1. Motivation

(a) Goal: do simple for-loops in $O(\log n)$ time.

(b) Sanity check: array of length $n$, add $k$ to all entries in $[i \ldots j]$, find sum of $[i \ldots j]$. Static balanced tree works.

(c) Sanity check: dynamic connectivity on a line (add/remove edges).

(d) Reduces to predecessor / successor.

(e) Lines lead to implicit ordering. No keys, but well defined left/right.

(f) General purpose tool: splay trees.
   (i) Almost the same as usual binary search trees.
   (ii) Move searched node to root using sequence of ‘rotates’, which are local adjustments.
   (iii) Simplified form of splay theorem: sequence of $n$ operations take $O(n \log n)$ time.

(g) Augmentation: can store information about each subtree at its root, as long as such info can be computed in $O(1)$ time from info about children.

(h) E.g. size, sum, minimum, maximum, subinterval of maximum sum.

(i) Any balanced binary search trees (BBSTs) work, but these are simplest.

2. Dynamic Connectivity of Lines

(a) Graph is a collection of lines, but no prior embedding given.

(b) Maintain ordering of each line in a BBST.

(c) Removal of edge: two orderings.

(d) Adding edge, segments $v_1 \ldots v_k, u_1 \ldots u_l$, adding $v_k u_1$.

(e) Easier case: $v_1 < v_2 < \ldots < v_k, u_1 < u_2 < \ldots < u_k$. Join two trees.

(f) Harder case: $v_1 < v_2 < \ldots < v_k, u_1 > u_2 > \ldots > u_k$.

(g) Need to ‘flip’ $u_1 > u_2 > \ldots > u_k$.

(h) Introduce ‘flip’ bit on each BBST node, indicating subtree is flipped.

(i) Change order of left/right when such bits encountered in accessing a node.

(j) Alternatively, can push such bits downwards in the tree, ensuring the part that we access doesn’t have flips.

3. Dynamic Problems on Static Trees

(a) Change weight of node, find weight of an entire subtree.

(b) Preorder traversal of tree.

(c) Each subtree corresponds to a contiguous section of traversal, line problem again.

(d) Change weight of node, find weight of a path.

(e) Heavy-light decomposition: decompose tree into paths.
(f) Any path encounters at most $O(\log n)$ of these paths.

4. Dynamic Connectivity of Trees
   (a) Graph guaranteed to be a tree.
   (b) Restricted version: new edges adjacent to at least one root.
   (c) Maintain preorder becomes splicing in/out intervals of arrays.
   (d) Corresponds to split/join of trees.
   (e) Edges on non-root vertices: need to reroot tree.
   (f) Root-independent version instead of preorder traversal.
   (g) Eulerian tour: traverse tree, every time visiting a vertex, write it down.
   (h) Root can be changed by rotating the array, one split and one join.

5. Dynamic Connectivity of Undirected Graphs
   (a) Maintain spanning forest.
   (b) Connect two trees: concatenate their Eulerian tours (subject to rotations).
   (c) Edge deletion: need to check whether there is an edge connecting the two pieces.
   (d) Potentially $O(n)$ edges adjacent to current component.
   (e) Search on smaller piece, if no edges across exist, piece halves in size.
   (f) Vertex count of connected component decreases by factor of 2, $O(\log n)$ layers.
   (g) Need to step through all off-tree edges of a subtree in output sensitive time.
   (h) Transform into bounded degree, label nodes with off-tree edges.
   (i) Each edge checked move to a higher level.
   (j) Label edges with levels $0 \ldots \log n$.
   (k) Each component using level $l$ edges have at most $O(n/2^l)$ vertices.
   (l) $G_l$: graph containing edges of level $l$ or higher.
   (m) Maintain maximum weighted spanning forests for each $G_l$.
   (n) Search for replacement edges in decreasing order of levels.
   (o) Promoting level of edge requires checking the cycle on all previous levels.