VILLANOVA UNIVERSITY
IGNITE CHANGE. GO NOVA.

CSC 1052 – Algorithms & Data Structures II:

Binary Search Trees

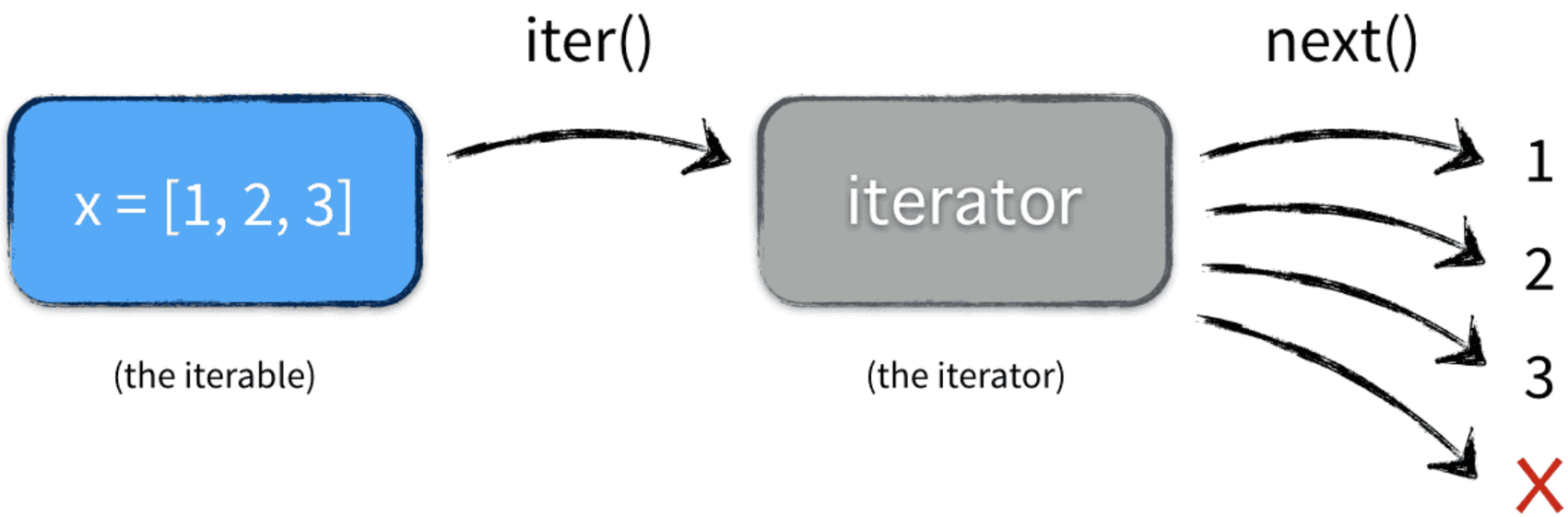
Professor Henry Carter
Spring 2017

Recap

- The binary tree interface extends the collection with a few new methods
  - Min, max, getIterator
- The recursive structure of the tree lends itself to recursive implementations
  - Iterative versions will likely use a stack
- Careful definition of the base and recursive cases can lead to very simple code
  - E.g., considering the base case of an empty tree instead of a single node

What's the deal with iterators?

- Iterators allow observation of all data in the specified order
- The standard iterator() factory method and the for-each loop syntax will use inorder traversal
- Specify postorder or preorder with the getIterator() method



Implementation

- Create a ”snapshot” of the tree
  - Based on traversal order
- Store snapshot as a queue
- Perform iterator operations on the queue
  - next() -> ?
  - hasNext() -> ?
  - ~~remove()~~

Snapshot

```
public Iterator<T> getIterator(BSTInterface.Traversal orderType)
// Creates and returns an Iterator providing a traversal of a "snapshot"
// of the current tree in the order indicated by the argument.
// Supports Preorder, Postorder, and Inorder traversal.
{
  final LinkedQueue<T> infoQueue = new LinkedQueue<T>();
  if (orderType == BSTInterface.Traversal.Preorder)
    preOrder(root, infoQueue);
else
if (orderType == BSTInterface.Traversal.Inorder)
    inOrder(root, infoQueue);
else
if (orderType == BSTInterface.Traversal.Postorder)
    postOrder(root, infoQueue);
```

Iterator() anonymous class

```
return new Iterator<T>()
{
  public boolean hasNext()
  // Returns true if the iteration has more elements; otherwise returns false.
  {
    return !infoQueue.isEmpty();
  }

  public T next()
  // Returns the next element in the iteration.
  // Throws NoSuchElementException - if the iteration has no more elements
  {
    if (!hasNext())
      throw new IndexOutOfBoundsException("illegal invocation of next " +
                                          " in BinarySearchTree iterator.\n");
    return infoQueue.dequeue();
  }

  public void remove()
  // Throws UnsupportedOperationException.
  // Not supported. Removal from snapshot iteration is meaningless.
  {
    throw new UnsupportedOperationException("Unsupported remove attempted on "
                                            + "BinarySearchTree iterator.\n");
  }
};
```

Snapshot

```
public Iterator<T> getIterator(BSTInterface.Traversal orderType)
// Creates and returns an Iterator providing a traversal of a "snapshot"
// of the current tree in the order indicated by the argument.
// Supports Preorder, Postorder, and Inorder traversal.
{
  final LinkedQueue<T> infoQueue = new LinkedQueue<T>();
  if (orderType == BSTInterface.Traversal.Preorder)
    preOrder(root, infoQueue);
else
if (orderType == BSTInterface.Traversal.Inorder)
    inOrder(root, infoQueue);
else
if (orderType == BSTInterface.Traversal.Postorder)
    postOrder(root, infoQueue);
```

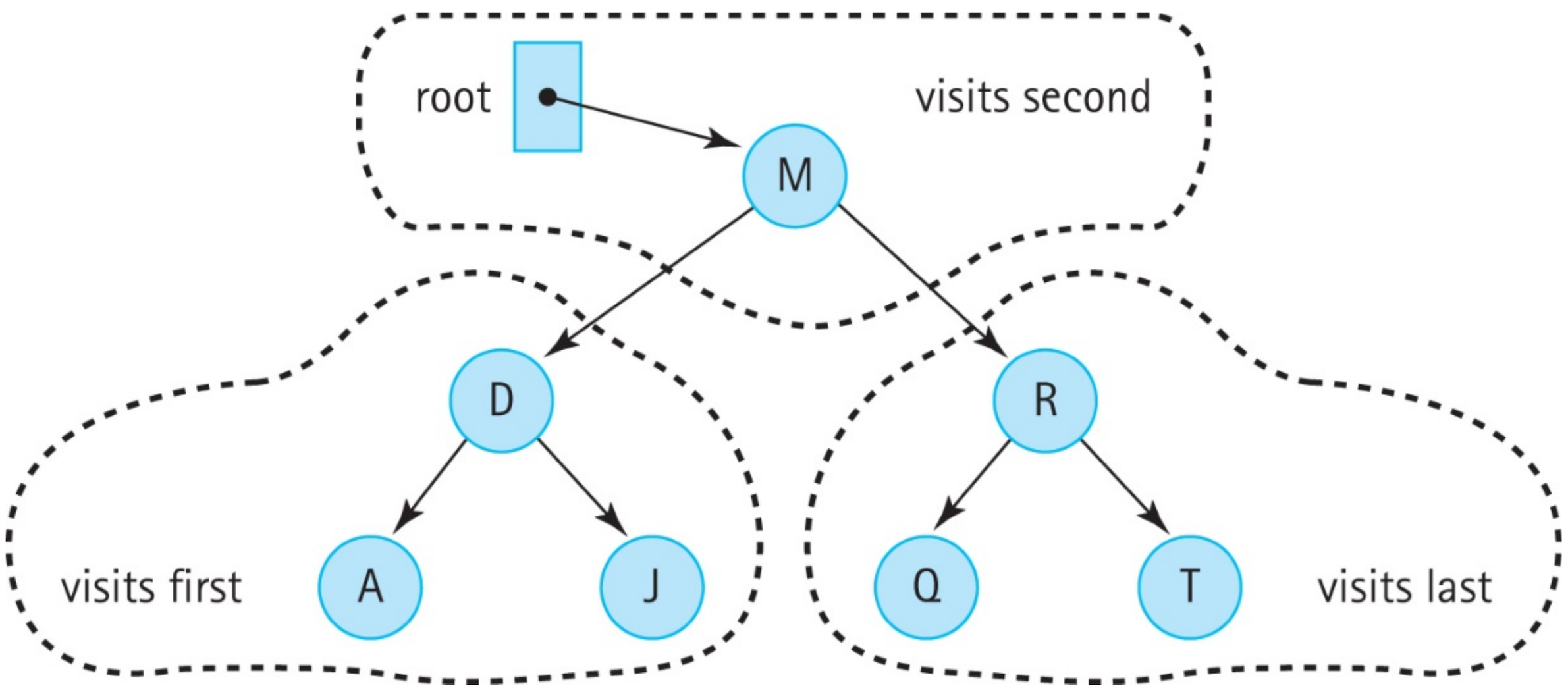
Snapshot

- Add all the node data objects to the queue in a specific order
- Traverse the tree in the specified order
- For each “visit”, enqueue() the data element

Snapshot Recursion

- Base case?
- Base case check?
- Recursive case?

Inorder Example



Inorder

```
private void inOrder(BSTNode<T> node, LinkedQueue<T> q)
// Enqueues the elements from the subtree rooted at node into q in inOrder.
{
  if (node != null)
  {
    inOrder(node.getLeft(), q);
    q.enqueue(node.getInfo());
    inOrder(node.getRight(), q);
  }
}
```

Preorder, Postorder

```
private void preOrder(BSTNode<T> node, LinkedQueue<T> q)
// Enqueues the elements from the subtree rooted at node into q in preOrder.
{
  if (node != null)
  {
    q.enqueue(node.getInfo());
    preOrder(node.getLeft(), q);
    preOrder(node.getRight(), q);
  }
}
```

```
private void postOrder(BSTNode<T> node, LinkedQueue<T> q)
// Enqueues the elements from the subtree rooted at node into q in postOrder.
{
  if (node != null)
  {
    postOrder(node.getLeft(), q);
    postOrder(node.getRight(), q);
    q.enqueue(node.getInfo());
  }
}
```

Code Samples

```
BinarySearchTree<Character> example = new BinarySearchTree<Character>();
Iterator<Character> iter;

example.add('P'); example.add('F'); example.add('S'); example.add('B');
example.add('H'); example.add('R'); example.add('Y'); example.add('G');
example.add('T'); example.add('Z'); example.add('W');
```

Code Samples

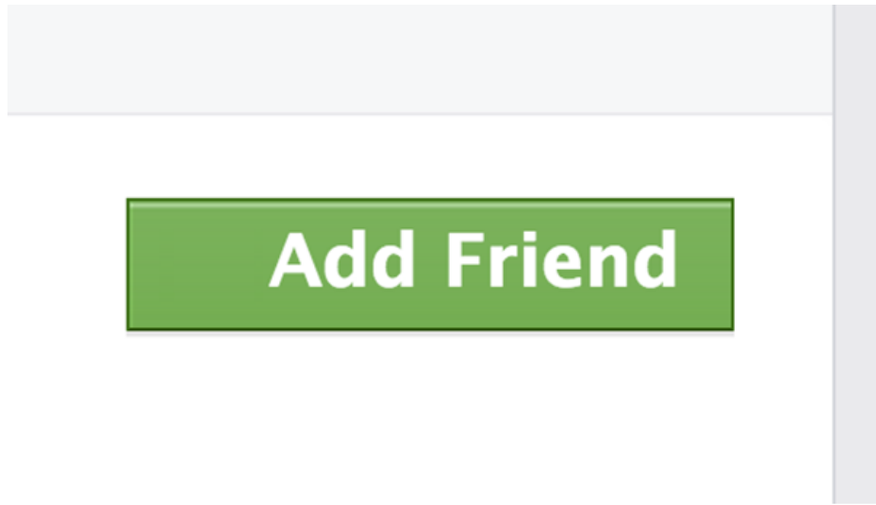
```
// Preorder
System.out.print("\nPreorder:  ");
iter = example.getIterator(BSTInterface.Traversal.Preorder);
while (iter.hasNext())
  System.out.print(iter.next());


// Inorder
System.out.print("Inorder:   ");
iter = example.getIterator(BSTInterface.Traversal.Inorder);
while (iter.hasNext())
  System.out.print(iter.next());


// Inorder again
System.out.print("\nInorder:   ");
for (Character ch: example)
   System.out.print(ch);
```

Add

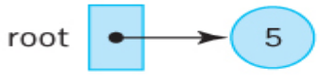
- We again use a recursive implementation for binary search trees
- Search for the next empty child along the path starting from the root and maintaining the binary search property
- Should the recursion return void or a pointer to a new node?



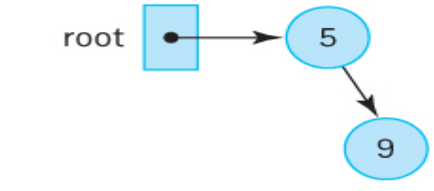
Add Illustration

(a) root

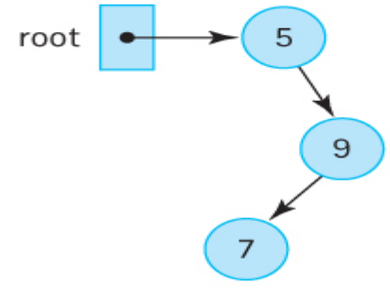
(b) add 5



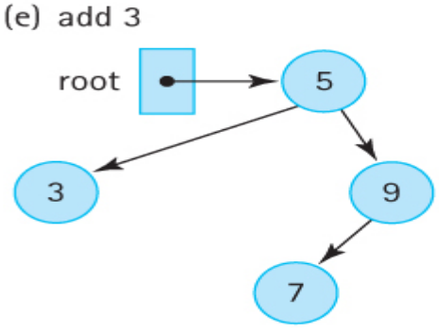
(c) add 9



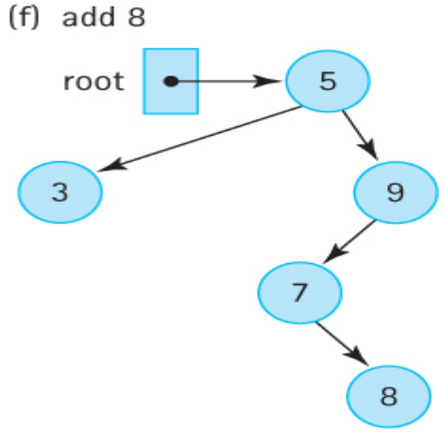
(d) add 7



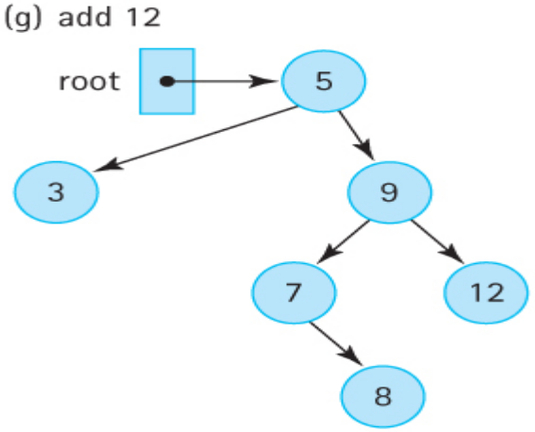
(e) add 3



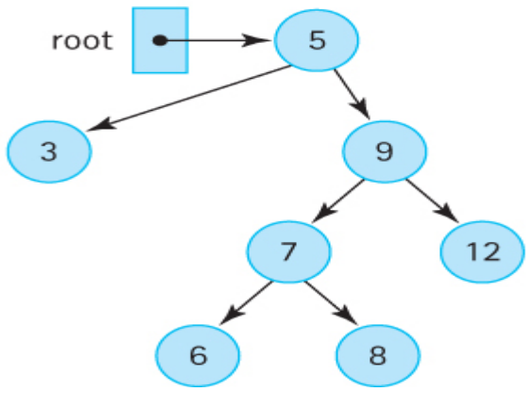
(f) add 8



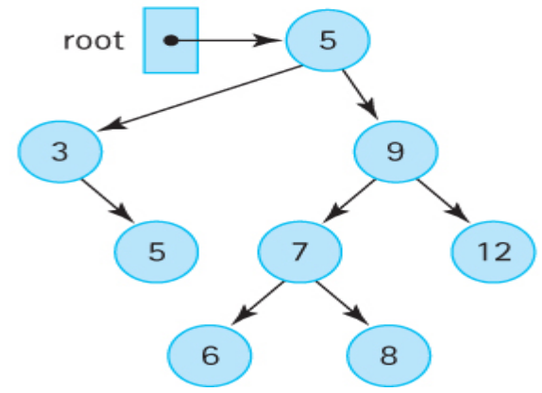
(g) add 12



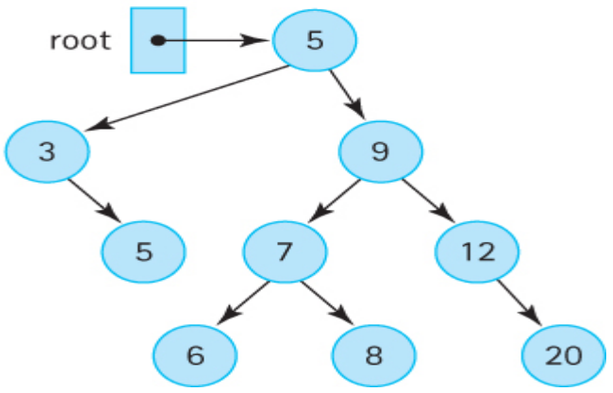
(h) add 6



(i) add 5



(j) add 20



Add code

```
private BSTNode<T> recAdd(T element, BSTNode<T> node)
// Adds element to tree rooted at node; tree retains its BST property.
{
  if (node == null)
    // Addition place found
    node = new BSTNode<T>(element);
  else if (element.compareTo(node.getInfo()) <= 0)
    node.setLeft(recAdd(element, node.getLeft()));     // Add in left subtree
  else
    node.setRight(recAdd(element, node.getRight()));   // Add in right subtree
  return tree;
}

public boolean add (T element)
// Adds element to this BST. The tree retains its BST property.
{
  root = recAdd(element, root);
  return true;
}
```

Practice

- Draw the tree resulting from the following add operations: 8, 1, 11, 6, 7, 9, 5, 4

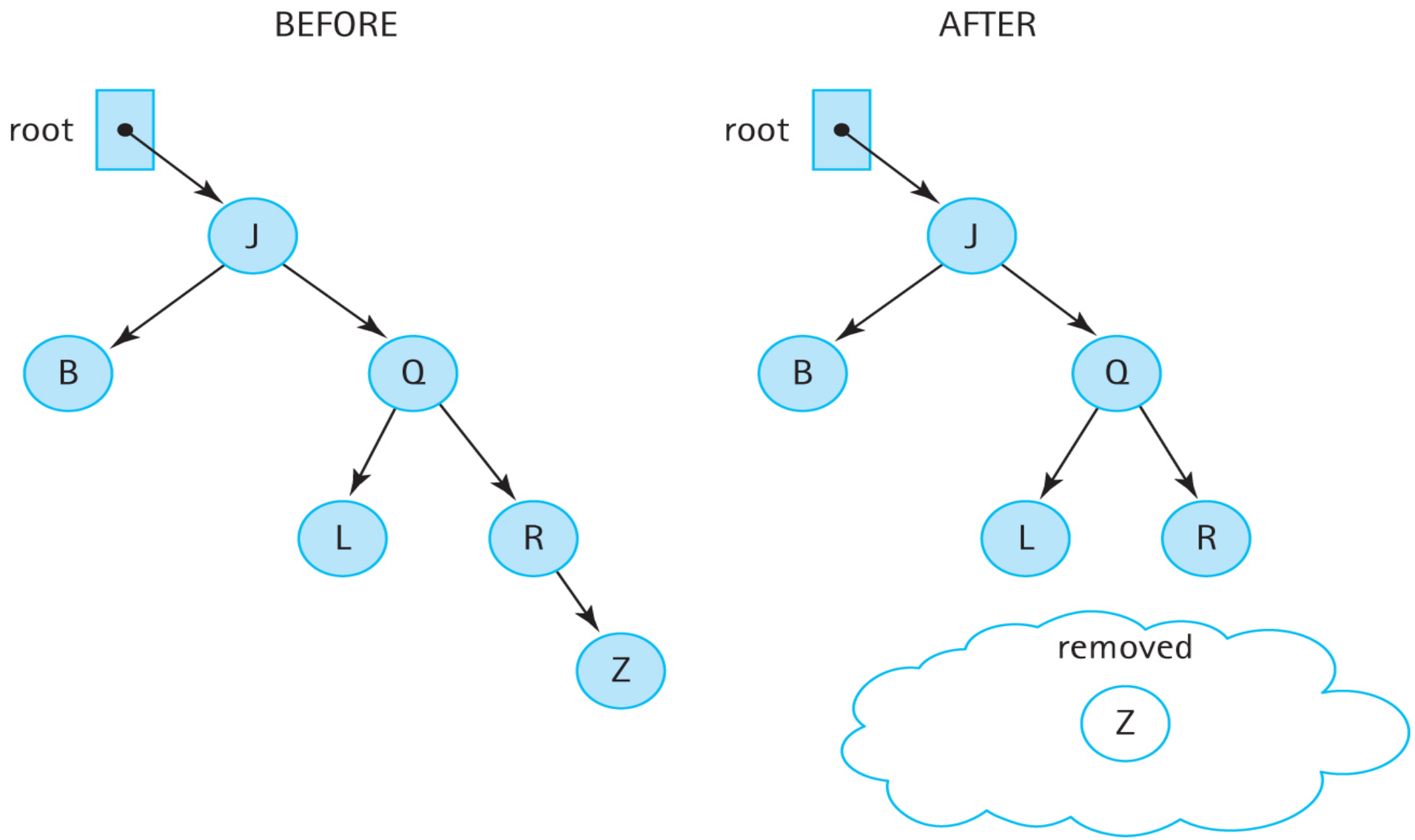
Remove

- Recursively search for the target object
- Re-structure the tree to eliminate the object
- Set the class variable found depending on whether or not the object was in the tree

Cases

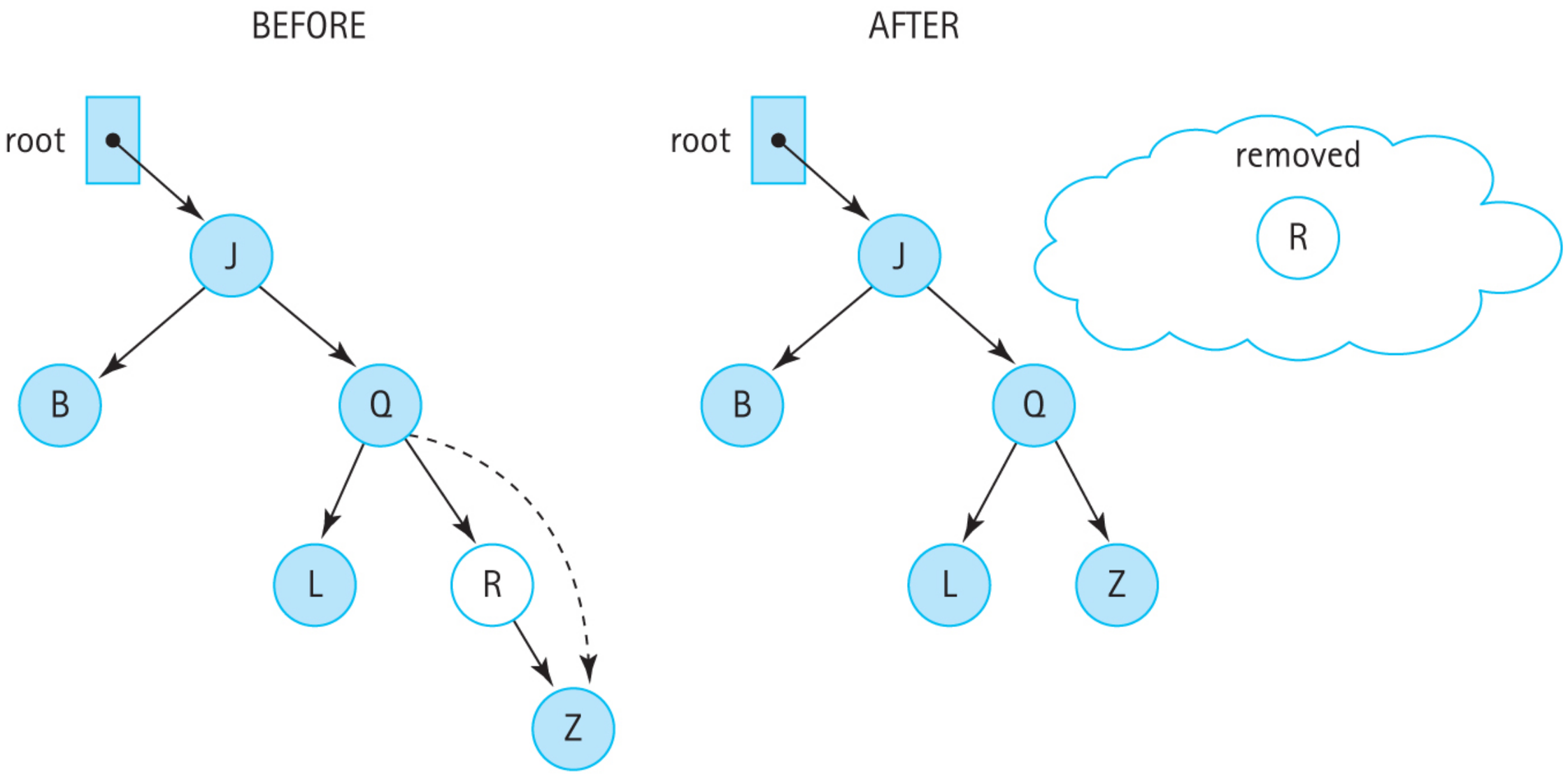
- The node containing the target has:
  - No children
  - One child
  - Two children

Example: no children



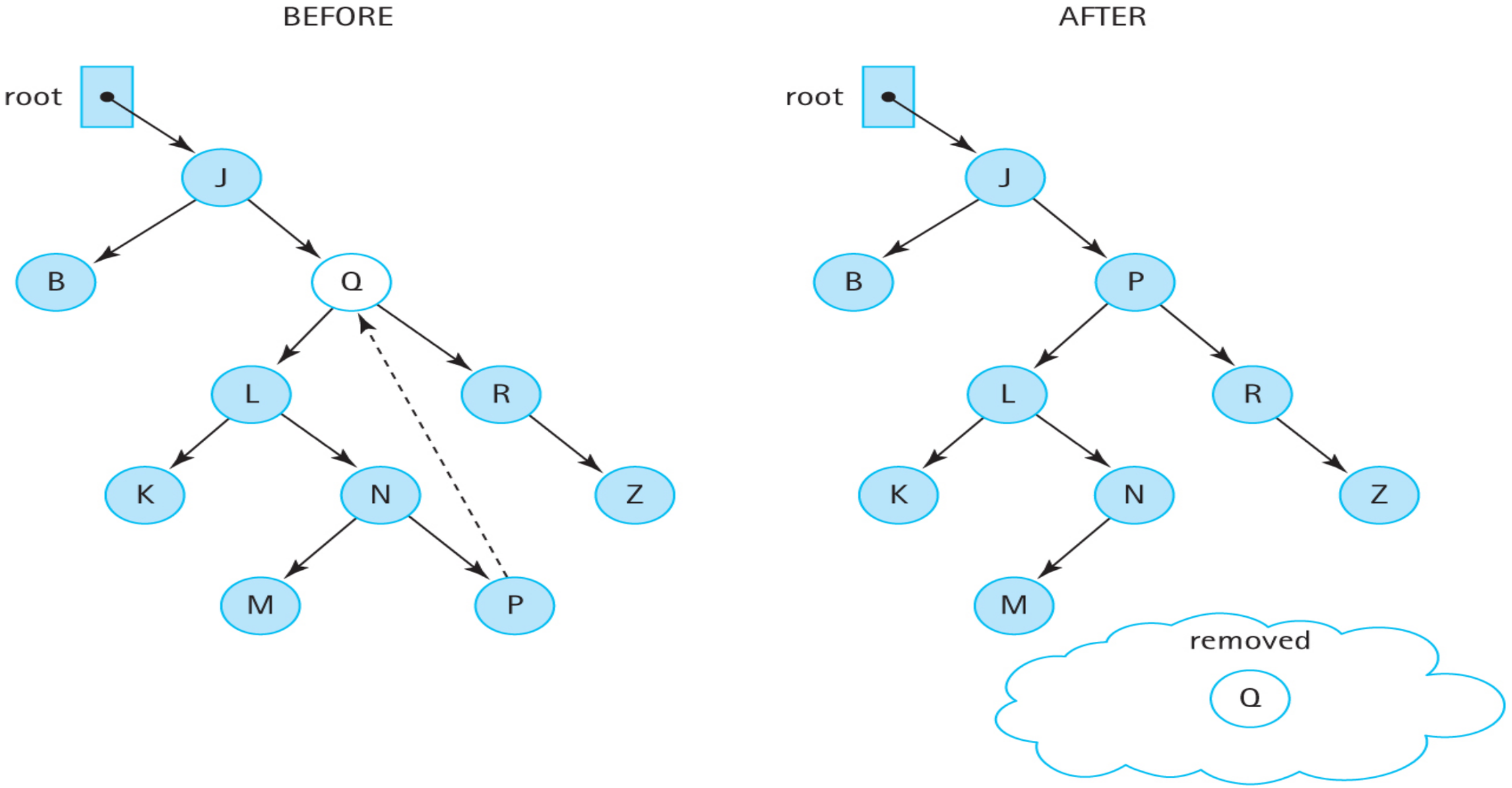
Remove the node containing Z

Example: one child



Remove the node containing R

Example: two children



Remove the node containing Q

Implementation in four parts

- Helper method
- Search method
- Remove node method
- Find max method

Helper

```
public boolean remove (T target)
// Removes a node with info i from tree such that
// comp.compare(target,i) == 0 and returns true;
// if no such node exists, returns false.
{
  root = recRemove(target, root);
  return found;
}
```

Search

```
private BSTNode<T> recRemove(T target, BSTNode<T> node)
// Removes element with info i from tree rooted at node such that
// comp.compare(target, i) == 0 and returns true;
// if no such node exists, returns false.
{
   if (node == null)
      found = false;
   else if (comp.compare(target, node.getInfo()) < 0)
      node.setLeft(recRemove(target, node.getLeft()));
   else if (comp.compare(target, node.getInfo()) > 0)
      node.setRight(recRemove(target, node.getRight()));
   else
   {
      node = removeNode(node);
      found = true;
   }
   return node;
}
```

Remove

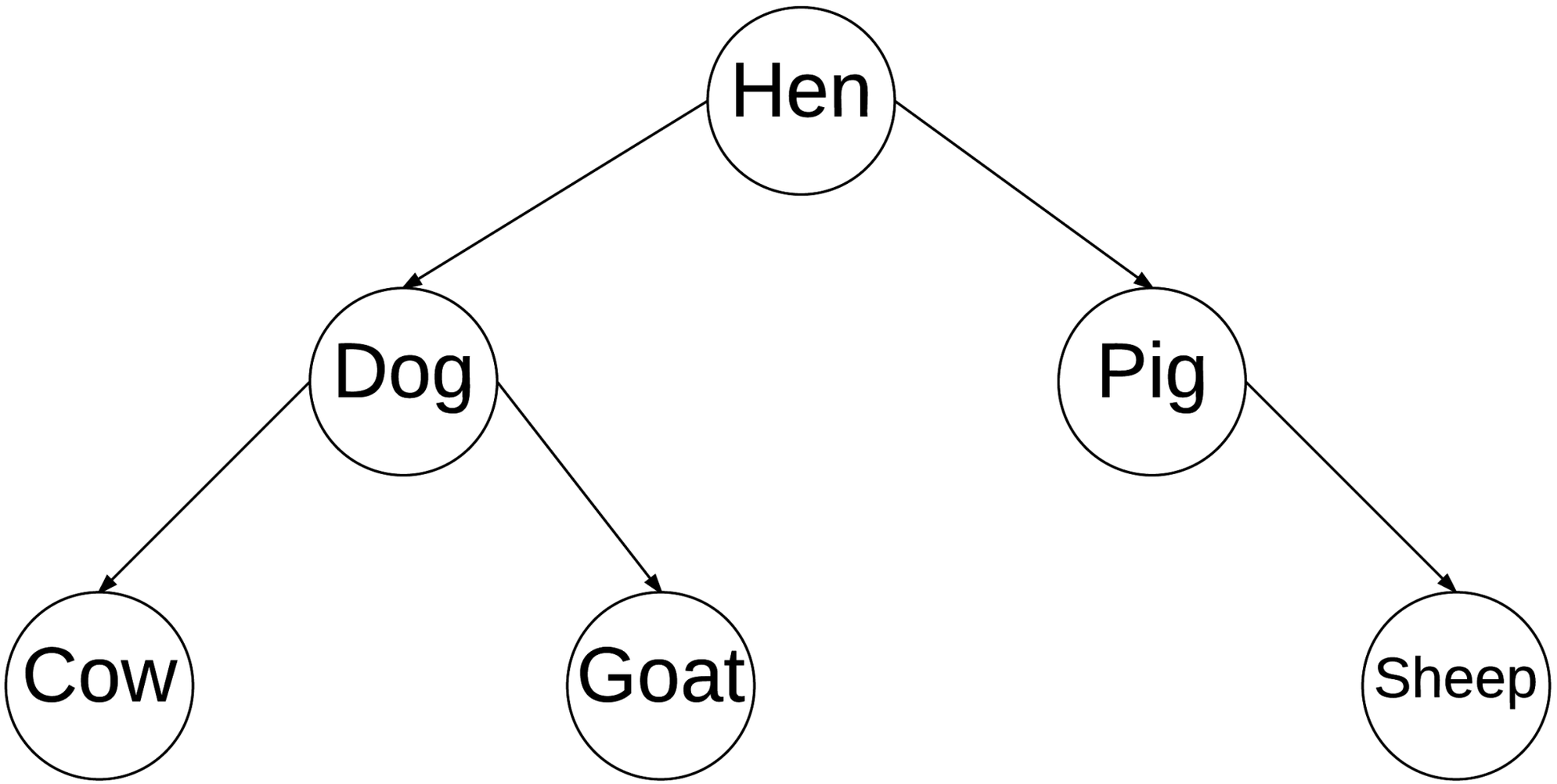
```
private BSTNode<T> removeNode(BSTNode<T> node)
// Removes the information at node from the tree.
{
  T data;
  if (node.getLeft() == null)
    return node.getRight();
  else if (node.getRight() == null)
    return node.getLeft();
  else
  {
    data = getPredecessor(node.getLeft());
    node.setInfo(data);
    node.setLeft(recRemove(data, node.getLeft()));
    return node;
  }
}
```

Find max

```
private T getPredecessor(BSTNode<T> subtree)
// Returns the information held in the rightmost node of subtree
{
  BSTNode temp = subtree;
  while (temp.getRight() != null)
    temp = temp.getRight();
  return temp.getInfo();
}
```

Practice

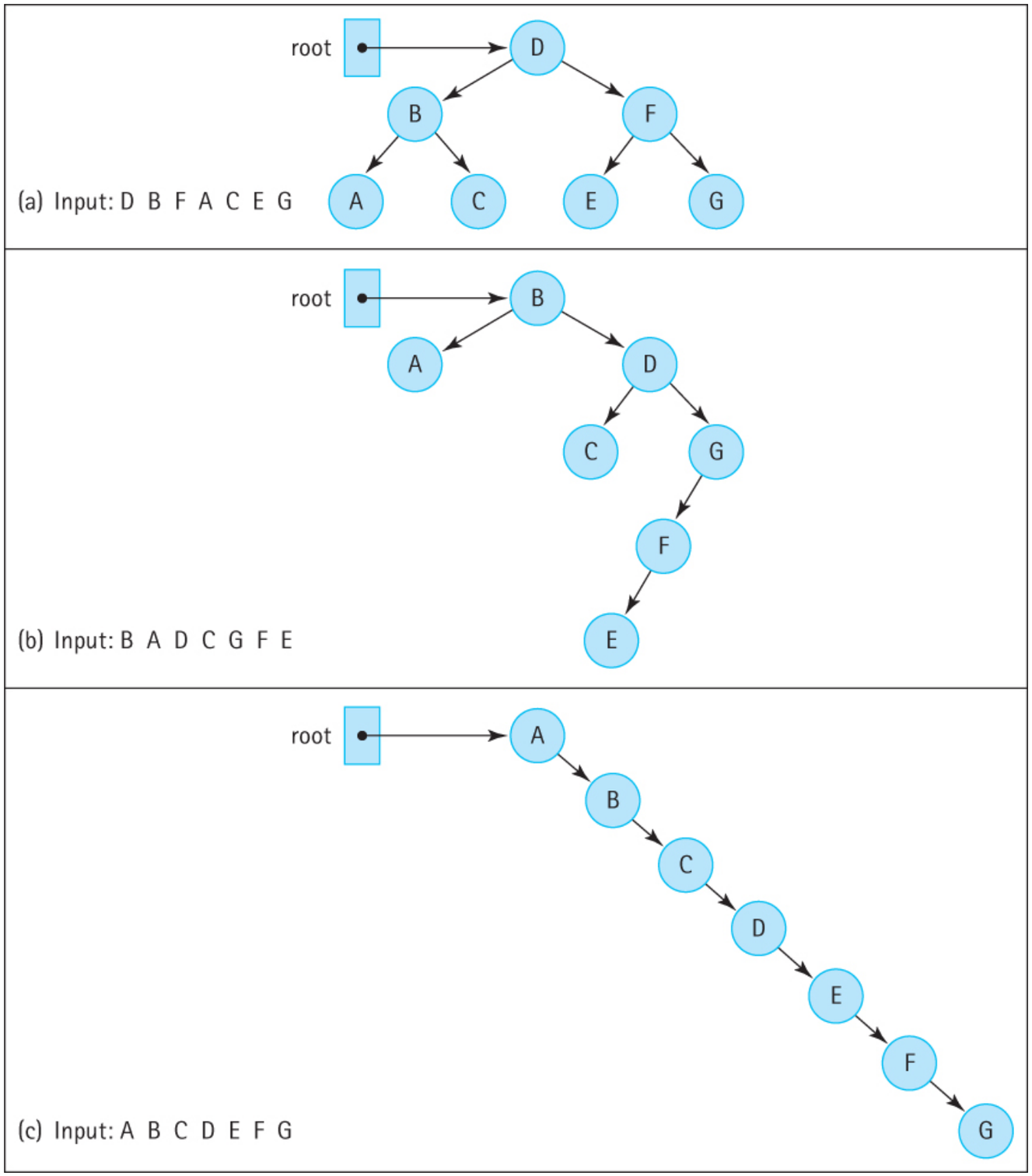
- Remove the following nodes in order:
  - Sheep, Hen, Dog, Cow



Efficiency

- Get, contains: O(??)
- Add, remove: O(??)
- What is the single factor that can drastically affect these results?

Balance


root
D
B
F
A
C
E
G
(a) Input: D B F A C E G
root
B
A
D
C
G
F
E
(b) Input: B A D C G F E
root
A
B
C
D
E
F
G
(c) Input: A B C D E F G

Recap

- Binary search trees allow for (on average) logarithmic order of growth for ALL collection methods
- Preorder, inorder, and postorder traversals can be implemented recursively and reordered with very few modifications
- Add and remove require some careful decision-making for how to maintain efficiency AND correct binary search structure

Chapter 1: Java

- Java class structure
- Inheritance
- Direct vs indirect addressing (memory layout)
- Big-O notation

Chapter 2: Stacks

- Abstraction and interfaces
- LIFO data structure
- Linked list implementation structure
- Applications: expression matching and postfix evaluator

Chapter 3: Recursion

- Recursive definitions and algorithms
- Binary search
- Recursive processing of linked lists
- Towers of Hanoi and t-square fractals
- How/when to avoid recursion

Chapter 4: Queues

- FIFO data structure
- Array and linked list tradeoffs
- Queue variants
- Application: average wait time

Chapter 5: Collections

- Content-based access
- Defining object equals and compareTo
- Sorted vs unsorted collections
- Collection variants

Chapter 6: Lists

- Linear-relation ordering (with content AND index-based access)
- Java Iterators
- List variants
- Application: card deck

Chapter 7: Binary Search Trees

- A nonlinear linked structure
- Traversal orderings
- Recursive method implementation
- (Typically) logarithmic order of growth

Next Time...

- Final exam
  - Saturday @ 1:30 PM
  - Closed book/notes
  - Calculators only

- Check the course webpage for practice problems
- Peer Tutors
  - http://www.csc.villanova.edu/help/