Review: Binary Search Tree (BST)

- **Structure** property (binary tree)
  - Each node has \( \leq 2 \) children
  - Result: keeps operations simple
- **Order** property
  - All keys in left subtree smaller than node’s key
  - All keys in right subtree larger than node’s key
  - Result: easy to find any given key

---

**BST: Efficiency of Operations?**

- Problem: operations may be inefficient if BST is unbalanced.
- Finds, inserts, deletes:
  - \( O(n) \) in the worst case
- **BuildTree**
  - \( O(n^2) \) in the worst case

---

**How can we make a BST efficient?**

**Observation**
- BST: the shallower the better!

**Solution**: Require and maintain a **Balance Condition** that
1. Ensures depth is always \( O(\log n) \) — strong enough!
2. Is efficient to maintain — not too strong!

- When we build the tree, make sure it’s balanced.
- **BUT...**Balancing a tree only at build time is insufficient because sequences of operations can eventually transform our carefully balanced tree into the dreaded list 😞
- So, we also need to also keep the tree balanced as we perform operations.

---

**Potential Balance Conditions**

1. Left and right subtrees of the **root** have equal number of nodes
   - **Too weak!** Height mismatch example:

2. Left and right subtrees of the **root** have equal height
   - **Too weak!** Double chain example:

3. Left and right subtrees of every node have equal number of nodes
   - **Too strong!** Only perfect trees \( 2^n – 1 \) nodes

4. Left and right subtrees of every node have equal height
   - **Too strong!** Only perfect trees \( 2^n – 1 \) nodes
The AVL Balance Condition

Left and right subtrees of every node have heights differing by at most 1

Definition: \( \text{balance} (\text{node}) = \text{height} (\text{node}.\text{left}) - \text{height} (\text{node}.\text{right}) \)

AVL property: for every node \( x \), \(-1 \leq \text{balance}(x) \leq 1\)

- Ensures small depth
  - Will prove this by showing that an AVL tree of height \( h \) must have a number of nodes exponential in \( h \) (i.e., height must be logarithmic in number of nodes)
- Efficient to maintain
  - Using single and double rotations

The AVL Tree Data Structure

An AVL tree is a self-balancing binary search tree.

Structural properties
1. Binary tree property (same as BST)
2. Order property (same as for BST)

1. Balance property:
   - balance of every node is between -1 and 1

Result: Worst-case depth is \( O(\log n) \)

- Named after inventors Adelson-Velskii and Landis (AVL)
  - First invented in 1962

Is this an AVL tree?

Yes! Because the left and right subtrees of every node have heights differing by at most 1

Implementing AVL Trees

Node structure

Tree operations
(We'll want to be sure these operate in \( O(\log n) \) worst case time.)

The shallowness bound

Let \( S(h) = \) the minimum number of nodes in an AVL tree of height \( h \)
- \( S(h) \) grows exponentially in \( h \).
  - (Can be proved, but we will not do it in class.)
  - Therefore, a tree with \( n \) nodes has a logarithmic height
  - Thus FIND can be done in \( O(\log n) \) time.
  - We will also see that INSERT and DELETE can be done in \( O(\log n) \) time, while maintaining the AVL property.
An AVL Tree

AVL tree operations
- AVL find: Same as BST find
- AVL insert: First BST insert, then check balance and potentially "fix" the AVL tree. Four different imbalance cases.
- AVL delete: The "easy way" is lazy deletion, otherwise, do the deletion and then check for several imbalance cases (we will skip this).

Insert: detect potential imbalance
1. Insert the new node as in a BST (a new leaf)
2. For each node on the path from the root to the new leaf, the insertion may (or may not) have changed the node's height
3. So after insertion in a subtree, detect height imbalance and perform a rotation to restore balance at that node

Fix: Apply “Single Rotation”
- Single rotation: The basic operation we’ll use to rebalance
  - Move child of unbalanced node into parent position
  - Parent becomes the “other” child (always okay in a BST!)
  - Other subtrees move in only way BST allows (next slide)

Case #1: Example
Insert(6)
Insert(3)
Insert(1)
Third insertion violates balance property
- happens to be at the root
What is the only way to fix this?

The example generalized
- Insertion into left-left grandchild causes an imbalance
  - 1 of 4 possible imbalance causes (other 3 coming up!)
- Creates an imbalance in the AVL tree (specifically a is imbalanced)
The general left-left case
• So we rotate at a
  – Move child of unbalanced node into parent position
  – Parent becomes the "other" child
  – Other sub-trees move in the only way BST allows:
    • using BST facts: X < b < Y < a < Z
• A single rotation restores balance at the node
  – To same height as before insertion, so ancestors now balanced

Another example: insert(16)

The general right-right case
• Mirror image to left-left case, so you rotate the other way
  – Exact same concept, but need different code

Two cases to go
Unfortunately, single rotations are not enough for insertions in the left-right subtree or the right-left subtree

Simple example: insert(1), insert(6), insert(3)
  – First wrong idea: single rotation like we did for left-left

Two cases to go
Unfortunately, single rotations are not enough for insertions in the left-right subtree or the right-left subtree

Simple example: insert(1), insert(6), insert(3)
  – Second wrong idea: single rotation on the child of the unbalanced node

Sometimes two wrongs make a right 😊
• First idea violated the order property
• Second idea didn’t fix balance
• But if we do both single rotations, starting with the second, it works! (And not just for this example.)
• Double rotation:
  1. Rotate problematic child and grandchild
  2. Then rotate between self and new child
The general right-left case

Insert, summarized

Pros and Cons of AVL Trees

Arguments against AVL trees:
1. Difficult to program & debug (but done once in a library!)
2. More space for height field
3. Asymptotically faster but rebalancing takes a little time
4. If “amortized” logarithmic time is enough, use “splay trees.”