ dimensions into a (static) data structure supporting range query: find $k$ points in given axis-aligned box (rectangle in 2D)

OR count # points

2D:

3D:

1D: query = interval

- sorted array: binary search, walk right
  $\Rightarrow O(lg n + k)$ to report $k$
  (count in $O(lg n)$ via 2 binary searches + subtract)

- finger search tree: (dynamic)
  search, finger search right by 1, ...
  $\Rightarrow O(lg n + k)$ also
  (counting harder...)
1D range tree:
- complete BST (static ~ for dynamic, use AVL)
- range-query([a, b]):
  - search(a)
  - search(b)
  - trim common prefix
  - return $O(\log n)$ nodes & subtrees "in between"
- $O(\log n)$ to implicitly represent answer
- $O(\log n + k)$ to traverse $k$ outputs
- $O(\log n)$ count via subtree size augmentation
2D range tree:
- primary 1D range tree
  keyed on \( x \) coordinate
  storing all points

- every node \( v \) in primary \( x \)-tree stores secondary 1D range tree,
  keyed on \( y \) coordinate, storing all points in \( v \)'s subtree

- range-search:
  - use primary \( x \)-tree to find points
    in correct \( x \) range (implicitly)
    - \( O(lg n) \) points: check manually
    - \( O(lg n) \) subtrees: for each \( v \),
      use \( v \)'s secondary \( y \)-tree
        to find points in correct \( y \) range
        (implicitly)
  \( \Rightarrow \) implicit representation as \( O(lg^2 n) \)
  nodes & subtrees of secondary trees)

  \( \Rightarrow \) \( O(lg^2 n + k) \) to report \( k \) answers
  - \( O(lg^2 n) \) to count via subtree size
Space: \(O(n \log n)\)
- \(O(n)\) for primary tree
- each point appears in \(O(\log n)\) secondary trees (one per ancestor)

OR: each level of primary tree stores all points in secondary trees

\(d\)-\(D\) range trees:
- recurse from primary \(\rightarrow\) secondary \(\rightarrow\)...

- query: \(O(\log^d n + k)\)
- space: \(O(n \log^{d-1} n)\)

Chazelle's improvement:
- \(O(\log^{d-1} n + k)\)
- \(O(n \left(\frac{\log n}{\log \log n}\right)^{d-1})\)
(see 6.851)