1. Day, Stout, Warren Algorithm – **O (log n)**
   1. **Balances a COMPLETE binary search tree in memory**
   2. **Creating a backbone** by doing **right rotate** on every node that has a left child
      1. Complexity: O (n)
   3. **Balancing the backbone** by perform a series of **left rotate**
      1. Find the number of rows complete = log 2 (n)
      2. Find the expected nodes at the bottommost level = n – (2log 2 (n) – 1)
      3. Initial rotation for the bottommost level
         1. Left rotates on odd nodes starting from root
         2. Number of rotating = number of expected bottommost nodes
         3. If we expected 2 nodes on the bottommost level, the first 2 odd nodes will be rotate
      4. Now, left rotating ALL odd nodes on the backbone
         1. Number of rotating = the expected height of tree = rows completed +1;
   4. Unlike AVL trees or self-balancing binary search trees, we only want to call the DSW algorithm once, when our tree is finished and we don't expect any more insertions (or deletions).
2. AVL Tree – **O (log n)**
   1. Dynamically **self-balancing tree** – always stay balanced by adjusting itself
   2. AVL is a BST in which the balance factor of each node is 0, -1, and 1
   3. **AVL Inserting – O (log n)**
      1. Do a normal BST insert
      2. Starting from the newly inserted node, travel up to find the first unbalanced node
      3. Rebalance the tree by performing appropriate rotations
         1. Simple right rotation O (1)
         2. Simple left rotation O (1)
         3. Left-right rotation (double rotation)
         4. Right-left rotation (double rotation)
   4. **AVL** is more balanced, but required more rotations during insertion and delete compare to **Red Black Tree**
      1. Use AVL when you need searching more frequently
      2. Use Red Black Tree when you need to insert and delete lots of data
3. Heaps – **O (log n)**
   1. A **COMPLETE binary tree** that maintains heap property
      1. The value of each node is >= (max heap) or <= (min heap) the value of its children
   2. Represented as array (start index at 1)
      1. Root node is A[1]
      2. Value of node i is A[i]
      3. Parent of node i is A[i/2]
      4. Left child of node i is A[i\*2]
      5. Right child of node i is A[i\*2 +1]
   3. Level order reversal
   4. Array is filled with no hole in it
   5. Heap Operations
      1. Insertion – **O (log n)**
         1. Add the element at the bottom of the heap (at the end of the array)
         2. Compare with its parent node, swap up if needed
      2. ExtractMax or ExtractMin – **O (log n)**
         1. Remove root
         2. Move last item to root
         3. **Heapify:** Swap down with its children recursively until in correct order (larger child in max heap, and smaller child in min heap)
      3. Heapify – **O (log n)**
         1. Swap down with its children to maintain the heap property
            1. Smaller child in min heap
            2. Larger child in max heap
      4. Build Heap – **O** **(n log n)**  or **O(n) ???**
         1. Build a heap from an unsorted array
         2. Use Heapify on each node
            1. Each Heapify call takes O (log n))
            2. Call Heapify (n/2) times
      5. Heap Sort – **O (n log n)**
         1. Uses binary heap to sort an array
            1. Fill array with random number
            2. Build Heap (min – ascending order; max – descending order) from an array
            3. Extract root until heap is empty

Use Heapify to restore the heap property

1. Disjoint Sets – **O (log n)**
   1. A collection of distinct dynamic sets
   2. Each set has a representative
   3. Disjoint Set Operations
      1. MAKE-SET(x) – create a tree containing x – O (1)
      2. FIND-SET (x) – return the representative of the set contain x - O(n)
         1. Follow the chain of parents to root
      3. UNION (x,y) – combine the two sets containing x and y - O (1)
         1. Let the root of 1 set point to the root of the other
   4. UNION by RANK – **O (log n)**
      1. Let the root with smaller rank point to the root with larger rank
   5. Path Compression – **O (log n)**
      1. Used in FIND-SET(x)
      2. Make each node in the path from x to root point directly to the root, thus reduce the tree height
   6. Disjoint Set Forests
      1. Union – Find Algorithm that use Union by Rank and Path Compression
         1. Make-Set (x)
         2. Union (x,y) – O (log n)
            1. Find-Set(x)
            2. Find-Set(y)
            3. Link(Find-Set(x), Find-Set(y))

**Complete Tree**: a tree that is filled in every row except the last row in which it was filled from LEFT to RIGHT

**Perfect Tree:** a tree that is completely filled in every row

**Balance factor:** difference in height between the left subtree and the right subtree of each node