

CMSC 132: Object-Oriented Programming II



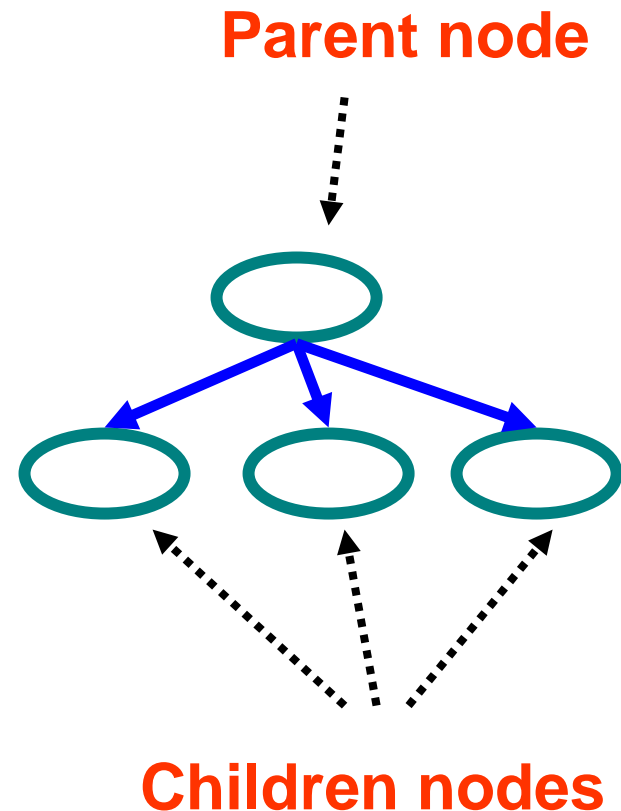
Trees & Binary Search Trees

Department of Computer Science

University of Maryland, College Park

Trees

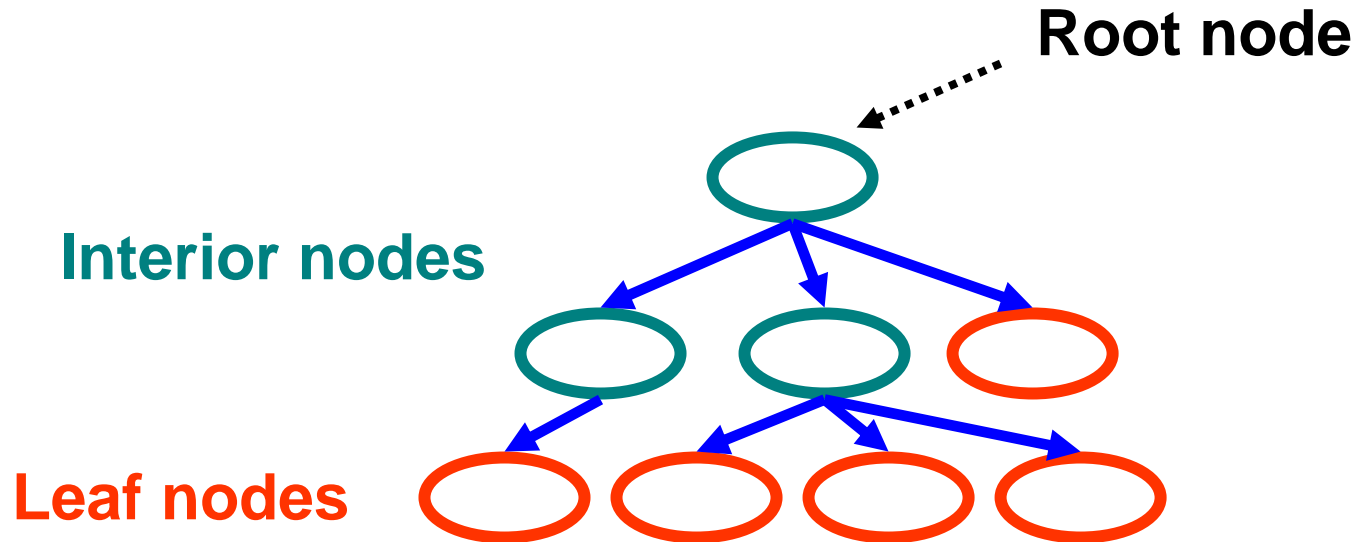
- Trees are hierarchical data structures
 - One-to-many relationship between elements
- Tree node / element
 - Contains data
 - Referred to by only 1 (**parent**) node
 - Contains links to any number of (**children**) nodes



Trees

■ Terminology

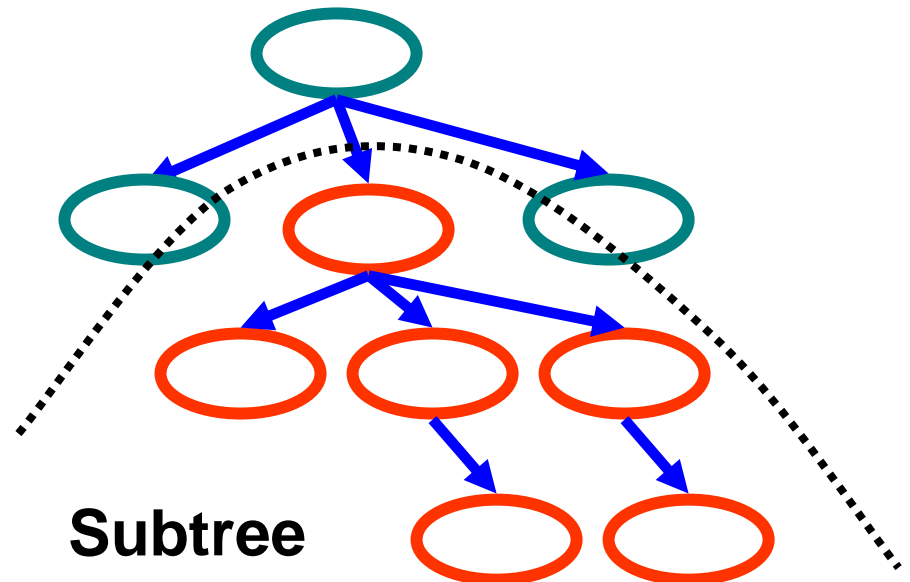
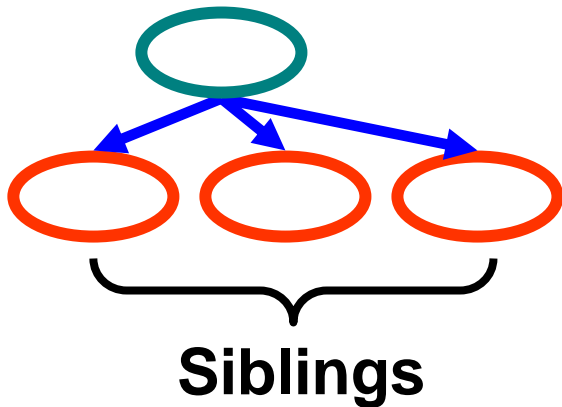
- Root \Rightarrow node with no parent
- Leaf \Rightarrow all nodes with no children
- Interior \Rightarrow all nodes with children



Trees

■ Terminology

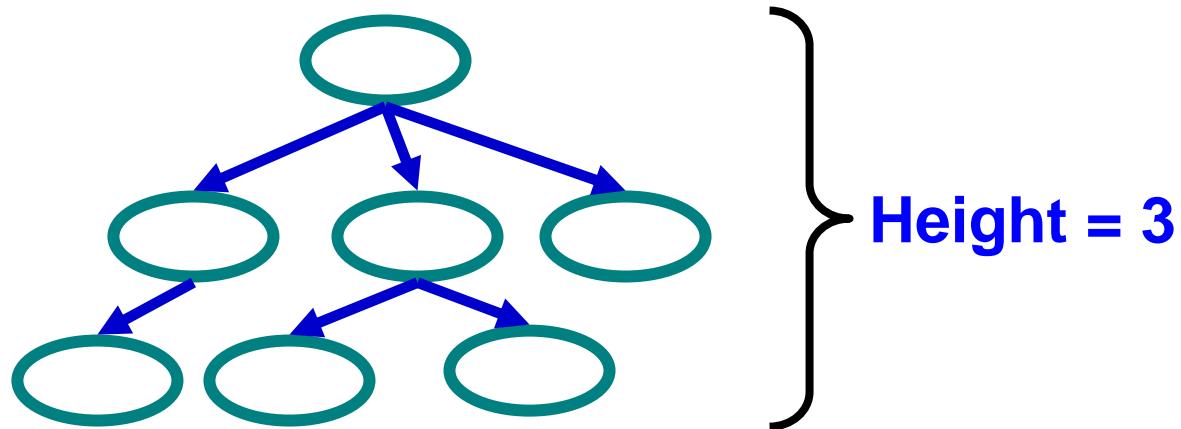
- Sibling \Rightarrow node with same parent
- Descendent \Rightarrow children nodes & their descendants
- Subtree \Rightarrow portion of tree that is a tree by itself
 \Rightarrow a node and its descendants



Trees

■ Terminology

- **Level** \Rightarrow is a measure of a node's distance from root
- **Definition of level**
 - If node is the root of the tree, its level is 1
 - Else, the node's level is 1 + its parent's level
- **Height (depth)** \Rightarrow max level of any node in tree

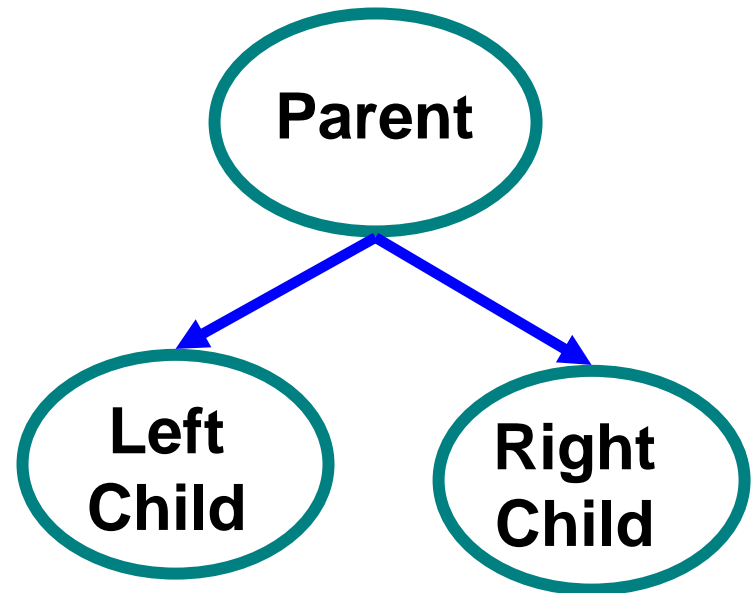
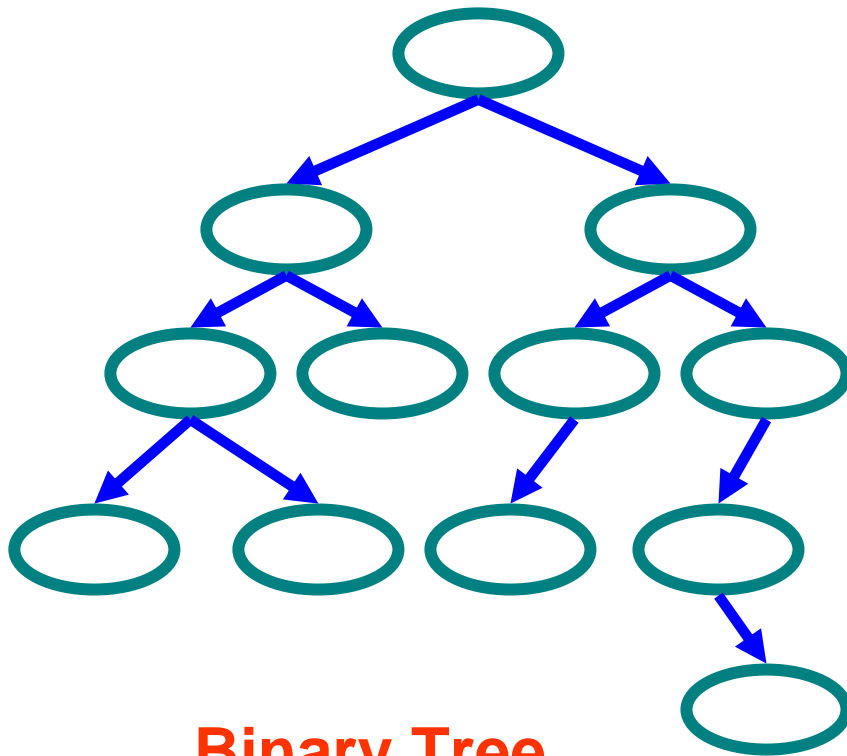


Binary Trees

■ Binary tree

■ Tree with 0–2 children per node

■ Left & right child / subtree



Tree Traversal

- Often we want to

1. Find all nodes in tree
2. Determine their relationship

- Can do this by

1. Walking through the tree in a prescribed order
2. Visiting the nodes as they are encountered

- Process is called **tree traversal**

Tree Traversal

■ Goal

- Visit every node in binary tree

■ Approaches

■ Depth first

■ Preorder \Rightarrow parent before children

■ Inorder \Rightarrow left child, parent, right child

■ Postorder \Rightarrow children before parent

■ Breadth first \Rightarrow closer nodes first

Tree Traversal Methods

■ Pre-order

1. Visit **node** // **first**
2. Recursively visit left subtree
3. Recursively visit right subtree

■ In-order

1. Recursively visit left subtree
2. Visit **node** // **second**
3. Recursively right subtree

■ Post-order

1. Recursively visit left subtree
2. Recursively visit right subtree
3. Visit **node** // **last**

Tree Traversal Methods

■ Breadth-first

```
BFS(Node n) {
    Queue Q = new Queue();
    Q.enqueue(n);           // insert node into Q
    while ( !Q.empty()) {
        n = Q.dequeue();   // remove next node
        if ( !n.isEmpty()) {
            visit(n);      // visit node
            Q.enqueue(n.Left()); // insert left subtree in Q
            Q.enqueue(n.Right()); // insert right subtree in Q
        }
    }
}
```

Tree Traversal Examples

■ Pre-order (prefix)

■ $+ \times 2 3 / 8 4$

■ In-order (infix)

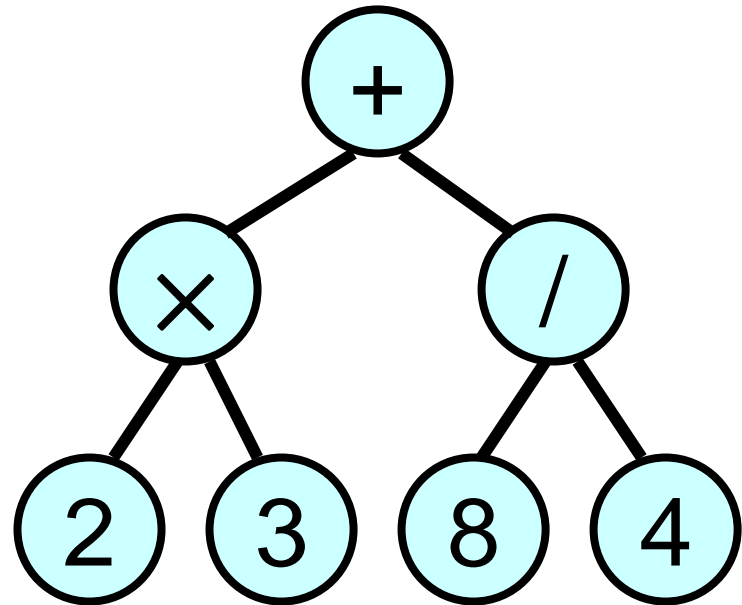
■ $2 \times 3 + 8 / 4$

■ Post-order (postfix)

■ $2 3 \times 8 4 / +$

■ Breadth-first

■ $+ \times / 2 3 8 4$



Expression tree

Tree Traversal Examples

■ Pre-order

■ 44, 17, 32, 78,
50, 48, 62, 88

■ In-order

■ 17, 32, 44, 48,
50, 62, 78, 88

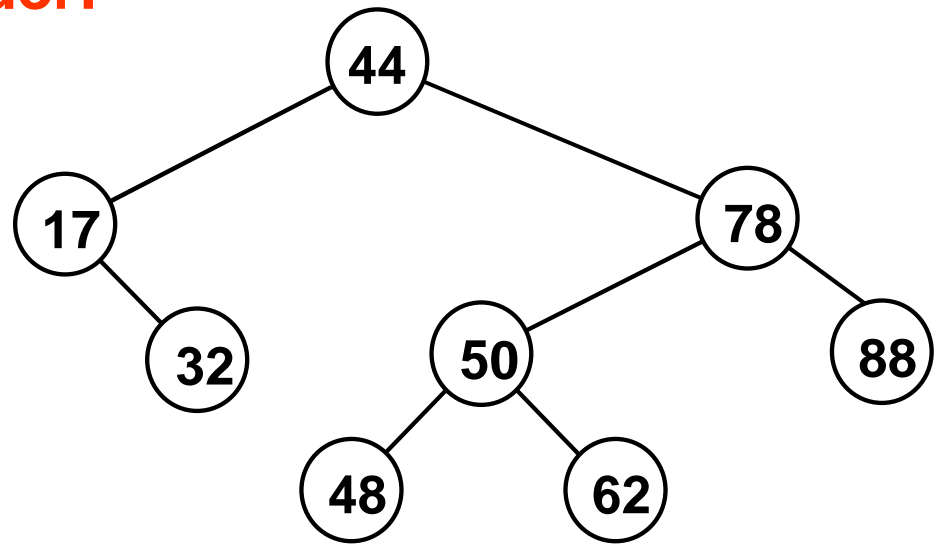
■ Post-order

■ 32, 17, 48, 62,
50, 88, 78, 44

■ Breadth-first

■ 44, 17, 78, 32,
50, 88, 48, 62

Sorted
order!

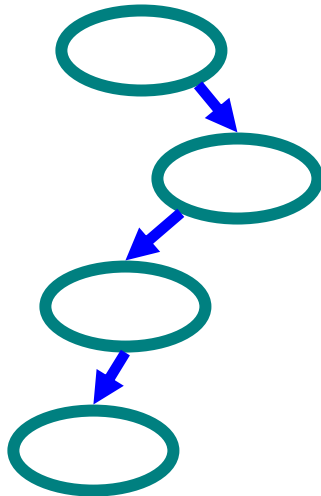


Binary search tree

Types of Binary Trees

■ Degenerate

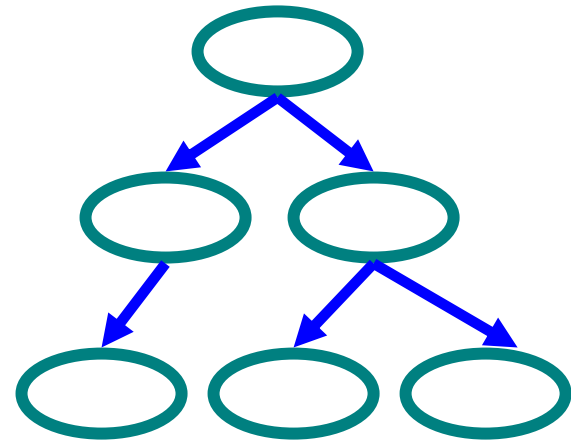
- Mostly 1 child / node
- Height = $O(n)$
- Similar to linear list



**Degenerate
binary tree**

■ Balanced

- Mostly 2 child / node
- Height = $O(\log(n))$
- Useful for searches



**Balanced
binary tree**

Binary Search Trees

■ Key property

■ Value at node

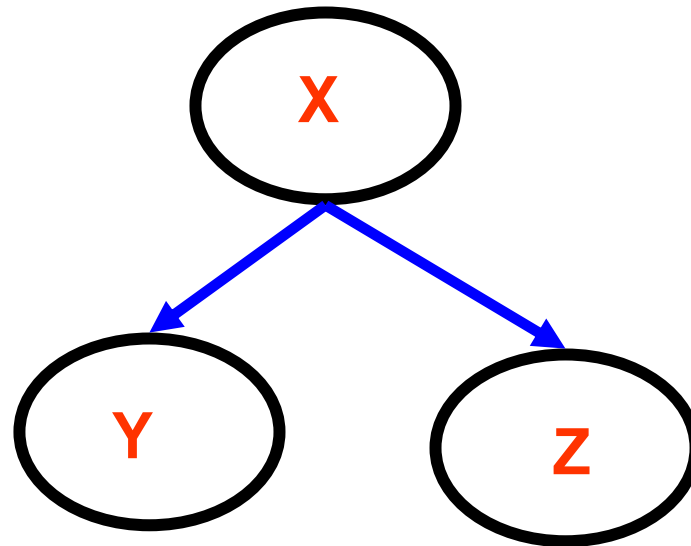
■ Smaller values in left subtree

■ Larger values in right subtree

■ Example

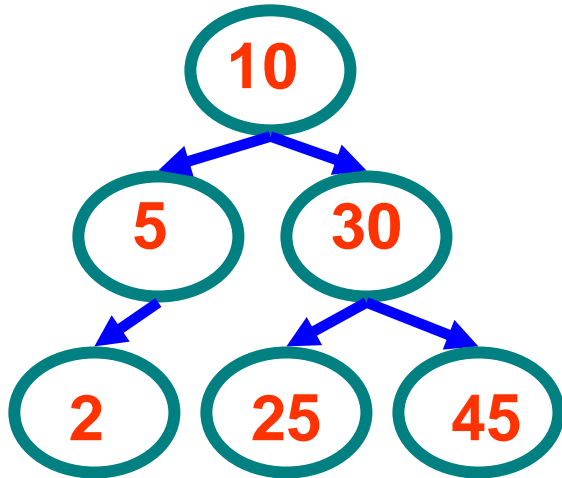
■ $X > Y$

■ $X < Z$

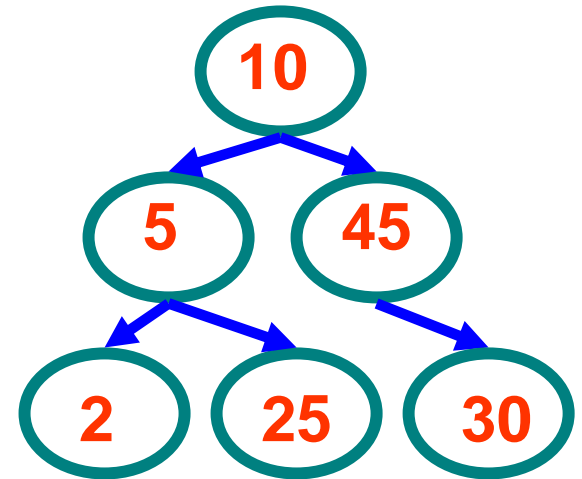
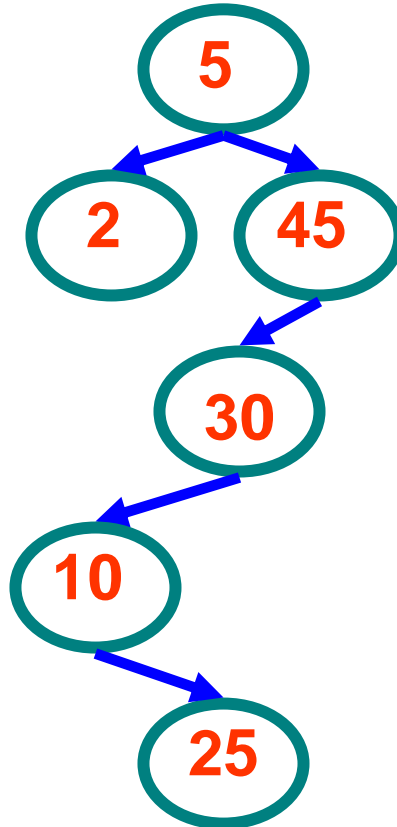


Binary Search Trees

■ Examples



Binary
search trees



Non-binary
search tree

Binary Tree Implementation

```
Class Node {  
    Value data;  
    Node left, right;    // null if empty  
  
    void insert ( Value data1 ) { ... }  
    void delete ( Value data2 ) { ... }  
    Node find ( Value data3 ) { ... }  
    ...  
}
```

Iterative Search of Binary Tree

```
Node Find( Node n, Value key) {  
    while (n != null) {  
        if (n.data == key)           // Found it  
            return n;  
        if (n.data > key)           // In left subtree  
            n = n.left;  
        else                         // In right subtree  
            n = n.right;  
    }  
    return null;  
}  
Find( root, keyValue );
```

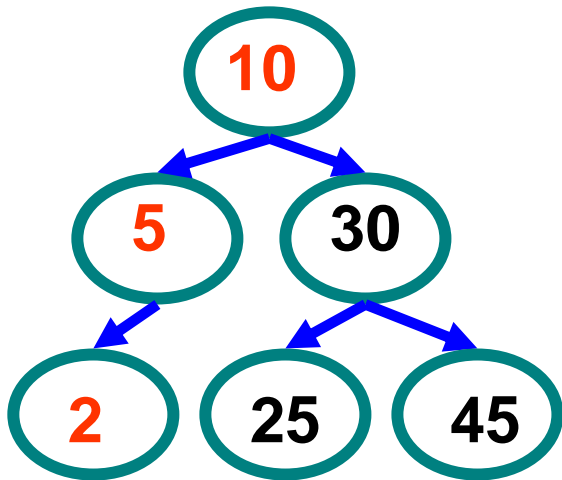
Recursive Search of Binary Tree

```
Node Find( Node n, Value key) {  
    if (n == null)                // Not found  
        return( n );  
    else if (n.data == key)        // Found it  
        return( n );  
    else if (n.data > key)         // In left subtree  
        return Find( n.left, key );  
    else                            // In right subtree  
        return Find( n.right, key );  
}
```

```
Find( root, keyValue );
```

Example Binary Searches

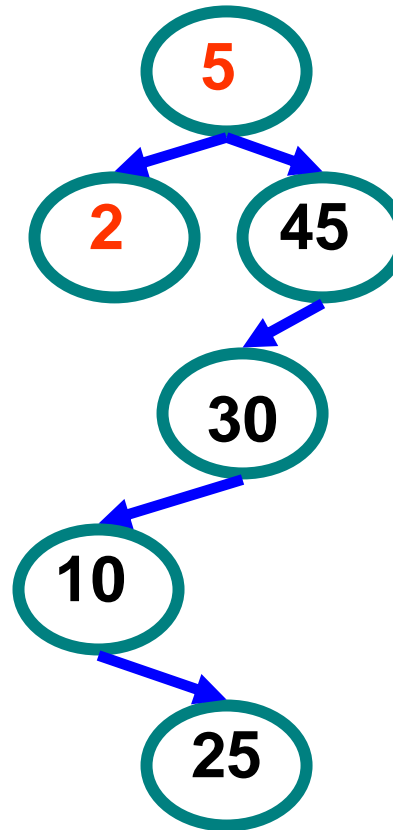
■ Find (2)



$10 > 2$, left

$5 > 2$, left

$2 = 2$, found

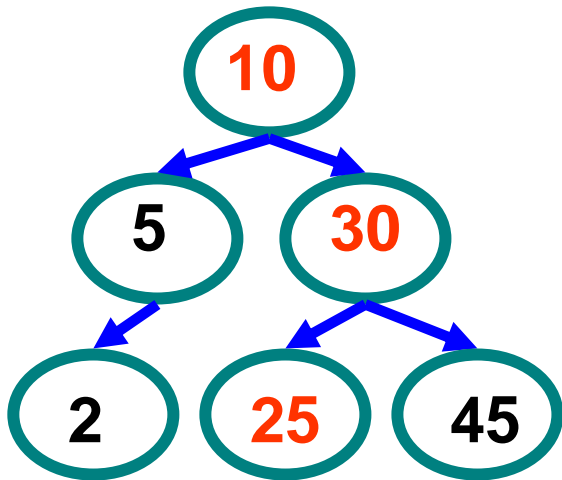


$5 > 2$, left

$2 = 2$, found

Example Binary Searches

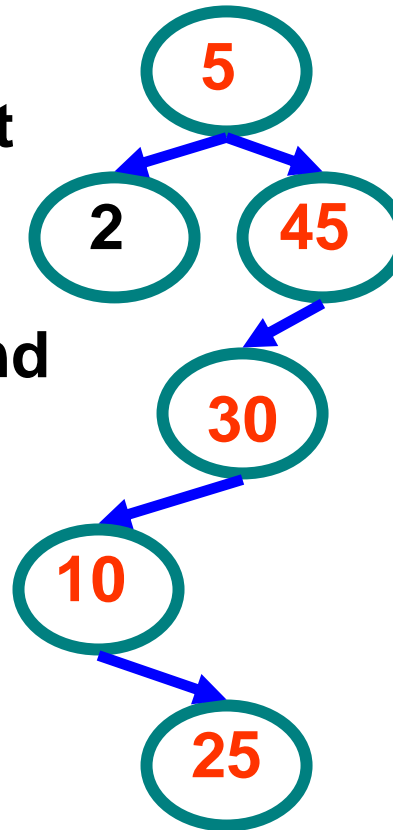
■ Find (25)



10 < 25, right

30 > 25, left

25 = 25, found



5 < 25, right

45 > 25, left

30 > 25, left

10 < 25, right

25 = 25, found

Binary Search Properties

■ Time of search

- Proportional to height of tree

- Balanced binary tree

 - $O(\log(n))$ time

- Degenerate tree

 - $O(n)$ time

 - Like searching linked list / unsorted array

■ Requires

- Ability to compare key values

Binary Search Tree Construction

- **How to build & maintain binary trees?**
 - **Insertion**
 - **Deletion**
- **Maintain key property (invariant)**
 - **Smaller values in left subtree**
 - **Larger values in right subtree**

Binary Search Tree – Insertion

■ Algorithm

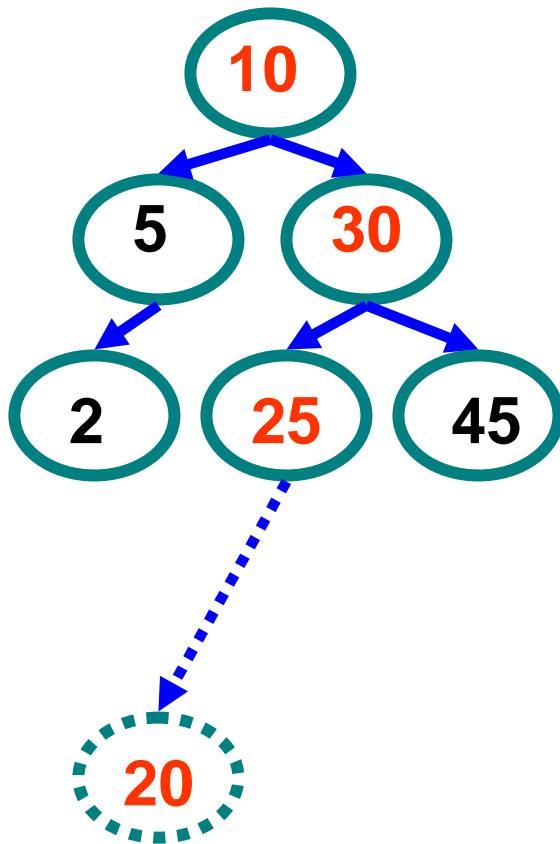
1. Perform search for value X
2. Search will end at node Y (if X not in tree)
3. If $X < Y$, insert new leaf X as new left subtree for Y
4. If $X > Y$, insert new leaf X as new right subtree for Y

■ Observations

- $O(\log(n))$ operation for balanced tree
- Insertions may unbalance tree

Example Insertion

■ Insert (20)



$10 < 20$, right

$30 > 20$, left

$25 > 20$, left

Insert 20 on left

Binary Search Tree – Deletion

■ Algorithm

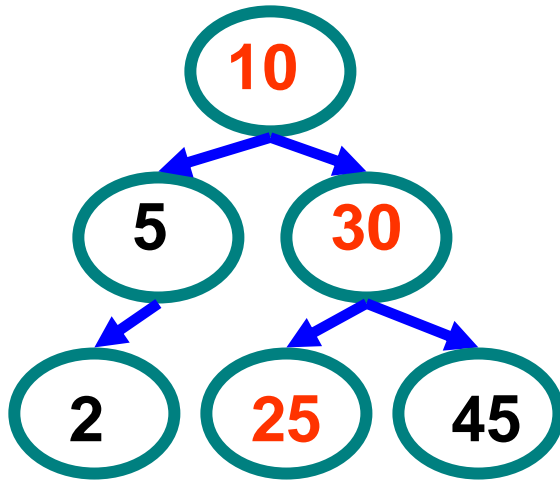
1. Perform search for value X
2. If X is a leaf, delete X
3. Else // must delete internal node
 - a) Replace with largest value Y on left subtree
OR smallest value Z on right subtree
 - b) Delete replacement value (Y or Z) from subtree

■ Observation

- $O(\log(n))$ operation for balanced tree
- Deletions may unbalance tree

Example Deletion (Leaf)

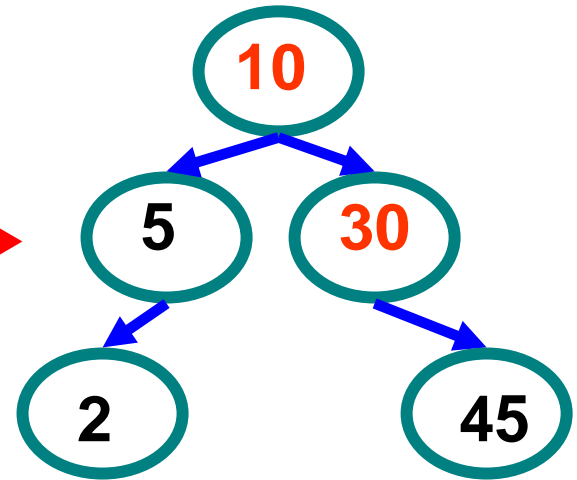
■ Delete (25)



$10 < 25$, right

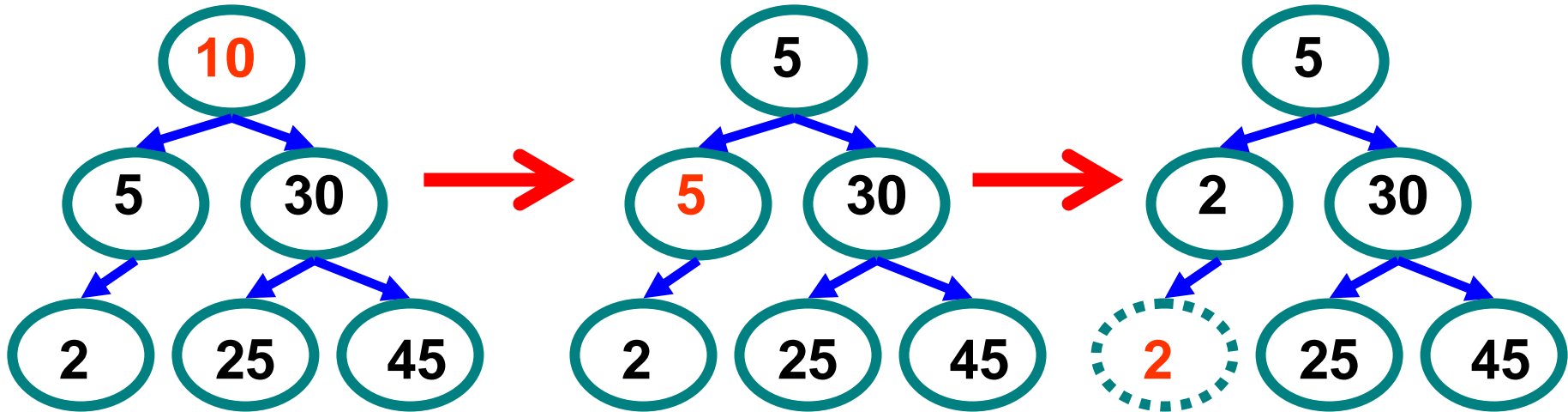
$30 > 25$, left

$25 = 25$, delete



Example Deletion (Internal Node)

■ Delete (10)



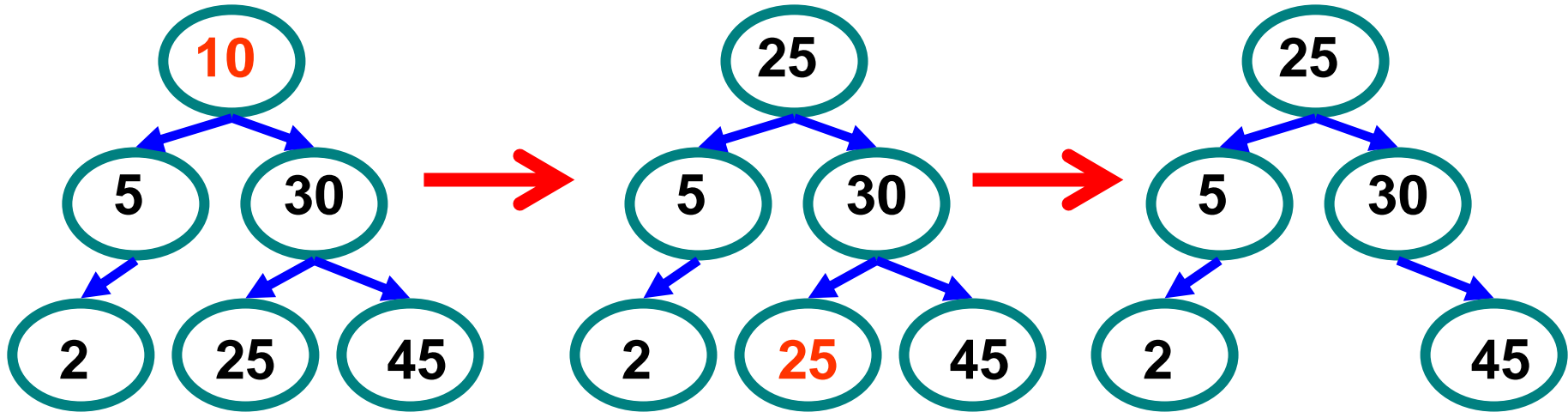
Replacing 10
with **largest**
value in left
subtree

Replacing 5
with **largest**
value in left
subtree

Deleting leaf

Example Deletion (Internal Node)

■ Delete (10)



Replacing 10
with **smallest**
value in right
subtree

Deleting leaf

Resulting tree

Building Maps w/ Search Trees

- Search trees often used to implement maps
 - Each non-empty node contains
 - Key
 - Value
 - Left and right child
- Need to be able to compare keys
 - Generic type `<K extends Comparable<K>>`
 - Denotes any type K that can be compared to K's

Polymorphic Binary Search Trees

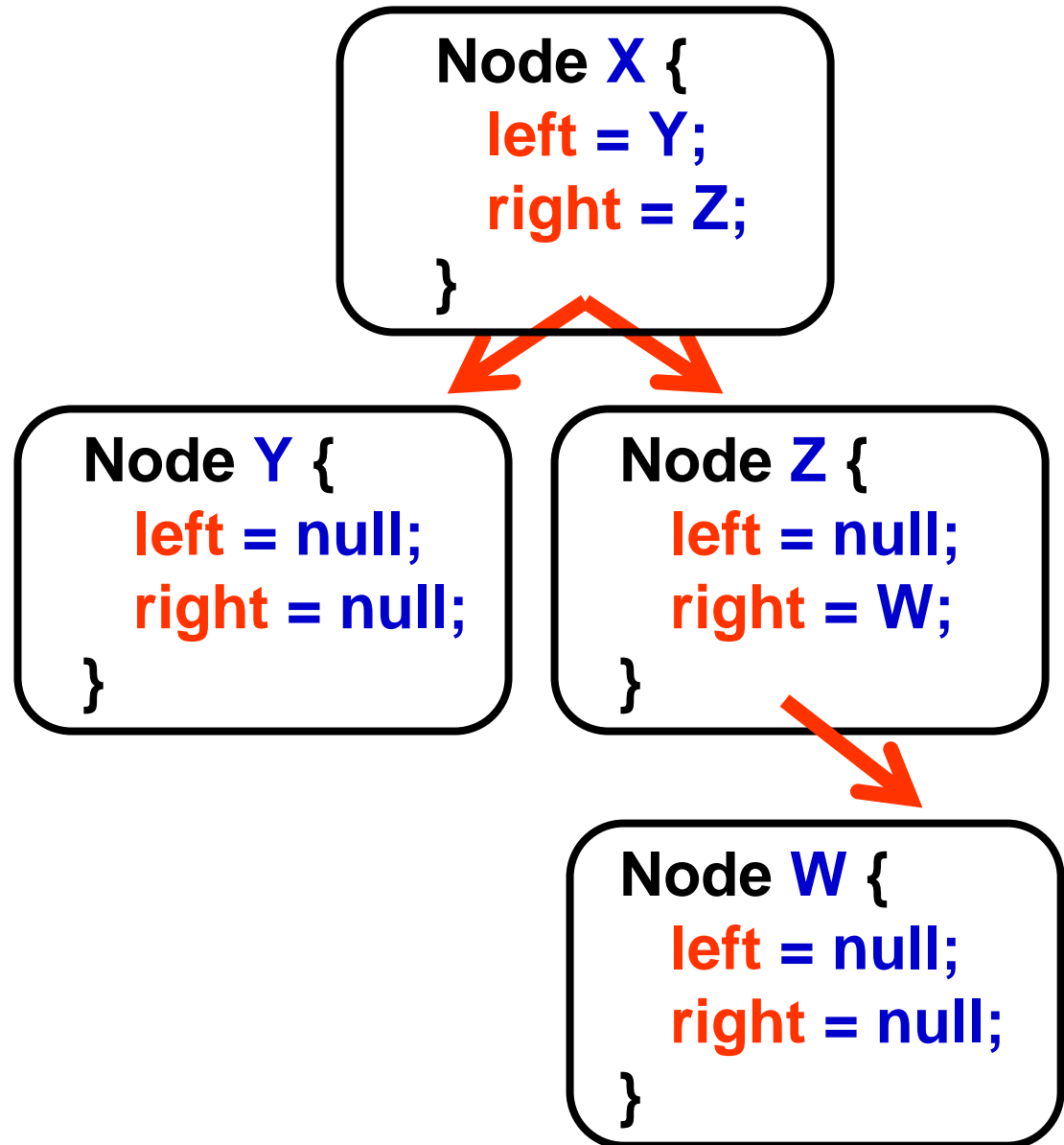
- **What do we mean by polymorphic?**
- **Implement two subtypes of Tree**
 1. **EmptyTree**
 2. **NonEmptyTree**
- **Use EmptyTree to represent the empty tree**
 - **Rather than null**
- **Invoke methods on tree nodes**
 - **Without checking for null**
 - **Get empty or nonempty functionality**
 - **Selected by type of tree node**

Polymorphic Binary Tree Implement.

```
Interface Tree {  
    Tree insert ( Value data1 ) { ... }  
}  
Class EmptyTree implements Tree {  
    Tree insert ( Value data1 ) { ... }  
}  
Class NonEmptyTree implements Tree {  
    Value data;  
    Tree left, right; // Either Empty or NonEmpty  
    Tree insert ( Value data1 ) { ... }  
}
```

Example : Standard Binary Tree

```
Class Node {  
    Node left, right;  
}
```



Example : Polymorphic Binary Tree

```
Class EmptyTree {  
}  
Class NonEmptyTree {  
  Tree left, right;  
}
```

