Binary Search Trees

Binary tree in which, for any node $x$:
- Keys in the left subtree are $< x$’s key
- Keys in the right subtree are $> x$’s key

Operations:
- Min
- Max
- Predecessor($x$)
- Successor($x$)
- Insert($x$)
- Delete($x$)
- Search(key)

If the height is $h$, all operations can run in $O(h)$
- Worst-case height?
- Best-case height?
- How to guarantee best case?
- Average-case? Expected $O(\log n)$ for a random sequence of Inserts

Balanced Search Trees

Guarantee height is $O(\log n)$
- So that each operation is $O(\log n)$

Many different possibilities
- Red-Black trees, B-Trees (covered in CLRS)
- AVL trees
  - height of subtrees at a node differs by at most 1
- 2-3 trees
  - Nodes can have 2 or 3 children; all leaves at the same level
- B-Trees
  - Generalization of 2-3 trees
- Splay trees
  - Move inserted/searched nodes to the top of the tree, double-rotating along the way. Amortized cost $O(\log n)$ per operation
- Binomial heap
  - Forest of trees with $2^i$ nodes. Merging/priority queue.
- Fibonacci heaps
  - Merging/priority queue. Amortized $O(1)$ for Insert/Min/Union/DecreaseKey. Amortized $O(\log n)$ for Delete/DeleteMin
Red-Black Trees

Binary search tree guaranteed to be approximately balanced

- The maximum leaf depth is $\leq 2$ minimum leaf depth
- Every node is colored red or black
- Root is black
- Leaf nodes are NIL (and black) (we won’t show these nodes)
- If a node is red, its children are black
- Paths from any node to its leaf descendants have the same number of black nodes (black-height)

Height of Red-Black Tree is $O(\log n)$

Black subtree: ignoring red nodes

- Ignore them by having parents of red nodes adopt their grandchildren
- Each black node can have 2, 3, or 4 children
- Every leaf has the same depth, k
- # black nodes $\geq 2^{k-1}$

Now, don’t ignore red nodes

- # black and red nodes $= n \geq 2^{k-1}$
  - $\lg n \geq \lg(2^{k-1} - 1) \geq \lg(2^k - 1)$; so, $k < \lg n$
  - Longest path $\leq 2k$ (longest by alternating red & black)
  - Longest path $\leq 2\lg n$

Inserting into a Red-Black Tree

Insert as new red leaf in correct spot

- May create red/red parent/child conflict
- Work up the tree, either removing the conflict, or raising it higher in the tree
  - Max number nodes processed $= O(\log n)$
- If the root is red, make it black

Working up the tree

Case 1: Uncle is red
Recolor three nodes; keep working

Case 2: red child is bent
Rotate red child with parent
Continue to case 3

Case 3: Uncle black red child straight
Rotate and recolor parent and grandparent. Done!

Try an example

- Insert 1, .., 8 into an empty Red-Black tree
- Work in pairs
- Save results at the end of inserting each key
Red-Black Trees

Other operations
- Search
  - Same as any other BST
- Successor
  - Like any other BST: if node has right subtree, find its MIN
  - Else, return closest biggest ancestor
- Delete
  - Complicated re-balancing: still $O(\log n)$, though

Augmenting Search Trees
Chapter 14

Augmenting Search Trees

Basic idea:
- Want to maintain extra information about data
- Imagine we can calculate the data for a node based only on that node and its children
  - In a parse tree, such data is called a synthesized attribute, passing data up the parse tree
  - We can maintain the data without affecting run-time (other than by a constant factor).
    - Only information we need to update is from the modified node up the path to the root
- Examples
  - Min/Max node in sub-tree
  - Size of sub-tree
  - Depth of sub-tree
- Counter-examples
  - Order statistic (rank)
  - Median
  - Average?

Augmenting RB-Tree to calculate Rank in $O(\log n)$

Augment RB-Tree with size of $x$ (# nodes in subtree rooted at $x$)
- Clearly, this is a synthesized attribute (can be calculated at any node based only on left and right child)
- To calculate rank of $x$, add:
  1 + nodes in left-subtree
  + # of left ancestors + # nodes in their left subtrees (retrieve from left ancestors)
Interval Trees

Interval \([a, b] = \{ x \in \mathbb{R} \mid a \leq x \leq b \}\)
- Closed interval (we could define with open if desired)
- Interval tree is search tree with:
  - Key = left-hand endpoint of interval
  - Additional data = right-hand endpoint of interval
  - Synthesized attribute = maximum endpoint of subtree

Search(interval)
- Given an interval \(i\), return an overlapping interval (\(i\) overlaps \(j\) if \(\exists p\in i\) and \(p\in j\)) in the tree

Recursive Algorithm
- Search(root, \([x,y]\))
  - if root = nil then return nil
  - if \([x, y]\) overlaps \([\text{root}_l, \text{root}_r]\) then return root
  - if \(y < \text{root}_l\) then
    - entirely left of root: can't be in right sub-tree
    - return Search(root\_left-child, \([x, y]\))
  - assert(\(x > \text{root}_r\))
    - does overlap in left sub-tree
    - doesn't overlap; not less than
  - if \(x > \text{left-child(root)}_{\text{max-right}}\) then
    - can't be in left sub-tree
    - return Search(right-child(root), \([x, y]\))
  - else /* \(x > \text{left-child(root)}_{\text{max-right}}\) does overlap in left sub-tree */
    - return Search(left-child(root), \([x, y]\))