Search Trees.

1. Trees useful structure for hierarchical information, searching, etc.
   With large data, repeated linear search in link list too slow.
   With appropriate binary search trees, we can do searches, inserts, deletes, find successor, find predecessor etc in O(log n).

   Hashing is convenient if you just need to do inserts and lookups, but suppose you wanted to do:

   Lookup-by-prefix: type in first few letters of someone's name and have it output all names in the database that begin with those letters.

   Find-in-range[key1, key2]: output all keys between key1 and key2

   A version of lookup(x) that returns the closest key to x when x is not in the database.

   Also consider operations like:

   Output the kth smallest element; assume all keys are distinct.

   Do a version of lookup(x) that tells us the rank of x (how many keys are smaller than x).

2. A recursive definition of the tree:
   tree either empty, or
   it has a root node r, with zero or more non-empty subtrees, whose roots are children of r.

   Drawn as r at the root, and edges from r to the roots of subtrees T1, T2, .., Tk

```
                 root

            k1     k2   ....   kl

        Subtree1  Subtree2     Subtreel
```

3. No other structure: arbitrary distribution of depths, keys etc.

4. Example.

```
                          A

            B       M       L       T

                 C   F  X  Y  G
```

5. A path in tree is the sequence of nodes, or edges.
   Path length is the number of edges.
   Paths are unique in a tree.
   Depth of a node is measured as path length from the root.

Parent-child relationshio.
Ancestor-descendant relationship.

6. The Directory structure example.
the tree organizes the directory structure, making it
intuitive to traverse.

```
                      /usr

              alex          bill          charles

   research  teaching   junk

  paper books  c1 c2 c2
```

7. Tree Traversals.
   1. Listing of sub-directories ---- Pre-order traversal:
      print the node, then recursively traverse each
      subtree.

   2. Compute dir sizes --- post-order traversal:
      compute sizes of all subtrees, then add them
      to get the size of the dir.

8. Binary Trees. Each node has 0, 1, or 2 children.
Most common form of trees:
heap; expression trees; branch-and-bound algorithms.

9. Binary SEARCH Trees.
Important application of binary trees in searching and
implementing a search ADT.
For simplicity, we assume that objects are identified by
numeric keys (a totally ordered universe suffices).
Assume also all keys are distinct.

What makes a binary tree a SEARCH tree is the "ordering" property:

for every node X, the keys in the left subtree < key(X)
the keys in the right subtree > key(X).

Example.

```
      a search tree;                not a search tree;

          10                           10

       4        12                  8       15

         7          50          20          9
```

```
   5         30
```

10. Find operation.

```
if (t == null) return null
elseif (x < t->key) return find(x, t->left)
elseif (x > t->key) return find(x, t->right)
else return t;        // match
```

11. FindMax or FindMin.

```
if (t != null)
        while (t->right != null)
                t = t->right
return t;
```

12. Insert.

To insert X, mimick find(X). If X found, nothing to do.
(may be update some record).
Otherwise, insert X at the last spot on the path.

```
if (t == null)
   t = new node x; // insert x here //
elseif (x < t->key)  insert (x, t->left)
elseif (x > t->key)  insert (x, t->right)
else;                  // already exists; do nothing.
```

13. Delete/Remove.

As always, this is the most complex operation.

a. if the node X is a leaf, easily removed.
b. if node X has only one child, then just bypass it;
  make the child directly linked to X;
c. if node X has two children, general strategy is to
  replace X with the smallest node in the right subtree
  of X; and recursively delete that node;
  the second deletion turns out to be easy;
  because the second node can't have two children.

14. Examples.

I: delete 5

```
           6                     6

      2         8           2         8

   1     4               1    4

      3   5                 3
```

II. delete 4

```
            6                       6

       2          8            2          8

    1     4                 1    3

         3
```

III. delete 2; swap it with min in right subtree (3); then delete 3

```
            6                       6

     ->  2          8             3          8

    1      5                  1     5

         3                      4

           4
```

15. Average case analysis.

In the worst-case, a binary search tree on n nodes can have height n-1. Thus, every insert, delete, find can cost O(n) time, the same as the linked list.

Fortunately, there are ways to keep the tree roughly balanced so that height remains O(log n).

Given a set of keys, it's easy to build a perfect balanced search tree: use the median division rule. Later we will see how to achieve this dynamically.

If we start from an empty tree and do a series of insertions, what does the tree look like. Of course, the worst-case can be a single path: linear depth.

A simple analysis shows that if insertions are done randomly (assume keys drawn from [1,n]), then average depth is log n. More precisely, the sum of the depths of all the nodes is O(n log n).

Let D(n) be the internal path length.
If i nodes in the left, and (n-i-1) on the right, then

$D(n) = D(i) + D(n-i-1) + (n-1)$.

Under random insertions, the value of i varies uniformly between 0 and n-1, and so

$D(n) = 2/n \sum_{j=0}^{n-1} D(j) + (n-1)$

This is a famous recurrence, which solves to $D(n) = O(n \log n)$.

16. Picture of a random tree.

```
============================================================
===============
```

AVL Trees.

1. AVL named after its inventors: Adelson-Velskii and Landis.

It is a BST with a balance condition, which ensures that for every sequence of inserts and deletes, an AVL tree on n nodes has height O(log n).

2. Simply requiring that the left and right children of the root have the same height does not work. They can each have height n/2, singly branching paths.

```
        O
      O   O
    O       O
  O           O
```

Requiring that every node's children shall have exactly the same height is too restrictive; achieved only if the tree is perfectly balanced with 2^k -1 nodes.

The AVL tree tries the next best thing: for any node, the heights of left and right children can differ by at most 1.

[[That is, at each node ensure that the max path length along left and right braches differs by at most 1.
If you want to use "subtree heights" then it helps to define the height of empty tree as -1.]]

Examples.

```
           AVL                          not an AVL tree


         5                    7

     2       8          2            8
```

```
   1   4   7         1    4

     3                  3   5
```

3. What can be the most lop-sided AVL tree?
    Let S(h) be the min number of nodes in h-high AVL tree.
    $S(h) = S(h-1) + S(h-2) + 1$
    Like Fibonacci numbers.
    $h = 1.44 \log (n+2)$.

  Example of such a tree.

4. Of course, given a set, one can always build a balanced tree of
    O(log n) height.
  The problem is insertions and deletions get the tree out of balance.
  AVL trees do simple operations on the tree to regain balance.
  These operations, called ROTATIONS, involve only changing a
    couple of pointers.

5. We will only discuss insertions; deletions are similar though
    a bit more messy.
  How can an insertion violate AVL condition?
  At some node x, before the height diff between children was 1, but
    the insertion may make that difference 2.
  We will fix this violation from bottom (leaf) up.

6. Suppose A is the first node from bottom that needs rebalancing.
  Then, A must have two subtrees whose heights differ by 2.

  This imbalance must have been caused by two cases:

  (1) The new insertion occurred into
    the left subtree of the left child of A; or
    the right subtree of the right child of A.

  (2) The new insertion occurred into
    the right subtree of the left child of A; or
    the left subtree of the right child of A.

  We can see that (1) and (2) are different in that the former
    is an outside (left-left, or right-right) violation, while
    the latter is an inside (left-right) or (right-left) violation.

  We fix (1) with a SINGLE ROTATION; and (2) with a DOUBLE ROTATION.

7. SINGLE ROTATION.

    Consider the before and after pictures.
    Node k2 is in violation because its left child is 2 levels
    deeper than its right child; node k1 is *okay*.
    This is the only possible type (1) situation that can cause
    k2 to be in violation *without* k1 being in violation.

```
         k2                       k1

     k1                               k2
--                TreeZ
--           TreeY             TreeX     TreeY    TreeZ
--    TreeX
```

```
* Rotation at   k1 = 7, k2 = 8.

          5                          5

     2         8                2         7

   1    4     7               1    4     6    8

      3     6                    3
```

New TreeX must be at the same level as TreeY *before*, and the new insertion must increase its level by 1.

The ``single rotation'' simply tilts the balance to the right, making k1 the new root. Note that "key order is preserved." Now, X, Y, Z are all at the same level, so both k1 and k2 are balanced.

Since the new height of this subtree is exactly the same as the old height, we are finished: *no further rotations up the tree are needed*.

8. Example

Insert   3, 2, 1, 4, 5, 6, 7.

```
     3         2          2              2

   2   -> fix   1   3   ->    1   3   -> fix    1    4   ->

  1                            4              3   5

                                5
```

```
     2                  4                4

   1    4    -> fix      2     5     ->     2      5

      3   5           1   3     6        1   3     6
```

```
              6                                   7


                   4

    -> fix       2         6

               1    3    5    7
```

9. DOUBLE ROTATION

If however it's the Y tree that is too deep, then single rotation doesn't fix it.

```
          k2                           k1

       k1                                 k2

              TreeZ                 TreeX
   TreeX                                    TreeZ

          TreeY                          TreeY
```

10. We need to look one level deeper inside Y; say, the root of this subtree is k2, and the two subtrees are B and C. We perform a double rotation.

```
          k3                          k2

      k1                          k1      k3

          k2    D
                                      B     C
   A                           A             D
        B    C
```

[[One of B or C is at level A+1, but don't know which one. To make sure other being shallow doesn't cause problems, we make both at level below A.]]

11. Neither k1 nor k3 can be the roots; k1 as root is what single

rotation does, which doesn't work; and k3 is already root
and we have imbalance. So, the solution is to make k2 the root
and redistribute the subtrees, preserving order.

Example. To the old example, insert 16, 15, ..., 10, 8, 9.

```
       4                                    4

   2        6                           2        6

 1    3    5   7     -> fix           1    3     5     15

              16                                  7    16

             15
```

next insert 14, and look at the part of the tree rooted at 6, which has AVL violation.

```
        6 (k1)                              7

      5      15 (k3)     -> fix          6       15
                           insert 13
          7 (k2)   16                 5       14    16

               14                            13
```

-> fix by single rotation at 7:

```
             7

       4           15

    2     6     14      16         and so on...

  1   3    5    13
```