Searching Hierarchies

Searching Hierarchies

- One powerful use of binary trees is as a structure that more naturally captures what we do in ***binary search.***
- **Recall:** binary search *quickly* finds a key in a fixed-sized array by looking in the left or right portion of the array.
    - NOTE: we assume the array is *sorted*
    - [Khan Academy Demo of Binary Search on Fixed Arrays]
- A tree naturally supports this sort of operation…
- AND a tree can be dynamically modified (insert/delete)!
- Trees, thus, are a dynamic structure that supports efficient "look-up".

Binary Search Trees

Binary Search Trees

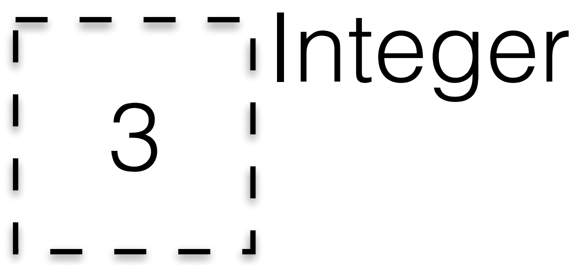
- Binary Search Trees are implemented as "generic" structures with 2 parameters:
  - Key
  - Value
- Ex. Mug shots
  - key = String corresponding to a person's name
  - value = BufferedImage corresponding to a person's mug shot
- Ex. Word count
  - key = String (the word)
  - value = Integer (the # of occurrences of that word)

NOTE: this would have to be an **Integer** not an **int**.

Aside: Generics

Generics: Primitives vs. Objects

- When dealing with generics (think: Java "Collections") in Java, types <u>*must*</u> to be *objects* (i.e., can't specify *primitives*).
- What about primitive types (int, float, boolean, etc.)?
- [enter Java **"wrapper" classes**]
- Java has wrapper classes that… "wrap" or "box" primitive types:
  - Ex. **int** is wrapped by **Integer**
  - Ex. **float** is wrapped by **Float**
  - Ex. **boolean** is wrapped by **Boolean**
- Java sometimes takes care of this for you if it can infer the type — this is known as **"autoboxing"** and **"unboxing"**



“Auto-bots-ing” :)

Back to
Binary Search Trees

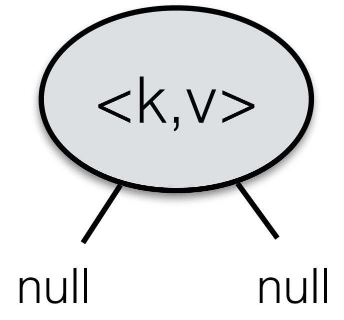
Binary Search Trees

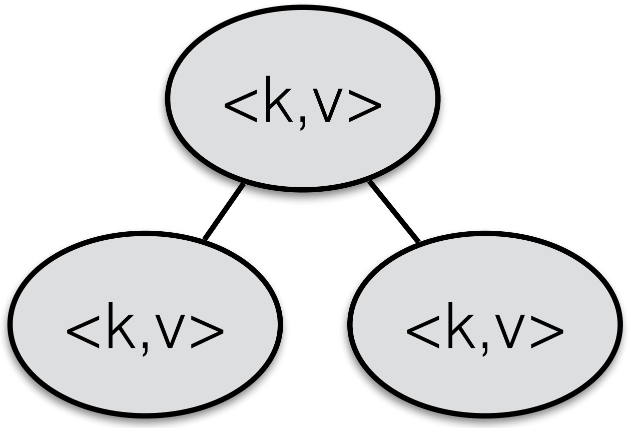
```
/**
 * Generic binary search tree
 */
public class BST<K extends Comparable,V> {
    private K key;
    private V value;
    private BST<K,V> left, right;

    /**
     * Constructs leaf node -- left and right are null
     */
    public BST(K key, V value) {
        this.key = key; this.value = value;
    }

    /**
     * Constructs inner node
     */
    public BST(K key, V value, BST<K,V> left, BST<K,V> right) {
        this.key = key; this.value = value;
        this.left = left; this.right = right;
    }


    // . . . the rest of the class defined below . . .
}
```





Binary Search Trees: Searching

Binary Search Trees: find()

- Think about how trees could be useful for searching…
- BST property says:
    - ***keys*** in the ***left subtree*** are ***less than*** those in the parent.
    - ***keys*** in the ***right subtree*** are ***greater than*** those in the parent.
    - we assume uniqueness right now (i.e., no duplicate node keys).
- Q: How to find nodes with a given ***search key***?
- A: compare the node keys!
    - if equal ==> done!
    - if less than ==> recurse left!
    - if greater than ==> recurse right!
- Problem: How to compare nodes?

Java's "Comparable" Interface

```java
public class BST<K extends Comparable,V> {

   // . . . the rest of the class defined below . . .

}
```

“Comparable”

- We have to be able to compare objects!
- But we don't know what kinds of things are in the tree — it's generic.
- If we knew the keys were ints, we could use < and > (e.g., <int>), but what do we do for generics?
- Java provides an interface **Comparable**
    - it defines a method **compareTo()** to test two objects.
        - returns **0** if a and b are equal
        - returns a **negative number** if a < b
        - returns a **positive number** a > b
- So we have to specify that it's not any old class for the key, but one that ***implements*** “Comparable”. Confusingly, we say "**K extends Comparable**" even though it's an interface.

“Comparable”

```
/*
 * A Super Bowl 2015 example for using compareTo().
 */
String s1 = "Seahawks";
String s2 = "Patriots";

int result = s1.compareTo(s2);
if (result > 0)
    System.out.println(s1 + " is greater!");
else if (result == 0)
    System.out.println("I guess they are equal...");
else if (result < 1)
    System.out.println(s2 + " is greater!");
else
    System.out.println("ERROR! compareTo() shouldn't return anything other than -1, 0, or 1...");
```

```
Output: Seahawks is greater!
```

Binary Search Trees: find()

```java
/**
 * Returns the value associated with the search key, or throws an exception if not in BST
 */
public V find(K search) throws InvalidKeyException {
    System.out.println(key); // to illustrate

    int compare = search.compareTo(key);
    if (compare == 0)
        return value;
    if (compare < 0 && hasLeft())
        return left.find(search);
    if (compare > 0 && hasRight())
        return right.find(search);

    throw new InvalidKeyException(search.toString());
}
```

Custom Exceptions

Custom Exceptions

- Q: Why would you ever want to create a custom exception?

```java
/**
 * An exception for when a key is not found in a BST
 */
public class InvalidKeyException extends RuntimeException {
    public InvalidKeyException(String message) {
        super(message);
    }
}
```

- Q: Why would returning **null** maybe not be a good idea when searching for some key/value pair in a BST?

Binary Search Trees: Inserting

Binary Search Trees: insert()

- Similar to searching — i.e., find()
- Want to preserve BST property when inserting…
- Q: How to insert new node (key/value pair) into BST?
- A: find where to insert by comparing the node **keys**!
  - if equal ==> replace value with new one!
  - if we reach a leaf ==> add new node there!
  - compareTo() indicates if the key-to-insert is less than/greater than the current key ==> recurse left/right as needed!

Binary Search Trees: insert()

```
/**
 * Inserts the key & value into the tree (replacing old key if equal)
 */
public void insert(K key, V value) {
    int compare = key.compareTo(this.key);
    if (compare == 0) {
        // replace
        this.value = value;
    }
    else if (compare < 0) {
        // insert on left (new leaf if no left)
        if (hasLeft())
            left.insert(key, value);
        else
            left = new BST<K,V>(key, value);
    }
    else if (compare > 0) {
        // insert on right (new leaf if no right)
        if (hasRight())
            right.insert(key, value);
        else
            right = new BST<K,V>(key, value);
    }
}
```

Binary Search Trees: Deleting

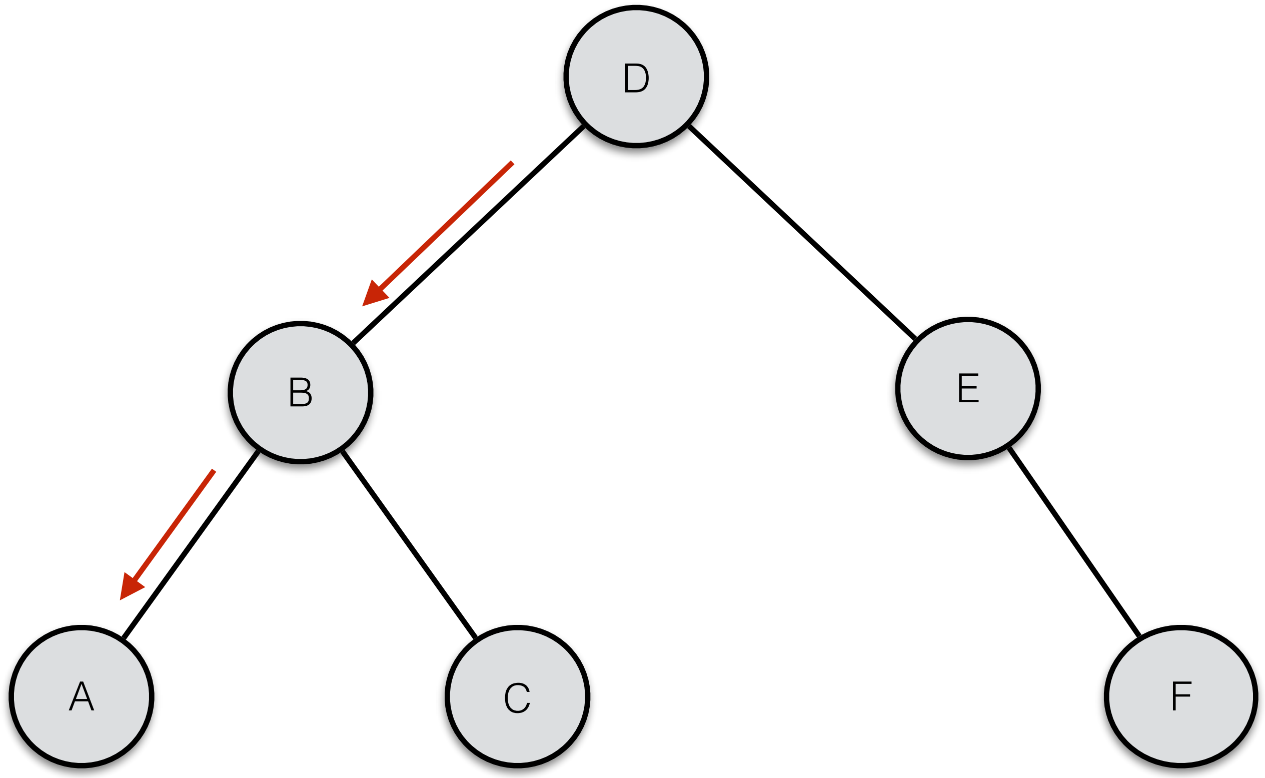
Binary Search Trees: delete()

- Deletion is trickier…
- Q: How to delete node (key/value pair) from BST ***and*** still preserve the BST property?
- A: find the node to delete by comparing the node **keys**!
  - if equal ==> delete…but what about children?
    - *[different cases are addressed in subsequent slides…]*
  - compareTo indicates if the key-to-delete is less than/greater than the current key ==> recurse left/right as needed!

Binary Search Trees: delete()

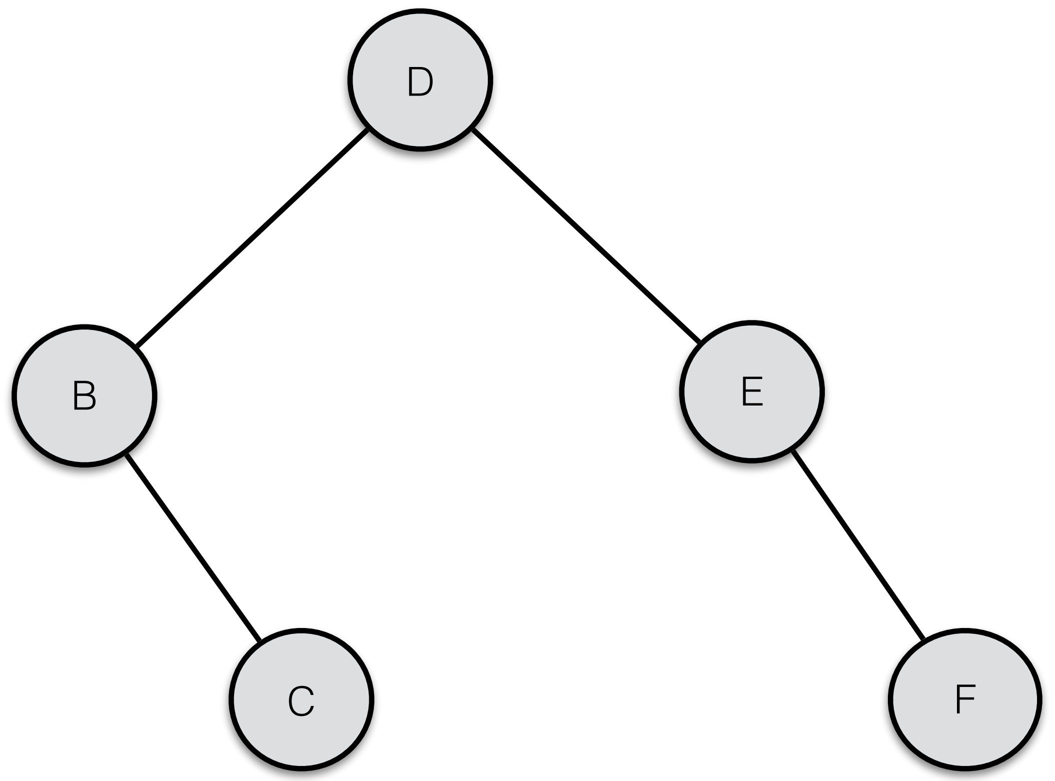
- **Easy case:** deleting if node has **no children.**
- trick: we always say point it at the "other" (which is null in this case).
    - we do this to make the case of deleting a node with *one* child easier (next slide)

delete("A")



Binary Search Trees: delete()

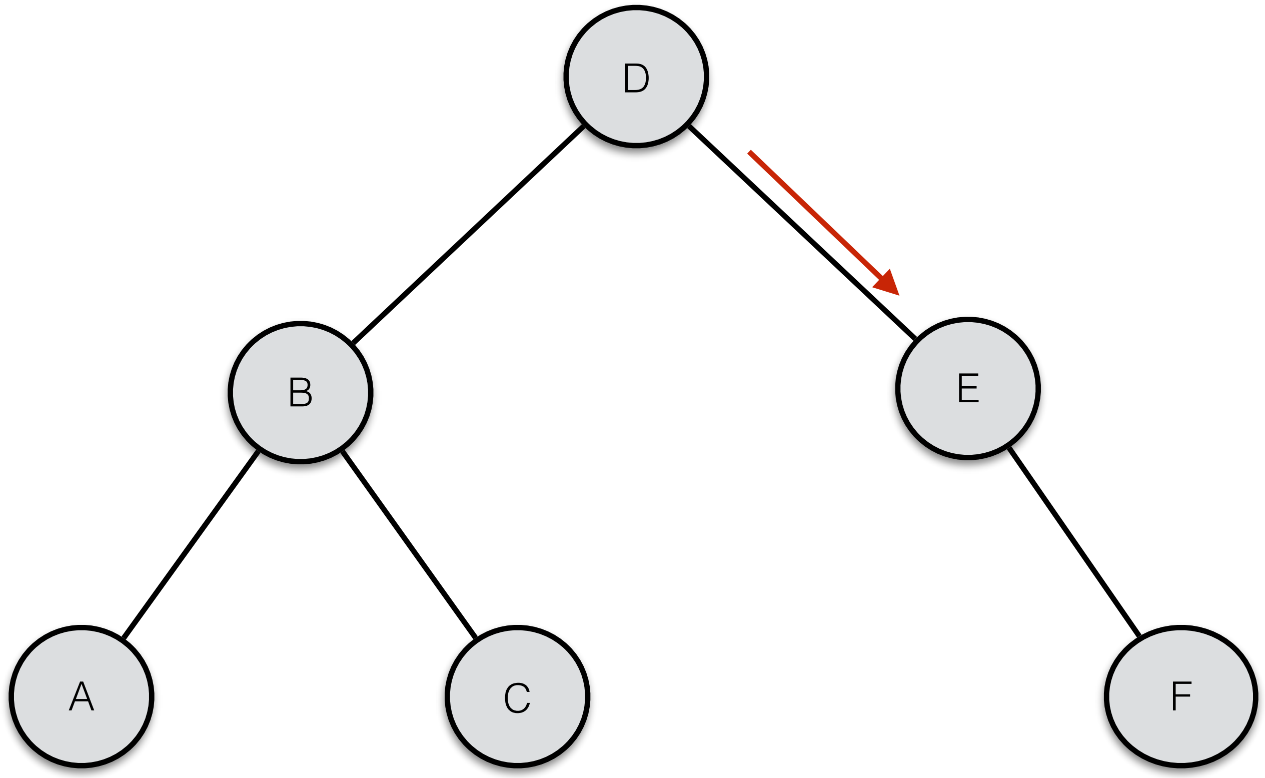
delete("A")



Binary Search Trees: delete()

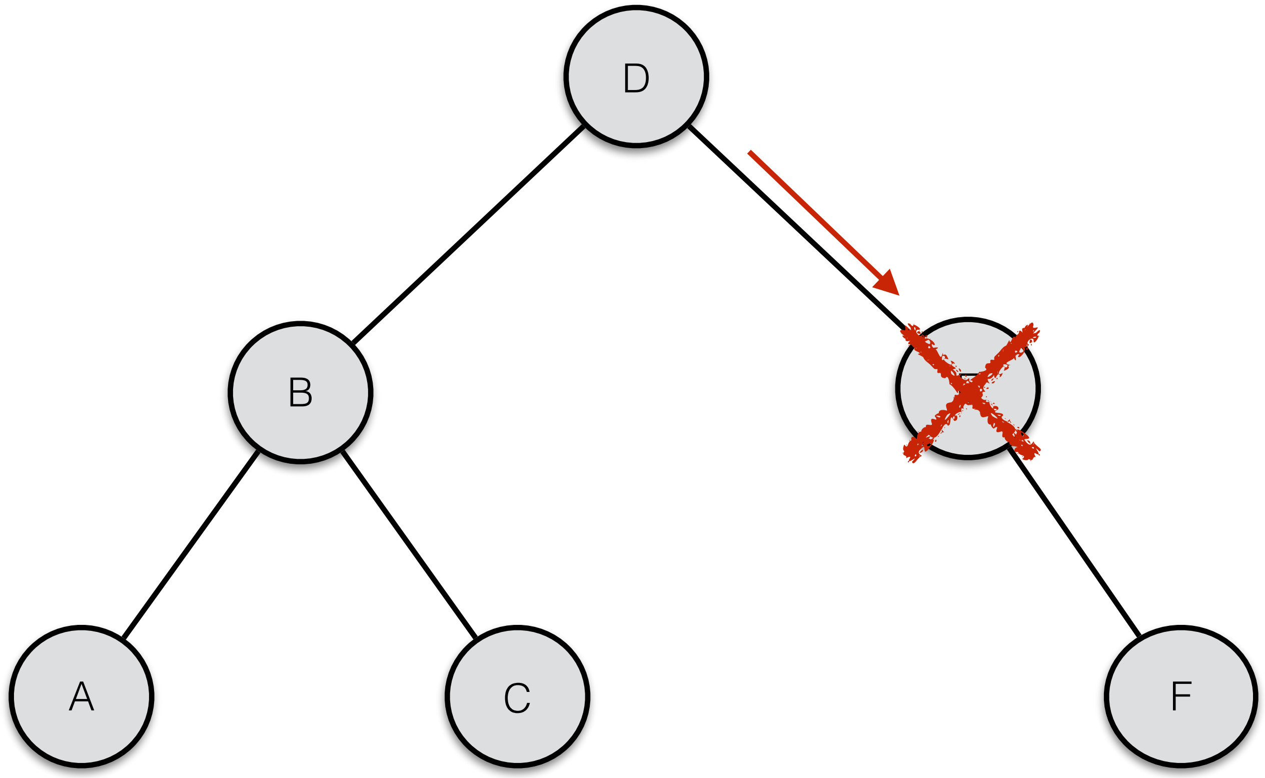
- **Easy case:** deleting if node has **1 child.**
- trick: we always say point it at the "other" (which is null).
    - "E" has no left
    - *replace* "E" with right child "F"

delete("E")



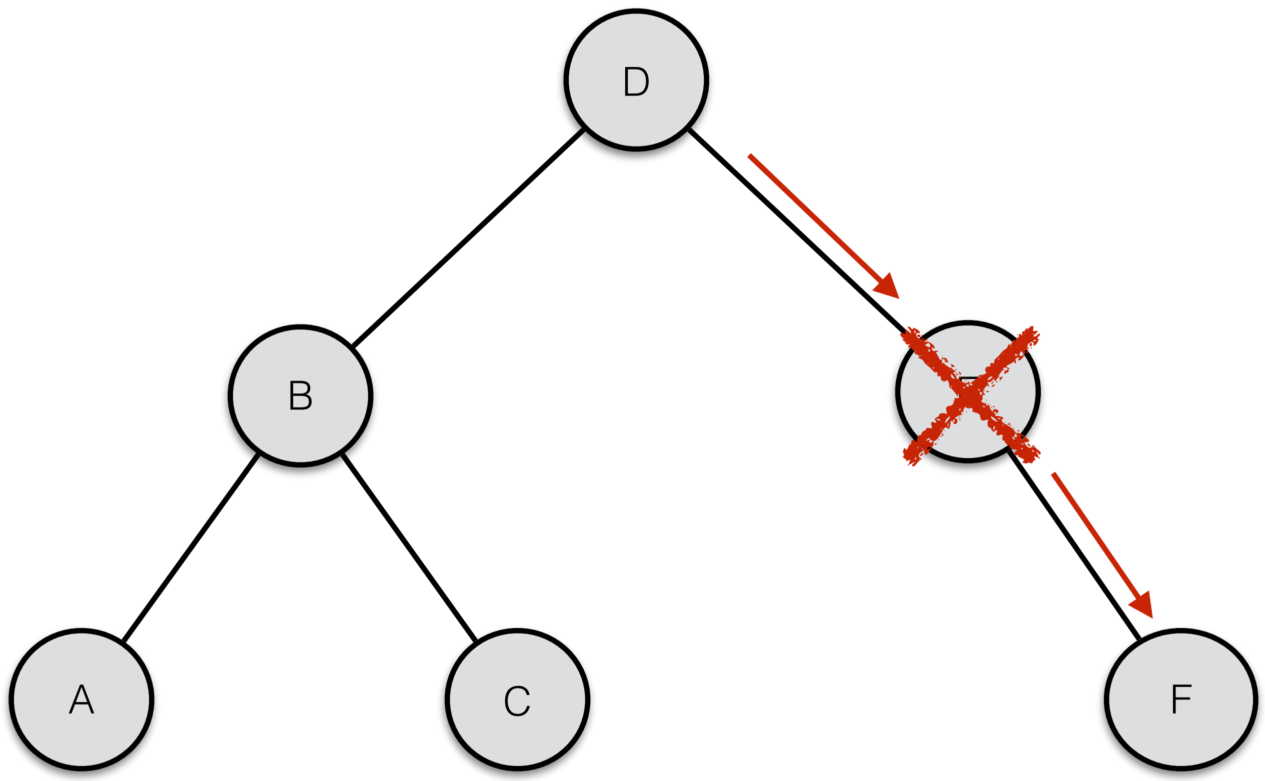
Binary Search Trees: delete()

delete("E")



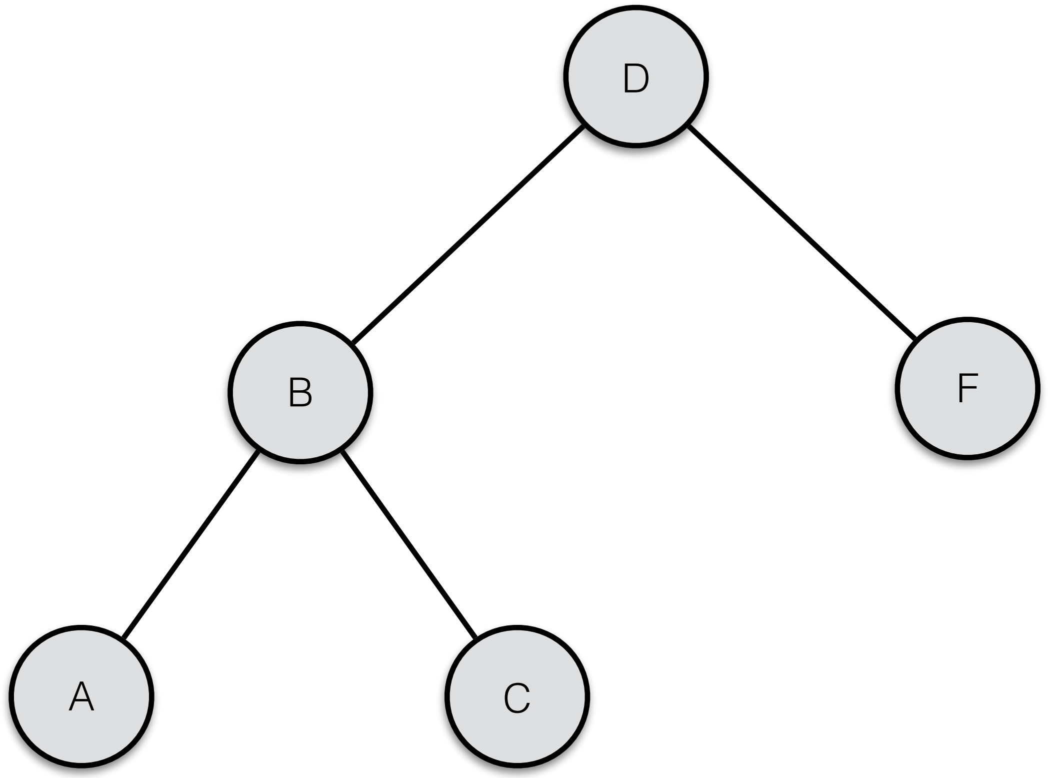
Binary Search Trees: delete()

delete("E")



Binary Search Trees: delete()

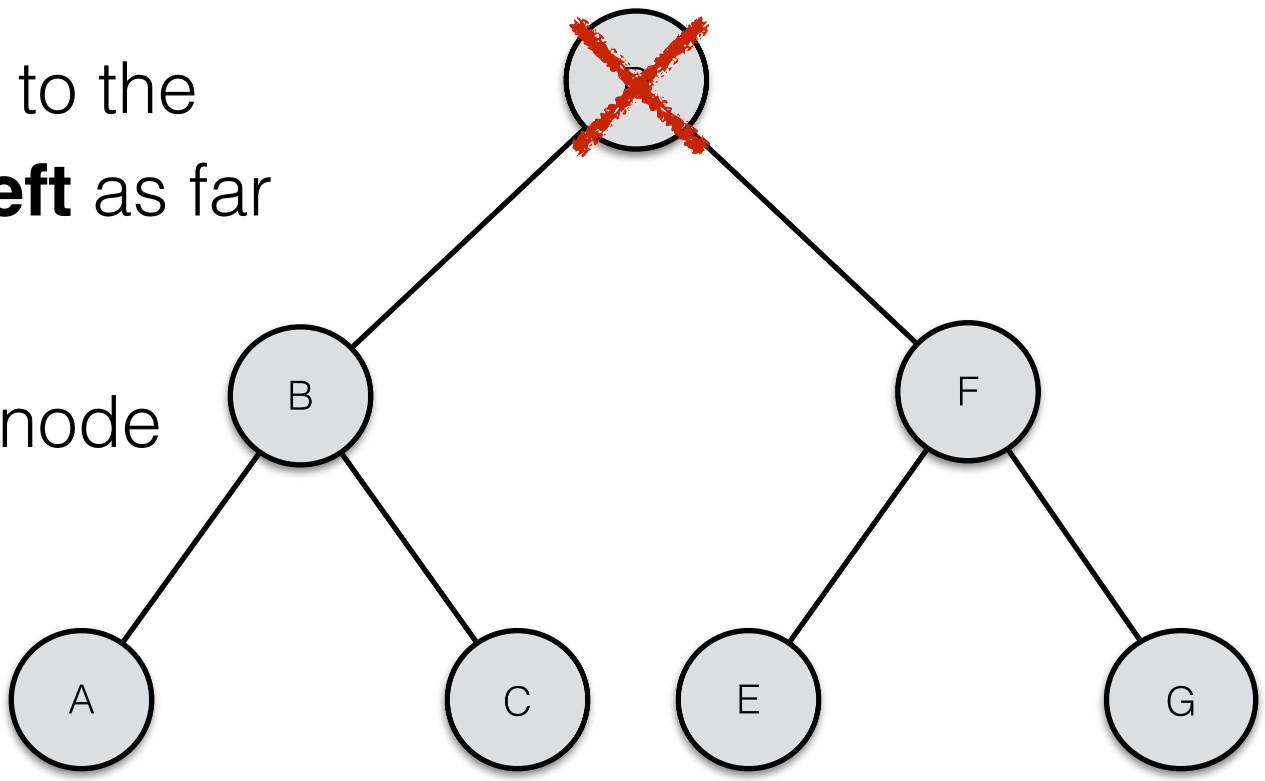
delete("E")



Binary Search Trees: delete()

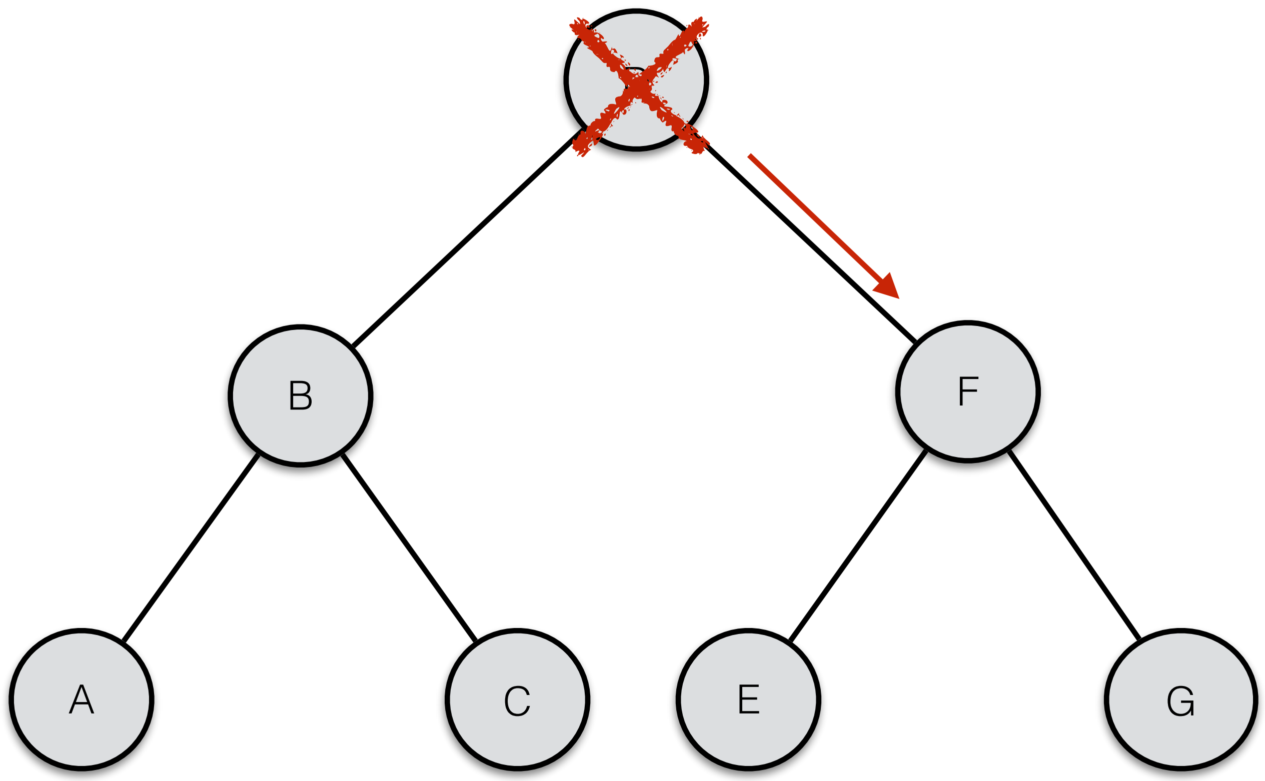
- **Problem case:** deleting if node has **2 children.**
- Can't point parent at both…
- **Solution:** replace by immediate predecessor ("C") or successor ("E")!
    - Immediate successor is smallest thing in right subtree
    - Find immediate successor by going to the right (once), then following the tree **left** as far down as possible
    - Recursion will stop when you find a node with 0 or 1 child — easy to delete!

delete("D")



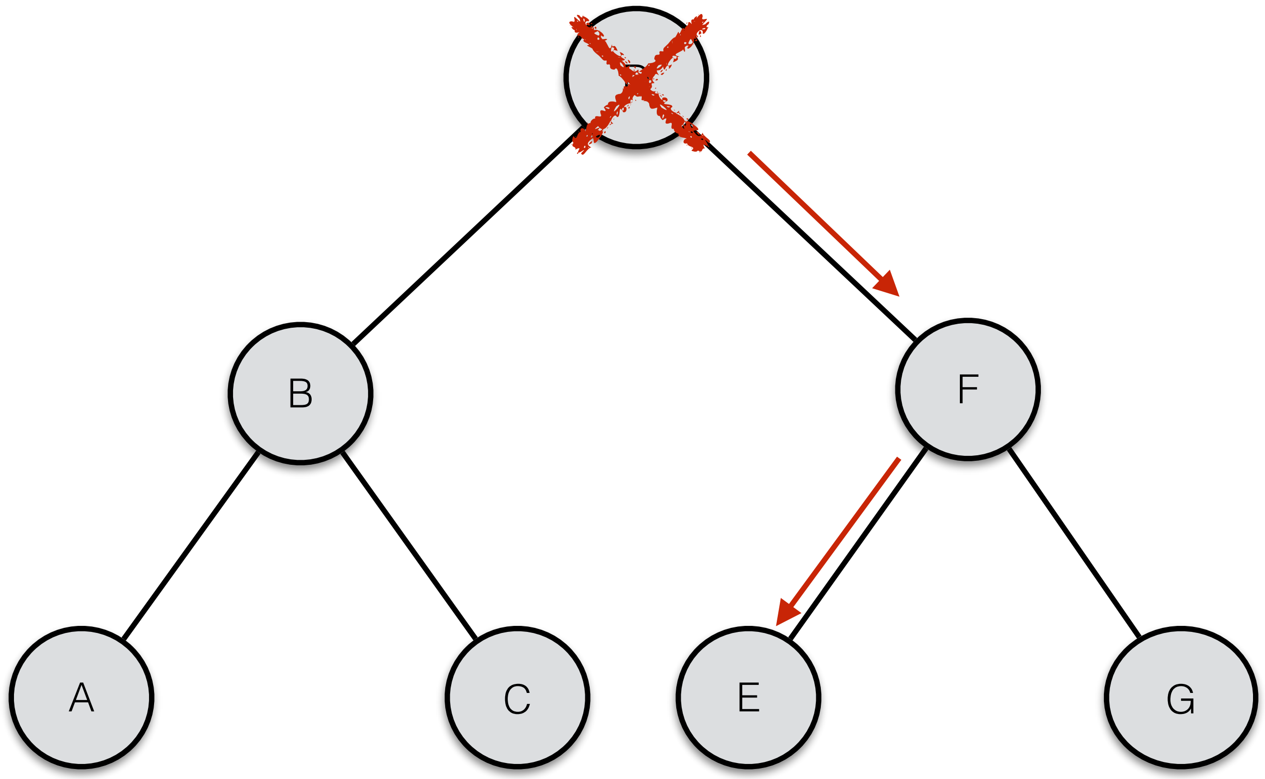
Binary Search Trees: delete()

delete("D")



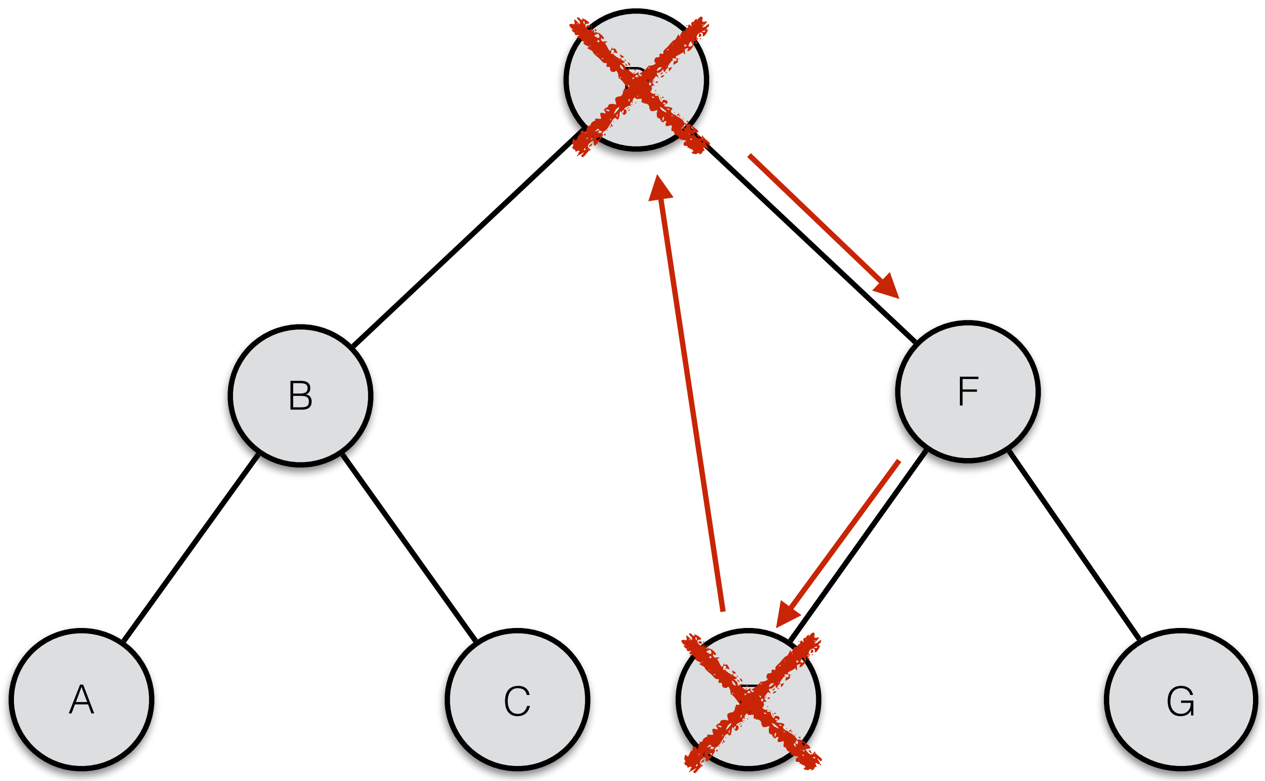
Binary Search Trees: delete()

delete("D")



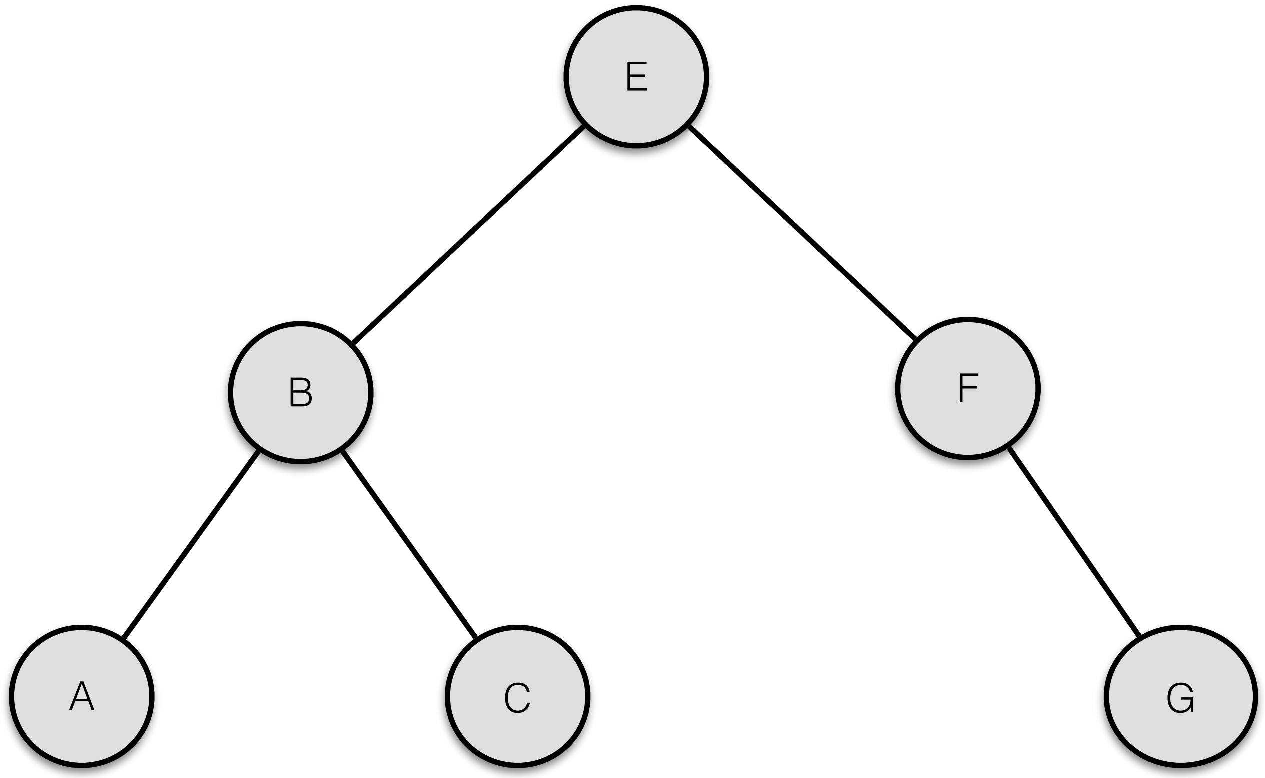
Binary Search Trees: delete()

delete("D")



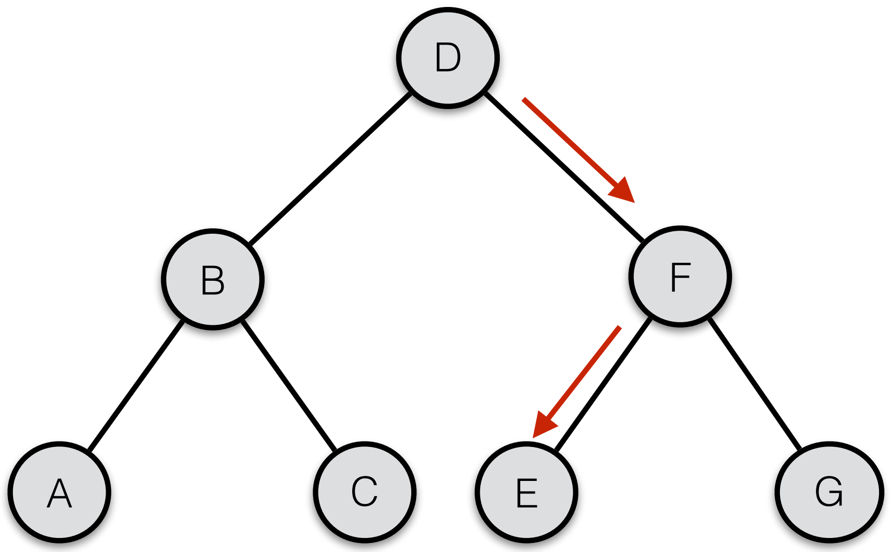
Binary Search Trees: delete()

delete("D")



Binary Search Trees: delete()

```
/**
 * Deletes the key and returns the modified tree, which might not be the same object as the original tree
 * Thus must afterwards just use the returned one
 */
public BST<K,V> delete(K search) throws InvalidKeyException {
    int compare = search.compareTo(key);
    if (compare == 0) {
        // Easy cases: 0 or 1 child -- return other
        if (!hasLeft()) return right;
        if (!hasRight()) return left;
        // If both children are there, rotate up the successor (smallest on right)
        // Find it
        BST<K,V> successor = right;
        while (successor.hasLeft()) successor = successor.left;
        // Delete it
        right = right.delete(successor.key);
        // And take its key & value
        this.key = successor.key;
        this.value = successor.value;
        return this;
    }
    else if (compare < 0 && hasLeft()) {
        left = left.delete(search);
        return this;
    }
    else if (compare > 0 && hasRight()) {
        right = right.delete(search);
        return this;
    }
    throw new InvalidKeyException(search.toString());
}
```



Implications
of
Insertions & Deletions

Implications of Insertions & Deletions

- Lots of insertions/deletions leads to an **unbalanced** tree structure
- **Worst case:** could end up with linear chain of nodes
  - “vine” not tree…
- To ensure a **balanced** tree requires more bookkeeping and occasional fix-ups
- We will see Red-Black Trees towards the end of the term!