CS136

# Binary Search Trees

# Review

- Trees give many O(lg n) operations
- Trees are only efficient when **balanced**
- Can have arbitrary trees
- **Heap**: parent larger than children
- Heap sort:
  - build a heap O(n lg n), read out values O(n lg n)
  - Works in place!
- **Binary search tree**: left < parent < right

# BST Interface

- Insert (“add”) key-value pair
- Get value from key
- Size
- Clear
- Remove value associated with key
- Contains key
- Iterate
- Balance?

# Example Usage: Dictionary

- Create a BST of ComparableAssociations
  - Order BST by key
  - Two objects are equal if keys are equal
- What would add(word, def) and lookup(word) look like using a BST?
- Different dictionary implementations in CS136

# Abstractions on Abstractions

- Assume we already have **BinaryTree<T>** (we can just grab it from structure5)
- Build **BST<T>** by abstracting around **BT<T>**
- class **BinarySearchTree<T>** { **BinaryTree<T>** root; }

# Helper Method

- Contains/get/remove/insert all depend on **locating** the correct place in the tree for the specific key
- Let’s abstract this functionality

# Locate

```
protected BT locate(BT top, Object value) {
  // pre: top and value are non-null
  // post: returns “highest” node with the desired value,
  //       or node to which value should be added
  Object topValue = top.value();
  BT child;
  // found at top: done
  if (topValue.equals(value)) return top;
  // look left if less-than, right if greater-than
  if (ordering.compare(topValue,value) < 0) {
      child = top.right();
  } else {
      child = top.left();
  }
  // no child there: not in tree, return this node,
  // else keep searching
  if (child.isEmpty()) { return top; }
  else { return locate(child, value); }
}
```

# Contains() is easy

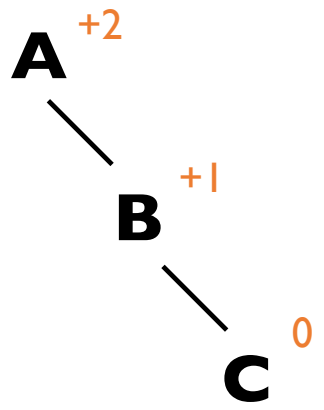
- Did locate return the value that we expected?

# Add (“insert”)

- If the tree is empty, insert at the root
- Otherwise, locate()
  - Replace (push down) parent or
  - Insert left or
  - Insert right
- What order should we insert to maintain balance?
- How can we maintain balance if we can’t insert in this order

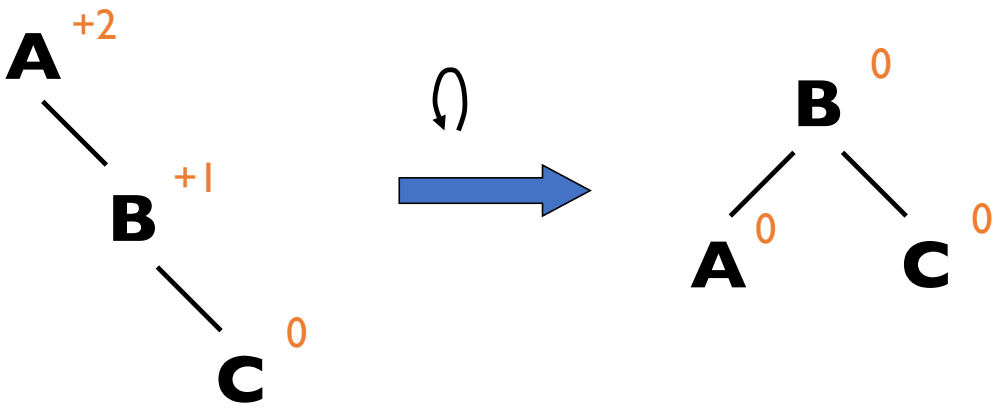
# Digression 1

- Can we design a binary search tree that always gives expected O(lg n) operations, regardless of insertion order?
- Yes!
- AVL trees; Adelson-Velsky and Landis, 1962
- Red-Black Trees; Baker, 1972
- Splay Trees; Sleator and Tarjan, 1985



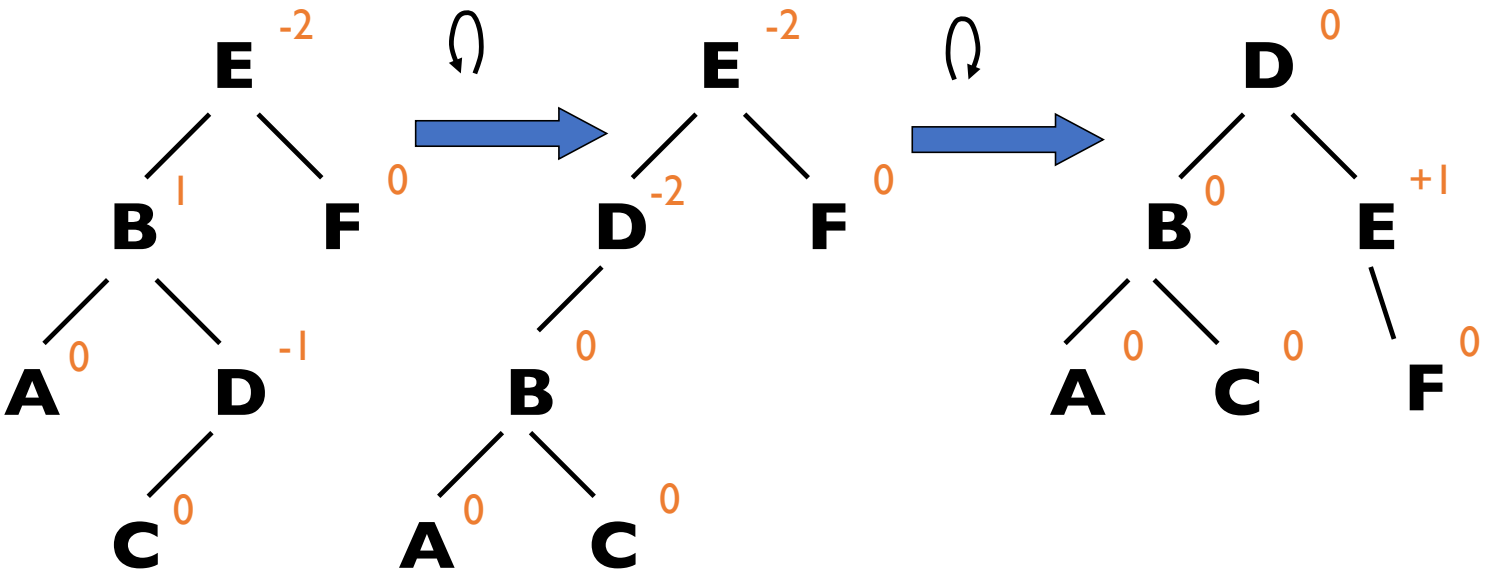

- The balance factor of a node is the *height* of its right subtree minus the height of its left subtree. A node with balance factor 1, 0, or -1 is considered balanced.
- A node with any other balance factor is considered unbalanced and requires rebalancing the tree.

# Single Rotation




Unbalanced trees can be rotated to achieve balance.

# Double Rotation



# Digression 2: BST-sort

- Build a BST sort analogous to Heap sort
- Is this a good idea?
- To be continued…