Binary Tree

- Each node has at most one left and one right child.
- If you need to search in the tree for an item, you have to look at all the items (traverse every node).
Given a pointer to the root, you can search with

- **Depth-first search**
  - root - children
  - V I H A B S
  - or V B S T I A H ...

- **Breadth-first search**
  - root - level 1 - level 2 ...
  - V I B H A S
  - or V B I S A H ...

- **Preorder**
  - root - left - right
  - V I H A B S

- **Postorder**
  - left - right - root
  - H A I S B V

- **In order**
  - left - root - right
  - H I A V B S
Search for an item in a generic binary tree

```
   7
  / \
 9   3  Worst case - O(n)
 / \  Best case - O(1) << find it right away
10  6
```

Expected case - $O(n)$
Find the min in a generic binary tree

- need to look at all elements

  best case \( O(n) \)

  worst case \( O(n) \)
Binary Search Tree

→ tree is sorted!

```
   a
  /   \
 /     \   b ≤ a
|       |
c ≥ a
```

To find an element, you have to search but you may not need to traverse entire tree.
Because it is sorted, you only have to look at one node per level in your search.

worst case is $O(\# \text{levels})$ and $\# \text{levels}$ in worst case is $O(n)$ in an unbalanced tree.

So worst case $O(n)$ even in binary search tree.

Example:

$3 < 6 \rightarrow 6 \rightarrow 0$

$3 \geq 6 \rightarrow 5 \rightarrow 2 \rightarrow 0$

$3 > 2 \rightarrow 8 \rightarrow 0$

$3 = 3! \rightarrow _0 \rightarrow 0$

$\{ \text{unbalanced tree} \}$
Best case $O(1)$ as before
(find it right away)

Average case?

Really complex proof, but
it turns out it is $O(\log n)$. 
Find Min in a Binary Search Tree
- find the left-most node.

worst case: $T(n) = n$

```
BSTMin (Tree t) {
    while (t->left) {
        t = t->left;
    }
    return t;
}
```

best case: $T(n) = 1$
Binary Search Tree: Successor
- given one element in a tree, how do you find the next largest element in the tree?
- assume distinct elements

Case I: there is a right subtree
next largest element is Min of right subtree.

next_largest(4) = 6
next_largest(8) = 9
next_largest(10) = 11
Case II: there is NO right subtree.
then next largest element is
lowest ancestor
whose left child is also an ancestor or self

```
        8
       / \
      4   10
     / \  /  \
    2  6  9  11

next_largest(2) = 4
next_largest(9) = 10
next_largest(6) = 8
next_largest(11) = nil
```

Runtime: $O(\# \text{ levels})$
because we either go down the tree or up the tree.
Binary Search Tree :: Insert

```java
assume tree is non-null.

Insert(Tree t, Node n) {
    NodeI'mAt = t.root;
    while (NodeI'mAt != null) {
        possibleParent = NodeI'mAt;
        if (n.value < NodeI'mAt.value) {
            // go left
            NodeI'mAt = NodeI'mAt.left;
        } else {
            // go right
            NodeI'mAt = NodeI'mAt.right;
        }
    }
    add n as left or right child of possibleParent
}
```
Binary Search Tree :: Delete

Case 1: Node to delete has no children
- just remove it

DELETE (5, 3, 10) = (2, 3, 10)
Binary Search Tree: Delete

Case 2: node to delete has 1 child
- replace deleted node with child
Binary Search Tree :: Delete

Case 3: node to delete has 2 children
  - find deleted node's successor
    (will be in right subtree of deleted node, and will not have a left child)
  - replace deleted node with successor

DELETE \((3, 3, 10)\) = \((2, 5, 8)\)
Runtime of Insert and Delete

Insert $\rightarrow$ $O(\# \text{ levels})$

Delete
$\rightarrow$ 1) Find node to delete (search)
2) In worst-case, also need to find successor

Both are $O(\# \text{ levels})$ so Delete is $O(\# \text{ levels})$
So many algorithms are $O(\# \text{ levels})$!
In worst case, $\# \text{ levels} = n$

To minimize the $\#$ of levels in a tree, we need a balanced tree.

How can we ensure good performance, without doing too much tree rebalancing work?
and

Евгений Михайлович Ландис

came up with a balanced tree data structure
in 1962. Published in English in Journal of Soviet
Math, where their names were translated as

Georgii M. Adelson-Velskii and E. M. Landis.

we call their trees AVL trees. (Why not AJBT trees?)
AVL Trees

tree height: maximum length of a path from root to any leaf

AVL rule: for any node, height of left subtree and height of right subtree differ by at most ONE

Is this rule enough to guarantee the height is $O(\log n)$ in worst-case?
AVL Tree?

1. Yes
2. Yes
3. Yes
4. Yes
5. No
Does AVL rule guarantee height $O(\log n)$?

Let $\#$ nodes in left subtree = $n_{\text{left}}$
\# nodes in right subtree = $n_{\text{right}}$

$n_{\text{total}} = n_{\text{left}} + n_{\text{right}} + 1$
Because left and right subtree are AVL trees,

worst case is when they are not even,

so let $h_{\text{right}} = h_{\text{left}} - 1$.

or $h_{\text{right}} = h_{\text{left}} - 1$.

\[ h_{\text{total}} = h_{\text{left}} + 1 \]

So... $n = n_{\text{left}} + n_{\text{right}} + 1$

root has height $h$, left subtree has height $h - 1$, and right subtree has height $h - 2$.

can be written as $n_h = n_{h-1} + n_{h-2} + 1$.
worst case

\[ n_h = n_{h-1} + n_{h-2} + 1 \]

\[ n_0 = 1 \]

\[ n_1 = 1 + 0 + 1 = 2 \]

\[ n_2 = 2 + 1 + 1 = 4 \]

Note: 4 is the minimum number of nodes needed to have a legal AVL tree of height 2

\[ n_3 = 4 + 2 + 1 = 7 \]

7 is the minimum number of nodes needed to have a legal AVL tree of height 3

goal: is height \( O(\log n) \)?
\[ n_h = n_{h-1} + n_{h-2} + 1 \]

\[ \begin{array}{c|c}
  n_0 &= 1 \\
  n_1 &= 2 \\
  n_2 &= 4 \\
  n_3 &= 7 \\
  n_4 &= 12 \\
  n_5 &= 20 \\
  n_6 &= 33 \\
  \vdots & \vdots \\
  n_1 &= 1 \\
  n_2 &= 2 \\
  n_3 &= 3 \\
  n_4 &= 5 \\
  n_5 &= 8 \\
  n_6 &= 13 \\
  \vdots & \vdots \\
\end{array} \]

Fibonacci numbers!

note:

minimum # of nodes needed to have a height \( h \) is worst case for that height because more nodes would mean the tree is more balanced...
\[ n_h \geq \text{Fib}(h+3) - 1 \]

\[ n_h \geq \frac{\phi^{h+3} - \hat{\phi}^{h+3}}{\sqrt{5}} - 1 \]

\[ \text{Fib}(\hat{\phi}) = \frac{\phi - \hat{\phi}}{\sqrt{5}} \]

\[ \phi = \frac{1+\sqrt{5}}{2} \quad \hat{\phi} = \frac{1-\sqrt{5}}{2} \]

\[-1 < \hat{\phi} < 1\]

so \( \frac{\phi^{h+3}}{\sqrt{5}} \ll 1 \)

(either negative or very small positive)

\[ n_h > \frac{\phi^{h+3}}{\sqrt{5}} - 1 - 1 \]

\[ \sqrt{5}(n_h + 2) > \phi^{h+3} \]
\[ \sqrt{5 \left( n_h + 2 \right)} > \phi^{h+3} \]

\[ \log_\phi \left( \sqrt{5 \left( n_h + 2 \right)} \right) > h + 3 \]

\[ h < \log_\phi \sqrt{5} + \log_\phi \left( n_h + 2 \right) - 3 \]

\[ h < \log_\phi \left( n_h + 2 \right) - 1.33 \]

\[ h \in O \left( \log n \right) \quad \phi = \frac{1 + \sqrt{5}}{2} \approx 1.618 \]

is \( h \in \Omega \left( \log n \right) \)? yes, because the best case is a complete balanced tree

so \( h \in \Theta \left( \log n \right) \)
Then new node

is new node

type of child

Parent same

but from nomenclature AVL rule broken

Insert: Insert as before

but # levels is \( \Theta \log n \)

Search: \( \Theta (\# \text{ levels}) \) like always

AVL Tree: Run times
Total: \( O(\log n) \).

To find where to rebalance, so \( O(\log n) \) to rebalance.

Traverse path from new node to root

Insert: \( \Theta(\log n) \) to search
AVL::Delete

Delete as before, but then you may need to rebalance.
- again a $O(\log n)$ path traversed from deleted node to root to do rebalancing.

Traverse from 10 to root to rebalance.
Runtime of Delete

$O(\log n)$ to the delete.

then traverse path from deleted node to root to find rebalancing candidates

$\rightarrow O(\log n)$

($O(1)$ work to do each rebalancing)

Total: $O(\log n)$ to delete
Other Binary Search Trees

1. Red/black trees (ch. 13 of book)
   - Paths alternate between black and red nodes
   - New nodes are red. Then you rotate nodes if you need to.
   - Insertion/Deletion: $O(\log n)$
   - Height: At most $2 \log(n+1)$.

2. Splay Trees
   - Search target moved to root
   - Node just inserted moved to root
   - Not balanced, so worst case search is $O(n)$
   - But AMORTIZED search is $O(\log n)$
Amortized Analysis,

time required to perform a sequence
of operations is averaged.

This is not the performance of the expected
case or a "typical case"

but the average of a sequence of operations' time in worst-case.

Good when the algorithm involves work done in
one run to optimize a later run.