Skip List

- Simple data structure, easier to implement than red/black trees.
- Sorted linked list with "skip" pointers.
- Construction:
  - Every element in the "bottom list" (list 0)
  - Element passed to list i with probability \(1/2^i\).
  - In general, element in list \(i\) is passed to list \(i+1\) with prob. \(1/2\).

Skip Lists

- Analysis:
  - Expected \(O(\log n)\) lists.
  - Expected total number of nodes is \(n \cdot n/2 \cdot n/4 \cdot \ldots \cdot n/2^m = O(n)\).
  - Search: find range in last list, examine this range next one down, etc.
  - Each range is expected to be constant length:
    \(O(1)\) work per range, \(O(\log n)\) total.
- Works only if deletions are not malicious.

Extending Red-Black Trees

- Dynamic order statistics:
  - \(\text{Select}(T, i)\) - i-th smallest in tree rooted at \(x\).
  - \(\text{Rank}(T, x)\) - rank of \(x\) in tree \(T\).
- New field: Size of subtree
- Idea: sufficient in order to know whether to go right or left.
- Ex: Rank: go down, each right - add size of left subtree +1.
  When elements found, add i - left subtree.
  For \(h\) in the figure:
  \(\text{size}(A) + 1 + \text{size}(D) + 1 + 1 + 0 = 5\).

Dynamic selection continued

- In order to be able to claim \(O(\log n)\) time, need to be able to update extra fields during:
  - Insertion/deletion into red-black tree
  - Rotations.
- Easy to see that this is trivial in this case: only local data needed.
- Example:

Interval intersection

- Goal:
  - Maintain collection of intervals (support deletion/insertion)
  - Given an interval \(I\), produce interval \(J\) from data structure, such that \(I\) intersects \(J\).
- Ignore open vs closed, each interval starts at a new point.

Interval data structure

- Interval - Node
  - Low endpoint - key
  - New info: max endpoint in subtree
- Easy to maintain during insertion/deletion/rotation
Using the data structure

- `rotate(T)`
  while `x ≠ NIL & I ∩ int(x) ≠ ∅`
  if `left(x) ≠ NIL & max(left(x)) > low(I)`
  then `x = left(x)`
  else `x = right(x)`

- Theorem: if overlap exists, and search goes left, then there exists an overlap on left. Same for "goes right".

- Proof:
  Case 1: goes right.
    Case 1a: overlap in right subtree - done.
    Case 1b: no overlap in right subtree.
      we went right -- `left(x) = NIL` or `max(left(x)) < low(I)`
      no overlap -- exists interval `I'` s.t. `low(I') > high(I)`
      but tree ordered by low -- all intervals in right tree have lows above `high(I)`?
      no overlap there.

Case 2:

- Case 2: we go left.
  Case 2a: overlap on left - done.
  Case 2b: no overlap on left.
    we went left -- `left(x) ≠ NIL`, `max(left(x)) ≥ low(I)`
    no overlap -- exists interval `I'` s.t. `low(I') > high(I)`
    but tree ordered by low -- all intervals in right tree have lows above `high(I)`.
    no overlap there.

Case 2:

- Case 2: we go left.
  Case 2a: overlap on left - done.
  Case 2b: no overlap on left.
    we went left -- `left(x) ≠ NIL`, `max(left(x)) ≥ low(I)`
    no overlap -- exists interval `I'` s.t. `low(I') > high(I)`
    but tree ordered by low -- all intervals in right tree have lows above `high(I)`.
    no overlap there.