Splay Trees.

1. Invented by Daniel Sleator and Robert Tarjan.
   Self-adjusting Binary Search Trees.
   JACM, pp. 652-686, July 1985.

2. Balanced search trees (AVL, weight balanced, B-Trees etc) have O(log n) worst case search, but are either complicated to implement, or require extra space for balance, or both.
   Splay trees don't maintain any explicit balance condition; rather they apply a simple restructuring heuristic, called splaying EVERY time the tree is accessed.

3. Intuition: for search cost to be small in a binary search tree, the item to be accessed must be close to the root. Many tree-restructuring rules try to move the accessed item closer to the root. Two classical heuristics are:

   3a. Single rotation: after accessing item i at a node x, rotate the edge joining x to its parent; unless x is the root.

   3b. Move to Root: after accessing i at a node x, rotate edges joining x and p(x), and repeat until x becomes the root.

   Unfortunately, it can be shown that neither heuristic guarantees the O(log n) search cost: There are arbitrarily long access sequences where time per access is O(n).

4. Sleator-Tarjan's heuristic is similar to move-to-front, but its swaps depend on the structure of the tree.

   To splay at a node x, repeat the following step until x becomes the root:

   Case 1 (zig): [terminating single rotation]
   if p(x) is the root, rotate the edge between x and p(x); and terminate.

Fig.

Case 2 (zig-zig): [two single rotations]
if p(x) is not the root and x and p(x) are both left or both right children, first rotate the edge joining p(x) with its grandparent g(x), and then rotate the edge joining x with p(x).

Fig.

Case 3 (zig-zag): [double rotation]
if p(x) is not the root and x is a left child and p(x) is a right child, or vice versa, then rotate the edge joining x with p(x) and then rotate the edge joining x its new p(x).

Fig.

5.

Fact 1. Splaying at a node x of depth d takes O(d) time.

Fact 2. Splaying moves x to the root, and roughly halves the depth of every node on the access path.

6. UPDATE OPERATIONS.

access (i,t): if item i is in tree t, return a pointer to its location; otherwise, return a null pointer.

insert (i,t): insert item i in tree t, assuming it is not there already;

delete (i,t): delete i from t, assuming it is present.

join (t1, t2): This operation assumes that all items in t1 are less than all those in t2. This combines trees t1 and t2 into

a single tree and returns the resulting tree.

split (i,t): Construct and return two trees: t1, which contains all items less than or equal to i, and t2, which contains all items greater than i. This operation destroys t.

7. How to perform the operations.

a. To perform access(i,t), we search down from the root, looking for i. If search reaches a node x containing i, WE SPLAY AT X and return the pointer to x.

If search reaches a null node, we SPLAY the last non-null node, and return a null pointer.

Fig. 6.

b. Because of the splaying's effect of moving x to the root, insert and delete are easily implemented using join and split.

c. To perform join(t1, t2), we first access the largest item in t1. Suppose this item is i. After the access, i is at the root of t1. Because i is the largest item in t1, the root must have a null right child. Simply make t2's root to be the right child of t1. Return the resulting tree.

d. To perform split(i,t), first perform access(i,t). If root contains an item greater than i, then break the left child link from the root, and return the two subtrees. Otherwise, break the right child link from the root, and return the two subtrees.

[In both join and split, take extra care if one of the subtrees is empty.]

e. To do insert(i,t), perform split(i,t). Replace t with a new tree consisting of a new root node containing i, whose left and right subtrees are t1 and t2 returned by the split.

f. To do delete(i,t), perform access(i,t), and then replace t by the join of its left and right subtrees.

Figs.

8. Theorems.

A sequence of m arbitrary splay tree operations takes O(m + \sum_{j=1}^m log n_j) time; where n_j is the size of the tree at operation j.

Thus, the amortized cost per operation is O(log n).