



**METROPOLITAN
TIRANA
UNIVERSITY**

Course: Data Structures and Algorithms

Trees

Evis Plaku

Why trees?

When your data structures have more
branches than your family tree...



Modern family tv show

Introduction to trees

Why trees?



Trees organize data in a **hierarchical structure**, resembling branching systems in nature



Benefits

- **Efficient searching:** fast data retrieval (e.g., BST)
- **Hierarchical representation:** ideal for modeling relationships (e.g., file systems)
- **Flexibility:** easy insertion, deletion, and updates
- **Optimized traversals:** efficient ways to visit all nodes

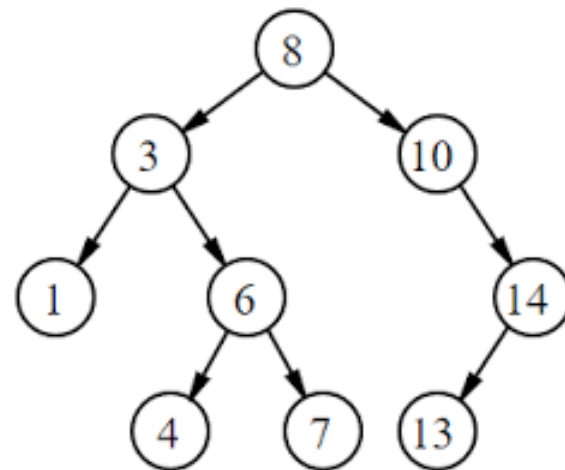
Use cases

- **File systems:** organize files and directories and **databases:** fast indexing and searching
- **Decision trees:** machine learning classification
- **Expression trees:** represent mathematical expressions
- **Game trees:** decision-making in games

Introduction to trees

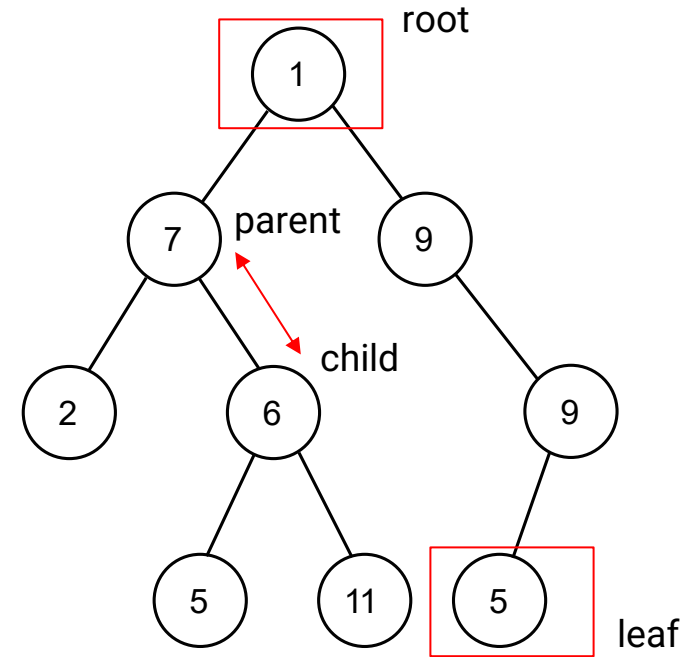
A hierarchical data structure consisting of nodes connected by edges

- A tree is a way to organize data in a hierarchy
- It has a **root** node, which is the starting point
- Each node can have **child nodes** that branch out
- No cycles: you can't go back to the same node by following the branches



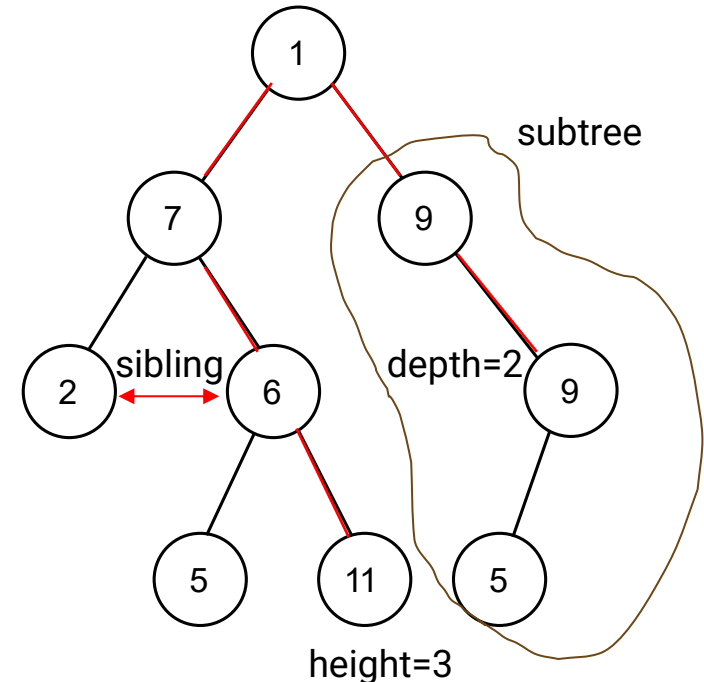
Terminology of trees

- **Node**: a basic unit of a tree that contains data
- **Root**: the topmost node in the tree, where the hierarchy starts
- **Leaf**: a node with no children, located at the "end" of branches
- **Parent**: a node that has one or more child nodes. **Child**: a node directly connected to another node above it (the parent)



Terminology of trees

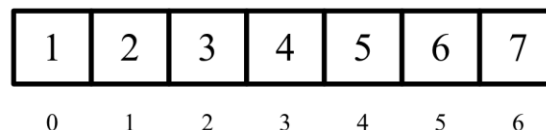
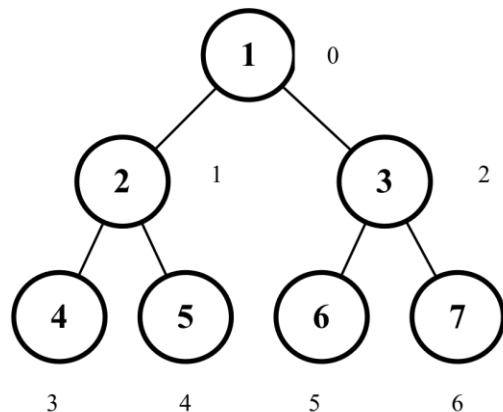
- **Sibling**: nodes that share the same parent
- **Depth**: the number of edges from the root to a node
- **Height**: the number of edges from a node to its farthest leaf
- **Level**: a node's level is distance from the root
- **Subtree**: a tree formed by a node and all its descendants



Array representation of trees

- The root is at index 0
- For any node at index i :
 - Left child is at index $2i + 1$
 - Right child is at index $2i + 2$

Trees can be represented
using an **array**



Array representation of trees

- The ArrayTree initializes an array tree to store the nodes
- Methods setValue and getValue are used to respectively set and retrieve node values
- Methods getLeftChild and getRightChild are used to respectively get the indices of the left and right child

```
public class ArrayTree {  
  
    private int[] tree; // Array to hold tree nodes  
    private int size; // Size of the tree  
  
    public ArrayTree(int size) {  
        this.size = size;  
        this.tree = new int[size];  
    }  
  
    public void setValue(int index, int value) {  
        // TODO: Add logic to set value at the given index  
    }  
  
    public int getValue(int index) {  
        // TODO: Add logic to return the value at the given index  
        return -1; // Placeholder  
    }  
  
    public int getLeftChild(int index) {  
        // TODO: Add logic to calculate left child index  
        return -1; // Placeholder  
    }  
  
    public int getRightChild(int index) {  
        // TODO: Add logic to calculate right child index  
        return -1; // Placeholder  
    }  
}
```

Linked list representation of trees

- Each node in the tree is an object that contains:
 - **Data**: the value stored in the node
 - **Left child pointer**: a reference to the left child node
 - **Right child pointer**: a reference to the right child node

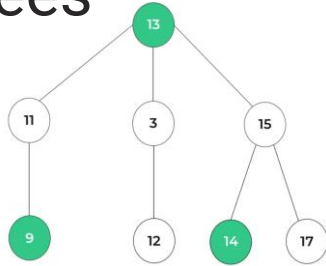
```
public class TreeNode {  
  
    int data;  
    TreeNode leftChild;  
    TreeNode rightChild;  
  
    public TreeNode(int data) {  
        this.data = data;  
        this.leftChild = null;  
        this.rightChild = null;  
    }  
}
```

No predefined size, unlike array representation.

The tree grows and shrinks as needed.

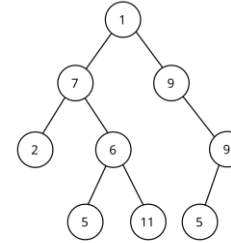
Easy to insert, delete, and manipulate nodes

Types of trees



General trees

- **A tree where each node can have any number of children**
- No restriction on the number of children a node can have
- Used for more complex hierarchical structures



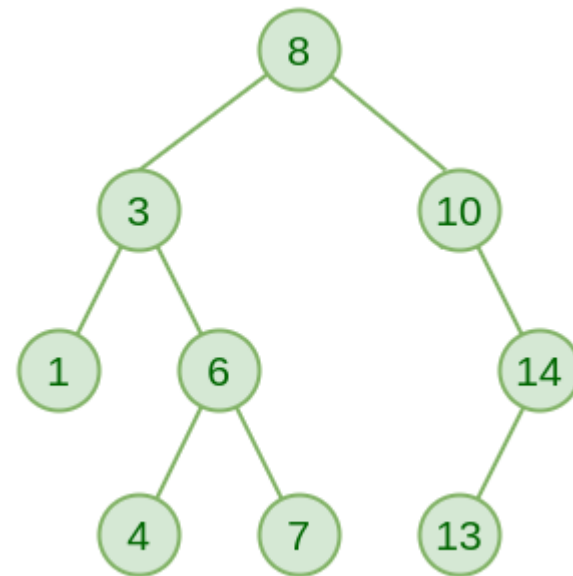
Binary trees

- **A tree where each node has at most two children**
- Each node has two child pointers (left and right)
- Common in computer science due to simplicity and efficiency in operations

Binary Search Trees (BST)

Binary search trees: properties and applications

- Binary Search Tree (BST): a binary tree where for every node
 - The left child's value is less than the node's value
 - The right child's value is greater than the node's value
- Searching for a value has an average time complexity of $O(\log n)$ if the tree is balanced
- All nodes follow the *left* < *parent* < *right* rule



Applications of binary search trees



Database indexing



File system directories



Online ticketing systems



Game leaderboards



Navigation systems

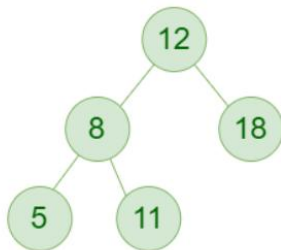


Online shopping filters

Full trees and perfect trees

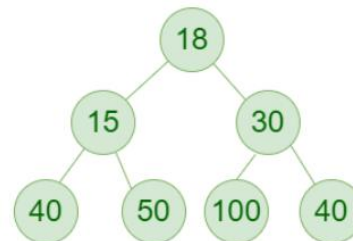
Full binary tree

- A binary tree where every node has either 0 or 2 children
- All internal nodes (non-leaf nodes) must have exactly two children
- Leaf nodes have no children



Perfect binary tree

- A binary tree where all leaf nodes are at the same level, and every internal node has exactly two children
- All levels are fully populated
- The height of the tree is well-defined

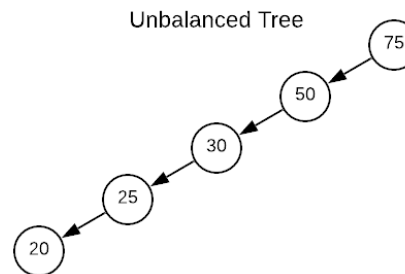
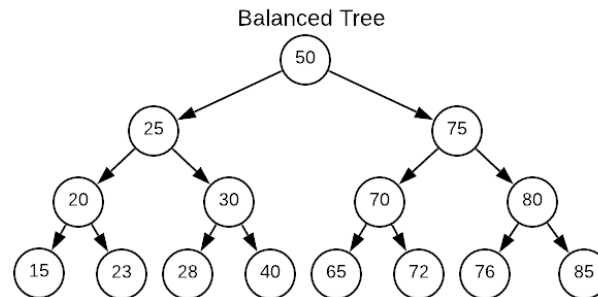


Balanced binary search trees



A Binary Search Tree is balanced when the height difference between left and right subtrees is at most 1

- Ensures $O(\log n)$ time complexity for operations
- Prevents degeneration into a linear structure
- Maintains optimal memory and space usage
- Supports efficient algorithms

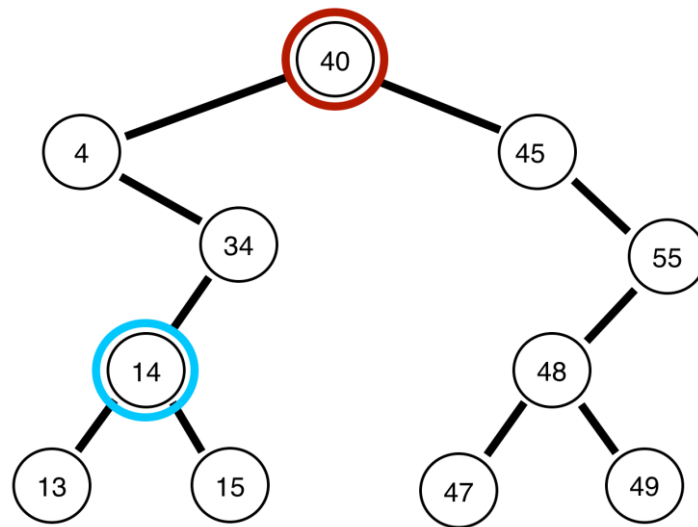


Key operations in BSTs: insertion

- Start at the root
 - Compare the value with the current node
 - If the value is less, move to the left child; if greater, move to the right child
 - Repeat until you find an empty spot (null) where the new node will be inserted
- $O(h)$ where h is the height of the tree
- Best case: $O(\log n)$ for balanced tree
 - Worst case: $O(n)$ for skewed tree

Key operations in BSTs: searching

- Start at the root
- Compare the target value with the current node's value
- If less, move to the left child; if greater, move to the right child
- Repeat until the target is found or you reach a null (value not in tree)

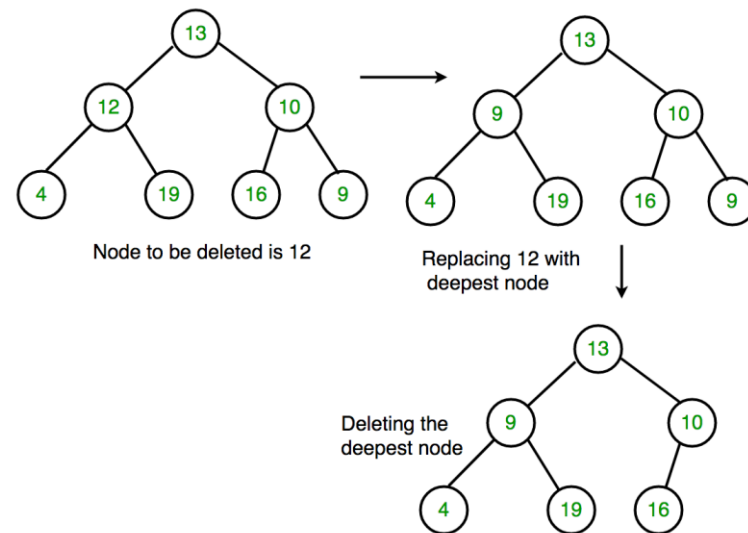


$O(h)$ where h is the height of the tree

- Best case: $O(\log n)$ for balanced tree
- Worst case: $O(n)$ for skewed tree

Key operations in BSTs: deletion

- Case 1: node has no children (leaf): simply remove the node
- Case 2: node has one child: replace the node with its child
- Case 3: node has two children: find the in-order successor (or predecessor), replace the node with it, and remove the successor



$O(h)$ where h is the height of the tree

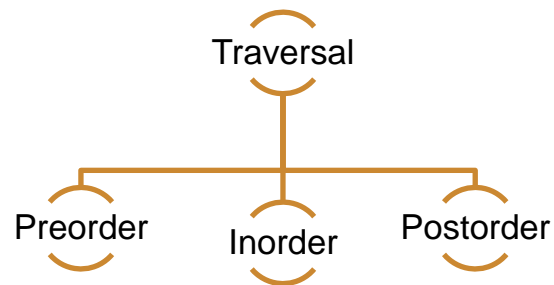
- Best case: $O(\log n)$ for balanced tree
- Worst case: $O(n)$ for skewed tree

Traversals in Binary Search Trees



Traversal means visiting all nodes in a tree in a systematic way

- Traversals allow us to search, print, or manipulate every node in a desired order
- Traversals help visualize and debug tree structure during operations like insertions or deletions
- Traversals define how data is saved to or read from a tree (e.g., saving a tree to a file)

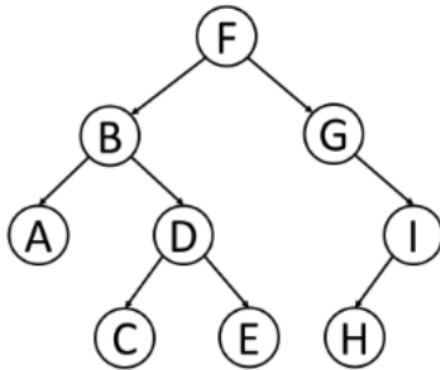


Order matters!

Preorder traversal (Node → Left → Right)

Order

1. Visit the current node
2. Traverse the left subtree
3. Traverse the right subtree



Preorder:

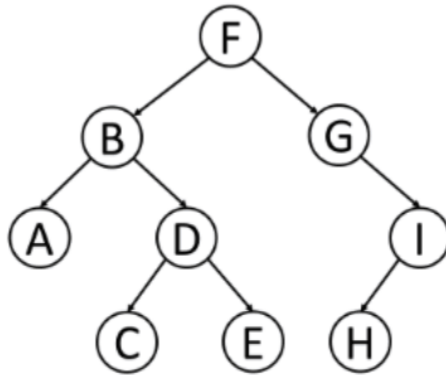
--	--	--	--	--	--	--	--	--	--

- **Copying or cloning trees** (ensures the structure is preserved starting from the root)
- **Serializing trees** (preorder is used to flatten trees into strings or arrays for storage and reconstruction)
- **Evaluating prefix expressions** (preorder visits operators before operands)

Inorder traversal (Left → Node → Right)

Order of traversal

1. Traverse the left subtree
2. Visit the current node
3. Traverse the right subtree



Inorder:

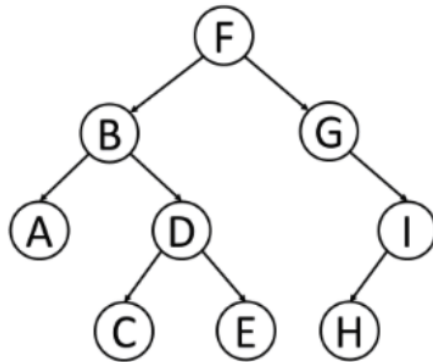
--	--	--	--	--	--	--	--	--	--

- **Retrieving sorted data** (used to extract elements from a BST in non-decreasing order)
- **BST validation** (if inorder traversal yields a non-sorted list, the tree violates BST properties)
- **Infix expressions evaluation** (especially in expression trees for mathematical expressions)

Postorder traversal (Left → Right → Node)

Order of traversal

1. Traverse the left subtree
2. Traverse the right subtree
3. Visit the current node



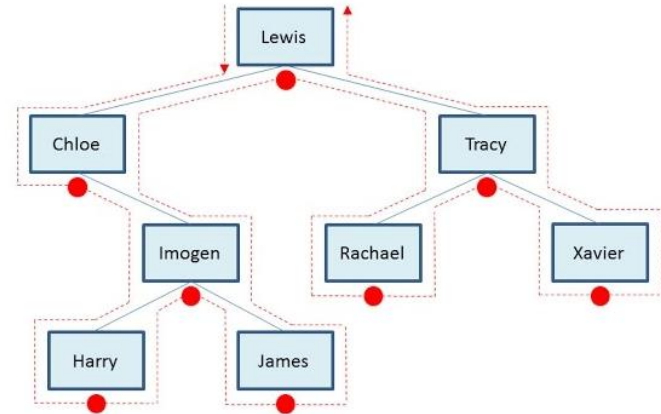
Postorder:

--	--	--	--	--	--	--	--	--	--

- **Tree deletion** (safely deletes child nodes before the parent)
- **Recursive tree operations** (great for structural modifications like pruning, transformations)
- **Postfix expressions evaluation** (in expression trees, postorder supports evaluating mathematical expressions)

Example of traversal in real-world use cases

- Assume a BST is storing Student or Employee data, sorted by id
- **Inorder** traversal is useful retrieving for all records in ascending order (generate reports, ranked lists etc)
- **Preorder** traversal is useful to serialize or transfer data (export the tree to a file)



- **Postorder** traversal is useful to remove records (e.g., delete students graduation this year)

Operations complexity in BST

Operation	Balanced BST	Unbalanced BST
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Traversal	$O(n)$	$O(n)$



Balanced BSTs ensure efficient operations with $O(\log n)$ time

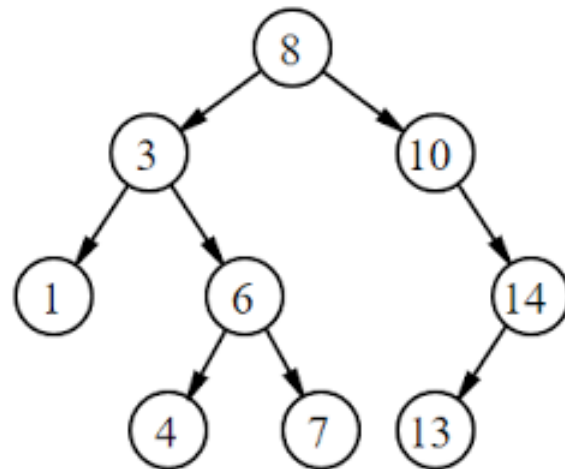
Unbalanced BSTs degrade to $O(n)$ time in worst cases

Key Takeaways







Trees are hierarchical structures ideal for representing nested or ordered data

- Binary Search Trees (BSTs) maintain a sorted structure: **Left < Root < Right**
- BST operations like search, insert, and delete are efficient when the tree is balanced
- BSTs are widely used in databases, indexing, and building efficient search systems



| Helpful resources on recursion

-  [VisualAlgo](#) to visualize BST tree and operations
-  [Software testing](#) tutorial on Binary Search Trees in Java
-  [Implementing](#) a Binary Search Tree in Java
-  [LeetCode](#) set of problems on trees

In a tree, roots come first
just like in life

