Your task

- Find an algorithm to balance an unbalanced tree, using your playing cards
- Balanced = difference between levels is at most 1
- Record your algorithm
- Record the number of operations it takes for your algorithm to balance the tree

Balancing trees

- 2 methods:
  - balancing globally
  - balancing locally
- We’ll look at several algorithms of each type

Binary search-based balance

- Store all tree elements into an array
  - as they arrive, or via inorder traversal
- Find midpoint of array
  - this becomes the root of the tree
- While there are still elements in the array to place,
  - halve the array
  - construct leftmost subtree using first half
  - construct rightmost subtree using second half
- This algorithm uses recursion
Pros/cons

- **Pros:**
  - easy to understand
  - tree will be perfectly balanced
- **Cons:**
  - must destroy and then rebuild the entire tree
  - requires an array for storage, in addition to the data structure for the tree

DSW algorithm

- Colin Day, Quentin F. Stout, Bette L. Warren
- **Idea:** rotate nodes until the tree is balanced
  - rotate left or right
- **2 step process:**
  - Step 1: Transform tree into a backbone (looks like a linked list)
  - Step 2: Transform backbone into a tree by rotating every other node
- End result is a perfectly balanced tree

DSW algorithm: clockwise (CW) rotation

- **Input:** node Q, its parent P, and its left child R
- if parent is not the root of the tree,
  - R becomes the child of P
- right subtree of R becomes left subtree of Q
- R becomes the parent of Q

Note: counterclockwise (CCW) rotation is symmetric to this

DSW algorithm: step 1

- Set a temporary pointer \( temp \) to the root of the tree
- While we haven’t reached a leaf node,
  - if \( temp \) has a left child,
    - rotate left child CW around \( temp \) --> left child is now parent of \( temp \)
    - point \( temp \) at this new parent node
  - else, point \( temp \) at the right child
DSW algorithm: step 2

- $n =$ number of nodes in the tree
- Set $m = 2^{\text{floor}(\log(n+1))} - 1$;
- Rotate $n-m$ nodes starting from the root
  - rotation done on every other node
- while $m > 1$
  - divide $m$ by 2 (INTEGER DIVISION!)
  - rotate $m$ nodes starting from the root

Complexity of DSW

- First pass:
  - best case: one pass through while loop
    - traverse $n$ nodes, with no rotations
  - worst case: root has no right child
    - traverse $2n-1$ nodes, with $n-1$ rotations
  - both are $O(n)$

- Second pass:
  - loop iterates $m - \log(m+1)$ times
  - number of rotations is $n - m + (m - \log(m+1)) = n - \text{floor}(\log(n+1))$
  - $O(n)$
- Complexity for the entire process: $O(n) + O(n) = O(n)$
  - good running time
  - no additional storage needed

AVL tree

- Adel'son-Vel'skii and Landis
- Definition: a tree in which the height of left and right subtrees differ by at most one
- Nodes contain balance factors
  - balance factor = height of right subtree – height of left subtree
- Balancing an AVL tree is done **locally**
  - does not guarantee that tree will be perfectly balanced
Searching an AVL tree

- Height of a tree is bounded
  - lower bound is \( \lg(n+1) \)
  - upper bound is \( 1.44 \lg(n+2) - 0.328 \)
- Worst case search time is \( O(lg\ n) \)
- Perfectly balanced binary trees are better for searching (in terms of complexity), but AVL trees are still very good

Balancing an AVL tree

Rule of thumb:
“rotate towards the smaller subtree”

Balancing an AVL tree: insertion

- Must be done whenever \( \text{abs(balance factor)} > 1 \)
- Four possible cases for rebalancing:
  1. insert a node into the right subtree of the right child
  2. insert a node into the left subtree of the right child
  3. insert a node into the left subtree of the left child
  4. insert a node into the right subtree of the left child
- Note: the height of the tree will be preserved when we are finished!
  means we don't have to rebalance anything above this

Insertion: case 1

- Recalculate balance factors
- Rotate unbalanced node's right child about the unbalanced node (CCW)
- Move left subtree of child to right subtree of former unbalanced node (if necessary)
**Insertion: case 2**

- Recalculate balance factors
- Rotate left child of right child node about the right child (CW)
- Rotate left child of unbalanced node about the unbalanced node (CCW)
- Rearrange left and right subtrees of rotated nodes (if necessary)

**Balancing an AVL tree: deletion**

- Not as simple as insertion
  - may need to rebalance nodes above the original unbalanced node!
- Use deletion by copying, then apply rotations
- Move upward from the leaves to the roots when rebalancing
- Worst case: need to rebalance every node on the path from the deleted node to the root $\rightarrow O(\log n)$

**Deletion: 4 cases**

Let P be the root of the subtree of interest, Q be the right child of P, and R be the left child of Q.
Assume node is deleted from left subtree of P.

Case 1: Q has balance factor of 1 before deletion
Case 2: Q has balance factor of 0 before deletion
Case 3: Q has balance factor of -1 and R has balance factor of -1 before deletion
Case 4: Q has balance factor of -1 and R has balance factor of 1 before deletion

**Algorithms for Cases 1-4**

- Cases 1 and 2: rotate Q around P (CCW)
  - left subtree of Q becomes right subtree of P
- Cases 3 and 4: 2 rotations
  - rotate R around Q (CW)
  - rotate R about P (CCW)
  - right subtree of R becomes left subtree of Q
  - left subtree of R becomes right subtree of P
Complexity of insertion and deletion

- At most $1.44 \lg(n+2)$ searches
- Insertion: at most 2 rotations
- Deletion: at most $1.44 \lg(n+2)$ rotations
- Deletion is more costly, but require rebalancing less often than insertion

Self-adjusting trees

- Idea: move frequently accessed nodes up towards the root of the tree
- Q: What are some possible applications for this?
- Implementation: 2 schemes
  - order nodes by # times accessed
    - each node needs a counter
  - move a node towards the root each time it is accessed
    - no counter necessary

Self-restructuring trees

- Single rotation: rotate a child about its parent if the child's key is accessed
  - unless it is the root
- Move to root: Repeat single rotation until the child is the root
- Complexity of moving a node is approximately $2 \ln 2 \lg n$

Splaying

- Modification of “move-to-root” strategy
- Not a balanced tree!
  - goal is to move accessed element to the top
- Idea: apply single rotations (in pairs) to {child, parent, grandparent} nodes until the accessed node is the root of the tree
Splaying: 3 cases

- Let R = accessed node, Q = its parent, P = its grandparent
- Case 1: Q is the root
  - singular splay: rotate R about Q
- Case 2: Homogeneous config
  - R and Q are both left children (or right children)
  - homogeneous splay: rotate Q about P, then R about Q

Complexity of splaying

- Access one node: $O(lg n)$
  - same as worst case in balanced tree
- Sequence of $m$ node accesses: $O(m lg n)$

Splaying: 3 cases (cont.)

- Case 3: Heterogeneous config
  - R is right child of Q, and Q is left child of P, OR
  - R is left child of Q, and Q is right child of P
  - heterogeneous splay: rotate R about Q and then about P

Semisplaying

- Idea: do first splay rotation; second splay rotation involves the parent nodes
- Accessed node is near the root rather than at the root
- Advantage: resulting tree is more balanced, and remains (somewhat) balanced
Midterm evaluation

- What 1-2 things are going well for your learning in the course so far (please be as specific as you can)?
- What 1-2 things are not going well for your learning?
- What can I, the teacher, do differently, and what can you, the students do differently to improve the second half of this course?
- On average, how much time per week do you spend on the homework assignments?
- Other comments?