Trees

A Introduction to Trees

A tree is a non-linear data structure that represents hierarchical relationships between elements. It consists of nodes connected by edges, with a single node designated as the root. Each node can have zero or more child nodes, and nodes without children are called leaves. Trees are widely used in computer science for various applications, including representing hierarchical data, organizing information, and implementing efficient search algorithms.
The terminology used in trees includes:

- **Tree:** A collection of nodes connected by edges, with a hierarchical structure.
- **Node:** An individual element in a tree that contains data and may have links to other nodes.
- **Root:** The topmost node in a tree, which has no parent.
- **Child:** A node that is directly connected to another node when moving away from the root.
- **Parent:** A node that has one or more children.
- **Leaf:** A node that does not have any children.
- **Subtree:** A tree formed by a node and all its descendants.
- **Height:** The length of the longest path from the root to a leaf.
- **Depth:** The length of the path from the root to a specific node.
- **Degree:** The number of children a node has.

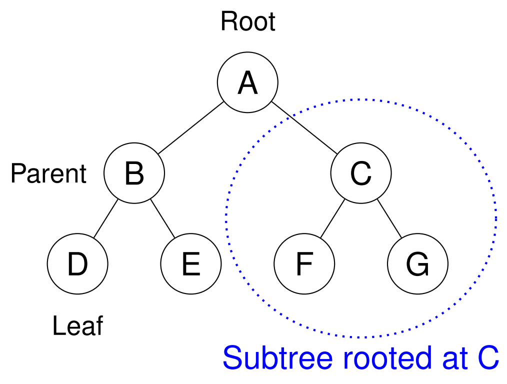


Figure 1: An example of a tree showing root, parent, leaf, and subtree.

For exmple, in the example tree shown in Figure 1:

- **Root:** Node A is the root of the tree; it has no parent.
- **Parent and Child:** Node B is a child of A and a parent to D and E.
- **Leaf:** Nodes D, E, F, and G are leaves; they have no children.
- **Subtree:** The subtree rooted at C includes nodes C, F, and G.

- **Level/Depth:** The root is at level 0, its children (B, C) at level 1, and their children (D, E, F, G) at level 2.
- **Height:** The height of the tree is 2, which is the length of the longest path from the root (A) to a leaf (D, E, F, or G).
- **Degree:** The degree of node A is 2 (it has two children: B and C), the degree of node B is 2 (it has two children: D and E), and the degree of nodes D, E, F, and G is 0 (they have no children).

Trees can be classified into various types based on their structure and properties. The choice of tree type depends on the specific requirements of the application, such as search efficiency, memory usage, and balancing. In this document, we will explore some common types of trees, including binary trees, binary search trees (BST), AVL trees, and red-black trees.

Practice Exercise:

Consider the following systems, each of which uses a different data structure:

- **File System Hierarchy:** Organizes files and folders in a nested relationship.
- **Undo/Redo Functionality in Text Editors:** Maintains a history of actions.
- **Customer Service Call Queue:** Handles incoming calls in the order they arrive.
- **Web Browser History:** Allows navigation back and forth through visited pages.
- **Company Organizational Chart:** Represents reporting relationships between employees and managers.

For each system above, reflect on why a tree or a linear data structure (list, stack, or queue) is appropriate. What would be the challenges or limitations if you used a different type of structure? Discuss your thoughts with your classmates and be ready to share your insights.

B Binary Trees

A binary tree is a special type of tree where each node has at most two children, referred to as the left child and the right child. Binary trees are widely used in computer science for various applications, including search algorithms, expression parsing, and data storage.

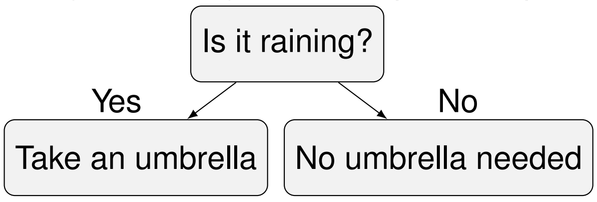


Figure 2: A simple decision tree example.

Figure 2 illustrates a simple decision tree used for making decisions based on conditions. In this example, the root node asks whether it is raining, and based on the answer, it leads to two possible outcomes: taking an umbrella or not needing one.
Applications of binary trees include:

- Decision-making processes: Decision trees are used to model decisions and their possible consequences, helping in decision analysis, machine learning, and classification or regression tasks.

- Hierarchical data representation: Binary trees can represent hierarchical structures such as organizational charts, file systems, and family trees.
- Expression parsing and evaluation: Binary trees can represent mathematical expressions, where internal nodes are operators and leaf nodes are operands. This allows for efficient evaluation of expressions.
- Huffman coding trees: Binary trees are used in Huffman coding for data compression, where the tree structure represents variable-length codes for characters based on their frequencies.

C Binary Search Trees (BST)

A binary search tree (BST) is a specialized type of binary tree that maintains a specific order among its nodes. In a BST, for each node:

- All values in the left subtree are less than the value of the node.
- All values in the right subtree are greater than the value of the node.

This property allows for efficient searching, insertion, and deletion operations in logarithmic time on average.

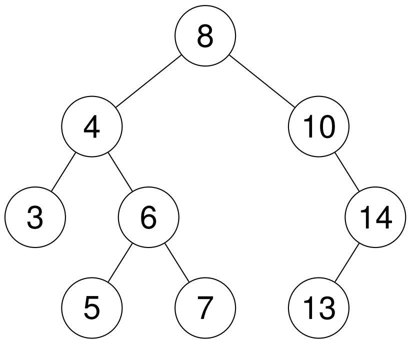


Figure 3: An example of a binary search tree (BST).

Figure 3 illustrates a binary search tree. It represents a collection of integers where each node follows the BST property. For example, the left child of node 8 (which is 3) is less than 8, and the right child (which is 10) is greater than 8. This structure allows for efficient searching, insertion, and deletion operations. For instance, to search for the value 6, we start at the root (8), move left to 3, then right to 6, finding the value in three comparisons.
BSTs are used in various applications, including:

- Maintaining a sorted collection of elements, allowing for efficient searching, insertion, and deletion operations.
- Implementing associative arrays or dictionaries, where keys are stored in a sorted manner.
- Supporting dynamic sets of data that require frequent updates while maintaining order.
- Serving as the basis for more advanced data structures like AVL trees and red-black trees, which provide additional balancing properties.

Operations on Binary Search Trees Binary Search Trees (BSTs) support several fundamental operations that allow efficient management of ordered data. The most common operations are:

- **Search:** Find whether a value exists in the BST. Starting from the root, compare the target value with the current node. If equal, the value is found. If less, move to the left child; if greater, move to the right child. Repeat until the value is found or a leaf is reached.
- **Insertion:** Add a new value to the BST. Start at the root and traverse the tree as in search, moving left or right depending on the value. Insert the new node at the appropriate child position, maintaining the BST property.
- **Deletion:** Remove a value from the BST. There are three cases:
  - The node is a leaf: Simply remove it.
  - The node has one child: Replace the node with its child.
  - The node has two children: Find the node's inorder successor (smallest value in the right subtree) or inorder predecessor (largest value in the left subtree), replace the node's value with it, and delete the successor/predecessor node.
- **Traversal:** Visit all nodes in a specific order. Common traversals include, taking the tree in Figure 3 as an example:
  - **Inorder (Left, Root, Right):** Produces values in sorted order. The tree of Figure 3 would yield the sequence 3, 4, 5, 6, 7, 8, 10, 13, 14.
  - **Preorder (Root, Left, Right):** Useful for copying the tree. The sequence for the tree would be 8, 4, 3, 6, 5, 7, 10, 14, 13.
  - **Postorder (Left, Right, Root):** Useful for deleting the tree. We get the sequence 3, 5, 7, 6, 4, 13, 14, 10, 8.

Time Complexity:

- In a balanced BST, search, insertion, and deletion all have average and worst-case time complexity of $O(\log n)$, where n is the number of nodes.
- In a degenerate (unbalanced) BST, these operations can degrade to $O(n)$.

Insertion and Deletion: Suppose we have the BST from Figure 3. To insert the value 1:

- Start at the root (8), move left to 4, then left to 3. Since 3 has no left child, insert 1 as the left child of 3.

To delete the value 6:

- Find node 6. It has two children (5 and 7). Replace 6 with its inorder successor (7), and remove node 7.

These operations are illustrated in Figure 4.
A binary search tree (BST) is a specialized type of binary tree that maintains a specific order among its nodes. In a BST, for each node:

- All values in the left subtree are less than the value of the node.
- All values in the right subtree are greater than the value of the node.

This property allows for efficient searching, insertion, and deletion operations in logarithmic time on average.

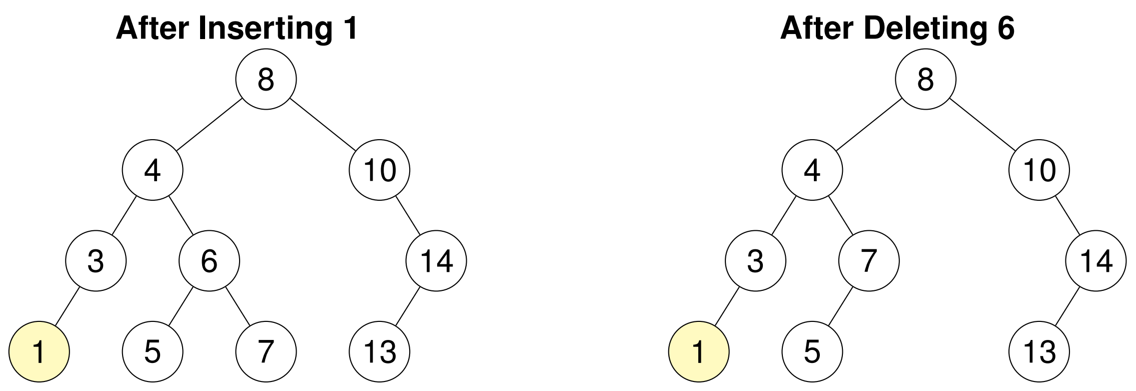


Figure 4: BST after inserting 1 (left) and after deleting 6 (right).

Applications BSTs are used in various applications, including:

- Maintaining a sorted collection of elements, allowing for efficient searching, insertion, and deletion operations.
- Implementing associative arrays or dictionaries, where keys are stored in a sorted manner.
- Supporting dynamic sets of data that require frequent updates while maintaining order.
- Serving as the basis for more advanced data structures like AVL trees and red-black trees, which provide additional balancing properties.

Practice Exercise:

1. Construct a binary search tree by insering the following values in order: 8, 4, 10, 3, 6, 14, 5, 7, 13.
2. Draw the resulting BST after each insertion.
3. Perform Inorder, Preorder, and Postorder traversals on the final BST and write down the resulting sequences.
4. Delete the value 6 and the value 3 from the BST. Draw the resulting BST after each deletion.

D AVL Trees

An AVL tree is a self-balancing binary search tree that maintains its balance through rotations during insertion and deletion operations. The balance condition ensures that the heights of the two child subtrees of any node differ by at most one, leading to efficient search, insert, and delete operations.

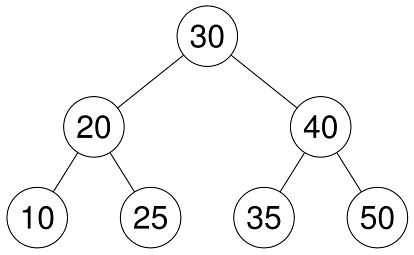


Figure 5: An example of an AVL tree.

The AVL tree in Figure 5 maintains the AVL property, where the heights of the left and right subtrees of each node differ by at most one. For example, the height of the left subtree rooted at node 20 is 2, while the height of the right subtree rooted at node 40 is also 2, maintaining balance. This is called the **balance factor** given by the formula:

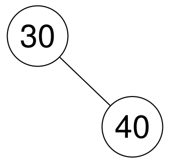
$$\text{Balance Factor} = \text{Height of Left Subtree} - \text{Height of Right Subtree}$$

AVL trees are named after their inventors, Adelson-Velsky and Landis, who introduced them in 1962. They are one of the first self-balancing trees and remain widely used due to their efficient operations. Balancing is achieved through rotations, which are performed during insertion and deletion to maintain the AVL property.

Operations on AVL Trees A node is considered unbalanced if the height difference between its left and right subtrees exceeds one. To restore balance, rotations are performed. The balance factor of a node is defined as the height of the left subtree minus the height of the right subtree. The possible values for the balance factor are -1, 0, or +1. Rotations in AVL trees can be classified into:

- **Single Rotation:** A single rotation is performed when a node becomes unbalanced due to an insertion or deletion in one of its subtrees. There are two types:
  - **Left Rotation:** Used when the right subtree is taller. It involves rotating the unbalanced node to the left, making its right child the new root of the subtree.
  - **Right Rotation:** Used when the left subtree is taller. It involves rotating the unbalanced node to the right, making its left child the new root of the subtree.
- **Double Rotation:** A double rotation is performed when a node becomes unbalanced due to an insertion or deletion in the opposite subtree. There are two types:
  - **Left-Right Rotation:** A left rotation followed by a right rotation.
  - **Right-Left Rotation:** A right rotation followed by a left rotation.

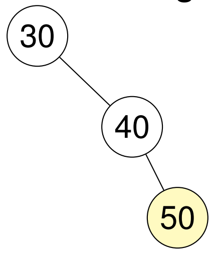
In what follows, we will illustrate how these rotations work with examples using this AVL tree:



This tree is balanced with a height of 1, where the left subtree has a height of 0 and the right subtree rooted at 40 has a height of 1. Since the balance factor of node 30 is -1 (0 - 1 = -1), it is considered balanced.
In this AVL tree, we said that node 30 is **right-heavy** because its right subtree (rooted at 30) is taller than its left subtree (none). This can be caracterized by a negative balance factor of -1 for node 30. Similarly, we will say that a node is **left-heavy** if its left subtree is taller than its right subtree, characterized by a positive balance factor of +1.

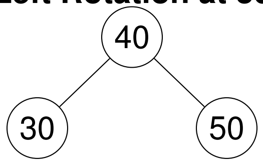
Left Rotation in AVL Tree Let’s insert the value **50** into the AVL tree above. This will cause an imbalance that requires a left rotation to restore balance. First, applying the same insertion process as in a binary search tree, we get:

After Inserting 50



After inserting 50, the tree becomes unbalanced at node 30 (balance factor = -2). Since node 30 is right-heavy and its right child (node 40) is also right-heavy, we perform a **left rotation** at node 30 to restore balance:

Left Rotation at 30



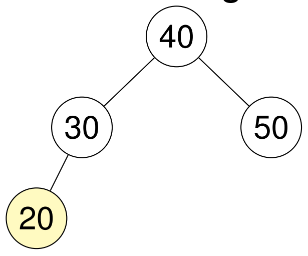
The left rotation involves the following steps:

- Node 40 becomes the new root of the subtree.
- Node 30 becomes the left child of node 40.
- Node 30 take the ownership of the left child of node 40, which is missing in this case.

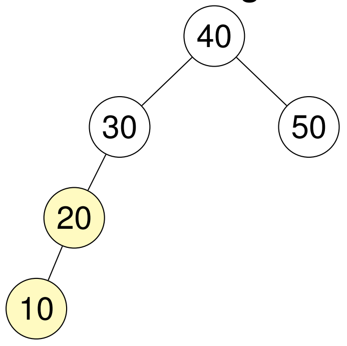
The resulting tree is now balanced, with a height of 1. The balance factor of node 40 is 0 (1 - 1 = 0), and the balance factor of nodes 30 and 50 is also 0 (0 - 0 = 0). This ensures that the AVL tree property is satisfied.

Right Rotation in AVL Tree Let's expand the AVL tree further by inserting the value **20**, then the value **10**. This will cause an imbalance that requires a right rotation to restore balance.

After Inserting 20

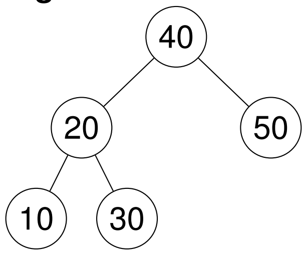


After Inserting 10



After these insertions, the tree becomes unbalanced at node 30 (balance factor = +2). Since node 30 is left-heavy and its left child (node 20) is also left-heavy, we perform a **right rotation** at node 30 to restore balance:

Right Rotation at 30



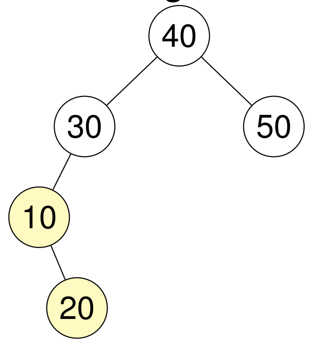
The right rotation involves the following steps:

- Node 20 becomes the new root of the subtree.
- Node 30 becomes the right child of node 20.
- Node 30 takes the ownership of the right child of node 20, which is missing in this case.

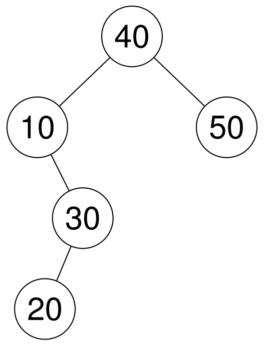
The resulting tree is now balanced, with a height of 2. The balance factor of node 30 is 0 (1 - 1 = 0), the balance factor of node 40 is 0 (0 - 1 = -1), and the balance factor of node 20 is 0 (0 - 0 = 0). This ensures that the AVL tree property is satisfied.

Double Rotation in AVL Tree Instead of inserting 20 then 10, let's insert the value **10** first, then the value **20**. This will cause an imbalance that requires a double rotation to restore balance.

After Inserting 10 then 20

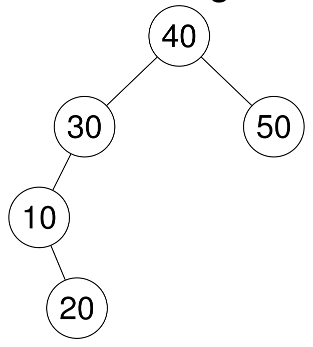


If we do a simple right rotation at node 30, we would get the following tree:

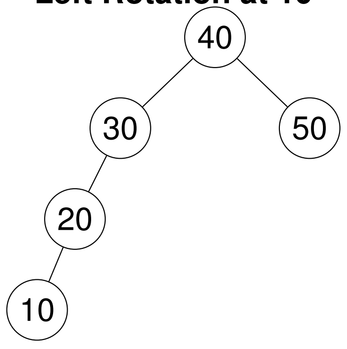


Indeed, the right rotation implies that node 30 takes the ownership of the right child of node 10, that is node 20. The resulting tree is still unbalanced at node 10 (balance factor = -2), which means that we need to perform a double rotation to restore balance.
Since node 30 is left-heavy and its left child (node 10) is right-heavy, we perform a **left-right rotation**, that is, a left rotation at node 10 followed by a right rotation at node 30:

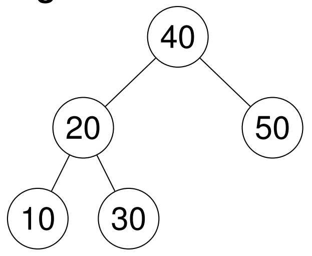
Initial Tree (Before Left-Right Rotation at 10)



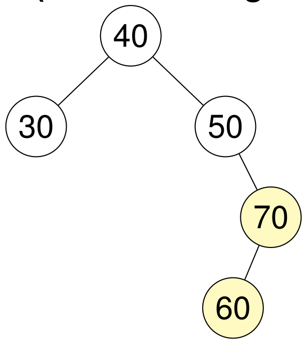
Left Rotation at 10

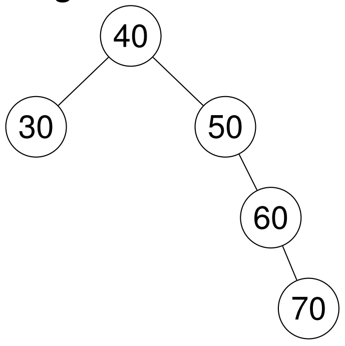


Right Rotation at 30



The resulting tree is now balanced, with a height of 2. The balance factor of node 20 is 0 (0 - 0 = 0), and the balance factor of node 40 is 1 (2 - 1 = 1). This ensures that the AVL tree property is satisfied.
Similarly, if we had inserted the value 70 then the value 60, we would have needed a double rotation to restore balance. The process would be similar, involving a right rotation followed by a left rotation:


Initial Tree (After Inserting 70 then 60)


Right Rotation at 70

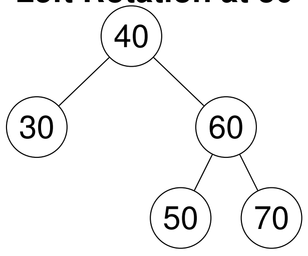

Left Rotation at 50

Table 1 summarizes the different types of rotations in AVL trees, showing the tree structure before and after each rotation.

Practice Exercise:

Given the following sequence of values: 10, 20, 30, 25, 15, 40, 35, insert them into an AVL tree and show the resulting tree after each insertion. After each insertion:

- Draw the tree.
- Calculate the balance factor for each node.
- Determine if a rotation is needed (single or double).
- Apply the necessary rotation(s) and redraw the tree.

E Red-Black Trees

A red-black tree is a type of self-balancing binary search tree that maintains balance through color properties. Each node is colored either red or black, and the tree satisfies specific properties that ensure it remains balanced during insertions and deletions.

- Each node is either red or black.
- The root node is always black.
- All leaves (NIL nodes) are black.
- If a red node has children, both children must be black (no two red nodes can be adjacent).

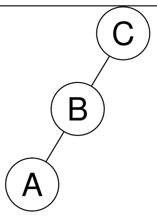
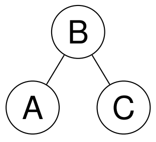
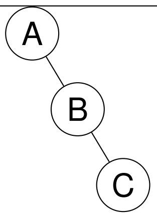
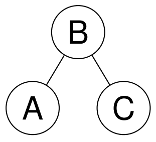
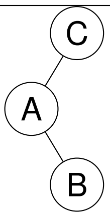
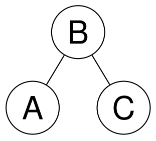
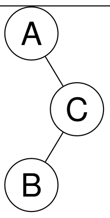
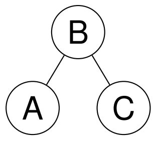
Rotation Type	Before Rotation	After Rotation
Right Rotation		
Left Rotation		
Left-Right Rotation		
Right-Left Rotation		

Table 1: AVL tree rotations

- Every path from a node to its descendant leaves must have the same number of black nodes (black height).

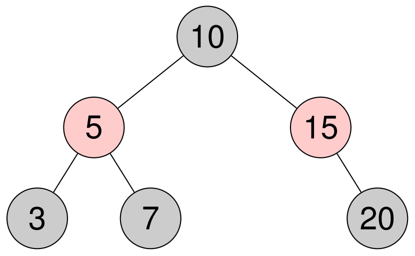


Figure 6: An example of a red-black tree.

Figure 6 illustrates a red-black tree. The root node (10) is black, and its children (5 and 15) are red. The left child of 5 (3) and the right child of 15 (20) are black. This structure satisfies all the properties of a red-black tree, ensuring balance during operations.
The balancing property of red-black trees is measured by the **black height**, which is the number of black nodes from a node to its descendant leaves. This property ensures that the longest path from the root to a leaf is no more than twice as long as the shortest path, maintaining logarithmic height.
A detailed discussion of red-black trees is beyond the scope of this course. However, it is important to note that red-black trees are widely used in practice, including applications such as:

- Implementing associative arrays or dictionaries, where keys are stored in a sorted manner.

- Maintaining a sorted collection of elements, allowing for efficient searching, insertion, and deletion operations.
- Supporting dynamic sets of data that require frequent updates while maintaining order.
- Serving as the basis for more advanced data structures like B-trees and B+ trees, which are used in databases and file systems.

Operations on Red-Black Trees Red-black trees support several fundamental operations that allow efficient management of ordered data. The most common operations are:

- **Search:** Find whether a value exists in the red-black tree. The search operation is similar to that of a binary search tree, starting from the root and traversing left or right based on comparisons.
- **Insertion:** Add a new value to the red-black tree. The insertion process involves inserting the new node as in a binary search tree, then recoloring and performing rotations to maintain the red-black properties.
- **Deletion:** Remove a value from the red-black tree. The deletion process is more complex than insertion, involving recoloring and rotations to maintain balance after removing a node.
- **Traversal:** Visit all nodes in a specific order. Common traversals include inorder, preorder, and postorder, similar to binary search trees.
- **Balancing:** After insertion or deletion, the tree may become unbalanced. Red-black trees use rotations and recoloring to restore balance while maintaining the red-black properties.