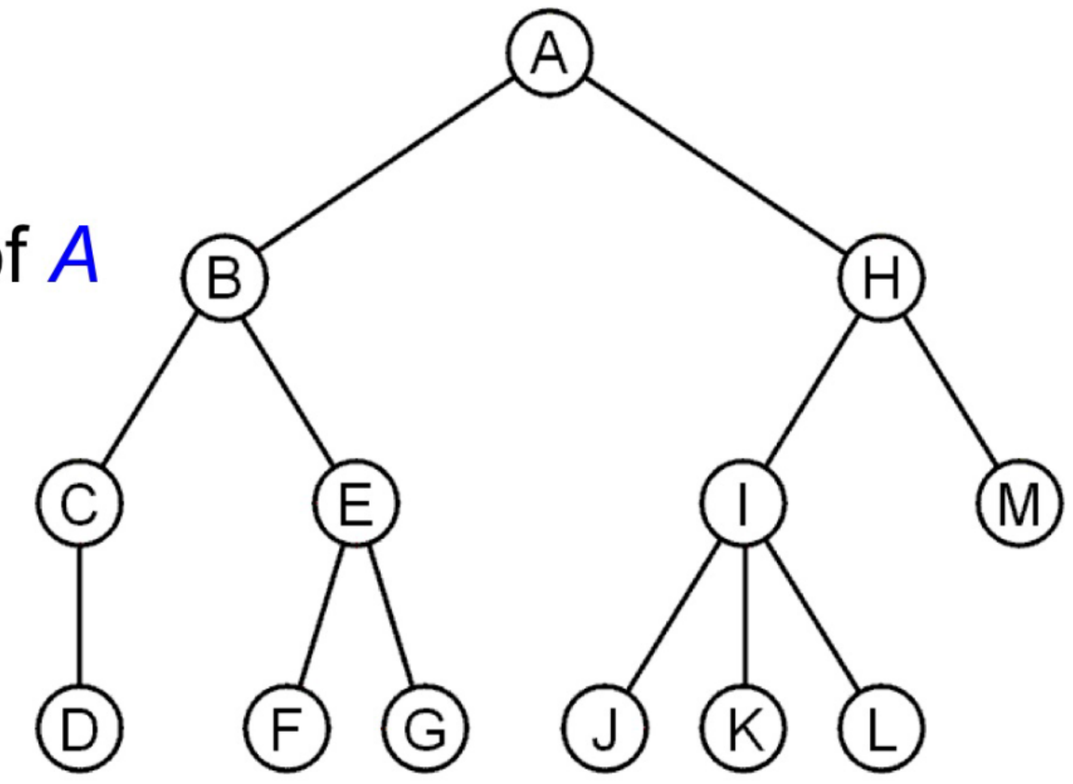
COL106
Data Structures and Algorithms

Subodh Sharma and Rahul Garg

Binary Search Trees

Recall Trees

- Collection of nodes with parent-child relationship
- *A* is the *root* node
- *B* is *parent* of C & E
- *B* is *ancestor* of C, E, D, F, G
- D, C and G are *descendants* of *A*
- *C* is the *sibling* of E
- *I* and *M* are the *children* of H
- *D, F, G, J, K, L, M* are *leaves*



Recall Tree Traversals

- Pre-order traversal: Processes a node before processing its children
- Post-order traversal: Processes a node after processing its children
- In-order traversal: Process left subtree followed by processing the node and then the right sub-tree
- Euler tour traversal: Traverse around the tree visiting each node thrice
- Breath-first traversal: Processes all the nodes at a level from left to right beginning at the root
- Traversals using recursion, stacks, queues

Forest

- A collection of trees is called a forest
- A tree can be converted into a forest by removing the root
- A forest may be converted into a tree by adding a root node and make all the “tree-roots” its children

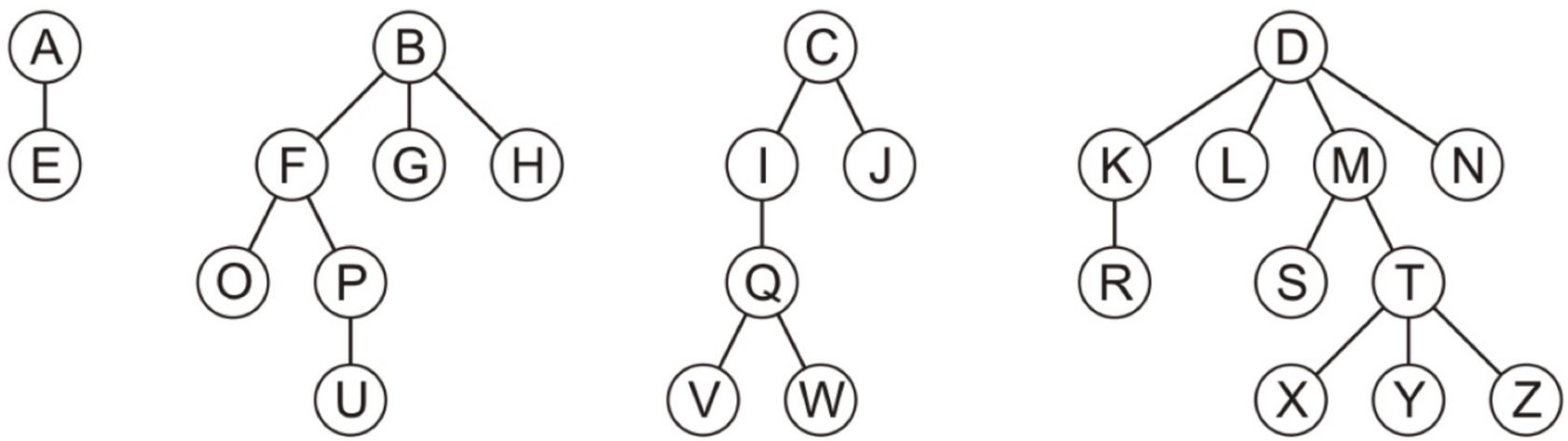


Figure courtesy: Douglas Wilhelm Harder, University of Waterloo, Canada

Dictionaries using Binary Trees

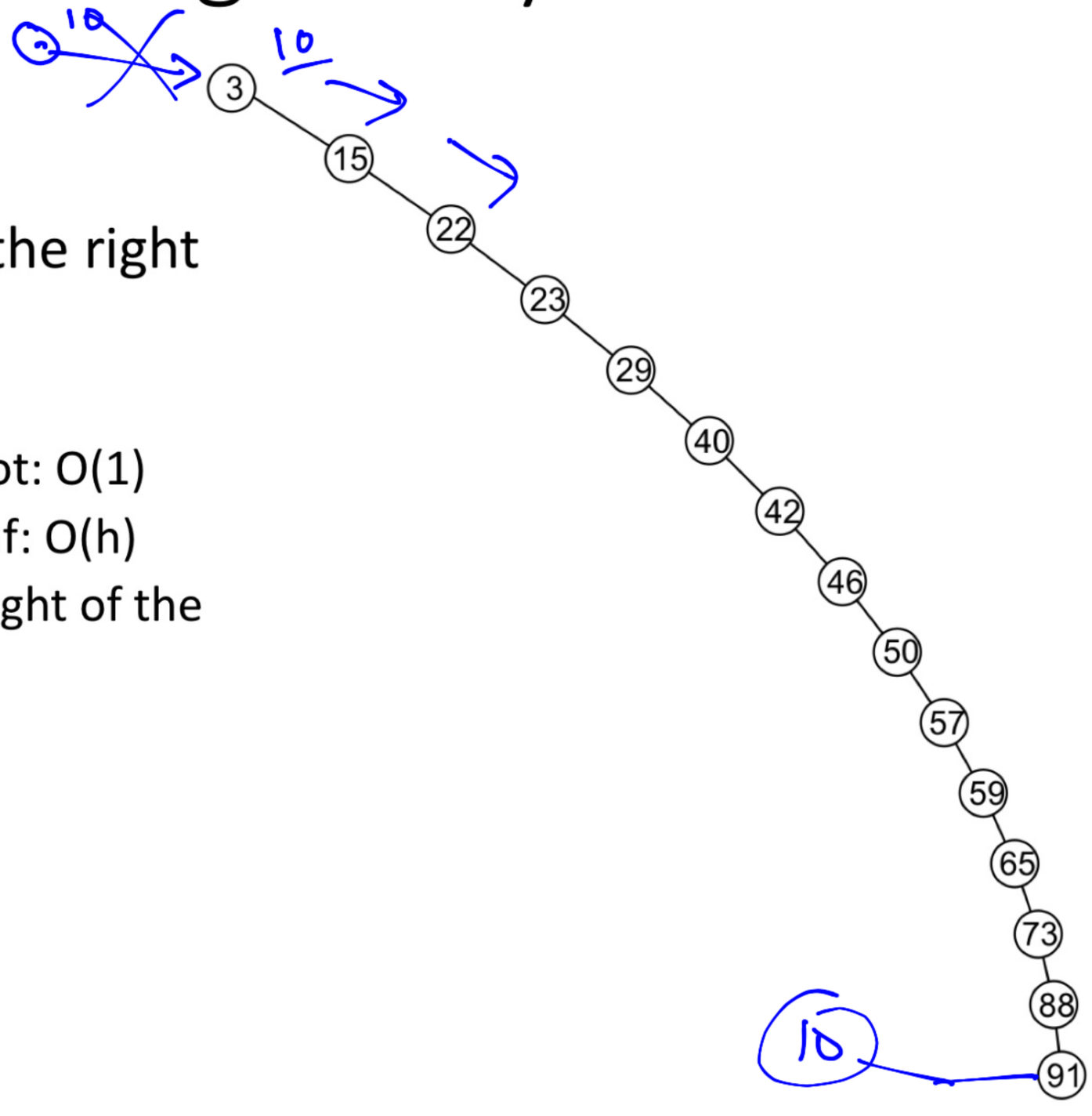
- Insert
- Find
- Remove

Dictionaries using Binary Trees

- Insertion
  - Always insert in the right subtree
  - Two cases:
    - Insert at the root: O(1)
    - Insert at the leaf: O(h)

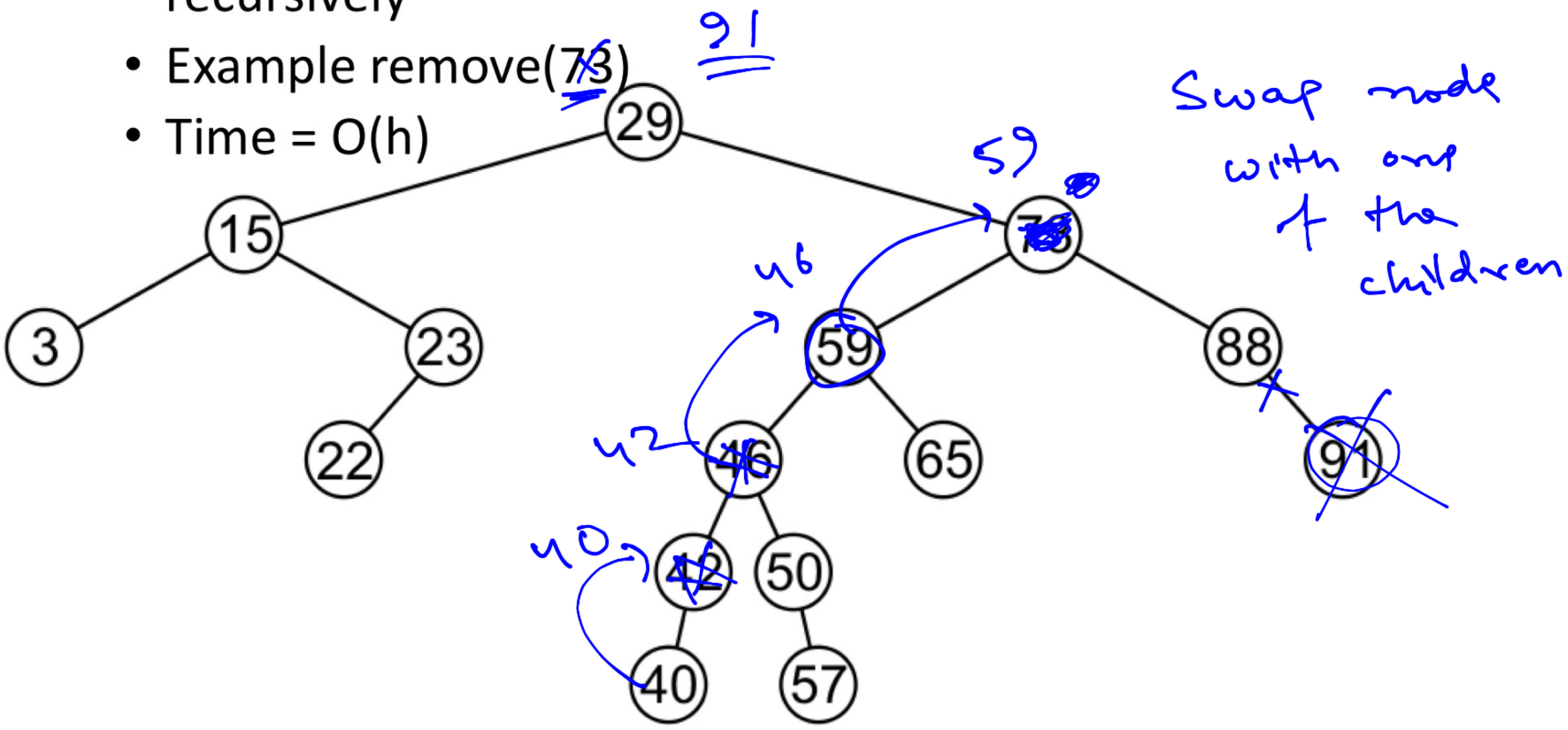
    where h is the height of the tree
    - h = O(N)

Insert(10)



Dictionaries using Binary Trees

- Remove
  - Remove the node and promote one of the child recursively
  - Example remove(73)
  - Time = O(h)



Dictionaries using Binary Trees

- Find
  - Use any tree traversal to find the given key
  - Time complexity O(N)

Dictionaries using Binary Trees

- Insert: O(1) or O(h)
- Find: O(N) after locating the key.
- Remove: O(h)
- h can be O(N) in the worst case
- How can we reduce the height of the tree?

Balanced Binary Trees

- Ensure that the left and right subtrees are always "balanced"
- Two types of maintaining balance
  - Height balanced trees
  - For all u, | height(u->left) – height(u->right) | ≤ 1
  - Weight balanced trees
  - For all u, | count(u->left) – count(u->right) | ≤ 1

Insertion in a Height Balanced BST

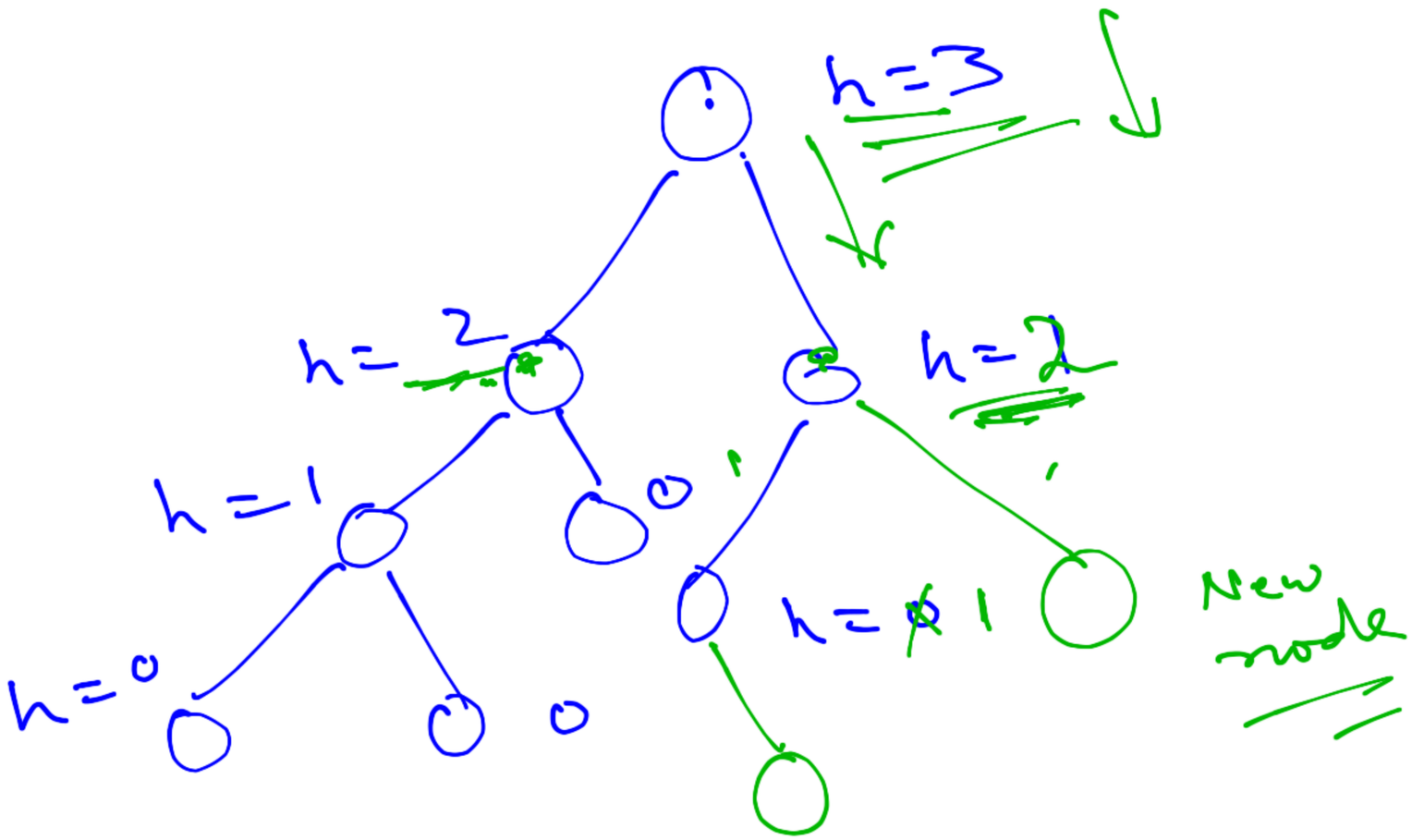
Insert(T, u)

Maintain an additional variable height at every node

- If any of the child is missing, insert there and update height of T
- Else insert in the subtree with smaller height
  - No need to update height of T. Why?

- If both subtrees have the same height
  Insert in any subtree
  Update height of T.

Insertion Example



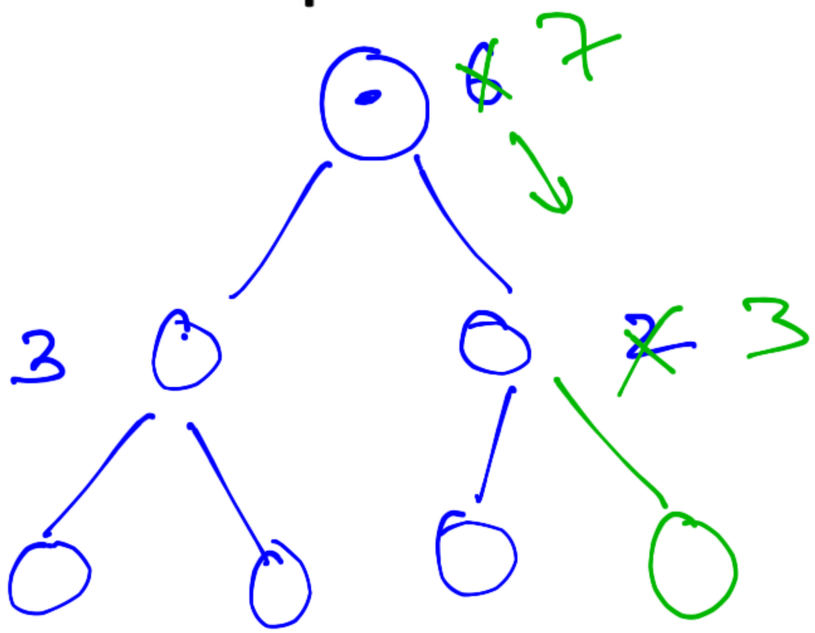
Insertion in Weight Balanced BST

Insert(T, u)

Maintain an additional variable weight at every node

- If any of the child is missing, insert there and update weight of T
- Else insert in the subtree with smaller weight and update weight of T

Insertion Example



Balanced Binary Trees

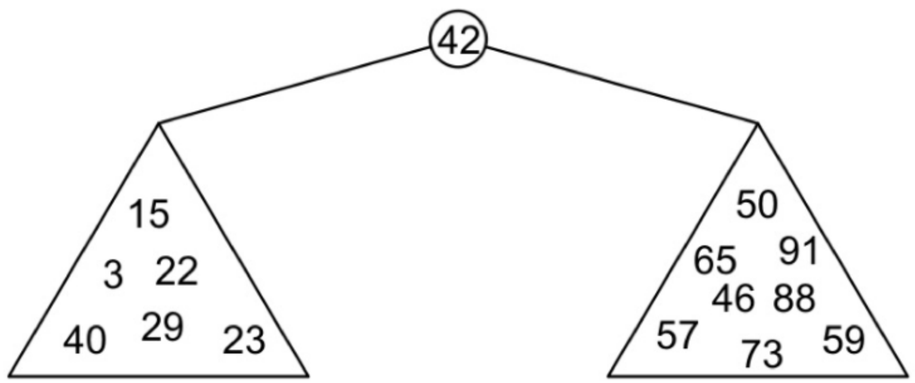
Homework: Prove that the maximum height of a balanced (height or weight) tree with N nodes will be O(log(N))

Balanced Binary Trees

- Insert: O(h)
- Find: O(N)
- Remove: O(h)
- Now h = O(log(N))
- How do we reduce the find time?
- Use Binary Search Tree

Binary Search Trees

- A binary search tree is a binary tree T indexed by key k such that
  - (Key, value) pairs are stored at every node v
  - Keys stored at nodes in the left subtree of every node *v* are less than or equal to *k*
  - Keys stored at nodes in the right subtree of *v* are greater than or equal to *k*



Binary Search Tree Applications

- Sorting
- Ordered dictionaries
- Others...

Sorting

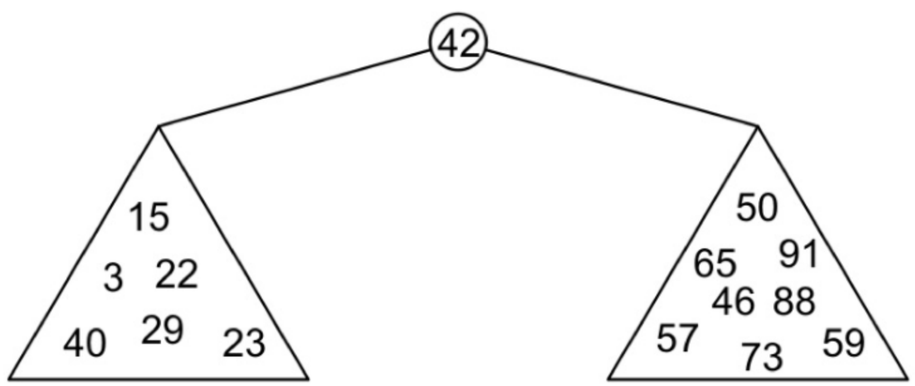
- Arrays
  - Find: O(log(n))
  - Insert: O(n)
  - Remove: O(n)
- Linked lists
  - Find: O(n)
  - Insert: O(1)
  - Remove: O(n)
- Hashing
  - Find: O(1)
  - Insert: O(1)
  - Remove: O(1)
  - Collisions, hash table size to be known in advance
- Balanced Binary Search Trees
  - Find: O(log(n))
  - Insert: O(log(n))
  - Remove: O(log(n))

Ordered Dictionaries

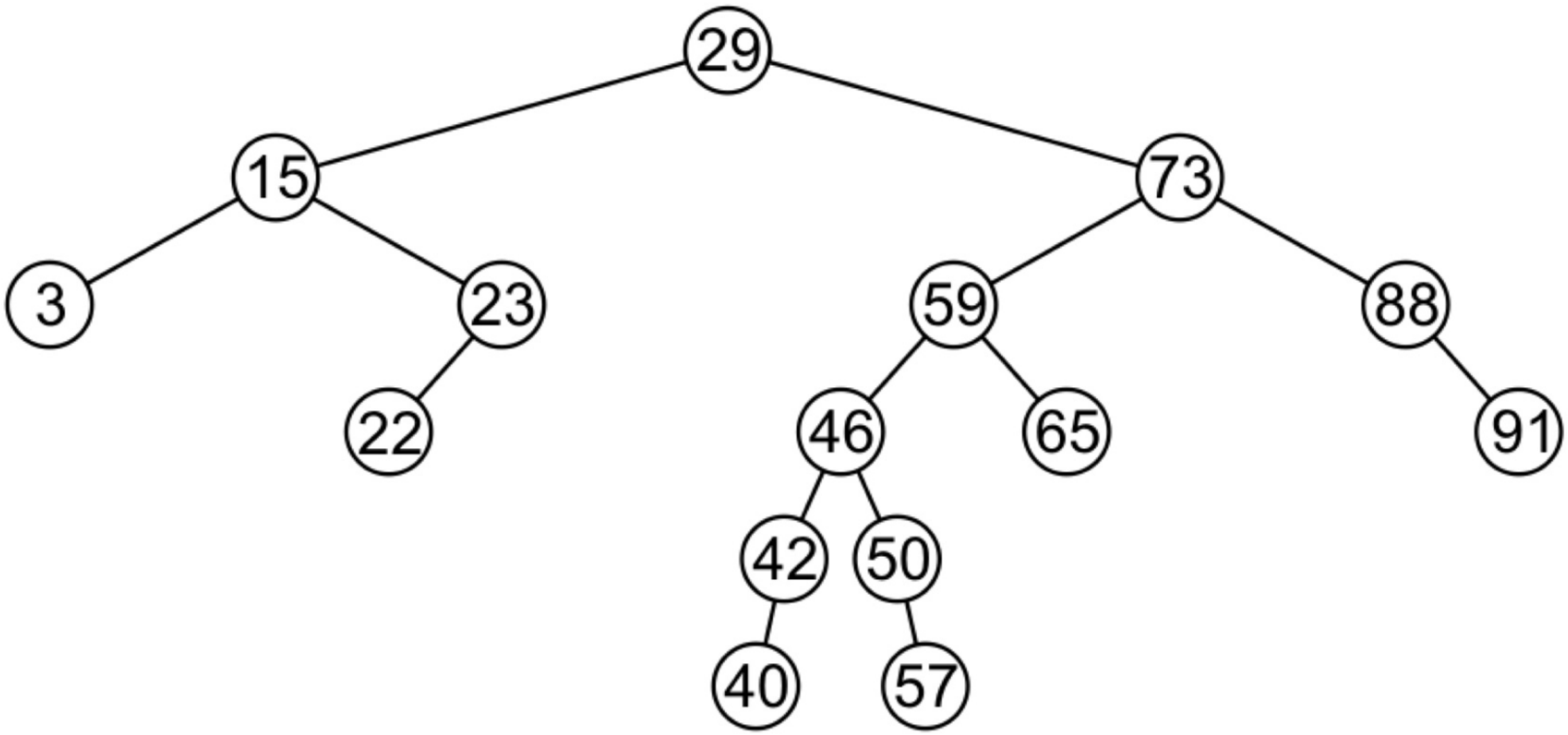
- In addition to **insert**, **remove** and **find**, ordered dictionaries have the following:
- **findMin()**
- **findMax()**
- **Predecessor(k)**
- **Successor(k)**
- Not possible using hashing
- Queues and arrays are too slow
- BST ideal for ordered dictionaries

Binary Search Trees

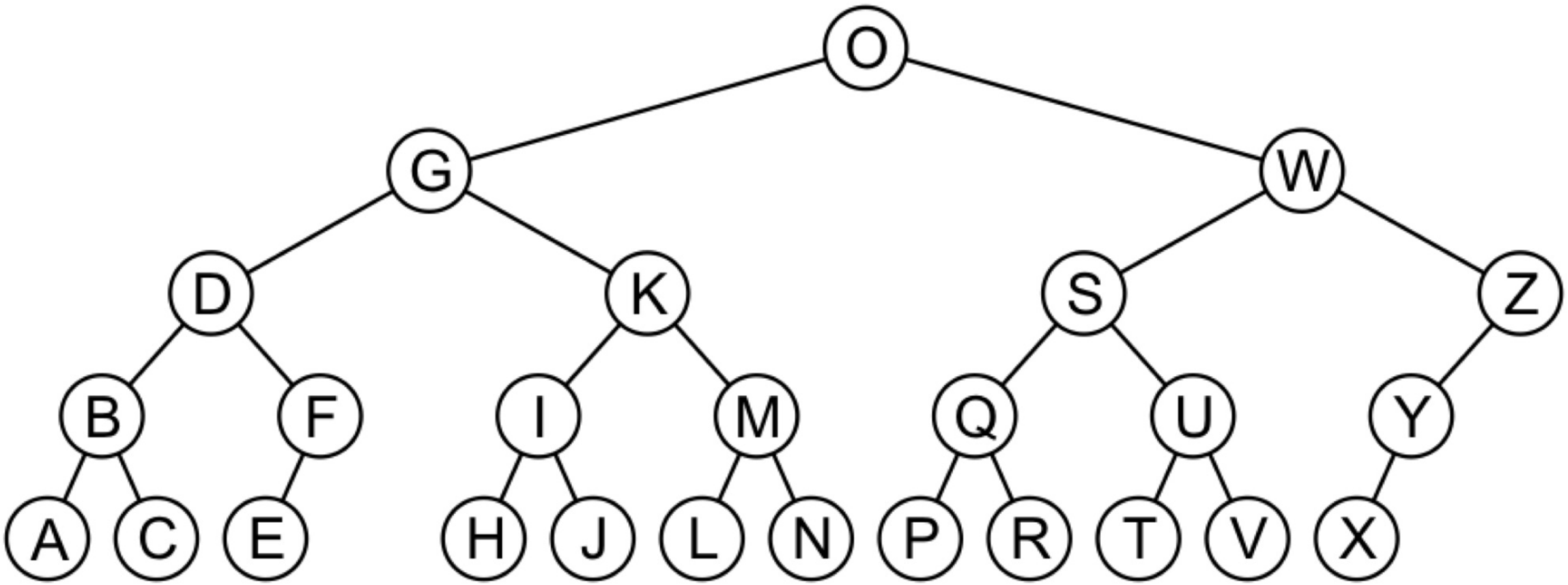
- A binary search tree is a binary tree T indexed by key k such that
  - (Key, value) pairs are stored at every node v
  - Keys stored at nodes in the left subtree of every node *v* are less than or equal to *k*
  - Keys stored at nodes in the right subtree of *v* are greater than or equal to *k*



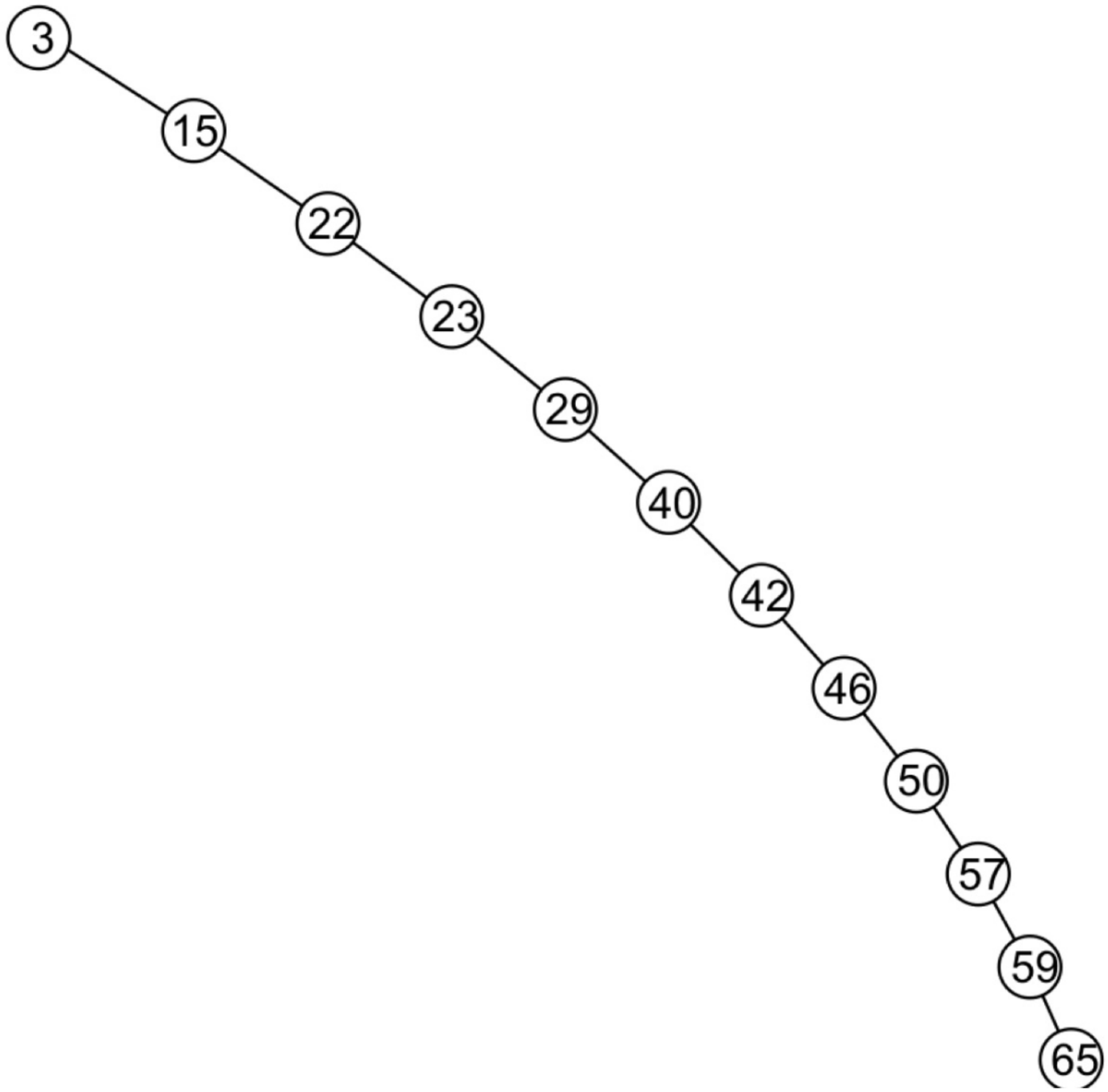
Examples of Binary Search Tree



Examples of Binary Search Tree

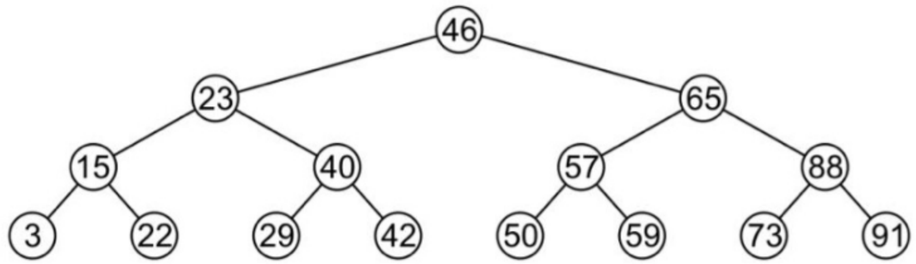


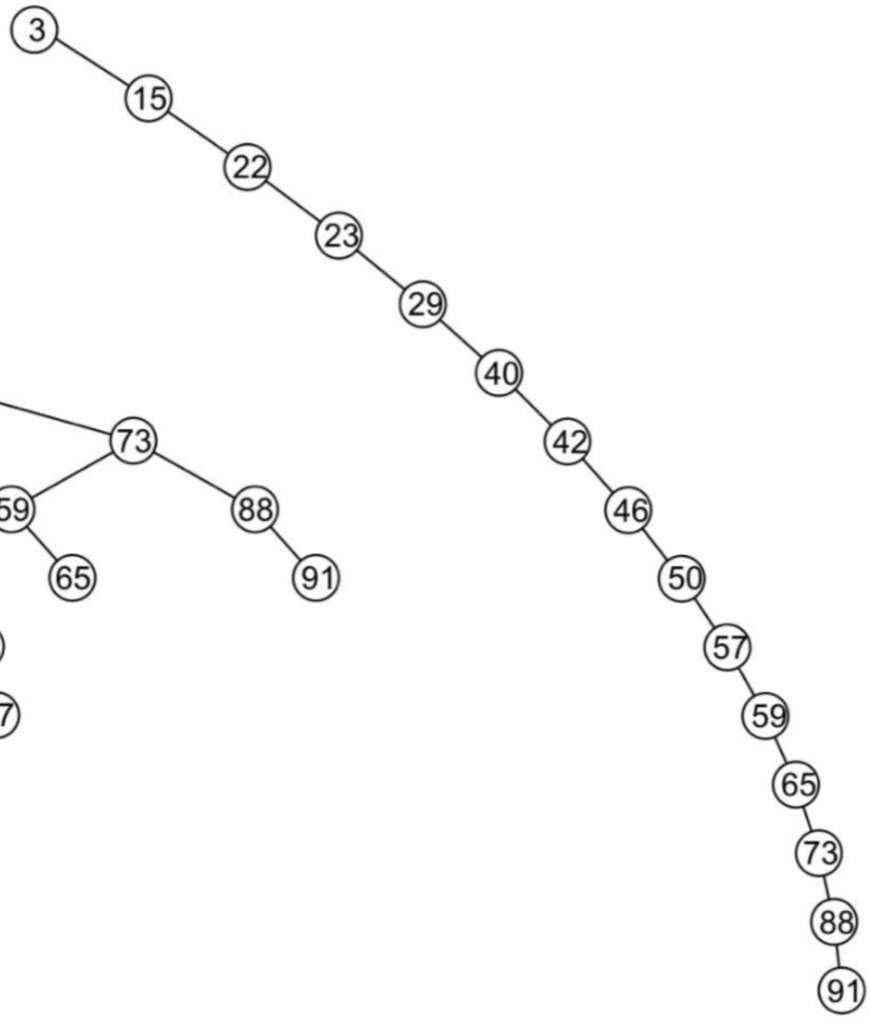
Examples of Binary Search Tree

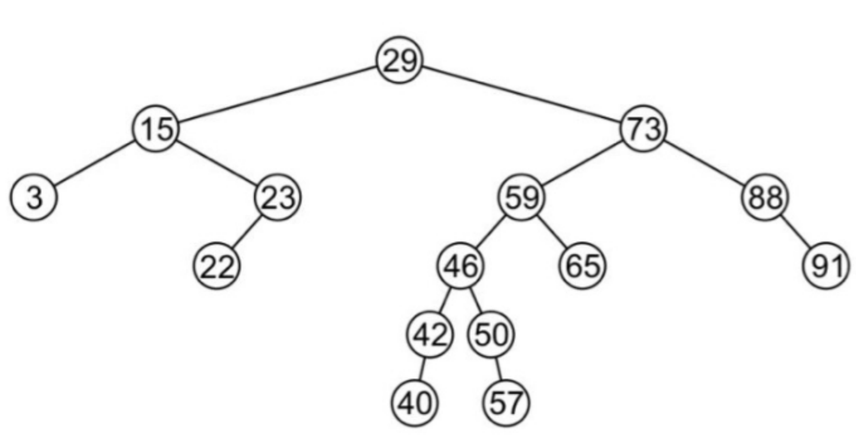


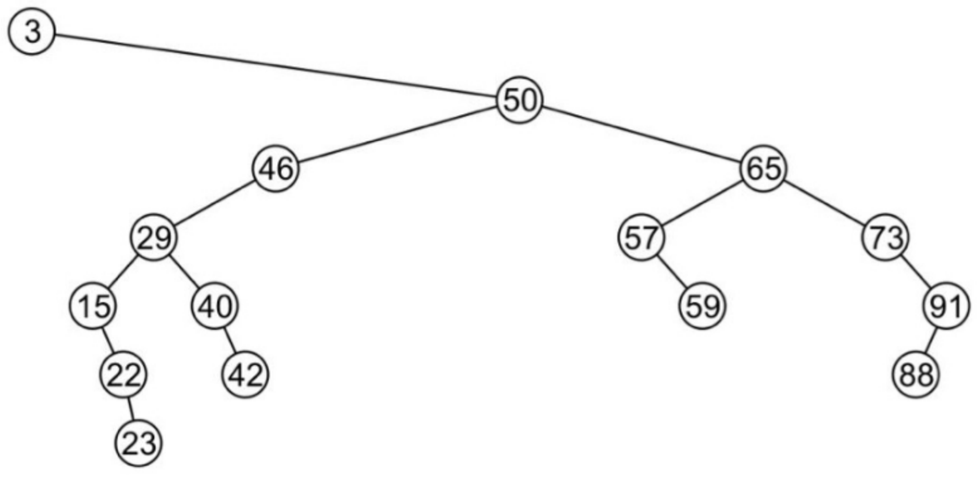
Examples

All these binary search trees store the same data







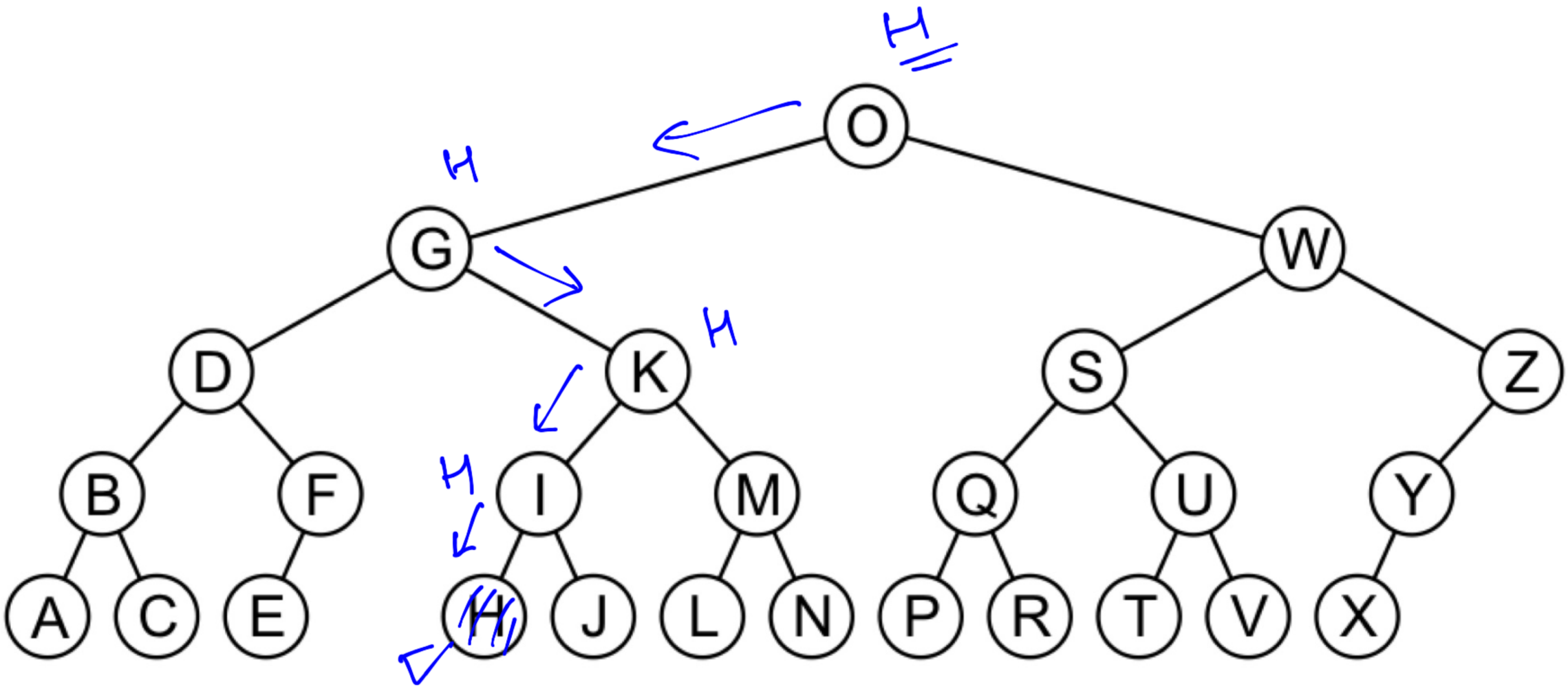


Searching for a key in a BST

- Start with the root
- If (root == null), return “not found”
- Compare the input key k with the root’s key
- If (k == root->key) then key found. Return “found”, root.
- if (k < root->key) then search in the left subtree
- if (k > root->key) then search in the right subtree
- Total time taken: O(h) where h is the height of the tree

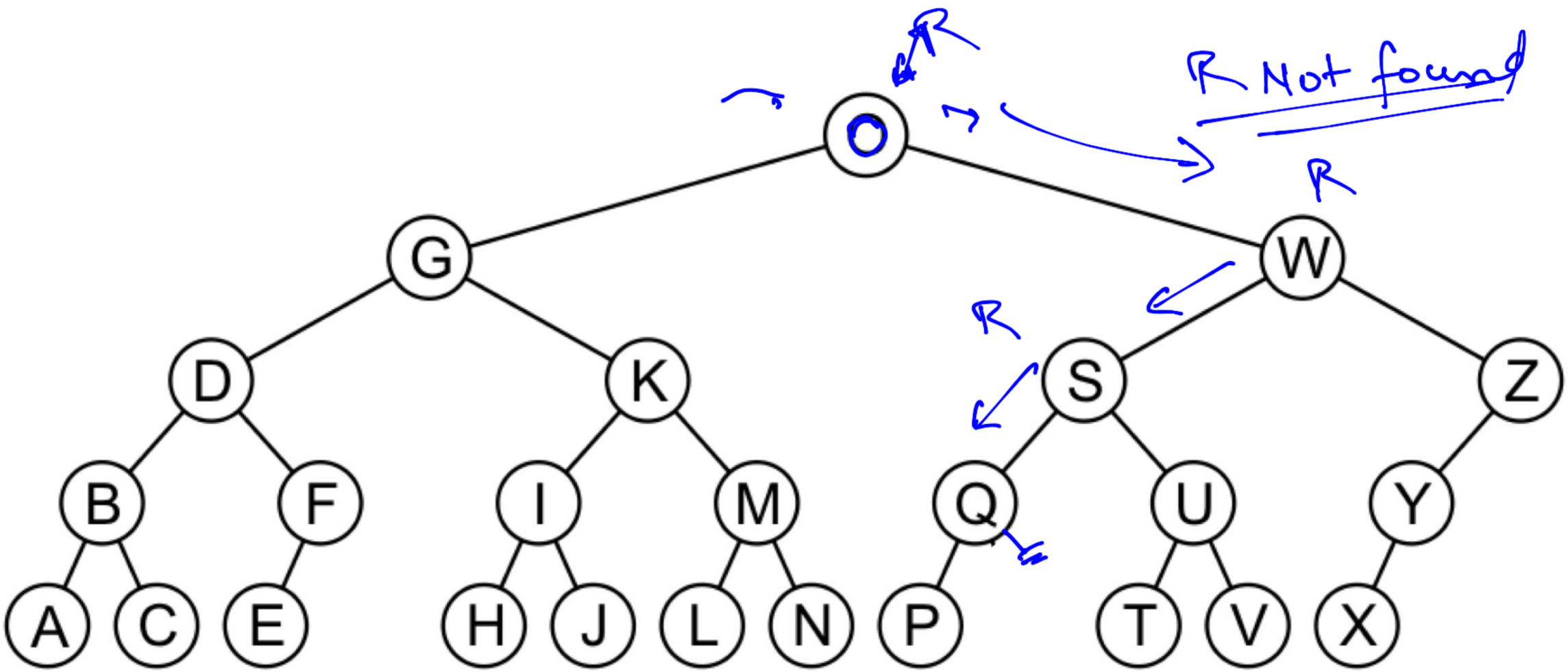
Searching for a key in a BST

- Search for H



Searching for a key in a BST

- Search for R



Binary Search Tree: Find (Recursive)

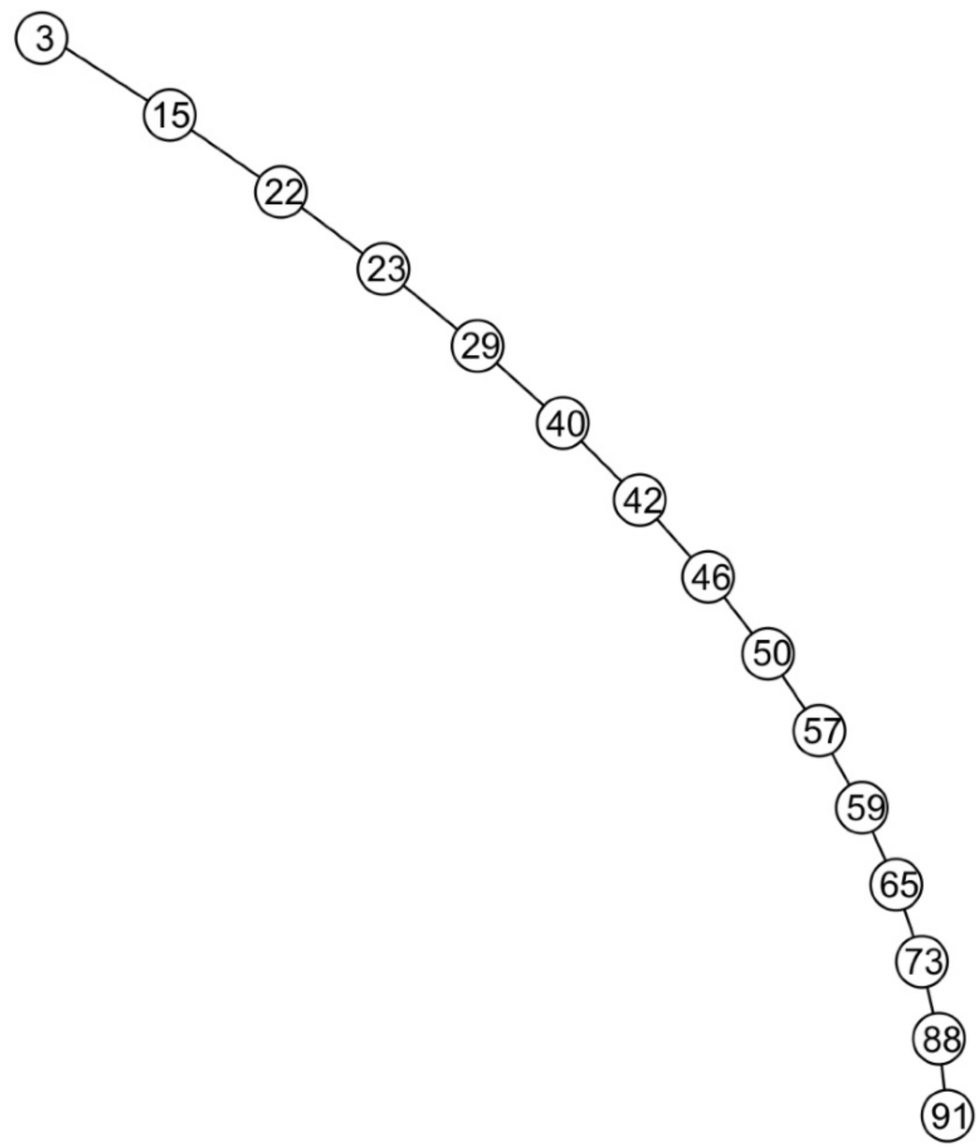
```
BinaryTreeNode *find(int key, BinaryTreeNote *root) {
    // Searches for the key in the BST rooted at root
    // Returns the node pointer if found and null if not
    if (root == 0) return root; // Key not found
    if (key == root->data) return root; // Key found
    if (key < root->data) // Search in the left subtree
        return find(key, root->left);
    if (key > root->data) // Search in the right subtree
        return find(key, root->right);

}
```

Binary Search Tree: Find (Iterative)

```
BinaryTreeNode *find(int key, BinaryTreeNote *root) {
    // Searches for the key in the BST rooted at root
    // Returns the node pointer if found and null if not

    // Assumes short circuit evaluation of boolean expressions
    while ((root != 0) && (key != root->data)) {
        if (key < root->data) // Search in the left subtree
            root = root->left;
        if (key > root->data) // Search in the right subtree
            root = root->right;
    }
    return root;
}
```

Find: Analysis

- Number of steps = O(h) which can be O(n) in the worst case where
  - h = height of the tree
  - n = number of nodes in the tree



Finding the Minimum Key in a BST

```
BinaryTreeNode *findMin(BinaryTreeNode *root) {
    while (root->left != NULL)
        root = root->left;
    return root;
}
```

Finding the Maximum Key in a BST

```
BinaryTreeNode *findMax(BinaryTreeNode *root) {
    while (root->right != NULL)
        root = root->right;
    return root;
}
```

Insertion in a BST

Insert(v, T)

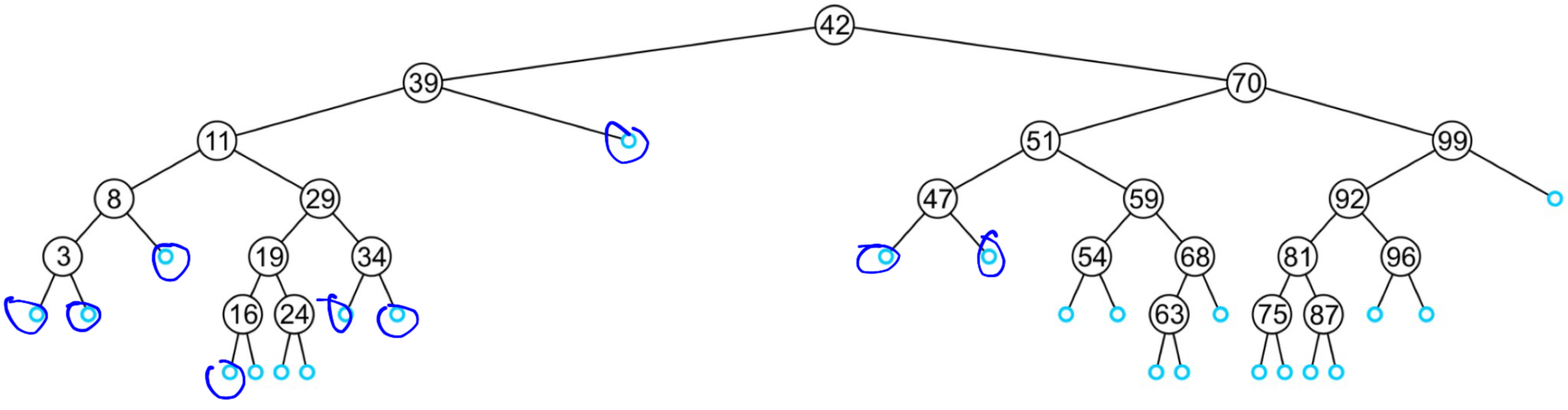
// Inserts a new node v (with null left and right pointers) in the BST T

Find the right place for v in the tree T

Make v the “suitable child” of the “suitable node” in the tree

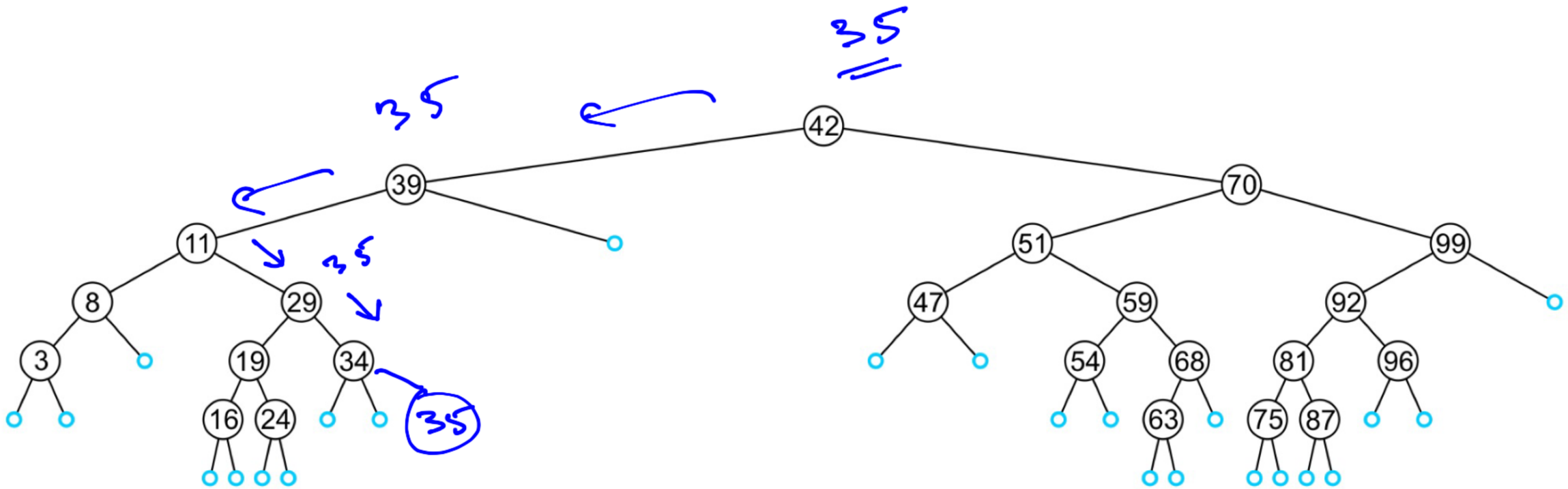
Insertion in the BST

- The new node is inserted at one of the leaves
- Where exactly it needs to be inserted depends on the value of the key of the new node



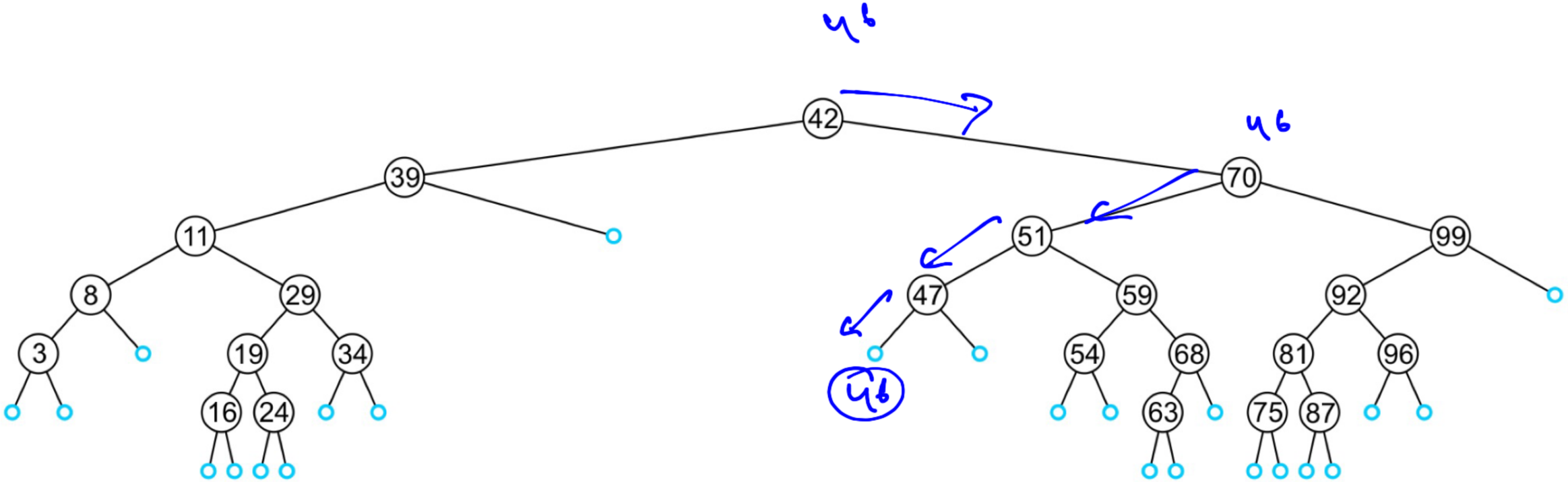
Insertion in the BST

Insert (35)



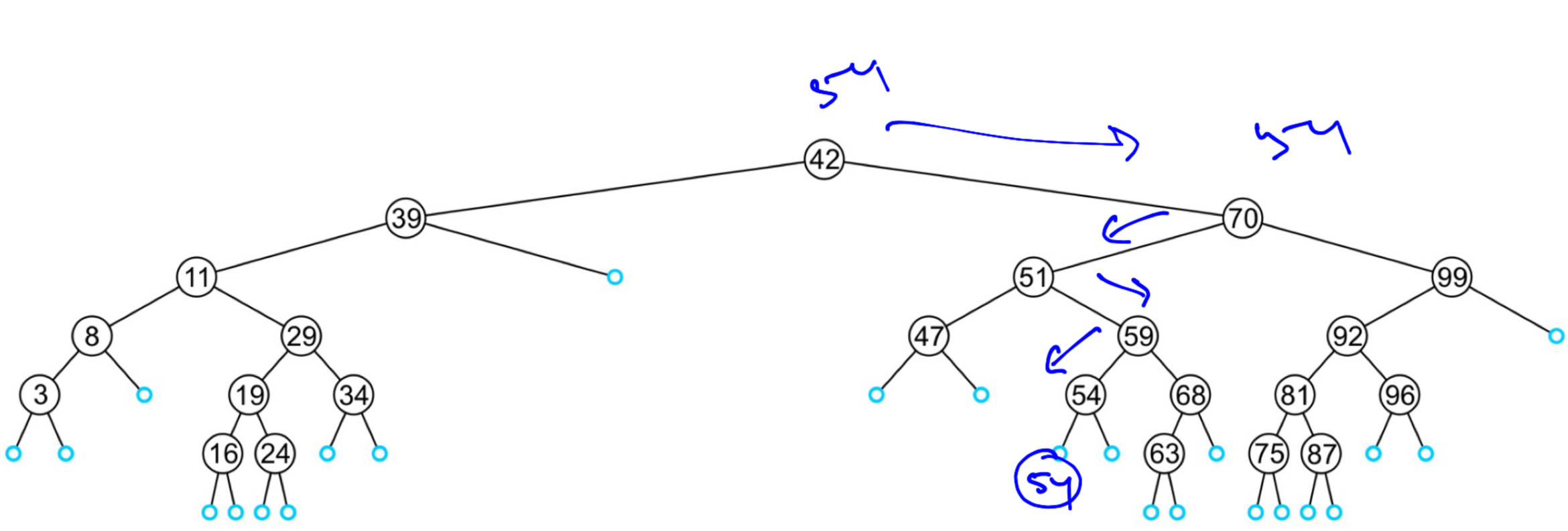
Insertion in the BST

Insert (46)



Insertion in the BST

Insert (~~77~~) 54



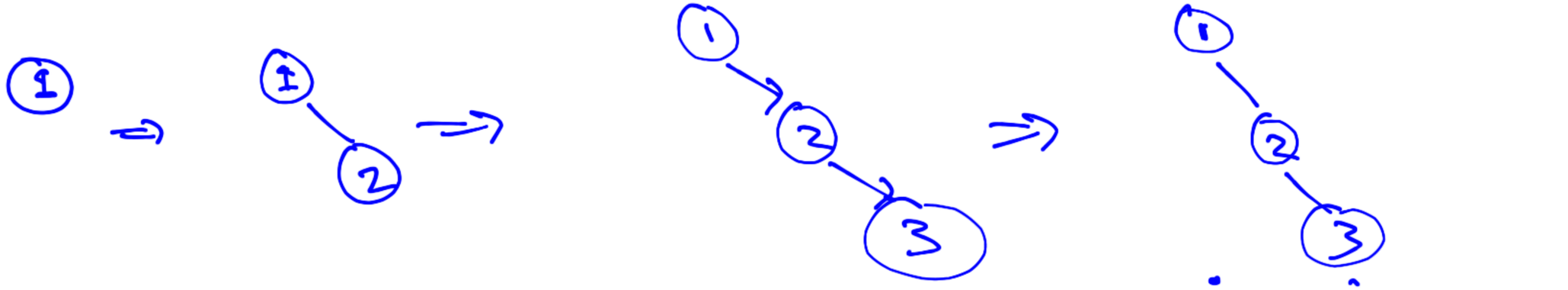
Insert Sample Code

```
void insert(BinaryTreeNode *root, BinaryTreeNode *v) {
    // Inserts node v into the BST T
    // Assumes that v->left and v->right are null
    // Assumes that v->key is not already present in T

    if (v->key < root->key) {
        // Insert on the left side of the root
        if (root->left == 0) {
            // We have found the right place to insert
            root->left = v;
            v->parent = root; // v->left and v->right are assumed to be null
        } else // Insert in the left subtree
            insert(root->left, v);

    } else { // Insert on the right side of the root
        if (root->right == 0) {
            // We have found the right place to insert
            root->right = v;
            v->parent = root;
        } else // Insert in the left subtree
            insert(root->right, v);
    }
}
```

Runtime Analysis: Insert

- What is the worst-case running time of N insert operations on a BST?

Input Insert 1, 2, 3 . . . . N



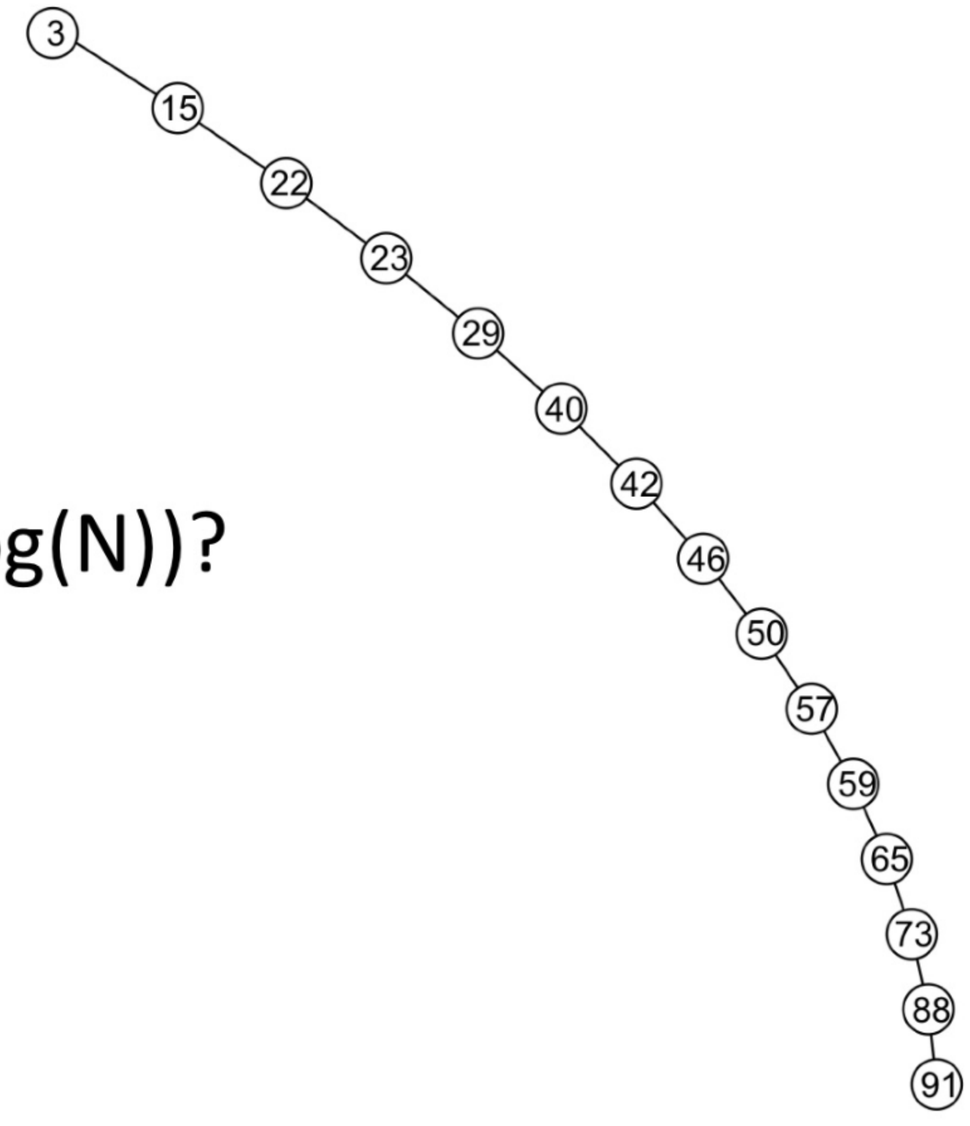
Time to insert i^{th} element : i

Total time: $O\left(\sum_{i=1}^{N} i\right)$

$= O\left(\frac{N(N+1)}{2}\right) = O(N^2)$

Runtime Analysis: Insert

- What is the worst-case running time of N insert operations on a BST?
- Runtime: O(h)
- h = N
- How to make it faster?
- Is it possible to make h = O(log(N))?



Runtime Analysis: Insert

- Worst-case running time of N insert operations on a binary tree?

Runtime Analysis: Insert

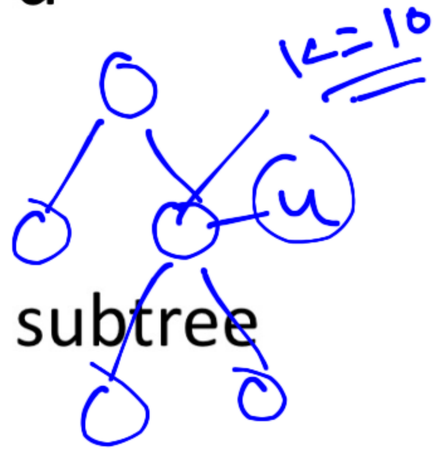
- Worst-case running time of N insert operations on a binary tree?
  - How to create a balanced binary search tree?

Runtime Analysis: Insert

- Worst-case running time of N insert operations on a binary tree?
- O(h) where h is the height of the tree
  - How to create a balanced tree?

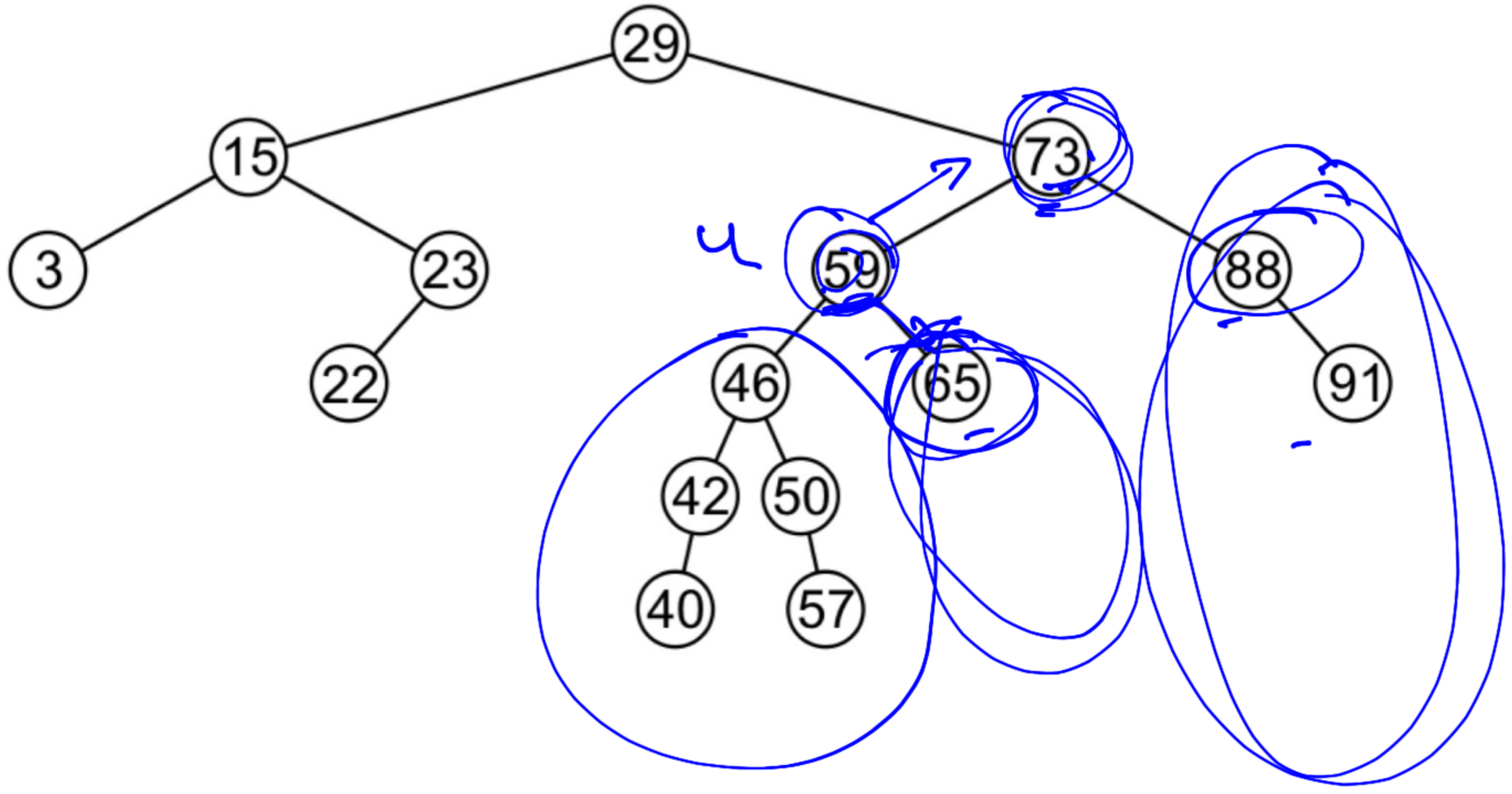
Finding Successor in a BST

- Given a BST T and a node u, successor(u) is the node v in T with smallest key greater than u->key
- Two cases depending on the right subtree of u
- Case 1:
  - u has a right subtree
  - Successor(u) is the node with smallest key in the subtree u->right
  - Can be found by findmin(u->right)



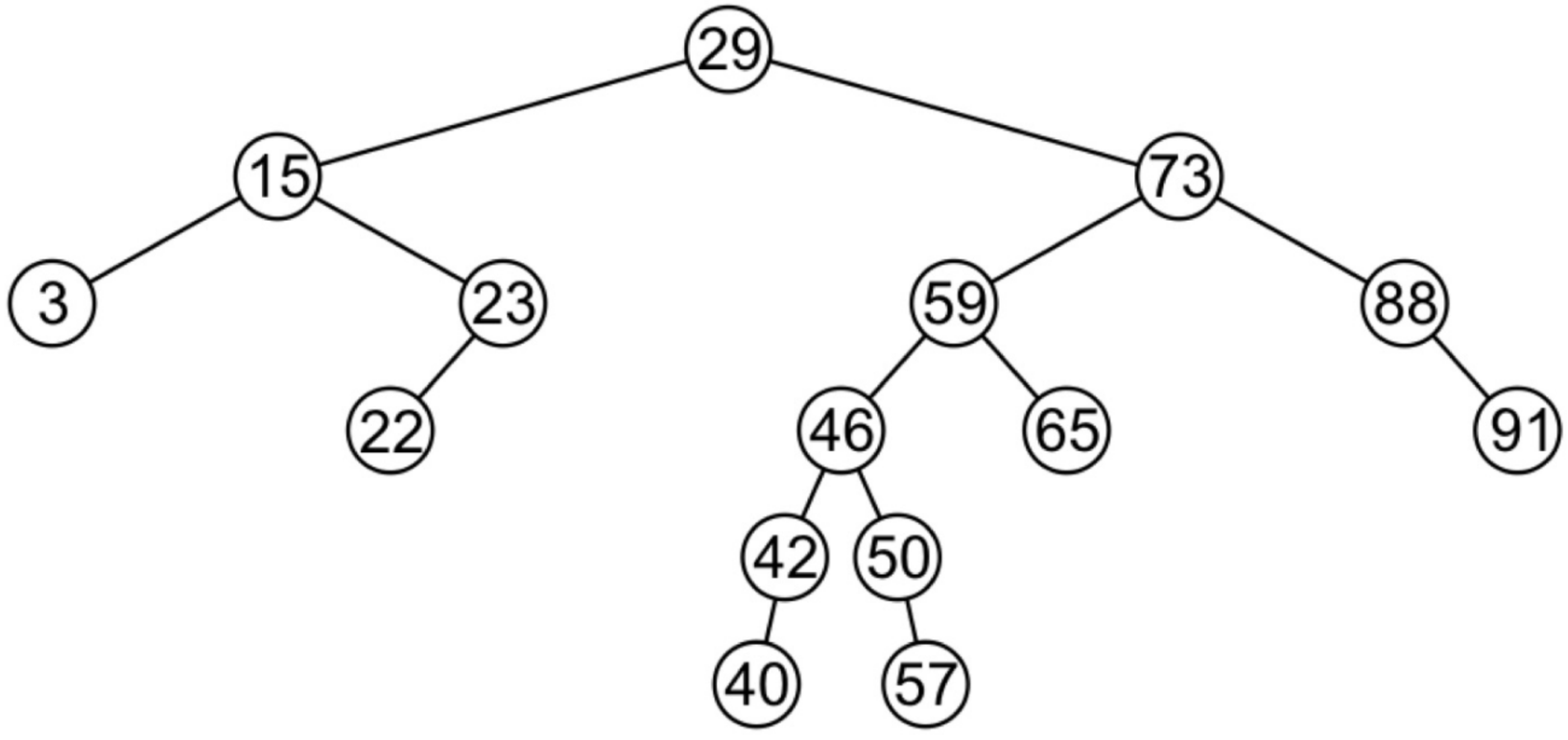
Finding Successor in a BST

- Case 1:
  - u has a right subtree
  - Successor(u) is node with smallest key in the subtree u->right
  - Why?
  - Consider root



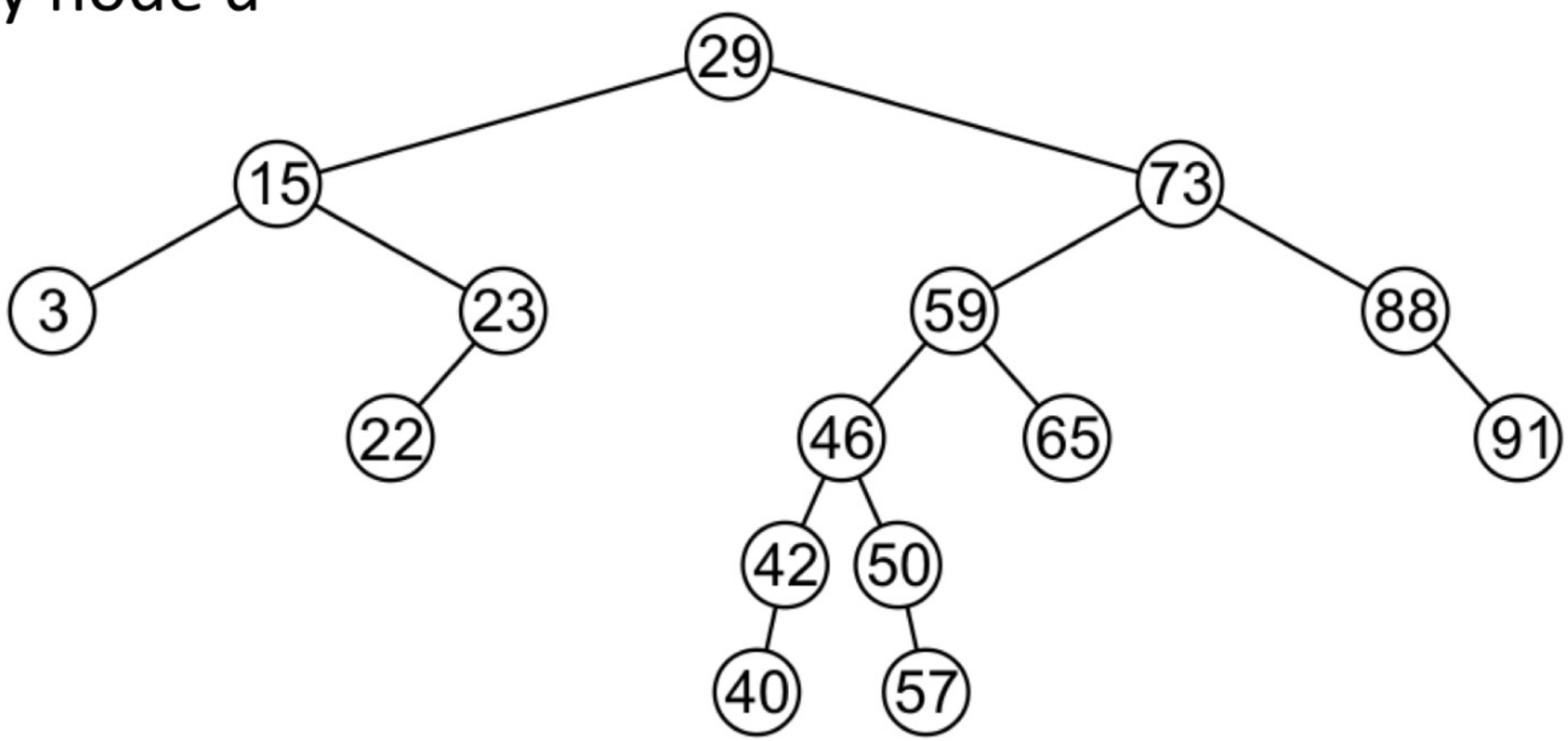
Finding Successor in a BST

- Case 1:
  - u has a right subtree
  - Successor(u) is node with smallest key in the subtree u->right
  - Why?
  - Consider node 59



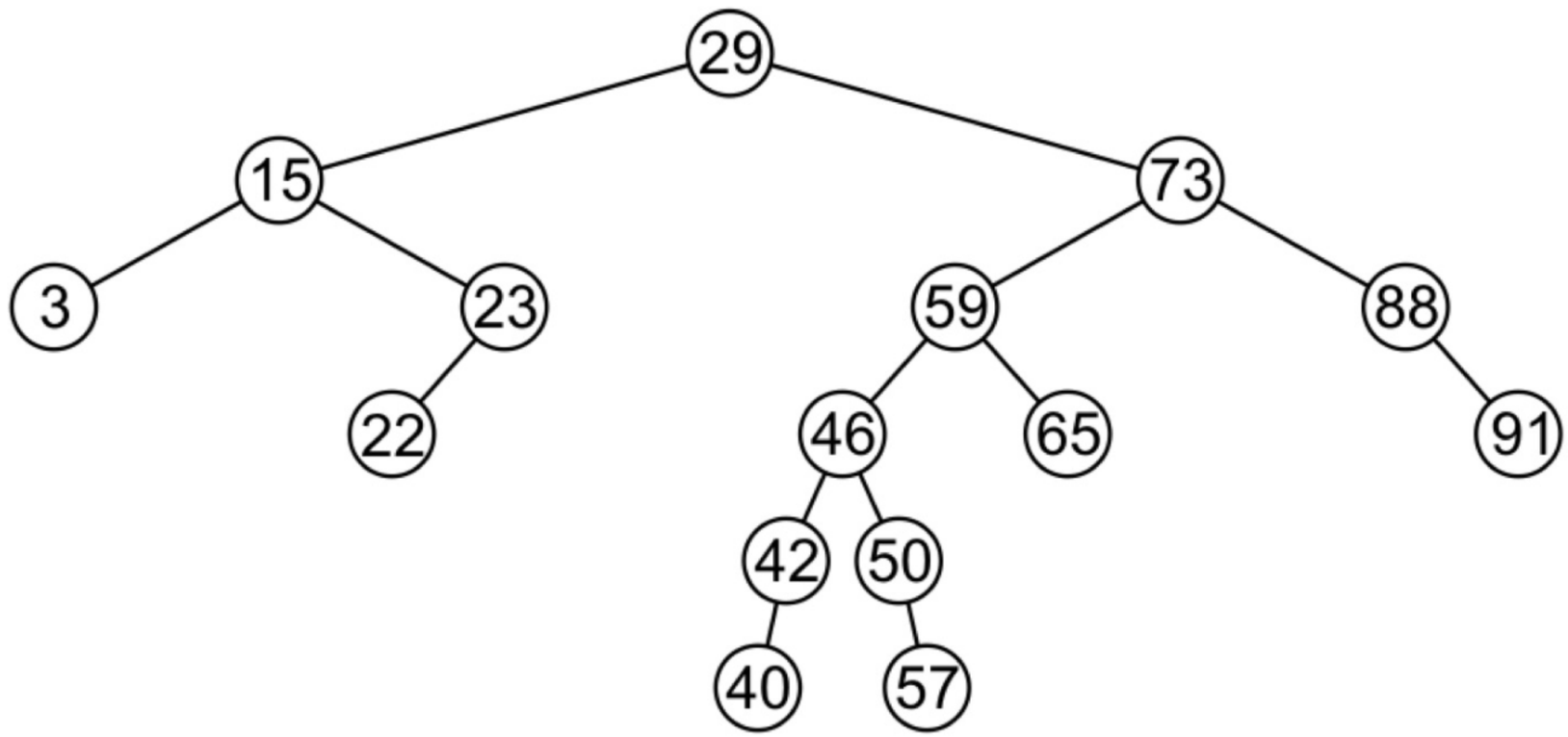
Finding Successor in a BST

- Case 1:
  - u has a right subtree
  - Successor(u) is node with smallest key in the subtree u->right
  - Why?
  - Consider any node u



Successor(u) – Case 1

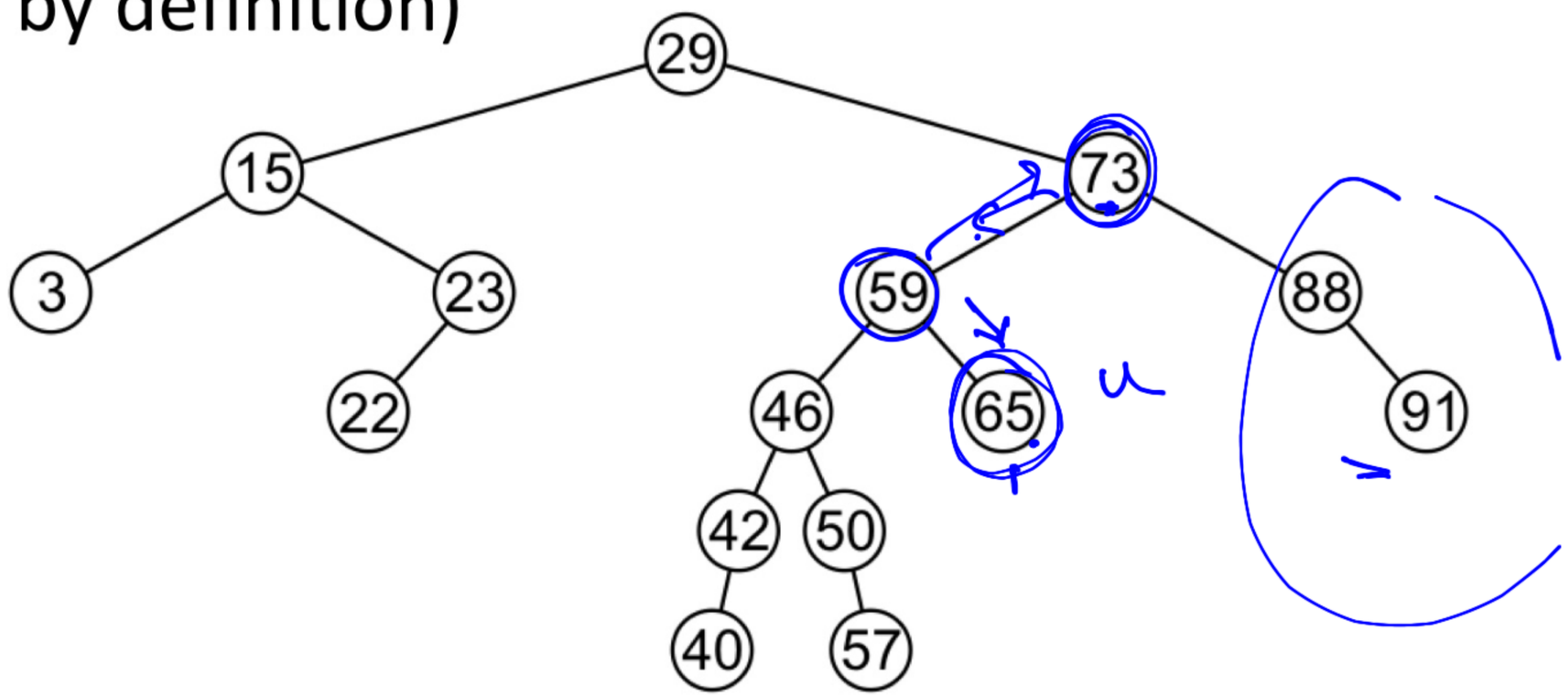
- Tree may be partitioned as
- subtree(u), subtree(sibling(u))
- subtree(parent(u)), subtree(sibling(parent(u)))



Successor(u) – Case 2

Successor (65)

- Case 2
  - u->right = 0
- successor(u) is the lowest ancestor of u whose left child is also an ancestor of u (note u is also in ancestor(u) by definition)
- Why?



Finding successor: Sample Code

```
BinaryTreeNode *successor(BinaryTreeNode *u) {
    if (u->right != 0)
        return findMin(u->right);
    // Assumes short circuit left to right evaluation
    else while ((u->parent != 0) && (u == u->parent->right)) {
        u = u->parent;
    }
    return u->parent;
}
```

Analysis: successor

- Number of steps taken to find the successor?

Analysis: successor

- Number of steps taken to find the successor?
- Time complexity: O(h) (h is the height of the tree)

Removing a Node from a BST

- Remove(T, k): Remove a node in T that has key k
- Assume that no two nodes have the same key
- u = find(T, k)
- There are three cases:
  - u has no children
  - u has one child
  - u has two children

BST Removal: Case 1

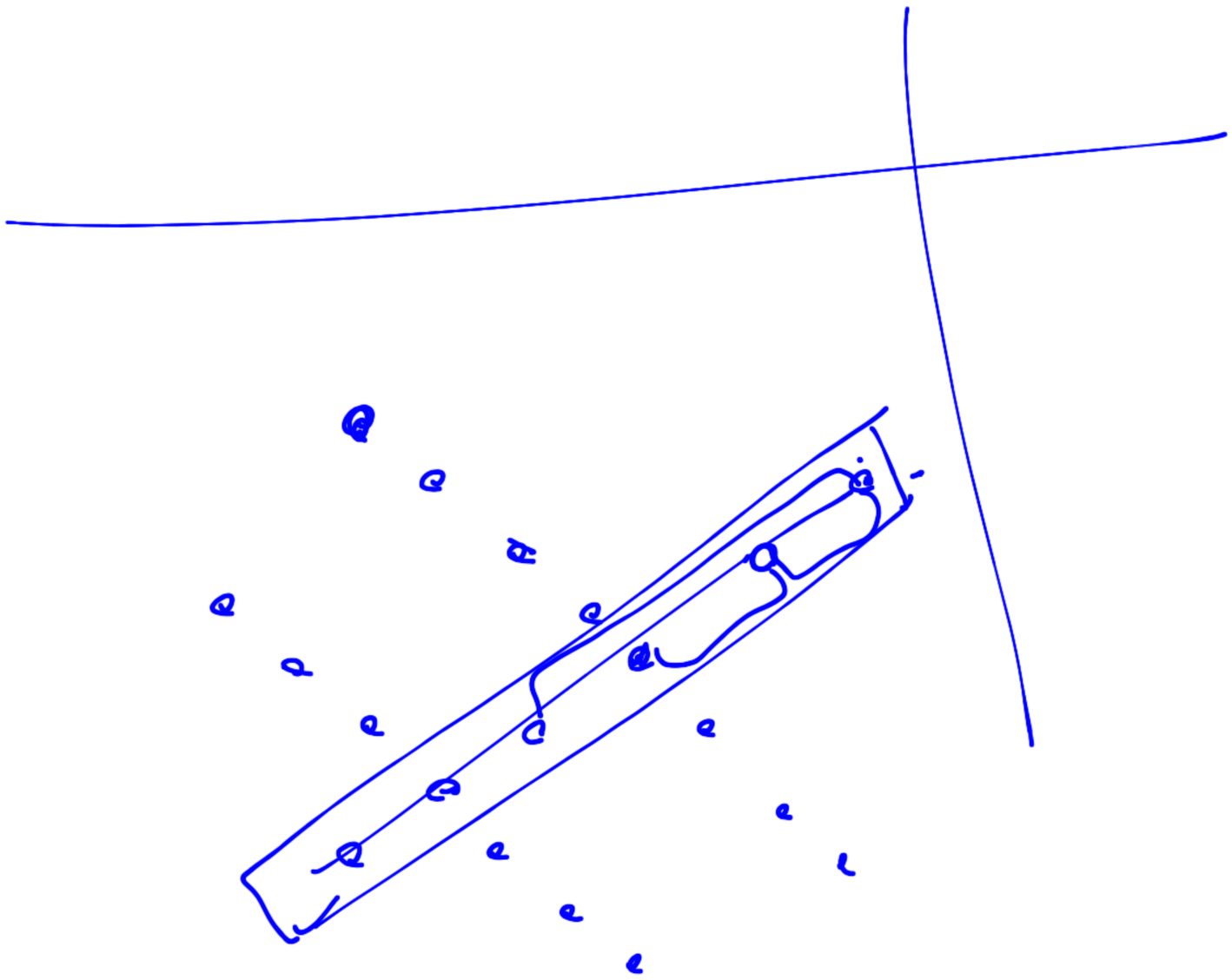
- If u has no children, simply remove it and free the memory

BST Removal: Case 2

- If u has exactly one child
  - Make the parent(u) point to the child
  - Two sub-cases here
    - u == u->parent->left
    - u == u->parent->right

BST Removal: Case 3

- Case 3: u has two children
- Find v = successor(u) [can also use predecessor(u)]
- v has only one child (prove it)
- Replace u with v and remove v



Thank You