CIS 313: Intermediate Data Structure

seventh week of the term
expected behavior

• if list a is chosen randomly from among all n! permutations
• how long does “for \( i=1 \) to \( n \) T.insert(a_i)” take?
• worst case: \( O(n^2) \)
• want to argue: on average \( O(n \ l g \ n) \)

• main fact: expected search time \( (1+1/n) \) in BST built from randomly chosen permutation is \( 2 \cdot \ln(n + 1) + O(1) \approx 1.38 \log_2 n + O(1) \)
observations

• this does not bound the height of the tree

• exercise 12.4-2, p 303: describe a binary search tree on n nodes such that the average depth of a node in the tree is \( \Theta(\lg n) \) but the height of the tree is \( \omega(\lg n) \)

• stronger result: height of randomly built BST is is \( \Theta(\lg n) \)

• new goal: maintain BST whose height is is \( \Theta(\lg n) \) in the worst case

• self balancing search trees: AVL, red-black, B-trees
balanced tree

• not realistic to expect perfectly balanced tree
• one attempt (not common): *weight-balance*, where the number of nodes in left and right subtrees of any node must be close to each other
• better: *height-balance*, the height of the left and right subtrees must be close
• AVL: differ by one
• red-black: differ by factor of two
• balance maintained by rotations
rotation: single
rotations: double

Composed from two single rotations.
AVL trees

- (not in text)
- named after inventors Adelson-Velskii and Landis
- store at each node the balance factor:
  - \( \text{bf}(p) = \text{height}(p.\text{lchild}) - \text{height}(p.\text{rchild}) \)
  - requirement: for every node \( p \), \( \text{bf}(p) \) equals -1, 0, or 1
- requires two bits extra storage at each node
AVL height is $O(\log n)$

- let $G_k$ be an AVL tree (shape) of height $k$ with the fewest number of nodes
- $G_k$ can be constructed inductively as a node with a $G_{k-1}$ left child and a $G_{k-2}$ right child
- define $g_k$ to be the number of nodes in a $G_k$ tree
  - $g_0 = 1$, $g_1 = 2$, $g_k = 1 + g_{k-1} + g_{k-2}$
  - sequence: 1, 2, 4, 7, 12, 20
- fact: $g_k = F_{k+3} - 1$ ("easy" to prove with induction)
trees $G_k$ and values $g_k$
AVL tree height: the punchline

• if \( n \) is the number of nodes in an AVL tree of height \( H \) then
  \[
  n \geq g_H = F_{H+3} - 1
  \]

• we know \( F_k = \left\lfloor \phi^k / \sqrt{5} \right\rfloor \), where \( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.618 \)

• \( \lg F_{H+3} \geq \log \frac{\phi^{H+3}}{\sqrt{5}} - 1 = (H + 3) \log \phi - \log \sqrt{5} - 1 \geq (H + 3) \log \phi - 4 \)

• so \( (H + 3) \log \phi - 4 \leq \log F_{H+3} \leq \log(n + 1) \) (take log of both sides of top line)

• moving terms around: \( H \leq \frac{\log(n+1)+4}{\log \phi} - 3 \approx 1.44 \log(n + 1) + O(1) \)
AVL insertion

- insert node as with a BST (add it to a null pointer)
- update balance factors along path from new node to root
- the balance factors of some nodes may in violation: 2 or -2
- find the critical node: the lowest out of balance node
- perform the appropriate rotation

- note: this will affect the balance factors of nodes above it
- total insertion time $O(\lg n)$
AVL insertion

Four Possible Cases

\[\text{bf}(x) = +2 \text{ and } \text{bf}(x.\text{left}) = 1\]
rightRotate(x)
\[\text{bf}(x) = +2 \text{ and } \text{bf}(x.\text{left}) = -1\]
leftRotate(x.left)
rightRotate(x)
\[\text{bf}(x) = -2 \text{ and } \text{bf}(x.\text{right}) = -1\]
leftRotate(x)
\[\text{bf}(x) = -2 \text{ and } \text{bf}(x.\text{right}) = 1\]
rightRotate(x.right)
leftRotate(x)

Pictures from Wikipedia
2-3 and 2-3-4 trees

• quick intro here, we will return to them later as B-trees
• a 2-3 tree is a B-tree of order 3 (see ex 18-2, p 503, of text)
• these use multi-way search nodes
• must be perfectly balanced: all paths from the root to a null node have the same length
• insertions cause splits rather than rotations

• important: red-black trees (our real focus) are a binary implementation of 2-3-4 trees
multiway search nodes

4

elements < 4

elements > 4

4

elements < 4

elements > 4 and < 10

elements > 10

4 10

4 10 20
example
insertion: splitting nodes

• can split a node when it is full or has overflowed
• splitting on insertion can be bottom-up
  • put node at bottom of tree, if over-flow, split on the way up
• or top-down
  • when looking for insertion point, if full node seen, split it
• most B-tree implementations use bottom up (less space)
splitting a full node

1. Insert 10 into the root.
2. Split the root into two nodes, 4 and 20.
3. Insert 10 into the parent node.
red-black trees and 2-3-4 trees

• a 2-3-4 tree node would need up to 4 child pointers
• frequently unused so waste of space
• red-black tree is binary tree implementation of 2-3-4 tree
• uses rotations to handle the splits
• need one bit to indicate color
  • descending the tree, black means ”new node”
  • red means “belong to parent”

• Java uses RB trees in the TreeMap class
  (https://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html)
2-3-4 nodes as RB nodes (2- and 3-nodes)
2-3-4 nodes as RB nodes (4-nodes)
example RB tree
viewed as 2-3-4 tree
red-black tree rules

1. every node is either red or black
2. the root is black
3. every leaf (null) is black
4. if a node is red, both of its children are black
5. for each node, all simple paths from the node to descendant leaves contain the same number of black nodes