PROGRAMMING AND DATA STRUCTURES

BINARY TREES

HOURIA OUDGHIRI

FALL 2021

OUTLINE

- Characteristics of binary trees
- Binary Search Trees (BST)
- Operations on BSTs
- Implementation of the BST class
- Heap
- Operations on the Heap
- Implementation of the Heap class

STUDENT LEARNING OUTCOMES

At the end of this chapter, you should be able to:

- Describe the properties of binary trees/BST/Heap
- Trace operations on binary trees/BST/Heap
- Implement BST/Heap generic data structures
- Use the BST/Heap data structures
- Evaluate the complexity of the operations on BST/Heap

What is a binary tree?

- Data organized in a binary tree structure
- Easy and efficient access and update in large collections of data
- Used for efficient search operations
- Wide range of applications: mathematical expressions, game strategies, decision trees, data compression, …

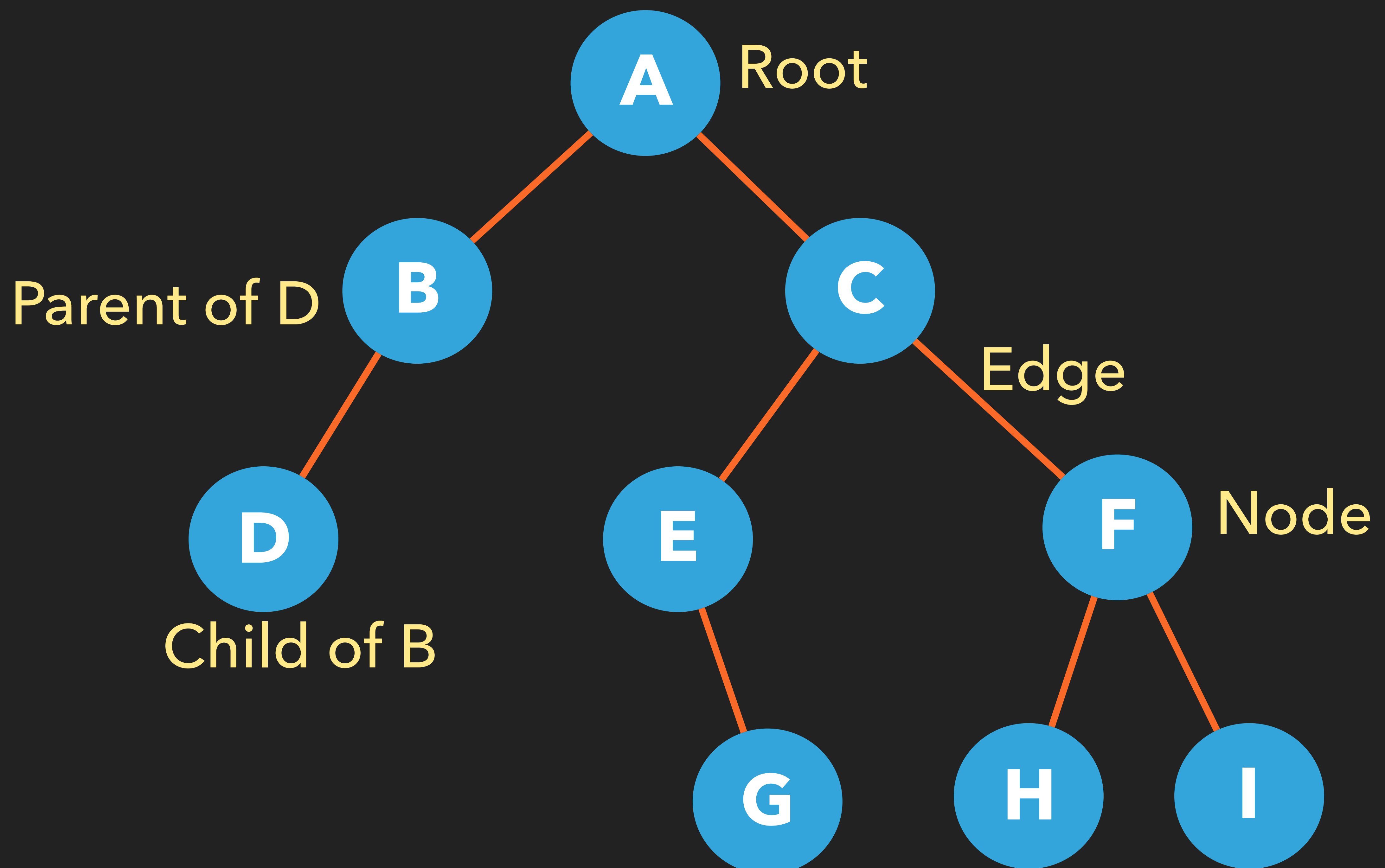
Properties

- Set of elements called nodes (vertices) interconnected with edges (arcs)
- The first node is called the root
- The root is connected to two binary trees (left subtree and right subtree)

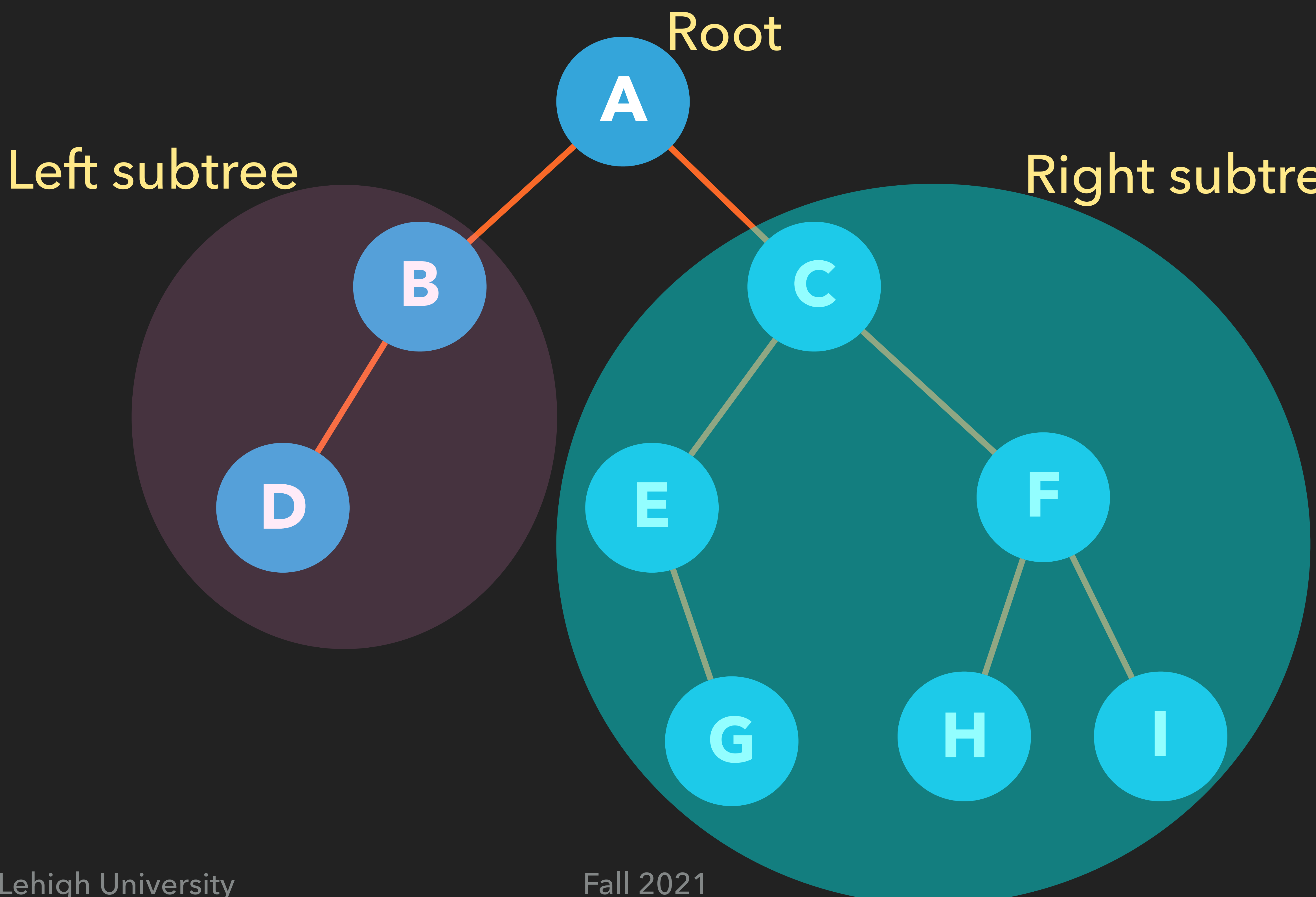
Properties

- Every node has a parent (except the root) and may have one child or two children at most
- The root is the ancestor of all the nodes in the tree

Properties



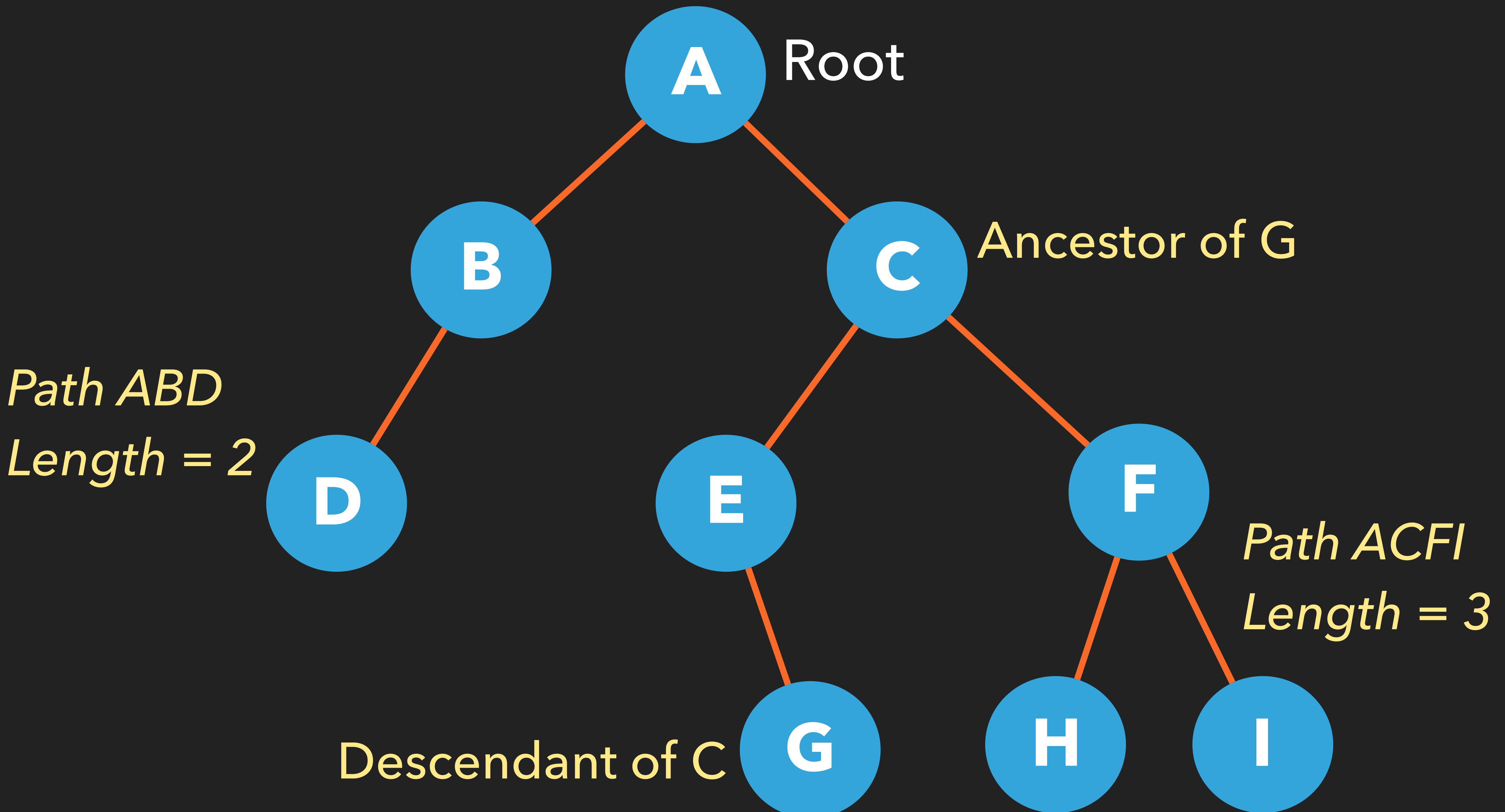
Properties



Properties

- **Path**: Sequence of connected nodes starting at any level of the tree
- **Length of a path**: the number of nodes in the sequence - 1 (number of edges)
- If there is a path from node P to node Q, Q is the **descendant** of P and P is an **ancestor** of Q

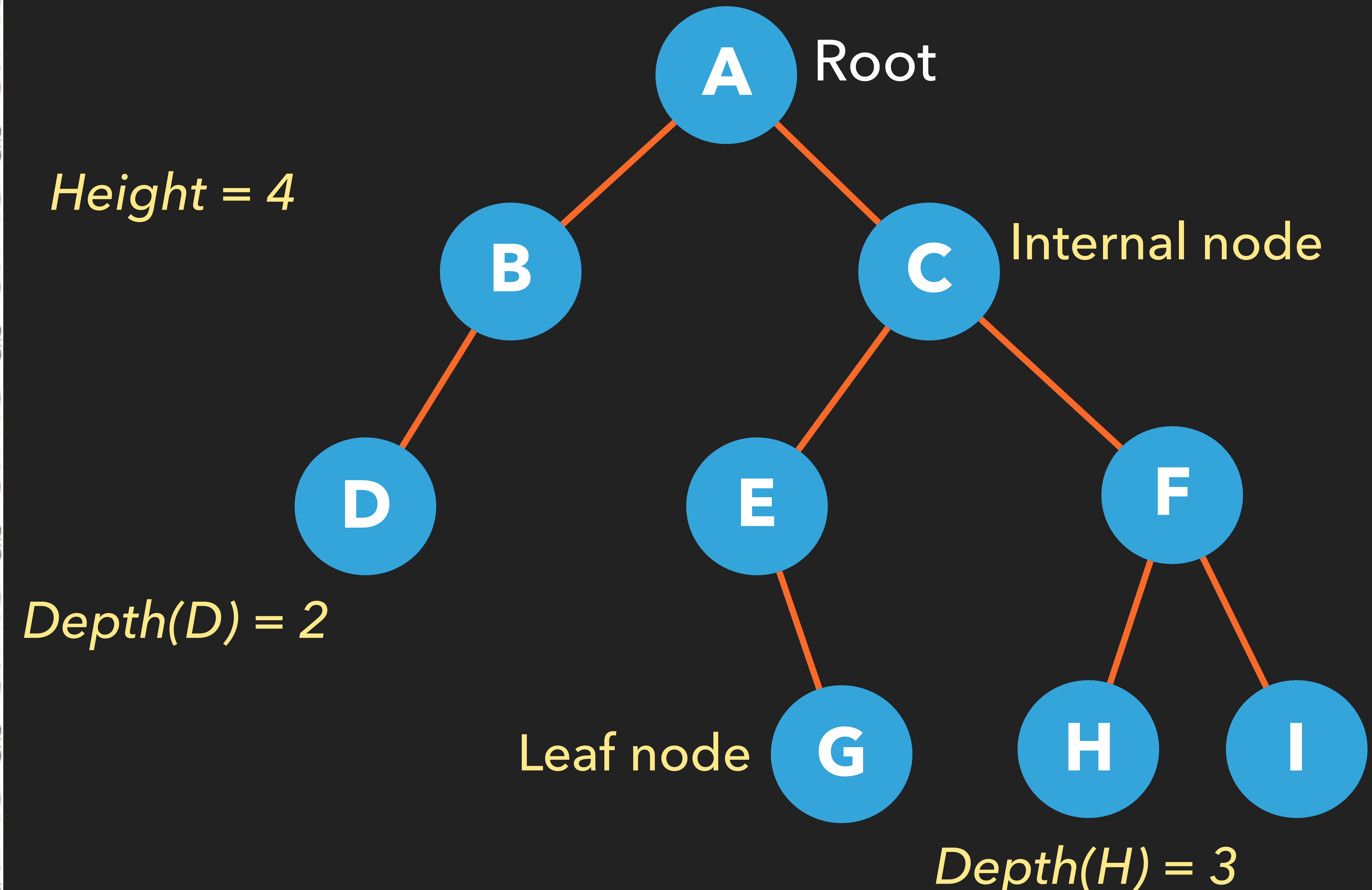
Properties


A
Root
B
C
Ancestor of G
Path ABD
Length = 2
D
E
F
Path ACFI
Length = 3
G
Descendant of C
H
I

Properties

- Depth of a node: Length of the path from the root to the node
- Height of a tree: the depth of the deepest node + 1
- Leaf node: node that has no children
- Internal node: node that has at least one child

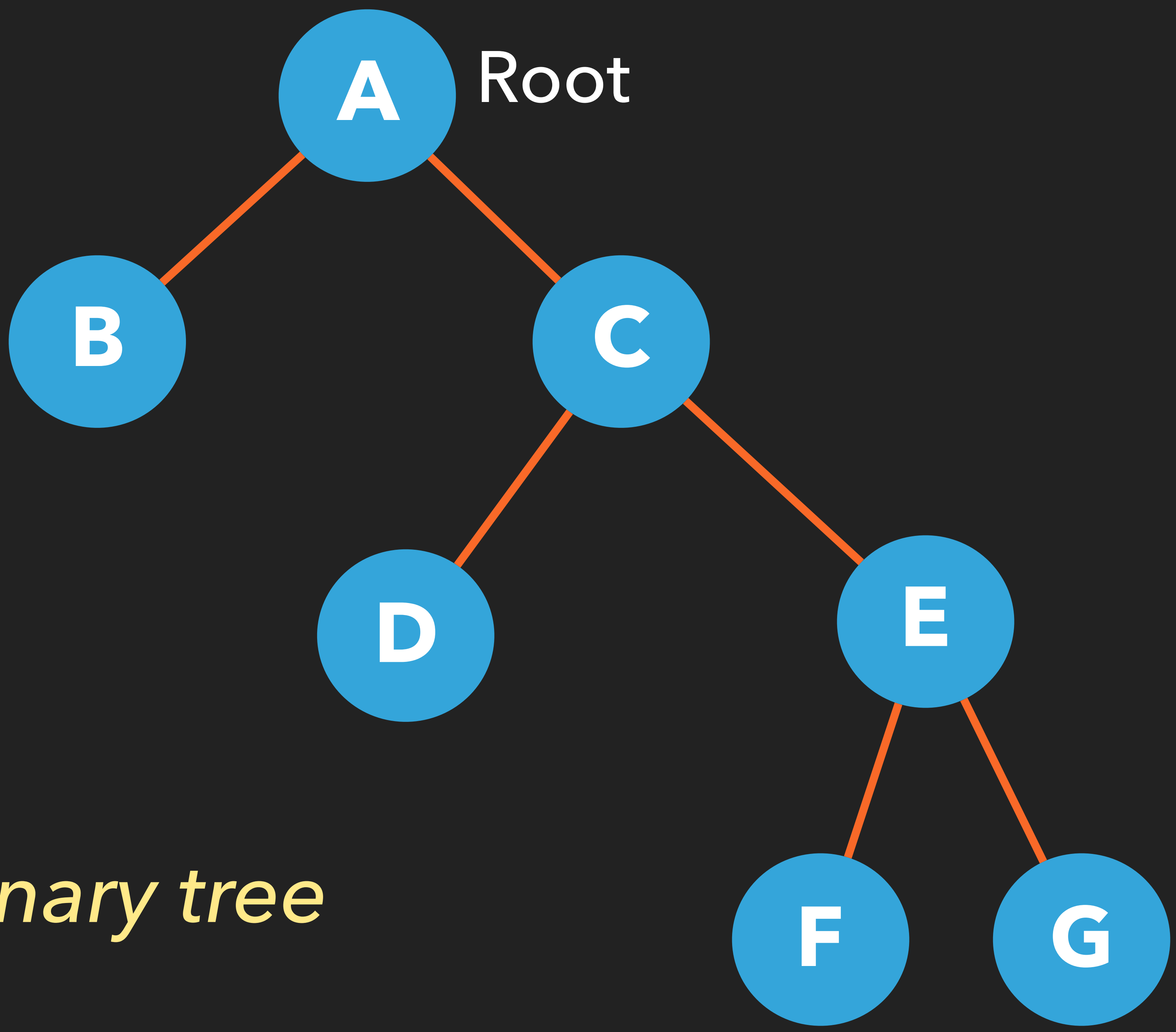
Properties


Height = 4
A
Root
B
C
Internal node
D
E
F
Depth(D) = 2
Leaf node
G
H
I
Depth(H) = 3

Properties

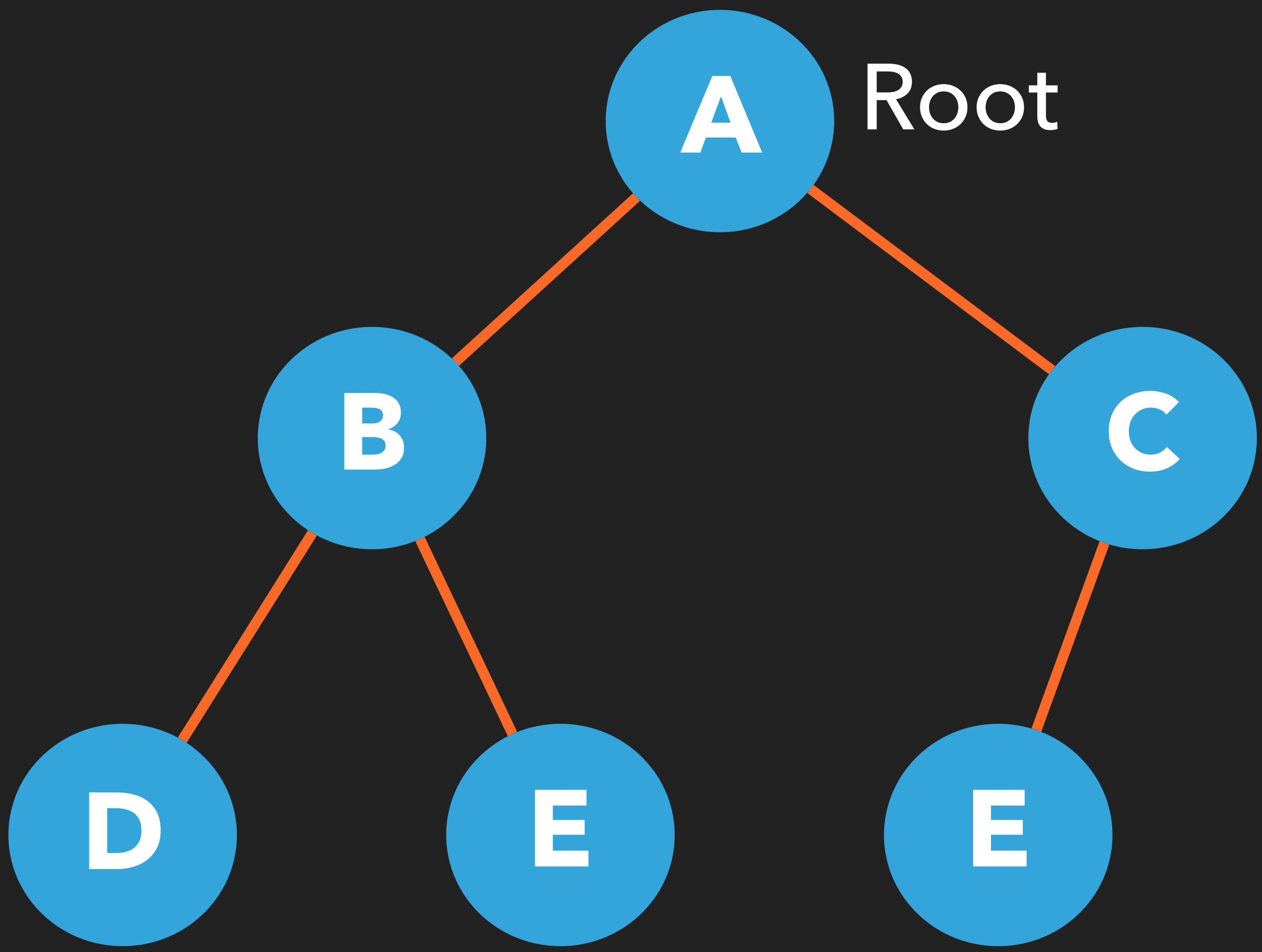
- **Full Binary Tree**: each node is a leaf or an internal node with exactly two children
- **Complete Binary Tree**: Every level is filled except the last level, and the leaves on the last level are placed leftmost

Properties



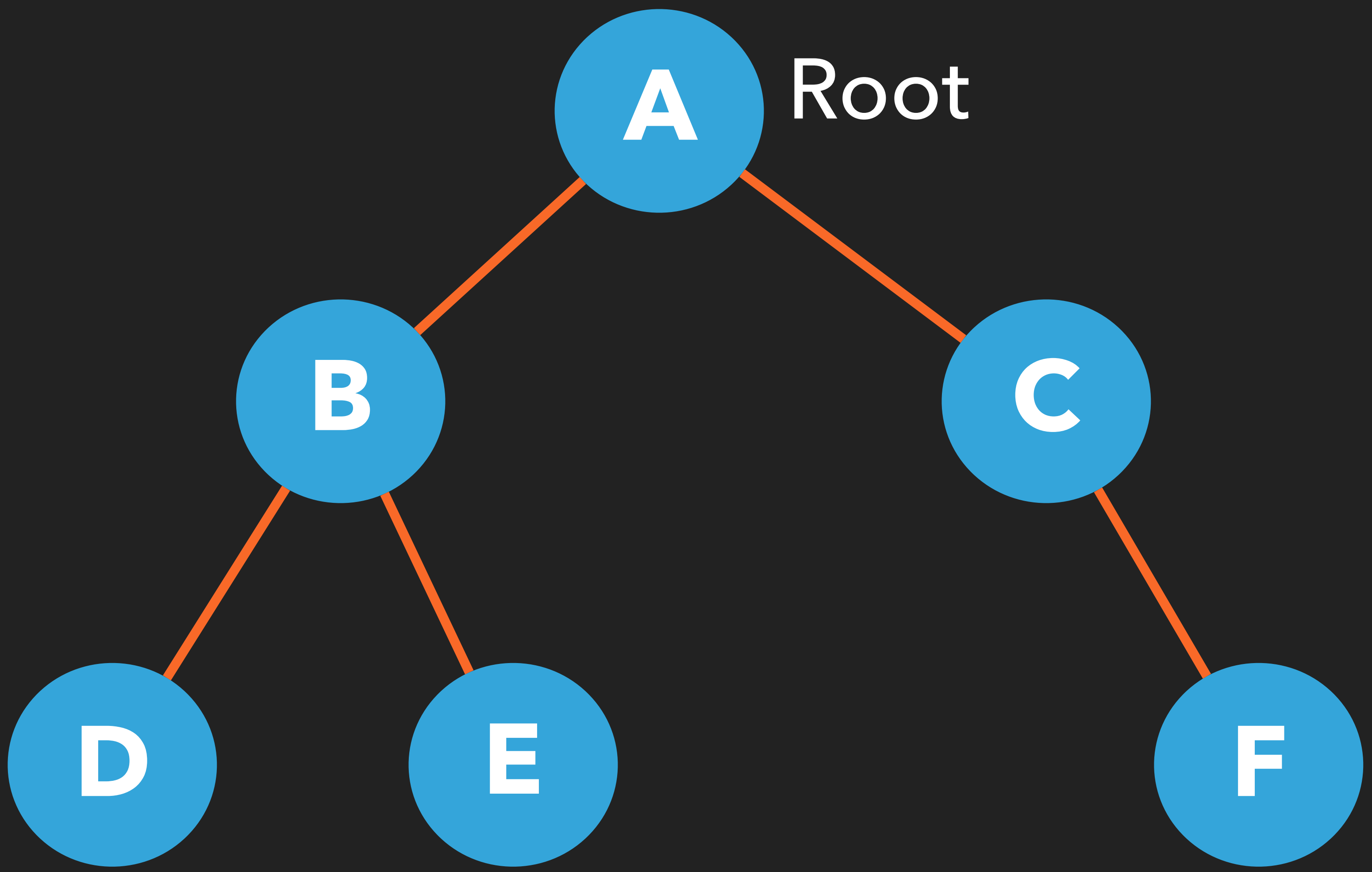
Full binary tree

Properties



Complete binary tree but not full

Properties

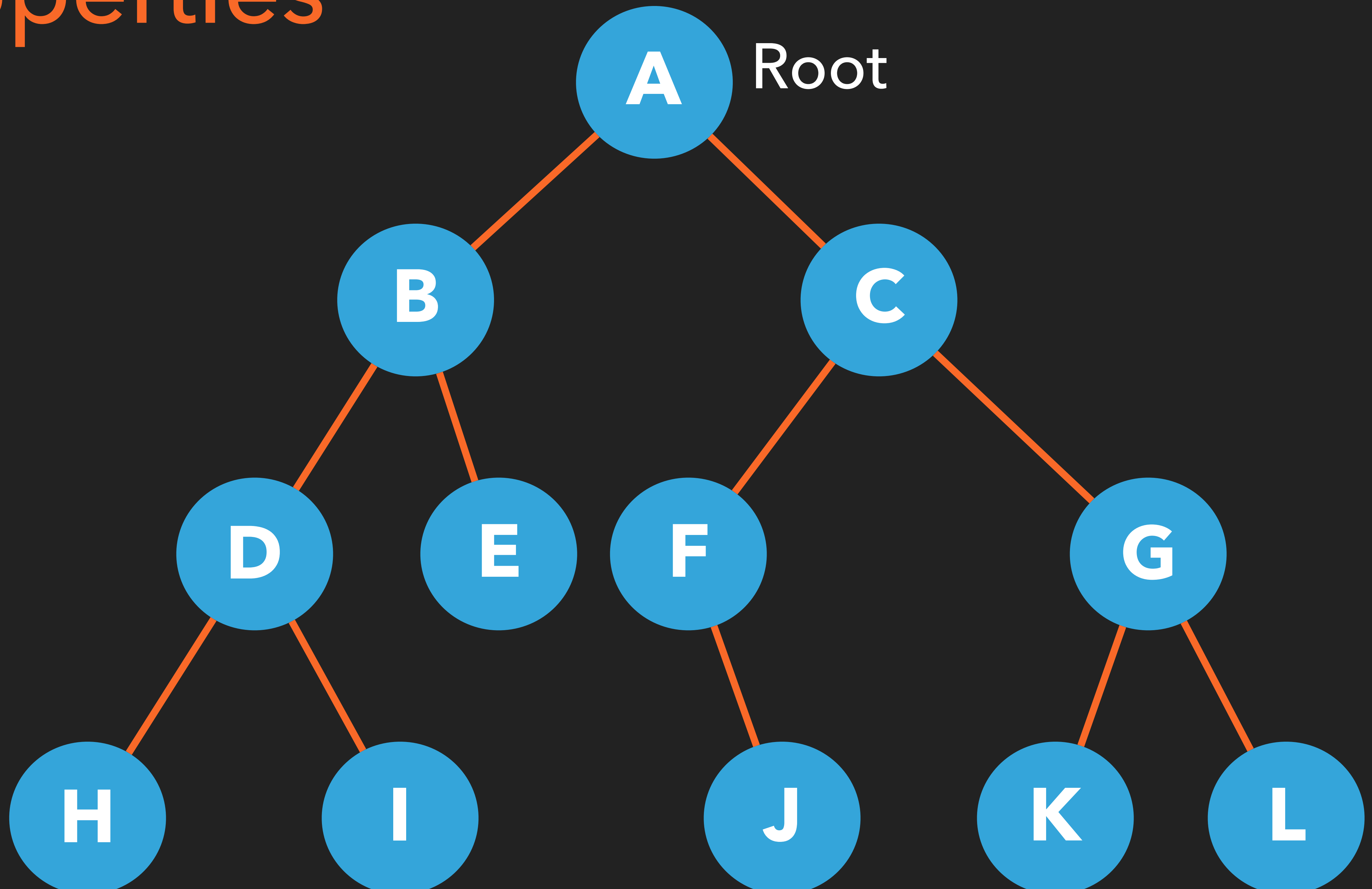


Not Complete - not full

Properties

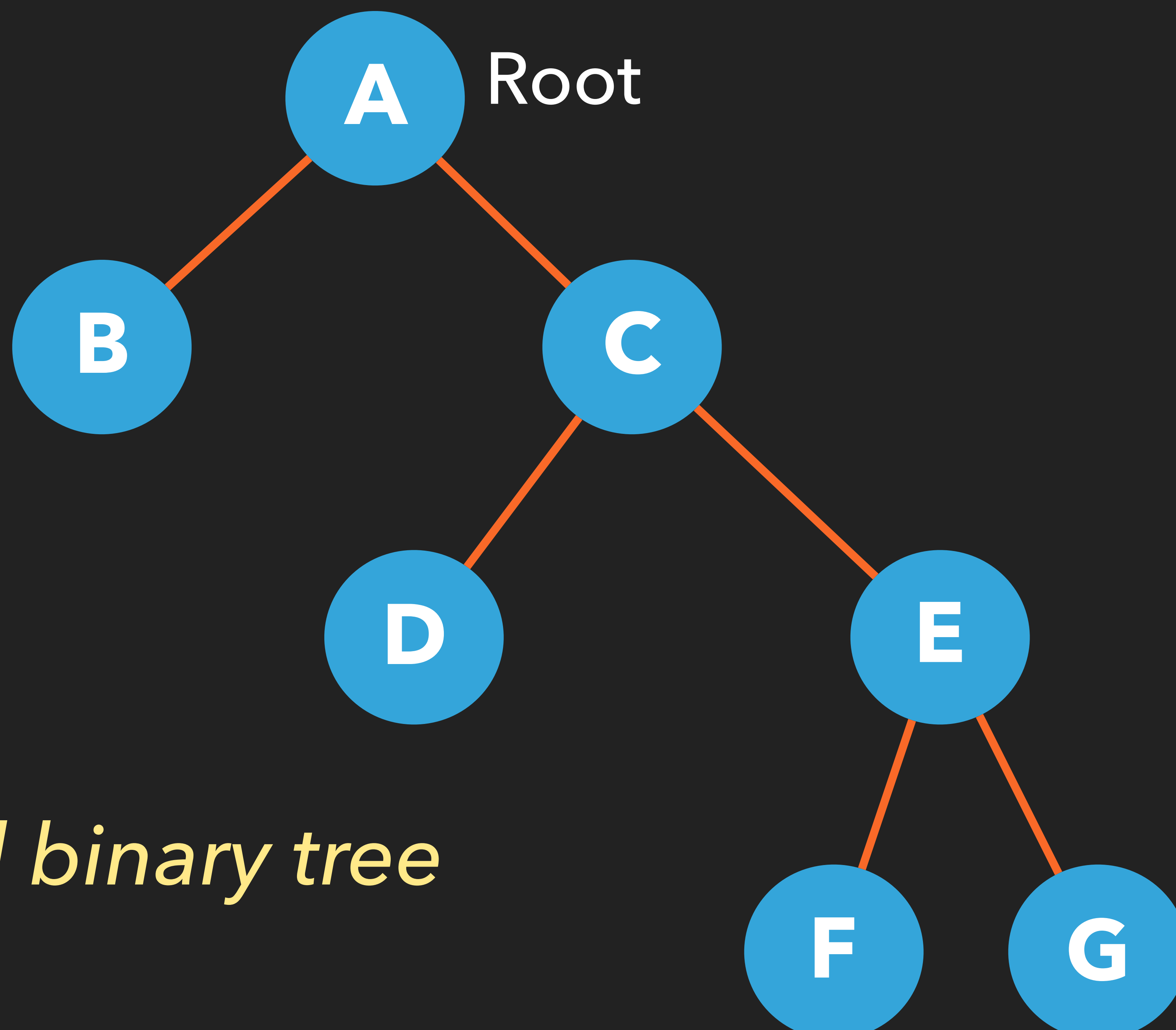
- **Balanced Binary Tree**: for each node, the height of the left subtree and the height of the right subtree differ by at most 1

Properties



Balanced binary tree

Properties



Unbalanced binary tree

Practice

Root?

Depth(35)?

Path from 70 to 72?

Height of the tree?

Leaf nodes?

Internal nodes?

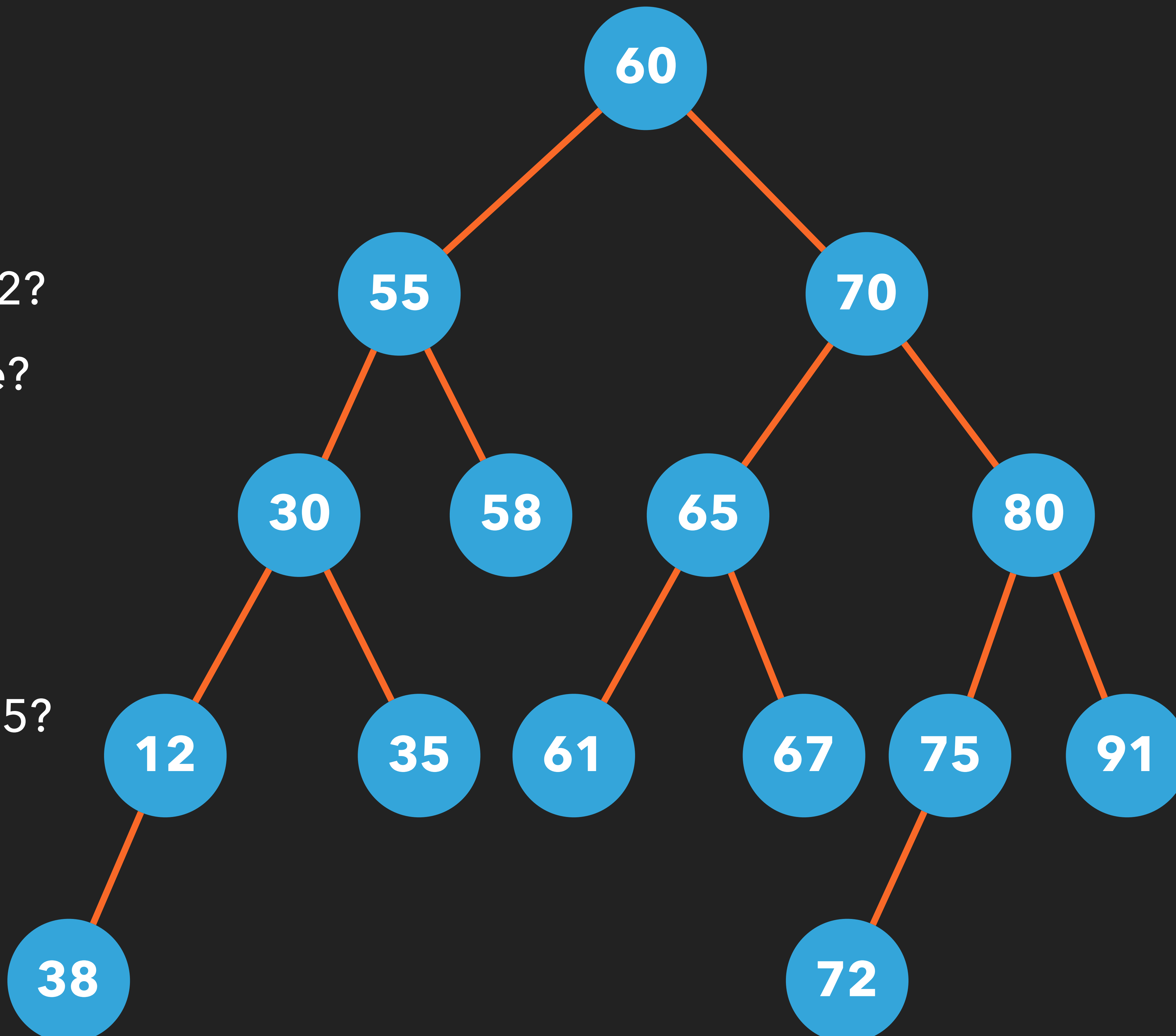
Children of 55?

Descendants of 55?

Full?

Complete?

Balanced?



Binary Tree Traversal

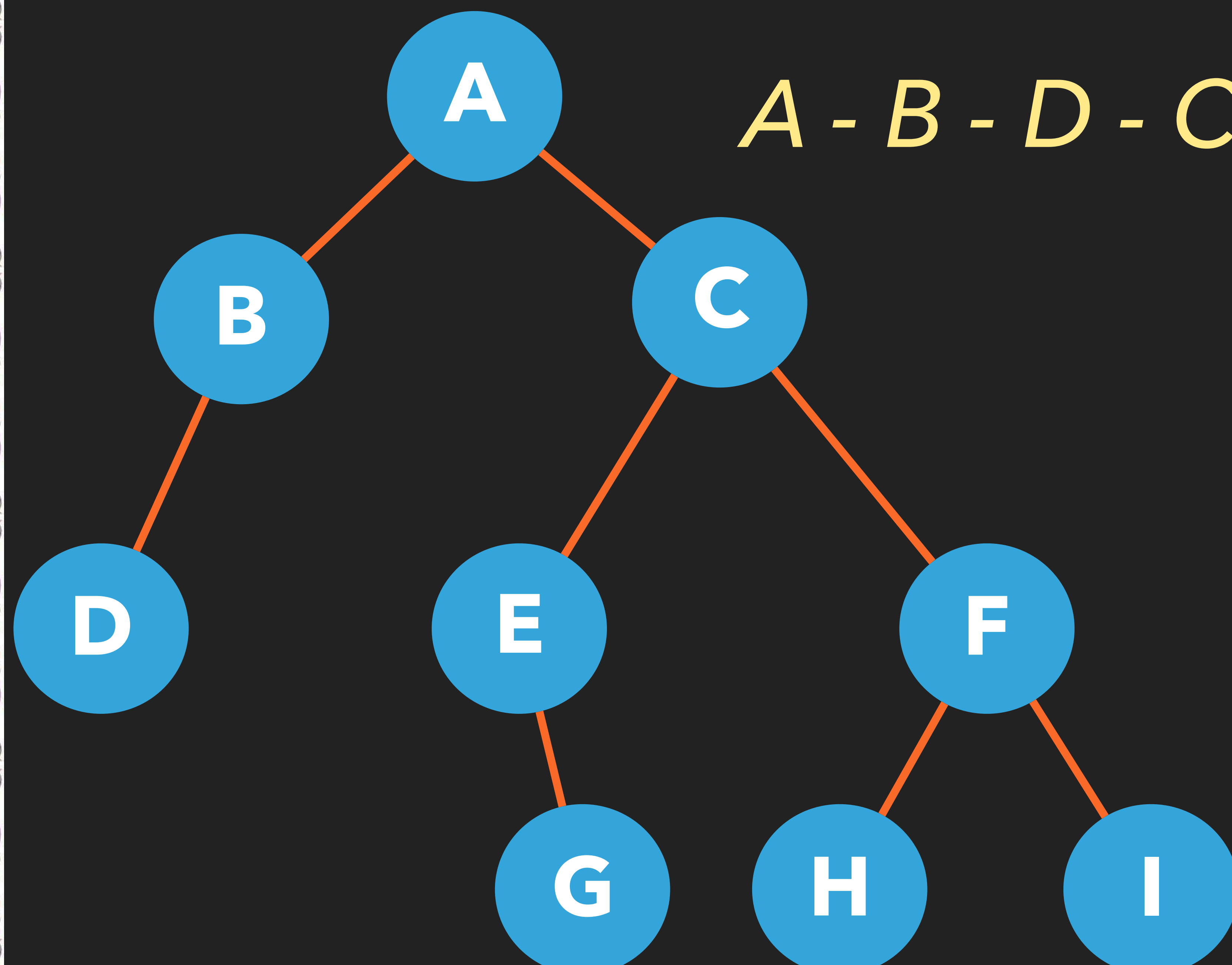
- Any process of visiting all the nodes in the tree is called traversal
- Three common traversals
  - Preorder
  - Inorder
  - Postorder

Preorder Traversal

- Any node is visited before its children
- V-L-R: Visit Node - Go Left - Go Right
  - Visit the node
  - Traverse the left subtree
  - Traverse the right subtree

Preorder Traversal

- V-L-R: Visit Node - Go Left - Go Right



A - B - D - C - E - G - F - H - I

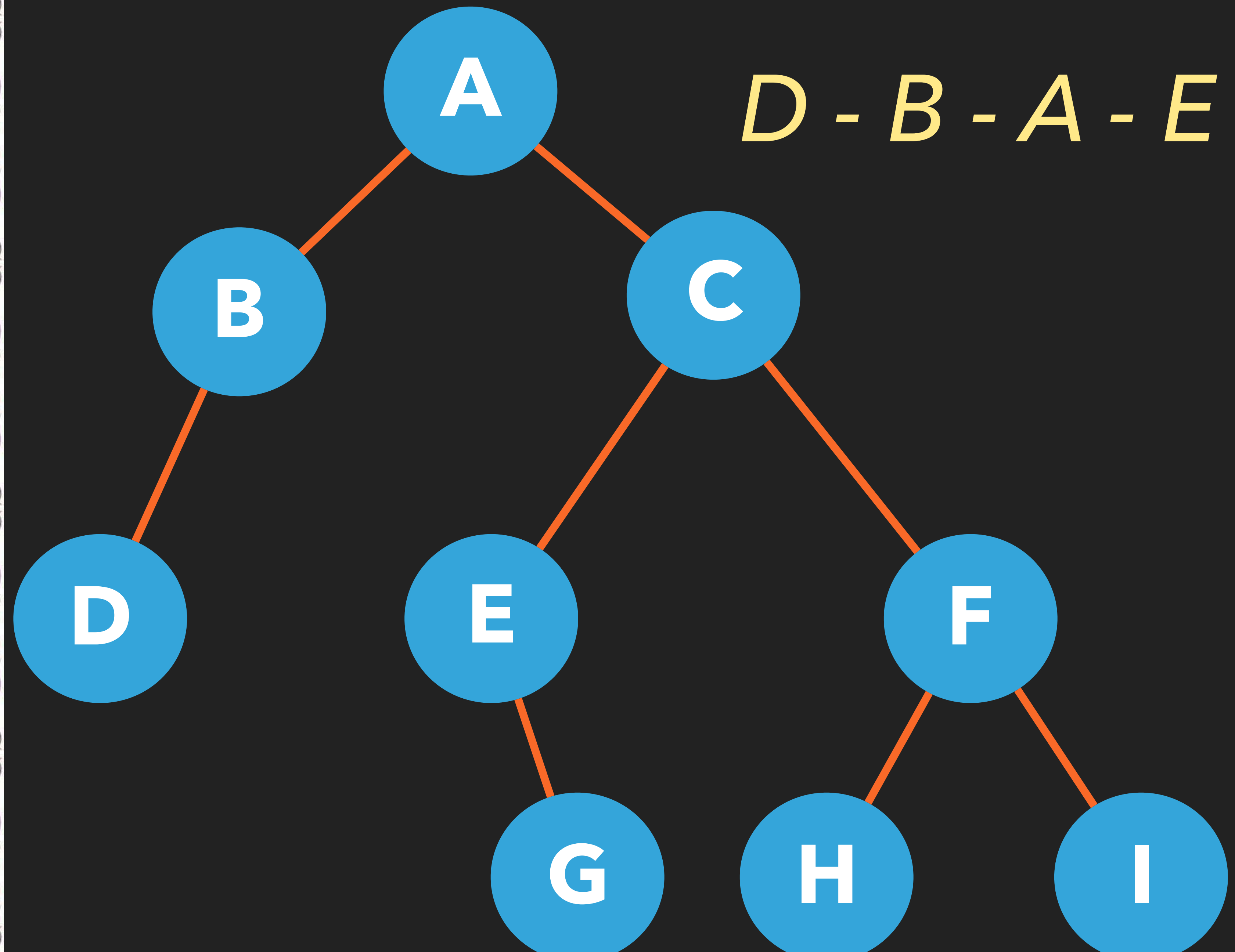
Inorder Traversal

- Any node is visited after its left subtree and before its right subtree
- L-V-R: Go Left - Visit Node - Go Right
  - Traverse the left subtree
  - Visit the node
  - Traverse the right subtree

In order Traversal

- L-V-R: Go Left - Visit Node - Go Right

D - B - A - E - G - C - H - F - I

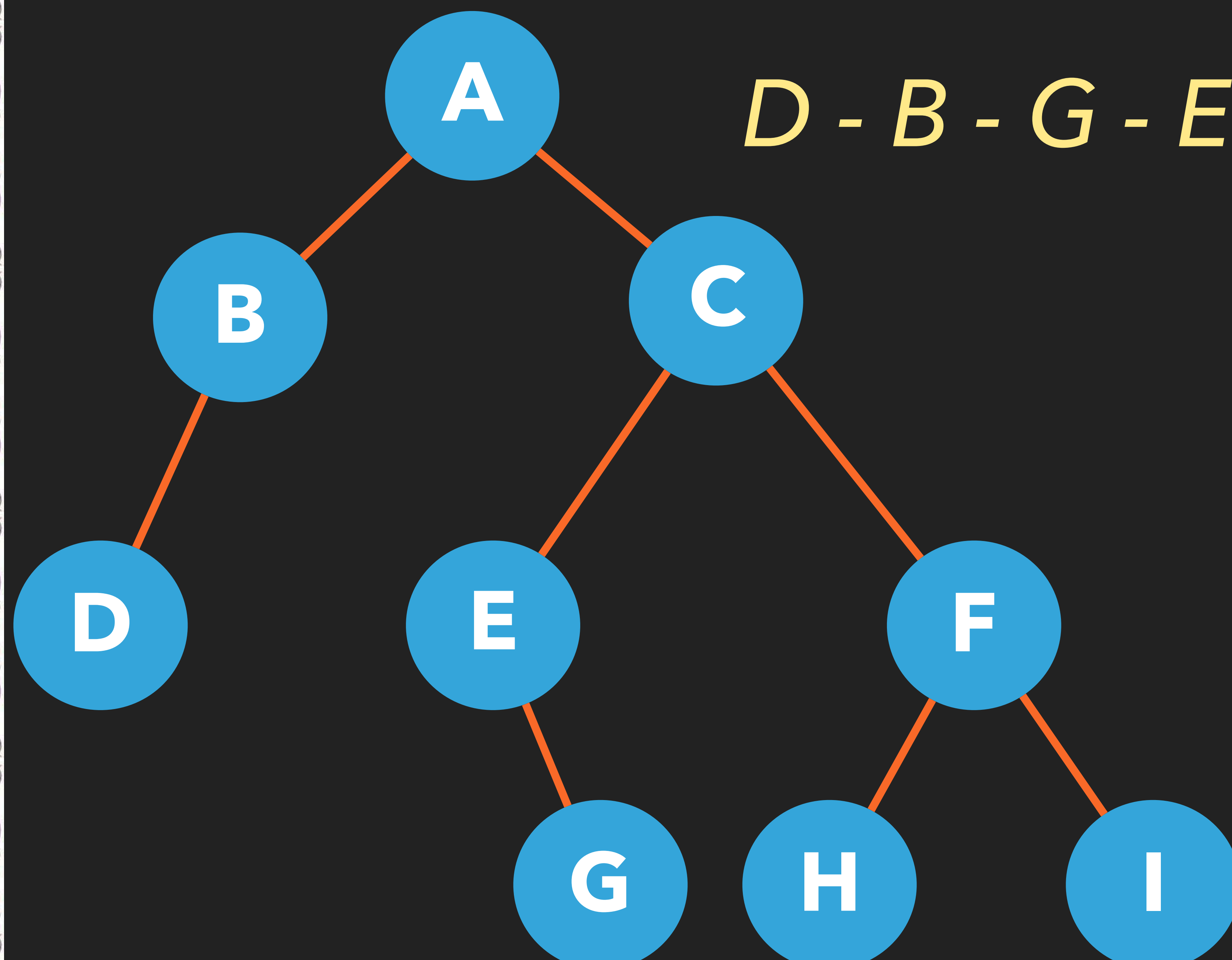


Postorder Traversal

- Any node is visited after its left subtree and right subtree
- L-R-V: Go Left - Go Right - Visit Node
  - Traverse the left subtree
  - Traverse the right subtree
  - Visit the node

Postorder Traversal

- L-R-V: Go Left - Go Right - Visit Node



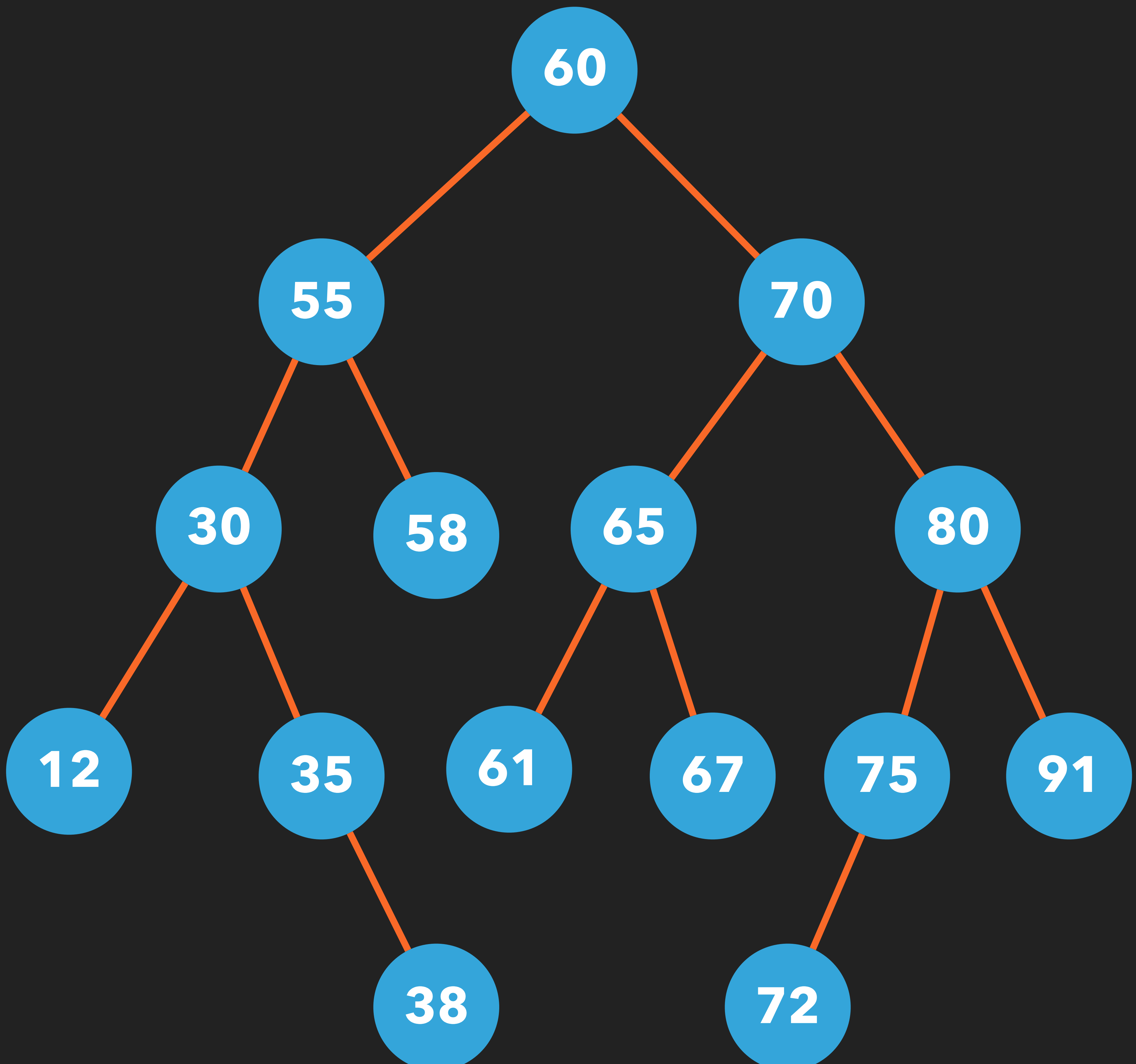
D - B - G - E - H - I - F - C - A

Practice

Preorder:

Inorder:

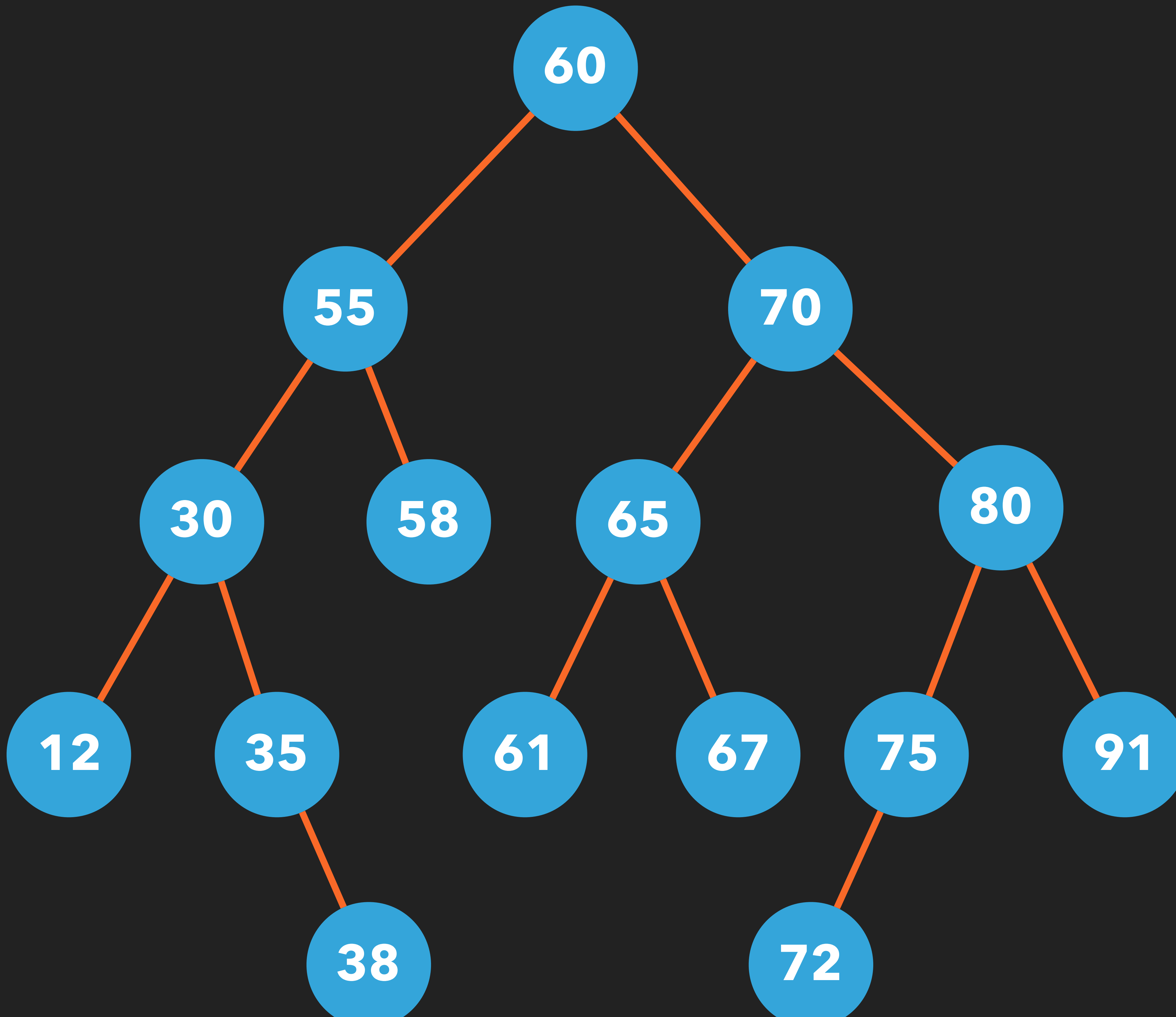
Postorder:



Binary Search Tree (BST)

- Special binary tree
  - BST has a root, a left subtree (L) and a right subtree (R)
  - The value of the root is greater than the value of every node in L
  - The value of the root is less than the value of every node in R
  - L and R are also BSTs
  - Used for efficient search in large data sets

BST

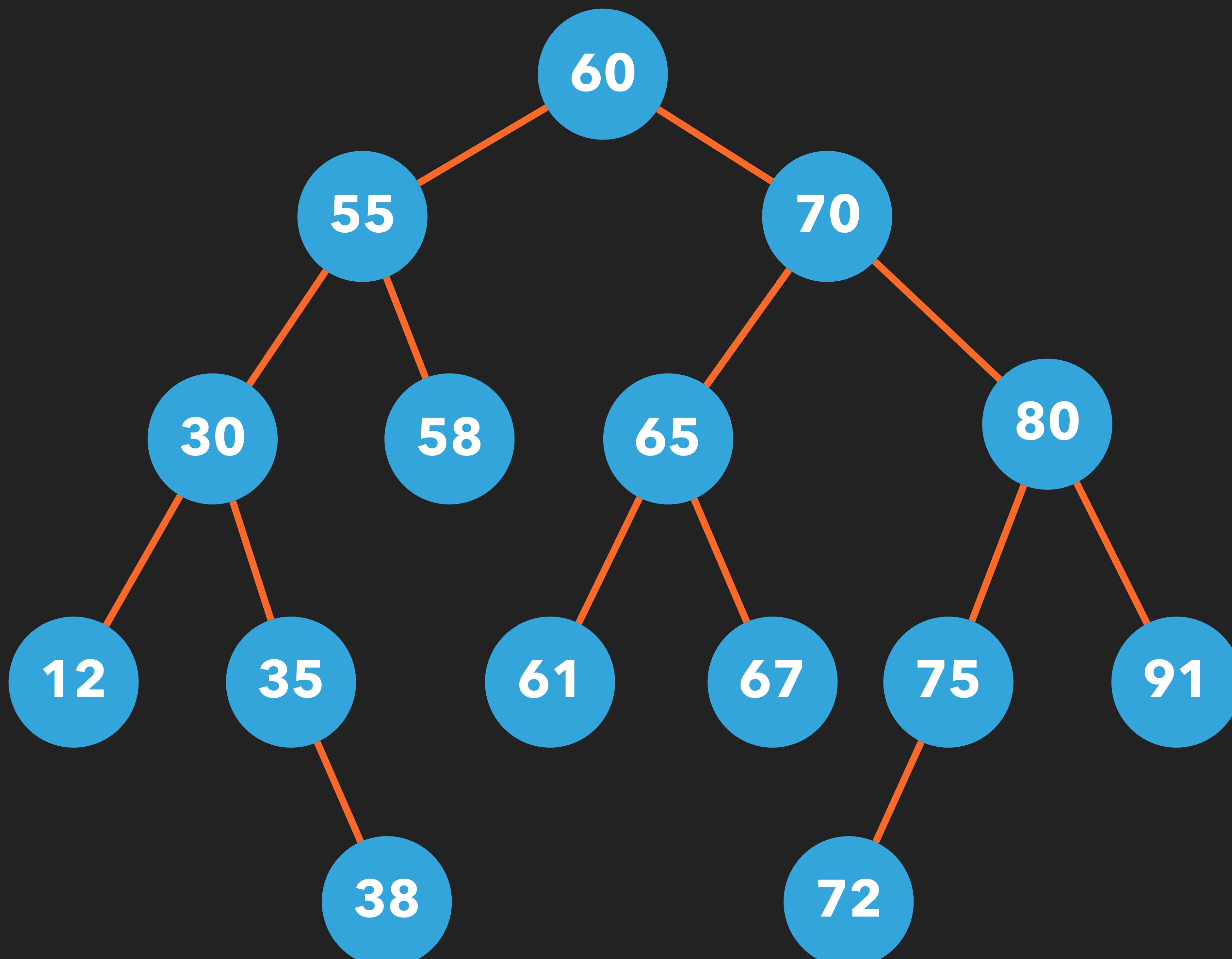


BST

- Common operations on BST
  - Search for a specific value in the BST
  - Add a node to the BST while keeping the BST properties
  - Remove a node from the BST while keeping the BST properties
  - Traverse the BST (preorder, inorder, postorder)

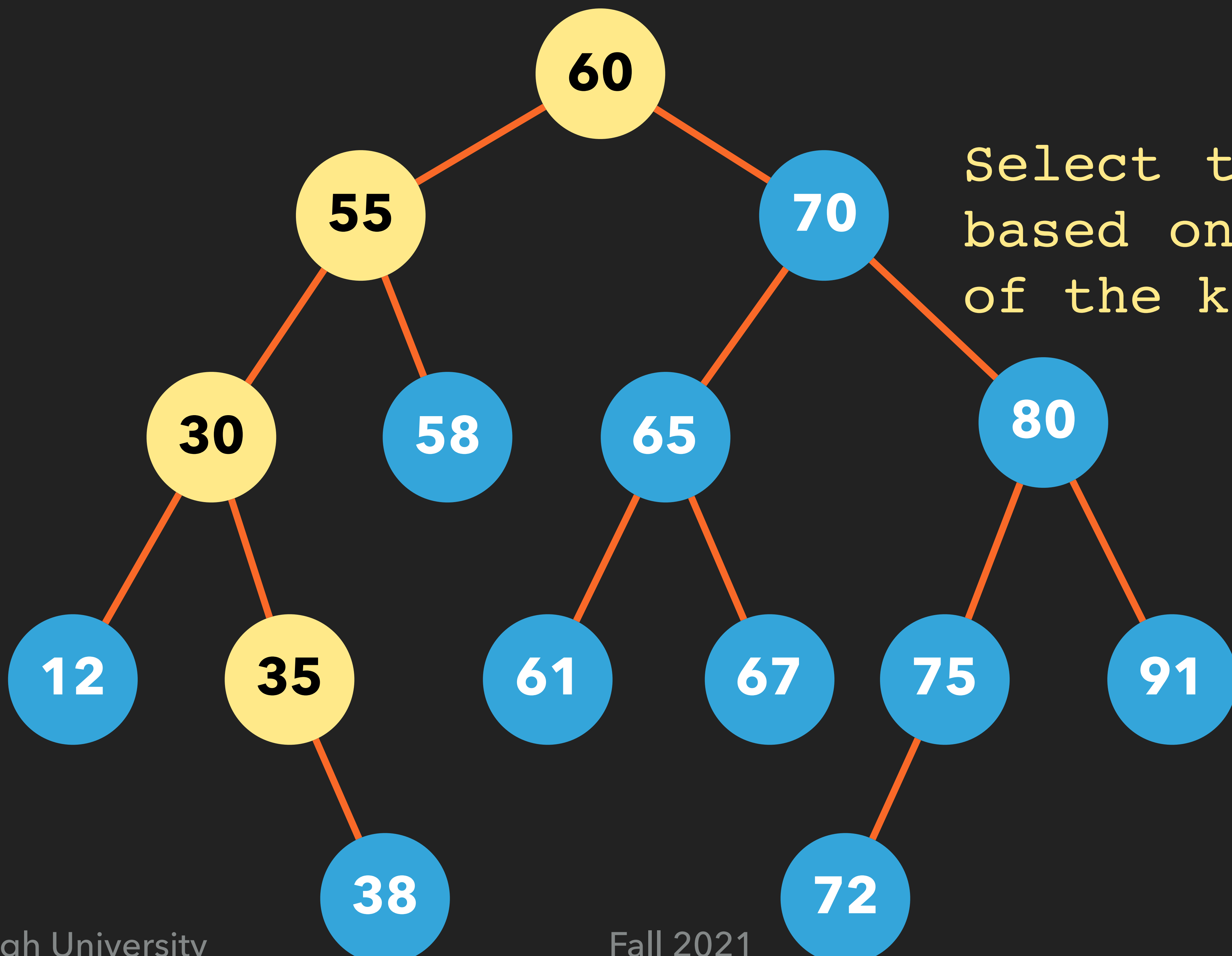
BST - Search

Search for the value 35



BST - Search operation

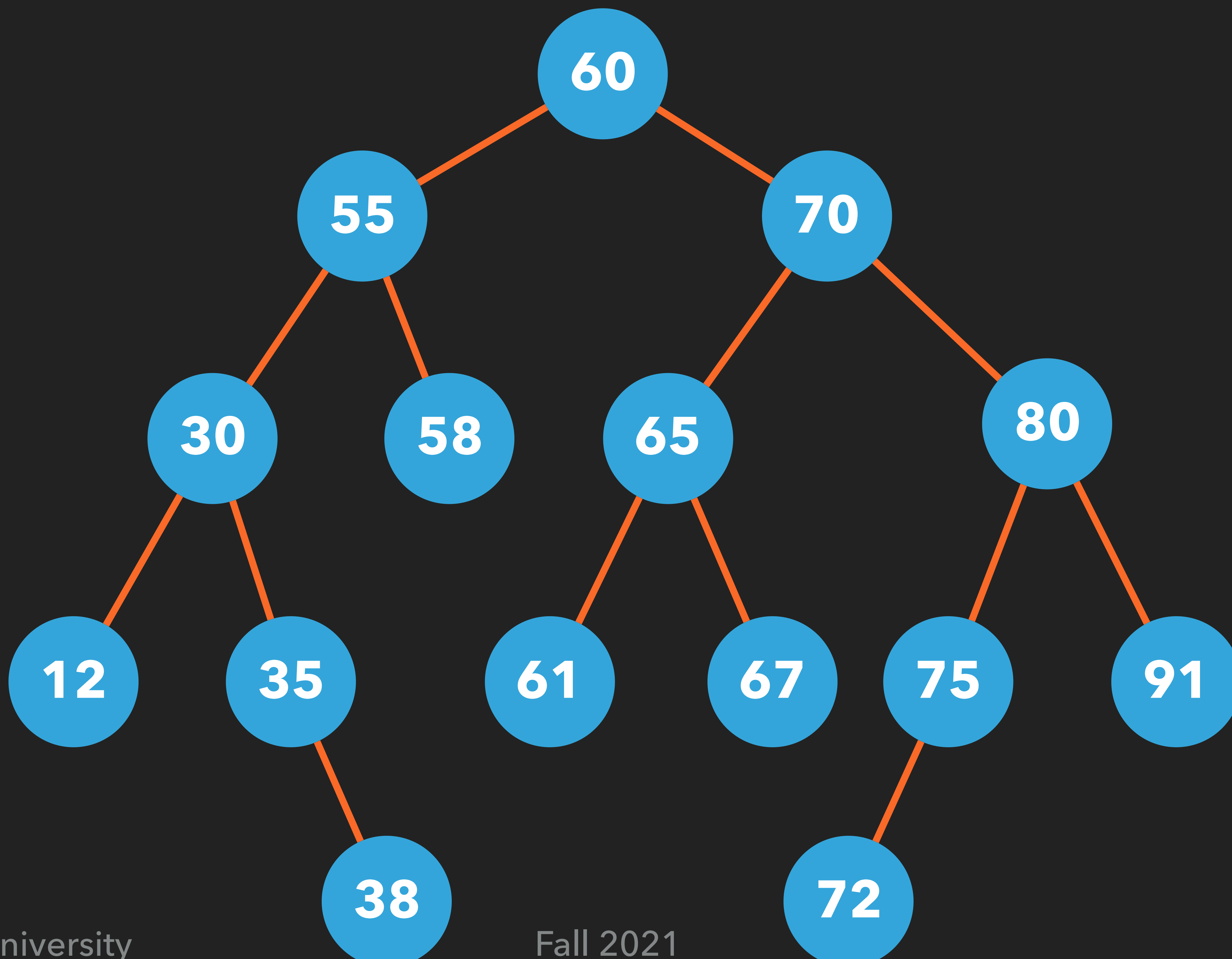
Search for the value 35



Select the subtree based on the value of the key

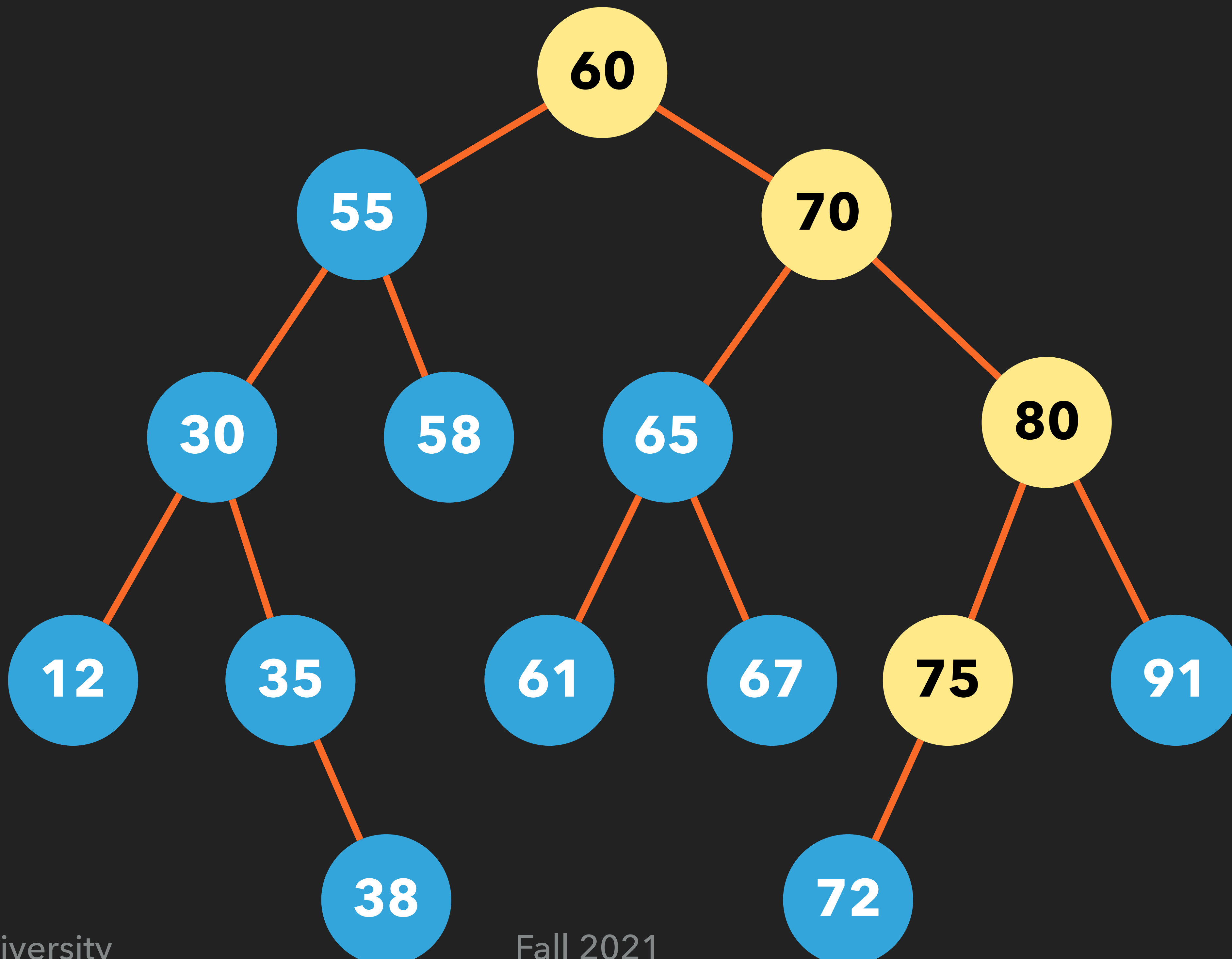
BST - Search operation

Search for the value 75



BST - Search operation

Search for the value 75



BST - Search algorithm

contains

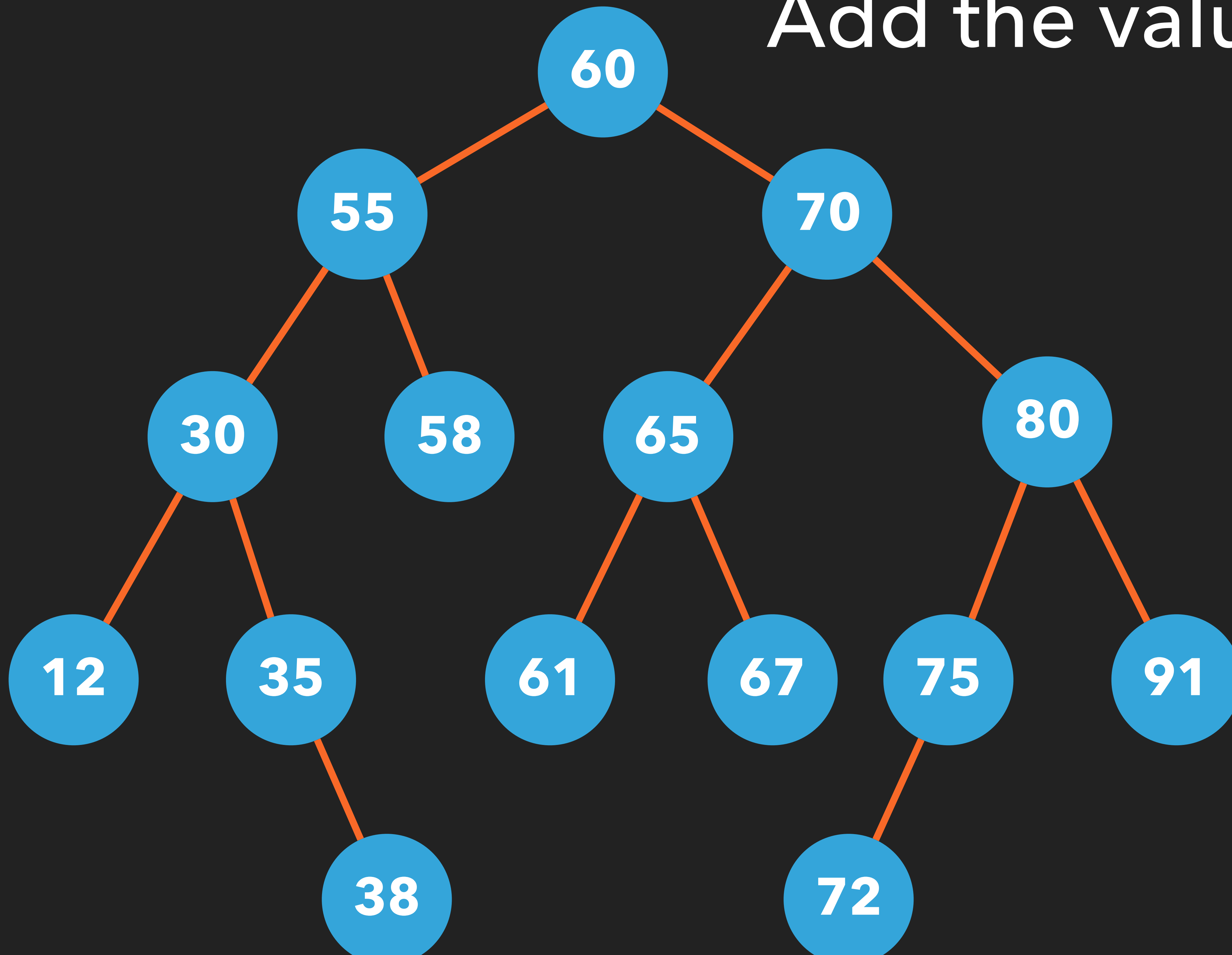
```
boolean contains (item)

  current node = root // start from the root
  while(current node is not null){
     if(the value of the current node == item)
        return true
     else if (value of the current node > item)
        current node is set to its left child
     else
        current node is set to its right child
  }
  return false

end contains
```

BST - Add operation

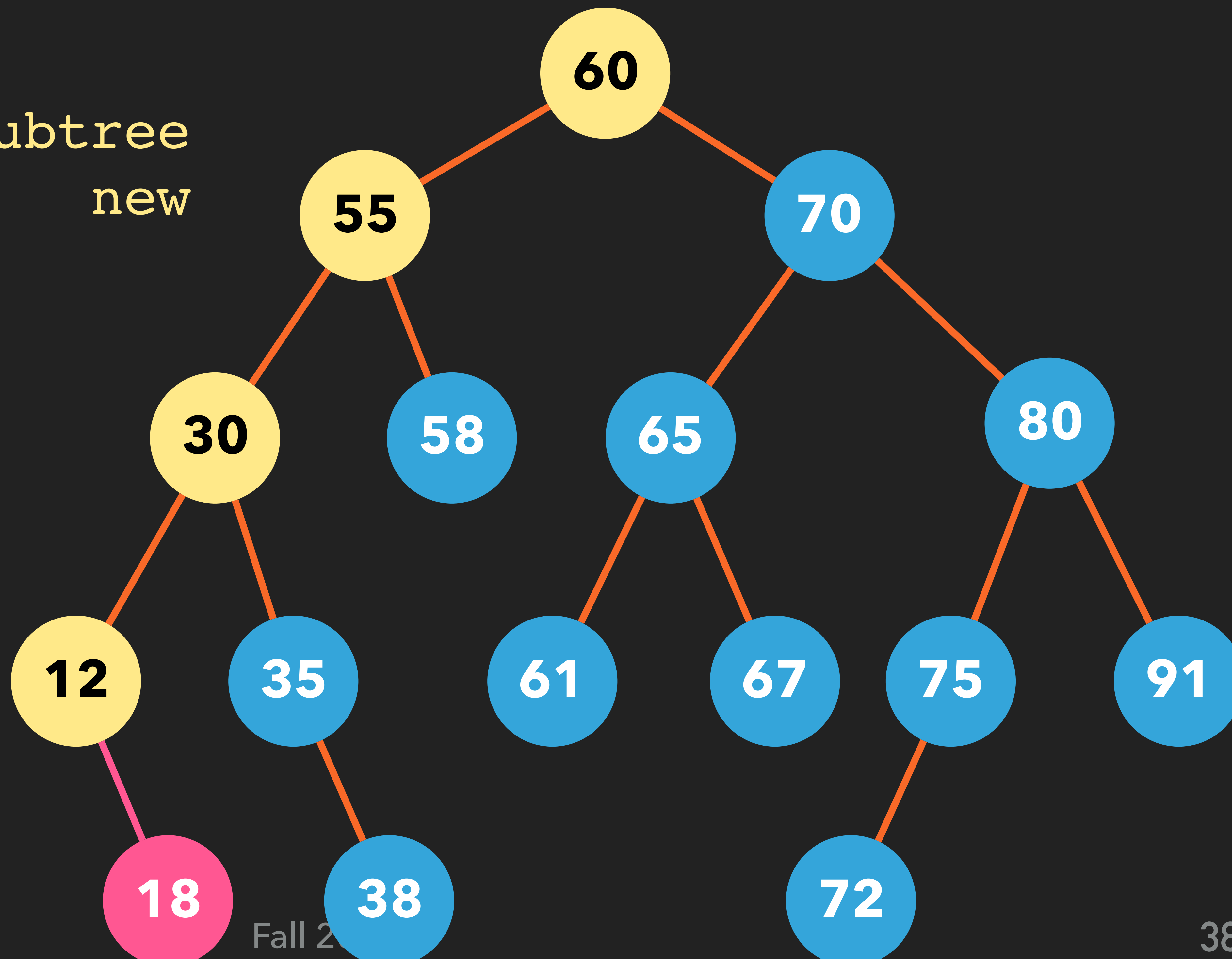
Add the value 18



BST - Add operation

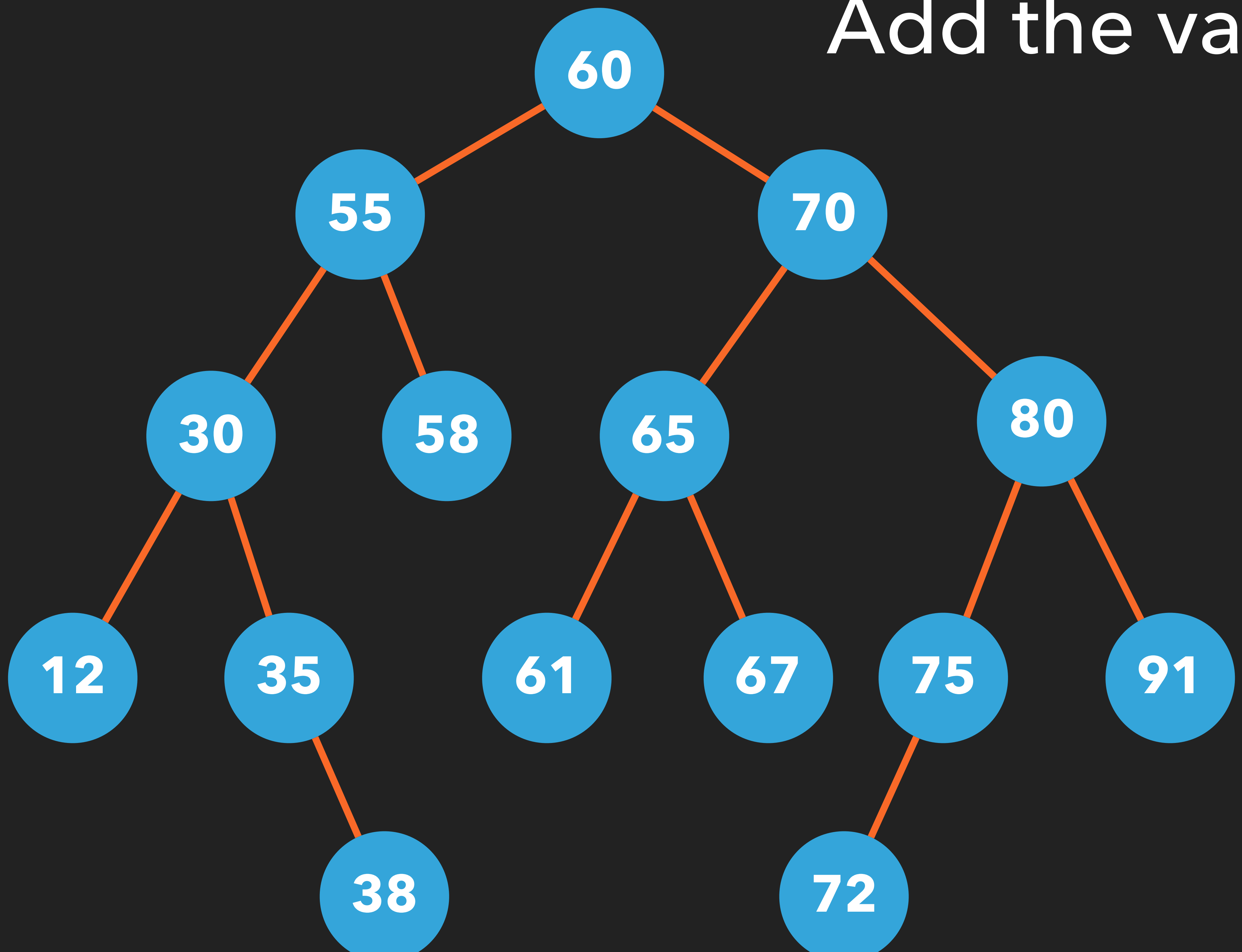
Add the value 18

Find the subtree where the new value fits



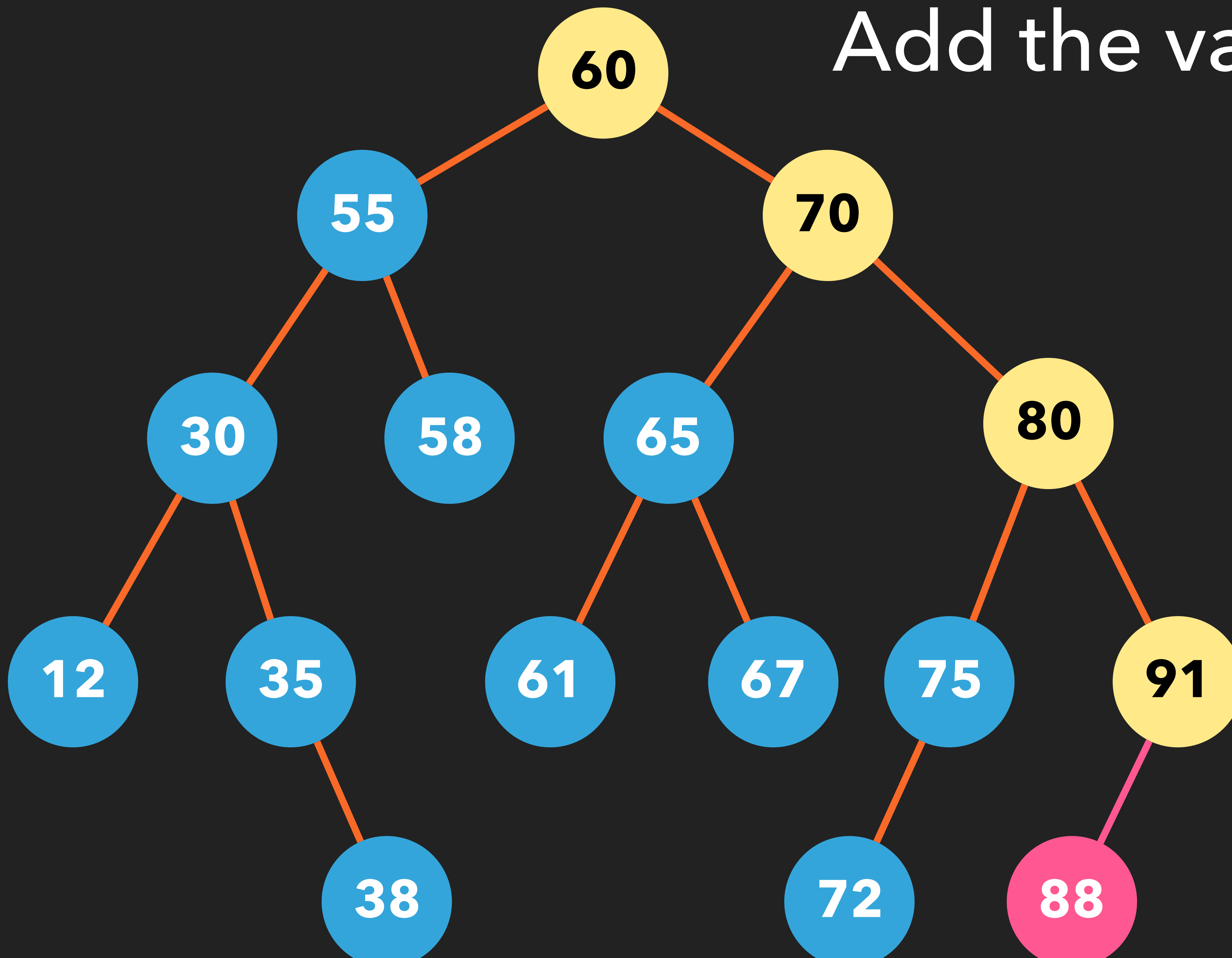
BST - Add operation

Add the value 88



BST - Add operation

Add the value 88



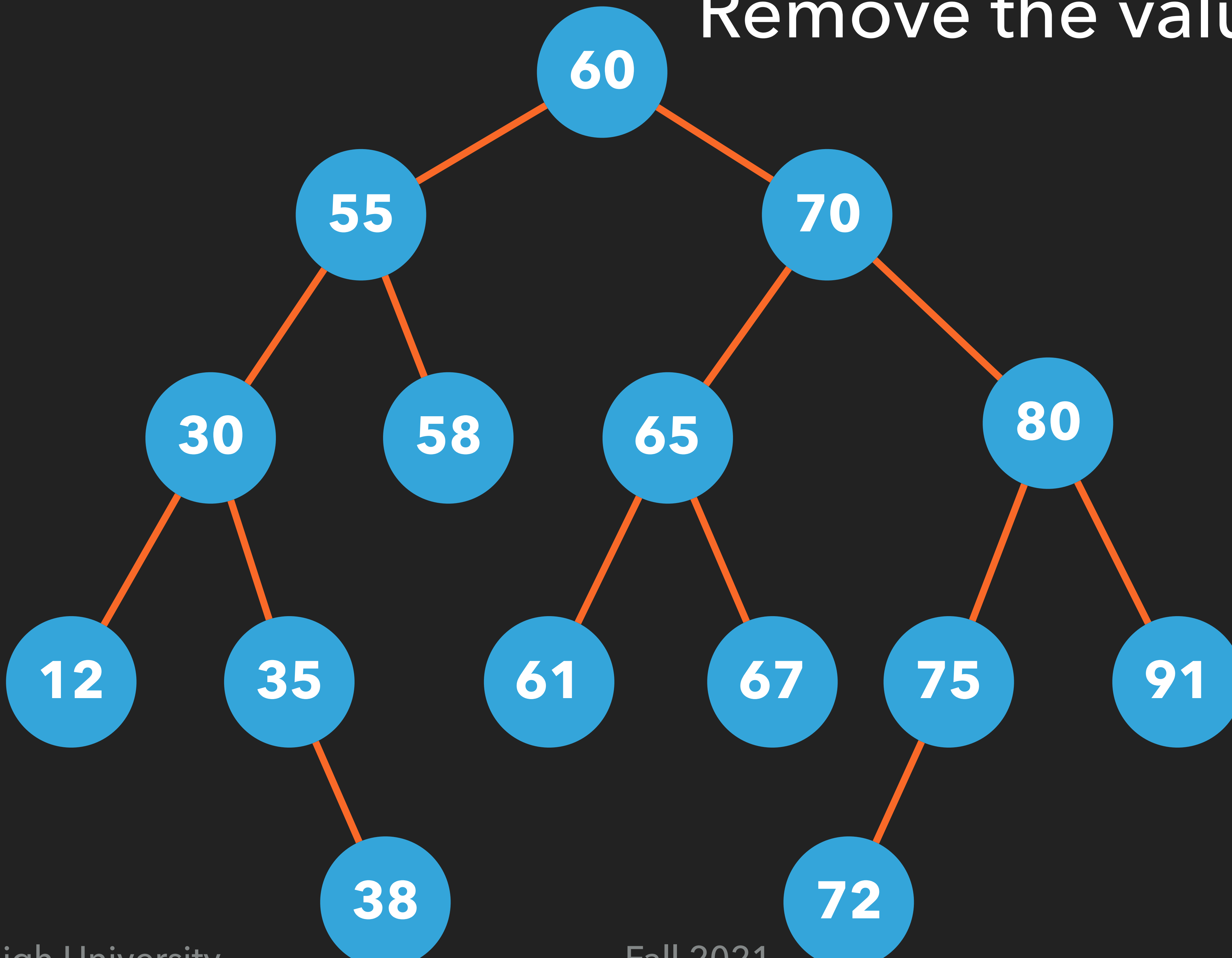
BST - Add operation

add

```
boolean add (item)
   current node = root
   while(current is not null){
     parent = current node
     if( the value of the current node == item)
        return false (duplicates are not allowed)
     else if (value of the current node > item)
        current is set to the left child
     else
        current node is set to the right child
   }
   if (the value of the parent node > item)
      Create a left child to parent (value=item)
   else
      Create a right child to parent (value=item)
   end if
   return true
end add
```

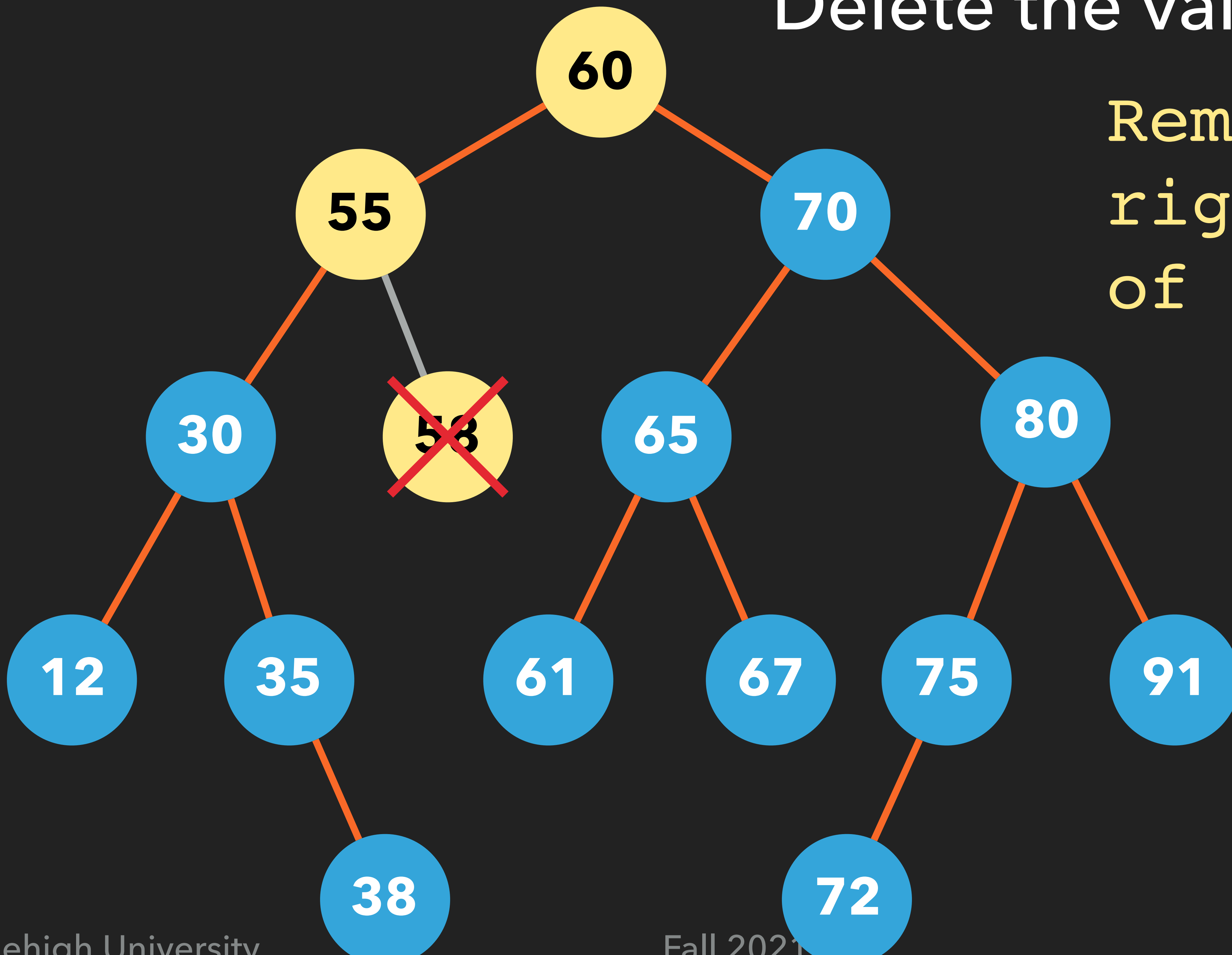
BST - Remove operation

Remove the value 58 (Leaf)



BST - Remove operation

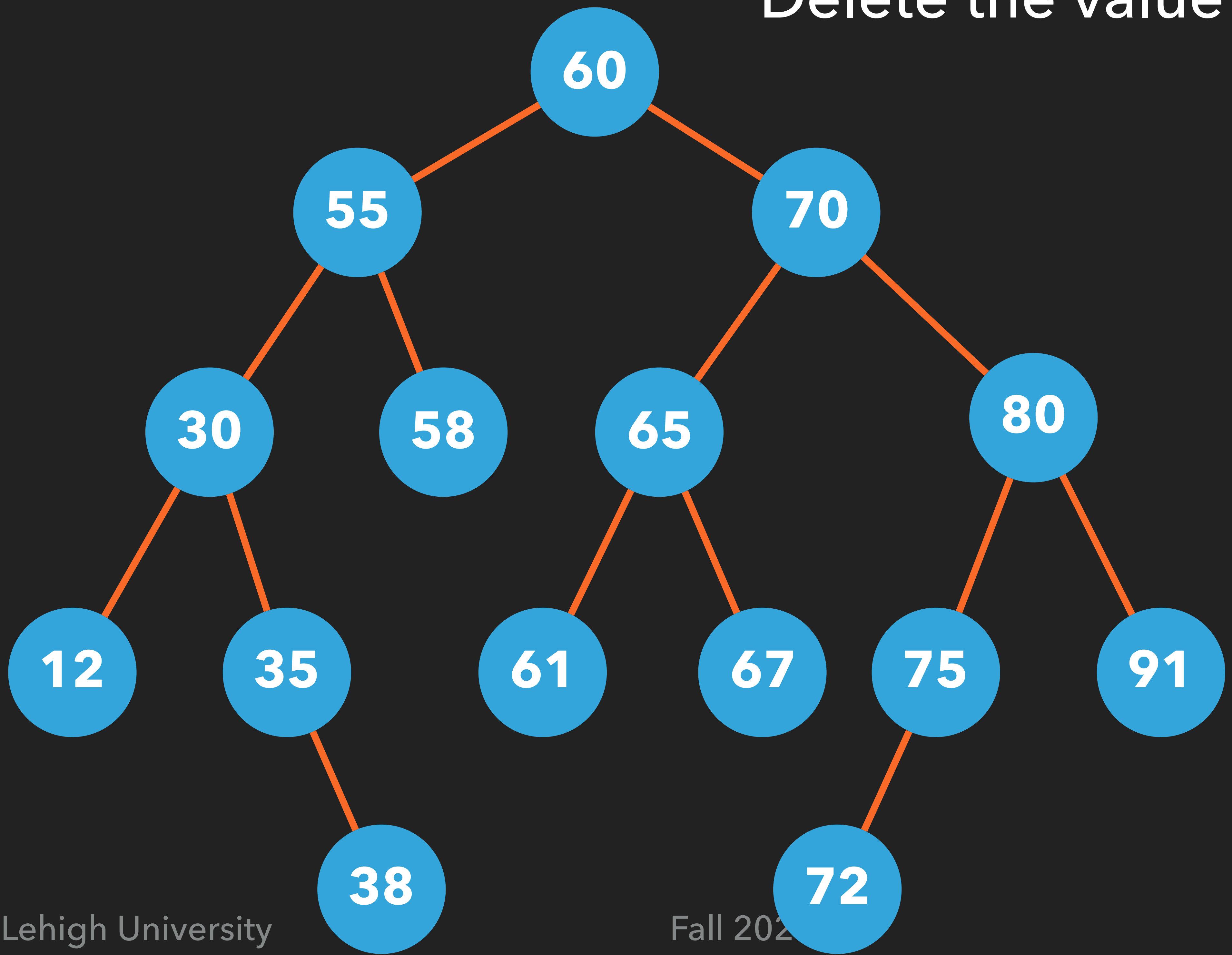
Delete the value 58 (Leaf)



`Remove the right child of 55`

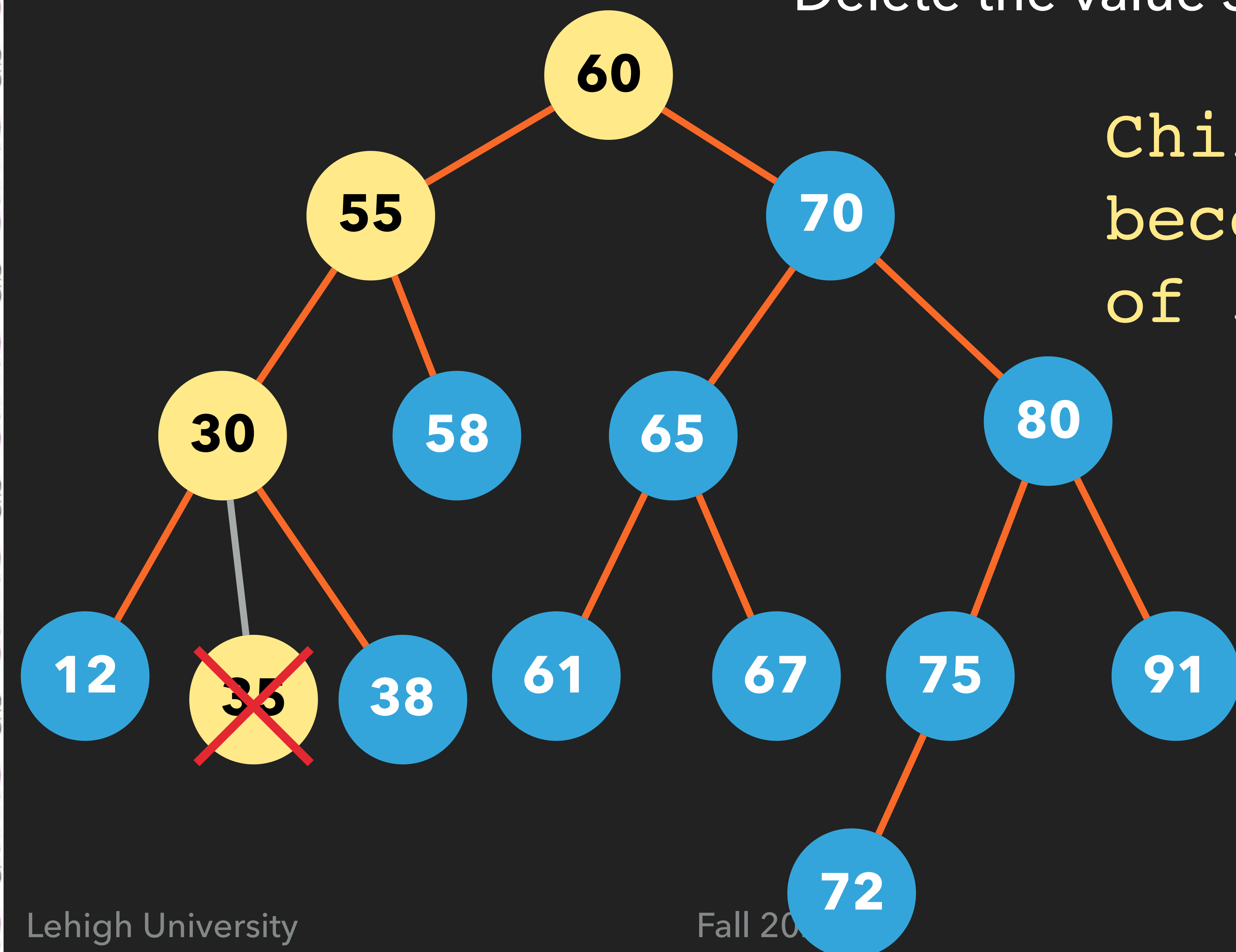
BST - Remove operation

Delete the value 35 (one child)



BST - Remove operation

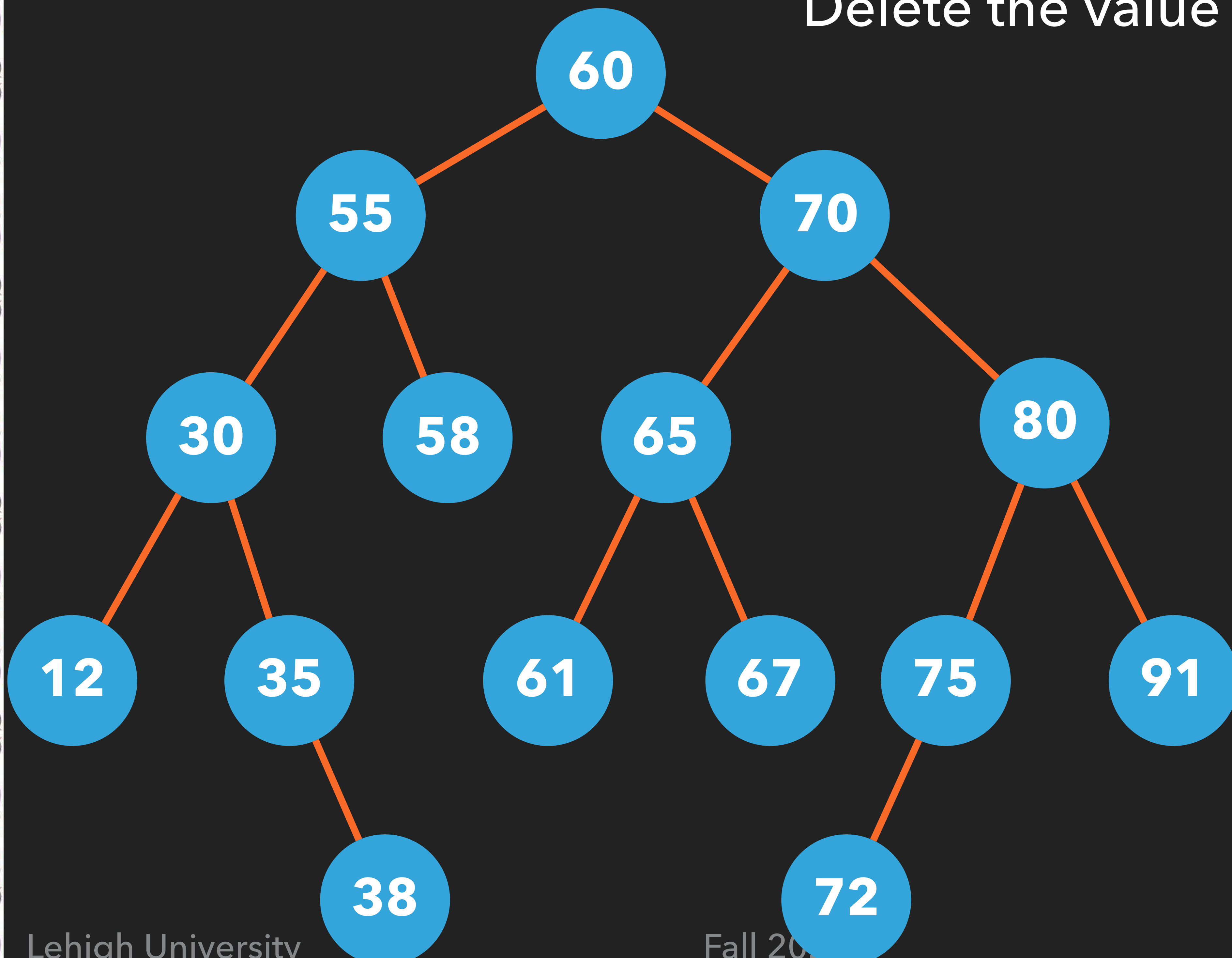
Delete the value 35 (has one child)



Child of 35 becomes child of 30

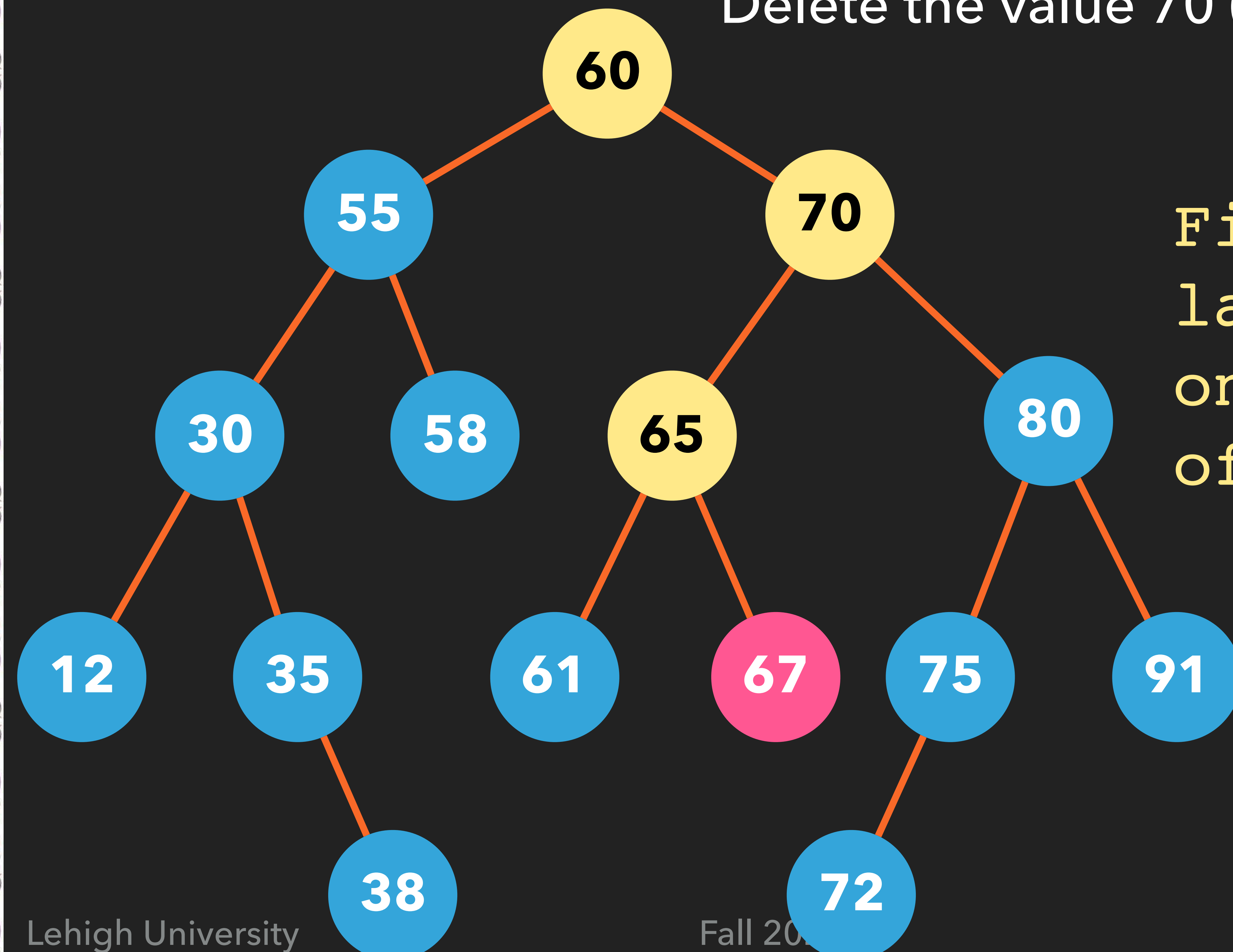
BST - Remove operation

Delete the value 70 (two children)



BST - Remove operation

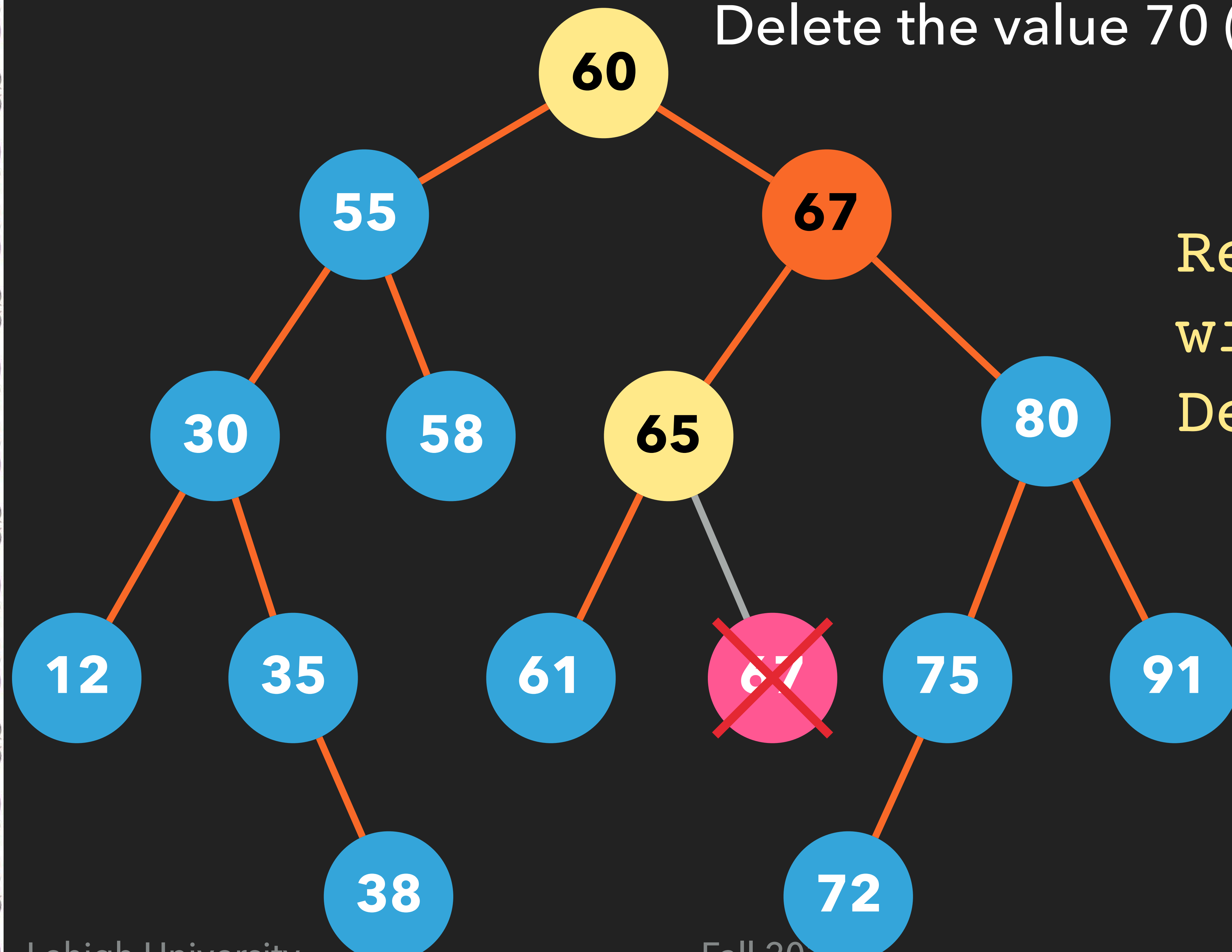
Delete the value 70 (has two children)



Find the largest node on the left of 70

BST - Remove operation

Delete the value 70 (has two children)




```
Replace 70
with 67 -
Delete 67
```

BST - Remove operation

remove

```
boolean remove (item)
  node = search(item) // find node with value item first
  if (node == null)
     return false (item not found in the BST)
  else
   if (node has no children)
      remove link to node (parent points to null)
   else if (node has one child)
      replace node with node’s child
   else if (node has two children)
      find the largest node on the left subtree of node
      copy the value of the largest node to node
      remove the largest node
   end if
  end if
  return true
end remove
```

Traversals

```
preorder(){
  preorder(root)}
preorder(node){
   print node
   preorder(left child of node)
   preorder(right child of node)}
```

```
inorder(){
  inorder(root) }
inorder(node){
   inorder(left child of node)
   Print node
   inorder(right child of node)}
```

```
postorder(){
  postorder(root)}
postorder(node){
   postorder(left child of node)
   postorder(right child of node)
   print node    }
```

BST - Implementation

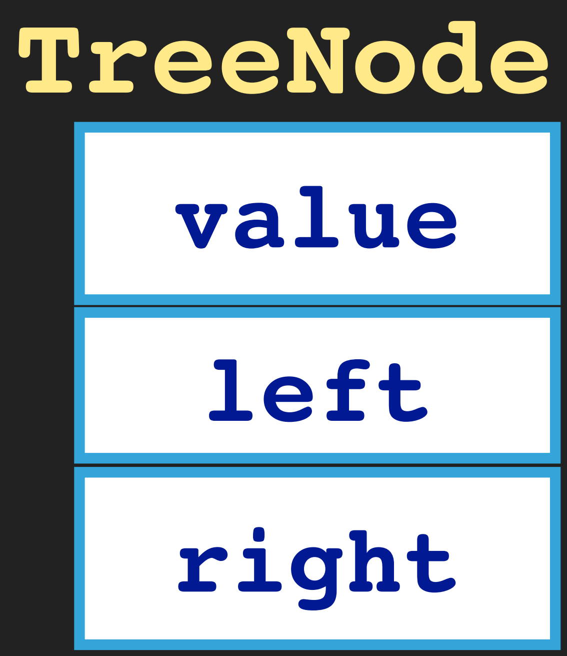
- BST may be implemented in two ways
  - Array Based BST
  - Linked BST

BST - Implementation

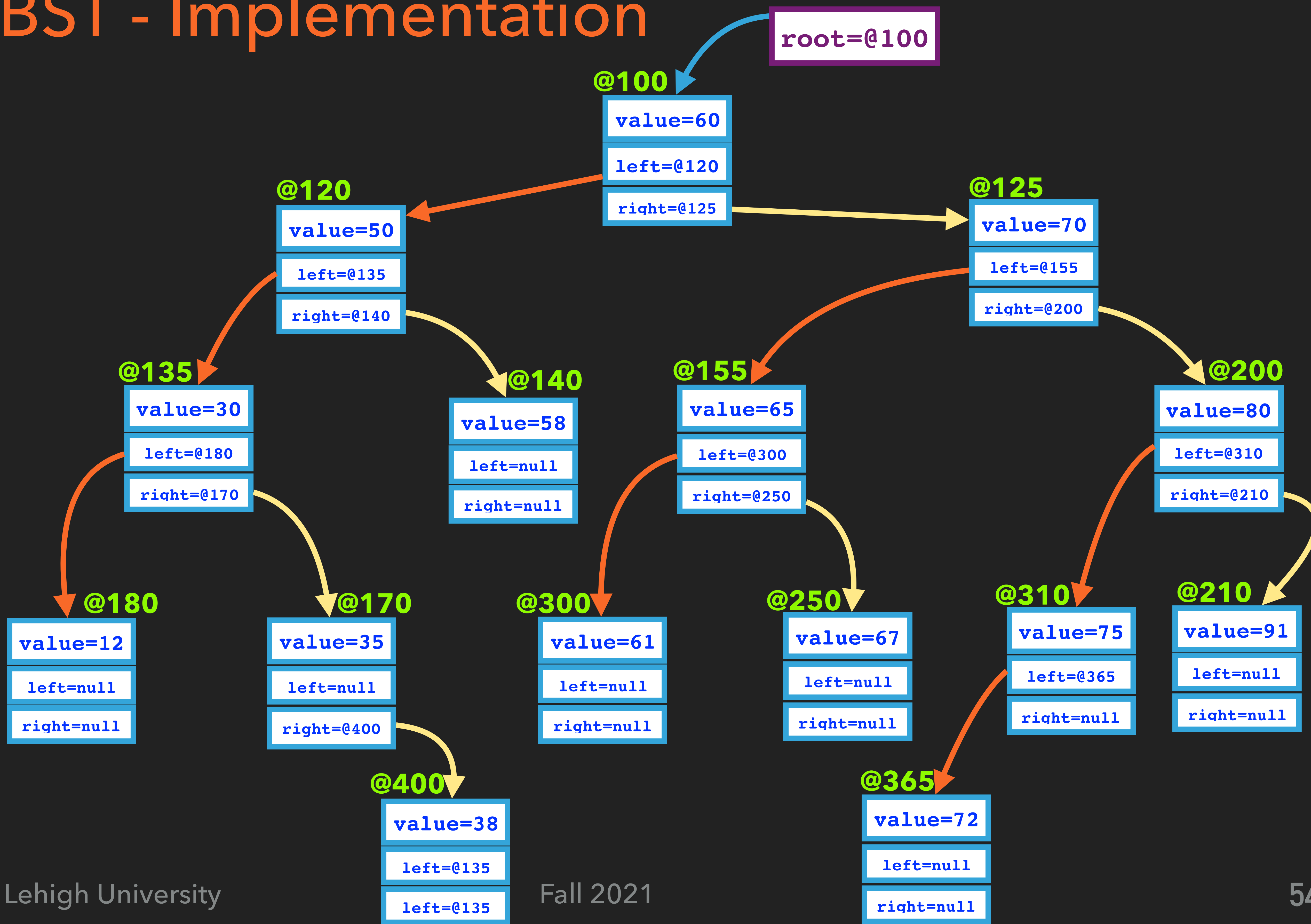
- Nodes of the tree are stored in an array
- Children of a node follow the node (at specific indices)

BST - Implementation

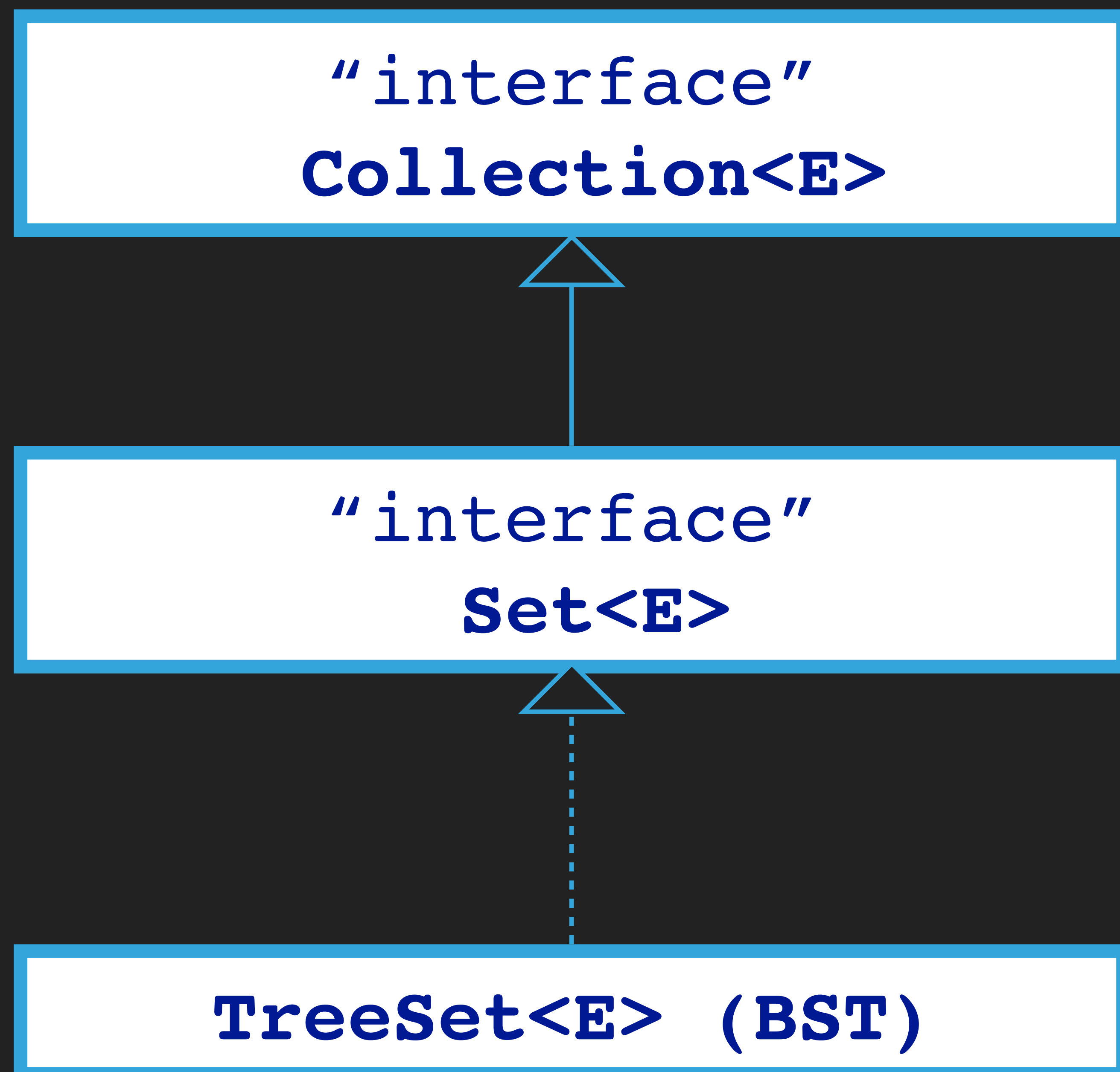
- Nodes of the tree are linked
- Every node has two references in addition to its value: left child and right child



BST - Implementation



BST - Implementation



BST - Implementation

has

```
BST<E extends Comparable<E>>
-----------------------------
-root: TreeNode
-size: int
-----------------------------
+BST()
+size(): int
+isEmpty(): boolean
+clear(): void
+contains(E): boolean
+add(E): boolean
+remove(E): boolean
+inorder(): void
+preorder(): void
+postorder(): void
```

```
TreeNode
-----------------------------
value: E
Left: TreeNode
Right: TreeNode
-----------------------------
TreeNode(E val)
```

BST - Implementation

BST.java

```
public class BST<E extends Comparable<E>> {
  private TreeNode root;
  private int size;

  private class TreeNode{
    E value;
    TreeNode left;
    TreeNode right;
    TreeNode(E val){
      value = val;
      left = right = null;
    }
  }
```

BST - Implementation

BST.java

```
BST(){
    root = null;
    size = 0;
}
public int size() {
   return size;
}
public boolean isEmpty() {
   return (size == 0);
}
public void clear() {
   root = null;
}
```

BST - Implementation

BST.java

```
// Search method
public boolean contains(E item) {
  TreeNode node = root;
  while (node != null) {
    if( item.compareTo(node.value) < 0)
      node = node.left;
    else if (item.compareTo(node.value)> 0)
      node = node.right;
    else
      return true;
  }
  return false;
}
```

BST.java

```
// Method add()
public boolean add(E item) {
   if (root == null) // first node to be inserted
      root = new TreeNode(item);
   else {
      TreeNode parent, node;
      parent = null; node = root;
      while (node != null) {// Looking for a leaf node
         parent = node;
         if(item.compareTo(node.value) < 0) {
            node = node.left; }
         else if (item.compareTo(node.value) > 0) {
            node = node.right; }
         else
            return false; // duplicates are not allowed
      }
      if (item.compareTo(parent.value)< 0)
         parent.left = new TreeNode(item);
      else
         parent.right = new TreeNode(item);
   }
   size++;
  return true;
}
```

BST - Implementation

BST.java

```
// Method remove()
public boolean remove(E item) {
  TreeNode parent, node;
  parent = null; node = root;
  // Find item first
  while (node != null) {
      if (item.compareTo(node.value) < 0) {
        parent = node;
        node = node.left;
      }
      else if (item.compareTo(node.value) > 0) {
        parent = node;
        node = node.right;
      }
      else
        break; // item found
  }
```

BST - Implementation

BST.java

```
if (node == null) // item not in the tree
     return false;

// Case 1: node has no children
if(node.left == null && node.right == null){
   if(parent == null){ // delete root
     root = null;
   }
   else{
     changeChild(parent, node, null);
   }
}
```

BST - Implementation

BST.java

```
// case 2: one right child
else if(node.left == null){
    if (parent == null){ // delete root
        root = node.right;
    }
    else{
        changeChild(parent, node, node.right);
    }
}
// case 2: one left child
else if(node.right == null){
    if (parent == null){ // delete root
        root = node.left;
    }
    else{
        changeChild(parent, node, node.left);
    }
}
```

BST - Implementation

BST.java

```
// Case 3: node has two children
else {
   TreeNode rightMostParent = node;
   TreeNode rightMost = node.left;
   // go right on the left subtree
   while (rightMost.right != null) {
      rightMostParent = rightMost;
      rightMost = rightMost.right;
   }
   // copy the value of rigthMost to node
   node.value = rightMost.value;
   //delete rigthMost
      changeChild(rightMostParent, rightMost,
                  rightMost.left);
}
size--;
return true;
}
```

BST - Implementation

BST.java

```
private void changeChild(TreeNode parent,
        TreeNode node, TreeNode newChild){
    if(parent.left == node)
      parent.left = newChild;
    else
      parent.right = newChild;
  }
```

BST - Implementation

BST.java

```
// Recursive method inorder()
public void inorder() {
  inorder(root);
}
private void inorder(TreeNode node) {
  if (node != null) {
    inorder(node.left);
    System.out.print(node.value + " ");
    inorder(node.right);
  }
}
```

BST - Implementation

BST.java

```
// Recursive method preorder()
public void preorder() {
  preorder(root);
}
private void preorder(TreeNode node) {
  if (node != null) {
    System.out.print(node.value + " ");
    preorder(node.left);
    preorder(node.right);
  }
}
```

BST - Implementation

BST.java

```
// Recursive method postorder()
public void postorder() {
  postorder(root);
}
private void postorder(TreeNode node)  {
  if (node != null) {
    postorder(node.left);
    postorder(node.right);
    System.out.print(node.value + " ");
  }
}
```

BST - Testing

Test.java

```
// Testing the class BST
   public static void main(String[] args){
      BST<String> bst = new BST<>();
      bst.add("Kiwi");
      bst.add("Strawberry");
      bst.add("Apple");
      bst.add("Banana");
      bst.add("Orange");
      bst.add("Lemon");
      bst.add("Watermelon");
      bst.inorder();
      bst.remove("Banana");
      System.out.println(bst.contains("Banana"));
      bst.inorder();
      bst.remove("Orange");
      bst.inorder();
      bst.remove(“Kiwi");
      bst.inorder();
   }
```

BST - Implementation

- Performance of the BST operations

Method	Complexity
`BST()`	`O(1)`
`size()`	`O(1)`
`clear()`	`O(1)`
`isEmpty()`	`O(1)`
`contains(E)`	`O(n) / O(log n)`
`add(E)`	`O(n) / O(log n)`
`remove(E)`	`O(n) / O(log n)`
`inorder()`	`O(n)`
`preorder()`	`O(n)`
`postorder()`	`O(n)`

BST - Testing

Test.java

```
// Testing the class BST
   public static void main(String[] args){
      BST<String> bst = new BST<>();
      bst.add(“Apple");
      bst.add("Banana");
      bst.add(“Kiwi");
      bst.add(“Lemon");
      bst.add("Orange");
      bst.add("Strawberry");
      bst.add("Watermelon");
      bst.inorder();
   }
```

Order of the added values affects the shape of the tree

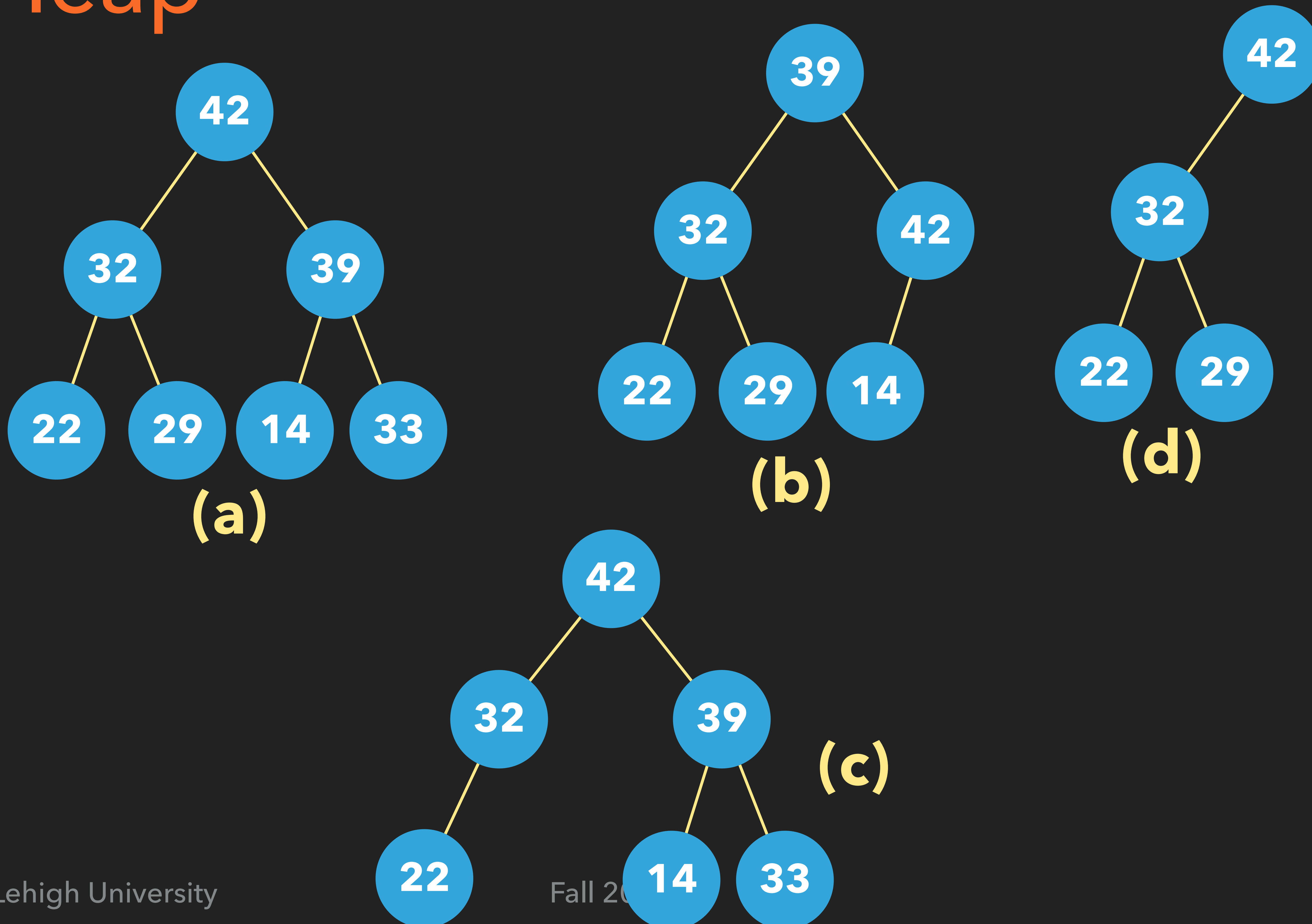
Summary

- Binary trees and binary search trees
- Operations: Search, Add, Remove, Traversals
- Implementation - Linked Nodes
- Order of add operations has an effect on the shape of the BST

Heap

- Special binary tree
  - Complete binary tree - All the levels are filled except the last level
    All leaves on the last level are placed leftmost
  - Every node is greater than or equal to any of its children (Max Heap) [Min Heap: less than or equal]
  - Used for efficient sorting

Heap

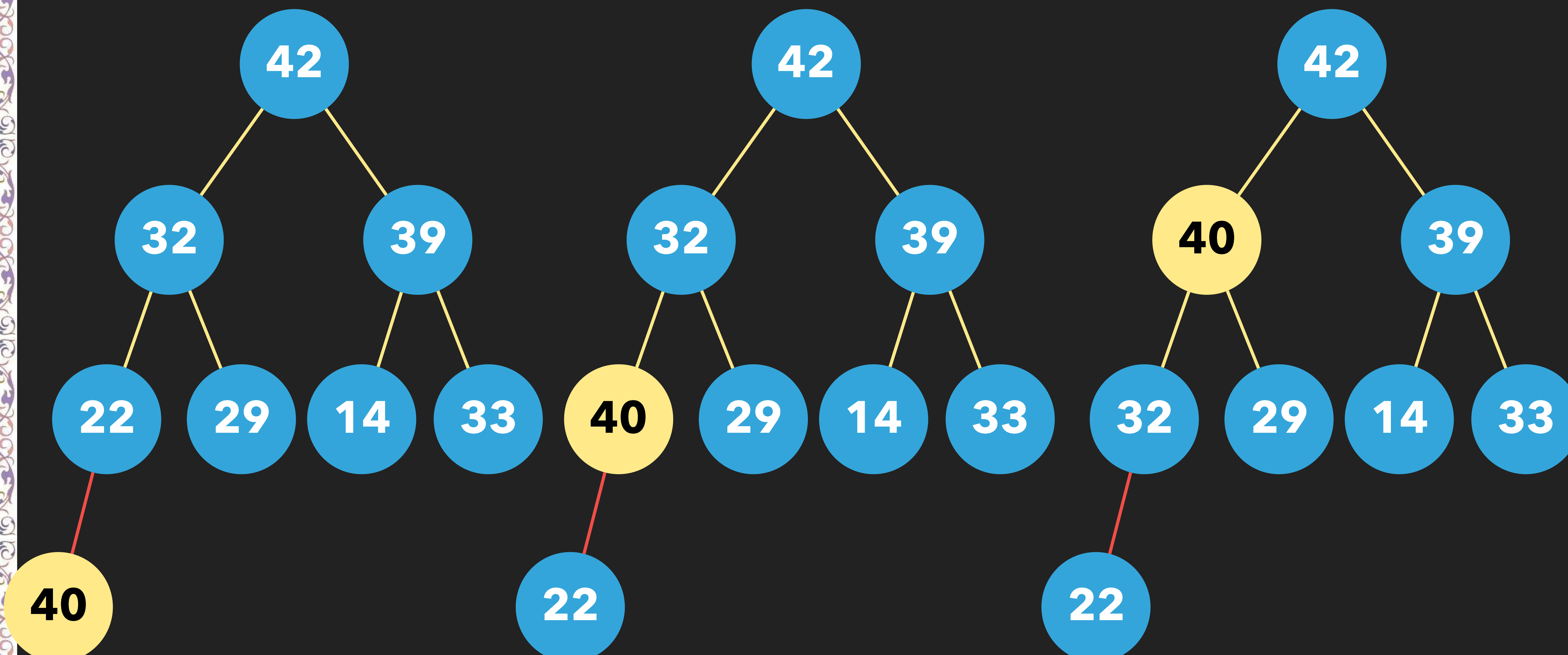

42
32
39
22
29
14
33
(a)
39
32
42
22
29
14
(b)
42
32
22
29
(d)
42
32
39
22
14
33
(c)

Heap

- Two main operations on the Heap
  - Adding a new node while keeping the heap properties
  - Removing a node while keeping the heap properties

Heap

- Adding a new node to the heap (40)



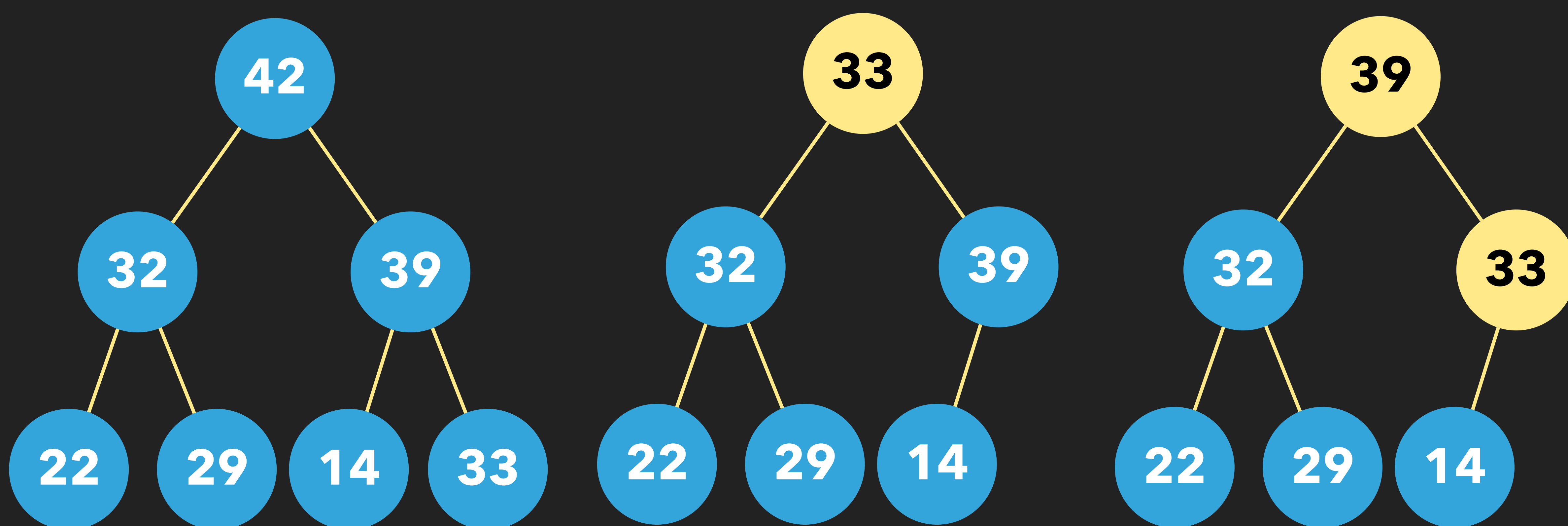
Heap

- Adding a new node to the heap

```
Algorithm add
  Add the new node at the end of the heap
  Current node = added node
  While (current node > its parent)
    Swap current node with its parent
    Current node becomes the parent
End
```

Heap

- Removing a node from the heap (42)



Heap

- Removing a node from the heap (root)

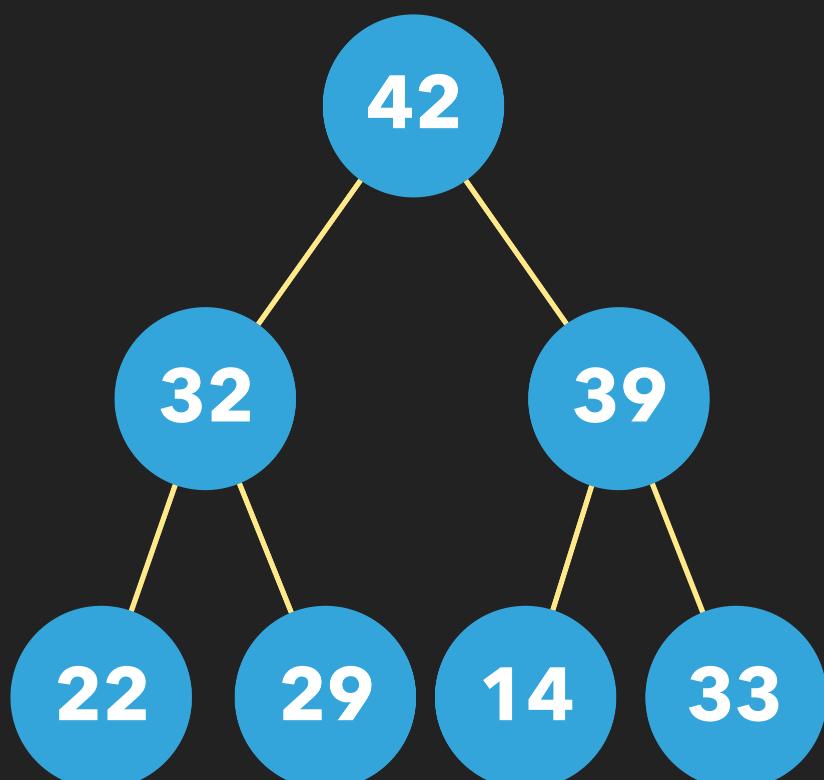
```
Algorithm remove
  Move the last node to replace the root
  Current node = root
  While (current node < its children)
    Swap current node with the largest
    of its children
    Current node becomes the largest child
End
```

Heap

- Heap implementation
  - ArrayList to store the heap nodes
  - Easy access to children and parent

Heap

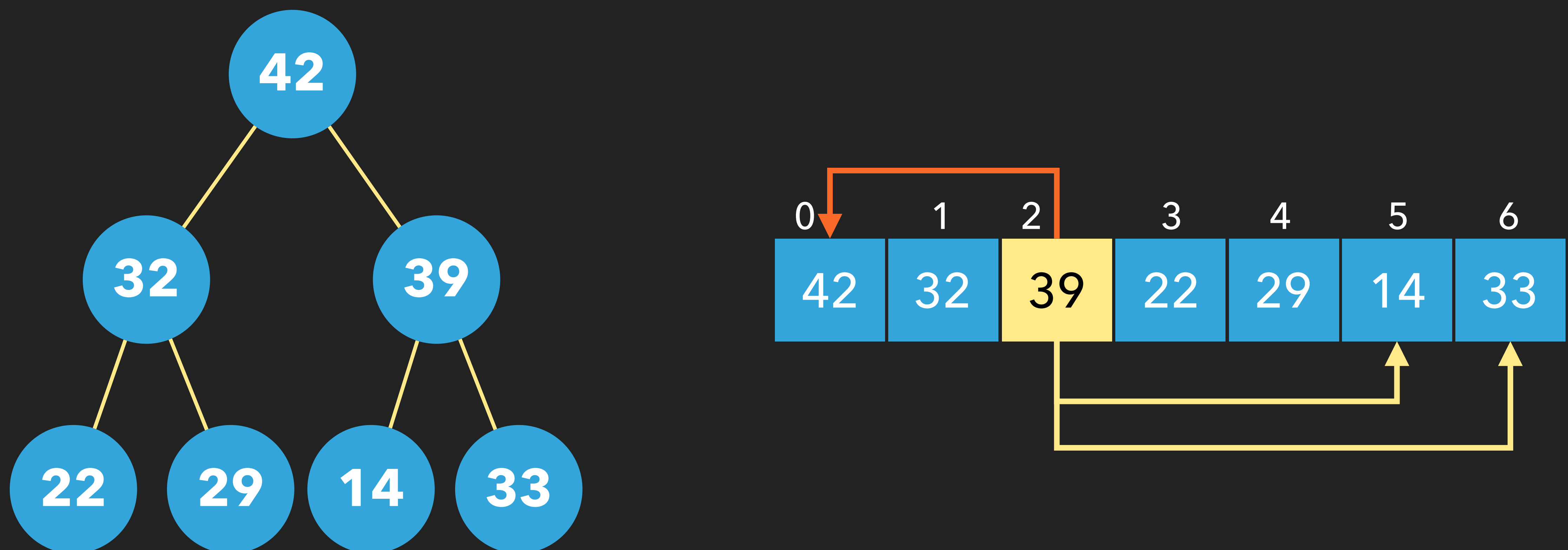
- Heap implementation



0	1	2	3	4	5	6
42	32	39	22	29	14	33

Heap

- Heap data structure




```
IndexOf(Parent) = (IndexOf(current) - 1) / 2

IndexOf(Left child) =  2 * IndexOf(current) + 1
IndexOf(Right child) = 2 * IndexOf(current) + 2
```

Heap

Heap<E extends Comparable<E>>
-list: ArrayList<E>
+Heap() +add(E): void +remove(): E +contains(E): boolean +size(): int +isEmpty(): boolean +clear(): void +toString(): String

Heap

Heap.java

```
public class Heap<E extends Comparable<E>> {
  private ArrayList<E> list;
  public Heap(){
    list = new ArrayList<>();
  }
  public int size(){
    return list.size();
  }
  public boolean isEmpty(){
    return list.isEmpty();
  }
  public void clear(){
     list.clear();
  }
  public String toString(){
    return list.toString();
  }
```

Heap

Heap.java

```
public boolean contains(E item) {
    for(int i=0; i<list.size(); i++) {
      if(list.get(i).equals(item))
        return true;
    }
    return false;
 }
```

Heap

Heap.java

```
public void add(E item) {
    list.add(item); //append item to the heap
    int currentIndex = list.size()-1;
    //index of the last element
    while(currentIndex > 0) {
      int parentIndex = (currentIndex-1)/2;
      //swap if current is greater than its parent
        E current = list.get(currentIndex);
        E parent = list.get(parentIndex);
      if(current.compareTo(parent) > 0) {
        list.set(currentIndex, parent);
        list.set(parentIndex, current);
      }
      else
        break; // the tree is a heap
      currentIndex = parentIndex;
    }
}
```

Heap

Heap.java

```
public E remove() {
  if(list.size() == 0) return null;
 //copy the value of the last node to root
  E removedItem = list.get(0);
  list.set(0, list.get(list.size()-1));
 //remove the last node from the heap
  list.remove(list.size()-1);
  int currentIndex = 0;
```

Heap

Heap.java

```
while (currentIndex < list.size()) {
  int left = 2 * currentIndex + 1;
  int right = 2 * currentIndex + 2;
  //find the maximum of the left and right nodes
  if (left >= list.size())
      break; // no left child
  int maxIndex = left;
  E max = list.get(maxIndex);
  if (right < list.size()) // right child exists
    if(max.compareTo(list.get(right)) < 0)
        maxIndex = right;
```

Heap

Heap.java

```
    // swap if current is less than max
    E current = list.get(currentIndex);
    max = list.get(maxIndex);
    if(current.compareTo(max) < 0){
      list.set(maxIndex, current);
      list.set(currentIndex, max);
      currentIndex = maxIndex;
    }
    else
      break; // the tree is a heap
  }
  return removedItem;
}
```

Heap

Test.java

```
public class TestBST {
  public static void main(String[] args) {
    Heap<String> heap = new Heap<>();
    heap.add(“Apple”);
    heap.add("Banana");
    heap.add(“Kiwi");
    heap.add(“Lemon");
    heap.add("Orange");
    heap.add("Strawberry");
    heap.add("Watermelon");
    System.out.println(“Heap: “ + heap.toString());
    System.out.pritnln(“Removed: “ + heap.remove());
    System.out.println(“Heap: “ + heap.toString());
    System.out.println(“Heap contains Pear?: “
                      + heap.contains(“Pear”));
  }
}
```

Heap

- Performance of the Heap operations

Method	Complexity
`Heap()`	**O(1)**
`size()`	**O(1)**
`clear()`	**O(1)**
`isEmpty()`	**O(1)**
`add(E)`	**O(log n)**
`remove(E)`	**O(log n)**
`contains(E)`	**O(n)**
`toString()`	**O(n)**

Summary

- Binary trees - Special binary trees: BST, Heap
- Operations: Search, Add, Remove, Traversals
- Implementation - Linked Nodes for BST, ArrayList for Heap
- Performance of the operations on BST/Heap
- BST height determined by the order of insertion of the nodes
- Heap is a balanced binary tree
  (height = `log`(number of nodes))