CBS Department

Binary Search Tree (BST)

Soma Soleiman Zadeh

Data Structures & Algorithms (CBS 216)

Spring 2025 - 2026

Week 14

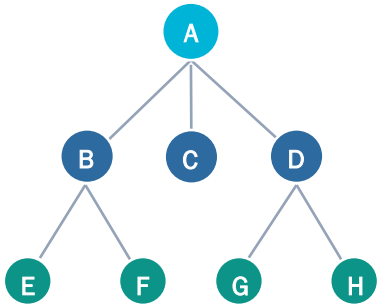
May 06-07, 2026

Outline

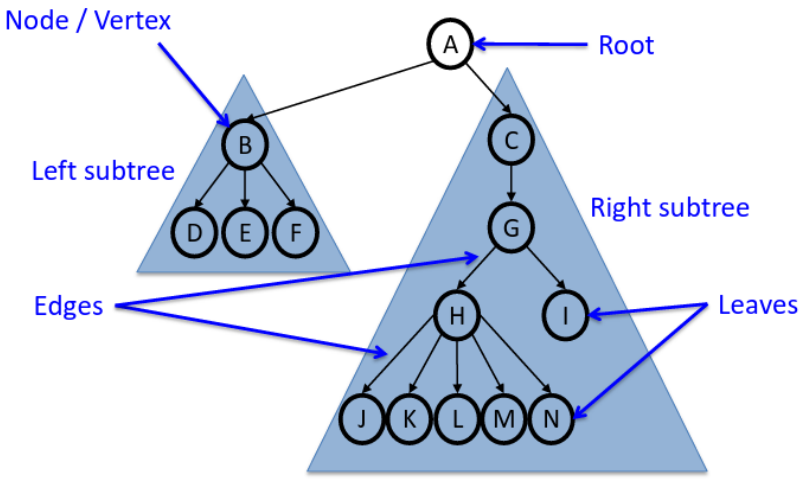
- Basics of **Binary Trees**
- **Binary Search Tree (BST)**
- Operations on **BST**
- Time Complexity
- Applications of **BST**

What is a Tree?

- A **Tree** is a **non-linear hierarchical data structure**.
- A Tree is made of **nodes** connected by **edges,**
  - **NO cycles** allowed.
- It has one special node called the **Root** at the top.
- Every node except the root has exactly one parent.
- Nodes with no children are called **Leaf nodes**.



Tree Terminology

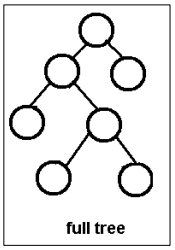


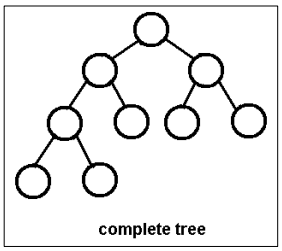
What is a Binary Tree

- A **Binary Tree** is a tree where <u>each node has at most 2 children</u>.
  - ✓ **Left Child** → The sub-tree on the left side of a node.
  - ✓ **Right Child** → The sub-tree on the right side of a node.
  - ✓ **Max Degree** → Each node has a degree ≤ 2 (0, 1, or 2 children).
  - ✓ **Full Binary Tree** → Every node has 0 or 2 children (no singles).
  - ✓ **Complete Binary Tree** → All levels filled except possibly the last, filled left-to-right.

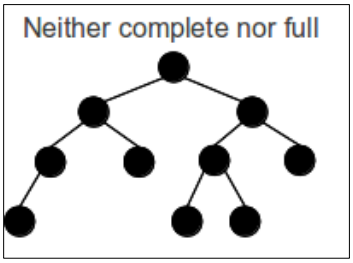
Full Binary Tree vs. Complete Binary Tree

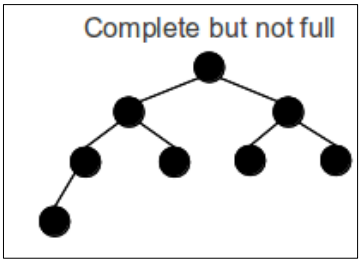
- A **Full Binary Tree** is a type of binary tree where every node has either **zero** or **two** children. No node in a Full Binary Tree can have exactly one child.
- A **Complete Binary Tree** is a type of binary tree where every level is entirely filled, except possibly the deepest one, which must be filled from left to right.

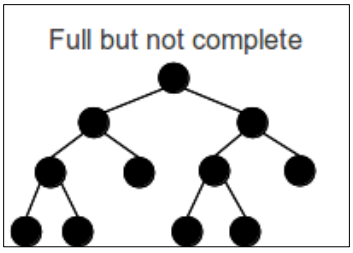


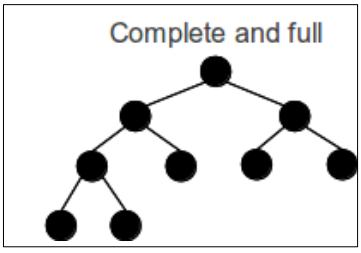


Full Binary Tree?
Complete Binary Tree?









What is a Binary Search Tree (BST)?

- **Binary search trees (BST)** are binary trees where values are placed in a way that **supports efficient searching**.
- Main rule in a **BST:**

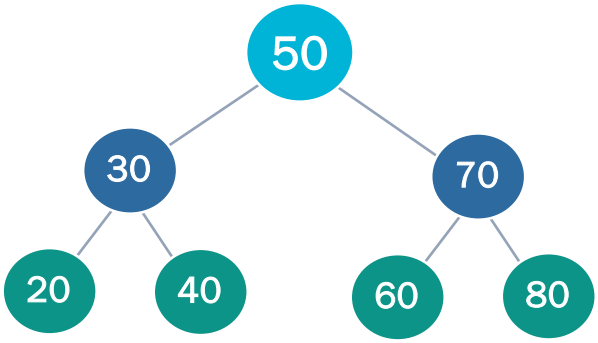
> all values in the **left subtree** < the value in **current node** < all values in the **right subtree**

- This rule must hold for **EVERY** subtree.

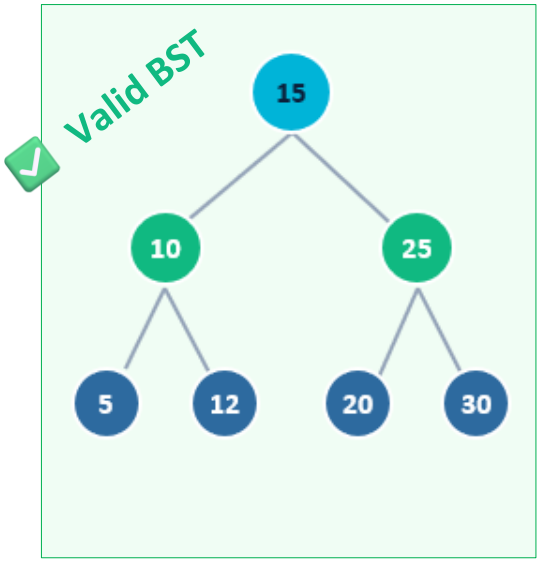
Binary Search Tree (BST)?

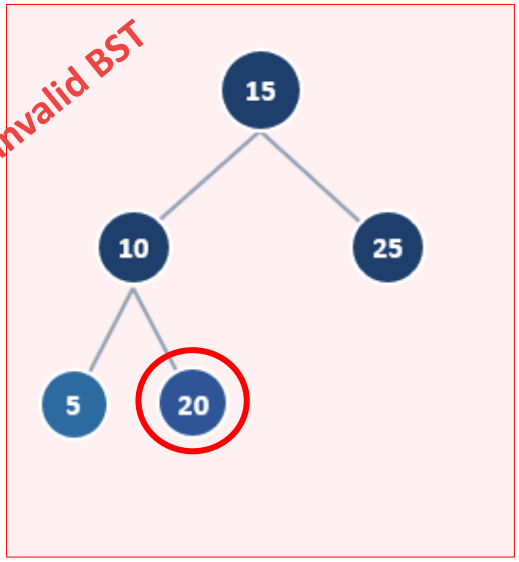
- A **BST** is a binary tree where for every node **N**:
  - All values in the **LEFT** subtree are **LESS THAN N's value**,
  - All values in the **RIGHT** subtree are **GREATER THAN N's value**.

< 30



Valid BST vs. Invalid BST





BST Node Structure

- Each **BST** node holds:
  - a **key/value**,
  - a **pointer to the left child**,
  - a **pointer to the right child**.

Memory layout of one BST node

- **NULL** pointers indicate the absence of a child.
  - The **leaf node** has two **NULL** pointers in the left and right directions.
- The tree itself only needs to remember the **root** pointer.

BST Node Structure

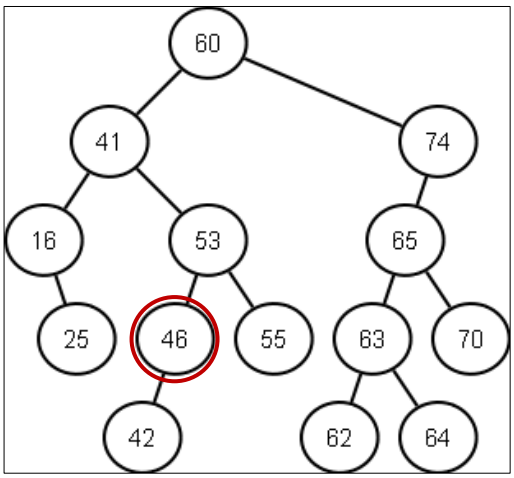
```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

Operations on BST

- **Searching**
- **Insertion**
- **Removal**
- **Traversal**
  - **Depth-First Traversal**
  - **Breadth-First Traversal**

Searching in a BST

- Suppose we are searching for **46** in the BST.
  1. Start at the **root** node.
  2. If the **root** is the **searched value**, return it.
  3. If the **searched value** is less than the root value, search the **left subtree**.
  4. If the **searched value** is greater than the root value, search the **right subtree**.
  5. If the **searched value** is not in the BST, return **None**.



Insertion in a BST

- To insert into a **Binary Search Tree (BST)**, we must maintain the nodes in sorted order.
- **There is only one place an item can go.**
- **Example:**

Insert the numbers **20, 10, 15, 25, 5** in to an initially empty binary search tree in the order listed.

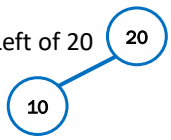
Insertion in a BST

1. Start at the **root** node.
2. Compare each node:
    - Is the inserted value lower than the current node? Go **left**.
    - Is the inserted value higher than the current node? Go **right**.
3. Continue to compare nodes with the new value until there is no right or left to compare with. That is where the new node is inserted.

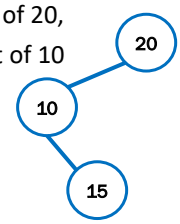
Example of Insertion: 20, 10, 15, 25, 5

Inserting **20** → Root

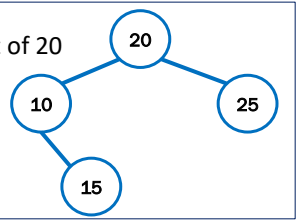
Inserting **10** → 10<20 → Left of 20



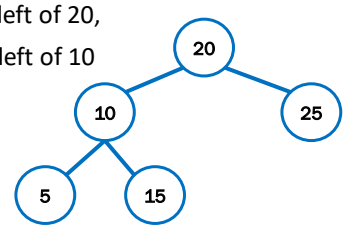
Inserting **15** → 15<20 →Left of 20,
15>10 →right of 10



Inserting **25** → 25>20 → right of 20

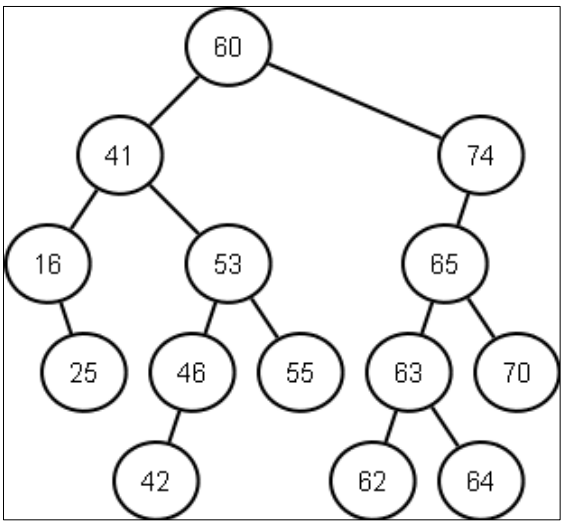


Inserting **5** → 5<20 → left of 20,
5<10 → left of 10



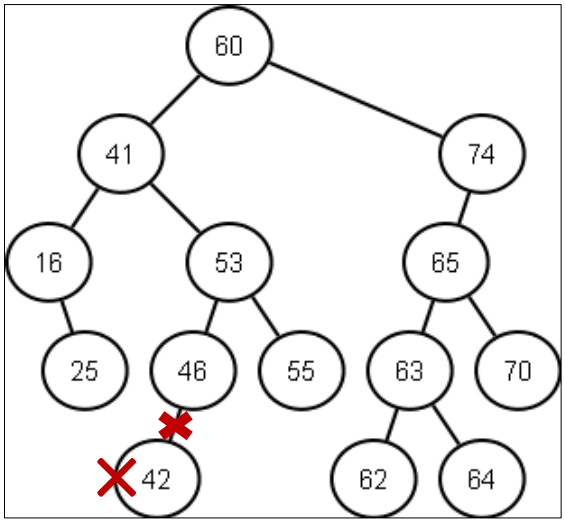
Removal from a BST

- To **delete a node**, we must be sure to **link up the subtree(s) of the node properly**.
  - Removing a **leaf node**
  - Removing a node with **one child**
  - Removing a node with **two children**



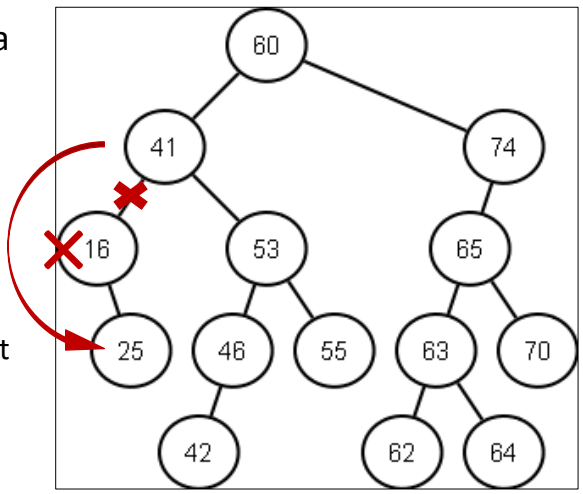
Removing a Leaf Node from a BST

- Suppose we are going to remove **42** (a leaf node) from the BST.
- All we need to do is:
  - Setting the left pointer of **46** to **Null**,
  - Deleting **42**.



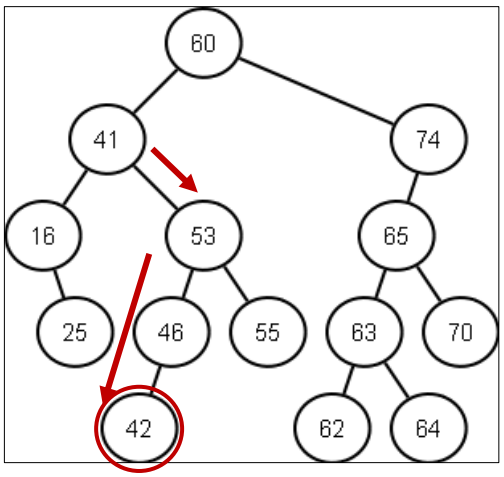
Removing a Node with Only One Child from a BST

- Suppose we are going to remove **16** (a node with only one child) from the BST.
- Node **16** has only one right child.
- All we need to do is:
  - Setting the pointer from the parent node (41) to the right child.
  - Deleting **16**.

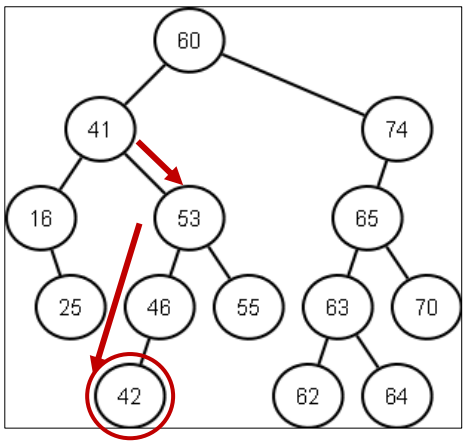


Removing a Node with Two Children from a BST

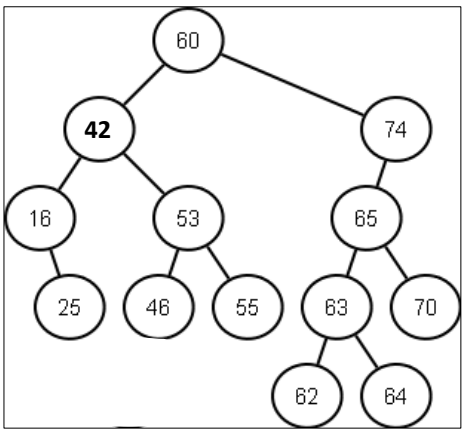
- Suppose we are going to remove **41** (a node with two children) from the BST.
- Node **41** has both left and right children.
- We need to find the next biggest number after **41**, then replace **41** with that number. How?
  - First, go to the right child of 41 (Node 53).
  - Then go as far left as possible (reaching 42).
  - Replace **41** with **42**.



Removing a Node with Two Children from a BST



After Removing 41
And replacing it with 42



Concept of Balance in Binary Search Tree

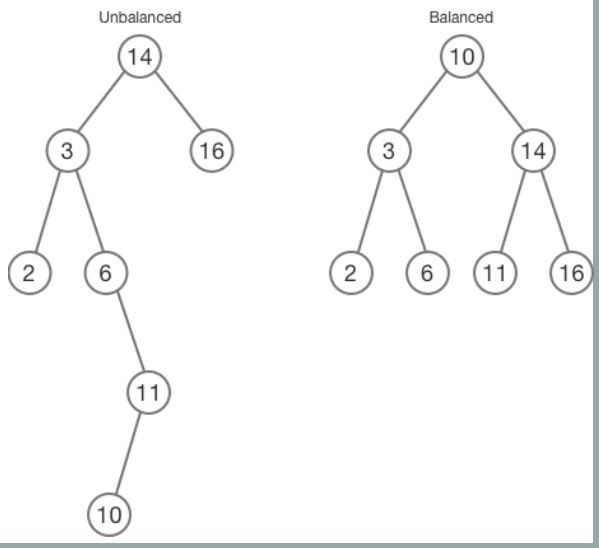
- A **Binary Search Tree** is balanced if the height of any node's left and right subtrees differs by at most one.

Balance = height(left subtree) – height(right subtree)

  - If **Balance <= 1**, the **Binary Search Tree** is called a **balanced BST**.
  - If **Balance > 1**, the **Binary Search Tree** is called an **unbalanced BST**.

Balanced BST vs. Unbalanced BST

- **Binary Search Trees** needing the search operation are fast, especially if the tree is a complete binary tree.
- **Balanced BST** operations (insertion, deletion, search) are efficient with a time complexity of O(log n).

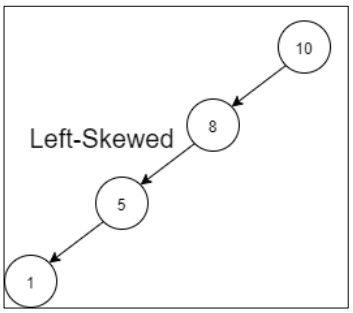


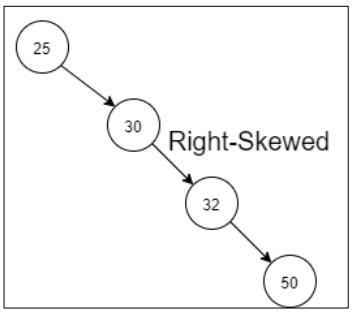
Why Balance Matters in BST?

- **Balanced BST**s maintain **O(log n)** time complexity for all operations (Searching, Insertion, Removal, etc.)
- **Unbalanced BST**s can degrade to **O(n)** time complexity in worst cases.
- There are some variants of BST, called **Self-balancing BST**s (like **AVL trees**, **Red-Black trees, Splay trees**). These BST variants maintain balance automatically through rotations.

Skewed BST

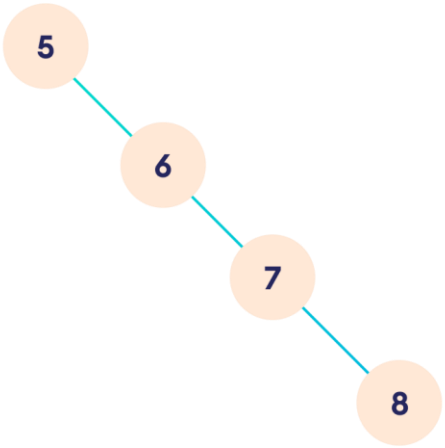
- **Left_Skewed BST** is an unbalanced BST in which all the nodes have a left child or no child at all. **All the right children are NULL**.
- **Right_Skewed BST** is an unbalanced BST in which all the nodes have a right child or no child at all. **All the left children are NULL**.





Disadvantage of Skewed BST

- A **skewed BST** suffers from significant performance degradation because it stops acting like a tree and starts behaving like a linear data structure, such as a linked list, with time complexity **O(n)** instead of **O(log n)** for most operations on a BST.



BST Operations' Time Complexity – Average & Worst Cases

Operation	Average Case	Worst Case (Skewed BST)
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$
Find Min/Max Value	$O(\log n)$	$O(n)$

Applications of Binary Search Tree

Database Indexing

BSTs are used in database management systems to index table columns for fast record retrieval.

File Systems

NTFS (Windows) and HFS+ (macOS) use variations of BSTs to organize directory entries and file metadata.

Network Routing

IP routing tables use radix trees (a BST variation) to match longest-prefix network addresses.

Statistics & Analytics

Special types of BSTs support rank and select queries in O(log n) time.