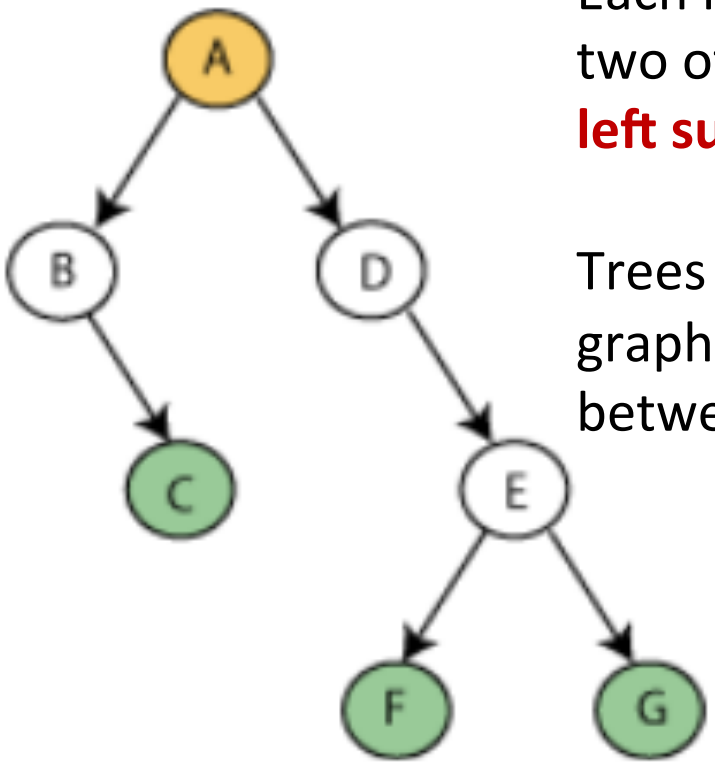
Binary Trees

Terminology, implementation, traversals, binary search trees (BSTs), BST insertions, BST searches, generic BSTs

Binary tree definition

A **binary tree** is a hierarchical data structure that consists of **nodes**.



Each node contains an element and two other trees, which are called the **left subtree** and the **right subtree**.

Trees are **directed acyclic connected** graphs. There is exactly one path between any pair of nodes.

Recursive binary tree definition

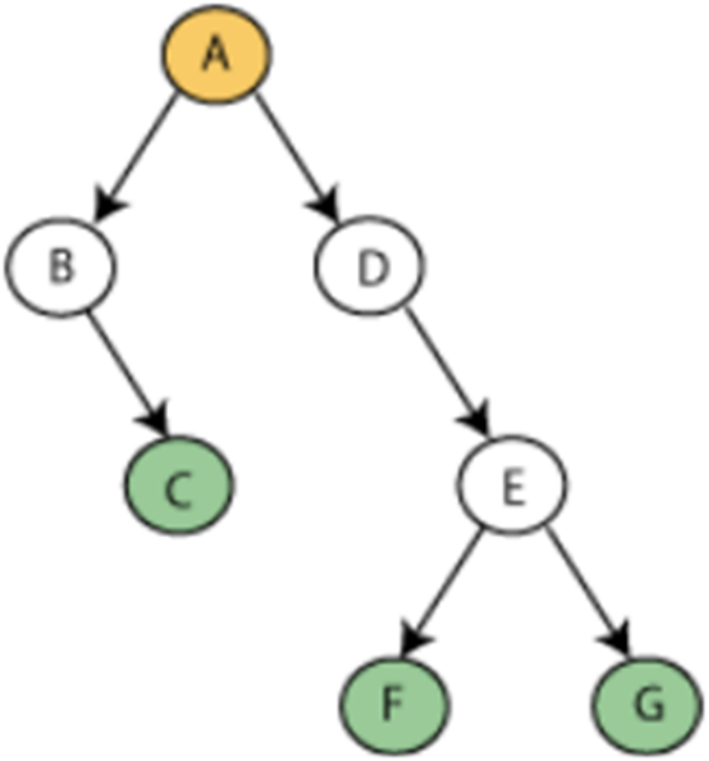
A binary tree:

is empty

-or-

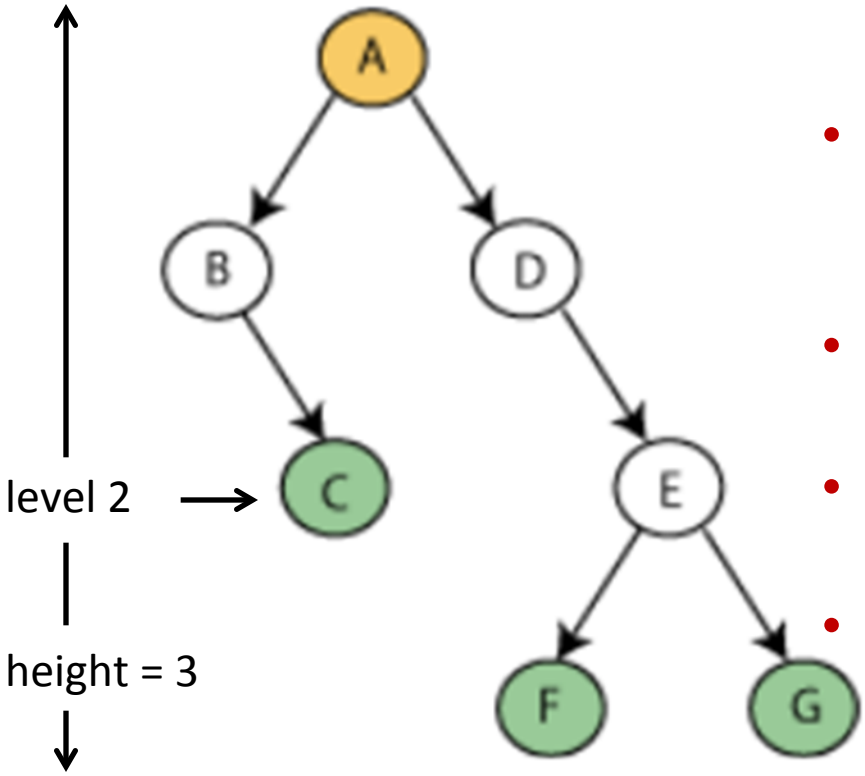
consists of an element and a left subtree and a right subtree, both of which are binary trees.

Terminology



- **Root node** = node at the top of the tree.
- Each non-root node has a **parent**, which is the node directly above.
- Parent nodes have **children**. A child can be a **left child** or a **right child**.
- **Leaf node** = node with no children.
- **Branch** node = any internal node (neither a leaf nor the root).
- Two nodes with a common parent are **siblings**.

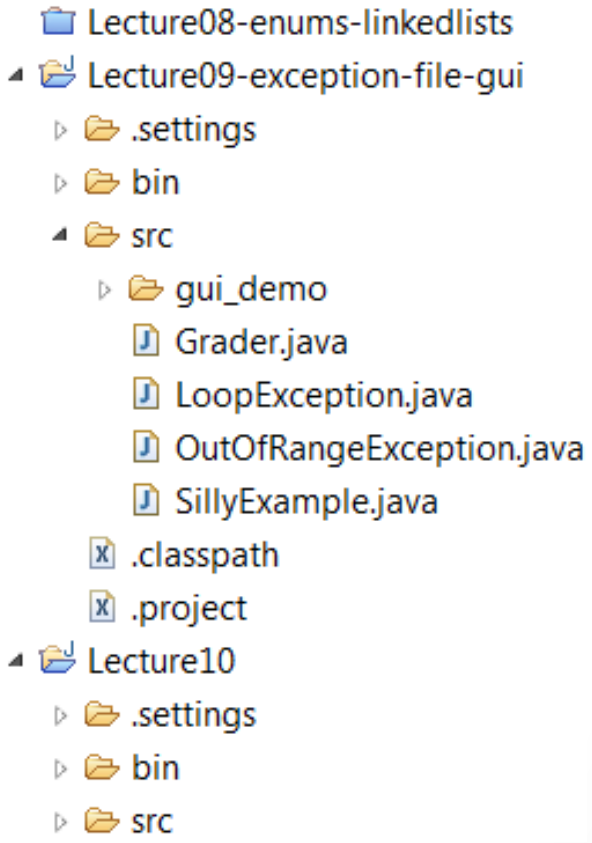
Terminology (cont)



- Any node in a subtree of a given node is a **descendant**. The given node is an **ancestor**.
- **Depth of a node** = length of path from root to the node. (Root has depth 0.)
- **Height of a tree** = maximum depth.
- **Level** = set of all nodes at a given depth.
- **Full binary tree** = every node has exactly 0 or 2 children.

Tree usage

- Trees are used for LOTS of concepts in computer science:
  - Any hierarchical structure
  - Directory structures
  - Parsing algorithms
  - Routing algorithms
  - Some searching algorithms
    - Binary search trees
  - Some sorting algorithms
    - Red black trees
    - Balanced binary trees
  - And so on …



Binary tree implementation

Start with a Tree inner class (equivalent to the Node inner class for a linked list). Our example code has int data.

```
private class Tree {
    public int data;
    public Tree left;
    public Tree right;

    public Tree(int data, Tree left, Tree right) {
       this.data = data;
       this.left = left;
       this.right = right;
}
```

Tree implementation (cont 1)

Wrap the Tree node class inside an outer class.

```
public class BinaryTree {
  private Tree root;

  public BinaryTree() {
    root = null;
  }
  private class Tree {
    public int data;
    public Tree left;
    public Tree right;

    public Tree(int data, Tree left, Tree right) {
      this.data = data;
      this.left = left;
      this.right = right;
  }
```

Binary tree traversals

- **Traversing** a data structure means "visiting" all of the elements in the data structure, looking at each element exactly once.
- There are 3 standard binary tree traversals
  - **Preorder**. Visit the root first, then the left subtree, then the right subtree.
  - **Inorder**. Visit the left subtree first, then the root, then the right subtree.
  - **Postorder**. Visit the left subtree first, then the right subtree, then the root.
- For our examples, “visiting” will simply print out the node’s data value.

Preorder traversal

Root – left – right

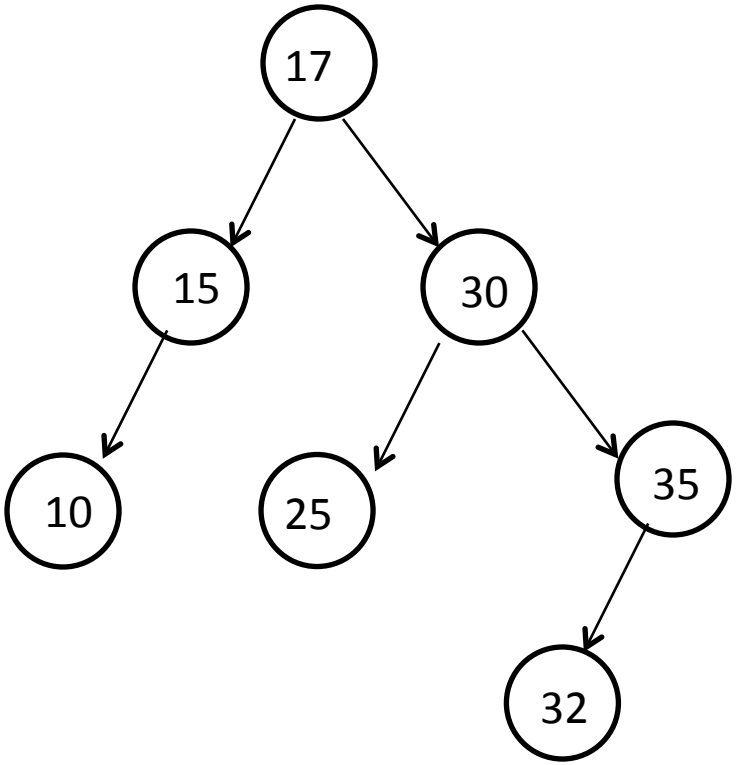
```
if (root != null)
    visit root
    traverse root.leftSubtree
    traverse root.rightSubtree
```

Actual code:

```
public void visit(Tree n) {
   System.out.print(n.data + " ");
}
public void preorder(){
    preorder(root);
}
private void preorder(Tree n) {
   if (n != null) {
      visit(n);
      preorder(n.left);
      preorder(n.right);
   }
}
```

Preorder traversal trace

17 15 10 30 25 35 32



Method template

Use the following template to write recursive methods on binary trees (including traversals).

```
public void method( ) {
  method(root);
}

private void method (Tree n) {
  // code goes here
}
```

Additional parameters to the public method?

- Expand the private method parameter list to include them.
- Make them actual parameters to the call to the private method.

Inorder traversal

Left – root – right

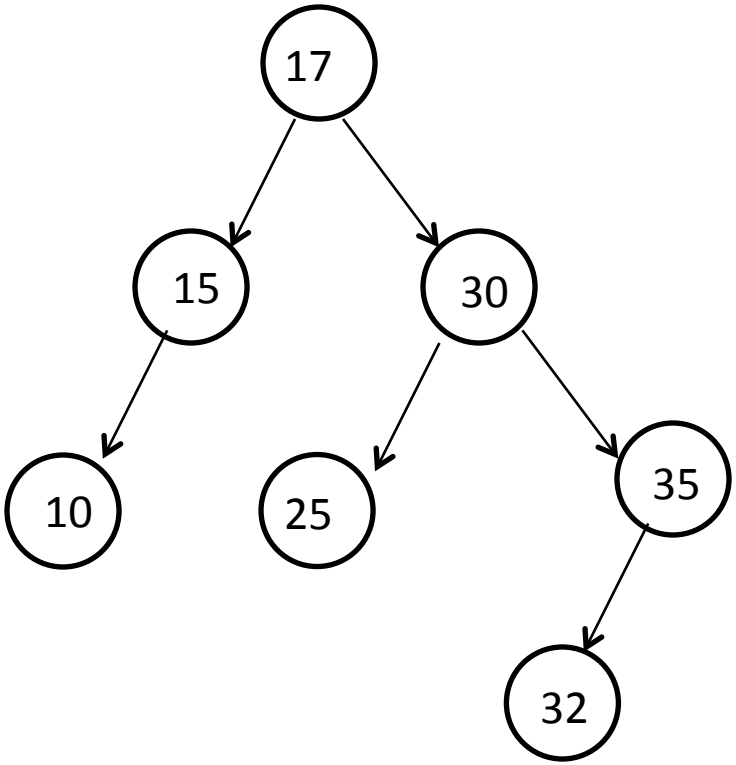
```
if (root != null)
   traverse root.leftSubtree
   visit root
   traverse root.rightSubtree
```

Actual code:

```
public void visit(Tree n) {
   System.out.print(n.data + " ");
}
public void inorder(){
    inorder(root);
}
private void inorder(Tree n) {
   if (n != null) {
      inorder(n.left);
      visit(n);
      inorder(n.right);
   }
}
```

Inorder traversal trace

10 15 17 25 30 32 35



Postorder traversal

Left – right– root

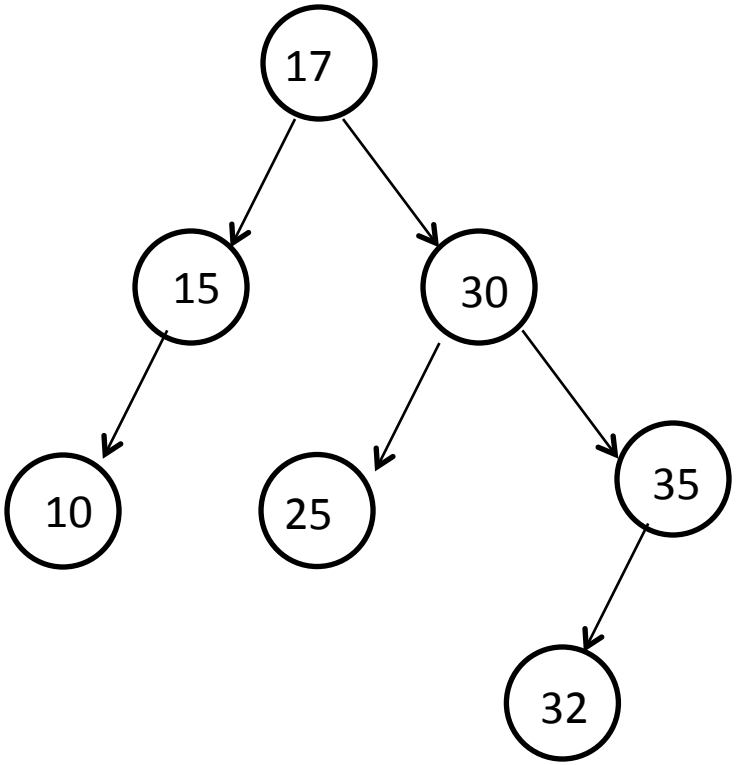
```
if (root != null)
   traverse root.leftSubtree
   traverse root.rightSubtree
   visit root
```

Actual code:

```
public void visit(Tree n) {
   System.out.print(n.data + " ");
}
public void postorder(){
   postorder(root);
}
private void postorder(Tree n) {
   if (n != null) {
      postorder(n.left);
      postorder(n.right);
      visit(n);
   }
}
```

Postorder traversal trace

10 15 25 32 35 30 17

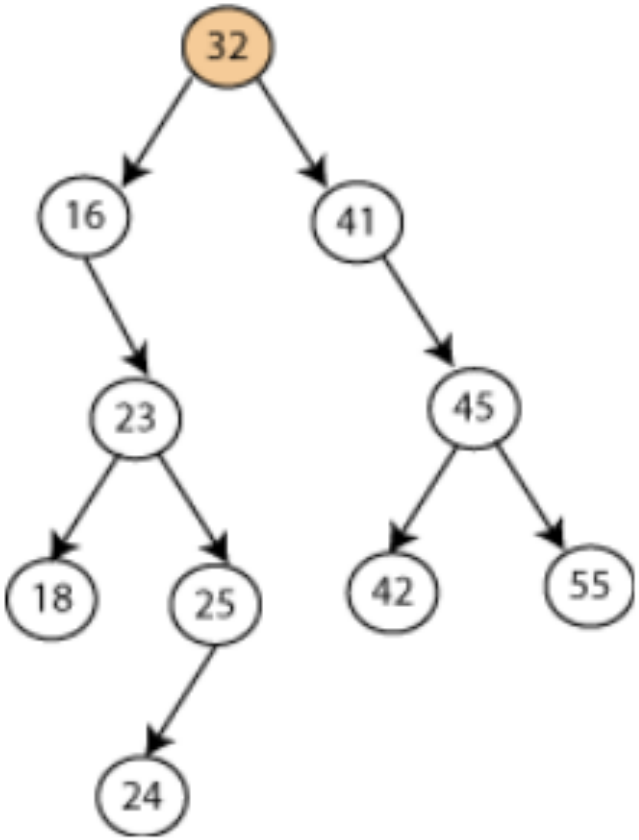


Binary search trees

- A **binary search tree** is a binary tree in which:
  - The elements are comparable
  - The tree is empty

    -- or --
    - the root element is greater than all elements of the left subtree
    - the root element is less than all elements of the right subtree
    - the right and left subtrees of the root are also binary search trees
- Note that binary search trees cannot have duplicate elements.

Binary search tree example



- Inorder traversal visits nodes in ascending order.
- Inorder traversal for this tree:
  16 18 23 24 25 32 41 42 45 55

Binary search tree behaviors

Essential binary tree behaviors:

- **create.** Create an empty binary tree.
- **search(element-key).** Search for an element in a tree, typically on some key field of the item. For key searches, the elements must be ordered according to their keys.
- **insert(element).** Insert a new element in a tree. The element must be inserted in a correct position so the tree remains as a binary search tree.
- **traverse().** Travel through the tree to visit each element. An inorder traversal visits the elements in sorted order.

Code basics

Start with the same binary tree structure as before.

```
public class BinarySearchTree {
   private Tree root;

   public BinarySearchTree() {
      root = null;
   }
   private class Tree {
      public int data;
      public Tree left;
      public Tree right;

      public Tree(int data, Tree left, Tree right) {
         this.data = data;
         this.left = left;
         this.right = right;
   }
```

Insertions keep tree integrity

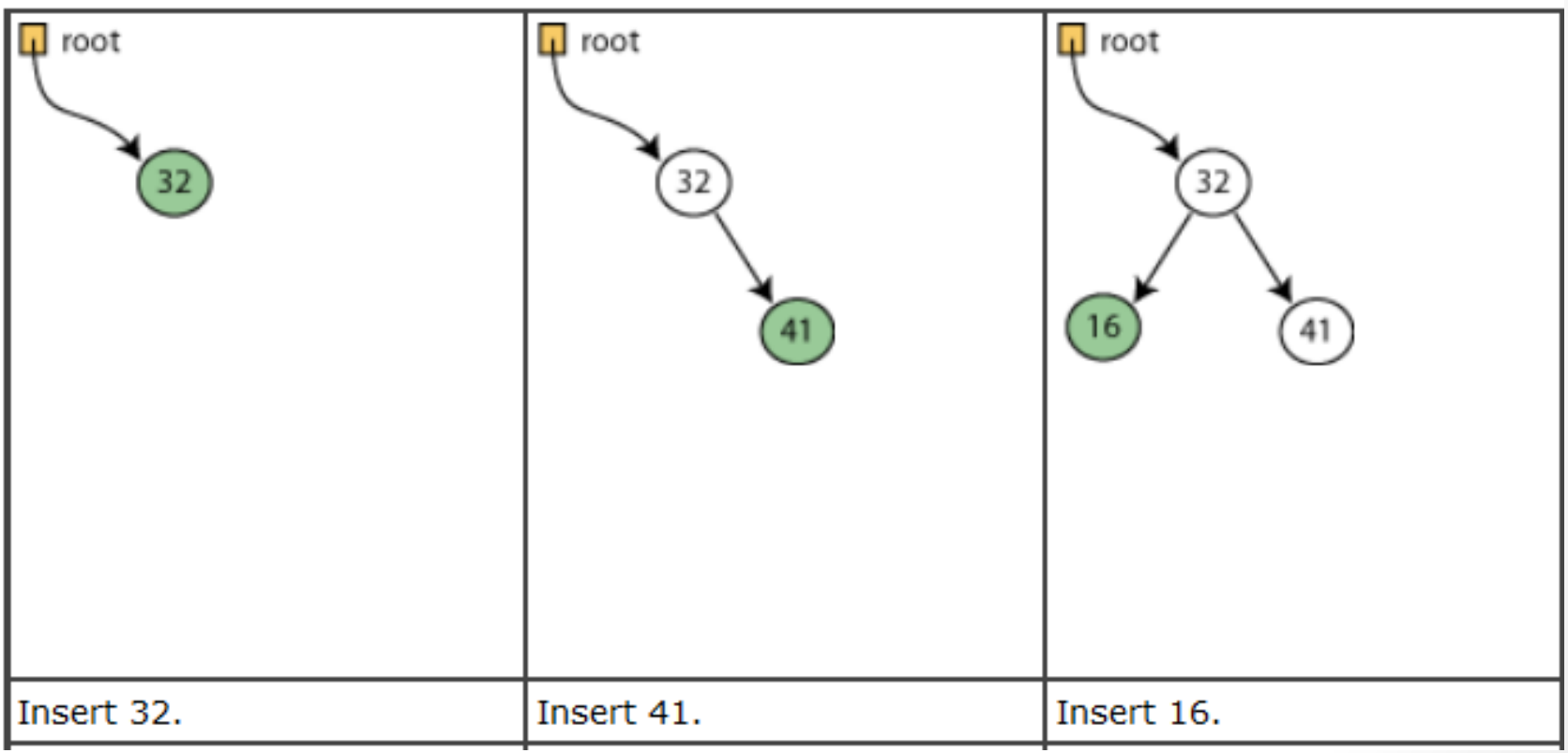
To insert an item into a binary search tree:

- If the root is null, make the item the root.
- Else, compare the item to the root element.:
  - If the item is less than the root, insert it into the left subtree.
  - If the item is greater than the root, insert it into the right subtree.
  - If the item equals the root, do not insert it (no duplicates allowed) .

Insertion under this algorithm is always at a leaf.

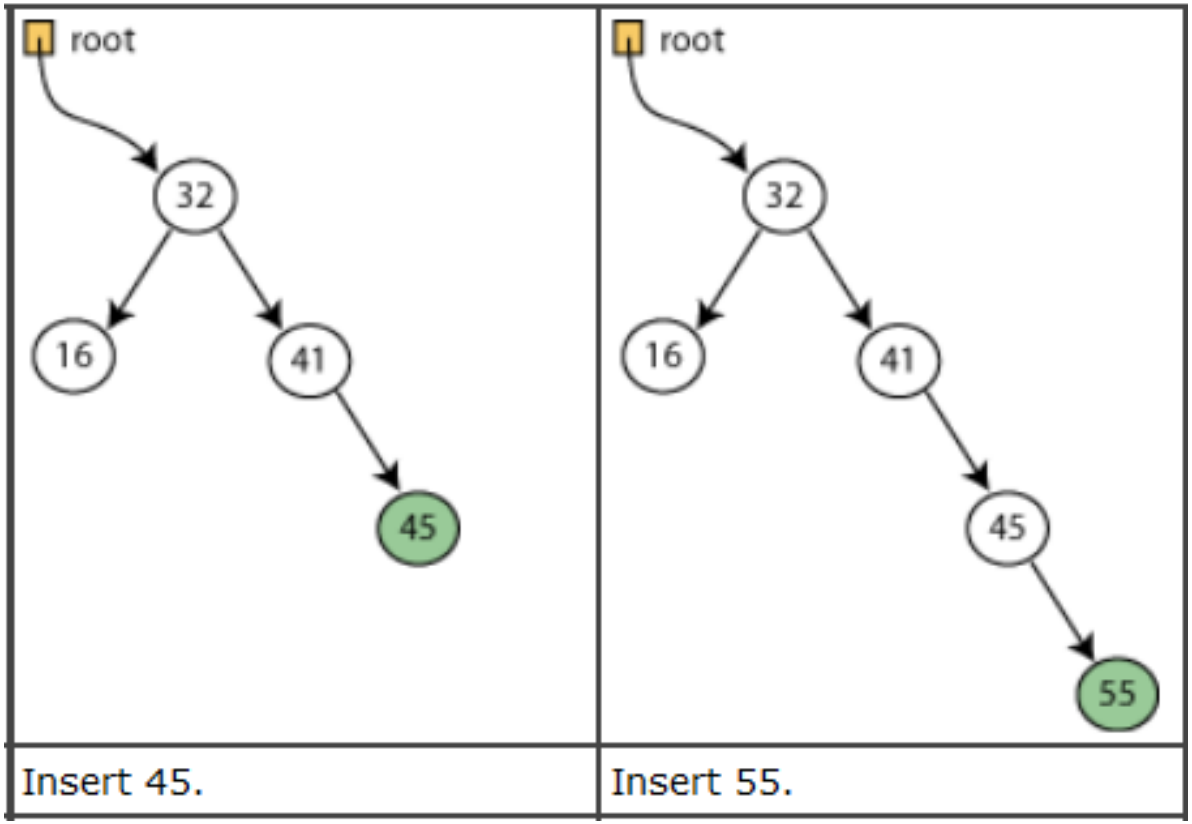
Example insertion/construction

- Construct a binary search tree of this sequence of integers:
  32, 41, 16, 45, 55, 23, 25, 24, 18, 42



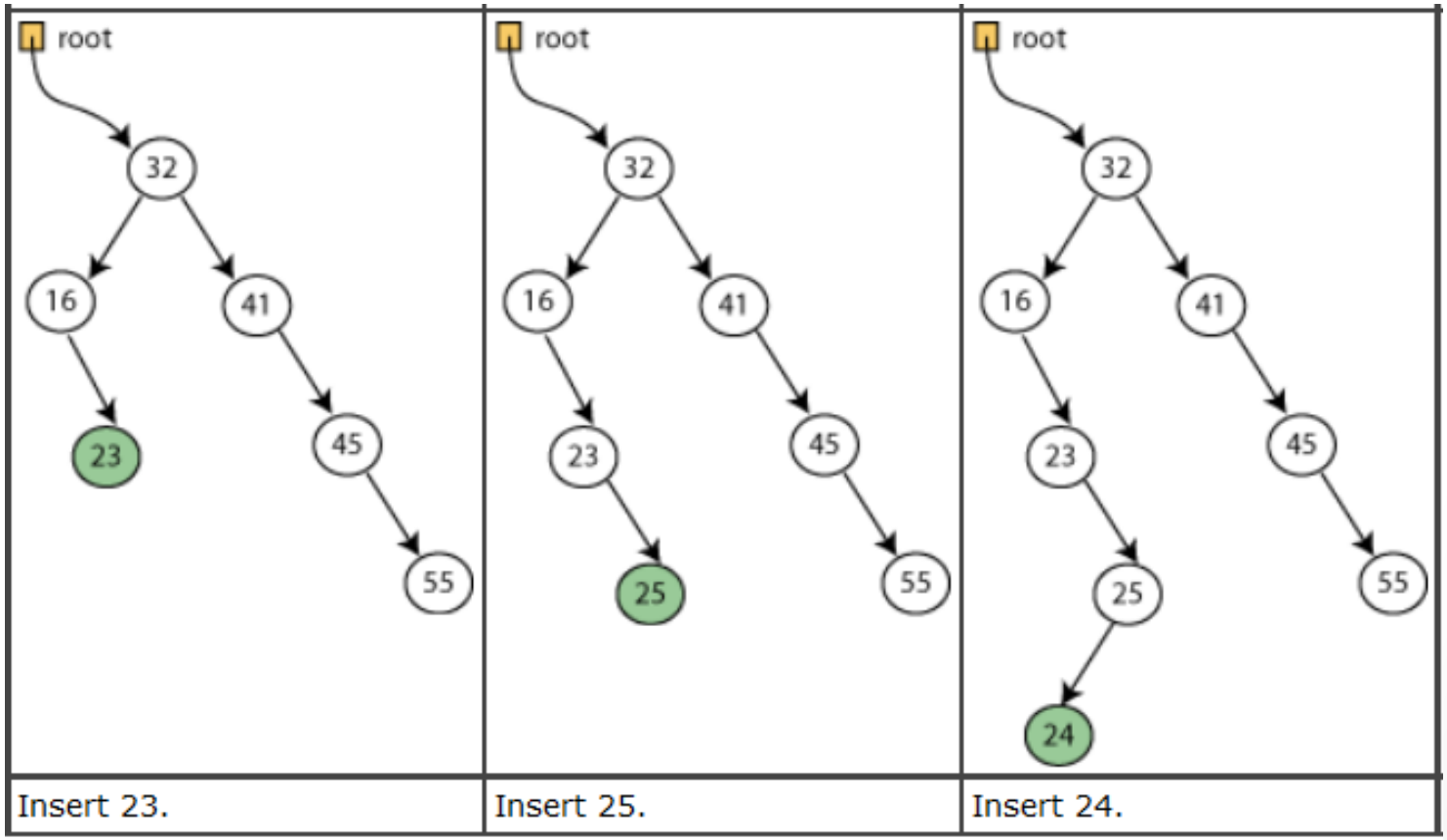
Example (cont 1)

- Construct a binary search tree of this sequence of integers: 32, 41, 16, 45, 55, 23, 25, 24, 18, 42



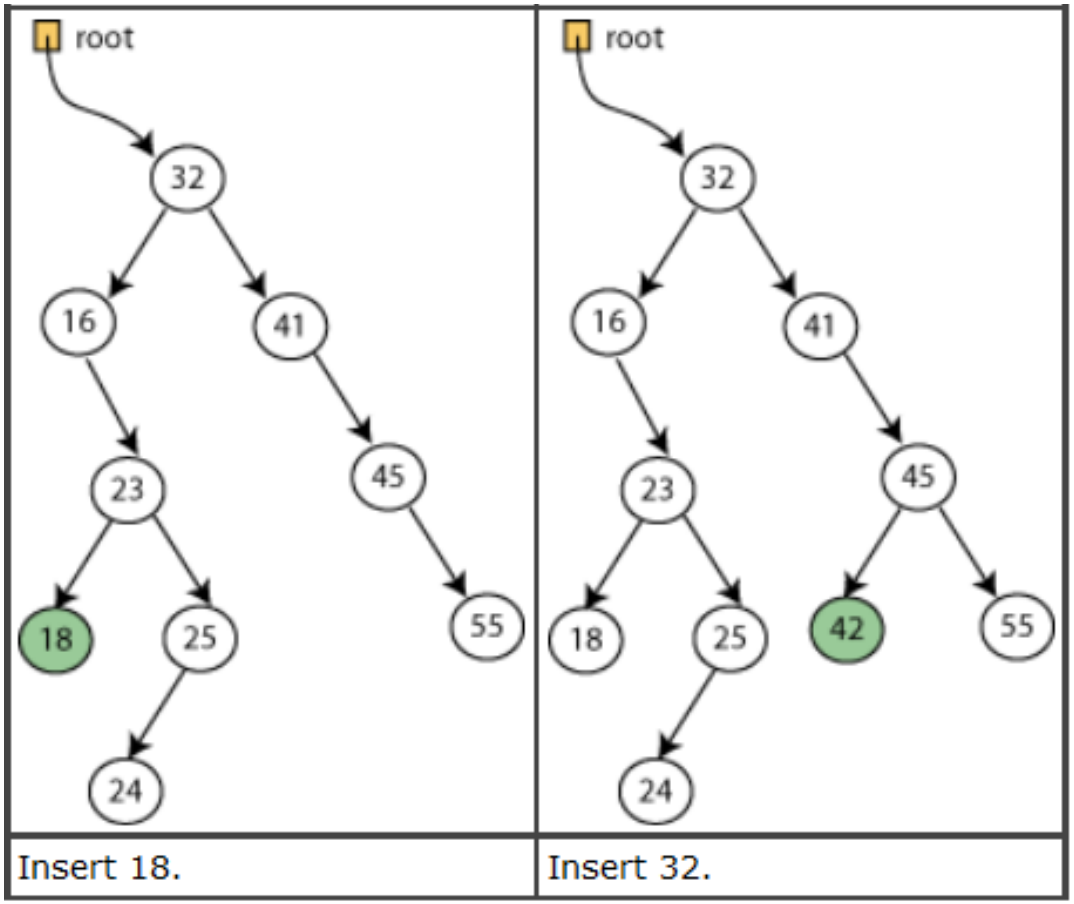
Example (cont 2)

- Construct a binary search tree of this sequence of integers: 32, 41, 16, 45, 55, 23, 25, 24, 18, 42



Example (cont 3)

- Construct a binary search tree of this sequence of integers: 32, 41, 16, 45, 55, 23, 25, 24, 18, 42



Insertion code – part 1

BinarySearchTree delegates responsibility for insertions to Tree.

```
public class BinarySearchTree {
   private Tree root;
   // constructor etc here

    public void insert(int x) {
       if (root == null)
          root = new Tree(x, null, null);
       else
          root.insert(x);
    }
    private class Tree {
   // variables, constructor etc here

      public void insert(int x) {
      // code to insert goes here
   }
```

Insertion code – part 2

Tree.insert() code.

```
private class Tree {
   // Tree data, constructor here
   public void insert(int x) {
      if (x < data) {
         if (left == null)
            left = new Tree(x, null, null);
         else
            left.insert(x);
      }
      else if (x > data) {
         if (right == null)
            right = new Tree(x, null, null);
         else
            right.insert(x);
      }
   }
```

Searching in a BST

Basic search algorithm:

```
if the tree is empty
    return false (or null, if searching on keys)
else if the root matches the item searched for
    return true  (or the object, if searching on keys)
else if the root is less than the item searched for
    search in the right subtree
else
    search in the left subtree
```

BST search code – setup

Use the same BinarySearchTree code as before, except the data type is Bicycle (instead of int).

```
private class Tree {
   public Bicycle data;
   public Tree left;
   public Tree right;

   public Tree(Bicycle data, Tree left, Tree right) {
        this.data = data;
        this.left = left;
        this.right = right;
   }
   public Bicycle search(int target) { // etc
```

BST search code

The code for searching is in two places:

1. Outside the inner class. Client calls are to this method.
2. Inside the inner class. Inner class code chases through the tree.

The code below is the method outside the inner class.

```
public Bicycle search(int key) {
  if (root == null)
     return null;
  return root.search(key); //Call to inner class search
}
```

Code on the following slide is inside the inner tree class.

BST search code

```
public Bicycle search(int target) { // Inside Tree
   if (data.getId() == target)
      return data;
   else if (data.getId() > target){
      if (left != null)
         return left.search(target);
      else
         return null;
   }
   else {
      if (right != null)
         return right.search(target);
      else
         return null;
   }
}
```

Making BSTs generic

The tree code is easily changed to generic as long as the tree elements are comparable. The declaration for the BinarySearchTree class would be thus:

```
public class BinarySearchTree<E extends Comparable<E>>
```

Instead of using < or >, use the compareTo() method, which is declared in Comparable.

The remainder of the code is almost identical to the BinarySearchTree with Bicycle data. Only use the type E instead of the type Bicycle.

Generic BST

```
public class BinarySearchTree<E extends Comparable<E>>{
   private Tree root = null;

   private class Tree {
      public E data;
      public Tree left;
      public Tree right;

      // Tree constructor
      public Tree(E data, Tree left, Tree right) {
         this.data = data;
         this.left = left;
         this.right = right;
      }
```

Generic BST (continued)

```
// inside Tree definition
public void insert(E item) {
    if (item.compareTo(data) < 0) {
       if (left == null)
          left = new Tree(item, null, null);
       else
          left.insert(item);
    }
    else {
       if (right == null)
          right = new Tree(item, null, null);
       else
          right.insert(item);;
    }
 }
```

BST issues

- Efficient searches.
  - A **balanced** binary search tree is one where both its left and right subtrees are balanced and their heights differ by 0 or 1.
  - Searching in a BST is very efficient if the tree is balanced – O(log n). But search can be O(n) for unbalanced trees.
- Removing an element from a BST.
  - If the element doesn't have 2 children, it's easy (lift the subtree up one level).
  - If the element has 2 children, replace it with the leftmost child of the right subtree.