CSCI 136
Data Structures &
Advanced Programming

Binary Seach Trees II

Binary Search Trees II

Binary Search Tree Implementation

BST Implementation

- The BST holds the following items
  - BinaryTree root: the root of the tree
  - BinaryTree EMPTY: a static empty BinaryTree
    - To use for all empty nodes of tree
  - int count: the number of nodes in the BST
  - Comparator<E> ordering: for comparing nodes
    - Note: E must implement Comparable
- Two constructors: One takes a Comparator
  - The other creates a NaturalComparator

BST Implementation: locate

- Several methods search the tree
  - get, contains, add, remove,
- We factor out common code: locate method
- *protected* locate(BinaryTree<E> *node*, E *v*)
  - Returns a BinaryTree<E> in the subtree with root *node* such that either
    - *node* has its value equal to *v*, or
    - *v* is not in this subtree and *node* is where *v* would be added as a (left or right) child
- How would we implement locate()?

BST Implementation: locate

BinaryTree locate(BinaryTree root, E val)

if root's value equals val return root

child ← child of root whose subtree should hold val

if child is emptry tree, return root

// val not in subtree based at root

else //keep looking

return locate(child, val)

BST Implementation: locate

- What about this line?

  child ← child of root whose subtree should hold value
- If the tree can have multiple nodes with same value, then we need to be careful
- Convention: During *add* operation, only move to right subtree if value to be added is *greater than* value at node
  - We'll look at *add* later
- Let's look at *locate* now....

The code : locate

```
protected BinaryTree<E> locate(BinaryTree<E> root, E value) {
        E rootValue = root.value();
        BinaryTree<E> child;

        // found at root: done
        if (rootValue.equals(value)) return root;

        // look left if less-than, right if greater-than
        if (ordering.compare(rootValue,value) < 0)
            child = root.right();
        else
            child = root.left();

        // no child there: not in tree, return this node,
        // else keep searching
        if (child.isEmpty()) return root;
        else
            return locate(child, value);
}
```

Other core BST methods

- locate(v) returns either a node containing v or a node where v can be added as a child
- locate() is used by
  - `public boolean contains(E value)`
  - `public E get(E value)`
  - `public void add(E value)`
  - `Public void remove(E value)`
- Some of these also use another utility method
  - `protected BT predecessor(BT root)`
- Let's look at contains() first...

Contains

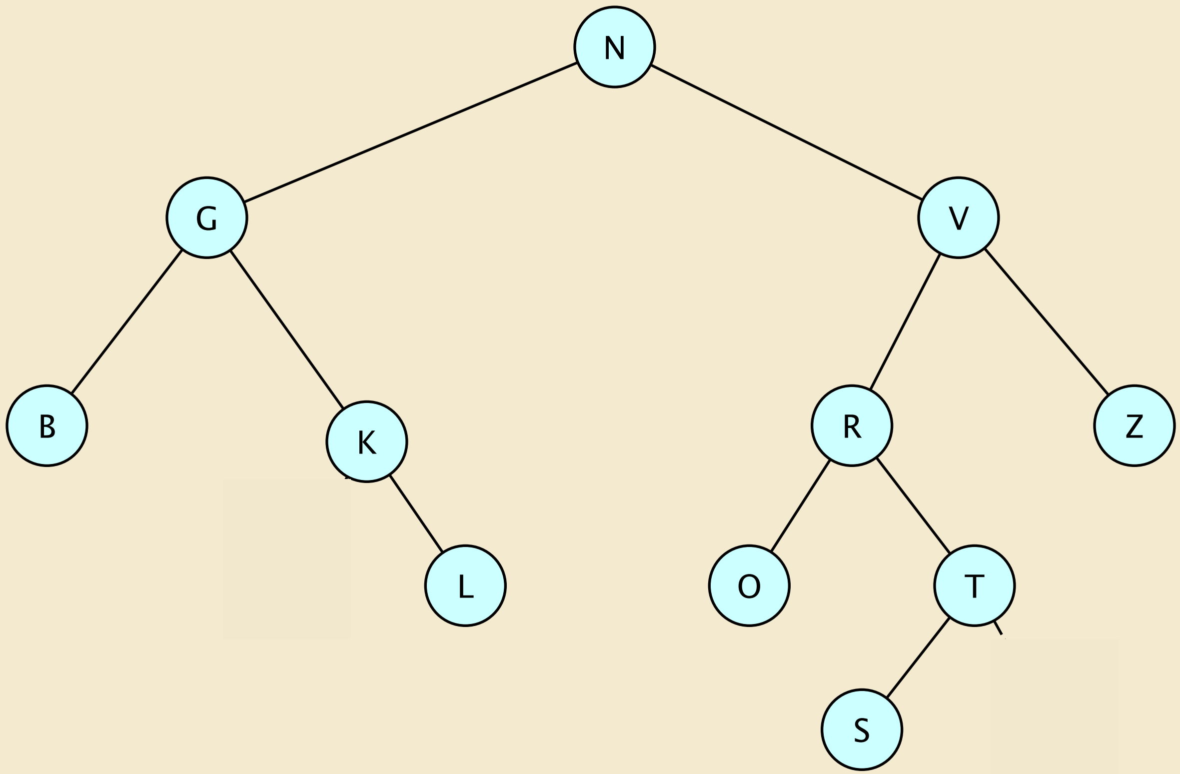
```
public boolean contains(E value){
   if (root.isEmpty()) return false;

   BinaryTree<E> possibleLocation = locate(root,value);

   return value.equals(possibleLocation.value());
}
```

First (Bad) Attempt: add(E value)

```
public void add(E value) {
        BinaryTree<E> newNode = new BinaryTree<E>(value,EMPTY,EMPTY);
        if (root.isEmpty()) root = newNode;
        else {
                BinaryTree<E> insertLocation = locate(root,value);
                E nodeValue = insertLocation.value();
        if (ordering.compare(nodeValue,value) < 0)
                insertLocation.setRight(newNode);
        else
                insertLocation.setLeft(newNode);
        }
        count++;
}
```

Problem: If repeated values are allowed, left subtree might not be empty when setLeft is called

Add: Repeated Nodes



Where would a new K be added?
A new V?

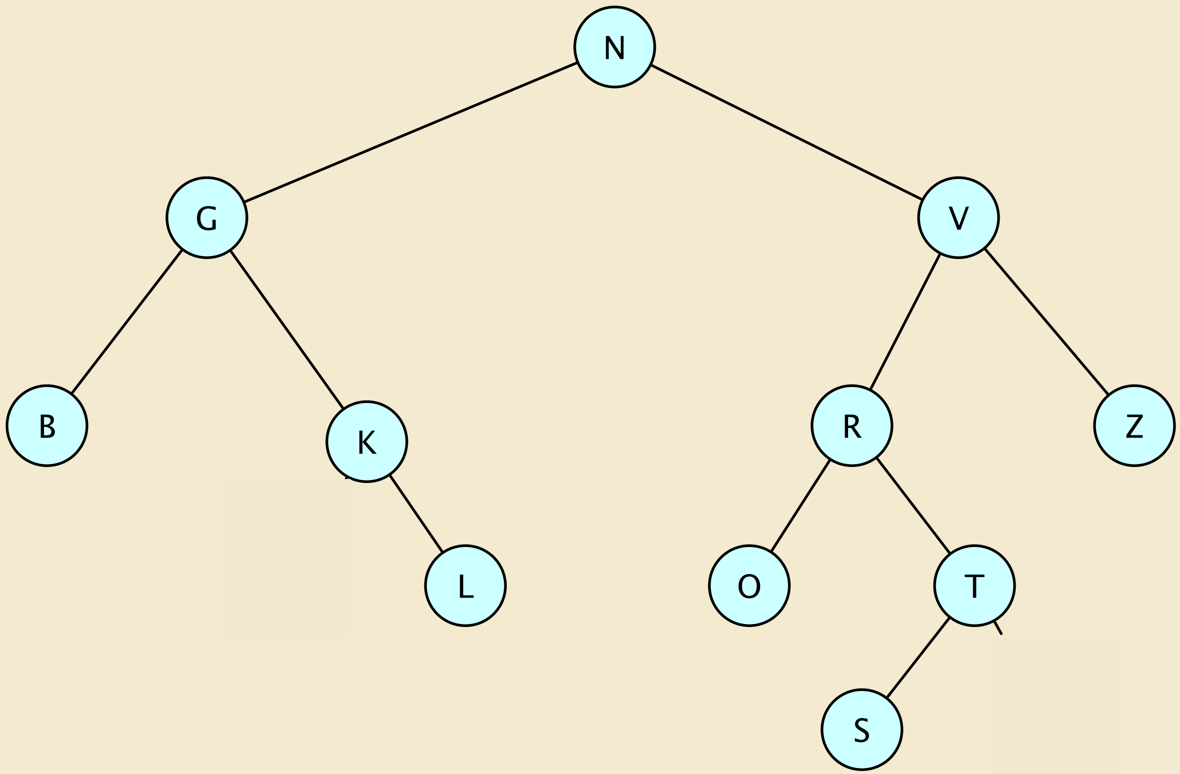
Add Duplicate to Predecessor

- If insertLocation has a left child then
  - Find insertLocation's predecessor
  - Add repeated node as right child of predecessor
  - If insertLocation has a left subtree that's where predecessor will be
    - This claim requires justification!
    - We'll return to this claim later

Corrected Version: add(E value)

```java
BinaryTree<E> newNode = new BinaryTree<E>(value,EMPTY,EMPTY);
if (root.isEmpty()) root = newNode;
else {
    BinaryTree<E> insertLocation = locate(root,value);
    E nodeValue = insertLocation.value();
    if (ordering.compare(nodeValue,value) < 0)
        insertLocation.setRight(newNode);
    else
        if (insertLocation.left().isEmpty())
            insertLocation.setLeft(newNode);
        else
            // if value is in tree, we insert just before
            predecessor(insertLocation).setRight(newNode);
}
count++;
```

Add: Repeated Nodes



Where would a new K be added?
A new V?

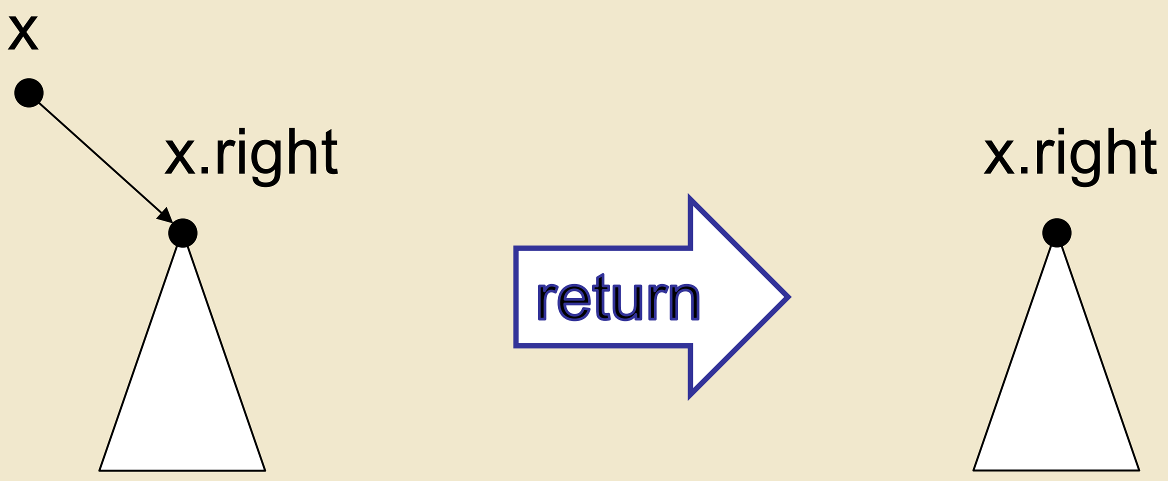
Predecessor

```
protected BinaryTree<E> predecessor(BinaryTree<E> root) {
    Assert.pre(!root.isEmpty(), ”Root has predecessor");
    Assert.pre(!root.left().isEmpty(),"Root has left child.");

    BinaryTree<E> result = root.left();

    while (!result.right().isEmpty())
        result = result.right();

    return result;
}
```

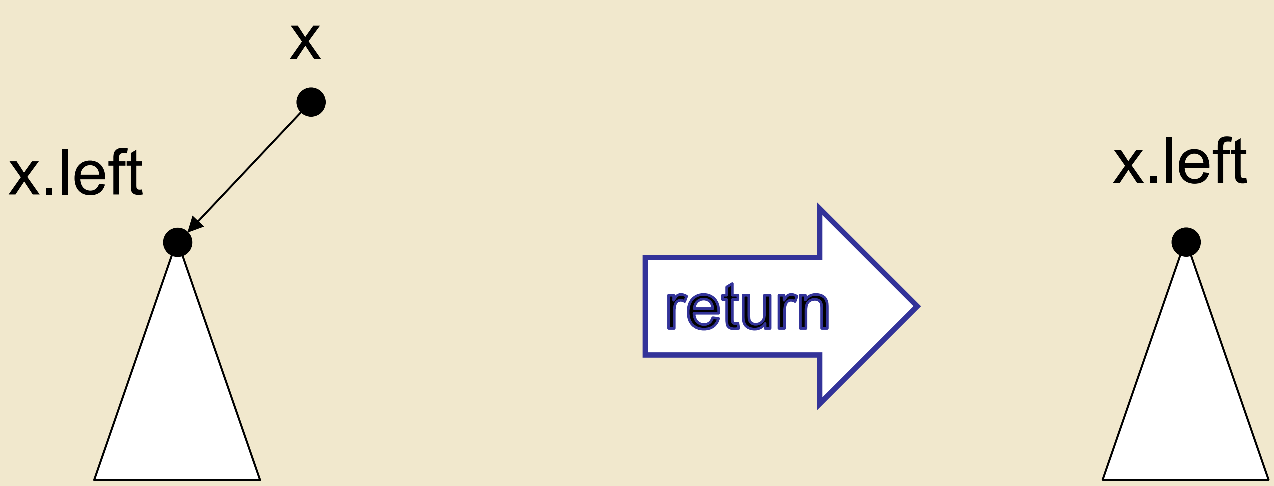
Removal

- Removing the root is a (not so) special case
- Let's figure that out first
  - If we can remove the root, we can remove any element in a BST in the same way
    - Every node is the root of a subtree
    - After removing *node* as root of its own subtree, add subtree back as child of *node*'s parent
- We need to implement:
  - `public E remove(E item)`
  - `protected BT removeTop(BT top)`
  - `remove(item)` finds `node` top holding item, then calls `removeTop(node)`

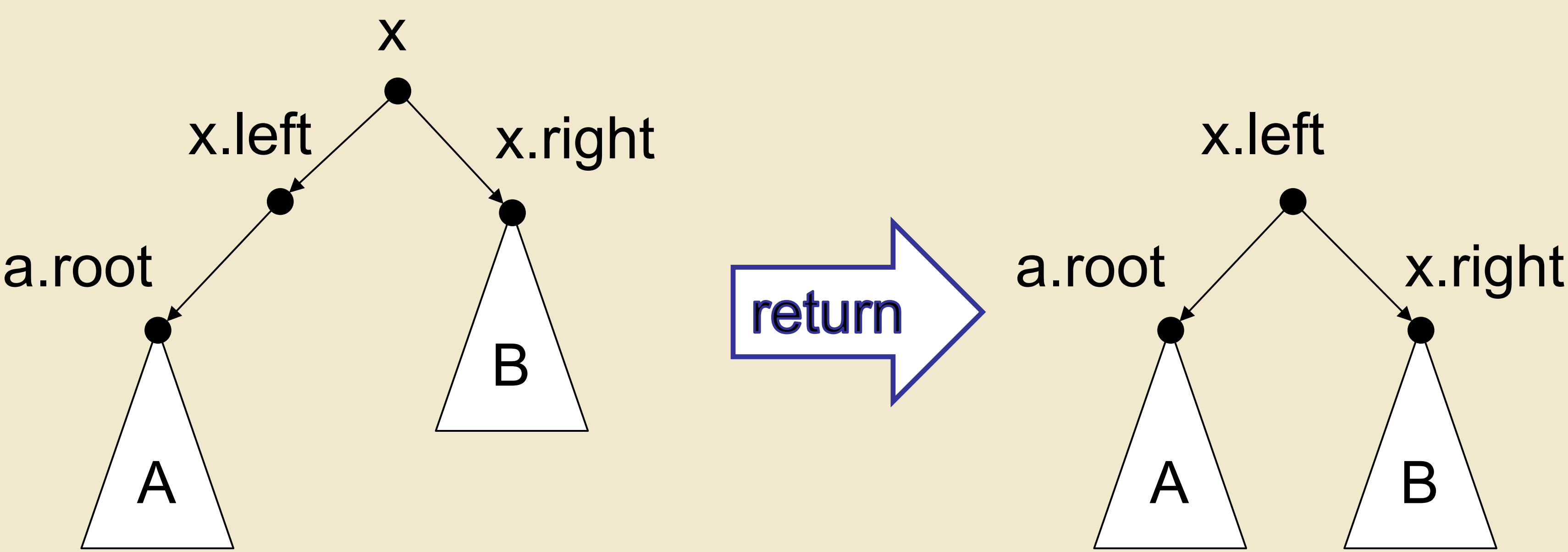
Case 1: No Left Subtree



Case 2: No Right Subtree



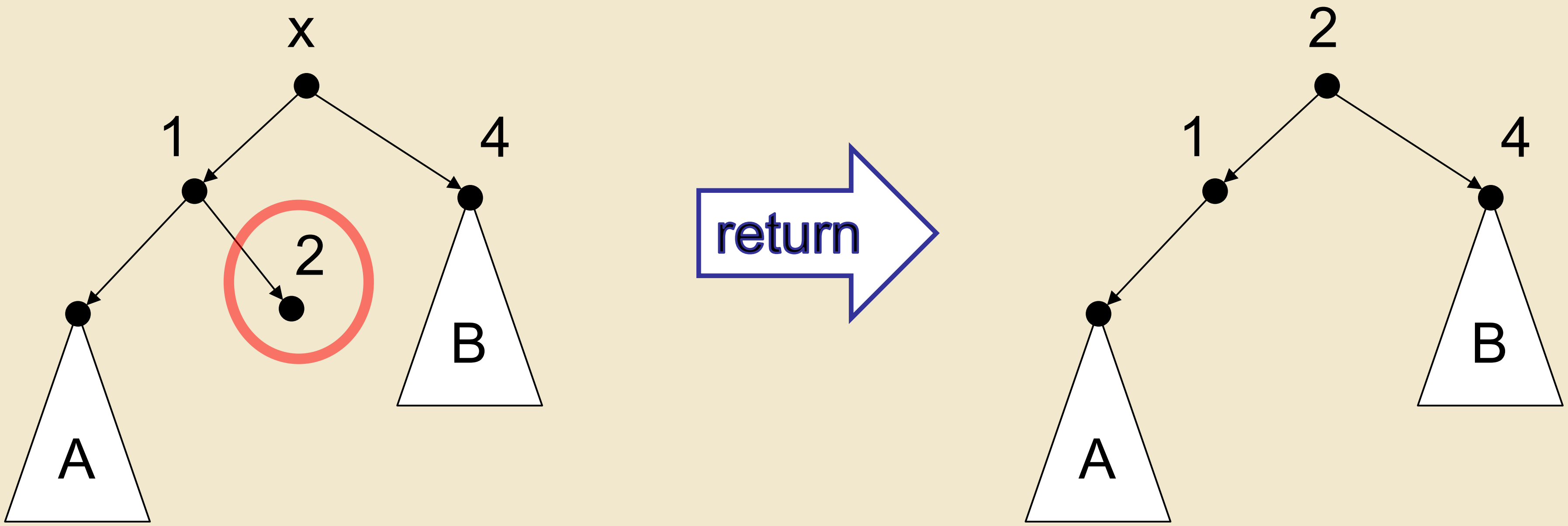
Case 3: Left Has No Right Subtree



Case 4: General Case

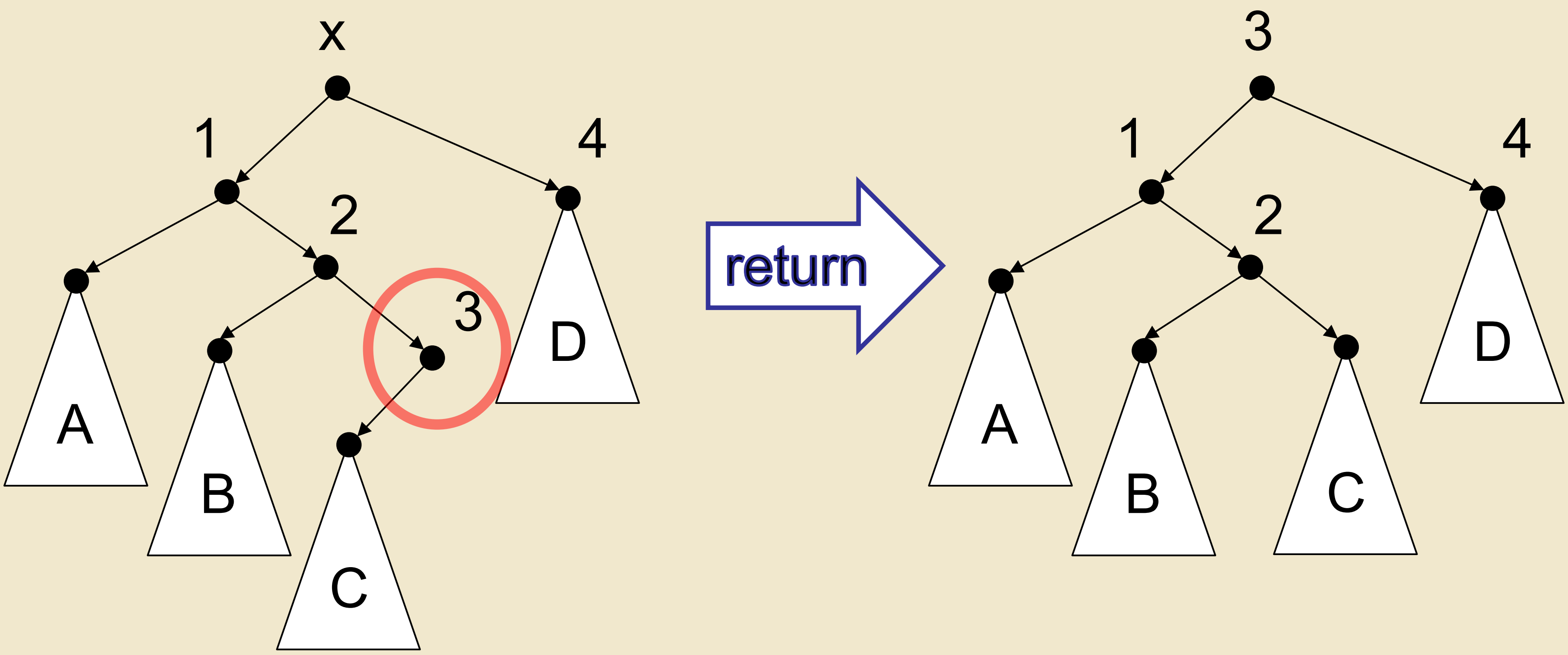
- Consider BST requirements:
  - Left subtree must be <= root
  - Right subtree must be > root
- Strategy: replace the root with the largest value that is less than or equal to it
  - predecessor(root) : rightmost left descendant
- This may require reattaching the predecessor's left subtree!

Case 4: General Case



Replace root with predecessor(root),
then patch up the remaining tree

Case 4: General Case



Replace root with predecessor(root), then patch up the remaining tree

RemoveTop(topNode)

Detach left and right sub-trees from root (i.e. topNode)

*If either left or right is empty, **return** the other one*

If left has no right child

*make right the right child of left then **return** left*

Otherwise find largest node C in left

// C is the right child of its own parent P

// C is the predecessor of right (ignoring topNode)

Detach C from P; make C's left child the right child of P

Make C new root with left and right as its sub-trees

Summary & Observations

- A Binary Search Tree is structured so that in-order traversal produces items in sorted order
- Add and remove preserve BST order
- Public methods depend on locate/predecessor protected methods
- Locate/predecessor methods take O(h) time
  - So all search and update methods take O(h) time
- Next step: Make h small!
  - Impose additional structure on BST to do so