DATA STRUCTURE ALGORITHMS AND APPLICATIONS (CT- 159)

Lecture – 9
Binary Search Tree

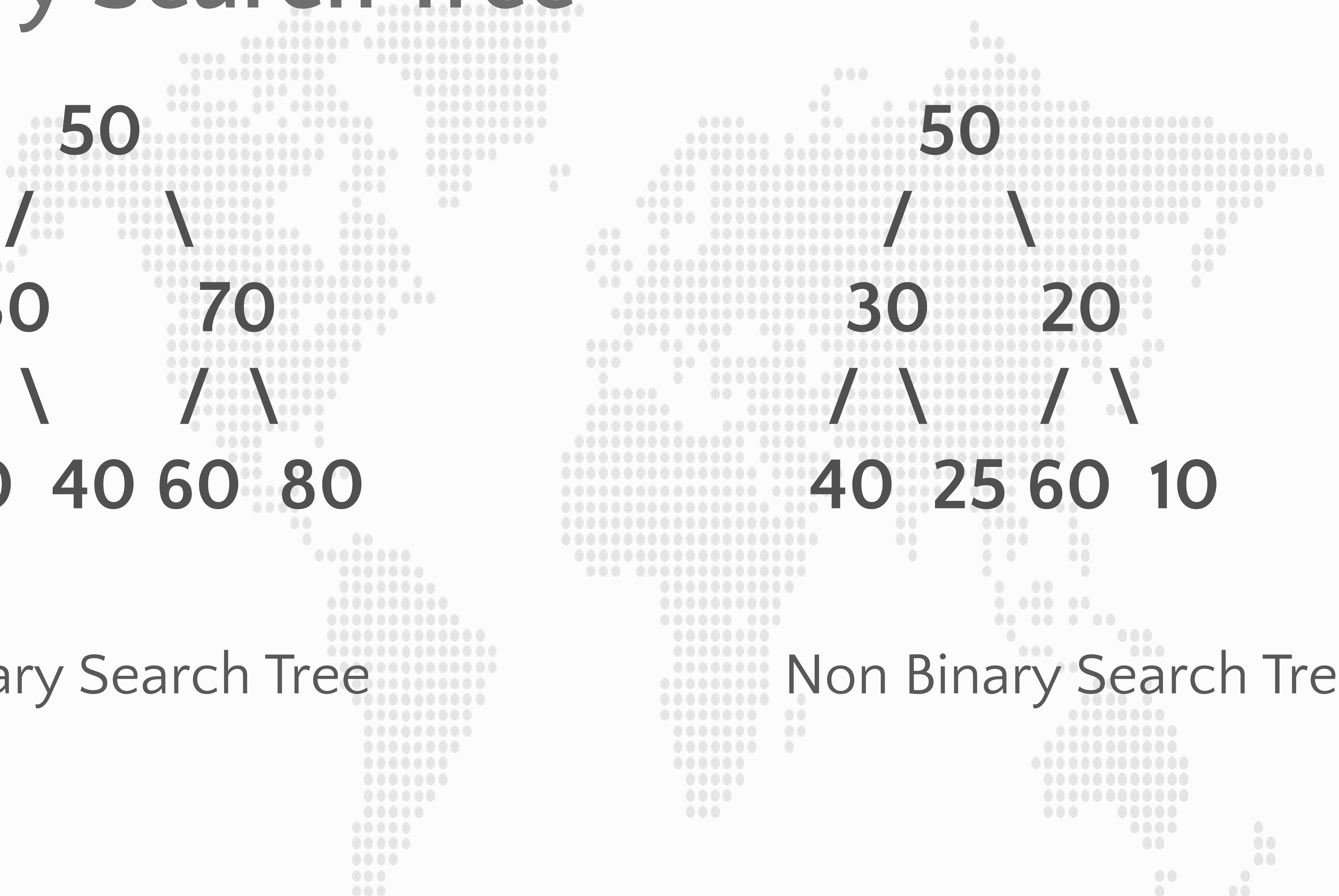
Instructor: Engr. Nasr Kamal

Department of Computer Science and Information Technology

Binary Search Tree (BST)

- Stores keys in the nodes in a way so that searching, insertion and deletion can be done efficiently.
- Binary search tree property
  - For every node X, all the keys in its left subtree are smaller than the key value in X, and all the keys in its right subtree are larger than the key value in X

Binary Search Tree



```
     50
    /   \
  30      70
 / \     / \
20  40 60  80
```

Binary Search Tree

```
     50
    /   \
  30     20
 / \    / \
40  25 60  10
```

Non Binary Search Tree

Searching in BST

The left subtree contains nodes with values smaller than the current node, and the right subtree contains nodes with values larger than the current node.

This makes searching efficient, with an average time complexity of O(logn), where n is the number of nodes in the tree.

Steps for Searching in BST

Start at the root: Compare the value you are searching for with the root node's value.
Compare the target value with the current node:

- If the target value is **equal** to the current node, you have found the value, and the search ends.
- If the target value is **less** than the current node, search the **left subtree** because the value must be in the left if it exists.
- If the target value is **greater** than the current node, search the **right subtree** because the value must be in the right if it exists.

Repeat the process recursively: Move down the tree, continuing the comparisons, until you either find the target value or reach a leaf node (a node with no children), in which case the value is not present in the tree.
If you reach a null node (i.e., you've gone past a leaf), the value is not found in the tree.

Example

Searching for 60:

- Start at the root (50). Compare 60 with 50.
  - 60 > 50, so move to the right subtree.
- Now at 70. Compare 60 with 70.
  - 60 < 70, so move to the left subtree.
- Now at 60. Compare 60 with 60.
  - The values are equal, so the value 60 is found.

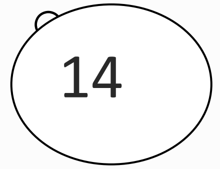
```
      50
     /  \
   30    70
  / \    / \
20 40  60  80
```

TIME COMPLEXITY:
Average case: O(logn), because each step cuts the remaining tree in half, similar to binary search.
Worst case: O(n), in the case of an unbalanced tree where each node has only one child (essentially forming a linked list).

Creating a BST

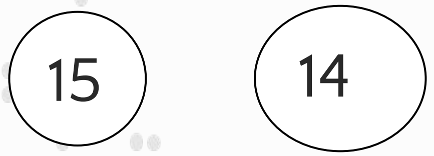
- The first number in the list is placed in a node, designated as the *root* of the binary tree.
- Initially, both left and right sub-trees of the *root* are empty. We take the next number and compare it with the number placed in the *root*. If it is the same, this means the presence of a duplicate (do nothing).
- Otherwise, we create a new tree node and put the new number in it. The new node is turned into the left child of the *root* node if the number is less than the one in the *root*. The new node is turned into the right child if the number is greater than the one in the *root*.

Creating BST



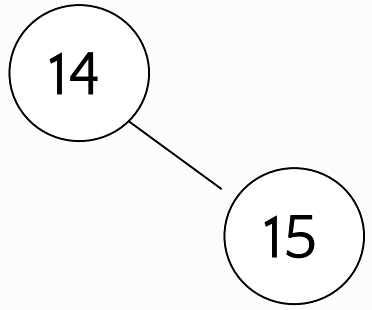
14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

First number in the list became the *root*



14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

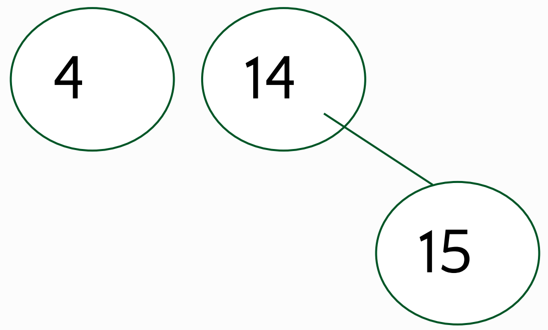
A new node is created to insert it in the binary tree



14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

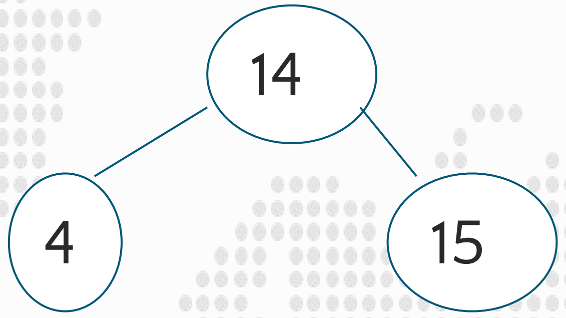
The second node is added into the tree

Creating a BST



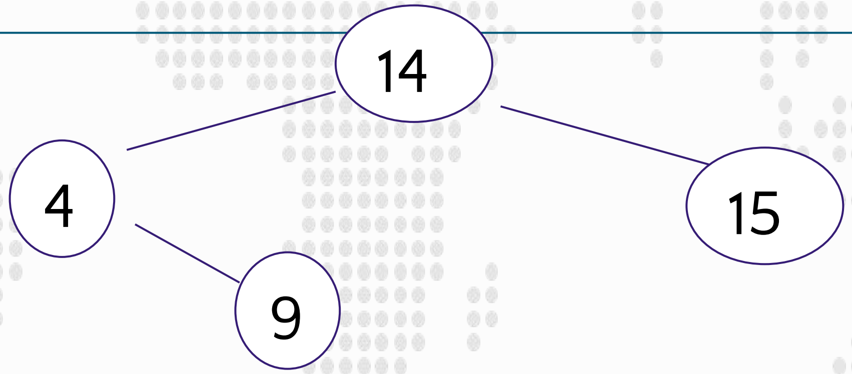
14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

A new node is created and number 4 put into it



14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

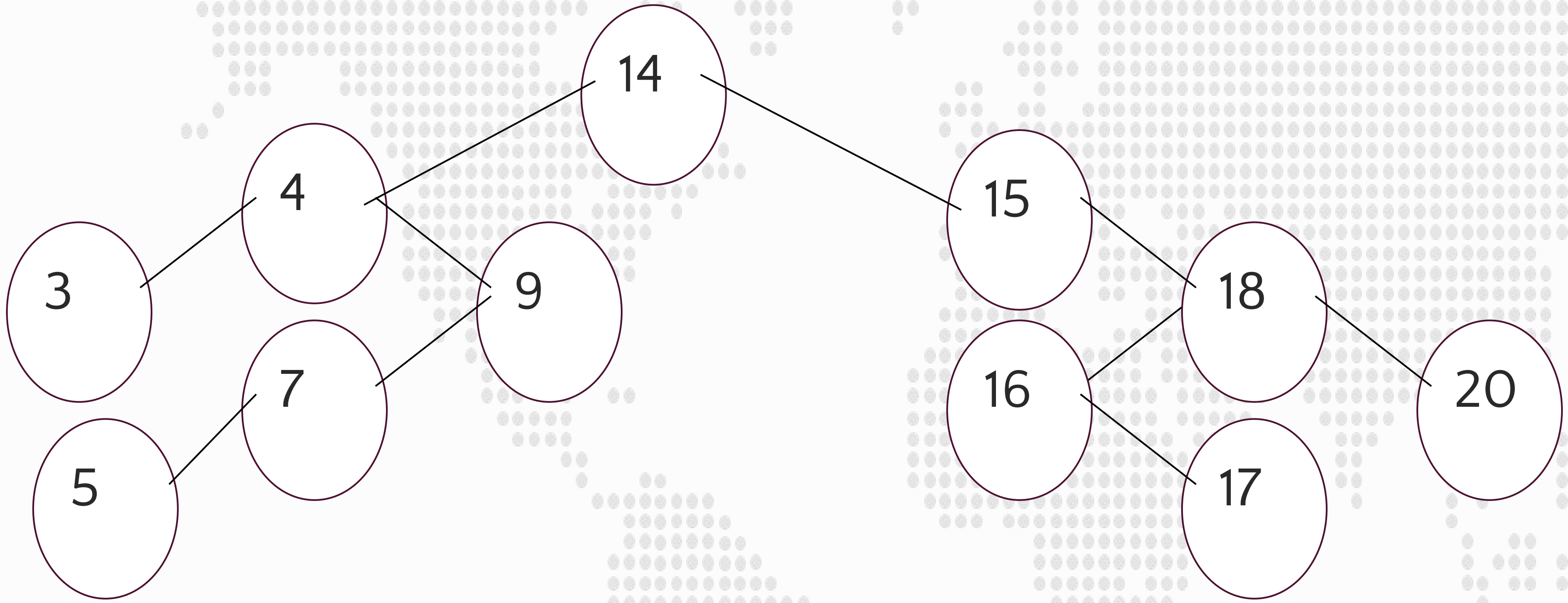
The node is added as the left child of the *root* node



14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

A new node is added in the tree

Creating BST



14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

Binary tree of whole list of numbers

Insert a Node in BST

- If root is nullptr: Create a new node with value
- If value < root.data: Set root.left to insertNode(root.left, value) (recursively insert in the left subtree)
- Else If value > root.data: Set root.right to insertNode(root.right, value) (recursively insert in the right subtree)

Insert a Node in BST

```
class TreeNode {
public: int val;
TreeNode* left;
TreeNode* right;
TreeNode(int value) : val(value),
left(NULL), right(NULL) {
} };
class BinarySearchTree {
public: TreeNode* root;
BinarySearchTree() : root(NULL) {}
void insert(int key) {
root = insertRec(root, key);
}
```

```
private: // Recursive function to insert a
node
TreeNode* insertRec(TreeNode* node, int
key) {
// Base case: if the tree is empty, create a
new node
if (node == NULL) return new
TreeNode(key);
// Otherwise, recur down the tree
 if (key < node->val)
node->left = insertRec(node->left, key);
else if (key > node->val)
node->right = insertRec(node->right, key);
return node; }
```

Delete a Node from BST

- Find the Node: Start from the root and traverse the tree based on the value you want to delete.
- Move left if the value is smaller.
- Move right if the value is larger.

Delete the Node:
– Case 1: Node has no children: Simply remove the node.
– Case 2: Node has one child: Replace the node with its only child.
– Case 3: Node has two children: Find the inorder successor (smallest value in the right subtree), copy its value to the node to be deleted, and recursively delete the inorder successor.

Delete a Node from BST

```
 void deleteNode(int key) {
root = deleteRec(root, key);
}
// Recursive function to delete a node
TreeNode* deleteRec(TreeNode* node, int key) {
 // Base case: if the tree is empty
 if (node == NULL) return node;
// Recur down the tree
if (key < node->val)
 node->left = deleteRec(node->left, key);
 else if (key > node->val)
node->right = deleteRec(node->right, key);
```

```
else {
// Node with only one child or no
child
if (node->left == NULL) {
TreeNode* temp = node->right;
 delete node;
return temp;
}
else if (node->right == NULL) {
TreeNode* temp = node->left;
delete node;
return temp; }
```

Finding a Node Min/Max Values in BST

Three of the easiest things to do with BSTs are find a particular value, find the minimum value, and find the maximum value.

```
 // Find the node with the minimum value (used for deleting nodes)
TreeNode* minValueNode(TreeNode* node) { TreeNode* current = node;
while (current && current->left != NULL)
current = current->left;
 return current; }
```

```
 // Find the node with the maximum value (used for deleting nodes)
TreeNode* maxValueNode(TreeNode* node) { TreeNode* current = node;
while (current && current->right != NULL)
current = current->right;
 return current; }
```

InOrder Traversal in BST

```
void inorder() { inorderRec(root); cout << endl; }
```

```
 // Inorder traversal function
void inorderRec(TreeNode* root) {
 if (root != NULL) {
inorderRec(root->left);
cout << root->val << " ";
 inorderRec(root->right);
} }
```

Issues with BST

- **Unbalanced Tree:** In the worst case (e.g., inserting sorted data), the tree can become unbalanced, degrading to a linked list, which results in O(n) time complexity for operations.
- **Rebalancing Required:** To maintain efficiency, balancing techniques such as AVL or Red-Black Trees are often used, but these come with added complexity.
- **Memory Usage:** BSTs use more memory than arrays or linked lists due to the storage overhead for pointers (two for each node).

Computational Complexity Comparison of Data Structures

Operations	Arrays	Linked List	Binary Tree	Binary Search Tree
Access (Get by index)	O(1)	O(n)	O(n)	O(log n) (balanced) / O(n) (unbalanced)
Search	O(n)	O(n)	O(n)	O(log n) (balanced) / O(n) (unbalanced)
Insertion	O(n) (at worst)	O(1) (head) / O(n) (tail)	O(log n) (balanced)	O(log n) (balanced) / O(n) (unbalanced)
Deletion	O(n)	O(1) (head) / O(n) (tail)	O(log n) (balanced)	O(log n) (balanced) / O(n) (unbalanced)