DATA STRUCTURES AND ALGORITHMS

- LECTURE 8-

(2ND YEAR OF STUDY)

Skopje, 2023/24

Contents

8. Trees

8.1. Introduction

8.2. Types of Tree Data Structures

8.3. Binary Search Trees

8.4. AVL Trees

8.5. Red-Black Trees

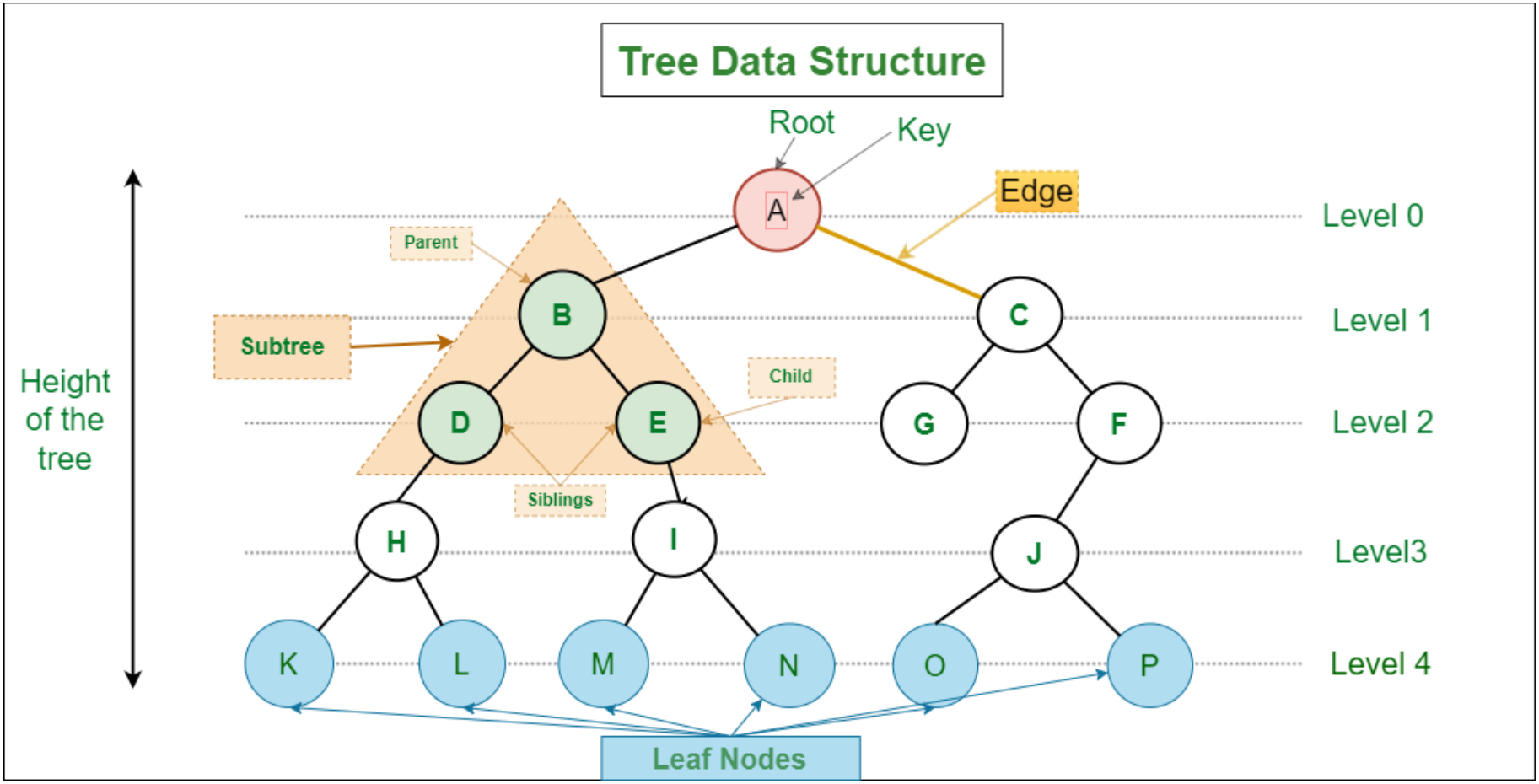
8.6. Applications of Trees

8.1. Introduction

- A **Tree data structure** is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search.
- It is a collection of ***nodes*** that are connected by ***edges*** and has a hierarchical relationship between the nodes.
- The topmost node of the tree is called the **root**, and the nodes below it are called the **child** nodes. Each node can have multiple child nodes, forming a recursive structure.

8.1. Introduction

- *Tree data structure* has ***root***, ***branches***, and ***leaves*** connected with one another.



8.1. Introduction

- **Basic Terminologies In Tree Data Structure:**
  - **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. **{B}** is the parent node of **{D, E}**.
  - **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: **{D, E}** are the child nodes of **{B}.**
  - **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. **{A}** is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.

8.1. Introduction

- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. **{K, L, M, N, O, P, G}** are the leaf nodes of the tree.
- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. **{A,B}** are the ancestor nodes of the node **{E}**.
- **Descendant:** Any successor node on the path from the leaf node to that node. **{E,I}** are the descendants of the node **{B}.**

8.1. Introduction

- **Sibling:** Children of the same parent node are called siblings. **{D,E}** are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level **0**.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree:** Any node of the tree along with its descendant.

8.1. Introduction

- A ***tree*** consists of a ***root***, and zero or more ***subtrees*** T_1, T_2, … , T_k such that there is an ***edge*** from the root of the tree to the root of each subtree.
- Representation of a Node in Tree Data Structure:

```
struct Node {
        int data;
        struct Node* first child;
        struct Node* second child;
        …
        struct Node* n-th child;
    }
```

8.1. Introduction

- **Number of edges:** An ***edge*** can be defined as the connection between two ***nodes***. If a tree has *N* nodes then it will have (*N-1*) edges. There is *only* one path from each node to any other node of the tree.
- **Depth of a node:** The depth of a node is defined as the *length* of the path from the root to that node. Each edge adds 1 unit of length to the path. So, it can also be defined as the *number* of edges in the path from the root of the tree to the node.

8.1. Introduction

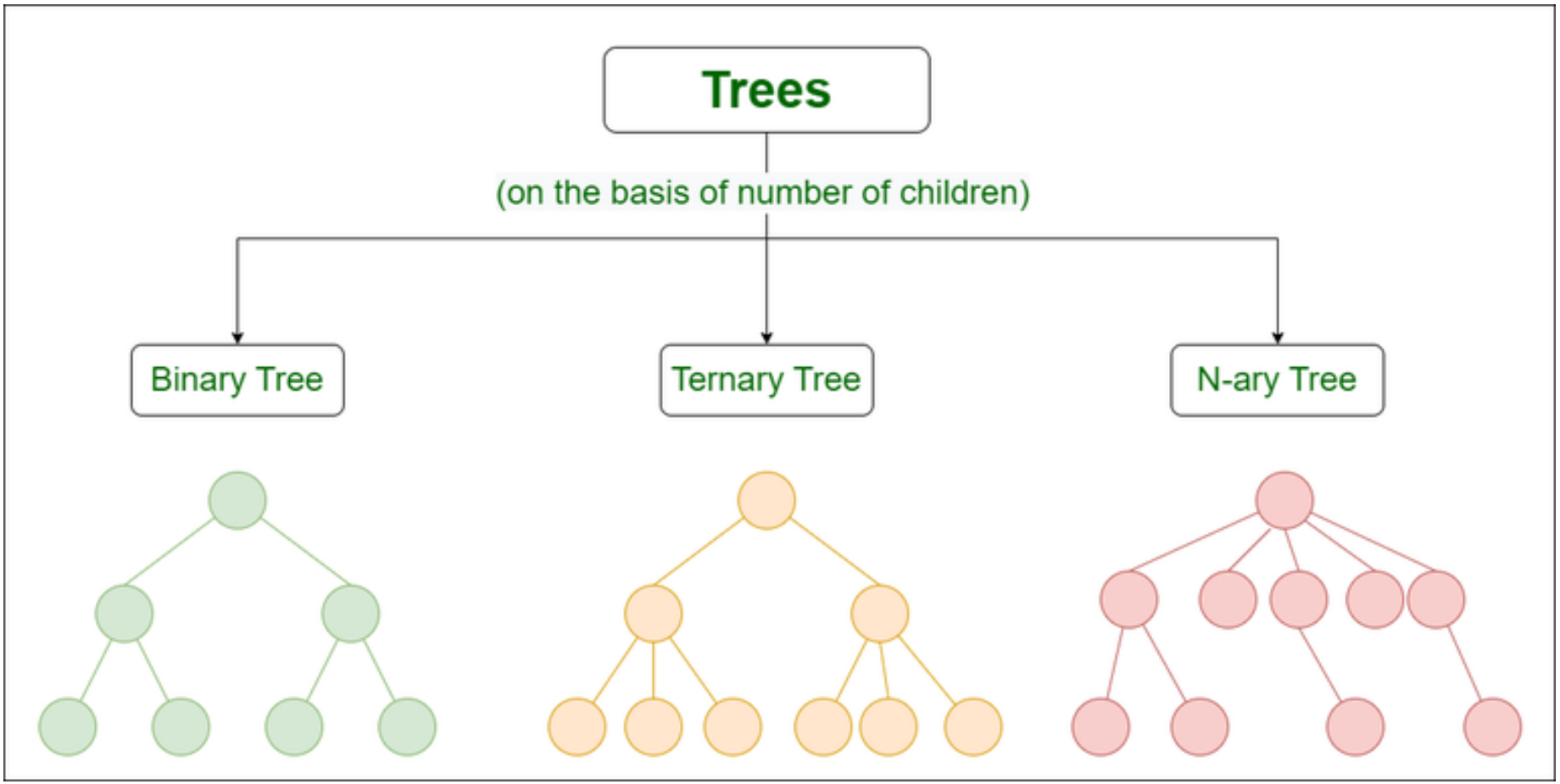
- **Height of a Node:** The height of a node can be defined as the *length* of the longest path from the node to a leaf node of the tree.
- **Height of the Tree:** The height of a tree is the *length* of the longest path from the root of the tree to a leaf node of the tree.
- **Degree of a Node:** The total *count* of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be **0**. The degree of a tree is the maximum degree of a node among all the nodes in the tree.

8.1. Introduction

- **Basic Operation Of Tree Data Structure:**
  - **Create** – Creates a tree in the data structure.
  - **Insert** – Inserts data in a tree.
  - **Search** – Searches specific data in a tree to check whether it is present or not.
  - **Traversal** – Tree traversal means going through every tree node ***exactly once*** and performing some operation upon its contents.

8.2. Types of Tree Data Structures

- The Trees can be classified into the types:

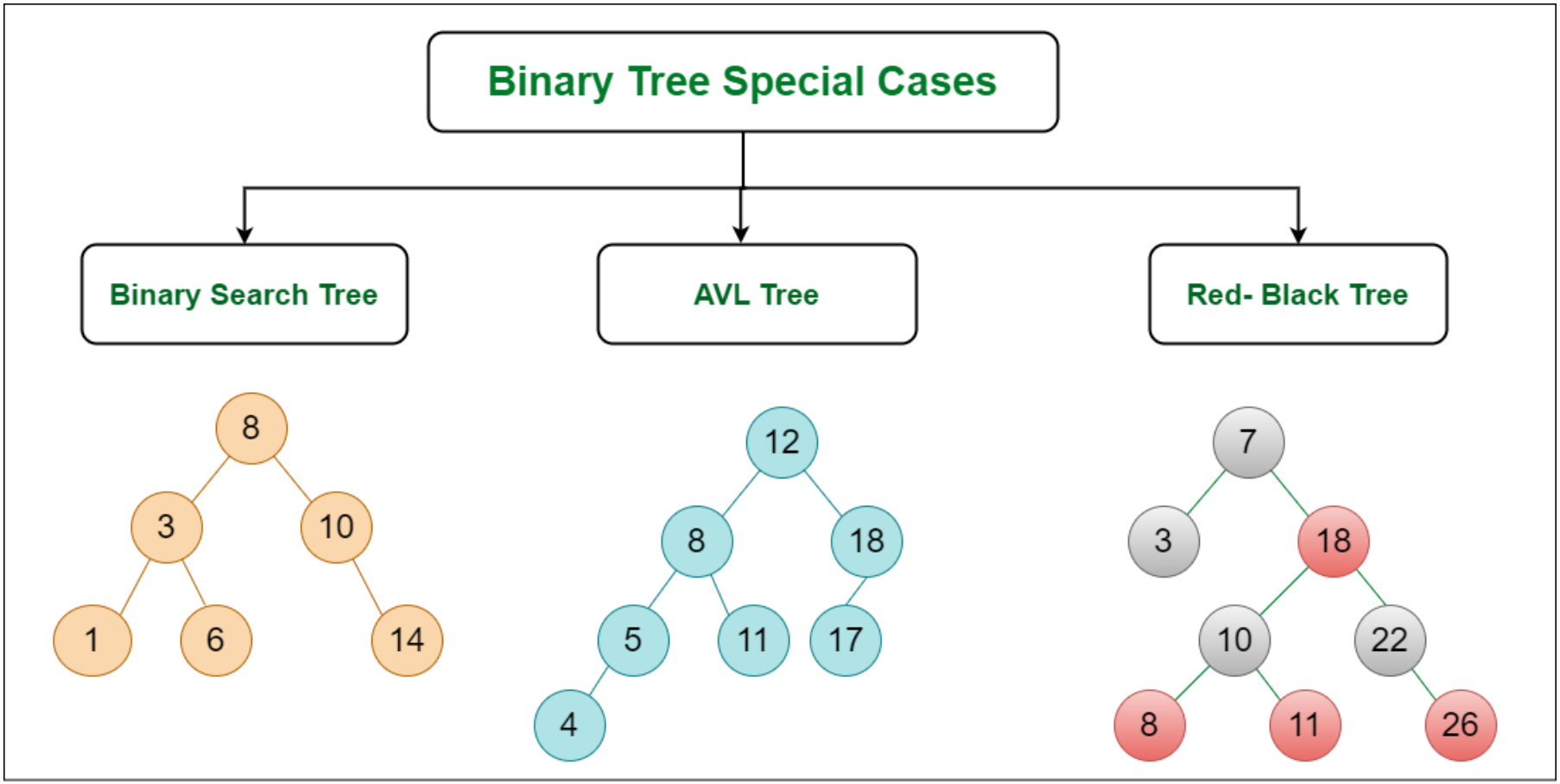


8.2. Types of Tree Data Structures

- **Binary tree:** In a binary tree, each node can have a ***maximum of two*** children linked to it. Some common types of binary trees include full binary trees, complete binary trees, balanced binary trees, and degenerate or pathological binary trees.
- **Ternary Tree:** A Ternary Tree is a tree data structure in which each node has ***at most three*** child nodes, usually distinguished as "left", "mid", and "right".
- **N-ary Tree or Generic Tree:** Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children. Unlike the linked list, each node stores the address of ***multiple nodes***.

8.2. Types of Tree Data Structures

- **On the basis of node values,** the Binary Tree can be classified into the following important special types:



8.3. Binary Search Tree

- **Binary Search Tree (BST)** is a node-based binary tree data structure which has the following properties:
  - The *left* subtree of a node contains only nodes with keys *lesser* than the node's key.
  - The *right* subtree of a node contains only nodes with keys *greater* than the node's key.
  - The left and right *subtree* each must also be a *binary search tree*.

8.3. Binary Search Tree

- Handling approach for Duplicate values in the Binary Search Tree:
  - You can not allow the duplicated values at all.
  - We must follow a consistent process throughout, i.e. either store duplicate value at the left or store the duplicate value at the right of the root, but be consistent with your approach.
  - We can keep the counter with the node and if we found the duplicate value, then we can increment the counter.

8.3. Binary Search Tree

- There are two basic ways of *binary tree traversal*:
  - *Depth-first*;
    - **Pre-order Traversal** – perform Traveling a tree in a *pre-order manner* in the data structure.
    - **In-order Traversal** – perform Traveling a tree in an *in-order* manner.
    - **Post-order Traversal** – perform Traveling a tree in a *post-order* manner.
  - *Breadth-first*;
    - **Level-order Traversal** – perform Traveling a tree in a *level-order* manner.

8.3. Binary Search Tree

- Upon *depth-first* traversal, a ***stack*** is used as an auxiliary data structure. Every *depth-first* binary tree traversal produces the same dynamic of filling the stack auxiliary structure.
  - These traversals are commonly ***recursive***.
- Upon *breadth-first* traversal, a ***queue*** is used as an auxiliary data structure. The *breadth-first* traversal means that first all descendants of a current node will be traversed, before going "deeper" into the tree.
  - These traversals are commonly ***iterative***.

8.3. Binary Search Tree

- Various operations that can be performed on a BST:
  - **Insert a Node into a BST:** A new key is always inserted at the leaf. Start searching a key from the root till a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.
    - **Time Complexity: *O(N)***, where ***N*** is the number of nodes of the BST. **Auxiliary Space: *O(1)***.
  - **Delete a Node of BST:** It is used to delete a node with specific key from the BST and return the new BST.
    - **Time Complexity: *O(N)***, where ***N*** is the number of nodes of the BST. **Auxiliary Space: *O(1)***.

8.3. Binary Search Tree

- Most of the BST operations (e.g., search, max, min, insert, delete, …, etc.) take ***O(h)*** time, where ***h*** is the height of the BST.
- The cost of these operations may become ***O(n)*** for a *skewed Binary tree*.
- If we make sure that the height of the tree remains ***O(log(n))*** after every insertion and deletion, then we can guarantee an upper bound of ***O(log(n))*** for all these operations.

8.3. Binary Search Tree

- **Inorder traversal**: In case of binary search trees (BST), Inorder traversal gives nodes in ***non-decreasing order***. We visit the left child first, then the root, and then the right child.
  - **Time Complexity:** O(N), where N is the number of nodes of the BST.
  - **Auxiliary Space:** O(1)
- Output of the example:

  20 30 40 50 60 70 80

```
/* Let us create following BST
          50
       /     \
     30       70
    /  \     /  \
  20   40   60   80
*/
```

8.3. Binary Search Tree

- **Preorder traversal**: Preorder traversal first visits the root node and then traverses the left and the right subtree. It is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree.
  - **Time Complexity:** O(N), where N is the number of nodes of the BST.
  - **Auxiliary Space:** O(1)
- Output of the example:

  50 30 20 40 70 60 80

```
/* Let us create following BST
          50
       /     \
     30       70
    /  \     /  \
  20   40   60   80
*/
```

8.3. Binary Search Tree

- **Postorder traversal:** Postorder traversal first traverses the left and the right subtree and then visits the root node. It is used to delete the tree. In simple words, visit the root of every subtree last.
  - **Time Complexity:** O(N), where N is the number of nodes of the BST.
  - **Auxiliary Space:** O(1)
- Output of the example:

  20 40 30 60 80 70 50

```
/* Let us create following BST
          50
       /     \
     30       70
    /  \     /  \
  20   40   60   80
*/
```

8.3. Binary Search Tree

- **Level order traversal:** Level order traversal of a BST is *breadth first traversal* for the tree. It visits all nodes at a particular level first before moving to the next level.
  - **Time Complexity:** O(N), where N is the number of nodes of the BST.
  - **Auxiliary Space:** O(1)

- Output of the example:

  50

  30 70

  20 40 60 80

```
/* Let us create following BST
          50
       /     \
     30       70
    /  \     /  \
  20   40   60   80
*/
```

8.4. AVL Tree

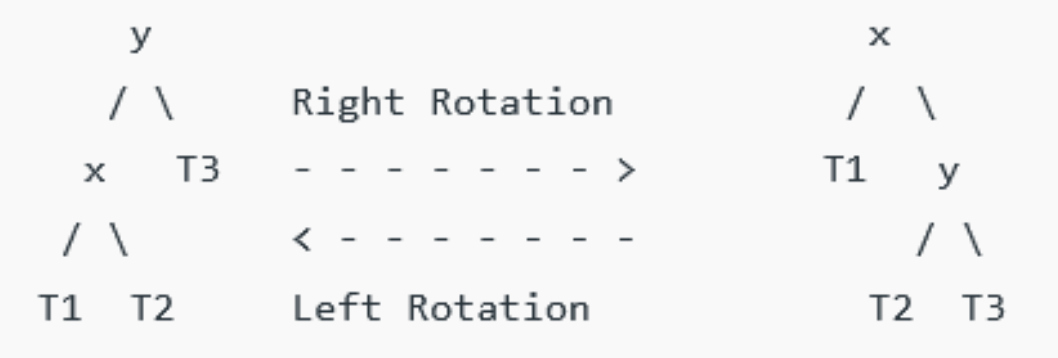
- An **AVL tree** is defined as a self-balancing **Binary Search Tree (BST)** where the difference between heights of left and right subtrees for *any* node ***cannot be more than one***.
  - The difference between the heights of the left subtree and the right subtree for any node is known as the **balance factor** of the node.
  - The AVL tree is named after its inventors, Georgy Adelson-Velsky and Evgenii Landis, who published it in their 1962 paper “An algorithm for the organization of information”.

8.4. AVL Tree

- The height of an AVL tree is always ***O(log(n))***, where ***n*** is the number of nodes in the tree.
  - To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing.
  - Following are two basic operations that can be performed to balance a BST without violating the BST property: (keys(left) < key(root) < keys(right)).
    - Left Rotation
    - Right Rotation

8.4. AVL Tree

- T1, T2 and T3 are subtrees of the tree, rooted with y (on the left side), or x (on the right side).



- Keys in both of the above trees follow the following order:

  keys(T1) < key(x) < keys(T2) < key(y) < keys(T3).

- So, BST property is not violated anywhere.

8.5. Red-Black Tree

- **Red-Black Tree** is a binary search tree in which every node is colored with either ***red*** or ***black***.
- Red Black Trees are self-balancing, meaning that the tree adjusts itself automatically after each insertion or deletion operation.
- It has a good efficient worst case running time complexity. The height of a Red-Black tree is always ***O(log n)***, where ***n*** is the number of nodes in the tree.

8.5. Red-Black Tree

- **Properties of Red-Black Tree:**
  - The Red-Black tree satisfies all the properties of BST, in addition to that it satisfies following additional properties:

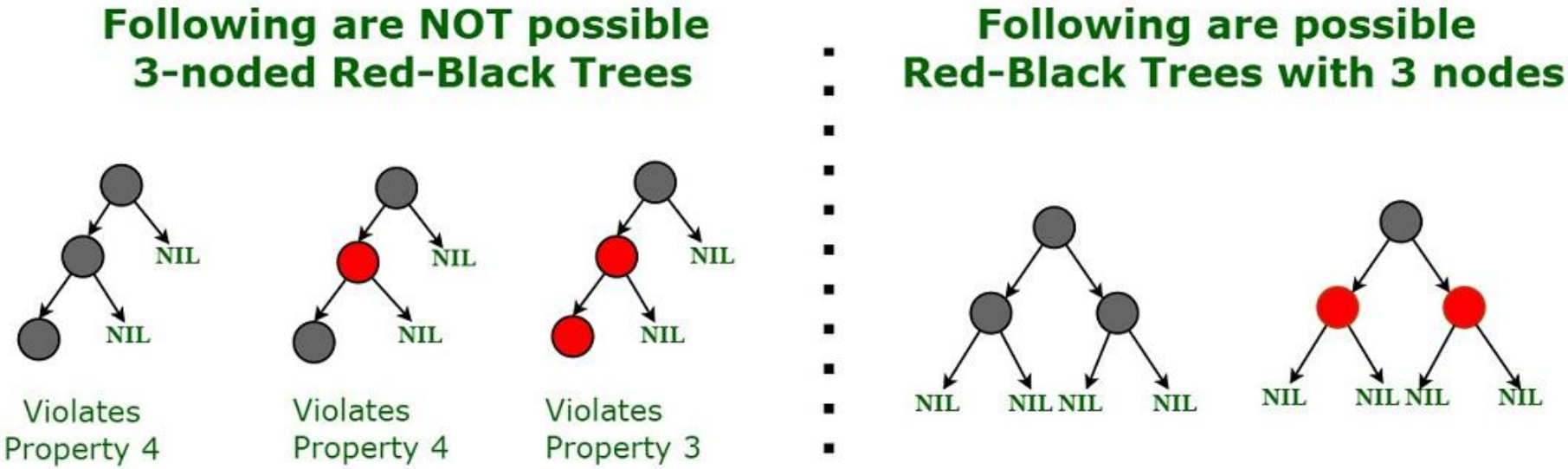
1. **Root property:** The root is black.
2. **External property:** Every leaf (Leaf is a NULL child of a node) is black in Red-Black tree.
3. **Internal property:** The children of a red node are black. Hence possible parent of red node is a black node.
4. **Depth property:** All the leaves have the same black depth.
5. **Path property:** Every simple path from root to descendant leaf node contains same number of black nodes.

8.5. Red-Black Tree

- The AVL trees are *more balanced* compared to Red-Black Trees, but they may cause *more rotations* during insertion and deletion.
- So if your application involves ***frequent insertions and deletions***, then Red-Black trees should be preferred.
- And, if the insertions and deletions are less frequent and ***search is a more frequent operation***, then AVL tree should be preferred over the Red-Black Tree.

8.5. Red-Black Tree

- A simple example to understand balancing is, that a chain of 3 nodes is not possible in the Red-Black tree. We can try any combination of colors and see if all of them violate the Red-Black tree property.



8.5. Red-Black Tree

- **Interesting points about Red-Black Tree:**

1. The ***black height*** of the red-black tree is the *number of black nodes* on a path from the root node to a leaf node. Leaf nodes are also counted as black nodes. So, a red-black tree of height *h* has *black height* >= *h*/2.
2. Height of a red-black tree with *n* nodes is:
   $h <= 2*log_2(n+1)$.
3. All leaves (NIL) are black.
4. The ***black depth*** of a node is defined as *the number of black nodes* from the root to that node i.e. the number of black ancestors.
5. Every red-black tree is a special case of a binary tree.

8.6. Applications of Trees

- **Need for Tree Data Structure**

  1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer.

  2. Trees (with some ordering, e.g. BST) provide moderate access/search (quicker than Linked List and slower than Arrays).

  3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).

  4. Like Linked Lists and unlike Arrays, Trees don't have an upper limit on the number of nodes as nodes are linked using pointers.

8.6. Applications of Trees

- **Application of Tree Data Structure:**
  - **File System:** This allows for efficient navigation and organization of files.
  - **Data Compression:** *Huffman coding* is a popular technique for data compression that involves constructing a binary tree where the leaves represent characters and their frequency of occurrence. The resulting tree is used to encode the data in a way that minimizes the amount of storage required.
  - **Compiler Design:** In compiler design, a syntax tree is used to represent the structure of a program.
  - **Database Indexing:** The tree data structures are used in database indexing to efficiently search for and retrieve data.

8.6. Applications of Trees

- **Advantages of Tree Data Structure:**
  - Trees offer **Efficient Searching** depending on the type of tree, with average search times of **O(log(n))** for balanced trees, like AVL.
  - Trees provide a hierarchical representation of data, making it **easy to organize and navigate** large amounts of information.
  - The recursive nature of trees makes them **easy to traverse and manipulate** using recursive algorithms.

8.6. Applications of Trees

- **Disadvantages of Tree Data Structure:**
  - Unbalanced Trees, meaning that the height of the tree is skewed towards one side, which can lead to **inefficient search times.**
  - Trees demand **more memory space requirements** than some other data structures, like arrays and linked lists, especially if the tree is very large.
  - The implementation and manipulation of trees **can be complex** and require a good understanding of the algorithms.