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 \( x \)’s children & \( x.f \)
- 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:
  - `insert(x)` / `delete(x)` / `successor(x)`
  - `rank(x)`: find x's index in sorted order (= # elements < x if all distinct)
  - `select(i)`: find element of rank i

- Idea: use easy tree augmentation to store subtree size: \( f(\text{subtree}) = \# \text{nodes in it} \)
  \[ \Rightarrow x.\text{size} = 1 + \sum (c.\text{size} \text{ for } c \text{ in } x.\text{children}) \]

- Say, AVL trees \( \Rightarrow \) binary (2-3 trees also work)

- `rank(x)`:  
  - `rank = x.\text{left. size} + 1^*`
  - Walk up to root from x
    - When go left \((x \to x')\):  
      \[ \text{rank } += x'.\text{left. size} + 1 \]

- `select(i)`:  
  - \( x = \text{root} \)
  - `rank = x.\text{left. 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: `insert(-\infty)` would change all ranks
Finger search trees: [Brown & Tarjan 1980]
- goal: if already found y, search(x from y) should only take \(O(\log |\text{rank}(x) - \text{rank}(y)|)\)
- idea: level-linked 2-3 trees
  - each node points to next & previous on same level
- maintain during split/merge:
- store all keys in the leaves:
  \[
  2 \ 5 \ 7 \ \Rightarrow \ 2 \ 5 \ 7
  \]
  - nonleaf nodes don't store keys
  - maintain min & max of each subtree (via easy tree augmentation)
  \Rightarrow can still do (top-down) search(x):
  - say at vertex v with children \(c_1, c_2, c_3\)
  - look at min & max of each child \(c_i\)
  - if \(c_i, \text{min} \leq x \leq c_i, \text{max}\): go down to \(c_i\)
  - if \(c_i, \text{max} < x < c_{i+1}, \text{min}\):
    \begin{align*}
    &\text{return } c_i, \text{max} \text{ (predecessor)} \\
    &\text{or } c_{i+1}, \text{min} \text{ (successor)}
    \end{align*}
- **search**(x from y):
  - v = leaf containing y  
    (given)
  - if v.min ≤ x ≤ v.max:
    do top-down search for x from v
    (i.e. within rooted subtree at v)
  - if x < v.min:  v = v.level.left
  - else if x > v.max:  v = v.level.right
  - v = v.parent
  - repeat

**Analysis:**
- start at leaf level (height 0)
- each round, go up 1 level

⇒ at step i, level link (height i) skips
  ≈ ci keys/ranks, where c ∈ [2, 3]

⇒ if |rank(x) - rank(y)| = k
  then reach x in Θ(lg k) steps
  (and downward search also Θ(lg k))
Orthogonal range searching:
preprocess $n$ points in $d$ dimensions into a (static) data structure supporting range query:
find points in given axis-aligned box (rectangle in 2D)
OR count # points

2D:

\[
\begin{array}{c}
\text{query = interval} \\
\text{sorted array: binary search, walk right } \\
\Rightarrow O(lg \ n + k) \text{ to report } k \\
\text{ (count in } O(lg \ n) \text{ via 2 binary searches + subtract)}
\end{array}
\]

- 1D: 

- finger search tree: (dynamic) search, finger search right by 1, ... 
  \Rightarrow O(lg \ n + k) \text{ 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 \(O(\lg n)\) nodes & subtrees "in between"
- \(O(\lg n)\) to implicitly represent answer
- \(O(\lg n + k)\) to traverse \(k\) outputs
- \(O(\lg n)\) 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)
\[ \Rightarrow \text{implicit representation as } O(lg^2 n) \]
  nodes & subtrees (of secondary trees)
\[ \Rightarrow O(lg^2 n + k) \text{ to report } k \text{ answers} \]
- O(lg^2 n) to count via subtree size
Space: $O(n \lg n)$
- $O(n)$ for primary tree
- each point appears in $O(\lg n)$ 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: $O(\lg^d n + k)$
- space: $O(n \lg^{d-1} n)$

Chazelle's improvement:
- query: $O(\lg^{d-1} n + k)$
- space: $O(\frac{n(\frac{\lg n}{\lg \lg n})^{d-1}}{\lg \lg n})$
  (see 6.851)